ODD: Object Design Document

Riferimento	
Versione	2.0
Data	16/01/2022
Destinatario	Prof. Carmine Gravino
Presentato da	Danilo Aliberti
Approvato da	

Revision history

Data	Versione	Descrizione	Autore
17/11/2021	0.1	Stesura iniziale	Danilo Aliberti
29/11/2021	1.0	Aggiornamento	Danilo Aliberti
14/12/2021	1.1	Aggiornamento	Danilo Aliberti
02/01/2022	1.2	Aggiornamento	Danilo Aliberti
16/01/2022	1.5	Revisione e aggiornamento	Danilo Aliberti

Sommario

1. Introduzione

1.1 Object-design trade-offs

La fase dell'Object Design fa sorgere diversi compromessi di progettazione ed è necessario stabilire quali punti far prevalere e quali invece rendere opzionali. Per quanto concerne la realizzazione del sistema sono stati individuati i seguenti trade-offs:

Buy vs Build: la necessità di sviluppare l'applicazione web in tempi e costi ristretti ci porta ad utilizzare dei componenti off-the-shelf, si sceglie quindi di utilizzare dei framework moderni che siano adatti allo sviluppo della piattaforma in modo da renderla quanto più fedele all'idea iniziale. Tuttavia, si sceglie di limitare l'utilizzo di tali componenti alle tecnologie che si occuperanno del back-end in quanto non si vuole rinunciare ad un'interfaccia ed un'esperienza unica.

Efficienza vs. Tempistiche: La buona riuscita e portata a termine della realizzazione dell'applicazione necessita che gli sviluppatori abbiano acquisito le conoscenze necessarie all'utilizzo delle tecnologie scelte per l'implementazione; solo in questo modo si garantirà come risultato finale un sistema efficiente e di qualità. Per questa ragione si terrà conto delle eventuali tempistiche necessarie agli sviluppatori per consentire loro tale acquisizione; si prioritizzerà quindi l'efficienza, a discapito della tempistica.

Comprensibilità vs Costi: Una documentazione completa e dettagliata garantirà la comprensibilità del codice dell'applicazione per chi non ne ha partecipato alla creazione e ad eventuali nuovi sviluppatori coinvolti in un secondo tempo ed estranei all'originaria implementazione. A rispondere a questa esigenza sarà l'utilizzo diffuso, nel codice, di commenti, con lo scopo di facilitarne la comprensione e di conseguenza la fase di eventuale modifica. La comprensibilità del codice sarà quindi un elemento prioritario, a cui non verranno applicati gli effetti di eventuali compromessi economici.

1.1.1 Componenti off-the-shelf

Il back-end sarà costruito utilizzando il framework **Django,** nota soluzione nell'ambito delle applicazioni distribuite in Python. Composto da un core ben ottimizzato e prodotto con l'obiettivo di ridurre il tempo di sviluppo delle applicazioni, consentendo agli sviluppatori di concentrarsi maggiormente sulla logica di business dell'applicazione, piuttosto che sulla comunicazione tra le varie componenti. Sarà utilizzata una delle ultime versioni, ovvero la **3.2.6**, in questo modo si potrà

utilizzare tutta la potenza e flessibilità del framework, oltre ad utilizzare numerose librerie compatibili con questa versione. I moduli di interesse per questo progetto saranno:

- **Django ORM**;
- **Django Security**, che si occupa della sicurezza dell'intero sistema;
- **Django All-Auth**, una libreria che integra i sottosistemi di registrazione e autenticazione via Social Network;
- **Django Template Engine**, che sarà utilizzato per il front-end;
- **Django Admin Site,** utilizzato per l'amministrazione generale di tutto il framework. Permette la gestione e la modellazione di tutte le entità.

