Progetto sulla Gestione dell’Università e relativa Programmazione Esami

Politecnico di Torino

**Algoritmi e Calcolatori**

Andrea Chiapparo

Luca Montangero

Riccardo Maria Galiano

*Indice*

* **Introduzione …………………………………………………………………………...**
* **Gestione Università …………………………………………………………………...**

* **Descrizione Dettagliata Progetto – Parte “Gestione Università” …………………..**
* **Descrizione Dettagliata Progetto – Parte “Gestione Esami” ……………………….**
* **Descrizione Dettagliata Progetto – Main …………………………………………….**
* **…**

ALGORITMI E CALCOLATORI A.A. 2020/2021

**GESTIONE UNIVERSITÀ E PROGRAMMAZIONE ESAMI**

**INTRODUZIONE**

Il seguente progetto si pone l'obiettivo di occuparsi della gestione delle risorse e dei dati inerenti l'università con annessa allocazione delle giornate di esame. Infatti la problematica principale nella direzione di una università è proprio quella che riguarda la gestione dell’allocazione delle giornate di esame. Tale gestione richiede che, a fronte delle regole di ateneo, si organizzino le date degli esami in modo da non violare alcun regolamento interno all'università stessa.

Tale progetto si basa a tal proposito sulla creazione e definizione di una base dati iniziale che, mediante dei comandi da terminale, può essere modificata mediante l’inserimento di nuovi dati o l’aggiornamento di quelli già esistenti.

Spiegazione del fatto che si hanno dei file di input come database, delle mappe in memoria e dei file di output.

**GESTIONE UNIVERSITA’**

Al fine di avere una percezione chiara ed esaustiva di codesto progetto si predilige abbracciare una filosofia che, come riportato nelle righe avvenire, segue un approccio top-down nella spiegazione delle classi e dei relativi metodi che ne fanno parte.

Il seguente programma è sorretto da una classe principale, detta “University”, che orchestra l’intero progetto e all’interno della quale sono presenti le strutture dati dei cinque principali attori nonché:

* gli studenti e i professori che a loro volta sono due classi (rispettivamente “Student” e “Professor”) ereditate da una ulteriore classe “madre” detta “UniversityMember” dal momento che le due condividono delle info comuni;
* le aule alle quali viene dedicata una classe detta “Classroom”;
* i corsi di studio rappresentati da una classe detta “StudyCourse”;
* i corsi che sono una classe detta “Course” che fanno parte dei Cds; Considerata la suddivisione dei corsi per anno accademico abbiamo deciso di utilizzare un’altra classe “SpecificYearCourse” nella quale specificare le varie informazioni relative all’anno accademico di un corso;

Nella gestione di una università un altro aspetto molto importante e concreto oltre alla base dati risulta essere la definizione degli esami. Infatti al fine di pianificare le date di esame e quindi i tre periodi di sessione (invernale, estiva ed autunnale) il programma si avvale dell’uso di tre ulteriori classi:

* “SessionYear” che tiene conto del calendario accademico e definisce tramite metodi appositi le date degli appelli per gli esami;
* “ExamDay” che organizza gli esami fissati in un determinato giorno nei diversi slot orari;
* “Exam” nonché una classe che si occupa della gestione delle info relative ad un determinato esame(settato per ogni oggetto SpecifiYearCourse);

Le sopra citate sono le classi principali alle quali si accostano però anche delle classi secondarie che conseguono al medesimo obiettivo (nonché il funzionamento del progetto) come:

* classe “Parse”, contiene le principali funzioni che vengono richiamate in altre classi più volte; pertanto al fine di evitare di riprodurre l’intero codice di una data funzione ogni volta si ha che al momento del bisogno ci si limita solamente a richiamarla, in quanto la sua definizione è presente dentro “Parse”;
* classe “Date” che semplifica la gestione delle date che pertanto vengono viste come degli oggetti, nonché istanziazione di tale classe; in particolare permette di gestire le date rispettando il calendario (come ad esempio il numero di giorni in uno specifico mese);
* classe “invalidDbException” che viene usata quando in fase di lettura di un database si individua una certa incongruenza provvedendo quindi al lancio di una eccezione coerente con il problema verificatosi; pertanto essa si occupa di eccezioni relative ad incongruenze che vengono individuate a posteriori;
* classe “DbException” che viene usata nel momento in cui si stanno scrivendo delle info all’interno di un database portando al sollevamento di una eccezione nel caso in cui la scrittura non rispetta alcuni vincoli prestabiliti come ad esempio il formato di un file; essa si occupa di eccezioni relative ad incongruenze che vengono individuate non più a posteriori bensì all’atto della scrittura;
* la classe “SessionLog” la quale si occupa della generazione di un file di warning nel momento in cui non si riesce a generare le date della sessione di esame portando al conseguente rilassamento di un vincolo;
* la classe SystemLog che si occupa di comunicare eventuali errori all’utente alal fine di ogni comando

**DESCRIZIONE DETTAGLIATA PROGETTO – PARTE “GESTIONE UNIVERSITA’”**

I metodi della classe “University” vengono riportati di seguito:

* **readStudents**

Prototipo: void University::readStudents()

Questo metodo si occupa della lettura del database degli studenti denominato: db\_studenti.txt .

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “InteroStudente” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”. Successivamente si usa uno dei tanti metodi statici della classe “Parse”, in questo caso quello denominato: “getMatr”; In esso un oggetto stringstream (ovvero un oggetto stream che consente di effettuare delle operazioni sulle stringhe) permette di salvare solo la matricola dello studente, precedentemente contenuta insieme al carattere “s” nel primo elemento di “InteroStudente”.

Infine si controlla se la matricola associata allo studente corrente è già esistente in memoria; in tal caso viene lanciata una eccezione di avviso, diversamente si procede con il salvataggio nella mappa in memoria dello studente e delle relative informazioni contenute nei diversi campi del vector “InteroStudente”:

* “InteroStudente[0]” contiene s-matricola;
* “InteroStudente[1]” contiene uno o più nomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroStudente[2]” contiene uno o più cognomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroStudente[3]” contiene l’indirizzo e-mail dello studente;

Vista la presenza della “modalità versioning” si lascia anche la possibilità di caricare le diverse informazioni aggiuntive riguardanti gli studenti; siccome tale modalità si verifica nel momento in cui “\_version” = 2 si distingue mediante un if il caricamento delle info base dal caricamento dettato dalla modalità versioning.

Si noti che anche nel caso delle informazioni aggiuntive dettate dalla modalità versioning si fa sempre riferimento alla classe madre “UniversityMember” per settare variabili private utilizzabili sia per oggetti Professor che Student.

Pertanto alle info precedenti sugli studenti si aggiungono le tre seguenti:

* “InteroStudente [4]” contiene la data di nascita;
* “InteroStudente [5]” contiene l’anno di immatricolazione;
* “InteroStudente [6]” contiene l’indirizzo di casa;

La chiave della mappa degli studenti è quindi la matricola dello studente (privata del carattere “s”) in quanto la matricola è univoca per ogni studente.

Il metodo si conclude con la chiusura del file database degli studenti “db\_studenti.txt”.

* **readVersion**

Prototipo: void University::readVersion()

Questo metodo si occupa della lettura del file “versione.txt” contenente il numero corrispondente alla versione impostata per la “modalità versioning”.

Si ricorda che si scrive su un apposito file, “versione.txt”, la versione adottata affinché qualora il programma dovesse terminare si tiene traccia del contenuto che possedeva la variabile “\_version”; pertanto risulta fondamentale andare a leggere la versione che il programma stava utilizzando prima dello spegnimento, da qui l’importanza del metodo readVersion().

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso a causa della sua inesistenza, si procede con la lettura del numero corrispondente alla modalità versioning contenuta nell’unica linea presente nel file; il risultato viene salvato nella stringa “line” e poi convertito in un intero e destinato alla variabile “\_version”.

Come di buona norma il metodo si conclude con la chiusura del file mediante l’apposito metodo close.

* **readProfessor**

Prototipo: void University::readProfessor()

Questo metodo si occupa della lettura del database dei professori denominato: db\_professori.txt .

Tale metodo si sviluppa in modo analogo al precedente degli studenti.

Si riportano di seguito le informazioni contenute nei diversi campi del vector “InteroProfessore”:

* “InteroProfessore [0]” contiene d-matricola (anche in questo caso il carattere “d”;
* “InteroProfessore [1]” contiene uno o più nomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroProfessore [2]” contiene uno o più cognomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “InteroProfessore [3]” contiene l’indirizzo e-mail del professore;

Vista la presenza della “modalità versioning” si lascia anche la possibilità di caricare le diverse informazioni aggiuntive riguardanti i professori; siccome tale modalità si verifica nel momento in cui \_version = 2 si distingue mediante un if il caricamento delle info base dal caricamento dettato dalla modalità versioning. Pertanto alle info precedenti sugli studenti si aggiungono le tre seguenti:

* “InteroProfessore [4]” contiene la data di nascita;
* “InteroProfessore [5]” contiene l’anno di immissione di ruolo;
* “InteroProfessore [6]” contiene l’indirizzo di casa;

La chiave della mappa dei professori è quindi la matricola del professore (privata del carattere “d”) in quanto la matricola è univoca per ogni professore.

Il metodo si conclude con la chiusura del file database dei professori “db\_ professori.txt”.

* **readClassroom**

Prototipo: void University::readClassroom()

Questo metodo si occupa della lettura del database delle aule denominato: db\_aule.txt .

Tale metodo si sviluppa in modo analogo al precedente degli studenti.

Si riportano di seguito le informazioni contenute nei diversi campi del vector “InteraClasse”:

* “InteraClasse [0]” contiene A-codice identifico dell’aula;
* “InteraClasse [1]” contiene la stringa che specifica il tipo di aula: “A” in caso di aula, “L” in caso di laboratorio;
* “InteraClasse [2]” contiene il nome dell’aula;
* “InteraClasse [3]” contiene la capienza massima dell’aula;
* “InteraClasse [4]” contiene la capienza massima dell’aula se adibita per un esame;

Anche in questo caso, considerata la presenza della “modalità versioning”, si lascia anche la possibilità di caricare le diverse informazioni aggiuntive riguardanti le aule; siccome tale modalità si verifica nel momento in cui \_version = 3 si distingue mediante un if il caricamento delle info base dal caricamento dettato dalla modalità versioning. Pertanto, alle info precedenti sulle aule, si aggiungono per esempio le quattro seguenti strumentazioni richieste in aula per sostenere un esame:

* “InteraClasse [4]”: tavole da disegno;
* “InteraClasse [5]”: computer;
* “InteraClasse [6]” : proiettore;
* “InteraClasse [7]” : lavagne;

La chiave della mappa delle aule è quindi la matricola delle aule (privata del carattere “A”) in quanto la matricola è univoca per ogni aula.

Il metodo si conclude con la chiusura del file database dei professori “db\_ aule.txt”.

* **readStudyCourse**

Prototipo: void University::readStudyCourse()

Questo metodo si occupa della lettura del database dei corsi di studio denominato: db\_corsi\_studio.txt .

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “InteroCorsoDiStudi” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”. Successivamente mediante un oggetto stringstream viene salvato solo il codice identificativo del corso di studio, precedentemente contenuto insieme al carattere “C” nel primo elemento di “InteroCorsoDiStudi”.

In particolare vengono riportate di seguito le informazioni contenute nel vector “InteroCorsoDiStudi”:

* “InteroCorsoDiStudi [0]” contiene C-codice identificativo del corso di studio;
* “InteroCorsoDiStudi [1]” contiene le due stringhe che specificano il tipo di corso di studio: BS in caso di laurea triennale, MS in caso di laurea magistrale;
* “InteroCorsoDiStudi [2]” contiene l’elenco dei codici identificativi dei diversi corsi raggruppati per semestre all’interno delle parentesi graffe;
* “InteroCorsoDiStudi [3]” contiene l’elenco dei codici identificativi dei corsi spenti separati da una virgola;

Visto il contenuto dell’elemento “InteroCorsoDiStudi [2]” nasce l’esigenza di separare i diversi corsi per ogni semestre. Si usa la funzione posCurlyBrackets la quale, passandole “InteroCorsoDiStudi [2]”, restituisce un vector di interi denominato “posSem” che contiene le diverse posizioni che occupano le parentesi graffe (aperte e chiuse) all’interno di “InteroCorsoDiStudi [2]”, posizione intesa come numero di caratteri che, a partire dal primo presente in “InteroCorsoDiStudi [2]”, si trova la parentesi graffa aperta o chiusa.

A questo punto, grazie ad un ciclo for, per ogni parentesi graffa aperta, che vista la formattazione del file “db\_corsi\_studio.txt“ si trovano nelle posizioni pari del vector “posSem” (motivo per cui l’indice del for viene incrementato di due ad ogni ciclo), si eseguono le seguenti operazioni :

1. si prende la posizione della parentesi graffa aperta, la si salva in una variabile di appoggio “posStart”;
2. si calcola la distanza “len” tra la posizione della “{“ e la posizione della successiva “}”; distanza intesa sempre in termini di caratteri presenti tra le due parentesi;
3. tramite il metodo delle stringhe “substr” si copiano in una stringa tutti i caratteri presenti nella stringa “InteroCorsoDiStudi [2]” che vanno dalla posizione presente in “posStart” (ovvero quella della parentesi graffa aperta) per un numero di caratteri pari al valore contenuto nella variabile “len”; il metodo “substr” restituisce così una stringa contenente i codici di tutti i corsi di un solo semestre; tale stringa viene poi copiata in coda, per ogni ciclo for, nel vector “semestri”.

Alla fine di questo for il vector “semestri” conterrà i corsi di ogni semestre disposti al suo interno in modo che ogni campo di “semestri” contiene tutti i corsi di un certo semestre nel seguente modo:

* “semestri [0]” contiene tutti i corsi del 1o anno 1o semestre;
* “semestri [1]” contiene tutti i corsi del 1o anno 2o semestre;
* “semestri [2]” contiene tutti i corsi del 2o anno 1o semestre;
* “semestri [3]” contiene tutti i corsi del 2o anno 2o semestre;
* “semestri [4]” contiene tutti i corsi del 3o anno 1o semestre;
* “semestri [5]” contiene tutti i corsi del 3o anno 2o semestre;

Si fa notare che il vector ”semestri” non è detto a priori che sia composto da sei campi in quanto questo dipende dal tipo di percorso di studio (triennale o magistrale) e anche dal numero di corsi presenti nel file “db\_corsi\_studio.txt”; quello appena riportato pertanto era solo un esempio generale con fine esplicativo.

Nel caso in cui la linea letta da file, “line”, contenga anche i corsi spenti, ovvero nel caso in cui il vector “InteroCorsoDiStudi” sia composto da un numero di campi pari a quattro (motivo per cui è presente un if con controllo sulla dimensione del vector pari a 4) allora si copiano in una stringa di appoggio mediante il metodo delle stringhe ”substr” tutti i caratteri presenti nel quarto campo di “InteroCorsoDiStudi” che vanno dal secondo carattere (in quanto il primo, vista la formattazione del file, è una parentesi quadra aperta) e fino al penultimo (in quanto l’ultimo, vista la formattazione del file, è una parentesi quadra chiusa); a questo punto l’elenco dei corsi spenti privato delle parentesi quadre (aperta e chiusa) viene passato alla funzione “splittedLine” insieme al carattere separatore, che in questo caso è la virgola “,”, la quale pertanto restituisce pertanto tutti i corsi spenti posizionati ognuno in un campo diverso del vector di stringhe “corsiSpenti”.

Dopo aver letto dalla riga tutte le informazioni inerenti un determinato corso di studio e sistemate in delle apposite variabili si passa alla creazione di un oggetto di appoggio denominato “SCourse” passando al relativo costruttore il codice identificativo del corso di studio e l’informazione inerente il tipo di corso di studio, ovvero percorso triennale o magistrale.

Con l’ausilio del metodo “addSemesterCourses”, appositamente creato per gli oggetti della classe “StudyCourse” (come dimostra il file “StudyCourse.h”), si riempie la mappa privata interna dei semestri dell’oggetto di appoggio poc’anzi creato, “SCourse”, la quale contiene i diversi corsi per ogni di semestre di ogni anno; pertanto mediante un ciclo for, per ogni stringa del vector “semestri” si procede come spiegato di seguito:

1. Si calcola l’anno associato al corso/i presente nella i-esima posizione in “semestri”; siccome il vector “semestri” contiene i corsi di ogni semestre disposti al suo interno in ordine crescente per semestre ovvero partendo dal 1o semestre del 1o anno fino al 2o semestre del 3o (o fino al 2o semestre del 2o anno in caso di laurea magistrale) è facile dedurre come l’anno sia pari al numero del semestre (incrementato di 1 perché la numerazione degli indici in un vector parte da 0) diviso 2 (in quanto ogni anno contiene due semestri);
2. Analogamente si calcola il semestre (se 1o o 2o) associato all’i-esimo corso/i che pertanto risulta essere uguale al resto dell’operazione che ha portato al calcolo dell’anno;
3. Si riempie la mappa privata dell’oggetto “SCourse” grazie al metodo “addSemesterCourses” al quale viene passato: l’anno associato all’i-esimo corso/i, il semestre (1o o 2o) associato all’i-esimo corso/i ed infine l’i-esimo corso/i;

Successivamente in caso di presenza di corsi spenti attraverso il metodo “addOffCourses”, appositamente creato per gli oggetti della classe “StudyCourse” (come dimostra il file “StudyCourse.h”), si riempie la lista di stringhe privata interna dei semestri spenti dell’oggetto di appoggio “SCourse”.

Come ultima operazione si riempie la mappa interna privata denominata “\_studyCourse” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso e come value corrispondente l’oggetto di appoggio “SCourse”.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Tale metodo si conclude con la chiusura del file database dei corsi di studio “db\_corsi\_studio.txt”.

* **readCourse**

Prototipo: void University::readCourse()

Questo metodo si occupa della lettura del database dei corsi denominato: db\_corsi.txt .

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “interoCorso” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”.

Il file “db\_corsi.txt” presenta una struttura secondo la quale:

* ogni riga che comincia con il carattere “c” contiene le info generali di quel corso;
* ogni riga che comincia con il carattere “a” contiene le info di quel corso specifiche per un certo anno accademico;

Pertanto si controlla il primo elemento del vector “interoCorso” (ovvero “interoCorso [0]”) e nel caso in cui fosse uguale alla stringa “c” si passa all’elaborazione delle info generali in particolare a seguire vengono riportate le info presenti nei diversi campi del vector “interoCorso”:

* “interoCorso [1]” contiene il codice identificativo del corso salvato poi in una variabile di appoggio denominata “lastReadCourse”;
* “interoCorso [2]” contiene il titolo del corso;
* “interoCorso [3]” contiene il numero di cfu del corso;
* “interoCorso [4]” contiene il numero di ore di lezione del corso;
* “interoCorso [5]” contiene il numero di ore di esercitazione del corso;
* “interoCorso [6]” contiene il numero di ore di laboratorio del corso;

Si riempie in seguito la mappa interna privata denominata “\_courses” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso (contenuto nella variabile “lastReadCourse” e come value corrispondente l’oggetto della classe “Course”, inizializzato mediante costruttore al quale vengono passati (come da definizione presente nel file “Course.h”) i seguenti dati:

* codice identificativo del corso contenuto in “interoCorso [1]”;
* titolo del corso contenuto in “interoCorso [2]”;
* numero di cfu del corso contenuto in “interoCorso [3]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
* numero delle ore di lezione del corso contenuto in “interoCorso [4]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
* numero delle ore di esercitazione del corso contenuto in “interoCorso [5]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
* numero di ore di laboratorio del corso contenuto in “interoCorso [6]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);

Nel caso in cui il primo elemento del vector “interoCorso” (ovvero “interoCorso [0]”) contenga la stringa “a” allora si passa all’elaborazione dei dati inerenti uno specifico anno accademico.

Per migliorare la leggibilità del codice si copia il vector “interoCorso” in un altro chiamato “SpecificYearCourse”. Pertanto tale vector presenta per quello specifico anno accademico le seguenti informazioni:

* “SpecificYearCourse [1]” contiene l’anno accademico del corso (anno inizio – anno fine);
* “SpecificYearCourse [2]” contiene l’info se tale corso è attivo/non attivo;
* “SpecificYearCourse [3]” contiene il numero di versioni in parallelo;
* “SpecificYearCourse [4]” contiene l’elenco degli id dei professori che insegnano quel corso quel particolare anno, compreso il professore titolare, e relative ore di lezione, laboratorio ed esercitazione proprio come riportato nel file;
* “SpecificYearCourse [5]” contiene le info inerenti l’esame di quel corso proprio come sono scritte nel file;
* “SpecificYearCourse [6]” contiene l’elenco dei codici dei corsi raggruppati ovvero dei corsi che si trovano nello stesso anno accademico insieme a quello in esame;

Si salva in una variabile di appoggio denominata “acYear” l’anno accademico contenuto ora in “SpecificYearCourse [1]”; si controlla se quel determinato corso in quell’anno fosse attivo o meno andando a confrontare il contenuto del campo “SpecificYearCourse [2]” con la stringa “attivo”; nel caso in cui lo è viene settata come true la variabile di appoggio “isActive” che tiene appunto conto dello stato del corso; si procede analogamente per il numero di versioni in parallelo contenute nella stringa “SpecificYearCourse [3]” e poi salvate in una variabile intera mediante il metodo di conversione delle stringhe stoi.

Visto il contenuto di “SpecificYearCourse [4]” il quale contiene i diversi dati proprio come scritti nel file (ovvero con caratteri delimitatori come parentesi graffe, quadre e virgole) nasce l’esigenza di togliere le parentesi quadre motivo per cui tale vector viene passato al metodo delle stringhe substr che copia nella stringa “profSenzaQuadre” tutti i caratteri presenti in “SpecificYearCourse [4]” dal secondo al penultimo andando ad escludere così il primo e l’ultimo, rispettivamente “[“ e “]”.

Successivamente si passa il vector “profSenzaQuadre” alla funzione getProfPar che restituisce il vector di stringhe “profCorsoPar” composta da un numero di campi pari al numero di versioni in parallelo dove ogni campo contiene quindi tutti i professori, compreso il titolare, che insegnano quel determinato corso parallelo.

In seguito vengono processate le informazioni riguardanti gli esami contenute in “SpecificYearCourse [5]” copiandole, private delle parentesi graffe aperte e chiuse (mediante il metodo substr) in un vector di stringhe di appoggio detto “examData”. Passando quest’ultimo alla funzione splittedLine insieme al carattere separatore “,” si ottiene il vector “splittedExamData” composto dalle seguenti voci:

* “splittedExamData [0]” contiene la durata dell’esame in minuti;
* “splittedExamData [1]” contiene i minuti necessari per accedere all’esame;
* “splittedExamData [2]” contiene i minuti necessari per abbandonare il luogo dell’esame una volta terminato;
* “splittedExamData [3]” contiene la tipologia di esame “S”/”O” (ovvero scritto/orale);
* “splittedExamData [4]” contiene la tipologia di locazione in cui verrà sostenuto l’esame ”A”/”L” (ovvero aula/laboratorio);

In seguito si prende la stringa contente l’elenco dei codici dei corsi raggruppati contenuti in “SpecificYearCourse [6]” privandoli delle parentesi graffe aperte e chiuse sempre mediante il metodo substr, copiando della stringa tutti i caratteri esclusi il primo (ovvero “{“ ) e l’ultimo (ovvero "}”). La stringa risultante viene passata alla funzione “splittedLine” insieme al carattere separatore che in questo caso è “,”; si ottiene quindi un vector di stringhe “idGrouped”.

Risulta necessario inoltre sapere in quali Corsi di Studio appartiene tale corso (appena letto da file) allora mediante un for si scandagliano tutti i corsi di studio in memoria; sole se tale corso appena letto risulta essere attivo si recupera, l’anno e il semestre in cui il corso “lastReadCourse” è insegnato nel corso di studio puntato dal for; tale anno e semestre vengono salvati in una stringa detta “result” nel formato yy\_semestre.

Solo nel momento in cui tale corso appartiene al corso di studio, ovvero solo nel momento in cui la stringa “result” non è vuota, si procede con il copiare il contenuto di “result” in yy\_semester”. Infine si salva in un vector di interi detto “studyCourse” gli id dei corsi di studio in cui il corso “lastReadCourse” appartiene; in particolare ci si avvale del metodo **assignStudyCourse** il quale controlla se il corso passato come parametro appartiene al corso di studio per il quale esso stesso viene chiamato; restituisce un bool pari a true nel caso in cui appartiene diversamente resituisce un booleano pari a false.

Solo al termine del for si può procedere con il caricare in memoria le informazioni sino ad ora processate per quel determinato corso in riferimento all’anno accademico contenuto in “acYear”; in particolare si usa il metodo della classe Course, **addSpecificYearCourse**, che aggiunge nella mappa privata “\_courseOfTheYear” (della classe “Course”) in corrispondenza dell’anno di inizio dell’anno accademico, “acYear”, le informazioni lette (infatti si ricorda come la mappa privata “courseOfTheYear” abbia come key l’anno di inizio dell’anno accademico e come value un oggetto della classe SpecificYearcourse).

Prima di concludere il ciclo di while si provvede ad incrementare di una unità il contatore “line\_counter”.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo, tenendo conto che possono esserci per uno stesso anche due o più righe che iniziano con la stringa “c” ovvero più anni accademici inerenti quello specifico corso.

Tale metodo si conclude con la chiusura del file database dei corsi di studio “db\_corsi.txt”.

* **addStuds**

Prototipo: void University::addStuds(const std::string &fileIn)

Metodo lanciato in risposta al comando: -a:s <student\_list\_file>.

Questo metodo si occupa dell’inserimento di nuovi studente (quindi privi di matricola) nella mappa in memoria degli studenti. Solo al termine di diversi controlli come ad esempio la conferma della non presenza di campi vuoti e dimostrata la validità di questi si procede con l’inserimento del singolo studente nella map apposita per gli studenti e solo se tutti gli studenti letti da file non presentano errori che invaliderebbero il database si procede con la scrittura di db\_studenti.txt.

Tali studenti sono riportati nel file il cui nome viene passato nel parametro “fileIn” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, così come nei metodi precedenti viene splittata tramite la “splittedline” prima citata salvando i vari campi di informazione in un vector di stringhe detto “infoStud”.

Si procede con l’ottenere il codice univoco identificativo per il nuovo studente da inserire nella variabile intera “matr” mediante l’ausilio dell’apposito metodo “getNewStudentId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Vista la presenza della “modalità versioning” si controlla il valore assunto dalla variabile privata “\_version” che tiene traccia appunto della versione adottata dall’utente. Si distinguono i due seguenti casi:

Se “\_version” vale 2 (ovvero se la “modalità versioning” è stata attivata) allora:

* si controlla che la dimensione del vector “infoStud” sia pari a sei campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); qualora non fossero pari a sei si usa una stringa detta “error” nella quale si tiene traccia dell’errore di formato verificatosi nel file, specificando anche la linea in corrispondenza della quale è stato rilevato; quest’ultima peculiarità è possibile attuarla grazie all’uso di una variabile contatore che tiene conto del numero di linee lette; dichiarata sotto il nome di “line\_counter”, tale variabile intera infatti viene incrementata al termine di ogni ciclo del while.

Infine si procede con il settare al valore false una variabile boolenana detta “doDbWrite” la cui importanza è possibile coglierla nella parte finale di tale metodo;

* si controlla se il vector “infoStud” contenga almeno un campo vuoto mediante il metodo “controlFieldsVectorAreEmpty”; quest’ultimo infatti restituisce true nel momento in cui è presente anche solo un campo vuoto nel vector “infoStud” passatogli come parametro. In tal caso si procede col “prendere nota” di quanto verificatosi sempre all’interno della stringa “error”; in particolare si salva al suo interno un messaggio, sottoforma di stringa, col quale si informa dell’assenza di un dato riguardante lo studente specificando anche in questo caso la linea del file in corrsipondenza della quale si è verficato ciò. Infine si procede, anche in questo caso, con il settare al valore false la variabile “doDbWrite”;
* solo nel caso in cui il vector “infoStud” disponesse di sei elementi allora si procederebbe con il salvataggio nella mappa in memoria dello studente e delle relative informazioni contenute nei diversi campi del vector “infoStud”;

Se “\_version” vale 1 (ovvero se la “modalità versioning” NON è stata attivata) allora:

si controlla che la dimensione del vector “infoStud” sia pari a tre campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); Così come per il primo caso di versioning si eseguono i dovuti controlli sulla linea letta da file per poi, in caso di esito positivo caricare le informazioni lette nell’apposita struttura dati; Unica differenza: viene utilizzato un costrutture “Student” che non considera le informazioni aggiuntive dovute alla versioning.

Si riportano di seguito le informazioni base contenute nei diversi campi del vector “infoStud” necessarie a salvare lo studente in memoria:

* “infoStud [0]” contiene uno o più nomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “infoStud [1]” contiene uno o più cognomi dello studente (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “infoStud [2]” contiene l’indirizzo e-mail dello studente;

Nel caso in cui la “modalità versioning” sia stata attivata allora alle informazioni di base precedenti dovrebbero essere aggiunte le tre caratteristiche riportate di seguito:

* “infoStud [3]” contiene la data di nascita;
* “infoStud [4]” contiene l’anno di immatricolazione;
* “infoStud [5]” contiene l’indirizzo di casa;

In ambo i casi la chiave della mappa degli studenti è quindi la matricola dello studente stesso in quanto univoca per ognuno di essi.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Come di buona norma si procede con la chiusura del file contenente gli studenti da aggiungere in anagrafica.

In seguito si controlla il valore assunto dal flag precedentemente presentato, “doDbWrite”:

* Se pari a false (ovvero in caso di errore verificatosi in fase di lettura del file “fIn”) si procede con il sollevamento di una eccezione comunicando tutti gli errori individuati e raccolti nella stringa “error”.

**Si noti inoltre come, in fase di progettazione, si è deciso, dove possibile, di effettuare in primis tutti i controlli relativi ad ogni riga del file somministrato al programma e solo in seguito, ovvero solo dopo aver raccolto tutte le eventuali irregolatità, provvedere al sollevamento di una eccezione.**

* Se è uguale a true (ovvero se NON si sono rilavati errori di alcun tipo) allora si prosegue con la riscrittura automatizzata del file database degli studenti db\_studenti.txt, dal momento che nuovi studenti sono stati aggiunti nella base dati; la riscrittura avviene attraverso il metodo dbStudsWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

**Si noti pertanto come, in fase di progettazione, si è preferito proseguire con la riscrittura del file solo nel momento in cui tutti gli studenti contenuti nel file non presentano errori.**

* **addProfessors**

Prototipo: void University::addProfessors(const std::string &fileIn)

Metodo lanciato in risposta al comando: -a:d <prof\_list\_file>.

Questo metodo si occupa dell’inserimento di nuovi professsori (quindi privi di matricola) nella mappa in memoria dei professori e, solo al termine di diversi controlli tutti andati a buon fine, anche nel database db\_professori.txt.

Questo metodo si sviluppa in modo analogo al precedente degli studenti.

Tali professori sono riportati nel file il cui nome viene passato nel parametro “fileIn” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. Così come nei metodi precedenti, ogni riga letta da file viene splittata tramite la “splittedline” prima citata salvando i vari campi di informazione in un vector di stringhe detto “infoProf”.

Si procede con l’ottenere il codice univoco identificativo per il nuovo docente da inserire nella variabile intera “matr” mediante l’ausilio dell’apposito metodo “getNewProfessorId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Vista la presenza della “modalità versioning” si controlla il valore assunto dalla variabile privata “\_version” che tiene traccia appunto della versione adottata dall’utente. Si distinguono i due seguenti casi:

Se “\_version” vale 2 (ovvero se la “modalità versioning” è stata attivata) allora:

* si controlla che la dimensione del vector “infoProf” sia pari a sei campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); qualora non fossero pari a sei si usa una stringa detta “error” nella quale si tiene traccia dell’errore di formato verificatosi nel file, specifiando anche la linea in corrsipondenza della quale è stato rilevato.

