Money Expense

Documento di specifica dell'architettura del sistema

Scopo del documento

Il seguente documento si presta a specificare i dettagli dell'architettura interna del sistema mediante l'utilizzo di diagrammi di classi. Per questa parte il medium di comunicazione sarà il linguaggio UML.
Rispetto ai documenti **D1-G09** e **D2-G09** verranno ulteriormente dettagliati i vari componenti specificati tramite i requisiti funzionali di Money Expense.
Per finire si impiegherà il codice in OCL (Object Constraint Language) per stabilire alcune condizioni a livello logico che dovranno essere rispettate dagli attributi e dalle operazioni delle classi.

Diagramma delle classi

Ogni classe identifica un nome, una serie di attributi che rappresentano i dati gestiti dalla classe stessa, e un elenco di metodi che definiscono le operazioni previste all'interno della classe. Inoltre, le classi possono essere associate ad altre classi, permettendo di specificare le relazioni tra di loro.
Questo permette di comprendere il comportamento del sistema in questione a livello implementativo.
Nelle prossime pagine vengono rappresentati i diagrammi delle classi principali del nostro sistema.
Potrebbe essere necessario introdurre ulteriori classi nel documento per una spiegazione più approfondita di alcune parti. In tal caso, tali classi saranno rappresentate nei diagrammi con un contorno tratteggiato per evidenziare che verranno discusse e approfondite in dettaglio nelle fasi successive.

Classi ausiliarie

Per rappresentare al meglio le varie classi individuate all'interno del progetto, facciamo uso delle classi ausiliarie.
Queste ultime ci permettono di definire a livello globale tipi e dati necessari per ottenere una semantica consistente. Inoltre, in caso di eventuali modifiche consentono di apportare alterazioni alla logica del codice in maniera rapida ed efficiente, riducendo così i tempi necessari per il refactoring.
Forniamo quindi di seguito, la classe **Date**, la quale indica una data, specificando giorno, mese e anno. Ci servirà ad indicare il giorno in cui una data transazione è stata registrata oppure il giorno di inizio e fine blocco utente.
Introduciamo anche la classe **Money**, la quale rappresenta un importo di denaro e include il valore dell'importo (e.g. 10) e la valuta di riferimento (e.g. USD, EUR, ...).
In questo modo le transazioni e i movimenti multi-valuta possono essere comodamente effettuati sulla piattaforma.

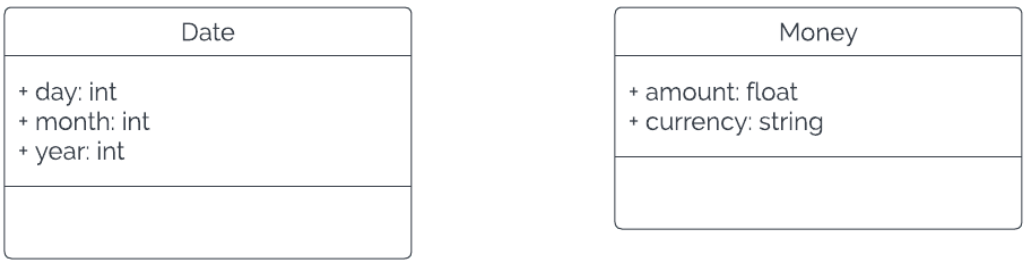


Figura 1. Classi ausiliarie

Classe dell'utente

Con riferimento al *context diagram*, specifichiamo ora tutte le informazioni di un determinato utente.
Dopo la revisione del *context diagram*, gli utenti che vengono ad interagire direttamente in piattaforma saranno i seguenti:

- **"User registrato"**
- **"Amministratore"**

Questi attori interni al nostro sistema sono estremamente simili.
Infatti, un "Amministratore" è un "Utente registrato" che può svolgere le stesse operazioni di un "Utente registrato" con l'unica differenza di essere in grado di bloccare determinati utenti il cui comportamento è ritenuto illecito o degno di essere sanzionato.
Per rappresentare questa situazione, possiamo creare una generica superclasse **user** che includa tutti gli attributi e le funzionalità comuni ad entrambi gli attori.
Essa sarà successivamente specializzata, generando la sottoclasse **admin**, che aggiungerà la funzionalità di blocco utente.

In particolare, **user** possiede i campi necessari per essere registrati (e.g. nome, cognome, email e password).
Aggiungiamo inoltre un attributo booleano che chiamiamo ***isBlocked***, che indica qualora l'utente fosse stato bloccato da un amministratore o meno.

Ogni user ha un attributo chiamato ***idAuth***.
In ***idAuth*** viene salvato un id di sessione in formato stringa nel caso in cui l'utente risulti autenticato alla piattaforma e NULL altrimenti. Questo renderà più conveniente il processo di distinzione di utenti autenticati rispetto a quelli non.

Un utente può eseguire una serie di operazioni relativamente al suo stato in piattaforma (e.g. ***login()***, ***logout()***, ***signUp()***). Queste funzionalità verranno illustrate più in dettaglio nei successivi diagrammi.

L'utente può anche aggiornare i propri dati e modificare la password mediante gli appositi metodi.

