VSP - arc42 Dokumentation:
Verteiltes Tron Spiel (Demo): “Bikes”

Praktikumsgruppe 02
Ashkan Haghighi Fashi
Minh Lê

31.01.2022 / Version 3.0

1. Einführung und Ziele

Im Rahmen der Veranstaltung Verteilte Systeme entwickeln wir einen Tron-Klon den wir “Bikes" nennen. Diese Dokumentation umfasst die Entwicklung des Spiels als eine Standalone Applikation. In diesem 1vs1 Multiplayer Spiel steuern beide Spieler jeweils ein Motorrad, welches eine Trail hinter sich zieht. Auf einem 2-dimensionalen Feld versuchen sich die Spieler gegenseitig auszuweichen bzw. den Gegner den Weg abzuschneiden. Derjenige, der zuerst in eine Trail oder Wand fährt, verliert das Spiel.

1.1. Anforderungen (Requirements)

- In einer Lobby kann man Räume betreten, in denen 2 Spieler gegeneinander antreten. Man kann auch einen Raum erstellen.
- Finden sich 2 Spieler in einem Raum startet das Spiel.
- Bikes wird in einer Rasterwelt gespielt, die von 4 Wänden umgeben ist.
- Ein Spieler belegt jeweils einen Kasten.
- Beide Spieler bewegen sich im gleichen Takt automatisch einen Kasten weiter.
- Zu Beginn stehen die Spieler so, dass sie sich in einem gewissen Abstand gegenseitig anschauen.

1.1. Qualitätsziele (Quality Goals)

ID	Priority	Quality Goal	Scenario / Erklärung / Beispiele
Q1	1	Performance (Efficiency, Reliability)	Der Server darf nicht von Teilnehmern blockiert werden
Q2	2	Fairness (Suitability)	Die Spieler werden immer fair behandelt
Q3	3	Extendibility (Maintainability)	Die noch folgenden Anforderungen sollen einfach erweiterbar sein
Q4	4	Distributed (Compatibility)	Die Architektur soll ein verteiltes System sein

1.2. Stakeholder

Role/Name	Contact	Expectations

Spieler		Das Spiel soll fair, flüssig und spaßig sein
Entwickler/Architekten /Tester/Betreiber	Minh Lê, Ashkan Haghighi Fashi	• Der Anwendungscode soll leicht zu verstehen und erweiterbar sein • Das Spiel soll möglichst wenige Bugs haben und leicht zu testen sein • Wir nutzen alles was uns gegeben wird (z.B. view-library) • Wir wollen zusammen als Team an dem Projekt arbeiten • Wenn wir am Ende ein verteiltes Spiel haben, können wir es unkompliziert aufsetzen.
Product Owner	Martin Becke	Ziel: Verteiltes "Tron" Spiel (Demo) **Anforderungen** • Umsetzung in einer Objekt-orientierten Sprache (Beispiel: Java) • Server darf nicht von Teilnehmern blockiert werden • Weitere Anforderungen werden im Praktikum bekannt gegeben. **Abgabe**: AdvancedClient-Server Spielvariante. Abgabe wird geprüft auf • Berücksichtigung der Anforderungen • Feste Methode zum Projektmanagement (Nachweis) • Feste Methode für Dokumentation (Wichtig: systematisch und methodentreu) – Vorschlag: ARC42 • Kommunikationsalternative mit RPC Schnittstelle – Eigene RPC Umsetzung kein Framework => Lösungsteil A • Protokoll Definition mit Fehlersemantiken • Problemlösungsstrategie müssen aus Referenzliteratur oder akzeptierter Fremdliteratur abgeleitet werden. **Abnahme**: • Spieldemo mit n Instanzen und vollständige Dokumentation **Bemerkung Dokumentation (Minimum):** • Klare Darstellung der Struktur in mindestens 2 Hierachieebenen • Komponentendiagramm, Klassendiagramm, Verteilungsdiagramm • Klare Darstellung des Verhaltens durch Sequenzdiagramm, Aktivitätsdiagramm, Zustandsdiagramm **Bemerkung Code:** • Dependency-Inversion-Prinzip • Code muss zur Dokumentation passen und Dokumentation zum Code • Einsatz von Frameworks müssen vom Lehrenden akzeptiert werden **Nachtrag aus Kundengespräch (28.10):** • Spieler sind fair zu behandeln • N Spiele sollen vom System unterstützt werden • Maximal 2 Spieler je Spiel • Jeder Spieler bekommt seine eigene Gui • Jeder Spieler kann sich selbst einen Spiel über einen

		Spielraum-Namen zuordnen ● Steuerung über Tastatur ● Spieler bewegt sich automatisch geradeaus. Spieler kann Richtung manipulieren, aber nicht zurück ● Nach Beendigung des Spiels kann neues Spiel gestartet werden

2. Randbedingungen (Architecture Constraints):

- Eine Liste an Constraints, welche sich durch die Wünsche des Kunden nunmal so ergeben.
- Bezogen auf die Architektur unserer Anwendung
- Sind verhandelbar mit dem Kunden
- Ggf. unterteilbar in technische, organisatorische, politische Constraints, Konventionen etc.

2.1. Allgemeine Randbedingungen

Randbedingung	**Erklärung**
Projektmanagement	übliches Vorgehensmodell (Anf., Entw. Impl. Test. Wart.) und verfeinern iterativ, Gitlab

2.2. Technische Randbedingungen

Randbedingung	**Erklärung**
Implementation	Java, JavaFX, JVM, view-library
Design	Es handelt sich später um eine Applikation für ein verteiltes System
Repository	HAW-Gitlab
Styling	Styling der Anwendung in CSS

2.3. Organisatorische Randbedingungen

Randbedingung	Erklärung
Abgabetermin	Bis zum 30.01.22 muss alles so gut wie fertig sein
Dokumentation	ARC42 Template
Kommunikation	Über Teams/Whatsapp, weil wir es in Präsenz zur Uni nie schaffen
Bearbeitung der Dokumentation	Kollaboratives arbeiten über Google Docs

2.4. Konventionen

Randbedingung	Erklärung
Sprache in den Dokumenten	Englische Klassennamen, englische Methodennamen, deutsche Kommentare, deutsche Dokumentation

3. Kontextabgrenzung (Context View):

3.1. Fachlicher Kontext

Der fachliche Kontext in dem sich diese Applikation bewegt ist unserer Meinung nach recht simpel. Das System selbst kann von mehreren Akteuren genutzt werden, und es bietet ihnen notwendige Funktionalitäten für Raumverwaltung an (UC1 bis UC5). Und das Spielen von Tron selbst (UC6 bis UC8). Auf Figure.2 ist dieser Sachverhalt als Usecase Diagramm dargestellt.