Infine si procede con il settare al valore false una variabile boolenana detta “doDbWrite” come nel caso del metodo “addStuds”;

* si controlla se il vector “infoProf” contenga almeno un campo vuoto mediante il metodo controlFieldsVectorAreEmpty; quest’ultimo infatti restituisce true nel momento in cui è presente anche solo un campo vuoto nel vector “infoProf” passatogli come parametro. In tal caso si procede col “prendere nota” di quanto verificatosi sempre all’interno della stringa “error”; in particolare si salva al suo interno un messaggio, sottoforma di stringa, col quale si informa dell’assenza di un dato riguardante il docente specificando anche in questo caso la linea del file in corrsipondenza della quale si è verficato ciò. Infine si procede, anche in questo caso, con il settare al valore false la variabile boolenana detta “doDbWrite”;
* solo nel caso in cui il vector “infoProf” dispone di sei elementi coerenti con il formato richiesto allora si procede con il salvataggio nella mappa in memoria del professore e delle relative informazioni contenute nei diversi campi del vector “infoProf”;

Se “\_version” vale 1 (ovvero se la “modalità versioning” NON è stata attivata) allora:

si controlla che la dimensione del vector “infoProf” sia pari a tre campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); qualora non fossero pari a tre si procede come nel caso precedente: l’errore verrà caricato in una stringa error e segnalato all’utente solo alla chiusura del programma. Anche in questo caso la variabile booleana “doDbWrite” ci permetterà di non scrivere errori nel database db\_professori.txt;

* si controlla se il vector “infoProf” contenga almeno un campo vuoto mediante il metodo controlFieldsVectorAreEmpty; quest’ultimo infatti restituisce true nel momento in cui è presente anche solo un campo vuoto nel vector “infoProf” passatogli come parametro. In tal caso si procede col “prendere nota” di quanto verificatosi sempre all’interno della stringa “error” e settando la solita variabile “doDbWrite” a false;
* solo nel caso in cui il vector “infoProf” dispone di tre elementi coerenti con il formato voluto allora si procede con il salvataggio nella mappa in memoria del docente e delle relative informazioni contenute nei diversi campi del vector “infoProf”;

Si riportano di seguito le informazioni base contenute nei diversi campi del vector “infoProf” necessarie a salvare il professore in memoria:

* “infoProf [0]” contiene uno o più nomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “infoProf [1]” contiene uno o più cognomi del professore (in quanto la funzione splittedLine è stata applicata rispetto al carattere “ ;”;
* “infoProf [2]” contiene l’indirizzo e-mail del professore;

Nel caso in cui la “modalità versioning” sia stata attivata allora alle informazioni di base precedenti verrebbero aggiunte le tre riportate di seguito:

* “infoProf [3]” contiene la data di nascita;
* “infoProf [4]” contiene l’anno di immisione in ruolo;
* “infoProf [5]” contiene l’indirizzo di casa;

In ambo i casi la chiave della mappa dei professori è quindi la matricola del docente stesso in quanto univoca per ognuno di essi.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Dopo la chiusura del file di aggiunta per gli studenti si procede con il controllo della variabile “doDbWrite”:

* Se pari a false (ovvero in caso di errore verificatosi in fase di lettura del file “fIn”) si procede con il sollevamento di una eccezione comunicando tutti gli errori individuati e raccolti nella stringa “error”.
* Se è uguale a true (ovvero se NON si sono rilavati errori di alcun tipo) allora si prosegue con la riscrittura automatizzata del file database degli studenti db\_professori.txt.

**Si noti pertanto come, in fase di progettazione, si è preferito proseguire con la riscrittura del file solo nel momento in cui tutti i docenti contenuti nel file non presentino errori.**

* **addClassrooms**

Prototipo: void University::addClassrooms(const std::string &fileIn)

Metodo lanciato in risposta al comando: -a:a <aule\_list\_file>.

Questo metodo si occupa dell’inserimento di nuove aule (quindi prive di codice identificativo) nella mappa in memoria delle aule e, solo al termine di diversi controlli tutti andati a buon fine, anche nel database db\_aule.txt.

Questo metodo si sviluppa in modo analogo ai precedenti per studenti e professori.

Tali aule sono riportati nel file il cui nome viene passato nel parametro “fileIn” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “infoClassroom”.

Si procede con l’ottenere il codice univoco identificativo per la nuova aula da inserire nella variabile intera “id” mediante l’ausilio dell’apposito metodo “getNewClassroomId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Vista la presenza della “modalità versioning” si controlla il valore assunto dalla variabile privata “\_version” che tiene traccia appunto della versione adottata dall’utente. Si distinguono i due seguenti casi:

Se “\_version” vale 3 (ovvero se la “modalità versioning” è stata attivata) allora:

* si controlla che la dimensione del vector “infoClassroom” sia pari a sette campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); qualora non fossero pari a sette verrà segnalato l’errore nella solita stringa error e verrà di conseguenza settata la variabile booleana “doDbWrite”;
* si controlla se il vector “infoClassroom” contenga almeno un campo vuoto mediante il metodo controlFieldsVectorAreEmpty; quest’ultimo infatti restituisce true nel momento in cui è presente anche solo un campo vuoto nel vector “infoClassroom” passatogli come parametro. In tal caso verrà segnalato l’errore e settato “doDbWrite”;
* solo nel caso in cui il vector “infoClassroom” dispone di sette elementi allora si procede con il salvataggio nella mappa in memoria dell’aula e delle relative informazioni contenute nei diversi campi del vector “infoClassroom”; il metodo insert si noti essere inserito all’interno di un try in quanto l’ operazione di caricamento in memoria prevede la chiamata del costruttore della classe Classroom il quale a sua volta controlla il contenuto del primo campo del vector e lanciando una eccezione qualora esso non sia uguale alla stringa “A” o “L” (nonché gli unici due caratteri concessi che permettono di distinguere se l’aula in questione risulti essere un’aula o un laboratorio). Nel caso in cui il carattere scritto nella riga del file sia uno tra i due permessi allora l’istruzione di inserimento va a buon fine diversamente si procede con il sollevamento di una eccezione la quale viene raccolta nel suddetto metodo, addClassrooms, mediante l’apposito catch; all’interno di quest’ultimo, in particolare, si tiene traccia in una stringa dell’errore verificatosi all’interno del file evidenziando anche la riga, mediante il contatore “line\_counter”, in cui è stato usato un carattere non valido.

Se “\_version” vale 1 (ovvero se la “modalità versioning” NON è stata attivata) allora:

* si controlla che la dimensione del vector “infoClassroom” sia pari a quattro campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); qualora non fossero pari a quattro si usa la solita stringa “error” nella quale si tiene traccia dell’errore.

Infine si procede con il settare al valore false la solita variabile boolenana “doDbWrite” la cui importanza è possibile coglierla nella parte finale di tale metodo;

* si controlla se il vector “infoClassroom” contenga almeno un campo vuoto mediante il metodo controlFieldsVectorAreEmpty; quest’ultimo infatti restituisce true nel momento in cui è presente anche solo un campo vuoto nel vector “infoClassroom” passatogli come parametro. In tal caso si procede col “prendere nota” di quanto verificatosi sempre all’interno della stringa “error”. Infine si procede, anche in questo caso, con il settare al valore false la variabile boolenana “doDbWrite”;
* solo nel caso in cui il vector “infoClassroom” dispone di quattro elementi allora si procede con il salvataggio nella mappa in memoria dell’aula e delle relative informazioni contenute nei diversi campi del vector “infoClassroom”.

Si riportano di seguito le informazioni base contenute nei diversi campi del vector “infoClassroom” necessarie a salvare l’aula in memoria:

* “infoClassroom [0]” contiene la stringa che specifica il tipo di aula: “A” in caso di aula, “L” in caso di laboratorio;
* “infoClassroom [1]” contiene il nome dell’aula;
* “infoClassroom [2]” contiene la capienza massima dell’aula;
* “infoClassroom [3]” contiene la capienza massima dell’aula se adibita per un esame;

Nel caso in cui la “modalità versioning” è stata attivata allora alle informazioni di base precedenti si aggiungono le tre riportate di seguito:

* “infoClassroom [4]” contiene la strumentazione richiesta in aula per sostenere un esame: tavole da disegno
* “infoClassroom [5]” contiene la strumentazione richiesta in aula per sostenere un esame: computer;
* “infoClassroom [6]” contiene la strumentazione richiesta in aula per sostenere un esame: proiettore;
* “infoClassroom [7]” contiene la strumentazione richiesta in aula per sostenere un esame: lavagna;

In ambo i casi la chiave della mappa delle aule è quindi la matricola dell’aula stessa in quanto univoca per ognuna di esse.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Come di buona norma si procede con la chiusura del file contenente le aule da aggiungere in anagrafica.

In seguito si controlla il valore assunto dal flag precedentemente presentato, “doDbWrite” per far si che il db sia effettivamente aggiornato o meno.

* **addStudyCourses**

Prototipo: bool University::addStudyCourses(const std::string &fin)

Metodo lanciato in risposta al comando: -a:f <corsi\_studio\_list\_file>.

Questo metodo si occupa dell’inserimento dei nuovi corsi di studio (quindi privi di codice identificativo) nella mappa in memoria dei corsi di studio. Tali corsi di studio sono riportati nel file il cui nome viene passato nel parametro “fin” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (restituendo in tal caso un booleano pari a false), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “tokens”. Si controlla che tale vector contenga due campi ovvero che siano state riportate nella riga per quel determinato corso di studio tutte e due le info necessarie per salvarlo in memoria ovvero: il tipo di corso di studio (ovvero le due stringhe che specificano il tipo di corso di studio: BS in caso di laurea triennale, MS in caso di laurea magistrale) e l’elenco dei corsi presenti in tale percorso di studio. In caso in cui il numero di campi dovesse essere diverso da due viene lanciata una eccezione in cui si specifica il numero corrispondente alla riga in cui si è verificato l’errore (tale variabile contatore che tiene conto del numero di linee lette viene incrementata ogni volta che si rientra nel ciclo del while ed è dichiarata sotto il nome di “line\_counter”).

Nel caso in cui la riga contenga tutte le informazioni necessarie si procede con l’ottenere il codice univoco identificativo per il nuovo corso di studio da inserire mediante l’ausilio dell’apposito metodo “getStudyCourseId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Ottenuta la matricola del corso di studio salvata nella variabile “codCorso”, visto il valore della stringa “tokens [1]” (contenente l’elenco dei codici identificativi dei diversi corsi raggruppati per semestre all’interno delle parentesi graffe) nasce l’esigenza di separare i diversi corsi per ogni semestre. Si usa il metodo delle stringhe find\_first\_of il quale, relativamente alla stringa “tokens [1]”, passandogli i due caratteri “{“ e “}”, restituisce un intero (salvato nella variabile di appoggio “found”) indicante la prima posizione in corrispondenza della quale si trova nella stringa uno dei due caratteri passati. Dal momento che la find\_first\_of si blocca la primo carattere trovato allora si ripete tale operazione per tutti i rimanenti caratteri di “tokens [1]” successivi a quello appena individuato (motivo per cui nella find\_first\_of è presente come parametro “found+1”) mediante un ciclo while. Le diverse posizioni trovate vengono di volta in volta per ogni ciclo while accodate in un vector di interi detto “posSem” il quale pertanto contiene le diverse posizioni che occupano le parentesi graffe (aperte e chiuse) all’interno di “tokens [1]”, posizione intesa come numero di caratteri che, a partire dal primo presente in “tokens [1]”, si trova la parentesi graffa aperta o chiusa.

A questo punto, grazie ad un ciclo for, per ogni parentesi graffa aperta, che vista la formattazione del file “db\_corsi\_studio.txt“ si trovano nelle posizioni pari del vector “posSem” (motivo per cui l’indice del for viene incrementato di due ad ogni ciclo), si eseguono le seguenti operazioni :

1. si prende la posizione della parentesi graffa aperta, la si salva in una variabile di appoggio “posStart”;
2. si calcola la distanza “len” tra la posizione della “{“ e la posizione della successiva “}”; distanza intesa sempre in termini di caratteri presenti tra le due parentesi;
3. tramite il metodo delle stringhe “substr” si copiano in una stringa tutti i caratteri presenti nella stringa “tokens [1]” che vanno dalla posizione presente in “posStart” (ovvero quella della parentesi graffa aperta) per un numero di caratteri pari al valore contenuto nella variabile “len”; il metodo “substr” restituisce così una stringa contenente i codici di tutti i corsi di un solo semestre; tale stringa viene poi copiata in coda, per ogni ciclo for, nel vector “semestri”.

Alla fine di questo for il vector “semestri” conterrà i corsi di ogni semestre disposti al suo interno in modo che ogni campo di “semestri” contiene tutti i corsi di un certo semestre nel seguente modo:

* “semestri [0]” contiene tutti i corsi del 1o anno 1o semestre (della laurea triennale o magistrale);
* “semestri [1]” contiene tutti i corsi del 1o anno 2o semestre (della laurea triennale o magistrale);
* “semestri [2]” contiene tutti i corsi del 2o anno 1o semestre (della laurea triennale o magistrale);
* “semestri [3]” contiene tutti i corsi del 2o anno 2o semestre (della laurea triennale o magistrale);
* “semestri [4]” contiene tutti i corsi del 3o anno 1o semestre (della laurea triennale);
* “semestri [5]” contiene tutti i corsi del 3o anno 2o semestre (della laurea triennale);

Si controlla a questo punto che, a seconda del tipo laurea, il numero di semestri sia coerente ovvero si controlla che il numero di semestri per una laurea triennale siano sei o che per una laurea triennale siano quattro. Nel caso in cui così non fosse si lancia una eccezione indicando anche la linea in corrispondenza della quale si è verificato l’errore.

Dopo aver letto dalla riga tutte le informazioni inerenti un determinato corso di studio e sistemate in delle apposite variabili si passa alla creazione di un oggetto di appoggio denominato “SCourse” passando al relativo costruttore il codice identificativo del corso di studio e l’informazione inerente il tipo di corso di studio, ovvero percorso triennale o magistrale.

Con l’ausilio del metodo “addSemesterCourses”, appositamente creato per gli oggetti della classe “StudyCourse” (come dimostra il file “StudyCourse.h”), si riempie la mappa privata interna dei semestri dell’oggetto di appoggio poc’anzi creato, “SCourse”, la quale contiene i diversi corsi per ogni di semestre di ogni anno; pertanto mediante un ciclo for, per ogni stringa del vector “semestri” si procede come spiegato di seguito:

1. Si calcola l’anno associato al corso/i presente nella i-esima posizione in “semestri”; siccome il vector “semestri” contiene i corsi di ogni semestre disposti al suo interno in ordine crescente per semestre ovvero partendo dal 1o semestre del 1o anno fino al 2o semestre del 3o (o fino al 2o semestre del 2o anno in caso di laurea magistrale) è facile dedurre come l’anno sia pari al numero del semestre (incrementato di 1 perché la numerazione degli indici in un vector parte da 0) diviso 2 (in quanto ogni anno contiene due semestri);
2. Analogamente si calcola il semestre (se 1o o 2o) associato all’i-esimo corso/i che pertanto risulta essere uguale al resto dell’operazione che ha portato al calcolo dell’anno;
3. Si riempie la mappa privata dell’oggetto “SCourse” grazie al metodo “addSemesterCourses” al quale viene passato: l’anno associato all’i-esimo corso/i, il semestre (1o o 2o) associato all’i-esimo corso/i ed infine l’i-esimo corso/i;

Come ultima operazione si riempie la mappa interna privata denominata “\_studyCourse” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso e come value corrispondente l’oggetto di appoggio “SCourse”.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

La chiave della mappa dei corsi di studio è quindi il codice identificativo in quanto univoco per ogni corso di studio.

Dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database dei corsi di studio db\_corsi\_studio.txt, dal momento che nuovi corsi di studio sono stati aggiunti nella base dati, il tutto attraverso il metodo dbStudyCourseWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database.

* **addCourses**

Prototipo: bool University::addCourses(const std::string &fin)

Metodo lanciato in risposta al comando: -a:c <corsi\_list\_file>.

Questo metodo si occupa dell’inserimento dei nuovi corsi (quindi privi di codice identificativo) nella mappa in memoria dei corsi. Tali corsi sono riportati nel file il cui nome viene passato nel parametro “fin” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (restituendo in tal caso un booleano pari a false), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “specificYearCourse”. Si controlla che tale vector contenga dieci campi ovvero che siano state riportate nella riga per quel determinato corso di studio tutte e dieci le info necessarie per salvarlo in memoria ovvero:

1. anno accademico;
2. titolo;
3. numero di cfu;
4. numero di ore destinate alle lezioni;
5. numero di ore destinate alle esercitazioni;
6. numero di ore rivolte ai laboratori;
7. numero di versioni in parallelo esistenti per questo corso;
8. elenco degli id dei professori che insegnano quel corso in quel particolare anno, compreso il professore titolare, e relative ore di lezione, laboratorio ed esercitazione;
9. elenco delle info riguardanti l’esame comprendenti la durata dell’esame, il tempo necessario per accedere all’esame, il tempo necessario per abbandonare l’esame, modalità di svolgimento, luogo in cui si terrà;
10. elenco degli id dei corsi raggruppati a quello attualmente in esame;

In caso in cui il numero di campi dovesse essere diverso da dieci viene lanciata una eccezione in cui si specifica il numero corrispondente alla riga in cui si è verificato l’errore (tale variabile contatore che tiene conto del numero di linee lette viene incrementata ogni volta che si rientra nel ciclo del while ed è dichiarata sotto il nome di “line\_counter”).

Nel caso in cui la riga contenga tutte le informazioni necessarie ci si assicura che il corso che viene aggiunto non esista già nel database motivo per cui si procede con un ciclo for nel quale si scandaglia l’intera mappa dei corsi in memoria mediante l’uso di un iteratore che quindi parte dal primo corso in mappa sino all’ultimo (questo possibile grazie all’uso rispettivamente dei metodi begin ed end rivolti agli iteratori). Nel caso in cui dovesse essere presente in memoria un corso uguale a quello che si vuole caricare (appena letto da file), avente pertanto stesso titolo e stesso numero di cfu (caratteristiche che assieme, in assenza di un id, risultano univoche nell’identificazione di un corso) si procede con il lancio di una eccezione che avvisa l’utente della presenza di un corso già esistente informandolo anche del numero della riga corrispondente a quanto accaduto (ancora una volta mediante l’uso di una variabile contatore, “line\_counter”).

Nel caso in cui tale corso non sia già esistente si procede quindi con l’ottenere il codice univoco identificativo per il nuovo corso da inserire mediante l’ausilio dell’apposito metodo “getNewCourseId” creato per la classe “University” (tale metodo viene descritto nel dettaglio nelle righe avvenire).

Si riempie in seguito la mappa interna privata denominata “\_courses” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) con il metodo “insert” al quale viene passato come chiave della mappa il codice del corso (contenuto nella variabile “newIdCourse” e come value corrispondente l’oggetto della classe “Course”, inizializzato mediante costruttore al quale vengono passati (come da definizione presente nel file “Course.h”) i seguenti dati:

1. titolo del corso contenuto in “specificYearCourse [1]”;
2. numero di cfu del corso contenuto in “interoCorso [2]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
3. numero delle ore di lezione del corso contenuto in “interoCorso [3]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
4. numero delle ore di esercitazione del corso contenuto in “interoCorso [4]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);
5. numero di ore di laboratorio del corso contenuto in “interoCorso [5]” (convertito da stringa ad intero mediante il metodo delle stringhe stoi);

Si salva in una variabile di appoggio denominata “acYear” l’anno accademico contenuto ora in “SpecificYearCourse [0]”; il numero di versioni in parallelo contenute nella stringa “SpecificYearCourse [6]” vengono salvate in una variabile intera mediante il metodo di conversione delle stringhe stoi.

Visto il contenuto di “SpecificYearCourse [7]” il quale contiene i diversi dati proprio come scritti nel file (ovvero con caratteri delimitatori come parentesi graffe, quadre e virgole) nasce l’esigenza di togliere le parentesi quadre motivo per cui tale vector viene passato al metodo delle stringhe substr che copia nella stringa “profSenzaQuadre” tutti i caratteri presenti in “SpecificYearCourse [7]” dal secondo al penultimo andando ad escludere così il primo e l’ultimo, rispettivamente “[“ e “]”.

Successivamente si passa il vector “profSenzaQuadre” alla funzione getProfPar che restituisce il vector di stringhe “profCorsoPar” composta da un numero di campi pari al numero di versioni in parallelo dove ogni campo contiene quindi tutti i professori, compreso il titolare, che insegnano quel determinato corso parallelo.

In seguito vengono processate le informazioni riguardanti gli esami contenute in “SpecificYearCourse [8]” copiandole, private delle parentesi graffe aperte e chiuse (mediante il metodo substr) in un vector di stringhe di appoggio detto “examData”. Passando quest’ultimo alla funzione splittedLine insieme al carattere separatore “,” si ottiene il vector “splittedExamData” composto dalle seguenti voci:

* “splittedExamData [0]” contiene la durata dell’esame in minuti;
* “splittedExamData [1]” contiene i minuti necessari per accedere all’esame;
* “splittedExamData [2]” contiene i minuti necessari per abbandonare il luogo dell’esame una volta terminato;
* “splittedExamData [3]” contiene la tipologia di esame “S”/”O” (ovvero scritto/orale);
* “splittedExamData [4]” contiene la tipologia di locazione in cui verrà sostenuto l’esame ”A”/”L” (ovvero aula/laboratorio);

In seguito si prende la stringa contente l’elenco dei codici dei corsi raggruppati contenuti in “SpecificYearCourse [9]” privandoli delle parentesi graffe aperte e chiuse sempre mediante il metodo substr, copiando della stringa tutti i caratteri esclusi il primo (ovvero “{“ ) e l’ultimo (ovvero "}”). La stringa risultante viene passato alla funzione “splittedLine” insieme al carattere separatore che in questo caso è “,”; si ottiene quindi un vector di stringhe “idGrouped”.

QUI MANCA STESSO PEZZO DELLA READCOURSE… vedi messaggi telegram

Come ultima operazione si riempie la mappa interna privata denominata “\_courses” della classe “University” (rispettando la convenzione dell’anteporre un underscore al nome dei dati privati di una classe) in particolare con il metodo “at” si recupera il nuovo corso da aggiungere, mediante il suo id ricavato, e si procede con l’inserimento delle seguenti informazioni inerenti quel corso:

* anno accademico;
* l’informazione booleana che mi dice che il corso è attivo (in quanto appena aggiunto);
* numero di corsi paralleli;
* vector con un numero di campi pari al numero di verisoni parallele del seguente corso; ogni campo comprende quindi le diverse informazioni riguardanti il prof titolare e tutti gli altri prof che insegnano quel determinato corso parallelo con relative ore di insegnamento;
* vector con le Informazioni dell’esame;
* vector con i diversi corsi raggruppati;
* ………YY\_SEMESTER………………. INERENTE IL PEZZO CHE MANCA
* ………STUDYCOURSE………………. INERENTE IL PEZZO CHE MANCA

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

La chiave della mappa dei corsi di studio è quindi il codice identificativo in quanto univoco per ogni corso di studio.

Dopo la chiusura del file il metodo si conclude con la riscrittura automatizzata del file database dei corsi di studio db\_corsi\_studio.txt, dal momento che nuovi corsi di studio sono stati aggiunti nella base dati, il tutto attraverso il metodo dbCourseWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire).

Tale metodo restituisce, come si può notare dal prototipo, un valore true se giunti sino alla riscrittura del file database stesso.

* **getNewStudentId**

Prototipo: const int University::getNewStudentId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “s”) rivolto agli studenti da aggiungere nel database.

Inizialmente si controlla se il database degli studenti risulti vuoto o meno; qualora la mappa degli studenti in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata una matricola priva di significato pratico come: “s00000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultimo studente della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera il value associato all’iteratore “last” (ovvero la matricola associata all’ultimo studente) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewProfessorId**

Prototipo: const int University::getNewProfessorId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “d”) rivolto ai professori da aggiungere nel database. Tale metodo si sviluppa in modo analogo a quello degli studenti e professori.

Inizialmente si controlla se il database dei professori risulti vuoto o meno; qualora la mappa dei professori in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata una matricola priva di significato pratico come: “d00000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultimo professore della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera il value associato all’iteratore “last” (ovvero la matricola associata all’ultimo professore) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewClassroomId**

Prototipo: const int University::getNewClassroomId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “A”) rivolto alle aule da aggiungere nel database. Tale metodo si sviluppa in modo analogo a quello degli studenti.

Inizialmente si controlla se il database delle aule risulti vuoto o meno; qualora la mappa delle aule in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata un codice identificativo privo di significato pratico come: “A000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultima aula della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera il value associato all’iteratore “last” (ovvero l’id associato all’ultima aula) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewStudyCourseId**

Prototipo: const int University::getNewStudyCourseId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo id (privato del carattere “C”) rivolto ai corsi di studio da aggiungere nel database. Tale metodo si sviluppa in modo analogo a quello degli studenti.

Inizialmente si controlla se il database dei corsi di studio risulti vuoto o meno; qualora la mappa dei corsi di studio in memoria sia vuota tale metodo restituisce il valore 1 al fine di impedire che venga generata un codice identificativo privo di significato pratico come: “C000”.

Si dichiara un iteratore “last” con deduzione di tipo (ovvero “auto”) che punta all’ultimo corso di studio della mappa.

Si dichiara una variabile denominata “toReturn” la quale recupera la chiave associata all’iteratore “last” (ovvero l’id associato all’ultimo corso di studi) e la si incrementa di 1 in quanto si vuole creare una nuova matricola consecutiva all’ultima già presente in memoria.

Infine si ritorna, come mostra il prototipo di tale metodo, un numero intero, nonché la nuova matricola generata.

Si noti come questo metodo è stato marcato come const al fine di impedire che il risultato restituito da tale metodo, ovvero la nuova matricola generata, venga modificato.

* **getNewCourseId**

Prototipo: const int University::getNewCourseId() const

Tale metodo, come anticipato nei precedenti, si occupa della generazione di un nuovo codice identificativo rivolto ai corsi da aggiungere nel database.

Si controlla inizialmente se la mappa dei corsi in memoria è vuota o meno; in tal caso l’id da restituire coincide con il primo che si può avere ovvero “01AAAAA”.

In caso contrario, si prende l’ultimo corso presente in memoria (avente quindi l’ultimo id) e lo si salva in un oggetto stringstream detto “Id”. Successivamente si estraggono da quest’ultimo i due campi di cui è composto ovvero una parte numerica detta “num”, costituita da un numero a due cifre, e una parte alfabetica detta “cod” composta da quattro cifre.

Mediante un ciclo for si analizza l’ultima parte dell’id ovvero quella alfabetica iniziando dall’ultimo carattere spostandosi così in ogni ciclo ai caratteri precedenti (in quanto come ogni somma anche quella che opera con dei caratteri prevede di partire dalle posizioni meno significative).

Partendo dall’ultimo carattere si controlla se questo risulta diverso dal carattere “Z”:

* se risulta diverso da “Z” allora si incrementa di uno il carattere in questione, ovvero si passa al carattere successivo in ordine alfabetico e si setta a true un flag booleano detto “noZ” che appena diventa vero provoca al prossimo ciclo l’uscita dal for in quanto non sono necessarie ulteriori modifiche del codice identificativo se non l’incremento della lettera prima citata;
* se risulta uguale a “Z” allora si setta il carattere in questione con quello successivo nell’alfabeto ovvero “A” e si passa con il prossimo ciclo for per l’analisi del carattere nella posizione sempre più vicina al primo; pertanto si riparte dal confronto con il carattere “Z”.

Solo nel momento in cui tutti i caratteri della stringa sono pari ad “A” si prosegue con l’incremento di uno della parte numerica, questo viene fatto con un if esterno al for, diversamente si prosegue col conservare il valore vecchio.

Successivamente mediante il metodo setw si decide di scrivere la parte numerica sempre su due cifre anteponendo uno 0 mediante il metodo setfill nel caso in cui la variabile “cod” sia costituita da un numero ad una cifra.

Si copia l’oggetto stringstream in una stringa la quale conterrà pertanto il nuovo codice identifcativo che verrà restituito come mostrato dal prototipo del metodo.

* **updateStuds**

Prototipo: void University::updateStuds(const std::string &fin)

Metodo lanciato in risposta al comando: -u:s <student\_list\_file>.

Tale metodo consente di aggiornare le informazioni riguardanti degli studenti già presenti in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Si ricorda che:

* al fine di poter aggiornare i dati relativi ad un determinato studente risulta obbligatoria la presenza della sua matricola all’interno delle righe del suddetto file;
* in questo file viene rispettato il medesimo formato già usato nel file db\_studenti.txt dove pertanto il carattere separatore è nuovamente “;” dando la possibilità di poter lasciare vuoti alcuni campi, sottintendendo in tal caso il voler conservare i dati già in essere nel database.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate.

Sucessivamente si controlla che il primo campo del vector “infoStud” (ovvero “infoStud [0]”) contenga effettivamente una matricola; ci si avvale così di un ulteriore metodo della classe Parse, detto **controllItCanBeAnId**, il quale a sua volta controlla innanzitutto se “infoStud [0]” NON sia costituito da 6 caratteri (dove 6 è il numero di caratteri concessi per una matricola di uno studente); in caso affermativo procede con il restituire un boolenao pari a false; nel caso in cui invece sia costituito da 6 caratteri passa al controllare che effettivamente questi sei caratteri siano tutti dei numeri mediante il metodo **checkStoi**; in particolare esso, ricevendo la matricola dello studente privata del carattere iniziale “c”, controlla che i caratteri dell’id siano tutti compresi tra il 48-esimo e il 57-esimo del codice ASCII (corrispondenti rispettivamente al numero 0 e al numero 9). In caso positivo restituisce la stringa contenente la matricola convertita in intero diversamente solleva una eccezione; quest’ultima viene raccolta dal metodo controllItCanBeAnId (motivo per cui il metodo checkStoi viene collocato in una try) che restituisce in tal caso un boolenao pari a false; diversamente, se i controlli invece vanno a buon fine, restituisce true.

Il valore di ritorno del metodo controllItCanBeAnId viene così conservato in un flag detto “canBeAnId” che riassumendo vale:

* true se la matricola è potenzialmente valida (“potenzialmente“ in quanto occorre ancora controllare che esista);
* false se la matricola non è costituita da 6 numeri;

In seguito si salva la matricola dello studente privata del carattere iniziale ‘c’ nella variabile “nMatr”.

Prima di procedere con l’aggiornamento vero e proprio risulta necessario effettuare una serie di controlli riguardanti le informazioni appena lette dalla riga del file; si effettuano così i controlli riportati di seguito:

* Se “\_version” vale 2 (ovvero se la “modalità versioning” è stata attivata) allora si controlla che la dimensione del vector “infoStud” sia pari a sei campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); qualora non fossero pari a sei si usa una stringa detta “error” nella quale si tiene traccia dell’errore di formato verificatosi nel file e settando la variabile “doDbWrite” a false;
* se “\_version” vale 1 (ovvero se la “modalità versioning” NON è stata attivata) allora si controlla che la dimensione del vector “infoStud” sia pari a tre campi (i quali verranno illustrati nelle righe avvenire per mero ordine espositivo); qualora non fossero pari a tre si usa la medesima stringa “error”.

