Corso di Programmazione ad Oggetti - A.A. 2015/2016
Corso di laurea in Ingegneria e Scienze Informatiche - Università di Bologna

Relazione relativa al Progetto ScienceSchoolSchedule: Sistema per la creazione e la gestione dell'orario universitario

Componenti del gruppo:

Anna Termopoli
Massimiliano Micca
Galya Genova,
Francesco Ceroni

Capitolo 1

Analisi

1.1 Requisiti

Il software che vi proponiamo, mira all'implementazione digitale dell'orario universitario. ScienceSchoolSchedule è un software il cui scopo è la creazione di un orario attraverso differenti funzionalità come quelle d'inserimento di una prenotazione.

Requisiti

Questa versione digitale dell'orario permetterà all'utente(la segreteria universitaria) di prenotare un'aula in base al giorno, l'orario, il docente e il corso da lui presieduto

- Si potrà inserire una prenotazione, cancellarla
- Il titolare di un corso potrà essere modificato da un nuovo docente
- Ci sarà la possibilità di aggiungere un aula differente da quelle presenti
- La tabella con tutte le prenotazioni potrà essere resettata
- Ci sarà la possibilità di esportare la tabella in un file formato Excel
- La tabella con le prenotazioni potrà avere viste differenti

1.2 Analisi e modello del dominio

Alla prima apertura del programma l'utente si troverà difronte a due pannelli distinti, in uno sarà presente una tabella vuota nata per contenere le prenotazioni , e nell'altro pannello saranno presenti tutte le funzionalità necessarie per creare l'orario.Ciascun utente dovrà essere in grado di compiere diverse azioni tra cui l'inserimento (FrameInsert), la cancellazione (FrameCancel), la creazione di un

aula (**FrameModify**), la modifica del titolare di un corso(**FrameModify**), l'esportazione in Excel (**MyListenerExcel**).

La difficoltà primaria sarà quella di riuscire a correlare le varie prenotazioni delle aule con attenti controlli per evitare errori di inserimento. Sarà importante gestire le prenotazioni delle aule non solo in base a quelle già esistenti ma anche secondo i requisiti richiesti espressamente dal committente. Molto importante sarà anche gestire l'implementazione delle viste, in quanto, come è già stato scritto, esistono diverse modalità per visualizzare le tabelle.

Alla base di tutto sarà necessario gestire un file di dati in cui vengono memorizzate tutte le prenotazioni ma anche il salvataggio delle modifiche che l'utente può fare sui dati già presenti di default. ScienceSchoolSchedule dovrà essere esportato come file Excel grazie all'utilizzo di apposite librerie.

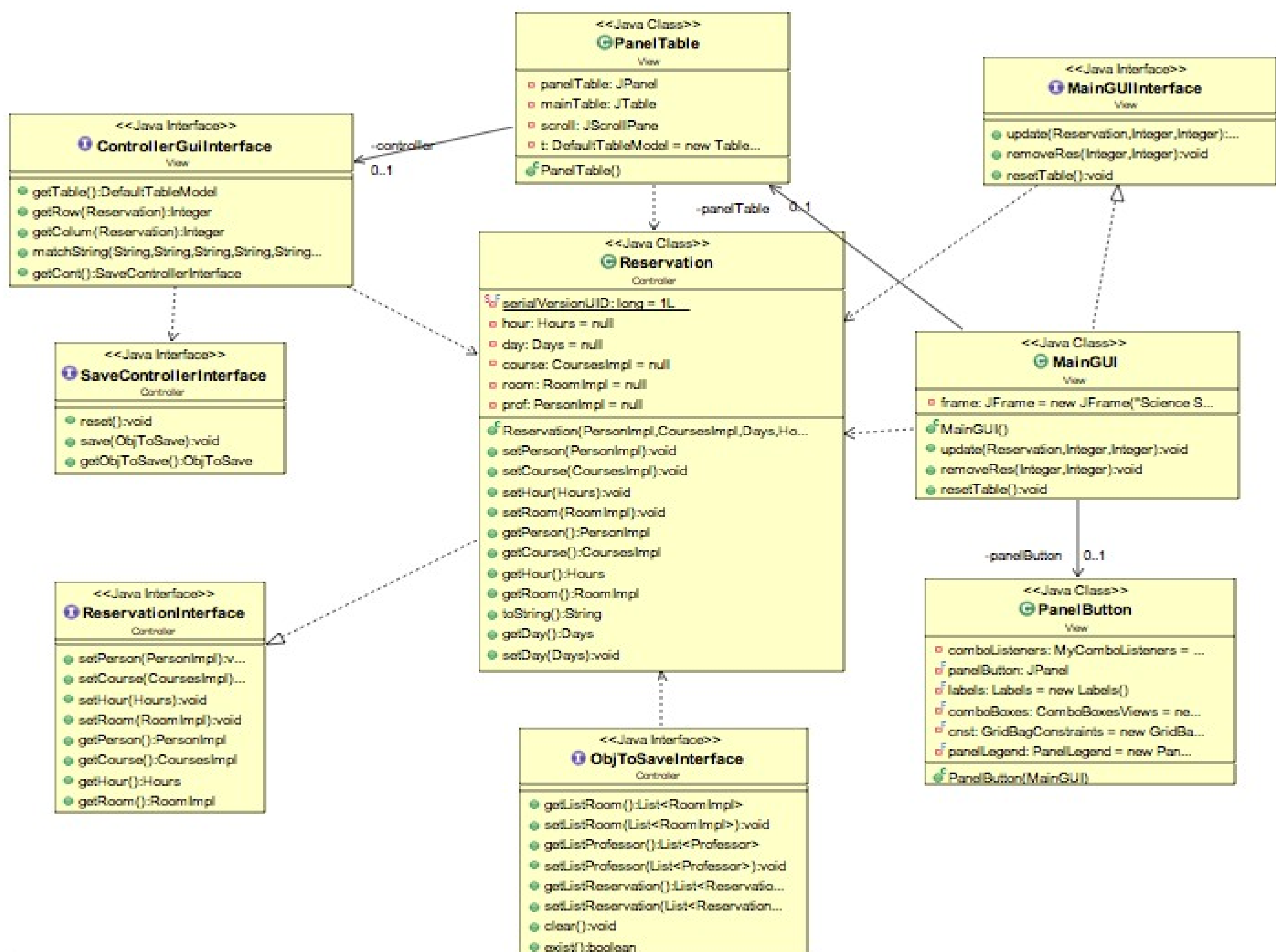


Figura 1: UML rappresentante lo schema generale del software.