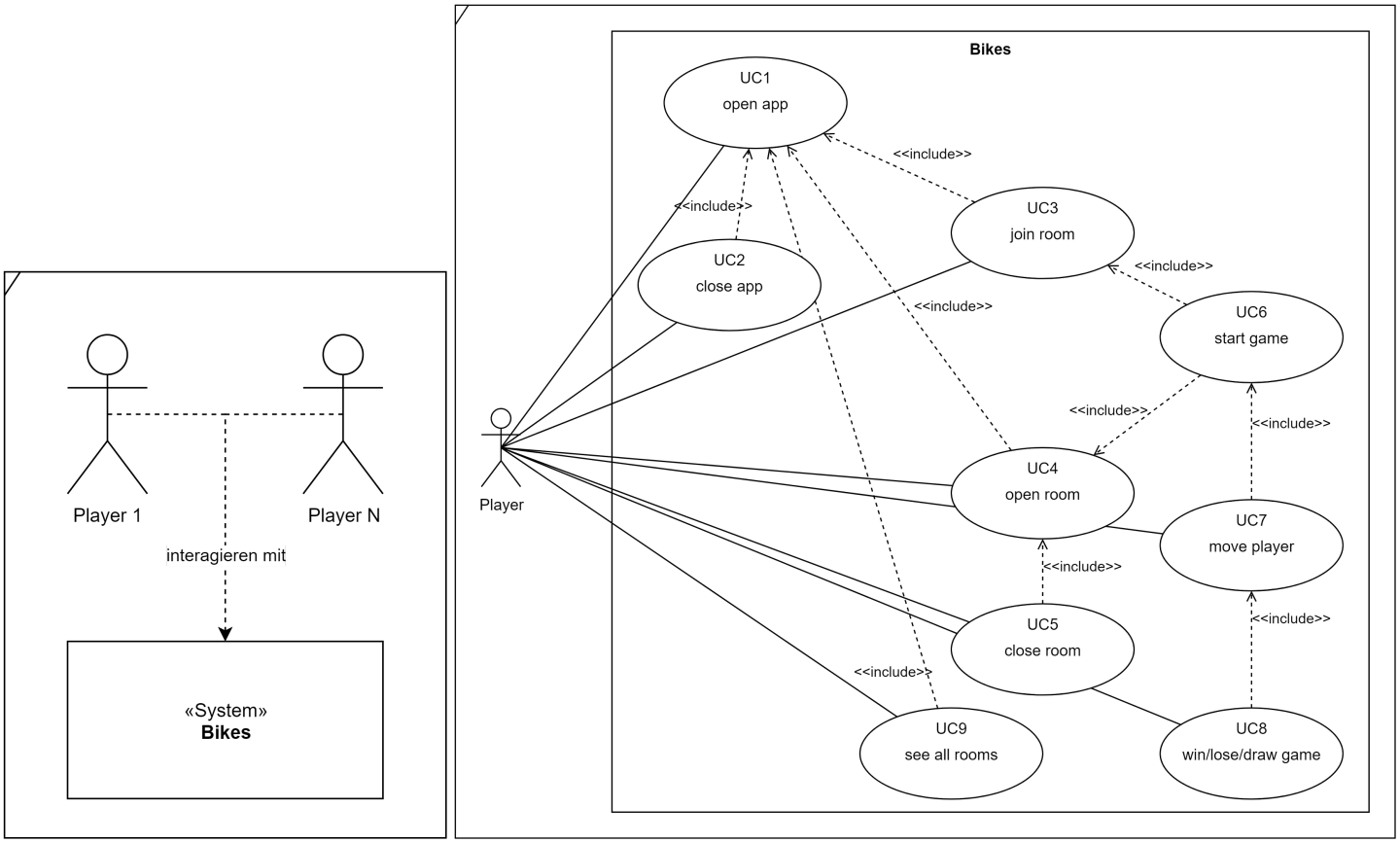


Figure 1. + 2. Fachlicher Kontext und Use Case Diagramm

In dieser Tabelle liegt eine kurze Beschreibung zu jedem Usecase vor.

Usecase	Beschreibung
UC1	Man startet die Applikation und wird mit einer nutzbaren GUI begrüßt.
UC2	Wenn man die Applikation schließt, wird alles aufgeräumt und Räume die man erstellt hat werden geschlossen.
UC3	Man kann einem Raum beitreten, um ein Spiel zu starten.
UC4	Ein Raum kann geöffnet werden, zu dem andere Spieler im LAN Zugang haben können.

UC5	Man kann eine Raum schließen, und das Spiel wird nicht gewertet.
UC6	Wenn sich genug Spieler in einem Raum gefunden haben, kann ein Spiel gestartet werden.
UC7	Das ist der eigentliche Spielspaß. Alle Spieler werden gleich behandelt.
UC8	Jedes Spiel muss ein Ende haben. Bikes ist keine Ausnahme.
UC9	Man kann den Status aller Räume einsehen.

3.2. Technischer Kontext

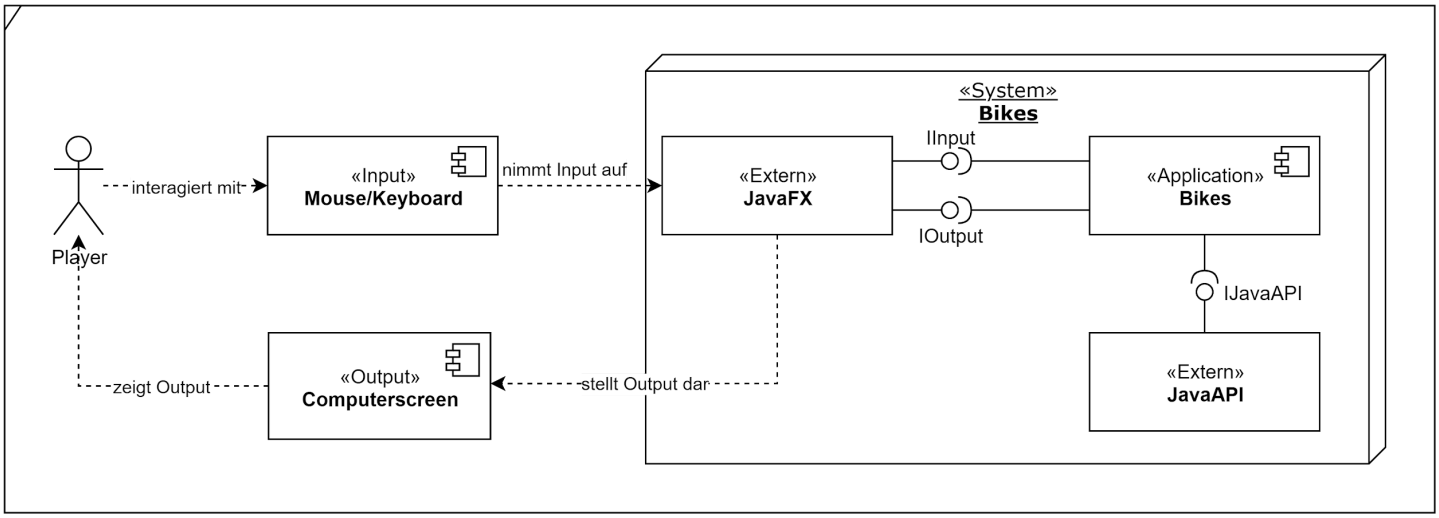


Figure 3. Technischer Kontext unserer Anwendung

Mit den ersten Überlegungen, wie der technische Kontext dieses Systems aussehen könnte, kann man sich überlegen welche Funktionalitäten noch nötig sind um es zu realisieren.

