

The Game

Simone Luzi

`simone.luzi@studio.unibo.it`

January 2024

Il progetto consiste nella realizzazione di una versione digitale e distribuita del gioco da tavolo “The Game” con un’interfaccia grafica (o “Graphical User Interface”, GUI) molto semplice e minimale.

In breve, una partita di questo gioco consiste in una sessione in cui uno o più giocatori collaborano per disfarsi delle carte presenti nel mazzo principale in cui si trovano in ordine casuale le carte numerate dal 2 al 98. Le carte devono essere giocate in 4 “mazzetti” di cui 2 partono dal numero 1 e possono accogliere solo carte più grandi dell’ultima giocata, mentre gli altri due partono dal 99 e possono accogliere solo le carte più piccole. L’unica eccezione che consente di giocare carte che non rispettino questo ordine è quando si ha una carta che differisce esattamente di una decina dall’ultima giocata.

Una ulteriore regola del gioco è che durante la partita non si possono nominare i numeri, quindi non è stata prevista una modalità di comunicazione; questo non nega ai partecipanti di utilizzare dei servizi terzi per effettuare delle chiamate vocali, purché rispettino le regole. Il manuale originale del gioco con tutte le regole è consultabile qui

Dopo aver avviato il software un giocatore può creare una nuova partita ed aspettare che i suoi amici si colleghino prima di iniziare effettivamente il gioco oppure può entrare nella stanza di una partita già creata ma non ancora iniziata.

L’utente che ha avviato la partita sarà il primo a giocare.

Durante il proprio turno un giocatore deve posizionare almeno due carte (oppure tre se si sceglie una difficoltà superiore) in uno specifico mazzetto tra quelli validi; quando il giocatore vorrà concludere il suo turno dovrà ripescare fino ad avere nuovamente 6 carte in mano (o fino alla fine del mazzo); sarà poi il turno del giocatore seguente.

Il gioco finisce quando si giocano tutte le carte oppure un giocatore non può giocare il numero di carte minime. In seguito si può scegliere di avviare una nuova partita con quel gruppo di amici o uscire dal gioco.

Indice

1	Obiettivi del progetto	3
1.1	Scenari d'uso	3
1.2	Politica di autovalutazione	5
2	Analisi dei requisiti	5
2.1	Requisiti funzionali	6
2.2	Requisiti non funzionali	7
2.3	Requisiti implementativi	8
2.4	Assunzioni	9
2.5	Tecnologie	9
2.6	Modelli/paradigmi	10
3	Design	10
3.1	Struttura	11
3.2	Comportamento	14
3.3	Interazioni	16
4	Dettagli implementativi	20
4.1	Cluster	20
4.2	Attori	21
4.3	Messaggi	22
5	Validazione e testing	23
6	Istruzioni per il deployment	24
7	Esempi d'uso	25
8	Conclusioni	31
8.1	Sviluppi futuri	32
8.2	Lezioni imparate	32

1 Obiettivi del progetto

Questo progetto si pone l'obiettivo finale di sviluppare una versione digitale e distribuita del gioco da tavolo "The Game". Oltre a questa finalità, però, si vuole realizzare un prodotto di buona fattura, perciò è possibile individuare ulteriori aspetti fondamentali come l'utilizzo delle buone pratiche di programmazione, l'attenzione alla qualità del codice con la relativa documentazione e la presenza di funzionalità aggiuntive rispetto alla semplice replica del gioco, il tutto considerando lo studio e l'utilizzo di una nuova tecnologia.

Uno degli obiettivi più importanti è il corretto utilizzo di uno strumento di build automation, in questo caso Gradle, per semplificare la gestione delle dipendenze, la compilazione e la distribuzione del software. Gradle semplifica enormemente il controllo e la modifica delle librerie utilizzate, consente di configurare e generare automaticamente i pacchetti eseguibili e permette di ignorare i fastidiosi problemi relativi alla presenza di una versione non compatibile di Java per eseguire il programma. Utilizzando propriamente questo strumento si migliora l'efficienza e la coerenza nello sviluppo e nel rilascio del software, annullando il pericolo di avere progetti non più eseguibili a distanza di anni a causa della mancanza di vecchie dipendenze, all'utilizzo di librerie deprecate o che prevedevano installazioni complesse e poco chiare.

Un altro obiettivo da raggiungere era quello di utilizzare per la prima volta la Continuous Integration, in questo caso tramite la pipeline di GitLab, facendone un uso molto basilare e limitato al building e testing.

Infine per realizzare il prodotto, ovviamente, era necessario modellare e programmare le regole del gioco e studiare il framework Akka per capire come implementare un'architettura multiplayer basata sul paradigma ad attori [1]. Questo approccio consente la gestione distribuita del gioco attraverso attori che scambiano messaggi, facilitando la costruzione di un sistema reattivo e scalabile.

Per quanto riguarda gli effettivi risultati attesi, sono i seguenti:

- un gioco funzionante sia in versione multiplayer che in versione singleplayer.
- estrema facilità di esecuzione del programma tramite il Gradle Wrapper [8], senza dover effettuare ulteriori operazioni a parte build e run (avendo solamente Java 5 o superiore installato).
- un file JAR contenente tutte le dipendenze, correttamente eseguibile con versione di Java pari ad 17 o superiore.

1.1 Scenari d'uso

I possibili scenari d'uso in cui un utente interagisce con il processo (fig. 1) sono semplici e partono dal presupposto che l'utente voglia giocare a "The Game":

- un utente vuole giocare la versione singleplayer del gioco, quindi avvia il programma, inserisce il suo nome, sceglie la difficoltà, crea la partita e la avvia subito. Al termine della partita può scegliere di giocare nuovamente.

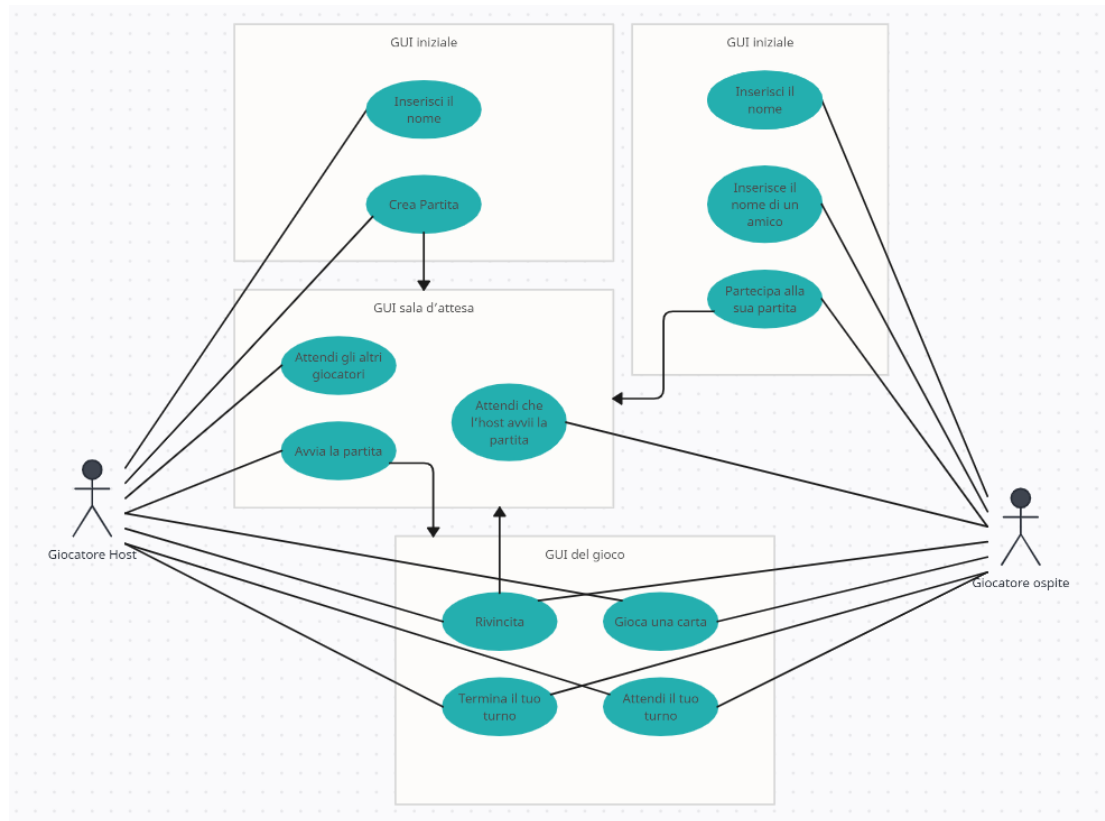


Figura 1: Diagramma dei casi d'uso dal punto di vista dei giocatori

- un utente vuole giocare con degli amici quindi inserisce il suo nome, sceglie la difficoltà, crea la partita e comunica ai suoi amici di inserire il suo nome per poter giocare insieme. Nella schermata della sala d'attesa vedrà comparire il nome degli amici connessi e quando tutti saranno presenti avvierà la partita. Questo utente sarà il primo a giocare e una volta terminato il suo turno toccherà a quello del giocatore seguente, quindi aspetterà che arrivi di nuovo il suo turno. Al termine della partita può scegliere se chiudere il programma o giocare di nuovo con quel gruppo venendo riportato nella schermata della sala d'attesa.
- un utente vuole giocare con un suo amico che ha già creato la partita, quindi inserisce il suo nome e quello dell'amico e preme il pulsante partecipa; successivamente vedrà la schermata della sala d'attesa con i nomi dei giocatori connessi e qui non potrà far altro che aspettare l'inizio della partita. I turni si svolgeranno come al solito e al termine della partita sceglierà se giocare di nuovo oppure abbandonare il gruppo che potrà comunque effettuare una nuova partita anche senza di lui.

1.2 Politica di autovalutazione

Questo progetto non punta a realizzare un prodotto accattivante, vendibile o perfetto, è considerabile più come un pretesto per studiare ed utilizzare una tecnologia inerente al mondo dei sistemi distribuiti e realizzare un progetto di buona qualità utilizzando anche uno strumento di build automation e altri strumenti avanzati come la Continuous Integration.

La qualità del software, quindi, non deve essere valutata in base allo stile dell'interfaccia grafica, al fatto che non ci sia una totale assenza di bug o alla bellezza delle regole del gioco.

Un primo elemento da valutare è sicuramente l'effettiva modellazione e programmazione del gioco, perché chiaramente, per poter considerare il software funzionante deve essere possibile giocare sia una in modalità singleplayer che in modalità multiplayer nel rispetto delle regole definite dal gioco originale. Per raggiungere questo risultato occorre aver compreso ed utilizzato propriamente il modello ad attori, pertanto il codice prodotto deve aderire alle pratiche e alle convenzioni di programmazione specifiche di questo paradigma in modo da ottenere una buona chiarezza, manutenibilità e efficacia nella gestione del sistema.

