Documentazione Progetto Ingegneria Informatica

Fioravanti Massimo Gatti Michele Gianotti Federica

18 maggio 2018

Indice

1 Introduzione

1.1 Descrizione e analisi del problema

Le specifiche richiedono:

- L'implementazione client-server del gioco da tavola sagrada, sia in versione single che multi giocatore. La modalità a giocatore singolo richiede regole aggiuntive, quindi non è un sotto caso dell'altra modalità.
- La possibilità di ospitare partite multiple sia nel client che nel server. Ciò dovrebbe derivare trivialmente da una buona progettazione.
- La possibilità di caricare da file le carte Finestra. Le carte finestra sono poco più di matrici dotate di un paio di campi extra. Questo permette di trattarle, una volta caricate, come oggetti immutabili. Inoltre, per la loro piccola dimensione, di oggetti serializzabili. Questo riduce le carte finestra a poco più di una semplice classe dotata di un metodo che permette calcolare se la mossa di un giocatore è legale o meno. Il problema del loro utilizzo consiste unicamente nell'attività di trasferirle da server a client, poiché esse sono l'unica componente del gioco che non può essere già presente nel client al momento della creazione del gioco. La presenza di una carte caricabili a run time spinge a separare il costruttore del gioco dal suo set up in termini di regole. Risulta più semplice fare si che il costruttore ritorni il gioco in uno stato di pre set up, durante il quale è possibile caricare il mazzo delle carte finestra ed eventuali parametri di set up, come il numero di giocatori.
- Possibilità di unirsi a una partita in corso. Ciò significa che l'intero stato del gioco può essere trasferito in qualunque momento, da qualsiasi client.
- Il server deve automaticamente ignorare il turno di un giocatore non connesso. Questa richiesta è estremamente problematica, poiché implica che il server deve essere in qualche modo consapevole dell'esistenza delle regole e del loro significato, e contemporaneamente, a quale connessione è associato quale giocatore. Per impedire tale violazione di accesso ai dati è utile abbozzare una struttura simile ad un IA. Ogni volta che una partita inizia, una finta connessione viene stabilità con il server, e anziché essere gestita da un utente reale, un semplice oggetto si prende cura di eseguire i comandi dei giocatori mancanti, e di chiudere eventuali partite rimaste senza giocatori. Tale soluzione obbliga l'introduzione di uno stato protocollare finto, della possibilità di disporre di connessioni non giocanti all'interno del gioco, e della separazione tra il concetto di connessione e di giocatore.
- La presenza di un timer che avvii la partita in automatico. Un timer nel client previene la necessità che il server tenga traccia di tale evento.
- Un giocatore deve poter osservare tutte le azioni eseguite dai giocatori dall'inizio allo stato corrente della partita. Risulta quindi utile formulare le modifiche operate al model come azioni, dotate di nome e parametri, al fine di tenerne traccia e poterle presentare al giocatore.

Il gioco sagrada dispone delle seguenti caratteristiche:

- Pochissime azioni. Un turno di un giocatore è spesso composto dalla selezione e piazzamento di un singolo dado. Se si identifica i dadi e le locazioni tramite indici è possibile notare che la maggior parte di esse richiedono solo due o tre indici. Ciò significa che un azione è spesso composta da meno di 200 bit e tali bit sono composti per la maggior parte da Int. Ciò significa che sincronizzare le azioni tra client risulta essere estremamente poco costoso, a differenza di altre soluzioni, come ad esempio interrogare il server ogni volta che un dato è richiesto. L'intera sequenza di azioni da inizio a fine della partita è inferiore ai 10kb.

- Gioco a turni, sprovvisto di componenti reali. Ogni componente del gioco è dotata di un numero finito di configurazioni possibili, poiché non è necessario calcolare distanze reali tra componenti. Ciò significa che l'intero gioco può essere rappresentato da una macchina a stati finiti, il che ci concede di fare si che ogni membro di ogni classe del model sia un intero, un enum, una stringa o un riferimento a un altro membro del model. Implementare e calcolare l'hash di un oggetto del genere dovrebbe essere triviale, a patto che non sia dotato di riferimenti ciclici. Ciò ci spinge a rappresentare l'oggetto come un albero, così che sia possibile calcolare, attraverso l'hash, se la rappresentazione del gioco in un client sia identica a quella nel server. Ciò è utile per identificare errori di sincronizzazione.

- Eventi casuali. Alcuni eventi, come l'estrazione del dado, sono casuali. Per semplicità è utile isolare tutti gli eventi casuali e assicurasi che siano riferiti allo stesso RNG, così che sia più semplice identificare eventuali errori.

1.2 Overview della soluzione

Per i motivi sopra descritti l'implementazione dello strato di rete di Sagrada è adatta a seguire lo schema presentato da Microsotf per la realizzazione del codice di rete di Age Of Empire, e che rappresenta ora la fondazione delle best practice nell'ambito della programmazione di rete di videogiochi strategici. `https://www.gamasutra.com/view/feature/131503/1500_archers_on_a_288_network_.php` Sagrada non necessità di tutte le feature presentate in tale articolo, trattandosi di un gioco a turni non è necessario quantizzare il tempo in sotto unità, e la presenza di core multipli nelle macchine moderne previene la necessità di legare il thread di rete al thread della interfaccia grafica. Ciò che conta è ogni client, incluso il server, disponga di una copia del model simulata in maniera indipendente da ogni altra macchina, e che lo scopo del server sia quello di assicurarsi che i comandi legali inviati da ogni macchina siano ricevuti da tutti i giocatori connessi, nell'ordine di invio.

L'obbiettivo finale è quindi disporre di uno strato di rete del tutto invisibile sia all'interfaccia che ai dati sottostanti, poiché l'interfaccia grafica dovrebbe produrre le azioni che il giocatore intende eseguire, senza preoccuparsi chi riceva tali comandi, mentre la simulazione del gioco dovrebbe eseguire tali operazione, senza curarsi di chi li genera.

Questa decisione riguardo lo strato di rete forza una scelta architetturale riguardante il model. Tutte le modifiche della simulazione devono avvenire

esclusivamente attraverso azioni, e tali azioni devono poter essere trasferite da client a server e viceversa. La maniera più semplice di realizzare tale soluzione è assicurasi che tutte le azioni implementino l'interfaccia Serilizable, e che nessuna azione contenta membri non primitivi. Questo assicura che la risoluzione di ogni azione possa essere effettuata senza dover conoscere lo stato reale della memoria della macchina che ha emesso il comando, poiché non dipende dalla posizione locale degli oggetti.

Queste due semplici assunzioni producono i seguenti effetti:

- Il trasferimento dell'intero stato del gioco da server a client deriva trivialmente dal trasferimento di un comando, poiché una sequenza di comandi identifica univocamente uno stato del gioco.
- Salvataggi su disco sono triviali, poiché se l'itero stato può essere trasferito attraverso la rete allora può essere contenuto in un file.
- Trasferimento delle carte finestra estremamente semplificato. Se le carte finestra sono serializzabili, allora è possibile caricarle trivialmente da file inserirle in un azione e inviarla al client.
- Strato di rete game independant. Fin tanto che ogni azione è serializzabile allora la sincronizzazione tra server e client si riduce ad assicurasi che le stringhe emesse dai client siano ricevute da tutti gli altri giocatori in ordine corretto. Questo significa che tutti i giochi a turni possono essere gestiti dallo strato di rete attraverso queste regole.

