Dipartimento di Ingegneria e
Scienza dell'Informazione

Progetto:

Co.Inqui

Titolo del documento:

Documento di Architettura

Document Info:

Doc. Name	D3_Co.Inqui_Architettura		
Doc. Number	D3	Group number	G30
Description	Documento di architettura con uso di diagrammi delle classi e codice in OCL.		

SCOPO DEL DOCUMENTO

Il presente documento espone la definizione dell'architettura dell'applicazione Co.Inqui usando diagrammi delle classi scritti in UML (Unified Modeling Language) e codice in OCL (Object Constraint Language).

Mediante gli use case diagrams, il context e il component diagram presentati nel precedente documento (D2) andremo a definire in maniera più dettagliata e chiara l'architettura del sistema presentato fino ad ora.

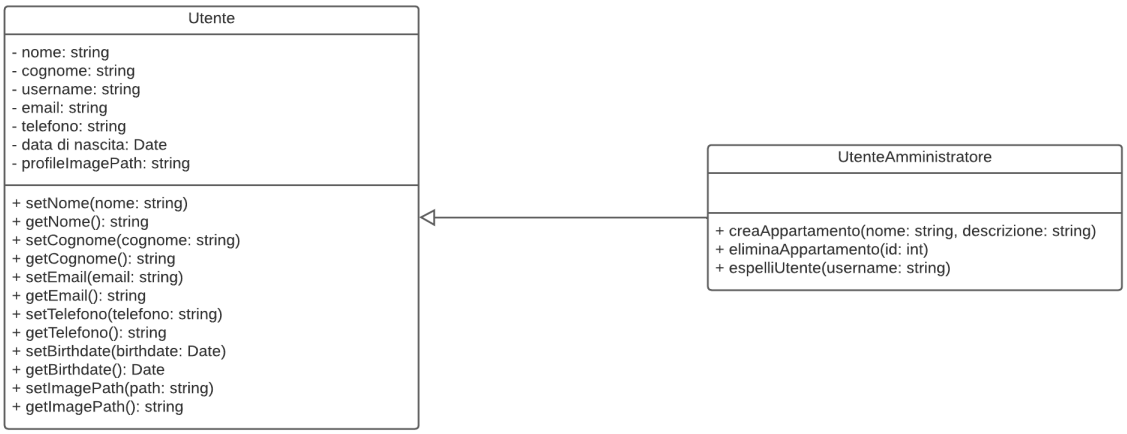
Illustreremo prima, tramite il class diagram, le classi che dovranno essere implementate e le varie associazioni tra di esse e, successivamente, la logica che regolamenta il comportamento del software, mediante la codifica in OCL.

DIAGRAMMA DELLE CLASSI

Nel presente capitolo vengono presentate le classi previste nell'ambito del progetto CO.Inqui. Ogni componente presente nel diagramma dei componenti, presentato nel documento precedente, diventa una o più classi. Tutte le classi individuate sono caratterizzate da un nome, una lista di attributi che identificano i dati gestiti dalla classe e una lista di metodi che definiscono le operazioni previste all'interno della classe. Ogni classe può essere anche associata ad altre classi e, tramite questa associazione, è possibile fornire informazioni su come le classi si relazionano tra loro.

UTENTE

Nei documenti precedenti è sempre stata evidenziata una differenza tra utente generico (autenticato e non) e utente amministratore. L'utente generico, una volta eseguito l'accesso al sistema, può utilizzare le funzionalità dell'applicazione quali accedere ad un appartamento, scrivere in bacheca, aggiungere Task alla ToDo list e tutte le altre funzionalità offerte dall'applicazione, mentre l'utente amministratore è un utente che ha la possibilità di eseguire delle azioni amministrative sugli appartamenti, perché è lui il creatore o perché il ruolo gli è stato ceduto.
La classe **UtenteAmministratore** è una sottoclasse di **Utente**, per questo eredita tutti gli attributi e i metodi di quest'ultima, ai quali vengono però aggiunte delle funzioni esclusive. Le due classi sono quindi associate e collegate tramite una generalizzazione.



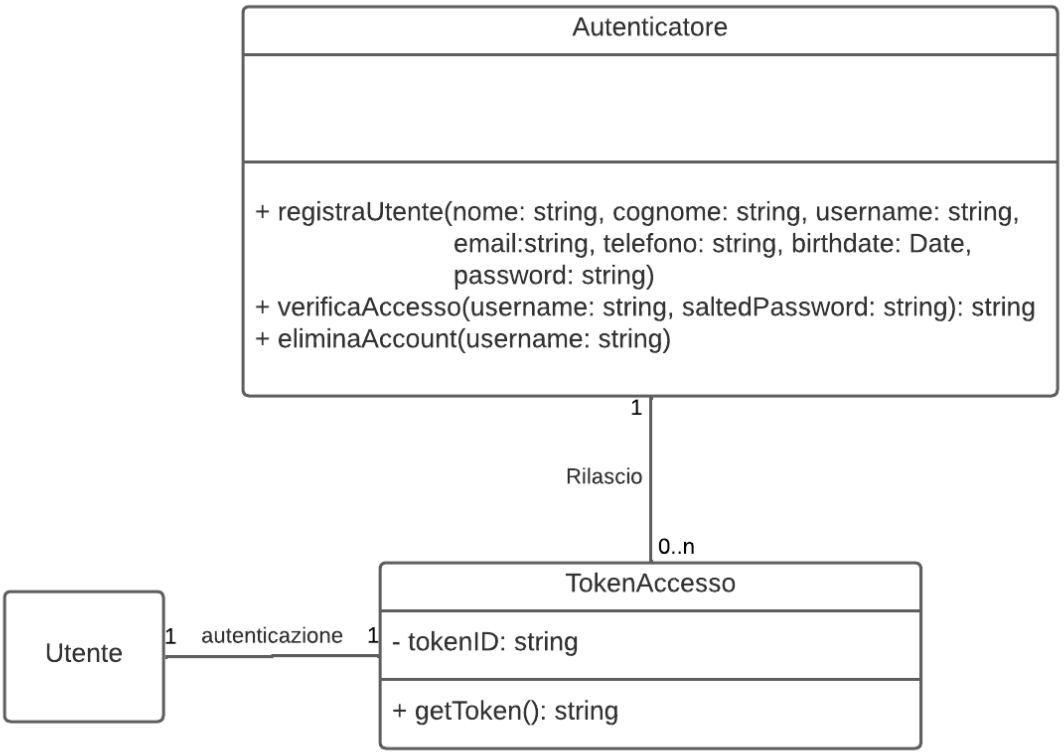
AUTENTICAZIONE

Per gestire il processo di autenticazione sono state identificate due classi ovvero **Autenticatore** e **TokenAccesso**.

