Perudo online

Enrico Lumini
enrico.lumini@studio.unibo.it

November 2023

Abstract

Il progetto prevede lo sviluppo di una versione online del gioco da tavolo "Perudo". In questo gioco, da 2 a 6 giocatori, ogni giocatore ad inizio partita ha a disposizione 5 dadi ed un bicchiere. Tutti i giocatori mischiano i propri dadi all'interno del proprio bicchiere ed il giocatore di turno effettua una supposizione sui dadi totali. Il giocatore successivo può rilanciare la supposizione oppure dubitare. In caso di dubito il giocatore che ha sbagliato perde un dado e si incomincia un nuovo turno, mischiando nuovamente i dadi. Se invece il giocatore rilancia si va avanti finché un giocatore non dubita. Lo scopo del gioco consiste nell'essere l'ultimo giocatore ad avere almeno un dado a disposizione.

Contents

1 Goal/Requirements **2**
- 1.1 Functional requirements . . . 2
- 1.2 Non-functional requirements . . . 5
- 1.3 Scenarios . . . 5
  - 1.3.1 Pre-match scenario . . . 6
  - 1.3.2 During-match scenario . . . 6
  - 1.3.3 Post-match scenario . . . 7
- 1.4 Self-assessment policy . . . 7

2 Design **9**
- 2.1 Application structure . . . 9
  - 2.1.1 Client structure . . . 10
  - 2.1.2 Server structure . . . 15
  - 2.1.3 Messages structure . . . 18
- 2.2 Behaviour . . . 19
- 2.3 Interaction . . . 21

3 Implementation Details **22**
- 3.1 Client . . . 22
- 3.2 Server . . . 22
- 3.3 Common . . . 23

4 Validation **24**

5 Deployment **26**

6 Usage Examples **27**

7 Conclusions **34**
- 7.1 Future Works . . . 34
- 7.2 What did i learned . . . 34

Chapter 1

Goal/Requirements

L'obiettivo del progetto è quello di realizzare una versione distribuita del gioco da tavolo "Perudo" che permetta la presenza di molteplici partite in contemporanea.

1.1 Functional requirements

Di seguito sono elencati i requisiti funzionali dell'applicazione sotto forma di domande e risposte:

- *Come è possibile entrare in una lobby?*

  All'avvio dell'applicazione ogni utente può creare una nuova lobby oppure unirsi ad una già esistente, selezionandola da una apposita lista mostrata nella prima schermata. Una lobby più contenere da 2 a 6 giocatori, non è possibile pertanto unirsi ad una lobby già piena.

- *Ci sono limitazioni riguardo la scelta dei nomi?*

  In una stessa lobby non possono essere presenti due giocatori aventi lo stesso nome; tuttavia, uno stesso nome può essere riutilizzato in differenti lobby.

- *Cosa accade se prima che inizi la partita, l'utente che ha creato la lobby esce?*

  Se l'utente che ha creato la lobby crasha, esce dall'applicazione o dalla lobby stessa tutti gli altri utenti appartenenti a quella lobby vengono notificati e riportati alla schermata iniziale mentre la lobby viene eliminata.

- *Cosa accade invece se è un giocatore, diverso dal creatore, a lasciare la lobby?*

  Se un giocatore non creatore della lobby crasha, chiude l'applicazione o esce dalla lobby prima che la partita abbia inizio, quest'ultimo viene rimosso dalla stanza ma la lobby non cessa di esistere. Un'altro giocatore,

compreso il giocatore che ha appena lasciato la stanza, potrà rientrare all'interno della lobby occupando il posto che si è liberato.

- *Chi può avviare una partita?*

  Una partita può essere avviata solo dal creatore della lobby e solamente quando quest'ultima è al completo. Da quando la partita inizia, non è più possibile per nuovi giocatori unirsi alla lobby.

- *Cosa accade quando la partita viene avviata?*

  Quando la partita inizia a tutti i giocatori viene mostrata la schermata di gioco. Al creatore della lobby viene assegnato il primo turno, seguendo poi l'ordine di connessione per i successivi. Vengono poi infine mischiati tutti i dadi e mostrati ai rispettivi giocatori.

- *Come si svolge un turno?*

  Il giocatore che inizia il turno può solo effettuare la prima supposizione mentre i giocatori che seguono possono scegliere se rilanciare oppure dubitare. Al primo giocatore del turno non è pertanto ammesso dubitare in quanto non è ancora presente nessuna supposizione.

- *Come si possono usare i dadi jolly?*

  I dadi jolly assumono sempre il valore dell'ultima puntata, a meno di casi particolari (vedi "turni particolari" sotto). E' possibile convertire una scommessa in dadi jolly rilanciando un numero di dadi jolly pari alla metà più uno del numero di dadi dell'ultima supposizione. E' sempre possibile inoltre ritornare ai dadi normali rilanciando con un numero pari al doppio più uno dei dadi jolly.

- *Ci sono restrizioni per fare una supposizione o un rilancio?*

  Il primo giocatore del turno può puntare un qualsiasi numero di dadi di qualsiasi valore, mentre tutti gli altri giocatori, se vorranno rilanciare, dovranno aumentare o il valore dei dati o il loro numero oppure convertire in dadi jolly, a meno di casi particolari (vedi "turni particolari" sotto).

- *Cosa succede a seguito di un rilancio?*

  A seguito di un rilancio il turno viene passato al giocatore successivo, il quale a sua volta più decidere se rilanciare nuovamente o dubitare. Il valore del rilancio (numero e valore dei dadi) viene mostrato a video a tutti i giocatori.

- *Cosa succede a seguito di un dubito?*

  Nel momento in cui un giocatore sceglie di dubitare viene verificata la veridicità dell'ultima supposizione. Se quest'ultima risulta vera il giocatore che ha dubitato perde un dado mentre in caso contrario lo perde il giocatore che ha fatto la supposizione. A tutti i giocatori, inclusi i due coinvolti, vengono mostrati tutti i dadi in gioco, in modo che possa essere verificata anche visivamente la scommessa. Per iniziare un nuovo turno tutti i giocatori devono confermare di aver visto i dadi.

- *Sono presenti turni particolari?*

  Quando uno dei giocatori rimane con un solo dado si dichiara "palifico". Nel turno successivo, definito "turno palifico", il valore del dado definito dalla prima scommessa non può essere modificato, è pertanto possibile solo aumentare il numero di dadi o convertirli in dadi jolly oppure dubitare. Inoltre i dadi jolly non assumono più il valore dell'ultima puntata, ma vengono contati a parte.

- *Cosa succede se un giocatore non prende una scelta?*

  Se il giocatore non effettua nessuna scelta entro 60 secondi il sistema effettua autonomamente la puntata minima. Pertanto se il giocatore è il primo del turno verrà puntato un dado di valore due mentre se il giocatore è successivo verrà mantenuto il valore del dado ed incrementato di uno il numero.

- *Cosa succede se uno o più giocatori non confermano la visione dei dadi dopo un dubito?*

  Se dopo 30 secondi uno o più giocatori non confermano la visione di tutti i dadi in gioco il sistema provvede alla conferma in autonomia, permettendo al gioco di andare avanti, grazie all'inizio di un nuovo turno.

- *Cosa succede quando un giocatore perde tutti i dadi?*

  Il giocatore che finisce i suoi dadi perde la partita ed ha la possibilità di rimanere all'interno della lobby, continuando a vedere le puntante degli altri giocatori, oppure di abbandonarla, in quest'ultimo caso verrà portato alla schermata iniziale. Per gli altri giocatori rimasti il gioco continua normalmente.

- *Quando termina la partita? Cosa succede quando termina?*

  La partita termina quando rimane un solo giocatore con almeno un dado a

disposizione. Al giocatore che ha vinto viene notificata la vittoria mentre a tutti gli alti la sconfitta. Tutti i giocatori vengono poi riportati alla schermata iniziale.

- *Cosa succede se un giocatore crasha o chiude l'applicazione a partita in corso?*

  Se un giocatore abbandona la partita per un qualsiasi motivo, quest'ultima viene immediatamente terminata notificando tutti i giocatori e riportandoli alla schermata iniziale. Essendo le scommesse effettuate su tutti i dadi di gioco dei giocatori ancora in partita, al mancare di uno o più giocatori quest'ultime risulterebbero invalide.

- *Cosa succede se il server crasha?*

  Se il server crasha, indipendentemente da quando ciò accade, a tutti i giocatori connessi in quel momento viene mostrato un popup di errore e l'applicazione viene chiusa.

