

PROGETTAZIONE

PROGETTAZIONE ARCHITETTURALE

Requisiti non Funzionali

Dall'analisi del problema sono emersi i seguenti requisiti non funzionali come più rilevanti:

- Estendibilità e manutenibilità
- Usabilità
- Sicurezza e privacy dei dati

Per il nostro sistema sono molto importanti i requisiti di estendibilità e manutenibilità perché si prevede in futuro di aggiungere nuove funzionalità e perché l'applicazione potrebbe venire usata successivamente da altre palestre con esigenze differenti. Perciò è importante adottare componenti piccoli, atomici e che possano essere modificati facilmente. È inoltre importante avere una progettazione orientata ad una astrazione dei modelli di dati.

Un altro requisito importante è quello dell'usabilità dell'applicazione perché potrebbe essere scarsa l'esperienza degli utenti con i mezzi informatici ed è quindi opportuno avere interfacce grafiche chiare e intuitive. Inoltre, l'usabilità non genera grossi conflitti con gli altri requisiti.

Per la sicurezza si ritiene opportuno l'uso di protocolli sicuri di cifratura delle comunicazioni come il TLS che è una soluzione standard, ampiamente usata e testata da molti soggetti terzi. Questo potrebbe portare a un lieve peggioramento delle prestazioni ma non avendo il sistema alcun vincolo di temporizzazione non appare un problema.

Quanto riguarda i dati degli utenti si pone molta attenzione alla privacy e protezione. Si adotta quindi una struttura a livelli per collocare i dati da proteggere nei layer più interni.

Scelta architetturale

Dal punto di vista architettonico si è scelto di adottare un'architettura client/server a tre livelli.

- *L1 – Client*

Si è deciso di sviluppare due client: uno per il gestore e uno per gli affiliati. Difatti queste due tipologie di utenti hanno funzionalità diverse tra loro e si ritiene opportuno ai fini di modularità, estendibilità e manutenibilità disaccoppiare i rispettivi client. Inoltre, come specificato nella "Tabella Minacce/Controlli" il gestore effettua l'accesso solo da una macchina sicura situata nei locali della palestra e quindi questo agevola la scelta di separare il suo client dagli altri. Tali client comunicano con il livello server tramite TLS per rispettare le esigenze di sicurezza.

- *L2 – Server*

Si è scelto di usare cinque server: uno per il gestore, uno per gli affiliati, uno per il login, uno per la generazione di ricevute, uno per i log. Per la comunicazione tra i livelli client e server si utilizza un broker come intermediario. La comunicazione tra client e broker avviene tramite HTTPS (che si basa su TLS). La comunicazione tra server e broker avviene tramite invocazione di metodi remoti. Così facendo il broker riceverà le richieste dai client e invocherà gli opportuni metodi remoti dei server per soddisfarle rimanendo a livello applicativo. Si decide di scomporre il più possibile a livello server per perseguire gli obiettivi di estendibilità, manutenibilità e modularità. Si preferisce quindi adottare componenti piccoli. Questi server sono protetti in una sottorete della rete locale della palestra e non possono essere acceduti da internet in quanto tutte le loro comunicazioni passano solo tramite il broker che fa da filtro. Il server per le ricevute è stato inserito in quanto la funzionalità di generare le ricevute è in comune sia agli affiliati che al gestore e perciò era opportuno metterla in un componente a sé stante anziché in uno dei due server di gestore o affiliato. Il server dei log permette al gestore di visualizzare i log, si occupa quindi di

reperirli dalla persistenza. Quanto riguarda la scrittura dei log ogni server produce le entry da sé quando opportuno aggiornando la persistenza. L'aggiunta di un broker permette un disaccoppiamento tra client e server che permette di avere una composizione di server libera e indipendente. Si è scelto di usare cinque server distinti per le esigenze di cui sopra, ma nulla vieterebbe di averne un numero diverso senza alterare i client. Se in futuro venissero introdotti server secondari replicati per fault-tolerance questo non intaccherebbe l'architettura.

- *L3 – Persistenza*

A tale livello sono presenti due server di persistenza: uno con il DBMS per i dati del sistema e l'altro per i log. Tale scelta è stata fatta affinché eventuali problemi di una macchina non alterino l'altra e per mantenere un approccio modulare. Si prevede anche qui un altro broker per filtrare le comunicazioni tra i livelli persistenza e server. I server soprastanti (L2) e tale broker comunicano tramite metodi remoti: i server invocano i metodi remoti del broker opportuni per effettuare le operazioni in modo indiretto sui DB. La scelta di adottare un broker anche qui è per avere disaccoppiamento tra livello server e livello persistenza. Si utilizza il pattern DAO in quanto permette un buon grado di disaccoppiamento tra logica di business e logica di persistenza agevolando la manutenibilità e riducendo la sovrapposizione di responsabilità. Con il pattern DAO affidiamo ad opportune classi l'interazione con il DB e questo si concilia con la presenza del broker nel quale inserire tali classi. Tramite una factory i server del livello due ottengono gli oggetti opportuni con i quali effettuare le operazioni remote.

A tale livello collociamo anche i sistemi esterni. Grazie al disaccoppiamento dato dal broker i server soprastanti sono totalmente ignari della distinzione tra DB e sistemi esterni: dal loro punto di vista le chiamate al broker sono indistinte. Perciò non vi è differenza tra una chiamata a un DB e quella a un sistema esterno: eventuali modifiche future al livello di persistenza non si ripercuoterebbero nei server soprastanti (per esempio se in futuro si integrassero eventuali funzionalità ora demandate ai sistemi esterni).

Pattern & Design Principle

Per ottenere disaccoppiamento tra client e server si adotta il pattern Broker. Tra le altre cose il Broker permette di redirigere le richieste dei client e filtrare le richieste di client non autorizzati. I server espongono i propri servizi con cui solamente il broker può interfacciarsi. Inoltre, tale scelta applica il principio di inversione delle dipendenze: il broker nasconde la struttura dei server agevolando modularità ed estendibilità.

Per i medesimi motivi si decide di inserire un secondo broker tra i livelli server e persistenza in modo tale da agevolare la modularità e l'inversione delle dipendenze.

Un canale sicuro sarà implementato con il protocollo TLS tra client e broker e tra broker e server per garantire la protezione e l'integrità dei dati.

Si adotta il pattern architettonale MVP per il livello server che ha una view estremamente ridotta, passiva e usata solo per esporre i metodi remoti al broker.

Per il livello client si adotta il pattern MVVM, orientato alla presentazione e quasi senza business logic.

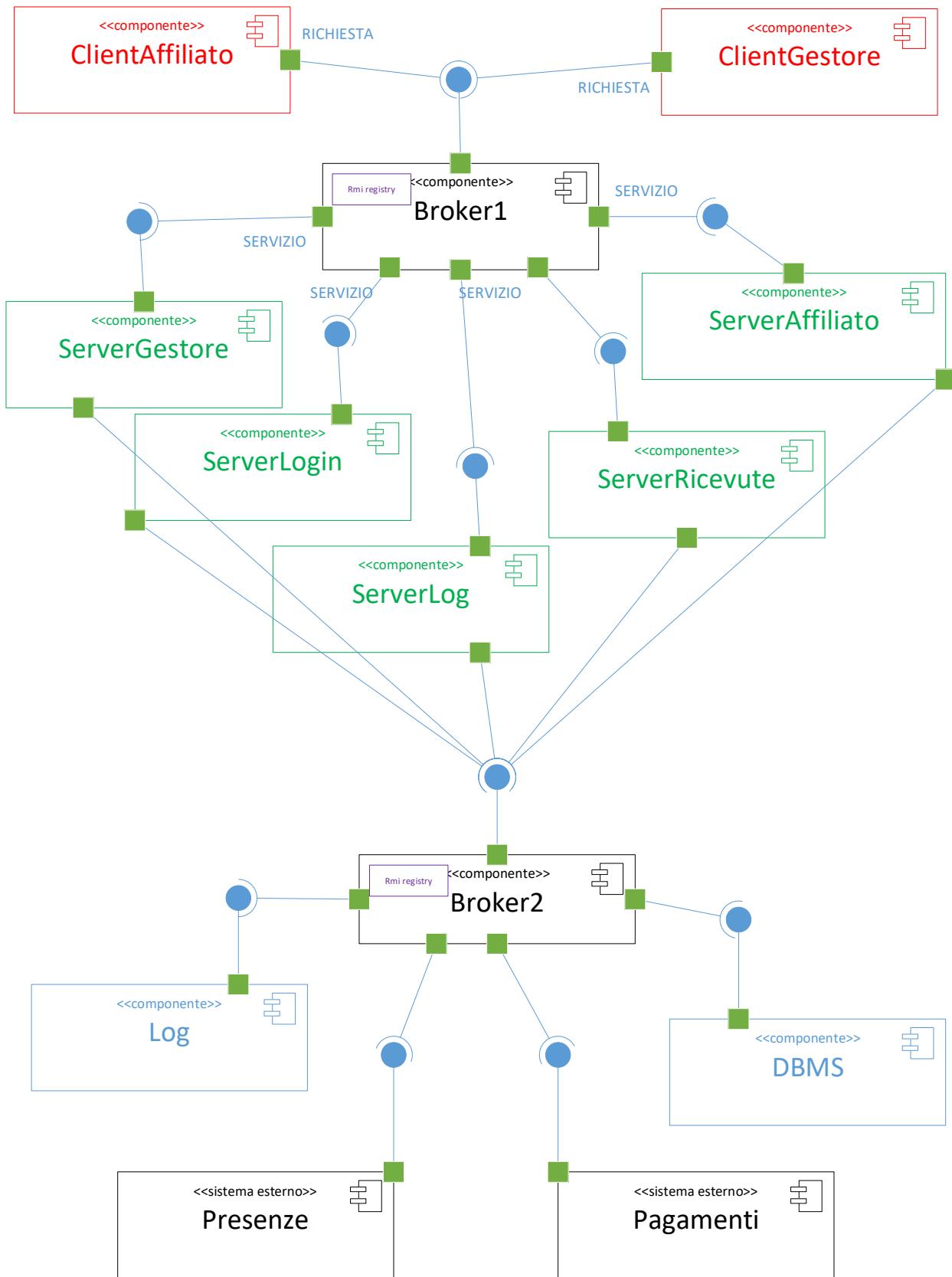
Scelte Tecnologiche

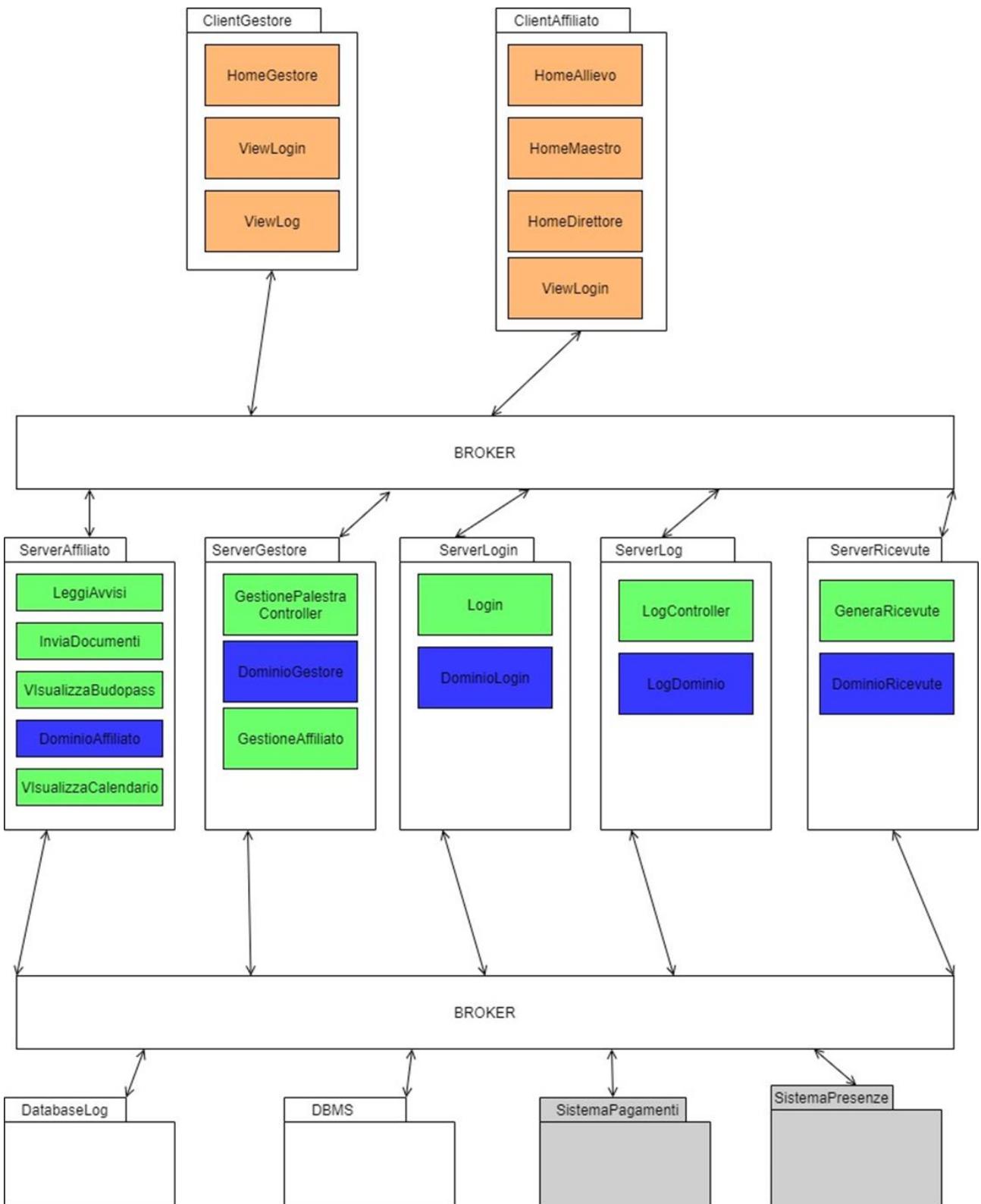
Si decide di adottare Java RMI per l'invocazione dei metodi remoti sia tra il primo broker e i server che tra i server e il secondo broker. Java RMI permette di effettuare chiamate remote in un approccio object-oriented. Inoltre, l'uniformità data dalla JVM permette di avere un modello dei dati uguale dappertutto e di non doversi occupare di operazioni di marshalling. L'uso di chiamate remote permette allo sviluppatore di doversi occupare solo del livello applicativo e non di ciò che accade sotto tale livello dello stack di rete.

Si decide di collocare l'RMIRRegistry in locale ai broker. I server registrano i propri servizi nel registry del primo broker che quest'ultimo usa per invocare i metodi. Dualmente, i server invocano i servizi registrati dal secondo broker nel suo registry.

In RMI non vi è un modo standard per poter inviare file in quanto questi non sono serializzabili. Perciò visto che nel nostro dominio i file da inviare in genere sono PDF molto piccoli si decide, come mostrato nella progettazione di dettaglio, di modellare avvisi e documenti come array di byte e di inserire un limite massimo molto basso per la dimensione dei file che si possono inviare. Gli array di byte sono serializzabili e quindi inviabili tramite RMI.

Broker2 e server comunicano con RMI e nel broker2 si utilizza il pattern DAO per l'accesso alla persistenza. Le factory del pattern DAO vengono situate nei server. Tali factory si occupano di interrogare l'RMIRRegistry del broker2 per ottenere i riferimenti remoti che restituiscono al server il quale potrà invocare gli opportuni metodi remoti.

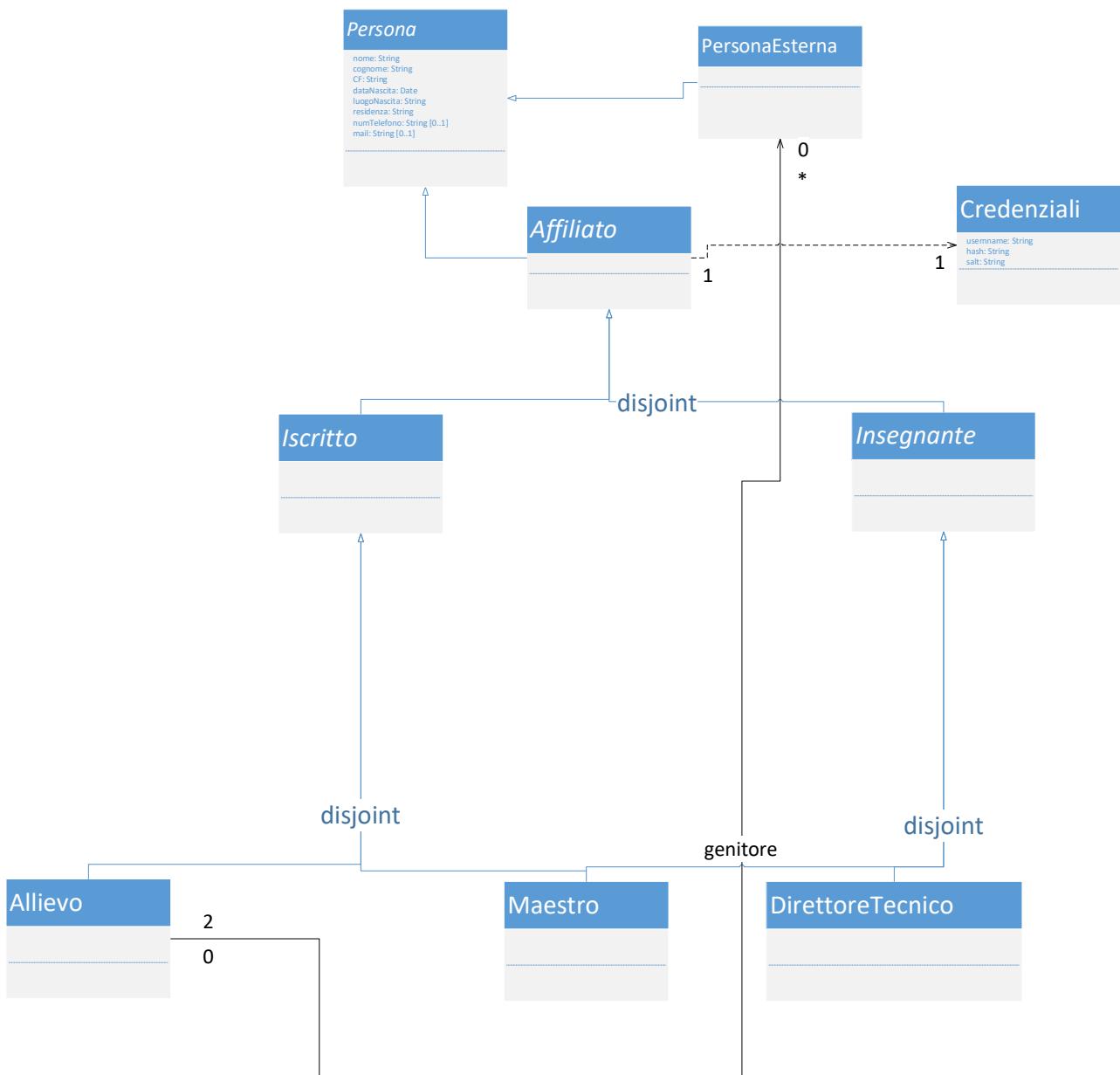
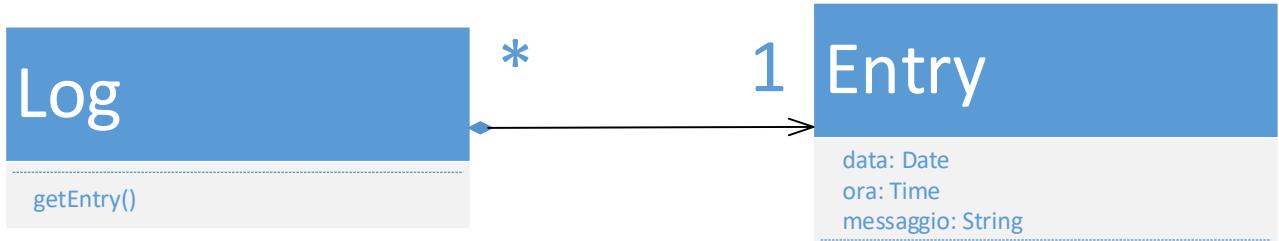




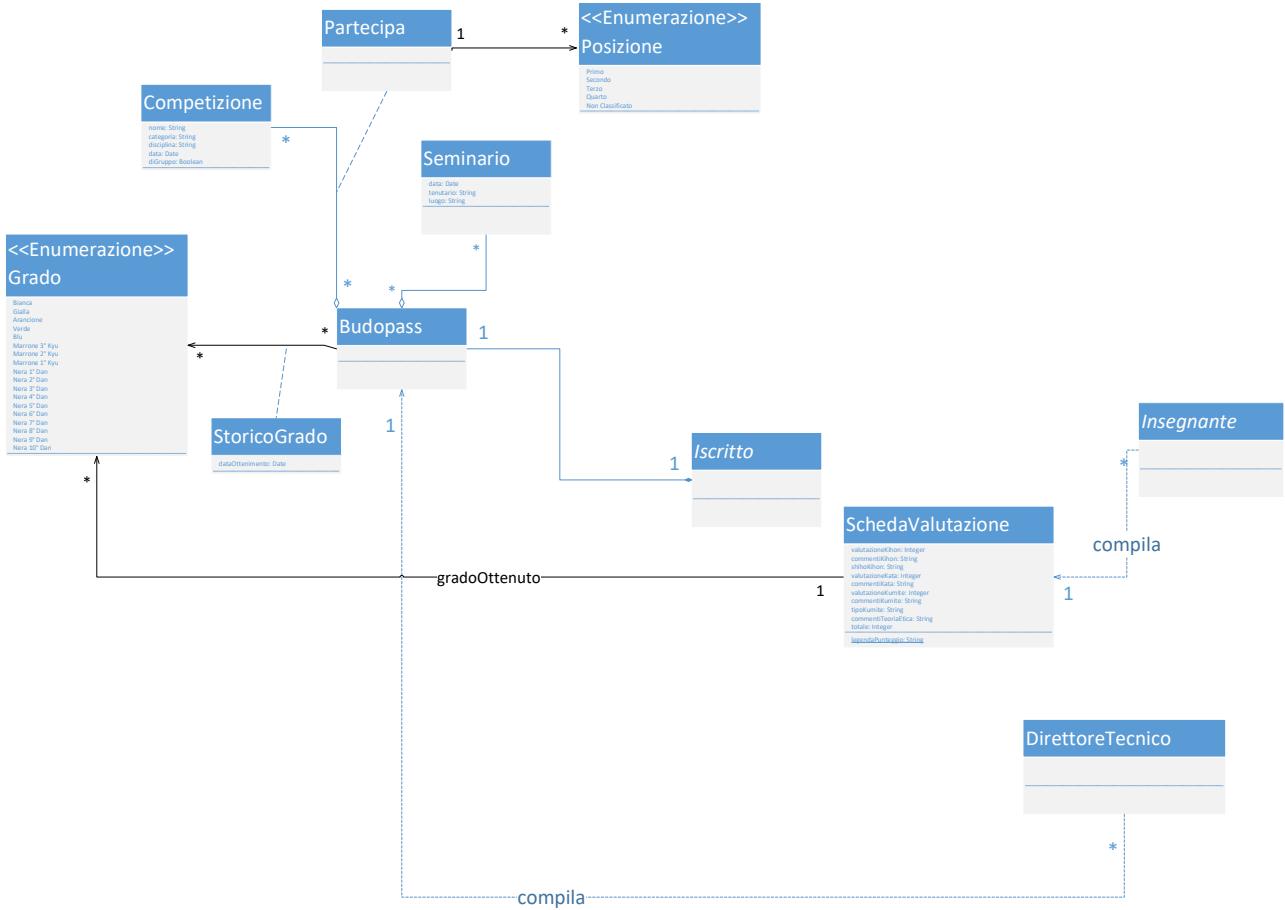
PROGETTAZIONE DETTAGLIO

Struttura

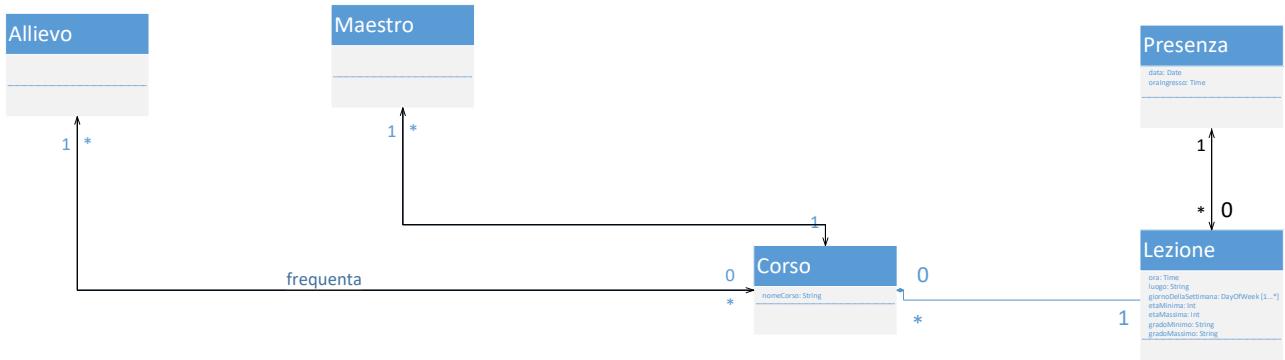
Si dettaglia ulteriormente il modello del dominio svolto in analisi.