1.3 Uso di feature del linguaggio

Serializzazione Java dispone di un potente meccanismo di serializzazione, che permette di ottenere oggetti senza conoscerne a priori la classe. Questo significa che l'introduzione di una nuova azione richieda una modifica della view per permetterne l'utilizzo, ma non richiede nessuna modifica al di fuori del codice della rete, del model o del controllore. Questo garantisce che estendere le operazioni di gioco sia quasi triviale, poiché richiede solamente la creazione di una classe dotata di un paio di funzioni.

Riflessione Poiché tutte le azioni hanno circa la stessa struttura è utile standardizzare la generazione di tool tip attraverso la riflessione e permette di creare in maniera automatica la linea di comando. Quando l'utente digita una stringa si assume che la prima parola della stringa sia il nome della classe del comando, e le altre parole siano gli interi che compongono i campi del oggetto che si vuole creare.

Espressioni lambda L'esistenza delle espressioni lambda permette di semplificare l'associazione dei componenti della gui alle operazioni equivalenti. Tale soluzione, unita all'utilizzo di file fxml rende quasi l'intera view modificabile senza dover apportare cambiamenti al codice sorgente.

2 Networking

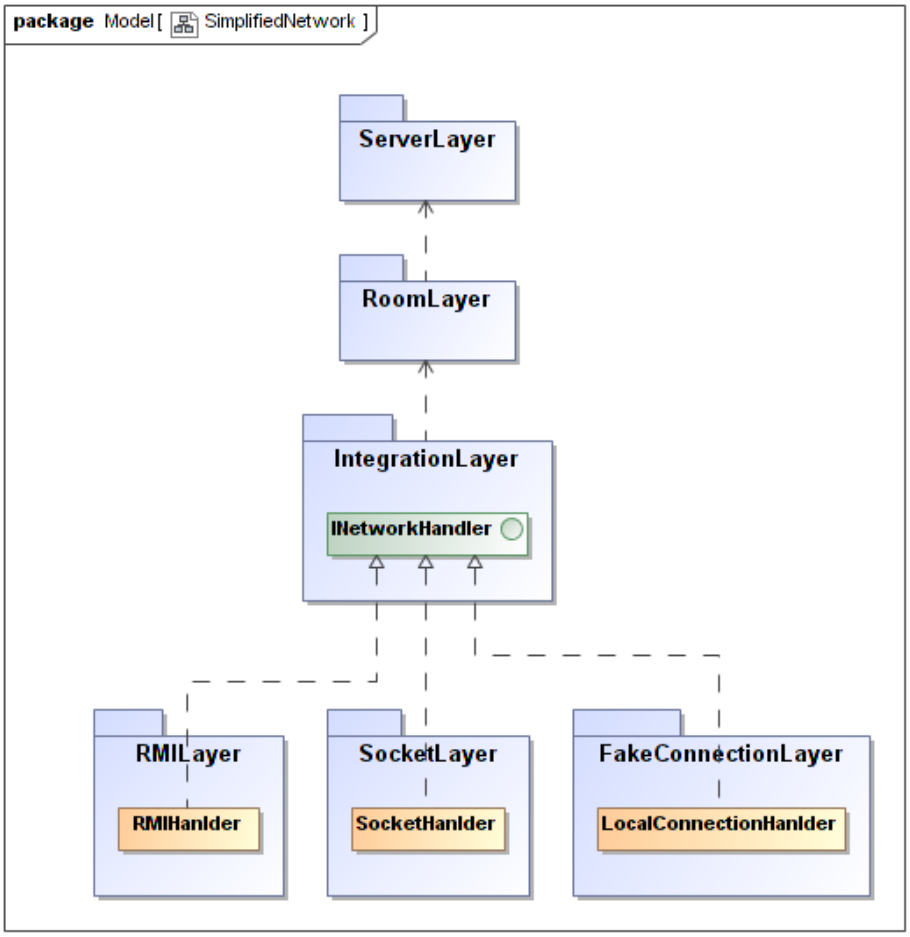
Come indicato nella introduzione, lo scopo della rete è quello di intercettare le comunicazioni tra controller e model, distribuire i comandi da eseguire a ogni client e assicurare che le simulazioni del gioco rimangano sincronizzate.

2.1 Requests e Enforces

Per comodità si userà la seguente notazione: si identificano con requests i messaggi inviati dal client e processati dal server, si identificano con enforces i messaggi inviati dal server e ricevuti da un client. Mentre il protocollo di rete deve garantire che sia le requests che le enforces devono essere consegnate e devono essere integre, le requests non devono essere necessariamente processate dal server, ed esso può decidere di ignorarle senza nessun acknowledgement. Le enforce, invece, devono necessariamente essere processate dal client, e il server assumerà che ciò è accaduto prima che la prossima enforce sia risolta.

2.2 Analisi ad alto livello

Il server deve essere in grado di gestire più di una partita contemporaneamente. Questo comporta l'introduzione di almeno due strati di rete, il livello di stanza, nel quale viene gestito tutto ciò che riguarda il gioco in corso. Ed il livello di server, nel quale viene gestita la creazione di stanze, l'inizio delle connessioni di un giocatore e le richieste di un giocatore di partecipare a una partita. Inoltre il server deve poter gestire contemporaneamente connessioni rmi e socket. Il che significa che è richiesto almeno un altro strato di rete, ovvero lo strato di integrazione, dove si nasconde la differenza tra le due implementazioni. Dato che si è precedentemente deciso di gestire la mancanza di giocatori all'esterno della rete allora è utile aggiungere un ulteriore componente dello strato di integrazione, cioè una finta connessione locale. Il vantaggio di disporre uno strato protocollare fittizio permette anche di rimuovere la necessità che le classi del ServerLayer espongano getter, poiché le informazioni possono essere ottenute da un client fittizio. Questo toglie la necessità di gestire un flusso di informazioni provenienti dall'alto e impedisce errori di threading.



2.3 Server Layer

Il server layer è lo strato di rete esposto all'utilizzatore. La principale delle sue occupazioni è tenere traccia delle stanze di gioco esistenti, accettare le richieste di creazioni di stanze, l'arrivo di nuovi giocatori e lo spostamento di un giocatore da una stanza ad un altra.

Moduli Poiché il server layer non deve essere a conoscenza di quali siano i protocolli che implementano l'integration layer, ne consegue che il server layer non può neanche sapere come una connessione venga stabilita. Questo ci spinge ad inserire il concetto di modulo di rete. Ogni layer che intende essere integrato nello stack protocollare deve implementare l'interfaccia INetworkModule. Tale interfaccia è composta dalle funzioni banali start, stop e isRunning, inoltre chi implementa tale interfaccia deve esporre una callback che egli deve invocare ogni volta che un utente si connette alla rete. Questa callback trasferisce un oggetto che implementa INetworkHandler, il quale avvolge la connessione reale con il client, e permette al RoomLayer di comunicare correttamente con l'utente.