Capitolo 2

Design

2.1 Architettura

Per la realizzazione di questo progetto abbiamo deciso di utilizzare il pattern architetturale MVC ripartito nel seguente modo:

- **Model**: si occupa del dominio del modello. In questa parte vengono definite tutte le entità del software, le diverse interazioni che devono avere fra loro, e anche le eccezioni. Il model è di fondamentale importanza in quanto vengono definite le entità che stanno alla base di tutto il progetto.

- **View**: si occupa dell'implementazione e visualizzazione grafica dei vari componenti. Molto importante è gestire al meglio la comunicazione con il controller per richiamare i dati dal file ma anche per i warning che devono apparire alla giusta occorrenza. La GUI dovrà visualizzare la tabella dell'orario, le funzionalità di inserimento, modifica, cancellazione, esportazione, vista. Inoltre verranno anche gestiti i dati stampati sulla tabella.

- **Controller**: è colui che fa comunicare tutte le vari parti del pattern MVC ,si occupa della parte logica del software, e creerà anche opportuni warning per consentire all'utente di creare un'orario il più funzionale possibile. Inoltre si occupa della creazione e gestione del file. In questa parte viene anche gestita l'esportazione della tabella in Excel(con apposite librerie). In diverse parti, il controller e la view si sono trovati a lavorare insieme per dare una logica e continuità al software.

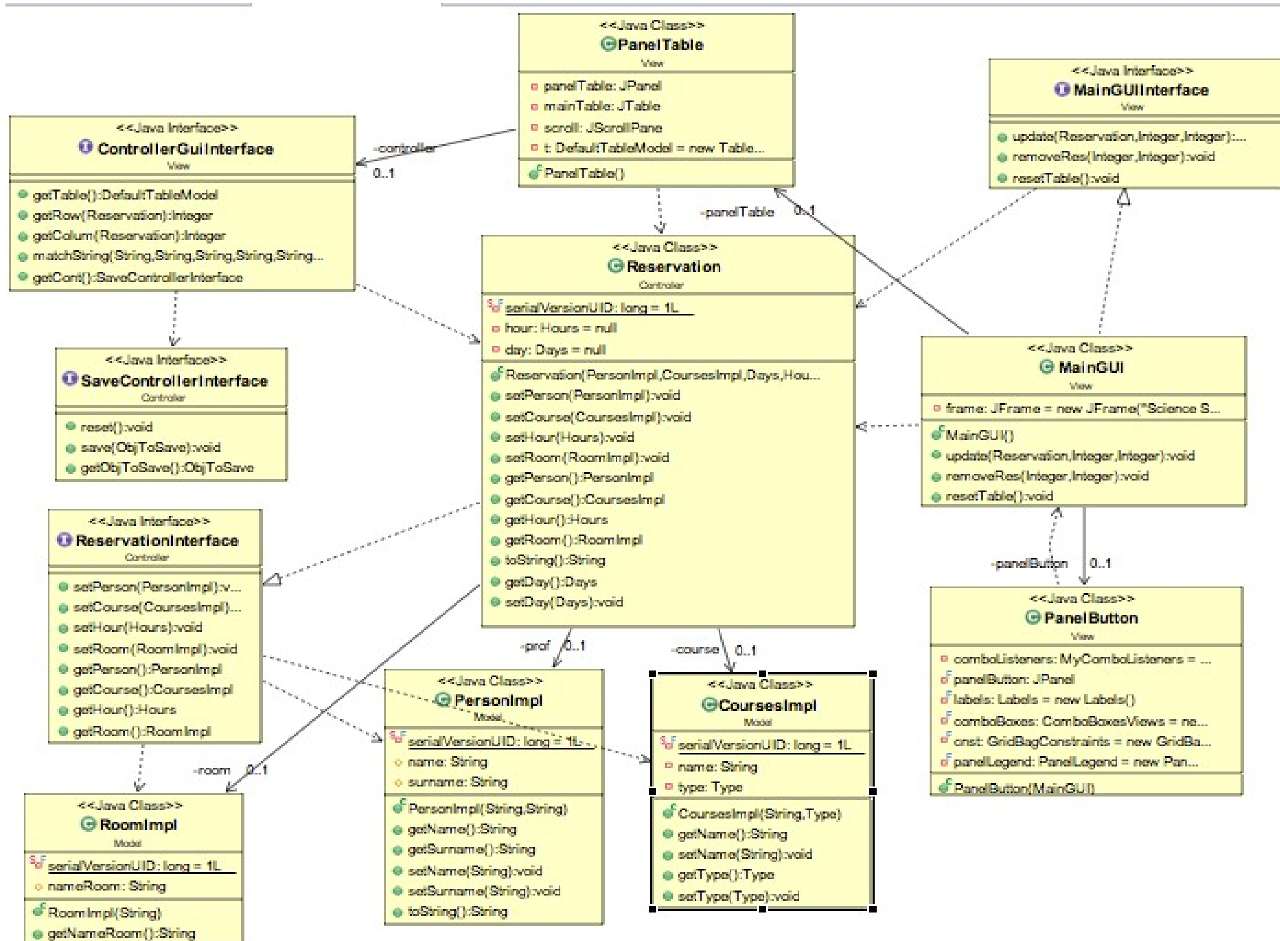


Figura 2. UML rappresentante lo schema MVC dell'applicazione.

2.2 Design dettagliato

2.2.1 Model

Il model si occupa della gestione e incapsulamento dello stato della applicazione, definisce i dati e le operazioni che possono essere eseguite.

Il model ha inoltre la responsabilità di notificare ai componenti della View eventuali modifiche, aggiornamenti in seguito alle richieste del Controller al fine di permettere alla View di presentarli agli utenti.

La parte del model è stata fatta da **Ceroni Francesco**.
La parte del Controller delle eccezioni e della gestione della tabella è stata fatta in collaborazione con il sig.re **Micca**
La progettazione del Model è iniziata dalla classe **Days** dove vengono specificati i giorni della settimana passandoli come enumeratori, successivamente la classe **Hours** formata dai vari periodi di lezioni della stessa tipologia della classe **Days**.
Successivamente è stata fatta la classe **Person** che estende la classe **ProfessorImpl** da dove eredità i metodi.
In seguito si è passati alla creazione delle varie liste, prima di tutto la **ListCourses** formata da tutti i corsi della facoltà, è stata implementata con enumeratori dove per ogni elemento gli viene passato il nome del corso e la tipologia di appartenenza che viene suddivisa in:

Triennale

- 1°Anno che non fa distinzioni tra le 2 tipologie di corso che sono Ingegneria e Scienze
- 2°Anno che invece fa distinzioni, quindi avremo le lezioni obbligatorie in comune tra i 2 corsi e la divisione tra Ingegneria e Scienze.
- 3°Anno vale la stessa modalità del 2°Anno in più avremo i corsi opzionali.

