1506
UNIVERSITÀ
DEGLI STUDI
DI URBINO
CARLO BO

Progetto di Programmazione
Modellazione ad Oggetti

Sessione estiva 2023/2024

Professoressa Sara Montagna

Giulia Costa
N. Matricola: 308558

Alessandro ScodaVolpe
N. Matricola: 296151

Indice

1 Specifica del Progetto

Considerando l'uso sempre crescente di dispositivi elettronici e il ritmo frenetico della vita moderna, abbiamo deciso di sviluppare questo gestionale per offrire agli utenti un sistema completo ed efficiente per la gestione delle finanze personali. Il gestionale integra funzionalità simili a quelle di un servizio di pagamento digitale, permettendo di tracciare tutte le operazioni effettuate. Inoltre, include un'agenda personale per annotare eventi e avere promemoria, nonché una funzione "Salvadanaio" per mettere da parte denaro e raggiungere i propri obiettivi finanziari.

Il nome dell'applicativo indica le sue due funzioni principali: la pianificazione di attività attraverso l'uso di un calendario ('Plan') e la gestione di un proprio conto con saldo, inclusa la possibilità di effettuare movimenti da e verso i salvadanai creati dall'utente ('Pay').

Ogni utente dovrà prima di tutto registrarsi, fornendo un nome, un utente e una password, informazioni che verranno salvate per le sessioni successive. Successivamente, potrà accedere all'applicazione autenticandosi con utente e password, consentendogli di gestire le proprie attività e finanze.
Al momento del login sarà possibile accedere al calendario delle proprie attività, ai servizi finanziari e alla creazione di obbiettivi di risparmio, oltre che ad una lista di transazioni effettuate nella sessione attuale.

Dal calendario gli utenti potranno muoversi tra i vari mesi, visualizzare gli eventi salvati in precedenza, aggiungere nuovi eventi, specificando data, orario, nome, descrizione e stato di completamento e visualizzare una legenda con lo stato di tutti gli eventi presenti.
Gli eventi saranno modificabili ed eliminabili.
Sarà anche possibile programmare eventi su più giornate, con la possibilità di modificare separatamente la descrizione di ogni singola attività per giorno, mentre orario, stato e nome saranno modificati per tutti i giorni contemporaneamente. Gli utenti potranno inoltre gestire l'eliminazione di singole attività o dell'intero evento.

Tramite i servizi finanziari di deposito e prelievo sarà possibile effettuare movimenti dall'esterno verso il conto e dal conto per le spese, ciascun movimento avrà una causale, un importo e una data.
L'applicativo terrà conto delle transazioni effettuate in quella sessione tramite una lista visualizzabile dall'utente. Le transazioni verranno visualizzate secondo un ordine cronologico e conterranno informazioni sul movimento (causale, importo, data) compreso il tipo di operazione effettuata rispetto all'account, indicando con un segno negativo le uscite e con un segno positivo le entrate. Inoltre se associate ad un obbiettivo, conterrano il nome relativo ad esso.
I prelievi dall'account saranno permessi solo se il saldo sarà sufficiente.

Sarà presente una lista di obbiettivi di risparmio creati durante la sessione, che sarà visualizzabile e da cui si potranno aggiungere o eliminare.
Ciascuno di essi avrà un nome, una descrizione, una data e una soglia di risparmio, ovvero un ammontare entro il quale non sarà possibile prelevare denaro da esso ma solo depositarlo fino al suo raggiungimento.
Gli obbiettivi saranno modificabili per quanto riguarda descrizione e soglia di risparmio ma non saranno rinominabili e non sarà possibile crearne con lo stesso nome.
L'utente inoltre avrà la possibilità di fare previsioni sul raggiungimento di una certa soglia di risparmio entro una data finestra temporale, tenuto conto delle mensilità di versamento, scegliendo se tenere conto o meno del saldo già versato

1.1 Requisiti

Il diagramma di casi d'uso di PlanPay illustra le principali interazioni tra l' utente e il sistema. Per garantire una user experience ottimale, PlanPay mette a disposizione un'interfaccia grafica (GUI) intuitiva. All'interno del diagramma sono visualizzati in modo dettagliato i passi descritti nella specifica del progetto.

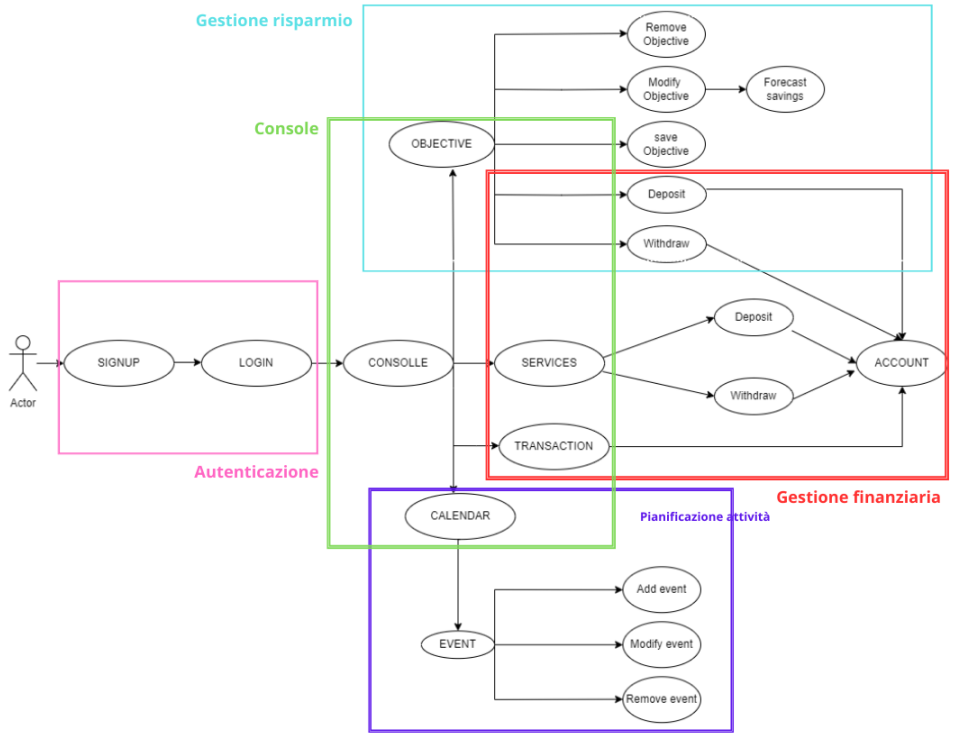

Figura 1: *Use case di PlanPay*

1.2 Modello del dominio

Il modello del dominio di PlanPay descritto di seguito illustra le entità principali coinvolte nell'applicazione e le relazioni tra di esse. Questa fase di analisi è cruciale e precede la progettazione e lo sviluppo del software.
Nel dominio di PlanPay, il LoginController rappresenta l'entità responsabile della gestione delle autenticazioni degli utenti. Esiste una relazione uno-a-uno con il ConsoleController, il che significa che per ogni sessione utente autenticata, c'è una console associata. Il ConsoleController gestisce l'interazione dell'utente con le funzionalità principali dell'applicazione sotto definite.
L'interfaccia Data gestisce i dati associati agli obbiettivi, agli eventi e alle transazioni, fungendo da interfaccia comune per l'accesso alle informazioni.
L'Account rappresenta il conto finanziario dell'utente e può essere coinvolto in varie operazioni finanziarie, gestite tramite la classe astratta AbstractOperations, che consente operazioni comuni effettuate sull' account, come depositi e prelievi.
La relazione di composizione uno-a-molti fra Account e Transaction tiene traccia dei movimenti effettuati da AbstractOperations.
Objective, invece, rappresenta gli obiettivi impostati dall'utente, sui quali è possibile effettuare operazioni di deposito e prelievo, anch'esse gestite dalla classe astratta AbstractOperation.
Il ConsoleController ha una relazione di composizione uno-a-molti con AbstractOperations, permettendo la gestione di più operazioni finanziarie.
CalendarP rappresenta il calendario dell'utente e può contenere più eventi, indicati da una relazione uno-a-molti. Ogni calendario infatti può avere molti eventi associati, che a loro volta possono essere collegati ad un singolo utente.

