OSSERVAZIONI

Giovanni Tentelli

Riccardo Fonti

24 ottobre 2022

PREFAZIONE

Observations vuole essere uno strumento a disposizione di docenti, pedagogisti ed educatori che operano nella scuola italiana e che ogni giorno si trovano a dover fronteggiare la pluralità di linguaggi che i loro alunni riescono a mettere in atto.

Questo strumento può risultare utile per monitorare, attraverso l'osservazione, i molteplici messaggi messi in atto e che meritano di essere accolti e compresi, all'unico scopo di riuscire a stabilire relazioni significative e progettare gli interventi più idonei per ogni singolo alunno.

Da sempre l'osservazione è lo strumento più idoneo se si vuole conoscere e comprendere anche il più semplice dei fenomeni, registrarne la frequenza e capire il contesto nel quale si manifesta.

Anche nella scuola osservare da un punto di vista oggettivo è una pratica da preservare e incentivare perché fornisce punti di vista significativi che, se letti con occhio attento, possono aprire orizzonti nuovi nei quali riprogettare le attività didattiche, rimodulare gli interventi e affinare tecniche sempre più idonee e personalizzate per garantire una formazione ampia e adatta al contesto nel quale ogni alunno è inserito.

Questo strumento vuole rappresentare un prototipo di quanto possa essere realizzato nell'applicazione finale e di come la tecnologia e la tecnica possano ancora essere di supporto in ambito formativo ed educativo.

Indice

Capitolo 1

Analisi

1.1 Requisiti

Il software che si vuole realizzare è chiamato “Observations” e vuole essere uno strumento di aiuto per un utente che deve gestire le osservazioni relative ad uno studente e renderle di facile utilizzo. L’utente (osservatore) sceglie uno studente da osservare in un momento della giornata scolastica (es: prima ora, matematica, italiano, ecc.) e decide quali tipologie di comportamento intende osservare. Lo scopo è tenere un archivio di ogni giorno, di ogni determinato studente, di ogni preciso momento in cui è stata effettuata una osservazione e quali e quanti tipi di comportamenti sono stati rilevati. Questi dati vengono poi resi intuitivi per l’utente attraverso grafici all’utente.

1.1.1 Requisiti funzionali

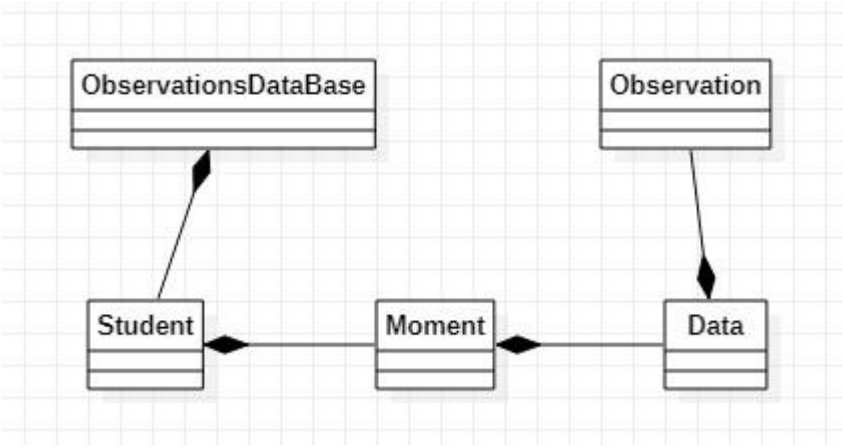
- L’applicazione dovrà gestire tutto tramite java anziché appoggiarsi su un database.
- Verrà registrato il nome dello studente, il momento della giornata, il giorno, il tipo di comportamento e l’orario nel quale si è verificato il dato comportamento.
- L’utente tramite interfaccia grafica potrà scegliere in ordine: uno studente o crearne uno nuovo, dopodiché dovrà scegliere un momento (o crearne uno nuovo), infine la data e gli atteggiamenti da osservare.
- Il programma creerà 2 file contenenti i momenti della giornata e le tipologie di comportamento: questi 2 file avranno dati precaricati standard e verranno aggiornati nel momento in cui l’utente esprimerà una preferenza non presente in elenco.