1.1.2 Riuso e flessibilità

1.1.2.1 Design patterns

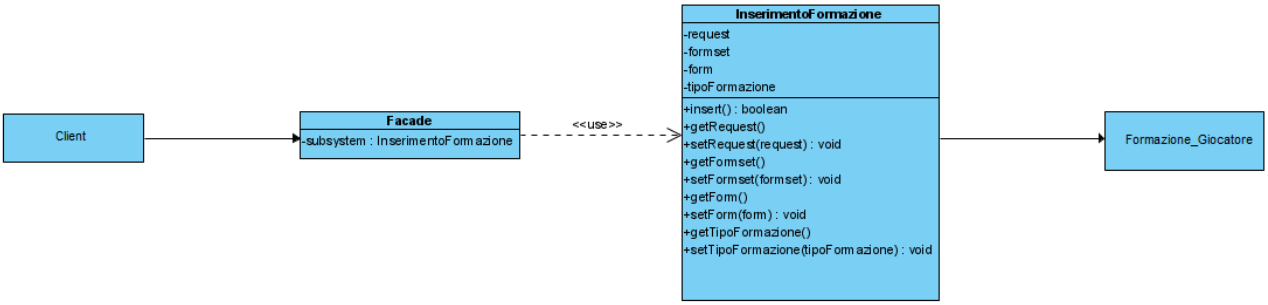
Per rendere più rapida la fase di sviluppo, abbiamo ricorso all'utilizzo di soluzioni che risolvono problemi ricorrenti e prevedibili, presenti anche nel nostro sistema. Di seguito sono elencati i design pattern utilizzati in questo progetto, suddivisi secondo la classificazione GoF e schematizzati mettendo in evidenza la descrizione del pattern, del problema che questo risolve e la sua soluzione.

1.1.2.1.1 Design patterns strutturali

Facade

Biaset fa uso del design pattern Facade per nascondere la complessità dell'operazione di inserimento della formazione (titolare e riserve).

- **Descrizione:** fornisce un'unica interfaccia per accedere ad un insieme di oggetti che prendono parte a blocchi di codice complessi.
- **Soluzione:** viene definita una classe ad alto livello (InserimentoFormazione) che semplifica l'utilizzo delle funzionalità, andando a gestire il controllo dell'inserimento della formazione titolare e delle riserve, controllando eventuali duplicati inseriti e appartenenza del giocatore alla squadra che sta inserendo la formazione.