Interface	**Beschreibung**
IInput	Funktionen, die wir von JavaFX brauchen um Event Handling durchzuführen
IOutput	Funktionen, die für die Ausgabe am Bildschirm benötigt werden
IJavaAPI	Kann viele Funktionen beinhalten z.B. das gesamte java.util Package

4. Lösungsstrategie (Solution Strategy):

Um eine konkrete Lösungsstrategie zu entwickeln, um Bikes zu implementieren haben wir wie die meisten Gruppen eine Funktionsliste erstellt. Zudem haben sich in den letzten Abschnitten viele nötige Funktionalitäten herausgebildet, weshalb das Zusammenstellen der Funktionsliste ein wichtiger Schritt in der Anforderungsanalyse ist. In den Funktionen soll auch eine Abbildung auf unsere Qualitätsziele vorgenommen werden.

4.1. Funktionsliste

In der folgenden Tabelle sind eben viele Funktionen aufgelistet mit ein paar Beschreibungen. Das Eintragen geschah in Tandem mit der Bausteinsicht und Laufzeitsicht aus dem Entwurf Abschnitt unten mit den Usecases im Vordergrund. Irrelevante Methoden wie getter, setter und ähnliches wurden bewusst rausgelassen.

Akteur	Methodensignatur	Vorbedingung	Nachbedingung	Semantik
View	showOverlay(String overlay)	Spieler interagiert mit GUI	Zeigt einen neuen Screen/aktualisiert den aktuellen Screen	
View	checkInput(String input)	Ungültiger Text wurde eingegeben	Fehlernachricht als Alert Dialog wird angezeigt	Die Funktion soll dem Nutzer visualisieren, dass seine angeforderte Aktion nicht erfolgreich ausgeführt werden konnte
View	draw(List<Coordinate> trail, Color color)			Malt die Kacheln auf das Spielfeld mit der Farbe color
View	drawGame()			Malt den Hintergrund des Spielfelds
Controller	startGame()	Spiel ist noch nicht gestartet, 2 Spieler sind im Room	Spiel wird gestartet und angezeigt	
Controller	endGame()	Spiel ist bereits gestartet und Spieler gewinnt, beendet Anwendung oder verlässt Raum	Spiel wird beendet und Game Over Screen wird gezeigt	
Controller	restartGame()	Spiel wurde beendet	Startet das Spiel im selben Raum neu	
Controller	pauseGame()	Spiel wurde gestartet	Spiel wird pausiert bei einer Person (Q2) und zeigt einen Pause Screen	
Controller	resumeGame()	Spiel wurde pausiert	Spiel wird fortgesetzt	
Controller	registerKeyboardEvents()			Bindet die

				verschiedenen Input Möglichkeiten (Maus und Tastatur) an Funktionen
Controller	registerGameKeyboardEvents(Player Number playerNumber)			Bestimmt die Ingame Keyboard Funktionen
Controller	handlePressed<whatever>			Funktion aus dem Controller an das Model übergeben
Controller	handleSuccess()	Eine Funktion wurde asynchron aufgerufen		Ein Callback, das den Erfolg einer Methode signalisiert
Controller	handleFailed()	Eine Funktion wurde asynchron aufgerufen		Ein Callback, das den Fehlschlag einer Methode signalisiert
Controller	tick()	Spiel wurde gestartet		beinhaltet den Mainloop
Lobby	joinRoom(String roomName)	Raum ist geöffnet	Spieler tritt Raum bei	
Lobby	leaveRoom(String roomName)	Raum ist geöffnet	Spieler verlässt Raum	
Lobby	closeRoom(String roomName)	Raum ist geöffnet und soll geschlossen werden	Raum wird geschlossen und gelöscht	
Lobby	addRoom(IRoom room)		Fügt einen Raum zur Lobby hinzu	
Room	updateStatus()			Überprüft, ob alle Spieler im Raum ready sind
Room	addPlayer(String name, PlayerNumber playerNumber)		Fügt einen Spieler zum Raum hinzu	
Room	restartAt(PlayerNumber playerNumber)		Lässt einen Spieler zurücksetzen	
Room	hitOpponent()	Spieler ist in anderen Spieler reingefahren	Bestimmt den Gewinner und Verlierer	Checkt ob ein Bike in den Gegnertrail gefahren ist
Player	moveBike()		Nach einer gewissen Zeit(Zeit ist für alle gleich Q2), bewegt sich Bike in die gewollte Richtung	

Player	setPlayerStuff()			Setzt die passenden Infos zu PLAYER_1 und PLAYER_2
Player	isSuicidal()	Spieler ist in sich selbst gefahren	Bestimmt den Gewinner und Verlierer	Checkt ob ein Bike in den sich selbst gefahren ist
Player	droveIntoWall(int rows, int cols)	Spieler ist in Wand gefahren	Bestimmt den Gewinner und Verlierer	Checkt ob ein Bike in die Wand gefahren ist

Objektmenge	**Erläuterung**
Event	Eine Art Objekt, mit der wir herausfinden können, was angeklickt wurde, oder welche Taste gedrückt wurde
Direction	enum, z.B. UP, DOWN, LEFT, RIGHT
State	enum, z.B. LOBBY, START_GAME, GAME_OVER
Color	JavaFx Klasse, lässt sich als String de-/serialisieren
Coordinate	Aus view-library, immutable, beschreibt einen Punkt
Player	Unser Spielerobjekt mit Namen, Farbe...
PlayerNumber	identifiziert einen Player, es gibt bisher nur 2 also kann es z.B. 1 und 2 sein
Room	Objekt, welches 2 Spieler, einen Namen und boolean ob alle ready sind hält
Lobby	Objekt, welches alle aktuell existierenden Räume hält

Um die Objektmenge in der Middleware zu unterstützen wurden folgende Formate für JSON festgelegt (siehe in der Middleware Doku, Abschnitt 4.2 Message)

Coordinate:
message = {
“type”: “edu.cads.bai5.vsp.tron.view.Coordinate”,
“value”: [obj.x, obj.y]
}

Color:
message = {
“type”: “javafx.scene.paint.Color”,
“value”: obj.toString()
}
Deserialisierung mit Color.valueOf(String)

PlayerNumber:
message = {
“type”: “de.ashminh.bikes.application.model.room.PlayerNumber”,
“value”: obj
}
Deserialisierung bei allen Enum erfolgt über valueOf(String)

PlayerDto:
message = {
“type”: “de.ashminh.bikes.common.PlayerDto”,
“value”: [obj.coordinate, obj.colors]
}
Weil obj.colors vom Typ List<Color> ist, sieht die explizite Form so aus:
message = {
“type”: “de.ashminh.bikes.common.PlayerDto”,
“value”: [
[obj.coordinate.x, obj.coordinate.y],
[obj.colors[0].toString, obj.colors[1].toString, ...]
]
}

Direction:
message = {
“type”: “de.ashminh.bikes.application.model.room.Direction”,
“value”: obj
}

Da das Anzeigen der Overlays durch eine Funktion übernommen wird, wir aber die einzelnen Overlays betonen wollten, listet die untere Tabelle unsere Overlays separat von der obigen Funktionsliste auf.