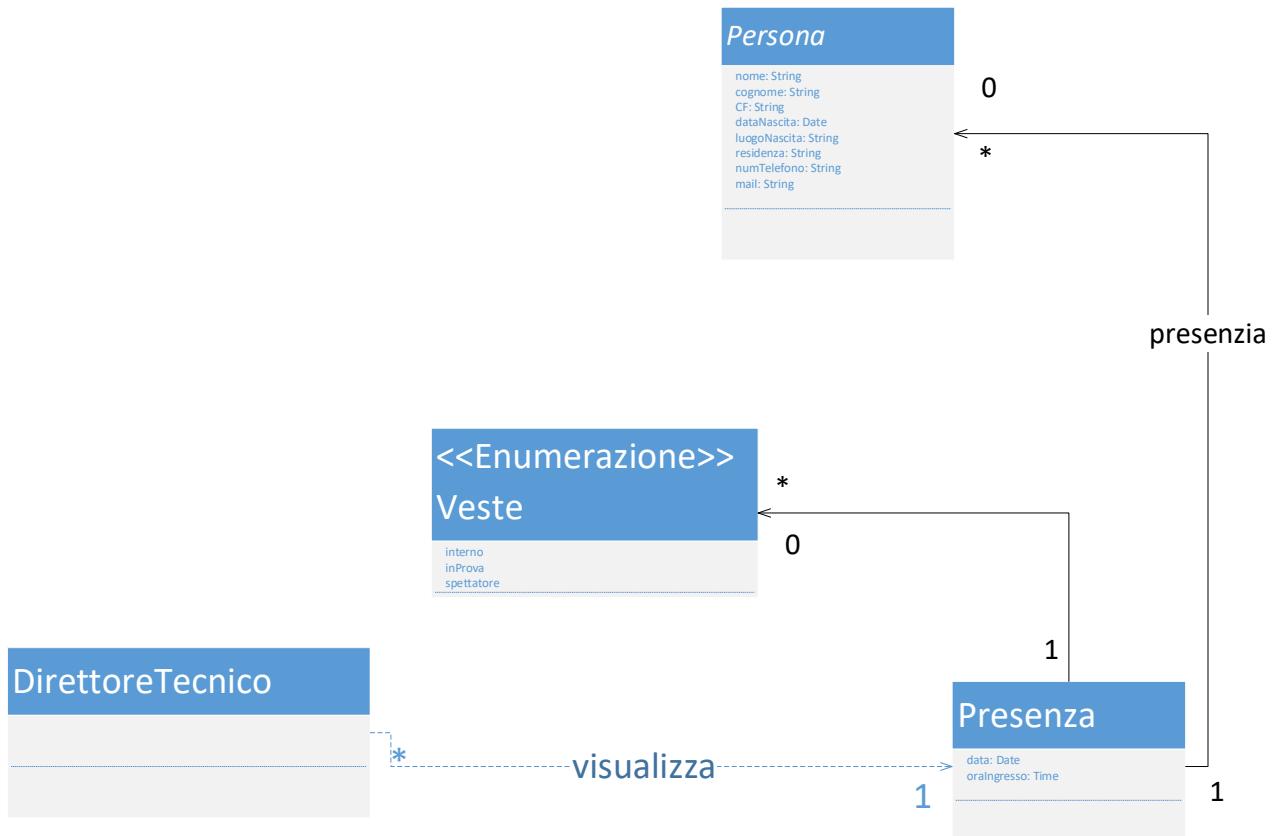
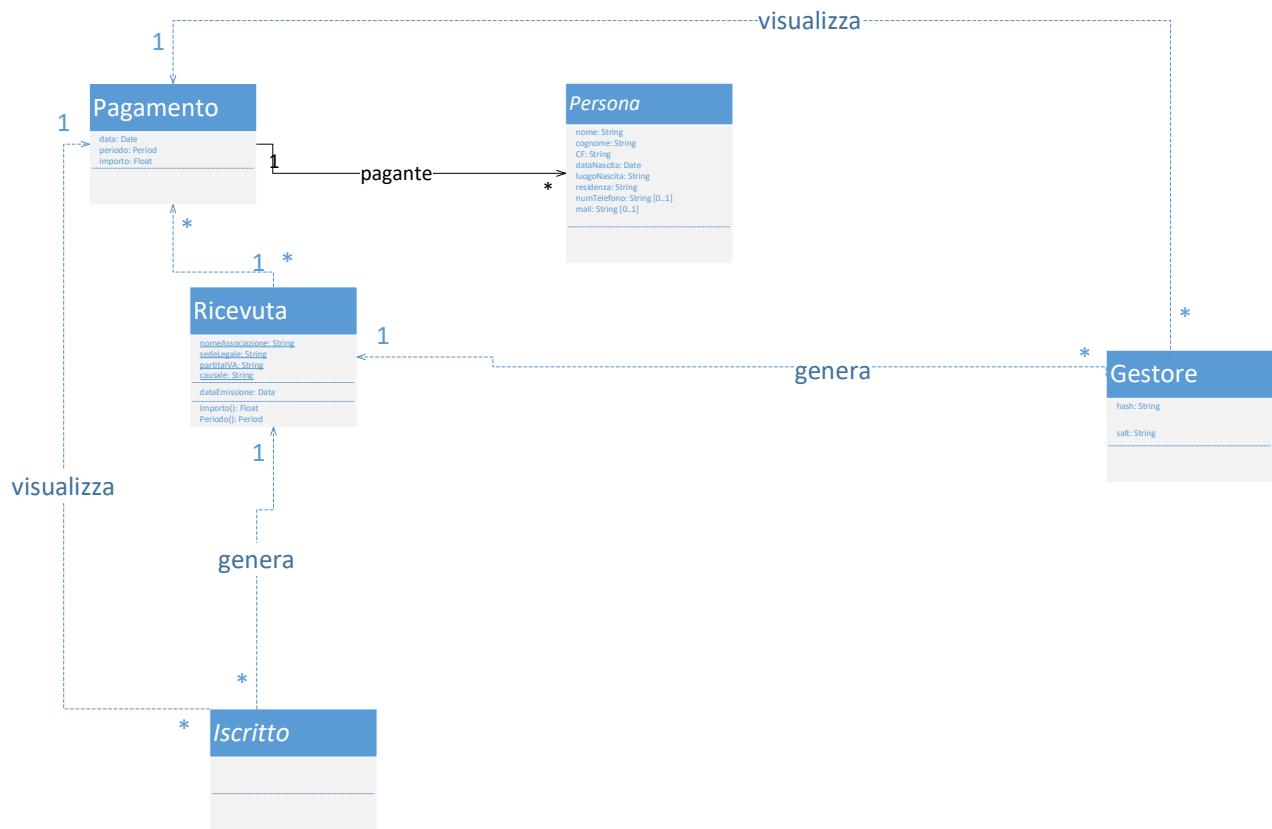


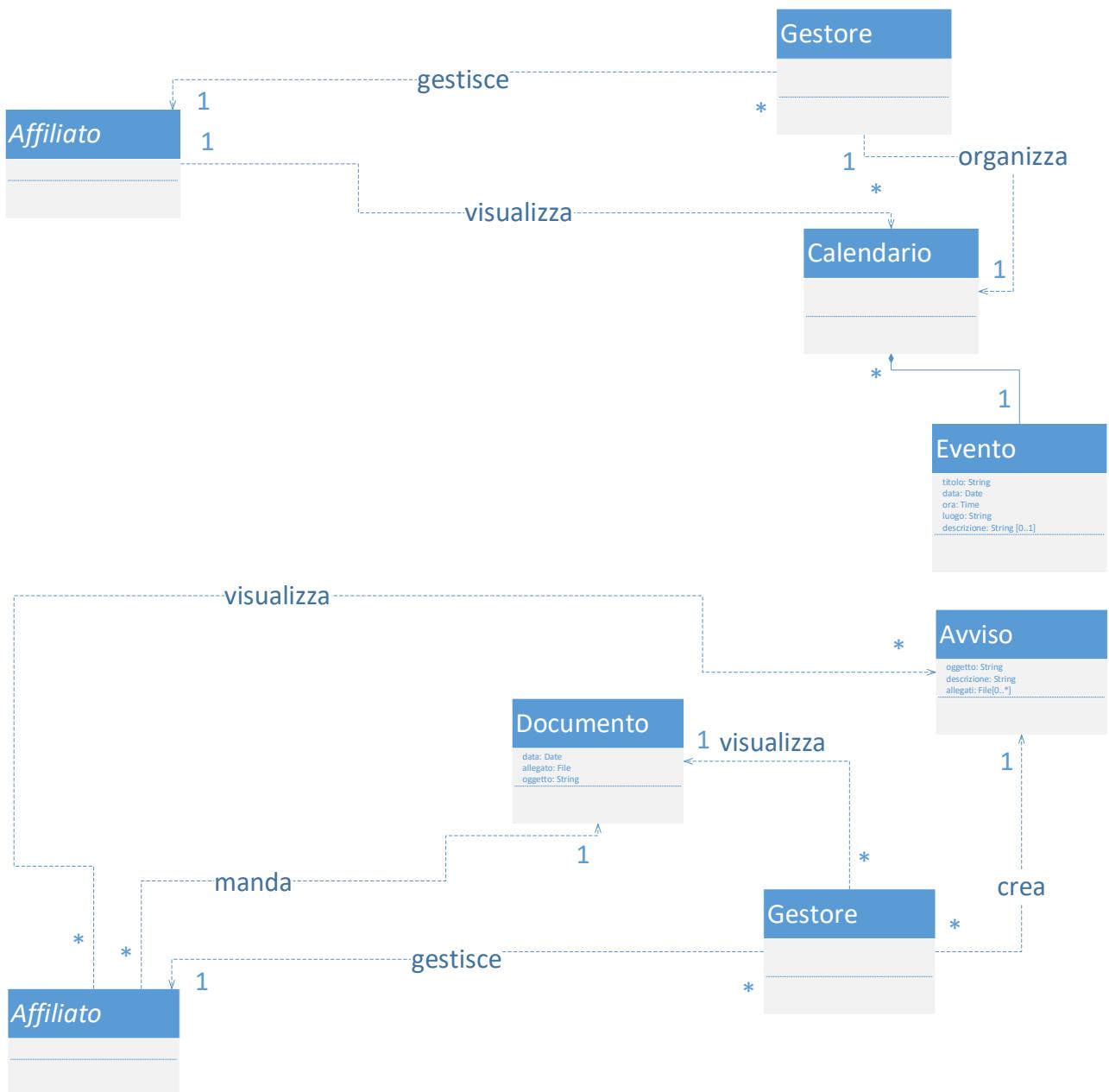
Si è aggiunta la classe credenziali. La classe concreta PersonaEsterna è stata introdotta per poter modellare i genitori degli allievi minorenni e i partecipanti esterni alle lezioni (visitatori o in prova).



Si sono aggiunte le frecce di navigabilità. Nelle nostre relazioni di composizione e aggregazione la freccia di navigabilità risulta sempre opposta al verso del diamante quindi da qui in poi viene omessa per semplicità. Inoltre, da qui in poi si evidenziano in nero le frecce di navigabilità non presenti in analisi.





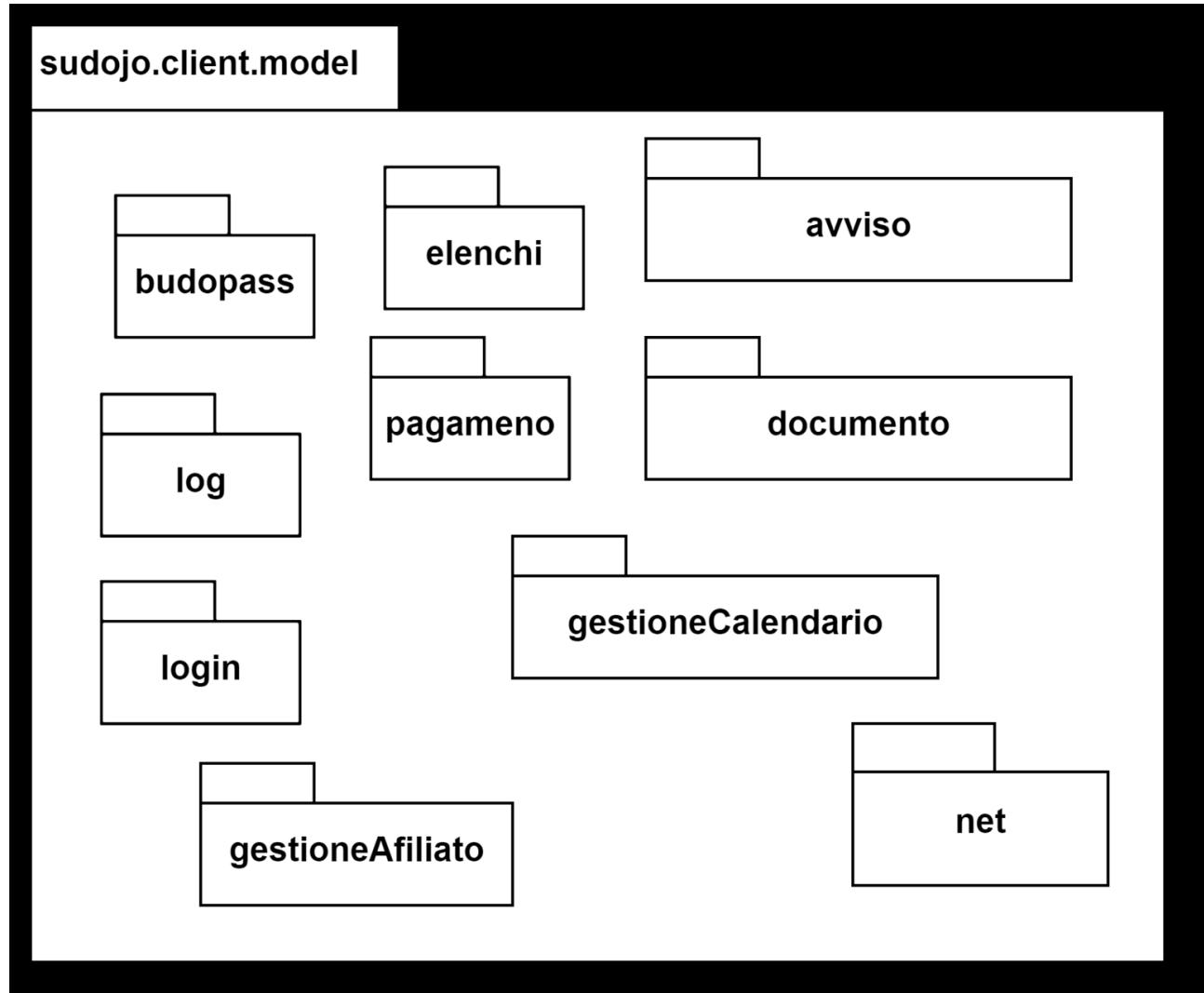


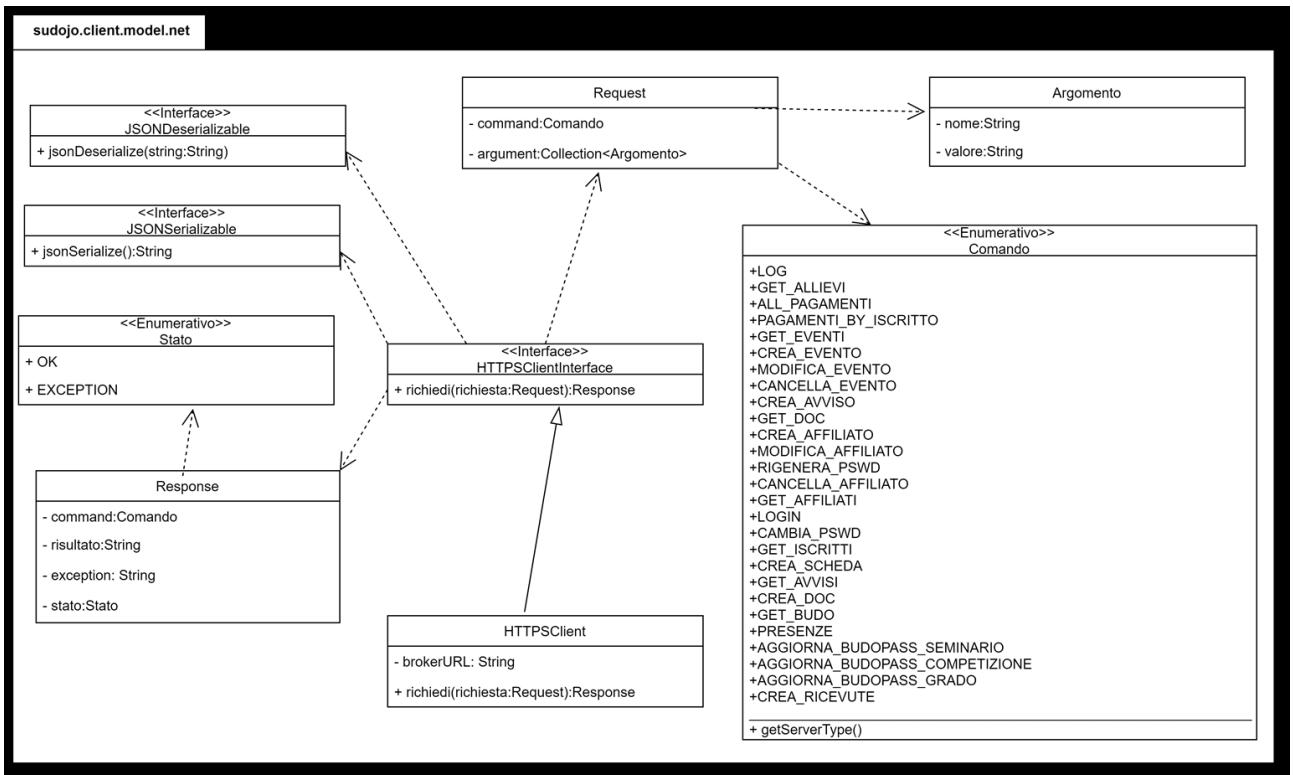
Struttura client

Come detto in precedenza, la struttura è basata su MVVM.

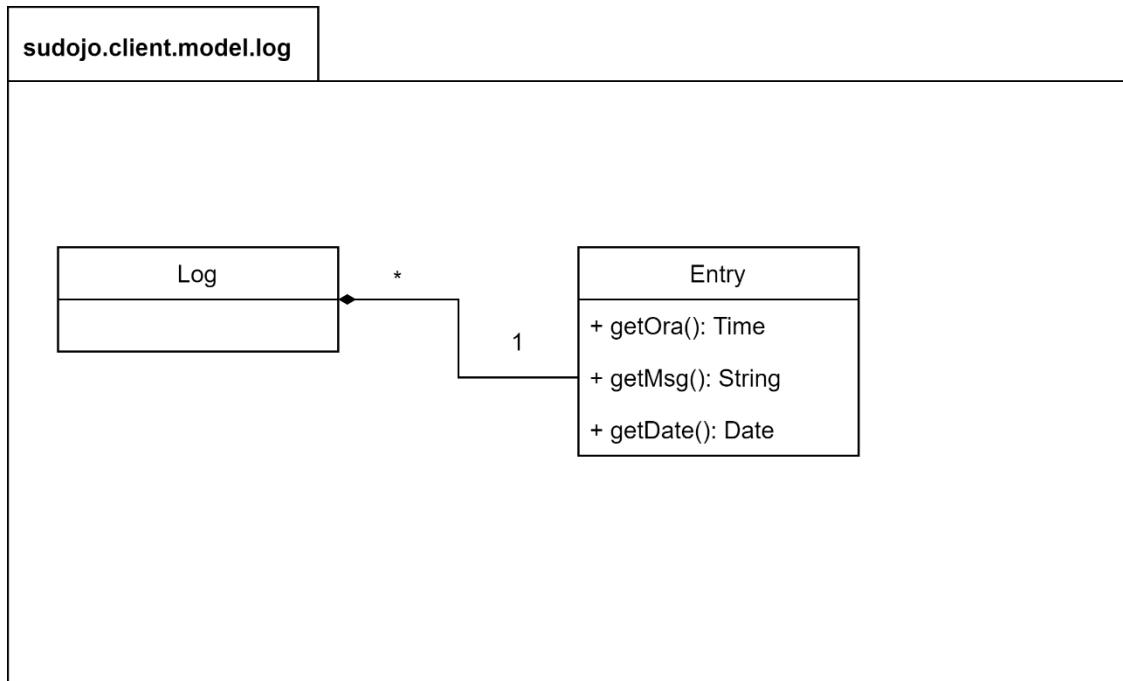
- Model: contiene il modello dei dati ed elabora le richieste HTTPS per il broker.
- View: presentazione delle maschere all'utente. Si usa il pattern Observer per il binding con il ViewModel.
- ViewModel: responsabile della logica della vista.

