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

2. 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.

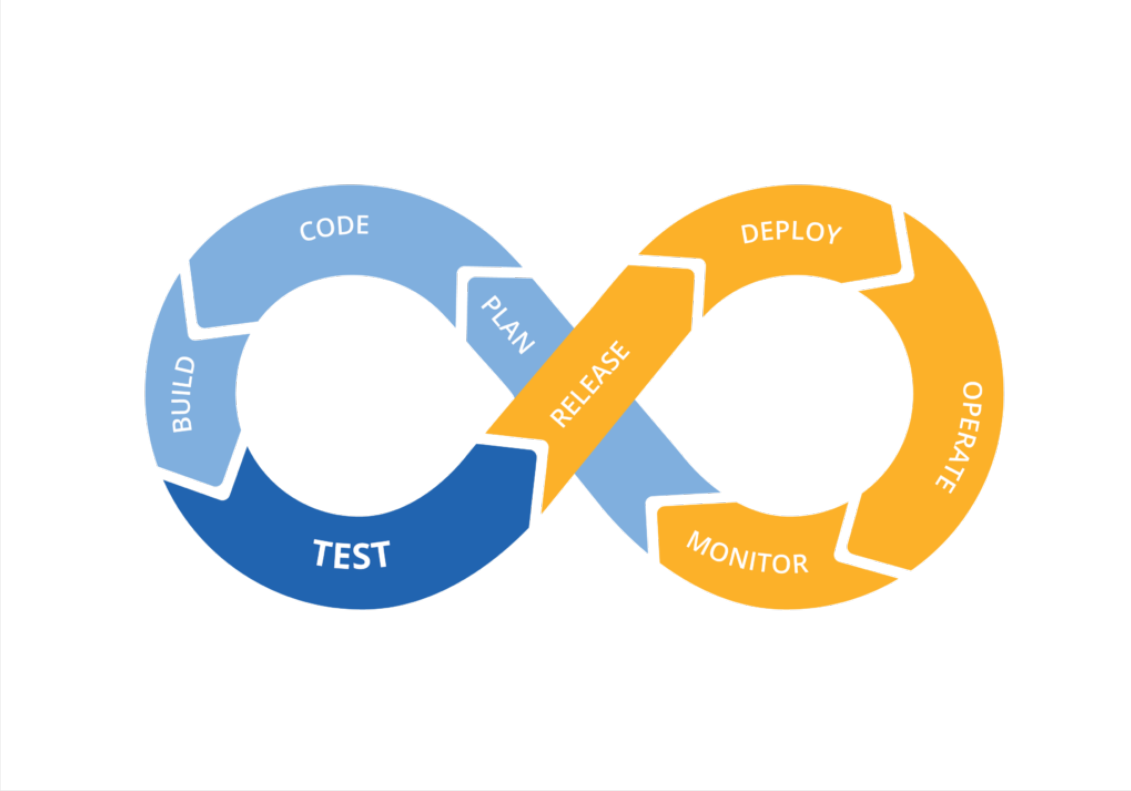


Abbildung 1: CI/CD Pipeline [1]

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 großte 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

SomeIntroducation

2.1 OOP Design Patterns

some Introduction

2.1.1 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.

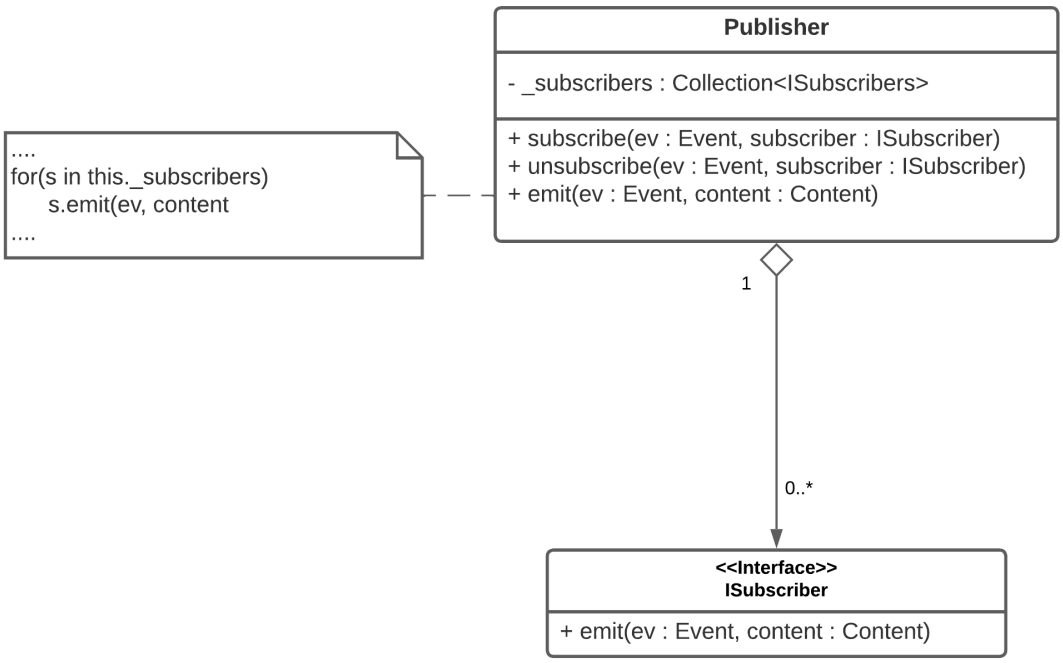


Abbildung 2: Klassendiagrammm Observer [2]

2.1.2 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.