Infine, l'utente amministratore ha la possibilità di creare un **blocco_utente**, il quale comporterà due eventi:

1. Creazione e salvataggio del record **blocco_utente** nel database..
2. Modifica dello stato dell'utente in questione, viene impedito l'accesso alla piattaforma fino alla revoca del blocco.

Il blocco ha come attributi: l'utente a cui è riferito, la data di inizio, un'eventuale data di fine (se a NULL il blocco è permanente) e un motivo, dove si spiega cosa ha causato il blocco.
Per implementare alcune delle funzionalità sono necessarie delle classi non ancora specificate, indicate mediante una linea tratteggiata.

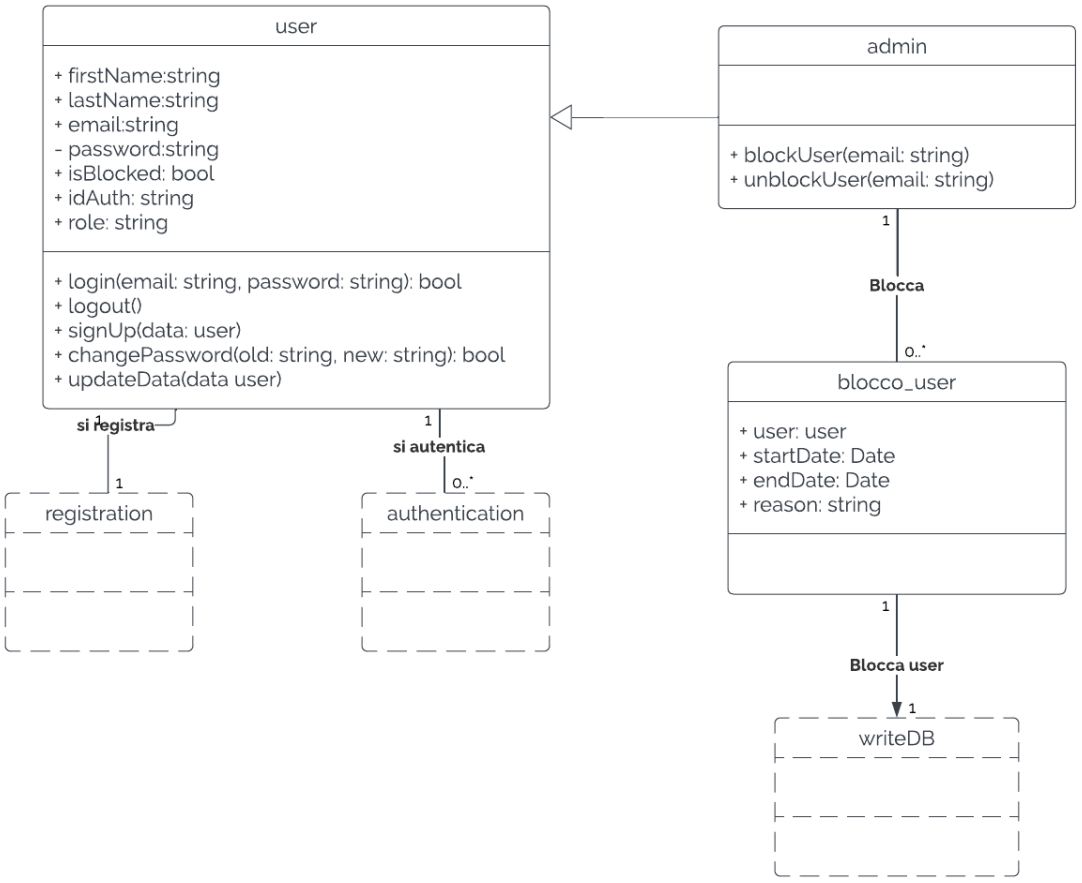


Figura 2. Classi per utenti e blocchi utente

Classe per gestione autenticazione e registrazione

Per poter sfruttare le funzionalità della piattaforma è necessario essere registrati e autenticati. Il processo di registrazione e autenticazione è gestito dal nostro sistema mediante apposite classi.

Poiché il nostro sistema conserva i dati utente e password in un database interno e sono gli utenti stessi a interagire con queste classi, presenteremo anche i loro elementi mediante linea tratteggiata.

Per gestire l'intero flow di autenticazione creiamo la classe **authentication**, la quale rappresenta i dati ricevuti dall'utente per effettuare l'accesso alla piattaforma. Eventualmente, i dati possono contenere anche una variabile booleana che indica se l'utente richiede l'autenticazione a due fattori. Se segnata a true, verrà effettuato anche un controllo mediante SMS.

Si procede quindi a interrogare il database e una volta ricevuto l'esito del tentativo d'accesso, questo verrà salvato nella classe stessa.

Un'istanza di questa classe è creata ogni qual volta un utente utilizzi il metodo ***login()***. Questa classe ha come molteplicità uno a molti, in quanto un utente può autenticarsi più volte in piattaforma (e.g. scadenza sessione).

Per la registrazione dell'utente alla piattaforma è stata ideata la classe **registration**, la quale contiene i dati necessari alla creazione dell'utente.
È presente inoltre un campo esito o ***outcome*** che indica il risultato del tentativo di registrazione.
Questo viene determinato svolgendo dei controlli locali sui dati e verificando nel database l'univocità dell'email inserita.