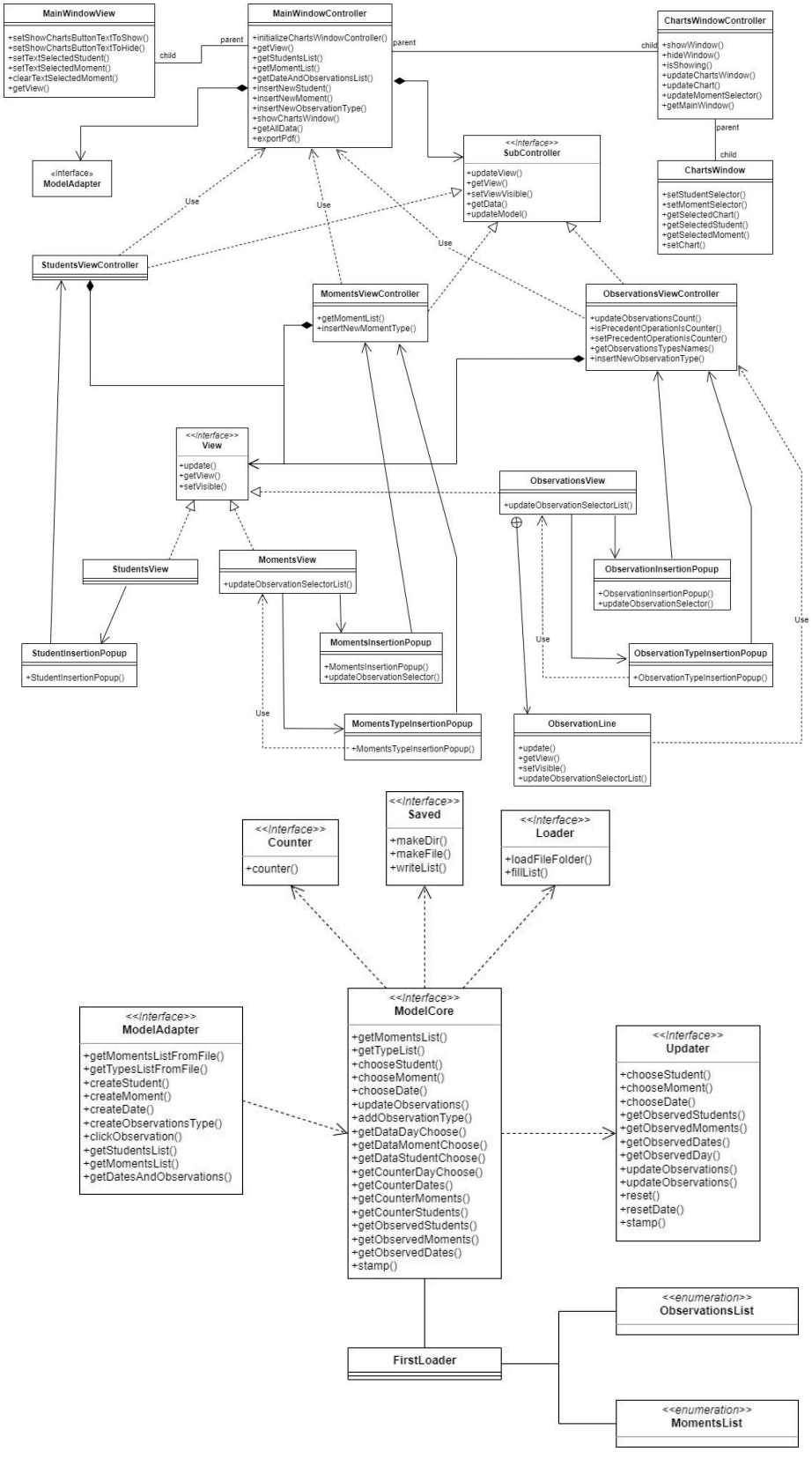
1.1.2 Requisiti non funzionali

- L’utente potrebbe non essere del tutto preparato all’uso di software.
- Meno cose gli si lascia fare, meglio è, bisogna gestire tutto in maniera semplice e intuitiva.

1.2 Analisi e modello del dominio

Observations fornisce all’utente uno schema visivo di ciò che intende osservare. L’utente quindi sceglie uno studente, un momento nel quale lo vuole osservare e setta la data. Poi sceglie gli atteggiamenti che l’alunno man mano mette in atto: ad ogni click sugli atteggiamenti scelti il software mostrerà il numero di click di ciascuno di essi in tempo reale. L’utente potrà vedere anche i dati di ogni singola giornata rappresentati come atteggiamenti e orario in cui ciascuno di essi è stato rilevato; ciò consentirà di analizzarne la frequenza. I grafici che potrà visualizzare lo aiuteranno nel compito valutativo.




MainWindowView
+setShowChartsButtonTextToShow()
+setShowChartsButtonTextToHide()
+setTextSelectedStudent()
+setTextSelectedMoment()
+clearTextSelectedMoment()
+getView()
child
parent
MainWindowController
+initializeChartsWindowController()
+getView()
+getStudentsList()
+getMomentList()
+getDateAndObservationsList()
+insertNewStudent()
+insertNewMoment()
+insertNewObservationType()
+showChartsWindow()
+getAllData()
+exportPdf()
parent
child
ChartsWindowController
+showWindow()
+hideWindow()
+isShowing()
+updateChartsWindow()
+updateChart()
+updateMomentSelector()
+getMainWindow()
parent
child
ChartsWindow
+setStudentSelector()
+setMomentSelector()
+getSelectedChart()
+getSelectedStudent()
+getSelectedMoment()
+setChart()
«interface»
ModelAdapter
<<Interface>>
SubController
+updateView()
+getView()
+setViewVisible()
+getData()
+updateModel()
Use
Use
Use
StudentsViewController
MomentsViewController
+getMomentList()
+insertNewMomentType()
ObservationsViewController
+updateObservationsCount()
+isPrecedentOperationIsCounter()
+setPrecedentOperationIsCounter()
+getObservationsTypesNames()
+insertNewObservationType()
<<Interface>>
View
+update()
+getView()
+setVisible()
ObservationsView
+updateObservationSelectorList()
StudentsView
MomentsView
+updateObservationSelectorList()
ObservationInsertionPopup
+ObservationInsertionPopup()
+updateObservationSelector()
Use
StudentInsertionPopup
+StudentInsertionPopup()
MomentsInsertionPopup
+MomentsInsertionPopup()
+updateObservationSelector()
Use
ObservationTypeInsertionPopup
+ObservationTypeInsertionPopup()
Use
Use
MomentsTypeInsertionPopup
+MomentsTypeInsertionPopup()
ObservationLine
+update()
+getView()
+setVisible()
+updateObservationSelectorList()
<<Interface>>
Counter
+counter()
<<Interface>>
Saved
+makeDir()
+makeFile()
+writeList()
<<Interface>>
Loader
+loadFileFolder()
+fillList()
<<Interface>>
ModelAdapter
+getMomentsListFromFile()
+getTypesListFromFile()
+createStudent()
+createMoment()
+createDate()
+createObservationsType()
+clickObservation()
+getStudentsList()
+getMomentsList()
+getDatesAndObservations()
<<Interface>>
ModelCore
+getMomentsList()
+getTypeList()
+chooseStudent()
+chooseMoment()
+chooseDate()
+updateObservations()
+addObservationType()
+getDataDayChoose()
+getDataMomentChoose()
+getDataStudentChoose()
+getCounterDayChoose()
+getCounterDates()
+getCounterMoments()
+getCounterStudents()
+getObservedStudents()
+getObservedMoments()
+getObservedDates()
+stamp()
<<Interface>>
Updater
+chooseStudent()
+chooseMoment()
+chooseDate()
+getObservedStudents()
+getObservedMoments()
+getObservedDates()
+getObservedDay()
+updateObservations()
+updateObservations()
+reset()
+resetDate()
+stamp()
<<enumeration>>
ObservationsList
FirstLoader
<<enumeration>>
MomentsList

Capitolo 2

Design

2.1 Architettura

Per realizzare Observations ci siamo affidati al pattern architetturale MVC con alcune varianti per coordinarci al meglio e lavorare in maniera asincrona fino al merge finale.

