Politecnico di Torino

A.a. 2022/2023

Relazione del progetto: “Gestione università e programmazione esami”

Corso di Algoritmi e Calcolatori

Docenti:

Savino Alessandro
Maunero Nicolò
Roascio Gianluca

Studenti:

S228083 Brignone Luca

Indice:

Prologo

Le necessità espresse dalle specifiche di progetto, prevedono l'implementazione di un programma capace di creare e gestire in modo automatico una "base dati", con un numero minimo di interazioni da parte dell'utente, il quale può interfacciassi unicamente mediante un numero ristretto di comandi preventivamente definiti. Inoltre con le medesime modalità di accesso il sistema deve permettere l'uso dei dati immessi per la creazione automatica di una calendarizzazione degli esami immessi. Alle precedenti indicazioni si aggiungono una serie di vincoli interni che che il sistema dovrà verificare ed opportunamente segnalare all'utente in caso vengano violati.

Struttura

Per migliorare le caratteristiche di manutenibilità il progetto è stato ripartito secondo le funzioni svolte dalla particolare sezione permettendo così a ciascuna di essere indipendente e modulare rispetto alle altre. Questa compartimentazione permette anche di differenziare le funzioni di gestione, creazione, riempimento e controllo dei dati e strutture dal mantenimento dei dati e l'interazione con l'utente Quest'ultima funzione, come verrà descritto in modo più dettagliato nel seguito, sarà prerogativa del main.
Per tutto il progetto si è fatto largo uso dei contenitori messi a disposizione dalla libreria **stl**, oltre ad alcune funzioni della libreria **<algorithm>**, a queste si aggiungono le librerie base per l'input/output, sia da tastiere **<ostream>** che da file **<fstream>**, e altre come la **<string>**.
Di seguito si andrà a descrivere le modalità di implementazione del progetto, dandone una visione discorsiva, invece in appendice si può trovare una sua schematizzazione con le strutture, i tipi e i metodi usati.

Main

Come già accennato sopra, il main è l'unica interfaccia del programma con l'utente, esso si preoccupa di interpretare correttamente i comandi dati per argomento all'avvio del programma e richiamare le funzioni opportune allo svolgimento dell'azione voluta.
Come prima fase il comando viene identificato dal numero di argomenti che compongono l'argomento al lancio del programma, e mediante un primo **switch-case**, ci si occupa di assegnare le parti che compongo il comando a variabili specifiche che verranno ulteriormente elaborate a seconda delle necessità. In seconda battuta il primo campo dell'argomento, che rappresenta l'identificatore del comando, viene usato per "spostarsi" lungo una "serie" di **switch-case** nidificati, e in particolare si usa il primo carattere per identificare la tipologia di comando:

-a —> aggiunta (add)

-u —> aggiornamento (update)
-i —> inserimento (insert)
-s —> impostazione (set)
-g —> generazione sessioni d'esame (generate)

e il secondo carattere per identificare la tipologia di file o stringa/e da gestire:

s —> studenti
d —> docenti
a —> aule
c —> corsi
f —> corsi di studio
current_a —> definizione periodi d'esame
set_availability —>definizione periodi di indisponibilità dei docenti

Nel caso la prima o la seconda componente dell'identificatore del comando non vengano riconosciute è prevista la generazione di una stringa di errore che è possibile ritrovare nel case "**default**".
Se l'identificazione del comando ha avuto successo ci si troverà all'interno del corrispettivo **case**, dove vengono richiamate le funzioni o metodi per creare, aggiornare o semplicemente leggere la base dati oppure schedulare gli esami.
Nel main vengono inoltre mantenuti tutti i "contenitori", derivati dal popolamento del database, in particolare tutti i dati sono memorizzati in liste di "tipo_classe" questo per semplificare, secondo quella che è stata l'impostazione iniziale del progetto, le operazioni di aggiunta, inserimento e aggiornamento a cui possono essere sottoposti i dati. Unica eccezione è il mantenimento delle date che compongono le sessioni d'esame, per le quali si è utilizzato una mappa, avente per chiave l'anno accademico, e come secondo parametro un vettore contenente le date delle sessioni d'esame.

Classi primarie

Person

Rappresenta la classe che mantiene le caratteristiche base, quali nome, cognome, mail ed id, comuni alle classi "**Student**" e "**Professor**", inoltre predispone i metodi per la scrittura e l'accesso a tali dati prevedendo nel caso particolare della generazione dell'id l'uso di un metodo "**virtuale**" che verrà implementato con le caratteristiche specifiche definite dalle classi derivate.

Professor

Sfrutta l'ereditarietà sulla classe "**Person**" per definire le sue caratteristiche base e introduce dei dati specifici, quali le indisponibilità. Per quanto riguarda l'accesso ai dati, dà implementazione al metodo "virtuale" per la generazione

dell'identificativo oltre a implementare i metodi per la scrittura, aggiornamento e accesso alle indisponibilità.
La generazione dell'id è necessaria nel caso il file di input non sia di database, per cui si prevede che la generazione degli id sia data dall'incremento dell'ultimo id disponibile o a partire dall'id "d000001".
Le indisponibilità a loro volta sono divise in due contenitori specifici a seconda della loro funzione, il primo memorizza il contenuto del file "**db_indisponibilità.txt**", quindi le indisponibilità settate esternamente dall'utente, mentre il secondo mantiene le indisponibilità derivate dal posizionamento di un corso nella programmazione degli esami.

Student

Anch'esso eredita i metodi della classe "**Person**", ma in questo caso non si aggiungono altri dati ad'hoc, invece si dà implementazione al metodo virtuale per la generazione degli id.

Classroom

La classe classroom memorizza le caratteristiche utili ad identificare l'aula, come il suo id e il nome, oltre alle caratteristiche "tecniche", come la sua tipologia, la capienza e la capienza in condizione d'esame.
A questi si accompagnano tutti i metodi necessari per effettuare la scrittura, l'aggiornamento e l'accesso dei dati da essa contenuti.

AssociateProfessor

Il "professore associato" è la classe che rappresenta il professore che è possibile trovare all'interno dell'organizzazione di un corso. Esso mantiene un puntatore al professore, memorizzato nella **lista** "**databaseOfProfessors**" identificato mediante il suo id, oltre all'indicazione se è titolare della versione del corso e alle ore di lezione, esercitazione e laboratorio che gli sono state assegnate.

Course

Questa classe memorizza gli attributi che differenziano i corsi tra loro, come il suo id, l'anno accademico, il titolo, i CFU, il numero di versioni, oltre alle ore destinate a lezione, esercitazione e laboratorio. A queste si aggiungono una serie di informazioni legate alla programmazione dell'esame, quali il tipo d'esame, l'aula in cui si dovrà svolgere l'esame, le tempistiche, ovvero tempo di entrata, di uscita e durata dell'esame, oltre a una serie di corsi ad esso raggruppati, per cui sarà necessario procedere con una programmazione negli stessi orari.

Per quanto riguarda la tipologia di esame si è optato per la definizione di un **enumerativo** per l'identificazione del tipo d'esame e di un corrispondente vettore di stringhe per la stampa.
Alle precedenti informazioni bisogna ancora aggiungere una serie di informazioni che distinguono le differenti versioni tra loro, per cui si introduce l'id della versione e una **lista** di professori assegnati a tale versione del corso, l'elemento base che costituisce tale lista è la classe "**AssociateProfessor**" prima descritta.
Si fa notare che per come è stato strutturato il programma, la classe e il modo in cui questa viene mantenuta, in relazione ad altri corsi, si ha che ogni corso è ripetuto per un numero pari al numero di versioni del corso rappresentato, mantenendo le caratteristiche generali e comuni invariate tra le diverse versioni e modificando invece l'id della versione in questione e la lista di professori ad essa associati.

CourseOfStudy

La classe "**CourseOfStudy**" è strutturata per registrare le caratteristiche di un corso di studi, quindi il sui id, la distinzione tra un corso di studi triennale (BS) o magistrale (MS) oltre alla **lista** di id dei corsi di cui è composto, divisi per semestre, e una **lista** di corsi spenti.
La classe mette a disposizione una serie di metodi, come nei precedenti casi, per la scrittura, aggiornamento e lettura dei dati stessi, ma oltre a queste funzionalità offre in particolare due metodi atti a gestire lo spegnimento del corso, ovvero il trasferimento o la copia del corso nella lista dei corsi spenti, o la sua operazione complementare.
Come nel caso dei corsi, anche qui si è utilizzato un **enumerativo** per rappresentare la tipologia di corso di studi.
La struttura in cui vengono memorizzati i corsi, è una **mappa**, la quale ha per chiave il semestre di riferimento, partendo da 0 per il primo semestre primo anno fino a 4 o 6 rispettivamente per l'ultimo semestre del terzo o secondo anno. I corsi spenti fanno riferimento alla stessa struttura, ma vengono memorizzati con chiave negativa (-1) a rappresentare con maggior forza che non si tratta di un semestre "valido". I corsi vengono inizialmente memorizzati in apposite liste temporanee e poi posizionati nella lista del semestre corrispondente.

Date

La classe è stata sviluppata per poter operare indifferentemente su formati data, nel senso stretto del termine, cioè giorno, mese e anno, utile nel gestire le indisponibilità dei professori, gli anni accademici dei corsi e delle sessioni passate per argomento al lancio del programma, ma è anche capace di gestire ore e minuti, così da poter essere usata anche per la programmazione degli

esami, nonché per le indisponibilità temporanee dei professori dovute alla programmazione degli esami stessi.
Questo comporta però anche delle limitazioni, infatti si è dovuta operare una scelta per l'overload degli operatori, privilegiando una o l'altra funzionalità a seconda del contesto in cui sarebbe stato usato più frequentemente l'oggetto.

SessionScheduler

Questa classe si occupa dei popolare le strutture ad uso interno per la programmazione delle sessioni, e procedere alla programmazione stessa degli esami. Per portare a termine tale compito sono stati implementati tre dati interni, una **lista** per mantenere solamente i corsi che devono essere programmati, una **mappa** per organizzare i raggruppamenti e le sessioni dei corsi, ed un contenitore di tipo **vettore** di **pair**, per l'organizzazione del calendario della sessione.
In particolare la **mappa** si compone di una chiave, che rappresenta il semestre di appartenenza dei corsi, e di una **matrice** costituita da un **vettore** di **vettore**, dove il primo indica il raggruppamento, mentre il secondo tiene le informazioni sui corsi raggruppati. Scendendo più nel dettaglio questo secondo vettore è costituito da una **struttura**, la quale memorizza il corso, una **lista** di corsi di studio che fanno uso di tale corso, con annesso booleano per ciascun corso di studi, che rappresenta la violazione del vincolo 1, e un **vettore** di booleani che rappresentano i 4 vincoli che è possibile violare, a questo vettore si aggiunge un quinto booleano, il primo per posizione, che serve a distinguere il rilassamento del vincolo 4 dalla sua violazione. Per quanto riguarda la chiave della mappa, si è tenuto il seguente ordinamento, -1 indica i corsi raggruppati spenti, lo 0 e l'1 rispettivamente il primo e il secondo semestre. Invece il secondo vettore, contenente la struttura, si precisa che questa è ripetuta per ogni versione dei corsi raggruppati nel vettore, e si aggiunge inoltre che il vettore stesso è ordinato per numero di iscritti ai corsi che contiene.
L'ultimo contenitore da analizzare è quello che costituisce il calendario per lo scheduling, che è strutturato con un vettore di pair, esso contiene come primo elemento le aule, e come secondo elemento una **struttura** avente per variabili l'id e versione del corso, l'id del corso di studi e l'id dell'aula, questo servirà a facilitare la stampa de calendario.
Anche in questo caso è opportuno far notare che il vettore è ordinato per capienza delle aule, ovvero secondo il primo elemento del pair. La scelta di utilizzare un vettore di pair anziché una mappa è conseguente alla semplicità di accesso, ma in particolare perché la classe delle aule implementa un overload dell'operatore "<" che non avrebbe permesso tale tipologia ordinamento.
La classe inoltre implementa tutti i metodi necessari a popolare, ed effettuare un controllo sui dati inseriti oltre che per programmare le sessioni, verificando opportunamente la violazione dei vincoli, definiti dalle specifiche del progetto.

A parte trattiamo invece l'output, questo si compone di due metodi che si occupano separatamente di creare e scrivere il file di programmazione delle date e di segnalazione della violazione dei vincoli. Per fare ciò questi fanno uso di una terza struttura, formata dall'id e versione del corso, dall'id del singolo corso di studi, dall'id dell'aula in cui l'esame è stato programmato e un vettore di quattro booleani, che identificano i vincoli violati.
Tale struttura come già indicato è usta nei due metodi di output, ed in particolare serve per creare una lista di suoi elementi, così da ordinarli secondo le specifiche per l'output.

Classi di complemento

ConstConversion

Si tratta di una classe template, la quale è stata sfruttata per convertire gli iteratori, delle liste o vettori usati da const a iteratori normali In alternativa a questa soluzione si sarebbe reso necessario cambiare il tipo di ritorno delle funzioni che fanno uso di questa classe, ma ciò avrebbe comportato accettare che i dati puntati dall'iterare fossero modificabili anche quando non fosse necessario o non voluto. La soluzione di creare un override delle funzioni stesse non sarebbe invece stata praticabile, poiché sarebbe unicamente cambiato il tipo di ritorno.

ErrorHandling

Questa classe è stata creata per gestire gli errori, ma per via della struttura del programma, granate degli errori è ritornata come come codice dalla funzione o metodo e gestito internamente elle stesse in termini di messaggio da restituire all'utente. A quanto detto fa eccezione un solo caso che a causa della necessità dell'utilizzo del ritorno per l'output su file, è stato necessario creare un messaggio di errore da lanciare mediante **throw**.
Nello stesso file si trova altresì un **enumerativo** contenete tutti i codici di errore utilizzati nello svolgimento del progetto.