1.2 Non-functional requirements

Ai requisiti funzionali sopra descritti, si aggiungono i seguenti requisiti non funzionali:

- L'applicazione deve essere semplice da utilizzare e l'interfaccia grafica deve essere intuitiva.
- Gli utenti che appartengono ad una stessa partita devono poter visualizzare lo stato di tutti gli altri giocatori, pertanto ne devono poter vedere il numero di dadi rimasti e la loro ultima scommessa.
- A tutti gli utenti all'interno di una stessa partita deve essere data la possibilità di verificare visivamente il risultato dei dubito effettuati dai vari giocatori.
- L'architettura scelta deve permettere all'applicazione di essere scalabile, pertanto di poter gestire più partite in contemporanea senza che ognuna interferisca in alcun modo con le altre.
- Casi di disconnessione improvvisa degli utenti o crash del server devono essere gestiti nella maniera più consona.

1.3 Scenarios

Di seguito sono descritti gli scenari in cui gli utenti si possono trovare all'interno dell'applicazione.

1.3.1 Pre-match scenario

Avviata l'applicazione, un utente può scegliere se creare una nuova lobby o se unirsi ad una già esistente, selezionandola dalla lista presentata nella schermata iniziale. Nel primo caso l'utente dovrà specificare il nome della lobby, la sua capienza e il suo username mentre nel secondo caso basterà indicare solo lo username. L'utente che ha creato la lobby può poi decidere di avviare la partita se la lobby è stata riempita.

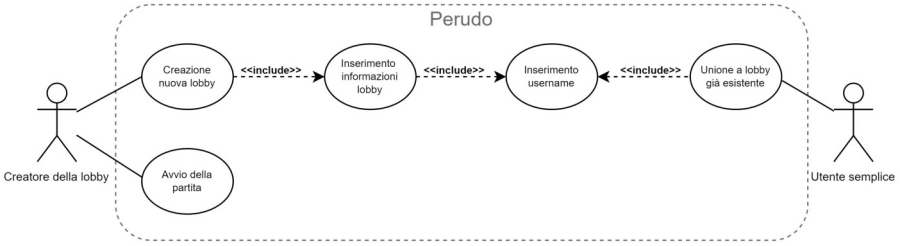


Figure 1.1: Pre-match use case diagram

1.3.2 During-match scenario

Una volta iniziata la partita, e quindi mischiati tutti i dadi, il primo giocatore di turno può effettuare la prima supposizione. Entro la fine del turno il giocatore può quindi selezionare il numero ed il valore dei dadi per comporre la scommessa. I giocatori successivi oltre a poter rilanciare la scommessa possono anche dubitare.
Tutti gli altri giocatori non di turno non hanno la possibilità di interagire con il turno corrente ma solo di visualizzare lo stato generale del gioco, cioè dal numero di dadi rimasti agli altri giocatori e l'ultima scommessa di quest'ultimi. Nel caso in cui un giocatore all'interno del proprio turno dubiti verranno mostrati a tutti gli altri giocatori ancora in gioco l'insieme dei dadi, ogni giocatore dovrà poi confermarne la visione.
Infine, l'ultimo scenario possibile si ha quando un giocatore perde tutti i dadi e quindi di conseguenza viene eliminato, in quel caso gli verrà data la possibilità di lasciare il gioco, non potrà interagire in alcun altro modo con la partita.

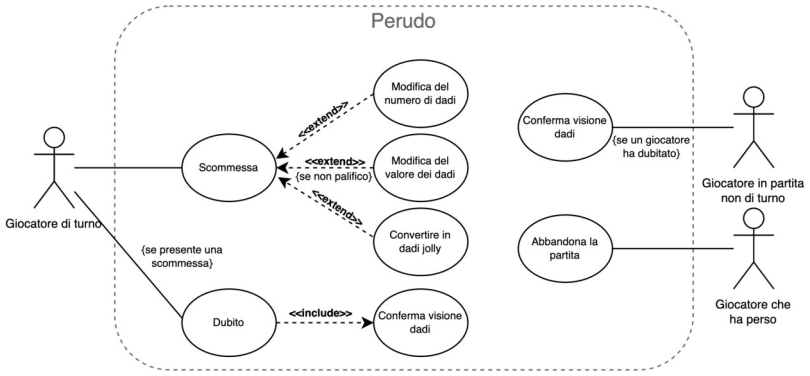


Figure 1.2: During-match use case diagrama

1.3.3 Post-match scenario

A fine partita viene mostrato a tutti i giocatori ancora all'interno della lobby un popup che indica loro se hanno perso o vinto la partita. Ogni giocatore, a seguito della conferma del seguente messaggio, viene riportato alla schermata inziale.

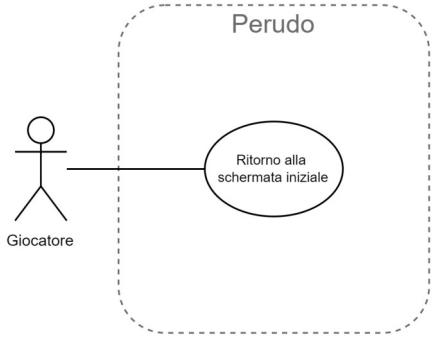


Figure 1.3: End-match use case diagrama

1.4 Self-assessment policy

Il collaudo dell'applicazione sarà svolto tramite una serie di test automatici, realizzati attraverso il fremework JUnit. I test verificheranno che le interazioni tra i vari client ed il server avvengano come previsto e copriranno tutti gli aspetti rilevanti dell'applicazione:

- Connettività
- Corretta gestione delle lobby di gioco

- Corretta gestione della partita

Inoltre, essendo l'applicazione puramente grafica, verranno effettuati test manuali su quest'ultima, per assicurarne il corretto funzionamento.

Chapter 2

Design

2.1 Application structure

L'architettura scelta per questo progetto è di tipo *client-server*. Come descritto più nel dettaglio in seguito, la logica di gioco è gestita quasi interamente dal *server*: le uniche operazioni che il *client* svolge sono operazioni a livello locale che non impattano sugli altri giocatori. Di seguito uno schema della struttura implementata.

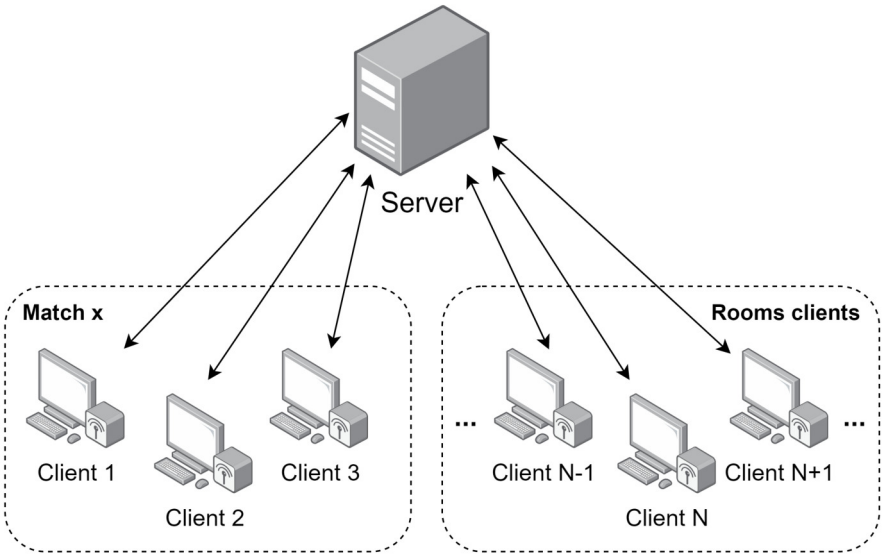


Figure 2.1: Architettura dell'applicazione

La comunicazione tra *client* e *server* avviene attraverso una **connessione TCP**: nello specifico il *server* mantiene una *socket* per ogni *client* che richiede

di connettersi. Tale implementazione consente una comunicazione bidirezionale tra *client* e *server*, in particolare si hanno le seguenti interazioni:

- i *client* inviano **richieste** al *server*.
- il *server* elabora la richiesta del *client* ed invia un messaggio di **risposta**.
- Talvolta il *server* invia messaggi al *client* senza che essi facciano una richiesta esplicita (**notifiche**). Le notifiche sono utilizzate sia per gestire aspetti grafici, ad esempio aggiornare la lista delle lobby presenti o l'ultima scommessa di un giocatore, ma anche per gestire aspetti core, come ad esempio notificare l'inizio del turno o il termine della partita stessa.

E' presente inoltre un'ultima tipologia di interazione per informare i vari *client* connessi dello stato del *server*. In particolare il *server* ogni tot secondi, di default 3, invia un messaggio di tipo **server status** al *client*, quest'ultimo potrà pertanto capire in ogni momento se il *server* è ancora online oppure no, semplicemente controllando la ricezione di questi messaggi. In particolare se i *client* dopo un certo intervallo di tempo, di default 4 secondi, non ricevono nessun messaggio di tipo **server status** dal *server* notificano all'utente che il *server* non risponde e l'applicazione viene chiusa.

