Version 1.1

# Design

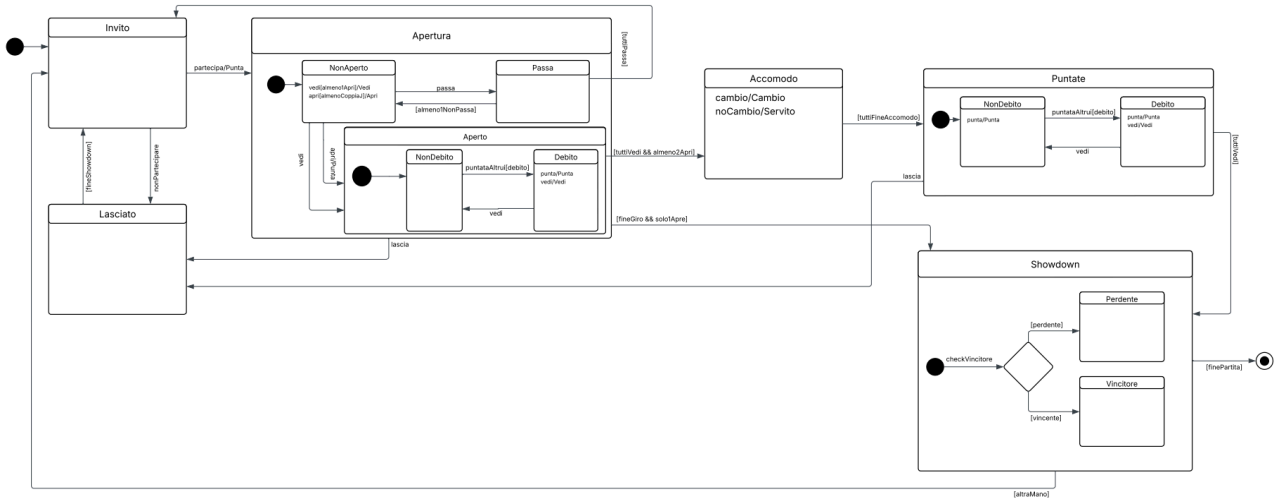
| Versione | Data | Descrizione |
|---|---|---|
| 1.0 | 19 Gennaio 2026 | Prima versione del documento |
| 1.1 | 28 Gennaio 2026 | Aggiunti package gui e database alla sezione Metriche e qualità del codice |

# Modelling

Il diagramma Use Case si trova nel documento Requisiti, mentre Class, Package e Component si trovano in questo documento nella sezione Software Design Patterns.
Abbiamo utilizzato LucidChart e PlantUML, due tool online che permettono la creazione di diagrammi UML.
Le immagini dei diagrammi UML che vedremo nel documento si trovano nella cartella JustPoker/docs/UMLdiagrams della repository GitHUB.

## Final State Diagram

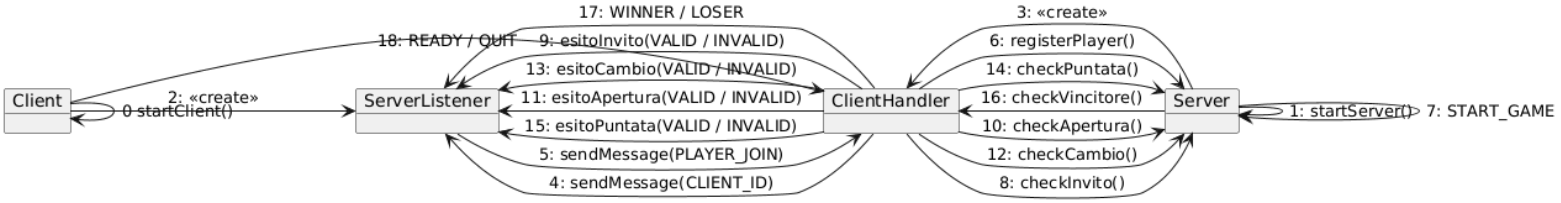
Il diagramma rappresenta gli stati in cui si può trovare un giocatore durante le diverse fasi di gioco, che variano in base alle proprie azioni o a quelle degli altri giocatori.




## Communication Diagram

Il diagramma rappresenta la sequenza di scambio di messaggi tra un client ed il server, considerando un flusso di gioco corretto.

