

Michele Fiorito – 826607

Andrea Guglielmini – 826160

Progetto Finale

Lorenzo il Magnifico

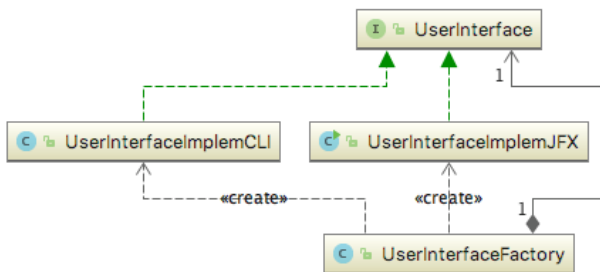
Documentazione dell'applicazione e Manuale d'Uso per l'Utente

Client

Architettura

Il gioco presenta due diversi *Clients*; l'utente, all'avvio del programma, può decidere se utilizzare l'**interfaccia grafica** o l'alternativa modalità **da terminale**. La scelta non influisce sul normale svolgimento del gioco.

Tutto questo è possibile mediante l'utilizzo di una struttura ad interfacce che, potenzialmente, permette di aggiungere al client una terza modalità di gioco senza dover modificare alcuna riga di codice.

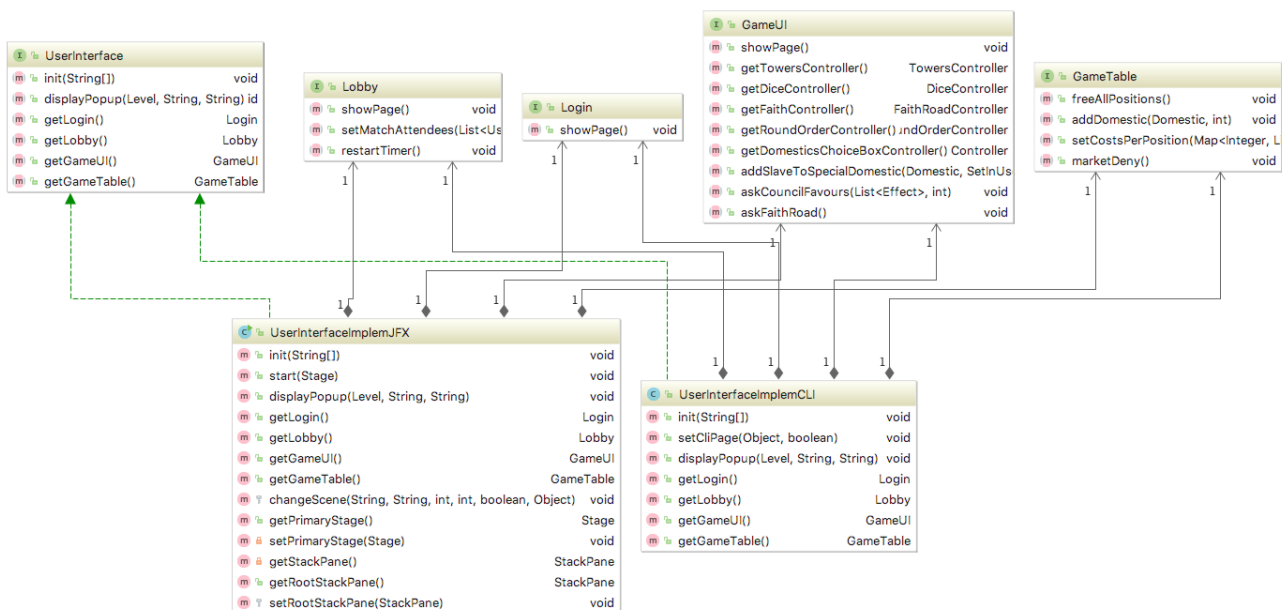


L'utilizzo del **factory design pattern** permette al *Main* di disinteressarsi completamente della natura della grafica.

`UserInterfaceFactory` oltre a garantire la creazione di un oggetto grafico che implementa `UserInterface` assicura anche che sia *unico*; una volta istanziato infatti la

factory mantiene un riferimento all'oggetto permettendo di accedervi in qualsiasi punto del Client chiamando il metodo `getInstance()`.

Le due classi `UserInterfaceImplemCLI` e `UserInterfaceImplemJFX` contengono i riferimenti a tutti i componenti del gioco implementati con la rispettiva tecnologia:



- **Login:** Questa interfaccia ha un solo metodo: *showPage()*; che permette di cambiare pagina e mostrare quella di interesse, il resto della logica viene implementato dalla classe che estende l'interfaccia stessa.
- **Lobby:** Simile alla *Login interface*, aggiunge però la possibilità di aggiornare la lista dei giocatori in attesa dell' inizio della partita (*sendMatchAttendees(userList)*) e la possibilità di resettare il countdown verso l'avvio automatico della stessa (*resetTimer()*)
- **GameTable:** Questa interfaccia racchiude le funzioni *base* del campo di gioco, cioè la possibilità di posizionare un familiare sul tavolo (*addDomestic(domestic, positionNumber)*), di impostare i costi per ogni singola posizione (*setCostPerPosition(CostsList)*) o di ripulire il tavolo per l'inizio di una nuova mached (*freeAllPositions()*).
- **GameUI:** Contiene i riferimenti ai controllori di tutti gli altri elementi della *UserInterface*: dadi, torri, ordine di gioco e percorso fede. Espone inoltre tre metodi fondamentali che permettono al server di interagire con l'utente:
 - **addSlaveToSpecialDomestic():** Utilizzato per aggiungere schiavi quando viene utilizzato un *familiare speciale* inviato dal server.
 - **askCouncilFavours(howManyFavours):** Chiede all' utente di scegliere *#howManyFavours* differenti favori del consiglio.
 - **askFaithRoad():** Utilizzato per sapere se l'utente vuole supportare o meno la Chiesa.

La differenza tra client grafico e terminale consiste nella diversa implementazione delle sopracitate interfacce, in particolare la *GameUI interface* e *GameTable interface*.

Datawarehouse

Vi è inoltre una terza classe, non ancora trattata, che viene utilizzata come database locale:

Datawarehouse	
getInstance()	Datawarehouse
registerPlayerStateObserver(PlayerStateObserver)	void
registerTurnObserver(TurnObserver)	void
getMyUser()	User
setMyUser(User)	void
getMyUsername()	String
getFamilyColor(String)	FamilyColor
getPlayerState(String)	PlayerState
setPlayerState(String, PlayerState)	void
getGameUser(String)	GameUser
setGameUser(String, GameUser)	void
getWhoseTurn()	String
setWhoseTurn(String)	void
getMatchAttendees()	List<User>
setMatchAttendees(List<User>)	void

Datawarehouse è un **singleton** che contiene le principali informazioni inviate dal server e le rende sempre reperibili agli oggetti che li richiedono.

La stessa classe permette di registrare **observers** per i *PlayerStates* ed il cambio di turno.

Non appena il server notifica il client con nuovi dati relativi a queste informazioni, Datawarehouse avvisa gli oggetti che si sono registrati.

Gli osservatori, necessariamente, devono implementare la classe *PlayerStateObserver* oppure *TurnObserver*.

Client Grafico

Per la realizzazione del Client Grafico è stato utilizzato **JavaFX** insieme alla libreria **JFoenix** per arricchire la collezione di componenti grafici utilizzabili e **JFX3DModelImporter** per caricare oggetti 3D da file.

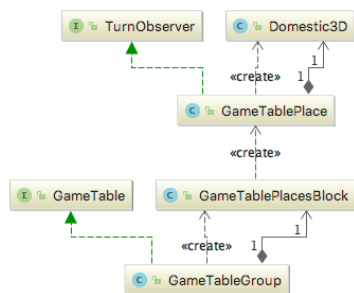
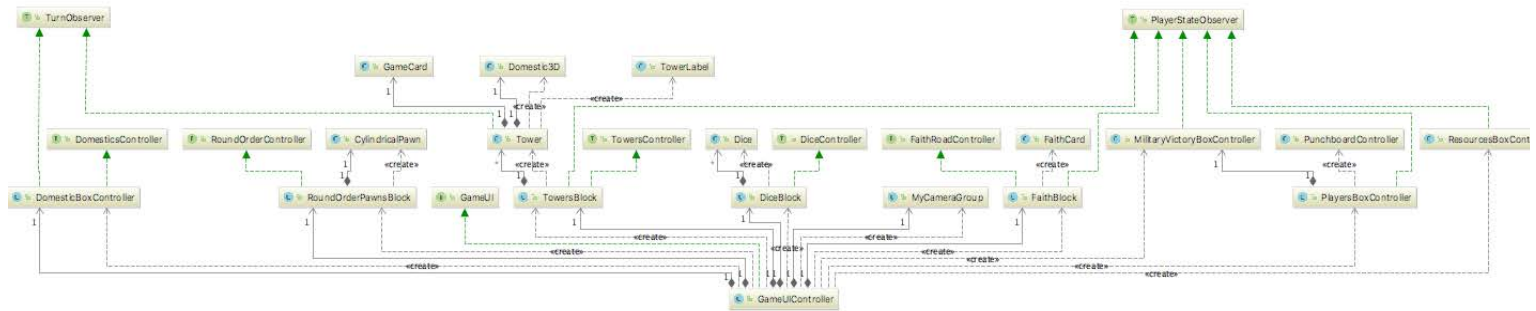
Le interfacce **Login** e **Lobby** precedentemente discusse sono state implementate dal controller FXML delle relative pagine.

La grafica del gioco è invece composta da diversi oggetti che lavorano sinergicamente:



Progetto Finale

...

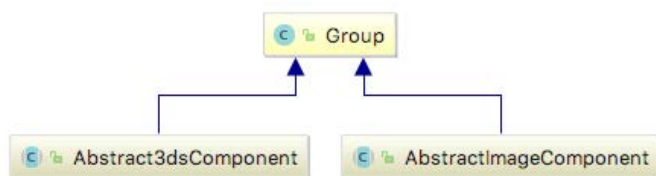


Le varie viste sono state realizzate con l'ausilio di FXML, l'interfaccia di gioco consiste infatti in uno **StackPane** padre di tutti gli elementi grafici, due **AnchorPane** nella parte alta e bassa della schermata che permettono di mostrare alcune informazioni sull'avanzamento della partita ed una **SubScene** centrale.

La *visuale* della sottoscena viene gestita mediante una *telecamera* (*MyCameraGroup*), a questo oggetto vengono applicate trasformazioni ed animazioni permettendo di spostare la visuale di in modo fluido.

