ChessTime
Spezifikation der Softwareanforderungen
Version 1.0

Inhaltsverzeichnis

1 - Vorwort

1.1 - Ziel des Dokuments

Ziel dieses Dokuments ist es, eine Übersicht über die Architektur der ChessTime App bzw. des Servers zu verschaffen. Die Architektur wird aus mehreren Sichten und mithilfe von Diagrammen und Texten näher erläutert.

1.2 - Geltungsbereich

Android-App mit einer auf JServlets basierender Server-Komponente, die das Spielen von Schach online ermöglicht.

1.3 - Definitionen, Akronyme, und Abkürzungen

Siehe Glossar.

1.4 - Referenzen

- Glossar
- Vision
- Software Development Plan
- Risk List

2 - Architektur der ChessTime App

2.1 - Architekturtreiber

Im Folgenden werden Anforderungen und Einschränkungen für Client und Server, die die Architektur wesentlich beeinflussen können aufgelistet.

2.2 - Allgemeine Anforderungen

Sicherheit

Alle Nutzerdaten müssen vor unbefugtem Zugriff geschützt sein. Es werden daher nur für das Ausführen des Spiels relevante Daten und anonymisierte Fehlermeldungen an den Server übermittelt.

2.3 - Anforderungen an den Client

Performance

Die Performance-Anforderungen des Clients beziehen sich auf das Gerät *Google Nexus 5* mit der Android-Version 6.0.1. Die App soll im Durchschnitt mit weniger als 5 übersprungenen Frames pro Sekunde ausgeführt werden. Das Starten der App darf die Zeit von 10 Sekunden nicht überschreiten. Die Reaktionszeit auf Benutzereingaben muss unter 0,5 Sekunden liegen.

Plattform

Der Client wird als Android-App mit minSdkVersion 20 (entspricht Android 5.0) und targetSdkVersion 24 (entspricht Android 7.0) entwickelt. Geräte unter Api-Level 20 werden nicht unterstützt.

2.4 - Anforderungen an den Server

Plattform

Der Server wird als Java Servlet entwickelt.

Performance

Die Performance Anforderungen beziehen sich auf einen Server, der die Hardwareanforderungen erfüllt. Der Server muss bis zu 1000 Spieler gleichzeitig unterstützen. Dabei darf die Übertragung eines Spielzugs nicht 1,5 Sekunden unterschreiten.

Hardwareanforderungen

- Netzwerkverbindung möglich
- mind. 2 GB freier Festplattenspeicher
- mind. 2 GB Arbeitsspeicher

2.5 - Architektur Konzepte

Model-View-Presenter (MVP)

Die Benutzeroberfläche der App wird nach dem MVP Konzept realisiert. Das Ziel von MVP ist es, die Darstellung der GUI (View), Verhalten der GUI (Presenter) und die Daten der GUI (Model) zu trennen. Im Idealfall können dadurch Model, View oder Presenter ausgetauscht und verändert werden, ohne eine Änderung in den anderen Schichten vorzunehmen.

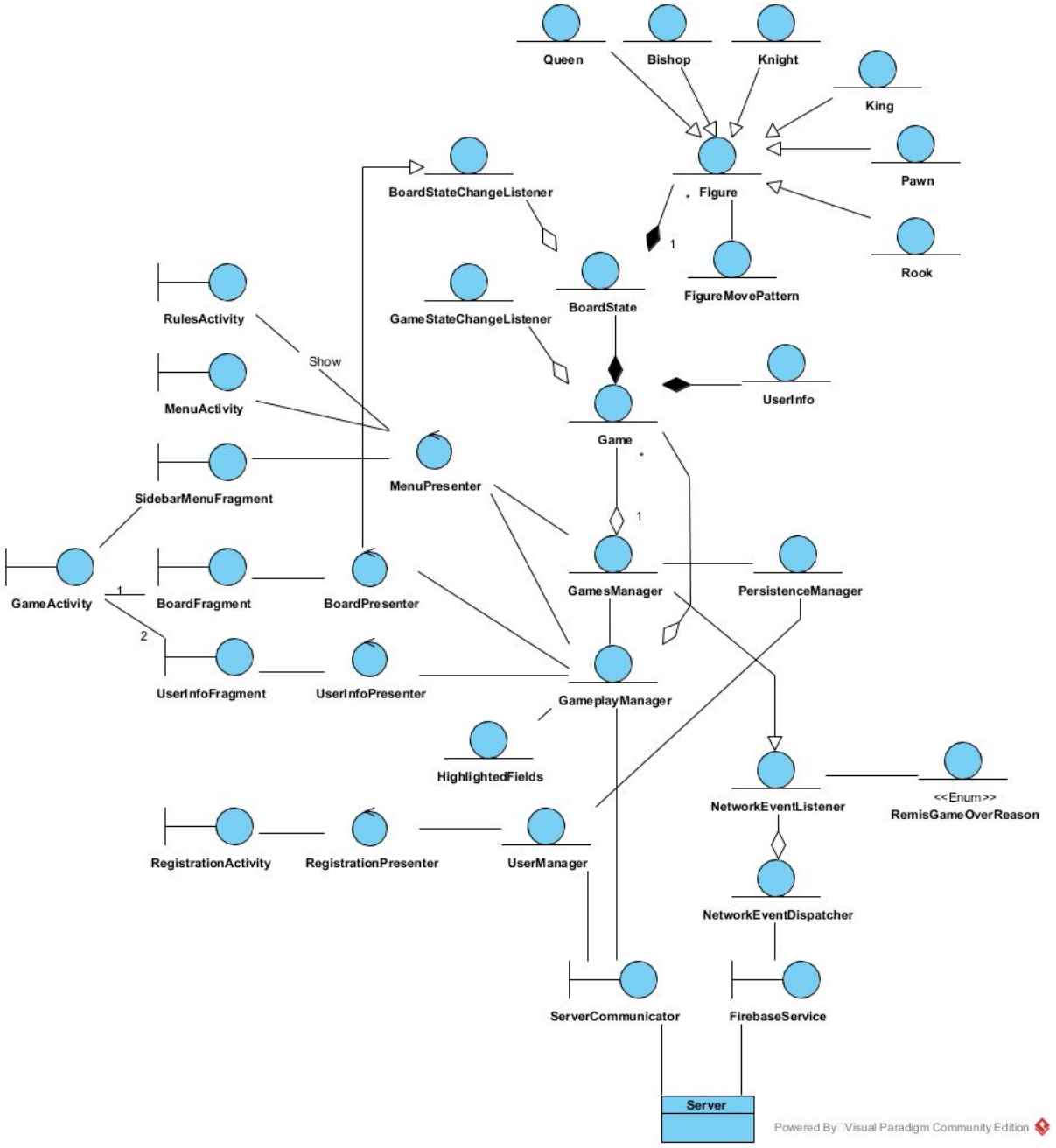
Listener (Observer)

Wird verwendet, um Klassen von Ereignissen zu benachrichtigen, ohne dabei ihre Details zu kennen.

3 - Systemarchitektur Client

3.1 - Klassensicht

3.1.1 - Klassendiagramm



3.1.2 - Beschreibung wichtigster Klassen

FirebaseService
Empfängt Nachrichten die vom Server über Firebase gesendet werden. Wenn eine Nachricht empfangen wird, wird die entsprechende Methode im Listener aufgerufen, die diese dann an die entsprechenden Klassen weiterleiten tut.

ServerCommunicator
Wird von anderen Klassen genutzt, um Aktionen (z.B. Spieler bewegt eine Figur) an den Server zu übermitteln oder Informationen abzurufen (z.B. Elo-Zahl).

GamesManager
Enthält die GameInfos der laufenden Spiele, enthält Create-, Read-, Update-, Delete-Operationen. Reagiert auf Network Events und passt das Datenmodell entsprechend an.

GameplayManager
Enthält alle benötigten Methoden, um ein Spiel zu spielen und leitet die für den Server relevanten Methodenaufrufe an den RestCommunicator weiter. Bezieht sich immer auf das aktuell geöffnete Spiel.

UserManager
Speichert und lädt das Passwort des Benutzers und benachrichtigt den RestCommunicator, wenn der Spieler sich registrieren möchte.

GameActivity
Dient als Container für alle notwendigen Fragmente, um das Spiel zu spielen und die App von einem Spiel aus zu navigieren.

BoardFragment / Presenter
Sorgt für die Darstellung und Veränderung des Spielfelds.

UserInfoFragment / Presenter
Stellt die Informationen eines Spielers in Bezug auf ein Spiel dar. In der GameActivity sind zwei UserInfoFragments enthalten. Eines um die Informationen des Spielers darzustellen und ihm durch einen Button die Möglichkeit zu bieten, Remis zu beantragen und zum Anderen, um die Informationen des Gegenspielers anzuzeigen, einschließlich ob er ein Remis beantragt hat.

