

Dokumentation VSP WiSe22/23, Gruppe Alpha-1

1. Einführung und Ziele

Dieses Projekt entsteht im Rahmen des Moduls "Verteilte Systeme" und hat zum Ziel, das Multiplayer-Spiel "Tron" zu entwickeln. Das Spiel soll lokal an einem Computer oder verteilt auf mehrere Computer gespielt werden können.

1.1 Aufgabenstellung

Allgemeine Spielprinzipien:

- Das Spiel wird mit mehreren Spielern gespielt, die jeweils ein Motorrad in einer Arena (das Spielfeld) steuern.
- Spieler bewegen sich stetig vorwärts, in einer konstanten Geschwindigkeit.
- Spieler können nach links oder rechts steuern und sich so über das Spielfeld nach oben, unten, links, rechts bewegen.
- Spieler ziehen farbige "Schatten" hinter sich auf, die für ihre Lebenszeit auf dem Spielfeld bleiben und durch die Vorwärtsbewegung länger werden.
- Ein Spieler stirbt, wenn er gegen die Wand der Arena oder den Schatten eines anderen Spielers fährt. Treffen zwei Spieler aufeinander, sterben beide.
- Wenn ein Spieler stirbt, verschwindet sein Schatten aus der Arena und er kann nicht weiterspielen.
- Alle Spieler spielen gegeneinander. Gewonnen hat der, der am längsten überlebt. Sterben die letzten beiden Spieler gleichzeitig, ist es unentschieden.

Weitere Anforderungen:

- Das Spiel soll über eine Konfigurationsdatei konfiguriert werden können.
- Das Spiel kann entweder lokal oder im Netzwerk gespielt werden. Dies soll ebenfalls über die Konfigurationsdatei einstellbar sein.

1.2 Qualitätsziele

Ziel	Beschreibung
Wohldefinierte Schnittstellen	Die Entwickler sollen sich gut um ihre Schnittstellen kümmern.
Kompatibilität	Mindestens 2 Teams müssen miteinander spielen können.
Fehlertoleranz	Ein Spiel soll ungestört durchspielbar sein. Auch wenn ein Teilnehmer abstürzt, läuft das Spiel weiter.
Fairness	Das Spiel soll fair sein. Alle Spieler starten mit fairen Konditionen und folgen den gleichen Regeln.
Bedienbarkeit	Spieler sollen das Spiel einfach bedienen können und Spaß haben.

1.3 Stakeholders

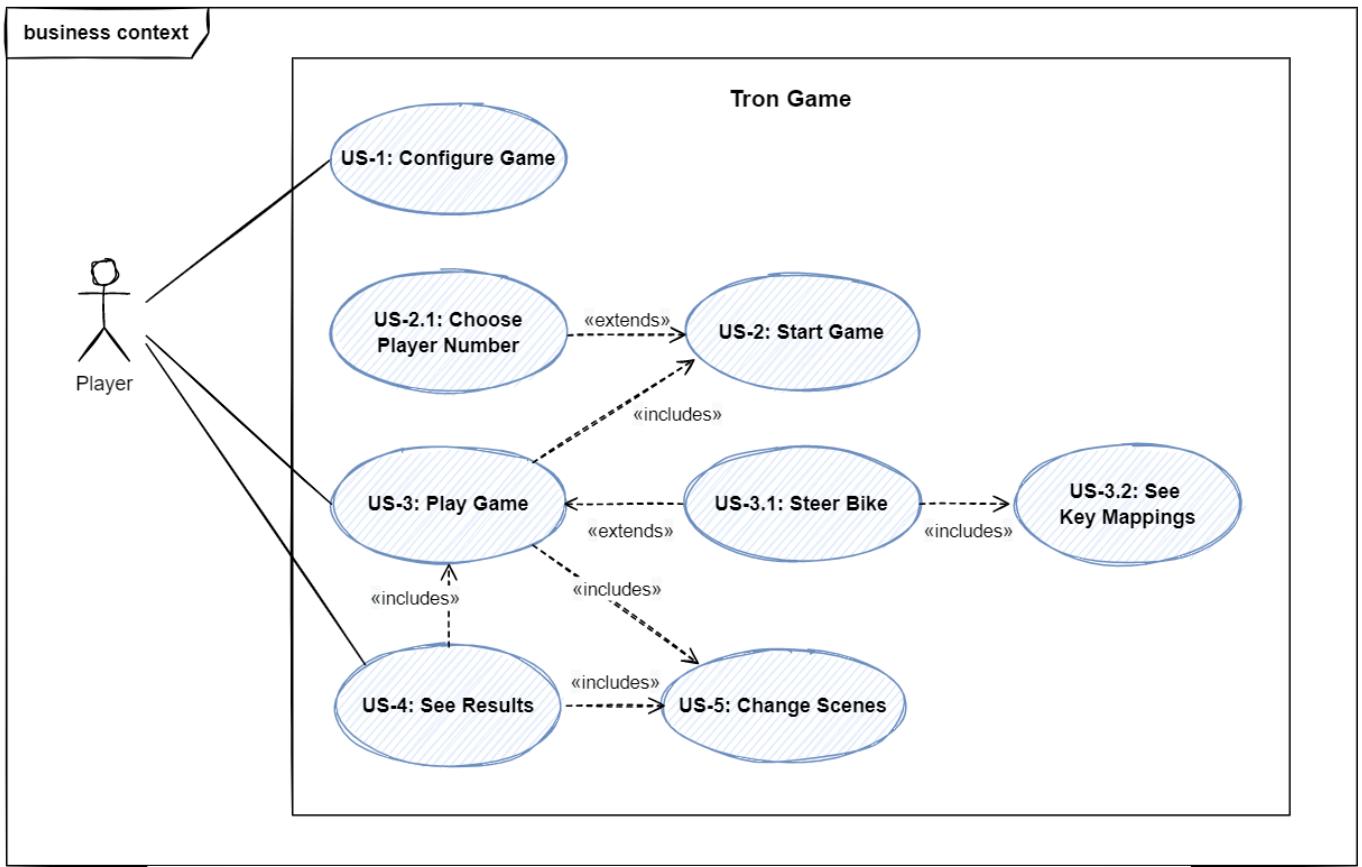
Rolle	Kontakt	Erwartungen
Dozent / Kunde	Martin Becke: martin.becke@haw-hamburg.de	Saubere Architektur mit Pattern und wohldefinierten Schnittstellen, Lernfortschritt der Entwickler
Entwickler	Sandra: sandra.koenig@haw-hamburg.de Inken: inkendulige@haw-hamburg.de Majid: majid.moussaadoyi@haw-hamburg.de	Spaß an der Entwicklung, Architekturentwurf üben, Verteilte Systeme besser verstehen, Zeitmanagement
Spieler	Teilnehmer des Moduls VS WiSe22/23	Stabile Anwendung, Spaß am Spielen

2. Randbedingungen

Technische Randbedingung	Beschreibung
Java in der Version 17	Zur Implementierung wird Java verwendet, da das ganze Team die Sprache beherrscht. Die Version muss zum Image der Rechner im Raum 7.65 passen. Es wird Java in der Version 17 verwendet, da es sich um die neueste LTS-Version handelt.
View Library	Es wird die zur Verfügung gestellte JavaFX View Library verwendet, um Zeit in der UI-Erstellung zu sparen.
Versionsverwaltung in GitLab	Alle Dateien dieses Projektes werden über Git verwaltet. Das zentrale Remote Repository befindet sich im GitLab der HAW. Wegen technischer Störung der HAW wurde am 30.12.2022 auf gitlab.com gewechselt
Konventionen	Beschreibung
Dokumentation	Gliederung erfolgt nach dem deutschen arc42-Template, um Struktur zu wahren.
Sprache	Die Dokumentation erfolgt auf deutsch, während die Diagramme auf Englisch gehalten werden, um die Umsetzung in (englischen) Code zu erleichtern.

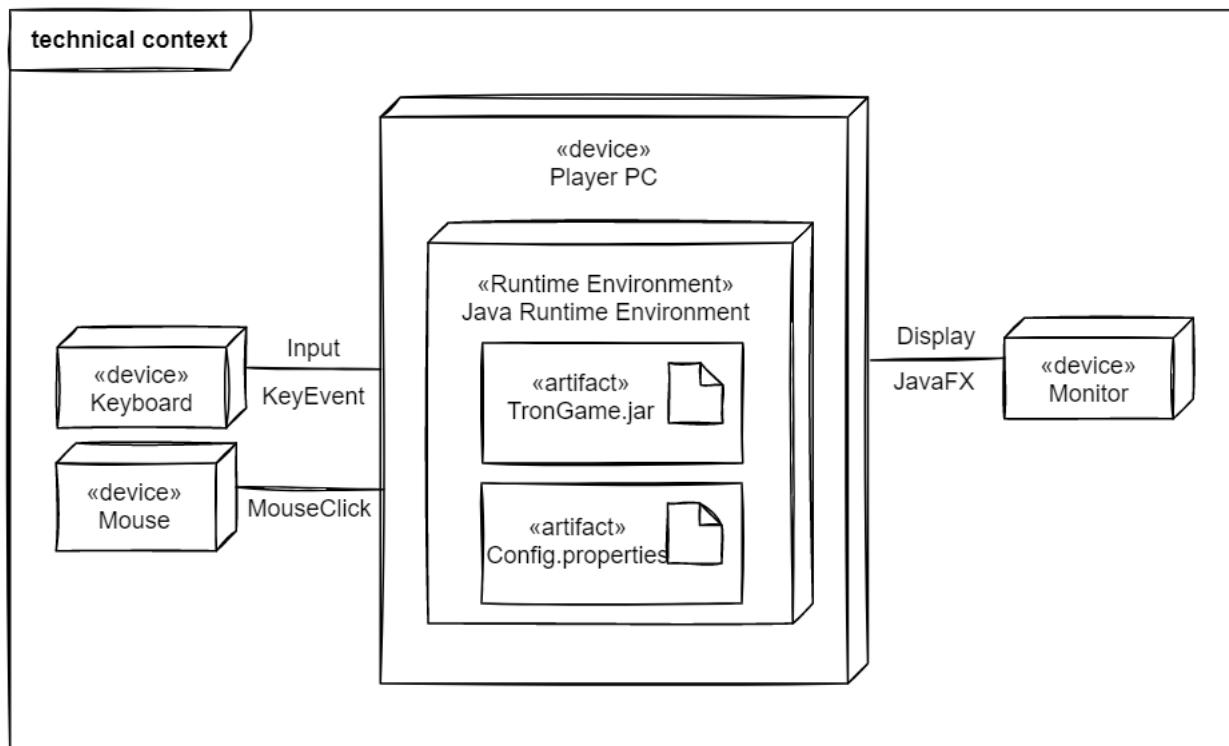
3. Kontextabgrenzung

3.1 Business Kontext



Details siehe [Use Cases](#).

3.2 Technischer Kontext



4. Lösungsstrategie

Aus den Use Cases ergeben sich folgende benötigte Objekte, denen in folgenden Abschnitten Funktionen zugeordnet werden.

Objekt	Erklärung
Configuration	Verwaltet die anpassbaren Werte und stellt Defaultwerte bereit.
ITronView	Stellt die Hauptkomponente der UI dar.
ITronModel	Verwaltet die gesamte Spielelogik.
ITronController	Verwaltet die Benutzereingaben.
IGame	Stellt die Spielelogik und startet die Spielschleife.
IGameManager	Managt die Spieler die, mitspielen wollen sowie in welcher Phase sich das Spiel befindet.
IArena	Verwaltet die Spieler innerhalb der Arena, merkt sich die Positionen der Spieler, sowie die Schatten der Spieler.
ICollisionDetector	Ist für die Überprüfung ob ein Spieler, mit einem anderen Spieler, dessen Schatten oder einer Arenawand zusammengestoßen ist.
IPlayer	Verwaltet einen einzelnen Spieler, mit den Koordinaten, der Richtung und der Farbe eines Spielers.
IUpdateListener	Bekommt von dem Model Aktualisierungen und aktualisiert die View dementsprechend.
Overlay	Eine Oberfläche, die in der UI angezeigt werden kann. Für jede Scene(Menu, Waiting, Countdown, Ending) wird ein Overlay benötigt.
TronViewBuilder	Builder-Objekt, das den Zusammenbau der View (aus ITronView, Overlays und IUpdateListener) übernimmt.

4.1.2 Configuration

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
1	<code>loadConfigFromFile() : void</code>	Configuration	Es existiert ein Config-Objekt. Es existiert eine TronConfig.properties File.	Properties-Objekt ist erzeugt.	Die Konfigurationsdatei ist ein '.properties' File, in Form: 'Key', 'Value' und wird in ein Properties-Objekt eingelesen, worauf die Configuration eine Referenz hält. Prüft mit <code>isConfigValid()</code> auf valide Werte.	Bei fehlerhaften Werten oder fehlender .properties-Datei an der erwarteten Speicheradresse, wird neue .properties-Datei erstellt (<code>reloadConfig()</code>).
1	<code>isConfigValid() : boolean</code>	Configuration	Es existiert ein Properties-Objekt.	-	Der Inhalt des Properties-Objekt wird auf fehlende 'Keys' geprüft und ob die 'Values' sich im richtigen Wertebereich befinden. Verwendet dafür <code>hasAllAttributes()</code> , <code>isSteerValid</code> , <code>isValidIpAddress</code> , <code>isNumber</code> .	-
1	<code>reloadConfig() : Properties</code>	Configuration	Schreibrechte.	Properties mit 'default'-Werten wurde erzeugt.	Es wird eine neues Properties-Objekt auf Basis vom im Programmcode festgelegten 'Key-Value-Paaren' in der Config erstellt. Das Properties-Objekt wird auch an der hinterlegten Speicheradresse lokal in Form einer .properties-Datei hinterlegt.	-

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
1	<code>setKeyMappings() : void</code>	Configuration	Es existiert ein Properties-Objekt mit validen Daten.	In dem Config-Objekt existiert eine Map, welche als 'Key' alle Tasten enthält, welche zum lenken genutzt werden können. Als Value erhält die Map ein Steer-Objekt welches die Player-ID, sowie die Direction hält.	Die Methode zieht sich aus dem properties-Objekt die Tastenbelegungen aller Spieler.	-
1	<code>getAttribute(key : String) : String</code>	Configuration	Es existiert ein Properties-Objekt mit validen Daten.	Es wurde der passende 'Value' zum 'Key' zurückgegeben.	Die Methode greift auf ein Properties-Objekt zu und zieht sich den ersten 'Value' welcher zu dem Eingabeparameter String passt.	-
3.1	<code>getSteer(key : KeyCode) : Steer</code>	Configuration	KeyMappings wurden erfolgreich erstellt.	Steer-Objekt	Als 'Value' enthält die Map ein Steer-Objekt, welches die Player-ID und die Direction enthält. Die Methode gibt das zur Taste gehörende Steer-Objekts zurück.	Gibt null zurück, wenn es für die eingegebene Taste keinen Treffer gibt.
1	<code>hasAllAttributes() : Steer</code>	Configuration	Es existiert ein Properties-Objekt.	Abhängig davon ob Einträge in der Properties fehlen wird ein boolean zurückgegeben.	Die Methode greift auf ein Properties-Objekt zu und vergleicht dieses mit der Default-Properties. Wenn Keys fehlen ist dies gleichzusetzen damit, dass Attribute fehlen.	-
1	<code>isNumber(strings : String...) : boolean</code>	Configuration	Es existiert ein Properties-Objekt.	Abhängig davon ob alle der Eingabestrings nummerisch sind oder nicht erhält man true oder false.	Es wird über die Eingabe-Strings iteriert und für jeden String geschaut ob er nummerisch ist oder nicht. Sollte auch nur einer nicht nummerisch sein ist das gesamte Ergebniss false	-
1	<code>isSteerValid(strings : String...) : boolean</code>	Configuration	Es existiert ein Properties-Objekt.	Abhängig davon ob die Eingabe-Strings zu einem Steer umgewandelt werden können erhält man true oder false	Anhand der Länge der Eingabe und des Inhalts wird überprüft, ob aus der Eingabe ein valides Steer-Objekt erstellt werden könnte.	-
1	<code>isValidKey(tempStringArray : String[] , index : int) : boolean</code>	Configuration	Es existiert ein Properties-Objekt.	Abhängig davon ob die Player-Tasten valide sind erhält man true oder false	Es wird überprüft, ob es sich bei den Tasten für die Steuerung um valide Eingaben handelt. Als valide gelten alle Zahlen, Buchstaben mit Außnahme (ä,ö,ü) und alle Pfeiltasten.	-
1	<code>isValidIpAddress(ipAddress : String) : boolean</code>	Configuration	Es existiert ein Properties-Objekt.	-	Es wird überprüft, ob es sich um eine richtig geformte IPv4 Adresse handelt.	
3.2	<code>getKeyMappingForPlayer(id : int) : String</code>	Configuration	Es existiert ein Properties-Objekt.	String mit Tasten für die Steuerung des Spielers in Tupel-Form Bsp.: "A,Z"	Die Eingabe-Id wird auf den Spieler in der Properties gemappt, sodass z.B. die ID = 1, den Value zum Eintrag P_EINS (Key) aus der Properties zieht.	-

4.1.1. ITronView

Es wird die zur Verfügung gestellte view library verwendet. Die ITronView wird lediglich um Setter für ROWS und COLUMNS erweitert.

4.1.3 ITronModel

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
2	initialize(config : Configuration, modus : Modus, singleView : boolean, executorService : ExecutorService) : void	ITronModel	Der GameManager-Objekt existiert wurde instanziert.	GameManager wurde initialisiert.	Das Model wird mit den erforderlichen Abhängigkeiten gespeist.	-
2	playGame(listener : IUpdateListener, playerNumber : int) : void	ITronModel	Der GameManager wurde instanziert.	GameManager beim Spiel registriert.	Der Initiator des Spiels startet beim Model ein Spiel mit der gewünschten Spielerzahl	Wenn der ModelState nicht dem Status MENU oder WAITING entspricht, wird der Aufruf ignoriert.
3.1	handleSteerEvent(registrationId : int, key : String) : void	ITronModel	Der GameManager wurde instanziert. Die registrationId ist dem Model bekannt.	Das Model hat auf das Event reagiert.	Ein Tastenanschlag wird an das Model weitergegeben.	Falls Key nicht bekannt ist oder die registrationId unbekannt ist, wird der Tastenanschlag verworfen.

