Progetto di Programmazione e Modellazione ad Oggetti. Realizzazione del gioco Campo Minato

Studente: Rosario Zefiro
Matricola: 307593

Docente: Sara Montagna

Università degli Studi di Urbino Carlo Bo
Anno accademico 2024/2025

Indice

1 Analisi

Il progetto implementa Campo Minato, un gioco solitario con pochi fronzoli ma di grande successo tra gli utenti PC, in particolare di sistemi Windows su cui era presente a cavallo tra gli anni '90 e 2000, che non rinunciavano a una partita in momenti di svago o durante una pausa caffè. Con questo progetto si desidera rispolverare un vecchio classico dei videogiochi, con l'obiettivo di implementarne le funzionalità nella modalità di partita classica e di implementare funzionalità aggiuntive al gioco di base, riguardanti la gestione di più giocatori e la memorizzazione del risultato di partite passate, a fini di calcoli statistici.

Nella modalità di partita classica a Campo Minato, il giocatore interagisce con una griglia di caselle, chiamata campo, di M righe ed N colonne, dove in qualche casella della griglia è nascosta una mina. La partita inizia con ogni casella del campo coperta (in particolare, non è possibile per l'utente stabilire all'inizio della partita in quali celle le mine sono presenti) e lo scopo del gioco è scoprire ogni singola casella del campo ad eccezione delle sole caselle che contengono una mina. Se l'utente dovesse scoprire una casella in cui è presente una mina, la partita si considera persa.

Per facilitare lo svolgimento della partita, nel momento in cui viene scoperta una casella senza mina, se la casella ha una o più celle nel suo più stretto intorno (quindi nelle sue caselle adiacenti) contenenti una mina ne viene riportato il numero in corrispondenza della casella stessa. L'utente dovrà combinare le quantità di mine contigue a due o più celle vicine tra loro per scoprire quali celle contengono potenzialmente una mina e conseguentemente evitarle.

Per l'utente è possibile inoltre, a fini mnemonici, posizionare una bandierina su ogni casella sulla quale è considerata certa la presenza di una mina.

1.1 Requisiti

Requisiti funzionali

I requisiti funzionali descrivono le operazioni fondamentali che il sistema deve saper fare e le modalità di interazione con cui le operazioni devono essere messe a disposizione dell'utente. Di seguito vengono elencati i requisiti funzionali che sono stati definiti per il sistema in oggetto.

- Presentazione delle funzionalità e visualizzazione dei risultati mediante una interfaccia grafica e attivazione delle funzionalità mediante eventi scaturiti dall'interazione dell'utente con le componenti di una finestra (e.g. bottoni)
- Consentire all'utente di scegliere la difficoltà della partita, caratterizzata dalla dimensione del campo e dal numero di mine. La scelta verrà effettuata tra 3 diversi livelli di difficoltà chiamati facile, medio e difficile, e un'opzione aggiuntiva che consente all'utente di personalizzarla con valori inseriti manualmente.
- Creare una nuova partita secondo la difficoltà scelta, posizionando le mine casualmente all'interno del campo.

- Gestire una partita di Campo Minato, da un lato consentendo al giocatore di interagire con le caselle del campo per scoprirle o inserirci/toglierci la bandierina, e dall'altro garantire che venga rispettata ogni regola di gioco. La scopertura di una casella e il posizionamento/rimozione di una bandierina devono essere effettuati rispettivamente mediante un click col tasto sinistro e un click col tasto destro del mouse sulla casella di interesse.
- Mostrare all'utente lo stato della partita. In particolare le caselle scoperte e quelle ancora coperte, il numero di mine contigue ad una casella scoperta senza mina, le bandierine posizionate sul campo.
- Terminare la partita in caso di vittoria o di sconfitta. A partita terminata, il campo non può più essere sensibile a eventi da parte dell'utente.
- Salvare lo stato di una partita che durante la sua esecuzione viene interrotta dall'utente e ripristinarlo per riprenderla in un secondo momento
- Registrare un giocatore consentendogli di inserire le credenziali di accesso, nome e cognome, e dargli la possibilità di autenticarsi in un secondo momento.
- Mostrare lo storico delle partite effettuate da un giocatore.

L'applicazione deve rendere fruibili le diverse funzionalità in seguenti schermate:

Schermata	Descrizione
Registrazione giocatore	Consente all'utente di registrarsi, inserendo i propri dati
Login giocatore	Consente all'utente di autenticarsi, fornendo le credenziali di accesso
Setup partita	Consente all'utente di impostare i parametri di difficoltà della partita e di lanciarne una nuova con i parametri scelti.
Partita	Consente all'utente di giocare la partita, gestendone la logica.
Storico partite del giocatore	Consente all'utente di visualizzare lo storico delle partite che ha fatto in passato
Menu principale	Deve essere la schermata di avvio dell'applicazione e mostra all'utente una schermata di menu principale, in cui ogni opzione rimanda una diversa schermata tra le sopra riportate.

Requisiti non funzionali

I requisiti non funzionali descrivono gli attributi di qualità del sistema dal punto di vista dell'ottimizzazione dei processi di esecuzione, computazionale e di memoria, della progettazione architetturale e della protezione dei dati. Per il progetto in oggetto, sono stati individuati i seguenti requisiti non funzionali.

- Scalabilità del sistema. Il sistema deve essere facilmente manutenibile. In conseguenza a questa caratteristica, l'aggiunta di nuove funzionalità e la modifica di funzionalità esistenti apporterà interventi che coinvolgano il minor numero possibile di componenti che costituiscono l'architettura del sistema.
- Reattività del sistema. Il sistema deve rispondere con rapidità ad ogni input dell'utente. Questa caratteristica è ancora più critica nell'utilizzo dell'interfaccia grafica a causa della maggior richiesta computazionale che questa scelta comporta, e per essere ottenuta è necessario creare una architettura dei componenti che sia snella sfruttando i principali design pattern disponibili in letteratura, oltre a implementare algoritmi efficienti.

1.2 Modello di dominio

Il modello di dominio, come da nome, descrive il dominio del sistema, costituito dall'insieme di tutti i dati presi dalla realtà di riferimento che il sistema riconosce e che processa durante la propria esecuzione.

Lo scopo di questo modello è di individuare le funzionalità incapsulate in ogni dato e i legami che sussistono tra diversi dati. Queste informazioni costituiscono le interfacce dei dati stessi, cioè il modo in cui è possibile interagire con essi, astraendo i dettagli sul come le funzionalità siano implementate.