Un'altra caratteristica che contribuisce alla qualità del prodotto è il corretto utilizzo, seppur basilare, di strumenti più avanzati rispetto alla programmazione standard. In particolare l'utente percepisce subito il valore e la comodità dell'utilizzo di Gradle che lo solleva dalla gestione manuale delle dipendenze, dalla possibile incompatibilità con la sua versione di Java o della generazione degli eseguibili, mentre lo sviluppatore apprezza il fatto di capire immediatamente se l'ultima versione del codice prodotto ha invalidato il processo di build o di testing utilizzando la pipeline di GitLab per ottenere una Continuous Integration.

Un ultimo aspetto da considerare è la qualità del codice generato che prende in esame aspetti quali la leggibilità, la manutenibilità, il riuso del codice, la qualità della documentazione, la coerenza, la conformità agli standard e molti altri.

Infine per valutare l'efficacia dei risultati si possono individuare i seguenti punti:

- il programma è effettivamente eseguibile in maniera facile e immediata?
- tutti gli eseguibili prodotti funzionano correttamente nelle giuste condizioni?
- è possibile giocare e concludere una partita in modalità singleplayer?
- è possibile giocare e concludere una partita in modalità multiplayer?

2 Analisi dei requisiti

Tramite l'analisi dei requisiti si definiscono in modo chiaro le funzionalità e i vincoli del progetto, fornendo una base solida per la progettazione e lo sviluppo di un sistema distribuito.

2.1 Requisiti funzionali

1. Utilizzo appropriato di uno strumento di build automation per rendere il software facilmente portabile ed eseguibile:
 - a) download automatico di tutte le dipendenze utilizzando il Gradle Wrapper.
 - b) possibilità di eseguire il software, dopo aver effettuato la build, tramite l'apposito comando senza preoccuparsi della propria versione di Java, data la sua presenza all'interno del Wrapper.
 - c) generazione automatica di un eseguibile che sia utilizzabile in autonomia con la giusta versione di Java.
2. Modellazione ed implementazione delle regole del gioco:
 - a) numero di carte in mano e numero di carte minime da giocare ogni turno che dipendono dalla difficoltà selezionata alla creazione della partita.
 - b) possibilità di giocare le carte solamente nei "mazzetti" validi.
 - c) corretta gestione della fine del gioco, sia per la vittoria che per il Game Over.
 - d) corretta gestione dei turni, che comprende l'impossibilità di concludere il proprio turno finché non si è giocato il numero minimo di carte, l'inizio del turno del giocatore seguente al termine di un altro turno e l'impossibilità di giocare carte quando non è il proprio turno.
 - e) impossibilità di giocare le carte in "mazzetti" non validi.
3. Gestione e corretto funzionamento delle interfacce grafiche:
 - a) impossibilità di creare una partita o partecipare ad un'altra se non si inserisce il nome.
 - b) possibilità di scegliere tra le diverse difficoltà.
 - c) interfaccia della sala d'attesa reattiva in modo da poter visualizzare il nome di nuovi giocatori che si aggiungono alla partita
 - d) facoltà di far iniziare la partita concessa solamente al giocatore che la ha creata.
 - e) schermata di gioco che si apre automaticamente per tutti quando l'host (il giocatore che ha creato la partita) avvia la partita, con le carte del giocatore già visibili.
 - f) tutti i pulsanti disabilitati quando non è il proprio turno ma che si abilitano automaticamente appena quest'ultimo inizia.
 - g) ogni pulsante relativo ad un "mazzetto" sempre disabilitato a meno che il giocatore non abbia selezionato una carta giocabile lì.
 - h) selezionando una carta e poi un "mazzetto" valido il pulsante della carta scompare e il valore della stessa viene sostituito a quello presente nel "mazzetto".

- i) il pulsante per concludere il turno si abilita solamente dopo che l'utente ha giocato il numero minimo di carte.
 - j) comunicazione della fine della partita con una vittoria o un Game Over e possibilità di effettuare un rematch.
- 4. Progettare un sistema funzionante tramite Akka Cluster [2] che preveda anche una funzionalità di discovery mediante identificativo per poter trovare una partita utilizzando il nome del giocatore che la ha avviata:
 - a) possibilità di creare un attore in locale per il giocatore.
 - b) possibilità di creare un attore tramite Akka Sharding [3] che sia raggiungibile da qualsiasi nodo del cluster tramite il suo identificativo; questo è fondamentale sia per l'attore della sala d'attesa sia per quello che gestisce lo stato del gioco.
 - c) scambio di messaggi funzionante tra ogni tipo di attore.
 - d) rendere tutta la configurazione del cluster trasparente all'utente che non dovrà preoccuparsi di nulla.
- 5. Corretta gestione dei giocatori e delle partite multiplayer:
 - a) possibilità di giocare in modalità singleplayer o multiplayer.
 - b) Possibilità di connettersi ad una partita già avviata a patto di conoscere il nome del giocatore che l'ha creata.
 - c) creazione di una partita basata sul numero dei giocatori e sulla difficoltà selezionata.
 - d) effettivo inserimento di tutti i giocatori presenti nella sala d'attesa al momento della creazione della partita.
 - e) corretta gestione dei turni; quando un giocatore termina il suo turno inizierà sempre quello del giocatore seguente.
 - f) possibilità di effettuare un rematch con tutti i giocatori che erano presenti nella partita.
 - g) possibilità di iniziare comunque il rematch anche se qualche utente è uscito dal gioco.

2.2 Requisiti non funzionali

1. La consistenza è il requisito non funzionale più importante poiché assicura che i dati e lo stato degli attori siano coerenti tra i nodi del cluster; questo è fondamentale per evitare comportamenti imprevisti, consentire la corretta sincronizzazione tra le parti del sistema e mantenere l'integrità delle informazioni durante le operazioni distribuite. Ad esempio è cruciale distribuire le carte iniziali in modo che non ci siano duplicati, avere lo stesso ordine delle carte presenti nel mazzo, aggiornare lo stato dei "mazzetti" ogni volta che qualcuno gioca una carta e notificare tutti

- i giocatori quando uno di loro pesca per non far pescare le stesse carte anche ai giocatore seguenti.
2. Il sistema deve avere una bassa latenza, in questo modo i giocatori non devono aspettare più del dovuto per iniziare il loro turno e ricevere gli aggiornamenti delle carte giocate dagli altri.
 3. Possibilità di configurare il cluster Akka per scalare facilmente nel caso si volessero effettuare migliaia di partite in simultanea.
 4. Buona usabilità delle interfacce, ad esempio utilizzando i colori appropriati o disponendo i componenti in maniera corretta, per rendere chiaro il flusso di interazione e il processo di utilizzo del sistema.
 5. Buone prestazioni del sistema per garantire delle risposte tempestive durante le interazioni e le operazioni di comunicazione, in modo da fornire una esperienza utente fluida e appagante.

2.3 Requisiti implementativi

1. Modellare ogni entità del gioco come attore:
 - a) un attore che rappresenti il giocatore, che abbia i riferimenti delle interfacce grafiche per aggiornarle, che riceva i messaggi con gli aggiornamenti sullo stato del gioco e che invii messaggi per aggiornare gli altri delle suo mosse.
 - b) un attore per gestire la sala d'attesa, a cui i giocatori si devono collegare per giocare insieme e che coordini l'inizio effettivo della partita.
 - c) un attore per creare lo stato iniziale del gioco da condividere con tutti e che si occupi di inoltrare i messaggi che gli arrivano contenenti la mosse del giocatore di turno a tutti gli altri.
2. Definire tante tipologie differenti di messaggi i quali possono essere riconosciuti dai vari attori per agire diversamente a diversi eventi accaduti.
3. Configurazione del cluster Akka e di Akka Sharding per permettere la creazione degli attori e lo scambio dei messaggi:
 - a) creare un file di configurazione in cui si definisce la tipologia di sistema, l'hostname, i "seed nodes", la quantità dei log da visualizzare ed altro.
 - b) eseguire il programma sulla porta 2551 se libera, altrimenti una porta libera casuale.
 - c) creazione dell'"actor system" utilizzando la configurazione apposita.
4. Gestire il fatto che lo scambio di messaggi determina una comunicazione asincrona.
5. Utilizzare Akka per effettuare un bilancio di carico in modo da distribuire l'esecuzione degli attori nei nodi con meno lavoro.

2.4 Assunzioni

1. Per eseguire il programma occorre avere una macchina con accesso ad internet e l'autorizzazione per clonare il repository, oltre ad avere una versione di Java 5 o superiore per poter utilizzare il Gradle Wrapper.
2. La versione sviluppata, visto che doveva essere testata in locale, non necessita di una connessione di rete stabile per effettuare le partite in modalità multiplayer; in uno scenario reale, ovviamente, questo requisito diventerebbe obbligatorio.
3. I giocatori concluderanno il loro turno in un quantitativo di tempo ragionevole per non far attendere all'infinito i compagni.

2.5 Tecnologie

1. Git come sistema di controllo delle versioni per tracciare le modifiche al codice sorgente nel corso del tempo.
2. Gradle come strumento di build automation per gestire le dipendenze, generare gli eseguibili ed eseguire il programma senza avere la corretta versione di Java installata.
3. Java come principale linguaggio di programmazione.
4. Il framework Akka, in particolare:
 - a) Akka base per la gestione degli attori.
 - b) Akka Cluster per la creazione di un cluster di nodi per far comunicare attori presenti in nodi differenti.
 - c) Akka Sharding per aumentare la scalabilità del sistema ma soprattutto per poter identificare gli attori tramite il loro nome, senza doversi preoccupare delle porte fisiche.
5. Checkstyle[4] per analizzare il codice e trovare casi di mancato rispetto delle regole di stile o delle convenzioni, in modo da risolvere subito queste situazioni e aumentare la qualità per quanto riguarda coerenza, leggibilità e conformità agli standard.
6. JUnit [9] come strumento per testare il codice prodotto.
7. Il plugin Shadow [11] per generare un file JAR autonomo con tutte le dipendenze incluse visto che altri strumenti, come il plugin “application”, non gestiscono bene il framework Akka, anche a causa della gestione della serializzazione di classi o oggetti. Inoltre visto che Akka utilizza un file “reference.conf” con una configurazione di default, per il corretto funzionamento del JAR è necessario concatenare questo file con il proprio “application.conf”, così da estendere la configurazione con quella specifica di questo progetto.