MenuActivity
Diese Activity stellt die Benutzerschnittstelle zu den Menüfunktionen dar. Sie wird beim Starten der App als Hauptmenü angezeigt. Die Menüfunktionen sind:

- *Spielregeln anzeigen*
- *Spiel suchen*
- Eine Lister aller laufenden Spiele, die mit einem Tap geöffnet werden können

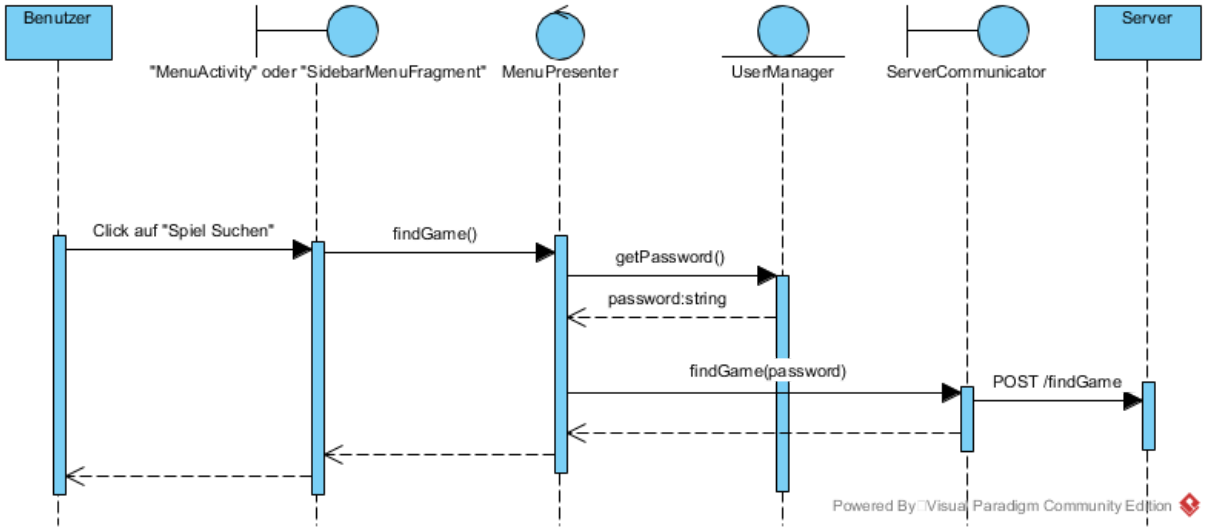
SidebarMenuFragment
Dieses Fragment stellt ebenfalls die Benutzerschnittstelle zu den Menüfunktionen dar. Sie wird jedoch in der Sidebar der GameActivity angezeigt.

MenuPresenter
Presenter für *MenuActivity* und *SidebarMenuFragment*.

3.2 - Anwendungsfall Realisierungen

3.2.1 - Gegenspieler suchen

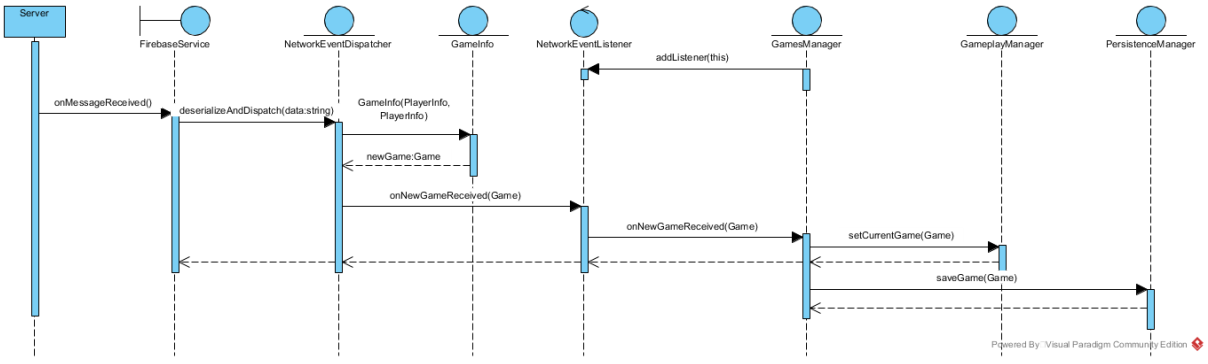
3.2.1.1 - Request an Server senden



Beschreibung

Dieser Anwendungsfall beginnt, wenn der Benutzer in einem der Menüs auf Gegenspieler suchen tippt, wodurch der *MenuPresenter* benachrichtigt wird, der das Passwort vom *UserManager* abruft. Danach teilt er dem *ServerCommunicator* mit, dass er eine Gegenspieler-Suchanfrage an den Server stellen möchte, der diese dann an den Server weiterleitet.

3.2.1.2 - Response von Server verarbeiten

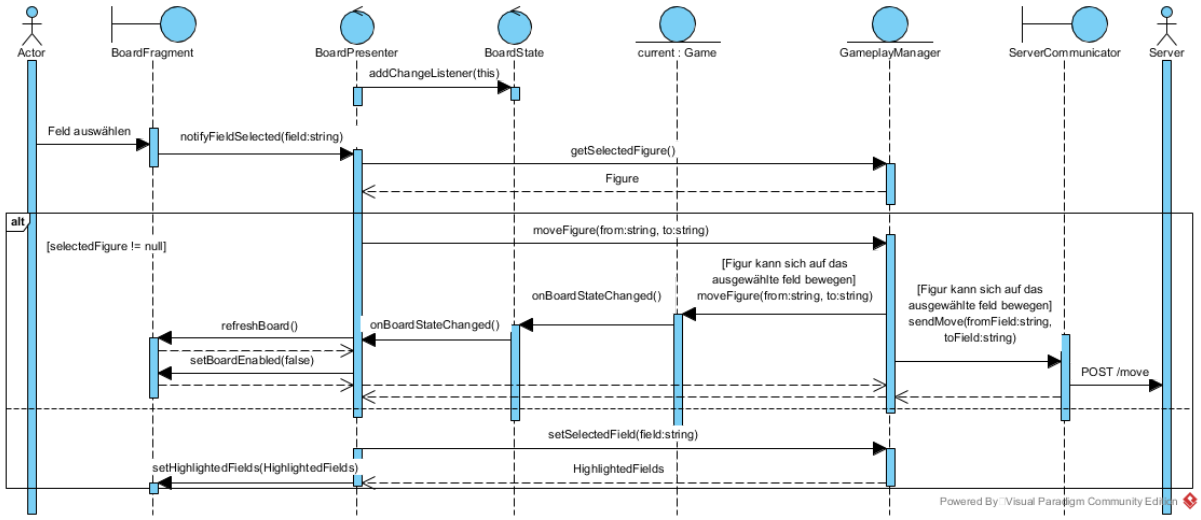


Beschreibung

Wenn der Server einen Gegenspieler findet, versendet er eine Firebase-Message mit einem Json-Objekt an die App. Diese wird von der *FirebaseService* Klasse empfangen. Der *NetworkEventDispatcher* erstellt aus den Informationen des Json-Strings ein Objekt der Klasse *Game* und benachrichtigt den NetworkEventListener über das neu empfangene Spiel. Über einen Listener bekommt der *GameManager* das *Game* Objekt, der dieses dann zu der Liste der laufenden Spiele hinzufügt und es als das aktuell gespielte Spiel setzt.

3.2.2 - Figur auswählen / Figur bewegen

Sequenzdiagramm



Beschreibung
Wenn der Benutzer auf ein Feld des Schachbretts tippt, wird dies vom *BoardFragment* an den *BoardPresenter* weitergeleitet.
Als Nächstes wird die aktuell ausgewählte Figur vom *GameplayManager* abgefragt. Wenn eine Figur ausgewählt ist, wird sie auf das gewählte Feld bewegt. Sollte keine Figur ausgewählt sein, wird die Figur auf dem gewählten Feld ausgewählt.

Figur bewegen
Wenn das ausgewählte Feld ein Feld ist, auf dieses sich die Figur laut Regeln bewegen kann, ändert der *GameplayManager* die Position der Figur im Datenmodell (Game Objekt) und dieses benachrichtigt dann über seinen Listener den *BoardPresenter*, der die View aktualisiert.
Der Zug wird an den Server gesendet.

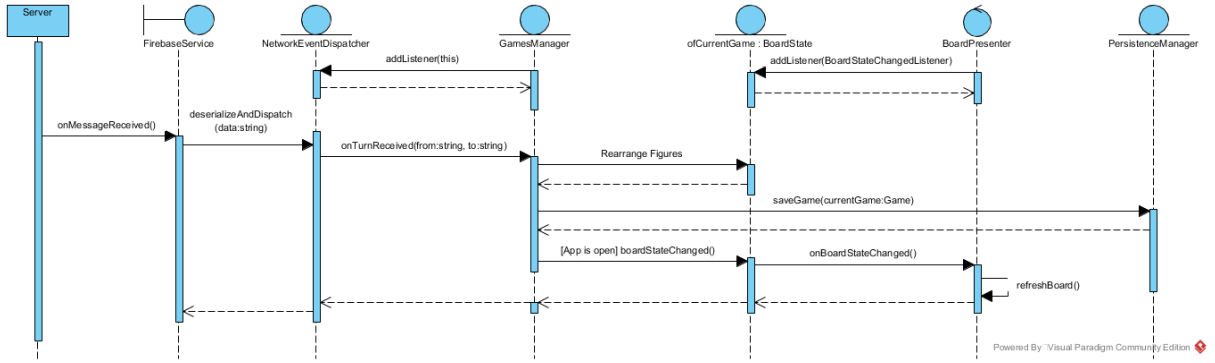
Figur auswählen
Das ausgewählte Feld wird im *GameplayManager* gespeichert.
Als Nächstes fordert der *BoardPresenter* vom *GameplayManager* alle Felder an, die für das vom Benutzer gewählte Feld hervorgehoben werden. Hervorgehobene Felder können sein:

- Felder auf die sich die Figur bewegen kann
- Felder auf die sich die Figur bewegen kann und damit eine gegnerische Figur schlagen kann
- Felder auf die sich die Figur bewegen könnte, die aber durch eigene Figuren blockiert sind
- Das Feld der ausgewählten Figur, um es hervorzuheben

Der *BoardPresenter* leitet die hervorgehobenen Felder an das *BoardFragment* weiter, welches diese dann darstellt.