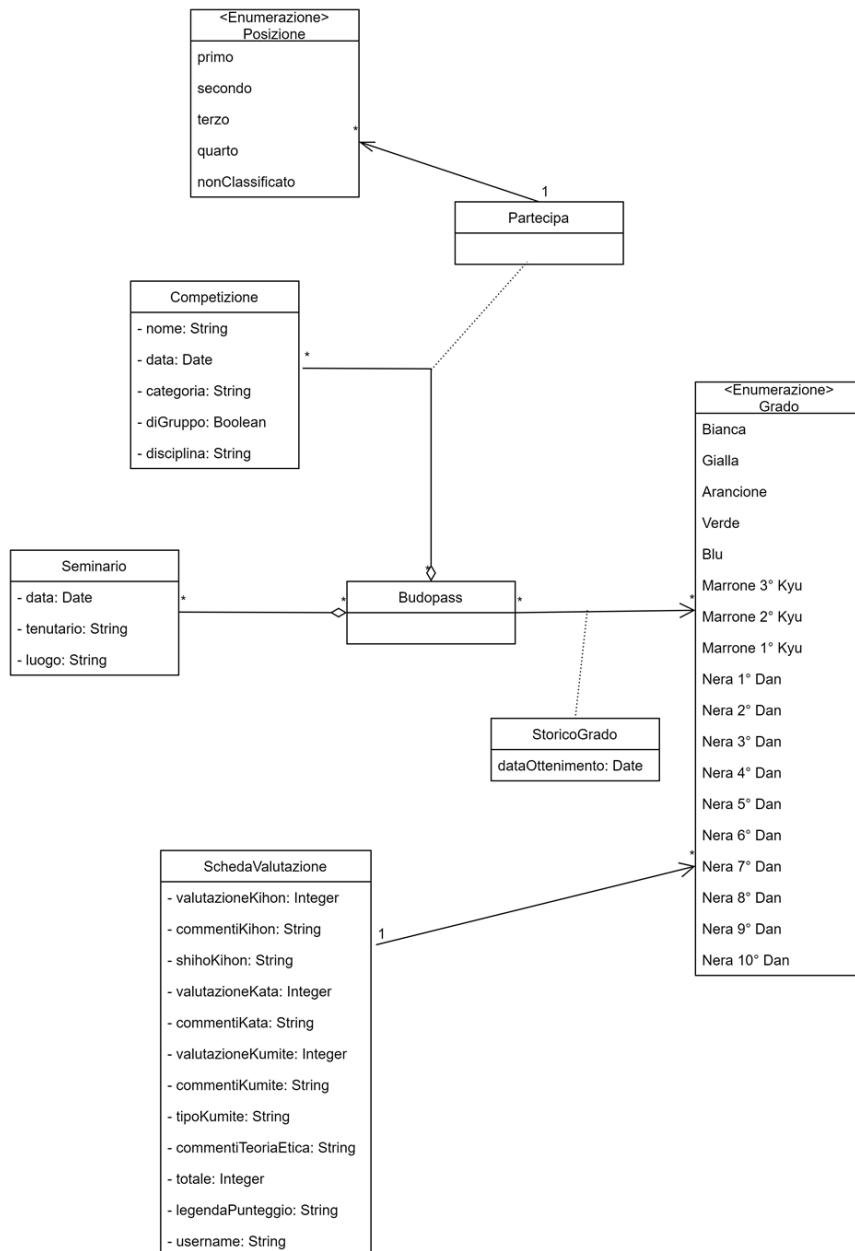
MODEL:

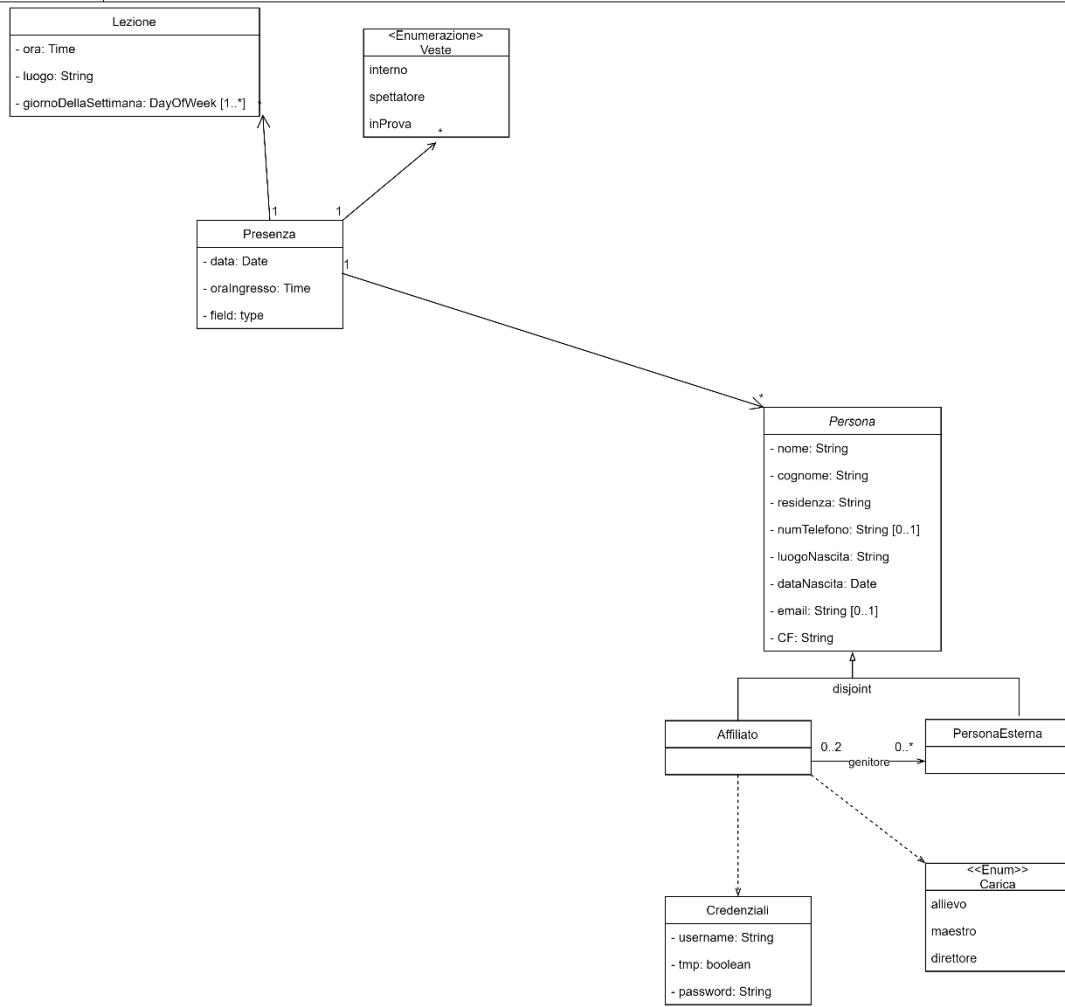




La comunicazione tra client e broker avviene tramite HTTPS (protocollo basato su TLS). Per lo scambio dei dati si utilizza il formato JSON. Il client invia le richieste specificando il comando richiesto tramite l'apposito enumerativo. Le richieste al broker vengono effettuate dalla classe concreta `HTTPSClient` che implementa l'interfaccia `HTTPSClientInterface`.





sudojo.client.model.gestioneAffiliato**sudojo.client.model.gestioneCalendario**

sudojo.client.model.documento

Documento

- oggetto: String
- data: Date
- userMittente: String
- nomeMittente: String
- cognomeMittente: String
- allegato: byte[]

Considerando che per il nostro dominio in genere i file da inviare sono PDF abbastanza piccoli e considerando quanto detto sopra su RMI si impone un limite basso di dimensione massima ai file in modo tale da poterli modellare come array di byte serializzabili.

sudojo.client.model.avviso

Avviso

- oggetto: String
- descrizione: String
- allegato: byte[] [0..*]

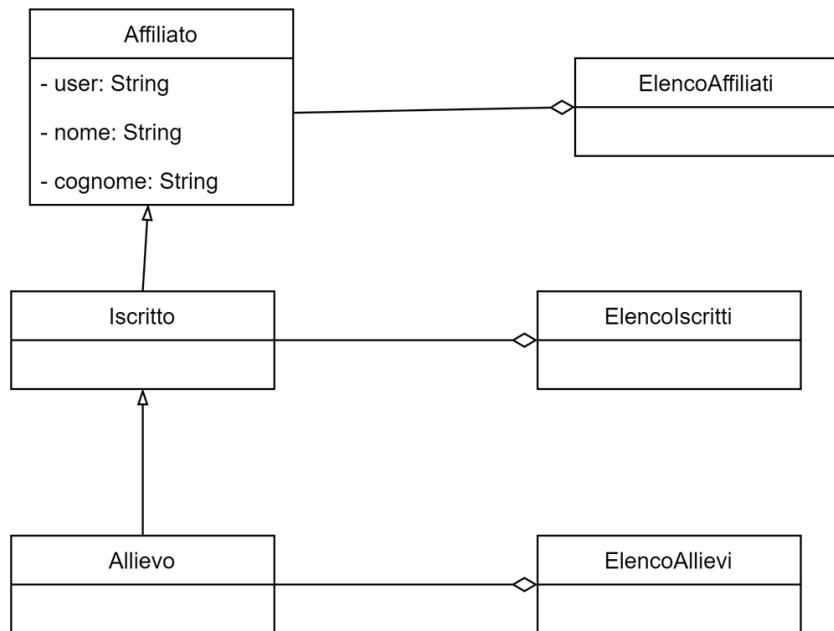
sudojo.client.model.login

| <<Enum>> |
|-----------------------------|
| StatoLogin |
| ACCETTATO |
| RIFIUTATO |
| TMP |

Quando viene aggiunto un nuovo affiliato vengono generate delle credenziali con password temporanea. Al primo login verrà chiesto all'utente di inserire una nuova password. Inoltre, qualora un affiliato debba avere problemi con le credenziali o desiderasse cambiarle dovrà per forza rivolgersi di persona al gestore il quale, dopo le opportune verifiche, rigenererà la nuova password temporanea per l'utente.

L'enum StatoLogin è ciò che viene ritornato dal broker dopo una richiesta di login. L'esito del login può essere "accettato" qualora username e password siano corrette e la password non è temporanea, "rifiutato" qualora le credenziali siano errate e "tmp" qualora siano corrette e la password sia temporanea. In questo caso il sistema chiederà all'utente di inserire la password nuova.

sudojo.client.model.elenchi



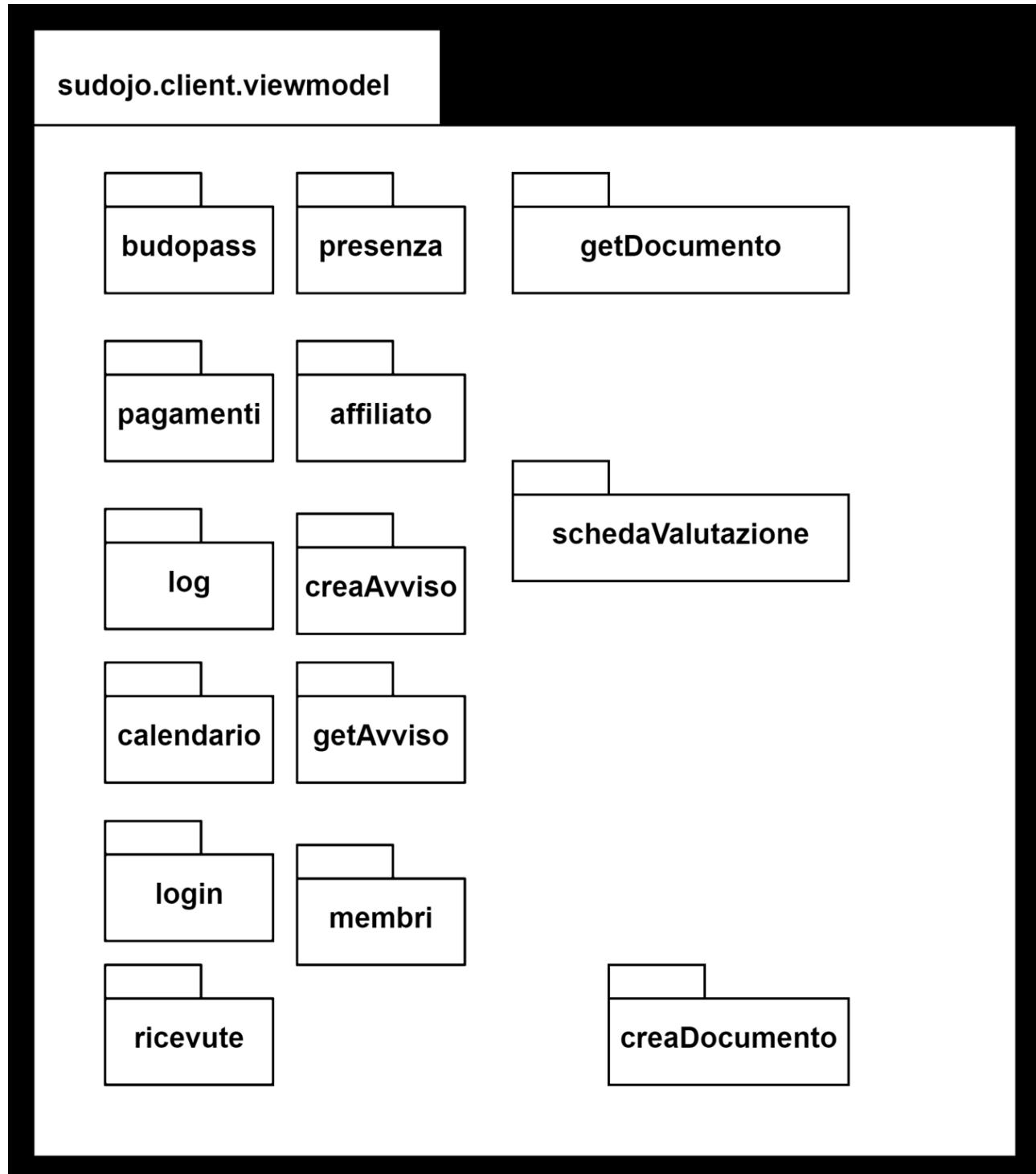
Come si può notare negli scenari e nei casi d'uso vi sono molte operazioni che richiedo l'elenco di affiliati o iscritti o allievi. Per tutte queste operazioni servono solamente le informazioni di username, nome e cognome e non tutte le informazioni presenti nella classe Persona. Perciò si introducono queste nuove classi in un opportuno package a parte il cui scopo è essere usate negli elenchi.

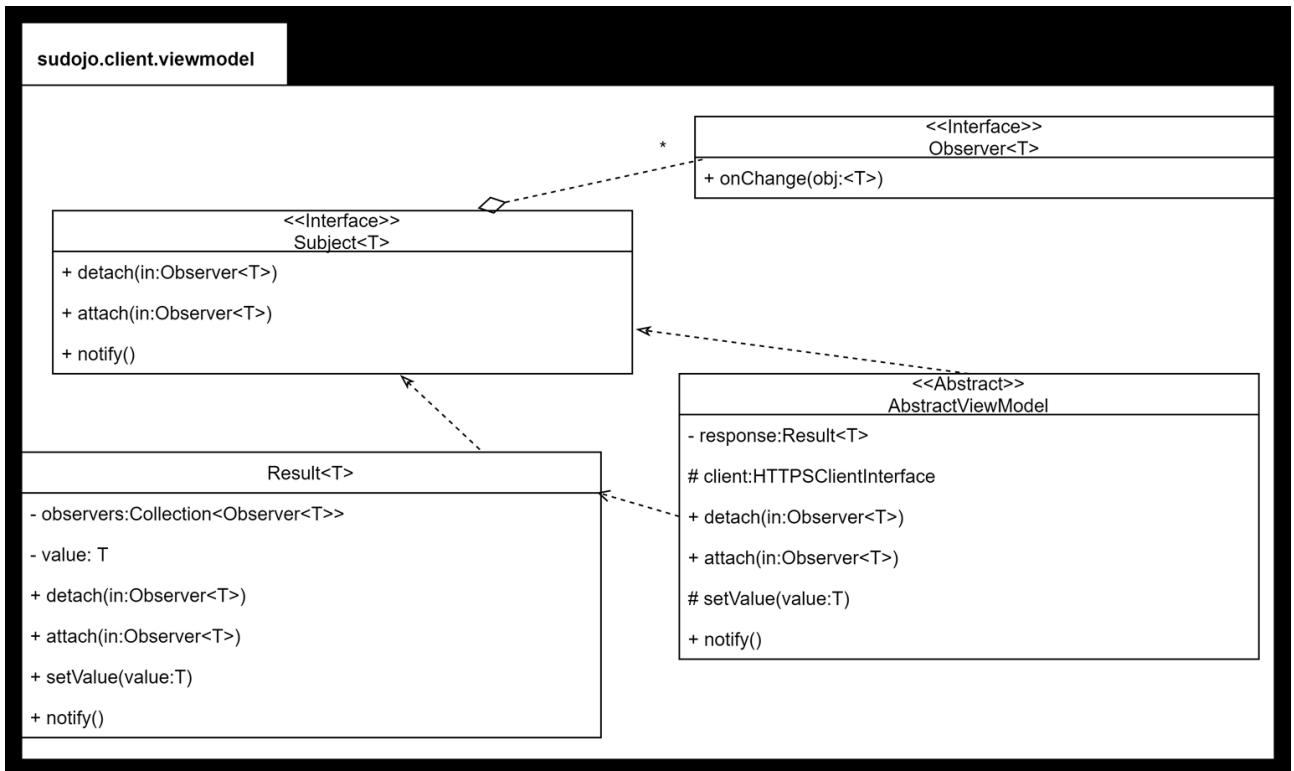
sudojo.client.model.pagamento

| Pagamento | Ricevuta |
|--|--|
| - data: Date - periodo: Period - importo: float - CFPagante: String - cognomePagante: String - userIscritto: String - nomeIscritto: String - cognomeIscritto: String - nomePagante: String | - nomePalestra: String - sedeLegale: String - P.A.: String - importo: float - cognomePagante: String - periodoRiferimento: Period - cognomeIscritto: String - dataNascitaIscritto: Date - luogoNascitaIscritto: String - dataEmissioneRicevuta: Date - residenzaIscritto: String - causale: String - nomeIscritto: String - nomePagante: String |

In questo package si modellano i pagamenti e le ricevute. I pagamenti vengono ottenuti dal sistema esterno come oggetti ben formati, auto-contenuti e aventi tutti i dati necessari. Le ricevute vengono generate dall'apposito server contenente la logica di business necessaria. Queste sono le classi che modellano tali entità.

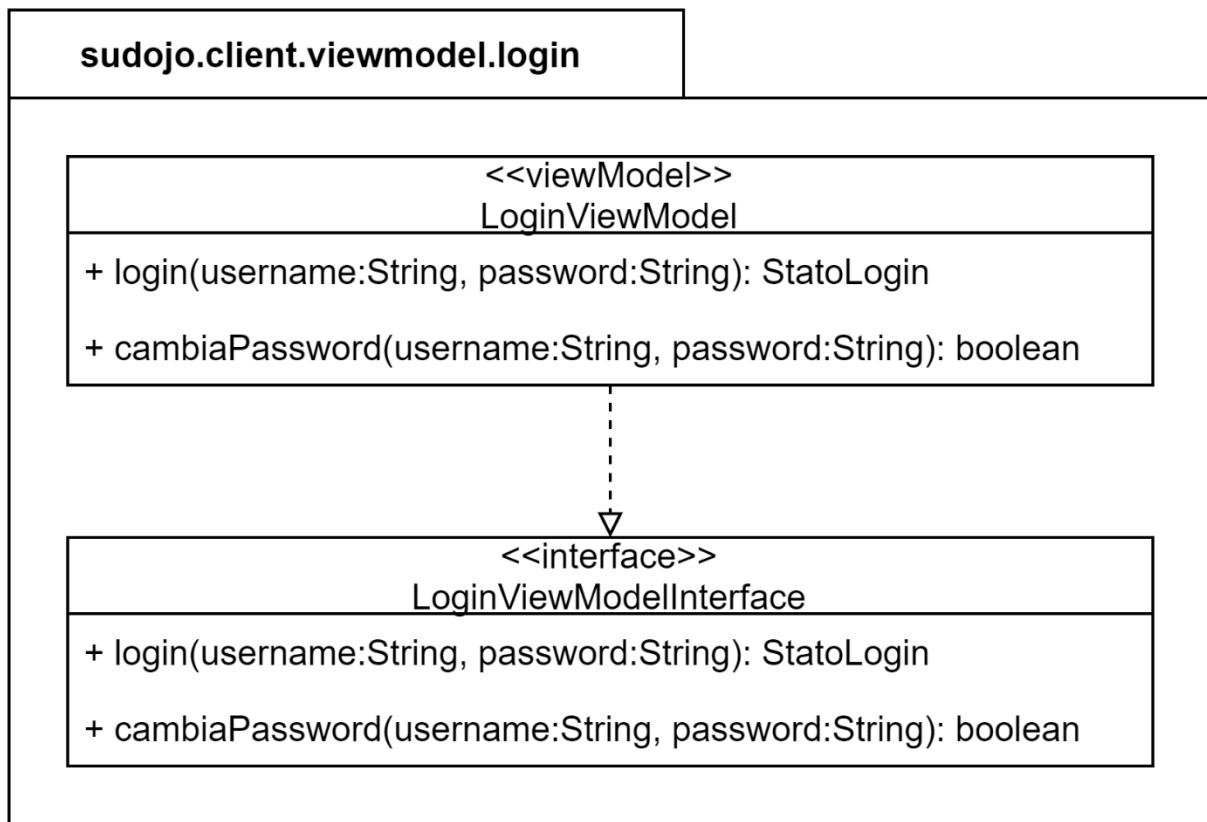
VIEWMODEL:





Per il binding tra view e.viewmodel si usa il pattern observer. La view all'occorrenza invoca gli opportuni metodi del.viewmodel per poter eseguire le funzionalità. I dati della risposta HTTPS arrivano al.viewmodel che modifica il proprio stato notificando tramite il pattern observer la view che aggiorna l'output di conseguenza.

Le classi del.viewmodel che estendono AbstractViewModel sono denotate dallo stereotipo <<viewModel>>.



sudojo.client.viewmodel.pagamenti

<<viewModel>>
GetPagamentiViewModel

- + getAllPagamenti(): List<Pagamento>
- + getPagamentiByIscritto(usernameIscritto:String): List<Pagamento>

<<interface>>
GetPagamentiViewModellInterface

- + getAllPagamenti(): List<Pagamento>
- + getPagamentiByIscritto(usernameIscritto:String): List<Pagamento>

sudojo.client.viewmodel.schedaValutazione

<<viewModel>>
CreaSchedaValutazioneViewModel

- + creaSchedaValutazione(scheda:SchedaValutazione): boolean

<<interface>>
CreaSchedaValutazioneViewModellInterface

- + creaSchedaValutazione(scheda:SchedaValutazione): boolean

sudojo.client.viewmodel.log

<<viewModel>>
GetLogViewModel

+ getLog(): List<Log>



<<interface>>
GetLogViewModellInterface

+ getLog(): List<Log>

sudojo.client.viewmodel.presenza

<<viewModel>>
GetPresenzeViewModel

+ getPresenze(): List<Presenza>



<<interface>>
GetPresenzeViewModellInterface

+ getPresenze(): List<Presenza>

sudojo.client.viewmodel.getDocumento

<<viewModel>>
GetDocumentiViewModel

+ getDocumenti(): List<Documeto>



<<interface>>
GetDocumentiViewModellInterface

+ getDocumenti(): List<Documeto>

sudojo.client.viewmodel.creaDocumento

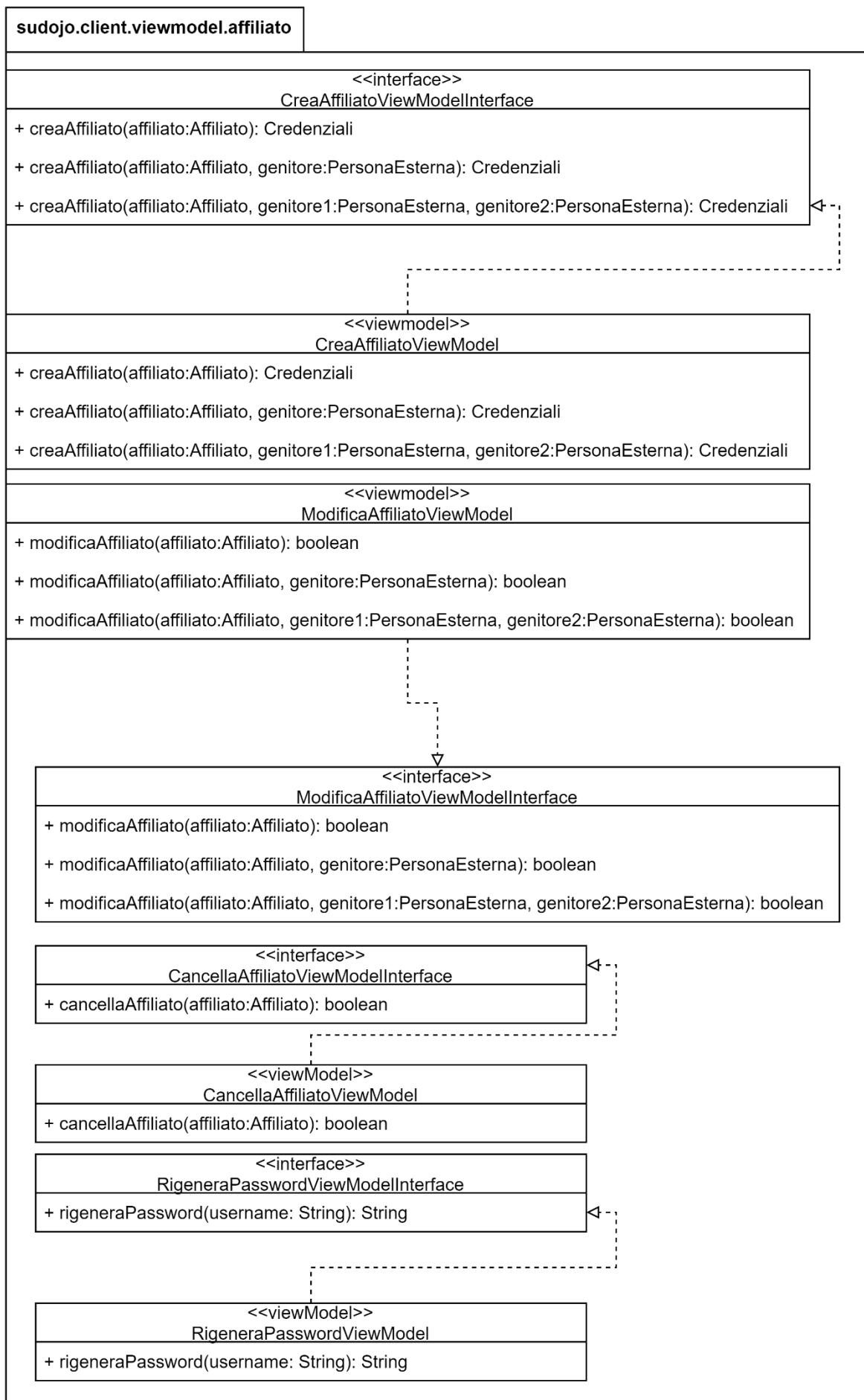
<<interface>>
CreaDocumentoViewModellInterface