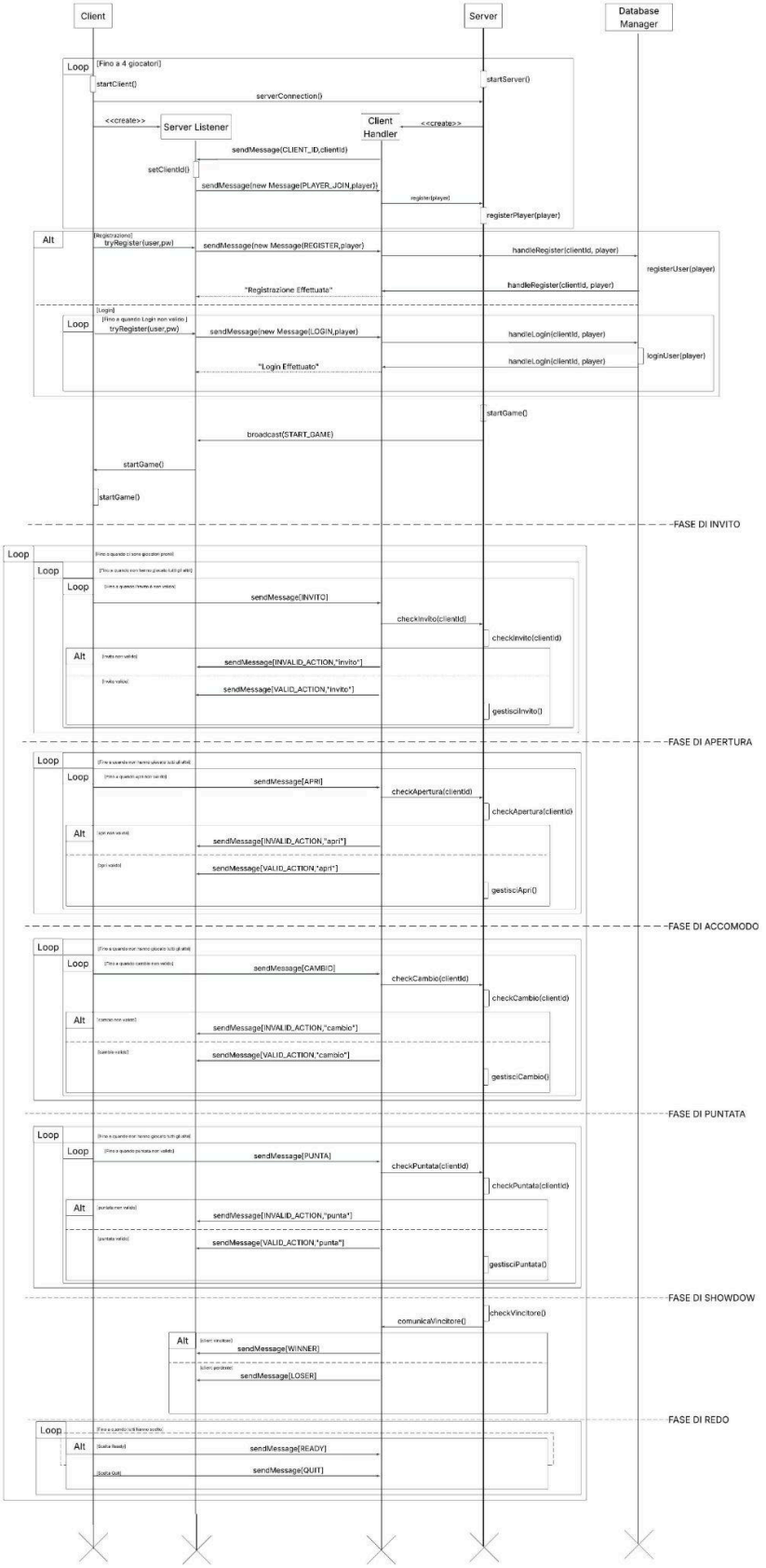
Version 1.1


Client
ServerListener
ClientHandler
Server
0 startClient()
2: «create»
18: READY / QUIT
17: WINNER / LOSER
9: esitoInvito(VALID / INVALID)
13: esitoCambio(VALID / INVALID)
11: esitoApertura(VALID / INVALID)
15: esitoPuntata(VALID / INVALID)
5: sendMessage(PLAYER_JOIN)
4: sendMessage(CLIENT_ID)
3: «create»
6: registerPlayer()
14: checkPuntata()
16: checkVincitore()
10: checkApertura()
12: checkCambio()
8: checkInvito()
1: startServer()
7: START_GAME