La figura 1.2.1. mostra il modello di dominio del progetto in esame. Partendo da Partita, essa fornisce ogni funzionalità di gestione di una partita, come il suo avvio e il suo arresto, e le attività di interazione col campo da gioco. Essa è costituita da un componente chiamato Campo che incapsula lo stato della partita in termini di posizionamenti delle mine e delle bandierine inserite nel campo, nonché delle caselle che sono state scoperte dal giocatore e fornisce le funzionalità necessarie a modificarlo, quali la scopertura o il toggle di una bandierina su una casella.

Poiché Campo è strettamente legato al concetto di partita, è stato scelto di rendere lo stesso inaccessibile in maniera diretta ad ogni componente al di fuori di Partita, lasciando a quest'ultima interfaccia il compito di mediare nella fruizione delle funzionalità di aggiornamento e di interrogazione su Campo.

Una casella del campo è rappresentata dall'interfaccia omonima Casella che descrive le caratteristiche di una casella generica, indipendentemente dal fatto che sia con o senza mina, fornendone le funzionalità di base. La presenza o meno di una mina in una casella è rappresentata da due sottotipi di Casella, chiamati rispettivamente CasellaConMina e CasellaSenzaMina.

Di una casella senza mina è necessario tenere traccia di quali caselle nel suo intorno siano con mina e quali siano senza, di conseguenza l'interfaccia CasellaSenzaMina è stata provvista di due associazioni: la prima collegata a se stessa per rappresentare l'insieme delle caselle adiacenti che non hanno mina e la seconda collegata a CasellaConMina per rappresentare quello delle caselle che ce l'hanno, nonché delle funzionalità necessarie a gestire i due insiemi.

Ogni casella è identificata all'interno del campo da una posizione, gestita dall'interfaccia omonima Posizione ed è il "mezzo" mediante il quale Campo memorizza il proprio stato, riceve le richieste di operazioni di modifica allo stesso e ne fornisce i risultati, giocando un ruolo fondamentale nella esecuzione della partita. Il modello prevede l'esistenza di due ulteriori interfacce di cui Partita possiede un riferimento, chiamate Giocatore e Report.

La prima contiene le informazioni associate al giocatore che ha creato la partita e le funzionalità necessarie e modificarle; la seconda contiene le informazioni inerenti alla partita a fini statistici, come la difficoltà, la durata e l'esito della partita, comprensivo della percentuale di mine trovate. L'analisi del modello si conclude con l'interfaccia Storico, che gestisce i report di partite passate concluse da un giocatore. Tale dato tiene traccia quindi del singolo Giocatore di cui rappresenta lo storico, e dell'insieme di tutti i report che costituiscono lo storico.

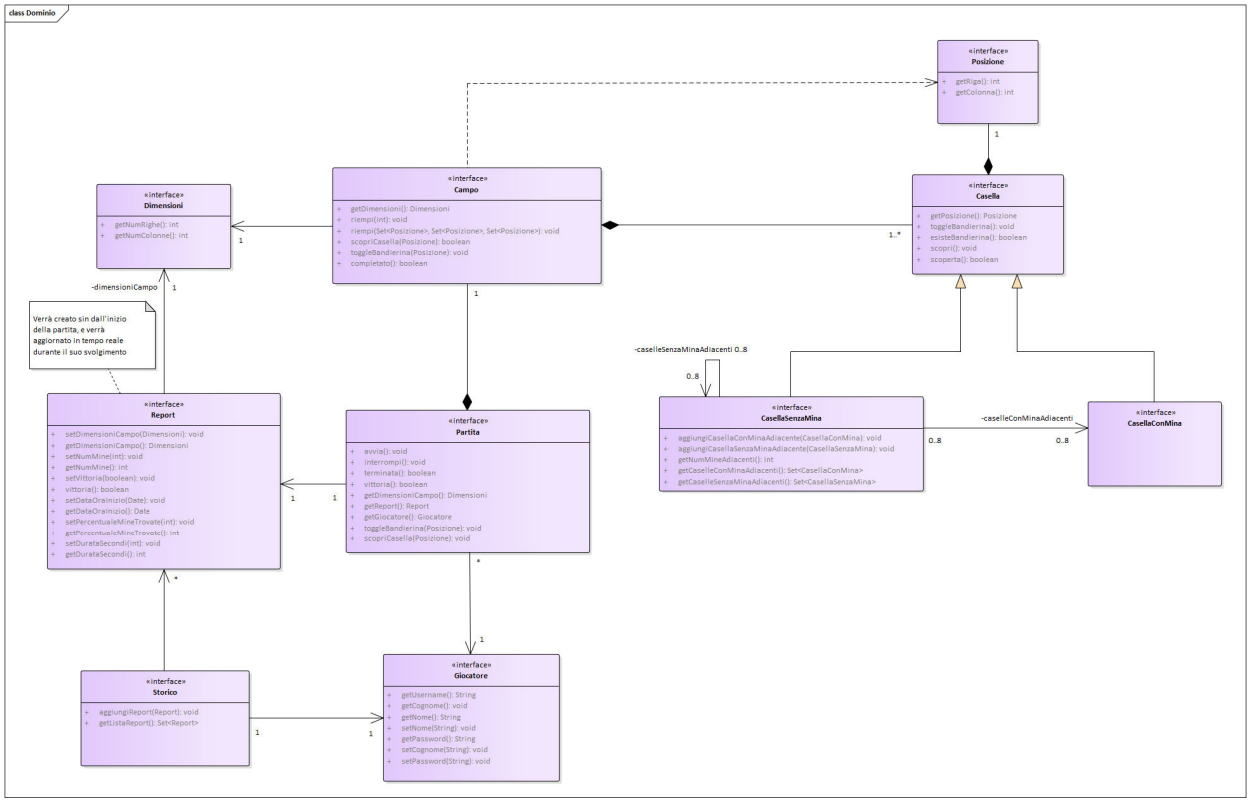

Figura 1.2.1: modello di dominio

2 Design

2.1 Architettura

Il sistema è stato implementato utilizzando la specifica del design pattern architetturale MVC, che vede la logica dell'applicazione divisa in tre componenti distinti:

- Model: si occupa della gestione dei dati, implementando la logica di business del dominio applicativo. E' il responsabile nell'esecuzione delle operazioni di interrogazione e di aggiornamento dei dati;

- View: si occupa della logica di presentazione interagendo con l'utente. Definisce le regole con cui i dati provenienti dal Model devono essere mostrati e le regole con cui le funzionalità devono essere rese disponibili all'utente.

- Controller: fa da collante tra la View e il Model. Intercetta le interazioni dell'utente con la View. Ad ogni interazione intercettata esegue una specifica operazione di interrogazione o aggiornamento del Model, per poi aggiornare conseguentemente la View con il risultato ottenuto dall'operazione stessa.

