HTWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Fakultät Ingenieurwissenschaften

9010 - Bachelorarbeit

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

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

21. September 2022

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Motivation

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

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

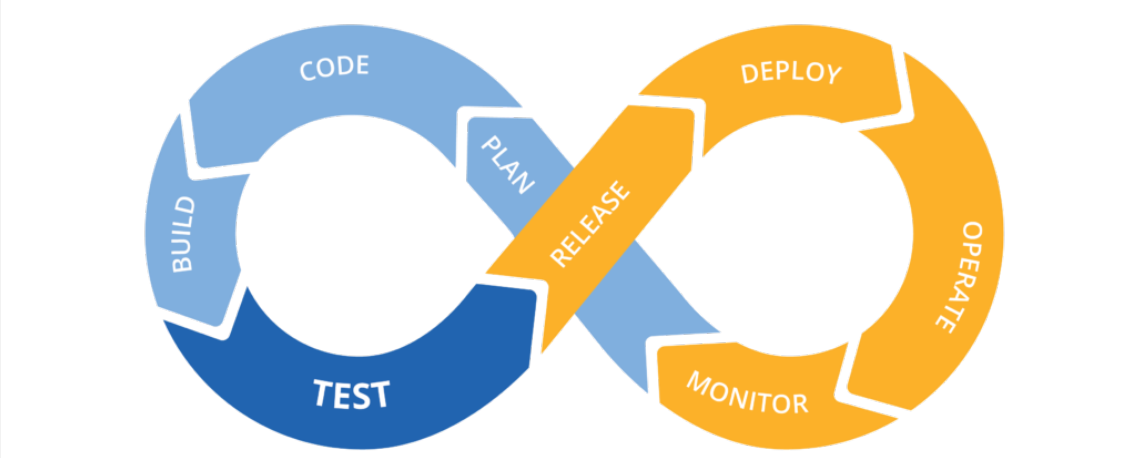


Abbildung 1: CI/CD Pipeline [1]

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 große 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 verbesseren 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

Im Kapitel werden die Grundlagen kurz dargestellt, die für den Entwurf einer Anwendung relevant sind. Die dargestellten Informationen werden auch in weiteren Kapiteln benutzt. Wenn es Bedarf besteht mehr sich darüber zu informieren, müssen externe Quellen benutzen werden (siehe Literaturverzeichnis).

2.1 OOP Design Patterns

Design Patterns (dt. Entwurfsmuster) sind typische Lösungen für oft auftretende Probleme in verschiedenen Software-Anwendungen [Shv]. Die Entwurfsmuster beziehen sich auf Lösungen in OOP Sprachen, da es am meistens benutzten Programmiersprachen sind. Jeder Entwurfsmuster beschreibt ein Problem und eine Lösung zu diesem Problem.

2.1.1 Adapter

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

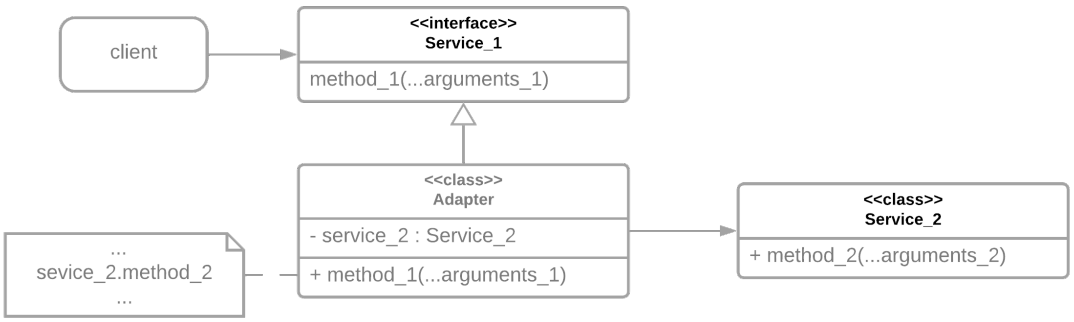


Abbildung 2: Klassendiagrammm Adapter

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.

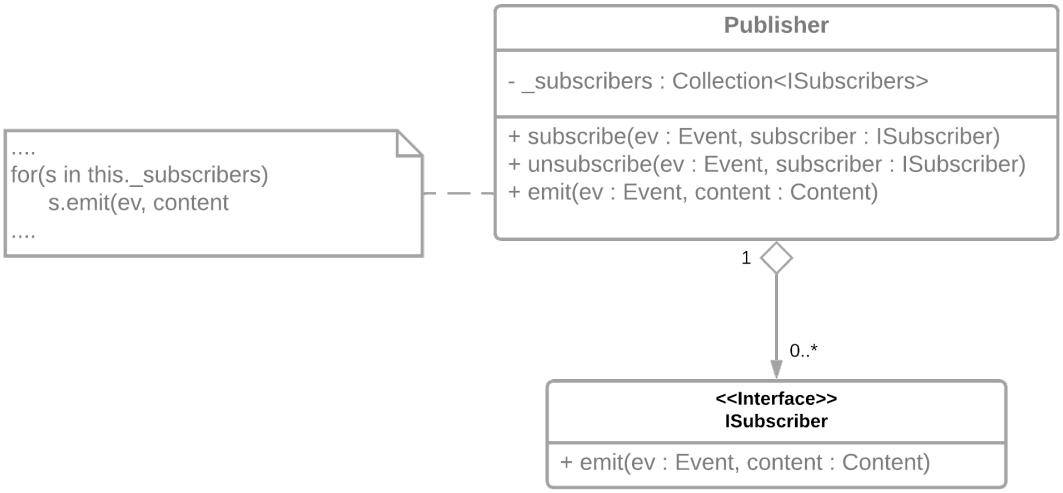


Abbildung 3: Klassendiagrammm Observer

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. Es wird eine Klasse definiert, die eine Main-Methode besitzt und die Main-Methode definiert einen Ablauf aus mehreren abstrakten Methoden, die in jeweiliger Implementierung der Klasse definiert werden sollen. Somit haben alle Implementierung der Klasse gleichen Ablauf, der in der Main-Methode definiert ist, aber einzelne Schritte des Ablaufs sind unterschiedlich.

2.1.5 Builder

Das OOP Design Pattern **Builder** (dt. Erbauer) ermöglicht das Erstellen von komplexen, zusammengesetzten Objekten in einzelne einfache Schritte zu zerlegen. Damit ist möglich eine Menge an Methodenaufrufen, die in eine bestimmten Reihenfolge aufgerufen werden sollen, in einer Methode zu isolieren.

2.1.6 Facade

Das OOP Design Pattern **Facade** (dt. Fassade) ermöglicht für eine komplexe Klasse oder eine Menge an Klassen, die aus vielen Methoden besteht, eine einfachere Klasse zu erstellen, die nur die notwendigen Methoden der komplexeren Klasse besitzt, ohne das Verhalten zu verändern. Die Fassade bietet ein einfaches Interface für ein komplexes System.

2.1.7 Singleton

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 Instanzierung nicht direkt mittels eines Konstruktors der Instanz sondern indirekt mittels einer Methode einer anderen Instanz (Fabrik), die die Aufgabe der Instanzierung übernimmt.

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

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

2.2 Architekturen

2.2.1 Model-View-Presenter Architektur

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

- **Model** - enthält die komplete 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**.

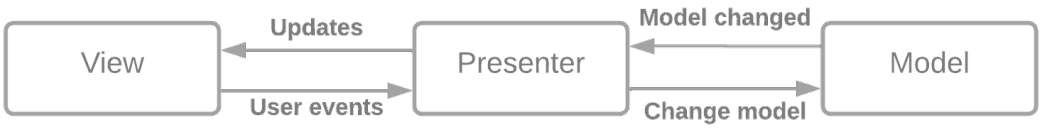


Abbildung 4: Datenfluss in MVP Architektur

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** interpetieren kann (z.B. Button “Speichern” soll grün werden).
- **View** verarbeitet das Ergebnis und zeigt es an (z.B. Button wird grün auf der Oberfläche angezeigt)

2.2.2 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.[Andb]

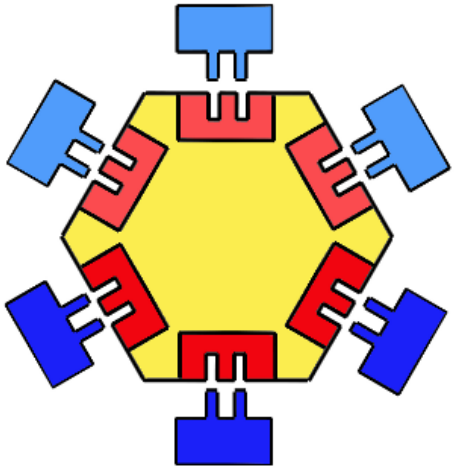


Abbildung 5: Port und Adapter Architektur [2]

[2] $https://www.dossier-andreas.net/software_architecture/ports_and_adapters.html$

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 zehn 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 für Defekte zu minimieren, können automasierte 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 automasierten Tests, die im Folgenden beschrieben werden.

2.3.1 Testing Pyramide

Die Abbildung 6 ist ein Vergleich zwischen den verschiedenen Testtypen zueinander. Es werden fünf Testtypen miteinander verglichen, die in den nächsten Kapiteln beschrieben werden. Die Kategorien, in denen die Testtypen miteinander verglichen werden, sind:

- Anzahl an Tests
- Komplexität
- Zerbrechlichkeit
- Wartungskosten
- Laufzeit
- Zeit um den Fehler zu lokalisieren

Die Farben in der Pyramide symbolisieren die Priorisierung jedes einzelnen Testtypes. Laut der Pyramide sollen zum Beispiel Unittests vor Integrationtests priorisiert werden. D.h. die Integrationtests testen keine Funktionalitäten, die mit Unittests abgedeckt werden können.

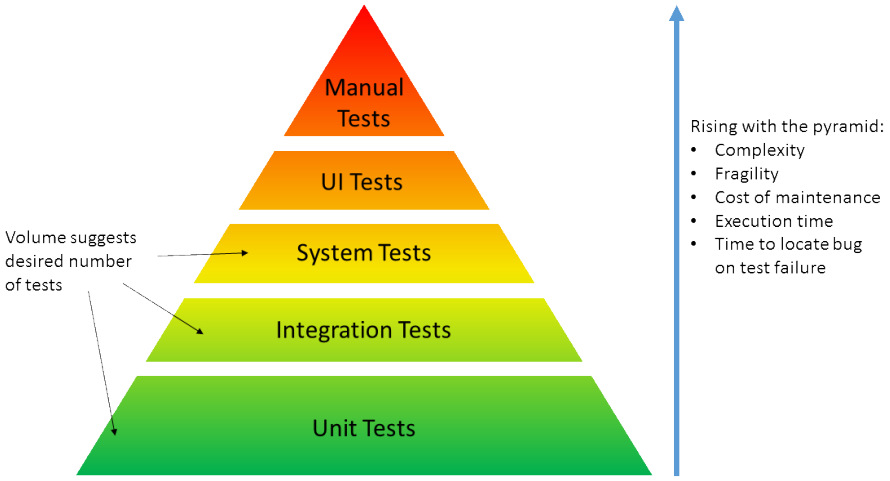


Abbildung 6: Caption written below figure [3]

[3] https://www.cqse.eu/de/news/blog/junit3-migration/

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 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 Laufzeitumgebeung, um den Benutzer zu simulieren, ein UI Test wird nur mittels der Ausgabe an der Oberfläsche 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 werdern, ist die Anzahl dieser Tests sehr gering.

2.3.6 Manual Tests

Die manuellen Tests testen die Anwendung genauso wie die System und UI Tests. Der Unterschied ist, dass alle Schritte des Tests (Umgebung erstellen, Test auswerten, Test beenden) von einem Menschen manuell gemacht werden. Dadurch, dass fast jede Tätigkeit bei der Durchführung manuell erledigt wird, ist die Laufzeit jedes Tests sehr groß. Es ist sinnvoll bei regelmäßiger Wiederholung eines manuellen Tests, ihn mittels eines System- oder UI-Tests zu automatisieren.

2.4 Softwareentwicklung Prinzipien

2.4.1 Interface segregation principle

Das Interface-Segregation-Prinzip (dt. Schnittstellenaufteilungsprinzip) ist Begriff aus den objektorientierten Entwurf. 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()

Das Nutzen von **ISP** ist, dass wenn mehrere Interfaces gleiche Methode besitzen sollen, ist es am besten die gleichen Methoden in ein anderes Interface zu isolieren. Das bringt besseres Verständnis über die Struktur der Anwendung und minimiert Anzahl an Teilen der Software, die im Falle einer Änderung, mitgeändert werden sollen.

2.4.2 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 Zeichnen, dass das Modul aufgeteilt werden soll.