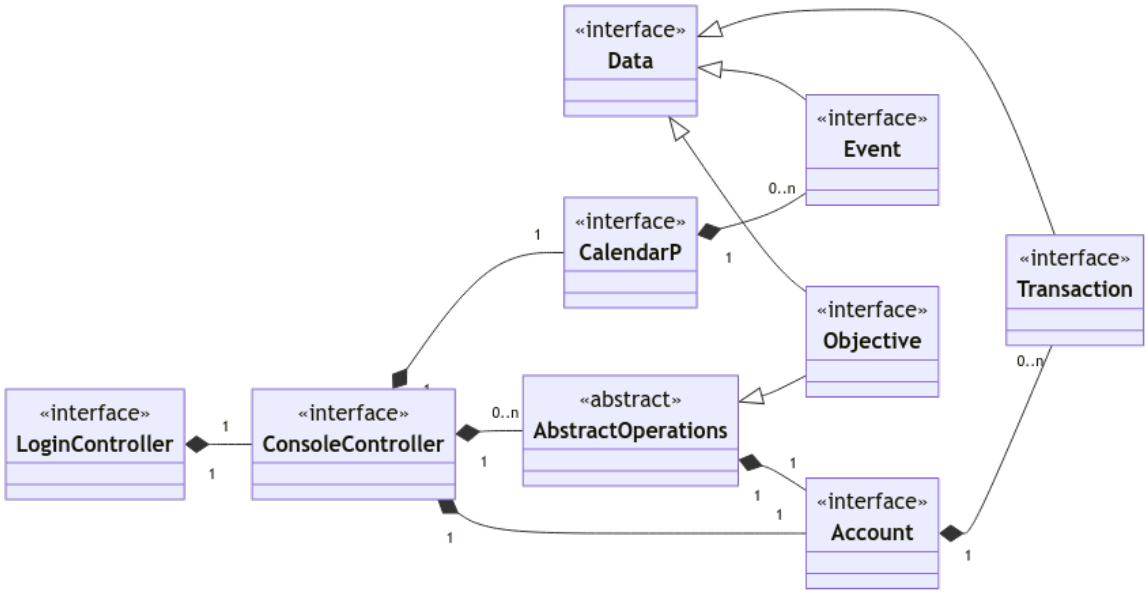

Figura 2: *Diagramma UML di PlanPay*

2 Design

Le classi principali di PlanPay sono LoginController e ConsoleController, fondamentali per accedere a tutte le funzionalità dell'applicazione sviluppata. Il LoginController gestisce l'autenticazione degli utenti, permettendo loro di accedere al sistema in modo sicuro. Una volta autenticato, l'utente interagisce con il ConsoleController, che funge da hub centrale per la gestione delle operazioni principali dell'applicazione. Tutte le macrofunzioni sono gestite dalle seguenti classi: CalendarImpl, TransactionImpl, Services e ObjectiveImpl, che integragiscono con il ConsoleController.

- *La classe CalendarImpl* gestisce tutte le operazioni relative al calendario dell'utente, come l'aggiunta, la modifica e la rimozione di eventi, utilizzando le istanze di EventImpl per rappresentare le informazioni.
- *La classe TransactionImpl* si occupa di tener traccia di ogni movimento finanziario.
- *Services* gestisce le operazioni di deposito e prelievo, consentendo movimenti di denaro dall'esterno verso l'account dell'utente e viceversa.
- *ObjectiveImpl* gestisce gli obiettivi di risparmio dell'utente che possono essere creati, modificati e eliminati. Inoltre permette operazioni finanziarie simili a quelle gestite da Services, consentendo versamenti e prelievi sul saldo degli obbiettivi. Gli utenti possono anche fare previsioni sul raggiungimento di una certa soglia di risparmio entro una data finestra temporale.

Di seguito saranno definiti la visione architetturale implementata, lo use case e il modello del dominio. I dettagli implementativi delle classi e delle operazioni descritte saranno trattati nella sezione successiva, fornendo una panoramica completa del funzionamento interno dell'applicazione.

2.1 Architettura

Nel progetto PlanPay, abbiamo adottato il pattern architetturale Model-View-Controller (MVC) per garantire una separazione chiara delle responsabilità, facilitando così la manutenzione e l'estensibilità del codice.

Il Controller si occupa della gestione dell'interazione dell'utente. È responsabile di ricevere l'input, elaborare le richieste e aggiornare il Model e la View. In sostanza, agisce come un intermediario tra il Model e la View. Nel progetto PlanPay, sono stati implementati due controller principali, ciascuno con ruoli specifici, per garantire una gestione ottimale delle operazioni e delle interazioni dell'utente: LoginControllerImpl, responsabile dell'autenticazione, e ConsoleControllerImpl, che gestisce le operazioni principali dell'applicazione. Questa separazione permette una maggiore chiarezza, manutenibilità e modularità del codice. In generale entrambi svolgono le seguenti principali funzioni:

- *Gestione dell'input utente:* il Controller riceve gli input dall'utente, come click sui pulsanti e inserimento di dati nelle form, provenienti dalle varie View implementate.
- *Elaborazione delle richieste:* una volta ricevuto l'input, il Controller elabora la richiesta. Questa elaborazione può includere la validazione dei dati, la gestione delle eccezioni e la logica di business. Ad esempio, per le richieste di login o salvataggio di un obiettivo, vengono richiamati i metodi presenti nel Model.
- *Aggiornamento del Model:* il Controller aggiorna il Model in base all'input dell'utente. Ad esempio, se l'utente crea un nuovo evento nel calendario, il Controller aggiorna il Model aggiungendo questo evento.
- *Aggiornamento della View:* questo è l'ultimo passaggio, utile per riflettere le modifiche a livello grafico. Questo può includere l'aggiornamento della lista degli obiettivi o il caricamento degli eventi.

Il Model è responsabile della gestione dei dati e della logica interna, interfacciandosi con dei file di testo e rispondendo alle richieste del Controller. In particolare, nel nostro progetto, questi componenti si occupano del signup e login dell'utente, della gestione delle transazioni finanziarie, obbiettivi personali, gestione del calendario e interazione con dei file di testo.