Il pattern MVC garantisce una soluzione architetturale robusta grazie alla separazione delle responsabilità dei compiti di presentazione e quelle di gestione dei dati in componenti diversi, semplificando di conseguenza l'attività di implementazione del sistema e di modifica del codice, potendo agire sui singoli componenti in modo indipendente dagli altri.

Per la progettazione del sistema in oggetto si è scelto di combinare la robustezza del pattern MVC a una implementazione modulare dei due componenti View e Controller, al fine di rendere l'applicazione facilmente adattabile in caso di aggiunta di funzionalità, e la conseguente aggiunta di schermate mediante le quali presentarle all'utente.

La figura 2.1.1 mostra l'architettura dei componenti del sistema, in cui sono state definite due interfacce, chiamate ModularView e ModularController, che estendono le interfacce di base dei componenti View e Controller e che si appoggiano ai rispettivi moduli chiamati SubView e SubController per gestire gli eventi di interazione provenienti da diverse schermate dell'applicazione.

Ogni singola schermata, quindi, sarà gestita nella sua logica di presentazione da una specifica SubView, la quale comunicherà con un SubController dedicato per attivarne le funzionalità. Tali due componenti forniscono una interfaccia basilare dei due rispettivi moduli, che viene poi estesa da tipi più specifici, ognuno progettato su misura per ogni diversa schermata, al fine di soddisfarne i requisiti. Sono state progettate in totale sei diverse sotto-interfacce di SubView e di relativi SubController per altrettante schermate.

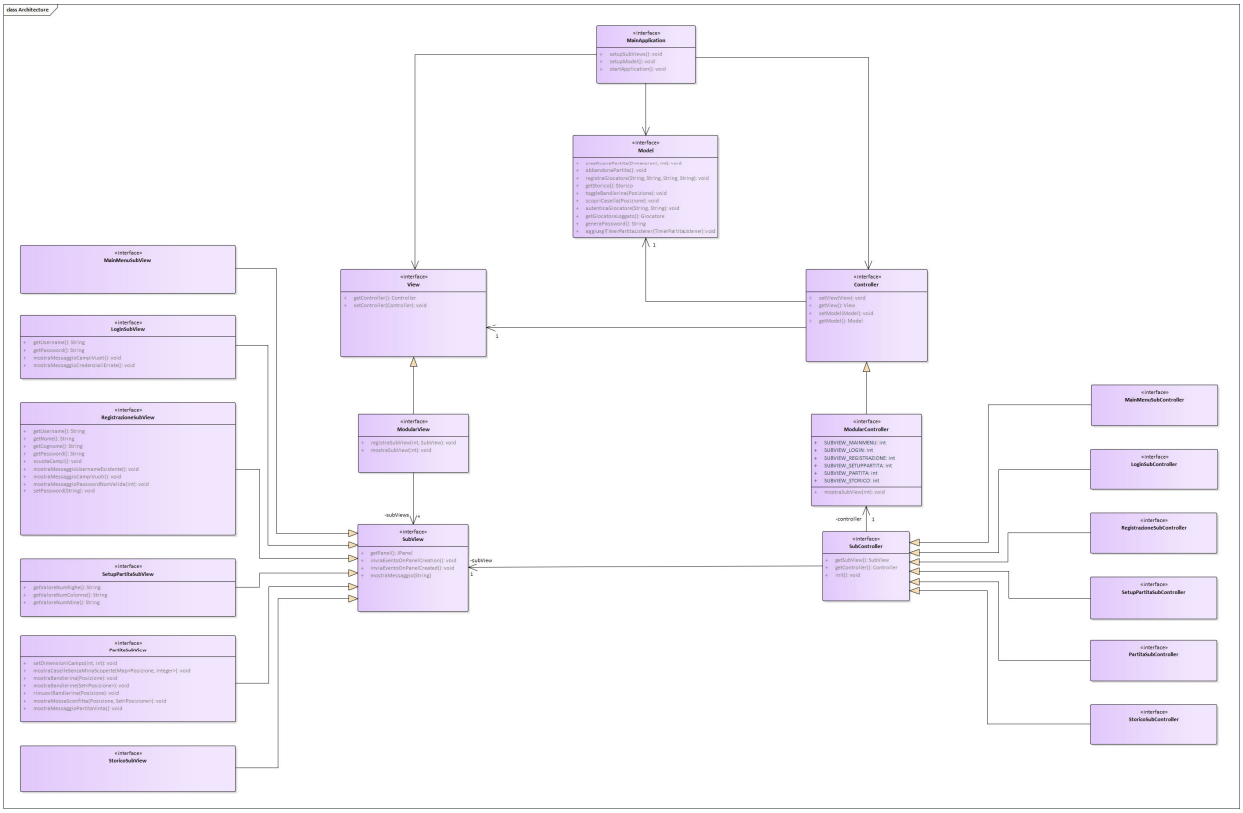

Figura 2.1.1: Architettura

La tabella 2.1.1 riporta, per ogni schermata del sistema, i tipi specifici di SubView e SubController associati.

Schermata	SubView	SubController
Menu principale	MainMenuSubView	MainMenuSubController
Login giocatore	LoginSubView	LoginSubController
Registrazione giocatore	RegistrazioneSubView	RegistrazioneSubController
Scelta difficoltà partita	SetupPartitaSubView	SetupPartitaSubController
Partita	PartitaSubView	PartitaSubController
Storico partite giocatore	StoricoSubView	StoricoSubController

2.2 Comunicazione tra View e Controller

Per consentire alla View, mediante le sue SubView, di notificare il Controller di eventi generati durante la propria esecuzione, è stato utilizzato il pattern Observer, illustrato in figura 2.2.1. Nella soluzione proposta, ogni SubController "osserva" gli eventi generati dalla SubView a cui è associato, al fine di attuare una operazione in risposta a ciascuno di essi.

Gli eventi generati da una SubView sono di due tipi:

- Eventi di pre e post creazione della schermata: utili per eseguire operazioni prima e dopo la creazione della schermata. Tra le operazioni necessarie da attuare in queste casistiche è presente il reperimento preliminare dal Model dei dati che si necessita mostrare nella schermata sin dalla sua creazione.
- Eventi di interazione con l'utente: utili a reagire ad ogni interazione attivando una diversa operazione di modifica o interrogazione del Model

Per osservare gli eventi della prima categoria, l'observer dovrà possedere una implementazione dell'interfaccia SubViewListener, e per osservare gli eventi della seconda categoria, dovrà possedere una implementazione di una sotto-interfaccia di SubViewListener, il cui tipo è specifico per ogni diverso tipo di SubView, al fine di circoscrivere le tipologie di eventi generabili da tipi diversi di SubView. Ogni diverso SubController, quindi fornirà alla SubView associata l'implementazione del sotto-tipo di SubViewListener compatibile con la stessa.

