Abschlussbericht Softwareprojekt: PsychicFramework

Ulrich Bätjer André Henniger Markus Hempel Arne Herdick

Betreuer: Sebastian Nielebock, Robert Heumüller
Professor Ortmeier
13.2.2017

Inhaltsverzeichnis

1 Ziel

Das Ziel des Projekts war die Erstellung eines Frameworks zur Nutzung von Sensordaten von Androidgeräten auf javafähigen PCs und die Demonstration der Funktionalität dieses Frameworks anhand von drei Beispielanwendungen, nämlich einer Maussteuerung, einer Spielsteuerung und die Bedienung eines Murmellabyrinthes mithilfe eines Roboters. Mit dem von uns erstellten Framework soll Drittpersonen die Umsetzung von Projekten mit Sensordaten erheblich erleichtert werden.

2 Usecase-Analyse

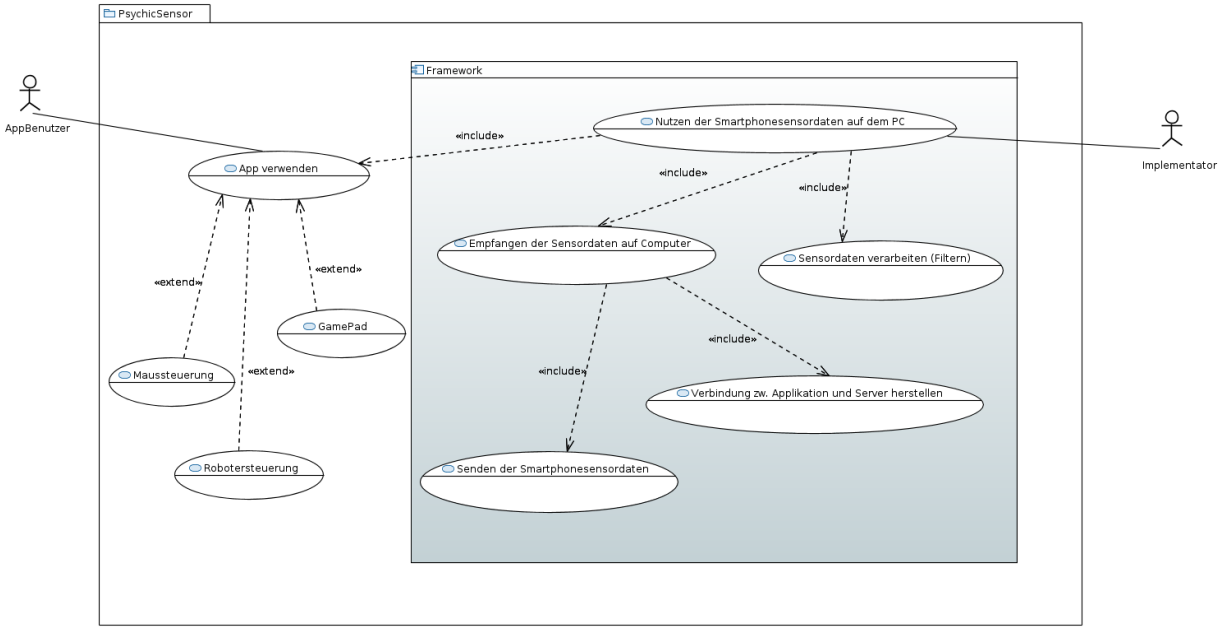

Abbildung 1: Usecase-Diagramm

Der AppBenutzer verwendet die App und einen vom Implementierer vorgefertigten Server, um Sensordaten auf eine bestimmte Art zu verwenden. Der Implementierer nutzt das Framework, um einen Server mit gewünschter Funktionalität umzusetzen, der dann von App-Benutzern verwendet werden kann. Hierzu muss das Senden und Empfangen der Sensordaten über eine Verbindung zwischen App und Server sowie das Verarbeiten dieser Daten berücksichtig werden.

Zur Demonstration dienen eine Maussteuerung, eine Robotersteuerung und ein Gamepad zur Spielsteuerung.

Die Maussteuerung soll eine normale Computer-Maus ersetzen können.

Die Robotersteuerung ist eine Proof-Of-Concept-Anwendung, die ermöglichen soll, mithilfe eines KUKA LBR iiwa 7 R800 ein Murmellabyrinth zu lösen. Ein Beispiel findet man in der Wikimedia Commons[1]. Das Murmellabyrinth soll dabei an der Werkzeugposition des Roboters befestigt sein, zum Beispiel mit einem Greifer.

Die Spielsteuerung sollte demonstrieren, dass es mit unserer Implementation möglich ist, Spiele zu steuern, die schnelle Reaktionen erfordern und dass unterschiedliche Sensortypen benutzt werden können. Wir haben uns für einen Controller für “Super Mario Kart” der Spielekonsole “Super Nintendo Entertainment System” entschieden. In diesem Spiel ist es erforderlich den gesteuerten Charakter nach Links und Rechts bewegen zu können. Außerdem gibt es die Möglichkeit Items zu werfen, und es gibt einen Mehrspielermodus.

[1] https://de.wikipedia.org/wiki/Datei:PuzzleOfDexterity.jpg

3 Anforderungen

Die zur Demonstration dienenden Anwendungen stellen schwer zu definierende Anforderungen an die Latenz und Frequenz der Sensordaten. Wir haben versucht, uns beim Festlegen der Grenzwerte auf bekannte Geräte zu beziehen, die keine Probleme bei der Bedienbarkeit haben. Im folgenden werden die von uns aufgestellten Anforderungen definiert.

3.1 Verbindungsqualität

3.1.1 Frequenz

Um eine Mindestanforderung für die Frequenz der Übertragung festzulegen, haben wir uns an Spielen für Konsolen orientiert, da diese vermutlich ähnliche Anforderungen an Reaktionsgeschwindigkeit stellen wie unsere Beispielimplementationen. Spiele scheinen auf aktuellen Konsolen, zum Beispiel der Xbox One und der PS4, mit mindestens 30 Bildern pro Sekunde zu laufen, siehe zum Beispiel diese Publikation von Ubisoft[ˆ2]. Deshalb haben wir die zu unterstützende Mindestfrequenz auf 30 Sensordaten pro Sekunde festgelegt. [ˆ2]: http://blog.ubi.com/watch-dogs-next-gen-game-resolution-dynamism/

3.1.2 Latenz

Eine akzeptable Grenze für die Latenz festzulegen war schwierig, da Latenzen im Millisekundenbereich nur schwerlich per Hand festzulegen sind. Wir haben daher versucht, uns über die Latenzen professionell hergestellter kabelloser Eingabegeräte zu informieren. Leider ist auch das schwierig, da Hersteller dazu meist keine Informationen veröffentlichen. Um dennoch eine maximale Latenz festlegen zu können, haben wir den Input-Lag von Konsolen untersucht. In diesem Artikel von www.eurogamer.net[ˆ1] werden die in Spielen auftretenden Input-Lags untersucht. Zwischen den Spielen wurde eine Differenz des Input-Lags von mehr als 50ms festgestellt. Da beide Spiele bedienbar sind, sollte eine Latenz unter diesem Wert keine Probleme verursachen. [ˆ1]: http://www.eurogamer.net/articles/digitalfoundry-lag-factor-article?page=2

3.1.3 Jitter

Ein dritter Parameter für die Verbindungsqualität ist der Jitter, das heißt wie sehr sich die Periodizität der ankommenden Sensordaten von der Periodizität der gesendeten Sensordaten unterscheidet. Je geringer der Jitter ist, desto besser ist die Verbindung. Unsere einzige Anforderung an diesen Aspekt der Verbindung war, dass sich kein Jitter bemerkbar macht.

