Roomunity

DOCUMENTO DI ARCHITETTURA

Gruppo	**G34**	**Membri**	**Marco Pulze,** **Sara Boscardin,** **Luca Damaschetti**
Titolo Documento	**Documento di Architettura**		

SOMMARIO

DOCUMENTO DI ARCHITETTURA

0) SCOPO DEL DOCUMENTO

Il presente documento riporta la definizione dell'architettura del progetto RoomUnity, attraverso i diagrammi delle classi e codice.

Nel precedente documento è stato presentato il Diagramma degli Use Case, il Diagramma di Contesto e il Diagramma dei Componenti. Tenendo conto di questa progettazione, viene definita l'architettura del sistema dettagliando da un lato le classi che dovranno essere implementate a livello di codice e dall'altro la logica che regola il comportamento del software. Le classi vengono rappresentate tramite un Diagramma delle Classi in linguaggio UML (Unified Modeling Language).

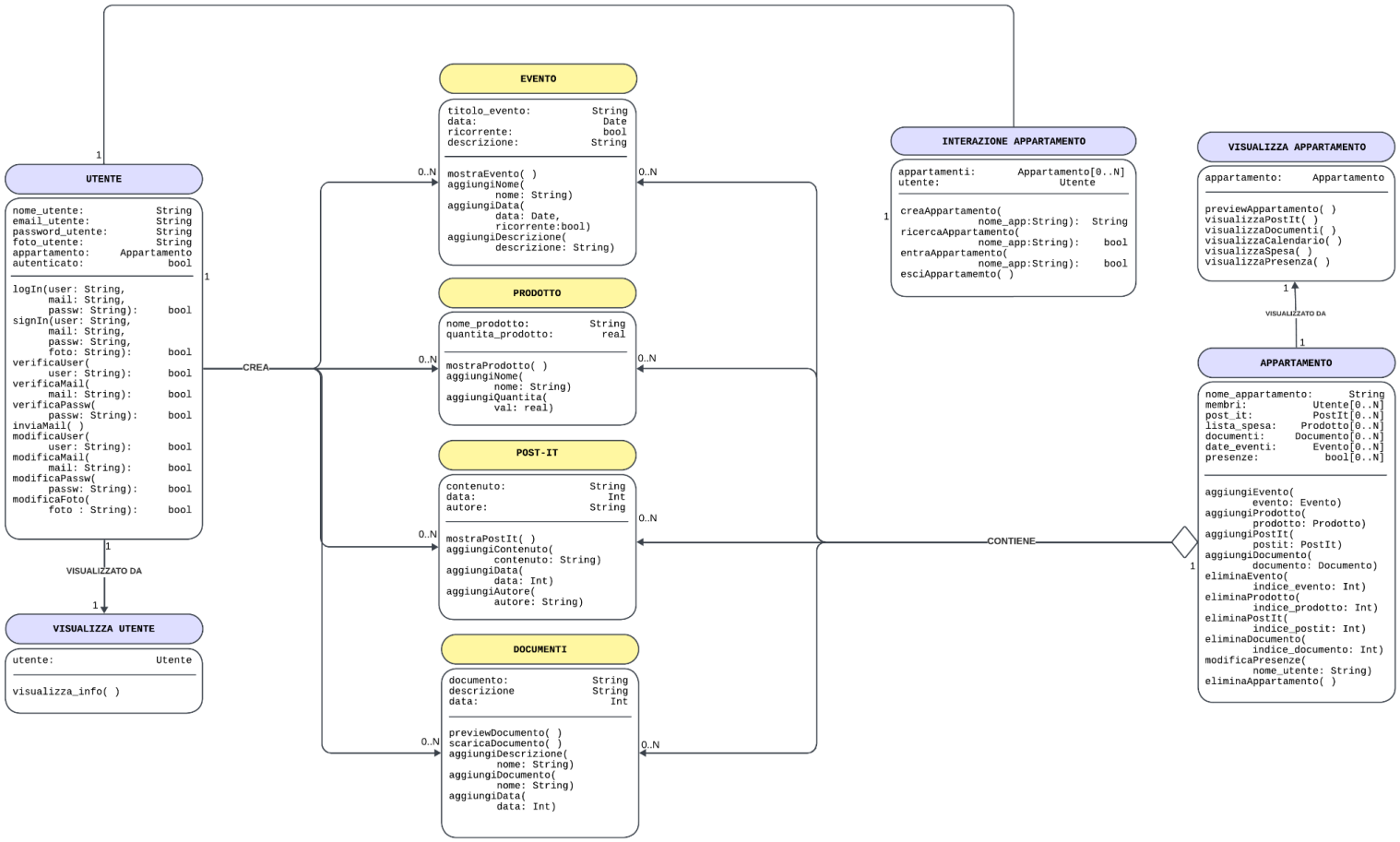
La logica viene descritta in OCL (Object Constraint Language), perché tali concetti non sono esprimibili in nessun altro modo formale nel contesto di UML.

