Software Architecture Design

Game Repository - Relazione finale

Conte Valerio D. - M63001606
Di Giacomo Nike - M63001641
Falino Alessandro - M63001658

Glossario

- **game (**oppure **gioco, partita)** è l'entità principale del sistema. Un game è composto da un round, caratterizzato poi da più turni.
- **player (**oppure **giocatore)** è l'entità che può giocare i turni nel round di un game.
- **round** è l'entità che contiene le informazioni di gioco principali. Ogni game è composto da un numero finito di round, impostato nel nostro caso ad un valore di default pari a 1 durante il quale ciascun giocatore effettua la propria giocata.
- **turn (**oppure **turno)** è l'unità indivisibile di un round e rappresenta le azioni di un giocatore durante il round. Quindi contiene informazioni sull'esito dell'esecuzione dei test ed i relativi risultati nonché le classi generate per risolvere il problema.

Indice

1 Introduzione

Questo elaborato descrive le metodologie, le fasi di sviluppo e gli artefatti prodotti nel corso Software Architecture Design dell'anno 2023/24 per la realizzazione di un task specifico all'interno del progetto ENACTEST. In particolare, il progetto ha l'obiettivo principale di creare un gioco con lo scopo di promuovere le attività di *testing* e che permetta ai partecipanti di cimentarsi in sfide contro sistemi automatizzatie, eventualmente, altri giocatori.

1.1 Caso di studio

Il caso oggetto di studio durante il corso prevede l'estensione dello scenario di gioco implementato precedentemente. In generale, il Gaming Test consiste nel fatto che un giocatore possa effettuare una partita contro un singolo avversario robotico testando una singola classe. In particolare, all'atto della creazione della partita, il giocatore sceglie quale *tool* sfidare. In fase preliminare, gli strumenti automatici scelti sono **EvoSuite** e **Randoop**.

1.1.1 Task assegnato

Il task assegnato prevede la progettazione e lo sviluppo dei requisiti relativi al mantenimento delle partitegiocate. Di seguito è riportata la formulazione del task:

"Verificare e gestire la coerenza dei dati della partita del giocatore attualmente salvati nel File System di T8 con quelli mantenuti nel database del gioco in T4. Eventualmente rivedere le API offerte da T4 per consentire di salvare anche i dati relativi ai Turni di gioco in T4."

1.2 Metodologie adottate

In questa sezione, sono descritti gli approcci utilizzati durante lo svolgimento del task. Il lavoro dei teams è stato suddiviso in **3 iterazioni**, della durata di circa due settimane, al termine delle quali ogni gruppo ha presentato, in sessione plenaria, le attività svolte ottenendo dei *feedback*.
Per adeguarsi al ritmo iterativo di sviluppo sono state svolte le seguenti attività:

- Allineamento all'inizio dell'iterazione, per la definizione degli obiettivi e dei task;
- Allineamento alla fine dell'iterazione, per valutare i feedback ricevuti;
- Aggiornamento dei task e della loro assegnazione sul progetto Github;

Figura 1: Backlog e scheduling delle attività

2 Analisi dell'architettura di partenza

2.1 Diagramma dei casi d'uso, Requisiti Funzionali e Requisiti non Funzionali

Figura 2: Use Case Diagram

In questa sezione, è descritta l'architettura di partenza del software, che abbiamo provveduto a modificare. Per quanto riguarda il diagramma dei casi d'uso, è caratterizzato da un unico attore: il **Game Engine**, che rappresenta l'utente utilizzatore del servizio che si occupa di implementare le logiche di gioco. Questo utente garantisce che tutti i dati salvati all'interno del **Game Repository** siano coerenti. I **casi d'uso** individuati sono i seguenti:

- **Gestione Partite**: Il sistema gestisce partite con turni e test, registrando i risultati nel Game Repository. I requisiti principali includono la creazione, cancellazione e modifica delle partite, oltre alla possibilità di cercare e recuperare informazioni sulle partite, eventualmente filtrando i risultati. I **requisiti funzionali** che vengono evidenziati sono i seguenti: Creazione partita, Cancellazione Partita, Modifica Partita, Ricerca Partita, Recupero partite.
- **Gestione Round**: Il sistema gestisce round in cui i giocatori generano classi per risolvere un problema. Il punteggio viene salvato nel turno. Ogni round ha un vincitore. Requisiti chiave includono la creazione, cancellazione e modifica dei round, oltre alla possibilità di cercare e recuperare informazioni sui round di una partita. I **requisiti funzionali** che vengono

evidenziati sono i seguenti: Creazione round, Modifica Round, Ricerca Round, Recupero Round.

- **Gestione Turni**: Il sistema gestisce i turni in cui i giocatori generano classi di test eseguite dal sistema di testing, che assegna un punteggio. Il Game Repository salva i file prodotti e i punteggi. Gli utenti possono cancellare o modificare i turni dei giocatori. I **requisiti funzionali** implementati sono: Creazione Turno, Cancellazione Turno, Modifica Turno, Ricerca Turno, Recupero turni.

- **Gestione risultati Robot**: Il sistema gestisce i risultati dei robot, Randoop ed EVOsuite, per ogni classe di test o livello di una partita. I punteggi dei robot sono memorizzati durante la creazione del livello e recuperati successivamente. Il Game Repository deve salvare i punteggi, distinguendo tra i due tipi di strumenti automatici e associando a ciascun risultato un livello di difficoltà. I **requisiti funzionali** generati sono: Creazione Risultati Robot, Cancellazione risultati Robot, Ricerca Risultati Robot.

Per quanto riguarda i **requisiti non funzionali** invece, ne sono stati identificati cinque:

- **Gestione dei file dei giocatori**: Durante il turno, i file generati dai giocatori devono essere conservati per garantire prestazioni accettabili del database e consentire operazioni di lettura e aggiornamento.

- **Osservabilità:** Il sistema deve fornire contatori e metriche per analizzare le performance e condurre attività di ottimizzazione.

- **Compliance:** Il sistema deve esporre un'interfaccia che implementa lo standard Open API, poiché sarà utilizzato da altri servizi.

- **Paginazione:** Nei casi in cui vengono recuperate più partite, round o turni, devono essere implementati meccanismi di paginazione per limitare la quantità di dati letti e trasmessi.

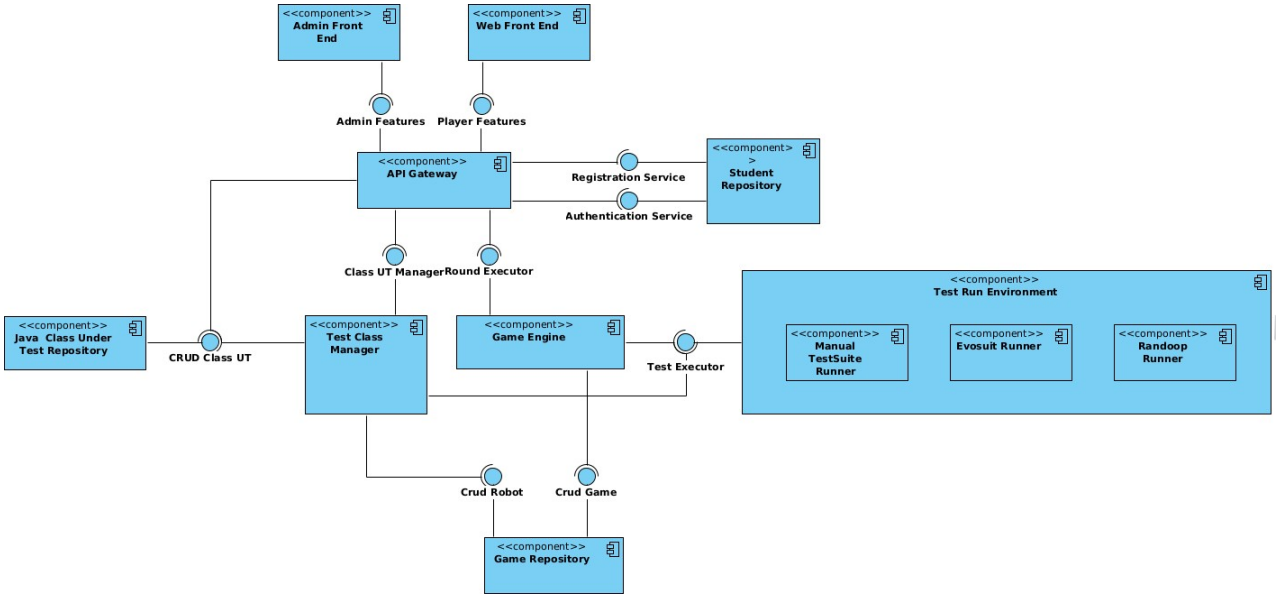
- **Sicurezza:** Ogni richiesta al sistema deve essere autenticata, con la gestione dell'autenticazione affidata a un altro servizio nel contesto specifico di questo progetto.

2.2 Architettura Generale

Fissati i requisiti, si è scelto di analizzare il problema con un approccio ispirato a **Domain Driven Design** [1] in cui si è cercato di progettare le funzionalità dal punto di vista delle entità del sistema. Questa strategia mira a dividere l'architettura iniziale in componenti indipendenti.

1**: Il Domain-Driven Design** (DDD) è un insieme di principi e modelli che aiutano gli sviluppatori a creare eleganti sistemi ad oggetti. Se applicato correttamente, può portare ad astrazioni di software chiamate domain models (modelli di dominio). Questi modelli incapsulano la logica aziendale complessa, colmando il divario tra la realtà aziendale e codice.

