D-pac

- Distributed Pacman -

Manuel Bottazzi	manuel.bottazzi@studio.unibo.it
Giulia Lucchi	giulia.lucchi7@studio.unibo.it
Federica Pecci	federica.pecci2@studio.unibo.it
Margherita Pecorelli	margherita.pecorelli@studio.unibo.it
Chiara Varini	chiara.varini@studio.unibo.it

Product owner: Manuel Bottazzi

Alma Mater Studiorum - Università di Bologna
via Sacchi,3 – 47521 Cesena, Italia

Indice

1 Processo di sviluppo

Il processo di sviluppo che si è scelto di adottare per l'applicazione D-pac è quello Agile, basato su Scrum. La figura del Product Owner è ricoperta da Manuel Bottazzi.

Prima di procedere con lo sviluppo dell'applicazione, si sono pianificati 5 sprint da 20 ore e 6 meeting, nel corso di ogni sprint, si sono decisi, in itinere, i task che i membri del team avrebbe dovuto portare a termine.

Ogni sprint inizia con una discussione in merito ai task assegnati nello sprint precedente, valutando il lavoro svolto durante la settimana appena trascorsa. Successivamente, si identificano gli obiettivi e si stimano i tempi di svolgimento dei nuovi task assegnati, scelti in base al lavoro ancora da svolgere. Infine, il team compila il documento dello sprint backlog per la settimana successiva.

In Figura 1, si può vedere la pianificazione dei vari sprint, descritti in modo più dettagliato nel file Product-Backlog, presente nella cartella `process` del progetto.

Oltre al processo di sviluppo Agile Scrum, sopra descritto, si sono utilizzati:

- IntellJ IDEA come ambiente di sviluppo;
- Travis CI come "Continuos Integration Service";
- Trello come applicazione web per la gestione del progetto;
- Gradle come sistema per l'automazione dello sviluppo;
- Git come sistema di versione di controllo;
- GitHub come repository del team.

Inoltre, si è scelto di utilizzare la libreria ScalaTest come strumento principale per svolgere i test. Essi non sono stati unificati, ma riguardano funzionalità e domini separati. La copertura dei test risulta quasi totale per la parte del model e del controller dell'applicazione. Invece, per quanto concerne il sistema ad attori, presente sia nel codice Client che in quello Server, non vi sono test ma, si è comunque verificata la corretta interazione tra le parti simulando lo scambio di messaggi tra di esse e verificando, in ultimo, l'esattezza del contenuto del messaggio ricevuto.
Questi test sono tutti visibili nella cartella `src/test`.

	Distributed Pacman Product Backlog								
	Task ID	Story / Description	Priority	Estimate	Points Left After Sprint				
					1	2	3	4	5
Sprint 1	1	Implementazione struttura del game model	2	8	0	-	-	-	-
	2	Comportamento in prolog	1	8	0	-	-	-	-
	3	Implementazione di una prima versione del controller del client di una partita in single player	3	10	5	0	-	-	-
	4	GUI Versione Base (Single Player)	4	10	4	0	-	-	-
Sprint 2	5	Completamento del model e prima versione funzionante	5	10	10	2	0	-	-
Sprint 3	6	Struttura ad attori del Client	6	10	10	10	0	-	-
	7	Server : Struttura ad attori	7	25	25	25	2	-	-
	8	Model della parte di comunicazione e integrazione	10	7	7	7	2	-	-
	9	Integrazione view - controller	11	15	15	15	5	0	-
	10	Comunicazione Message Passing Client-Server	9	8	8	8	0	-	-
	11	Comunicazione Peer to Peer	8	30	30	30	5	0	-
Sprint 4	12	Server: Implementazione del comportamento	14	20	20	20	20	4	0
	13	Completamento modello MVC	15	12	12	12	12	0	-
	14	Completamento della comunicazione p2p	12	15	15	15	15	4	0
	15	Completamento della comunicazione client-server	13	10	10	10	10	0	-
Sprint 5	16	Server: Database	16	5	5	5	5	5	0
	17	Server: Automazione e miglioramenti	17	5	5	5	5	5	0
	18	Revisione e integrazione	18	10	10	10	10	10	0
	19	Test & Bug Fixing	19	10	10	10	10	10	0
				228	201	184	101	38	0
				Consumed:	27	17	83	63	38

Figure 1: Product Backlog

2 Requisiti

Lo scopo dell'applicazione consiste nel riproporre il gioco "Pacman" nella sua forma distribuita, permettendo quindi ad un gruppo di persone di giocare nella stessa partita da computer diversi.

2.1 Requisiti funzionali

I requisiti funzionali dell'applicazione sono i seguenti:

1. **registrazione e login** sull'applicazione;
2. **configurazione di una nuova partita** in cui l'utente deve impostare il range di giocatori contro cui giocare, scegliere il personaggio che vuole interpretare e selezionare il campo da gioco desiderato;
3. **visualizzazione dei punteggi** totalizzati dal giocatore nelle partite giocate precedentemente;
4. **visualizzazione di una porzione ristretta del campo da gioco** rispetto alla posizione del proprio personaggio, corredata di relativa miniatura dell'intera mappa del campo, in modo tale che sia possibile individuare in ogni momento la posizione degli altri giocatori;
5. **logout** dell'applicazione.

2.2 Requisiti non funzionali

I requisiti non funzionali dell'applicazione sono i seguenti:

1. **sicurezza**: la password inserita dal giocatore in fase di registrazione o login viene crittografata dall'applicazione e non vengono date informazioni dettagliate all'utente quando esso la inserisce;
2. **reattività dell'applicazione**: non è verificata a livello implementativo con una misura quantitativa, ma solamente a livello di percezione dell'utente. In merito a ciò possiamo dire che l'applicazione risulta abbastanza reattiva;
3. **applicazione "user-friendly"**: l'interfaccia grafica risulta molto intuitiva e di assai facile comprensione per un qualsiasi utente;
4. **perdita di connessione**: non sono state gestite situazione in cui l'utente si disconnette in maniera improvvisa dalla rete. Di conseguenza l'applicazione risulta bloccarsi;
5. **scalabilità**: l'applicazione è pensata per partite sull'ordine delle decine di giocatori e, anche se il numero di giocatori dovesse aumentare fino ad arrivare alle centinaia, secondo noi, per le tecnologie utilizzate, dovrebbe comunque risultare ugualmente performante. I test sono stati effettuati solo su un range di giocatori da 3 a 5, come riportato in seguito;

3 Design Architetturale

L'architettura dell'applicazione è progettata utilizzando i seguenti tre pattern architetturali:

- Client-Server;
- Peer to Peer;
- MVC.

3.1 Architettura Client-Server

L'architettura in Figura 2 viene utilizzata per la registrazione, il login, la configurazione della nuova partita, il logout e la visualizzazione delle partite precedentemente giocate.

Il server, inoltre, utilizza un database per la memorizzazione di tutti i dati dell'applicazione, fra cui i dati dei giocatori registrati e le partite giocate. Questo permette di eseguire velocemente operazioni come:

- controllo sulla creazione di username univoci;
- caricamento delle partite giocate in precedenza;
- creazione di team di gioco;
- gestione della scelta del campo (spiegato in dettaglio di seguito).

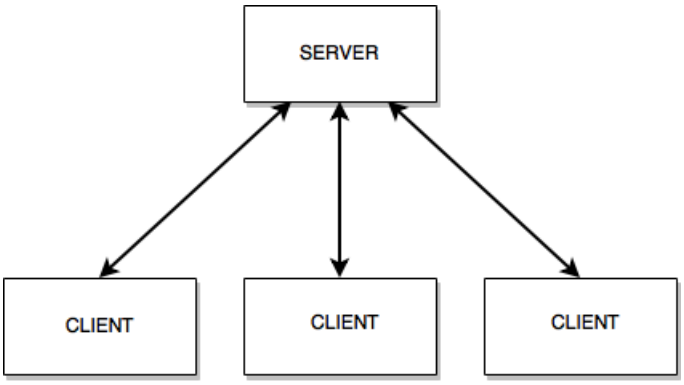

Figure 2: Pattern Architetturale: client-server

Per la comunicazione client-server si è deciso di utilizzare l'architettura ad attori e di seguito riportiamo lo schema generale.

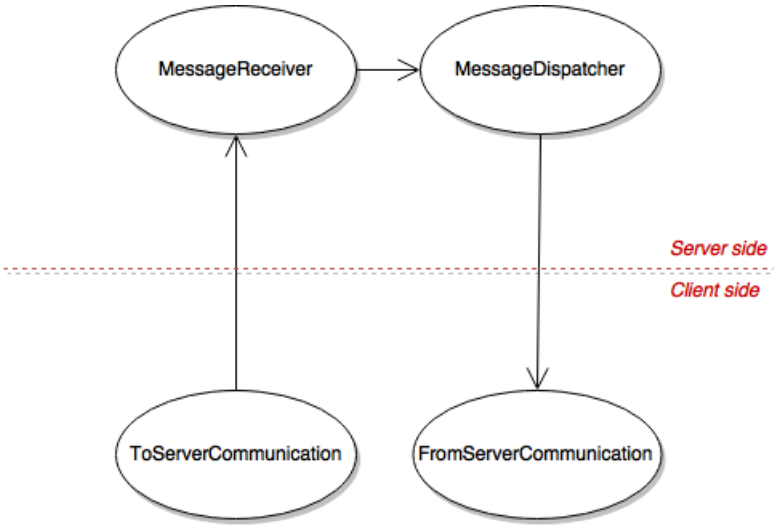

Figure 3: Architettura ad attori: client-server

L'attore `ToServerCommunication` invia un messaggio di richiesta o notifica al server. Quest'ultimo, tramite l'attore `Message Receiver` si occupa della ricezione dei messaggi inviati dal client; una volta ricevuti, li smista in base al contenuto del messaggio, reindirizzandoli all'attore corretto. Dopo aver gestito nella maniera opportuna il messaggio, l'attore `Message Dispachter` si occupa di inviare la risposta al client.
L'attore `FromServerCommunication` riceve la risposta dal server e si occupa della gestione del messaggio.

3.2 Architettura Peer to Peer

Per quanto riguarda lo scambio di dati tra i vari dispositivi partecipanti ad una stessa partita, si è adottata un'architettura Peer to Peer (vedi Figura 4) in quanto si ritiene che essa si adatti bene al contesto di questa applicazione, le cui necessità sono:

- ottime prestazioni relative alla scalabilità del sistema nel caso questa debba sostenere il carico di molti dispositivi;
- evitare l'utilizzo di un server che, potenzialmente, causerebbe problemi nel caso si verifichi un guasto durante una partita ed i giocatori (client) non potessero portarla a termine a causa di un server inagibile.

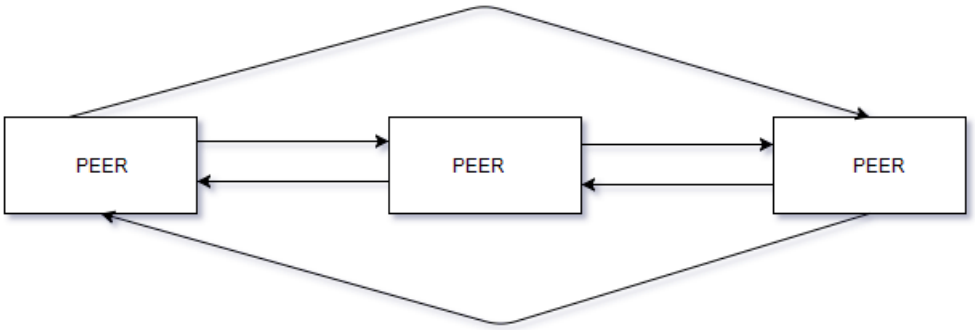

Figure 4: Pattern Architetturale: Peer to Peer

Come rappresentato in figura 5, ogni peer si compone di tre entità:

- `P2PCommunication`: si occupa della configurazione del peer, prima per quanto riguarda la parte server, poi per quella client;
- `ServerPlayingWorkerThread`: contiene le informazioni sempre aggiornate di se stesso;
- `ClientPlayingWorkerThread`: reperisce presso gli altri peer le loro informazioni aggiornate.