La view si è occupata di creare l'interfaccia grafica scorporando tutti gli aspetti implementativi dei vari controlli su chi dovesse fare cosa. In particolare, Observation lancia il main view controller che è a capo dei vari controllori di stato come studente, momento e osservazioni fatte nelle date; ogni controller ha quindi la sua view interna che gestisce la propria parte e mostra un popup per l'inserimento o la scelta di un nuovo item.

I grafici sono implementati nella view tramite 2 classi chartfactory e chartdatafiler che servono a prendere i dati e trasformarli per la visualizzazione su grafico a torta.

Il model in particolare ha creato dei test incrementali durante tutta la stesura del codice allo scopo di verificare la correttezza di ciò che doveva fare l'applicazione. Sempre il model ha creato un adapter in grado di convertire i dati in ingresso e in uscita della view al model e viceversa.

La view ha tanti controller quanti gliene servono per monitorare tutte le varie parti di ciò che viene mostrato e richiamare i comandi dalla classe adapter.

2.2 Design dettagliato

Gestione dei dati

2.2.1 Giovanni Tentelli

Premessa: L'applicazione Observations lancia prima di tutto il main windows controller, il cuore della view che gestisce il comportamento di ogni singola classe ad essa collegata.

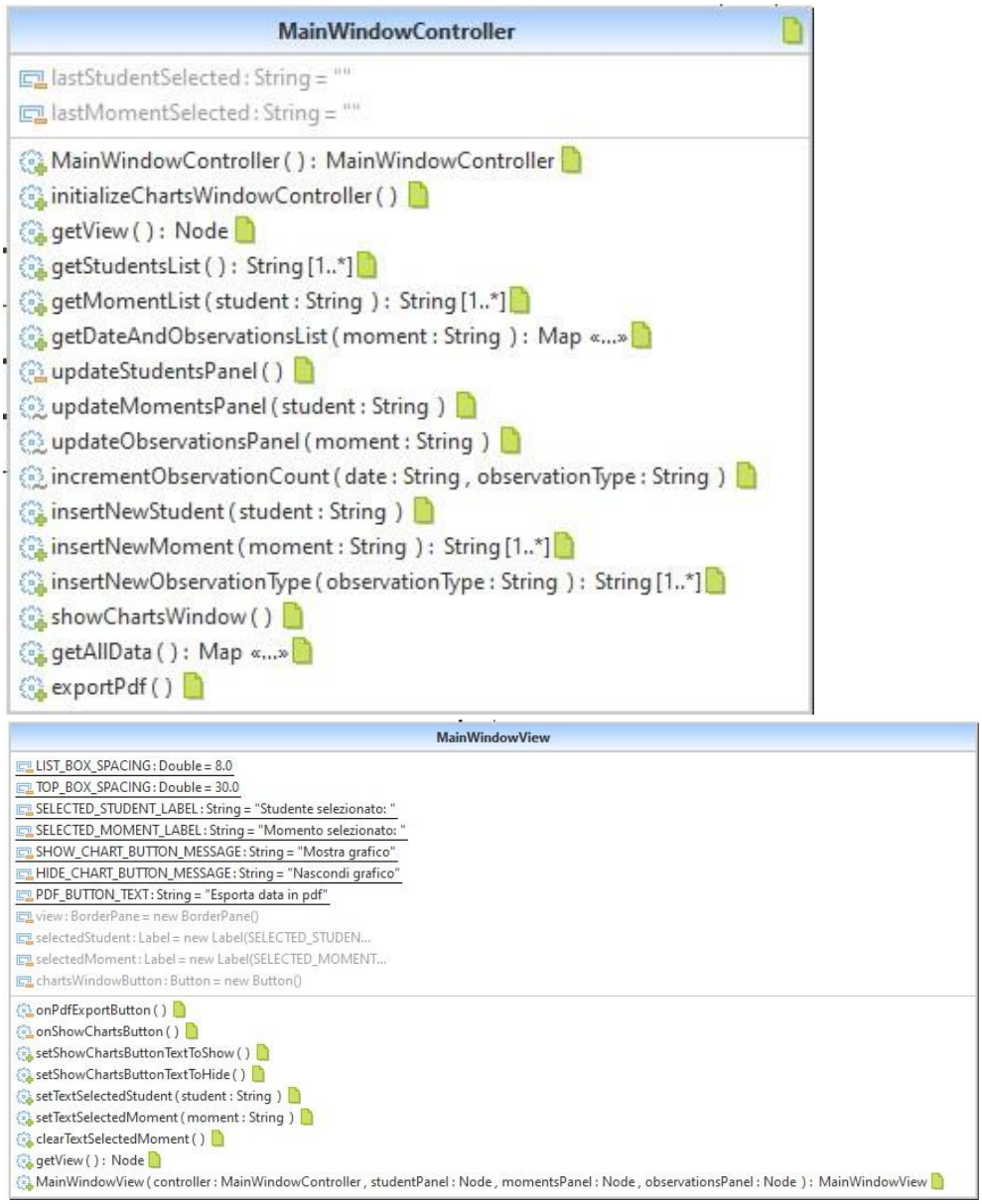


Figura 2.2.1.1: UML di MainWindow

Problema: Come gestire i vari stati dell'applicazione come studente, momento e osservazioni fatte?

Soluzione: Creare un controller per ogni sezione o componente dell'interfaccia utente che non necessita di interagire direttamente con altre sezioni o componenti; lasciando quindi gestire le interazioni esterne alla parte che controllano al MainWindowController, questo per evitare la situazione di avere un unico massiccio controller e migliorare la leggibilità del codice uniformando al meglio le classi. Stesso ragionamento si applica all'integrazione di popups come finestra distaccata al fine di evitare di appesantire l'interfaccia utente.