+ creaDocumento(doc:Documento): boolean

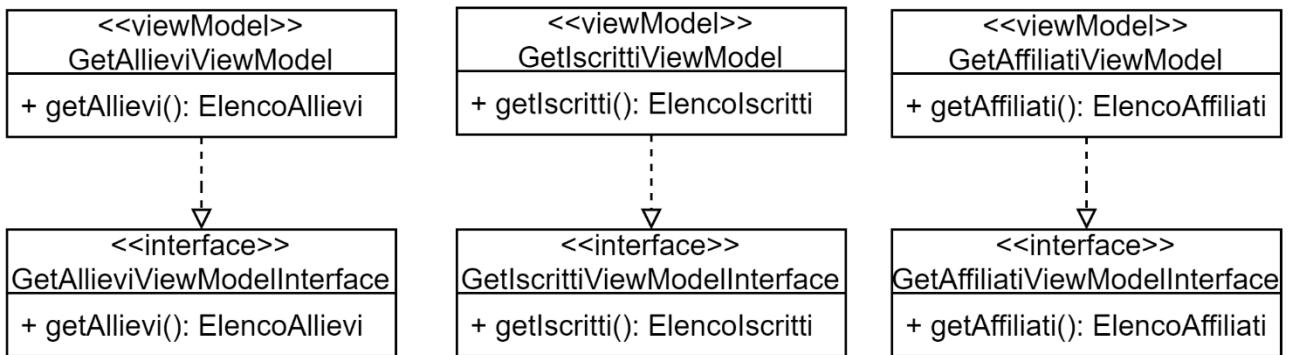


<<viewModel>>
CreaDocumentoViewModel

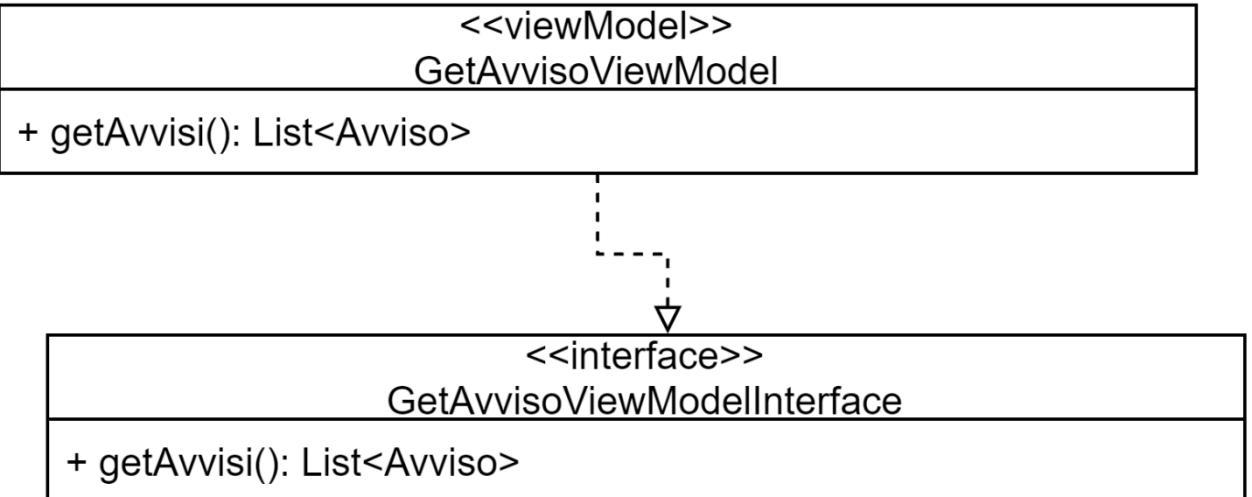
+ creaDocumento(doc:Documento): boolean



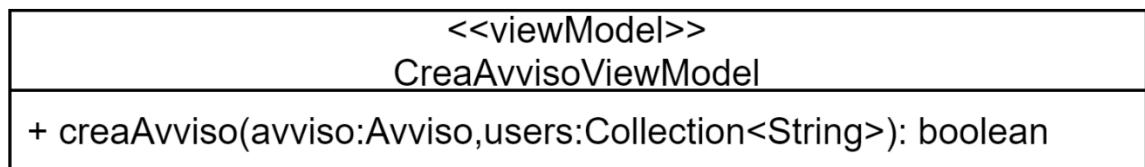
sudojo.client.viewmodel.membri



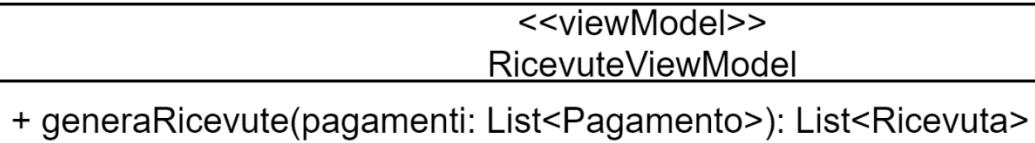
sudojo.client.viewmodel.getAvviso

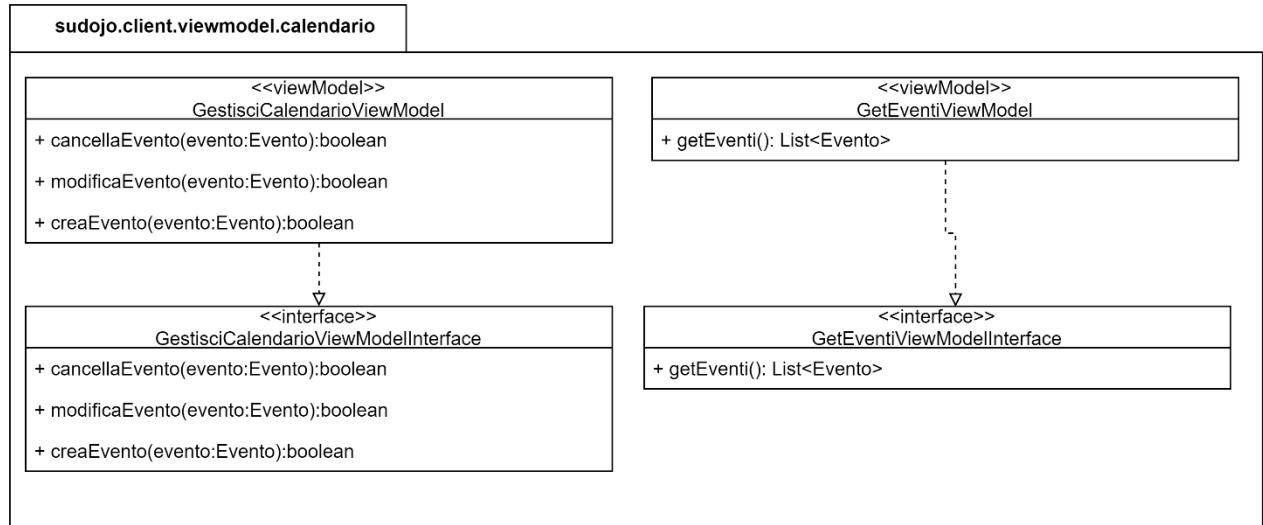
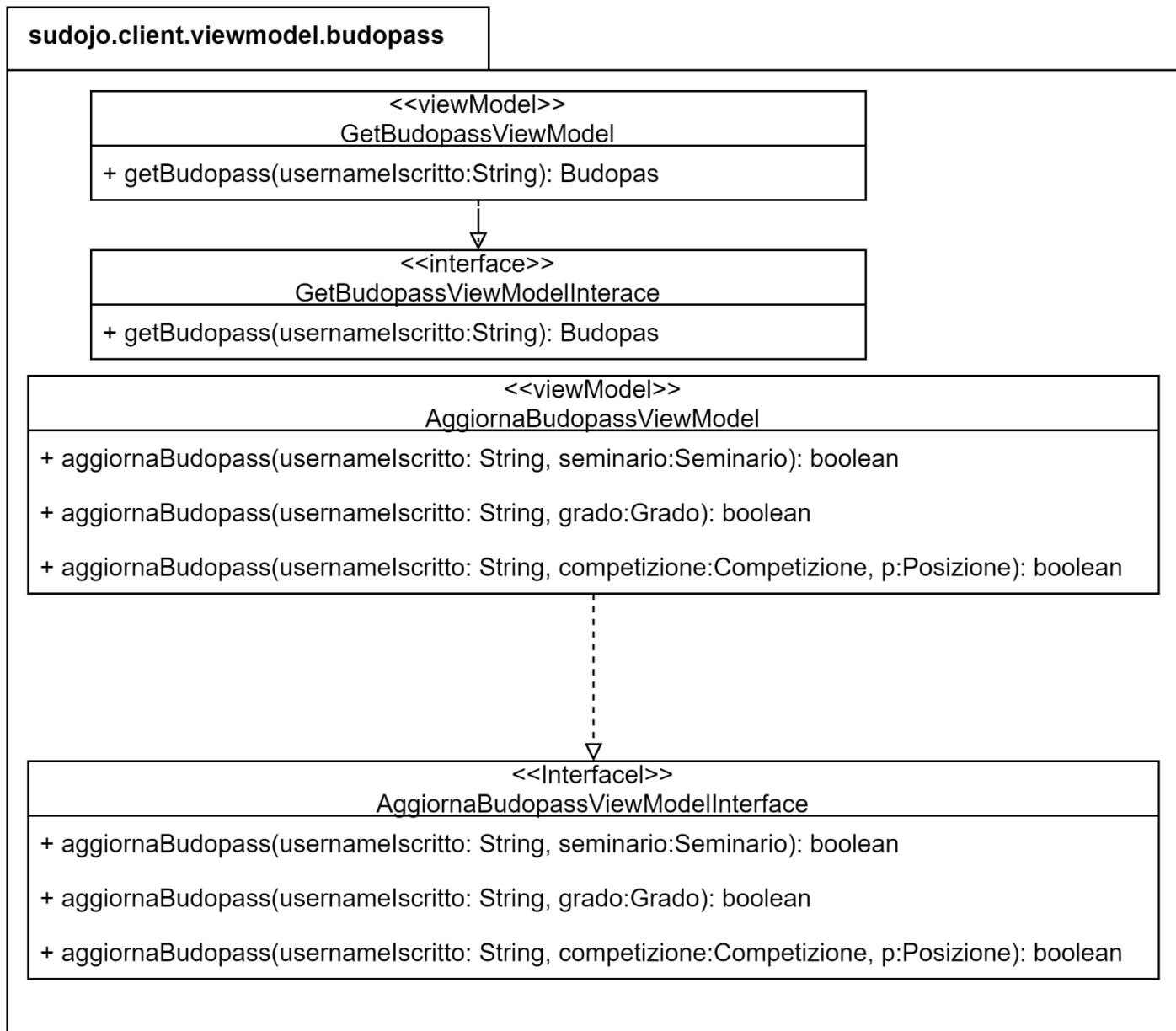


sudojo.client.viewmodel.creaAvviso

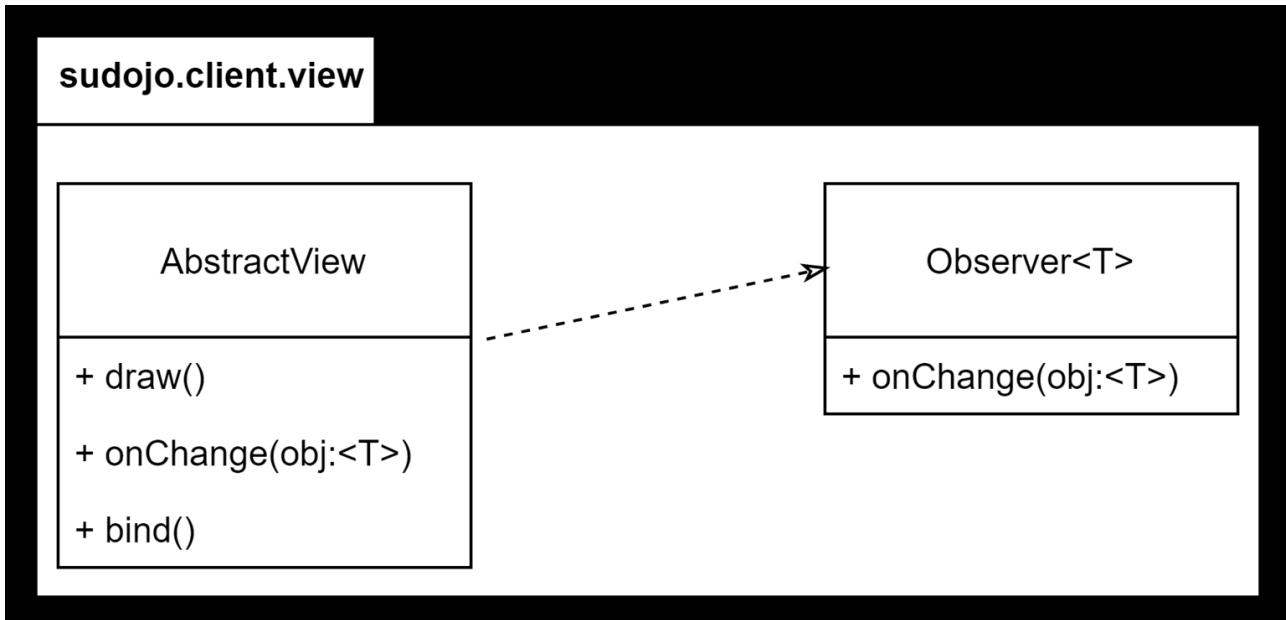


sudojo.client.viewmodel.ricevute

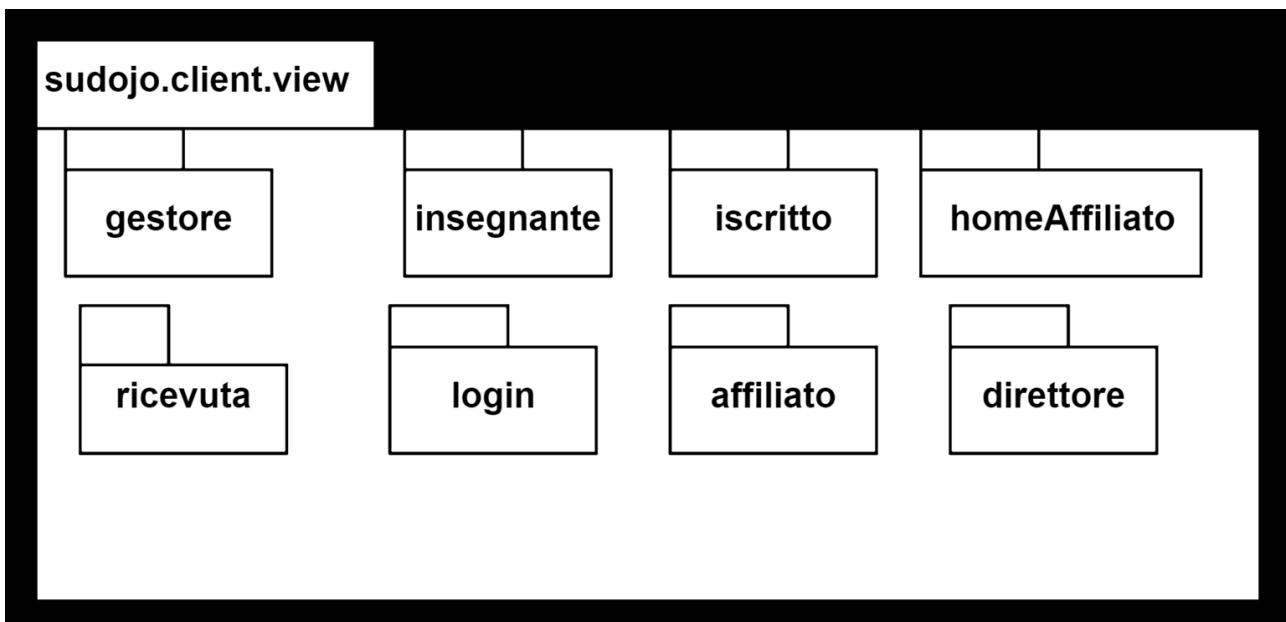




VIEW:



Lo stereotipo `<<view>>` indica che la classe è una maschera utente ed estende la classe astratta `AbstractView`. Così facendo le view sono agganciabili al.viewmodel e ne ricevono i cambiamenti.



sudojo.client.view.homeAffiliato

<<view>>
HomeMaestro

<<view>>
HomeAllievo

<<view>>
HomeDirettore

sudojo.client.view.direttore

<<view>>
MascheraVisualizzaPresenze

<<view>>
MascheraCompilaBudopass

<<view>>
CompilazioneBudopass

<<view>>
ConfermaBudopass

sudojo.client.view.affiliate

<<view>>
MascheraVisualizzaCalendario

<<view>>
MascheraLeggiAvvisi

<<view>>
MascheraInviaDocumenti

sudojo.client.view.iscritto

<<view>>
MascheraBudopass

<<view>>
MascheraVisualizzaPagamentiScritto

Elenco pagamenti

sudojo.client.view.login

<<view>>
ViewLogin

Username...

Password...

Login

sudojo.client.view.ricevuta

<<view>>
ElencoPagamentiPerRicevuta

sudojo.client.view.insegnante

<<view>>
MascheraCompilaSchedaValutazione

<<view>>
CompilazioneScheda

sudojo.client.view.gestore

<<view>>
HomeGestore

<<view>>
ViewLog

<<view>>
SceltaScrittoGeneraRicevuteGestore

<<view>>
MascheraVisualizzaPagamentiGestore

<<view>>
MascheraGestioneCalendario

<<view>>
AggiuntaEvento

<<view>>
CreazioneAvviso

<<view>>
MascheraVisualizzaDocumenti

<<view>>
ModificaEvento

<<view>>
CancellazioneEvento

<<view>>
CancellazioneAffiliato

<<view>>
MascheraNotificaAvvisi

<<view>>
MascheraGestioneAffiliato

<<view>>
AggiuntaAffiliato

<<view>>
AggiuntaGenitori

<<view>>
ModificaAffiliato

Creazione Affiliato

Nome Cognome

Inserisci Nome...

Inserisci Cognome...

Codice Fiscale

Inserisci Codice Fiscale...

Residenza

Inserisci Residenza...

Cellulare

Inserisci Cellulare...

Data di Nascita

6/09/2021

Luogo di Nascita

Inserisci Luogo di Nascita

Email

Inserisci Email...

Carica Affiliato

Scegli la Carica dell'Affiliato ▾

Crea Affiliato

Creazione Evento

Nome dell'Evento

Inserisci Nome dell'Evento...

Luogo dell'Evento

Inserisci Luogo dell'Evento...

Data e Ora dell'Evento

6/09/2021

12:00

Descrizione dell'Evento

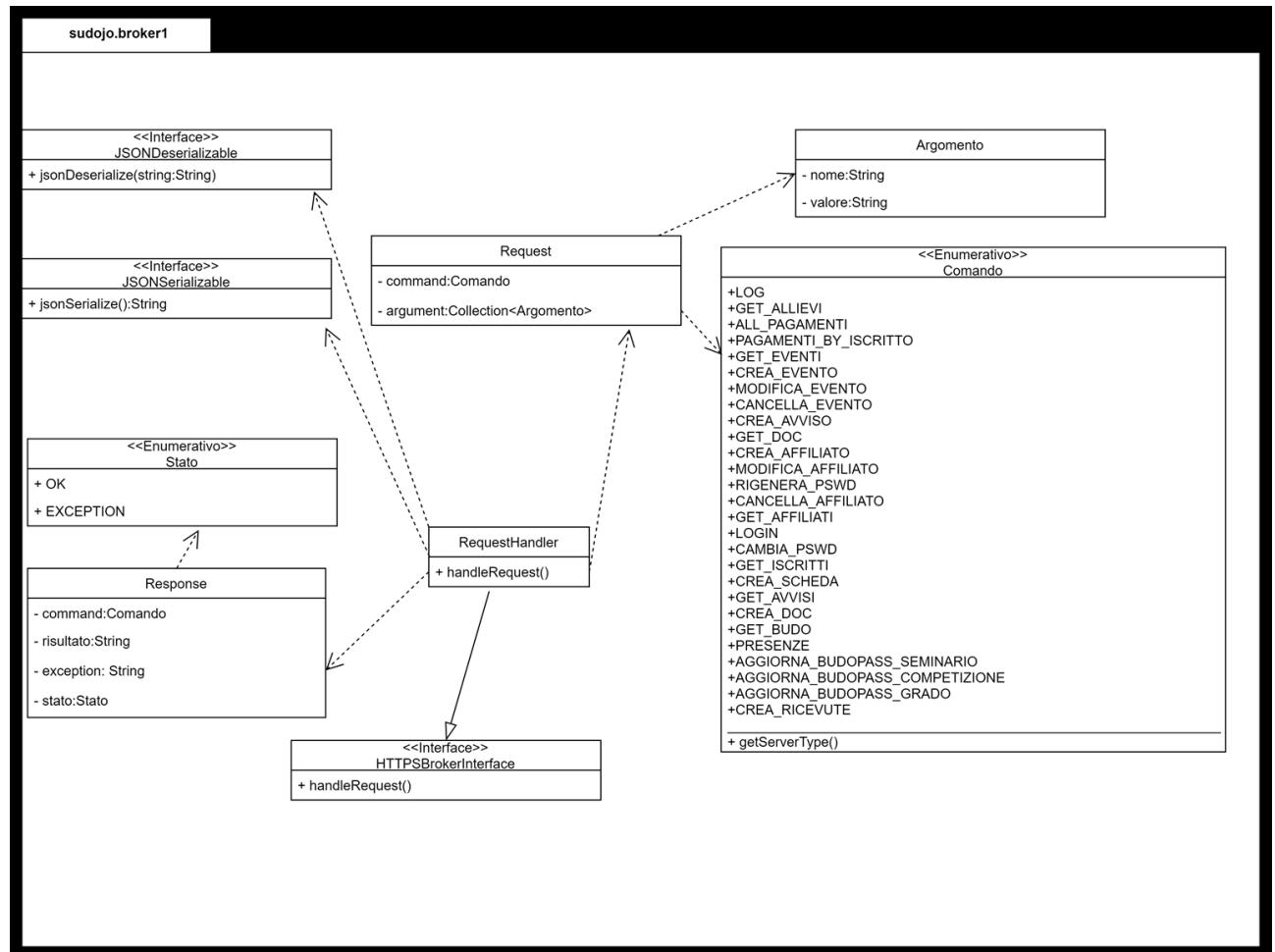
Inserisci Descrizione dell'Evento...