Tutti i componenti grafici estendono la classe *Group*, è garantita quindi una elevata flessibilità in quanto possiamo trarre vantaggio di tutte le funzionalità ereditate tra cui la possibilità di traslare, animare e ridimensionare qualsiasi componente grafico.

Per massimizzare la *code reusability* sono state utilizzate due classi astratte **AbstractImageComponent** e **Abstract3dsComponent**.

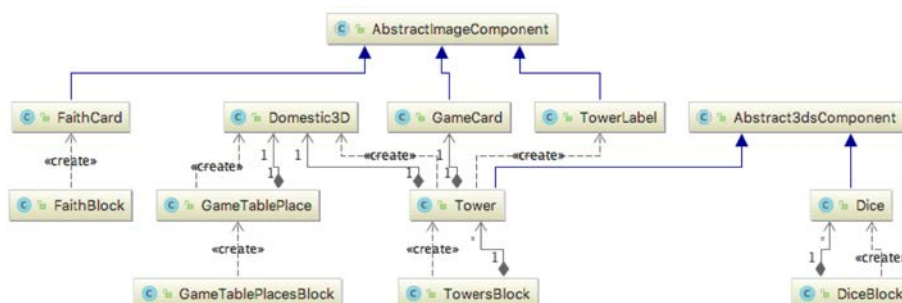


```

classDiagram
    class AbstractImageComponent {
        loadImage(String, double, double, double, double, double, double, double) void
        getImageView() ImageView
        getImage() Image
        getTranslate() Translate
    }
    class Abstract3dsComponent {
        load3ds(String, double, double, double, double, double, double, double, double, double) void
        getTranslate() Translate
    }
    class Group {
    }
    class FaithCard {
    }
    class Domestic3D {
    }
    class GameCard {
    }
    class TowerLabel {
    }
    class Dice {
    }
    AbstractImageComponent <|-- FaithCard
    AbstractImageComponent <|-- Domestic3D
    AbstractImageComponent <|-- GameCard
    AbstractImageComponent <|-- TowerLabel
    Abstract3dsComponent <|-- Dice
    Group <|-- AbstractImageComponent
    Group <|-- Abstract3dsComponent
    
```

```

classDiagram
    class AbstractImageComponent {
        loadImage(String, double, double, double, double, double, double, double) void
        getImageView() ImageView
        getImage() Image
        getTranslate() Translate
    }
    class Abstract3dsComponent {
        load3ds(String, double, double, double, double, double, double, double, double, double) void
        getTranslate() Translate
    }
    class Group {
    }
    class FaithCard {
    }
    class Domestic3D {
    }
    class GameCard {
    }
    class TowerLabel {
    }
    class Dice {
    }
    AbstractImageComponent <|-- FaithCard
    AbstractImageComponent <|-- Domestic3D
    AbstractImageComponent <|-- GameCard
    AbstractImageComponent <|-- TowerLabel
    Abstract3dsComponent <|-- Dice
    Group <|-- AbstractImageComponent
    Group <|-- Abstract3dsComponent
    
```



La prima aiuta a creare, posizionare e ridimensionare un'immagine qualsiasi. La seconda contiene la logica per caricare velocemente un oggetto 3D da file .3ds

Per mostrare gli elementi in sovraimpressione è stato utilizzato il componente *Dialog* di **JFoenix**.



Al lancio della **CLI** viene avviato un *Thread* che rimane in ascolto sullo *stdin*; non appena viene rilevata una stringa, questa è divisa in un array di singole parole.

Tutti i metodi tranne showPage() sono annotati e invocabili da CLI

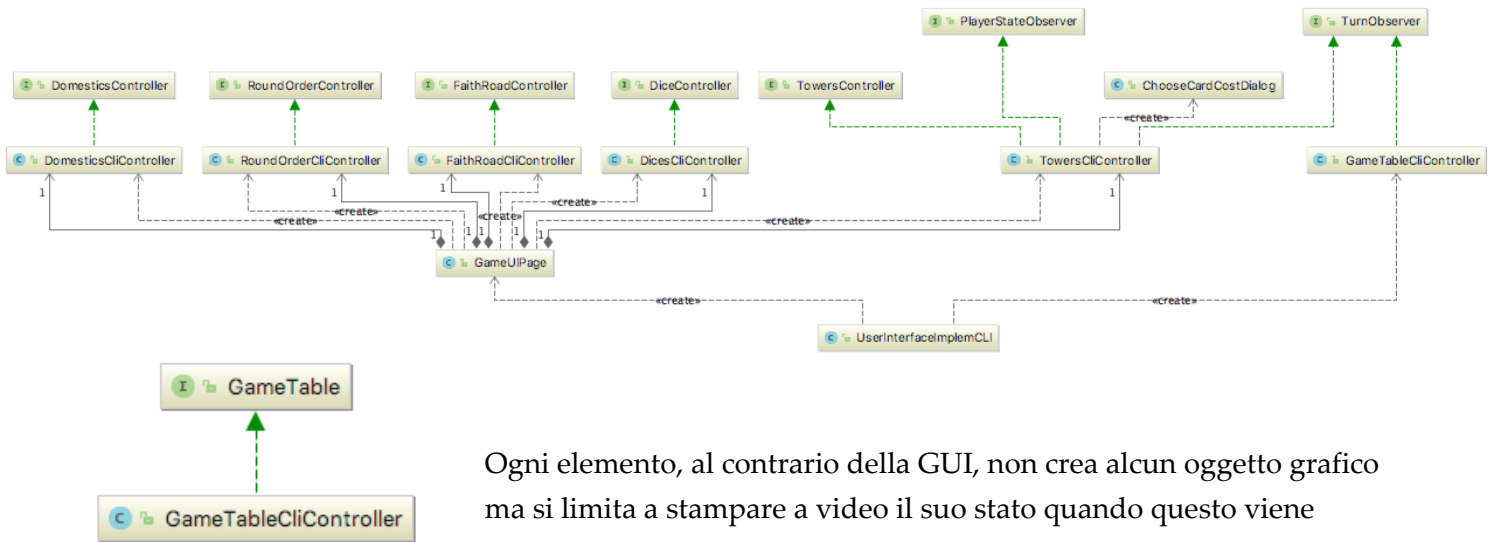
Viene poi invocato il metodo `tryToCallMethod(arrayParole[0])` per vedere se quel metodo esiste nel controller della schermata che l'utente sta visualizzando, in caso di esito positivo il metodo viene invocato utilizzando come argomenti le restanti parole dell'array.

Ogni volta che viene ‘*cambiata pagina*’, il controllore della schermata successiva deve necessariamente informare *CliController* mediante *setCliPage(object, printHelpMessage)*.

Tutti i metodi *invocabili* da terminale sono annotati con `@Command(description = "Descrizione della funzionalità")`, in questo modo è impossibile per l'utente chiamare metodi interni (ad esempio, `toString()`)

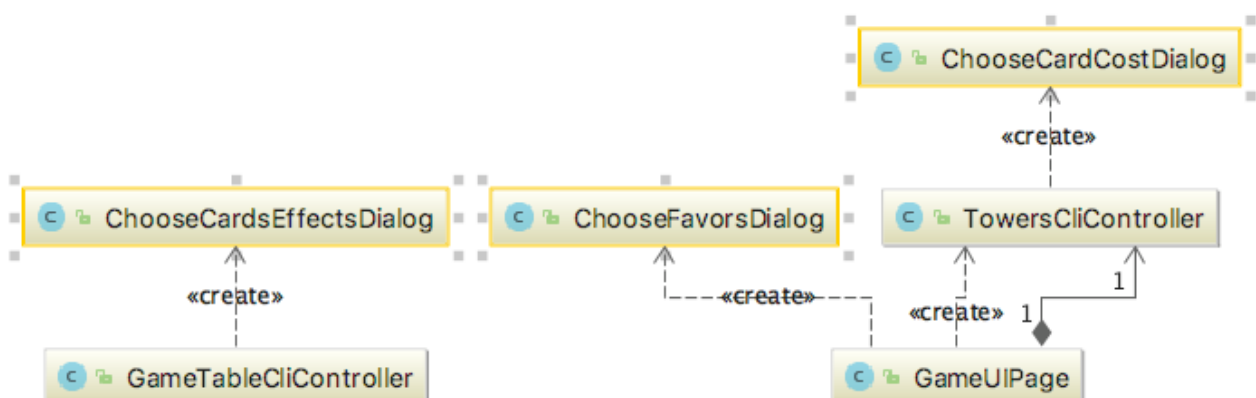
L'utilizzo delle annotazioni (fornite dalla libreria *Cliche*), permettono di mostrare all'utente un *help message* dinamico che lo informa sulle possibili funzioni e sulla descrizione dei singoli parametri.

Analogamente alla GUI, anche nella CLI sono state implementate tutte le interfacce di gioco:



Ogni elemento, al contrario della GUI, non crea alcun oggetto grafico ma si limita a stampare a video il suo stato quando questo viene modificato.

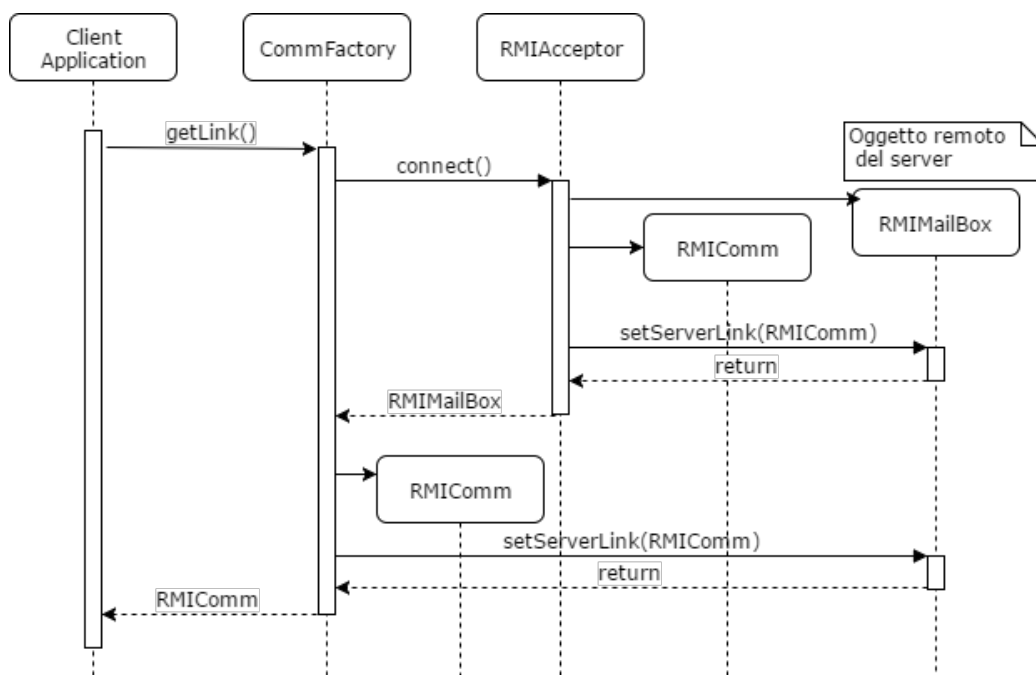
Quando è necessario interagire con l'utente, ad esempio per chiedere che effetti vuole attivare per ogni carta, viene utilizzato un controller a parte in modo da restringere i potenziali comandi utilizzabili:



Networking e Comunicazione

La connessione tra client e server è standardizzata mediante l'interfaccia *CommLink* tramite la quale è possibile inviare azioni e impostare il metodo di callback per la ricezione. Tale interfaccia è implementata sia mediante connessioni *socket* sia mediante *RMI*: in entrambi i casi l'oggetto *BaseAction* da inviare viene serializzato in formato *json* prima dell'inoltro e, successivamente, deserializzato in fase di ricezione prima di essere passato alla funzione di callback. Tale metodo è eseguito, in entrambi i casi, in un thread apposito associato all'oggetto *CommLink* in uso; in questo modo è garantita l'esecuzione sequenziale delle azioni ricevute, evitando parallelismi indesiderati.