1) DIAGRAMMA DELLE CLASSI

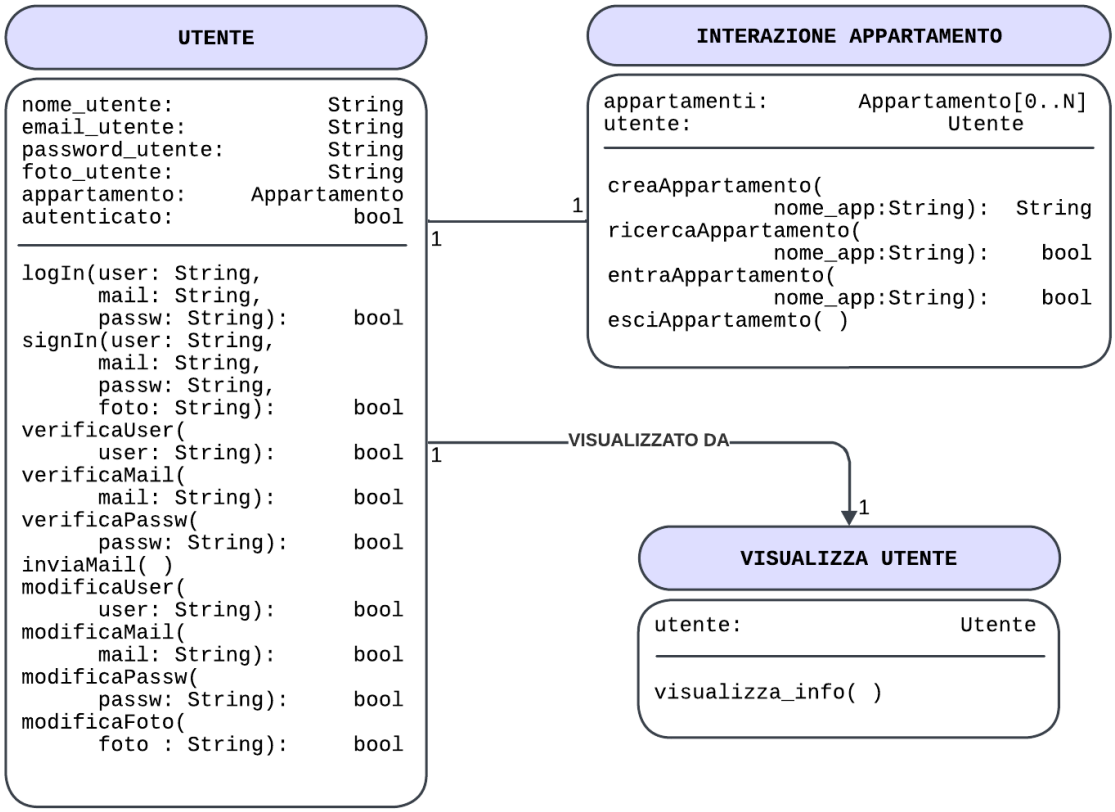
Nel presente capitolo viene presentato e spiegato il diagramma delle classi previste per il progetto RoomUnity. Ogni componente presente nel diagramma dei componenti diventa una o più classi. Tutte le classi individuate sono caratterizzate da un nome della classe, una lista di attributi, che identifica i dati gestiti dalla classe, e una lista di metodi, che definisce le operazioni previste all'interno della classe. Ogni classe, oltre a definire un particolare aspetto del progetto, può essere anche associata ad altre classi e dunque è possibile fornire informazioni su come le classi interagiscono e si relazionano tra loro.

Di seguito, sono riportate le classi individuate a partire dai diagrammi di contesto e dei componenti. Nel primo diagramma presentato vi è una rappresentazione completa per dare una visione d'insieme delle classi del progetto, in seguito verranno fatti degli approfondimenti sulle singole macroaree del progetto.

1.1) UML COMPLESSIVO



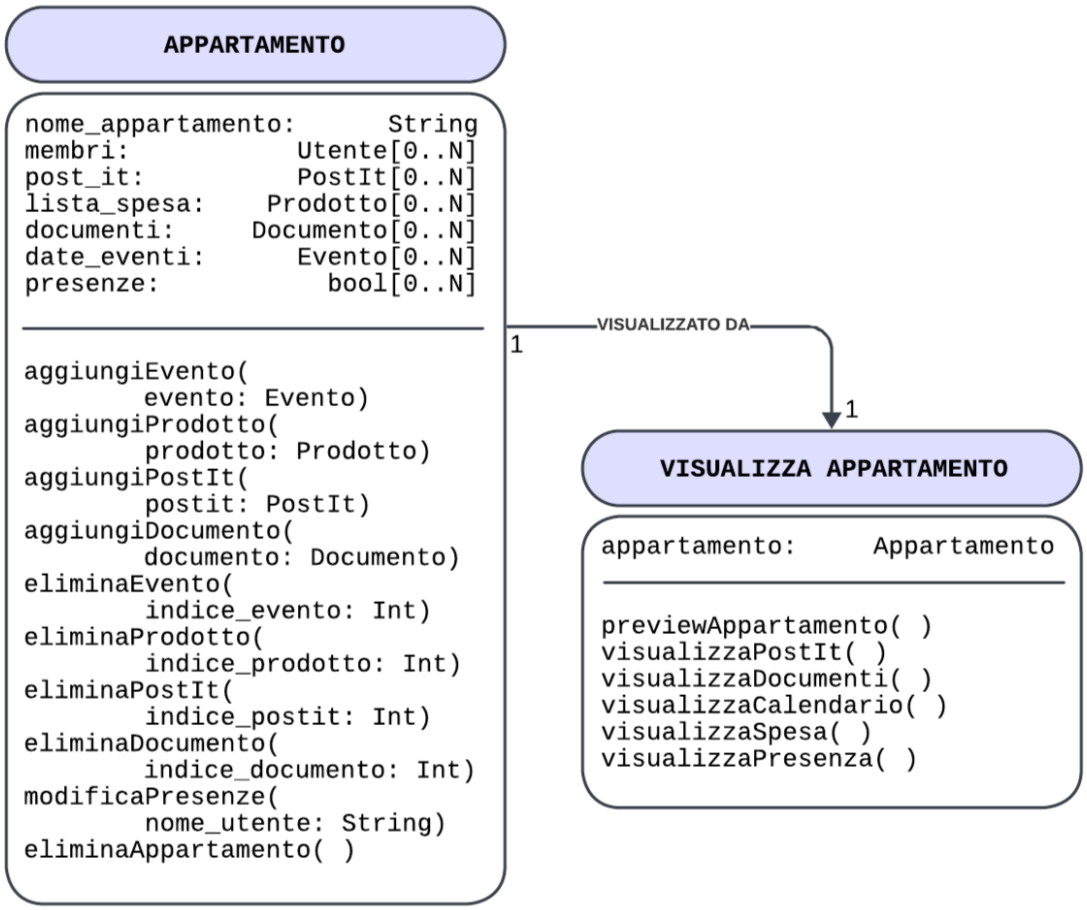
1.2) UML UTENTE



Le due classi che dirigono principalmente il progetto di RoomUnity sono: la classe appartamento, su cui verrà posta l'attenzione nella prossima sezione, e la classe utente. Quest'ultima contiene le informazioni riguardanti l'utente inserite nel momento del sign in (o modificate in seguito). All'interno degli attributi della classe utente è stata inserita, al mero scopo logico, una variabile autenticato/logged, la quale indica se l'istanza utente rappresenta un utente loggato o meno, in quanto dispongono di interazioni differenti con il sistema (questo a livello di progettazione software può essere implementato mediante il controllo della presenza di un nome utente non nullo). Questa variabile cambia appunto di stato nel momento in cui viene effettuato il metodo login or sign in.
Vi sono poi una serie di metodi che agiscono sulla modifica delle informazioni di base dell'utente (al di fuori del nome, che viene utilizzato come metodo identificativo e univoco del singolo utente/account) e di metodi (privati) che

agiscono sulla verifica delle informazioni inserite in fase di modifica, ma anche in fase di login e sign-in, affinché vengano rispettati i requisiti di sicurezza.
Oltre alle classiche informazioni contenute all'interno di un profilo utente di una qualsiasi applicazione, è stato necessario inserire una variabile *appartamento*, che connette l'utente al proprio appartamento, o, nel caso in cui la variabile non fosse presente, separa le varie funzionalità di un utente con appartamento virtuale da quelle di un utente non ancora entrato in un appartamento.
Si è optato, in quanto fulcro della progettazione di RoomUnity, per definire una classe (a livello logico) Interazione Appartamento che definisce tutti metodi che gli utenti (in appartamento o meno) possono effettuare per interagire con la classe appartamento. Questa, infatti, permette all'utente di: creare e aggiungere al database un nuovo appartamento tramite il metodo *creaAppartamento* o di cercare un appartamento già esistente tramite il metodo *ricercaAppartamento* (privato), che va a definire una funzione di supporto per confermare la presenza o meno di un nome univoco dell'appartamento nel database utilizzato da *creaAppartamento* e da *entraAppartamento*. La classe possiede come attributo la lista di tutti gli appartamenti nel database e il nome dell'utente che vuole utilizzare i suoi metodi. Questo perché, se un utente vuole utilizzare il metodo per interagire con un appartamento, allora, tramite il suo identificativo, gli sarà possibile aggiornare i dati dell'utente (appartamento).
Si è poi pensato di aggiungere una classe per gestire la visualizzazione delle informazioni dell'utente e dell'appartamento, per semplificare e dividere la parte di front-end da quella di back-end, per aiutare in fase di sviluppo e scrittura del codice.

1.3) UML APPARTAMENTO

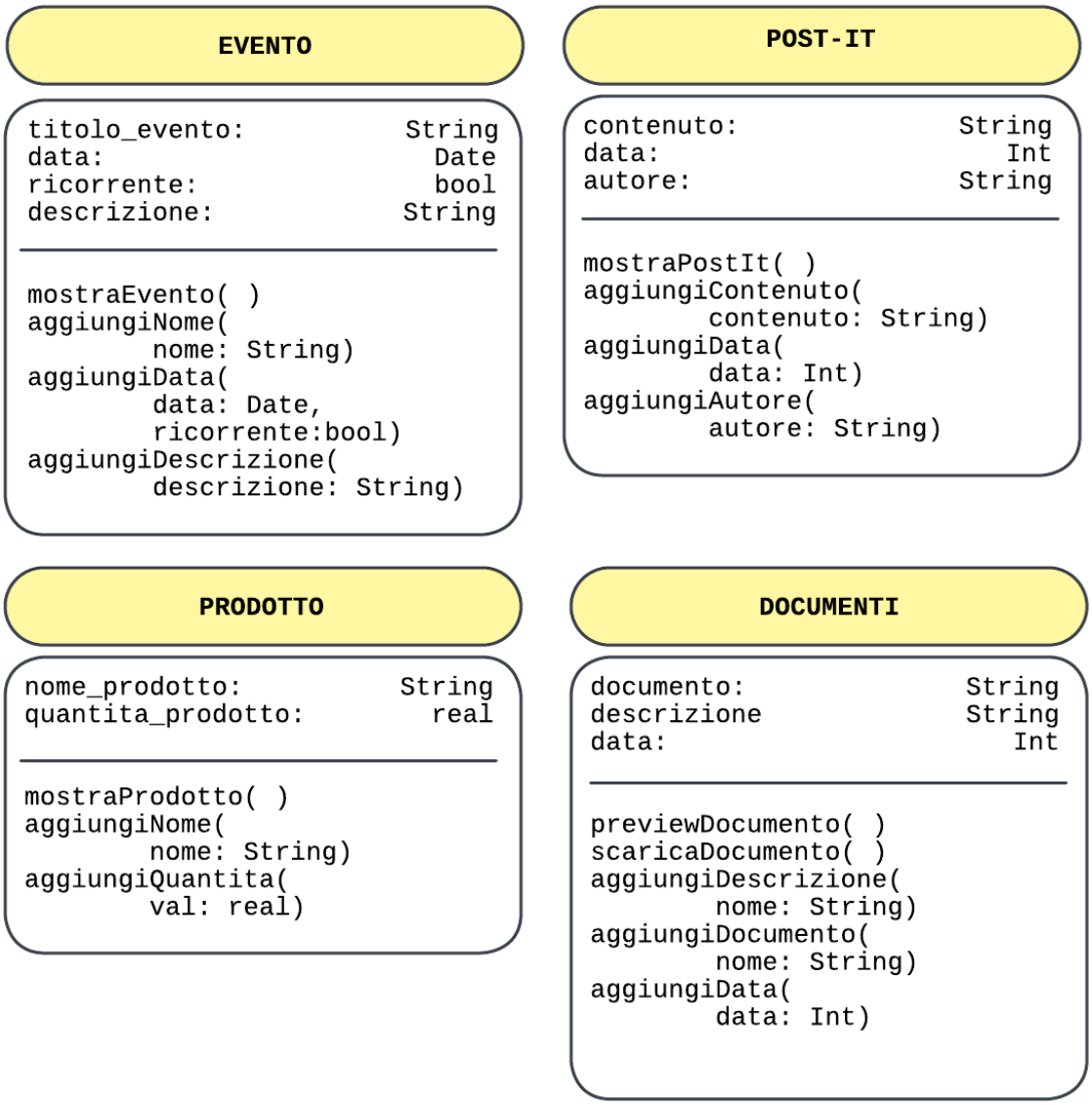


La classe Appartamento contiene i dati relativi alle informazioni dell'appartamento
virtuale, sono memorizzati:

- nome univoco (codice alfanumerico) dell'appartamento;
- membri dell'appartamento;
- presenza dei membri in casa;
- post-it in bacheca;
- lista della spesa;
- date ed eventi;
- sezione documenti.

Questi ultimi quattro sono liste riempite da istanze definite appositamente per l'appartamento (relativamente: Post-It, Prodotti, Eventi, Documenti). Questi elementi vanno così a definire i vari e principali metodi della classe Appartamento, come metodi per aggiungere o togliere istanze.
Oltre a questo, vi è un metodo *modificaPresenze* che, una volta inserito come parametro un utente, ne varia il suo stato di presenza o meno in casa.
In aggiunta ai metodi appena citati, vengono definite delle funzioni di gestione e amministrazione dell'appartamento: *eliminaAppartamento* che permette, solo all'admin o al membro più "anziano" nell'appartamento (la lista dei membri è stata pensata come una coda, in questo modo l'utente che ha creato l'appartamento virtuale sta in testa ed è facilmente rintracciabile) di eliminare l'appartamento.
Come per la classe utente, si è pensato di inserire una classe *visualizzaAppartamento*, per sottolineare le funzioni che andranno a definire il front-end dell'applicazione. Viene infatti definita, oltre ad una generica preview dell'appartamento, anche la singola possibilità di visualizzare le sezioni/ feature dell'appartamento stesso.

1.4) UML SEZIONI APPARTAMENTO



Questa serie di classi definiscono le vere e proprie pedine della scacchiera che è il progetto RoomUnity. In breve, ogni singola classe definisce in maniera astratta come i coinquilini dello stesso appartamento virtuale andranno ad interagire. Vi sono inoltre una serie di metodi e funzioni volte all'inserimento o all'eventuale aggiornamento dei dati delle istanze stesse e dei metodi per mostrare l'istanza a livello di front-end.

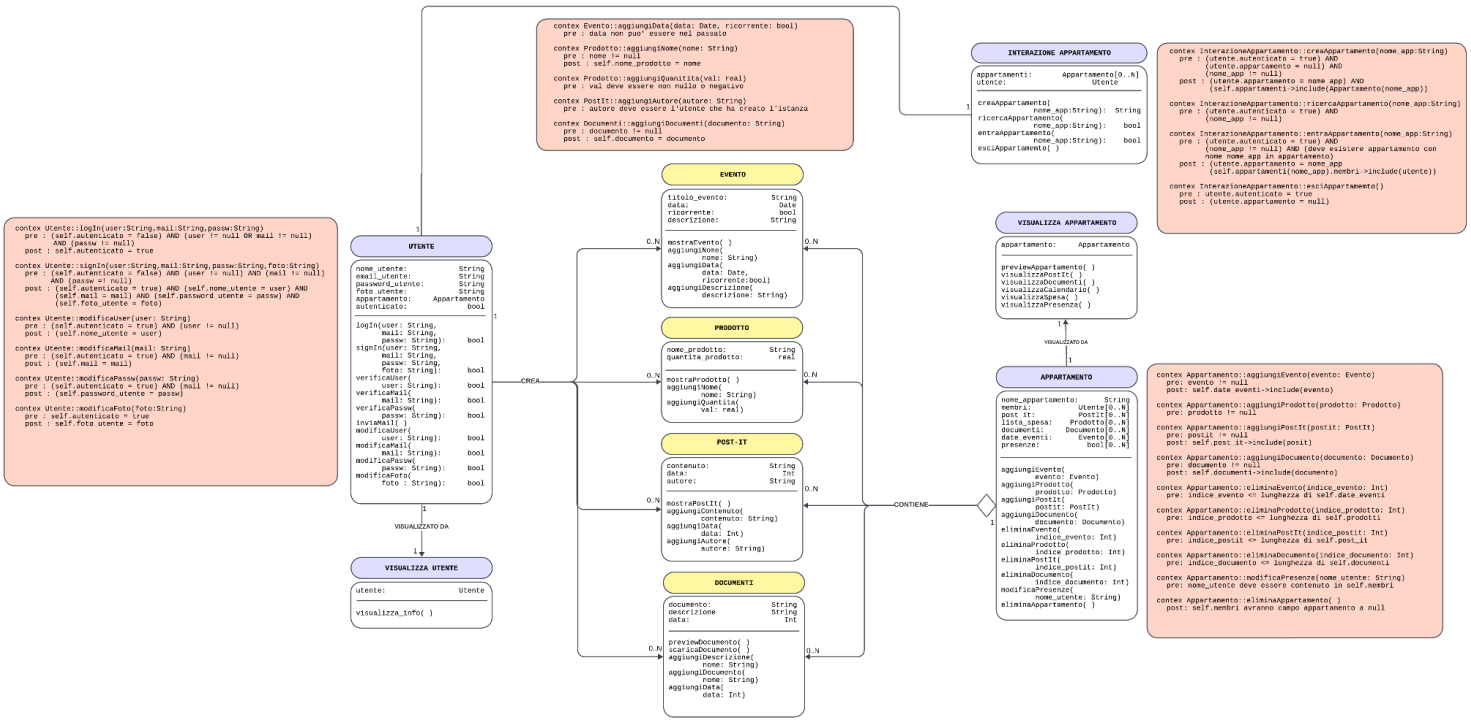
2) CODICE IN OCL

In questo capitolo sono descritti, tramite il linguaggio formale OCL, i vincoli e le espressioni relative ai modelli UML appena descritti. Questo perchè UML è spesso limitante e non sufficientemente preciso per specificare tutte le regole e i vincoli del sistema.

Il linguaggio OCL invece permette di esprimere le condizioni che devono essere rispettate dagli elementi del modello, come invarianti di classe, precondizioni e postcondizioni dei metodi. Le espressioni OCL sono tipicamente utilizzate nei diagrammi delle classi UML per aggiungere dettagli precisi e non ambigui che non possono essere espressi graficamente. Infatti, essendo un linguaggio formale, OCL è privo di effetti collaterali e non può alterare il modello o i dati, rendendolo ideale per la specifica dei requisiti e la validazione del modello. Inoltre, OCL è strettamente integrato con UML, il che consente di mantenere la consistenza tra il modello grafico e le specifiche testuali.

A seguire è riportato l'OCL complessivo del progetto RoomUnity.

2.1) OCL COMPLESSIVO



A seguire il diagramma complessivo di OCL accostato all'UML.

2.2) OCL UTENTE

I primi due constraint riguardano l'accesso dell'utente all'applicazione. Le imposizione di questi constraint sono che, durante la chiamata della funzione, i campi devono essere non nulli e l'utente non dev'essere loggato (autenticato = true).
I seguenti tre constraint controllano unicamente che alla funzione vengano passati tutti i parametri richiesti e non siano nulli. Successivamente, la funzione si occuperà di modificare gli attributi della classe Utente.

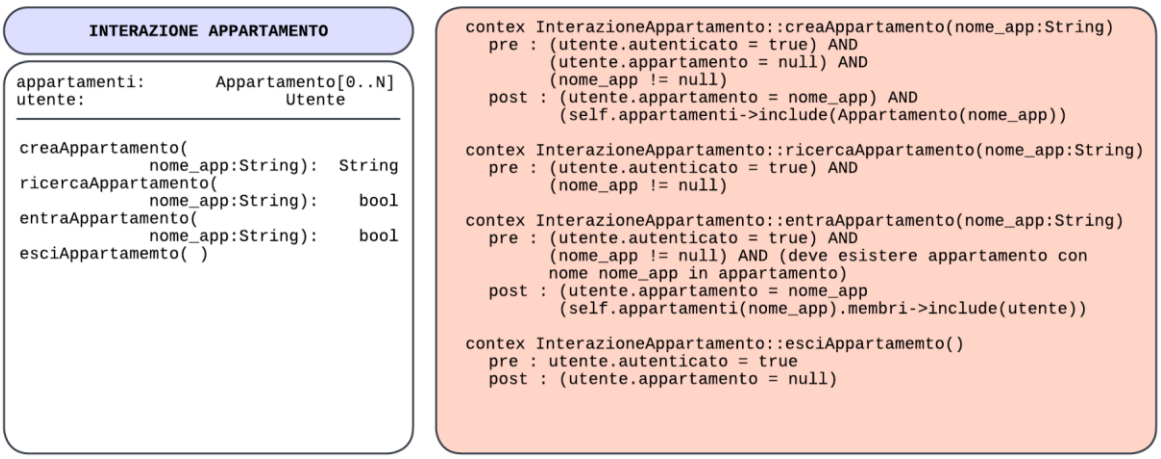
```
contex Utente::logIn(user:String,mail:String,passw:String)
   pre : (self.autenticato = false) AND (user != null OR mail != null)
         AND (passw != null)
   post : self.autenticato = true

contex Utente::signIn(user:String,mail:String,passw:String,foto:String)
   pre : (self.autenticato = false) AND (user != null) AND (mail != null)
        AND (passw =! null)
   post : (self.autenticato = true) AND (self.nome_utente = user) AND
          (self.mail = mail) AND (self.password_utente = passw) AND
          (self.foto_utente = foto)

contex Utente::modificaUser(user: String)
   pre : (self.autenticato = true) AND (user != null)
   post : (self.nome_utente = user)

contex Utente::modificaMail(mail: String)
   pre : (self.autenticato = true) AND (mail != null)
   post : (self.mail = mail)

contex Utente::modificaPassw(passw: String)
   pre : (self.autenticato = true) AND (mail != null)
   post : (self.password_utente = passw)

contex Utente::modificaFoto(foto:String)
   pre : self.autenticato = true
   post : self.foto_utente = foto
```

UTENTE

```
nome_utente:              String
email_utente:             String
password_utente:          String
foto_utente:              String
appartamento:        Appartamento
autenticato:                bool
-------------------------------------
logIn(user: String,
      mail: String,
      passw: String):       bool
signIn(user: String,
      mail: String,
      passw: String,
      foto: String):        bool
verificaUser(
      user: String):        bool
verificaMail(
      mail: String):        bool
verificaPassw(
      passw: String):       bool
inviaMail( )
modificaUser(
      user: String):        bool
modificaMail(
      mail: String):        bool
modificaPassw(
      passw: String):       bool
modificaFoto(
      foto : String):       bool
```

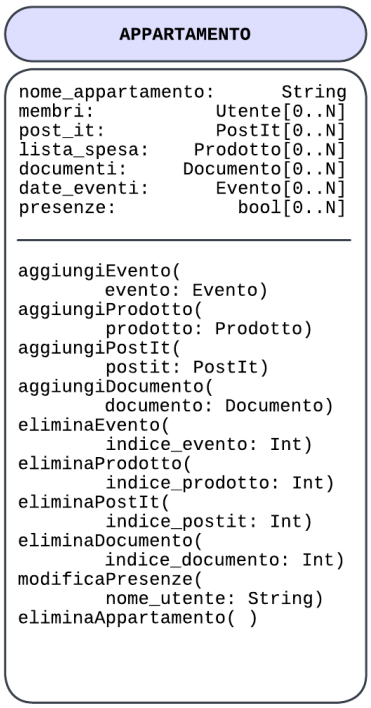
Il primo constraint della classe interazione, impone che (come tutte le altre funzioni della seguente classe) il parametro passato non sia nullo. Inoltre, è necessario che l'utente (Utente) chiamante la funzione sia un utente autenticato. Successivamente, il constraint impone di aggiungere l'appartamento appena creato come valore del parametro appartamento dell'utente (e viceversa: l'utente creatore deve essere inserito all'interno della lista membri dell'appartamento creato)e alla lista di tutti gli appartamenti.
La funzione entra appartamento *entraAppartamento* impone che il parametro nome_app passato, oltre a non essere un valore nullo, deve

esistere come nome di un appartamento all'interno della lista degli appartamenti.
L'ultima funzione impone che a termine dell'operazione sia necessario togliere il collegamento bidirezionale che connette l'appartamento e l'utente: campo appartamento nell'istanza utente e l'elemento membri dell'istanza appartamento.



2.3) OCL APPARTAMENTO

Come nella classe precedente, anche le seguenti funzioni impongono che i valori passati siano non nulli. La funzione *aggiungiEvento* impone che venga aggiunto l'evento all'attributo dell'istanza (date_eventi). La cosa è analoga per *aggiungiPostIt* e *aggiungiDocumenti* (non per la lista della spesa in quanto ne varia unicamente la quantità).
In seguito alle seguenti 4 funzioni di eliminazione degli elementi, viene imposto che l'indice da rimuovere non possa essere fuori dal range della dimensione del numero degli elementi della Lista associata.
La funzione *modificaPresenze* deve ricevere il nome di un membro che deve essere attualmente all'interno dell'appartamento.
Per concludere, la funzione che elimina l'appartamento deve necessariamente, a termine dell'operazione, rimuovere il valore dell'attributo *appartamento* dai singoli (ex)membri.




```
contex Appartamento::aggiungiEvento(evento: Evento)
  pre: evento != null
  post: self.date_eventi->include(evento)

contex Appartamento::aggiungiProdotto(prodotto: Prodotto)
  pre: prodotto != null

contex Appartamento::aggiungiPostIt(postit: PostIt)
  pre: postit != null
  post: self.post_it->include(posit)

contex Appartamento::aggiungiDocumento(documento: Documento)
  pre: documento != null
  post: self.documenti->include(documento)

contex Appartamento::eliminaEvento(indice_evento: Int)
  pre: indice_evento <= lunghezza di self.date_eventi

contex Appartamento::eliminaProdotto(indice_prodotto: Int)
  pre: indice_prodotto <= lunghezza di self.prodotti

contex Appartamento::eliminaPostIt(indice_postit: Int)
  pre: indice_postit <= lunghezza di self.post_it

contex Appartamento::eliminaDocumento(indice_documento: Int)
  pre: indice_documento <= lunghezza di self.documenti

contex Appartamento::modificaPresenze(nome_utente: String)
  pre: nome_utente deve essere contenuto in self.membri

contex Appartamento::eliminaAppartamento( )
  post: self.membri avranno campo appartamento a null
```

2.4) OCL SEZIONI

Le classi riguardanti le varie sezioni (calendario, lista spesa, bacheca, documenti) mantengono i constraint riguardanti l'inserimento non nullo di dati. Oltre a ciò, la data che deve essere inserita per un evento non può essere antecedente alla data attuale.
Il valore inserito come prezzo di un prodotto deve essere non negativo.
Il campo autore di un post-it deve corrispondere al chiamante della funzione aggiungiPostIt( ), ovvero il creatore del post-it stesso.