3.2 Maussteuerung

Zusätzlich zu den Anforderungen an die Netzwerkparameter, die flüssige und direkte Steuerung garantieren sollen, muss die Maus zumindest auch einen Links- und Rechtsklick zur Verfügung stellen, um eine normale Maus zu emulieren.

3.3 Spielsteuerung

Für die Spielsteuerung haben wir einen SNES-Controller emuliert, mit dem wir das Spiel „Super Mario Kart“ spielen können. Wir haben uns für dieses Spiel aufgrund der geringen geforderten Button-Anzahl entschieden, weil es ohne haptisches Feedback schwierig ist Knöpfe zu treffen und sich auf das Spiel zu konzentrieren. Um die Anforderungen an die Spielbarkeit festzulegen, haben wir die Zeiten einiger Läufe mit nativen Controllern auf der ersten Karte des Spiels, “Mario Circuit 1”, gemessen. Dadurch hatten wir einen Vergleichswert von

1:20, die wir mit unserem Controller mindestens erreichen wollten, bei der dieser einem nativen Controller ähnlich ist.

Außerdem sollte die Anordnung der Buttons in etwa dem nativen Controller entsprechen, weshalb ein flexibles Layouting vom Server aus möglich sein muss.

Um die Itemmechanik von "Super Mario Kart" zu unterstützen, wollten wir lineare Bewegungen in einer separaten Achse auswerten, was die gleichzeitige Nutzung mehrerer Sensoren erforderte.

Da wir den Mehrspielermodus des Spiels ebenfalls nutzen wollten, musste die Verbindung mehrerer Clients zur gleichen Zeit unterstützt werden.

3.4 Robotersteuerung

Die Robotersteuerung stellte keine Anforderungen, die nicht bereits durch die Maussteuerung und der Spielsteuerung gestellt wurden, da die Anforderungen an Latenz und Frequenz nicht höher sind, und Buttons auch schon von der Maus- und Spielsteuerung benötigt werden.

3.5 Wiederverwendbarkeit

Da wir ein entwicklerfreundliches Framework erstellen wollten, mussten wir darauf achten dass unser Projekt nicht nur für unsere Beispiele nutzbar ist. Es sollte nicht notwendig sein, die App zu verändern, um andere Applikationen zu entwickeln. Wichtig war auch, alle möglichen Anforderungen an die Nachbearbeitung der Daten auf dem Server unterstützen zu können.

Außerdem wollten wir die Anforderungen an den PC für den Server und das Handy für die App möglichst gering halten.

3.6 Bedienbarkeit

Die App sollte möglichst benutzerfreundlich erstellt sein. Das bedeutete für uns, möglichst wenig Konfiguration vom Nutzer zu fordern.

4 Analyse unserer Ergebnisse

Aus den Requirements ist ein Client-Server-Framework entstanden. Die Clients laufen in einer App auf Android-Geräten, und die Server sind Java-Anwendungen auf PCs. Mit unserem Framework können andere Entwickler einfach Daten nutzen, die Sensoren der Clients generieren und an die Server schicken, indem sie eine Java-Klasse erweitern. Im Folgenden stellen wir unsere Ergebnisse kurz vor und vergleichen sie mit den gestellten Requirements.

4.1 Frequenz

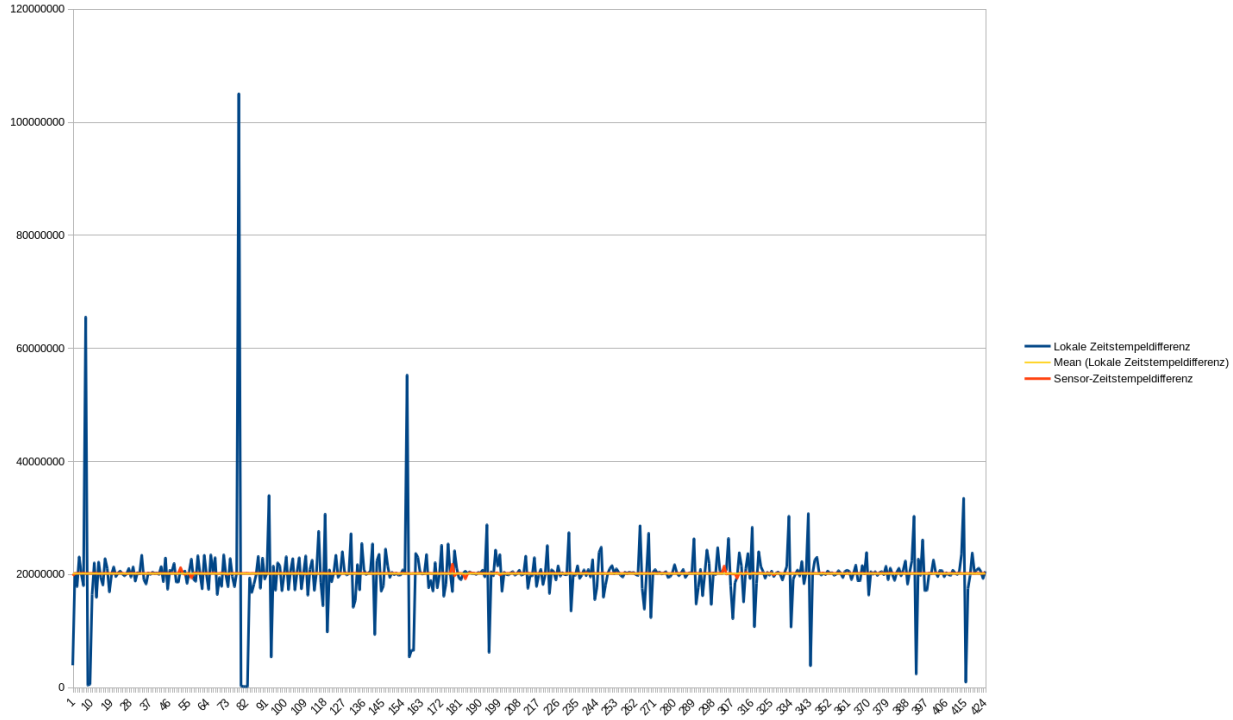

Abbildung 2: Frequenz-Ergebnisgraph

Die y-Achse ist in Nanosekunden angegeben, die x-Achse gibt die Nummer der Messung seit ihrem Beginn an. Da sich die Timestamps des Handys und des Servers nie exakt synchronisieren lassen, lässt sich hier nur die Differenz zwischen den Ankunftszeiten auf dem PC, und den Sensor-Event-Timestamps auf dem Handy berechnen. Wie zu sehen ist, ist der Durchschnitt der Abstände zwischen dem Ankommen von Sensordaten ungefähr 20ms, was in einer Frequenz von 50Hz resultiert. Da 50Hz unsere Anforderung von 30Hz deutlich überschreitet, sehen wir unsere Anforderung an die Updatefrequenz als mehr als erfüllt an.