4.1.4 ITronController

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
2,3	initialize(model : TronModel) : void	ITronController	Model wurde instanziert.	Der Controller ist über sein Modell informiert.	Legt das Modell des Controllers fest, an das er die Eingaben leitet.	-
2	playGame(listener : IUpdateListener, playerCount : int) : void	ITronController	-	Das Model reagiert auf den Aufruf.	Beauftragt das Model, ein Game zu starten.	-
3.1	handleKeyEvent(registrationId : int, event : KeyEvent) : void	ITronController	-	Das Model wurde über einen Tastenanschlag informiert	Übersetzt ein KeyEvent zu dem dazugehörigen String und leitet diesen ans Model weiter.	-

4.1.5 IGame & Game

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlerseman
2	initialize(modus : GameModus, speed : int, rows : int, columns : int, waitingTimer : int, endingTimer : int, executorService : ExecutorService) : void	IGame	-	Game-Objekt ist initialisiert.	Initialisiert ein game nach Erstellung des Game-Objekts. Ähnliche Funktionalität wie Initialisierung mittels Konstruktors. Durch Anwendung des Factory-Patterns aber in Methode verlagert.	-
2	prepareForRegistration(playerCount : int) : void	IGame	Ein Game Objekt wurde erzeugt und im State INIT. PlayerCount ist zwischen 2 und 6.	Das Game Objekt ist bereit für den Spielstart.	Das Game wird für die Registrierung vorbereitet: Es merkt sich den gewünschten playerCount, erstellt eine Arena und startet einen Timer, nach dem die Vorbereitung beendet wird (waitingTimer der Config-File).	-

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlerseman
2	<code>register(gameManager : IGameManager, listener : IUpdateListener, playerId : int, managedPlayerCount : int) : void</code>	IGame	Ein Game Objekt wurde erzeugt und befindet sich im GameState REGISTRATION.	Das Game speichert sich seine Observer und managedPlayerCount Player erstellt.	Das Game merkt sich seine Observer, die es über das Spielgeschehen informieren soll und erstellt so viele Spieler, wie übergeben wird. Es gibt die IDs der erstellten Spieler zurück.	register wird aufgerufen während der GameState in einem andere Zusatnd als REGISTRATIO ist. - Die Registrierung wird nicht vollzogen. Es gibt keinen Hinweis oder Fehler.
3	<code>handleSteer(steer : Steer) : void</code>	IGame	Steer ist nicht NULL.	Player hat eine neue Direction.	Dem Player mit der id aus dem Steer Object wird als neue Direction, die Direction steer eingetragen.	-
2,3,4,5	<code>transitionState(newState : GameState) : void</code>	Game	GameState ist nicht NULL.	-	Der newState wird im Game gesetzt und der stateListener wird informiert (<code>executeState()</code>)	Wenn der newState den currentState entspricht passiert nicht
2,3,4,5	<code>executeState() : void</code>	Game	-	-	Führt basierend auf dem currentState, die nächste Methode aus.	-
2,4	<code>startTimer(waitingTimer : int, startedAt: GameState) : void</code>	Game	-	-	Started einen Timer mit der Zeit des waiting Timers und ruft wenn der Timer um ist <code>handleTimeOut(startedAt)</code> auf.	-
2,4	<code>handleTimeOut(startedAt : GameState) : void</code>	Game	GameState == REGISTRATION oder GameState == FINISHING	Game State == STARTING oder Game State == INIT	Ist der WaitingTimer abgelaufen und das Game befindet sich noch im REGISTRATION State, wird die Vorbereitung beendet. > 2 Spieler: Spiel wird gestartet, < 2 Spieler: Spiel kehrt ins Menü zurück, mit Methodenaufruf <code>transitionState()</code> . Befindet sich das Game im State FINISHING und der WaitingTimer ist abgelaufen, kehrt das Spiel ins Menü zurück	-
2	<code>isGameReady() : boolean</code>	Game	-	-	Es wird überprüft ob genügend Player im Spiel sind. Falls nicht gibt die Methode false zurück.	-
2	<code>isGameFull() : boolean</code>	Game	-	-	Es wird überprüft ob die gewünschte Spielerzahl bereits erreicht wurde. Falls nicht gibt die Methode false zurück.	-
2	<code>isRegistrationAllowed(playerCountToRegister : int) : boolean</code>	Game	-	-	Wenn der GameState sich auf REGISTRATION befindet und noch Plätze frei sind gibt die Methode true zurück. - Andernfalls false.	-
2	<code>createPlayers(count : int) : List<Integer></code>	Game	Count darf und kann nicht größer sein als 6.	neue Player-Objekte wurden erstellt.	Erstellt die übergebene Anzahl an Spieler und speichert diese im Game. Gibt eine Liste der Ids zurück.	-

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlerseman
2	<code>startGame() : void</code>	Game	Es müssen mindestens zwei Player erstellt sein.	Positionen der Player sind ermittelt und an diese verteilt.	Die Vorbereitung zur Verteilung der Player auf der Arena wird vorgenommen, die Informationen werden an die UpdateListener weitergereicht und das Game wechselt in den Status RUNNING.	-
3	<code>countDown() : void</code>	Game	Es muss ein Game-Objekt erstellt worden sein und das 'Game' wurde erfolgreich initialisiert.	-	Ein CountDown welcher für drei Sekunden runter zählt. In jeder Sekunde werden die Observer informiert.	-
3	<code>runGame() : void</code>	Game	-	-	Der <code>countDown()</code> und die <code>gameLoop()</code> werden gestartet.	-
3	<code>gameLoop() : void</code>	Game	Es muss ein Game-Objekt erstellt und das 'Game' erfolgreich vorbereitet worden sein && Der Countdown ist abgelaufen.	Es ist ein oder kein Spieler am Leben.	Die <code>gameLoop()</code> ist eine Schleife, in der die primäre Spiellogik implementiert ist. Sie berechnet in jedem Takt die neue Koordinate der Spieler anhand deren Direction (<code>calculateNextCoordinate()</code>). Ebenfalls überprüft sie, ob Spieler kollidieren oder ob ein Spiel zuende ist (<code>isGameOver()</code>).	-
3	<code>movePlayers() : void</code>	Game	Player ist "alive".	Player hält ein neues Coordinate Tupel in seiner Liste an Coordinates.	Wenn der Player noch am leben ist, wird für ihn seine nächste Position abhängig von seiner Direction berechnet.	-
3	<code>updateField() : void</code>	Game	-	Das Spielfeld wird aktualisiert.	Alle Kollisionen auf der Arena, entweder Player mit Player oder Player mit Arena werden ermittelt und die verunfallten Player werden durch <code>detectCollision()</code> ermittelt und entfernt. Im Anschluss werden die registrierten UpdateListener informiert.	-
3	<code>updatePlayerMap() : Map<Integer, List<Coordinate>></code>	Game	-	-	Erzeugt eine Map in einer Form, die die Listener zum Aktualisieren des Spielfeldes benötigen.	-
3	<code>calculateNextCoordinate(coordinate : Coordinate, direction : Direction) : Coordinate</code>	Game	Direction darf nicht NULL sein.	Es wurde eine neue Coordinate berechnet	In Abhängigkeit von der Direction wird eine neue Coordinate berechnet.	-
3	<code>isGameOver() : boolean</code>	Game	Das Game wurde gestartet.	Ergebnis ist wahr und die Loop geht weiter, oder false und die Loop wird beendet.	Wenn der Counter der aktiven Player < 2 dann gibt die Methode den Wert 'true' zurück andernfalls 'false'	-
3	<code>finishGame() : void</code>	Game	Die Gameloop ist beendet.	Das Spielergebnis steht fest.	Ermittelt das Ergebnis und informiert Listener über das Spiel Resultat.	-
3	<code>resetGame() : void</code>	Game	Die Gameloop ist beendet oder PREPARING war nicht erfolgreich.	Der Zustand des Game-Objekts ist wieder im 'INIT'-Zustand.	Setzt alle Werte des Games zurück und leert die Arena, wenn ein Spiel vorbei ist oder die Vorbereitung abgebrochen wurde.	-

4.1.6 IGameManager & GameManager

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
2,3,4	<code>handleManagedPlayers(id: int, managedPlayers : List<Integer>) : void</code>	IGameManager	Es existiert ein GameManager-Objekt	Der GameManager hält einen bzw. mehrere neue Player die er verwaltet.	Der GameManager wird über die Player informiert, welchen er verwalten soll. Intern erfolgt ein Mapping der Steuerung auf die playerId.	-
2,3,4	<code>handleGameState(gameState : GameState) : void</code>	IGameManager	Der GameManager wurde über handleGameState(gameState:GameState) über eine Veränderung informiert.	Der ModelState des GameManagers hat sich gemäß des GameStates verändert.	Das Model wechselt vom aktuellen ModelState in den nächsten ModelState abhängig von der Nachricht.	Ist eine Nachricht nicht gültig im aktuellen ModelState, wird sie ignoriert.
2,3,4	<code>executeState() : void</code>	GameManager	Es gab einen Zustandsübergang.	Die 'do's des States wurden durchgeführt.	In Abhängigkeit vom ModelState zeigt der GameManager Overlays an, initialisiert ein Game etc. (Verweis auf das State-Diagramm)	-
2,3,4	<code>transition(newState : ModelState) : void</code>	GameManager	-	Der Ablauf zum wechsel des Zustands wird initialisiert und der neue Zustand wird intern abgespeichert.	Intern wird der neue übergebene ModelState gesetzt und executeState() wird aufgerufen.	-
2,3,4	<code>updateListeners() : void</code>	GameManager	Es gab einen Zustandswechsel des ModelState.	Die UpdateListener wurden informiert.	Bei allen Zustandswechsel des ModelState's außer bei PLAYING werden die listener über den neuen ModelState informiert.	-
4	<code>reset() : void</code>	GameManager	Es existiert ein GameManager-Objekt.	GameManager hält keine Spieler Informationen mehr.	Der GameManager wird in einen Status gebracht, in dem er alle informationen über seine gehaltenen management Daten verliert.	-

4.1.7 IArena & Arena

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
3	<code>addPlayerPosition(playerId : int, coordinate : Coordinate) : void</code>	IArena	Der Spieler mit der playerId muss noch am leben sein. coordinate darf nicht NULL sein und muss sich innerhalb der Arena befinden.	Die Arena wurde aktualisiert	Die aktuell Head-Koordinate des übergebenen Players wird in die Arena eingetragen.	-

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
3	deletePlayerPositions(playerIds : List<Integer>) : void	IArena	Die Liste playerIds darf nicht leer sein.	Die Arena wurde aktualisiert und die übergebenen Spieler rausgelöscht.	Alle Koordinaten der übergebenen ID's werden aus der Arena entfernt.	Wenn die Liste der ID's leer ist, wird die Methode abgebrochen.
3	detectCollision(coordinate : Coordinate) : boolean	IArena	Die coordinate darf nicht NULL sein.		Es wird geschaut ob sich die Koordinate außerhalb der Arena befindet und ob sich an der Koordinate bereits der Schatten oder auch anderer Spieler befindet. Falls eine Kollision entdeckt wird, geben wir true zurück, ansonsten false.	
2	calculateFairStartingCoordinate(playerCount : int) : List<Coordinate>	IArena	Der playerCount muss zwischen 2 und 6 liegen.	Es wurden playerCount viele Startpositionen berechnet.	Es werden je nach Spieleranzahl und Arenagröße die fairsten Startpositionen berechnet.	
2	calculateStartingDirection(coordinate : Coordinate) : Direction	IArena	Die coordinate darf nicht NULL sein und muss sich innerhalb der Arena befinden.	Für die coordinate wurde eine Startrichtung berechnet.	Abhängig von der übergebenen Koodinate wird die Startrichtung berechnet.	

4.1.8 ICollisionDetector & CollisionDetector

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
3	detectCollision(players: List<Player>, arena : Arena) : void	ICollisionDetector	Anzahl aktiver Spieler > 1	-	Es wird überprüft ob ein Player mit einem anderen Player, dem Schatten eines anderen Player oder der Arenawand kollidiert.	-
3	detectHeadCollision(players: List<Player>) : boolean	CollisionDetector	Anzahl aktiver Spieler > 1	-	Es wird überprüft ob ein Player mit dem head eines anderen Players kollidiert.	-

4.1.9 IPlayer

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
3	addCoordinate(coordinate : Coordinate) : void	IPlayer	Die Coordinate ist nicht NULL	Die Koordinatenliste ist um +1 gestiegen.	Dem Spieler wird eine neue Koordinate in seine List hinzugefügt.	-
3	isAlive() : boolean	IPlayer	-	-	Der Spieler kann entweder noch aktiv am Spiel beteiligt sein oder nicht. Dies wird mit der Funktion abgefragt. Ein wechsel dieses Status erfolgt durch eine Kollision mit Playern (inkl. sich selbst) oder der Arena-Wand.	-
3	crash() : void	IPlayer	Der Spieler ist am Leben und in der aktuellen Spielrunde gecrashed	Der Spieler kann nicht mehr mitspielen	Setzt den alive-Status eines Spielers auf "false" nach einem Crash.	-
3	performDirectionChange() : Direction	IPlayer	Neuer Takt hat begonnen.	Die Direction des Players wurde der Action entsprechend verändert.	Pro Takt wird die Richtung jedes Spielers entsprechend seiner Action verändert. Die Action wird danach auf NONE gesetzt.	-

4.1.10 IUpdateListener & UpdateListener

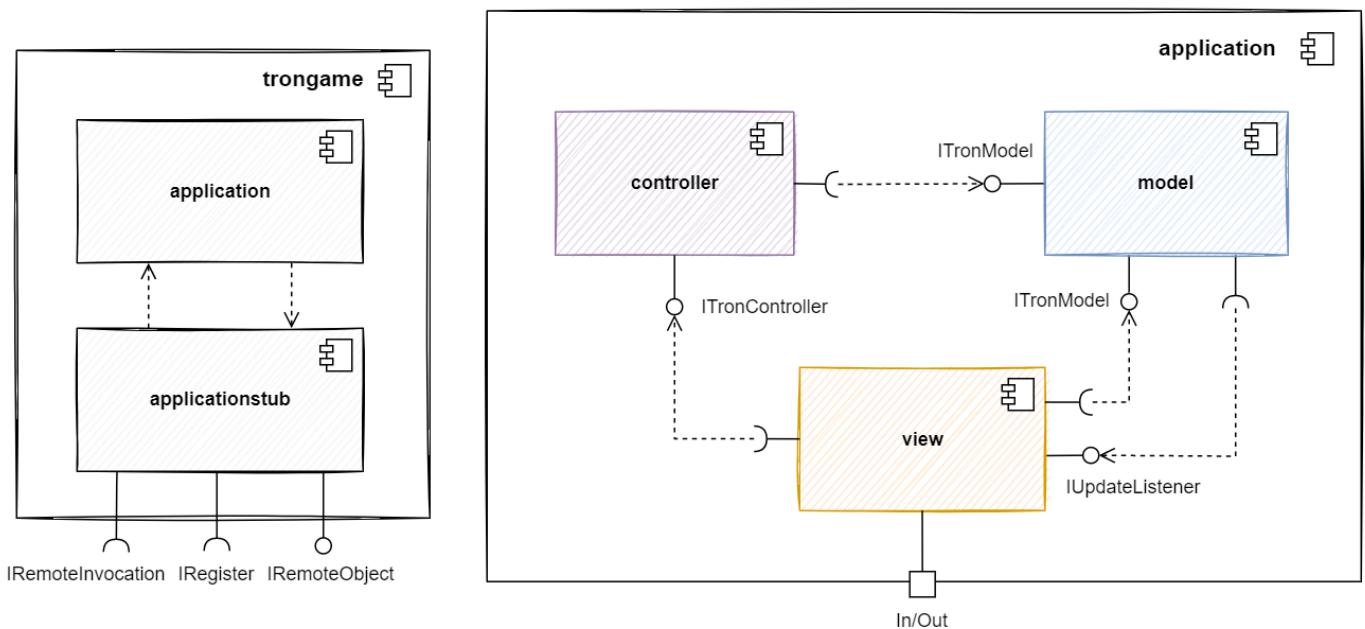
UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
2,3	updateOnRegistration(id : int) : void	IUpdateListener	-	UpdateListener kennt seine Registration-Id	Der UpdateListener wird vom Model über die vergebene Id informiert.	-
3,2	updateOnKeyMappings(mappings : Map<String, String>) : void	IUpdateListener	-	Die View wurde entsprechend aktualisiert	Der UpdateListener wird über die Keys, die im CountdownOverlay angezeigt werden sollen informiert und setzt diese im CountdownOverlay.	-
2	updateOnArena(rows : int, columns : int) : void	IUpdateListener	-	Die View wurde entsprechend aktualisiert	Informiert den UpdateListener über die Größe der Arena, die entsprechende View wird angepasst.	-
5	updateOnState(state : String): void	IUpdateListener	-	Die View wurde entsprechend aktualisiert	Der UpdateListener wird über den State des Model informiert und setzt die View auf das zum State passende Overlay.	-
2	updateOnGameStart() : void	IUpdateListener	-	Die View wurde entsprechend aktualisiert	Informiert den UpdateListener über den Start des Spieles, alle Overlays werden versteckt, sodass kein Overlay mehr angezeigt wird.	-
4	updateOnGameResult(color : String, result : String)	IUpdateListener	-	Die View wurde entsprechend aktualisiert	Informiert den UpdateListener über das Ende des Spiels, der Sieger wird angezeigt, das CountdownOverlay wird zurückgesetzt und alle Koordinaten werden auf der View gelöscht.	-
3	updateOnCountDown(value : int) : void	IUpdateListener	-	Die View wurde entsprechend aktualisiert	Der UpdateListener wird über den nächsten Schritt im Countdown informiert und setzt das CountdownOverlay auf den übergebenen Wert.	-
3	updateOnField(field : Map<Color, List<Coordinate>>) : void	IUpdateListener	-	Die View wurde entsprechend aktualisiert	Der UpdateListener wird über den aktuellen Zustand des Spielfeldes informiert und der aktuelle Zustand wird in die View übertragen.	-
2-5	initialize(mainView : ITronView, countdownOverlay : CountdownOverlay, endingOverlay : EndingOverlay, mainController : ITronController) : void	UpdateListener	-	-	Der UpdateListener wird initialisiert.	-

4.1.11 TronViewBuilder

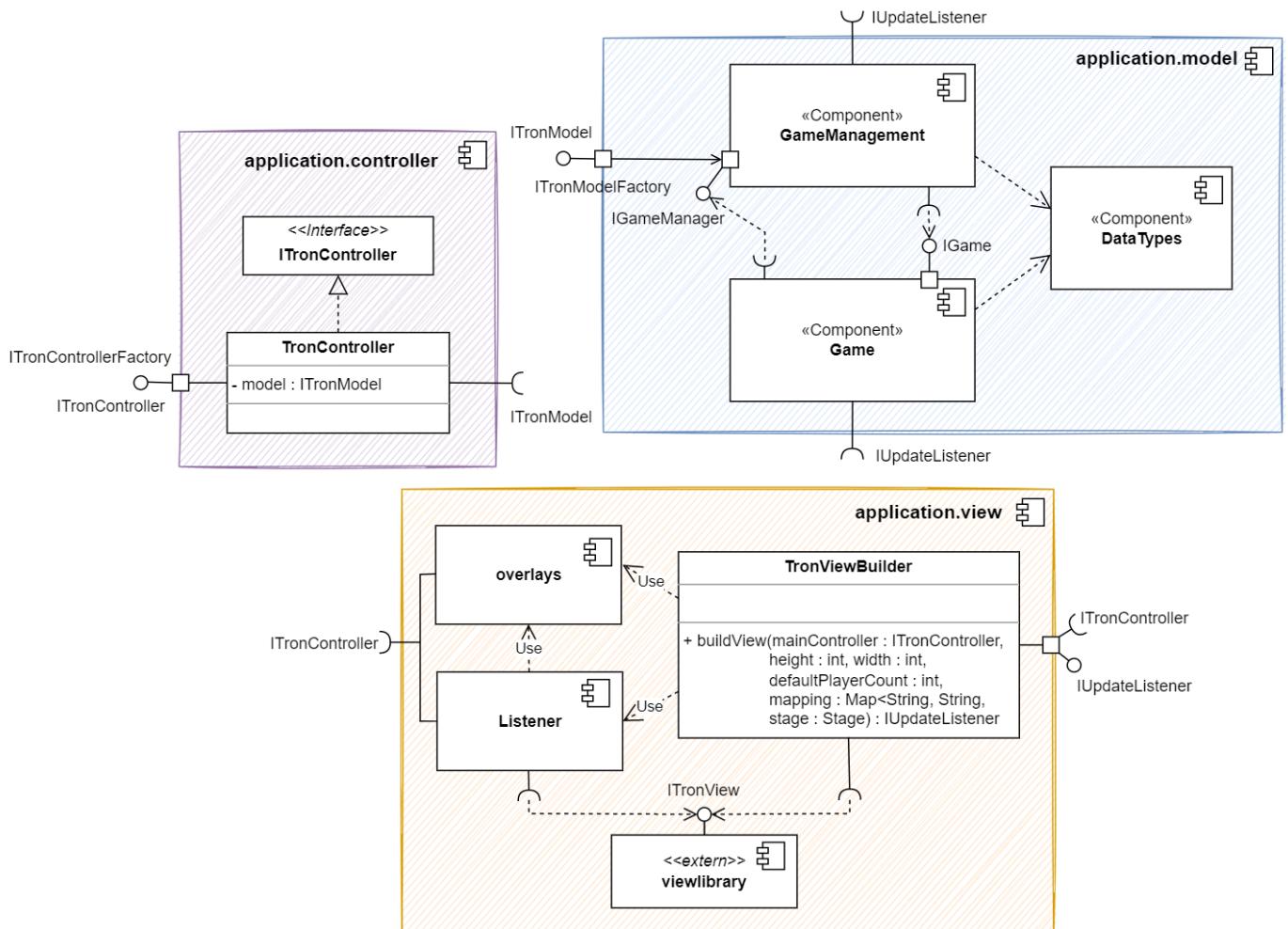
UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
2-5	buildView(controller : ITronController, height : int, width : int, defaultPlayerCount : int, idOverlayMapping : Map<String, String>, stage : Stage) : IUpdateListener	TronViewBuilder	ITronController wurde instanziert.	Instanzen von ITronView, IUpdateListener sowie Overlays wurden erzeugt.	Beauftragt den Builder, die View aus ihren verschiedenen Komponenten zusammenzubauen.	-

