HSB
Hochschule Bremen
City University of Applied Sciences

Entwicklungsarbeit

Autor: Philipp Hennken

Matrikelnummer: 5264799

E-Mail: phennken@stud.hs-bremen.de

Hochschule Bremen

Fakultät 4 – Elektrotechnik und Informatik

Studiengang: Informatik Software und Systemtechnik dual 2. Semester

Modul 2.3 - Implementierung von netzwerkbasierten grafischen Oberflächen

Prüfer: Prof. Dr. Ing. Heiko Mosemann

Inhalt

1. Erklärung über das eigenständige Erstellen der Arbeit

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Stellen der Arbeit, die anderen Werken dem Wortlaut oder dem Sinn nach entnommen wurden, sind durch Angaben der Herkunft kenntlich gemacht.

Diese Erklärung erstreckt sich auch auf in der Arbeit enthaltene Grafiken, Skizzen, bildliche Darstellungen sowie auf Quellen aus dem Internet.

Die Arbeit habe ich in gleicher oder ähnlicher Form auch auszugsweise noch nicht als Bestandteil einer Prüfungs- oder Studienleistung vorgelegt.

Ich versichere, dass die eingereichte elektronische Version der Arbeit vollständig mit der Druckversion übereinstimmt.

Philipp Hennken

Matrikelnummer: 5264799

Wildeshausen, den 02.07.2023

2. Einleitung

Im Rahmen des Moduls „Modul 2.3 - Implementierung von netzwerkbasierten grafischen Oberflächen“ ist die Prüfungsform für das Wintersemester 2022/2023 die Entwicklungsarbeit. Nach Abschnitt II: Prüfungsform § 7 Arten der Prüfungsleistungen, Studienleistungen der Prüfungsordnung umfasst eine Entwicklungsarbeit die Aufgabenstellung, die Anforderungsdefinition, einen Entwurf, das Quellprogramm, die Testdokumentation, Benutzungshinweise sowie ein Anwendungsbeispiel.

3. Aufgabenstellung

Die Aufgabenstellung stammt aus dem Dokument „Aufgabenstellung Entwicklungsarbeit GUIPROG im SoSe 23“ (vgl. Link zur Aufgabenstellung für die Entwicklungsarbeit), welches am 17.05.2023 veröffentlicht und vorgestellt wurde.

Die gesamte Aufgabenstellung besteht aus drei Teilaufgaben, welche miteinander verknüpft sind, allerdings auch unabhängig voneinander bearbeitet werden können.

Die fertige Entwicklungsarbeit soll bis zum 7. Juli 2023 über den Dateiaustauschdienst „GigaMove“ (vgl. Link zum Austauschdienst GigaMove) abgegeben werden.

3.1 Teilaufgabe 1

In Teilaufgabe 1 der Entwicklungsarbeit soll eine Software implementiert werden, die die Hardware mit allen Komponenten einer zivilen Drohne simuliert. Welche Hardware eine Drohne haben soll, ist nicht spezifiziert und soll von der geprüften Person selbst entschieden werden. Zu implementierende Sensoren, Motoren, Aktoren, Akkus und anderer, selbst hinzugefügter, womöglich sogar futuristischer Hardware, sind alle von der geprüften Person nach dem Nutzen zu bewerten und dementsprechend zu integrieren.

Es ist explizit nicht gefordert, dass die Drohne realistisch zu implementieren ist, da dies nicht der Bestandteil der Vorlesungen war. Als Inspiration wurde als Quelle die Internetseite von „MatLab“ mitgegeben, auf der eine Drohne mit ihren realistischen Parametern dargestellt ist. (vgl. Link zur Modellierung einer professionellen Drohne). Die dort aufgeführten Parameter können als Inspiration für die von der geprüften Person implementierte Drohne genutzt werden.

Die Software ist ein Drohnen-Controller, der als Schnittstelle zwischen der Planungssoftware und der Terrain-Simulation eingesetzt werden soll. Die Planungssoftware wird in Teilaufgabe 2 implementiert, die Terrain-Umgebung wird in Teilaufgabe 3 erstellt und angepasst. Die Kommunikation zwischen den Bestandteilen soll über TCP/IP oder UDP umgesetzt werden. Welche Netzwerkverbindung genutzt wird, oder ob sogar beide genutzt werden, soll von der geprüften Person entschieden werden.

3.2 Teilaufgabe 2

Teilaufgabe 2 fordert von der geprüften Person, dass eine Software zur Planung, Durchführung und Verwaltung von Flügen der zivilen Drohne erstellt wird.

Als Inspiration kann sich für die Funktionalität der Software an der Software „eMotion“ der Firma „AgEagle Aerial Systems Inc.“ (vgl. Link zu der eMotion Software) orientiert werden.

Die Software soll eine Bedienoberfläche beinhalten, die durch JavaFX realisiert wird. Dabei wird explizit darauf hingewiesen, dass keine professionelle Bedienoberfläche erwartet wird. Erwartet wird allerdings, dass moderne GUI-Elemente eingesetzt werden. Gefordert ist darüber hinaus ein Menü, in dem man einzelne Funktionen auswählen oder Einstellungen vornehmen kann. Unter „Menü“ ist ein Menü zu verstehen, wie es im mitgegebenen Tutorial vorgestellt wird (vgl. Link zum Tutorial für ein Menü).

Die Mindestanforderungen sind zwei Dialoge und drei Steuerelemente von zwei verschiedenen Grundtypen wie Knöpfen oder Reglern, welche geeignet visualisiert werden sollen. Die zu prüfende Person soll darauf achten, keine urheberrechtlich geschützten Grafiken oder Sounds zu verwenden.

Optional kann sich an der IBM-Design-Language zur Gestaltung der GUI orientiert werden. (vgl. Link zu IBM-Language-Design).

Der Drohnen-Controller aus Aufgabe 1 soll die anzuzeigenden Daten liefern, die in dieser Planungssoftware dann angezeigt werden. Geplante Flugdaten wie Einstellungen der Drohnen und Routen soll die Planungssoftware an den Drohnen-Controller senden. Verbindungen sollen entweder über TCP/IP oder UDP hergestellt werden.

3.3 Teilaufgabe 3

Die dritte Teilaufgabe dient der Visualisierung und Testung der Drohnenplanungssoftware und der Drohne. Dafür soll eine Terrain-Simulation eingesetzt werden, in der eine Landschaft zur

Verfügung gestellt wird, durch die die Drohne mit Kommandos vom Drohnencontroller bewegt werden kann und Sensordaten der virtuellen zivilen Drohne an den Drohnencontroller zurücksenden kann. Von dem Drohnen-Controller sollen die Daten dann wieder an die Planungssoftware weitergeleitet werden, die die Daten der benutzenden Person in der GUI darstellt. Die Kommunikation soll in dieser Teilaufgabe über TCP/IP aufgebaut werden.

Zur Unterstützung wurde ein Terrain bereits implementiert das in der Software Unity eingesetzt werden kann. Dort kann bereits eine Drohne durch Tasteneingaben gesteuert werden.

4. Anforderungsdefinition

Es wird allgemein gefordert, dass objektorientiert programmiert wird. Redundanter Code soll vermieden und Methoden sollen modular und überschaubar gehalten werden. Das in dem Modul „Grundlagen der Informatik“ Gelernte soll ebenfalls berücksichtigt werden, dies beinhaltet, dass Methoden nach dem Design-by-Contract-Verfahren mit Vor- und Nachbedingungen kommentiert werden sollen. Kommentare und Dokumentationskommentare sollen sinnvoll sein, Bezeichner sollen sinnvolle und verständliche Namen haben. Literale sollen nur in Ausnahmefällen eingesetzt werden, sonst sollen Konstanten eingesetzt werden.

5. Entwurf

Zur Strukturierung des Programms verwende ich das model-view-control-Konzept. Nach diesem Konzept werden unterschiedliche Teile des Programms in disparate Pakete verteilt. Neben den Paketen „model“, „view“ und „control“ wird für dieses Programm das Pakete „res“ angelegt. Außerdem wird für die grafische Benutzungsoberfläche ein Ordner angelegt, in dem alle FXML-Dateien gespeichert sind, die das Design der Benutzungsoberfläche vorgeben.

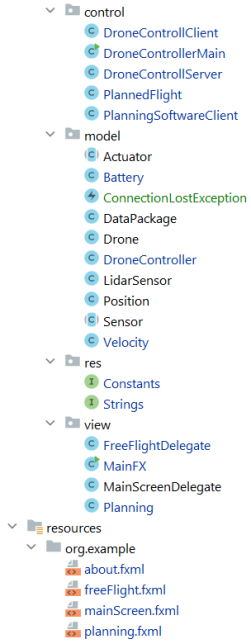

Abbildung 1: Aufteilung des Programms in Pakete

5.1 Pakete

Das „model“-Paket dient der Modellierung und der Festlegung von Eigenschaften des Programms. Es wird keine Logik im „model“-Paket implementiert, sondern nur alle benötigten Komponenten definiert. Daher befinden sich in diesem Paket die Klassen „Actuator“, „Battery“, „ConnectionLostException“, „DataPackage“, „Drone“, „LidarSensor“, „Position“, „Sensor“ und „Velocity“.

Das „view“ Paket beinhaltet alles für die Ergebnispräsentation und dient als Schnittstelle zu der benutzenden Person. In diesem Programm befindet sich daher alles, was mit der grafischen Benutzungsoberfläche zu tun hat, in diesem Paket.

Das „res“ Paket wird für alle Speicherungen von programmunabhängigen Daten genutzt. Dort liegen nur die Interfaces mit Konstanten und Literalen. In diesem Programm heißen die Klassen dafür „Constants“ und „Strings.

Das „control“ Paket beinhaltet die ganze Logik des Programms. Es steuert die Verbindungen zwsichen der Planungssoftware, also der Benutzungsoberfläche, dem Controller selbst und der Drohnensimulation Unity. In diesem Paket befinden sich also die Klassen „DroneControllerClient“, „DroneController“, „DroneControllerServer“, „DroneControllerMain“, „PlannedFlight“ und „PlanungsSoftwareClient“.

Im Unterordner „org.example“ des „resources“ Ordner liegen die vier FXML-Dateien für die grafische Benutzungsoberfläche. Für das Programm werden die „about.fxml“, die „freeFlight.fxml“, die „mainScreen.fxml“ und die planning.fxml“ Dateien benötigt. Diese beinhalten alle Bausteine der Anzeigen.

5.2 Aufbau des Programms - Softwareplanung

Entworfen werden soll eine zivile Drohne. Diese zu modellieren ist der Kernbestandteil dieser Entwicklungsarbeit. Dazu müssen alle relevanten Komponenten einer Drohne aufgelistet und anschließend implementiert werden. Der Entwurf beginnt somit mit dem Überblick über die Teile der Drohne, die später durch einen Drohnencontroller gesteuert werden sollen.

Außerdem ist eine grafische Benutzungsoberfläche gefordert. Diese soll der benutzenden Person ermöglichen, eine Route zu planen, die die Drohne dann anschließend abfliegt. Darüber hinaus wird ein freier Flug implementiert, sodass die benutzende Person die Drohne durch Pfeiltasten zu beliebigen Punkten steuern kann.

Angezeigt wird alles in der Terrain-Simulationssoftware Unity. Dort hinterlegt ist ein C#-Skript, das Kommandos des Drohnencontrollers in die Bewegung der Drohne übersetzt. Dieses Skript wird nicht explizit in diesem Dokument erläutert, da sich die Aufgabenstellung lediglich auf den Drohnencontroller, die Planungssoftware und deren Verbindungen beschränkt.

5.3 Komponenten einer Drohne

Alle Komponenten der Drohne können in drei Kategorien eingeteilt werden. Eine Drohne besitzt Sensoren, die Daten an den Controller liefern werden, Aktoren, die Daten vom Controller bekommen und dementsprechend agieren und Bauteile, die unabhängig von dem Drohnencontroller sind. Dazu zählen Teile wie die Rotorblätter, das Gehäuse und ähnliches. Diese Teile müssen in der Realität beachtet werden, da die Aktoren dementsprechend agieren müssen. In diesem Drohnencontroller wird laut Aufgabenstellung zur Vereinfachung des Programms der Realität nicht allzu viel Beachtung geschenkt, weswegen diese Komponenten ignoriert werden.

Für den Controller sind somit nur die Aktoren und die Sensoren relevant. Um den Code modular zu halten, werden die Aktoren und Sensoren von den Klassen „Aktoren“ und „Sensoren“ erben. Dadurch wird eine direkte Verbindung klar.

5.3.1 Aktoren

Aktor der Drohne ist die Batterie, die die benötigte Leistung zur Verfügung stellt.

Die Batterie wird eine gewisse Kapazität haben. Diese Kapazität wird sich nach einer gewissen Flugdauer minimieren. Sollte die Batterie leer sein, kann die Drohne nicht weiter gesteuert werden und das Programm wird beendet.

5.3.2 Sensoren

Sensoren geben dem Drohnencontroller alle Informationen, die für die benutzende Person interessant sind.

Daher wird ein Abstandsmesssensor integriert, der den Abstand von der Drohne zum Boden misst. Diese Funktion kann gut mit dem automatischen Flug verbunden werden. Das Terrain kann somit durch die simulierte Drohne ausgemessen werden.

Ebenfalls ist ein Positionssensor eingebaut, sodass die benutzende Person immer eine Position der Drohne angezeigt bekommt.

Auch die Geschwindigkeit wird durch einen Sensor der Drohne an die benutzende Person übermittelt.

5.3.4 Übersicht der Drohne

Die Drohne besteht somit aus den Attributen „lidarSensor“, die den Abstandsmesser zum Boden simuliert, einer Batterie „battery“, einer Geschwindigkeit „velocity“ und einer Position „position“. Außerdem hat die Drohne einen String „movement“, der an die Drohnensimulation übermittelt wird, sodass die diesen String in ein Bewegungskommando übersetzen kann, das dann ausgeführt wird. Übersichtlich dargestellt wird dies in einem UML-Diagramm.

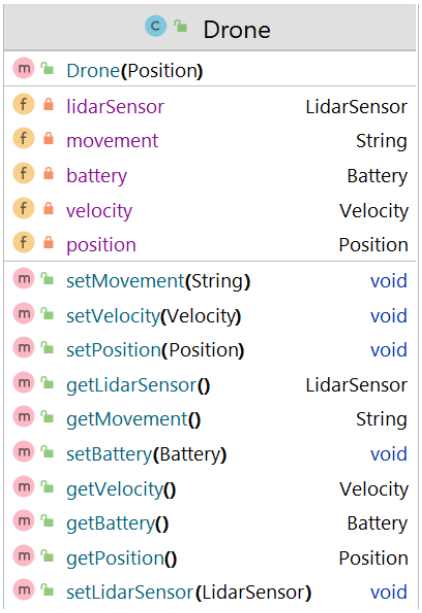

Abbildung 2: UML-Diagramm einer Drohne

5.4 Bewegung der Drohne

Die Bewegung der Drohne findet primär in der Terrain-Simulation statt. Aus dem Java-Programm werden die Kommandos über die TCP/IP-Verbindung an Unity geschickt, diese wandelt das Kommando in die Bewegung der Drohne um. Mögliche Bewegungen dabei sind die Bewegung nach vorne, nach hinten, nach links und nach rechts. Eine klassische Multicopter-Drohne kann ebenfalls nach oben und unten fliegen und sich auch auf der Stelle in beide Richtungen drehen.

Um die Drohne zu steuern, wird der benutzenden Person eine grafische Benutzungsoberfläche angezeigt. Diese grafische Benutzungsoberfläche sendet Kommandos an den Drohencontroller, der diese dann an Unity weiterleitet. Daher besitzt der Drohnencontroller sowohl eine Server- als auch eine Client-Klasse.

5.5 Verbindungen der Programme

Der Client sendet die Kommandos an Unity und wartet auf die Antwort, die alles aus einem „DataPackage“ beinhaltet. (Auszug aus dem Quellcode 1).

Das DataPackage besteht aus den Geokoordinaten, dem Abstand zum Boden sowie der Geschwindigkeiten in alle Richtungen der Drohne. Es wird immer als JSON-Objekt versendet. Dafür wird die GSON-Bibliothek genutzt.

Die Daten werden der Drohne im Drohnenconroller zugewiesen, sodass dort die gesamte Logik des Programms steckt. Dafür wird der von Unity erhaltene String mit Hilfe eines „BufferedReaders“ zeilenweise eingelesen und durch die Methode zurückgegeben. (Auszug aus dem Quellcode 1 – Zeilen 9 und 13).

Dabei kann es passieren, dass die Verbindung zu Unity abbricht. In dem Fall soll die „ConnectionToUnityLostException“ geworfen werden. (Auszug aus dem Quellcode 1 – Zeile 17). Diese gibt den Fehler an die benutzende Person aus und beendet das Programm, sodass es, nachdem Unity gestartet wurde, neu ausgeführt werden kann.

```
01    private String sendCommandAndWaitForAnswer(Socket socket, String
command) throws ConnectionLostException
02    {
03        PrintWriter outToUnity = null;
04        BufferedReader inFromUnity = null;
05        String inFromUnityString;
06        try
07        {
08            outToUnity = new PrintWriter(socket.getOutputStream());
09            inFromUnity = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
10            outToUnity.println(command);
11            outToUnity.flush();
12            inFromUnityString = inFromUnity.readLine();
13            return inFromUnityString;
14        }
15        catch (IOException | NullPointerException exception)
16        {
17            throw new
ConnectionLostException(Strings.CONNECTION_LOST_TO_UNITY);
18        }
19    }
```

Auszug aus dem Quellcode 1

Der Server greift sich die Daten von der Drohne und sendet sie dann an die Planungssoftware, wo sie in der grafischen Benutzungsoberfläche dargestellt werden. Der Server liest den String aus, den die grafische Benutzungsoberfläche zurückgibt und setzt die Bewegung der Drohne dementsprechend. Diese Bewegung wird dann im Client zurück an Unity geschickt. (Auszug aus dem Quellcode 1 – Zeile 10).

An die grafische Benutzungsoberfläche sendet der Server wiederrum ein Datenpaket, das die Attribute der Drohne beinhaltet.

5.6 Grafische Benutzungsoberfläche

Die grafische Benutzungsoberfläche dient der benutzenden Person dazu, das Programm innovativ zu verstehen und ermöglicht somit auch Laien die Benutzung des Programms. Der benutzenden Person wird ein Fenster angezeigt, in dem sie zwischen den Tabs „Flug“ und „Planung“ wechseln kann. Ebenfalls erscheint eine Menüleiste mit den Reitern „About“ und „Programm“. Unter dem Reiter „About“ befindet sich ein Fenster, das generelle Daten zum Programm angibt. Unter „Programm“ lässt sich das Programm beenden.

Im Tab „Flug“ lässt sich die Drohne mit den Kommandos „W“, „A“, „S“, „D“, „Up“ und „Down“ in jede beliebige Richtung fliegen. Die Tasten „TL“ und „TR“ lässt die Drohne auf der Stelle drehen. Jedes Kommando wird so lange ausgeführt, bis ein anderes Kommando gedrückt wird. Das Kommando „STOP“ lässt die Drohne stoppen und auf der Stelle schweben.

Im oberen Bereich werden Daten der Drohne angezeigt, dazu gehören die Geschwindigkeit der Drohne in jede Richtung, sowie die Koordinaten der Drohne in der Terrain-Simulation und der Abstand zum Boden. In der oberen rechten Ecke ist die Batterie zu sehen, die mit einer „ProgressBar“ realisiert ist. Darunter ist ein Slider und ein Knopf „Charge“, der die Drohne auf das Level des Sliders auflädt.

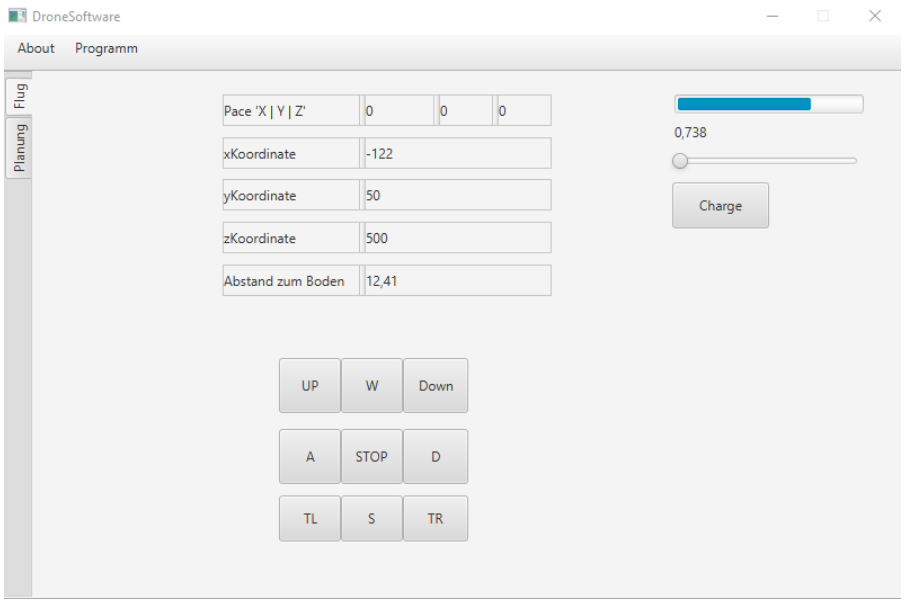

Abbildung 3: Grafische Benutzungsoberfläche im Tab "Flug"

Im Tab „Planung“ kann eine Route geplant und abgeflogen werden. Dafür stehen drei Textfelder zur Verfügung. Dort können die x-, die y- und die z-Koordinate des Ziels eingegeben

werden. Auf Knopfdruck des „Füge Koordinate hinzu“-Knopfes werden diese dann in der Liste angezeigt. Mit dem Knopf „Lösche letzte Koordinate“ wird die letzte Koordinate der Liste entfernt.

Sollte der Knopf „Fliege Weg ab“ gedrückt werden, fliegt die Drohne vom ersten bis zum letzten alle Punkte der Liste ab und hält bei jedem Punkt fünf Sekunden lang an.

In der rechten unteren Ecke befinden sich wieder alle wichtigen Informationen über die Drohne, die durch die benutzende Person ausgelesen werden können. Die Batterie ist an derselben Stelle wie im Tab „Flug“ und beinhaltet die gleichen Funktionen.

Alle Statusmeldungen werden über der Liste an die benutzende Person ausgegeben, dazu gehören Nachrichten über neu eingetragene Koordinaten, gelöschte Koordinaten oder Fehlermeldungen.

