DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

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
30.04.2022

Gutachter der DHBW Karlsruhe Daniel Lindner

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung und Problemstellung

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.

Die Aufgabenstellung der Studienarbeit sieht vor, ein Regelsystem für die Regelung der Position beziehungsweise Pose einer Drohne (Quadrokopter) zu implementieren.
Anmerkung: Im Zuge der Studienarbeit ergab sich eine Änderung der Hardware. Der in dieser Ausarbeitung betrachtete Code entspricht demnach nicht dem vollständigen im git-Repository verfügbaren Code.

Nachfolgend wird der Programmcode nach verschiedenen Aspekten der Software-Entwicklung analysiert. Bei den betrachteted Aspekten hält sich diese Ausarbeitung maßgeblich an die Anforderungen des Dozenten.

Sofern nicht anders erwähnt, wird die Betrachtung des Codes auf folgendem Commit durchgeführt: Commit `69bbef051d3781e02020c0f80b191117fc4bbe93`

2 Clean Architecture

2.1 Dependency Rule

Wie in Abbildung 1 (Seite 2) zu sehen, zeigen alle Abhängigkeiten in die gleiche oder eine darunterliegende Schicht. Aufrufe werden mittels *virtuellen* Klassen[1] und einer entsprechenden *Dependency Inversion* in weiter außen liegende Klassen ermöglicht.

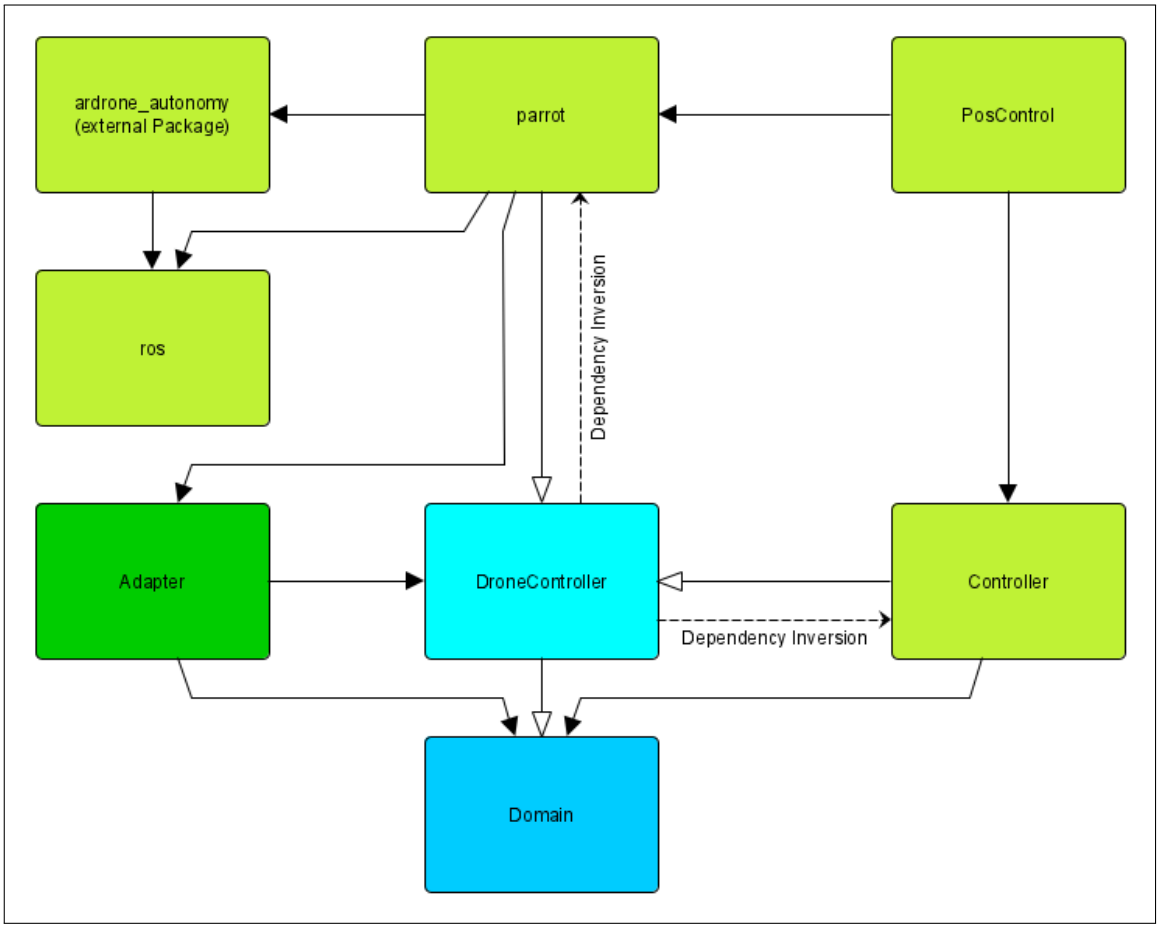


Abbildung 1: Architektur-Pakete des Projektes

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

Ein detallierteres Verständnis der Interaktion der Klassen soll durch Abbildung 2 (Seite 3) vermittelt werden.

[1] *virtuelle* Klassen in der Sprache c++ entsprechen *abstrakten* Klassen in Java.