3.2.3 - Spielzug empfangen

Sequenzdiagramm

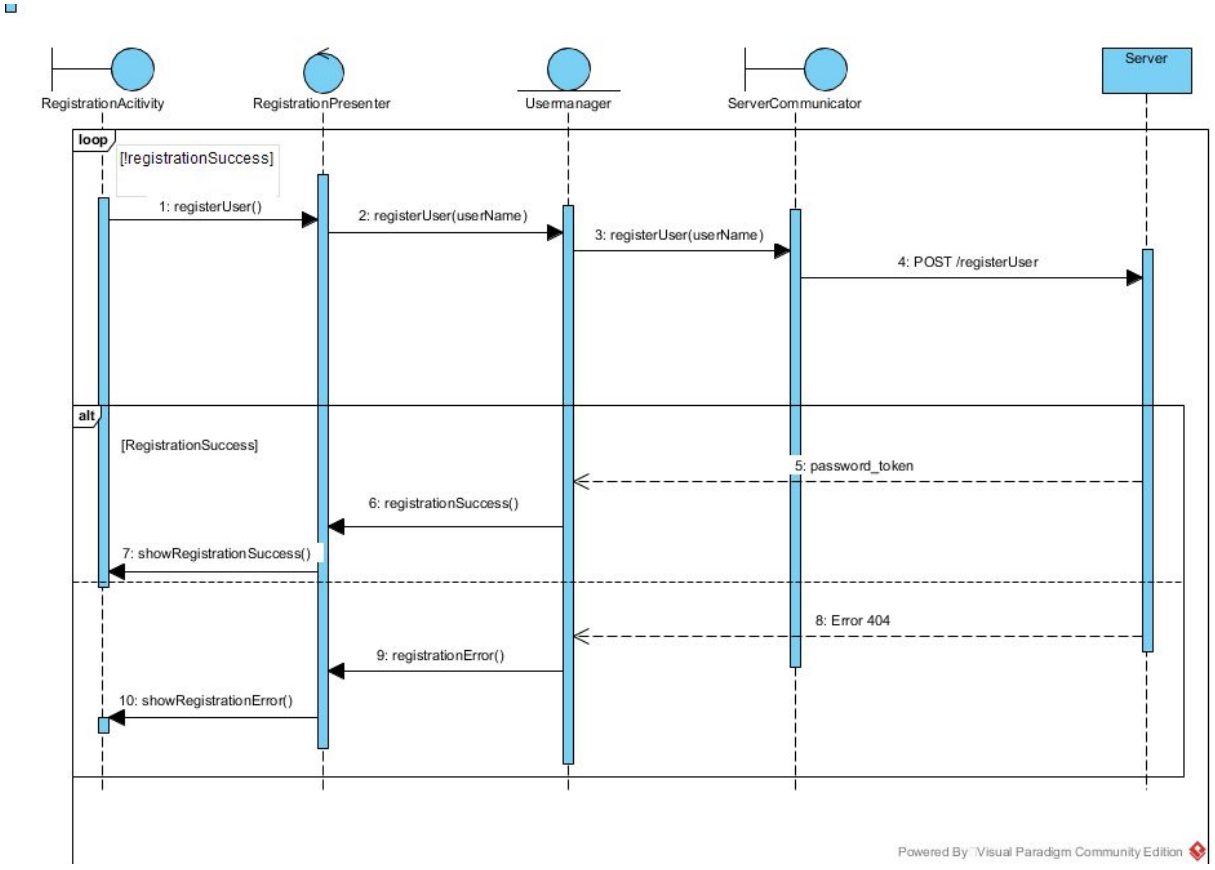


Beschreibung

Eine Firebase-Nachricht wird empfangen und an den NetworkEventDispatcher übergeben, der diese deserialisieren tut und seinen Listener aufruft, das Event an interessierte Klassen weiterzuleiten. Der GamesManager gehört zu einer dieser Klassen. Zunächst passt er das Datenmodell im Speicher an und leitet dann das aktuelle Spiel an den PersistenceManager weiter, der dieses auf den lokalen Speicher speichert.
Der GamesManager benachrichtigt im BoardState des aktuellen Spiels, dass das Board geändert wurde, welches er über seinen BoardStateChangeListener an den BoardPresenter weiterleitet, der dann die Darstellung des Boards aktualisiert.

3.2.4 - Benutzer registrieren / anmelden

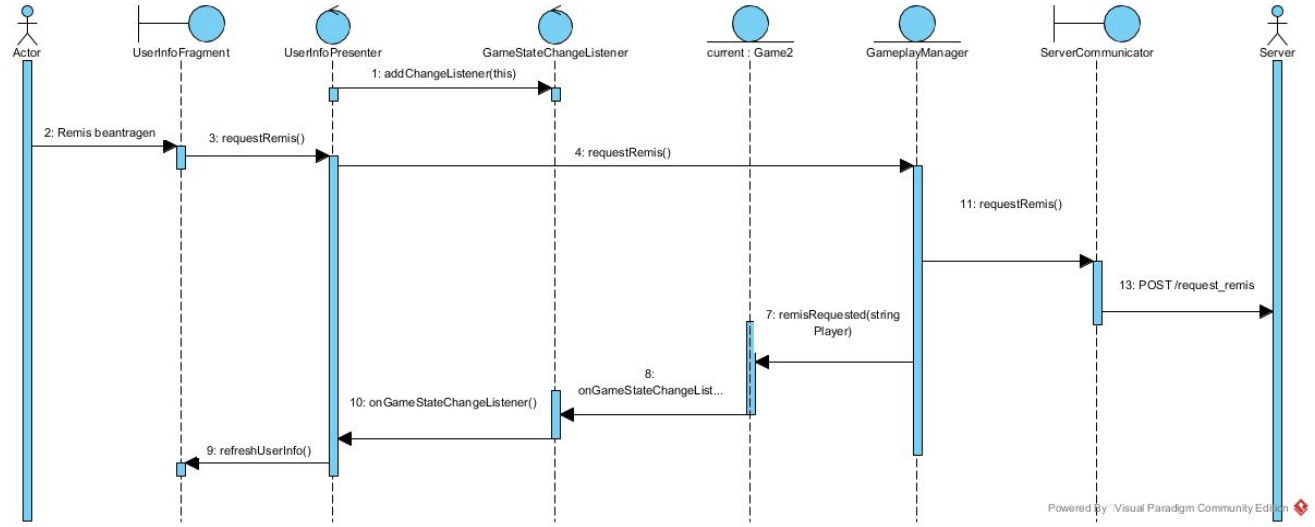
Sequenzdiagramm



Beschreibung
Wird die App vom Nutzer das erste Mal gestartet, so wird er von der *RegistrationActivity* aufgefordert einen Benutzernamen einzugeben.
Dieser wird vom *RegistrationPresenter* an den *Usermanager* weitergeleitet.
Der *Usermanager* ruft den *ServerCommunicator* auf, der den Benutzernamen an der Server versendet. Wenn der Name noch nicht vergeben ist, sendet der Server ein password_token über den *ServerCommunicator* an den *Usermanager* zurück.
Ist der Name bereits vergeben, sendet der Server einen 404 Error zurück. Der Usermanager benachrichtigt in beiden Fällen den *RegistrationPresenter,* um eine entsprechende Erfolgs- oder Fehlermeldung durch die *RegistrationActivity* anzuzeigen.
Dieser Vorgang wird solange wiederholt, bis ein gültiger Name gefunden wird.

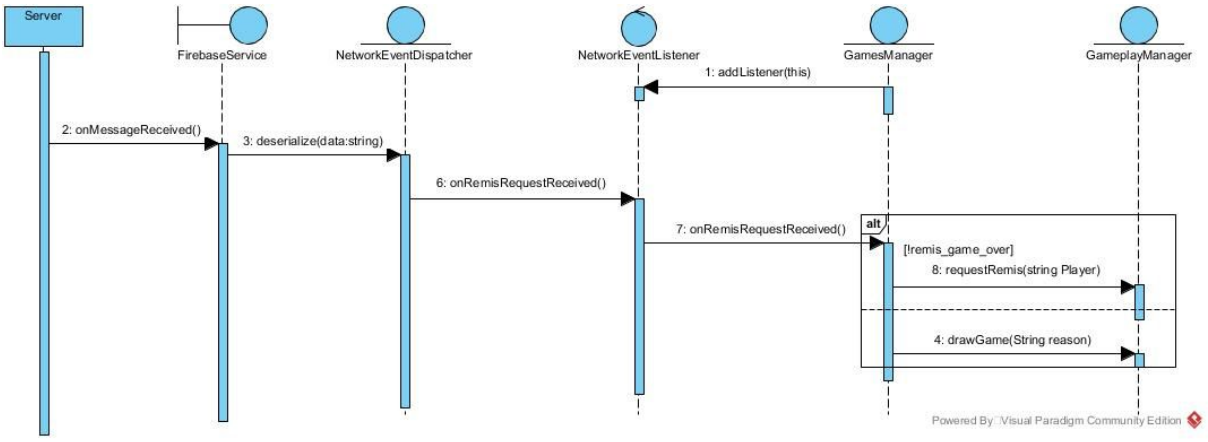
3.2.5 - Remis beantragen

3.2.5.1 - Request an den Server senden



Beschreibung
Wenn ein Spieler Remis beantragen möchte, tippt er auf einen Button im *UserInfoFragment*, welches den *UserInfoPresenter* informiert. Als Nächstes versendet dieser eine Remisanfrage an den *GameplayManager*. Über den *ServerCommunicator* wird eine POST Anfrage an den Server versendet. Zusätzlich wird eine Remisanfrage an das aktuelle Spiel versendet, welches den *UserInfoPresenter* mithilfe eines *GameStateChangeListener* im *UserInforfragment* aktualisiert, um die Remisanfrage anzuzeigen.