5. Bausteinsicht

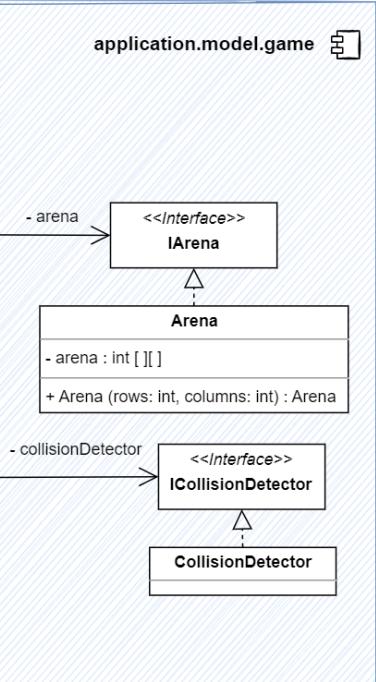
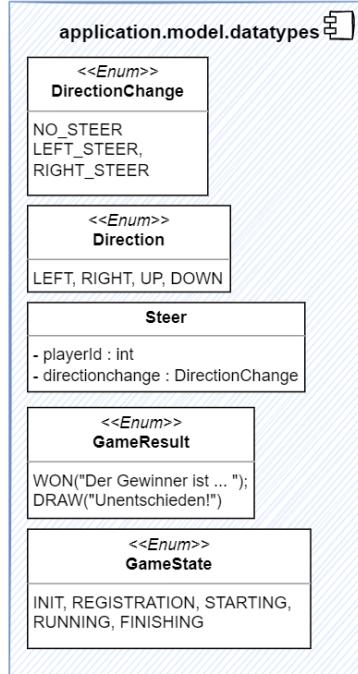
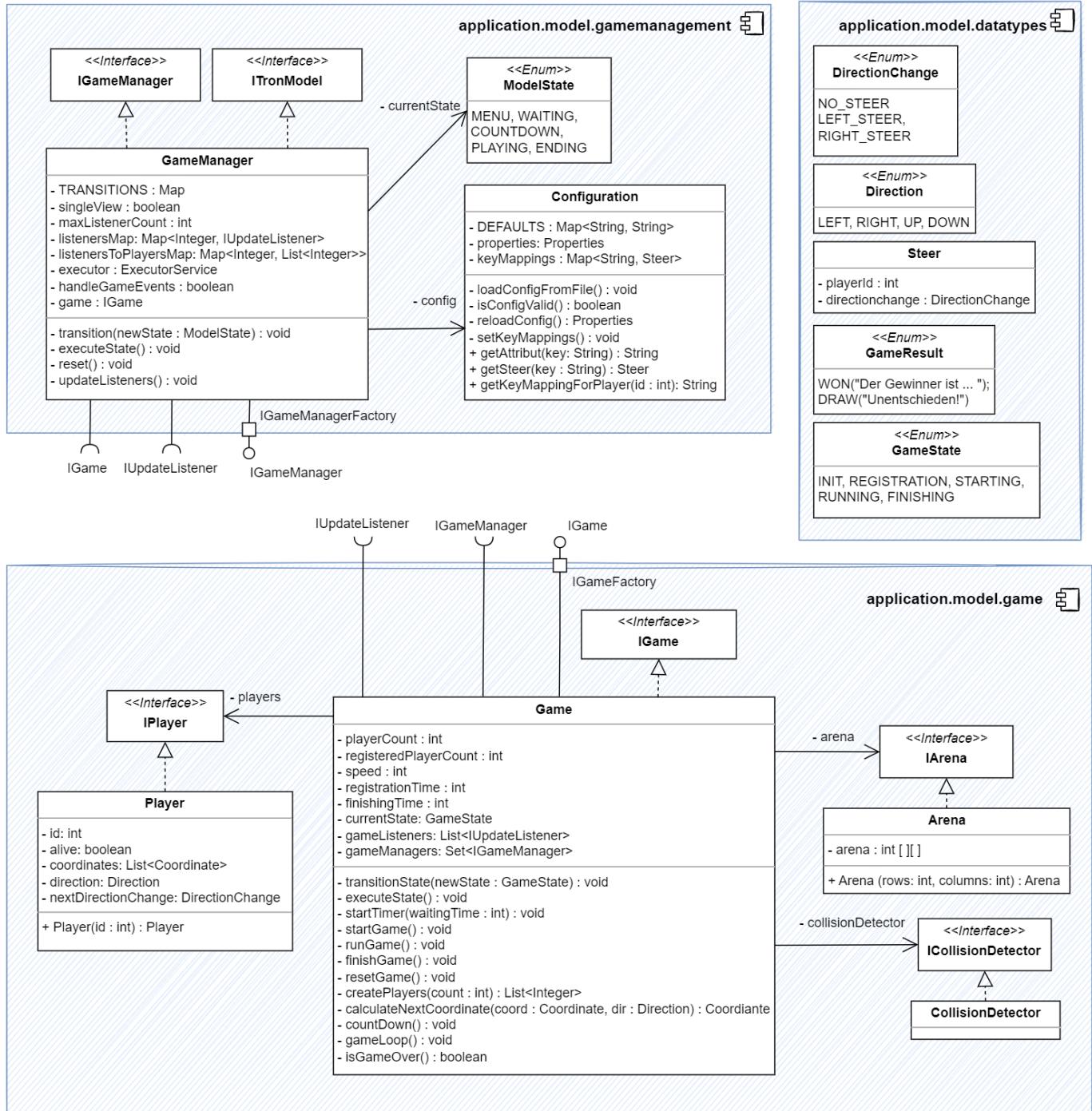
5.1 Ebene 1

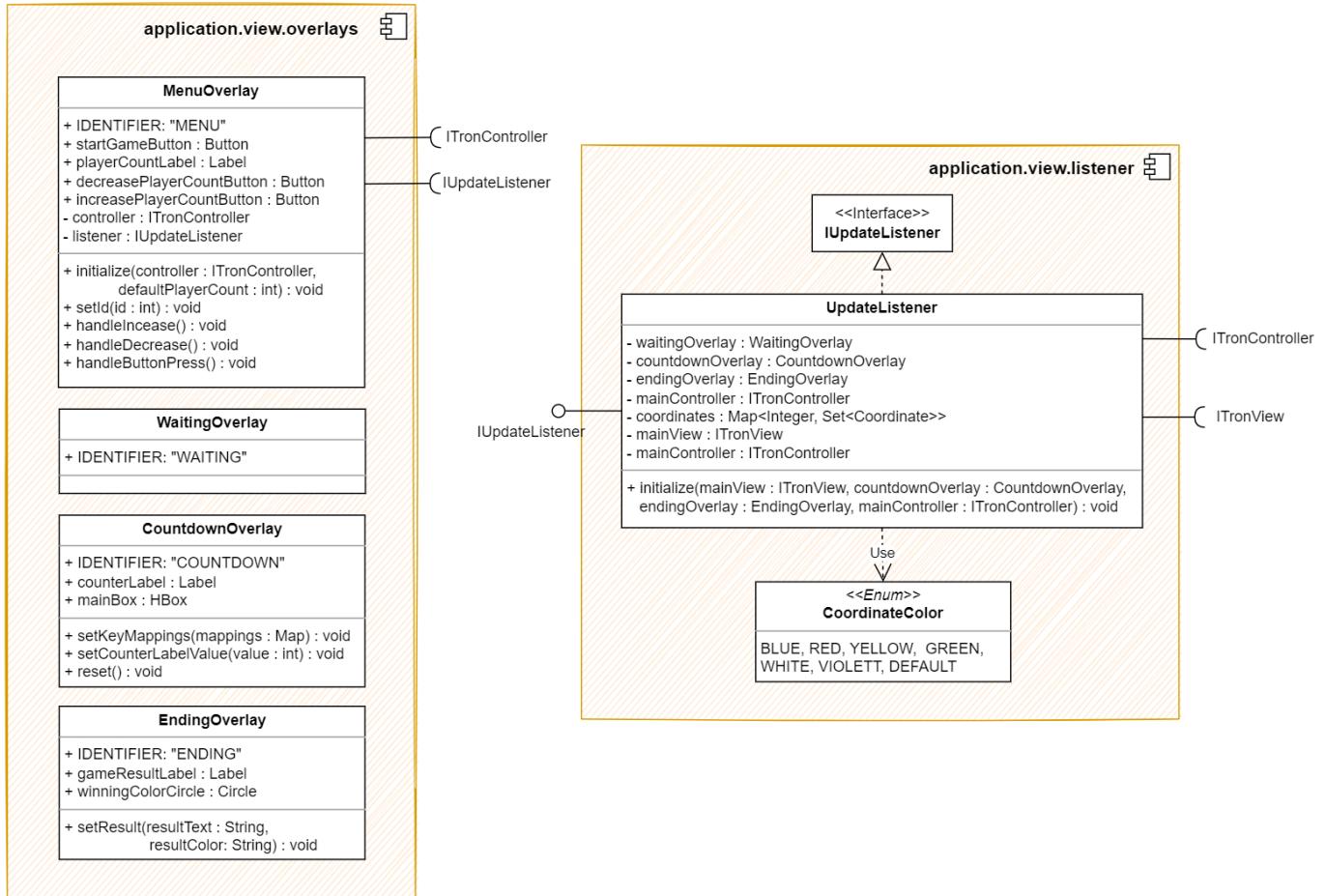


5.2 Ebene 2 : Application

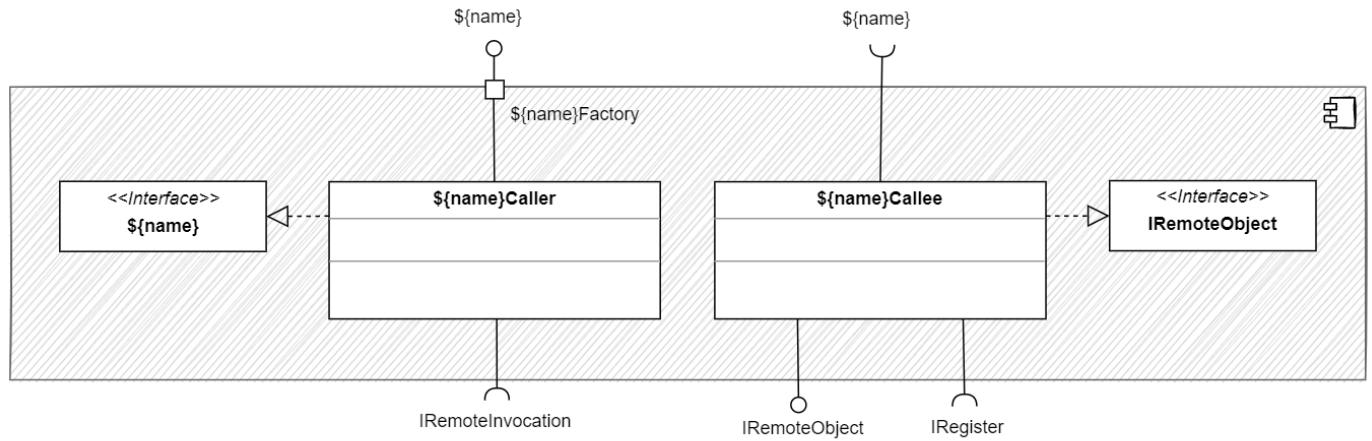


5.3 Ebene 3 : Application





5.4 Ebene 3 : ApplicationStub:

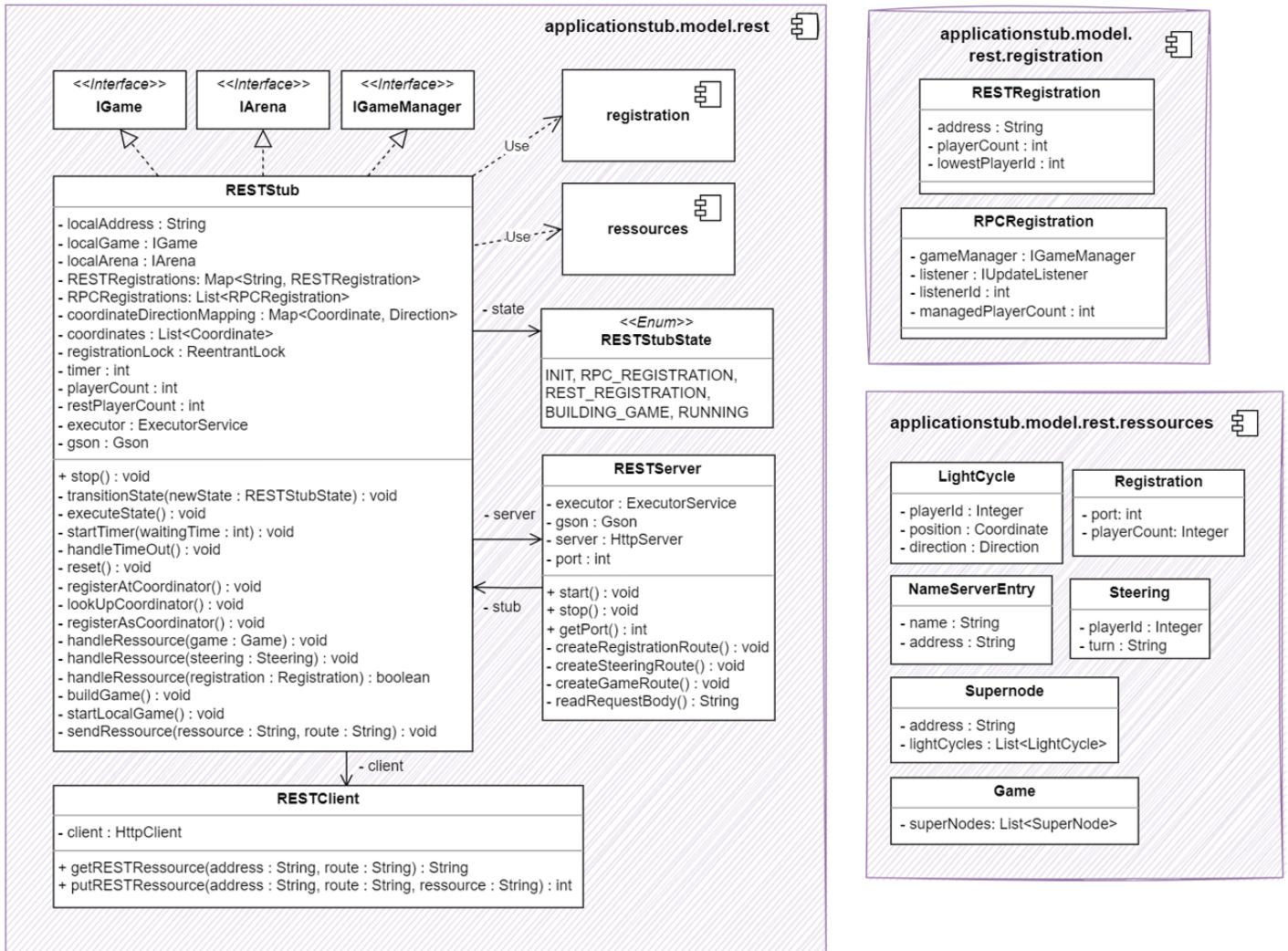


Für $name \in \{\text{IGameManager}, \text{IGame}, \text{IUpdateListener}\}$ gibt es im ApplicationStub eine Komponente der oben beschriebenen Form. Die Stubs verwenden darüber hinaus folgende Klassen:

- **RemoteId:** Stellt eine einzigartige ID für diesen Application Stub dar.
- **Service:** Enthält die Services, die Remote Objekte dieses Application Stubs bereitstellen können.

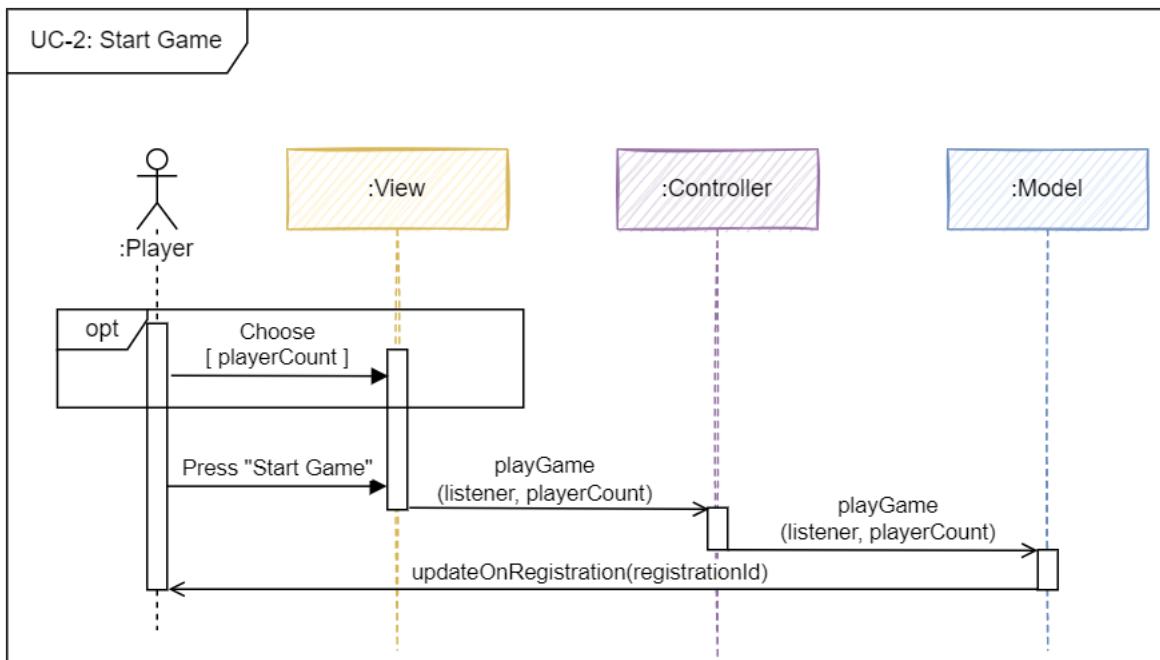
Caller-Objekte können darüber hinaus das ICaller-Interface implementieren, das lediglich dazu dient, die Remoted zu setzen.