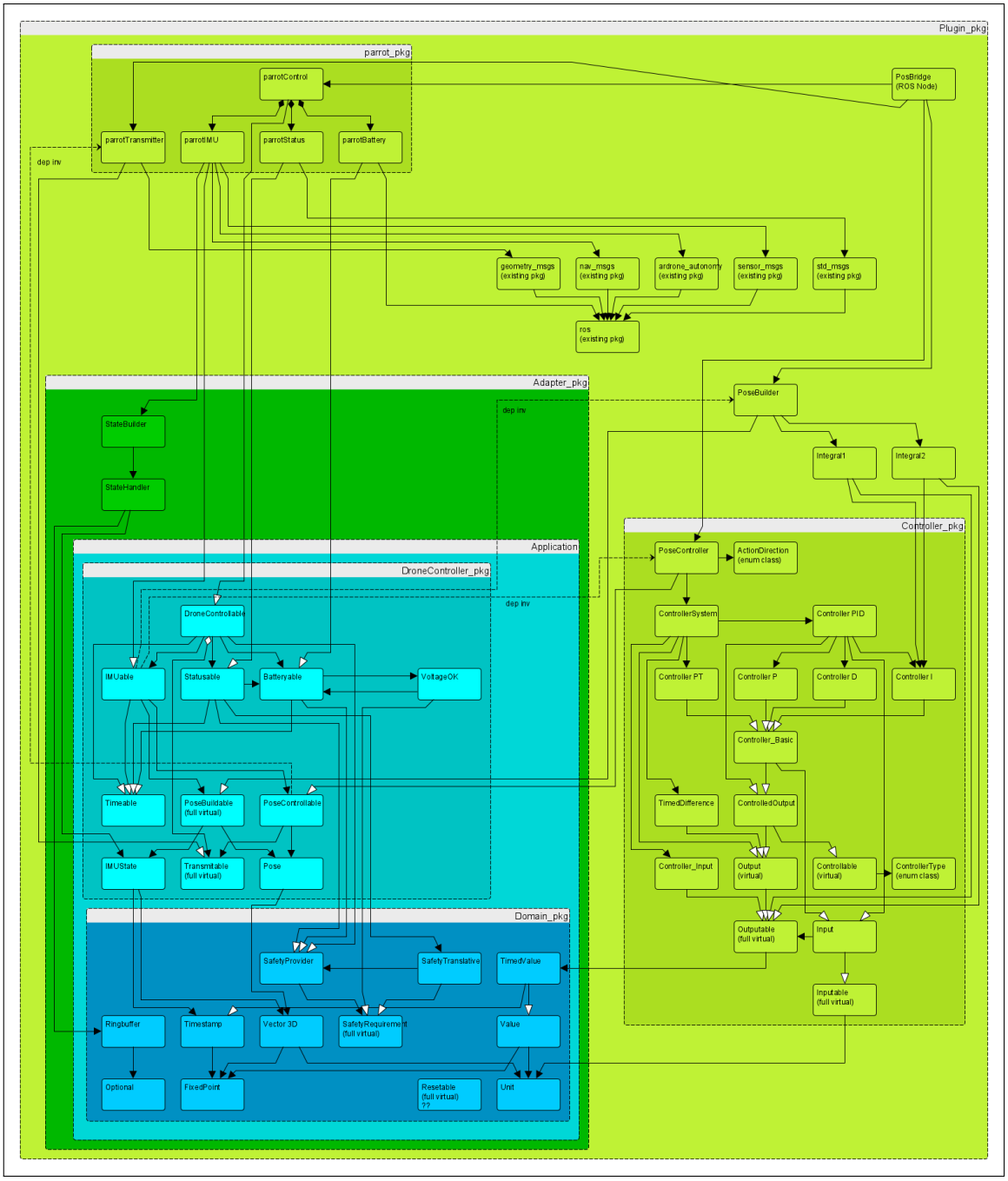


Abbildung 2: Architektur des Projektes

2.2 Domain Layer

Um ein Anlegen des *Abstraction Layers* aus Angeberei zu vermeiden[2], wurden die beiden Schichten *Abstraction Layer* und *Domain Layer* vereinigt. Tatsächlich könnten folgende Klassen in einem *Abstraction Layer* untergebracht werden:

- FixedPoint

[2]Siehe 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.

2.3 Application Layer

Der *Application Layer* bildet eine *Bridge* ab. Hierdurch kann die Hardware im zukünfigten 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 vorgesehenden 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 3) dem *Domain Layer* zugehörig.

2.4 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.

2.5 PlugIn Layer

Das Framework *ROS* wurde gezielt in den *PlugIn Layer* aufgenommen, um etwaigen Änderunges 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 einzusetztendes Drohne, sowie des Controllers im *PlugIn Layer* geführt. Somit ist ein einfacher Austausch möglich.

3 Entwurfsmuster

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

3.0.1 Fassade

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

Klassifikation

Objektbasiertes Strukturmuster[3]

Struktur

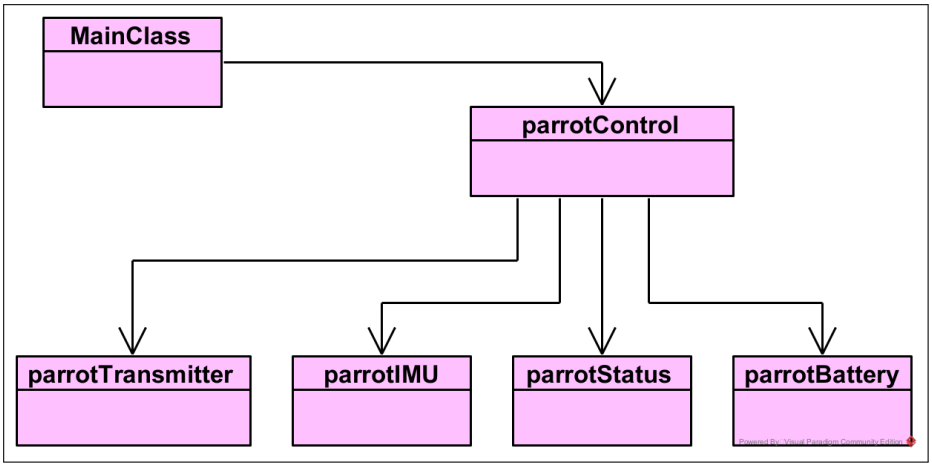


Abbildung 3: Fassade im Projekt

Akteure[3]

Akteur	Klassenbezeichnung
Fassade	parrotControl
SubsystemKlasse 0	parrotTransmitter
SubsystemKlasse 1	parrotIMU
SubsystemKlasse 2	parrotStatus
SubsystemKlasse 3	parrotBattery
Subsystem B	MainClass

Motivation

Für Dritte ist die Interaktionmit einer Klasse (Fassade) ist weniger kompliziert als die korrekte Initialisierung der von der Fassde gekapselten Klassen.

[3] Anhand der Vorlesungsfolien ‚Entwurfsmuster Fassade‘

3.0.2 Brücke

Klassifikation

Objektbasiertes Strukturmuster[4]

Struktur

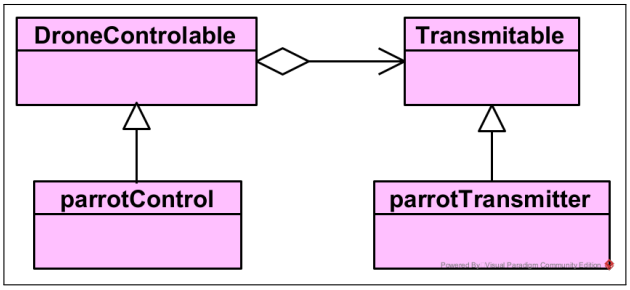


Abbildung 4: Brücke im Projekt

Akteure[5]

Basis-Paket:	DroneController
erbendes Packet:	parrot

Akteur	Klassenbezeichnung
Abstraction	DroneControlable
RefinedAbstraction	parrotControl
Implementor	Transmitable
ConcreteImplementor	parrotTransmitter

Motivation

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

3.0.3 Beobachter

Klassifikation

Objektbasiertes Verhaltensmuster[6]

[4]Information aus:
https://www2.htw-dresden.de/ muellerd/SWEngII_SS2019/09_Entwurfsmuster.pdf
[5]Anhand https://www.geeksforgeeks.org/bridge-design-pattern/
[6]Anhand der Vorlesungsfolien ‚Entwurfsmuster Beobachter'

Struktur

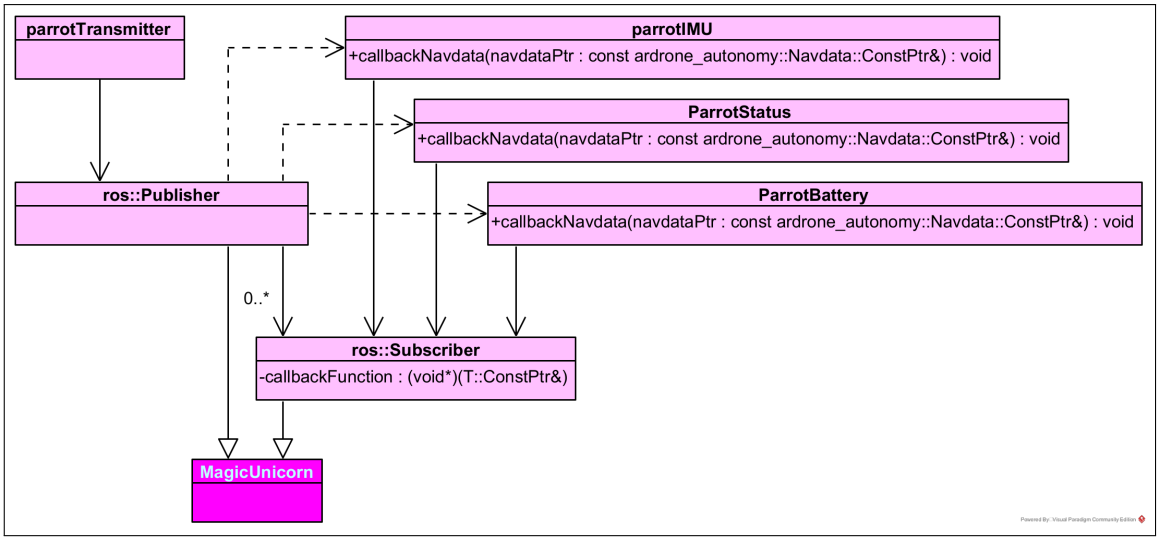


Abbildung 5: Observer im Projekt

Im *ROS*-Framework wird die Funktionalität mittels *Function Pointer* an den Beobachter übertragen. Eine Vererbung findet nicht statt.
Anmerkung: Die in der Abbildung als „MagicUnicorn„bezeichnete Klasse ist in diesem Zusammenhang die Untiefen des *ROS*-Frameworks, welches deutlich über das im verfügbaren Zeitraum aufbringbare Verständnis der Autor:innen hinausgeht.

Akteure[6]

Akteur	Klassenbezeichnung
Subjakt	ros::Publisher
KonkretesSubjekt	parrotTransmitter
Beobachter	ros::Subscriber
KonkreterBeobachter 0	parrotIMU
KonkreterBeobachter 1	parrotStatus
KonkreterBeobachter 2	parrotBattery

Motivation

Die *ROS-Publisher* und *Subscriber* kommunizieren über *ROS-Topics* miteinander. (vgl. [2])
Die Drohne ist ausschließlich mittels *ROS* erreichbar.

3.0.4 Eventbus

BESCHREIBUNG!

ACHTUNG, dieses Kapitel ist ein Spass-Kapitel!!

4 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 Anwednung dieser Vorgehensweise auf dem zugrundeliegenden Code analysiert.

4.1 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 10) begrenzt sich auf die Betrachtung der Regelung einer Drohne.

4.2 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.[7]

[7] Aber nie ganz auszuschließen.

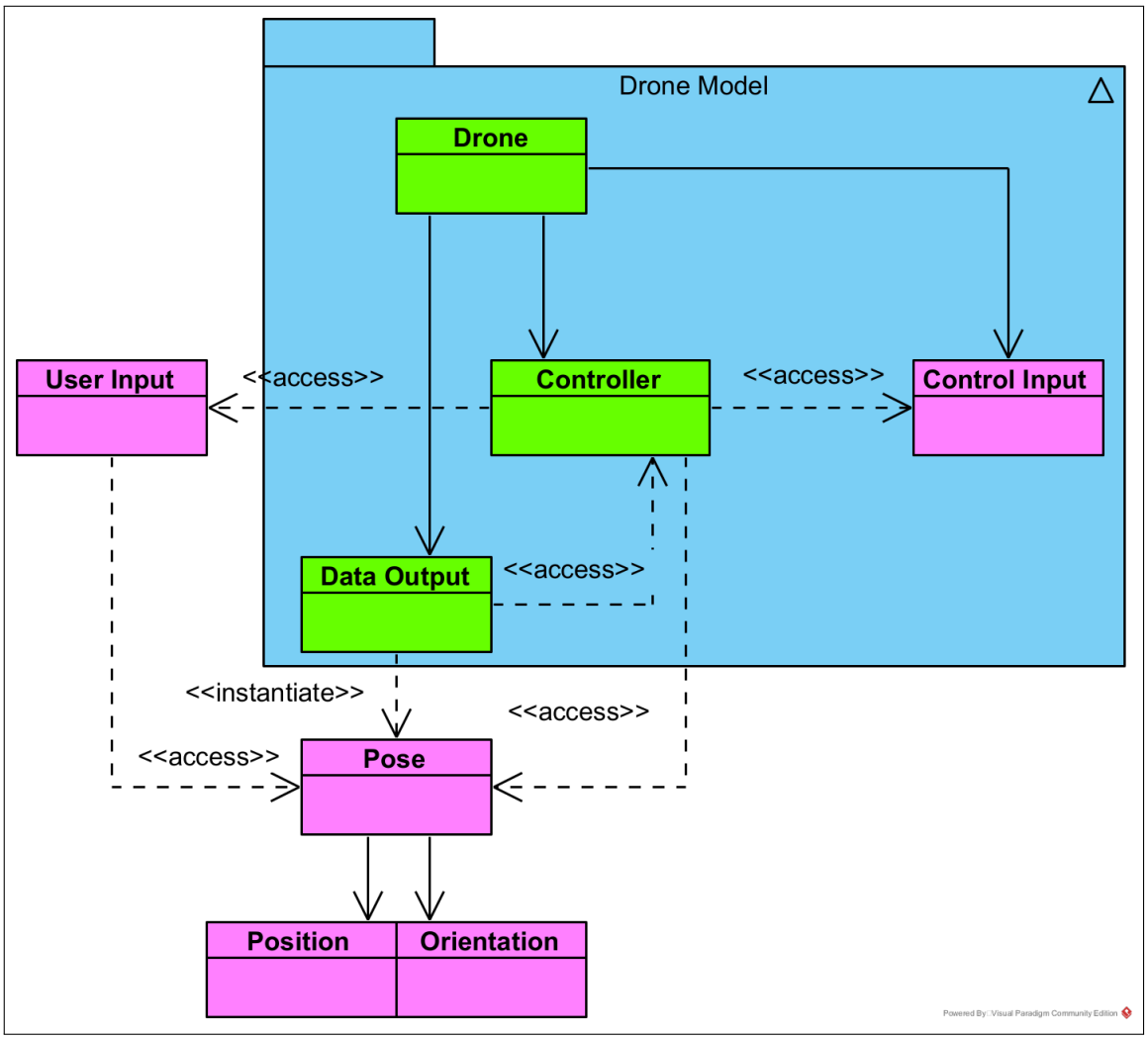


Abbildung 6: Domänenmodell des Projekts

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 Zuodnung 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

4.3 Repositories

In dem betrachteten Code befindet sich keine Interaktion mit persistentem Speicher. Die Regelung der Drohne wird online[8] durchgeführt und nicht innerhalb der Domäne aufgezeichnet. Aus den genannten Grüden folgt, dass der betrachtete Code keine *Repositories* beinhaltet.

4.4 Aggregates

Entsprechend Abbildung 6 (Seite 10) lässt sich im Domänenmodell ein Aggregat identifizieren:

Drone Model

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

4.5 Entities

Nachfolgend werden die *Entites* aus Abbildung 6 (Seite 10) 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

[8] *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*.

4.6 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. Heraus leitet sich eine Gleichheit der Objekte bei gleichen Attribut-Werten ab.

Nachfolgend werden die *Value Objects* aus Abbildung 6 (Seite 10) beschrieben.

Pose

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

5 Programming Principles

5.1 SOLID

5.1.1 Single Responsibility Principle

Jeder Klasse des zugrundeliegenden Codes kann eine definierte und subjketiv kleine Aufgabe zugewiesen werden. Als Beispiel soll die nachfolgende Beschreibung dienen.

Beispiel Klasse `Input`:
Die Aufgabe dieser Klasse ist es, einen Wert vom Typ Klasse `Value` für eine weitere Bearbeitung bereitzustellen.

- Implementiert Methode `getInputUnit()` aus Klasse `Inputable`.
- Erlaubt das Zuweisen eines Objekt-Pointers vom Typ Klasse `Outputable`, von welchem Daten abgefragt werden können.
- Gibt den zugewiesenen Klasse `Outputable`-Pointer zurück (Methode `getInputAddr`).

BESCHREIBUNG! *evtl Bild?? Dann auch mit Verweis auf die jeweiligen Basis-Klassen (URI)*
Link: Controller/Input.h

Neben der genannten semantischen Einschätzung des zugrundeliegenden Codes soll an dieser Stelle folgendes Theorem eingeführt werden:

Maag-Uhlmann-Theorem

Grundsätzlich kann ein Projekt mit der Menge an Objekten des Domänenmodells umgesetzt werden. Je höher das Verhältnis $q = \frac{usedClassCountProject + usedExternPackages}{ObjectCountDomainModel}$, desto eher ist eine feingranularere Aufgabenverteilung an die jeweiligen Klassen zu erwarten. Ein Wert $q = 1$ entspricht der Umsetzung aller Funktionalität innerhalb der Objekte (dann Klassen) des Domänenmodells, hiervon ist abzuraten. In unserem Projekt befinden sich 37 eingesetzte Klassen und Pakete, entgegen stehen 8 Objekte im Domänenmodell. Hieraus folgt: $q = 4.625$.

5.1.2 Open/Closed Principle

Beispiel (siehe Abbildung 7 (Seite 14)): Klasse `SafetyProvider` und Klasse `SafetyRequirement`
Die Klasse Klasse `SafetyProvider` soll sicherstellen, dass der Flug der Drohne sicher durchgeführt werden kann. Hierzu werden verschiedene Anforderungen (Klasse `SafetyProvider`) hinzugefügt. Bei der Klasse `SafetyProvider` handelt es sich um eine *full virtual* Klasse[9]1. Hiervon werden die tatsächlichen Anforderungen abgeleitet.

Link: Domain/SafetyProvider.h
Link: Domain/SafetyProvider.cpp
Link: Domain/SafetyRequirement.h

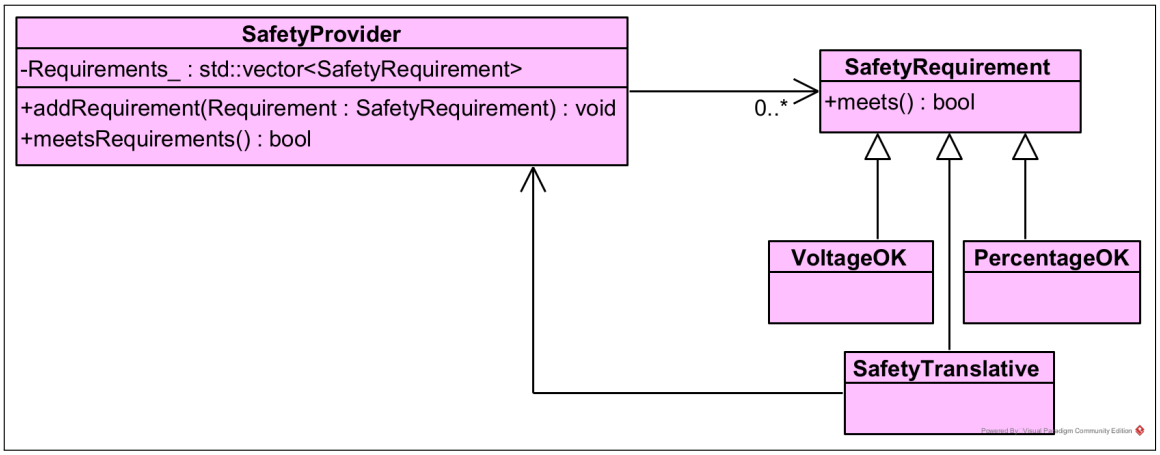


Abbildung 7: Beispiel-Klassendiagramm für das Open/Close-Principle

Die Klassen in Abbildung 7 entsprechen der Anordnung der Vorlesungsfolien „Programming Principles“(Seite 18).

5.1.3 Liskov Substitution Principle

c++ ist kovariant. Dieses Konzept wurde im Code wie folgt angewandt:

Beispiel:
Die in Kapitel 3.0.2 *Brücke* (Seite 7) beschriebene Brücke bedient sich der Kovarianz. Es werden Pointer auf Typen in Attributen gespeichert, welche als von diesen Typen abgeleiteten Klassen Instanziiert werden.

BESCHREIBUNG! *zweites Beispiel?*

5.1.4 Interface Segregation Principle

Beispiel im Code:
Die Klasse Klasse `ControlledOutput` setzt sich unmittelbar aus de beiden *virtual* Klasse (Klasse `Output` und Klasse `Controlable`) zusammen (siehe Abbildung 8 (Seite 15)). Hierdurch wird eine höhere Granularität erreicht.

5.1.5 Dependency Inversion Principle

Beispiel im Code:
Klasse `IMUable` besitzt als Attribut einen Pointer auf Klasse `PoseBuildable`, welch von Klasse `PoseBuilder` implementiert wird (siehe Abbildung 2 (Seite 3)). Eine Instanz der

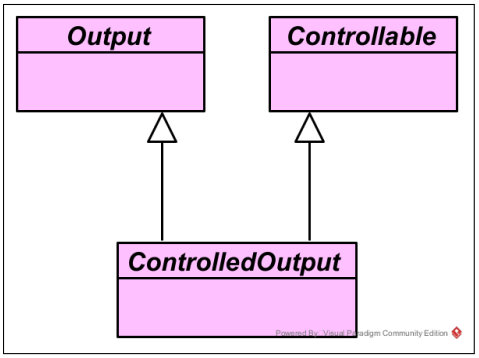


Abbildung 8: Beispiel-Klassendiagramm für das Interface Segregation Principle

Klasse `PoseBuilder` wird von der Instanz der Klasse `MainClass` instanziiert und an die Instanz der Klasse `IMUable` übergeben.

5.2 GRASP

5.2.1 Kopplung

Der Einsatz von *ROS* deutet an dieser Stelle auf eine lose Kopplung hin (siehe Kapitel 3.0.3 *Beobachter* (Seite 7)).[10]

Durch polymorphe Methodenaufrufe liegt eine mäßige Kopplung vor. Dies wurde in der Klasse `ControllerSystem` der Methode `addController` umgesetzt.
Link: Controller/ControllerSystem.h#L28 ff.

Abgesehen von den genannten Punkten ist der zugrundeliegende Code eher stark gekoppelt, da häufig statische Methodenaufrufe sowie Methodendefinitionen verwendet werden.

5.2.2 Kohäsion

An einem Beispiel (siehe Abbildung 9 (Seite 16)) wird der technische Ansatz dargestellt:
Methoden werden mit Attributen assoziiert und anschließend bewertet, inwiefern die Attribute der Klasse interagieren.

5.3 DRY

Dieses Prinzip fördert das Schonen von Ressourcen. Daher wurde dieses Prinzip in großen Teilen des zugrundeliegenden Code angewandt.

[10] vgl. Vorlesungsfolien ‚Programming Principles'(Seite 50)

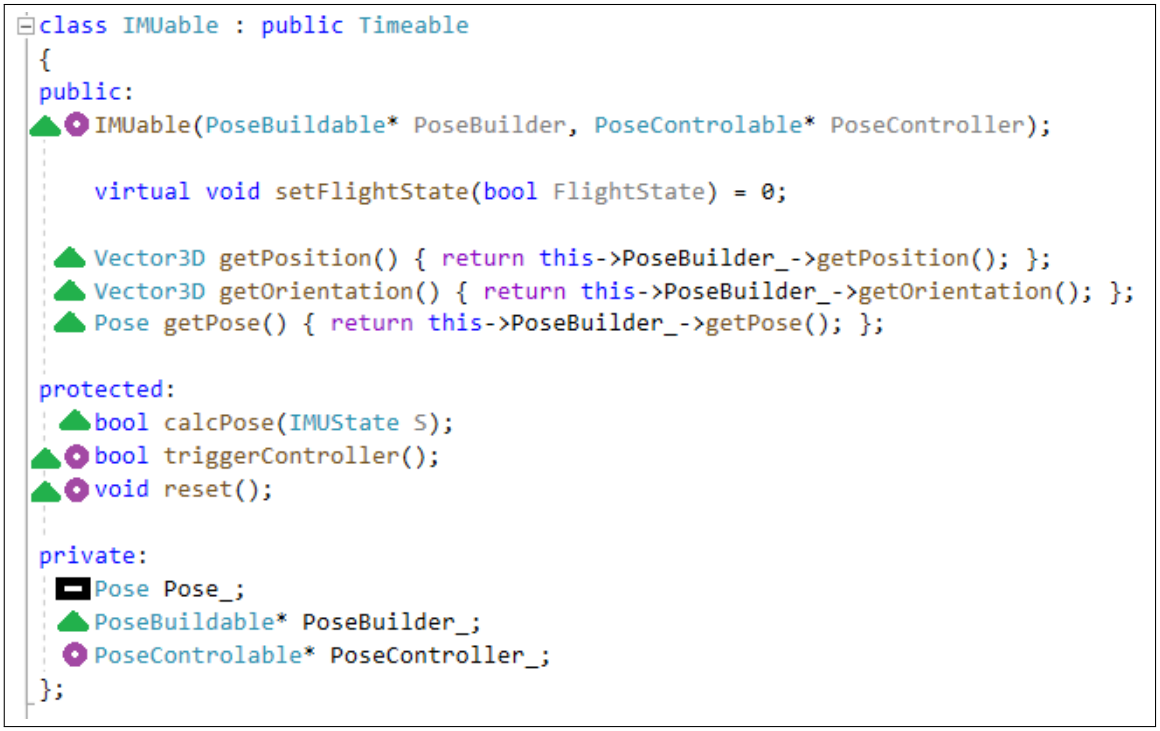


Abbildung 9: Kohäsion am Beispiel der Klasse IMUable

In Abbildung 9 werden Methoden zu Attributen zugeordnet, um den Grad der Kohäsion bestimmen zu können.
Es zeigt sich, dass die Variable `Pose_` nicht innerhalb der Klasse verwendet wird. Somit könnte dieses Attribut entfernt werden.
Die verbleibenden Attribute sind über Methoden miteinander verbunden und bilden somit eine hohe Kohäsion ab.

Sichtbar ist dies beondersbei Vererbung, da bereits vorhandene Funktionalität aus der Basisklasse übernommen wird und keine Neuimplementierung dieser stattfindet. zum Beispiel
Klasse `TimedValue` erbt von Klasse `Value` und Klasse `Timestamp`, wobei nur wenige notwendige Funktionen (Konstruktoren und einfache Arithmetik) hinzugefügt werden.
Link: Domain/TimedValue.h
Link: Domain/Value.h
Link: Domain/Timestamp.h

Negativ Beispiel siehe Kapitel 7.1.1 *Duplicated Code* (Seite 23).

Nachweis

Um die getroffene positiv-Aussage zu bestätigen, sollte an dieser Stelle ein Nachweis mit einem geeigneten *Duplication Checker*-Tool eingebracht werden. Leider war es den Autor:innen nicht möglich, dies mit geeignetem Erfolg umzusetzen. Wir bitten an dieser Stelle um Anerkennung der Bemühungen.

- http://www.ccfinder.net/ccfinderxos.html (nicht ausführbar)
- https://pmd.github.io/latest/pmd_userdocs_cpd.html (zu aufwändig)
- https://github.com/Acumatica/AntiPlagiarism (leider nur c#)

5.4 YAGNI

YAGNI (*You ain't gonna need it*) ist ein Prizip, 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.
An dieser Stelle unterstellen wir eine gute Projektplanung, welche einen Einsatz des YAGNI-Prinzips quasi obsolet macht.

Abweichend hiervon wurde dieses Prinzip im Zusammenhang mit der Anwendung von TDD umgesetzt (siehe Kapitel 6.2 *TDD* (Seite 22)).

5.5 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.

6 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.

6.1 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 tennen. Dies wurde durch verschiedene Projekte innerhalb einer *MS Visual Studio Solution* umgesetzt.

6.1.1 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 selbstsprechend. Wie allgemein wünschenswert: Alle Tests sind erfolgreich (siehe Abbildung 10 (Seite 19)).

- Link: Test_Domain/test.cpp
- Link: Test_DroneController/test.cpp
- Link: Test_Adapter/test.cpp
- Link: Test_Controller/test.cpp
- Link: Test_PlugIn/test.cpp

AAA-Normalform

Der allgemeine Aufbau eines Unit-Tests, wie er auch in diesem Projekt angewandt wurde, wird hier anhand eines Beispiels (siehe Abbildung 11 (Seite 19)) demonstriert.

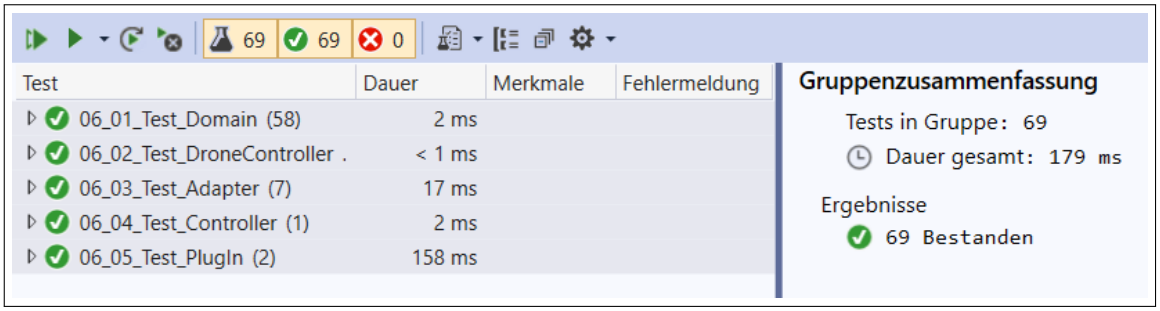


Abbildung 10: Ergebnisse der Unit-Tests

```
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 11: AAA-Normalform an einem Beispiel-Test

oranger Kasten: Arrage-Bereich
rosa Kasten: Act-Bereich
grüner Kasten: Assert-Bereich

Mocks

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 `Transmitable` aufruft.

Link: Test_Controller/test.cpp#L15

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]

6.1.2 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 6.2 *TDD* (Seite 22)).

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`:
Link: Test_Adapter/test.cpp#L110
Link: Test_Adapter/test.cpp#L139
Link: Test_Adapter/test.cpp#L175

Um die Debug-Tätigkeit zu vereinfachen, werden für einen Fehlerfall in aufwändigeren Tests menschenlesbare Informationen über die errechneten Ergebnisse zur Verfügung gestellt[11]. HIerdurch wird das Auffinden der Ursache des Fehlers erleichtert.

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 6.1.1 *Umgesetzte Unit Tests* (Seite 20)).

Professional

Allgemein wurde versucht, die Lesbarkeit der Tests zu gewährleisten, indem kurze und aussagekräftig[12] Tests geschrieben wurden. Hierzu wurden die Abschnitte aus Kapitel 6.1.2 *ATRIP-Regeln* (Seite 20) 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.
Beispiel im Code: Zeilenanzahl > 25 Zeilen
Link: Test_Adapter/test.cpp#L175

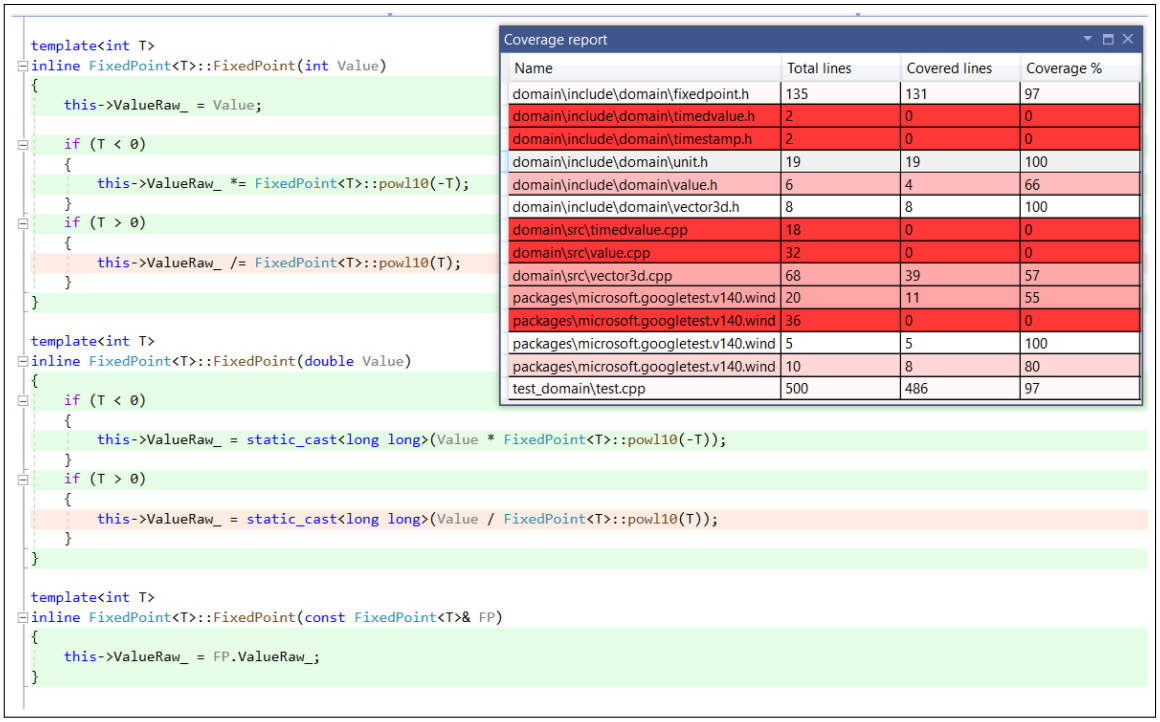
6.1.3 Code Coverage

Um die *Code Coverage* des Codes zu messen, wurde das Tool CPPCoverage eingesetzt. Hierbei handelt es sich um eine Erweiterung der eingesetzten IDE. Die vom Hersteller angebotene Erweiterung *Coverlet* ist nur für die *Enterprise*-Version der IDE freigegeben.

Die Auswertung des Tools beschränkt sich auf *Line Coverage* und beschreibt diese innerhalb der einzelnen Dateien (siehe Abbildung 12 (Seite 22)). Sofern eine abschließende Auswertung gewünscht ist, muss diese von Nutzenden für das gesamte getestete Projekt berechnet werden.

Durchführung der Bestimmung der Testabdeckung auf folgendem Commit:
Commit `83c6ff44f67ce1185056a88f00b23e00833a60fb`

[11] Das eingesetzte Test-Framework zeigt hier lediglich den Speicherinhalt in hexadezimalem Format an.
[12] ‚aussagekräftig‘ ist immer subjektiv