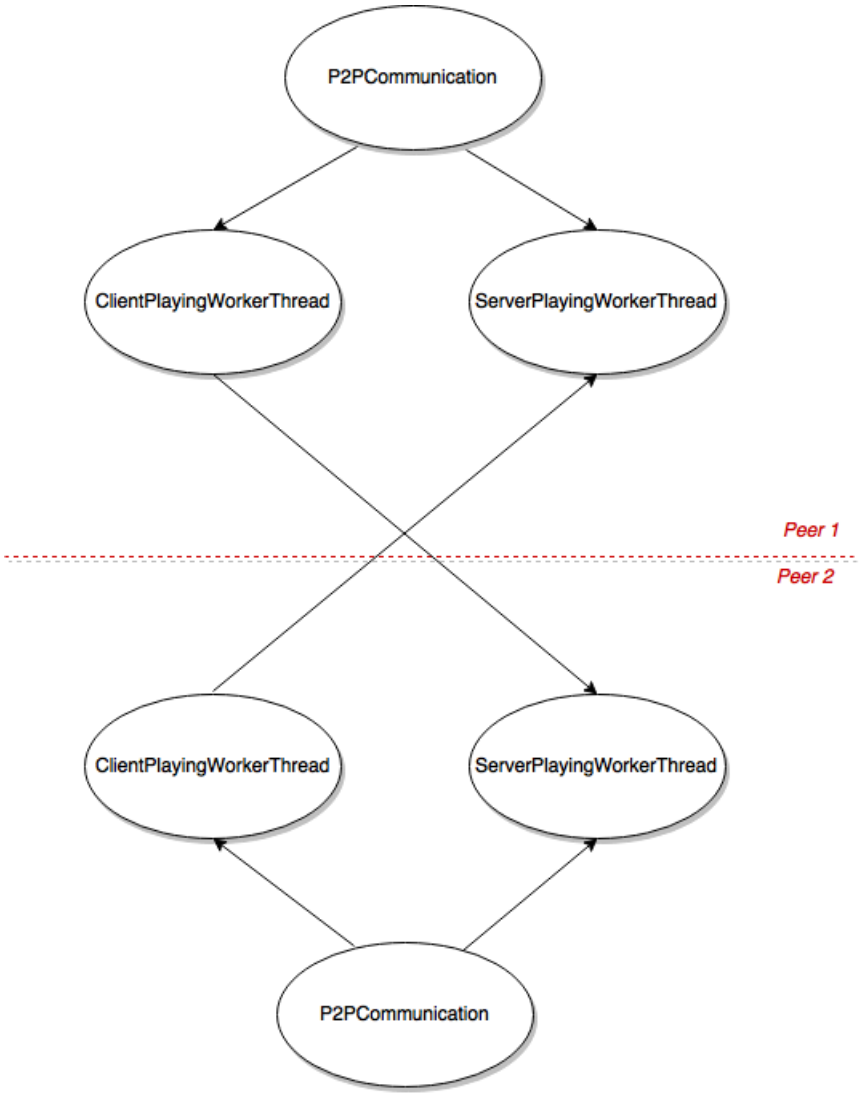

Figure 5: Architettura Peer to Peer

3.3 Architettura MVC

Nel codice relativo all'applicazione si utilizza il pattern architetturale MVC (Figura 6), che separa la logica di gioco, compresa nel model, dalla logica di presentazione, racchiusa nella view, e dal coordinamento dei due, gestito dal controller. L'utente si interfaccia al gioco tramite la view, la quale comunica con il controller. Quest'ultimo aggiorna lo stato del model, che a sua volta lo notifica di ogni suo cambiamento. Quando il controller viene notificato, aggiorna lo stato della view.

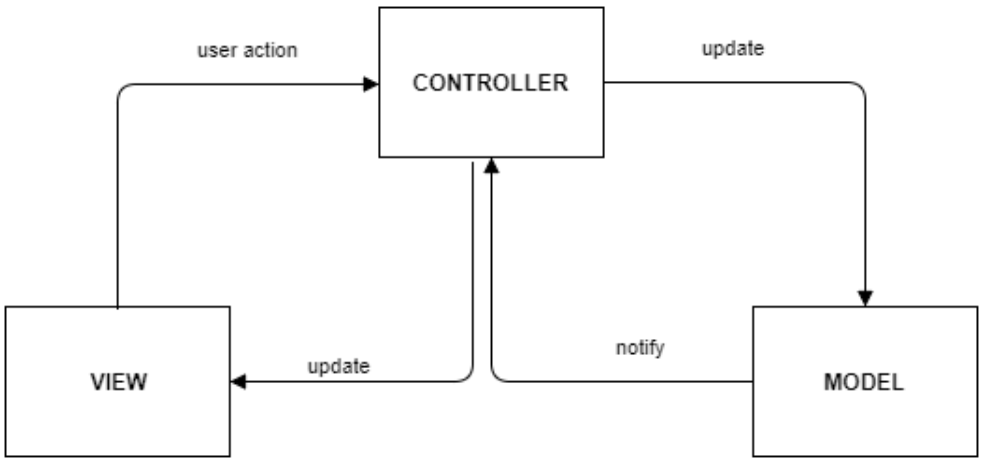

Figure 6: MVC - diagramma generale

3.3.1 Model

Il modello si occupa della logica del gioco e della gestione delle sue principali entità, come rappresentato in figura 7:

- i personaggi;
- la partita;
- il giocatore;
- il campo da gioco, composto dai vari elementi.

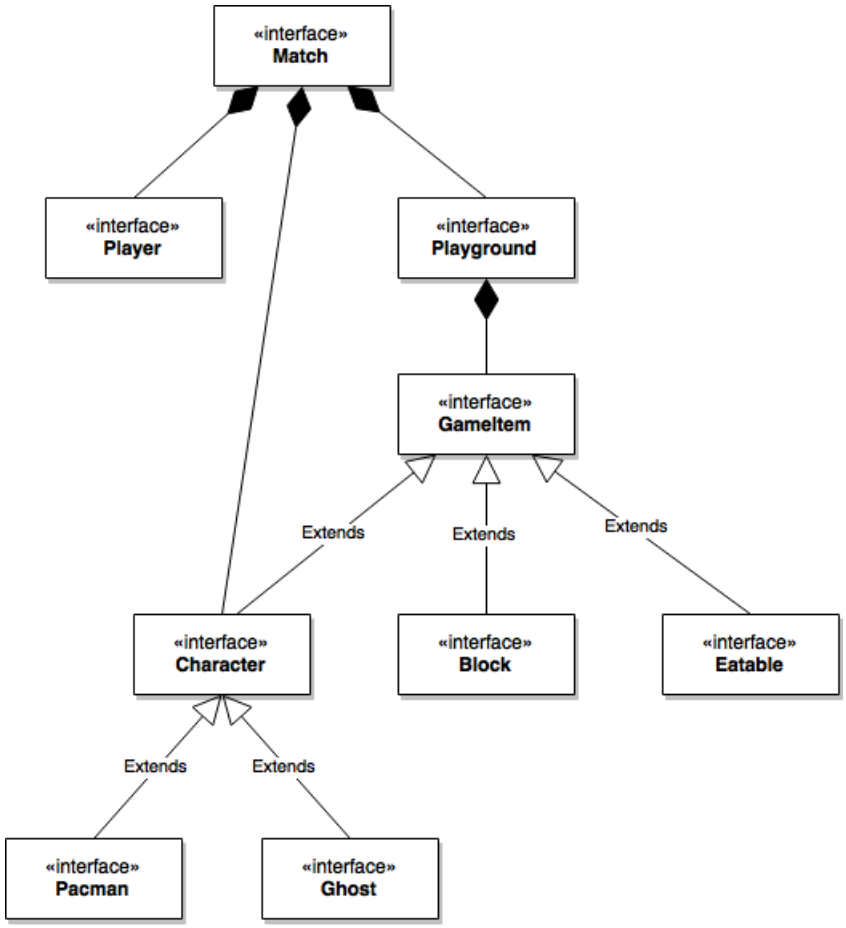

Figure 7: MVC - model

3.3.2 Controller

Come già anticipato, il controller si occupa del coordinamento fra view e model. Le principali funzionalità da gestire sono i personaggi, l'utente e la partita, per questo il controller viene suddiviso nel seguente modo:

- `ControllerCharacter`;
- `ControllerUser`;
- `ControllerMatch`.

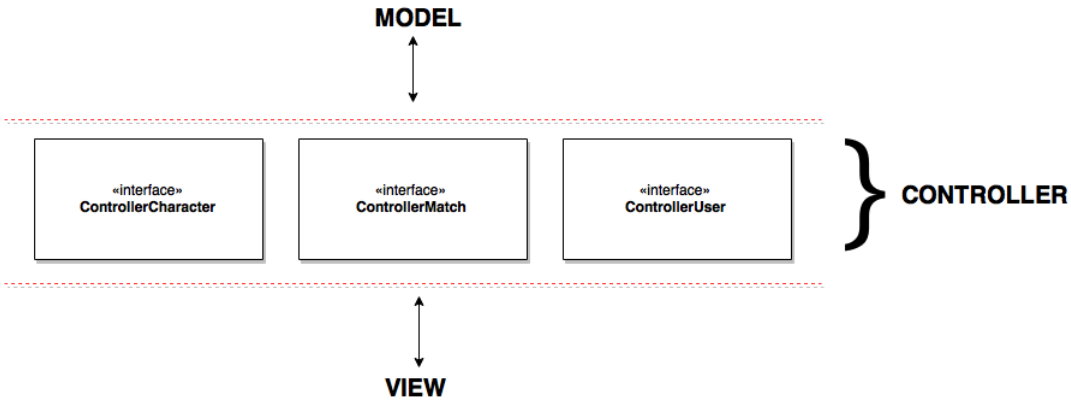

Figure 8: MVC - controller

3.3.3 View

La view è responsabile di:

- gestione dell'interfaccia grafica, in modo da renderla user-friendly;
- gestione del campo da gioco: si è deciso di rappresentare una sola porzione di campo così da ottenere dimensioni adatte per garantire all'utente un esperienza di gioco positiva. Di conseguenza, la porzione di mappa visualizzata segue continuamente i movimenti del personaggio usato dal giocatore. Inoltre per avere una visione complessiva del campo, si è deciso di dedicare una piccola porzione dello schermo alla mappa generale di gioco.

4 Design di dettaglio

Per lo sviluppo di questo progetto si è suddiviso il codice in due parti distinte:

- la parte Client, organizzata in modo tale da seguire il più fedelmente possibile il pattern MVC. Quest'ultimo intento è visibile nella costruzione dei package all'interno del progetto.
- la parte Server, gestita principalmente con il sistema ad attori;

I linguaggi utilizzati nell'implementazione del codice sono:

- `Prolog`, per l'implementazione della logica del gioco, data la natura stessa del linguaggio. Questa scelta ha permesso di esprimere in maniera veloce e concisa le regole basilari del gioco, grazie ai costrutti forniti. Infine, risulta anche alleggerita l'implementazione delle classi del model, in quanto si limitano a richiamare dei predicati Prolog senza dover implementarne tutta la logica;
- `Java`, per l'implementazione dell'interfaccia grafica e della parte di comunicazione Peer to Peer. In particolare, si sceglie questo linguaggio per l'implementazione della view in quanto le API fornite dalla libreria scala-swing non contengono i wrapper per elementi come il *JLayerdPane* e il *JDialog*.
- `Scala`, per l'implementazione del controller, del model del codice Client e dell'intera parte server. Scala è il linguaggio principale in quanto:
  - è necessario utilizzare un linguaggio OO per strutturare in maniera solida il progetto;
  - sono utili le feature funzionali per velocizzare l'implementazione del codice e renderlo più leggibile.