4.2 Latenz

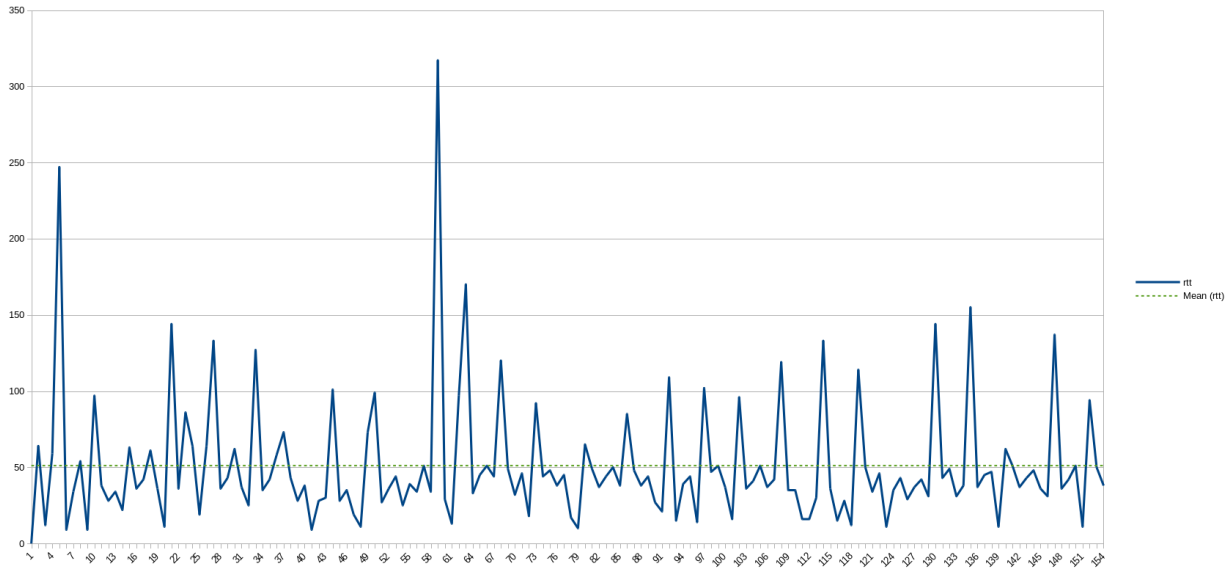

Abbildung 3: RTT-Ergebnisgraph

Die y-Achse ist in Millisekunden angegeben, die x-Achse gibt die Nummer der Messung seit ihrem Beginn an. Da sich die Timestamps des Handys und des Servers nie exakt synchronisieren lassen, haben wir die Round-Trip-Time gemessen. Wie zu sehen ist, liegt die durchschnittliche Round-Trip-Time bei 50ms; die Latenz zwischen Generierung der Sensordaten wird dementsprechend ungefähr bei 25 Millisekunden liegen. Wir sehen unsere Anforderung an die Latenz damit erfüllt.

4.3 Jitter

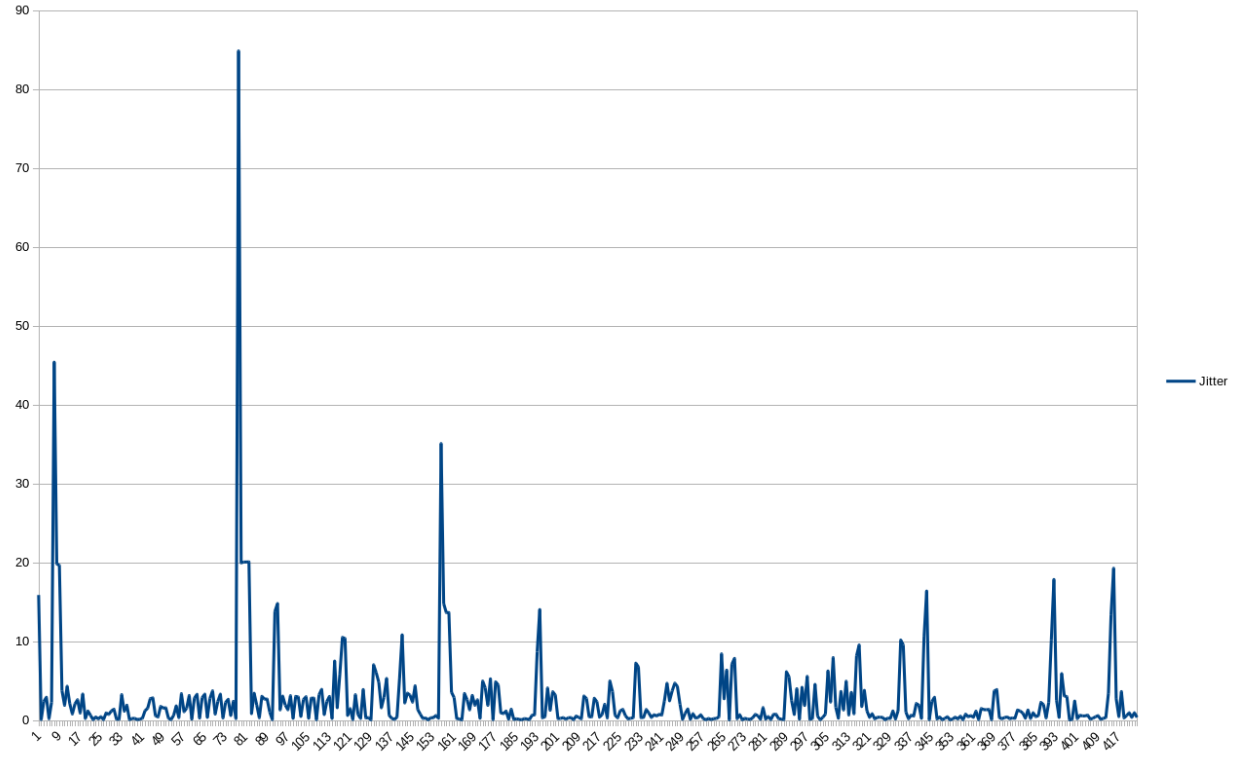

Abbildung 4: Jitter

Die y-Achse ist in Millisekunden angegeben, die x-Achse gibt die Nummer der Messung seit ihrem Beginn an. Wie zu sehen ist, bleibt die Differenz zwischen den Periodizitäten zumeist unter 10ms. In der Praxis waren diese Differenzen nie zu spüren. Da die Zeitstempel der Sensordaten übertragen werden, ist es auch möglich verspätete Pakete zu ignorieren, insofern stellt Jitter für uns kein Problem dar.