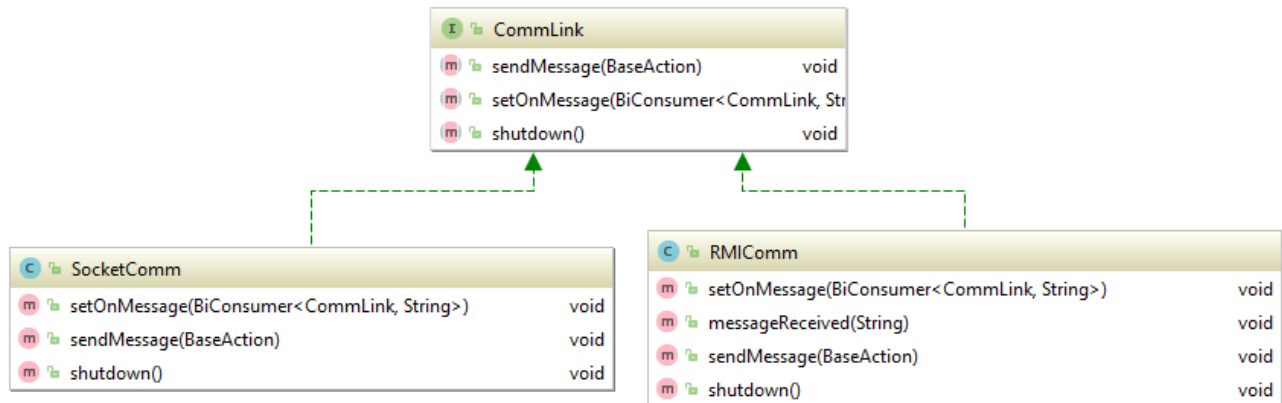
Per quanto riguarda l'implementazione via *socket* (*SocketComm*) viene utilizzato un ulteriore thread, oltre quello per l'esecuzione della callback, per la ricezione dei messaggi. Nel caso dell'implementazione *RMI* è invece presente un oggetto *MailBox* che fa da intermediario comune tra gli oggetti *RMIComm* client e server; tramite *RMIMailBox* i due oggetti *RMIComm* vengono registrati e mutuamente accoppiati in modo da garantire il corretto scambio dei messaggi. Tramite questo meccanismo, una volta chiamato il metodo *RMIComm::sendMessage* tramite *RMIMailBox*, viene chiamato in remoto il metodo di ricezione dell'oggetto *RMIComm* associato, il quale provvede ad accodare l'esecuzione della callback con il nuovo messaggio nel thread precedentemente descritto.



Esempio di connessione RMI

L'inizializzazione della connessione viene gestita tramite l'oggetto *CommFactory* del client, il quale contatta il *SocketAcceptor* relativo al tipo di connessione richiesta (*Socket* o *RMI*) e

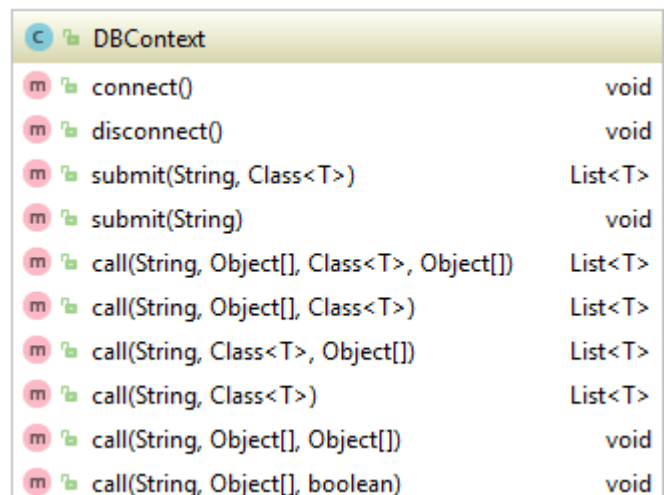
produce un oggetto *CommLink* generico pronto per l'utilizzo. Nel caso di connessione via socket viene contattato l'oggetto standard *ServerSocket* (inizializzato e utilizzato dal server all'interno di *SocketAcceptor*) e, sul server come sul client, viene inizializzato un nuovo *SocketComm*. Nel caso di RMI invece viene chiamato il metodo *RMIAcceptor::connect* in remoto sul server: in questo modo viene costruito l'oggetto *RMIMailBox* per la nuova connessione e viene associato all'oggetto *RMIComm* del server; infine è restituito al client, il quale crea a sua volta il proprio *RMIComm* e lo associa completando l'inizializzazione.



Networking.SQL

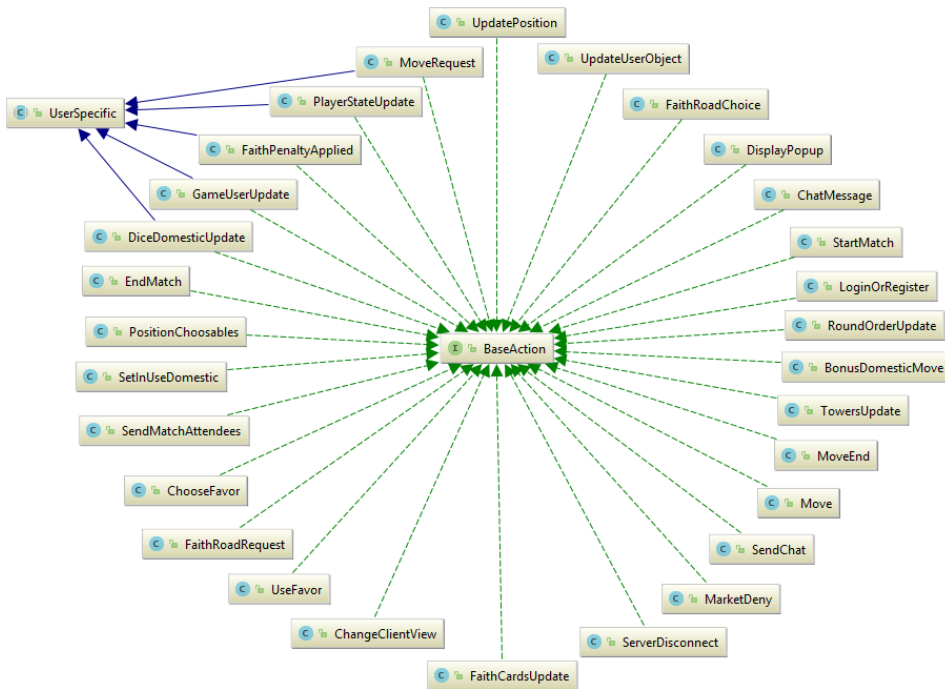
Nell'applicazione server è presente una connessione ad un database SQL tramite l'oggetto *DBContext*. Mediante questo oggetto è possibile fare *query* ed eseguire *stored procedures* su un database qualsiasi, convertendo i risultati direttamente in oggetti Java. La classe è implementata in modo generico, per garantire la massima flessibilità e presenta due metodi principali: *DBContext::submit* per l'interrogazione di tabelle e la ricezione dei risultati sotto forma di lista di oggetti; *DBContext::call* per la chiamata di *stored procedures*, con o senza parametri (sia in input che in output) con la possibilità di mappare i risultati su oggetti di un tipo specifico.

In questo caso l'oggetto *DBContext* viene utilizzato unicamente dalla classe *UserManager*, descritta altrove, per la gestione degli utenti tramite *stored procedures*.



BaseAction

Ogni oggetto utilizzato nella comunicazione implementa l'interfaccia *Action.BaseAction* e, più nello specifico, il metodo *BaseAction::doAction*: in questo modo ogni richiesta viene gestita indipendentemente dalle altre e indipendentemente dal metodo di comunicazione utilizzato (RMI o socket).

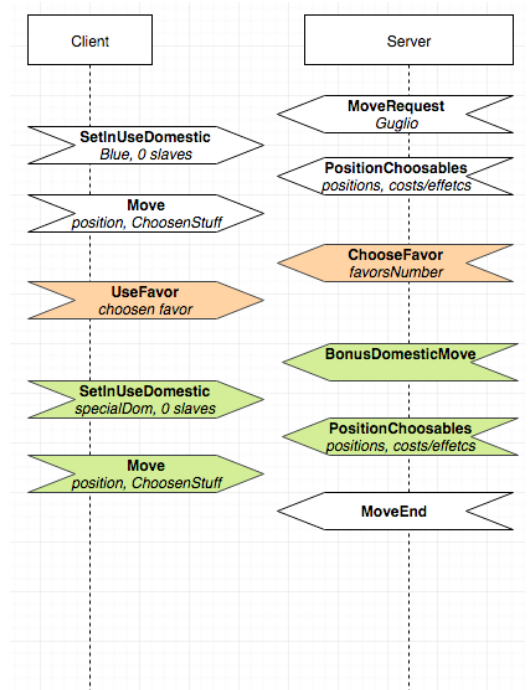
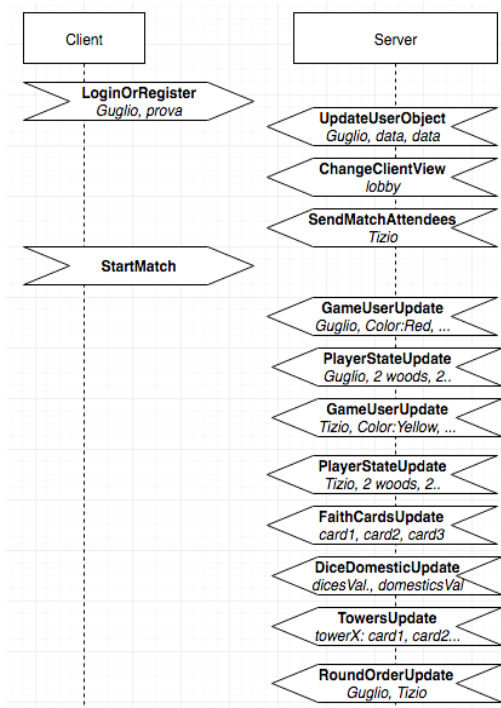


Gli oggetti *Actions* vengono serializzati e scambiati tra client e server per veicolare informazioni in contenitori specializzati. La tipologia di connessione o di user interface utilizzati non alterano in alcun modo la gestione o la creazione di questi messaggi. Di seguito un diagramma sintetico che permette di vedere questi oggetti in azione.