Infine si procede con il settare al valore false la variabile boolenana “doDbWrite”;

* si controlla che il primo campo del vector “infoStud”, che dovrebbe contenere la matricola dello studente, sia vuoto o meno (la cui presenza come spiegato prima deve essere obbligatoria). Qualora lo sia si prende nota dell’accaduto avvalendosi della stringa “error” nella quale si riporta in coda il messaggio dell’assenza della matricola, evidenziando la riga (contenuta in “line\_counter”) in cui si è verificato tale errore;

Infine si procede con il settare al valore false una variabile boolenana detta “doDbWrite”;

* si controlla il valore del flag booleano “canBeAnId”; solo se pari a false (ovvero se se la matricola non è costituita da 6 numeri) si procede anche in questo caso con il riportare nell’apposita stringa “error” il messaggio che informa della presenza di un codice identificativo non valido invidiuato nella riga indicata dal contatore “line\_counter”;

Infine si procede con il settare al valore false “doDbWrite”;

* si controlla infine che la matricola, sino ad ora potenzialmente valida, esista già nel database; al fine di rendere l’utente al corrente dell’accaduto, in caso di assenza si riporta nella stringa “error” la mancata presenza dello studente avente matricola pari a quella in “infoStud [0]”;

Infine si procede con il settare al valore false una variabile boolenana detta “doDbWrite”;

Solo nel momento in cui tutti i controlli sono andati a buon fine, ovvero solo se il flag “doDbWrite” è rimasto al valore di default true, si passa con il processare tutte le nuove informazioni presenti ognuna nei diversi campi di “infoStud” e grazie ad un for ci si muove tra i suoi diversi campi.

Dal momento che è lasciata la possibilità di poter conservare le informazioni già presenti allora si controlla che ogni campo pertanto non sia vuoto. A seconda quindi del valore che assume l’indice “i” si passa da un campo all’altro del vector e mediante l’ausilio di uno switch, in base al valore assunto da “i”, si sceglie quale informazione dello studente si vuole aggiornare. Si noti come è stato definito un elenco di etichette assegnate a dei valori interi costanti al fine di migliorare la leggibilità del codice stesso preservando la rapidità nella comprensione del flusso generale delle operazioni eseguite.

In particolare:

* siccome “infoStud [1]” potrebbe contenere il nuovo nome dello studente allora quando i = 1 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il nome dello studente.

Nel caso in cui il nuovo nome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo nome, “infoStud [1]”, al metodo updateName, creato appositamente per la classe “UniversityMember”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

* siccome “infoStud [2]” potrebbe contenere il nuovo cognome dello studente allora quando i = 2 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il cognome dello studente.

Nel caso in cui il nuovo cognome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo cognome, “infoStud [2]”, al metodo updateSurName, creato appositamente per la classe “UniversityMember”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

* siccome “infoStud [3]” potrebbe contenere la nuova email dello studente allora quando i = 3 significa che si sta puntando a tale campo pertanto si passa all’aggiornare la mail dello studente.

Nel caso in cui la nuova mail risulta diverso dalla precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio della nuova email, “infoStud [3]”, al metodo updateEmail, creato appositamente per la classe “UniversityMember”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

Nel caso in cui la versione sia la due allora potremmo avere l’aggiornamento dei seguenti campi:

* siccome “infoStud [4]” potrebbe contenere la nuova data di nascita dello studente allora quando i = 4 significa che si sta puntando a tale campo pertanto si passa all’aggiornare la data di nascita dello studente.

Nel caso in cui la nuova data di nascita risulta diversa dalla precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio della nuova data, “infoStud [4]”, al metodo updateBirth, creato appositamente per la classe “UniversityMember”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

* siccome “infoStud [5]” potrebbe contenere la nuova data di immatricolazione dello studente allora quando i = 5 significa che si sta puntando a tale campo pertanto si passa all’aggiornare la data di immatricolazione dello studente.

Nel caso in cui la nuova data di immatricolazione risulta diversa dalla precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio della nuova data, “infoStud [5]”, al metodo updateRegistrationOrEntry, creato appositamente per la classe “UniversityMember”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

* siccome “infoStud [6]” potrebbe contenere il nuovo indirizzo di casa dello studente allora quando i = 6 significa che si sta puntando a tale campo pertanto si passa all’aggiornare l’indirizzo dello studente.

Nel caso in cui il nuovo indirizzo risulta diversa dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo indirizzo, “infoStud [6]”, al metodo updateAdress, creato appositamente per la classe “UniversityMember”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Student” che si vuole aggiornare;

Si è usata l’espressione “potrebbe contenere” in quanto non si ha la certezza che tale campo verrà aggiornato o meno o che contenga delle informazioni diverse da quelle già esistenti.

Il ciclo while si conclude con incrementare di 1 la variabile contatore “line\_counter”.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Come di buona norma si procede con la chiusura del file contenente gli studenti da aggiornare nel database.

In seguito si controlla, come nei casi precedenti, il valore assunto dal flag precedentemente presentato, “doDbWrite”.

* **updateProfessors**

Prototipo: void University::updateProfessors(const std::string &fin)

Metodo lanciato in risposta al comando: -u:d <prof\_list\_file>.

Tale metodo consente di aggiornare le informazioni riguardanti i professori già presenti in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso. Il metodo ha lo stesso funzionamento della “updateStuds”. Per questo motivo si invita il lettore a leggere il metodo descritto precedentemente considerando questa volta un vector di stringhe definito come “InfoProf”.

* **updateClassroom**

Prototipo: void University::updateClassroom(const std::string &fin)

Metodo lanciato in risposta al comando: -u:a <aule\_list\_file>.

Tale metodo consente di aggiornare le informazioni riguardanti le aule già presenti in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Anche in questo caso il seguente metodo è molto simile ai metodi di update descritti in precedenza; per questo motivo non verrà descritta in dettaglio.

Finiti i soliti controlli di formato eseguiti anche per la “updateStuds” e “updateProfessors” si procede con il salvataggio delle informazioni su un vector di string relative all’aula letta dalla riga del file. Così, per ogni campo di informazioni, nel caso in cui non fosse vuoto(il campo non va aggiornato) si procede con l’aggiornamento.

Si noti come è stato definito un elenco di etichette assegnate a dei valori interi costanti anche per le Classrooms al fine di migliorare la leggibilità del codice stesso preservando la rapidità nella comprensione del codice e del flusso generale delle operazioni eseguite.

In particolare:

* siccome infoClassroom [1] potrebbe contenere il tipo di aula (aula o laboratorio) allora quando i = 1 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il tipo di aula.

Essendo un campo di tipo char allora nel caso in cui “infoClassroom [1]” contenga il carattere “L” allora si setta al valore true una variabile booleana di appoggio, detta “lab”; in caso contrario ovvero se “infoClassroom [1]” contiene il carattere “A” si procede con il settare “lab” al valore false.

Nel caso in cui il nuovo tipo risulti diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo tipo, il cui stato booleano è contenuto in “lab”, al metodo updateClassroom, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [2] potrebbe contenere il nuovo nome dell’aula allora quando i = 2 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il nome dell’aula.

Nel caso in cui il nuovo nome risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo nome, “infoClassroom [2]”, al metodo updateName, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [3] potrebbe contenere il numero di posti a sedere dell’aula allora quando i = 3 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di posti a sedere dell’aula.

Nel caso in cui il numero di posti a sedere dell’aula risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom [3]”, al metodo updateNSeats, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [4] potrebbe contenere il numero di posti a sedere dell’aula durante un esame allora quando i = 4 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di posti a sedere dell’aula durante un esame.

Nel caso in cui il numero di posti a sedere dell’aula in caso di esame risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom[4]”, al metodo updateNExamSeats, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare.

Nel caso in cui la versione sia la tre allora potremmo avere l’aggiornamento dei seguenti campi:

* siccome infoClassroom [5] potrebbe contenere il numero di tavole da disegno allora quando i = 5 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di tavole da disegno.

Nel caso in cui il numero di tavole da disegno risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom[5]”, al metodo updateDrawingTable, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [6] potrebbe contenere il numero di computer allora quando i = 6 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di computer.

Nel caso in cui il numero di computer risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom[6]”, al metodo updateComputer, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [7] potrebbe contenere il numero di proiettori allora quando i = 7 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di proiettori.

Nel caso in cui il numero di proiettori risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom[7]”, al metodo updateProjector, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

* siccome infoClassroom [8] potrebbe contenere il numero di lavagne allora quando i = 8 significa che si sta puntando a tale campo pertanto si passa all’aggiornare il numero di lavagne.

Nel caso in cui il numero di lavagne risulta diverso dal precedente, si prosegue con l’aggiornamento vero e proprio che prevede il passaggio del nuovo valore, “infoClassroom[8]”, al metodo updateBlackBoard, creato appositamente per la classe “Classroom”. Il tutto viene eseguito con un iteratore, “iter”, che punta all’oggetto “Classroom” che si vuole aggiornare;

Si è usata l’espressione “potrebbe contenere” in quanto non si ha la certezza che tale campo verrà aggiornato o meno o che contenga delle informazioni diverse da quelle già esistenti.

Il ciclo while si conclude con incrementare di 1 la variabile contatore “line\_counter”.

In seguito alla fine dell’intera lettura del file si procede con la chiusura di quest’ultimo e il controllo sulla solita variabile “doDbWrite” che ci porterà a riscrivere o meno il db\_aule.txt.

* **insertCourses**

Prototipo: bool University::insertCourses(const std::string &fin)

Metodo lanciato in risposta al comando: -i:c <file\_esami\_da\_aggiornare>.

Tale metodo consente di aggiornare le informazioni riguardanti un determinato corso già presente in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Si ricorda che:

* al fine di poter aggiornare i dati relativi ad un determinato corso risulta obbligatoria la presenza della sua matricola all’interno delle righe del suddetto file;
* in questo file viene rispettato il medesimo formato già usato nel file db\_corsi.txt dove pertanto il carattere separatore è nuovamente “;” dando la possibilità di poter lasciare vuoti alcuni campi, sottintendendo in tal caso il voler conservare i dati già in essere nel database relativi all’ano accdemico precedente.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “specificYearCourse”.

Successivamente si controlla se tale corso risulta già presente in memoria il tutto mediante una find applicata al primo campo del vector, “specificYearCourse” [0] in quanto quest’ultimo contine l’id univoco del corso stesso. Nel caso in cui tale corso non risulta presente allora la find restituisce l’iteratore all’ultimo elemento e in tal caso viene pertanto lanciata una eccezione che spiega l’assenza di tale corso.

Nel caso in cui sia presente invece si procede coll’inizializzare un iteratore, “course\_with\_given\_id”, con quello già in memoria e corrispondente quindi a ““specificYearCourse” [0]”.

Si controlla se il corso dell’anno precedente contiene tutte le informazioni il tutto mediante il metodo courseOfTheYearIsEmpty applicato all’oggetto corso dell’anno precedente.

Solo nel caso in cui tutti i campi dell’anno precedente non sono vuoti si passa con il riempire, mediante il metodo fillSpecificYearCourse, gli eventuali campi vuoti sul corso dell’anno attuale usando le informazioni inerenti lo stesso corso ma dell’anno precedente infatti tale metodo viene applicato all’oggetto corso dell’anno precedente (ovvero quello già presente in memoria) passando come parametro il corso dell’anno attuale.

Qualora vi sia almeno un campo vuoti in merito all’anno precedente allora si passa con il lanciare delle eccezioni al fine di rendere l’utente al corrente di quanto accaduto. In particolare se il corso dell’anno precedente non contiene tutti i campi, di numero sette, allora si lancia una relativa eccezione. Se sono tutti presenti invece l’assenza di info precedentemente individuata potrebbe essere dovuta alla presenza di campi vuoti per questo ci si avvale dell’uso di un for con il quale si controlla campo per campo quale di questi sia appunto vuoto e lanciando in tal caso una eccezione.

Superati questi controlli si passa con l’analizzare la linea appena letta.

Si salva in una variabile di appoggio, “acYear”, l’anno accademico del corso letto da file che corrisponde al secondo campo di “specificYearCourse”, ovvero “specificYearCourse [1]”.

Dal momento che un corso dura un anno accademico ma occupa due anni solari si passa col salvare il primo dei due, ovvero l’anno di inizio, in una variabile “acStartYear” avvalendosi del metodo delle stringhe substr.

Dai diversi campi del vector “specificYearCourse” estraggo le diverse info:

* “specificYearCourse [3]” contiene il numero di corsi in parallelo che si tengono per quello stesso corso e pertanto viene salvato questo dato come intero nella variabile “num\_parallel\_courses”;
* “specificYearCourse [4]” contiene l’elenco dei professori che insegnano tale corso ciascuno con la sua matricola e le sue ore, incluso il prof titolare per ogni versione in parallelo; quindi proprio come accadeva nella read\_Course si analizzano tale info ottenendo il vector “profCorsoPar” che contiene pe ogni suo campo tutti gli id dei prof che appunto insegnano tale corso;
* “specificYearCourse [5]” contiene le info riguardanti l’esame pertanto si salva nel vector “splittedExamData” le diverse info sull’esame come durata dell’esame, tempo necessario per accedere all’esame e per uscire, modalità di sostenimento dell’esame, scritto o orale, e relativo luogo dedicato allo svolgimento.
* “specificYearCourse [6]” contiene gli id di tutti i corsi ad esso raggruppati che vengono salvati separatamente nel vector “idGrouped”;

Successivamente mediante l’ausilio di un for si scandagliano tutti i corsi di studio salvati n memoria e per ognuno di questi si controlla se sia presente o meno il corso che si sta analizzando; in particolare si utilizza un apposito metodo detto isInWhichSemester il quale a sua volta controlla per ogni semestre di quel dato corso di studio è presente il corso in analisi restituendo in caso affermativo l’anno e il semestre in cui compare salvandolo in una apposita variabile di appoggio detta “res”.

Nel caso in cui è stato trovato in vector d’appoggio di interi detto “studyCourse” si salvano gli id dei corsi di studio in cui tale corso compare accodandoli di volta in volta nel vector.

SI controlla inoltre con il metodo controlGroupedCourses che i corsi raggruppati a quello in analisi non appartengano al suo medesimo corso di studio, sollevando in tal caso una eccezione.

Si precisa che tali operazioni appena spiegate risulteranno poi utili per la parte successiva del progetto ovvero la generazione esami.

Successivamente controllo se tale corso risulta attivo o meno, info contenuta in “specificYearCourse [2]” in particolare nel caso in cui risulta attivo setto al valore true un variabile di appoggio booleana detta “isActive”.

Diversamente viene impostata uguale al valore false ricordando che se un corso viene aggiornato diventando spento, ovvero non più attivo, occorre toglierlo dai corsi attivi e aggiungerlo ai corsi spenti procedura che viene messa in atto mediante il for; mediante tale for in particolare si vanno a scandagliare tutti i corsi di studio presenti in memoria e per ognuno di questi che contiene tale corso, mediante il metodo updateSemestersAndOffCourses, si procede col togliere il corso da quelli attivi e spostarlo in quelli spenti.

**Si presti attenzione al fatto che quando un corso diventa spento tale metodo, updateSemestersAndOffCourses, non solo sposta il corso da quelli attivi in quelli spenti, \_offCourses, ma elimina anche ogni sua informazione dalla mappa dei corsi; dal momento che la lista \_offCourses contiene solo i codici dei corsi spenti se in seguito si dovesse presentare la necessità di elaborare una sessione di esami inerente un periodo accademico in cui tale corso era ancora attivo risulta evidente la necessità di reperire le informazioni relative a quel corso che \_offCourse non contiene e che sono state eliminate dal metodo updateSemestersAndOffCourses; pertanto sempre nel metodo updateSemestersAndOffCourses si procede col salvare il codice del corso spento, l’anno accademico in cui è stato spento ed inoltre l’anno e il semestre di appartenenza il tutto nel vector “\_tempInfoNotActiveCoursesToWriteInTheDB”; in particolare ogni campo del vector “\_tempInfoNotActiveCoursesToWriteInTheDB” contiene queste tre info relative ad uno stesso corso spento.**

Infine si aggiorna il corso in memoria relativo all’anno accademico attuale usando le info salvate nelle diverse variabili di appoggio.

Si ricorda che tali operazioni vengono fatte per ogni riga del file perché presenti nel while sopra citato. Come di buona norma si chiude il file aperto all’inizio del metodo mediante l’apposito metodo close.

Successivamente per ogni corso in memoria si effettuano i seguenti controlli/riempimenti:

* Il metodo fillAcYearEmpty controlla per ogni accademico se per quel dato corso mancano degli anni accademici ovvero se c'è un gap di almeno due anni tra la prima testimonianza di corso individuata in memoria e l'ultima. In tal caso tutti gli anni accademici assenti vengono inizializzati con le stesse info del corso relativo all'ultimo anno accademico prima del gap,
* Il metodo notActive controlla per ogni anno accademico se un dato corso, nei diversi anni, è passato da non attivo ad attivo sollevando in tal caso una eccezione in quanto un corso se spento non può essere "riattivato";
* Il metodo sameSemester controlla se i diversi corsi raggruppati a quello che si sta analizzando appartengono allo stesso semestre scatenando una eccezione nel caso in cui non lo siano;
* Il metodo controlReciprocyGrouped controlla che ci sia reciprocità tra corsi raggruppati ovvero se un corso risulta raggruppato con un altro allora quest’ultimo deve risultare raggruppato con il primo;

Solo dopo aver ultimato i controlli si procede con l’aggiornamento dei file database mediante appositi metodi di scrittura su file.

Se giunto sino a questo punto il metodo è stato eseguito senza interruzioni pertanto viene restituito un bool di valore true come dimostra lo stesso prototipo del metodo in esame.

* **dbStudsWrite**

Prototipo: void University::dbStudsWrite()

Tale metodo si occupa della riscrittura del file database degli studenti; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_studenti.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “Student” si procede col scandagliare tutta la mappa di studenti in memoria. Grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “stud”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_studenti.txt”.

Si noti come mediante un if si è distinta la scrittura delle info base da quelle aggiuntive (riguardanti gli studenti) a seconda se è stata attivata o meno la “modalià versioning”.

* **dbProfsWrite**

Prototipo: void University::dbProfsWrite()

Tale metodo si occupa della riscrittura del file database dei professori; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_professorti.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “prof”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_professori.txt”.

Anche in questo caso vengono effetuate due tipi di stampa diversi in base alla versione.

* **dbClassRoomWrite**

Prototipo: void University::dbClassRoomWrite()

Tale metodo si occupa della riscrittura del file database delle aule; come per i metodi precedenti si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_aule.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Grazie all’overload dell’operatore “<<” si procede con la scrittura su file “fout” dell’oggetto “room”; in particolare l’overload dell’operatore “<<” è stato realizzato affinché in fase di scrittura venga rispettata la formattazione richiesta del file “db\_aule.txt” diversificandone la stampa in base alla versione.

* **dbStudyCourseWrite**

Prototipo: void University::dbStudyCourseWrite()

Tale metodo si occupa della riscrittura del file database dei corsi di studio; si apre anche in questo caso il file database in scrittura e modalità trunc.

Grazie all’overload dell’operatore “<<” si procede con la scrittura su file rispettando la formattazione richiesta del file “db\_ corsi\_studio.txt”.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **dbCourseWrite**

Prototipo: void University::dbCourseWrite()

Infine, tale metodo si occupa della riscrittura del file database dei corsi; il file database di nome “db\_corsi.txt” viene aperto in scrittura e in modalità trunc per poi, grazie all’operatore, scrivere su file le varie informazioni del corso tramite operatore “<<”.

* **controlDatabase**

Prototipo: bool University::controlDatabase(int startAcYear)

Tale metodo controlla se i file sono coerenti all’anno accademico che gli viene passato come parametro intero.

Inizialmente si controlla che i database non siano vuoti, operazione che viene attribuita al metodo dataBaseIsEmpty che per una semplice chiarezza espositiva verrà spiegato nelle righe avvenire.

Terminato il metodo, se non è stata sollevata alcuna eccezione da esso, si procede col controllare se un certo corso contenga effettivamente le info relative all’anno accademico richiesto, mediante il metodo controlExistenceSpecificYear appositamente creato. Questa operazione viene ripetuta per ogni corso presente in memoria grazie ad un for. In particolare al metodo controlExistenceSpecificYear, rivolto quindi agli oggetti della classe “Course”, viene passato il codice identificativo del corso e l’anno accademico di interesse.

Nello specifico controlExistenceSpecificYear, usando il metodo count, controlla se nella mappa “\_courseOfTheYear” (ovvero la mappa contenente le info di quel corso per i diversi anni accademici) sono presenti informazioni in merito all’anno passato come parametro al metodo. In caso negativo si procede con lo scatenare una eccezione diversamente viene restituito il valore bool true.

Concluso il for se no sono state sollevate eccezioni il metodo in esame, controlDatabase, restituisce true.

* **dataBaseIsEmpty**

Prototipo: bool University::dataBaseIsEmpty(int startAcYear)

Tale metodo come anticipato precedentemente si occupa di controllare se i database in memoria sono vuoti o meno. Tale controllo viene eseguito per ogni database del quale il progetto fa uso scatenando in caso di database vuoto una eccezione specifica. Tale controllo viene quindi rivolto ai seguenti database in memoria:

* \_professors;
* \_students;
* \_courses;
* \_studyCourse;
* \_acYearSessions al quale si controlla se è vuoto ma anche se non sono presenti info sulle sessioni di esami relative all’anno accademico in corso generando in entrambi i casi una eccezione;

Infine tale metodo mediante un for controlla che per ogni corso per quel determinato anno accademico ci siano studenti iscritti e qualora il numero degli iscritti sia uguale a 0 viene sollevata una eccezione specificando il corso che non presenta studenti.

* **controlGroupedCourses**

Prototipo: bool University::controlGroupedCourses(int idStudyCourse , std::vector<std::string> &idGrouped, std::string nameCourse , int line\_counter, std::string idCourse )

Il metodo seguente nasce col fine di controllare se i corsi raggruppati contenuti nel vector di stringhe “idGrouped”, passato come parametro, non siano appartenenti al corso di studio identificato dall’id contenuto nel parametro “idStudyCourse” e al quale appartiene il corso passato come parametro mediante “nameCourse” ed “idCourse”.

Si salva in un vector di stringhe detto “allCoursesOfCdS” tutti i corsi del corso di studio identificato dal parametro intero “idStudyCourse”. Con l’aiuto di un for, per ogni stringa del vector “idGrouped” si controlla se tale stringa, ovvero se tale corso raggruppato, sia presente tra i corsi del CdS in esame (alias corso di studio in esame). In caso affermativo si solleva una eccezione; con quest’ultima si rende l’utente partecipe del fatto che

il corso raggruppato j-esimo, ovvero “idGrouped [j]”, appartiene allo stesso CdS (indicato da “idCourse”) al quale appartiene il corso “idCourse”.

Il metodo coerentemente con il suo tipo di ritorno restituisce un bool di valore true perché giunti sino al termine del for.

* **thereIsAHoleInTheCoursesCodes**

Prototipo: void University::thereIsAHoleInTheCoursesCodes()

Questo metodo controlla che non ci siano buchi negli id dei corsi per tutti i corsi di studio ovvero se tutti i corsi hanno id che sono consecutivi.

Mediante un for, per ogni corso di studio si salvano tutti i suoi corsi in un vector di stringhe “codesReturned” in modo che ogni suo campo contenga l’id di uno specifico corso. Codesti id vengono salvati di volta in volta nel vector di stringhe “allCoursesCodesFromSemesters”. Alla fine di tale for “allCoursesCodesFromSemesters“ contiene tutti i codici di tutti i corsi di ogni CdS; in particolare ogni suo campo contiene solo un id.

Al fine di semplificare la ricerca dei buchi si procede con l’ordinare in modo crescente tutti gli id dei corsi presenti nel vector “allCoursesCodesFromSemesters”.

Dal momento che diversi corsi di studio possono avere dei corsi in comune allora si ha che il vector “allCoursesCodesFromSemesters” possa contenre delle repliche dgli id; pertanto si procede con i due seguenti metodi alla loro eliminazione dal vector stesso:

* il metodo unique colloca gli id che son delle repliche nella coda del vector restituendo la posizione a partire dalla quale si possono individuare le repliche nel vector stesso;
* si usa il metodo erase che elimina tutti gli elementi a partire dalla posizione restituita da unique fino alla fine del vector ovvero le posizioni in corrispondenza dele quali sono presenti le repliche;

In questo modo il vector “allCoursesCodesFromSemesters” contiene gli id dei corsi senza repliche e disposti in ordine crescente.

A partire dal secondo, per ogni corso presente nel vector si calcola la distanza tra esso e il suo precedente mediante il metodo checkDistance (di seguito spiegato) il quale in caso di buco tra i due genera una eccezione.

* **checkDistance**

Prototipo: void University::checkDistance(std::string &minor, std::string &major)

Tale metodo si occupa di controllare la distanza presente tra due id successivi passati come parametro, denominati “minor” e “major”. Si salva la dimensione espressa in bit delle due stringhe, “minor” e “major”, rispettivamente in “size\_min” e “size\_major”. Nel caso in cui le due dimensioni siano diverse, ovvero nel caso in cui le due stringhe siano composte da un numero di caratteri/numeri diversi si lancia una eccezione mostrando la presenza di un errore nel codice del corso.

Nel caso in cui siano uguali si procede con l’analisi della distanza.

Attraverso un for si salva la posizione in corrispondenza della quale i codici iniziano a cambiare: “posNotEqual”.

Si calcola la distanza in termini di numero di caratteri alfanumerici che si interpongono tra i due diversi (individuati dal for) e si salvano nella variabile “distance”.

Nel caso in cui “distance” sia uguale ad 1, si possono avere due scenari:

* se il controllo della distanza è fatto in posizione 0 bisogna controllare che i caratteri di dopo siano 0 (major) e 9 (minor), in quanto c'è ancora un numero prima delle lettere; dopo occorre controllare che le lettere siano tutte A (major) e Z(minor);
* se non si ha la prima parte del numero ma direttamente lettere allora per considerare tutti i casi si ha che devono essere vere due condizioni per scatenare l'eccezione: dove i caratteri iniziano ad essere diversi devono essere diversi da 0 o 9 e da A o Z;

siccome se un carattere nel “major” cambia allora vuol dire che tutti i caratteri precedenti a quelli nel “minor” sono tutti ‘Z’ allora si ha che

Se “distance” non è uguale ad 1 quindi i due id non sono consecutivi pertanto si lancia una eccezione;

Se si giunge alla fine del metodo allora non è stata sollevata alcuna eccezione di conseguenza i due id sono consecutivi.

* **controlReciprocyGrouped**

Prototipo: void University::controlReciprocyGrouped()

Tale metodo ha l’obiettivo di controllare se c’è reciprocità tra i corsi raggruppati: se un corso risulta raggruppato con un altro allora anche quest’ultimo deve esserlo con il primo diversamente viene sollevata una eccezione.

Il metodo si articola nel seguente modo:

* attraverso il primo for, per ogni corso presente nel database, si crea una mappa all’interno della quale si salva per ogni anno accademico un vettore contenente tutti i corsi raggruppati di quell’anno mediante il metodo getGroupedCourseFromAllYear; tale mappa viene denominata appunto “allGroupedCourse”;
* successivamente con il secondo for annidato nel primo si cicla sulla mappa appena creata, “allGroupedCourse”, e in particolare per ogni anno accademico si utilizza un vector al cui interno si salvano i corsi raggruppati aggiungendo in coda il corso considerato detto “groupedOfThisYear”;
* a questo punto per ogni corso raggruppato di “groupedOfThisYear” si procede coll’eseguire lo stesso passaggio effettuato nel ciclo esterno: mediante un terzo for, annidato al secondo, si mette a disposizione un altro vector contenente contenenti i corsi raggruppati a quello puntato dal for in quel determinato ciclo ottenendo così “allGroupedCourse2”; quest’ultimo pertanto contiene tutti i corsi raggruppati aggiungendo in coda, mediante il metodo push, il corso stesso;
* il quarto ed ultimo for viene creato quindi con il semplice scopo di confrontare il contenuto dei due vector realizzati: “allGroupedCourse” “allGroupedCourse2”;

ad ogni ciclo di for si confrontano i due vector e nel momento in cui un corso raggruppato compare in entrambi si procede con l’eliminazione, mediante il metodo erase, dal vector “allGroupedCourse2”;

se un corso raggruppato compare solo in uno dei due e non entrambi come ci si aspetta allora si solleva una eccezione con la quale si rende l’utente al corrente del fatto che la reciprocità dei corsi non è rispettata indicando che ciò accade per il corso.

Se al termine di questo quarto for il vector “allGroupedCourse2” contiene ancora elementi allora vuol dire che la reciprocità non è nuovamente soddisfatta in quanto c’è un corso che presenta dei corsi raggruppati in più;

* **dbCourseNotActive**

Prototipo: void University::dbCourseNotActive()

Tale metodo fa in modo di tenere traccia all’interno di un apposito file delle informazioni riguardanti i corsi spenti; informazioni quindi inerenti sessioni precedenti alla data di disattivazione e che, come è stato accennato nei metodi precedenti, sono contenute in “\_tempInfoNotActiveCoursesToWriteInTheDB”; tali info infatti se non scritte in un file ma lasciate salvate in un vector, come “\_tempInfoNotActiveCoursesToWriteInTheDB”, andrebbero perse in fase di terminazione del programma impedendo l’elaborazione di eventuali sessioni di esami relative ad anni in cui un corso, attualmente spento, era attivo .

Si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “offCourses\_db.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for si punta ai diversi elementi del vector “\_tempInfoNotActiveCoursesToWriteInTheDB” andando a scandagliare quindi tutte le info relative ai corsi spenti. In particolare tale vector contiene in ogni suo campo le seguenti tre informazioni:

* codice del corso;
* anno accademico a partire dal quale si è spento;
* anno e semestre di appartenenza (ad esempio 1 – 2 nel caso di un corso del 1o anno 2o semestre);

Per ogni ciclo for si procede con la scrittura sul file “fout” dei diversi campi del vector andando a capo con un endl non appena si passa all’elemento successivo. Alla fine di tale metodo pertanto il file “offCourses\_db.txt” dedica ogni sua riga alle tre info del corso spento.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **readCourseNotActive**

Prototipo: void University::readCourseNotActive()

Questo metodo si occupa della lettura dal file “offCourse\_db.txt” delle informazioni di corsi diventati spenti.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (addebitata all’assenza del file stesso), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio detto “line”, viene salvata di volta in volta in coda alle altre info nel vector “\_tempInfoNotActiveCoursesToWriteInTheDB”.

Sempre la stringa “line” viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “token” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”:

* token [0] contiene il codice del corso spento;
* token [1] contiene l’anno accademico a partire dal quale è diventato spento;
* token [2] contiene l’anno e il semestre di appartenenza;

Infine per il corso spento letto nella riga si va ad assegnare il semestre a cui apparteneva e l’anno accademico a partire dal quale è diventato spento mediante il metodo assignYY\_Sem.

* **versioning**

Prototipo: void University::versioning(std::string newVersion)

Metodo lanciato in risposta al comando: -up <versione>.

Questo metodo permette di gestire l’operazione di aggiornamento dei database già esistenti con l‘inclusione di informazioni aggiuntive inerenti gli studenti, i professori e le aule.

A tale metodo si passa come parametro una stringa contenente il numero corrispondente alla versione che si vuole mettere in atto e mediante il metodo stoi si trasforma tale stringa in un intero detto “version”:

* se version = 2 allora la procedura di aggiornamento riguarda il database degli studenti e dei professori ai quali vengono aggiunte le seguenti info: data di nascita ed indirizzo di casa per entrambi, data di immatricolazione per gli studenti e data di immissione di ruolo per i professori;

ai prof e agli studenti vengono aggiunti ad entrambi data di nascita ed indirizzo di casa mentre solo

* se version = 3 allora la procedura di aggiornamento riguarda le aule alle quali vengono aggiunte le info riguardanti le strumentazioni necessarie per sostenere un esame;

Si usa poi il metodo renameOldDataBase (spiegato in seguito per chiarezza espositiva) che si occupa del rinominare i database già esistenti aggiungendo “\_old” tra il nome del file e l’estensione “.txt”.