Le View sono responsabili della presentazione dei dati all'utente e dell'interazione con esso. Nel progetto PlanPay si occupano quindi di visualizzare le informazioni provenienti dal Model e di inviare l'input dell'utente ai Controller. Esempi significativi di 'visualizzazione dei dati' includono la rappresentazione di tutte le transazioni effettuate, di un calendario con gli eventi creati al suo interno, e di tutti gli obbiettivi con il relativo saldo. Interagendo con i Controller per la gestione dei dati, le View aggiornano l'interfaccia utente per riflettere le modifiche apportate.

3 Design dettagliato

Di seguito saranno riportate in maniera dettagliata tutte le implementazioni effettuate da ciascun membro del gruppo.

3.1 Giulia Costa

Classi e interfacce realizzate:

- *ConsolleController.java, ConsolleControllerImpl.java*
- *LoginController.java, LoginControllerImpl.java*
- *CalendarP.java, CalendarImpl.java, CalendarModel.java, DayCellRenderer.java*
- *Event.java, EventImpl.java, EventAdapter.java, ComparatorEvents.java*
- *Login.java, LoginImpl.java, UserCredentials.java*
- *Transaction.java*
- *Enum: Month.java, State.java*
- *Classi per eccezioni: AuthenticationException.java, RegistrationException.java, EventAlreadyExistsException.java, EventNotFoundException.java, IllegalOperationException.java, InvalidParameterException.java*
- *View: LoginView.java, SignupView.java, ConsolleView.java, CalendarView.java, EventView.java, SelectedEventView.java*
- *Test: LoginTest.java, CalendarTest.java, DataTest.java, EventFileTest.java*

Per ottimizzare la gestione e garantire l'affidabilità del gestionale implementato, si è scelto di utilizzare dei file di testo per la memorizzazione e la lettura dei dati relativi al login degli utenti e alla gestione degli eventi nel calendario.
Il funzionamento del sistema è principalmente concentrato in due componenti chiave: ConsolleControllerImpl e LoginControllerImpl. Queste classi forniscono i metodi necessari per eseguire il login e accedere alle funzionalità del software gestionale. Le principali operazioni supportate includono:

- **SignUp:** creazione del proprio account, con memorizzazione delle credenziali su un file di testo.
- **Login:** login dell'account, con validazione dei dati inseriti.
- **Aggiornamento del conto:** gestione delle operazioni finanziarie sul conto personale degli utenti, aggiornando il saldo personale.

- **Operazioni finanziarie:** consente agli utenti di effettuare operazioni finanziarie sul conto, quali depositi o prelievi.
- **Creazione e gestione degli Obiettivi:** consente agli utenti di impostare e monitorare obiettivi finanziari collegati ai loro conti.
- **Visualizzazione dei movimenti:** permette di consultare tutti i movimenti effettuati sul conto.
- **Gestione del calendario:** offre la possibilità di creare e gestire eventi all'interno del calendario, collegati al proprio nome utente.