8. La pipeline di GitLab [7] per ottenere la Continuous Integration configurata nel file “.gitlab-ci.yml”.

2.6 Modelli/paradigmi

1. È stato adottato un approccio “model-first”, in linea con il concetto “domain-driven design (DDD)” [6], che suggerisce come prima attività una comprensione approfondita del dominio del problema. Concentrarsi subito sugli aspetti relativi al Model consente di comprendere e definire per prima cosa i dati e le operazioni principali del sistema; chiarire questi concetti chiave fornire una base solida, consente di ridurre le ambiguità e guida le importanti scelte successive come la scelta dell’architettura e la struttura del sistema. In particolare in questo progetto ha permesso di comprendere chiaramente i ruoli che avrebbero avuto le entità fondamentali per il funzionamento del gioco che sono poi state modellate sotto forma di attori.
2. Per suddividere e organizzare il codice è stato utilizzato il pattern Model-View-Controller (MVC) [10] per strutturare il software in modo modulare e separare le responsabilità principali nei classici tre componenti; anche se a volte criticato, questo pattern offre una struttura chiara e organizzata che semplifica lo sviluppo e la manutenzione del codice. Questo approccio chiaro e scalabile, anche se semplice e limitante in alcuni casi, è risultato adatto ad un progetto di queste dimensioni.
3. Per gestire la concorrenza e la comunicazione asincrona in un ambiente distribuito è stato utilizzato il modello ad attori che promuove la modularità, l’isolamento e consente agli attori di operare in maniera indipendentemente, oltre a permettere la distribuzione del carico di lavoro e la scalabilità del sistema.
4. Un altro aspetto fondamentale è la ricezione e gestione asincrona dei messaggi, fondamentale per consentire agli attori di comunicare; la ricezione asincrona consente agli attori di rispondere in modo flessibile agli stimoli esterni, senza interrompere il flusso dell’esecuzione. Questo paradigma supporta scenari di comunicazione distribuita e contribuisce alla creazione di sistemi reattivi e responsivi, infatti in questo progetto sono gli attori i responsabili di aggiornare le interfacce grafiche quando ricevono determinati messaggi.

3 Design

In primo luogo si è scelto di procedere con lo sviluppo della versione singleplayer del gioco utilizzando il pattern MVC e partendo dalla realizzazione delle classi di Model.

In seguito, dopo aver testato il sistema ed averlo in parte predisposto alle modifiche imminenti, si è introdotto il framework Akka, sono stati creati gli attori per modellare le entità coinvolte durante le varie operazioni e si sono adattati i Controller precedentemente realizzati per il contesto multiplayer. Sono stati definiti i diversi tipi di messaggi

con le loro peculiarità per rispondere alle esigenze del sistema e per ogni attore è stato definito il comportamento in relazione alla specifica tipologia di messaggio ricevuto.

Di seguito verranno descritti in maniera approfondita gli aspetti fondamentali del progetto, ovvero la struttura, il comportamento e le interazioni del sistema.

3.1 Struttura

Come illustrato precedentemente, è stato adottato il pattern MVC per diversi motivi tra i quali separare le responsabilità, migliorare l'organizzazione del codice e la manutenibilità, rendere la struttura flessibile e facilitare la comprensione del codice. Questa divisione netta consente di modificare un componente senza influire pesantemente sugli altri, facilitando la gestione e lo sviluppo del software. Ad esempio la parte di View è stata realizzata in modo molto semplice ed essenziale visto che realizzare un gioco graficamente accattivante non era l'obiettivo del progetto e non sarebbe stato oggetto di valutazione; se in futuro si volesse migliorare l'aspetto visivo del gioco si potrebbero utilizzare componenti differenti ma facilmente integrabili a patto di implementare correttamente le interfacce predisposte.

Le classi di Model, invece, difficilmente saranno oggetto a grandi modifiche, se non nel remoto caso in cui si voglia aggiungere un'espansione del gioco con le relative nuove regole, per questo il loro sviluppo è stato effettuato con particolare attenzione. Per comprendere meglio gli aspetti fondamentali del gioco ecco una breve descrizione degli elementi di Model:

- un elenco delle possibili difficoltà del gioco.
- una classe che modelli le regole del gioco dove ad esempio vengono definite le carte del mazzo, le giocate valide, il numero di carte da pescare in base al numero di giocatori e alla difficoltà, il numero di carte minimo da giocare in un turno, ecc.
- una semplice classe per modellare i giocatori, in cui viene definito il loro nome e che memorizza e gestisce le carte che hanno in mano.
- una classe per gestire lo stato effettivo di una partita, in cui si mantiene il riferimento al mazzo mescolato con le carte da pescare, che distribuisce le carte, dove si mantiene il riferimento ai giocatori, che si occupa di memorizzare quali carte sono state giocate e che permette di verificare se la partita si è conclusa con una vittoria o un Game Over.

Per quanto riguarda i Controller, questi sono ciò che legano e mettono in relazione tutti gli altri componenti, infatti sono utilizzati per mettere in contatto tra loro i componenti di Model, i componenti di View e gli attori.

Per rappresentare le entità coinvolte nei vari scenari di utilizzo del sistema sono stati utilizzati degli attori, ovvero unità di esecuzioni leggere e isolate con un proprio stato ed un comportamento autonomo e reattivo; gli attori possono comunicare tra loro inviando messaggi asincroni per riferire cambiamenti dello stato del gioco, per sincronizzarsi o per stimolare reazioni degli altri. Per questo progetto sono stati realizzati i seguenti:

- un attore per rappresentare l'entità “giocatore”, l'elemento fondamentale del sistema che rappresenta l'utente e che si occupa di registrare le sue decisioni per poi comunicarle agli altri; inoltre aggiorna l'utente dei cambiamenti e degli eventi accaduti attraverso le interfacce grafiche.
- un attore per rappresentare la sala d'attesa a cui i giocatori si devono collegare per poter giocare insieme; questa entità è fondamentale per raggruppare i giocatori in modo da coordinare l'inizio della partita.
- un attore per gestire lo stato e lo svolgimento della partita; si occupa di creare gli elementi indispensabili per giocare, distribuisce le carte, aggiorna tutti i giocatori quando una di queste viene giocata, gestisce i turni dei giocatori e il termine della partita.
- un attore che si occupa esclusivamente dei log del sistema, per controllare lo stato del cluster e per facilitare il debug in caso di problemi.

Il framework Akka è stato utilizzato per realizzare un sistema distribuito e scalabile per permettere ad attori in esecuzione su contesti differenti di comunicare tra loro. In particolare all'esecuzione del programma viene creato un actor system, ovvero un ambiente dove è possibile creare, gestire e supervisionare gli attori precedentemente descritti; creare questo ambiente tramite la configurazione presente nel progetto consente di farlo partecipare ad un cluster, ovvero un insieme di molteplici istanze di actor system, definiti nodi, in cui possono interagire e collaborare tra loro. L'utilizzo di Akka Cluster Sharding, inoltre, consente di creare dei sottoinsiemi di attori accomunati da una “Shard Region”; questo componente aggiuntivo consente di distribuire dinamicamente gli attori tra i vari nodi del cluster per gestire grandi quantità di attori in modo scalabile, ma soprattutto consente di comunicare con gli attori conoscendo solamente il loro identificativo, consentendo di ignorare il nodo o la porta fisica del nodo dove sono eseguiti.

Nella fig. 2 sono riportate tutte le classi del progetto ad eccezione dei messaggi inviati dagli attori che, per motivi di chiarezza dello schema, sono stati omessi (sono tuttavia consultabili nella fig. 6 o facilmente individuati dal codice dei rispettivi attori che li ricevono e li inviano). Sono state colorate dello stesso colore le classi relative ai componenti di View e Controller connessi tra loro per poter individuare a colpo d'occhio questi elementi fondamentali per la comprensione della struttura del sistema.

L'organizzazione dei moduli del progetto (fig. 3) è stata effettuata in modo che i package rispettassero la struttura logica e funzionale del progetto, in modo da facilitare la navigazione e la manutenibilità e per effettuare una chiara separazione delle responsabilità. I package realizzati sono:

- Model, dove sono presenti i file che gestiscono i dati, lo stato e la logica dell'applicazione.
- View, che contiene le interfacce utente.