Version 1.1

# Sequence Diagram

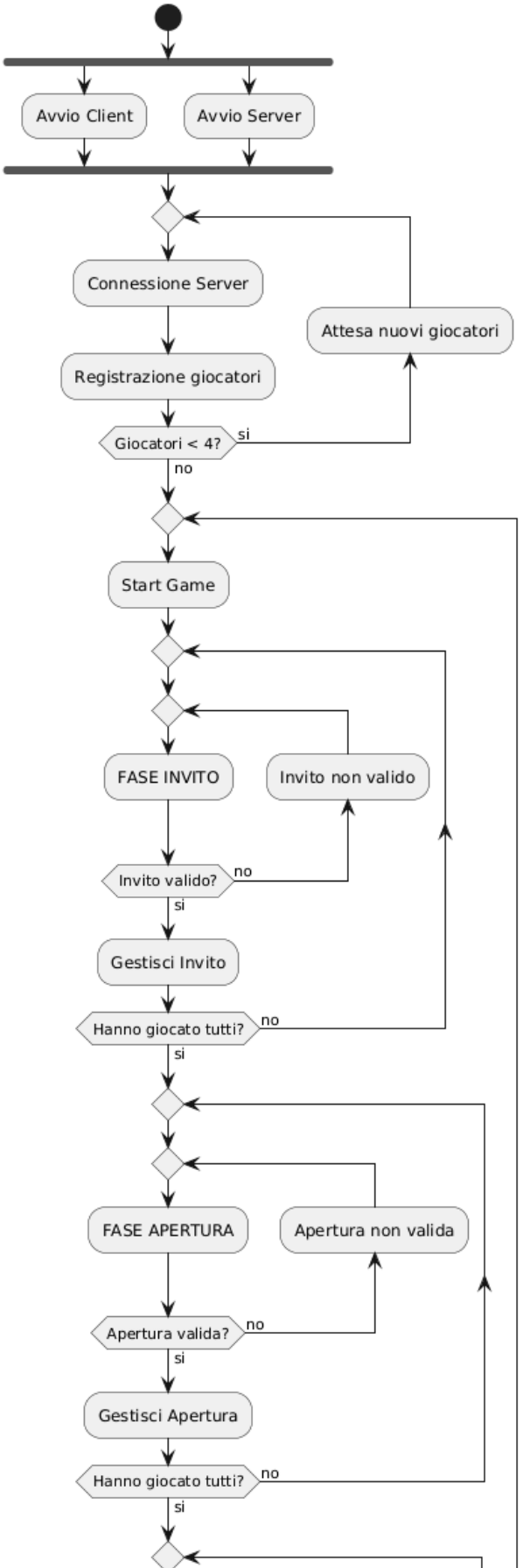
Il diagramma rappresenta la sequenza di scambio di messaggi tra un client ed il server dalla creazione di una connessione socket al flusso di gioco.



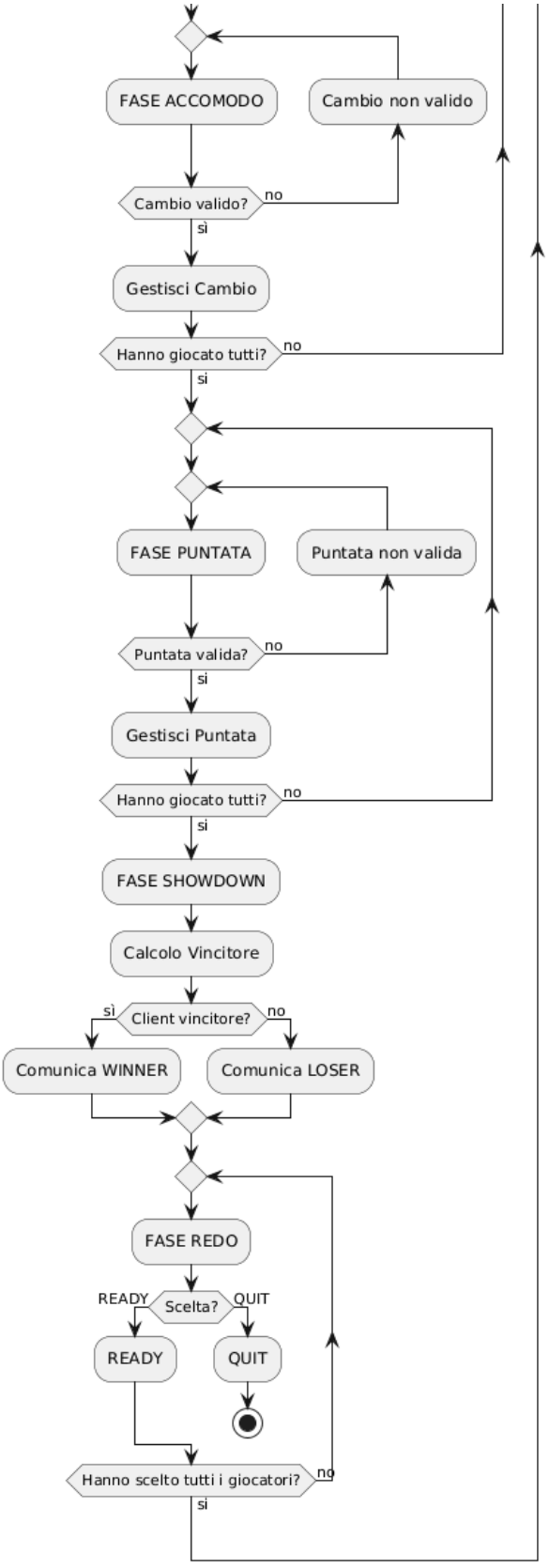
Version 1.1

# Activity Diagram

Il diagramma delle attività rappresenta il flusso delle operazioni svolte dai client e dal server, dalla fase di instaurazione della connessione socket fino al termine del gioco, includendo le diverse fasi di interazione tra i giocatori.