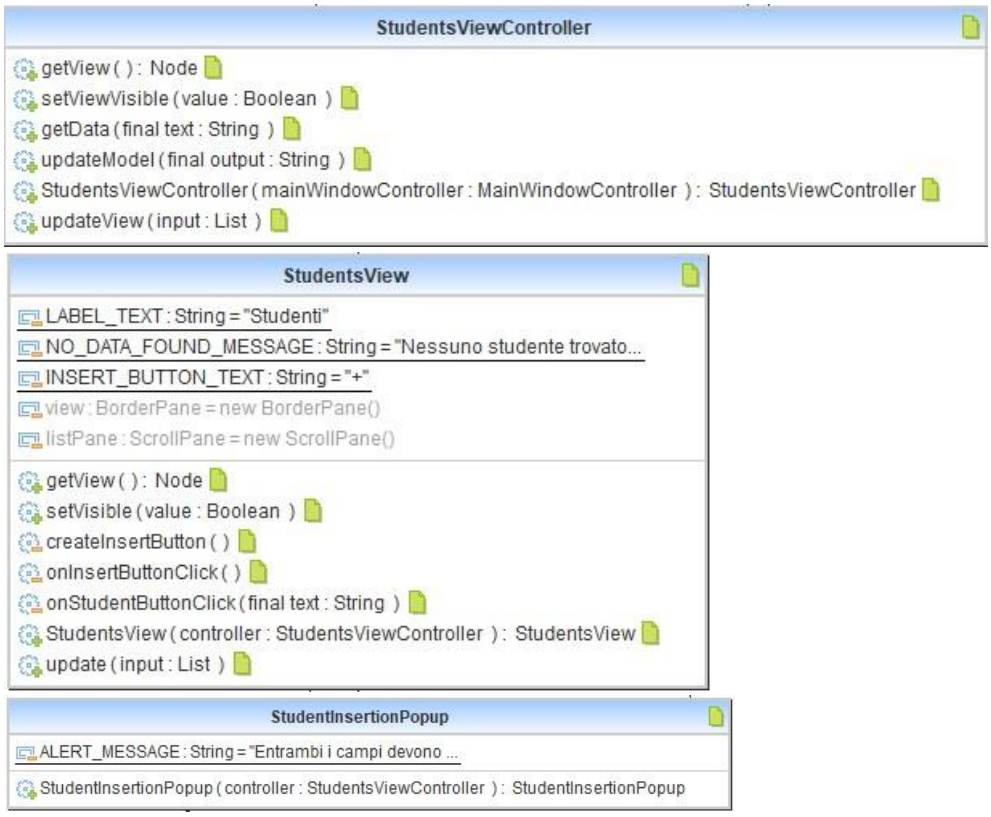


Figura 2.2.1.2: UML dei Controller e dei popup (moment e observation simili)

Problema: Come gestire l'incremento e l'inserimento di un nuovo comportamento e la relativa data?

Soluzione: Tramite il controller che si occupa di gestire le date e i comportamenti si inseriscono, tramite popup, questi nuovi item; successivamente la view di questo controller si occupa di visualizzare data, tipo di comportamento e numero di volte che questo viene cliccato.

Problema: Come gestire l'esportazione in pdf?

Soluzione: Tramite una libreria esterna apache.pdfbox ho gestito l'esportazione dei dati e salvataggio dei dati.

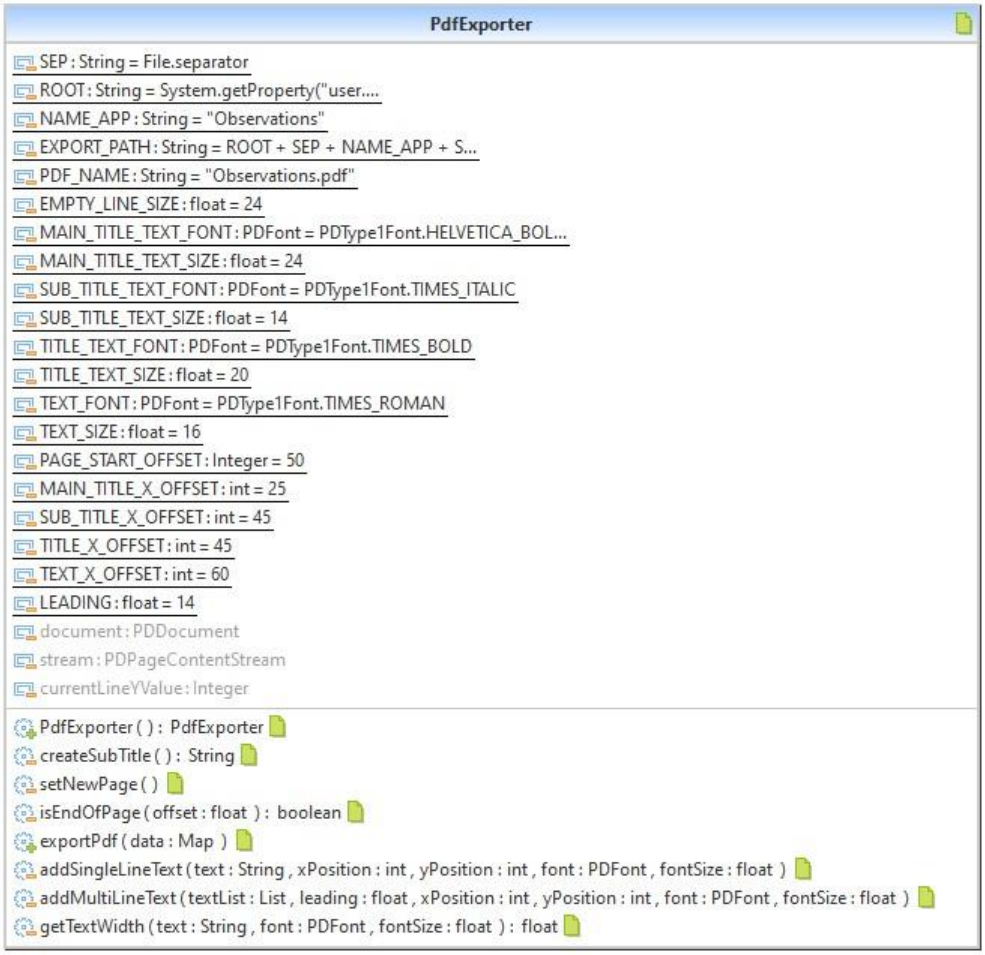


Figura 2.2.1.3: UML della classe che si occupa dell'esportazione in pdf.