Funzioni

In questa parte si darà una descrizione generale sulle funzionalità implementate dalle funzioni contenute nei file "**.h/.cpp**", e sulla loro organizzazione in raggruppamenti in base alla funzione espletata.
Si fa notare che i file che hanno come dicitura "**FileHandling**" raccolgono tutte le funzioni per la deserializzazione de file e riempimento di strutture o oggetti temporanei.

AddFileHandling

In questo file vengono espletate tutte le funzioni per la lettura dei file che possono andare ad aggiungere **nuovi** elementi ai file di database, oppure creare i file di database nel caso questi non siano presenti. Inoltre si occupano di leggere i file di database, per qualunque funzione si voglia eseguire su di essi, che sia di aggiunta, update, inserzione o scheduling.

StudentInputFile

Si occupa di della deserializzazione del file degli studenti, esso a seconda del flag "**isDb**" può espletare la doppia funzione di lettura del file "**db_studenti.txt**" (true) o di un qualunque file immesso.
L'implementazione si divide in una prima lettura linea per linea del file, quindi la linea letta viene divisa in parti, in base separatore "**;**". Le diverse componenti della linea passano lungo uno **switch-case**, dove vengono memorizzate nella variabile opportuna in base alla posizione occupata nel file.
Le verifiche sulla correttezza dei campi sono integrate nei metodi richiamati per la memorizzazione dei campi nell'oggetto "**dummyStudent**", si precisa però che per none e cognome è stato considerato ammissibile qualunque serie di caratteri ed in estremo anche la stringa vuota, mentre per la mail è stato fatto solo un controllo posizionale sui "**.**" e "**@**" mediante find dei rispettivi caratteri.

ProfessorInputFile

In maniera del tutto simile alla "StudentInputFile" si occupa di leggere i file relativi ai professori siano essi di database o aggiunta.

ClassroomInputFile

Anche questa funzione compie la lettura dei file di database o qualsiasi file immesso, a seconda del flag "**isDb**" passato alla funzione. Come nei precedenti due casi si ha la creazione di un elemento "**dummy**" il quale viene progressivamente completato con i dati letti linea per linea e deserializzati.

CourseInputFile

In maniera simile alle precedenti deseriallizzazioni anche la lettura dei file dei corsi, siano essi di database o altri file viene eseguito linea per linea con popolazione di un elemento denominato "**dummyCourse**", ma in questo caso la gestione è un po' più complessa.
La stringa viene infatti gestita in due macrosezioni, la prima, evidenziata in giallo nella didascalia a, viene deserializzata in prima lettura mediante lo switch-case, popolando il "**dummyCourse**" . La parte un azzurro invece

viene temporaneamente memorizzata in una stringa, quando si arriva al suo “case”, per essere deserializzata successivamente.

```
<anno_accademico>;<titolo>;<cfu>;<ore_lezione>;<ore_esercitazione>;<ore_laboratorio>;<versioni_in_parallelo>;[{<matricola_titolare>,<codice_versione>,[{<matricolare_prof1>,<ore_lez>,<ore_es>,<ore_lab>},..,{<matricolare_profn>,<ore_lez>,<ore_es>,<ore_lab>}]},...];{<durata_esame>,<t_ingresso>,<t_uscita>,<modalità>,<luogo(A/L)>,<numero_iscritti_esame>};{<id_corso1>,..,<id_corsoN>}
```

a. Esempio per la stringa di aggiunta di un corso per un generico file

Per la deserializzazione della parte azzurra si usa un particolare costrutto che riconoscendo il numero e il tipo di parentesi assegna la posizione corretta della sottostringa ottenuta usando il separatore “**,**”.
Si distinguono ancora due gestioni separate, una per la versione del corso, che viene memorizzato nel “**dummyCourse**”, e per i professori, che vanno a popolare un professore associato dummy, che a sua volta verrà inserito in una lista, e solo al termine della versione verrà assegnato al corso dummy. Questa metodologia comporta che ogni versione abbia un corso a se stante, avente le parti in giallo in comune e le parti in azzurro specifiche per la versione del corso che identifica.
La gestione non è del tutto uguale se il file è “**db_corsi.txt**” per via della disposizione diversa delle stringhe, in particolare le righe che cominciano per “**c**” saranno identificate come inizio del corso da un **if**, mentre quelle che iniziano per “**a**” saranno gli anni accademici per cui è definito il corso, ed anch’esse sono identificate da un if. Per il proseguitò della deserializzazione risulta del tutto simile a quella già descritta per un corso non di database.
Per quanto riguarda i controlli, sono eseguiti all’interno dei metodi della classe per i corsi e della classe per i professori associati, in particolare nella prima si verificherà anche la coerenza sulle ore assegnate a lezione, esercitazione e laboratorio, con la somma delle rispettive ore assegnate ad ogni professore.

CourseOfStudyInputFile

Come nei casi di studenti, professori e aule, si crea e si popola l’oggetto temporaneo “**dummyStudyCourse**”, il quale viene inserito nella lista a formare il database.
I file, siano essi di database o generici sono gestiti allo stesso modo, al più varia il case di partenza, poiché mentre nel database l’id del coso di studi è presente, nel file generico deve essere generato a partire dall’ultimo id valido o nel caso non ve ne siano partire dal primo disponibile.

ExamSessionInputFile

Questa volta si deve distinguere la stringa passata come argomento all'avvio del programma con la lettura del file "**db_periodi_esami.txt**", ciò è ottenuto con l'uso di un flag passato all funzione.
La deserializzazione in questo caso procede in due momenti, prima si ha una divisione della stringa in base agli " " e quindi l'identificazione dell'anno accademico, che è il primo campo, poi la deserializzazione della stringa contenente le date divisa prima per il carattere "**_**" e poi per "**-**", così si memorizzano i campi della data in un oggetto data.
Queste date vendono memorizzate in un **vettore** temporaneo, il quale contenendo tutte le sessioni viene controllata sulla base dell'ordinamento delle date, la durata della sessione, la coerenza rispetto all'anno accademico e l'ordinamento delle sessioni.
A questo punto il vettore in concomitanza con l'anno accademico va a formare un pair che viene inserito nella mappa che costituisce il database delle date d'esame.

ProfessorUnavailabilityInputFile

Anche in questo caso si rende necessaria una distinzione nella lettura del file, infatti la formattazione del file di database prevede una linea che identifica l'anno accademico a cui fanno riferimenti tutti i professori successivi, e che viene memorizzato in un oggetto data, in modo simile, avviene nel caso di un file generico, con il fatto che l'anno accademico è passato come argomento all'avvio del programma, e viene inoltrato alla funzione.
La deserializzazione dei professori avviene in modo uguale per entrambi i tipi di file, quindi con l'identificazione del professore, e l'associazione ad esso di una **lista** di indisponibilità, con lo stesso riconoscimento della data visto per la funzione precedente, tranne per i separatori che sono "**|**" e "**-**".
Le indisponibilità ottenute vengono controllate rispetto a quelle già esistenti, e rispetto al fatto che la data di fine sia dopo quella di inizio.
Se non ci sono stati errori vengono inserite nel professore corrispondente, e se non riesce l'inserimento, dovuto ad una precedente esistenza delle indisponibilità, si procede alla loro cancellazione e aggiornamento.

UpdateFileHandling

In questo set di funzioni si va ad agire su elementi del database già esistenti, andando a modificarne il contenuto secondo i nuovi dati ricevuti.
I controlli sulla validità dei compi sono gli stessi che vengono perforati per l'aggiunta dei campi medesimi, pertanto per una loro descrizione si rimanda all'" **AddFileHandling**".

StudentToUpdateFile

Il file contenente i dati aggiornati viene letto come nella "**StudentInputFile**", con l'unica differenza che in questo caso l'id dello studente deve essere presente. L'id vie quindi cercato nel database, risultando nel ritorno di un **iteratore**, cui campi vengono aggiornati dalla deserializzazione della stringa, fatta dividendo la stringa con il separatore "**;**" , e usando uno **switch-case** per il riconoscimento del campo.
Si fa notare che nell'update si permette di tralasciare dei campi, quindi i campi che risaluteranno vuoti rimarranno inalterati.

ProfessorToUpdateFile

L'update del database dei professori avviene in modo del tutto simile a quanto visto per l'aggiornamento del database degli studenti, e con metodologia e controlli simili a quelli visti per la "**ProfessorInputFile**".

ClassroomToUpdateFile

Anche per quanto riguarda le aule, si prevede la lettura del file di aggiornamento riga per riga, e dalla deserializzazione della stessa si ricava l'id dell'aula da modificare.
Dopo aver effettuato una ricerca dell'aula nel database, si aggiornano i dati della stessa con quelli ottenuti dalla deserializzazione.
Deserializzazione ottenuta dividendo la stringa e con uno **switch-case** identificando i campi corrispondenti, come nel caso precedente è possibile lasciare vuoti alcuni di questi campi, e ciò deve comportare una non alterazione dell'elemento puntato.

CourseOfStudyToUpdateFile

La funzione si propone di eseguire l'update dell'organizzazione di un corso. Questo compito è eseguito introducendo un file che presenta una formattazione simile al file di aggiunta dei corsi di studio, ma per il quale è obbligatorio inserire l'id del corso da modificare. Infatti in base a questo id viene effettuata una ricerca nel database, e se un riscontro è trovato si procede non l'aggiornamento della sua organizzazione.
La nuova organizzazione verrà inserita, semestre per semestre, in una lista temporanea, la quale dopo un controllo sulla validità dei campi da lei contenuti, e un controllo sull'eventuale ripetizione di corsi nell'organizzazione, procede all'inserimento della lista nel semestre appropriato del corso individuato nella fase precedente.
La procedura permette di omettere degli id nell'organizzazione del corso, che verranno ereditati dal database.
Si fa notare che il tipo di corso di studi, BS o MS, non è reso modificabile da questa funzione, come non è possibile modificare direttamente i corsi

spenti, cosa possibile solamente usando il comando di inserimento per i corsi.

InsertFileHandling

In questo file avviene la gestione dell'inserimento di un corso, il quale può esistere già in base dati, risultando in suo aggiornamento dei dati, oppure non risultare ancora present, nel qual caso si ha l'inserimento del corso nella base dati.

CourseToInsertFile

Come esposto sopra l'inserimento è caratteristico solo dei corsi, e prevede due comportamenti distinti a seconda dell'esistenza o meno del corso nel database.
Per distinguere i due comportamenti è necessario che le righe del file da leggere abbiano sia l'id del corso, il quale dovrà essere sempre presente in database, sia l'anno accademico su cui si vuole intervenire.
La lettura del file avviene sempre riga per riga, che viene deserializzata mediante uno **switch-case** per identificare coerentemente i vari campi. A partire dai primi due campi, id ed anno accademico, viene lanciata una **ricerca** sul database, che restituisce un **iteratore** congruo a seconda se entrambi i campi sono presenti o solo l'id è presente.

```
<id_corso>;<anno_accademico>;<attivo/non_attivo>;<versioni_in_parallelo>;[{<matricola_titolare>,<codice_versione>,[{<matricolare_prof1>,<ore_lez>,<ore_es>,<ore_lab>},..,{<matricolare_profn>,<ore_lez>,<ore_es>,<ore_lab>}]},...];{<durata_esame>,<t_ingresso>,<t_uscita>,<modalità>,<luogo (A/L)>,<numero_iscritti>};{<id_corso1>,..,<id_corsoN>}
```

b. Esempio per la stringa di inserzione di un corso per un file generico

Dall'iteratore trovato viene creata una copia temporanea che **eredita** tutti i campi del corso, quindi quest'ultimo verrà modificato dalla restante parte della deserializzazione della riga. Il procedimento della deserializzazione è simile a quello viso per il "**CourseInputFile**", con una divisione secondo la didascalia b, dove la parte in giallo è ottenuta da una deserializzazione diretta secondo il separatore "**;**", mentre la parte in blu è deserializzata in un secondo momento e fa uso di un sistema che riconoscendo il numero e tipo di parentesi assegna una "**profondità**" a ciascun campo Si ottiene così la corrispondenza dei campi con le variabili del professore associato temporaneo che viene impiegato.
Il corso così formato viene messo in una **lista** temporanea in cui vengono aggiunte tutte le versioni dello stesso corso con le modifiche lette.

Adesso la lista temporanea è aggiornato con i nuovi dati per tutte le versioni del corso, pronti per essere trasferite sul database. A questo punto si differenzia la gestione, infatti se il corso esiste come id ed anno accademico, si richiama una funzione di "**fill**", la quale dopo aver operato gli opportuni controlli sui dati aggiornati procede a sovrascrivere i dati nel database con quelli nuovi per quel corso. In alternativa, se esistesse solo il l'id del corso si avrà una nuova istanza del corso, quindi si richiama una funzione di "**insert**" che, dopo aver controllato i dati dei corsi presenti nella lista temporanea, provvede ad inserire il nuovo anno accademico per il dato corso nel database dei corsi, posizionandolo come ultimo inserimento per il corso in questione.
Si fa notare che come per i precedenti casi di update alcuni campi possono essere omessi, per questo motivo è stato utilizzato un metodo del corso che permette di ereditare tutti i componenti del corso stesso, in modo da avere il campo già completo e valido nel caso non sia presente il nuovo doto. Particolare importanza riveste inoltre la gestione del numero di versioni, che possono aumentare o diminuire indipendentemente dal corso da cui si eredita, questo pone un particolare problema sulla capacità di ereditare il corso da database, ma è stato risolto controllando la differenza nel numero di versioni del database e nel caso sia superiore , rispetto al nuovo corso inserito, non si avranno problemi nel completamento dell'aggiornamento, mentre se il numero è inferiore un ulteriore controllo è svolto per verificare la completezza dei nuovi campi inseriti. Nel caso ci sia un incongruenza rispetto all'originale, sia essa sulla mancanza di dati da cui ereditare, sia sui professori e rispettivi orari, un segnale di errore verrà fornito in concomitanza ad una stringa dell'errore accaduto.
Si fa notare un'ultima problematica, per la quale se si vuole impiegare l'ereditarietà dei campi è necessario che siano presenti gli identificatori del campo mancante. Per esempio se il dato mancante è nella parte gialla della didascalia b, o è la parte in blu nella sua totalità, è previsto che sia sempre presente il separatore ";". A questo si aggiunge, per campi più complessi, come quello che definisce le caratteristiche d'esame o i corsi raggruppati, la presenza obbligatoria degli identificatori del campo, quindi "{}", e i separatori interni tra gli elementi, dati dalle "," se si vogliono ereditare elementi interni.
passando invece alla parte in blu devono esserci gli identificatori "[]", perché venga riconosciuta la volontà di applicare una ridefinizione del corso. Al precedente identificatore si aggiunge la presenza delle "{}" per indicare la ridefinizione della versione, questi identificatori sono obbligatori anche nel caso non si voglia effettuare un cambiamento dei dati della versione del corso. Come per i campi dei corsi raggruppati si deve inserire

le “,” con lo stesso pattern indicato nelle specifiche per l’identificazione dei campi. Scendendo nell’organizzazione dei professori con i relativi orari, è nuovamente obbligatorio inserire gli identificatori del campo, che rappresenta i professori assegnati al corso, “[]”, a cui si aggiunge l’identificatore del singolo professore, anch’esso sempre presente. Di nuovo, internamente alle “{}” dovranno essere presenti i separatori degli elementi, rappresentati dalle “,” , e a sua volta lo stesso identificatore dovrà essere usato per separare tra loro le rispettive organizzazioni dei professori, nonché le versioni.
Si ricorda inoltre che a quanto detto fa eccezione l’id del corso e l'anno accademico, che sono elementi di riconoscimento del corso da modificare, e tramite i quali è possibile effettuare una ricerca sul database.