Version 1.1


FASE ACCOMODO
Cambio non valido
Cambio valido?
no
sì
Gestisci Cambio
Hanno giocato tutti?
no
si
FASE PUNTATA
Puntata non valida
Puntata valida?
no
si
Gestisci Puntata
Hanno giocato tutti?
no
si
FASE SHOWDOWN
Calcolo Vincitore
sì
Client vincitore?
no
Comunica WINNER
Comunica LOSER
FASE REDO
READY
Scelta?
QUIT
READY
QUIT
Hanno scelto tutti i giocatori?
no
si

Version 1.1

# Software architecture

## Viste architetturali

Descriviamo due punti di vista secondo Bass, di due classi diverse:

- Modulo: uso
    - il Client utilizza il ServerListener per poter gestire le risposte del server e aggiornare il proprio stato di Game
    - il Server utilizza diversi ClientHandler per poter gestire le richieste dei client e aggiornare il vero stato di Game
- Componenti e connettori: client/server
    - il sistema ha diversi client che comunicano tramite messaggi con un unico server

## Stile architetturale

Lo stile architetturale che segue il nostro progetto è Tipo di Dati Astratto, dove i due componenti comunicano tra loro tramite lo scambio di messaggi attraverso un canale prestabilito (socket):

- Componenti
    - Client: invia al server delle richieste per modificare lo stato del Game. Il client è gestito da un due thread, uno che invia le richieste al server e l'altro che gestisce le risposte ricevute
    - Server: componente gestore (manager) che controlla lo stato del Game memorizzando e modificandolo grazie allo scambio di messaggi (richieste) con il Client; è anche componente controllore poiché gestisce il flusso temporale dell'avanzamento di gioco. Il server è gestito da due thread, uno che invia le risposte al client e l'altro che gestisce lo stato del Game
- Connettori
    - Passaggio di messaggio: il client invia dunque delle chiamate di procedura al server (tramite messaggi) che si occuperà di controllarle ed eseguirle

## Librerie esterne con Maven

### JUnit

Abbiamo implementato JUnit come libreria esterna grazie a Maven. Questo ci ha permesso di specificare dei casi di test relativi al nostro dominio applicativo, in modo da poter testare e trovare bug nel codice: abbiamo infatti sfruttato le potenzialità di JUnit per analizzare i casi "limite" del nostro codice.

### Log4j

Abbiamo implementato Log4j come libreria esterna grazie a Maven. Questo ci ha permesso di specificare diversi livelli di conoscenza delle informazioni generate dalle diverse classi, poichè modificando il file log4j2.xml che si trova in poker/src/main/resources possiamo definire quali informazioni stampare a console.

Version 1.1

## H2

Abbiamo implementato H2 come libreria esterna grazie a Maven.  Questo ci ha permesso di implementare un database per poter registrare gli account della piattaforma JustPoker.