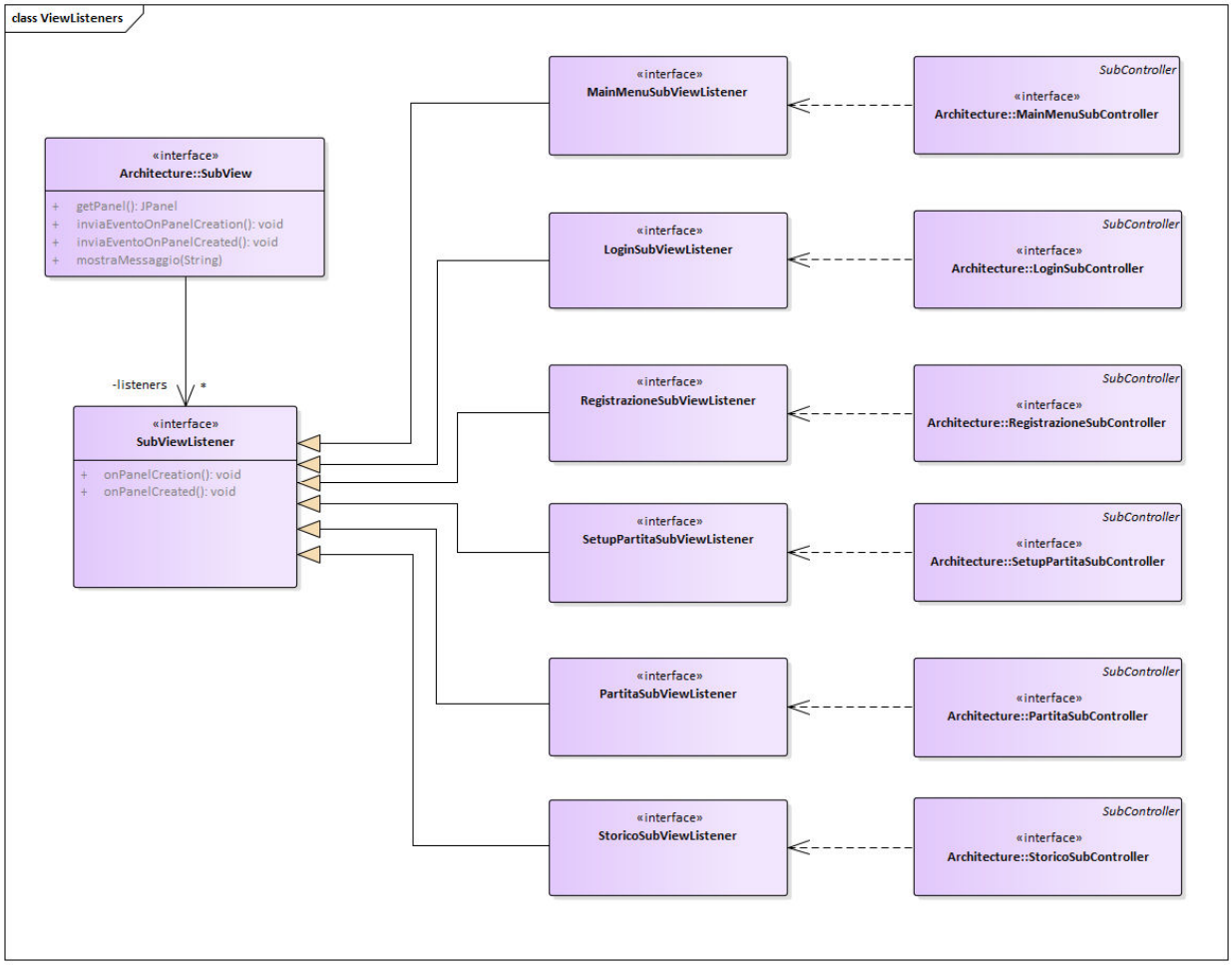

Figura 2.1.1: Comunicazione tra View e Controller

2.3. Comunicazione tra Controller e Model

Si è utilizzato il design pattern Observer anche per far comunicare il Controller, tramite i propri SubController, con il Model, in particolare per fornire aggiornamenti dello stato del Model che sono conseguenti a operazioni di modifica dello stesso.

La figura 2.3.1 illustra la soluzione proposta, per la quale gli eventi di aggiornamento generati dal Model sono stati divisi in tre tipologie, ciascuna delle quali associata all'implementazione di un observer specifico, come mostrato nella tabella 2.3.1.