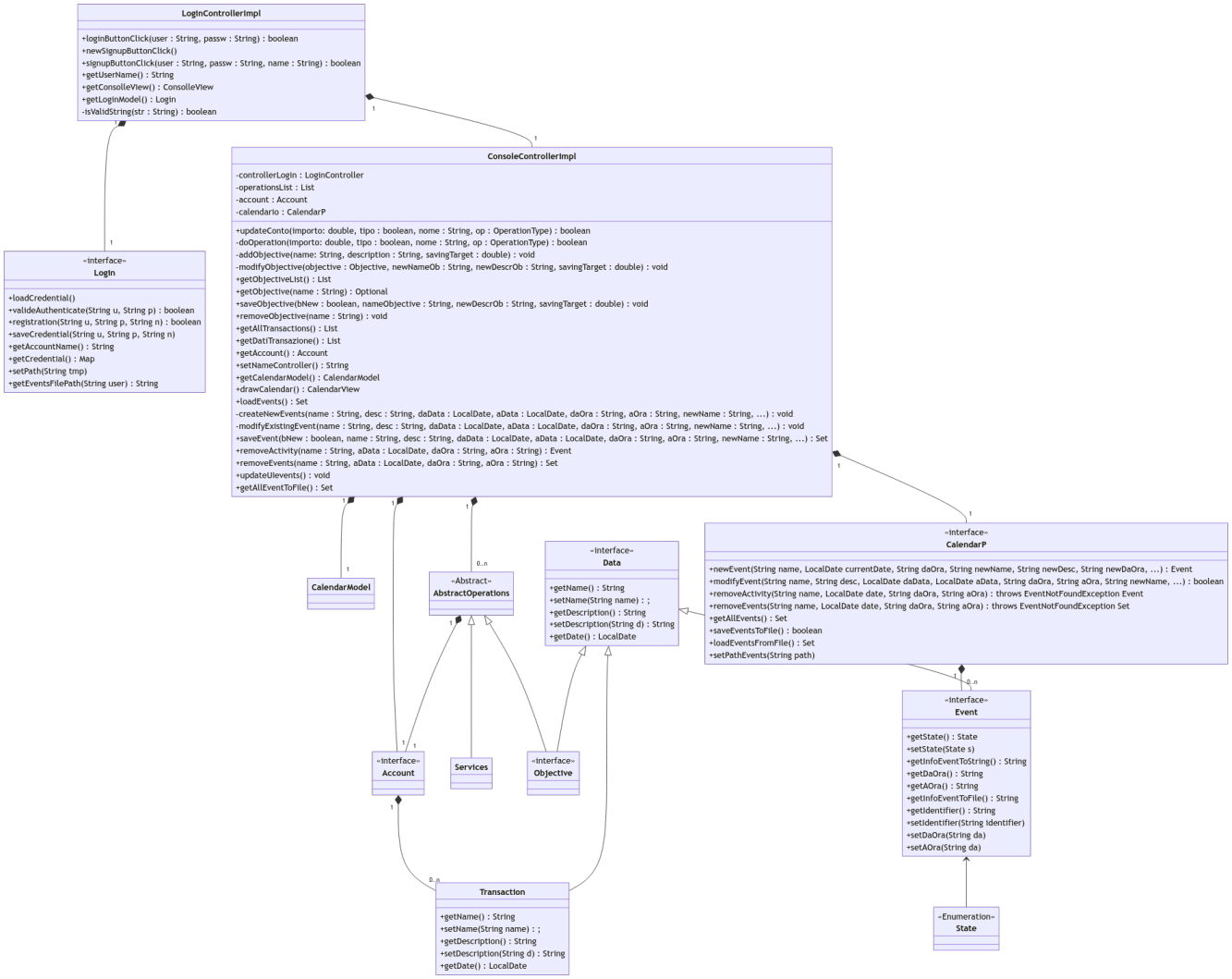

Figura 3: *UML della parte implementata.*

3.1.1 LOGIN

La funzionalità di Login è stata progettata attraverso le classi ***LoginView***, ***SignUpView***, ***LoginController***, e ***LoginImpl***.
La logica per la memorizzazione delle credenziali degli utenti, è stata implementata nella classe **LoginImpl**, responsabile della gestione dei dati relativi agli utenti, come le credenziali e i percorsi dei file degli eventi. Utilizza strutture dati come **HashMap** per memorizzare le credenziali, garantendo un accesso rapido ed efficiente. In questa struttura dati, la chiave è determinata dall' utente, mentre il valore è un oggetto UserCredentials che contiene la password, il nome dell'account e il percorso del file associato a ciascun utente.
Per il salvataggio delle credenziali, si è scelto di utilizzare un file di testo (credentials.txt) in cui vengono archiviati i dati richiesti durante il signup (nome user, utente e password), oltre al percorso del file di testo generato automaticamente per memorizzare gli eventi dell'account, denominato "nomeutente_events.txt". Questo approccio consente di accedere a un determinato profilo già salvato, permettendo di recuperare eventi precedentemente creati.
La scelta di creare un file di testo per ogni utente è stata adottata per assicurare che, nel caso di aggiunta di un nuovo user, i suoi eventi personali possano essere recuperati facilmente. Questo approccio garantisce la scalabilità dell'applicazione, consentendo di gestire senza problemi un numero crescente di utenti e dei loro dati associati.
La classe **LoginControllerImpl** si occupa di gestire il processo di login e registrazione degli utenti, verificando la validità delle credenziali e aggiornando l'interfaccia utente di conseguenza. Inoltre, gestisce la visibilità della Console solo dopo una corretta autenticazione.
SignUpView e **LoginView** gestiscono l'interfaccia utente per la registrazione e il login, rispettivamente. Entrambe le classi sono progettate per essere intuitive e sicure, con campi per l'inserimento delle credenziali e controlli per la loro validità. La visualizzazione della password in modalità nascosta è implementata in LoginView per migliorare la sicurezza.

3.1.2 CONSOLE

La progettazione di ***ConsolleView*** e del controller ***ConsoleControllerImpl*** è stata guidata da diversi principi chiave volti a migliorare la facilità di aggiornamento, la riutilizzabilità del codice e la manutenibilità dell'intera applicazione. Utilizzando **DefaultListModel e JList**, la console può aggiornare in modo dinamico e reattivo la lista delle transazioni e degli eventi. Queste strutture dati permettono di modificare facilmente il contenuto visualizzato senza dover ricostruire interamente la lista, garantendo così un'interfaccia utente sempre aggiornata e senza ritardi.
La lista delle transazioni viene popolata da istanze di TransactionImpl, che tengono traccia di depositi e prelievi avvenuti, più altre informazioni relative alla causale, alla data, all'entità che ha richiesto il movimento e all'ammontare. E' possibile accedere a queste informazioni tramite metodi getter derivati dall'interfaccia Data estesa da Tran-

saction e implementata da TransactionImpl.
La lista delle transazioni viene popolata solo quando viene effettuata un'operazione finanziaria (tramite l'uso delle funzionalità di Services o ObjectiveImpl).
La visualizzazione delle transazioni è gestita con testo scritto in HTML per garantire una visualizzazione più strutturata, inserendo segni + e - di colore rispettivamente verde e rosso, in modo da visualizzare chiaramente i movimenti effettuati sul conto, che sarà aggiornato di conseguenza ad ogni transazione effettuata.
Un altro aspetto importante è la riutilizzabilità del codice, ottenuta attraverso metodi come *setButtonIcon, createLabel, createButton*, che centralizzano la logica per impostare le icone dei pulsanti e creare etichette o pulsanti, permettendo di modificare solo i parametri di passaggio. Questo approccio riduce la duplicazione del codice e facilita la manutenzione. Le strutture dati utilizzate sono state scelte per la loro efficienza e per il loro adattamento alle necessità specifiche dell'applicazione. Ad esempio, la lista **List<AbstractOperations>** consente di gestire diverse operazioni (obiettivi e servizi) in modo uniforme, mentre il set **Set<Event>** nel calendario garantisce che ogni evento sia unico e ordinato rispetto al comparatore implementato. E' stato inoltre utilizzato l'Observer Pattern per garantire flessibilità dell'applicazione. L'Observer Pattern, assicura che ogni volta che cambia lo stato del modello, l'interfaccia utente venga aggiornata di conseguenza. Questo pattern è impiegato in metodi come *updateUIconto*, che aggiorna l'interfaccia della consolle sull' ammontare del conto a seguito di operazioni finanziarie e *updateUIevents*, che aggiorna l'interfaccia della consolle sugli eventi del giorno anche a seguito di creazione o cancellazione di nuovi eventi a tempo di esecuzione.

3.1.3 CALENDARIO

Le classi coinvolte nella gestione del calendario e degli eventi sono molteplici e lavorano insieme per garantire una gestione coordinata, oltre che un sistema flessibile e manutenibile.
La classe ***CalendarImpl*** contiene la logica per la gestione degli eventi, utilizzando la collezione **TreeSet** per memorizzare gli eventi, ordinati cronologicamente grazie al comparatore **ComparatorEvents**. Questo approccio garantisce non solo l'ordine degli eventi, utile per gestire la grafica e visualizzare i nuovi eventi all'interno del calendario ordinati per giorno e per orario di inizio, ma impedisce anche la duplicazione, facilitando le operazioni di ricerca, aggiunta e rimozione degli eventi.
Ogni evento è rappresentato da un'istanza di EventImpl, che include informazioni come nome, descrizione, data, stato e identificatore.
Inoltre, ogni salvataggio, modifica o cancellazione di un evento viene immediatamente riflesso sul file creato al momento del login, garantendo un aggiornamento in tempo reale e la ridefinizione dell'ordine degli elementi in caso di cancellazione.
La classe ***CalendarModel***, estendendo AbstractTableModel, è responsabile della logica di visualizzazione e gestione del calendario.
Utilizza una mappa (**Map<LocalDate, Set<Event>> cellData**) per associare le