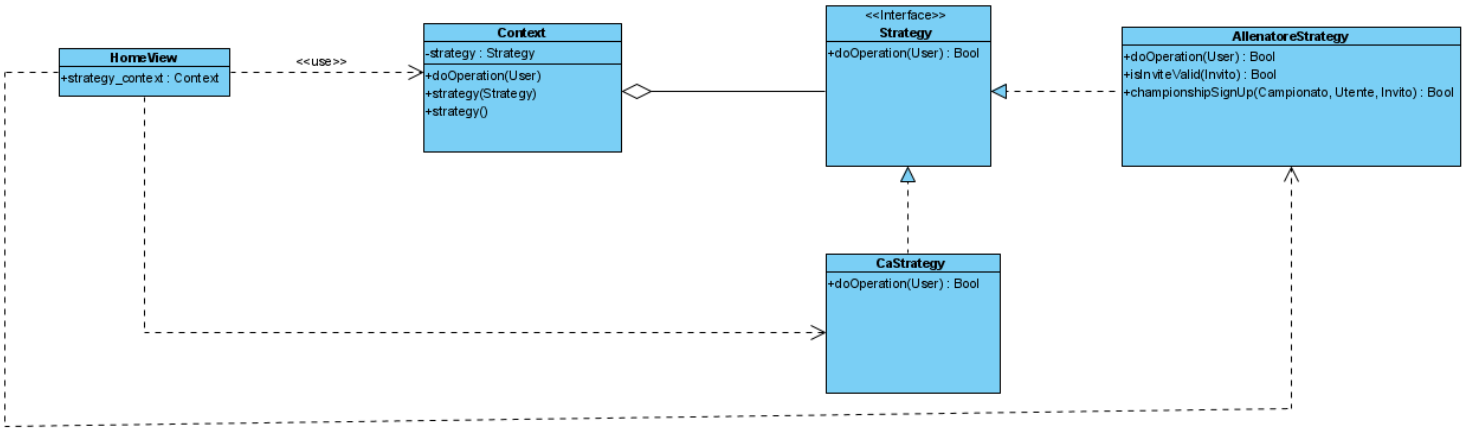


1.1.2.1.2 Design patterns comportamentali

Strategy

Lo Strategy Pattern è utilizzato per gestire a runtime la scelta del profilo utente. Quest'ultimo, una volta effettuato il primo accesso alla piattaforma, dovrà decidere se creare un proprio campionato e, di conseguenza, avere un profilo da Championship Admin associato, oppure se partecipare ad un campionato già esistente (profilo Allenatore).

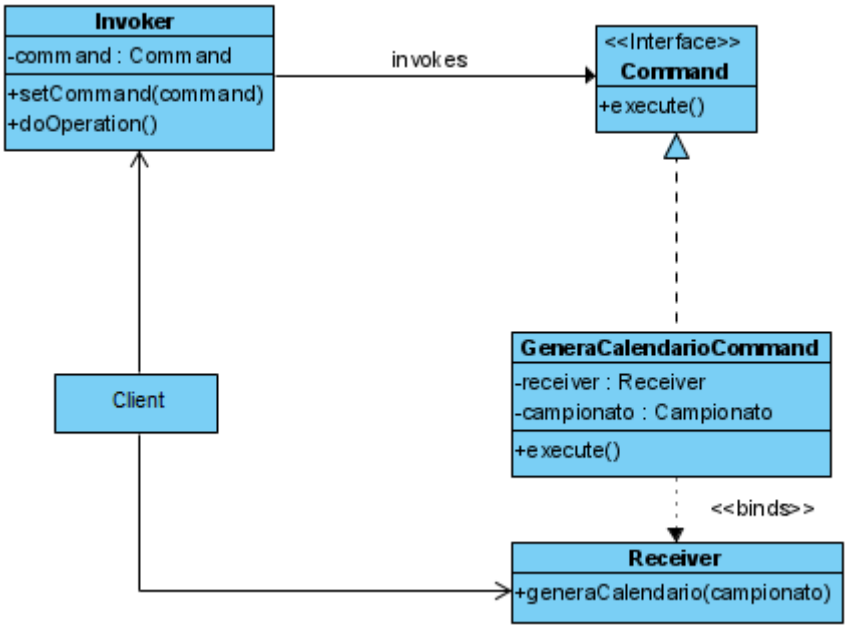
- **Descrizione:** questo pattern prevede che gli algoritmi siano intercambiabili tra loro, in base ad una specificata condizione, in modalità trasparente al client che ne fa uso.
- **Soluzione:** vengono definite "due strade" possibili per i due tipi di registrazione. Nella classe StrategyRegistrationView, il parametro "regtype" passato dal dispatcher stabilisce la scelta che dovrà essere effettuata. Successivamente, verrà inizializzato l'oggetto un oggetto di tipo Context con la strategia selezionata. Infine, il context chiamerà il metodo doOperation di una delle due classi che implementano l'interfaccia Strategy.



Command

Biaset fa uso del Command Pattern per la generazione automatica del calendario degli scontri.

- **Descrizione:** il pattern trasforma una richiesta in un oggetto stand-alone che contiene tutte le informazioni della richiesta stessa. Questa trasformazione fa sì che sia possibile passare richieste come parametri di metodi, ritardare o accodare un'esecuzione di una richiesta.
- **Soluzione:** viene definita un'interfaccia Command, implementata dal comando "concreto", ovvero GeneraCalendarioCommand. Quest'ultimo delegherà la richiesta al Receiver con la chiamata al metodo generaCalendario(), il quale effettuerà l'operazione richiesta.



1.1.2.1.3 Altri pattern

Model View Template

Il pattern è basato sulla separazione dei compiti fra i componenti software che interpretano tre ruoli principali:

- Il **Model** fornisce i metodi per accedere ai dati persistenti utili all'applicazione;
- Il **View** contiene la logica di business e interagisce col Model per prelevare i dati da inviare ad un Template;
- Il **Template** non è altro che il presentation layer, che gestisce completamente le user interfaces.