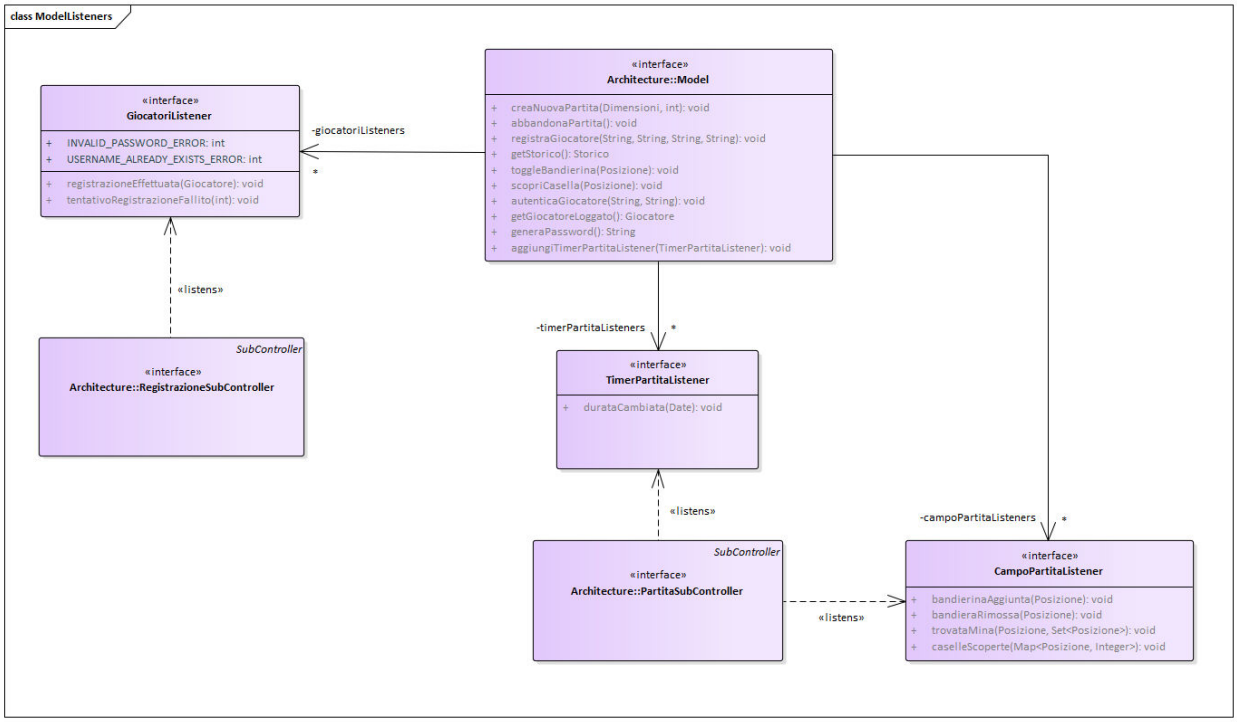

Figura 2.3.1: Comunicazione tra Controller e Model

Observer	Tipologia di eventi
GiocatoriListener	Eventi generati dall'operazione di registrazione di un nuovo giocatore
TimerPartitaListener	Eventi generati dall'operazione di aggiornamento del timer della partita in corso
CampoPartitaListener	Eventi generati da operazioni effettuate sul campo della partita in corso

3. Implementazione

3.1. Implementazione di ModularView

La figura 3.1.1 mostra l'implementazione di ModularView. Ogni tipo derivato di SubView è stato implementato da una classe specifica che estende una implementazione di base di SubView fornita dalla classe astratta SubViewBaseClass.
Quest'ultima è stata resa astratta allo scopo di rendere impossibile l'utilizzo di istanze create direttamente da SubView, e al contempo di definire il comportamento di base della generica SubView, delegando il comportamento specifico di ogni tipo alle rispettive classi limitato al metodo getPanel().

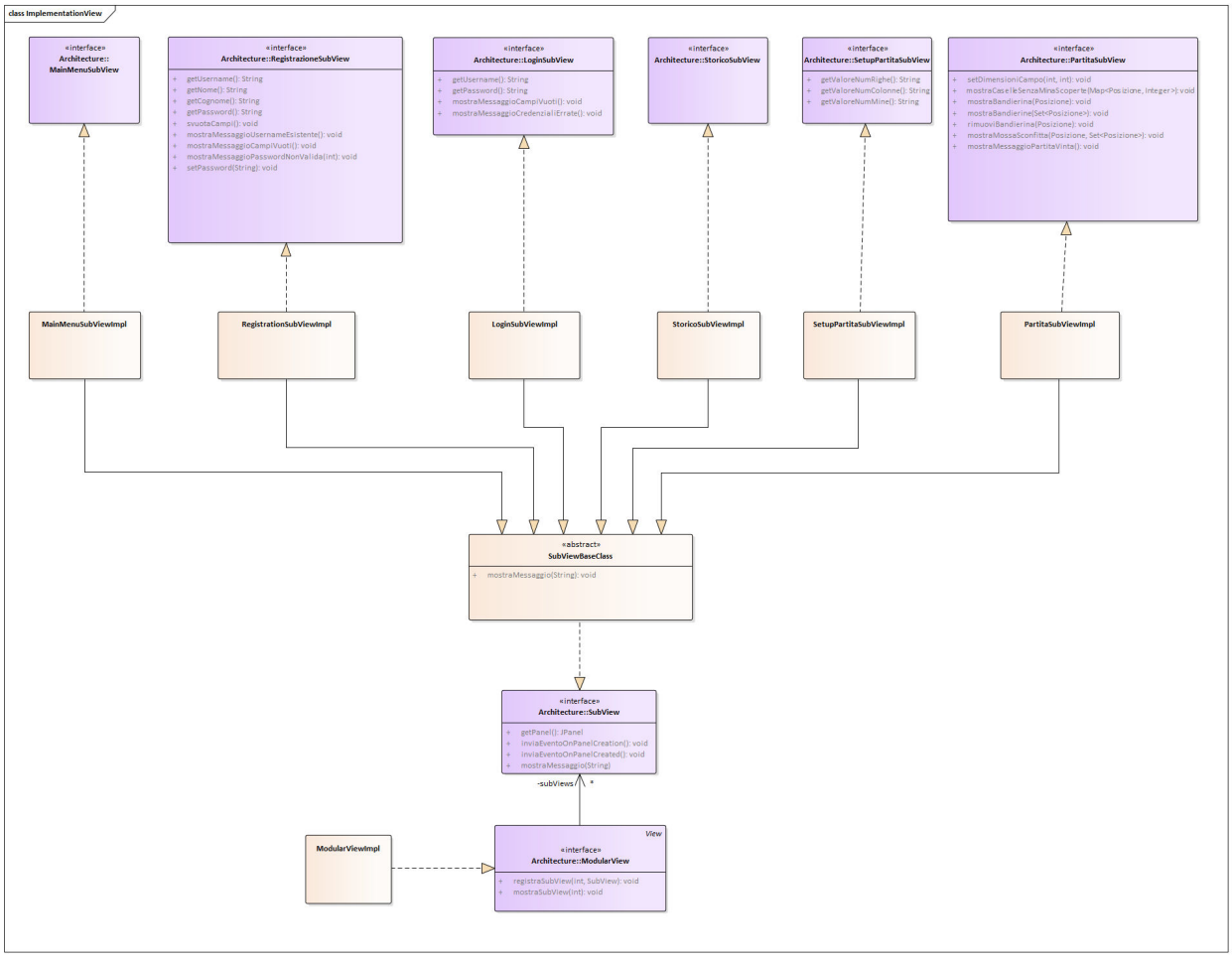

Figura 3.1.1: Implementazione di ModularView

3.2. Implementazione del ModularController

La figura 3.2.1. mostra l'implementazione del ModularController.
Conseguentemente alla simmetria che sussiste tra SubView e SubController, l'implementazione del ModularController proposta è stata realizzata secondo gli stessi principi attuati nell'implementare la ModularView e descritti nel capitolo 3.1.