Problema: Come gestire i grafici?

Soluzione: Ho implementato due classi che si occupano di gestire i grafici: chartfactory che è una classe statica che crea grafici dai dati passatogli dai poi passare alla view. Un'altra classe è chartdatafiler, la cui occupazione è quella di smistare i dati tramite filtri e riorganizzarli in un formato compatibile per l'inizializzazione di un grafico. Dal lato interfaccia utente.

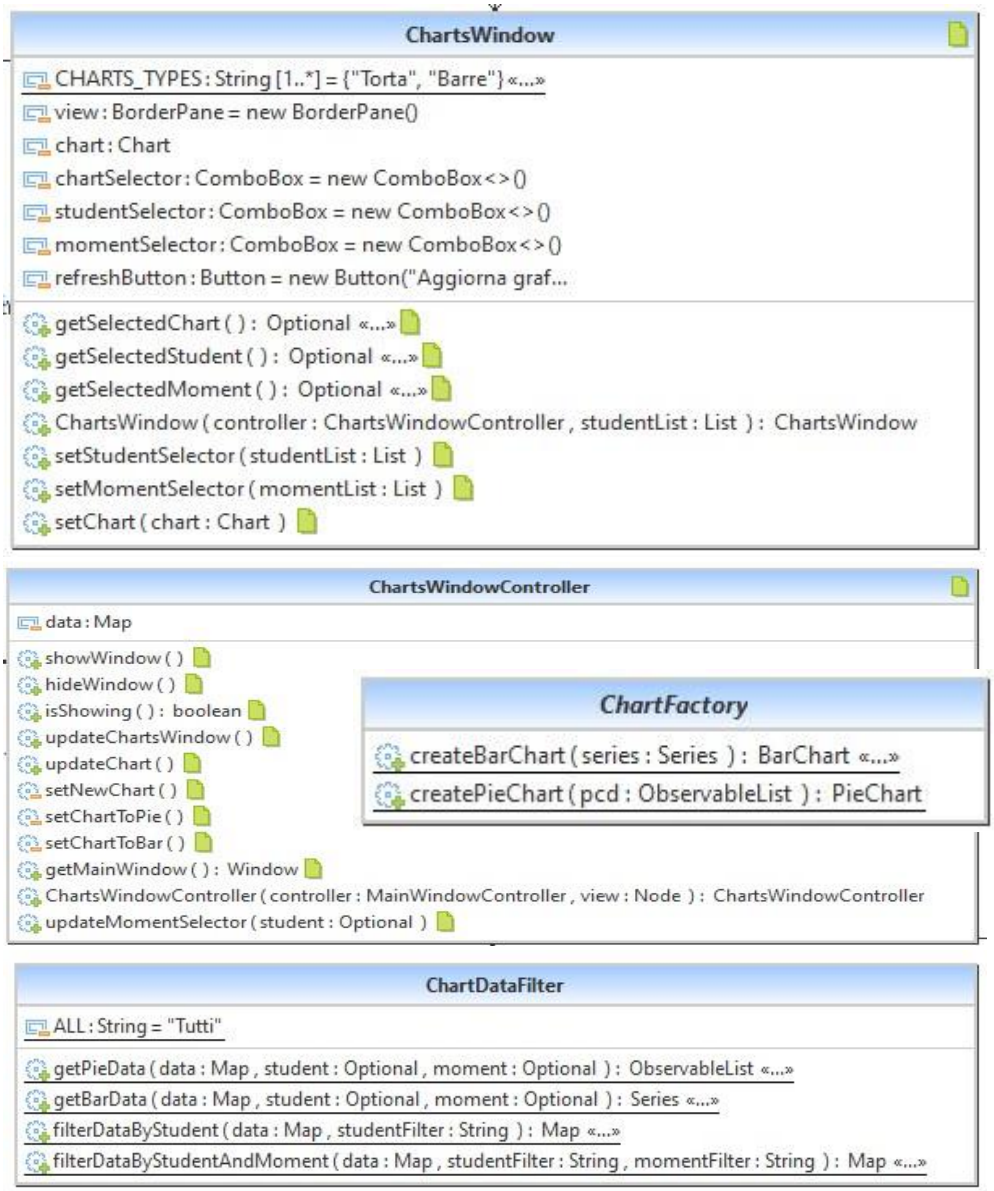


Figura 2.2.1.4: UML delle 4 classi chart

2.2.2 Riccardo Fonti

Premessa: Dai primi confronti tra colleghi era emerso il problema di come salvare lo stato di "selezionato" di studente, momento e data. Abbiamo deciso di delegare al model il salvataggio degli stati e ciò ha richiesto diversi sforzi. Sono state inserite variabili temporanee aggiuntive per garantire il mantenimento di tutti i dati temporali scelti in precedenza.

Problema: Come gestire i dati senza appoggiarsi ad un database per utilizzare java? In particolare, come gestire le 2 liste di dati per momenti e atteggiamenti aggiornabili e i files con i comportamenti e gli orari distinti per data, momento e studente?

Soluzione: Ho scelto di operare a gerarchie di cartelle partendo da una root folder chiamata students: al suo interno risiedono tutte le cartelle create, una per ogni studente. Dentro ogni cartella studente ci saranno tante cartelle quanti sono i momenti dedicati alle osservazioni; infine dentro le cartelle dei momenti della giornata ci saranno files che avranno come nome la data in cui vengono svolte le osservazioni e questi files a loro volta conterranno il tipo di osservazione fatta e l'orario (HH:mm:ss) in cui è stata effettuata. Le due liste verranno precaricate con dati forniti dal cliente e risiederanno nella cartella padre di students, per rendere più agevole la fruizione. La radice dei salvataggi sarà la cartella user dell'utente dove verrà creata una cartella Observations con al suo interno una cartella save.

Problema: Come risolvere la creazione di file e cartelle, l'aggiornamento dei vari files creati e il caricamento dei dati?

