



Hochschule für Technik,
Wirtschaft und Kultur Leipzig

FAKULTÄT INGENIEURWISSENSCHAFTEN

9010 - BACHELORARBEIT

**Architektur eines OCPP-Servers.
Implementierung als Bibliothek,
Framework und Standalone Anwendung.**

<i>Author</i>	Ivan Agibalov
<i>Betreuer</i>	Prof. Dr.-Ing. Andreas Pretschner
<i>2. Betreuer</i>	Andre Vieweg M. Sc

10. August 2022

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Tabellenverzeichnis	iii
1 Motivation	1
2 Grundlagen	2
2.1 OOP Design Patterns	2
2.1.1 Observer	2
2.1.2 Proxy	2
2.1.3 TemplateMethod	3
2.1.4 Builder	3
2.1.5 Facade	3
2.2 Architekturen	3
2.2.1 Clean Architectur	3
2.2.2 Model-View Architekturen	3
2.3 Software Testing	3
2.3.1 Testing Pyramide	3
2.3.2 Unit Tests	4
2.3.3 Integration Tests	4
2.3.4 SystemTests	4
2.3.5 UI Tests	5
2.3.6 Manual Tests	5
2.4 SOLID	5
2.4.1 single-responsibility principle	5
2.4.2 open-closed principle	5
2.4.3 Liskov substitution principle	5
2.4.4 interface segregation principle	5
2.4.5 dependency inversion principle	5
2.5 GRASP	5
2.6 OOP Principles	5
2.6.1 Abstraction	5
2.6.2 Encapsulation	6
2.6.3 Inheritance	6
2.6.4 Polymorphism	6

3	Software Architektur	7
3.1	Was ist eine Software Architektur	7
3.2	Ziele der Software Architektur	7
3.3	Technische Schulden	8
3.4	Qualität und Kosten der Software	9
3.5	Wichtigkeit der Testbarkeit der Software	10
3.6	Technische Umsetzung der Software Architektur	11
3.7	Abhängigkeiten im Programm	13
3.7.1	Port-Adapter-Controller	13
3.7.2	Controller-Dispatcher-UseCase-Interactor	14
3.7.3	Erstellen der Struktur	15
3.7.4	Utility Controllers	16
3.8	Datenfluss im Programm	17
3.8.1	Auswahl der Datenbank	21
3.8.2	Dependency Rule	21
3.8.3	Unterschied zu Layered Architektur	21
3.8.4	Implementierung der Testbarkeit	21
3.8.5	Dependency Injection	21
4	Aufgabenstellung	23
4.1	Anforderungen an den Standalone Server	23
4.2	Anforderungen an das Testframework	24
4.3	Anforderungen an ERK Automatisierungstool	24
5	Lösung der Aufgabe	24
6	Gewünschtes Interfaces	25
7	Implementierung	26
7.1	Achitecture des Frameworks	26
7.1.1	Ports	26
7.1.2	Adapters	26
7.1.3	Controllers	26
7.1.4	Dispatcher	27
7.1.5	UseCases	27
7.1.6	Interactors	27
7.1.7	Domain	27

7.2	Zugriff auf das Testframework	27
7.3	Testbeispiel	28
8	Übersichtsdiagramm	29
9	Conclusion	29
	Bibliography	30
	Appendix	31
A	Hello World Example	31
B	Flow Chart Example	31
C	Sub-figures Example	32

Abbildungsverzeichnis

1	CI/CD Pipeline	1
2	UML Observer	2
3	Testing Pyramide	4
4	Vergleich einer guten und einer schlechten Softwarearchitektur	10
5	some Caption	11
6	Klassendiagramm PAC	13
7	Klassendiagramm Controller-Dispatcher-UseCase-Interactor	14
8	Ablaufdiagramm Erstellen der Struktur	15
9	Klassendiagramm mit Utility Controllers	16
10	some Caption	17
11	Sequencediagramm vom Datenfluss “1” Blau	18
12	Datenfluss und Quellcode Abhängigkeiten	21
13	Entkopplung der Abhängigkeiten	22
14	Streamline results	32

Tabellenverzeichnis

1 Motivation

Die Entwicklung eines Softwaresystems ist ein wiederholender Prozess, der sich in mehreren Phasen unterteilen lässt. Alle Phasen beeinflussen sich gegenseitig, sodass man sie nicht unabhängig voneinander betrachten kann.

Die Abbildung 1 zeigt eine mögliche Aufteilung in Phasen der Entwicklung.

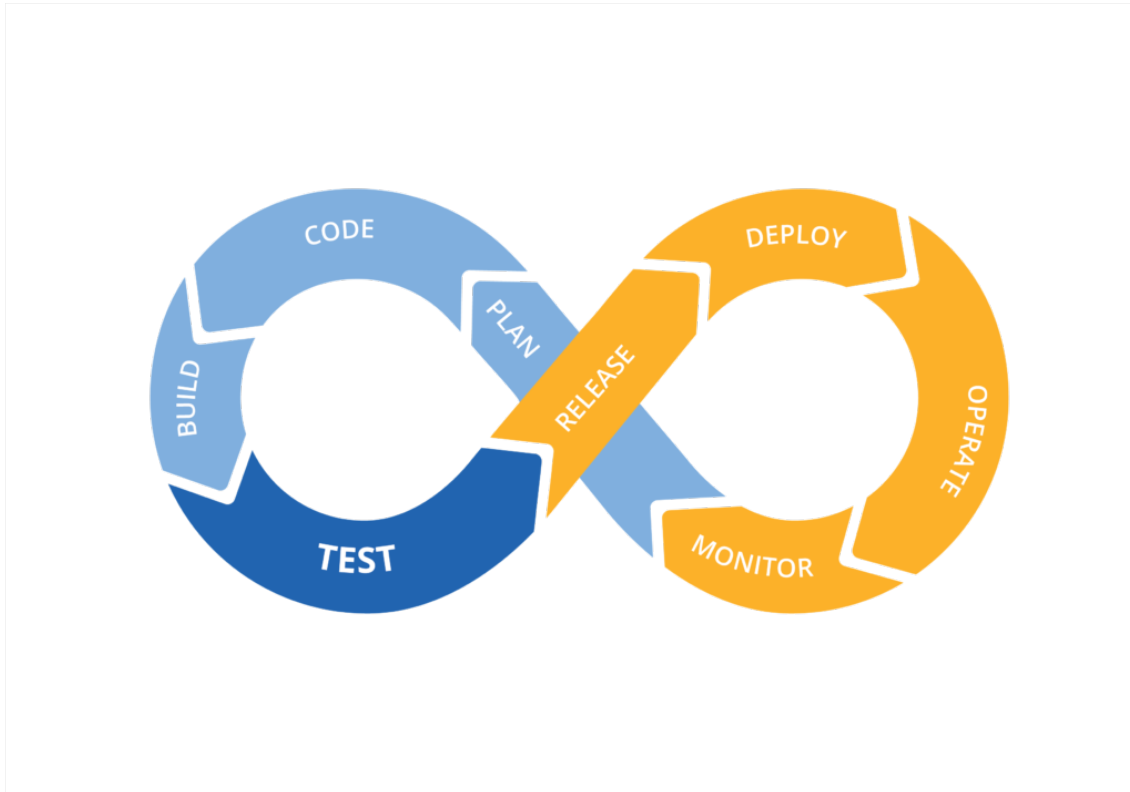


Abbildung 1: CI/CD Pipeline ¹

Man ist daran interessiert, die Gesamtzeit des Zyklus so klein wie möglich zu halten, denn somit können die neuen Funktionalität schneller von Kunden benutzt werden und die Bugs werden schneller eliminiert.

Im Groben kann man die Entwicklung in zwei Teilen teilen: Bevor die neue Version der Software freigegeben wird und nach der Freigabe der neuen Version. Die Phasen nach der Freigabe der neuen Version lassen sich fast vollständig automatisieren und verbrauchen dementsprechend nicht viel Ressourcen ab einem gewissen Moment. Der größte Anteil an Ressourcen wird in die ersten vier Phasen (Plan, Code, Build, Test) verbraucht, denn diese Aufgaben lassen sich schlecht bis gar nicht automatisieren.

Ein hoher Anteil an manuellen Prozessen, wie z.B. Testen, Erstellen, führt in der Softwareentwicklung zu längeren Zykluszeiten. Es ist bereits am Anfang des Projektes von Interesse die Gedanken daran zu machen, wie man so viel wie möglich automatisiert.

In dieser Arbeit werden Entscheidungen erläutert, welche bereits in den Phasen "Plan" und "Code" getroffen werden können. Das Ziel ist die Gesamtqualität der Software zu verbessern bei gleichbleibendem oder geringerem personellen Aufwand. Als Beispiel dient die Entwicklung eines OCPP Servers.

¹<https://blog.itil.org/2016/07/wort-zum-montag-cd-continous-delivery/>

2 Grundlagen

SomeIntroducation

2.1 OOP Design Patterns

some Introduction

2.1.1 Observer

Das OOP Design Pattern **Observer** ermöglicht dynamische Verbindungen zwischen den einzelnen Objekten im Programm, um über die geschehenen Ereignisse im Programm alle Interessenten zu informieren.

Das Pattern besteht aus 2 Teilen: **Publisher** und **Observers** oder **Subscribers**

Subscribers können bestimmte Events des **Publishers** abonnieren und deabonnieren. Der **Publisher** informiert alle auf das geschehene Event abonnierten **Subscribers**, bzw. wenn es auftritt. Den **Subscribers** kann im Falle des Eintretens des Events ein gewisses Verhalten vorgegeben werden.

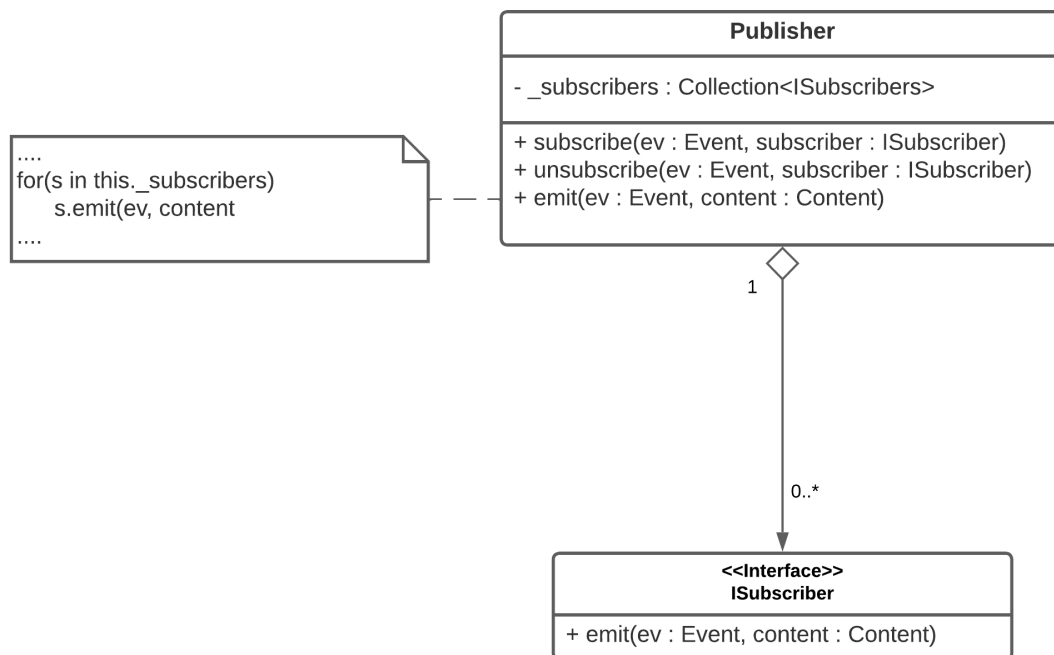


Abbildung 2: Klassendiagramm Observer ²

2.1.2 Proxy

Das OOP Design Pattern **Proxy** ermöglicht die Aufrufe von bestimmten Objekten zu empfangen und ein gewisses Verhalten vor sowie nach dem eigentlichen Aufruf zu definieren.

²Eigene Quelle

2.1.3 TemplateMethod

Das OOP Design Pattern **Template Method** ermöglicht den allgemeinen Ablauf in Form von einzelnen Schritten zu definieren. Einzelne Schritte können dabei bei der Implementierung neu definiert werden, um das gewünschte Verhalten festzulegen.

2.1.4 Builder

Das OOP Design Pattern **Builder** ermöglicht das Erstellen von komplexen, zusammengesetzten Objekten in einzelne einfache Schritte zu zerlegen

2.1.5 Facade

Das OOP Design Pattern **Facade** ermöglicht für eine komplexe Klasse, die aus vielen Methoden besteht, eine einfachere Klasse zu erstellen, die nur die notwendigen Methoden der komplexeren Klasse besitzt, ohne das Verhalten zu verändern.

2.2 Architecturen

2.2.1 Clean Architectur

some info about clean architecture

2.2.2 Model-View Architekturen

lol kek cheburek

2.3 Software Testing

Jede Software hat im Laufe der Zeit sehr viele Änderungen. Die erste Version einer Software kann beispielsweise mit einer Version der selben Software nach 10 Jahren, wenig bis keine Gemeinsamkeiten besitzen. Jede Änderung des Quellcodes ist ein Risiko für Softwareentwickler, da immer die Gefahr besteht versehentlich funktionierenden Code zu beschädigen.

Um das Risiko auf Defekte zu minimieren, können automatisierte Tests verwendet werden. Je nach Ausführung stellen diese fest, ob das bestehende Verhalten noch dem Sollverhalten entspricht. Hierbei wird nur das Verhalten geprüft, welches mit den entsprechenden Tests abgedeckt wurde.

Es gibt mehrere Typen von automatisierten Tests, die im Folgenden beschrieben werden.

2.3.1 Testing Pyramide

Some text to testing pyramid

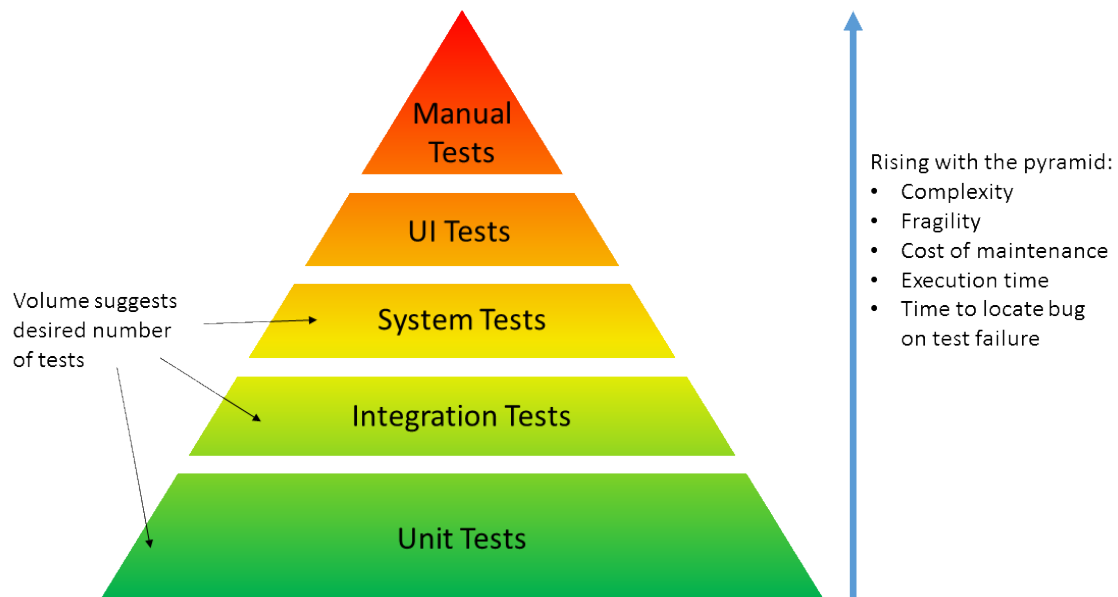


Abbildung 3: Caption written below figure ³

2.3.2 Unit Tests

Unit Tests überprüfen, ob die kleinsten Module (meistens einzelne Funktionen und Objekte), wie gewünscht funktionieren.

Alle Module, welche das zu testende Objekt verwendet, werden während der Ausführung von Unit Tests manipuliert. Auf diese Weise können diverse Situationen simuliert werden.

Unit Tests repräsentieren die Mehrzahl an Tests in einem Projekt. Schlägt ein Unit Test fehl, kann der Fehler im Programm sofort lokalisiert und dadurch die Dauer der Fehlersuche minimiert werden.

2.3.3 Integration Tests

Integration Tests stellen fest, ob die zusammengesetzte Module, Komponenten oder Klassen, welche die Unit Tests bestanden haben, wie gewünscht funktionieren.

Alle anderen verwendeten Module werden analog zu den Unit Tests manipuliert.

Die Anzahl an Integration Tests in einem Projekt ist geringer als Anzahl an Unit Tests. Auf der einen Seite ist ein Integration Tests größer als ein Unit Test. Auf der anderen Seite ist die Fehlersuche im Programm bei negativem Testergebnis deutlich komplexer und zeitintensiver, als bei Unit Tests.

Der Vorteil von Integrationstests liegt darin, dass mit ihrer Hilfe ein großer Teil des Codes mit Tests abgedeckt werden kann.

2.3.4 SystemTests

Bei Systemtests wird diese wie ein Produktivsystem gestartet und entsprechend getestet.

³<https://www.cqse.eu/de/news/blog/junit3-migration/>

2.3.5 UI Tests

Während der Ausführung von UI Tests (User Interface Tests) wird die Eingabe des Benutzers an der Oberfläche simuliert. Jeder UI Test braucht eine Laufzeitumgebung, um den Benutzer zu simulieren, ein UI Test wird nur mittels der Ausgabe an der Oberfläche validiert.

2.3.6 Manual Tests

some Info about manual tests

2.4 SOLID

some info about solid

2.4.1 single-responsibility principle

SRP

2.4.2 open-closed principle

OCP

2.4.3 Liskov substitution principle

LSP

2.4.4 interface segregation principle

ISP

2.4.5 dependency inversion principle

DIP

2.5 GRASP

GRASP steht für *General Responsibility Assignment Software Patterns* und beschreiben die Grundprinzipien für die Aufteilung der Verantwortung zwischen Klassen. [Rod]

2.6 OOP Principles

some info about oop principles

2.6.1 Abstraction

some about abstractions

2.6.2 Encapsulation

some about Encapsulation

2.6.3 Inheritance

some about Inheritance

2.6.4 Polymorphism

some about Polymorphism

3 Software Architektur

3.1 Was ist eine Software Architektur

Bevor man anfängt über die Software Architektur zu reden, muss man sie erstmal definieren. Es gibt keine einheitliche Definition einer Software Architektur. Verschiedene Autoren definieren es auch unterschiedlich.

Robert Martin definiert es als ein Gegenstand mit bestimmten Eigenschaften zu definieren. *Softwarearchitektur ist die Gestalt des Systems erstellt von derjenigen, die das entwickeln. Die Form dieser Gestalt ist das Aufteilen des Systems in Komponenten, die Anordnung (Arrangement) dieser Komponenten und die Wege, wie diese Komponente miteinander kommunizieren* [Mar18, S. 136]

Ralph Jonson definiert Software Architektur aus Sicht eines Projektes. *Architektur besteht aus den Entscheidungen, die man sich wünscht so früh wie möglich in einem Projekt zu treffen* [Fowa]

In dem ersten Teil des Kapitels wird die Softwarearchitektur aus Sicht eines Projektes betrachtet, indem es kurz beschrieben wird, welche Auswirkungen eine gute und eine schlechte Architektur auf ein Projekt haben kann.

In dem 2. Teil des Kapitels, wird die Architektur des OCPP Backend Servers beschrieben

- Aufteilung des Programms in einzelne Teile
- Testbarkeit und Erweiterbarkeit einzelner Teile des Programms
- Kommunikation zwischen den einzelnen Teilen

3.2 Ziele der Software Architektur

Das Ziel der Software Architektur ist das Minimieren der menschlichen Ressourcen, die benötigt werden um ein System zu entwickeln und zu unterstützen.[Mar18, S. 5]

Diese Aussage lässt sich sehr einfach überprüfen, indem man feststellt, ob jede neue Anforderungen an der Software mehr Ressourcen verbraucht als die vorherigen.

Das Ziel der Softwarearchitektur ist so viel Entscheidungen wie möglich so spät wie möglich zu treffen [Mar18, S. 136]

Beispiele für solche Entscheidungen wären:

- Datenbanksystem
- Transferprotokoll zu der Benutzeroberfläche (z.B. HTTP oder WS) falls vorhanden
- Wie und wo die Loggingdaten gespeichert werden (in einer Datei, Datenbank oder externe Server)

Auch die Tätigkeiten, die nicht mit Programmieren direkt zu tun haben, werden von den Entscheidungen in der Softwarearchitektur betroffen

- Deployment (Aufsetzung) der Software.
- Maintenance (Unterstützung) der Software.

Deployment der Software beinhaltet die Kosten die durch das Aufsetzen der neuen Version der Software entstehen.

Maintenance der Software beinhaltet die Kosten, die nach dem Beenden der Entwicklung bei kleineren Erweiterungen und Änderungen des Systems entstehen.

3.3 Technische Schulden

Bei den Änderungen oder Erweiterungen eines Systems oft entsteht ein Overhead, das durch die “Unsaubarkeit” des bestehenden Programms verursacht wird.

Dieses Overhead wird als technische Schulden (en. : Technical Debts) bezeichnet.

Die technischen Schulden entstehen dadurch, dass bei der Entwicklung eines Teiles des Systems wurde von den Entwicklungsteam weniger Zeit investiert um die nicht gewinnbringende Aufgaben zu erledigen. Beispiele für solche Tätigkeiten wären:

- Unittests
- Dokumentieren
- Code Review

Beispiele für Technische Schulden wären:

- Alte Funktionalitäten funktionieren nach der Änderung nicht mehr
- Aufdeckung eines Bugs erst nach einer gewissen Zeit in Produktionsversion der Software
- Implementieren der neuen Funktionalitäten verbraucht deutlich mehr Zeit

Eine klare Struktur der Software reduziert die Menge an technischen Schulden, die die Weiterentwicklung in der Zukunft verlangsamen.

Die Softwareentwickler können die ankommenden Aufgaben erledigen

- man hat bereits Vorgaben wie die Kommunikationswege zwischen den Modulen ist
- wie die Module benannt werden sollen
- an welchen Stellen das Modul in das System hinzugefügt werden soll
- die Menge an durch den Zufall entstehenden Bugs in anderen Teilen des Programms ist minimal

Durch die bereits definierten Kommunikationswege zwischen den Modulen, muss weniger Dokumentation geschrieben werden. Mit weniger Dokumentation findet man schneller die gesuchten Informationen.

Durch die einheitliche Bezeichnung der Teile des Modules kann man allein aus dem Namen des Modules seine Aufgaben ableiten.

Daher ist es vom Vorteil bevor man mit der Umsetzung des Softwaresystems anfängt, die oben genannten Aufgaben zu lösen, denn mit zunehmender Lebenszeit der Software nimmt die Änderungszeit zu.

Somit lassen sich die vorhandenen Ressourcen effizienter einsetzen.

3.4 Qualität und Kosten der Software

Am Anfang jedes neuen Projektes in der Softwareentwicklung muss die Entscheidung getroffen werden, wie qualitativ gut die Software am Ende sein soll. Damit sind die Eigenschaften/Funktionalitäten der Software gemeint, die für die Benutzer irrelevant sind, jedoch eine sehr große Bedeutung für das Entwicklungsteam haben.

Wenn man eine qualitativ gute Software hat, ergeben sich unter anderem folgende Vorteile:

- Bugs können schneller lokalisiert und beseitigt werden
- Neue Funktionalitäten können mit weniger Aufwand umgesetzt werden
- Die Änderungen der Funktionalitäten können schneller umgesetzt werden
- Die Wahrscheinlichkeit bestehende Funktionalitäten ungewollt zu ändern verringert sich
- Die Einarbeitungszeit von neuen Teammitgliedern verkürzt sich

Alle diese Vorteile hat man nicht kostenlos, denn dafür muss man auch Zeit investieren indem man:

- Regelmäßig die Software refactored
- Code Qualität überprüft
- Code Reviews durchführt
- Automatisierte Tests schreibt (Unit-, Integration- und Systemtests)
- Dokumentation aktuell hält
- Die technischen Schulden gering hält

Nicht in jedem Projekt ist das Umsetzen von oben genannten Eigenschaften möglich, denn man hat nicht genug Zeit oder das Budget ist zu klein dafür. Man kann aber gewisse Kriterien setzen um mit deren Hilfe bessere Entscheidung zu treffen:

- Wann soll die MVP¹ vorhanden sein.
- Wie viele Ressourcen man zur Verfügung hat
- Wie wahrscheinlich sind die Änderungen und Erweiterungen der Software
- Wie kritisch verschiedene Probleme und Ausfälle der Software sind

Auf der unteren Darstellung sieht man, dass auf längere Distanz eine gute Softwarearchitektur deutlich mehr Funktionalitäten besitzt als eine Software mit schlechter Architektur. Jedoch es gibt ein Zeitintervall, in dem die schlechtere Software besser da steht.

¹Minimum viable product

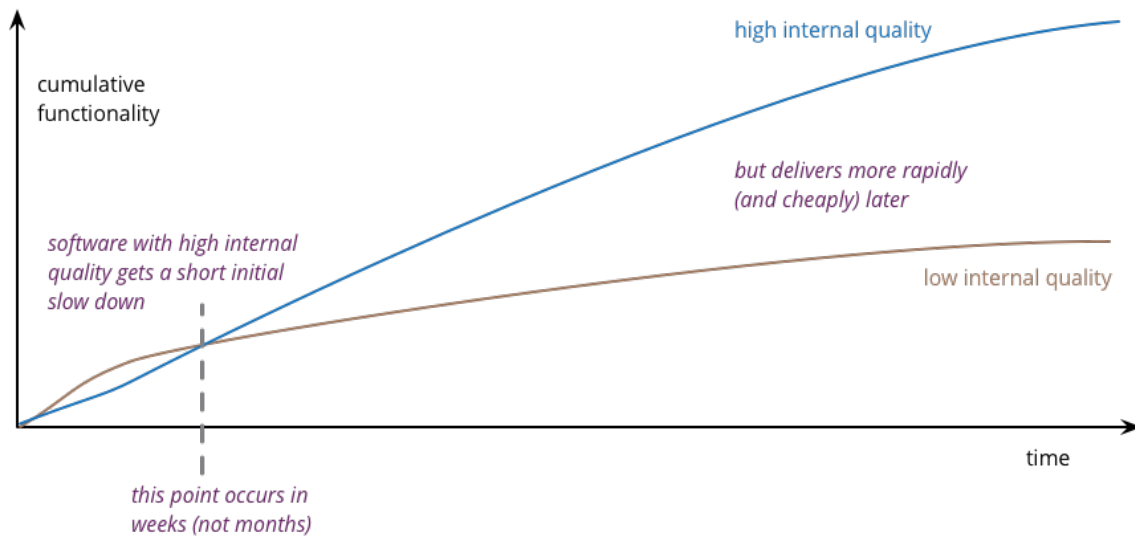


Abbildung 4: Vergleich einer guten und einer schlechten Softwarearchitektur ⁴

Diese Eigenschaft muss man immer beim Projektbeginn beachten, denn es wäre Zeitaufwändig für ein Studiumprojekt, für das man evtl. nur eine Woche Zeit hat, um eine komplexe Architektur zu implementieren, die ohne jeglichen Funktionalitäten mehrere Wochen gebrauchen wird.

Wenn man aber ein Projekt hat, das regelmäßig weiterentwickeln wird, ist es vom Vorteil gleich eine gute Architektur umzusetzen.

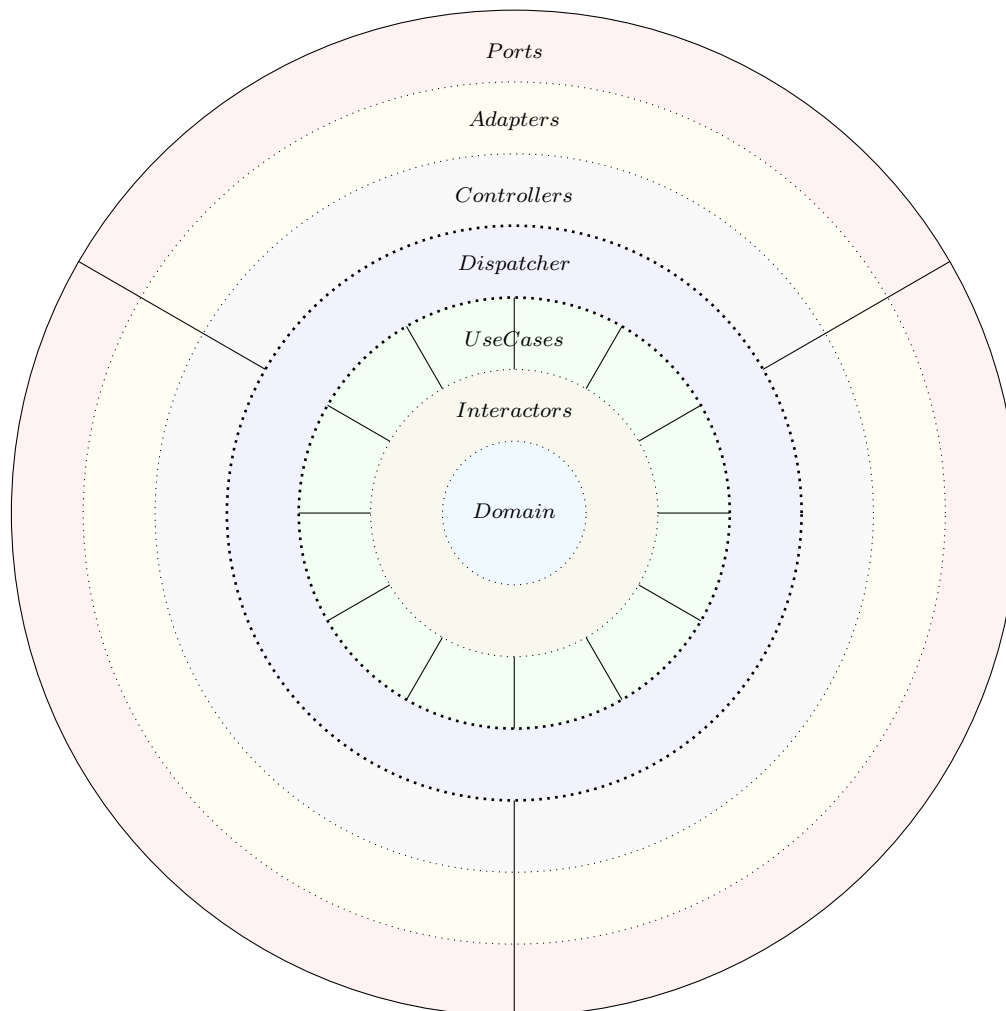
3.5 Wichtigkeit der Testbarkeit der Software

Jedes Teil der Software wird in seinem Lebenszyklus mehrmals geändert. Um die Funktionalität der neuen Version zu verifizieren, muss sie getestet werden. Es ist von Interesse diese Aufgabe zu automatisieren. Wie in den früheren Kapiteln bereits beschrieben wurde, am schnellsten findet man die Bugs, falls vorhanden, mit Unittests. Bei den Unittests müssen die Module (z.B. einzelne Klassen in Falle von OOP Sprachen) in verschiedenen Umgebungen überprüft werden. Das heißt, dass die Zustände von benutzten Modulen müssen einfach zu simulieren sein. Dies erfordert eine Planung der Softwarearchitektur im Voraus, um diese Eigenschaft zu implementieren um im Laufe der Entwicklung, Zeit durch automatisierte Tests zu sparen.

⁴<https://martinfowler.com/articles/is-quality-worth-cost.html>

3.6 Technische Umsetzung der Software Architektur

Der OCPP Backend Server wurde anhand “Clean Architecture” projektiert und entsprechend umgesetzt. Im Kapitel werden die benutzten Schichten beschrieben, welche Eigenschaften sie besitzen und wie sie miteinander verbunden sind. Auch wird kurz gezeigt wie es mit der Testbarkeit auf allen Niveaus (Unit, Integration und Systemtests), Änderbarkeit und Erweiterbarkeit der Software ist.



—— Teile des Programms wissen nichts voneinander

Abbildung 5: Darstellung der umgesetzten Architektur als **Clean Architecture** mit 7 Schichten. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms ⁵

Beschreibung der Darstellung: Jede Komponente bringt in das gesamt Programm folgende Teile:

- Port - z.B. WebSocket Server aufzubauen
- Adapter - umwandeln der ankommenden Nachrichten bzw. Ereignisse in die Typen definierten im Domain
- Controller - definiert alle Tätigkeiten, die die Komponente machen könnte

⁵Eigene Quelle

-
- UseCase - definiert den Ablauf an Tätigkeiten (Interactoren) beim Geschehen eines Ereignisses
 - Interactors - Hülle für alle definierten Tätigkeiten im Controller
 - Domain - definiert Typen benutzten der Komponente und deren Basic Verhalten, auch die Ereignisse die vom Dispatcher verteilt werden

3.7 Abhängigkeiten im Programm

Die Architektur lässt sich in zwei wesentlichen Teilen zerlegen

- Anbindung an Infrastruktur um das Programm (Port - Adapter - Controller)
- Innere Logik des Programms (Controller - Dispatcher - UseCase - Interactor)

Beispiele für die Infrastruktur sind: Datenbank, Persistenz, Schnittstellen (HTTP, USB usw)

Bei solcher Aufteilung ergeben sich folgende Vorteile:

- Innere Logik des Programms lässt sich mittels Integrationstests unabhängig von Schnittstellen abdecken. Damit ist die Laufzeit von jedem einzelnen Test ohne realen Schnittstellen ist schneller als mit realen Schnittstellen und man hat das gleiche Ergebnis bezüglich des Verhaltens des Programms.
- Die Innere Logik ist nicht an Schnittstellen gebunden, somit können alle Schnittstellen mit wenig Aufwand getauscht werden.

3.7.1 Port-Adapter-Controller

Am nächsten zu der **PAC**⁶ Struktur ist das Pattern **MVP**⁷. Dabei die Aufgaben von **View** entsprechen den Aufgaben von **Port**. Die Aufgaben von **Presenter** entsprechen den Aufgaben **Adapter**. Und die Aufgaben von dem Rest des Programms inklusiv **Controller** entsprechen den Aufgaben von **Model**.

Der Unterschied zu **MVP** besteht darin, dass **MVP** im klassischen Sinne nur für die Benutzeroberflächen gedacht ist, während in der hier beschriebenen Umsetzung wird es für alle Schnittstellen benutzt. **MVP** Architektur übernimmt in der gesamten Application eine zentrale Stelle und ist nur einmal zu treffen. **PAC** ist nur ein Teil der gesamten Application, wird an mehreren Stellen unterschiedlich benutzt und beschreibt nicht die Architektur der Application.

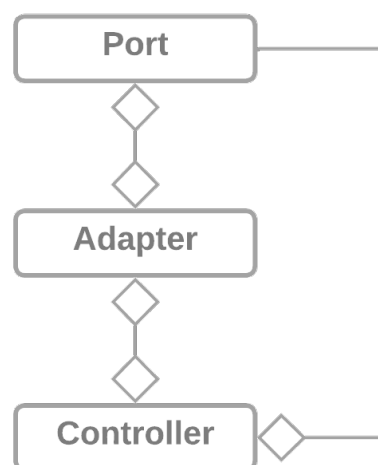


Abbildung 6: Klassendiagramm PAC ⁸

⁶Port-Adapter-Controller

⁷Model-View-Presenter

⁸Eigene Quelle

3.7.2 Controller-Dispatcher-UseCase-Interactor

In dem Teil wird die komplette innere Logik der Application beschrieben. Dafür ist nur **Controller** notwendig. D.h. beim Geschehen eines Ereignisses wird eine (oder mehrere) Method(en) in den jeweiligen Controllern aufgerufen, die das Ereignis entsprechend abarbeiten. Dabei hat man folgendes Problem: jeder **Controller** übernimmt mehrere Aufgaben (z.B. Kontrollieren von **Port** und **Adapter** und enthält Anwendungslogik) d.h. keine **SRP**

Eine mögliche Lösung wäre das Separieren von Kontrollieren von **Port** und **Adapter** und Anwendungslogik in 2 verschiedenen Teile. Die Anwendungslogik heißt **UseCase**.

Bei dieser Aufteilung besteht das Problem, dass beim Geschehen eines Ereignisses im **Controller** muss dieses Ereignis an das richtige **UseCase** zugeordnet werden. D.h. **Controller** besitzt eine weitere Verantwortlichkeit die sich in ein anderes Teil verschieben lässt. Dieses Teil heißt **Dispatcher**, dessen Aufgabe ist das Informieren aller daran Interessierten **UseCases** beim Geschehen eines Ereignisses.

Jedes **UseCase** kann mehrere aufeinander folgende Aufgaben machen. Alle Aufgaben müssen gleiche Funktionalitäten besitzen, z.B:

- der Anfang und das Ende in Logs aufzeichnen.
- Nach einer bestimmten Zeit gestoppt werden.

D.h. man braucht eine "Hülle" um jeder Methode - **Interactor**

Wenn man das alles zusammen entsteht folgendes Klassendiagramm:

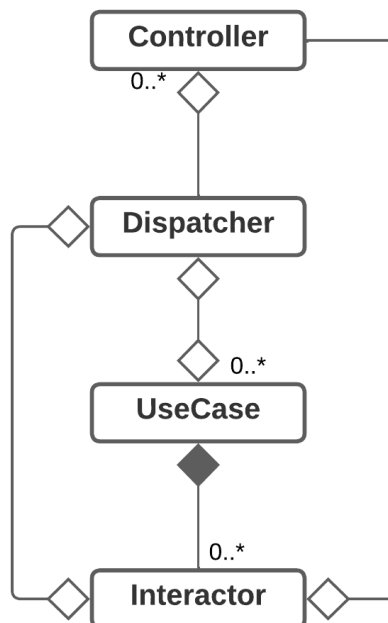


Abbildung 7: Klassendiagramm Controller-Dispatcher-UseCase-Interactor ⁹

⁹Eigene Quelle

3.7.3 Erstellen der Struktur

In den Kapiteln 3.7.2 und 3.7.1 werden die fertigen Strukturen beschrieben, diese Strukturen müssen am Anfang des Programms erstellt und miteinander verbunden werden.

Das Erstellen von der Struktur findet im **Main** statt und lässt sich in drei Schritte aufteilen:

- Erstellen alle Instanzen
- Verknüpfen alle Instanzen miteinander
- Starten alle Instanzen

Das Erstellen aller Instanzen lässt sich in zwei weitere Schritte aufteilen, die bedingt voneinander abhängen.

- Core (Controllers + Dispatcher + UseCases + Interactors)
- Schnittstellen (Port + Adapter + Controller)

Damit auch Integrationstests für den kompletten Core und jede Schnittstellen möglich sind, wäre es sinnvoll, dass beide Schritte explicit ausgeführt werden.

Ein möglicher Ablauf wäre:

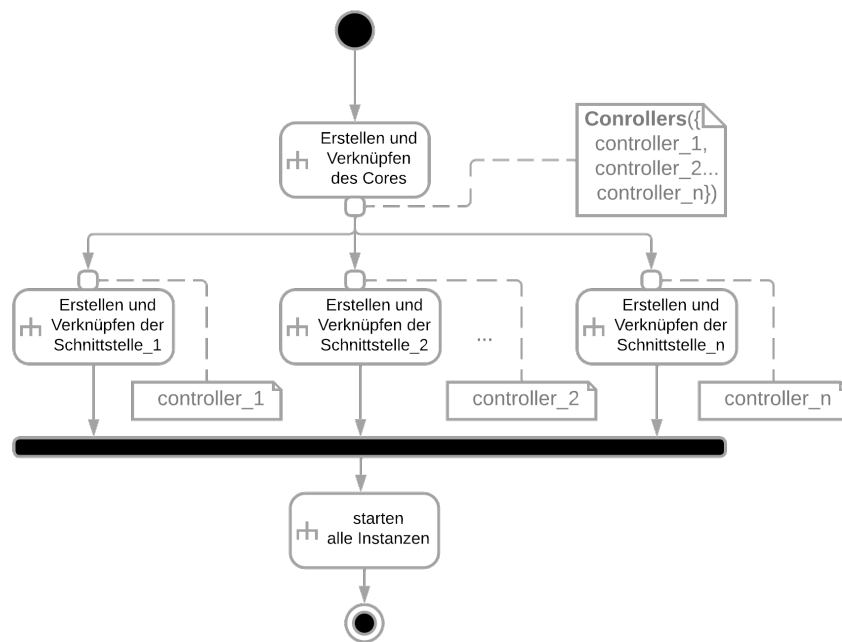


Abbildung 8: Ablaufdiagramm Erstellen der Struktur ¹⁰

Mit diesem Ablauf können mehrere **Mains** erstellt werden, die verschiedene Anwendungen für verschiedene Zwecke erstellen. Z.B. ein Framework braucht keine realen Anknüpfungen an die Infrastruktur (z.B. Datenbank) im Vergleich zu Standalone Anwendung. Oder es können verschiedene **Mains** für verschiedene Datenbanken.

¹⁰Eigene Quelle

3.7.4 Utility Controllers

In jeder Anwendung gibt es Teile bzw. Module, die aus allen Orten erreichbar sein sollen (z.B. Logger Controller oder Datum Controller).

Es gibt folgende Möglichkeiten es umzusetzen:

- Globale Objekte bzw. Instanzen (z.B. OOP Design Pattern **Singleton**)
- Initialisierung von den Entsprechenden Instanzen und Übergeben in dem Konstruktor (OOP Sprachen) oder mittels einem Setters

Bei der ersten Implementierung hat man das Problem, dass die benutzten Utility Controllers nicht ersetzbar sind. D.h. man kann dann alle Module sehr schwer mit Unittests abdecken, denn es werden immer auch die Utility Controllers mitgetestet. Ein weiteres Problem besteht darin, dass alle Utility Controllers eine Infrastruktur benutzen. (z.B. Logs in dem Datenbank speichern). Jeder Test wird dadurch deutlich länger laufen, als es sein könnte. Da es auch reele Infrastruktur sein wird, wird es die Parallelisierung des Tests deutlich erschwert, denn der Zustand der benutzten Infrastruktur durch mehrere unabhängig voneinander laufenden Tests geändert wird.

Bei der zweiten Möglichkeit müssen alle Utility Controllers entweder bei der Initialisierung im Konstruktor der jeweiligen Instanzen übergeben, dies führt zu einer längeren Parameterliste, die die Lesbarkeit des Codes erschwert oder mittels einer anderen Methode der Instanz, dies führt dazu, dass der Aufruf der Methode von dem Softwareentwickler vergessen werden kann.

Das Problem lässt sich zum Beispiel durch OOP design pattern **Factory**, das die Utility Controllers bereits enthält und bei der Initialisierung entsprechend in Konstruktor übergibt. Alle Module bzw. Klassen lassen sich somit auch mit Unittests abdecken, da die Utility Controllers entsprechend gemockt werden können.

In der Abbildung 9 ist die Verbindung jedes **Controllers** mit den Utility Controllers als Klassendiagramm dargestellt.

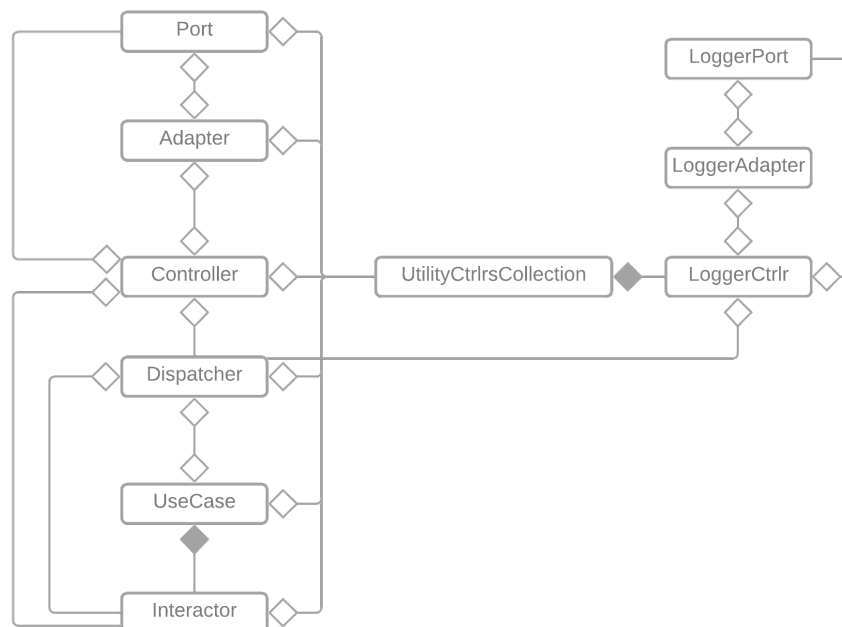


Abbildung 9: Klassendiagramm mit Utility Controllers ¹¹

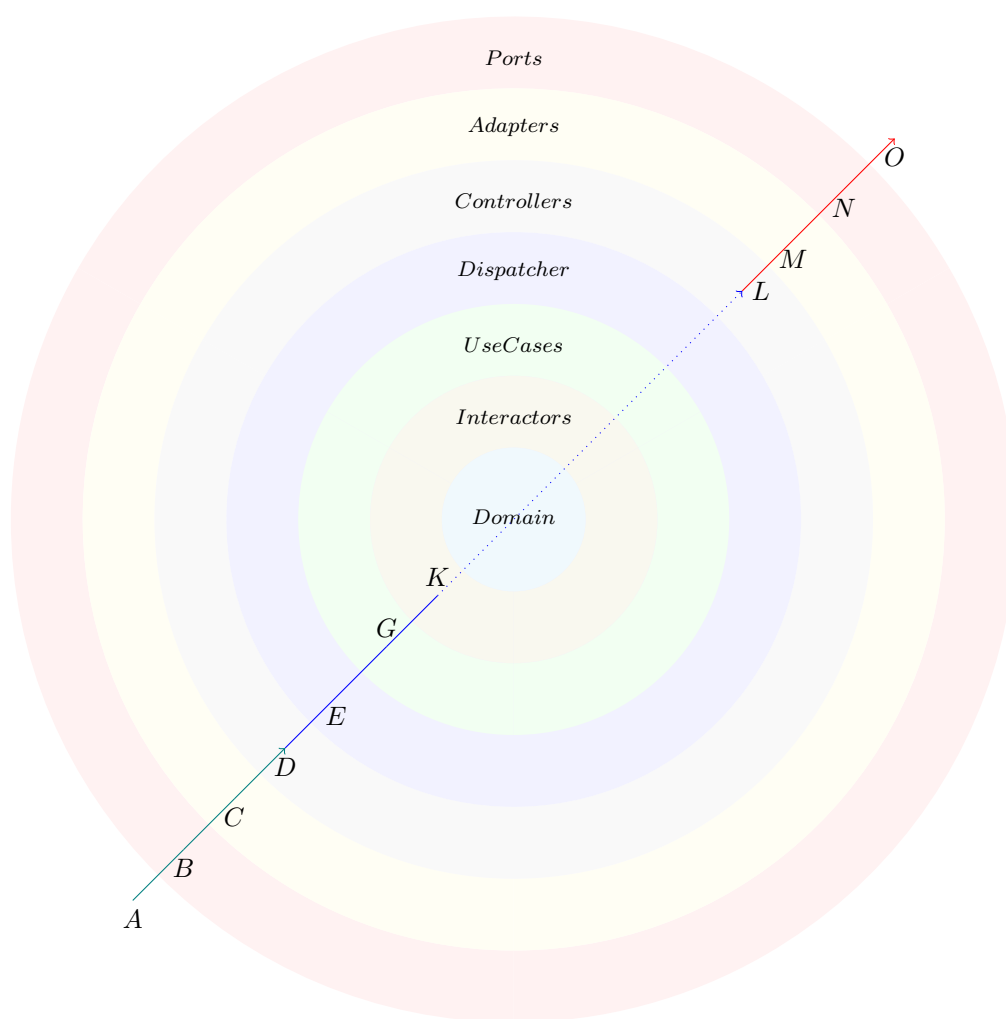
¹¹Eigene Quelle

3.8 Datenfluss im Programm

Im System gibt es 3 wichtige Datenflüsse, die durch Kombination miteinander die komplexen Abläufe im System umsetzen

1. (blau) Controller löst ein Ereignis im Dispatcher aus.
2. (rot) Controller spricht sein Port an (z.B. Speichern der Daten in der Datenbank oder OCPP Antwort abzuschicken)
3. (grün) Das Programm wird von einem externen System angesprochen (z.B. Ladesäule schickt eine OCPP Nachricht an den Server)

Wenn das geschehen ist, sieht der Datenfluss so aus:



—— Teile des Programms wissen nichts voneinander

Abbildung 10: Darstellung der umgesetzten Architektur als **Clean Architecture** mit 7 Schichten. Pfeile repräsentieren mögliche Datenflüsse ¹²

¹²Eigene Quelle

Darstellung des Datenflusses 1 als **Sequencediagramm**:

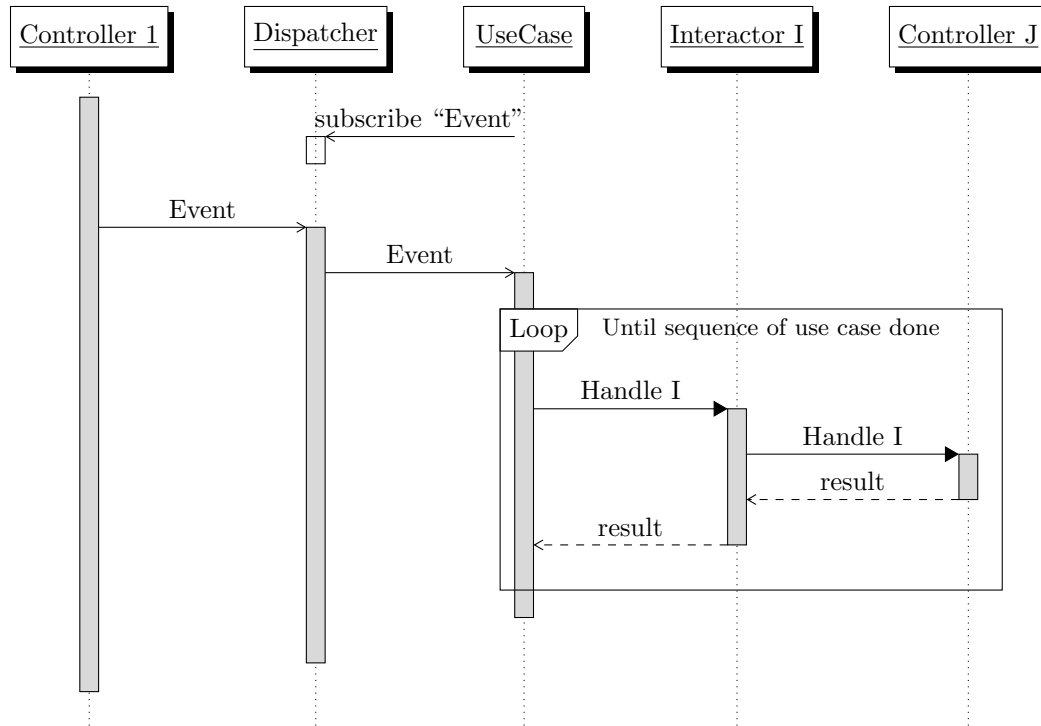


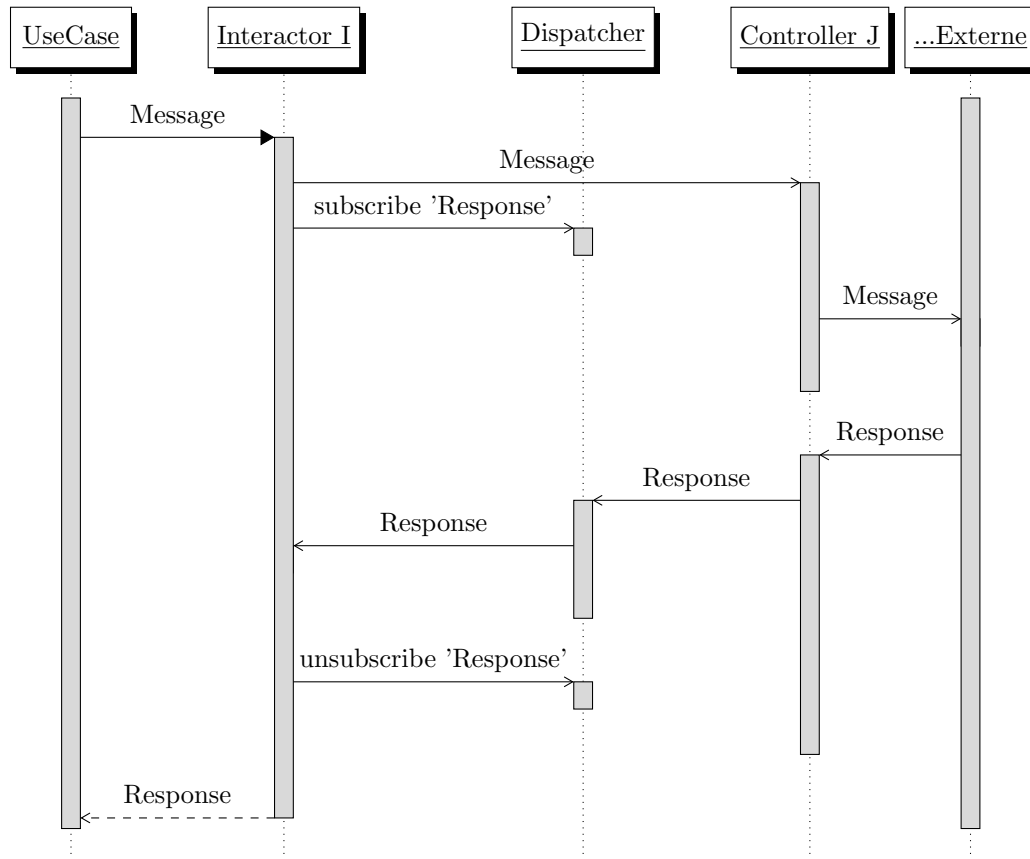
Abbildung 11: Sequencediagramm vom Datenfluss "1" Blau

Wenn im **Controller** ein Ereignis erzeugt wird, wird **Dispatcher** darüber informiert, ein oder mehrere **UseCases** haben bereits dieses Event bei **Dispatcher** aboniert. **Dispatcher** informiert alle auf das Ereignis abonierte **UseCases**. Jeder UseCases kann seinen eigenen Verhalten auf das Event definieren unabhängig voneinander. **UseCase** definiert einen Ablauf an **Interactoren**, die wie vorgeschrieben ausgeführt werden. Jeder **Interactor** ruft eine Methode von einem **Controller** auf und das Ergebnis wird an **UseCase** zurückgegeben, das vom UseCase entsprechend behandelt wird.

Dabei es gibt 2 Möglichkeiten wie das Ereignis vom **Controller J** das **Interactor I** erreichen kann:

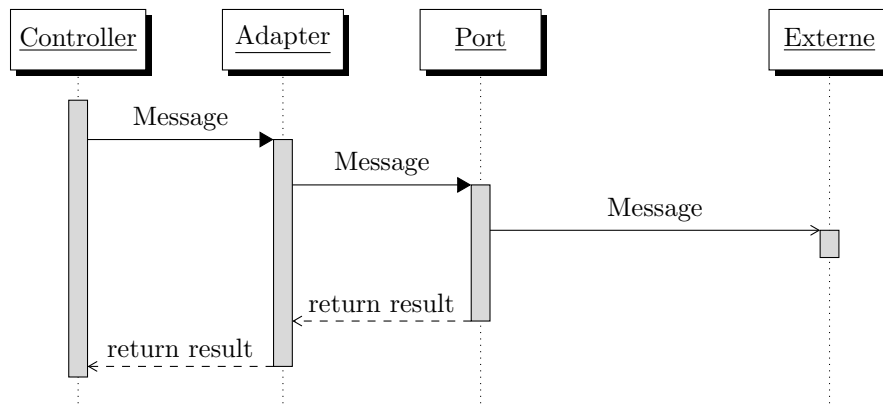
1. synchron - der Rückgabewert ist das Ergebnis der aufgerufenen Methode
2. asynchron - man wartet auf dazugehörige Antwort vom Port (z.B. auf OCPP Response warten, wenn man ein OCPP Request abschickt)

Darstellung der 2. Möglichkeit:



Der Rückgabewert wird beim synchronen Funktionsaufruf zurückgegeben, wie in der Abbildung 11 dargestellt. Der Rückgabewert beim asynchronen Funktionsaufruf, wird wie folgt definiert: Der Aufgerufene Interactor ruft eine Methode vom Controller auf, der die Nachricht an den externen Teilnehmer abschickt. Gleich danach abonniert der Interactor die Antwort auf die abgeschickte Nachricht. Wenn die Antwort ankommt, landet sie beim Dispatcher, der alle Abonnierten darüber informiert, unter anderem auch den Interactor. Der Interactor gibt diese Antwort als Rückgabewert der Funktionsaufruf

Darstellung des Datenflusses “2” als “sequencediagram”:



Darstellung der Datentransformation:

Controller - Adapter: Alle Informationen werden als Objekte übergeben, die im Domain definiert werden müssen.

```
1      OCPP20Message({
2          destination: {
3              chargerId : "some_unique_charger_id"
4          },
5          message : {
6              name : "BootNotification",
7              type : "Response",
8              payload : BootNotification({
9                  currentTime : Date(Thu Jul 28 2022 14:26:49 GMT
10                     +0200),
11                  interval : 30,
12                  status : "Rejected"
13              })
14      })
```

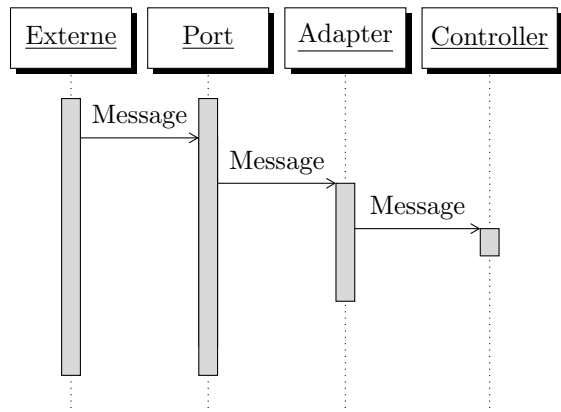
Adapter - Port: Alle Informationen, die gesendet werden (in dem Fall “message”), werden in der verständlichen Form (sie muss nicht mehr geändert werden) für den Port an Port weitergegeben. Über das Ziel müssen alle Informationen weitergegeben werden, so dass Port die entsprechende Verbindung zuordnen kann.

```
1      {
2          destination : {
3              chargerId : "some_unique_charger_id"
4          }
5          message : "[3, 'message_id_of_request', {currentTime :
6              'Thu Jul 28 2022 14:26:49Z', interval : 30, status :
              'Rejected' }] "
```

Darstellung des Datenflusses “3” als “sequencediagram”:

Beschreibung der Struktur des Programms: hier kommt das Bild nur mit der Structur

•



3.8.1 Auswahl der Datenbank

3.8.2 Dependency Rule

3.8.3 Unterschied zu Layered Architektur

3.8.4 Implementierung der Testbarkeit

- Humble Objects

3.8.5 Dependency Injection

In der Abbildung 12 sieht man ein Beispiel von einer Anwendung, die die von der Konsole ankommenden Zahlen quadriert und das Ergebnis zurückgibt.

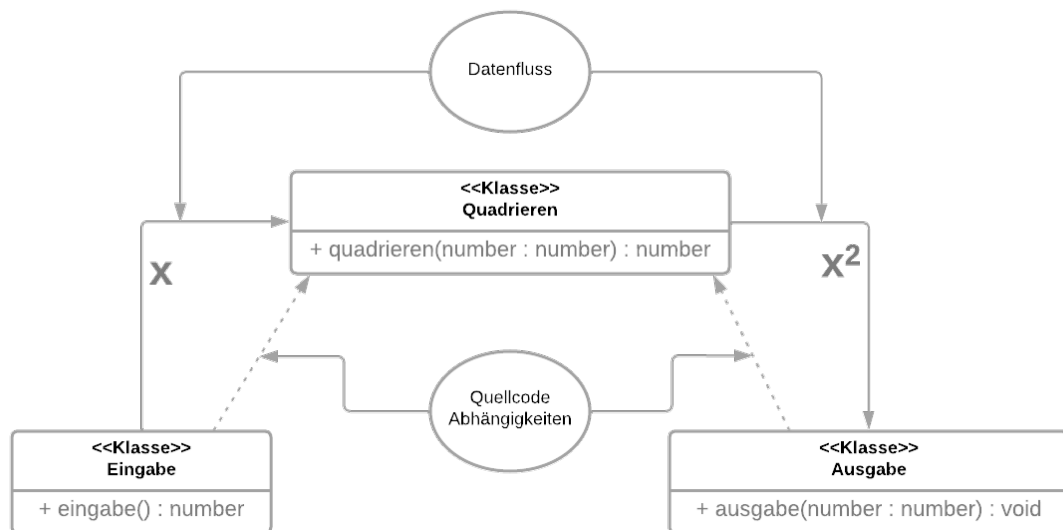


Abbildung 12: Datenfluss und Quellcode Abhängigkeiten

Source: Eigene Quelle

Die Funktion **Quadrieren** ist in dem Fall befindet sich auf einem höheren Niveau als Eingabe und Ausgabe, da das Quadrieren einer Zahl soll unabhängig von der Eingabe und Ausgabe sein.

Würde man aber bei der Ausgabe die Eingabeparameter von **number** auf **string** ändern, so müsste man auch die Ausgabe von Quadrieren von **number** auf **string** ändern. Dies könnte auch eine weitere Kette an Änderungen im Programm auslösen. Zum Beispiel müssen auch die Unittests von **Quadrieren** geändert werden. Somit ist **Quadrieren** abhängig von der **Ausgabe**

Das Problem lässt sich mittels Dependency Injection lösen. In OOP Sprachen kann man dafür “Interface” benutzen.

Die Lösung wurde dann so aussehen.

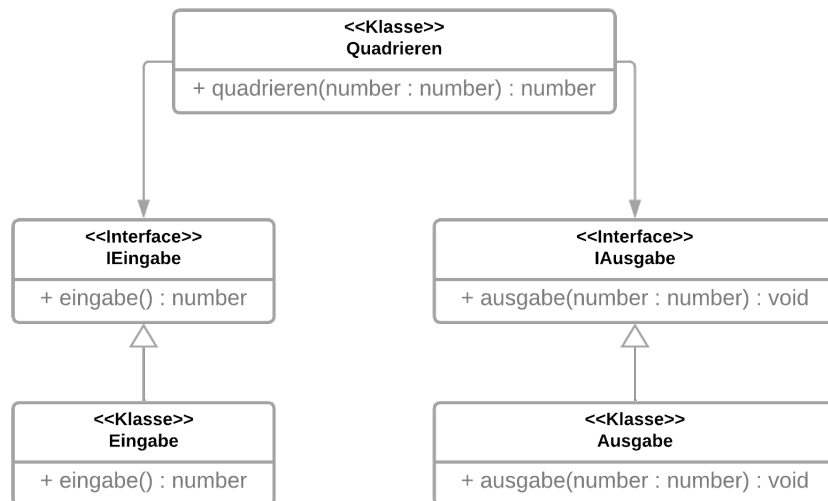


Abbildung 13: Entkopplung der Abhängigkeiten

Source: Eigene Quelle

Dies lässt sich mit **Interface** (für OOP Sprachen) umsetzen, in dem es frühestens bei der Initialisierung der **Quadrieren** Klasse das jeweilige Eingabe und Ausgabe Objekt übergeben wird.

Somit lässt sich die Funktion **Quadrieren** mit gefälschten Eingabe- und Ausgabeklasse mit Unit tests getestet werden.

Wenn man alle Klassen über Interface miteinander verbindet, ist es möglich, dass die Umgebung von jeder einzelnen Klasse bei den Unittests gefälscht wird und somit das Schreiben von Unittests sehr einfach wird.

Interfaces können sich natürlich auch ändern und dann muss man auch alle davon betroffenen Objekte entsprechend ändern, jedoch das passiert deutlich seltener als Änderung einer Klasse.

Auch mit Dependency Injection lassen sich externe Schnittstellen wie Datenbank oder Netzwerkschnittstellen schnell austauschen, denn man muss nur eine Klasse schreiben, die das Interface implementiert.

4 Aufgabenstellung

Zu Beginn der Praxisphase war die Entwicklung in der Abteilung, in der ich meine Praxisphase gemacht habe, und Testen an vielen Stellen an eine externe Schnittstelle gebunden. Um diese Abhängigkeit zu reduzieren und die Möglichkeit zu bekommen eigene Funktionalitäten hinzuzufügen, bekam ich die Aufgabe einen OCPP 1.6. Server zu implementieren.

Zu dem Zeitpunkt gab es bereits drei Stellen, bei denen der OCPP1.6 Server Funktionalitäten in unterschiedlichem Umfang benutzt wurden.

Das sind:

- Testframework für die automatisierten Systemtests von der Ladesäule
- Eigenständiger OCPP1.6 und später OCPP2.0 Server für die manuellen Tests
- Automatisierungstool für den ERK (Eichrechtkonformität) Prozess

Die dritte Aufgabe (Automatisierungstool für den ERK Prozess) hatte größere Priorität und somit wurde diese als erstes erledigt. Die Aufgabe hat dabei den kleinsten Anteil an OCPP Server Funktionalitäten gebraucht.

Anschließend wurden die Teilaufgaben erledigt, aufgrund der großen Überschneidung der programmiertechnischen Anforderungen, konnten Teile der vorhergehenden Aufgabe übernommen werden.

4.1 Anforderungen an den Standalone Server

Der Server, der im lokalen Netz auf einem Raspberry Pi läuft, soll für die manuellen Tests der Ladesäule benutzt werden. Dieser Server wird benutzt um die komplexeren Testfälle nachzubilden oder neue Funktionalitäten, die mit Hardware interagieren, zu testen. Der Server kann bei der Präsentation der Funktionalitäten genutzt werden.

Die Anforderungen an den Server sind:

- Der Server soll eine OCPP1.6 Schnittstelle besitzen.
- Der Nutzer soll in der Lage sein den Server zu parametrieren (z.B. einen neuen Benutzer hinterlegen)
- Der Nutzer soll in der Lage sein die Nachrichten an die Ladesäule manuell verschicken zu können

4.2 Anforderungen an das Testframework

Das Testframework soll für die automatisierten Systemtests der Software der Ladesäule (sowohl mit als auch ohne Hardware) benutzt werden.

Die Anforderungen an das Testframework sind:

- Der OCPP Server soll den Port selber auswählen können, um mehrere Tests parallel starten zu können.
- Das Verhalten von dem Testserver soll geändert werden können (auch während der Tests)
- Das Defaultverhalten von dem Testserver soll parametrierbar sein (z.B. einen Benutzer hinzufügen)
- Alle Events, die den Zustand der getesteten Ladesäule aufdecken, sollen beobachtbar sein (z.B. OCPP Nachrichten, Netzwerkevents usw.)

4.3 Anforderungen an ERK Automatisierungstool

Der Zertifizierung nach dem deutschen Eichrecht entsprechend, muss jede Ladesäule auf Eichrechtskonformität (ERK) überprüft werden. Dieser Prozess wird immer wieder auf die gleiche Art und Weise im gleichen Umfang durchgeführt. Er beinhaltet somit ein entsprechendes Automatisierungspotential.

Dafür muss für jede Ladesäule ein Ladevorgang gestartet werden (Transaction), währenddessen Strom fließt und gemessen wird. Um die Datenintegrität der Messwerte und deren Transport sicherzustellen wird der OCPP Server verwendet. Mit dessen Hilfe kann nachgewiesen werden, dass die Daten nirgendwo in der Software geändert und ebenso unverändert an den Server übertragen und dort abgerechnet wurden. Nach Beenden der Transaction werden mittels einer Drittsoftware die gemessenen Daten mit den Transactiondaten verglichen.

Die gewünschte Software soll demnächst von den Mitarbeitern im End-Of-Line benutzt werden, somit muss die Bedienbarkeit der Software sehr hoch sein, um die Fehlermöglichkeiten stark einzugrenzen und die Einarbeitungszeit zu reduzieren.

Die Anforderungen an das ERK Automatisierungstool sind:

- Leichte Bedienbarkeit der Software
- Leicht Integrierbar in das andere Automatisierungstool

5 Lösung der Aufgabe

hier wird die Lösung der Aufgabe beschrieben mit Bildern usw.

6 Gewünschtes Interfaces

Jedes Tool, Bibliothek, Framework, das von den anderen Menschen benutzt wird, soll soweit wie möglich selbsterklärend sein und intuitiv klar sein.

Damit diese Eigenschaft umgesetzt wird, könnte man die zukünftigen Anwendungen festlegen und daraus eine gute selbsterklärende Schnittstelle erstellen.

Jeder Ablauf eines Systemtests kann mit folgender Schema beschrieben werden:

- 1. Erstellen des Servers mit gewünschten Netzwerkeinstellungen
- 2. Parametrieren/Festlegen des gewünschten Verhaltens des Servers
- 3. Server starten
- 4. Festlegen die Bedingungen für den erfolgreichen Test
- 5. Warten bis der Test abgeschlossen wird
- 6. Ergebnisse validieren
- 7. Alle Instanzen löschen

7 Implementierung

Bei der vorgestellten Implementierung werden Systemtests verwendet. Bei dieser Art von Test wird die Software komplett und analog zum Produktivumfeld getestet. Deshalb sind Systemtests sehr zeitintensiv. Mit folgenden Möglichkeiten kann der Einsatz von Systemtest optimiert werden:

- mehrere Tests pro Setup definieren und ausführen.
- die Tests die in unterschiedlichen Setups ablaufen (d.h. nicht miteinander verbunden sind) parallel durchführen

Um die Lesbarkeit des Tests zu verbessern wäre es vom Vorteil, wenn die erstellte OCPPServer Testinstanz bereits ein vordefiniertes Verhalten besitzt, das man ändern kann.

Es soll auch möglich sein das vordefinierte Verhalten zu parametrieren. Dies erfordert Interfaces, die bestimmte Parameter der Instanz ändern können.

Da nur das Verhalten von dem Charging Point getestet werden soll, sollen nur die Ereignisse, die den Zustand des Charging Points abbilden, abrufbar sein. Zum Beispiel: geschickte Nachrichten von dem Charging Point zu dem Server, Reihenfolge der Nachrichten.

7.1 Achitecture des Frameworks

Es wurde entschieden das Framework in 7 Abstraktionsschichten aufzuteilen.

7.1.1 Ports

Ports haben die Aufgabe die Schnittstelle nach Außen aufzubauen und die Verbindungen zu (z.B. WebSocket Server, Datenbank)

In dem Framework wird nur ein Port gebraucht - WebSocket Server

7.1.2 Adapters

Adapters sollen die ankommenden Events/Messages vom Port an den zugehörigen Controller zu übersetzen.

In dem Framework wird nur einen Adapter gebraucht - OCPP16 Adapter

7.1.3 Controllers

Controllers besitzen alle Informationen die den Zustand des jeweiligen Components (Controller + Adapter + Port) abbilden.

In dem Framework werden mehrere Controllers gebraucht:

- OCPP Controller(übernimmt die Verantwortung über die Verbindungen zu den Charging Points)
- User Controller(übernimmt die Verantwortung über die Nutzer der Charging Points und ihrer Berechtigungen)
- Transaction Controller(übernimmt die Verantwortung über die Controller über die Ladevorgängen)

-
- Charger Controller(übernimmt die Verantwortung über die Charging Points, die von dem Server bekannt sind und ihrer Zuständen)
 - Payment Controller(übernimmt die Verantwortung über die Bezahlvorgang nach dem Ladevorgang)

Die Controller können von dem Nutzer des Frameworks parametrisiert werden, um das Verhalten des Servers zu ändern.

7.1.4 Dispatcher

Der Dispatcher informiert alle abonnierten UseCases über das eingetretene Event.

In dem Framework sind nur OCPP Events wichtig (Nachrichten und Verbindungsevent).

7.1.5 UseCases

UseCases beschreiben den Vorgang beim Auslösen eines Events, welches sie abonniert haben. Die vordefinierten UseCases dürfen nur Interactors benutzen um das Verhalten zu definieren.

Dieses Framework beinhaltet UseCases, welche ein vordefiniertes Verhalten besitzen. Diese kann auch entsprechend umgeschrieben werden.

7.1.6 Interactors

Eine atomare Operation im Programm (die Operation lässt sich nicht mehr sinnvoll im Rahmen der Anwendung aufteilen). Interactors benutzen Controller "Dependency Injection"

Benutzt mittels "Dependency Injection" die Controller.

In dem Framework werden sie nur als "Wrapper" für alle Funktionen von Controllern implementiert.

7.1.7 Domain

Eine Domain definiert alle Types und Interfaces der Applications. Sie beschreibt die Verbindungen zwischen den Interfaces und Types, die dann in den anderen 6 Layers umgesetzt werden.

7.2 Zugriff auf das Testframework

IRGENDWAS EINLEITENDES

- OCPPPort soll nur bei der Initialisierung der Instanz parametrisierbar sein (Netzwerkeinstellung)
- Adapters sollen nicht von der Seite des Frameworks aufrufbar sein
- Controller sollen nicht von der Seite des Frameworks aufrufbar sein
- Dispatcher darf nur zum Abonnieren/Deabonnieren benutzt werden um das Verhalten des Charging Points beobachten zu können
- UseCases sollen überschreibbar und erweiterbar sein, falls man bestimmtes Verhalten hinzufügen möchte.

- Interactors sollen von der Seite des Frameworks aufrufbar sein, um die Serverinstanz parametrieren zu können.
- Domain beinhaltet alle Typen die in den anderen Layers verwendet werden. Aus diesem Grund sollen die Typen von der Seite des Frameworks benutzbar sein.

7.3 Testbeispiel

```
describe("example test", () => {
  let testOcppInstanz: MyOCPPTestFramework;

  before(async () => {
    // 1. Create test server instanz
    testOcppInstanz = new MyOCPPTestFramework({ host: "https://127.0.0.1", port: 8080 });

    //2.1. rewrite behavior of server
    testOcppInstanz.rewriteUseCase("nameOfUseCase", {});
    //2.2. add behavior to server
    testOcppInstanz.addUseCase({});

    //2.3. parametrize the behavior
    testOcppInstanz.interactors.interactor_1();
    testOcppInstanz.interactors.interactor_2();

    // 3. start server
    await testOcppInstanz.start();
    return;
  });

  it("test 1", async () => {
    // 5. wait until test is done
    const res = await testOcppInstanz.waitForEvent("BootNotification");

    // 4. and 6. set the requirement test is valid, validate the result
    expect(res.name).equal("BootNotification");
  });

  it("test 2", async () => {
    // 5. wait until test is done
    const res = await testOcppInstanz.waitForNextEvent();

    // 4. and 6. set the requirement test is valid, validate the result
    expect(res.name).equal("Heartbeat");
  });

  after(async () => {
    // 7. stop server
    testOcppInstanz.stop();
    return;
  });
});
```

8 Übersichtsdiagramm

9 Conclusion

But the fact that some geniuses were laughed at does not imply that all who are laughed at are geniuses. They laughed at Columbus, they laughed at Fulton, they laughed at the Wright Brothers. But they also laughed at Bozo the Clown - Sagan [Sag93].

Bibliography

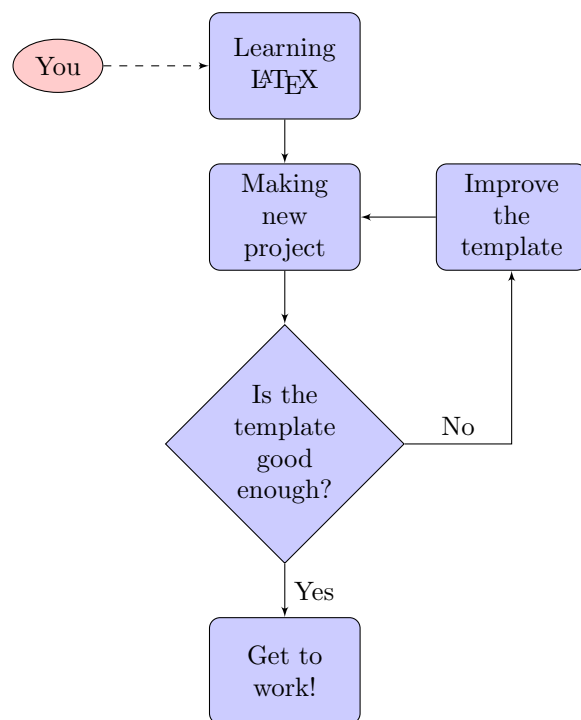
- [GGS82] U. Ghia, K. N. Ghia und C. T. Shin. „High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method“. In: *Journal of Computational Physics* 48 (1982), S. 387–411.
- [Sag93] Carl Sagan. *Brocas brain: reflections on the romance of science*. Presidio Press, 1993.
- [Mar18] Robert C. Martin. *Clean Architecture*. 2018. ISBN: 978-0-13-449416-6.
- [Fowa] Martin Fowler. URL: <https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf>.
- [Fowb] Martin Fowler. *Is High Quality Software Worth the Cost?* URL: <https://martinfowler.com/articles/is-quality-worth-cost.html>.
- [NTN] Department of Marine Technology NTNU. *IMT Software Wiki - LaTeX*. URL: <https://www.ntnu.no/wiki/display/imtsoftware/LaTeX> (besucht am 15. Sep. 2020).
- [Rod] Vladislav Rodin. URL: <https://tech-de.netlify.app/articles/de521476/index.html>.

Appendix

A Hello World Example

⋮
⋮
⋮

B Flow Chart Example



C Sub-figures Example

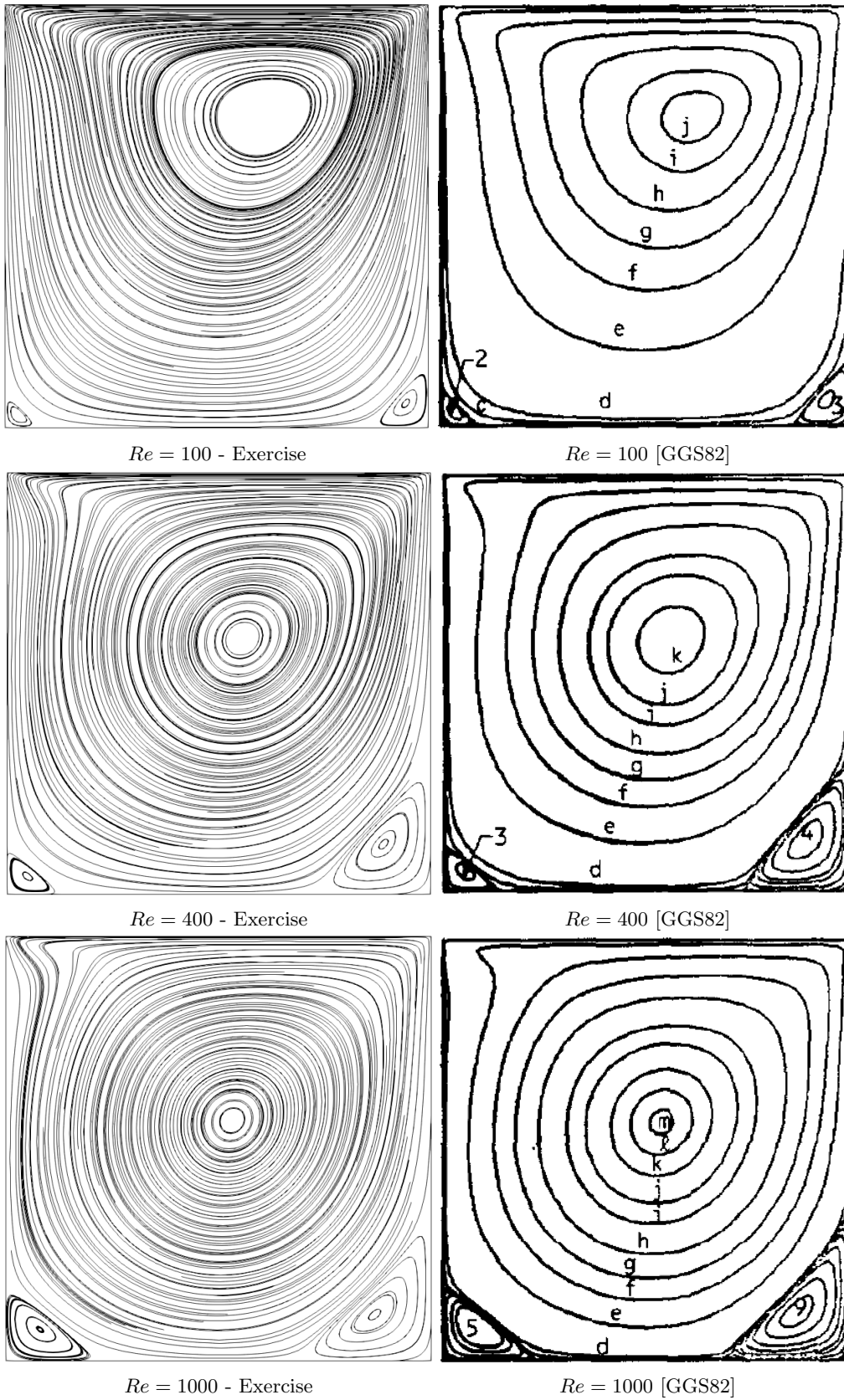


Abbildung 14: Streamlines for the problem of a lid-driven cavity.