2.4.3 Open–closed principle

Das Open-Closed-Prinzip (dt. Prinzip der Offen- und Verschlossenheit) 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, darf nach dem **OCP** die bestehende Software minimal geändert werden (idealerweise gar nicht). Das lässt sich in der Kombination mit **SRP** (siehe Kapitel 2.4.2) erreichen, indem alle Module bzw. Klassen nur eine Aufgabe erledigen und somit nur einen Grund für die Änderungen besitzen.

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. [dew]

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 7: Vergleich der schlechten(links) und guten(rechts) laut DIP Abhängigkeiten

2.4.5 Inversion of control

Inversion of control (IoC, dt. Steuerungsumkehr) 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.

2.4.6 Dependency Injection

Dependency Injection (DI, dt. Abhängigkeitsinjektion) ist ein Beispiel von **IoC**. In der Abbildung 8 ist ein Beispiel von einer Anwendung zu sehen, die die von der Konsole ankommenden Zahlen quadriert und das Ergebnis zurückgibt.

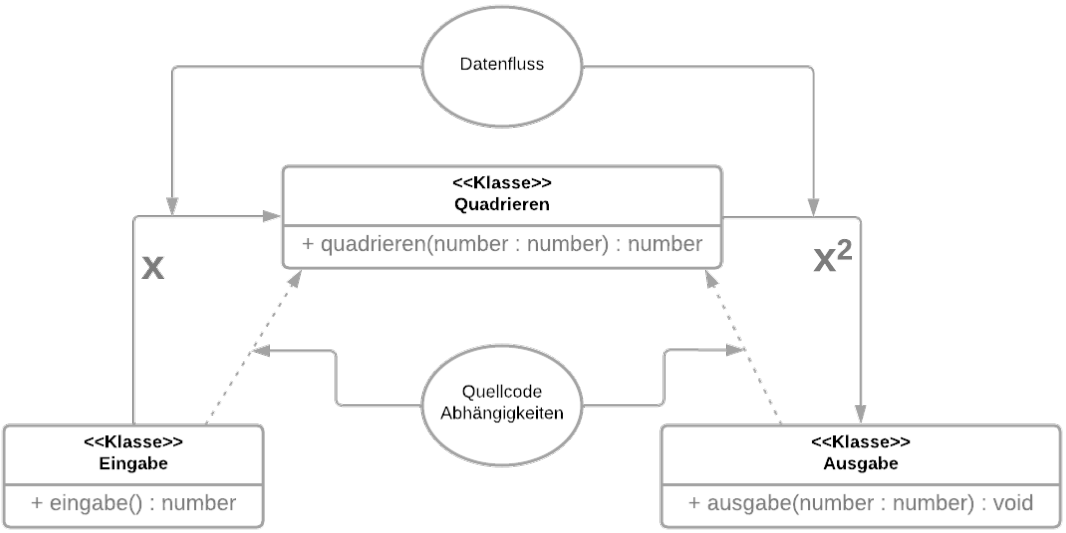


Abbildung 8: Datenfluss und Quellcode Abhängigkeiten

Die Funktion **Quadrieren** befindet sich in dem Fall auf einem höheren Niveau als **Eingabe** und **Ausgabe** (siehe 2.4.4), 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 Anderungen 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 mit Interface (für OOP Sprachen) umsetzen, in dem es frühestens bei der Initialisiereung der **Quadrieren** Klasse das jeweilige Eingabe und Ausgabe Objekt übergeben wird. Die Lösung sieht so aus:

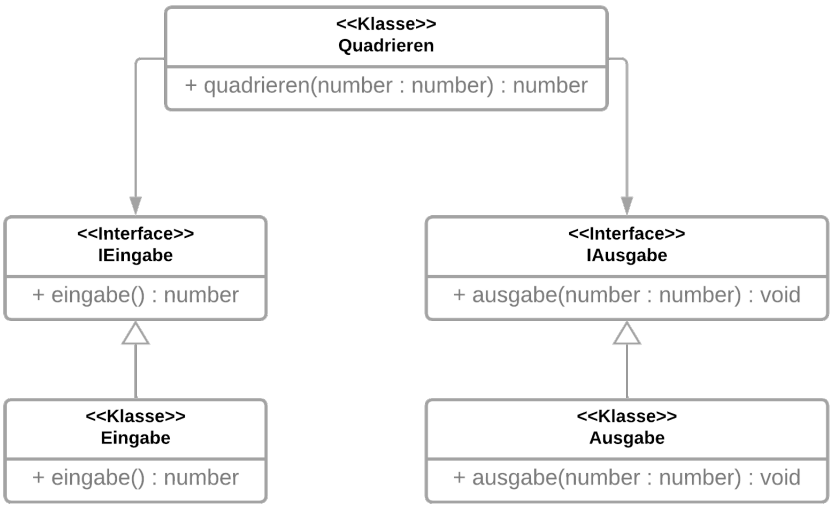


Abbildung 9: Entkopplung der Abhängigkeiten

Durch die Nutzung von Interfaces lässt sich die Funktion **Quadrieren** mit Unittests abdecken, indem die Klassen **Eingabe** und **Ausgabe** simuliert werden. Die Instanzen von simulierten Klassen werden bei der Initialisierung der Klasse **Quadrieren** übergeben. Wenn alle Klassen über Interface miteinander verbunden sind, ist es möglich, dass die Umgebung von jeder einzelnen Klasse bei den Tests simuliert 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 enwickelt 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 werden. Somit kann ein Server Ladesäulen von unterschiedlichen Herstellern gleichzeitig betreiben.

OCPP definiert eine Menge an Nachrichten, deren Inhalte und Struktrur. Es werden auch die Situationen beschrieben, in deren die Nachrichten abgeschickt werden dürfen. Das Übertragungsprotokoll, in dem der Nachrichtenaustausch stattfindet, ist vorgegeben.

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 Authoren 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 Anwedung 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 Erweitbarkeit 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 Technical Debts

Der Begriff Technical Debts (dt. technische Schulden) bezeichnet Overhead, welcher durch die "Unsauberkeit" des bestehenden Programms verursacht werden. Die zusätzlichen Leistungen müssen bei jeder Änderung und Erweiterung erbracht werden.

Die technischen Schulden entstehen dadurch, dass bei der Entwicklung eines Teiles des Systems von den Entwicklungsteam weniger Zeit für nicht gewinnbringende Aufgaben inverstiert 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 Begin der Umsetzung des Softwaresystems, die oben gennanten Aufgaben zu lösen, denn mit zunehmender Lebenszeit der Software nimmt die Änderungszeit zu. Somit lassen sich die vorhandenen Ressourcen effizienter eingesetzen.

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 deren Qualität unterschiedliche Menge an Zeit inverstiert wurde, miteinander verglichen.

	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

Tabelle 1: Vergleich zwei Anwendungen mit viel und wenig inverstierter Zeit in die Architektur

Um diese Vorteile (siehe Tabelle 1) zu erhalten, muss regelmäßig Zeit in folgende Tätigkeiten inverstiert 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 meißt entweder die Zeit oder das Budget oder beides zu gering ist. Mit Hilfe folgender Kriterien können einfacher Entscheidung getroffen werden:

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

In der Abbildung 10 ist zu erkennen, dass auf längerer Distanz eine gute Softwarearchitektur deutlich mehr Funktionalitäten besitzt als eine Software mit schlechter Architectur. Im vorderen Zeitinterval führt allerdings die Software mit der qualitativ schlechteren Architektur.

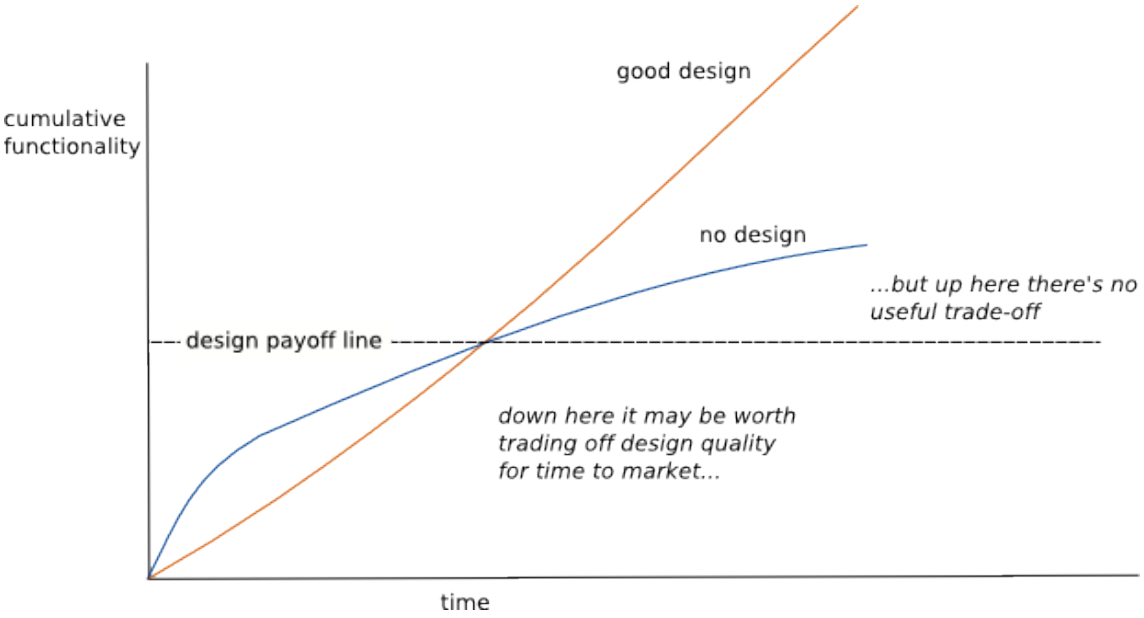


Abbildung 10: Vergleich einer guten und einer schlechten Softwarearchitektur [4]

Dieser Verlauf ist bei Projektbegin 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.

[4] 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. Die dargestellte Architektur ist Eigenwert.

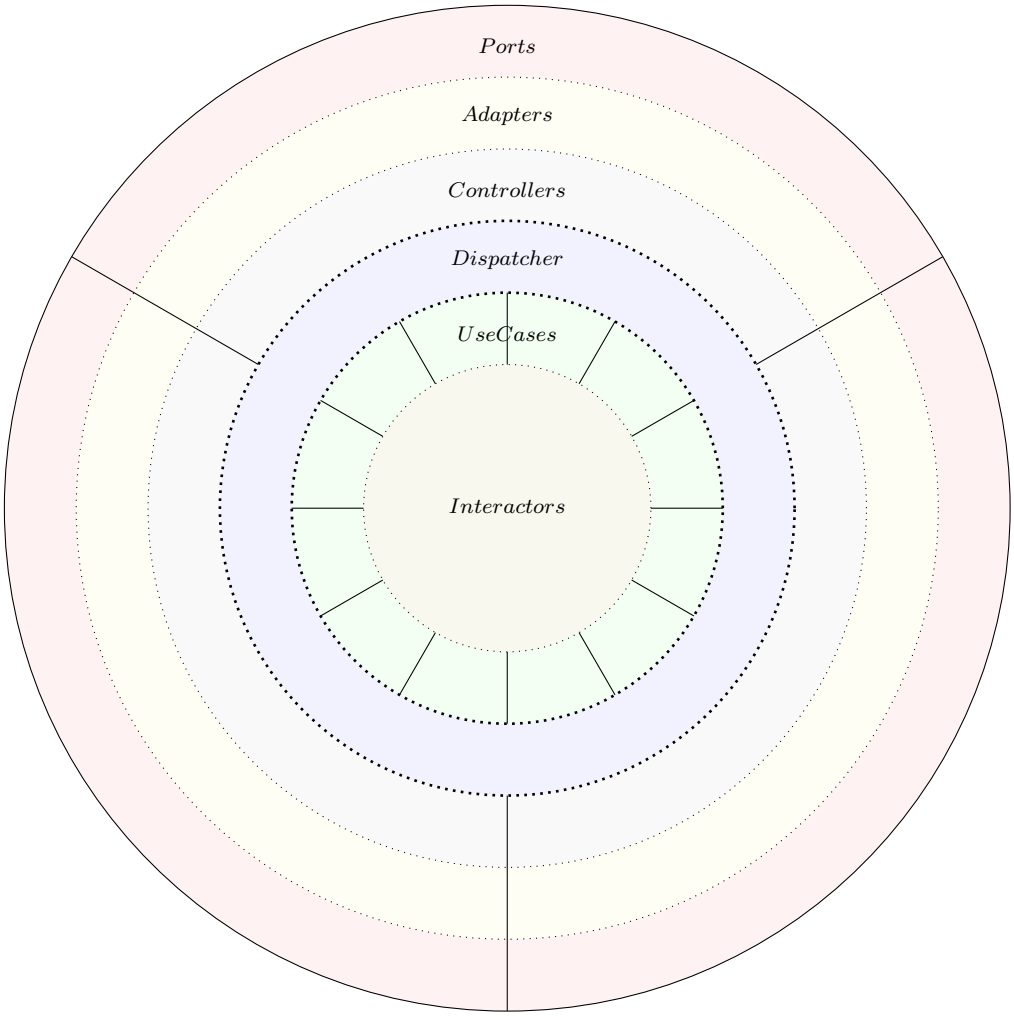


Abbildung 11: Darstellung der umgesetzten Architektur mit 6 Schichten

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 ankommende Ereignisse vom **Port** an den dazugehörigen **Controller** und in die andere Richtung zu übersetzen (z.B. Presenter im **MVC ??**)
- 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). Über alle Ereignisse wird **Dispatcher** informiert.