I pattern progettuali utilizzati nell'applicazione sono:

- Template Method: nel metodo `go(direction: Direction)` della classe astratta `CharacterImpl` si è richiamato il metodo astratto `checkAllPosition()`, il quale verrà poi implementato nelle sottoclassi, ossia `BasePacman` e `BaseGhost`;
- Singleton: le classi `MainFrame`, `ExecutorServiceUtility` (vedi Figura 14), `ServerPlayingWorkerThread` (vedi Figura 15) restituiscono un'unica istanza dell'oggetto. Inoltre si sono utilizzati degli object così da sfruttare la proprietà di singleton intrinseche dell'object stesso;
- Observer: di seguito vengono riportate le classi Observable con le rispettive classi Observer

Observable	**Observer**
`ControllerObservable`	`ControllerMatch`
`ClientOutcomingMessageHandlerImpl`	`ControllerMatch`
`ClientPlayingWorkerThread`	`ControllerCharacter`

- Strategy: la classe `BaseEatObjectStrategy` implementa la strategia da passare a Pacman e da utilizzare quando quest'ultimo mangia gli oggetti sparsi nel campo di gioco. La classe `PlaygroundPanel` utilizza il metodo `lookAt(list, strategy)` per caricare l'immagine giusta del labirinto. In questo caso la strategia passata è un'espressione booleana relativa alla posizione di un blocco del labirinto rispetto a quelli adiacenti.

- Builder: visibile nel package `src.main.java.client.view.playground`, usato per la creazione dei pannelli per i campi da gioco.

4.1 Scelte tecnologiche

- Akka: tecnologia scelta per l'implementazione della parte di comunicazione server-client, in quanto, tra quelle conosciute dal team, risulta la più adatta per lo svolgimento di questo compito e la più coerente con i linguaggi utilizzati per il progetto;

- Remote Method Invocation (RMI): tecnologia scelta per realizzare l'interazione tra i vari giocatori partecipanti ad una stessa partita. Questo framework supporta una gestione ad alto livello dei meccanismi di comunicazione tra computer connessi ad una stessa rete, astraendo da aspetti di mera implementazione, evitando quindi l'uso delle socket affinché sia possibile concentrarsi esclusivamente sui requisiti specifici di design dell'applicazione;

- XAMPP: tecnologia scelta per la creazione e gestione del DB utilizzato dall'applicazione;

- Balsamiq: per un primo studio, finalizzato alla creazione di un gioco facile e piacevole da utilizzare.

5 Implementazione

5.1 Manuel Bottazzi

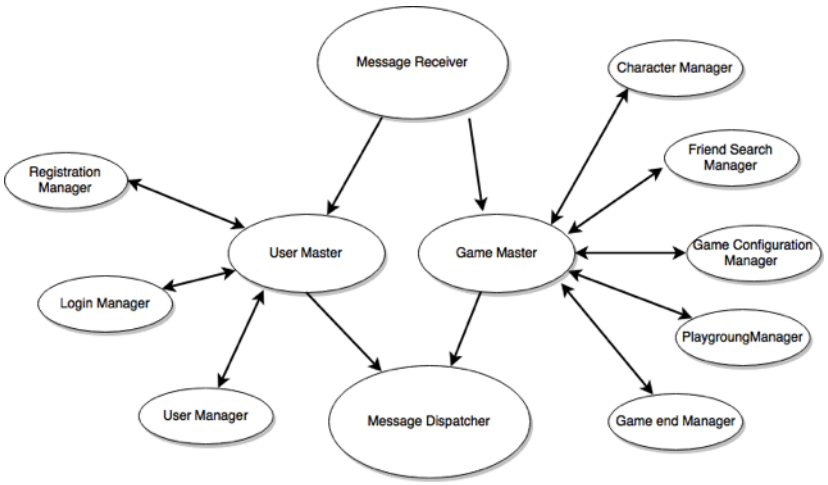

Figure 9: Organizzazione sistema ad attori - server side

La parte sviluppata principalmente riguarda il server e le sue funzionalità. Inoltre si sono sviluppate alcune funzionalità singole nella parte di model e di view del Client.

Rispetto al Server abbiamo deciso che esso venga interamente gestito e sviluppato con un sistema ad attori per favorire la gestione della concorrenza e della comunicazione. Come tecnologia per sviluppare questo abbiamo usato le librerie `core` e `remote` di Akka.

- `src/main/scala/actors`: un package che contiene gli attori che gestiscono il funzionamento del server. La struttura interna di questi ultimi è stata realizzata seguendo lo schema in Figura 9.

  Tale suddivisione è stata progettata in maniera tale da semplificare la gestione dei messaggi ed aumentare la leggibilità, fornendo anche una struttura logica allo schema ad attori.

  Per comodità e uniformità abbiamo gestito tutti i messaggi, sia quelli per la comunicazione remota che quelli interni al sistema stesso, tramite messaggi JSON.

  Di seguito, vengono esaminati i compiti svolti dai principali attori:

  - `messageReceiverActor`: un attore che ha il compito di ricevere i messaggi in arrivo dai vari client e di redirigerli ai vari attori dell'actor system a seconda del tipo del messaggio;

- `messageDispatcherActor`: un attore che ha il compito di ricevere i messaggi provenienti dal sistema locale e di spedirli ai client remoti. Per fare questo ad ogni messaggio è stato associato l'IP del client da cui esso proviene per riuscire a distingure tra di loro i vari client. Per motivi di comodità ed efficienza questo attore gestisce anche la lista degli utenti online sul server e la lista delle partite attualmente attive. Infatti queste informazioni vengono usate quasi esclusivamente in questa classe, a parte in 1 o 2 casi, e quindi sono stati posizionati qui. Inoltre questo attore è in grado di gestire diversi tipi di messaggi a seconda di quale attore del client debba ricevere i messaggi. I tre tipi principali sono :
  - `remoteMessage`: normali messaggi di risposta ad una richiesta del client, comunicate all'Inbox del client;
  - `configurationMessage`: messaggi di configurazione inviati ad un attore specifico per la configurazione della comunicazione peer-to-peer;
  - `notificationMessage`: messaggi che partono dal server, senza una specifica richiesta del client, usati per notificare qualche genere di evento. vengono inviati ad un apposito attore nel client.
- `userMasterActor e matchMasterActor`: due attori speciali che permettono di ottimizzare la distribuzione e la gestione dei messaggi ai singoli attori. Essi sono infatti gli unici attori che comunicano con i due visti in precedenza che si occupano della comunicazione e smistano i messaggi agli attori locali, loro figli. Gli attori stessi infatti sono suddivisi per aree tematiche, tutti quelli che svolgono funzioni legate alla configurazione di una partita dipendono da `matchMasterActor`, mentre tutti quelli legati alla gestione degli utenti (registrazione, login, ecc.) dipendono da `userMasterActor`;
- `RegistrationManagerActor`: un attore che gestisce la registrazione di un nuovo utente sul sistema. questo attore andrà anche ad inviare la richiesta all'attore che gestisce il database per aggiungere il giocatore ad esso;
- `LoginManagerActor`: un attore che gestisce il login di un utente ed effettua i controlli sulle credenziali inserite (sempre interfacciandosi con il databaase). Se il login ha successo verranno restituite al client anceh la lista dei suoi risultati precedenti;
- `DatabaseManagerActor`: un attore che gestisce il database del server. Tutte le richieste di inserimento o di query sul database passano attraverso questo attore;

  Questi primi tre attori rappresentano la parte legata alla gestione dell'utente. I prossimi invece rappresentano la parte di gestione di configurazione della partita.
- `CharacterManagerActor`: un attore che gestisce tutte le richieste legate ai personaggi del gioco. In particolare:

    - Fornisce al client la lista dei personaggi disponibili per la scelta;
    - Gestisce la scelta dei personaggi da parte del client e garantisce la scelta univoca controllando la disponibilità prima di assegnare un personaggio;
    - Fornisce le risorse grafiche per la visualizzazione del personaggio scelto e degli altri giocatori in partita.
  - `PlaygroundManagerActor`:l'attore che gestisce la parte relativa ai campi di gioco. Esso gestisce la richiesta delle immagini dei campi di gioco per la scelta e la spedizione del file relativo al campo di gioco scelto. Infatti poi nel client tale file verrà elaborato e trasformato in un Playground. Questo attore gestisce anche la votazione del campo da gioco da parte dei vari client, gestendo un eventuale pareggio scegliendo il campo scelto da Pacman. tale strategia è stata concordata in fase di design;
  - `GameConfigurationManagerActor`:l'attore che gestisce la configurazione e la scelta dei una partita da parte di un giocatore. in particolare;
    - Fornisce al client la lista dei range disponibili. è infatti possibile scegliere una serie di range di giocatori con cui giocare. Nella versione attuale del gioco il client utilizza soltanto il range 3-5 giocatori, ma non ci sarebbero controindicazioni all'utilizo di range più grandi. Andrebbero semplicemente aggiunte risorse grafiche per ulteriori personaggi;
    - Gestisce l'aggiunta di un giocatore ad una partita in base al range selezionato;
    - Gestisce inoltre la configurazione dei client nella fase della configurazione della comunicazione peer-to-peer. Per fare questo aspetta che tutti i client per una certa partita abbiano inviato il messaggio che indica che il loro server peer-to-peer è attivo e configurato, e a quel punto notifica tutti i client per quella partita affiché possa iniziare.
  - `GameEndManagerActor`:l'attore che gestisce la fine di una partita e il salvataggio dei risultati di una partita interfacciandosi con il database;

- `src/main/scala/test`: un package che contiene le classi necessarie ad un test locale del funzionamento del server. Esse infatti rappresentano un piccolo client che invia messaggi al server e stampa la risposta che riceve. Queste classi sono state usate per motivi di debug e test.

- `src/main/scala/utils`: un package che contiene le classi di utility per la gestione del server stesso. Le classi Utils e IOUTils sono state importate direttamente dal client.
  - `ActorsUtils`: usata per semplificare la gestione dei messaggi in formato JSON all'interno degli attori del server. In particolare viene usato il metodo messageType, che semplifica l'estrazione dell'object dei messaggi per il match-case del metodo onReceive degli attori;

  - `Timer`: una classe sviluppata in collaborazione con Giulia Lucchi, permette di gestire un timer che aspetta un certo intervallo di tempo in maniera non bloccante;

- `src/main/scala/model`:
  - `Character`: una classe che rappresenta un personaggio del gioco sul server. Per ogni personaggio vengono memorizzate le risorse relative alla sua visualizzazione rispetto alle varie direzioni, e inoltre offre la possibilità di memorizzare l'ip del giocatore che ha scelto quello specifico personaggio in una certa partita;
  - `Client`: un'interfaccia che rappresenta un client connesso al server. Esso è identificato dalla coppia username-ip;
  - `User`: un'interfaccia che rappresenta un utente sul server, essa contiene tutte le informazioni relative ad un certo utente, quali name, username, mail e password;
  - `Match`: classe che rappresenta una singola partita e il suo stato. Questa classe è utile infatti in fase di configurazione ed inizializzazione per gestire i giocatori connessi ad una certa partita;
  - `MatchResult`: un'interfaccia usata per gestire il risultato di una partita ricevuto dal client e successivamente salvato nel database.

- `src/scala/Main.scala`: una classe che rappresenta il Main del server stesso. Essa gestisce la configurazione del server per la configurazione settando in maniera automatica l'IP dell'host e inizializzando gli attori.