Akteur	Methodensignatur	Vorbedingung	Nachbedingung	Semantik
View	showOverlay(“home”)	Die Anwendung wurde gestartet	Zeigt den Home Screen mit Join und Open Button	
View	showOverlay(“open”)	Spieler drückt Open Button	Zeigt den Open Screen mit Textfield und Open Room Button	
View	showOverlay(“wait”)	Spieler öffnet einen Raum	Zeigt einen Wait Screen mit Informationen und Ready Button	Wenn Spieler den Raum betritt wird der Wait Screen aktualisiert
View	showOverlay(“lobby”)	Spieler drückt Join Button	Zeigt Lobby Screen mit allen verfügbaren Räumen und Update Button	

View	showOverlay(“gameover”)	Ein Spiel wurde beendet	Zeigt einen GameOver Screen, der den Gewinner anzeigt	
View	showOverlay(“pause”)	Ein Spiel wurde pausiert	Zeigt einen Pause Screen mit Continue, Leave	

Viele unserer Funktionen können wir klaren Zuständigkeiten zuordnen. Mit den Akteuren in der Funktionsliste wollten wir andeuten, dass wir uns für die Model-View-Controller Referenzarchitektur entschieden haben, weil die Zuständigkeiten für uns sehr sinnvoll erscheinen und die Referenzarchitektur von Vielen genutzt wird.

Es gibt viele Funktionen, die einfach nur Objekte darstellen oder nur Teil der Spiellogik sind. Diese gehören dann jeweils zur View oder dem Model. Eine Komponente wie der Controller diese Funktionen orchestrieren.

Die Benutzerinteraktion wird über JavaFX bereitgestellt für die View. Die View leitet es dem Controller weiter, damit das Model reagieren kann.

Das Model kann mit dem Controller asynchron kommunizieren, in dem es seine states ändert. Der Controller updatet sich permanent, weil er den Gameloop beinhaltet, dann kann er auch alles ablesen.

5. Bausteinsicht (Block View):

5.1. Level 1

Folgend ist abgebildet, wie man MVC auf die Bikes Applikation anwenden könnte. Wir haben uns für eine Schichtenarchitektur entschieden, weil sie für uns einfacher nachzuvollziehen ist.

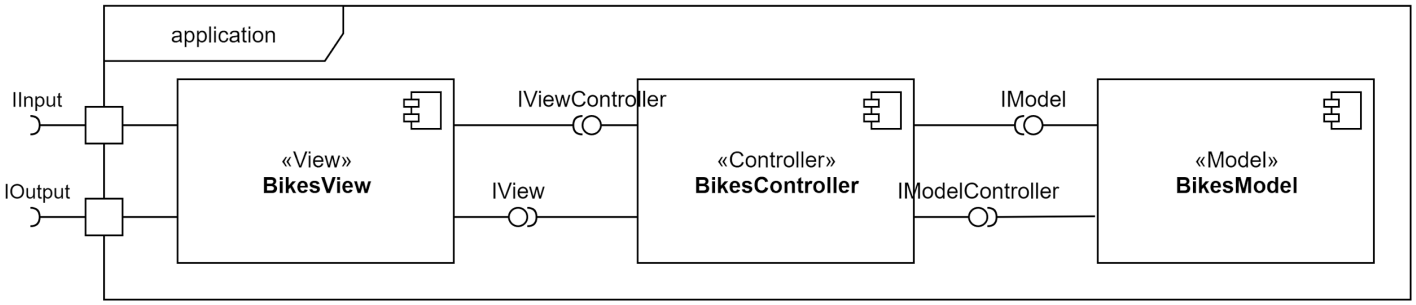


Figure 4. Komponentendiagramm unserer Bikes Anwendung

Interface	Funktionen
IView	showOverlay, hideOverlays, draw, drawGame
IViewController	tick, startGame, gameOver, setDirection, pauseGame, resumeGame

	Delegiert an Model mit: setDirection, handlePressedEscape, handleInputName, handlePressedJoin, handlePressedOpen,
IModelController	nur Callbacks: handleOpenRoomSuccessful, handleOpenRoomFailed handleJoinRoomSuccessful, handleJoinRoomFailed
IModel	Lobby Funktionen: openRoom, joinRoom, getAllRoomNames, closeRoom Room Funktionen: setDirection, getDirection, isCollided, moveBike, getTrail, getBikeFront, setPlayerName

5.2. Level 2

Unser Controller enthält vor allem viel Input Handling und Delegationsfunktionen. Er ist nicht weiter in mehrere Komponenten aufgeteilt. Folgend sieht man, dass diese View viele Overlays (GUI-Menüs) enthält, welche den Userinput an die Handling Funktionen der Schnittstelle IViewController übergeben.

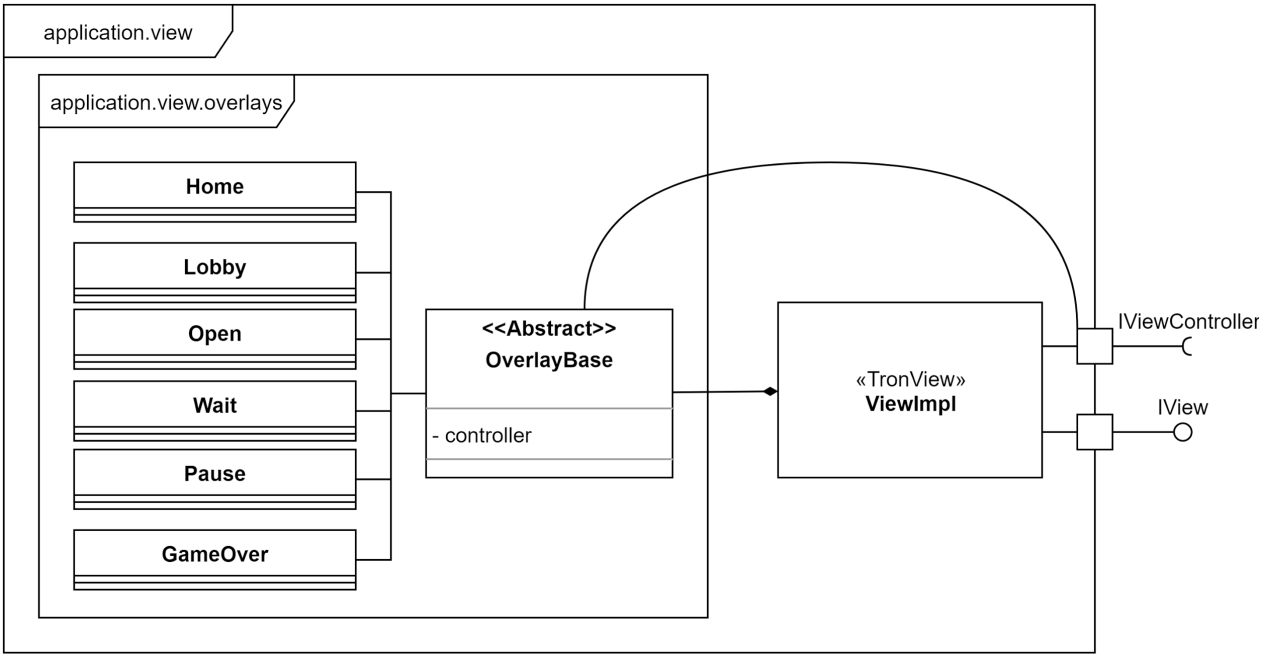


Figure 5. Zoom in die View

Das Model ist in drei Teile aufgeteilt. Die eigentliche Implementierung delegiert die Funktionsaufrufe an Lobby und Room. Das Room Objekt wird erst instanziiert, wenn man einen Raum in der Lobby öffnet, oder einem beitritt.