La struttura e la spiegazione di come è usato questo pattern (insieme alle motivazioni che hanno portato al suo utilizzo) sono riportati nel documento SDD.

1.2 Linee guida per la documentazione d'interfaccia

Agli sviluppatori, in fase di implementazione del sistema, è richiesto di attenersi a delle linee guida illustrate di seguito, al fine di rispettare una consistenza e coerenza di forma e facilitare la comprensione di ognuna delle funzionalità messe a punto.

Nomenclatura

Classi: ad ogni classe deve essere assegnato un nome che possa associarla in modo univoco e distinguibile all'entità di dominio, funzionalità o servizio a cui fa riferimento, è stato scelto inoltre di seguire la notazione Camel Case; in particolare, i nomi delle classi dovranno iniziare con lettera maiuscola, i nomi di metodi e attributi con lettera minuscola.
Metodi: ogni metodo segue la notazione in lowerCamelCase, fatta eccezione per quelli delle classi di Test e quelle del framework Django, che seguiranno la notazione **snake_case**, tipica di Python.
Eccezioni: il nome dell'eccezione, se creata, deve rispecchiare il problema che segnala.

Indentazione

Il codice Python dev'essere indentato in maniera appropriata (tramite quattro spazi). La prima riga di codice non può essere indentata. Il numero di spazi bianchi dev'essere uniforme in un blocco di codice. L'indentazione è fondamentale per definire blocchi di statements.

Organizzazione delle componenti

I nomi dei files (ove possibile), delle operazioni e delle variabili devono essere significativi e permetterne l'immediata identificazione e lo scopo.
Tutte le classi che realizzano un sottosistema devono essere racchiuse nello stesso package Python.
Tutte le risorse statiche, invece, devono essere collocate nella cartella "static".

Pagine HTML

Il codice HTML, sarà utilizzato per definire la struttura delle pagine dell'applicazione che verrà realizzata, mentre la parte di stile sarà realizzata con linguaggio CSS. La versione di riferimento che verrà utilizzata è la versione 5. Il codice dovrà essere indentato come nell'esempio:

```
<html>
        <head>
</head>
 <body>
```

</body>
</html>

Fogli di stile CSS

La versione di riferimento scelta per l'utilizzo di questo linguaggio è la 3. Per quanto concerne gli stili in comune che potranno essere utili in più punti devono essere inseriti in un foglio di stile globale, mentre gli stili definiti per una singola pagina vanno inseriti nei fogli di stile CSS presenti nella stessa cartella dove sono contenuti i file HTML a cui si riferiscono. Biaset utilizza il framework CSS **Bootstrap** per sviluppare le componenti dell'interfaccia utente.

Organizzazione del repository

Infine, sono definite le convenzioni per condividere il proprio lavoro attraverso un repository Git. Ogni membro del team è tenuto a realizzare il proprio codice coordinandosi con gli altri membri del team, utilizzando, per semplicità, un unico master branch.

1.3 Definizioni, acronimi e abbreviazioni

ODD: Object Design Document
Componente Off-The-Shelf: Servizi esterni di cui viene fatto utilizzo da terzi.
Framework: Software di supporto allo sviluppo.
HTML: Linguaggio di Mark-up per pagine web.
CSS: Linguaggio usato per definire la formattazione di pagine web.
Django: framework open source per lo sviluppo di applicazioni web in Python.
Bootstrap: framework per lo sviluppo di interfacce grafiche web.
App: applicazione interna al progetto, sottosistema.

1.4 Riferimenti

- **Design goals:** sezione 1.2 del SDD;
- **Scelta dell'ambiente di esecuzione:** sezioni 3.2 e 3.3 del SDD;
- **Django:** documentazione;

2. Packages

Il design della struttura segue quello imposto dal framework utilizzato (Django): il progetto è organizzato in **app.** I files di configurazione del framework saranno contenuti nella cartella **biaset/settings**. Saranno previsti 2 tipi di files di configurazione:

- *dev.py,* utilizzato in ambiente di sviluppo;
- *prod.py,* che sarà utilizzato in produzione.