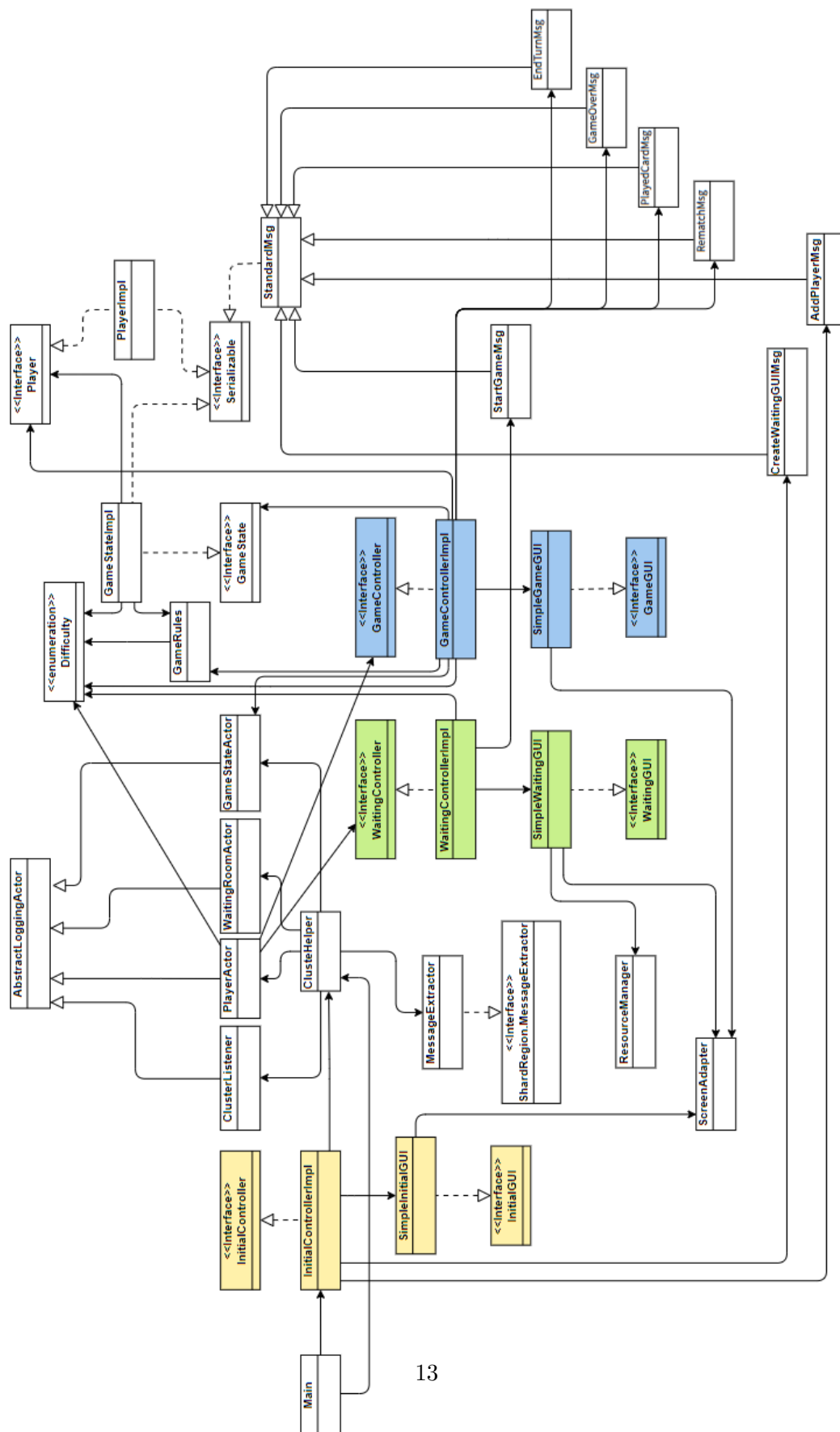


Figura 2: Class diagram che mostra le principali componenti del progetto

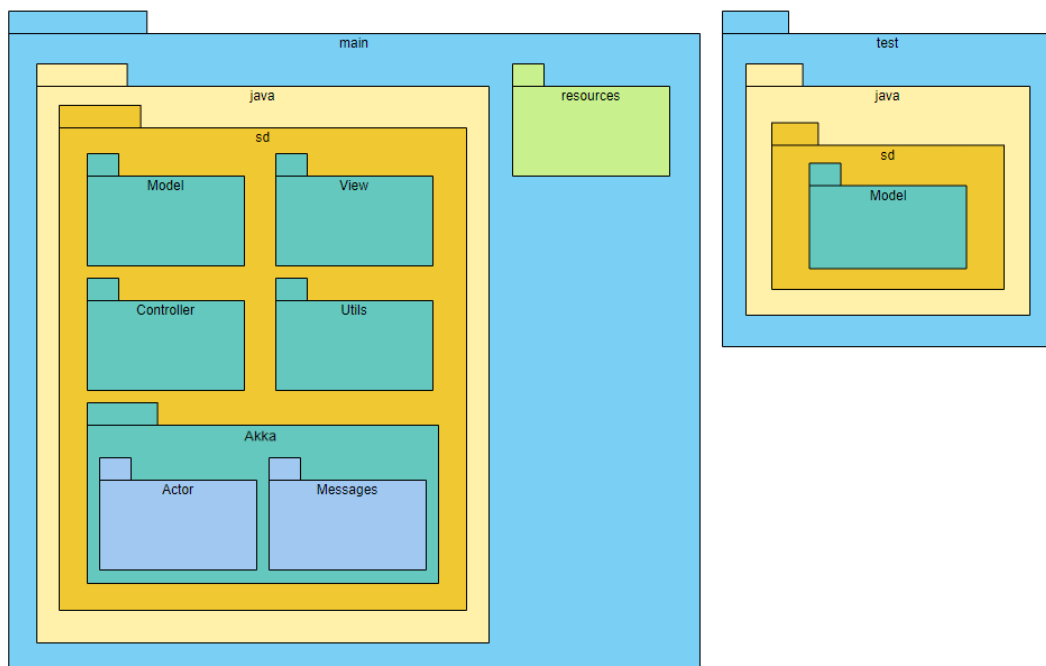


Figura 3: Package diagram che mostra i moduli del progetto

- Controller, con le classi responsabili di mettere in relazione gli attori, il Model e la View.
- Akka, dove si possono trovare i componenti relativi agli aspetti distribuiti del progetto e che contiene a sua volta una cartella contenente tutti gli attori e un'altra dove sono presenti tutte le diverse tipologie di messaggio fondamentali per la corretta interazione tra gli attori.
- Utils, dove sono presenti le classi che non fanno parte dei raggruppamenti descritti in precedenza e che servono, ad esempio, a facilitare l'utilizzo delle risorse del progetto, a gestire il cluster o a creare gli attori.
- Test, che ospita i file dedicati agli unit test indispensabili per garantire il corretto funzionamento dei componenti.

3.2 Comportamento

Per comprendere chiaramente le dinamiche del sistema è fondamentale capire come le entità chiave, che in questo caso sono rappresentate dai tre attori principali, agiscono in risposta a stimoli o eventi. Gli attori reagiscono in maniera differente ad ogni diversa tipologia di messaggio ricevuto, e a volte il comportamento cambia anche in base allo stato dell'attore al momento della ricezione.

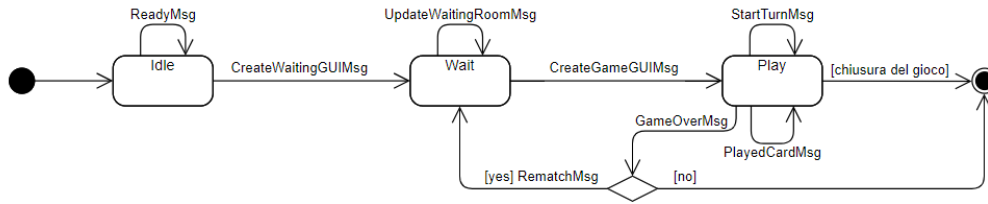


Figura 4: State Diagram dell'attore relativo al giocatore

L'attore più importante è quello relativo al giocatore, visto che si occupa di creare le interfacce, comunicare le mosse, ricevere gli aggiornamenti sullo stato del gioco e molto altro. In breve, il suo comportamento in base alla diversa tipologia di messaggio che riceve (fig. 4) è il seguente:

- **CreateWaitingGUIMsg**: crea il Controller responsabile di mostrare e gestire la GUI della sala d'attesa; nel caso in cui fosse il giocatore stesso ad aver creato la partita, memorizza la difficoltà selezionata.
- **UpdateWaitingRoomMsg**: aggiorna nella GUI della sala d'attesa l'elenco dei nomi dei giocatori connessi perché se ne è aggiunto uno nuovo e memorizza il riferimento dell'attore responsabile della sala d'attesa.
- **CreateGameGUIMsg**: chiude la GUI della sala d'attesa e crea il Controller responsabile di mostrare e gestire la GUI di gioco, nel caso in cui il giocatore fosse lo stesso che ha creato la partita, inizierebbe subito il suo turno.
- **StartTurnMsg**: nel caso in cui il giocatore non fosse lo stesso che ha terminato il turno, aggiorna il proprio stato del gioco e la GUI per rimuovere le carte pescate dal giocatore che ha concluso il turno; nel caso in cui fosse lui il giocatore seguente a quello che ha concluso, inizierebbe il suo turno.
- **PlayedCardMsg**: viene aggiornato il proprio stato del gioco perché un altro giocatore ha giocato una carta in un "mazzetto" e questo deve essere anche visualizzato nella GUI.
- **GameOverMsg**: si notifica all'utente che la partita è stata conclusa con un Game Over.
- **RematchMsg**: crea nuovamente la GUI della sala d'attesa e comunica all'attore che la gestisce di aggiungerlo di nuovo ai giocatori in lista.

L'attore che gestisce la sala d'attesa è molto semplice, funge da punto di riferimento per gli utenti che vogliono giocare insieme e si occupa di raggruppare i giocatori, salvare i loro riferimenti e coordinare l'inizio della partita. Questo attore riceve solamente due tipi di messaggi:

- **AddPlayerMsg**: un nuovo giocatore vuole unirsi a questo gruppo quindi aggiorna la sua lista e riferisce a tutti i giocatori di aggiornare le loro GUI per mostrare il suo nome.
- **StartGameMsg**: il giocatore che ha creato la partita vuole iniziare a giocare, quindi crea l'attore responsabile di coordinare la partita e gli comunica di distribuire le carte per iniziare il gioco.

L'ultima entità del sistema è quella responsabile di gestire lo stato e lo svolgimento della partita. Il relativo attore ha il compito di creare i componenti del gioco, distribuire le carte, ricevere le mosse di un giocatore, aggiornare gli altri di questa mossa, gestire i turni e comunicare il termine della partita. In particolare il suo comportamento in relazione ai messaggi che riceve è:

- **DealCardsMsg**: crea lo stato del gioco, distribuisce ad ogni giocatore le sue carte, definisce l'ordine dei giocatori e comunica a questi ultimi di creare la GUI del gioco.
- **PlayedCardMsg**: un utente ha giocato una carta quindi comunica a tutti gli altri l'evento, specificando giocatore, valore della carta e "mazzetto" in cui è stata giocata.
- **EndTurnMsg**: comunica a tutti che un determinato giocatore ha concluso il suo turno.
- **GameOverMsg**: comunica a tutti che si è verificato un Game Over.

Per capire il funzionamento del cluster, invece, occorre capire il ciclo di vita di un suo nodo (fig. 5) e i ruoli di alcuni di questi [5]. Per connettersi al cluster un nodo cerca un punto di accesso, ovvero prova a contattare dei nodi specifici detti "seed nodes"; questi ultimi gli forniscono le informazioni su come unirsi al cluster e quindi passare dallo stato "Joining" allo stato "Up". In questo stato il nodo è effettivamente connesso e può partecipare agli scambi di messaggi "Gossip" per mantenere aggiornato lo stato del cluster oltre a permettere ai suoi attori di comunicare con gli altri presenti in qualsiasi nodo. Nel caso di un guasto o di una disconnessione, un nodo può transitare in diversi stati per consentire ad Akka la corretta gestione del fallimento del nodo, ovvero "Down", "Leaving" ed "Exiting" per raggiungere infine "Removed". Il nodo leader gestisce le connessioni e disconnessioni dei nodi e coordina le operazioni di passaggio di stato; questo ruolo attivo e dinamico comporta la responsabilità di prendere decisioni critiche per il cluster e garantisce la coerenza e resilienza del cluster. L'incarico di leader può anche passare di proprietà ad altri nodi, in questo caso viene effettuata un'elezione per non lasciare il cluster privo di questo ruolo fondamentale.

3.3 Interazioni

Per avere una visione completa del sistema occorre anche comprendere come le entità comunicano e collaborano tra loro in modo da raggiungere gli obiettivi dell'applicazione.

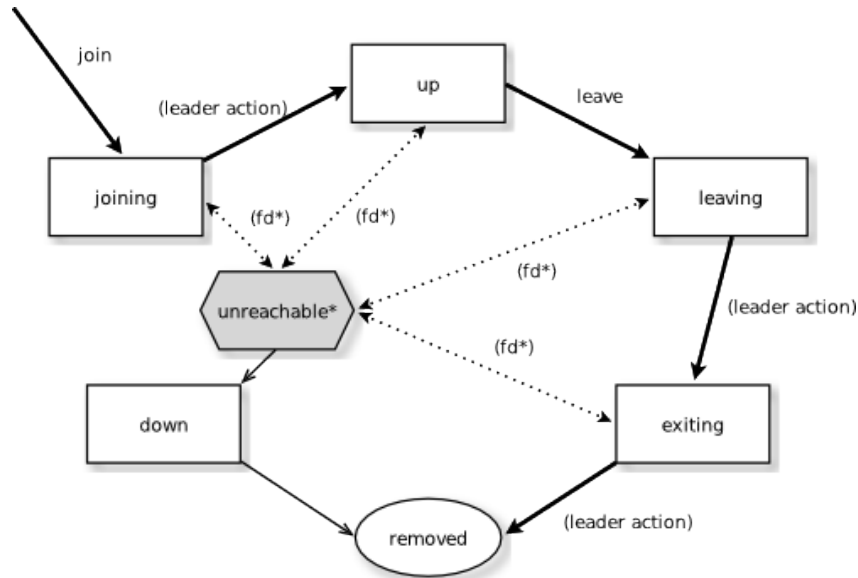


Figura 5: Gli stati relativi al ciclo di vita di un nodo del cluster (fd^* = failure detection)

Le interazioni relative al pattern MVC sono molto semplici, le azioni che l'utente effettua nella View vengono catturate dal relativo Controller che aggiorna il Model di conseguenza per mantenere la coerenza dei dati; quando invece sono i componenti di Model a subire delle variazioni, il Controller viene notificato e richiede l'aggiornamento della View. In questo progetto il Controller si occupa anche di gestire alcune operazioni riguardanti gli attori, come ad esempio inviare un messaggio per notificare la giocata di una carta oppure il termine del turno.