Nella parte di client ho contribuito a diverse funzionalità singole e sviluppato alcune classi, le cui principali sono:

- `IOUtils`: una classe di utility utilizzata per creare un istanza di un Playground facendo un'operazione di parsing su di un file testuale che permette all'utente di specificare la struttura del suo campo da gioco e di crearne di personalizzati. Contiene inoltre un metodo per il salvataggio su file dei log;

- package `main.java.client.view.playground`: In collaborazione con Chiara Varini abbiamo realizzato una prima versione delle classi che visualizzano il campo da gioco principale per la partita. La classe principale è un pannello che visualizza il contenuto di un Playground renderizzando i suoi componenti. In questo package abbiamo anche applicato il pattern builder per la creazione del pannello che contiene il campo da gioco. durante lo sviluppo di questa parte abbiamo anche cercato e sistemato le risorse grafiche utilizzate per la costruzione del campo da gioco, che si trovano in `src.main.resources.images`. Io in particolare ho creato le immagini relative ai block che costruiscono i muri del campo da gioco;

- packages `src.main.scala.client.model` e `src.main.scala.client.model.gameElement`: In collaborazione con Margherita Pecorelli ho realizzato una prima versione delle classi del modello che gestiscono gli elementi del gioco, che poi lei ha completato ed esteso;

  In tali package sono presenti la classe `Playground` che rappresenta un campo da gioco con tutti i suoi elementi ed una serie di classi ed intefacce che gestiscono ognuna uno degli elementi del gioco, dai componenti del muro (interfaccia `Block`) agli elementi mangiabili da parte di Pacman (interfaccia `Eatable` e sue implementazioni).

5.2 Giulia Lucchi

Il contributo nell'implementazione della parte client è stato principalmente nella parte di progetto che va a gestire il message passing e quindi la parte dell'architettura client-server, attraverso il paradigma ad attori. Oltre a ciò, ho collaborato anche per l'implementazione del modello del progetto, che comprende l'interazione con prolog e l'organizzazione del codice per la gestione dei personaggi.
Di seguito si potrá vedere in dettaglio il codice su cui ho lavorato:

- `src/main/java/Direction`: enumerazione scritta in java per gestire le direzioni dei personaggi nei movimenti. La scelta di utilizzare le enumerazioni di java è stata presa dopo un attiva ricerca di informazioni sulle enumerazioni in scala, valutate negativamente,quanto ti permettono di avere una semplice raccolta di valori rimanendo difficile l'inserimento di metodi e campi.

- `src/main/scala/client/communication/model`: In questo package sono presenti tutte le classi coinvolte per la gestione del message passing. É possibile visualizzare l'organizzazione interna del sistema ad attori grazie al seguente diagramma in figura 10.

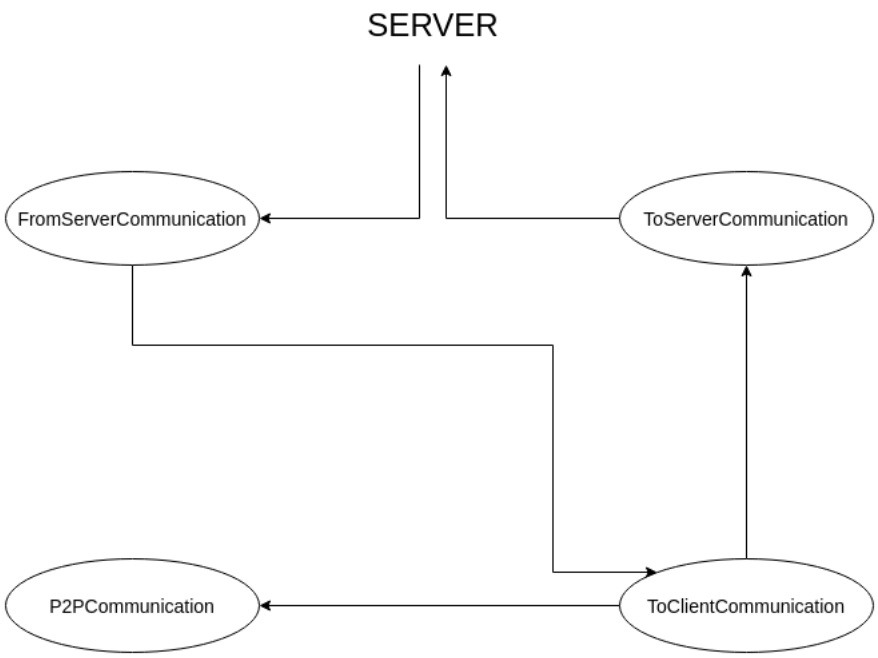

Figure 10: Organizzazione sistema ad attori - client side

I messaggi che si inviano questi attori sono tutti `JSONObject`. Questa è stata una decisione presa in accordo con la parte server per dare omogeneità e coesione al codice.

- `ToClientCommunication`: trait composto da metodi che gestiscono lo scambio di messaggi con il server.

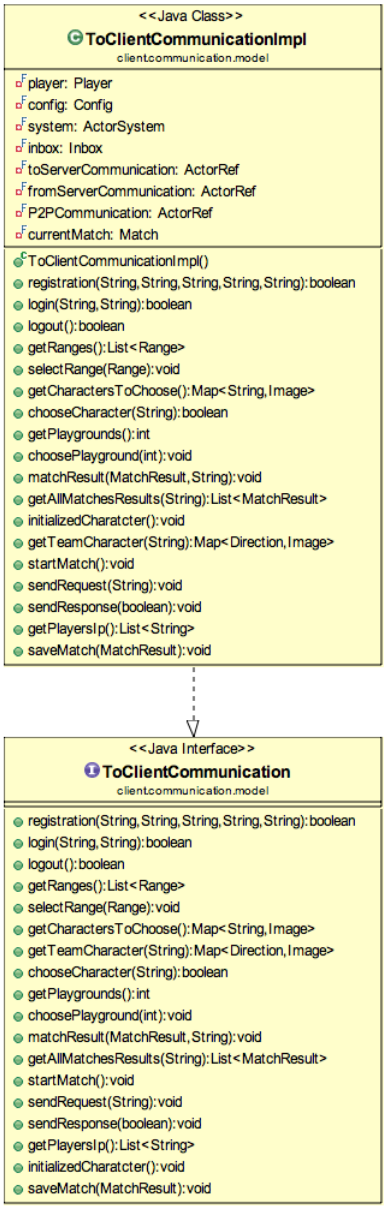

Figure 11: classe ToClientCommunicationImpl

– `ToClientCommunicationImpl`: implementazione dell'interfaccia sopra indicata, la quale viene richiamata dal controller con lo scopo di mandare e ricevere i messaggi dal server, in base alle necessità dell'applicazione. Il contenuto dei messaggi è facilmente intuibile dal nome dei metodi della classe, visibili nell'UML in figura 11.

Ho deciso di utilizzare la classe `Inbox` della libreria `akka.actor.dls.inbox` per riuscire a far integrare il paradigma ad oggetti con quello ad attori. La seguente classe infatti permette di creare un nuovo attore che attraverso particolari metodi `send()` e `receive()` può interrogare gli altri attori, evitando però di dover estendere UntypeAbstractActor per creare un vero e proprio attore.
Nel costruttore della classe in questione avviene la configurazione del `akka.actor.ActorSystem` della parte client e i relativi attori utilizzati per lo scambio di messaggi, mostrati in figura 10.

- `ToServerCommunication`: attore che ha il compito di mandare i messaggi al server. Per fare questo, l'attore comunica con il server selezionandolo attraverso la rete, tramite l'indirizzo ip, la porta, l'Actor System del server e nome dell'attore principale di ricezione dei messaggi nel server, attraverso il seguente codice:
  `context actorSelection`
  `äkka.tcp://DpacServer@192.168.1.17:2552/user/messageReceiver"`
- `FromServerCommunication`: attore che riceve i messaggi direttamente dal server. Le funzioni principali di tale attore consistono: in primo luogo nel ricevere i messaggi dal server e ridirigerli all'`Inbox` della classe `ToClientCommunicationImpl` così da riuscire a mandare al controller le informazioni che ha richiesto, ma anche nel notificare al controller e di conseguenza alla view di alcuni messaggi provenienti dal server.
  Questo viene fatto tramite la creazione della classe `ControllerObservable` che, estendendo la classe `java.util.Observable`, ha la funzione di Observable nell'attore.
- `P2PCommunication`: attore che comunica con la parte dell'architettura peer to peer. Esso entra in gioco tramite il messaggio mandato dall' `Inbox` della classe `ToClientCommunication` che lo notifica la fine della configurazione della partita e l'inizio del gioco stesso.
  Per l'implementazione di questa classe ho collaborato con la mia collega Federica Pecci, creando la gestione di base dei messaggi, ossia la creazione del messaggio stesso e l'interazione con il server, lasciando a lei la configurazione dell'architettura peer to peer.

- `src/main/scala/client/model/character`:
  - `Character`, `CharacterImpl`, `Ghost`, `Pacman`: questi trait e classi sono state implementate in collaborazione con la mia collega Margherita Pecorelli. In particolare, durante il secondo sprint ho creato da zero le classi, decidendone l'organizzazione e l'implementazione. Margherita ha preso in mano il codice a posteriori modificandolo in itinere e adattandolo gradualmente agli sviluppi fatti.
    La spiegazione in dettaglio dell'implementazione è spiegata in un modo esaustivo nella sottosezione, chiamata "Margherita Pecorelli"

  - `InizializedInfo`:trait gestisce l'estrazione delle informazioni dalla teoria scritta in prolog.
  - `InitializedInfoImpl`: object che implementa l'interfaccia `InizializedInfo`. Per effettuare l'estrazione delle informazioni dalla teoria di prolog è stato utilizzato un object `PrologConfig` che andrò a descrivere in seguito. Questo object gestisce in particolare le vite dei giocatori e le posizioni di partenza dei personaggi.

- `src/main/scala/client/model/utils`:
  - `Lives`: trait che rappresenta le vite dei personaggi nell'applicazione. Questo utilizzato in particolare per pacman.
  - `LivesImpl`: case class che implementa l'interfaccia `Lives`. Per gestire le vite abbiamo inserito setter e getter per recuperare il numero delle vite. Oltre ciò abbiamo aggiunto il decremento di una vita o di un numero di vite pari al parametro in ingresso al metodo
  - `PrologConfig`: object che gestisce l'interazione tra prolog e scala. In particolare viene istanziato un oggetto `Prolog`, al quale viene settata la teoria, riguardante la logica dell'applicazione. Inoltre sono stati inseriti un metodo che aggiungesse teoria a quella settata all'inizio dell'applicazione e metodi di conversione per le strutture dati da prolog a scala e viceversa.

- `src/main/scala/client/utils`:
  - `ActorUtils`: object che incapsula tutti i percorsi degli attori, per rendere più leggibile il codice, scritto nel package `communication`. Inoltre ho aggiunto il campo `serverIP` in modo da settarlo come input durante l'esecuzione dell'applicazione.
  - `ControllerObservable`: object che implementa un observable. I suoi metodi sono tutti `update()` all'Obsever in questione a cui manda il messaggio. Viene usato per aggiornare il controller dei messaggi del server, passati agli attori.

- `src/test/scala/client/communication/ActorTest`: object eseguibile che implementa il test della comunicazione fra client e server, fatto tramite stampe e ScalaTest.

Nella parte server invece ho gestito la parte riguardante il database e la connessione, dopo una scelta comune della tecnologia. Per motivi di tempo, ma principalmente della poca mole di dati non abbiamo reso la parte di interazione con i dati molto performante, essendo fuori dall'ambito di valutazione del progetto.
Ho infatti utilizzato principalmente un semplice database relazionale MySQL, gestito in locale con PHPmyAdmin. supportato dalla piattaforma di servizi XAMPP. Non abbiamo inserito il diagramma del database per la semplicità di questo composto da due tabelle con una relazione uno a molti, atte a gestire

partite di un giocatore e i dati del giocatore stesso.
Per accesso al database abbiamo utilizzato `JDBC`, ossia un driver che consente l'accesso e la gestione della persistenza dei dati.
Le classi coinvolte sono le seguenti:

- `src/min/scala/database`:
  - `ScalaJdbcConnect`: object che gestisce solamente la connessione con il database al quale le altre classi si riferiscono.
    Come si può notare dall'hostname del database abbiamo deciso di crearlo locale alla macchina che verrà utilizzata da server.
  - `DatabaseQuery`: object che utilizza la connessione creata in `ScalaJdbcConnect` per interrogare il database. I metodi che compongono questa classe sono metodi utilizzati per le specifiche richieste del server. Viene gestita la memorizzazione degli utenti per poter consentire all'applicazione registrazione e login e delle partite giocate per ogni giocatore, ma anche la verifica delle informazioni che restituiscono gli errori del login, che per motivi di sicurezza sarebbe meglio non far visualizzare sull'interfaccia grafica, come abbiamo deciso di fare.
  - `DatabaseTest`: object eseguibile che testa il funzionamento della connessione e delle interrogazioni al database. Viene testato tramite assert del ScalaTest e visualizzazione dei record nel database.