- Dispatcher - hat die Aufgabe die aufgetretenen Ereignisse an die **UseCases** zuzuordnen.
- 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 reelen Schnittstellen schneller als mit reelen 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 nähesten zu der **PAC**[5] Struktrur ist das Pattern **MVP** (siehe Kapitel 2.2.1). Dabei entsprechen die Aufgaben von **View** (Benutzerobefläche aktualisieren und Ereignisse empfangen) den Aufgaben von **Port**. Die Aufgaben von **Presenter** (Ereignisse von der Benutzerobefläche in das Datenmodel der Anwendung umwandlen) entsprechen den Aufgaben **Adapter**. Und die Aufgaben von dem Rest des Programms inklusiv **Controller** entspechen 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 Benutzerobeflächen gedacht ist, während es in der hier beschriebenen Umsetzung für alle Schnittstellen benutzt wird. **MVP** Architektur übernimmt in der gesamten Application eine zentralle Stelle und ist nur einmal vorhanden. **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 Steurung 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 2.4.2). Somit hat jeder Teil eine Aufgabe und lässt sich besser im Laufe der Zeit warten.

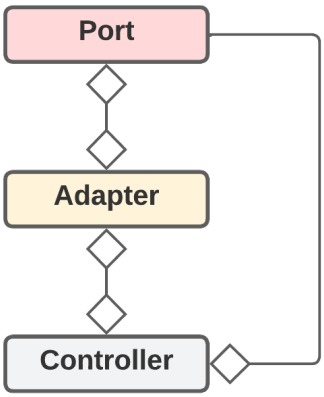


Abbildung 12: Objektendiagramm PAC

[5] Port-Adapter-Controller

3.2.1.2 Controller-Dispatcher-UseCase-Interactor

In dem Teil wird die Aufbau der inneren Logik der Application beschrieben. Für die Anwendungslogik ist ein Teil nach **SRP** (siehe Kapitel 2.4.2) notwendig. Das Teil heißt **UseCase**. Damit beim Geschehen eines Ereignisses alle **UseCases** informiert werden, wird ein weiteres Teil notwendig, das das Ereignis an die richtigen **UseCases** zuordnet. Das Teil heißt **Dispatcher**. Das Weiterleiten des Ereignisses an **Dispatcher** und das Kontrollieren von **Port** und **Adapter** wird vom **Controller** übernommen.

Jedes **UseCase** erledigt mehrere aufeinander folgende Aufgaben, die andere **Controller** benutzen. Es ist vom Vorteil, andere **Controller** nicht direkt aufzurufen. Es kann passieren, dass es gewünscht wird gleiche Funktionalität für alle solche Aufrufe hinzuzufügen, 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 Objektendiamm:

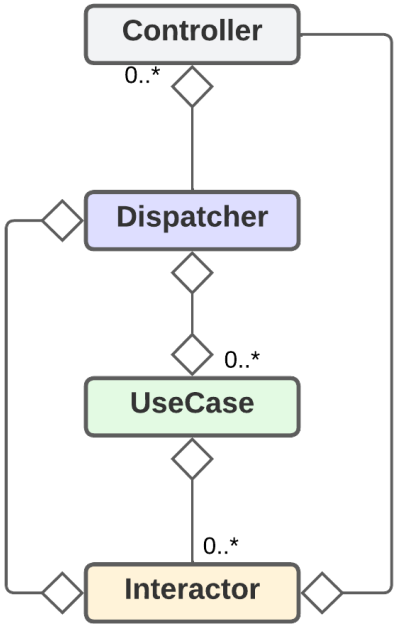


Abbildung 13: Objektendiagramm Controller-Dispatcher-UseCase-Interactor

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 benutzt Pattern **Builder** (siehe Kapitel 2.1.5).

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 sinnvol, dass beide Schritte explicit ausgeführt werden.

Ein möglicher Ablauf ist in Abbildung 14 dargestellt:

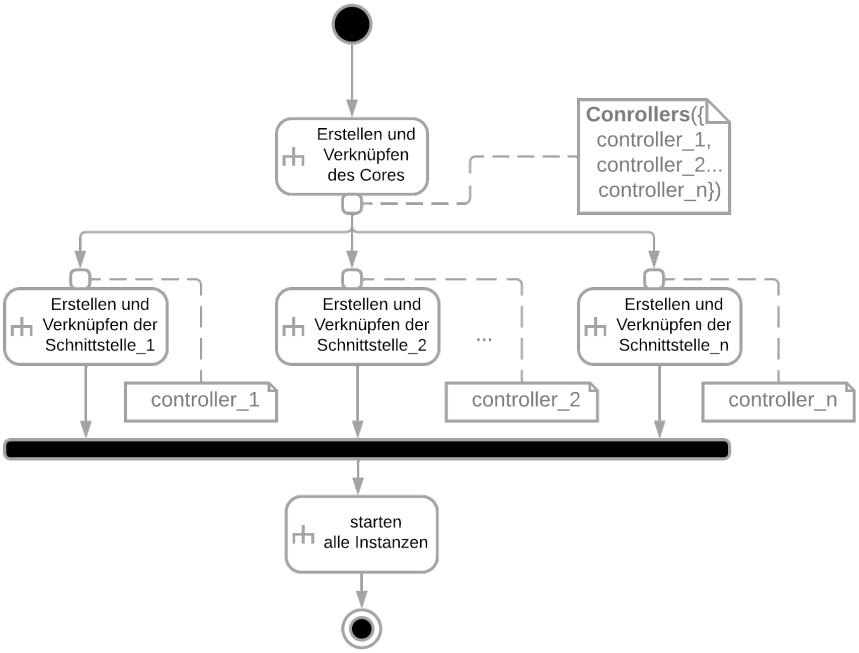


Abbildung 14: Ablaufiagramm Erstellen der Struktur

Mit diesem Ablauf können mehrere Hauptprogramme erstellt werden, die verschiedene Anwendungen für verschiedene Zwecke erstellen. Z.B. ein Framework braucht keine reelen 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 instanziert 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)
- Initialisiereung 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 dem Datenbank speichern). Jeder Test wird dadurch deutlich länger laufen. 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 laufendenen Tests geändert wird.

Bei der zweiten Möglichkeit müssen alle Utility Controllers entweder bei der Initialisiereung 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 verschieden Instanzen anbietet. Alle Module bzw. Klassen lassen sich somit auch mit Unittests abdecken, da die Utility Controllers entsprechend gemockt werden können.

In der Abbildung 15 ist die Verbindung jedes **Controller**s mit den Utility Controllers als Klassendiagramm dargestellt.

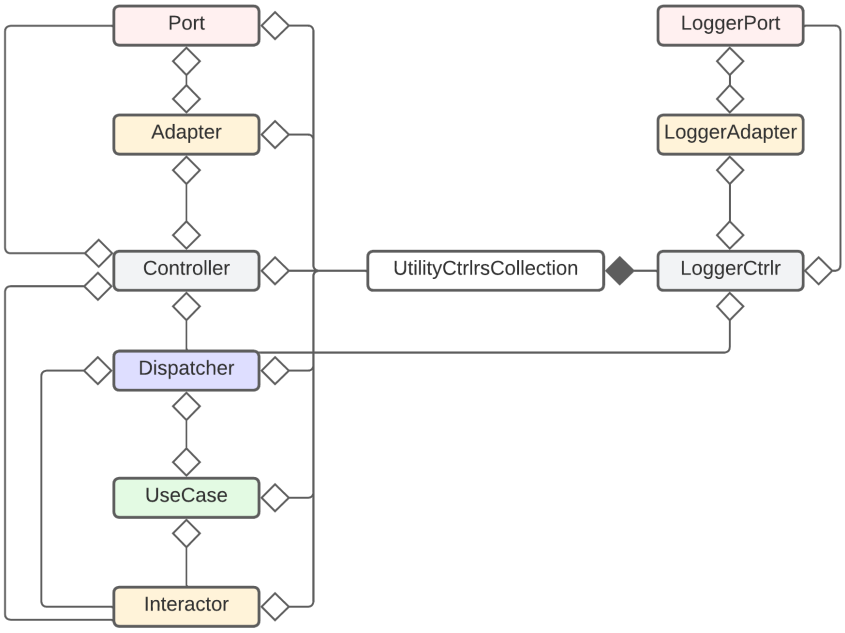


Abbildung 15: Objektendiagramm mit Utility Controllers

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** (siehe Kapitel 2.4.6) miteinander verknüpft. Beispiel für **Port-Adapter-Controller** Verbindung:

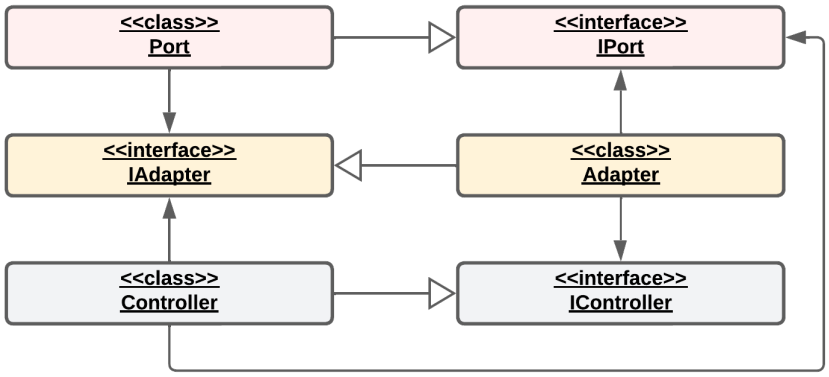


Abbildung 16: Klassendiagramm Port-Adapter-Controller

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

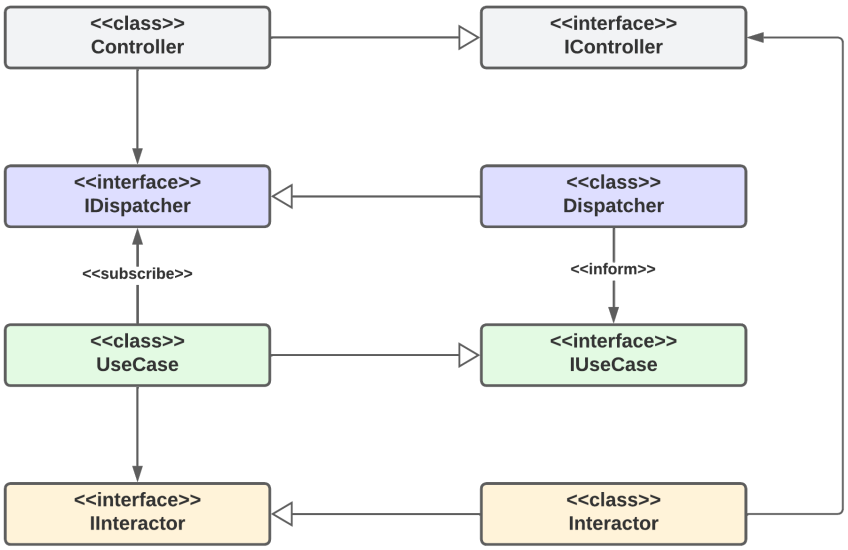


Abbildung 17: Klassendiagramm Controller-Dispatcher-UseCase-Interactor

Im Klassendiagramm 16 und 17 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). Dieser Datenfluss ist der umgekehrte Datenfluss "1".