È possibile dunque analizzare il seguente **Context Diagram**:

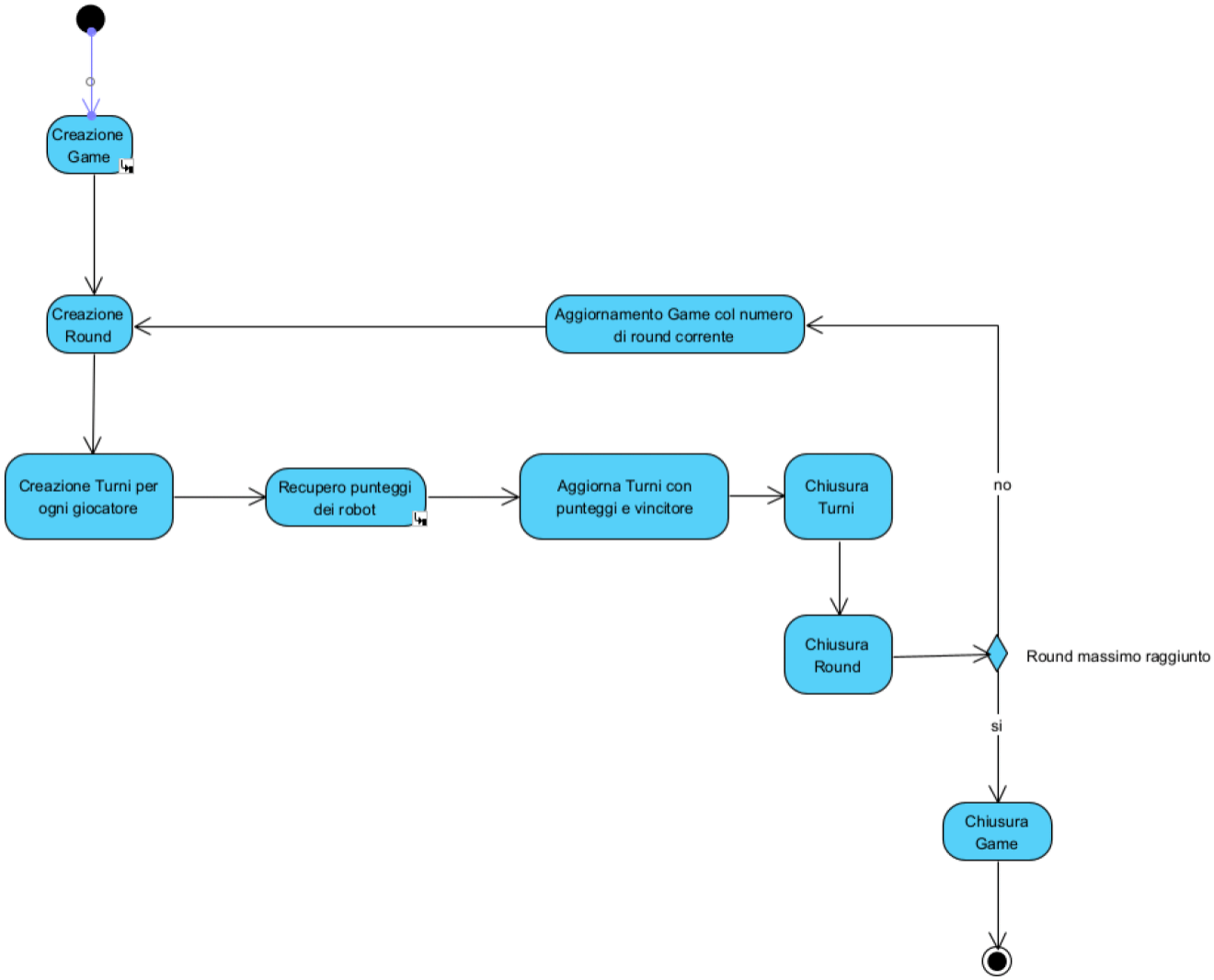


In tale diagramma è possibile individuare diversi componenti, ognuno dei quali ha un uso ben definito. L'architettura mostrata è di tipo *frontend - backend*. In particolare, sono stati previsti due tipi di *front-end* (uno per i giocatori e un altro per l'amministratore). Entrambi, si interfacciano con il componente API gateway.

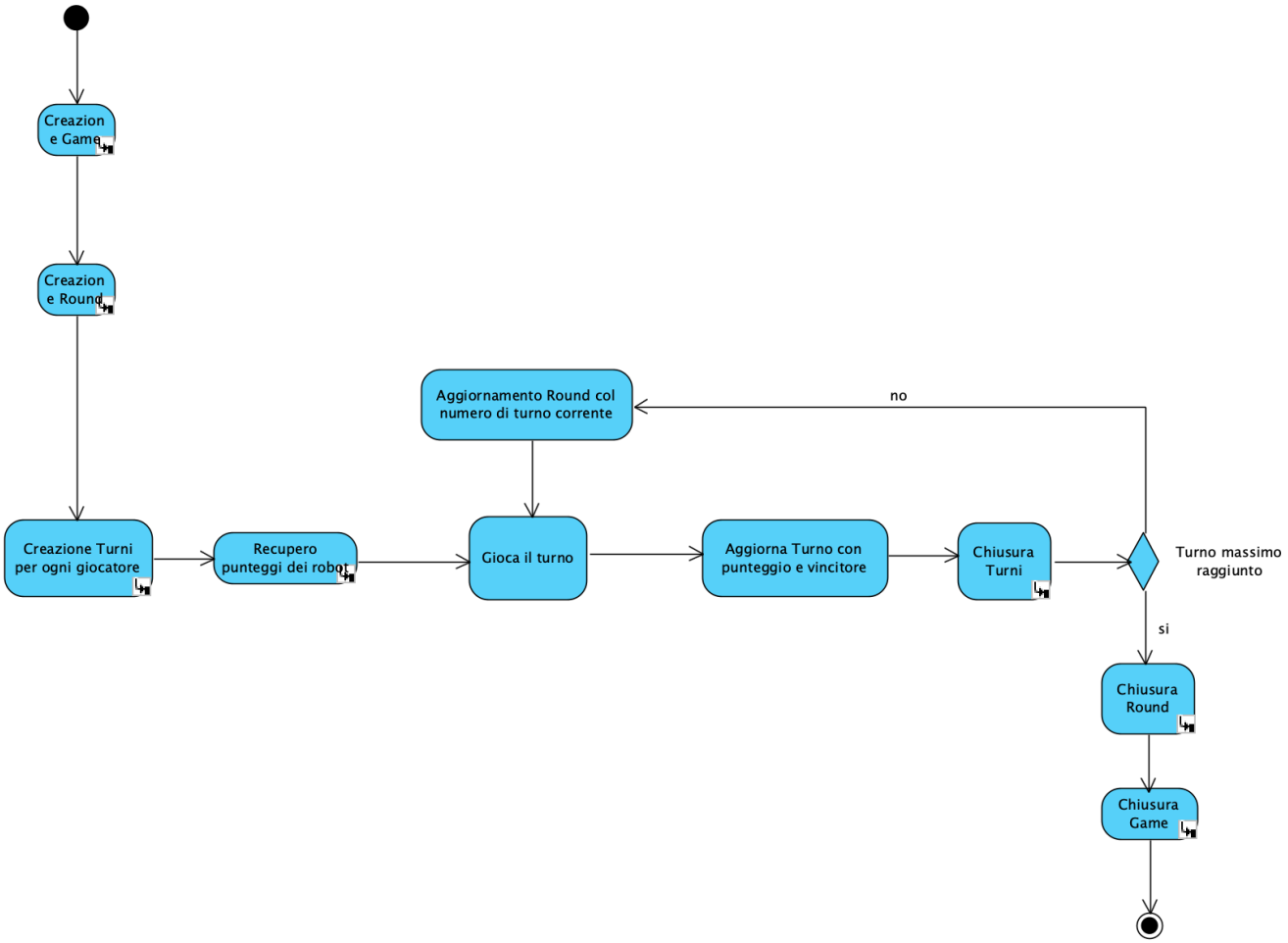
- **Admin Front End:** è un componente che utilizza l'interfaccia Admin Features, esposta dalla API Gateway. Consente di fornire una schermata personalizzata per l'admin che avrà a disposizione diverse funzionalità proprie della sola figura dell'amministratore.
- **Web Front End:** è un componente che utilizza l'interfaccia Player Features, esposta dalla API Gateway. Ha lo scopo di consentire ad un giocatore di intraprendere una partita, mediante l'utilizzo di una pagina web.
- **API Gateway:** è un componente che espone diverse interfacce (Admin Features, Player Features) e ne utilizza di altre messe a disposizione dagli altri componenti (Registration Service, Authentication Service, CRUD Class UT, Class UT Manager, Round Executor). Esso si occupa di indirizzare le richieste ai servizi giusti e di effettuare le operazioni di autenticazione e autorizzazione. Ed è l'unico punto di contatto tra back-end e front-end
- **Student Repository:** Questo componente è responsabile di salvare i dati relativi alle partite giocate da uno studente, ed espone inoltre due interfacce: **Registration Service e Authentication Service**
- **Java Class Under Test Repository:** Questo componente salva le classi testate, ed espone un'interfaccia, che identifica le funzionalità CRUD esposte dalla UT.
- **Test Class Manager:** Questo componente è responsabile dell'esecuzione del testing per ogni livello di difficoltà del gioco interagendo con il Test Run Environment

- **Game Engine:** Questo componente che rappresenta l'utente utilizzatore del servizio che si occupa di implementare le logiche di gioco utilizza l'interfaccia Test Executor e l'interfaccia CRUD Game**.**
- **Game Repository:** il componente ha la responsabilità di salvare i risultati dei test
- **Test Run Enviroment**: è un componente che a sua volta si di suddivide in altri tre sottocomponenti che identificano tre tipologie di Runner diversi, uno dedicato a ciascuna modalità di Test che è possibile adottare. Il componente Test Run Enviroment espone un'interfaccia: Test Executor.

Inoltre, il flusso della partita è riportato nel seguente **Activity Diagram** che abbiamo però provveduto a modificare, dato che nella versione originale del gioco non veniva data la possibilità di giocare più turni all'interno dello stesso Round (che ricordiamo di default abbiamo settato ad 1, in modo da permettere la presenza di un unico Round). L'activity Diagram di partenza è il seguente:

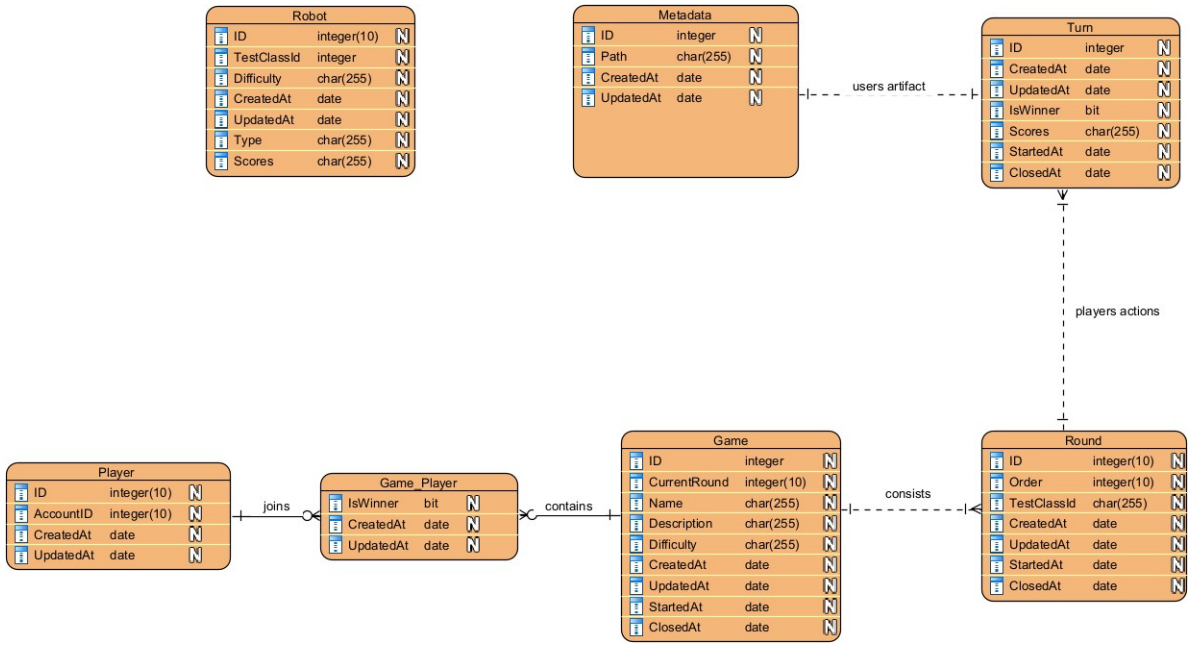


L'Activity Diagram modificato è invece il seguente:


Creazione Game
Creazione Round
Creazione Turni per ogni giocatore
Recupero punteggi dei robot
Gioca il turno
Aggiornamento Round col numero di turno corrente
Aggiorna Turno con punteggio e vincitore
Chiusura Turni
Turno massimo raggiunto
no
si
Chiusura Round
Chiusura Game

2.3 Diagramma ER

Di seguito si riporta il diagramma ER del sistema in esame

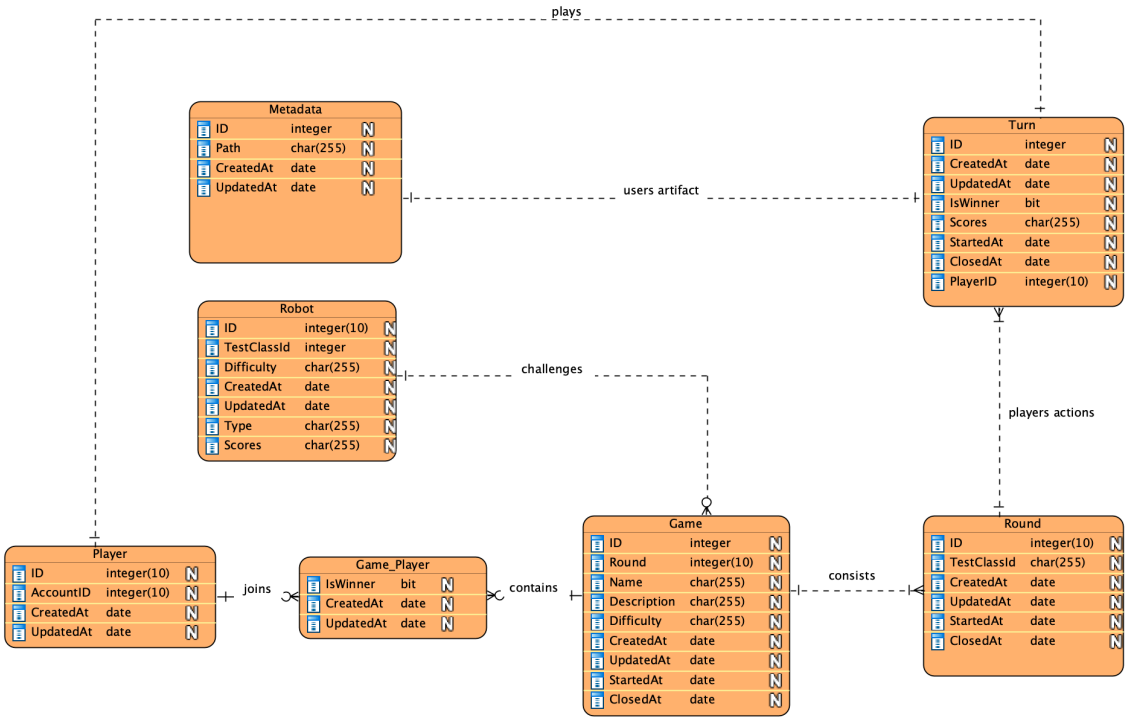


Ciascuna entità possiede i campi **CreatedAt** e **UpdatedAt**, utilizzati per tracciare la data di creazione e modifica. Inoltre, le entità in cui si tiene tracciare lo stato di apertura e chiusura posseggono inoltre gli attributi **StartedAt** e **ClosedAt**. È stata effettuata l'analisi dettagliata degli attributi per ciascuna entità:

- **Game**
  - **ID**: identificativo univoco
  - **Current Round**: indica il numero del round corrente
  - **Name**: nome della partita
  - **Description**: descrizione della partita
  - **Difficulty**: grado di difficoltà della partita
- **Player**
  - **ID**: identificativo univoco
  - **AccountID**: identificativo fornito dal componente Student Repository, che contiene i dati relativi ai giocatori

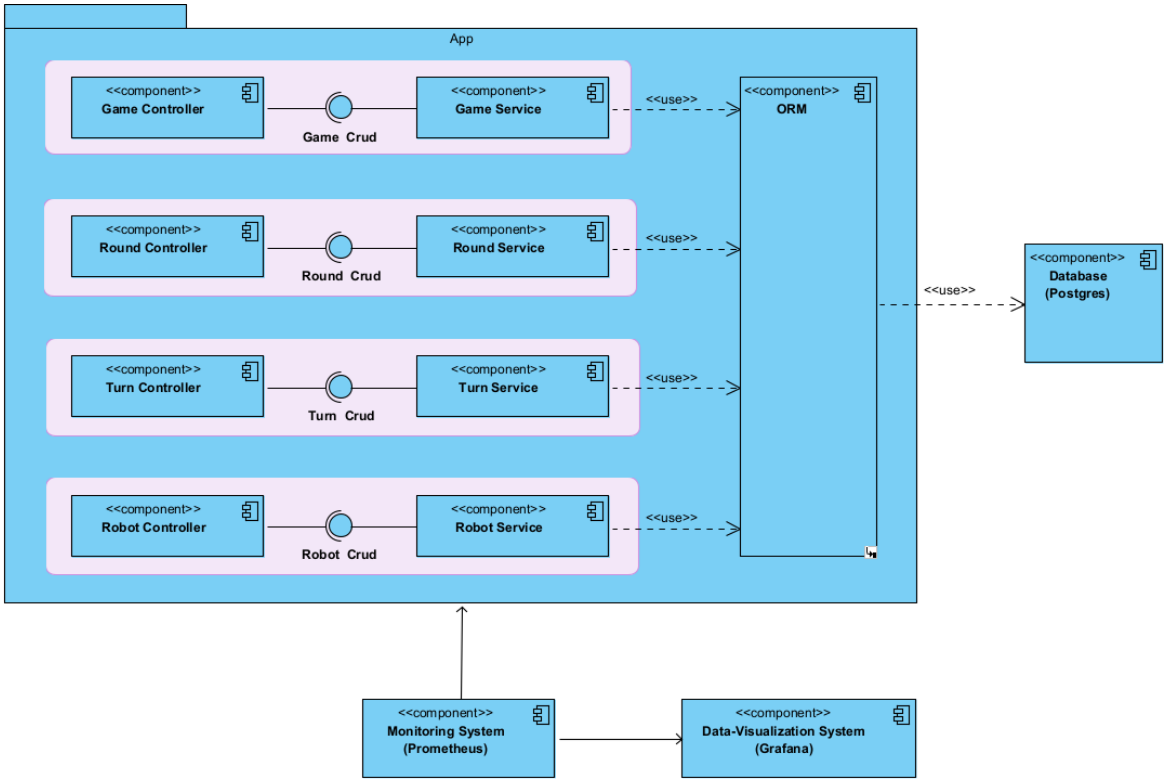
- **Game_Player**
    - **IsWinner**: valore booleano che indica il vincitore finale della partita
- **Round**
    - **ID**: identificativo univoco
    - **Order**: indica l'ordine dei round all'interno della partita (primo round, secondo, etc.)
    - **TestClassId**: Identificativo della classe da testare presente nel Class UT Repository
- **Turn**
    - **ID**: identificativo univoco
    - **IsWinner**: valore booleano che indica il vincitore del singolo round
    - **Scores**: i punteggi relativi al giocatore che ha completato un round, calcolati dal componente di testing e compilazione.
- **Metadata**
    - **ID**: identificativo univoco
    - **Path**: il percorso relativo nel file system in cui sono memorizzati i file delle classi testate dall'utente.

Il diagramma ER è stato modificato come segue aggiungendo una relazione **“uno a molti”** tra le entità Robot e Partita, dato che un Robot può giocare più partite, mentre una partita può essere giocata da uno e un solo Robot. Inoltre, è stato aggiunto un campo, Round, nell’entità Game che consente di identificare il round giocato (che sarà unico). Inoltre, è stata aggiunta una associazione **“uno a uno”** tra l’entità Player e l’entità Turn, dato che un turno può essere giocato da un solo player non essendo prevista la modalità multigiocatore ed inoltre, un giocatore può giocare un solo turno alla volta.