2.1.1 Client structure

Il *client* è stato organizzato seguendo il **pattern MVC**, in modo da poter suddividere con chiarezza le responsabilità delle varie parti:

- la **view** si occupa degli aspetti prettamente grafici.
- il **model** gestisce gli aspetti di comunicazione con il *server* oltre che mantenere e manipolare i vari dati
- il **controller** collega *view* e *model* aggiungendo un livello di astrazione

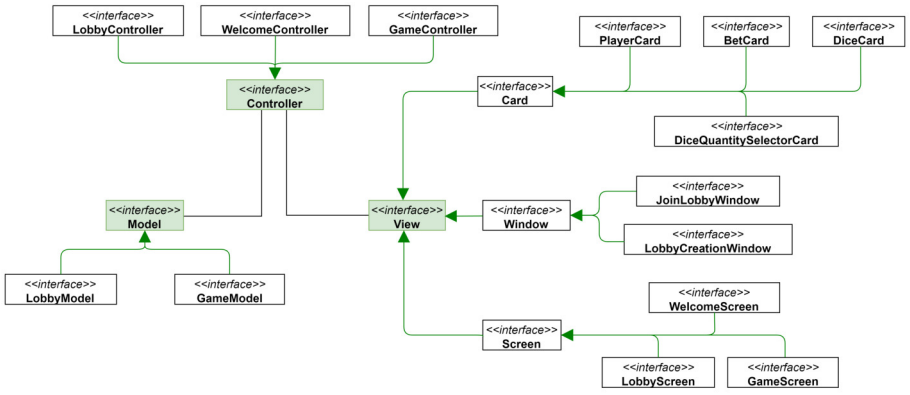


Figure 2.2: Architettura del client

View structure

L'interfaccia grafica è stata realizzata attraverso il framework *libgdx*[1]. Non essendo elemento centrale del progetto, non è stato dato troppo peso all'aspetto grafico dell'applicazione, che comunque risulta semplice ed utilizzabile fluidamente; La soluzione adottata tuttavia risulta essere piuttosto flessibile grazie al modo in cui è stata strutturata, e consente facilmente di migliorare l'aspetto dell'applicazione.
L'aspetto grafico è stato suddiviso in tre componenti principali, come visibile in figura 2.2:

- Gli **screen**, i quali rappresentano le schermate possibili dell'applicazione, in particolare:
  - **Welcome screen**, è la schermata che appare all'utente appena apre l'applicazione. Presenta un unico pulsante che permette a quest'ultimo di collegarsi al server ed accedere quindi all'interno dell'applicazione.
  - **Lobby screen**, è la schermata che appare all'utente appena la connessione con il server ha successo. In questa schermata viene mostrata la lista delle lobby già presenti e non ancora piene, oltre alla possibilità di creare una nuova lobby.
  - **Game screen**, è la schermata di gioco. Appare nel momento in cui il creatore della lobby, quando quest'ultima è piena, avvia la partita.
- Le **card**, utilizzate per rappresentare le aree di gioco, in particolare:
  - **Player card**, rappresenta l'area in cui viene mostrato il numero di dadi rimasti al giocatore e la sua ultima puntata. Ogni giocatore vede le *player card* di tutti gli altri.
  - **Bet card**, rappresenta l'area in cui il giocatore di turno può configurare la scommessa da fare.
  - **Dice card**, è l'area in cui il giocatore vede i suoi dadi e/o i dadi di tutti i giocatori nel momento in cui si ha un dubito.
  - **Dice quantity selector card**, è un classico selezionatore di numeri, inserito all'interno della *bet card* ed utilizzato per configurare la scommessa.
- Le **window**, utilizzate per creare delle schermate a popup, in particolare:
  - **Lobby creation window**, appare nel momento in cui un utente decide di creare una nuova *lobby*, richiede a quest'ultimo di specificare il nome della *lobby*, la sua capienza e l'username che lo caratterizzerà durante la partita.
  - **Join lobby window**, appare nel momento in cui un utente decide di entrare in una *lobby* già esistente e richiede all'utente di specificare lo username da utilizzare all'interno della partita.

[1] `https://libgdx.com/`

I comportamenti dei tre *screen* sopra elencati sono gestiti rispettivamente dai corrispondenti *controller* e *model*, in particolare il comportamento del *lobby screen* è gestito dal **LobbyController** e dal **LobbyModel** mentre quello del *game screen* è gestito dal **GameController** e dal **GameModel**. Il *welcome screen* è gestito solo dal *controller* (**WelcomeController**) in quanto troppo semplice per richiedere un *model*.

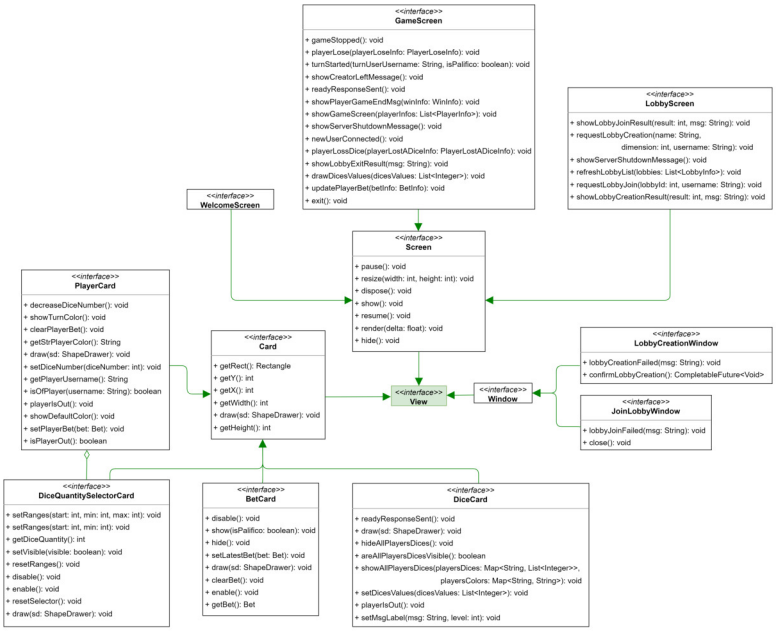


Figure 2.3: Diagramma delle classi della view

Model structure

Come precedentemente indicato il *model* mantiene i dati e lo stato locale dello *screen* ad esso associato. Ha il compito, in seguito ad invocazioni provenienti dal *controller* o da notifiche proveniente dal server, di aggiornare il suo stato e/o la *view*. Il *model* ha inoltre il compito di effettuare le richieste al server, per farlo si serve del **CommunicationManager**, il quale verrà presentato in maggior dettaglio in seguito.

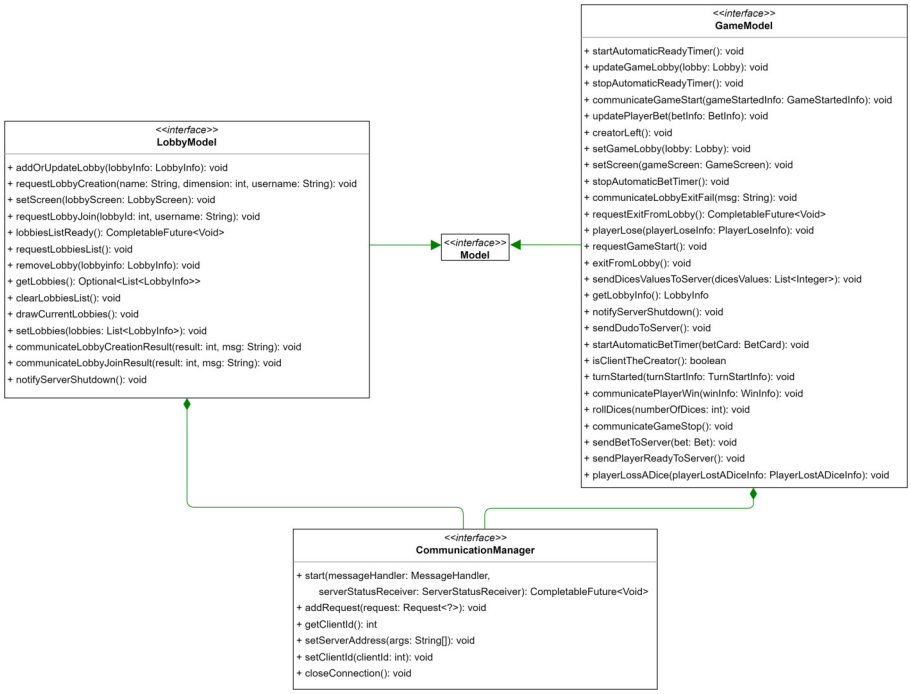


Figure 2.4: Diagramma delle classi del model

Controller structure

Il *controller* infine collega il *model* con lo *screen*, in particolare riceve gli input dall'utente, espressi da quest'ultimo sulla GUI, e li comunica al *model*, che avrà il compito di gestirli nella maniera adeguata.