OutputOnDatabaseHandling

In questo file sono state implementate tutte le funzioni che serviranno ad aggiornare e riscrivere i vari file di database, questi infatti vengono sempre aperti in modalità “**truncate**” comportando una cancellazione del contenuto originale, che viene sostituito dai nuovi campi aggiornati o semplicemente riscritto. Per permettere l’esecuzione di tale azione si è fatto largo uso dell’**overload** dell’operatore “**<<**” , ma non sempre è stato sufficiente, quindi sono stati impiegati anche metodi specifici per espletare alcune azioni.

updateStudentDatabaseFile

La funzione cicla con un **while** sulla lista che è il database degli studenti, e per ogni elemento dello stesso, identificato dall’**iteratore**, con l’overload dell’operatore “<<“, esegue la scrittura del pattern già formato secondo le specifiche.

updateProfessorDatabaseFile

Agisce in modo uguale alla stampa del database degli studenti, ma per la lista di database dei professori.

updateClassroomDatabaseFile

La stampa segue le stesso procedimento visto per la stampa del database degli studenti.

updateCourseOfStudyDatabaseFile

Anche per il database dei corsi di studio si procede scrivendo ogni elemento della **lista** di database secondo il pattern stabilito dalle specifiche usando l’operatore “**<<**”.

updateCourseDatabaseFile

In questo caso, la stampa del database dei corsi risulta un po' più complessa, infatti come prima stringa vengono stampate le caratteristiche generali, che non dovranno essere ripetute per i diversi anni accademici per cui è definito il corso. Queste caratteristiche univoche sono l'id, il titolo, i CFU, le ore di lezione, esercitazione e laboratorio, e saranno sempre precedute dall'identificatore "**c**".
Di seguito si avrà la stampa dell'anno accademico a cui si riferisce il corso che si sta stampando, preceduto dall'identificatore di linea "**a**". A questo fanno seguito, finche non saranno esaurite tutte le versioni, l'id della versione e l'organizzazione dei professori. Per ciascuno dei professori si stamperà il suo id seguito dalle ore assegnategli per lezione, esercitazione e laboratorio, rispettando le parentesi di **incapsulamento** definito dalle specifiche. In ultimo vengono stampati i corsi raggruppati, ricavati dall'ultimo elemento, cioè dall'ultima versione per quel corso di quell'anno accademico.
Ora fintanto che non si ha una variazione dell'id del corso su cui ci si sposta, percorrendo la **lista** di database non si avrà la stampa delle caratteristiche generali del corso, mentre fintanto che si avrà la variazione dell'anno accademico si avrà la stampa del nuovo anno accademico preceduto dall'identificatore e il pattern di apertura per l'organizzazione dei professori del "nuovo" corso.

updateExamSessionDatabaseFile

La stampa delle sessioni d'esame è semplificata dal fatto che il database è una **mappa**, quindi già organizzato per chiavi, che in questo caso sono gli anni accademici. Per questo motivo si usa un primo **iteratore** per muoversi lungo la mappa, e stampare l'anno accademico, mentre un secondo, è usato per scorrere il **vettore** dove sono memorizzate le date delle sessioni. Si farà uso solo di uno **switch-case** per inserire il separatore corretto tra la coppia di date che definisce la sessione e le date di sessioni differenti. Invece le date delle sezioni sono stampate dell'operatore "**<<**" contenuto in overload nella classe "**Date**".

updateUnavailabilityDatabaseFile

Per eseguire la stampa delle indisponibilità dei professori, si deve in primo luogo garantire la stampa in ordine di anno accademico, ciò è ottenuto con una ricerca del minimo anno accademico assegnato alle indisponibilità dei professori.
Congruentemente viene anche trovato il massimo, che fungerà da soglia superiore per il **while**, atto a "scorrere" tutti gli anni.

Un secondo **while** servirà a muoversi su tutti i professori, dove si userà l'anno accademico di riferimento per accedere alla **mappa** che memorizza le indisponibilità. Nel caso in cui il ritorno del metodo usato sia un **lista** di indisponibilità valida, si userà un iteratore ed un **while** per percorrerla e stampare l'anno accademico, se è il primo per quell'anno, l'id del professore, se è la prima indisponibilità per quel professore, e le indisponibilità stesse del professore, opportunamente separate da "**|**" per inizio e fine periodo di indisponibilità e da "**;**" tra i periodi di indisponibilità.

FindSomethingInList

In questo file vengono raggruppate tutte le funzioni che operando sulle liste di database o su loro copie o simili servono a trovare la posizione di un elemento delle stesse, o verificare l'esistenza dell'elemento cercato o trovare e restituire la lista collegata ad un certo elemento.
Il file però contiene anche una serie di funzioni di complemento usate in altre funzioni per operare un ordinamento o l'approssimazione dei tempi d'esame secondo gli slot.

findStudent

Questa funzione accetta come elementi passati una **lista** di studenti e in **id**, che scorrendo la lista dovrà essere trovato, restituendo l'**iteratore** della lista in cui si trova. Nel caso non venga trovato l'id dello studente cercato l'iteratore tornato punterà alla "**end**" della lista.

findProfessor

In maniera del tutto simile al caso precedente si avrà il ritorno dell'iteratore della lista in cui è stato trovato l'id del professore in oggetto.

findClassroom

Anche per le aule si scorre la **lista** di elementi fino a trovare l'elemento cercato, il quale viene ritornato come **iteratore**. La "**end**" della lista è ritornata nel caso l'elemento cercato non sia presente.

findCourse #1

Questa prima versione della find per i corsi fa uso solamente dell'id del corso da trovare, e come nei cari precedenti ritorna l'**iteratore** del corso se lo si è trovato, altrimenti la "**end**" della **lista**.

findCourse #2

Questa versione aggiunge l'elemento "**anno accademico**", quindi il ritorno sarà un **iteratore** che identifica il corso contemporaneamente per id ed

anno accademico. Il caso in cui non si trovasse il corso, la end della lista sarà il ritorno.

findCourse #3

Rispetto al caso "**findCourse #2**" si aggiunge la versione del corso da cercare, quindi la ricerca verrà operata prima per id poi per anno accademico e in ultimo per numero di versione. L'identificazione di un corso con tutte e tre queste caratteristiche comporterà il ritorno del suo **iteratore**, invece nel caso non si abbiano corrispondenze, verrà ritornata la "**end**" della lista.

findCourseLastForId #1

In questo caso si vuole trovare l'ultimo elemento di un corso, perciò si usa un **intero** che funge da posizione iniziale di ricerca, e che presumibilmente è ottenuto da una delle precedenti "**findCourse**". Si passa inoltre l'id del corso voluto , e con un **while** si scorre la lista dei corsi fino a trovare l'elemento successivo all'ultimo elemento avente l'id cercato.
Si fa notare che la funzione restituisce un'**iteratore** che punta all'elemento subito successivo all'elemento voluto.

findCourseLastForId #2

Questa find svolge la stessa funzione della precedente con la differenza che la posizione iniziale non è più un intero, ma un **iteratore**, anche qui presumibilmente restituito da una delle precedenti funzioni di "**findCourse**". Come nel caso sopra anche in questo l'iteratore restituito punterà all'elemento subito successivo all'ultimo id cercato.

findCourseLastForIdAndYear

La ricerca è condotta in modo uguale alle precedenti, ma si aggiunge l'elemento anno accademico come elemento di ricerca. Il ritorno risulta uguale ai precedenti casi.

findCourseTitle

La funzione serve a trovare eventuali **duplicati** dei titoli dei corsi.
Scorre il database dei corsi, cercando un determinato **titolo** di un corso, e restituisce un **booleano** che indica se è presente un duplicato (true) o meno (false).

findCourseOfStudy #1

La funzione riceve la lista di oggetti "**CourseOfStudy**" sulla quale effettuare la ricerca dell'id passato alla funzione. La lista viene dunque letta e i suoi elementi confrontati con l'id cercato.
La funzione ritornerà l'**iteratore** dell'elemento trovato, oppure in caso non vi siano riscontri, si ritorna l'elemento di end della lista.

findCourseOfStudy #2

La funzione restituisce una lista di corsi di studio che fanno uso di un certo corso, di cui viene fornito l'id. Assieme alla lista viene inserito come ultimi due elementi il semestre in cui si trova il corso, dove il penultimo elemento è un semestre valido, 0 per il primo e 1 per il secondo, mentre l'ultimo elemento della lista può assumere solo valore -1, indicando il semestre spento tra i semestri tra cui compare il corso.
Se i fattori non presentano un semestre valido, questo è rappresentato dal valore -2, sia per l'ultimo che per il penultimo elemento.

findCourseIdGrouped

Questa funzione, usata per lo scheduling, restituisce una lista di corsi che raggruppano nei loro corsi raggruppati un dato corso, passo per id alla funzione. Il corso viene cercato tra i corsi del database dei corsi.

findMaxAcademicYearUnavail

Serve a cercare nel database dei professori il massimo anno accademico per cui sono definite delle indisponibilità, restituendo tale anno come oggetto di tipo "**Date**".

comp

È una funzione di complemento usata per eseguire la sort della lista dei professori in base all'anno accademico minimo per le indisponibilità esposto dall'oggetto professore. Esso in particolare compara gli elementi della lista rispetto ad un minimo che la sort si costruisce internamente, e restituisce se l'elemento selezionato è minore del minimo.

sortMethodForProf

È una funzione di complemento usata per eseguire la sort della lista dei professori in base all'id esposto dall'oggetto professore. Esso in particolare compara gli elementi della lista rispetto ad un minimo che la sort si costruisce internamente, e restituisce se l'elemento selezionato è minore del minimo.

sortMethodForClassroom

Questa funzione è usta per riordinare le aule rispetto alla loro capacità in occasione degli esami. Si tratta di una comparazione "dell'**examCapacity**" dell'aula selezionata rispetto al minimo finora trovato dalla sort. In base al risultato di tale confronto verrà ritornato true se l'elemento ha valore inferiore al minimo, false nel caso opposto.

sortMethodForCourse

Ha lo stesso uso delle precedenti, ma per l'ordinamento della lista dei corsi che dovranno essere schedulati, in questo caso particolare. La lista dei corsi verrà riordinata dalla sort usando questa funzione, che avrà come ritorno true se il numero di studenti di un corso sono in numero minore rispetto al minimo, al contrario verrà restituito un false.

sortMethodForPrintSchedule

La stampa avviene, dato un giorno, raccogliendo tutti i corsi per un dato slot orario. Per ottenere il rispetto delle specifiche sull'output viene eseguito l'ordinamento del vettore creato secondo prima l'id del corso, poi per numero di versione ed in ultimo per id del corso di studi.
In particolare la funzione è usata in un sort dove compara un elemento del vettore rispetto ad un elemento minimo, restituendo un true se l'elemento è inferiore al minimo, o un false in caso contrario.

sortMethodForPrintWarnings

Ad ogni corso sono associati dei **booleani**, che rappresentano i vincoli violati. Per eseguirne la stampa è richiesto che i vincoli violati abbiano uno specifico ordinamento, cosa che si crea di fare con l'uso di questa funzione nella sort.
I vincoli devono essere ordinati prima per id del corso di studi, poi per corso ed in ultimo per numero del vincolo violato.
Il ritorno della funzione è un **booleano**, che è true se l'elemento da comparare è "minore" del minimo e false nel caso contrario.

approXimationFunct

È una implementazione equivalente della funzione "**ceil**" della libreria **math.h**.

