Verteilte Systeme Praktikum

### Tron Game Dokumentation

### Wintersemester 2022/2023

### Gruppe: Beta 5

#### Dominik Martin, Can Heintze, Dominik Mueller

# 

# Einführung und Ziele

## Aufgabenstellung

**Inhalt**

Dieses Dokument beschreibt eine Advanced Client – Server Spielvariante von Tron. Tron ist ein action Rennspiel, bei dem Motorräder einen Schatten hinterlassen. Fahren andere Spieler gegen den Schatten, sind sie tot, sollten Spieler „crashen“ oder auch gegen die Wand fahren, sind diese auch tot und haben verloren. Der letzte Überlebende des Spiels, ist der Gewinner.

Das Spiel soll folgende Anforderungen erfüllen:

### Use Cases

**Nummer:** UC-1

**Titel:** Spieleranzahl festlegen

**Akteur:** Spieler

**Ziel:** Die maximale Anzahl der Spieler für die nächste Spielrunde soll festgelegt werden.

**Auslöser:** Intention eines Spielers, eine vom Standardwert abweichende Anzahl der Mitspieler festzulegen.

**Vorbedingung:**

* Die Anwendung wurde erfolgreich gestartet.
* Es wird Bildschirm 1 angezeigt.

**Nachbedingung:**

* Das Eingabefeld enthält einen ganzzahligen Wert von 2 bis 6 für die Spieleranzahl
* Die anwendungsinterne Spieleranzahl wurde auf den eingegebenen Wert geändert, sodass dieser beim nächsten Spielstart verwendet wird.

**Erfolgsszenario:**

1. System erzeugt GUI Für Bildschirm 1 und zeigt diese an
2. Das System zeigt ein Eingabefeld für die Spieleranzahl an, das zunächst den konfigurierten Standardwert enthält.
3. Der Spieler wählt per Mausklick das Eingabefeld an.
4. Der Spieler ersetzt den vorherigen Inhalt des Eingabefeldes mit einem ganzzahligen Wert von 2 bis 6.
5. Beim Mausklick außerhalb des Eingabefeldes wird der gültige Eingabewert für den nächsten Spielstart übernommen.

**Fehlerfälle:**

1. a) Der Spielers versucht, einen nicht-numerischen Wert einzugeben.
2. a) 1. Die Eingabe des Spielers erscheint nicht im Eingabefeld.
3. a) 2. Weiter bei 3.

3. b) Der Spieler macht eine Eingabe, die keiner Ganzzahl von 2 bis 6 entspricht.

1. b) 1. Das Eingabefeld wird rot umrandet und es wird ein Warnhinweis über die ungültige Eingabe angezeigt. Der Start-Button wird ausgegraut und deaktiviert.
2. b) 2. Weiter bei 3.

**Erweiterungsfälle:**

1. a) Nach Abschluss des Erfolgsszenarios ist immer noch ein Warnhinweis vorhanden.
2. a) 1. Der Warnhinweis und die rote Umrandung des Eingabefeldes werden ausgeblendet.

Der Startbutton wird wieder freigegeben.

**Nummer**: UC-2

**Titel**: Starten eines Spiels

**Akteur**: Der Spieler

**Ziel**: Der Spieler möchte ein Spiel spielen

**Auslöser**: Der Spieler hat das Spiel gestartet, weil er ein Spiel spielen will

**Vorbedingungen**:

* Spiel (Programm) wurde gestartet

**Nachbedingungen:**

* Wartebildschirm mit allen Spielern wird angezeigt

**Erfolgsszenario**

1. Das Spiel zeigt den Startbildschirm an
2. Der Spieler wählt das Feld für die Eingabe der maximalen Spieleranzahl aus
3. Der Spieler gibt eine Spieleranzahl an
4. Das Spiel überprüft die eingegebene Spieleranzahl
5. Der Spieler klickt auf den Button “Spiel starten”
6. Das. System erstellt die GUI für das Wartezeit Fenster
7. Das Spiel zeigt das Wartezeit Fenster an