Un'istanza di questa classe è creata ogni qual volta un utente utilizza il metodo ***signUp()***. Questa classe ha molteplicità 1 ad 1, in quanto un utente può registrarsi una sola volta in piattaforma. In caso venisse revocata l'iscrizione e si procedesse di nuovo alla registrazione, l'utente generato e la registrazione sarebbero diverse.

Entrambe le classi espongono una serie di API pubbliche ed alcune *utility functions*, come il controllo della password forte, che tornano utili in diversi controlli che sono necessari (e più tardi spiegati nella parte di documento con OCL).

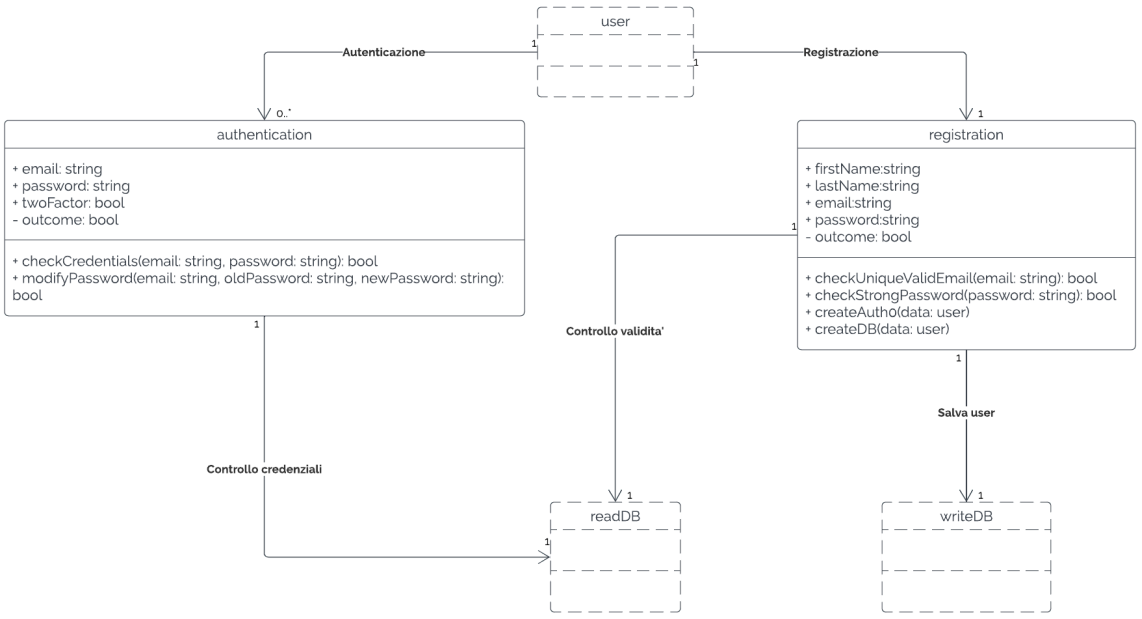


Figura 3. Classi per autenticazione e registrazione

Classe lettura database

Per poter effettuare molte delle operazioni sui dati è necessario prima poterli estrarre dal database (e.g. contesto dell'autenticazione).

A questo punto creiamo un'apposita classe **readDB** che si occuperà di tale compito.
La suddetta classe non presenta attributi, ma dispone di un ampio elenco di funzionalità rese disponibili per l'utilizzo esterno e per la ricezione dei dati necessari.
.
In particolare, le seguenti sono le classi che andranno ad interagire con **readDB**, richiedendo l'accesso ai dati memorizzati nel database:

- registration
- authentication
- transaction
- wallet
- category
- budget

In aggiunta, introduciamo una classe **reader** il cui obiettivo è generalizzare le diverse classi che interagiscono con la lettura del database (**readDB**).

La scelta di creare questa classe garantisce, anche in caso di future modifiche, flessibilità, robustezza e astrazione del codice.

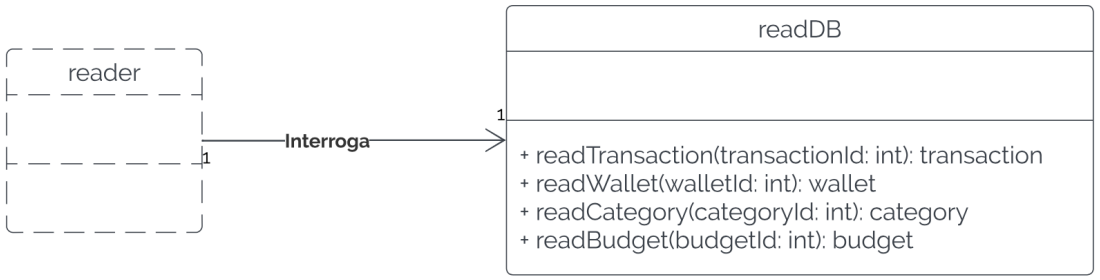


Figura 4. Classe per lettura database

Classe scrittura database

Per poter effettuare modifiche ai propri dati è necessario svolgere delle operazioni di scrittura al database.

Per questo motivo creiamo la classe **writeDB** che si occupa di scrivere i dati necessari all'interno del database ad ogni chiamata da parte di altre classi.