date agli eventi, consentendo un accesso immediato alle informazioni di una specifica data e garantendo un aggiornamento rapido sulla UI. Questo modello di dati è ideale per rappresentare il calendario, in quanto permette di gestire efficientemente gli eventi giornalieri. Il metodo *getValueAt* costruisce una rappresentazione HTML degli eventi per ogni cella del calendario, integrando la funzionalità 'toHtml' per colorare gli eventi in base al loro stato (da avviare, in corso, concluso).
La classe ***CalendarView*** rappresenta la finestra principale del calendario utilizzando CalendarModel per gestire la logica dei dati e aggiorna l'interfaccia in base alle interazioni dell'utente. CalendarView gestisce anche la selezione e la modifica degli eventi, aprendo finestre di dialogo (EventView e SelectedEventView) per consentire agli utenti di aggiungere, modificare, eliminare o visualizzare eventi specifici.
La classe ***EventView*** fornisce un'interfaccia dettagliata per la creazione, la modifica e la cancellazione degli eventi. Include campi per inserire il titolo, la descrizione, le date e gli orari, nonché il loro stato. Durante la creazione di un nuovo evento, viene generato un identificatore univoco che consente di gestire gli eventi collegati. EventView verifica anche la validità dei dati inseriti dall'utente, garantendo che le date e gli orari siano coerenti. Durante un nuovo inserimento, viene mostrato solo il tasto di salvataggio, mentre in caso di modifica viene reso visibile anche il bottone di cancellazione. Per eventi su più giornate, i bottoni visualizzati sono due: uno usato per cancellare l'evento del giorno selezionato e uno usato per cancellare l'intero evento. La cancellazione comporta anche l'aggiornamento della legenda nel calendario per lo stato dell'attività e della lista degli eventi del giorno o futuri nella console. Se un evento già esistente viene modificato, l'identificatore viene utilizzato per aggiornare attributi come nome, stato e orario per tutte le occorrenze. Tuttavia, la descrizione può essere modificato individualmente per ciascun giorno selezionato.
La data non è modificabile per scelta implementativa; pertanto, l'evento deve essere cancellato e poi ricreato se la data necessita di essere cambiata.
SelectedEventView, invece, offre una panoramica degli eventi in un determinato giorno, permettendo agli utenti di selezionarne uno specifico per visualizzarne i dettagli o modificarlo.
L'implementazione utilizza l'**Observer Pattern**, implementato tramite fireTableDataChanged e fireTableCellUpdated, assicurando che la vista del calendario venga aggiornata ogni volta che il modello cambia. Per quanto riguarda l'interfaccia ***Event***, è stata utilizzata l'**ereditarietà** a più livelli, assicurando che EventImpl non solo rispetti i requisiti specifici della classe Event, ma anche quelli della classe Data, obbligandola a implementare i metodi definiti in entrambe le interfacce. In questo modo, è possibile utilizzare oggetti di tipo Event per richiamare metodi di entrambe le interfacce, garantendo una maggiore flessibilità nel codice.

3.2 Alessandro Scodavolpe

Classi e Interfacce realizzate:

- *Account.java, AccountImpl.java*
- *AbstractOperations.java*
- *Objective.java, ObjectiveImpl.java*
- *Services.java*
- *Enum: OperationType.java*
- *Classi per eccezioni: IllegalInputExcpetion.java*
- *View: ServicesView.java, ObjectiveView.java, ConsolleObjectiveView.java, ListObjectiveView.java, ForecastView.java*
- *Test: AccountTest.java, ControllerOperationTest.java, ServicesTest.java, ObjectiveTest.java, TransactionTest.java*

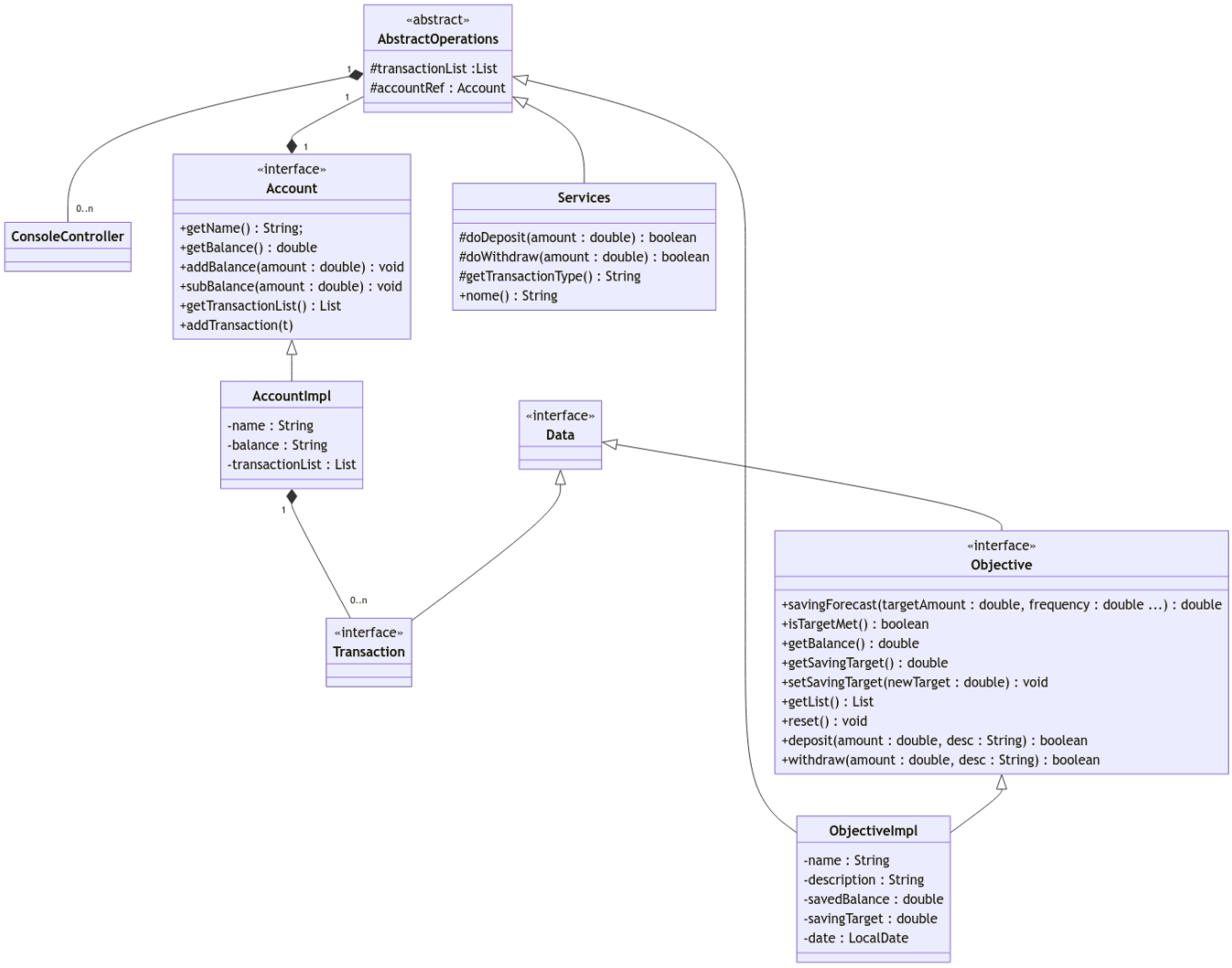

Figura 4: *UML della parte implementata.*

3.2.1 RISPARMIO

Per la gestione del risparmio le funzionalità principali sono implementate in ***ObjectiveImpl***. La classe dispone di campi come nome, descrizione e data di creazione, pertanto per favorire il riuso del codice, si è scelto di far estendere Data all'interfaccia Objective, per poter accedere e modificare queste informazioni. Vi è anche un valore che rappresenta l'ammontare che si vuole risparmiare, versando denaro sul saldo di Objective. Il nome dell'obbiettivo rappresenta la chiave di ricerca per la lista degli obbiettivi, pertanto si è scelto di non renderlo modificabile.
Objective dispone anche di una funzionalità aggiuntiva, che implementa un algoritmo per il calcolo dell'ammontare da versare per raggiungere una determinata soglia di risparmio, secondo un periodo di tempo espresso in anni e/o mesi. Il metodo calcola il