**Fehlerfall**

1. Das System zeigt an, dass die angegebene Spieleranzahl kein korrekter Wert ist

**Zugrundeliegende Anforderungen:**

* UC-1 Spieleranzahl festlegen

**Nummer:** UC-3

**Titel:** Spiel Beitreten

**Aktuer:** Spieler

**Ziel:** Einem Spiel beizutreten.

**Auslöser:** Spieler hat die intention einem Spiel beizutreten

**Vorbedingung:**

* Bildschirm 2 wurde angezeigt
* Der Spieler hat die Anzahl der Spieler festgelegt
* Der Spieler hat ein Spiel gestartet

**Nachbedingung:**

* Der Benutzer ist dem Spiel beigetreten

**Erfolgszenario:**

1. Das System zeigt den Wartescreen an
2. Das System startet einen Countdown
3. Der jeweilige Spieler klickt auf eine Taste in einer der vordefinierten steuerungsbereichen
4. Das System merkt sich, dass ein neuer Spieler auf dem gerade gedrückten steuerungsbereich beigetreten ist
5. Sobald der Countdown zu ende ist, wird das Spiel gestartet

**Erweiterungsfälle:**

5.a. Es sind nicht alle Spieler beigetreten:

5.a.1 Das spiel wird nach einem bestimmten Countdown gestartet

**Fehlerfälle:**

1. b. Es sind nicht genug spieler eingetreten (weniger als 2):

5.b.1 Das Spiel wird abgebrochen und das System kehrt zurück zum start bildschirm

**Häufigkeit:**

* jedes mal wenn ein Spiel gestartet wird, müssen Spieler beitreten

**Zugrundeliegende Anforderungen**

* UC-1: Spieleranzahl festlegen
* UC-2: Spiel Starten

**Nummer**: UC-4

**Titel**: Bewegung eines spielers

**Akteur**: Der Spieler

**Ziel**: Der Spieler möchte, dass sich seine Spielfigur in der nächsten Bewegung in seine gewünschte Richtung bewegt

**Auslöser**: Spieler drückt eine Richtungstaste Vorbedingung:

**Vorbedingungen:**

* Der Spieler befindet sich in einem laufenden Spiel
* Der Countdown nach dem Start des Spiels ist abgelaufen Nachbedingung
* Die Spielefigur hat sich in die gewünschte Richtung bewegt

**Erfolgsszenario**

1. Das Spiel zeigt das Spiel an
2. Das Spiel zeigt den Spieler in der richtigen Richtung an

**Nummer:** UC-5

**Titel:** Spieler Sterben / verlieren

**Aktuer:** Spieler

**Ziel:** Ein Spieler stirbt im Spiel bzw. verliert das Spiel

**Auslöser:**

* Ein Spieler fährt gegen eine Wand
* Ein Spieler fährt gegen den Schatten eines anderen Spielers oder gegen seinen eigenen Schatten
* Ein Spieler fährt gegen ein anderes Motorrad
* Beide Spieler sterben wenn sie einen Frontalen Zusammenstoß haben

**Vorbedingungen:**

* Der Spieler besitzt die Tron Applikation
* Der Spieler hat die Spieleranzahl wurde festgelegt
* Der Spieler hat das Spiel gestartet
* Spieler sind dem Spiel beigetreten

**Nachbedingungen:**

* Der Spieler kann nicht mehr weiter am Spiel teilnehmen
* Der Schatten des Spielers welcher gestorben ist, verschwindet aus dem Spiel

**Erfolgszenario:**

1. Ein Spieler spielt das Spiel und löst dabei einen der oben genannten Auslöser aus
2. Der Spieler verschwindet aus dem Spielfeld & sein Schatten verschwindet aus dem Spielfeld
3. Das Spiel wird weitergespielt, ohne den gerade gestorbenen Spieler

**Erweiterungsfälle:**

1.a Wenn mehrere Spieler gleichzeitig sterben (durch gleiche oder verschiedene Auslöser)

1.a.1 Alle Spieler, welche gleichzeitig sterben, scheiden gleichzeitig aus dem Spiel aus & deren Schatten verschwindet aus dem Spiel

**Häufigkeit:**

* In jedem Spiel, welches Gespielt wird, werden Spieler sterben / verlieren.

