



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

15. September 2022

Inhaltsverzeichnis

Abbildungsverzeichnis	iii
Tabellenverzeichnis	iv
1 Motivation	1
2 Grundlagen	1
2.1 OOP Design Patterns	1
2.1.1 Adapter	2
2.1.2 Observer	2
2.1.3 Proxy	3
2.1.4 Template method	3
2.1.5 Builder	3
2.1.6 Facade	3
2.1.7 Singleton	3
2.1.8 Factory method	3
2.2 Architekturen	4
2.2.1 Clean Architectur	4
2.2.2 Model-View-Presenter Architektur	5
2.2.3 Port und Adapter Architektur	6
2.3 Software Testing	6
2.3.1 Testing Pyramide	7
2.3.2 Unit Tests	7
2.3.3 Integration Tests	7
2.3.4 SystemTests	8
2.3.5 UI Tests	8
2.3.6 Manual Tests	8
2.4 Softwareentwicklung Prinzipien	8
2.4.1 Single-responsibility principle	8
2.4.2 Open–closed principle	8
2.4.3 Interface segregation principle	9
2.4.4 Dependency inversion principle	9
2.4.5 Inversion of control	10
2.4.6 Dependency Injection	10

2.5	OCPP Kommunikation	12
3	Software Architektur	13
3.1	Software Architektur aus Sicht der Projektführung	13
3.1.1	Ziele der Software Architektur	13
3.1.2	Technische Schulden	14
3.1.3	Qualität und Kosten der Software	15
3.1.4	Testbarkeit der Software ist wichtig	16
3.2	Technische Umsetzung der Software Architektur	17
3.2.1	Abhängigkeiten im Programm	19
3.2.2	Datenfluss im Programm	26
3.2.3	Erweiterung der Funktionalitäten	32
3.3	Anwendung	34
3.3.1	Anbindung in eine andere Anwendung als eine Komponente	34
3.3.2	Framework und Bibliothek	37
4	OCPP Server	38
4.1	Aufgabenbeschreibung	38
4.1.1	Standalone Server mit Datenbank	38
4.1.2	Standalone Server ohne Datenbank	39
4.1.3	Testframework	40
4.1.4	ERK Automatisierungstool	41
4.2	Allgemeine Lösung	43
4.2.1	Controllers	43
4.2.2	Dispatcher	44
4.2.3	UseCases	44
4.2.4	Interactors	45
4.3	Anwendung der allgemeinen Lösung	46
4.3.1	Standalone mit Datenbank	48
4.3.2	Standalone ohne Datenbank	49
4.3.3	Library	50
4.3.4	Framework	51
4.4	Vergleich der einzelnen Lösungen	53
	Bibliography	54

Abbildungsverzeichnis

1	CI/CD Pipeline	1
2	UML Adapter	2
3	UML Observer	2
4	Clean Architecture	4
5	Datenfluss in MVP Architektur	5
6	Port und Adapter Architektur	6
7	Testing Pyramide	7
8	Schlechte laut DIP Abhängigkeit	10
9	Gute laut DIP Abhängigkeit	10
10	Datenfluss und Quellcode Abhängigkeiten	11
11	Entkopplung der Abhängigkeiten	11
12	Vergleich einer guten und einer schlechten Softwarearchitektur	16
13	some Caption	17
14	Objektendiagramm PAC	20
15	Objektendiagramm Controller-Dispatcher-UseCase-Interactor	21
16	Ablaufdiagramm Erstellen der Struktur	22
17	Objektendiagramm mit Utility Controllers	24
18	Klassendiagramm Port-Adapter-Controller	24
19	Klassendiagramm Controller-Dispatcher-UseCase-Interactor	25
20	some Caption	26
21	Sequencediagramm vom Datenfluss 1 Grün	27
22	Sequencediagramm vom einfachen Datenfluss 2 Blau	28
23	Sequencediagramm vom komplexen Datenfluss 2 Blau	29
24	Kompletter Datenfluss	31
25	Vereinfachte Darstellung	34
26	Vereinfachte Darstellung einer Standalone Anwendung	34
27	Vereinfachte Darstellung einer Standalone Anwendung	34
28	Vereinfachte Darstellung der Architektur als Komponente	35
29	Vereinfachte Darstellung einer Standalone Anwendung mit der Komponente	36
30	Vereinfachte Darstellung des Datenflusses in einer Anwendung mit Komponente	36
31	Vereinfachte Darstellung des Datenflusses bei einer Bibliothek	37
32	Vereinfachte Darstellung des Datenflusses bei einem Framework	37
33	Übersichtdiagramm der Standalone-Anwendung mit Datenbank	39

34	Übersichtdiagramm der Standalone-Anwendung ohne Datenbank	40
35	Übersichtdiagramm der Framework-Anwendung	41
36	Übersichtdiagramm der ERK-Software	42
37	some Caption	43
38	Ablauf eines Interactors	45
39	Klassendiagramm eines Interactors	45
40	some Caption	47
41	some Caption	48
42	some Caption	49
43	some Caption	50
44	some Caption	51
45	Ablaufdiagramm eines Tests	52

Tabellenverzeichnis

1	Vergleich der Anzahl der Codezeilen	53
---	---	----

1 Motivation

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

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

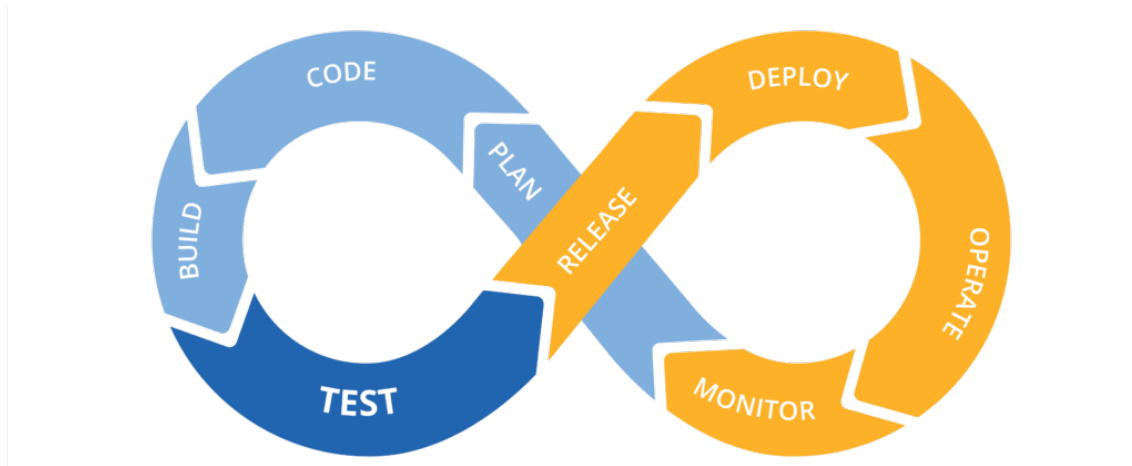


Abbildung 1: CI/CD Pipeline ¹

Das Ziel ist es, die Gesamtzeit des Zyklus so minimal wie möglich zu halten, um dem Kunden die neuen Funktionalitäten schneller zur Verfügung stellen und ebenso Bugs schneller eliminieren zu können.

Die Entwicklung von Software lässt sich in zwei übergeordnete Gruppen unterteilen: 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 sind ab einem gewissen Automatisierungsgrad ressourcenschonender. Der größte Anteil an Ressourcen wird in den 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 zu Beginn des Projektes sinnvoll, Möglichkeiten von Testautomatisierungen zu betrachten.

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.

2 Grundlagen

SomeIntroducation

2.1 OOP Design Patterns

some Introduction

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

2.1.1 Adapter

Der OOP Design Pattern **Adapter** ermöglicht zwei Objekte mit unterschiedlichen Schnittstellen miteinander zu verbinden. Adapter kann übergebene Daten transformieren (z.B. JSON in XML) oder die Schnittstelle des zweiten Objektes für das erste Objekt anpassen (z.B. Parameterliste oder Anzahl der Methoden).

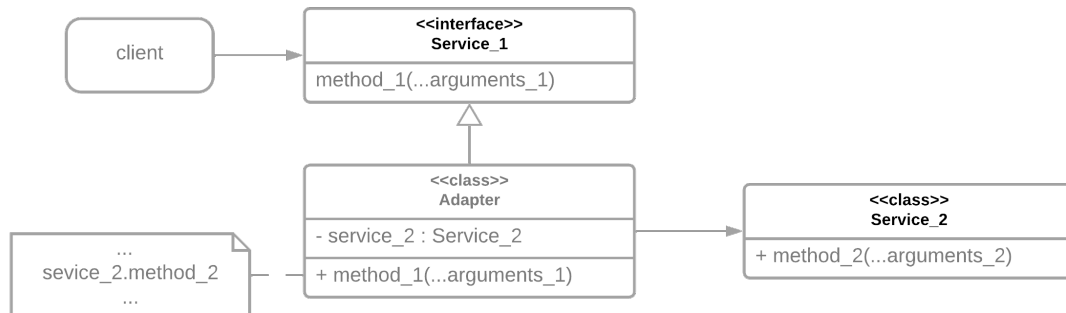


Abbildung 2: Klassendiagramm Adapter

2.1.2 Observer

Das OOP Design Pattern **Observer** (dt. Beobachter) 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.

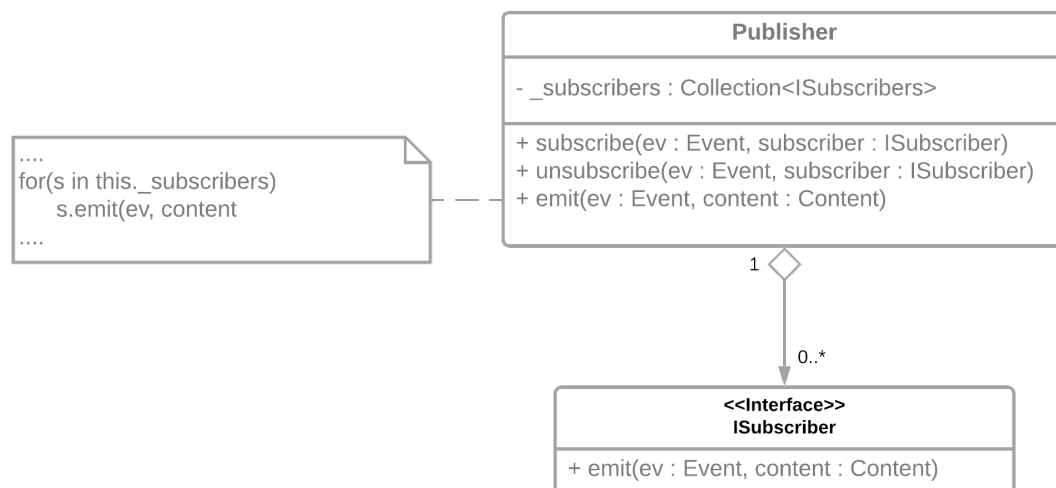


Abbildung 3: Klassendiagramm Observer

2.1.3 Proxy

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

2.1.4 Template method

Das OOP Design Pattern **Template Method** (dt. Schablonenmethode) 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.

2.1.5 Builder

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

2.1.6 Facade

Das OOP Design Pattern **Facade** (dt. Fassade) 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.

2.1.7 Singleton

Mehr zu Singleton kann *Refactoring guru* lesen.

Singleton (dt. Einzelstück) garantiert, dass eine Klasse nur eine Instanz in der Anwendung hat. Dieser Pattern kann sehr hilfreich sein, wenn es bekannt ist, dass es nur eine Instanz der Klasse im Programm gibt und es sie aus allen Stellen der Anwendung erreichbar sein soll. Das Verhalten ähnelt sich mit dem Verhalten einer globalen Variable deren Wert (die erstellte Instanz) nicht ersetzt werden kann.

Ein großer Nachteil bei der Benutzung des Singletons ist, dass die Teile des Programms, in denen Singleton aufgerufen wird, nicht unabhängig von ihm getestet werden können.

2.1.8 Factory method

Factory method (dt. Fabrikmethode) ist ein OOP Design Pattern, mit dem die Instanziierung nicht direkt mittels eines Konstruktors der Instanz sondern indirekt mittels einer Methode einer anderen Instanz (Fabrik), die die Aufgabe der Instanziierung übernimmt.

Die Vorteile der Übergabe der Instanziierung von Objekten an eine andere Instanz sind:

- Instanzierte Objekte lassen sich überwachen (z.B. zählen)
- Komplexe Instanziierung (z.B. mehrere Funktionsaufrufe) lassen sich hinter eine Methode verbergen.
- Lange Parameterliste lassen sich kürzen, indem die gleichen Parameter von der Fabrik übergeben werden.

2.2 Architekturen

2.2.1 Clean Architektur

Mehr zu den “Clean Architecture” kann man im Buch ... lesen.

Die Architektur besteht aus konzentrischen Kreisen. Die Kreise repräsentieren verschiedene Aufgaben, die eine Anwendung erledigen kann. In der Abbildung 4 sind vier Kreise dargestellt. Die Anzahl an Kreisen kann variieren, solange Dependency Rule erfüllt wird.

Die Dependency Rule besagt, dass die Kreise keine äußeren Kreise benutzen dürfen.

Folgende Eigenschaften hat diese Architektur laut Robert Martin:

- Unabhängig von dem Framework
- Testbar
- Unabhängig von der UI
- Unabhängig von der Datenbank
- Unabhängig von jeder externen Schnittstelle

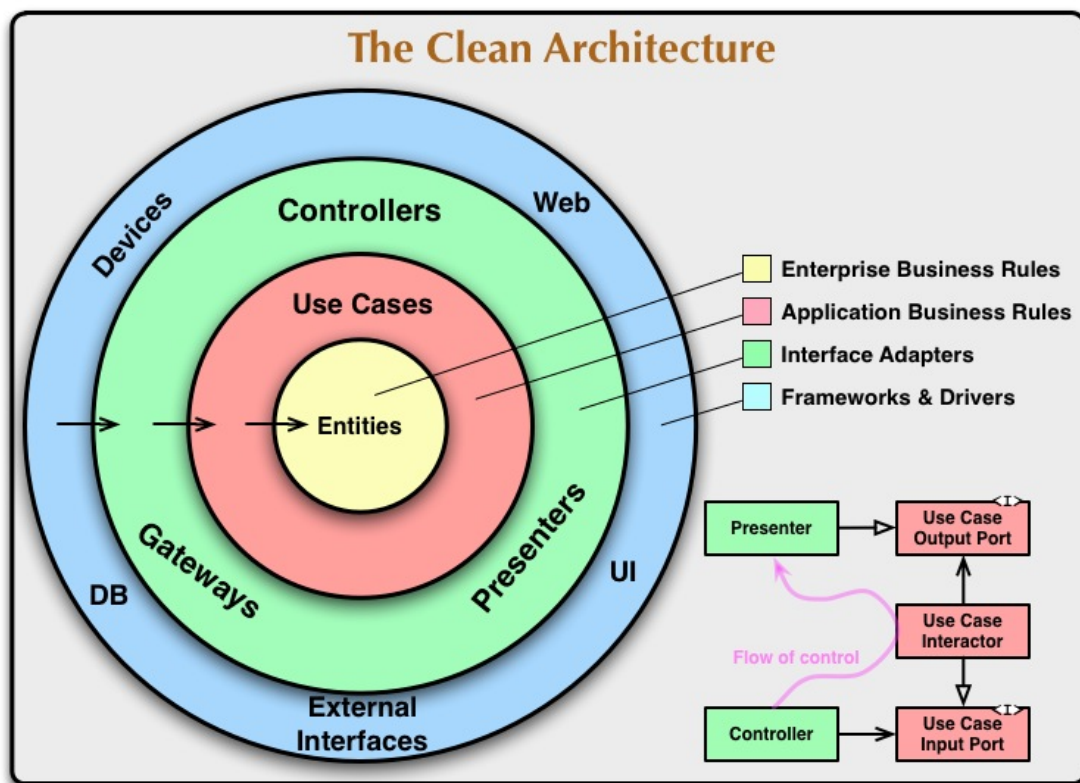


Abbildung 4: Clean Architecture ²

²<https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>

2.2.2 Model-View-Presenter Architektur

Mehr zu MVP (Model-View-Presenter) kann man *in folgende Quelle* lesen.

Model-View-Presenter Architektur wird in den Anwendungen benutzt, die eine Oberfläche besitzen. Die Architektur teilt die Anwendung in drei Teile:

- **Model** - enthält die komplette Logik des Programms.
- **View** - empfängt alle Ereignisse von der Benutzeroberfläche (UI) und enthält die Daten, die angezeigt werden sollen.
- **Presenter** - transformiert die Daten in beide Richtungen vom Model zu View und vom View zu Model

Eigenschaften der **MVP** Architektur:

- **Presenter** und **Model** lassen sich mit Unittests abdecken.
- Jede neue **View** braucht einen eigenen **Presenter**.