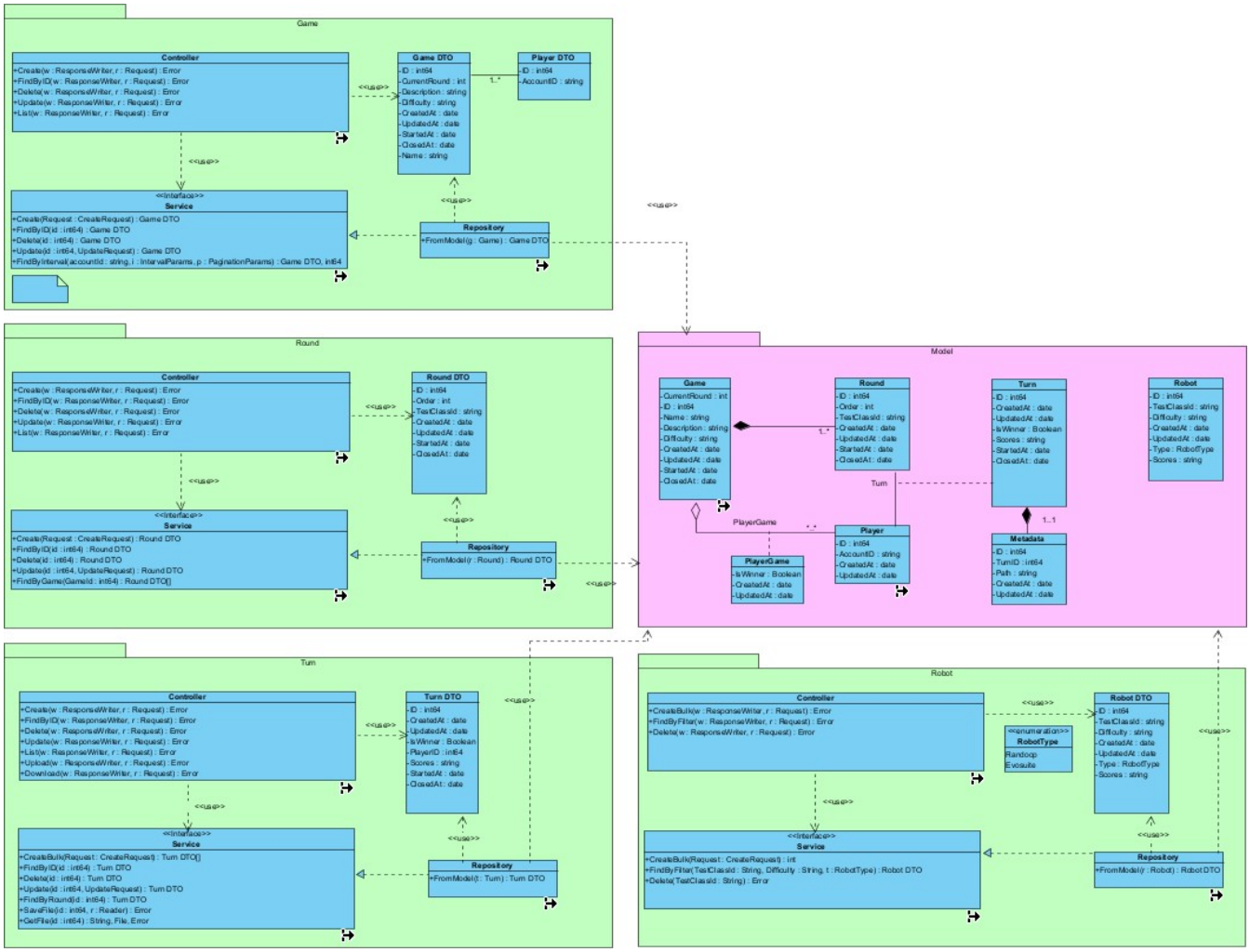


2.4 Architetture del Game Repository

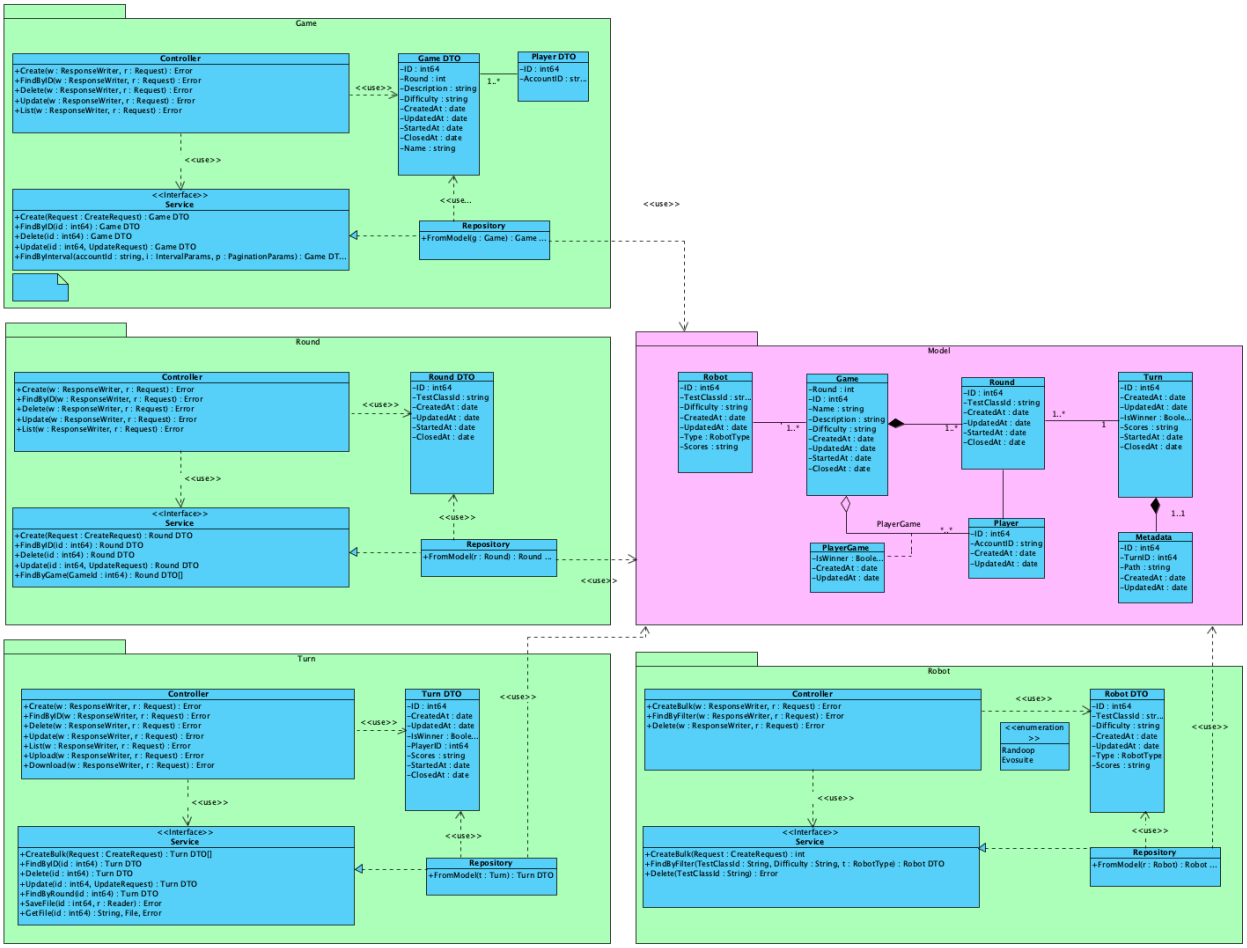
L'architettura del componente è ispirata al modello Domain Driven Design (DDD) in cui ogni entità risulta indipendente dalle altre. Ogni elemento logico dell'applicazione ha a sua disposizione un Controller che utilizza un Service. Questi due componenti (realizzati per ogni entità) interagiscono con un ORM che implementa lo schema relazionale sul database. Si noti che le fasce orizzontali rosa sono tra loro indipendenti e che l'implementazione delle classi Service sono astratte da un'interfaccia. Questa scelta, non solo garantisce un basso accoppiamento tra le entità, ma ne favorisce anche la testabilità in quanto fornendo un'implementazione mock del service è possibile testare indipendentemente le classi Controller e Service. Lo stile Domain Driven è un'estensione concettuale di quello esagonale e può essere utilizzato per implementare microservizi.



I domini applicativi rappresentati sono stati raffinati nel System Domain Model . Si noti che ciascuna entità (in verde) implementa sia il pattern DTO (dove, Il DTO è un oggetto che trasporta i dati tra i processi per ridurre il numero di chiamate ai metodi) che Dependency Injection (design pattern della OOP il cui scopo è quello di semplificare lo sviluppo e migliorare la testabilità di software di grandi dimensioni. Per utilizzare tale design pattern è sufficiente dichiarare le dipendenze di cui un componente necessita. Quando il componente verrà istanziato, un iniettore si prenderà carico di risolvere le dipendenze) e che inoltre, sono tra loro indipendenti; invece, le relazioni della base dati sono rappresentate nel package Model (in rosa).



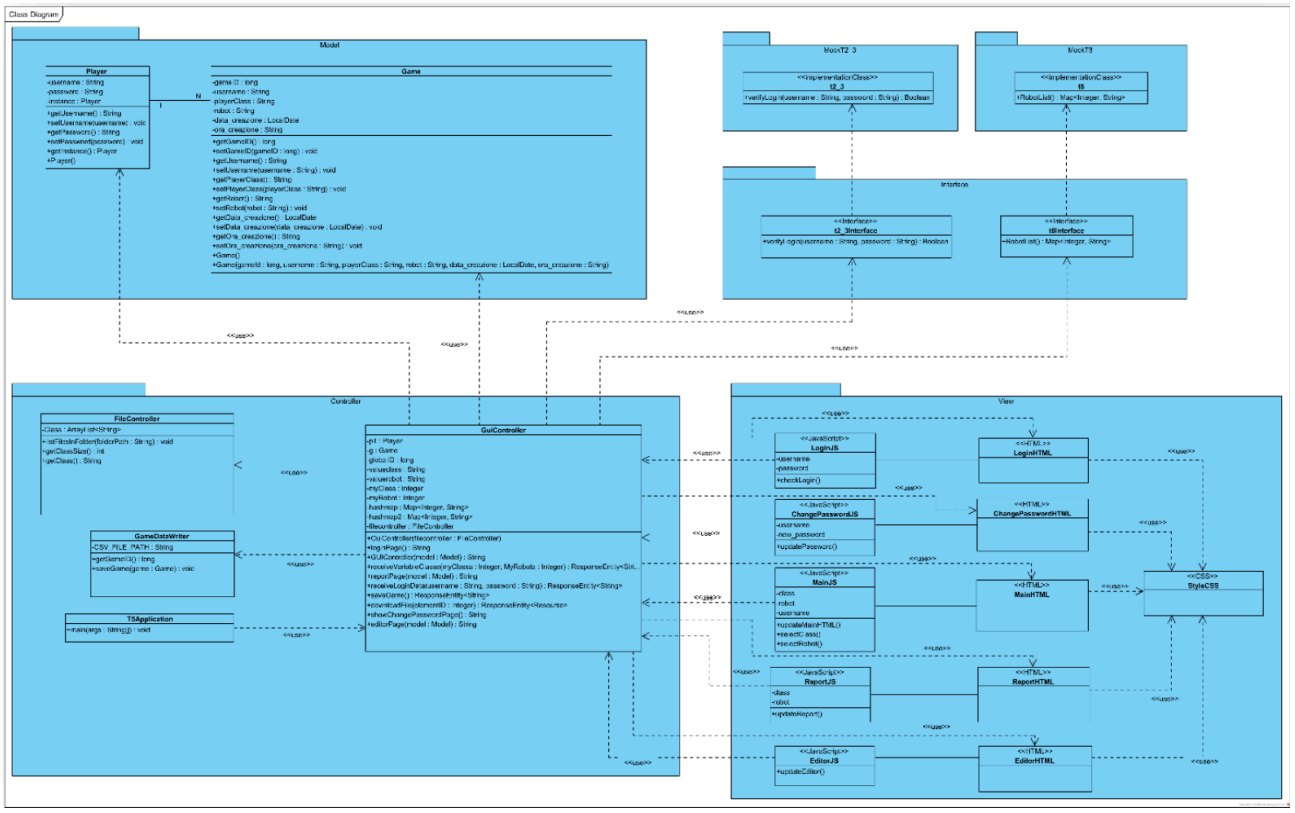
Il System Domain Model è stato modificato a seguito dei precedenti cambiamenti attuati al diagramma ER. Come è possibile notare, in entrambe le versioni del modello, abbiamo previsto la presenza della tabella associativa PlayerGame, questa scelta non è casuale, dato che nonostante ad oggi il gioco non preveda la modalità multigiocatore, non abbiamo voluto precludere la possibilità di implementarla, a coloro che in futuro vi lavoreranno.