La classe *Autenticatore* ha l'obiettivo di gestire le fasi di registrazione, accesso e rimozione di un account relativo all'applicazione.

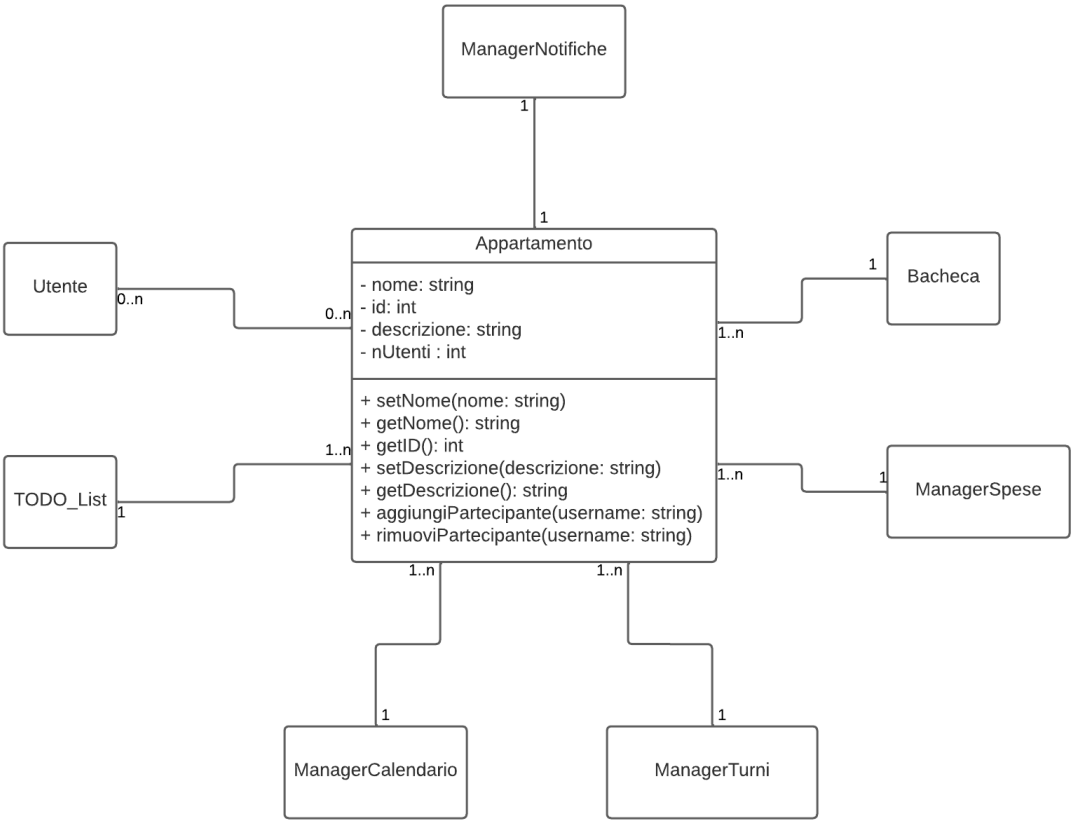
La funzione più importante tra queste è quella di *verificaAccesso* in quanto, una volta verificate le credenziali dell'utente che cerca di accedere all'applicazione, rilascia un *token di Accesso* che verrà poi implicitamente richiesto agli utenti ogni qual volta proveranno ad eseguire un'azione sull'applicazione per accertarsi della loro identità.

Di seguito vengono presentate nel dettaglio le suddette classi con rispettivi attributi e metodi.



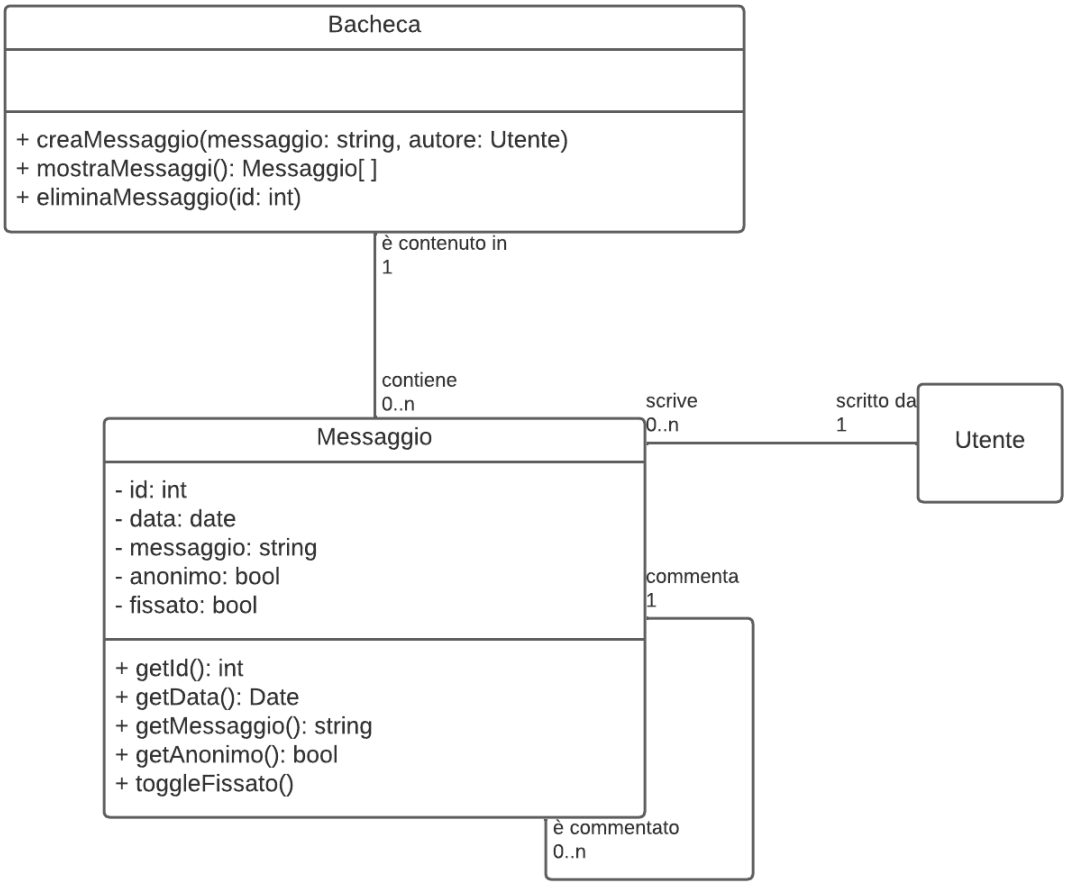
APPARTAMENTO

Dal component diagram notiamo che la componente “Homepage appartamento”, nonostante non svolga alcun compito, o funzione, specifico è di grande importanza, in quanto è collegata e serve ad interfacciarsi a tutte le altre componenti principali. Questa componente, insieme a quella chiamata “Gestione appartamenti”, sono state trasposte nel class diagram tramite un’unica classe **Appartamento**, la quale è associata a numerose classi. Essa ha l’obiettivo di gestire le informazioni relative all’appartamento virtuale quali il nome, l’id e l’aggiunta o rimozione dei partecipanti.



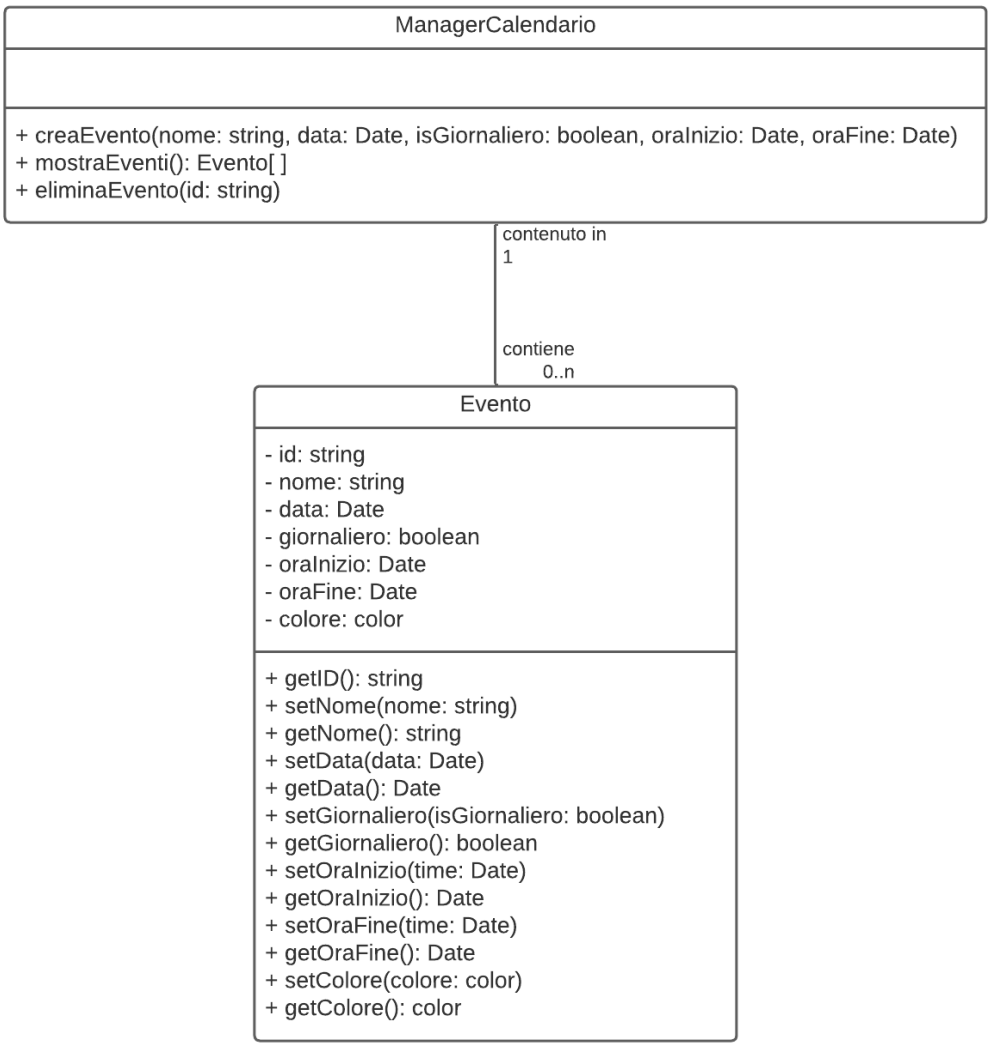
BACHECA

Per implementare la funzionalità della bacheca abbiamo creato, oltre alla classe **Bacheca**, una classe **Messaggio**.
La classe **Bacheca** serve a gestire la pagina omonima nella sua complessità, permette infatti di visualizzare i messaggi e crearne di nuovi o eliminarli, mentre la classe ausiliaria **Messaggio** si occupa delle caratteristiche specifiche di quest'entità. Quest'ultima classe è associata anche a sé stessa in quanto il sistema Co.Inqui implementa la possibilità di commentare i messaggi postati.



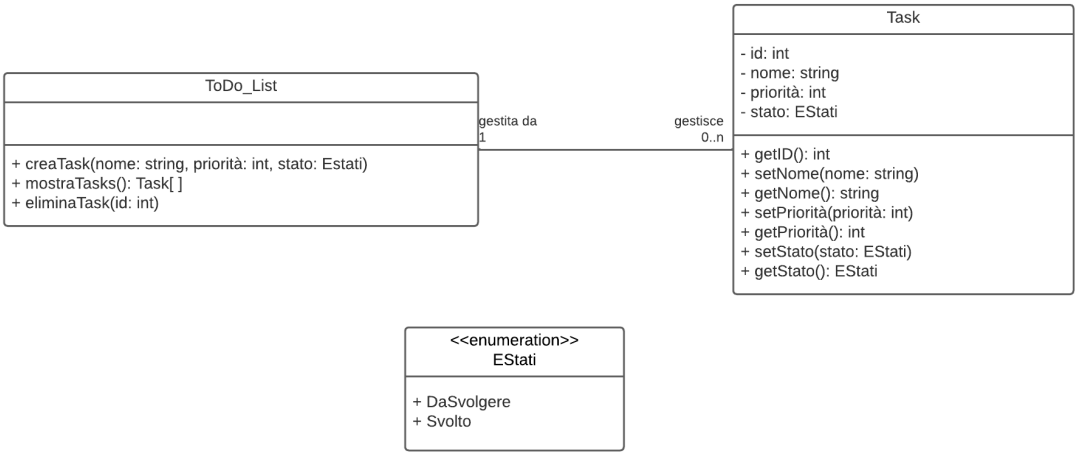
CALENDARIO

La componente “Calendario” presentata nel component diagram è stata esplicitata tramite l’uso di due classi ovvero **ManagerCalendario** e **Evento.**
La prima serve per gestire il calendario nella sua totalità, mostrando gli eventi già creati, con la possibilità di eliminarne alcuni o di crearne di nuovi.
La classe Evento, invece, concerne tutto ciò che riguarda l’evento nello specifico, come il nome, data e ora di inizio e fine, la possibilità che ricopra l’intera giornata o che sia ripetibile.



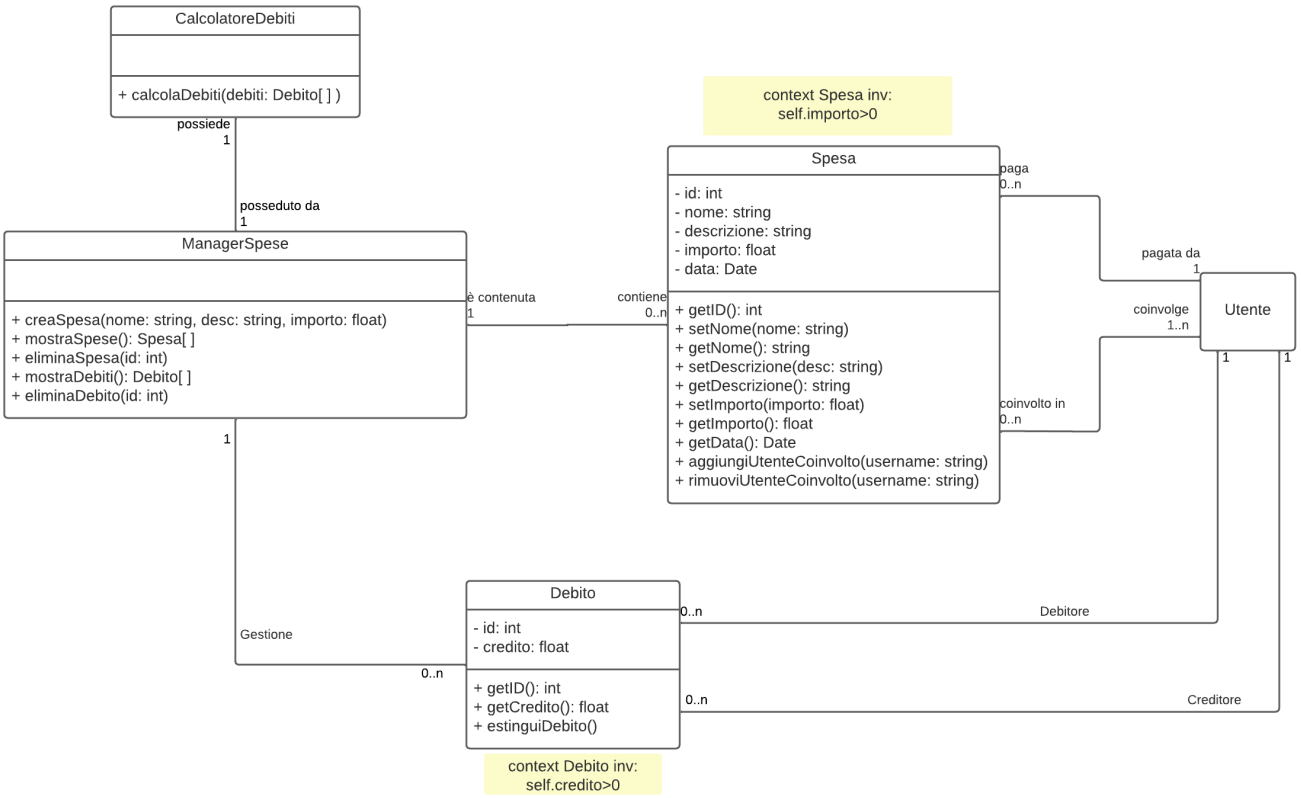
TO-DO LIST

La pagina e la componente “To-do list” vengono presentate nel class diagram tramite l’uso delle due classi **ToDo_List** e **Task**.
Analogamente a ciò che accade per il calendario e la bacheca, la prima serve per gestire la To-Do List nella sua interezza e complessità, mostrando le task, creandone di nuove o eliminandone di vecchie, mentre la classe Task serve per tutte le specificità dell’entità.
Abbiamo ritenuto necessario ricorrere all’uso di una enumerazione, chiamata **EStati,** che implementi gli stati possibili in cui le task si possono trovare.



SPESE

Le componenti “Gestione spese” e “Calcolatore debiti” presenti nel component diagram sono state trasposte nel class diagram mediante l’utilizzo di quattro classi associate tra loro e associate alla già nominata classe Utente.
Nello specifico, l’utente può creare una nuova istanza di spesa, i cui attributi sono definiti dalla classe **Spesa,** tutte le spese vengono gestite dalla classe **ManagerSpese** che mostra le spese sostenute, permette la creazione e la rimozione di spese, inoltre passa tutti gli importi alla classe **CalcolatoreDebiti**, che, come suggerisce il nome, calcola i debiti, ogni debito è caratterizzato da determinati parametri quali id e credito, che corrispondono agli attributi della classe **Debito.** Quest’ultima classe è associata all’utente che lo definisce come “Creditore” o “Debitore” a seconda che il credito sia positivo o negativo e serve per la gestione dei saldi tra gli utenti.

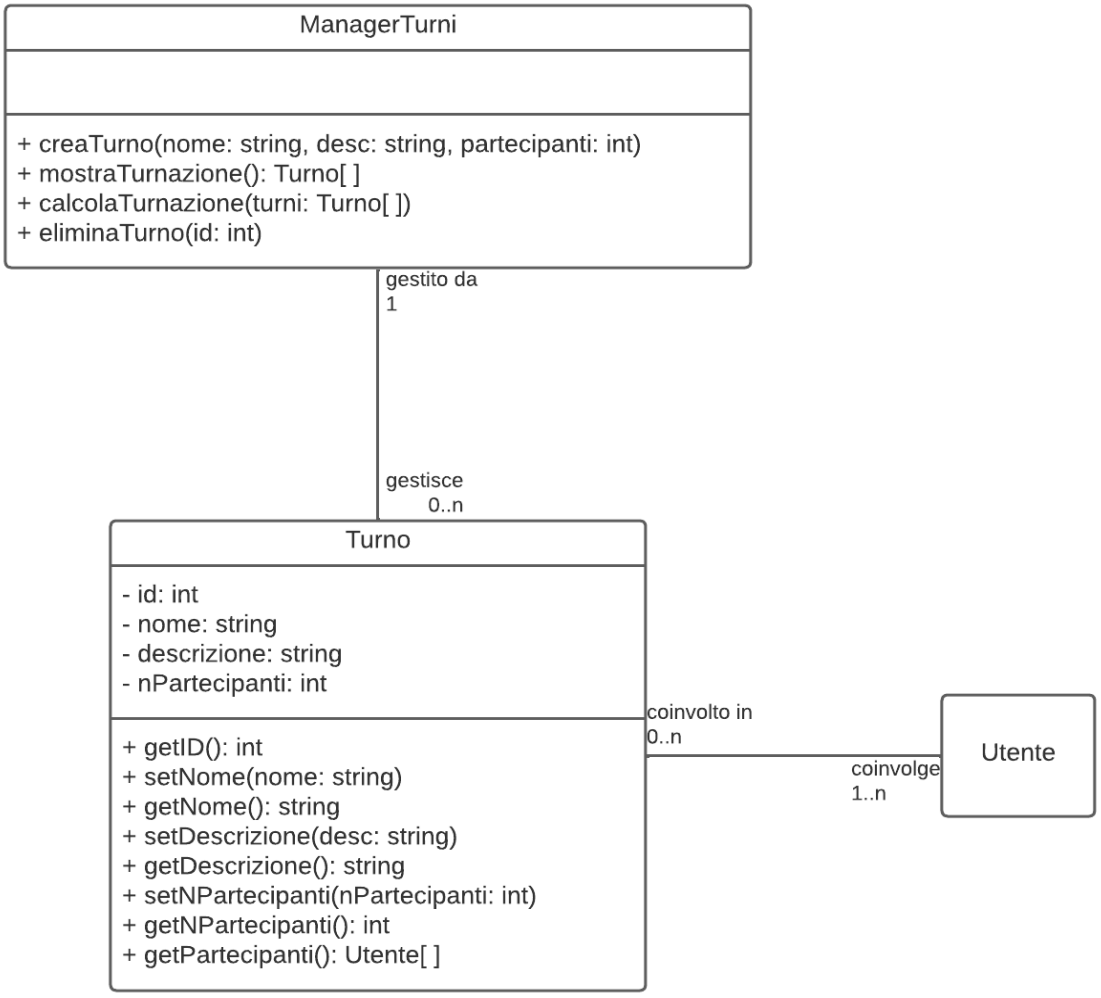


TURNI

Le due componenti “Gestione turni” e “Calcolatore turni” individuate nel component diagram vengono rappresentate nel nostro class diagram mediante l’uso di due classi tra loro associate: **Turno** e **ManagerTurni.**

La prima si occupa di tutti i parametri specifici di un’istanza di tipo turno, mentre la classe **ManagerTurni** si occupa della gestione dei vari turni, ne rende possibile la creazione di nuovi o l’eliminazione di vecchi.

Inoltre questa classe tramite il metodo calcolaTurnazione() si occupa direttamente anche dell’assegnazione delle persone ad ogni turno.



NOTIFICHE

L’invio delle notifiche non sarà gestito direttamente dal nostro software Co.Inqui, bensì ne è stata affidata la gestione ad un API esterna. Abbiamo definito, quindi, la classe **ManagerNotifiche** che si interfaccerà con la suddetta API per inviare le mail all’utente quando richiesto dal sistema.

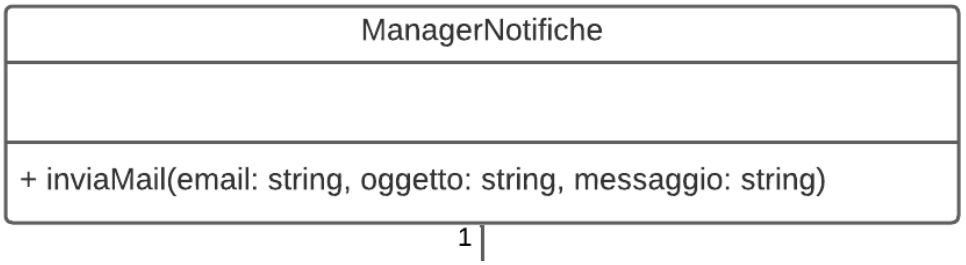
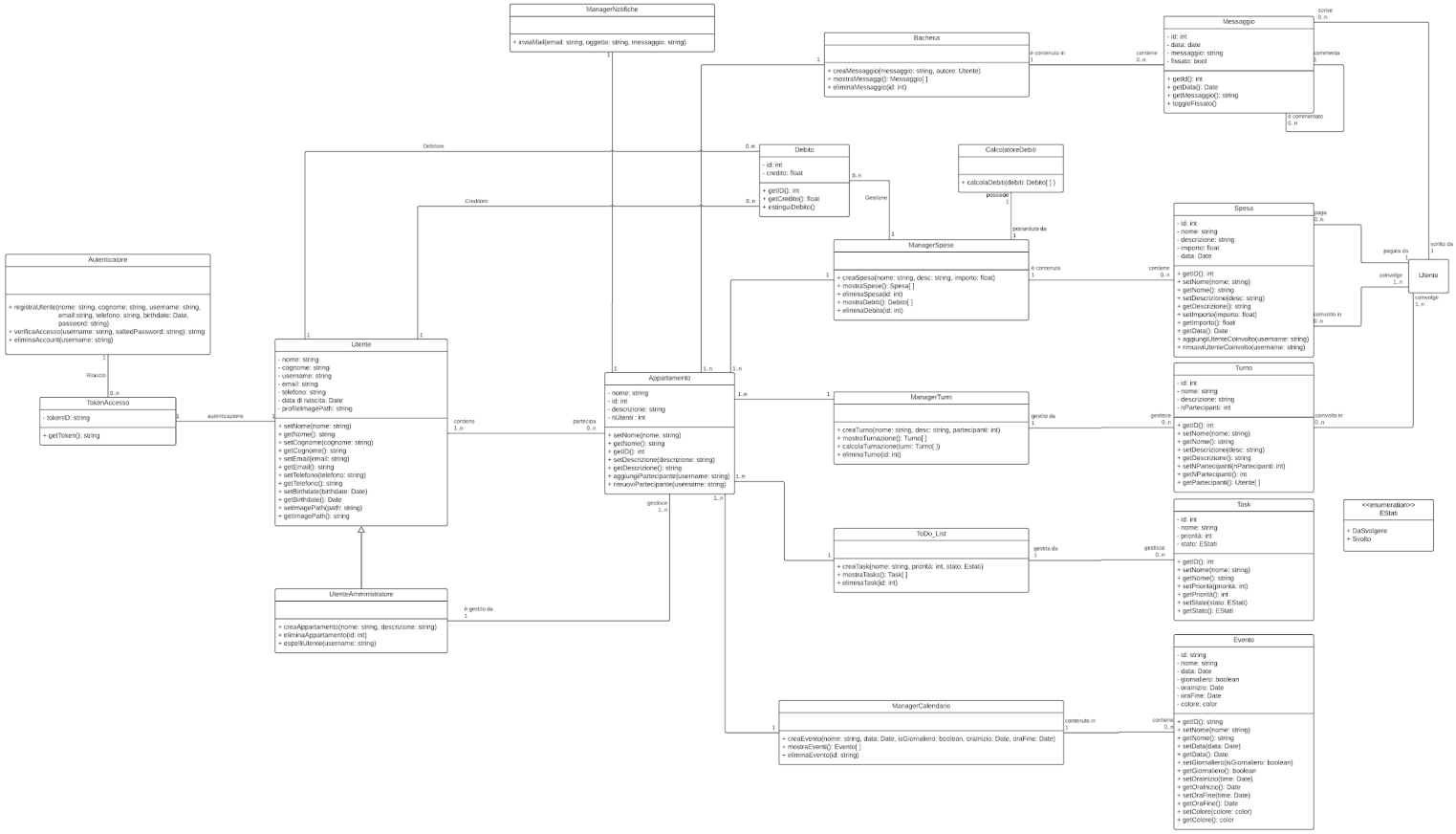


DIAGRAMMA DELLE CLASSI

Viene riportato di seguito il class diagram completo con tutte le classi presentate sopra.

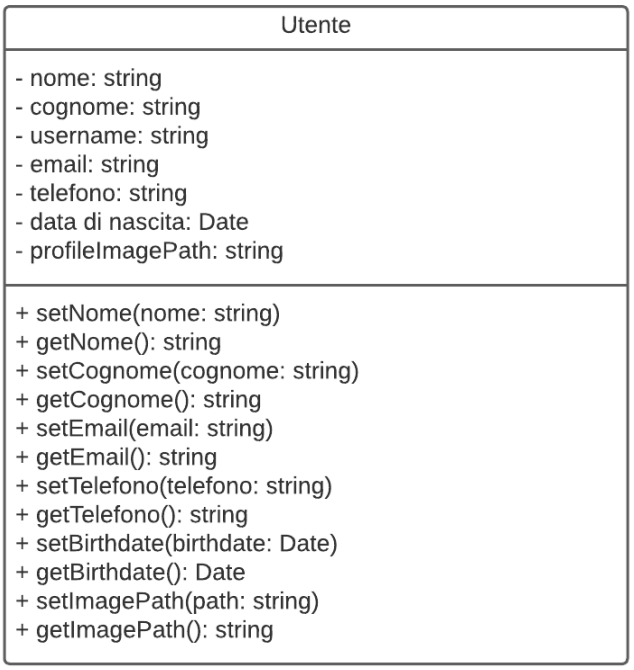


CODICE IN OBJECT CONSTRAINT LANGUAGE

In questa sezione, viene formalmente delineata la logica associata a specifiche operazioni di determinate classi. Questa logica è esposta attraverso l'utilizzo dell'Object Constraint Language (OCL), poiché i concetti in questione non possono essere espressi in alcun altro modo formale all'interno del contesto di UML.

LIMITAZIONE DATA DI NASCITA UTENTE

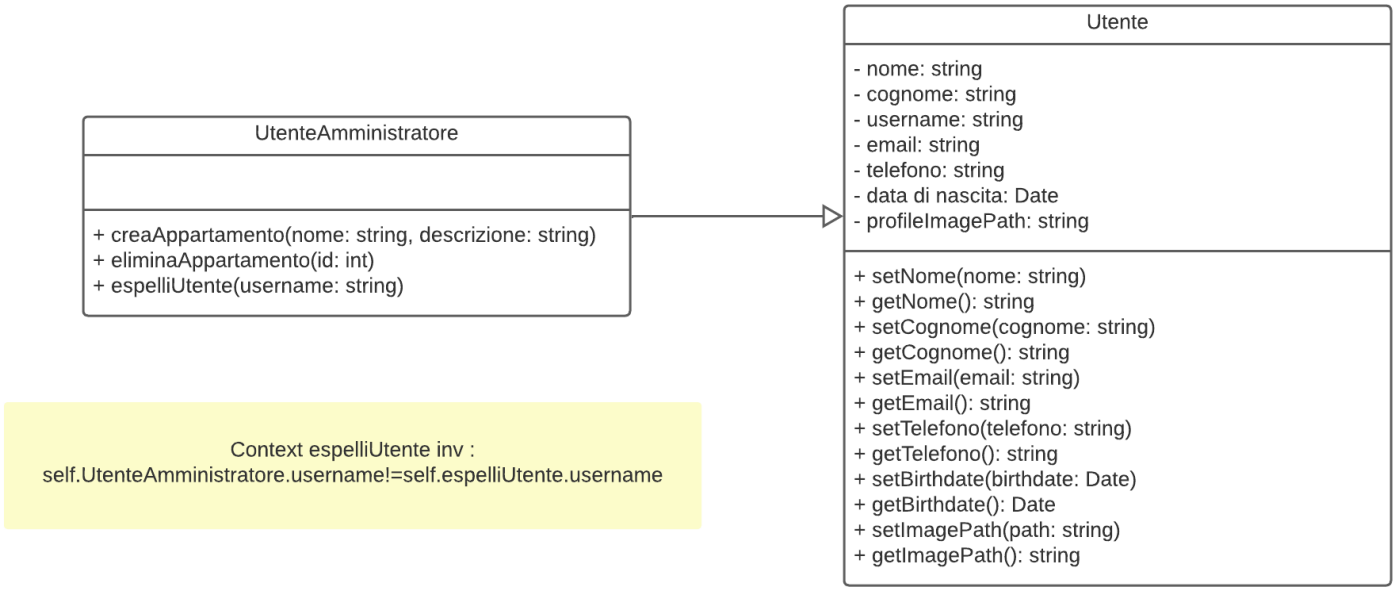
Ogni utente ha una data di nascita che deve per forza di cose essere minore della data di oggi.




```
Context Utente inv :
data di nascita <= TodayDate
```

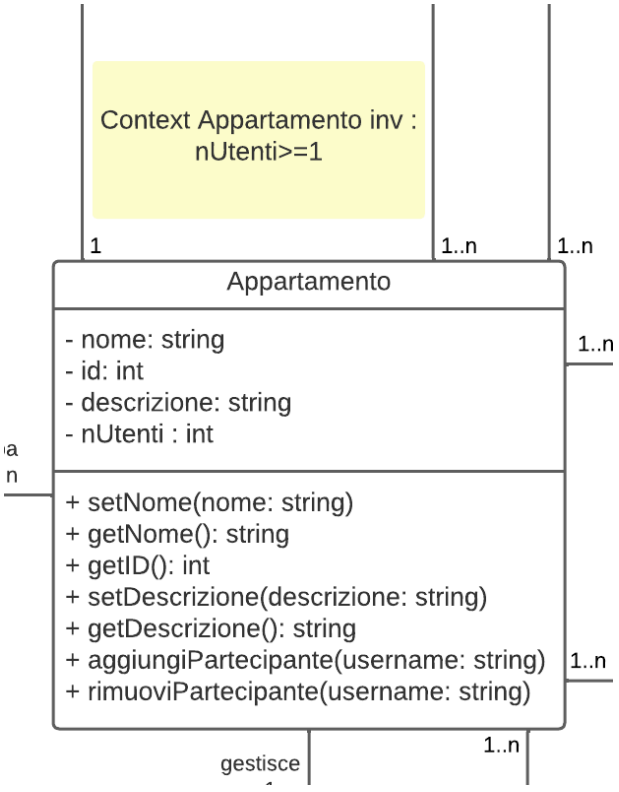
LIMITAZIONE ESPULSIONE UTENTI

Quando un utente admin decide di espellere un utente da un appartamento occorre verificare che l'username del soggetto da espellere non sia lo stesso dell'admin altrimenti si rischierebbe di avere un appartamento senza un admin.



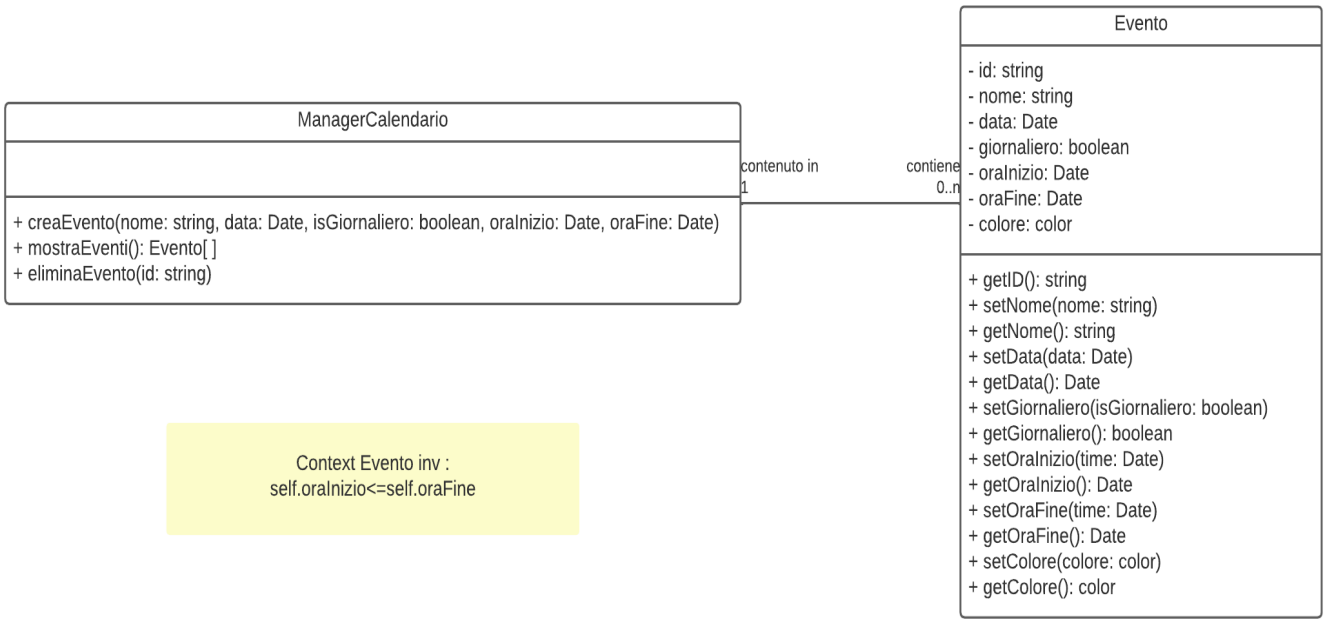
LIMITAZIONE UTENTI ALL'INTERNO DI OGNI SINGOLO APPARTAMENTO

Ogni appartamento deve avere al suo interno almeno un utente ovvero l'admin poichè un appartamento senza utenti non può esistere.



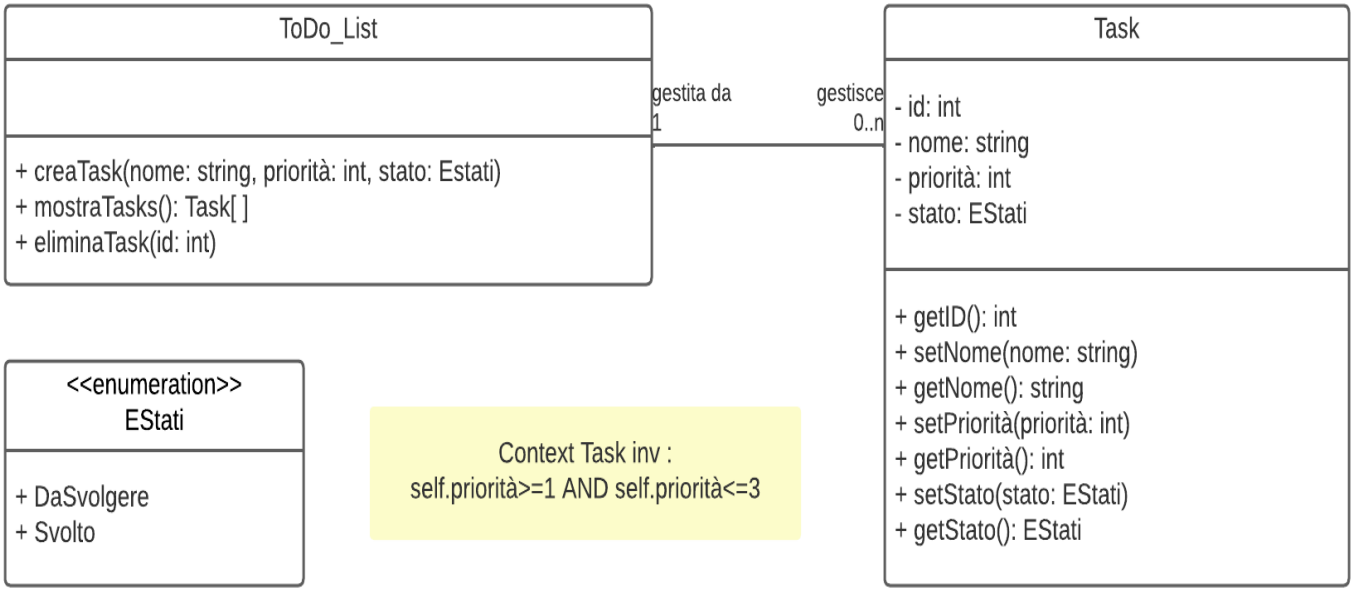
LIMITAZIONE ORARI EVENTI

Quando si va a creare un nuovo evento che andrà poi ad essere salvato anche su Google Calendar occorre verificare che l'orario di inizio sia minore o uguale all'orario di conclusione dell'evento.



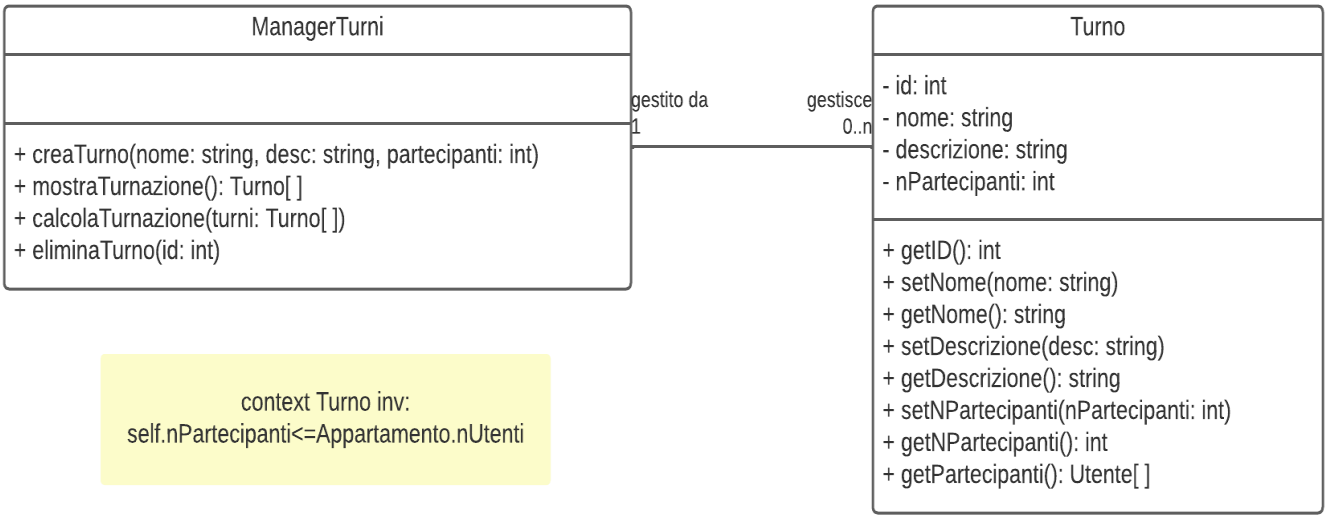
LIMITAZIONE PRIORITA' TASK

Ogni task ha un campo priorità che deve per forza essere un intero compreso tra 1 e 3 by design.



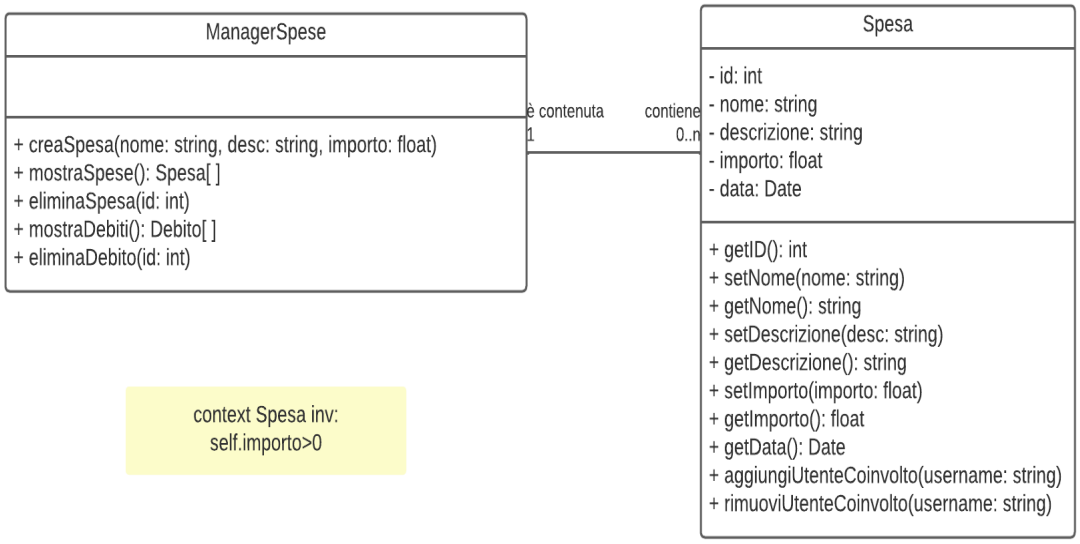
LIMITAZIONE NUMERO PARTECIPANTI TURNI

Quando si va a creare un nuovo turno occorre verificare che il numero di persone assegnate a tale turno sia minore o uguale al numero dei membri effettivi dell'appartamento.



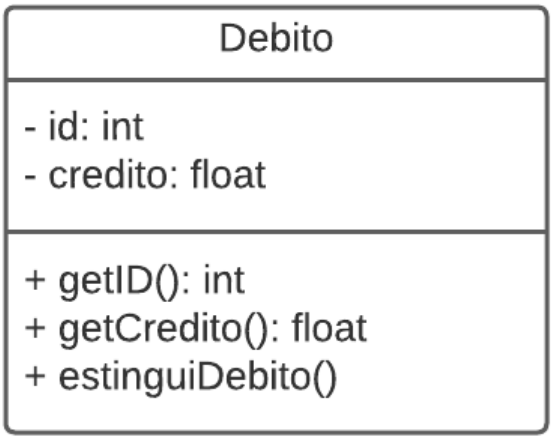
LIMITAZIONE NUMERO PARTECIPANTI TURNI

Ogni spesa deve avere un importo maggiore di zero poiché sarebbe impossibile avere una spesa negativa o uguale a 0 .



LIMITAZIONE VALORE DEL DEBITO

Ogni debito deve avere un valore maggiore di 0 poiché non può esistere un debito con valore negativo o uguale a 0.




```
context Debito inv:
self.credito>0
```

DIAGRAMMA DELLE CLASSI CON CODICE OCL

Per concludere riportiamo il diagramma delle classi con tutte le classi presentate fino ad ora e il codice OCL individuato.