Crea Evento

Struttura Broker 1

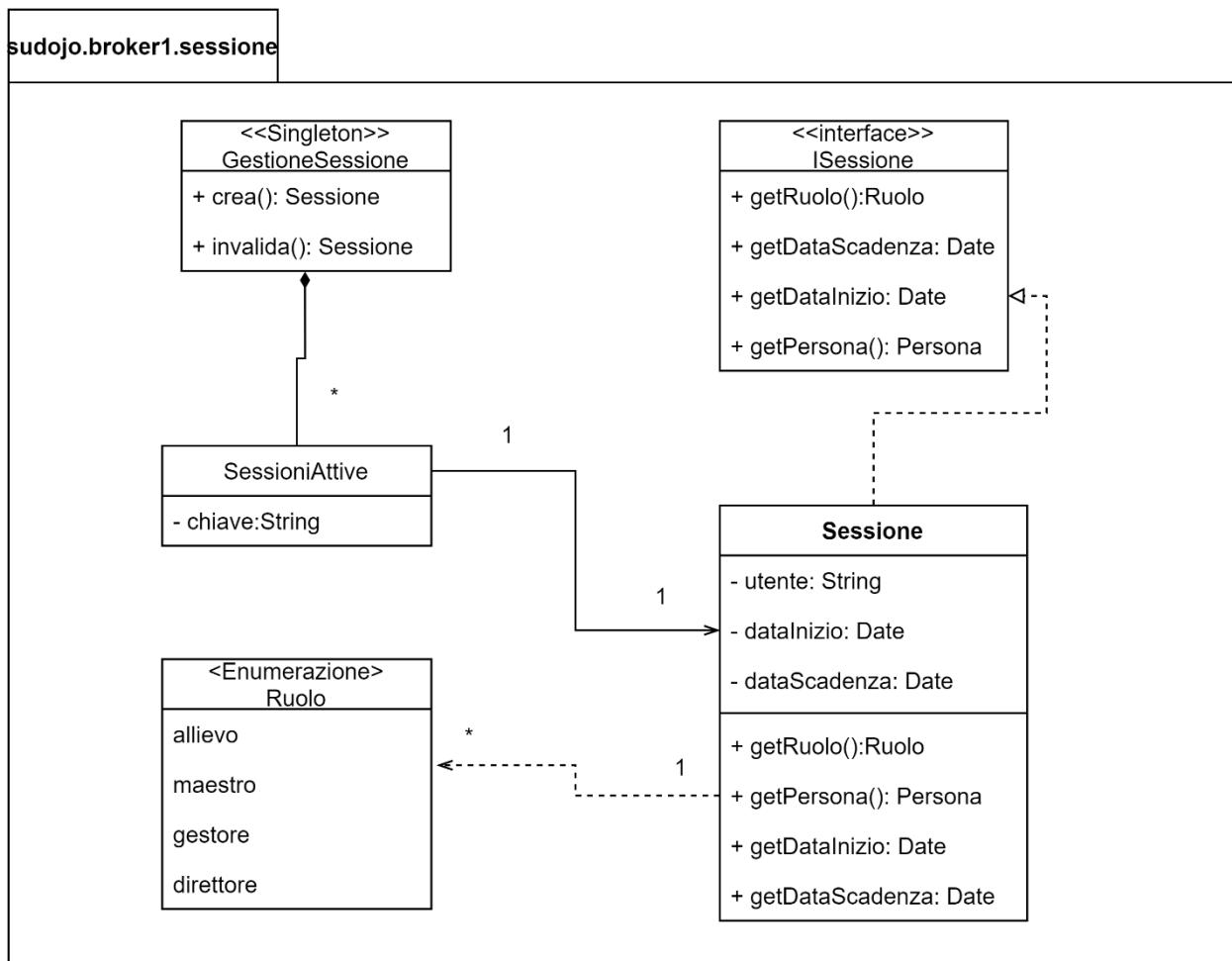
Il broker1 fa da intermediario tra i livelli client e server. Dal punto di vista dei client esso è un server HTTP. Esso riceve le richieste, invoca gli opportuni metodi remoti e manda la risposta. Comunica con i server tramite Java RMI e tramite i client con HTTPS. In entrambi i versi di comunicazione si utilizza TLS.

Ogni richiesta contiene un comando con l'operazione da svolgere e un elenco di argomenti formati da coppie nome-valore che rappresentano i parametri dell'operazione.



Ad occuparsi di ricevere le richieste HTTPS, invocare i metodi remoti ed elaborare la risposta è la classe `RequestHandler`.

Dopo il login è necessario mantenere traccia delle sessioni degli utenti per poter controllare che chi richiede una funzionalità abbia i permessi. Di questo se ne occupa la classe Sessione con l'ausilio dell'enumerativo Ruolo. Dopo il login viene creata la sessione dell'utente mantenendo username e ruolo. La gestione della sessione è affidata a un *singoletto*, rappresentato dalla classe GestioneSessione.



Il broker1 inoltre contiene le interfacce RMI che gli servono per poter richiedere i servizi remoti. Tali interfacce sono riportate nella view del livello server. Il broker1 ha in locale l'RMIRegistry al quale chiede tali servizi.

Struttura Server

La struttura di ognuno dei server è basata su MVP:

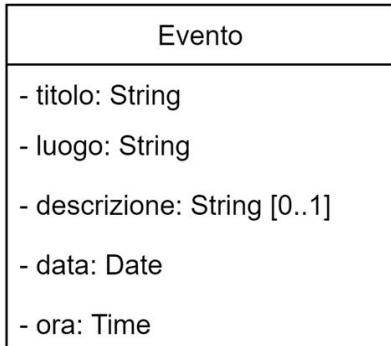
- Model: contiene il modello dei dati
- View: contiene le interfacce RMI
- Presenter: contiene la logica di business

MODEL:

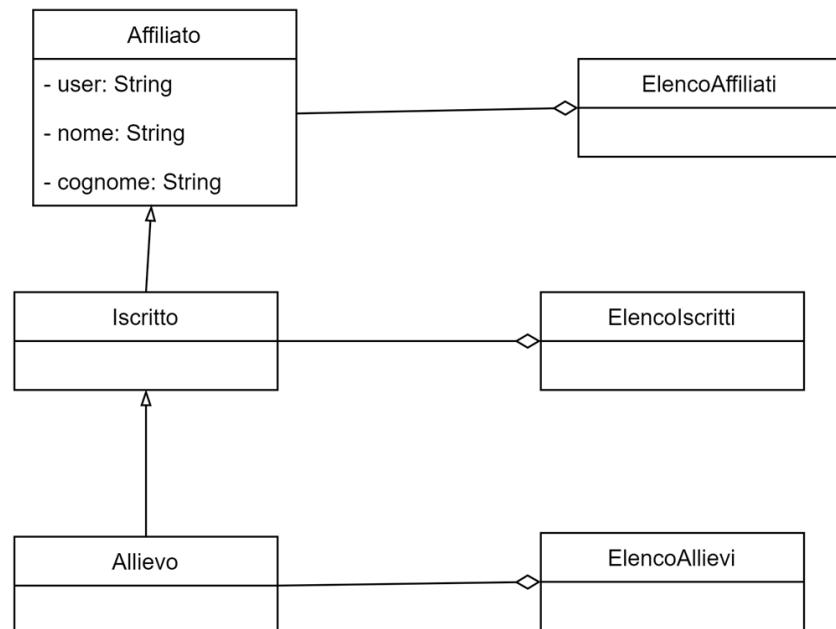
sudojo.server.gestore.model.documento

| Documento |
|---------------------------|
| - oggetto: String |
| - data: Date |
| - userMittente: String |
| - nomeMittente: String |
| - cognomeMittente: String |
| - allegato: byte[] |

sudojo.server.gestore.model.calendario



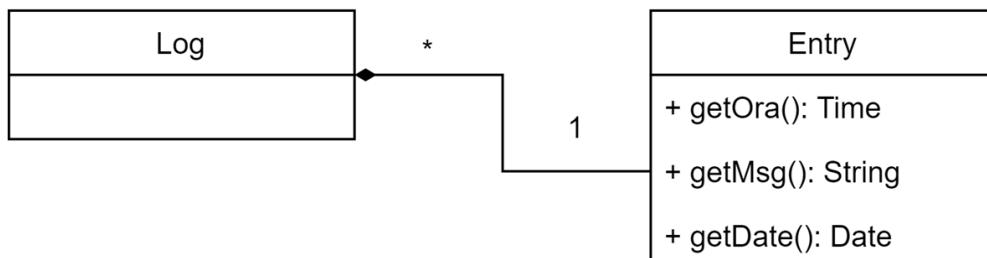
sudojo.server.gestore.model.elenchi



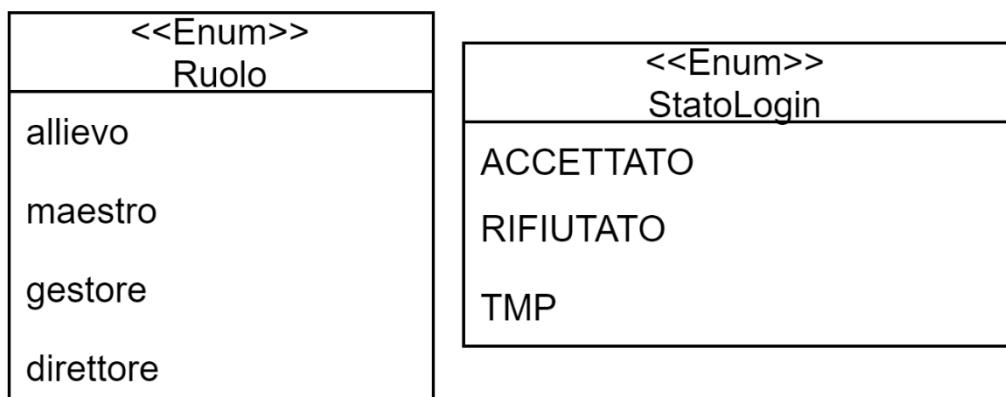
sudojo.server.ricevuta.model

| Pagamento | Ricevuta |
|--|--|
| - data: Date - periodo: Period - importo: float - CFPagante: String - cognomePagante: String - userIscritto: String - nomeliscritto: String - cognomeliscritto: String - nomePagante: String | - nomePalestra: String - sedeLegale: String - P.A.: String - importo: float - cognomePagante: String - periodoRiferimento: Period - cognomeliscritto: String - dataNascitalscritto: Date - luogoNascitalscritto: String - dataEmissioneRicevuta: Date - residenzalscritto: String - causale: String - nomeliscritto: String - nomePagante: String |

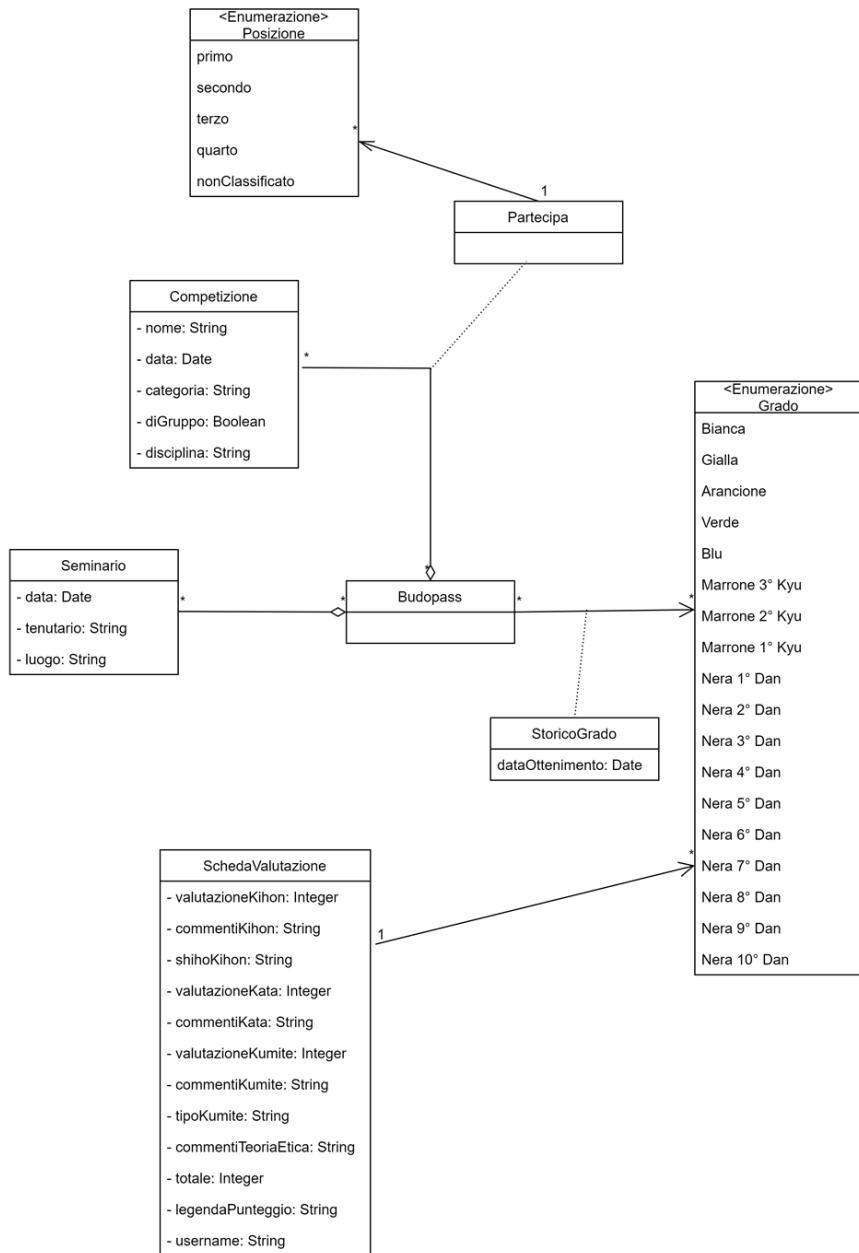
sudojo.server.log.model



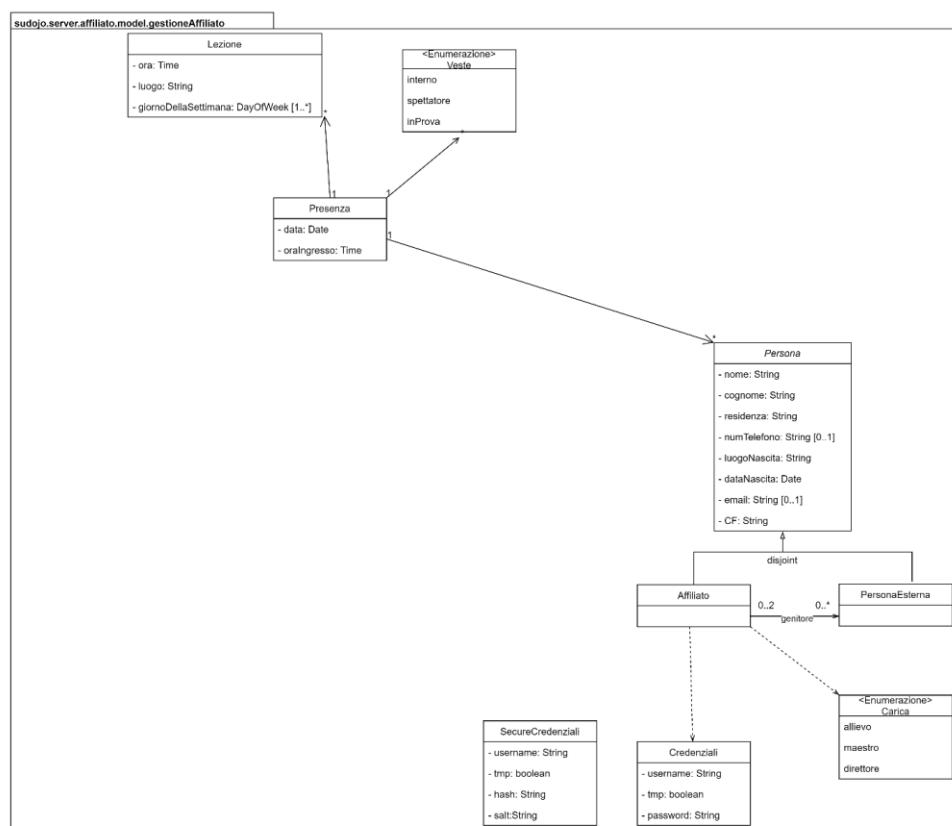
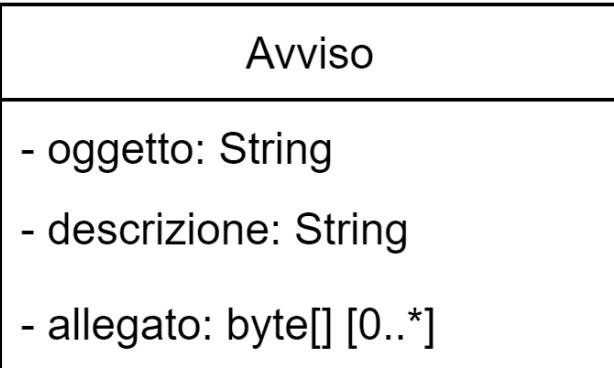
sudojo.server.login.model



sudojo.server.affiliato.model.budopass



sudojo.server.affiliato.model.avviso



PRESENTER:

Lo stereotipo <<rmi>> significa che la classe estende `java.rmi.UnicastRemoteObject` e che implementa la relativa interfaccia della view.

sudojo.server.login.presenter

<<rmi>> LoginRMI

- + `getRuolo(username:String): Ruolo`
- + `rigeneraPassword(username:String):String`
- + `cambiaPassword(username:String, password:String):boolean`
- + `login(username:String, password:String): StatoLogin`

sudojo.server.gestore.presenter

<<rmi>> CalendarioRMI

- + `getEventi(): List<Evento>`
- + `creaEvento(evento:Evento):boolean`
- + `modificaEvento(evento:Evento):boolean`
- + `cancellaEvento(evento:Evento):boolean`

<<rmi>> GetMembriRMI

- + `getIscritti(): ElencoIscritti`
- + `getAffiliato(): ElencoAffiliati`
- + `getAllievi(): ElencoAllievi`

<<rmi>> GetDocumentoRMI

- + `getDoc(inizio:Date, fine:Date):List<Documento>`

<<rmi>>

CreaDocumentoRMI

- + `creaDoc(doc:Documento):boolean`

sudojo.server.ricevute.presenter

<<rmi>>
RicevutaRMI

+ generaRicevute(pagamenti: Collection<Pagamento>): Collection<Ricevuta>

<<rmi>>
PagamentiRMI

+ getAllPagamenti(): Collection<Pagamento>

+ getPagamentiByIscritto(username:String):Collection<Pagamento>

sudojo.server.affiliato.presenter

<<rmi>>
SchedaValutazioneRMI

+ creaSchedaValutazione(scheda: SchedaValutazione):boolean

<<rmi>>
BudopassRMI

+ getBudo(user:String):Budopass

+ aggiornaBudo(user:String, seminario:Seminario):boolean

+ aggiornaBudo(user:String, grado:Grado):boolean

+ aggiornaBudo(user:String, competizione:Competizione, posizione: Posizione):boolean

<<rmi>>
CreaAvvisoRMI

+ creaAvviso(avviso:Avviso, users:Collection<String>):boolean

<<rmi>>
AffiliatoRMI

+ creaAffiliato(affiliato:Affiliato): Credenziali

+ creaAffiliato(affiliato:Affiliato, genitore:PersonaEsterna): Credenziali

+ creaAffiliato(affiliato:Affiliato, genitore1: PersonaEsterna, genitore2:PersonaEsterna): Credenziali

+ modificaAffiliato(affiliato:Affiliato): boolean

+ modificaAffiliato(affiliato:Affiliato, genitore:Genitore): boolean

+ modificaAffiliato(affiliato:Affiliato, genitore1: Genitore, genitore2:Genitore): boolean

+ cancellaAffiliato(username:String): boolean

<<rmi>>
PresenzaRMI

+ getPresenze(inizio:Date, fine:Date): Collection<Presenza>

<<rmi>>
GetAvvisoRMI

+ getAvvisi(inizio:Date, fine:Date, user:String):List<Avviso>

sudojo.server.log.presenter

<<rmi>>
LogRMI

+ getLog(inizio:Date, fine:Date): Collection<Log>

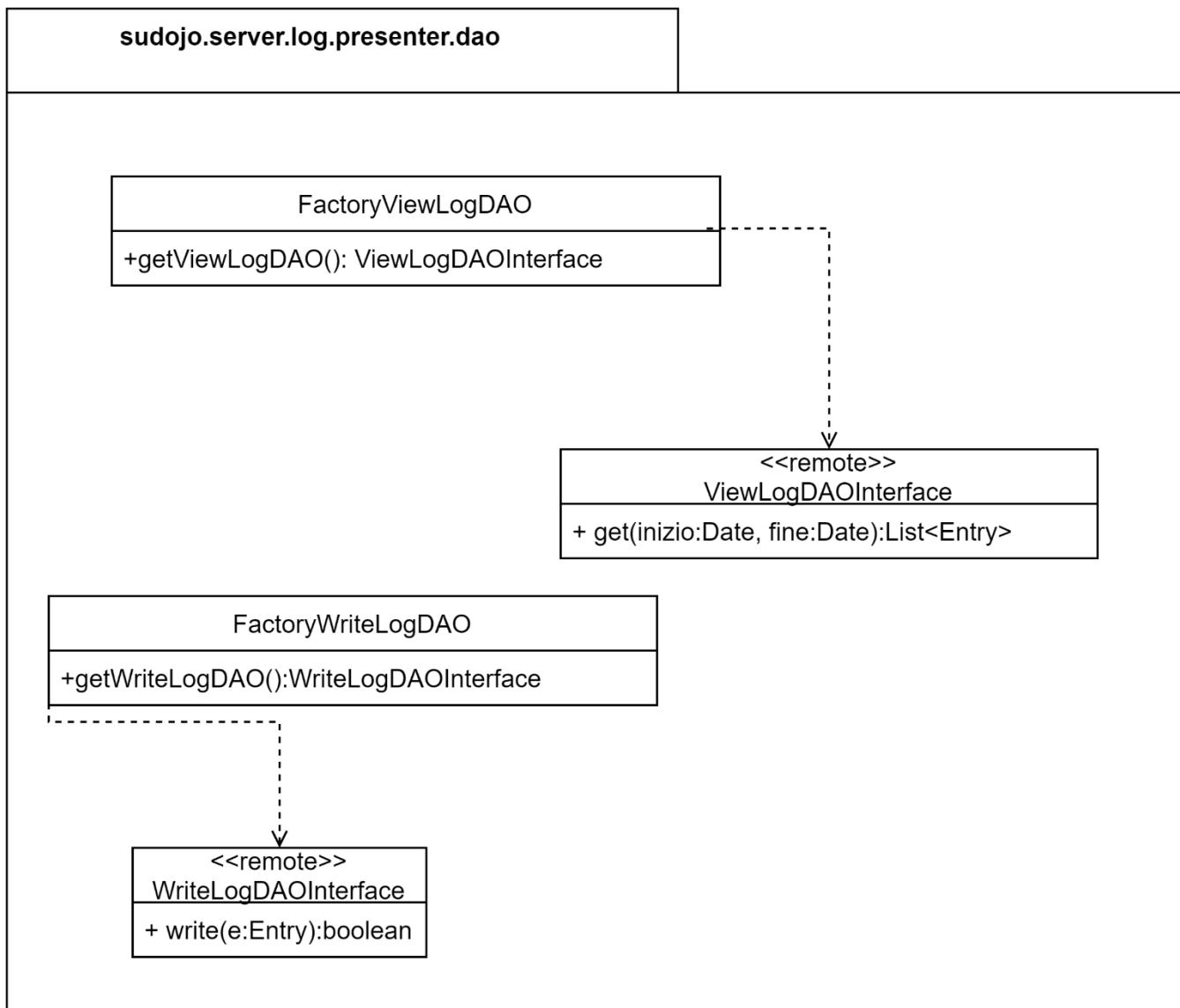
Si riportano ora i package inerenti all'interfacciamento con il broker2 nei presenter dei server.