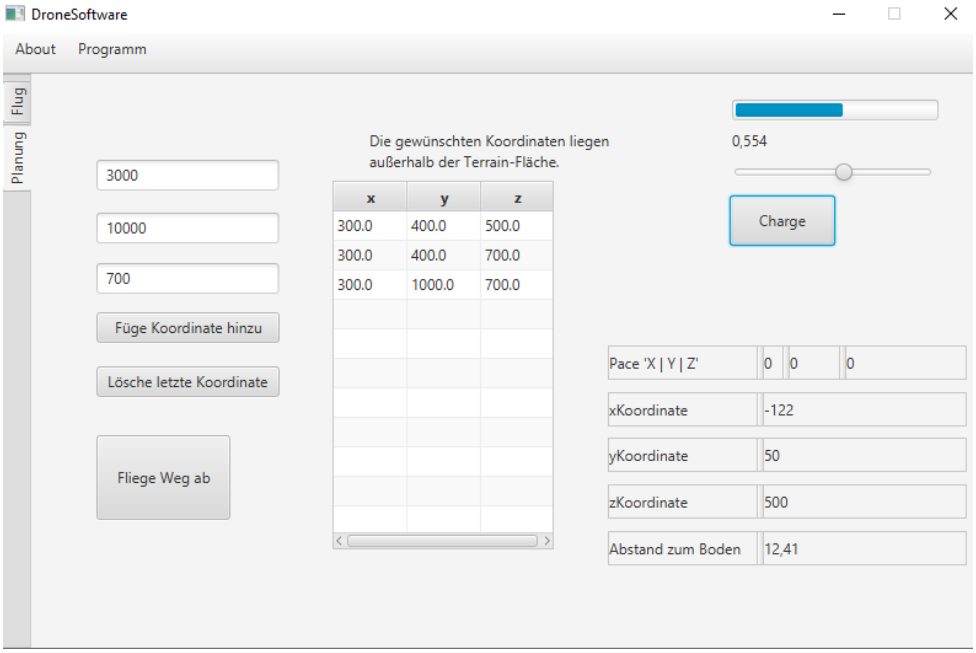

Abbildung 4: Grafische Benutzungsoberfläche im Tab "Planung"

Zum Anzeigen der Daten werden Properties und Bindings verwendet. Jede Anzeige eines Wertes ist ein Label. Dieses Label beinhaltet ein „TextProperty“ und wird zu der entsprechenden Property der Drohne „gebindet“. Das bedeutet, dass jede Änderung des Properties direkt in der grafischen Benutzungsoberfläche angezeigt wird. (Auszug aus dem Quellcode 2 – Zeilen 3 bis 11).

```
01    public static void setBindings(Label anzeigeX, Label anzeigeY,
Label anzeigeZ, Label anzeigeXVelocity, Label abstandZumBoden,
ProgressBar chargeLevelProgressBar, Label chargeLevelLabel, Label
anzeigeYVelocity, Label anzeigeZVelocity)
02    {
03        Bindings.bindBidirectional(anzeigeX.textProperty(),
DroneController.getClassInstance().getXKoordinateProperty(), new
NumberStringConverter());
04        Bindings.bindBidirectional(anzeigeY.textProperty(),
DroneController.getClassInstance().getYKoordinateProperty(), new
NumberStringConverter());
05        Bindings.bindBidirectional(anzeigeZ.textProperty(),
DroneController.getClassInstance().getZKoordinateProperty(), new
NumberStringConverter());
06        Bindings.bindBidirectional(anzeigeXVelocity.textProperty(),
DroneController.getClassInstance().getxVelocityProperty(), new
NumberStringConverter());
07        Bindings.bindBidirectional(abstandZumBoden.textProperty(),
DroneController.getClassInstance().getDistanceToGroundProperty(), new
NumberStringConverter());
08
Bindings.bindBidirectional(chargeLevelProgressBar.progressProperty(),
DroneController.getClassInstance().getChargeLevelProperty());
09        Bindings.bindBidirectional(chargeLevelLabel.textProperty(),
DroneController.getClassInstance().getChargeLevelProperty(), new
NumberStringConverter());
10        Bindings.bindBidirectional(anzeigeYVelocity.textProperty(),
DroneController.getClassInstance().getyVelocityProperty(), new
NumberStringConverter());
11        Bindings.bindBidirectional(anzeigeZVelocity.textProperty(),
DroneController.getClassInstance().getzVelocityProperty(), new
NumberStringConverter());
12    }
```

Auszug aus dem Quellcode 2

5.7 Planung eines Fluges

Bei der Planung des Fluges kann die benutzende Person Werte in Textfeldern eingeben. Eingabe von nutzenden Personen sind immer Gefahrenstellen für die Funktionalität des Programmes.

Fehlerhafte Eingaben müssen daher so weit behandelt werden, dass die benutzende Person die Fehler korrigieren kann und das Programm nicht abstürzt. Ein möglicher Fehler ist, dass ungültige Charaktere in das Textfeld eingegeben werden. Ungültige Charaktere wären beispielsweise Buchstaben oder Sonderzeichen. Ebenfalls müssen Kommazahlen durch den Dezimalpunkt und nicht das Dezimalkomma getrennt sein. Der Fehler, der dabei auftreten kann,

ist die „NumberFormatException“, da die Eingaben der benutzenden Person vom String zu einem Double Wert gecasted werden.

Tritt dabei ein Fehler auf, wird dieser gefangen und die nutzende Person wird über das Info-Textfeld darauf hingewiesen, dass die Eingabe ungültig war. Ebenfalls wird nochmal auf den Dezimalpunkt hingewiesen, da das eine der häufigsten Fehlerquellen sein wird, die nicht direkt ersichtlich ist. (Auszug aus dem Quellcode 3 – Zeilen 19 bis 21).

Eine andere Fehlerquelle ist, dass eine Position angeflogen werden soll, die außerhalb der Grenzen der Simulationssoftware liegt.

Dafür wurden die Grenzen der Simulationssoftware ausgelesen und in das Programm integriert, sodass die eingegebene Werte nicht außerhalb dieser Grenzen liegen. Sollte dies der Fall sein, wird der benutzenden Person das Feedback gegeben, dass diese Position nicht angeflogen werden kann. (Auszug aus dem Quellcode 3 – Zeilen 5 bis 18).

```
01    public void onAddCoordinatesClick()
02    {
03        try
04        {
05            if (Double.parseDouble(zCoordinateEntry.getText()) >=
Constants.MIN_Z_COODINATE &&
06                    Double.parseDouble(zCoordinateEntry.getText()) <=
Constants.MAX_Z_COODINATE &&
07                    Double.parseDouble(xCoordinateEntry.getText()) >=
Constants.MIN_X_COODINATE &&
08                    Double.parseDouble(xCoordinateEntry.getText()) <=
Constants.MAX_X_COODINATE)
09            {
10                positions.add(new Position(
11
Double.parseDouble(xCoordinateEntry.getText()),Double.parseDouble(yCoord
inateEntry.getText()),Double.parseDouble(zCoordinateEntry.getText())));
12                tableWithCoordinates.setItems(positions);
13                infoLabel.setText(Strings.POSITION_ADDED +
coordinateToString(positions.get(positions.size()-1)));;
14            }
15            else
16            {
17                infoLabel.setText(Strings.OUT_OF_BORDER);
18            }
19        }catch (NumberFormatException numberFormatException)
20        {
21            infoLabel.setText(Strings.WRONG_INPUT_COORDINATES);
22        }
23    }
```

Auszug aus dem Quellcode 3

Sollte kein Fehler bei der Eintragung gemacht werden, fliegt die Drohne den Weg ab, nachdem der entsprechende Knopf gedrückt wurde. Dafür werden die Kommandos immer so lange

```
1    private void flyToDestination(double destinationX, double
destinationY, double destinationZ)
2    {
3flyToYDestinationAndStop(Constants.MAX_HIGHT_OF_MOUNTAINS_PLUS_TWENTY);
4        flyToXDestinationAndStop(destinationX);
5        flyToZDestinationAndStop(destinationZ);
6        flyToYDestinationAndStop(destinationY);
7    }
```

Auszug aus dem Quellcode 4

gesendet, bis die Drohne an der richtigen Stelle für jede Koordinate ankommt. Zuerst fliegt die Drohne gerade, bis zur Höhe 150 Einheiten, nach oben. Das dient dazu, die Kollisionen auch mit den höchsten Bergen des Terrains zu vermeiden. Anschließend fliegt die Drohne zur richtigen x-Koordinate, danach zur richtigen Z-Koordinate und am Ende passt sich die y-Koordinate an, bis sich die Drohne an der richtigen Stelle befindet. (Auszug aus dem Quellcode 4 – Zeilen 4 bis 6).

Die Methoden „flyToXDestinationAndStop“, „flyToYDestinationAndStop“ und „flyToZDestinationAndStop“ sind sehr identisch. Das entsprechende Kommando zur Bewegung in die richtige Richtung wird so lange gesendet, bis es sich dem übergebenen Zielwert so weit annähert, dass die Drohne nur noch die Bremsdistanz davon entfernt ist. (Auszug aus dem Quellcode 5 – Zeile 5).

Vorher wird überprüft, ob die aktuelle Position größer ist als die Zielkoordinate. Davon abhängig ist dann, welches Kommando gesendet wird. (Auszug aus dem Quellcode 5 – Zeile 3).

Dann wird das Kommando „notMoving“ gesendet, was die nächste Bewegung in eine andere Richtung ermöglicht. (Auszug aus dem Quellcode 5 – Zeile 17).

```
01          private void flyToXDestinationAndStop(double destinationX)
02          {
03              if (droneController.getXKoordinateProperty().getValue() <=
destinationX)
04              {
05                  while
(droneController.getXKoordinateProperty().getValue() <= destinationX -
Constants.BRAKING_DISTANCE_X)
06                  {
07
FreeFlightDelegate.setCommandToController(Strings.D);
08                  }
09              }
10              else
11              {
12                  while
(droneController.getXKoordinateProperty().getValue() >= destinationX +
Constants.BRAKING_DISTANCE_X)
13                  {
14
FreeFlightDelegate.setCommandToController(Strings.A);
15                  }
16              }
17
FreeFlightDelegate.setCommandToController(Strings.NOT_MOVING);
18          }
19      }
```

Auszug aus dem Quellcode 5

5.8 Übersicht der gesamten Software.

Das gesamte Programm lässt sich übersichtlich als UML-Diagramm darstellen.

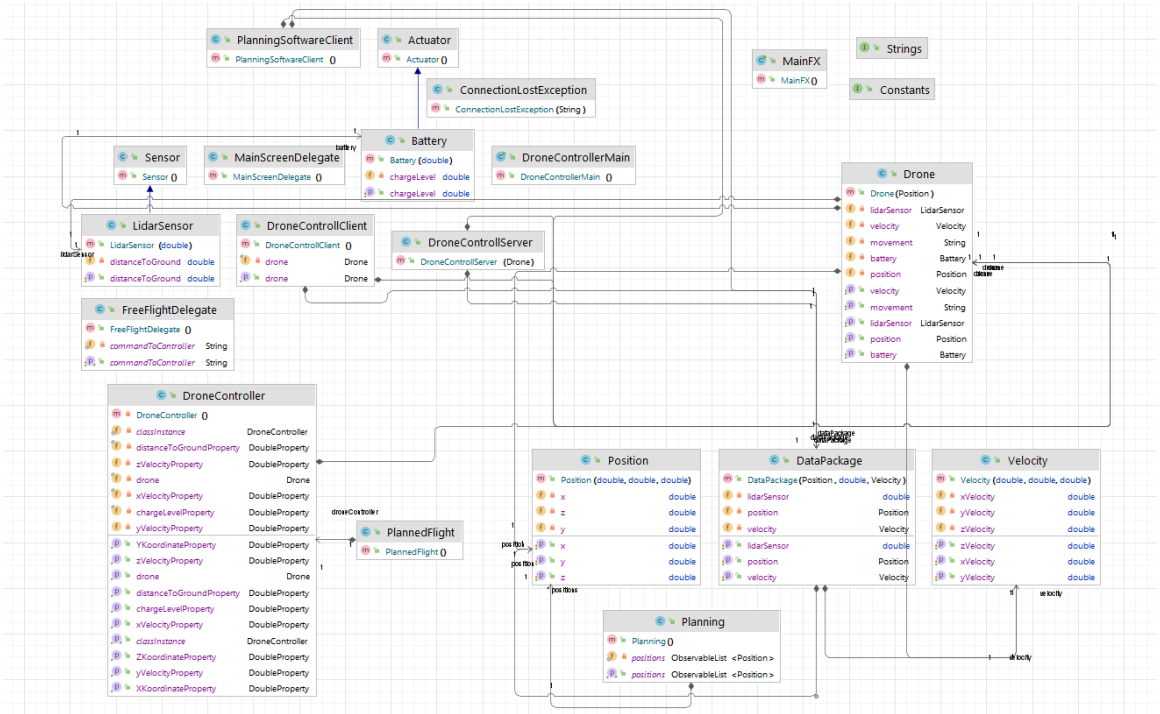

Abbildung 5: Übersicht der Software als UML-Diagramm

6. Benutzungshinweise

Zur Bedienung ist es erforderlich, das Unity-Projekt mit dem korrekten Skript vorm Start der Java-Anwendung zu starten. Erst dann hat das Programm die Möglichkeit, Werte zu empfangen und zurückzusenden.

Darüber hinaus sollte das Fenster mit der Drohnensimulation dauerhaft geöffnet bleiben, da man sonst die Auswirkungen der Eingaben nicht nachvollziehen kann.

Über die Tabs sind die beiden Steuermenus aufrufbar. Unter dem Tab „Flug“ befindet sich die Steuerung als freier Flug, unter dem Tab „Planung“ findet sich die Planung eines Fluges nach Koordinaten.

Im der Menu-Leiste am oberen Bildrand kann der benutzenden Person ein Text zur Version des Programms angezeigt werden. Außerdem ist es möglich unter dem Reiter Programm das Programm zu beenden.

Das Programm ist sehr benutzerfreundlich geschrieben, sodass falsche Eingaben einfach korrigiert werden können. Das Meiste des Programms ist mit Knöpfen zu bedienen, sodass dort keine Fehler passieren können und die benutzende Person viel herumprobieren kann, ohne Fehler anzurichten.

Die einzige Fehlerquelle ist die Eingabe von Zielkoordinaten. Dort müssen in die Felder durch Dezimalpunkt getrennte, rationale Zahlenwerte eingetragen werden. Die Werte für die x-Koordinate müssen kleiner als 800 Einheiten und größer als -800 Einheiten sein. Die Werte für die y-Koordinate müssen kleiner als 1800 Einheiten und größer als 180 Einheiten sein. Das liegt daran, dass die Welt in Unity außerhalb dieser Grenzen nicht definiert ist. Für die y-Koordinaten sind nur Werte erlaubt, die größer als Null sind.

Sollten dort Fehler gemacht werden, ist ein Warnhinweis hinterlegt, der eine Hilfestellung zur Fehlerbehebung vorschlägt. Auch diese Eingabe kann korrigiert werden und wird niemals falsch eingetragen werden, sodass mit den Eingaben ebenfalls experimentiert werden kann.

7. Testdokumentation

Tests dienen zur Überprüfung des Programms. Fehler im Programm sollen durch die Testung aller Komponenten des Programms ausgeschlossen und behoben werden, sodass benutzende Personen ungestört das vorliegende Programm verwenden können.

7.1 Durchführung der Tests

Zuerst werden die Verbindungen zwischen der Planungssoftware und der Simulationssoftware getestet. Sollte die Verbindung zwischen den beiden Teilen der Aufgabe stehen, so ist gleichzeitig sichergestellt, dass die Verbindungen zwischen dem DrohnenController und der Planungssoftware, sowie der Terrainsimulation und dem Drohnencontroller einwandfrei funktionieren.

Der erste Schritt ist, dass die Daten aus Unity in der grafischen Benutzungsoberfläche angezeigt werden sollen. Es ist zu sehen, dass die in Unity angezeigten Koordinaten der Drohne identisch zu den Koordinaten in der Planungssoftware sind. (vgl. Abbildung 5).

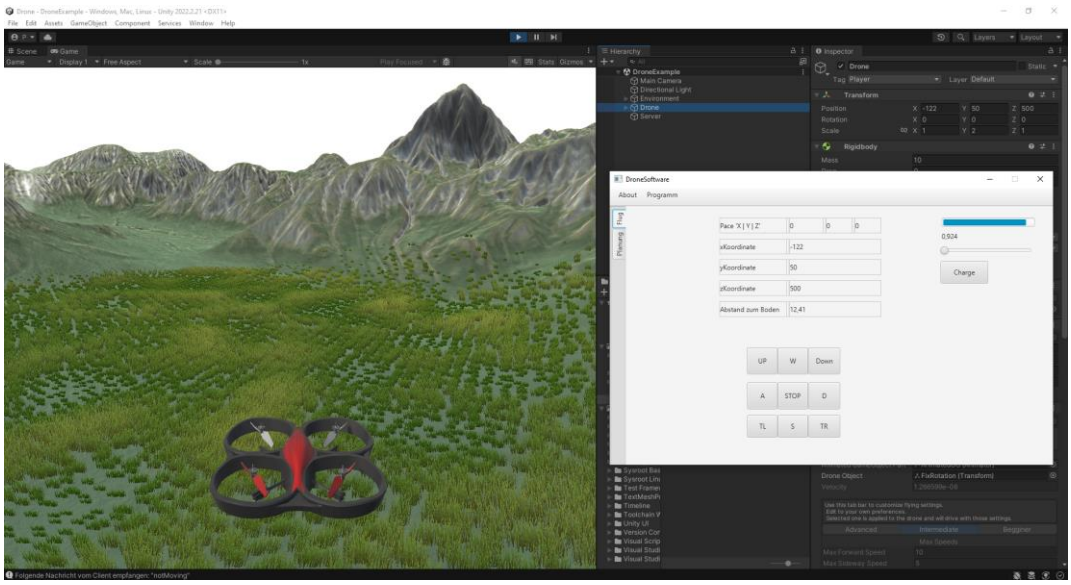

Abbildung 6: Daten aus Unity werden in der grafischen Benutzungsoberfläche angezeigt

Danach wird die Funktionalität jedes Knopfes des freien Fluges getestet. An der Verhaltensweise der Drohne wird die korrekte Ausführung überprüft. Gleichzeitig kann beobachtet werden, ob sich die Parameter der Drohne logisch korrekt verhalten. Sollte das Kommando „W“ durch die Betätigung des „W“-Knopfes geschickt werden, so darf sich nur der Wert der x-Koordinate ändern. Dazu muss auch eine Geschwindigkeit in z-Richtung größer als Null angezeigt werden. Ebenfalls sollte ein anderer Wert für den Abstand zum Boden entstehen, dies muss aber nicht zwingend der Fall sein.

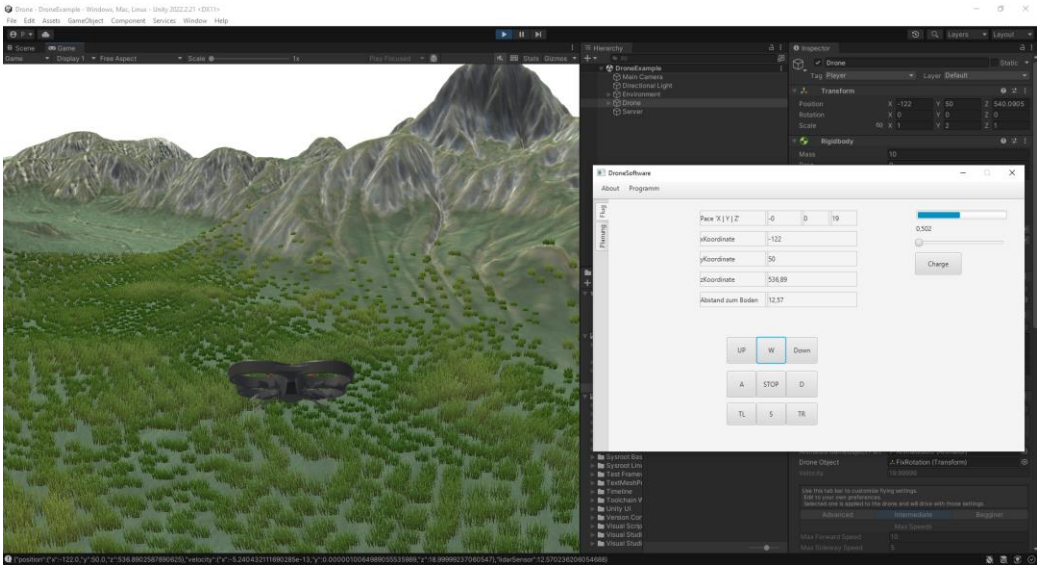

Abbildung 7: Test der Funktionalität für den "W"-Knopf

Es ist zu erkennen, dass sich die Drohne nach vorne bewegt. Lediglich die x-Koordinate ändert sich, die anderen Koordinaten sind bei gleichem Wert geblieben. Die Geschwindigkeit ist ebenfalls nur in z-Richtung auf 19 Einheiten pro Sekunde gestiegen. Wie vermutet, änderte sich auch der Abstand der Drohne zum Boden.

Das gleiche Testverfahren wird auch für die anderen Knöpfe durchgeführt.

Für den „S“-Knopf entstand folgendes Ergebnis:

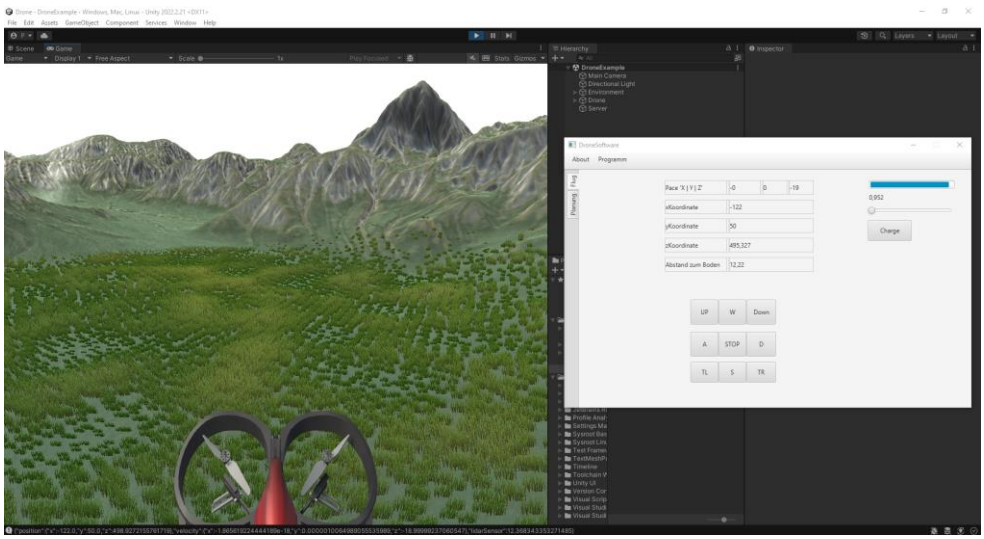

Abbildung 8: Test der Funktionalität für den "W"-Knopf

Hier ändert sich ebenfalls wie vermutet nur die z-Koordinate, nur wurde sie in diesem Testschritt kleiner. Die Geschwindigkeit in z-Richtung beträgt negative 19 Einheiten pro Sekunde, was den Erwartungen entspricht. Auch der Abstand zum Boden ist ungleich den Abständen zum Boden an den anderen Stellen.

Für die Knöpfe „A“ und „D“ werden die gleichen Ergebnisse vermutet, wie für die Knöpfe „S“ und „W“, nur das sich anstatt der z-Koordinate nun die x-Koordinate ändern soll. Außerdem muss die Geschwindigkeit nun in x- und nicht in z-Richtung passieren.