Allacciandosi a questa callback il server può attendere nuove connessioni e una volta ricevute, inserirle in una stanza. Dalla rete possono quindi arrivare solo le seguenti richieste da parte dell'utente:

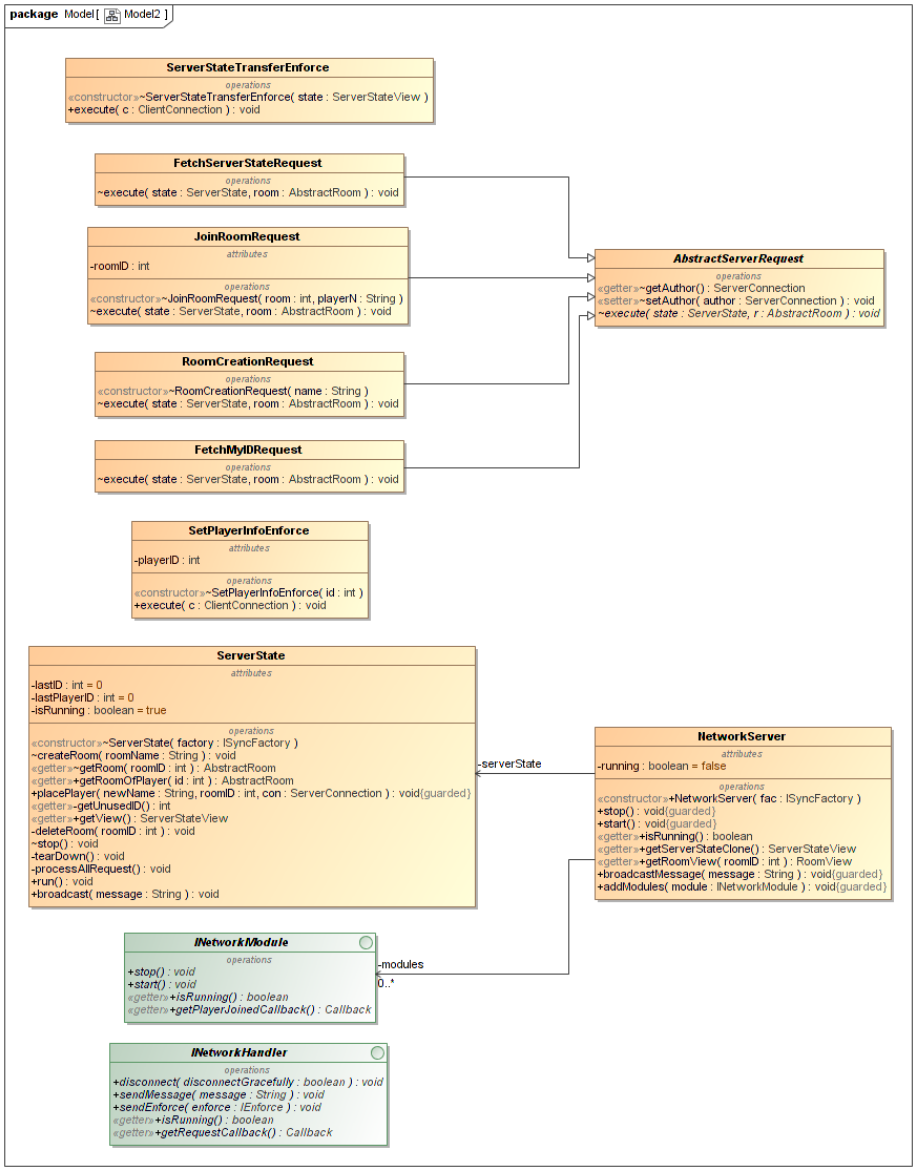
- **Fetch Server State Request:** Richiede al server che le informazioni sulle stanze presenti siano reinviate.
- **Join Room Request:** Richiede al server che la connessione che ha inviato questo messaggio venga trasferita nella stanza indicata.
- **Room Creation Request:** Richiede al server di creare una stanza con il nome indicato.
- **Fetch My ID Request:** Richiede al server di fornire l'id unico associato a tale connessione.

Il server deve quindi poter processare messaggi provenienti da una variegata forma di sorgenti. Risulta utile allora fare si che tutti le server request siano processate in maniera sequenziale in un thread a separato, e che quindi inviare una richiesta al server consista nell'aggiungerlo alla coda dei messaggi da processare. Questo garantisce, assieme alla mancanza di dati modificabili condivisi, che il server layer sia thread safe.

Mentre il server ha necessità di inviare solo le seguenti enforces:

- **Set Player Info Enforce:** Invia al client suo id corrente.
- **Server State Transfer Enforce:** Invia al client lo stato corrente del server, cioè il nome e il numero di giocatori in ogni stanza.

Il diagramma uml del server layer è quindi:



2.4 Room Layer

Il room layer è lo strato più complesso della rete, deve garantire che il gioco sotteso sia sincronizzato tra tutti i giocatori, deve tenere traccia di quali giocatori perdono la connessione, quali si invece si connettono e deve permettere lo scambio di messaggi tra giocatori.

Astrazioni a livello di stanza Per semplificare la gestione della partita in corso, è utile pensare alle connessioni e ai dati nella seguente maniera: Si differenzia tra gli oggetti di gioco appartenenti a un giocatore, la sedia di un giocatore ed il giocatore stesso. In particolare il giocatore è rappresentato dalla connessione al server, se il giocatore perde la connessione e si riconnette non ci curiamo di notare che si tratta della stessa persona, perché è possibile immaginare che nella realtà un giocatore smetta di giocare e un altra persona intervenga al suo posto. Semplicemente permettiamo alla nuova connessione di prendere la sedia che gli era una volta appartenuta, in modo da riprendere il suo posto. Tale sequenza di operazioni è compito del client, il server non si prenderà mai carico si eseguirla autonomamente. La sedia è l'indice assegnato a un giocatore nella stanza che permette di comprendere quali sono le componenti di gioco che appartengono a lui. Ad ogni nuovo giocatore è assegnato l'indice -1, che rappresenta il fatto che il giocatore vuole ricevere informazioni sulla progressione della partita ma non può eseguire attivamente azioni di gioco. Tali giocatori sono detti osservatori. Grazie a questa distinzione è possibile cambiare dinamicamente la sedia appartenente a un giocatore, sostituire giocatori mancanti, fare intervenire IA al posto di giocatori assenti. La differenza tra i componenti di un giocatore e la sua sedia è necessaria perché la rete non può conoscere il funzionamento del gioco, e quindi è necessario passare al gioco informazioni riguardanti chi sta tentando di eseguire un azione per scoprire se si tratta di un azione legale.

Un giocatore quando si collega a un server non può essere a conoscenza di quali stanze sono disposte ad accettare giocatori, di conseguenza è necessario che esista un luogo dove ospitare le connessioni che non hanno ancora deciso a quale gioco partecipare. Anziché gestire questa caratteristica a livello di server è più comodo assumere che i giocatori si uniscano automaticamente alla stanza di default del server, detta waiting room, identica alle stanze regolari, eccetto per il fatto che tutte le request diverse da server request vengono rifiutate.

ISync Lo strato di rete deve essere game agnostic, il che significa che lo strato di rete espone un interfaccia, e sta al gioco implementarla.