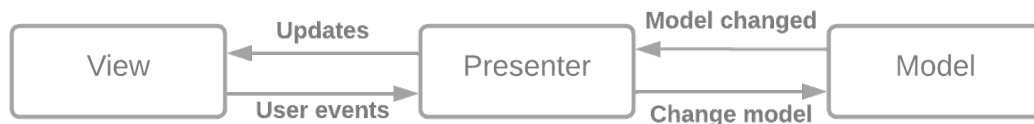


Abbildung 5: Datenfluss in MVP Architektur

Der Datenfluss in einer Anwendung, die mittels **MVP** implementiert ist, lässt sich wie folgt darstellen:

- In der Oberfläche (**View**) wird ein Ereignis erzeugt(z.B. ein Button wurde gedrückt).
- **View** gibt das Ereignis an **Presenter** weiter.
- Das Ereignis wird im **Presenter** einem im **Model** definierten Ereignis zugeordnet(z.B. Button "Speichern" wurde gedrückt)
- Das **Model** bearbeitet das Ereignis (z.B. die Datei wurde gespeichert) und gibt das Ergebnis zurück.
- **Presenter** empfängt das Ergebnis (z.B. die Datei wurde erfolgreich gespeichert) und wandelt das in ein Ereignis, das **View** interpretieren kann (z.B. Button "Speichern" soll grün werden).
- **View** arbeitet das Ergebnis und zeigt es an (z.B. Button wird grün auf der Oberfläche angezeigt)

2.2.3 Port und Adapter Architektur

Das Ziel von der Port und Adapter Architektur ist die Logik (das Verhalten) der Anwendung von den benutzen Schnittstellen zu trennen. Damit können die externen Schnittstellen schnell und problemlos ausgetauscht werden und es besteht die Möglichkeit die Anwendung ohne Schnittstellen zu starten.

Laut der Architekturbeschreibung besteht die Anwendung, die nach der Ports und Adapters Architektur umgesetzt ist, aus drei wesentliche Teile:

- Ports (blaues Teil)
- Adapters (rotes Teil)
- Core (gelbes Teil)

Im **Core** ist das komplette Verhalten der Anwendung definiert. Jeder **Port** verbindet die Anwendung mit einer anderen externen Anwendung (z.B. Benutzeroberfläche wird über HTTP-Schnittstelle verbunden oder Datenbank-Schnittstelle). Jeder **Port** benötigt einen **Adapter**, um die ankommenden Nachrichten für den **Core** zu transformieren bzw. ausgehende Nachrichten für den **Port** zu transformieren.[And]

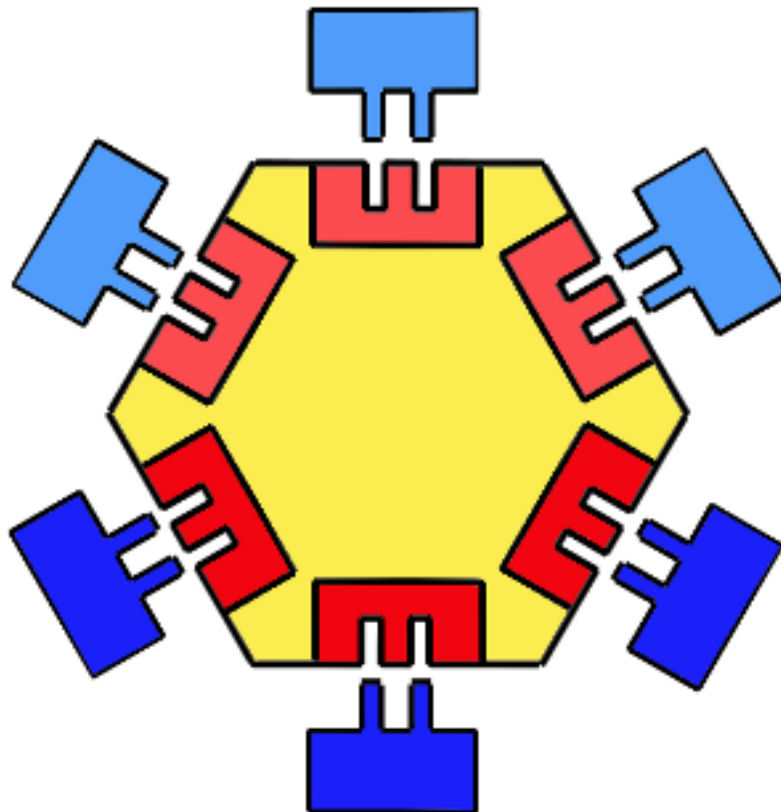


Abbildung 6: Port und Adapter Architektur ³

2.3 Software Testing

Jede Software wird im Laufe der Zeit viel geändert und erweitert. Die erste Version einer Software kann beispielsweise mit einer Version der selben Software nach 10 Jahren, wenig bis keine Gemein-

³https://www.dossier-andreas.net/software_architecture/ports_and_adapters.html

samkeiten besitzen. Jede Änderung des Quellcodes ist ein Risiko für Softwareentwickler, da immer die Gefahr besteht versehentlich funktionierenden Code zu beschädigen.

Um das Risiko für 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

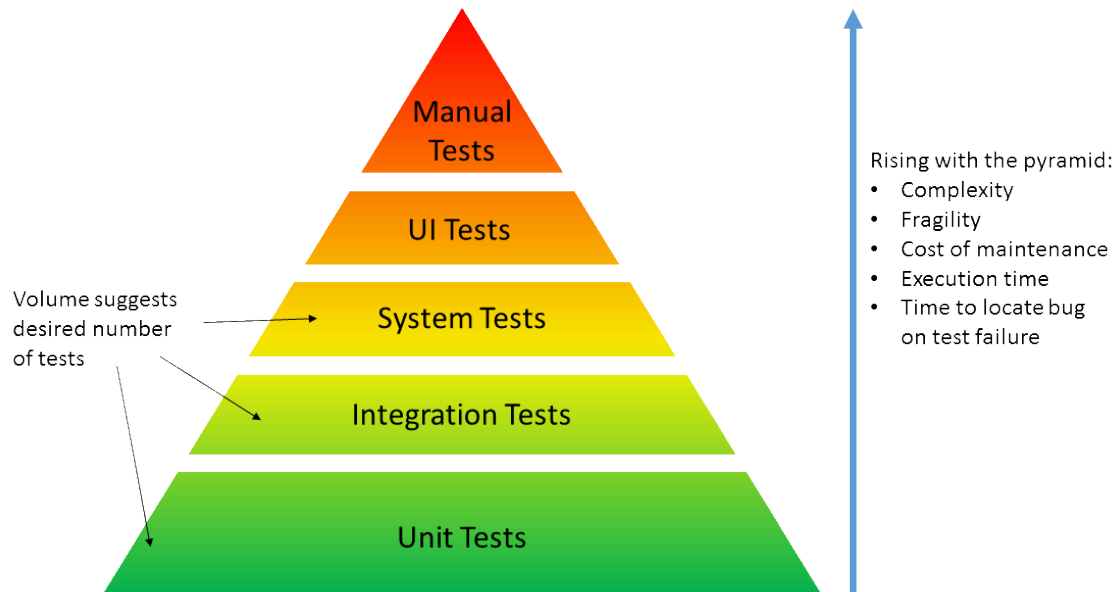


Abbildung 7: Caption written below figure ⁴

2.3.2 Unit Tests

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.

Da sie nur einen kleinen Teil des Programms abdecken, sind sie am meistens in einem Projekt repräsentiert und haben die geringste Laufzeit im Vergleich zu den anderen 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.

2.3.3 Integration Tests

Die Anzahl an Integration Tests in einem Projekt ist geringer als Anzahl an Unit Tests, da es mehrere Module gleichzeitig getestet werden. Auf einer Seite ist ein Integration Test größer als

⁴<https://www.cqse.eu/de/news/blog/junit3-migration/>

ein Unit Test. Auf der anderen Seite ist die Fehlersuche im Programm bei negativem Testergebnis deutlich komplexer und zeitintensiver, als bei Unit 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.

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 die zu prüfende Software wie ein Produktivsystem gestartet und entsprechend getestet. Alle externen Schnittstellen werden im Test simuliert und jeder Test wird nur an Ausgaben an externen Schnittstellen validiert. Da die komplette Anwendung getestet wird, ist die Anzahl an benötigten Tests geringer als bei Integrationstests. Die Laufzeit jedes einzelnen Tests ist sehr groß und die Fehlersuche dauert länger, da der Fehler sich schwer in der Anwendung lokalisieren lässt.

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.

Bei den Tests muss die Ausgabe an einer Benutzeroberfläche entsprechend angezeigt werden, dies nimmt viel Zeit in Anspruch. Da aber nur die Ausgaben an Benutzeroberflächen getestet und alle anderen internen Abläufe ignoriert werden, ist die Anzahl dieser Tests sehr gering.

2.3.6 Manual Tests

some Info about manual tests

2.4 Softwareentwicklung Prinzipien

2.4.1 Single-responsibility principle

Das Single Responsibility Prinzip besagt, dass es nie mehr als einen Grund geben sollte, ein Modul (z.B. eine Klasse) zu ändern. Zum Beispiel wenn ein Modul zwei Aufgaben erledigt, die sehr wenig oder gar nicht miteinander verbunden sind, sollte das Modul in zwei Module aufgeteilt werden.

Es lässt sich auch überprüfen, indem die Aufgabe und Verantwortlichkeit von einem Modul in einem Satz zusammengefasst werden. Wenn es dabei das Word “und” benutzt wird, ist es ein Zeichen, dass das Modul aufgeteilt werden soll.

2.4.2 Open–closed principle

Das Open-Closed-Prinzip (dt. Prinzip der Offen- und Verschlussenheit) beschäftigt sich mit Erweiterbarkeit und Änderbarkeit einer Software.

Die Definition von OCP ist: *A software artifact should be open for extension but closed for modification*[Mar18, S. 70]

Wenn eine neue Erweiterungen in der bestehenden Software gemacht werden soll, soll nach dem **OCP** die bestehende Software minimal geändert werden (idealerweise gar nicht). Das lässt sich in der Kombination mit **SRP** (siehe Kapitel 2.4.1) erreichen, indem alle Module bzw. Klassen nur eine Aufgabe erledigen und somit nur einen Grund für die Änderungen besitzen.

2.4.3 Interface segregation principle

Das Interface-Segregation-Prinzip (dt. Schnittstellenaufteilungsprinzip) ist Begriff aus des objekt-orientierten Entwurfs. Er besagt dass eine große Schnittstelle (Interface) in mehrere kleine Schnittstellen aufgeteilt werden soll.

Beispiel: es gibt eine Schnittstelle, die **IDatabase** heißt. Diese Schnittstelle definiert folgende Methoden:

- getUser()
- setUser()
- getTransaction()
- setTransaction()
- getCharger()
- setCharger()

In der definierten Schnittstelle ist zu sehen, dass es sich in drei Gruppen aufteilen lässt (jeweils Methoden für Charger, User und Transaction).

1. getUser(), setUser()
2. getTransaction(), setTransaction(),
3. getCharger(), setCharger()

Nach **ISP** sollen besser drei statt eine Schnittstellen definiert werden. Zum Beispiel mit folgenden Namen:

1. IDatabaseUser: getUser(), setUser(),
2. IDatabaseCharger: getCharger(), setCharger(),
3. IDatabaseTransaction: getTransaction(), setTransaction()

2.4.4 Dependency inversion principle

Das Dependency inversion principle(dt. Abhängigkeits-Umkehr-Prinzip) oder DIP ist ein Prinzip beim objektorientierten Entwurf von Software. Es beschäftigt sich mit der Abhängigkeit von Modulen.

Im Allgemeinen wird das DIP beschrieben durch:

Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängig sein. Beide sollten von Abstraktionen abhängig sein.

Abstraktionen sollten nicht von Details abhängig sein. Details sollten von Abstraktionen abhängig sein.

Laut DIP, die Aussage “Modul A benutzt Modul B” soll nie direkt passieren, sondern über ein Interface I.



Abbildung 8: Schlechte laut DIP Abhängigkeit

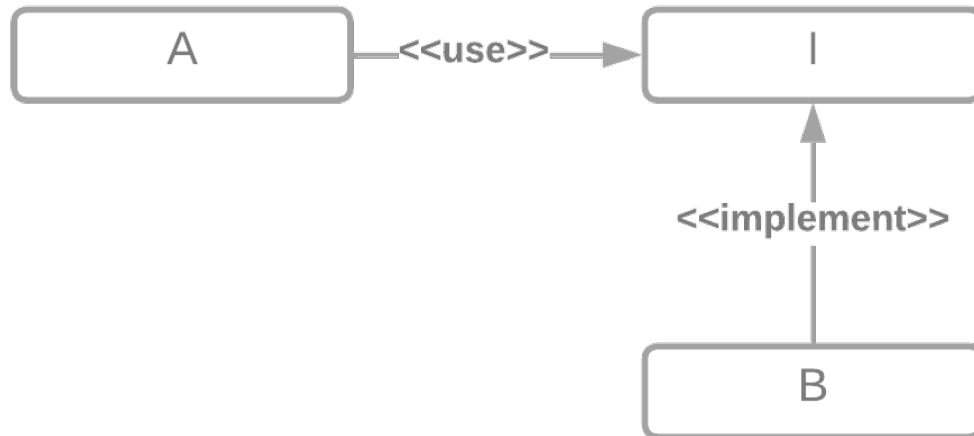


Abbildung 9: Gute laut DIP Abhängigkeit

2.4.5 Inversion of control

Inversion of control (IoC) ist ein Prinzip der Softwareentwicklung, bei dem die Ausführung vom geschriebenen Code an andere Stelle ausgelagert wird (z.B. ein Framework)[Aug].

Beispiel für IoC ist die Reaktion auf ein Ereignis mittels einem Callback. Es wird eine Methode (Callback) definiert, die an eine externe Stelle übergeben wird und beim Geschehen des Ereignisses zum späteren Zeitpunkt ausgeführt wird. Die Kontroller darüber, wann das Ereignis geschieht und wie die Zuordnung des Callbacks passiert, geht dadurch verloren.

Ein weiteres Beispiel von IoC ist "Dependency Injection".

2.4.6 Dependency Injection

In der Abbildung 10 ist ein Beispiel von einer Anwendung zu sehen, die die von der Konsole ankommenden Zahlen quadriert und das Ergebnis zurückgibt.

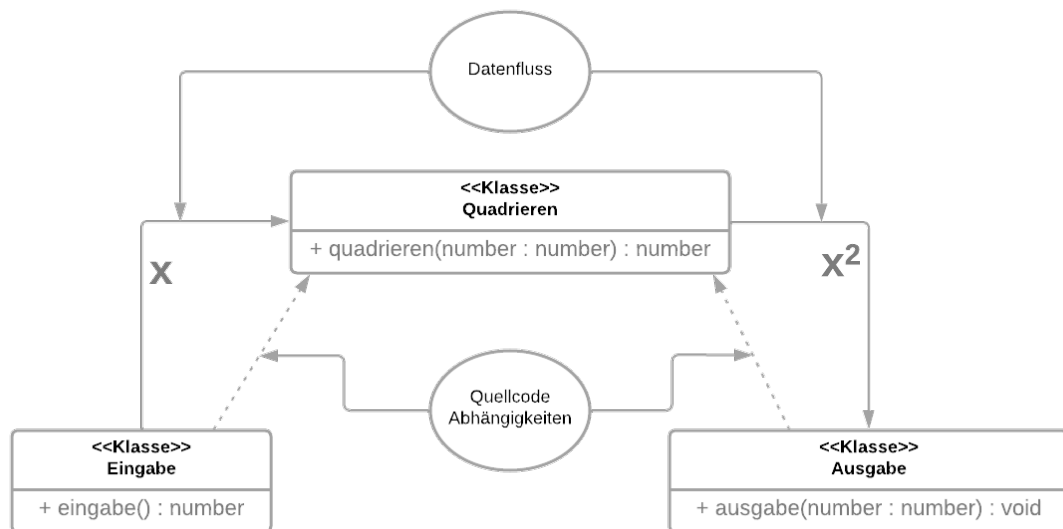


Abbildung 10: Datenfluss und Quellcode Abhängigkeiten

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

Wenn die Eingabeparameter bei der Ausgabe von **number** auf **string** geändert wird, muss die Ausgabe von **Quadrieren** von **number** auf **string** geändert werden. Dadurch kann auch eine weitere Kette an Änderungen im Programm ausgelöst werden. 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 lässt sich dafür "Interface" benutzen.

Die Lösung sieht so aus:

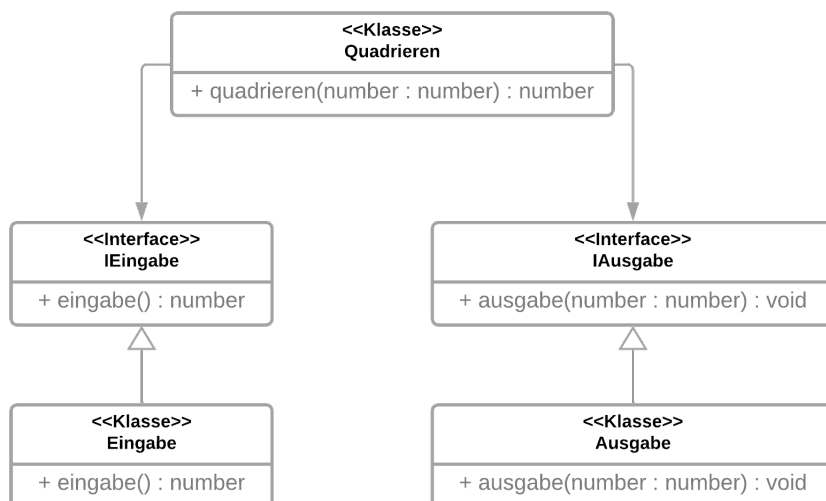


Abbildung 11: Entkopplung der Abhängigkeiten

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 alle Klassen über Interface miteinander verbunden sind, 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 auch ändern und dann müssen alle davon betroffenen Objekte entsprechend geändert werden, jedoch passiert das deutlich seltener als Änderung einer Klasse.