4.4 Anforderungen der drei Steuerungen

Unsere Anforderungen an die Netzwerkverbindungsqualität haben die Mindestanforderungen der drei Beispielimplementationen erreicht oder übertroffen, da die Steuerungen keine Probleme mit schlechten Reaktionszeiten zeigen.

4.4.1 Maussteuerung

Mit den Buttons in der App ist es möglich, alle Funktionen einer einfachen Maus zu ersetzen: Man kann die Maus bewegen, auf Elemente des Bildschirms klicken, und klicken und ziehen, so dass auch das Scrollen durch Inhalte problemlos möglich ist

4.5 Spielsteuerung

Nach ein wenig Eingewöhnungszeit ist es uns mit unserem Controller routinemäßig gelungen, die geforderten 1:20 zu unterbieten; es ist außerdem möglich, den Zwei-Spieler-Modus mit zwei Handys zu spielen. Die Itemwurfmechanik wird ebenfalls sowohl für rückwärts als auch für vorwärtsgewandte Würfe unterstützt.

4.6 Robotersteuerung

Da wir unsere Steuerung leider nur in der Simulationsumgebung von V-REP testen können, können wir die erfolgreiche Lösung eines Murmellabyrinthes nicht testen. Wir gehen jedoch aufgrund unserer Erfahrungen mit der Maus- und Spielsteuerung sowie den Ergebnissen der Simulation davon aus, dass es möglich ist, ein Murmellabyrinth mit unserer Robotersteuerung zu lösen.

4.7 Wiederverwendbarkeit

Letztlich ist es sehr einfach geworden, einen neuen Server zu implementieren, und es existiert eine umfassende Dokumentation sowie drei Beispiele, die bei einer neuen Implementation zu Hilfe stehen. Mit unserer Unterstützung fast aller Sensoren sowie flexibler Buttonlayouts können unterschiedlichste Anwendungen entwickelt werden. Durch unsere Wahl Javas als Implementationssprache laufen Server auf Windows, Linux und Apple-Geräten, die JDK 8 oder höher installiert haben. Um die Verbindung zu einem Handy aufzubauen, müssen sich Server und Client im gleichen WLAN-Netzwerk mit Zwischenclientkommunikation befinden, ein Zustand, der in den meisten Heimnetzwerken gegeben ist. Alle Android-Geräte mit Unterstützung für API-Level 19, also Geräte mit mindestens Android 4.4 können die PsychicSensors-App installieren und werden somit als Client unterstützt.

Da wir uns gegen eine native Implementation von Eingabemethoden, zum Beispiel über Blutooth HID-Profile, entschieden haben, ist der Implementierer nur durch die Java-Umgebung beschränkt, so lange er die gewünschten Funktionen nicht in einer nativen Programmiersprache umsetzt.

Wir sehen daher unsere Anforderung an das Framework, nicht nur von uns genutzt werden zu können, als erfüllt an.

4.8 Bedienbarkeit der App

Das User Interface der App ist recht minimalistisch gehalten. Mit einem Knopfdruck werden Server gesucht, mit einem weiteren wird die Verbindung zu einem der gefundenen Server aufgebaut. Es ist dem Nutzer möglich, eine Empfindlichkeit für die Sensoren festzulegen, die der Server dann je nach Anwendung interpretieren kann. Falls der Standard-Discovery-Port auf dem PC auf dem der Server läuft blockiert ist, kann dieser in der App geändert werden. Insgesamt erfüllt die App unsere Anforderungen an den Client, auch für gelegentliche Nutzer verständlich zu sein, sehr gut.

5 Projektverlauf

In diesem Abschnitt beschreiben wir den Verlauf des Projektes und unseren Entwicklungsprozess.

5.1 Projektplan

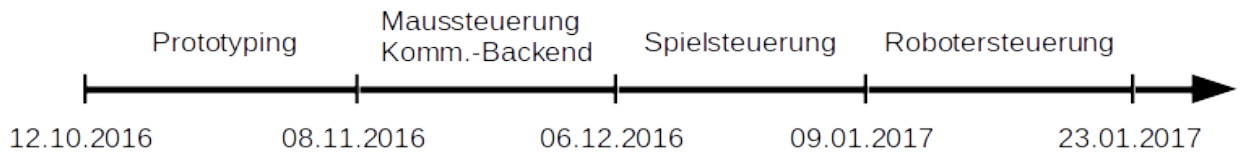

Abbildung 5: Entwicklungszeitstrahl

5.1.1 Prototyping

Um eine geeignete Übertragungsart der Sensordaten zu finden und vertraut mit dem Server-Client-Konzept zu werden wurden Prototypen erstellt. Hierbei wurden mehrere Varianten getestet: zum einem ob es besser ist Daten mit TCP oder UDP zu senden und zum anderen wurde verglichen, ob es besser ist den Server auf dem Smartphone oder auf dem PC zu betreiben.

5.1.2 Mouseserver & Kommunikations-Backend

Es wurde das Kommunikations-Backend erstellt, das die Kommunikation zwischen Server und Clients sicherstellt. Aufgrund der Zeitkritikalität und Indifferenz gegenüber fehlenden Paketen der Sensordaten haben wir uns hier für UDP entschieden. Der Befehlsaustausch erfolgt über TCP.

Parallel zur Fertigstellung des Kommunikations-Backends wurde eine Maussteuerung per Smartphone implementiert. Hierbei werden Gyroskopdaten verwendet und mit Hilfe der von Java vorgegebnen Robot-Klasse Inputs emuliert.

5.1.3 Spielsteuerung

Für die Umsetzung einer Spielsteuerung haben wir uns für Super Mario Kart auf dem Super Nintendo Entertainment System entschieden, da hier wenige Buttons benötigt werden und die Möglichkeit der Verwendung von mehr als einem Sensor besteht. Der Gravitationsensor wird zur Steuerung der Lenkbewegung und der lineare Beschleunigungssensor zur Steuerung des Itemwurfes verwendet.

5.1.4 Robotersteuerung

Zur Lösung eines Murmellabyrinths mit Hilfe des Gravitationsensors des Smartphones wurde eine Steuerung für den KUKA LBR iiwa R800 entworfen.

5.2 Entwicklungsprozess

Zur Entwicklung der Applikation wurde Android Studio und zur Entwicklung des Servers wurde IntelliJ IDEA verwendet. Als Version Control System diente Git.