5.3 Federica Pecci

- `src/main/resources/prolog/`
  - `logic`: un file scritto in prolog in cui viene definita la logica del gioco, realizzato in collaborazione con Margherita Pecorelli e Chiara Varini.
    Gli aspetti logici implementati riguardano:
    * la gestione delle posizioni dei personaggi attraverso due fatti: `pacman_initial_position(3,2).` e `ghost_initial_position(15,5).` ed un predicato: `next_position(-XPosition, -YPosition,+ NewXPosition, +YPosition).`;
    * la gestione del movimento dei personaggi attraverso il predicato: `move(+XCurrentPosition, +YCurrentPosition, +Direction, -NewXPosition, -NewYPosition).` Questo predicato verifica se nella direzione in cui si vuole spostare il personaggio è presente una strada (utilizzando anche il predicato `street(X,Y)`).;
    * la gestione delle vite dei personaggi attraverso due fatti: `pacman_lives(3).` e `ghost_lives(1).`;
    * la gestione del comportamento di Pacman quando mangia un elemento attraverso il predicato: `eat_object(+pacman(X,Y,_,Score), +EatableObjectList, -NewScore, -EatenObjectId).` Questo predicato associa un elemento `pacman(XPosition, YPosition,`

`Lives, Score)` con una lista di elementi `eatable_object(XPosition, YPosition, Value, Name)` e, nel caso in cui la posizione di uno di quest'ultimi coincida con quella di Pacman nella variabile di output verrà inserito il `newScore` del giocatore e l'id dell'oggetto mangiato, ossia il `Name`;

* la gestione della vittoria di Pacman e/o dei fantasmi attraverso i predicati:
  `pacman_victory(+pacman(_,_,Lives,_), +ListOfDots).` Pacman vince se ha almeno una vita e non ci sono più dots da mangiare
  `ghosts_victory(pacman(_,_,0,_)).` I fantasmi vincono quando Pacman non ha più vite;
* la gestione della sconfitta di Pacman e/o dei fantasmi attraverso i predicati:
  `eat_pacman(+pacman(X,Y,Lives,_), +GhostList, -NewPacmanLives, -NewGhostScore, -GhostName).` Se il ghost è nella stessa posizione di Pacman allora viene restituito il nuovo numero di vite di Pacman, il nuovo punteggio del fantasma e il nome dello stesso. Questo predicato viene richiamato a fronte di ogni spostamento. `ghost_defeat(+pacman(X,Y,_,Score), +GhostList, +NumberOfGhostEaten, -NewPacmanScore, -ListOfEatenGhostsColor).` Richiamato per dieci secondi una volta che Pacman ha mangiato un *Pill.*;

I fatti riguardanti le strade e gli eatableObjects vengono aggiunti alla teoria di prolog una volta che i giocatori hanno deciso il campo da gioco.

- `src/main/scala/client/communication/model/actor/`
  - `P2PCommunication`: classe realizzata in collaborazione con Giulia Lucchi e che rappresenta un attore con il compito di coordinare le varie fasi necessarie alla configurazione ottimale del Peer. Tale scopo è raggiunto mediante uno scambio di messaggi, il cui fulcro si trova appunto racchiuso in questa classe, e che coinvolge tutti i Client partecipanti alla partita ed il Server generale. La configurazione si compone principalmente di due fasi: i) il bootstrap della parte server (si vedrà nella classe `ServerPlayingWorkerThread` come esso avvenga nel dettaglio) del Peer che, una volta terminato, è seguito dall'invio di un messaggio al Server notificandolo che l'operazione è andata a buon fine. Il Server, a sua volta tiene memoria del messaggio ricevuto e attende che anche tutti gli altri Peer gli inviino tale messaggio. ii) il Server, ricevuti i messaggi da tutti i Peer, invia a sua volta un messaggio ad ognuno di essi notificandoli che possono avviare il bootstrap delle loro parti client (si vedrà nella classe `ClientPlayingWorkerThread` come esso avvenga nel dettaglio). Una

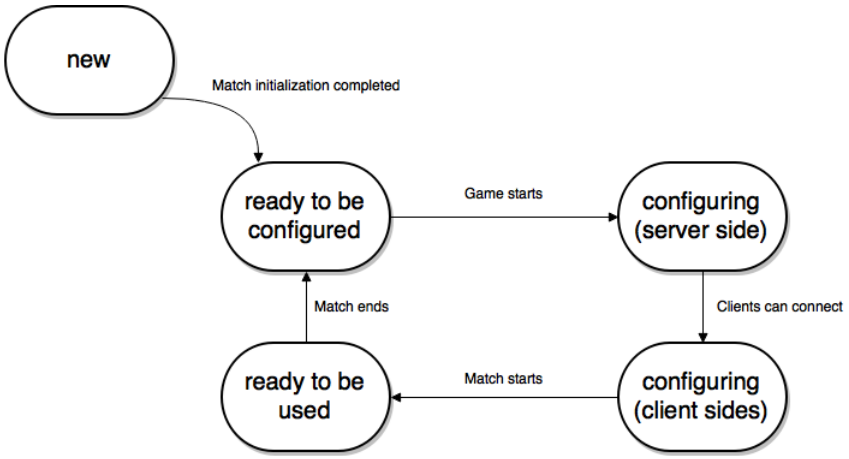

Figure 12: Diagramma raffigurante i vari stati assunti da un Peer durante la sua configurazione prima dell'inizio di una partita

visione d'insieme di quanto appena illustrato la si può avere osservando il diagramma degli stati in Figura 12;

- `src/main/java/p2pNetwork/game/`: contiene le classi e l'interfaccia cruciali per l'implementazione del meccanismo di scambio dei dati tra Peer. Considerando che un requisito del sistema consiste nel permettere ad ogni Peer di accedere ai dati degli altri Peer presenti nella rete e partecipanti alla stessa partita, si sceglie di adottare la tecnologia RMI, ossia Remote Method Invocation.
  - `PeerRegister`: un'interfaccia dove vengono definiti i metodi che possono essere invocati, da remoto, sugli oggetti relativi ad ogni Peer connesso alla rete durante lo svolgimento di una determinata partita;
  - `ServerPlayingWorkerThread`: una classe che si occupa della configurazione effettiva del Peer lato server. Avendo optato per la tecnologia RMI, si crea un registro (usualmente chiamato "rmiregistry") sull'host locale e si esportano su quest'ultimo gli oggetti necessari alla comunicazione tra i Peer. Nel caso specifico gli oggetti esportati sui registri sono relativi a:
    * posizione del giocatore;
    * stato del giocatore, il quale può essere vivo o morto.

    Questa classe, deve anche implementare i metodi definiti nell'interfaccia di cui sopra, i quali vengono poi invocati sugli oggetti contenuti nel registro. In questo caso, si è fatto uso del pattern Singleton (vedi Figura 15);

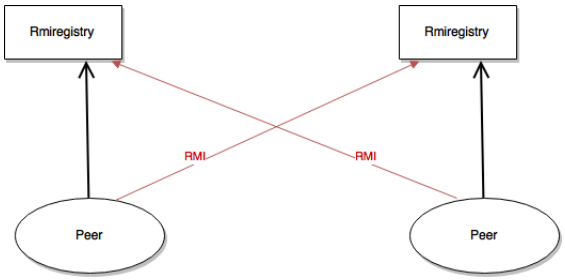

Figure 13: Remote method invocation

  - `ClientPlayingWorkerThread`: una classe che gestisce la configurazione effettiva del Peer lato client. Essendo necessario associare ad ogni processo client un riferimento al registro del processo server su cui il client è in ascolto, si ha che il numero di istanze di questa classe corrisponde al numero totale di *Peer - 1*. Infatti, ogni processo client ha l'indirizzo IP di un Server come parametro in ingresso nel costruttore. Perciò, il compito di ogni istanza di questa classe è quello di ricavare dal registro su cui il client è in ascolto le informazioni relative ad un altro Peer e che sono reperibili sul registro ad esso associato, per poi passarle al controller del gioco che si occupa di aggiornare il model e la view con i dati correnti. La maniera in cui è possibile ricavare le informazioni relative ad un altro Peer è quella di effettuare delle chiamate a metodo sugli oggetti contenuti nel suo registro.

  La Figura 13 mostra il funzionamento della tecnologia RMI applicata ad un Peer.

- `src/main/java/p2pNetwork/utils/`
  - `ExecutorServiceUtility`: una classe per gestire in maniera efficiente i *task* relativi alla configurazione dei Peer; in particolare, si è scelto di utilizzare gli `Executors` che hanno come implementazione `Thread Pools` (vedi Figura 14);
- `scr/main/scala/client/model/peerCommunication/`
  - `ClientOutcomingMessageHandler`: un'interfaccia per la gestione della comunicazione tra il Peer ed il controller del gioco cosicché il primo possa notificare il secondo al verificarsi di determinati eventi;
  - `ClientOutcomingMessageHandlerImpl`: una classe che implementa l'interfaccia di cui sopra;
- `scr/main/scala/test/prolog/` un package in cui viene testata la logica di comportamento dei personaggi racchiusa nel file `src/main/resources/prolog/logic`;

- `GhostTest`: una classe in cui viene testato il comportamento dei fantasmi;
- `PacmanTest`: una classe in cui viene testato il comportamento di Pacman.

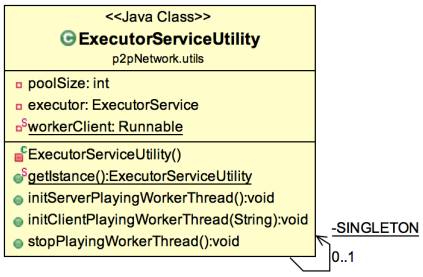

Figure 14: Pattern Singleton per la classe `ExecutorServiceUtility`

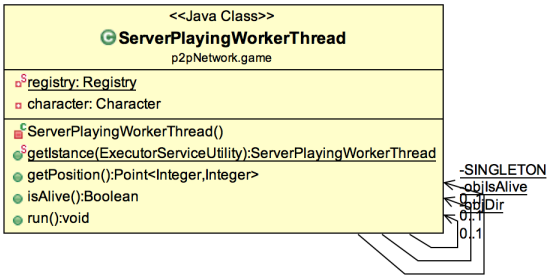

Figure 15: Pattern Singleton per la classe `ServerPlayingWorkerThread`

5.4 Margherita Pecorelli

- `src/main/resources/prolog/logic`: vedi descrizione precedente;
- `src/main/scala/client/controller`:
  - `ControllerCharacter`: (Figura 16) trait che si occupa della gestione dei personaggi, in particolare, la classe che lo implementa gestisce il movimento, la morte e le immagini di ognuno di loro;