Tale interfaccia prende il nome di ISync, e i suoi metodi principali sono:

- **sendItem():** un oggetto serializzabile viene inviato al ISync, sta al gioco operare in maniera opportuna per assicurarsi che tale nuova aggiunta sia platform independant.
- **isItemGood():** ritorna true se l'oggetto che si vuole aggiungere al gioco può essere aggiunto.
- **getHash(int x):** ritorna che l'hash del gioco aveva assunto dopo aver processato l'item numerolol X. Ciò è utile per controllare se i vari client sono ancora sincronizzati.
- **clear():** reimposta il valore del gioco allo stesso stato che avrebbe avuto se fosse stato appena costruito.

In realtà non sarebbe necessaria l'implementazione di getHash, poiché se lo strato di rete è corretto e il gioco implementa l'interfaccia correttamente, allora non serve mai controllare se sono sincronizzati. Risulta però comodo poter eseguire questo controllo comunque, a fini di debugging.

Servizi offerti Come indicato, è necessario che il room layer sia capace di inviare al server layer le requests ricevute dall'integration layer e di trasferire verso l'integration layer le enforce ricevute dal server layer. La seconda parte è triviale, poiché non vi è particolare differenza tra enforce eseguite a livello di stanza da quelle eseguite a livello di server. Tutte le enforce sono quindi inviate al livello inferiore senza curarsi del contenuto.

Le request sono invece differenti, non è scopo del livello di stanza essere a conoscenza del significato semantico dei messaggi di rete, di conseguenza non vengono trattate come segue. Quando un client desidera mandare una server request la incapsula nella EncapsulationRoomRequest e la invia al livello inferiore. Quando una encapsulationRoomRequest è ricevuta da una stanza essa ne estrae messaggio incapsulato e lo aggiunge alla lista dei messaggi server pendenti. Quando il thread del server può allora svuotare la coda delle richieste della stanza e processarle come esso desidera. Ogni stanza deve allora essere dotata di un suo thread, il quale processa i messaggi provenienti al livello inferiore e li invia al livello superiore. Similarmente al caso del livello del server, la garanzia di essere thread safe deriva dal fatto che nessun dato mutabile è condiviso e tutti i messaggi inviati da un thread all'altro sono processati quando il thread relativo lo ritiene corretto.

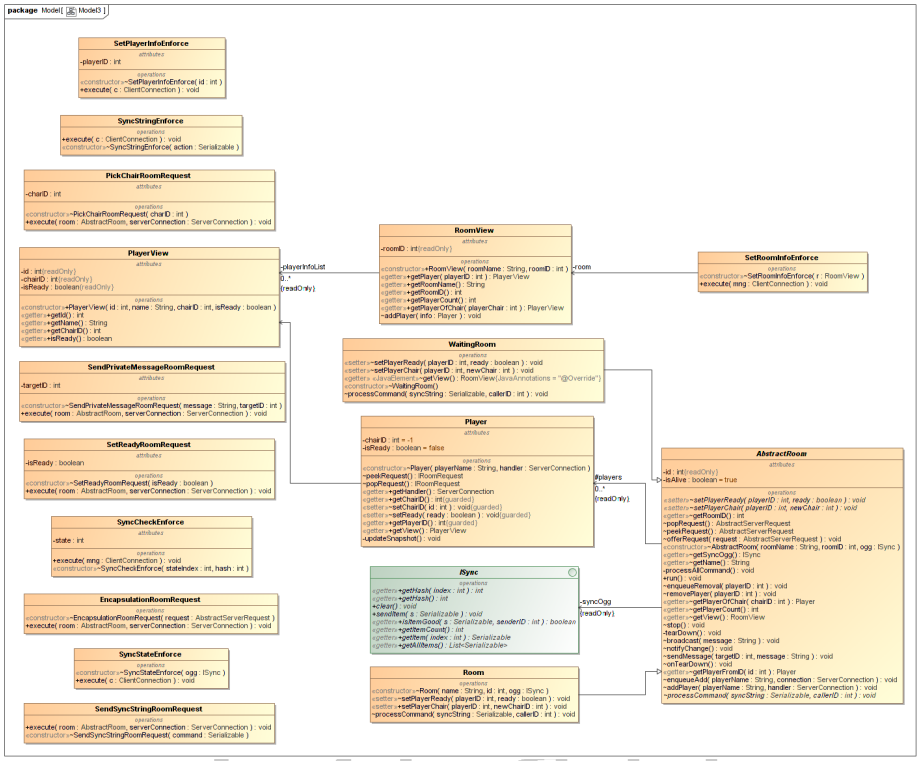
Ne risulta che i messaggi sono i seguenti:

- **Sync String Room Request:** invia l'oggetto serializzabile alla rete e assicura che, se accettato dal server, tale oggetto sia condiviso con tutti i partecipanti della partita.
- **Pick Chair ID:** richiede al server che una particolare sedia venga assegnata al giocatore che ha emesso questo messaggi.
- **Send Private Message Room Request:** richiede al server di inviare un messaggio di testo al giocatore con un particolare id.
- **Set Ready Room Request:** imposta lo stato di ready di questo giocatore. Quando tutti i giocatori sono stati impostati come ready allora il gioco può cominciare.
- **Encapsulation Room Request:** Invia una server request.

Le enforce che possono essere mandate da questo livello sono:

- **Sync String Enforce:** invia il prossimo oggetto serializzabile che deve essere computato dal client.
- **Send Message Enforce:** invia un messaggio di testo al giocatore di destinazione.
- **Send Room Info Enforce:** invia le informazioni sui giocatori al momento nella stanza. Questo include nomi, id e sedie utilizzate.

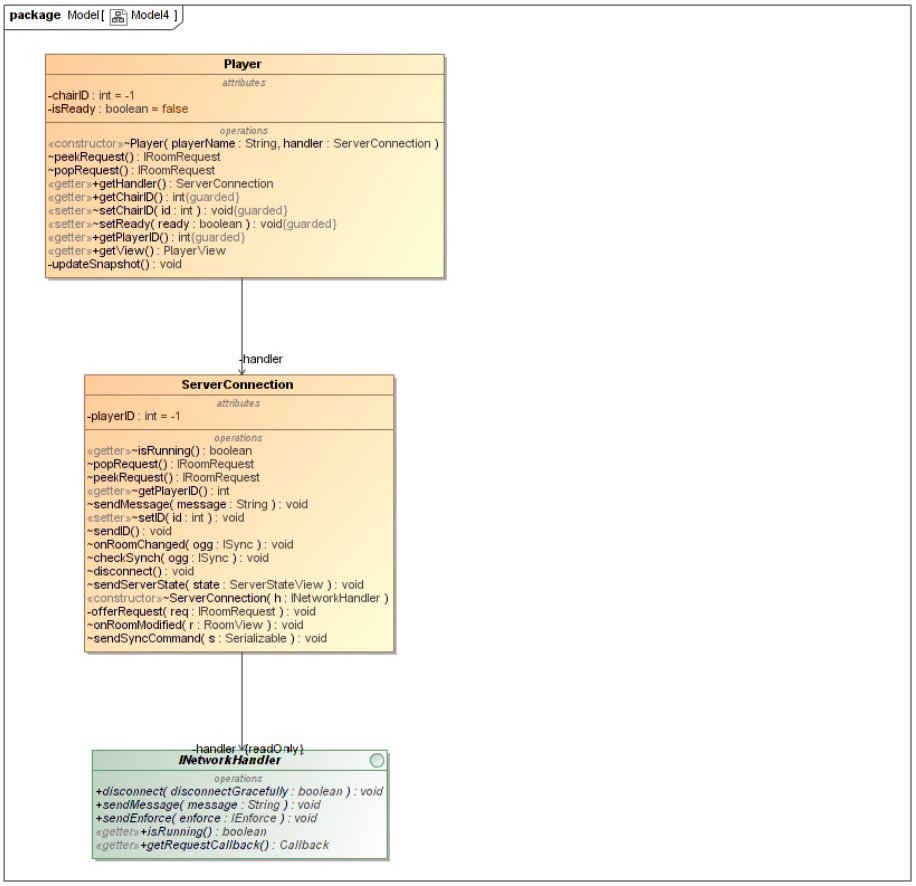
Il diagramma uml del room layer è quindi:



2.5 Integration Layer

Poiché vi sono diverse tecnologie di rete che devono coesistere nello stesso sistema allora è utile integrarle in un unico strato di rete. Socket dispone di tutte le caratteristiche di tcp, le chiamate di funzione non sono bloccanti, non è thread safe, è garantito l'arrivo dei messaggi, è garantito l'ordine dei messaggi e i messaggi sono testuali. RMI è uno strumento costruito sopra tcp e dispone di chiamate bloccanti, si possono trasferire oggetti serializzabili, multithreading da un lato della connessione è replicato dall'altro lato della connessione. Si potrebbe costruire un layer che aggiunga a tcp tutte queste caratteristiche, ma così facendo si terminerebbe con il replicare RMI. Molto più facile invece è aggiungere le funzionalità più semplici a socket, quali essere thread safe e poter trasferire oggetti, e rimuovere le feature più pericolose da rmi quali l'essere bloccanti e replicare i thread locali nella macchina remota. Risulta quindi che implementare queste richieste è parte degli strati inferiori, e che l'integration layer deve offrire l'interfaccia che essi devono implementare. Tale interfaccia prende il nome di INetworkHandler e chi la estende deve esporre una callback che viene chiamata ogni volta che egli riceve un messaggio e deve permettere di inviare room requests e stringhe rappresentanti dei messaggi. Il compito di essere thread safe è affidato al livello inferiore.

Il diagramma uml dell'integration layer è quindi:

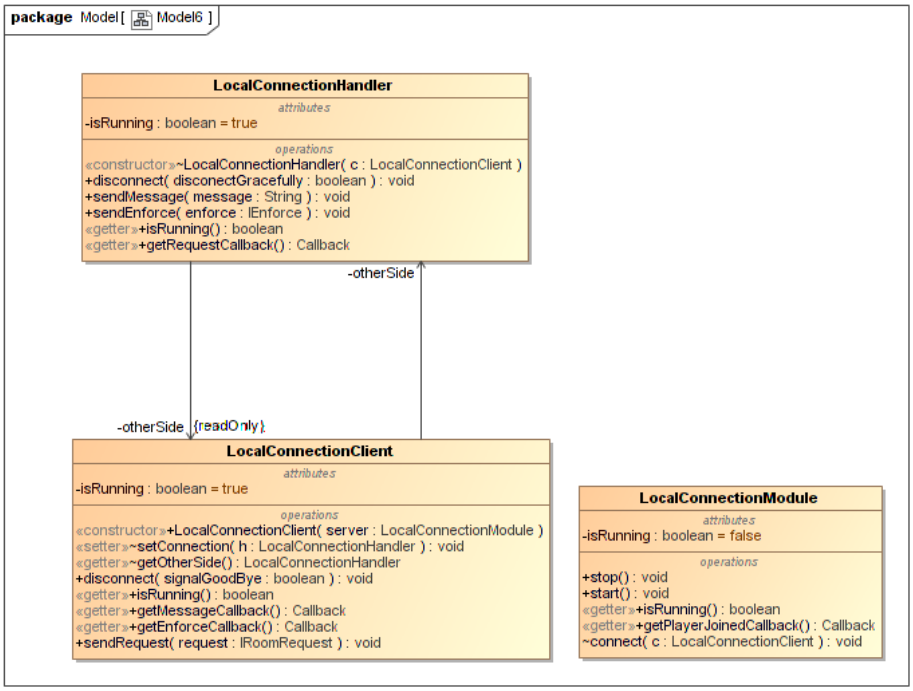


2.6 Fake Connection Layer

Per semplificare il debugging e per garantire che le connessioni locali non siano soggette a errori ambientali è utile implementare uno strato di rete fittizio. Sia la realizzazione di INetworkModule che di INetworkHandler è banale, poiché non necessita di affidarsi ad un successivo elemento dello stack protocollare e l'implementazione del client può chiamare direttamente le funzioni nel server. Quando si desidera creare una connessione nel client, si costruisce un LocalConnectionHandler, fornendogli il rifermento al modulo del server al quale si desidera connetterlo. L'oggetto appena costruito notifica il modulo della avvenuta connessione ed esso costruisce un INetworkHandler inserendo dentro di esso il riferimento al client. Infine l'handler è passato attraverso la callback, così che il server possa inserirlo nella stanza di attesa. Risulta quindi che entrambi i lati della connessione siano consapevoli di quale sia l'oggetto con il quale devono comunicare. Gli strati superiori garantiscono che la callback degli eventi ritorni immediatamente, concedendo al thread autore del messaggio di proseguire autonomamente. Questo garantisce che le chiamate non siano bloccanti e che

non si verifichino errori di threading. Non è necessaria l'aggiunta di ulteriori messaggi.

Il diagramma uml dell'integration layer è quindi:



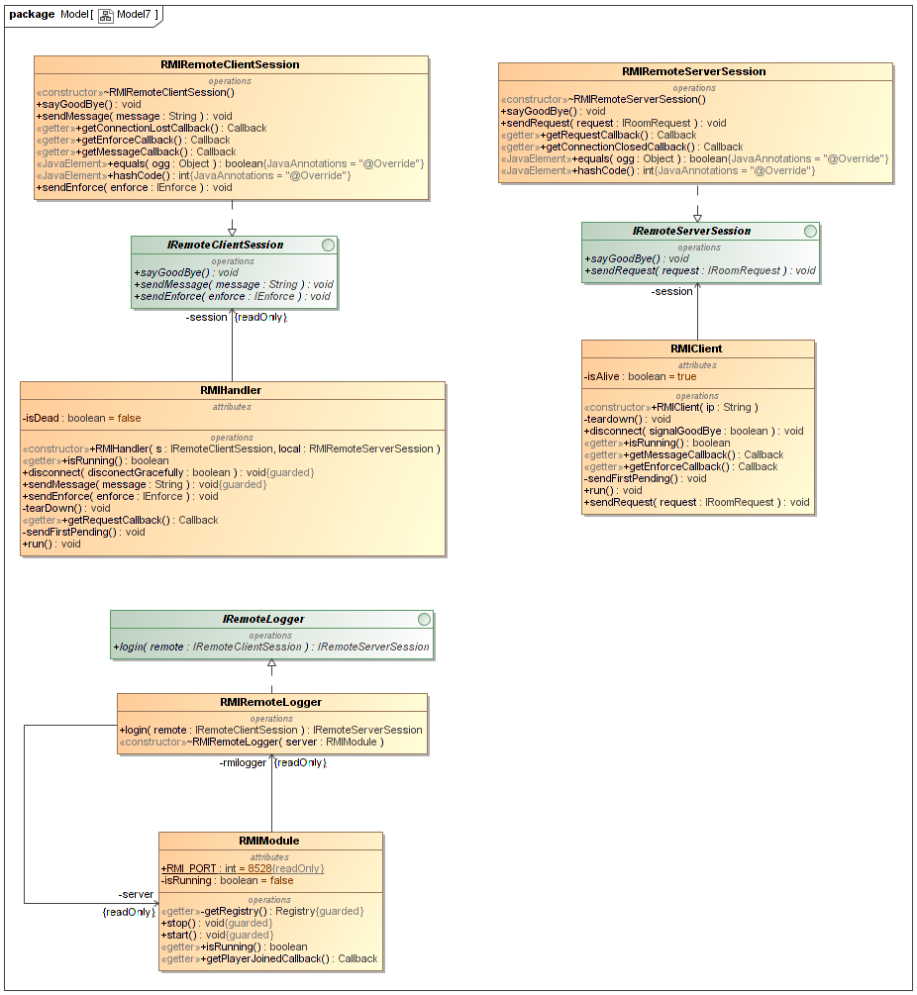
2.7 RMI

Grazie alle astrazioni esposte da rmi il suo schema concettuale è simile a quello delle fake connection. Client e server possono entrambi possedere un oggetto che dispone di un puntatore all'oggetto dell'altro. Non si desiderà però che un lato della connessione si debba fermare ad attendere che il messaggio sia completamente ricevuto dall'altro. La soluzione più semplice consiste nel creare un thread ogni volta che si intende inviare un messaggio. Soluzioni più performanti tipicamente includono un thread pool e un sistema di task, ma in tale contesto non possono essere utilizzate perché un thread pool dovrebbe comunque attendere la conferma della risoluzione della funzione invocata prima di poter proseguire all'invio del prossimo messaggio. Un altra soluzione consiste nel creare un thread per ogni connessione dedicato all'invio dei messaggi. Probabilmente ciò conduce a performance migliori se l'aumento dei giocatori aumenta, poiché richiede meno context switch, ma per mancanza di hardware non possiamo testare quale soluzione sia da adottare.

Risulta quindi necessario disporre di 3 oggetti remoti. Un remote che permette al giocatore di loggarsi. Un remote presente sul server e inviato dal logger al client. Un remote presente sul client e inviato al server nel momento del log in.

Una volta creati questi oggetti è triviale creare un comportamento identico a quello della fake connection.

Il diagramma uml del rmi layer è quindi:



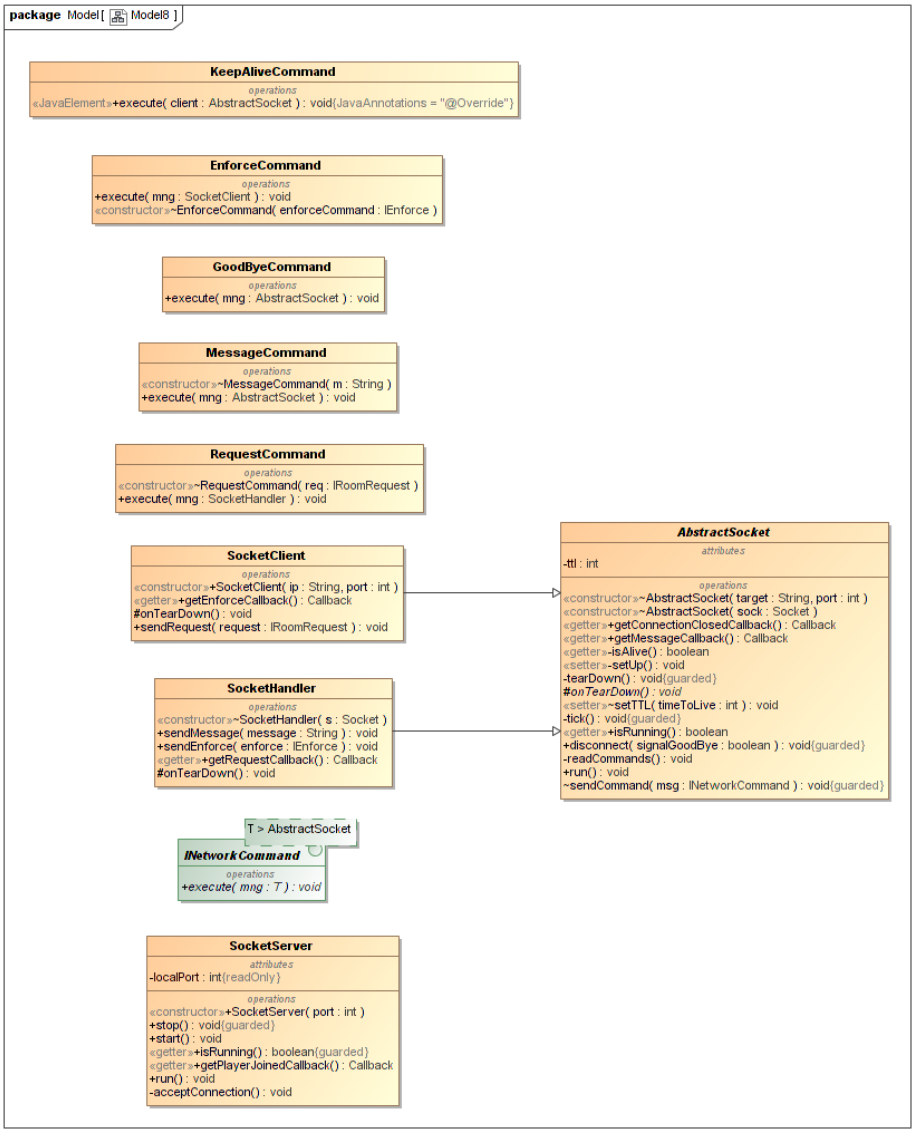
2.8 Socket Layer

Come è usuale si implementa il socket layer inserendo un thread che ha il compito di leggere i messaggi in arrivo, mentre eventuali autori di messaggi devono invocare le funzioni relative e attendere che i dati vengano inseriti nel buffer di invio prima di poter proseguire con le loro attività. Questo ci spinge a rendere gli invii del socket sincronizzati, poiché è l'unica maniera per assicurarsi che essi siano thread safe.