FillDatabaseList

Contiene le funzioni per poter eseguire l'aggiornamento o l'inserzione di un corso nel database, oltre a alle funzioni per controllare e aggiornare i professori ad essi assegnati. Come detto in precedenza il corso è presente ripetutamente per ogni versione per ogni anno accademico per cui è definito, sarà quindi necessario un

controllo sul numero e contenuto delle versioni e azioni congrue nel caso queste differiscano nel numero o in assenza di dati a loro assestanti.

fillCourseDatabase

Questa funzione è richiamata dalla "**CourseToInsertFile**" e serve ad aggiornare un corso già esistente in database, ossia corrispondente sia per id che per anno accademico. In questo caso, viene passato alla funzione la lista temporanea delle versioni del corso, costruita nei passaggi finali della **insert**. La nuova lista di corsi, che potrebbe avere elementi mancanti, viene letta, e per ogni elemento viene cercato il corrispettivo nel database, ovvero si cerca una corrispondenza per id, anno accademico e versione. Se questa viene trovata, si procede a verificare gli elementi presenti nel corso "dummy" e se presenti andranno a sovrascrivere i corrispondenti elementi del corso nel database, in caso contrario il database rimarrà invariato. Un controllo a parte è eseguito sui professori assegnati al corso, che vengono gestiti dalla funzione "**fillAssociateProfesor**", e successivamente controllati rispetto alle ore totali assegnate al corso stesso.
Si deve tenere comunque in conto i casi per cui il numero di versioni presente nel database è inferiore al numero che si vuole inserire, oppure il numero di versioni nel database è superiore. Nel primo caso si continuerà a scorrere la lista, ma in verrà effettuato un controllo sulla completezza dei campi del corso da inserire, e se il controlla ha buon fine varrà inserito dopo l'ultima versione del corso aggiornato.
Nel secondo caso le versioni in eccesso del database verranno cancellate.
Possibili errori nei dati valutati o la loro assenza comportano che il ritorno della funzione sia "false", in caso contrario il ritorno sarà true.

insertCourseDatabase

Questa funzione, come la precedente è richiamata dalla "**CourseToInsertFile**", e si propone di inserire nella base dati un corso per il quale è presente solo la corrispondenza per id.
In modo simile al precedente si scorre la **lista** di corsi dummy costruita dalla **insert**, senza però effettuare nessun controllo sui campi, poiché questi vengono ereditati nella insert dall'ultimo corso corrispondente per id e versione. Unica eccezione è fatta per i professori assegnati al corso, per cui è controllata la coerenza della somma delle ore con il numero previsto per la totalità del corso. Per questi è inoltre prevista la possibilità di ereditare elementi dall'ultimo corso di pari versione presente in database, per questo motivo si richiama la funzione "**insertAssociateProfessor**".

Se la lista dei corsi è completa e coerente verrà inserita nel database in ultima posizione rispetto al corso di riferimento.
In modo uguale alla funzione precedente ci si aspetta che la funzione in assenza di errori ritorni il **booleano** "true", in caso contrario il ritorno sarà "false"

fillAssociateProfessor

La funzione può esssere chiamata solo dalla "**fillCoureDatabase**" e si comporta in modo simile alla stessa, ma facendo riferimento questa volta alla lista di professori assegnati al corso.
Come accennato si scorrerà la **lista** dei professori del corso dummy verificando se questo va a ridefinire i campi del corrispondente, per posizione, professore del corso che si sta valutando nella "**fillCoureDatabase**". Si deve però distinguere i casi in cui la lista di professori del corso dummy abbia un numero di elementi superiore alla lista dei professori derivata dal corso in database, e il caso opposto di un numero di professori della lista temporale sia inferiore al database. Nel primo caso è necessario, non avendo più professori a cui sovrascrivere i dati, che i campi del professore in eccesso siano completi, nel qual caso si avrà l'inserimento dello stesso nella lista dei professori assegnati.
Se ricadiamo invece nel caso opposti si prevede la cancellazione dei professori in eccesso .
In qualunque momento è previsto in caso di incongruenza o errori nei campi la segnalazione dell'errore e contestualmente il ritorno della funzione sarà "false", nel caso sia andato tutto a buon fine il ritorno sarà "true".

insertAssociateProfessor

La funzione si occupa di verificare la completezza dei professori presenti nella **lista** dummy, rispetto alla **lista** dell'ultimo corso di pari id e versione in database, passate entrambe come argomento alla funzione. Come accennato si scorrerà la lista dei professori verificando se il professore presenta il campo che si sta verificando, e nel caso non sia presente, si procederà ad ereditarlo dal professore di pari posizione assegnato all'ultimo corso di pari id e versione in database. Nel caso la lista abbia un numero di elementi superiore a quella da cui può ereditare, i professori eccedenti vengono controllati a riguardo della completezza dei dati da loro contenuti, e se non presentano dati mancanti vengono inseriti nella lista di professori.
Anche i questo caso se si riscontrano errori nei dati o l'assenza degli stessi, un errore viene segnalato, e il booleano di ritorno della funzione sarà congruo con tale evenienza.

PatternConstrainVerification

In questo file si raccolgono le funzioni usate per la verifica della coerenza dei dati letti dai file di input, e che si vogliono usare per creare i file di database. Sono anche inserite due funzioni per la gestione dei corsi internamente ai corsi di studio.

parallelVersionProgression

Viene utilizzata nella lettura del file "**db_corsi.txt**" e serve a verificare la coerenza dell'id impostato per la versione con il numero di versione ottenuta dal contatore interno al **while** che deserializza l'organizzazione delle versioni e professori.
In pratica si vuole verificare che le versioni lette siano crescenti e consecutive, a partire della versione "base".
Se i due termini coincidono il ritorno della funzione sarà "true", in caso contrario sarà "false".

generateVersion

Genera e ritorna la versione del corso successiva all'ultimo inserito.

versionCoherencyTest

Come la "**parallelVersionProgression**" serve a verificare la consecutività e l'ordine ascendente delle versioni dei corsi, ma in questo caso gestiti nella lettura del file di update o inserimento di un corso già esistente.
Il ritorno è "true" se i termini della verifica coincidono, altrimenti si avrà "false".

examSessionAcademicYearCoherencyTest

La funzione si propone di verificare le date delle sessioni ottenute dal comando per impostare le sessioni esami, con riferimento l'anno accademico fornito nella stessa stringa.
Se le date passate nel vettore hanno per anno lo stesso fornito per riferimento, il booleano di ritorno sarà "true", in caso opposto sarà "false".

examSessionBeginEndVerification

Sulle stesse date usate nella funzione precedente si vuole verificare che, delle coppie, la data d'inizio sia prima di quella di fine.
Naturalmente se tale condizione è rispettata da tutte le coppie la funzione avrà ritorno positivo.

examSessionOrderVerification

In questa funzione si vuole controllare che non vi siano sovrapposizioni tra le date che definiscono le sessioni esami, aggiunte mediante l'apposito comando. Contestualmente si vuole controllare l'ordinamento delle sessioni, per cui queste non risultino tra loro scambiate. In caso le condizioni vengano rispettate dalle sessione il ritorno della funzione sarà "true", al contrario se anche solo una viola questi vincoli, il ritorno sarà "false".

sessionDurationConstrainVerification

La funzione si propone di verificare la durata delle rispettive sessioni d'esame impostate dall'opportuno comando, secondo le specifiche date. Se tutte e tre le sessioni hanno una durata corretta la funzione ritornerà "true", altrimenti il ritorno sarà "false".

unavailabilityDatesVerification

La funzione serve a controllare le date di indisponibilità, per il singolo professore, lette dal file sulle indisponibilità per un dato anno accademico. Si verificherà in particolare se una data nella **lista** è già presente nella lista stessa, e l'assenza di sovrapposizioni tra le date di indisponibilità stesse. A proposito di quest'ultima condizione è opportuno far notare che è considerata un errore, e verrà opportunamente segnalata. Se le condizioni risultano tutte verificate, allora la funzione ritornerà "true", in caso contrario il ritorno sarà "false".

putCourseInEndedCourses

Questa funzione, usata solamente per l'aggiornamento dei corsi , o l'inserimento di un corso per un anno accademico non ancora esistente, opera sui corsi di studio, ma in base allo stato in cui si trovano i corsi. In particolare se un corso passa da attivo a spento si verificherà la condizione di tutti gli altri corsi, di pari id, e se almeno uno è già spento l'id non verrà copiato nei corsi spenti, per i corsi di studio che ne fanno uso, in quanto è già presente. Nel caso invece tutti i corsi siano spenti, si procederà a rimuovere da tutti i corsi di studio il corso in questione dai corsi attivi, usando il metodo creato appositamente nella classe "**CourseOfStudy**".
Se l'esecuzione di questa operazione ha buon fine, la funzione ritornerà "true", il ritorno sarà "false" in caso opposto.

removeCourseFromEndedCourses

Questa funzione opera in modo contrario alla precedente, eliminando il corso dai corsi spenti, se tutti gli anni accademici di quel corso sono attivi.

In alternativa lasciando invariata la struttura dei corsi di studio se almeno uno degli anni accademici, per i l dato corso, è ancora spento.
Si vuole portare l'attenzione sul fatto che se si passa da una condizione in cui tutti gli anni accademici sono spenti ad una situazione in cui almeno uno torna ad essere attivo, il semestre sarà definito da una update dei corsi di studio, e non dalla presente funzione.
Se l'operazione eseguita in questa funzione dà esito positivo il ritorno della funzione sarà “true”, in caso contrario sarà “false”.

FieldVerificationForScheduling

In questo file si trovano le funzioni usate nello scheduling, per controllare la completezza dei campi introdotti dagli elementi di database. Normalmente ci si aspetta che le informazioni contenute nel database siano complete e valida, ma ciò che “preoccupa” in particolare è il disaccoppiamento, ma allo stesso tempo la dipendenza, tra i corsi e i professori, per i quali è mantenuto un puntatore all'interno dei corsi stessi.
Nello stesso file sono inseriti anche delle funzioni che operando sul database semplificano lo scheduling dei corsi.

courseFieldVerification

La funzione iterando su una lista di corsi, ne verifica la presenza e validità del suo contenuto. Per ogni corso avviene anche la verifica dei professori ad esso assegnati, con una verifica del puntatore al database dei professori da parte della funzione seguente, e il controllo della completezza dei dati contenuti “dall’associateProfessor”.
Ogni corso tiene anche una lista di corsi raggruppati, per i quali viene effettuata una verifica della loro esistenza nel database dei corsi di studio.
Se tutti i controlli sui corsi hanno avuto esito positivo, il ritorno della funzione è “true”, al contrario se i controlli non sono superati, sarà “false”.

ProfessorFieldVerification

Questa funzione è richiamata dalla precedente per operare il controllo sui professori che il corso memorizza mediante la lista dei professori assegnati.
Anche in questo caso se il professore ha i campo completi il ritorno della funzione sarà “true”, in caso opposti il ritorno sarà “false”.

classroomFieldVerification

In questo caso si vuole controllare la completezza dei dati contenuti dagli oggetti aula memorizzati nella lista di database.
Il ritorno della funzione, in modo simile alle percenti funzioni, sarà congruo rispetto al superamento delle verifiche.

regroupingCoursesForCommonCourse

La funzione restituisce una lista di stringhe di id di corsi, i quali hanno la caratteristica di essere raggruppati l'uno con l'altro. in particolare partendo da un corso, il suo id viene inserito in una lista, nella quale vengono inseriti anche i suoi corsi raggruppati e si attua una ricerca sui corsi che raggruppano il corso in esame. Da questa lista vengono eliminati eventuali duplicati e si prende l'id dell'esame successivo nella lista per effettuare lo stesso procedimento.
Questo ciclo, determinato da un while con un iteratore sulla lista viene ripetuto fintanto che non si hanno più corsi da dover raggruppare.

groupedCoursesVerification

In questa funzione di usa la lista formata dalla funzione precedente per cercare i corsi di studio che fanno uso dei corsi nella lista stessa, per fare ciò si usa la funzione precedentemente descritta "**findCourseOfStudy**".
Contemporaneamente si verifica che il semestre in cui sono inseriti i corsi sia uguale per tutti i corsi nella lista, questo è possibile perché ogni lista di corsi di studio per i rispettivi corsi ha come elementi terminali il semestre in cui si trova lo stesso corso.
Se i controlli hanno avuto buon esito la funzione si occupa "dell'espansione" dei corsi, ovvero viene popolata una struttura ad'hoc, che facilita lo scheduling e il reperimento delle informazioni necessarie alla stampa, la quale è ripetuta per un numero di volte pari al numero di versioni che ha i corso in questione.
Il ritorno è l'eventuale codice di errore generato dai controlli fatti nella funzione.

myUnique

Alla funzione viene passata la lista "intermedia" generata dalla funzione "**regroupingCoursesForCommonCourse**". Questa viene letta e i suoi elementi confrontati con tutti gli altri della lista, alla ricerca di eventuali duplicati.
il ritrovamento di un duplicato comporta la cancellazione dell'elemento stesso.

Gestione degli errori

La gestione degli errori è assegnata puntualmente ad ogni funzione o metodo che serve a leggere, caricare o controllare dati. In particolare ogni errore si caratterizza da un codice di errore univoco, rispetto al tipo di errore, a cui si

associa una stringa che descrive l'errore occorso e indica il campo. Oi capi soggetti all'errore in questione.
In generale nella lettura del file viene sempre in indicato il file in questione e la riga in cui si è riscontrato l'errore, a questo solitamente si aggiungo dei campi accessori, utili a identificare meglio il campo soggetto all'errore. Un esempio può essere la coppia di date che definiscono una sessione, la quale non rispetta certi vincoli, o l'id del professore titolare che non concordo con il primo campo dell'organizzazione dei professori, o non concorda con una sua ridefinizione. Per questo motivo questi dati vengono indicate direttamente all'utente come possibile problema.
Si fa notare che l'assenza di un dato nei file immessi comporta la segnalazione dell'errore per il parametro mancante, se questo deve essere presente.
Un possibile problema della gestione degli errori, può essere dovuto alla mancanza dei caratteri di separazione delle stringhe, per cui un errore potrebbe non essere immediatamente segnalato, ma presentarsi nel campo successivo. Questo potrebbe portare ad una incongruenza dell'errore segnalato, ma in assenza degli opportuni delimitatori, e non potendo garantire l'incorrettezza del campo precedente, si è optato per mantenere questa struttura.
Il codice di errore associato all'errore stesso viene, oltre ad essere ritornato dal metodo o dalla funzione, utilizzato internamente al metodo o funzione stessa per impedire qualsiasi ulteriore esecuzione della stessa. Inoltre la stringa viene solo generata all'interno dell'oggetto o della funzione, ma non avviene un output della stessa sul terminale, infatti fa parte del ritorno del metodo o funzione, fino a risalire al main dove la stampa dell'errore avviene in chiusura del programma.