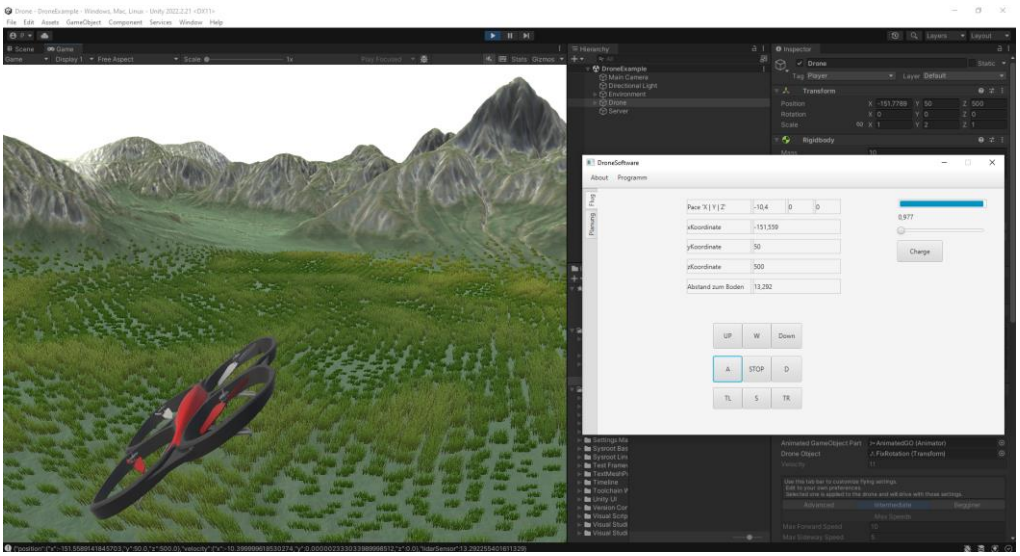

Abbildung 9: Test der Funktionalität für den "A"-Knopf

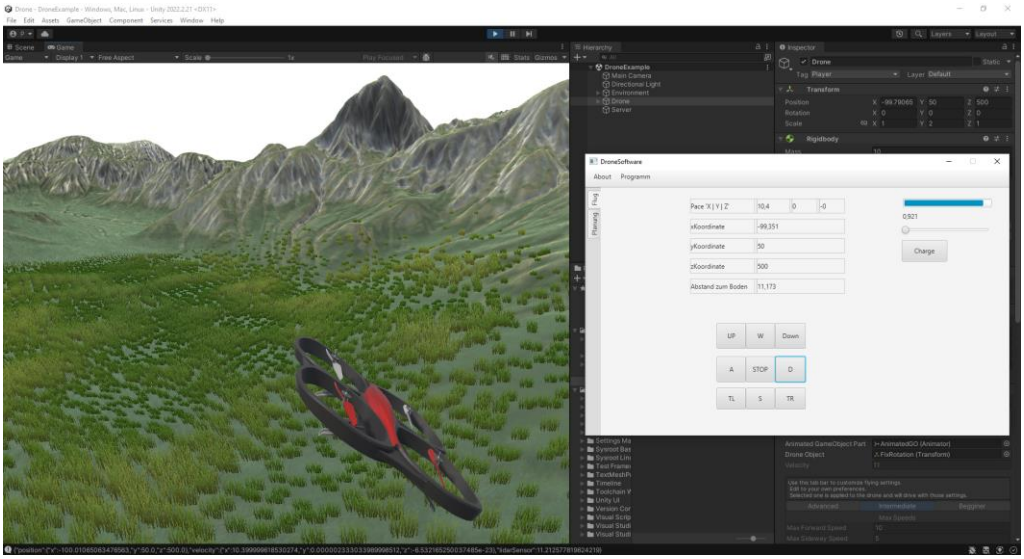

Abbildung 10: Test der Funktionalität für den "D"-Knopf

In beiden Beispielen ist die korrekte Funktionalität des Programms festzustellen. Die Koordinaten ändern ihre Werte wie gewünscht und auch die Anzeige der Geschwindigkeit in x-Richtung entspricht den Erwartungen.

Nun ist die Funktionalität der Knöpfe „Up“ und „Down“ zu überprüfen. Es gilt das Gleiche wie für die anderen Testdurchläufe, nur muss jetzt die Koordinaten- und Geschwindigkeitsänderung in y-Richtung passieren. Der Abstand zum Boden sollte proportional zur Änderung der y-Koordinate geschehen.

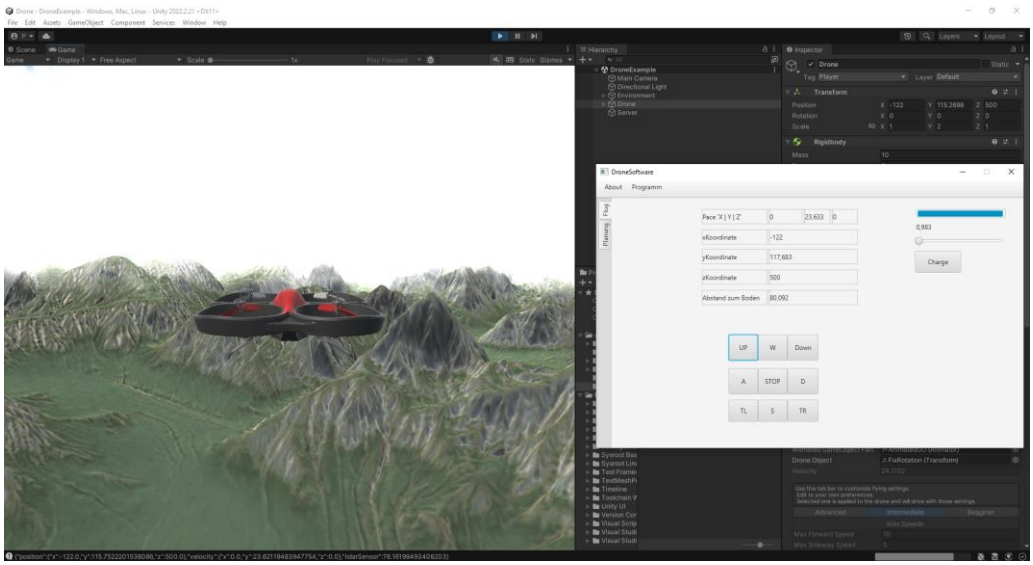

Abbildung 11: Test der Funktionalität für den "Up"-Knopf

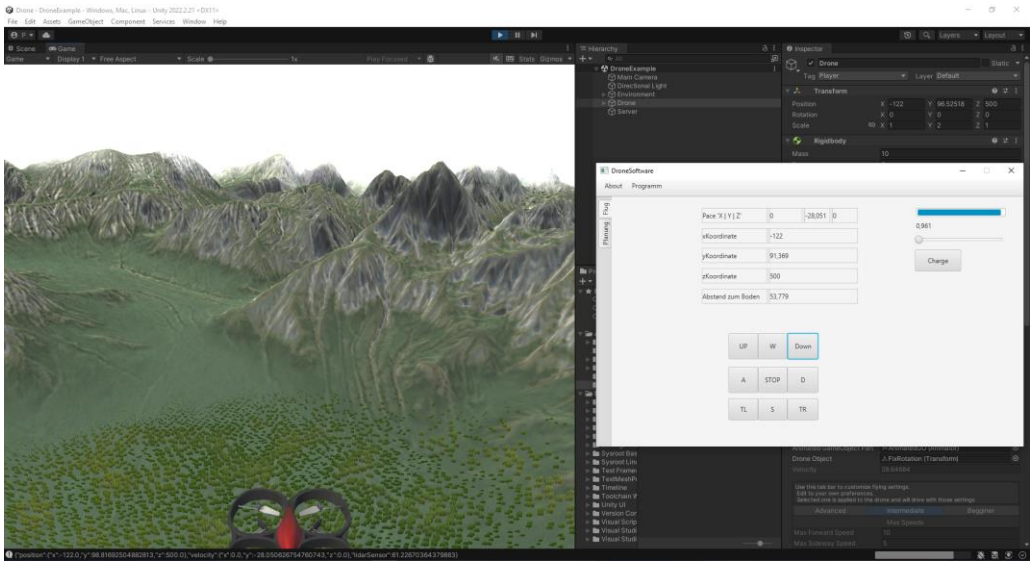

Abbildung 12: Test der Funktionalität für den "Down"-Knopf

Auch hier passiert das Erwartete. Dies ist daran zu erkennen, dass die Drohne offensichtlich höher beziehungsweise tiefer als zuvor fliegt. Außerdem passen die Ausgaben der grafischen Benutzungsoberfläche, da sich nur die y-Koordinate und die Geschwindigkeit in y-Richtung sowie der Abstand zum Boden ändern. Die Differenz zwischen dem Abstand zum Boden und der y-Koordinate sind allerdings gleichgeblieben, was die Proportionalität bestätigt.

Die Funktionalität der Dreh-Kommandos wird durch die Rotationskomponente in Unity überprüft. Standardmäßig ist diese auf 0 in y-Richtung. Sollte der „TL“-Knopf gedrückt werden, sollte dieser Wert bis zu einem Wert von -180 kleiner werden und dann umspringen auf 180, wonach er wieder bis -180 kleiner wird.

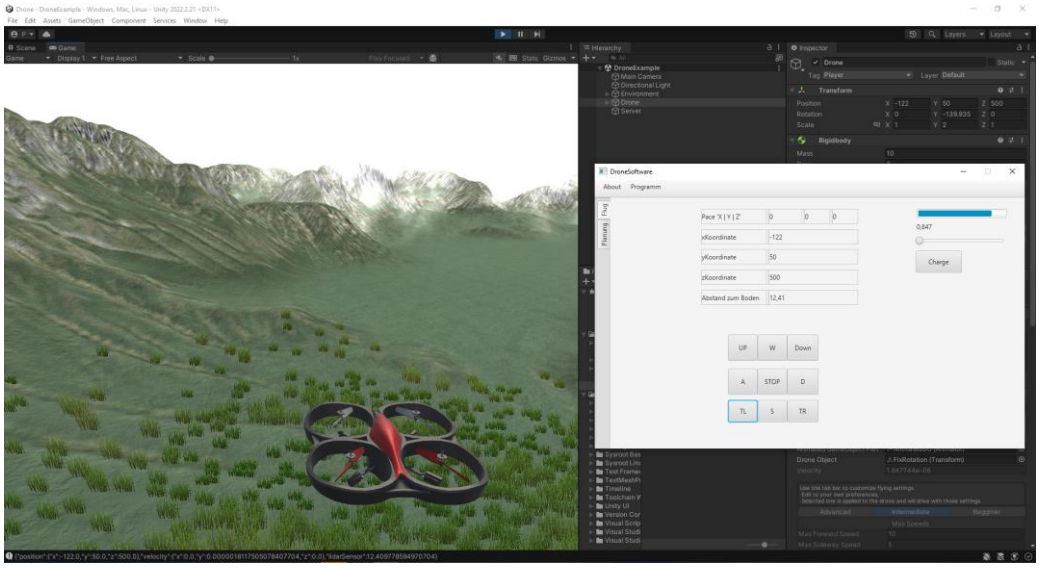

Abbildung 13: Test der Funktionalität für den "TL"-Knopf

Bei der Beobachtung des Programms und der Ausführung ist deutlich zu erkennen, dass sich die Drohne nach links um die y-Achse dreht. Die Rotationskomponente zeigt ebenfalls die korrekte Ausführung.

Für den Knopf „TR“ passiert das Gleiche in die entgegengesetzte Richtung.

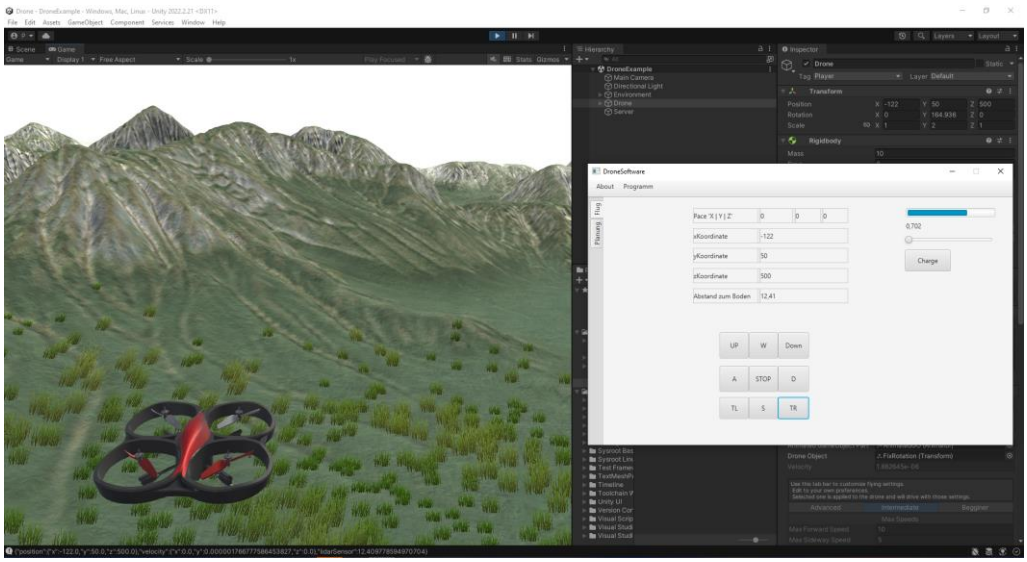

Abbildung 14: Test der Funktionalität für den "TR"-Knopf

Somit ist die Funktionalität aller Knöpfe erfolgreich getestet.

Anschließend wird die Eingabe einer Route in den Routenplaner getestet. Ziel des Tests ist, dass falsche Werte nicht zugelassen und eingetragen werden und die Möglichkeit besteht, diese zu korrigieren.

Zunächst werden valide Daten eingegeben, um zu testen, ob die Eintragung überhaupt funktioniert. Danach werden mehrere Punkte hinzugefügt und getestet, ob sich einzelne Punkte löschen lassen.

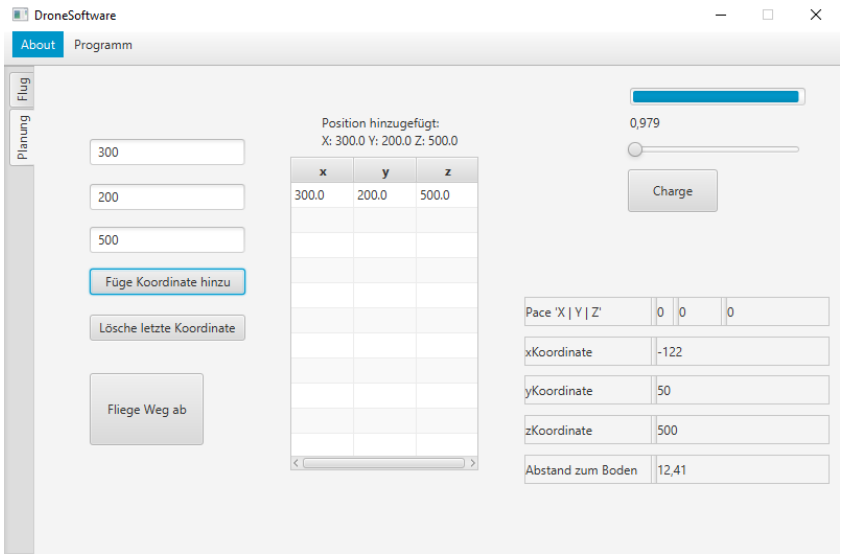

Abbildung 15: Hinzufügen einer Koordinate

Wie gewünscht wird die eingegebene Position der Liste hinzugefügt. Ein Textfeld bestätigt die Eintragung. Nun werden mehrere Daten hinzugefügt und dann der „Lösche letzte Koordinate“-Knopf betätigt.

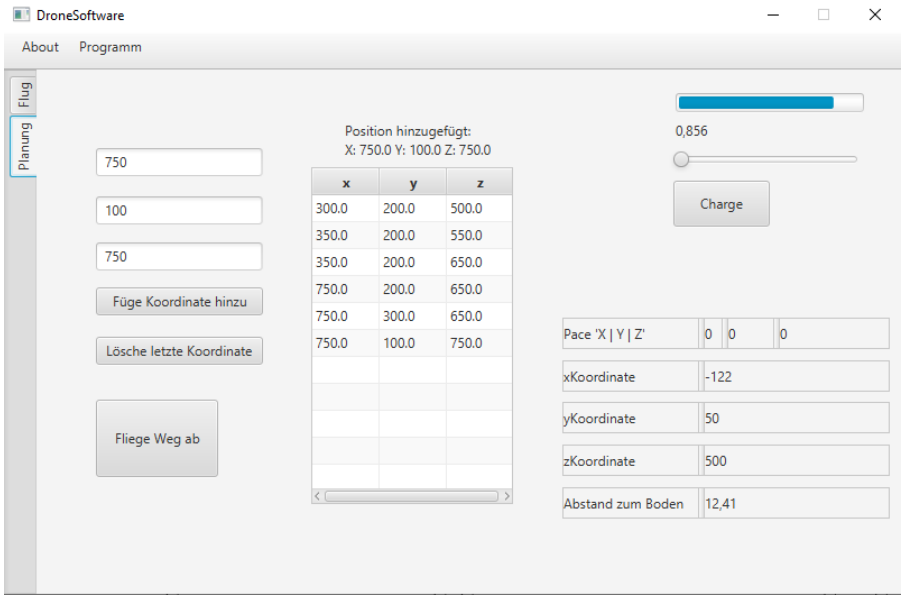

Abbildung 16: Liste nach der Eintragung mehrerer Koordinaten

An dieser Stelle wird auf den Knopf „Lösche letzte Koordinate“ gedrückt.

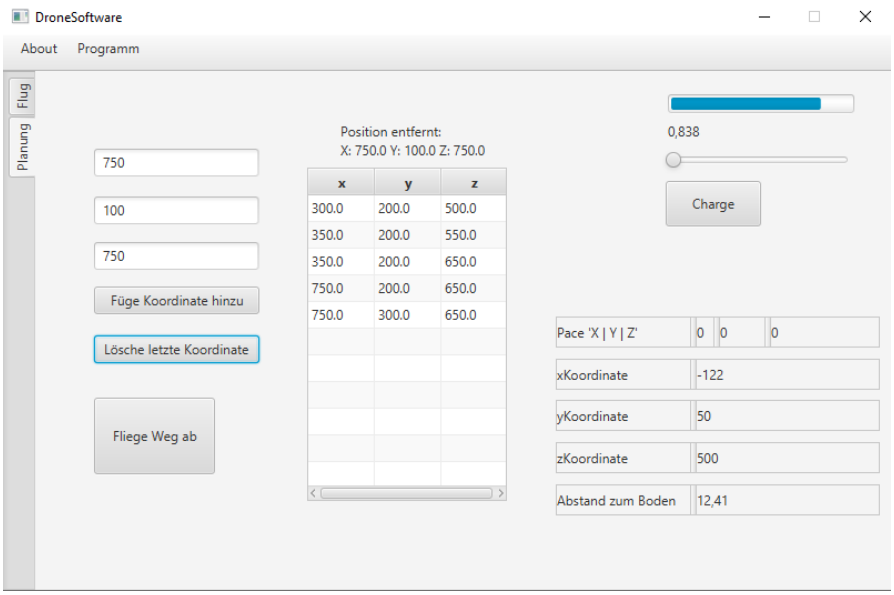

Abbildung 17: Tabelle nach Knopfdruck von "Lösche letzte Koordinate"

Wie zu sehen wird die letzte Koordinate gelöscht und das Programm steht zu einer Eingabe einer neuen Koordinate bereit. (vgl. Abbildungen 15 und 16).

Nun sollen falsche Eingaben getestet werden. Dazu werden zunächst Buchstaben und Sonderzeichen eingetragen. Außerdem wird das Trennen einer validen Zahl durch ein Dezimalkomma statt eines Dezimalpunktes getestet.

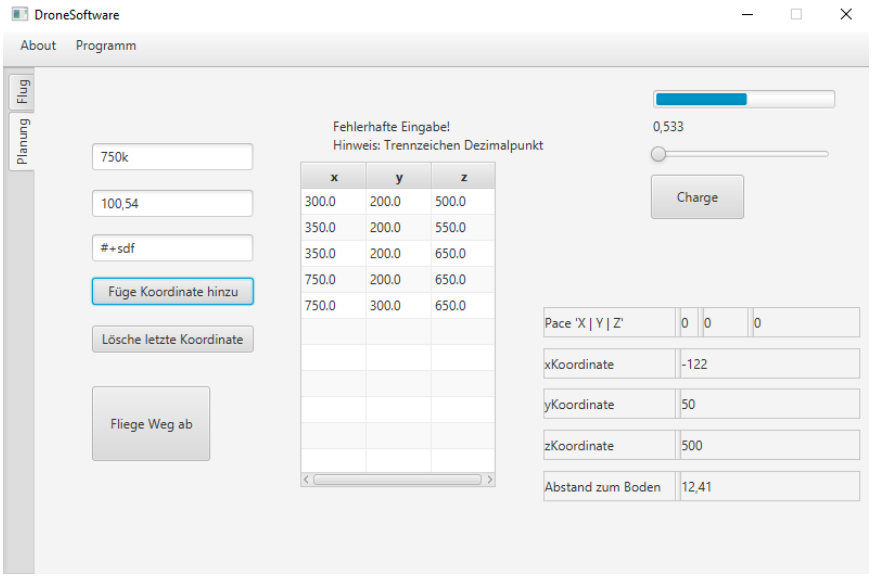

Abbildung 18: Test falscher Eingaben

Die Koordinate wird wie gewünscht nicht eingetragen. Außerdem wird ein Text angezeigt, dass eine fehlerhafte Eingabe vorliegt. Als Hinweis wird auf den Dezimalpunkt hingewiesen. Die Eingabe kann anschließend korrigiert werden und das Programm kann ungehindert agieren.

Die letzte Funktionalität ist das Abfliegen der Route. Dazu wird eine Route eingegeben und dokumentiert, ob die Punkte erreicht werden. Abgeflogen wird die Route nach der Betätigung des „Fliege Route ab“-Knopfes. Vom Startpunkt fliegt die Drohne zur ersten Zielkoordinate.

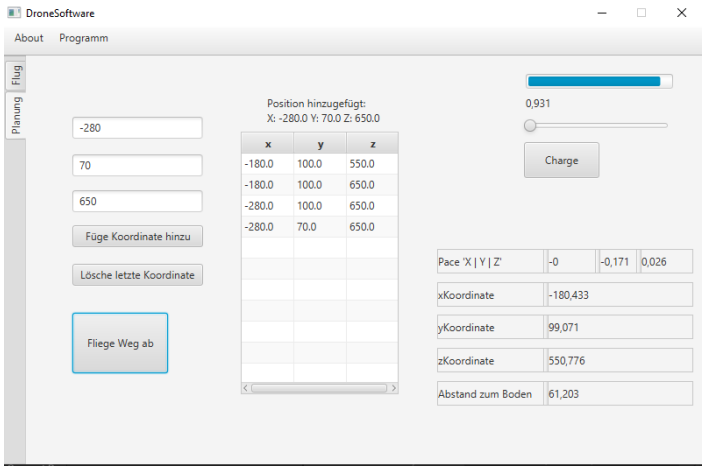

Abbildung 19: Erreichen der ersten Zielkoordinate

Die Zielkoordinate wurde nur um eine Einheit in y-Richtung verfehlt. Dort hält sie für 5 Sekunden und fliegt dann zur zweiten Zielkoordinate.

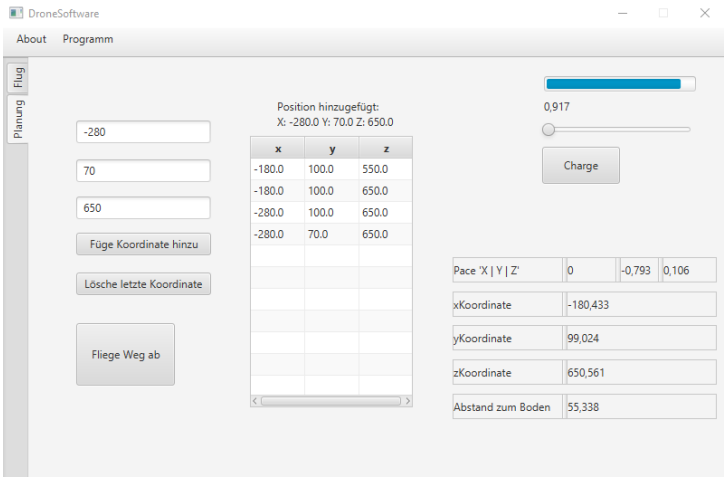

Abbildung 20: Erreichen der zweiten Zielkoordinate

Die zweite Zielkoordinate wurde ebenfalls nur um eine Einheit in y-Richtung verfehlt. Auch hier wird 5 Sekunden gehalten, bevor der Weg zur dritten Zielkoordinate fortgesetzt wird.