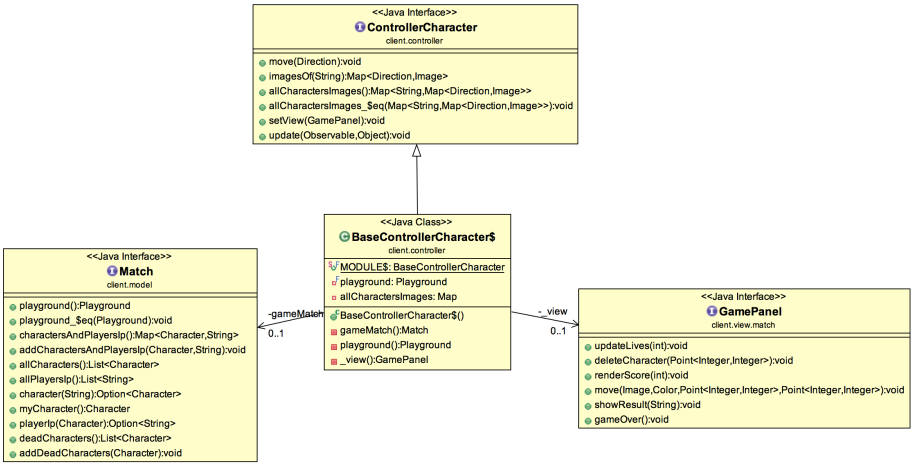

Figure 16: MVC: ControllerCharacter

– `ControllerMatch`: (Figura 17) trait che si occupa della gestione della partita, in particolare, la classe che lo implementa gestisce la richiesta e la scelta sia del range di giocatori, sia del personaggio, sia del campo da gioco; inoltre gestisce i messaggi di invito e di risposta a nuove partite, l'inizializzazione della partita e il salvataggio della fine di quest'ultima;

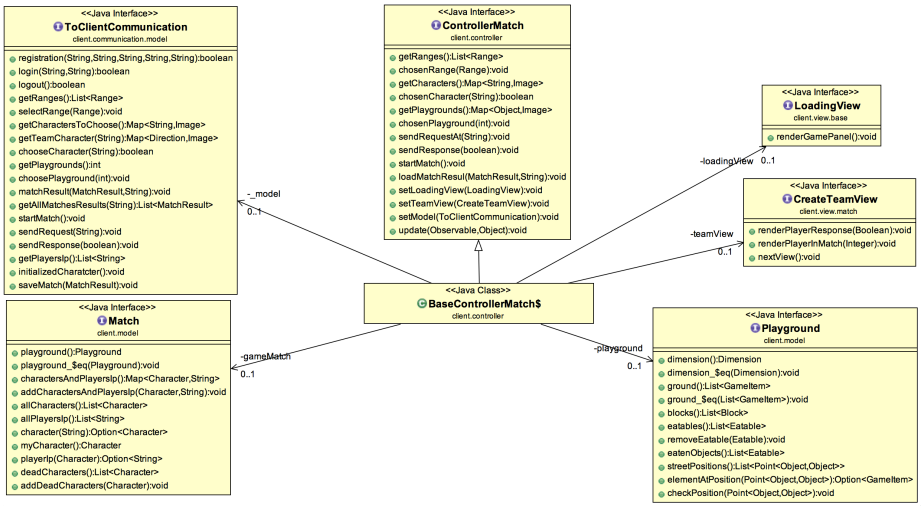

Figure 17: MVC: ControllerMatch

– `ControllerUser`: (Figura 18) trait che si occupa della gestione dello user corrente, in particolare, la classe che lo implementa gestisce la

registrazione, il login, il logout e la creazione di un player univoco;

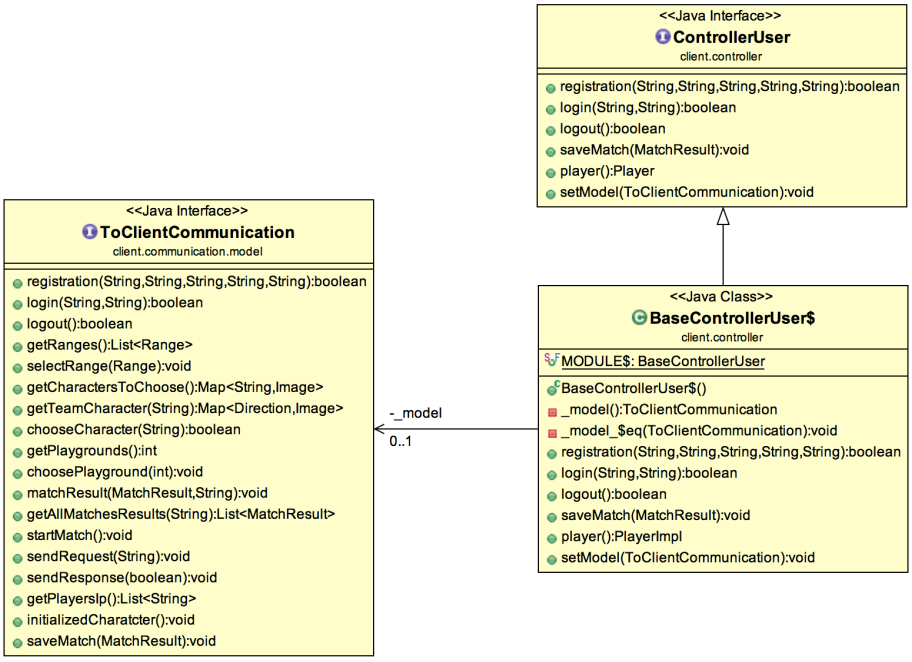

Figure 18: MVC: ControllerUser

- `src/main/scala/client/model`:
  - `Player`: (Figura 19) trait che gestisce il giocatore dello user corrente. La casse che la implementa, `PlayerImpl`, ha l'ip, la porta e lo username del giocatore, inoltre gestisce i risultati delle partite precedenti;

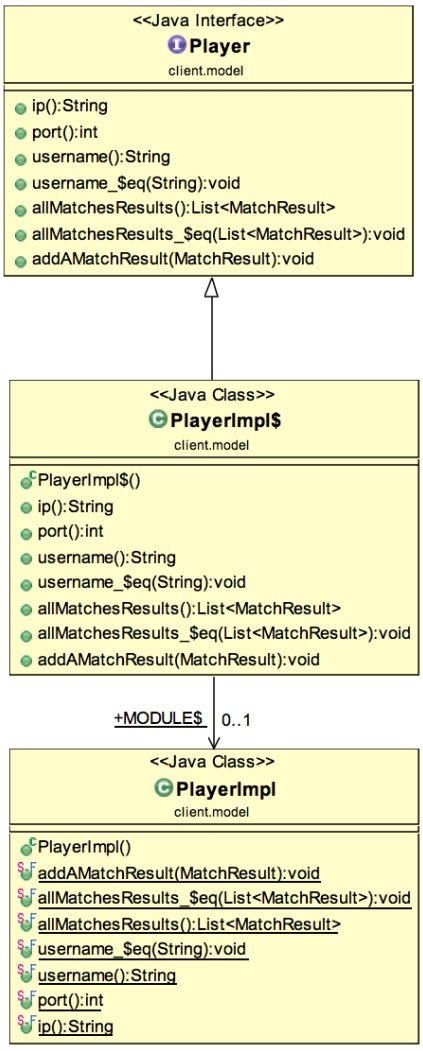

Figure 19: UML rappresentante un giocatore nel model del gioco

- `Match`: (Figura 20) trait che rappresenta la partita corrente. La classe che lo implementa, `MatchImpl`, gestisce il campo da gioco e tutti i personaggi con gli ip dei rispettivi giocatori (compreso quello dello user corrente);
- `Playground`: (Figura 20) trait che gestisce il campo da gioco della partita. La casse che la implementa, `PlaygroundImpl`, gestisce le dimensioni, gli oggetti presenti (tra cui i blocchi e gli oggetti mangiabili da Pacman) e le strade.
- `/character` (Figura 20):

    * `Character`: trait che rappresenta i personaggi. La classe che lo implementa, `CharacterImpl`, è una classe astratta che implementa i metodi in comune a tutti i personaggi (in particolare, si occupa di gestire il movimento di questi ultimi);
    * `Ghost`: trait che rappresenta i fantasmi del gioco. La classe che lo implementa, `BaseGhost`, controlla lo stato del fantasma (predatore o preda) in quel momento e, nelle opportune circostanze, consente a quest'ultimo di mangiare Pacman;
    * `Pacman`: trait che rappresenta Pacman. La classe che lo implementa, `BasePacman`, controlla lo stato del personaggio (predatore o preda) in quel momento e, nelle opportune circostanze, consente a quest'ultimo di mangiare un fantasma e/o uno degli oggetti presenti in campo;
- `/gameElement` (Figura 20):
    * `Eatable`: trait che rappresenta tutti gli oggetti presenti in campo mangiabili da Pacman. Le classi e le interfacce che lo implementano, `Fruit`, `Bell`, `Dot`, `GalaxianShip`, `Key` e `Pill`, gestiscono il proprio score e la propria posizione;
    * `Fruit`: trait che implementa `Eatable` e che rappresenta tutta la frutta presente in campo. Le classi che la implementano, `Apple`, `Cherry`, `Grapes`, `Orange` e `Strawberry`, gestiscono il proprio score e la propria posizione.
- `/utils` (Figura 21):
    * `EatObjectStrategy`: trait che rappresenta la strategia con la quale Pacman mangia gli oggetti nel campo. La classe che lo implementa, `BaseEatObjectStrategy`, gestisce il caso in cui Pacman mangi una pill: Pacman diventa il predatore e i fantasmi diventano le prede. Il tutto dura per circa 10 secondi;
    * `Timer`: trait che rappresenta un cronometro. La classe che lo implementa, `TimerImpl`, crea un nuovo thread che aspetti per il tempo determinato e che notifichi l'observer quando ha terminato il conto alla rovescia.