Magistrale

- 1°Anno
- 2°Anno in più del 1°Anno avremo i corsi opzionali.

Successivamente è stata creata la classe **ListProfessor** formata da tutti i professori della facoltà, dove per ogni professore viene passato il nome, cognome e i corsi da lui insegnati.
In seguito è stata creata la classe **ListRoom** formata da tutte le aule utilizzate dalla facoltà, nella classe viene passato il nome di ognuna di loro.
Infine abbiamo la classe **Type** dove vengono suddivisi i vari anni della facoltà e le varie tipologie di corso tra Ingegneria e Scienze, Opzionali e Obbligatori.

Successivamente sono state fatte le classi inerenti alle Eccezioni restituite dal programma suddivise in:

- **WarningException**, restituisce un messaggio di eccezione, un allert dove avverte l’utente della azione che si vuole effettuare.
- **ErrorException**, restituisce un messaggio di eccezione che non ti fa effettuare l’azione scelta.
- **FileOpenedException**, restituisce un messaggio di eccezione, che ti avverte del programma che è già aperto.

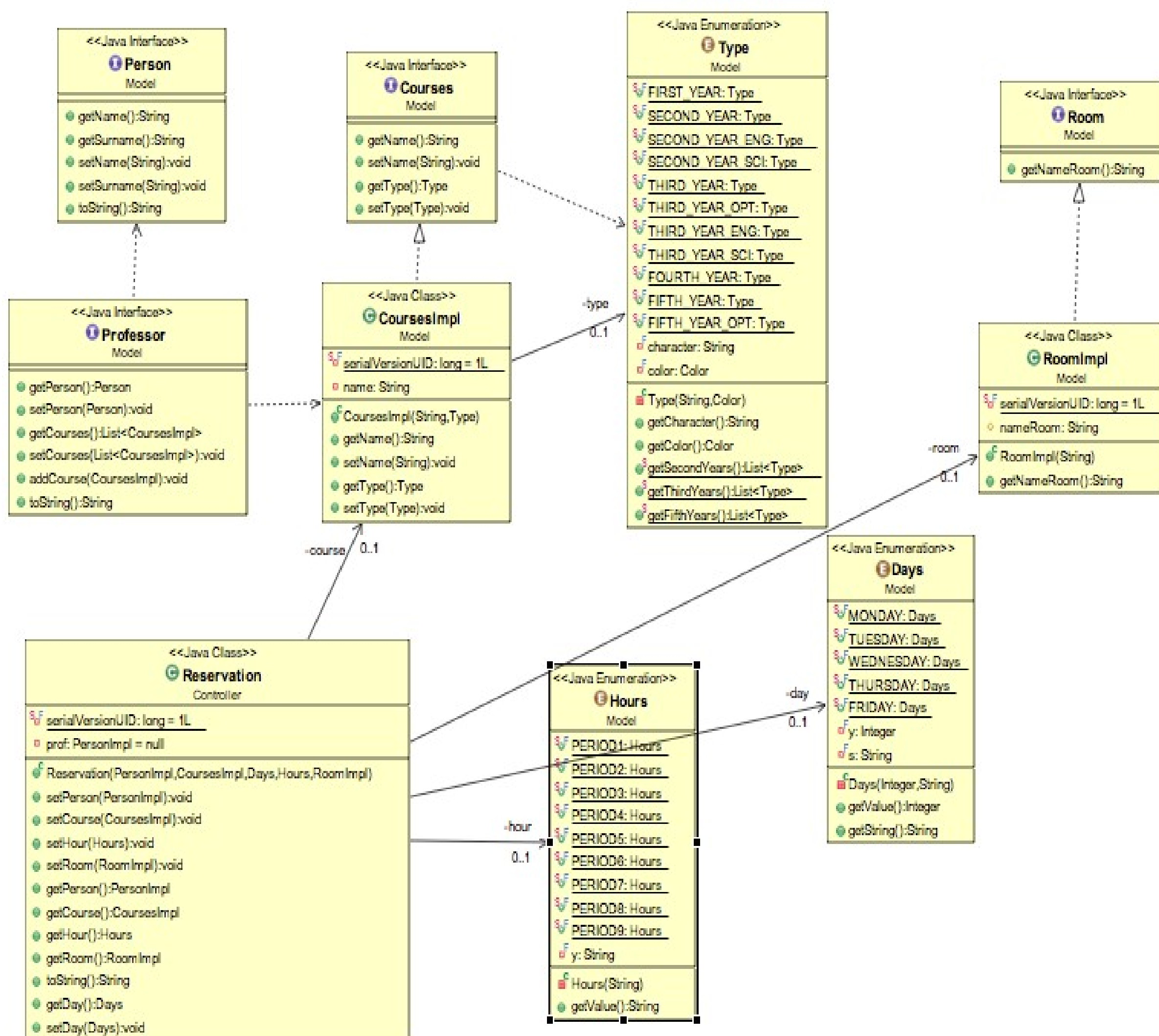


Figura 2.2.1 Model

2.2.2 View

La view si occupa di visualizzare attraverso un'interfaccia grafica gli aspetti primari del software: tabella, inserimento, cancellazione, modifica prenotazione, esportazione in Excel, viste.

Le ragazze **Anna Termopoli** e **Galya Genova** si sono suddivise il lavoro, ma hanno spesso collaborato per effettuare dei collegamenti tra listener e tabella.

Anna Termopoli:

La progettazione della view è partita dalla classe principale: la classe **MainGUI**. Nella classe **MainGUI** è stato creato un **JFrame** contenente due pannelli implementati nelle classi **PanelButton** e **PanelTable** . Nella classe **PanelButton** viene creato un JPanel con Layout impostato a **GridBagLayout** organizzato in questo modo:

1. Nella prima parte del pannello (quella a nord) abbiamo due JButton che sono implementati nella classe Buttons, che sono **Insert** e **generalView**.
   1.1. Il **JButton Insert** è collegato ad un listener che aprirà un JFrame istanziato nella classe **FrameInsert**. Nel frame della classe FrameInsert c'è un JPanel contenente le JComboBox istanziate nella classe **ComboBoxesViews**. La classe **FrameInsert** servirà quindi per prenotare un'aula a seconda del giorno, orario, professore e corso annesso. Nella classe **ComboBoxesViews** è stato creato un listener delle **JComboBox** contenente i professori(**cProfessor**), con lo scopo di rendere visibili nella **JComboBox** dei corsi(**cCorses**), solo i corsi che il docente insegna, per evitare di incombere nell'errore di scegliere un corso non presieduto dal prof selezionato. Il salvataggio della prenotazione sul file è stato fatto dentro il listener del JButton apply sempre nella classe **FrameInsert**. Ho deciso di aggiungere al **JFrame** delle **JComboBox** perché mi sembrava il metodo più intuitivo.