rapporto tra la frequenza di versamento e la soglia da raggiungere, per trovare il numero di mensilità in cui si dovrà versare denaro all'interno del periodo specificato, dividendo poi l'ammontare per questo valore, ottenendo il risultato. Per implementare questo algoritmo si è fatto ricorso alla libreria **Math** di *java.lang* che offre utili funzioni per il calcolo numerico, utilizzando il metodo *floor* per l'arrotondamento per difetto del valore reale risultante dall'operazione. Il metodo può tenere conto o meno del saldo presente al momento del calcolo; naturalmente in caso la soglia fosse eguagliata o superata dal saldo attuale la quantità da versare risulterà sempre nulla.
La classe ***ConsolleObjectiveView*** consente la visualizzazione di una finestra con la lista degli ObjectiveImpl creati e pulsanti per la creazione di nuove istanze o per l'eliminazione di tutta la lista. Gli obbiettivi sono rappresentati da istanze di ***ListObjectiveView***, classe la quale implementa un pannello in cui mostra nome e saldo dell'obbiettivo in questione. Quest'ultima è provvista di pulsanti per l'eliminazione dell'obbiettivo e per l'apertura della ***ObjectiveView*** corrispondente, che mostra una finestra per la modifica e il salvataggio dell'obbiettivo. ObjectiveView inoltre offre in questa finestra un bottone per l'accesso a ForecastView e un altro bottone per l'apertura di una finestra ServiceView, che consentirà di effettuare operazioni di deposito o prelievo riguardanti l'obbiettivo corrispondente.
Per la visualizzazione di un numero qualsiasi di obbiettivi su ConsolleObjectiveView è stato impiegata la classe di libreria swing *JScrollPane*, che ha consentito di inserire istanze di ListObjectiveView in un pannello provvisto di scrollbar in caso di necessità.
ForecastView rappresenta l'interfaccia attraverso la quale l'utente può sfruttare la funzionalità di previsione del metodo *savingForecast* di ObjectiveImpl, indicando negli appositi campi di testo e cliccando sul bottone 'Calcola' per ottenere il risultato a schermo.

3.2.2 GESTIONE FINANZIARIA

Il processamento dei pagamenti consiste nella modifica del saldo, implementato come un valore double, e nel salvataggio delle informazioni relative ai movimenti, istanziando una nuova *TransactionImpl* e aggiungendola a una struttura dati di tipo Lista. Entrambi sono campi privati accessibili solo da Account, per rispettare il principio di incapsulamento tipico della Programmazione ad Oggetti. Il ruolo di ***AccountImpl*** sarà quindi di gestire le richieste di deposito o prelievo dal conto e applicherà la seguente politica: consentirà sempre i depositi incrementando il proprio saldo, mentre in caso di prelievo verificherà che ci sia un ammontare sufficiente nel conto, impedendo di ottenere un saldo negativo. Per le operazione su Account si è sfruttato il Design Pattern **Template Method**. Questo paradigma si basa sull'uso di una classe astratta che definisce funzioni di base di un algoritmo che verranno poi estese e completate dalle sottoclassi concrete per avere comportamenti diversi. La classe astratta in questione è ***AbstractOperations***, la quale possiede le funzionalità base per fare richiesta di prelievo o deposito tramite i metodi deposit e withdraw. Il *template method* in questo caso consiste nella chiamata a dei metodi astratti che verranno concretizzati in maniera differente

nelle sottoclassi, mentre la funzione base è la creazione di un nuovo oggetto TransactionImpl fornendogli le informazioni relative al movimento, e salvandolo nella lista delle Transazioni presente nell'Account. Questa lista consentirà di tenere traccia di tutti i movimenti effettuati da ciascuna istanza di classe che estende AbstractOperations. Le classi ***Services e ObjectiveImpl*** sono le sottoclassi concrete di AbstractOperations e dispongono dei metodi ereditati da quest'ultima per effettuare richieste di deposito o prelievo ad Account. ***Services*** concretizza i template method per modificare semplicemente il contenuto del saldo tramite doWithdraw e doDeposit, facendone richiesta all'Account.
Objective andrà a modificare il proprio saldo qualora le richieste di prelievo o deposito ad Account venissero confermate. Il deposito in ObjectiveImpl infatti richiederà prima una richiesta di decremento del Saldo da Account per poi incrementare il proprio, mentre una richiesta di prelievo da un ObjectiveImpl sarà gestita in modo differente. In questo caso infatti ObjectiveImpl verificherà che l'ammontare del suo saldo sia non nullo e uguale o superiore alla soglia dell'obbiettivo inizializzata al momento dell'istanziamento. Dato che le istanze di ObjectiveImpl possono essere eliminate dall'utente una volta create, è presente una funzionalità che evita che il saldo versato venga perso, resettando la soglia risparmio dell'obbiettivo e depositando l'ammontare presente sul conto dell'utente.
ServicesView rappresenta la classe di interfaccia grafica, che mostra una finestra in cui l'utente può specificare una quantità monetaria espressa in euro e una causale, per poi eseguire l'operazione desiderata cliccando sul relativo pulsante. Si può accedere a questa interfaccia sia da ConsoleView che da ObjectiView cliccando sull'apposito pulsante, che salverà l'obbiettivo e visualizzerà ServiceView. In questo caso sarà presente una causale predefinita non modificabile, determinata dal nome dell'obbiettivo. Verranno eseguiti controlli sulla presenza o meno della causale (in caso si acceda tramite ConsoleView) e di una cifra valida, in caso negativo apparirà una finestra di warning, i campi verranno resettati e non verrà svolta alcuna operazione, nel caso in cui gli input siano corretti verrà effettuata una richiesta al Controller, che procederà ad effettuare controlli per distinguere il tipo di operazione e il suo segno, con eventuali chiamate ai metodi di Services o di Objective secondo il pattern MVC.
Per la formattazione delle cifre nei textField delle classi View è stato utilizzato *DecimalFormat*, che ha consentito di avere sempre input formattati come valuta quando necessario.

4 Sviluppo

Per garantire un'efficiente collaborazione e sviluppo del progetto, abbiamo adotatto una serie di metodologie e strumenti:

- **Eclipse**, come IDE per lo sviluppo in Java, ideale in quanto mette a disposizione innumerevoli funzionalità e plugin per rendere più agevole il lavoro.
- **WindowBuilder**, come PlugIn di Eclipse per facilitare lo sviluppo della GUI, consentendo di posizionare gli elementi grafici tramite un'interfaccia agevole e intuitiva.
- **GitHub**, come repository remota per ospitare il codice sorgente. Questa piattaforma ci ha permesso di sviluppare l'applicativo in maniera collaborativa, lavorando contemporaneamente sullo stesso progetto. Per facilitare il controllo di versione e l'archiviazione del codice abbiamo utilizzato Egit direttamente da Eclipse, che ci ha permesso una sincronizzazione più immediata e facilitata del codice, interfacciandosi direttamente con la repository GitHub.
- Per la modellazione e la creazione di diagrammi, abbiamo utilizzato **Mermaid Live Editor** per generare diagrammi UML e **Lucidchart** per definire i casi d'uso.
- Per mantenere un allineamento degli obbiettivi di sviluppo abbiamo organizzato call settimanali utilizzando **Google Meet**. Durante queste riunioni abbiamo discusso sui passi necessari per la creazione del progetto, sulle priorità e metodologie di sviluppo. Questo ci ha permesso di coordinare in maniera efficace il lavoro in team.