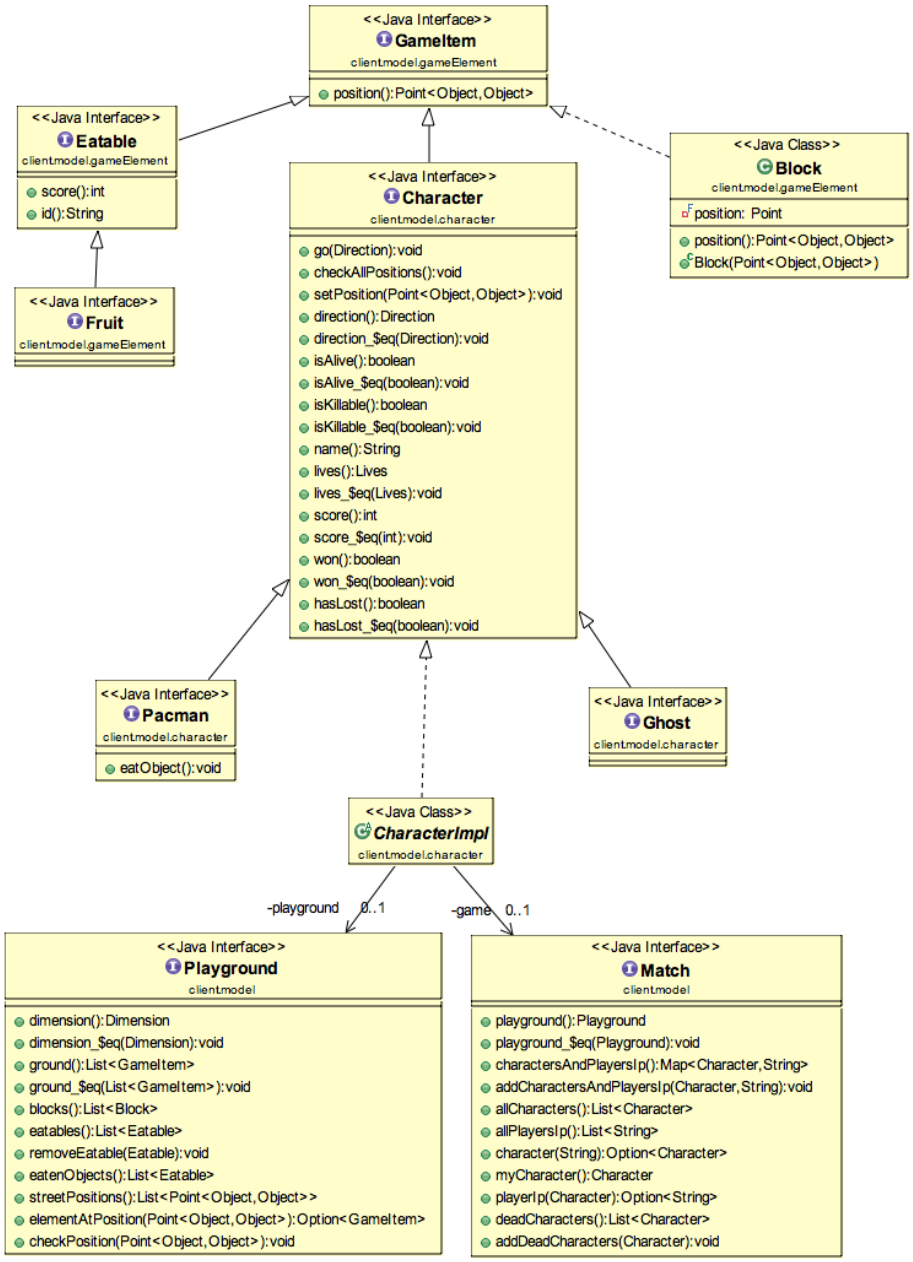

Figure 20: UML del model di gioco

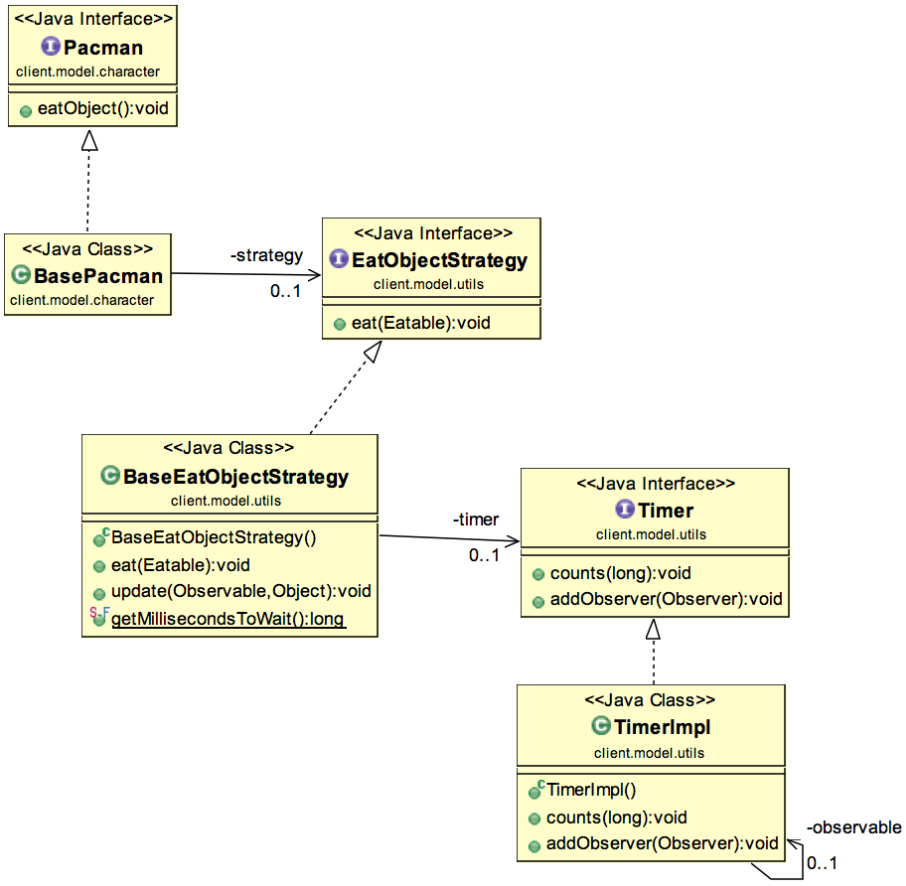

Figure 21: UML rappresentate il pattern Strategy utilizzato nel model

5.5 Chiara Varini

Come già descritto, la View dell'applicazione è implementata quasi interamente in Java, tuttavia si sono implementati, in Scala, due objects `Utils` e `Res` che si occupano del caricamento delle risorse e dei cast di tipo dalle strutture dati Scala passate dal controller e quelle Java utilizzate nella view.

Login

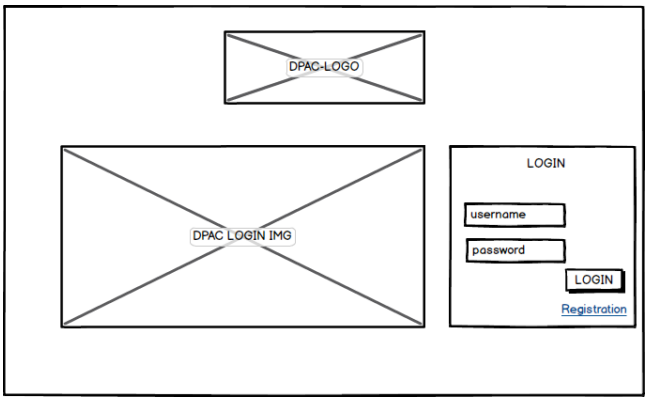

Figure 22: Scremata di Login

La schermata di Login è implementata nella classe java.client.view.base.LoginPanel.java. In questa viene creato un JPanel combinando un BorderLayout ed un GridBagLayout in modo da disporre ordinatamente tutti i componenti. L'accoppiata di questi due layout è utilizzata più volte nel codice della view. Per l'implementazione degli ActionListeners dei bottoni sono sempre utilizzate le *lambda* fornite da java 8.

Home

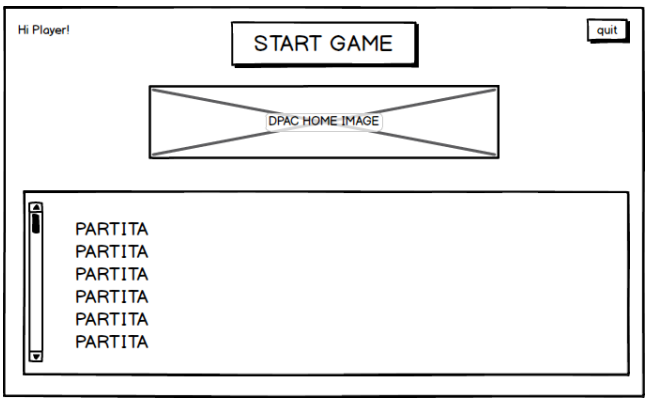

Figure 23: Scremata della Home

La scermata di Home è implementata nella classe java.client.view.base.Homepanel.java. In questa classe viene creata una JTable con tutte le partite svolte in precedenza dal giocatore. Questa JTabel viene poi inserita in uno JScrollPanel, così da

poter visualizzare tutte le partite anche nel caso in cui un giocatore ne svolga molte. Quando si preme il bottone `START GAME` viene visualizzato un JDialog implementato nella classe java.client.view.match.CreateTeamDialog.java, riportato nella Figura 24.

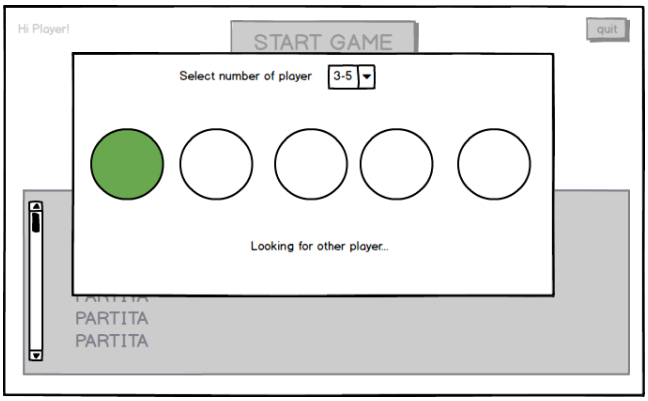

Figure 24: Schermata per la creazione di una nuova partita.

Ogni volta che un giocatore si aggiunge alla partita un'icona diventa verde, questo meccanismo è gestito dall'inner-class `PlayersPanel`. Quando il team è completato si passa direttamente alla schermata successiva grazie al metodo *nextView()*. In questa classe è gestita anche una classe interna `EnabledJComboBoxRenderer` per disabilitare un item del JComboBox, in quanto, per ora, non è possibile effettuare partite con un numero di giocatori superiore al 5 poiché non sono state inserite abbastanza immagini, dunque è stato disabilitato l'item "6-9".

Select Characters

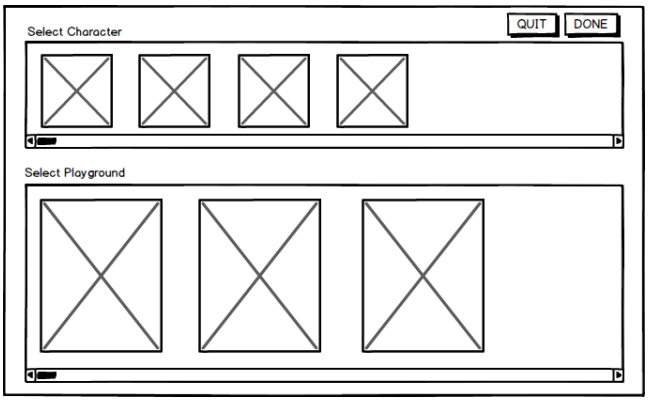

Figure 25: Schermata per la scelta del personaggio e del playground.

Una volta completata la squadra si passa alla schermata di selezione del personaggio e del playground, tutti inviati dal server. Questo pannello è implementato nella classe client.view.base.SelectCharacterPanel.java. Quando un giocatore sceglie un personaggio viene disabilitato per tutti gli altri utenti. Solo quando un giocatore ha scelto sia il personaggio che il playground di gioco si attiva il bottone *DONE* che permette il passaggio alla fase successiva del gioco.

Game

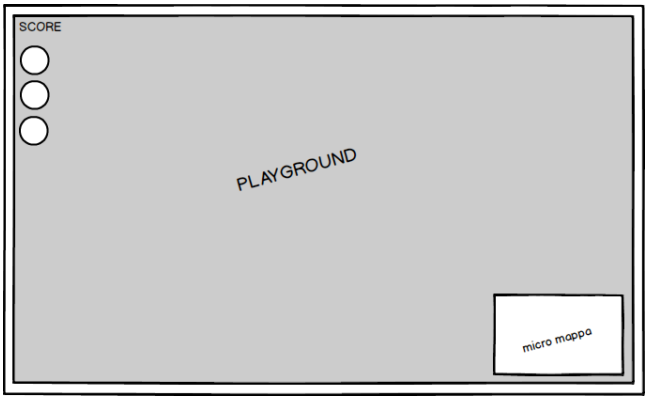

Figure 26: Schermata di gioco.

la classe che implementa la scremata di gioco è client.view.match.GamePanel.java. Questa classe estende da JLayeredPane e si compone di 4 parti:

- il pannello dove viene visualizzato lo svolgimento della partita (Playground);
- il pannello dove viene visualizzata l'intera mappa e gli spostamenti dei giocatori (MicroMap);
- il pannello dove vengono visualizzati lo score e, nel caso in cui si fosse scelto Pacman, il numero di vite rimaste;
- il pannello di "fine gioco" che si vinca o che si perda viene visualizzata una schermata con il risultato e il tasto *QUIT* per tornare alla home.