1.2. Di seguito ho creato un'altro **JButton** (**generalView**) che permette di visualizzare la tabella secondo la vista generale (cioè fa visualizzare tutte le prenotazioni presenti nel calendario). Il listener del secondo **JButton generalView** creato nella classe **PanelButton** è stato implementato dalla Sig.na **Genova**

2. Successivamente, scorrendo il **PanelButton** troviamo le **JComboBox** implementate nella classe **ComboBoxesViews**. Queste entità nascono per differenziare il modo in cui la tabella può essere vista. Affianco alla **JComboBox** abbiamo dei **JLabel** istanziati nella classe Label, che servono per descrivere le varie **JComboBox**. Ho scelto di implementare le **JComboBox** perché mi sembrava il modo più semplice per selezionare le viste. Il nostro relatore ci aveva suggerito di utilizzare delle **JTabbedPane** quindi dei folder per differenziare le diverse viste, ma a me sembrava più ordinato utilizzare le **JComboBox** anche per facilitare l'utente nella scelta di quale vista utilizzare (es: giorno -> lunedì). Nella classe **ComboBoxesViews** ,oltre ai metodi getter, ho implementato anche il metodo "**LisenerCombo**" che nel punto precedente avevo detto di aver utilizzato. I listener delle differenti **ComboBoxesViews** sono state tutte implementate in classe distinte.

3. Più sotto ho creato nella classe **Buttons** cinque **JButton** (**insertNew**, **saveInExcel**, **resetTable**, **cancel**, **exit**). Nel listener del bottone **InsertNew** ho richiamato la classe **FrameModify**.
La classe **FrameModify** serve per modificare il titolare di un corso o per inserire una nuova aula. Nella classe viene creato un nuovo **JFrame** (frameModify) contenente un **Panel** (panelModify) nel quale ho aggiunto due bottoni (prof, room).
Nel listener del **JButton** prof ho istanziato un nuovo **JFrame**, contenente **JButton**, **JLabel**, **JComboBox**, e **JTextField** per far scegliere all'utente il docente da modificare e inserire il nuovo titolare del corso. Nel listener del bottone **save**, salvo nella classe controller (istanziata nel Controller) il nuovo professore. Così facendo i dati sono direttamente salvati sul file. Ho aggiunto un errore nel caso in cui il

nome e cognome del nuovo prof siano stringhe vuote.
L’utente sarà obbligato ad inserire 2 stringhe piene altrimenti non ci sarà il salvataggio sul file.
Nel listener del **JButton** aula, ho istanziato un nuovo **JFrame** contenente **JButton**, **JLabel**, e **JTextField** per consentire all’utente di aggiungere una nuova aula, non ancora presente.
Nel listener del **Jbutton save** controlla che l’aula non sia già presente, facendo un confronto tra la nuova stringa inserita nel **JtextField** e la stringa presente nel file, se l’aula non
esiste, richiamo la classe controller per salvare il dato sul file. Finché l’aula inserita non è diversa da quella già presente sul file, essa non viene aggiunta.

4. Per ultimo, ho aggiunto al pannello **PanelButtons** un **JPanelPanelLegend**, che sarà utilizzato come legenda per il colore di un corso.
   In questo pannello ho creato due **JLabel**. Ho creato un ciclo for per tutti i **Type** e in un **JLabel** richiamo la funzione “**getColor**()” presente nella classe **Type**(che mi ritorna il colore associato) mentre nell’altro **JLabel** richiamo il metodo “**getCharacter**()” che mi ritorna una stringa con il nome del tipo di corso. Per questa implementazione ho preso spunto da un progetto degli anni precedenti che ho trovato su **BitBucket**. Mi sono appoggiata a quest’idea perché mi sembrava molto carina, facilmente intuibile e comprensibile. In questo modo sarà facile per l’utente capire di che tipo è il corso che gli interessa.
   In diverse occasioni ho aggiunto dei **Jpanel showMassageDialog**() oppure **showConfirmDialog**(). La mia prima intenzione è stata sempre quella di rendere le cose facili da gestire per l’utente. Penso che attraverso l’aggiunta di questi pannelli informativi, di errore e di conferma, ho aiutato e guidato il più possibile il fruitore nelle sue scelte. A volte come ad esempio nella modifica del professore o nell’aggiunta di un’aula, avviso l’utente che quello che sta inserendo non è giusto e lo obbligo a fare diversamente. Queste piccole accortezze sono molto importanti in quanto a volte possono sembrare accorgimenti banali, ma non lo sono affatto.

Galya Genova:

La mia parte consisteva nella implementazione del panello che contiene la tabella principale e le viste che permettono di visualizzare in modo personalizzato gli orari nella tabella.
Ho iniziato a implementare la classe **AbstractTableModel**() per personalizzare la mia **TableModel** (**TableGUI**), riscrivendo i metodi di default.
Andando avanti, ho deciso di cambiare il **AbstractTableModel**() con **DefaultTableModel**() che permette di usare i metodi come : **fireTableCellUpdate**(),**fireTableDataChanged**(), ecc.
Per adattare la grandezza delle colonne e le righe, ho trovato delle difficoltà, e ho usato il metodo **resizeColumnWidth**(**JTable table**), che ho trovato sul web. Questo metodo viene chiamato e implementato durante creazione della tabella nella classe **PanleTable**(), inoltre viene usato in alcune delle viste personalizzate. E' stata una mia decisione di non usare il metodo in tutte le viste, o più precisamente nelle viste per: **professore**, **corso**, **orario** e quella **generale**.
Un'altra parte della tabella consisteva nella impostazione di un renderer personalizzato. Per far ciò ho creato la classe **MyTableRenderer** che estende **JTextArea** e implementa **TableCellRenderer**.
Inizialmente essa estendeva **JLabel**, procedendo avanti, dopo tante prove ho deciso di cambiare la estensione con **JTextArea** che secondo me rendeva la visualizzazione della tabella migliore.
MyTableRenderer sovrascrive il metodo **getTableCellRendereComponent**(*JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column*).
Esso ha subito molte modifiche e grazie a **Massimiliano Micca** siamo riusciti a colorare le celle in base al tipo che sono.
Questo metodo funziona in tale modo:

- prende in input la tabella stessa e l'oggetto contenuto in ogni singola cella;
- per le celle scritte di default (cioè quelle contenenti gli orari e le aule ripetute per ogni singolo giorno) gli imposta un colore singolo per renderli individuabili.
- invece le celle contenenti le prenotazioni (nome corso e nome professore) vengono colorati a base del tipo del corso.