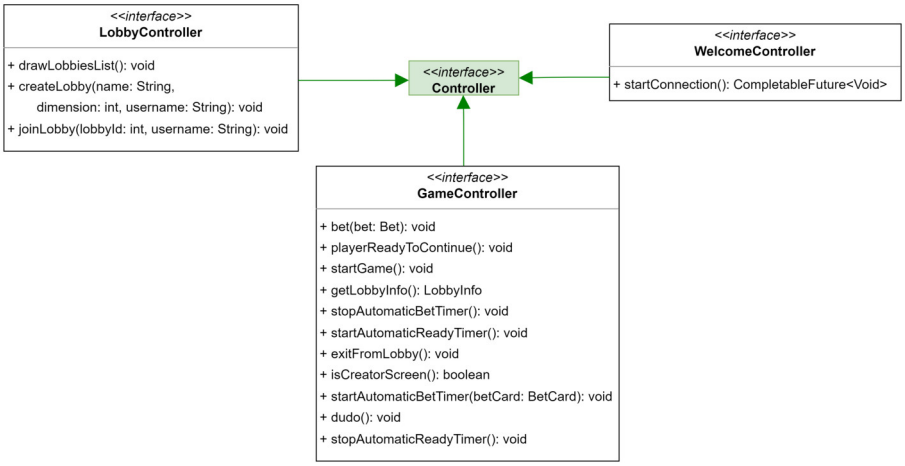


Figure 2.5: Diagramma delle classi del controller

Connection management

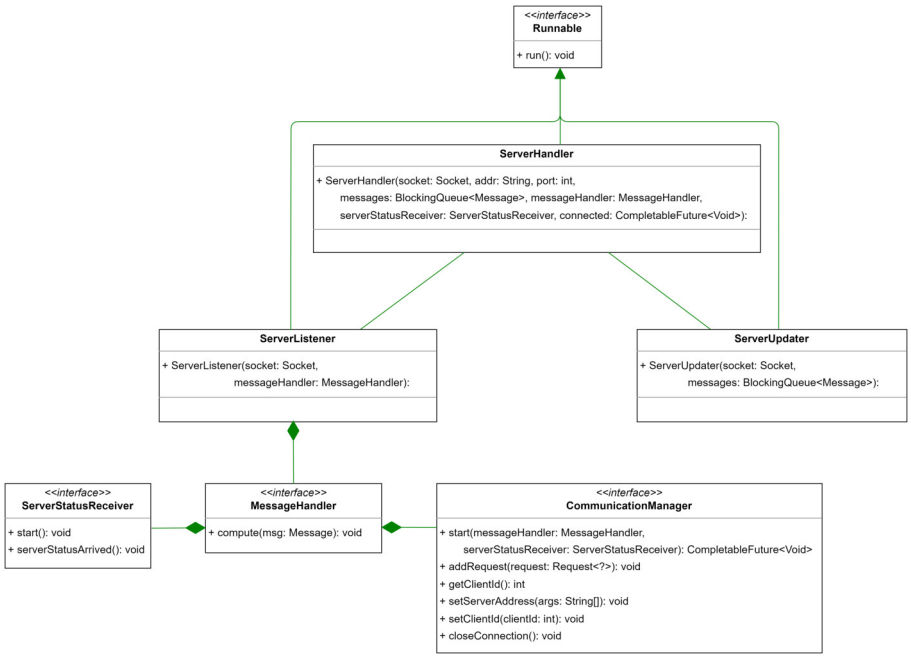


Figure 2.6: Architettura di gestione della connessione

Come accennato in precedenza la connessione e la comunicazione con il *server* sono gestiti dal **CommunicationManager**. Quest'ultimo dispone di un metodo **start** che permette di istanziare una nuova connessione con il server, per farlo crea un nuovo **ServerHandler**. Dopo aver istanziato la connessione con l'indirizzo e la porta specifice nel costruttore, il **ServerHandler** procede avviando l'esecuzione di un **ServerListener** e di un **ServerUpdater** e termina la propia esecuzione.
Il metodo **start** appena descritto ritorna una **CompletableFuture** in modo da comunicare eventuali errori durante la creazioni della connessione, ad esempio se il *server* dovesse risultare irraggiungibile, alla *view* e quindi di coseguenza all'utente.
I due stream della *socket* (input e output) sono pertanto gestiti da due diversi flussi d'esecuzione, permettendo al *client* di inviare un messaggio al *server* e allo stesso tempo di rimanere in ascolto di eventuali messaggi in ingresso.
Il **ServerUpdater** è l'entità che si occupa di inviare le richieste al *server*. Quest'ultime vengono prelevate da una coda bloccante **messages**, serializzate attraverso un serializzatore GSON[2], ed infine inviate al *server*. Ad inserire nuove richieste nella coda dei messaggi sono i *model* precedentemente descritti, utilizzando il metodo **addRequest** del **CommunicationManager**. Gestendo le richieste in questo modo si è sempre sicuri che il corretto ordine temporale sarà rispettato.
Il **ServerListener** si pone invece in ascolto di messaggi provenienti dal *server* e per ogni nuovo messaggio ricevuto, invoca il metodo **compute** del **MessageHandler**.
Il **MessageHander** per certi versi pertanto è l'elemento centrale del *client*: esso si occupa a tutti gli effetti di reagire ai messaggi proveniente dal *server*, siano essi risposte a richieste effettuate in precedenza o notifiche.
Come precedentemente indicato è inoltre presente un **ServerStatusReceiver** che ogni tot secondi, 4 di default, controlla che il server invii un messaggio di tipo **ServerStatus** per accertarsi che quest'ultimo sia online. Nel caso in cui il flusso dei seguenti messaggi cessi l'utente verrà notificato del crash del *server* e l'applicazione verrà chiusa.

2.1.2 Server structure

Model structure

Il diagramma delle classi mostrato in figura 2.7 riassume la struttura del *model*.
L'elemento centrale di questa architettura è l'entità **Game**. Ogni **Game** mantiene tutte le informazioni utili alla gestione di una singola partita, dal momento in cui viene creata, fino a quando non termina. All'interno di ogni **Game** viene mantenuta l'istanza della **Lobby** ad esso associato, per permettere a quest'ultimo di ottenere informazioni indispensabili per la gestione della partita, come ad esempio la lista di utenti in gioco o l'ordine di connessione di quest'ultimi, utilizzato per gestire i turni.
Infine l'oggetto **Game** è composto dall'oggetto **Bet**, che rappresenta una scommessa.

[2] `https://github.com/google/gson`

Quest'ultimo viene utilizzato per memorizzare l'ultima scommessa effettuta, quella che verrà poi controllata in caso di *dudo* da parte di un giocatore.
I diversi **Game** vengono gestiti dal **GameManager**. All'interno del *server* è presente una sola istanza di questa classe. Attraverso questa entità è possibile creare o rimuovere partite oltre che ottenere un riferimento ad una specifica partita utilizzando un determinato identificatore, come ad esempio l'id dell'utente, la sua socket oppure la lobby di appartenenza del **Game** stesso.

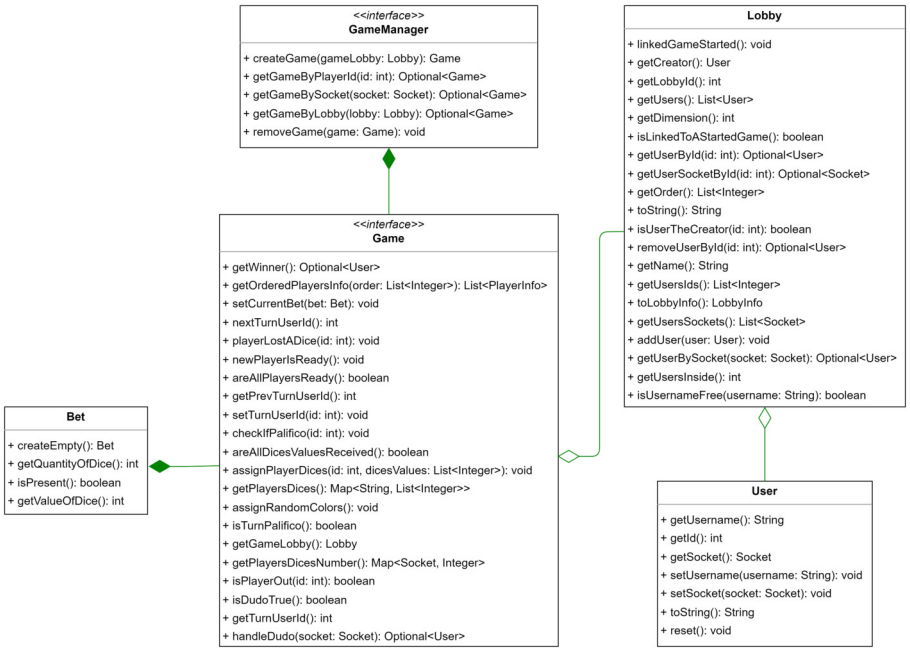


Figure 2.7: Model del server

Connection management

Lato *server*, le connessioni sono gestite in maniera molto simile a quanto avviene nei *client*. Quando viene invocato il metodo **start** del **PerudoServer** viene istanziato un nuovo **ServerSocket**, utilizzando la porta specificata nel costruttore. A questo punto il *server* si pone in ascolto di nuove connessioni attraverso il metodo **accept** del **ServerSocket**. La gestione di ogni connessione così creata è delegata al **ClientHandler**, il quale crea a sua volta un **ClientListener** ed un **ClientUpdater**, che rispettivamente operano in maniera analoga al **ServerListener** e al **ServerUpdater** descritti in precedenza: il **ClientListener** è in ascolto di nuove richieste che saranno inserite in una coda bloccante, mentre il **ClientUpdater** prevela ed elabora i messaggi, inviando le risposte/notifiche al *client*.
Il corrispettivo del **MessageHandler** lato *client* per il *server* è il **RequestHandler**.