Game Setup

Fase di Gioco

I messaggi colorati sono esecuzioni speciali legate ad alcune carte.



Gestione degli utenti

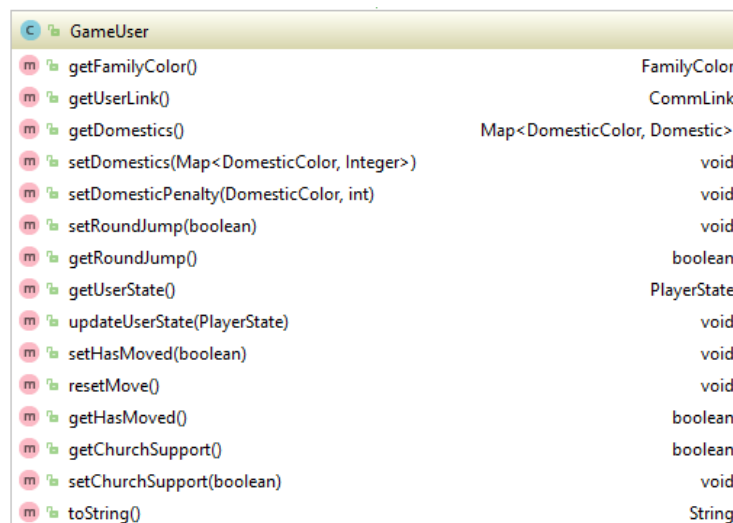
Per quanto riguarda l'autenticazione dell'utente questa avviene tramite username e password, codificata in MD5, i quali vengono verificati tramite un oggetto *UserAuthenticator* (nel nostro caso *Model.UserManager*), il quale implementa le funzionalità *CRUD* per gli oggetti *User*, connettendosi al database SQL tramite la classe *DBContext* citata precedentemente.

Una volta effettuato il login l'utente è univocamente identificato da un oggetto *User*, il quale, oltre a username e password, sarà aggiornato con un riferimento alla partita in corso e al giocatore di questa partita associato all'utente, ossia l'oggetto *GameUser*. Tramite questi riferimenti è garantito l'accesso a tutto ciò che riguarda l'utente durante l'esecuzione di una *BaseAction*.



GameUser

La classe *GameUser* rappresenta l'utente all'interno di una partita e contiene tutte le informazioni necessarie allo svolgimento della stessa, quali colore della famiglia, familiari aggiornati con il valore corrispondente, alcuni flag necessari per l'attivazione di effetti di gioco (*roundJump* e *churchSupport*), un riferimento all'oggetto *User* per risalire al *CommLink* associato e un contatore di mosse per gestire la fine del turno. Inoltre è presente un riferimento ad un oggetto *PlayerState*, nel quale è contenuto il vero e proprio stato attuale del giocatore.



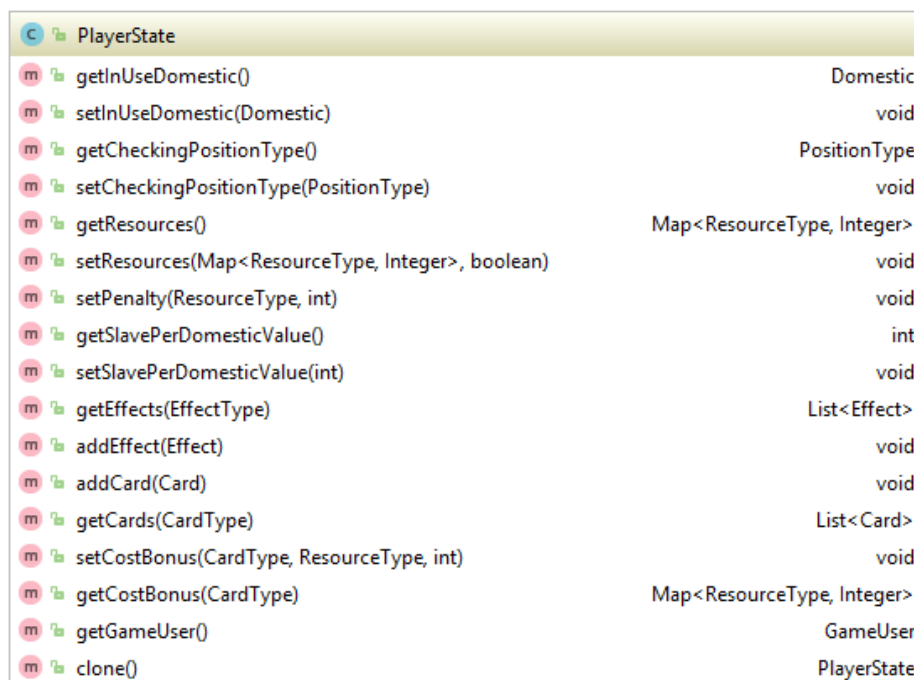
GameUser	
m	getFamilyColor() FamilyColor
m	getUserLink() CommLink
m	getDomestics() Map<DomesticColor, Domestic>
m	setDomestics(Map<DomesticColor, Integer>) void
m	setDomesticPenalty(DomesticColor, int) void
m	setRoundJump(boolean) void
m	getRoundJump() boolean
m	getUserState() PlayerState
m	updateUserState(PlayerState) void
m	setHasMoved(boolean) void
m	resetMove() void
m	getHasMoved() boolean
m	getChurchSupport() boolean
m	setChurchSupport(boolean) void
m	toString() String

I familiari sono aggiornati tramite un apposito setter che tiene conto di eventuali penalità sul valore (impostate direttamente sull'oggetto *GameUser* in questione); per l'informazione sul singolo familiare si utilizza la classe *Domestic*, nella quale è presente sia il valore che un flag ad indicare se il familiare è o meno in posizione: poiché durante l'esecuzione vengono create copie dello stesso familiare si è implementato un meccanismo che mantiene unico il riferimento a tale flag, in modo da garantirne la veridicità (ad esempio, quando il familiare nero della famiglia verde è in posizione, ogni sua copia restituirà *true* alla chiamata del metodo *Domestic::isInPosition*).

L'oggetto *PlayerState* contenuto nel *GameUser* è anch'esso utilizzabile tramite getter e setter (*getUserState* e *updateUserState*), ma viene passato sempre per copia in ogni sua parte, in modo da evitare modifiche indesiderate durante l'esecuzione. L'unico modo di aggiornare permanentemente il *PlayerState* è utilizzando il setter apposito.

PlayerState

La classe *PlayerState* contiene tutte le informazioni che riguardano lo stato attuale del giocatore: quantità di risorse, punti (vittoria, militari e fede), favori del consiglio, carte ottenute ed effetti, bonus sui costi e penalità sulle risorse ricevute. Inoltre è presente anche il familiare scelto per la mossa corrente e il tipo di posizione attualmente controllata, necessario per verificare le possibilità di posizionamento. Al contrario del *GameUser* il *PlayerState* non effettua copie tramite i getter, poiché è necessario che ogni modifica si ripercuota sull'oggetto, in modo da poter applicare più modifiche in sequenza senza perdere le precedenti (ad esempio, durante l'attivazione di vari effetti).



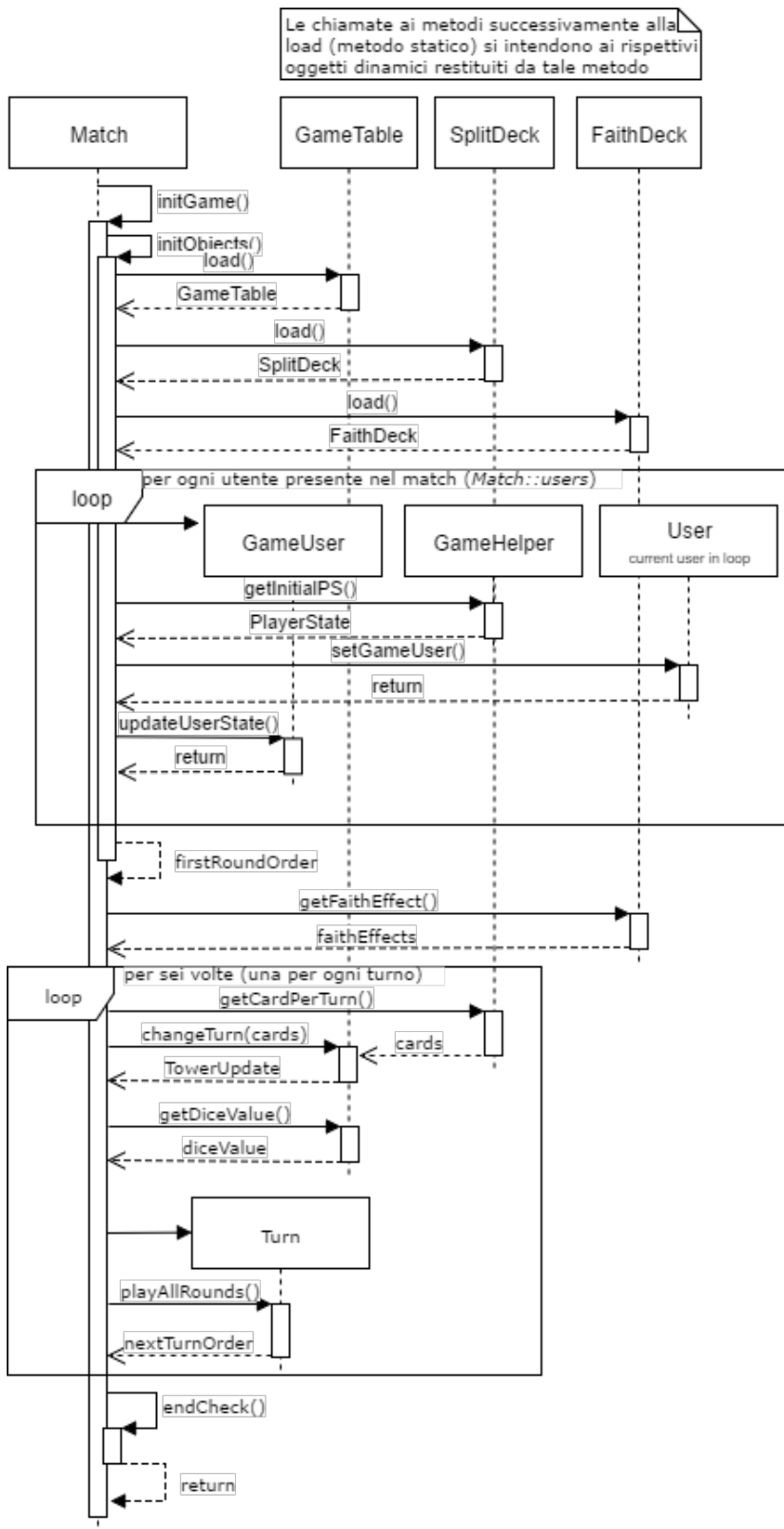
PlayerState		
getInUseDomestic()	Domestic	
setInUseDomestic(Domestic)		void
getCheckingPositionType()	PositionType	
setCheckingPositionType(PositionType)		void
getResources()	Map<ResourceType, Integer>	
setResources(Map<ResourceType, Integer>, boolean)		void
setPenalty(ResourceType, int)		void
getSlavePerDomesticValue()	int	
setSlavePerDomesticValue(int)		void
getEffects(EffectType)	List<Effect>	
addEffect(Effect)		void
addCard(Card)		void
getCards(CardType)	List<Card>	
setCostBonus(CardType, ResourceType, int)		void
getCostBonus(CardType)	Map<ResourceType, Integer>	
getGameUser()	GameUser	
clone()	PlayerState	

Le penalità sulle risorse sono applicate automaticamente tramite il setter *setResources*, come anche gli effetti delle carte aggiunte tramite *addCard*, mentre è necessario controllare i bonus applicabili tramite il getter *getCostBonus*, ma di ciò si occupa il metodo *Card::canBuy* prima di effettuare la verifica sui costi.

Match

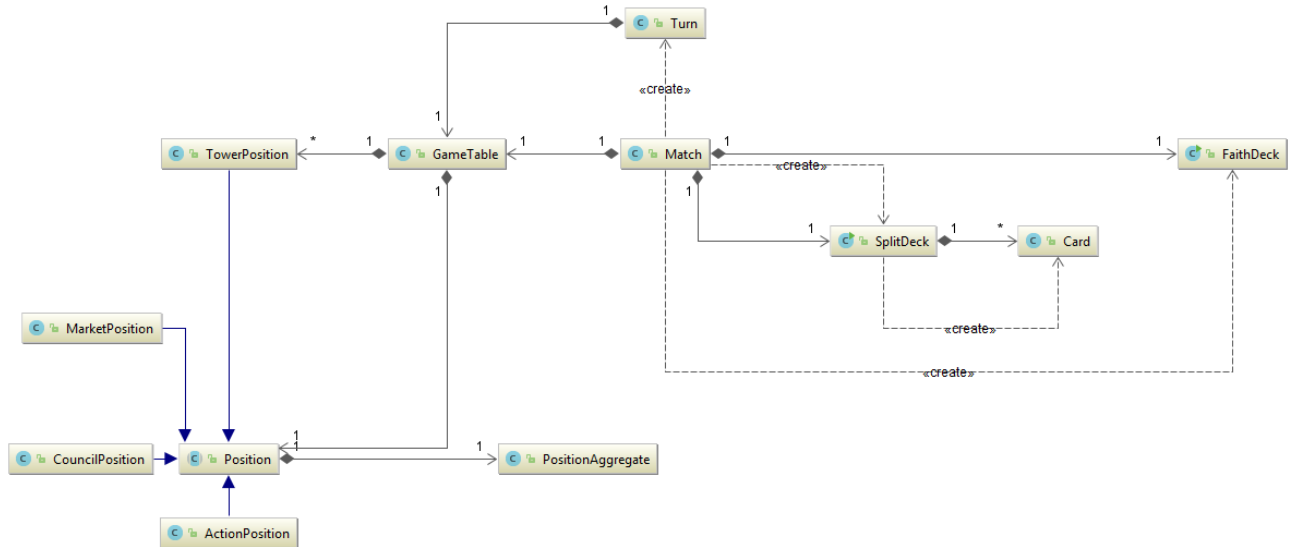
La classe *Match* si occupa di setup e di parte della gestione dello svolgimento del gioco. Ogni match è creato dalla classe *Lobby*, alla quale sono passati gli oggetti *User* appena dopo il login. Gli utenti vengono aggiunti alla partita tramite *Match::addUser* e una volta raggiunti

i due giocatori viene avviato un timer al termine del quale la partita ha inizio; questo timer viene riavviato se si aggiungono altri utenti, mentre al quarto partecipante la partita inizia immediatamente. Al termine della partita invece tutti i giocatori sono disconnessi e l'oggetto *Match* eliminato tramite *Lobby::clearMatch*.



La partita ha inizio con la chiamata a *Match::initGame* nel quale vengono caricati tutti gli oggetti di gioco necessari tramite *Match::initObjects*, quali posizioni della plancia e delle torri (*GameTable*), carte delle torri e del percorso fede, *GameUser* e relativi *PlayerState* per ogni utente; il metodo *initObjects* restituisce inoltre l'ordine di gioco per il primo turno. In seguito sono pescate le tre carte scomunica per poi entrare nel loop dei sei turni di gioco: posizionamento delle nuove carte sulle torri e reset di tutti le posizioni (*GameTable::changeTurn*), lancio dei dadi (*GameTable::getDiceValue*) e svolgimento del turno (*Turn::playAllRounds*). Una volta completata la partita viene chiamato *Match::endCheck* tramite il quale viene effettuata la conversione dei

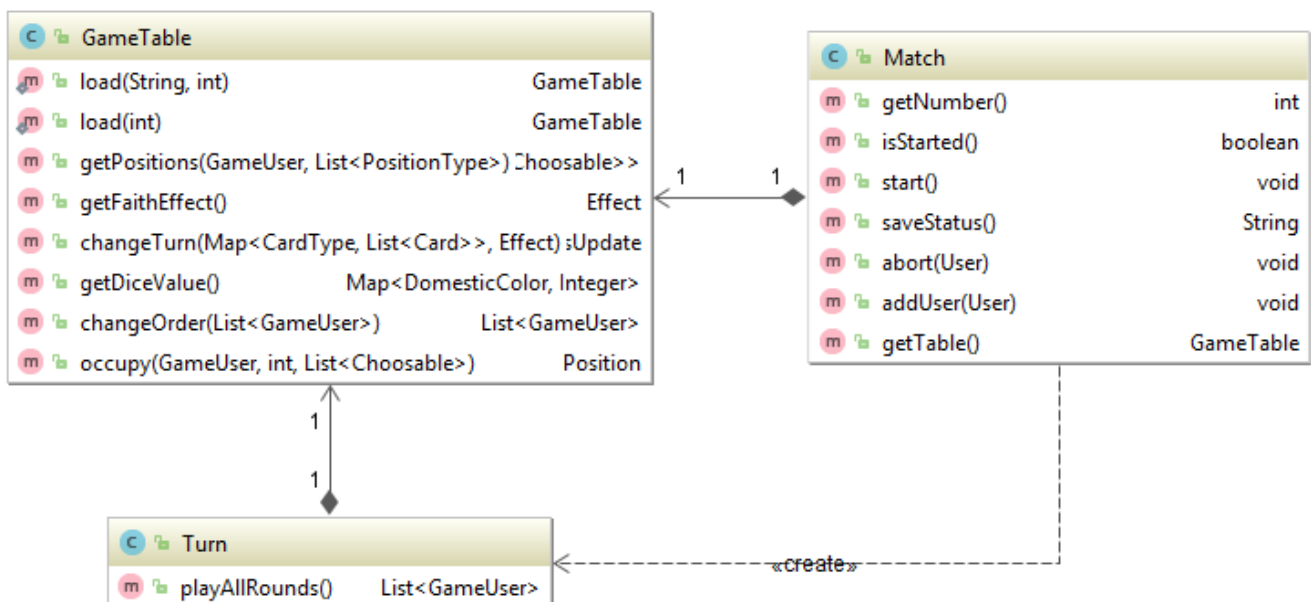
possedimenti di ogni giocatore in punti vittoria, tramite *Match::convertToVictory*, e viene stilata la classifica finale della partita: a questo punto il gioco è completato e si provvede all'housekeeping come descritto.



Ogni match viene eseguito su un thread separato gestito da un *SingleThreadPoolExecutor* modificato, *SingleThreadSchedExErr*, in modo da poter informare gli utenti e concludere il match in caso di eccezioni non gestite tramite il metodo *Match::executionErrorHandler*.

Turn

La classe *Turn* gestisce lo svolgimento di ognuno dei sei turni di gioco indipendenti. Ogni turno è suddiviso in quattro round (cinque nel caso di penalità per alcuni giocatori), durante



ognuno dei quali è richiesto a ciascun giocatore di effettuare un movimento (*Turn::waitMove*). I round si susseguono grazie alla chiamata ricorsiva di *Turn::playAllRounds*: in questo metodo viene richiesto ai giocatori di muovere e viene aggiornato l'ordine di gioco tramite *GameTable::changeRound*; inoltre al termine dei quattro (o cinque) round viene chiamato il metodo *Turn::faithCheck* che effettua per ogni giocatore il controllo del percorso fede, applicando gli effetti richiesti, e infine viene restituito l'ordine di gioco per iniziare il turno seguente.

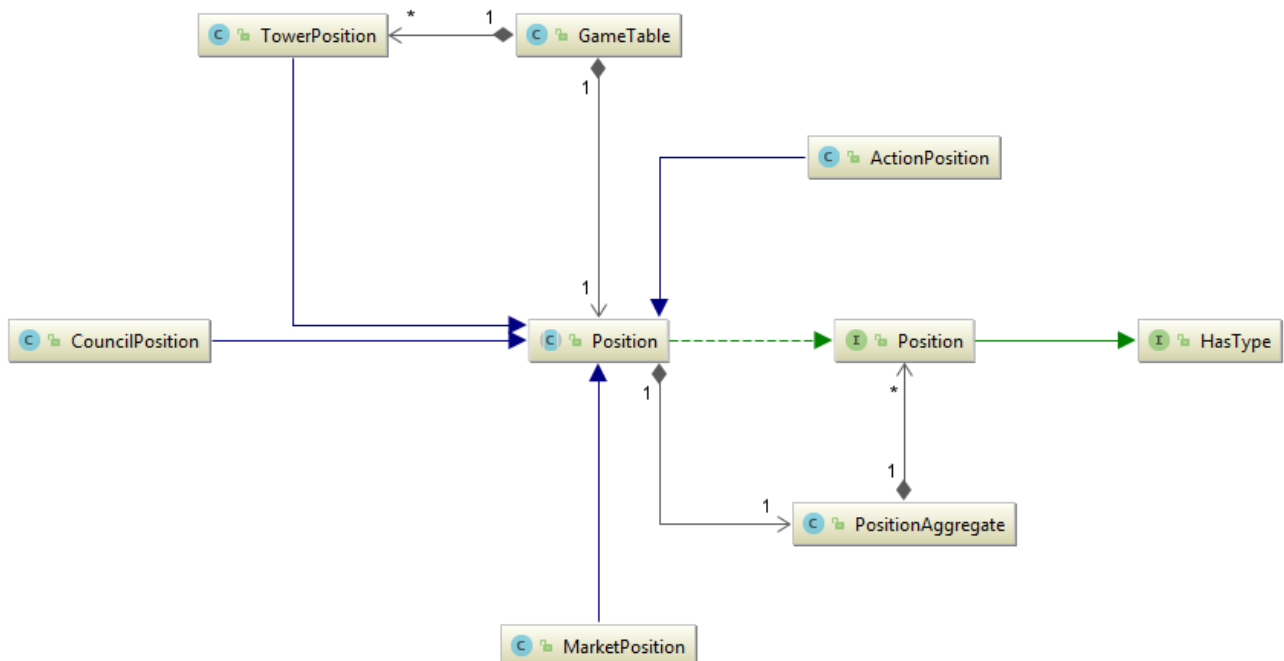
Oggetti di gioco

Ogni oggetto di gioco è stato riprodotto sotto forma di classe, in modo da poter controllare appieno le dinamiche di ogni azione. L'unica eccezione sono i punti e le risorse, per i quali viene utilizzato un *Enum* e una semplice mappa chiave-valore (*Map<Game.Usable.ResourceType, Integer>*).

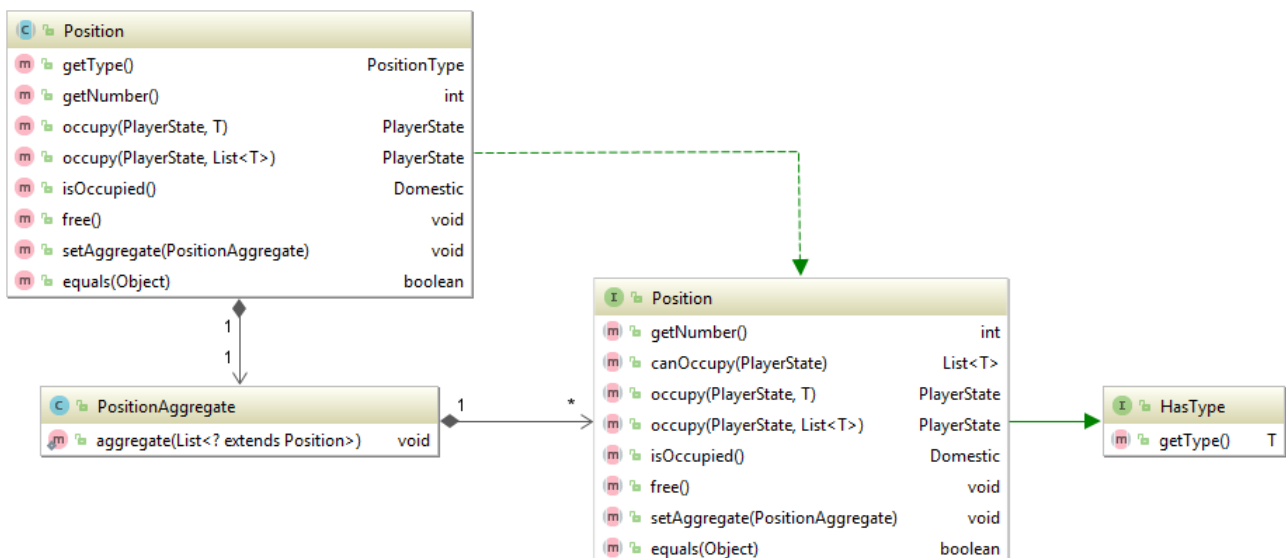
La plancia del giocatore è rappresentata dal *PlayerState* e i familiari dalla classe *Domestic*, entrambe già trattate.

Position e GameTable

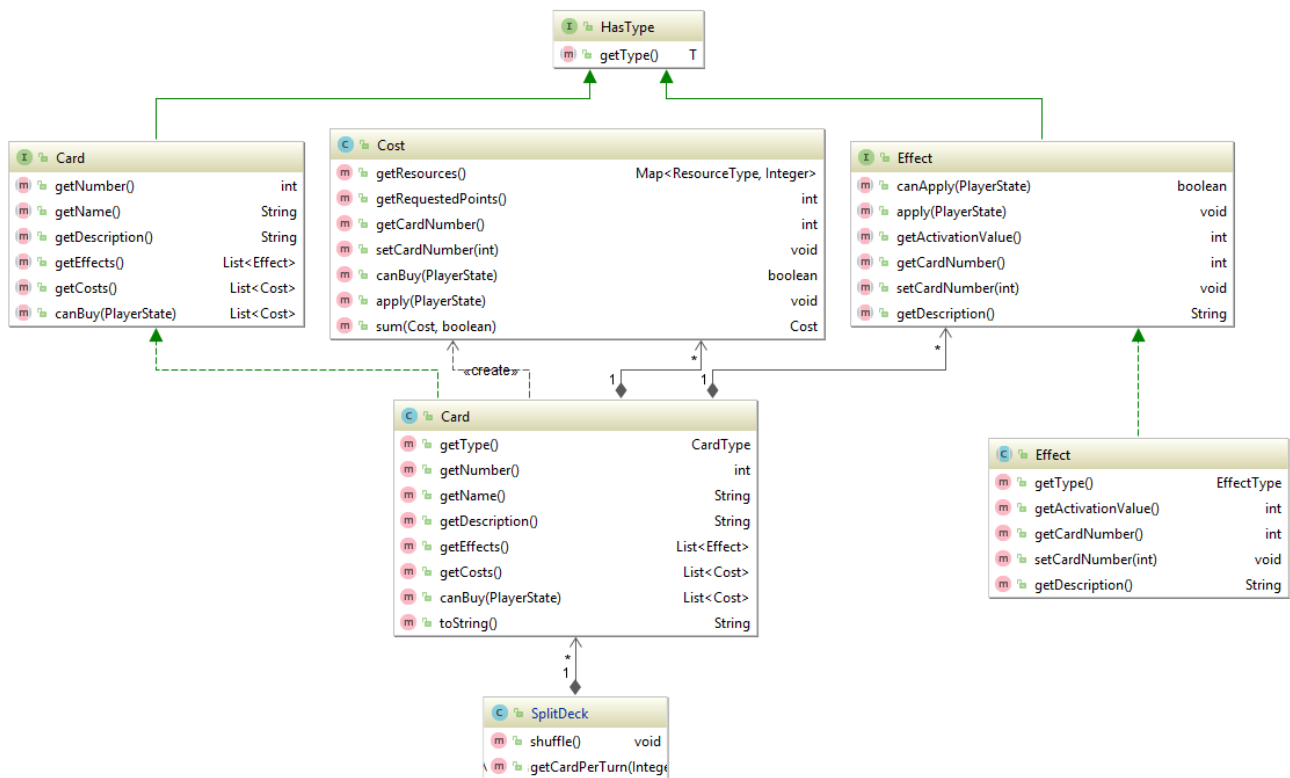
La plancia principale è rappresentata dalla classe *GameTable*, che racchiude tutte le posizioni di gioco; oltre ad ottenere informazioni sulle posizioni ed occuparle, rispettivamente tramite *getPositions* e *occupy*, sono presenti anche un metodo per l'aggiornamento dell'ordine di gioco (*changeOrder*), che calcola il nuovo ordine di gioco in base ai piazzamenti nel consiglio, un metodo per il cambio del turno (*changeTurn*), che aggiorna le carte sulle torri e l'effetto scomunica corrente e libera tutte le posizioni e un metodo per il "lancio" dei dadi (*getDiceValue*), che permette di ottenere il nuovo valore dei dadi quando necessario. L'oggetto in se non ha un costruttore pubblico, ma viene caricato interamente da un file in formato json tramite il metodo statico *GameTable::load*, che istanzia e inizializza tutte le posizioni necessarie in base al numero di giocatori della partita.



Alla base dell'oggetto *GameTable* vi è pertanto la classe astratta generica *Position*, con le sue quattro implementazioni specifiche: per ogni posizione infatti è dato un tipo (*Game.Positions.PositionType*), che ne identifica il generico (ossia se la posizione tratta di effetti da scegliere, nel caso di produzione e raccolto, o di costi negli altri casi), un numero identificativo univoco, un metodo *Position::free*, per liberare la posizione, e un duale *Position::occupy*, per occupare la posizione applicando gli effetti/costi scelti tra quelli ottenuti tramite il metodo *canOccupy*, che in base al *PlayerState* determina cosa è possibile fare nella posizione; infine ogni posizione appartiene ad un aggregato, impostato tramite *setAggregate*, (ad esempio, torre gialla, mercato, ecc.) in modo da garantire il corretto calcolo dei costi di occupazione.



Normalmente per occupare una posizione viene richiesta la lista dei costi sostenibili dal giocatore corrente tramite *Position::canBuy*, in seguito si sceglie uno di questi e si restituisce tramite *Position::occupy* insieme al *PlayerState*: a questo punto la posizione viene occupata, viene applicato l'effetto immediato, se presente, e viene aggiunta la carta al *PlayerState* nel caso di *TowerPosition*. Per quanto riguarda le *ActionPosition* invece il metodo *Position::canBuy* restituisce una lista di effetti attivabili tra i quali scegliere e infine da restituire sempre chiamando *Position::occupy*. All'interno del metodo *canBuy* sono applicati al *PlayerState* eventuali effetti permanenti ottenuti durante il gioco: in generale non è buona prassi utilizzare lo stesso oggetto *PlayerState* per il controllo consecutivo di più posizioni, poiché esso viene modificato durante l'esecuzione del metodo. Il processo di richiesta dei costi/effetti è gestito congiuntamente per tutte le posizioni da *GameTable::getPositions*, mentre per l'occupazione da *GameTable::occupy*.



Card e SplitDeck

Le carte delle torri sono racchiuse nella classe *SplitDeck*, caricata da json direttamente dal costruttore: le carte sono divise per periodo e mescolate di conseguenza tramite il metodo *SplitDeck::shuffle*, mentre l'accesso avviene pescando le carte per ogni turno, suddivise per tipo, mediante *SplitDeck::getCardPerTurn*.

La classe base *Server.Game.Cards.Card* racchiude in se la rappresentazione di ognuna delle 96 carte: ogni istanza è identificata da un numero univoco, possiede un tipo (*CardType*, dal quale si può risalire alla torre di appartenenza), un nome, una lista di effetti e una di costi. Per ottenere i costi sostenibili si utilizza il metodo *Card::canBuy* che effettua i dovuti controlli sul *PlayerState*.

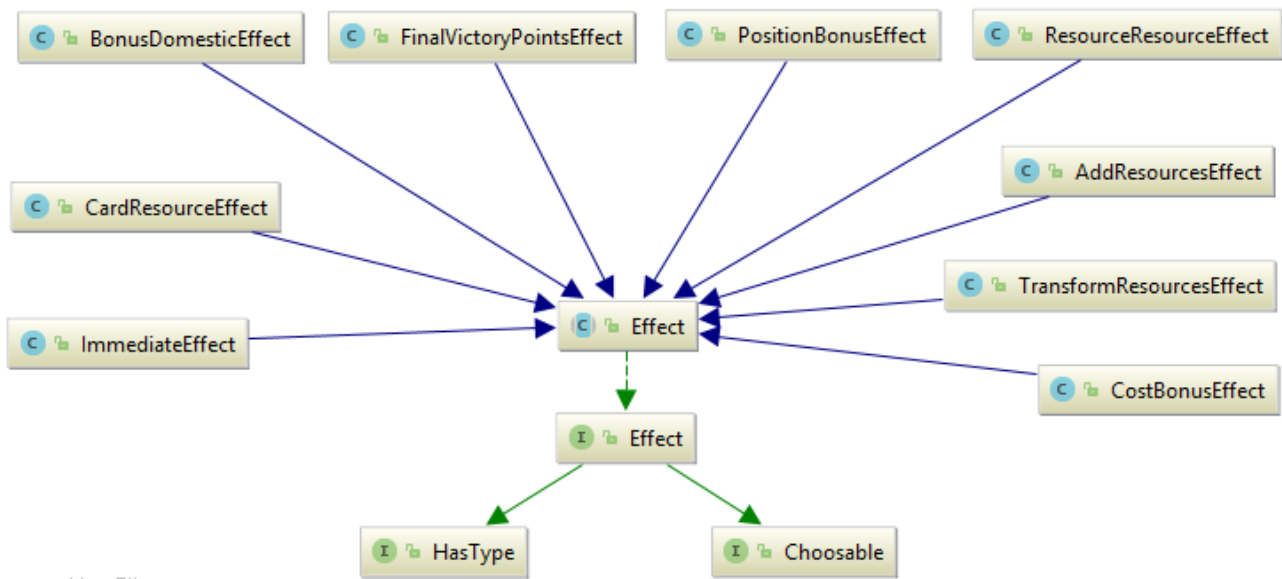
Cost e Effect

I due principali attori del gioco sono i costi, tramite i quali si perdono risorse e si ottengono effetti (comprando una carta o occupando una posizione), e gli effetti, tramite i quali si guadagnano punti e risorse.

I costi sono rappresentati dalla classe *Server.Game.Usable.Cost* che racchiude una lista di risorse necessarie a sostenere il costo e un eventuale livello di punti militari necessario. Il controllo della disponibilità di sufficienti risorse nel *PlayerState* avviene tramite *Cost::canBuy*,

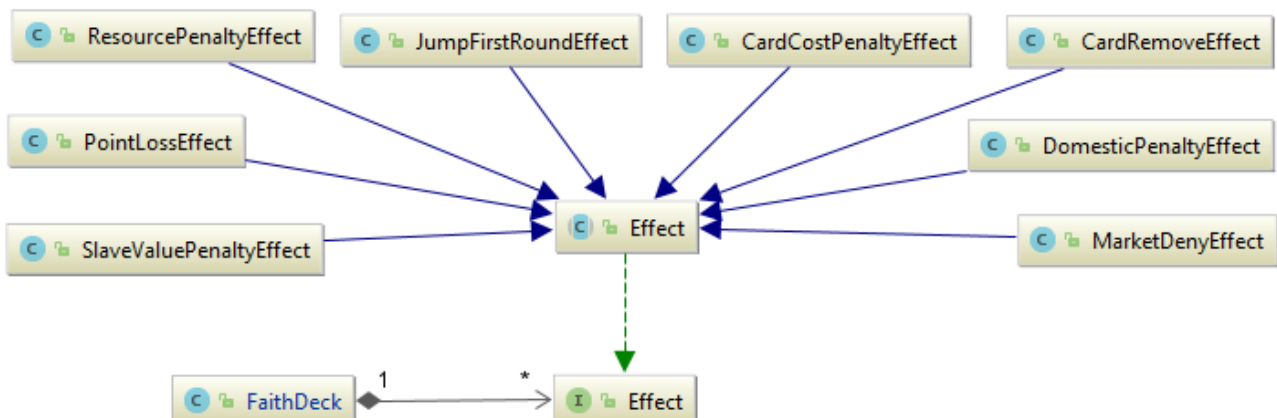
mentre la sottrazione delle risorse tramite *Cost::apply*; è presente anche un metodo per la somma/sottrazione di costi *Cost::sum*.

Gli effetti invece sono raggruppati dalla classe astratta *Server.Game.Effects.Effect* e presentano diverse implementazioni a seconda del tipo di effetto e dall'azione svolta: in generale ogni effetto ha un valore minimo di attivazione (valore del familiare necessario) e una descrizione; la possibilità di essere attivato è controllata tramite *Effect::canApply* e l'effetto è applicato tramite *Effect::apply*. Inoltre ogni effetto è classificato per tipo (*Game.Effects.EffectType*), in modo da poter controllare lo "scope" di attivazione.



FaithDeck

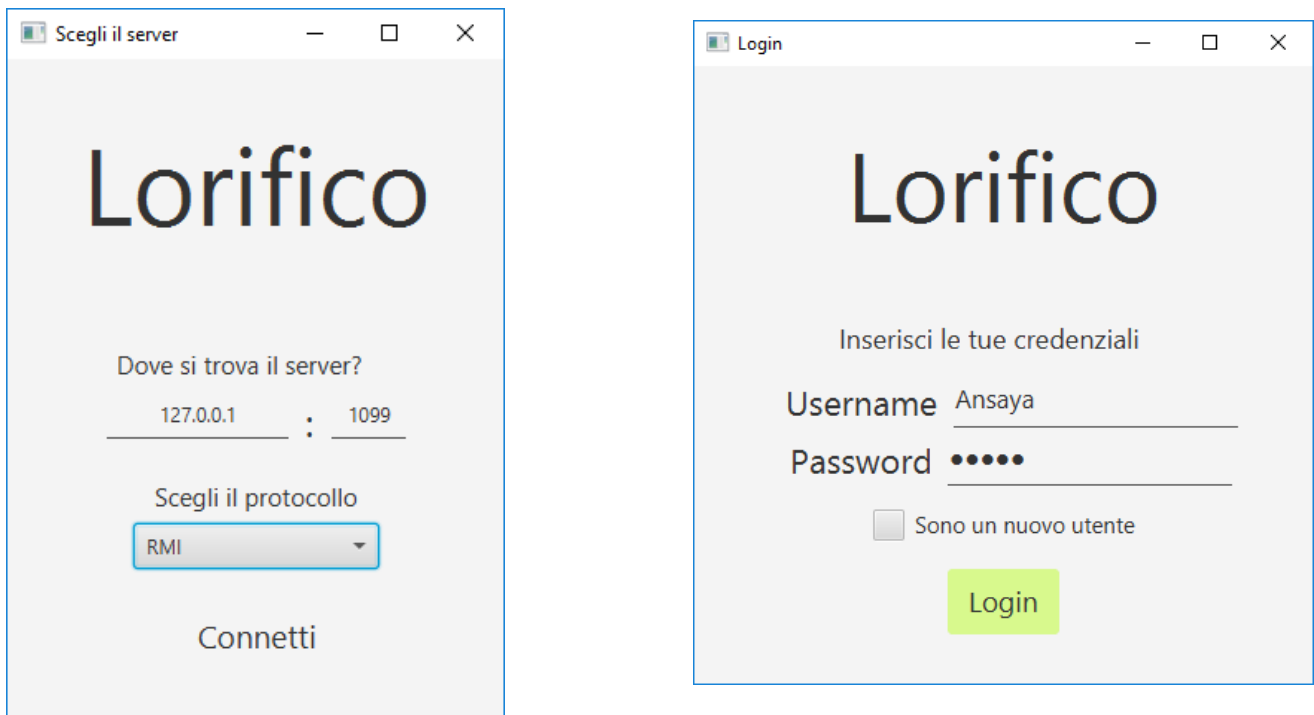
La classe *FaithDeck* è utilizzata per caricare gli effetti delle carte scomunica, suddivisi per periodo: il caricamento avviene da file json all'interno del costruttore. Sono presenti il metodo *shuffle* per mescolare le carte, mantenendo la suddivisione, e il metodo *getFaithEffect* per pescare le tre carte scomunica necessarie durante la partita. Oltre ad alcuni effetti standard presenti anche nelle carte sono stati implementati anche effetti utilizzati solamente nelle carte scomunica, ma comunque figli di *Server.Game.Effects.Effect*.



Manuale d'uso

Avviando l'applicazione client è necessario scegliere tra CLI e GUI scrivendo nella linea di comando "cli" oppure "javafx"; scegliendo la CLI verrete guidati direttamente dall'applicazione tramite la lista dei comandi disponibili per connessione e login, fino all'inizio del gioco: in ogni caso è possibile visualizzare una lista dei possibili comandi scrivendo "help" nella linea di comando.