Soluzione: Ho usato un approccio bottom up, sono quindi partito dal basso ed ho creato una prima classe Save per creare file e cartelle in path predefinito. Ho creato successivamente una classe Loader che leggesse i contenuti delle cartelle e dei files e mi restituisse una lista con i nomi delle cartelle stesse partendo da un path che le viene fornito. Quindi ho creato la prima vera classe che facesse l'avvio dell'applicazione: FirstLoader. Questa classe ha avuto diverse evoluzioni, in primis creava tutte le cartelle che sarebbero servite nella user folder, nella fase finale crea anche le due liste di dati precaricati che prende dalle 2 classi enum; in più ora ha un controllo per gli avvii successivi al primo, se le cartelle e i file sono già presenti non fa nulla.

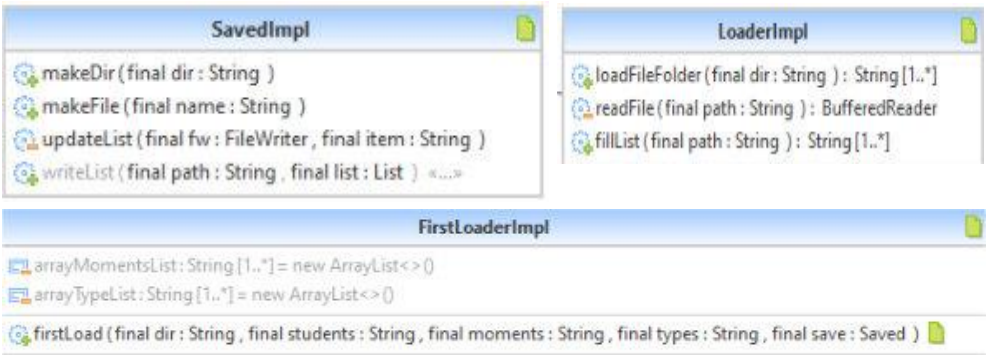


Figura 2.2.2.1: UML delle 3 classi e ciò che fanno

Problema: come aggiornare i dati sia delle liste che delle osservazioni del giorno?

Soluzione: ho creato una classe Updater che si occupa di gestire la selezione e creazione dello studente, poi del momento e infine della data. Se le stringhe fornite non sono presenti crea cartelle o file specifici; se presenti tiene in memoria tali dati e tramite le classi per il caricamento e il salvataggio dei dati aggiorna tutti i files e cartelle.

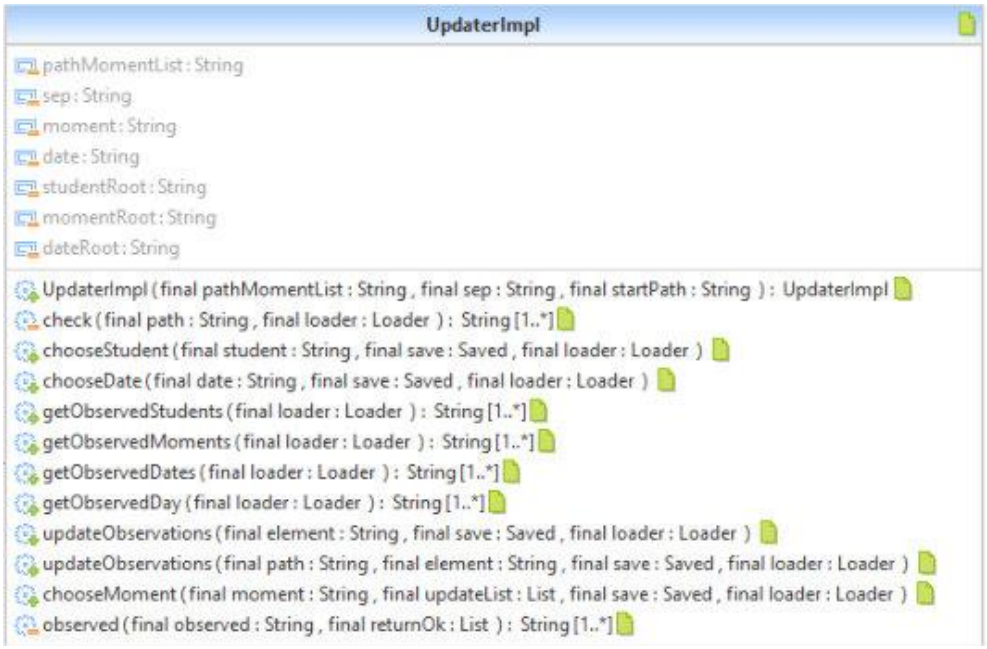


Figura 2.2.2.2: UML di updater

Problema: Come rendere intuitivo il tutto per le classi esterne al model?

Soluzione: Ho creato il ModelCore che racchiude diversi metodi per l'interazione con tutte le sotto parti, lo definisco il cervello, colui che si occupa di distribuire il lavoro. Ci sono metodi simili a quelli presenti in altre classi: volevo che l'interfaccia finale fosse stata ben divisa (anche se ridondante) e intuitiva. Ho eliminato le dipendenze verso le altre classi, ora solo questa classe dipende dalle altre creando riferimenti interni che poi usa per sé o passa a chi li necessita per funzionare.