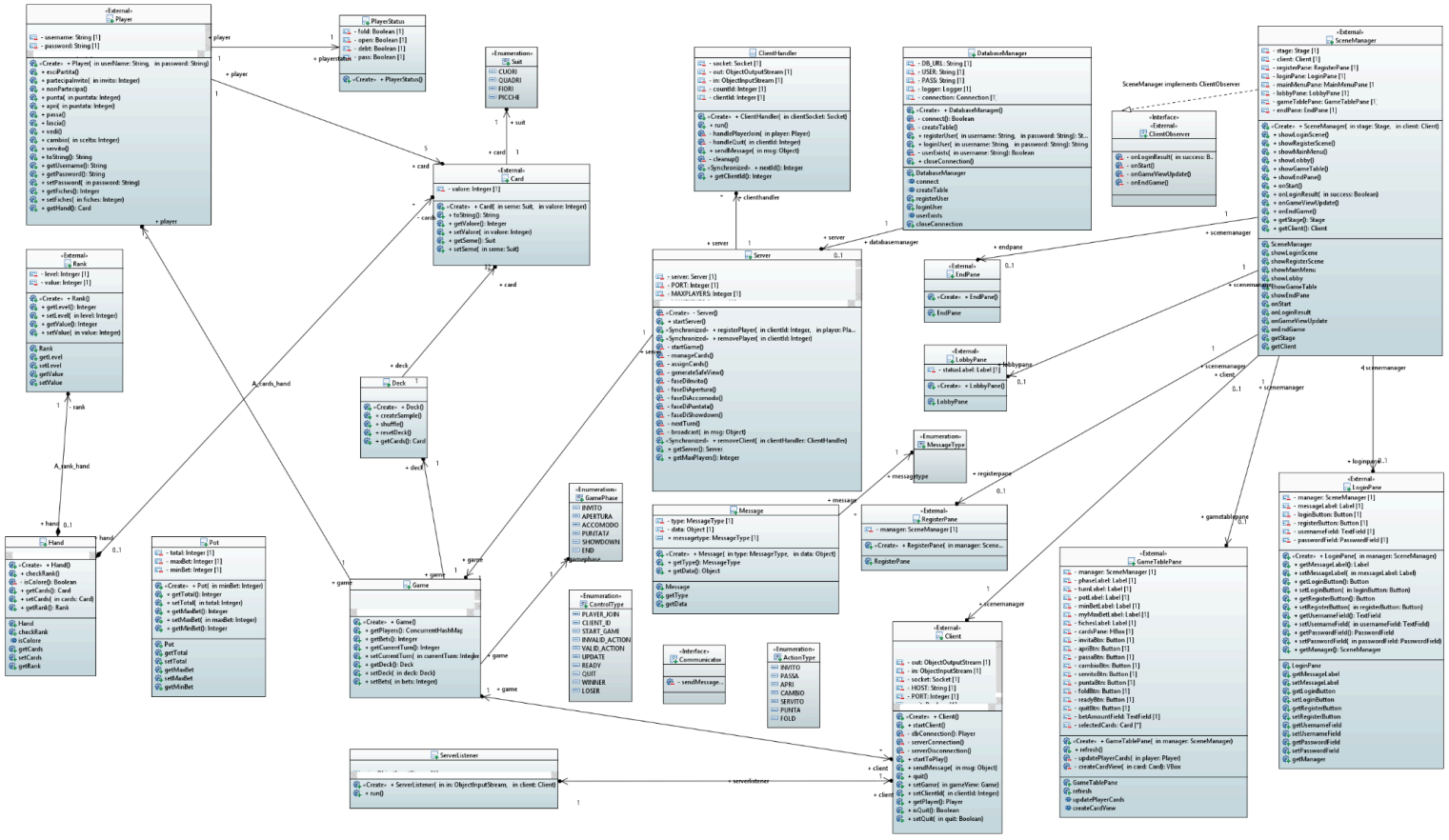
# Software design

## Design

Con il software design del sistema definiamo la struttura delle classi principali e come collaborano tra loro, con l'obiettivo di garantire chiarezza, manutenibilità ed estendibilità del codice. Il design è stato sviluppato a partire dal class diagram, dal quale è stata generata una prima versione del codice Java, modificata in seguito.
Le classi sono organizzate secondo una chiara separazione delle responsabilità, distinguendo le entità e la gestione dello stato del gioco.

### Class Diagram

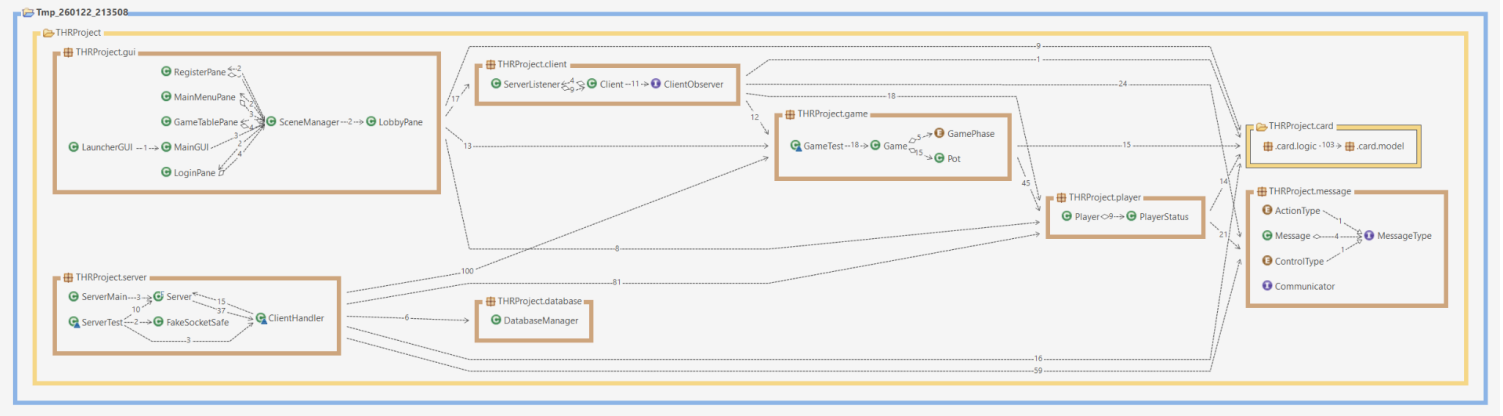
Dopo la modellazione del class diagram con Papyrus abbiamo generato e modificato il codice, implementando tutti i metodi necessari.
Abbiamo poi creato il diagramma completo dal codice tramite il tool Java Reverse di Papyrus: questo non genera però le associazioni, che sono state aggiunte a mano.