Die einfache Kombination aus drei Datenflüssen sieht so aus:

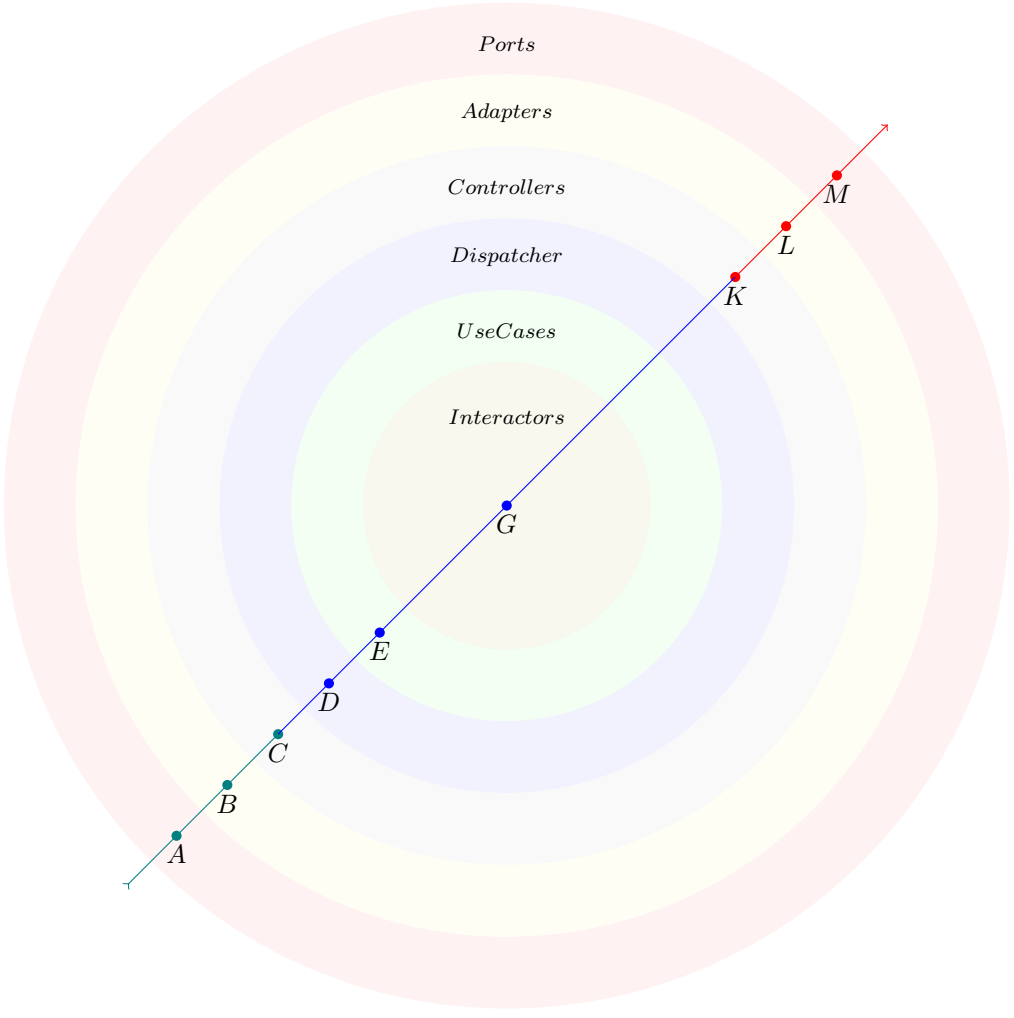


Abbildung 18: Darstellung des Datenflusses in der Architektur

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

3.2.2.1 Datenfluss 1 und 3

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

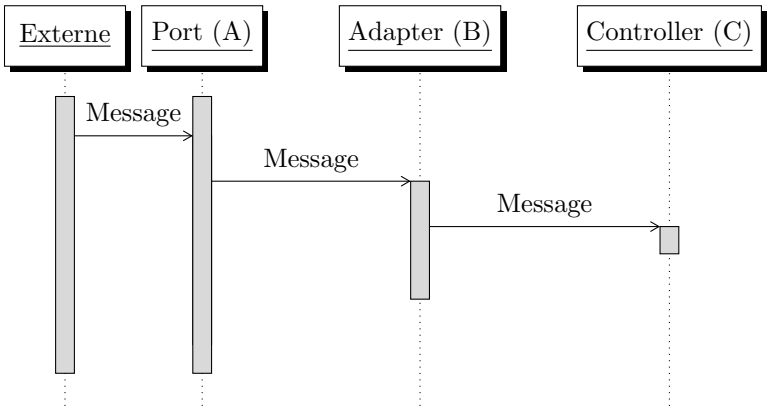


Abbildung 19: Sequencediagramm vom Datenfluss **1** Grün

Darstellung des Datenflusses **3** als **sequencediagram**:

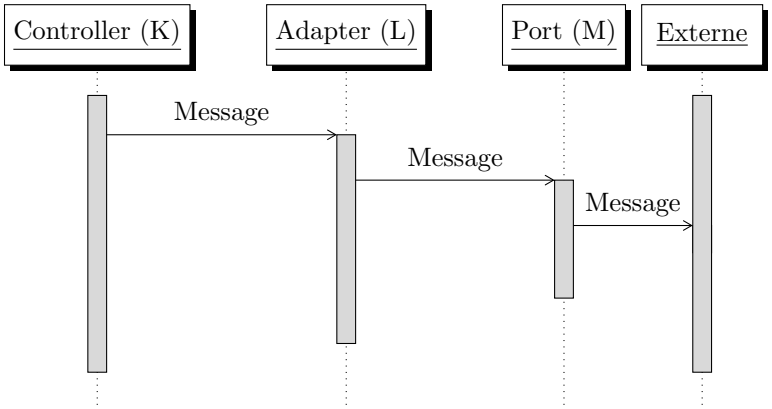


Abbildung 20: Sequencediagramm vom Datenfluss **3** Rot

Darstellung der Datentransformation:

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

Ein Beispiel der übergebenen Information an den **Controller** von **Adapter** und an den **Adapter** vom **Controller** für OCPP Kommunikation kann wie folgt aussehen:

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

Adapter - Port: Alle Informationen, die gesendet werden (in dem Fall "message"), werden in der verstandlichen Form (sie muss nicht mehr in der Anwendung geändert werden) für den Port an Port weitergegeben. Falls es notwendig ist, müssen alle Informationen über den Empfänger oder Sender 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 an den **Adapter** vom **Port** und an den **Port** vom **Adapter** im Falle einer OCPP Nachricht:

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

3.2.2.2 Datenfluss 2

Darstellung des Datenflusses **2** als **Sequencediagram**:

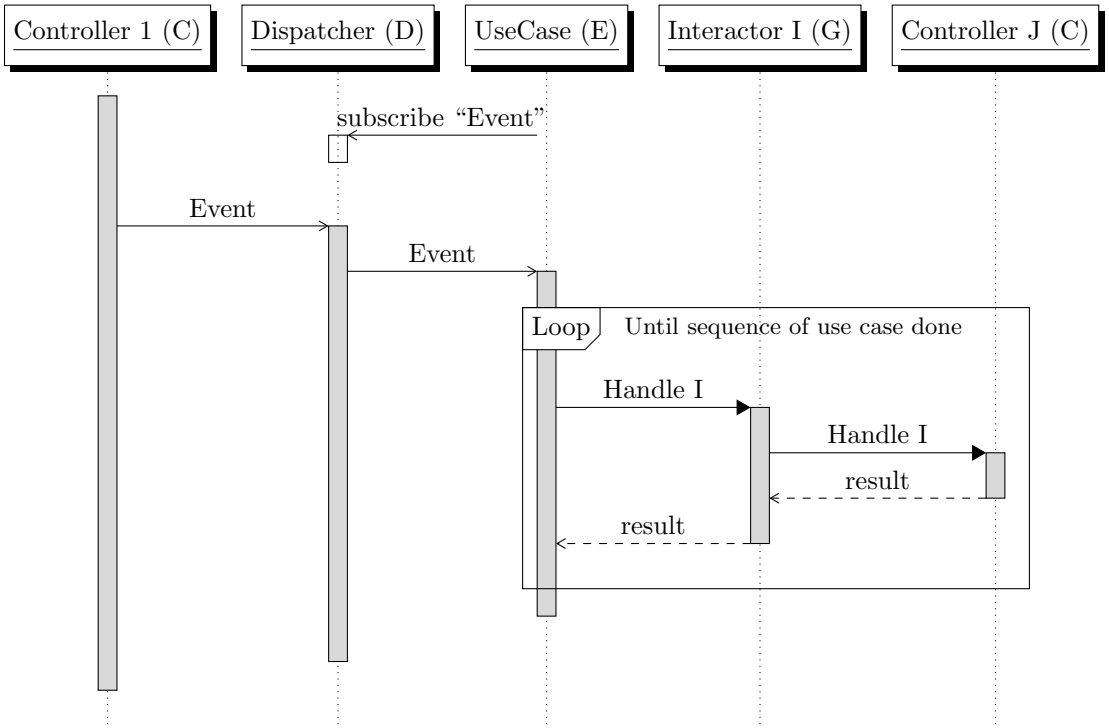


Abbildung 21: Sequencediagramm vom einfachen Datenfluss **2** Blau

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 abonierte **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 21)
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:

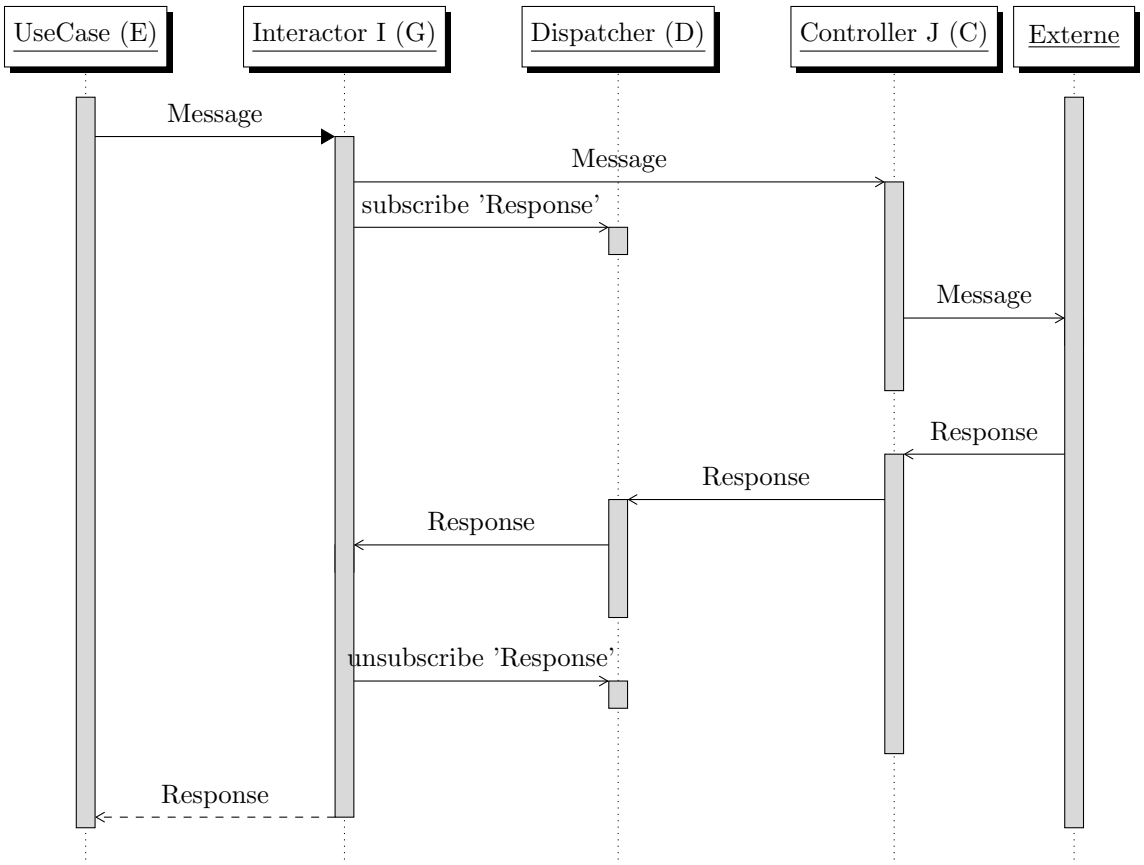


Abbildung 22: Sequencediagramm vom komplexen Datenfluss **2** Blau

Beim synchronen Funktionsaufruf wird der Rückgabewert, wie in Abbildung 21 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, unteranderem auch den **Interactor**. **Der Interactor** gibt diese Antwort als Rückgabewert des Funktionsaufrufs zurück.

3.2.2.3 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 23 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
- Aufzeichnung in jeder Komponente den Inhalt des ausgehenden Ereignisses
- Aufzeichnung allee Komponenten, 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 Integrationtests, da der Ablauf des Ereignisses auch bekannt ist.