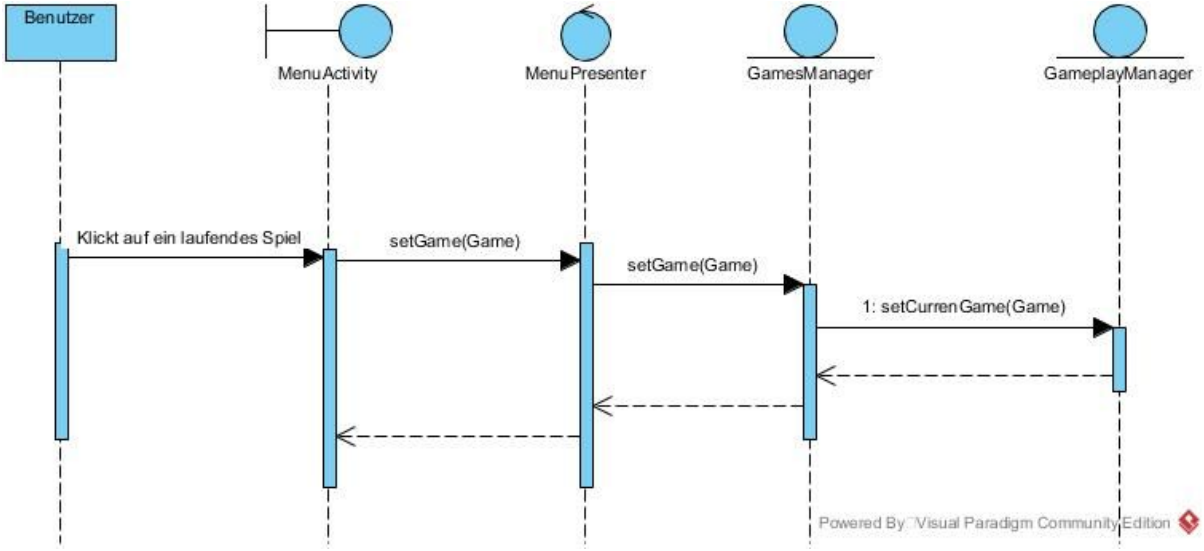
3.2.5.2 - Response von Server verarbeiten



Beschreibung
Wenn der Server eine Remisanfrage empfängt, versendet er eine Firebase-Message mit einem Json-Objekt an den Gegenspieler des Anfrager. Diese wird von der *FirebaseService* Klasse empfangen. Der *NetworkEventMessenger* erstellt aus den Informationen des Json-Strings, eine Nachricht an den GameManager. Über einen Listener bekommt der GameManager die Remisanfrage und überprüft, ob die Anfrage ein remis_game_over String enthält. Ist dies der Fall, endet das Spiel unentschieden. Im Falle einer Remisanfrage vom Gegner, ohne selbst Remis beantragt zu haben, wird diese an das *UserInfoFragment* versendet.

3.2.6 - Spiel fortsetzen

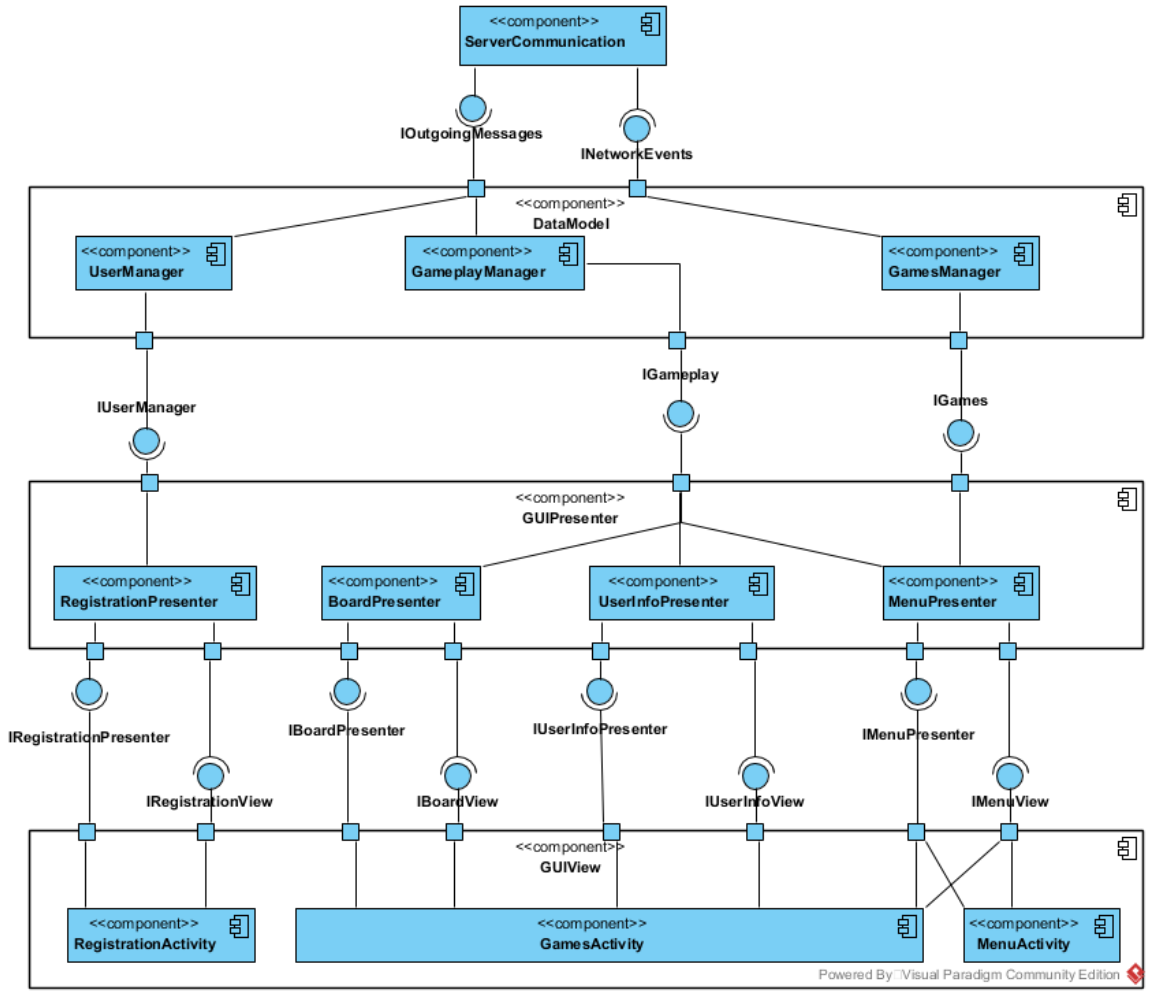
Sequenzdiagramm



Möchte der Spieler ein laufendes Spiel auswählen, so wählt er dieses in der *MenuActivity* aus. Die *MenuActivity* ruft den MeuPresenter auf, welcher die Nachricht an den *GamesManager* weiterleiten tut. Als Nächstes setzt er das Game-Objekt als aktuelles Spiel im *GameplayManager*.

3.3 - Komponentensicht

3.3.1 - Komponentendiagramm



3.3.2 - Beschreibung

Die App besteht aus vier Oberkomponenten, den drei Schichten der MVP Architektur und einer Weiteren zur Kommunikation mit dem Server.

GUIView

Enthält die Views, die passiven Benutzeroberflächen. Jede View implementiert ihr View Interface um ihre Anzeigen von Außen durch den Presenter verändern zu lassen. Jede View benötigt eine Implementierung ihres zugehörigen Presenter Interfaces, um Nutzer-Events an ihn weiterzuleiten.

GUIPresenter

Stellt die Verbindung zwischen dem DataModel und den Views dar. Jeder Presenter implementiert sein Presenter Interface und benötigt eine Implementierung des zugehörigen

View Interfaces. Weiterhin werden Interfaces zur Kommunikation mit dem DataModel benötigt.

DataModel
UserManager
Implementiert Registrierung des Benutzerkontos und Speicherung des Passworts.

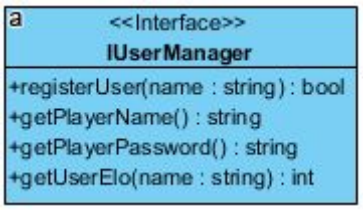
GamplayManager
Implementiert alle benötigten Methoden, um ein Spiel zu spielen und leitet die für den Server relevanten Methodenaufrufe über *IOutgoingMessages* an die *ServerCommunication* Komponente weiter.

GamesManager
Implementiert *IGames*. Bietet CRUD Operationen auf die laufenden Spiele.

ServerCommunication
IOutgoingMessages bietet die Möglichkeit Nachrichten an den Server zu versenden und ihren Rückgabewert auszulesen.
INetworkEvents Methoden werden aufgerufen, wenn eine asynchrone Nachricht vom Server empfangen wird.

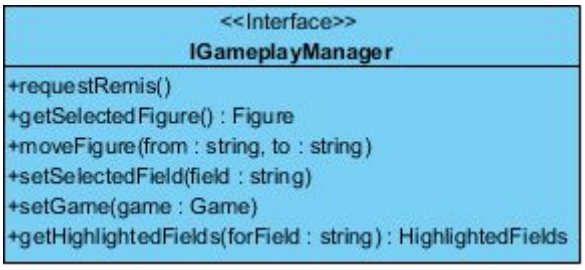
3.4 - Beschreibung wichtiger Interfaces und Klassen

3.4.1 - IUserManager



getPlayerName und *getPlayerPassword* beziehen ihre Rückgabewerte aus dem lokalen Speicher (PersistenceManager).
registerUser und *getUserElo* kommunizieren mit dem Server, um ihre Aufgaben zu erfüllen.
getUserElo gibt auf Angabe des Namens, den Elo-Wert eines Spielers zurück.

3.4.2 - IGameplayManager

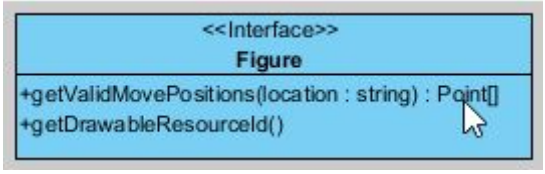


Funktionen beziehen sich immer auf das aktuell geöffnete Spiel.
requestRemis fordert beim Server Remis an.
getSelectedFigure gibt die ausgewählte Figur zurück. Null, wenn sich auf dem Feld keine befindet oder sie eine Gegnerische ist.
moveFigure bewegt eine Figur auf einem Feld auf ein Anderes. Wenn sich dort eine gegnerische Figur befindet, wird diese vom Feld entfernt.
setGame setzt das Game-Objekt auf, das sich die Funktionen beziehen.
getHighlightedFields gibt zurück, welche Felder für das ausgewählte Feld hervorgehoben werden sollen.

3.4.3 - NetworkEventListener

Klasse A hat eine Variable (oder Array) vom Typ *NetworkEventListener*.
Klasse B implementieren den *NetworkEventListener* und geben seine Referenz in die Variable der Klasse A. Wenn eines der im Listener bestimmten Events in Klasse A auftritt, ruft sie die entsprechenden Funktionen in ihre *NetworkEventListener*-Variable auf und benachrichtigt somit Klasse B, ohne eine direkte Referenz zu ihr haben zu müssen.
onRemisRequestReceived wird aufgerufen, wenn der Gegenspieler Remis beantragt hat. gameid sagt aus, um welches Game es sich handelt.