Version 1.1

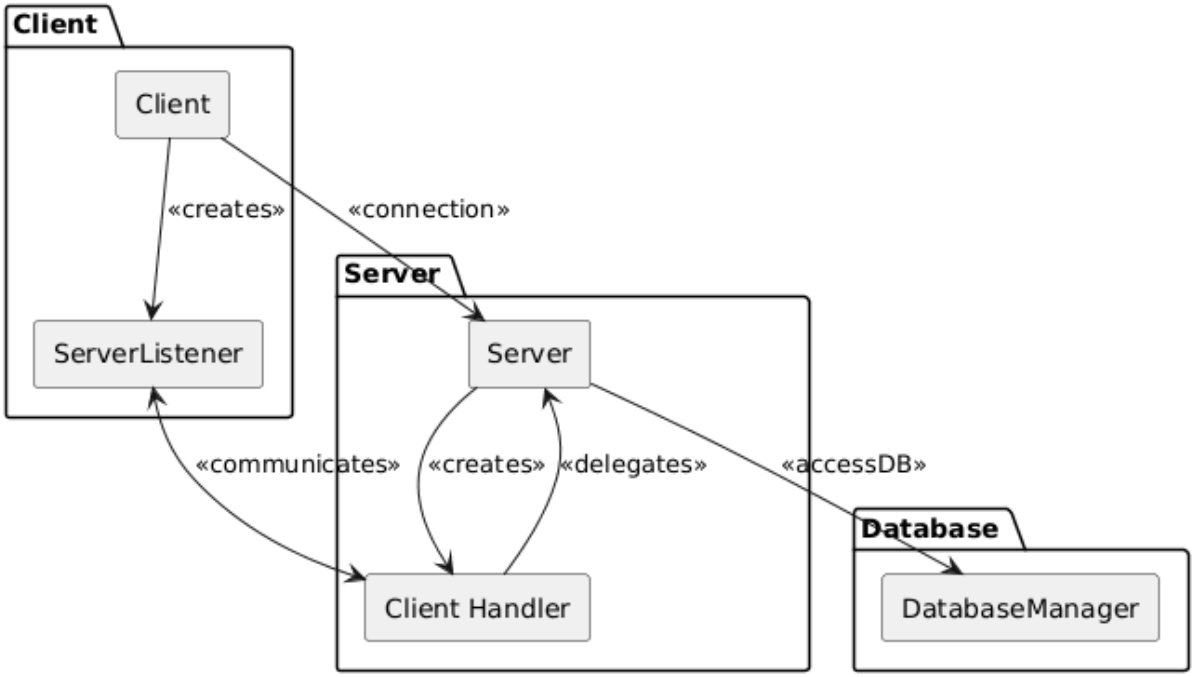
## Package Diagram

Il package diagram rappresenta quelli che sono i package e le relative classi del sistema e come interagiscono tra di loro.




## Component Diagram

Il component diagram rappresenta quelli che sono i componenti del sistema e come interagiscono tra di loro.



# Design Patterns

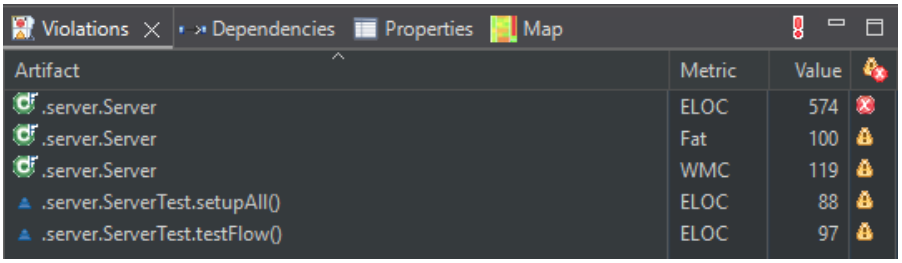
I pattern utilizzati nello sviluppo del codice sono:

- **Singleton**: abbiamo utilizzato questo pattern per il Server poiché ne esiste una sola istanza logica per poter garantire coordinamento centralizzato e coerenza dello stato globale. Il singleton impedisce la creazione di istanze multiple attraverso il suo costruttore privato: l'istanza è raggiungibile staticamente
- **Abstraction occurrence**: abbiamo utilizzato questo pattern in diversi punti
    - Player rappresenta l'astrazione del giocatore, l'entità stabile e persistente che identifica l'utente del sistema. Durante l'esecuzione del gioco, tuttavia, il giocatore può assumere stati differenti (associazione a PlayerStatus) e possedere mani di carte diverse (associazione ad Hand). Questa separazione consente di gestire correttamente l'evoluzione dello stato del giocatore nel tempo, migliorando la manutenibilità, la flessibilità del modello e la possibilità di estendere il sistema senza modificare l'astrazione principale
    - Game rappresenta l'astrazione dello stato del gioco, l'entità stabile e persistente. Durante l'esecuzione le carte che gestisce variano (associazione a Deck) e anche il piatto (associazione con Pot)
- **Observer pattern**: abbiamo utilizzato questo pattern poichè avevamo bisogno che la GUI potesse rispondere a degli stimoli del Client, che però arrivavano da uno scambio di messaggi con il server. In questo modo lo SceneManager diventa un osservatore del Client, l'osservato, che manda gli input (evitiamo così dipendenze cicliche tra package client e gui).