Quest'ultimo mette a disposizione un metodo **handle** per gestire le richieste provenienti dai *client*, ritornando il rispettivo risultato, che verrà poi inviato al *client* interessato dal **ClientUpdater**. Il **MessageHandler** per gestire le varie richieste, si serve sia del **GameManager**, visto in precedenza, che del **LobbyManager**. Quest'ultimo permette di gestire tutte le connessioni dei vari *client* al *server*, sia che quest'ultimi siano all'interno di una lobby oppure nella room principale.
Infine il **MessageHandler** si serve dell'**IdGenerator** per generare degli identificativi univoci sia per i *client* che si connettono che per le nuove lobby che vengono create. Non viene generato un identificativo per il **Game** in quanto esso è associato ad una ed una sola lobby, che lo identifica in modo univoco.

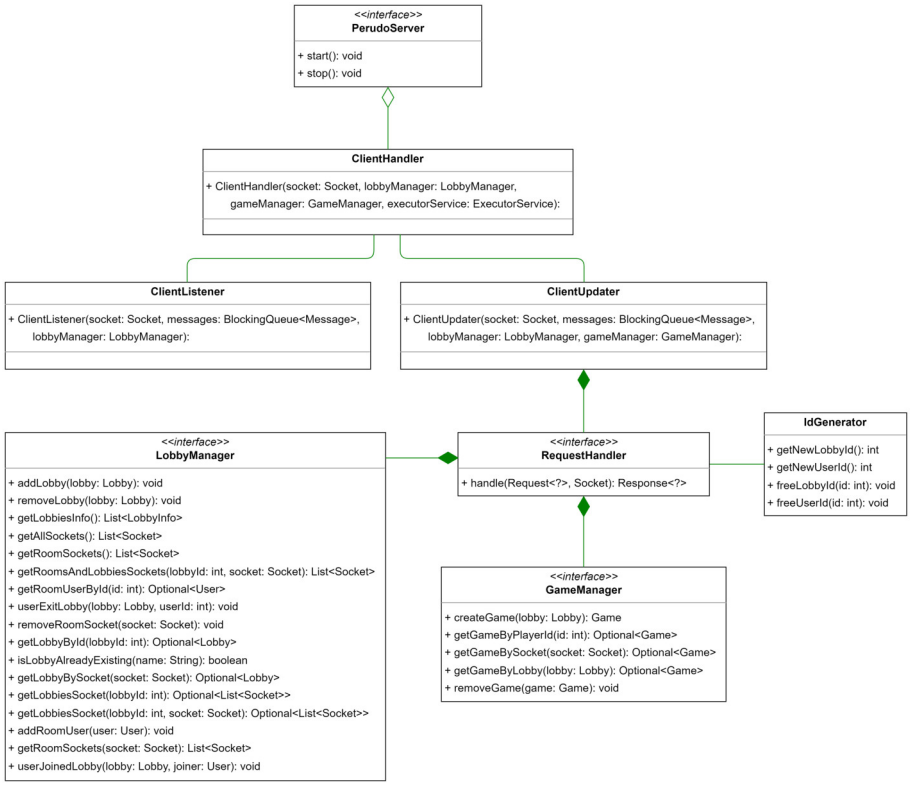


Figure 2.8: Gestione delle connessioni lato server

2.1.3 Messages structure

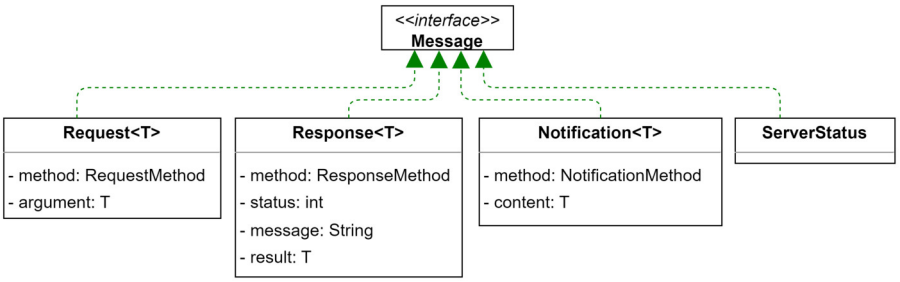


Figure 2.9: Struttura dei messaggi

All'interno dell'applicazione possono essere distinte tre diverse tipologie di messaggi, ciascuna delle quali implementa l'interfaccia **Message**. Tutti i messaggi, tranne **ServerStatus**, contengono un campo **method**, il cui tipo varia a seconda del tipo di messaggio. Questo campo è fondamentale per la deserializzazione del messaggio stesso, come si può osservare in figura 2.10

```
public class MessageDeserializer implements JsonDeserializer<Message> {
    @Override
    public Message deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context) throws JsonParseException {
        JsonObject jsonObject = json.getAsJsonObject();

        JsonElement methodTypeElement = jsonObject.get("method");
        if (methodTypeElement != null) {
            String methodType = methodTypeElement.getAsString();
            if (EnumUtils.enumContains(RequestMethod.values(), methodType)) {
                return context.deserialize(json, Request.class);
            }
            if (EnumUtils.enumContains(ResponseMethod.values(), methodType)) {
                return context.deserialize(json, Response.class);
            }
            if (EnumUtils.enumContains(NotificationMethod.values(), methodType)) {
                return context.deserialize(json, Notification.class);
            }
        }
        return context.deserialize(json, ServerStatus.class);
    }
}
```

Figure 2.10: Deserializzatore json per i messaggi

Segue una descrizione più dettagliata per le varie tipologie di messaggi:

- Le richieste (**Request**) sono semplici messaggi che viaggiano dal *client* al *server*. Esempi di richieste sono la **CreateLobbyRequest** e la **StartGameRequest**, rispettivamente utilizzate per richiedere la creazione di una nuova lobby e l'inizio di una partita. Il campo generico *argument* permette di specificare informazioni utili da spedire insieme al messaggio, come ad esempio le specifiche della lobby da creare.

- Le risposte (**Response**) sono messaggi che viaggiano dal *server* al *client*. Ogni **Response** è sempre generata dalla gestione di una **Request**. Oltre ad indicare metodo e risultato, le risposte contengono uno *status*, utile per capire se la richiesta è andata a buon fine, ed un messaggio testuale. le prima citate **CreateLobbyRequest** e la **StartGameRequest**, dopo essere state elaborate dal *server*, producono come risposta rispettivamente una **CreateLobbyResponse** e una **StartGameResponse**.

- Come le risposte, anche le notifiche (**Notification**) sono messaggi che viaggiano dal *server* al *client*, ma a differenza di quest'ultime non sono originati da una richiesta ma sono inviati dal *server* al verificarsi determinati eventi, come ad esempio quando un giocatore perde un dado (**PlayerLostADiceNotification**) o la vincita di un giocatore (**PlayerWinNotification**).

- L'ultima tipologia di messaggi (**ServerStatus**) viaggia dal *server* a tutti i *client* connessi, con lo scopo di comunicare a quest'ultimi il suo stato. Nello specifico, come già precedentemente indicato, i *client* rimangono in ascolto di questo flusso di messaggi, inviati dal *server* con una determinata cadenza; Nel caso in cui tale flusso dovesse terminare i *client* si accorgerebbero immediatamente dell'irraggiungibilità del *server*, notificandolo all'utente e chiudendo l'applicazione.