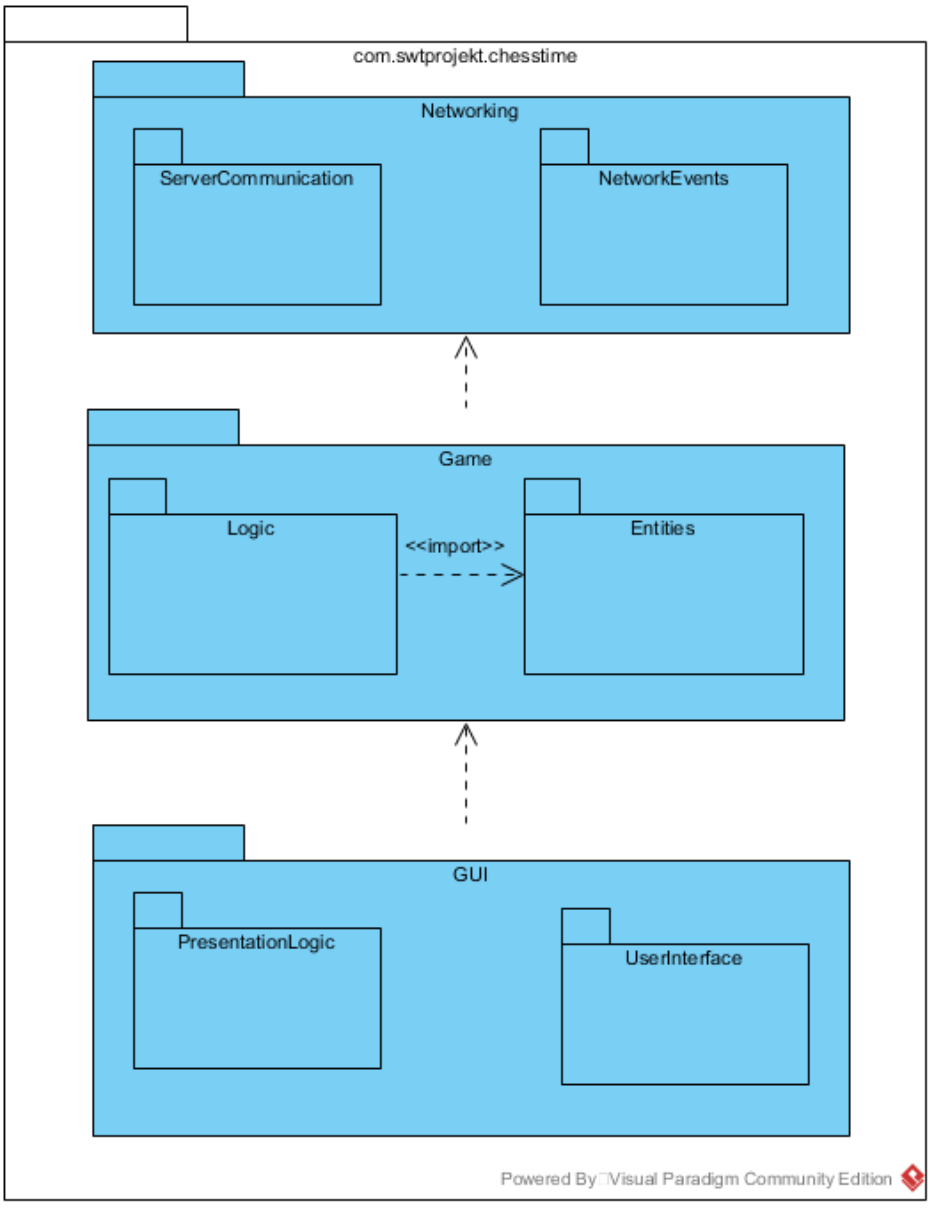
3.4.4 - Figure



figureDrawableReference enthält die Referenz ID für die Grafik, die für diese Figur angezeigt werden soll.
getValidMovePositions ist abstract und muss von jeder Figur-Subklasse (König, Bauer etc.) implementiert werden. Sie gibt im Bezug auf eine Position des Schachfeldes alle Positionen, auf die sie sich laut Spielregeln bewegen kann. Der Zustand des Schachfeldes (andere Figuren) werden dabei nicht in Betracht gezogen.

3.5 - Paketsicht

3.5.1 - Paketdiagramm



3.5.2 - Beschreibung

Das Game Package stellt die Buisiness-Layer des Systems dar. Hier wird das Spiel allein durch Daten repräsentiert. Das Unterpaket Logic enthält die Spiellogik und das Entities Package enthält die Benötigten Daten-Klassen.

GUI sorgt für der Darstellung der Daten aus dem Game Package für den Benutzer.

Das Networking Package ist für die gesamte Netzwerkkommunikation verantwortlich. ServerCommunication sorgt für ausgehenden Verkehr und NetworkEvents für das Empfangen asynchroner Nachrichten des Servers.

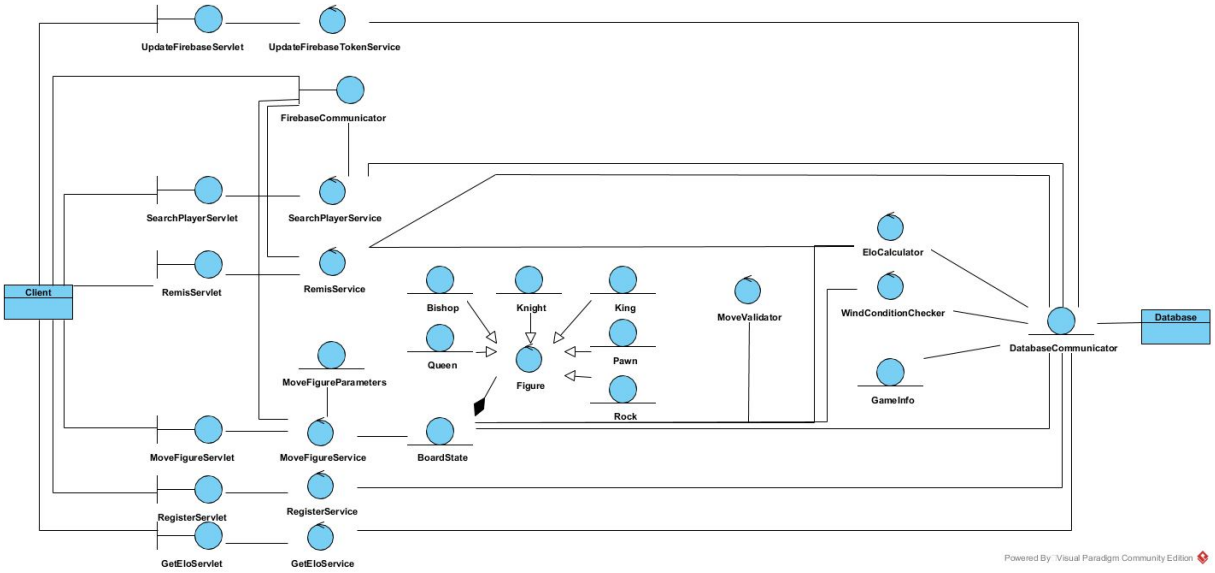
4 - Systemarchitektur Server

4.1 - Architekturkonzepte

Servlet
Ein Servlet ist eine Klasse, die auf HTTP-Requests auf einen (oder mehreren) Pfaden eines Servers reagiert.

4.2 - Klassensicht

4.2.1 - Klassendiagramm



4.2.2 - Beschreibung wichtigster Klassen

Servlets
Nehmen HTTP Requests entgegen, wandeln die Parameter in Java Klassen oder Datentypen um und leiten sie an den entsprechenden Service weiter, der sie verarbeitet.

FirebaseCommunicator
Enthält Methoden, um die Spieler asynchron über Events zu Informieren.

SearchPlayerService
Nimmt Spiel-Suchanfragen entgegen und benachrichtigt beide beteiligten Spieler über Firebase, wenn ein Spiel gefunden wird.

RemisService
Nimmt Post Remis-Anforderungen an. Bei der ersten Remis-Anforderung, wird der Gegenspieler darüber informiert. Wenn beide Spieler Remis beantragt haben, wird das Spiel beendet.

MoveFigureService

Nimmt den Spielzug eines Spielers entgegen, validiert ihn und leitet ihn an den Gegenspieler weiter.

EloCalculator
Passt die Elo-Zahl der Spieler nach Spielende an.

GetEloService
Gibt die Elo-Zahl eines Spielers, dessen Name als Parameter übergeben wurde, bei Aufruf zurück.

RegisterService
Empfängt einen gewünschten Benutzernamen und gibt, falls noch nicht vorhanden, das erzeugte Password-Token zurück.

UpdateFirebaseTokenService
Wird von Clients aufgerufen, um ihr neues FirebaseToken mitzuteilen, da Google dieses gelegentlich ändert.

BoardState
Repräsentiert den Zustand eines Spielfeldes und bietet Methoden, um diesen zu verändern.

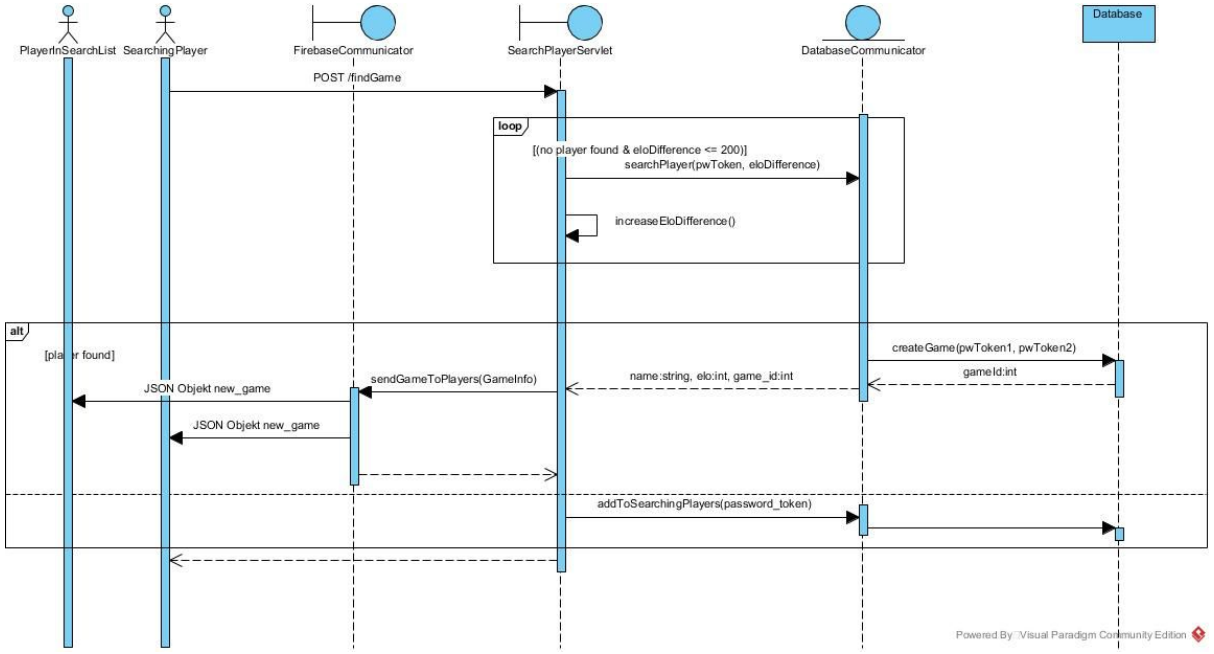
WinConditionChecker
Überprüft bei Angabe eines BoardStates, ob das Spiel gewonnen ist bzw. von wem.

MoveValidator
Überprüft bei Angabe eines Spielzugs und BoardStates, ob dieser möglich ist.

4.3 - Anwendungsfall / Realisierungen

4.3.1 - Gegenspieler suchen

Sequenzdiagramm

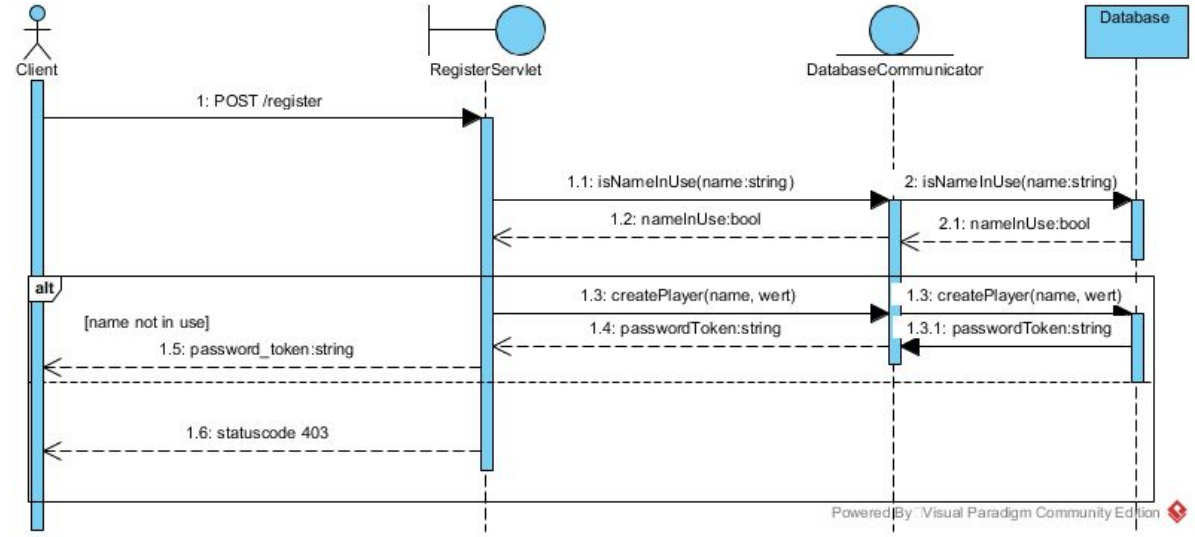


Beschreibung
Client schickt einen Post-Request an den Pfad *serverurl*/findGame. Dieser wird vom SearchPlayerServlet empfangen. Es wird versucht, einen anderen Spieler mit einer ähnlichen Elo-Zahl, in der Liste von suchenden Spielern als Gegenspieler zu wählen.
Sollte kein Gegenspieler gefunden werden, wird die maximale Elo-Differenz um 20 erhöht und nochmal versucht, bis eine Differenz von 200 erreicht wurde. Sollte dann immernoch kein Gegenspieler gefunden sein, wird der Spieler zu der Liste der suchenden Spieler hinzugefügt.
Sobald ein Gegenspieler gefunden wurde, wird in der *Database* ein Spiel angelegt und eine GameId generiert. Die Spielinformationen werden dann als Json String an beide Spieler über Firebase versendet.
Das Json Objekt enthält folgende Schlüssel:

- “kind” - Enthält den String “new_game”
- “opponent_name”
- “opponent_elo”
- “game_id”

4.3.2 - Spieler anlegen

Sequenzdiagramm



Beschreibung
Die Anfrage des Clients geht bei *serverurl*/register ein und wird vom RegisterServlet verarbeitet. Der Request hat folgende Parameter (URL-Encoded):

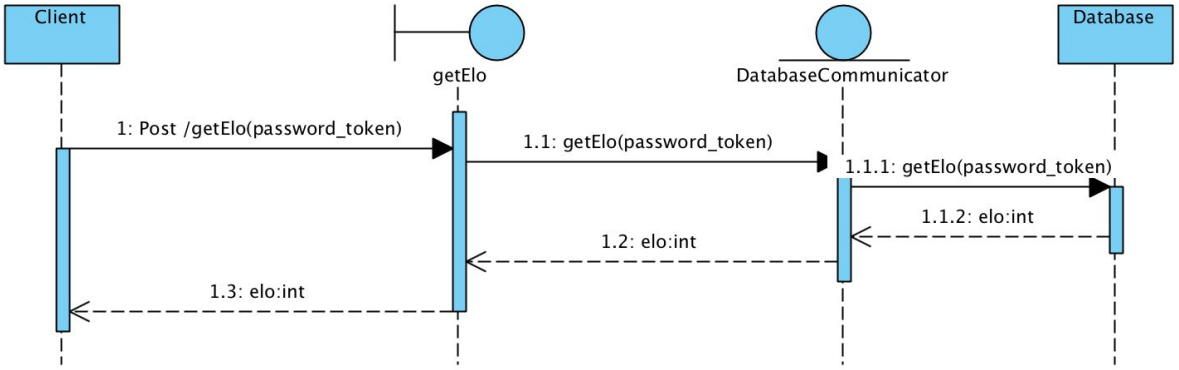
- Name: “name”
- Wert: den gewünschten Benutzernamen
- Name: “firebase_instance_id”
- Wert: die Firebase Instance Id der installierten App-Instanz

Ist der angeforderte Name noch nicht vergeben, ruft die Klasse die Methode createPlayer auf, welche einen neuen Spieler in der Datenbank anlegt. Die Datenbank erzeugt dann das Password Token, welches über die Response auf seinen Request an den Client zurückgesendet wird. Das Password Token hat den Schlüssel “password_token”.

Ist der angeforderte Name bereits vergeben, besitzt die Antwort auf den Post-Request den Statuscode 403, um dem Client mitzuteilen, dass der Name bereits vergeben ist.

4.3.3 - Elo abrufen

Sequenzdiagramm

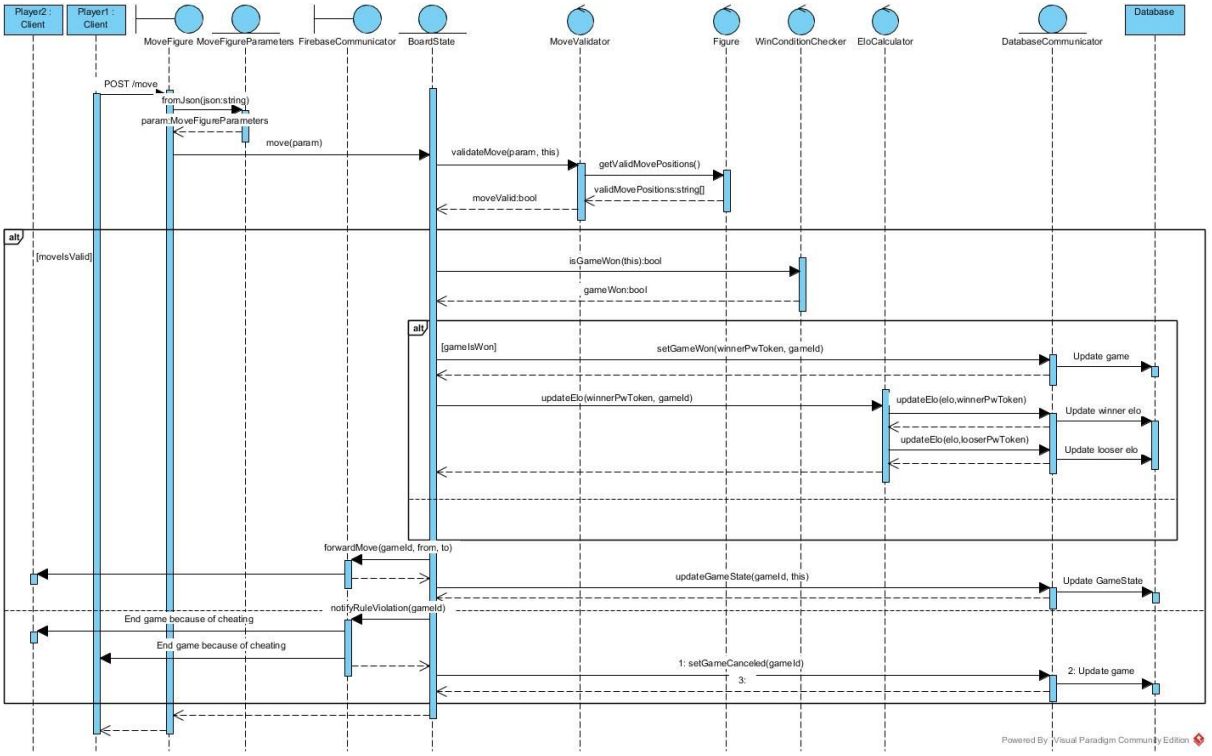


Beschreibung

Client schickt an die Klasse *getElo* ein Get-Request mit dem Pfad*serverurl*/getElo?name=*name*. Der Parameter “name” enthält den Namen des Spielers, dessen Elo abgefragt wird.
Die Antwort enthält den Parameter “elo”, der die Elo-Zahl enthält.

4.3.4 - Spielzug durchführen

Sequenzdiagramm



Beschreibung

Client schickt einen Post-Request an den Pfad*serverurl*/move. Dieser wird vom MoveFigureServlet empfangen. Der Request enthält folgende Parameter (URL-Encoded):

- “password_token” - Bei Registrierung erhalten
- “game_id” - Bei Spielstart erhalten
- “from” - Position der Figur, die bewegt werden soll (z.B. “e3”)
- “to” - Position, auf die die Figur bewegt werden soll (z.B. “e4”)

Der Move-Request wird an den BoardState des Spiels weitergeleitet, der diesen überprüfen und ins DatenModell schreiben soll.
Ist der Zug valide, wird überprüft ob das Spiel gewonnen wurde.

- Falls ja, wird der Zustand des Spiels in der Datenbank, von laufend auf beendet gesetzt und der Gewinner wird gespeichert. Weiterhin werden die Elo-Zahlen beider Spieler von EloCalculator angepasst.

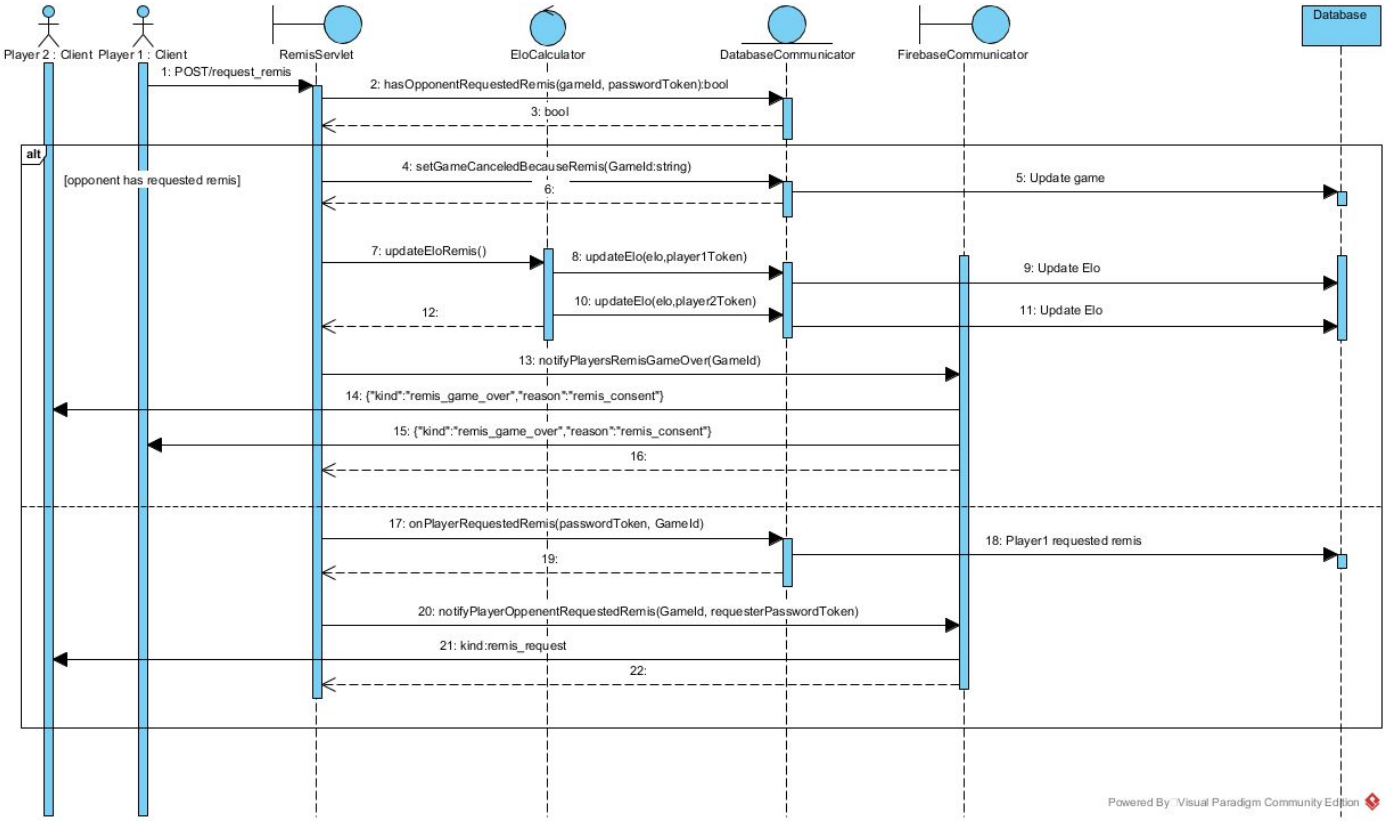
Der valide Spielzug, wird an den Gegenspieler als Firebase Message weitergeleitet. Sie enthält ein Json-Objekt mit folgenden Schlüsseln:

- “kind” - Enthält den String “oppenent_turn”
- “game_id”
- “from”
- “to”

Falls der Zug nicht valide ist, kann das Spiel nicht weitergeführt werden und beide Spieler erhalten über Firebase eine Mitteilung, dass das Spiel wegen Cheating (Betrug) vorzeitig beendet worden ist.

4.3.5 - Remis beantragen

Sequenzdiagramm



Beschreibung
Client schickt an die Klasse *Remis* ein Post-Request mit dem Pfad*serverurl*/request_remis.
Die Parameter sind:

- “password_token” - Bei Registrierung erhalten
- “game_id” - Bei Spielstart erhalten

Nun wird geprüft, ob der andere Spieler ebenfalls schon Remis beantragt hat. In diesem Fall wird das Spiel als Unentschieden beendet.

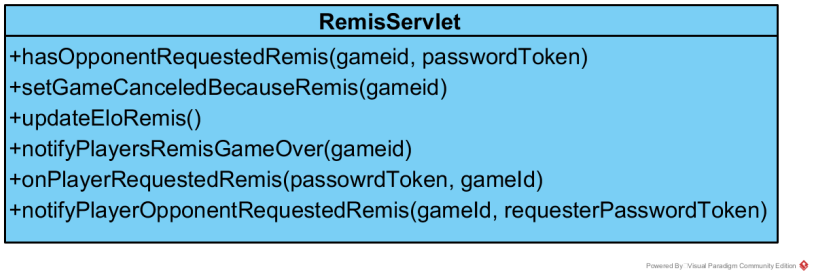
- Wenn es die erste Remis-Anforderung ist, geschieht Folgendes:
    - Das Spiel wird in der Datenbank auf “Unentschieden wegen Remis” gesetzt
    - Die neue Elo beider Spieler wird errechnet und in der Datenbank aktualisiert
    - Beide Spieler werden per Firebase über das Unentschieden Informiert. Dazu wird ein Json-Objekt mit folgenden Schlüsseln gesendet:
        - “kind” - Enthält den String “game_over”
        - “reason”- Enthält den String “remis_consent”
- Wenn es die zweite Remis-Anforderung ist, geschieht Folgendes:
    - Die Anfrage wird in der Datenbank im Spiel vermerkt
    - Dem Spieler, der nicht die Anfrage stellte, wird per Firebase mitgeteilt, dass der Gegenspieler eine Remisanfrage gestellt hat. Dazu wird ein Json-Objekt mit folgendem Schlüssel gesendet:
      “kind” - Enthält den String “remis_request”

4.4 - Beschreibung wichtiger Klassen

4.4.1 - Class Register

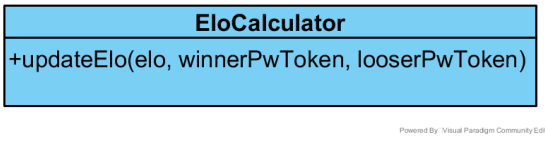
isNameInUse prüft in der *Datenbank* ob der gewünschte Spielername bereits vergeben ist.
createPlayer legt einen neuen Spieler in der *Datenbank* an.

4.4.2 - Class Remis



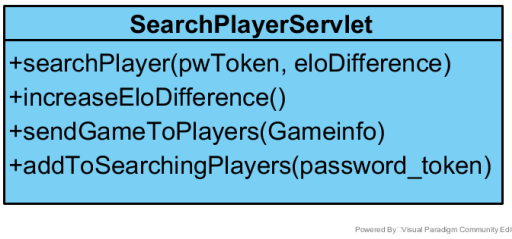
hasOpponentRequestedRemis überprüft, ob ein Spieler ein Remis beantragt oder ob ein Spieler ein Remis angenommen hat.
setGameCanceledBecauseRemis beendet das Spiel in der Datenbank.
updateEloRemis lässt die *EloCalculator* Klasse den Elowert neu berechnen, um ihn dann in der Datenbank einzutragen.
notifyPlayersRemisGameOver benachrichtigt die Spieler, dass das Spiel wegen eines Remis beendet wurde.
onPlayerRequestedRemis trägt über den *DatabaseCommunicator* in der Datenbank ein, dass eine Anfrage auf ein Remis läuft.
notifyPlayerOpponentRequestedRemis sendet dem Gegenspieler über den *FirebaseCommunicator* eine Remisanfrage.

4.4.3 - Class EloCalculator



updateElo berechnet die Elo zu dem gegebenen Spieler und übergibt diese.