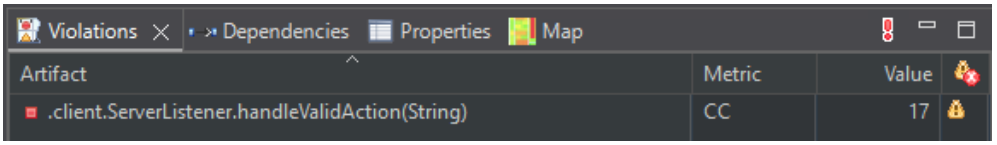
# Metriche e qualità del codice

Per il calcolo delle metriche riguardanti la qualità del codice utilizzeremo il plugin STAN:

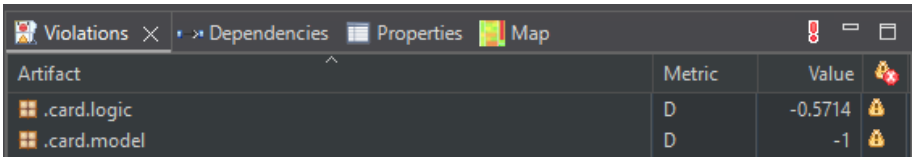
- Package server: i valori elevati di WMC (misura la complessità di una classe in base alla complessità ciclomatica dei suoi metodi) e FAT (misura il grado di funzionalità di una classe) nel package server sono coerenti con il suo ruolo di componente di coordinamento dell'architettura client/server poiché gestisce il flusso delle comunicazioni e l'orchestrazione delle operazioni. Le metriche riflettono quindi la nostra scelta architetturale e non un problema di qualità del codice



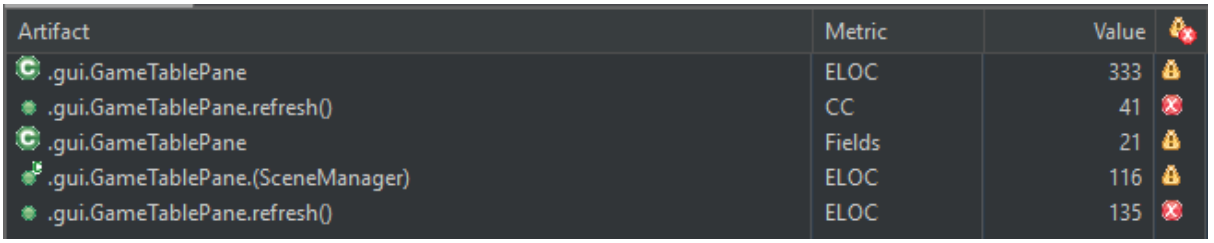

- Package client: presenta solo un valore elevato di CC nel metodo handleValidAction() che non può subire refactoring poiché spezzarlo romperebbe la sua coerenza concettuale (non ha dipendenze esterne ed ha basso ELOC)



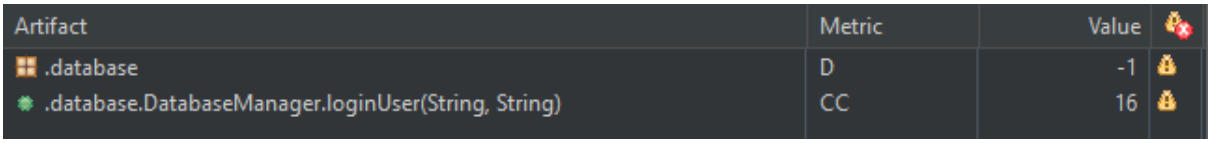
- Package card: ha subito refactoring ma comunque il suo package model è stabile (in quanto non presenta dipendenze in uscita ma viene utilizzato da numerosi altri package del sistema) e concreto (poiché non presenta classi astratte o interfacce). Tale scelta è motivata dal ruolo del package, che rappresenta il dominio centrale del gioco (carte, mazzo, mano), per il quale la stabilità è un requisito inconfutabile