Questi elementi vengono aggiunti al gamePanel con diversi *index*, ed è questo che permette di sovrapporli. Nel metodo *move()* di questa classe è visibile l'applicazione degli stream di java 8, riportata di seguito

```
@Override
public void move(final Image ci, final Color cc, final Point<
    Integer,Integer> oldp, final Point<Integer,Integer> newp) {
    ...
```

```
            playground.renderEatableList(PlaygroundImpl
                    .eatables()
                    .stream()
                    .filter(e->e.position().x()==oldp.x() && e.position
                        ().y()==oldp.y())
                    .collect(Collectors.toList()));
                    ...
}
```

MicroMapPanel La mappa completa del labirinto posizionata in basso a destra è implementata nella classe `MicroMapPanel.java`, in cui viene generato un pannello formato da tanti piccoli JPanel disposti su una griglia. Ogni volta che un personaggio si muove viene richiamato il metodo

```
public void moveCharacter(Color color, Point<Integer,Integer>
    newPosition, Point<Integer,Integer> oldPosition)
```

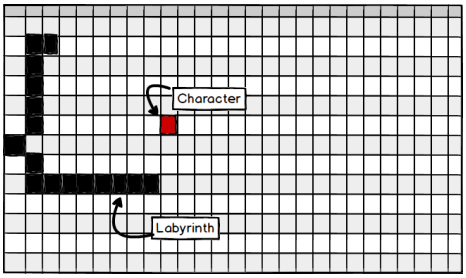

Figure 27: Mappa dell'intero labirinto.

Playground Nel package `client.view.playground` sono presenti tutte le classi utilizzate per creare il playground del gioco. In `BasePlaygroundPanel`, la classe che implementa l'interfaccia `BasePlaygroundView`, si trovano tutti i metodi di "base" per disegnare il campo da gioco come:

- disegnare un singolo blocco del labirinto;
- disegnare un frutto;
- cambiare la posizione ad un personaggio, il questo metodo si provvede anche a ridisegnare la porzione di mappa circostante al proprio personaggio.

I metodi di base forniti da `BasePlaygroundPanel` sono poi wrappati nella classe `PlaygroundPanel` che fornisce dei metodi di più alto livello come:

- aggiungere tutti i blocchi del labirinto in una volta sola;
- aggiungere tutti gli oggetti (eatables) in una volta sola.

In questa classe sono inseriti anche alcuni metodi privati utilizzati per la scelta dell'immagine da caricare in base alla posizione del blocco nel labirinto. Nel metodo *boolean lookAt(List<Block> blockList, Predicate<Point> predicate)* è stato utilizzato il pattern strategy per verificare se un blocco è vicino ad altri. Quindi la strategia consiste nel passare la posizione da verificare (sinistra, destra, sopra, sotto). Il metodo è riportato di seguito.

```
private boolean lookAt(List<Block> blockList, Predicate<Point>
    predicate){
        return blockList
                .stream()
                .map(b->b.position())
                .filter(predicate)
                .collect(Collectors.toList())
                .size() >= 1;
}
```

Tutte le dimensioni e proporzioni delle immagini sono ricavate da un oggetti di tipo `PlaygroundSettings` passato in ingresso al costruttore delle precedente classe. Per settare tutte le diverse proprietà è stato utilizzato il pattern `Builder` di cui se ne riporta lo schema nella Figura 28.

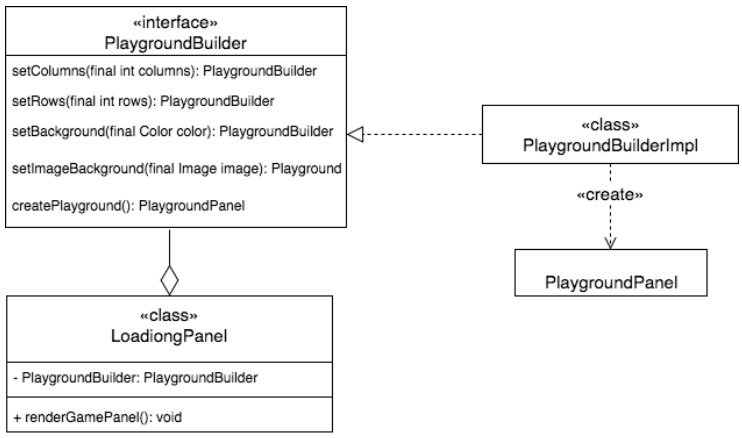

Figure 28: Scremata di gioco.

Utils

Un'ultima nota riguarda la classe `scala.client.view.Utils.scala`, infatti in questa classe i metodi di conversione dalle strutture scala a quelle java si sarebbero potuti implementare come *impliciti*, ma per questioni di tempo non si è provveduto ad implementare questa modifica.

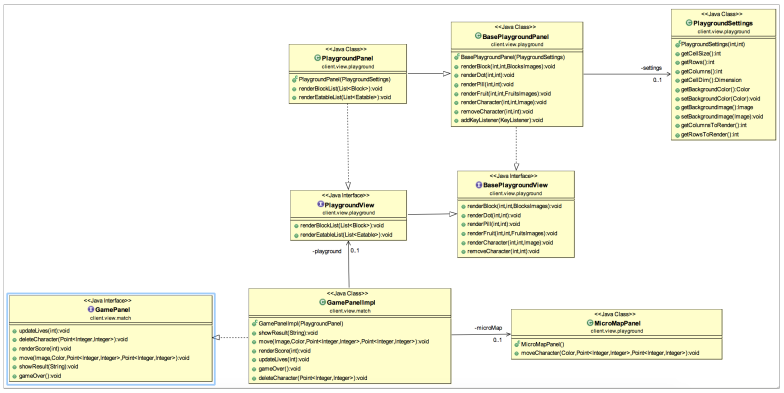

Figure 29

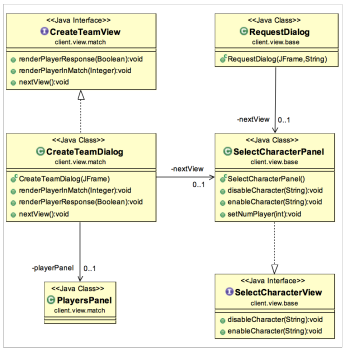

Figure 30

6 Retrospettiva

In fase di progettazione, abbiamo iniziato progettando l'architettura del gioco, in particolare ci siamo focalizzati sulla comunicazione client-server, sulla comunicazione peer to peer e sul model, creando i diagrammi sopra proposti in figura 2 e in figura 4.
Successivamente, in fase di programmazione, abbiamo iniziato, in contemporanea, a implementare la logica del gioco in prolog e la versione base del model. In seguito abbiamo iniziato a programmare una versione iniziale sia delle view che del controller. Questa operazione ci ha occupato i primi due Sprint.
Arrivati a una prima versione funzionante, abbiamo iniziato a lavorare parallelamente sulla comunicazione peer to peer e client-server, mentre un altra parte del team continuava a lavorare sul pattern architetturale MVC.
L'ultimo periodo è stato utilizzato per integrare e per verificare il corretto funzionamento delle varie parti del sistema e dell'applicazione stessa. Inoltre, durante tutto lo sviluppo, ognuno di noi, quando possibile, ha testato la propria parte di codice.

6.1 Sviluppi Futuri

L'applicazione è stata creata per essere pronta a nuove funzionalità e modifiche. Gli sviluppi futuri pensati sono i seguenti:

- possibilità di selezionare partite con diverso numero di giocatoti. Per ora sono possibili partite con un minimo di 3 giocatori ed un massimo di 5. Il codice è stato implementato, in modo tale da poter aggiungere facilmente altri range. Si potrà notare, durante l'esecuzione dell'applicazione, che è già presente il range 6-9 disabilitato;

- aggiunta della funzione "invita un amico", in modo da poter invitare un amico all'interno della propria partita tramite l'username univoco. Sia nella parte client sia nella parte server viene gestito a livello di codice, ma per il poco tempo, non essendo stato testato, si è deciso di rimuovere temporaneamente la funzionalità;

- versione multipacman del gioco, in cui più di un Pacman può giocare nella stessa partita, già parzialmente gestita a livello di model;

- aggiunta di livelli o di nuovi campi da gioco da sbloccare in base al punteggio totalizzato da un giocatore.

Infine si potrebbe rendere l'applicazione disponibile per smartphone, sicuramente più attrattiva a livello utente.

6.2 Commenti Finali

La valutazione finale del lavoro in team è stata nel complesso molto positiva, in quanto, nonostante i molteplici impegni lavorativi di ognuno di noi che ci

costringevano a non poterci vedere se non in rari momenti, siamo comunque riusciti ad organizzare il lavoro e a portarlo a termine. Per la motivazione sopra elencata, non siamo riusciti a rispettare in maniera categorica la deadline prefissa. Nonostante ciò, il team era abbastanza eterogeneo da poterci permettere di di suddividere i task, a seconda delle potenzialità di ognuno. Inoltre il gruppo è risultato essere coeso e disponibile alla collaborazione.
Infine abbiamo riscontrato alcune criticità. In primo luogo, abbiamo trascurato parzialmente la progettazione dell'architettura MVC, ossia ci siamo focalizzati sulla parte del model, mettendo da parte controller e view. Questo ha portato, nel momento di iterazione e integrazione de queste ultime due parti, a delle grosse difficoltà.
A posteriori, ci siamo resi conto che implementazione della parte distribuita avremmo dovuto farla in contemporanea all'implementazione dell'MVC del gioco. Inoltre, seppure cercando di utilizzare il processo SCRUM, abbiamo riscontrato difficoltà nel rilasciare release funzionanti con cadenza settimanale.

7 Guida all'uso

Alla consegna del progetto, abbiamo fornito due jar distinti: quello del server e quello dell'applicazione stessa. I due jar dovranno essere eseguiti su macchine differenti, ma collegate ad una stessa rete. In particolare, il jar dell'applicazione dovrà essere lanciato, fornendo come parametro in input l'IP della macchina su cui verrà eseguito il server. In particolare si dovranno usare il seguente comando per l'esecuzione dei jar:
parte server:
clonare il progetto DPAC-Server ed eseguire dalla cartella del progetto il jar
`java -jar "/out/artifacts/DPAC_Server_main_jar/DPAC_Server_main.jar"`
parte client:
`java -jar ".../DPAC_DistributedPacman_main_jar/DPAC_DistributedPacman_main_jar.jar" input`
Per una corretta esecuzione del server, sarà necessario scaricare dalla sezione download il file .sql del database e inserirlo in locale, attraverso il supporto di xampp o qualsiasi altro tool simile.
Il dover eseguire il server in questo modo è dovuto dai problemi che AKKA crea nel fare il jar, come può essere visibile anche della documentazione ufficiale di AKKA.

Avviata l'applicazione, sarà necessario registrarsi oppure effettuare il login, se la registrazione risulterà già completata.
Per partecipare ad una partita, si dovrà premere il tasto "START GAME" ed inserire il range dei giocatori desiderato, anche se momentaneamente obbligato. Sarà necessario aspettare il numero minimo di giocatori in attesa. Per l'avvio della partita, si dovrà aspettare 10 secondi oppure il raggiungimento massimo del numero di giocatori.
Dopodichè si dovrà scegliere il personaggio ed il campo da gioco desiderato. La scelta del personaggio può avvenire solo se esso non è già stato scelto da un altro giocatore, come sarà possibile vedere runtime dall'interfaccia grafica. Tra i campi da gioco disponibili, verrà scelto quello che ha ricevuto il numero maggiore di voti oppure, in caso di parità, quello selezionato da pacman.
Il movimento dei personaggi avviene tramite i tasti freccia.
Alla fine della partita però non è ancora possibile creare e giocare ad un'altra partita consecutivamente per problemi dovuti all'RMI.
Inoltre, a volte, per problemi di rete ci potrebbe essere qualche errore ed è opportuno per giocare riavviare il server. Questi ultimi due problemi, non essendo pienamente parte del programma e avendo poco tempo a disposizione abbiamo deciso di focalizzarci su altro.

Infine, si tenga presente che, come nel gioco originale, lo scopo della partita per pacman è quello di mangiare tutte le palline presenti nel campo, rimanendo con almeno una vita; quello dei fantasmi invece consiste nel mangiare pacman tante volte quante sono le sue vite.
Nel caso in cui pacman mangi una delle palline più grandi, i ruoli si invertireb-

bero: per 10 secondi pacman diventerebbe predatore e i fantasmi prede. In questo lasso di tempo, lo scopo di pacman rimane invariato, mentre quello dei fantasmi diventa non farsi magiare.