- le celle che non contengono dati rimangano in bianco;
- poi restituisce un componente, in questo caso la **JTextArea**;

La scelta di questa progettazione (la colorazione delle celle) e stata fatta per facilitare l'uso della applicazione all'utente e soprattutto, perché il file Excel fornito dal prof. **M. Viroli** (il relatore del progetto) conteneva la suddivisione delle materie per tipo individuato da un colore.

Una volta impostata la **TableGUI**(), tramite le classi sopra nominate, viene disegnata tramite la classe **ControllerGui**() (autore della classe **Massimiliano Micca**, il mio contributo è stato a creare i metodi **drawDefaultTable**() e **drawTable**()). Questa classe riempie la tabella con i titoli di default (gli orari e le aule ripetute per ogni singolo giorno) tramite il metodo **drawDefaulTable**() e visualizza tutte le prenotazione precedentemente salvate dall'utente con il metodo **drawTable**(). Essa restituisce la tabella **DefaultTableModel**(), così che ad ogni avvio dell'applicazione potrebbe visualizzare tutte le prenotazioni salvate sul file.
Finita la fase di progettazione della tabella o più precisamente la **TableGUI**(), essa viene istanziata nella classe **PanelTable** insieme a una **JTable** e **JScrollPane**.
PanelTable() contiene i metodi **update**(*TableGui* p), **remove**(*int row, int column*) e **setNew**(*Reservation res, int row, int colum*) che rispettivamente aggiornano la tabella quado viene rimosso o aggiunto un nuovo elemento.

La progettazione delle viste è stata implementata nel package **ViewBy** che contiene le seguenti classi:

- **AbstractViewBy**: classe astratta che viene estesa dalle classi che implementano la vista personalizzata.
  - **MyListenerCourse** > **ViewByCourse**
  - **MyListenerDays** > **ViewByDay**:
  - **MyListenerHour** > **ViewByHour**
  - **MyListenerProf** > **ViewByProf**
  - **MyListenerRoom** > **ViewByRoom**
  - **MyListenerView** > **ViewGeneral**

La mia decisione è stata a implementare la classe astratta per poter poi personalizzare il metodo **fillCells**(*DefaultTableModel table, Object day*) che restituisce una tabella di tipo **DefaultTableModel** in base di ogni vista. Le viste implementando questo metodo riempiono le proprie tabelle con i dati esistenti nella tabella principale (**TableGUI**()). Una volta completati sono richiamati nel rispettivo **Listener**. Le classi delle **Listener** invece vengono istanziate nella classe **MyComboListeners**, tramite i loro getter chiamo ogni listener nella propria **comboBox** (inseriti nel **PanleButton**()).

L'implementazione di questo package contiene ripetizione del codice, il motivo per qui ho deciso di costruire cosi le classi era per facilitare la personalizzazione di ogni singola vista e cercare di non andare in conflitto con le altre classi, soprattutto con quella della **TableGUI**(). Ho anche eliminato l'opzione di poter riportare delle modifiche sulle viste specificate dal nostro relatore (prof. **M.Viroli**) per la insufficienza di tempo.

Finita la fase della viste sono tornata alla costruzione dei listener sui pulsanti nella classe **Buttons**():

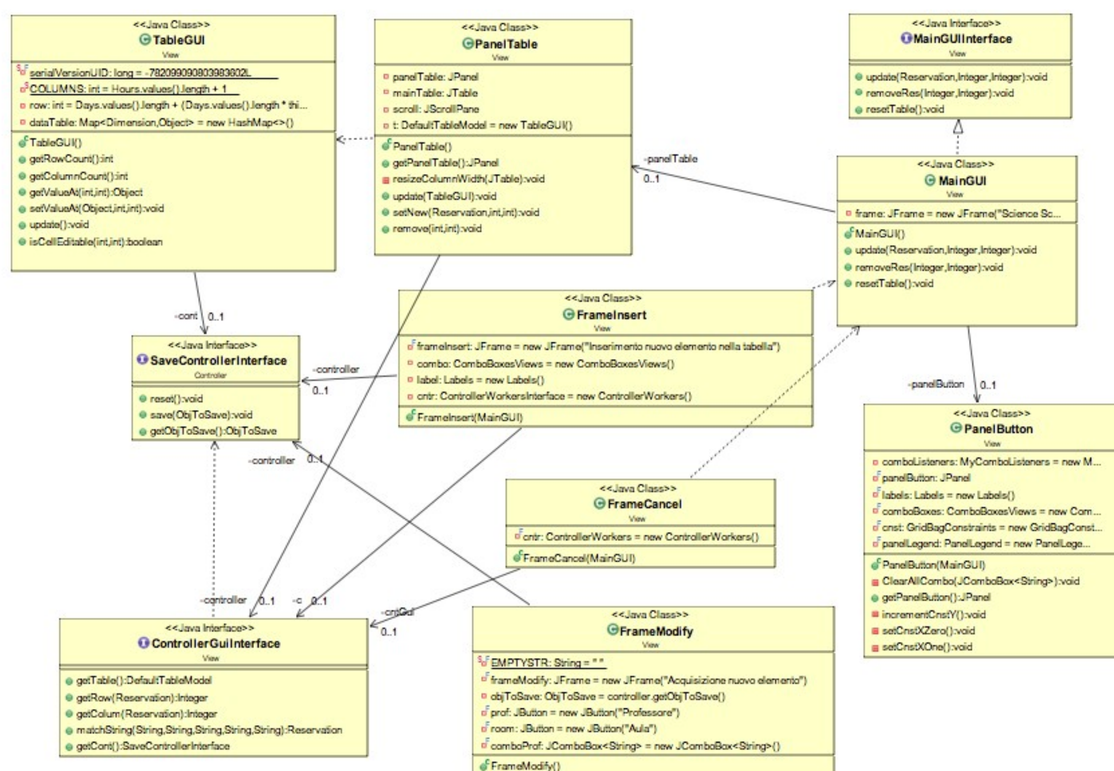
- **Esci**: Chiude il programma chiedendo (tramite un **JOptionPane**) all'utente se è sicuro di volerlo chiudere in questo momento.
- **Resetta**: Questo pulsante sostituisce l'opzione di dividere la vista per due semestri (è stata una decisione del gruppo). Esso in ogni caso dà possibilità di cominciare un nuovo file chiedendo (tramite un **JOptionPane**) all'utente prima di procedere alla eliminazione se il file è stato esportato in **Excel**. Il suo listener (collegato con il controller) cancella tutte le prenotazioni salvati sul file, e chiede (tramite un **JOptionPane**) all'utente se desidera di chiudere il programma per poter aggiornarlo. (non ho trovato il modo in qui fare **refresh** sulla tabella e ho inserito l'opzione che comunque mi permette di avvisare l'utente che l'applicazione bisogna essere riavviata per visualizzare la tabella aggiornata).
- **Cancella**: Il pulsante cancella e un'altra decisione di gruppo che ha sostituito l'implementazione dei pulsanti **UNDO**/**REDO** (che inizialmente erano progettati e implementati nella interfaccia grafica), permettendo comunque all'utente di scegliere l'informazione che desidera eliminare.