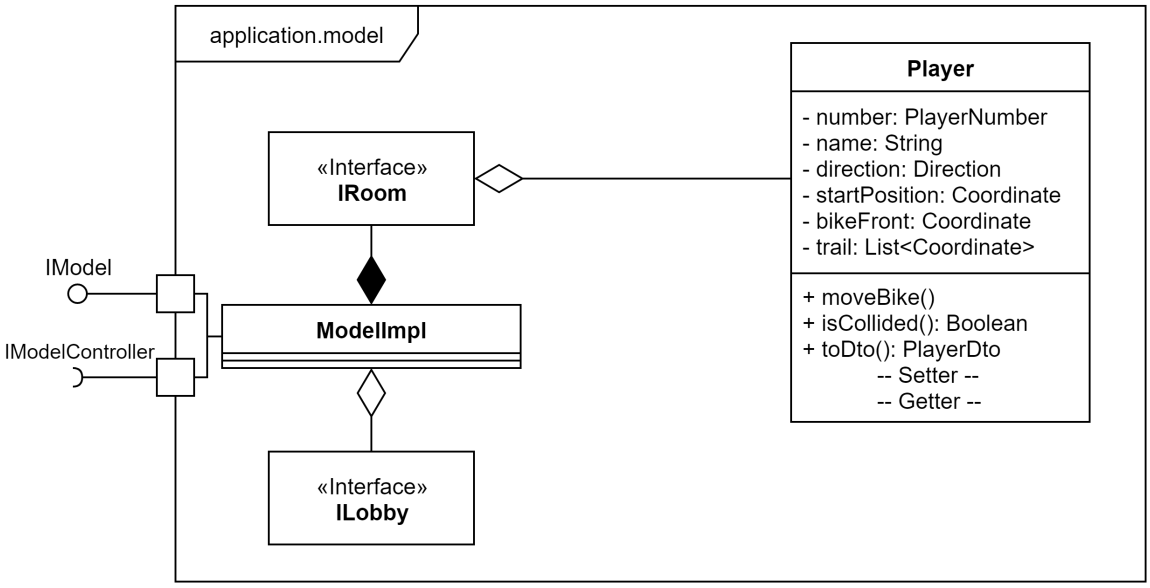


Figure 6. Zoom in das Model

Interface	Funktionen
IRoom	setDirection, getDirection, isCollided, moveBike, getTrail, getBikeFront, setPlayerName
ILobby	openRoom, joinRoom, getAllRoomNames, closeRoom

6. Laufzeitsicht (Runtime View):

- Zeigt konkrete Szenarien, wobei man nur die wichtigsten darstellen soll
- z.B. wichtige Use Cases, Interaktionen mit externen Schnittstellen, Start;Stop;Running, Error und Exceptions
- Als Sequenzdiagramm
    - Mögliche Szenarien: Raum öffnen, Raum beitreten, Raum schließen, Raum verlassen, Spiel spielen, Spiel pausieren