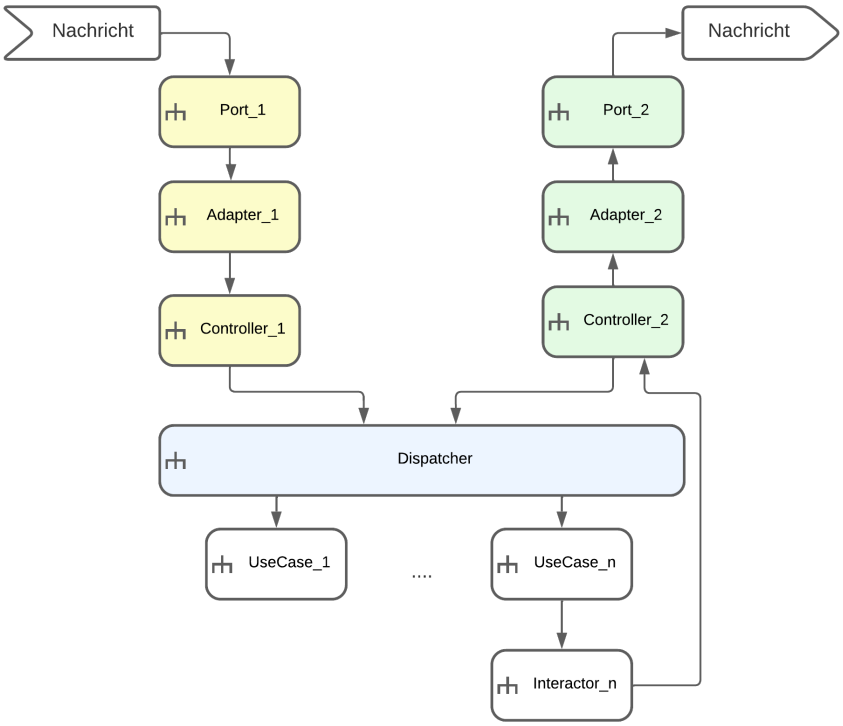


Abbildung 23: Kompletter Datenfluss

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 entspechend 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 entspechend mit Unittests abdecken lässt.

Die zweite Möglichkeit streut das Verhalten für ein Ereignis im Projekt. Das verbessert die Lesbarkeit des jeweiligen Teils und erfühlt **OCP** (Open-Closed-Prinzip, siehe Kapitel 2.4.3), jedoch die Schwierigkeit bringt alle solche Teile im Projekt zu finden. Das Gesamtverhalten lässt sich erst mittels einem Integrationtest 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 Anwednung.

Grundsätzlich lässt sich die Abbildung 11 folgendeweise vereinfachen:

Abbildung 24: Vereinfachte Darstellung der Architektur

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

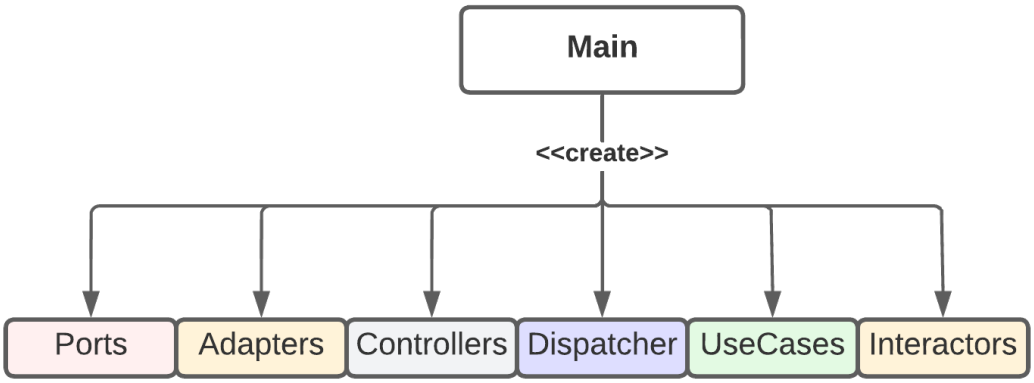


Abbildung 25: Vereinfachte Darstellung der Struktur einer Standalone Anwendung

Der Datenfluss lässt sich so darstellen:

Abbildung 26: Vereinfachte Darstellung des Datenflusses in 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:

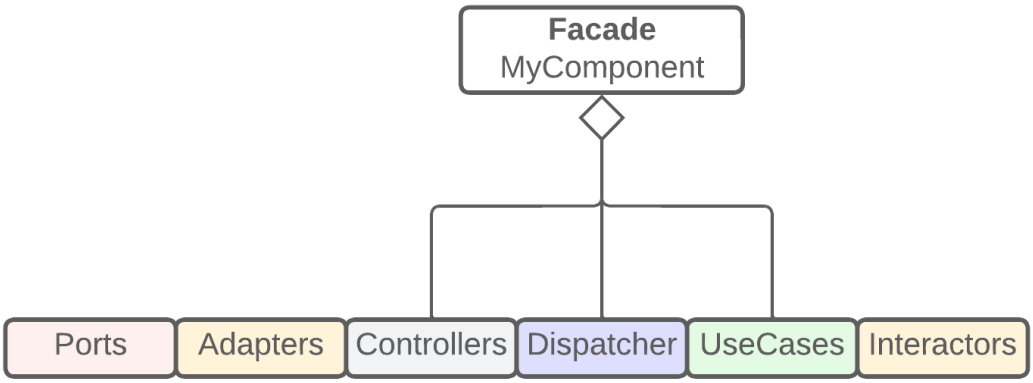


Abbildung 27: Vereinfachte Darstellung der Architektur als Komponente

Laut der Darstellung 27 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 Anwednung 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:

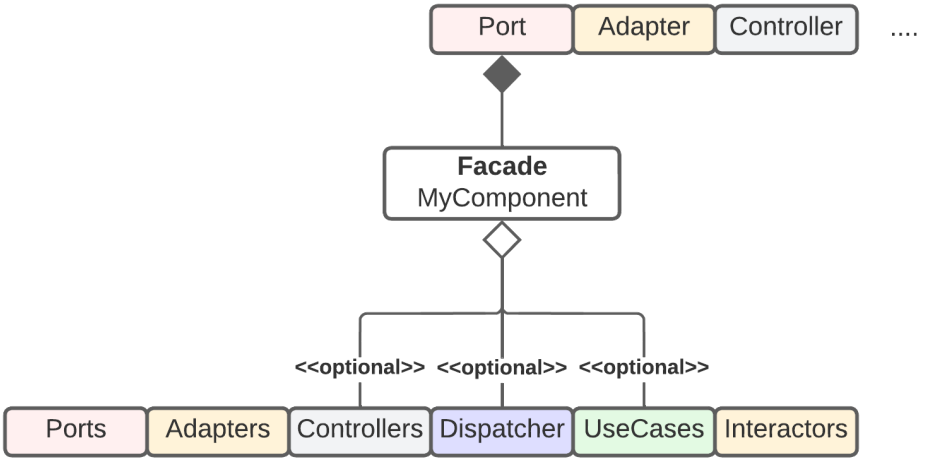


Abbildung 28: Vereinfachte Darstellung einer Standalone Anwendung mit der Komponente

Der Datenfluss in der Anwendung:

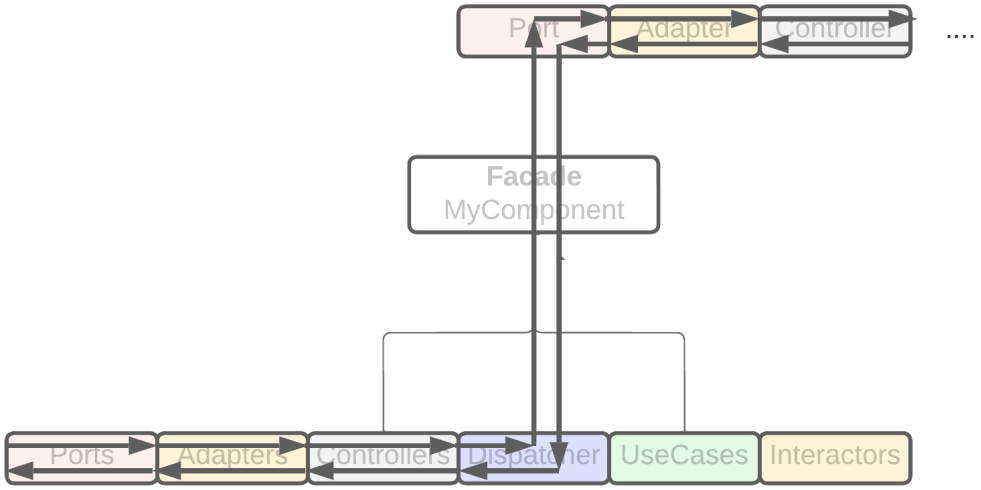


Abbildung 29: Vereinfachte Darstellung des Datenflusses in einer Anwendung mit Komponente

3.3.2 Framework und Bibliothek

Im Abschnitt 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. [6] Definition der Bibliothek ist:

> Bibliothek ist eine Sammlung ähnlicher Objekte, die zur gelegentlichen Verwendung gespeichert werden.[7]

Der Unterschied zwischen Framework und Bibliothek besteht darin, dass bei Framework der geschriebene Code von Framework aufgerufen wird und bei Bibliothek der geschriebene Code den Code von Bibliothek (Inversion Of Control, Kapitel 2.4.5). [8] Beispiele für Bibliotheken sind alle Implementierungen von Netzwerkprotokollen. Das komplette Verhalten muss vom Clientcode definiert werden.

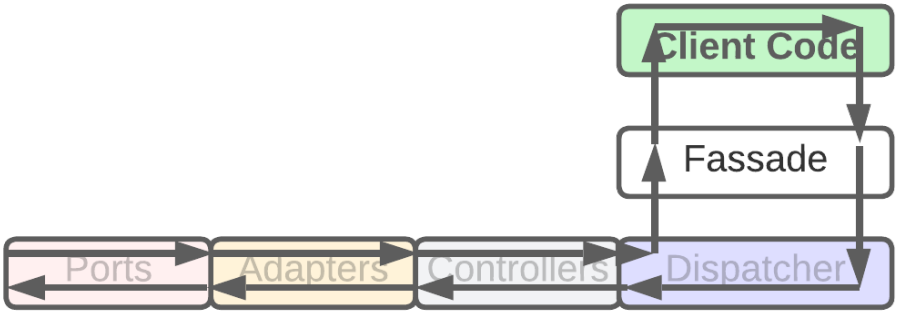


Abbildung 30: Vereinfachte Darstellung des Datenflusses bei einer Bibliothek

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

Abbildung 31: Vereinfachte Darstellung des Datenflusses bei einem Framework

Vergleich des Client Codes von Bibliothek und Framework:

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

[6] https://it-talents.de/it-wissen/framework/
[7] https://www.computerweekly.com/de/definition/Bibliothek-Library#: :text=In%20der%20Informatik%20bezeichnet%20der,und%20p
[8] https://martinfowler.com/bliki/InversionOfControl.html

4 OCPP Server

4.1 Aufgabenbeschreibung

Als Beispiel der Anwendung der beschriebenen Architektur dient OCPP 1.6. Server. Es sollen insgesamt mehrere unterschiedliche Anwednungen entstehen, die für unterschiedliche Zwecke gedacht sind.

Das sind:

- Testframework für die automatisierten Systemtests.
- Eigenständiger OCPP1.6 und später OCPP2.0 Server (mit und ohne Datenbank).
- Bibliothek, die in einer anderen Anwendung benutzt wird.

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: die angefragten Verbindungen von bekannten Ladesäulen akzeptieren bzw. von unbekannten Ladesäulen ablehnen.
- Benutzerverwaltung: die Authentifizierung von bekannten Benutzern an Ladesäulen zulassen bzw. von unbekannten Benutzern ablehnen.
- Preisverwaltung: unterschiedliche Preise für unterschiedliche Benutzer emöglichen.
- Ladevorgangverwaltung: Ladevorgänge serverseitig unterstützen.
- Benutzeroberflächeverwaltung: ermöglichen alle Bereiche des Servers manuell ändern zu können.
- Ladesäulekommunikationverwaltung: die Kommunikation mit den Ladesäulen mittels OCPP Protokoll ermöglichen.
- Loggingverwaltung: 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 32 ist ein Übersichtdiagramm der Standalone-Anwendung mit Datenbank.

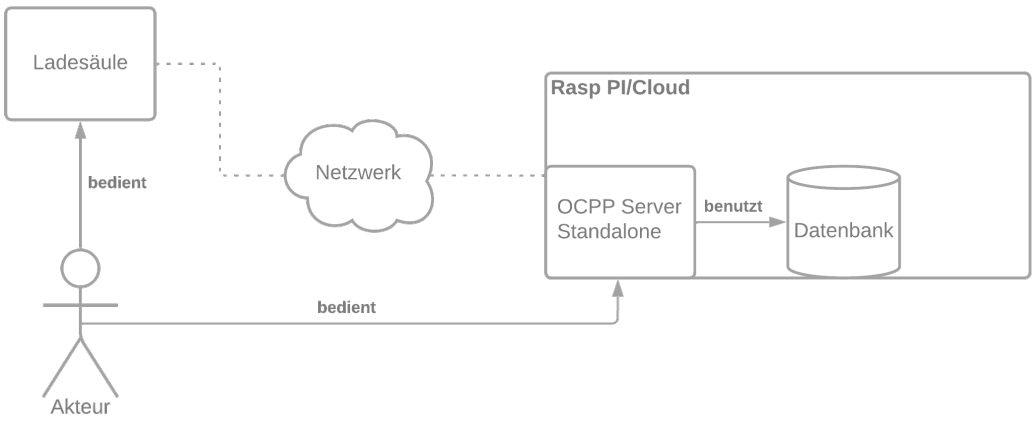


Abbildung 32: Übersichtdiagramm der Standalone-Anwendung mit Datenbank

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 des Einsatzes 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 33 ist ein Übersichtdiagramm der Standalone-Anwendung ohne Datenbank.