Aggiornamento corsi di studio

Come visto dai metodi e funzioni precedenti, le specifiche prevedevano la possibilità di alterare l'organizzazione dei corsi di studio mediante una modifica dei corsi. Infatti a seconda dei casi, lo spegnimento di un corso può comportare la copia del corso stesso nei corsi spenti, oppure la rimozione dei corsi e inserimento nei soli corsi spenti. In particolare questa seconda opzione risulta problematica poiché si perde traccia di ogni riferimento al semestre in cui dovrebbe essere situato nel momento in cui si voglia riattivare.
Per ovviare a questa problematica si è deciso di introdurre una nuova funzione, non prevista dalle specifiche, e che permette l'aggiornamento dell'organizzazione dei corsi di studio.
Per l'utilizzo di questa funzionalità si deve specificare oltre al file con righe di formato riportato sotto e nell' esempio, l'identificatore del comando, che in questo caso è "**-u:f**".
Per una descrizione della funzione in questione si rimanda al capitolo dedicato ai file di aggiornamento "**UpdateFileHandling**", invece qui ci concentriamo sul come questa funzione è stata pensata.

La stringa di update per i corsi di studio deve avere come primo elemento l'id del corso da aggiornare, se questo non sarà presente, o non lo si trova nel database dei corsi di studio verrà segnalato mediante una stringa di errore. Come secondo elemento si ha l'organizzazione dei corsi nei diversi semestri, e come nei precedenti casi di update, sarà possibile omettere dei campi, che verranno ereditati dal corrispondente corso in database. Per questo secondo elemento, se si vogliono effettuare modifiche, sarà necessario inserire gli identificatori di campo. Ovvero saranno necessarie le "**[]**" per indicare che ci sarà una ridefinizione del corso di studi, e le "**{}**" per indicare la ridefinizione del semestre, in assenza di queste ultime il semestre in questione non subirà modifiche. Internamente alle "**{}**" si possono omettere dei campi, ma sarà sempre necessito inserire il separatore "," per indicare il numero di campi da ereditare. Inoltre l'ereditarietà dei campi è posizionale, quindi dato un semestre si avrà corrispondenza uno a uno sulle posizioni del semestre da ereditare rispetto a quello nuovo. Questo organizzazione però comporta che venga prestata molta attenzione al numero di campi che si vogliono ereditare, poiché se si eccede il numero di campi ereditabili per il semestre, verrà segnalato un errore.
Si informa anche che vi sono alcuni elementi non modificabili, come il tipo di corso di studi "**BS/MS**", e l'elenco dei corsi spenti, che pertanto non verranno mai alterati, poiché definisce una caratteristica peculiare del corso di studi, il primo, e il secondo deve rimanere modificabile unicamente variando lo stato dei corsi impiegati.

C0001;[{01AAAAA, 01AAAAB},{01AAAAC,},{,01AAAAC,}**,,**{},{01AAAAD}]

c. esempio di stringa usata per aggiornare i corsi, il secondo semestre eredita il secondo elemento, il terzo semestre eredita il primo e ultimo elemento, il quarto semestre rimane invariato , il quinto semestre eredita l'unico elemento

Pianificazione degli esami

A questo punto ci si aspetta di avere tutti i dati necessari alla pianificazione delle sessioni, ciò comporta che esistano i diversi file di database e che questi siano completi e non contengano errori nella loro formattazione del testo o nei loro campi. Rispetto a questa ultima informazione, nello scheduling, non viene effettuato nessun controllo sulla validità dei dati, ma verrà solo controllata la loro esistenza.
Partendo dal main si caricano tutti i file di database, creando le liste di oggetti relative a ciascun file. Successivamente si esegue un **while**, dove viene creato, mediante il costruttore della classe "**SessionScheduler**", l'oggetto examPlanning. Quindi si usa uno **switch-case**, che fa uso della variabile usata come condizione del while, per identificare la sessione che si vuole schedulare. Per ciascuna di esse viene chiamato un metodo apposito, il "**goupedCoursesScheduling**" che si occupa di schedulare i corsi per la sessione che si sta programmando. Si fa

osservare che tale metodo è chiamato più volte, in particolare tre volte, uno per ogni semestre che si vuole schedulare per la sessione in oggetto. Rispetto a quanto detto i semestri sono identificati come “0” per il primo semestre, “1” per il secondo semestre e “-1” per i corsi spenti, inoltre si darà priorità nella programmazione al semestre di numero uguale alla sessione che si sta schedulando.
In ultimo avviene la stampa del file di calendario con la relativa programmazione dei corsi, e il file relativo alla violazione dei vincoli. La stampa avviene contestualmente al termine della programmazione della sessione, e prima di ripetere la programmazione per la sessione successiva, o uscire dal **while** se si è programmata l’ultima sessione d’esami.

Il costruttore

Per quanto riguarda il costruttore della classe, riceve come elementi tutti i database, tranne la mappa con le date delle sessioni d’esame, per cui viene solo passata la data di inizio e fine della sessione che si sta schedulando, per l’anno accademico desiderato, quest’ultimo anche passato al costruttore. Il costruttore procede quindi prelevando dal database dei corsi, solamente quelli che fanno riferimento all'anno accademico passato come riferimento, e si costruisce una lista interna, denominata “**_coursesToSchedule**“, che racchiude unicamente i corsi da schedulare.
Durante la formazione della lista interna dei corsi, viene controllata la completezza dei campi dei corsi che si sta andando ad utilizzare, compresa la presenza e completezza dei professori assegnati ai rispettivi corsi.
Si vuole evidenziare, e ricordare che la lista creata rimane impostata come il database, ovvero per ciascun corso è presente una sua “ripetizione” pari al numero di versioni che deve avere.
Dalla lista appena creata si vanno a controllare possibili raggruppamenti tra i corsi, i quali vengono raggruppati in liste apposite e inseriti nella struttura “**groupedCoursesToPlan**” in base al semestre di riferimento.
Contestualmente a questa operazione viene controllata la coerenza dei corsi raggruppati rispetto al semestre al quale si riferiscono.
Si aggiunge che il metodo che esegue l’organizzazione dei corsi nei raggruppati, esegue anche un’ordinamento, in senso crescente, degli stessi in base al numero di iscritti al corso stesso.
In ultimo il costruttore imposta la struttura del calendario , prima andando ad ordinare le aule, con un apposito metodo di sort, e poi inserendole come chiave della mappa “**_datesPlanning**”, senza però ancora andare a creare, per ciascuna di queste la matrice che rappresenterà il calendario. Questa ultima azione è svolta da un metodo a se stante, che

riceve la mappa prime citata, oltre alle date di inizio e fine della sessione, e crea internamente la matrice della struttura "**examScheduled**", di dimensione paria alla differenza tra le date per il numero di slot giornalieri.

La pianificazione

Quando dal main viene richiamato il metodo "**groupedCoursesScheduling**", viene eseguita la programmazione della sessione. Il metodo riceve come parametri la sessione che si sta schedulando e il semestre che si vuole schedulare, questo serve a distinguere la tipologia di programmazione, infatti nel caso queste siano concordi, stesso numero, sarà necessario programmare il primo e secondo appello. Nel caso invece per cui siano diversi si dovrà programmare un solo appello, a partire dalla terza settimana. In aggiunta viene anche passata la data di inizio della sessione, il. Database dei professori, in originale, e l'anno accademico di riferimento.
La programmazione è stata pensata per lavorare su delle copie, fatte internamente a questa funzione, dei contenitori denominati "**_datesPlanning**", "**_groupedCoursesToPlan**" e del database dei professori. Questo perché in caso si modificasse gli originali, sarebbe complicato tenere traccia di tutte le modifiche fatte, in particolare per quanto riguarda il posizionamento dei raggruppamenti, la violazione dei vincoli e l'aggiunta delle indisponibilità dei professori. In questo modo invece possiamo "buttare" la copia e ripartire da una situazione valida, data dalle strutture originali.
Proseguendo, un **while** più esterno indicherà lo scheduling che si sta effettuando, mentre un for puntando al semestre della mappa dei corsi raggruppati, recupererà i vettori di corsi raggruppati, che verranno letti da un iteratore.
Un secondo **while** si occupa di verificare il posizionamento del corso puntato dall'iteratore, previo controllo del rispetto dei quattro vincoli imposti dalle specifiche.
Nel caso il posizionamento del corso non abbia avuto successo, si passerà allo slot successivo, fintanto che non si raggiungerà l'ultimo slot, quindi si cambierò giorno si ripartirà dallo slot iniziale (0). Se si raggiunge la fine della matrice del calendario si procederà a rilassare progressivamente i vincoli, fino al rilassamento del quarto, per il quale è previsto un comportamento specifico.
Contestualmente al cambiamento di slot o giorno, viene eseguito il reset dei contenitori di copia, con il metodo appositamente creato "**resetDataFromDatabase**". Se il posizionamento ha esito positivo viene usato il metodo "**coursePositioning**" che si occupa di popolare

correttamente la struttura del giorno, ora e aula identificati come disponibili dalla verifica dei vincoli.
Il posizionamento del corso implica un effettivo aggiornamento dei dati originali, prima di passare al prossimo corso, facente parte dello stesso raggruppamento.
Il **ciclo** precedente è ripetuto fintanto che non si riesce a posizionare tutti i corsi raggruppati secondo le specifiche, ovvero per lo stesso giorno e stesso slot orario di inizio, progressivamente rilassando i vincoli se necessario. Terminati tutti i posizionamenti del raggruppamento si uscirà dal secondo while e il for passerà a puntare il raggruppamento successivo, per cui si ripete il procedimento descritto.
Come già accennato, un comportamento diverso è implementato qualora venga rilassato il constrain 4, infatti si prevede che in questo caso venga scorso tutto il calendario per trovare il giorno e slot in cui si è riusciti a posizionare il numero massimo i corsi del raggruppamento.
Conseguentemente ci si posizionerà su tale giorno e slot e si effettuerà il posizionamento come per un qualsiasi corso. Per tuti i corsi che non sarà possibile posizionare verrà segnalata la violazione del solo vincolo 4, e conseguentemente non appariranno nella programmazione.
La verifica dei vincoli è eseguita usando dei metodi interni, in particolare di seguito si darà una descrizione di come sono implementati i diversi controlli.

constrain_1 (vincolo 1)

Il primo vincolo prevede che non vi sia lo stesso corso di studio ad una distanza inferiore a due giorni dal giorno selezionato per inserire un corso.
Il controllo prevede quindi di iniziare il controllo da due giorni prima rispetto al giorno selezionato, facendo attenzione se ci si trova nei primi giorni del calendario. In modo simile, il limite superiore è due giorni dopo il giorno selezionato, anche questa volta si dovrà prestare attenzione, ma rispetto alla fine del calendario. Quindi si userà un primo while per muoversi sui giorni, un secondo sugli slot del calendario e un terzo per muoversi sulle aule.
Un quarto while è usato per spostarsi, con un iteratore, sulla lista dei corsi di studio contenuta dalla struttura "**courseOrgBySemester**", derivata dal corso che si sta cercando di posizionare.
Per ogni elemento della lista è effettuata una find sulla lista dei corsi di studio contenuta nella struttura "**examScheduled**" dello slot del calendario. Da questa se viene ritornato un elemento valido verrà settato il **booleano** corrispondente alla violazione del primo vincolo per il corso in uso e viene segnalata la violazione anche per il corso che ha portato alla violazione e per tutte le sue versioni.

Il controllo in particolare procederà prima a parità di giorno e slot su tutte le aule, poi verrà variato lo slot, e verrà ripetuta la ricerca su tutte le aule. Solo quando si sarà raggiunto l'ultimo slot per il dato giorno si passerà al giorno successivo, rifacendo la ricerca su tutte le aule per tutti gli slot presenti.
Congruentemente al risultato della ricerca, verrà ritornato dal metodo un "**true**", che segnalerà l'avvenuta violazione.
Nel caso opposto, per cui non si hanno riscontri il metodo ritornerà "**false**".

constrain_2 (vincolo 2)

Con il secondo vincolo si vuole verificare che il secondo appello venga posizionato ad una distanza di almeno 14 giorni dal primo appello, e nel caso questo vincolo sia violato, venga garantita una distanza di almeno sei ore del secondo appello rispetto al primo appello.
Tale vincolo iene controllato con una struttura simile a quella vista per il vincolo 1, per cui si ha un primo **while** per scorrere sui giorni, un secondo gli slot ed un terso le aule. Il giorno di partenza è calcolato sottraendo le due settimane al giorno di partenza, mentre il limite superiore è la somma dei due termini. Nella definizione dei limiti dell'intervallo di controllo si deve fare attenzione che gli stessi limiti non escano dal range valido per il calendario.
Il controllo viene quindi effettuato comparando id del corso e della versione che si vuole inserire con quelle che stanno occupando il giorno, lo slot e aula puntata. Se non vi è corrispondenza si passa a valutare le aule successive a parità di slot. Avendo terminato tutte le aule da controllare si passa allo slot successivo e si esegue il medesimo controllo.
Quando si raggiunge l'ultimo slot per il giorno selezionato si passa al giorno successivo riesegguendo la procedura prima indicata fino al termine del range impostato.
Se durante la verifica è stata trovata la programmazione del primo appello del corso all'interno dell'intervallo, verrà segnalata la violazione del vincolo come ritorno e verrà segna contemporaneamente la violazione nel **booleano** corrispondente al vincolo, contenuto nella struttura "**courseOrgBySemester**".
Se non vi sono stati problemi nel posizionamento il ritorno del metodo sarà "false".

constrain_3 (vincolo 3)

Il terzo vincolo richiede di evitare la programmazione di un esame nei giorni o nelle ore di indisponibilità di un professore.
Tale funzione è eseguita prendendo dal corso lealista di professori, e con un iteratore verificare le indisponibilità di ciascuno di questi.