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

2.5 OCPP Kommunikation

Das OCPP (engl. Open Charge Point Protocol) ist ein Kommunikationsstandard zwischen der Ladesäule (Charging Station) und dem Server (Charging Station Management System) und entwickelt um jegliche Ladetechnik zu unterstützen.[OCA, Part 0, 3. Seite]

Mithilfe von standardisierter OCPP Kommunikation können die Software für die Ladesäule und den Server unabhängig voneinander geschrieben. Somit kann ein Server Ladesäulen von unterschiedlichen Herstellern gleichzeitig betreiben.

OCPP definiert eine Menge an Nachrichten, welche Inhalte und Struktur sie haben dürfen und in welcher Situation die Nachrichten abgeschickt werden dürfen. Es wird auch das Übertragungsprotokoll vorgegeben, in dem die Kommunikation stattfindet.

Für diese Arbeit sind folgende Eigenschaften des OCPP-Protokolls von Bedeutung:

- Übertragungsprotokoll ist WebSocket, dadurch wird die Implementation des **Ports** vorgegeben (siehe Kapitel 3.2)
- Inhalt und Struktur jeder Nachricht, gibt das Verhalten des **Adapters** vor (siehe Kapitel 3.2)
- Vordefiniertes Verhalten gibt eine Menge an **UseCases** vor. (siehe Kapitel 3.2)

3 Software Architektur

Es existiert keine einheitliche Definition einer Software Architektur. Verschiedene Autoren definieren es unterschiedlich.

Robert Martin definiert es als ein Gegenstand mit bestimmten Eigenschaften zu definieren.
The architecture of a software system is the shape given to that system by those who build it. The form of that shape is in the division of that system into components, the arrangement of those components, and the ways in which those components communicate with each other. [Mar18, S. 136]

Ralph Jonson definiert Software Architektur aus Sicht eines Projektes.
Architecture is the set of design decisions that must be made early in a project [Fowa]

Beide Definitionen schließen sich gegenseitig nicht aus und sollten gleichwertig bei der Planung einer Anwendung beachtet werden.

In dem ersten Teil des Kapitels wird die Softwarearchitektur aus Sicht eines Projektes betrachtet. Hierbei wird kurz beschrieben, welche Auswirkungen die Menge an investierter Zeit in die Architektur auf ein Projekt hat.

Im zweiten Teil des Kapitels, werden folgende Themen der Architektur im allgemeinen näher beleuchtet:

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

3.1 Software Architektur aus Sicht der Projektführung

3.1.1 Ziele der Software Architektur

Jede Software erfüllt bestimmte Anforderungen, die von Außen gestellt werden. Diese Anforderungen sind meistens nicht von Softwareentwicklern definiert und beziehen sich nicht auf Informatikgebiete (z.B. Banksoftware oder eine Smartphone Anwendung). Das Erfüllen von diesen Anforderungen ist das Ziel von jedem Softwareprojekt. Jedoch entstehen bei der Umsetzung viele Herausforderungen, die für Außenstehende nicht bekannt oder nicht relevant sind. (Auswahl einer Datenbank, Optimierung der Ressourcenverwendung, innere Struktur der Anwendung usw.). Die dabei getroffenen Entscheidungen haben eine große Auswirkung auf die Entwicklungszeit bzw. benötigten Ressourcen und somit kann das Ziel der Software Architektur wie folgt definiert werden:

The goal of software architecture is to minimize the human resources required to build and maintain the required system[Mar18, S. 5]

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

The strategy [...] is to leave as many options open as possible, for as long as possible [Mar18, S. 136]

Beispiele für solche Entscheidungen sind:

- 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 anderen Tätigkeiten, die nicht direkt das Programmieren betreffen, sind von den Entscheidungen in der Softwarearchitektur betroffen:

- Deployment (Verteilung) der Software.
- Maintenance (Wartung) der Software.

Deployment der Software beinhaltet die Kosten die durch die Verteilung 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.

3.1.2 Technische Schulden

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

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

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

- Unittests
- Dokumentieren
- Code Review

Beispiele für Technische Schulden sind:

- 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.

Beispiele an Entscheidungen und Lösungswegen, die durch klare Struktur der Software bereits getroffen, sind:

- die Kommunikationswege zwischen den Modulen sind vorgegeben
- 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 dokumentiert werden. Mit weniger Dokumentation können die gesuchten Informationen schneller gefunden werden.

Durch die einheitliche Bezeichnung der Teile des Modules ist es möglich aus dem Namen des Modules seine Aufgaben abzuleiten.

Daher ist es von Vorteil vor Beginn der Umsetzung des Softwaresystems, die oben genannten Aufgaben zu lösen, denn mit zunehmender Lebenszeit der Software nimmt die Änderungszeit zu.

Somit lassen sich die vorhandenen Ressourcen effizienter einsetzen.

3.1.3 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.

In der nachfolgenden Tabelle werden die Software, in die unterschiedliche Menge an Zeit investiert wurde.

	viel Zeit	wenig Zeit
Bugs fixen	schnell	langsam
Neue Funktionalität integrieren	schnell	langsam
Änderungen umsetzen	schnell	langsam
Einarbeitungszeit	gering	groß
Wahrscheinlichkeit alte Funktionalitäten zu beschädigen	klein	groß

Um diese Vorteile zu erhalten, muss regelmäßig Zeit in folgende Tätigkeiten investiert werden:

- Durchführen von Code Refactoring
- Überprüfen der Code Qualität
- Durchführen von Code Reviews
- Erstellen automatisierte Tests (Unit-, Integration- und Systemtests)
- Dokumentation aktualisieren
- Technischen Schulden gering halten

Nicht in jedem Projekt ist das Umsetzen von oben genannten Eigenschaften möglich, da meist entweder die Zeit oder das Budget oder beides zu gering ist. Mit Hilfe folgender Kriterien können einfacher Entscheidungen getroffen werden:

- Wann soll die erste Version der Anwendung vorhanden sein.
- Wie hoch ist die Anzahl der zur Verfügung stehenden Ressourcen
- Wie wahrscheinlich sind die Änderungen und Erweiterungen der Software
- Wie kritisch verschiedene Probleme und Ausfälle der Software sind

In der Abbildung 12 ist zu erkennen, dass auf längerer Distanz eine gute Softwarearchitektur deutlich mehr Funktionalitäten besitzt als eine Software mit schlechter Architektur. Im vorderen Zeitintervall führt allerdings die Software mit der qualitativ schlechteren Architektur.

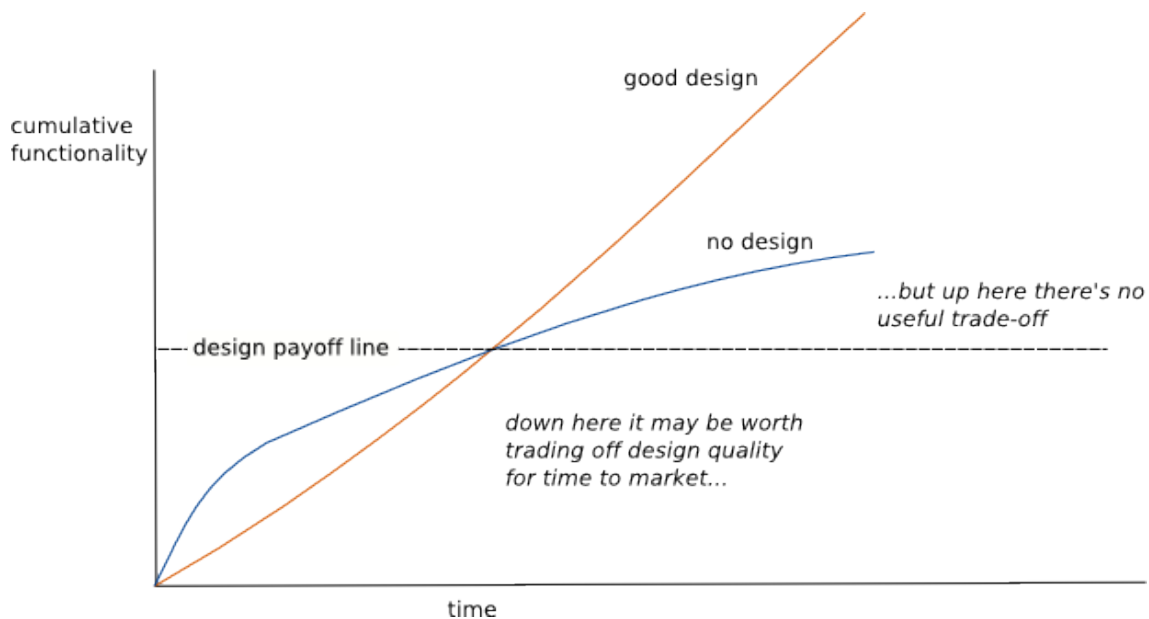


Abbildung 12: Vergleich einer guten und einer schlechten Softwarearchitektur ⁵

Dieser Verlauf ist bei Projektbeginn zu beachten. Es ist nicht von Vorteil bei einem Projekt, für das es nur eine Woche Zeit gibt, viel Zeit in die Architektur einzuplanen, die ohne jeglichen Funktionalitäten mehrere Wochen gebrauchen wird.

Wenn das Projekt regelmäßig weiterentwickeln wird, ist es vom Vorteil gleich viel Zeit in der Architektur zu investieren.

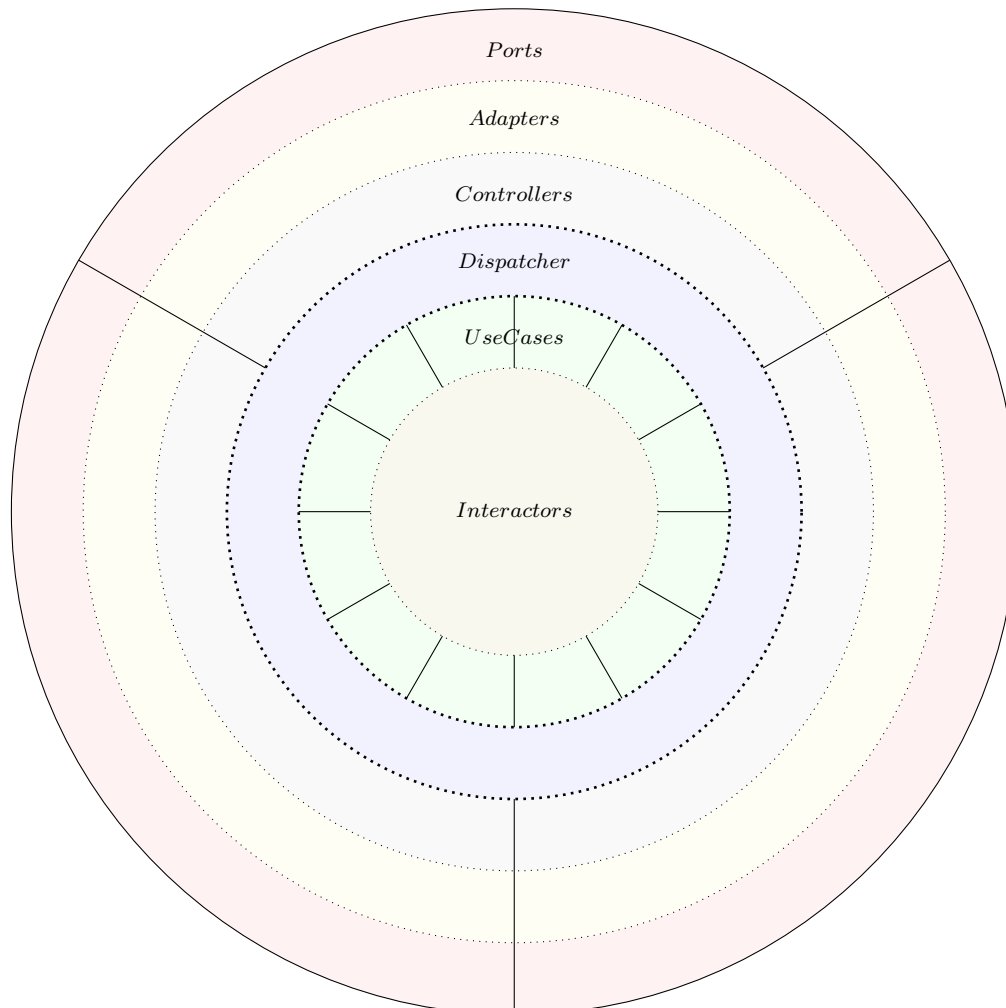
3.1.4 Testbarkeit der Software ist wichtig

Jeder 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 Kapitel bereits beschrieben wurde, werden die Bugs am schnellsten mit Hilfe von Unittests gefunden. Bei 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 einfach zu simulieren sein müssen. Dies erfordert eine Planung der Softwarearchitektur im Voraus, um diese Eigenschaft zu implementieren und damit im Laufe der Entwicklung, Zeit durch automatisierte Tests zu sparen.

⁵<https://martinfowler.com/bliki/DesignStaminaHypothesis.html>

3.2 Technische Umsetzung der Software Architektur

Im Kapitel werden Teile der Anwendung beschrieben, welche Eigenschaften sie besitzen und wie sie miteinander verbunden sind. Auch wird kurz gezeigt wie die Testbarkeit (Unit-, Integration- und Systemtests), Änderbarkeit und Erweiterbarkeit der Software ist. Ein wichtiger Teil der Beschreibung ist der Datenfluss im Programm.



—— Teile des Programms wissen nichts voneinander

Abbildung 13: Darstellung der umgesetzten Architektur mit 6 Schichten. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms

Folgende Komponenten sind im Programm definiert:

- Port - hat die Aufgabe die Schnittstelle nach Außen aufzubauen und die Verbindungen zu kontrollieren (z.B. WebSocket Server, Datenbank).
- Adapter - hat die Aufgabe ankommenden Ereignisse vom Port an den dazugehörigen Controller und in die andere Richtung zu übersetzen (z.B. Presenter im MVC 2.2.2)
- Controller - besitzt alle Informationen, die den Zustand jeweiliger Komponente (**Controller** + **Adapter** + **Port**) abbilden. Controller ermöglicht auch die dazugehörige Schnittstelle zu verwenden (z.B. eine Nachricht abschicken).

-
- UseCase - beschreibt den Vorgang beim Auslösen eines Ereignisses, welches sie abonniert haben. Die vordefinierten UseCases dürfen nur **Interactors** verwenden, um andere **Controller** anzusprechen.
 - Interactors - Eine atomare Operation im Programm (die Operation lässt sich nicht mehr sinnvoll im Rahmen der Anwendung aufteilen). Für jede Methode des **Controllers**, die vom **UseCase** benutzt wird, gibt es einen **Interactor**. Es gibt auch Möglichkeiten für komplexe **Interactoren**.

3.2.1 Abhängigkeiten im Programm

Die im Kapitel 3.2.1 beschriebene 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 schneller als mit realen Schnittstellen. Die Information über das Verhalten des Programms ist gleich.
- Die Innere Logik ist nicht an Schnittstellen gebunden, somit können alle Schnittstellen mit wenig Aufwand getauscht werden.

3.2.1.1 Port-Adapter-Controller

Am nächsten zu der **PAC**⁶ Struktur ist das Pattern **MVP**⁷. Dabei die Aufgaben von **View** (Benutzeroberfläche aktualisieren und Ereignisse empfangen) entsprechen den Aufgaben von **Port**. Die Aufgaben von **Presenter** (Ereignisse von der Benutzeroberfläche in das Datenmodell der Anwendung umwandeln) entsprechen den Aufgaben **Adapter**. Und die Aufgaben von dem Rest des Programms inklusiv **Controller** entsprechen den Aufgaben von **Model** (ankommende Ereignisse abarbeiten und das Ergebnis zurückgeben).

Der Unterschied zu **MVP** besteht darin, dass **MVP** im klassischen Sinne nur für die Benutzeroberflächen gedacht ist, während es in der hier beschriebenen Umsetzung für alle Schnittstellen benutzt wird. **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 komplette Architektur der Application.

Die Aufgabe von jeder **PAC**-Struktur ist die Steuerung von einem Aufgabengebiet (z.B. HTTP Schnittstelle oder OCPP Schnittstelle). Der Grund für das Aufteilen in 3 verschiedene Teile (Adapter, Port und Controller) ist **SRP** (Kapitel 2.4.1). Somit hat jeder Teil eine Aufgabe und lässt sich besser im Laufe der Zeit unterstützen.

⁶Port-Adapter-Controller

⁷Model-View-Presenter

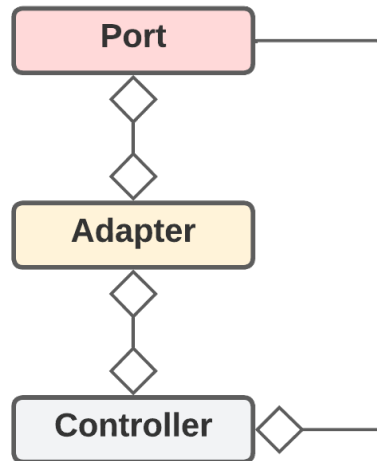


Abbildung 14: Objektdiagramm PAC

3.2.1.2 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 entsteht das Problem, dass jeder **Controller** mehrere Aufgaben übernimmt (z.B. Kontrollieren von **Port** und **Adapter** und enthält Anwendungslogik). Das widerspricht dem **SRP**, das in dem Kapitel 2.4.1 beschrieben ist.