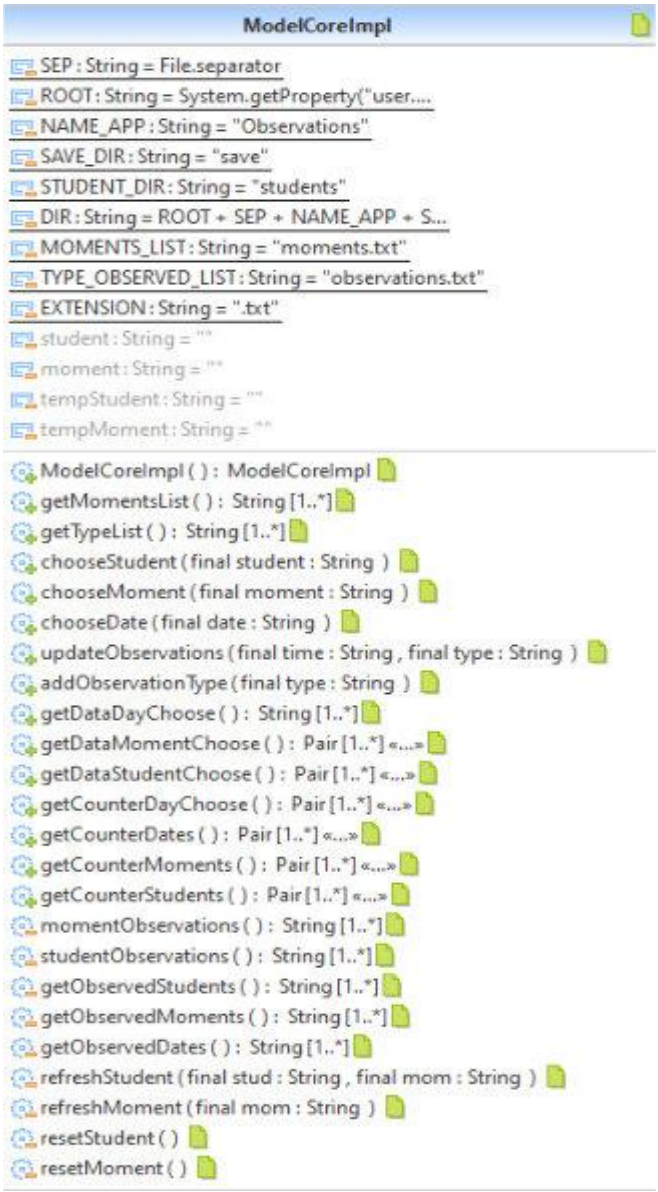


Figura 2.2.2.3: UML di ModelCore: è una classe ponte, si crea al suo interno i nuovi oggetti delle classi che serviranno a tutto il sistema e richiede dati passando i riferimenti alle classi necessarie.
Gli unici metodi pubblici che fanno effettivamente qualcosa sono i getCounterStudents/Moments/Dates, questi hanno un controllo interno che salva la posizione corrente della selezione se presente, servono esclusivamente per i grafici o per tenere il conteggio.
Tutti gli altri metodi sono conversione di dati e passaggio di riferimenti per evitare dipendenze.

Problema: Come fare in modo che controller e model si possano confrontare? Il collega ha progettato il tutto basandosi sulle mappe e liste di stringhe e interi, io tutto su stringhe, interi e pair (classe presa dalle esercitazioni a lezione).

Soluzione: La classe adapter converte i dati di ritorno del model nelle mappe richieste dal controller. Adapter richiede semplici stringhe con i dati per studente, momento, date, osservazioni, poi basta che il controller utilizzi il metodo corretto e passi la stringa giusta, il model crea ciò che gli viene passato solo ed esclusivamente in ordine e con il metodo corretto. Se il controller manda una stringa date, utilizzando il comando per creare lo studente, crea una cartella studente che acquisisce come nome la data mandata.

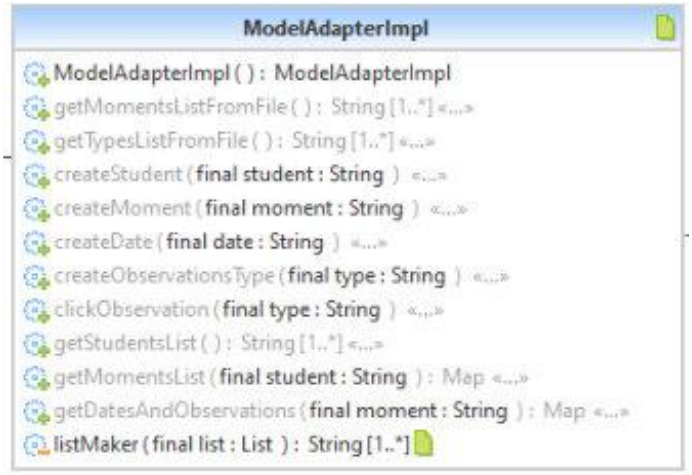


Figura 2.2.2.4: UML di Adapter. Questa è la classe che mi è servita a trasformare tutti i dati ancora per me sconosciuti che la view/controller passa al model, e convertire i dati che il model fornisce in ciò che la view/controller si aspetta.

Capitolo 3

Sviluppo

3.1 Testing automatizzato

Per la parte di model è stata utilizzata la libreria JUnit per creare una classe di Test che comprende 7 test al suo interno per verificare il corretto funzionamento delle parti create (non presenti nella release finale*):

- Test per Save per la creazione di cartelle
- Test per Loader per la lettura delle cartelle dato un path
- Test per Save per la creazione di files in un path
- Test della classe Updater per l'aggiornamento dei files
- Test per la lettura di file dato un path
- Test per ModelCore: creazione e selezione di studente, momento e data, aggiornamento della data e delle liste, cambio di data e cambio momento.
- Test per ModelAdapter

Tutti i test hanno inoltre delle println per mostrare i contenuti e verificarli visivamente.

3.2 Metodologia di lavoro

3.2.1 Giovanni Tentelli

- Observation and AppLauncher main classes
- controller classes (org.observation.controllers.*)
- gui classes (org.observation.gui.*)
- chartFactory class (org.observation.chartFactory)
- utility classes (org.observations.utility.*)

3.2.2 Riccardo Fonti

- classes model (org.observations.model.*)
- interfaces smodel (org.observations.model)
- class adapter (org.observations.model.core)
- interfaces adapter (org.observations.model)

3.3 Note di sviluppo

3.3.1 Giovanni Tentelli

- Per via della classe principale Observation, che contiene il metodo main e lancia l'applicazione, estende la classe di javafx: Application; mi ha reso difficile l'esecuzione dell'applicazione tramite .jar. Ho quindi risolto aggirando il problema creando una classe con metodo main a parte che lancia Observation e usando il plugin gradle shadow per produrre il jar.
- Ho fatto uso di espressioni lamba qualora fosse possibile e non ostacola la leggibilità del programma.

3.3.2 Riccardo Fonti

- Ho copiato la classe Pair per avere a disposizione le coppie di dati per il conteggio delle tipologie di osservazione fatte o il quantitativo di momenti o date osservate.
- Ho usato lambda per la sort di ordinamento delle liste salvate e dei dati che vengono passati alle classi richiedenti.