6.1. Use Cases auf Level 0

- **UC1: open app**
- **UC2: close app**
- **UC3: join**

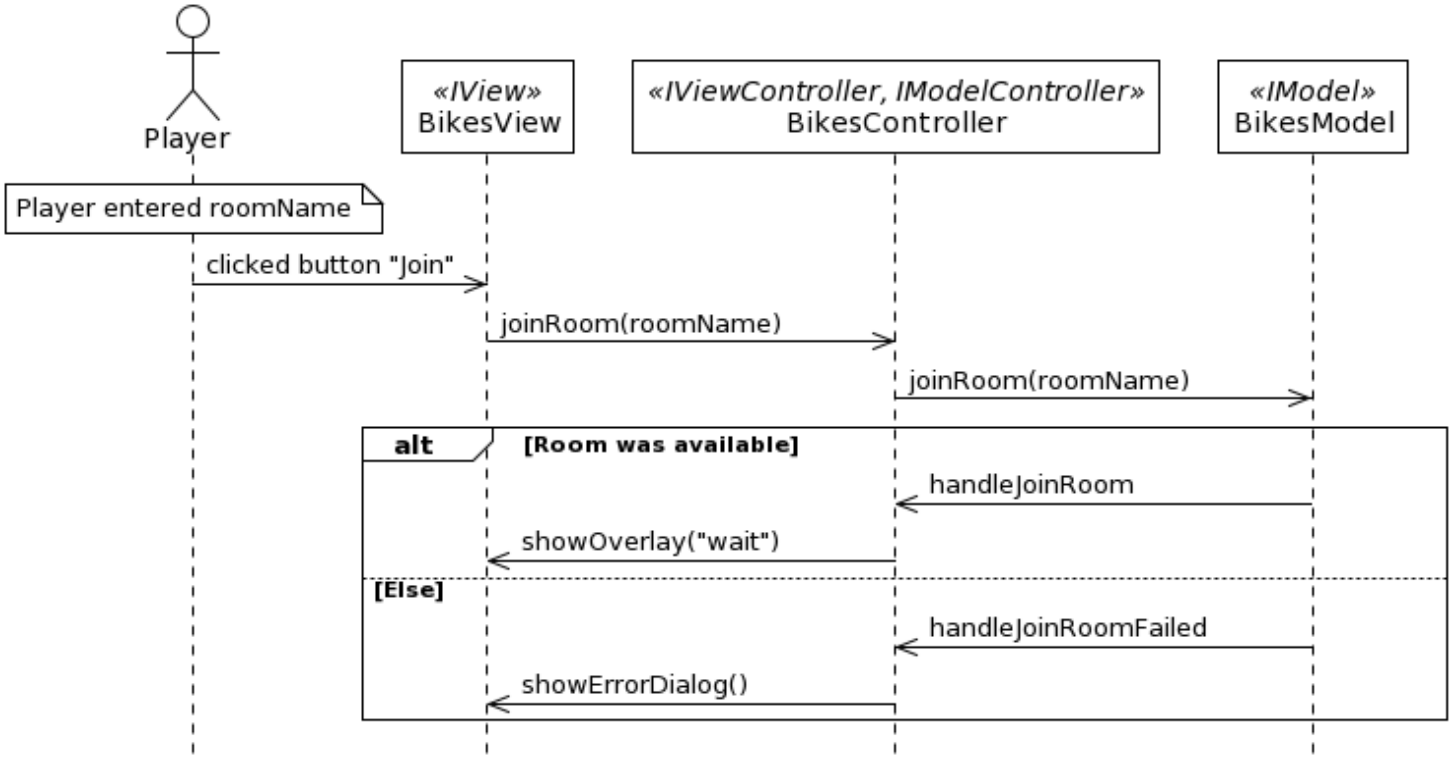


Figure 7. UC3: join

- **UC4: open**

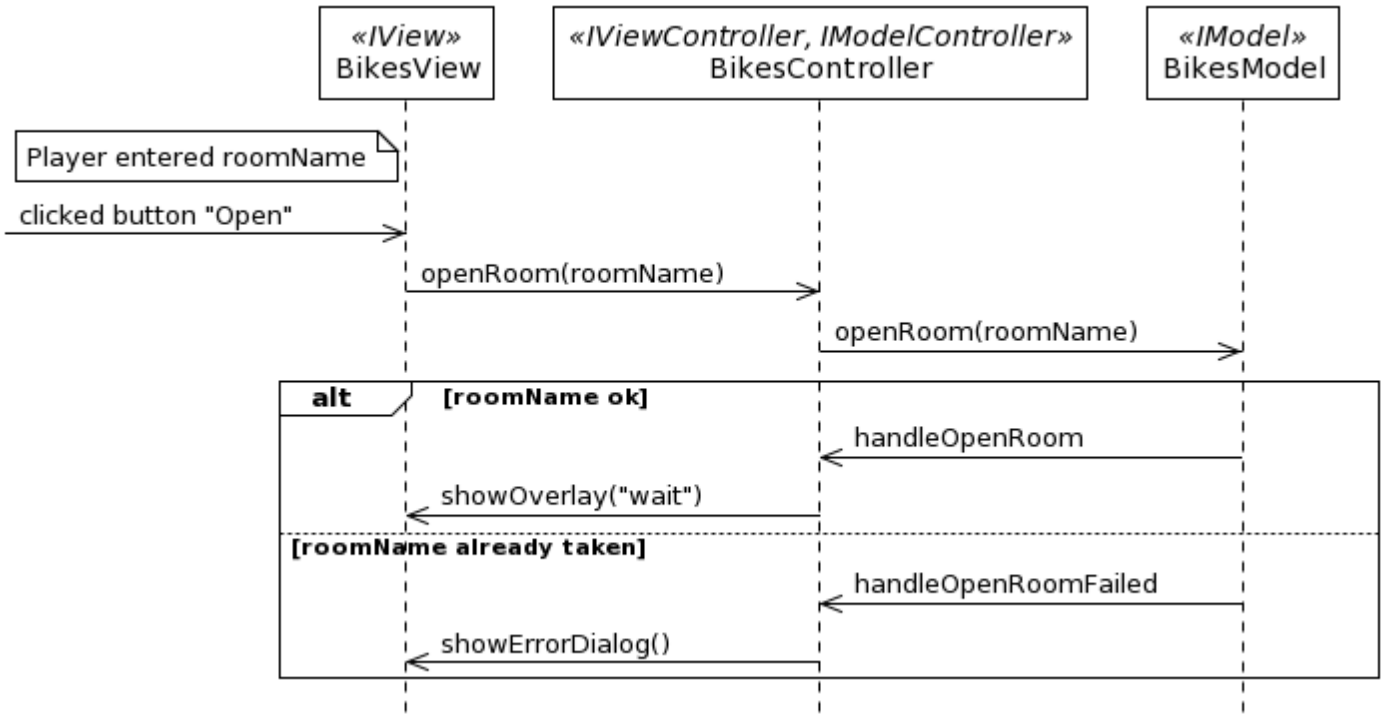


Figure 9. UC5: open

Wenn das Spiel verteilt gespielt wird, registriert sich das IRoom Objekt im ServerStub mit einem Namen, der eine Kombination aus Raumname und Klassenname ist.