Nell'immagine che segue sono indicati tutti metodi dei vari messaggi elencati:

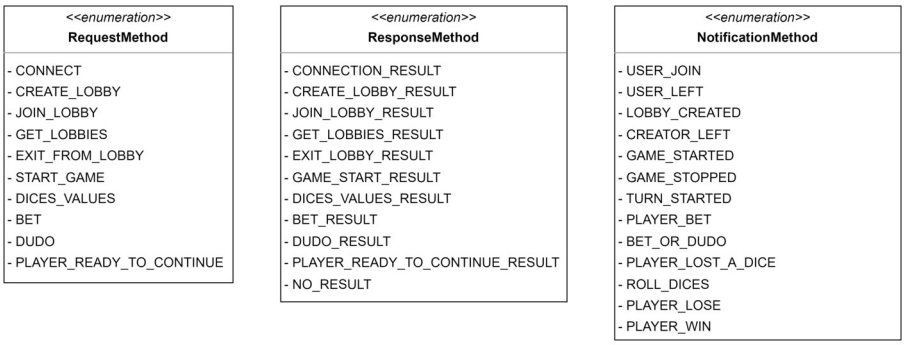


Figure 2.11: Metodi per tipologia di messaggio

2.2 Behaviour

Il diagramma degli stati in figura 2.12 mostra il comportamento dell'applicazione dal punto di vista di un utente.

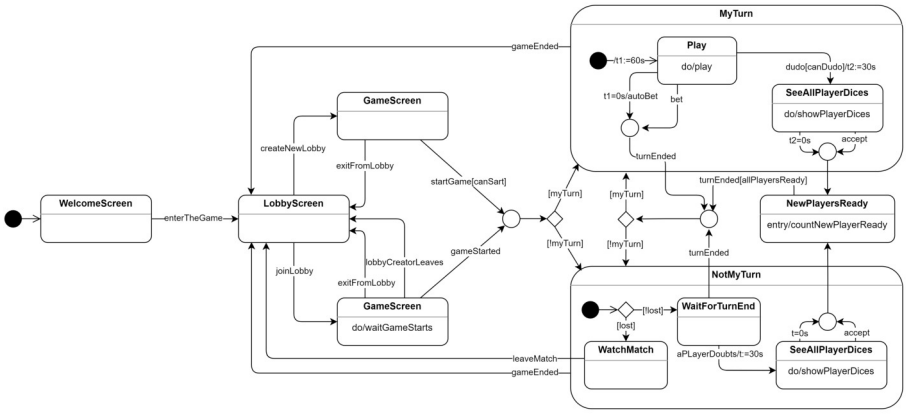


Figure 2.12: Metodi per tipologia di messaggio

Quando l'applicazione viene avviata l'utente si trova nel menù iniziale. Qui l'utente può scegliere di entrare all'interno del gioco oppure uscirne, chiudendo direttamente l'applicazione.
Entrato nell'applicazione all'utente viene mostrato il menù principale, nel quale può scegliere di creare una nuova lobby o di unirsi ad una lobby già esistente.
I giocatori che si sono uniti ad una lobby attendono a questo punto l'inizio della partita che può essere determinato solamente dal creatore della lobby e soltanto quando quest'ultima è piena. Ogni utente all'interno della lobby ha la possibilità di ritornare al menù principale prima che la partita inizi. Nel caso in cui sia l'utente che ha creato la lobby ad uscire, tutti gli utenti connessi vengono trasferiti al menù iniziale e la lobby viene eliminata.
Una volta avviata la partita è deciso l'ordine di gioco. Il giocatore di turno può decidere se scommettere o dubitare, ove possibile, mentre i giocatori non di turno attendono il loro. Nel caso di scommessa il turno termina immediatamente mentre nel caso in cui il giocatore di turno dubiti vengono mostrati ad ogni giocatori tutti i dadi in partita. A questo punto tutti i giocatori hanno al massimo 30 secondi per osservare tali dadi e comunicare di essere pronti per l'inizio di un nuovo turno. Appena tutti i giocatori in gioco hanno comunicato la loro volontà di continuare, manualmente o automaticamente allo scadere dei 30 secondi, il turno termina.
Il giocatore di turno ha 60 secondi per decidere se scommettere o dubitare, ove possibile. Allo scadere dei 60 secondi, il sistema effettuerà in autonomia una scommessa, in particolare quella minima se il giocatore non ne ha impostata nessuna oppure quella scelta fino a quel momento dal giocatore ma non ancora confermata.
Tutti gli utenti all'interno di una partita in corso possono lasciare l'applicazione chiudendola, questo però determinerà la terminazione di quest'ultima ed il trasferimento di tutti i giocatori alla schermata iniziale.

2.3 Interaction

Il diagramma delle interazioni di figura 2.13 mostra cosa accade quando un utente richiede la creazione di una lobby ed un altro richiede di di unirsi alla medesima lobby.
Per ogni istanza attiva del *client*, il *server* crea un **ClientListener**, un **ClientUpdater** e un **RequestHandler**, che si occupano di gestire la connessione con lo specifico *client*.

- Il **ClientListener** riceve le richieste e le inoltra al **ClientUpdater**, inserendole in una coda bloccante condivisa tra entrambe le istanze.
- Il **ClientUpdater** risolve le richieste attraverso il **RequestHandler** ed invia le risposte al *client*

Come si può notare quando il **RequestHandler** del *client c1* gestisce la richiesta di creazione di una nuova lobby, oltre a creare un messaggio di risposta, produce anche una notifica che viene inviata al *client c2*. Così facendo, il *client c2* è a conoscenza del fatto che una nuova lobby è stata creata e può reagire di conseguenza aggiornando la lista di lobby possibili a cui può accedere. Lo stesso comportamento si può osservare a seguito dell'elaborazione, da parte del **RequestHandler** *c2*, della richiesta di *join* da parte del *client c2*. In particolare il **RequestHandler** *c2*, oltre a creare un messaggio di risposta, invia una notifica al *client c1* per informarlo che un nuovo utente si è connesso alla lobby. Nel caso fossero presenti più utenti all'interno della room principale, verrebbero anch'essi notificati in modo da aggiornare la loro lista con la giusta quantità di utenti connessi alle varie lobby.
Per facilitare la lettura, il diagramma in figura riporta alcune semplificazioni lato *client*. Ogni *client* dovrebbe essere infatti costituito da 3 entità: un **ServerUpdater**, un **ServerListener** ed un **MessageHandler** (in corrispondenza alle entità lato server *ClientUpdater*, *ClientListener* e *RequestHandler*).

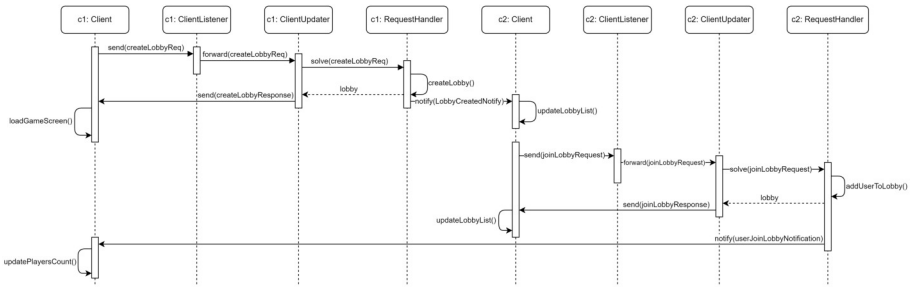


Figure 2.13: Rappresentazione semplificata di creazione e unione a una lobby

Chapter 3

Implementation Details

Il sistema è stato implementato sotto forma di progetto multi-modulo. I 4 moduli principali sono:

- **`client`**
- **`server`**
- **`common`**
- **`test`**

Il modulo **`test`** è stato utilizzato per testare l'interazione tra le varie componenti del sistema.
L'intero sistema è stato documentato attraverso l'uso di *dokka*[1]. La documentazione così prodotta è disponibile e navigabile al seguente link `https://enricolumini.github.io/PerudoOnlineDoc`.

3.1 Client

Il *client* è stato implementato seguendo il pattern *MVC*. L'interfaccia grafica è stata implementata attraverso l'utilizzo del framework *libgdx*[2]. Il *model* ha il compito di gestire la connessione con il *server* oltre che ad aggiornare e mantenere lo stato della *view* a cui è collegato. Il *controller* ha il compito di collegare la *view* al *model*, trasferendo gli input dell'utente dall'interfaccia grafica al *model* interessato, in modo che quest'ultimo li possa elaborare.

3.2 Server

Il *server* contiene il vero e proprio *model* del sistema. Le informazioni mantenute localmente dai vari *client* sono infatti soltanto informazioni ausiliarie ed il *server*

[1] `https://github.com/Kotlin/dokka`
[2] `https://libgdx.com/`

è l'unica fonte di verità all'interno del sistema. In questo modo si ha un maggiore controllo sulle operazioni effettuate dagli utenti, eliminando la possibilità di operazioni illegali o non autorizzate.