Come prima, lo stereotipo <<remote>> indica che l'entità è un'interfaccia e che estende `java.rmi.Remote`.

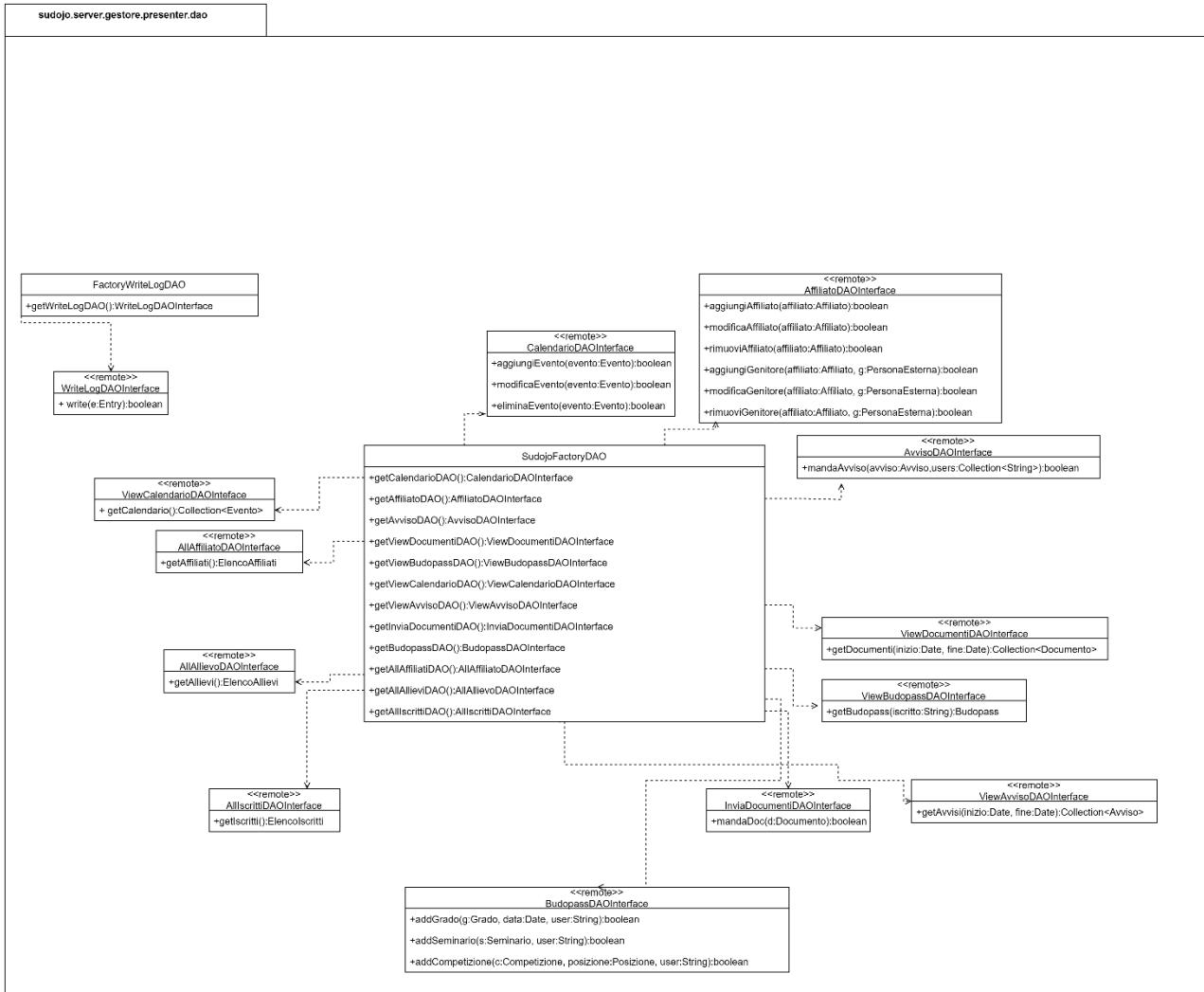
Per il pattern DAO vengono adottate le seguenti factory:

- `FactoryViewLogDAO`, per poter leggere i log
- `FactoryWriteLogDAO`, per poter scrivere i log
- `PagamentiFactoryDAO`, per poter interagire con i pagamenti
- `PresenzeFactoryDAO`, per poter interagire con le presenze
- `CredenzialiFactoryDAO`, per gestire le autenticazioni
- `SudojoFactoryDAO`, per gestire tutte le altre principali operazioni del sistema

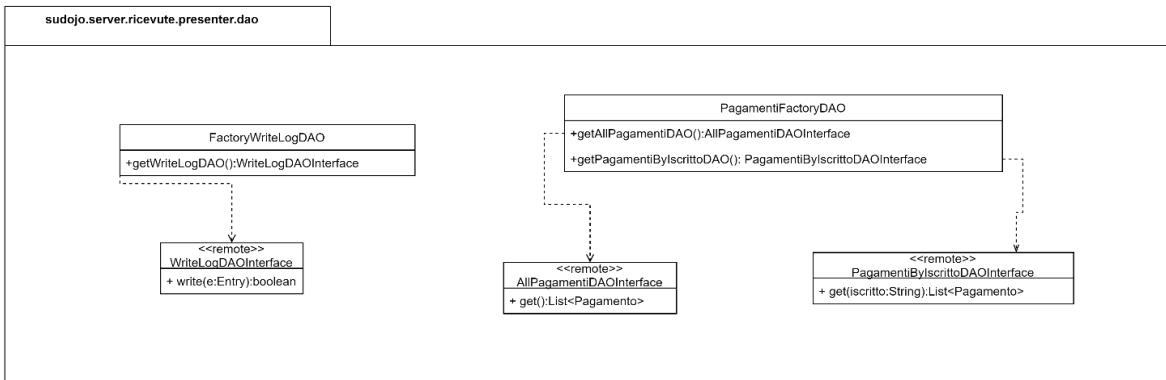
In virtù di modularità ed estendibilità si è deciso di separare il più possibile le factory. Non tutti i server conterranno tutte le factory, ma ogni server contiene solo quelle di cui necessita. In particolare, `FactoryWriteLogDAO` viene inserita in tutti i server in quanto ognuno di essi necessita di poter scrivere log. I server invocano la factory la quale interroga il registry e ritorna l'oggetto remoto sul quale invocare i metodi DAO per interagire con il livello di persistenza.



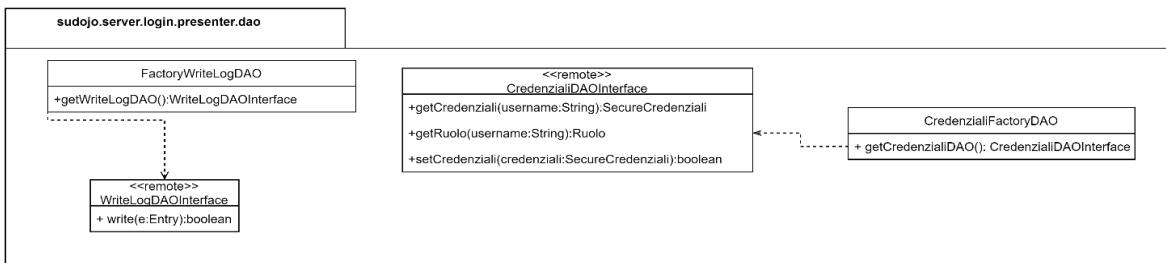
Nel server dei log vengono inserite le factory per poter leggere e scrivere i log.



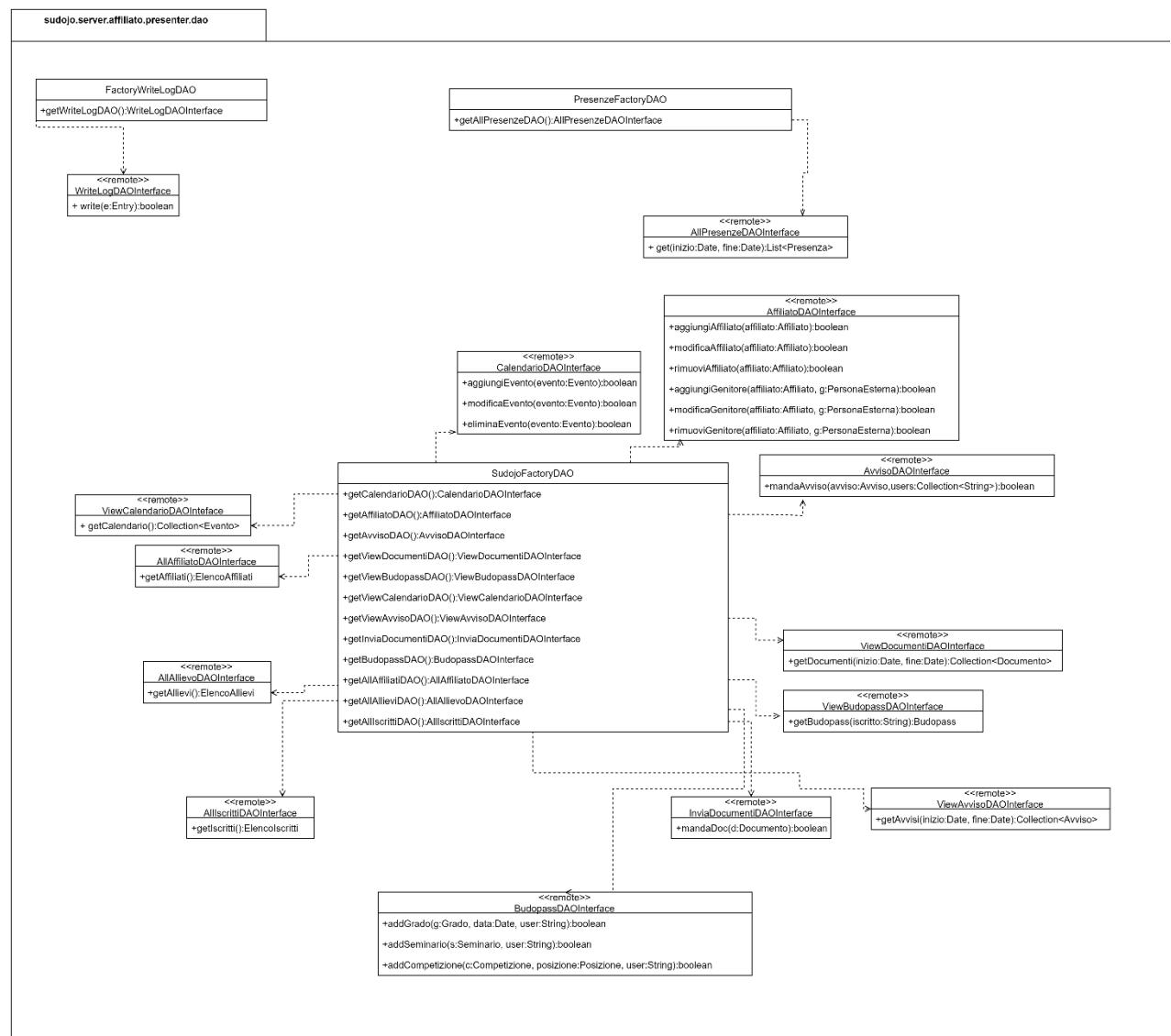
Nel server del gestore vengono inserite le factory `FactoryWriteLogDAO` e `SudojoFactoryDAO`.



Nel server delle ricevute vengono inserite le factory per scrivere i log e per gestire i pagamenti.



Nel server del login vengono inserite FactoryWriteLogDAO e CredenzialiFactoryDAO.



Nel server dell'affiliato vengono inserite FactoryWriteLogDAO, PresenzeFactoryDAO e SudojoFactoryDAO.

VIEW:

Lo stereotipo <<remote>> indica che l'entità è un interfaccia e che estende l'interfaccia java.rmi.Remote.

sudojo.server.log.view

<<remote>>
LogInterfaceRMI

+ getLog(inizio:Date, fine:Date): Collection<Log>

sudojo.server.login.view

<<remote>>
LoginInterfaceRMI

+ getRuolo(username:String): Ruolo
+ cambiaPassword(username:String, password:String):boolean
+ rigeneraPassword(username:String):String
+ login(username:String, password:String): StatoLogin

sudojo.server.ricevute.view

<<remote>>

RicevutanterfaceRMI

+ generaRicevute(pagamenti: Collection<Pagamento>): Collection<Ricevuta>

<<remote>>

PagamentilInterfaceRMI

+ getAllPagamenti(): Collection<Pagamento>

+ getPagamentiByIscritto(username:String):Collection<Pagamento>

sudojo.server.affiliato.view

<<remote>>

SchedaValutazioneInterfaceRMI

+ creaSchedaValutazione(scheda: SchedaValutazione):boolean

<<remote>>

PresenzaInterfaceRMI

+ getPresenze(inizio:Date, fine:Date): Collection<Presenza>

<<remote>>

BudopassInterfaceRMI

+ getBudo(user:String):Budopass

+ aggiornaBudo(user:String, seminario:Seminario):boolean

+ aggiornaBudo(user:String, grado:Grado):boolean

+ aggiornaBudo(user:String, competizione:Competizione, posizione:Posizione):boolean

<<remote>>

CreaAvvisoInterfaceRMI

+ creaAvviso(avviso:Avviso, destinatari: Collection<String>):boolean

<<remote>>

AffiliatoInterfaceRMI

+ creaAffiliato(affiliato:Affiliato): Credenziali

+ creaAffiliato(affiliato:Affiliato, genitore:PersonaEsterna): Credenziali

+ creaAffiliato(affiliato:Affiliato, genitore1: PersonaEsterna, genitore2:PersonaEsterna): Credenziali

+ modificaAffiliato(affiliato:Affiliato): boolean

+ modificaAffiliato(affiliato:Affiliato, genitore:Genitore): boolean

+ modificaAffiliato(affiliato:Affiliato, genitore1: Genitore, genitore2:Genitore): boolean

+ cancellaAffiliato(username:String): boolean

<<rmi>>

GetAvvisoInterfaceRMI

+ getAvvisi(inizio:Date, fine:Date, user:String):List<Avviso>

sudojo.server.gestore.view

<<remote>>
CalendarioInterfaceRMI

- + getEventi(): List<Evento>
- + creaEvento(evento:Evento):boolean
- + modificaEvento(evento:Evento):boolean
- + cancellaEvento(evento:Evento):boolean

<<rmi>>
GetDocumentoInterfaceRMI

- + getDoc(inizio:Date, fine:Date):List<Documento>

<<rmi>>
CreaDocumentoInterfaceRMI

- + creaDoc(doc:Documento):boolean

<<remote>>
GetMembrInterfaceRMI

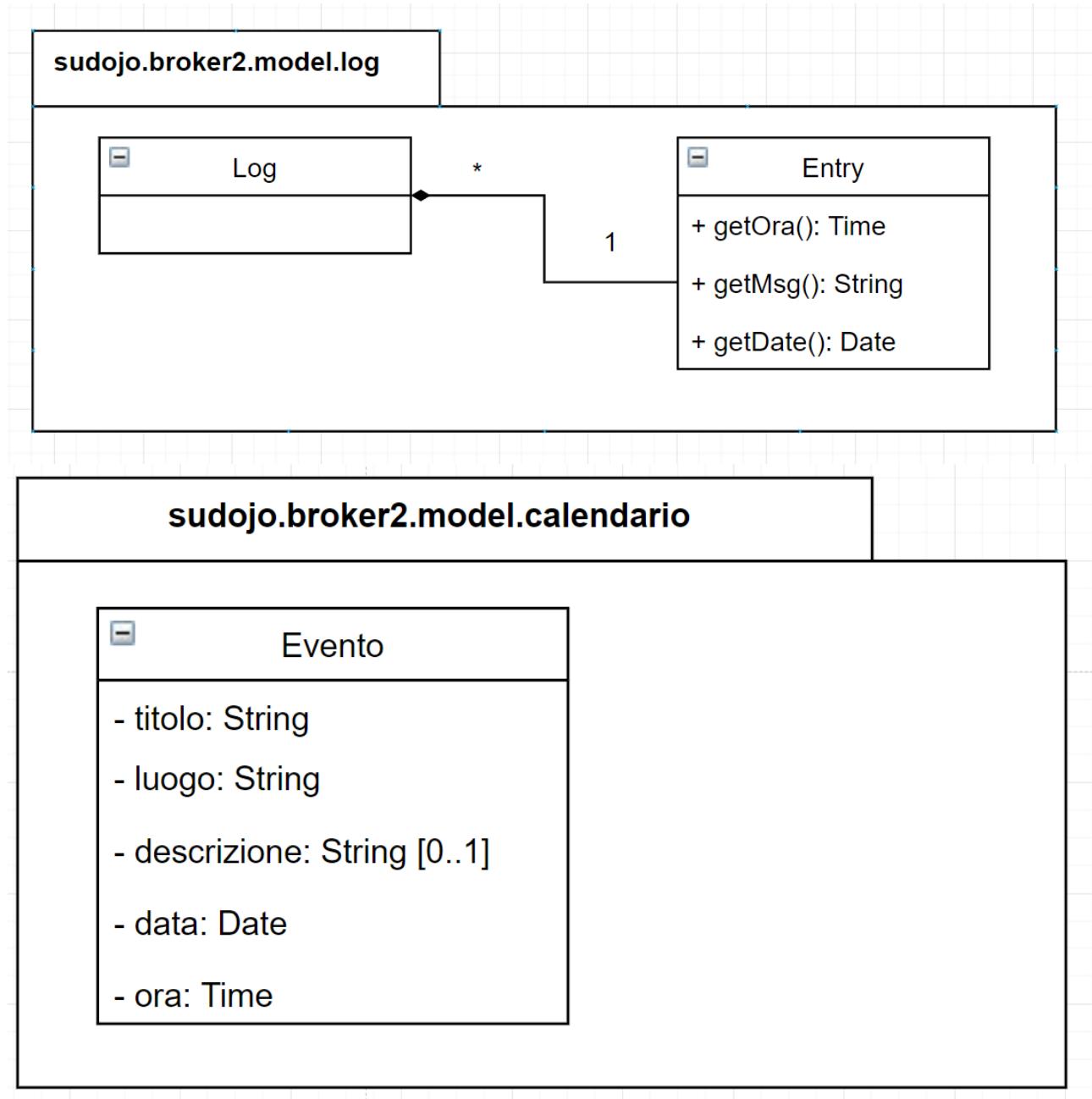
- + getIscritti(): ElencoIscritti
- + getAffiliato(): ElencoAffiliati
- + getAllievi(): ElencoAllievi

Struttura Broker2

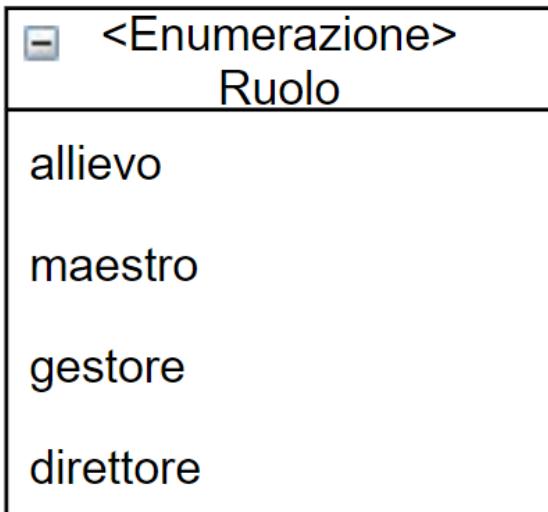
Il broker2 è organizzato secondo il pattern MVP:

- Model, contiene il modello dai dati che servono al broker2
- View, contiene le interfacce RMI
- Presenter, contiene le implementazioni delle interfacce RMI con la logica di persistenza per l'accesso ai DB e ai sistemi esterni

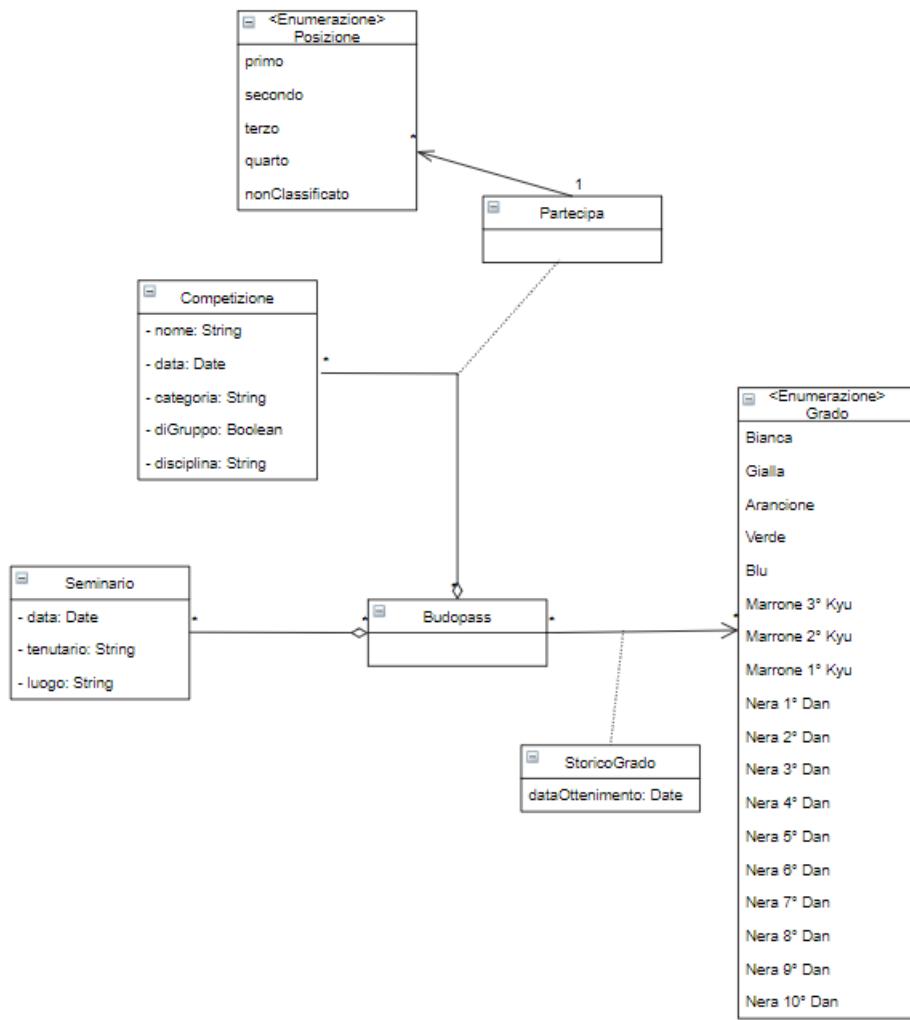
MODEL:

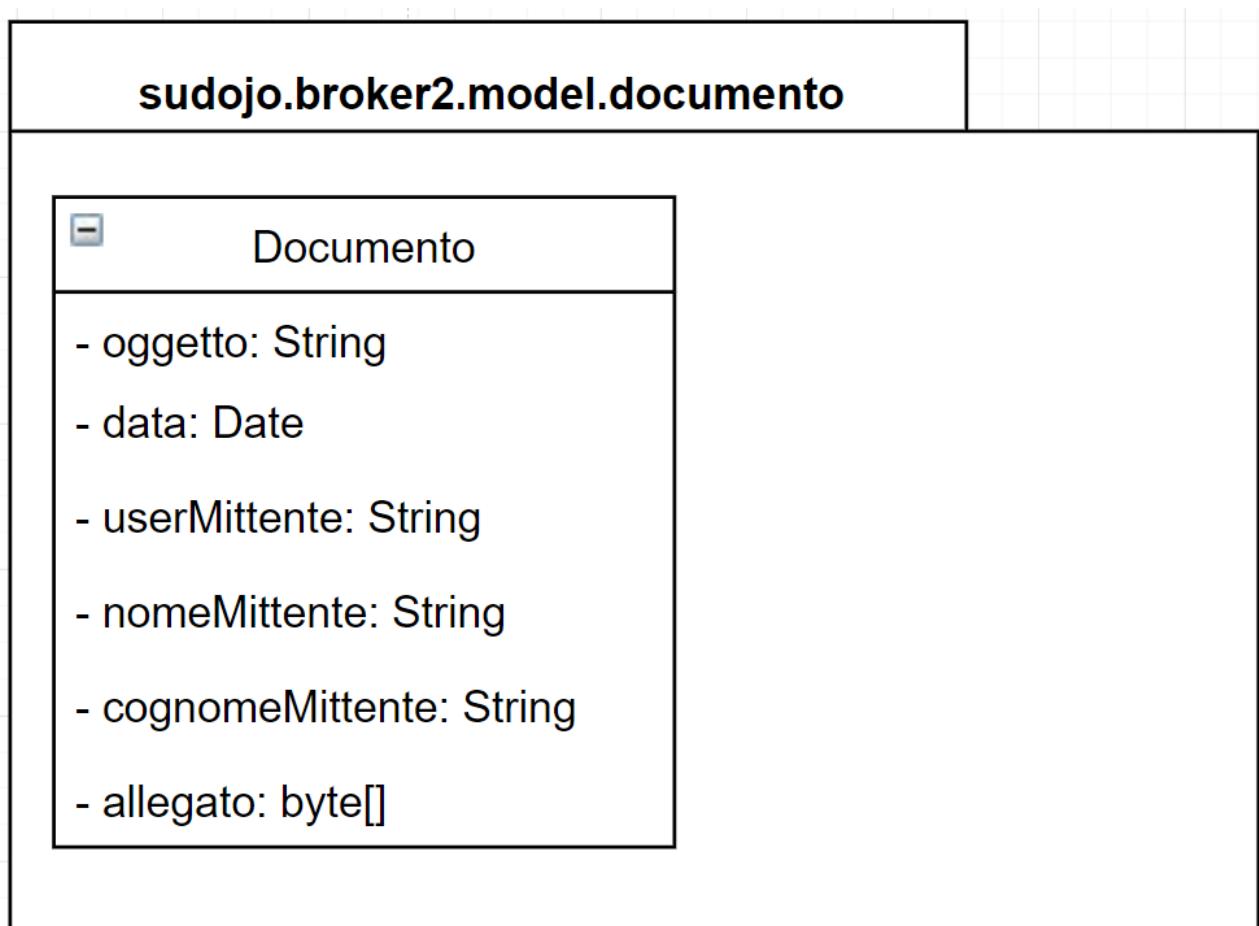
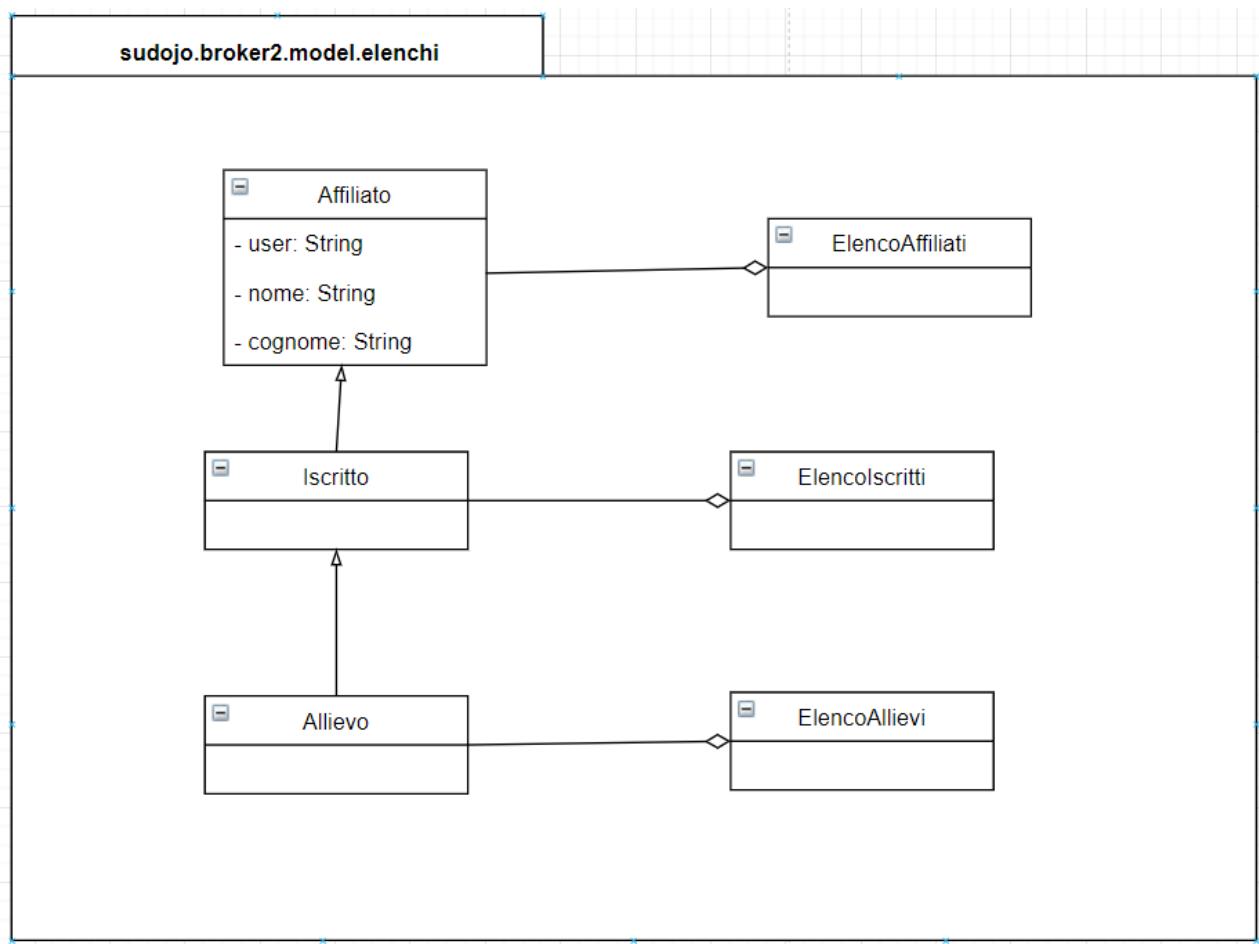


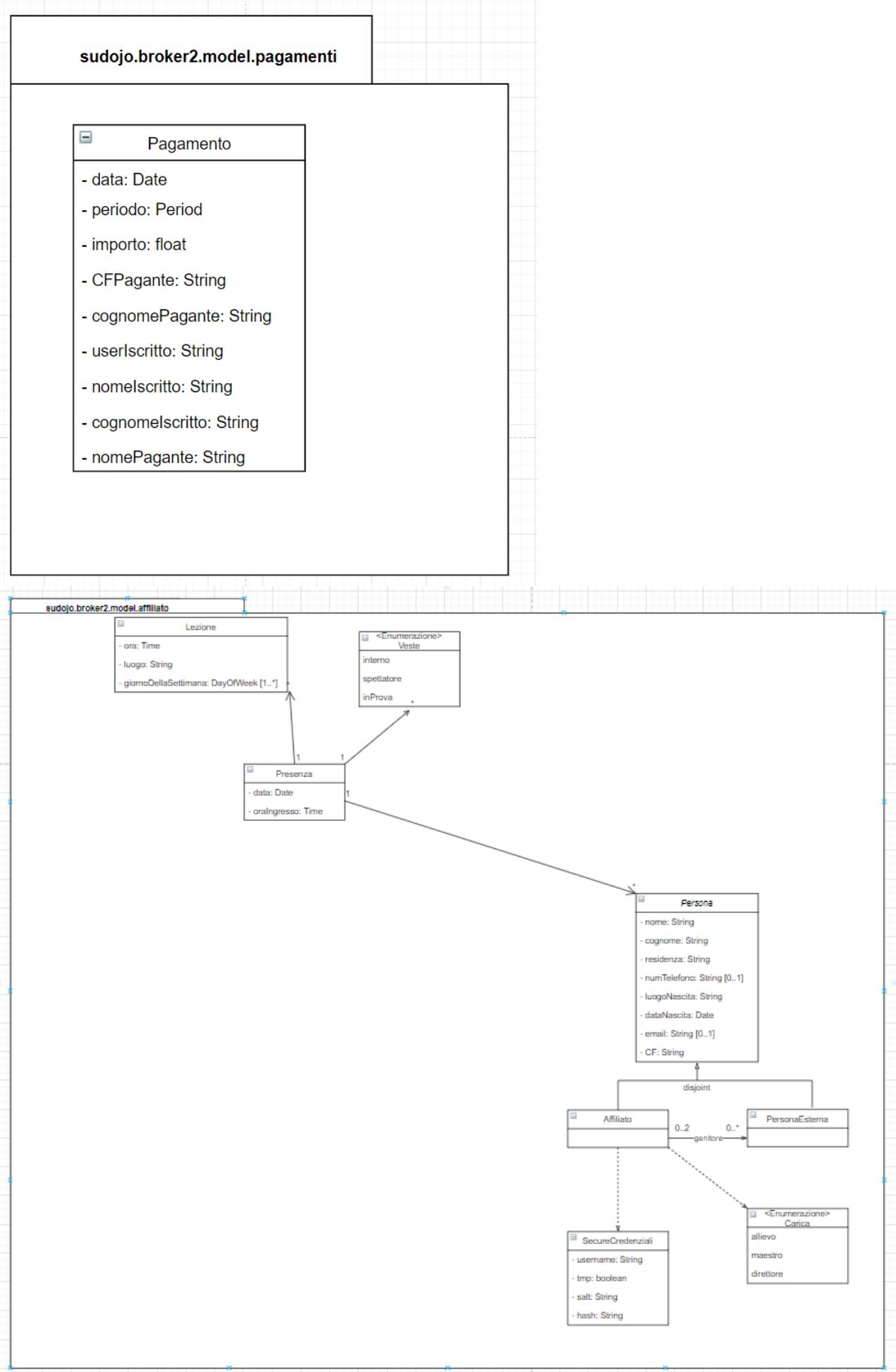
sudojo.broker2.model.login



sudojo.broker2.model.budopass

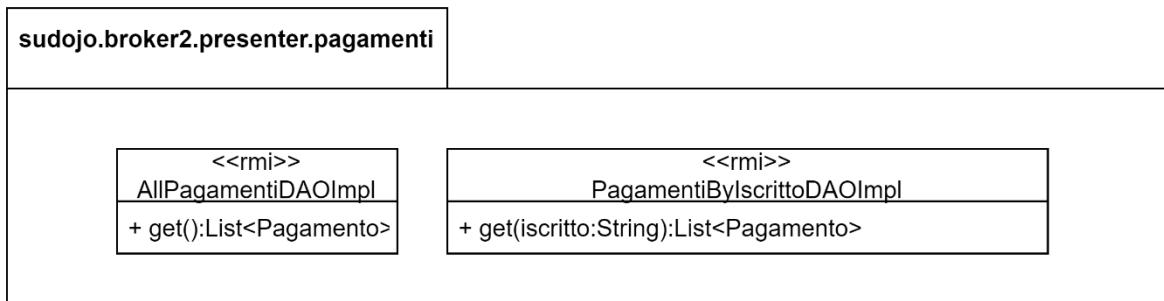
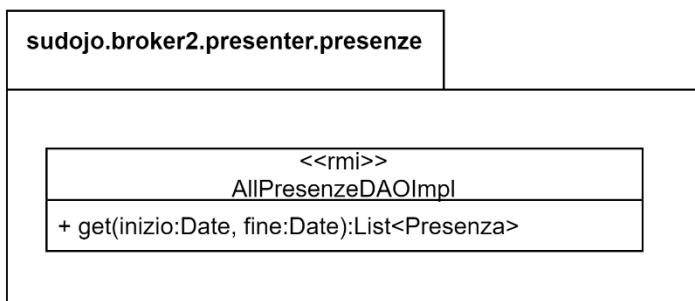
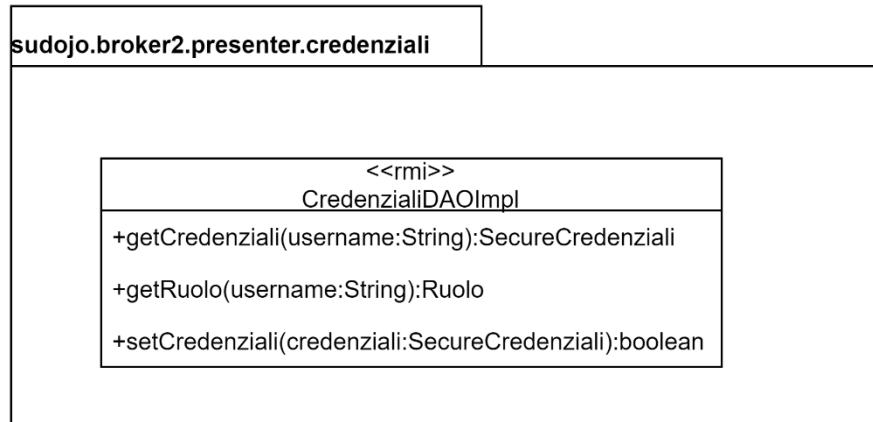


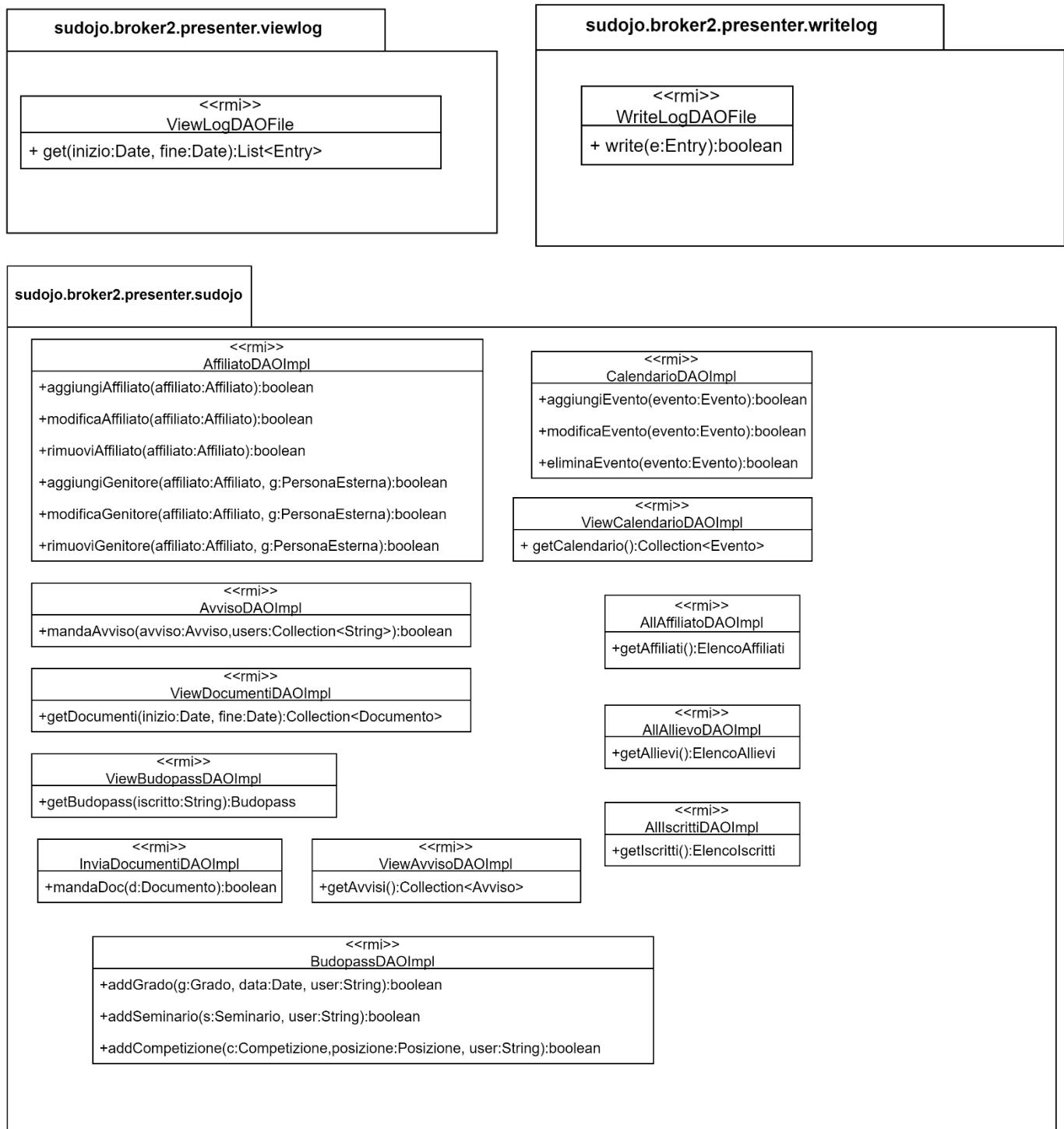




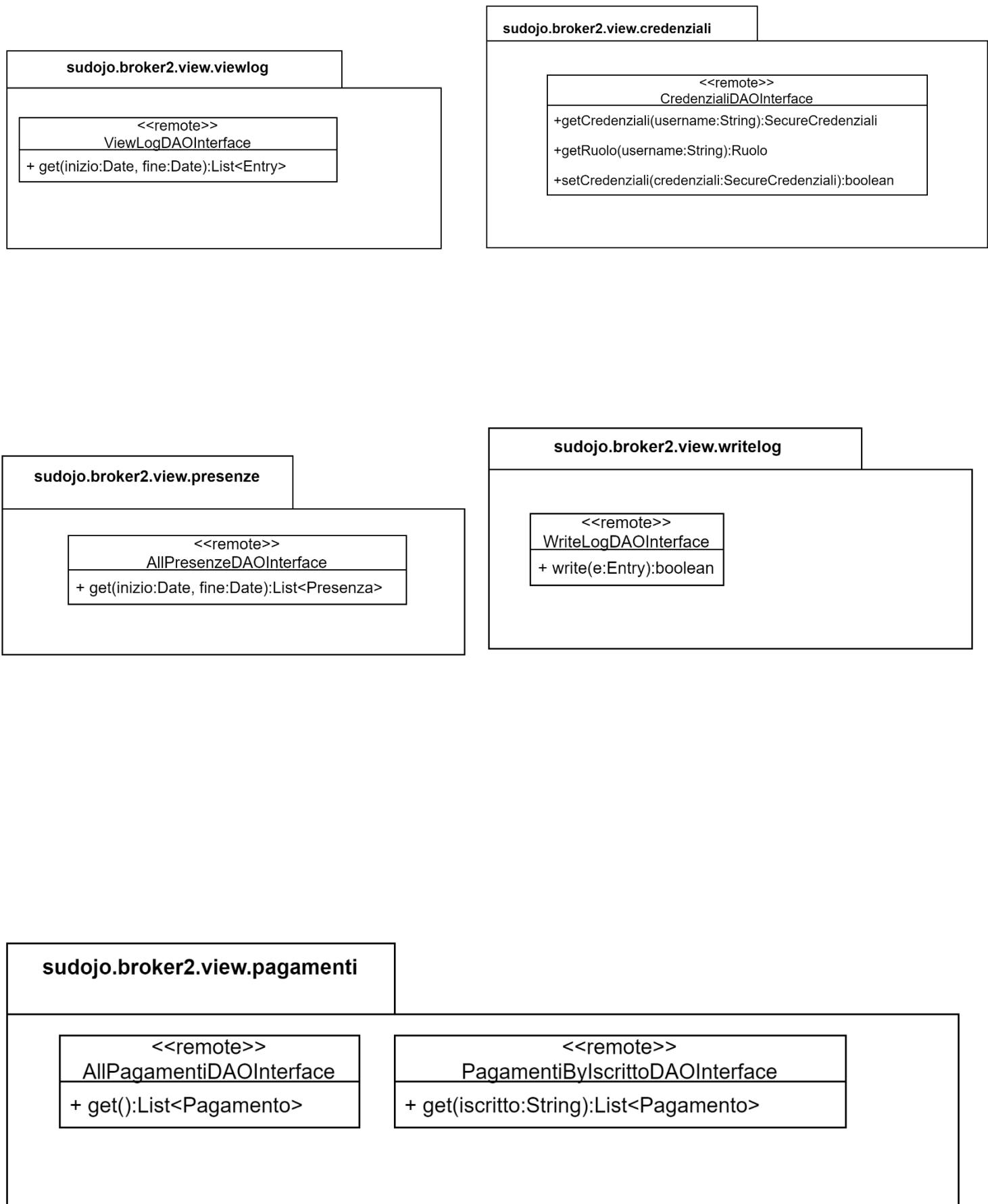
PRESENTER:

Lo stereotipo <<rmi>> indica che la classe estende `java.rmi.UnicastRemoteObject` e che implementa la relativa interfaccia della view.





VIEW:



sudojo.broker2.view.sudojo

<<remote>>
ViewCalendarioDAOInterface
+ getCalendario():Collection<Evento>

<<remote>>
AllAffiliatoDAOInterface
+getAffiliati():ElencoAffiliati

<<remote>>
CalendarioDAOInterface
+aggiungiEvento(Evento:Evento):boolean
+modificaEvento(Evento:Evento):boolean
+eliminaEvento(Evento:Evento):boolean

<<remote>>
AllAllievoDAOInterface
+getAllievi():ElencoAllievi

<<remote>>
AllIscrittiDAOInterface
+getIscritti():ElencoIscritti

<<remote>>
AffiliatoDAOInterface
+aggiungiAffiliato(Affiliato:Affiliato):boolean
+modificaAffiliato(Affiliato:Affiliato):boolean
+rimuoviAffiliato(Affiliato:Affiliato):boolean
+aggiungiGenitore(Affiliato:Affiliato, g:PersonaEsterna):boolean
+modificaGenitore(Affiliato:Affiliato, g:PersonaEsterna):boolean
+rimuoviGenitore(Affiliato:Affiliato, g:PersonaEsterna):boolean

<<remote>>
AvvisoDAOInterface
+mandaAvviso(avviso:Avviso,users:Collection<String>):boolean

<<remote>>
ViewBudopassDAOInterface
+getBudopass(iscritto:String):Budopass

<<remote>>
ViewAvvisoDAOInterface
+getAvvisi(inizio:Date, fine:Date):Collection<Avviso>

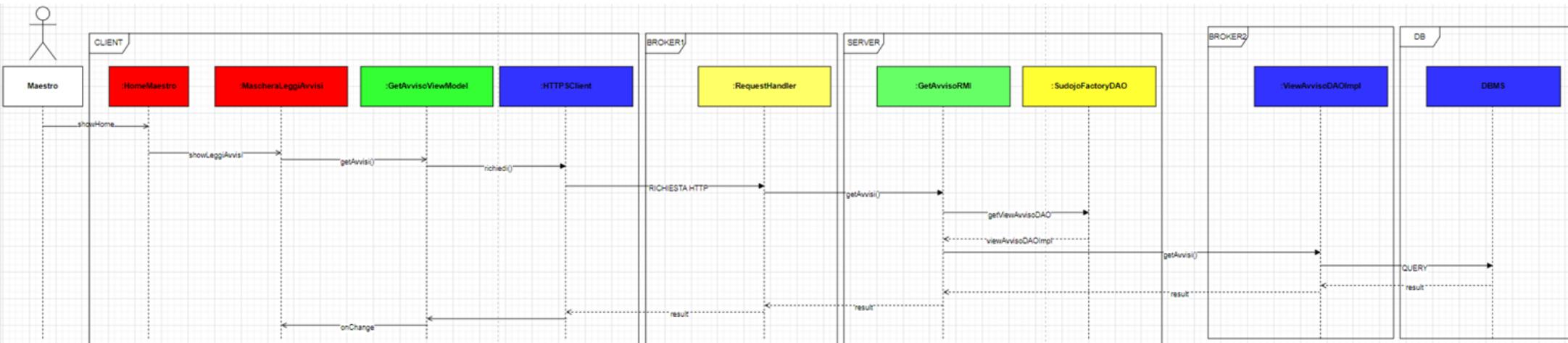
<<remote>>
InviaDocumentiDAOInterface
+mandaDoc(d:Documento):boolean

<<remote>>
ViewDocumentiDAOInterface
+getDocumenti(inizio:Date, fine:Date):Collection<Documento>

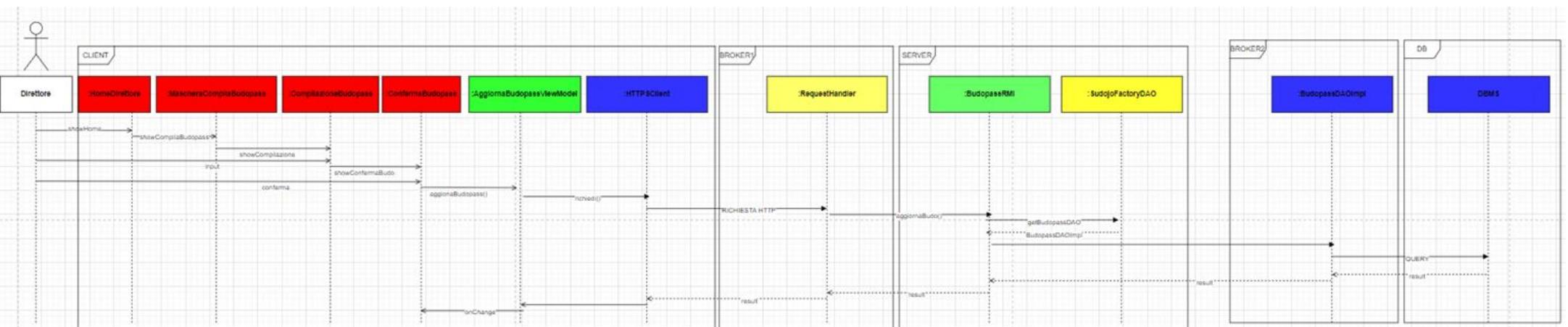
<<remote>>
BudopassDAOInterface
+addGrado(g:Grado, data:Date, user:String):boolean
+addSeminario(s:Seminario, user:String):boolean
+addCompetizione(c:Competizione, posizione:Posizione, user:String):boolean

Si vuole progettare un'architettura modulare, estendibile, manutenibile e orientata al riuso. Tutte le scelte effettuate sono volte a rendere l'applicazione poco rigida, viscosa, immobile e fragile. Si è deciso di creare una struttura estremamente modulare e disaccoppiata tramite l'uso dei due broker che fanno da intermediari favorendo la separazione delle responsabilità. Si è cercato di scorporare le funzionalità in componenti piccoli per favorire riuso, manutenzione e perseguire il principio di singola responsabilità. Nei server e nel broker2 si hanno delle view contenenti solo interfacce, una per ogni classe concreta. Così facendo le view presentano i servizi al livello soprastante tramite l'uso di componenti astratti perseguiendo i principi di inversione delle dipendenze e di interface segregation. Si hanno quindi tante classi concrete, una per ogni funzionalità, ben nascoste e protette. Ognuna di esse ha la propria interfaccia presentata ai clienti del servizio. Si è cercato di perseguire anche il principio open/closed. Ne risulta un sistema modulare, riutilizzabile e fortemente disaccoppiato.

Interazione – Leggi avvisi

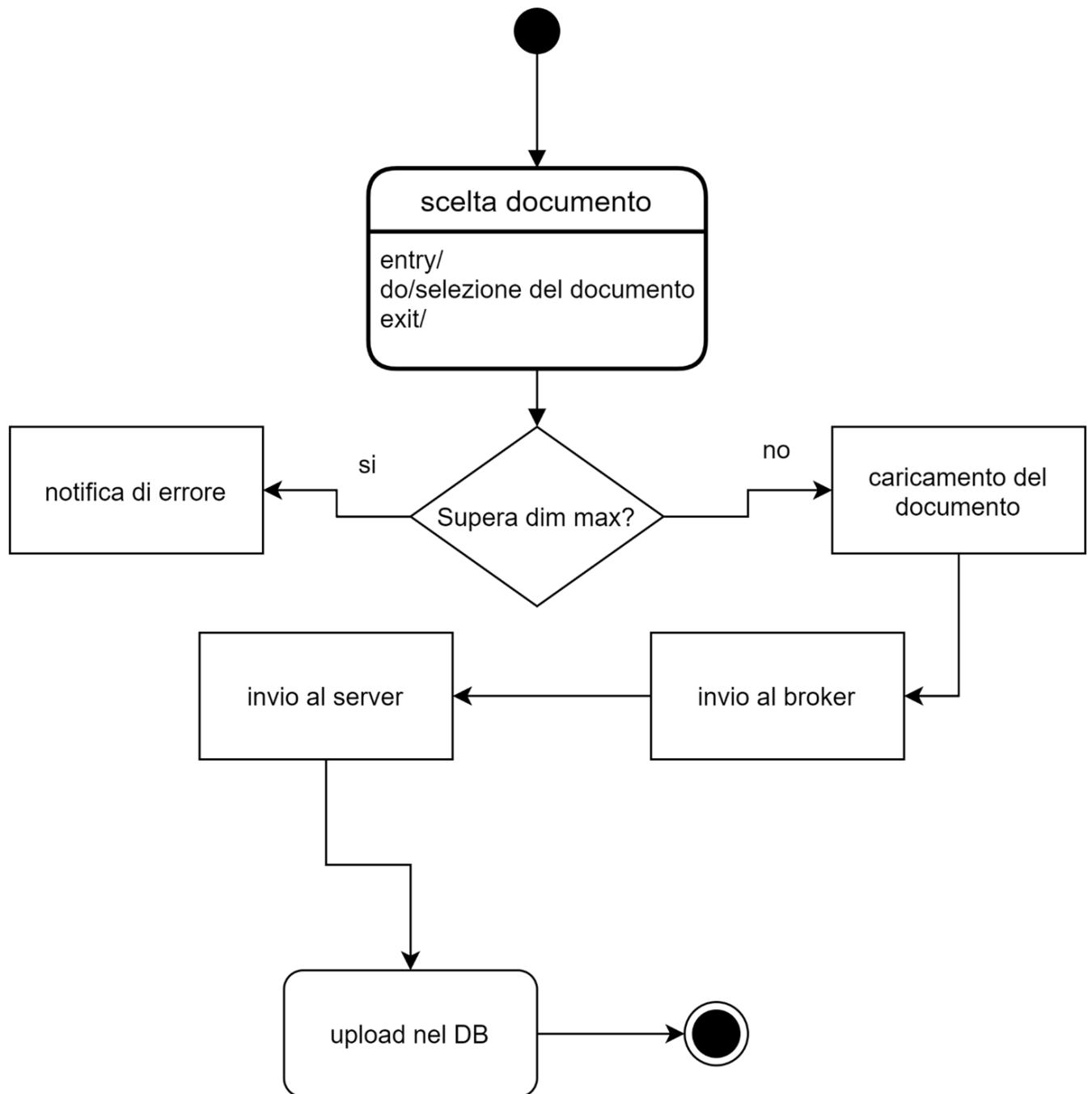


Compila Budopass

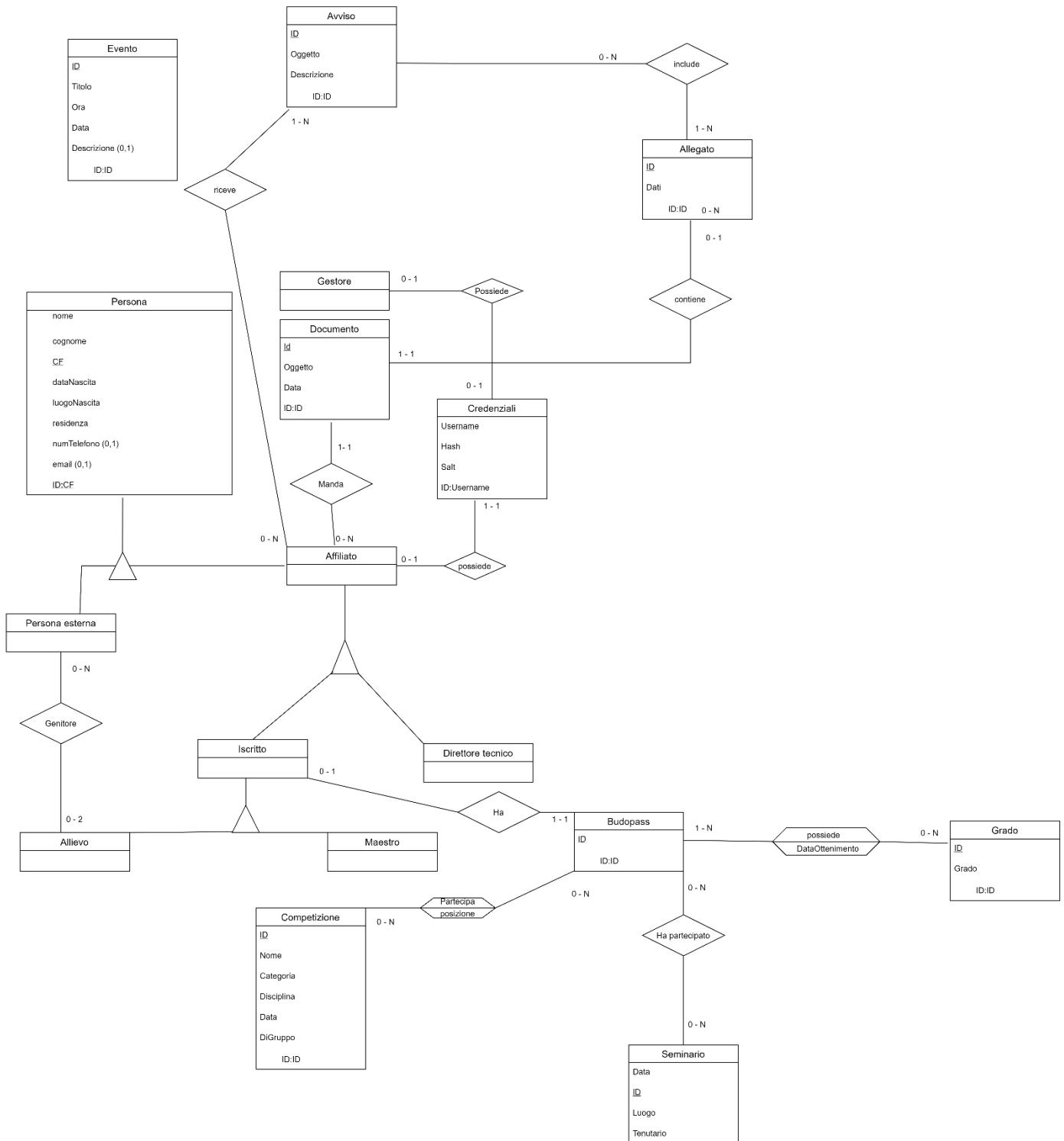


Comportamento

Invia Documenti



Persistenza



Si prevedono i seguenti trigger:

- Nella relazione Partecipa, il campo “posizione” può assumere solo i valori descritti nel dominio (primo, secondo, terzo, quarto, non qualificato)
- Non possono essere inseriti eventi in date passate

Inoltre, si adempie a tali vincoli a livello client impedendo all’utente di inserire valori non validi.

FORMATO DEI LOG

I log vengono salvati come file in un filesystem il cui formato è

<TIMESTAMP> <DESCRIZIONE>

laddove il campo descrizione contiene l'agente del sistema che ha scritto nel log e il messaggio da salvare. Si prevede l'uso di un opportuno demone che si occupi di ricevere le richieste di inserimento e reperimento di log e che abbia i permessi esclusivi sul file.

Collaudo

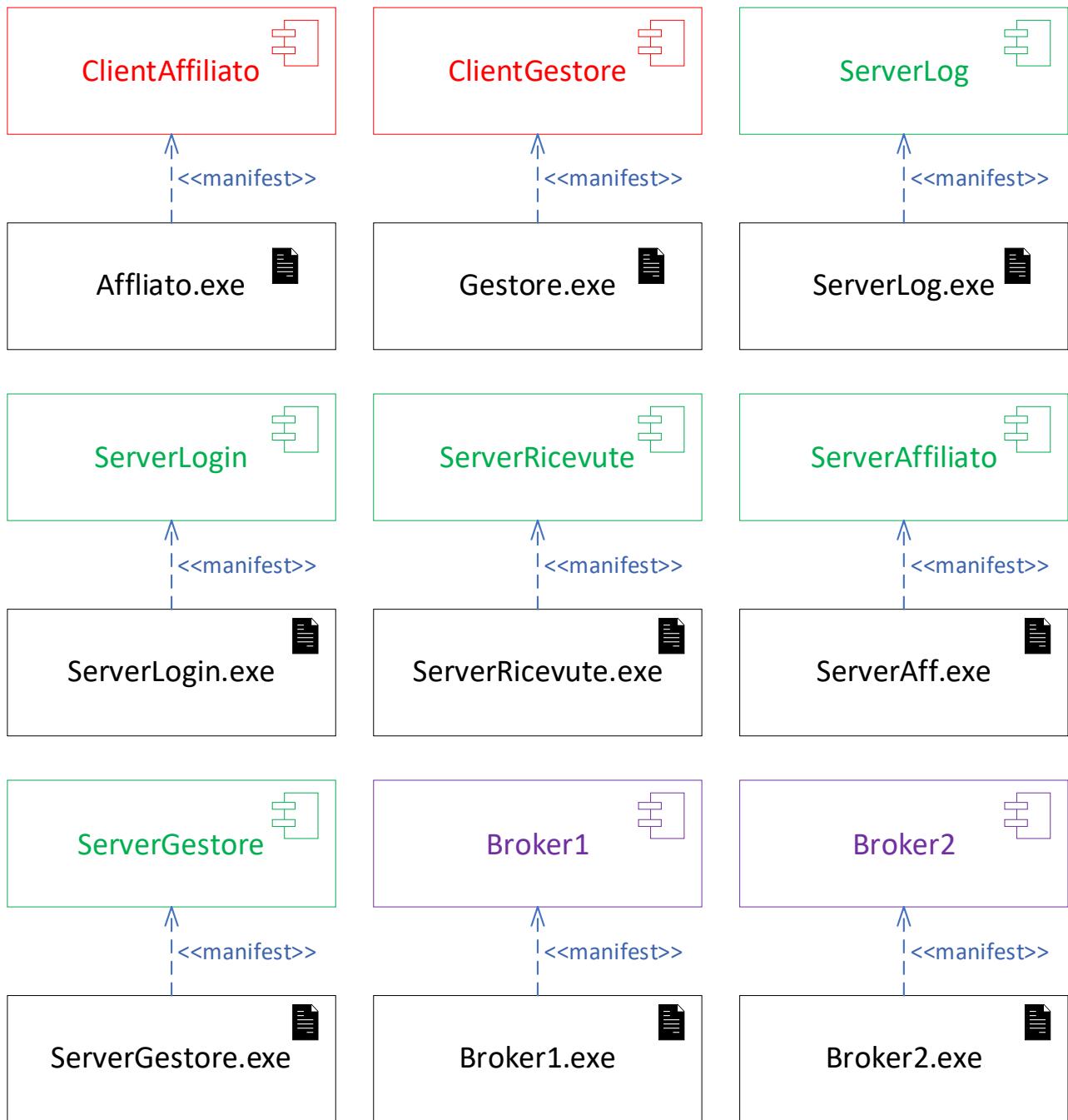
```
public class TestFactoryAffiliato{

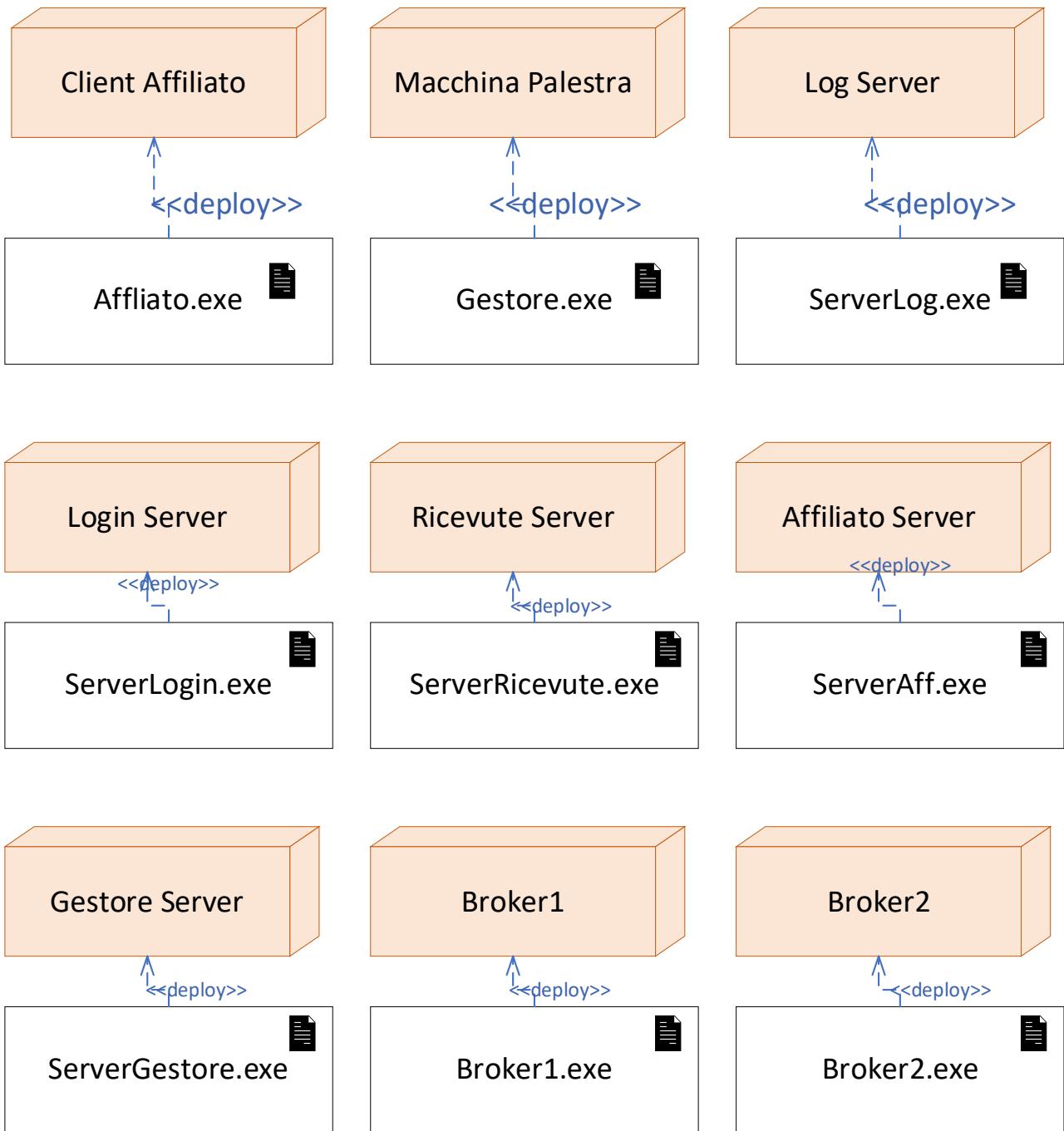
    @Test
    public void test{
        SudojoFactoryDAO factory = new SudojoFactoryDAO();
        AffiliatoDAOInterface affiliatoDAO
=factory.getAffiliatodAO();
        Affiliato aff = new Affiliato("Paolo", "Ciaccia", "via
Antonio Zoccoli 23", "Poggibonsi", new Date(10, 10, 1000),
"CCCPAO00J10G888A");
        affiliatoDAO.aggiungiAffiliato(aff);

        assertEquals(aff.getNome(), "Paolo");
        assertEquals(aff.getCognome(), "Ciaccia");
        assertEquals(aff.getResidenza(), "via Antonio Zoccoli
23");
        assertEquals(aff.getLuogoNascita(), "Poggibonsi");
        assertEquals(aff.getDataNascita(), new Date(10, 10,
1000));
        assertEquals(aff.getCF(), "CCCPAO00J10G888A");

        AllAffiliatoDAOInterface all =
factory.getAllAffiliatiDAO();
        ElencoAffiliati elenco = all.getAffiliati()
        assertEquals(elenco.contains(aff.getCF()), true);
        Affiliato aff2 = elenco.get(aff.getCF());
        assertEquals(aff.getNome(), aff2.getNome());
        assertEquals(aff.getCognome(), aff2.getCognome());
        assertEquals(aff.getResidenza(), aff2.getResidenza());
        assertEquals(aff.getLuogoNascita(),
aff2.getLuogoNascita());
        assertEquals(aff.getDataNascita(),
aff2.getDataNascita());
        assertEquals(aff.getCF(), aff2.getCF());
    }
}
```

Deployment

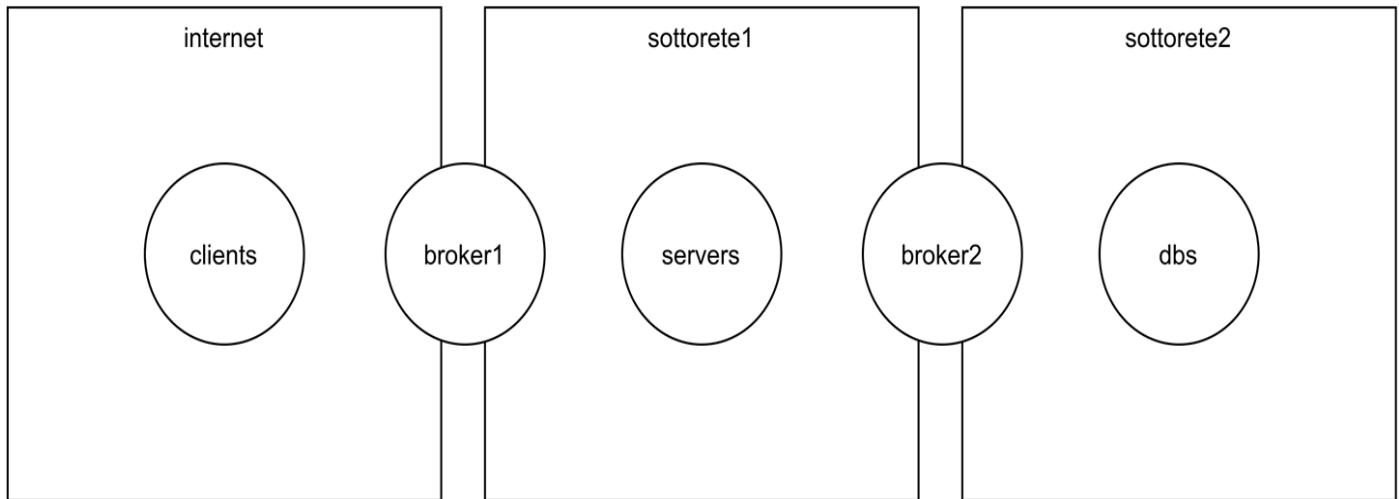




Problematiche

Siccome si utilizza Java RMI, in linea di massima chiunque potrebbe comunicare con il registry e questo rappresenta un inconveniente. Si vuole far sì che solo i server previsti possano registrare servizi nel registry del broker1 e richiedere metodi dal registry del broker2.

Soluzioni



Tutti i server leciti vengono situati in una sottorete della rete privata della palestra detta sottorete1. Inoltre, i registry vengono situati in locale ai rispettivi broker. Per ovviare i problemi di cui sopra si adotta un firewall che permette di impedire le comunicazioni da internet verso il registry1 per fare in modo che quest'ultimo possa comunicare solo con chi è presente nella sottorete1.

Similmente, tramite firewall si fa in modo che solo la sottorete1 possa raggiungere il registry del broker2.

Il broker2 deve potersi interfacciare anche con i sistemi esterni, i quali potrebbero essere collocati in qualsiasi rete, perciò, è opportuno fare in modo che non possano comunicare con il registry del broker2 impedendo ciò tramite firewall. Il firewall può bloccare le comunicazioni che vengono da fuori che utilizzano JRMP, il protocollo usato da Java RMI, per evitare che raggiungano il registry.