Anche questa classe non ha alcun attributo ma fornisce varie funzionalità utili per altre classi.
In particolare, le seguenti sono le classi che andranno ad interagire con **writeDB**, richiedendo la scrittura dei dati memorizzati nel database:

- registration
- transaction
- wallet
- category
- budget

Come prima, introduciamo una classe **writer** che andrà a generalizzare le diverse classi che interagiscono con la scrittura sul database (**writeDB**).

In generale, quello che la classe permette di fare, è la creazione, modifica o eliminazione dei dati interni al database.

La scelta di creare questa classe garantisce, anche in caso di future modifiche, flessibilità, robustezza e astrazione del codice.

Figura 5. Classe per scrittura database

Classe transaction

Descriviamo ora la classe che si occupa di gestire le transazioni generate dall'utente sulla piattaforma.

Per fare ciò specifichiamo la classe **transaction**.

Una transazione ha una data in cui è stata eseguita, la categoria a cui appartiene, il wallet da cui è stata eseguita, l'importo di denaro che questa transazione ha spostato, la tipologia di transazione e una descrizione.

Questa classe ha come unico metodo la modifica dei dati della transazione.
Potrebbe riscontrarsi utile nei casi in cui:

- l'utente decidesse di modificare qualche parametro.
- avesse commesso errori in fase di inserimento.

Per ottenere spiegazioni su classi o tipi ancora non citati (e.g. wallet) vedere sezioni successive del documento.

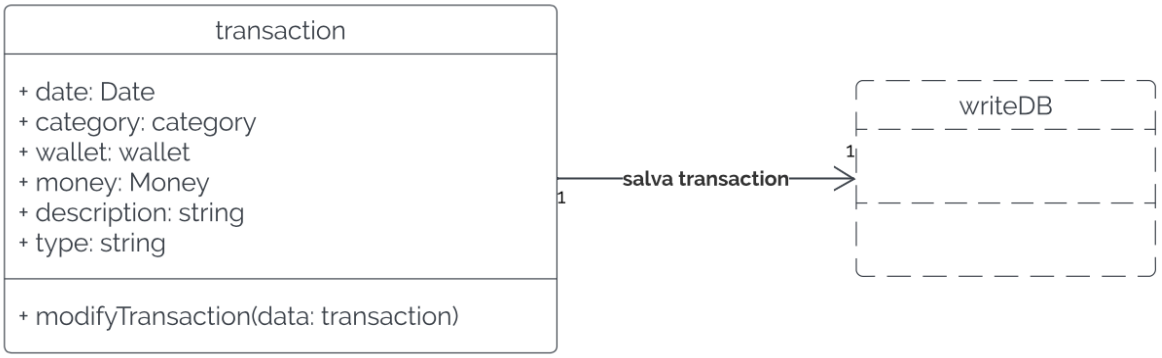


Figura 6. Classe per gestione transazioni

Classe wallet

Passiamo ora alla classe per la gestione dei wallet.

La classe **wallet** salverà al suo interno:

- Il nome del wallet, scelto dall'utente.
- Il quantitativo di denaro disponibile.
- Una breve descrizione testuale del motivo della sua esistenza o contenuto.
- Eventuali categorie ad esso associate.
- Un colore assegnato dall'utente.

Come nel caso della classe **transaction**, la classe **wallet** presenta un unico metodo di modifica dei dati del wallet.
Potrebbe tornare utile nei casi in cui:

- Venisse effettuata una transazione da parte dell'utente.
- L'utente decidesse di modificare direttamente la wallet.

Questa classe è infatti utilizzata dalla classe transazione per indicare da quale wallet è stato generato un movimento di capitale.

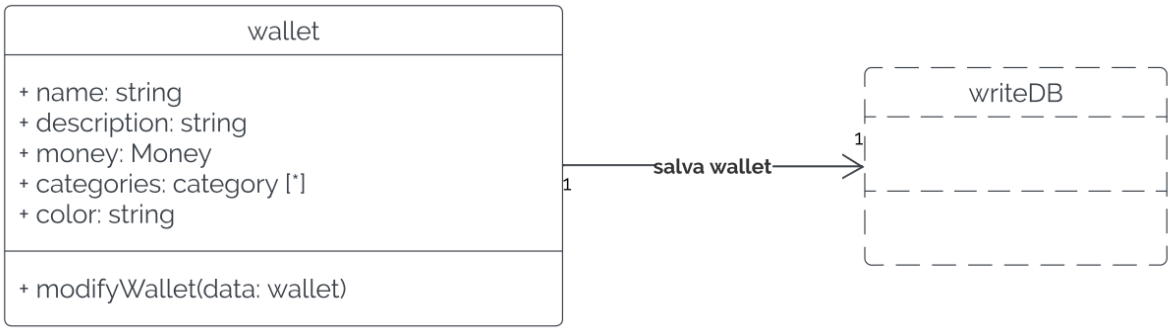


Figura 7. Classe per gestione wallet

Classe category

Passiamo ora a descrivere la classe **category**.

Questa classe si occupa di descrivere una categoria nel nostro sistema. Questa categoria avrà come attributi:

- Nome.
- Eventuali tag associati (utili in caso di applicazioni di filtri).
- Colore selezionato dall'utente.

Come funzionalità viene fornita la possibilità di modificare i dati relativi alla categoria.