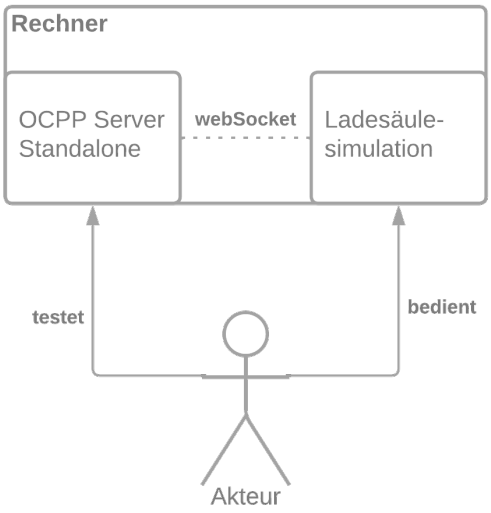


Abbildung 33: Übersichtdiagramm der Standalone-Anwendung ohne Datenbank

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 34 ist ein Übersichtdiagramm der Framework-Anwendung.

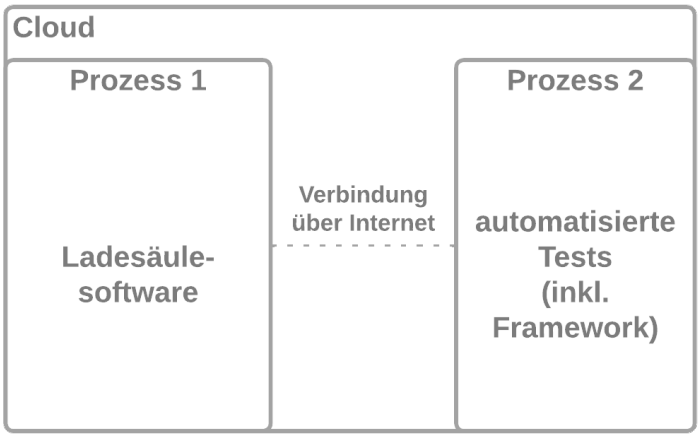


Abbildung 34: Übersichtdiagramm der Framework-Anwendung

4.1.4 Bibliothek

Der OCPP Server als Bibliothek soll in eine andere Software integriert werden, um mithilfe der ausgetauschten Nachrichten das Verhalten der Ladesäule (im allgemeinen Sinne) auswerten zu können. Es werden nicht alle Funktionalitäten des Servers benötigt. Die Software beinhaltet neben der OCPP Schnittstelle (mittels Bibliothek) auch weitere Schnittstellen.

Die Anforderungen an die Bibliothek sind:

- OCPP Server soll parametrierbar sein
- OCPP Nachrichten sollen im Cache gespeichert werden können
- Leichte Integrierbarkeit in die andere Software

Die nachfolgende Abbildung 35 ist ein Übersichtdiagramm der Software.

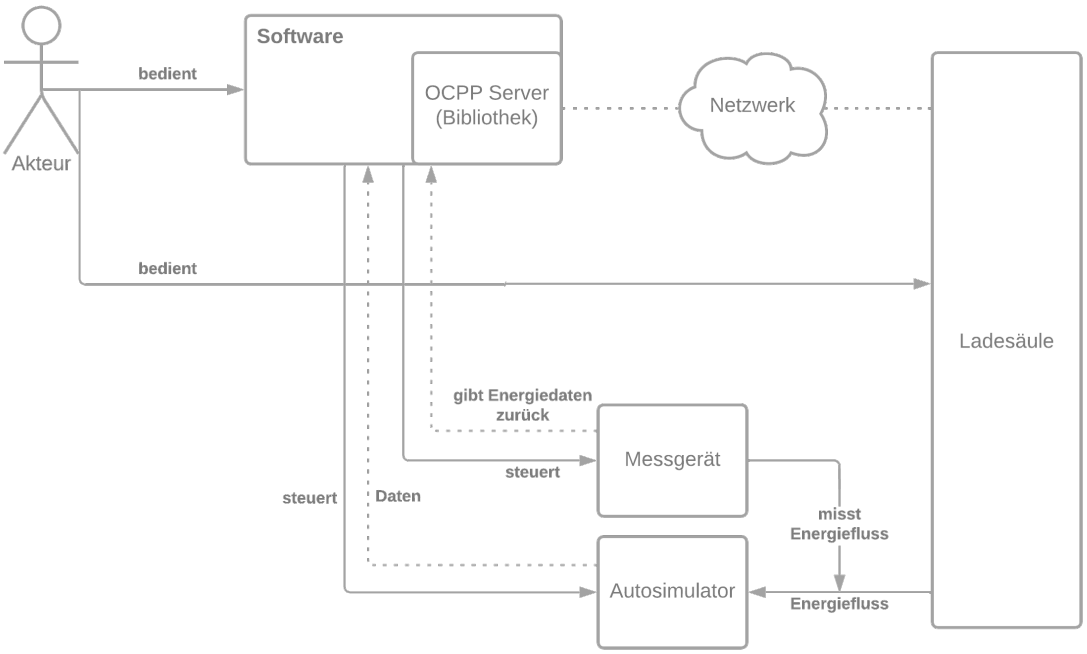


Abbildung 35: Übersichtdiagramm der Software

4.2 Allgemeine Lösung

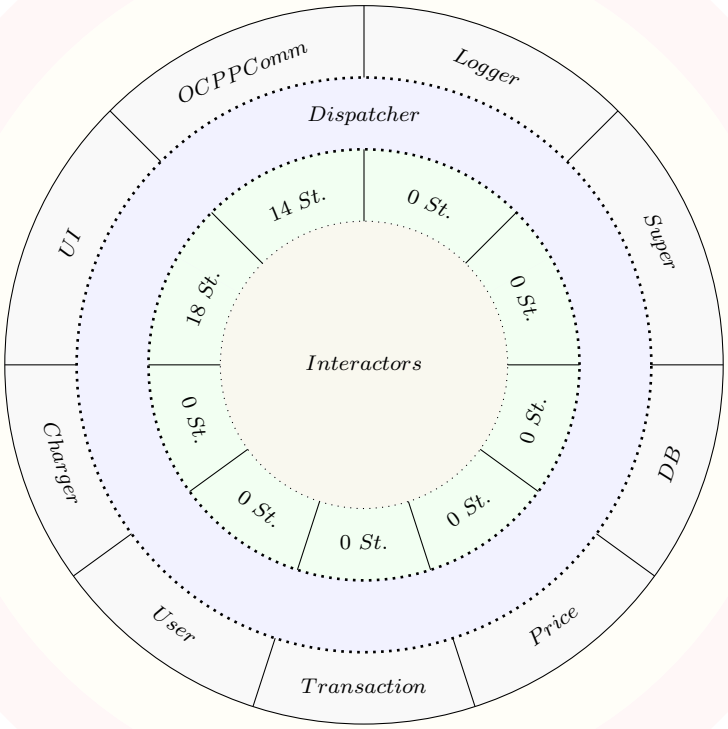


Abbildung 36: Darstellung des inneren Kreises der Architektur in jeder Anwendung

In der Abbildung 36 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 Anwednungen 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 Informtationen der bekannten Ladesäulen, ermöglicht folgende Sachen: neue Ladesäule anzulegen, Ladesäuleinformation ändern bzw. löschen, Überprüfen ob gegegbene Ladesäule bekannt ist.
7. **User-Controller** kontrolliert alle Informtationen 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 Ladevorgänge 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, Standardpreis zu ändern.

4.2.2 Dispatcher

Der **Dispatcher** verteilt die in **Controllers** aufgetretenen 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 (vom **Port** zum **Controller** oder vom **Controller** zum **Port**)
- Name
- Inhalt

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 reagieren 14 auf die Ereignisse vom “OCPPComm” Controller und 18 auf die Ereignisse vom “UI” 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:

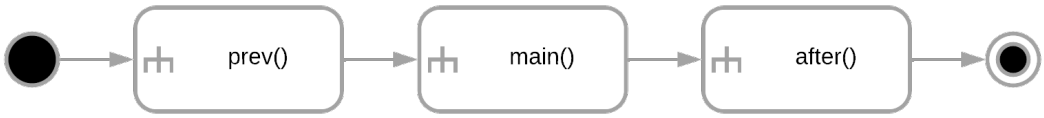


Abbildung 37: Ablauf eines Interactors

Aufbau des **Interactors**:

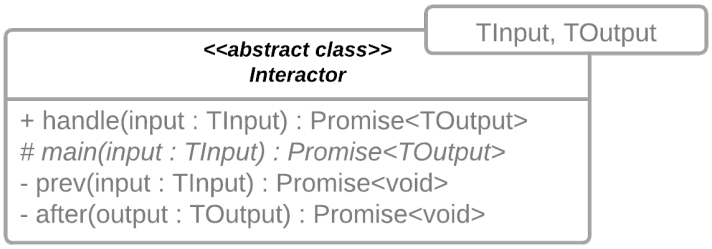


Abbildung 38: Klassendiagramm eines Interactors

Fast alle **Interactoren** hüllen nur Einzelmethoden von **Controllern** ab. Es gibt nur ein **Interactor**, der komplexes Verhalten besitzen. Das ist:

- Abschicken einer OCPP Request Nachricht. Der Rückgabewert ist die Antwort auf die geschickte Nachricht (OCPP Response). Siehe Abbildung 22.

4.3 Anwendung der allgemeinen Lösung

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

Der Unterschiede zwischen den einzelnen Lösungen sind:

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

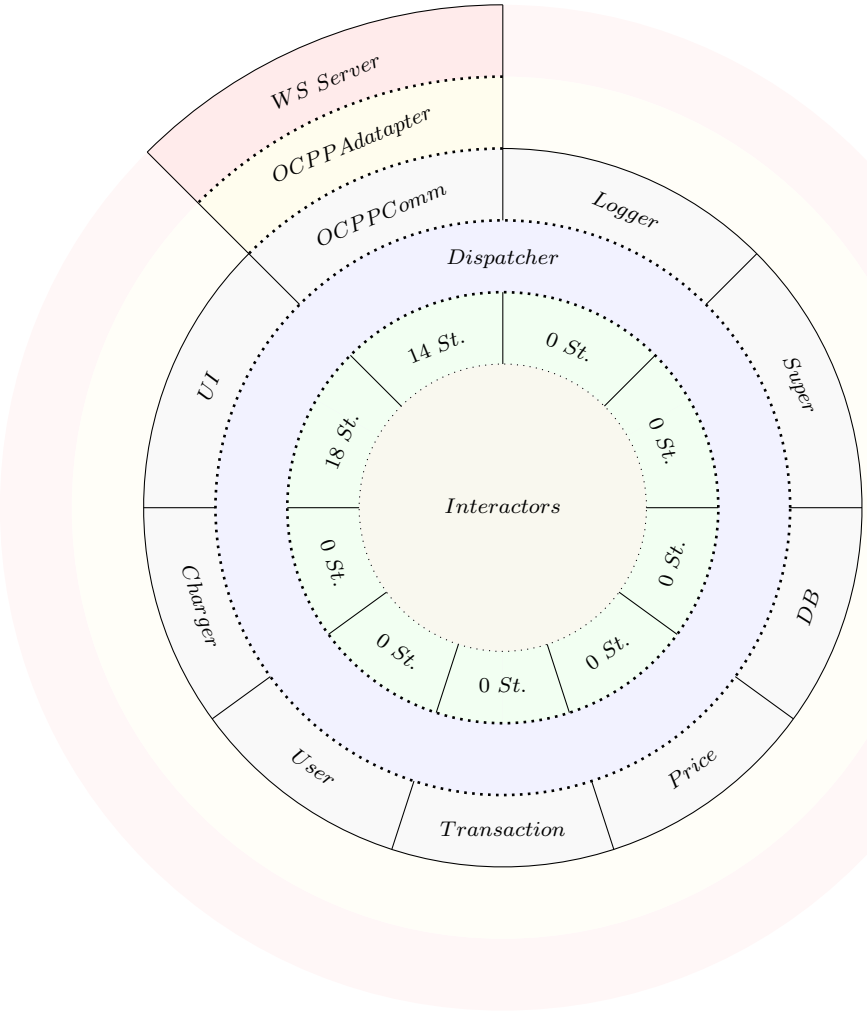
Jeder **Controller** außer "Super"-Controller kann eine reele Schnittstelle benutzen, falls sie in der Anwendung gebraucht wird. In jeder Anwendung kann eine Schnittstelle von mehreren **Controllers** verwendet werden. Das lässt sich zum Beispiel mittel **ISP** (siehe Kapitel 2.4.1) umsetzen.

Jede Schnittstelle kann drei Versionen haben:

- reele Schnittstelle
- Schnittstelle ist simuliert. 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 erfühlt 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** erfühlt (siehe Kapitel 2.4.3). 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 diesen 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 reel und gleich sein.



——— *Teile des Programms wissen nichts voneinander*

Abbildung 39: Darstellung des allgemeinen Teils jeder Anwendung

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 verwendeten Schnittstellen reel sind. Das sind:

- Datenbank (um verschiedene Daten zu speichern)
- HTTP Server (für Benutzerinteraktionen)
- Filesystem (für Logging)
- Websocket Server (für OCPP Schnittstelle)

Die Anwendung lässt sich wie folgt darstellen:

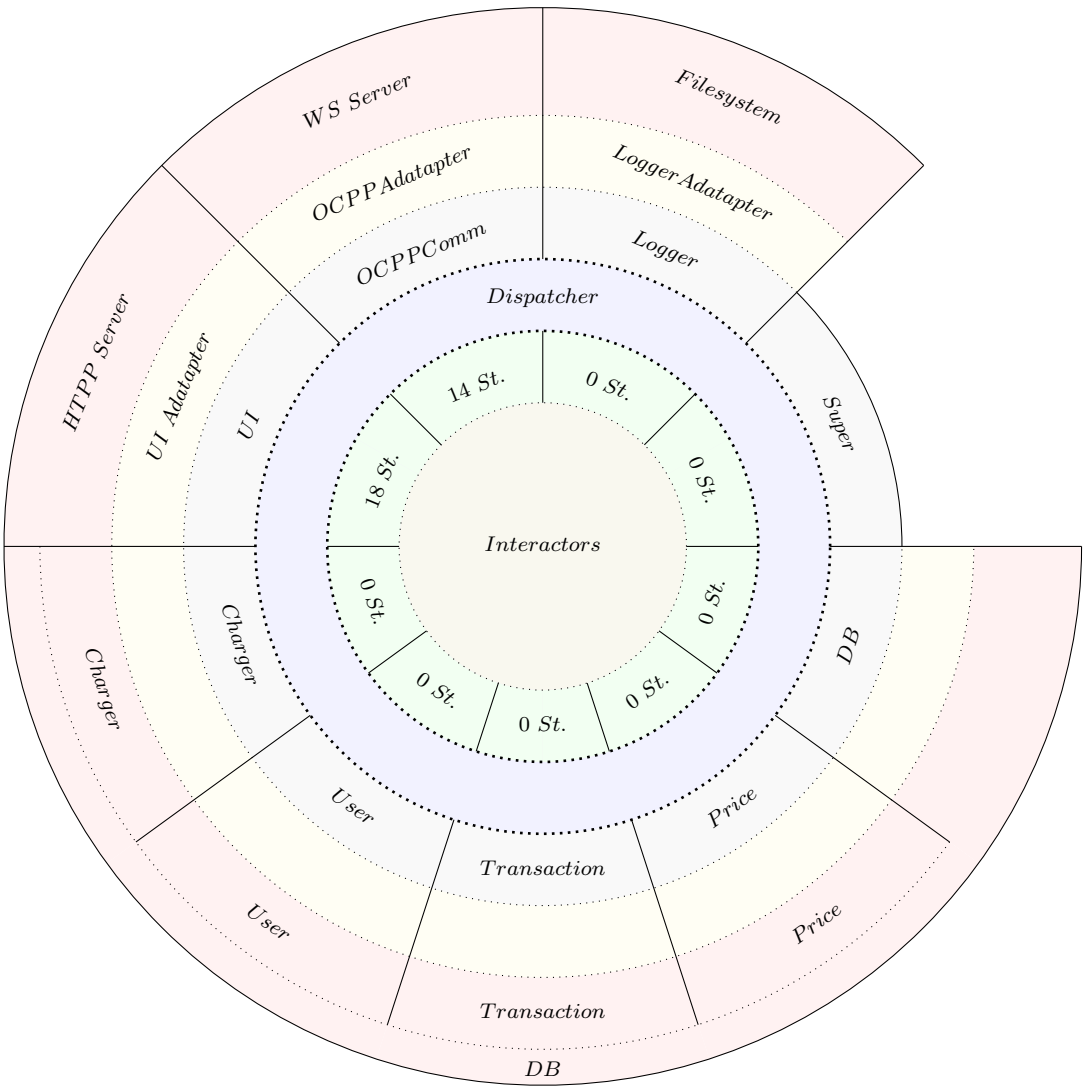


Abbildung 40: Darstellung der umgesetzten Architektur der Standalone Anwendung mit Datenbank

4.3.2 Standalone ohne Datenbank

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

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

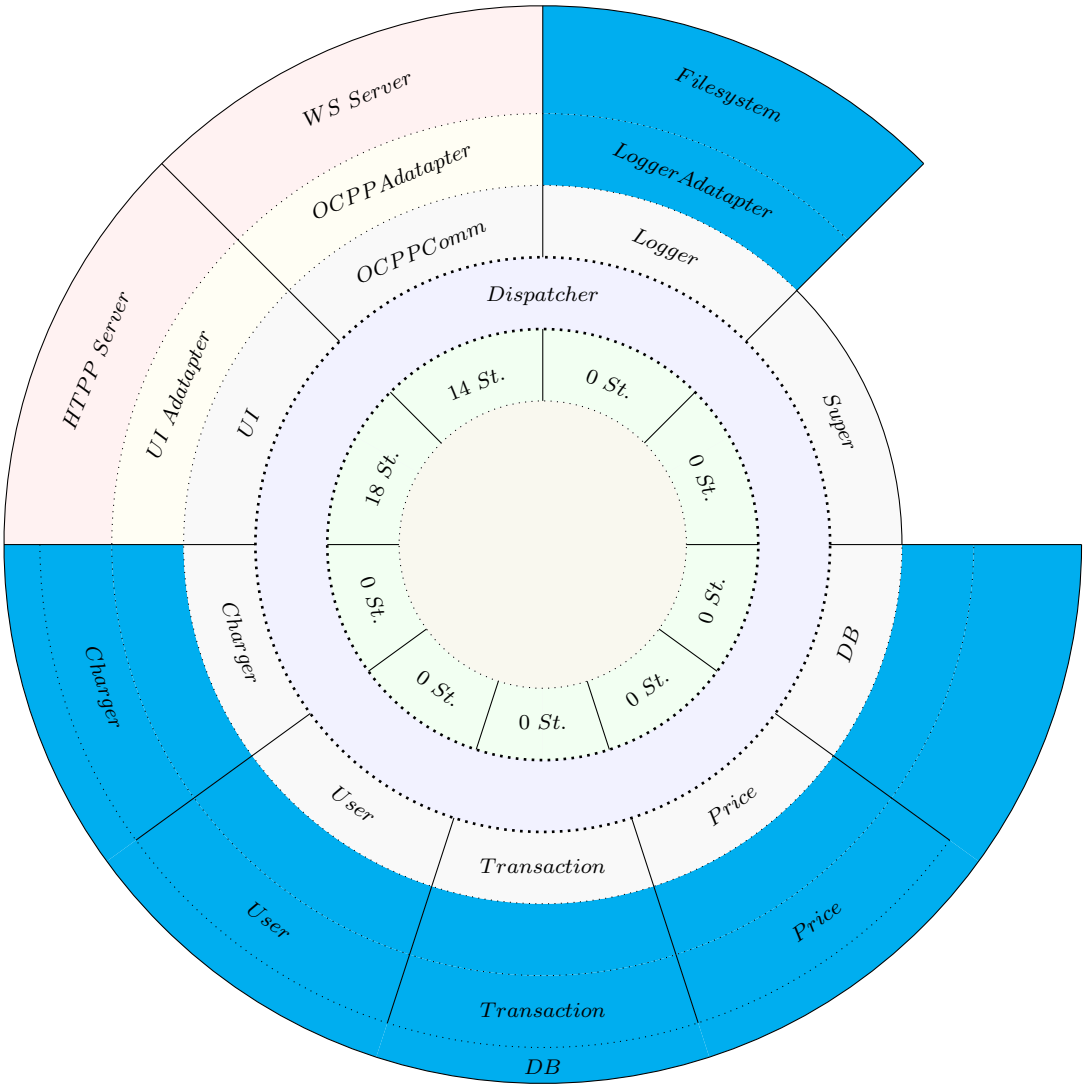


Abbildung 41: Darstellung der umgesetzten Architektur der Standalone Anwendung ohne Datenbank

4.3.3 Bibliothek

Die Bibliothek Anwendung soll in eine andere Hauptanwendung integriert werden. Die Bibliothek soll sich genauso gegenüber der getesteten Ladesäule verhalten, wie eine Standalone-Anwendung. Somit soll Datenbank-Schnittstelle simuliert werden. Es wird auch gewünscht die Logs von der Bibliothek aufzuzeichnen, damit ist eine simulierte Filesystem-Schnittstelle notwendig. Die Bibliothek wird direkt (mittels Methodenaufrufen) über UI-Controller angesteuert, somit wird HTTP-Schnittstelle nicht verwendet.

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

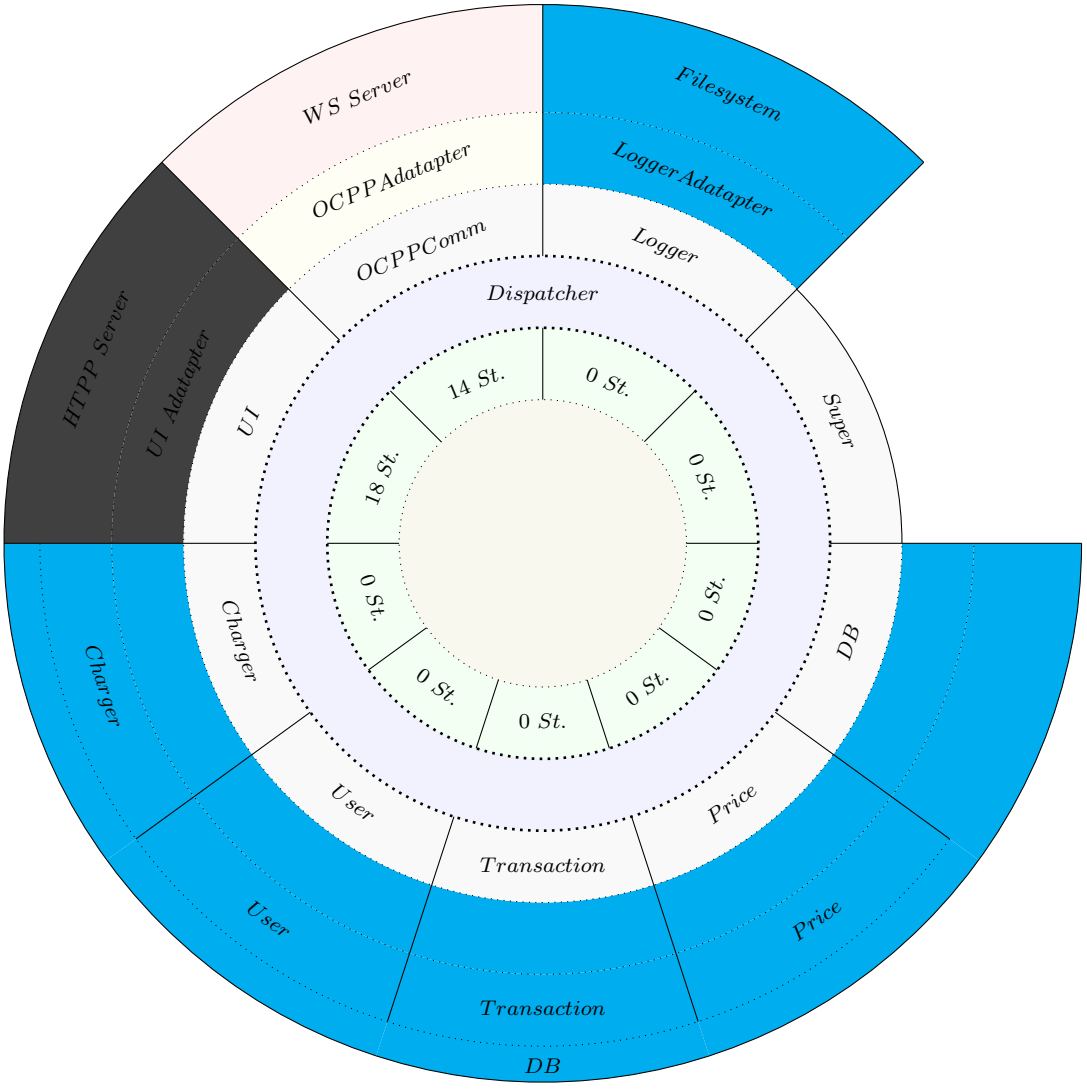


Abbildung 42: Darstellung der umgesetzten Architektur einer Bibliothek

Die Struktur der Bibliothek als Klassendiagramm sieht wie folgt aus:

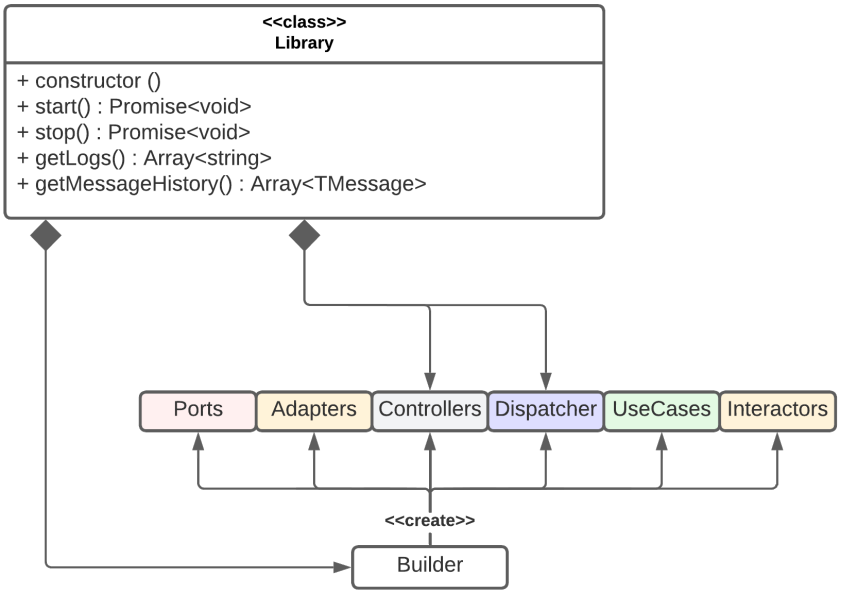


Abbildung 43: Struktur der Bibliothek

4.3.4 Framework

Die Framework Anwendung soll nur OCPP-Schnittstelle benutzen. Die HTTP- und Filesystem-Schnittstelle werden gar nicht gebraucht. Die Datenbank-Schnittstelle soll simuliert werden, da es gewünscht wird, manche Informationen zu speichern um sie später verwenden bzw. auswerten zu können.

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

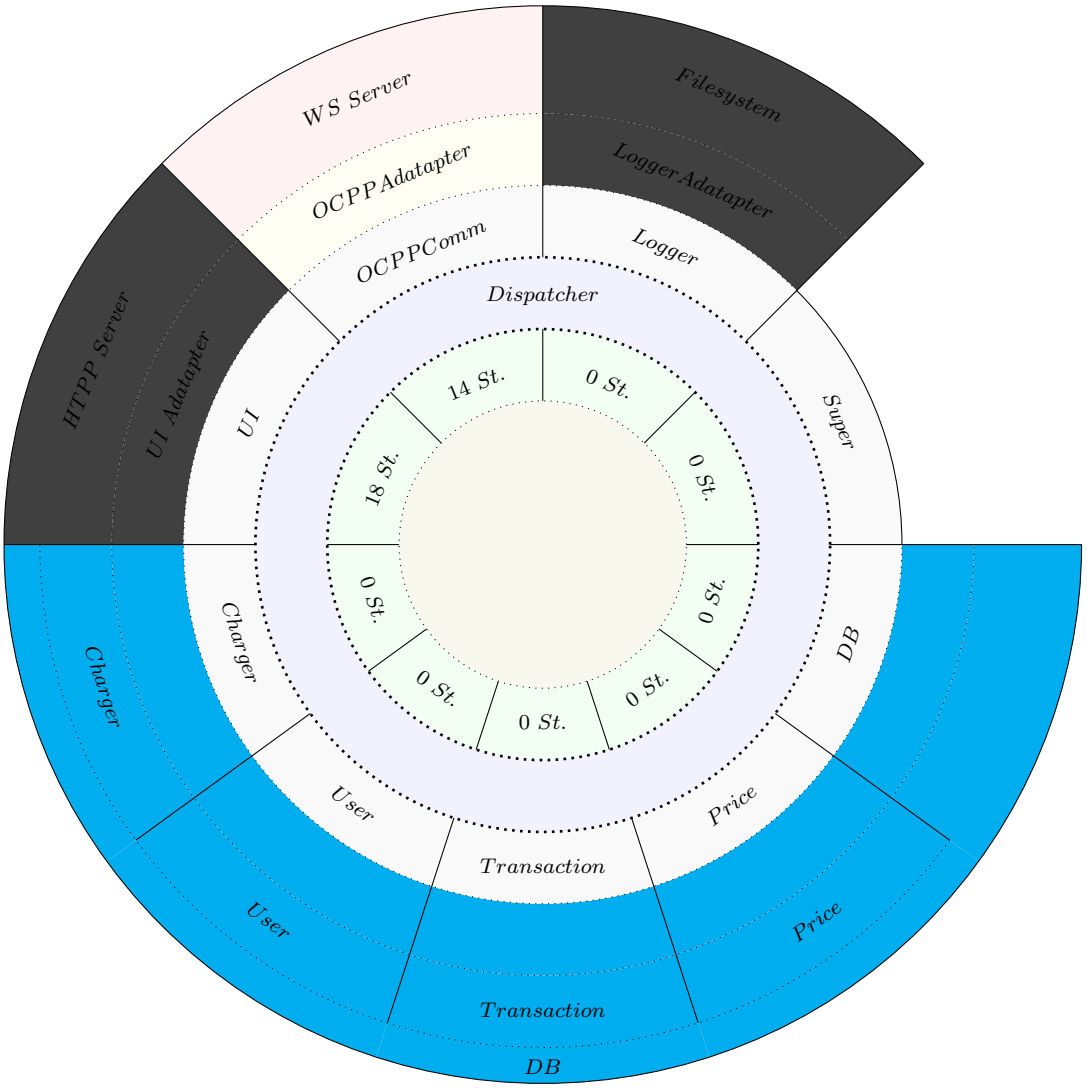


Abbildung 44: Darstellung der umgesetzten Architektur des Frameworks

Das Framework wird von Dritten benutzt. Aus diesem Grund benötigt es ein gutes Interface, das als Fassade (siehe Kapitel 2.1.6) zu der komplexen Struktur in der Abbildung 44 dient. Die allgemeinen Regeln der Aufbau einer Framework-Anwendung sind im Kapitel 3.3.1 beschrieben.

Die Struktur des Frameworks als Klassendiagramm sieht wie folgt aus:

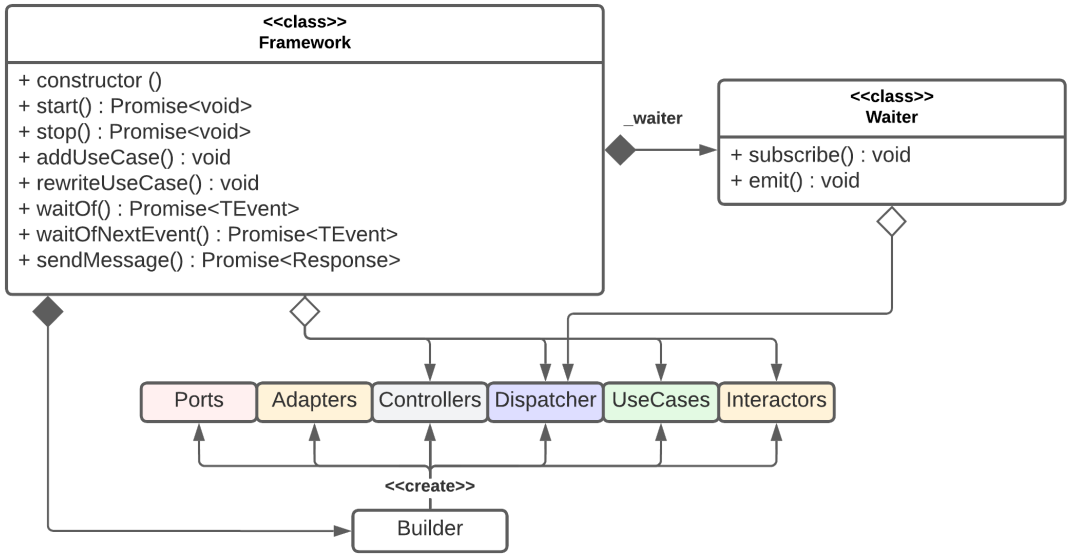


Abbildung 45: Struktur des Frameworks

Das Interface des Frameworks 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 Phasen zugeordnet werden. In der nachfolgenden Abbildung 46 ist eine Zuordnung der Methoden zu jeweiliger Phase in Form eines Ablaufdiagramms zu sehen:

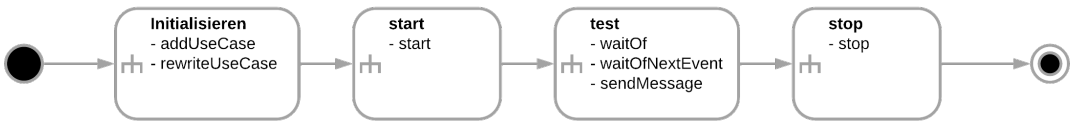


Abbildung 46: Ablaufdiagramm eines Tests

4.4 Vergleich der einzelnen Lösungen

In der Tabelle 2 werden vier Lösungen jeweils miteinander verglichen. In den Zellen werden gemeinsame Codezeilen von zwei Anwendungen abgebildet.

Anzahl an gemeinsame Codezeilen für alle Projekte ist 25.799 (entspricht Abbildung 39).

	Standalone mit DB	Standalone ohne DB	Framework	Library
Standalone mit DB	28.466			
Standalone ohne DB	26.210	26.417		
Framework	25.799	25.972	26.888	
Library	25.799	25.972	25.799	26.112

Tabelle 2: Vergleich der Anzahl der Codezeilen

Literaturverzeichnis

[Fow03] Martin Fowler. *Patterns of Enterprise Application Architecture*. 2003. ISBN: 978-0-321-12742-6.

[Mar09] Robert C. Martin. *Clean Code*. 2009. ISBN: 978-0-13-235088-4.

[Mar18] Robert C. Martin. *Clean Architecture*. 2018. ISBN: 978-0-13-449416-6.

[Anda] Martin Andenmatten. *CD –CONTINUOUS DELIVERY*. URL: https://blog.itil.org/2016/07/wort-zum-montag-cd-continous-delivery/ (besucht am 20. Sep. 2022).

[Andb] Dossier Andreas. *Hexagonal architectur*. URL: https://www.dossier-andreas.net/software_architecture/ports_and_adapters.html (besucht am 9. Sep. 2022).

[Aug] Stephan Augsten. *Inversion of control*. 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 (besucht am 14. Sep. 2022).

[Com] Redaktion ComputerWeekly.de. *Bibliothek (Library)*. URL: https://www.computerweekly.com/de/definition/Bibliothek-Library%5C#:~:text=In%5C%20der%5C%20Informatik%5C%20bezeichnet%5C%20der,und%5C%20physische%5C%20Speichereinheiten%5C%20wie%5C%20Tapes. (besucht am 20. Sep. 2022).

[dew] dewiki. *Dependency-Inversion-Prinzip*. URL: https://dewiki.de/Lexikon/Dependency-Inversion-Prinzip (besucht am 19. Sep. 2022).

[Fowa] Martin Fowler. URL: https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf (besucht am 23. Juli 2022).

[Fowb] Martin Fowler. URL: https://martinfowler.com/bliki/InversionOfControl.html (besucht am 20. Sep. 2022).

[Fowc] Martin Fowler. *Is High Quality Software Worth the Cost?* URL: https://martinfowler.com/articles/is-quality-worth-cost.html (besucht am 29. Mai 2019).

[Lan] Jesko Landwehr. *Was ist ein Framework? – Definition & Erklärung*. URL: https://it-talents.de/it-wissen/framework/ (besucht am 20. Sep. 2022).

[OCA] OCA. URL: https://www.openchargealliance.org/protocols/ocpp-201/ (besucht am 12. Sep. 2022).

[Sch] Andi Scharfstein. *Migrating our JUnit 3 Tests to JUnit 4*. URL: https://www.cqse.eu/de/news/blog/junit3-migration/ (besucht am 20. Sep. 2022).

[Shv] Alexander Shvets. *OOP Design Patterns*. URL: https://refactoring.guru/ (besucht am 19. Sep. 2022).

[Vya] Anshul Vyas. *Model View Presenter*. URL: https://anshul-vyas380.medium.com/model-view-presenter-b7ece803203c (besucht am 20. Sep. 2022).