A seconda del valore assunto dalla variabile “version” si procede con lo scrivere i nuovi database contenenti le info aggiuntive. Si ricorda che le info aggiuntive se non modificate dall’utente mediante un apposito metodo vengono salvate (e pertanto anche scritte nel nuovo file database) con dei valori di default. In particolare:

* se version = 2 allora si procede con la scrittura del database degli studenti e dei professori (db\_studenti.txt e db\_professori.txt ) che pertanto adesso conterranno queste nuove info;
* se verison = 3 allora si procede con la scrittura del database delle aule che pertanto adesso conterrà queste nuove info (db\_aule.txt);

Il tutto si conclude con il metodo writeVersion che scrive su un apposito file la versione adottata affinchè qualora il programma dovesse terminare si tiene traccia del contenuto che possedeva la variabile \_version.

* **renameOldDataBase**

Prototipo: bool University::renameOldDataBase(int version)

Questo metodo è stato creato per rinominare i database già esistenti aggiungendo “\_old” tra il nome del file e l’estensione “.txt”. A seconda del valore che assume version si decide su quale database rivolgere l’operazione di modifica del nome del file:

* se version = 2 occorre rinominare i database degli studenti e dei professori e siccome i nomi dei file sono noti a priori si dichiarano due vettori di caratteri per database:

“oldname [ ]” = “db\_studenti.txt” “newname [ ]” = “db\_studenti\_old.txt”

“oldname2 [ ]” = “db\_professori.txt” “newname2 [ ]” = “db\_professori\_old.txt”

Si fa uso infine del metodo rename, apposito per i file, che rinomina i file database degli studenti e dei

professori con i nuovi nomi (“newname [ ]” e “newname2 [ ]”) al posto dei precedenti (“oldname [ ]” e

“oldname2 [ ]”).

Nel caso in cui i metodi rename restituiscono dei valori diversi da zero significa che il metodo non è andato a buon fine pertanto si solleva una eccezione con la quale si comunica che i file non sono stati rinominati adeguatamente.

se version = 3 l’operazione di aggiornamento è dedicata al database delle aule e siccome il nome del file è noto a priori si dichiarano due vettori di caratteri:

“oldname [ ]” = “db\_aule.txt” “newname [ ]” = “db\_aule\_old.txt”

Si fa uso infine del metodo rename, apposito per i file, che rinomina il file database delle aule con il nuovo

nome (“newname [ ]”) al posto del precedente (“oldname [ ]”).

Nel caso in cui il metodo rename restituisce un valore diverso da zero significa che il metodo non è andato a buon fine pertanto si solleva una eccezione con la quale si comunica che il file non è stato rinominato adeguatamente.

* **addStudyPlan**

Prototipo: void University::addStudyPlan(std::string fin)

Metodo lanciato in risposta al comando: -a:p <piani\_di\_studio\_file>.

Questo metodo si occupa del salvataggio in memoria di nuovi piani di studio (ulteriori a quelli presenti nel file “db\_piano\_studi.txt”) per alcuni studenti.

Tali piani di studio sono riportati nel file il cui nome viene passato nel parametro “fin” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate.

Ogni riga viene sottoposta a controlli di formato e coerenza come ad esempio ci si assicura che la matricola sia presente nella riga o che sia ammissibile tramite la funzione già citata “controlItCanBeAnId” (presente in Parse). Nel caso in cui i controlli non vadano a buon fine verrà segnalato in una stringa “error” e verranno settati due booleani “doDbWrite” ed “isOk”; il primo di carattere generale (se almeno una volta sarà false allora il database non verrà riscritto; il secondo ci permetterà di fare i controlli sulle altre righe del file segnalando eventuali altri errori specificandone il tipo e la riga del file da ricontrollare).

* Si controlla anche che la stringa “acYearRegistration” contenga effettivamente un anno accademico e che questo non sia vuoto; ci si avvale così di un ulteriore metodo della classe Parse, detto “**controllItCanBeAnAcYear”**, il quale a sua volta controlla innanzitutto se “acYearRegistration” NON sia costituito da 9 caratteri (dove 9 è il numero di caratteri concessi per definire un anno accademico ricordando infatti come quest’ultimo si manifesti nel formato AAAA-AAAA, rispettivamente anno di inizio e di fine); in caso affermativo procede con il restituire un boolenao pari a false; nel caso in cui invece sia costituito da 9 caratteri si estraggono da “ acYearRegistration” le stringhe “acStartYear” e “acEndYear” contenenti rispettivamente l’anno di inizio e di fine di quello accedemico.

Il metodo controllItCanBeAnAcYear si conclude con un ultimo check ovvero si controlla che i due anni “acStart” ed “acEnd” siano effettivamente consecutivi; nel caso in cui non lo siano il metodo restituisce un bool pari a false; diversamente, se tutti i precedenti controlli vanno a buon fine, restituisce true.

Il valore di ritorno del metodo controllItCanBeAnId viene così conservato in un flag detto “canBeAnAcYear” che riassumendo vale:

* true se l’anno accademico è valido;
* false se l’anno accademico non è valido;

Si controlla il valore del flag booleano “canBeAnAcYear”; solo se pari a false (ovvero se l’anno accademico non è costituita da 2 numeri consecutivi) si procede anche in questo caso con il riportare nell’apposita stringa “error” il messaggio che informa della presenza di un anno accademico non valido inviduato nella riga indicata dal contatore “line\_counter”;

Infine si procede con il settare al valore false due variabili boolenane dette “doDbWrite” ed “isOk”;

* Solo se i controlli precedenti sono andati tutti a buon fine si procede con le seguenti istruzioni.

Mediante il metodo substr si copiano all’interno di una stringa detta “allCourses” i codici di tutti i corsi appartenenti al piano di studio dello studente presenti in “line”.

* Infine si recupera lo studente già presente nel database (avvalendosi della matricola in quanto chiave della mappa privata “\_students”) al fine di salvare il piano di studio di quello specifico studente appena letto da file; si procede usando un apposito metodo detto “**addStudyPlanPerStudent** “della classe “Student” al quale si passa l’anno accademico di immatricolazione dello studente e il vector con i corsi del suo piano di studio; tale metodo a sua volta per quel determinato studente salva in memoria:
* l’anno di immatricolazione nella variabile intera privata “\_yearRegistration” della classe Student;
* i corsi del piano di studio nella variabile vector privata “\_studyPlan” sempre della classe Student;

Nel caso in cui il piano di studio in memoria del medesimo studente fosse già esistente allora il metodo addStudyPlanPerStudent solleva una eccezione la quale viene catturata dal metodo addStudyPlan (motivo per cui addStudyPlanPerStudent è stato collocato all’interno di un try). L’aver scatenato una eccezione è giutificato dal fatto che lo studente appena letto da file non deve già possedere un piano di studio nel momento in cui si usa il comando di aggiunta; quello è consentito solo con metodi di update; in caso di errore in particolare si provvede a “prendere nota” all’interno della stringa “error”.

* In seguito si salva in memoria per ogni corso lo studente che lo frequenta; si usa il metodo “**registerStudentsToSpecificyearCourses”** al quale si passa il vector dei corsi, l’oggetto stud e l’anno di immatricolazione, sotto-forma di intero; tale metodo prima di procedere con il caricamento vero e proprio esegue un controllo inerente l’unicità dei corsi presenti nel piano di studio mediante un ulteriore metodo della classe Student, controlUnicity. La seguente indentazione contiene la spiegazione di tale funzione:

Il metodo **controlUnicity,** controlla quante sue occorrenze siano presenti nel medesimo vector dei corsi di studio (“controlUnicityCourses” nonché il vector “courses” del metodo chiamante). Nel caso in cui siano presenti più testimonianze del corso si procede innanzitutto con il prendere nota dell’accaduta all’interno di una stringa “error”, interna al metodo, nella quale si riporta appunto che il corso avente id pari a “controlUnicityCourse [i]” risulta presente più volte. Si setta al valore false un flag detto “isOk” e solo in seguito si procede con l’eliminazione delle repliche di “controlUnicityCourse [i]” dal vector “controlUnicityCourse”; dal momento che il vector “controlUnicityCourse” è stato modificato si preferisce ricominciare il for dal primo elemento per ulteriori controlli come ad esempio dimostrare che un corso esista, che il corso fosse attivo nell’anno di iscrizione dello studente e che non sia ormai spento in quell’anno.

Se i controlli vanno a buon fine si procede con il registrare lo studente in corrispondenza di quel determinato corso in quell’anno specifico in memoria.

Alla fine del metodo registerStudentsToSpecificyearCourses si controlla se il flag è false ed in caso affermativo, ovvero se si sono riscontrate delle irregolarità, allora si procede con il comunicarle al metodo chiamante mediante il sollevameno di una eccezione che verrebbe catturata dal metodo addStudyPlan (motivo per cui il metodo registerStudentsToSpecificyearCourses è contenuto in una try).

* Si incrementa infine di una unit il contatore di righe lette da file, “line\_counter”.

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

In seguito si controlla il valore assunto dal flag precedentemente presentato, “doDbWrite” e conseguente scrittura del database in caso di assenza di errori, il tutto attraverso il metodo dbStudyPlanWrite (che verrà spiegato solo per puro ordine espositivo nelle righe avvenire). Inoltre si procede anche alla riscrittura del file database dei corsi db\_corsi.txt in quanto sono state aggiunte nuove informazioni riguardanti i corsi, il tutto mediante il metodo dbCourseWrite.

**Quindi alla fine del metodo addStudyPlan è noto:**

* **sia per gli studenti quali corsi appartengono al loro piano di studio (**“\_studyPlan”**) in quell’anno accademico (**“\_yearRegistration”**) grazie a addStudyPlanPerStudent;**
* **viceversa per i corsi quali studenti lo frequentano in quell’anno accademico grazie a registerStudentsToSpecificyearCourses;**

**Si noti come si è rispettata la medesima scelta adottata per le procedure di “aggiunta”, ovvero si noti come anche in questo metodo, in fase di progettazione, si è preferito proseguire con la riscrittura del file solo nel momento in cui tutti gli studenti contenuti nel file non presentano errori.**

* **dbStudyPlanWrite**

Prototipo: void University::dbStudyPlanWrite()

Tale metodo si occupa della riscrittura del file database dei piani di studio degli studenti; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_piano\_studi.txt” in modalità scrittura e trunc affinchè eventuali contenuti già presenti nel file vengano sovrascritti

Ogni riga del file è dedicata al piano di studio di un certo studente.

Dal momento che non si ha la certezza che il piano di studi di un dato studente sia stato già definito in precedenza si ha che la stampa avviene solo per quegli studenti il cui piano non è vuoto; il tutto grazie ad un if che controlla che il valore booleano restituito dall’apposito metodo studyPlanIsEmpty sia false.

* **readStudyPlan**

Prototipo: void University::readStudyPlan()

Questo metodo si occupa della lettura del database del piano di studio degli studenti denominato: db\_piano\_studi.txt. Come si può facilmente intuire lo scheletro di tale metodo è analogo a quello di addStudyPlanin quanto le operazioni sono le medesime dal momento che cambia solo il file sul quale si opera.

Per completezza si riportano di seguito i diversi passaggi.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate.

In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga è salvata in un oggetto stringa di appoggio detto “line”.

In seguito si passa all’analisi delle info contenute nella stringa “line”:

* si copia la matricola dello studente presente in “line” (che copre tutti i caratteri a partire da quello alla posizione 0 per un numero di caratteri pari a 7) mediante il metodo substr all’interno di una stringa detta appunto “matr”;
* si copia l’anno accademico di immatricolazione dello studente presente in “line” (che copre tutti i caratteri a partire da quello alla posizione 8 per un numero di caratteri pari a 9) mediante il metodo substr all’interno di una stringa detta appunto “acYearRegistration”;
* si copiano i codici dei corsi appartenenti al piano di studio dello studente presenti in “line” (che copre tutti i caratteri a partire da quello alla posizione 19 per un numero di caratteri pari a quelli che consentono di arrivare alla fine dell’elenco) mediante il metodo substr all’interno di una stringa detta appunto “allCourses”;

Ci si avvale poi di un oggetto stringstream “ss” per estrare la parte numerica della matricola dello studente isolandola pertanto dal carattere ‘s’.

Si applica la funzione splittedLine alla stringa contenente l’elenco dei corsi considerando come carattere separatore “;” e ottenendo così il vector di stringhe “courses”; ogni campo di “courses” è dedicato ad uno specifico corso.

Successivamente si dichiara un oggetto di appoggio detto “stud”, della classe Student, al fine di salvare il piano di studio di quello specifico studente appena letto da file; si procede usando un apposito metodo detto addStudyPlanPerStudent al quale si passa l’anno accademico di immatricolazione dello studente e il vector con i corsi del suo piano di studio; tale metodo a sua volta per quel determinato studente salva in memoria:

* l’anno di immatricolazione nella variabile intera privata “\_yearRegistration” della classe Student;
* i corsi del piano di studio nella variabile vector privata “\_studyPlan” sempre della classe Student;

Successivamente si salva in memoria per ogni corso lo studente che lo frequenta nel seguente modo:

si usa il metodo registerStudentsToSpecificyearCourses al quale si passa il vector di corsi, l’oggetto stud e l’anno di immatricolazione, sotto-forma di intero; tale metodo a sua volta per ogni corso che gli viene passato come parametro nel vector “courses” applica la funzione registerStudentsToSpecificYear; quest’ultimo metodo, rivolto agli oggetti della classe “Course”, assegna a quel determinato corso in quel determinato anno accademico, ovvero all’oggetto privato “\_courseOfTheYear”, l’oggetto studente con le info appena lette da file.

**Quindi alla fine del metodo readStudyPlan è noto:**

* **sia per gli studenti quali corsi appartengono al loro piano di studio (**“\_studyPlan”**) in quell’anno accademico (**“\_yearRegistration”**) grazie a addStudyPlanPerStudent;**
* **viceversa per i corsi quali studenti lo frequentano in quell’anno accademico grazie a registerStudentsToSpecificyearCourses;**

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Il metodo si conclude con la chiusura del file che ha consentito l’aggiunta di ulteriori piani di studio.

* **updateStudyPlan**

Prototipo: void University::updateStudyPlan(std::string fin)

Metodo lanciato in risposta al comando: -u:p <piani\_di\_studio\_aggiornato>.

Tale metodo consente di aggiornare le informazioni riguardanti i piani di studio degli studenti già presenti in anagrafica con i nuovi dati riportati in un file apposito, il cui nome viene passato nel parametro “fin” del metodo stesso.

Si ricorda che la modifica risulta “secca” ovvero che il piano di studi può essere aggiornato solo per intero e questo fa si che in tale metodo si eseguano le medesime operazioni presenti nel metodo addStudyPlan perché la struttura del file è la medesima, con l'ulteriore controllo di coerenza sull'anno di iscrizione dello studente. Pertanto per brevità si rimanda alla spiegazione fatta sul metodo precedente.

* **insertStudentGrades**

Prototipo: bool University::insertStudentsGrades(std::string fin)

Metodo lanciato in risposta al comando: -i:v <id\_corso>\_<data\_appello>(\_[\*]).csv

Questo metodo consente di leggere e salvare in memoria le matricole e i voti degli studenti che hanno superato un determinato esame il cui corso corrispondente è contenuto nel nome del file insieme alla data dell’appello. Il nome del file viene passato come parametro al metodo.

Si apre quindi il file in modalità lettura e si eseguono i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso.

Successivamente si estrae dal nome del file sia l’id del corso che la data dell’appello, contenuti nel nome del file nonché parametro di tipo stringa detto “fin”; in particolare si usa il metodo find\_last\_of che applicato a “fin” restituisce la posizione occupata, all’interno della stringa, dai caratteri che gli vengono passati ovvero ‘/\\’; in questo modo si conosce la posizione (in termini di numero di caratteri) fino alla quale sono presenti le info di interesse; pertanto usando il metodo substr si copiano tutti i caratteri di “fin” a partire dal primo sino a quello di ultimo interesse, nonché quello restituito dal metodo find\_last\_of (si aggiunge +1 in quanto la numerazione parte da 0); si ottiene la stringa “base\_filename” che contiene quindi solo l’id del corso e la data dell’appello; ci si avvale nuovamente del metodo substr per salvare le due info in due stringhe separate: “idCorso” e “appealDate”; quest’ultimo viene convertito in intero e salvato in “appealYear”.

Prima di procedere con il caricamento dei voti si controlla che:

* il corso esista effettivamente in memoria mediante il metodo find (al quale viene passato l’id del corso in quanto univoco). In caso di assenza si solleva una eccezione con la quale si tiene l’utente al corrente della non esistenza del corso.
* la data dell’appello fornita esista anch’essa, il tutto avvalendosi del metodo controlAppeal rivolto agli oggetti della classe “Course”; tale metodo in prima battuta controlla se esiste un corso con quell’id in quell’anno accademico e solo in caso affermativo controlla anche se esistono delle date d’esame relative a quel corso in quell’anno; in particolare per quest’ultimo controllo si avvale del metodo getAllAppeals che riesce ad indicare quali sono le date di tutti gli appelli di quel determinato corso, accedendo al vector di “Date” privato detto “\_howManyTimesIAmAssignedInASession” (che a sua volta contiene per un dato semestre le date degli appelli);

Giunti a questo punto si passa alla lettura vera e propria del file che avviene ancora una volta mediante un while con il quale quindi ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga è salvata in un oggetto stringa di appoggio detto “line”.

Si estraggono da essa i seguenti dati:

* si copia la matricola dello studente presente in “line” (che copre tutti i caratteri a partire da quello alla posizione 0 per un numero di caratteri pari a 7) mediante il metodo substr all’interno di una stringa detta appunto “idmatr”;
* si copia l’esito dell’esame presente in “line” (che copre tutti i caratteri a partire da quello alla posizione 8 fino al successivo, nonché alla posizione 9) mediante il metodo substr per poi convertirlo in intero salvandolo nella variabile “mark”

Prima del salvataggio si effettua un ulteriore controllo; in particolare con il metodo find, a cui viene passata la matricola dello studente (in quanto chiave univoca di tale mappa), si procede con il ricercare se tale studente sia presente o meno in anagrafica lanciando una eccezione in caso di mancata presenza.

Nel caso in cui lo studente esista si dichiara un oggetto di appoggio “stud” inizializzato con le info dello studente in anagrafica precedentemente cercato con la find.

Infine solo se l’esame è superato, ovvero solo se il voto è maggiore o uguale a 18, si procede con il caricamento che prevede di caricare:

* l’anno accademico;
* lo studente che ha superato l’esame;
* l’anno in cui si è tenuto l’appello;
* il voto di conseguimento;
* la data dell’appello;

Queste info vengono usate dal metodo modifyStudentAsPassedToSpecYearCourse al fine di aggiornare anche i dati privati del corso di quell’anno (“\_courseOfTheYear”); infatti esso contiene la funzione addGradeToStudent (tipica della classe SpecificYearCourse) che aggiorna anche il numero totale di studenti che ha superato quello stesso esame; numero che viene decrementato di una unità.

Si ricorda che tutte queste operazioni vengono effettuate su ogni riga del file.

Il metodo insertStudentGrades si conclude con la chiusura del file e la riscrittura del file database contenente gli appelli già avvenuti.

* **registerStudentsToSpecificYearCourses**

Prototipo: void University::registerStudentsToSpecificYearCourses

(std::vector<std::string>& courses, Student& stud, int acYearRegistration)

Questo metodo salva in memoria per ogni corso (che gli viene passato come parametro nel vector di stringhe “courses”) lo studente che lo frequenta con il relativo anno accademico (dati che sono anch’essi parametri del metodo stesso, rispettivamente “stud” ed “acYearRegistration”).

Con l’aiuto di un ciclo for si scandaglia l’intero vector “courses” e per ogni suo corso si controlla prima che tale corso esista in anagrafica, lanciando una eccezione in caso di assenza.

Qualora l’esito sia affermativo si procede invece coll’utilizzare la funzione registerStudentsToSpecificYear, creata appositamente per gli oggetti della classe “Course”, alla quale viene passato l’anno e lo studente affinché salvi in memoria in corrispondenza di quel determinato corso e anno lo studente che lo ha frequentato (ovvero andando a lavorare sull’oggetto privato “\_courseOfTheyear”).

* **dbAppealsWrite**

Prototipo: void University::dbAppealsWrite()

Tale metodo si occupa della riscrittura del file database degli appelli; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome “db\_appelli.txt” in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “Course” si procede col scandagliare tutta la mappa di corsi in memoria. Per ogni ciclo for, quindi per ogni corso, si usa la funzione dedicata ai corsi detta getAcYaerStudExam che restituisce un vector di stringhe il cui contenuto è strutturato nel seguente modo: ogni suo campo contiene le seguenti info separate dal carattere “;”:

* codice del corso;
* anno accademico in cui si è tenuto il corso;
* data (yyyy-mm-dd) dell’appello;
* matricole degli studenti che hanno superato l’esame affiancate dal relativo voto ciascuno racchiuso in delle parentesi graffe;

Infine con un ulteriore for si stampa ciascun campo andando a capo al passaggio del campo successivo.

Il metodo si conclude con la chiusura del file “db\_appelli.txt”.

* **readPassedAppeals**

Prototipo: void University::readPassedAppeals()

Questo metodo consente la lettura del database degli appelli denominato: db\_appelli.txt .

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “appeal” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”. Si passa con l’estrarre le diverse info dal vector “appeal”:

* appeal [0] contiene il codice identificativo del corso che viene salvato nella stringa “idCorso”;
* appeal [1] contiene l’anno accademico che viene salvato nella stringa “acyear”;
* appeal [2] contiene la data dell’appello che viene salvata nella stringa “appealDate”;
* appeal [3] contiene l’elenco degli studenti con relativi voti racchiusi in parentesi graffe che vengono salvati in un’unica stringa detta “allStudPassedExam”;

Dopo aver individuato il corso presente in memoria usando il metodo at applicato all’id si usa il metodo assignStudToAppealPerYear al quale vengono passate le info precedenti.

Tale funzionalità permette come dice il nome stesso di assegnare a quel determinato corso tutti gli studenti che hanno superato il relativo esame (“allStudPassedExam “) registrando anche il relativo voto.

Il metodo si conclude come di buona norma con la chiusura del file “db\_appelli.txt”

* **writeVersion**

Prototipo: void University::writeVersion(int version)

Questo metodo consente di salvare in un apposito file, “versione.txt”, il valore assunto dalla variabile “version” che altrimenti andrebbe perso qualora il device sul quale gira il programma dovesse spegnersi.

Pertanto si apre il file nelle seguenti due modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Si procede con lo scrivere il valore corrispondente alla eventuale “modalità versioning” adottata, contenuto nel parametro verison.

Il metodo si conclude con la chiusura del file “versione.txt”.

**PROGRAMMAZIONE ESAMI**

Una volta implementata la base dati è necessario pianificare le date di esame; per poter mettere in piedi una sessione di esami all’interno di un anno accademico occorre in primis definire i tre periodi: sessione invernale, estiva ed autunnale.

Tali periodi devono avere una durata di sei settimane per i primi due e di sole quattro per l’ultima.

Tale implementazione però deve andare incontro anche a quelle che sono le esigenze dei professori i quali appunto possono mostrare dei periodi di indisponibilità dei quali bisogna tener conto.

Nella predisposizione degli esami bisogna tenere in conto di alcune regole, le seguenti:

* ogni singola giornata della settimana (dal lunedì al sabato) deve essere divisa in fasce orarie ciascuna della durata di 2 ore a partire dalle 8 sino alle 20; prima di collocare un esame in uno slot orario bisogna quindi tener conto non solo della durata dell’esame in se ma anche dei relativi tempi di ingresso ed uscita; la somma delle tre voci, arrotondata per eccesso, determina il numero di fasce orarie necessarie;
* ogni corso necessita di 2 appelli nel suo semestre di appartenenza ed uno solo nei rimanenti periodi di sessione; in particolare nella sessione autunnale tutti i corsi presentano un solo appello;
* nel semestre di appartenenza le prime due settimane sono rivolte solo ai primi appelli mentre le due successive sono dedicate ai secondi appelli e agli esami relativi a corsi del semestre opposto;
* gli esami spenti ma ancora presenti nei corsi di studio per ogni periodo di sessione vengono trattati come se appartenessero ad un altro semestre pertanto possiedono un solo appello per periodo collocato nella seconda metà;
* la distanza minima tra due appelli dello stesso esame deve essere di 14 giorni mentre quella tra due dello stesso anno di uno stesso corso di studio deve essere di 2 giorni;
* esami con più versioni o raggruppati devono essere programmati negli stessi giorni e nelle stesse fasce orarie;

Oltre a quelle appena citate occorre soddisfare due ulteriori regole di “buon senso” ovvero:

* in una stessa fascia oraria non possono essere disposti esami che fanno capo ad un medesimo docente;
* non dovrebbero essere programmati esami tenuti da un professore durante i suoi periodi di indisponibilità comunicati;

Come avvenuta per la prima parte del progetto anche per questa seconda occorre infine riportare su file di output i risultati ottenuti in modo che vengano comunicati all’utente garantendo anche una permanenza nel tempo degli stessi dati affinché questi ultimi non vengano persi a causa dello spegnimento del device.

Dal momento che la predisposizione degli esami risulta essere un problema di difficoltà elevata e siccome non sempre è possibile soddisfare le diverse richieste dei professori come anche rispettare i diversi vincoli imposti allora si lascia la possibilità di poter rilassare alcuni vincoli, tra quelli sopra citati, osservando il seguente schema progressivo:

* distanza minima tra esami dello stesso corso di studio;
* distanza minima tra appelli dello stesso esame
* programmazione durante i periodi di indisponibilità;

In tal caso si comunicano per ogni sessione, all’interno di un apposito file, i vincoli non rispettati.

**DESCRIZIONE DETTAGLIATA PROGETTO – PARTE “PROGRAMMAZIONE ESAMI”**

Si riporta la spiegazione dei metodi interni alla classe “University” che consentono la definizione degli esami e delle relative sessioni:

* **readSessionAcYear**

Prototipo: void University::readSessionAcYear()

Questo metodo si occupa della lettura dei tre periodi di esame in un determinato anno accademico scritte nel

file “dateSessioni.txt”.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “;“ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “infoSessions” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”:

* “infoSessions [0]” contiene l’anno di inizio e di fine del anno accademico;
* “infoSessions [1]” contiene il primo periodo di esame ovvero quello invernale composto da data di inizio e di fine;
* “infoSessions [2]” contiene il secondo periodo di esame ovvero quello estivo composto da data di inizio e di fine;
* “infoSessions [1]” contiene il terzo periodo di esame ovvero quello autunnale composto da data di inizio e di fine;

Mediante la funzione getAcStartYear si estrae da “infoSessions [0]” solo l’anno di inizio; che viene salvato in una variabile di appoggio “acStartYear”. Si dichiara poi una stringa nella quale …………………

Infine si dichiara un oggetto di appoggio detto “sessionYear” che viene inizializzato con le informazioni appena lette dalla riga grazie al costruttore implicitamente richiamato. Tale costruttore a sua volta ………………

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Tale metodo si conclude con la chiusura del file database “dateSessioni.txt”.

* **readProfsNoAvailability**

Prototipo: void University::readProfsNoAvailability()

Questo metodo permettere di leggere il file “tutte\_le\_indisponibilita.txt” ovvero il file database contenente tutte le date di indisponibilità dei professori per i diversi anni accademici.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “profAvailability” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “ ; ”:

* “profAvailability [0]” contiene l’anno accademico;
* “profAvailability [1]” contiene la matricola del professore;
* “profAvailability [2]” contiene il primo periodo di indisponibilità del professore;
* “profAvailability [N]” contiene l’ennesimo periodo di indisponibilità del professore infatti non è specificato il numero massimo di indisponibilità che un docente può presentare;

Dai primi due campi si estraggono rispettivamente il primo anno di quello accademico, che viene salvato in “acYear”, e la matricola del docente privata del carattere iniziale ‘d’, salvata in “nMatr”.

Solo nel caso in cui il professore appena letto da file esiste in memoria si procede col salvataggio delle sue indisponibilità; in particolare, mediante un for si cicla sui rimanenti campi di “profAvailability” e per ognuno di essi (ovvero per ogni periodo di indisponibilità) si procede con il salvataggio in memoria; salvataggio che consiste nell’andare a settare il singolo periodo di indisponibilità nella mappa dedicata associata al prof appena letto, grazie al metodo setNoAvailabilities; in particolare tale mappa, “\_noAvailab”, è un dato privato della classe “Professor” che possiede come chiave l’inizio dell’anno accademico e come value la coppia di date (inizio – fine) del periodo di indisponibilità.

Quanto spiegato si ricorda essere fatto per ogni riga del file (ovvero per ogni professore scritto nel file) dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Prima di porre fine al metodo si procede con la chiusura del file, “tutte\_le\_indisponibilita.txt”, mediante il metodo close.

* **setSessionPeriod**

Prototipo: bool University::setSessionPeriod

(std::string &acYear, std::string &winterSession,

std::string &summerSession, std::string &autumnSession)

Metodo lanciato in risposta al comando: -s current\_a <anno\_accademico> “<data\_inizio>\_<data\_fine>”

“<data\_inizio>\_<data\_fine>” “<data\_inizio>\_<data\_fine>”

Questo metodo permette di salvare i tre periodi delle sessioni: invernale, estiva, autunnale. Queste ultime vengono passati come parametri del metodo insieme all’anno accademico relativo.

In particolare a partire dall’anno accademico si estrae quello di inizio mediante la funzione getAcStartYear che viene salvato poi nella variabile di appoggio “acStartYear”.

Mediante un if si controlla se in memoria sono già presenti eventuali info riguardanti le sessioni del medesimo anno e in tal caso vengono eliminate (come richiesto da consegna).

Si crea una stringa ospitante il nome del file che a sua volta viene passato dal comando per la generazione delle date di esame.

Si procede con la dichiarazione di un oggetto della classe SessionYear, detto “sessionYear”, il quale viene inizializzato con le seguenti info: anno accademico iniziale, periodi delle tre sessioni e nome del file.

* **Costruttore - classe “SessionYear”**:

SessionYear::SessionYear(std::string &acYear, std::string &winterSession, std::string &summerSession, std::string &autumnSession,

std::string &output\_file\_name) : \_sysLog(output\_file\_name)

Come ogni costruttore anche questo si occupa dell’inizializzazione dei dati privati in particolare:

* inizializza il vector di stringhe “\_sessionNames” che contiene i nomi delle tre sessioni nello specifico “winter”, “summer” ed “autumn”;
* per ogni sessione di quello specifico anno accademico aggiunge in memoria il corrispondente periodo con le due date, inizio e fine mediante il metodo addSession, sollevando anche eventuali eccezioni ove occorre;
* **addSession - classe “SessionYear”**:

bool SessionYear::addSession(std::string &acYear, std::string &sessionDates,

std::string &name)

Questo metodo per ogni sessione di un determinato anno accademico aggiunge in memoria il corrispondente periodo con le due date, inizio e fine provvedendo anche al sollevare eventuali eccezioni on caso di errore.

In particolare tale metodo salva in memoria l’anno accademico di inizio;

Usando il metodo getDates, appartenente alla classe Parse, si passa la stringa “sessionDates”(nonché parametro del metodo addSession) contenente un periodo di sessione di esame. Esso restituisce un vector di oggetti Date, all’interno del quale nel primo campo è presente la data di inizio mentre nel secondo la data di fine del periodo stesso (date entrambe espresse nel formato yy,mm,dd).

