

RaHM-Lab
Positionsregelung Drohne
Projekt Software Engineering 2

für die Prüfung zum
Bachelor of Science

im Studiengang Informatik
an der DHBW Karlsruhe

von

Michael Maag
Theresa Uhlmann

17.05.2022

Gutachter der DHBW Karlsruhe

Daniel Lindner

Inhaltsverzeichnis

Abbildungsverzeichnis	I
Abkürzungsverzeichnis	II
Tabellenverzeichnis	III
Code-Verzeichnis	IV
1 Einleitung	1
2 Problemstellung.....	2
3 Architektur	3
3.1 Dependency Rule.....	3
3.2 Domain Layer.....	4
3.3 Application Layer	5
3.4 Adapter Layer	5
3.5 PlugIn Layer	5
4 Entwurfsmuster	7
4.0.1 Fassade	7
4.0.2 Brücke	7
4.0.3 Beobachter	8
4.0.4 Eventbus.....	9
5 Domain Driven Design	10
5.1 Domänenmodell.....	10
5.2 Ubiquitous Language	10
5.3 Repositories	12
5.4 Aggregates	12
5.5 Entities.....	12
5.6 Value Objects.....	13
6 Programming Principles	14
6.1 SOLID.....	14
6.1.1 Single Responsibility Principle	14
6.1.2 Open/Closed Principle	14
6.1.3 Liskov Substitution Principle	14
6.1.4 Interface Segregation Principle	14
6.1.5 Dependency Inversion Principle.....	15
6.2 GRASP	15
6.2.1 Kopplung	15
6.2.2 Kohäsion.....	15

6.3	DRY	15
6.4	KISS	16
6.5	YAGNI	16
6.6	Conway's Lay	16
7	Software Tests	17
7.1	Unit Tests	17
7.1.1	Umgesetzte Unit Tests	17
7.1.2	ATRIP-Regeln	19
7.1.3	Code Coverage.....	20
7.2	TDD	20
8	Refactoring	22
8.1	Code Smells	22
8.1.1	Duplicated Code	22
8.1.2	Long Method.....	22
8.1.3	Large Class	22
8.1.4	Switch Statement.....	22
8.1.5	Code Comments	22
8.2	Refactoring	23
8.2.1	Method Extraction.....	23
8.2.2	Interface Extraction	23
8.2.3	Rename Class.....	23

Literaturverzeichnis

Anhang

Abbildungsverzeichnis

1	Architektur-Pakete des Projektes	3
2	Architektur des Projektes	4
3	Fassade im Projekt	7
4	Brücke im Projekt	8
5	Observer im Projekt	9
6	Domänenmodell des Projekts	11
7	Ergebnisse der Unit-Tests	18
8	AAA-Normalform an einem Beispiel-Test	18

Abkürzungsverzeichnis

USB Universal Serial Bus

Tabellenverzeichnis

Code-Verzeichnis

1 Einleitung

Diese Ausarbeitung wird als Prüfungsleistung für das Modul *Software Engineering 2* angefertigt. Die Inhalte basieren maßgeblich auf der zugrundeliegenden Vorlesung. Die Prüfungsleistung wird auf entstandenem Code einer Studienarbeit ausgeführt.

BESCHREIBUNG! *noch auf die Gliederung eingehen*

2 Problemstellung

Diese Ausarbeitung baut auf dem Code einer Studienarbeit auf.

Die Problemstellung der zugrundeliegenden Studienarbeit lautet wie folgt: „

BESCHREIBUNG! *Text hier einfügen!*

“

BESCHREIBUNG! *Darf das hier überhaupt so? Immerhin wird SWE vor der Studienarbeit abgegeben... Also brauchen wir eine Dependency Inversion für das Zitat :D*

3 Architektur

3.1 Dependency Rule

Wie in Abbildung 1 (Seite 3) zu sehen, zeigen alle Abhängigkeiten in die gleiche oder eine darunterliegende Schicht. Aufrufe werden mittels *virtuellen* Klassen und einer entsprechenden *Dependency Inversion* in weiter außen liegende Klassen ermöglicht.

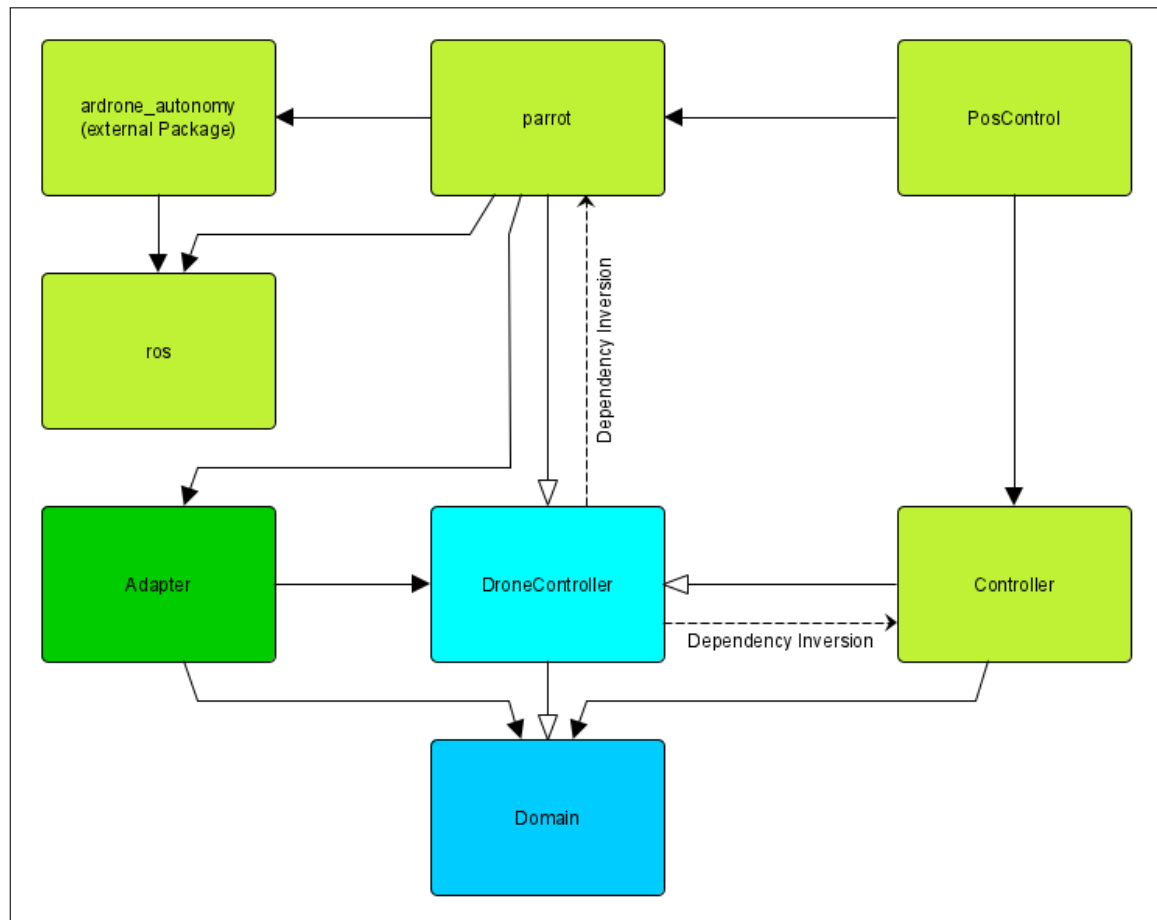


Abbildung 1: ARCHITEKTUR-PAKETE DES PROJEKTES

Abbildung 1 zeigt das Konzept der Architektur. Die Bedeutungen der Pfeile orientieren sich an UML-Klassendiagrammen. Die Farbgebung unterscheidet die verschiedenen Schichten der *Clean Architecture*: *Domain-Layer* (blau), *Application-Layer* (hellblau), *Adapter-Layer* (grün), *PlugIn-Layer* (hellgrün).

Ein detaillierteres Verständnis der Interaktion der Klassen soll durch Abbildung 2 (Seite 4) vermittelt werden.

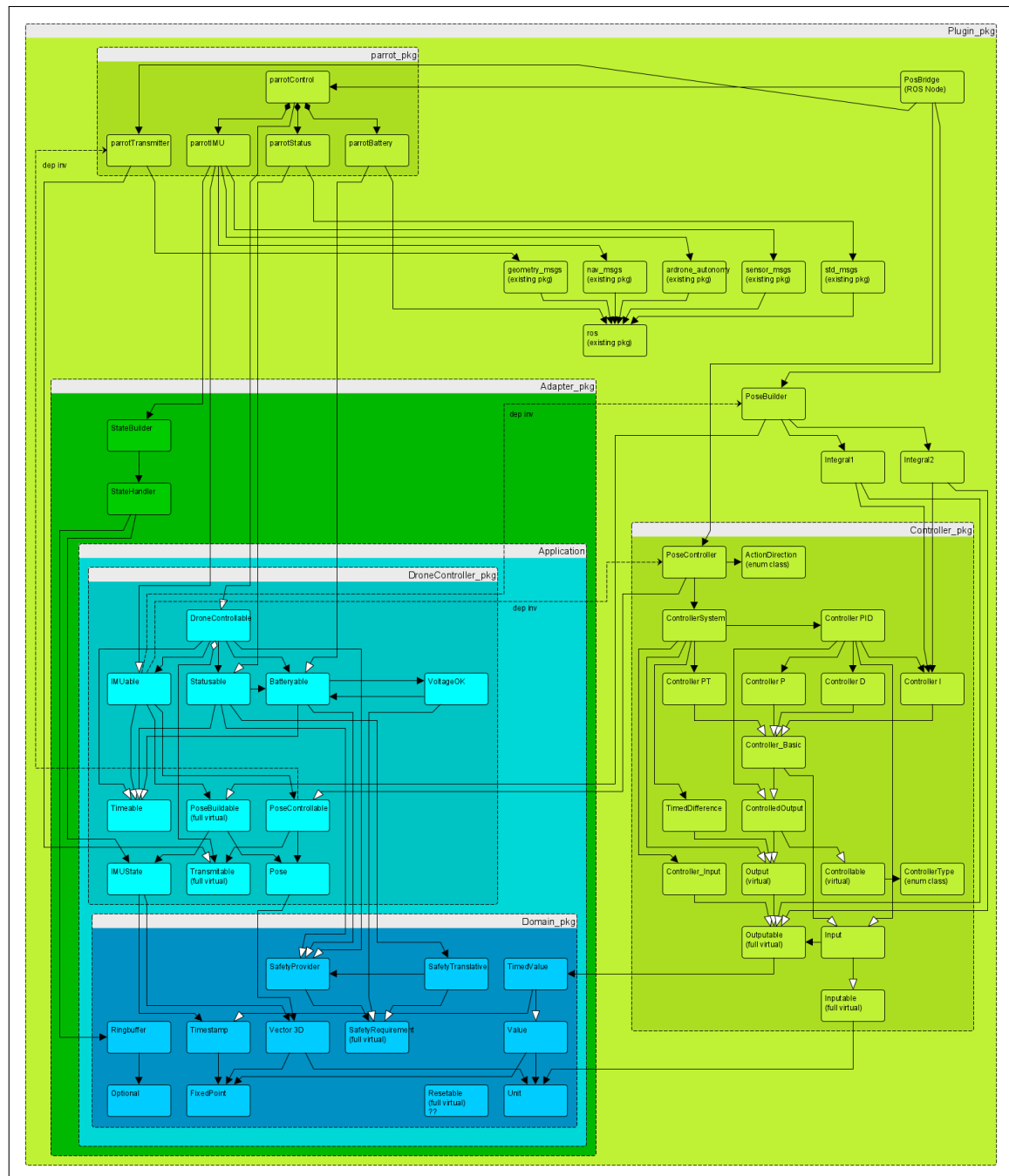


Abbildung 2: ARCHITEKTUR DES PROJEKTES

3.2 Domain Layer

Um ein Anlegen des *Abstraction Layers* aus Angeberei zu vermeiden¹, wurden die beiden Schichten *Abstraction Layer* und *Domain Layer* vereinigt. Tatsächlich könnten folgende Klassen in einem *Abstraction Layer* untergebracht werden:

- FixedPoint

¹Siehe Seite 23 in den Vorlesungsfolien zu *Clean Architecture*

- Unit
- Value
- Vector3D
- Timestamp
- TimedValue
- Optional

Da die Klasse `SafetyRequirement`, Klasse `SafetyProvider` und Klasse `SafetyTranslative` den Use Case ‚überwacht einen sicheren Start beziehungsweise Flug‘ abbilden, sind diese Klassen entgegen der Aufteilung innerhalb des betrachteten Projekts dem *Application Layer* zuzuordnen.

3.3 Application Layer

Der *Application Layer* bildet eine *Bridge* ab. Hierdurch kann die Hardware im zukünftigen Projektverlauf ausgetauscht werden, ohne den Kern der Anwendung zu beeinflussen. Darüber hinaus bietet das Paket `DroneController` die grundlegenden Funktionalitäten für die Positionsregelung einer Drohne an. Umfangreichere Funktionen, wie zum Beispiel die Kommunikation mit der Hardware, werden in dafür vorgesehenen Paket des *PlugIn Layers* implementiert.

Die Klasse `IMUState` und die Klasse `Pose` bilden Daten ab. Somit sind diese entgegen der Darstellung in Abbildung 2 (Seite 4) dem *Domain Layer* zugehörig.

3.4 Adapter Layer

Die Klasse `StateBuilder` ermöglicht eine Transformation der Informationen der *IMU-Message* in die dafür vorgesehene Klasse `IMUState`. Die entstehende Instanz wird anschließend zur weiteren Verarbeitung in die Implementierung der Klasse Klasse `PoseBuildable` übergeben.

3.5 PlugIn Layer

Das Framework *ROS* wurde gezielt in den *PlugIn Layer* aufgenommen, um etwaigen Änderungen dieses Frameworks mit mäßigem Aufwand begegnen zu können. *Anmerkung:* „The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot

applications. From drivers to state-of-the-art algorithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all open source.“[1]

Des Weiteren wird die vollständige Implementierung der einzusetzenden Drohne, sowie des Controllers im *PlugIn Layer* geführt. Somit ist ein einfacher Austausch möglich.

4 Entwurfsmuster

BESCHREIBUNG! *Blabla für Einleitung des Kapitels*

Bei der Beschreibung der einzelnen Entwurfsmuster orientiert sich diese Arbeit an der Beschreibungsstruktur der GoF (Gang of Four).

4.0.1 Fassade

Die Klasse Klasse `PoseController` kann als Fassade angesehen werden, da hier die Interaktion mit den darin verwalteten Klassen gekapselt wird.

BESCHREIBUNG! *fällt evtl raus??*

Klassifikation

Struktur und beteiligte Akteure

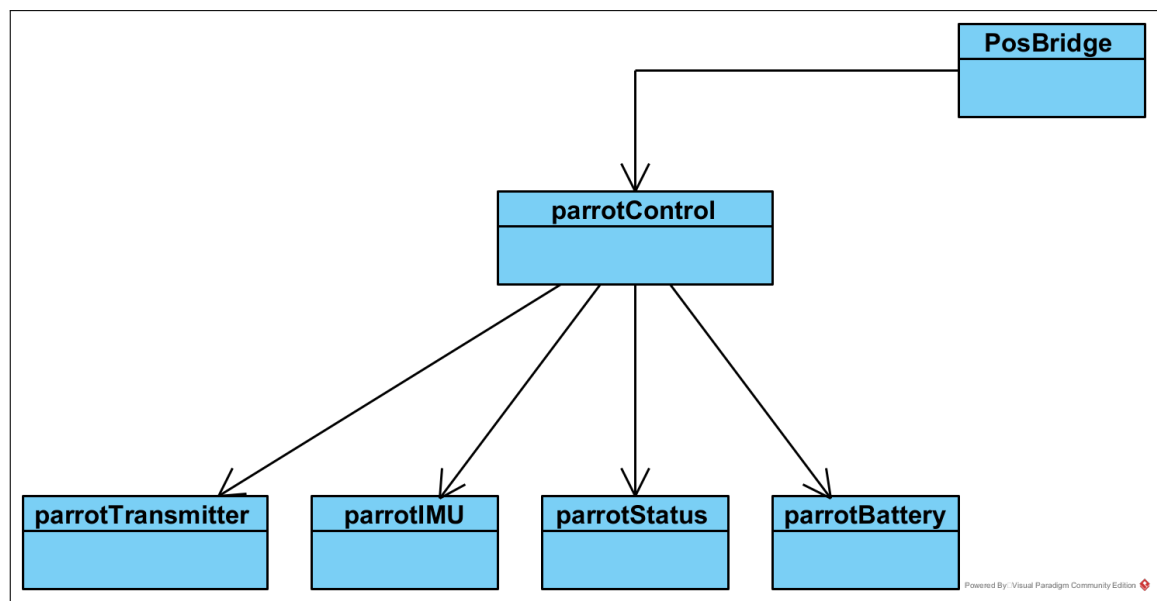


Abbildung 3: FASSADE IM PROJEKT

Abbildung 3 zeigt **BESCHREIBUNG!**

Motivation

Auch hier Design-Entscheidung. Interaktion mit einer Klasse ist einfacher, als sich um alles selbst kümmern zu müssen.

4.0.2 Brücke

DroneController_pkg zu parrot_pkg

„Eigentlich basiert jede Dependency Inversion auf einer Brücke“

Klassifikation

Struktur und beteiligte Akteure

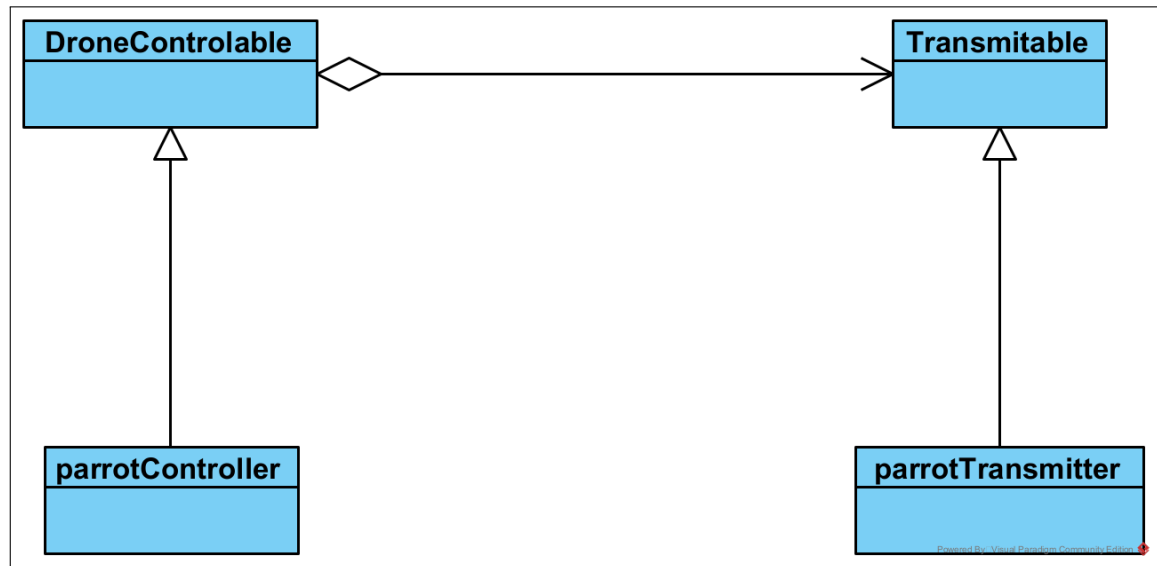


Abbildung 4: BRÜCKE IM PROJEKT

Abbildung 4 zeigt **BESCHREIBUNG!**

Motivation

Das ist eine Design-Entscheidung um im weiteren Lebenszyklus der *Application* verschiedene Dronen als *PlugIn* einbinden zu können.

4.0.3 Beobachter

BESCHREIBUNG! *hier einfach mal ROS blabla. evtl von der ROS Homepage übersetzen und so*

Klassifikation

Struktur und beteiligte Akteure

Motivation

BESCHREIBUNG! *Ist halt Grundlage von ROS*

Die *ROS-Publisher* und *Subscriber* kommunizieren über *ROS-Topics* miteinander.

(vgl. [2])

The and the turtlesim node are communicating with each other over a ROS Topic. turtlesim is publishing the key strokes on a topic, while turtlesim subscribes to the same topic to receive the key strokes.

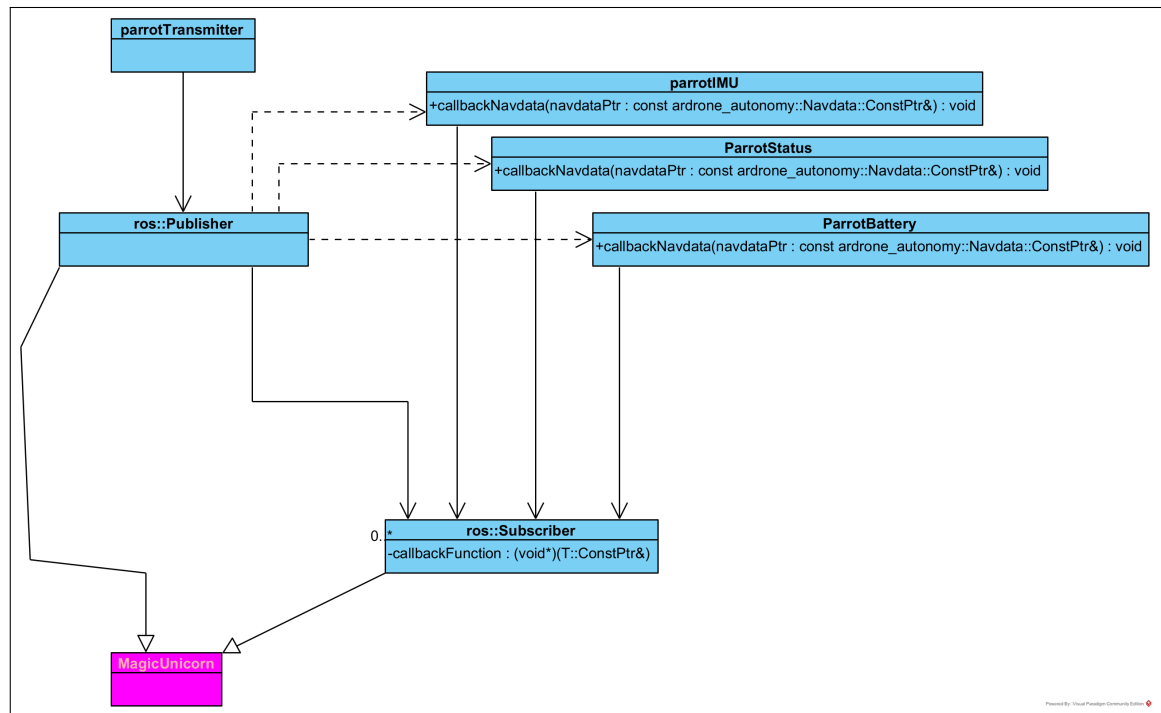


Abbildung 5: OBSERVER IM PROJEKT

Abbildung 5 zeigt **BESCHREIBUNG!**

BESCHREIBUNG! *evtl überarbeiten ;)* Alles, was wir nicht verstanden haben (oder nicht genauer erörtern wollten) wird durch den Platzhalter *MagicUnicorn* veranschaulicht. Ernsthaft, bei Nachfragen selbst mal den ROS-Jungel durchkämmen.

4.0.4 Eventbus

BESCHREIBUNG!

ACHTUNG, dieses Kapitel ist ein Spass-Kapitel!!

5 Domain Driven Design

Unter *Domain Driven Design* (DDD) versteht sich eine Vorgehensweise für Software-Modellierung.

Um Software zu modellieren, welche nach DDD entworfen werden soll, wird ein Domänenmodell von der Rolle *Domänen-Expert*in* erstellt. Personen, die diese Rolle bekleiden benötigen keine Programmierkenntnisse. Die Implementierung einer Software bildet die grundlegenden Zusammenhänge des Domänenmodells ab.

Die nachfolgenden Unterkapiteln wird die Anwendung dieser Vorgehensweise auf dem zugrundeliegenden Code analysiert.

5.1 Domänenmodell

Um Personen der Rolle *Domänen-Expert*in* vor unnötigen Programmier-Details (Komplexität) zu schützen, wird ein *Domänenmodell* entwickelt. Hier bildet sich lediglich die *inhärente Komplexität* ab.

Im vorliegenden Projekt können grundsätzlich zwei verschieden abgegrenzte Domänenmodelle zugrunde gelegt werden:

- Ein Modell, welches sich auf die regelungstechnischen Zusammenhänge des Projekts bezieht.
- Ein Modell, welches die Interaktion mit der Drohne beschreibt.

Das Modell in Abbildung 6 (Seite 11) begrenzt sich auf die Betrachtung der Regelung einer Drohne.

5.2 Ubiquitous Language

Die Ubiquitous Language ist eine einheitliche Sprache, auf die sich Entwickler und Domänenexperten einigen und gemeinsam verwenden. In diesem Projekt ist aus Zeitgründen keine solche einheitliche Sprache definiert worden, da die Rollen *Domänen-Expert*in* und *Entwickler*in* von der gleichen Person eingenommen wurden. Verständnisprobleme sind hierbei nicht zu erwarten.²

²Aber nie ganz auszuschließen.

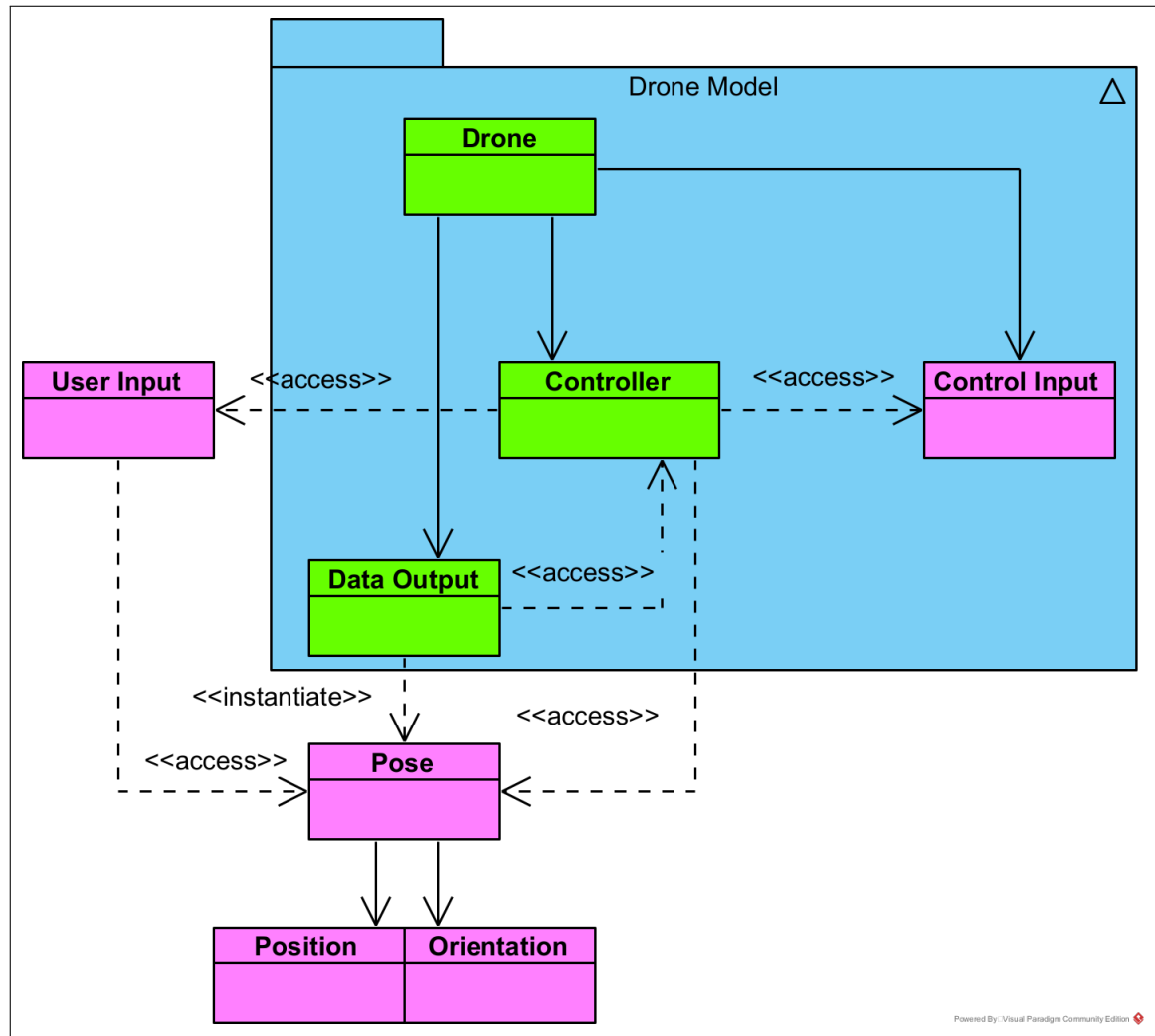


Abbildung 6: DOMÄNENMODELL DES PROJEKTS

Die gestrichelten Pfeile deuten eine Nutzung der entsprechenden Klasse an, wobei hier kein Attribut innerhalb der ausgehenden Klasse vorgesehen.

Das Aggregat wird als blauer Kasten dargestellt.

Objekte, welche in Kawasaki-grün eingefärbt sind, entsprechen *Entities*.

Objekte, welche in hell-lila eingefärbt sind, entsprechen *Value Objects*.

Um zukünftigen Anwendenden den Zusammenhang zwischen Domänenmodell und Code aufzuzeigen, soll anschließend eine Zuordnung von Objekten des Domänenmodells zu Klassen im Code folgen.

Domänenmodell	Klassenbezeichnung
Drohne	parrotControl
Data Output	parrotIMU
Control Input	parrotTransmitter
Controller	PoseController
Position	Vector3D
Orientation	Vector3D
User Input	ROS-Message

5.3 Repositories

In dem betrachteten Code befindet sich keine Interaktion mit persistentem Speicher. Die Regelung der Drohne wird online³ durchgeführt und nicht innerhalb der Domäne aufgezeichnet. Aus den genannten Gründen folgt, dass der betrachtete Code keine *Repositories* beinhaltet.

5.4 Aggregates

Entsprechend Abbildung 6 (Seite 11) lässt sich im Domänenmodell ein Aggregat identifizieren:

Drone Model

Das Aggregat *Drone Model* bildet die Funktionalität einer physisch vorhandenen Drohne ab.

5.5 Entities

Nachfolgend werden die *Entities* aus Abbildung 6 (Seite 11) beschrieben.

Drone

Das Objekt *Drone* bildet die physische Drohne ab. Das physisch vorhandene Pendant kann Zustände ändern (ruhend, fliegend, verunfallt), so auch das Objekt der Domäne.

Data Output

³*online* ist in diesem Kontext ein Fachbegriff der Robotik. Hierunter wird eine unmittelbare Verarbeitung eingehender Daten verstanden.

In dem Objekt *Data Output* werden Posen aus Beschleunigungsdaten und der Ausrichtung der Drohne erzeugt. Es handelt sich hier explizit nicht um eine *Pure Fabrication* (*Factory*), da hier eine Funktionalität der Drohne (Übergabe von Beschleunigungsdaten und Ermittlung der lokalen Pose) umgesetzt wird. Diese Funktionalität ist für die Domäne essentiell. Für die Umrechnung von Beschleunigungsdaten zu einer Position werden Integrale gebildet und somit Zustände zur Laufzeit verändert. Dies erhebt das Objekt *Data Output* zu einem *Entity*.

Controller

Im *Controller*-Objekt werden entsprechend regelungstechnischer Ansätze Stellgrößen gebildet. Hierzu werden im Allgemeinen *PID-Regelglieder* eingesetzt, diese enthalten ein integralbildendes Regelglied. Hieraus folgt, wie bereits für das Objekt *Data Output* angemerkt, die Eigenschaft als *Entity*.

5.6 Value Objects

Ein beliebiges Objekt ohne eigene Identität wird auch als *Value Object* (VO) bezeichnet. Die Kategorisierung als solches ergibt sich durch die *immutable*-Eigenschaft. Hieraus leitet sich eine Gleichheit der Objekte bei gleichen Attribut-Werten ab.

Nachfolgend werden die *Value Objects* aus Abbildung 6 (Seite 11) beschrieben.

Pose

Eine Pose ist eine mathematische Beschreibung für einen Punkt und eine Ausrichtung in einem Raum. Nimmt die Drohne eine andere Pose ein, wird hierfür eine separate Instanz generiert.

Die von der Pose gekapselten Objekte *Position* und *Orientierung* werden als mathematische Vektoren im 3D-Raum abgebildet.

Control Input

Der *Control Input* ist ein Objekt, welches Steuerungsdaten an die physische Drohne übersendet. Im Sinne der Domäne entspricht das Objekt *Control Input* einem *Value Object*. Sämtliche Parameter werden bei der Initialisierung des Objekts übergeben.

User Input

Als *User Input* werden regelungstechnische Führungsgrößen verstanden, welche an den Controller der Drohne übergeben werden. Diese besitzen keinen Zustand.

6 Programming Principles

6.1 SOLID

BESCHREIBUNG! *Sem5 VL4*

6.1.1 Single Responsibility Principle

BESCHREIBUNG! *Was genau soll hier beschrieben werden? nochmal Vorlesung anhören!*

BESCHREIBUNG! *Hier evtl eine Tabelle mit allen Klassen (nach Layer und Alphabet geordnet?) und dem von uns definierten Zweck?*

Beispiel: Klassen aus DroneController_pkg mit jeweiligen Aufgaben beschreiben. => Michael

6.1.2 Open/Closed Principle

BESCHREIBUNG! *Vorlesung Sem5 VL xx*

Offen für Erweiterung. Geschlossen gegenüber Veränderungen.

BESCHREIBUNG! *TODO: VORLESUNG ANHÖREN!*

Beispiel: Value und TimedValue ??

Beispiel: Der ganze Input/Output Kram für die Controller

Beispiel Closed: ControllerSystem

Gegenbeispiel: Integral1 und Integral2, Erklärung: zu Aufwändig und idR nicht benötigt in diesem Anwendungsfall.

6.1.3 Liskov Substitution Principle

BESCHREIBUNG! *Vorlesung Principals Folie 23.*

Kompletter Code ist theoretisch Kovariant.

Hier als Beispiel das Controller-Array in ControllerSystem.

6.1.4 Interface Segregation Principle

implizit Single responsibility.

Klient soll nur das gezeigt bekommen, was er tatsächlich auch benötigt.

Beispiel: DroneController_pkg

6.1.5 Dependency Inversion Principle

BESCHREIBUNG! *Transmittable in PoseController*

Die Brücke.

IMUable => PoseBuilder PoseControlable => parrotTransmitter

PoseControlable ist zwar maßgeblich virtual, aber implementiert eine Methode (transmitAction). Damit sind wir uns unklar, ob das noch als dependency inversion zählt. Auch so bei DroneControlable => parrotStatus

6.2 GRASP

6.2.1 Kopplung

PoseController ist ein Monolith.

ROS ist strukturell sehr lose gekoppelt (verschiedene Prozesse, die mit Topics (Observer) miteinander kommunizieren).

Brücke ist eher lose gekoppelt.

6.2.2 Kohäsion

mit niedriger Kohäsion wird das single responsibility principle. Wird angewandt. Beispiel: Controller-Aufbau in Domain.

6.3 DRY

Don't repeat yourself

Aus aufwändiger Domain-Architektur ergibt sich, dass das umgesetzt wurde => viele kleine Klassen.

Sichtbar besonders in der Umsetzung von Full Virtual Klassen (Interfaces in Java) und darauf aufbauend Klassen, die diese Methoden implementieren. Beide werden in „höherwertigen“ Klassen eingesetzt. zum Beispiel Outputable und Output.

Wenn DRY nicht angewendet, ist halt bescheuert - weil faule Menschen wollen doch nicht extra Mehrarbeit. **BESCHREIBUNG!** *evtl andere fragen, wie hier argumentiert wurde*

BESCHREIBUNG! *Umsetzung mit Duplication Checker Tool*

<https://cppdepend.com/blog/?p=556>

Duplication **Commit** `edd7726743b9eb67e159112919224d54c54be670` für die Klasse `FixedPoint<T>`

6.4 KISS

BESCHREIBUNG! *Sem5 VL6*

evtl Vector3D mit den 3 rotate_ Methoden...? Statt eine massive Methode, die alle 3 erlaubt...

6.5 YAGNI

BESCHREIBUNG! *Sem5 VL6-2 Min36*

YAGNI (*You ain't gonna need it*) ist ein Prinzip, nach dem Code nur erzeugt wird, wenn dieser unmittelbar benötigt wird. Auch absehbar benötigte Funktionalität ist zurückzustellen.

In diesem Projekt wurde dieses Prinzip aus der Präferenz des Entwicklers heraus nicht angewandt: Grundlegende Funktionalitäten werden hinzugefügt, *WEIL* sie benötigt werden, nicht *WENN* sie benötigt werden.

BESCHREIBUNG! Gedanklich hinundher springen?? Widerspricht guter Planung (Danke Theresa :))

Abweichend hiervon wurde dieses Prinzip im Zusammenhang mit der Anwendung von TDD umgesetzt (siehe Kapitel 7.2 *TDD* (Seite 20)).

6.6 Conway's Lay

Die Entwicklung der betrachteten Software basiert auf einer Person, daher kann davon ausgegangen werden, dass kein Kommunikationsproblem vorliegt. Kommunikation mit Personen, die nicht Teil des Entwicklungsteams sind (Auftraggeber beziehungsweise Betreuer der Studienarbeit), wird hierbei nicht betrachtet.

7 Software Tests

Es gibt folgende Test-Arten:

- Unit Test
- Integration Test
- System Test
- Acceptance Test

Allerdings wird in dieser Ausarbeitung nur auf Unit Tests eingegangen.

7.1 Unit Tests

In dem betrachteten Code wurde auf das Google Test-Framework (*gtest* und *gmock*) zurückgegriffen. Hierbei handelt es sich um ein weitverbreitetes Test-Framework.

Es wurde darauf geachtet, den produktiv-Code vom Test-Code zu trennen. Dies wurde durch verschiedene Projekte innerhalb einer *MS Visual Studio Solution* umgesetzt.

7.1.1 Umgesetzte Unit Tests

Als Nachweis über das Vorhandensein der Unit-Tests soll an dieser Stelle ein Verweis auf die jeweiligen .cpp-Dateien genügen. Diese sind durch ihren Inhalt für sich selbst-sprechend. Wie allgemein wünschenswert: Alle Tests sind erfolgreich (siehe Abbildung 7 (Seite 18)).

- <https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e3>
- <https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e3>
- <https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e3>
- <https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e3>
- <https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e3>

BESCHREIBUNG! *hier Links statt der kompletten URI*

AAA-Normalform

Der allgemeine Aufbau eines Unit-Tests, wie er auch in diesem Projekt angewandt wurde, wird hier anhand eines Beispiels (siehe Abbildung 8 (Seite 18)) demonstriert.

Mocks

Test	Dauer	Merkmale	Fehlermeldung	Gruppenzusammenfassung
▶ ✓ 06_01_Test_Domain (58)	2 ms			Tests in Gruppe: 69
▶ ✓ 06_02_Test_DroneController .	< 1 ms			⌚ Dauer gesamt: 179 ms
▶ ✓ 06_03_Test_Adapter (7)	17 ms			Ergebnisse
▶ ✓ 06_04_Test_Controller (1)	2 ms			✓ 69 Bestanden
▶ ✓ 06_05_Test_Plugin (2)	158 ms			

Abbildung 7: ERGEBNISSE DER UNIT-TESTS

```

3      #include <gtest/gtest.h>
4      #include "Domain/Header.h"
5
6
7      TEST(Class_FixedPoint, InitInt10_raw0)
8      {
9          int Value = 10;
10         FixedPoint<0> FP(Value);
11
12
13         double Result = FP.getValueRaw();
14
15
16         EXPECT_EQ(Result, 10);
17     }
18

```

Abbildung 8: AAA-NORMALFORM AN EINEM BEISPIEL-TEST

oranger Kasten: Arrange-Bereich

rosa Kasten: Act-Bereich

grüner Kasten: Assert-Bereich

Der in diesem Abschnitt beschriebene Test enthält ein Mock. Hier wird sichergestellt, dass die Klasse `PoseController` nach der Berechnung der aktuellen Pose (Übergabe des Parameters in Methode `feedbackPose`) die Methode `transmitAction` der Implementierung der Klasse `Transmittable` aufruft.

<https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e57b68>

BESCHREIBUNG! *hier Links statt der kompletten URI*

Ein Sonderfall ergibt sich für die *Verify*-Phase des Mocks innerhalb des *gmock*-Frameworks. Diese wird nicht benötigt: „instead of checking the system state at the very end, mock objects verify that they are invoked the right way and report an error as soon as it

arises, giving you a handle on the precise context in which the error was triggered. This is often more effective and economical to do than state-based testing.“[3]

7.1.2 ATRIP-Regeln

Automatic

Um diese Eigenschaft zu gewährleisten, wurden Parameter *hard coded* in die Tests eingeführt (keine manuellen Interaktionen). Im betrachteten Code war negativ-Beispiel auffindbar.

Thorough

Die Anforderungen an die Testbarkeit dieses Projekts halten sich in Grenzen. Daher ist davon auszugehen, dass große Teile der Funktionalität nicht durch Tests abgedeckt sind.

Anmerkung: In Zuge der *TDD* Entwicklung der Klasse `FixedPoint<T>` wurde eine vollständige Testabdeckung gewährleistet (siehe Kapitel 7.2 *TDD* (Seite 20)).

Repeatable

Um diese Eigenschaft zu gewährleisten, wurden Parameter *hard coded* in die Tests eingeführt (Kein Zufall, keine Abhängigkeit von externen/berechneten Parametern). Durch den Einsatz eines Test-Frameworks können die Unit Tests jederzeit durchgeführt werden.

Independent

Tests wurden so geschrieben, dass sich diese nicht gegenseitig beeinflussen und jeweils nur einen Funktionlität abprüft.

Im Gegenzug wurden an diversen Stellen Tests für verschiedene parameter der selben Methode separat geschrieben. Hier würde beim Fehlerfall dieser Methode mehrere Tests scheitern. zum Beispiel Abprüfen der Methode `getMedianState`:

<https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e57b68>

<https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e57b68>

<https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b5814c1bb731abf0f7af4352289e57b68>

BESCHREIBUNG! *hier Links statt der kompletten URI*

Um die Debug-Tätigkeit zu vereinfachen, werden für einen Fehlerfall in aufwändigeren Tests menschenlesbare Informationen über die errechneten Ergebnisse zur Verfügung gestellt⁴. Hierdurch wird das Auffinden der Ursache des Fehlers erleichtert.

⁴Das eingesetzte Test-Framework zeigt hier lediglich den Speicherinhalt in hexadezimalen Format an.

Die Nutzung von Mocks sorgt unter anderem für eine Unabhängigkeit von Fehlerfällen anderer Klassen durch ein definiertes Verhalten des Mocks für genau den Test, in dem dieser Mock trainiert wurde (siehe Kapitel 7.1.1 *Umgesetzte Unit Tests* (Seite 17)).

Professional

Allgemein wurde versucht, die Lesbarkeit der Tests zu gewährleisten, indem kurze und aussagekräftig⁵ Tests geschrieben wurden. Hierzu wurden die Abschnitte aus Kapitel 7.1.2 *ATRIIP-Regeln* (Seite 19) mit doppelten Leerzeilen getrennt und zudem an geeigneten Stellen Leerzeilen zur Übersichtlichkeit eingefügt.

Leider konnte dies aufgrund der Komplexität des zu testenden Use Cases nicht immer eingehalten werden: zum Beispiel <https://github.com/MobMonRob/ROSLabDrohne/blob/9edc7b581> (Zeilenanzahl > 25 Zeilen)

BESCHREIBUNG! *hier Links statt der kompletten URI*

7.1.3 Code Coverage

BESCHREIBUNG! *Reka fragen*

using coverlet with Visual Studio

Getting Started

- installiere Microsoft.NET.Sdk
- installiere Microsoft.NET.Test.Sdk (NuGet.org)

using Google Test with Visual Studio

7.2 TDD

Test Driven Development (TDD) ist ein Prozess in der Software-Entwicklung, wonach jede Klasse beziehungsweise Funktion im produktiven Code bereits im Vorfeld durch einen Test abgedeckt wird.

Hierbei unterscheidet sich das *TDD* von *Test First* dadurch, dass *TDD* lediglich jeweils einen Test vor dem produktiven Code liegt, wobei *Test First* eine beliebige Anzahl an Tests bereitstellen kann, bevor produktiver Code entsteht. **BESCHREIBUNG!** *Quelle die letzte Vorlesung - Sem6 VL 3 oder so?*

⁵‘aussagekräftig, ist immer subjektiv

Im zugrundeliegenden git-Repository wird die Entwicklung einer Klasse nach dem *Test First*-Prinzip im **Branch** `createFixedPoint` mit folgendem **Commit** gestartet.
Commit `a4f9bd383c32a7d6d1b0be2319764a664199c2d2`

8 Refactoring

8.1 Code Smells

8.1.1 Duplicated Code

Wie bereits in Kapitel 6.3 *DRY* (Seite 15) beschrieben, finden sich Code-Duplikate im Code. Der hier zu erwähnende Code befindet sich in folgender Klasse: Klasse `FixedPoint<T>` (Commit `edd7726743b9eb67e159112919224d54c54be670`)

+ Funktionen (arithmetische Operatoren)

8.1.2 Long Method

Klasse `PoseBuilder`, Methode `updatePose` Commit `73f3400f4866b27200a37be5d3a80fa8394b2c48`

Wie innerhalb der Methode ersichtlich ist wird ein manipulierender Code durch einen Block eingefasst. Dieser Block kann in eine eigene Methode verschoben werden und somit in der betrachteten Methode zu einer Funktionalität umgeändert werden.

8.1.3 Large Class

Commit `73f3400f4866b27200a37be5d3a80fa8394b2c48` Klasse `FixedPoint<T>`

Sehr viele Methoden / Funktionen, um die Funktionalität gewährleisten zu können. Ist eine Klasse, die Arithmetik implementiert. **BESCHREIBUNG!** *bitte in schönem Deutsch*

8.1.4 Switch Statement

BESCHREIBUNG! *Theresa denkt sich einen wundervollen Namen für PosBridge aus und dann den Commit mit diesem Namen hier angeben.*

Methode `callbackKeys`

Grundsätzlich sollen *Switch Statements* vermieden werden. In diesem Fall wird davon ausgegangen, dass keine weitere Funktionalität zur Drohne hinzugefügt wird.

8.1.5 Code Comments

An dieser Stelle sollen lediglich Links aufgezählt werden.

- <https://github.com/MobMonRob/ROSLabDrohne/blob/d39df8957ec8659b8a6683d8d8ba853d>
- <https://github.com/MobMonRob/ROSLabDrohne/blob/d39df8957ec8659b8a6683d8d8ba853d>

- <https://github.com/MobMonRob/ROSLabDrohne/blob/d39df8957ec8659b8a6683d8d8ba853d>

Informative Comments und damit sinnvoll:

- <https://github.com/MobMonRob/ROSLabDrohne/blob/d39df8957ec8659b8a6683d8d8ba853d>
- <https://github.com/MobMonRob/ROSLabDrohne/blob/d39df8957ec8659b8a6683d8d8ba853d>

8.2 Refactoring

8.2.1 Method Extraction

Angewandt auf die in Kapitel 8.1.2 *Long Method* (Seite 22) beschriebene Methode.

Commit 9dd6821a9c63bf1682806b8ab319d6004247d3ca

8.2.2 Interface Extraction

Da sich die Hardware (Drohnen-PlugIn) im Projektverlauf geändert hat, wurde diese Chance genutzt, eine Bridge einzuführen. Dies hatte den Hintergrund, einen Umstieg auf andere Hardware zukünftig zu erleichtern. Hierzu sind diverse, teilweise virtuelle⁶, Klassen entstanden.

Commit 2b1bccb0f9b2671ace7fb91b6ed86e2be2284596

BESCHREIBUNG! *Outputable und Inputable suchen... Da wurde das nochmal angewandt.*

8.2.3 Rename Class

In der Vorlesung wird das Umbenennen einer Methode angesprochen. Ziel ist hier, die Verständlichkeit des Codes zu erhöhen. In dem dieser Programmentwurf zugrundeliegenden Code wurden Klassen gefunden, bei welchen die Bezeichnung nicht selbstsprechend erscheint. Um dieser Problematik entgegenzuwirken, soll eine dieser betreffenden Klassen umbenannt werden. Hierbei handelt es sich um die Klasse Klasse **PosBridge**.

Commit

BESCHREIBUNG! *muss noch gemacht werden :D*

⁶virtuelle Klassen in der Sprache c++ entsprechen *abstrakten* Klassen in Java.

Literaturverzeichnis

- [1] ROS - Robot Operating System,
online, <http://ros.org>
veröffentlicht -unbekannt-, abgefragt 10.04.2022
- [2] Understanding ROS Topics,
online, <http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics>
veröffentlicht -unbekannt-, verändert 18.07.2019, abgefragt 27.03.2022
- [3] Legacy gMock FAQ,
online, https://google.github.io/googletest/gmock_faq.html
veröffentlicht -unbekannt-, abgefragt 22.04.2022
- [4] Saks D., Better even at the lowest levels,
online, <https://www.embedded.com/better-even-at-the-lowest-levels/>
veröffentlicht 01.11.2008, verändert 05.12.2020, abgefragt 28.07.2021
- [5] Application Note Object-Oriented Programming in C,
online, https://www.state-machine.com/doc/AN_OOP_in_C.pdf
veröffentlicht 06.11.2020, abgefragt 28.07.2021
- [6] Kirk N., How do strings allocate memory in c++?,
online, <https://stackoverflow.com/questions/18312658/how-do-strings-allocate-memory-in-c>
veröffentlicht 19.08.2013, abgefragt 17.08.2021
- [7] Bansal A., Containers in C++ STL (Standard Template Library),
online, <https://www.geeksforgeeks.org/containers-cpp-stl/>
veröffentlicht 05.03.2018, verändert 12.07.2020, abgefragt 17.08.2021
- [8] Automatic Storage Duration,
online, <https://www.oreilly.com/library/view/c-primer-plus/9780132781145/ch09lev2sec2.html>
veröffentlicht -unbekannt-, abgefragt 17.08.2021
- [9] Noar J., Orda A., Petruschka Y., Dynamic storage allocation with known durations,
online, <https://www.sciencedirect.com/science/article/pii/S0166218X99001754>
veröffentlicht 30.03.2000, abgefragt 17.08.2021

Anmerkung: Wird hier ein Veröffentlichungsdatum als “-unbekannt-“ markiert, so konnte diese Angabe weder auf der entsprechenden Webseite, noch in deren Quelltext ausfindig gemacht werden.

Anhang