Le interazioni più importanti del sistema, però, avvengono tra gli attori: per garantire la corretta comunicazione e coordinazione nel sistema distribuito gli attori effettuano uno scambio asincrono di messaggi, ciò significa che possono inviare e ricevere messaggi senza dover attendere una risposta immediata. Questa natura asincrona favorisce la scalabilità e la reattività del sistema.

Normalmente per inviare un messaggio ad un attore del cluster in esecuzione in un altro nodo bisognerebbe avere il suo riferimento oppure conoscere il nodo e la porta fisica che utilizza; tramite Akka Sharding, però, è possibile ignorare questi elementi perché basta conoscere l'identificativo dell'attore per comunicare con lui. Per poterlo fare, però, è necessario definire un componente aggiuntivo, il "MessageExtractor", che è responsabile di estrarre informazioni dai messaggi, facilitando il processo di routing e garantendo che i messaggi raggiungano la destinazione corretta. Il MessageExtractor consente di mappare gli identificatori degli attori, consentendo una comunicazione efficiente e mirata, e può anche definire diverse modalità di consegna del messaggio, nel caso in cui si volesse modificare o adattare in qualche modo il contenuto.

Approfondendo le modalità di interazione bisogna distinguere due tipi di attori: quelli creati localmente e quelli gestiti da Akka Sharding. Per creare i primi si utilizza uno

specifico comando che li mette in esecuzione in locale nel proprio nodo del cluster; per inviare dei messaggi a questi attori occorre un riferimento che contiene nodo e porta fisica tramite il quale è possibile raggiungerlo. Per quanto riguarda gli altri, prima di poter essere creati è necessario avviare una “Sharding Region”, ovvero un componente responsabile della gestione degli attori all’interno di una partizione specifica, anche detta “shard”. La Shard Region è identificata da un nome e può contenere solo uno specifico tipo di attori; la loro creazione, a differenza degli attori locali, viene effettuata automaticamente quando viene inviato un messaggio in quella regione e non esiste ancora un attore con l’identificatore presente nel messaggio. Questo identificatore viene recuperato proprio grazie al MessageExtractor menzionato prima. Quando un attore locale invia un messaggio ad un attore presente in uno shard tramite l’apposita Shard Region e il suo identificativo, quest’ultimo può ottenere il suo riferimento dal messaggio ricevuto che può utilizzare per rispondergli; in questo modo è possibile la comunicazione tra questi due tipi di attori.

Come detto precedentemente, gli attori non si limitano a scambiarsi messaggi tra di loro, a volte creano altri attori, come nel caso di WaitingRoomActor che crea GameStateActor, oppure creano altri componenti come quest’ultimo attore quando istanzia il GameState. Inoltre il PlayerActor si relaziona frequentemente con i Controller, infatti è proprio questo attore che crea sia il WaitingController che il GameController e li utilizza per aggiornare le GUI quando necessario.

Nella fig. 6 sono mostrate le interazioni tra gli attori principali del sistema e i controller responsabili di aggiornare le interfacce grafiche e di raccogliere le interazioni effettuate dall’utente. Al termine di alcune linee di vita sono presenti tre puntini di sospensione ad indicare che, dopo aver avviato un rematch, le entità possono scambiarsi gli stessi messaggi visti precedentemente.

Per approfondire ulteriormente lo scambio di messaggi, elenchiamo brevemente quali attori inviano quali messaggi, a chi e perché:

- InitialController:
 - invia un CreateWaitingGUIMsg al PlayerActor per far visualizzare la sala d’attesa.
 - invia un AddPlayerMsg al WaitingRoomActor per connettersi alla partita.
- WaitingRoomActor:
 - invia un UpdateWaitingRoomMsg a tutti PlayerActor connessi alla partita ogni volta che un nuovo giocatore si unisce.
 - invia un DealCardsMsg al GameStateActor per iniziare la partita.
- WaitingController:
 - invia un StartGameMsg al WaitingRoomActor per far creare il GameStateActor.

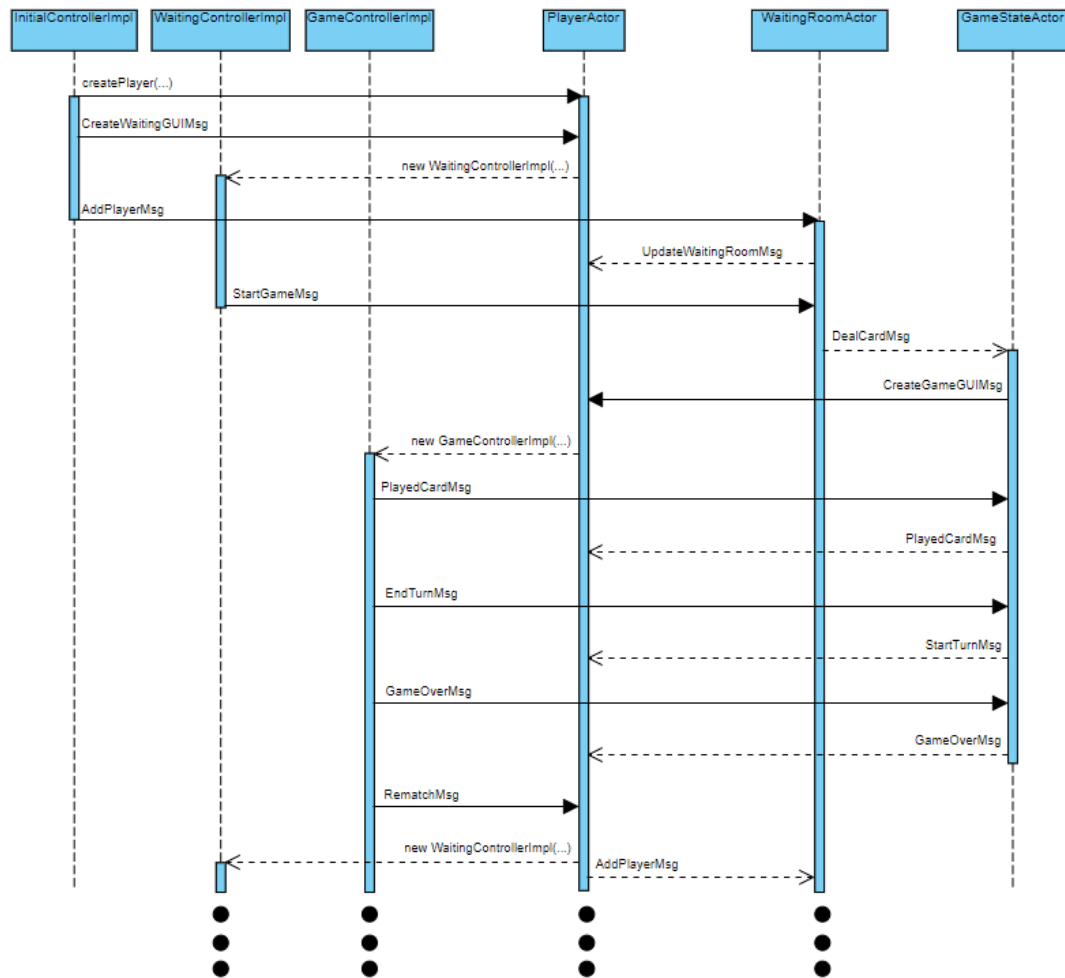


Figura 6: Sequence Diagram contenente le interazioni tra le entità del sistema

- GameStateActor:
 - invia un CreateGameGUIMsg ad ogni giocatore per creare il GameController ed iniziare la partita.
 - invia un PlayedCardMsg a tutti i PlayerActor della partita, tranne al giocatore che ha effettuato la mossa, per comunicare quale carta è stata giocata, dove e da chi.
 - invia un StartTurnMsg a tutti i giocatori quando uno di loro finisce il turno.
 - invia un GameOverMsg a tutti i giocatori quando uno di loro determina un Game Over.
- PlayerActor:
 - invia un AddPlayerMsg al WaitingRoomActor quando l'utente sceglie di fare un rematch.

4 Dettagli implementativi

In questa sezione verranno approfonditi alcuni aspetti relativi all'effettivo sviluppo del sistema. Il codice è già provvisto di abbondante documentazione visto che le classi e metodi sono stati forniti di una “Javadoc” che ne spiega funzionalità e modalità di utilizzo; per questo motivo non verranno esaminati aspetti relativi ai componenti di MVC ma ci si concentrerà sugli aspetti più interessanti del progetto, ovvero il cluster, gli attori e lo scambio dei messaggi.

4.1 Cluster

Come abbiamo già detto il cluster permette ad attori in esecuzione su nodi diversi di comunicare tra loro. Per facilitare la gestione del cluster è stata creata una classe con tutti metodi statici chiamata ClusterHelper; questa classe consente di effettuare operazioni quali avviare l'actor system, connettersi al cluster, creare i PlayerActor ed ottenere il riferimento alla Shard Region utilizzate per i WaitingRoomActor e i GameStateActor.

In particolare l'avvio del cluster è un momento fondamentale in quanto deve essere effettuato con alcune attenzioni. Per avviare correttamente il processo di formazione è fondamentale comprendere il ruolo dei “seed nodes”, ovvero nodi iniziali nel cluster Akka, configurati con indirizzi IP e porte specifici, che sono responsabili di avviare il processo di formazione del cluster e di introdurre nuovi nodi. Quando si avvia un actor system in modalità cluster, i nodi cercheranno i seed nodes come punto di ingresso per unirsi al cluster, per questo è fondamentale che il primo programma eseguito utilizzi una porta specifica presente nel file di configurazione, in modo da diventare lui stesso un seed node. Successivamente, quando altri nodi verranno avviati su altre porte, questi cercheranno e troveranno un seed node che li permetterà di unirsi al cluster.

Nel caso in cui tutti i seed nodes escano dal cluster a causa della terminazione del programma al quale erano legati, il cluster continuerebbe comunque a funzionare visto

che viene applicata automaticamente una strategia per il quale un altro nodo viene eletto come leader. Il leader ha l'obiettivo di mantenere l'integrità del cluster e quindi si occupa di prendere delle decisioni critiche per evitare il cosiddetto “split brain”, che si verifica quando due o più partizioni di nodi nel cluster perdono la connessione tra loro e iniziano a operare indipendentemente. Questo processo contribuisce a garantire la resilienza del cluster anche in caso di fallimento o terminazione di uno o più seed nodes.