Capitolo 4

Commenti finali

4.1 Autovalutazione e lavori futuri

4.1.1 Giovanni Tentelli

Questo progetto ha messo a dura prova la mia capacità come neofita programmatore e incrementare le mie capacità di analisi, programmazione, organizzazione e debugging. Durante il progetto ho avuto problemi a confrontarmi con Git e le sue necessità, a studiare e comprendere Gradle, specialmente risolvere la gestione delle dipendenze con build.gradle, e organizzarmi il mio posto di lavoro.
Nonostante programmare non sia il mio lato forte ho dato il mio massimo per consegnare un'applicazione funzionante, dinamica e comprensibile.
Se c'è una cosa che mi ha storto il naso è come avrei potuto fare una migliore organizzazione delle interfacce e delle classi per ottenere un codice più pulito.
Concludendo mi sento sodisfatto di come è stato chiuso il progetto, come è stata conclusa la mia parte di progetto e della raccolta di tutta l'esperienza ricevuta a lavorare con un linguaggio come java.

4.1.2 Riccardo Fonti

Sono soddisfatto del lavoro svolto e consapevole e di dover maturare ancora esperienza. Sono consapevole del fatto che vengono concesse diverse ore, ma per una prima esperienza il doverle distribuire in un breve lasso temporale, nel mio caso è stato alquanto difficoltoso, essendo io un lavoratore studente non proprio giovane.

Mi sono ritrovato a lavorare al progetto 9-10 ore consecutive nei giorni liberi, per cercare di portare a termine la mia parte, nel tentativo di compensare le intere giornate in cui gli impegni lavorativi e familiari non me lo permettevano. Questa seconda esperienza (considero la prima il robot componibile proposto a lezione/laboratorio) mi è servita per cercare di mettere in pratica ciò che ho imparato durante il corso, consapevole che questo non è un traguardo ma solo un nuovo inizio per un sentiero da percorrere con gli strumenti che ci sono stati forniti, ricordandomi che nella vita bisogna "sporcarsi le mani" e impegnarsi tutti i giorni per migliorare. Per me è stato molto bello e divertente realizzare questa "alpha stable" di software, mi ha mostrato le numerose lacune e il potenziale che ho.

4.2 Difficoltà incontrate e commenti per i docenti

4.2.1 Giovanni Tentelli

Da sempre mi sono considerato un "lupo solitario", una persona a sé e cerco sempre di evitare i gruppi, non che non riesco a connettermi, o relazionarmi, o, in questo caso, lavorare per un obiettivo comune.
Nonostante ciò, sono sempre stato cordiale e aperto, sapevo delle condizioni del mio collega e mi sono adattato di conseguenza.
Ho dato spazio per idee nuove e soluzioni e dato la mia disponibilità qualora il mio collega incappasse in problemi che non poteva risolvere da sé.

4.2.2 Riccardo Fonti

Ho cercato di portare a termine il lavoro nei tempi previsti, con discrezione e grande rispetto per il collega, mettendo a sua disposizione il mio poco tempo e all'unico scopo di suscitare in lui una partecipazione attiva e proficua al percorso intrapreso in comune accordo.
Quando Giovanni ha deciso di non usare il repository concordato ad inizio progetto, in agosto, per utilizzarne uno suo personale ho iniziato a riscontrare alcune difficoltà. Questo cambio mi è stato comunicato solo verso la fine di settembre, rendendomi impossibile trasferire i miei commit (**che potete trovare qui**).
Un'altra difficoltà riscontrata sempre a fine settembre si è verificata quando ha condiviso con me il suo materiale e ho scoperto che JUnit non era utilizzabile, perché generava errori; infatti, nella release finale, Giovanni ha rimosso le mie classi di test. Ho aggirato questi problemi continuando a lavorare sul vecchio repository facendo poi upload manuale sul nuovo.
Le ultime difficoltà riscontrate sono state la stesura della parte di relazione in comune, dell'intera grafica e di tutte le immagini presenti (Giovanni mi ha passato i due uml rifatti alla fine), poiché ho dovuto rifarla due volte: due giorni prima della consegna, Giovanni mi ha segnalato di aver modificato alcune classi della sua parte senza fornirmi sorgenti corretti e che avrei dovuto modificare le immagini e uml, quindi rigenerare il tutto, poiché lui

non ricordava cosa avesse aggiunto nelle classi. Ho partecipato anche alla stesura della parte di design della view per aiutarlo, dal momento che si sentiva in difficoltà.
Avrei preferito avere tempi più lunghi per poter realizzare il progetto: 2/3 mesi per me sono pochi perché i momenti a mia disposizione sono dettati da numerosi impegni lavorativi e famigliari.

Appendice A

Guida Utente

All'avvio del software verrà mostrata la lista di studenti osservati se presenti e verrà data la possibilità di sceglierne uno o aggiungere un nuovo studente.
In basso saranno presenti due pulsanti: uno che apre la finestra per visualizzare i grafici e uno per creare (o sovrascrivere se già presente) un file pdf contente il nome di ogni studente e tutte le osservazioni fatte e il relativo numero di click eseguiti che si troverà nella cartella user/Observations. Una volta selezionato lo studente comparirà una cosa analoga per i momenti. Nel riquadro delle date c'è la possibilità di inserire un nuovo atteggiamento e di creare una nuova data, una volta scelti tutti gli atteggiamenti tramite il tasto "+" si dovrà scegliere la data in alto e cominciare a cliccare i bottoni dopo ogni atteggiamento riscontrato. Una volta selezionata la data ci sarà la possibilità di cliccare ogni singolo atteggiamento ogni qualvolta si presenti.

Esercitazioni di laboratorio B

B.0.1 Giovanni Tentelli

- Persi sul portatile

B.0.2 Riccardo Fonti

- Laboratorio 05
  https://virtuale.unibo.it/mod/forum/discuss.php?d=87881#p135206
- Laboratorio 06
  https://virtuale.unibo.it/mod/forum/discuss.php?d=87880#p135209
- Laboratorio 07
  https://virtuale.unibo.it/mod/forum/discuss.php?d=88829#p136412
- Laboratorio 08*
  https://github.com/ThErgony/OOP-Lab08
- Laboratorio 10*
  https://github.com/ThErgony/OOP2021-Lab10

* scusate ero convinto di averli caricati anche su virtuale ma non me li ritrovo.