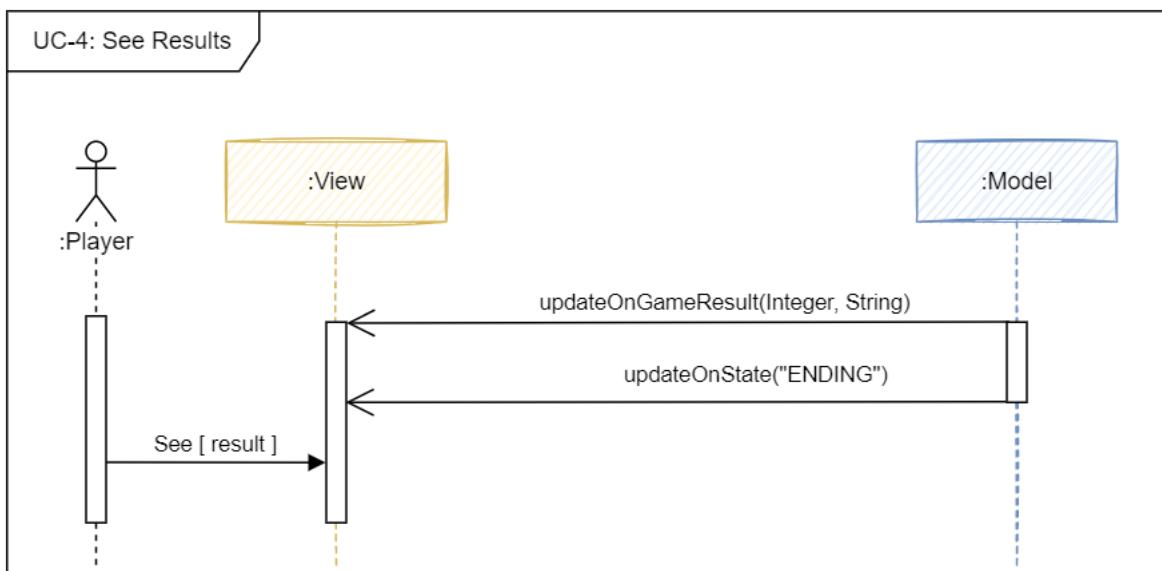
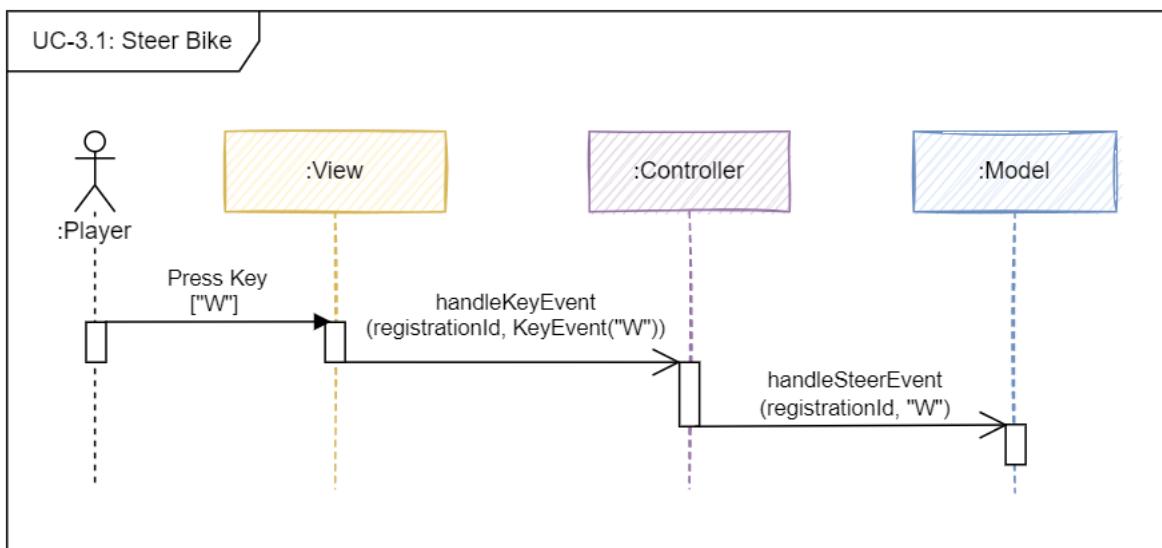
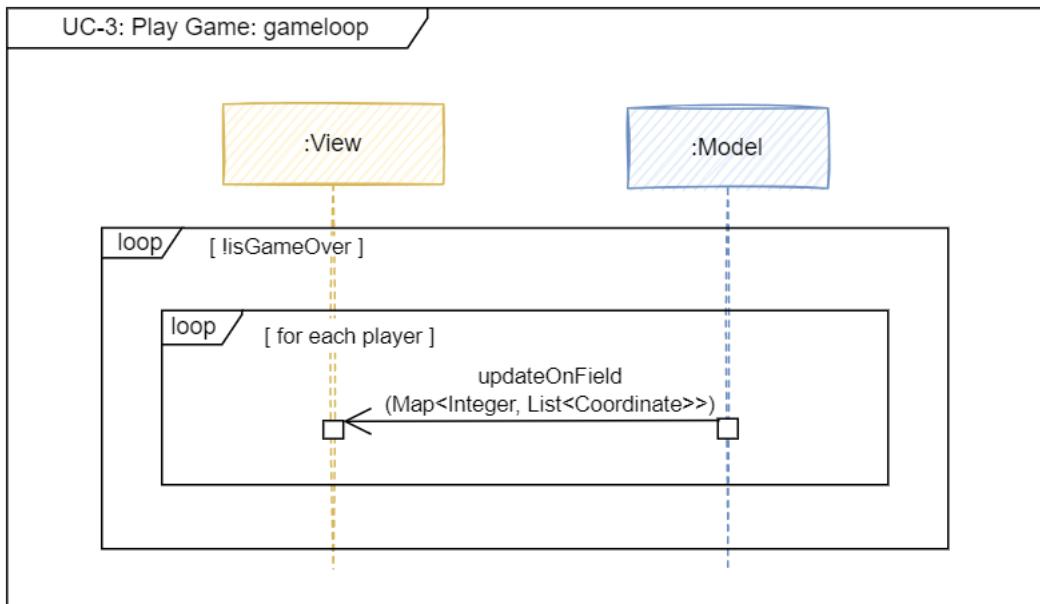
Zusätzlich befindet sich folgende rest-Komponente im ApplicationStub. Für REST siehe [Abschnitt 8.2.4](#).

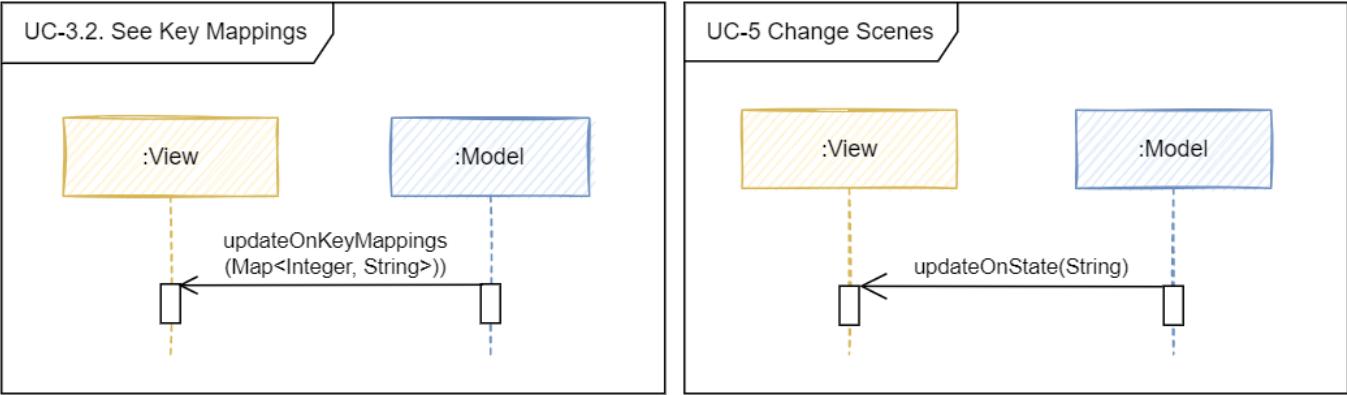


6. Laufzeitsicht

6.1 Use Cases auf MVC-Ebene



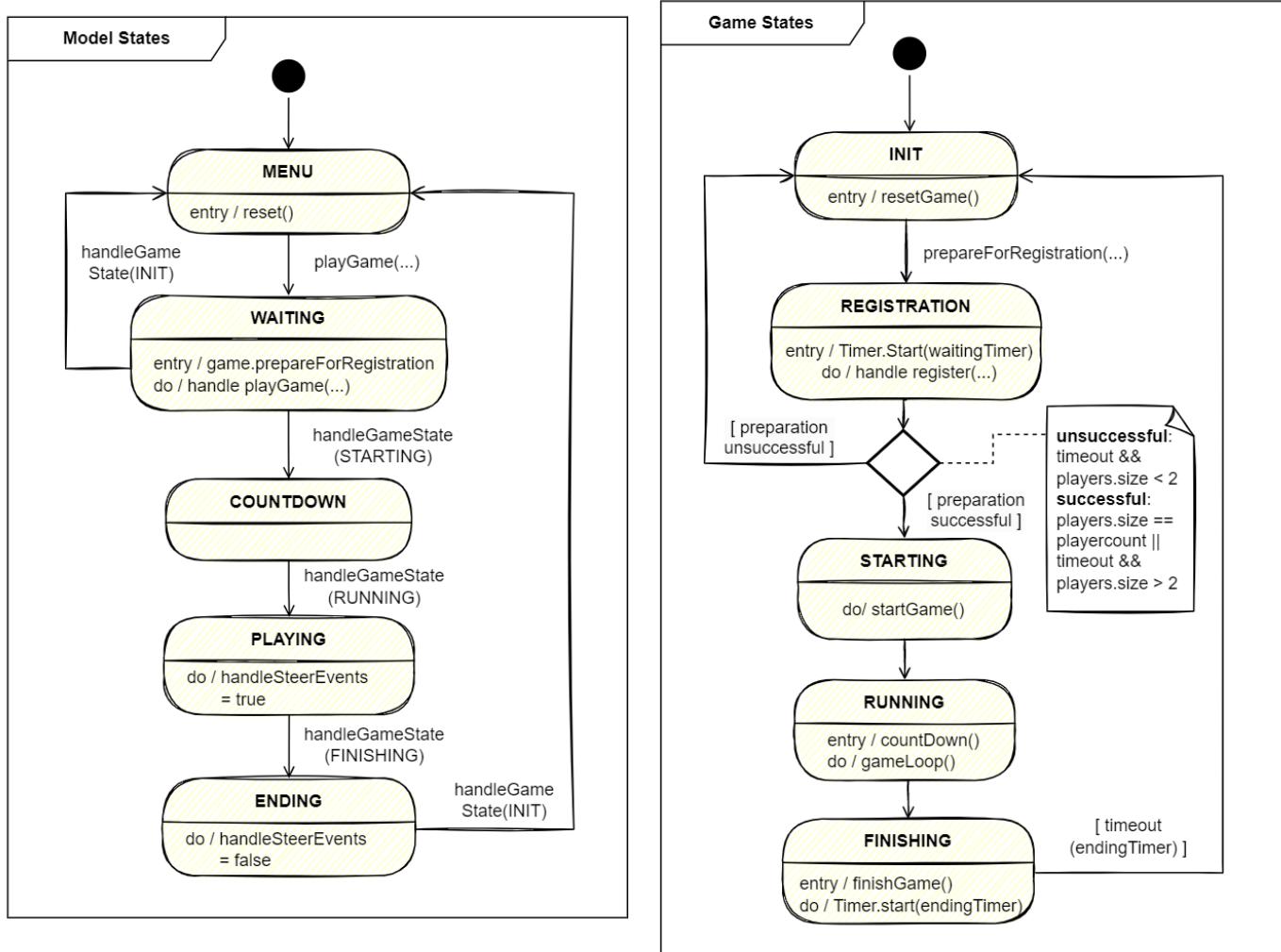




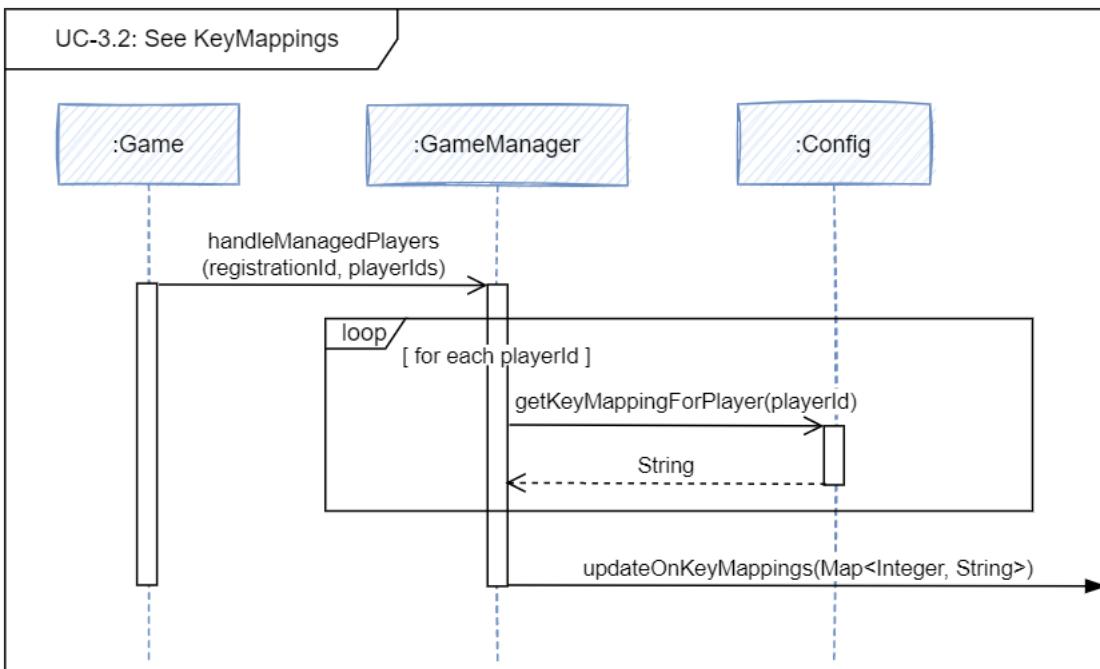
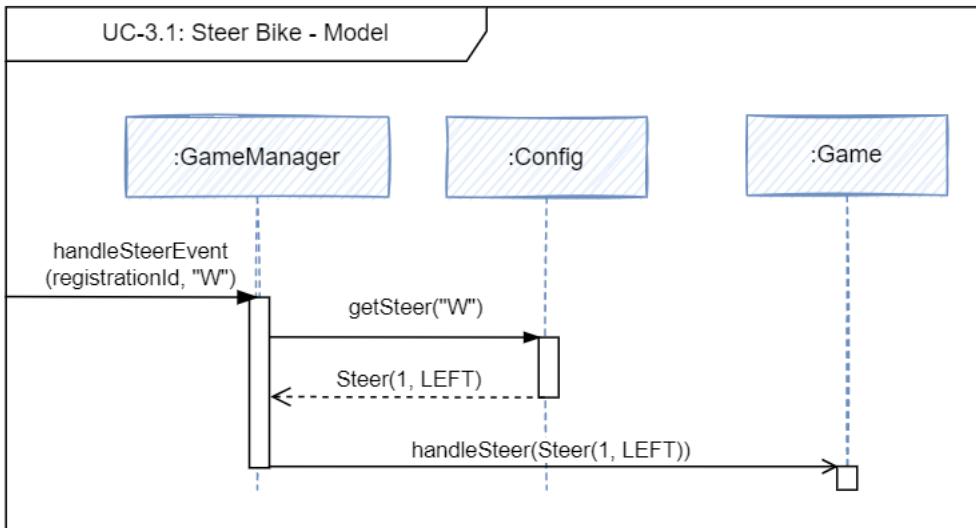
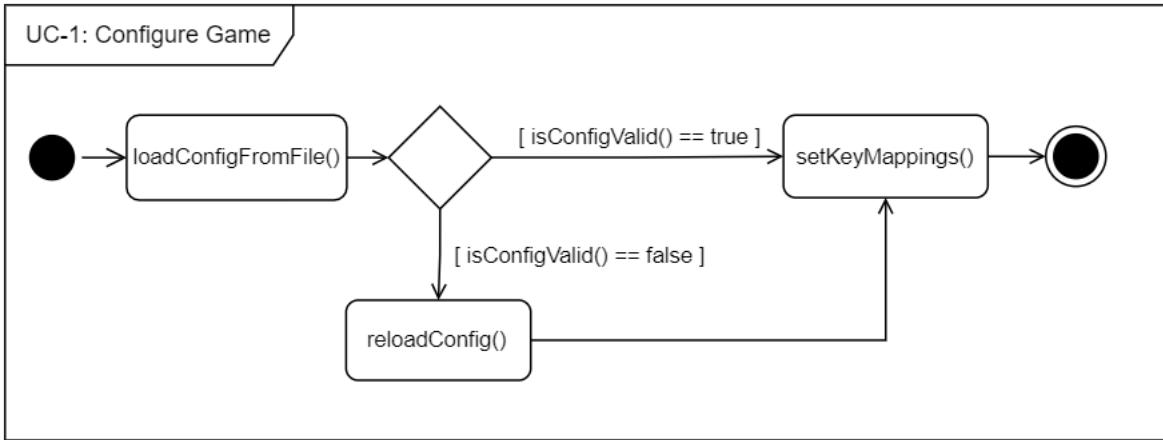
6.2 Model

6.2.1 States im Model

Die in den States beschriebenen Aktivitäten sind in Abschnitt [Activities im Model](#) näher beschrieben.