Game
Controller
+Create(w : ResponseWriter, r : Request) : Error
+FindByID(w : ResponseWriter, r : Request) : Error
+Delete(w : ResponseWriter, r : Request) : Error
+Update(w : ResponseWriter, r : Request) : Error
+List(w : ResponseWriter, r : Request) : Error
Game DTO
-ID : int64
-Round : int
-Description : string
-Difficulty : string
-CreatedAt : date
-UpdatedAt : date
-StartedAt : date
-ClosedAt : date
-Name : string
Player DTO
-ID : int64
-AccountID : str...
<<use>>
1..*
<<Interface>>
Service
+Create(Request : CreateRequest) : Game DTO
+FindByID(id : int64) : Game DTO
+Delete(id : int64) : Game DTO
+Update(id : int64, UpdateRequest) : Game DTO
+FindByInterval(accountId : string, i : IntervalParams, p : PaginationParams) : Game DT...
Repository
+FromModel(g : Game) : Game ...
Round
Controller
+Create(w : ResponseWriter, r : Request) : Error
+FindByID(w : ResponseWriter, r : Request) : Error
+Delete(w : ResponseWriter, r : Request) : Error
+Update(w : ResponseWriter, r : Request) : Error
+List(w : ResponseWriter, r : Request) : Error
Round DTO
-ID : int64
-TestClassId : string
-CreatedAt : date
-UpdatedAt : date
-StartedAt : date
-ClosedAt : date
<<Interface>>
Service
+Create(Request : CreateRequest) : Round DTO
+FindByID(id : int64) : Round DTO
+Delete(id : int64) : Round DTO
+Update(id : int64, UpdateRequest) : Round DTO
+FindByGame(GameId : int64) : Round DTO[]
Repository
+FromModel(r : Round) : Round ...
Model
Robot
-ID : int64
-TestClassId : str...
-Difficulty : string
-CreatedAt : date
-UpdatedAt : date
-Type : RobotType
-Scores : string
Game
-Round : int
-ID : int64
-Name : string
-Description : string
-Difficulty : string
-CreatedAt : date
-UpdatedAt : date
-StartedAt : date
-ClosedAt : date
Round
-ID : int64
-TestClassId : string
-CreatedAt : date
-UpdatedAt : date
-StartedAt : date
-ClosedAt : date
Turn
-ID : int64
-CreatedAt : date
-UpdatedAt : date
-IsWinner : Boole...
-Scores : string
-StartedAt : date
-ClosedAt : date
PlayerGame
-IsWinner : Boole...
-CreatedAt : date
-UpdatedAt : date
Player
-ID : int64
-AccountID : string
-CreatedAt : date
-UpdatedAt : date
Metadata
-ID : int64
-TurnID : int64
-Path : string
-CreatedAt : date
-UpdatedAt : date
Turn
Controller
+Create(w : ResponseWriter, r : Request) : Error
+FindByID(w : ResponseWriter, r : Request) : Error
+Delete(w : ResponseWriter, r : Request) : Error
+Update(w : ResponseWriter, r : Request) : Error
+List(w : ResponseWriter, r : Request) : Error
+Upload(w : ResponseWriter, r : Request) : Error
+Download(w : ResponseWriter, r : Request) : Error
Turn DTO
-ID : int64
-CreatedAt : date
-UpdatedAt : date
-IsWinner : Boole...
-PlayerID : int64
-Scores : string
-StartedAt : date
-ClosedAt : date
<<Interface>>
Service
+CreateBulk(Request : CreateRequest) : Turn DTO[]
+FindByID(id : int64) : Turn DTO
+Delete(id : int64) : Turn DTO
+Update(id : int64, UpdateRequest) : Turn DTO
+FindByRoundId(id : int64) : Turn DTO
+SaveFile(id : int64, r : Reader) : Error
+GetFile(id : int64) : String, File, Error
Repository
+FromModel(t : Turn) : Turn DTO
Robot
Controller
+CreateBulk(w : ResponseWriter, r : Request) : Error
+FindByFilter(w : ResponseWriter, r : Request) : Error
+Delete(w : ResponseWriter, r : Request) : Error
<<enumeration>>
Randoop
Evosuite
Robot DTO
-ID : int64
-TestClassId : str...
-Difficulty : string
-CreatedAt : date
-UpdatedAt : date
-Type : RobotType
-Scores : string
<<Interface>>
Service
+CreateBulk(Request : CreateRequest) : int
+FindByFilter(TestClassId : String, Difficulty : String, t : RobotType) : Robot DTO
+Delete(TestClassId : String) : Error
Repository
+FromModel(r : Robot) : Robot ...

3 Implementazione

3.1 Analisi e modifiche dell'architettura del task T5

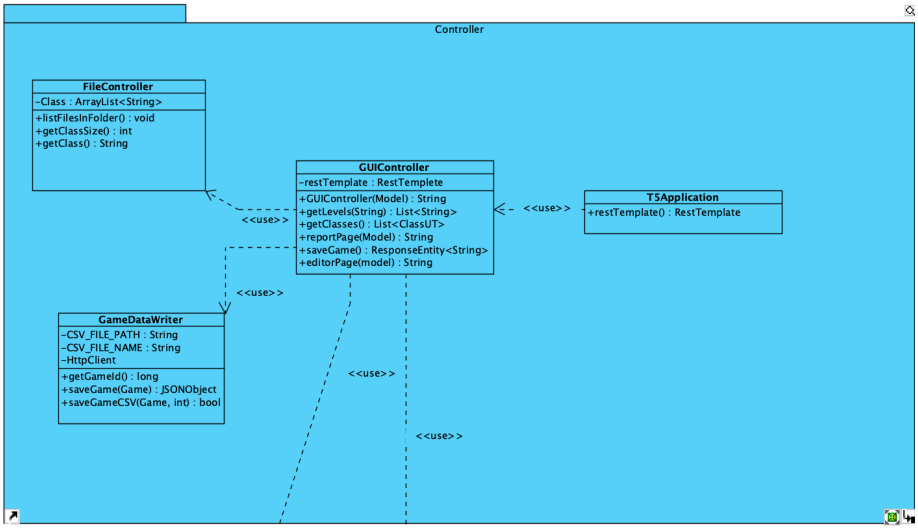
Preliminarmente abbiamo analizzato l'architettura della **Student Repository** (task T5) per comprendere al meglio come operare e per capire come effettuare le dovute modifiche ai diagrammi forniti. La student repository è descritta dal seguente Class Diagram, che ricalca la struttura del pattern MVC.



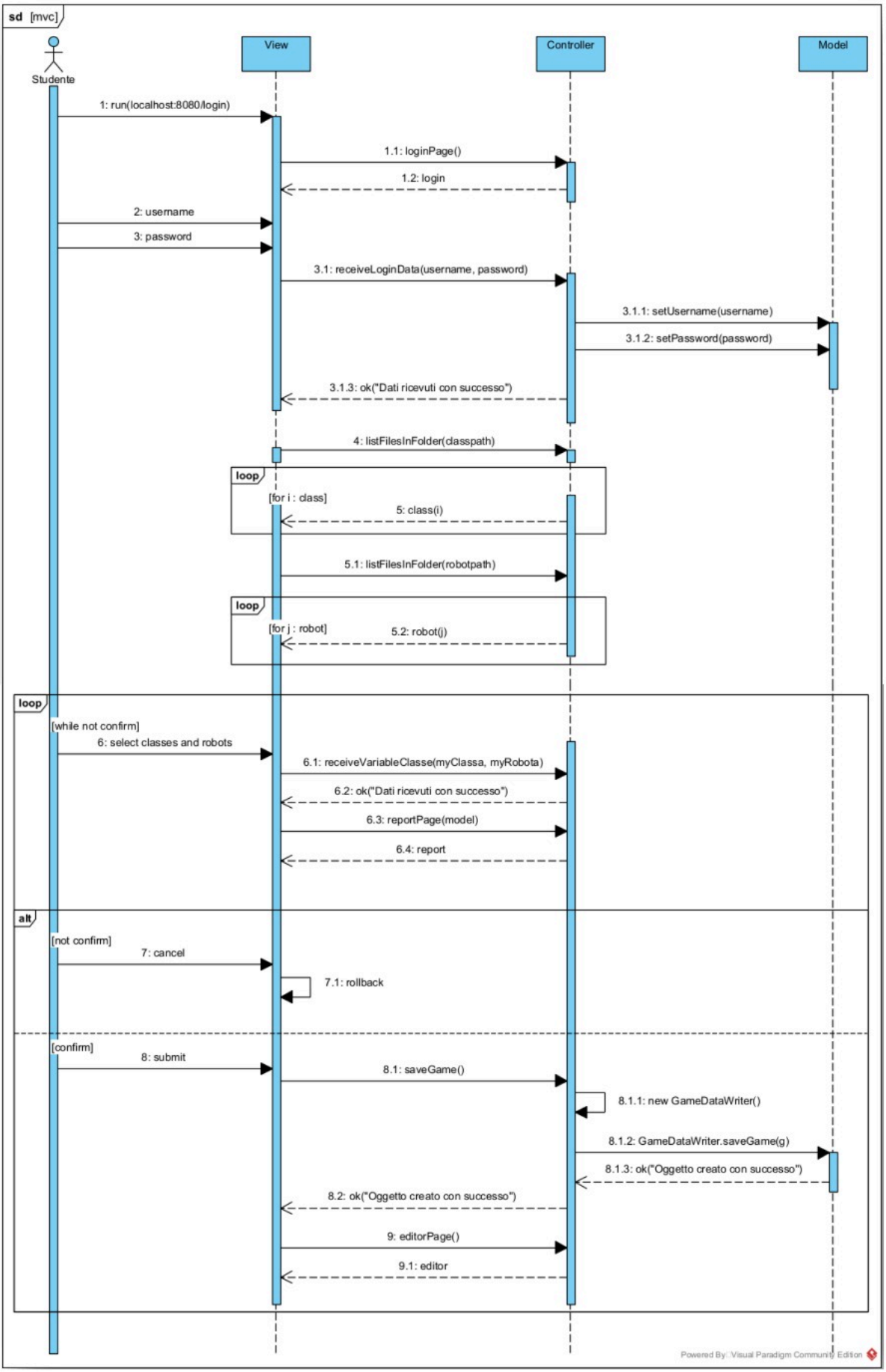
È possibile identificare quattro packages fondamentali: Controller, Model, View ed infine Interface.

- **Model**: è caratterizzato dalla classe Game e dalla classe Player
- **Controller**: è caratterizzato dalle classi GUIController, FileController, GameDataWriter e T5Application

Quest'ultimo package è stato modificato da noi, dato che avevamo la necessità, come vedremo anche nel seguito, di effettuare il salvataggio in FileSystem dei dati relativi alla partita creando un file CSV. Pertanto, a seguito dell'implementazione di metodi ad hoc, il Controller ha assunto la seguente forma, mentre tutto il resto è rimasto invariato.