3.3 Common

Il modulo *common* contiene una serie di elementi utili sia al *client* che al *server*, come ad esempio la definizione dei vari messaggi possibili e le informazioni ad essi associati.

Chapter 4

Validation

La fase di verifica si è composta sia di **test manuali**, atti a verificare la correttezza della componente grafica, sia di **test automatici** realizzati attraverso il framework *Junit*[1]. In figura 4.1 sono mostrati i test implementati.

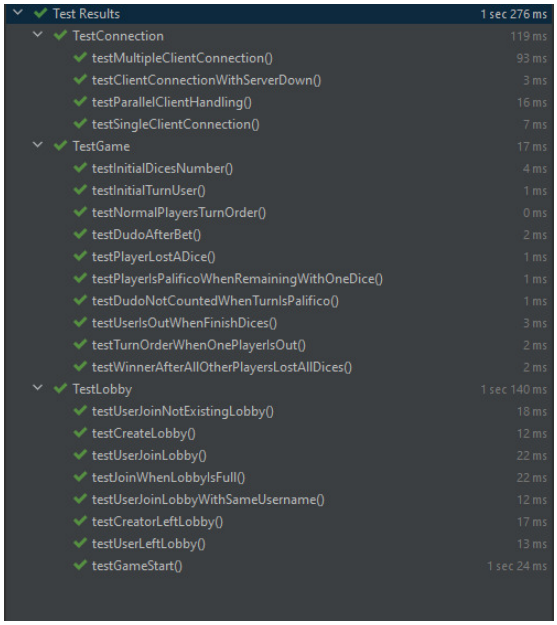


Figure 4.1: test Junit

Per onorare la pratica della **Continuous Integration**, sono state sfruttate le *CI/CD Pipelines* di *GitLab*[2]. Nel dettaglio, ad ogni push di un commit su

[1] `https://junit.org/junit5/`
[2] `https://docs.gitlab.com/ee/ci/pipelines/`

GitLab, viene creata una nuova pipeline seguendo le istruzioni riportate nel file .gitlab-ci.yml, riportato in figura 4.2. In particolare viene istanziato un nuovo container, sul quale vengono eseguiti tutti i test precedentemente riportati. Nei test in cui deve essere verificata la connettività *client/server* le varie istanze vengono create e gestite direttamente dai test stessi.

```
image: gradle:latest

build:
  stage: build
  script: gradle classes testClasses

test:
  stage: test
  script: gradle test
  artifacts:
    when: always
    reports:
      junit: '**/build/test-results/test/**/TEST-*.xml'
```

Figure 4.2: File .gitlab-ci.yml

Chapter 5

Deployment

Essendo il progetto un applicativo *Java*, l'unico prerequisito per poterlo eseguire è che la macchina che lo esegue sia dotata di una JVM.
Per eseguire il *server* è necessario collocarsi con un terminale all'interno della radice del progetto e lanciare il seguente comando:

```
$ ./gradlew server:run
```

Il *server* così lanciato accetterà connessioni sulla porta 4444. Qualora questa porta sia già occupata o si voglia per un qualsiasi motivo modificare questo valore, è possibile farlo specificando il numero di porta nel comando precedentemente riportato:

```
$ ./gradlew server:run -pport=`port`
```

L'avvio di un *client* risulta molto simile:

```
$ ./gradlew client:run
```

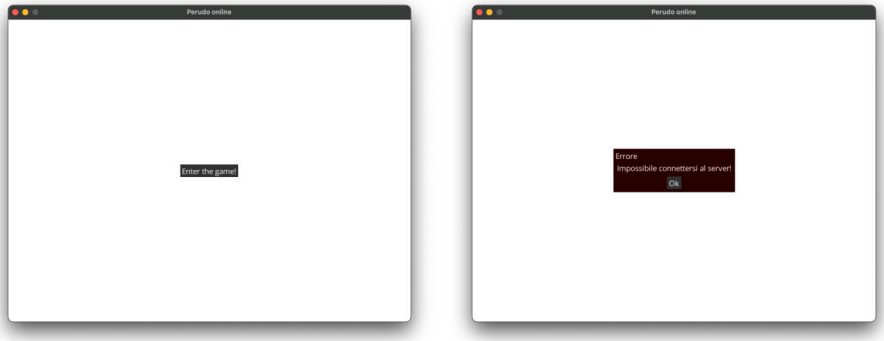
Di default, il *client* tenterà di connettersi alla porta 4444 del *localhost*; tuttavia è possibile specificare sia l'indirizzo che la porta verso cui effettuare la connessione attraverso il seguente comando:

```
$ ./gradlew client:run -phost=`host` -pport=`port`
```

Chapter 6

Usage Examples

Dopo aver avviato un'istanza del *server* è possibile eseguire i *client*. Nella schermata iniziale verrà mostrata la possibilità di connettersi al *server*. Nel caso in cui quest'ultimo non sia raggiungibile verrà mostrato il messaggio in fig 6.1b.



(a) Schermata inziale (b) *server* offline

Figure 6.1: Schermata iniziale

Se il *server* è stato avviato e la connessione è avvenuta con successo all'utente viene mostrato il menù principale, dal quale può scegliere se creare una nuova lobby o unirsi ad una già esistente, come si può osservare in figura 6.2.

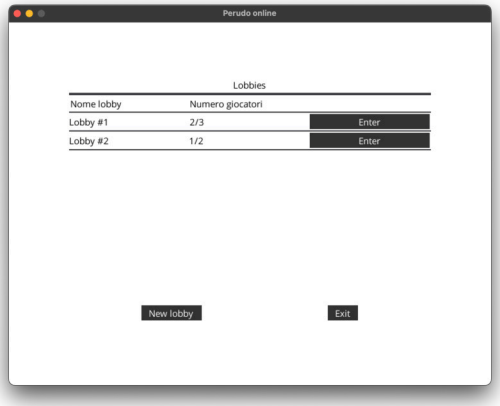
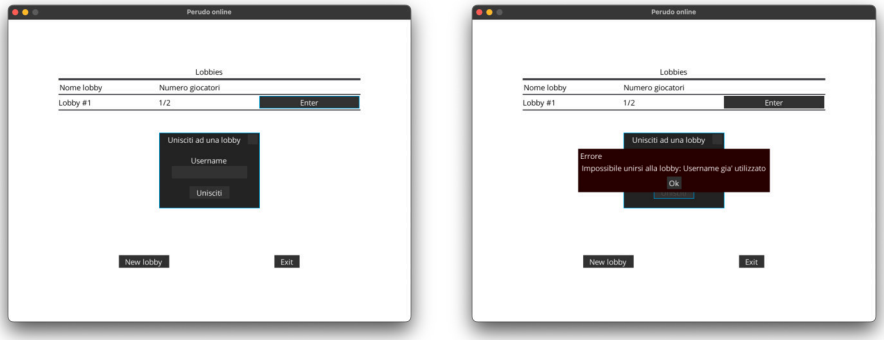


Figure 6.2: Menù principale con due lobby presenti

Se un utente decide di creare una nuova lobby gli viene mostrata la schermata in figura 6.3, nella quale deve indicare il nome e la capienza della lobby oltre allo username che lo identificherà all'interno della partita.

Figure 6.3: Creazione di una lobby

Se invece l'utente decide di unirsi ad una lobby già esistente gli verrà mostrata la schermata in figura 6.4a nella quale deve indicare lo username che lo identificherà all'interno della partita. Se lo username inserito è già stato utilizzato all'interno della lobby all'utente viene mostrata la schermata in figura 6.4b costringendolo ad inserirne uno nuovo per completare l'unione alla lobby.



(a) Unione ad una lobby (b) Username già utilizzato

Figure 6.4: Schermata di unione ad una lobby

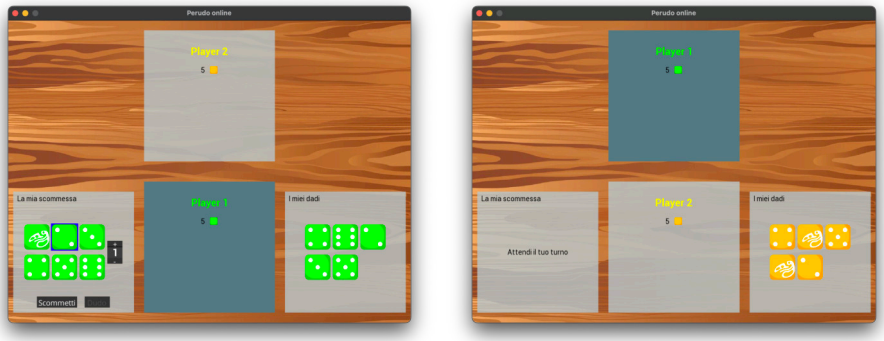
A seguito della creazione di una nuova lobby o all'unione ad una lobby già esistente agli utenti viene mostrata la schermata di gioco. In particolare al creatore della lobby viene mostrata la schermata in figura 6.5a mentre ad un utente che si è unito la schermata in figura 6.5b. Come è possibile osservare in figura 6.5a, il creatore ha la possibilità di far partire la partita, solo nel momento in cui la lobby è piena. Nel caso in cui la lobby non sia stata ancora riempita il pulsante "*Start*" è disabilitato. Tutti gli utenti possono sempre uscire, prima che la partita venga avvita, utilizzando il pulsante "*exit*".