La classe **category** viene utilizzata dalla classe **transaction**, per indicare la categoria per cui è avvenuta la transazione, dalla classe **wallet** e **budget**, qualora l'utente volesse associare una categoria a tali risorse.

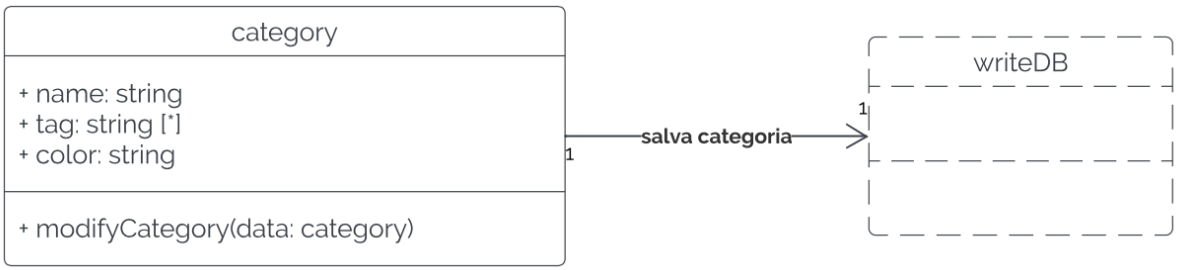


Figura 8. Classe per gestione categoria

Classe budget

Un'ulteriore funzionalità della piattaforma consiste nel creare e gestire dei budget di spesa.

Creiamo dunque l'apposita classe **budget**, la quale ha:

- Un nome associato.
- Una data di inizio e fine.
- Un capitale di partenza come budget.
- Il capitale attuale.
- La categoria associata.
- Un colore selezionato dall'utente.

Come funzionalità viene fornita la possibilità di modificare i dati relativi al budget.
È quindi necessario poter accedere alla classe per la scrittura del database.

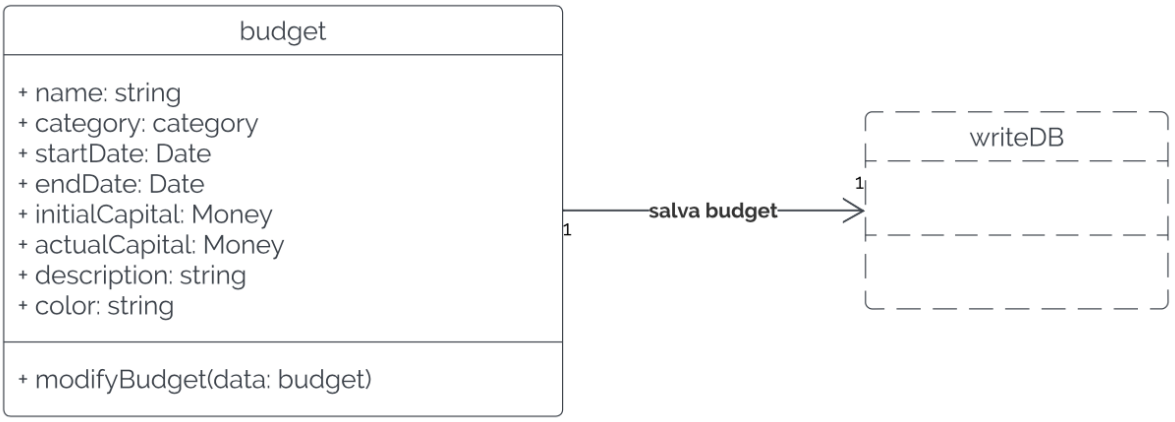


Figura 9. Classe per gestione budget

Classe ricerca per categoria e risultati ricerca

In piattaforma è possibile ricercare transazioni, wallet, budget e chat per categorie.

Di conseguenza, introduciamo una classe **search** che fornisce tutti gli oggetti precedentemente elencati in base ad una categoria selezionata.

Questi vengono riportati nella classe **searchResults**, la quale contiene quattro liste:

1. Una contenente le transazioni
2. Una contenente i wallet
3. Una contenente i vari budget
4. Una per le chat.

Questa classe contiene anche una data di tipo **Date**.
La classe **reasearchResults** ha come funzionalità quella di poter selezionare uno degli oggetti che ha trovato nella ricerca, in modo da consentire all'utente di vederli in maggiore dettaglio, ed eventualmente di modificarli.

Per garantire la correttezza della funzionalità è necessario che la classe **search** interagisca con il database.