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

Bei dieser Aufteilung besteht das Problem, dass beim Auftreten 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** und seine Aufgabe ist alle Abonnenten **UseCases** beim Auftreten eines Ereignisses zu informieren.

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

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

Das heißt, eine "Hülle" wird um jeder Methode gebraucht. Die Hülle heißt **Interactor**.

Wenn alles zusammengeführt wird, entsteht folgendes Objektdiagramm:

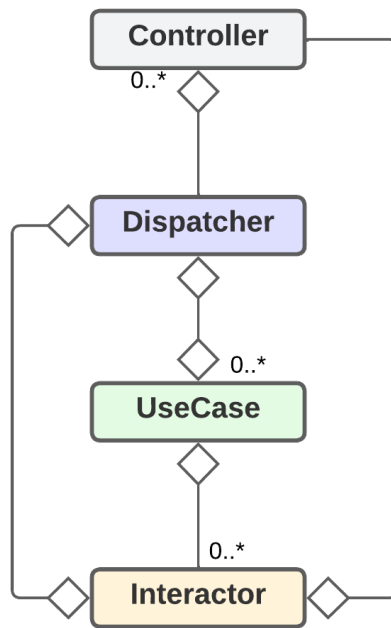


Abbildung 15: Objektendiagramm Controller-Dispatcher-UseCase-Interactor

3.2.1.3 Erstellen der Struktur

In den Kapiteln 3.2.1.2 und 3.2.1.1 werden die fertigen Strukturen beschrieben, diese Strukturen müssen am Anfang des Programms erstellt und miteinander verbunden werden.

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

- Erstellen aller Instanzen
- Verknüpfen aller Instanzen miteinander
- Starten aller Instanzen

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

- Kern (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 ist in Abbildung 16 dargestellt:

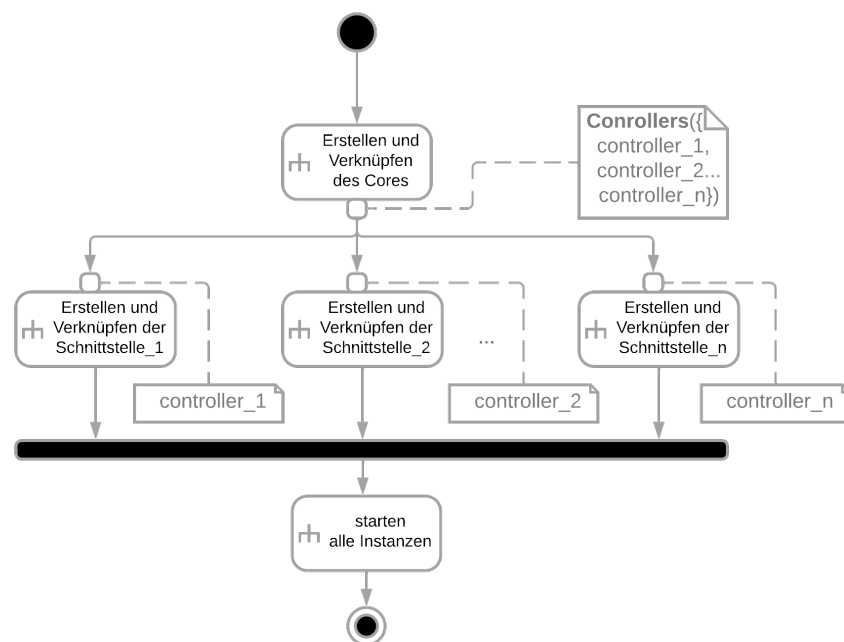


Abbildung 16: Ablaufdiagramm Erstellen der Struktur

Mit diesem Ablauf können mehrere Hauptprogramme 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 Hauptprogramme für verschiedene Datenbanken geben.

3.2.1.4 Utility Controllers

In jeder Anwendung gibt es Teile bzw. Module, die aus allen Komponenten des Programms erreichbar sein sollen (z.B. Logger Controller oder Datum Controller), und es gibt auch Klassen, die regelmäßig instanziiert werden.

Das Problem dabei ist, dass in das neue erstellte Objekt die Utility Controllers immer neben den anderen Argumenten mitübergeben werden müssen. Das erschwert die Lesbarkeit des Codes und kann eine Reihe an Änderungen an vielen Stellen mit sich ziehen, falls die Konstruktoren geändert werden.

Es gibt folgende Möglichkeiten das Problem zu lösen:

- Globale Objekte bzw. Instanzen (z.B. OOP Design Pattern **Singleton** 2.1.7)
- Initialisierung von den Entsprechenden Instanzen und Übergeben in dem Konstruktor (OOP Sprachen) oder mittels einer Settermethode.

Bei der ersten Implementierung besteht das Problem, dass die benutzten Utility Controllers nicht ersetzbar sind. D.h. die Module lassen sich 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 der Datenbank speichern). Jeder Test wird dadurch deutlich länger laufen. Da es auch reale 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 oder mittels einer Settermethode der Instanz übergeben werden. Die erste Möglichkeit führt zu einer längeren Parameterliste, die die Lesbarkeit des Codes erschwert. Die zweite Möglichkeit führt dazu, dass der Aufruf der Methode von dem Softwareentwickler vergessen werden kann.

Das Problem lässt sich zum Beispiel durch OOP design pattern **Factory** (Kapitel 2.1.8) umgehen, das die Utility Controllers bereits enthält und bei der Initialisierung entsprechend in die Konstruktor übergibt. Jeder Teil des Programms besitzt eine Referenz auf der Instanz der Fabrik, die ein kürzeres (kleinere Übergabeparameterliste) Interface für das Erstellen von verschiedenen Instanzen anbietet. Alle Module bzw. Klassen lassen sich somit auch mit Unittests abdecken, da die Utility Controllers entsprechend gemockt werden können.

In der Abbildung 17 ist die Verbindung jedes **Controllers** mit den Utility Controllers als Klassendiagramm dargestellt.

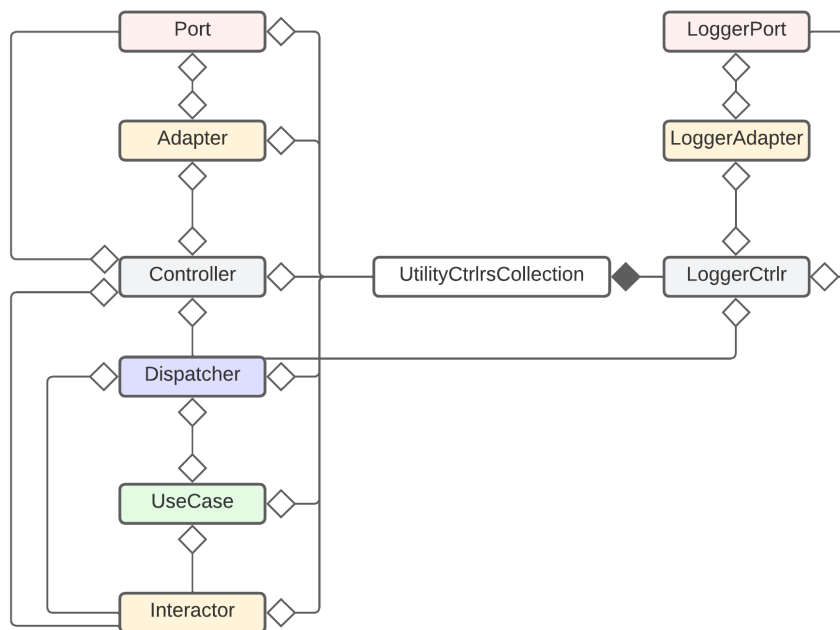


Abbildung 17: Objektendiagramm mit Utility Controllers

3.2.1.5 Verbindung der einzelnen Schichten miteinander und die Testbarkeit

Die früheren Kapitel beschreiben mittels Objektendiagramm die Struktur der Anwendungen nach dem Starten.

Die Schichten sind mittels **Dependency Injection** miteinander verknüpft.

Beispiel für **Port-Adapter-Controller** Verbindung:

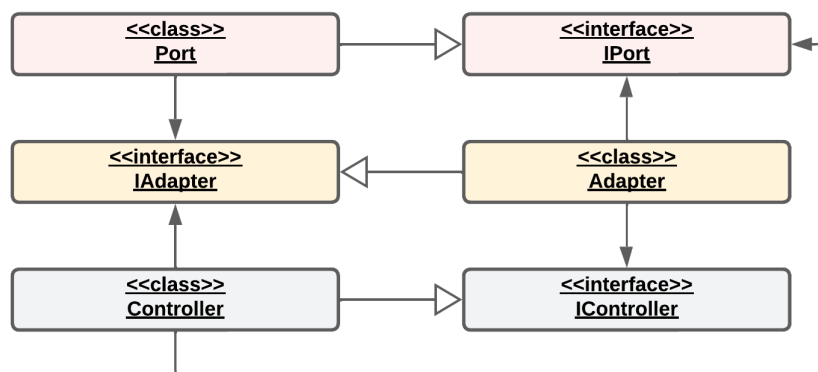


Abbildung 18: Klassendiagramm Port-Adapter-Controller

Beispiele für **Controller-Dispatcher-UseCase-Interactor** Verbindung:

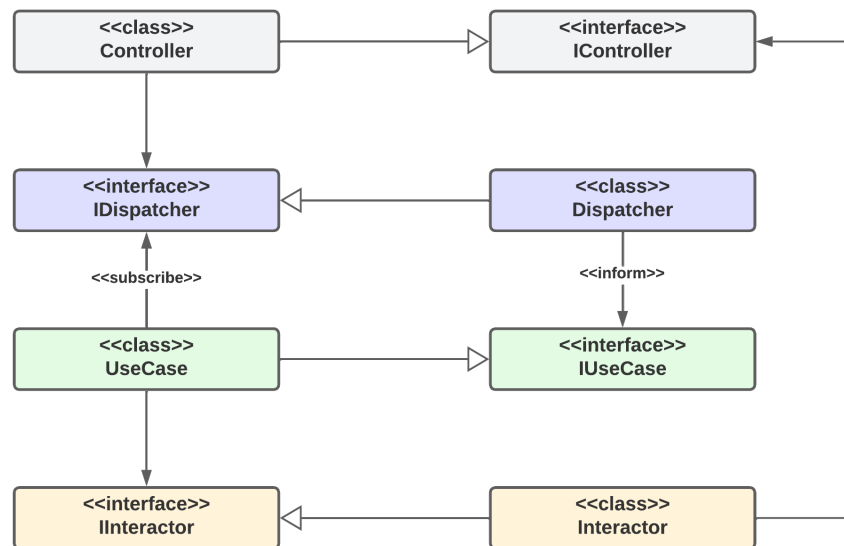


Abbildung 19: Klassendiagramm Controller-Dispatcher-UseCase-Interactor

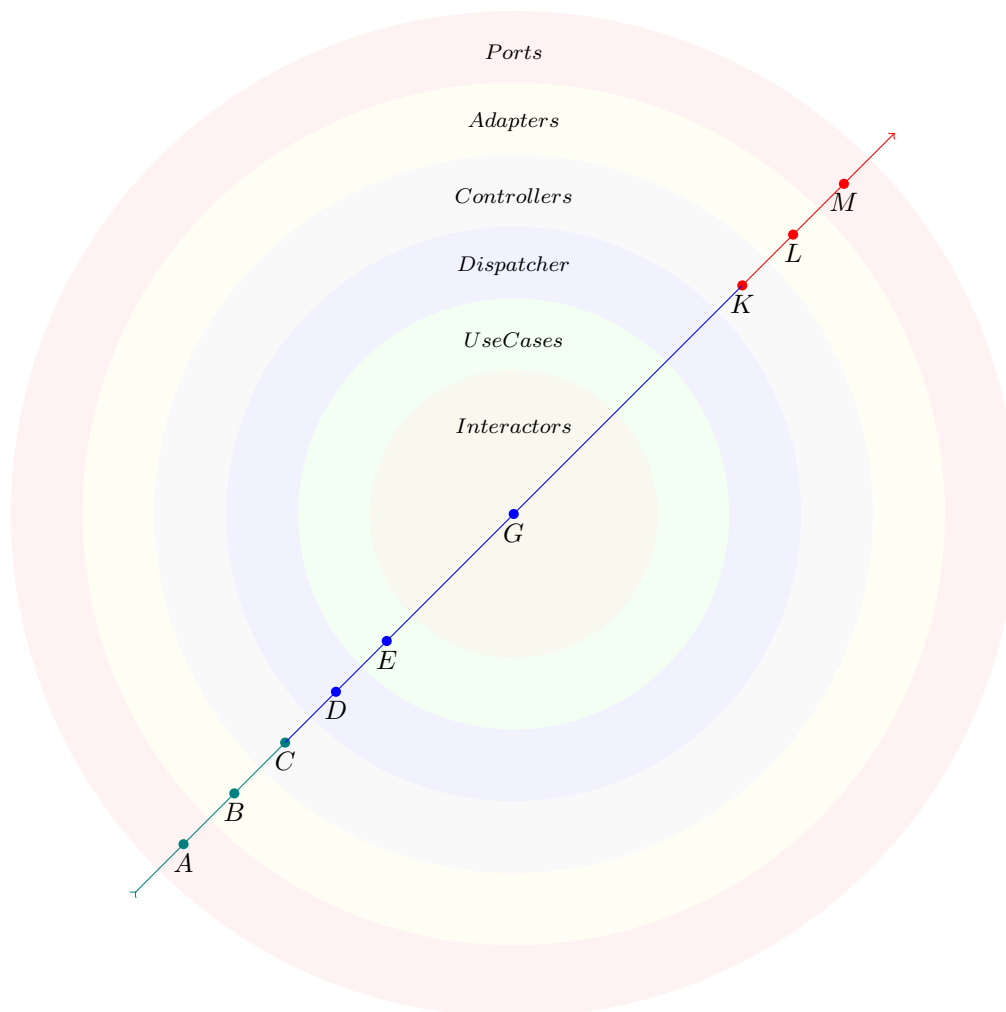
Im Klassendiagramm 18 und 19 sind alle Klassen miteinander über ein Interface verbunden. Dies ermöglicht leichte und schnelle Ersetzbarkeit der Schichten und somit lässt sich jeder Zustand der Umgebung um einer Schicht simulieren. Das ist Voraussetzung für Testbarkeit jeder einzelnen Schicht unabhängig von den anderen Schichten.

3.2.2 Datenfluss im Programm

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

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

Die Kombination aus drei Datenflüssen sieht so aus:



—— Teile des Programms wissen nichts voneinander

Abbildung 20: Darstellung der umgesetzten Architektur als **Clean Architecture** mit sieben Schichten. Pfeile repräsentieren mögliche Datenflüsse.

Jeder Punkt in der Darstellung repräsentiert eine Stelle, in der die Daten bearbeitet und weitergegeben werden.

3.2.2.1 Datenfluss 1

Darstellung des Datenflusses **1** als **sequencediagram**:

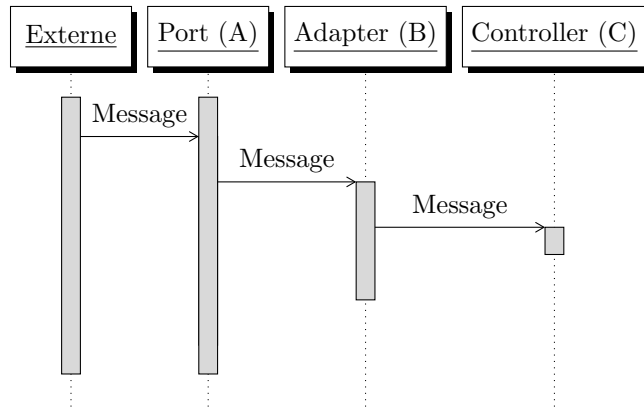


Abbildung 21: Sequencediagramm vom Datenfluss **1** Grün

In dem Fall wird an einem **Port** ein Ereignis ausgelöst (z.B. HTTP Nachricht), wird es weiter an **Adapter** weitergegeben, in dem die notwendigen Daten (z.B. Benutzerinformation) aus der ankommenden Nachricht separiert werden. Die gewonnenen Daten werden an **Controller** weitergegeben.