- **UC5: close**

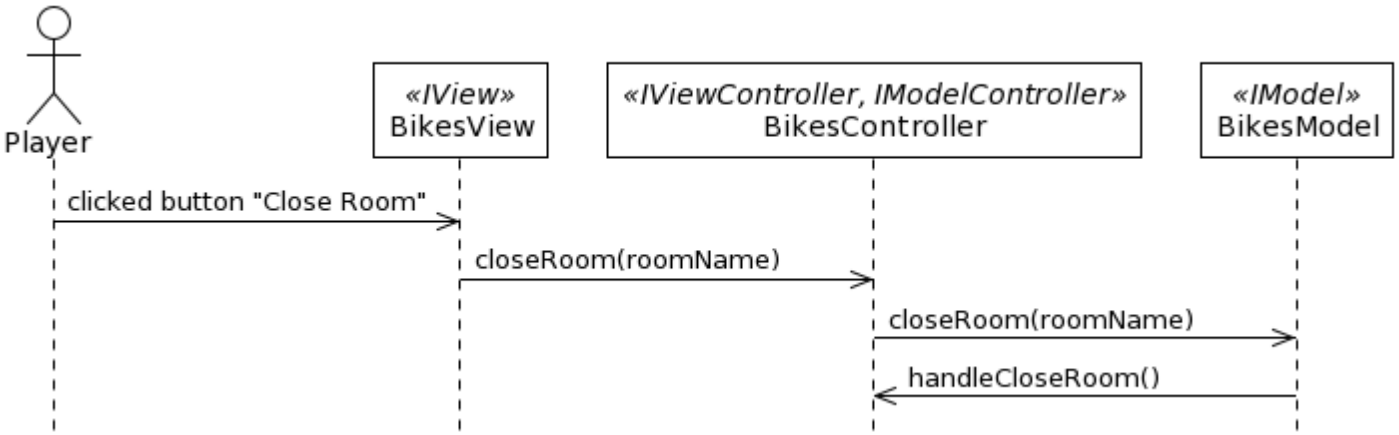


Figure 10. close

- **UC6:**
- **UC7: move bike**

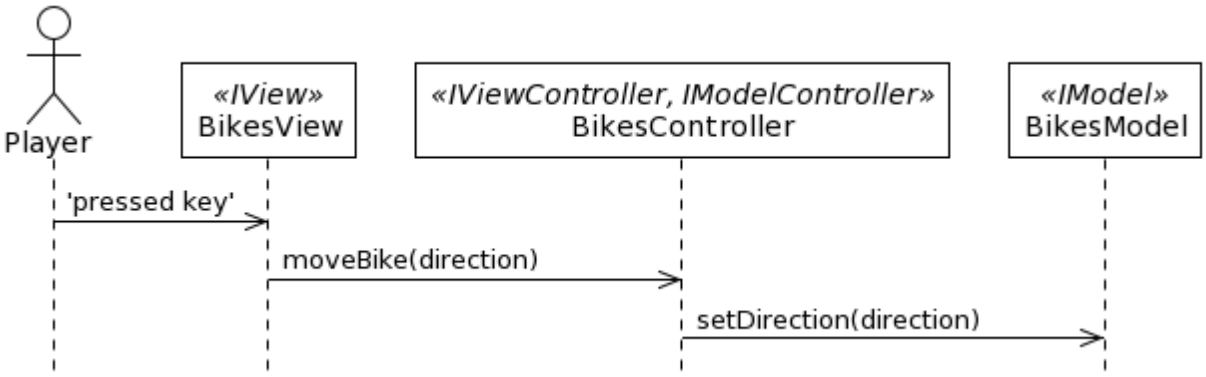


Figure 11. move bike

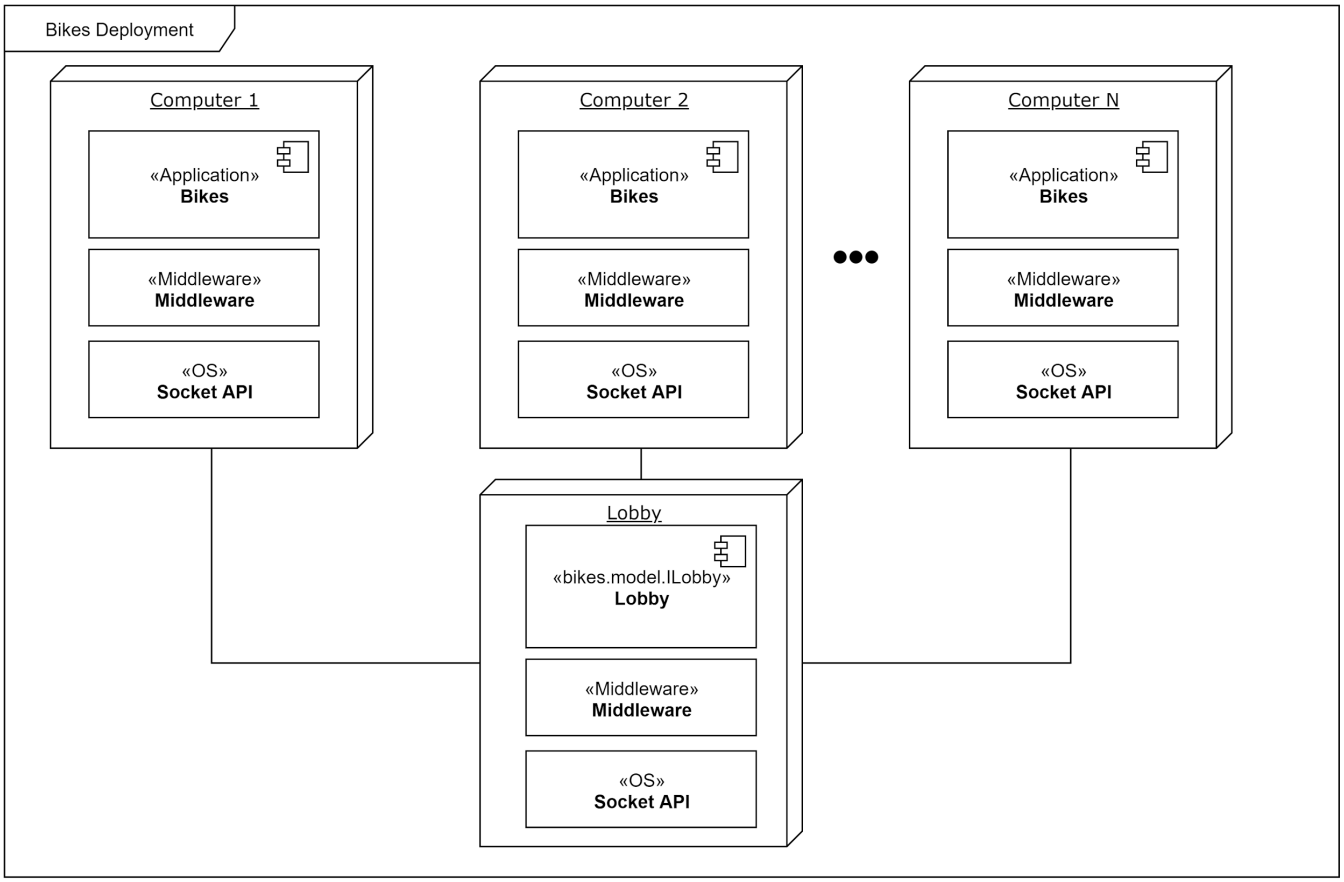
- **UC8:**

- **UC9:**

7. Verteilungssicht (Deployment View):

Bevor man miteinander spielen kann, muss auf einem Computer ein Lobby Objekt instanziiert sein. Dieses Objekt bietet seine Schnittstelle für jeden Spieler an. Wenn ein Spieler Bikes startet, kann er über die Lobby erstellte Räume ansehen, öffnen oder beitreten.

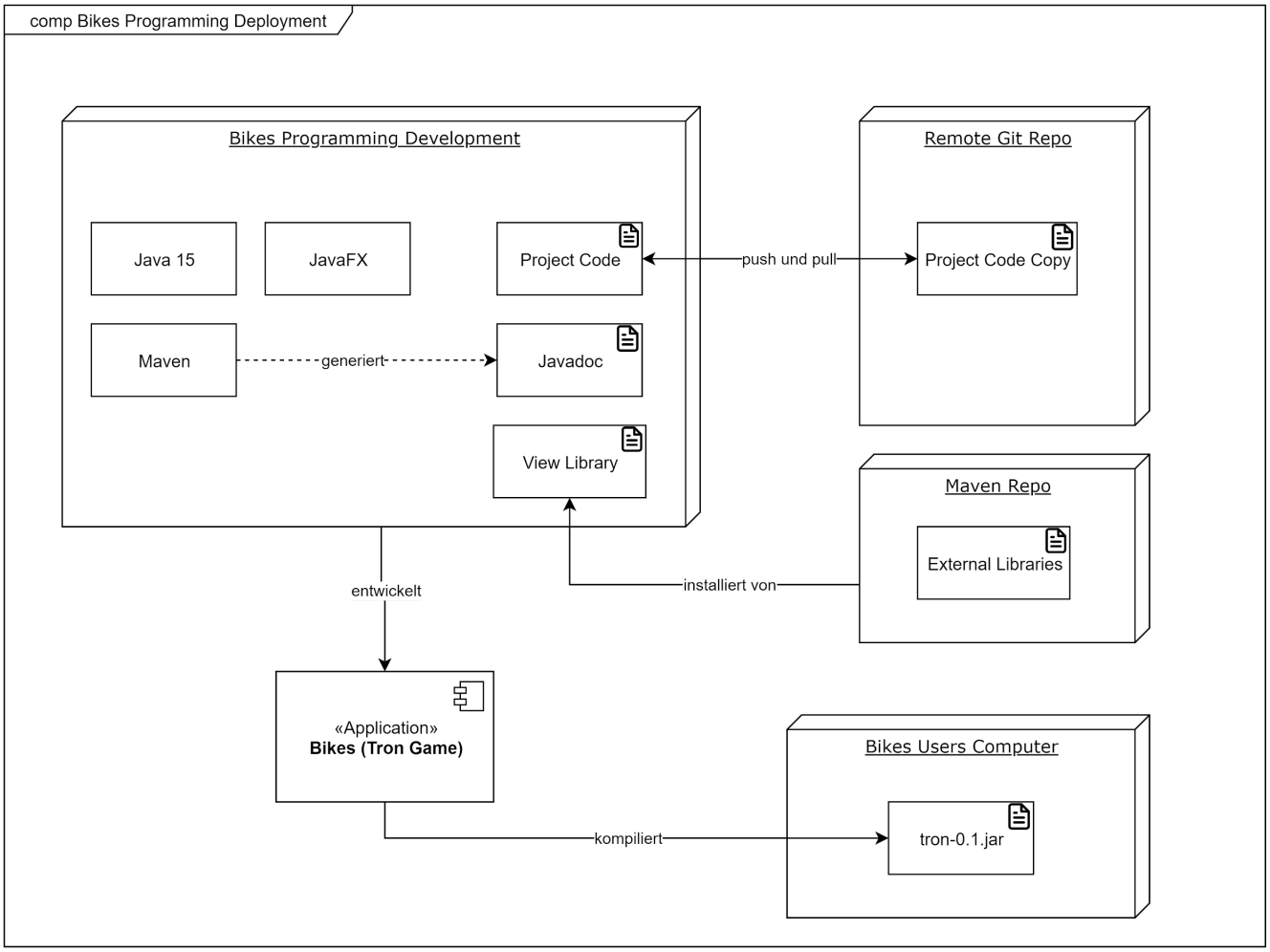
Beim Beitreten in einen Raum gibt die Lobby den Namen des Raums zurück, mit dem man einen IRoom Provider ansprechen kann. Im Model wird dann ein AppStubCaller Objekt instanziiert was die Aufrufe an die IRoom Schnittstelle über die Middleware an den entsprechenden ServerStub weiterleitet. Damit schneiden wir die Applikation in IModel direkt zwischen IRoom und ModelImpl.