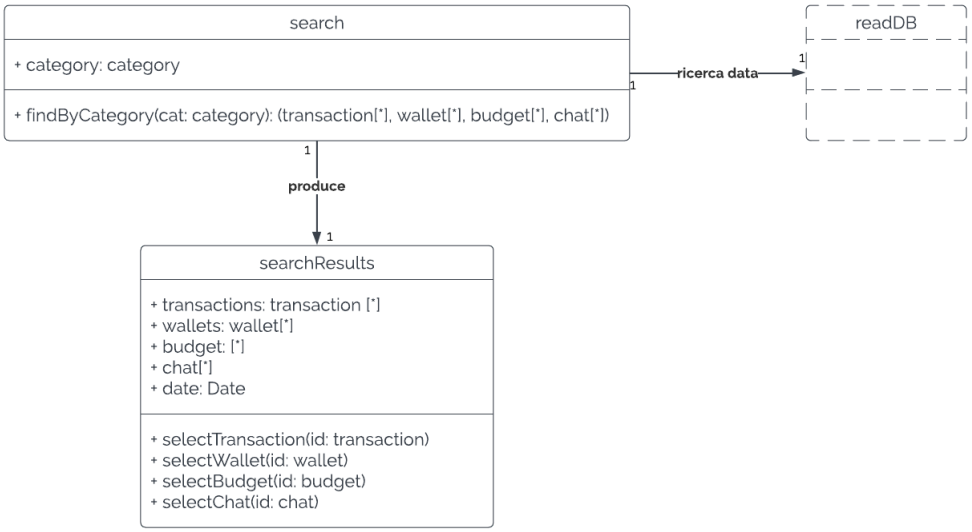


Figura 10. Classe per gestione ricerca per categoria

Classe chat

La piattaforma offre la possibilità di mantenere una chat virtuale con il ChatGPT bot.

Per gestire dunque questa funzionalità, creiamo l'apposita classe **chat**, la quale salva al suo interno un chat id, l'utente che ha interagito con il bot, il flow di messaggi, salvato mediante array di stringhe, la data in cui è avvenuta la conversazione, ed eventuali categorie, che l'utente potrà inserire, se in futuro vorrà accedervi più velocemente.
La classe espone la funzionalità di invio messaggio al bot. Una volta inviata la richiesta, il bot genererà una risposta e i messaggi verranno salvati sul database.

Per garantire la correttezza della funzionalità è necessario che la classe **chat** interagisca con il database.

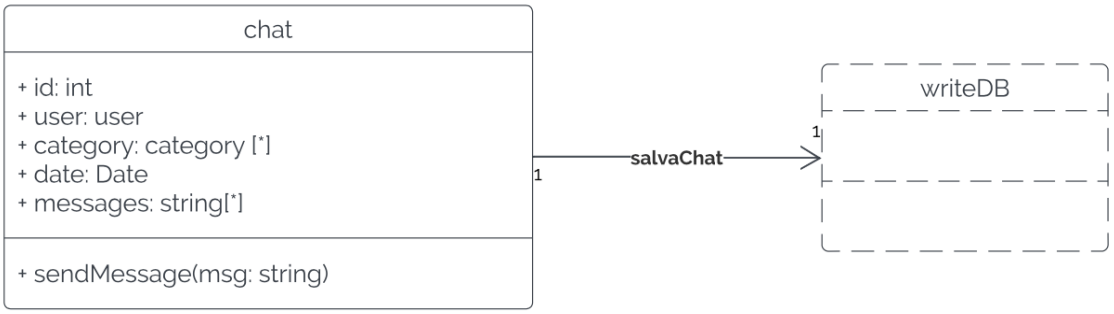


Figura 8. Classe per gestione chat utente

Diagramma della classe export data

Un'ulteriore funzionalità che la piattaforma concede ai propri utenti è l'esportazione di tutti i propri dati registrati in piattaforma, in un formato di dati universale, il *csv*. Per fare questo è stata ideata e realizzata la classe **export_data**.
La classe non ha alcun attributo di per sé che la contraddistingua. Tuttavia, ha un metodo che permette la lettura dei dati dal database, e li incolonna in maniera consistente. Ha quindi necessità di poter interrogare il database per svolgere la sua funzione.
Una volta elaborato questo file, sarà possibile per l'utente scaricarlo in locale.