3.2.2.2 Datenfluss 2

Darstellung des Datenflusses **2** als **Sequencediagramm**:

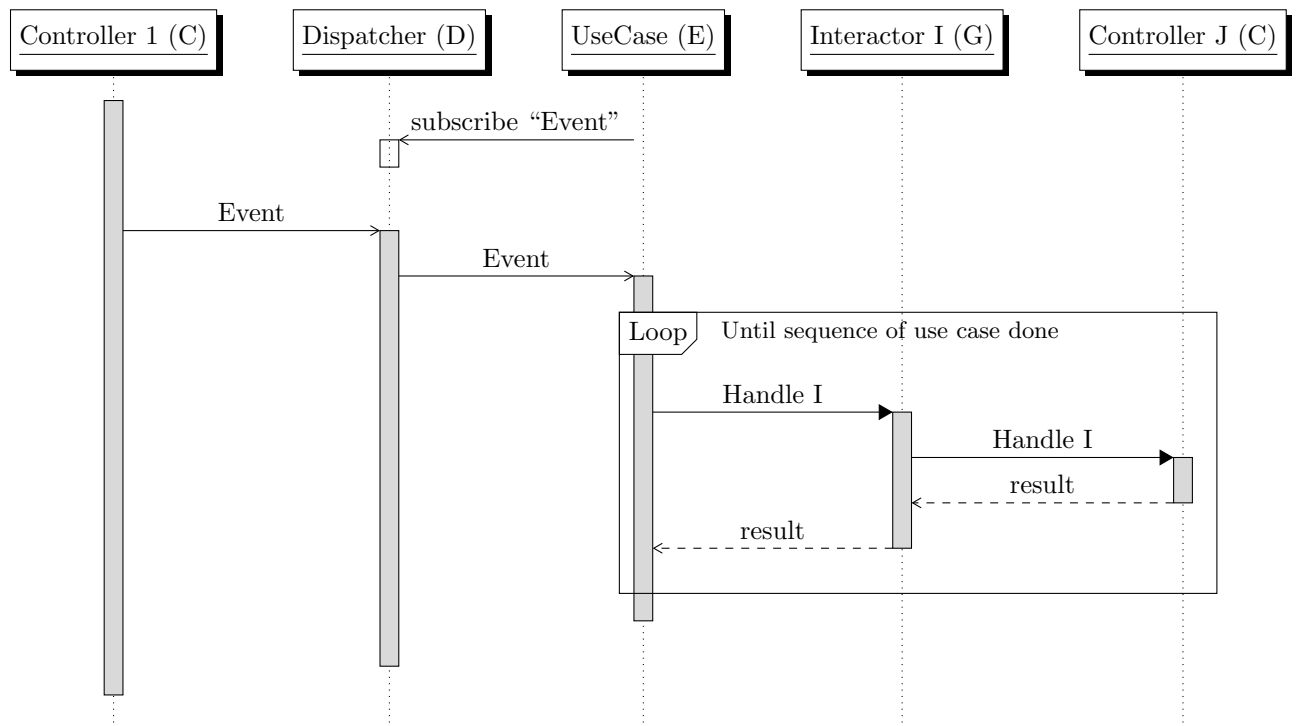


Abbildung 22: Sequencediagramm vom einfachen Datenfluss **2** Blau

Wenn ein Ereignis den **Controller** erreicht, wird der **Dispatcher** darüber informiert. Ein oder mehrere **UseCases** haben bereits dieses Ereignis beim **Dispatcher** abonniert. der **Dispatcher** informiert alle auf das Ereignis abonnierte **UseCases**. Jeder **UseCase** kann unabhängig voneinander sein eigenes Verhalten auf das Event definieren. **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 den **UseCase** zurückgegeben.

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

1. synchron - der Rückgabewert ist das Ergebnis der aufgerufenen Methode (siehe Darstellung 22)
2. asynchron - auf die dazugehörige Antwort vom Port wird gewartet(z.B. auf OCPP Response warten, wenn ein OCPP Request abgeschickt wird)

Darstellung der 2. Möglichkeit: Beim synchronen Funktionsaufruf wird der Rückgabewert, wie

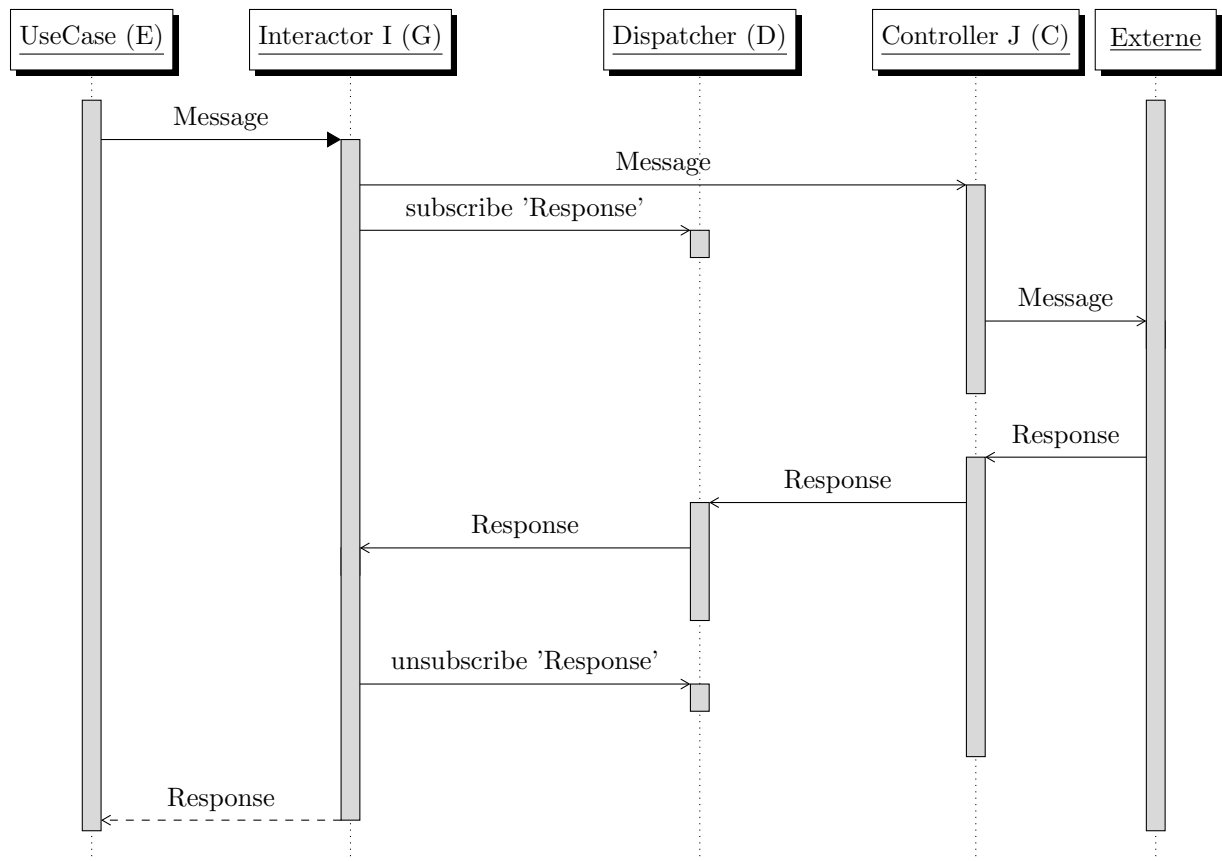
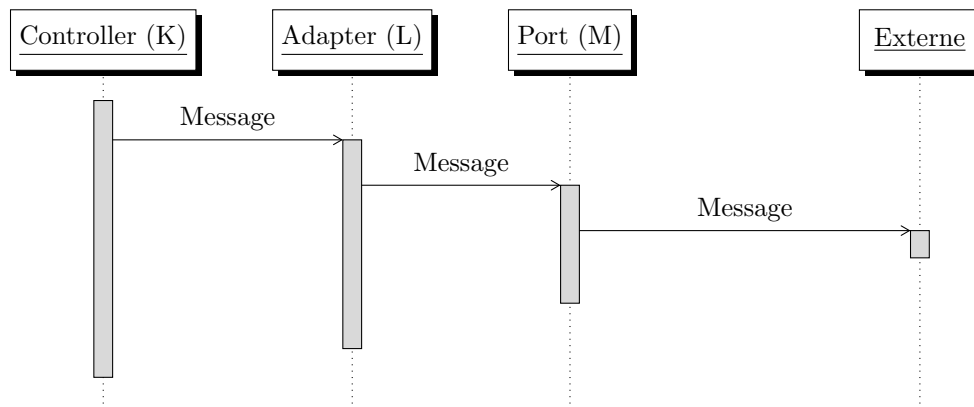


Abbildung 23: Sequencediagramm vom komplexen Datenfluss **2** Blau

in Abbildung 22 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 Abonnenten darüber informiert, unter anderem auch den **Interactor**. Der **Interactor** gibt diese Antwort als Rückgabewert des Funktionsaufrufs zurück.

3.2.2.3 Datenfluss 3

Darstellung des Datenflusses 3 als sequencediagram:



Darstellung der Datentransformation:

Controller - Adapter: Alle Informationen werden mittels Strukturen übergeben, die in jeweiliger Anwendung definiert sind. D.h. für verschiedene Anwendung, die unterschiedliche Strukturen definieren, wird das übergebene Objekt anderes aussehen.

Ein Beispiel der übergebenen Information sieht wie folgt aus:

```
1  OCPP20Message({
2      destination: {
3          charger : Charger({id : "some_unique_charger_id"})
4      },
5      message : Message({
6          name : "BootNotification",
7          type : "Response",
8          payload : BootNotification({
9              currentTime : Date(Thu Jul 28 2022 14:26:49 GMT+020
10             0),
11             interval : 30,
12             status : "Rejected"
13         })
14     })
15 })
```

Adapter - Port: Alle Informationen, die gesendet werden (in dem Fall "message"), werden in der verständlichen Form (sie muss nicht mehr in der Anwendung geändert werden) für den Port an Port weitergegeben. Falls es notwendig ist, müssen müssen alle Informationen über das Ziel der Nachricht weitergegeben werden, sodass der Port die entsprechende Verbindung zuordnen kann. Die Struktur der übergebenen Information wird durch den Port und das benutzte Übertragungsprotokoll bestimmt.

Ein Beispiel der übergebenen Information im Falle einer OCPP Nachricht:

```
1  {
2      destination : {
3          chargerId : "some_unique_charger_id"
4      }
5      message : "[3, 'message_id_of_request', {currentTime : 'Thu Jul
6              28 2022 14:26:49Z', interval : 30, status : 'Rejected'}]"
7  }
```

3.2.2.4 Logging

Ein wichtiger Bestandteil jeder Software ist das Logging von unterschiedlichen Ereignissen in der Software. Das Ziel vom Logging ist später von den Softwareentwicklern verschiedene Fehler in der Software so schnell wie möglich zu finden und zu beseitigen.

Dafür muss es möglich sein mit den Logs die Fehler so genau wie möglich in der Software zu lokalisieren (welche Komponente oder welche Methode den Fehler hervorgerufen hat) und welche Ereigniskette den jeweiligen Fehler hervorgerufen hat.

In dem Kapitel 3.2.2 sind alle möglichen Wege basierend auf der beschriebenen Struktur im Kapitel 3.2.1 für die ankommenden Ereignissen in der Software beschrieben.

Die nachfolgende Abbildung 24 zeigt den kompletten Ablauf beim Geschehen eines Ereignisses. Eine mögliche Systematisierung des Loggings in der Appliaction wäre:

- Aufzeichnung in jeder Komponente den Inhalt des ankommenden Ereignisses auf
- Aufzeichnung in jeder Komponente den Inhalt des ausgehenden Ereignisses auf
- Aufzeichnung alle Komponenten auf, an die das Ereignis weitergegeben wird

Somit lassen sich die Fehler auf die Komponente genau lokalisieren, d.h. ein entsprechender Unittest lässt sich schreiben, der diesen Fehler abdeckt, bzw. einen Integrationstests, da der Ablauf des Ereignisses auch bekannt ist.

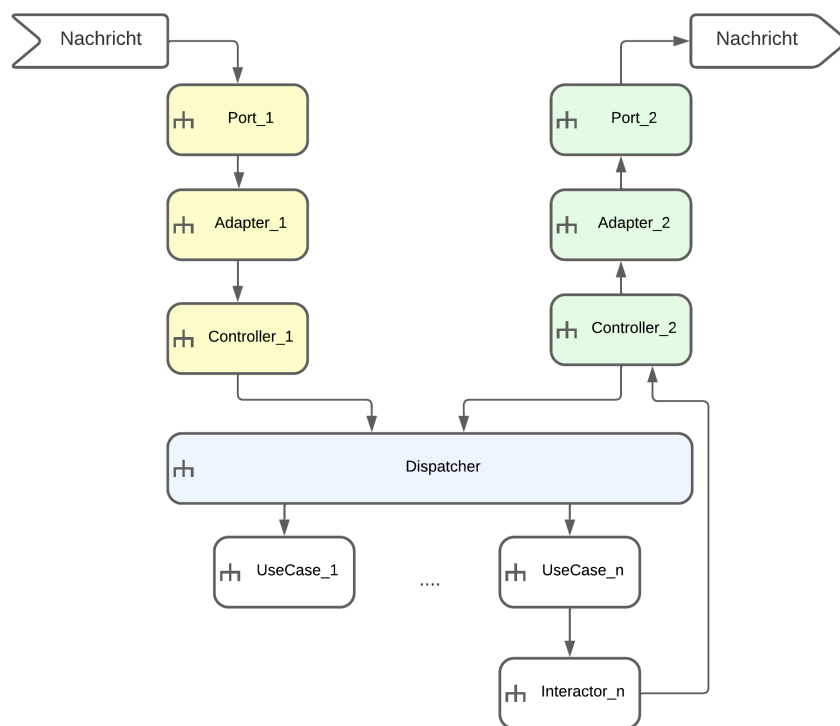


Abbildung 24: Kompletter Datenfluss

3.2.3 Erweiterung der Funktionalitäten

Ein wichtiges Ziel der Architektur ist die leichte Erweiterung der Funktionalitäten der Anwendung. Im Kapitel werden mehrere Szenarien betrachtet und entsprechend beschrieben, an welchen Stellen was geändert bzw. hinzugefügt werden soll. Bei allen Änderungen soll das Hauptprogramm angepasst werden. (z.B. eine entsprechende Instanz wird gestartet)

3.2.3.1 Erweiterung des Verhaltens für ein Ereignis

Bei diesem Szenario wird betrachtet, dass ein Ereignis, für das neues Verhalten geändert werden soll, an **Dispatcher** ankommt und entsprechend an alle dafür verantwortlichen **UseCases** weiterleitet. Es gibt hier zwei Möglichkeiten:

1. bestehenden **UseCase** um die neue Funktionalität erweitern.
2. neuen **UseCase** erstellen, das das gleiche Ereignis handelt.

Bei der ersten Möglichkeit ist das komplette Verhalten für ein Ereignis an einem Ort definiert und entsprechend mit Unittests abdecken lässt.

Die zweite Möglichkeit streut das Verhalten für ein Ereignis im Projekt. Das verbessert die Lesbarkeit des jeweiligen Teils und erfüllt **OCP** (Open-Closed-Prinzip, siehe Kapitel 2.4.2), jedoch die Schwierigkeit bringt alle solche Teile im Projekt zu finden. Das Gesamtverhalten lässt sich erst mittels einem Integrationstest abdecken, was eine mögliche Fehlersuche erschweren kann. Für den Fall, dass das neue Verhalten unabhängig von dem bestehenden Verhalten ablaufen soll ist es eine gute Möglichkeit.

3.2.3.2 Eine bestehende Schnittstelle um ein Ereignis erweitern

In dem Fall wird nur der Datenweg vom **Port-Adapter-Controller** betrachtet und welche Änderungen da entsprechend gemacht werden müssen.

Der Weg vom **Port** zum **Controller**

- **Port** - in dem Teil sollen keine Änderungen gemacht werden.
- **Adapter** - in dem Teil soll das Validieren des ankommenden Ereignisses hinzugefügt werden. Normalerweise würde das Ereignis als ein unbekanntes Ereignis markiert und dann weitergeleitet.
- **Controller** - in dem Teil sollen keine Änderungen gemacht werden.

Der Weg vom **Controller** zum **Port**

- **Controller** - das Absenden soll ermöglicht werden, evtl. auch ein **Interactor** hinzufügen
- **Adapter** - um das Umwandeln des Ereignisses soll erweitert werden.
- **Port** - in dem Teil sollen keine Änderungen gemacht werden.

3.2.3.3 Neue Schnittstelle hinzufügen

Wenn die Anwendung um eine neue Schnittstelle erweitert werden soll, müssen folgende Teile erstellt werden:

- **Port**
- **Adapter**
- **Controller** evtl. auch **Interactoren** hinzufügen

Jede Schnittstelle bringt meistens auch mehrere **UseCases** mit, die das neue Verhalten implementieren, und entsprechend neue Ereignisse, damit die **UseCases** gestartet werden können. Dies bedingt eine mögliche Erweiterung (hängt von der gewählten Programmiersprache ab) des **Dispatchers**, um die neue Ereignisse.

3.3 Anwendung

3.3.1 Anbindung in eine andere Anwendung als eine Komponente

Im Kapitel wird betrachtet, dass die Software ein Teil der anderen Anwendung ist. Zum Beispiel kann ein Kern umgesetzt werden, der dann in jeweiligen Anwendungen angepasst wird oder es handelt sich nur um eine Komponente für eine andere Anwendung.

Grundsätzlich lässt sich die Abbildung 43 folgendeweise vereinfachen:



Abbildung 25: Vereinfachte Darstellung

Und bei einer Standalone Anwendung gibt es eine Main-Methode, die diese Struktur erstellt.