Inoltre, dato che abbiamo previsto la creazione e il salvataggio dei dati della partita in un file CSV, implementando l'apposita funzione, abbiamo stilato il seguente **Sequence Diagram** per far comprendere al meglio lo sviluppo di un game fino al salvataggio nel CSV. Dall'analisi del diagramma si riscontra che: lo studente a seguito del login o della registrazione, sceglie la classe con cui giocare e il robot con il quale intende sfidarsi, può essere evinto nel loop sottostante. Successivamente alla selezione e all'inizio del game, viene creato il file CSV.


sd [mvc]
Studente
View
Controller
Model
1: run(localhost:8080/login)
1.1: loginPage()
1.2: login
2: username
3: password
3.1: receiveLoginData(username, password)
3.1.1: setUsername(username)
3.1.2: setPassword(password)
3.1.3: ok("Dati ricevuti con successo")
4: listFilesInFolder(classpath)
loop
[for i : class]
5: class(i)
5.1: listFilesInFolder(robotpath)
loop
[for j : robot]
5.2: robot(j)
loop
[while not confirm]
6: select classes and robots
6.1: receiveVariableClasse(myClassa, myRobota)
6.2: ok("Dati ricevuti con successo")
6.3: reportPage(model)
6.4: report
alt
[not confirm]
7: cancel
7.1: rollback
[confirm]
8: submit
8.1: saveGame()
8.1.1: new GameDataWriter()
8.1.2: GameDataWriter.saveGame(g)
8.1.3: ok("Oggetto creato con successo")
8.2: ok("Oggetto creato con successo")
9: editorPage()
9.1: editor
Powered By: Visual Paradigm Community Edition

3.2 Modifica delle API – Game

Le API messe a disposizione sono state modificate al fine di implementare il requisito richiesto. Per quanto riguarda l'API del Game, le modifiche apportate sono ritrovabili nel file **"game.go".** In particolare, la struct Game, identifica un tipo di dato caratterizzato da diversi campi, in particolare:

- **ID:** identifica l'identificativo univoco della partita
- **Name:** identifica il nome della partita
- **Round**: è un campo aggiunto da noi per sottolineare il fatto che la partita sarà caratterizzata da un unico Round
- **Class:** identifica il nome della classe che si intende testare durante il game
- **Description:** identifica un campo dedicato ad una eventuale descrizione della partita che si intende giocare
- **Difficulty**: identifica la difficoltà del game giocato (Easy, Medium, Hard)
- **CreatedAt:** è un campo di tipo time che consente di identificare il momento in cui il game è stato creato
- **UpdatedAt**: è un campo di tipo time che consente di identificare il momento in cui il game è stato aggiornato
- **StartedAt:** è un campo di tipo time che consente di identificare il momento in cui il game è iniziato
- **ClosedAt**: è un campo di tipo time che consente di identificare il momento in cui il game è terminato
- **Players**: identifica l'insieme di giocatori che prendono parte al gioco, ed è stato creato nell'ottica di una eventuale modifica al gioco che preveda la modalità multigiocatore
- **Robot:** identifica il robot che si intende sfidare durante il game in esame

```
type Game struct {
        ID          int64      `json:"id"`
        Name        string     `json:"name"`
        Round       int        `json:"round"`
        Class       string     `json:"class"`
        Description string     `json:"description"`
        Difficulty  string     `json:"difficulty"`
        CreatedAt   time.Time  `json:"createdAt"`
        UpdatedAt   time.Time  `json:"updatedAt"`
        StartedAt   *time.Time `json:"startedAt"`
        ClosedAt    *time.Time `json:"closedAt"`
        Players     []Player   `json:"players,omitempty"`
        Robot       int64      `json:"robot,omitempty"`
}
```

Di conseguenza, è stato necessario apportare modifiche anche alla funzione **fromModel,** presente nello stesso file "**game.go",** aggiungendo il campo Round. Questo metodo ha lo scopo di ritornare l'oggetto Game, nel quale vengono inizializzati i relativi campi, partendo dalle informazioni presenti nel Model.

```
func fromModel(g *model.Game) Game {
        return Game{
                ID:          g.ID,
                Name:        g.Name,
                Round:       g.Round, // GRUPPO A3
                Class:       g.Class,
                Description: g.Description.String,
                Difficulty:  g.Difficulty,
                CreatedAt:   g.CreatedAt,
                UpdatedAt:   g.UpdatedAt,
                StartedAt:   g.StartedAt,
                ClosedAt:    g.ClosedAt,
                Players:     parsePlayers(g.Players),
                Robot:       g.Robot,
        }
}
```

Naturalmente, avendo modificato questi campi delle strutture presenti è stata necessaria anche un'ulteriore modifica nel file "model.go". La modifica ha riguardato l'aggiunta del campo "Round" alla struttura Game, che identifica l'oggetto gioco. Come possiamo notare, dato che l'intenzione è consentire, in una futura versione l'estensione del gioco su più Round, è stato previsto un array "Rounds", però commentato.

```
type Game struct {
        ID          int64          `gorm:"primaryKey;autoIncrement"`
        Name        string         `gorm:"not null"`
        Round       int            `gorm:"default:1"`
        Class       string         `gorm:"not null"`
        Description sql.NullString `gorm:"default:null"`
        Difficulty  string         `gorm:"not null"`
        CreatedAt   time.Time      `gorm:"autoCreateTime"`
        UpdatedAt   time.Time      `gorm:"autoUpdateTime"`
        StartedAt   *time.Time     `gorm:"default:null"`
        ClosedAt    *time.Time     `gorm:"default:null"`
        Players     []Player       `gorm:"many2many:player_games;foreignKey:ID;joinForeignKey:GameID;References:AccountID;joinReferences:PlayerID"`
        Robot       int64          `gorm:"default:null"`
```

3.3 Modifica delle API – Round

Per quanto riguarda l'API del Round, le modifiche apportate sono ritrovabili nel file **"service.go".** Abbiamo provveduto all'aggiunta della funzione **FindByGame,** tale metodo è associato ad un oggetto di tipo Repository ed ha lo scopo di cercare e restituire un Round in base all'Id del game. È possibile analizzare più nel dettaglio il metodo: viene dichiarato un oggetto: rounds, preso dal modello dati dell'applicazione. Viene poi fatta una query sul Database, al fine di trovare l'oggetto Round che corrisponde ad un round del game con l'id specificato. Il round viene preso mediante la funzione fromModel ed infine lo si restituisce.

```
func (rs *Repository) FindByGame(id int64) ([]Round, error) {
        var rounds []model.Round

        err := rs.db.
                Where(&model.Round{GameID: id}).
                Find(&rounds).
                Error

        resp := make([]Round, len(rounds))
        for i, round := range rounds {
                resp[i] = fromModel(&round)
        }

        return resp, api.MakeServiceError(err)
}
```

Per quanto riguarda Round, stiamo facendo riferimento alla struttura presente nel file "**round.go**" della API di round. Tale struttura è caratterizzata da diversi campi:

- **ID:** rappresenta l'identificativo del Round
- **TestClassId:** rappresenta l'identificativo della classe da testare
- **GameID:** che rappresenta l'identificativo del game che stiamo giocando
- **CreatedAt:** è un campo di tipo time che consente di identificare il momento in cui il round è stato creato
- **UpdatedAt:** è un campo di tipo time che consente di identificare il momento in cui il round è stato aggiornato
- **StartedAt:** è un campo di tipo time che consente di identificare il momento in cui il round è iniziato
- **ClosedAt:** è un campo di tipo time che consente di identificare il momento in cui il round è terminato

```
type Round struct {
        ID          int64      `json:"id"`
        TestClassId string     `json:"testClassId"`
        GameID      int64      `json:"gameId"`
        CreatedAt   time.Time  `json:"createdAt"`
        UpdatedAt   time.Time  `json:"updatedAt"`
        StartedAt   *time.Time `json:"startedAt"`
        ClosedAt    *time.Time `json:"closedAt"`
}
```

3.4 Modifica delle API - Turn

L'unica modifica apportata alla API riguardante il turno, è stata fatta a livello del file **"service_test.go".** Semplicemente abbiamo previsto anche in questo caso la presenza di un unico round dato che il concetto di turno e Round andavano in conflitto tra loro dato che non c'era una netta differenza tra l'uno e l'altro.

```
func (suite *RepositorySuite) SeedTestData() {
        // Create a game with rounds and turns
        suite.T().Helper()

        // Create a test game
        game := model.Game{
                Name: "Test Game",

                /* Futura implementazione di più rounds
                **
                Rounds: []model.Round{
                        {
                                Order:       1,
                                TestClassId: "test",
                                Turns: []model.Turn{
                                        {
                                                PlayerID: 1,          // Replace with your desired player ID
                                                Scores:   "10,20,30", // Replace with your desired scores
                                        },
                                        // Add more turns as needed
                                },
                        },
                        // Add more rounds as needed
                },
                */
        }
```

3.5 Modifiche al Codice del T5

Il task T5 fornisce l'API per la gestione dei dati inerenti alla creazione e l'avvio di una partita, in particolare si occupa del front-end dell'applicazione che permette all'utente di autenticarsi e di iniziare una partita, e del back-end necessario per effettuare gli opportuni salvataggi in database e su filesystem. Analizzando quanto presente in T5, ci siamo resi conto che la logica di salvataggio dei dati della partita su filesystem ancora non era stato implementata, per cui una volta compresa la struttura della ***Student Repository*** abbiamo apportato le opportune modifiche al codice.

Il pattern adottato per il T5 è il ***Model View Controller***, dove:

- La *View* gestisce la logica di presentazione e coincide con il front-end della Web-App che consente al giocatore di effettuare registrazione e login, scegliere classe e robot da sfidare e di iniziare una nuova partita; è stato realizzato utilizzando Javascript, HTML e CSS;
- Il *Model* gestisce la logica di business e definisce le entità di interesse, consentendo dunque l'accesso ai dati necessari e l'aggiornamento delle viste: lo studente che effettuerà la partita (**Player**), della classe da testare (**ClassUT**) e la partita stessa (**Game**); è stato realizzato in Java;
- Il *Controller* gestisce infine la logica di controllo, dunque i dati di input provenienti dall'applicazione e le elaborazioni necessarie per aggiornare il Model; in particolare sono state realizzate due importanti classi Java, cioè **GUIController** che, attraverso il framework Spring, si occupa di gestire le operazioni CRUD definendo diversi metodi per ogni route e operazione, e **GameDataWriter** che definisce i metodi per l'interazione col database del volume T4 (Game Repository) e con il FileSystem del T5 (Student Repository).

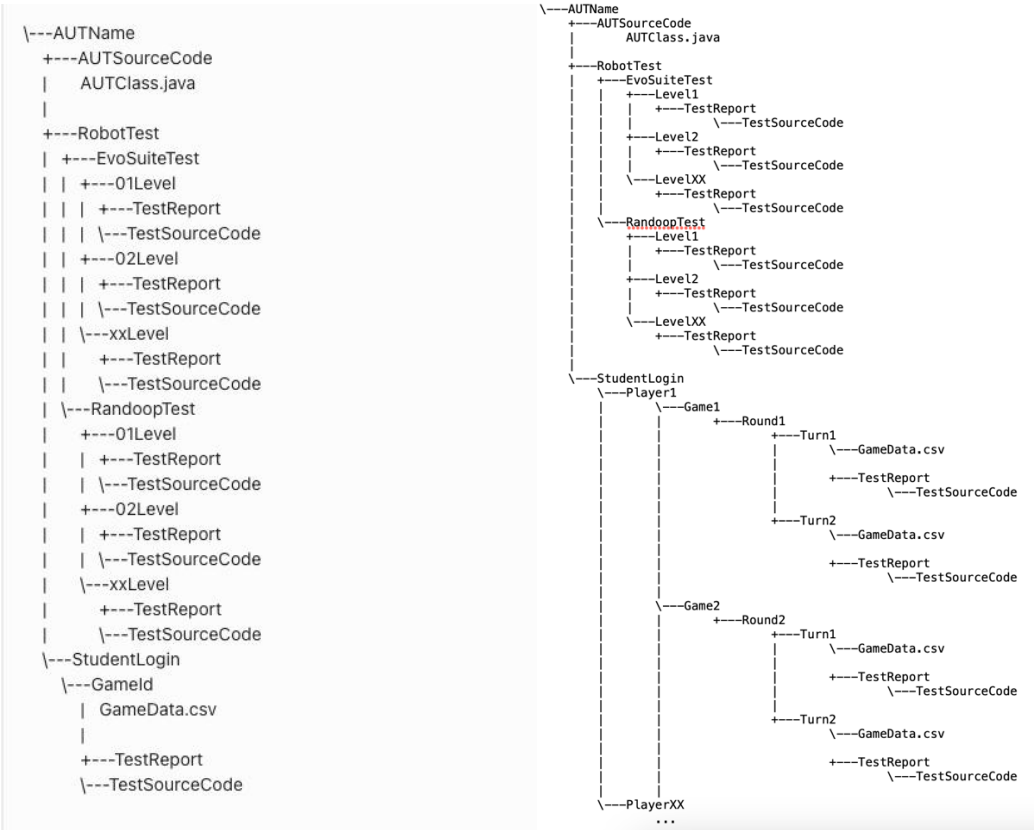
Le prime modifiche effettuate riguardano il Model, in particolare per garantire la coerenza dei dati salvati in Database e quelli in File System abbiamo dovuto modificare la classe Game per adattarla alla struttura del Database di T4. Vediamo dunque che anche qui non sono presenti i campi *currentRound* e *order* e che sono stati aggiunti i campi *playerId* e *robot.*

```
public class Game {
    private long id;
    private String name;
    private int round;
    private String testedClass;
    private String description;
    private String difficulty;
    private LocalDate createdAt;
    private LocalDate updateAt;
    private LocalDate startedAt;
    private LocalDate closedAt;
    private long playerId; // Adattare per il multi-player
    private String robot;
```