Un'app non rappresenta altro che un sottosistema del progetto; ad esempio, il sottosistema Gestione Utenza sarà contenuto nell'app "**gestioneutenza**", nella cartella **gestioneutenza**.
All'interno di ogni app, le Views associate saranno contenute nel file *views.py*; i modelli, invece, nel file *models.py;* i test in *tests.py* e così via. I design pattern avranno le loro strutture in cartelle a loro dedicate, poste all'interno del sottosistema interessato.
Gli oggetti che lavorano insieme sono raggruppati, non sono distribuiti in tutta l'applicazione. Ciò aumenta anche la manutenibilità del codice.
Le classi e le strutture dati utilizzate saranno salvate in files nel formato *.py.*
I **templates** saranno posizionati nella cartella **templates**.

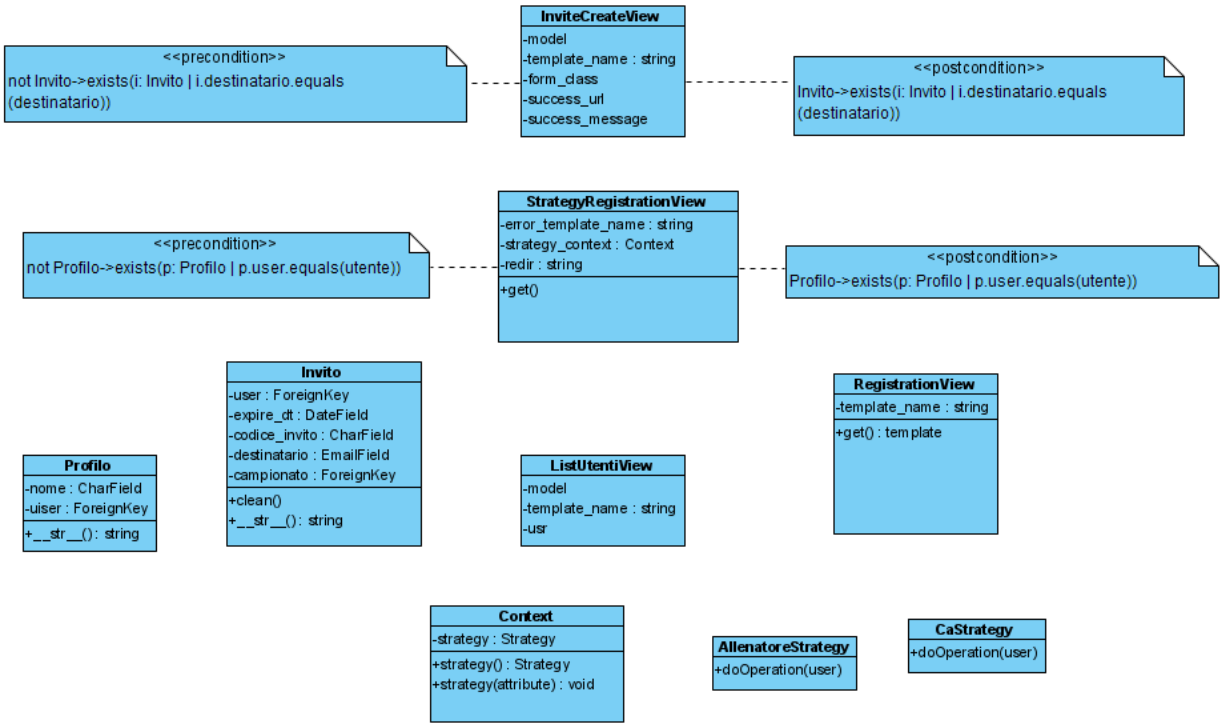
3. Class interfaces

Questa sezione contiene la specifica di ogni classe. Sono riportati, per ognuna di esse, i contratti rilevati, ovvero precondizioni, postcondizioni e invarianti.

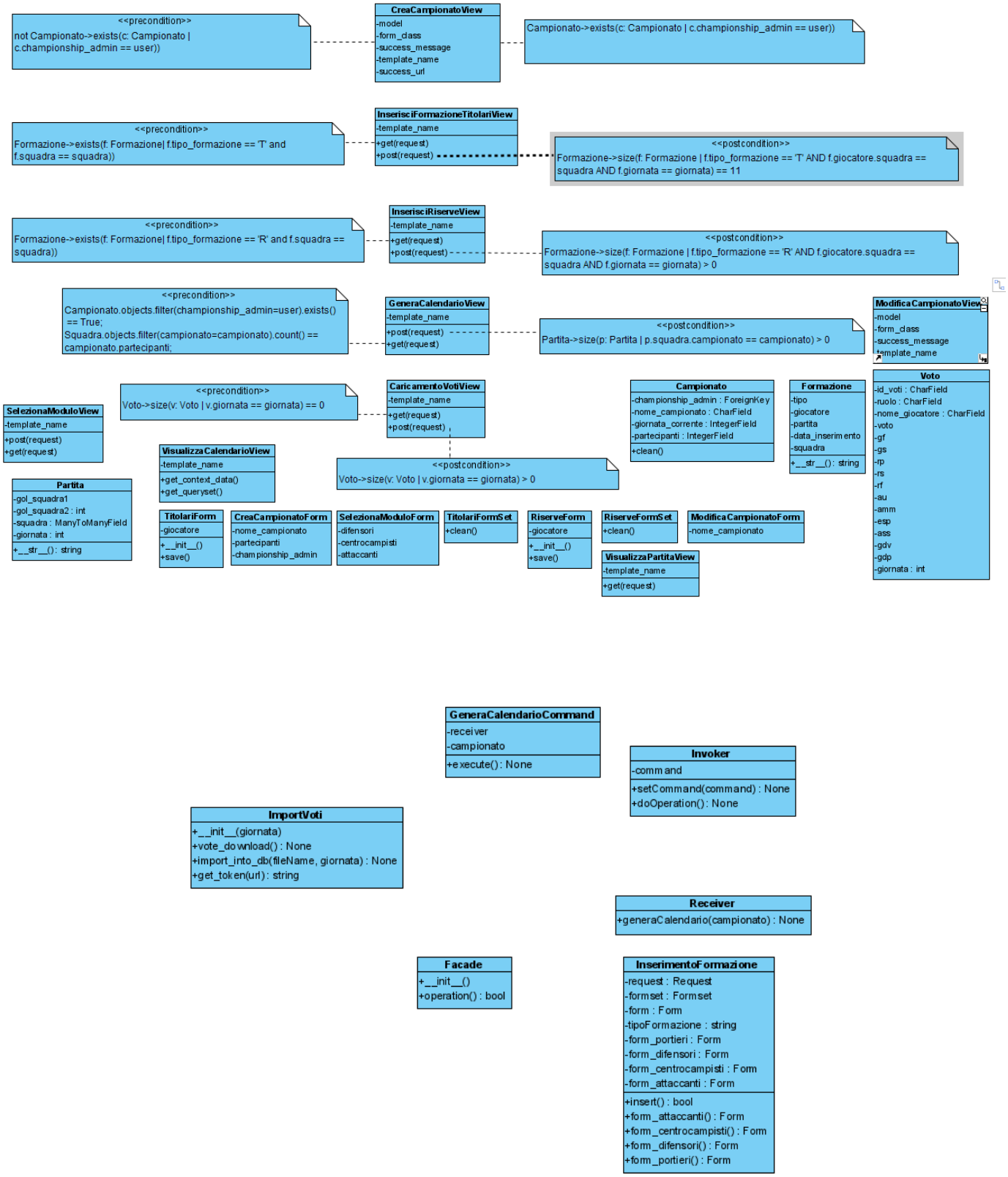
Documentazione Sphinx

La documentazione dettagliata del progetto è accessibile tramite l'indirizzo https://woofz.github.io/biaset/ .