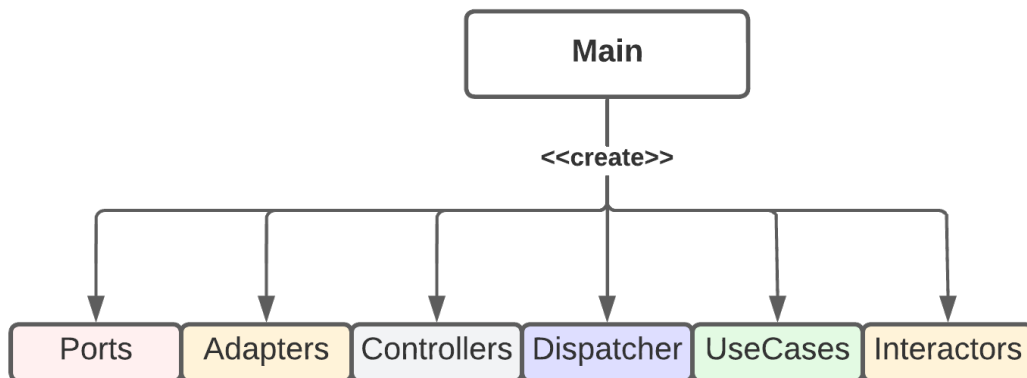


Abbildung 26: Vereinfachte Darstellung einer Standalone Anwendung

Der Datenfluss lässt sich so darstellen:



Abbildung 27: Vereinfachte Darstellung einer Standalone Anwendung

Wenn es in einer anderen Anwendung verwendet wird, braucht die Komponente eine Fassade (Kapitel 2.1.6), damit die wichtigen Teile der Komponente zugreifbar sind und der Rest verborgen bleibt. Die Fassade baut die gesamte Struktur der Komponente auf.

Eine fertige Komponente, die in die anderen Anwendungen sich integrieren lässt:

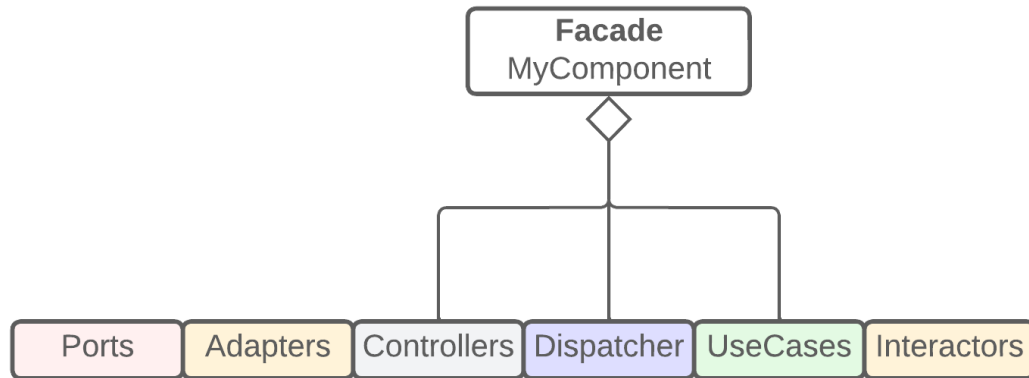


Abbildung 28: Vereinfachte Darstellung der Architektur als Komponente

Laut der Darstellung 28 hat die Anwendung nur den Zugriff auf drei Teile der Komponente:

- **Controllers** - um die Zustände des jeweiligen Controllers abzufragen und zu ändern.
- **Dispatcher** - um alle Ereignisse in der Komponente abzufangen.
- **UseCases** - um das Verhalten auf gewisse Ereignisse ändern zu können.

Die Komponente kann bestimmte Ereignisse selber abarbeiten und die restliche Anwendung wird darüber nicht informiert oder das Ereignis weiterleiten, dass es von der Anwendung selbst abgearbeitet wird. Die Komponente wird von der eigentlichen Anwendung unabhängig entwickelt, es kann passieren, dass der Datentyp des Ereignisses von der Komponente nicht mit dem Datentyp der Anwendung übereinstimmt. Ein **Adapter** wäre eine mögliche Lösung für das Problem. Das bedeutet, dass die Komponente nur von dem **Port** der jeweiligen Anwendung benutzt wird.

Die Vereinfachte Darstellung der Anwendung mit der Komponente:

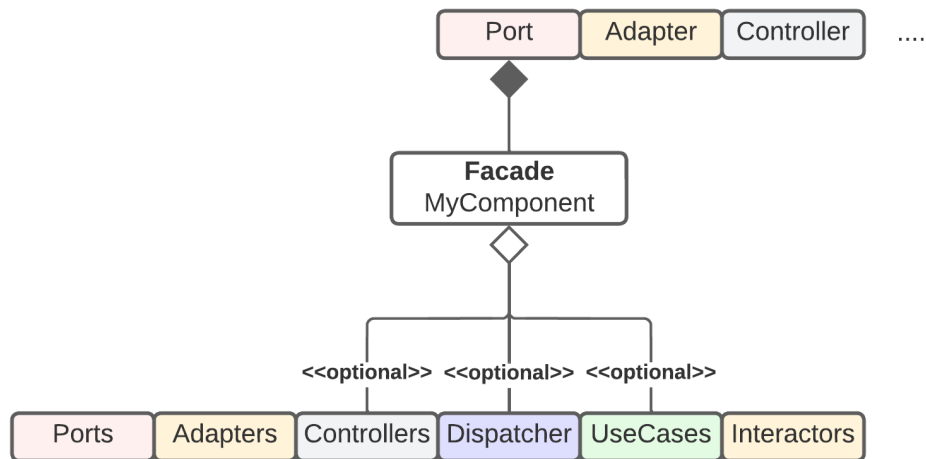


Abbildung 29: Vereinfachte Darstellung einer Standalone Anwendung mit der Komponente

Der Datenfluss in der Anwendung:

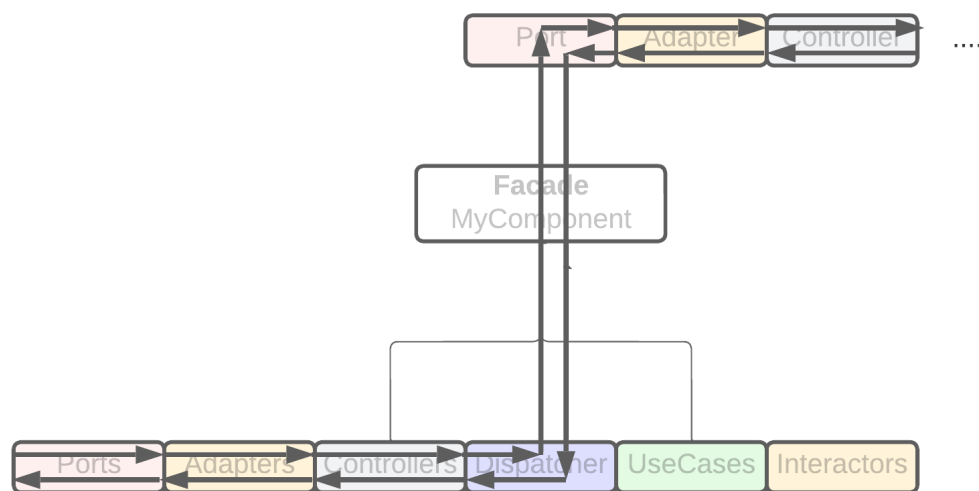


Abbildung 30: Vereinfachte Darstellung des Datenflusses in einer Anwendung mit Komponente

3.3.2 Framework und Bibliothek

Im Kapitel wird der Unterschied zwischen einem Framework und einer Bibliothek beleuchtet.

Ein Framework ist eine Softwareplattform, die die Struktur und Architektur des künftigen Softwareprodukts bestimmt. Jedes Framework enthält ein vorgefertigtes “Gerüst” – die Vorlagen, Standardmodule und APIs, die dem Entwickler zur Verfügung stehen.⁸

Definition der Bibliothek ist:

Bibliothek ist eine Sammlung ähnlicher Objekte, die zur gelegentlichen Verwendung gespeichert werden.⁹

Der Unterschied zwischen dem Framework und Bibliothek besteht darin, dass beim Framework der geschriebene Code vom Framework aufgerufen wird und bei der Bibliothek der geschriebene Code den Code von der Bibliothek (Inversion Of Control, Kapitel 2.4.5).¹⁰

Beispiele für Bibliotheken sind zum Beispiel alle Implementierungen von Netzwerkprotokollen. das komplette Verhalten muss vom Clientcode definiert werden.

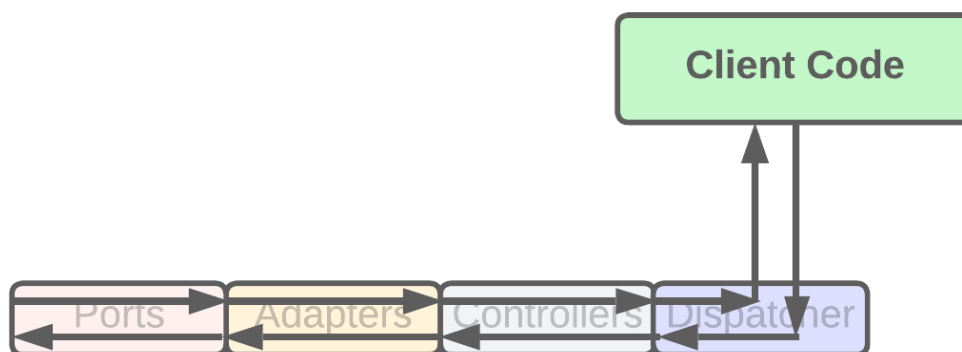


Abbildung 31: Vereinfachte Darstellung des Datenflusses bei einer Bibliothek

In der dargestellten Architektur könnte es zum Beispiel heißen, dass das Defaultverhalten (Use-Cases Schicht) erweitert oder geändert wurde.



Abbildung 32: Vereinfachte Darstellung des Datenflusses bei einem Framework

Vergleich des Client Codes von der Bibliothek und Framework:

	Framework	Bibliothek
Typen, Konstruktoren	vom Framework	Selbst geschrieben
Struktur der Anwendung	vom Framework vorgegeben	Selbst geschrieben
Anteil im Projekt	groß	gering

⁸<https://it-talents.de/it-wissen/framework/>

⁹<https://www.computerweekly.com/de/definition/Bibliothek-Library#:~:text=In%20der%20Informatik%20bezeichnet%20der,und%20p>

¹⁰<https://martinfowler.com/bliki/InversionOfControl.html>

4 OCPP Server

4.1 Aufgabenbeschreibung

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. (mit und ohne einer Datenbank)
- 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.

Der Server soll mehrere Bereiche unterstützen, die in Abhängigkeit von jeweiligem Zustand des Bereiches das Verhalten des Servers ändern.

Die gewünschte Bereiche, die der Server unterstützen soll, sind:

- Ladesäuleverwaltung soll die angefragten Verbindungen von bekannten Ladesäulen akzeptieren bzw. von unbekannten Ladesäulen ablehnen.
- Benutzerverwaltung soll die Authentifizierung von bekannten Benutzern an Ladesäulen zulassen bzw. von unbekannten Benutzern ablehnen.
- Preisverwaltung soll unterschiedliche Preise für unterschiedliche Benutzer verwalten.
- Ladevorgangverwaltung soll die Zustände jedes Ladevorgang verwalten können.
- Benutzeroberflächeverwaltung soll ermöglichen alle Bereiche des Servers manuell ändern zu können.
- Ladesäulekommunikationverwaltung soll die Kommunikation mit den Ladesäulen mittels OCPP Protokoll ermöglichen.
- Loggingverwaltung soll die wichtigen Ereignisse im Server aufzeichnen.

4.1.1 Standalone Server mit Datenbank

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

Die nachfolgende Abbildung 33 ist ein Übersichtdiagramm der Standalone-Anwendung mit Datenbank.

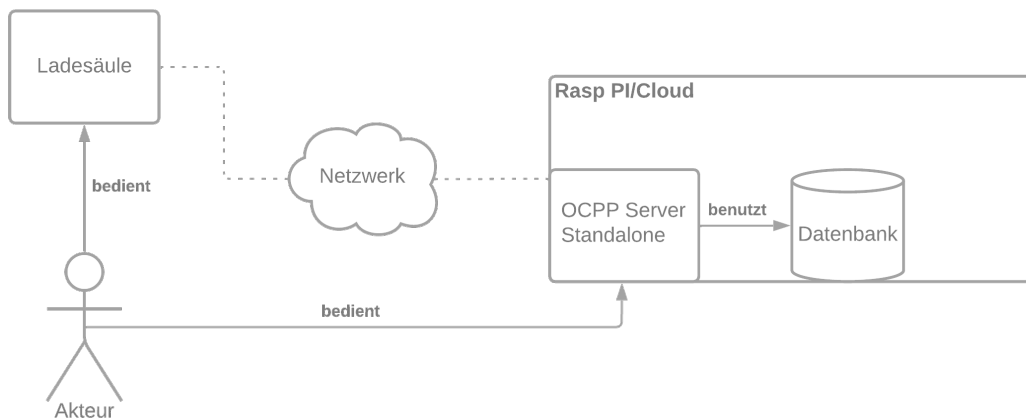


Abbildung 33: Übersichtdiagramm der Standalone-Anwendung mit Datenbank

4.1.2 Standalone Server ohne Datenbank

Der Standalone Server ohne Datenbank soll für die manuellen Tests der Software der Ladesäule benutzt werden. Die Ladesäulesoftware und der Server sollen auf einem Rechner laufen und das Ziel der Einsetzung des Servers ist die Funktionalitäten der Ladesäulesoftware zu überprüfen. Der Server soll schnell zu installieren sein und keine externen Abhängigkeiten haben.

Die Anforderungen an Standalone Server ohne Datenbank sind:

- Datenbank darf nicht benutzt werden.
- Gleiche Funktionalität wie bei dem Standalone Server mit Datenbank (siehe Kapitel 4.1.1)

Die nachfolgende Abbildung 34 ist ein Übersichtdiagramm der Standalone-Anwendung ohne Datenbank.

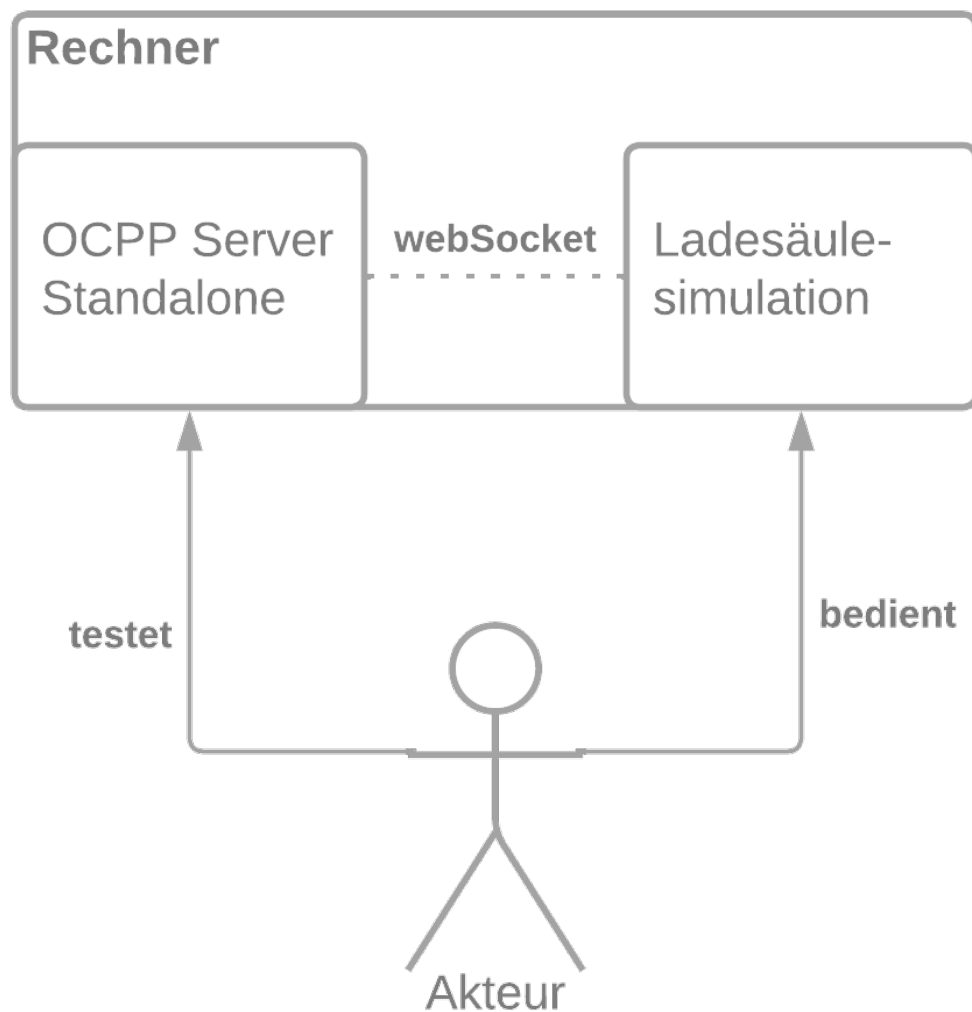


Abbildung 34: Übersichtdiagramm der Standalone-Anwendung ohne Datenbank

4.1.3 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.)

Die nachfolgende Abbildung 35 ist ein Übersichtdiagramm der Framework-Anwendung.