Socket può solo inviare stringhe, quindi a questo livello è necessario convertire i messaggi in testo. Dato vi sono vari tipi di messaggio, oltre alle room requests, allora si deve introdurre un ulteriore livello di messaggi: I server com-

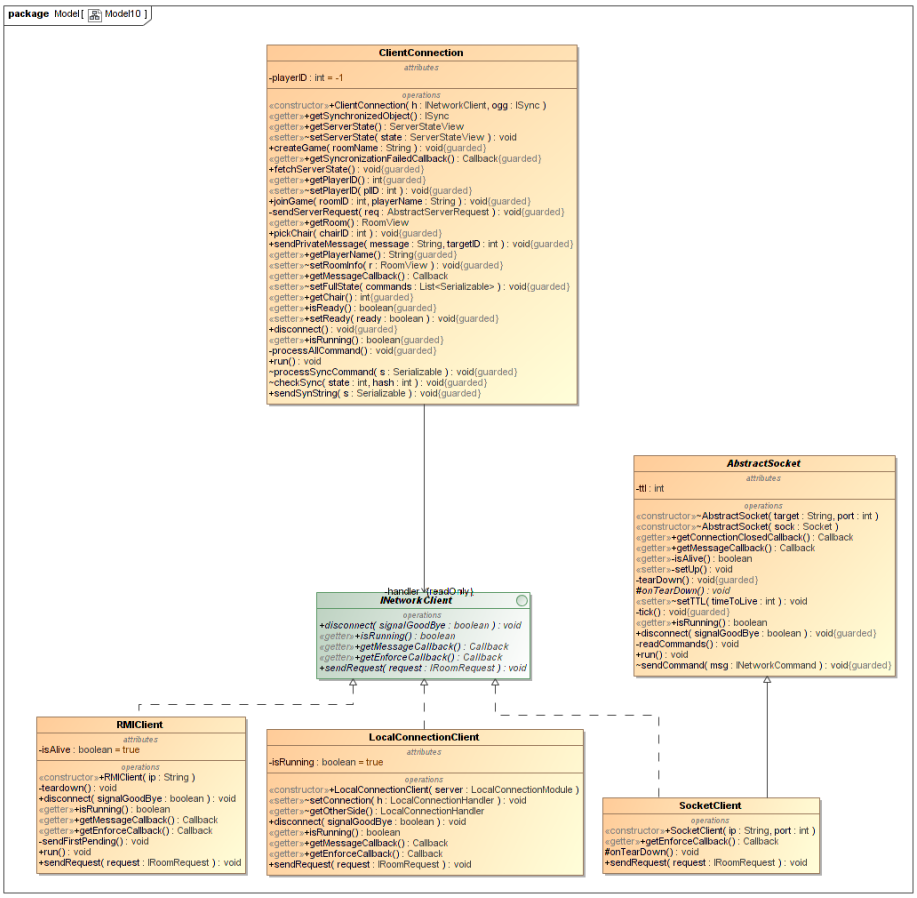
mand, inviati da client a server e i client command inviati da serve a client. Uno di questi messaggi avvolge le enforce, uno di questi messaggi avvolge le room requests, il resto implementa le altre funzioni.

Il diagramma uml della socket layer è quindi:



2.9 Client

Lo stack protocollare del client è invece di gran lunga inferiore, poiché è solo necessario che esso gestisca le informazioni di gioco ricevute dalla rete e offra una interfaccia funzionale alle possibili richieste del server.


package Model [Model10]
ClientConnection
attributes
-playerID : int = -1
operations
«constructor»+ClientConnection(h : INetworkClient, ogg : ISync)
«getter»+getSynchronizedObject() : ISync
«getter»+getServerState() : ServerStateView
«setter»~setServerState(state : ServerStateView) : void
+createGame(roomName : String) : void{guarded}
«getter»+getSyncronizationFailedCallback() : Callback{guarded}
+fetchServerState() : void{guarded}
«getter»+getPlayerID() : int{guarded}
«setter»~setPlayerID(pID : int) : void{guarded}
+joinGame(roomID : int, playerName : String) : void{guarded}
-sendServerRequest(req : AbstractServerRequest) : void{guarded}
«getter»+getRoom() : RoomView
+pickChair(chairID : int) : void{guarded}
+sendPrivateMessage(message : String, targetID : int) : void{guarded}
«getter»+getPlayerName() : String{guarded}
«setter»~setRoomInfo(r : RoomView) : void{guarded}
«getter»+getMessageCallback() : Callback
«setter»~setFullState(commands : List<Serializable>) : void{guarded}
«getter»+getChair() : int{guarded}
«getter»+isReady() : boolean{guarded}
«setter»+setReady(ready : boolean) : void{guarded}
+disconnect() : void{guarded}
«getter»+isRunning() : boolean{guarded}
-processAllCommand() : void{guarded}
+run() : void
~processSyncCommand(s : Serializable) : void{guarded}
~checkSync(state : int, hash : int) : void{guarded}
+sendSynString(s : Serializable) : void{guarded}
AbstractSocket
attributes
-ttl : int
operations
«constructor»~AbstractSocket(target : String, port : int)
«constructor»~AbstractSocket(sock : Socket)
«getter»+getConnectionClosedCallback() : Callback
«getter»+getMessageCallback() : Callback
«getter»+isAlive() : boolean
«setter»-setUp() : void
-tearDown() : void{guarded}
#onTearDown() : void
«setter»~setTTL(timeToLive : int) : void
-tick() : void{guarded}
«getter»+isRunning() : boolean
+disconnect(signalGoodBye : boolean) : void{guarded}
+readCommands() : void
+run() : void
~sendCommand(msg : INetworkCommand) : void{guarded}
-handler :{readOnly}
INetworkClient
operations
+disconnect(signalGoodBye : boolean) : void
«getter»+isRunning() : boolean
«getter»+getMessageCallback() : Callback
«getter»+getEnforceCallback() : Callback
+sendRequest(request : IRoomRequest) : void
RMIClient
attributes
-isAlive : boolean = true
operations
«constructor»+RMIClient(ip : String)
-teardown() : void
+disconnect(signalGoodBye : boolean) : void
«getter»+isRunning() : boolean
«getter»+getMessageCallback() : Callback
«getter»+getEnforceCallback() : Callback
-sendFirstPending() : void
+run() : void
+sendRequest(request : IRoomRequest) : void
LocalConnectionClient
attributes
-isRunning : boolean = true
operations
«constructor»+LocalConnectionClient(server : LocalConnectionModule)
«setter»+setConnection(h : LocalConnectionHandler) : void
«getter»+getOtherSide() : LocalConnectionHandler
+disconnect(signalGoodBye : boolean) : void
«getter»+isRunning() : boolean
«getter»+getMessageCallback() : Callback
«getter»+getEnforceCallback() : Callback
+sendRequest(request : IRoomRequest) : void
SocketClient
operations
«constructor»+SocketClient(ip : String, port : int)
«getter»+getEnforceCallback() : Callback
#onTearDown() : void
+sendRequest(request : IRoomRequest) : void

3 Model

3.1 Accesso alle informazioni

Oltre all'hiding delle informazioni legato all'ingegneria del software, si possono notare altri due tipi di hiding delle informazioni. Il primo è legato al fatto che le regole possono impedire a un giocatore di avere accesso a una componente del gioco fino a quando alcune condizioni sono soddisfatte, il secondo riguarda il fatto che tramite ispezione della memoria un giocatore potrebbe riuscire a ottenere accesso a informazioni che non li appartengono.