Successivamente, ci siamo concentrati sul Controller, in particolare abbiamo implementato un nuovo metodo **saveGameCSV** nella classe **GameDataWriter** per effettuare il salvataggio dei dati di gioco su filesystem all'interno di un file CSV, secondo la struttura della nuova Student Repository.

```
public boolean saveGameCSV(Game game, int turnID) {
    long playerID = game.getPlayerId();
    long gameID = game.getId();
    int roundID = game.getRound();

    // Al path bisogna aggiungere PlayerID/GameID/RoundID/TurnID e poi il nome del file
    String fileName = CSV_FILE_PATH + playerID + "/" + gameID + "/" + roundID + "/" + turnID + CSV_FILE_NAME;

    Path path = Paths.get(fileName);

    if (!Files.exists(path)) {
        try {
            Files.createDirectories(path.getParent());
        } catch (IOException e) {
            System.out.println("Errore durante la creazione della directory.");
            e.printStackTrace();
        }
    }



    try {
        File file = new File(fileName);

        if (!file.exists()) {
            file.createNewFile();
        }

        FileWriter writer = new FileWriter(file);
        CSVPrinter csvPrinter = new CSVPrinter(writer, CSVFormat.EXCEL);

        csvPrinter.printRecord(
                "GameID",
                "Name",
                "Round",
                "Class",
                "Description",
                "Difficulty",
                "CreatedAt",
                "UpdatedAt",
                "StartedAt",
                "ClosedAt",
                "PlayerID",
                "Robot"
            );

        csvPrinter.printRecord(
            game.getId(),
            game.getName(),
            game.getRound(),
            game.getTestedClass(),
            game.getDescription(),
            game.getDifficulty(),
            game.getCreatedAt(),
            game.getUpdateAt(),
            game.getStartedAt(),
            game.getClosedAt(),
            game.getPlayerId(),
            game.getRobot()
        );

        csvPrinter.flush();
        csvPrinter.close();
        writer.close();

        System.out.println("Game è stato salvato correttamente nel file CSV.");

        return true;
    } catch (IOException e) {
        System.out.println("Errore durante la scrittura del file CSV.");
        e.printStackTrace();

        return false;
    }

}
// FINE MODIFICHE 2.0
```

A sinistra osserviamo la precedente gestione del File System del T5 mentre a destra quella nuova, migliorata in quanto nella cartella *StudentLogin* vengono cretate di volta in volta cartelle e apposite sulla base degli id del giocatore, della partita, dell round e del turno, evitando sovrascritture di file indesiderate.



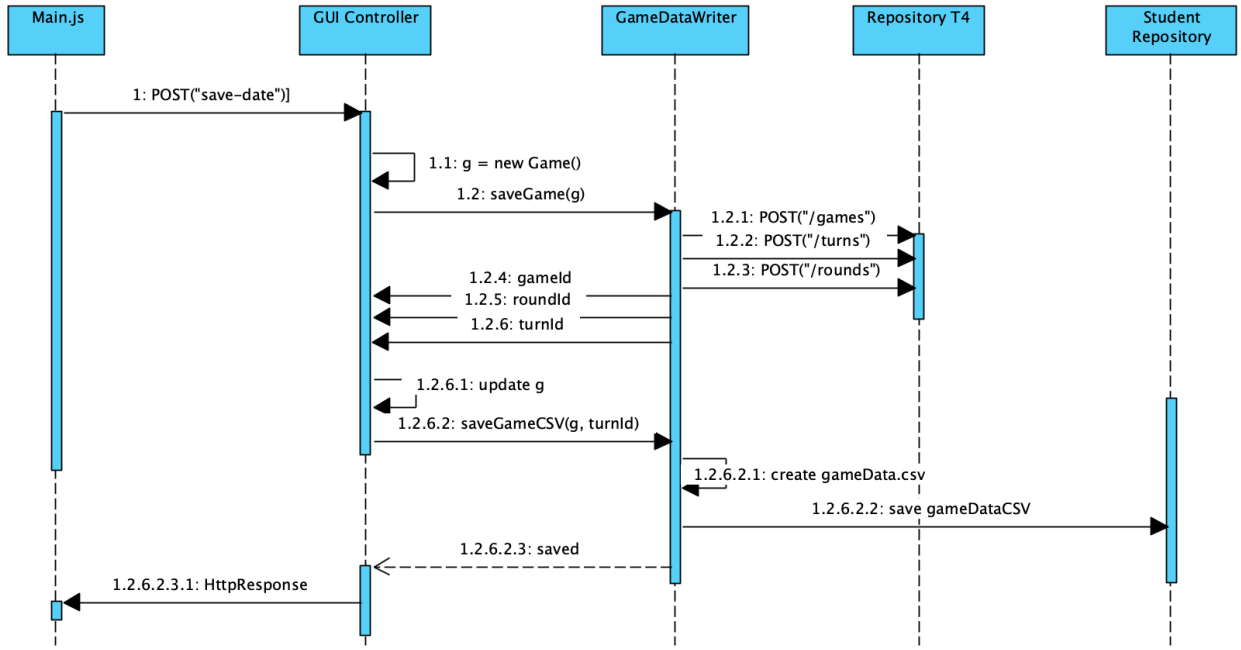
Abbiamo inoltre apportato le opportune modifiche per prelevare correttamente i dati che vengono trasferiti tramite oggetti JSON sulla base delle modifiche alla classe Game. Nella classe **GUIController**, come accennato in precedenza, tramite il framework *Spring* sono stati realizzati i metodi che si occupano di gestire gli input dello studente, in particolare vi sono una serie di *PostMapping* per le route di interesse; di interesse per noi è stato il metodo da eseguire all'avvio della partita, che viene attivato quando il **main.js** della web app effettua una POST alla route *"/save-data"*, tale metodo è **saveGame** che abbiamo modificato aggiungendo la chiamata alla funzione *saveGameCSV* e modificando opportunamente la creazione dell'oggetto Game. Esso riceve in input: robot, classe, playerId e difficulty e crea successivamente l'oggetto "game", riempiendone gli appositi campi tramite i metodi forniti dal Model. Il suo output è una stringa JSON gli ID di partita, round e turno.

```java
@PostMapping("/save-data")
public ResponseEntity<String> saveGame(@RequestParam("playerId") int playerId, @RequestParam("robot") String robot,
        @RequestParam("classe") String classe, @RequestParam("difficulty") String difficulty,
        HttpServletRequest request) {

    if (!request.getHeader(name:"X-UserID").equals(String.valueOf(playerId)))
        return ResponseEntity.badRequest().body(body:"Unauthorized");

    /*
     * DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm");
     * LocalTime oraCorrente = LocalTime.now();
     * String oraFormattata = oraCorrente.format(formatter);
     */

    GameDataWriter gameDataWriter = new GameDataWriter();

    Game g = new Game(playerId, description:"descrizione", name:"nome", difficulty);

    // Aggiungere orario alla data
    g.setCreatedAt(LocalDate.now());
    g.setTestedClass(classe);
    g.setRobot(robot);

    JSONObject ids = gameDataWriter.saveGame(g);

    if (ids == null)
        return ResponseEntity.badRequest().body(body:"Bad Request");

    long gameID = ids.getLong(name:"game_id");
    int roundID = ids.getInt(name:"round_id");
    int turnID = ids.getInt(name:"turn_id");

    g.setId(gameID);
    g.setRound(roundID);

    boolean saved = gameDataWriter.saveGameCSV(g, turnID);

    if (!saved)
        return ResponseEntity.internalServerError().body(body:"Game not saved in filesystem");

    return ResponseEntity.ok(ids.toString());
}
```