```
template<int T>
inline FixedPoint<T>::FixedPoint(int Value)
{
    this->ValueRaw_ = Value;

    if (T < 0)
    {
        this->ValueRaw_ *= FixedPoint<T>::powl10(-T);
    }
    if (T > 0)
    {
        this->ValueRaw_ /= FixedPoint<T>::powl10(T);
    }
}

template<int T>
inline FixedPoint<T>::FixedPoint(double Value)
{
    if (T < 0)
    {
        this->ValueRaw_ = static_cast<long long>(Value * FixedPoint<T>::powl10(-T));
    }
    if (T > 0)
    {
        this->ValueRaw_ = static_cast<long long>(Value / FixedPoint<T>::powl10(T));
    }
}

template<int T>
inline FixedPoint<T>::FixedPoint(const FixedPoint<T>& FP)
{
    this->ValueRaw_ = FP.ValueRaw_;
}
```

Coverage report

Name	Total lines	Covered lines	Coverage %
domain\include\domain\fixedpoint.h	135	131	97
domain\include\domain\timedvalue.h	2	0	0
domain\include\domain\timestamp.h	2	0	0
domain\include\domain\unit.h	19	19	100
domain\include\domain\value.h	6	4	66
domain\include\domain\vector3d.h	8	8	100
domain\src\timedvalue.cpp	18	0	0
domain\src\value.cpp	32	0	0
domain\src\vector3d.cpp	68	39	57
packages\microsoft.googletest.v140.wind	20	11	55
packages\microsoft.googletest.v140.wind	36	0	0
packages\microsoft.googletest.v140.wind	5	5	100
packages\microsoft.googletest.v140.wind	10	8	80
test_domain\test.cpp	500	486	97

Abbildung 12: Code Coverage Report

Der Report beschreibt den *Coverage*-Bestimmung für das *Domain*-Package. Die Tests sind paketweise getrennt.
Die getesteten Zeilen werden in den Dateien grün, die ungetesteten rot, eingefärbt.
Im Report werden alle von diesem Testlauf berührten Dateien aufgelistet. Die Einfärbung der Dateien im Report ist abhängig von der prozentualen Test-Abdeckung. In Abbildung 12

6.2 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.

Hierbeit 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.

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

7 Refactoring

7.1 Code Smells

7.1.1 Duplicated Code

Wie bereits in Kapitel 5.3 *DRY* (Seite 15) angemerkt, findet sich Code-Duplikate im Code:

zum Beispiel
Commit `edd7726743b9eb67e159112919224d54c54be670`
Klasse `FixedPoint<T>` (mehrfach in verschiedenen Methoden)
zum Beispiel
Methode `operator+(const FixedPoint<TIn>& FP) [const]` (mutable und const Spezifizierer)
Identischer Code ist in nachfolgend genannten Methoden vorhanden.
Link: Domain/FixedPoint.h#L145
Link: Domain/FixedPoint.h#L157

7.1.2 Long Method

Link: PosControl/PoseBuilder.cpp#L68 Commit `73f3400f4866b27200a37be5d3a80fa8394b2c48`
Klasse `PoseBuilder`, Methode `updatePose`

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 Funtionalität umgeändert werden.

7.1.3 Large Class

Link: Domain/FixedPoint.h Commit `73f3400f4866b27200a37be5d3a80fa8394b2c48`
Klasse `FixedPoint<T>`

Um die Funktionalität (Arithmetik) gewährleisten zu können, sind in dieser (aktuell) Klasse 35 Methoden definiert. Nach unserer subjektiven Definition ist eine Klasse ab 15 Methoden eine *Large Class*.

7.1.4 Code Comments

An deiser Stelle sollen lediglich Links aufgezählt werden.

- Link: Domain/TimedValue.cpp#L17
  ToDo-Kommentar, nicht in den Code!

- Link: Domain/TimedValue.cpp#L39
  ToDo-Kommentar, nicht in den Code!
- Link: PosControl/PoseBuilder.cpp#L103
  impliziter ToDo-Kommentar, nicht in den Code!

Informative Comments und damit sinnvoll:

- Link: Domain/Vector3D.cpp#L99
- Link: DroneController/Rotor.cpp#L26

7.1.5 Switch Statement

Link: PosControl/MainClass.cpp#L67 Commit `3c5aeed610150bd19f75241af48a2777cb4c582a`
Klasse `MainClass` Methode `callbackKeys`

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

7.2 Refactoring

7.2.1 Method Extraction

Angewandt auf die in Kapitel 7.1.2 *Long Method* (Seite 23) beschriebene Methode.
Commit `9dd6821a9c63bf1682806b8ab319d6004247d3ca`
Klasse `PoseBuilder` Methode `updatePose(IMUState State)`

Hier waren semantisch gekapselte Aufgaben innerhalb einer Methode vorhanden. Diese wurden in separate Methode ausgelagert und nachfolgend der Aufruf der Use Cases.
vorher:
Link: PosControl/PoseBuilder.cpp#L71

nacherher:
Link: PosControl/PoseBuilder.cpp#L71
Link: PosControl/PoseBuilder.cpp#L94
Link: PosControl/PoseBuilder.cpp#L108

7.2.2 Rename Class

In der Vorlesung wird das Umbenennen einer Methode angesprochen. Ziel ist hier, die Verständlichkeit des Codes zu erhöhen. In dem dieser Programmentwurf zugrundelie-

genden 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 `PosBridge` nach Klasse `MainClass`.
Commit `3c5aeed610150bd19f75241af48a2777cb4c582a`
Link: PosControl/MainClass.cpp

7.2.3 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 zukünfitig zu erleichtern. Hierzu sind diverse, teilweise virtuelle[1], Klassen entstanden.

Commit `2b1bccb0f9b2671ace7fb91b6ed86e2be2284596`

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

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

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