Nel listing seguente sono riportati i passaggi responsabili dell'avvio del cluster.

```
private static final int DEFAULT_PORT = 2551;
private static final int RANDOM_PORT = 0;

public static ActorSystem startCluster() {
    Config config = ConfigFactory.load();
    String path = "akka.remote.artery.canonical.port";
    int port = isDefaultPortAvailable() ? DEFAULT_PORT : RANDOM_PORT;
    ActorSystem system = ActorSystem.create("ClusterSystem",
        config.withValue(path, ConfigValueFactory.fromAnyRef(port)));
    // avvia l'attore responsabile dei log del sistema
    system.actorOf(ClusterListener.props(), "clusterListener");
    return system;
}
```

Un altro aspetto interessante è come viene creata una Shard Region oppure come si ottiene il suo riferimento: bisogna conoscere il suo nome, definire quale tipologia di attori può ospitare e specificare quale MessageExtractor sarà responsabile della gestione dei messaggi in arrivo. Di seguito un breve listing che mostra il codice necessario per creare la Shard Region relativa al WaitingRoomActor.

```
public static ActorRef getWaitingRegion(final ActorSystem system) {
    return ClusterSharding.get(system).start(
        "Waiting",
        WaitingRoomActor.props(),
        ClusterShardingSettings.create(system),
        new MessageExtractor()
    );
}
```

4.2 Attori

Come già detto gli attori possono essere creati tramite un metodo specifico (ovvero `system.actorOf`) oppure creati automaticamente da Akka Sharding e in entrambi i casi occorre specificare un nome univoco. In ambedue le situazioni non è possibile effettuare semplicemente una “new” come accade con gli oggetti di normali classi Java, quello che viene fatto è inviare all'actor system delle istruzioni per delegare la creazione di un nuovo attore del tipo specificato. Per convenzione un metodo “props” viene inserito nelle classi degli attori per fornire la configurazione per la creazione di un'istanza dell'attore; qui

vengono specificati i dettagli per la creazione dell'attore, come la classe al quale bisogna far riferimento ed eventuali parametri da passare al costruttore. Sfruttare questa possibilità può rivelarsi utile visto che consente di passare immediatamente delle informazioni all'attore anche se non si istanzia personalmente; in seguito l'unico modo per modificare lo stato di un attore sarà inviargli dei messaggi.

Visto che l'effettiva creazione e la dismissione degli attori è gestita da Akka, è importante sapere che è possibile personalizzare in parte queste situazioni tramite degli appositi metodi predisposti da Akka che vengono chiamati automaticamente in determinati momenti della vita dell'attore. Tramite il metodo "preStart" si possono effettuare inizializzazioni o caricare le configurazioni necessarie prima che l'attore inizi a svolgere il suo lavoro; in `ClusterListener`, ad esempio, vengono effettuate delle sottoscrizioni le quali definiscono gli eventi che, una volta accaduti, invieranno una notifica all'attore. Quando l'attore viene arrestato o terminato, invece, viene chiamato il metodo "postStop" che solitamente è utilizzato per effettuare la pulizia delle risorse o per eseguire azioni di chiusura prima che l'attore venga dimesso; in `ClusterListener`, ad esempio, si rimuove l'iscrizione dalle notifiche del cluster così da evitare sprechi di memoria e risorse.

L'ultima caratteristica importante da discutere riguardo gli attori è la modalità con il quale riceve e gestisce i messaggi. Ogni attore deve implementare il metodo astratto "createReceive", che definisce il comportamento dell'attore, ovvero come reagisce alle diverse tipologie di messaggio che può ricevere; questo metodo restituisce una funzione che rappresenta la logica di gestione dei messaggi dell'attore. Utilizzando classi di messaggi personalizzati è possibile estrarre molteplici informazioni da un singolo messaggio, anche di tipi non banali purché serializzabili; questa flessibilità consente di rappresentare concetti specifici e, reagendo in modo appropriato a ciascun tipo di input, è possibile coordinare efficacemente il sistema distribuito. Nel listing seguente viene riportato l'esempio del `WaitingRoomActor` dove `handleAddPlayer` e `handleStartGame` sono dei metodi che, preso in input i rispettivi messaggi, definiscono il comportamento dell'attore.

```
@Override
public Receive createReceive() {
    return receiveBuilder()
        .match(AddPlayerMsg.class, this::handleAddPlayer)
        .match(StartGameMsg.class, this::handleStartGame)
        .matchAny(message -> System.out.println("\n# UNEXPECTED
            MESSAGE TO " + getSelf() + ": " + message))
        .build();
}
```

4.3 Messaggi

Per poter comunicare con attori presenti nelle `Shard Region` è necessario definire delle proprie tipologie di messaggi in modo da poter configurare il `MessageExtractor` a riconoscere le informazioni presenti al loro interno. Per rendere più chiara e snella la gestione dei messaggi è stata realizzata una classe astratta con tutte le caratteristiche base che doveva avere un messaggio valido; tutti i messaggi specifici utilizzati nelle diverse occasioni

sono un'estensione di quello astratto. Questo consente di semplificare enormemente il lavoro del MessageExtractor, che dovrà specificare solamente la gestione di un singolo tipo di messaggio, quello comune a tutti.

Analizzare i metodi presenti all'interno di MessageExtractor potrebbe aiutare a chiarire il suo compito, che può risultare leggermente confusionario:

- `entityId`: estrae l'identificativo dell'attore a partire dal messaggio ricevuto in modo da poter inviarlo alla giusta entità presente all'interno di una Shard Region; visto che tutti i messaggi condividono la struttura di quello astratto, è sufficiente gestire quella tipologia in cui l'identificativo corrisponde al destinatario del messaggio.
- `entityMessage`: questo metodo può essere utilizzato per convertire un messaggio esterno in un messaggio interno comprensibile dagli attori della Shard Region; nel nostro caso non è stato necessario effettuare nessun mapping.
- `shardId`: determina l'identificativo dello shard associato a un messaggio; visto che ogni attore non locale viene assegnato ad uno shard ed ognuno di questi appartiene ad un nodo, per poter comunicare con gli attori è fondamentale conoscere lo shard in cui sono stati inseriti. Utilizzando una funzione di hash che utilizza gli identificatori degli attori, inoltre, è possibile distribuire gli shard, e quindi gli attori, in modo uniforme tra i vari nodi. Per questo progetto sono stati utilizzati un numero di shard pari a 10 ma in un contesto reale bisognerebbe adattare questa specifica in base al carico di lavoro atteso, alla dimensione del cluster, alla memoria disponibile e alla capacità computazionale degli shard.

I messaggi strutturati per poter raggiungere gli attori delle Shard Region in realtà sono stati utilizzati anche per comunicare con gli attori locali; questi ultimi non ne avrebbero bisogno visto che vengono contattati tramite il loro riferimento, ma si è scelto questo approccio per due motivi principali. Il primo è per una maggiore chiarezza e comodità, introdurre una differenza tra i messaggi destinati ai due diversi tipi di attori avrebbe comportato più confusione e difficoltà di gestione. Il secondo è che nel caso in cui in futuro si volessero rendere anche gli attori attualmente locali contattabili tramite una Shard Region per migliorare ulteriormente la scalabilità del sistema, allora non occorrerebbe modificare i messaggi che ricevono perché sarebbero già predisposti a questo utilizzo.

5 Validazione e testing

Uno dei criteri per la valutazione era quello di produrre codice di buona qualità e per rispettare questo requisito si sono adottate diverse “best practice” della programmazione. Ad esempio si è ridotto al minimo la duplicazione del codice richiamando metodi appositi, sfruttando classi di utility con metodi statici, inserendo gli stream e lambda dove possibile, inserendo la Javadoc nei metodi e nelle classi e altro ancora. In particolare Checkstyle può essere considerato uno strumento di supporto per la validazione automatica del codice; questo plugin, infatti, ha consentito di migliorare la qualità complessiva del codice aiutando a mantenere uno stile coerente e che rispetti le linee guida, fondamentale per agevolare la manutenzione del software.

Sono stati realizzati tramite JUnit un insieme di test automatici che hanno come obiettivo quello di verificare che tutti gli elementi fondamentali del gioco, ovvero le componenti del Model, fossero perfettamente funzionanti. Tramite una specifica classe di test ci si è assicurato che le regole di gioco modellate fossero in linea con quelle reali in modo da permettere solamente delle mosse valide e che si verificassero i giusti cambiamenti a seconda della difficoltà selezionata. Altri test hanno coinvolto gli aspetti relativi allo stato del gioco, come la distribuzione iniziale delle carte, le modifiche generate da una mossa di un giocatore o le conseguenze della pescata di fine turno; per ottenere un gioco funzionante tutte queste operazioni dovevano essere il più solide e consistenti possibile.

Questi test, seppur abbastanza semplici, hanno ottenuto un gran valore quando sono stati combinati con la Continuous Integration configurata su GitLab. Infatti le classi relative al Model sono state leggermente modificate nel corso dello sviluppo del progetto, a volte per delle dimenticanze relative a regole di minor importanza, altre volte a causa di casi limite che non erano stati inizialmente considerati. Dopo queste modifiche, anche nel caso in cui ci si fosse scordati di avviare i test per vedere se tutto fosse ancora funzionante, questo sarebbe stato fatto in automatico e ciò avrebbe fatto subito notare eventuali errori, fornendo una sicurezza importante, ovvero quella di non lavorare mai su un sistema non più funzionante.

Per motivi di tempo, purtroppo, non è stato possibile sviluppare dei test automatici anche per quanto riguarda gli aspetti relativi al cluster e agli attori; per farlo si sarebbe potuto utilizzare un modulo di testing che Akka mette a disposizione grazie al quale si può effettuare sia il test degli attori individuali che quello del comportamento del cluster. Inizialmente per verificare che il comportamento del cluster e degli attori fosse in linea con i requisiti del sistema, sono stati effettuati dei test manuali, utilizzando i log di ClusterListener e delle stampe inserite in posizioni strategiche nei vari controller e attori; dopo che il sistema ha incominciato a funzionare e i vari flussi erano gestiti in maniera corretta i log e le stampe sono stati rimossi.

6 Istruzioni per il deployment

Come riportato nel file README.md, oltre ad avere l'autorizzazione ad accedere al repository, l'unico requisito per eseguire il programma è avere Java 5 o superiore installato nella propria macchina per poter utilizzare il Gradle Wrapper.

Uno degli obiettivi era proprio quello di utilizzare al meglio uno strumento di build automation per rendere il più semplice possibile l'utilizzo del programma. Per eseguirlo, infatti, basta aprire un terminale a linea di comando e seguire i seguenti passaggi:

1. clonare il repository con il seguente comando, assicurandosi di avere l'autorizzazione per accedervi

```
git clone https://dvcs.apice.unibo.it/pika-lab/courses/ds/projects/  
sd-project-luzi-2021.git
```

2. entrare nella cartella del progetto con il comando

```
cd sd-project-luzi-2021
```

3. Effettuare la build del progetto tramite il Gradle Wrapper

```
./gradlew build
```

4. Dopo aver aspettato qualche minuto che la build sia completata, eseguire il programma tramite il Gradle Wrapper

```
./gradlew run
```

Nel caso in cui si volesse eseguire il programma senza utilizzare il Gradle Wrapper, dopo aver effettuato la build verrà generato un file JAR contenente tutte le dipendenze che può essere eseguito in autonomia se nella propria macchina è installato Java 17 o superiore. Per eseguirlo è possibile utilizzare il comando:

```
java -jar .\build\libs\The_Game.jar
```

Ovviamente è anche possibile spostare il JAR dalla sua posizione iniziale all'interno del progetto ed eseguirlo tramite doppio click.