Aus am Anfang erstellten Use Cases wurden benötigte Funktionen und Subfunktionen abgeleitet. Diese wurden anfangs in JIRA eingetragen und nach und nach abgearbeitet. Bei Auftreten eines Problems oder einer fehlenden Funktionalität wurde ein neuer Issue erstellt und hinzugefügt.

Wir haben uns keine Aufteilung der Aufgaben festgelegt. Issues wurden nach ihrer Dringlichkeit im Projektverlauf und der Länge ihrer Bestandszeit geordnet abgearbeitet.

6 Lessons learned

Es ist deutlich effektiver sich gemeinsam zu festen Zeiten zu treffen, um zusammen zu arbeiten als allein. Dadurch haben wir direktes Feedback zu neuen Ideen und deren Umsetzungen von den anderen Teammitgliedern bekommen. Pairprogramming erlaubt Diskussionen und konstruktive Lösungen von neuen Problemen durch andere zu erhalten. Uns fiel es schwer Grenzwerte für messbare Requirements zu erstellen, die das Projekt beschreiben. Ein frühzeitiges Festlegen des deployment formats verhindert die Benutzung von Softwarearchitekturen, die eine spätere Bereitstellung der Software erschweren.

7 Architektur

Das Psychic-Framework ist dreigeteilt in die App, den Server und einen gemeinsamen Kern. In dem Teil, der den Server enthält, befindet sich außerdem die Datenpipeline.

7.1 Gemeinsamer Kern

Im gemeinsamen Kern (dem `common`-Package) sind alle Klassen enthalten, die sowohl vom Server als auch von der App benötigt werden. Das sind unter anderem Klassen wie zum Beispiel `SensorData`, `SensorType` oder `AbstractCommand`, die in der Kommunikation verwendet werden, aber auch Klassen die vom Client und vom Server separat benutzt werden, z.B. `ConnectionWatch` oder `DiscoveryThread`.

7.2 Architektur der App

Die App ist dafür zuständig, den Nutzer Server finden zu lassen und mit Servern zu kommunizieren. Sie erlaubt es dem Nutzer außerdem, die Sensitivität für alle Sensoren zu ändern und den Gerätenamen festzulegen. Wenn der Nutzer sich dafür entschieden hat, mit einem Server in Verbindung zu treten, wird die App anfangen, die Daten der angefragten Sensoren zu übermitteln. Außerdem stellt sie die vom Implementierer angefragten Buttons dar, und Benachrichtigt den Server über Knopfdrücke.

Die App startet in der `DiscoveryActivity`. Hier wird durch den `DiscoveryClient` ein Broadcast auf einem DiscoveryPort nach verfügbaren Servern durchgeführt (Server-Discovery). Bei Fund eines Servers wird dieser in der Activity angezeigt. Durch Click des Servers wird in die `SendActivity` übergegangen.

Diese Activity instanziiert `NetworkClient`, durch den eine Initialisierung der `DataConnection` und `CommandConnection` erfolgt. Das serverspezifische Buttonlayout wird über die `CommandConnection` erhalten und mit Hilfe des `LayoutParser` umgesetzt. Die Verbindung zum Server kann durch das Betätigen des Disconnectbuttons beendet werden und man gelangt wieder in die `DiscoveryActivity`.

Von beiden vorher genannten Activities gelangt man durch einen Knopfdruck in die `OptionsActivity`. Hier lassen sich Sensitivitätseinstellungen der einzelnen Sensoren vornehmen. Durch Betätigen des “Zurück”-Buttons gelangt man in die Activity, aus der die OptionsActivity aufgerufen wurde.