Il listener di questo pulsante richiama ogni volta una nuova istanza della classe **MyListenerCancel**(*mainGUI*) che implementa **ActionListener**.
Questo listener (con la condizione di avere una lista non vuota di prenotazioni) ad ogni avvio genera un frame contenente numero di pulsanti **JRadioButton** rispettivo alle prenotazioni salvati sul file, e per ogni pulsante crea il nome a base della stringa di ogni prenotazione esistente sul file. Il frame contiene il pulsante per la cancellazione dei dati salvati. Esso costruisce il suo listener in questo modo:

- Se l'utente non ha selezionato nessun dato, lo avvisa e non fà nulla (tramite un **JOptionPane**)
- Se l'utente ha selezionato qualsiasi dato chiede di confermare (tramite un **JOptionPane**).
  In caso di risposta positiva cancella l'informazione dal file e aggiorna la tabella (Qui abbiamo lavorato assieme a **Massimiliano Micca** per poter risolvere i problemi riscontrati durante la implementazione, esattamente sulla eliminazione delle prenotazioni dal file e l'aggiornamento della tabella).

Alla fine dell'ultima implementazione (sopra indicata) l'interfaccia grafica era conclusa. Ci siamo concentrati sulla stesura dei **warning** e **errori** che vengono visualizzati durante l'inserimento di un nuovo dato sulla tabella.

Figura 2.2.2 View



2.2.3 Controller

La mia parte consisteva nel fare i controlli per la **Gui**.
Mentre il Sig.re **Ceroni** decideva come strutturare i dati nel model, mi sono concentrato sul salvataggio dei dati in fomato Excel.
Questo compito non è stato facile, dopo varie ricerche sul web per capire come creare un file **xls**/**xlsx** (estensione dei file Excel) sono riuscito a capire il funzionamento delle librerie.
Almeno quanto basta per poter creare il file a mio piacimento.

La classe **ExportToXls** :

- Legge dal file la lista delle **Reservation** tramite **SaveController**;
- Con il metodo **makeMap** costruisco una mappa dove la chiave è il numero della riga e il valore è una lista con le stringhe corrispettive;
- Il metodo create() costruisce il foglio di Excel (per implementare questo metodo ho guardato molti esempi sul web);

- il metodo **save**(*String period*) unico metodo public, che richiamato dalla **Gui** al momento del esportazione, prende il campo in input per settare il nome del file "**period**.**xls**" e anche il nome del foglio in **Excel**.

Il passo successivo è stato pensare alla persistenza dei dati. Come prima battuta con il Sig.re **Ceroni**, abbiamo pensato di creare delle classi **Enum** per tutti i dati, successivamente ci siamo accorti che non potevamo fare la modifica dei dati. Quindi:

- Ho creato la classe **Reservation** per i dati della tabella, un oggetto che prende come campi del costruttore (**PersonImpl** , **CoursesImpl** , **Days** , **Hours**, **RoomImpl**).

- Per i dati modificabili nel tempo, inizialmente salvavo i vari oggetti in un unico file, ma al momento di ricaricarli avevo riscontrato dei problemi. Allora ho pensato di costruire classe **ObjToSave** per salvare un unico oggetto sul file, costituito da tre liste (oggetti stanze, oggetti professori con i relativi corsi e oggetti Reservation), al suo interno sono definiti i metodi per fare il **get** e il **set** dei vari campi. A questo punto salvo un unico oggetto in un **file.dat** usando la classe **SaveController**, anche in questo caso il metodo **save**(**ObjToSave**) è visibile esternamente insieme al metodo get per l'oggetto salvato. Nel caso non trovi già un file di salvataggio della app ne crea uno prendendo i dati dalle classi **Enum**.

Arrivati a questo punto, ho iniziato a fare il **ControllerWorkers** utile per:

- aggiungere dati;
- eliminare dati;
- salvare i dati sul file;
- fare controlli per l'inserimento;
- sono presenti anche i metodi per fare i **get** delle liste, in base al tipo di dato che si vuole ricercare, es. **getByDay**(Day d) torna una lista con tutte le **Reservation** con il campo **day** uguale a quello richiesto, ecc…

Successivamente ho lavorato a stretto contatto con le Sig.ne **Galya Genova** e **Anna Termopoli**, per costruire la classe **ControllerGui**, aiutarle a inserire i metodi del **Controller** e per provare a risolvere alcuni problema della visualizzazione della tabella.

Il progetto è arrivato a buon punto, però mancano da implementare i controlli per l'inserimento delle nuove **Reservation**.

Per risolvere la questione, il sig.re **Francesco Ceroni** ha creato le classi delle **Exception** nel model, io ho pensato a creare le classi:

- **ValidationWarning**: questa classe è stata implementata insieme alla Sig.na **Galya Genova**, serve per:
    - Controllare se i professori svolgono più di 6 ore al giorno di lezione e in tal caso lancia un errore che viene gestito adeguatamente;
    - Controllare se i professori svolgono più di 4 giorni a settimana in tal caso lancia un messaggio di errore come sopra;
    - Controllare se gli studenti volgono più di 4 giorni a settimana di lezione, anche in questo caso viene lanciato un messaggio di errore;
- **ValidationError**: questa classe è stata implementata per controllare se la cella in cui si vuole aggiungere la nuova prenotazione è già occupata, in tal caso manda un messaggio di errore;
- **ValidationCDL**: ultima, ma non per importanza, questa classe serve per far i controlli sui vari tipi di corsi inseriti:
    - Il primo controllo che svolge è controllare se un prof. È già occupata in quella fascia oraria, se si lancia un warning;
    - Successivamente passa a controllare ogni CDL.,
es. :
non si possono inserire due corsi obbligatori sovrapposti o sovrapporre un corso opzionale a un corso obbligatorio, invece un corso di scienze può essere sovrapposto a un corso di ingegneria. A seconda del caso lancia un warning o un error.
- **MyValidate:** serve per racchiudere le varie validate in un unico metodo richiamabile dal **ControllerWorkers** prima che si inserisca una **Reservation**.