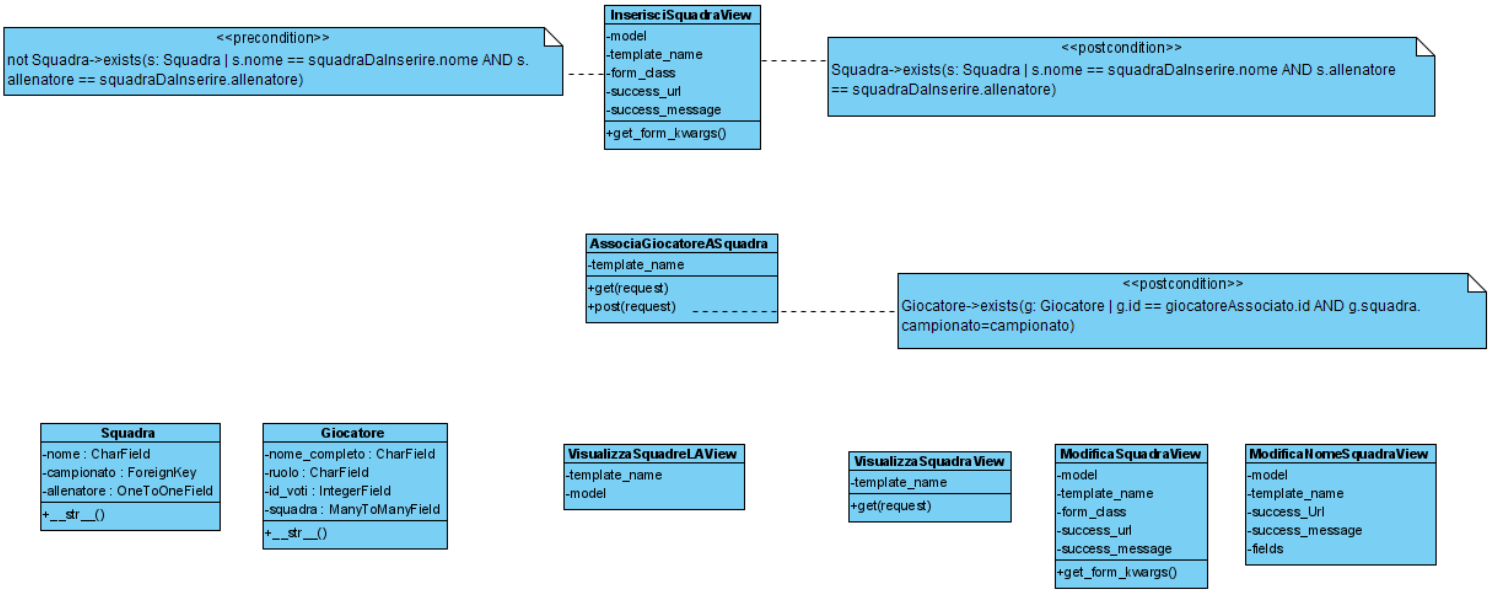
3.1 Gestione Utenza



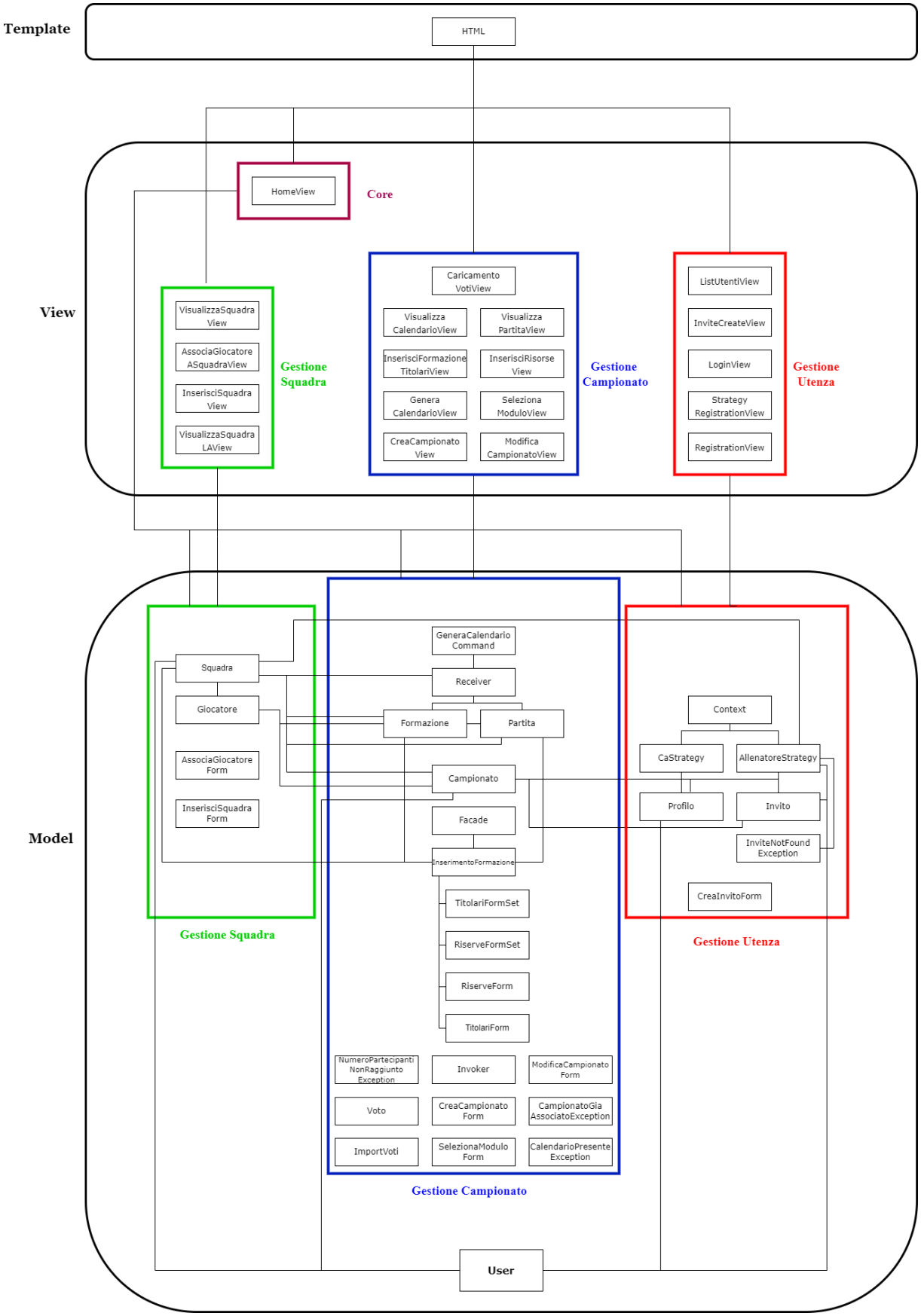
3.2 Gestione Campionato



3.3 Gestione Squadra



4. Class Diagram



5. Glossario

Object Design Trade-Offs: Decisioni sulla priorità da assegnare tra design goals che potrebbero andare in conflitto.

Class Interfaces: Descrive le classi e le loro interfacce pubbliche (overview di ogni classe, sue dipendenze con altre classi e package, i suoi attributi e operazioni pubblici, casi eccezionali)

Design pattern: descrivono object design parziali che risolvono questioni specifiche.

Database: Sistema di memorizzazione dati.

HTML: Linguaggio di programmazione utilizzato per lo sviluppo di pagine Web.

CSS: Linguaggio per la definizione degli stili delle pagine web

Framework: Software di supporto allo sviluppo.

Off-The-Shelf: Servizi esterni al sistema di cui viene fatto utilizzo.