**Zugrundeliegende Anforderungen:**

* UC-4 Bewegung eines spielers

**Nummer**: UC-7

**Titel**: Startpositionen der Spieler

**Akteur**: Das System

**Ziel**: Alle Spieler besitzen Startpositionen, welche ein faires Spiel gewährleisten

**Auslöser**: Ein Spiel wurde gestartet

**Vorbedingungen**

* Ein Spiel wurde gestartet
* Das Spiel wurde geladen

**Nachbedingungen**: Spieler befinden sich auf fairen Startpositionen

**Erfolgsszenario**

1. Das System zeigt das Spiel an
2. Das System berechnet faire Startpositionen
3. Das System setzt die Spieler bei den Startpositionen ein

**Zugrundeliegende Anforderungen:**

* UC-2 starten eines Spiels

**Nummer:** UC-8

**Titel:** Spieler hinterlassen Schatten

**Aktuer:** Spieler

**Ziel:** Der Spieler bzw. das Motorrad des Spielers hinterlässt einen Schatten

**Vorbedingung:**

* Der Spieler besitzt die Applikation
* Der Spieler hat das Spiel gestartet
* Die Spieler sind dem Spiel beigetreten
* Die Spieler können sich auf dem Spielfeld bewegen

**Nachbedingungen:**

* Nach jeder Bewegung taucht an der vorherigen Position des Spielers ein Schatten auf

**Erfolgszenario:**

1. Der Spieler bewegt sich auf dem Spielfeld
2. Bei jeder Bewegung des Spielers wird ein stück schatten mehr generiert und taucht hinter dem Spieler auf in der Farbe des Spielers

**Erweiterungsfälle:**

**Fehlerfälle:**

**Häufigkeit:**

* In jedem Spiel, welches gespielt wird, wird bei jedem Spieler ein Schatten hinterlassen

**Zugrundeliegende Anforderungen**

* UC-4 Bewegung eines spielers

**Nummer:** UC-9

**Titel:** Game over Screen

**Aktuer:** System

**Ziel:** Der Game over screen wird angezeigt mit den jeweiligen Gewinnern oder Unentschieden

**Auslöser:** Ein Spiel wurde beendet. bzw. in einem Spiel verbleiben weniger als 2 Spieler

**Vorbedingung:**

* Ein Spiel wurde gestartet und von mindestens 2 Spielern gespielt
* Das Spiel wurde beendet indem im Spiel weniger als 2 Spieler verblieben sind.

**Nachbedingungen:**

* Der Startbildschirm wird nach dem “Game over” screen angezeigt

**Erfolgszenario:**

1. Das Spiel wird beendet
2. Das System erstellt die GUI für den Game Over screen
3. Das System wechselt vom “Spielfeld” screen zum “Game over” screen
4. Das System startet einen 10 sek. countdown.
5. Auf dem Game Over Screen wird der Spieler angezeigt, welcher gewonnen hat. Dies wird durch “Spieler X” angezeigt, in der jeweiligen Spieler Farbe
6. Sobald der 10 sek. countdown abgelaufen ist, wird auf den Startbildschirm der Applikation gewechselt.

**Erweiterungsfälle:**

4.a Wenn das Spiel mit einem Unentschieden beendet wurden ist

4.a.1: Anstatt “Spieler X” wird der Schriftzug “Unentschieden” angezeigt

**Fehlerfälle:**

**Häufigkeit:**

* Jedes mal, wenn ein Spiel gespielt wurden ist, wird dieser Screen angezeigt.

**Zugrundeliegende Anforderungen:**

* UC-5 Spieler sterben / verlieren

## Motivation

Die wesentliche Motivation für uns, dieses Spiel zu implementieren ist es, die PVL zu erhalten. Weitere Motivationspunkte wären aber auch, neues zu lernen und unser bisheriges Wissen zu vertiefen.

## Qualitätsziele

|  |
| --- |
| **Qualitätsziele:** |
| Gut definierte Schnittstellen |
| Kompatibilität zu einer anderen Gruppe (Mindestens zwei Teams müssen miteinander spielen können) |
| Fehlertoleranz (Wenn ein Spieler abstürzt, egal welcher Spieler, dann geht das Spiel trotzdem weiter) -> Stabilität |
| Das Spiel soll gleich schnell laufen für alle (keine Jitter-abhängigkeit) |
| Ein Spiel mit 6 Leuten, soll einmal komplett ohne Fehler durchlaufen. |

## Stakeholder

**Inhalt**

Unsere Stakeholder sind die Entwickler (Studenten), der Kunde (Professor) und die Spieler (Studenten).