Essendo addSession un metodo generale, ovvero applicabile per ogni tipo di periodo di sessione (sia che esso sia invernale, estivo o autunnale), si ha che risulta necessario individuare il tipo di sessione sul quale esso opera e che gli viene passato come parametro nella stinga “name”; pertanto mediante un if si controlla se la stringa “name” risulta uguale a “\_sessionNames [2]”, ovvero ad “autumn”.

In caso affermativo si controlla con il metodo checkGapGiven che la distanza tra la data di inizio e di fine della sessione autunnale sia di 4 settimane, come richiesto da consegna.

In caso negativo il metodo si preoccupa di scatenare una eccezione al fine di rendere l’utente al corrente di quanto accaduto.

Nel caso in cui il periodo passato al metodo non risulti essere quello autunnale allora si procede col controllare direttamente se la distanza tra la data di inizio e di fine della sessione sia di 6 settimane, come richiesto da consegna per le due rimanenti sessioni (invernale ed estiva). In caso affermativo si controlla il tipo di sessione, se “summer” o “winter”, al fine di rendere l’utente al corrente dell’accaduto.

Solo dopo aver superato con successo tali controlli si può procedere con il caricamento vero e proprio delle info raccolte. Si dichiara una variabile“s” della struttura session che viene inizializzata con le tre seguenti informazioni: nome della sessione (“winter” o “summer” o “autumn”) e data di inizio e di fine della sessione; in questo modo si può usare il metodo insert al fine di inserire nella mappa privata “\_yearSession” (della classe SessionYear) la struct session “s” appena creata.

**La mappa “\_yearSession” in particolare è un container (della classe SessionYear) avente come:**

* **chiave “winter” o “autumn” o “summer”;**
* **value la struct session**

Infine con il metodo setCaldendar si inserisce nel calendario (nonché dato privato della classe SessionYear) i giorni delle sessioni appena elaborate, dove ogni giorno è un oggetto della classe ExamDay.

**La mappa “\_yearCalendar” in particolare è un container (della classe SessionYear) avente come:**

* **chiave la data (stringa formato “aaaa-mm—gg”);**
* **value oggetti della classe “ExamDay”**

Come mostra il prototipo il metodo si conclude con la restituzione di un booleano pari a true.

* **checkGapGIven – classe “Date”:**

bool Date::checkGapGiven(int weeks, Date d)

Questo metodo, come anticipato, si occupa di controllare se tra la distanza tra la data di inizio della sessione (nonché l’oggetto della classe Date che invoca il metodo) e la data di fine periodo (ovvero il secondo parametro, Date “d”) intercorrono il numero di settimane passato come primo parametro (alias int “weeks”).

Si converte il numero di settimane “weeks”, passato come primo parametro, in numero di giorni salvati nella variabile intera “numDays”.

Successivamente si aggiunge il numero di giorni calcolato alla data iniziale della sessione nonché alla data alla quale è applicato il metodo stesso checkGapGiven; data iniziale recuperabile mediante il this.

La data ottenuta si salva in un oggetto di appoggio della classe Date ed infine la si compara con la data di fine che viene passata come secondo parametro del metodo stesso, “d”.

La comparazione tra le due date viene fatta mediante un apposito metodo della classe Date, isEqual, il quale controlla che sia i giorni, sia i mesi che gli anni siano uguali. Solo nel caso in cui lo siano tutti e tre restituisce un booleano pari a true; quest’ultimo esito è anche il valore di ritorno del metodo checkGapGIven.

* **setCaldendar – classe “SessionYear”:**

bool SessionYear::setCaldendar(std::vector<Date> dates)

Questo metodo inserisce nel calendario dell’anno accademico i giorni presenti nel periodo di sessione che gli viene passato come parametro; quest’ultimo infatti è un vector di oggetti Date composto da due elementi dove il primo è la data di inizio mentre il secondo è la data di fine del periodo della sessione.

Mediante un for, per ogni giorno della sessione:

1. si dichiara un oggetto della classe ExamDay detto “examDay” inizializzato con la data del giorno presente nel periodo di sessione sul quale il ciclo for sta puntando;
2. viene riempita la mappa privata “\_yearCalendar”, della classe SessionYear, con la data appartenente alla sessione puntata dal for (convertita in stringa grazie al metodo toString degli oggetti della classe Date) e l’oggetto “examDay” appena creato;

CONTINUAZIONE SPIEGAZIONE METODO setSessionPeriod:

Solo dopo aver realizzato l’oggetto “sessionYear” si può procedere con il salvare in memoria le info raccolte all’interno della mappa privata presente nell’oggetto della classe “University”.

**La mappa “\_acYearSessions” in particolare è un container (della classe University) avente come:**

* **chiave l’anno di inizio accademico;**
* **value oggetti della classe “SessionYear”, in questo caso l’oggetto “sessionYear”;**

Solo in seguito si procede con il salvare in un apposito file, “datesessioni.txt”, le date dei tre periodi di esame di quello specifico anno accademico grazie al metodo dbDateSessionWrite.

In particolare “datesessioni.txt” tiene traccia delle tre sessioni per i diversi anni accademici ricordando come tali tre periodi siano unici per uno stesso anno accademico e questo proprio grazie all’if iniziale presente nel metodo che si sta spiegando, setSessionPeriod.

* **setProfsNoAvailability**

Prototipo: bool University::setProfsNoAvailability

(std::string acYear, const std::string &fin)

Metodo lanciato in risposta al comando: -s set\_availability <anno\_accademico> <file\_indisponibilità>.

Questo metodo si occupa dell’inserimento di nuovi periodi di indisponibilità nella mappa in memoria, in particolare costituiranno delle nuove indisponibilità per i prof che non ne posseggono una mentre si andranno a sostituire a quelle già esistenti per i docenti che già ne posseggono una o più. Tali indisponibilità sono riportati nel file il cui nome viene passato nel parametro “fileIn” del metodo stesso, insieme all’anno accademico a cui fanno riferimento.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso (restituendo un booleano pari a false), grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “profAvailability”.

Il contenuto di tale vector come anche la struttura del metodo sono analoghe a quelle viste nel metodo readProfsNoAvailability, per completezza vengono riportate di seguito.

Ogni elemento del vector contiene l’info presente nella riga compresa tra due caratteri separatori “ ; ”:

* “profAvailability [0]” contiene l’anno accademico;
* “profAvailability [1]” contiene la matricola del professore;
* “profAvailability [2]” contiene il primo periodo di indisponibilità del professore;
* “profAvailability [N]” contiene l’ennesimo periodo di indisponibilità del professore infatti non è specificato il numero massimo di indisponibilità che un docente può presentare;

Dal secondo campo si estrae la matricola del docente privata del carattere iniziale ‘d’, salvata in “nMatr”.

Diversamente l’anno accademico iniziale viene estratto dal parametro passato al metodo per poi essere salvato nella variabile di appoggio “year”.

Solo nel caso in cui il professore appena letto da file esiste in memoria si procede col salvataggio delle sue indisponibilità; in particolare, mediante un for si cicla sui rimanenti campi di “profAvailability” e per ognuno di essi (ovvero per ogni periodo di indisponibilità) si procede con il salvataggio in memoria; salvataggio che consiste nell’andare a settare il singolo periodo di indisponibilità nella mappa dedicata associata al prof appena letto, grazie al metodo setNoAvailabilities; in particolare tale mappa, “\_noAvailab”, è un dato privato della classe “Professor” che possiede come chiave l’inizio dell’anno accademico e come value la coppia di date (inizio – fine) del periodo di indisponibilità.

Quanto spiegato si ricorda essere fatto per ogni riga del file (ovvero per ogni professore scritto nel file) dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Prima di porre fine al metodo si procede con:

* la chiusura del file mediante il metodo close;
* la riscrittura del file database che contiene tutte le indisponibilità (“tutte\_le\_indisponibilita.txt”) dal momento che ne sono state aggiunte delle nuove le quali andrebbero perse qualora il device dovesse essere spento;
* **allProfsNoAvailabilities**

Prototipo: std::vector<std::string> University::allProfsNoAvailabilities()

Tale metodo restituisce un vector contente in ogni suo campo tutti i periodi di indisponibilità di uno stesso professore e questo per ogni docente che ha comunicato un intervallo di giorni in cui non è disponibile.

Tale metodo quindi nasce con lo scopo di avere, al momento del bisogno, tutte le indisponibilità di tutti ii prof “a portata di mano” ovvero affinché siano facilmente reperibili.

A tale scopo si dichiara un vector di stringhe dove ogni suo campo sarà destinato al termine di tale metodo a contenere tutti i periodi di assenza del professore.

Pertanto mediante un ciclo for si scandaglia l’intera mappa dei professori presenti già in memoria e per ognuno di essi ci si avvale del metodo outputNoAvailabilities che ritorna tutte le indisponibilità per un dato professore; queste via via (grazie ad un ulteriore for annidato nel primo) vengono inserite in coda al vector “allProfAvailabilities” cosicché al termine del for esterno nel vector siano presenti per ogni campo tutti i periodi di indisponibilità di un certo professore rispettando il medesimo formato osservato nel file:

anno accademico (inizio – fine); matricola docente; periodi di indisponibilità (aaaa–mm–gg | aaaa–mm–gg);

Il metodo si conclude col ritorno del vector “allProfAvailabilities”.

* **dbDateSessionWrite**

Prototipo: void University::dbDateSessionsWrite()

Tale metodo si occupa della scrittura delle date delle sessioni sul file “dateSessioni.txt”; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for, per ogni anno accademico della mappa “\_acYearSessions” si eseguono le due seguenti operazioni:

* si dichiara un oggetto della classe SessionYear, detto “dateSessions”, inizializzato con le info del value della mappa “\_acYearSessions” corrispondente a quel determinato anno;
* si scrive sul file alcune delle info contenute nell’oggetto “dateSession”, come l’anno accademico e le date (inizio e fine) dei tre periodi di sessione nel seguente formato:

aaaa-aaaa ; yyyy-mm-dd\_yyyy-mm-dd ; yyyy-mm-dd\_yyyy-mm-dd ; yyyy-mm-dd\_yyyy-mm-dd .

Questo è possibile grazie all’overload dell’operatore “<<”, fatto per gli oggetti della classe SesionYear”.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **dbNoAvailabilityWrite**

Prototipo: void University::dbNoAvailabilityWrite()

Questo metodo si occupa della scrittura del file database, “tutte\_indisponibilita.txt”, destinato a contenere tutte le indisponibilità presenti in memoria le quali altrimenti andrebbe perse qualora il device sul quale gira il programma dovesse spegnersi.

Come ogni altro metodo volto alla scrittura di un file anche in questo caso si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database di nome in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Questo è uno dei casi in cui è possibile apprezzare l‘utilità del metodo precedentemente esposto in quanto, dopo aver aperto il file “tutte\_indisponibilita.txt”, si salvano in un vector di appoggio (“allprofsAvailabilities“) mediante il metodo allProfsNoAvailabilities tutte le indisponibilità di ciascun docente che ne ha comunicato almeno una.

Mediante l’ausilio di un for e di un indice che punta ai diversi campi del vector si procede con lo scandagliare tutte le indisponibilità ovvero tutto la variabile “allprofsAvailabilities“. Per ogni ciclo for si scrive l’i-esimo campo del vector nonché la stringa composta da:

anno accademico (inizio – fine); matricola docente; periodi di indisponibilità (aaaa–mm–gg | aaaa–mm–gg).

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **setExamDate**

Prototipo: bool University::setExamDate(std::string acYear,

std::string outputNameFile)

Metodo lanciato in risposta al comando: -g <anno\_accademico> <output\_file>.

Questo metodo organizza gli esami di un determinato anno accademico; gli viene passato come primo parametro la stringa “acYear” contenente l’anno accademico mentre come secondo parametro la stringa “outputNameFile” con all’interno il nome della base del file che ospiterà le date per le tre sessioni.

Si inizia con l’estrarre l’anno di inizio dalla stringa “acYear” e salvarlo nella variabile intera “startAcYear”.

Con l’ausilio del metodo controlDatabase, come spiegato abbondantemente nelle pagine precedenti, si controlla che i file siano coerenti all’anno accademico che gli viene passato come parametro intero (“startAcYear”).

A questo punto all’interno di un while si accede alla mappa privata “\_acYearSession” in corrispondenza dell’anno accademico iniziale, “startAcYear”, e si chiama il metodo generateNewYearSession il quale genera le sessioni d’esame per quello specifico anno accademico. Tale metodo infatti necessita delle seguenti info che gli vengono passati come parametri:

* la stringa “outputNameFile” con all’interno il nome della base del file che ospiterà le date per le tre sessioni;
* tutti i corsi che vengono insegnati all’interno dell’università e che sono già presenti in memoria nella **mappa privata “\_courses” (chiave: id del corso, value: oggetto della classe Course) della classe University;**
* tutti i professori che insegnano all’interno dell’università e che sono già presenti in memoria nella

**mappa privata “\_professors” (chiave: id del docente, value: oggetto della classe Professor) della classe University;**

* tutte le aule che fanno parte dell’università e che sono già presenti in memoria nella

**mappa privata “\_classroom” (chiave: id dell’aula, value: oggetto della classe Classroom) dell classe University;**

* variabile contatore “constraintRelaxParameter” interna al metodo che tiene conto del numero di vincoli che vengono rilassati ad ogni ciclo;

Se la definizione della sessione di esami va a buon fine il metodo generateNewYearSession restituisce coerentemente un booleano di valore true che viene salvato in una variabile flag detta “esito”.

Si noti come ogni volta prima che si concluda un ciclo si incrementi la variabile contatore “constraintRelaxParameter” cosicché nel successivo ciclo di while all’atto della definizione della sessione di esami (mediante il metodo generateNewYearSession) si tenga conto che nel precedente ciclo è stato rilassato un vincolo;

Si ricorda che il while procede fino a quando si verificano entrambe le seguenti 2 condizioni:

* “esito” è false ovvero fino a quando la definizione della sessione di esami del ciclo precedente non è andata a buon fine per tutti i corsi;
* “constraintRelaxParameter” è minore di 4 ovvero se ci sono ancora vincoli che possono essere rilassati permettendo al ciclo dopo la definizione della sessione di esami;

Quindi il metodo generateNewYearSession se riesce a definire la sessione senza rilassare vincoli restituisce true (“esito”=true) e quindi si esce dal while al primo ciclo; diversamente “constraintRelaxParameter” viene incrementato di una unità e al prossimo ciclo vale 1 quindi il metodo tiene conto che è stato rilassato il primo vincolo (ovvero “distanza minima tra esami dello stesso corso di studi”), in quanto i vincoli hanno un ordine progressivo di rilassamento come da consegna, e definisce la sessione di esami sulla base di questa consapevolezza; se continua a non andare a buon fine richiedendo il rilassamento di un ulteriore vincolo, il secondo (ovvero “distanza minima tra appelli dello stesso esame”), allora si incrementa di uno “constraintRelaxParameter” che arrivando a 2 permette di definire la sessione al prossimo ciclo sulla base del secondo rilassamento;

nel caso in cui il rilassamento del terzo vincolo (ovvero “programmazione durante i periodi di indisponibilità”) non fosse sufficiente e di aiuto alla definizione della sessione di tutti i corsi allora la variabile “esito” rimane false e “constraintRelaxParameter“ diventa 4 permettendo l’uscita dal while.

Si precisa che gli esami già piazzati però non verranno di nuovo piazzati, costoro infatti rimangono inseriti e ad ogni vincolo rilassato si prova ad inserire gli esami restanti comunicando nei warnings, come da consegna, se un esame venisse assegnato grazie ad un vincolo rilassato.

Fuori dal while si controlla il valore registrato dal flag “esito” e si solleva una eccezione nel caso in cui sia false comunicando all’utente l’impossibilità nel generare le date degli esami nonostante il rilassamento dei tre vincoli.

* **generateNewYearSession – classe “SessionYear”:**

bool SessionYear::generateNewYearSession(std::string &fout, std::map<std::string, Course> &courses,

std::map<int, Professor> &professors,

std::map<int, Classroom> &allUniversityClassrooms, int relaxPar)

Questo metodo genera le sessioni d’esame per un qualsiasi anno accademico.

In particolare tale metodo, mediante delle apposite variabili, definisce i diversi contesti nei quali ci si può trovare nella definizione delle sessioni di esami, discernendo le diverse situazioni che si possono venire a creare, e andando a rilassare, quando necessario, in modo progressivo e ordinato i diversi vincoli fino a quando non si creano i presupposti che consentono di definire tutti gli esami.

Pertanto tale metodo gestisce principalmente le variabili che definiscono i vari scenari in quanto per la realizzazione vera e propria della sessione e della relativa collocazione degli esami si avvale di un ulteriore metodo usato al suo interno, detto generateThisSession. Quest’ultimo a seconda del contesto che gli viene passato attraverso i parametri determina se è possibile o meno definire la sessione di esame e in caso affermativo colloca gli esami.

Mediante un for si va a rilassare progressivamente il secondo vincolo (ovvero la distanza tra gli appelli dello stesso esame); infatti ad ogni ciclo di for si procede con il diminuire di una unità la variabile, “gapAppealsSameCourse”, che tiene traccia della distanza in giorni tra due appelli dello stesso esame;

il primo ciclo prevede l’assenza del rilassamento del secondo vincolo in quanto la variabile viene inizializzata con il valore 14, solo a partire dal secondo ciclo si inizia con il rilassamento.

Si distinguono i diversi scenari descritti dal for:

* al primo ciclo “gapAppealsSameCourse” vale 14 (il che vuol dire che il secondo vincolo non è stao ancora rilassato) quindi si controlla il valore assunto dalla variabile booleana “winter” e se pari a false (ovvero se la sessione invernale non è stata ancora definita) si procede col chiamare il metodo generateThisSesion che prova a definire la sessione in base al contesto che gli viene passato come parametro nonché: il tipo di sessione di cui si tratta (in questo caso invernale), tutti i corsi, tutti i professori, tutte le aule, il numero di vincoli rilassati, la distanza tra appelli dello stesso esame, la variabile che tiene traccia del dover sistemare o meno nello stesso giorno gli appelli dello stesso esame a 6 ore di distanza. Tale metodo se andato a buon fine restituisce un booleano pari a true mentre se la sessione invernale non è stata definita allora restituisce un valore pari a false cosicché al prossimo ciclo di for possa essere ridefinita con parametri diversi. Analogamente si procede con la sessione estiva ed autunnale avvalendosi del medesimo metodo.

Solo nel caso in cui tutte e tre sessioni le sessioni sono state definite (ovvero solo nel caso in cui sia “winter” che “autumn” e “summer” sono true) si procede con il generare e lo scrivere i tre file destinati a contenere le date delle tre sessioni mediante il metodo generateOutputFiles; usando invece il metodo allExamAppealsWrite si scrivono tutti gli appelli su un file database al fine d tenerne traccia in caso di spegnimento del device; infine con l’aiuto del metodo writeWarnigs si comunica all’utente come da consegna su un apposito file gli eventuali vincoli rilassati. Vengono così settate al valore true sia la variabile “exitloop”, al fine di uscire dal for, che la variabile result, affinchè il metodo generateNewYearSessionrestituisca true rendendo il metodo chiamante al corrente del fatto che le tre sessioni sono state definite correttamente.

Nel caso in cui tutte e tre le sessioni non sono state definite allora “result” viene settato a false affinché si possa rimanere nel ciclo for ovvero affinché si possa provare a definire le sessioni rimanenti con un nuovo scenario: iniziare a rilassare il secondo vincolo diminuendo il numero di giorni tra due appelli dello stesso esame a 13.

Prima di passare al nuovo ciclo for (ovvero a rilassare il secondo vincolo) si fa attenzione al controllare il valore del parametro “relaxPar” in quanto qualora quest’ultimo valesse 0 (nessun vincolo rilassato) o 1 (solo il primo vincolo rilassato) si ha che non si può procedere col rilassare anche il secondo perché lo scioglimento dei vincoli deve seguire uno schema progressivo come da consegna; motivo per cui nel caso in cui “relaxPar” sia minore di 2, si procede col settare e true la variabile che fa uscire dal for, “exitloop”, impedendo il rilassamento del secondo vincolo e concludendo il metodo con la restituzione del valore false.

* se giunti al secondo ciclo di for significa che i vincoli precedenti sono stati rilassati e che non tutte e tre le sessioni sono state definite motivo per cui si procede col rilassamento del secondo; pertanto si decrementa di una unità la variabile “gapAppealsSameCourse” permettendo che i due appelli degli stessi corsi possano essere collocati anche a 13 giorni di distanza l’uno dall’altro: “gapAppealsSameCourse” = 13. Solo per le sessioni che non state definite nei cicli precedenti, ovvero per le sessioni i cui corrispondenti booleani valgono false (“winte” o “summer” o “autumn”) ) si procede col chiamare il metodo generateThisSesion che prova a definire la sessione in base al contesto che gli viene passato come parametro contesto che risulta diverso da quello del ciclo for precedente in quanto “gapAppealsSameCourse “ vale adesso 13.

Tale metodo se andato a buon fine restituisce un booleano pari a true mentre se la sessione non è stata definita allora restituisce un valore pari a false cosicché al prossimo ciclo di for possa essere ridefinita con parametri diversi.

Solo nel caso in cui tutte e tre sessioni le sessioni sono state definite (ovvero solo nel caso in cui sia “winter” che “autumn” e “summer” sono true) si procede con il generare e lo scrivere i tre file destinati a contenere le date delle tre sessioni mediante il metodo generateOutputFiles; usando invece il metodo allExamAppealsWrite si scrivono tutti gli appelli su un file database al fine d tenerne traccia in caso di spegnimento del device; infine con l’aiuto del metodo writeWarnigs si comunica all’utente come da consegna su un apposito file gli eventuali vincoli rilassati. Vengono così settate al valore true sia la variabile “exitloop”, al fine di uscire dal for, che la variabile result, affinchè il metodo generateNewYearSessionrestituisca true rendendo il metodo chiamante al corrente del fatto che le tre sessioni sono state definite correttamente.

Nel caso in cui tutte e tre le sessioni non sono state definite allora “result” viene settato a false affinché si possa rimanere nel ciclo for ovvero affinché si possa provare a definire le sessioni rimanenti con un nuovo scenario: continuare a rilassare il secondo vincolo diminuendo di una unità il numero di giorni tra due appelli dello stesso esame.

Risulta inoltre essere chiaro come a partire da questo secondo ciclo di for in poi non si possa entrare nell’if relativo a “relaxPar”.

Risulta essere chiaro che per i cicli successivi aventi “gapAppealsSameCourse” compreso tra 2 e 13 si procede analogamente a questo appena spiegato.

* se si giunge al ciclo for in cui “gapAppealsSameCourse“ sia pari a 0 allora viene settato al valore true il flag booleano “sameDayTwoAppealsSameExamAtLeastSixHours” al fine che si tenga al corrente il metodo generateThisSession del fatto che il secondo vincolo è stato rilassato sino al punto che si possono collocare gli appelli di uno stesso esame nello stesso giorno, purché questi ultimi siano ad una distanza di almeno 6 ore l’uno dall’altro.

Da questo punto in poi si procede come negli altri casi ovvero chiamando il metodo generateThisSession per quelle sessioni che non sono state definite del tutto.

Pertanto solo nel caso in cui tutte e tre le sessioni sono state definite allora si procede con il generare e scrivere i tre file contenenti le date delle tre sessioni e con lo scrivere il database di tutti gli appelli come anche il file relativo ai warnings;

Inoltre se tutte e tre le sessioni sono state definite il metodo restituisce un booleano pari a ture diversamente il metodo restituisce false rendendo il metodo chiamante che la generazione dei periodi di esame non è andata a buon fine;

* **generateThisSession – classe “SessionYear”:**

bool SessionYear::generateThisSession(std::string sessName, std::map<std::string, Course> &courses, std::map<int, Professor> &profs, std::map<int, Classroom> &allUniversityClassrooms, int relaxPar, int gapAppeals, bool sixHours)

Tale metodo, come anticipato più volte in precedenza, in base allo scenario che gli viene passato mediante parametri si occupa della realizzazione vera e propria della sessione e della relativa collocazione degli esami restituendo un booleano a seconda se la realizzazione è andata a buon fine per tutti i corsi (true) o meno (false). In particolare come si può notare dal prototipo gli vengono passati come parametri:

* stringa “sesName” contente il tipo di sessione di cui si tratta (o “winter” o “summer” o “autumn”);
* la mappa “courses” contenente tutti i corsi presenti nell’università;
* la mappa “profs” contenente tutti i professori che insegnano nell’università;
* la mappa “allUniversityClassrooms” contenente tutte le aule presenti nell’università;
* l’intero “relaxPar” rappresentante il numero di vincoli rilassati;
* l’intero “gapAppelas” che indica la distanza in giorni tra appelli dello stesso esame;
* il booleano “sixHours” che quando vale true indica il fatto che il secondo vincolo è stato rilassato sino al punto che si possono collocare gli appelli di uno stesso esame nello stesso giorno, purché questi ultimi siano ad una distanza di almeno 6 ore l’uno dall’altro;

Si prende l’intervallo di date della sessione in questione di quel determinato anno accademico e la si salva in un oggetto, detto “thisSession”, della classe session; da tale oggetto si estrae la data d’inizio e di fine della sessione che vengono salvate rispettivamente nei due seguenti oggetti della classe Date: “startDate” ed “endDate”.

Si inizia col controllare se fino ad ora sono stati rilassati vincoli ovvero se è la prima volta che si cerca di definire quella determinata sessione per quel determinato anno; in tal caso si procede col caricare nel vector privato “\_allExamAppealsToDo” della classe SessionYear tutti gli appelli di tutti i corsi che bisogna effettuare in quell’anno di quella specifica sessione (o invernale o estiva o autunnale);

diversamente se relaxPar è diverso da 0 vuol dire che tale vector, “\_allExamAppealsToDo”, è già stato caricato in precedenza;

Nella corrente spiegazione, al fine di migliorare la leggibilità e la collocazione dei diversi concetti nel codice, si introduce l’indentazione riportata di seguito.

Mediante un for esterno si cicla su ogni data della sessione in questione (o invernale o estiva o autunnale) cosicché ad ogni ciclo di for si passi da un giorno al suo successivo sino al termine del periodo della sessione. Il for pertanto si avvale come indice di un oggetto della classe Date, detto “currentExamDay”, inizializzato con la data corrispondente al primo giorno della sessione (questo grazie al costruttore che riceve come parametro anno, mese e giorno estratti mediante appositi metodi dalla data iniziale).

Si escludono dalle istruzioni avvenire tutte le date che sono pari a “Sunday” in quanto non è possibile collocare nessun esame la domenica (questo grazie ad un if subito dopo il for esterno).

Per ogni data della sessione si procede con il secondo for, interno al primo, che cicla invece su tutti gli appelli da assegnare; tale for infatti usa un indice che parte da 0 sino al numero di elementi presenti nel vector privato “\_allExamAppealsToDo”.

Quando uno o più esami vengono collocati nel calendario della sessione si ha che essi vengono tolti dal vector “\_allExamAppealsToDo”, in quanto quest’ultimo contiene solo quelli da fissare; di conseguenza i rimanenti esami presenti in tale vector scalano di posizione quindi il secondo for al prossimo ciclo si perderebbe alcuni esami (ovvero gli esami che sono stati scalati non verrebbero puntati dal for, il quale indicherebbe gli esami successivi); per ovviare a questo problema si ha che quando degli esami vengono tolti da “\_allExamAppealsToDo” (perché collocati) si setta a true il flag “pop” cosicché al prossimo ciclo si entri nell’if fatto sul flag “pop”, il quale forzando “indexExam” a 0 assicura che il for riparta dalla prima posizione del vector “\_allExamAppealsToDo”;

per completezza si ripristina il flag “pop” al valore false lasciando che diventi true solo alla prossima collocazione di uno o più esami.

Per ogni data della sessione e per ogni appello si procede come segue.

Si salva nella stringa “codCurrentAppeal” l’appello che sta puntando il for interno e in seguito si salva in un apposito vector di stringhe, “coursesGrouped”, tutti i corsi raggruppati a quello contenuto in “codCurrentAppeal“ grazie al metodo getGroupedCourses. Con l’aiuto di un ulteriore for si procede col salvare nello stesso ordine in un vector di oggetti della classe Course, “coursesToConsiderInThisLoop”, tutti i corsi i cui id sono presenti in “coursesGrouped”.

Siccome esistono alcuni esami che presentano due appelli allora questi compariranno due volte nel vector “\_allExamAppealsToDo” mentre una volta solo nei due vector “CoursesToConsiderInThisLoop” e “coursesGrouped”.

I controlli riportati di seguito si ricordano essere fatti solo per la sessione invernale ed estiva in quanto in quella autunnale ogni esame sia esso attivo o spento, del semestre di appartenenza o meno, necessita di un solo appello (questo giustifica l’if effettuato su “sessName” che deve essere diverso dalla stringa “autumn”).

Si usa un terzo for, annidato nel secondo, perché occorre verificare che la data puntata dal primo for sia accettabile per tutti gli esami da inserire in questo giro ovvero per tutti gli esami presenti nel vector “coursesToConsiderInThisLoop” che sono quelli raggruppati a quello puntato dal secondo for.

In particolare mediante tale for si controlla che ogni corso raggruppato contenuto in “CoursesToConsiderInThisLoop” rispetti tutti i seguenti vincoli:

* se la data in questione deve ospitare il primo appello di un corso attivo e del semestre di appartenenza allora si controlla che tale primo appello cada nella prima metà della sessione, ovvero nella 1a o 2a settimana;
* se la data in questione deve ospitare il secondo appello di un corso attivo e del semestre di appartenenza allora si controlla che il suo secondo appello cada nella seconda metà della sessione, ovvero nella 3a o 4a settimana;
* se la data in questione deve ospitare l’unico appello di un corso spento e/o di un semestre opposto allora si controlla che tale appello cada nella seconda metà della sessione, ovvero nella 3a o 4a settimana;

Tale controllo viene fatto grazie al metodo **dateIsOK** al quale viene passato un corso raggruppato alla volta; solo nel caso in cui vengono soddisfatti tutti i precedenti vincoli allora il metodo dateIsOK restituisce un booleano pari a true, diversamente ritorna un booleano false, valori che vengono salvati nell’omonimo flag “dateIsOK”.

Osservando la condizione del for si deduce che:

* non appena un corso raggruppato non soddisfa tutti i requisiti allora viene settato a false il flag “dateIsOK” provocando l’uscita dal for al ciclo successivo;
* se un corso raggruppato soddisfa tutti i vincoli allora il flag “dateIsOK” viene settato a true consentendo il controllo del successivo corso raggruppato;

Alla fine di tale for quindi:

* se il flag “dateIsOk” vale false significa che almeno un corso raggruppato non soddisfa i vincoli motivo per cui non ha senso procedere con ulteriori istruzioni e controlli ovvero si passa al ciclo successivo del secondo for nonché al prossimo esame da piazzare;
* se vale true è perché tutti i corsi raggruppati hanno soddisfatto i vincoli imposti pertanto si può procede con gli ulteriori controlli riguardanti i prof e le aule perché tali corsi sono potenzialmente collocabili nella data puntata dal primo for.

Se tutti i corsi raggruppati hanno soddisfatto i vincoli imposti allora si procede con il controllare se i prof sono disponibili e se nei due giorni precedenti non ci siano corsi dello stesso corso di studio e semestre.

Siccome tali controlli riguardano tutti i corsi raggruppati presenti in “coursesToConsiderInThisLoop” allora si fa un for rivolto a tale vector dove l’indice pertanto spazia dal primo all’ultimo elemento; per ogni corso raggruppato si eseguono le istruzioni riportate di seguito:

* Si salva in un oggetto d’appoggio detto “courseToConsider” il corso che si sta analizzando che corrisponde a quello puntato dal for nel vector “coursesToConsiderInThisLoop”; questo serve affinché a partire da “courseToConsider” si recuperi mediante l’apposito metodo getThisYearCourse il medesimo corso però con le specifiche informazioni dell’anno in analisi; questo viene salvato nell’oggetto “sp” della classe SpecificYearCourse.
* Risulta utile recuperare, se già definita in precedenza, la data del primo appello sempre del medesimo esame e questo è possibile grazie a metodo lastDateAssignationInGivenSession; quest’ultimo noto il semestre della sessione, passato come parametro, restituisce un oggetto Date contenente la data del primo appello, se questa esiste, oppure una data di default, qualora la data del primo appello non è stata ancora assegnata. Tale data viene salvata nell’oggetto “lastDateAssignation”.
* Successivamente si controlla se la data in analisi che si vuole riempire (ovvero quella puntata dal primo for) coincide con la data del primo appello il che vuol dire che il secondo vincolo è stato rilassato sino al punto che la distanza tra appelli dello stesso esame è pari a zero; pertanto solo se il flag “sixHours” è true posso procedere con due ulteriori controlli:
* se la data del primo appello non è stata ancora assegnata nel giorno che si sta analizzando (ovvero se “lastDateAssignation“ contiene la data di default) allora devo procedere con i controlli delle righe avvenire;
* se la data del primo appello è stata già assegnata al giorno in analisi (ovvero quello puntato dal primo for) allora ci si avvale del metodo **getEndHourOfThisCourseExam** il quale restituisce l’ora finale dell’utimo slot ospitante il primo appello; informazione che viene salvata nella variabile intera “endHourSlot”.
* In seguito si controlla che i docenti siano disponibili (ovvero che la data NON cada nel periodo di indisponibilità o che il docente NON sia occupato con un altro esame) e che nei due giorni precedenti non siano presenti esami già assegnati con stesso corso di studio e stesso anno dell’esame da assegnare. A tal fine si usa il metodo checkIfProfsAvailableAndGapSameSemesterCourses al quale viene passato come parametro:
* “courseToConsider”, il corso che si sta analizzando nonché quello puntato dal for;
* “currentExamDay” ovvero la data dell’esame puntata dal primo for;
* mappa “profs” contenente tutti i professori che insegnano nell’università
* mappa “allUniversityClassrooms” contenente tutte le aule presenti nell’università;
* “relaxPar” rappresentante il numero di vincoli rilassati;
* Il semestre corrispondente alla sessione in questione;
* “roomsFounded” che viene passato vyoto al metodo che ma che sarà destinato a contenere le aule disponibili per quell’esame;
* “endHourSlot” contiene l’ora finale dell’utimo slot ospitante il primo appello;

Quindi nella variabile “startExamHour” si salva il valore restituito da tale metodo che è pari o all’eventuale ora di inizio dello slot destinato ad ospitare l’esame oppure al valore -1 qualora non sono stati trovati slot consecutivi ad accogliere l’esame o se la data cade in un periodo di indisponibilità di un prof. Tale valore viene salvato in coda al vector “startHourPerCourse” il quale tiene traccia dell’eventuale ora di inizio per i diversi corsi presenti nel vector “coursesToConsiderInThisLoop”.

* Dopo si controlla la disponibilità delle aule e che venga rispettato il vincolo delle 6 ore nel caso in cui il secondo cade nello medesimo giorno.

If …

Else ….

Dal ripulire il vettore di aule fino alla fine del metodo

* **getAllExamAppealsToDo – classe “SessionYear”:**

SessionYear::getAllExamAppealsToDo(std::string sessName, std::map<std::string, Course> &courses)

Questo metodo, estrae tutti gli appelli da fare inerenti alla sessione (o “winter” o “summer” o “autumn”) e ai corsi che gli vengono passati come parametro.

Si crea un vector di stringhe detto “allExamAppealToDo” e si ricava il numero del semestre corrispondente alla sessione di esami, ovvero: “1” nel caso di session invernale, “2” nel caso di sessione estiva, “3” nel caso di quella autunnale.

A questo punto si analizza ogni corso mediante un for e per ognuno di essi si eseguono le istruzioni avvenire:

* si recupera il corso con le info specifiche dell’anno in questione e viene salvato in un oggetto di appoggio, “specificYY”, della classe SpecificYearCourse;
* si recupera il semestre e il booleano rappresentante il fatto di essere attivo o meno salvati rispettivamente in “semester” e “isActive”;
* nel caso in cui si tratti di un corso spento si recupera anche il primo anno a partire dal quale è diventato spento salvandolo nella variabile intera “acYearOff”;
* si controlla se tale corso è attivo e se tale sessione cade nello stesso semestre di appartenenza del corso; in tal caso l’esame richiede due appelli pertanto vengono accodate nel vector “allExamAppealToDo “due testimonianze di tale corso, ovvero viene salvato in coda per due volte consecutive il codice identificativo del corso;
* si controlla se tale sessione cade nello stesso semestre di appartenenza del corso e se la data per la quale si vuole definire la sessione è precedente all’anno in cui il corso è stato spento; in tal caso allora il corso il corso risulta ancora attivo nel periodo per il quale si chiede di definire gli esami pertanto necessita di due appelli; come nel caso precedente si provvede ad accodare due testimonianze di tale corso nel vector “allExamAppealToDo”
* in tutti gli altri casi (come ad esempio: corso spento, o non appartenente al semestre in questione o entrambi) si dispone per tale esame un solo appello motivo per cui si accoda nel vector “allExamAppealToDo” una sua sola testimonianza;

Il metodo si conclude con la restituzione del vector “allExamAppealToDo” ovvero di tutti gli appelli da effettuare per tutti i corsi per la sessione indicata in “sessName”.

* **dateIsOK – classe “SessionYear”:**

bool SessionYear::dateIsOK(Date &newDate, const Course &course,

std::string &sessName, int gapAppeals)

Tale metodo controlla se la data che gli viene passata come parametro è accettabile o meno per l’esame di quel determinato corso passato come parametro sapendo che:

* nella 1a e 2a settimana vengono collocati gli esami di corsi attivi e del semestre di appartenenza
* nella 3a e 4a settimana vengono collocati gli esami di corsi non attivi o non appartenenti al semestre corrente o entrambi;

Si passano al metodo i seguenti parametri:

* “newDate” nonché l’oggetto data da controllare;
* “course” nonché l’oggetto corso del quale si vuole piazzare l’esame;
* “sessName” ovvero il nome della sessione (o invernale o estiva o autunnale);
* “gapAppeals” ovvero la distanza in giorni concessa tra i due appelli del medesimo esame;

Si recupera il corso specifico di quell’anno e viene salvato in un oggetto d’appoggio detto “sp” della classe SpecificYearCourse; in seguito si salvano le matricole di tutti i docenti che insegnano quello specifico corso, in un vector detto “allProfsMatrThisCourse”.

Infine si ricava il codice identificativo del corso che viene salvato in una stringa detta “idCorso”.

Si usa un flag booleano “iCanBeAssigneToFirstTwoWeekOfExamSession” il cui valore è determinato dal bool restituito dal metodo **canIBeAssigneToFirstTwoWeekOfExamSession**; quest’ultimo controlla se il corso in analisi è dello stesso semestre della data che gli viene passata come parametro al fine di capire quanti appelli dedicare in quella sessione a tale corso. In particolare:

* iCanBeAssigneToFirstTwoWeekOfExamSession vale true se il corso è attivo e appartiene allo stesso semestre della data usata come parametro; in tal caso occorre riservare due appelli a tale corso: il 1o appello in una delle due settimane mentre il 2o appello nelle due successive;
* iCanBeAssigneToFirstTwoWeekOfExamSession vale false se il corso è spento o non appartiene allo stesso semestre della data che si sta considerando; pertanto tale corso necessita di un solo appello che andrà collocato in una delle due seconde settimane;

Se il corso è spento o non del semestre di appartenenza (ovvero se il flag vale false) si controlla se la data passata come parametro al metodo,” newDate”, appartenga alle prime due settimane perché se così fosse, per quanto spiegato prima, tale esame in analisi non potrebbe essere collocato nella seguente data; in codesta situazione occorre quindi uscire dal metodo il quale deve restituire un bool pari a false al fine di comunicare al metodo chiamante l’impossibilità nel collocare l’esame “course “ nella data “newDate”. Quanto appena spiegato è il riflesso delle istruzioni presenti nell’if applicato al flag all’interno del quale si controlla che la distanza tra “newDate” e la data di inizio della sessione, “startDate”, sia minore di 14 giorni provvedendo in tal caso a restituire un bool pari a false.

A questo punto possono presentarsi tre casi ovvero la data che si sta analizzando può ospitare:

* un primo appello di un corso attivo e del semestre di appartenenza pertanto bisogna controllare che la data passata rientri nelle prime due settimane;
* un secondo appello pertanto occorrerà controllare che la data si trovi almeno 14 giorni dopo la data di inizio sessione
* un esame spento e/o esame di un semestre opposto che sono già stati controllati nell’if precednete pertanto se giunti a questo punto soddisfano il vincolo imposto di trovarsi nella seconda metà della sessione diversamente il metodo si sarebbe già concluso con la restituzione di un booleano false;

Pertanto le istruzioni avvenire riguardano prettamente corsi attivi e del semestre di appartenenza; da qui in poi risulta importante capire se si tratta o meno di un secondo appello e quindi si usa l’apposito metodo amIAssignedAlreadyInThisSession che restituisce un intero rappresentante il numero di appelli già fissati per quel determinato corso in quella sessione. Tale informazione viene salvata in una apposita variabile, “howManyTimesIAmAssignedAlreadyInThisSession”. Si controlla il valore assunto da quest’ultima variabile:

* Se vale 0, ovvero se non è stato già fissato alcun appello in merito a tale corso, allora quello che si vuole colllocare è appunto il primo appello; pertanto occorre controllare che la data in questione cada nella prima metà della sessione; pertanto se il gap tra la data in questione e la data di inizio della sessione è maggiore di 14 giorni, ovvero cade nella seconda metà della sessione, allora il metodo restituisce false.
* Se vale 1, ovvero se è già stato fissato un appello, allora quello che si vuole collocare è il secondo. Si procede in questo caso con l’eseguire le seguenti istruzioni.
* Si recupera la data del primo appello mediante il metodo lastDateAssignationInGivenSessione e la si salva nell’oggetto Date, “lastDateAssignation”;
* Si controlla coll’apposito metodo, whatIsTheGap, se la distanza in numero di giorni tra la data del primo appello, “lastDateAssignation”, e la data in analisi passata al metodo, “newDate”, sia inferiore o meno alla distanza consentita, gapAppelas; quest’ultima variabile passata come parametro al metodo dateIsOk può valere 14 giorni come può valere anche un numero inferiore a seconda se il secondo vincolo è stato rilassato o meno. In ogni caso se la distanza è inferiore a quella consentita fino ad ora (gapAppelas) allora si procede col restituire un false; se tale controllo dovesse andare a buon fine si procede con il successivo;
* Si ricorda infine che un docente può richiedere un gap tra i due appelli maggiore dei 14 giorni quindi per ogni professore che insegna tale corso si controlla che il gap da lui richiesto, “requiredGap”, sia maggiore del gap presente tra il primo appello e a data in analisi; in tal caso la funzione dateIsOK provvede alla restituzione di un bool pari a false.

Se giunti sino alla fine di tale metodo vuol dire che indipendentemente dal tipo di esame (se attivo o spento, se del semestre di appartenenza o opposto, se primo o secondo appello) i controlli eseguiti sino ad ora sono andati a buon fine; pertanto la data passata come parametro al metodo dateIsOK è una data papabile motivo per cui la funzione restituisce un bool pari a true.

* **checkIfProfsAvailableAndGapSameSemesterCourses – classe “SessionYear”:**

int SessionYear::checkIfProfsAvailableAndGapSameSemesterCourses(

Course &course, Date &currentExamDay, std::map<int, Professor> &profs, std::map<int, Classroom> &classrooms, int relaxPar, int session, std::vector<int> &roomsFounded, int endHourSlot)

Questo metodo riceve come parametro una data papabile ad ospitare un esame e controlla rispetto a questa data che i docenti siano disponibili (ovvero che la data NON cada nel periodo di indisponibilità o che il docente NON sia occupato con un altro esame) e che nei due giorni precedenti non siano presenti esami già assegnati con stesso corso di studio e stesso anno dell’esame da assegnare.

Inoltre controlla anche se ci sono aule con la capienza giusta tali da poter ospitare gli studenti che devono sostenere l’esame e allo stesso modo se tali aule in quel determinato giorno sono libere (e non occupate da eventuali altri esami).

In particolare tale metodo riceve mediante parametri tali informazioni:

* “course”, il corso che si sta analizzando;
* “currentExamDay”, la data potenziale ad ospitare l’esame che verrà sottoposta ai controlli di questo metodo;
* “profs”, la mappa contenente tutti i professori che insegnano nell’università
* “allUniversityClassrooms”, la mappa contenente tutte le aule presenti nell’università;
* “relaxPar” rappresentante il numero di vincoli rilassati;
* “session”, il semestre corrispondente alla sessione in questione;
* “roomsFounded”, il vector di interi destinato a contenere gli id delle aule disposte per quell’esame;

“endHourSlot” contiene l’ora finale dell’utimo slot ospitante il primo appello;

In primis si recuperano le informazioni relative all’esame di quel corso di quello specifico anno accademico mediante il metodo getExamSpecificYear; quest’ultimo restituisce un oggetto della classe Exam che viene salvato così in “examToAssign” (si ricorda che essendo un oggetto della classe Exam contiene le seguenti info: durata in minuti dell’esame, tempo necessario per accedere ed abbandonare l’esame, modalità dell’esame ed infine luogo). Successivamente si salva in una variabile intera, “numSlots” il numero di slot necessari per poter svolgere tale esame; tale valore viene calcolato e restituito dal metodo della classe esami, howManySlots; quest’ultimo nello specifico somma le tre info seguenti (nonché dati privati dell’oggetto “examToAssign”): “\_time” (durata dell’esame), “\_enterTime” (tempo necessario per accedere all’esame) e “\_leaveTIme” (tempo necessario per abbandonare l’esame); sapendo che ogni fascia oraria ha la durata di 2 ore converte tale somma nel numero di slot necessari eseguendo una approssimazione per eccesso, in caso di risultato non intero.

A questo punto ci si affida al metodo isPossibleToAssignThisExam il quale si occupa del controllo vero e proprio ovvero controlla, rispetto alle informazioni passate come parametro, che i docenti siano disponibili (ovvero che la data NON cada nel periodo di indisponibilità o che il docente NON sia occupato con un altro esame) e che nei due giorni precedenti non siano presenti esami già assegnati con stesso corso di studio e stesso anno dell’esame da assegnare. Inoltre controlla anche se ci sono aule con la capienza giusta tali da poter ospitare gli studenti che devono sostenere l’esame e allo stesso modo se tali aule in quel determinato giorno sono libere (e non occupate da eventuali altri esami).

Così il metodo checkIfProfsAvailableAndGapSameSemesterCourses termina col restituire lo stesso numero intero restituito a sua volta proprio dal metodoisPossibleToAssignThisExam;

quest’ultimo in particolare trasmette alla funzione chiamante:

* l’ora di inizio dell’esame se l’esame in questione soddisfa i diversi controlli richiesti;
* il numero “-1” qualora l’esame in questione NON soddisfa i diversi controlli richiesti;
* **isPossibleToAssignThisExam – classe “SessionYear”:**

int SessionYear::isPossibleToAssignThisExam(Course course,

Date currentExamDay, std::map<int, Professor> &profs,

std::map<int, Classroom> &classrooms, int numSlot, int relaxPar, int session, std::vector<int> &roomsFounded, int endHourSlot)

Questo metodo come anticipato prima controlla che nei due giorni precedenti non siano presenti esami già assegnati con stesso corso di studio e stesso anno dell’esame da assegnare e attraverso un metodo interno, **isPossibleToAssignThisExamToProfs,** controlla anche:

* che i docenti siano disponibili (ovvero che la data NON cada nel periodo di indisponibilità o che il docente NON sia occupato con un altro esame);
* che ci siano aule con la capienza giusta tali da poter ospitare gli studenti che devono sostenere l’esame e allo stesso modo se tali aule in quel determinato giorno hanno un certo numero di slot consecutivi liberi (e non occupate da eventuali altri esami);

In particolare tale metodo riceve mediante parametri tali informazioni:

* “course”, il corso che si sta analizzando;
* “allUniversityProfs”, la mappa contenente tutti i professori che insegnano nell’università
* “allUniversityClassrooms”, la mappa contenente tutte le aule presenti nell’università;
* “numSlotsRequired”, il numero di fasce orarie richieste;
* “relaxPar” rappresentante il numero di vincoli rilassati;
* “session”, il semestre corrispondente alla sessione in questione;
* “roomsFounded”, il vector di interi destinato a contenere gli id delle aule disposte per quell’esame;

“endHourSlot” contiene l’ora finale dell’utimo slot ospitante il primo appello;

Si procede nel seguente ordine:

* Si controlla che il corso da inserire non abbia un esame dello stesso corso di studi e dello stesso anno in un range di due giorni precedenti alla data analizzata; in particolare questo controllo va fatto SOLO se il primo vincolo non è stato già rilassato ovvero se “relaxPar” è strettamente minore di 1, in altre parole quando “relaxPar” vale 0. In caso affermativo si recupera la data di inizio della sessione in questione e la si salva in un apposito oggetto, “dayOne”, della classe Date. Si calcola il gap, inteso come numero di giorni, presente tra la data di inizio della sessione, “dayOne”, e il giorno in quesitone, “currentExamDay”; in questo ci si avvale di un apposito metodo detto whatIsTheGap.

La distanza in giorni viene salvata in una apposita variabile detta “pos”; nel caso in cui la distanza dovesse essere superiore a due giorni si forza “pos” a due in quanto tale variabile viene riutilizzata nel for successivo; con quest’ultimo si controlla in un gap di giorni che va da 0 al numero presente in “pos” se sono presenti in tale gap esami dello stesso corso di studi e dello stesso anno rispetto a quello in questione. In caso affermativo si procede con la restituzione di un intero pari a “-1” comunicando al metodo chiamante come la ricerca non sia andata a buon fine; pertanto l’esame non può essere piazzato in quel giorno.

* Se il controllo è andato a buon fine o se il primo vincolo è stato rilassato (permettendo così di saltare le precedenti istruzioni) allora si prosegue controllando anche:
* che i docenti siano disponibili (ovvero che la data NON cada nel periodo di indisponibilità o che il docente NON sia occupato con un altro esame);
* che se ci sono aule con la capienza giusta tali da poter ospitare gli studenti che devono sostenere l’esame e allo stesso modo se tali aule in quel determinato giorno hanno un certo numero di slot consecutivi liberi (e non occupate da eventuali altri esami);

Questi due ultimi controlli vengono racchiusi nel metodo isPossibleToAssignThisExamToProfs.

Così il metodo isPossibleToAssignThisExam termina col restituire lo stesso numero intero restituito a sua volta proprio dal metodoisPossibleToAssignThisExamToProfs;

quest’ultimo in particolare trasmette alla funzione chiamante:

* l’ora di inizio dell’esame se l’esame in questione soddisfa i diversi controlli richiesti;
* il numero “-1” qualora l’esame in questione NON soddisfa i diversi controlli richiesti;
* **isPossibleToAssignThisExamToProfs – classe “ExamDay”:**

int ExamDay::isPossibleToAssignThisExamToProfs(Course course, std::map<int, Professor> &allUniversityProfs, std::map<int, Classroom> &allUniversityClassrooms, int numSlotsRequired, int relaxPar, std::vector<int>& idRoomsFounded, int endHourSlot)

Questo metodo come anticipato prima controlla che:

* che i docenti siano disponibili (ovvero che la data NON cada nel periodo di indisponibilità o che il docente NON sia occupato con un altro esame);
* che ci siano aule con la capienza giusta tali da poter ospitare gli studenti che devono sostenere l’esame e allo stesso modo se tali aule in quel determinato giorno hanno un certo numero di slot consecutivi liberi (e non occupate da eventuali altri esami);

In particolare tale metodo riceve mediante parametri tali informazioni:

* “course”, il corso che si sta analizzando;
* “allUniversityProfs”, la mappa contenente tutti i professori che insegnano nell’università;
* “allUniversityClassrooms”, la mappa contenente tutte le aule presenti nell’università;
* “numSlotsRequired”, il numero di fasce orarie richieste;
* “relaxPar” rappresentante il numero di vincoli rilassati;
* “session”, il semestre corrispondente alla sessione in questione;
* “idRoomsFounded”, il vector di interi destinato a contenere gli id delle aule disposte per quell’esame;

“endHourSlot” contiene l’ora finale dell’utimo slot ospitante il primo appello;

Mediante un for si puntano i diversi slot presenti nella giornata infatti l’indice parte da 0 e giunge al numero 5 (proprio perché l’ultimo slot va dalle 18 sino alle 20 come da consegna); a questo punto per ogni ciclo for, ovvero slot per slot, si effettuano i seguenti controlli:

* Si salva in una apposita variabile intera, “slotHour”, l’ora di inizio corrispondente allo slot puntato dal for;
* Si controlla se tra lo slot in cui è presente il primo appello e lo slot in cui potrebbe essere collocato il secondo sia presente una distanza di almeno 6 ore, condizione imposta da consegna nel caso in cui il secondo vincolo dovesse essere rilassato fino al punto che i due appelli dello stesso esame cadono nel medesimo giorno; in tal caso si impone il flag “toContinue” al valore false al fine di proseguire con gli ulteriori controlli in quanto quello della distanza minima di sei ore non è rispettato;
* Se il flag “toContinue” è true, ovvero se il vincolo delle sei ore è stato rispettato, si procede col cercare delle aule che abbiano la giusta capienza da poter ospitare gli studenti che devono sostenere l’esame; ci si affida al metodo searchAvailableClassroomInThisSlot che cerca l’aula con la giusta capienza controllando inoltre se è libera a partire dallo slot “slotHour” e per un numero di slot pari a “numSlotsRequired”, nonché per tutta la durata dell’esame (informazione che conosco in quanto passata come parametro); se la ricerca è andata a buon fine, ovvero se è stata trovata un’aula che soddisfa tali requisiti, allora il metodo restituisce un booleano pari a true che viene salvato nel flag “thereAreEnoughClassrooms”.
* Solo l’aver trovato l’aula disponibile (ovvero solo l’avere “thereAreEnoughClassrooms“ pari a true) permette di controllare successivamente se negli slot in cui l’aula è libera anche i professori sono disponibili; tale controllo viene affidato al metodo checkProfsAvaibility il quale controlla che i docenti del corso e di tutti i corsi paralleli siano liberi in quella giornata e in quell’ora; si ricorda infatti che:
* un docente potrebbe essere impegnato con un altro esame o che la data papabile dell’esame cada in un suo periodo di indisponibilità;
* gli esami dei corsi paralleli devono essere sostenuti il medesimo giorno nelle medesime fasce orarie;

Si distinguono pertanto due casi:

* se il controllo è andato a buon fine (ovvero il docente è libero in quel giorno e in quello slot orario “slotHour”) si incrementa una variabile contatore, “numSlotsFoundedSoFar”, che tiene traccia del numero di slot CONSECUTIVI in cui sia l’aula che il docente sono disponibili; si conserva anche nella variabile “startHourSlot” quella che potrebbe essere l’ora di inizio dell’esame.

Siccome il controllo sulla disponibilità del docente è fatto su una singola fascia oraria e siccome un esame può necessitare di più slot allora si controlla se il numero di slot papabili sino ad ora individuati, “numSlotsFoundedSoFar”, ha raggiunto il numero richiesto per sostenere l’esame, “numSlotsRequired”;

* in caso affermativo si procede con il settare la variabile “foundedStartHourSlot“ con l’orario di inizio dell’esame, “startHourSlot“, poc’anzi trovato. Osservando la seconda condizione del for si deduce facilmente come l’aver individuato lo slot orario permette di uscire dal loop in modo anticipato;
* nel caso in cui il numero di slot papabili sino ad ora individuati, “numSlotsFoundedSoFar”, non hanno raggiunto il numero richiesto per sostenere l’esame, “numSlotsRequired”, allora occorre avanzare con l’ausilio del for alla prossima fascia oraria ripetendo i controlli solo sui docenti;
* se il controllo non è andato a buon fine (ovvero il docente non è disponibile in quel giorno e/o in quello slot orario “slotHour”) allora si procede con l’azzerare la variabile contatore “numSlotsFoundedSoFar” in quanto l’esame deve essere sostenuto in slot orari consecutivi; inoltre si setta a false il flag “classroomIsOk” affinché si controlli, nel prossimo ciclo for, la disponibilità dell’aula a partire dalla nuova fascia oraria e per le successive fasce richieste, “numSlotsRequired”;

Terminato il for se non si è trovato un insieme di slot consecutivi tali da ospitare l’esame allora il metodo ritorna un numero intero pari a “-1” al fine di comunicare al metodo chiamante l’insuccesso verificatosi nella ricerca; diversamente, se sono stati trovati degli slot adatti, il metodo si conclude con l’inserimento di ulteriori info nel vector privato “\_tempGroupedCourseClassrooms” e con la restituzione dell’ora di inizio dell’esame, “foundedStartHourSlot”; in particolare “\_tempGroupedCourseClassrooms” contiene le aule che potrebbero essere occupati da corsi raggruppati per far si che una stessa aula non venga assegnata ad un altro corso raggruppato.

* **searchAvailableClassroomsInThisSlot – classe “ExamDay”:**

bool ExamDay::searchAvailableClassroomsInThisSlot(

std::map<int, Classroom> &allUniversityClassrooms, int numSeatsToSeach, std::vector<int> &idRoomsFounded, int slotHour, int numSLotsRequired)

Tale metodo, come anticipato,cerca le aule con la giusta capienza controllando inoltre se sono libere a partire dallo slot “slotHour” e per un numero di slot pari a “numSlotsRequired”, nonché per tutta la durata dell’esame. In particolare si passa a tale metodo le seguenti informazioni:

* “allUniversityClassrooms”, la mappa contenente tutte le aule presenti nell’università;
* **“**numSeatsToSeach”, il numero totale di posti a sedere da trovare (ovvero studenti che sostengono l’esame);
* **“**idRoomsFounded”, vector destinato a contenere alla fine del metodo gli id delle aule che ospiteranno l’esame in questione (ovviamente se le aule superano i controlli imposti);
* “slotHour”, ora iniziale del possibile slot destinato ad ospitare l’esame;
* “numSlotsRequired” contiene il numero di slot necessari per lo svolgimento dell’esame;

Mediante un for si scandaglia tutta la mappa “allUniversityClassrooms” ovvero tutte le aule presenti nell’università e per ognuna di essi si eseguono i seguenti controlli:

* Si salva in un oggetto di appoggio, “room”, della classe Classroom, l’aula puntata dal for;
* Si controlla che tale aula non sia stata in precedenza già assegnata ad un altro corso raggruppato e questo è possibile effettuando una ricerca all’interno del vector “\_tempGroupedCourseClassrooms” mediante il metodo find; nel momento in cui tale aula puntata dal for non risulta essere già stata assegnata allora si procede con l’effettuare gli ulteriori controlli; diversamente si passa al ciclo for successivo ovvero ad un’altra aula;
* Nel caso in cui tale aula non sia stata già assegnata allora si controlla mediante un for che l’aula sia libera a partire dallo slot “slotHour” e per il numero di slot necessari, numSlotsRequired”; non appena risulta occupata in uno degli slot orari compresi tra “slotHour” e “slotHour” + ”numSLotsRequired” allora si setta a false un flf, “roomIsOk” che permette di uscire immediatamente dal loop quindi facendo si che si analizzi l’aula successiva puntata dal for esterno;
* Se l’aula invece risulta disponibile in quelle fasce orarie, ovvero “roomIsOk” è true, allora si procede col tenere traccia dei posti contenuti in quell’aula libera;

in particolare, siccome il numero di studenti da posizionare potrebbe essere maggiore del numero di posti a sedere che dispone una singola aula allora risulta necessario individuare più aule; questo conduce all’utilizzo di una variabile contatore (“totSeatsFoundedSoFar”) che, non appena viene individuata un’aula libera, viene incrementata di un numero pari a quello dei posti a sedere della classe appena trovata; inoltre si salva tale aula appena individuata in coda nel vector “idRoomsFounded” (passato come parametro);

non appena il numero di posti a sedere sino ad ora individuati, “totSeatsFoundedSoFar”, risulta maggiore o uguale rispetto al numero di studenti che devono sostenere l’esame, “numSeatsToSeach”, allora il metodo si conclude con la restituzione di un bool pari a true comunicando al metodo chiamante che la ricerca è andata a buon fine e che le aule individuate sono ora presenti nel parametro “idRoomsFounded”;

diversamente se non sono stati trovati un numero di aule sufficienti tale da ricoprire il numero di posti richiesti nella fascia oraria che va da “slotHour” a “slotHour” + ”numSLotsRequired” allora il metodo restituisce un bool pari a false;

* **checkProfsAvaibility – classe “ExamDay”:**

bool ExamDay::checkProfsAvaibility(SpecificYearCourse &specificCourse, std::map<int, Professor> &allUniversityProfs, int relaxPar, int slotHour)

Tale metodo, come anticipato, controlla che i docenti del corso e di tutti i corsi paralleli siano liberi in una determinata giornata e in un certo slot. Il metodo infatti necessita di tali informazioni passategli come parametro:

* **“**specificCourse”, corso in questione con le informazioni specifiche dell’anno in analisi;
* **“**allUniversityProfs”, la mappa contenente tutti i professori che insegnano nell’università
* **“**relaxPar” rappresentante il numero di vincoli rilassati;
* **“**slotHour”, ora iniziale del possibile slot destinato ad ospitare l’esame;

Tutte le matricole dei professori dei corsi paralleli al corso “specificCourse” vengono salvati nella vector “profsMatr”; successivamente si trasforma la data in data stringa salvandola in “dataAsString”.

A questo punto mediante l’ausislio di un for si controlla per ogni professore dei corsi paralleli se in quello specifico slot orario risulta libero da altri eventuali esami e se tale data non cade in un suo periodo di indisponibilità. Questi ultimi due controlli vengono effettuati appellandosi al metodo amIavailable che ritorna false se anche solo uno dei precedenti vincoli non viene rispettato.

* Appena un docente non soddisfa tali vincoli si esce subito dal for e il metodo provvede alla restituzioone di un booleano pari a false comunicando l’insuccesso della ricerca al metodo chiamante;
* Se TUTTI I DOCENTI DEI CORSI PARALLELI rispettano i vincoli imposti da consegna allora il metodo restituisce un booleano pari a true;
* **amIavailable – classe “Professor”:**

bool Professor::amIavailable(std::string date, int hh)

Tale metodo, come spiegato brevemente in precedenza, controlla per un determinato docente se nel giorno “date” nello slot orario “hh” risulta libero da altri eventuali esami e se tale data non cade in un suo periodo di indisponibilità.

Il metodo infatti necessita di tali informazioni passategli come parametro:

* “date”, nonché la stringa contenente la data da controllare;
* “hh”, ovvero l’intero corrispondente all’ora iniziale dello slot sottosposto a controlli;

Il metodo inizia con il controllare appunto se la data potenziale all’esame cada o meno nei periodi di indisponibilità comunicati dal docente; pertanto si controlla prima se il docente ha comunicato dei periodi di indisponibilità per quell’anno accademico; tale controllo si ricorda essere fatto solo il terzo vincolo non è stato rilassato (diversamente si potrebbe collocare un esame in un giorno che cade nel periodo di indisponibilità comunicato dal professore e questo giustifica la seconda condizione nell’if inerente “relaxPar”<3); se il docente non ha comunicato indisponibilità allora si passa alla seconda parte del metodo; mentre in caso affermativo si scandagliano grazie ad un for tutti i periodi di indisponibilità del docente (che si sa essere una mappa privata della classe Professor avente come chiave un oggetto Date) ; per ognuno di essi, ovvero per ogni ciclo for, si estrae il giorno di inizio e di fine del periodo di indisponibilità e li si salvano in degli oggetti Date di appoggio detti rispettivamente “lower” ed “upper”; a questo punto si controlla se la data papabile ad ospitare l’esame è contenuta in tale intervallo (ovvero se “lower”<=”d”<=”upper”):

* Se è contenuta allora vuol dire che si sta provando a collocare un esame in un giorno che cade in uno dei periodi di indisponibilità del docente motivo per cui in tal caso il metodo si conclude con la restituzione di un boolenao pari a false evitando di proseguire con ulteriori controli e comunicando al metodo chiamente l’insuccesso della ricerca;
* Se invece non è contenuta allora si può proseguire con il controllo successivo;

Solo se giunti a questo punto (ovvero solo se la data non è interna al periodo di indisponibilità) allora si procede con la seconda parte del metodo ovvero col controllare se nell’ora “hh” nella data “d”il docente risulta già impegnato con un altro esame.

si controlla prima se il docente ha altri esami in quella giornata:

* In caso negativo (libero da esami) il metodo ritorna un boleano pari a true comunicando che in quella data, a prescindere dallo slot, il docente è completamente disponibile;
* In caso afermativo (occupato in data “d” da un esame) si passa al controllare se il docente ha altri esami nell’ora specifica “hh”:
* In caso negativo allora il metodo ritorna un boleano pari a true comunicando che nell’ora ”hh” in data “d” il docente è completamente disponibile;
* In caso affermativo allora il metodo ritorna un booleano pari a false;
* **generateOutputFiles – classe “SessionYear”:**

void SessionYear::generateOutputFiles(std::string &outputFileName,

int session, std::map<std::string, Course> &courses)

Tale metodo genera il file di output degli appelli inerenti la sessione di esame passatagli come parametro.

Esso riceve come parametri di input:

* “outputFileName” nonché la stringa contenente la base del nome dei tre file;
* “session” ovvero un intero che indica la sessione (invernale, estiva, autunnale);
* “courses” che rappresenta la mappa di tutti i corsi dell’università;

Si dichiara un oggetto “ssFout” della classe stringstream che verrà usato per costruire il nome del file.

Si salva in una stringa, detta “key”, il tipo di sessione corrispondente all’intero presente in “session”.

Successivmente si applica la funzione splittedLine al nome del file passato come parametro, “outputFileName”, rispetto al carattere “.” al fine di isolare il nome indicato da linea di comando dall’estensione “.txt”. Si salva il risultato di tale funzione in un vector di stringhe detto “token”; pertanto:

* “token [0]” contiene la base del nome del file suggerito da linea di comando;
* “token [1]” contiene l’estensione del file ovvero “txt”;

Si inserisce la base del nome del file, “token [0]”, nell’oggetto stringstream “ssFout”.

A questo punto si distinguono tre casi a seconda del contenuto della stringa “key” ovvero a seconda di quale sessione si desidera generare il file; pertanto:

* Se “key” contiene la stringa “winter” allora si accoda al già presente contenuto di “ssFout” la seguente parte finale: “\_s1.txt”;
* Se “key” contiene la stringa “summer” allora si accoda al già presente contenuto di “ssFout” la seguente parte finale: “\_s2.txt”;
* Se “key” contiene la stringa “autumn” allora si accoda al già presente contenuto di “ssFout” la seguente parte finale: “\_s3.txt”;

Dovendo generare un file si dichiara un oggetto “outputSession” della classe “fstream” che si occupa dell’apertura del file il cui nome completo è presente nella stringstream “ssFout”.

In particolare si genera tal file aprendolo nelle due seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Dopo aver controllato che l’apertura sia avvenuta correttamente, al fine di lanciare una eccezione in caso di mancata apertura, si procede col salvare in due oggetti della classe Date, “dayOne” e “lastDay”, rispettivamente il giorno di inizio e di fine della sessione in questione.

Mediante un for si scandaglia l’intera sesione in analisi dal primo all’ultimo giorno, ovvero da “dayOne” a “lastDay” (infatti nel momento in cui si raggiunge l’ultimo giorno viene attivato un flag booleano, continueLoop”, che al prossimo ciclo di for impedisce l’accesso nel loop).

Per ogni ciclo for ovvero per ogni giorno del calendario in memoria si etrae la chiave, nonché il nome del giorno (inteso come yyyy/mm/dd) salvato in una stringa detta “day” e l’oggetto della classe ExamDay salvato in “examDay”. A questo punto solo per i giorni aventi almeno uno slot pieno si effettuano le seguenti istruzioni di stampa (pertanto si evitano i giorni vuoti in cui non è stato fissato alcun esame):

* si dedica una intera riga stampando all’interno il giorno, ovvero la stringa “day”;
* si recuperano tutti gli slot di tale giorno salvandoli nel vector di stringhe “allSlots”;
* mediante un for si cicla su ogni elemento del vector ovvero su ogni slot; si dedica così una intera riga del file al contenuto dell’i-esimo elemento del vector “allSlots” ovvero: ora iniziale, ora finale seguiti dall’elenco dei codici dei corsi corrispondenti agli esami che si tengono in quello slot accomapgnati anche dall’id del corso di studio di appartenenza, come richiesto da consegna;

Dopo aver inserito le info relative all’intera sessione di esame il metodo si conclude con la chiusura del file mediante l’apposito metodo close.

* **generateWarnings – classe “SessionLog”:**

void SessionLog::generateWarnings(std::vector<Course> &courses, int relaxPar, int gap, int year, std::map<std::string,int> gapProfs, int session)

Questo metodo gestisce le diverse situazioni di warnings, a seconda del contesto che gli viene passato come parametro, salvandole inoltre nei dati privati degli oggetti di una apposita classe detta “SessionLog”; gli oggetti di quest’ultma classe infatti tengono traccia di diverse informazioni importanti mediante i seguenti dati privati:

* “\_output\_file\_name”, la stringa contenente la base del nome usato per i file delle sessioni e dei warnings;
* “cdsCodes”, la stringa …
* “\_logPerSession”, la mappa avente come chiave l’intero rappresentante la sessione e come value la stringa con all’interno salvati tutti i warnings relativi;

Questo metodo riceve come parametri:

* “courses”, il vector contenente i corsi rispetti ai quali si devono generare uno o più warnings;
* “relaxPar”, l’intero indicante il numero di vincoli sino ad ora rilassati;
* “gap”, l’intero rappresentante la distanza in giorni tra due appelli del medesimo esame;
* “year”, l’intero che si riferisce all’anno accademico;
* “gapProfs” nonché la mapppa contenente per ogni professore la distanza minima richiesta dal docente stesso per i suoi esami (che si ricorda essere a sua volta dato privato per gli oggetti della classe SessionYear);
* “session” ovvero l’intero rappresentante il tipo di sessione (o invernale o estiva o autunnale);

Tale metodo inizia invocandone un secondo detto generateWarningGapProfs; quest’ultimo, come lascia pensare il nome stesso, salva nella stringa presente nella mappa “\_logPerSession” (in corrispondenza della sessione in questione) l’id del professore e del corso da lui insegnato, il cui gap comunicato non è stato rispettato, evidenziando anche a quale intervallo di giorni si allude.

Solo in seguito si passa con l’analizzare gli eventuali warnings derivanti da vincoli rilassati andando a distinguere, sulla base del valore di “relaxPar”, i diversi scenari mutuamente esclusivi di seguito citati:

* Se “relaxPar” vale 1 (ovvero se la distanza minima tra esami dello stesso corso di studi è stata ridotta a meno di due giorni) ci si affida all’apposito metodo, generateWarningGapSameStudyCourse, che salva nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione e dell’anno in questione) il codice identificativo del corso di studio e del corso per il quale è stato violato il primo vincolo, riportandoli, come da consegna, nel medesimo ordine con cui sono stati appena spiegati.

Al termine dell’esecuzione di tale funzione si esce dal ramo dello switch;

* Se “relaxPar” vale 2 significa che sia il primo che il secondo vincolo (ovvero distanza minima tra appelli dello stesso esame ridotta a meno di 14 giorni, sino eventualmente alla situazione peggiore di esami nello stesso giorno con almeno sei ore di distanza l’uno dalll’altro) sono stati rilassati; pertanto ci si affida ai metodi generateWarningGapSameStudyCourse e generateWarningGapAppeals rispettivamente per la comunicazione dello scioglimento del primo e del secondo vincolo; in particolare generateWarningGapAppeals salva nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione e dell’anno in questione) il codice identificativo del corso di studio e del corso per il quale è stato violato il secondo vincolo, riportandoli, come da consegna, nel medesimo ordine con cui sono stati appena spiegati.

Al termine dell’esecuzione di tale funzione si esce dal ramo dello switch;

* Se “relaxPar” vale 3 significa che sia il primo che il secondo e terzo vincolo (uno o più esami sostenuti da un professore collocati in un giorno interno al periodo di indisponibilità comunicato dal docente stesso) sono stati rilassati; pertanto, come si può facilmente intuire si fa uso dei metodi generateWarningGapSameStudyCourse, generateWarningGapAppeals e generateWarningGapAvaibilityProfs rispettivamente per la comunicazione dello scioglimento del primo, del secondo e del terzo vincolo; in particolare generateWarningGapAvaibilityProfs salva nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione e dell’anno in questione) il codice identificativo del corso di studio e del corso per il quale è stato violato il terzo vincolo, riportandoli, come da consegna, nel medesimo ordine con cui sono stati appena spiegati.

Al termine dell’esecuzione di tale funzione si esce dal ramo dello switch;

* Se “relaxPar” assume qualsiasi altro valore che non sia uno dei tre mostrati in precedenza si provvede direttamente ad usicre dal ramo di default dello switch;
* **generateWarningGapProfs – classe “SessionLog”:**

void SessionLog::generateWarningGapProfs(std::map<std::string,int> gapProfs, int year, int session)

Tale metodo, come preannunciato, salva nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione e dell’anno in questione) l’id del professore e del corso da lui insegnato, il cui gap comunicato non è stato rispettato, evidenziando anche l’intervallo particolare di giorni violato. Tale metodo riceve mediante parametri le seguenti informazioni utili:

* “gapProfs” nonché la mapppa contenente per ogni professore la distanza minima richiesta dal docente stesso per i suoi esami;
* “year”, l’intero che si riferisce all’anno accademico;
* “session” ovvero l’intero rappresentante il tipo di sessione (o invernale o estiva o autunnale);

Si dichiara un oggetto detto “ss” della classe stringstream utile al fine di comporre le diverse informazioni prima che queste vengano inserite nella stringa interna alla mappa privata “\_logPerSession”.

Si inizia con il copiare l’anno “year” in “ss” affinchè sia noto l’anno a cui il warnings in questione si riferisce.

Successivamente per ogni elemento della mappa “gapProfs” (passata come parametro al metodo) si esegue l’istruzione seguente, basata sull’accodare di volta in volta le informazioni riportate di seguito nel medesimo ordine con cui vengono spiegate nell’imminente:

* matricola del docente e del corso da lui insegnato (nonché chiave della mappa privata “\_gapProfs” della classe “SessionYear”), il cui gap comunicato non è stato rispettato;
* stringa di contenuto: “GAP\_PROFESSORE di”;
* intero rappresentante il gap comunicato dal docente insegnato (nonché value della mappa privata “\_gapProfs” della classe “SessionYear”);
* stringa di contenuto: “non rispettata” seguita dal carattere speciale ‘\n’ che consente di andare a capo;

Infine si copia il contenuto di “ss” in una stringa detta “toReturn” la quale viene passata, insieme all’intero indicante il tipo di sessione “session”, al metodo appendLogPerSession.

Il metodo appendLogPerSession semplicemente accoda nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione) il contenuto della stringa “toReturn”.

* **generateWarningGapSameStudyCourse – classe “SessionLog”:**

void SessionLog::generateWarningGapSameStudyCourse(std::vector<Course> &courses, int year, int session)

Tale metodo, come preannunciato, salva nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione e dell’anno in questione) il codice identificativo del corso di studio e del corso per il quale è stato violato il primo vincolo, riportandoli, come da consegna, nel medesimo ordine con cui sono stati appena spiegati.

Tale metodo riceve mediante parametri le seguenti informazioni utili:

* “courses”, il vector contenente i corsi rispetti ai quali si devono generare uno o più warnings;
* “year”, l’intero che si riferisce all’anno accademico;
* “session” ovvero l’intero rappresentante il tipo di sessione (o invernale o estiva o autunnale);

Si dichiara un oggetto detto “ss” della classe stringstream utile al fine di comporre le diverse informazioni prima che queste vengano inserite nella stringa interna alla mappa privata “\_logPerSession”.

Successivamente per ogni elemento del vector “courses” (passato come parametro al metodo) si recupera il corso con le informazioni inerenti l’anno in questione salvandolo in un oggetto detto “sp” della classe SpecificYearCourse.

In seguito si esegue l’istruzione seguente, basata sull’accodare di volta in volta le informazioni riportate di seguito nel medesimo ordine con cui vengono spiegate nell’imminente:

* codice identificativo del corso di studio e del corso per il quale è stato violato il primo vincolo;
* stringa di contenuto: “regola GAP\_CORSI\_STESSO\_CORSO\_DI\_STUDIO non rispettata” seguita dal carattere speciale ‘\n’ che consente di andare a capo;

Infine si copia il contenuto di “ss” in una stringa detta “toPass” la quale viene passata, insieme all’intero indicante il tipo di sessione “session”, al metodo appendLogPerSession.

Il metodo appendLogPerSession semplicemente accoda nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione) il contenuto della stringa “toPass”.

* **generateWarningGapAppeals – classe “SessionLog”:**

void SessionLog::generateWarningGapAppeals(std::vector<Course> &courses, int gap, int year, int session)

Tale metodo, come preannunciato, salva nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione e dell’anno in questione) il codice identificativo del corso di studio e del corso per il quale è stato violato il secondo vincolo, riportandoli, come da consegna, nel medesimo ordine con cui sono stati appena spiegati.

Tale metodo riceve mediante parametri le seguenti informazioni utili:

* “courses”, il vector contenente i corsi rispetti ai quali si devono generare uno o più warnings;
* “gap”, l’intero rappresentante la distanza in giorni tra due appelli del medesimo esame;
* “year”, l’intero che si riferisce all’anno accademico;
* “session” ovvero l’intero rappresentante il tipo di sessione (o invernale o estiva o autunnale);

Si dichiara un oggetto detto “ss” della classe stringstream utile al fine di comporre le diverse informazioni prima che queste vengano inserite nella stringa interna alla mappa privata “\_logPerSession”.

Successivamente per ogni elemento del vector “courses” (passato come parametro al metodo) si recupera il corso con le informazioni inerenti l’anno in questione salvandolo in un oggetto detto “sp” della classe SpecificYearCourse.

In seguito si esegue l’istruzione seguente, basata sull’accodare di volta in volta le informazioni riportate di seguito nel medesimo ordine con cui vengono spiegate nell’imminente:

* codice identificativo del corso di studio e del corso per il quale è stato violato il secondo vincolo;
* stringa di contenuto: “regola GAP\_APPELLI: questo esame è stato programmato con uno scarto di”;
* l’intero “gap”, rappresentante la distanza in giorni tra due appelli del medesimo esame, seguito dal carattere speciale ‘\n’ che consente di andare a capo;

Infine si copia il contenuto di “ss” in una stringa detta “toPass” la quale viene passata, insieme all’intero indicante il tipo di sessione “session”, al metodo appendLogPerSession.

Il metodo appendLogPerSession semplicemente accoda nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione) il contenuto della stringa “toPass”.

* **generateWarningGapAvaibilityProfs – classe “SessionLog”:**

void SessionLog::generateWarningGapAvaibilityProfs(std::vector<Course>& courses, int year, int session)

Tale metodo, come preannunciato, salva nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione e dell’anno in questione) il codice identificativo del corso di studio e del corso per il quale è stato violato il terzo vincolo, riportandoli, come da consegna, nel medesimo ordine con cui sono stati appena spiegati.

Tale metodo riceve mediante parametri le seguenti informazioni utili:

* “courses”, il vector contenente i corsi rispetti ai quali si devono generare uno o più warnings;
* “year”, l’intero che si riferisce all’anno accademico;
* “session” ovvero l’intero rappresentante il tipo di sessione (o invernale o estiva o autunnale);

Si dichiara un oggetto detto “ss” della classe stringstream utile al fine di comporre le diverse informazioni prima che queste vengano inserite nella stringa interna alla mappa privata “\_logPerSession”.

Successivamente per ogni elemento del vector “courses” (passato come parametro al metodo) si recupera il corso con le informazioni inerenti l’anno in questione salvandolo in un oggetto detto “sp” della classe SpecificYearCourse.

In seguito si esegue l’istruzione seguente, basata sull’accodare di volta in volta le informazioni riportate di seguito nel medesimo ordine con cui vengono spiegate nell’imminente:

* codice identificativo del corso di studio e del corso per il quale è stato violato il terzo vincolo;
* stringa di contenuto: “regola AVAIBILITY\_PROFS non rispettata”, seguita dal carattere speciale ‘\n’ che consente di andare a capo;

Infine si copia il contenuto di “ss” in una stringa detta “toPass” la quale viene passata, insieme all’intero indicante il tipo di sessione “session”, al metodo appendLogPerSession.

Il metodo appendLogPerSession semplicemente accoda nella stringa della mappa privata “\_logPerSession” (in corrispondenza della sessione) il contenuto della stringa “toPass”.

* **writeWarnings – classe “SessionLog”:**

void SystemLog::writeWarnings()

Questo metodo si occupa della generazione dei tre file di output inerenti i warnings delle tre sessioni d’esame. Si dichiara un oggetto detto “ss” della classe stringstream al fine di comporre quelli che saranno i nomi dei tre file.

Mediante un for si distinguono le tre generazioni, una generazione (ovvero un ciclo for) per ogni file.

In ogni ciclo for si eseguono le istruzioni riportate di seguito:

* si inserisce in “ss” la bese del nome del file presente nella variabile privata “\_output\_file\_name” della classe SessionLog alla quale appartiene lo stesso metodo writeWarnings;
* si accoda la stringa “\_s” seguita dal numero assunto dall’indice “i” del for;
* si inserisce in coda la stringa di contenuto “\_warnings.txt”;
* Si dichiara un oggetto “fout” della classe fstream che consente di aprire il file il cui nome è presente in “ss”; in particolare l’apertura avviene in modalità “out”.
* Infine si scrive nel file “fout” il contenuto della stringa (nonché value della mappa) interna alla mappa privata “\_logPerSession” (della classe SessionLog)inerente la sessione indicata dall’indice “i” del for (nonché chiave della mappa)
* **allExamAppealsWrite – classe “SessionYear”:**

void SessionYear::allExamAppealsWrite(std::map<std::string, Course> &courses)

Questo metodo permette di tenere traccia di tutti gli appelli già definiti in un apposito file database detto “allExamAppealsDb.txt”; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database in entrambe le seguenti modalità:

* out affinché si possa scrivere su tale file;
* trunc affinché eventuali contenuti già presenti nel file vengano sovrascritti;

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe “Course” si procede col scandagliare tutta la mappa dei corsi in memoria. Per ogni ciclo for si recuperano, grazie al metodo getAcYearAppeals, tutti gli appelli per i diversi corsi e salvati momentaneamente nel vector di stringhe “allAppealsPerCourses“; con l’ausilio di un ulteriore for si stampano i diversi elementi di quest’ultimo vector andando a capo ad ogni appello; tutto questo grazie all’overload dell’operatore “<<”. In particolare tale database rispetta il seguente formato: codice id del corso, anno accademico seguito dai tre periodi delle sessioni con i rispettivi appelli.

Come buona norma di programmazione il metodo si conclude con la chiusura del file usando l’apposito metodo close.

* **controlDatabase GIA’ SPIEGATO PRIMA**
* **dataBaseIsEmpty GIA’ SPIEGATO PRIMA**
* **controlGroupedCourses GIA’ SPIEGATO PRIMA**
* **thereIsAHoleInTheCoursesCodes GIA’ SPIEGATO PRIMA**
* **checkDistance GIA’ SPIEGATO PRIMA**
* **controlReciprocyGrouped GIA’ SPIEGATO PRIMA**
* **dbCourseNotActive GIA’ SPIEGATO PRIMA**
* **readCourseNotActive GIA’ SPIEGATO PRIMA**
* **versioning GIA’ SPIEGATO PRIMA**
* **renameOldDataBase GIA’ SPIEGATO PRIMA**
* **addStudyPlan GIA’ SPIEGATO PRIMA**
* **dbStudyPlanWrite GIA’ SPIEGATO PRIMA**
* **readStudyPlan GIA’ SPIEGATO PRIMA**
* **updateStudyPlan GIA’ SPIEGATO PRIMA**
* **insertStudentGrades GIA’ SPIEGATO PRIMA**
* **registerStudentsToSpecificYearCourses GIA’ SPIEGATO PRIMA**
* **dbAppealsWrite GIA’ SPIEGATO PRIMA**
* **readPassedAppeals GIA’ SPIEGATO PRIMA**
* **readAllExamAppeals**

Prototipo: void University::readAllExamAppeals()

Questo metodo si occupa della lettura del database contenente tutti gli appelli dei diversi corsi denominato: allExamAppealsDb.txt.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Tale riga, salvata in un oggetto stringa di appoggio “line”, viene passata, insieme al carattere separatore “ ; “ alla funzione “splittedLine” la quale restituisce un oggetto vector di stringhe detto “appeals” in modo che ogni elemento del vector contenga l’informazione presente nella riga compresa tra due caratteri separatori “;”. In particolare:

* “appeals [0]” contiene l‘id del corso che viene salvato nella variabile intera “idCorso”;
* “appeals [1]” contiene l’anno accademico della sessione di esami con anno di inizio e anno di fine che viene salvato nella stringa “acYear”;
* “appeals [2]” contiene gli appelli e le info delle tre sessioni di esami;

Successivamente mediante l’ausilio nuovamente della funzione splittedLine si separano le info riguardanti i tre periodi di sessione, considerando come carattere separatore “%”. Si ottiene il vector di stringhe “infoEachSession” dove ogni suo campo è dedicato ad una particolare sessione, ovvero:

* “infoEachSession [0]” contiene gli appelli e le info della sessione invernale;
* “infoEachSession [1]” contiene gli appelli e le info della sessione esitva;
* “infoEachSession [2]” contiene gli appelli e le info della sessione autunnale;

In seguito per ogni sessione, ovvero per ogni campo del vector “infoEachSession“ (ovvero grazie ad un for),si estraggono i seguenti dati:

* salvo nel vector di interi “posSquareBrackets” le posizioni occupate dalle parentesi “[“ e “]” all’interno di “infoEachSession [i]”;
* salvo nella stringa “session” il tipo di sessione in questione (“winter” o “summer” o “autumn”); dato che ricopre tutti i caratteri che vanno dal primo di “infoEachSession [i]” sino a quello precedente alla prima “[“;
* salvo nella stringa “appealsPerSessionString” le date e le info degli appelli della sessione in questione;
* salvo nel vector di interi “posCurlyBrackets” le posizioni occupate dalle parentesi “{“ e “}” all’interno di “appealsPerSessionString”;
* estraggo (all’interno di un for) le date degli appelli con annesse le relative informazioni, dedicando ogni campo del vector di stringhe “appealInfo” ad una data di un appello con i suoi relativi dati (dove per dati si intende l’ora di inizio dell’esame separato dal carattere “|” dall’id delle aule disposte per quell’esame);
* Solo dopo aver ordinato le informazioni in apposite variabili si procede al caricamento vero e proprio in memoria avvalendosi dell’apposito metodo **assignInfoAppealPerSession** della classe University (metodo che verrà spiegato nelle righe avvenire al fine di garantire e preservare un certo ordine per quanto riguarda la descrizione del codice);

Quanto spiegato si ricorda essere fatto per ogni riga del file dal momento che tali operazioni sono tutte presenti nel while inizialmente citato di tale metodo.

Tale metodo si conclude con la chiusura del file database dei corsi di studio allExamAppealsDb.txt.

* **writeVersion GIA’ SPIEGATO PRIMA**
* **assignInfoAppealPerSession**

Prototipo: void University::assignInfoAppealPerSession(std::string acYear, std::string idCorso, std::string session, std::vector<std::string> appealInfo)

Questo metodo, come anticipato, si occupa del caricamento vero e proprio in memoria delle informazioni riguardanti una sessione di esami (o “winter” o “summer” o “autumn” di un determinato anno accademico e di un determinato corso) appena letta dal database allExamAppealsDb.txt con il metodo readAllExamAppeals.

Infatti tale metodo riceve come parametri:

* “acYear” nonché la stringa contenente l’anno accademico della sessione di esami composta da anno di inizio e di fine;
* “idCorso” nonché la stringa contenente il codice identificativo del corso;
* “session”, stringa con all’interno il tipo di sessione (o “winter” o “summer” o “autumn”);
* “appelInfo” ovvero un vector di stringhe dove ogni campo presenta una data di un appello con i suoi relativi dati (dove per dati si intende l’ora di inizio dell’esame separato dal carattere “|” dall’id delle aule disposte per quell’esame);

Si usa il metodo **getExamSlotPerYear** (della classe Course) il quale, per un determinato anno accademico ricevuto come parametro, restituisce il numero di slot necessari per sostenere l’esame del corso in questione, “idCorso”; tale dato viene salvato nella variabile intera “numSlot”.

Grazie al metodo **getAcStartYear** (della classe Parse), noto l’anno accademico “acYear”, si riesce inoltre a recuperare la data di inizio che viene salvata nella variabile intera “startAcYear”.

Infine mediante il metodo **getProfsPerYear** (dela classe Course) si recuperano tutte le matricole di tutti i docenti di uno specifico anno per tutti i corsi paralleli; questa informazione come si può immaginare viene salvata in un vector di interi dove ogni campo è destinato ad ospitare l’id di un docente, vector che prende il nome di “allProfsPerYearCourse”. Si precisa inoltre come getProfsPerYear si avvalga al suo interno di un ulteriore metodo detto getAllProfMatr (della classe SpecificYearCourse) che accedendo al vector di struct privato “\_professors” (della classe SpecificYearCourse) riesce a recuperare l’id di tutti i prof di tutte le versioni in parallelo.

Successivamente con l’ausilio di un for si scandaglia il vector “appelInfo” in cui ogni suo campo presenta una data di un appello con i suoi relativi dati (appelli, e quindi campi, che potranno essere uno o due a seconda della sessione “session” passata come parametro); per ogni campo, quindi per ogni appello, si eseguono le istruzioni riportate di seguito:

* Si passa la data del singolo appello con le relative informazioni alla funzione splittedLine insieme al carattere separatore “,” che vengono salvati nei diversi campi del vector di stringhe “infoOfSingleAppeal”:
* “infoOfSingleAppeal [0]” contiene la data dell’appello (stringa con formato yyyy-mm-dd);
* “infoOfSingleAppeal [1]” contiene l’ora di inizio dell’esame;
* “infoOfSingleAppeal [2]” contiene le aule che ospitano l’esame divise dal carattere separatore “|”;
* Si salva la data dell’appello della sessione “session” accodandola col metodo emplace\_back nel vector “appealDate” di oggetti della classe Date;
* Con il metodo **assignAppealsToProf** (della classe University) si provvede ad aggiornare la mappa privata “\_professors” (della classe University costituita da oggetti Professor) con le informazioni riguardanti l’esame che vengono passate come parametro: id del corso, data dell’appello, ora di inizio dell’esame (sotto forma di intero), numero di slot orari necessari per l’esame, le matricole di tutti i docenti di uno specifico anno per tutti i corsi paralleli. Questo metodo verrà spiegato nelle righe avvenire al fine di garantire e preservare un certo ordine per quanto riguarda la descrizione del codice;
* Con il metodo **assignAppealsToClassroom** (della classe University) si provvede ad aggiornare la mappa privata “\_classrooms” (della classe Univeristy costituita da oggetti Classroom) con le informazioni riguardanti l’esame che vengono passate come parametro: data dell’appello, ora di inizio dell’esame (sotto forma di intero), numero di slot orari necessari per l’esame, la sringa con le aule che ospiteranno l’esame divise dal carattere separatore “|”. Questo metodo verrà spiegato nelle righe avvenire al fine di garantire e preservare un certo ordine per quanto riguarda la descrizione del codice;
* Si recupera l’oggetto corso corrispondente all’esame in questione e lo si salva nella variabile “course”;
* Con il metodo **assignAppealsToCalendar** (della classe SessionYear) si provvede ad aggiornare la mappa privata “\_yearCalendar” (della classe SessionYear) che si trova a sua volta come value nella mappa privata “\_acYearSessions” (della classe University costituita da oggetti SessionYear); l’aggiornamento avviene con le informazioni riguardanti l’esame che vengono passate come parametro: data dell’appello, ora di inizio dell’esame (sotto forma di intero), oggetto “course”. Questo metodo verrà spiegato nelle righe avvenire al fine di garantire e preservare un certo ordine per quanto riguarda la descrizione del codice;

Al termine del for infine con il metodo **assignAppealToSpecificYear** (della classe Course) si provvede a salvare per quel determinato anno e corso le date dell’appello nella mappa privata “\_howManyTimesIAmAssignedInASession” (della classe SpecificYearCourse); in particolare vengono passati al metodo tali informazioni: l’anno accademico, la sessione e le date degli appelli sotto forma di vector di oggetti della classe Date.

* **assignAppealsToCalendar – classe “SessionYear”**

void SessionYear::assignAppealsToCalendar(std::string appeal,

int startSlotHour, Course &course, int numSlots)

Questo metodo, come preannunciato, salva nella mappa privata “\_yearCalendar” (della classe SessionYear) le informazioni che gli vengono passati come parametro che sono le seguenti:

* “appeal” contenente la data dell’appello (stringa con formato yyyy-mm-dd);
* “startSlotHour” ovvero l’intero rappresentante l’ora di inizio dello slot ospitante l’esame;
* “course” nonché l’oggetto della classe Course corrispondente all’esame in questione;
* “numSlots” ovvero il numero di slot necessari allo svolgimento dell’esame;

**Si ricorda come la mappa privata “\_yearCalendar” (della classe SessionYear) abbia come:**

* **key: stringa della data dell’appello nel formato yyyy-mm-dd;**
* **value: oggetti della classe ExamDay che contengono le info dell’appello;**

All’interno della mappa “\_yearCalendar” si recupera l’oggetto della classe ExamDay avente come chiave l’appello contenuto nella stringa “appeal”; a questo punto si passano al metodo **assignExamToExamDay** (della classe ExamDay): l’ora di inizio dello slot ospitante l’esame, l’oggetto della classe Course corrispondente all’esame in questione e il numero di slot necessari allo svolgimento dell’esame;

tale metodo a sua volta per ogni fascia oraria dedicata all’esame in questione (ovvero per ogni fascia oraria che va da “startSlotHour” a “startSlotHour”+ 2\*“numSlots”) setta il dato privato “\_slots” con il corso in questione.

**Si ricorda come la mappa privata “\_slot” (della classe ExamDay) abbia come:**

* **key: l’ora di inizio della fascia oraria;**
* **value: vector di oggetti della classe Course i cui esami si svolgono nella fascia oraria indicata dalla key;**

Pertanto nella mappa “\_slot” si posiziona il corso in questione, “course”, nel vector di oggetti Course in corrispondenza dello slot orario puntato dal for;

* **assignAppealToSpecificYear – classe “Course”**

bool Course::assignAppealToSpecificYear(std::string acYear,

std::string session, std::vector<Date> appealsPerSession,

std::vector<int> startSlotPerAppeal,

std::vector<std::string> classroomsPerAppeal)

Questo metodo, come preannunciato, salva nella mappa privata “\_howManyTimesIAmAssignedInASession” (della classe SpecificYearCourse) le date dell’appello avvalendosi delle informazioni che gli vengono passati come parametro che sono le seguenti:

* “acYear”, nonché la stringa contenente l’anno accademico;
* “session” ovvero la stringa indicante la sessione;
* “appealsPerSession” nonché il vector di oggetti Date che contiene gli appelli per sessione;
* “startSlotPerAppel” nonché il vector contenente per ogni appello l’ora di inizio del primo slot ospitante l’esame;
* “classroomsPerAppeal” ovvero un vector di stringhe contenente le aule per ogni appello;

**Si ricorda come la mappa privata “\_courseOfTheYear” (della classe SessionYear) abbia come:**

* **key: l’anno di inizio dell’anno accademico;**
* **value: oggetti della classe SpecificyearCourse che contengono le info dei corso di uno specifico anno accademico;**

All’interno della mappa “\_courseOfTheYear” si recupera l’oggetto della classe SpecificYearCourse avente come chiave l’anno di inizio dell’annoaccademic**o** contenuto nell’intero “acStartYear”; a questo punto si passano al metodo **assignAppeals** (della classe SpecificYearCourse): la sessione, vector di oggetti Date che contiene gli appelli per sessione, l’ora di inizio dei primi slot ospitanti gli esami e il vector di stringhe contenente le aule per ogni appello.

Tale metodo a sua volta per ogni appello passatogli come parametro esegue le istruzioni evidenziate di seguito andando a settare i diversi parametri privati delgli oggetti della classe SpecificYearcourse:

* salva nella mappa privata “\_startSlotPerEachAppeal” l’ora di inizio del primo slot ospitante l’esame di un determinato appello.

**Si ricorda come la mappa privata “\_startSlotPerEachAppeal” (della classe SpecificYearCourse) abbia come:**

* **key: un numero intero che consente di distinguere i diversi appelli e che va da 0 a 3 per un totale così di quattro appelli (0, 1, 2, 3);**
* **value: l’ora di inizio dello sot ospitante l’esame di quel determinato appello;**
* si apllica la funzione splittedLine all’i-esimo campo del vector “classroomsPerAppeal” (contenente tutte le aule dell’appello puntato dal for) pasando “|” come carattere separatore; pertanto si ottiene il vector di stringhe “allClassroomsPerAppeal” che pertanto nel complesso contiene tutte le aule per l’appello puntato dal for (quindi ogni campo di “allClassroomsPerAppeal” contiene l’id di una singola aula); infine campo per campo di quest’ultimo vector, ovvero per ogni aula, si provvede alla conversione in intero (mediante il metodo checkedStoi) del singolo campo, ottenendo quindi un vector di interi detto “allClassroomsPerAppealInt”;
* salva nella mappa privata “\_roomsEachAppeal” gli id di tutte le aule in corrispondenza di quel determinato appello.

**Si ricorda come la mappa privata “\_roomsEachAppeal” (della classe SpecificYearCourse) abbia come:**

* **key: un numero intero che consente di distinguere i diversi appelli e che va da 0 a 3 per un totale così di quattro appelli (0, 1, 2, 3);**
* **value: per ogni key è presente un vector di interi; ciascun vector contiene nei suoi diversi campi gli id delle aule che ospitano l’esame;**
* infine va a settare la mappa privata “\_howManyTimesIAmAssignedInASession” in particolare a seconda della sessione (o “winter” o “summer” o “autumn” rappresentate da un intero) si salvano le date degli appelli presenti in quella sessione.

**Si ricorda come la mappa privata “\_howManyTimesIAmAssignedInASession” (della classe SpecificYearCourse) abbia come:**

* **key: il tipo di sessione (rappresentata da un intero 0=invernale, 1=estiva, 2=autunnale);**
* **value: per ogni key è presente un vector di oggetti della classe Date; ciascun contiene nei suoi diversi campi le date dell’esame;**
* **assignAppealsToProf**

Prototipo: void University::assignAppealsToProf(std::string idCorso,

std::string appeal, int startHour, int numSlots,

std::vector<int> allProfsPerYearCourse)

Con il metodo assignAppealsToProf si provvede ad arricchire la mappa privata “\_professors” (della classe University costituita da oggetti Professor) con le informazioni riguardanti l’esame che vengono passate mediante i seguenti parametri:

* “idCorso”, stringa contenente l’id del corso;
* “appeal” ovvero la stringa con all’interno la data dell’appello;
* “startHour” ovvero l’ora di inizio dell’esame;
* “numSlots” nonché l’intero rappresentante il numero di slot orari necessari per l’esame;
* “allProfsPerYearCourse”, il vector di stringhe con le matricole di tutti i docenti di uno specifico anno per tutti i corsi paralleli;

Mediante un for si scandaglia l’intero vector “allProfsPerYearCourse “affinchè per tutti i docenti del corso in analisi si proceda secondo quanto segue; siccome ogni oggetto della classe Professor presenta la mappa privata “\_examToDo” occorre quindi per ogni professore puntato dal primo for andare ad analizzare tutti gli slot orari che occupa l’esame in questione, ovvero tutte le fasce orarie che vanno da “startHour” a “startHour”+2\*“numSlots”; al fine di realizzare ciò ci si avvale di un secondo for annidato nel primo.

**Si ricorda come ogni docente ha una mappa privata “\_examToDo” (della classe Professor) nella quale si tiene traccia degli esami che un docente possiede e del giorno e dell’ora in cui avvengono:**

* **key: una stringa con la data dell’appello nel formato yyyy-mm-dd;**
* **value: per ogni key è presente una ulteriore mappa la quale presenta come key l’ora di inizio di uno slot e come value l’id del corso che occupa lo slot; si segue lo schema: data → slot → id\_esame;**

Così all’interno della funzione assignAppealsToProf grazie ad un ulteriore metodo, **addNewExam** (della classe Professor), si procede con l’aggiungere un esame alla mappa privata degli esami che un docente deve preparare, “\_examToDo”. In particolare tale metodo distingue due casi: se “\_examToDo” contiene già la mappa con la data dell’esame o meno; in caso negativo provvede alla creazione della mappa diversamente nella mappa già esistente si limita ad assegnare l’id dell’esame ad un determinato slot orario.

* **assignAppealsToClassroom**

Prototipo: void University::assignAppealsToClassroom(std::string appeal,

int startSlotHour, std::string classrooms, int numSlot)

Con il metodo assignAppealsToClassroom si provvede ad arricchire la mappa privata “\_classrooms” (della classe Univeristy costituita da oggetti Classroom) con le informazioni riguardanti l’esame che vengono passate mediante i seguenti parametri:

* “appeal” ovvero la stringa con all’interno la data dell’appello;
* “startsSlotHour” ovvero l’ora di inizio dell’esame;
* “classrooms”, la sringa con le aule che ospiteranno l’esame divise dal carattere separatore “|”;
* “numSlot” nonché l’intero rappresentante il numero di slot orari necessari per l’esame;

Con un for ci si assicura di applicare ad ogni aula che ospiterà l’esame il metodo **setDisavailability**; quest’ultimo a sua volta va a settare la mappa privata “\_disavailableRooms“ (posseduta dagli oggetti della classe Classroom) con la data e lo slot occupato dall’esame al fine di tener traccia del fatto che quella determinata aula in quel giorno e orario risulta occupata. Se l’esame necessita di più fasce orarie significherà che per quella data saranno presenti più chiavi con la medesima data ma slot diversi e consecutivi.

**Si ricorda infatti come la mappa privata “\_disavailableRooms” (della classe Classroom) abbia come:**

* **key: la stringa con data e ora di inizio dell’esame;**
* **value: un valore intero; risulta importante solo che esista la chiave per attestare il fatto di essere occcupata in quel giorno e ora;**
* **readAllMinDistanceRequest**

Prototipo: void University::readAllMinDistanceRequest()

Questo metodo si occupa della lettura del database tutte\_le\_richieste.txt contenente tutti i gap voluti da tutti i professori nei diversi anni accademici.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline.

Come già accaduto piu volte si passa la riga letta alla funzione splittedLine al fine di poter operare singolarmente sulle seguenti informazioni:

* “infoRequest [0]”: anno accademico; tale stringa viene passata al metodo **getAcStartYear** che restituisce l’intero corrispondente all’anno di inizio, “acStartYear”;
* “infoRequest [1]”: matricola del docente;
* “infoRequest [2]”: id del corso;
* “infoRequest [3]”: gap richiesto dal docente; questa stringa viene somministrata al metodo **checkedStoi** il quale controlla che tale informazione sia costituita esclusivamente da cifre, provvedendo in tal caso alla restituzione dell’equivalente intero;

Solo dopo aver estratto le diverse informazioni si provvede a caricare il gap di giorni richiesto dal docente nell’oggetto SessionYear presente nella mappa “\_acYearSessions” (della classe University) in corrispondenza dell’anno “acStartYear”. In particolare l’operazione di caricamento viene affidata al metodo **addProfGap** che va a settare opportunamente la mappa privata “\_gapProfs” (dell’oggetto SessionYear recuperato).

**Si ricorda infatti come la mappa privata “\_gapProfs” (della classe SessionYear) abbia come:**

* **key: la stringa con matricola\_docente - id\_corso;**
* **value: il gap corrispondente;**
* **setMinDistance**

Prototipo: void University::setMinDistance(std::string acYear, std::string name)

Metodo lanciato in risposta al comando: -s set\_min\_distance <anno\_accademico> <file\_richieste>.

Questo metodo si occupa del salvataggio in memoria di nuove distanze minime richiesti dai docenti per i loro esami (ulteriori a quelle presenti nel file “tutte\_le\_richieste.txt”).

Tali gap sono riportati nel file il cui nome viene passato nel parametro “fin” del metodo stesso.

Dopo aver aperto il file in modalità lettura ed eseguiti i relativi controlli, al fine di lanciare una eccezione in caso di mancata apertura del file stesso, grazie all’uso di un while ci si assicura di leggere tutte le righe del file, su ognuna delle quali vengono eseguite le operazioni di seguito spiegate. In particolare la lettura di ogni riga viene fatta mediante il metodo getline. Lettura che viene eseguita in modo analogo a quella effettuata nel metodo precedente, readAllMinDistanceRequest.

Si riportano di seguito le diverse informazioni presenti nella riga e separate dalla funzione splittedLine:

* “infoRequest [0]”: matricola del docente;
* “infoRequest [1]”: id del corso;
* “infoRequest [2]”: gap richiesto dal docente; questa stringa viene somministrata al metodo **checkedStoi** il quale controlla che tale informazione sia costituita esclusivamente da cifre, provvedendo in tal caso alla restituzione dell’equivalente intero;

Come richiesto da consegna, solo nel caso in cui la distanza minima risulta essere maggiore o uguale a 14 giorni si procede con le consecutive istruzioni diversamente si continua con la letura della prossima riga.

In seguito le diverse informazioni presenti in ogni riga vengono sottoposte a premurosi controlli provvedendo, qualora questi ultimi non dovessero andare a buon fine, a settare al valore false un apposito flag booleano, “doDbWrite” e a tener traccia dell’errore verificatosi e della riga in cui è stato individuato in una apposita stringa detta “error”.

In particolare si controlla che:

* il corso, come anche il docente, esistano, ovvero siano presenti rispettivamente nelle apposite mappe private: “\_course” e “\_professor”;
* il corso venga tenuto da quel determinato docente nell’anno accademico in questione; per portare a termine tale controllo ci si avvale del metodo **profHaveThisCourse** (della classe Course). A sua volta tale metodo recupera in primis le matricole di tutti i professori che hanno insegnato quel corso accedendo alla mappa privata “\_courseOfTheYear” e usando come chiave di ricerca l’anno in questione. Solo in seguito ricerca tra costoro la matricola del docente in analisi provvedendo alla restituzione di un bool pari a true in caso di presenza;
* le sessioni di esame per quel determinato anno accademico siano già state definite,

Solo nel caso tutti i precedenti controlli siano andati a buon fine si provvede a caricare il gap di giorni richiesto dal docente nell’oggetto SessionYear presente nella mappa “\_acYearSessions” (della classe University) in corrispondenza dell’anno “acStartYear”. In particolare l’operazione di caricamento viene affidata al metodo **addProfGap** (spiegato nel metodo readAllMinDistanceRequest).

Ogni ciclo del while si ultima con l’incremento del contatore “line\_counter” che “tiene a mente” il numero di righe sino a quel momento lette.

Terminato il while si controlla ancora una volta il valore del flag “doDbWrite”:

* se pari a true, ovvero se tutti i precedenti controlli sono andati a buon fine, si provvede alla riscrittura del database delle distanze minime richiesti dai docenti per i loro esami (“tutte\_le\_richieste.txt”) in quanto ulteriori gap sono stati appena aggiunti; riscrittura affidata al metodo **minDistanceRequestWrite.**
* se pari a false, ovvero se almeno un controllo non è andato a buon fine, allora si procede con il sollevamento di una eccezione che verrà opportunamente catturata e gestita dal main (come verrà spiegato di seguito nella sezione “main”);

**Si noti pertanto come si è rispettata la medesima scelta adottata per le procedure di “aggiunta”, ovvero si noti come anche in questo metodo, in fase di progettazione, si è preferito proseguire con la riscrittura del file solo nel momento in cui tutti gli studenti contenuti nel file non presentano errori.**

* **minDistanceRequestWrite**

Prototipo: void University::minDistanceRequestWrite()

Tale metodo si occupa della riscrittura del file database tutte\_le\_richieste.txt contenente tutti i gap voluti da tutti i professori nei diversi anni accademici; si dichiara quindi un file detto “fout” che, mediante il metodo open, apre il file database nelle modalità out (affinché si possa scrivere su tale file) e trunc (affinché eventuali contenuti già presenti nel file vengano sovrascritti).

Mediante l’ausilio di un for e di un iteratore che punta ad oggetti della classe SessionYear si procede col scandagliare tutta la mappa “\_acYearSessions”; per ogni anno accademico si provvede a salvare nei diversi campi del vector “profsOfSpecificSessionYear” tutti i docenti che hanno richiesto un gap in quell’anno; in particolare ci si affida al metodo **getProfsOfGapProfsString** (della classe SessionYear) che accedendo alla mappa privata “\_gapProfs” (della classe SessionYear) resituisce, di tutti i docenti che hanno richiesto un gap, la matricola del professore, l’id del corso e il relativo gap (rispettando il formato definito per il database);

Con un ulteriore for interno al primo si fa in modo di scrivere nel database ogni singolo campo del vector “profsOfSpecificSessionYear” occupandosi di andare a capo prima di passare al campo successivo;

Il metodo si conclude con la chiusura del file.

* **checkVersioningRequest FA PARTE DI UNIVERSITY**

Prototipo: bool University::checkVersioningRequest(int newVersion)

* **revertChanges FA PARTE DI UNIVERSITY**

Prototipo: void University::revertChanges(int newVersion)

* **revertChanges2to1 FA PARTE DI UNIVERSITY**

Prototipo: void University::revertChanges2to1()

* **revertChanges3to2 FA PARTE DI UNIVERSITY**

Prototipo: void University::revertChanges3to2()

**VEDERE QUALI DI QUESTI SONO STATI Già DESCRITTI E QUALI DA DESCRIVERE:**

* **fillGroupedCourse**

Prototipo: void University::fillGroupedCourse

(std::vector<std::string> &idGroupedLetti, std::string &idCourse,

std::string &acYear, int line)

Questo metodo

* **requestChanges**

Prototipo: void University::requestChanges(std::string acYear, std::string fin)

Metodo lanciato in risposta al comando: -s request\_changes <anno\_accademico> <file\_corsi\_da\_spostare>.

* **ifThereAreAlreadyCoursesFillYYSemesterVar**

Prototipo: void University::ifThereAreAlreadyCoursesFillYYSemesterVar

(StudyCourse &sCourse)

* **whatIsMyStudyCourse**

Prototipo: int University::whatIsMyStudyCourse(Course &courseToConsider,

int acStartYear)

* **controlAGAINGroupedCoursesDifferentCds\_Reciprocy**

Prototipo: bool University::controlAGAINGroupedCoursesDifferentCds\_Reciprocy()

* **removeThisAppealAndGroupedInfo**

Prototipo: void University::removeThisAppealAndGroupedInfo(int acYear,

std::string idCourse, int numSession, int numAppeal, Date& date, int& startSlot, std::map<std::string,std::vector<int>>& classrooms)

* **reassignThisAppealInfo**

Prototipo: void University::reassignThisAppealInfo(int acYear, std::string idCourse, int numSession, int numAppeal, Date date, int startSlot,

std::map<std::string, std::vector<int>> classrooms)

RICORDARE: NEI COSTRUTTORI VENGNO FATTE LE READ IN MODO CHE QUANDO INIZIA UN COMANDO PRIMA MI PORTO IN CASA LE INFO DEI DB E DOPO FACCIO QUELLO CHE IL COMANDO MI CHIEDE perché ALLA FINE DI OGNI COMANDO LE VARIABILI SI SVUOTANO

**Rivedere quando me lo dice Andrea**:

* add: course e StudyCourses
* update: course e StudyCourses

**Aggiungere metodi intermedi nella gestione università**

**Guardare file con cose sistemate**

**COSE DA MODIFICARE**

1. ho mess nella generateThisSession che il controllo delle 6 ore va fatto solo se è un secondo appello e quindi questa funzione che mi avevi chiesto serve per capire se ci sono 6 ore tra l'ultimo slot del primo appello e il primo slot del secondo appello che voglio assegnare



1. ho aggiunto labOrClass







1. rivedere il metodo generateThisSession e checkAvailableProf…. E quelli intenri perché andrea li ha modificati per permettere che i raggruppati vengano TUTTI sistemati negli stessi slot orari nello stesso giorno

**DESCRIZIONE DETTAGLIATA PROGETTO – MAIN**

Il main prevede come prima operazione quella di istanziare un oggetto della classe University, detto “poliTo”.

All’atto della dichiarazione dell’oggetto “poliTo” viene chiamato in modo implicito il costruttore della classe Universty (**University::University()** )il quale procede con il lanciare i diversi metodi che si occupano della lettura dei database:

readVersion, readStudents, readProfessor, readClassroom, readStudyCourse, readCourse, readSessionAcYear, readProfsNoAvailability, readCourseNotActive, readStudyPlan, readPassedAppeals, readAllExamAppeals, readAllMinDistanceRequest.

Lo stesso costruttore provvede, metodo per metodo, anche al sollevamento di una eccezione nel caso in cui una funzione di esse dovesse riscontrare uno degli errori citati nelle operazioni di read esposte in precedenza.

Successivamente si procede con il controllare che il numero di parametri siano almeno tre. I parametri possono essere anche quattro o più (a seconda del tipo di comando lanciato dal’utente) ma sicuramente saranno almeno i tre seguenti:

* 1. “argv [0]” contiene il nome del programma stesso;
  2. “argv [1]” contiene la stringa che rappresenta la prima parte del comando richiesto;
  3. “argv [2]” contiene l nome del file indicato dall’utente;

Nel caso in cui il numero di parametri passati da linea di comando siano minori di 3 si procede con il sollevamento di una eccezione al fine di informare l’utente della carenza di informazioni passate al programma.

Nel caso in cui tale controllo sia andato a buon fine si chiama una funzione dichiarata nel main detta **program** che riceve come parametri l’oggetto “poliTo” della classe University e il vettore “argv” con i comandi letti da linea di comando. Questa funzione program al suo interno se avvale dell’aiuto di una seconda funzione, sempre interna al main, detta **returnCode**; quest’ultima in particolare riceve come parametro il vettore “argv” e restituisce un intero corrispondente al comando inserito dall’utente da terminale ovvero corrispondente all’attuale contenuto di “argv [1]” ed eventualmente “argv [2]”. Si noti come anche per questa funzione, returnCode, sia stato definito un elenco di etichette assegnate a dei valori interi costanti al fine di migliorare la leggibilità del codice stesso preservando la rapidità nella comprensione del codice e del flusso generale delle operazioni eseguite. Il valore intero restituito dalla funione returnCode viene salvato in una apposita variabile detta “code”. A questo punto mediante uno switch relativo all’intero “code” si distinguono i diversi metodi da richiamare corrispondenti ai comandi richiesti dall’utente.

In particolare in ogni ramo dello switch si provvede inizialmente ad eseguire il metodo corrispondente al comando passando al metodo le diverse informazioni scritte dall’utente da linea di comando e presenti nei campi del vettore “argv”, a partire da “argv [2]”; dalle descrizioni dei metodi implementati si è notato come questi ultimi possano sollevare delle eccezioni, nel momento in cui dovessero riscontrare delle irregolarità di qualsiasi genere, motivo per cui i metodi stessi vengono eseguiti all’interno di una try. Nel caso in cui dovessero sollevare una eccezione infatti questa verrebbe “catturata” e presa in esame nel catch adiacente; all’interno del catch si recupera il messaggio dell’eccezione sollevata che viene annotato in una stringa detta “strToAddToLog” affiancando così agli errori che hanno impedito la corretta esecuzione del metodo anche il messaggio di mancata esecuzione.

Una volta completata la stringa “strToAddToLog“ con le informazioni necessarie si chiama il metodo **appendToLog** della classe SystemLog volto ad aggiungere nella variabile privata “\_log” Il contenuto della stringa “strToAddToLog” ed a settare a true il booleano privato “\_notOK”. Il ramo catch termina con il porre a false il flag “isOk”.

Nel caso in cui invece non siano state sollevate eccezioni ovvero nel caso in cui “isOk” sia pari a true si provvede invece a salvare nella stringa “strToAddToLog” il messaggio di avvenuta esecuzione del metodo in maniera corretta; messagio che viene passato anche in questo caso al metodo **appendToLog** della classe SystemLog che, come per l’altro scenario, provvede ad aggiungere nella variabile privata “\_log” Il contenuto della stringa “strToAddToLog” ed a settare questa volta al valore false il booleano privato “\_notOK”.

Terminata la funzione program si ritorna nel main dove si distinguono i due possibili scenari verificatesi:

* Se il metodo non é stato eseguito correttamente, ovvero se la variabile privata “\_notOk” della classe SystemLog è true (valore booleano recuperato mediante l’apposito metodo **errorsOccured** della classe SystemLog) allora si procede a stampare sullo stream di errore, ovvero su “cerr”, i messaggi di errore riscontrati sino ad ora e raccolti premurosamente, come visto in precedenza, nella stringa privata “\_log” della classe SystemLog; in particolare il contenuto della stringa privata viene recuperato con l’apposito metodo getLog della medesima classe, SystemLog. Si ricorda inoltre come “cerr” sia un oggetto della classe ostream che rappresenta e raccoglie gli stream di errore volti ad informare l’utente.
* Se il metodo é stato eseguito correttamente, ovvero se la variabile privata “\_notOk” della classe SystemLog è false allora si procede a stampare sullo stream di output, ovvero su “cout”, il messaggio di corretta esecuzione del comando richiesto; messagio raccolto premurosamente, come visto in precedenza, nella stringa privata “\_log” della classe SystemLog; in particolare il contenuto della stringa privata viene recuperato con l’apposito metodo getLog della medesima classe, SystemLog.

Si riportano di seguito le spiegazioni dei metodi interni delle altre classi che fino ad ora non sono stati presentati al fine di non appesantire la lettura.

Classe Date:

* **Date::Date(int year, int month, int day)**

Nel momento in cui un oggetto della classe Date viene dichiarato passando tre interi come parametri interviene tale costruttore che consente di settare con i valori passati come parametro i dati privati della classe ovvero: “\_day”, “\_month”, “\_year” (avvalendosi del metodo **setDate**).

Infine provvede anche a settare il vector privato “\_weekday” con le stringhe contenenti i nomi dei giorni della settimana.

* **void Date::setDate(int yy, int mm, int dd)**

Grazie a questo metodo è possibile, per gli oggetti della classe Date, settare i dati privati “\_year”, “\_month” e “\_day” provvedendo a lanciare una eccezione nel momento in cui:

* l’anno è al di fuori del range tra 1900 e 2100;
* il mese non è compreso tra 1 e 12;
* se il giorno non è compreso tra 1 e l’ultimo del mese tenendo conto anche del mese (grazie al vector “\_days” che tiene traccia del numero massimo di giorni mese per mese) e del fatto che l’anno possa essere bisestile (controllo affidato al metodo **leapYear**);
* **Date& Date::operator++( )**

Si tratta dell’overloading dell’operatore “++” usato come prefisso; in tal caso la data viene incrementata prima della valutazione in una eventuale espressione; pertanto l’espressione verrebbe determinata considerando il valore incrementato; al suo interno l’incremento vero e proprio delle variabili private è affidato al metodo **helpIncrement**;

* **Date Date::operator++(int)**

Si tratta dell’overloading dell’operatore “++” usato come postfisso; in tal caso la data viene incrementata dopo la valutazione in una eventuale espressione; pertanto l’espressione verrebbe determinata considerando il valore NON incrementato; al suo interno l’incremento vero e proprio delle variabili private è affidato al metodo **helpIncrement**;

* **Date& Date::operator+=(unsigned int additionalDays)**
* **bool Date::leapYear(int testYear)**

Questo metodo controlla se l’anno passato come parametro è bisestile o meno ritornando rispettivamente true o false;

* **bool Date::endOfMonth(int testDay) const**

Questo metodo applicato ad una certa data determina se il giorno passato come parametro è l’ultimo del mese o meno restituendo rispettivamente true o false;

* **void Date::helpIncrement( )**

Questo metodose rivolto ad un oggetto della classe Date consente di incrementare i dati privati affinchè assumano i valori della data successiva controllando anche se la data, rispetto a cui è applicato, è l’ultima del mese provvedendo in questo caso ad incrementare anche il mese.

* **unsigned int Date::getMonth() const**

Questo metodo se applicato ad un oggetto della classe Date restituisce il valore privato del suo mese sotto forma di const (affinchè non venga modificato in quanto tale operazione spetterebbe al metodo setDate).

* **unsigned int Date::getDay() const**

Questo metodo se applicato ad un oggetto della classe Date restituisce il valore privato del suo giorno sotto forma di const (affinchè non venga modificato in quanto tale operazione spetterebbe al metodo setDate).

* **ostream& operator<<(ostream& output, const Date& d)**

Si tratta dell’overloading dell’operatore “<<” che se applicato ad un oggetto della classe Date consente di stampare su uno stream di output la data nel formato yyyy-mm-dd.

* **istream& operator>>(istream& input, Date& d)**

Si tratta dell’overloading dell’operatore “>>” che se applicato ad un oggetto della classe Date consente di leggere da uno stream di input una data andando a settare i dati privati dell’oggetto stesso a cui è rivolto (avvalendosi in particolare del metodo **setDate** già spiegato).

* **bool Date::operator>(const Date & date) const**

Si tratta dell’overloading dell’operatore di confronto maggiore “>” che ritorna il booleano true se la data posta alla sua sinistra è successiva a quella posta alla sua destra, diversamente restituisce il booleano false.

* **bool Date::operator<(const Date & date) const**

Sitratta dell’overloading dell’operatore di confronto minore “<” che ritorna il booleano true se la data posta alla sua sinistra è precedente a quella posta alla sua destra, diversamente restituisce il booleano false.

* **std::string Date::getWeekDay()**

Questo metodo permette di determinare il giorno di una settimana a cui corrisponde la data chiamante il metodo.

* **bool Date::checkGapGiven(int weeks, Date d)**

Questo metodo, come anticipato, si occupa di controllare se tra la distanza l’oggetto della classe Date che invoca il metodo e il secondo parametro (Date “d”) intercorrono il numero di settimane passato come primo parametro (alias int “weeks”).

* **Date Date::add(int daysToAdd)**

Questo metodo applicato ad un oggetto della classe Date restituisce la data che si trova dopo un numero di giorni indicato dal parametro “daysToAdd”.

* **Date::Date(const std::string &date)**

Nel momento in cui un oggetto della classe Date viene dichiarato passandoo una stringa come parametro interviene tale costruttore che consente di settare con i valori presenti nella stringa i dati privati della classe ovvero: “\_day”, “\_month”, “\_year” (avvalendosi del metodo **setDate**).

* **unsigned int Date::getYear() const**

Questo metodo se applicato ad un oggetto della classe Date restituisce il valore privato del suo anno sotto forma di const (affinchè non venga modificato in quanto tale operazione spetterebbe al metodo setDate).

* **std::string Date::toString()**

Questo metodo se applicato ad un oggetto della classe Date restituisce la stringa contente la data nel formato yyy-mm-dd;

* **bool Date::isEqual(const Date compare)**

Questo metodo consente di confrontare la data chiamante il metodo con la data passata come parametro; in particolare se le due date coincidono allora restituisce un booleano pari a true diversamente ritorna false.

* **int Date::whatIsTheGap(Date& date)**

Questo metodo restituisce un numero intero rappresentante la distanza in giorni tra la data chiamante il metodo stesso e l’oggetto Date passato come parametro.

* **bool Date::operator<=(const Date & d) const**

Sitratta dell’overloading dell’operatore di confronto minore o uguale “<=” che ritorna il booleano true se la data posta alla sua sinistra è precedente o coincidente a quella posta alla sua destra, diversamente restituisce il booleano false.

* **bool Date::operator>=(const Date & d) const**

Si tratta dell’overloading dell’operatore di confronto maggiore o uguale “>=” che ritorna il booleano true se la data posta alla sua sinistra è successiva o coincidente a quella posta alla sua destra, diversamente restituisce il booleano false.

* **bool Date::operator==(const Date& compare) const**

Si tratta dell’overloading dell’operatore di confronto sull’uguaglianza “==” che ritorna il booleano true se la data posta alla sua sinistra è uguale a quella posta alla sua destra; se diverse restituisce il booleano false.

* **Date Date::subtract(int daysToSubtract)**

Questo metodo applicato ad un oggetto della classe Date restituisce la data che si trova prima di un numero di giorni pari a quello indicato dal parametro “daysToSubtract”.

* **Date Date::operator--(int)**

Si tratta dell’overloading dell’operatore “--” usato come postfisso; in tal caso la data viene decrementata dopo la valutazione in una eventuale espressione; pertanto l’espressione verrebbe determinata considerando il valore NON decrementato; al suo interno l’incremento vero e proprio delle variabili private è affidato al metodo **helpDecrement**;

* **void Date::helpDecrement()**

Questo metodose rivolto ad un oggetto della classe Date consente di decrementare i dati privati affinchè assumano i valori della data precedente controllando anche se la data, rispetto a cui è applicato, è la prima del mese provvedendo in questo caso a decrementare anche il mese.

* **Date &Date::operator--( )**

Si tratta dell’overloading dell’operatore “--” usato come prefisso; in tal caso la data viene decrementata prima della valutazione in una eventuale espressione; pertanto l’espressione verrebbe determinata considerando il valore decrementato; al suo interno l’incremento vero e proprio delle variabili private è affidato al metodo **helpDecrement**;

|  |  |  |  |
| --- | --- | --- | --- |
| STUDENTI | SISTEMA OPERATIVO | COMPILATORE | VERSIONE COMPILATORE |
| Andrea Chiapparo |  |  |  |
| Luca Montangero |  |  |  |
| Ricardo Maria Galiano |  |  |  |

Andrea Chiapparo

Luca Montangero

Riccardo Maria Galiano