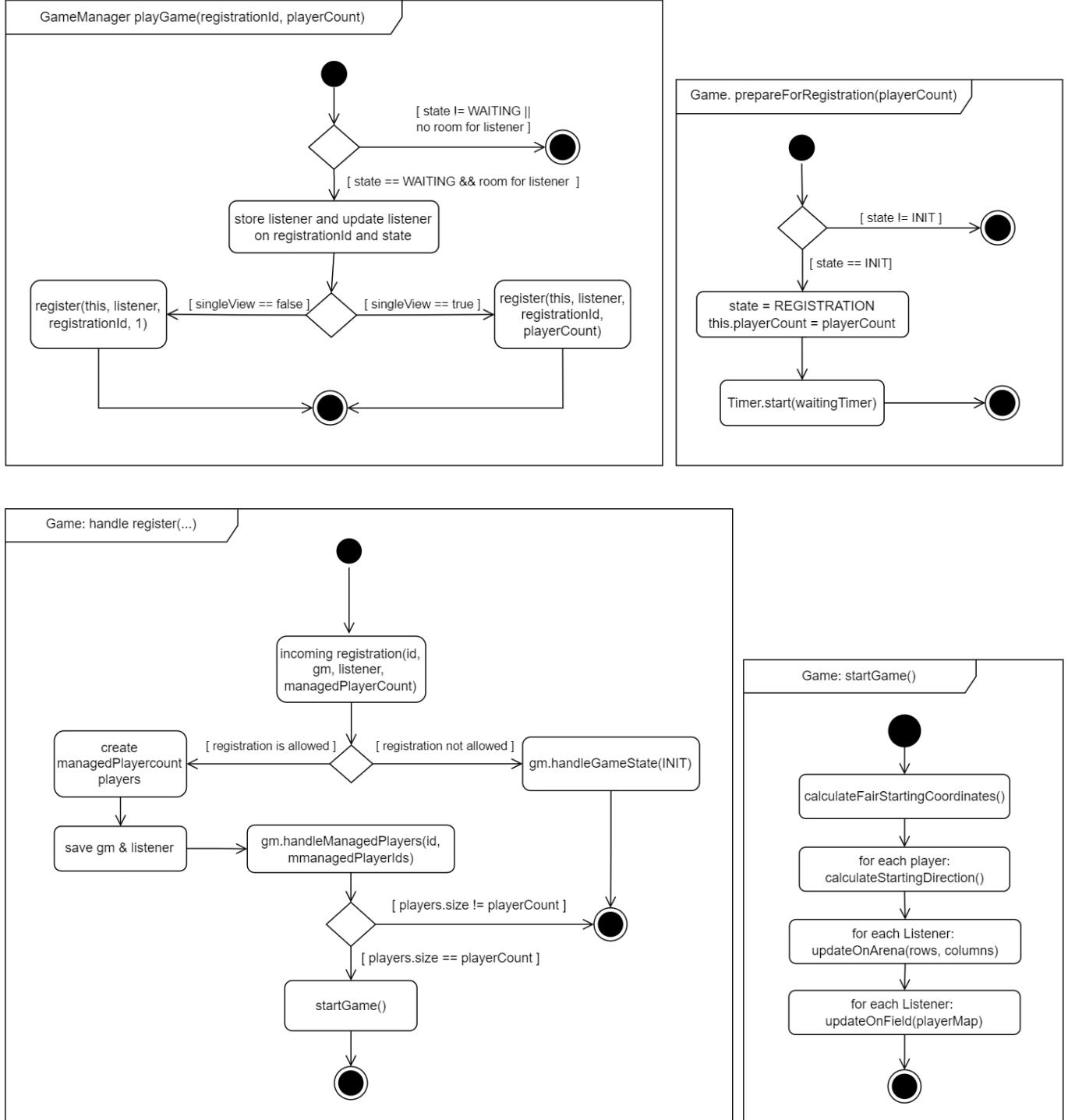


6.2.2 Details der Use Cases im Model

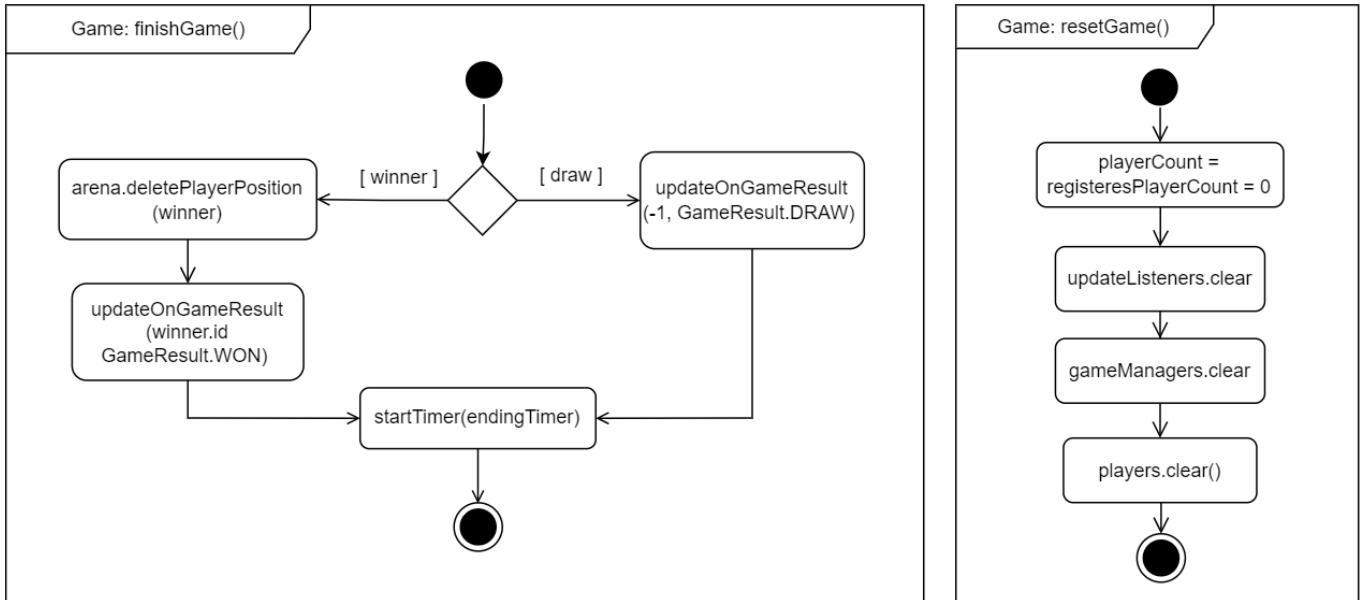


6.2.3 Activities im Model

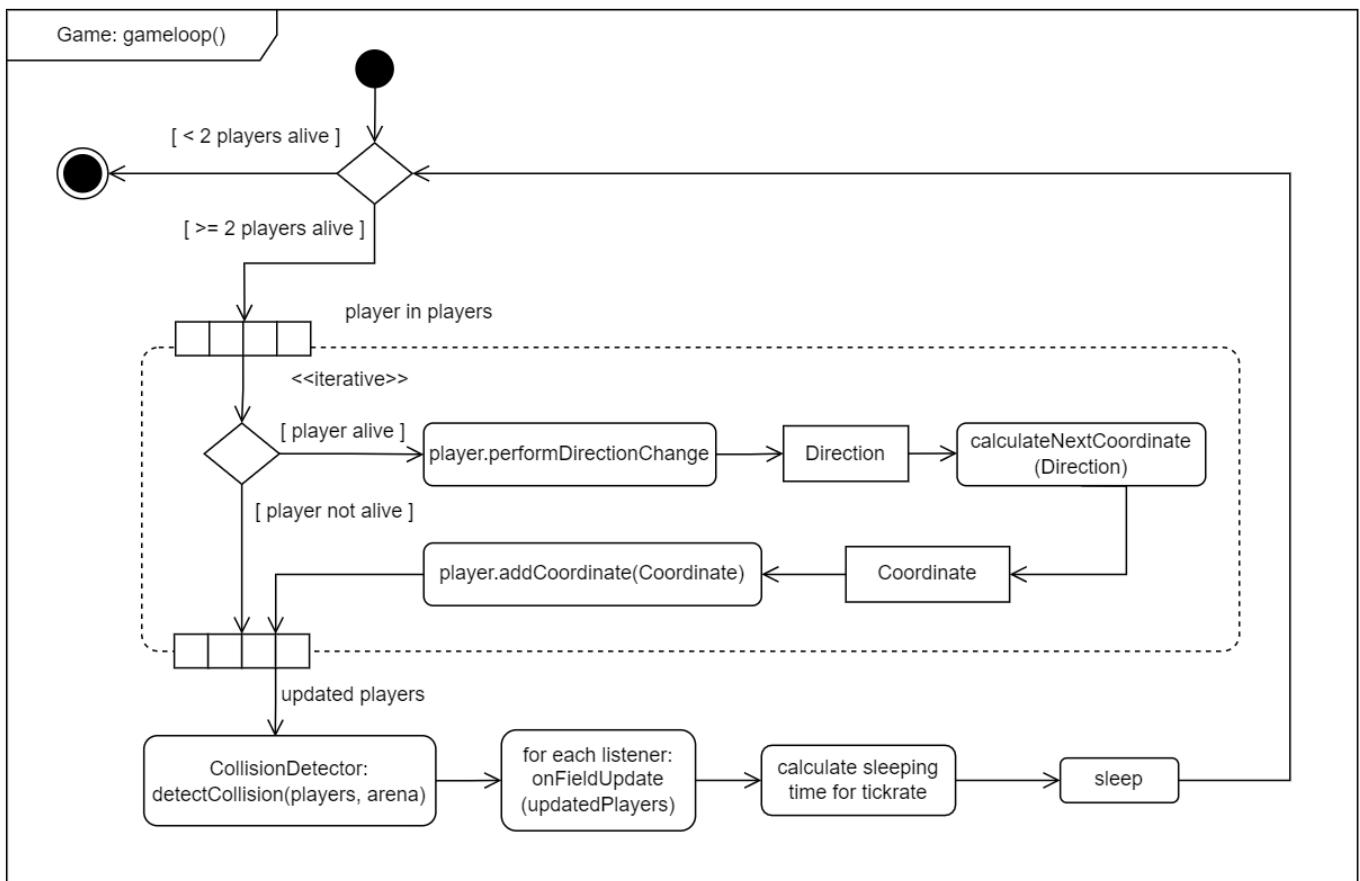
6.2.3.1 Start

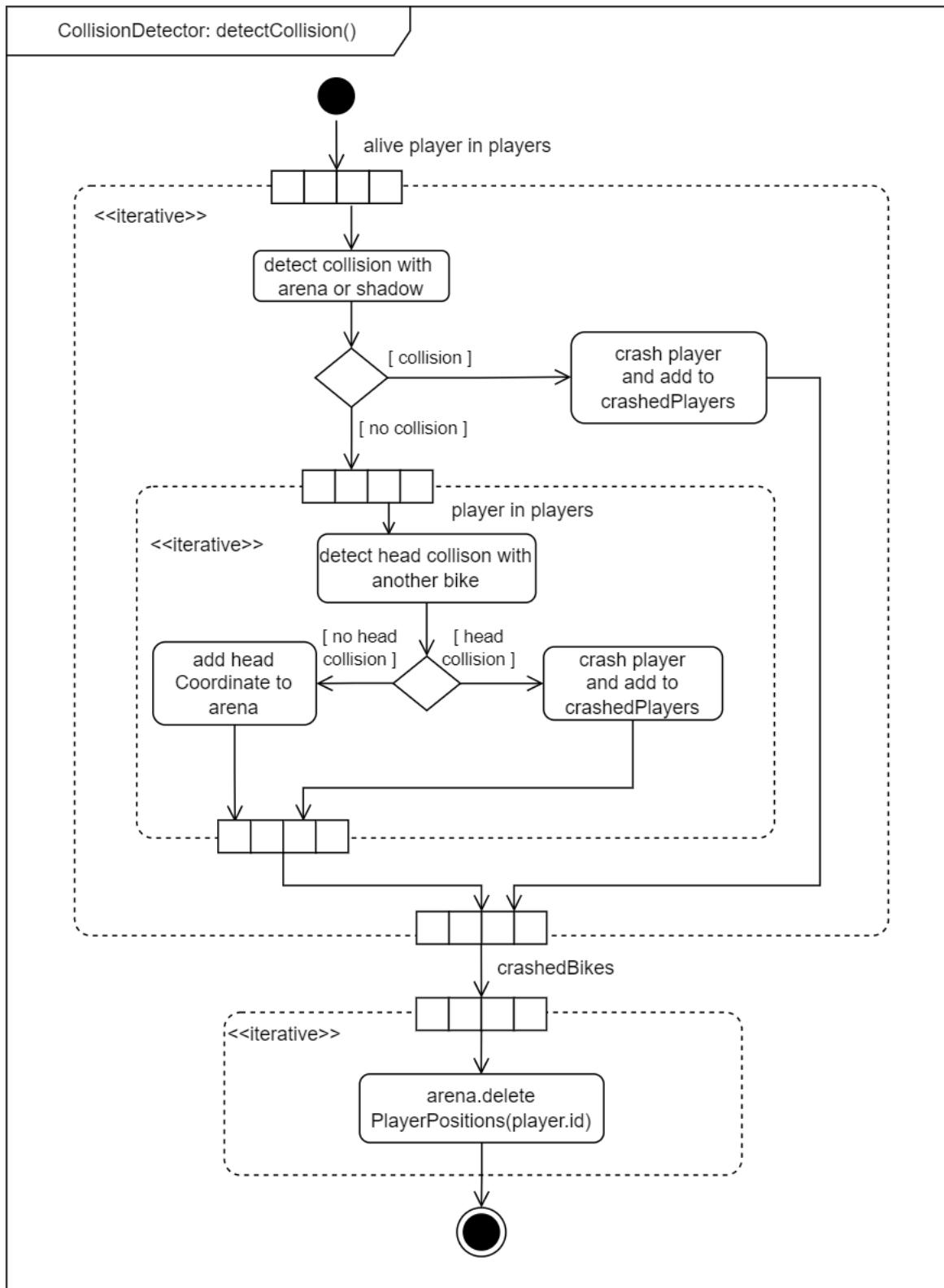


6.2.3.2 End

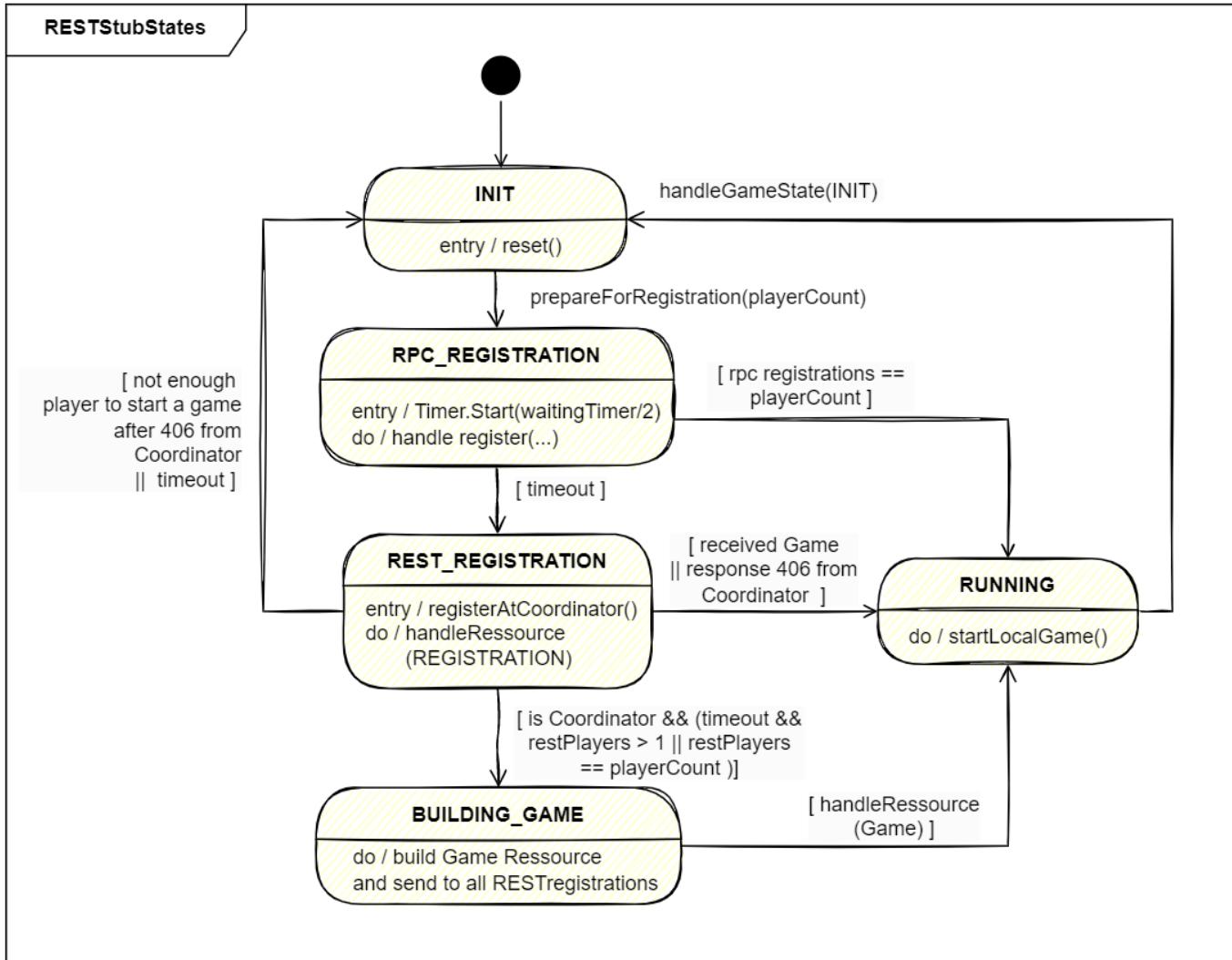


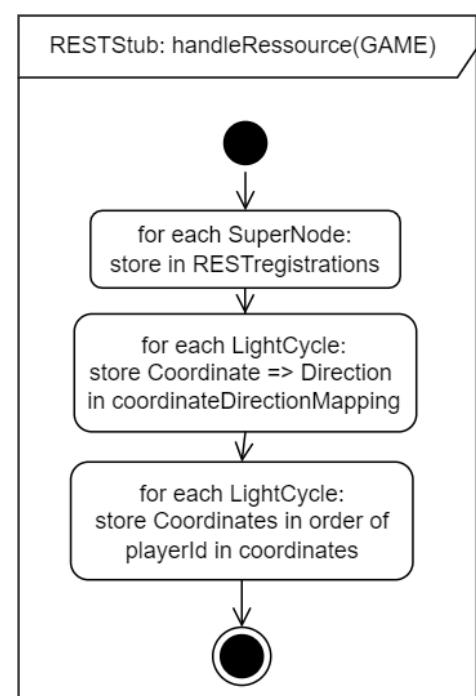
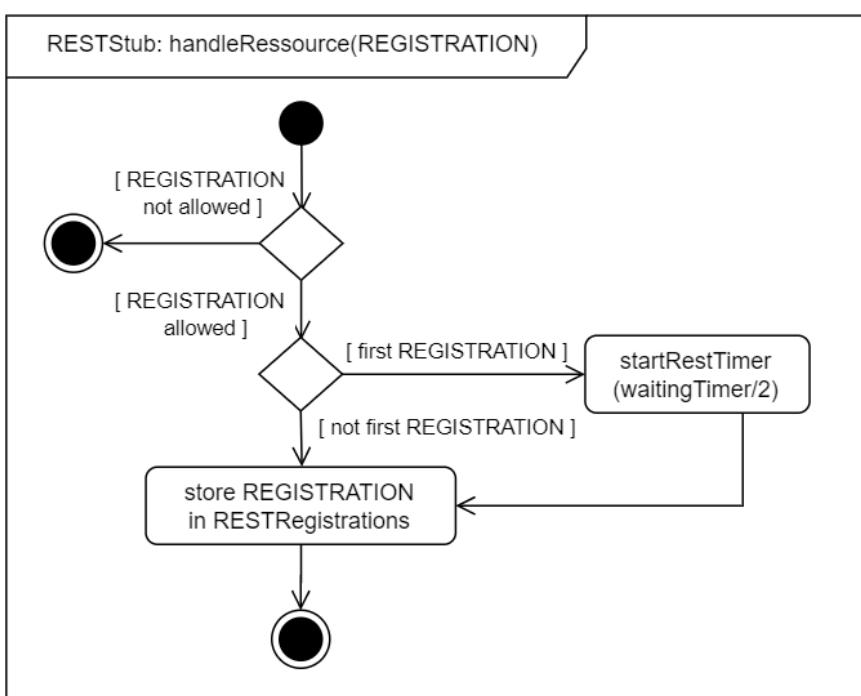
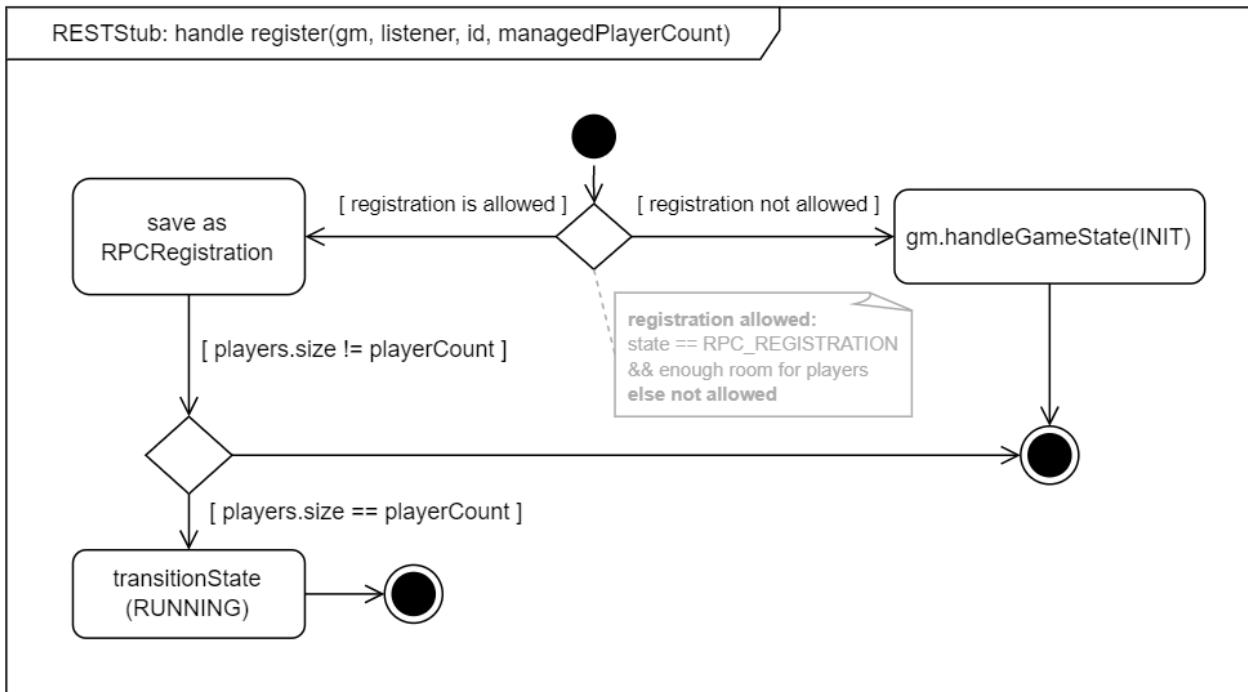
6.2.3.3 Play