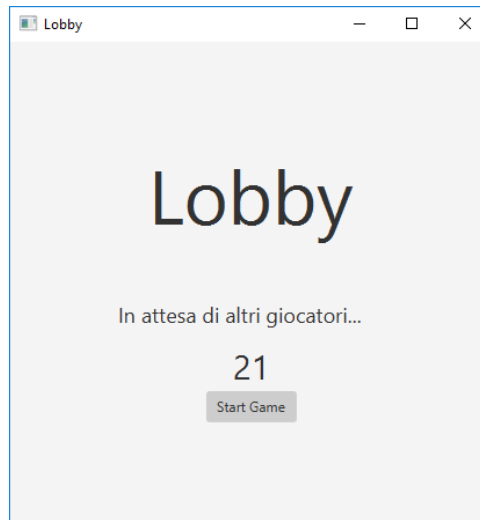
Per quanto riguarda la GUI, una volta noti l'indirizzo e la porta da utilizzare per la connessione al server (1099 per RMI, 8080 per socket) è sufficiente completare i campi della schermata iniziale e connettersi per accedere alla schermata di login, dalla quale è possibile registrarsi o autenticarsi con un account già esistente.



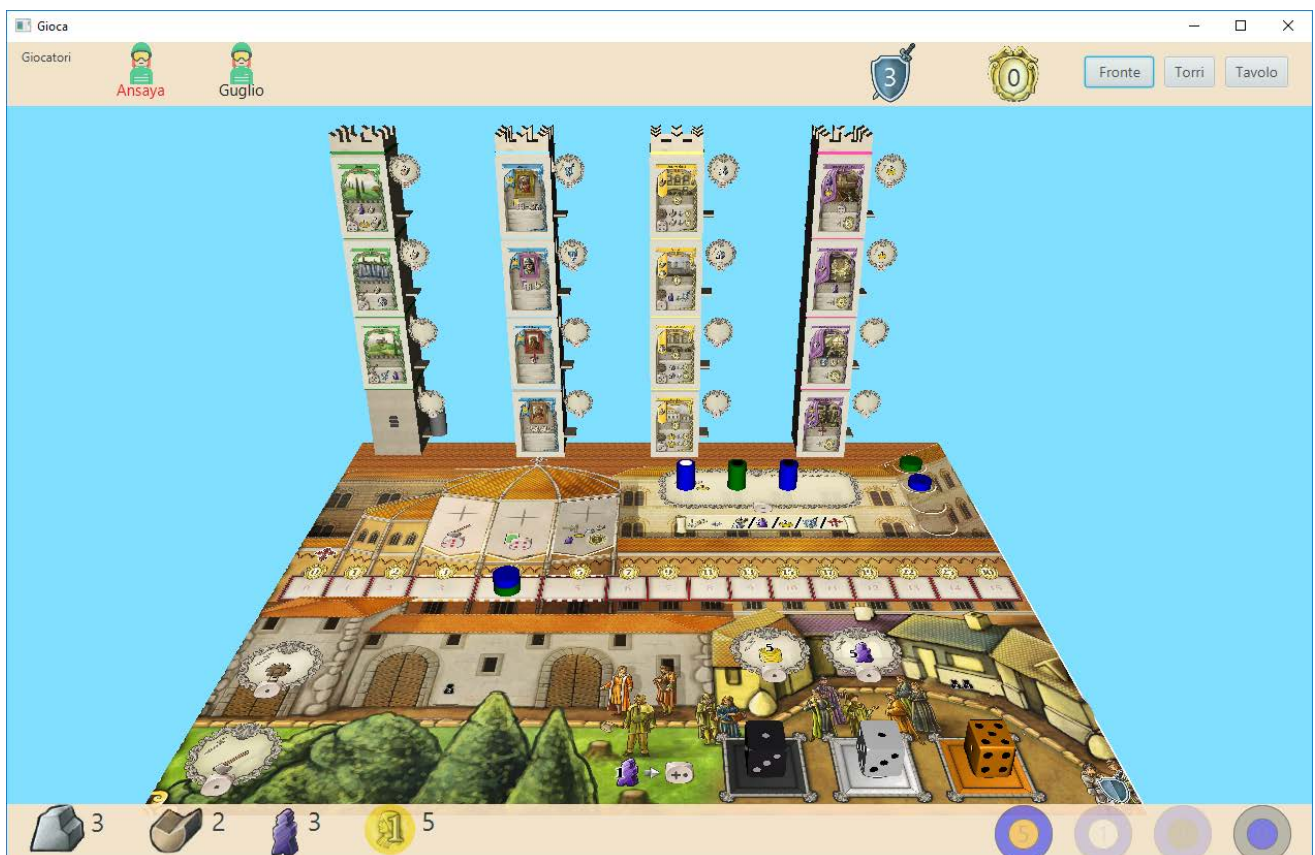
Una volta effettuato l'accesso si entrerà automaticamente in sala d'attesa per la partita corrente: appena il timer inizia a scorrere significa che almeno due giocatori sono presenti e la partita può cominciare; nel caso la sala d'attesa raggiunga i quattro giocatori la partita sarà avviata automaticamente. Inoltre è possibile avviare la partita immediatamente premendo il tasto "Start Game" una volta che il timer sta scorrendo.

All'avvio della partita si verrà automaticamente portati alla schermata del terreno di gioco, dalla quale sarà possibile osservare tutte le mosse dei giocatori ed effettuare le proprie. Sul fondo della vista sono presenti le quattro torri, mentre al centro la plancia principale; la

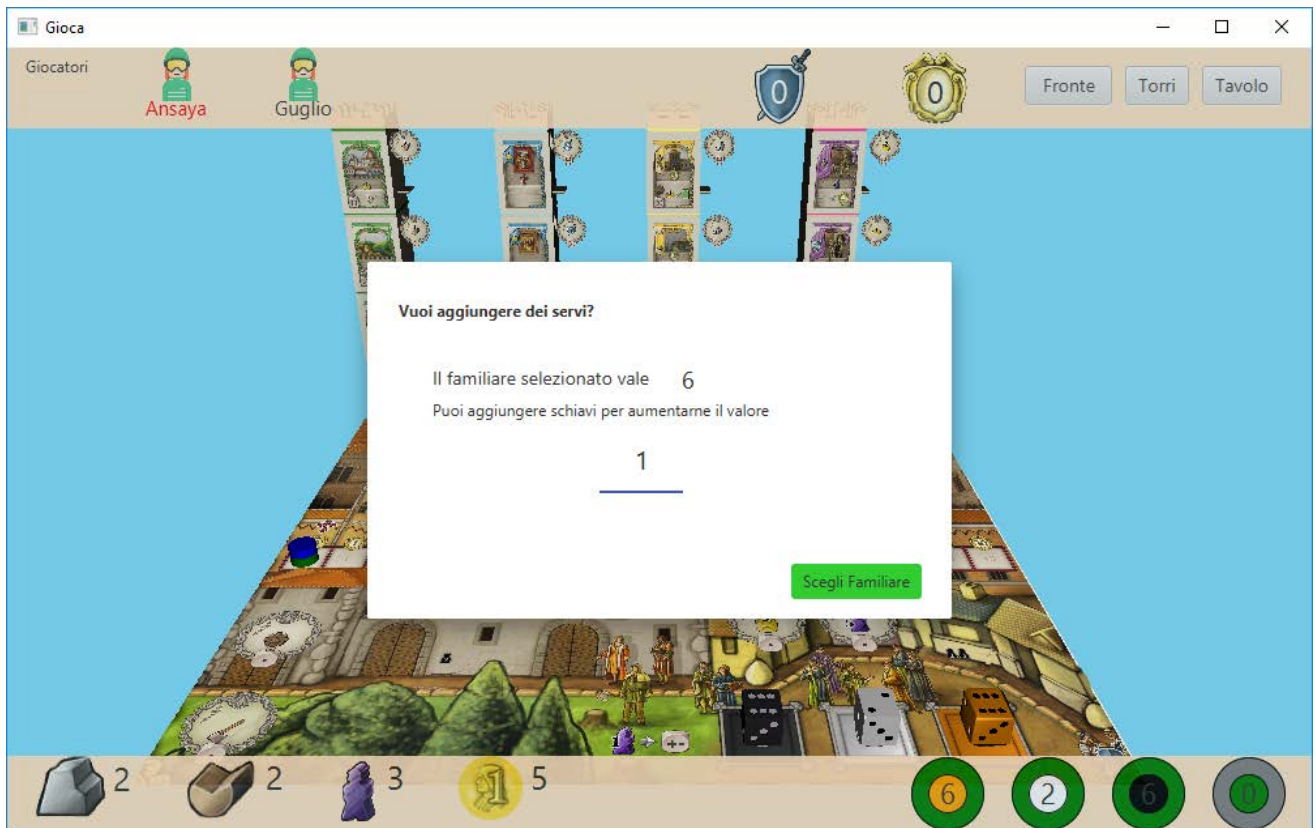
navigazione attraverso le tre posizioni base (tavolo, torri e fronte) avviene grazie ai tasti nell'angolo in alto a destra della finestra.



Le informazioni relative al proprio stato sono visualizzate nella barra inferiore: a sinistra le risorse, mentre a destra i familiari. Nella barra superiore si trovano, a sinistra, i partecipanti, cliccando sui quali è possibile visualizzare la rispettiva plancia giocatore, mentre a destra i punti militari e i punti vittoria relativi al proprio giocatore; nel caso il mouse si trovi sopra l'indicatore di un altro giocatore verranno visualizzati i punti relativi a quest'ultimo.



Per quanto riguarda i movimenti, il giocatore attualmente attivo è evidenziato in rosso e potrà muovere selezionando uno dei familiari nell'angolo in basso a destra. Una volta selezionato il numero di schiavi da applicare al familiare e confermata la scelta è sufficiente muovere il puntatore sulle carte o sulle posizioni della plancia per verificarne la disponibilità: in caso positivo le carte si solleveranno, mentre le posizioni si illumineranno al passaggio del mouse.



Per la selezione è necessario premere sulla carta / posizione desiderata e selezionare il costo da pagare o gli effetti da attivare per ogni carta in proprio possesso, nel caso di produzione o raccolto, oppure l'effetto del consiglio da attivare. Successivamente al completamento del turno sarà necessario attendere che ogni giocatore completi a sua volta la propria mossa: ogni movimento è comunque osservabile sul campo di gioco virtuale.

Una volta completata la partita sarà visualizzata la classifica finale che decreterà il vincitore e sarà possibile giocare una nuova partita effettuando nuovamente il login.