7 Esempi d'uso

Questa sezione può essere considerata come un approfondimento del sottocapitolo “Scenari d'uso” con le immagini delle schermate del programma a supporto; come già detto gli scenari d'uso sono:

1. Un utente vuole giocare in versione singleplayer, quindi avvia il gioco, inserisce il suo nome, sceglie la difficoltà (fig. 7) e appena entrato nella sala d'attesa (fig. 8) non aspetta nessuno ed avvia il gioco. Durante la partita non dovrà aspettare le mosse di nessun altro giocatore visto che sarà sempre il suo turno; una volta conclusa la partita con una vittoria o un Game Over può scegliere se continuare a giocare o chiudere il programma.
2. Un utente vuole giocare in versione multiplayer, quindi avvia il gioco, inserisce il suo nome, sceglie la difficoltà, entra nella sala d'attesa e aspetta che tutti i suoi amici si colleghino. Quando tutti i nomi dei suoi amici saranno comparsi nell'interfaccia grafica (fig. 9) vorrà dire che sono connessi, quindi può premere il pulsante per avviare la partita e iniziare a giocare; la schermata di gioco verrà aperta a tutti i giocatori connessi. Inizierà subito il suo turno (fig. 10), quindi dopo aver giocato il numero minimo di carte (fig. 11) potrà concludere il suo turno (fig. 12); in seguito dovrà aspettare che tutti gli altri giocatori effettuino le loro mosse prima di poter di nuovo giocare. Quando la partita sarà conclusa può scegliere se chiudere il programma o iniziare una nuova partita (fig. 13) con tutti i giocatori che vorranno fare altrettanto, in questo caso verrà riportato alla schermata d'attesa

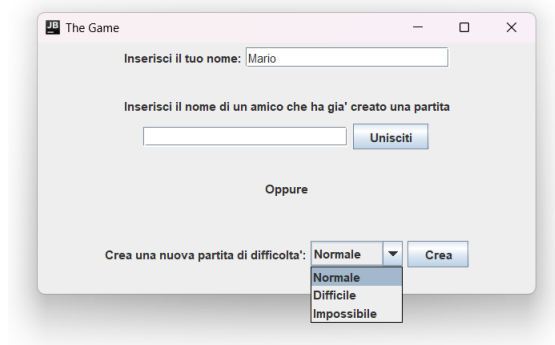


Figura 7: Inserimento nome e scelta difficoltà

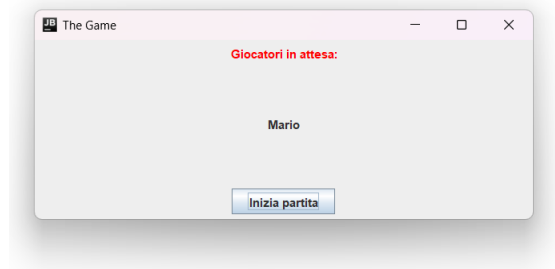


Figura 8: Sala d'attesa senza altri giocatori

dove potrà aspettare di nuovo che tutti i partecipanti si colleghino per poi iniziare nuovamente a giocare.

3. L'ultimo scenario è molto simile al precedente, le uniche differenze sono che invece di selezionare la difficoltà ed avviare una nuova partita inserirà il nome dell'amico che ne ha già avviata una e premerà il pulsante "Unisciti" (fig. 14). Successivamente potrà solamente visualizzare i nomi dei partecipanti nella schermata della sala d'attesa ed aspettare che il creatore della partita decida di iniziare a giocare (fig. 15). Una volta iniziata la partita dovrà attendere il suo turno (fig. 16) prima di iniziare a giocare (fig. 17). Al termine della partita, se sceglie di giocare ancora dovrà aspettare nuovamente che il giocatore che l'ha creata la avvii.

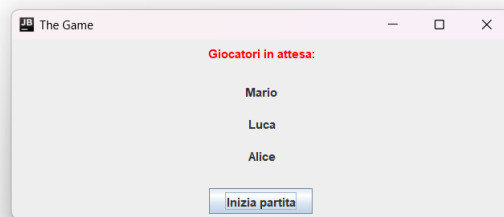


Figura 9: Sala d'attesa con altri due giocatori

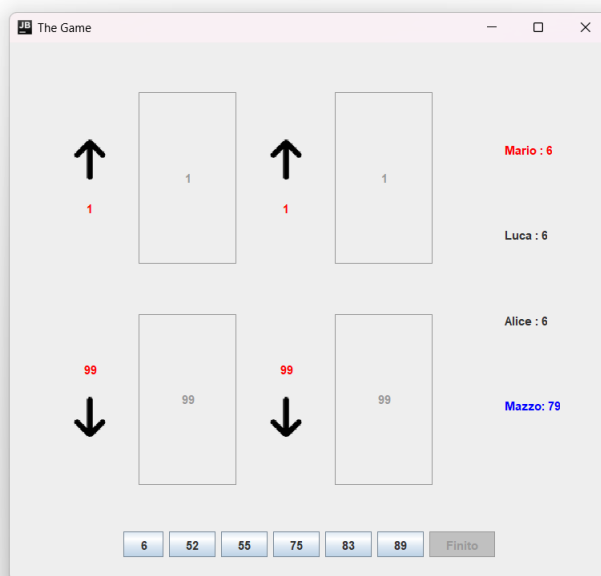


Figura 10: Inizia il turno del giocatore che ha creato la partita che può selezionare una carta da giocare

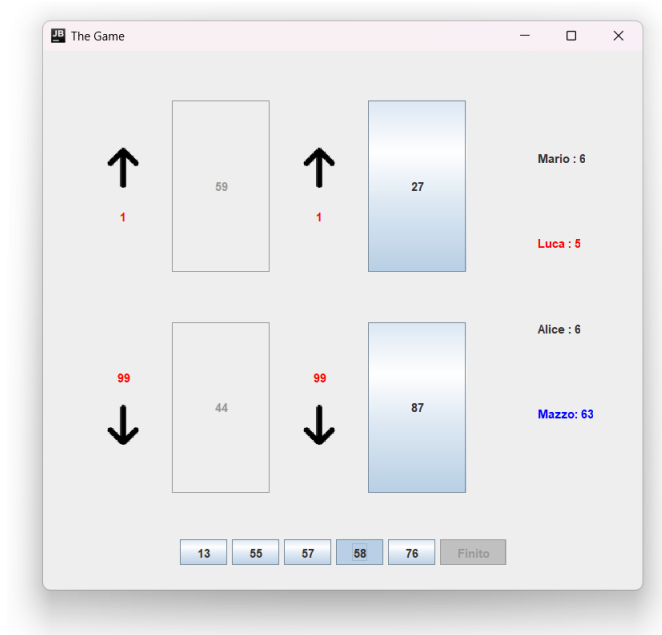


Figura 11: Selezionando una carta si abilitano i mazzi in cui può essere giocata

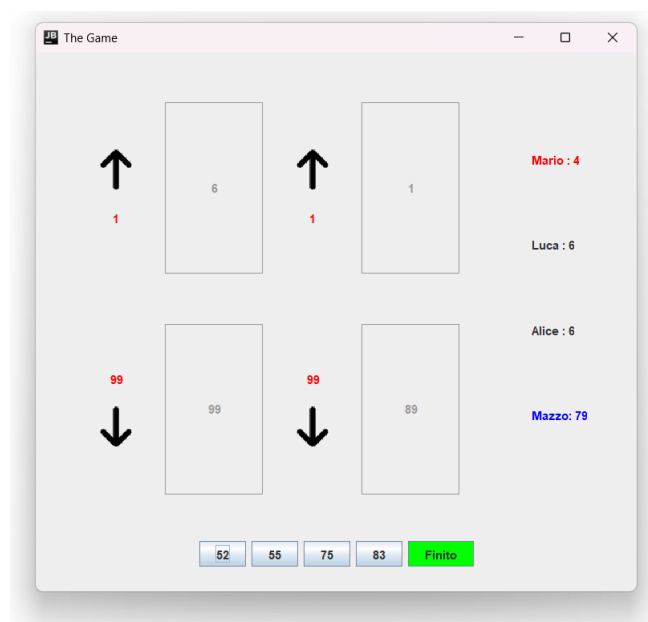


Figura 12: Dopo aver giocato il numero minimo di carte il pulsante “Finito” diventa verde e cliccabile