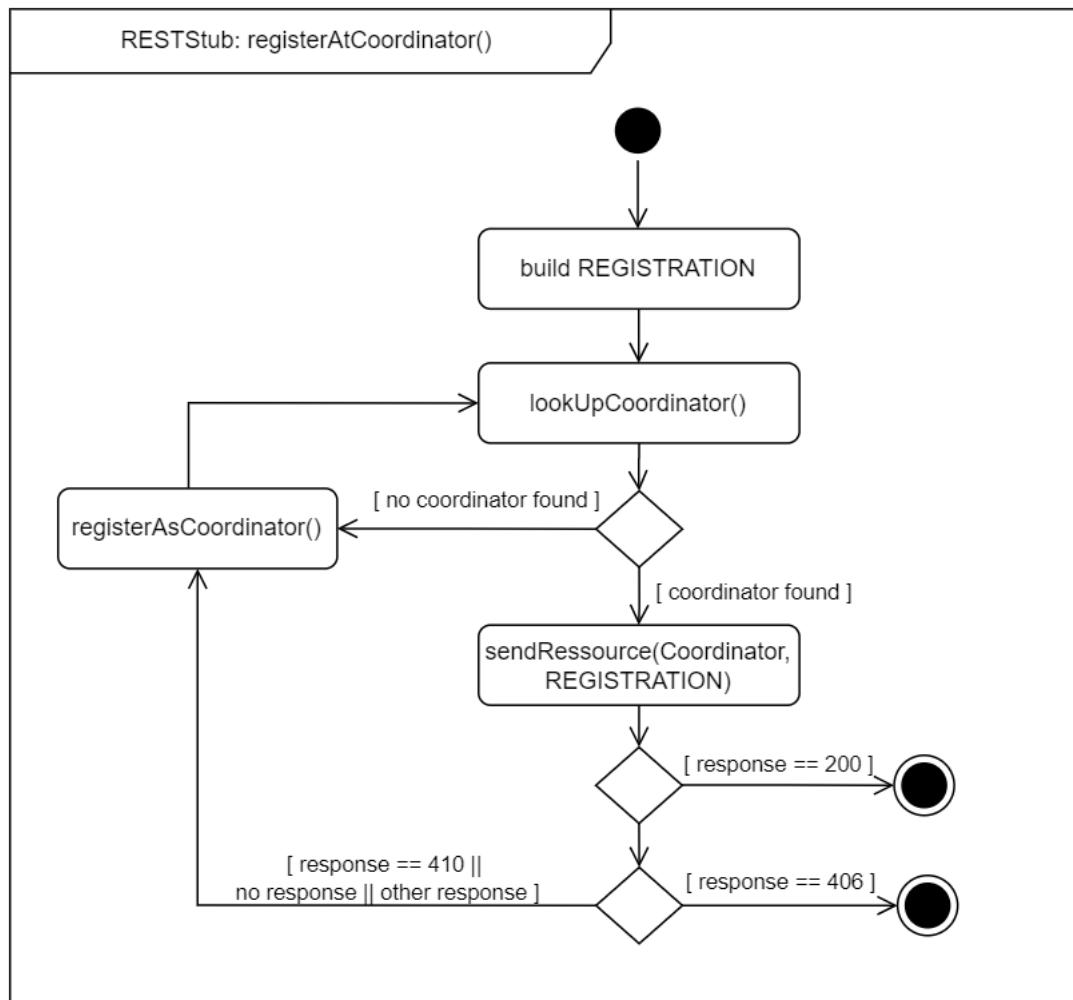
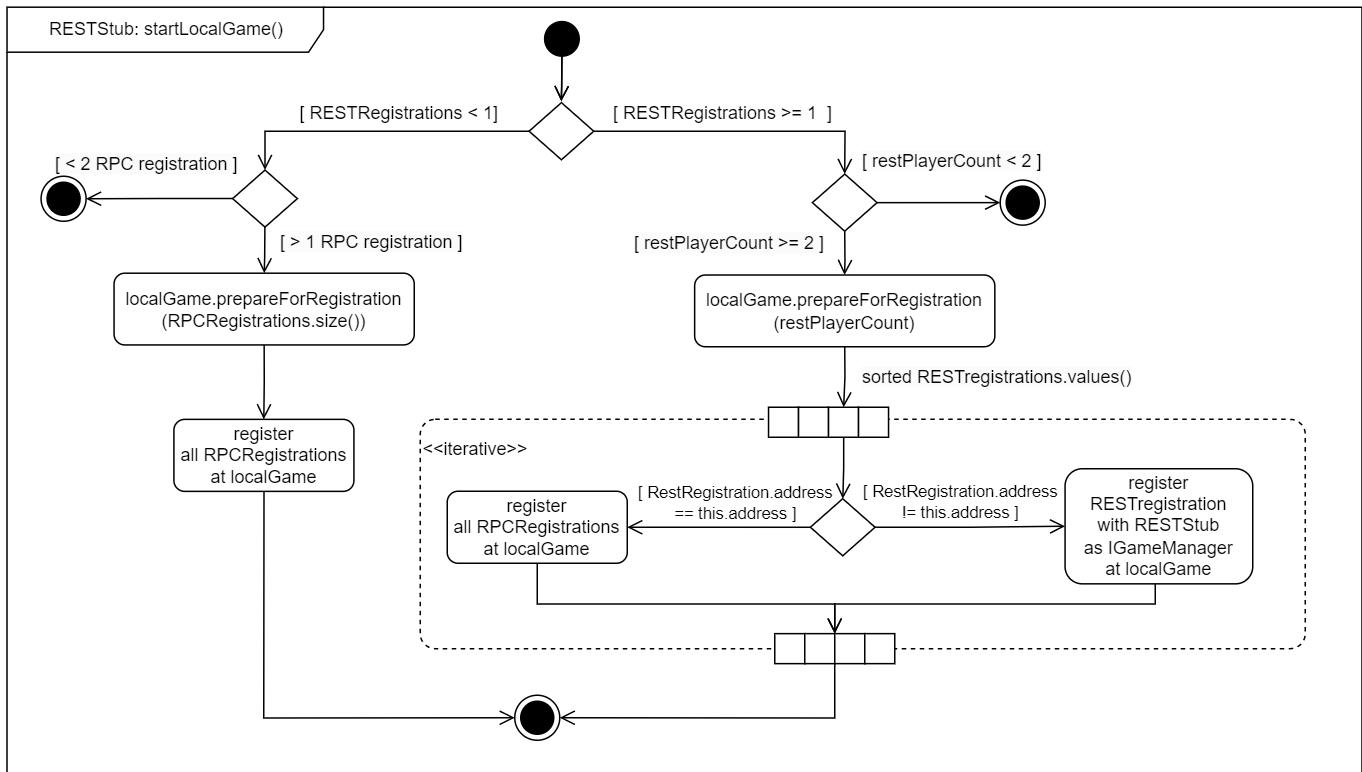




6.3 ApplicationStub : REST







7. Verteilungssicht

Es sind beliebige Verteilungen möglich.

8. Querschnittliche Konzepte

8.1 Application

8.1.1 Modus

Die Application kann in drei verschiedenen Modi gestartet werden: **LOCAL**, **RPC**, **REST**. Dies geschieht über die `tronConfig.properties` Datei.

Der Modus wird aus der Datei gelesen und ein entsprechendes Enum zum instanziieren von Objekten in Factories verwendet. Im RPC-Modus kommen die im `ApplicationStub` unter [Abschnitt 5.4](#) beschriebenen Caller- und Callee-Stubs zum Einsatz. Im REST-Modus wird der im `ApplicationStub` unter Abschnitt 5.4 beschriebene `RESTStub` sowie die in RPC verwendeten Stubs verwendet.

8.1.2 Registration-ID

Um zu ermöglichen, dass mehrere Views mit dem Model kommunizieren können, registriert sich die View als Listener beim Model und erhält anschließend eine Registration-ID vom Model. Kommunikation, die über den Controller ans Model geht, muss anschließend diese ID enthalten. Dadurch kann das Model speichern, welche Spieler zu welcher Registration-ID gehören, sodass nur berechtigte Tastenanschläge vom Model verarbeitet werden.

8.2 ApplicationStub

8.2.1 Remoteld

Der `ApplicationStub` kann verschiedene Services anbieten. In manchen Fällen kann es notwendig sein, den Service eines konkreten `ApplicationStubs` anzufragen. Dazu erhält jeder `ApplicationStub` eine **RemoteId**, über die er eindeutig identifiziert werden kann.

8.2.2 Namensraum

Zur Identifizierung der vom `ApplicationStub` angebotenen Services wird ein hierarchischer Namensraum verwendet. Jeder Service kann eindeutig über die Kombination aus `ServiceId` und `Remoteld` des anbietenden `ApplicationStubs` bestimmt werden.

8.2.3 Service Call Protokoll

Die Anfrage, einen angebotenen Service auszuführen, besteht aus folgenden Bestandteilen:

- **serviceId**: die Id des Services
- **intParameters** : Parameter des Services vom Typ int
- **stringParameters**: Parameter des Services vom Typ String

Services können nach folgenden Regeln angefragt werden:

Service	ServiceId	intParameters	stringParameters
PREPARE	0	playercount für das Spiel	keine
REGISTER	1	registrationId, managedPlayerCount	Remoteld des <code>IGameManager</code> Remote Objekts, Remoteld des <code>IUpdateListener</code> Remote Objekts
HANDLE_STEER	2	playerId, Ordinal des DirectionChange	keine
HANDLE_MANAGED_PLAYERS	3	registrationId, Liste mit managedPlayer ids	keine
HANDLE_GAME_STATE	4	Ordinal des GameState	keine
UPDATE_arena	5	Rows, Columns	keine
UPDATE_STATE	6	Ordinal des GameState	keine
UPDATE_START	7	keine	keine
UPDATE_RESULT	8	SpielerId des Gewinners, Ordinal des ResultTextes	keine
UPDATE_COUNTDOWN	9	CountDown value	keine
UPDATE_FIELD	10	PlayerCount, PlayerId, X, Y, ..., PlayerId, X, Y, ...	keine

8.2.4 Spiel mit den Implementierungen anderer Teams

Für das Zusammenspiel mit anderen Teams wird eine Synchronisation des Spiels mittels REST gewählt. Dazu wurde unter allen teilnehmenden Teams ein REST-Protokoll erarbeitet, das von jeder Implementierung umgesetzt werden muss. Daten die Configuration werden dabei nicht ausgetauscht. Sowohl Speed als auch Arenagröße müssen für ein synchrones Spiel vorher abgesprochen werden. Zentral in dem Protokoll ist, dass zunächst ein Coordinator bestimmt werden muss. Dies geschieht in Absprache mit den anderen Teams über einen Name Server, bei dem sich ein Coordinator als `tron.coordinator` anmeldet. Implementiert wird das Protokoll durch die rest-Komponente im Applicationstub, dargestellt in [Abschnitt 5.4](#).

Das REST-Protokoll ist zu finden unter: <https://gitlab.com/jessyvere/rest-protocol>

Der verwendete Name Server: <https://name-service.onrender.com/swagger/index.html>

9. Architekturentscheidungen

Entscheidung	Qualitätsmerkmale	Beschreibung
Komponenten nach MVC	Erweiterbarkeit, Wartbarkeit, Übertragbarkeit	Die Einführung vom MVC-Pattern soll die Bearbeitung an der Applikation vereinfachen und der Applikation eine verständliche Struktur geben.
Factory Method	Lose Kopplung, Erweiterbarkeit, Wartbarkeit	An vielen Stellen werden Factories verwendet, um die Objekterzeugung vom Rest der Implementierung löszulösen und eine Implementierung gegen Schnittstellen anstatt gegen konkrete Objekte zu unterstützen
Singleton Pattern	Lose Kopplung	Einige Objekte werden als Singleton realisiert, da sie an unterschiedlichen Stellen gebraucht werden und dies das erfüllen von Dependencies erleichtert.

10. Qualitätsanforderungen

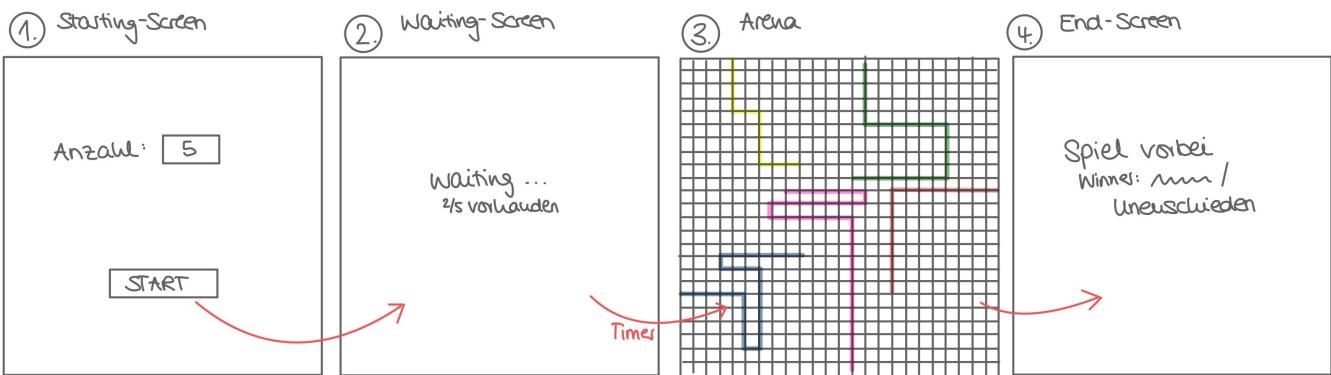
siehe Abschnitt 1.2.

11. Risiken und technische Schulden

Entscheidung	Beschreibung
Dependencies	Dependencies werden derzeit manuell über initialize-Methoden, Konstruktoren oder Singletons realisiert.
Singleton Pattern	Singletons werden an einigen Stellen verwendet, um das Realisieren von Dependencies zu erleichtern.
Manuelles Testen	Das Definieren einer Teststrategie mit automatisierten Tests wurde versäumt und erschwert zukünftiges Refactoring, Erweiterungen sowie die Sicherstellung der Funktionalität.

Anhang

Storyboard



Use Cases

UC-1: Configure Game

Akteur: Spieler

Ziel: Spiel nach seinen Wünschen konfigurieren

Auslöser: Öffnen der Config-File

Nachbedingungen: Neue Konfigurationsdaten sind gespeichert und werden bei Applikationsstart verwendet.

Standardfall:

- Der Benutzer bearbeitet die Daten der Config-Datei:
waitingTimer (Sekunden)
endingTimer (Sekunden)
defaultPlayerNumber (Ganzzahl zwischen 2-6)
speed (Ganzzahl zwischen 1-500)
height (Ganzzahl)
width (Ganzzahl)
rows (Ganzzahl)
columns (Ganzzahl)
controls p1-6 (bsp: W,A)
gameMode (LOCAL, NETWORK)
nameServer (IP:Port)
nameServerHost (TRUE, FALSE)
- Der Benutzer speichert die Datei.
- Der Benutzer startet das System.
- Das System lädt die Daten aus der Konfigurationsdatei.
- Das System überprüft die Daten der Konfigurationsdatei auf Fehler.
- Das System zeigt den Menu Screen an.

Fehlerfälle:

- Das System findet einen Fehler in der Konfigurationsdatei oder findet die Konfigurationsdatei nicht.
 - Das System erstellt eine neue .properties-Datei mit default Werten.

UC-2: Start Game

Akteur: Spieler

Ziel: Tron-Spiel spielen.

Vorbedingungen: Das System befindet sich im Starting-Screen.

Nachbedingungen: Es wird ein Tron-Game begonnen und angezeigt.

Standardfall:

- Das System zeigt den Starting Screen mit der defaultPlayerNumber und einen "Spiel starten" Button an.
- Der Spieler betätigt den Button.

3. Das System wechselt in den Waiting-Screen.
4. Das System initialisiert ein Game mit der defaultPlayerNumber.
5. Das System startet den Waiting-Timer.
6. Das System registriert die Spieler beim Game.
7. Das System berechnet die Startpositionen und die Startrichtung der Spieler.
8. Das System wechselt in den Arena-Screen.
9. Es beginnt UC-3: Play Game

Erweiterungsfälle:

- 7.a Nach Ablauf des Waiting-Timers konnte die Spieleranzahl nicht erreicht werden mit ≥ 2 Spieler.
 - 7.a.1 weiter im Standardfall Punkt 7.
- 2.a UC-2.1: Choose Player Number: Der Spieler wählt eine andere Spielerzahl.
 - 2.a.1 Der Spieler wählt eine Spielerzahl zwischen 2-6 aus einem Drop Down Menü.
 - 2.a.2 Das System verwendet den eingegebenen Wert anstelle des Defaultwertes.
 - 2.a.3 Weiter im Standardfall Punkt 2.

Fehlerfälle:

- 7.b Nach Ablauf des Waiting-Timers konnte die Spieleranzahl nicht erreicht werden mit < 2 Spieler.
 - 7.b.1 Das System setzt die Spieldaten zurück.
 - 7.b.2 Das System kehrt zum Starting-Screen zurück.

UC-3: Play Game

Akteur: Spieler

Ziel: Gewinnen

Vorbedingungen: UC-2: Spiel Starten erfolgreich abgeschlossen.

Nachbedingungen: Das Spiel ist mit "Gewinner" oder "unentschieden" geendet.

Standardfall:

1. Das System zeigt das Spielfeld in der konfigurierten Größe an.
2. Das System zeigt einen Countdown(3-2-1) an, der 3 Sekunden lang ist.
3. Das System zeigt während des Countdowns die Farben der Spieler und die Steuerung an.
4. Das System bewegt die Bikes stetig in die aktuelle Richtung in der konfigurierten Geschwindigkeit vorwärts.
5. Das System vergrößert den Schatten des Bikes mit jeder Vorwärtsbewegung.
6. Der / Die Mitspieler stirbt / sterben bei Kollision.
7. Das System zeigt die Schatten der gestorbenen Spieler nicht mehr an.
8. Das System beendet das Spiel, wenn nur noch einer oder kein Spieler mehr am Leben ist.
9. Ausführung UC-4: See Results

Erweiterungsfälle:

- 4.a UC-3.1: Steer Bike: Der Spieler steuert sein Bike durch Tasteneingaben
 - 4.a.1 Das System verarbeitet die Tasteneingabe abhängig von der Konfiguration des Spielers.
 - 4.a.2 Das System ändert die Richtung des Bikes des Spielers abhängig von der Taste.

UC-4: See Results

Akteur: Spieler

Ziel: Ergebnisse des letzten Spiels ansehen

Vorbedingungen: UC-3: Spiel Spielen erfolgreich abgeschlossen.

Nachbedingungen: Die Applikation ist in den Starting-Screen zurückgekehrt.\

Standardfall:

1. Das System startet den Ending-Timer.
2. Das System zeigt den Ending-Screen an, in dem das Ergebnis des Spiels ("Gewinner ist ..." oder "Unentschieden!") und die Farbe des Gewinners, falls es einen gibt, angezeigt wird.
3. Das System wechselt zurück zum Starting Screen, wenn der konfigurierte End-Timer angelaufen ist.

Middleware

1. Einführung und Ziele

In diesem Dokument wird der Entwurf einer Middleware mit Remote Method Invocation (RMI) beschrieben.

1.1 Aufgabenstellung

Die Middleware bildet eine Zwischenschicht zwischen der Applikation und dem Betriebssystem. Sie soll den Aufruf von Funktionen zwischen voneinander unabhängigen Nodes ermöglichen:

- Sie bietet der Applikation eine einfache Schnittstelle zum Aufrufen von Funktionen.
- Sie verpackt diese Aufrufe in Nachrichten und versendet sie über das Netzwerk.
- Sie entpackt Nachrichten aus dem Netzwerk und gibt sie an die Applikation weiter.
- Sie kommuniziert dafür mit der Middleware anderer Nodes, d.h. sie kann andere Nodes finden und mit ihnen kommunizieren.

1.2 Qualitätsziele

Ziel	Beschreibung
Offenheit	Anbieten von einfachen und offenen Schnittstellen
Skalierbarkeit	<ul style="list-style-type: none">• Größenskalierbarkeit: Es müssen sich 2-6 Nodes beteiligen können.• Geographische Skalierbarkeit: Die Anwendung läuft in einem LAN beim Kunden (Raum 7.85)• Administrative Skalierbarkeit: Es gibt eine administrative Domäne.
Transparenz	<ul style="list-style-type: none">• Access: Die Applikation und die Spieler merken nicht, ob ein Methodenaufruf lokal oder remote ausgeführt wird.• Location: Weder Nutzer noch Anwendung wissen, mit welchem Computer sie sprechen (keine Eingabe von IP oder ähnliches).• Relocation: Im Betrieb nicht zu erwarten.• Migration, Replication, Concurrency: Für Kunden irrelevant: Für Kunden irrelevant.• Failure: Dem Entwicklungsteam überlassen.

1.3 Stakeholders

Rolle	Kontakt	Erwartungen
Dozent / Kunde	Martin Becke: martin.becke@haw-hamburg.de	Wohldefinierte Schnittstellen, Lernfortschritt der Entwickler
Entwickler	Sandra: sandra.koenig@haw-hamburg.de Inken: inkendulige@haw-hamburg.de Majid: majid.moussaadoyi@haw-hamburg.de	Stabile Anwendung, Anforderungen an Middleware verstehen und umsetzen
Spieler	Teilnehmer des Moduls VS WiSe22/23	Kriegt nicht mit, dass es eine Middleware gibt.

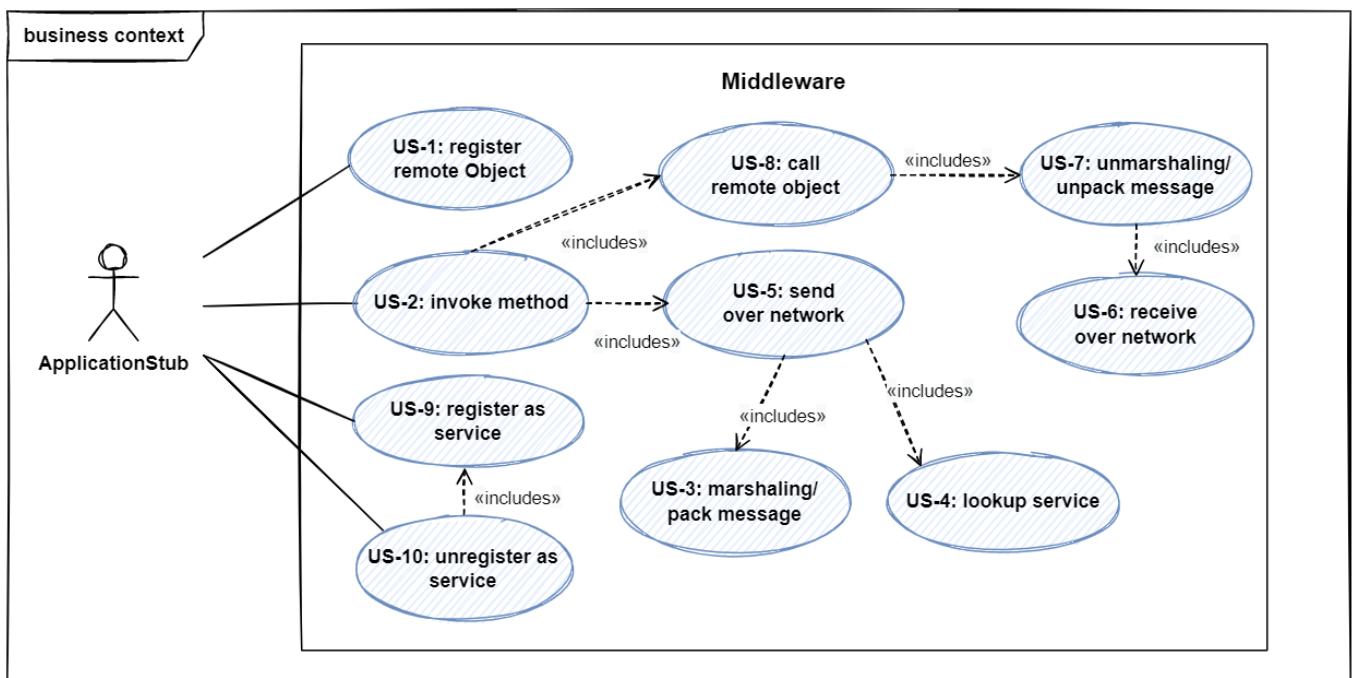
2. Randbedingungen

Technische Randbedingung	Beschreibung
Java in der Version 17	Zur Implementierung wird Java verwendet, da das ganze Team die Sprache beherrscht. Die Version muss zum Image der Rechner im Raum 7.65 passen. Es wird Java in der Version 17 verwendet, da es sich um die neueste LTS-Version handelt.
Kommunikation	Die Kommunikation der Middleware erfolgt transient über RMI.

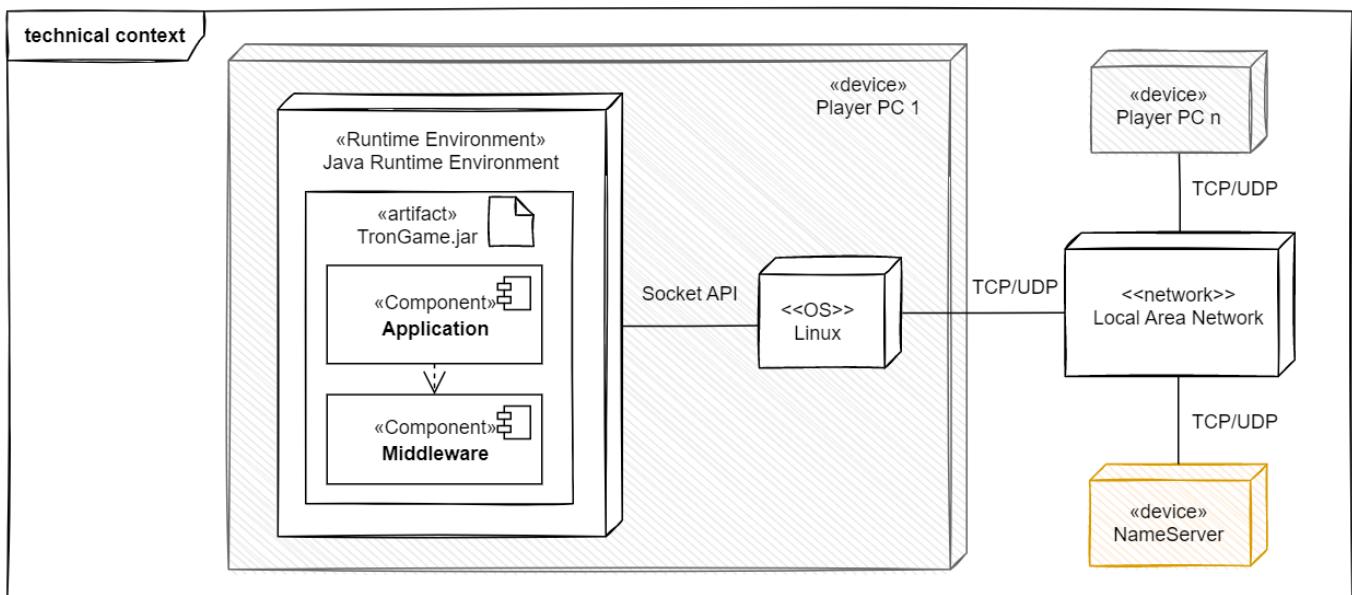
Technische Randbedingung	Beschreibung
MTU	Es kann sich auf eine MTU von 1500 Byte verlassen werden.
Konventionen	Beschreibung
Dokumentation	Gliederung nach dem deutschen arc42-Template, um Struktur zu wahren.
Sprache	Die Dokumentation erfolgt auf deutsch, während die Diagramme auf Englisch gehalten werden, um die Umsetzung in (englischen) Code zu erleichtern.

3. Kontextabgrenzung

3.1 Business Kontext



3.2 Technischer Kontext



4. Lösungsstrategie

Aus den Use Cases ergeben sich folgende benötigte Objekte, denen in folgenden Abschnitten Funktionen zugeordnet werden.

Objekt	Erklärung
IRemoteInvocation (ClientStub)	Ermöglicht den Aufruf von Services über die Middleware.
InvocationType {RELIABLE, UNRELIABLE}	Stellt die Art der Invocation dar und wird von IRemoteInvocation definiert.
Marshaller (ClientStub)	Nimmt Invoke-Aufrufe entgegen und verpackt sie in Nachrichten. Führt LookUps am NamingService durch.
ISender (ClientStub)	Nimmt Nachrichten vom Marshaller entgegen und schickt sie ins Netzwerk.
Protocol {TCP, UDP}	Stellt den zu verwendenden Protocol-Typen dar und wird vom ISender definiert.
IReceiver (ServerStub)	Lauscht auf einem Port und nimmt Nachrichten entgegen, die an den Unmarshaller weitergereicht werden.
IUnmarshaller (ServerStub)	Nimmt Nachrichten entgegen und entpackt sie zu MethodCalls. Kennt Remote Objekte und ruft MethodCall auf ihnen auf.
IRemoteObject	Remote Objekte können einen Service ausführen. Sie sind dem Unmarshaller bekannt.
IRegister	Ermöglicht das registrieren von Remote Objekten in der Middleware.
INamingService	Verwaltet Namen und dazugehörige Adressen. Ermöglicht Registrierung und LookUp von Services.
NameResolver	Realisiert die Namensauflösung (LookUp)
NameServer	Bietet ein zentrales Register für Services an, die nachgeschlagen werden können.

4.2.1 IRemoteInvocation

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
2	<pre>invoke(remoteId: String, serviceId: int, type: InvocationType, parameters: int[], stringParameters : String[])</pre>	IRemoteInvocation(ClientStub)	ClientStub wurde erstellt	Die Methode wird durch ein Remote-Object ausgeführt	Ruft eine Methode auf einem einem Remote-Object auf.	-

4.2.2 Marshaller

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
3,4	<pre>runInvocationTaskHandler() : void</pre>	Marshaller	-	Ein Taskhandler bearbeitet InvocationTasks	Der Aufruf der <code>invoke</code> Methode legt InvocationTasks in einer Queue ab, die ein Taskhandler der Reihe nach bearbeitet. Verwendet <code>marshal()</code> und <code>lookUp()</code> .	-
3	<pre>marshal(call : ServiceCall) : byte[]</pre>	Marshaller	-	-	Verpackt den Aufruf eines Services in ein byte Array.	-
4	<pre>lookUp(task : InvocationTask) : InetSocketAddress</pre>	Marshaller	-	-	Befragt den Naming Service nach der für den InvocationTask benötigten Adresse.	-

4.2.3 ISender & Sender

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
5	<pre>send(message: byte[], address: InetSocketAddress, protocol: Protocol) : void</pre>	ISender(ClientStub)	Der Adressat ist bekannt	Methodenaufruf wurde ans Netzwerk weitergereicht	Der Sender öffnet einen dem Protokoll entsprechenden Socket an den angebenen Adressaten und schickt die Nachricht über den Socket.	Wenn der Adressat nicht erreicht wird, wird die Nachricht verworfen.

4.2.4 IReceiver & Receiver

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
6	<code>start() : void</code>	IReceiver	-	Receiver hat jeweils einen lauschenden TCP- und UDP-Socket	Startet den Receiver (lauschende Sockets werden geöffnet). Verwendet dazu <code>createReceiverSockets()</code> , <code>startTcpReceiver()</code> und <code>startUdpReceiver()</code> .	-
6	<code>stop() : void</code>	IReceiver	-	Die Sockets sind geschlossen	Beendet den Receiver (lauschende Sockets werden geschlossen).	-
6	<code>createReceiverSockets() : void</code>	Receiver	-	Die Sockets sind erstellt und an eine InetSocketAddress gebunden	Es werden zuhörende Sockets mit zufälligem Port erstellt. UDP und TCP lauschen auf dem selben Port.	-
6	<code>startTcpReceiver() : void</code>	Receiver	-	TCP-Socket wartet auf eingehende Verbindungen	Es wird ein Thread gestartet, der auf eingehende Verbindungen lauscht.	-
6	<code>startTcpSocketHandler() : void</code>	Receiver	-	Ein Handler bearbeitet die Sockets eingehender Verbindungen	Es wird ein Thread gestartet, der die durch eingehende Verbindungen entstandenen Sockets des Tcp ServerSockets bearbeitet.	Bei Socket Error wird die Bearbeitung abgebrochen und die eingegangene Nachricht ignoriert.
6	<code>startUdpReceiver() : void</code>	Receiver	-	UDP-Socket bearbeitet eingehende Datagramme	Es wird ein Thread gestartet, der die eingehenden Udp Datagramme bearbeitet.	Bei Socket Error wird die Bearbeitung abgebrochen und die eingegangene Nachricht ignoriert.

4.2.5 IUnmarshaller & Unmarshaller

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
7,8	<pre>addToQueue(message : byte[]) : void</pre>	IUnmarshaller	-	Message ist in einer Queue zum Bearbeiten hinterlegt.	Fügen die übergebene Message in die Queue des Unmarshalls.	-

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
7,8	<code>stop() : void</code>	IUnmarshaller	-	Der Unmarshaller Thread ist gestoppt.	Stoppt den Unmarshaller, der wiederum den Receiver stoppt.	-
7,8	<code>startServiceCallHandler() : void</code>	Unmarshaller	-	Messages in der Queue werden bearbeitet.	Startet einen Thread, der die Nachrichten in der Queue bearbeitet. Verwendet <code>unmarshal</code> und <code>callOnRemoteObject</code>	-
7	<code>unmarshal(message : byte[]) : ServiceCall</code>	Unmarshaller	-	-	Entpackt eine Nachricht.	-
8	<code>callOnRemoteObject(call : ServiceCall) : void</code>	Unmarshaller	-	Der Service wurde vom Remote Object ausgeführt.	Ruft einen Service auf einem Remote Objekt auf.	Ist das Remote Objekt unbekannt, oder ist der Service dem Remote Objekt unbekannt, wird der Aufruf abgebrochen.

4.2.6 IRemoteObject

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
8	<code>call(serviceId, intParameters : int[], stringParameters : String[]) : void</code>	IRemoteObject	Remote-Object muss registriert sein	Methode wurde vom Remote Object ausgeführt	Ein Service wird auf einem Remote Object aufgerufen.	Sind die Parameter fehlerhaft oder die Service ID unbekannt, bricht das Remote-Object den Aufruf ab.

4.2.6 IRegister

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
1	<code>registerRemoteObject(serviceId: int, remoteId : String, remoteObject : IRemoteObject) : void</code>	IRegister	-	Das Remote Objekt wurde mit der Service ID im Register hinterlegt	Registeriert ein Remote Object beim ServerStub, damit es von diesem aufgerufen werden kann.	-

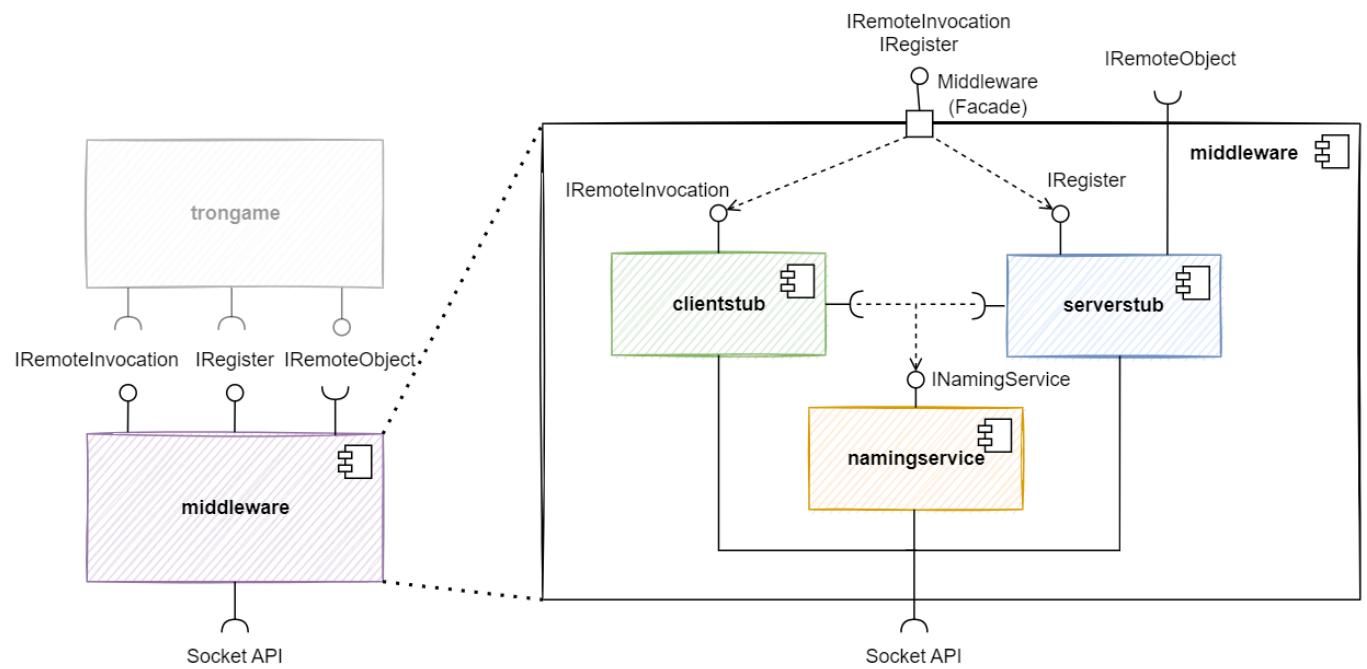
4.2.6 INamingService, NameResolver, NameServer

UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
4	<code>lookUpService(remoteId: String, serviceId: int) : String</code>	INamingService	-	Der ClientStub kennt die Informationen des Services (IP:Port, remoteld)	Gibt die Adresse eines Services zurück.	-

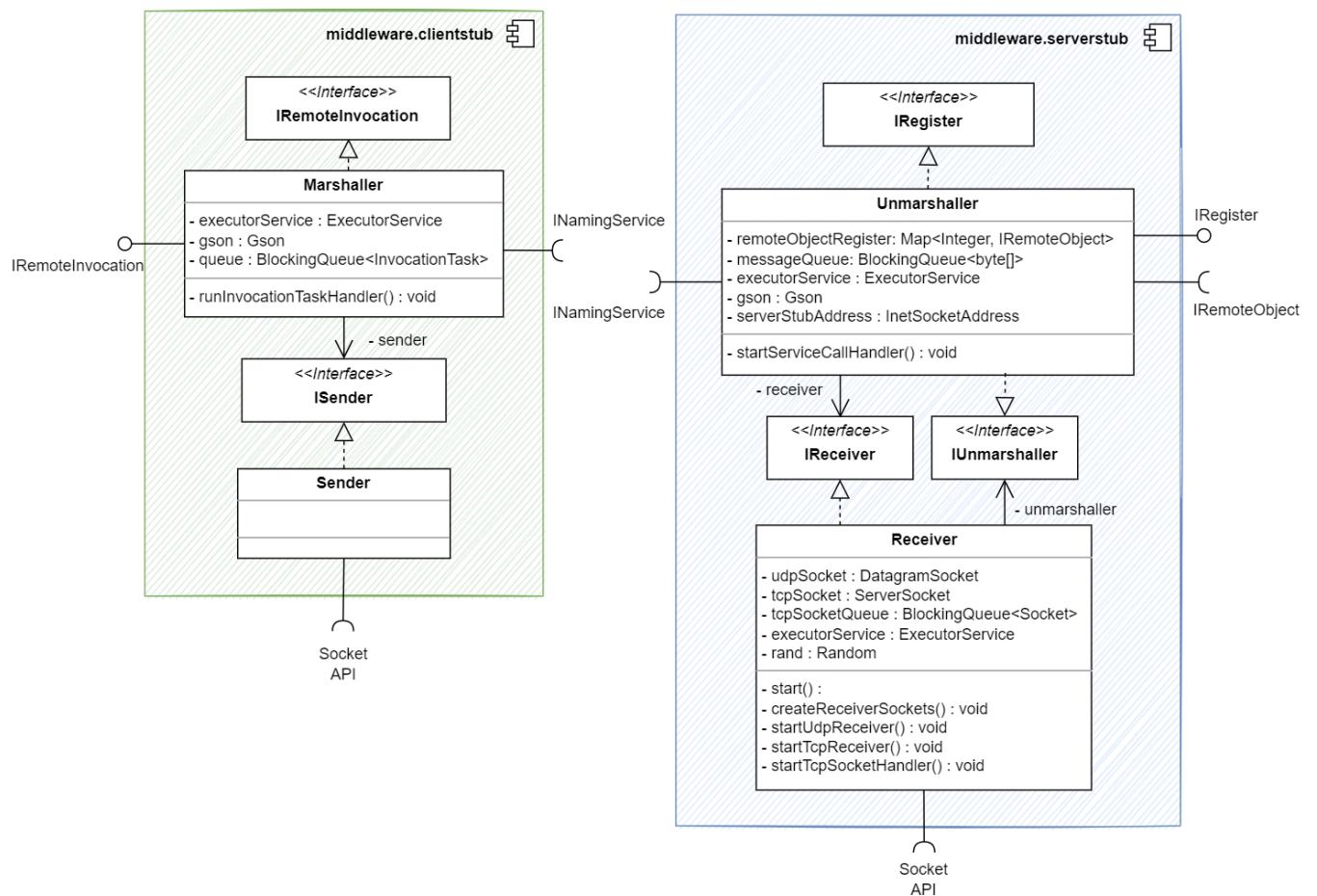
UC	Funktion	Objekt	Vorbedingung	Nachbedingung	Ablaufsemantik	Fehlersemantik
9	<code>registerService(remoteID: String, serviceId: int, address: String) : void</code>	INamingService	-	Service ist unter der remoteld und Serviceld aufzufinden	Ein Service wird beim Naming Service mit seiner remoteld und Id des angebotenen Service registriert.	Fehlerhafte Registrierungsanfragen werden verworfen.
10	<code>unregisterService(remoteID: String) : void</code>	INamingService	-	Die Services mit der übergebenen remoteld sind nicht mehr registriert	Die Services mit der übergebenen remotelD werden aus dem Naming Service entfernt.	Fehlerhafte Abmeldungsanfragen werden verworfen.
4,9,10	<code>sendRequest(messageType : byte, serviceId : int, remoteId : String, address : String, awaitResponse : boolean) : String</code>	NameResolver	Die Applikation wurde gestartet und der NameServer ist erreichbar.	NameResolver kennt NameServer.	Sendet eine Nachricht an den NameServer und gibt die Antwort zurück, wenn es keine Antwort gibt, wird ein leerer String zurückgegeben.	
4	<code>startClearCache() : void</code>	NameResolver	-	-	In regulären Abständen wird der Cache geleert.	
4	<code>lookUpCache(remoteId : String, serviceId : int) : String</code>	NameResolver	-	-	Im Cache wird nach den Informationen gesucht, wenn sie im Cache nicht gefunden werden, wird NULL zurückgegeben.	
4,9,10	<code>start() : void</code>	NameServer	Der NameServer wird noch nicht gestartet.	Der NameServer läuft	Startet den NameServer.	
4,9,10	<code>runServerSocket() : void</code>	NameServer		Ein ServerSocket lauscht auf eingehende Nachrichten	Startet den ServerSocket und wartet auf eingehende Nachrichten.	
4,9,10	<code>processMessage(message : NamingServiceMessage, clientSocket : Socket) : void</code>	NameServer		-	Verarbeitet eine eingehende Nachricht.	
4,9,10	<code>stop() : void</code>	NameServer	-	Der NameServer ist gestoppt	Stoppt den NameServer.	