Tale controllo presenta però una problematica, infatti essendo che lavoriamo su delle copie dei contenitori originali, si ha che la lista presenta ancora il puntatore valido alla lista di professori originale, che non vogliamo sia modificata. Per questo motivo ad ogni ciclo del while, inserito per scorrere la lista di professori assegnati al corso si effettua una ricerca per id del professore puntato dall'iteratore nella copia della lista di professori. Ottenuto l'oggetto professore dal duplicato della lista dei professori, viene eseguito il metodo "**isAvailExamProgramming**", il quale verifica tutte le indisponibilità del professore, siano esse derivate dal file di indisponibilità, che dal posizionamento del professore per altri esami.
Se il controllo non ha individuato problematiche il programma assegna l'indisponibilità del professore per data e slot orario corrente. Altrimenti la violazione del vincolo viene segnalata e viene interrotto il while, comportando il ritorno del metodo a "**false**".
Si fa notare che fino ad adesso le date e slot sono state gestite come indici della matrice, cosa che non è possibile usare per il controllo corrente, si necessita quindi di convertire gli indici in due variabili di tipo data, che indicheranno l'inizio e la fine dell'indisponibilità "reale" del professore.

constrain_4 (vincolo 4)

In questo metodo si effettua il controllo sulla capienza, il tipo di aula, e la disponibilità dell'aula destinati all'esame per il corso puntato.
Per eseguire questa verifica si tiene costante il giorno e slot da verificare, mentre con un while si corrono tutte le aule, di queste viene prima verificato se il tipo di aula è congruente con il tipo di aula definita per l'esame dal corso, e se il numero di studenti iscritti al corso possono essere contenuti nell'aula puntata. Avendo trovato l'aula quindi si verifica la disponibilità della stessa per una durata pari al numero di slot necessari all'esame. Se l'aula risulta vuota, questa viene ritornata allo scheduling, in caso contrario si passerà e verificare la disponibilità sulle successive.
Nel caso non si trovi un'aula disponibile, o con un numero di slot occupabili inferiore al necessario il ritorno del metodo sarà "**false**", in caso contrario verrà ritornato il numero dell'aula con ritorno del metodo "**true**".

La stampa

Per l'output del calendario, come quello dei warning, si fa riferimento a due metodi pubblici, ciascuno dei quali assolve ad una delle due funzioni.
Il primo metodo, denominato "**outputSessionFile**", che si occupa dell'output del calendario, prevede il passaggio al metodo del nome del file, oltre al numero della sessione e la data di inizio sessione, dopo di che

userà solo dati interni all'oggetto stesso. In particolare si utilizzerà il calendario costituito dal contenitore "**_datesPlanning**" , da cui si leggeranno gli elementi tramite tre while. Il primo while servirà a spostarsi sui giorni , il secondo sugli slot orari e il terzo sulle classi, in particolare fissato un giorno e un slot si recupera il contenuto di ogni aula, e si popolerà per ciascuno di essi contenente dei dati, una struttura ad'hoc chiamata "**expandedScheduleForPrint**". Questa struttura verrà a sua volta inserita in una lista, che successivamente verrà riordinata dall'algoritmo di sort "**sortMethodForPrintSchedule**", descritto in precedenza.
A questo punto si distinguono due rami di stampa, il primo prevede che la lista di elementi precedente inseriti, "**previousSchedule**", sia vuota, per cui si legge la lista riordinata, e ne si stampano gli elementi. Nel secondo caso la lista presenta lo scheduling del precedente slot orario, per cui si effettua un controllo sulla presenza di corsi ripetuti tra i due slot, nel cui caso non si avrà la stampa del corso, ma solo il separatore "**;**".
Si ripeterà questa routine per tutti gli slot di tutti i giorni, stampando ad ogni variazione de giorno o dello slot il valore aggiornato della data o dell'orario, secondo il formato previsto dalle specifiche, e tenendo in conto che per la data non verrà stampata la domenica, risultando in un doppio incremento ogni sei giorni.
Per la stampa dei warning ci si affida invece al metodo "**outputWarningFile**", il quale prevede di scorrere ogni elemento del calendario, per ogni aula, e riutilizzando la struttura "**expandedScheduleForPrint**", andare a creare una lista con i soli elementi che presentano delle violazioni dei vincoli. Successivamente si riordinerà tale lista con un algoritmo di **sorting** che fa uso della funzione "**sortMethodForPrintWarnings**", vista in precedenza.
In ultimo si effettuerà la stampa della lista, ordinata secondo le specifiche, per la quale è stato introdotto l'ulteriore accorgimento per evitare la stampa di warning ripetuti per ogni versione.

Considerazioni

in quest'ultimo capitolo si vogliono dare le ultime indicazioni su problematiche incontrate nello stesura del progetto e la metodologia con cui sono state risolte o segnalate.
iniziando dai file di input, siano essi di aggiunta, di aggiornamento o inserimento ci si aspetta che l'ultimo campo della riga non termini con il separatore ";". A tale scopo si esegue un controllo sul numero dei campi al momento della lettura, e un controllo diretto sull'ultimo carattere della riga. Se il numero di campi è congruo con il tipo di comando utilizzato il controllo dell'ultimo carattere di riga può comportare la segnalazione di un numero di elementi discordante con

quanto il sistema si aspetta. In alternativa se la presenza del ";" è dovuta alla mancanza dell'ultimo campo, il programma provvederà a segnalarlo.
Per quanto riguarda gli stessi file di test sono stati raggruppati in sette test, con i primi due creati per testare la capacità di creare i file di database e verificare il riconoscimento degli errori, e i restanti cinque usati per la verifica e generazione dei file di programmazione delle sessioni d'esame. Si aggiunge che ogni test ha una propria cartella nella quale si trova un "README.txt" nel quale si spiega a cosa servono i file di test in esso contenuti, e come ci si aspetta che siano i file di database o di schedule generati.
A queste cartelle di test si aggiungono un corrispondente numero di cartelle, contenenti il risultato del corrispondente test, inoltre contengono anch'esse un file "README.txt" nel quale in cui si dà un'overview di cosa si cerca di ottenere o verificare con i file contenuti nella cartella.
Per semplicità i file di test sono stati generati mediante un altro programma scritto in Python, reperibile al link https://github.com/LBrignone/AutomaticTesting_Progetto_AeC.git, questo sistema è stato utilizzato per cercare di ridurre la possibilità di errore in fase di test, e ridurre così il tempo necessario a svolgere gli stessi test. Detto ciò questo programma ausiliario non è completo di controlli, quindi in caso di utilizzo potrebbero essere necessarie alcune modifiche.
Passando invece ai corsi di studio si fa osservare che normalmente l'aggiunta di un nuovo corso di studi non prevede la presenza della sezione dei corsi spenti, ciò però comporterebbe delle incongruenze nel database. Infatti così facendo si potrebbe avere un corso di studi che presenta un corso solo negli attivi, nonostante questo sia completamente o parzialmente sprento. Per ovviare a questa incongruenza, contestualmente all'aggiunta del corso di studi si ha una ricerca nel database dei corsi inseriti, come attivi, e se compaiono anche come spenti si crea una lista temporanea di corsi spenti da assegnare al corso di studi. Si fa notare che questa misura non è però garanzia della validità del corso di studi nella sua totalità, ma è atta a prevenire errori o dimenticanze da parte dell'utente. infatti il modo più corretto di agire prevede l'utilizzo del comando di aggiornamento dei corsi di studi, con il quale si può ridefinire il corso di studi nella sua totalità.
Per quanto riguarda la programmazione degli esami si vuole aggiungere un'ultima nota, per quanto riguarda un possibile problema riscontrato nell'implementazione del progetto. Infatti nel database un corso può avere dei corsi ad esso raggruppati, e da questi si possono distinguere diverse casistiche di comportamento.
Il primo caso prevede che nei corsi raggruppati non vi siano corsi che fanno parte dello stesso corso di studi, e o aventi professori in comune, per cui la programmazione potrebbe eseguire un posizionamento senza rilassamento dei vincoli.

Un secondo caso considera la possibilità di avere tra i raggruppati un corso che ha un corso di studi in comune, per cui si avrebbe la violazione del primo vincolo. Tale eccezione è stata gestita dando priorità al raggruppamento, per cui il programma cercherà di eseguire il posizionamento del raggruppamento nel suo complesso, per un dato giorno e orario di inizio, ma questo non avrà successo fintanto che il primo vincolo sarà in vigore. Al rilassamento del vincolo il posizionamento avrà successo e verrà segnalata la violazione solamente per i corsi interessati.
Tra le casistiche si è tenuto in conto anche la possibilità di errori nella preparazione allo scheduling, con l'assegnazione tra i corsi raggruppati di corsi spenti per un dato anno accademico, o di corsi appartenenti ad altri semestri. Questa tipologia di errore è particolare in quanto può dipendere dall'anno che si sta programmando. A questo scopo viene eseguito un controllo sulla coerenza del semestre tra tutti i corsi di un dato raggruppamento, e se si hanno delle discordanze si ha l'interruzione dello scheduling con segnalazione dell'errore in questione.
Un'ultimo punto è la possibilità che vi siano professori condivisi tra i corsi di uno stesso raggruppamento. Questo comporterà, come nel caso del vincolo uno, che i corsi in questione non vengano posizionati fintanto che non viene rilassato il terzo vincolo, dopo di che si avrà la segnalazione della violazione per i soli corsi che fanno uso di tali professori, solamente se risultano già impegnati.
Nel caso si abbiano problemi con la verifica del progetto nel file ".zip" si dà disponibilità di clonare il progetto dal link https://github.com/LBrignone/Progetto_AeC.git.

Appendice e calcolo della complessità

In questa sezione si elencheranno le funzioni ed i metodi per ciascuna classe, usati nel progetto con la relativa complessità computazionale.
I costruttori e i distruttori, qualora non riportino una struttura diversa da quella di default non verranno riportati.
Si riporta che tutte le funzioni e metodi che presentano una complessità $\mathcal{O}(1)$ indica una complessità costante, per cui si trascura la possibile costante moltiplicativa M.

Funzione / Metodo	Complessità
Person	
Person(const string& name, const string& surname, const string& mail)	$\mathcal{O}(1)$
string& getName() const	$\mathcal{O}(1)$
void setName(const string& name)	$\mathcal{O}(1)$

Funzione / Metodo	Complessità
string& getSurname() const	$\mathcal{O}(1)$
void setSurname(const string& surname)	$\mathcal{O}(1)$
string& getMail() const	$\mathcal{O}(1)$
bool setMail(const string& mail)	$\mathcal{O}(n)$
string& getId() const	$\mathcal{O}(1)$
const bool operator <(const Person& personToCompare) const	$\mathcal{O}(1)$
Professor	
Professor(const Professor& toCopy)	$\mathcal{O}(1)$
bool setId(const string& id)	$\mathcal{O}(1)$
bool generateNewId(const string& id)	$\mathcal{O}(1)$
bool isAvailExamProgramming(const Date& startData, const Date& stopData, const Date& academicYear)	$\mathcal{O}(n)$ n -> dimensione lista indisponibilità
const list<AvailForExam>& getUnavailListByAcademicYear(string& errorHandling, const Date& academicYear) const	$\mathcal{O}(\log(n))$ n -> dimensione della mappa indisponibilità
Date getMinDateForUnavail() const	$\mathcal{O}(1)$
Date getMaxDateForUnavail() const	$\mathcal{O}(1)$
bool setUnavailability(const list<AvailForExam>& unavailDatesList, const Date& academicYear)	$\mathcal{O}(\log(n))$ n -> dimensione della mappa
void appendUnavailabilityForExam(const Date &startUnavail, const Date &stopUnavail)	$\mathcal{O}(1)$
void clearMapAcademicYearUnavailability(const Date& academicYear)	$\mathcal{O}(\log(n) + m)$ n -> dimensione della mappa m -> numero di chiavi nella mappa
const Professor operator ++(int)	$\mathcal{O}(1)$
bool operator <(const Professor& toCompare)	$\mathcal{O}(1)$
Professor& operator =(const Professor& toCopy)	$\mathcal{O}(n)$
Professor& operator =(const list<Professor>::const_iterator & toCopy)	$\mathcal{O}(n)$
ostream& operator <<(ostream& os) const	$\mathcal{O}(1)$
Student	
bool setId(const string& id)	$\mathcal{O}(1)$
bool generateNewId(const string& id)	$\mathcal{O}(1)$

Funzione / Metodo	Complessità
ostream& operator <<(ostream& os) const	$\mathcal{O}(1)$
Calssroom	
Classroom(const Classroom& toCopy)	$\mathcal{O}(1)$
const string& getId() const	$\mathcal{O}(1)$
bool setId(const string& id)	$\mathcal{O}(1)$
bool generateNewId(const string& id)	$\mathcal{O}(1)$
char getType() const	$\mathcal{O}(1)$
bool setType(const char& type)	$\mathcal{O}(1)$
const string& getClassroomName() const	$\mathcal{O}(1)$
void setClassroomName(const string& classroomName)	$\mathcal{O}(1)$
int getCapacity() const	$\mathcal{O}(1)$
bool setCapacity(const int& capacity)	$\mathcal{O}(1)$
int getExamCapacity() const	$\mathcal{O}(1)$
bool setExamCapacity(const int& examCapacity)	$\mathcal{O}(1)$
bool operator <(const Classroom& classroomToCompare)	$\mathcal{O}(1)$
Classroom& operator =(const Classroom& toAssign)	$\mathcal{O}(1)$
ostream& operator <<(ostream& os) const	$\mathcal{O}(1)$
AssociateProfessor	
AssociateProfessor(const AssociateProfessor& toCopy)	$\mathcal{O}(1)$
list<Professor>::iterator getProfessorPointer() const	$\mathcal{O}(1)$
void setProfessorPointer(list<Professor>::iterator addressOfProfessor)	$\mathcal{O}(1)$
int getLessonH() const	$\mathcal{O}(1)$
bool setLessonH(const int& lessonH)	$\mathcal{O}(1)$
int getExerciseH() const	$\mathcal{O}(1)$
bool setExerciseH(const int& exerciseH)	$\mathcal{O}(1)$
int getLabH() const	$\mathcal{O}(1)$
bool setLabH(const int& labH)	$\mathcal{O}(1)$
bool getIsMain() const	$\mathcal{O}(1)$
void setIsMain(const bool& isMain)	$\mathcal{O}(1)$
bool getToCheck() const	$\mathcal{O}(1)$

Funzione / Metodo	Complessità
void setToCek(const bool& modify)	$\mathcal{O}(1)$
void clear(const list<Professor>::iterator initToProfessorListEnd)	$\mathcal{O}(1)$
AssociateProfessor& operator =(const AssociateProfessor& toCopy)	$\mathcal{O}(1)$
Course	
Course(const Course& toCopy)	$\mathcal{O}(n)$
string getId() const	$\mathcal{O}(1)$
bool setId(const string& id)	$\mathcal{O}(1)$
bool generateNewId(const string& lastId)	$\mathcal{O}(n)$
int getStartYear() const	$\mathcal{O}(1)$
bool setStartYear(const int& startYear)	$\mathcal{O}(1)$
string getParallelCoursesId() const	$\mathcal{O}(1)$
bool setParallelCoursesId(const string& parallelCoursesId)	$\mathcal{O}(1)$
void clearParallelCourseId()	$\mathcal{O}(n)$
string getTitle() const	$\mathcal{O}(1)$
void setTitle(const string &title)	$\mathcal{O}(1)$
int getCfu() const	$\mathcal{O}(1)$
bool setCfu(const int& cfu)	$\mathcal{O}(1)$
int getParallelCoursesNumber() const	$\mathcal{O}(1)$
bool setParallelCoursesNumber(const int& parallelCoursesNumber)	$\mathcal{O}(1)$
bool isActiveCourse() const	$\mathcal{O}(1)$
void setActiveCourse(const bool& activeCourse)	$\mathcal{O}(1)$
int getCourseLessonH() const	$\mathcal{O}(1)$
bool setCourseLessonH(const int& courseLessonH)	$\mathcal{O}(1)$
int getCourseExerciseH() const	$\mathcal{O}(1)$
bool setCourseExerciseH(const int& courseExerciseH)	$\mathcal{O}(1)$
int getCourseLabH() const	$\mathcal{O}(1)$
bool setCourseLabH(const int& courseLabH)	$\mathcal{O}(1)$
list<AssociateProfessor>& getListAssistant()	$\mathcal{O}(1)$
const list<AssociateProfessor>& getListAssistant() const	$\mathcal{O}(1)$

Funzione / Metodo	Complessità
int setListAssistant(const list<AssociateProfessor>& assistant, string& errorInAssistant)	$\mathcal{O}(n)$ n -> dimensione lista professori assegnati al corso
void setListAssistantNoChecks(const list<AssociateProfessor>& assistant)	$\mathcal{O}(n)$ n -> dimensione lista professori assegnati al corso
string getExamType()	$\mathcal{O}(1)$
bool setExamType(const string& examType)	$\mathcal{O}(n)$
char getExamClassroomType() const	$\mathcal{O}(1)$
bool setExamClassroomType(char examClassroomType)	$\mathcal{O}(1)$
int getEntranceTime() const	$\mathcal{O}(1)$
bool setEntranceTime(int entranceTime)	$\mathcal{O}(1)$
int getExitTime() const	$\mathcal{O}(1)$
bool setExitTime(int exitTime)	$\mathcal{O}(1)$
int getExamDuration() const	$\mathcal{O}(1)$
bool setExamDuration(int examDuration)	$\mathcal{O}(1)$
int getPartecipants() const	$\mathcal{O}(1)$
bool setPartecipants(int partecipants)	$\mathcal{O}(1)$
list<string> getListGroupedId() const	$\mathcal{O}(1)$
bool setListGroupedId(const list<string> &groupingId)	$\mathcal{O}(n)$ n -> dimensione lista corsi raggruppati
bool appendGroupedId(const string& toAppend)	$\mathcal{O}(1)$
void deleteGroupedId()	$\mathcal{O}(1)$
void inheritCourse(const list<Course>::const_iterator& toInherit)	$\mathcal{O}(n)$ n -> dimensione lista professori assegnati al corso
void clearCFields()	$\mathcal{O}(n)$ n -> dimensione della stringa
void clearAFields()	$\mathcal{O}(n)$ n -> dimensione della stringa
ostream& printCourseOrganization(ostream& os) const	$\mathcal{O}(1)$
ostream& printCourseOrganizationAcademicYearOpening(ostream& os) const	$\mathcal{O}(1)$
ostream& printCourseOrganizationVersionOpening(ostream& os, const bool& first) const	$\mathcal{O}(1)$

Funzione / Metodo	Complessità
void printCourseOrganizationAcademicYearClosing(ostream& os) const	$\mathcal{O}(n)$ n -> dimensione lista corsi raggruppati
bool operator <(const Course& course) const	$\mathcal{O}(1)$
Course& operator =(const Course& toCopy)	$\mathcal{O}(n)$ n -> dimensione lista professori assegnati al corso
Course& operator =(const list<Course>::iterator& toCopy)	$\mathcal{O}(n)$ n -> dimensione lista professori assegnati al corso
ostream& operator <<(ostream& os) const	$\mathcal{O}(n)$ n -> dimensione lista professori assegnati al corso
CourseOfStudy	
string getCourseOfStudyId() const	$\mathcal{O}(1)$
bool generateCourseOfStudyId(const string& lastCourseOfStudy)	$\mathcal{O}(1)$
bool setCourseOfStudyId(const string& toSetcourseOfStudyId)	$\mathcal{O}(1)$
string getGraduationType() const	$\mathcal{O}(1)$
bool setGaraduationType(const string& graduationType)	$\mathcal{O}(n)$ n -> dimensione del vettore
const list<string>& getListOfCoursesBySemester(const int& key) const	$\mathcal{O}(\log(n))$ n -> dimensione della mappa
bool setListOfCoursesBySemester(string& errorHandling, const int& semesterKey, const string& courseId)	$\mathcal{O}(n \cdot m)$ n -> dimensione della mappa m -> dimensione della lista
void setCompleteListOfCoursesBySemester(const int& semesterKey, const list<string>& courseIdList)	$\mathcal{O}(\log(n) \cdot m)$ n -> dimensione della mappa m -> dimensione della lista
bool deleteEndedCourseFromActiveCourse(string& errorHandling, const string& courseId, const bool& allInactive)	$\mathcal{O}(n \cdot m)$ n -> dimensione della mappa m -> dimensione della lista dei corsi nel semestre
bool activateCourseFromEndedCourse(string& errorHandling, const string& courseId, const bool& allActive)	$\mathcal{O}(n)$ n -> dimensione lista dei corsi nei semestri
int findCourse(int startSemester, const string& courseId)	$\mathcal{O}(n)$ n -> dimensione lista dei corsi nei semestri
int findCourse(int startSemester, const string& courseId) const	$\mathcal{O}(n)$ n -> dimensione della mappa o della lista
ostream& operator <<(ostream& os) const	$\mathcal{O}(n \cdot m)$ n -> dimensione della mappa m -> dimensione della lista

Funzione / Metodo	Complessità
Date	
Date(const Date& dateToCopy)	$\mathcal{O}(1)$
Date()	$\mathcal{O}(1)$
Date(const int& hour)	$\mathcal{O}(1)$
Date(const int& day, const int& month, const int& year)	$\mathcal{O}(1)$
Date(const int& minutes, const int& hour, const int& day, const int& month, const int& year)	$\mathcal{O}(1)$
int getMinutes() const	$\mathcal{O}(1)$
int getHour() const	$\mathcal{O}(1)$
bool setHour(int hour)	$\mathcal{O}(1)$
int getDay() const	$\mathcal{O}(1)$
bool setDay(int day)	$\mathcal{O}(1)$
int getMonth() const	$\mathcal{O}(1)$
bool setMonth(int month)	$\mathcal{O}(1)$
int getYear() const	$\mathcal{O}(1)$
bool setYear(int year)	$\mathcal{O}(1)$
string getCompleteDate() const	$\mathcal{O}(1)$
void getAcademicYear(ostream& os) const	$\mathcal{O}(1)$
string getAcademicYear() const	$\mathcal{O}(1)$
void getTimeSlot(ostream& os)	$\mathcal{O}(1)$
void increaseAcademicYear()	$\mathcal{O}(1)$
Date& operator =(const Date& date)	$\mathcal{O}(1)$
bool operator <=(const Date& date) const	$\mathcal{O}(1)$
bool operator >=(const Date& date) const	$\mathcal{O}(1)$
bool operator >(const Date& date) const	$\mathcal{O}(1)$
bool operator <(const Date& date) const	$\mathcal{O}(1)$
bool operator ==(const Date& date) const	$\mathcal{O}(1)$
bool operator !=(const Date& date) const	$\mathcal{O}(1)$
int operator -(const Date& rValDate) const	$\mathcal{O}(1)$
Date operator ++(int)	$\mathcal{O}(1)$

Funzione / Metodo	Complessità
Date operator +(const int& timeLapse) const	$\mathcal{O}(1)$
ostream& operator <<(ostream& os) const	$\mathcal{O}(1)$
SessionScheduler	
SessionScheduler(string& errorHandling, const int& refAcademicYear, const list<Course>& databaseCourses, list<Professor>& databaseProfessor, const list<Classroom>& databaseClassroom, const list<CourseOfStudy>& databaseCourseOfStudy, const Date& startRef, const Date& stopRef, const int& sessionNumber)	$\mathcal{O}((n^3 \cdot m) + (n^3 \cdot r))$ n -> dimensione lista dei corsi m -> dimensione lista corsi raggruppati r -> dimensione lista corsi di studio
void groupedCoursesScheduling(const int& sessionNumber, const int& semesterToSchedule, const Date& startDate, list<Professor>& databaseProfessorList, const int& academicYearRef)	$\mathcal{O}(n \cdot m \cdot j \cdot p \cdot k \cdot r \cdot q)$ n -> dimensione vettore raggruppamenti m -> dimensione vettore corsi raggruppati j -> giorni del calendario p -> slot del giorno k -> dimensione vettore aule r -> dimensione lista corsi di studio q -> dimensione lista corsi di studio in calendario
void outputSessionFile(const string& fileBaseName, const int& sessionNumber, const Date& sessionDateStartRef)	$\mathcal{O}(n \cdot m \cdot p \cdot r \cdot k)$ n -> giorni del calendario m -> slot del giorno p -> dimensione vettore aule r -> dimensione lista corsi di studio per giorno, slot e aula k -> dimensione stringa
void outputWarningFile(const string& fileBaseName, const int& sessionNumber)	$\mathcal{O}(n \cdot m \cdot p \cdot k)$ n -> dimensione della mappa m -> dimensione vettore raggruppamenti p -> dimensione vettore corsi raggruppati K -> dimensione lista corsi di studio
bool coursesForGivenAcademicYear(string& errorHandling, const list<Course>& databaseCourses, const int& refAcademicYear, list<Professor>& databaseProfessor)	$\mathcal{O}(n)$ n -> dimensione della mappa o della lista
int groupingCoursesBySemester(string& errorHandling, const list<CourseOfStudy>& databaseCourseOfStudy)	$\mathcal{O}((n^3 \cdot m) + (n^3 \cdot r))$ n -> dimensione lista corsi m -> dimensione lista corsi temporanea r -> dimensione lista corsi di studio
bool sessionScheduleFromDate(string& errorHandling, const Date& startDate, const Date& stopDate, const int& sessionNumber)	$\mathcal{O}(n \cdot m)$ n -> dimensione vettore aule m -> numero di elementi da inserire per il calendario

Funzione / Metodo	Complessità
bool constrain_1(const vector<pair<Classroom, vector<vector<examScheduled>>>>& copyOfDatesPlanning, const int& dayRef, map<int, vector<vector<courseOrgBySemester>>>& copyCoursesForConstrainViolation, const int& semesterRef, const int& groupRef, const int& courseRef)	$\mathcal{O}(n \cdot m \cdot p \cdot r \cdot k)$ n -> giorni da controllare m -> slot del giorno da controllare p -> dimensione vettore aule r -> dimensione lista corsi di studio k -> dimensione lista corsi di studio in calendario
bool constrain_2(const vector<pair<Classroom, vector<vector<examScheduled>>>>& copyOfDatesPlanning, const int& dayRef, const int& slotRef, const int& constrainRelaxed, map<int, vector<vector<courseOrgBySemester>>>& copyCoursesForConstrainViolation, const int& semesterRef, const int& groupRef, const int& courseRef)	$\mathcal{O}(n \cdot m \cdot p \cdot \log(r))$ n -> giorni da controllare m -> slot del giorno da controllare p -> dimensione vettore aule r -> accesso alla mappa
bool constrain_3(const vector<pair<Classroom, vector<vector<examScheduled>>>>& copyOfDatesPlanning, const int& academicYearRef, const Date& startSessionDate, const int& dateIncrement, const int& slotRef, const Course& courseToInsert, list<Professor>& professorListToVerifyAndUpdate, const bool& constrain4Violated)	$\mathcal{O}(n \cdot m)$ n -> dimensione lista corsi m -> dimensione lista professori assegnati
bool constrain_4(const vector<pair<Classroom, vector<vector<examScheduled>>>>& copyOfDatesPlanning, const Course& courseToInsert, const int& dayRef, const int& slotRef, int& classroomChosen)	$\mathcal{O}(n \cdot m)$ n -> dimensione vettore aule m -> numero di slot per il giorno
void coursePositioning(vector<pair<Classroom, vector<vector<examScheduled>>>>& copyOfDatesPlanning, const Course& courseToInsert, const int& classroomChosen, const int& dayRef, const int& slotRef, const list<string>& courseOfStudyRelatedToCourse)	$\mathcal{O}(n)$ n -> dimensione lista corsi di studio da posizionare in calendario
void resetDataFromDatabase(map<int, vector<vector<courseOrgBySemester>>>& groupedCourses, vector<pair<Classroom, vector<vector<examScheduled>>>>& datesPlanning)	$\mathcal{O}(n \cdot m \cdot k \cdot (r \cdot \log(j) + p \cdot \log(j))$ n -> dimensione della mappa m -> dimensione vettore raggruppamenti k -> dimensione vettore corsi raggruppati r -> dimensione vettore vincoli j -> dimensione mappa originale p -> dimensione lista corsi di studio
AddFileHandling	
int StudentInputFile(string& errorHandling, const string& studentsFileName, list<Student>& studentList, const bool& isDb)	$\mathcal{O}(n \cdot m)$ n -> numero righe file m -> numero “elementi” della riga
int ProfessorInputFile(string& errorHandling, const string& professorFileName, list<Professor>& professorList, const bool& isDb)	$\mathcal{O}(n \cdot m)$ n -> numero righe file m -> numero “elementi” della riga