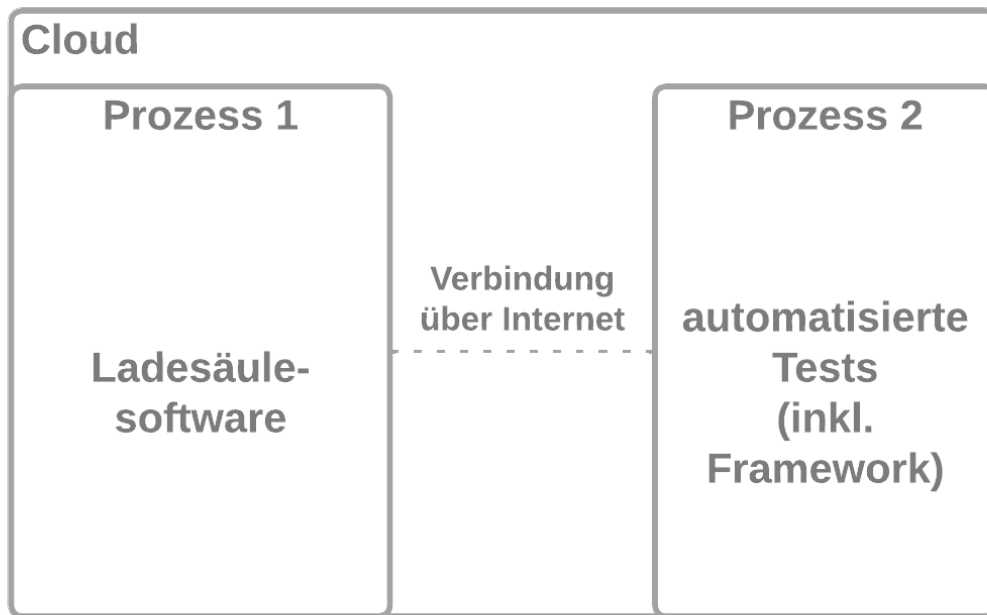


Abbildung 35: Übersichtdiagramm der Framework-Anwendung

4.1.4 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

Die nachfolgende Abbildung 36 ist ein Übersichtdiagramm der ERK Software.

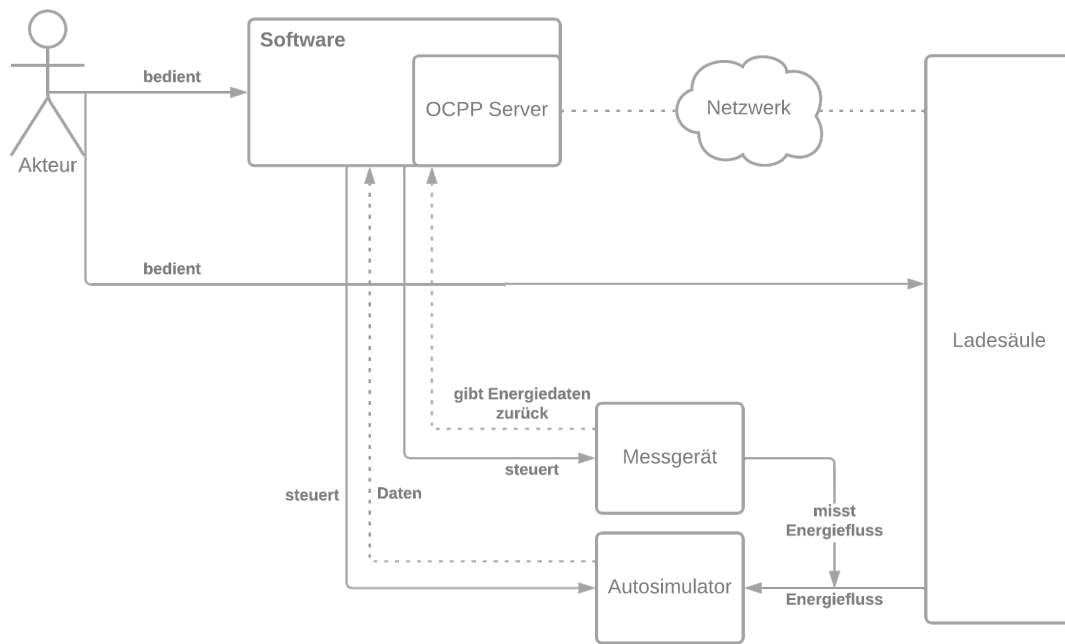
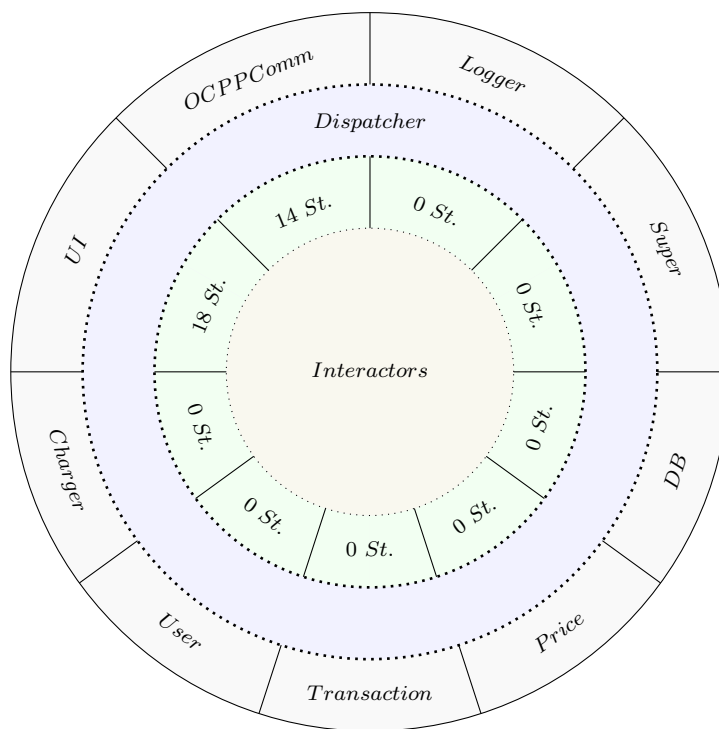


Abbildung 36: Übersichtdiagramm der ERK-Software

4.2 Allgemeine Lösung



—— Teile des Programms wissen nichts voneinander

Abbildung 37: Darstellung der umgesetzten Architektur. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms

In der Abbildung 37 ist der innere Kreis der Architektur dargestellt. Der innere Kreis definiert das Verhalten der Anwendung ohne sich auf bestimmte Schnittstellen (z.B. Datenbank oder HTTP) zu binden. Die Struktur des inneren Kreises des OCPP Servers ist im Kapitel 3.2.1.2 beschrieben. Der Datenfluss des inneren Kreises ist im Kapitel 3.2.2 beschrieben. Der innere Kreis wird bei allen Anwendungen ohne jeglichen Änderungen benutzt.

4.2.1 Controllers

1. **Super-Controller** besitzt die Aufgabe, den Zugriff auf die anderen **Contollers** zu ermöglichen. In der Lösung wird das nur von **UI-Controller** verwendet, somit besteht die Möglichkeit das Programm über **HTPP-Schnittstelle** zu verwalten. (z.B. Zustände von den **Controllern** abzufragen).
2. **OCPP-Controller** kontrolliert den Websocket-Server, ermöglicht das Abschicken und Aufnehmen von Netzwerk- und OCPP Ereignissen.
3. **UI-Controller** kontrolliert den HTTP-Server, ermöglicht das Aufnehmen von HTTP Nachrichten.
4. **Logger-Controller** kontrolliert den Zugriff auf das Filesystem, ermöglicht das Logging im Programm.

-
5. **DB-Controller** kontrolliert den Zugriff auf die Datenbank, indem die Verbindung erstellt wird.
 6. **Chager-Controller** kontrolliert alle Informationen der bekannten Ladesäulen, ermöglicht folgende Sachen: neue Ladesäule anzulegen, Ladesäuleinformation ändern bzw. löschen, Überprüfen ob gegebene Ladesäule bekannt ist.
 7. **User-Controller** kontrolliert alle Informationen über die bekannten Nutzer der Ladesäulen, ermöglicht folgende Sachen: neuen Nutzer anzulegen, Nutzerinformation ändern bzw. löschen, Überprüfen ob gegebene Nutzer bekannt ist.
 8. **Transaction-Controller** kontrolliert alle Informationen über die Transactionen (Ladevorgang) an den Ladesäulen, ermöglicht folgende Sachen: neue Transaction starten und stoppen.
 9. **Price-Controller** kontrolliert alle Informationen über den Preis, ermöglicht folgende Sachen: Preis für den Nutzer anzulegen und ändern, Defaultpreis zu ändern.

4.2.2 Dispatcher

Der **Dispatcher** verteilt die aufgetretenen in **Controllers** Ereignisse an die UseCases und Interactors, für die das Ereignis zutrifft.

Jedes Ereignis ist eindeutig durch folgende Eigenschaften definiert:

- Name des **Controllers**, in dem das Ereignis aufgetreten ist.
- Richtung des Ereignisses (vom **Port** zum **Controller** oder vom **Controller** zum **Port**)
- Name des Ereignisses
- Inhalt des Ereignisses

Es besteht die Möglichkeit Ereignisse im **Dispatcher** mit Filtern zu abonnieren. Zum Beispiel über alle Ereignisse vom bestimmten Controller informieren.

4.2.3 UseCases

Jedes Ereignis in jedem **Controller** wird an Dispatcher weitergegeben, der alle abonnierten **UseCases** darüber informiert.

Insgesamt gibt es 32 **UseCases**. Davon 14 reagieren auf die Ereignisse vom “OCPComm” Controller und 18 reagieren auf die Ereignisse vom “HTPPComm” Controller. Alle anderen **Controller** definieren keine eigene **UseCases** auf verschiedene Ereignisse.

Das Defaultverhalten aller Anwendungen ist in den **UseCases** beschrieben. Es besteht aber die Möglichkeit das Verhalten zur Laufzeit zu ändern, dies wird zum Beispiel beim Framework aktiv benutzt.

4.2.4 Interactors

Der allgemeine Ablauf eines **Interactors** in der Anwendung:

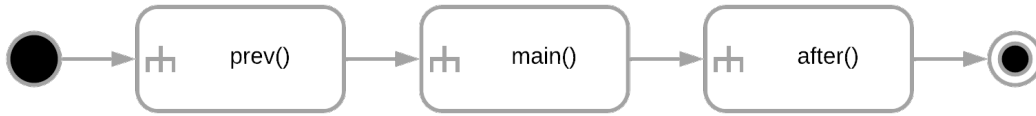


Abbildung 38: Ablauf eines Interactors

Aufbau des **Interactors**:

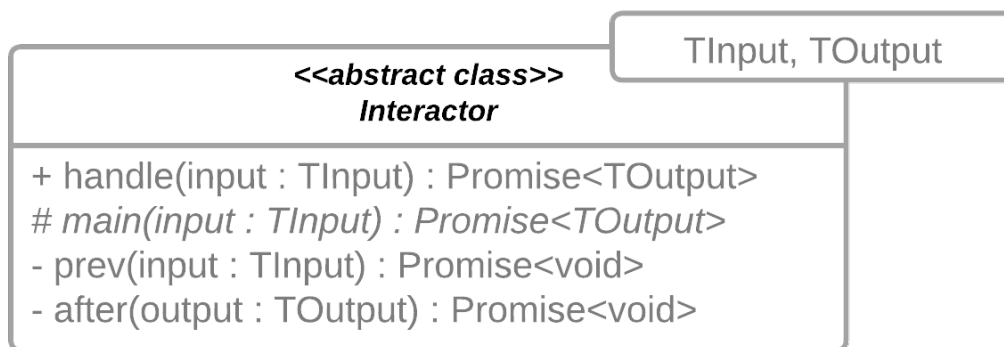


Abbildung 39: Klassendiagramm eines Interactors

Fast alle **Interactoren** hüllen nur Einzelmethode(n) von **Controllern** ab.

Es gibt nur ein **Interactor**, das komplexes Verhalten besitzen. Das ist:

- Abschieken einer OCPP Request Nachricht. Der Rückgabewert ist die Antwort auf die geschickte Nachricht (OCPP Response). Siehe Abbildung 23.

4.3 Anwendung der allgemeinen Lösung

Jede umgesetzte Lösung benutzt das im Kapitel 4.2 beschriebene Verhalten und Struktur.

Der Unterschiede zwischen den einzelnen Lösungen sind:

- **Ports**, die von **Controllers** benutzt werden
- Fassade der Anwendung (betrifft nur Library und Framework)

Jeder **Controller** außer “Super”-Controller kann eine reele Schnittstelle benutzen, falls sie in der Anwendung gebraucht wird. In jeder Anwendung kann eine Schnittstelle von mehreren **Controllers** verwendet werden. Das lässt sich zum Beispiel mittel **ISP** (siehe Kapitel 2.4.3) umsetzen.

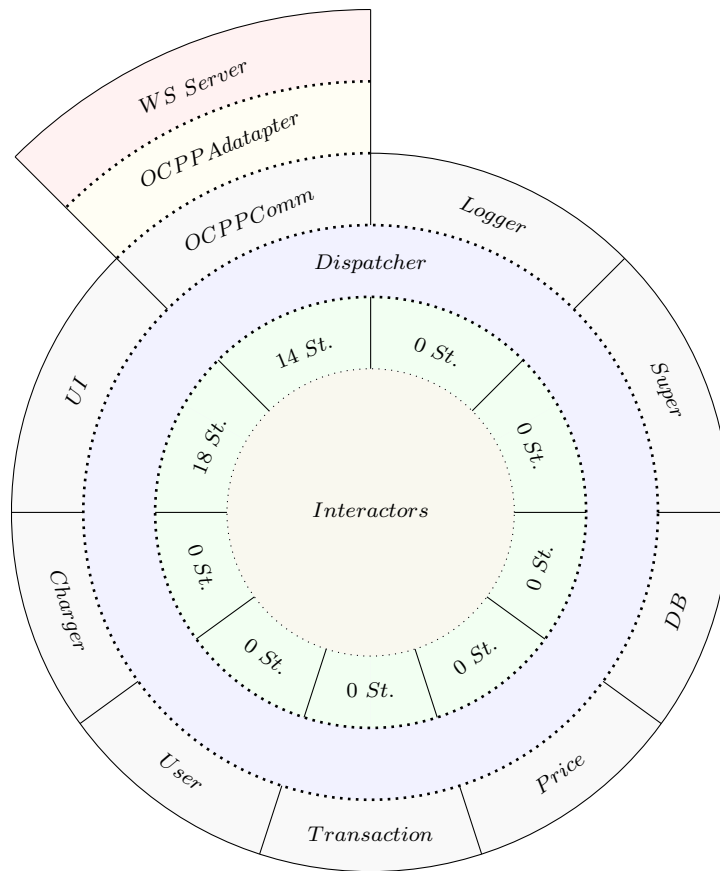
Jede Schnittstelle kann drei Versionen haben:

- reele Schnittstelle
- Schnittstelle ist gefälscht. Zum Beispiel statt Datenbank Cache verwenden. Das Verhalten der gefälschten Schnittstelle ist wichtig.
- Schnittstelle wird nicht benutzt, sie muss aber trotzdem übergeben werden, damit Parameterliste erfüllt wird. Das Verhalten der Schnittstelle ist unwichtig.

Da jede Schnittstelle drei Komponenten benötigt (**Controller**, **Adapter** und **Port**) und jede Komponente nur einen Aufgabenbereich erfüllt bzw. nur einen Grund für die Änderung besitzt, lassen sich einzelne Komponente ohne Änderungen an bestehender Software für verschiedene Anwendung getauscht werden. Damit ist **OCP** erfüllt (siehe Kapitel 2.4.2). Das wird bei der Umsetzung von allen Aufgaben aktiv benutzt, d.h. wenn es an einem Teil einer Anwendung irgendwas geändert wird, wird andere Anwendung, die dieses Teil nicht benutzt, nicht von den Änderungen betroffen.

Jede Anwendung soll OCPP Kommunikation serverseitig können. Aus diesem Grund wird OCPP Schnittstelle (in dem Fall WebSocket-Server) immer real und gleich sein.