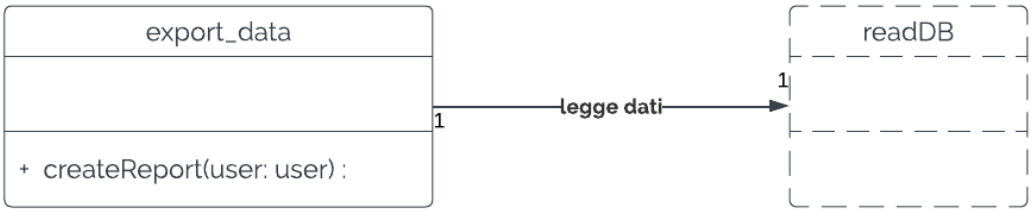
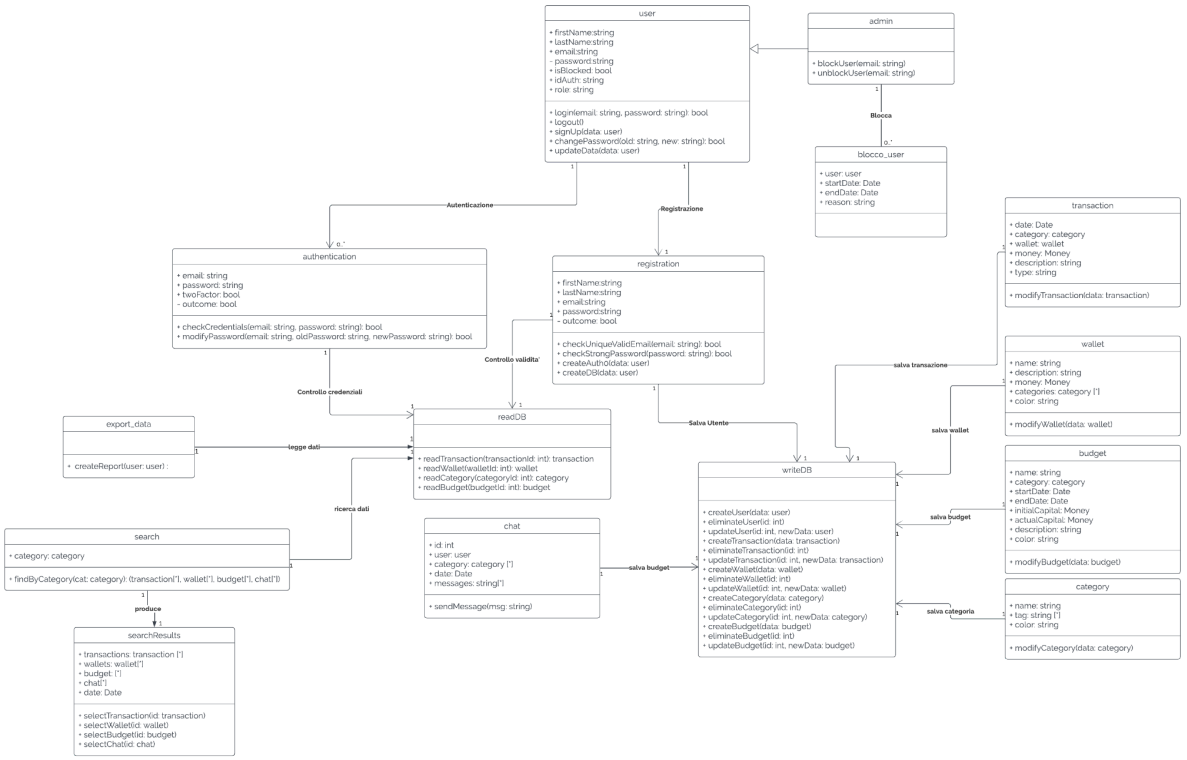


Figura 8. Classe per gestione esportazione dati

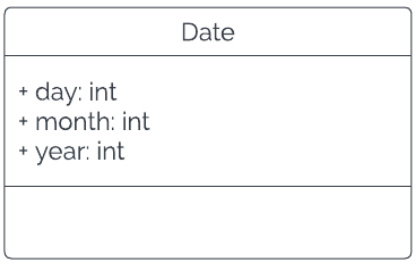
Diagramma delle classi completo

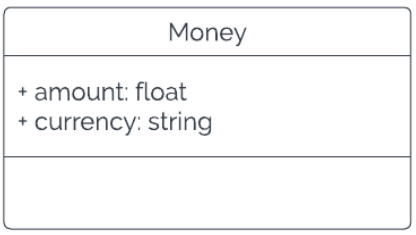
Di seguito viene riportato il diagramma delle classi completo, con tutte le classi precedentemente descritte e messe in relazione.

In questo modo si ha una visione di insieme della piattaforma, dei suoi componenti principali e delle funzionalità richieste in essa.



Con le classi ausiliarie:





OCL o Object Constraints Language

Nella seguente sezione del documento riportiamo invece tutte le limitazioni ed i vincoli del nostro progetto.
Questi ultimi sono necessari per mantenere la consistenza e uniformità sia nella piattaforma che a livello di implementazione del codice. Verrà utilizzato il linguaggio OCL che non solo permetterà di evitare ambiguità ma consentirà altresì di ottenere un'espressività superiore a quella di UML.

Vincoli OCL sulla classe Date

La classe **Date**, affinché sia valida e significativa, deve essere all'interno delle seguenti invarianti logiche:

context: Date
inv: year >= 1970 AND year <= 2200
inv: month > 0 AND month < 13
inv: day > 0 AND day < 32

Vincoli OCL sulla classe Money

La classe Money, affinché sia valida e significativa, deve essere all'interno delle seguenti invarianti logiche:

context: Money
inv: amount > 0 and amount < 100.000.000.000
inv: currency in ['USD', 'EUR', 'GBP', 'YUAN']

Vincoli OCL sulla classe user

Nella classe **user**, se vogliamo effettuare una nuova registrazione, è necessario che l'utente non sia già autenticato all'interno della piattaforma. Controlliamo quindi che l'attributo ***idAuth*** sia messo a NULL. Stesso discorso vale nel caso noi volessimo fare una login.
Discorso diametralmente opposto va invece svolto per i metodi ***logout()***, ***updateData()***, ***resetPassword()***, che richiedono specificamente all'utente di essere già autenticato in piattaforma, altrimenti l'esecuzione di tali metodi non avrebbe alcun senso.
Rappresentiamo dunque tali vincoli di seguito:

context: user
inv: role in ['normal', 'admin']

context: user::signUp(dati:user)
pre: idAuth = NULL

context: user::login(email: string, password: string)
pre: idAuth = NULL AND isBlock = false
post: idAuth!= NULL

context: user::logout()
pre: idAuth != NULL

context: user::updateData(data:user)
pre: idAuth != NULL

context: user::resetPassword(data:user)
pre: idAuth != NULL

Vincoli OCL sulla classe admin

Nella classe admin, purché questo utente possa svolgere le sue funzionalità in maniera corretta, deve essere loggato in piattaforma (effettuare operazione di login) e l'utente fornito deve non essere nullo.