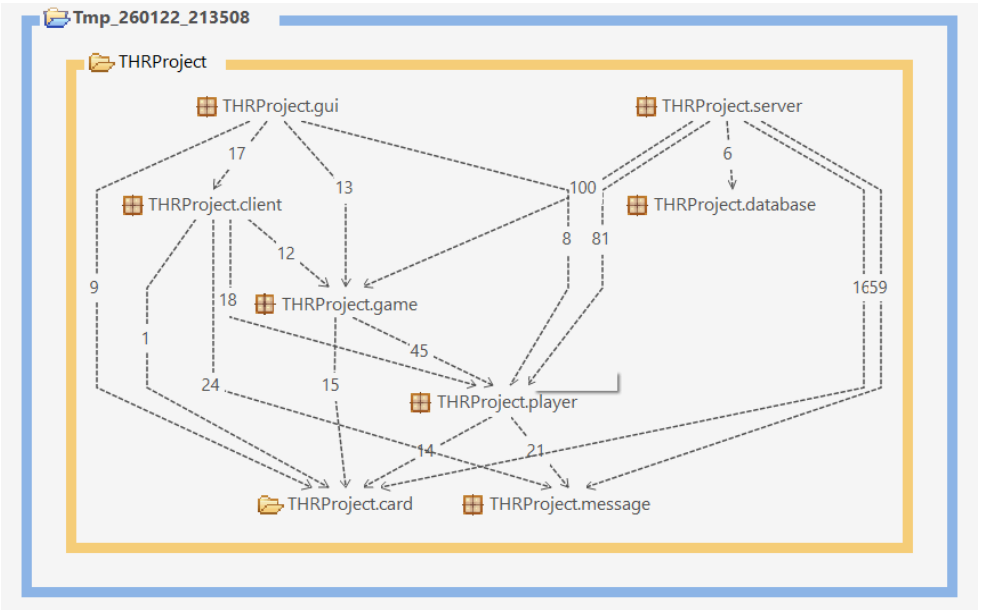

- Package gui: presenta dei valori di CC e di ELOC abbastanza alti, ma ciò è giustificato dal fatto che la maggior parte della grafica è gestita proprio dalla classe GameTablePane.




- Package database: presenta un valore di D anomalo, poiché il package contiene una sola classe, che è concreta, e non dipende da altri package. E' quindi molto stabile e concreto: questo rispecchia il suo ruolo, poiché rappresenta un componente di basso livello e centrale dell'architettura, pensato per fornire servizi agli altri package senza dipendere da essi.

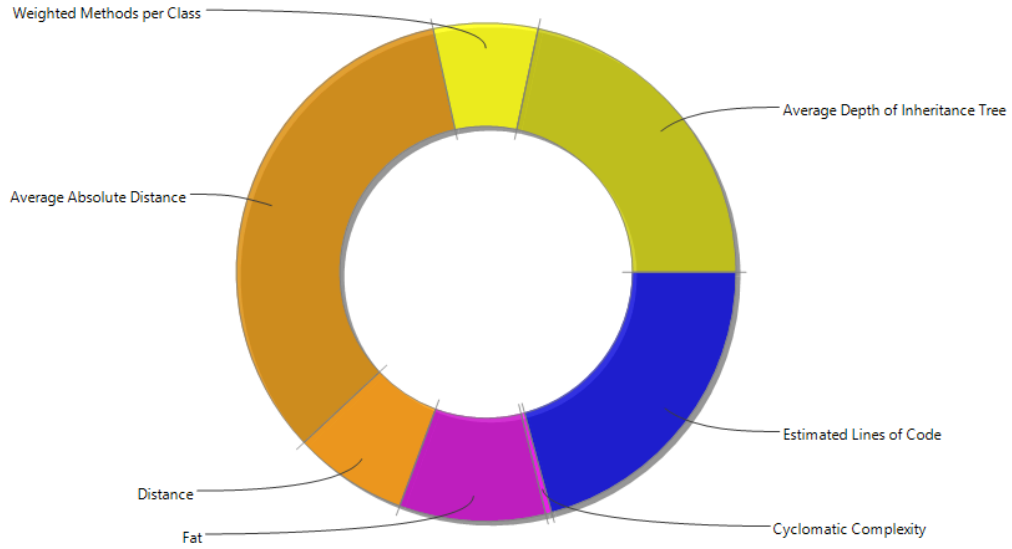



- Non sono presenti dipendenze cicliche tra i package




- Il grado di inquinamento è di 1.66, valore che indica un ottimo grado di qualità del codice, che può comunque essere ottimizzato. Questo valore indica infatti quanti

difetti ha (code smells) e quanto è complesso il codice. Questo grafico rappresenta come classi/package si discostano da quella che è la media dei valori




- Weighted Methods per Class (WMC): complessità totale dei metodi in una classe
- Average Depth of Inheritance Tree (DIT): profondità media della gerarchia di ereditarietà delle classi
- Estimated Lines of Code (ELOC) linee di codice stimate per classe o package
- Cyclomatic Complexity (CC): misura del numero di percorsi logici nel codice
- Feature Access To Class (FAT): numero di attributi/metodi utilizzati dalle altre classi (proporzionale all'accoppiamento)
- Distance (D): quanto un package è bilanciato tra astrazione e instabilità
- Average Absolute Distance: media assoluta delle distanze dei package

- Questo grafico rappresenta  invece la media dei valori: si vede infatti come i predominanti siano ELOC e FAT, ed è proprio il package Server a generare quei dati