Zusammengefasst sieht das gleiche Teil jeder Anwendung so aus:



—— Teile des Programms wissen nichts voneinander

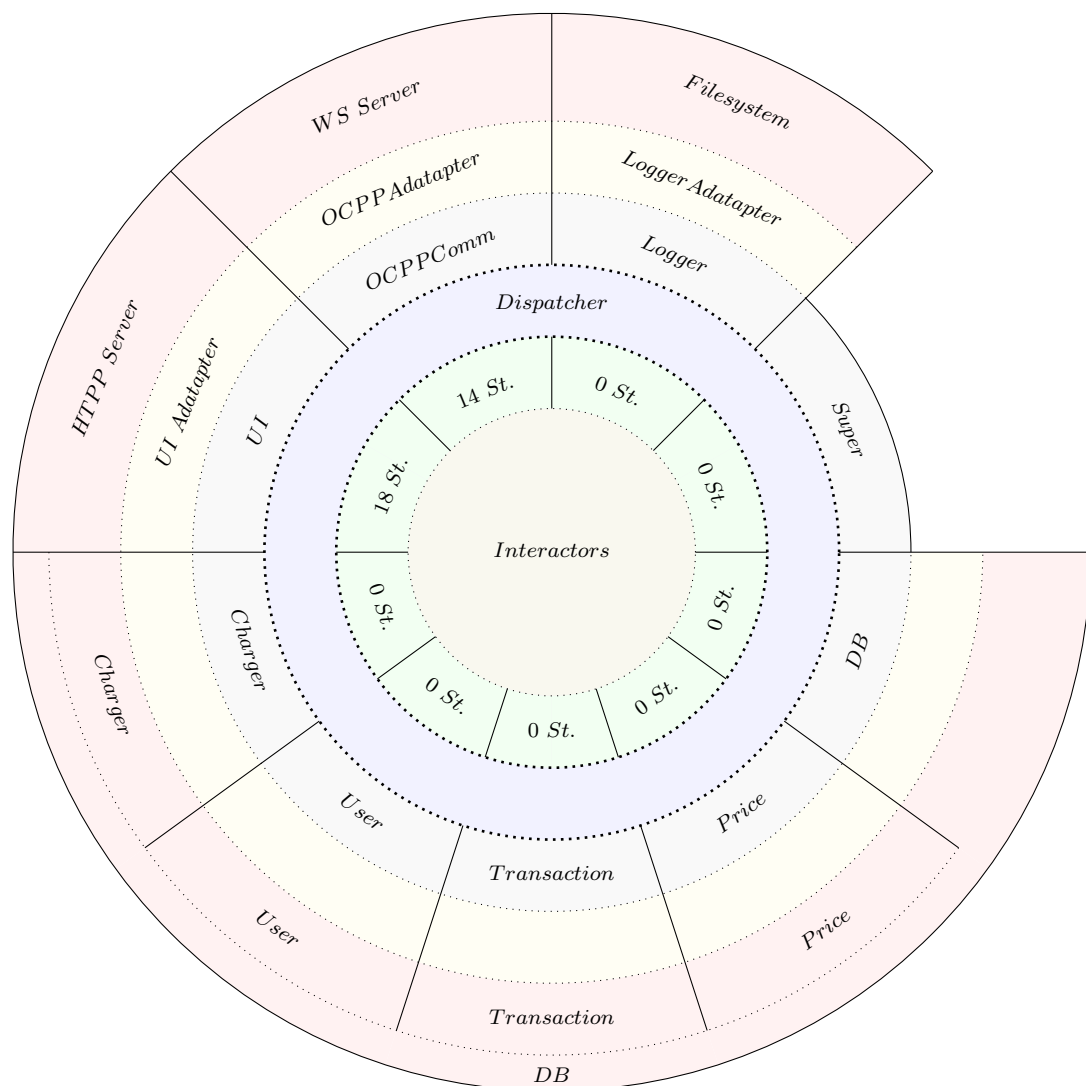
Abbildung 40: Darstellung der umgesetzten Architektur. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms

4.3.1 Standalone mit Datenbank

Die Anwendung soll dauerhaft auf einem Server laufen und die Informationen dürfen nicht verloren gehen. Somit sollen alle wichtigen Daten und die Logs gespeichert werden. Eine Standalone-Anwendung mit Datenbank benötigt, dass alle gebrauchten Schnittstellen reel sind. Das sind:

- Datenbank (um verschiedene Daten zu speichern, wird von Charger-, User-, Transaction- und Price-Controllern verwendet)
- HTTP Server (für Benutzerinteraktionen)
- Filesystem (für Logging)
- Websocket Server (für OCPP Schnittstelle)

Die Anwendung lässt sich wie folgt darstellen:



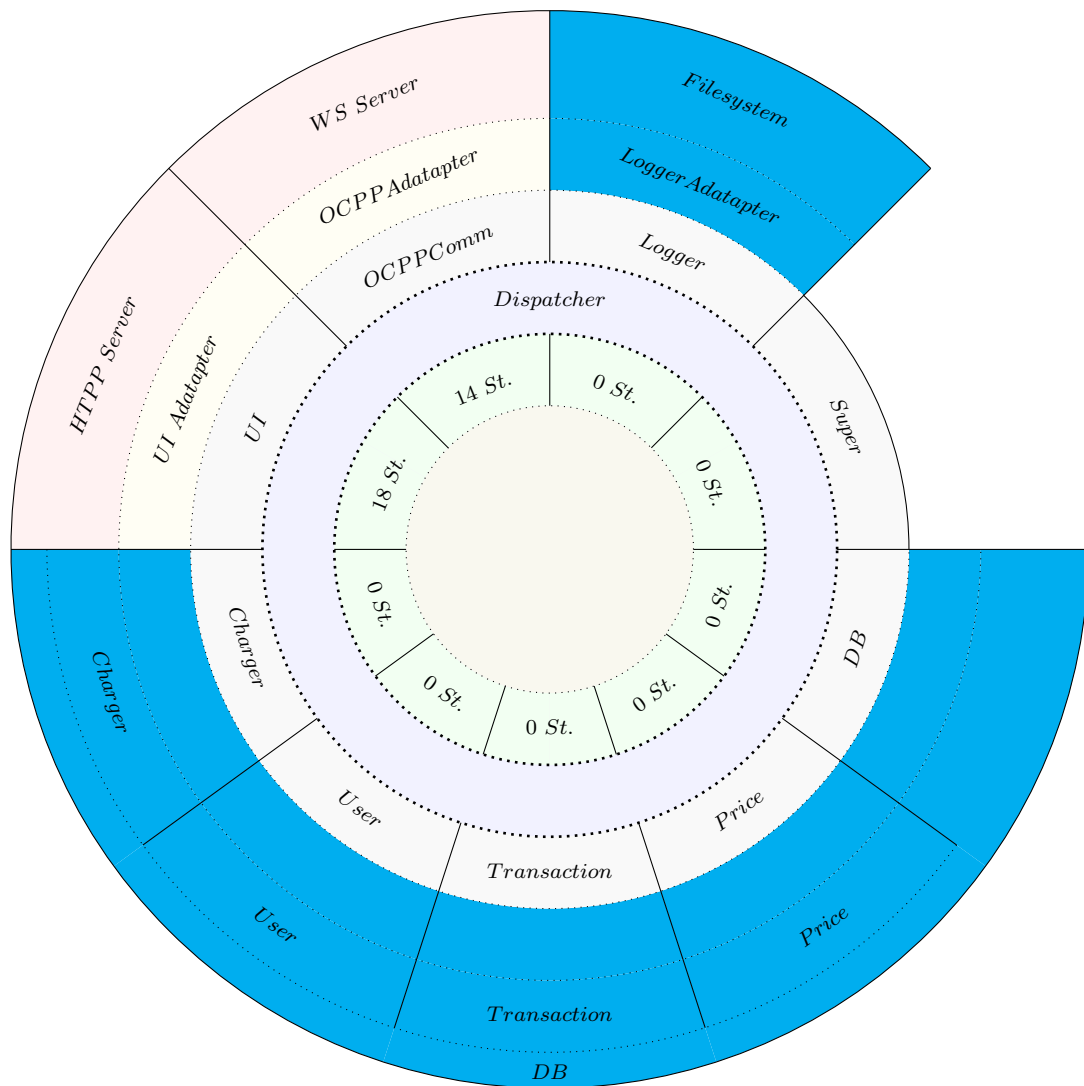
— Teile des Programms wissen nichts voneinander

Abbildung 41: Darstellung der umgesetzten Architektur. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms

4.3.2 Standalone ohne Datenbank

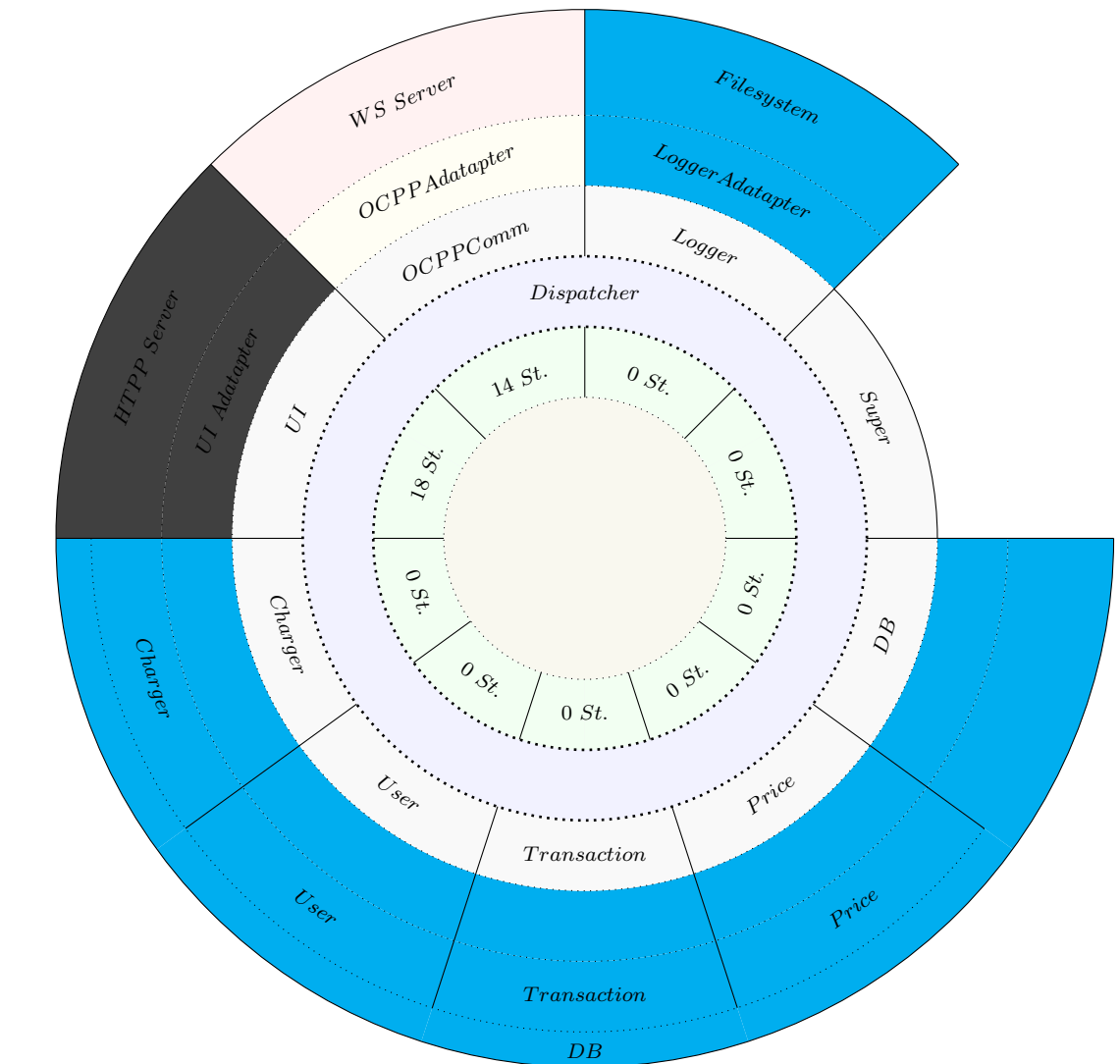
Die Anwendung soll auch in der Lage dauerhaft laufen, sie braucht keine externen Abhängigkeiten, die extra installiert bzw. gepflegt werden sollen. Die Anwendung soll genauso verhalten, wie die Anwendung mit einer realen Datenbank, somit sollen Filesystem- und Datenbank-Schnittstelle gefälscht werden.

Die Anwendung lässt sich wie folgt darstellen (Die blaue Farbe stellt gefälschte Komponenten dar):



— Teile des Programms wissen nichts voneinander

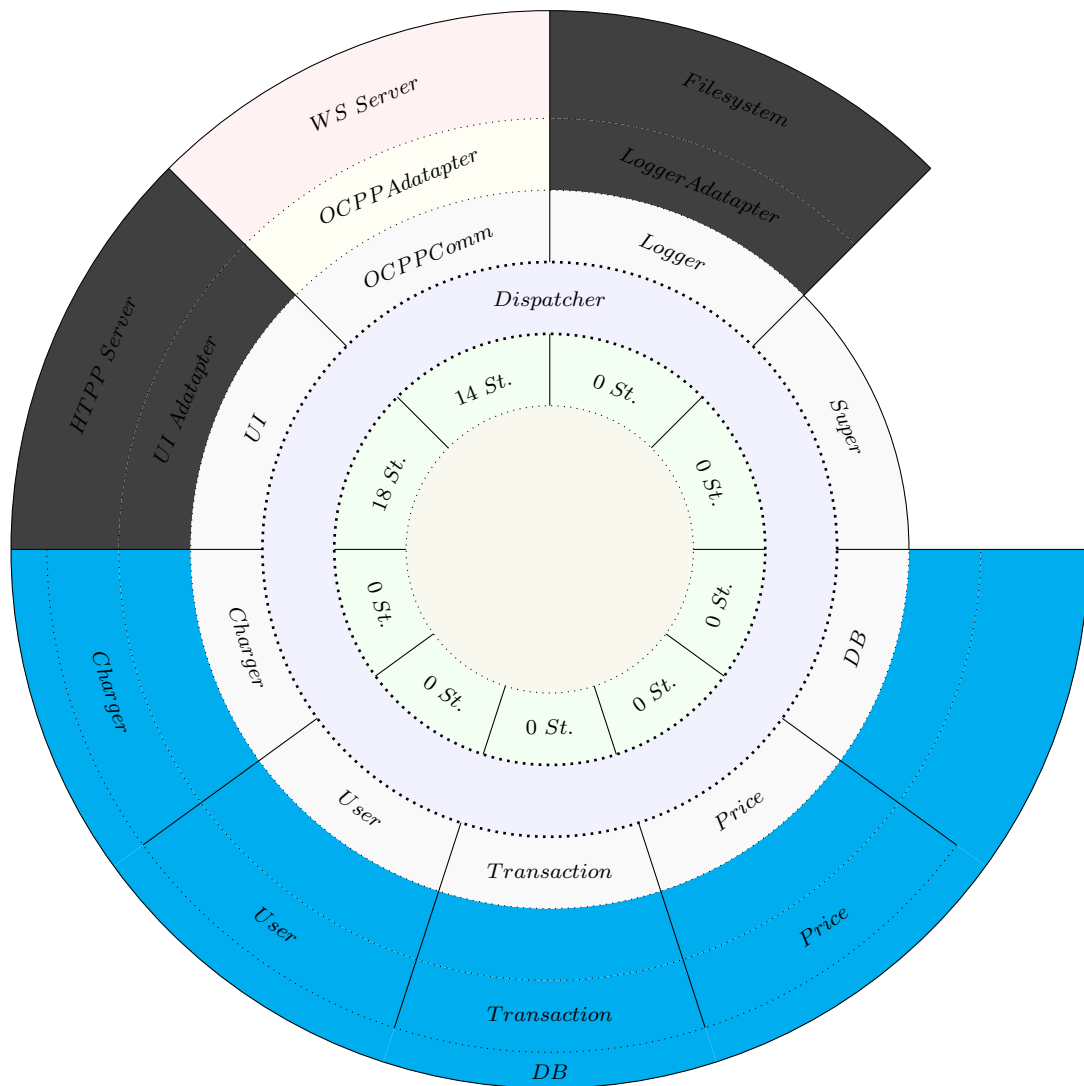
Abbildung 42: Darstellung der umgesetzten Architektur. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms



—— Teile des Programms wissen nichts voneinander

Abbildung 43: Darstellung der umgesetzten Architektur. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms

Die Anwendung lässt sich wie folgt darstellen (Die blaue Farbe stellt gefälschte Komponenten dar, die schwarze Farbe stellt gefälschte Komponente dar):



—— Teile des Programms wissen nichts voneinander

Abbildung 44: Darstellung der umgesetzten Architektur. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms

Das Framework wird von den Dritten benutzt. Aus diesen Grund benötigt es ein gutes Interfaces, das zugleich als Fassade (siehe Kapitel 2.1.6) zu der Struktur in der Abbildung 44 darstellt.

Die Struktur als Klassendiagramm: **HIER SOLL DIE Struktur des FRAMEWORKS SEIN**

Das Interface besteht aus folgende Methoden:

- start
- stop
- addUseCase
- rewriteUseCase
- waitOf
- waitOfNextEvent
- sendMessage

Jeder Test lässt sich in vier Phasen unterteilen, jede Methode bzw. Funktion des Frameworks soll einer dieser Phase zugeordnet werden. In der nachfolgenden Abbildung 45 ist eine Zuordnung der Methoden zu jeweiliger Phase im Form eines Ablaufdiagramms zu sehen:

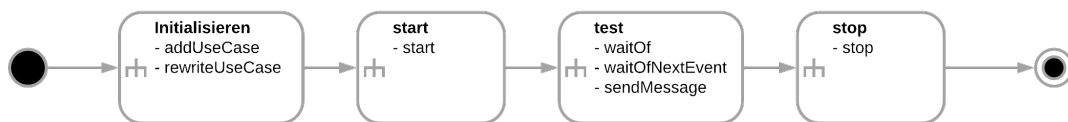


Abbildung 45: Ablaufdiagramm eines Tests

4.4 Vergleich der einzelnen Lösungen

In der Tabelle 1 werden vier Lösungen jeweils miteinander verglichen. In den Zellen werden gemeinsame Codezeilen von zwei Anwendungen abgebildet.

	Standalone mit DB	Standalone ohne DB	Framework	Library
Standalone mit DB	28466			
Standalone ohne DB	26210	26417		
Framework	25799	25972	26888	
Library	25799	25972	25799	26112

Tabelle 1: Vergleich der Anzahl der Codezeilen

Bibliography

- [Mar18] Robert C. Martin. *Clean Architecture*. 2018. ISBN: 978-0-13-449416-6.
- [And] Dossier Andreas. *Hexagonal architectur*. URL: https://www.dossier-andreas.net/software-architecture/ports_and_adapters.html (besucht am 9. Sep. 2022).
- [Aug] Stephan Augsten. URL: <https://www.dev-insider.de/was-bedeutet-inversion-of-control-a-1110688/#:~:text=Definition%20%E2%80%9ESteuerungsumkehr%E2%80%9C%20Was%20bedeutet%20Inversion%20of%20Control%3F&text=Inversion%20of%20Control%20ist%20ein, die%20Steuerung%20bestimmter%20Programmteile%20%C3%BCbernimmt>.
- [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).
- [OCA] OCA. URL: <https://www.openchargealliance.org/protocols/ocpp-201/>.