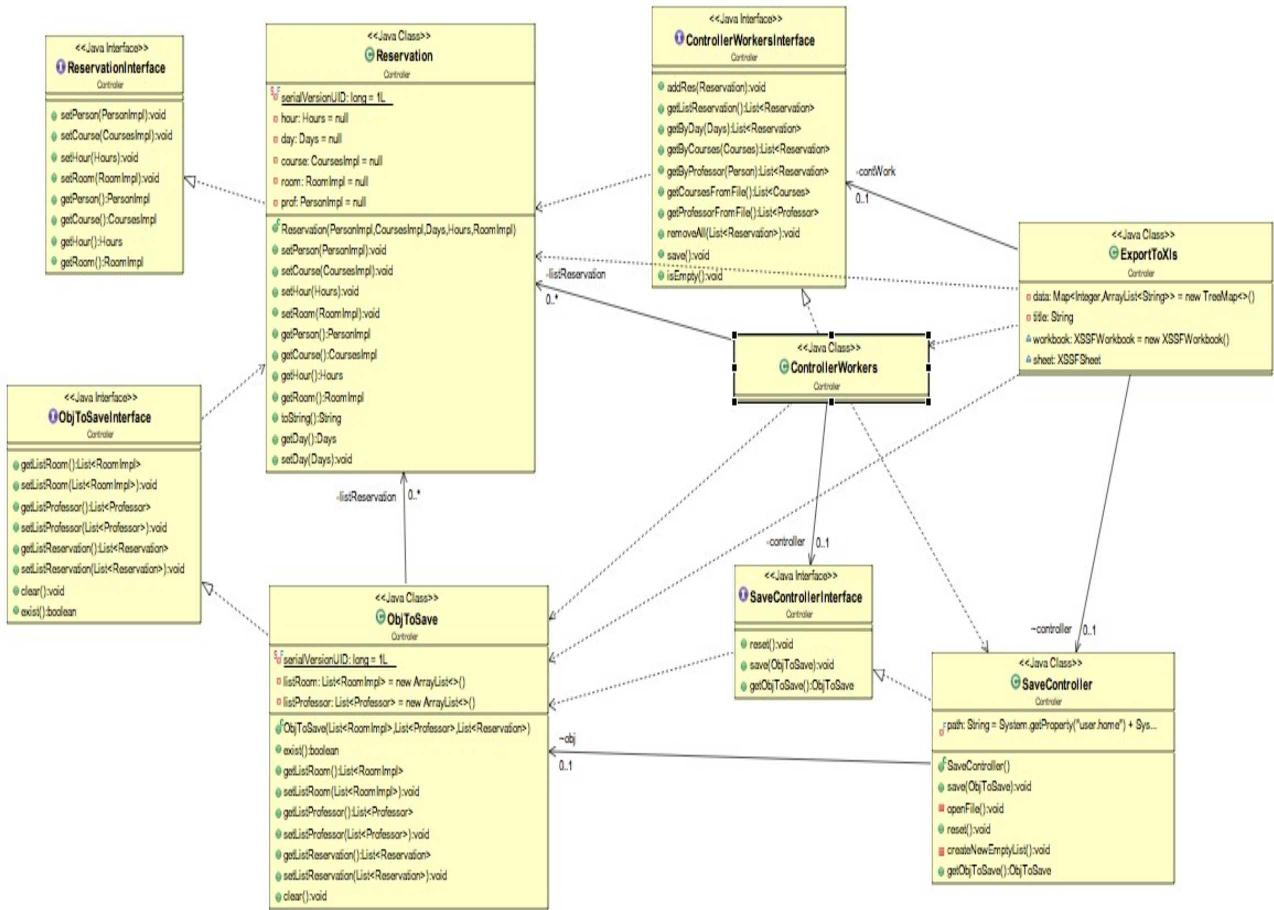


Figura 2.2.3 Controller

Capitolo 3

Sviluppo

3 Testing automatizzato

3.1 Metodologia di lavoro

Ci siamo trovati inizialmente per discutere del progetto e per definire le linee guida del lavoro di ognuno di noi, dopo di chè abbiamo svolto separatamente ognuno la propria parte.

La suddivisione del lavoro si è attenuta a quanto dichiarato precedentemente:

- **Model**: Francesco Ceroni
- **View**: Anna Termopoli e Galya Genova
- **Controller**: Massimiliano Micca

Spesso ci siamo ritrovati tutti e 4 per svolgere il progetto in gruppo, affinché potessimo collegare in modo efficiente le varie parti. Anche quando svolgevamo il progetto in autonomia rimanevamo in comunicazione soprattutto tramite BitBucket.

Nel primo periodo abbiamo iniziato a creare le classi in modo autonomo, ma mano a mano ci siamo necessariamente dovuti vedere di persona. Durante l'ultima fase ci siamo uniti spesso per cercare di collaborare insieme e risolvere i vari problemi che si erano verificati.
I problemi che abbiamo riscontrato sono nati quando abbiamo unito il codice e le varie parti dei collaboratori. In diverse occasioni la parte della View ha collaborato con il Controller (e viceversa) per chiedere spiegazioni sul codice implementato. In fase di sviluppo si è utilizzato molto **BitBucket** con **Mercurial** come DVCS col fine di memorizzare e rendere disponibile ogni cambiamento e inserimento su codice.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

Abbiamo cercato di implementare al meglio tutti i requisiti richiesti dal committente con l'idea di creare un'applicazione semplice e facilmente usabile dall'utente.
L'interfaccia grafica, infatti, ci sembra abbastanza semplice da capire anzi, abbiamo aggiunto anche numerosi pannelli di avviso o di errore, per accompagnare l'utente nelle sue scelte.