(a) Schermata creatore (b) Schermata un utente normale

Figure 6.5: Schermata di gioco

A seguito dell'avvio della partita da parte del creatore della lobby a tutti i giocatori in partita viene presentata il tabellone di gioco. Nell'immagine 6.6a è presentata la schermata dell'utente di turno mentre in figura 6.6b quella dell'utente non di turno. Come si può osservare nella scheda di destra vengono mostrati i propri dadi, 5 inizialmente, mentre nella schermata di sinistra l'utente di turno

ha la possibilità di configurare la scommessa o dubitare, se possibile. Nelle card centrali sono disponibili le informazioni sui giocatori in partita, come il nome, il numero di dadi rimasti e l'ultima scommessa effettuata, non ancora presente nelle immagini sottostanti in quanto la partita è appena iniziata.



(a) Giocatore di turno (b) Giocatore non di turno

Figure 6.6: Schermata di inizio partita

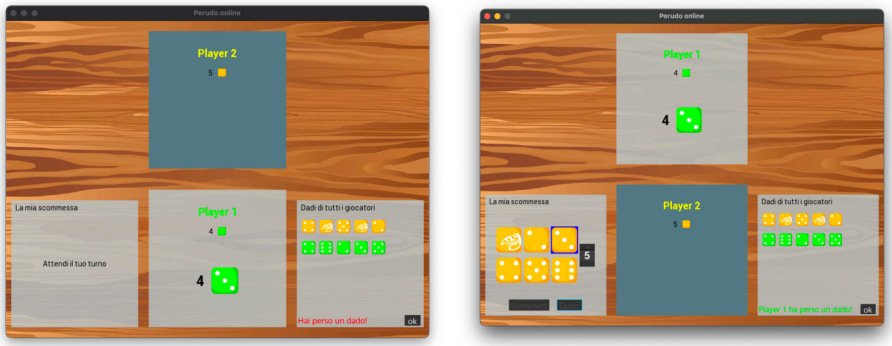
Dopo che l'utente ha fatto la sua scelta il turno viene passato al giocatore successivo. In figura 6.7b e 6.7a sono presentate le schermate mostrate ai due utenti in gioco dopo che il giocatore del turno procedente ha effettuato la prima scommessa. Come si può osservare il giocatore "*Player 2*" è ora il giocatore di turno.

(a) Giocatore non di turno (b) Giocatore di turno

Figure 6.7: Schermata di gioco a seguito di una scommessa

A questo punto ad ogni rilancio della scommessa il turno viene passato di giocatore in giocatore, fino a che un utente non dubita. A questo punto vengono mostrati a tutti gli utenti i dadi di tutti i giocatori ancora in partita ed il gioca-

tore che ha sbagliato perde un dado, secondo le regole presentate in precedenza. Nelle immagini che seguono sono presentate le schermate visibili rispettivamente all'utente che perde un dado, immagine 6.8a, ed all'utente che vince il dubito, immagine 6.8b. Il turno riparte non appena tutti i giocatori confermano la visione dei dadi.

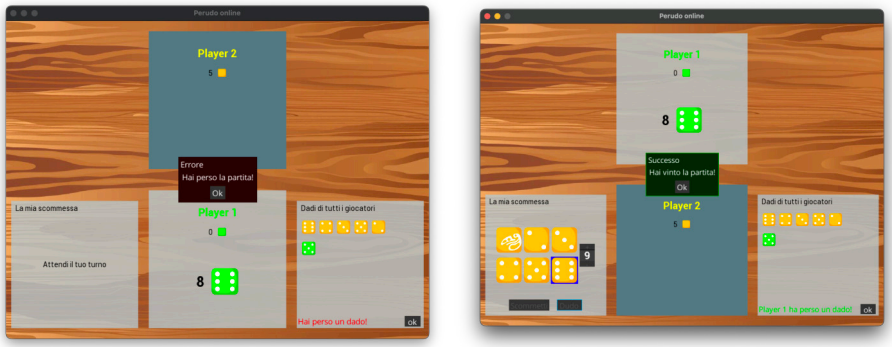


(a) Giocatore perde un dado

(b) Giocatore vince il dubito

Figure 6.8: Schermata di gioco a seguito di un dubito

La partita prosegue nello stesso modo fino a che non rimane un solo giocatore con almeno un dado, in questo caso viene mostrato un messaggio a tutti i giocatori indicando se hanno vinto o perso la partita, come si può osservare in figura 6.9.



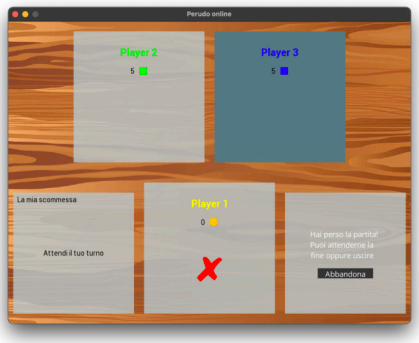
(a) Perdita della partita

(b) Vincita della partita

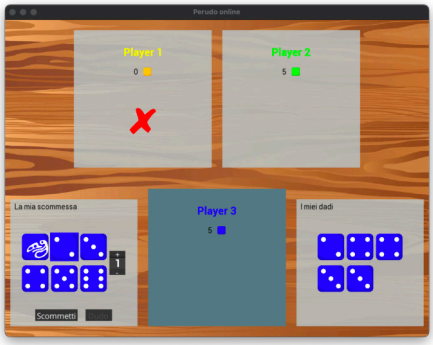
Figure 6.9: Schermata di gioco al termine della partita

Nel caso in cui la partita sia giocata da più di due giocatori ed uno di essi perda tutti i suoi dadi, lasciando ancora in gioco un numero di giocatori superiore o uguale a due, gli viene mostrata la schermata in figura 6.10a. Tutti gli altri giocatori vengono informati del fatto che quest'ultimo non sia più in partita

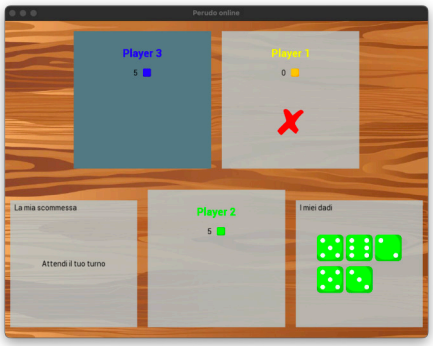
con una x rossa sulla sua card, come mostrato in figura 6.10b e 6.10c. Come si può osservare in figura 6.10a il giocatore che ha perso la partita può rimanere all'interno della lobby oppure abbandonarla, in quest'ultimo caso verrà riportato al menù principale.



(a) Giocatore che ha perso



(b) Utente di turno



(c) Utente non di turno

Figure 6.10: Perdita di tutti i dadi da parte di un giocatore

Infine vengono presentate le schermate visualizzate nel caso in cui un utente abbandoni una partita in corso, figura 6.11a, oppure in seguito all'irraggiungibilità del *server*, figura 6.11b.

(a) Utente abbandona una partita in corso

(b) Server irraggiungibile

Figure 6.11: Messaggi di errore

Chapter 7

Conclusions

Il risultato ottenuto è una versione *online*, *scalabile* e *consistente* del gioco da tavolo *perudo*, che soddisfa tutti i requisiti prefissati nelle sezioni precedenti.

7.1 Future Works

Possibili sviluppi futuri possono riguardare:

- Aggiungere una modalità di autenticazione avanzata per gli utenti, con username e password.
- Implementare un vero e proprio database per avere una persistenza dei dati. In questo modo gli utenti potrebbero ad esempio analizzare le loro partite recenti o stilare delle classifiche.
- "Abbellire" l'interfaccia grafica, in quanto non particolarmente curata per questo progetto (siccome non rientrava tra i focus principali del corso).

7.2 What did i learned

Questo progetto mi ha permesso di sperimentare in prima persona lo sviluppo di un sistema distribuito sotto ogni suo aspetto: dalla fase di design e progettazione, fino alla fase di implementazione e testing, permettendomi di interfacciarmi con problemi e situazioni in cui finora non mi ero ancora imbattuto.
Questo progetto mi ha inoltre permesso di approfondire lo sviluppo di applicazioni attraverso l'uso di *socket* e comprendere quindi la realtà dietro ad applicazioni di questo tipo.