Die Stakeholder mit deren Kontakt werden in der unteren Tabelle aufgelistet:

| Rolle | Kontakt | Erwartungshaltung |
| --- | --- | --- |
| *Entwickler* | [Dominik.martin@haw-hamburg.de](mailto:Dominik.martin@haw-hamburg.de) | *Ein gutes Spiel zu programmieren & die PVL zu erhalten* |
| *Entwickler* | [*Can.heintze@haw-hamburg.de*](mailto:Can.heintze@haw-hamburg.de) | *Ein gutes Spiel zu programmieren & die PVL zu erhalten* |
| *Entwickler* | [*Dominik.mueller@haw-hamburg.de*](mailto:Dominik.mueller@haw-hamburg.de) | *Ein gutes Spiel zu programmieren & die PVL zu erhalten* |
| *Kunde* | [*Martin.becke@haw-hamburg.de*](mailto:Martin.becke@haw-hamburg.de) | *Ein Lauffähiges Spiel, bei dem 6 Spieler gleichzeitig ein komplettes Spiel ohne fehler durchspielen können.* |
| *Spieler* | *n/a* | *Ein funktionierendes Spiel spielen und dabei Spaß haben* |

# 

# Kontextabgrenzung

## A picture containing shape Description automatically generatedFachlicher Kontext

## Technischer Kontext

Graphical user interface

Description automatically generated

# 

# Lösungsstrategie

### Funktionale Anforderungen

|  |  |  |  |
| --- | --- | --- | --- |
| **Methoden Signatur** | **Beschreibung** | **Klasse** | **UseCase** |
| void showStartScreen () | * Zeigt bildschirm 1 an: * Zeigt den Start Button an * Ermöglicht dem Spieler auszuwählen, wie viele Spieler mitspielen sollen | ScreenHandler | UC-1 |
| void showLobbyScreens(int, int) | * Zeigt Bildschirm 2 und / oder 3 an. Also den Wartebildschirm für den Spieler beitritt oder den Endbildschirm, mit der Spieler Nummer vom Gewinner | ScreenHandler | UC-3 / UC-9 |
| Void showGameScreen(Map<Integer, int[][]> | * Zeigt den Game Bildschirm an * Zeichnet die jeweiligen Motorräder * Bekommt eine Map, der key steht für den jeweiligen Spieler und der Value sind dann die jeweiligen Koordinaten fuer diesen Spieler | ScreenHandler | UC-4 |
| Void init(Stage) | * Initialisiert die View | ScreenHandler | UseCase Übergreifend |
| Boolean checkInputString(String) | * Prueft, ob der Input des Spielers valide ist. (also ob die korrekten Pfeiltasten gedrückt wurden sind etc.) | InputHandler | UC-4 / UC-3 |
| Void forwardInput (String) | * Gibt den Spieler input an den Controller weiter | InputHandler | UC-4 / UC-3 |
| Void handle(KeyEvent) | * Nimmt den Spieler input entgegen und ruft forwardInput & checkInputString auf | InputHandler | UC-4 / UC-3 |
| Void endgame() | * Das Spiel wird beendet, indem es den 4 Bildschirm anzeigt mit dem jeweiligen Gewinner etc. * Ruft createWinnerScreen auf | Lobby | UC-9 / UC-5 |
| Void createWinnerScreen() | * Überprüft den Gewinner Status aus der gamelogic und zeigt den 4ten Bildschirm an | Lobby | UC-9 / UC-5 |
| Lobby initLobby() | * Zeigt Bildschirm 2 an (wartescreen) | Lobby | UC-3 |
| Void playerJoin(int) | * Erstellt die jeweiligen spieler, wenn diese dem Spiel aus dem 2ten Bildschirm beitreten * Die view wird dabei aktualisiert um die beigetretenen Spieler anzuzeigen | Lobby | UC-3 |