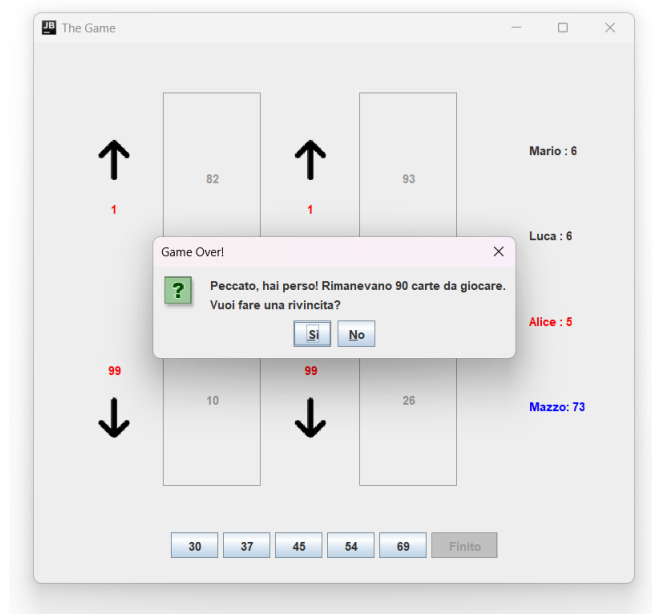


Figura 13: Al termine della partita viene chiesto se l'utente desidera rigiocare

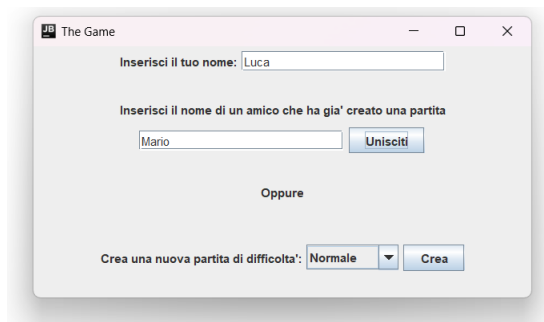


Figura 14: Inserimento del proprio nome e di quello dell'amico con cui giocare

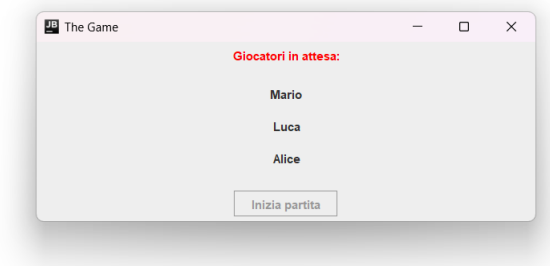


Figura 15: Visualizzazione dei partecipanti senza poter iniziare la partita

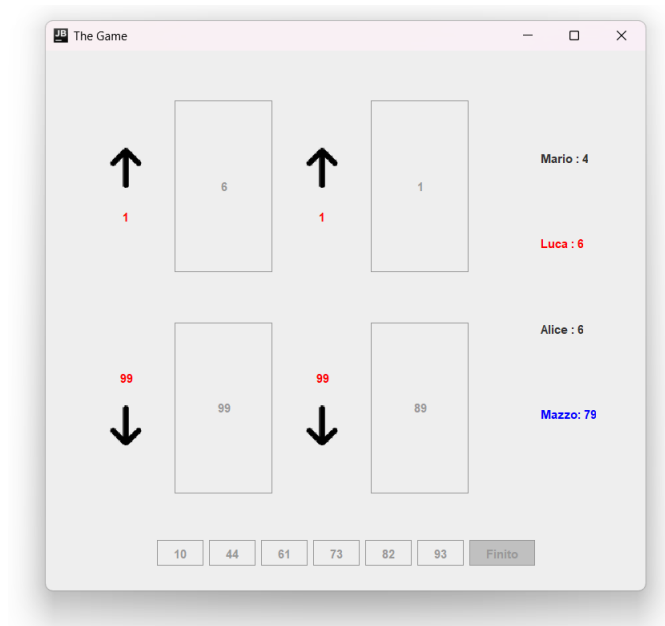


Figura 16: Il giocatore attende che arrivi il suo turno

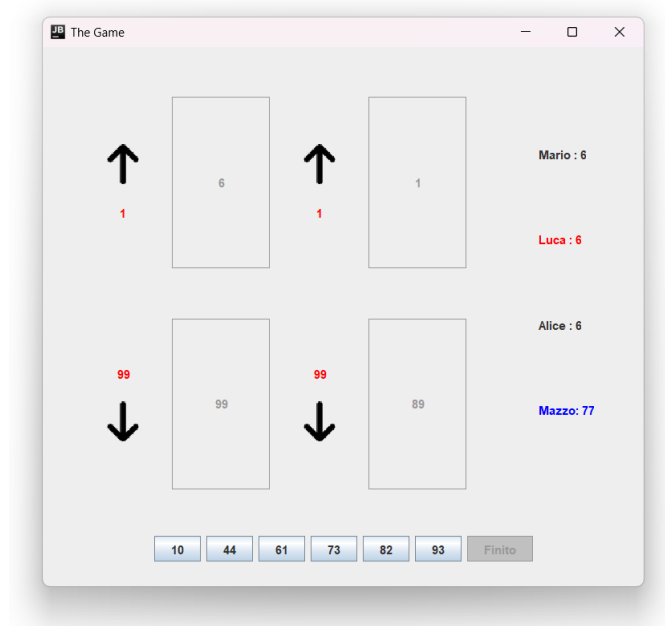


Figura 17: Quando arriva il turno del giocatore le sue carte vengono abilitate

8 Conclusioni

Questo progetto ha portato alla realizzazione di un sistema in cui è possibile giocare ad una versione digitale completa del gioco da tavolo “The Game” ed ha fornito una panoramica approfondita sulle sfide e le soluzioni nel campo della programmazione concorrente e distribuita.

Realizzare un videogioco funzionante rappresenta spesso una sfida impegnativa perché è necessario mettere in pratica molte conoscenze acquisite durante gli anni per sviluppare un prodotto di buona qualità; in questo caso il tutto risultava ancora più complesso a causa dell’esigenza di comprendere come configurare un sistema distribuito utilizzando un framework conosciuto solo superficialmente. Infatti, sebbene il modello ad attori e lo scambio asincrono di messaggi siano concetti abbastanza intuitivi, la configurazione di un cluster Akka con più nodi all’interno di una sola macchina raggiungibili attraverso le porte fisiche ma che allo stesso tempo permetta di comunicare con degli attori conoscendo solamente il loro identificativo, è stato molto più complicato del previsto.

Sono soddisfatto del prodotto finale per diversi motivi: il primo, e forse il più importante, è stato quello di aver utilizzato efficacemente le possibilità fornite da Gradle in modo da non dovermi preoccupare di non riuscire ad utilizzare questo programma tra diversi anni; diverse volte, infatti, mi è capitato di non riuscire più ad eseguire vecchi progetti a causa di dipendenze mancanti, versioni incompatibili o altri problemi, con mio grande dispiacere.

Ho provato piacere anche ad utilizzare, seppur in maniera molto semplice, la Continuous Integration visto che è stata la prima volta che ho utilizzato questa tecnologia. Questa funzionalità mi aveva sempre incuriosito e utilizzarla mi ha permesso di capire meglio le potenzialità che può portare se correttamente integrata ad un progetto oltre a contribuire, insieme a Gradle, ad ottenere un processo di sviluppo fluido e affidabile.

Un’altra grande soddisfazione è stata quella di essere riuscito da solo a superare le difficoltà riscontrate per la configurazione del cluster per il mio caso specifico; l’utilizzo di questa tecnologia non è affatto banale ed è confortante essere riuscito a comprendere e ad utilizzare Akka Cluster e Akka Sharding nonostante tutti gli errori e le incomprensioni iniziali commesse durante lo sviluppo. In particolare la partecipazione al cluster mediante seed nodes ha mostrato la robustezza del sistema in presenza di nodi irraggiungibili e l’utilizzo delle regioni di shard, inizialmente non previsto, ha consentito una distribuzione efficiente degli attori individuabili semplicemente tramite il loro identificativo, garantendo anche la scalabilità del sistema. Tutto ciò ha dimostrato l’efficacia di Akka nel costruire sistemi scalabili e resilienti.

L’esperienza acquisita nella progettazione e implementazione di questo progetto ha ampliato la comprensione della programmazione ad attori, dei modelli distribuiti e della gestione di un cluster. Questo percorso di apprendimento si è rivelato fondamentale per esplorare le potenzialità di Akka in scenari reali e ha fornito una base solida per affrontare sfide future nella programmazione distribuita.

8.1 Sviluppi futuri

Il sistema è stato sviluppato in modo da avere tutte le caratteristiche necessarie per il corretto funzionamento del gioco e per gestire adeguatamente ogni possibile scenario; nonostante questo ci sarebbero molti miglioramenti possibili per quanto riguarda il testing, l'aggiunta di funzionalità accessorie, i controlli da effettuare e la gestione degli errori.

Uno degli aspetti tralasciati per motivi di tempo è proprio quello relativo al testing: sono stati realizzati dei test automatici solo per le componenti relativi al Model mentre sarebbe stato molto interessante capire come poter testare efficacemente il cluster e gli attori, utilizzando anche degli appositi strumenti forniti da Akka.

Per motivi di praticità, inoltre, si è realizzato un sistema da eseguire in una sola macchina; in uno scenario reale, ovviamente, si sarebbe dovuto configurare il sistema in modo tale da funzionare con computer differenti collegati in rete e non è stata fatta nessuna prova a riguardo. Il sistema necessario a gestire questa situazione dovrebbe prevedere sempre qualche seed node attivo in degli indirizzi IP fissi e impostati all'interno del file di configurazione, in questo modo si garantisce la funzionalità, l'affidabilità e la resilienza del cluster. Una volta configurato per funzionare tramite internet sarebbe stato necessario sviluppare anche una gestione delle disconnessioni, non presente in questo progetto.

A causa del funzionamento degli shards non sono stati effettuati dei controlli sulla presenza di giocatori con lo stesso nome o dell'effettiva presenza di partite avviate al quale connettersi. Visto che, una volta avviata una regione, un attore viene automaticamente creato tramite l'identificatore presente all'interno di un messaggio inviato, un modo per poter effettuare questi controlli sarebbe avviare tutti gli attori destinati agli shards in una modalità di errore che provvede immediatamente ad eliminarli a meno che non vengano correttamente creati tramite specifici messaggi.

Un'ultima possibile aggiunta che non riguarda Akka, invece, è l'inserimento delle regole relative all'espansione di "The Game" chiamata "On Fire" per aumentare ulteriormente il livello di sfida e che prevede l'inserimento di carte speciali le quali, una volta giocate, obbligano i giocatori ad avere alcuni accorgimenti strategici.

8.2 Lezioni imparate

Tra le conoscenze acquisite e messe in pratica relative a questo progetto possiamo considerare:

- la gestione di un cluster Akka per creare un sistema distribuito, creando un file di configurazione appropriato e comprendendo il ruolo fondamentale dei seed nodes per gestire in modo efficace la comunicazione e la sincronizzazione tra i nodi del cluster.
- l'adozione del modello ad attori per gestire la concorrenza e la distribuzione; ciò permette di costruire sistemi scalabili e reattivi ponendo particolare attenzione alla gestione degli stati, alla modularità e la manutenibilità del codice.

- l'integrazione di Akka Sharding per aggiungere scalabilità al sistema e comunicare con gli attori grazie unicamente al loro identificativo, che ha richiesto una comprensione delle regioni, degli shard e della gestione dei messaggi.
- come coordinare gli attori del cluster tramite la comunicazione asincrona di messaggi specifici adatti alle esigenze del sistema, in particolare per quanto riguarda la condivisione di uno stato iniziale e la gestione reattiva di un'interfaccia grafica legata ad un attore.
- l'implementazione di pratiche di validazione continua, come l'utilizzo di Checkstyle e il testing automatico tramite Continuous Integration, per prevenire ed individuare errori in modo tempestivo, oltre a mantenere una buona qualità del codice nel corso del tempo.
- un'efficace utilizzo di uno strumento di build automation per la gestione delle dipendenze, per facilitare l'esecuzione e per generare eseguibili contenenti tutte le dipendenze e i file di configurazione necessari.

Riferimenti bibliografici

- [1] Actor lifecycle. <https://doc.akka.io/docs/akka/current/typed/actor-lifecycle.html>.
- [2] Akka cluster documentation. <https://doc.akka.io/docs/akka/current/cluster-usage.html>.
- [3] Akka cluster sharding documentation. <https://doc.akka.io/docs/akka/current/cluster-sharding.html>.
- [4] Checkstyle. <https://checkstyle.sourceforge.io/>.
- [5] Cluster membership service. <https://doc.akka.io/docs/akka/current/typed/cluster-membership.html>.
- [6] Domain-driven design (ddd). <https://romanglushach.medium.com/domain-driven-design-ddd-a-guide-to-building-scalable-high-performance-systems-5314>.
- [7] Gitlab ci/cd pipelines. <https://docs.gitlab.com/ee/ci/pipelines/>.
- [8] Gradle wrapper reference. https://docs.gradle.org/current/userguide/gradle_wrapper.html.
- [9] Junit. <https://junit.org/junit5/>.
- [10] Model view controller (mvc). <https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained/>.
- [11] Shadow. <https://github.com/johnrengelman/shadow>.