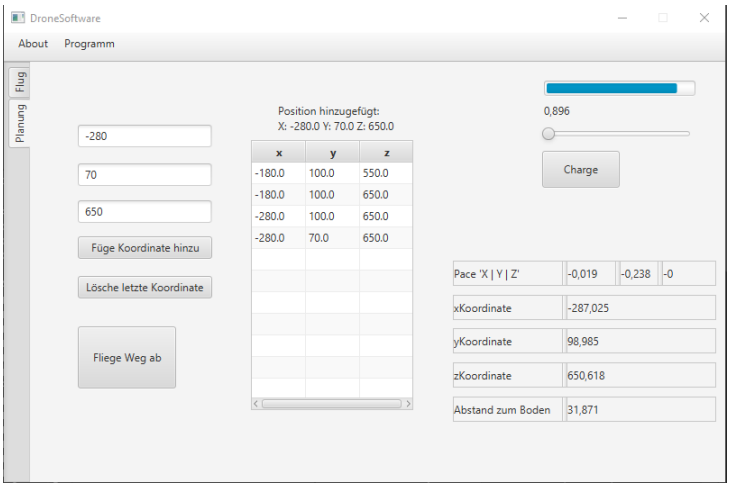

Abbildung 21: Erreichen der dritten Zielkoordinate

Diese Zielkoordinate wurde um sieben Einheiten in x-Richtung und eine Einheit in y-Richtung verfehlt. Dies lag an einer Kollision mit dem Terrain. Trotzdem wurde die Koordinate ungefähr erreicht, wo die Drohne hält und dann nach 5 Sekunden die vierte Zielkoordinate ansteuert.

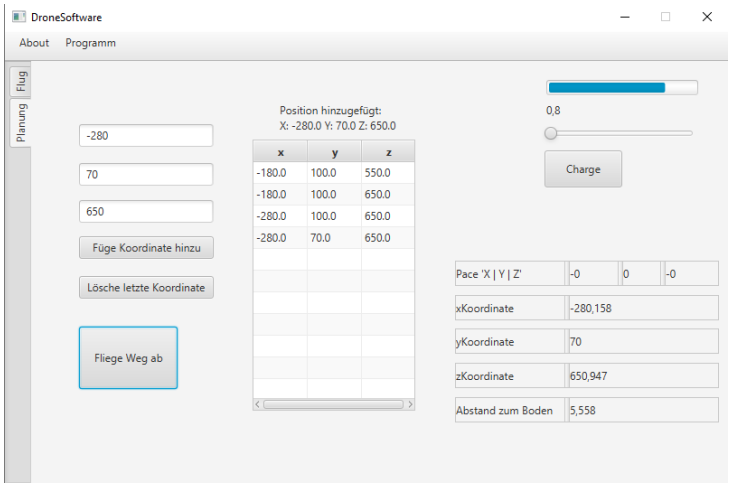

Abbildung 22: Erreichen der vierten Zielkoordinate

Die vierte Zielkoordinate wird wieder genau getroffen. Die größte Abweichung ist die Abweichung von 0,947 Einheiten in z-Richtung.

Wie zu sehen, fliegt die Drohne alle Zielkoordinaten bis auf kleinste Abweichungen automatisch an und liefert an den Stellen die Koordinaten sowie den Abstand zum Boden. An

jeder Stelle hält die Drohne für fünf Sekunden, in dieser Zeit sind die Screenshots der Abbildungen 18 bis 21 entstanden.

7.2 Fazit des Tests

Der Test aller Komponenten des Programms lief gut. Alle Funktionalität der grafischen Benutzungsoberfläche wurden getestet und liefern das gewünschte Ergebnis. Das Programm kann von einer benutzenden Person genutzt werden, ohne dass die Gefahr eines Absturzes des Programms besteht.

Lediglich der Test des geplanten Flugs lief nicht optimal. Die Drohne auf den exakten Ort zu manövrieren, ist durch die Steuerungsweise schwierig. Dies stellt aber kein großes Problem dar, da eine benutzende Person in diesem Fall durch die manuelle Steuerung einen Punkt nachmessen könnte.

8. Anwendungsbeispiel

Im Anwendungsbeispiel möchte die benutzende Person eine Route planen. Vom Startpunkt aus soll zunächst zu der Koordinate (-175 | 55 | 665), dann zu der Koordinate (-50 | 80 | 420) und zum Schluss zur Koordinate (-355 | 180 | 200) fliegen. An jedem Punkt soll der Abstand zum Boden ausgelesen werden. Dazu gibt die benutzende Person die Koordinaten in die Liste der Planungssoftware ein.

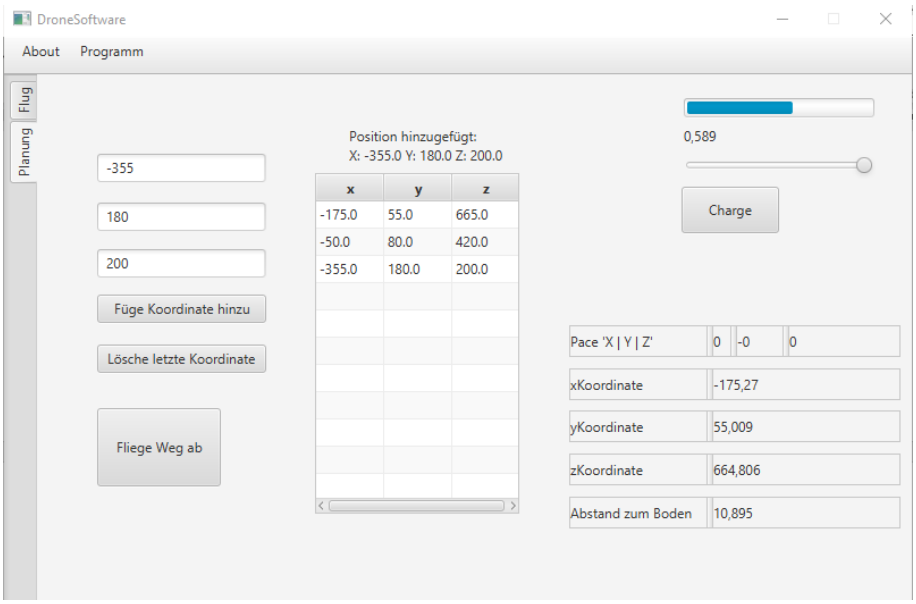

Abbildung 23: Ankunft am ersten Zielpunkt im Anwendungsbeispiel

Am ersten Zielpunkt angekommen kann die benutzende Person die Koordinaten einsehen und den Abstand zum Boden von 10,895 Einheiten notieren. Anschließend fliegt die Drohne den zweiten Zielpunkt an.

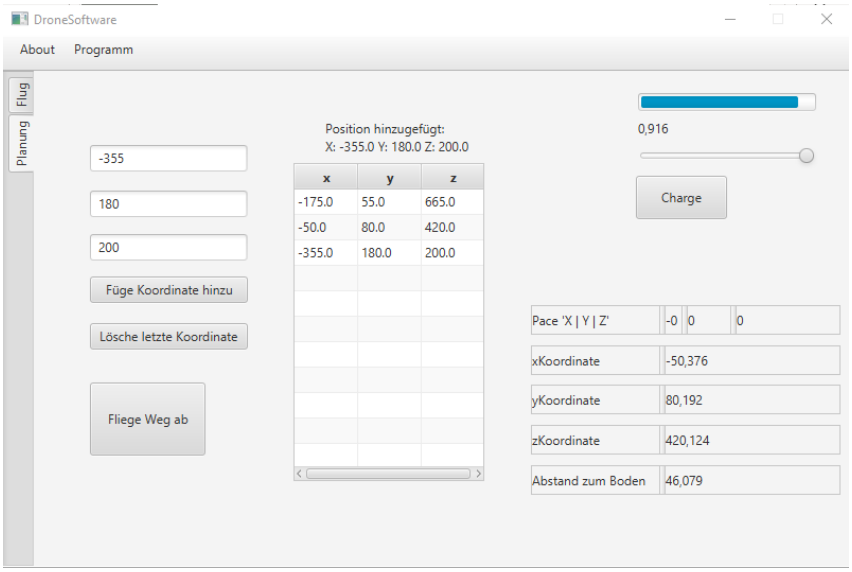

Abbildung 24: Ankunft am zweiten Zielpunkt im Anwendungsbeispiel

Bei der Ankunft an diesem Punkt kann wieder der Abstand zum Boden notiert werden. Dieser ist an der zweiten Zielkoordinate 46,079 Einheiten. Nun wir der letzte Zielpunkt angesteuert.

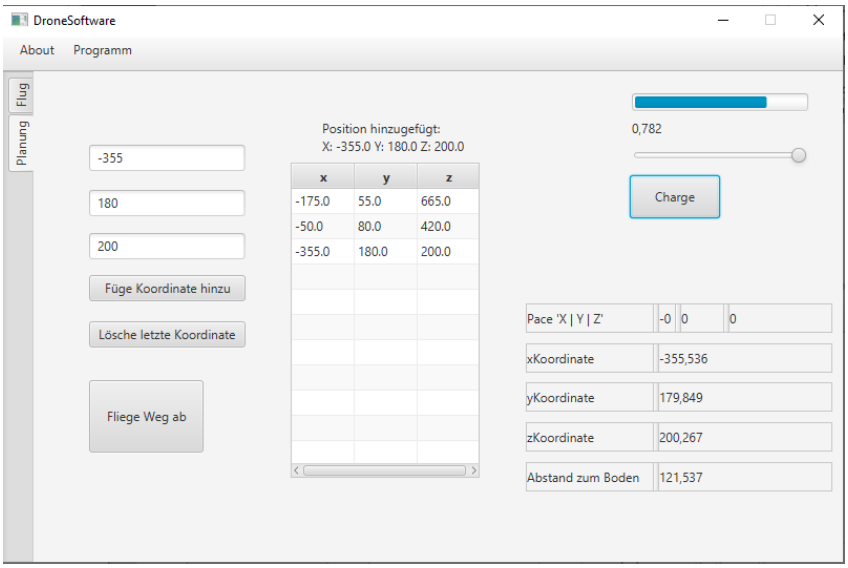

Abbildung 25: Ankunft am letzten Zielpunkt im Anwendungsbeispiel

Auch dort wird die Distanz zum Boden herausgeschrieben. An diesem Punkt beträgt diese 121,537 Einheiten.

Anschließend können neue Daten eingegeben werden. Außerdem könnten Ziele wieder manuell angeflogen werden. Da die benutzende Person dann aber alle wichtigen Information hat, kann sie das Programm über den Menüpunkt „Programm beenden“ verlassen.

9. Verweise

Hier finden sich alle Verweise zu im Dokument verwendeten Quellen.

9.1 Abbildungsverzeichnis

9.2 Verzeichnis über Auszüge aus dem Quellcode

9.3 Onlineverzeichnis

Link zu der eMotion Software: https://ageagle.com/drone-software/emotion/. Zuletzt aufgerufen am 03.07.2023.

Link zu IBM-Language-Design: https://www.ibm.com/design/language/. Zuletzt aufgerufen am 03.07.2023.

Link zum Austauschdienst GigaMove: https://help.itc.rwth-aachen.de/service/1jeqhtat4k0o3/. Zuletzt aufgerufen am 03.07.2023.

Link zum Tutorial für ein Menü: https://docs.oracle.com/javafx/2/ui_controls/menu_controls.htm#BABGHADI. Zuletzt aufgerufen am 03.07.2023.

Link zur Aufgabenstellung für die Entwicklungsarbeit: https://aulis.hs-bremen.de/login.php?target=file_1829937_download&cmd=force_login&lang=de. Zuletzt aufgerufen am 03.07.2023.

Link zur Modellierung einer professionellen Drohne: https://de.mathworks.com/discovery/drone-simulation.html. Zuletzt aufgerufen am 03.07.2023.

10. Anhang

Angehängt sind der gesamte Quellcode und die Javadoc-Dokumentation.

10.1 Quellcode

Hier ist der gesamte Quellcode, nach Paketen und Klassen sortiert, abgebildet.

10.1.1 Control Package

10.1.1.1 DroneControllClient

```
001    package org.example.control;
002    import com.google.gson.Gson;
003    import org.example.res.Constants;
004    import org.example.res.Strings;
005    import org.example.model.*;
006
007    import java.io.BufferedReader;
008    import java.io.IOException;
009    import java.io.InputStreamReader;
010    import java.io.PrintWriter;
011    import java.net.Socket;
012    import java.net.UnknownHostException;
013
014
015    /**This class is the client of the DroneController setting up a
connection to Unity.
016     * @author philipp.hennken
017     * @version 18.0.2
018     */
019    public class DroneControllClient implements Runnable
020    {
021        private Socket socket = null;
022        private final Drone drone =
DroneController.getClassInstance().getDrone();
023        private DataPackage dataPackage = new DataPackage(new
Position(0,0,0), 0.0, new Velocity(0,0,0));
024
025        /**
026         Runs the communication loop between the controller and the
Unity simulation.This method establishes a socket connection to the Unity
simulation at the specified host and port.
027         It continuously sends movement commands to the simulation and
receives data packages containing drone position and lidar sensor
information. The loop runs indefinitely until the program is terminated.
028         @pre The host and port values must be valid and accessible.
029         @post The drone's position and lidar sensor information are
updated based on the received data package.
030         */
031        public void run()
032        {
033            socket = buildConnection(Strings.LOCALHOST,
Constants.PORT_CONTROLLER_UNITY);
```

```
034             Gson gsonObject = new Gson();
035 
036             while (true)
037             {
038                 try
039                 {
040                     dataPackage =
gsonObject.fromJson(sendCommandAndWaitForAnswer(socket,
gsonObject.toJson(drone.getMovement())),DataPackage.class);
041                 } catch (ConnectionLostException e)
042                 {
043                     throw new RuntimeException(e);
044                 }
045                 drone.setPosition(dataPackage.getPosition());
046 
drone.getLidarSensor().setDistanceToGround(dataPackage.getLidarSensor());
047                 drone.setVelocity(dataPackage.getVelocity());
048                 System.out.println(drone.getVelocity().toString());
049                 sleep(Constants.GET_DATA_SLEEP);
050             }
051         }
052 
053 
054         /**
055          This method pauses the current thread for the given duration,
allowing other threads to execute.
056          The duration is specified in milliseconds.
057          @param duration The duration in milliseconds for which the
thread should be sleeping.
058          @throws RuntimeException if the thread is interrupted while
sleeping.
059          @pre The duration must be a non-negative value.
060          @post The execution of the current thread is paused for the
specified duration.
061          */
062         private void sleep(long duration)
063         {
064             try
065             {
066                 Thread.sleep(duration);
067             }
068             catch (InterruptedException e)
069             {
070                 throw new RuntimeException(e);
071             }
072         }
073 
074         /**
075          * It sends the specified command to the simulation and waits
for the response.
076          * The response is returned as a string.
077          *
078          *@param socket The socket used for communication with the Unity
simulation.
079          *@param command The command to be sent to the Unity simulation.
080          *@return The response received from the Unity simulation.
081          *@throws RuntimeException if an I/O error occurs during the
communication.
082          *@pre The socket must be open and connected to the Unity
simulation.
083          *@pre The command must be a non-null string.
084          *@post The command is sent to the Unity simulation, and the
response is returned as a string.
```

```
085          *@post The input and output streams with the Unity simulation
are properly closed.
086          */
087         private String sendCommandAndWaitForAnswer(Socket socket, String
command) throws ConnectionLostException
088         {
089             PrintWriter outToUnity = null;
090             BufferedReader inFromUnity = null;
091             String inFromUnityString;
092             try
093             {
094                 outToUnity = new PrintWriter(socket.getOutputStream());
095                 inFromUnity = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
096                 outToUnity.println(command);
097                 outToUnity.flush();
098                 inFromUnityString = inFromUnity.readLine();
099                 return inFromUnityString;
100             }
101             catch (IOException | NullPointerException exception)
102             {
103                 throw new
ConnectionLostException(Strings.CONNECTION_LOST_TO_UNITY);
104             }
105         }
106
107
108         /** This method sets up a connection between a host and port and
this Socket.
109          *
110          *@param host The host address to connect to.
111          *@param port The port number to connect to.
112          *@return The socket object representing the connection to the
host and port.
113          *@pre The host must be a valid IP address or domain name.
114          *@post If an exception occurs during the connection building
process, an appropriate error message is printed.
115          */
116         private Socket buildConnection(String host, int port)
117         {
118             Socket newSocket = null;
119             try
120             {
121                 newSocket = new Socket(host, port);
122             }
123             catch (UnknownHostException e)
124             {
125                 System.out.println(Strings.ERROR_IP_ADRESSE);
126             }
127             catch (IOException ee)
128             {
129
System.out.println(Strings.IO_ERROR_DURING_CONNECTION_BUILD);
130             }
131             catch (SecurityException eee)
132             {
133                 System.out.println(Strings.CONNECTION_DENIED);
134             }
135             catch (IllegalArgumentException eeee)
136             {
137                 System.out.println(Strings.ILLEGAL_PORT);
138             }
139             finally
```

```
140             {
141                 return newSocket;
142             }
143         }
144 
145         public Drone getDrone()
146         {
147             return drone;
148         }
149     }
```

10.1.1.2 DroneController

```
001     package org.example.control;
002     import javafx.beans.property.DoubleProperty;
003     import javafx.beans.property.SimpleDoubleProperty;
004     import javafx.beans.value.ChangeListener;
005     import javafx.beans.value.ObservableValue;
006     import org.example.model.Battery;
007     import org.example.model.Drone;
008     import org.example.model.Position;
009 
010     /**This class controls the drone and its attributes. The drone
instance is created and managed here.
011      * @author philipp.hennken
012      * @version 18.0.2
013      */
014     public class DroneController
015     {
016         private static DroneController classInstance;
017 
018 
019         /** Returns the singleton instance of the DroneController class.
020          *
021          * @return DroneController singleton instance that is not null.
022          * @pre none.
023          * @post The singleton instance of the DroneController class is
returned.
024          */
025         public static DroneController getClassInstance()
026         {
027             if (classInstance == null)
028             {
029                 classInstance = new DroneController();
030             }
031             return classInstance;
032         }
033 
034 
035         private final Drone drone;
036         private final DoubleProperty xVelocityProperty;
037         private DoubleProperty yVelocityProperty;
038         private DoubleProperty zVelocityProperty;
039         private final DoubleProperty xKoordinateProperty;
040         private final DoubleProperty yKoordinateProperty;
041         private final DoubleProperty zKoordinateProperty;
042         private final DoubleProperty chargeLevelProperty;
043         private final DoubleProperty distanceToGroundProperty;
044 
045         /** This method sets every property to a new
SimpleDoubleProperty.
```

```
046          * It calls the propertyListeners method.
047          *
048          * @pre none.
049          * @post Every Property is set to a new SimpleDoubleProperty.
050          */
051         private DroneController()
052         {
053             this.drone = new Drone(new Position(0.0,0.0,0.0));
054             this.xVelocityProperty = new SimpleDoubleProperty();
055             this.yVelocityProperty = new SimpleDoubleProperty();
056             this.zVelocityProperty = new SimpleDoubleProperty();
057             this.xKoordinateProperty = new SimpleDoubleProperty();
058             this.yKoordinateProperty = new SimpleDoubleProperty();
059             this.zKoordinateProperty = new SimpleDoubleProperty();
060             this.chargeLevelProperty = new SimpleDoubleProperty();
061             this.distanceToGroundProperty = new SimpleDoubleProperty();
062             propertyListeners();
063         }
064 
065         /** For every property a listener monitors if the property
changed. Then the drone singleton wil be updated.
066          *
067          * @pre The necessary properties and drone instance must be
properly initialized.
068          * @post Property listeners are set up.
069          */
070         private void propertyListeners()
071         {
072             this.xVelocityProperty.addListener(new
ChangeListener<Number>()
073             {
074                 @Override
075                 public void changed(ObservableValue<? extends Number>
observableValue, Number alteGeschwindigkeit, Number updatedXVelocity)
076                 {
077                     drone.getVelocity().setxVelocity((Double)
updatedXVelocity);
078                 }
079             });
080 
081             this.yVelocityProperty.addListener(new
ChangeListener<Number>()
082             {
083                 @Override
084                 public void changed(ObservableValue<? extends Number>
observableValue, Number alteGeschwindigkeit, Number updatedYVelocity)
085                 {
086                     drone.getVelocity().setyVelocity((Double)
updatedYVelocity);
087                 }
088             });
089 
090             this.zVelocityProperty.addListener(new
ChangeListener<Number>()
091             {
092                 @Override
093                 public void changed(ObservableValue<? extends Number>
observableValue, Number alteGeschwindigkeit, Number updatedZVelocity)
094                 {
095                     drone.getVelocity().setzVelocity((Double)
updatedZVelocity);
096                 }
097             });
```

```
098
099             this.distanceToGroundProperty.addListener(new
ChangeListener<Number>()
100             {
101                 @Override
102                 public void changed(ObservableValue<? extends Number>
observableValue, Number oldDistanceToGround, Number
updatedDistanceToGround)
103                 {
104                     drone.getLidarSensor().setDistanceToGround((Double)
updatedDistanceToGround);
105                 }
106             });
107
108             this.chargeLevelProperty.addListener(new
ChangeListener<Number>()
109             {
110                 @Override
111                 public void changed(ObservableValue<? extends Number>
observableValue, Number oldChargeLevel, Number updatedChargeLevel)
112                 {
113                     drone.setBattery(new Battery((Double)
updatedChargeLevel));
114                 }
115             });
116
117
118             this.xKoordinateProperty.addListener(new
ChangeListener<Number>()
119             {
120                 @Override
121                 public void changed(ObservableValue<? extends Number>
observableValue, Number alteXKoordinate, Number neueXKoordinate)
122                 {
123                     drone.getPosition().setX((Double) neueXKoordinate);
124                 }
125             });
126
127             this.yKoordinateProperty.addListener(new
ChangeListener<Number>()
128             {
129                 @Override
130                 public void changed(ObservableValue<? extends Number>
observableValue, Number alteYKoordinate, Number neueYKoordinate)
131                 {
132                     drone.getPosition().setY((Double) neueYKoordinate);
133                 }
134             });
135
136             this.zKoordinateProperty.addListener(new
ChangeListener<Number>()
137             {
138                 @Override
139                 public void changed(ObservableValue<? extends Number>
observableValue, Number alteZKoordinate, Number neueZKoordinate)
140                 {
141                     drone.getPosition().setZ((Double) neueZKoordinate);
142                 }
143             });
144         }
145
146         public synchronized DoubleProperty getxVelocityProperty()
147         {
```

```
148             return xVelocityProperty;
149         }
150 
151 
152         public DoubleProperty getXKoordinateProperty()
153         {
154             return xKoordinateProperty;
155         }
156 
157 
158         public DoubleProperty getYKoordinateProperty()
159         {
160             return yKoordinateProperty;
161         }
162 
163 
164         public DoubleProperty getZKoordinateProperty()
165         {
166             return zKoordinateProperty;
167         }
168 
169         public DoubleProperty getChargeLevelProperty()
170         {
171             return chargeLevelProperty;
172         }
173 
174         public DoubleProperty getDistanceToGroundProperty()
175         {
176             return distanceToGroundProperty;
177         }
178 
179         public DoubleProperty getyVelocityProperty()
180         {
181             return yVelocityProperty;
182         }
183 
184         public DoubleProperty yVelocityPropertyProperty()
185         {
186             return yVelocityProperty;
187         }
188 
189         public DoubleProperty getzVelocityProperty()
190         {
191             return zVelocityProperty;
192         }
193 
194         public DoubleProperty zVelocityPropertyProperty()
195         {
196             return zVelocityProperty;
197         }
198 
199         public Drone getDrone()
200         {
201             return drone;
202         }
203     }
```

10.1.1.3 DroneControllerMain

```
01    package org.example.control;
02 
```

```
03    /**This class starts threads for the DroneController.
04     * @author philipp.hennken
05     * @version 18.0.2
06     */
07    public class DroneControllerMain
08    {
09        /**This method is called when the programm is executed.
10         * It starts every Thread needed for the connections.
11         *
12         * @param args The command-line arguments passed to the program.
13         * @pre
14         * @post startThreads() is called.
15         */
16        public static void main(String[] args)
17        {
18            startThreads();
19        }
20
21        /** This method starts the client and server threads for the
connections of the drone-controllers and Unity.
22         * @pre None.
23         * @post The client and server threads are created and started
for drone control.
24         * @post The drone control operations are performed concurrently
by the client and server threads.
25         */
26        private static void startThreads()
27        {
28            DroneControllClient droneClient = new DroneControllClient();
29            DroneControllServer droneServer = new
DroneControllServer(droneClient.getDrone());
30            Thread clientThread = new Thread(droneClient);
31            Thread serverThread = new Thread(droneServer);
32            clientThread.start();
33            serverThread.start();
34        }
35
36
37    }
```

10.1.1.4 DroneControllServer

```
01    package org.example.control;
02    import com.google.gson.Gson;
03    import org.example.res.Constants;
04    import org.example.res.Strings;
05    import org.example.model.DataPackage;
06    import org.example.model.Drone;
07    import org.example.model.Position;
08    import org.example.model.Velocity;
09
10    import java.io.BufferedReader;
11    import java.io.IOException;
12    import java.io.InputStreamReader;
13    import java.io.PrintWriter;
14    import java.net.ServerSocket;
15    import java.net.Socket;
16
17    /** This class represents a server for handling request of the
planning software and sends data of a drone
18     * it gets from Unity.
```

```java
     * @author philipp.hennken
     * @version 18.0.2
     */
    public class DroneControllServer implements Runnable
    {

        private ServerSocket serverSocket;
        private final Drone drone;
        private DataPackage dataPackage = new DataPackage(new
Position(0,0,0), 0.0 , new Velocity(0,0,0));

        /** This constructor sets the drone to the given drone.
         *
         * @param drone Drone object that is associated with the server.
         * @pre drone must not be null.
         * @post The DroneControllServer instance is created with the
specified Drone object.
         */
        public DroneControllServer(Drone drone)
        {
            this.drone = drone;
        }

        /** This method runs the Thread for the server of the
droneController wich sends data to the planning software if
         * it gets a request.
         *
         * @throws RuntimeException if an I/O error occurs during the
server operation.
         * @pre The server socket port must be available and not already
in use.
         * @post The server is running and handling client connections
for data exchange with the drone.
         */
        @Override
        public void run()
        {
            try
            {
                serverSocket = new
ServerSocket(Constants.PORT_CONTROLLER_PLANNING);
            }
            catch (IOException e)
            {
                throw new RuntimeException(e);
            }

            Socket client;
            PrintWriter printWriter;
            BufferedReader bufferedReader = null;
            Gson gson = new Gson();
            try
            {
                System.out.println(Strings.SERVER_IS_RUNNING);
                while(true)
                {
                    dataPackage.setPosition(drone.getPosition());

dataPackage.setLidarSensor(drone.getLidarSensor().getDistanceToGround());
                    dataPackage.setVelocity(drone.getVelocity());
                    client = serverSocket.accept();
                    printWriter = new
PrintWriter(client.getOutputStream(), true);
```

```
73                      printWriter.println(gson.toJson(dataPackage));
74                      bufferedReader = new BufferedReader(new
InputStreamReader(client.getInputStream()));
75                      drone.setMovement(bufferedReader.readLine());
76                      printWriter.close();
77                      client.close();
78                  }
79              }
80              catch (IOException e)
81              {
82                  throw new RuntimeException(e);
83              }
84          }
85      }
```

10.1.1.5 PlannedFlight

```
001     package org.example.control;
002
003     import org.example.res.Constants;
004     import org.example.res.Strings;
005     import org.example.model.Position;
006     import org.example.view.FreeFlightDelegate;
007     import org.example.view.Planning;
008
009     /** This class plans a flight to every coordinate the user puts in
the list of the GUI.
010      * @author philipp.hennken
011      * @version 18.0.2
012      */
013     public class PlannedFlight implements Runnable
014     {
015
016         private final DroneController droneController;
017
018         /** This constructor sets an object of DroneController to the
DroneController instance and returns it.
019          *
020          * @pre The DroneController instance must be not null.
021          * @post The droneController field is assigned an instance of
the DroneController class.
022          */
023         public PlannedFlight()
024         {
025            this.droneController = DroneController.getClassInstance();
026         }
027
028         /**This method runs the thread that lets the drone fly to every
coordinate in the table of the GUI.
029          *
030          * @throws RuntimeException if an InterruptedException occurs
during the thread sleep.
031          * @pre
032          * @post The drone has flown to each destination specified in
the Planning.
033          *
034          */
035         @Override
036         public void run()
037         {
038             for (Position position: Planning.getPositions())
```

```
039             {
040                 double destinationX = position.getX();
041                 double destinationY = position.getY();
042                 double destinationZ = position.getZ();
043 
flyToDestination(destinationX,destinationY,destinationZ);
044                 try
045                 {
046                     Thread.sleep(Constants.WAIT_AT_DESTINATION_SLEEP);
047                 } catch (InterruptedException e)
048                 {
049                     throw new RuntimeException(e);
050                 }
051             }
052         }
053 
054         /**This method lets the drone fly to the correct destination.
055          *
056          * @param destinationX The x-coordinate of the destination
position.
057          * @param destinationY The y-coordinate of the destination
position.
058          * @param destinationZ The z-coordinate of the destination
position.
059          * @pre The drone must be in a navigable state.
060          * @post The drone has flown to the destination coordinates.
061          */
062         private void flyToDestination(double destinationX, double
destinationY, double destinationZ)
063         {
064 
flyToYDestinationAndStop(Constants.MAX_HIGHT_OF_MOUNTAINS_PLUS_TWENTY);
065             flyToXDestinationAndStop(destinationX);
066             flyToZDestinationAndStop(destinationZ);
067             flyToYDestinationAndStop(destinationY);
068         }
069 
070         /** This method lets the drone fly to the correct y-coordinate.
071          *
072          * @param destinationY The y-coordinate destination to which the
drone should be moved.
073          * @pre The destinationY parameter must be within the valid
range of y-coordinates.
074          * @post The drone has reached the specified y-coordinate
destination and is hovering in place.
075          */
076         private void flyToYDestinationAndStop(double destinationY)
077         {
078             if (droneController.getYKoordinateProperty().getValue() <=
destinationY)
079             {
080                 while
(droneController.getYKoordinateProperty().getValue() <= destinationY -
Constants.BRAKING_DISTANCE_Y)
081                 {
082 
FreeFlightDelegate.setCommandToController(Strings.UP);
083                 }
084             }
085             else
086             {
087                 while
(droneController.getYKoordinateProperty().getValue() >= destinationY +
```

```
Constants.BRAKING_DISTANCE_Y)
088                 {
089
FreeFlightDelegate.setCommandToController(Strings.DOWN);
090                 }
091             }
092
FreeFlightDelegate.setCommandToController(Strings.NOT_MOVING);
093         }
094
095         /** This method lets the drone fly to the correct z-coordinate.
096          *
097          * @param destinationZ The z-coordinate destination to which the
drone should be moved.
098          * @pre The destinationZ parameter must be within the valid
range of z-coordinates.
099          * @post The drone has reached the specified y-coordinate
destination and is hovering in place.
100          */
101         private void flyToZDestinationAndStop(double destinationZ)
102         {
103             if (droneController.getZKoordinateProperty().getValue() <=
destinationZ)
104             {
105                 while
(droneController.getZKoordinateProperty().getValue() <= destinationZ -
Constants.BRAKING_DISTANCE_Z)
106                 {
107
FreeFlightDelegate.setCommandToController(Strings.W);
108                 }
109             }
110             else
111             {
112                 while
(droneController.getZKoordinateProperty().getValue() >= destinationZ +
Constants.BRAKING_DISTANCE_Z)
113                 {
114
FreeFlightDelegate.setCommandToController(Strings.S);
115                 }
116             }
117
FreeFlightDelegate.setCommandToController(Strings.NOT_MOVING);
118         }
119
120         /** This method lets the drone fly to the correct x-coordinate.
121          *
122          * @param destinationX The x-coordinate destination to which the
drone should be moved.
123          * @pre The destinationX parameter must be within the valid
range of x-coordinates.
124          * @post The drone has reached the specified x-coordinate
destination and is hovering in place.
125          */
126         private void flyToXDestinationAndStop(double destinationX)
127         {
128             if (droneController.getXKoordinateProperty().getValue() <=
destinationX)
129             {
130                 while
(droneController.getXKoordinateProperty().getValue() <= destinationX -
Constants.BRAKING_DISTANCE_X)
```

```
131                 {
132
FreeFlightDelegate.setCommandToController(Strings.D);
133                 }
134             }
135             else
136             {
137                 while
(droneController.getXKoordinateProperty().getValue() >= destinationX +
Constants.BRAKING_DISTANCE_X)
138                 {
139
FreeFlightDelegate.setCommandToController(Strings.A);
140                 }
141             }
142
FreeFlightDelegate.setCommandToController(Strings.NOT_MOVING);
143         }
144     }
```

10.1.1.6 PlanningSoftwareClient

```
01    package org.example.control;
02
03    import com.google.gson.Gson;
04    import javafx.application.Platform;
05    import org.example.res.Constants;
06    import org.example.res.Strings;
07    import org.example.model.*;
08    import org.example.view.FreeFlightDelegate;
09
10    import java.io.BufferedReader;
11    import java.io.IOException;
12    import java.io.InputStreamReader;
13    import java.io.PrintWriter;
14    import java.net.Socket;
15
16    /**This class
17     * @author philipp.hennken
18     * @version 18.0.2
19     */
20    public class PlanningSoftwareClient implements Runnable
21    {
22        private final Drone drone =
DroneController.getClassInstance().getDrone();
23        private DataPackage dataPackage = new DataPackage(new
Position(0,0,0), 0.0, new Velocity(0,0,0));
24
25        /** Runs the continuous loop to communicate with the drone
controller server and update the drone's state.
26         * @pre The drone object and its components are initialized.
27         * @post The drone's attributes are updated.
28         */
29        @Override
30        public synchronized void run()
31        {
32            try
33            {
34                Gson gson = new Gson();
35                BufferedReader inFromDroneControllerServer = null;
36                PrintWriter outToDroneController = null;
```

```
37                  int i = 0;
38                  while (true)
39                  {
40                      Socket clientSocket = new Socket(Strings.LOCALHOST,
Constants.PORT_CONTROLLER_PLANNING);
41                      inFromDroneControllerServer = new BufferedReader(new
InputStreamReader(clientSocket.getInputStream()));
42                      outToDroneController = new
PrintWriter(clientSocket.getOutputStream(), true);
43
outToDroneController.println(FreeFlightDelegate.getCommandToController());
44                      dataPackage =
gson.fromJson(inFromDroneControllerServer.readLine(),DataPackage.class);
45                      drone.getBattery().calculateActuatorData();
46                      drone.setLidarSensor(new
LidarSensor(dataPackage.getLidarSensor()));
47                      drone.setPosition(dataPackage.getPosition());
48                      drone.setVelocity(new
Velocity(drone.getVelocity().getxVelocity(),
drone.getVelocity().getyVelocity(), drone.getVelocity().getzVelocity()));
49                      Platform.runLater(() ->
50                      {
51                          setPropertyValues();
52                      });
53                      Thread.sleep(Constants.GET_DATA_SLEEP);
54                  }
55              } catch(IOException | InterruptedException e)
56                  {
57                      throw new RuntimeException(e);
58                  }
59
60          }
61
62          /**Sets the property values of the drone controller based on the
current state of the drone.
63           * @pre The drone object and its components are initialized.
64           * @post The property values of the drone controller are updated.
65           */
66          private void setPropertyValues()
67          {
68
DroneController.getClassInstance().getXKoordinateProperty().setValue(drone.
getPosition().getX());
69
DroneController.getClassInstance().getYKoordinateProperty().setValue(drone.
getPosition().getY());
70
DroneController.getClassInstance().getZKoordinateProperty().setValue(drone.
getPosition().getZ());
71
DroneController.getClassInstance().getxVelocityProperty().setValue(drone.ge
tVelocity().getxVelocity());
72
DroneController.getClassInstance().getyVelocityProperty().setValue(drone.ge
tVelocity().getyVelocity());
73
DroneController.getClassInstance().getzVelocityProperty().setValue(drone.ge
tVelocity().getzVelocity());
74
DroneController.getClassInstance().getChargeLevelProperty().setValue(drone.
getBattery().getChargeLevel());
75
DroneController.getClassInstance().getDistanceToGroundProperty().setValue(d
```

```
rone.getLidarSensor().getDistanceToGround());
76          }
77      }
```

10.1.2 Model Package

10.1.2.1 Actuator

```
01      package org.example.model;
02
03      /** Tis is the abstract class for Actuators the drone has.
04       * @author philipp.hennken
05       * @version 18.0.2
06       */
07      public abstract class Actuator
08      {
09          /**
10           * The abstract class for the calculation of data of Actuators
the drone has.
11           */
12          public abstract void calculateActuatorData();
13      }
```

10.1.2.2 Battery

```
01      package org.example.model;
02
03      import javafx.fxml.FXMLLoader;
04      import javafx.scene.control.Dialog;
05      import org.example.res.Constants;
06
07      import java.awt.*;
08      import java.io.File;
09      import java.io.IOException;
10
11      /**This class represents a battery of a drone.
12       * @author philipp.hennken
13       * @version 18.0.2
14       */
15      public class Battery extends Actuator
16      {
17          private double chargeLevel;
18
19          /**Creates a battery with a chargeLevel in percent.
20           *
21           * @param chargeLevel The initial charge level of the drone's
battery.
22           * @pre chargeLevel value must be a valid double integer.
23           * @post the batteries charge level is set to the parameters
value.
24           * @post the batteryLevel has been calculated.
25           */
26          public Battery(double chargeLevel)
27          {
28              this.chargeLevel = chargeLevel;
29              calculateActuatorData();
30          }
31
32          /** This method lowers the chargeLevel at every method call.
33           *
34           * @pre The chargeLevel is a valid double value.
```

```
35          * @post The chargeLevel is 0.000002 percent points lower tha
before.
36          */
37         @Override
38         public void calculateActuatorData()
39         {
40                 chargeLevel = chargeLevel -
Constants.CHARGE_LEVEL_CONSTANT;
41                 if (chargeLevel <= 0)
42                 {
43                     System.exit(0);
44                 }
45         }
46
47         public double getChargeLevel()
48         {
49             return chargeLevel;
50         }
51
52         public void setChargeLevel(double chargeLevel)
53         {
54             this.chargeLevel = chargeLevel;
55         }
56     }
```

10.1.2.3 ConnectionLostException

```
01     package org.example.model;
02
03     public class ConnectionLostException extends Exception
04     {
05         public ConnectionLostException(String s)
06         {
07             System.out.println(s);
08             System.exit(1);
09         }
10     }
```

10.1.2.4 DataPackage

```
01     package org.example.model;
02
03     /**This class contains every data the program gets from unity.
04      * @author philipp.hennken
05      * @version 18.0.2
06      */
07     public class DataPackage
08     {
09         private Position position;
10         private Velocity velocity;
11         private double lidarSensor;
12
13
14         /**This constructor initializes the DataPackage object with the
given position and lidar sensor data.
15          * @param position The position data to be included in the
DataPackage.
16          * @param lidarSensor The lidar sensor data to be included in the
DataPackage.
```

```
17          * @pre The lidarSensor parameter must be a valid double value
representing lidar sensor data.
18          * @post The position and idarSensor field are assigned to the
position data.
19          */
20         public DataPackage(Position position, double lidarSensor,
Velocity velocity)
21         {
22             this.position = position;
23             this.lidarSensor = lidarSensor;
24             this.velocity = velocity;
25         }
26
27         public Position getPosition()
28         {
29             return position;
30         }
31         public double getLidarSensor()
32         {
33             return lidarSensor;
34         }
35
36         public void setLidarSensor(double lidarSensor)
37         {
38             this.lidarSensor = lidarSensor;
39         }
40
41         public void setPosition(Position position)
42         {
43             this.position = position;
44         }
45
46         public Velocity getVelocity()
47         {
48             return velocity;
49         }
50
51         public void setVelocity(Velocity velocity)
52         {
53             this.velocity = velocity;
54         }
55     }
```

10.1.2.5 Drone

```
01     package org.example.model;
02
03     /** THis class implements a drone with all its components.
04      * @author philipp.hennken
05      * @version 18.0.2
06      */
07     public class Drone
08     {
09         private Battery battery;
10         private Velocity velocity = new Velocity(0,0,0);
11         private Position position;
12         private String movement;
13         private LidarSensor lidarSensor = new LidarSensor(0.0);
14
15         /** This constructor creates a new Drone with the specified
position and a Battery with chargeLevel one.
```

```
16          *
17          * @param position The initial position of the drone.
18          * @pre The position parameter must be a valid Position object.
19          * @post A new instance of the Drone class is created.
20          */
21         public Drone(Position position)
22         {
23             this.position = position;
24             this.battery = new Battery(1);
25         }
26
27         public Velocity getVelocity()
28         {
29             return velocity;
30         }
31
32         public void setVelocity(Velocity velocity)
33         {
34             this.velocity = velocity;
35         }
36
37         public Position getPosition()
38         {
39             return position;
40         }
41
42         public void setPosition(Position position)
43         {
44             this.position = position;
45         }
46
47         public String getMovement()
48         {
49             return movement;
50         }
51
52         public void setMovement(String movement)
53         {
54             this.movement = movement;
55         }
56
57         public void setBattery(Battery battery)
58         {
59             this.battery = battery;
60         }
61
62         public Battery getBattery()
63         {
64             return battery;
65         }
66
67         public LidarSensor getLidarSensor()
68         {
69             return lidarSensor;
70         }
71
72         public void setLidarSensor(LidarSensor lidarSensor)
73         {
74             this.lidarSensor = lidarSensor;
75         }
76     }
```

10.1.2.6 LidarSensor

```
01    package org.example.model;
02
03    /**This class represents a lidarSensor of the drone measuring the
distance from the drone to the ground.
04     * @author philipp.hennken
05     * @version 18.0.2
06     */
07    public class LidarSensor extends Sensor
08    {
09        private double distanceToGround;
10
11        /**
12         * @param distanceToGround The distance to the ground measured by
the Lidar sensor.
13         * @pre The distance to the ground must be a valid value.
14         * @post A LidarSensor object is created with the
distanceToGround.
15         */
16        public LidarSensor(double distanceToGround)
17        {
18            this.distanceToGround = distanceToGround;
19        }
20
21        public double getDistanceToGround()
22        {
23            return distanceToGround;
24        }
25
26        public void setDistanceToGround(double distanceToGround)
27        {
28            this.distanceToGround = distanceToGround;
29        }
30    }
```

10.1.2.7 Position

```
01    package org.example.model;
02
03    /**This class keeps a position with its attributes.
04     * @author philipp.hennken
05     * @version 18.0.2
06     */
07    public class Position
08    {
09        private double x;
10
11        //height of the drone is y
12        private double y;
13        private double z;
14
15        /** Creates a position object in a three-dimensional map.
16         * @param x The X coordinate.
17         * @param y The Y coordinate.
18         * @param z The Z coordinate.
19         * @pre The coordinates must be valid values.
20         * @post A Position object is created with the specified
coordinates.
21         *
22         */
```

```
23          public Position(double x, double y, double z)
24          {
25              this.x = x;
26              this.y = y;
27              this.z = z;
28          }
29
30         // private Vector3D positionVector = new Vector3D(xCoordinate,
yCoordinate, zCoordinate);
31
32
33          public double getX()
34          {
35              return x;
36          }
37
38          public void setX(double x)
39          {
40              this.x = x;
41          }
42
43          public double getY()
44          {
45              return y;
46          }
47
48          public void setY(double y)
49          {
50              this.y = y;
51          }
52
53          public double getZ()
54          {
55              return z;
56          }
57
58          public void setZ(double z)
59          {
60              this.z = z;
61          }
62
63          @Override
64          public String toString()
65          {
66              return "Position{" +
67                      "xCoordinate=" + x +
68                      ", yCoordinate=" + y +
69                      ", zCoordinate=" + z +
70                      '}';
71          }
72      }
```

10.1.2.8 Sensor

```
1     package org.example.model;
2
3     public abstract class Sensor
4     {
5
6     }
```

10.1.2.9 Velocity

```
01    package org.example.model;
02
03    import com.google.gson.annotations.SerializedName;
04
05    /**This class keeps a velocity with its attributes.
06     * @author philipp.hennken
07     * @version 18.0.2
08     */
09    public class Velocity
10    {
11        @SerializedName("x")
12        private double xVelocity;
13        @SerializedName("y")
14        private double yVelocity;
15        @SerializedName("z")
16        private double zVelocity;
17
18        /**Constructs a new Velocity object with the specified
velocities.
19         *
20         * @param xVelocity the x-component of the velocity
21         * @param yVelocity the y-component of the velocity
22         * @param zVelocity the z-component of the velocity
23         * @pre xVelocity, yVelocity, and zVelocity must be valid double
values.
24         * @post The Velocity object is initialized with the provided
velocities.
25         */
26        public Velocity(double xVelocity, double yVelocity, double
zVelocity)
27        {
28            this.xVelocity = xVelocity;
29            this.yVelocity = yVelocity;
30            this.zVelocity = zVelocity;
31        }
32
33        public double getxVelocity()
34        {
35            return xVelocity;
36        }
37
38        public void setxVelocity(double xVelocity)
39        {
40            this.xVelocity = xVelocity;
41        }
42
43        public double getyVelocity()
44        {
45            return yVelocity;
46        }
47
48        public void setyVelocity(double yVelocity)
49        {
50            this.yVelocity = yVelocity;
51        }
52
53        public double getzVelocity()
54        {
```

```
55              return zVelocity;
56          }
57
58          public void setzVelocity(double zVelocity)
59          {
60              this.zVelocity = zVelocity;
61          }
62
63          @Override
64          public String toString()
65          {
66              return "Velocity{" +
67                      "xVelocity=" + xVelocity +
68                      ", yVelocity=" + yVelocity +
69                      ", zVelocity=" + zVelocity +
70                      '}';
71          }
72      }
```

10.1.3 Res Package

10.1.3.1 Constants

```
01      package org.example.res;
02
03      /** This class contains every necessary constant.
04       * @author philipp.hennken
05       * @version 18.0.2
06       */
07      public interface Constants
08      {
09          long GET_DATA_SLEEP = 1;
10          int PORT_CONTROLLER_PLANNING = 55555;
11          int WINDOW_WIDTH = 800;
12          int WINDOW_HIGHT = 500;
13          double NUMBER_TO_PERCENT = 100;
14          long WAIT_AT_DESTINATION_SLEEP = 5000;
15          double MAX_HIGHT_OF_MOUNTAINS_PLUS_TWENTY = 150;
16          double BRAKING_DISTANCE_Y = 18.6;
17          double BRAKING_DISTANCE_X = 7.6;
18          double BRAKING_DISTANCE_Z = 20;
19          int PORT_CONTROLLER_UNITY = 10666;
20          double MIN_Z_COODINATE = 180;
21          double MIN_X_COODINATE = -800;
22          double MAX_Z_COODINATE = 1800;
23          double MAX_X_COODINATE = 800;
24          double MIN_Y_COODINATE = 0;
25          double CHARGE_LEVEL_CONSTANT = 0.000002;
26      }
```

10.1.3.2 Strings

```
01      package org.example.res;
02
03      /**This class contains every necessary string used in the program.
04       * @author philipp.hennken
05       * @version 18.0.2
06       */
07      public interface Strings
```

```
08    {
09        String ERROR_IP_ADRESSE = "IP-Adresse konnte nicht ermittelt
werden.";
10        String IO_ERROR_DURING_CONNECTION_BUILD = "Beim Verbindungsaufbau
ist ein IO-Fehler aufgetreten.";
11        String CONNECTION_DENIED = "Verbindungsaufbau nicht erlaubt.";
12        String ILLEGAL_PORT = "Illegaler Port.";
13        String LOCALHOST = "localhost";
14        String SERVER_IS_RUNNING = "Server läuft";
15        String NOT_MOVING = "notMoving";
16        String W = "W";
17        String A = "A";
18        String S = "S";
19        String D = "X";
20        String UP = "Up";
21        String DOWN = "Down";
22        String TURN_LEFT = "TurnLeft";
23        String TURN_RIGHT = "TurnRight";
24        String TITLE_DRONE_SOFTWARE = "DroneSoftware";
25        String PATH_TO_MAIN_SCREEN_FXML =
"C:\\Users\\philipp.hennken\\eclipse-
workspace\\EA_Philipp_Hennken_Semester2\\src\\main\\resources\\org.example\
\mainScreen.fxml";
26        String LITTLE_X = "x";
27        String LITTLE_Y = "y";
28        String LITTLE_Z = "z";
29        String POSITION_ADDED = "Position hinzugefügt: \n";
30        String WRONG_INPUT_COORDINATES = "Fehlerhafte Eingabe!\nHinweis:
Trennzeichen Dezimalpunkt";
31        String X_COLON = "X: ";
32        String Y_COLON = " Y: ";
33        String Z_COLON = " Z: ";
34        String POSITION_REMOVED = "Position entfernt: \n";
35        String NO_COORDINATES_TO_REMOVE = "Keine löschbare Koordinate \n
verfügbar.";
36        String CONNECTION_LOST_TO_UNITY = "Die Verbindung zu Unity ist
verloren gegangen";
37        String OUT_OF_BORDER = "Die gewünschten Koordinaten liegen
\naußerhalb der Terrain-Fläche.";
38    }
```

10.1.4 View Package

10.1.4.1 FreeFlightDelegate

```
001    package org.example.view;
002    import javafx.fxml.FXML;
003    import javafx.scene.control.Label;
004    import javafx.scene.control.ProgressBar;
005    import javafx.scene.control.Slider;
006    import org.example.res.Constants;
007    import org.example.res.Strings;
008    import org.example.control.DroneController;
009
010
011    /**This class is the connection between the planned flight and the
GUI.
012     * @author philipp.hennken
013     * @version 18.0.2
014     */
015    public class FreeFlightDelegate
```

```
016    {
017        private static String commandToController = Strings.NOT_MOVING;
018        public Label anzeigeX;
019        public Label anzeigeY;
020        public Label anzeigeZ;
021        public Label anzeigeXVelocity;
022        public ProgressBar chargeLevelProgressBar;
023        public Label chargeLevelLabel;
024        public Slider chargingSlider;
025        public Label abstandZumBoden;
026        public Label anzeigeYVelocity;
027        public Label anzeigeZVelocity;
028
029        /** Calls the method to set every binding.
030         * Is called when the FreeFlightDelegate is referenced.
031         *
032         * @pre none.
033         * @post Every binding will be made.
034         */
035        @FXML
036        private void initialize()
037        {
038            Planning.setBindings(anzeigeX, anzeigeY, anzeigeZ,
anzeigeXVelocity, abstandZumBoden, chargeLevelProgressBar,
chargeLevelLabel, anzeigeYVelocity, anzeigeZVelocity);
039        }
040
041        /** If the W-Button is clicked. The commandToController value
will be set to "W".
042         * @pre W-Button is pushed.
043         * @post commandToController value is "W".
044         */
045        public void onSendWClick()
046        {
047            commandToController = Strings.W;
048        }
049
050        /** If the A-Button is clicked. The commandToController value
will be set to "A".
051         @pre A-Button is pushed.
052         @post commandToController value is "A".
053         */
054        public void onSendAClick()
055        {
056            commandToController = Strings.A;
057        }
058
059        /** If the D-Button is clicked. The commandToController value
will be set to "D".
060         *
061
062         @pre D-Button is pushed.
063         @post commandToController value is "D".
064         */
065        public void onSendDClick()
066        {
067            commandToController = Strings.D;
068        }
069
070        /** If the S-Button is clicked. The commandToController value
will be set to "S".
071         @pre S-Button is pushed.
072         @post commandToController value is "S".
```

```
073          */
074         public void onSendSClick()
075         {
076             commandToController = Strings.S;
077         }
078 
079         /** If the "notMoving" button is clicked. The
commandToController value will be set to "notMoving".
080          @pre "notMoving" button is pushed.
081          @post commandToController value is "notMoving".
082          */
083         public void onSendStopClick()
084         {
085             commandToController = Strings.NOT_MOVING;
086         }
087 
088         /** If the "up" button is clicked. The commandToController value
will be set to "up".
089          @pre "up" button is pushed.
090          @post commandToController value is "up".
091          */
092         public void onSendUpClick()
093         {
094             commandToController = Strings.UP;
095         }
096 
097         /** If the "down" button is clicked. The commandToController
value will be set to "down".
098          @pre "down" button is pushed.
099          @post commandToController value is "down".
100          */
101         public void onSendDownClick()
102         {
103             commandToController = Strings.DOWN;
104         }
105 
106         /** If the "turnLeft" button is clicked. The commandToController
value will be set to "turnLeft".
107          @pre "turnLeft" button is pushed.
108          @post commandToController value is "turnLeft".
109          */
110         public void onSendTurnLeftClick()
111         {
112             commandToController = Strings.TURN_LEFT;
113         }
114 
115         /** If the "turnRight" button is clicked. The
commandToController value will be set to "turnRight".
116          * @pre "turnRight" button is pushed.
117          * @post commandToController value is "turnRight".
118          */
119         public void onSendTurnRightClick()
120         {
121             commandToController = Strings.TURN_RIGHT;
122         }
123 
124         /**
125          * If the chargeFrone button is clicked the chargeLevel will set
to the sliders Value.
126          * @pre The chargingSlider value is set.
127          * @post The drone's battery charge level is updated.
128          */
129         public void chargeDrone()
```

```
130         {
131 
DroneController.getClassInstance().getDrone().getBattery().setChargeLevel(c
hargingSlider.getValue() / Constants.NUMBER_TO_PERCENT);
132         }
133 
134 
135 
136         public static String getCommandToController()
137         {
138             return commandToController;
139         }
140 
141         public static void setCommandToController(String command)
142         {
143             commandToController = command;
144         }
145 
146     }
```

10.1.4.2 MainFX

```
01    package org.example.view;
02 
03    import javafx.application.Application;
04    import javafx.fxml.FXMLLoader;
05    import javafx.scene.Scene;
06    import javafx.stage.Stage;
07    import org.example.res.Constants;
08    import org.example.res.Strings;
09    import org.example.control.DroneControllerMain;
10    import org.example.control.PlanningSoftwareClient;
11 
12    import java.io.File;
13    import java.io.IOException;
14 
15    /**This class starts every party of the program.
16     * @author philipp.hennken
17     * @version 18.0.2
18     */
19    public class MainFX extends Application
20    {
21 
22 
23        /** This method is called when the programm is started.
24         * calls the runProgram-method.
25         * @param args Program arguments.
26         * @pre Unity has to be started to be able to connect properly.
27         * @post the GUI is started.
28         * @post The DroneControllerMain is started.
29         * @post The PlanningSoftware thread is started.
30         */
31        public static void main(String[] args)
32        {
33            runProgram();
34 
35        }
36 
37        /**It starts the DroneController and the Thread for the planning
software.
38         * Additionally, it starts the GUI that is displayed to the user.
```

```
39          * @pre Unity has to be started to be able to connect properly.
40          * @post the GUI is started.
41          * @post The DroneControllerMain is started.
42          * @post The PlanningSoftware thread is started.
43          */
44         private static void runProgram()
45         {
46             DroneControllerMain.main(null);
47             PlanningSoftwareClient planningSoftwareClient = new
PlanningSoftwareClient();
48             Thread planningThread = new Thread(planningSoftwareClient);
49             planningThread.start();
50             launch();
51         }
52 
53         /**
54          * Starts the application by loading the main screen FXML file,
creating a scene, and displaying it on the stage.
55          * @param stage The primary stage of the JavaFX application.
56          * @throws Exception If an exception occurs during the start of
the application.
57          * @pre The application is launched.
58          * @post The main screen of the application is displayed on the
stage.
59          */
60         @Override
61         public void start(Stage stage) throws Exception
62         {
63             FXMLLoader fxmlLoaderMainScreen = new FXMLLoader();
64             String absolutePath = Strings.PATH_TO_MAIN_SCREEN_FXML;
65             fxmlLoaderMainScreen.setLocation(new
File(absolutePath).toURI().toURL());
66             Scene scene = null;
67 
68             try
69             {
70                 scene = new Scene(fxmlLoaderMainScreen.load(),
Constants.WINDOW_WIDTH, Constants.WINDOW_HIGHT);
71             } catch (IOException ioException)
72             {
73                 throw new RuntimeException(ioException);
74             }
75 
76             stage.setTitle(Strings.TITLE_DRONE_SOFTWARE);
77             stage.setResizable(false);
78             stage.setScene(scene);
79             stage.show();
80         }
81 
82     }
```

10.1.4.3 MainScreenDelegate

```
01     package org.example.view;
02     import javafx.fxml.FXMLLoader;
03     import javafx.scene.control.Dialog;
04     import java.io.File;
05     import java.io.IOException;
06 
07     /**This class starts every party of the program.
08      * @author philipp.hennken
```

```
09      * @version 18.0.2
10      */
11     public class MainScreenDelegate
12     {
13
14         public void onAboutClick() throws IOException
15         {
16             FXMLLoader loader = new FXMLLoader();
17             String absolutePath = "C:\\Users\\philipp.hennken\\eclipse-
workspace\\EA_Philipp_Hennken_Semester2\\src\\main\\resources\\org.example\
\about.fxml";
18             loader.setLocation(new File(absolutePath).toURI().toURL());
19             Dialog<Void> dialog = loader.load();
20             dialog.show();
21         }
22
23         /**Stops the execution of the program.
24          * @post The program execution is terminated with an exit status
of 0.
25          */
26         public void stopProgramm()
27         {
28             System.exit(0);
29         }
30     }
```

10.1.4.4 Planning

```
001     package org.example.view;
002
003     import javafx.beans.binding.Bindings;
004     import javafx.collections.FXCollections;
005     import javafx.collections.ObservableList;
006     import javafx.fxml.FXML;
007     import javafx.scene.control.*;
008     import javafx.scene.control.cell.PropertyValueFactory;
009     import javafx.util.converter.NumberStringConverter;
010     import org.example.res.Constants;
011     import org.example.res.Strings;
012     import org.example.control.PlannedFlight;
013     import org.example.control.DroneController;
014     import org.example.model.Position;
015
016     /**This class sets everything up that is necessary for displaying
data for the planned flight.
017      * @author philipp.hennken
018      * @version 18.0.2
019      */
020     public class Planning
021     {
022
023         public TextField xCoordinateEntry;
024         public TextField yCoordinateEntry;
025         public TextField zCoordinateEntry;
026         public TableView<Position> tableWithCoordinates;
027         public Label infoLabel;
028         public TableColumn<Position, Double> xCoordinateTableColumn;
029         public TableColumn<Position, Double> yCoordinateTableColumn;
030         public TableColumn<Position, Double> zCoordinateTableColumn;
031         public ProgressBar chargeLevelProgressBar;
032         public Label chargeLevelLabel;
```

```
033         private static ObservableList<Position> positions =
FXCollections.observableArrayList();
034         public Slider chargingSlider;
035         public Label anzeigeXVelocity;
036         public Label anzeigeX;
037         public Label anzeigeY;
038         public Label anzeigeZ;
039         public Label abstandZumBoden;
040         public Label anzeigeYVelocity;
041         public Label anzeigeZVelocity;
042
043         /**This Method is called when the FXML-File "planning.fxml" is
loaded. It sets the ValueFactories for the table columns
044          * and binds the Charge-Level-ProgressBar to it´s Property as
well as the ChargeLevel-Label.
045          * @pre
046          * @post
047          */
048         @FXML
049         public void initialize()
050         {
051            xCoordinateTableColumn.setCellValueFactory(new
PropertyValueFactory<Position,Double>(Strings.LITTLE_X));
052            yCoordinateTableColumn.setCellValueFactory(new
PropertyValueFactory<Position,Double>(Strings.LITTLE_Y));
053            zCoordinateTableColumn.setCellValueFactory(new
PropertyValueFactory<Position,Double>(Strings.LITTLE_Z));
054            setBindings(anzeigeX, anzeigeY, anzeigeZ, anzeigeXVelocity,
abstandZumBoden, chargeLevelProgressBar, chargeLevelLabel,
anzeigeYVelocity, anzeigeZVelocity);
055         }
056
057         /**
058          * Sets up bidirectional bindings between the provided labels,
progress bar, and the properties in the DroneController.
059          * @param anzeigeX The label displaying the X coordinate.
060          * @param anzeigeY The label displaying the Y coordinate.
061          * @param anzeigeZ The label displaying the Z coordinate.
062          * @param anzeigeXVelocity The label displaying the velocity.
063          * @param abstandZumBoden The label displaying the distance to
the ground.
064          * @param chargeLevelProgressBar The progress bar displaying the
battery charge level.
065          * @param chargeLevelLabel The label displaying the battery
charge level.
066          * @pre The setBindings method is called with the required
labels and progress bar.
067          * @post Bidirectional bindings are made between the labels,
progress bar, and the properties in DroneController.
068          */
069         public static void setBindings(Label anzeigeX, Label anzeigeY,
Label anzeigeZ, Label anzeigeXVelocity, Label abstandZumBoden, ProgressBar
chargeLevelProgressBar, Label chargeLevelLabel, Label anzeigeYVelocity,
Label anzeigeZVelocity)
070         {
071             Bindings.bindBidirectional(anzeigeX.textProperty(),
DroneController.getClassInstance().getXKoordinateProperty(), new
NumberStringConverter());
072             Bindings.bindBidirectional(anzeigeY.textProperty(),
DroneController.getClassInstance().getYKoordinateProperty(), new
NumberStringConverter());
073             Bindings.bindBidirectional(anzeigeZ.textProperty(),
DroneController.getClassInstance().getZKoordinateProperty(), new
```

```
NumberStringConverter());
074             Bindings.bindBidirectional(anzeigeXVelocity.textProperty(),
DroneController.getClassInstance().getxVelocityProperty(), new
NumberStringConverter());
075             Bindings.bindBidirectional(abstandZumBoden.textProperty(),
DroneController.getClassInstance().getDistanceToGroundProperty(), new
NumberStringConverter());
076
Bindings.bindBidirectional(chargeLevelProgressBar.progressProperty(),
DroneController.getClassInstance().getChargeLevelProperty());
077             Bindings.bindBidirectional(chargeLevelLabel.textProperty(),
DroneController.getClassInstance().getChargeLevelProperty(), new
NumberStringConverter());
078             Bindings.bindBidirectional(anzeigeYVelocity.textProperty(),
DroneController.getClassInstance().getyVelocityProperty(), new
NumberStringConverter());
079             Bindings.bindBidirectional(anzeigeZVelocity.textProperty(),
DroneController.getClassInstance().getzVelocityProperty(), new
NumberStringConverter());
080         }
081
082         /**Handles the event when the "Add Coordinates" button is
clicked.
083          * Updates the table view and info label.
084          * @pre The "Add Coordinates" button is clicked.
085          * @post The entered coordinates are added to the list of
positions if they are within the valid range.
086          */
087         public void onAddCoordinatesClick()
088         {
089             try
090             {
091                 if (Double.parseDouble(zCoordinateEntry.getText()) >=
Constants.MIN_Z_COODINATE &&
092                         Double.parseDouble(zCoordinateEntry.getText())
<= Constants.MAX_Z_COODINATE &&
093                         Double.parseDouble(xCoordinateEntry.getText())
>= Constants.MIN_X_COODINATE &&
094                         Double.parseDouble(xCoordinateEntry.getText())
<= Constants.MAX_X_COODINATE &&
095                         Double.parseDouble(yCoordinateEntry.getText())
>= Constants.MIN_Y_COODINATE)
096                 {
097                     positions.add(new Position(
098
Double.parseDouble(xCoordinateEntry.getText()),Double.parseDouble(yCoordina
teEntry.getText()),Double.parseDouble(zCoordinateEntry.getText())));
099                     tableWithCoordinates.setItems(positions);
100                     infoLabel.setText(Strings.POSITION_ADDED +
coordinateToString(positions.get(positions.size()-1)));;
101                 }
102                 else
103                 {
104                     infoLabel.setText(Strings.OUT_OF_BORDER);
105                 }
106             }catch (NumberFormatException numberFormatException)
107             {
108                 infoLabel.setText(Strings.WRONG_INPUT_COORDINATES);
109             }
110         }
111
112         /**Handles the event when the "Fly to Coordinates" button is
clicked.
```

```
113          * Creates a new instance of PlannedFlight and starts it in a
separate thread.
114          * @pre The "Fly to Coordinates" button is clicked.
115          * @post A new instance of PlannedFlight is created and started
in a separate thread.
116          */
117         public void onFlyToCoordinatesClick()
118         {
119             PlannedFlight plannedFlight = new PlannedFlight();
120             Thread plannedFlightThread = new Thread(plannedFlight);
121             plannedFlightThread.start();
122         }
123 
124         /**Converts a Position object to a string representation.
125          * @param position The Position object to convert.
126          * @return The string representation of the Position object.
127          */
128         private String coordinateToString(Position position)
129         {
130             return Strings.X_COLON + position.getX() + Strings.Y_COLON +
position.getY() + Strings.Z_COLON + position.getZ();
131         }
132 
133         /**Deletes the last coordinate from the list of positions.
134          * @pre The list of positions is not empty.
135          * @post The last coordinate is removed from the list of
positions.
136          */
137         public void deleteLastCoordinate()
138         {
139             try
140             {
141                 infoLabel.setText(Strings.POSITION_REMOVED +
coordinateToString(positions.get(positions.size()-1)));
142                 positions.remove(positions.get(positions.size()-1));
143             }catch (IndexOutOfBoundsException indexOutOfBoundsException)
144             {
145                 infoLabel.setText(Strings.NO_COORDINATES_TO_REMOVE);
146             }
147         }
148 
149         /** Charges the drone's battery based on the value of the
charging slider.
150          * @pre The chargingSlider value is within the valid range.
151          * @post The drone's battery charge level is set based on the
value of the chargingSlider.
152          */
153         public void chargeDrone()
154         {
155 
DroneController.getClassInstance().getDrone().getBattery().setChargeLevel(c
hargingSlider.getValue() / Constants.NUMBER_TO_PERCENT);
156         }
157 
158         public static ObservableList<Position> getPositions()
159         {
160             return positions;
161         }
162     }
```

10.1.5 FXML-Dateien

10.1.5.1 about.fxml

```
01    <?xml version="1.0" encoding="UTF-8"?>
02
03    <?import java.lang.*?>
04    <?import java.util.*?>
05    <?import javafx.scene.*?>
06    <?import javafx.scene.control.*?>
07    <?import javafx.scene.layout.*?>
08
09    <?import javafx.scene.text.Text?>
10    <Dialog xmlns="http://javafx.com/javafx/17.0.2-ea"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="org.example.view.MainScreenDelegate">
11        <dialogPane>
12            <DialogPane>
13                <content>
14                    <GridPane>
15                        <Text wrappingWidth="200" text="Version 1.0
Autor: Philipp Hennken Dieses Programm dient zum manuellen und geplanten
Flug einer Drohne in der Simulationssoftware Unity. Unter dem Reiter 'Flug'
kann die Drohne manuelle geflogen und die Werte ausgelesen werden. Unter
dem Reiter 'Planung' kann eine vorher definierte Route abgeflogen werden.
An jedem Punkt hält die Drohne für eine gewisse Zeit an, sodass die Daten
ausgelesen werden können. Bei Rückfragen kontaktieren Sie mich gerne unter
phennken@stud.hs-bremen.de" GridPane.columnIndex="0" GridPane.rowIndex="0"
/>
16
17                    </GridPane>
18                </content>
19                <buttonTypes>
20                    <ButtonType text="OK"/>
21                </buttonTypes>
22            </DialogPane>
23        </dialogPane>
24    </Dialog>
```

10.1.5.2 freeFlight.fxml

```
01    <?xml version="1.0" encoding="UTF-8"?>
02
03    <?import javafx.scene.control.*?>
04    <?import javafx.scene.layout.*?>
05
06    <AnchorPane prefHeight="400.0" prefWidth="600.0"
xmlns="http://javafx.com/javafx/17.0.2-ea"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="org.example.view.FreeFlightDelegate">
07
08        <Pane prefHeight="400.0" prefWidth="775.0"
BorderPane.alignment="CENTER">
09            <children>
10                <GridPane layoutX="219.0" layoutY="250.0"
prefHeight="168.0" prefWidth="168.0">
11                    <columnConstraints>
12                        <ColumnConstraints hgrow="SOMETIMES"
maxWidth="81.0" minWidth="10.0" prefWidth="75.0" />
13                        <ColumnConstraints hgrow="SOMETIMES"
maxWidth="78.0" minWidth="10.0" prefWidth="75.0" />
```

```
14                          <ColumnConstraints hgrow="SOMETIMES"
maxWidth="78.0" minWidth="10.0" prefWidth="100.0" />
15                      </columnConstraints>
16                      <rowConstraints>
17                          <RowConstraints maxHeight="63.0" minHeight="10.0"
prefHeight="47.0" vgrow="SOMETIMES" />
18                          <RowConstraints maxHeight="69.0" minHeight="0.0"
prefHeight="69.0" vgrow="SOMETIMES" />
19                          <RowConstraints minHeight="10.0"
prefHeight="30.0" vgrow="SOMETIMES" />
20                      </rowConstraints>
21                      <children>
22                          <Button mnemonicParsing="false"
onAction="#onSendWClick" prefHeight="49.0" prefWidth="59.0" text="W"
GridPane.columnIndex="1" />
23                          <Button mnemonicParsing="false"
onAction="#onSendAClick" prefHeight="49.0" prefWidth="73.0" text="A"
GridPane.rowIndex="1" />
24                          <Button mnemonicParsing="false"
onAction="#onSendDClick" prefHeight="49.0" prefWidth="73.0" text="D"
GridPane.columnIndex="2" GridPane.rowIndex="1" />
25                          <Button mnemonicParsing="false"
onAction="#onSendSClick" prefHeight="49.0" prefWidth="73.0" text="S"
GridPane.columnIndex="1" GridPane.rowIndex="2" />
26                          <Button mnemonicParsing="false"
onAction="#onSendStopClick" prefHeight="49.0" prefWidth="73.0" text="STOP"
GridPane.columnIndex="1" GridPane.rowIndex="1" />
27                          <Button mnemonicParsing="false"
onAction="#onSendUpClick" prefHeight="49.0" prefWidth="73.0" text="UP" />
28                          <Button mnemonicParsing="false"
onAction="#onSendDownClick" prefHeight="49.0" prefWidth="73.0" text="Down"
GridPane.columnIndex="2" />
29                          <Button mnemonicParsing="false"
onAction="#onSendTurnLeftClick" prefHeight="49.0" prefWidth="73.0"
text="TL" GridPane.rowIndex="2" />
30                          <Button mnemonicParsing="false"
onAction="#onSendTurnRightClick" prefHeight="49.0" prefWidth="73.0"
text="TR" GridPane.columnIndex="2" GridPane.rowIndex="2" />
31                      </children>
32                  </GridPane>
33                  <SplitPane dividerPositions="0.4241379310344828,
0.7758620689655172, 0.8931034482758621" layoutX="169.0" layoutY="21.0"
prefHeight="28.0" prefWidth="292.0">
34                      <items>
35                          <Label prefHeight="28.0" prefWidth="120.0"
text="Pace 'X | Y | Z'" />
36                          <Label fx:id="anzeigeXVelocity" prefHeight="26.0"
prefWidth="60.0" />
37                      <Label fx:id="anzeigeYVelocity" prefHeight="26.0"
prefWidth="60.0" />
38                      <Label fx:id="anzeigeZVelocity" prefHeight="26.0"
prefWidth="60.0" />
39                      </items>
40                  </SplitPane>
41                  <SplitPane dividerPositions="0.5" layoutX="169.0"
layoutY="59.0" prefHeight="28.0" prefWidth="292.0">
42                      <items>
43                          <Label prefHeight="28.0" prefWidth="120.0"
text="xKoordinate" />
44                          <Label fx:id="anzeigeX" prefHeight="26.0"
prefWidth="186.0" />
45                      </items>
46                  </SplitPane>
```

```
47                <SplitPane dividerPositions="0.5" layoutX="169.0"
layoutY="96.0" prefHeight="28.0" prefWidth="292.0">
48                    <items>
49                        <Label prefHeight="28.0" prefWidth="120.0"
text="yKoordinate" />
50                        <Label fx:id="anzeigeY" prefHeight="26.0"
prefWidth="188.0" />
51                    </items>
52                </SplitPane>
53                <SplitPane dividerPositions="0.5" layoutX="169.0"
layoutY="134.0" prefHeight="28.0" prefWidth="292.0">
54                    <items>
55                        <Label prefHeight="28.0" prefWidth="120.0"
text="zKoordinate" />
56                        <Label fx:id="anzeigeZ" prefHeight="26.0"
prefWidth="189.0" />
57                    </items>
58                </SplitPane>
59                <SplitPane dividerPositions="0.5" layoutX="169.0"
layoutY="172.0" prefHeight="28.0" prefWidth="292.0">
60                    <items>
61                        <Label prefHeight="26.0" prefWidth="120.0"
text="Abstand zum Boden" />
62                        <Label fx:id="abstandZumBoden" prefHeight="26.0"
prefWidth="189.0" />
63                    </items>
64                </SplitPane>
65             <ProgressBar fx:id="chargeLevelProgressBar" layoutX="570.0"
layoutY="21.0" nodeOrientation="LEFT_TO_RIGHT" prefHeight="18.0"
prefWidth="168.0" progress="0.0" />
66             <Label fx:id="chargeLevelLabel" layoutX="570.0" layoutY="40"
prefHeight="28.0" prefWidth="68.0" text="Label" />
67             <Slider fx:id="chargingSlider" layoutX="568.0"
layoutY="73.0" prefHeight="14.0" prefWidth="168.0" />
68             <Button layoutX="568.0" layoutY="99.0"
mnemonicParsing="false" onAction="#chargeDrone" prefHeight="41.0"
prefWidth="86.0" text="Charge" />
69
70
71           </children>
72        </Pane>
73    </AnchorPane>
```

10.1.5.3 mainScreen.fxml

```
01    <?xml version="1.0" encoding="UTF-8"?>
02
03    <?import javafx.scene.control.*?>
04    <?import javafx.scene.layout.*?>
05
06    <BorderPane maxHeight="-Infinity" maxWidth="-Infinity" minHeight="-
Infinity" minWidth="-Infinity" prefHeight="400.0" prefWidth="775.0"
xmlns="http://javafx.com/javafx/17.0.2-ea"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="org.example.view.MainScreenDelegate">
07
08            <top>
09            <MenuBar prefHeight="33.0" prefWidth="605.0">
10                <Menu text="About">
11                    <MenuItem onAction="#onAboutClick" text="About" />
12                </Menu>
```

```
13                <Menu text="Programm">
14                    <MenuItem onAction="#stopProgramm" text="Programm
beenden" />
15                </Menu>
16            </MenuBar>
17            </top>
18       <center>
19          <TabPane prefHeight="367.0" prefWidth="218.0" side="LEFT"
tabClosingPolicy="UNAVAILABLE" BorderPane.alignment="CENTER">
20            <tabs>
21                <Tab text="Flug">
22                    <content>
23                        <fx:include source="freeFlight.fxml" />
24                    </content>
25                </Tab>
26                <Tab text="Planung">
27                  <content>
28                    <fx:include source="planning.fxml" />
29                </content>
30              </Tab>
31            </tabs>
32          </TabPane>
33       </center>
34          </BorderPane>
```

10.1.5.4 planning.fxml

```
01    <?xml version="1.0" encoding="UTF-8"?>
02
03    <?import javafx.scene.control.*?>
04    <?import javafx.scene.layout.*?>
05
06    <Pane prefHeight="400.0" prefWidth="775.0"
xmlns="http://javafx.com/javafx/17.0.2-ea"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="org.example.view.Planning">
07       <children>
08          <TextField layoutX="53.0" layoutY="70.0"
promptText="xKoordinate" fx:id="xCoordinateEntry" />
09          <TextField fx:id="yCoordinateEntry" layoutX="53.0"
layoutY="113.0" promptText="yKoordinate" />
10          <TextField fx:id="zCoordinateEntry" layoutX="53.0"
layoutY="154.0" promptText="zKoordinate" />
11          <Button layoutX="53.0" layoutY="194.0" mnemonicParsing="false"
onAction="#onAddCoordinatesClick" prefHeight="25.0" prefWidth="149.0"
text="Füge Koordinate hinzu" />
12          <Label fx:id="infoLabel" layoutX="275.0" layoutY="40.0"
prefHeight="46.0" prefWidth="202.0" />
13          <Button layoutX="53.0" layoutY="294.0" mnemonicParsing="false"
onAction="#onFlyToCoordinatesClick" prefHeight="69.0" prefWidth="109.0"
text="Fliege Weg ab" />
14           <ProgressBar fx:id="chargeLevelProgressBar" layoutX="570.0"
layoutY="21.0" nodeOrientation="LEFT_TO_RIGHT" prefHeight="18.0"
prefWidth="168.0" progress="0.0" />
15           <Label fx:id="chargeLevelLabel" text="Label" layoutX="570.0"
layoutY="40" prefHeight="28.0" prefWidth="68.0"/>
16          <TableView fx:id="tableWithCoordinates" layoutX="245.0"
layoutY="87.0" prefHeight="300.0" prefWidth="180.0">
17            <columns>
18              <TableColumn prefWidth="60.0" text="x"
fx:id="xCoordinateTableColumn" />
```

```
19              <TableColumn fx:id="yCoordinateTableColumn"
prefWidth="60.0" text="y" />
20                <TableColumn fx:id="zCoordinateTableColumn"
prefWidth="60.0" text="z" />
21             </columns>
22           </TableView>
23            <SplitPane dividerPositions="0.5" layoutX="469.0"
layoutY="221.0" prefHeight="28.0" prefWidth="292.0">
24                <items>
25                    <Label prefHeight="28.0" prefWidth="120.0" text="Pace
'X | Y | Z'" />
26                    <Label fx:id="anzeigeXVelocity" prefHeight="26.0"
prefWidth="97.0" />
27                    <Label fx:id="anzeigeYVelocity" prefHeight="26.0"
prefWidth="98.0" />
28                    <Label fx:id="anzeigeZVelocity" prefHeight="26.0"
prefWidth="97.0" />
29                </items>
30            </SplitPane>
31            <SplitPane dividerPositions="0.5" layoutX="469.0"
layoutY="259.0" prefHeight="28.0" prefWidth="292.0">
32                <items>
33                    <Label prefHeight="28.0" prefWidth="120.0"
text="xKoordinate" />
34                    <Label fx:id="anzeigeX" prefHeight="26.0"
prefWidth="186.0" />
35                </items>
36            </SplitPane>
37            <SplitPane dividerPositions="0.5" layoutX="469.0"
layoutY="296.0" prefHeight="28.0" prefWidth="292.0">
38                <items>
39                    <Label prefHeight="28.0" prefWidth="120.0"
text="yKoordinate" />
40                    <Label fx:id="anzeigeY" prefHeight="26.0"
prefWidth="188.0" />
41                </items>
42            </SplitPane>
43            <SplitPane dividerPositions="0.5" layoutX="469.0"
layoutY="334.0" prefHeight="28.0" prefWidth="292.0">
44                <items>
45                    <Label prefHeight="28.0" prefWidth="120.0"
text="zKoordinate" />
46                    <Label fx:id="anzeigeZ" prefHeight="26.0"
prefWidth="189.0" />
47                </items>
48            </SplitPane>
49            <SplitPane dividerPositions="0.5" layoutX="469.0"
layoutY="372.0" prefHeight="28.0" prefWidth="292.0">
50                <items>
51                    <Label prefHeight="26.0" prefWidth="120.0"
text="Abstand zum Boden" />
52                    <Label fx:id="abstandZumBoden" prefHeight="26.0"
prefWidth="189.0" />
53                </items>
54            </SplitPane>
55           <Button layoutX="53.0" layoutY="238.0" mnemonicParsing="false"
onAction="#deleteLastCoordinate" prefHeight="25.0" prefWidth="149.0"
text="Lösche letzte Koordinate" />
56       </children>
57         <Slider fx:id="chargingSlider" layoutX="568.0" layoutY="73.0"
prefHeight="14.0" prefWidth="168.0"/>
58         <Button layoutX="568.0" layoutY="99.0" mnemonicParsing="false"
```

```
prefHeight="41.0" prefWidth="86.0" text="Charge" onAction="#chargeDrone"/>
59    </Pane>
```

10.1.6 Unity Code

```
1 // using is import in Java
2 using System;
3 using System.Collections;
4 using System.Collections.Generic;
5 using System.Net;
6 using System.Net.Sockets;
7 using System.Text;
8 using System.Threading;
9 using UnityEngine;
10
11
12 // : is extends in Java
13 public class DroneServer : MonoBehaviour
14 {
15     // const is final in JAVA
16     private const string HOST_NAME = "127.0.0.1";
17     private const int PORT = 10666;
18     private const int BUFFER_SIZE = 1024;
19     private const string LISTENER_INFO = "Server wartet auf Client-
Anfrage...";
20     private const string REQUEST_INFO = "Folgende Nachricht vom Client
empfangen: ";
21     private const string STANDARD_DATA = "Ping vom Server";
22     private string clientMessage1;
23     private double lidarSensor;
24     //private float xKoordinate;
25     //private float yKoordinate;
26     //private float zKoordinate;
27     private Vector3 position;
28      private Vector3 velocity;
29     private RaycastHit raycast;
30
31     // A SerializeField can be seen in Unity-Editor.
32     [SerializeField] private GameObject drone;
33
34     // TCPListener to listen for incomming TCP connection requests.
```

```
35      // https://learn.microsoft.com/de-
de/dotnet/api/system.net.sockets.tcplistener?view=net-7.0
36      private TcpListener tcpListener;
37
38      // Background thread for server workload.
39      // https://learn.microsoft.com/de-
de/dotnet/api/system.threading.thread?view=net-7.0
40      private Thread tcpListenerThread;
41
42      // Handle to connected TCP client.
43      // https://learn.microsoft.com/de-
de/dotnet/api/system.net.sockets.tcpclient?view=net-7.0
44      private TcpClient connectedTcpClient;
45
46
47
48      // Start is called on the frame when a script is enabled just before
any of the Update methods are called the first time.
49      // Since Start is only called once, it should be used to initialize
things that need to persist
50      // throughout the life of the script but should only need to be
setup immediately before use.
51      void Start()
52      {
53          // Start background thread
54          this.tcpListenerThread = new Thread(new
ThreadStart(ListenForIncommingRequests));
55          this.tcpListenerThread.IsBackground = true;
56          this.tcpListenerThread.Start();
57      }
58
59
60      // Update is called every frame, if the MonoBehaviour is enabled.
61      void Update()
62      {
63          //yKoordinate = drone.transform.position.y; // Besser in
FixedUpdate() --> Framerateunabh�ngig, aufgerufen durch Physik-Engine
64
65          //Debug.Log(yKoordinate);
66          // TODO handle shutdown via Escape key
67          if (Input.GetKeyDown(KeyCode.Escape))
68          {
69              ;
70          }
```

```
71
72          if (clientMessage1.Contains("W"))
73          {
74              drone.GetComponent<DroneMovement>().W = true;
75          }
76          if (clientMessage1.Contains("A"))
77          {
78              drone.GetComponent<DroneMovement>().A = true;
79          }
80          if (clientMessage1.Contains("S"))
81          {
82              drone.GetComponent<DroneMovement>().S = true;
83          }
84          if (clientMessage1.Contains("X"))
85          {
86              drone.GetComponent<DroneMovement>().D = true;
87          }
88          if (clientMessage1.Contains("Up"))
89          {
90              drone.GetComponent<DroneMovement>().I = true;
91          }
92          if (clientMessage1.Contains("Down"))
93          {
94              drone.GetComponent<DroneMovement>().K = true;
95          }
96          if (clientMessage1.Contains("TurnLeft"))
97          {
98              drone.GetComponent<DroneMovement>().J = true;
99          }
100          if (clientMessage1.Contains("TurnRight"))
101          {
102              drone.GetComponent<DroneMovement>().L = true;
103          }
104
105      }
106
107      void FixedUpdate()
108      {
109          //yKoordinate = drone.transform.position.y;
110          //xKoordinate = drone.transform.position.x;
111          //zKoordinate = drone.transform.position.z;
```

```
112         position = drone.transform.position;
113         velocity = drone.GetComponent<Rigidbody>().velocity;
114         if (Physics.Raycast(drone.transform.position, -Vector3.up, out
this.raycast))
115                 {
116                     // Calculate the distance from the drone to the
ground
117                     this.lidarSensor = raycast.distance -1;
118                 }
119     }
120
121
122     // https://learn.microsoft.com/de-
de/dotnet/api/system.net.sockets.tcplistener?view=net-7.0
123     // https://learn.microsoft.com/de-
de/dotnet/api/system.net.sockets.tcpclient?view=net-7.0
124     private void ListenForIncommingRequests()
125     {
126         try
127         {
128             this.tcpListener = new
TcpListener(IPAddress.Parse(HOST_NAME),PORT);
129             this.tcpListener.Start();
130             Debug.Log(LISTENER_INFO);
131
132             Byte[] bytes = new Byte[BUFFER_SIZE];
133             while (true)
134             {
135                 using (this.connectedTcpClient =
tcpListener.AcceptTcpClient())
136                 {
137                     using (NetworkStream stream =
this.connectedTcpClient.GetStream())
138                     {
139                         int length;
140                         while ((length = stream.Read(bytes, 0,
bytes.Length)) != 0)
141                         {
142                             var incommingData = new byte[length];
143                             Array.Copy(bytes, 0, incommingData, 0,
length);
144                             string clientMessage =
Encoding.ASCII.GetString(incommingData);
145                             Debug.Log(REQUEST_INFO + clientMessage);
146                             clientMessage1 = clientMessage;
```

```
147                             SendMessage();
148                         }
149                     }
150                 }
151             }
152         }
153         catch (SocketException socketException)
154         {
155             Debug.Log(socketException.ToString());
156         }
157     }
158 
159 
160     // https://learn.microsoft.com/de-
de/dotnet/api/system.net.sockets.tcpclient?view=net-7.0
161     private void SendMessage()
162     {
163         if (this.connectedTcpClient != null)
164         {
165             try
166             {
167                 NetworkStream stream =
this.connectedTcpClient.GetStream();
168                 if (stream.CanWrite)
169                 {
170                     byte[] serverMessageAsByteArray =
Encoding.ASCII.GetBytes(GetData());
171                     stream.Write(serverMessageAsByteArray, 0,
serverMessageAsByteArray.Length);
172                     Debug.Log(GetData());
173                 }
174             }
175             catch (SocketException socketException)
176             {
177                 Debug.Log(socketException);
178             }
179         }
180 
181 
182     }
183 
184 
```

```
185     private string GetData ()
186     {
187         // TODO how to simulate data from sensors, actors etc.?
188         // https://docs.unity3d.com/ScriptReference/GameObject.html
189         //
https://docs.unity3d.com/ScriptReference/Vector3.Distance.html
190         // https://learn.unity.com/tutorial/let-s-try-shooting-with-
raycasts#
191
192         // Drohne bewegen
193         // M�glichkeit 1: Kraft auf die Drohne aus�ben:
drone.GetComponent<Rigidbody>().AddForce(Vector3.forward * 30f);
194         // M�glichkeit 2: Tastendruck simulieren:
drone.GetComponent<DroneMovement>().W = true;
195         // (Funktioniert hier beides nicht, kann nur aus dem Main-
Thread aufgerufen werden)
196
197         // Aktuelle Flugh�he an Client senden
198         //return xKoordinate.ToString() + ";" + yKoordinate.ToString()
+ ";" + zKoordinate.ToString() + "\n"; // \n is needed for Java line read
199         DroneData droneData = new DroneData();
200         droneData.position = position;
201         droneData.velocity = velocity;
202         droneData.lidarSensor = this.lidarSensor;
203
204
205         string jsonString = JsonUtility.ToJson(droneData);
206         Debug.Log(jsonString);
207         return jsonString + "\n";
208     }
209
210
211 }
212
213 public class DroneData
214     {
215         public Vector3 position;
216         public Vector3 velocity;
217         public double lidarSensor;
218     }
219
```

10.2 JavaDoc Dokumentation

Contents

Class Hierarchy

Classes

- java.lang.Object
    - Application
        - org.example.view.MainFX (in 4.2, page 34)
    - org.example.control.DroneControllClient (in 1.1, page 6)
    - org.example.control.DroneControllServer (in 1.4, page 10)
    - org.example.control.DroneController (in 1.2, page 7)
    - org.example.control.DroneControllerMain (in 1.3, page 9)
    - org.example.control.PlannedFlight (in 1.5, page 11)
    - org.example.control.PlanningSoftwareClient (in 1.6, page 12)
    - org.example.model.Actuator (in 2.1, page 14)
        - org.example.model.Battery (in 2.2, page 15)
    - org.example.model.DataPackage (in 2.3, page 16)
    - org.example.model.Drone (in 2.4, page 18)
    - org.example.model.Position (in 2.6, page 20)
    - org.example.model.Sensor (in 2.7, page 22)
        - org.example.model.LidarSensor (in 2.5, page 20)
    - org.example.model.Velocity (in 2.8, page 22)
    - org.example.view.FreeFlightDelegate (in 4.1, page 30)
    - org.example.view.MainScreenDelegate (in 4.3, page 35)
    - org.example.view.Planning (in 4.4, page 36)

Interfaces

- org.example.res.Constants (in 3.1, page 26)
- org.example.res.Strings (in 3.2, page 27)

Exceptions

- java.lang.Object
- java.lang.Throwable
- java.lang.Exception
- org.example.model.ConnectionLostException (in 2.9, page 24)

Chapter 1

Package org.example.control

Package Contents *Page*

Classes

1.1 Class DroneControllClient

This class is the client of the DroneController setting up a connection to Unity.

1.1.1 Declaration

```
public class DroneControllClient
 extends java.lang.Object implements java.lang.Runnable
```

1.1.2 Constructor summary

DroneControllClient()

1.1.3 Method summary

getDrone()
run() Runs the communication loop between the controller and the Unity simulation.This method establishes a socket connection to the Unity simulation at the specified host and port.

1.1.4 Constructors

- **DroneControllClient**

```
public DroneControllClient()
```

1.1.5 Methods

- **getDrone**

```
public org.example.model.Drone getDrone()
```

- **run**

```
public void run()
```

  - **Description**
    Runs the communication loop between the controller and the Unity simulation.This method establishes a socket connection to the Unity simulation at the specified host and port. It continuously sends movement commands to the simulation and receives data packages containing drone position and lidar sensor information. The loop runs indefinitely until the program is terminated.

1.2 Class DroneController

This class controls the drone and its attributes. The drone instance is created and managed here.

1.2.1 Declaration

```
public class DroneController
 extends java.lang.Object
```

1.2.2 Method summary

1.2.3 Methods

- **getChargeLevelProperty**

```
public DoubleProperty getChargeLevelProperty()
```

- **getClassInstance**

```
public static DroneController getClassInstance()
```

  - **Description**
    Returns the singleton instance of the DroneController class.
  - **Returns** – DroneController singleton instance that is not null.

- **getDistanceToGroundProperty**

```
public DoubleProperty getDistanceToGroundProperty()
```

- **getDrone**

```
public org.example.model.Drone getDrone()
```

- **getXKoordinateProperty**

```
public DoubleProperty getXKoordinateProperty()
```

- **getxVelocityProperty**

```
public synchronized DoubleProperty getxVelocityProperty()
```

- **getYKoordinateProperty**

  ```
  public DoubleProperty getYKoordinateProperty()
  ```

- **getyVelocityProperty**

  ```
  public DoubleProperty getyVelocityProperty()
  ```

- **getZKoordinateProperty**

  ```
  public DoubleProperty getZKoordinateProperty()
  ```

- **getzVelocityProperty**

  ```
  public DoubleProperty getzVelocityProperty()
  ```

- **yVelocityPropertyProperty**

  ```
  public DoubleProperty yVelocityPropertyProperty()
  ```

- **zVelocityPropertyProperty**

  ```
  public DoubleProperty zVelocityPropertyProperty()
  ```

1.3 Class DroneControllerMain

This class starts threads for the DroneController.

1.3.1 Declaration

```
public class DroneControllerMain
 extends java.lang.Object
```

1.3.2 Constructor summary

DroneControllerMain()

1.3.3 Method summary

main(String[]) This method is called when the programm is executed.

1.3.4 Constructors

- **DroneControllerMain**

```
public DroneControllerMain()
```

1.3.5 Methods

- **main**

```
public static void main(java.lang.String[] args)
```

  - **Description**
    This method is called when the programm is executed. It starts every Thread needed for the connections.
  - **Parameters**
    - `args` – The command-line arguments passed to the program.

1.4 Class DroneControllServer

This class represents a server for handling request of the planning software and sends data of a drone it gets from Unity.

1.4.1 Declaration

```
public class DroneControllServer
 extends java.lang.Object implements java.lang.Runnable
```

1.4.2 Constructor summary

DroneControllServer(Drone) This constructor sets the drone to the given drone.

1.4.3 Method summary

run() This method runs the Thread for the server of the droneController wich sends data to the planning software if it gets a request.

1.4.4 Constructors

- **DroneControllServer**

```
public DroneControllServer(org.example.model.Drone drone)
```

- **Description**
  This constructor sets the drone to the given drone.
- **Parameters**
  - `drone` – Drone object that is associated with the server.

1.4.5 Methods

- **run**

```
public void run()
```

  - **Description**
    This method runs the Thread for the server of the droneController wich sends data to the planning software if it gets a request.
  - **Throws**
    - `java.lang.RuntimeException` – if an I/O error occurs during the server operation.

1.5 Class PlannedFlight

This class plans a flight to every coordinate the user puts in the list of the GUI.

1.5.1 Declaration

```
public class PlannedFlight
 extends java.lang.Object implements java.lang.Runnable
```

1.5.2 Constructor summary

PlannedFlight() This constructor sets an object of DroneController to the DroneController instance and returns it.

1.5.3 Method summary

run() This method runs the thread that lets the drone fly to every coordinate in the table of the GUI.

1.5.4 Constructors

- **PlannedFlight**

```
public PlannedFlight()
```

  - **Description**
    This constructor sets an object of DroneController to the DroneController instance and returns it.

1.5.5 Methods

- **run**

```
public void run()
```

  - **Description**
    This method runs the thread that lets the drone fly to every coordinate in the table of the GUI.
  - **Throws**
    - `java.lang.RuntimeException` – if an InterruptedException occurs during the thread sleep.

1.6 Class PlanningSoftwareClient

This class

1.6.1 Declaration

```
public class PlanningSoftwareClient
 extends java.lang.Object implements java.lang.Runnable
```

1.6.2 Constructor summary

PlanningSoftwareClient()

1.6.3 Method summary

run() Runs the continuous loop to communicate with the drone controller server and update the drone's state.

1.6.4 Constructors

- **PlanningSoftwareClient**

```
public PlanningSoftwareClient()
```

1.6.5 Methods

- **run**

```
public synchronized void run()
```

– **Description**

Runs the continuous loop to communicate with the drone controller server and update the drone's state.

Chapter 2

Package org.example.model

Package Contents *Page*

Classes

2.1 Class Actuator

Tis is the abstract class for Actuators the drone has.

2.1.1 Declaration

```
public abstract class Actuator
 extends java.lang.Object
```

2.1.2 All known subclasses

Battery (in 2.2, page 15)

2.1.3 Constructor summary

Actuator()

2.1.4 Method summary

calculateActuatorData() The abstract class for the calculation of data of Actuators the drone has.

2.1.5 Constructors

- **Actuator**

```
public Actuator()
```

2.1.6 Methods

- **calculateActuatorData**

```
public abstract void calculateActuatorData()
```

  - **Description**
    The abstract class for the calculation of data of Actuators the drone has.

2.2 Class Battery

This class represents a battery of a drone.

2.2.1 Declaration

```
public class Battery
 extends org.example.model.Actuator
```

2.2.2 Constructor summary

Battery(double) Creates a battery with a chargeLevel in percent.

2.2.3 Method summary

calculateActuatorData() This method lowers the chargeLevel at every method call.
getChargeLevel()
setChargeLevel(double)

2.2.4 Constructors

- **Battery**

```
public Battery(double chargeLevel)
```

  - **Description**
    Creates a battery with a chargeLevel in percent.
  - **Parameters**
    - `chargeLevel` – The initial charge level of the drone's battery.

2.2.5 Methods

- **calculateActuatorData**

```
public void calculateActuatorData()
```

  - **Description**
    This method lowers the chargeLevel at every method call.

- **getChargeLevel**

```
public double getChargeLevel()
```

- **setChargeLevel**

```
public void setChargeLevel(double chargeLevel)
```

2.2.6 Members inherited from class Actuator

`org.example.model.Actuator` (in 2.1, page 14)

- `public abstract void` **`calculateActuatorData()`**

2.3 Class DataPackage

This class contains every data the program gets from unity.

2.3.1 Declaration

```
public class DataPackage
 extends java.lang.Object
```

2.3.2 Constructor summary

DataPackage(Position, double, Velocity) This constructor initializes the DataPackage object with the given position and lidar sensor data.

2.3.3 Method summary

getLidarSensor()
getPosition()
getVelocity()
setLidarSensor(double)
setPosition(Position)
setVelocity(Velocity)

2.3.4 Constructors

- **DataPackage**

```
public DataPackage(Position position,double lidarSensor,Velocity
    velocity)
```

  - **Description**
    This constructor initializes the DataPackage object with the given position and lidar sensor data.
  - **Parameters**
    - `position` – The position data to be included in the DataPackage.
    - `lidarSensor` – The lidar sensor data to be included in the DataPackage.

2.3.5 Methods

- **getLidarSensor**

```
public double getLidarSensor()
```

- **getPosition**

```
public Position getPosition()
```

- **getVelocity**

```
public Velocity getVelocity()
```

- **setLidarSensor**

```
public void setLidarSensor(double lidarSensor)
```

- **setPosition**

```
public void setPosition(Position position)
```

- **setVelocity**

```
public void setVelocity(Velocity velocity)
```

2.4 Class Drone

THis class implements a drone with all its components.

2.4.1 Declaration

```
public class Drone
 extends java.lang.Object
```

2.4.2 Constructor summary

Drone(Position) This constructor creates a new Drone with the specified position and a Battery with chargeLevel one.

2.4.3 Method summary

getBattery()
getLidarSensor()
getMovement()
getPosition()
getVelocity()
setBattery(Battery)
setLidarSensor(LidarSensor)
setMovement(String)
setPosition(Position)
setVelocity(Velocity)

2.4.4 Constructors

- **Drone**

```
public Drone(Position position)
```

  - **Description**
    This constructor creates a new Drone with the specified position and a Battery with chargeLevel one.

- **Parameters**
  - `position` – The initial position of the drone.

2.4.5 Methods

- **getBattery**

```
public Battery getBattery()
```

- **getLidarSensor**

```
public LidarSensor getLidarSensor()
```

- **getMovement**

```
public java.lang.String getMovement()
```

- **getPosition**

```
public Position getPosition()
```

- **getVelocity**

```
public Velocity getVelocity()
```

- **setBattery**

```
public void setBattery(Battery battery)
```

- **setLidarSensor**

```
public void setLidarSensor(LidarSensor lidarSensor)
```

- **setMovement**

```
public void setMovement(java.lang.String movement)
```

- **setPosition**

```
public void setPosition(Position position)
```

- **setVelocity**

```
public void setVelocity(Velocity velocity)
```

2.5 Class LidarSensor

This class represents a lidarSensor of the drone measuring the distance from the drone to the ground.

2.5.1 Declaration

```
public class LidarSensor
 extends org.example.model.Sensor
```

2.5.2 Constructor summary

LidarSensor(double)

2.5.3 Method summary

getDistanceToGround()
setDistanceToGround(double)

2.5.4 Constructors

- **LidarSensor**

  ```
  public LidarSensor(double distanceToGround)
  ```

  - **Parameters**
    - `distanceToGround` – The distance to the ground measured by the Lidar sensor.

2.5.5 Methods

- **getDistanceToGround**

  ```
  public double getDistanceToGround()
  ```

- **setDistanceToGround**

  ```
  public void setDistanceToGround(double distanceToGround)
  ```

2.6 Class Position

This class keeps a position with its attributes.

2.6.1 Declaration

```
public class Position
 extends java.lang.Object
```

2.6.2 Constructor summary

Position(double, double, double) Creates a position object in a three-dimensional map.

2.6.3 Method summary

getX()
getY()
getZ()
setX(double)
setY(double)
setZ(double)
toString()

2.6.4 Constructors

- **Position**

  ```
  public Position(double x,double y,double z)
  ```

  - **Description**
    Creates a position object in a three-dimensional map.
  - **Parameters**
    - `x` – The X coordinate.
    - `y` – The Y coordinate.
    - `z` – The Z coordinate.

2.6.5 Methods

- **getX**

  ```
  public double getX()
  ```

- **getY**

  ```
  public double getY()
  ```

- **getZ**

```
public double getZ()
```

- **setX**

```
public void setX(double x)
```

- **setY**

```
public void setY(double y)
```

- **setZ**

```
public void setZ(double z)
```

- **toString**

```
public java.lang.String toString()
```

2.7 Class Sensor

2.7.1 Declaration

```
public abstract class Sensor
 extends java.lang.Object
```

2.7.2 All known subclasses

LidarSensor (in 2.5, page 20)

2.7.3 Constructor summary

Sensor()

2.7.4 Constructors

- **Sensor**

```
public Sensor()
```

2.8 Class Velocity

This class keeps a velocity with its attributes.

2.8.1 Declaration

```
public class Velocity
 extends java.lang.Object
```

2.8.2 Constructor summary

Velocity(double, double, double) Constructs a new Velocity object with the specified velocities.

2.8.3 Method summary

getxVelocity()
getyVelocity()
getzVelocity()
setxVelocity(double)
setyVelocity(double)
setzVelocity(double)
toString()

2.8.4 Constructors

- **Velocity**

  ```
  public Velocity(double xVelocity,double yVelocity,double
      zVelocity)
  ```

  - **Description**
    Constructs a new Velocity object with the specified velocities.
  - **Parameters**
    - `xVelocity` – the x-component of the velocity
    - `yVelocity` – the y-component of the velocity
    - `zVelocity` – the z-component of the velocity

2.8.5 Methods

- **getxVelocity**

  ```
  public double getxVelocity()
  ```

- **getyVelocity**

  ```
  public double getyVelocity()
  ```

- **getzVelocity**

```
public double getzVelocity()
```

- **setxVelocity**

```
public void setxVelocity(double xVelocity)
```

- **setyVelocity**

```
public void setyVelocity(double yVelocity)
```

- **setzVelocity**

```
public void setzVelocity(double zVelocity)
```

- **toString**

```
public java.lang.String toString()
```

2.9 Exception ConnectionLostException

2.9.1 Declaration

```
public class ConnectionLostException
 extends java.lang.Exception
```

2.9.2 Constructor summary

ConnectionLostException(String)

2.9.3 Constructors

- **ConnectionLostException**

```
public ConnectionLostException(java.lang.String s)
```

2.9.4 Members inherited from class Throwable

`java.lang.Throwable`

- `public final synchronized void` **addSuppressed**`(Throwable` **arg0**`)`
- `public synchronized Throwable` **fillInStackTrace**`()`
- `public synchronized Throwable` **getCause**`()`
- `public String` **getLocalizedMessage**`()`
- `public String` **getMessage**`()`
- `public StackTraceElement` **getStackTrace**`()`
- `public final synchronized Throwable` **getSuppressed**`()`
- `public synchronized Throwable` **initCause**`(Throwable` **arg0**`)`
- `public void` **printStackTrace**`()`
- `public void` **printStackTrace**`(java.io.PrintStream` **arg0**`)`
- `public void` **printStackTrace**`(java.io.PrintWriter` **arg0**`)`
- `public void` **setStackTrace**`(StackTraceElement[]` **arg0**`)`
- `public String` **toString**`()`

Chapter 3

Package org.example.res

Package Contents *Page*

Interfaces

3.1 Interface Constants

This class contains every necessary constant.

3.1.1 Declaration

```
public interface Constants
```

3.1.2 Field summary

BRAKING_DISTANCE_X
BRAKING_DISTANCE_Y
BRAKING_DISTANCE_Z
CHARGE_LEVEL_CONSTANT
GET_DATA_SLEEP
MAX_HIGHT_OF_MOUNTAINS_PLUS_TWENTY
MAX_X_COODINATE
MAX_Z_COODINATE
MIN_X_COODINATE
MIN_Y_COODINATE
MIN_Z_COODINATE
NUMBER_TO_PERCENT
PORT_CONTROLLER_PLANNING
PORT_CONTROLLER_UNITY

WAIT_AT_DESTINATION_SLEEP
WINDOW_HIGHT
WINDOW_WIDTH

3.1.3 Fields

- `long` **GET_DATA_SLEEP**
- `int` **PORT_CONTROLLER_PLANNING**
- `int` **WINDOW_WIDTH**
- `int` **WINDOW_HIGHT**
- `double` **NUMBER_TO_PERCENT**
- `long` **WAIT_AT_DESTINATION_SLEEP**
- `double` **MAX_HIGHT_OF_MOUNTAINS_PLUS_TWENTY**
- `double` **BRAKING_DISTANCE_Y**
- `double` **BRAKING_DISTANCE_X**
- `double` **BRAKING_DISTANCE_Z**
- `int` **PORT_CONTROLLER_UNITY**
- `double` **MIN_Z_COODINATE**
- `double` **MIN_X_COODINATE**
- `double` **MAX_Z_COODINATE**
- `double` **MAX_X_COODINATE**
- `double` **MIN_Y_COODINATE**
- `double` **CHARGE_LEVEL_CONSTANT**

3.2 Interface Strings

This class contains every necessary string used in the program.

3.2.1 Declaration

```
public interface Strings
```

3.2.2 Field summary

3.2.3 Fields

- `java.lang.String` **ERROR_IP_ADRESSE**
- `java.lang.String` **IO_ERROR_DURING_CONNECTION_BUILD**
- `java.lang.String` **CONNECTION_DENIED**
- `java.lang.String` **ILLEGAL_PORT**
- `java.lang.String` **LOCALHOST**
- `java.lang.String` **SERVER_IS_RUNNING**
- `java.lang.String` **NOT_MOVING**
- `java.lang.String` **W**

- `java.lang.String` **A**
- `java.lang.String` **S**
- `java.lang.String` **D**
- `java.lang.String` **UP**
- `java.lang.String` **DOWN**
- `java.lang.String` **TURN_LEFT**
- `java.lang.String` **TURN_RIGHT**
- `java.lang.String` **TITLE_DRONE_SOFTWARE**
- `java.lang.String` **PATH_TO_MAIN_SCREEN_FXML**
- `java.lang.String` **LITTLE_X**
- `java.lang.String` **LITTLE_Y**
- `java.lang.String` **LITTLE_Z**
- `java.lang.String` **POSITION_ADDED**
- `java.lang.String` **WRONG_INPUT_COORDINATES**
- `java.lang.String` **X_COLON**
- `java.lang.String` **Y_COLON**
- `java.lang.String` **Z_COLON**
- `java.lang.String` **POSITION_REMOVED**
- `java.lang.String` **NO_COORDINATES_TO_REMOVE**
- `java.lang.String` **CONNECTION_LOST_TO_UNITY**
- `java.lang.String` **OUT_OF_BORDER**

Chapter 4

Package org.example.view

Package Contents *Page*

Classes

4.1 Class FreeFlightDelegate

This class is the connection between the planned flight and the GUI.

4.1.1 Declaration

```
public class FreeFlightDelegate
 extends java.lang.Object
```

4.1.2 Field summary

abstandZumBoden
anzeigeX
anzeigeXVelocity
anzeigeY
anzeigeYVelocity
anzeigeZ
anzeigeZVelocity
chargeLevelLabel

chargeLevelProgressBar
chargingSlider

4.1.3 Constructor summary

FreeFlightDelegate()

4.1.4 Method summary

chargeDrone() If the chargeFrone button is clicked the chargeLevel will set to the sliders Value.
getCommandToController()
onSendAClick() If the A-Button is clicked.
onSendDClick() If the D-Button is clicked.
onSendDownClick() If the "down" button is clicked.
onSendSClick() If the S-Button is clicked.
onSendStopClick() If the "notMoving" button is clicked.
onSendTurnLeftClick() If the "turnLeft" button is clicked.
onSendTurnRightClick() If the "turnRight" button is clicked.
onSendUpClick() If the "up" button is clicked.
onSendWClick() If the W-Button is clicked.
setCommandToController(String)

4.1.5 Fields

- `public Label` **anzeigeX**
- `public Label` **anzeigeY**
- `public Label` **anzeigeZ**
- `public Label` **anzeigeXVelocity**
- `public ProgressBar` **chargeLevelProgressBar**
- `public Label` **chargeLevelLabel**
- `public Slider` **chargingSlider**
- `public Label` **abstandZumBoden**
- `public Label` **anzeigeYVelocity**
- `public Label` **anzeigeZVelocity**

4.1.6 Constructors

- **FreeFlightDelegate**

```
public FreeFlightDelegate()
```

4.1.7 Methods

- **chargeDrone**

```
public void chargeDrone()
```

  - **Description**
    If the chargeFrone button is clicked the chargeLevel will set to the sliders Value.

- **getCommandToController**

```
public static java.lang.String getCommandToController()
```

- **onSendAClick**

```
public void onSendAClick()
```

  - **Description**
    If the A-Button is clicked. The commandToController value will be set to "A".

- **onSendDClick**

```
public void onSendDClick()
```

  - **Description**
    If the D-Button is clicked. The commandToController value will be set to "D".

- **onSendDownClick**

```
public void onSendDownClick()
```

  - **Description**
    If the "down" button is clicked. The commandToController value will be set to "down".

- **onSendSClick**

```
public void onSendSClick()
```

  - **Description**
    If the S-Button is clicked. The commandToController value will be set to "S".

- **onSendStopClick**

  ```
  public void onSendStopClick()
  ```

  - **Description**
    If the "notMoving" button is clicked. The commandToController value will be set to "notMoving".

- **onSendTurnLeftClick**

  ```
  public void onSendTurnLeftClick()
  ```

  - **Description**
    If the "turnLeft" button is clicked. The commandToController value will be set to "turnLeft".

- **onSendTurnRightClick**

  ```
  public void onSendTurnRightClick()
  ```

  - **Description**
    If the "turnRight" button is clicked. The commandToController value will be set to "turnRight".

- **onSendUpClick**

  ```
  public void onSendUpClick()
  ```

  - **Description**
    If the "up" button is clicked. The commandToController value will be set to "up".

- **onSendWClick**

  ```
  public void onSendWClick()
  ```

  - **Description**
    If the W-Button is clicked. The commandToController value will be set to "W".

- **setCommandToController**

  ```
  public static void setCommandToController(java.lang.String
      command)
  ```

4.2 Class MainFX

This class starts every party of the program.

4.2.1 Declaration

```
public class MainFX
 extends Application
```

4.2.2 Constructor summary

MainFX()

4.2.3 Method summary

main(String[]) This method is called when the programm is started.
start(Stage) Starts the application by loading the main screen FXML file, creating a scene, and displaying it on the stage.

4.2.4 Constructors

- **MainFX**

```
public MainFX()
```

4.2.5 Methods

- **main**

```
public static void main(java.lang.String[] args)
```

  - **Description**
    This method is called when the programm is started. calls the runProgram-method.
  - **Parameters**
    - `args` – Program arguments.

- **start**

```
public void start(Stage stage) throws java.lang.Exception
```

  - **Description**
    Starts the application by loading the main screen FXML file, creating a scene, and displaying it on the stage.
  - **Parameters**

* `stage` – The primary stage of the JavaFX application.

– **Throws**

* `java.lang.Exception` – If an exception occurs during the start of the application.

4.3 Class MainScreenDelegate

This class starts every party of the program.

4.3.1 Declaration

```
public class MainScreenDelegate
 extends java.lang.Object
```

4.3.2 Constructor summary

MainScreenDelegate()

4.3.3 Method summary

onAboutClick()
stopProgramm() Stops the execution of the program.

4.3.4 Constructors

- **MainScreenDelegate**

```
public MainScreenDelegate()
```

4.3.5 Methods

- **onAboutClick**

```
public void onAboutClick() throws java.io.IOException
```

- **stopProgramm**

```
public void stopProgramm()
```

– **Description**
Stops the execution of the program.

4.4 Class Planning

This class sets everything up that is necessary for displaying data for the planned flight.

4.4.1 Declaration

```
public class Planning
 extends java.lang.Object
```

4.4.2 Field summary

abstandZumBoden
anzeigeX
anzeigeXVelocity
anzeigeY
anzeigeYVelocity
anzeigeZ
anzeigeZVelocity
chargeLevelLabel
chargeLevelProgressBar
chargingSlider
infoLabel
tableWithCoordinates
xCoordinateEntry
xCoordinateTableColumn
yCoordinateEntry
yCoordinateTableColumn
zCoordinateEntry
zCoordinateTableColumn

4.4.3 Constructor summary

Planning()

4.4.4 Method summary

chargeDrone() Charges the drone's battery based on the value of the charging slider.
deleteLastCoordinate() Deletes the last coordinate from the list of positions.
getPositions()
initialize() This Method is called when the FXML-File "planning.fxml" is loaded.
onAddCoordinatesClick() Handles the event when the "Add Coordinates" button is clicked.
onFlyToCoordinatesClick() Handles the event when the "Fly to Coordinates" button is clicked.

setBindings(Label, Label, Label, Label, Label, ProgressBar, Label, Label, Label) Sets up bidirectional bindings between the provided labels, progress bar, and the properties in the DroneController.

4.4.5 Fields

- `public TextField` **xCoordinateEntry**
- `public TextField` **yCoordinateEntry**
- `public TextField` **zCoordinateEntry**
- `public` **tableWithCoordinates**
- `public Label` **infoLabel**
- `public` **xCoordinateTableColumn**
- `public` **yCoordinateTableColumn**
- `public` **zCoordinateTableColumn**
- `public ProgressBar` **chargeLevelProgressBar**
- `public Label` **chargeLevelLabel**
- `public Slider` **chargingSlider**
- `public Label` **anzeigeXVelocity**
- `public Label` **anzeigeX**
- `public Label` **anzeigeY**
- `public Label` **anzeigeZ**
- `public Label` **abstandZumBoden**
- `public Label` **anzeigeYVelocity**
- `public Label` **anzeigeZVelocity**

4.4.6 Constructors

- **Planning**

```
public Planning()
```

4.4.7 Methods

- **chargeDrone**

  ```
  public void chargeDrone()
  ```

  - **Description**
    Charges the drone's battery based on the value of the charging slider.

- **deleteLastCoordinate**

  ```
  public void deleteLastCoordinate()
  ```

  - **Description**
    Deletes the last coordinate from the list of positions.

- **getPositions**

  ```
  public static <any> getPositions()
  ```

- **initialize**

  ```
  public void initialize()
  ```

  - **Description**
    This Method is called when the FXML-File "planning.fxml" is loaded. It sets the ValueFactories for the table columns and binds the Charge-Level-ProgressBar to it´s Property as well as the ChargeLevel-Label.

- **onAddCoordinatesClick**

  ```
  public void onAddCoordinatesClick()
  ```

  - **Description**
    Handles the event when the "Add Coordinates" button is clicked. Updates the table view and info label.

- **onFlyToCoordinatesClick**

  ```
  public void onFlyToCoordinatesClick()
  ```

  - **Description**
    Handles the event when the "Fly to Coordinates" button is clicked. Creates a new instance of PlannedFlight and starts it in a separate thread.

- **setBindings**

  ```
  public static void setBindings(Label anzeigeX, Label anzeigeY,
      Label anzeigeZ, Label anzeigeXVelocity, Label abstandZumBoden,
      ProgressBar chargeLevelProgressBar, Label chargeLevelLabel,
      Label anzeigeYVelocity, Label anzeigeZVelocity)
  ```

  - **Description**

    Sets up bidirectional bindings between the provided labels, progress bar, and the properties in the DroneController.

  - **Parameters**

    * `anzeigeX` – The label displaying the X coordinate.
    * `anzeigeY` – The label displaying the Y coordinate.
    * `anzeigeZ` – The label displaying the Z coordinate.
    * `anzeigeXVelocity` – The label displaying the velocity.
    * `abstandZumBoden` – The label displaying the distance to the ground.
    * `chargeLevelProgressBar` – The progress bar displaying the battery charge level.
    * `chargeLevelLabel` – The label displaying the battery charge level.