| Void setCurrentPlayerCount(int) | * Setzt den CurrentPlayerCount | Lobby | Use Case Übergreifend |
| Void updateView(int) | * Ruft showStartScreen(int) auf | Lobby | Use Case Übergreifend |
| Map<Integer, List<String>> getPlayerMappings() | * Gibt die Playermappings zurück * Die Playermappings sind die jeweiligen Tastenbelegungen für die jeweiligen Spieler | Config | Use Case Übergreifend |
| Int getLobbyTimerDuration() | * Gibt die Sekunden Anzahl für die Dauer der Lobby (wartebildschirm) an | Config | UC-3 |
| Int getPlayerCount() | * Gibt die Maximal mögliche Spieleranzahl zurück (6) | Config | UC-3 / Use Case Übergreifend |
| Void setPlayerCount() | * Setzt den playerCount | Config | Use Case Übergreifend |
| Int[] getBoardSize() | * Gibt die Größe des Boards zurück | Config | UC-7 |
| Int getCellSize() | * Gibt die groeße der jeweiligen Zellen zurück | Config | Use Case Übergreifend |
| Void loadConfigFile(String) | * Ladet die Config Datei und setzt die jeweiligen Werte | Config | Use Case Übergreifend |
| Void startLobbyTimer(int, Runnable) | * Startet den Timer für den zweiten Bildschirm (Wartebildschirm) | ThreadedTimerImpl | UC-3 |
| Void startGameTimer(int, Runnable) | * Starten den Timer für das Spiel * Startet einen Thread, welcher für die Aktualisierung der Ticks im Spiel sind | ThreadedTimerImpl | UC-4 / UC-8 |
| StopGameTimer | * Stoppt den Spiel Timer | ThreadedTimerImpl | UC-4 / UC-8 |
| Void setPlayerMove(int,int) | * Ruft setIntendedDirection * Setzt die Bewegung für einen Spieler | PlayerManager | UC-4/UC-8 |
| Void setIntendedDirection(int, int) | * Setzt die Bewegung für einen Spieler | PlayerManager | UC-4/UC-8 |
| List<Players> getLivingPlayers() | * Gibt eine Liste an Spieler zurück, welche noch leben | PlayerManager | Use Case Übergreifend |
| Void notifyCollisions(int[][]) | * Setzt den Lebens Status auf False, wenn eine Kollision vorhanden ist | PlayerManager | UC-5 |
| Void commitPlayerMoves() | * Ruft commitMove auf die jeweiligen Spieler auf und setzt die Richtung für den Spieler | PlayerManager | UC-4 |
| Void createPlayer(List<String>, int) | * Erzeugt spieler | PlayerManager | UC-3 |
| List<int[]> getPlayerPositions(int) | * Gibt eine Liste an Positionen des jeweiligen Spielers zurück | PlayerManager | Use Case Übergreifend |
| Void killPlayer(int) | * Setzt den Lebensstatus eines Spielers auf False | PlayerManager | UC-5 |
| Plaer getPlayerById(int) | * Gibt den jeweiligen Spieler einer Spieler ID zurück | PlayerManager | Use Case Übergreifend |
| Map<Integer, int[][]> checkPlayerCollision(List<int[]>) | * Prüft, ob es eine Kollision gibt und gibt die Koordinaten der Kollision zurück | PlayerManager | UC-5/UC-4 |
| Void killPlayers() | * Löscht alle Spieler | PlayerManager | UC-5 / UC-9 |
| List<Player> getPlayers() | * Gibt eine Liste an Spielern zurück | PlayerManager | Use Case Übergreifend |
| Void commitMove(int) | * Setzt die jeweilige Richtung für einen Spieler, dabei wird die Richtung als int der Methode übergeben. * Es gibt noch eine Überladung mit void commitMove() | Player | UC-4 |
| Void changePosition(int,int) | * Ändert die Position auf die neuen x,y Koordinaten eines Spielers | Player | UC-4 |
| Bool checkBackwards(int[]) | * Prüft ob ein Spieler versucht, einen Schritt zurück zu machen. * Wenn ja, dann wird das Motorrad trotzdem ein Schritt in die bisherige Richtung gesetzt | Player | UC-4 |
| Void resetNextId() | * Setzt die nextId variable zurück. * Dies wird genutzt, damit, wenn ein neues Spiel startet, die Spieler, wieder mit ID’s 1-6 erstellt werden | Player | Use Case Übergreifend |
| Void startGame() | * Das Spiel wird gestartet * Die Startpositionen der Spieler wird berechnet indem calculateStartPositions aufgerufen wird | GameLogic | UC-2 |
| Bool updateTick() | * Dies aktualisiert den Spiel Status mit den aktuellen Werten für das Board (Positionen etc.), nach jedem Tick | GameLogic | Use Case Übergreifend |
| List<int[]> checkCollision | * Prüft auf eine Kollision und gibt eine Liste mit Koordinaten, an denen die Kollision stattgefunden hat, | GameLogic | UC-5 |
| Void calculateStartPositions() | * Berechnet die Startpositionen für die jeweiligen Spieler am Anfang des Spiels | GameLogic | UC-7 |
| Int[] getWinnerStatus() | * Gibt den Ausgangsstatus des Spiels zurück | GameLogic | Use Case Übergreifend |
| Void removePlayers() | * Ruft killPlayers auf den PlayerManager auf | GameLogic | UC-5 / UC-9 |
| Void updateBoard() | * Aktualisiert das Board mit den jeweils neuen Positionen der Spieler und Schatten | GameLogic | UC-4 / UC-8 |
| Void setStartPosition(List<Player>, int) | * Setzt die Start Positionen der Spieler nach der Berechnung der Start Positionen | GameLogic | UC-2 / UC-4 |
| Void remainingTime(int) | * Zeigt den Wartebildschirm an | Board | UC-3 |
| Void updateBoard(List<int[]>) | * Aktualisiert das Spiel Feld mit den jeweiligen Positionen der Motorräder und Schatten indem es diese einer „Obstacles“ Liste anfügt. | Board | UC-4 / UC-8 |
| Board initBoard() | * Initialisiert das Spiel Feld | Board | Use Case Übergreifend |
| Void updateView(int, Map<Integer, int[][]> | * Ruft showGameScreen auf den ScreenHandler auf und zeichnet somit die Motorräder | Board | UC-4 / UC-8 |
| Void updateView(int, int) | * Ruft showLobbyScreens auf * Wird genutzt, wenn das Spiel beendet ist und auf den Endscreen gewechselt werden soll | Board | UC-9 |
| List<int[]> getObstacles() | * Gibt eine Liste mit Koordinaten zurück, an denen sich die „Obstacles“ auf dem Spielfeld befinden | Board | Use Case Übergreifend |
| Int[] getBoardSize() | * Gibt die Größe des Spiel Felds zurück | Board | Use Case Übergreifend |
| Void clearBoard() | * Löscht alles aus der Liste der Obstacles raus und räumt somit das Spiel Feld leer | Board | UC-5 / UC-9 |
| Void removeObstacles(List<int[]>) | * Löscht die Obstacles an einer bestimmten Position | Board | UC-5 |
| Void playerJoin(int) | * Wird aufgerufen, wenn ein Spieler beitritt, | GameManagerImpl | UC-3 |
| Void loadLobby() | * Ruft initLobby auf die Lobby auf und zeigt somit den Wartebildschirm an | GameManagerImpl | UC-3 |
| Bool isReadyToPlay() | * Prüft ob die derzeitige Spieler Anzahl größer 1 ist und kleiner gleich die Maximal Spieler Anzahl | GameManagerImpl | UC-3 |
| Void startGame() | * Initialisiert das Spiel Feld | GameManagerImpl | UC-2 / UC-3 |
| Void updateView(int) | * Ruft updateView auf der Lobby auf | GameManagerImpl | Use Case Übergreifend |
| Void removePlayers() | * Setzt den currentPlayerCount auf 0 und entfernt die Spieler | GameManagerImpl | Use Case Übergreifend |
| Void endgame() | * Säubert das Spiel Feld und lässt die Spieler entfernen | GameManagerImpl | UC-9 |
| Void onKeyPress(String) | * Hier werden Aktionen ausgeführt, sobald ein Spieler auf eine Taste klickt. * Je nach Situation, bewegt der Spieler damit sein Motorrad oder er tritt dem Spiel bei | PlayerInputManagerImpl | UC-3 / UC-4 |
| String[] getValidKeys() | * Gibt zugelassene Tasten zurück | PlayerInputManagerImpl | Use Case Übergreifend |
| Void switchPlayerController() | * Wechselt den PlayerController, so, dass es im Wartebildschirm den PlayerController für das Spiel Beitreten nutzt und im Spiel dann den PlayerController für das Bewegen der Spieler nutzt | PlayerInputManagerImpl | UC-3/UC-4 |
| IPlayerController createPlayerController(String) | * Erstellt die jeweiligen PlayerController je nach Bedarf | PlayerControllerFactory | Use Case Übergreifend |
| Void loadMappings() | * Ladet und setzt die Tastenbelegungen der jeweiligen Spieler | IPlayerController | Use Case Übergreifend |
| Int gerDirectionForKey(String) | * Gibt für die jeweilige Taste die dazugehörige Richtung zurück | PlayerControllerMovementImpl | UC-4 |
| Int getPlayerForKey(String) | * Gibt die PlayerID für die jeweilige Taste zurück | PlayerControllerMovementImpl | UC-4 |
| Int getPlayerForKey(String) | * Gibt die PlayerID für die jeweilige Taste zurück * Wird beim Spiel Beitritt genutzt | PlayerControllerOnbaordingImpl | UC-3 |
| Int changeToNextScene() | * Wechselt zur nächsten Szene, also zum nächsten Bildschirm | SceneChangerImpl | Use Case Übergreifend |
| Int commitNadChangeToNextScene(int) | * Setzt den PlayerCount, welcher vom Spieler im Start Bildschirm angegeben wurde und wechselt zur nächsten Szene | SceneChangerImpl | UC-1 |
| Int changeToPreviousScene() | * Wechselt zur vorherigen Szene, also zum vorherigen Bildschirm | SceneChangerImpl | Use Case Übergreifend |
| Void registerNextSceneCallback(Runnable) | * Registriert den Callback für die nächste Szene | SceneChangerImpl | Use Case Übergreifend |
| Void registerPreviousSceneCallback  (Runnable) | * Registriert den Callback für die vorherige Szene | SceneChangerImpl | Use Case Übergreifend |
| Void init() | * Initialisiert die jeweiligen Szenen | SceneChangerImpl | Use Case Übergreifend |
| Void updateTick() | * Stoppt den Game Timer wenn das Spiel zu ende ist | SceneChangerImpl | UC- 5 |