Node	Description
Computer	Auf diesen Computern wird die Applikation gestartet
Lobby	Zentrales remote Lobby Object, das Dienste anbietet um 2 Spieler miteinander zu verbinden

Figure 12. Die laufenden Nodes aus Sicher der Applikation

Folgend als Ergänzung eine Verteilungssicht bezüglich unserer Programmierungsmethodik. Also mit was wir Programmieren, wo wir etwas speichern etc.



Node	Description
Bikes Programming Development	Hier wird unsere Anwendung entwickelt, programmiert in Java 15, Maven installiert
Bikes Users Computer	Hier wird die Anwendung beim Benutzer aufgerufen, als jar Datei
Maven Repo	Die externen, aus Maven installierten, Libraries kommen von hier
Remote Git Repo	Unser Git Repository, in welchem wir unser Projekt aufbewahren und zusammen dran arbeiten

Figure 13. Unsere Programmierverwaltung

8. Risiken und Schwächen:

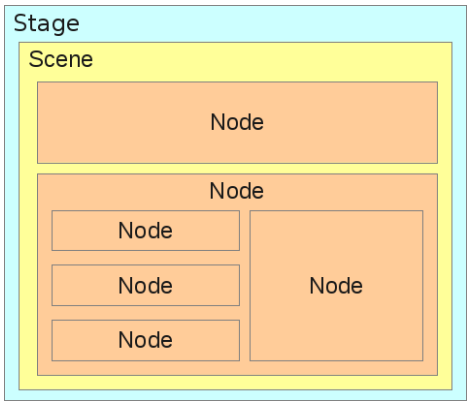
Folgend eine Liste an uns bekannten Risiken sowie Schwächen unserer Applikation. Einige der Schwächen wurden im Laufe des Praktikums entweder umgangen, gelöst oder konnten auf Grund von Zeitmangel nicht weiter behandelt werden und sind dementsprechend hier aufgelistet.

Risiko/Schwäche	Beschreibung

Nur 2 Spieler pro Raum	Aktuell ist es in unserer Anwendung nur möglich, dass 2 Spieler ein Spiel spielen können
Spieler gebunden zur Raumerstellung/beitretung	Spieler werden erst dann erstellt, wenn ein Raum geöffnet/beigetreten wird
Nur eine Lobby	ist ein Bottleneck

9. Sonstiges:

Aus dem Code gelernt:

- Ablauf: EntryPoint -> TronGame -> StartMenu -> printet ne Zeile und hided alle Overlays (Overlays sind im Prinzip verschiedene Screens, welche aufeinander gestacked werden können)
- GUI Zeug (unser Game ist wie ein Theaterstück aufgebaut):
    - Stage: Unser Hauptfenster, in welchem alles reinkommt, also unsere Bühne
    - Scene: Eine einzelne Szene
    - Node: Ein einzelnes Element, wie z.B. ein Knopf, also unser Schauspieler
    - Canvas: Ein Bereich auf dem gezeichnet werden kann
    - Layout Panes: Container, welche für flexible Arrangement von UI Elementen sorgen (z.B. automatische Änderung bei Fenstergrößenänderung
        - StackPane: Ein Stack mit Kindern an Szenen, Nodes etc.., die von vorne nach hinten, hier wird es base genannt und wir müssen brav unsere Elemente hier einfügen
        - VBox: Vertical Box, also von oben nach unten
        - HBox: Horizontal Box
    - 

    - overlays: eine HashMap mit jedem Screen drin, z.B. “fog” ist ein grauer Rectangle
- Coordinate: Unser Bike besteht im Prinzip aus einer Koordinate, wobei die restlichen die Trail sind
    - Bike ist also eine Liste aus Coordinates mit x und y

10. Glossar:

- Alle wichtigen Begriffe zur Verständigung mit den Stakeholdern hier klar definiert

Term	Definition
Bikes	Unser Tron Spiel heißt “Bikes” als Anlehnung an den simplen Namen “Snake”
Lobby	Eine Lobby, welche alle aktuell offenen Räume hält
Room	Ein virtueller Raum, in welchem 2 Spieler beitreten können, die dann gegeneinander spielen können
Player	Ein Spielerobjekt mit eigenem Namen, Farbe etc.
PlayerNumber	Entweder PLAYER_1 oder PLAYER_2
Direction	Direction des Bikes: LEFT, RIGHT, UP, DOWN
Front	Das Vorderteil eines Bikes
Trail	Die hinterlassene Spur eines Bikes
Overlay	Die gezeigten GUI für den Nutzer, z.B. Pause, GameOver...

11. Links:

- Whimsical Board (Wireframes): https://whimsical.com/bike-9Jux9y4piyyziTLWZNz65K
- Vorlage zu MVC: https://lh3.googleusercontent.com/proxy/zBI_-1GLI9dTL6bcNgoxrW6fhaszk2qrXD1anIgpTpIX246a1a75avJSNrdwWj_VBbkxfO7R7p2X_geeoc6PjkJELFVkC98iErQhKJosOZTfesrmAW1BRhW3J1_EbDPKyMHYci6rqAXu5Kc
- 4 Sichten draw.io Datei: https://drive.google.com/file/d/1HO3g81q3KP1Rrl1GRQMs47E4TFC_0Z5E/view?usp=sharing
- Bausteinsichten: https://drive.google.com/file/d/1FTZQEul05cbgINwkrSih-vdUZqa5Fe1x/view?usp=sharing
- Einfaches MVC Beispiel: https://www.codeproject.com/Articles/25057/Simple-Example-of-MVC-Model-View-Controller-Design