4.1 Testing

Per il testing delle funzionalità dell'applicativo è stato utilizzato il framework JUnit 5, che ha consentito la verifica della correttezza dei metodi implementati nella varie classi che costituiscono la parte di Model e Controller del progetto. Si è fatto ricorso a questo framework in fase di debug, in quanto ci ha consentito di isolare e testare parti del codice di una classe, permettendoci una facile individuazione di errori e problemi durante lo sviluppo.
In particolare è stato possibile verificare che i metodi testati restituissero il valore che ci si aspettava, o che sollevassero un'exception corretta per tutte le classi testate. In questa maniera abbiamo potuto poi dedicarci al testing della GUI e quindi della parte di View di progetto, che per funzionare correttamente richiedeva la correttezza delle funzionalità della parte di Controller e Model.

Le classi per il testing sono presenti nel package Test e sono le seguenti:

- *AccountTest.java*
- *CalendarTest.java*
- *ControllerOperationTest.java*
- *DataTest.java*
- *EventFileTest.java*
- *LoginTest.java*
- *ObjectiveTest.java*
- *ServicesTest.java*
- *TransactionTest.java*

4.1.1 Giulia Costa

Classi di test coinvolte:

- **CalendarTest:**
  si focalizza sulla verifica delle funzionalità delle classi coinvolte per la gestione degli eventi nel calendario, inclusi la creazione, modifica, rimozione e gestione generale degli eventi. La classe di test controlla che gli eventi vengano creati e aggiunti correttamente al calendario. Inoltre, se ci sono eventi su più giornate, verifica che le modifiche al nome, allo stato e agli orari siano applicate a tutti gli eventi associati, mentre le modifiche alla descrizione siano specifiche per ciascuna giornata. Viene inoltre testato che, per gli eventi su più giornate, in caso di cancellazione di una singola attività, venga eliminata solo la giornata selezionata, mentre in caso di cancellazione dell'intero evento, vengano eliminati tutti gli eventi associati.
  Infine, si testano le eccezioni specifiche del dominio per garantire che il sistema gestisca correttamente le condizioni di errore, come la creazione di eventi duplicati o l'uso di parametri non validi.

- **EventFileTest:**
  è progettata per assicurarsi che gli eventi del calendario possano essere salvati su file e caricati correttamente, assicurando che i dati degli eventi rimangano intatti e accessibili.

- **DataTest:**
  si concentra sulla verifica delle funzionalità delle classi *EventImpl TransactionImpl*, che rappresentano rispettivamente un evento e una transazione. I test assicurano che i getter e setter delle proprietà degli eventi e delle transazioni funzionino

correttamente, verificando che le proprietà possano essere impostate e recuperate come previsto.
Inoltre, vengono testati i metodi equals e hashCode per garantire che gli eventi possano essere correttamente confrontati e utilizzati nelle strutture dati basate su hash.

- **LoginTest:**
  verifica le funzionalità di autenticazione e registrazione degli utenti gestite dalle classi *LoginControllerImpl* e *LoginImpl*. Questi test assicurano che gli utenti possano registrarsi correttamente, che le credenziali degli utenti vengano salvate e caricate correttamente da un file, e che gli utenti possano autenticarsi con successo.
  Inoltre, vengono testate le eccezioni relative a registrazione e login per garantire che il sistema gestisca correttamente le condizioni di errore, come tentativi di registrazione di utenti duplicati o login con credenziali errate.

4.1.2 Alessandro Scodavolpe

Classi di test coinvolte:

- **AccountTest:**
  verifica le funzionalità dell'account testandone l'inizializzazione dei campi e i metodi di incremento e decremento del saldo, testando che non sia possibile mandare in negativo il saldo.

- **ControllerOperationTest:**
  dopo aver effettuato test sull'inizializzazione dei campi principali del controller come l'account, la lista obbiettivi e i dati delle transazioni, si procede con la verifica dei metodi di gestione degli obbiettivi, come la ricerca nella lista, il salvataggio, la modifica e l'eliminazione di obbiettivi, inoltre si testano i casi in cui viene sollevata un'exception:
  in caso di tentativo di salvataggio di un obbiettivo con lo stesso nome di un altro presente in lista oppure quando si tenta la modifica o l'eliminazione di un obbiettivo non più presente nella lista.
  Vengono poi testate le funzionalità di gestione delle richieste di prelievo e deposito sia per Services che per ObjectiveImpl, verificando i saldi del conto e dell'obbiettivo vengano aggiornati al valore corretto. Inoltre si verifica che il tentativo di operazione su obbiettivi non salvati generi un'exception e che si possa prelevare da un obbiettivo solo quando questo ha un saldo uguale o superiore alla soglia di risparmio fissata.

- **ObjectiveTest:**
  si verifica la correttezza dei metodi getter e setter per i campi di ObjectiveImpl, in aggiunta sono testati i metodi ereditati e concretizzati da AbstractOperations per i movimenti finanziari e la funzionalità di previsione dei versamenti per il risparmio,

verificando che quest'ultima sollevi excpetion in caso di dati di input errati. Infine viene verificato che il reset degli obbiettivi in preparazione all'eliminazione agisca come previsto, portando a zero la soglia di risparmio e muovendo tutto il saldo su Account.

- **ServicesTest:**
  testing dell'inizializzazione dei campi di Services e della correttezza delle funzioni ereditate e concretizzate da AbstractOperations per depositare e prelevare su Account.

- **TransactionTest:**
  vengono effettuate operazioni per verificare il corretto istanziamento e popolamento di *transactionList* in AccountImpl. In seguito si accerta che le transazioni presenti nella lista *transactionList* di AccountImpl, contengano le informazioni corrette, testando anche il funzionamento del metodo *getAllTransaction()* di ConsoleControllerImpl e i metodi *getTransactionType()* di Services e ObjectiveImpl

4.2 Note di sviluppo

4.2.1 Giulia Costa

- *Optional:* usati per gestire l'assenza di valori e lanciare eccezioni personalizzate (ad esempio EventNotFoundException, EventAlreadyExistsException).

- *Stream:* facilita operazioni complesse su collezioni come filtraggio e mappatura, migliorando la concisione e la leggibilità del codice.

- *Lambda Expression:* sono state utilizzate soprattutto per operazioni su collezioni, come iterazioni, filtri e mappe, migliorando la leggibilità del codice.

- *BufferWriter e BufferReader:* per lettura e scrittura di file di testo.

- *Stringbuilder e HTML:* per la visualizzazione degli eventi nella tabella.

- *Comparator:* per garantire un ordine cronologico degli eventi, facilitando le operazioni di ricerca, aggiunta, rimozione e visualizzazione sul calendario degli eventi e sul file di testo.

- *JCalendar:* usato per fornire una componente grafica avanzata che facilita la gestione e la visualizzazione del calendario degli utenti. In particolar modo questa libreria mette a disposizione una serie di strumenti che permettono di selezionare date, visualizzare mesi e anni, e interagire con gli eventi in modo intuitivo.

- *JPasswordField:* per garantire la sicurezza dell'input delle password durante il processo di autenticazione. Questa componente di Swing offre una casella di testo che maschera l'input dell'utente con caratteri di default (come asterischi), impedendo la visualizzazione della password.