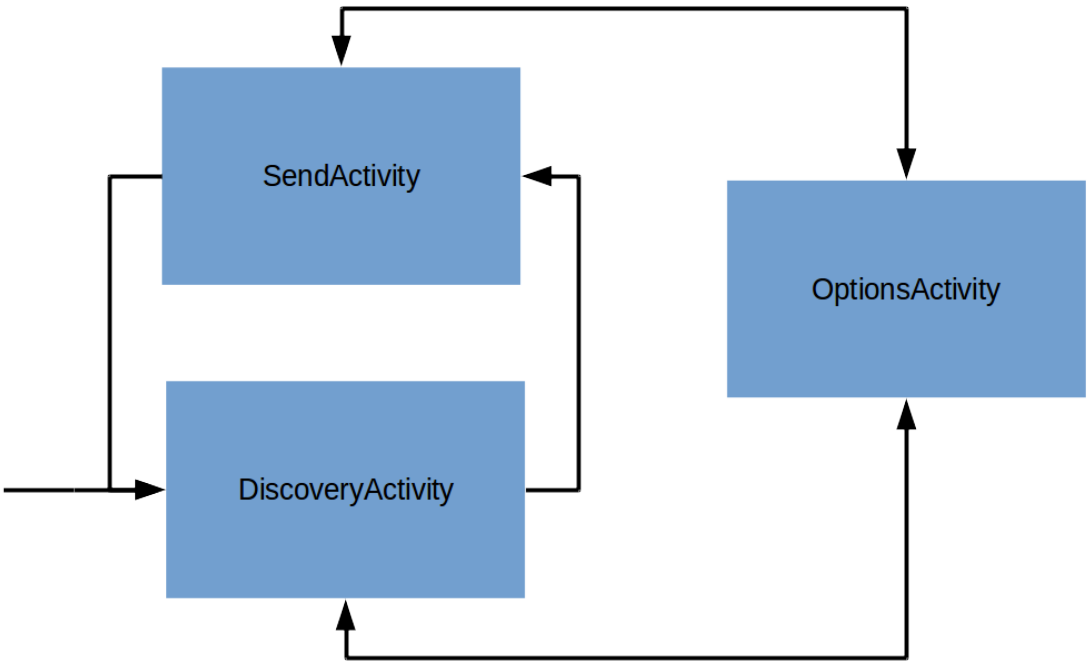

Abbildung 6: Konzept der App

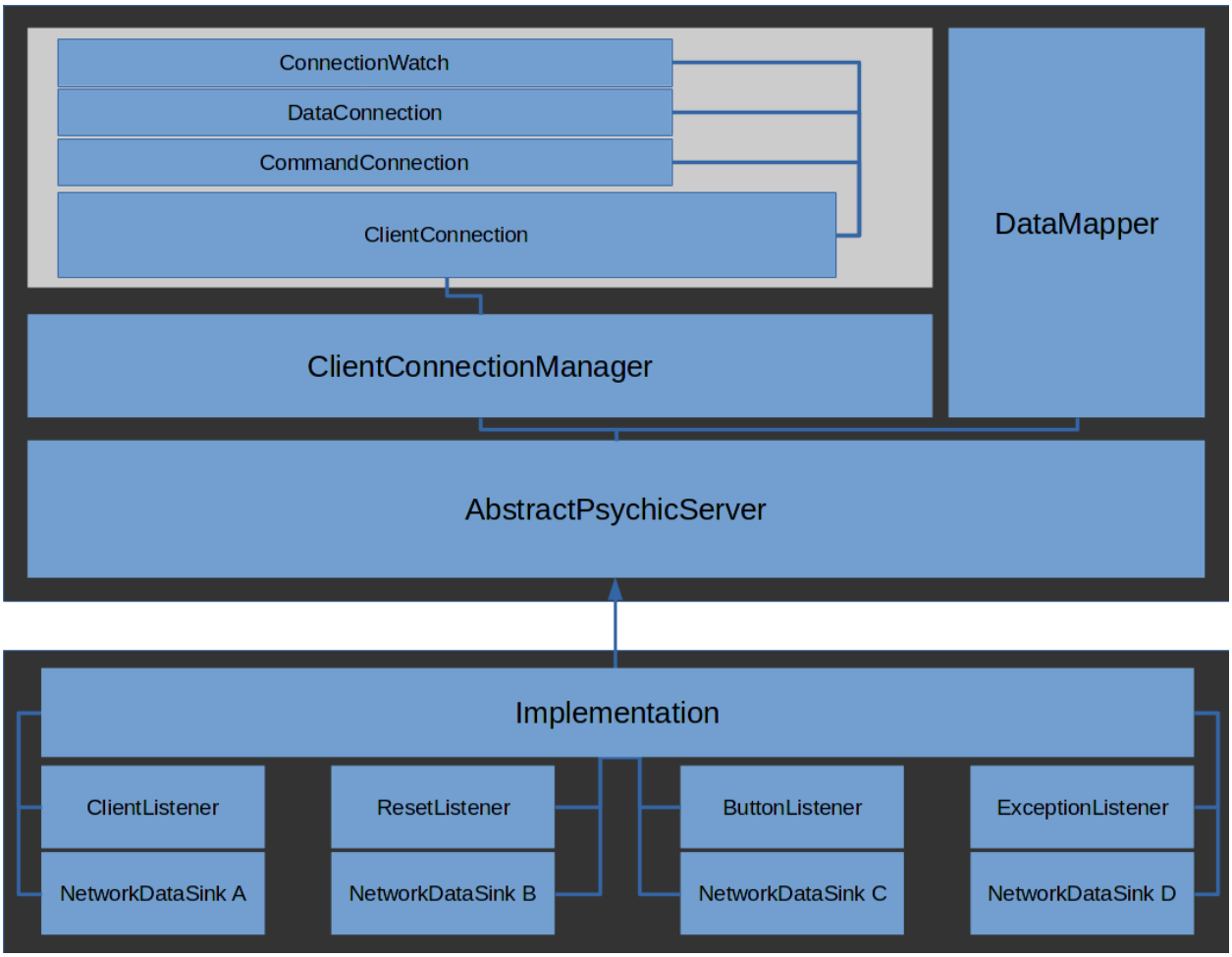

Abbildung 7: Architektur des Servers

7.3 Architektur des Servers

Die Architektur des Servers kann geteilt werden in den `PsychicServer`-Teil, der die von uns geschriebenen Klassen beinhaltet, und den Teil, den der Implementierer erstellen muss. Der Server-Teil des Psychic-Frameworks dient dazu, die Erstellung von neuen Servern möglichst einfach zu machen. Beispiele lassen befinden sich im `examples`-Package.

7.3.1 PsychicServer-Teil

7.3.1.1 AbstractPsychicServer

Die Hauptklasse unserer Seite ist der `AbstractPsychicServer`. Diese Klasse beinhaltet fast alle Funktionen, mit denen der Implementierer interagiert, zum Beispiel um Datensenken für Sensoren zu registrieren. Die Klasse beinhaltet eine `DataMapper`-Instanz, der von den `ClientConnection`-Instanzen weitergeleitete Daten nach Client und Sensortyp aufgeschlüsselt an die registrierten Datensenken weiterleitet. Hier werden auch die Kontrollnachrichten, die nicht in der `ClientConnection` behandelt werden, behandelt. Das beinhaltet Knopfdrücke aller Clients, Resetevents aller Clients und Verbindungsanfragen, derer dann die Listenerimplementationen des Implementierers benachrichtigt werden.

7.3.1.2 ClientConnection

Jeder Client wird von einer `ClientConnection`-Instanz verwaltet. Jede dieser Instanzen wiederum besitzt eine `DataConnection`- und eine `CommandConnection`-Instanz. Die `CommandConnection` implementiert das Senden und Empfangen von `AbstractCommand`-Objekten, und die `DataConnection` nimmt alle Sensordaten des Clients an. Die `ClientConnection` wird über Callbacks von Empfangsereignissen benachrichtigt. Die empfangenen Sensordaten werden mit der korrekten Nutzersensitivität an den `DataMapper` weitergeleitet. Mit Ausnahme von wenigen Kontrollnachrichten, die in der `ClientConnection` behandelt werden können, wie zum Beispiel die Nachrichten, die die Nutzersensitivität enthalten, werden alle an den `AbstractPsychicServer` weitergeleitet. Zusätzlich zu den Nutzersensitivitäten speichert jede `ClientConnection` auch den Wertebereich der Sensoren des verbundenen Handys.

7.3.1.3 ClientConnectionManager

Der ClientConnectionManager verwaltet alle `ClientConnection`-Instanzen. Er speichert die vom Implementierer vorgegebenen Sensorengeschwindigkeiten, die Knopfkonfiguration und die benötigten Sensoren. Verändert sich der Zustand dieser Anforderungen, benachrichtigt der `ClientConnectionManager` alle verbundenen Clients. Zusätzlich können von dieser Klasse neue `ClientConnection`-Instanzen angefragt werden.

7.3.2 Implementiererteil

7.3.2.1 Callbacks

Der Implementierer muss (oder kann, falls er die `PsychicServer`-Klasse benutzt) die vier Interfaces `ClientListener`, `ResetListener`, `ButtonListener` und `ExceptionListener` implementieren. Er wird dann vom `AbstractPsychicServer` bei relevanten Ereignissen benachrichtigt.

7.4 Daten-Pipeline

Die sogenannte Daten-Pipeline ist die Implementation der Sensordatenverarbeitung im Psychic-Framework. Alle Sensordaten, die den Server erreichen, werden durch eine vom Implementierer erstellbare Pipeline verarbeitet. Um eine Pipeline-Stück zu erstellen, muss lediglich `AbstractFilter` erweitert werden. Mehrere dieser Stücke können dann miteinander verbunden werden, so dass alle Daten die komplette Pipeline

durchlaufen. Damit können zum Beispiel Gyroskopdaten erst mit einem Filter integriert werden, um sie danach zu glätten. Das letzte Element der Pipeline vor der Verwendung könnte dann z.B. noch zu kleine Ausschläge ignorieren, bevor die Daten schlussendlich der Anwendung aufbereitet zugeführt werden.

Die Pipeline unterstützt auch das Weiterleiten von Sensordaten an mehrere seperate weiterführende Pipelines.