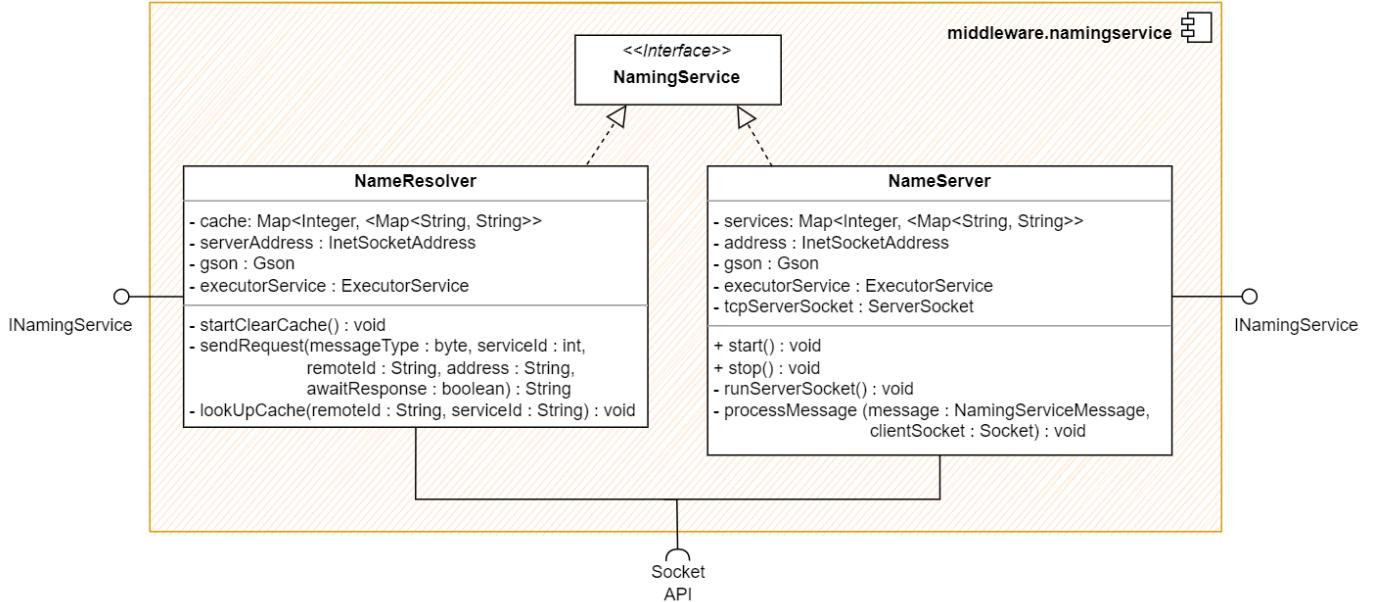
5. Bausteinsicht

Ebene 1:



Ebene 2:

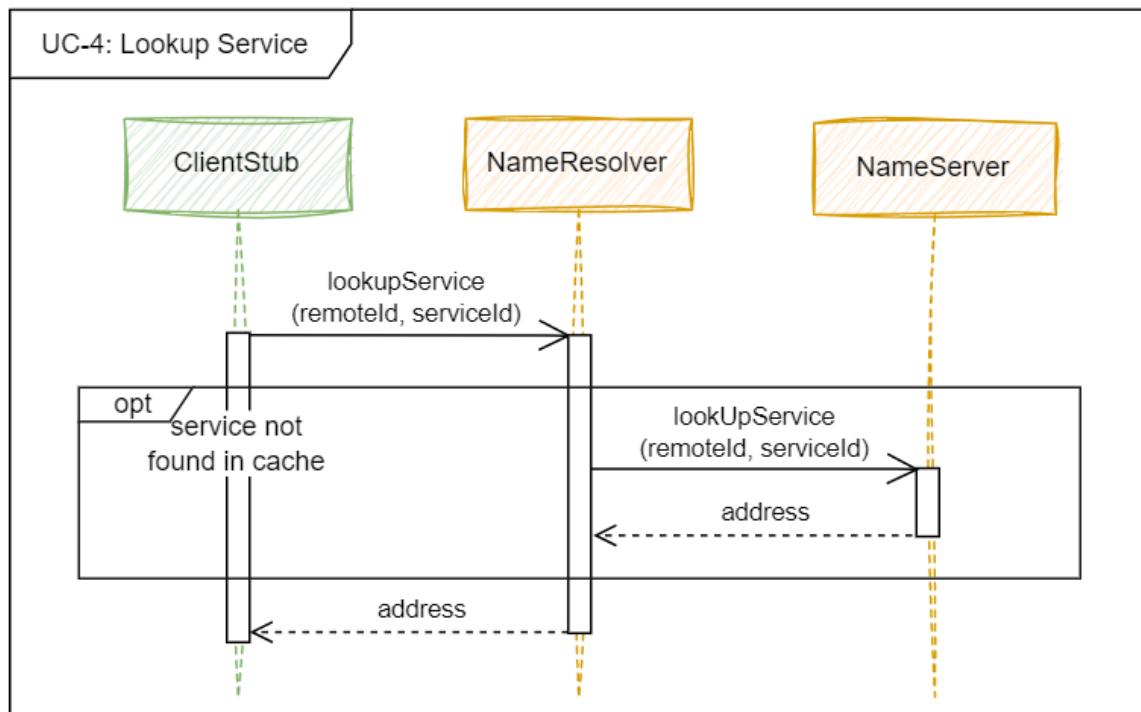


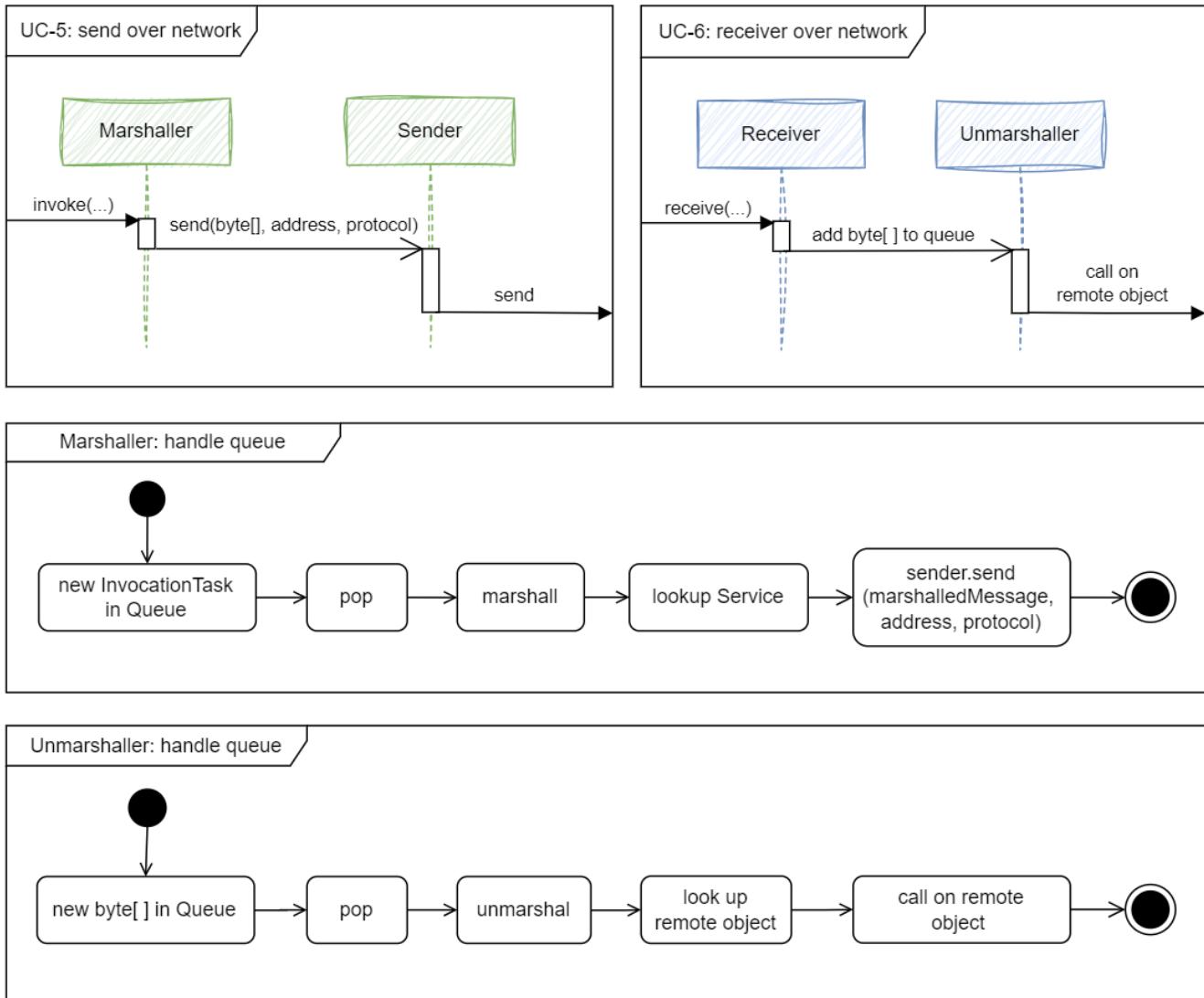


Darüber hinaus werden im Diagramm nicht ausmodellierte Records für die Erleichterung des Datenaustausches verwendet:

Record	package	verwendet von	Beschreibung
InvocationTask	middleware.clientstub	Marshaller	Stellt einen invoke-Aufruf als Objekt dar, damit es in einer Queue gespeichert werden kann.
ServiceCall	middleware	Marshaller, Unmarshaller	Stellt den Aufruf eines Services als Objekt dar und ist in Remote Method Invocation näher beschrieben
NamingServiceMessage	namingservice	NameResolver, NameServer	Dient dem Datenaustausch zwischen Resolver und Server und ist in Naming Service näher beschrieben.

6. Laufzeitsicht





7. Verteilungssicht

8. Querschnittliche Konzepte

8.1 Nachrichtenaustausch

8.1.1 Remote Method Invocation

Die Middleware bietet über ihre Schnittstellen Remote Method Invocation an, d.h. es können Services auf Remote Objekten aufgerufen werden. Ein solcher **ServiceCall** wird im JSON-Format kodiert. Ein Beispiel dafür:

```
{  
    "serviceId": 1,  
    "intParameters": [  
        0,  
        1  
    ],  
    "stringParameters": [  
        "4c3e90e6-9ec6-4a86-88cb-5e05311f8927",  
        "4c3e90e6-9ec6-4a86-88cb-5e05311f8927"  
    ]  
}
```

8.1.2 NamingService

Der NamingService verwendet in JSON kodierte Nachrichten. Ein Beispiel für eine solche **NamingServiceMessage**:

```
{  
    "messageType": 1,  
    "serviceId": 0,  
    "remoteId": "3d46fa0c-0c8e-4555-ad7c-6e31d11920fd",  
    "address": "25.35.92.196:36648"  
}
```

Dabei bezeichnet messageType die Art der Nachricht:

Nachrichtentyp	messageType	serviceld	remoteld	address
REGISTER	1	ID des angebotenen Services	remoteld des Serviceanbieters	Adresse des Serviceanbieters
UNREGISTER	2	NO_SERVICE	remoteld, die aus dem NamingService entfernt werden soll	NO_ADDRESS
LOOKUP	3	ID des gesuchten Services	remoteld des gewünschten Serviceanbieters oder NO_REMOTE_ID	NO_ADDRESS
RESPONSE	4	NO_SERVICE	NO_REMOTE_ID	Adresse des Serviceanbieters

8.2 Identifier

Es wird ein hierarchischer Namensraum verwendet. Jeder Service trägt eine ID (serviceld). Jeder Service kann von mehreren Remote Objekten angeboten werden. Diese werden anhand ihrer remoteld identifiziert. Zur Namensauflösung wird ein zentraler Name Server verwendet, dessen Adresse allen Nodes bekannt sein muss.

Anhang

Use Cases

UC-1: Register Remote Object

Akteur: ApplicationStub-Remote-Objekt

Ziel: Vom ServerStub aufgerufen werden können.

Auslöser: Start der Applikation als NETWORK-Game

Vorbedingungen: ServerStub wurde erstellt.

Nachbedingungen: Der ServerStub merkt sich das Remote Objekt mit aufrufbaren Methoden (Ordinal des ENUM, zB: 1 = DRAW, 2 = REGISTER).

Standardfall:

1. Das System erstellt ein Remote Object.
2. Das System registriert das Remote Object beim ServerStub mit den ServiceIds der Services, die es anbietet.
3. Das System speichert das RemoteObject mit den ServiceIds im ServerStub.

UC-2: Invoke Method

Akteur: ApplicationStub-Caller

Ziel: Eine Methode remote ausführen.

Auslöser: Aufruf von invoke(...) durch Caller-Objekt

Vorbedingungen: ClientStub wurde erstellt. NameServer läuft.

Nachbedingungen: Die Methode wird durch ein Remote-Object ausgeführt (Callee).

Standardfall:

1. ApplicationStub-Caller ruft die invoke(...) Methode der Middleware mit der remoteId, der serviceId, den Parametern und dem InvocationType auf.
2. Der ClientStub verpackt den Methodenaufruf in eine Nachricht (UC-4: Marshalling).
3. Der ClientStub übersetzt den InvocationType aus der invoke-Methode in das zu verwendene Protokoll.
4. Der ClientStub führt einen Lookup nach der RemoteId und ServiceId durch (UC-3: Lookup Service).
5. Der ClientStub versendet die Nachricht aus dem Marshalling an den Service aus Schritt 4 und dem Protokoll aus Schritt 3 (UC-5: Send over Network).
6. Der angesprochene ServerStub erhält die Nachricht und liest sie aus (UC-6: Receive over Network)
7. Der ServerStub entpackt die Nachricht (UC-7: Unmarshalling)
8. Der ServerStub sucht in seiner Liste nach dem angesprochenen Remote Object
9. Der ServerStub führt einen MethodenCall auf dem angesprochenen Remote Object durch (US-8: Call Remote Object)

UC-3: Marshalling/Pack Message

Akteur: Marshaller (ClientStub)

Ziel: Methodenaufruf in Nachricht verpacken

Auslöser: invoke(...) wurde von einem Caller-Objekt des ApplicationStubs aufgerufen

Vorbedingungen: Der Adressat ist bekannt.

Nachbedingungen: Die Nachricht wurde dem Sender übergeben.

Standardfall:

1. Der Marshaller verpackt die Informationen der invoke(...) Methode in einen ServiceCall.
2. Der Marshaller verpackt den MethodCall ins JSON-Format.
3. Der Marshaller erzeugt aus dem JSON ein byte-Array.

UC-4: Lookup Service

Akteur: Marshaller (ClientStub)

Ziel: Adresse des gesuchten Services herausfinden

Auslöser: invoke(CalleelD, Methodename, Parameterliste) wurde von einem Caller-Objekt des ApplicationStubs aufgerufen

Vorbedingungen: NamingServer muss laufen.

Nachbedingungen: Der ClientStub kennt die Informationen des Services (IP:Port, CalleelD).

Standardfall:

1. Der Marshaller führt einen LookUp bei seinem lokalen NameResolver durch mit den Parametern RemoteId und ServiceId des invoke(...) Aufrufs.
2. Der NameResolver sendet eine LookUp-Anfrage an den zentralen NameServer mit den gleichen Daten.
3. Der NamingServer sucht in seiner Tabelle nach dem angefragten Service.
4. Der NamingServer antwortet mit Adresse des Services.
5. Der NameResolver merkt sich den Service in seinem Cache.
6. Der NameResolver gibt dem Marshaller die Informationen über den Service zurück.

Erweiterungsfall:

- 2.a Der NameResolver findet den gewünschten Service in seinem Cache.
- 2.a.1 Weiter in Schritt 6 des Standardfalls.

UC-5: Send over Network

Akteur: Sender (ClientStub)

Ziel: Nachricht über das Netzwerk schicken.

Auslöser: Der Marshaller hat eine Nachricht verpackt und dem Sender übergeben

Vorbedingungen: Der Adressat ist bekannt.

Nachbedingungen: Nachricht wurde ans Netzwerk weitergereicht.

Standardfall:

1. Der Sender öffnet einen Socket an den angegebenen Adressaten mit dem übergebenen Protokoll.
2. Der Sender schickt die Nachricht über den Socket.

Fehlerfall:

- 2.a Der angegebene Adressat ist nicht erreichbar.
- 2.a.1 Der Methodenaufruf wird verworfen.

UC-6: Receive over Network

Akteur: Receiver (ServerStub)

Ziel: Nachricht verarbeiten.

Auslöser: Es kommt eine Nachricht rein.

Vorbedingungen: Receiver lauscht auf Nachrichten.

Nachbedingungen: Receiver hat die Nachricht weitergegeben zum Verarbeiten. Der Receiver befindet sich wieder im listen-Status.

Standardfall:

1. Der Receiver liest die Nachricht vom Socket.
2. Der Receiver übergibt die ausgelesene Nachricht dem Unmarshaller.
3. Der Receiver kehrt in den listen-Status zurück.

UC-7: Unmarshalling/Unpack Message

Akteur: Unmarshaller (ServerStub)

Ziel: Nachricht in Methodenaufruf umwandeln

Auslöser:Unmarshaller erhält eine Nachricht vom Receiver

Vorbedingungen: Der Receiver lauscht auf Nachrichten.

Nachbedingungen: UC-8 wird ausgeführt

Standardfall:

1. Der Unmarshaller entpackt die Nachricht vom Receiver in serviceId und Parameter.
2. UC-2: Call Remote Object durchführen.

UC-8: Call Remote Object

Akteur: Unmarshaller (ServerStub)

Ziel: Methode aufrufen

Auslöser: UC-7 wurde durchgeführt.

Vorbedingungen: Remote-Object muss registriert sein.

Nachbedingungen: Methode wird aufgerufen auf Remote Object.

Standardfall:

1. Unmarshaller sucht im Remote-Object-Register nach dem aufzurufenden Remote Object mit der gesuchten serviceId.
2. Unmarshaller übergibt dem Remote-Object alle nötigen Informationen für den Methodenaufruf.
3. Das Remote-Object ruft die Methode auf.

Fehlerfall:

- 2.a Remote-Object mit der serviceId gibt es nicht im Register.
 - 2.a.1 Der Unmarshaller verwirft die Nachricht.
- 3.a Die Parameter (z.B. serviceId unbekannt) sind fehlerhaft.
 - 2.a.1 Das Remote-Object bricht den Aufruf ab.

UC-9: Register as Service

Akteur: ServerStub

Ziel: Das Remote Objekt sind beim NamingService registriert.

Auslöser: Remote Objekt registriert sich beim ServerStub.

Vorbedingungen: Die Applikation wurde gestartet und der NameServer wurde gestartet. NameResolver kennt NameServer (Config).

Nachbedingungen: Das Remote Objekt ist beim NamingService unter seiner remoteld und serviceld aufzufinden

Standardfall:

1. Der ServerStub schickt einen Registrierungsauftrag an den lokalen NameResolver.
2. Der NameResolver öffnet einen TCP-Socket mit der IP und dem Port zum NameServer.
3. Der NameResolver schickt eine Registrierungsnachricht mit der serviceId (Ordinal des Enums der angebotenen Methode), seiner remoteID und der Adresse des ServerStubs.
4. Der NameServer erhält die Nachricht.
5. Der NameServer speichert den Service mit serviceId, remoteId und Adresse des ServerStubs.

Fehlerfall:

- 4.a Die Nachricht ist falsch kodiert.
- 4.a.1 Der NameServer verwirft die Nachricht.

UC-10: Unregister as Service

Akteur: ServerStub

Ziel: Die Remote Objekte sind nicht mehr beim NamingService registriert.

Auslöser: Die Applikation wird geschlossen.

Vorbedingungen: Der NameServer ist erreichbar.

Nachbedingungen: Die Remote Objekte des ServerStubs sind nicht mehr registriert.

Standardfall:

1. Der ServerStub schickt einen unregister-Aufruf an den lokalen NameResolver.
2. Der NameResolver öffnet einen TCP-Socket mit der IP und dem Port zum NameServer.
3. Der NameResolver schickt eine Abmeldungsnachricht mit der RemoteID an den NameServer.
4. Der NameServer erhält die Nachricht.
5. Der NameServer entfernt alle Services mit der RemoteID aus seiner Tabelle.

Fehlerfall:

- 5.a Die Nachricht ist falsch kodiert.
- 5.a.1 Der NameServer verwirft die Nachricht.