# Bausteinsicht

## Komponenten Diagramm:

Dies ist das Komponenten Diagramm, welches die unsere Komponenten auf höchster Ebene zeigt.

## Diagram Description automatically generated

Hier folgen nun die Komponenten Diagramme auf zweiter Ebene:

## Diagram Description automatically generated

Diagram

Description automatically generated

Diagram, engineering drawing

Description automatically generated

Nun Folgen unsere Klassediagramme:

Graphical user interface, application, Teams

Description automatically generated

Diagram

Description automatically generated

Graphical user interface, application

Description automatically generated

Graphical user interface

Description automatically generated

Graphical user interface, application, Teams

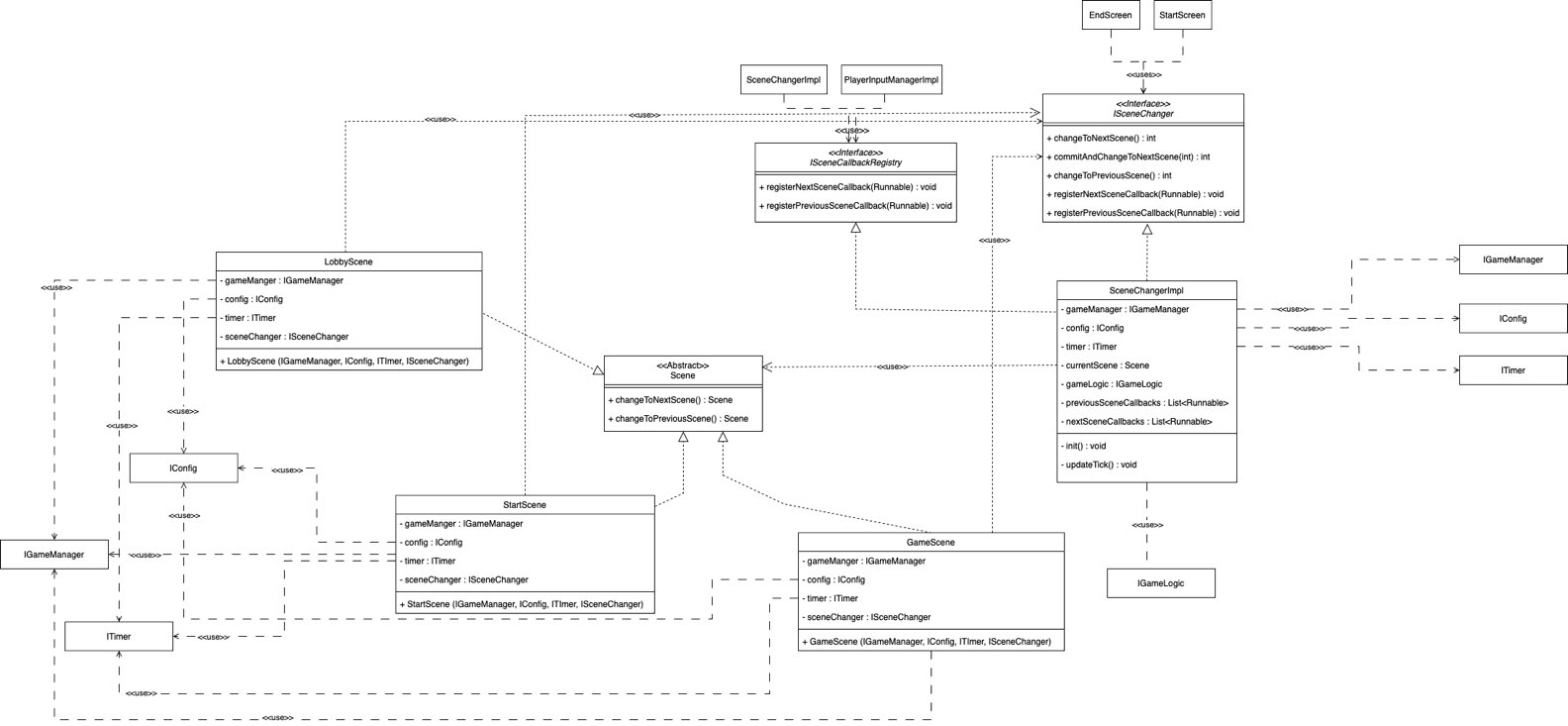
Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated



Graphical user interface, application

Description automatically generated

Graphical user interface, application, Teams

Description automatically generated

# Laufzeitsicht

In diesem Kapitel beschreiben wir, was in der Tron Standalone Applikation während der Laufzeit passiert. Die Beschreibung erfolgt mittels Sequenzdiagrammen.

Die Sequenzdiagramme zeigen die Methodenaufrufe zwischen den Komponenten auf Zwei Ebenen auf.

Sequenzdiagramme der **ersten** Ebene:

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Sequenzdiagramme der **zweiten** Ebene:

Diagram

Description automatically generated

Diagram

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Diagram

Description automatically generated

Diagram

Description automatically generated

Diagram, schematic

Description automatically generated

Diagram

Description automatically generated

Graphical user interface, diagram, text

Description automatically generated

# Architekturentscheidungen

Für die Architektur in unserer Tron Applikation haben wir uns für die **MVC-Architektur** (Model-View-Controller) entschieden. Wir haben uns für diese Architektur entschieden, da es die Wiederverwendung von Code vereinfacht, indem es klar definiert, wie die jeweiligen Komponenten der Tron Applikation interagieren. Außerdem können, nach einer guten Definition von Schnittstellen, diese Komponenten unabhängig voneinander entwickelt werden, so dass es die Verteilung im Team erleichtert.

Zwischen den Model-View-Controller Komponenten haben wir keine eigens entwickelte Datentypen übergeben, sondern haben an den jeweiligen Schnittstellen immer mit Java Datentypen wie Strings, Integern, Maps gearbeitet. Außerdem haben wir auch alle Exceptions direkt in derselben Komponente behandelt. Dies haben wir gemacht, um den Traffic zwischen den Komponenten zu reduzieren.