Il gruppo si era infatti proposto (se ci fosse stato abbastanza tempo) di implementare cose aggiuntive.
Le nostre idee più interessanti sono le seguenti:

- Avremmo voluto implementare due metodi **"UNDO & REDO"** ma abbiamo deciso di implementare il metodo **"CANCEL"** nella classe **FrameCancel** ,che dà la possibilità attraverso dei **JRadioButton** di scegliere la prenotazione da cancellare.
  Questa decisione nasce per facilitare l'utente nella scelta della prenotazione da cancellare, in quanto con il metodo **CANCEL** il fruitore può scegliere direttamente cosa eliminare. Il metodo "**UNDO**" avrebbe dovuto eliminare l'ultima prenotazione inserita mentre il metodo "**REDO**" avrebbe dovuto riaggiungere l'ultima;
- Avremmo voluto implementare la scelta del semestre.
  Infatti la nostra applicazione non dà la possibilità di scegliere il semestre ma l'utente può salvare la tabella in un file Excel e resettare le prenotazioni presenti sull'applicazione e sul file.

Anna Termopoli:

Sono quasi soddisfatta del mio lavoro, avrei avuto bisogno di più tempo per implementare al meglio quello che io e la Sig.na **Genova** avevamo pensato.
Complessivamente penso di aver creato un interfaccia grafica abbastanza facile da capire e da usare. Sicuramente se avessi avuto più ore avrei potuto creare un'interfaccia ancora più funzionale, con più dettagli e più curata.
Per quanto riguarda il lavoro di gruppo sono soddisfatta, è stata una bella esperienza anche se molto impegnativa soprattutto per me che sono pendolare, ma lavorando insieme ho potuto imparare tante cose.
Ci siamo supportati e aiutarti in varie occasioni.

Galya Genova:

Il mio lavoro consisteva nella creazione della interfaccia grafica dell'applicazione. Una parte del lavoro è stato svolto insieme alla Sig.na **Termopoli**.
All'inizio abbiamo contribuito tutti quanti alla progettazione della struttura del progetto, una volta finita ognuno ha iniziato a implementare la sua parte del codice,
Io e la Sig.na **Termopoli** abbiamo collaborato insieme per organizzare le classi principali che formano la **GUI** .
In seguito per i lavori finali abbiamo collaborato con Sig.re **Micca**.
Avendo a disposizione tempo limitato, sono comunque contenta del lavoro che ho svolto. Avrei potuto implementare meglio la struttura della **GUI**, usando il pattern **Observer**, ma purtroppo l'idea mi è venuta verso la fine dell'implementazione dell'elaborato. Inoltre potevo evitare certi ripetizioni del codice, ma non avendo tempo a sufficienza, ho deciso di lasciare cosi la struttura della mia parte di codice che comunque è funzionante, suddivisa e organizzata in maniera sufficientemente bene.
Sono contenta di aver lavorato in un gruppo di 4 persone. Lavorare in tanti non è semplice, ma questa esperienza mi ha insegnato a collaborare e vedere i lati diversi nella programmazione in un gruppo, anziché da soli.

Massimiliano Micca:

Avrei voluto implementare in maniera più pulita le mie classi, lasciando interagire la **Gui** solo con una classe del controller (**ControllerWorkers**) e tenere le altre classi, ove possibile, interne al package.
Ovviamente, se il tempo era maggiore, avrei voluto fare un Controller più funzionale, cioè :

- Meno scritture e letture sul file;
- Riuscire a fare meno cicli possibili.

Secondo il mio parere per fare un lavoro simile in maniera ottimale bisognerebbe dedicargli più tempo, ma andrebbe fuori dai crediti assegnati al corso.

Francesco Ceroni:

Sono abbastanza soddisfatto del mio lavoro, se c'era più tempo si potevano anche implementare altre funzioni del programma.
Finita la mia parte ho cercato di aiutare il più possibile il gruppo.
Per quanto riguarda il lavoro di gruppo anche io sono molto soddisfatto, l'esperienza è stata bellissima ma anche molto complicata ad incastrare i giorni liberi per trovarsi tutti insieme essendo anche io pendolare, come ha detto anche la mia collega sopra ci siamo sempre supportati e aiutati uno con l'altro.

4.2 Difficoltà incontrate e commenti per i docenti

Anna Termopoli: avendo il Mac ho avuto molta difficoltà nel installare Mercurial infatti per il primo periodo, mano a mano che scrivevo il codice, lo passavo alla Sig.na **Genova** che caricava le mie classi su **BitBucket**.
Successivamente ho chiesto aiuto ad un ragazzo del mio anno che ha saputo risolvermi il problema.

Galya Genova: Non avendo visto certi argomenti a lezione, ho riscontrato delle difficoltà nella progettazione della **GUI**, più precisamente nello studio delle tabelle e nella loro implementazione in generale.
Una volta provate e capite le funzionalità è stato facile usare classi e librerie nuove.
Ho riscontrato spesso dei problemi con il caricamento e lo scaricamento del codice dal server, è stato un problema comune di tutto il gruppo, infatti ci ha causato molta perdita di tempo che poteva essere utilizzata per la programmazione dell'applicazione e per il miglioramento del codice.
In conclusione ho imparato molto da questo corso grazie alle difficoltà che ho trovato e agli sbagli che ho fatto, perché sbagliando si impara.

Massimiliano Micca: come tutte i membri del gruppo il problema maggiore è stato utilizzo di **Mercurial**, perché al minimo cambiamento delle classi dei colleghi segnalava degli errori con, conseguente perdita di tempo.
Un'altra difficoltà è stato capire il funzionamento delle librerie esterne per creare il foglio Excel.

Appendice A

Guida utente

Lo scopo dell'applicazione è quello di dare la possibilità all'utente di organizzare un orario universitario.

L'utente con il bottone "inserisci" ha la possibilità di inserire una prenotazione (docente, materia ,orario ,aula ,giorno) nella tabella. Se vuole vedere il calendario principale con tutte le prenotazioni potrà semplicemente cliccare su bottone "vista generale".

L'utente potrebbe visualizzare l'orario in diversi modi (giorno, aula, orario, docente, corso). Cliccando sulla vista scelta si aprirà una nuova finestra in cui sarà visualizzata la tabella con tutte le prenotazioni secondo la vista desiderata.

L'utente ha la possibilità attraverso il bottone "modifica nome prof/ aggiungi aula" di modificare il nome del titolare di un corso oppure di aggiungere una nuova aula.

Cliccando su pulsante "esporta in excel", la tabella verrà esportata su un file Excel e alla fine del processo, verrà anche indicato il percorso di salvataggio.

Con la funzionalità "resetta la tabella" , l'utente cancellerà tutte le prenotazioni e riavviando il programma, potrà aggiungerne delle nuove.

Il pulsante "cancella" fà scegliere all'utente che prenotazione vuole cancellare (ne può selezionare una o più di una oppure tutte quante).

Infine per uscire dall'applicazione basterà cliccare sul pulsante "esci" oppure cliccando nella classica "X" rossa in alto.