Funzione / Metodo	Complessità
int ClassroomInputFile(string& errorHandling, const string& classroomFileName, list<Classroom>& classroomList, const bool& isDb)	$\mathcal{O}(n \cdot m)$ n -> numero righe file m -> numero "elementi" della riga
int CourseInputFile(string& errorHandling, const string& courseFileName, list<Course>& courseList, list<Professor>& tmpProfessorList, const bool& isDb);	$\mathcal{O}(n \cdot m \cdot p)$ n -> numero righe file m -> numero "elementi" della riga p -> dimensione lista professori database
int CourseOfStudyInputFile(string& errorHandling, const string& courseOfStudyFileName, list<CourseOfStudy>& studyCoursesList, const bool& isDb);	$\mathcal{O}(n \cdot m \cdot p \cdot r \cdot k \cdot v)$ n -> numero righe file m -> numero "elementi" della riga p -> dimensione mappa semestri r -> dimensione lista corsi semestre k -> dimensione della mappa v -> dimensione della lista
int ExamSessionInputFile(string& errorHandling, const string& examSessionStringFileName, map<Date, vector<Date>>& examSessionPerAcademicYear, bool readDatabase);	$\mathcal{O}(n \cdot m \cdot k \cdot r)$ n -> numero righe file m -> numero coppie di date k -> numero date r -> numero campi della data
int ProfessorUnavailabilityInputFile(string& errorHandling, const string& professorUnavailabilityFile, list<Professor>& professorList, const string& academicYear, const bool& isDb);	$\mathcal{O}(n \cdot m \cdot k \cdot r)$ n -> numero righe file m -> numero "elementi" della riga k -> numero coppie date r -> numero campi della data
UpdateFileHandling	
int StudentToUpdateFile (string& errorHandling, const string& studentsFileName, list<Student>& studentList)	$\mathcal{O}(n \cdot m \cdot p)$ n -> numero righe file m -> numero "elementi" della riga p -> dimensione lista studenti database
int ProfessorToUpdateFile (string& errorHandling, const string& professorFileName, list<Professor>& professorList)	$\mathcal{O}(n \cdot m \cdot p)$ n -> numero righe file m -> numero "elementi" della riga p -> dimensione lista professori database
int ClassroomToUpdateFile (string& errorHandling, const string& classroomFileName, list<Classroom>& classroomList)	$\mathcal{O}(n \cdot m \cdot p)$ n -> numero righe file m -> numero "elementi" della riga p -> dimensione lista aule database
int CourseOfStudyToUpdateFile (string& errorHandling, const string& courseOfStudyFileName, list<CourseOfStudy>& courseOfStudyList)	$\mathcal{O}(n \cdot m \cdot p \cdot r \cdot k)$ n -> numero righe file m -> numero "elementi" della riga p -> numero "elementi" del semestre r -> dimensione della mappa k -> dimensione della lista corsi del semestre
InsertFileHandling	

Funzione / Metodo	Complessità
CourseToInsertFile(string& errorHandling, const string& courseFileName, list<Course>& databaseCourseList, list<Professor>& professorList, list<CourseOfStudy>& courseOfStudy)	$\mathcal{O}(n \cdot m \cdot p \cdot r \cdot k)$ n -> numero righe file m -> numero "elementi" della riga p ->dimensione lista corsi di studio r -> dimensione della mappa k -> dimensione della lista corsi del semestre
OutputOnDatabaseHandling	
int updateStudentDatabaseFile(string& errorHandling, const string& databaseStudentFileName, const list<Student>& updatedStudentList)	$\mathcal{O}(n)$ n -> dimensione lista studenti
int updateProfessorDatabaseFile(string& errorHandling, const string& databaseProfessorFileName, const list<Professor>& updatedProfessorList)	$\mathcal{O}(n)$ n -> dimensione lista professori
int updateClassroomDatabaseFile(string& errorHandling, const string& databaseClassroomFileName, const list<Classroom>& updatedClassroomList)	$\mathcal{O}(n)$ n -> dimensione lista aule
int updateCourseOfStudyDatabaseFile(string& errorHandling, const string& databaseCourseOfStudyFileName, const list<CourseOfStudy>& updatedCourseOfStudyList)	$\mathcal{O}(n)$ n -> dimensione lista corsi di studio
int updateCourseDatabaseFile(string& errorHandling, const string& databaseCourseFileName, const list<Course>& updatedCourseList)	$\mathcal{O}(n \cdot (m + k))$ n -> dimensione lista corsi di studio m -> dimensione lista corsi raggruppati k -> dimensione lista professori assegnati al corso
int updateExamSessionDatabaseFile(string& errorHandling, const string& databaseExamSessionFileName, const map<Date, vector<Date>>& updatedExamSessionMap)	$\mathcal{O}(n \cdot m)$ n -> dimensione mappa sessioni m -> dimensione vettore date sessioni
int updateUnavailabilityDatabaseFile(string& errorHandling, const string& databaseUnavailabilityFileName, list<Professor>& updatedProfessorList)	$\mathcal{O}(n \cdot m \cdot k \cdot \log(r))$ n -> numero anni indisponibilità m -> dimensione lista professori k -> dimensione lista indisponibilità r -> dimensione mappa indisponibilità
FindSomethingInList	
list<Student>::const_iterator findStudent(const list<Student>& studentList, const string& idToFind)	$\mathcal{O}(n)$ n -> dimensione lista studenti
list<Professor>::const_iterator findProfessor(const list<Professor>& professorList, const string& idToFind)	$\mathcal{O}(n)$ n -> dimensione lista professori
list<Classroom>::const_iterator findClassroom(const list<Classroom>& classroomList, const string& idToFind)	$\mathcal{O}(n)$ n -> dimensione lista aule

Funzione / Metodo	Complessità
list<Course>::const_iterator findCourse(const list<Course>& courseList, const string& idToFind)	$\mathcal{O}(n)$ n -> dimensione lista corsi
list<Course>::const_iterator findCourse(const list<Course>& courseList, const string& idToFind, const int& academicYear)	$\mathcal{O}(n)$ n -> dimensione lista corsi
list<Course>::const_iterator findCourse(const list<Course>& courseList, const string& idToFind, const int& academicYear, const string& parallelVersion)	$\mathcal{O}(n)$ n -> dimensione lista corsi
list<Course>::const_iterator findCourseLastForId(const list<Course>& courseList, const string& idToFind, const list<Course>::const_iterator& startPos)	$\mathcal{O}(n)$ n -> dimensione lista corsi
list<Course>::const_iterator findCourseLastForIdAndYear(const list<Course>& courseList, const string& idToFind, const int& academicYear, const list<Course>::const_iterator& startPos)	$\mathcal{O}(n)$ n -> dimensione lista corsi
bool findCourseTitle(const list<Course>& courseList, const string& titleToFind)	$\mathcal{O}(n)$ n -> dimensione lista corsi
list<CourseOfStudy>::const_iterator findCourseOfStudy(const list<CourseOfStudy>& courseOfStudyList, const string& idToFind)	$\mathcal{O}(n)$ n -> dimensione lista corsi di studio
list<string> findCourseOfStudy(string& errorHandling, const list<CourseOfStudy>& courseOfStudyList, const string& idToFind)	$\mathcal{O}(n \cdot m)$ n -> dimensione lista corsi di studio m -> dimensione lista corsi
list<string> findCourseIdGrouped(const list<Course>& courseList, const string& idToFind)	$\mathcal{O}(n \cdot m)$ n -> dimensione lista corsi m -> dimensione lista corsi raggruppati
Date findMaxAcademicYearUnavail(const list<Professor>& professorList)	$\mathcal{O}(n)$ n -> dimensione lista professori
bool comp(Professor professorToCompare, Professor minimum)	$\mathcal{O}(1)$
bool sortMethodForProf(Professor professorToCompare, Professor minimum)	$\mathcal{O}(1)$
bool sortMethodForClassroom(Classroom classroomToCompare, Classroom minimum)	$\mathcal{O}(1)$
bool sortMethodForCourse(struct courseOrgBySemester courseToCompare, struct courseOrgBySemester minimum)	$\mathcal{O}(1)$
bool sortMethodForPrintSchedule(const struct expandedScheduleForPrint& structToCompare, const struct expandedScheduleForPrint& minimum)	$\mathcal{O}(n)$ n -> dimensione delle stringhe da comparare

Funzione / Metodo	Complessità
bool sortMethodForPrintWarnings(const struct expandedScheduleForPrint& structToCompare, const struct expandedScheduleForPrint& minimum)	$\mathcal{O}(n)$ n -> dimensione delle stringhe da comparare
int approXimationFunct(const int& rightVal, const int& leftVal)	$\mathcal{O}(1)$
FillDatabaseList	
bool fillCourseDatabase (string& errorHandling, int versionCounter, list<Course>& databaseList, list<Course>& dummyCoursesList, list<Professor>& professorList)	$\mathcal{O}(n \cdot m \cdot p)$ n -> dimensione lista corsi temporanea m -> dimensione lista professori assegnati al corso temporaneo p -> dimensione lista professori assegnati al corso di database
bool insertCourseDatabase (string& errorHandling, int versionCounter, list<Course>& databaseList, list<Course>& dummyCourseList, const list<Professor>& profesorList)	$\mathcal{O}(n \cdot m \cdot p)$ n -> dimensione lista corsi temporanea m -> dimensione lista professori assegnati al corso temporaneo p -> dimensione lista professori assegnati al corso di database
bool fillAssociateProfessor (string& errorHandling, list<AssociateProfessor>& associateProfessorListDb, const list<AssociateProfessor>& associateProfessorListDummy, const list<Professor>& professorList)	$\mathcal{O}(n \cdot m)$ n -> dimensione lista professori assegnati al corso temporaneo m -> dimensione lista professori assegnati al corso di database
bool insertAssociateProfessor (string& errorHandling, const list<AssociateProfessor>& associateProfessorListDb, list<AssociateProfessor>& associateProfessorListDummy, const list<Professor>& professorList)	$\mathcal{O}(n \cdot m)$ n -> dimensione lista professori assegnati al corso temporaneo m -> dimensione lista professori assegnati al corso di database
PatternConstrainVerification	
bool parallelVersionProgression (string& errorHandling, int prevVersionid, const string& versionToVerify)	$\mathcal{O}(1)$
string generateVersion (int versionToGenerate)	$\mathcal{O}(1)$
bool versionCoherencyTest (string& errorHandling, int versionProgression, const string& versionToVerify)	$\mathcal{O}(1)$
bool examSessionAcademicYearCoherencyTest (string& errorHandling, int academicYearRef, const vector<Date>& sessionToVerify)	$\mathcal{O}(1)$
bool examSessionBeginEndVerification (string& errorHandling, const vector<Date>& sessionToVerify)	$\mathcal{O}(1)$
bool examSessionOrderVerification (string& errorHandling, const vector<Date>& sessionToVerify)	$\mathcal{O}(1)$

Funzione / Metodo	Complessità
bool sessionDurationConstrainVerification (string& errorHandling, const vector<Date>& sessionToVerify)	$\mathcal{O}(1)$
bool unavailabilityDatesVerification (const AvailForExam& dateToVerify, const list<AvailForExam>& datesToVerifyWith)	$\mathcal{O}(n)$ n -> dimensione lista date di indisponibilità
bool putCourseInEndedCourses(string& errorHandling, const Course& courseToCompare, list<Course>& courseList, list<CourseOfStudy>& courseToHandle)	$\mathcal{O}(n \cdot m \cdot p)$ n -> dimensione lista corsi di studio m -> dimensione della mappa del corso di studio p -> dimensione della lista dei corsi nel semestre
bool removeCourseFromEndedCourses(string& errorHandling, const Course& courseToCompare, list<Course>& courseList, list<CourseOfStudy>& courseToHandle)	$\mathcal{O}(n \cdot m)$ n -> dimensione lista corsi di studio m -> dimensione lista dei corsi nei semestri
FieldVerificationForScheduling	
bool courseFieldVerification (string& errorHandling, const list<Course>& courseListToVerify, list<Professor>& professorList)	$\mathcal{O}(n \cdot m \cdot p)$ n -> dimensione lista corsi m -> dimensione lista corsi raggruppati p -> dimensione lista corsi
bool professorFieldVerification(string& errorHandling, const Professor& professorToVerify)	$\mathcal{O}(1)$
bool classroomFieldVerification(string& errorHandling, const list<Classroom>& databaseClassroomToVerify)	$\mathcal{O}(n)$ n -> dimensione lista aule
list<string> regroupingCoursesForCommonCourse(const list<Course>& courseToSchedule, const Course& idToFind)	$\mathcal{O}(n \cdot m \cdot p)$ n -> dimensione lista corsi già usati m -> dimensione lista corsi raggruppati con il corso p -> dimensione lista corsi
int groupedCoursesVerification(string& errorHandling, const list<string>& groupedCourses, const list<Course>& coursesToSchedule, const list<CourseOfStudy>& databaseCourseOfStudy, list<struct courseOrgBySemester>& courseListToSchedule, int& semester)	$\mathcal{O}(n \cdot m \cdot p)$ n -> dimensione lista corsi raggruppati m -> dimensione lista corsi di studio p -> dimensione lista corsi del semestre
void myUnique(list<string>& courseList)	$\mathcal{O}(n^2)$ n -> dimensione lista corsi

Dettagli tecnici di implementazione

Sistema operativo	macOS Monterey v. 12.6.2
Compiler	Apple clang version 14.0.0

Debugger	LLDB v.14.0.6
Processore	Intel Core i7