- *JDateChooser:* per fornire una componente grafica avanzata che facilita la selezione delle date da parte degli utenti.

Gli algoritmi degni di nota che sono stati utilizzati, sono indicativamente 3:

- ***getExistingEvent*** per la classe *CalendarImpl* il cui scopo è quello di cercare un evento esistente in un determinato giorno che si sovrapponga a un intervallo di tempo specificato.
  La funzione prende in input un identificatore dell'evento, una data, un'ora di inizio e un'ora di fine. Inizia filtrando tutti gli eventi presenti in quella data, che non abbiano lo stesso identificatore, per evitare di considerare l'evento stesso come un conflitto. Successivamente, la funzione applica un ulteriore filtro agli eventi trovati per controllare se l'intervallo di tempo del nuovo evento si sovrappone a quello di un evento esistente. La sovrapposizione è determinata verificando se l'ora di inizio del nuovo evento è prima dell'ora di fine di un evento esistente e se l'ora di fine del nuovo evento è dopo l'ora di inizio di un evento esistente. Inoltre, viene verificato se l'ora di inizio o l'ora di fine del nuovo evento coincidono con quelle di un evento esistente. Se viene trovato un evento che soddisfa le condizioni di sovrapposizione, la funzione lo restituisce come Optional<Event>. Se nessun evento soddisfa le condizioni, viene restituito un Optional.empty().

- ***initializeDays*** per la classe *CalendarModel* è responsabile dell'inizializzazione di una lista di giorni per il mese corrente. L'obiettivo principale di questa funzione è creare una lista che rappresenti il calendario del mese, posizionando i giorni corretti nei rispettivi giorni della settimana e riempiendo gli spazi vuoti (all'inizio e alla fine) con valori null per completare la visualizzazione settimanale. La funzione inizia creando un nuovo ArrayList vuoto chiamato 'days', che verrà popolato con i giorni del mese. Viene determinato il primo giorno del mese, calcolato il numero di giorni nel mese corrente e individuato il giorno della settimana del primo giorno del mese (tramite un valore numerico che indica il giorno della settimana). Successivamente, vengono aggiunti valori null per riempire gli spazi vuoti all'inizio, in modo da posizionare il primo giorno del mese nel giorno corretto della settimana. Poi, vengono aggiunti tutti i giorni del mese. Infine, vengono nuovamente aggiunti valori null per riempire gli spazi vuoti alla fine, assicurando che la lista abbia una lunghezza che sia un multiplo del numero di giorni della settimana, completando così la visualizzazione settimanale.

- ***updateEventsUI*** per la classe *ConsolleView* progettata per aggiornare l'interfaccia utente con gli eventi attuali e futuri. Inizia recuperando tutti gli eventi salvati chiamando il metodo controller.getAllEventToFile(). Se ci sono eventi presenti, la funzione filtra quelli odierni utilizzando il metodo filterEventsByDate e successivamente filtra gli eventi futuri con il metodo filterFutureEvents. Il metodo filterEventsByDate prende in input l'insieme di tutti gli eventi e filtra quelli che si svolgono nella data odierna. Questi eventi vengono raccolti in un TreeSet ordinato utilizzando un ComparatorEvents personalizzato. Il metodo filterFutureEvents

prende in input l'insieme di tutti gli eventi e l'insieme degli eventi odierni. Utilizza un Set di identificatori degli eventi odierni per escludere questi eventi dal filtraggio successivo. Filtra quindi gli eventi futuri, raccogliendoli in un TreeSet ordinato. Se ci sono eventi futuri, questi vengono aggiunti alla lista dei modelli di eventi con una separazione visiva e un'intestazione "Prossimi eventi", seguiti dalle informazioni dettagliate di ciascun evento futuro.

4.2.2 Alessandro Scodavolpe

- *Stream:* impiegati per gestire l'attraversamento della lista obbiettivi per l'istanziamento di elementi grafici in ConsolleObjectiveView, oltre per l'aggiornamento della lista nelle interfacce grafiche relative, consentendo una scrittura del codice snella e leggibile.

- *Optional:* costrutto utilizzato per gestire i valori restituiti dalle operazioni di ricerca su ObjectiveList in ConsoleController, durante la costruzione dell'interfaccia grafica relativa agli obbiettivi.

- *Exception:* molto utili per ObjectiveImpl, ServicesView, ObjectiveView e ForecastView poichè hanno facilitato la validazione dei parametri di input immessi dall'utente tramite la GUI, oltre che per la gestione tramite finestre di dialogo delle chiamate a ConsoleController.

Principali algoritmi realizzati:

- **savingForecast** implementato per la classe ObjectiveImpl, che consente di calcolare l'importo che si dovrebbe versare per raggiungere la soglia di risparmio voluta, dati una certa frequenza (il numero di mesi che separa i versamenti) e un determinato periodo di tempo. Il primo passo di questo algoritmo è calcolare il periodo di tempo stabilito, esprimendolo sia in anni (dodicesimi) che in mesi. Viene poi eseguito un controllo dei dati di input per evitare che si forniscano valori non coerenti (ad esempio, un numero di mesi tra i versamenti superiore al periodo stabiliito, oppure valori negativi o nulli).
  In caso gli input non vengano validati, viene sollevata un'eccezione di tipo IllegalInput, segnalando la non correttezza dei parametri. In caso contrario, viene controllato che la frequenza sia inferiore a 12 mesi per poter calcolare il numero di volte in cui verrebbero effettuati versamenti in un singolo anno utilizzando il periodo espresso in dodicesimi; se invece la frequenza è superiore o uguale ai 12 mesi significa che viene effettuato un solo versamento all'anno o meno, per cui si utilizza il periodo espresso in anni per il calcolo.
  Viene effettuato un arrotondamento per difetto sfruttando la funzione *floor* della libreria *Math* di *java.lang* che restituisce il maggior intero inferiore al valore double fornito come parametro. Successivamente in caso si sia scelto (mediante parametro boolean passato al metodo) di tenere conto del saldo già presente nell'obbiettivo, viene eseguito il calcolo del bilancio residuo , in caso contrario

non viene calcolato il bilancio residuo e si procede con l'operazione finale in cui si divide l'ammontare complessivo da raggiungere per il totale di mensilità in cui si effettuerà il versamento. Questo valore verrà poi restituito.

- **doOperation** per la classe ServicesView, il quale si occupa di effettuare le opportune chiamate al metodo updateConto di ConsoleController.
  Per farlo viene prima eseguita una validazione dell'ammontare inserito dall'utente, evitando che vengano inseriti valori negativi o nulli e sollevando un eccezione di tipo IllegalOperationException in caso affermativo.
  Nel passo successivo la funzione si occupa tramite una switch di distinguere tra operazioni svolte per l'Account o per ObjectiveImpl, interfacciandosi con ConsoleController e fornendo i giusti parametri in input per poter processare correttamente il pagamento (ammontare, causale, tipo di movimento, tipo di operazione).
  In caso l'operazione riguardi un'istanza di ObjectiveImpl, viene svolto un controllo sulla presenza di quest'ultimo nella lista obbiettivi di ConsoleController e in caso negativo l'operazione viene annullata e viene mostrata una finestra di dialogo per indicare l'errore.