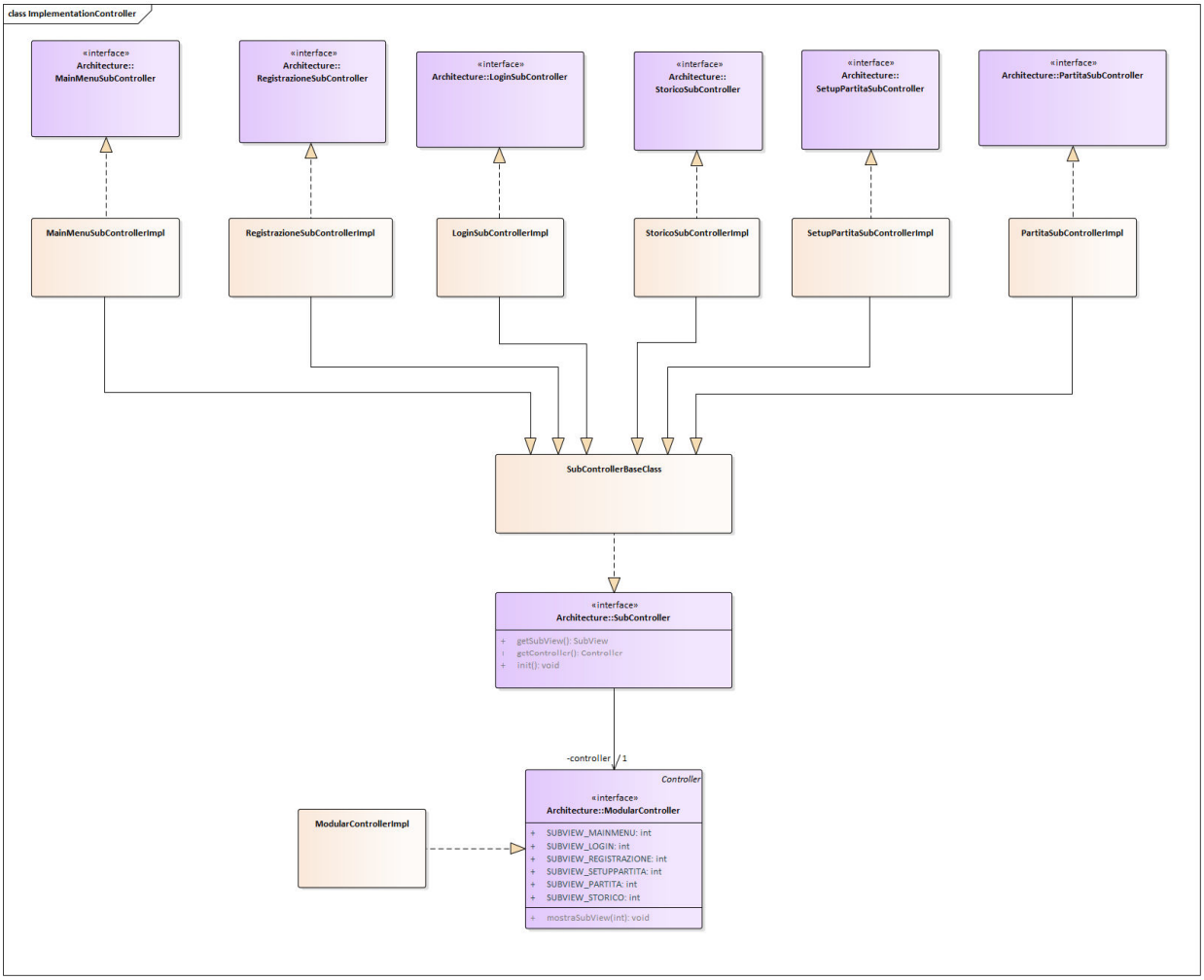

Figura 3.2.1: Implementation Modular Controller

3.3. Implementazione del Model

La figura 3.3.1 illustra l'implementazione del Model.

La soluzione individuata prevede la realizzazione di diversi gestori, adibiti a implementare funzionalità di tipo diverso e sui quali il Model si affida nella esecuzione delle diverse operazioni richieste dal Controller. La tabella 3.3.1 elenca i gestori individuati e i rispettivi compiti gestiti dagli stessi.

Gestore	Funzionalità
GestorePartite	Funzionalità inerenti alla gestione delle partite
GestoreGiocatori	Funzionalità inerenti alla gestione dei giocatori
GestoreDataset	Funzionalità inerenti alla persistenza dei dati