4.4.4 - Class SearchPlayer

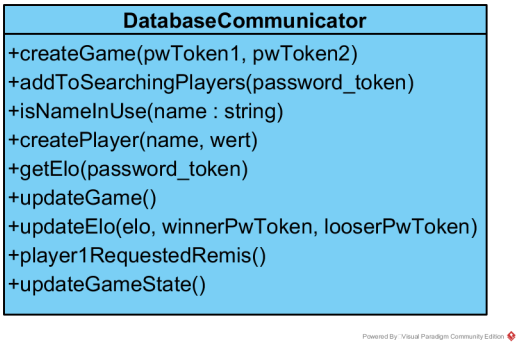


searchPlayer meldet dem *DatabaseCommunicator,* dass ein Gegenspieler gesucht wird.
increaseEloDifference weitet den Elo-Suchbereich aus.
sendGameToPlayers sendet das erstellte Spiel dem *FirebaseCommunicator*
addToSearchingPlayers meldet dem *DatabaseCommunicator,* dass der Spieler als Gegenspieler suchend verwaltet wird.

4.4.5 - Class GetElo

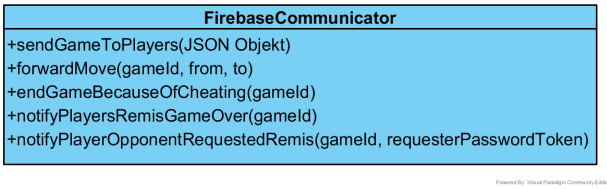
getElo ruft die Elozahl des Spielers vom *DatabaseCommunicator* ab

4.4.6 - Class DatabaseCommunicator



createGame legt ein Spiel mit den übergebenen Spielern in der Datenbank an.
addToSearchingPlayers trägt einen Spieler als Gegenspieler suchend in der Datenbank ein
isNameInUse überprüf,t ob der übergebene Name in der Datenbank bereits vorhanden ist.
createPlayer legt einen Spieler mit dem übergebenen Name in der Datenbank an.
getElo fragt den zum *password_token* zugeordneten Elo Wert in der Datenbank ab.
updateGame aktualisiert den Spielstatus in der Datenbank.
UpdateElo aktualisiert den Elo Wert des übergebenen Spielers in der Datenbank.
player1RequestedRemis trägt in der Datenbank ein, dass der Spieler ein Remis beantragt hat.
updateGameState trägt den aktuellen Spielstatus des Spielfeldes in der Datenbank ein.

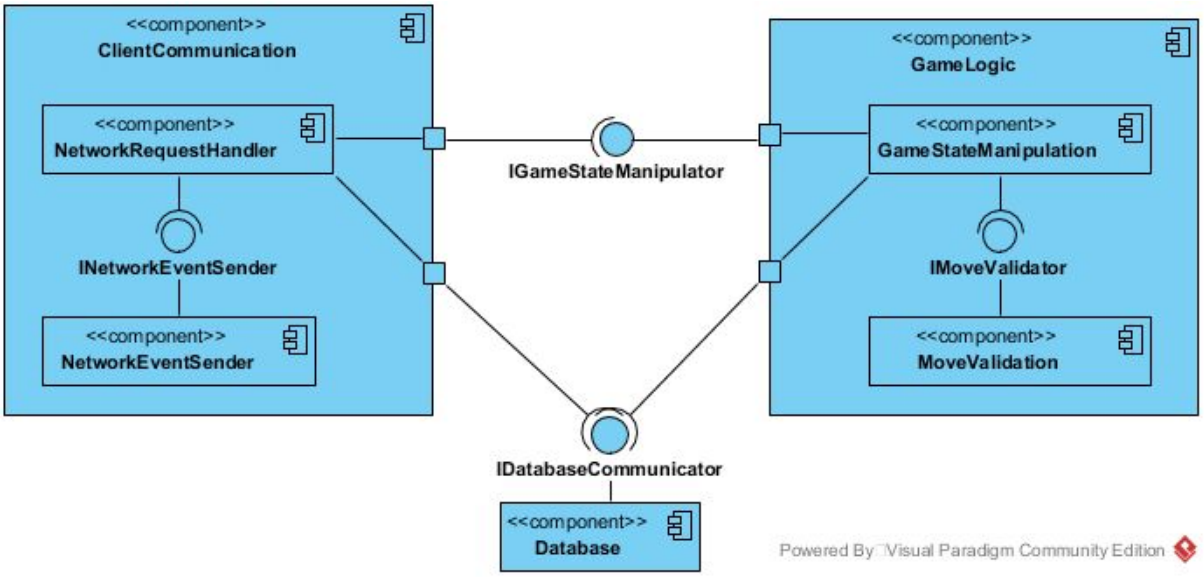
4.4.7 - Class FirebaseCommunicator



sendGameToPlayers sendet den Spielern ein Spiel *JSON Objekt.*
forwardMove übermittelt den Spielzug per Firebase an den Gegenspieler (Client).
endGameBecauseOfCheating übermittelt dem Client per Firebase, dass das Spiel wegen Cheating (Betrug) beendet wurde.
notifyPlayersRemisGameOver übermittelt dem Client per Firebase, dass das Spiel beendet wurde wegen eines Remis.
notifyPlayerOpponentRequestedRemis übermittelt dem Gegenspieler (Client) eine Anfrage das Spiel mit einem Remis zu beenden.

4.5 - Komponentensicht

4.5.1 -Komponentendiagramm



4.5.2 - Beschreibung

ClientCommunication
Die ClientCommunication-Komponente kommuniziert als einzige Komponente direkt mit dem Client.

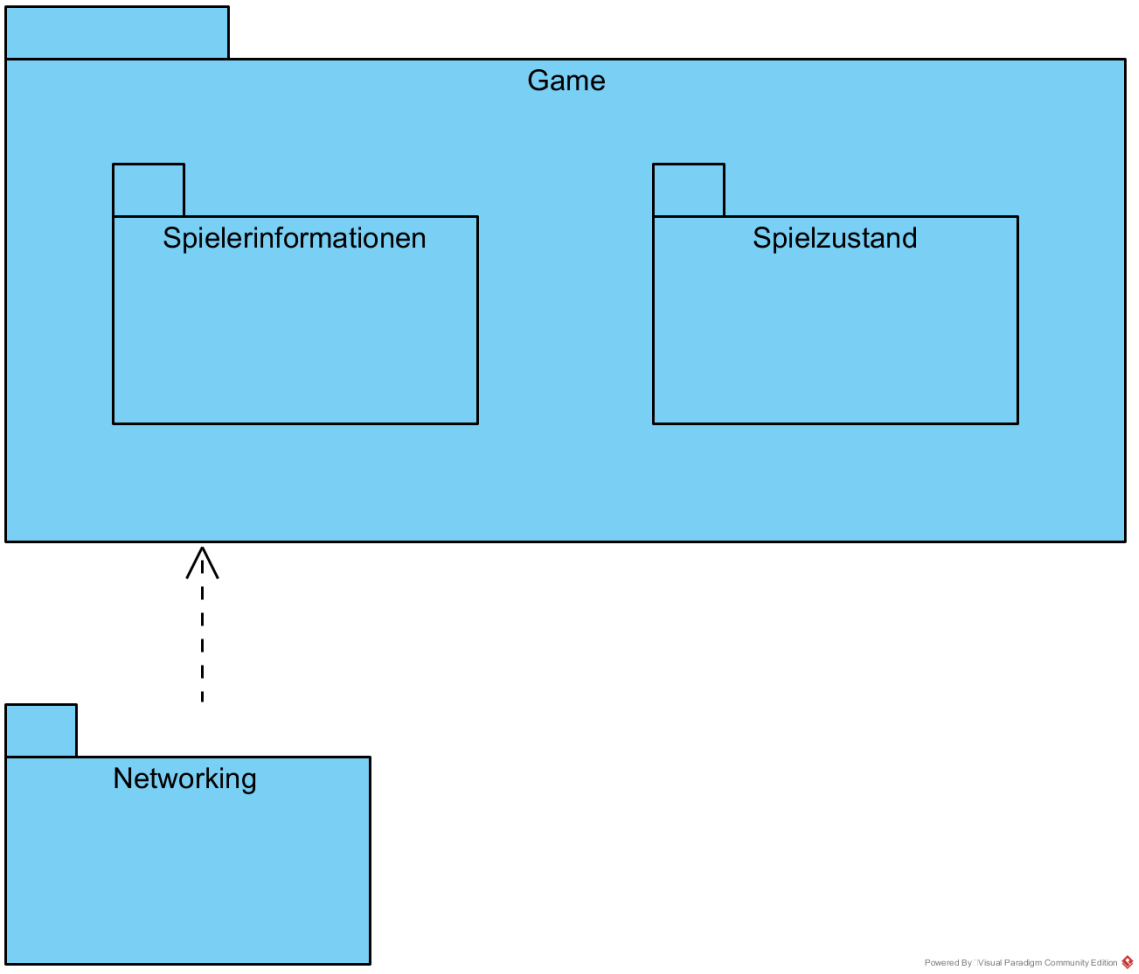
NetworkRequestHandler
Der NetworkRequestHandler empfängt die HTTP-Requests des Clients und sorgt für ihre Durchführung. Bei Requests bei denen das Spielbrett verändert wird, wird dies über das IGameStateManipulator Interface ausgeführt. Bei allen anderen Requests, kann direkt auf die Datenbank zugegriffen werden.

NetworkEventSender
Der NetworkEventSender wird verwendet, um asynchrone Nachrichten an einen Spieler zu versenden, d.h. dass sie nicht als Antwort auf einen HTTP-Request folgt und zu jeder Zeit gesendet werden kann.

GameStateManipulation
Diese Komponente bietet die Möglichkeit, den Zustand des Spielfelds eines Spiels zu manipulieren.

4.6 - Paketsicht

4.6.1 - Paketdiagramm



4.6.2 - Beschreibung

Das Unterpaket Spielerinformationen enthält alle Informationen zum Spieler und das Spielzustand Package enthält den aktuellen Spielzustand.
Das Networking Package ist für die gesamte Netzwerkkommunikation verantwortlich.