context: admin::blockUser(emailBlock: string)
pre: login(email: string, password: string) = true AND emailBlock != NULL
context: admin::unblockUser(emailBlock: string)
pre: login(email: string, password: string) = true AND emailBlock != NULL

Vincoli OCL sulla classe authentication

Nella classe authentication, quando viene il momento di fare una modifica della password, va verificata l'email dell'utente, va controllato che la password vecchia corrisponda a quella precedente, e che quella nuova sia sufficientemente robusta e sicura (come descritto nel **RNF 1**).

context: authentication::modifyPassword(email: string, old_psw: string, new_psw: string) : bool
pre: old_psw = user[email].password AND checkStrongPassword(new_psw: string) = true

Vincoli OCL sulla classe registration

Nella classe registration, perché il tutto vada a buon fine, è necessaria un'email valida e non già memorizzata e una password forte. Per questo vengono fatti i seguenti controlli:

context: registration::createDB(new_user: user) : bool
pre: checkUniqueValidemail(email: string) = true

AND isStrongPassword(new_psw: string) = true
post: outcome = true

Vincoli OCL sulla classe transaction

Nella classe transaction, la descrizione deve avere dimensioni limitate, e la tipologia può essere di tipi limitati (income oppure expenses).

context: transaction
inv: description-> length() <= 200
inv: type == 'income' OR type == 'expenses'

Vincoli OCL sulla classe wallet

Nella classe wallet, affinché i dati siano utilizzabili, devono valere le seguenti invarianti:

context: wallet
inv: description-> length() <= 200
inv: color >= "#000000" AND color <= "#FFFFFF"

Vincoli OCL sulla classe budget

Nella classe budget, affinché i dati siano utilizzabili, devono valere le seguenti invarianti:

context: budget
inv: description-> length() <= 200
inv: color >= "#000000" AND color <= "#FFFFFF"
inv: start_date < end_date

Vincoli OCL sulla classe category

Nella classe category, affinché i dati siano utilizzabili, devono valere le seguenti invarianti:

context: category
inv: tag-> length() <= 10
inv: color >= "#000000" AND color <= "#FFFFFF"

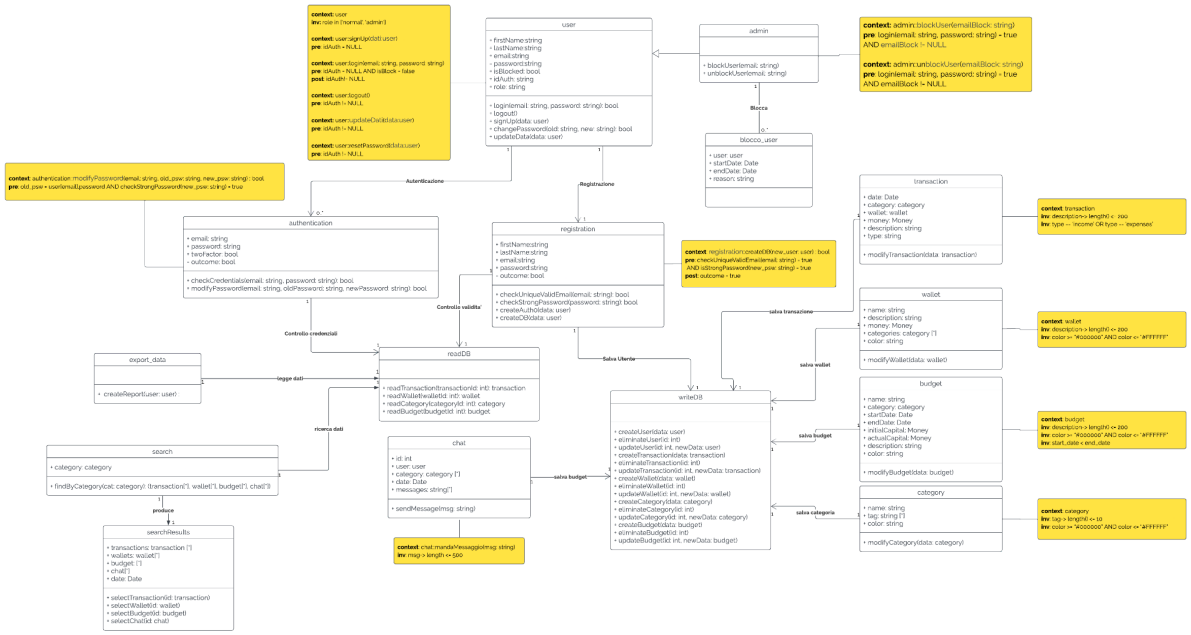
Vincoli OCL sulla classe chat

Nella classe chat, quando un messaggio viene inviato dall'utente, tale messaggio dovrà avere lunghezza inferiore ai 500 caratteri.

context: chat::sendMessage(msg: string);
inv: msg-> length <= 500

Diagramma delle classi con relativo OCL

Al termine di tale documento, inseriamo il diagramma delle classi mostrato precedentemente, con annesso anche i vari vincoli OCL prima elencati, mettendo quindi in evidenza tutte le varie regole, limiti ed invarianti.



Con le classi ausiliarie: