UNIVERSITÀ DI TRENTO

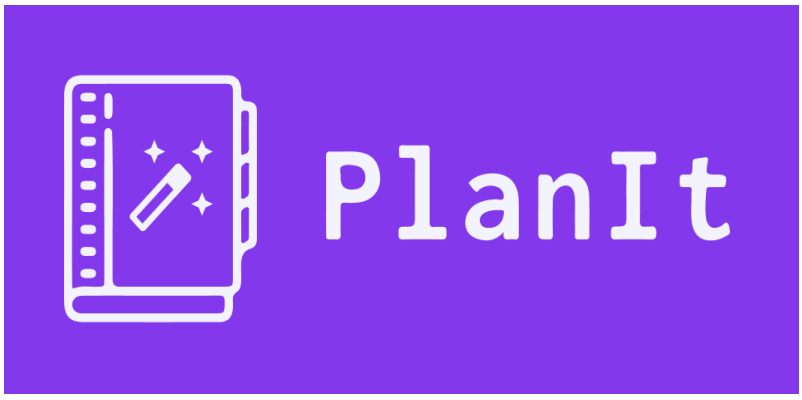
Dipartimento d'Ingegneria e
Scienze dell'informazione

Progetto:

PlanIt

Titolo del documento:

Documento di Architettura (Diagrami delle classi e codice in OCL)



Gruppo T56
Gabriele Lacchin, Denis Lucietto ed Emanuele Zini

Indice

Scopo del documento

Il presente documento riporta la definizione dell'architettura del progetto PlanIt usando diagrammi delle classi in Unified Modeling Language (UML) e codice in Object Constraint Language (OCL). Nel precedente documento D2 è stato presentato il diagramma degli use case, il diagramma di contesto e quello dei componenti con le relative descrizioni e specificazioni. Ora, 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. La logica viene descritta in OCL perché tali concetti non sono esprimibili in nessun altro modo formale nel contesto di UML. Tutti i diagrammi, che verranno presentati, hanno una descrizione a loro associati per delinearne il loro scopo e funzionalità.

- diagramma delle classi;
- object constraint language;
- diagramma delle classi e codice object constraint language.

ù

Fare link immagine e lista di riferimenti, linking

1 Diagramma delle classi

Nel presente capitolo vengono presentate le classi previste nell'ambito del progetto PlanIt. Ogni componente presente nel diagramma dei componenti è stata utilizzato per fare 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. Anche se, sottolineiamo, che le associazioni fornite tra le classi sono quelle non banali e che non richiedono un'ulteriore specificazione. Riportiamo di seguito le classi individuate a partire dai diagrammi di contesto e dei componenti. In questo processo si è proceduto anche nel massimizzare la coesione e minimizzare l'accoppiamento tra classi, cercando di evitare la ridondanza.

DCL1 : **Tipologie di utenti**

Analizzando il diagramma di contesto realizzato per il progetto PlanIt si nota la presenza di tre attori: "utente autenticato standard", "utente non autenticato" e "utente autenticato premium", per questo motivo sono state delineate tre classi, ovvero "Utente", "UtenteAutenticato", "UtenteNonAutenticato".

Si è deciso di fare la classe "Utente" in modo tale di unire le due tipologie di utenti, "UtenteAutenticato" e "UtenteNonAutenticato", che si legano a "Utente" con delle generalizzazioni. L'unica funzionalità che ha "Utente" è il GoHomepage()" che non è altro che l'azione che riporta l' "Utente" nell' Homepage del PlanIt; le funzionalità, che possono essere fatte nella pagina Homepage, verranno descritte nella presentazione della classe "Gestione Homepage" DCL8.

L' "UtenteNonAutenticato" è colui che utilizza il sito in versione demo; che, come abbiamo già descritto in precedenza nel D1 e D2, ha la possibilità di usare il sito con delle limitazioni sulle funzionalità. Queste funzionalità sono presenti in "GestioneSalvataggioDatiInLocale": l'importanza di questa classe è anche quella di salvare in locale le modifiche effettuate dall' "UtenteNonAutenticato", infatti nel caso in cui questo decidesse di autenticarsi nel sito, le modifiche effettuate nella versione demo verranno trasferite e salvate nel suo account di "UtenteAutenticato".

L' "UtenteAutenticato" è colui che accede al sito autenticandosi, ottenendo tutte le funzionalità del sito, però avendo delle limitazioni rispetto all' utente autenticato che si abbona al sito D1 RF4.2. La gestione della sottoscrizione al sito e il passaggio all'account utente autenticato premium è gestito da "GestioneImpostazioniAccount" (DCL3), che verrà presentato successivamente.

La gestione dell'autenticazione viene gestita dalla classe "InterfacciaAuth0"(DCL2), a cui la classe "UtenteAutenticato" è legata mediante un'associazione. Questa classe viene descritta nel paragrafo successivo. Gli unici attributi che ha "UtenteAutenticato" sono i suoi dati più importanti, ovvero: "UserID", "Email" e "Username". Questi attributi, ottenuti da Interfaccia di Auth0 dopo l'autenticazione, insieme ad altre impostazioni dell'account, sono salvate nel database MongoDB. Per questo motivo, "UtentiAutenticato" è legato con "GestioneChiamateMongoDB"(DCL13) che verrà descritto in seguito.

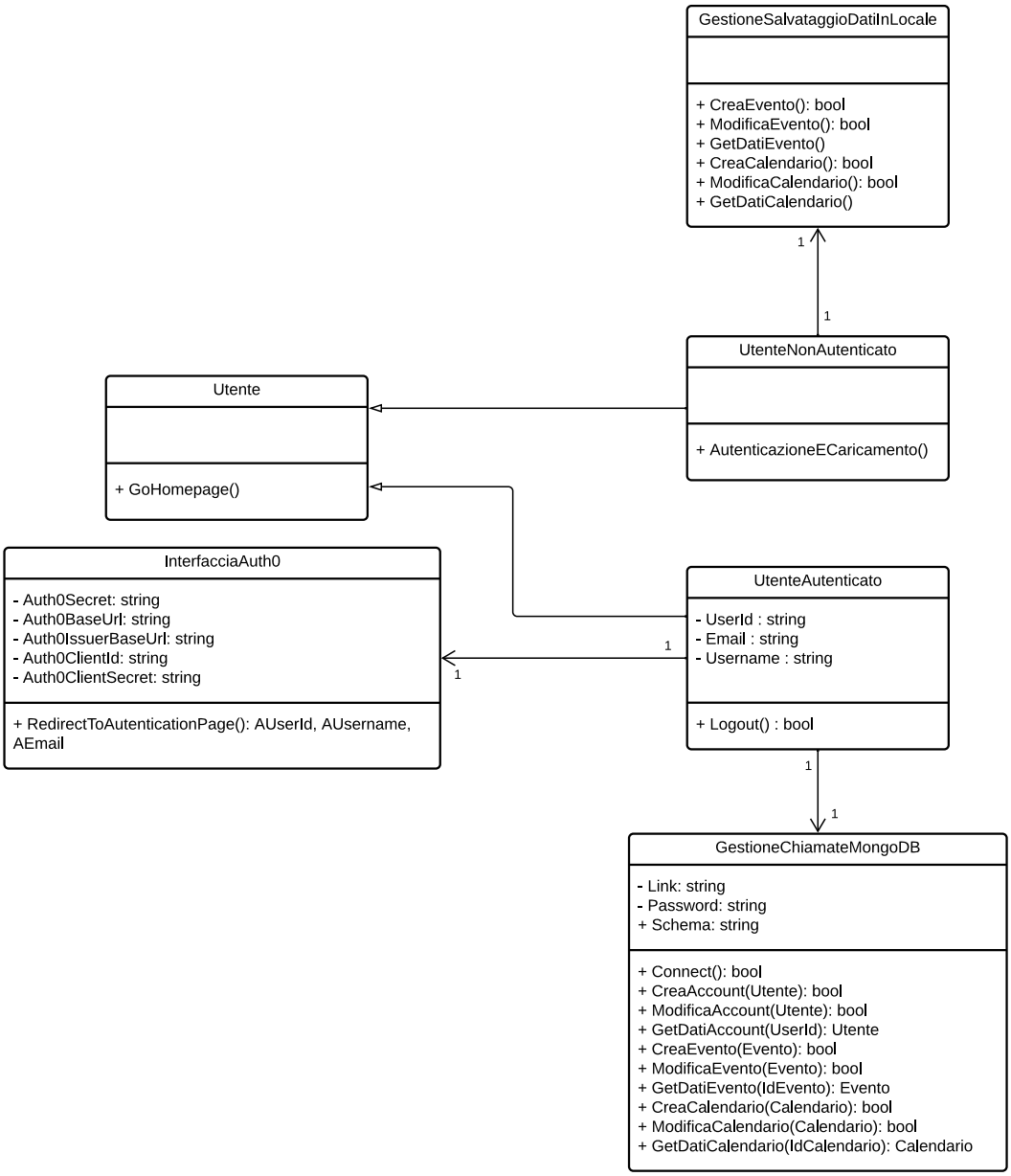


Immagine PNG/SVG Diagramma delle classi "Tipologie di Utenti"

DCL2 : **Interfaccia Auth0**

Il diagramma di componenti presentato nel documento D2 presenta un componente chiamato allo stesso modo, ovvero sempre "Interfaccia Auth0" (D2 ACI28). Questa classe rappresenta il meccanismo di autenticazione degli utenti attraverso il sistema esterno presente anche nel diagramma di contesto, ovvero Auth0. Questa classe si interfaccia con questo sistema esterno di autenticazione, che gestirà completamente la procedura di autenticazione. Il software di PlanIt memorizzerà solo l'"Email", "UserId" e "Username", valori restituiti dall'autenticazione; questi valori sono salvati nella classe, presentata precedentemente, "UtenteAutenticato", istanziandola. Infatti un utente autenticato non può esser istanziato se non dopo aver effettuato l'autenticazione.

Questa classe fa partire la procedura di autenticazione per un "Utente" una volta che si invoca la funzione "AccessoWebApp()", presente nella classe "GestioneHomepage", descritta in DCL8, o si invoca la funzione "Autenticazione()" presente nella classe "UtenteNonAutenticato" (DCL1). Ricordiamo, che per autenticazione si intende o registrazione o login, a seconda se l' "Utente" abbia o meno un suo account su PlanIt.

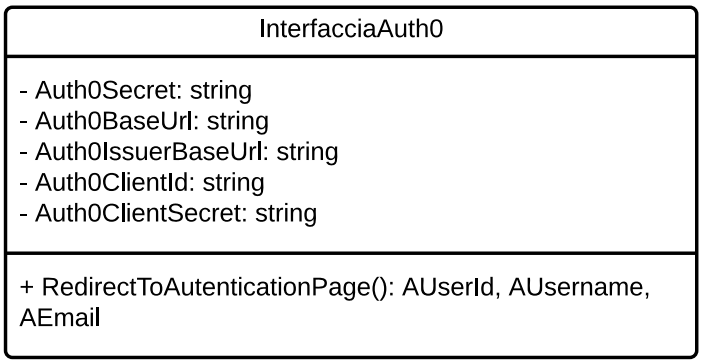


Immagine PNG/SVG Diagramma delle classi "InterfacciaAuth0"

DCL3 : Impostazioni Utente

Il diagramma di componenti analizzato presenta un componente “Gestione impostazioni account” e altri componenti utilizzati per gestire le impostazioni dell’account. Dunque, è stato identificato la classe “GestioneImpostazioniUtente” che permette all’utente autenticato, insieme alle classi a cui è associato, di modificare e gestire il proprio account secondo le proprie preferenze per le seguenti impostazioni, nella pagina “Impostazioni Account”: sistema di mappe preferito da utilizzare nell’aggiunta del luogo dell’evento, ore di sonno, sottoscrizione all’abbonamento (divenendo “utente autenticato premium”), scelta metodo di pagamento, scelta metodo di utilizzo Google Calendar, modifica password ed username. “GestioneImpostazioniUtente” è collegato a “UtenteAutenticato” in modo tale che ogni volta che l’utente autenticato va a modificare le proprie impostazioni account, queste sono salvate su MongoDB; infatti, ricordiamo, che “UtenteAutenticato” è collegato alla classe “Gestione chiamate MongoDB” che verrà descritto successivamente in (DCL13).

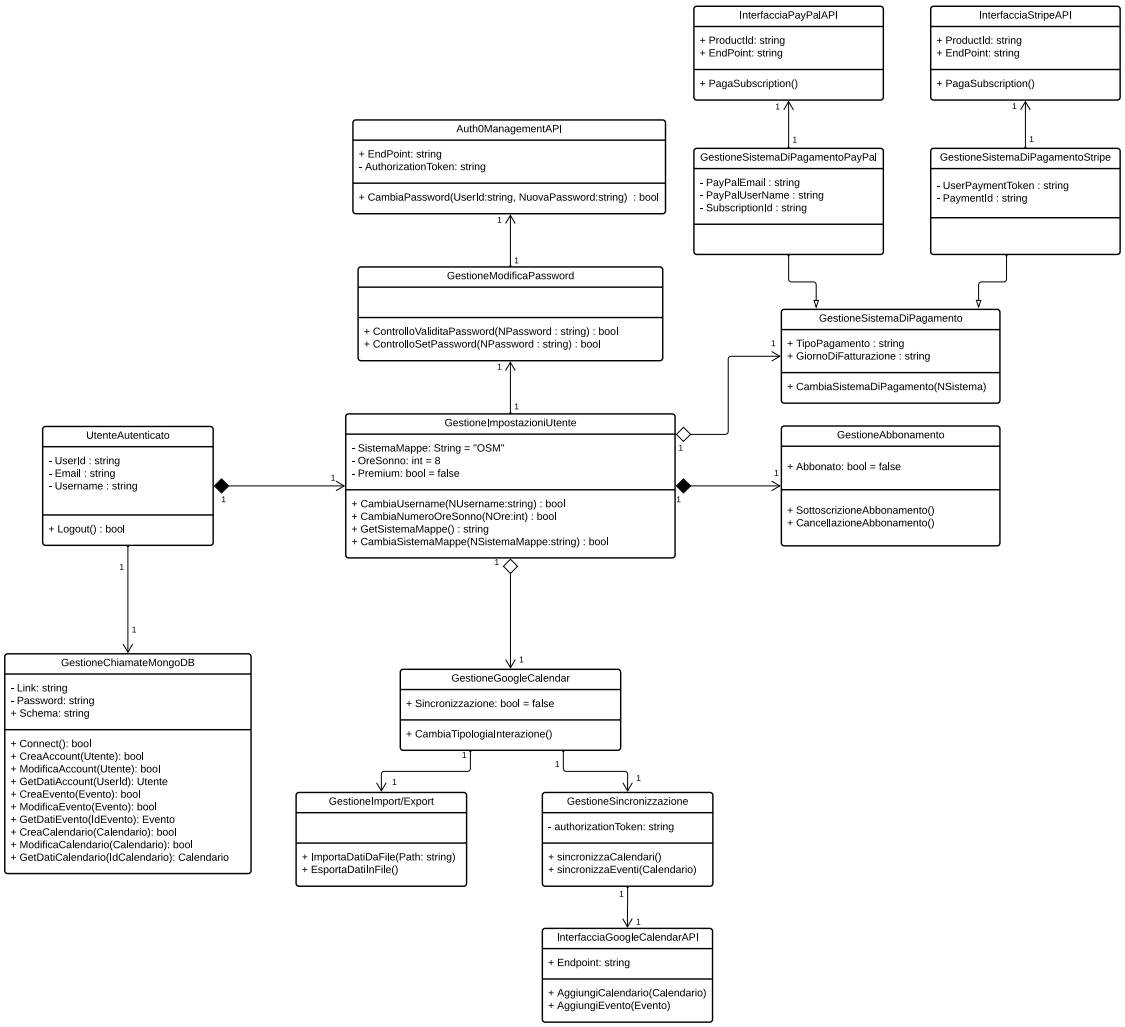
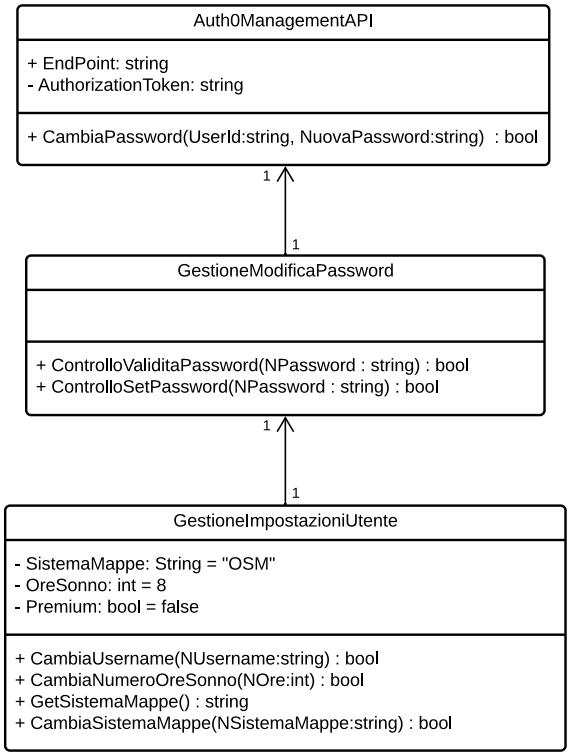


Immagine PNG/SVG Diagramma delle classi "Impostazioni Utente"

3.1 : Gestione modifica password e Auth0 management API

Analizzando il diagramma di componenti si nota la presenza di "Gestione modifica password" e "Interfaccia management API Auth0", per questo motivo sono state fatte queste classi con gli stessi nomi. Il loro scopo è gestire il modifica password, azione che può essere fatta dalle impostazioni account, e il reset password, azione che si può fare nel caso in cui si volesse recuperare e cambiare la password quando si è dimenticata, partendo dalla pagina di autenticazione di PlanIt. La nuova password inserita viene controllata in "GestioneModificapassword" mediante la funzione "ControlloValiditaPassword", in modo tale che la password segui le regole delineate in D1 RNF2.1. Una volta che la nuova password passa il controllo, viene inviata ad Auth0 grazie alla funzione "CambiaPassword(UserId, nuovaPassword) presente in "Auth0ManagementAPI".



3.2 : Abbonamento

Il diagramma di contesto analizzato nel documento D2 presenta un terzo attore che è stato già citato in DCL1, ovvero l'utente autenticato premium. Quest'ultimo usufruisce di un abbonamento al sito, accedendo alla versione premium della piattaforma. Per questo motivo è stata fatta la classe "Abbonamento", il cui scopo è gestire la sottoscrizione e cancellazione dell'abbonamento. Inoltre, grazie a questa classe, è possibile sapere se un utente autenticato sia standard (attributo "Abbonato" = false) o premium (Abbonato = true).

Infine, come si può notare, è presente una composition tra "Impostazioni Utente" e "Abbonamento" il cui scopo è indicare che la classe "contenuta", "Abbonamento", esiste solo con la classe "contenitrice", "Impostazioni Utente"; infatti la classe "Abbonamento" ha senso ed esiste solo se l'utente ha le "Impostazioni Utente", ovvero è autenticato.

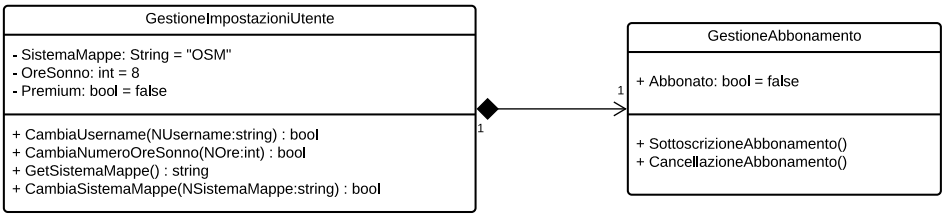


Immagine PNG/SVG Diagramma delle classe "Gestione modifica password e Auth0 management API"
Immagine PNG/SVG Diagramma delle classi "Abbonamento"

3.3 : Gestione Google Calendar

Analizzando il diagramma di componenti realizzato per il progetto PlanIt, si nota la presenza dei componenti "Gestione utilizzo Google Calendar", "Gestione sincronizzazione" e "Gestione import/export file", per questo motivo sono state fatte queste tre classi ("GestioneGoogleCalendar", "GestioneImport/Export" e "GestioneSincronizzazione"). Queste classi hanno lo scopo di gestire l'integrazione ed interazione di PlanIt con Google Calendar.

"GestioneGoogleCalendar" contiene l'attributo "sincronizzazione" il cui scopo è quello di salvare quale sia il metodo di integrazione di Google Calendar utilizzato dall'utente autenticato; infatti l'utente autenticato può scegliere solo un metodo alla volta di interazione con Google Calendar. Inoltre, questa classe permette il cambio della tipologia di interazione con Google Calendar usufruendo delle classi "GestioneImport/Export" e "GestioneSincronizzazione" che gestiscono rispettivamente i processi di import/export di file di eventi o calendari di Google Calendar da e/o verso PlanIt e la sincronizzazione (l'account Google viene collegato con quello PlanIt) con Google Calendar, con cui avviene l'importazione ed esportazione di eventi e calendari automaticamente.

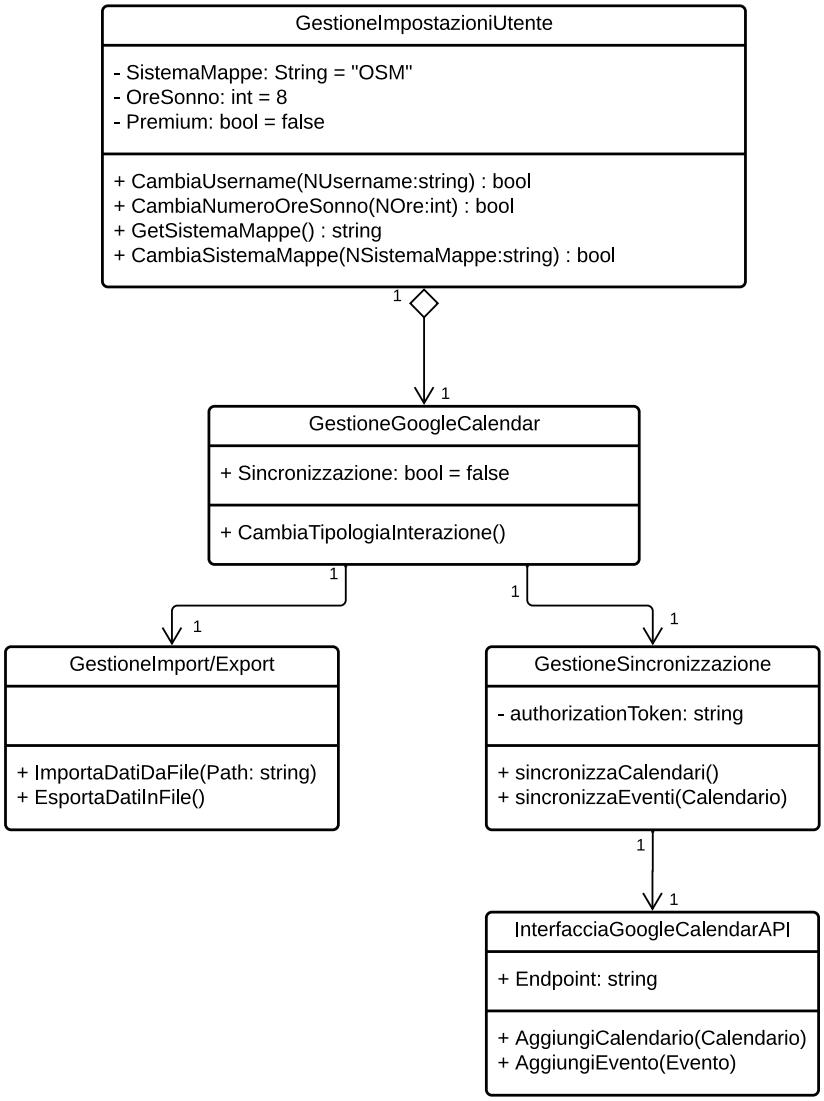


Immagine PNG/SVG Diagramma delle classi "Gestione Google Calendar"

3.4 : Gestione sistema di pagamento

Il diagramma dei componenti analizzato, realizzato per il progetto PlanIt, presenta i componenti "Gestione metodo di pagamento", "Interfaccia PayPal" e "Interfaccia Payments", per questo motivo sono state pensate queste cinque classi: "GestioneSistemaDiPagamento", "GestioneSistemaDiPagamentoStripe", "GestioneSistemaDiPagamentoPayPal", "InterfacciaPayPalAPI" e "InterfacciaStripeAPI". La funzione di queste classi è gestire il pagamento dell'abbonamento e cambiamento del metodo di pagamento di quest'ultimo.

"GestioneSistemaDiPagamento" salva il giorno di fatturazione dell'abbonamento dell'utente autenticato premium e permette, come si nota, il cambiamento del metodo di pagamento, il quale può esser scelto tra PayPal e Stripe.

"GestioneSistemaDiPagamentoPayPal", "GestioneSistemaDiPagamentoStripe, "InterfacciaPayPalAPI" e "InterfacciaStripeAPI" gestiscono il processo di pagamento dell'abbonamento mediante il metodo di pagamento "PayPal" e "Stripe" rispettivamente, conservando, una volta inserite, le credenziali di pagamento per questi sistemi dell'utente autenticato premium, in modo tale che il pagamento dell'abbonamento possa avvenire in automatico ogni mese.

Come si può notare, è presente un'"aggregation" tra "GestioneImpostazioniUtente" e "GestioneSistemiDiPagamento" che serve ad indicare che "GestioneSistemiDiPagamento" "is part of" "GestioneSistemiDiPagamento".

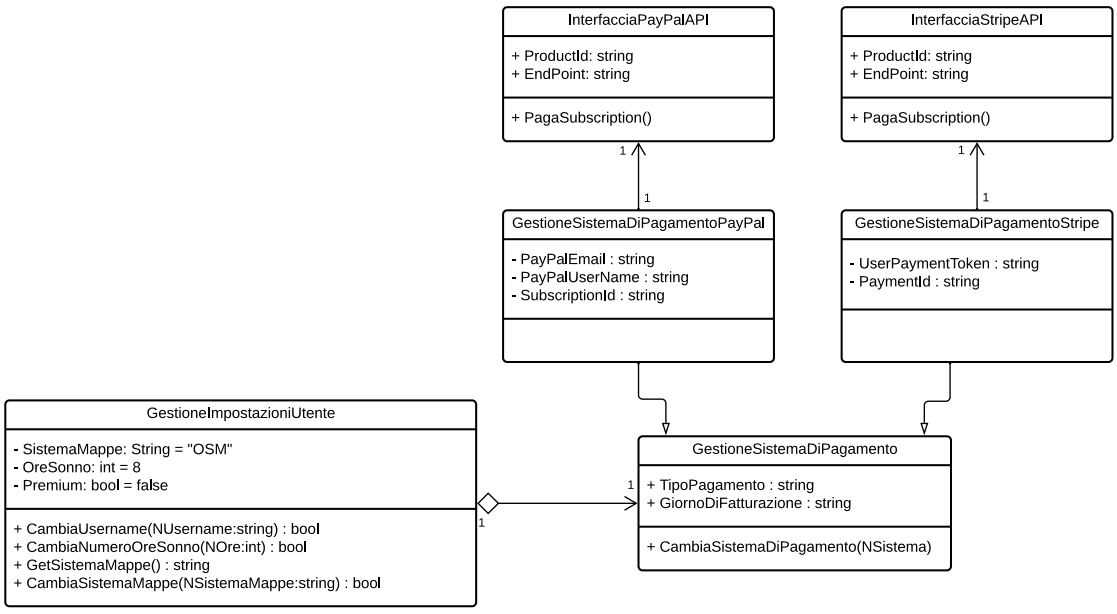


Immagine PNG/SVG nome file

DCL4 : Gestione Visualizzazione Calendari

Analizzando il diagramma di componenti fatto per il progetto PlanIt si può vedere la presenza di “Gestione visualizzazione calendari ed eventi”, “Gestione Attività”, “Gestione filtro impegni” e “GestioneFiltriCalendari”. Per questa ragione sono state ideate le seguenti classi: “GestioneVisualizzazioneCalendari”, “GestioneInformazioniEvento”, “GestioneFiltroEventi”, “GestioneFiltriCalendari” e “GestioneAttività”. Queste classi permettono la visualizzazione di calendari ed eventi nella schermata Calendario D1 FE2anche visualizzandoli seguendo dei filtri che può inserire l’utente.
“GestioneVisualizzazioneCalendari” ottiene da “GestionechiamateMongoDB” i calendari dell’utente che mostra a quest’ultimo. Dopo, grazie a “GestioneFiltriCalendari”, possono essere mostrati i calendari secondo dei filtri definiti dall’utente: ovvero questo’ultimo definisce quale calendari visualizzare nella schermata "Calendario".
In seguito, “Gestione filtro Eventi” può mostrare una lista di eventi secondo dei criteri delineati dall’utente.
“Gestione Attività” permette all’utente di visualizzare la lista degli eventi giornalieri, permettendo a quest’ultimo di segnare il completamento o meno dell’attività o anche l’eliminazione di questa. L’eliminazione o l’indicazione di non completamento dell’attività porta alla riprogrammazione del calendario.
Infine, la classe "GestioneInformazioneEvento" permette all’utente, nel caso in cui andasse a premere un evento presente nella schermata "Calendario", di visualizzare tute le informazioni riguardanti l’evento creato. Per sapere quali informazioni vengono visualizzate si guardi D2 ACI20.

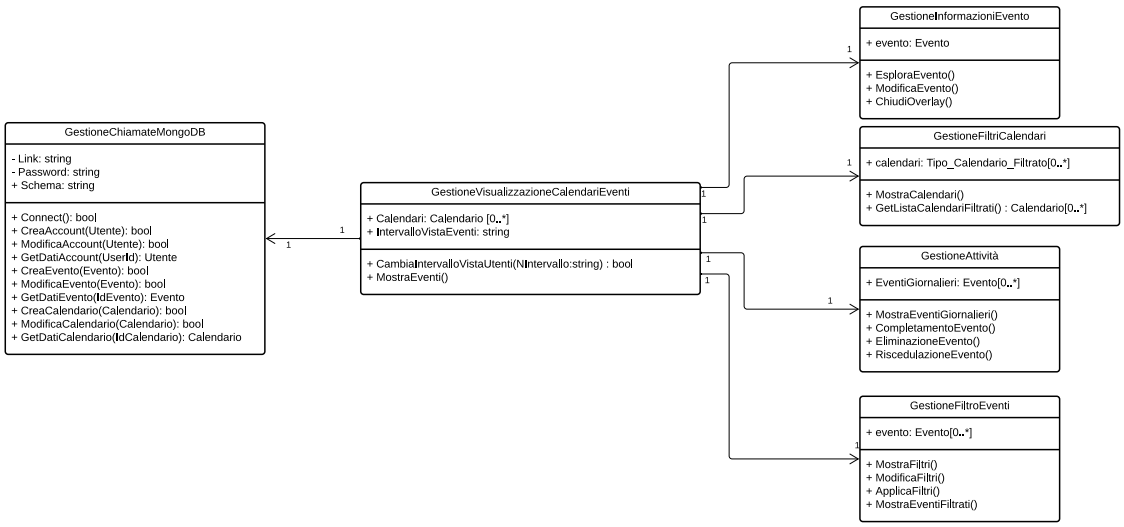


Immagine PNG/SVG Diagramma delle classi "Gestione Visualizzazione Calendari"

DCL5 : **Dashboard**

Il diagramma dei componenti esibisce i seguenti componenti: “Gestione Dashboard”, “Gestione eventi settimanali” e “Gestione eventi giornalieri”. Dunque, sono state progettate le classi “GestioneDashboard”, “GestioneDashboardEventiSettimanali” e “GestioneDashboardEventiGiornalieri”, che hanno il fine è di mostrare dei grafici per fornire delle informazioni sull’uso del tempo all’utente. La classe “GestioneDashboard” ottiene da “GestioneChiamateMongoDB” i calendari di un utente, in modo tale da poter avere anche tutti i suoi eventi. Grazie agli eventi mostra una HeatMap, il cui scopo è già stato descritto in D2 UC8, e passa gli eventi settimanali a “GestioneDashboardEventiSettimanali" e quelli giornalieri a “GestioneDashboardEventiGiornalieri”.
Dunque, grazie a ciò, queste due classi possano mostrare altri grafici riguardanti gli eventi, nel periodo di tempo considerato, mediante i grafici citati in D2 UC8(grafico a barre per “GestioneDashboardEventiSettimanali” e grafico a torta con lista di eventi per “GestioneDashboardEventiGiornalieri”).
Infine, è giusto citare che l’utente possa interagire con tutti i grafici sopra citati, selezionando particolari calendari o attività che sono presenti nei rispettivi grafici: in questo modo si può ottenere una visualizzazione più "personalizzata".

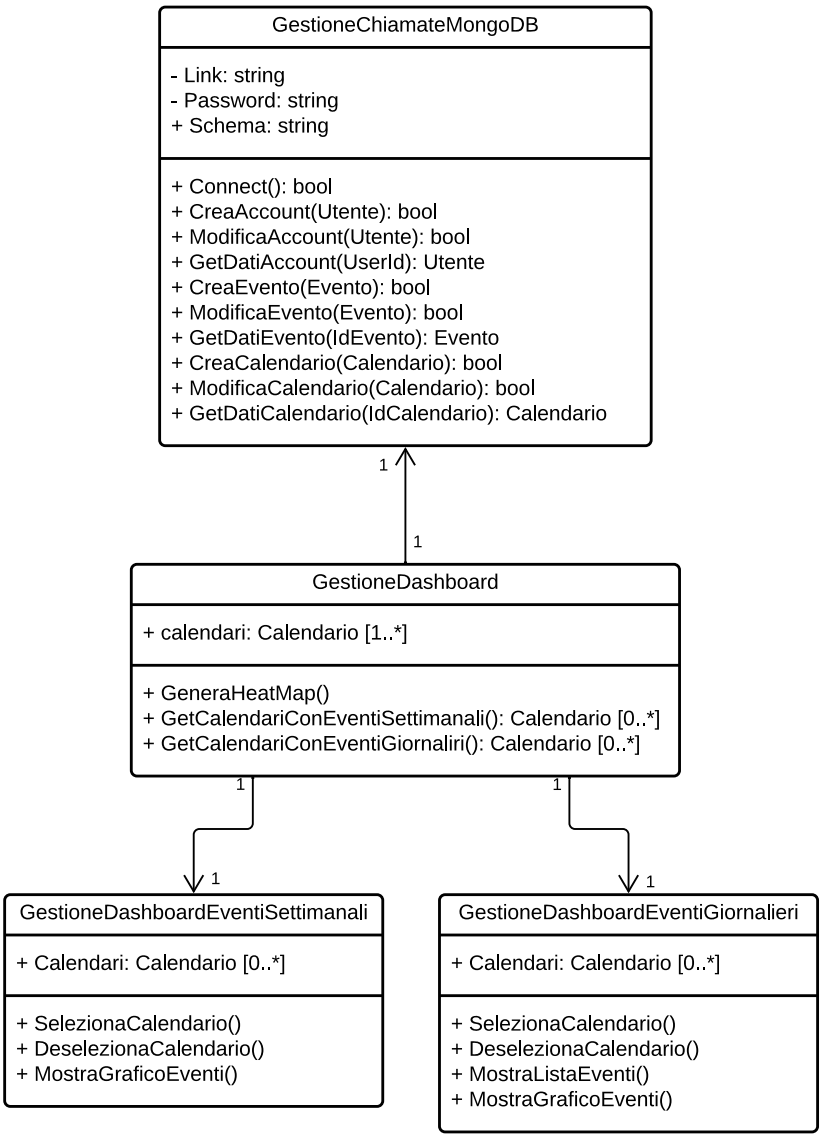


Immagine PNG/SVG Diagramma delle classi "Dashboard"

DCL6 : **Gestione evento e calendario**

Il diagramma dei componenti analizzato, realizzato per il progetto PlanIt, presenta i componenti “Gestione creazione o modifica evento” e “Gestione creazione e modifica calendario” e dei componenti a loro associati. Per questo motivo, sono state fatte le classi “Calendario” e “Evento” e delle classi a loro associate, che si possono vedere nella foto sottostante, che hanno l’obiettivo di permettere all’utente di poter creare o modificare eventi e calendari secondo le modalità già descritte in D2 UC4e D2 UC10.

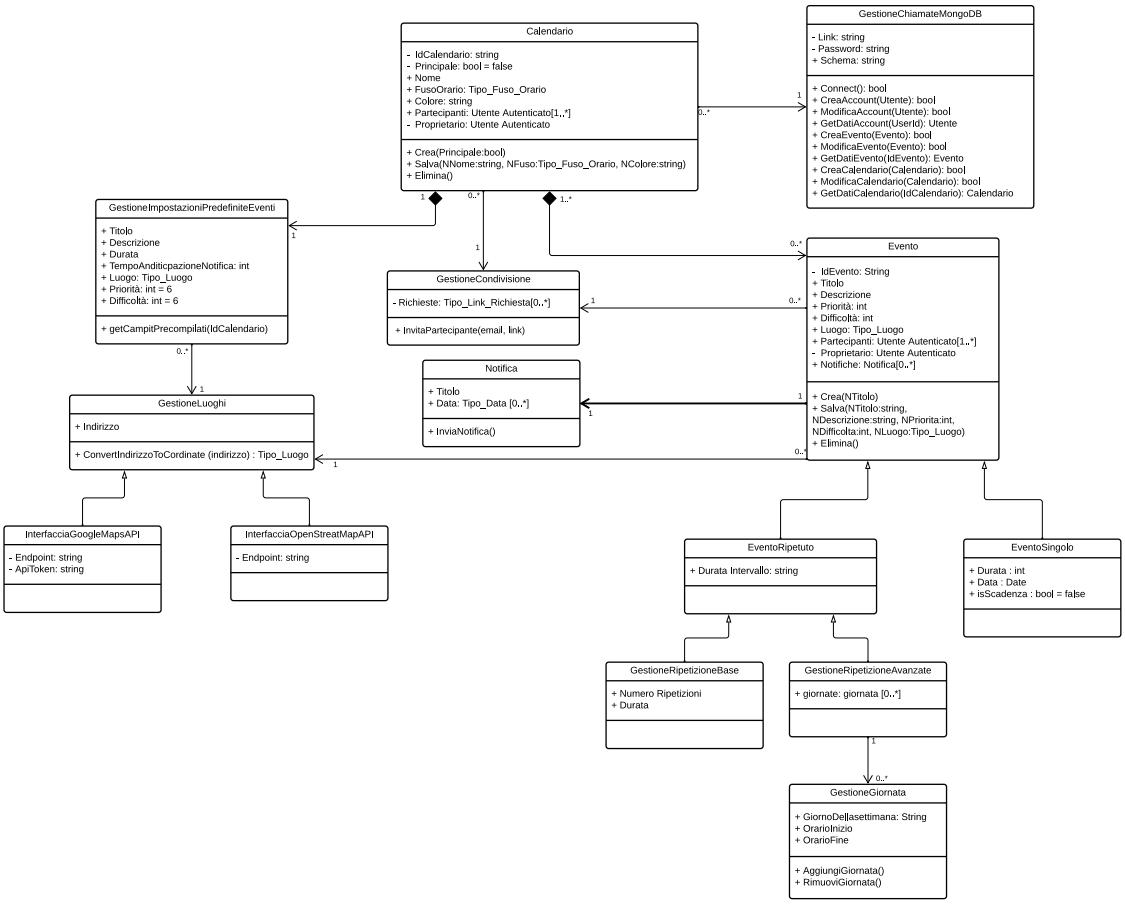


Immagine PNG/SVG Diagramma delle classi "Gestione evento e calendario"

6.1 : Gestione creazione o modifica calendario

Analizzando il diagramma di componenti fatto per il progetto PlanIt, si può vedere la presenza del componente "Gestione creazione o modifica calendario", per questa ragione sono state ideate le classi "GestioneImpostazioniPredefiniteEventi" e "Calendario" che gestiscono la procedura di creazione o modifica di un calendario, secondo le impostazioni scelte dall'utente.
Quando l'utente va a modificare o creare un calendario può andare anche a definire dei valori con cui precompilare alcuni campi presenti nella creazione o modifica evento: lo scopo di "GestioneImpostazioniPredefiniteEventi" è quello di gestire la procedura di definizione di questi campi, andando anche a salvare i loro valori. Una volta che un calendario è stato creato o modificato, questo viene salvato su MongoDB grazie alla classe "GestioneChiamateMongoDB" che lo invia al sistema esterno di archiviazione MongoDB (si noti l'associazione tra "Calendario" e "Gestione chiamate MongoDB). "GestioneChiamateMongoDB" viene presentato più nel dettaglio successivamente (DCL13).
Si specifica, che i vari set e get degli attributi presenti in "GestioneImpostazioniPredefiniteEventi" e "Calendario" non sono stati scritti, nella lista di funzioni di tali classi, in quanto ritenute banali.

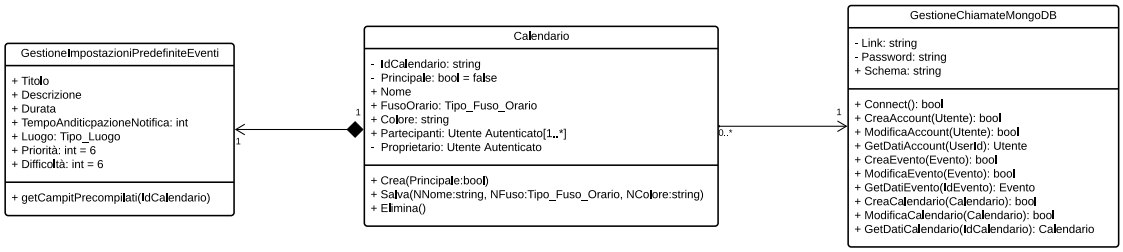


Immagine PNG/SVG Diagramma delle classi "Gestione creazione o modifica calendario"

6.2 : Gestione creazione o modifica evento

(ricordati di menzionare impostazioni predefinite eventi) → ottenute dal calendario, in realtà ne ho parlato nelle altre classi che sono collegate con impostazioni predefinite eventi

Analizzando il diagramma di componenti, fatto per il progetto PlanIt, si può vedere la presenza dei componenti “Gestione creazione o modifica evento” e “Gestione notifiche”, per questa ragione sono state ideate le classi “Notifica” e “Evento”.
La seconda classe citata gestisce la procedura di creazione o modifica dell’evento secondo i valori scelti per i vari attributi presenti nella classe “Evento” (si veda anche D2 UC4). Invece, la classe “Notifica” gestisce la procedura di creazione o modifica della notifica per quel specifico evento che si sta creando o modificando, per questo motivo è presente un’associazione tra “Evento” e “Notifica”.
Come si può notare è presente una “composition” tra “Evento” e “Calendario” che serve ad indicare che un evento, “oggetto contenuto”, esiste solo con l’oggetto “contenitore” “Calendario”: infatti un evento non può non essere non associato ad un calendario.
Sono state definite anche le classi “EventoSingolo”, “EventoRipetuto”, “GestioneRipetizioneBase”, "GestioneRipetizioneAvanzate” e “GestioneGiornata” che hanno la funzionalità di gestire il processo di definizione temporale di quando avviene l’evento che si sta modificando o creando. Sono state individuate due classi distinte “GestioneRipetizioneBase” e “GestioneRipetizioneAvanzate”, utilizzate per gestire la definizione di un evento ripetuto, in quanto la prima classe citata viene utilizzata per sovrintendere la determinazione di un evento di cui viene definito il numero di volte in cui viene ripetuto senza indicare quali giornate, e la seconda classe, al contrario, viene utilizzata per sovrintendere la determinazione di un evento ripetuto di cui si indica anche le giornate in cui è presente.
Si specifica, che i vari set e get degli attributi presenti in “Evento”, “Notifica”, “EventoRipetuto”, EventoSingolo”, "GestioneRipetizioneBase”, “GestioneRipetizioneAvanzate” e “GestioneGiornata” non sono stati scritti, nelle liste di funzioni di tali classi, in quanto ritenute banali.

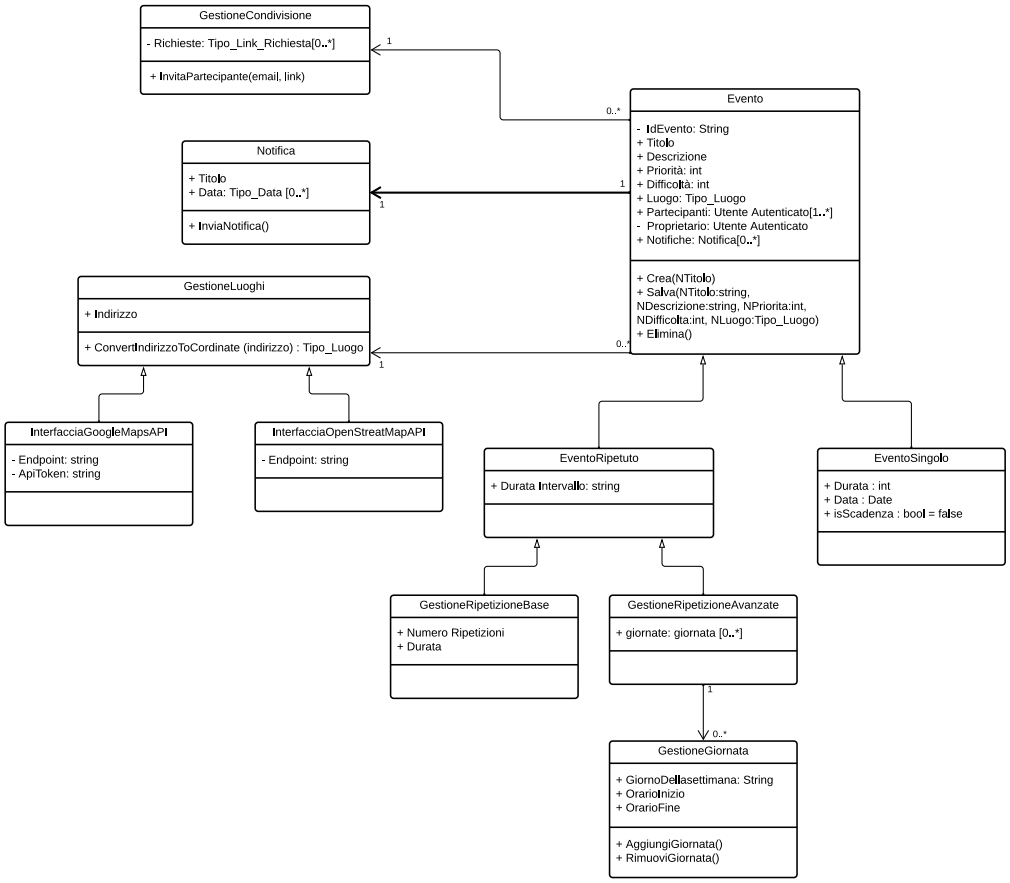


Immagine PNG/SVG Gestione creazione o modifica evento

6.3 : Gestione Luoghi

Il diagramma dei componenti analizzato, realizzato per il progetto PlanIt, presenta un componente "Interfaccia mappe", per questa ragione sono state fatte le classi "Gestione Luoghi", "InterfacciaGoogleMapsAPI" e "InterfacciaOpenStreetMapAPI", che hanno lo scopo di gestire la definizione delle coordinate del luogo dove avviene un evento. Poiché la definizione del luogo di eventi può esser definito sia dalle "GestioneImpostazioni PredefiniteEventi" in fase di creazione o modifica calendario (il luogo appartiene ad uno dei campi che può esser precompilato) e può esser definito anche in fase di creazione o modifica evento, questa classe "Gestione Luoghi" è legata sia a "GestioneImpostazioni PredefiniteEventi" sia a "Evento". Sono state inserite due classi "InterfacciaGoogleMapsAPI" e "InterfacciaOpenStreetMapAPI", che hanno lo scopo di gestire la procedura di conversione dell'indirizzo del luogo inserito in coordinate, mediante l'utilizzo delle piattaforme "Google Maps" e "OpenStreetMap", rispettivamente. Infatti, si ricorda, che l'utente autenticato può decidere, secondo le proprie preferenze, da impostazioni account (DCL3) quale sistema di mappe utilizzare.

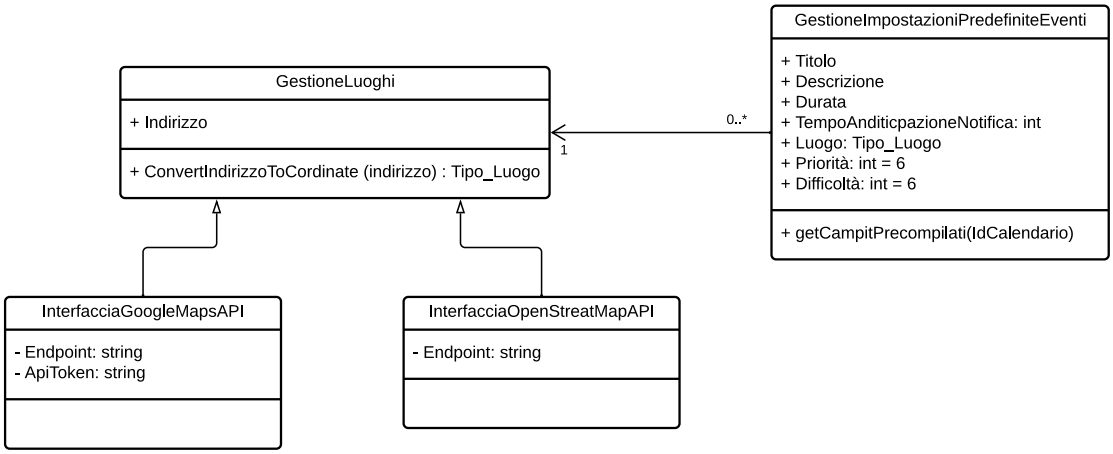
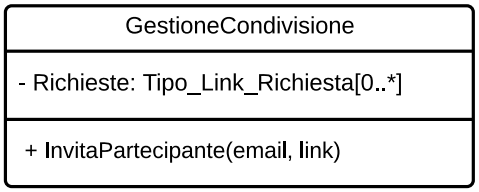


Immagine PNG/SVG Diagramma delle classi "Gestione luoghi"

6.4 : Gestione condivisione

Analizzando il diagramma di componenti, fatto per il progetto PlanIt, si può vedere la presenza del componente “Gestione condivisione”, allora è stata fatta la classe “Gestione-Condivisione", che ha la funzionalità di gestire la procedura di condivisione di calendari ed eventi; infatti, come si può osservare dall'immagine in DCL6, questa classe ha un' “association” sia con “Evento” che con “Calendario”.
La condivisione viene effettuata utilizzando l'email dei partecipanti a cui condividere l'evento o il calendario e un link, di tipo stringa, contenente il riferimento al dato di tipo "Tipo_Link_Richiesta" (DCL14), che contiene le informazioni fondamentali riguardanti l' “Evento” o “Calendario” che si sta condividendo. Per poter capire cosa contenga l'oggetto di tipo Tipo_Link_Richiesta a cui fa riferimento “link”, si legga DCL14. Dunque, grazie all'email del partecipante e tale link, questa classe invia la richiesta di condivisione, salvando l'esito di tale richiesta nell'attributo booleano “Accettata”. Nel caso in cui la richiesta fosse accettata, gli “utenti autenticati”, che hanno accettato la condivisione, vengono aggiunti nell'attributo "Partecipanti" di o “Evento” o “Calendario”, a seconda di cosa si sta condividendo.



DCL7 : Interfaccia Cloudflare

Analizzando il diagramma di contesto realizzato per il progetto PlanIt, si nota la presenza del sistema esterno “Cloudflare”, per questa ragione è stata fatta questa classe “Interfaccia-CloudFlare” che permette ottenere il log di tutte le connessioni che sono state effettuate per accedere al sito, con il resoconto se sono state filtrare o meno.
Per ottenere tali informazioni la classe si interfaccia al sistema esterno Cloudflare.

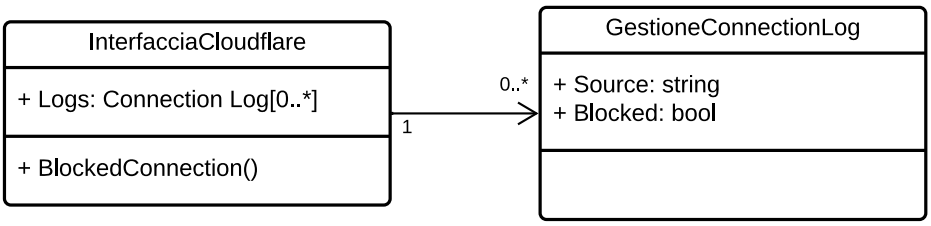


Immagine PNG/SVG Diagramma delle classi "Gestione Condivisione"
Immagine PNG/SVG Diagramma delle classi "Interfaccia CloudFlare"

DCL8 : **Gestione Homepage**

Il diagramma dei componenti analizzato, realizzato per il progetto PlanIt, presenta le componenti “Gestione Homepage” e “Interfaccia Iubenda”, dunque sono state ideate le classi “InterfacciaIubenda”, “GestioneHomepage” e “Tipo_Cookie”.

typo, "lisa" al posto di "lista"

La classe “GestioneHomepage” gestisce la procedura di scelta, effettuata da un utente che vuole entrare nel sito, di entrare nel sito in modalità demo (in questo caso c’è l’esecuzione della funzione "AccessoProvaDemo()"), accedendo da utente non autenticato, o entrare nella web app autenticandosi (c’è l’esecuzione del metodo "AccessoProvaDemo()"), ovvero registrandosi o accedendo tramite Auth0 su PlanIt. Infatti, qualora l’utente abbia deciso di accedere alla web app con l’ autenticazione, verrà indirizzato alla pagina di gestione login e registrazione di Auth0 grazie all’ “InterfacciaAuth0”, attraverso la funzione “RedirectToAuthentication()” presente in quest’ultima classe citata. Ad autenticazione avvenuta, l’utente, divenuto, autenticato verrà indirizzato alla web app.
Dunque, questa classe gestisce l’indirizzamento dell’utente nella modalità del sito scelta.
“GestioneHomepage” mostra anche il banner di cookie contenente le cookie policy che possono essere accettate o meno dall’utente. Il banner di cookie, come anche le politiche di privacy, sono gestite e ottenute dalla classe “InterfacciaIubenda”, a cui “GestioneHomepage” è legata.
“InterfacciaIubenda”, interfacciandosi con Iubenda, invia ad’ “GestioneHomepage” il codice del banner di cookie contenente anche il link alla pagina di politiche di privacy. Una volta accettati i cookie, questi vengono salvati in "GestioneHomepage" nell’array “Cookie”, il cui tipo è “Tipo_cookie”, dove sono presenti i cookie salvati nel browser dell’ Utente.
Si sottolinea che solo i cookie accettati saranno presenti nell’attributo "Cookie" presente in "GestioneHomepage"; infatti, come si può notare, grazie all’ "InterfacciaIubenda" è possibile accettare anche solo una parte dei cookie.

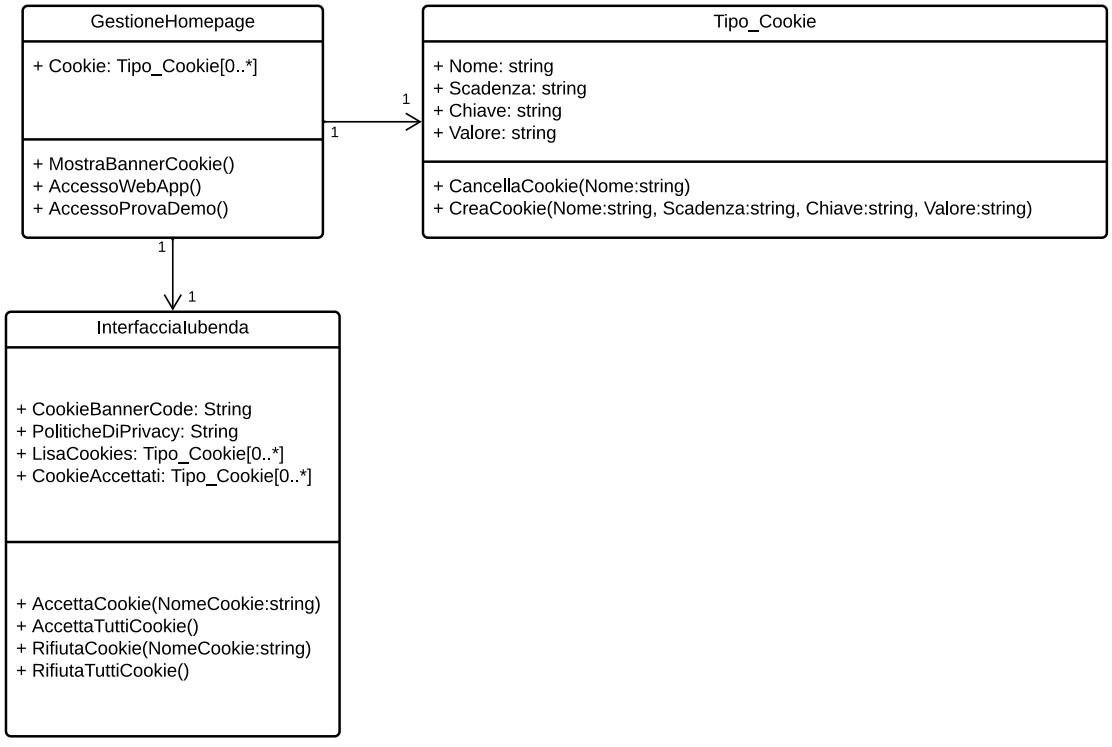


Immagine PNG/SVG Diagramma delle classi "Homepage"

DCL9 : Tipo calendario filtrato

Nella classe "GestioneFiltriCalendari" (DCL4) è presente un attributo di tipo "Tipo_Calendario_Filtrato", tipo di dato che non avevamo ancora descritto. La classe "Tipo_Calendario_Filtrato" è stata delineata per definire il tipo di calendario che viene utilizzato nella classe "GestioneFiltriCalendari". Questo tipo permette di andare a selezionare e deselezionare un calendario, azioni fondamentali per andare a filtrare i calendari.

DCL10 : Tipo Fuso orario

Nella classe "Calendario" (DCL6.1) è presente un attributo di tipo "Tipo_Fuso_Orario", tipo di dato che non avevamo ancora descritto.
La classe "Tipo_Fuso_Orario" è stata delineata per definire il tipo di dato dell'attributo "FusoOrario": questo tipo permette di salvare, prendere (get) e impostare (set) l'offset del fuso orario del calendario che stiamo andando a creare o modificare e il nome della località che ha tale fuso orario. L'intero presente in "GMToffset" è un offset rispetto al GMT, ovvero l'orario di Greenwich, dunque l'attributo salvato è ad esempio: "+1", "+2", "-1", e così via. Le funzione set() e get() di "GMToffset" e "Località" non sono state inserite nel diagramma della classe perché ritenute banali. Per vedere l'utilità della funzionalità del fuso orario di un calendario, si legga D2 UC10.1.

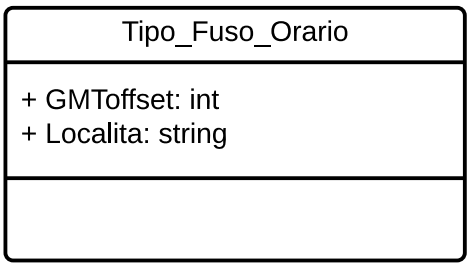
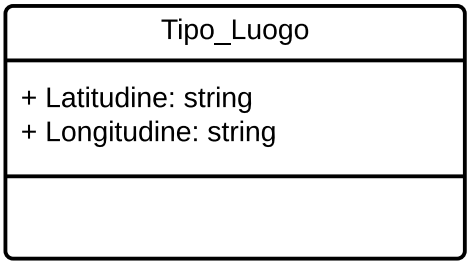


Immagine PNG/SVG Diagramma delle classi "Tipo calendario filtrato"
Immagine PNG/SVG "Diagramma delle classi "Tipo Fuso Orario"

DCL11 : Tipo_Luogo

Nelle classi "GestioneImpostazioniPredefiniteEventi" (DCL6.1) e "Evento" (DCL6.2) è presente l'attributo "Luogo" di tipo "Tipo_Luogo", tipo di dato che non avevamo ancora descritto. Questo tipo permette di salvare il valore della "Latitudine" e "Longitudine", ovvero le coordinate, del luogo dove avviene un evento. La definizione del luogo, dove avviene un evento, è presente nella classe creazione o modifica di un evento o nelle impostazioni predefinite di calendario, dove andiamo a definire il luogo con cui andare a precompilare gli eventi che stiamo creando appartenenti a quel calendario. Il valore della latitudine e longitudine di un luogo viene restituito dalla funzione "ConvertiIndirizzoToCoordinate(Indirizzo)" presente in "GestioneLuoghi", classe già presentata in DCL6.3.



DCL12 : Tipo Data

Nella classe "Notifica" (DCL6.2) è presente l'attributo "Data" di tipo "Tipo_Data", tipo di dato che non avevamo ancora descritto. Questo tipo di dato permette di andare a salvare, definire e ottenere il valore di ora, minuti, giorno, mese anno di quando deve essere inviata una notifica.
Le funzione set() e get() di "Ora", "Minuti", "Giorno", "Mese" e "Anno" non sono state inserite nel diagramma della classe perché ritenute banali.

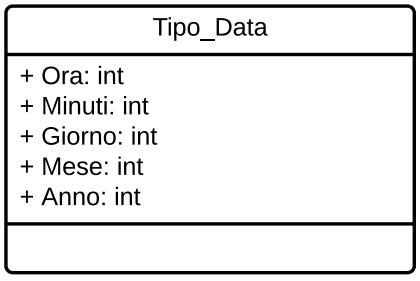
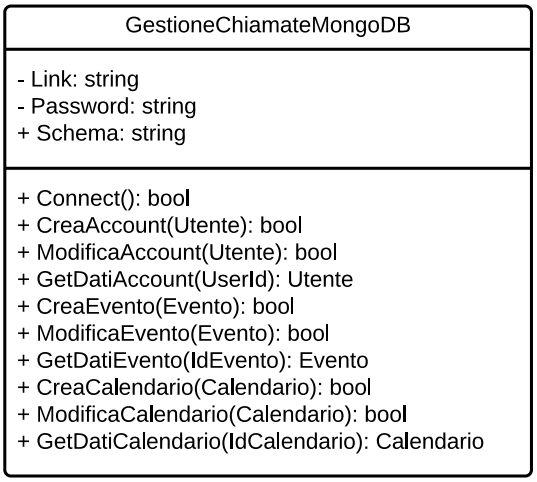


Immagine PNG/SVG Diagramma delle classi "Tipo Luogo"
Immagine PNG/SVG Diagramma delle classi "Tipo Data"

DCL13 : **Gestione chiamate MongoDB**

Il diagramma dei componenti analizzato, realizzato per il progetto PlanIt, presenta un componente “Gestione chiamate MongoDB”, per questo motivo è stata fatta questa classe, "GestioneChiamateMongoDB", il cui scopo è gestire, come dice il nome, le chiamate al sistema esterno MongoDB. Queste interazioni con MongoDB, possono essere sia di accesso a contenuti presenti nel database MongoDB, sia invio di dati a quest'ultimo. I dati che vengono salvati su MongoDB, e quindi gestiti nelle chiamate di questa classe, riguardano: i calendari, eventi e l'account (ovvero, "UserID" e "Email", "Username", attributi presenti in "UtenteAutenticato", DCL1 e le impostazioni utente, ovvero gli attributi presenti in “GestioneImpostazioniUtente”, DCL3, di un utente autenticato). Per questa ragione sono presenti, in questa classe, delle funzioni per permettere l'invio e la richiesta di ricezione di questa tipologia di dati.



DCL14 : **Tipo Link Richiesta**

Nella classe "Gestione condivisione" è presente il parametro "link", stringa che fa riferimento ad un oggetto di tipo "Tipo_Link_Richiesta", tipo di dato che non avevamo ancora descritto. L'oggetto a cui fa riferimento il “link” contiene: "Account", "IDEvento" o "IDCalendario". L'Account è fondamentale per poter individuare univocamente chi ha inviato la richiesta, invece "IDEvento" o "IDCalendario" sono fondamentali per definire cosa si sta condividendo. Infatti, nel caso in cui si accettasse la richiesta, il sistema deve sapere a quale oggetto “Evento” o “Calendario”, presente su MongoDB (in quanto questi oggetti vengono salvati su questo sistema esterno di archiviazione), si sta facendo riferimento; infatti, nel database c'è una singola istanza per ciascuno oggetto che è stato condiviso.
Ottenendo il riferimento a tale oggetto di tipo “Tipo_Link_Richiesta”, l' “UtenteAutenticato”, a cui abbiamo inviato l'invito alla condivisione, può accettare ("accettaRichiesta()") o meno ("rifiutaRichiesta()") la condivisione. Nel caso in cui accettasse la condivisione, come specificato in (DCL6.4), tale “UtenteAutenticato” viene aggiunto all'attributo “Partecipanti” dell'oggetto che si stava condividendo e l' “Evento” o “Calendario” accettato viene aggiunto nella lista di eventi condivisi o “Calendari”, a seconda se si è ricevuto la richiesta di condivisione per un “Calendario” o “Evento, del proprio account PlanIt.

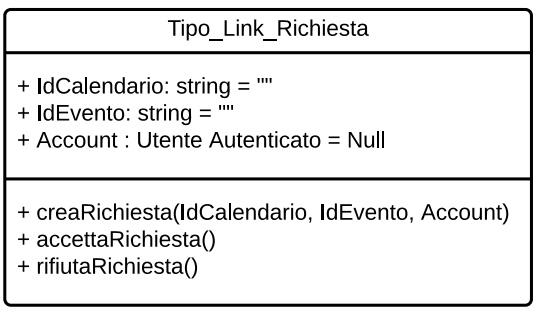


Immagine PNG/SVG Diagramma delle classi "Gestione chiamate MongoDB"
Immagine PNG/SVG Diagramma delle classi "Tipo Link Richiesta"

DCL15 : **Diagramma delle classi complessivo**

Riportiamo di seguito il diagramma complessivo, con tutte le classi sopra descritte.
Il primo grafico riportato mostra tutto il diagramma delle classi comprendendo sia i tipi di dati e anche classi che sono scollegate tra loro. Il secondo grafico riportato mostra la maggior parte delle classi che abbiamo collegato tra loro.
Il terzo grafico riportato mostra le classi che gestiscono l' "Homepage".
Il quarto grafico riportato mostra le classi che gestiscono l' interfaccia di Iubenda.
Il quinto grafico riportato mostra dei tipi di dati usati nella classi

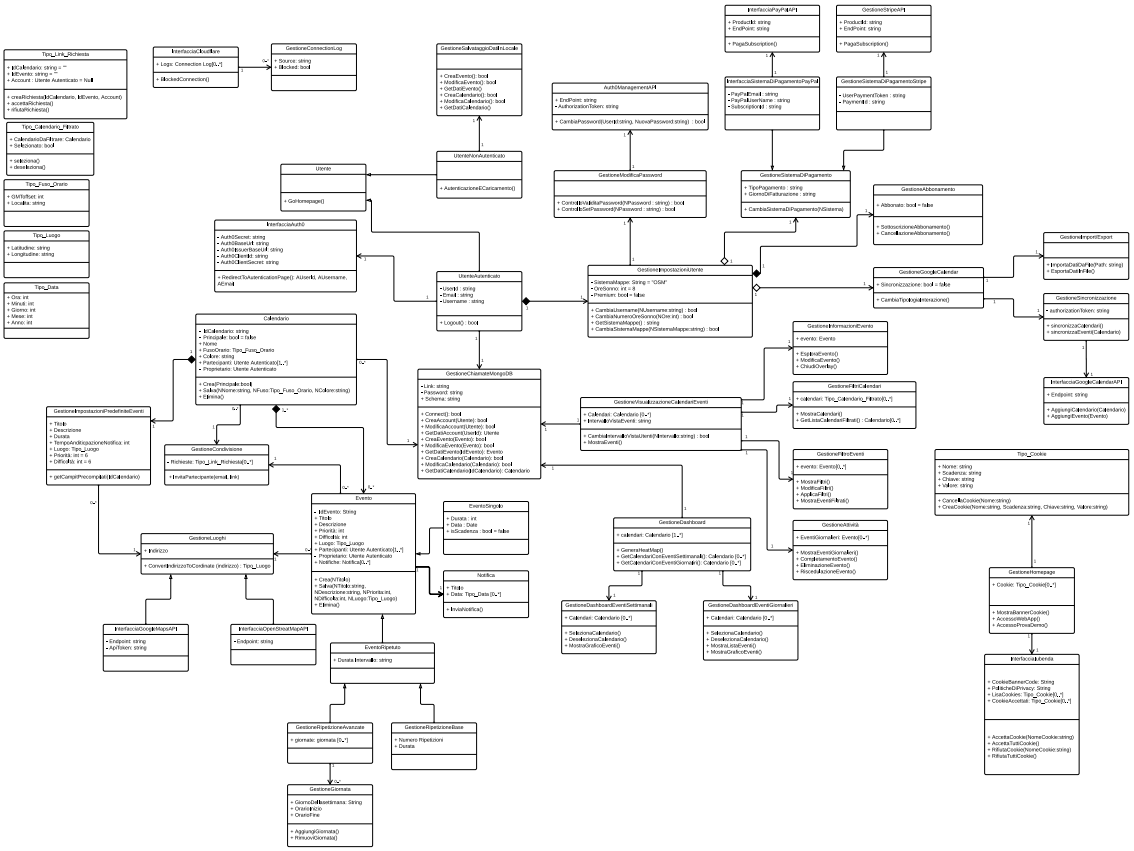
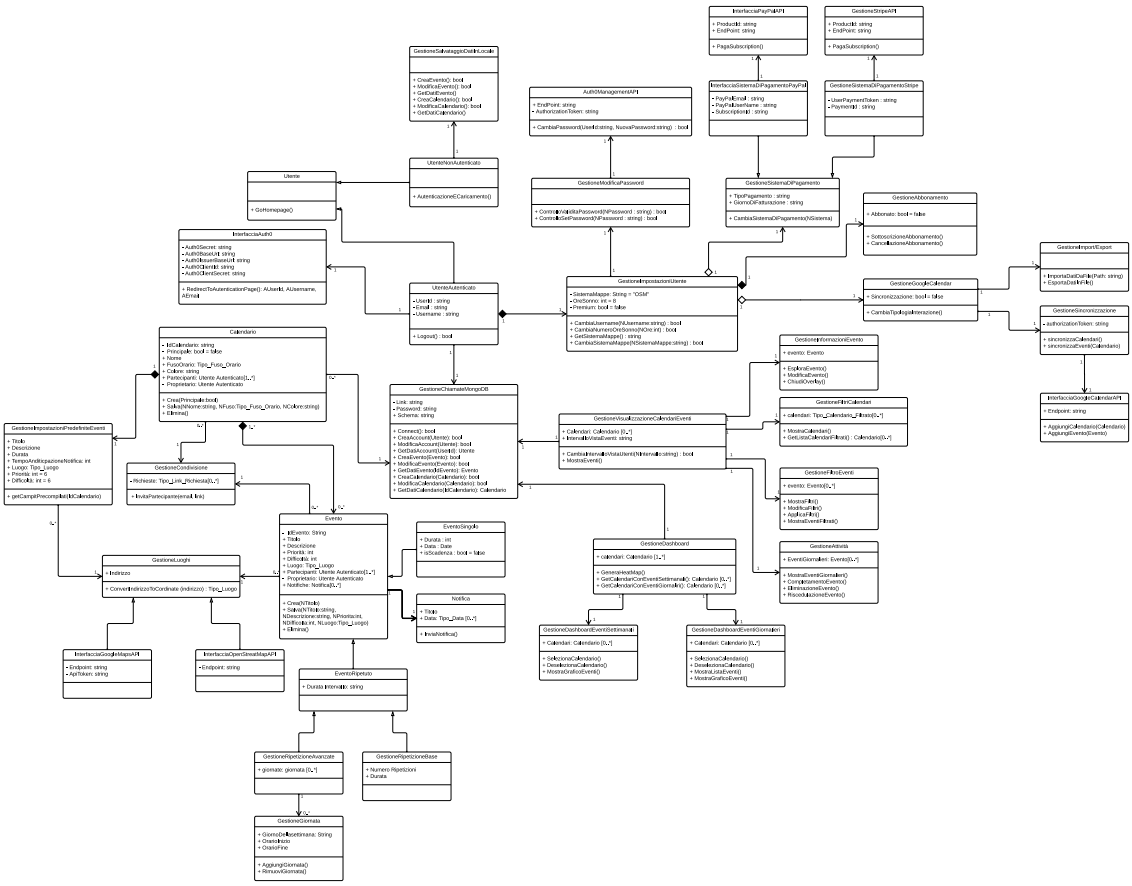


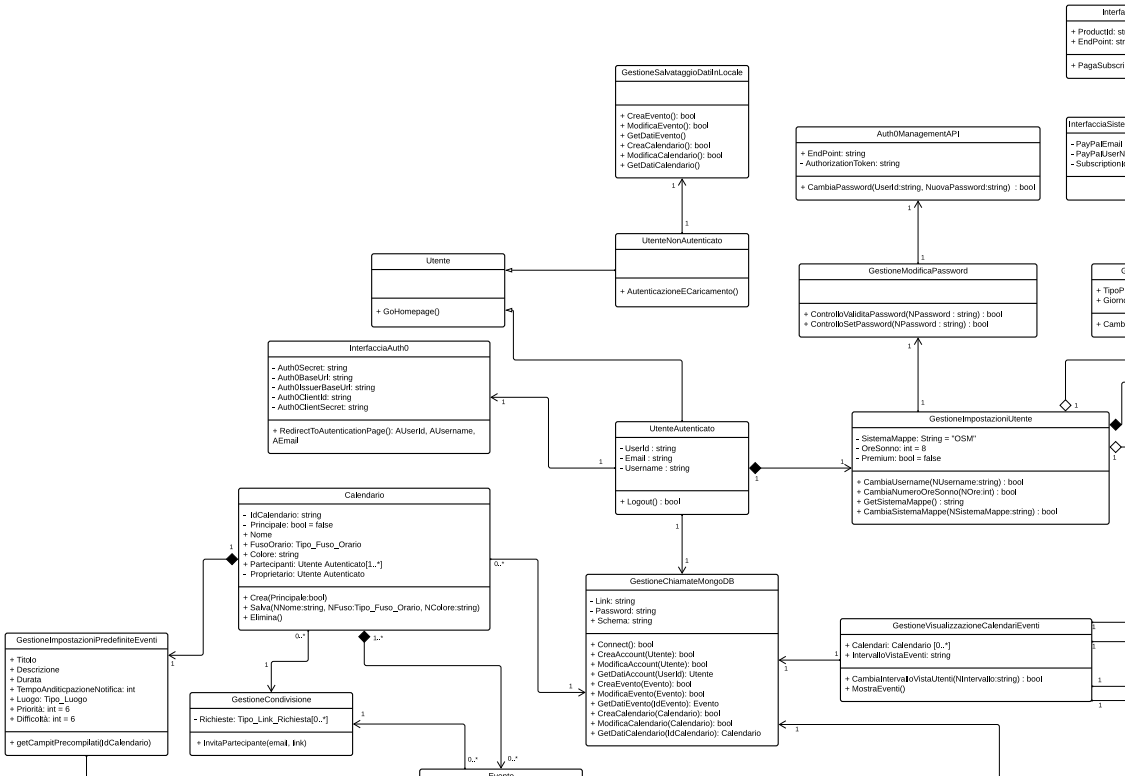
Diagramma delle classi, grafico totale

Immagine PNG/SVG Diagramma delle classi, grafico totale

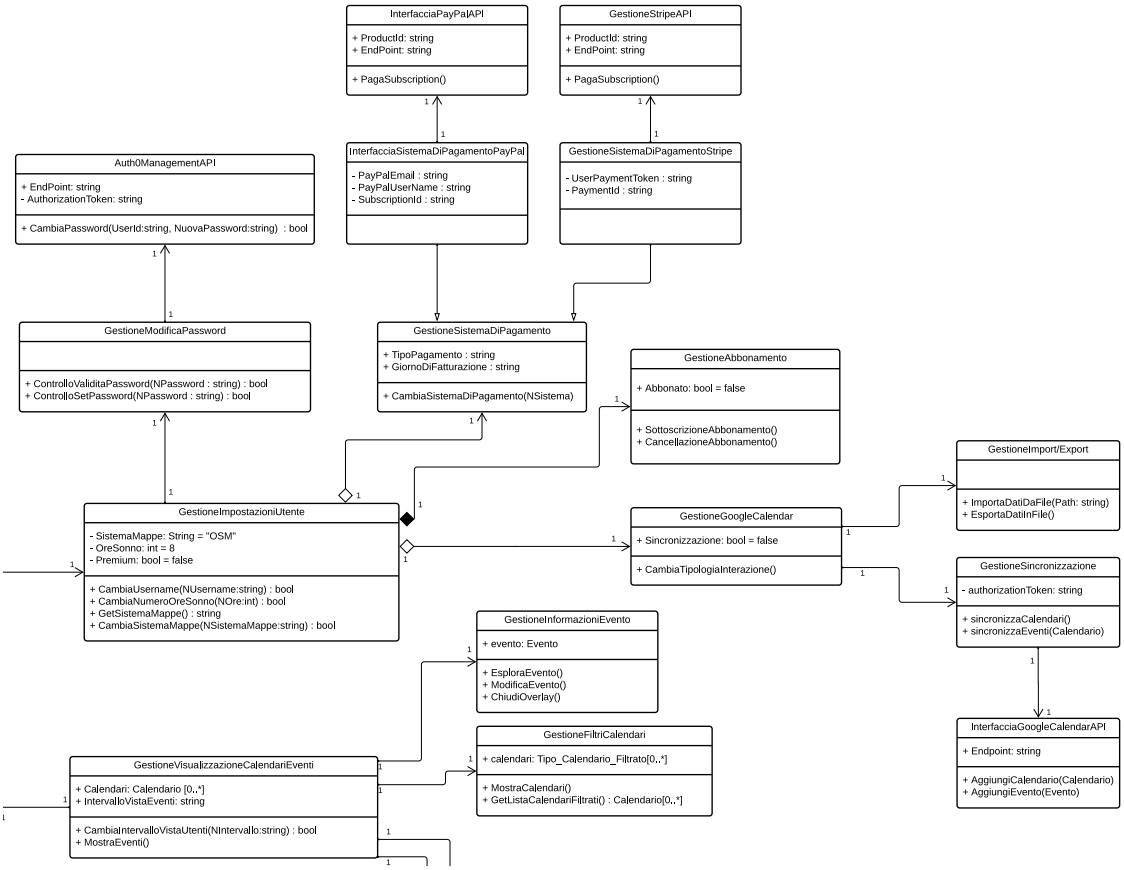


Secondo grafico, visuale totale

Immagine PNG/SVG Secondo grafico, visuale totale

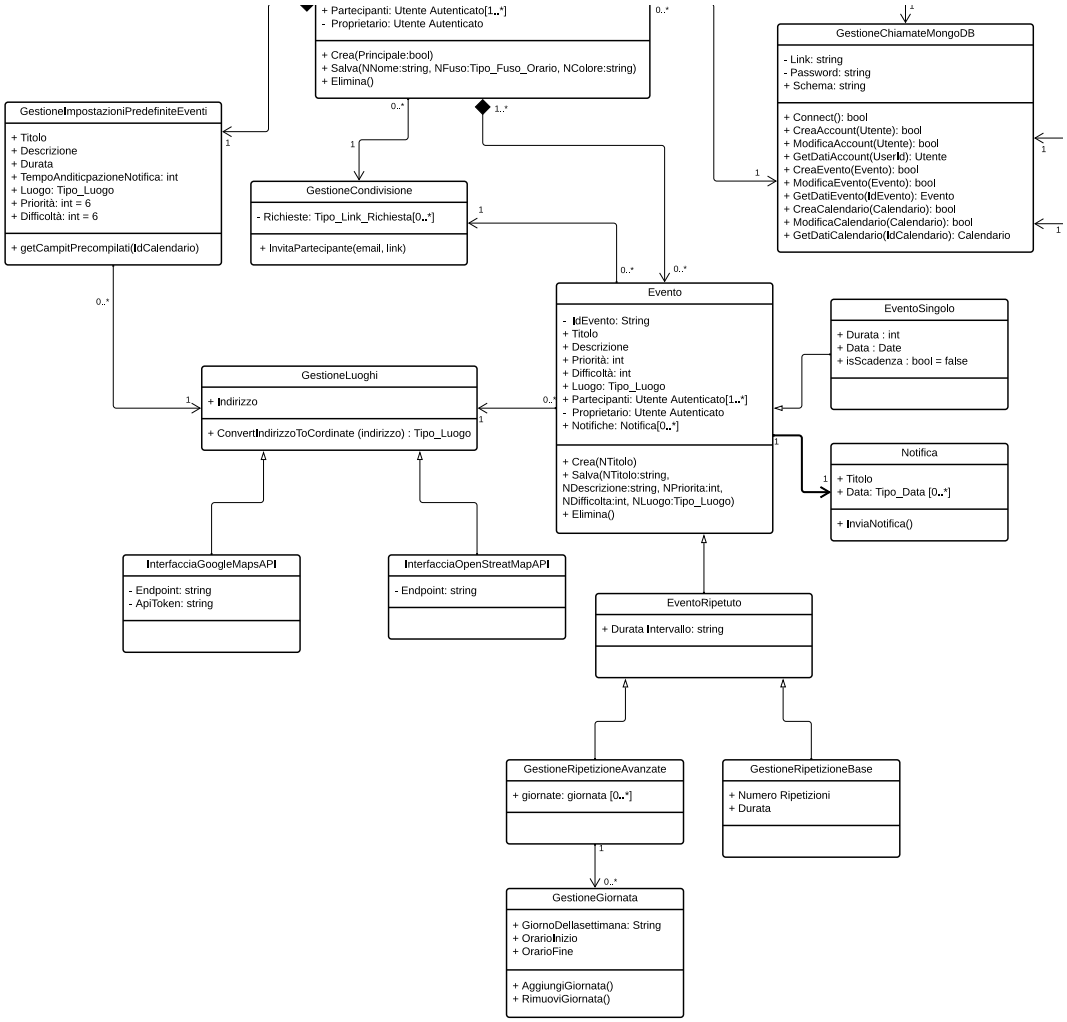


Secondo grafico, visuale in alto a sinistra

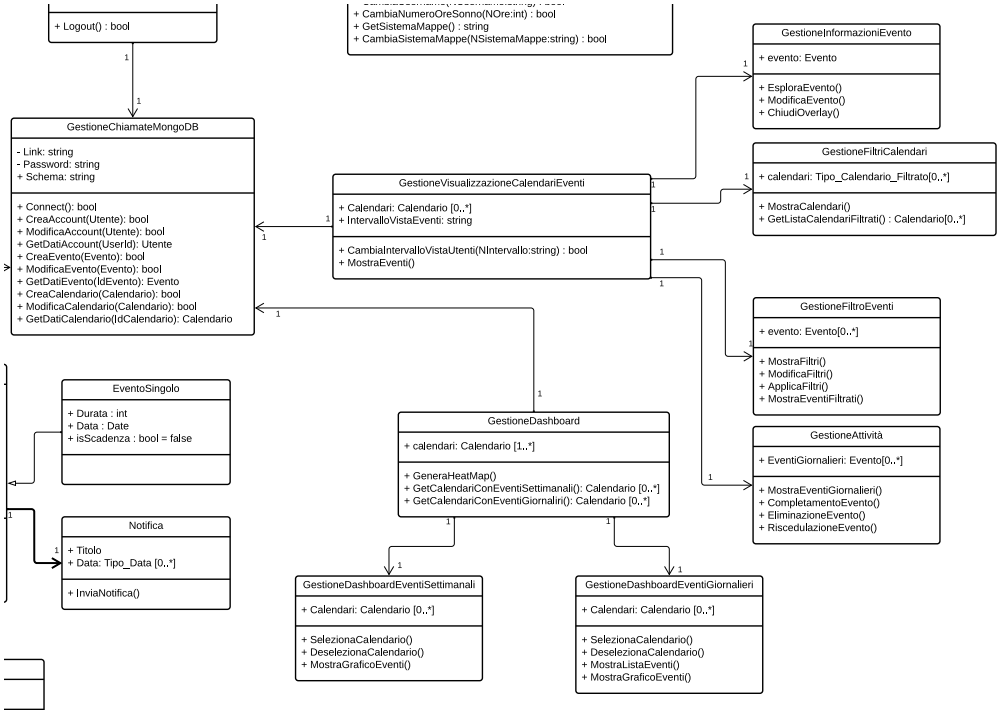


Secondo grafico, visuale in alto a destra

Immagine PNG/SVG Secondo grafico, visuale in alto a sinistra
Immagine PNG/SVG Secondo grafico, visuale in alto a destra

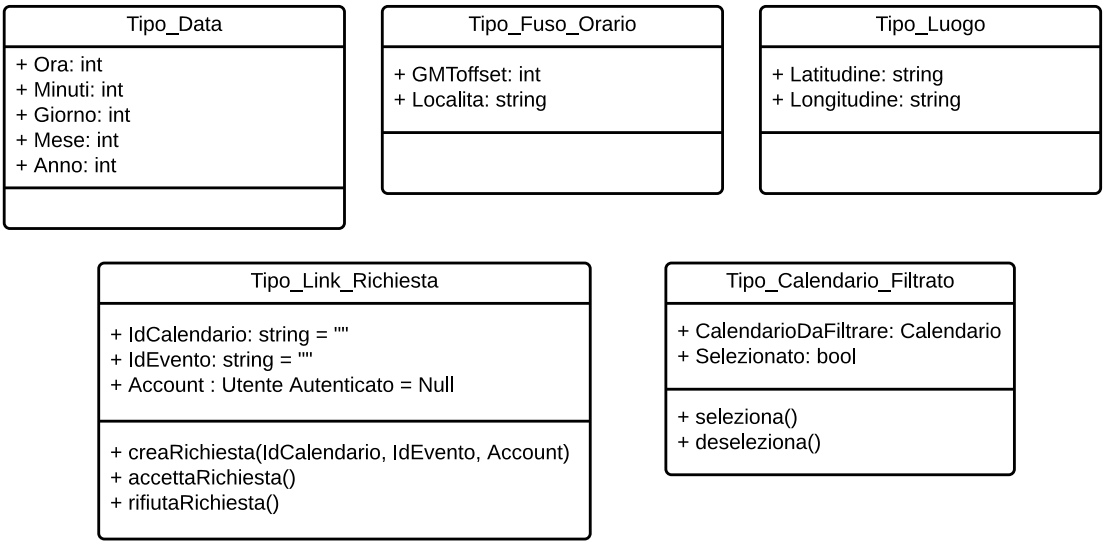


Secondo grafico, visuale in basso a sinistra

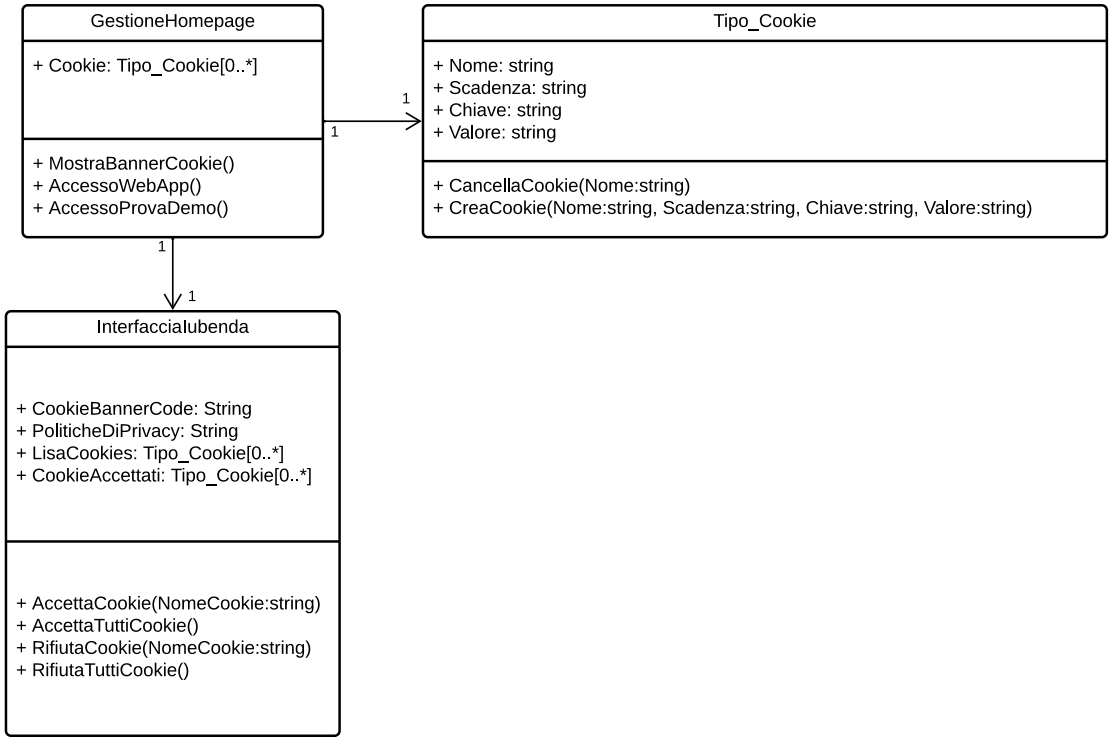


Secondo grafico, visuale in basso a destra

Immagine PNG/SVG Secondo grafico, visuale in basso a sinistra
Immagine PNG/SVG Secondo grafico, visuale in basso a destra

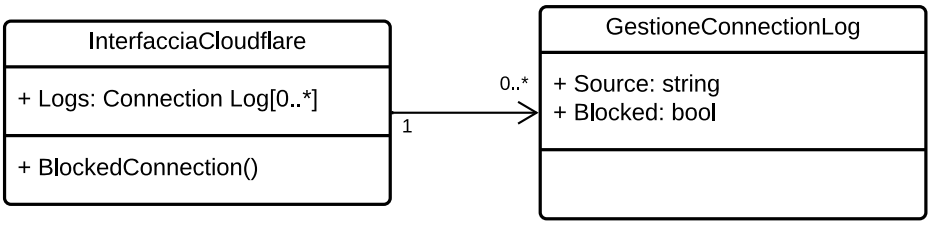


Terzo grafico, Tipi di dati



Quarto grafico, Gestione Homepage

Immagine PNG/SVG Diagramma delle classi, Tipi di dati
Immagine PNG/SVG Diagramma delle classi "Homepage"



Quinto grafico, Interfaccia CloudFlare

Immagine PNG/SVG Diagramma delle classi "Interfaccia CloudFlare"

2 Object Constraint Language

In questo capitolo è descritta in modo formale la logica prevista nell'ambito di alcune operazioni di alcune classi. Tale logica viene descritta in Object Constraint Language (OCL) perché tali concetti non sono esprimibili in nessun altro modo formale nel contesto di UML.

OCL1 : **Utenti**

Nella classe "Utente_Non_Autenticato" (DLC1.1) è presente il metodo "Autenticazione()", la cui esecuzione comporta l'inizializzazione degli attributi UserId, Email e Username, a dei valori non vuoti; infatti grazie a questo metodo l' utente non autenticato diventa utente autenticato.
Questa condizione è espressa in OCL attraverso una postcondizione con questo codice:

```
CONTEXT Utente_Non_Autenticato::AutenticazioneECaricamento() :
POST : UserId <> "" AND Email <> "" AND Username <> ""
```

Dopo il metodo "Autenticazione()", ci sono tre condizioni che si verificano per ogni "Utente_Autenticato": UserId, Email e Username non devono essere vuoti.
Questa condizione è espressa in OCL, attraverso un invariante con questo codice:

```
CONTEXT Utente_Autenticato :
INV : UserId <> "" AND Email <> "" AND Username <> ""

CONTEXT Utente_Autenticato::Logout() :
POST : UserId  = "" AND Email  = "" AND Username  = ""
```

Nella classe "Utente_Autenticato" (DCL1.1) è presente il metodo "Logout()", la cui esecuzione avviene se e solo se l'UserId, l'Username e l'Email non sono vuoti, ovviamente questo avviene sempre per ogni utente autenticato, in quanto gli utenti autenticati non potrebbero essere definiti tali con gli attributi, sopra citati, vuoti.
Dopo l'esecuzione del Logout(), ci sono tre condizioni che si verificano: UserId, Email e Username sono vuoti, in quanto l'utente autenticato è uscito dal sito e non più attivo su esso.
Queste precondizioni e postcondizioni sono rappresentati con il seguente codice OCL:

Non so quanto senso abbia mettere postcondizione per il logout

da chiedere

OCL2 : **Auth0**

Nella classe "Interfaccia_Auth0" (DLC1.2) è presente il metodo "RedirectToAutenticationPage()", la cui esecuzione restituisce i valori "AUserId", "AUsername" e "AEmail" che vengono attribuiti ai rispettivi attributi presenti in "Utente_Autenticato".
Questa condizione è espressa in OCL attraverso una postcondizione con questo codice:

```
CONTEXT Interfaccia_Auth0:
INV : Auth0Secret <> "" AND Auth0BaseUrl <> "" AND Auth0IssuerBaseUrl
    <> "" AND Auth0ClientId  <> "" AND Auth0ClientSecret<> ""

CONTEXT Interfaccia_Auth0::RedirectToAutenticationPage() :
POST : Utente_Autenticato.UserId = AUserId AND
    Utente_Autenticato.Email = AEmail AND Utente_Autenticato.Username
    = AUsername
```

OCL3 : **Impostazioni Utente**

Nella classe "ImpostazioniUtente" (DCL1.3) ci sono delle condizioni che devono essere sempre vere. Infatti l'attributo "SistemaMappe" deve essere uguale alla stringa "OSM", che si riferisce al sistema esterno di mappe OpenStreeMap, o alla stringa "GM", Google Maps. Infatti il sistema di mappe deve essere per forza scelto fra un di questi due.
Queste condizioni sono espresse in OCL attraverso la seguente espressione:

```
CONTEXT ImpostazioniUtente:
INV : (SistemaMappe = "OSM" OR SistemaMappe = "GM") AND OreSonno >= 0
    AND Premium = Abbonamento.Abbonato
```

Inoltre, in questa classe, è presente anche il metodo "CambiaUsername(NUsername)" per cui devono esserci delle condizioni che devono essere verificate prima dell'esecuzione. Infatti il nuovo username, NUsername, con cui si vuole modificare l'username presente, non deve essere una stringa vuota.
Questa condizione è espressa in OCL attraverso la seguente precondizione:

```
CONTEXT ImpostazioniUtente::CambiaUsername(NUsername):
PRE : NUsername <> "" AND Utente_Autenticato.Username <> NUsername
POST : Utente_Autenticato.Username = NUsername
```

Nella classe "ImpostazioneUtente" (DCL1.3) per il metodo "CambiaNumeroOreSonno (Nore)" c'è una precondizione che deve essere verificata, prima dell'esecuzione del metodo, ovvero che NOre, le nuove ore che stiamo inserendo con numero di ore di sonno, deve essere maggiore e uguale a 0 e diverso dalle ore di sonno già inserite in "ImpostazioniUtente", quindi in OCL abbiamo:

```
CONTEXT ImpostazioniUtente::CambiaNumeroOreSonno(Nore):
PRE : NOre >= 0 AND NOre <> self.OreSonno
POST : self.OreSonno = NOre
```

In seguito, in questa classe è presente il metodo "GetSistemaMappe()" per cui si verifica una postcondizione, ovvero che la stringa che viene restituita da questa funzione, che indica il metodo di sistema di mappe scelto dall'utente autenticato, deve essere uguale all'attributo "SistemaMappe" presente in "ImpostazioniUtente".
Questa postcondizione è espressa dal seguente codice OCL:

```
CONTEXT ImpostazioniUtente::GetSistemaMappe():
POST : return = SistemaMappe
```

Infine, nella classe "ImpostazioniUtente" (DCL1.3) per il metodo "CambiaSistemaMappe(NSistemaMappe)" c'è una precondizione che deve essere verificata, prima dell'esecuzione del metodo, ovvero che il nuovo sistema di mappe scelto dall'Utente, NSistemaMappe, e il sistema di mappe presente ora in "ImpostazioniUtente" deve essere uguale a "OSM" (stringa che si riferisce a OpenStreetMap) o "GM" (Google Maps). Dopo, l'esecuzione del metodo si verifica una postcondizione; infatti dopo la modifica del sistema di mappe, il sistema di mappe presente nell'attributo in "SistemaMappe" è uguale al sistema di mappe nuovo, NSistemaMappe, che è stato usato come parametro nel metodo "CambiaSistemaMappe(NSistemaMappe).
Queste condizioni nel contesto, "CambiaSistemaMappe(NSistemaMappe), sono espresse con il seguente codice OCL:

```
CONTEXT ImpostazioniUtente::CambiaSistemaMappe(NSistemaMappe):
PRE : (NSistemaMappe = "OSM" AND self.SistemaMappe = "GM") OR
    (NSistemaMappe = "GM" AND self.SistemaMappe = "OSM")
POST : self.SistemaMappe = NSistemaMappe
```

da chiedere

OCL4 : Gestione Modifica Password

Nella classe "GestioneModificaPassword" (DCL1.3.1) c'è una condizione che deve essere sempre verificata, ovvero che l'attributo "NuovaPassword" non deve essere vuoto.
Per questo motivo, è stato fatto il seguente codice OCL per esprimere la seguente condizione:
Inoltre, in questa classe è presente il metodo "ControlloValiditaPassword(NPassword) il quale,dopo aver eseguito, ritorna un booleano. Il valore di questo dipende dal fatto se la nuova password inserita, NuovaPassword, segua o meno gli standard già presentati in D2-RNF2 (anche D1-RNF2.1).

non c'e piu l'invariante

```
CONTEXT
    GestioneModificaPassword::ControlloValiditaPassword(NPassword) :
POST : return = NPassword.size() >= 8 AND NPassword.exists(carattere
    | carattere = [a, ..., z]) AND NPassword.exists(carattere |
```

```
    carattere = [A, ..., Z]) AND NPassword.exists(carattere |
    carattere = [0, ..., 9]) AND NPassword.exists(carattere |
    carattere = [!, #, \$, \%, \&, \*])
```

Dopo l'esecuzione del metodo "ControlloSetPassword(NPassword)", nella classe "GestioneModificaPassword" si verifica una condizione, più specificamente una postcondizione, visto che avviene dopo l'esecuzione del metodo sopra citato. Infatti l'attributo "nuovaPassword" ha il valore di Npassword, parametro, meglio dire password, utilizzato per modificare la password di un utente autenticato.
Questa condizione è espressa con il seguente codice OCL:

```
CONTEXT GestioneModificaPassword::ControlloSetPassword(NPassword) :
PRE : self.ControlloValiditaPassword(NPassword)
POST : Auth0ManagementAPI.CambiaPassword (Utente_Autenticato.UserId,
    NPassword)
```

OCL5 : **Auth0 management API**
Nella classe "Auth0 management API" (DCL1.3.1) c'è una condizione che deve essere sempre verificata, ovvero che l'attributo "EndPoint", stringa che indica l'URL di Auth0 a cui posso accedere per modificare la password, e "AuthorizationToken", token dato da Auth0 agli utenti autenticati per accedere alle API di Auth0, non devono essere vuoti.
Questa invariante è espressa da OCL mediante la seguente espressione OCL:

```
CONTEXT Auth0ManagementAPI :
INV : EndPoint <> "" AND AuthorizationToken <> ""
```

Inoltre, in questa classe è presente il metodo "CambiaPassword(UserId, NuovaPassword)", il quale, per eseguire, deve essere verificata una precondizione, ovvero che UserId, Id dell'utente autenticato che sta cambiando la password, non deve essere vuota. Inoltre, il metodo "ControlloValiditàPassword(Nuovapassword)" della classe "GestioneModificaPassword", mi deve aver restituito un valore "true".
Questa precondizione è specificata dal seguente codice OCL:

```
CONTEXT Auth0ManagementAPI::CambiaPassword(UserId, NuovaPassword) :
PRE : UserId <> "" AND
    GestioneModificaPassword.ControlloValidiaPassword(NuovaPassword)
```

OCL6 : **Abbonamento**
Nella classe "Abbonamento", è presente il metodo "SottoscrizioneAbbonamento()". Affinché possa avvenire la sottoscrizione all'abbonamento deve essere verificata una precondizione, ovvero che l'attributo "Abbonato", presente in questa classe, sia uguale a "false". Dopo l'esecuzione di tale metodo si verifica una postcondizione: l'attributo "Abbonato" è uguale a "true".
Questa pre e post-condizione sono espressi in OCL con il seguente codice:

```
CONTEXT Abbonamento::SottoscrizioneAbbonamento() :
PRE : Abbonato = false
POST : Abbonato = true
```

Inoltre, nella classe "Abbonamento", è presente anche il metodo "CancellazioneAbbonamento()". Affinché possa avvenire la camcellazione all'abbonamento deve essere verificata una precondizione, ovvero che l'attributo "Abbonato", presente in questa classe, sia uguale a "true". Dopo l'esecuzione di tale metodo si verifica una postcondizione: l'attributo "Abbonato" è uguale a "false"; infatti l'abbonamento viene annullato.
Questa pre e post-condizione sono espressi in OCL con il seguente codice:

```
CONTEXT Abbonamento::CancellazioneAbbonamento() :
PRE : Abbonato = true
```

```
POST : Abbonato = false
```

OCL7 : **Visualizzazione calendari ed eventi**

da sistemare

```
CONTEXT VisualizzazioneCalendariEventi :
INV : IntervalloVistaEventi = "Giorno" OR IntervalloVistaEventi =
    "Settimana" OR IntervalloVistaEventi = "Mese" AND Calendari =
    GestioneFiltriCalendari.GetListaCalendariFiltrati()
```

Nella classe "Visualizzazione Calendario", con l'esecuzione del metodo "CambiaIntervalloVistaUtenti(NIntervallo)" si ha che l'attributo di questa classe "IntervalloVistaUtenti" è uguale al parametro "NIntervallo" utilizzato nel metodo sopra citato.
Questa postcondizione è espressa con il seguente codice OCL:

```
CONTEXT VisualizzazioneCalendariEventi::
    CambiaIntervalloVistaUtenti(NIntervallo) :
PRE : NIntervallo = "Giorno" OR NIntervallo = "Settimana" OR
    NIntervallo = "Mese"
POST : IntervalloVistaEventi = NIntervallo
```

OCL8 : **Calendario**
Nella classe "Calendario" deve essere verificata sempre una condizione, ossia che l'IdCalendario, Id che identifica univocamente un calendario nel database MongoDB, e il suo nome non devono essere stringhe vuote. Inoltre, anche il proprietario di tale calendario deve essere diverso da "Null".
Queste invarianti presentati, sono espressi dal seguente codice OCL:

```
CONTEXT Calendario :
INV : IdCalendario <> "" AND Nome <> "" AND Proprietario <> Null
```

Inoltre, in questa classe è presente un metodo "Crea(Principale)" (il parametro e attributo "Principale" serve ad indicare se stiamo andando a modificare il calendario principale o uno degli altri calendari di un utente) che, con la sua esecuzione, fa in modo che l'attributo "Proprietario" della classe "Calendario" sia uguale ad "UserId", Id dell'utente che ha creato tale calendario.
Questa postcondizione è presenta dall'OCL:

```
CONTEXT Calendario::Crea(Principale)
POST : Proprietario = Utente_Autenticato.UserId AND Partecipanti ->
    exists(u | u.UserId = Utente_Autenticato.UserId )
```

Infine, con il metodo "Salva(NNome,NFuso,NColore) andiamo a modificare i seguenti attributi della classe "Calendario": "Nome", "Fuso", "Colore". Infatti questi attributi ottengono il valore dei nuovi attributi assegnati con il metodo salva, ovvero NNome, NFuso e NColore.
Questa postcondizione è espressa in OCL nel seguente modo:

```
CONTEXT Calendario::Salva(NNome, NFuso, NColore)
POST : Nome = NNome AND Fuso = NFuso AND NColore = Colore
```

OCL9 : **Evento**
Nella classe "Evento", ci sono delle condizioni che devono essere sempre verificate, ovvero che l' "IdEvento", Id che identifica univocamente un evento un calendario nel database MongoDB, e il "Titolo" non devono essere vuoti; inoltre, anche il proprietario, ovvero l'utente autenticato che sta creando l'evento, deve essere diverso da "Null", dunque in OCL si ha:

```
CONTEXT Evento :
INV : IdEvento <> "" AND Proprietario <> Null AND Titolo <> ""
```

In questa classe, è presente il metodo "Crea(NTitolo). Affinchè quest'ultimo possa eseguire, "NTitolo", ovvero il titolo che si vuole attribuire all'evento che si sta creando, deve essere diverso dalla stringa vuota. Infine, dopo l'esecuzione di questo metodo, l'attributo "Proprietario" deve essere uguale ad UserId, Id dell'utente autenticato che sta creando l'evento, e "Titolo" a "NTitolo", il titolo scelto per l'evento.
Tali condizioni sono presentati dal seguente codice OCL:

```
CONTEXT Evento::Crea(NTitolo)
PRE : Ntitolo <> ""
POST : Proprietario = Utente_Autenticato.UserId AND Titolo = NTitolo
    AND Partecipanti -> exists(u | u.UserId =
    Utente_Autenticato.UserId )
```

Infine, in questa classe è presente il metodo "Salva(NTitolo , NDescrizione , NPriorita , NDifficolta , NLuogo)". Per questo metodo, per eseguire, si deve verificare, come condizione, che il titolo inserito per l'Evento da salvare sia diverso dalla stringa vuota.
Dopo l'esecuzione del metodo, gli attributi dell'oggetto "Evento" saranno uguali ai parametri inseri in tale metodo "Salva".
Queste condizioni sono rappresentante dal seguente codice OCL:

```
CONTEXT Evento::Salva(NTitolo, NDescrizione, NPriorita, NDifficolta,
    NLuogo)
PRE : Ntitolo <> ""
POST : Titolo = NTitolo AND Descrizione = NDescrizione AND Priorita =
    NPriorita AND Difficolta = NDifficolta AND Luogo = NLuogo
```

OCL10 : **Gestione Chiamate MongoDB**
Nella classe "Gestione Chiamate MongoDB" ci sono delle condizioni che devono essere sempre verificate; infatti l'attributo "Link", attributo in cui è salvato l'URL del database MongoDB con cui interagisce questa classe, e la "Password", necessaria per poter accedere e interagire sempre con il database MongoDB, non devono essere fuori. Infatti, se lo fossero, la classe "Gestione Chiamate MongoDB" non potrebbe gestire le chiamate tra MongoDB e l'applicativo.
Queste condizioni sono presentate con del codice OCL nel seguente modo:

```
CONTEXT GestioneChiamateMongoDB :
INV : Link <> "" AND Password <> ""
```

OCL11 : **Tipo Data**
Nella classe "Tipo_Data" sono presenti delle condizioni che devono essere verificate, ovvero che gli attributi "Ora", "Minuti", "Giorno" e "Mese" devono avere dei valori che siano coerenti agli intervalli temporali imposti dalla convenzione sul tempo.
Tali condizione sono presentati dal seguente codice OCL:

```
CONTEXT Tipo_Data:
INV : Ora >= 0 AND Ora <= 23 AND
Minuti >= 0 AND Minuti <= 60 AND
Giorno >= 1 AND Giorno <= 31
Mese >= 1 AND Mese <= 12
```

OCL12 : **Tipo Luogo**
Anche nella classe "Tipo_Luogo" sono presenti dei requisiti che devono essere sempre rispettati. Gli attributi "Latitudine" e "Longitudine" non devono essere delle stringhe vuote.
Queste condizioni sono descritte in OCL nel seguente modo:

```
CONTEXT Tipo_Luogo:
INV : Latitudine <> "" AND Longitudine <> ""
```

OCL13 : **Tipo Fuso Orario**

In questa classe, ci sono delle condizioni che devono essere sempre rispettate, ovvero che l'attributo "GMToffset" abbia un valore che sia coerente con l'intervallo di valori che può assumere l'offset di un fuso orario rispetto a Greenwich secondo le convenzioni poste.
Dunque, è stato fatto questo codice OCL:

```
CONTEXT Tipo_Fuso_Orario:
INV : GMToffset >= -12 AND GMToffset <= 12
```

OCL14 : **Tipo Calendario Filtrato**

Anche la classe "Tipo_Calendario_Filtrato" presenta delle condizioni che devono essere sempre verificate, cioè che l'attributo "CalendarioFiltrato", che contiene un calendario dell'utente autenticato, deve essere diverso da "Null". Infatti questo tipo di dato è utilizzato per andare a filtrare i calendari posseduti da un utente autenticato, dunque per forza deve essere diverso da "Null".
Queste condizioni sono presentate dal seguente codice OCL:

```
CONTEXT Tipo_Calendario_Filtrato
INV: CalendarioFiltato <> Null
```

OCL15 : **Tipo Link Richiesta**

La classe "Tipo_Link_Richiesta" ha dei requisiti riguardo a dei suoi attributi che devono essere sempre rispettati. Infatti, questo tipo di dato, utilizzato per fare le richieste di condivisione di eventi o calendari, non può avere entrambe le stringhe "IdCalendario", id utlizzato per identificare univocamente un calendario, "IdEvento", id che identifica univocamente un Evento, vuote. Inoltre anche "Account" deve avere un valore diverso da "Null". Questo è necessario, perché un oggetto di tipo "Tipo_Link_Richiesta" viene istanziato se e solo se si vuole condividere effettivamente un calendario o evento che si creando o modificando appartenente ad un utente autenticato avente, ovviamente, un account.
Queste condizioni sono rappresentate dall'OCL:

```
CONTEXT Tipo_Link_Richiesta:
INV : (IdCalendario <> "" AND IdEvento = "") OR (IdCalendario = ""
    AND IdEvento <> "") AND Account <> Null
```

Inoltre, in questa classe, nel contesto del metodo "creaRichiesta (IdCalendario, IdEvento, Account) ci sono delle precondizioni che devono essere rispettate prima di una sua esecuzione. Infatti, prima dell'esecuzione del metodo, devono essere rispettate le condizioni già citate nelle invarianti, ma che stavolta sono imposte sui parametri di questa funzione. Si noti che i parametri della funzione sono diversi dagli attributi, prima dell'esecuzione di questa funzione. Infine, dopo l'esecuzione di questo metodo, abbiamo delle postcondizioni. Infatti, dopo che avviene questa funzione, che può essere intesa come un vero e proprio costruttore di un oggetto di questo tipo di dato, gli attributi di un oggetto di questa classe hanno assumono i valori inseriti come parametri per il metodo "creaRichiesta".
Queste postcondizioni e precondizioni sono espresse nel seguente codice OCL:

```
CONTEXT Tipo_Link_Richiesta::creaRichiesta(IdCalendario, IdEvento,
    Account)
PRE : (IdCalendario <> "" AND IdEvento = "") OR (IdCalendario = ""
    AND IdEvento <> "") AND Account <> Null
POST : self.IdCalendario = IdCalendario AND self.IdEvento = IdEvento
    AND  self.Account = Account
```

OCL16 : **Gestione Luoghi**
Questa classe ha delle precondizioni che devono essere rispettate per l'esecuzione del metodo "ConvertIndirizzoToCordinate(indirizzo)", ovvero che l'indirizzo che deve essere convertito in delle coordinate geografiche deve essere diverso da una stringa vuota.
Questa precondizione è espressa nel codice OCL:

```
Context GestioneLuuoghi::ConvertIndirizzoToCordinate (indirizzo):
PRE : indirizzo <> ""
```

OCL17 : **Google Maps API**
La classe "Google Maps API" ha delle invarianti che devono essere sempre verificate, ovvero che l' "EndPoint", URL con cui si accede alle API di Google Maps, e ApiToken, che è il "token" con cui possiamo accedere alle API di Google Maps, non devono essere stringhe vuote.
Queste invarianti sono descritte con questo codice OCL:

```
CONTEXT GoogleMapsAPI :
INV : Endpoint <> "" AND ApiToken <> ""
```

OCL18 : **OpenStreatMap API**
La classe "OpenStreetMap API" ha un' invariante che deve essere sempre verificata, ovvero che l' "EndPoint", URL con cui si accede alle API di OpenStreetMap, non deve essere una stringa vuota.
Questa invariante è descritta in OCL:

```
CONTEXT OpenStreetMapAPI :
INV : Endpoint <> ""
```

OCL19 : **Impostazioni Ripetizione Base**
La classe "ImpostazioniRipetizioneBase" ha delle invarianti che devono essere sempre rispettate, ovvero che "NumeroRipetizioni" dell'evento, di cui stiamo impostando le ripetizioni, deve essere maggiore o uguale a 1 e che la durata di tale evento sia maggiore o uguale a 0.
Queste invarianti sono presentate dal seguente codice OCL:

```
CONTEXT ImpostazioniRipetizioneBase:
INV : NumeroRipetizioni >= 1 AND Durata >= 0
```

OCL20 : **Giornata**
La classe "Giornata", utilizzata per indicare il giorno in cui avviene un evento ripetuto, ha un'invariante; questa riguarda l'attributo "GiornoDellaSettimana", che ovviamente, deve assumere il valore di un giorno della settimana esistente.
Questa invariante è descritta dal seguente OCL:

```
CONTEXT Giornata:
INV : GiornoDellaSettimana = "L" OR
GiornoDellaSettimana = "Ma" OR
GiornoDellaSettimana = "Me" OR
GiornoDellaSettimana = "G" OR
GiornoDellaSettimana = "V" OR
GiornoDellaSettimana = "S" OR
GiornoDellaSettimana = "D"
```

OCL21 : **Evento Ripetuto**
La classe "Evento Ripetuto" ha un'invariante che riguarda l'attributo "DurateIntervallo", ovvero periodo di tempo per cui deve essere valido tale evento ripetuto. Infatti "DurataIntervallo" deve avere un valore di periodo temporale che sia valido, ovvero o durata di un giorno,

o mese, o settimana o anno.
Questo è il codice OCL che descrive questa invariante:

```
CONTEXT EventoRipetuto:
INV : DurataIntervallo = "Giorno" OR DurataIntervallo = "Settimana"
    OR DurataIntervallo = "Mese" OR
DurataIntervallo = "Anno"
```

OCL22 : **Evento Singolo**
La classe "Evento Singolo", che indica un evento che è valido solo per un giorno di cui indichiamo la data e la sua durata, ha delle condizioni che devono essere sempre rispettate. L'attributo "Durata" di un oggetto, istanza di questa classe, deve avere un valore maggiore o uguale a 0 e la "Data" deve essere diversa da "Null".
Per questo motivo è stato scritto questo codice OCL:

```
CONTEXT EventoSingolo:
INV : Durata >= 0 AND Data <> Null
```

OCL23 : **Sistema Di Pagamento**
Questa classe presenta delle invarianti che devono essere sempre verificate. "TipoPagamento" deve assumere un valore che sia uguale a "PayPal" o "Stripe", che sono i sistemi di pagamento messi a disposizione dall'applicativo per il pagamento dell'abbonamento.
Questa condizione è rappresentata dal seguente codice OCL:

```
CONTEXT SistemaDiPagamento :
INV : TipoPagamento = "PayPal" OR TipoPagamento = "Stripe"

CONTEXT SistemaDiPagamento::CambiaSistemaDiPagamento(NSistema) :
POST : TipoPagamento = NSistema
```

OCL24 : **Stripe API**
La classe "Stripe API" ha delle condizioni che devono essere sempre verificate; infatti l'attributo "ProductId", Id che indica il prodotto abbonamento che si sta acquistando e che sarà presente nella lista di abbonamenti del proprio account di Stripe, una volta che la sottoscrizione all'account premium sarà effettuata, e l' "EndPoint", URL delle API di Stripe con cui l'applicativo va ad interagire, non devono essere delle stringhe vuote. Dunque, in codice OCL abbiamo:

```
CONTEXT StripeAPI :
INV : ProductId <> "" AND EndPoint <> ""
```

Infine, in questa classe, nel contesto del metodo "PagaSubscription()" sono preesente delle postcondizioni. Infatti "UserPaymentToken", token che indica l'user dell'account Stripe con cui si va a pagare l'abbonamento, e "PaymentId", Id che identifica univocamente il prodotto, in questo caso l'abbonamento, che si sta acquistando, non devono essere delle stringhe vuote: se lo fossero l'acquisto dell'abbonamento non potrebbe concludersi con successo.
Queste postcondizioni sono espresse nel seguente codice OCL:

```
CONTEXT StripeAPI::PagaSubscription() :
POST : UserPaymentToken <> "" AND PaymentId <> ""
```

OCL25 : **PayPal API**
Nella classe, "PayPal API", come avviene nella classe "Stripe API" (OCL24), gli attributi "EndPoint" e "ProductId" non possono essere delle stringhe vuote, dunque esprimiamo queste invarianti nel seguente codice OCL:

```
CONTEXT PayPalAPI :
INV : ProductId <> "" AND EndPoint <> ""
```

Inoltre, in questa classe, nel contesto del metodo "PagaSubscrition()" è presente una post-condizione. Infatti, dopo l'esecuzione di questo metodo, gli attributi "PayPalEmail", "PayPalUserName" e "SubscriptionId", che corrispondono alle credenziali PayPal dell'utente autenticato, che decide di pagare l'abbonamento con questo metodo, non sono delle stringhe vuote.
Questa postcondizione è espressa in OCL nel seguente modo:

```
CONTEXT PayPalAPI::PagaSubscription() :
POST : PayPalEmail <> "" AND PayPalUserName <> "" AND SubscriptionId
    <> ""
```

OCL26 : **Abbonamento**
In questa classe, nel contesto del metodo "SottoscrizioneAbbonamento()", ci sono delle condizioni che devono essere rispettate prima dell'esecuzione del metodo. Infatti, per sottoscrivere un abbonamento, l'attributo "Abbonato" deve essere uguale a false.
Inoltre, ovviamente, dopo l'esecuzione del metodo "Abbonato" diventa uguale a true, in quanto si è sottoscritto l'abbonamento.
Questa precondizione e postcondizione sono presentate nel seguente modo con il codice OCL:

```
CONTEXT Abbonamento::SottoscrizioneAbbonamento() :
PRE Abbonato = false
POST Abbonato = true
```

Nella classe "Abbonamento", nel contesto del metodo "CancellazioneAbbonamento()" sono presenti le stesse pre e post-condizioni presentate per "SottoscrizioneAbbonamento()", ma, ovviamente, invertite. Infatti con questa funzione si fa esattamente il procedimento inverso.
Queste condizioni sono espresse in OCL nel seguente modo;

```
CONTEXT Abbonamento::CancellazioneAbbonamento() :
PRE Abbonato = true
POST Abbonato = false
```

OCL27 : **GoogleCalendar**
Nella classe "Google Calendar" c'è il metodo "CambiaTipologiaInterazione()". La sua esecuzione modifica il valore dell'attributo booleano "sincronizzazione" che diventa l'inverso del valore precedente all'esecuzione del metodo.
Tale postcondizione è rappresentata dal codice OCL:

```
CONTEXT Abbonamento::CambiaTipologiaInterazione() :
POST : sincronizzazione = NOT sincronizzazione@pre
```

OCL28 : **Gestione Import/Export**
Nella classe "Gestione Import/Export" ci sono delle condizioni che devono essere sempre verificate, infatti l'attributo "Sincronizzazione", booleano che indica se stiamo usando o meno la sincronizzazione come metodo di interazione con Google Calendar, deve essere uguale a "false". Infatti l'interazione con Google Calendar o è fatta mediante import/export di file o mediante la sincronizzazione automatica, non si possono fare entrambe contemporaneamente.
Questa invariante è presenta in OCL nel seguente modo:

```
CONTEXT GestioneImportExport :
INV : Sincronizzazione = false
```

OCL29 : **Gestione Sincronizzazione**
Nella classe "Gestione Sincronizzazione" sono presente delle condizioni che devono essere sempre rispettate. Infatti, contrariamente dalla classe "Gestione Import/Export", l'attributo "Sincronizzazione" deve essere uguale a "true"; questo indica che stiamo utilizzando la sincronizzazione come metodo di integrazione di Google Calendar con PlanIt. Inoltre, anche "authorizationToken", token necessario per poter accedere ad un account Google e, quindi, necessario per poter interagire con l'account Google Calendar di un utente, non deve essere una stringa vuota.
Queste inviarianti presentate sono descritte da questo codice OCL:

```
CONTEXT GestioneSincronizzazione :
INV : Sincronizzazione = true AND authorizationToken <> ""
```

OCL30 : **Google Calendar API**
In questa classe, è presente un'invariante che deve essere sempre valida; infatti l'attributo "EndPoint", URL delle API di Google Calendar, non deve essere una stringa vuota, cosa necessaria per poter interagire mediante sincronizzazione con l'applicativo Google Calendar.
Questa condizione è espressa nel seguente codice OCL:

```
CONTEXT GoogleCalendar :
INV : Endpoint <> ""
```

OCL31 : **Iubenda**
La classe "Iubenda" ha delle condizioni che devono essere sempre rispettate. Infatti l'attrivuto "CookieBannerCode", codice che mi serve per accedere al banner di cookie di Iubenda, e le "PolitricheDiPrivacy", stringa che contiene le politiche di privacy ottenute da Iubenda, non devono essere delle stringhe vuote.
Queste invarianti sono descritte dal seguente codice OCL:

```
CONTEXT Iubenda :
INV CookieBannerCode <> "" AND PolitricheDiPrivacy <> ""
```

Inoltre, in questa classe nel contesto del metodo "AccettaCookie(NomeCookie)", sono presenti delle postcondizioni. Infatti nel caso venisse accettato un cookie, questo viene aggiunto alla lista di cookie contenuto in "CookieAccettati"; per questo motivo il size di questo array aumenta di uno, rispetto al suo size prima dell'esecuzione del metodo. E, ovviamente, dopo aver accettato il cookie, questo sarà presente nell'array "CookieAccettati".
Queste postcondizioni sono descritte in OCL nel seguente modo:

```
CONTEXT Iubenda::AccettaCookie(NomeCookie) :
POST : CookieAccettati.size() = CookieAccettati.size()@pre +1 AND
    CookieAcettati -> exists(c | c.Nome = NomeCookie)
```

Nel contesto del metodo RifiutaCookie(NomeCookie), sono presenti delle postcondizioni. Infatti, in quanto tale "NomeCookie" viene rifiutato dall'utente, questo non viene aggiunto dall'array di cookie "CookieAccettati", dunque la dimensione di questa lista rimane la stessa e tale cookie non è presente in questa lista.
Queste postocondizioni sono descritte in OCL:

```
CONTEXT Iubenda::RifiutaCookie(NomeCookie) :
POST : CookieAccettati.size() = CookieAccettati.size()@pre AND NOT
    (CookieAcettati -> exists(c | c.Nome = NomeCookie))
```

In questa classe, nel contesto del metodo "AccettaTuttiCookies()", l'array di "CookieAccettati" viene inizializzata ottenendo tutti i cookie presenti in "ListaCookie", di conseguenza ha anche la sua stessa dimensione.
Tutto ciò sono delle postcondizioni del metodo "AccettaTuttiCookies()" che vengono descritte con il seguente codice OCL:

```
CONTEXT Iubenda::AccettaTuttiCookies() :
POST : CookieAccettati.size() = ListaCookie.size() AND ListaCookies
    -> forAll (lc | CookieAccettati -> exists(c | c.Nome = lc.Nome))
```

Infine, sempre nella classe "Iubenda" ma nel contesto del metodo "RifiutaTuttiCookies", come condizione dopo l'esecuzione di tale metodo, possiamo dire di avere un stato opposto descritto dopo l'esecuzione del metodo "AccettaTuttiCookies()". Infatti con questo metodo si vanno a rifiutare tutti i cookie presenti in "ListaCookies", dunque l'array "CookieAccettati" avrà una dimensione pari a zero.
Questa postcondizione è descritta dal seguente codice OCL:

```
CONTEXT Iubenda::RifiutaTuttiCookies() :
POST : CookieAccettati.size() = 0
```

OCL32 : **Tipo Cookie**
Nella classe "Tipo_Cookie", che serve per contenere un cookie che è stato accettato dall'utente, deve essere sempre valida una condizione, ovvero che l'attributo "Nome", stringa che contiene il nome, per l'appunto, del cookie accettato, non deve essere una stringa vuota.
Questa invariante è descritta dal seguente codice OCL:

```
CONTEXT Tipo_Cookies :
INV : Nome <> ""
```

In questa classe, nel contesto del metodo "CreaCookie(Nome, Scadenza, Chiave, Valore)" sono presenti del pre- e postcondizioni. Infatti, prima dell'esecuzione del metodo, il "Nome" del cookie, di cui stiamo andando a creare un oggetto di tipo "Tipo_Cookie", deve essere contenuto nell'array dei "CookieAccettati" presente nella classe Iubenda.
Dopo l'esecuzione di questo metodo "CreaCookie", tale oggetto di tipo "Tipo_Cookie" viene restituito alla classe "Homepage", dunque sarà presente nell' array "Cookie" presente nell' "Homepage".
Questa pre- e post-condizione sono presentate nel seguente codice OCL:

```
CONTEXT Tipo_Cookie::CreaCookie(Nome, Scadenza, Chiave, Valore):
PRE Iubenda.CookieAccettati -> exists(c | c.nome=Nome)
POST  Homepage.Cookie -> exists(c | c.nome=Nome AND c.Scadenza =
    Scadenza AND c.Chiave = Chiave AND c.Valore = Valore)
```

Infine, nel classe "Tipo_Cookie", nel contesto del metodo "CancellaCookie" è presente una postcondizione. Infatti, avendo un cookie contenuto in oggetto di tipo "Tipo_Cookie", questo cookie può essere rifiutato ogni volta che sia vuole mediante questo metodo. Dopo l'esecuzione di questo metodo, il cookie rifiutato verrà eliminato dalla lista cookie presente nell'attributo "Cookie" della classe "Homepage".
Questa postcondizione è descritta in OCL nel seguente modo:

```
CONTEXT Tipo_Cookie::CancellaCookie(Nome)
POST : NOT ( Homepage.Cookie -> exists (c | c.Nome = Nome))
```

3 Diagramma delle classi e codice Object Constraint Language

4 Legenda Riferimenti

riferimenti

Diagramma delle classi	DCL
Object Constraint Language	OCL