A sua volta, *saveGame* richiama un metodo omonimo appartenente alla classe **GameDataWriter** che prende in input l'oggetto *game* e si occupa di interagire tramite operazioni CRUD HTTP con il database di T4 per inizializzare e salvare i dati relativi a partita, round e turno appena creati, restituendo in output una HttpResponse che consente al controller di capire se le operazioni sono andate a buon fine o meno; alla fine viene utilizzato un ulteriore metodo di tale classe, cioè **saveGameCSV** di cui detto prima, che riceve in input l'oggetto *game* e l'intero *turnID* e che consente il salvataggio dei dati ella partita sul file CSV nella directory specificata, in risposta viene dornito un booleano *saved* che conente al controller di capire se il savataggio in FileSystem è stato eseguito correttamente o meno. Vediamo dunque le modifiche apportare al metodo saveGame della classe GameDataWriter:

```java
public JSONObject saveGame(Game game) {
    try {
        String time = ZonedDateTime.now(ZoneOffset.UTC).format(DateTimeFormatter.ISO_INSTANT);
        JSONObject obj = new JSONObject();

        obj.put("difficulty", game.getDifficulty());
        obj.put("name", game.getName());
        obj.put("description", game.getDescription());
        obj.put("startedAt", time);

        JSONArray playersArray = new JSONArray();
        playersArray.put(String.valueOf(game.getPlayerId()));

        obj.put("players", playersArray);

        HttpPost httpPost = new HttpPost("http://t4-g18-app-1:3000/games");
        StringEntity jsonEntity = new StringEntity(obj.toString(), ContentType.APPLICATION_JSON);

        httpPost.setEntity(jsonEntity);

        HttpResponse httpResponse = httpClient.execute(httpPost);
        int statusCode = httpResponse.getStatusLine().getStatusCode();

        if (statusCode > 299) {
            System.err.println(EntityUtils.toString(httpResponse.getEntity()));
            return null;
        }

        HttpEntity responseEntity = httpResponse.getEntity();
        String responseBody = EntityUtils.toString(responseEntity);
        JSONObject responseObj = new JSONObject(responseBody);

        Long gameID = responseObj.getLong("id");

        JSONObject round = new JSONObject();
        round.put("gameId", gameID);
        round.put("testClassId", game.getTestedClass());
        round.put("startedAt", time);

        httpPost = new HttpPost("http://t4-g18-app-1:3000/rounds");
        jsonEntity = new StringEntity(round.toString(), ContentType.APPLICATION_JSON);

        httpPost.setEntity(jsonEntity);

        httpResponse = httpClient.execute(httpPost);
        statusCode = httpResponse.getStatusLine().getStatusCode();

        if (statusCode > 299) {
            System.err.println(EntityUtils.toString(httpResponse.getEntity()));
            return null;
        }

        responseEntity = httpResponse.getEntity();
        responseBody = EntityUtils.toString(responseEntity);
        responseObj = new JSONObject(responseBody);

        // salvo il round id che l'Api mi restituisce
        Integer roundID = responseObj.getInt("id");

        JSONObject turn = new JSONObject();

        turn.put("players", playersArray);
        turn.put("roundId", roundID);
        turn.put("startedAt", time);

        httpPost = new HttpPost("http://t4-g18-app-1:3000/turns");
        jsonEntity = new StringEntity(turn.toString(), ContentType.APPLICATION_JSON);

        httpPost.setEntity(jsonEntity);

        httpResponse = httpClient.execute(httpPost);
        statusCode = httpResponse.getStatusLine().getStatusCode();

        if (statusCode > 299) {
            System.err.println(EntityUtils.toString(httpResponse.getEntity()));
            return null;
        }

        responseEntity = httpResponse.getEntity();
        responseBody = EntityUtils.toString(responseEntity);

        JSONArray responseArrayObj = new JSONArray(responseBody);

        // salvo il turn id che l'Api mi restituisce
        Integer turnID = responseArrayObj.getJSONObject(0).getInt("id");

        JSONObject resp = new JSONObject();
        resp.put("game_id", gameID);
        resp.put("round_id", roundID);
        resp.put("turn_id", turnID);

        return resp;
    } catch (IOException e) {
        e.printStackTrace();
        return null;
    }

}
```

Nel complesso, possiamo visualizzare le interazioni tra le varie classi di cui si è parlato visualizzando il seguente **Sequence Diagram,** esplicativo della funzionalità implementata per maggior chiarezza.



4 Testing

Il testing della soluzione avviene in maniera automatica ed è stato integrato nel ciclo di sviluppo e rilascio del software.

4.1 Unit Testing

Il componente testato è il servizio REST descritto nei paragrafi precedenti. Per garantire una maggiore robustezza del sistema, l'oggetto principale del testing di unit è l'interfaccia input/output dell'applicazione. Sono oggetto di questa attività i controller, ognuno dei quali viene testato con JSON sintatticamente validi verificando il risultato a un particolare codice dello standard HTTP. In questa fase, è stato utilizzato un approccio table testing secondo cui per ogni componente è stata progettata una tabella di test case, come mostrato nella seguente tabella:

TC	Descrizione	Risultato atteso (Codice HTTP)
T01-GameNotExists	Ricerca di partita non esistente	404
T02-GameExists	Ricerca di partita esistente	200
T03-BadID	Ricerca partita con id invalido	400
T11-BadJson	Creazione partita con *JSON* invalido	400
T12-GameCreated	Creazione partita con *JSON* valido	201

4.2 Integration Testing

Il testing di integrazione mette insieme il componente REST Server con un'istanza di un database reale. Con riferimento all'architettura della soluzione, sono state testate le funzioni del layer Service che utilizza la base dati ed è utilizzata dai controller. In questa attivit., l'obiettivo principale è quello di verificare la coerenza dei dati e la correttezza dello schema descritto precedentemente. Analogamente al testing di unit., è stato utilizzato l'approccio del table testing. Ad esempio, per la procedura di salvataggio dei file appartenenti a un giocatore, . stata progettata la Tabella 11. Dal punto di vista implementativo, la funzione riceve in ingresso:

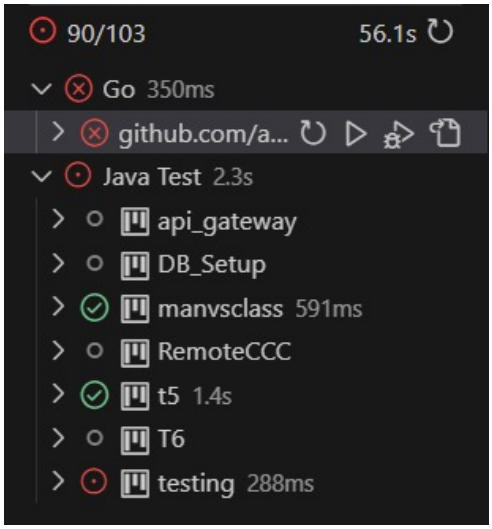
- **id**: identificativo del turno del giocatore rappresentato come intero a 64bit;
- **reader**: flusso di byte astratto dall'interfaccia io.Reader;

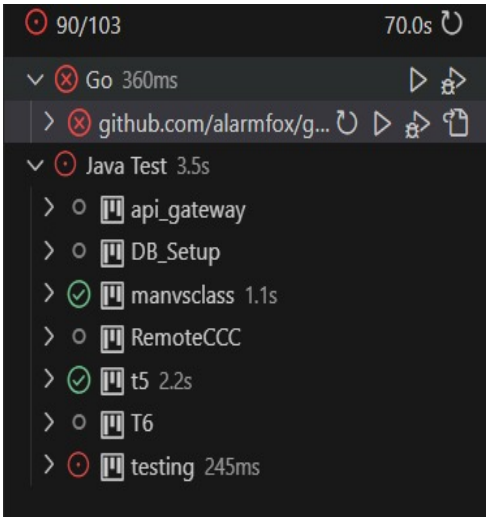
TC	Descrizione	Risultato atteso
T51-NotAZip	Il corpo della richiesta non è uno zip	ErrNotAZip
T52-Success	Il file ZIP viene correttamente salvato	OK
T53-EmptyFile	Il file ZIP è vuoto	OK
T54-InvalidTurnID	L'ID del turno è negativo	ErrNotFound
T55-NullBody	Se l'ingresso ha valore NULL	ErrInvalidParam
T56-NotFound	Il turno non viene trovato all'interno del database	ErrNotFound

Inoltre, per rendere predicibile il valore dei valori assegnati dal database (ad esempio il valore delle sequenze per l'assegnazione degli ID quando viene effettuata una INSERT) dopo ogni test case viene eseguita l'operazione TRUNCATE con lo scopo di effettuare il reset di tutte le strutture del database e rendere il testing riproducibile. In Figura è mostrata l'esecuzione del test di integrazione con il target test-integration. In particolare, sono previsti alcuni parametri di configurazione:

- CI=1: attraverso il parametro DB_URI viene specificato un'istanza già esistente per il database. L'operazione comporta la perdita di dati;
- CI non specificato: viene creata un'istanza usa e getta del database con Docker;

Si lasciano di seguito i risultati dei test di integrazione svolti precedentemente e a seguito delle modifiche.

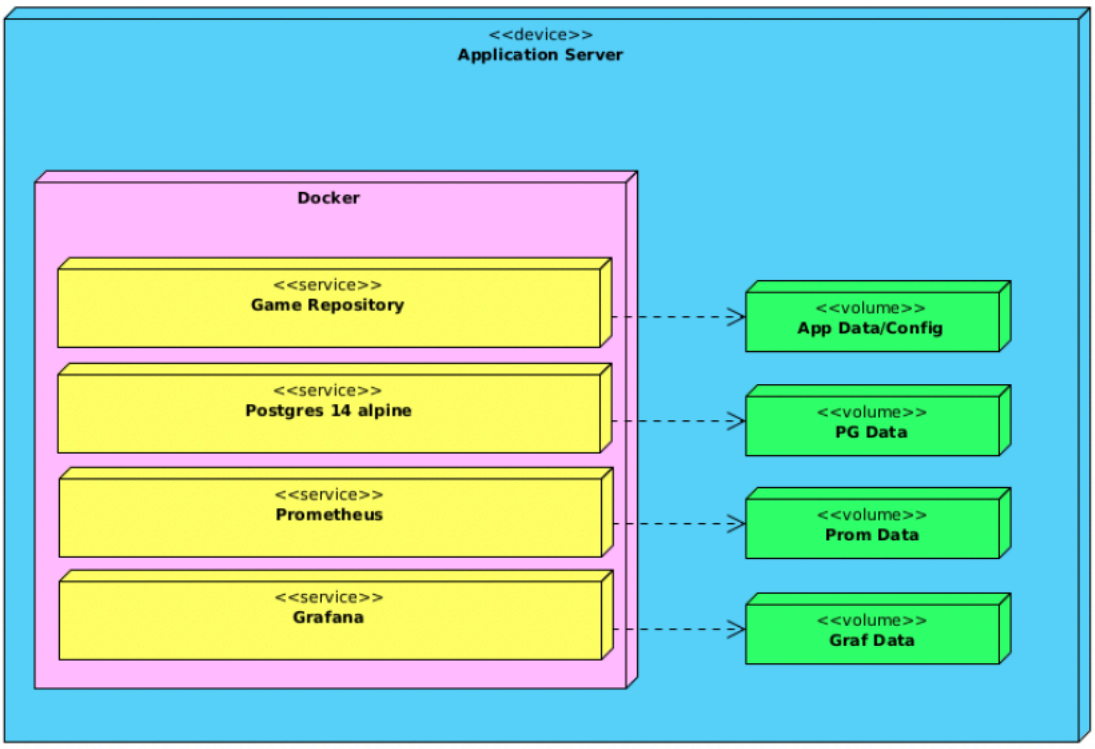




5 Deployment

In questa sezione, si fa riferimento all'installazione del solo componente Rest Server; per gli altri componenti e per maggiori dettagli si rimanda alla guida ufficiale di installazione e configurazione ed inoltre alla documentazione della predenete versione del software.

In generale, l'architettura per installare il volume T4 è illustrata nel seguente deployment diagram:



Il container T4 si compone infatti di quattro servizi e per ognuno dei quali viene creato un apposito volume:

1. **GameRepository**, scritto in Go, che definisce entità ed API per la gestione di partite, giocatori, robot, round e turni;
2. **Postgres**, cioè il database con cui il Game Repository interagisce per salvare i dati;
3. **Prometheus**, cioè un servizio di monitoraggio di diverse metriche per ambienti distribuiti;
4. **Grafana**, che consente di visualizzare le metriche raccolte da Prometheus.

Per quanto concerne il volume T5, in seguito alle modifiche è stato necessario ricompilare le classi per aggiornare l'artefatto *t5-0.0.1-SNAPSHOT.jar*, per farlo bisogna installare **Maven** sulla propria macchina, pois spostarsi tramite terminale nella directory **"T5-G2/t5/"** del progetto ed eseguire il comando: **mvn clean install**. Successivamente bisogna far ripartire l'installazione del gioco lanciando il file **installer.bat** nella directory principale del progetto.

Visualizziamo il diagramma di deployment del volume T5: