

**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 Pakete .....	3
4 Entwurfsmuster .....	4
4.0.1 Beobachter .....	4
4.0.2 Brücke .....	4
4.0.3 Fassade .....	4
5 Domain Driven Design .....	5
5.1 Domänenmodell .....	5
5.2 Ubiquitous Language .....	5
5.3 Repositories .....	5
5.4 Aggregates .....	5
5.5 Entities .....	5
5.5.1 Surrogatschlüssel .....	6
5.6 Value Objects .....	6
6 Programming Principles .....	7
6.1 SOLID .....	7
6.1.1 Single Responsibility Principle .....	7
6.1.2 Open/Closed Principle .....	7
6.1.3 Liskov Substitution Principle .....	7
6.1.4 Interface Segregation Principle .....	7
6.1.5 Dependency Inversion Principle .....	8
6.2 GRASP .....	8
6.2.1 Kopplung .....	8
6.2.2 Kohäsion .....	8
6.3 DRY .....	8
6.4 KISS .....	9
6.5 YAGNI .....	9
6.6 Conway's Lay .....	9

7	Software Tests .....	10
7.1	Unit Tests .....	10
7.1.1	Umgesetzte Unit Tests .....	10
7.1.2	ATRIP-Regeln .....	10
7.1.3	Code Coverage .....	10
7.2	TDD .....	11
8	Refactoring .....	12
9	Fazit und Ausblick .....	13
	Literaturverzeichnis	
	Anhang	

# Abbildungsverzeichnis

# **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**

Diagramm clean Architecture

Diskutieren, ob DroneController\_pkg Adapter oder Application ist. Wenn Adapter, dann nach draußen. Wenn Application, dann ist Controller ein PlugIn.

### **3.1 Dependency Rule**

### **3.2 Pakete**

## 4 Entwurfsmuster

### 4.0.1 Beobachter

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

### 4.0.2 Brücke

DroneController\_pkg zu parrot\_pkg

### 4.0.3 Fassade

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

*BESCHREIBUNG! fällt evtl raus??*

## 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.

BESCHREIBUNG! *Sem5 VL7* BESCHREIBUNG! *Wenn möglich Zitat auf Vorlesung DDD Seite 11*

BESCHREIBUNG! *word-cloud basteln?*

### 5.1 Domänenmodell

### 5.2 Ubiquitous Language

BESCHREIBUNG! *Sem5 VL7-1 Min64*

Ist eine Definition aller genutzten Begriffe.

Existiert nicht in diesem Projekt. Weil keine Zeit / keine Lust.

### 5.3 Repositories

BESCHREIBUNG! *Sem5 VL9-1 Min12*

### 5.4 Aggregates

BESCHREIBUNG! *Sem5 VL8-2 Min33*

BESCHREIBUNG! *Sem5 VL9-1 Min6*

### 5.5 Entities

BESCHREIBUNG! *Sem5 VL7-3 Min46*

BESCHREIBUNG! *Sem5 VL8-1 Min84*

### 5.5.1 Surrogatschlüssel

BESCHREIBUNG! *Sem5 VL8-2 Min7*

Mögliches Beispiel: Sequenz in ROS-Messages.

Weiterhin wird der Zeitstempel mitgeführt. Allerdings nicht mit dem Zweck als Schlüssel.

## 5.6 Value Objects

BESCHREIBUNG! *Sem5 VL7-3 Min35*

BESCHREIBUNG! *Sem5 VL7-3 Min49*

BESCHREIBUNG! *Sem5 VL8-1 Min36*

Sind immer gleich bei gleichen Variablen-Werten.

FixedPoint Unit Value TimeStamp Vector3D TimedValue

## 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>

## 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*

You ain't gonna need it

Erst Code hinzufügen, wenn benötigt.

Wurde im Projekt teilweise durchgeführt - git-Historie durchschauen... Fuck ist das aufwändig... -.-

Mein allgemeiner Programmier-Stil entspricht nicht vollständig diesem Prinzip. Grundlegende Funktionalitäten werden hinzugefügt, WEIL sie benötigt werden, nicht WENN es soweit ist :D Immerhin weiß man ja schon so grob, was alles im Projekt benötigt wird... Zugegeben, hier wird oft erst die leere Funktion eingebaut und der benötigte Code dann, wenn es an den jeweiligen Test/Ausprobieren geht. Beispiel: Vector3D::rotate() Body

Basis von TDD => Schreibe nur, was du wirklich brauchst.

## 6.6 Conway's Lay

**BESCHREIBUNG!** *Sem5 VL6-2 Min65*



## 7 Software Tests

Es gibt folgende Test-Arten:

- Unit Test
- Integration Test
- System Test
- Acceptance Test
- **BESCHREIBUNG!** *fehlt hier was?*

Allerdings wird in dieser Ausarbeitung nur auf Unit Tests eingegangen.

### 7.1 Unit Tests

Wahl fällt auf *banditcpp*, da es sich hierbei um ein *headers only* Framework handelt. Daraus folgt, dass die Test-Umgebung nicht in den Produktiv-Code eingebunden werden muss. Es kann quasi als PlugIn „aufgesetzt“ werden.

**BESCHREIBUNG!** *Bandit bietet leider keine Code Coverage...*

#### 7.1.1 Umgesetzte Unit Tests

verweis auf eine `tst.cpp` Datei (oder mehrere?)

Mocks

Test `callTransmitter` `Test_Application/test.cpp`

#### 7.1.2 ATRIP-Regeln

wurde angewandt. Verweis auf einen beliebigen Test.

#### 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?*

Im zugrundeliegenden `git`-Repository wird die Entwicklung einer Klasse nach dem *Test First*-Prinzip im **Branch** `createFixedPoint` mit dem **Commit** `a4f9bd383c32a7d6d1b0be2319764a6` gestartet.

## 8 Refactoring

## **9 Fazit und Ausblick**

# Literaturverzeichnis

- [1] Intel's First Microprocessor: Its invention, introduction, and lasting influence,  
online, <https://www.intel.de/content/www/de/de/history/museum-story-of-intel-4004.html>  
veröffentlicht -unbekannt-, verändert 08.04.2020, abgefragt 06.09.2021
- [2] 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
- [3] Application Note Object-Oriented Programming in C,  
online, [https://www.state-machine.com/doc/AN\\_OOP\\_in\\_C.pdf](https://www.state-machine.com/doc/AN_OOP_in_C.pdf)  
veröffentlicht 06.11.2020, abgefragt 28.07.2021
- [4] 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
- [5] 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
- [6] Automatic Storage Duration,  
online, <https://www.oreilly.com/library/view/c-primer-plus/9780132781145/ch09lev2sec2.html>  
veröffentlicht -unbekannt-, abgefragt 17.08.2021
- [7] 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