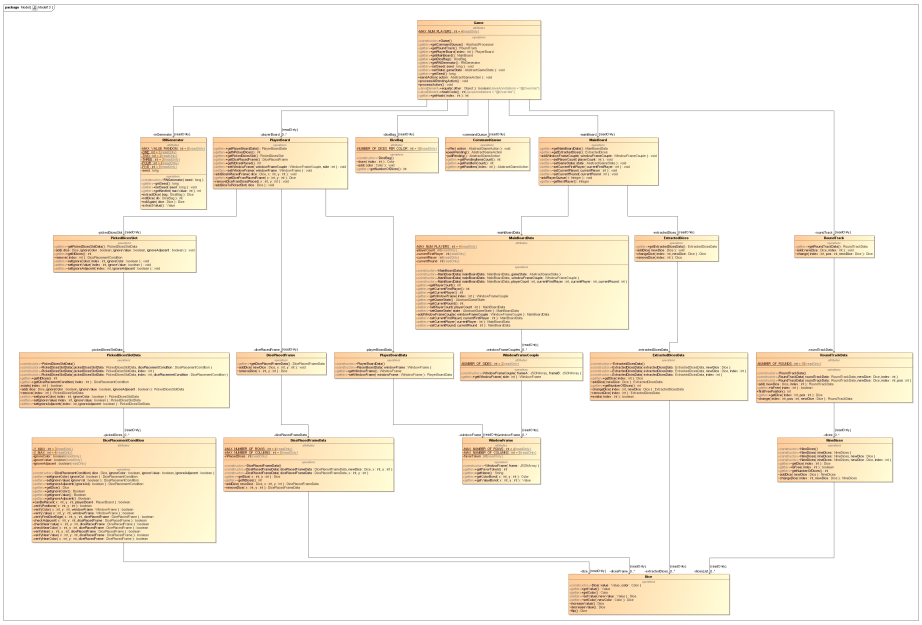
La prima problematica può essere risolta dalla view, anche se bisogna notare che in alcuni giochi l'information hiding può diventare così esteso da dover essere gestito all'interno del model. Fortunatamente sagrada non è così complesso.

La seconda problematica richiede soluzioni molto più complesse, e è stato indicato che non è necessario soddisfare tale requisito.

3.2 Componenti Implicite

Implementazioni triviali di un gioco da tavolo consistono nell'identificare come oggetto ogni componente del gioco. Sagrada si presta bene a questa soluzione, eccetto per un dettaglio, alcune carte fanno riferimento al fatto che devono essere attivate dopo aver selezionato un dado, similarmente altre carte affermano che è necessario ricollocare un dado già piazzato, ignorando alcune restrizioni di piazzamento. La soluzione è quindi dividere in due l'azione di piazzamento dei dadi. Si ottengono quindi l'azione di selezione di un dado tra quelli estratti, e un azione di piazzamento di un dado precedentemente selezionato. Quando una carta afferma di modificare le regole di piazzamento essa fa riferimento ai dadi selezionati ma non piazzati. Un problema che nasce da questa soluzione è che è necessario tenere traccia della sorgente di un dado per poter stabilire quali sono le modalità di piazzamento di un dado.

Un altra relazione implicita è quella legata al fatto che il setup del gioco è parte del gioco stesso e non una sequenza di operazioni da svolgersi prima dell'inizio. Selezionare il numero di giocatori, preparare il tavolo in funzione di tale numero e selezionare la scheda da usare sono tutte operazioni che sono parte del gioco. Si può pensare allora che il gioco appena costruito rappresenti il gioco ancora imballato, e che in seguito il gioco si evolva attraverso la selezione della tipologia di partita, del set up e delle azioni di gioco.



3.3 Azioni

Nella maggior parte dei prodotti si può distinguere tra il concetto di model e di azioni che operano sul model. Riteniamo che nel contesto della rappresentazione digitale di un gioco ciò non sia vero, poiché si può immaginare che un azione operata da un giocatore possa essere soggetta a azioni di altri giocatori che la negano o la modificano in qualche maniera. In alternativa potrebbe capitare che l'effetto di un azione sia parzialmente o interamente costruito a run time. Questo ci spinge a considerare le azioni come parte del model e a considerare come parte del controller solo l'attivazione di tali funzioni.

Si ha allora che l'unico setter pubblico esposto dall'intero pacchetto può essere solo la funzione che permette di inviare comandi. Ogni altra funzione può essere invece package-private o private. Il vantaggio di tale soluzione è che ciò permette di assicurarsi che tutti e soli li stati raggiungibili dal gioco siano stati che effettivamente possono essere raggiunti durante una partita. Ciò concede inoltre di avere forti restrizioni riguardo ciò che la view deve essere in grado di mostrare e non.

Infine, la possibilità di osservare le azioni giocate è parte delle specifiche, quindi utilizzare un sistema a ad azioni risolve automaticamente tale problema.

La conseguenza più significativa di tale soluzione è che risulta quindi che gli stati del model sono tutti e solo gli stati legali di una partita.

3.4 Carte Strumento

La parte più complessa del gioco consiste nella gestione delle carte strumento, poiché esse sono di varia natura e con effetti assai differenti. La maniera più semplice per gestirle è fare si che quando si attiva una carta non la si risolve

immediatamente, si entra invece in uno stato di gioco nel quale esso è disposto ad accettare i parametri secondi i quali tale carta verrà risolta. Deve quindi essere presente un azione per impostare ogni singolo parametro di attivazione di ogni carta. Volendo si potrebbe implementare tale funzione tramite riflessione, ma se fosse necessario in futuro fornire parametri più complessi tale soluzione non sarebbe adatta. Poiché la quantità di carte è ridotta si è optato per implementare manualmente ognuna di esse.

4 Threading

Come ci si aspetta, tutti le componenti di un programma dovrebbero essere thread safe. Poiché il volume dei cambiamenti è ridotto, non c'è motivo di utilizzare strutture di condivisione mutabili. Più semplice è invece sfruttare strutture immutabili, così da non doversi preoccupare della sincronizzazione.

I principali thread persistenti all'interno del programma sono:

- **Server Layer Thread:** processa le richieste di entrata e uscita da una stanza.
- **Room Layer Thread:** processa le richieste di stanza, mantiene sincronzzato il gioco, gestisce il dropout dei giocatori.
- **Socket Reading Layer:** legge i messaggi dagli strati inferiori e li propaga agli strati superiori.
- **Java fx thread:** tutte le modifiche di java fx devono essere operate all'interno del suo thread. Fortunatamente java fx espone la funzione runLater che permette di assicurare che le funzioni vengano chiamate all'interno del thread corretto.
- **Client Network Thread:**Le modiche al gioco devono essere effettuate all'interno del thread della rete, al fine di assicurare che la rete possa essere certa che il gioco è nello stato corretto.

Per quanto riguarda il model esso deve assicurare che il thread della gui possa leggere informazioni mentre esse vengono modificate dal thread della rete. Sagrada fortunatamente non è eccessivamente complesso, e la quantità di modifiche della memoria per secondo è ridotta, quindi è possibile adottare una soluzione basata su oggetti immutabili anziché basata su mutex. La usa quindi una struttura simile Plmpl, adattata a java: Si mantengono allora tutti gli oggetti osservabili come oggetti immutabili, ma si spostano tutti i dati non immutabili da essi contenuti in un oggetto immutabile. Ogni volta che si intende modificare il contenuto di un oggetto osservabile si rigenera l'oggetto dei dati modificato a dovere e si notifica gli osservatori del cambiamento. Risulta inoltre necessario imporre che tutti gli oggetti mutabili abbiano un life time identico a quello di game, poiché questo garantisce che ogni metodo o ritorna un riferimento che costante per tutta la vita del gioco ed è quindi thread safe o ritorna un rifermento che è immutabile e quindi thread safe.