7.5 Server-Discovery

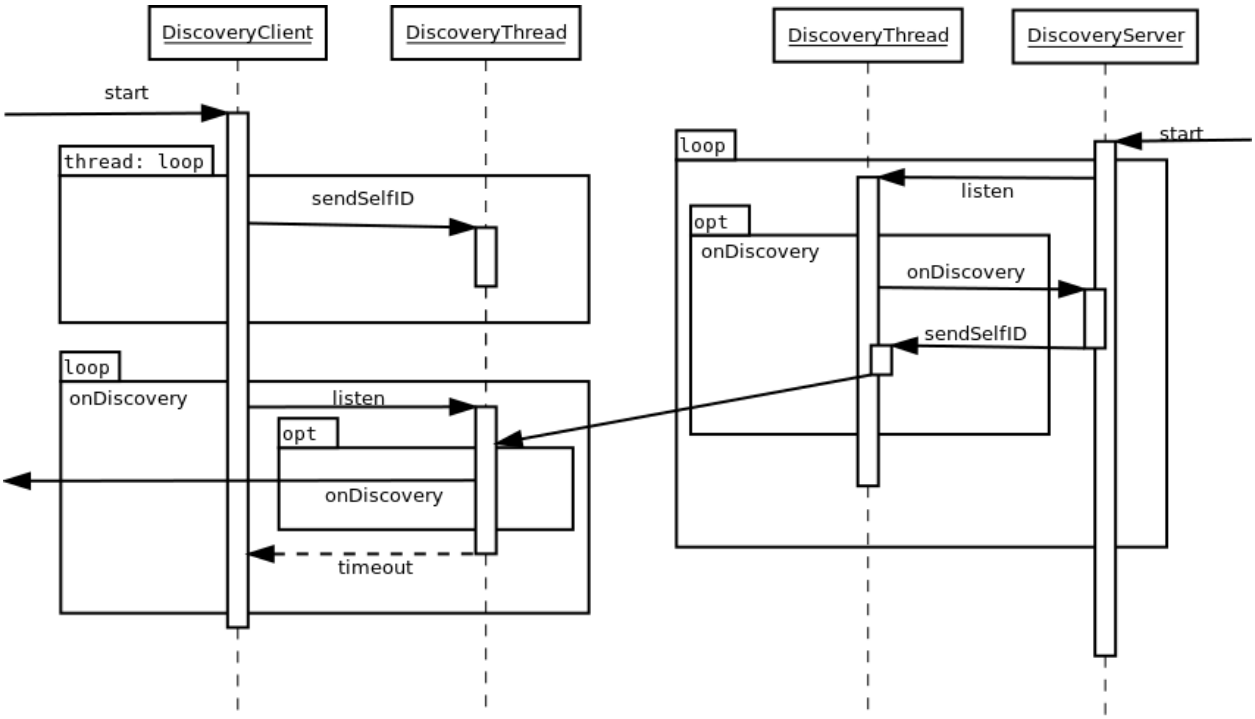

Abbildung 8: Sequenzdiagramm der Discovery-Phase

Grundlage der Serverfindung auf PC und Handy ist der sogenannte `DiscoveryThread`, in dem sich Funktionen befinden, die von Server und Client benötigt werden. Wichtig sind für die Funktionsweise insbesondere `sendSelfID`, `listen` und der `onDiscovery`-Callback.

`sendSelfID` sendet die für weitere Kommunikation notwendigen Informationen in Form eines sich selbst beschreibenden `NetworkDevice`. Der Client verwendet diese Funktion, um Server via Broadcast zu finden: Empfängt der Server ein solches Paket, wird `onDiscovery` aufgerufen, und der `DiscoveryServer` schickt an den Sender des Paketes seine eigene Identifikation via `sendSelfID`. Dann ist der Server dem `DiscoveryClient` bekannt, und der `onDiscovery`-Callback wird aufgerufen. Die App zeigt dem Nutzer den Server dann an, so dass dieser dem Client eine Verbindungsanfrage stellen kann. Erlaubt die Implementation dem Client die Verbindung, ist die Verbindung aufgebaut.

`listen` ist auf Client und Server dafür zuständig, per `sendSelfID` gesendete Pakete zu empfangen, und den `onDiscovery`-Callback auszulösen. Der Port, auf dem der Server die Broadcasts erwartet, muss in der App eingegeben werden, falls nicht der Standardport `8888` verwendet wird.

7.6 Kommunikation

Alle Daten und Commands werden als serialisierte Java-Objekte übertragen. Das Framework nutzt für alle Ports zufällige freie Ports, ausgenommen den für die Serverfindung genutzten, da dieser der App im Voraus

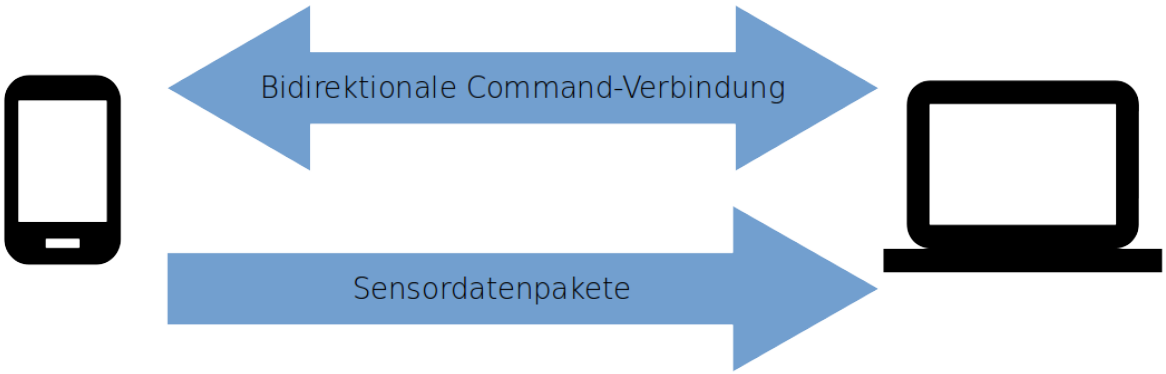

Abbildung 9: Architektur Verbindung

bekannt sein muss. Daten nutzen eine unidirektionale Verbindung vom Client zum Server, während die Kontrollverbindung bidirektional ist

7.6.1 Kontrollverbindung

Die Kontrollverbindung läuft über TCP; für jede neue Kontrollnachricht wird eine neue Verbindung aufgebaut. Jede der Kontrollnachrichten ist ein `AbstractCommand`. Diese Klasse enthält ein Enum, dass den Typ des Kontrollpakets beschreibt, so dass es einfach in die entsprechende Klasse gecastet werden kann. Diese Klassen können Instanzen aller serialisierbaren Klassen enthalten, so dass alle benötigten Daten einfach mitgeliefert werden können.

Da die Kontrollverbindung bidirektional ist, kann der Client auch `AbstractCommand`-Objekte an den Server schicken. Das wird unter anderem für die Knöpfe verwendet, aber auch um den Server über den Wertebereich von Sensoren zu informieren.

7.6.2 Datenverbindung

Nachdem der Nutzer sich mit einem Server verbunden hat, übermittelt dieser die aktuell benötigte Konfiguration. Nachdem die App diese umgesetzt hat, sind die benötigten Sensoren aktiviert. Alle generierten Daten werden ohne weitere Verarbeitung serialisiert und an den Datenport des Servers übermittelt, um den Rechenaufwand in der App möglichst gering zu halten. Die Übermittlung läuft über UDP, so dass kein expliziter Verbindungsaufbau für jedes Paket nötig ist.

8 Eigenständigkeitserklärung

Die Verfasser erklären, dass die vorliegende Arbeit von ihnen selbstständig, ohne fremde Hilfe und ohne die Benutzung anderer als der angegebenen Hilfsmittel angefertigt wurde.

9 Anhang: Klassendiagramme

Server-Klassendiagramm auch unter https://github.com/enra64/psychic-giggle/tree/master/Abschlussbericht/diagram.png

App-Klassendiagramm auch unter https://github.com/enra64/psychic-giggle/tree/master/Abschlussbericht/diagram_app.png

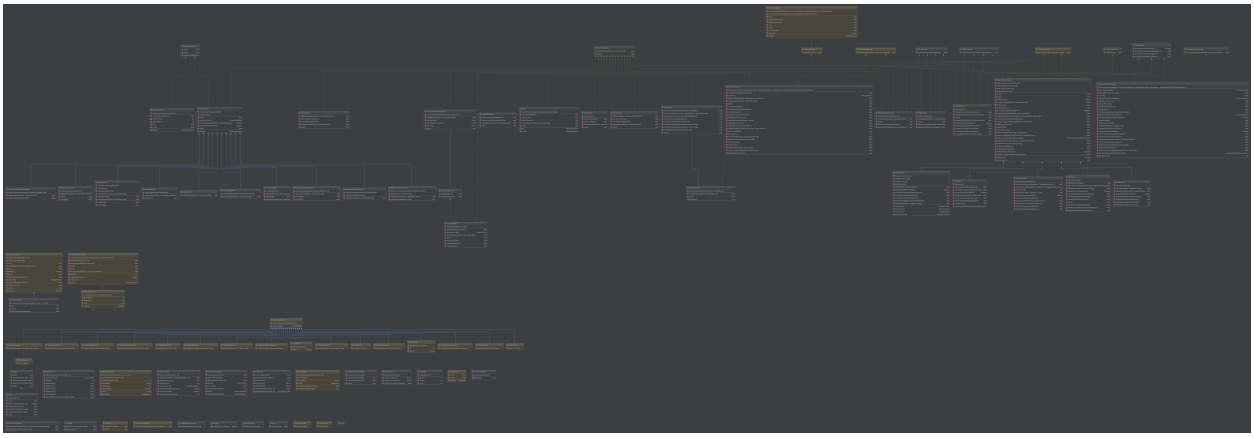

Abbildung 10: Klassendiagramm des Servers

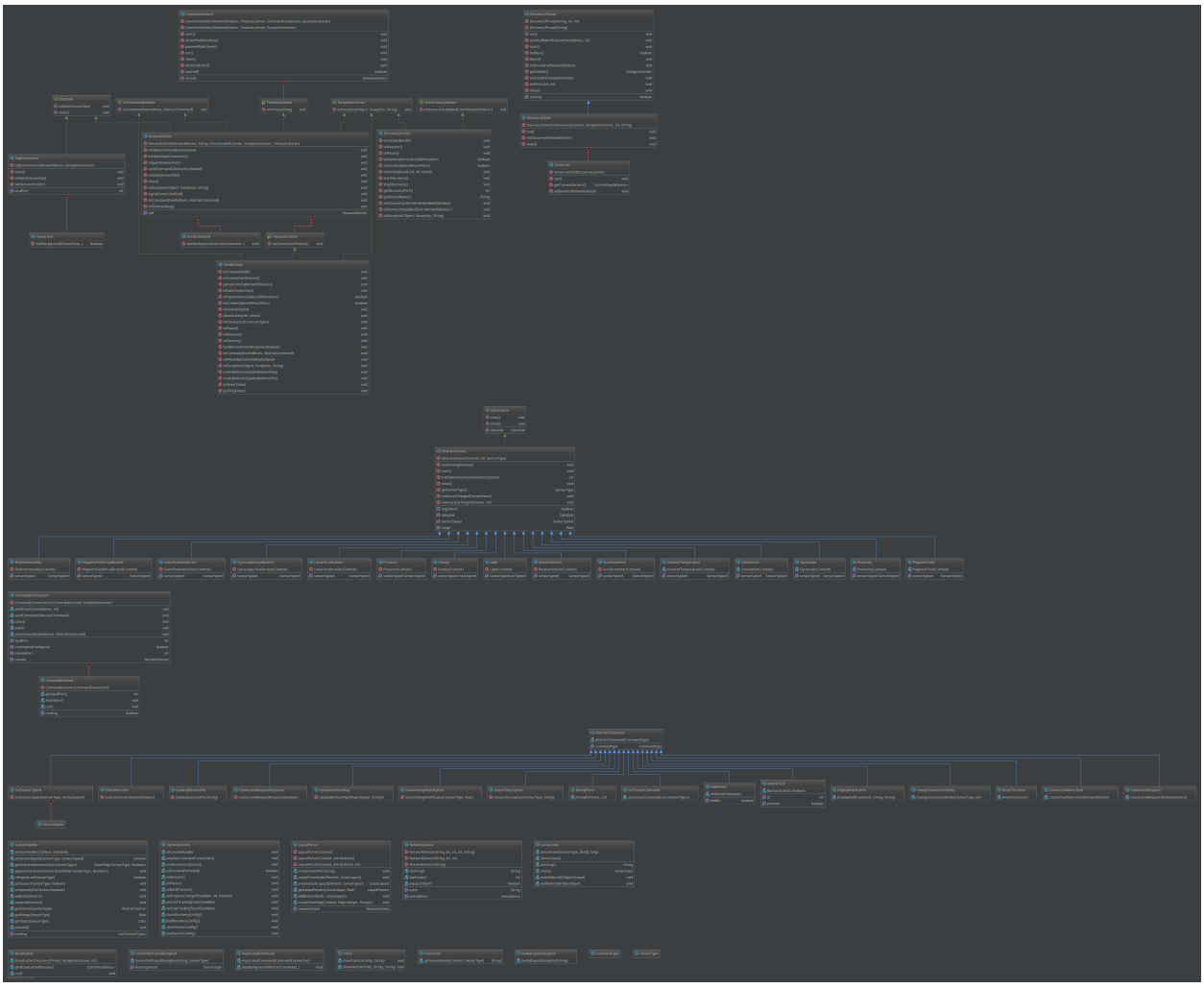

Abbildung 11: Klassendiagramm der App

Abbildungsverzeichnis