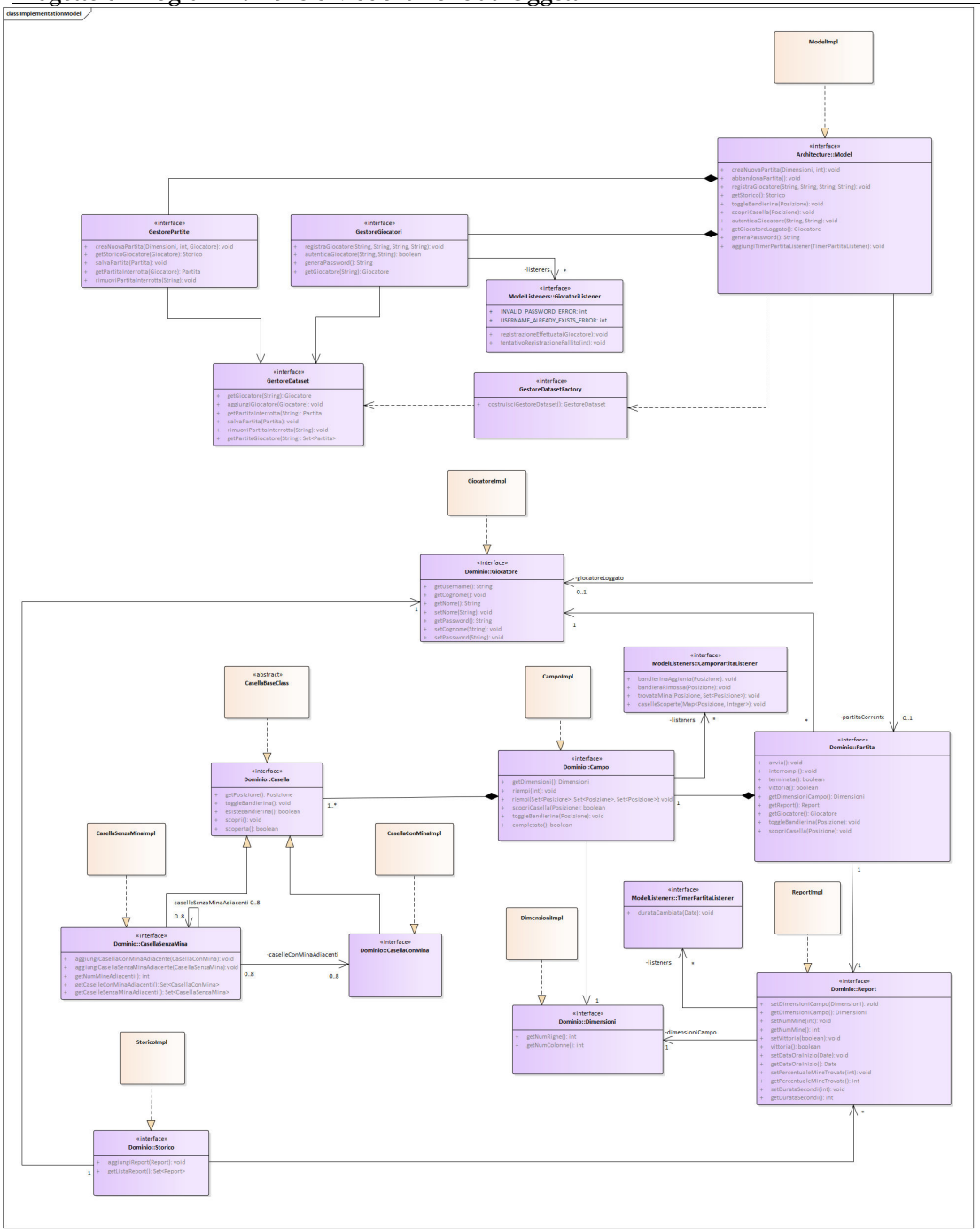

Figura 3.3.1: Implementazione del model

3.4. Implementazione dei gestori

La figura 3.4.1 illustra l'implementazione dei gestori introdotti nel capitolo 3.3.

La soluzione individuata fa uso del Pattern Builder implementato dalle classi GiocatoreBuilder, PartitaBuilder, ReportBuilder e CampoBuilder per la creazione di una partita con i tutti i relativi componenti da cui è costituita e di un giocatore, al fine di rendere più semplice la creazione di dati complessi del Model e di rendere la creazione più flessibile rispetto a eventuali attributi aggiuntivi che verranno eventualmente introdotti in successive iterazioni implementative del sistema. Il pattern Builder è stato utilizzato in combinazione con il pattern Factory allo scopo di creare, in sostituzione al classico costruttore vuoto, builder che contengano sin da subito le informazioni essenziali necessarie a istanziare il dato di interesse. L'utilizzo del pattern Factory impedisce che vengano utilizzati builder con le informazioni essenziali mancanti, rendendo inaccessibile ogni costruttore del builder.

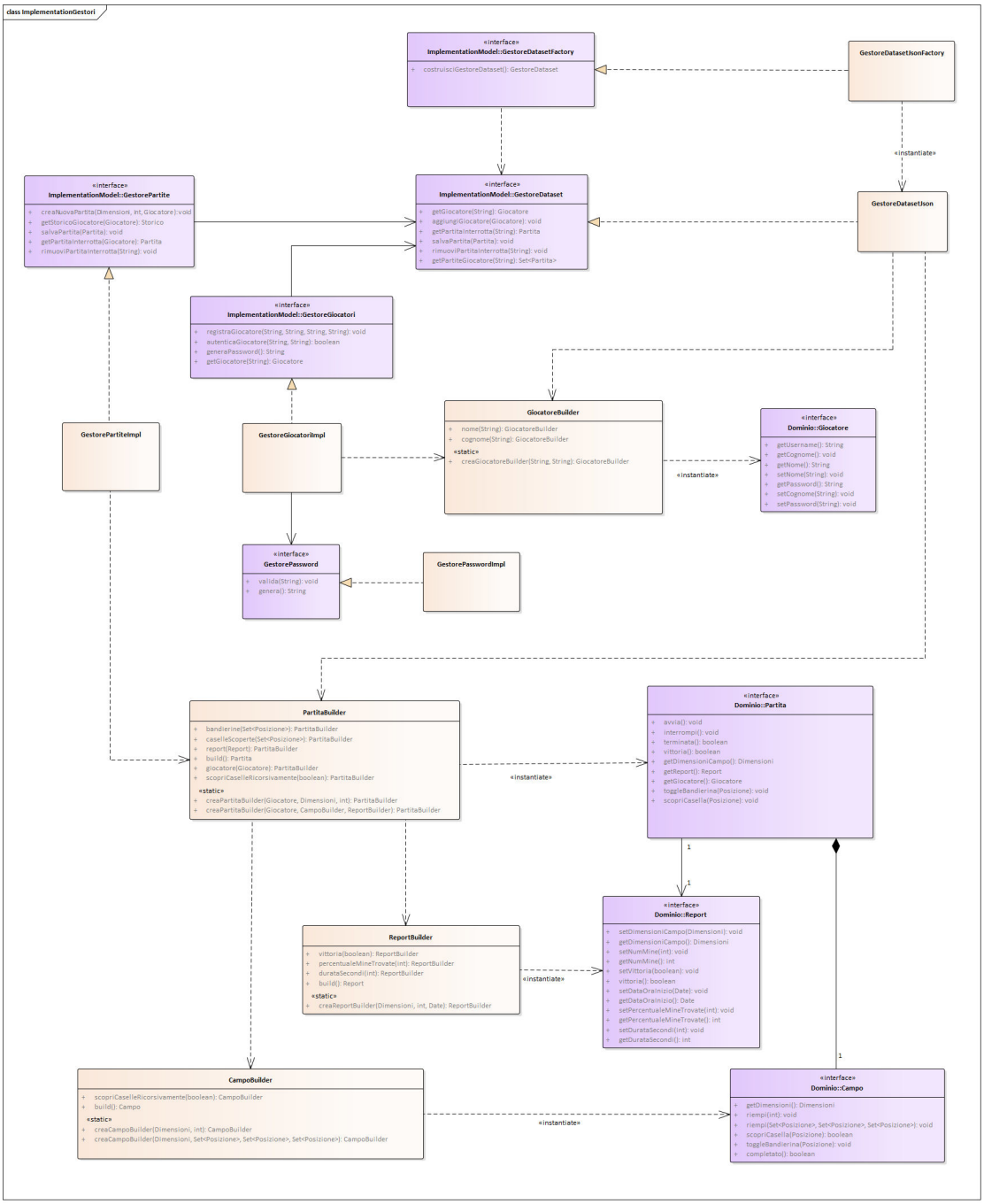

Figura 3.4.1: Implementazione dei gestori

3.5. Implementazione del TimerPartita

La figura 3.5.1 illustra l'implementazione del TimerPartita

La soluzione individuata fa uso della classe TimerTask fornita dalla libreria standard Java e che è stata estesa dalla classe TimerPartitaTask che implementa concretamente il timer della partita.

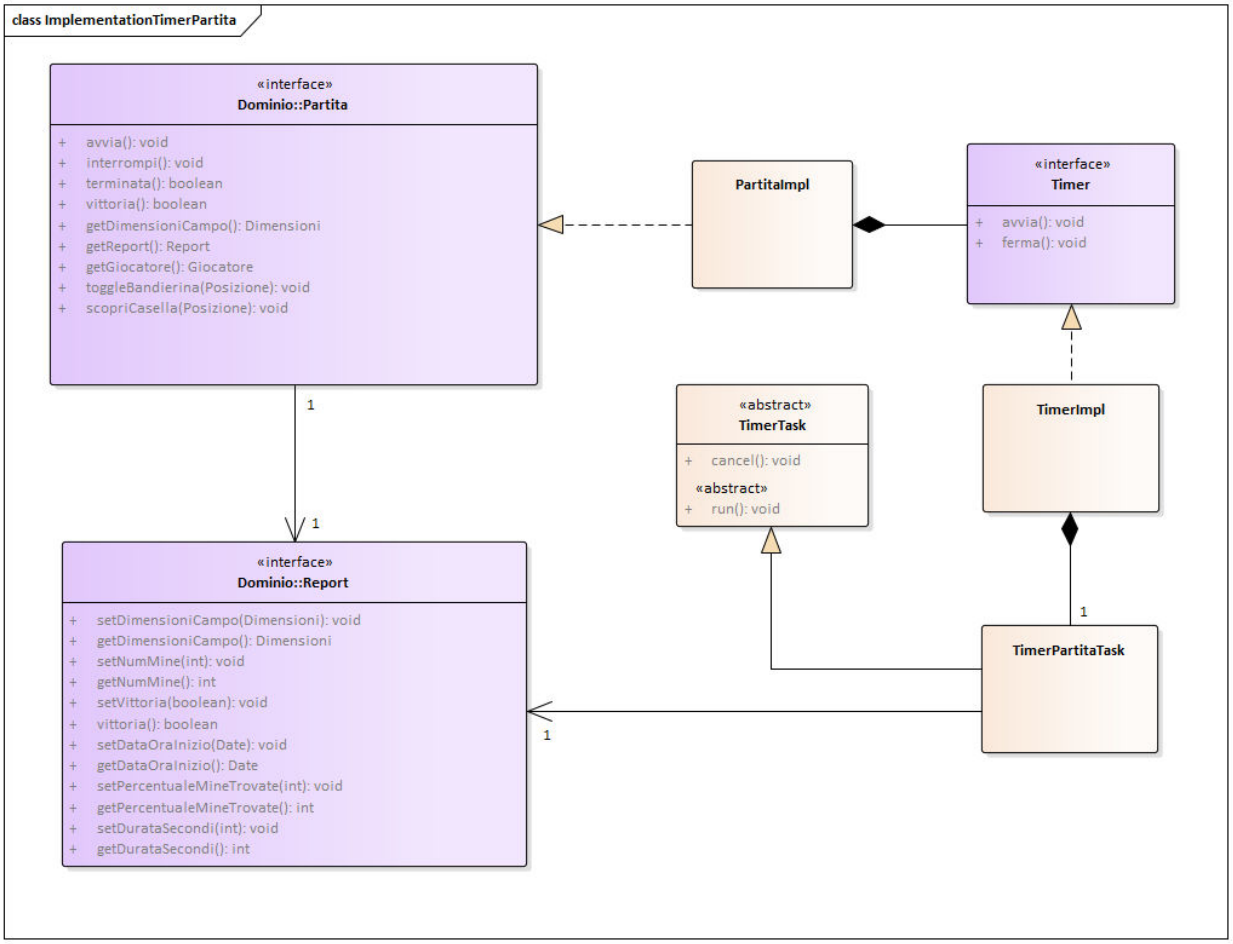

Figura 3.5.1: Implementazione del Timer Partita

4. Lavori Futuri e Funzionalità non Implementate

Nel corso dello sviluppo del progetto non è stato possibile completare alcune delle funzionalità previste in fase di analisi dei requisiti. In particolare:

- **Timer della partita**
  Era prevista l'implementazione di un sistema di cronometraggio che misurasse la durata della partita in tempo reale, con la possibilità di metterlo in pausa e riprenderlo. Sebbene sia stato definito il design e predisposta la struttura della classe *TimerPartita*, non si è giunti a una versione pienamente funzionante e integrata con il resto del sistema.

- **Gestione della persistenza dei dati riguardanti i giocatori e le partite giocate**
  Non è stata implementata la gestione completa dei giocatori e del loro storico di partite. Questo include:
  - registrazione e autenticazione dei giocatori tramite credenziali (username e password);
  - salvataggio permanente dei dati relativi agli utenti;
  - memorizzazione dei risultati delle partite (data, durata, esito, statistiche);
  - consultazione dello storico delle partite da parte del singolo giocatore.

Queste funzionalità costituiscono una naturale evoluzione del progetto e rappresentano sviluppi futuri utili per rendere l'applicazione più completa e aderente ai requisiti iniziali.

5. Conclusioni

La realizzazione di questo progetto mi ha permesso di mettere in pratica e consolidare i concetti appresi durante il corso di Programmazione e Modellazione ad Oggetti. Nonostante alcune funzionalità non siano state completate, il percorso di sviluppo mi ha dato l'occasione di confrontarmi con problematiche reali di progettazione e implementazione, stimolando la mia capacità di ragionamento critico e di problem solving.

Il corso, oltre a fornirmi gli strumenti tecnici, è stato per me un'esperienza formativa importante: mi ha aiutato a crescere non solo come studente, ma anche come futuro sviluppatore, facendomi capire l'importanza di un buon design software e dell'organizzazione modulare del codice.

Avrei voluto fare di più e meglio, ma il tempo che riesco a dedicare all'università non è molto, dovendo bilanciare studio, lavoro e famiglia. Inoltre, non essendo studente frequentante a causa degli impegni lavorativi, ho incontrato ulteriori difficoltà nel seguire il percorso con continuità. Nonostante ciò, ho sempre cercato di aggiornarmi e colmare le lacune utilizzando strumenti personali, come manuali, libri di programmazione e materiali di supporto.

In conclusione, posso dire che questo progetto ha rappresentato una sfida entusiasmante e un'occasione per riscoprire il piacere della programmazione, accompagnato da un corso che ho trovato stimolante e ben strutturato. Ringrazio la docente per la disponibilità e la passione trasmessa, che hanno reso questo percorso ancora più significativo e motivante.

Link al repository

https://github.com/rzefiro/campominato.git