2.1.3 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.

2.1.4 Builder

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

2.1.5 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.

2.1.6 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 man weißt, man hat nur eine Instanz der Klasse im Programm und auf sie möchte man aus allen Stellen der Anwendung zugreifen. 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.

2.1.7 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 werden von der Fabrik übergeben.

2.2 Architekturen

2.2.1 Clean Architectur

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

Die Architecture besteht aus konzentrischen Kreisen. Die Kreise repräsentieren verschiede Aufgaben, die eine Anwendung erledigen kann. In der Abbildung 3 sind vier Kreise dargestellt. Die Anzahl an Kreisen kann sich variieren, solange Dependency Rule erfüllt wird.

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

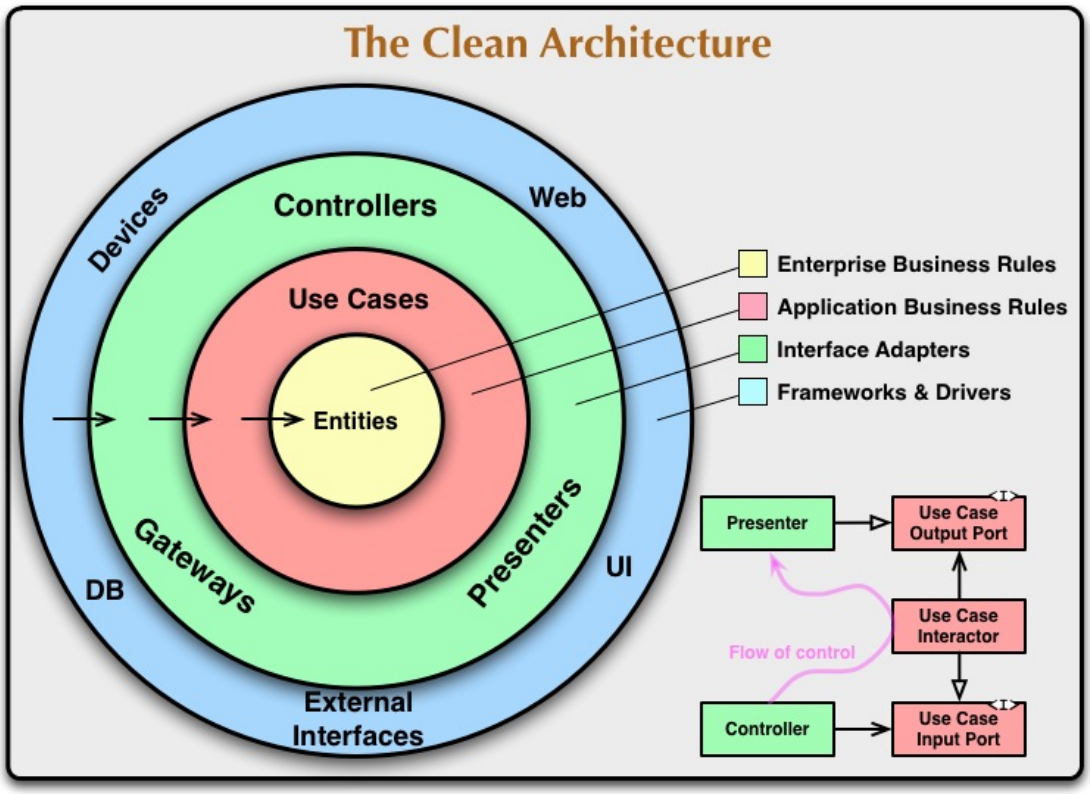


Abbildung 3: Clean Architecture [3]

[3] 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 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.
- Jedes neues **View** braucht ein eigenes **Presenter**.

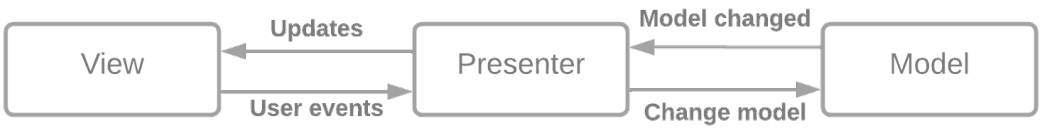


Abbildung 4: Datenfluss in MVP Architektur [4]

Der Datenfluss in einer Anwendung, die mittels MVP implementiert ist:

- 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 SSpeichern"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 SSpeicherßoll grün werden).
- View abarbeit das Ergebnis und zeigt es an (z.B. Button wird grün auf der Oberfläche angezeigt)

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 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

Some text to testing pyramid

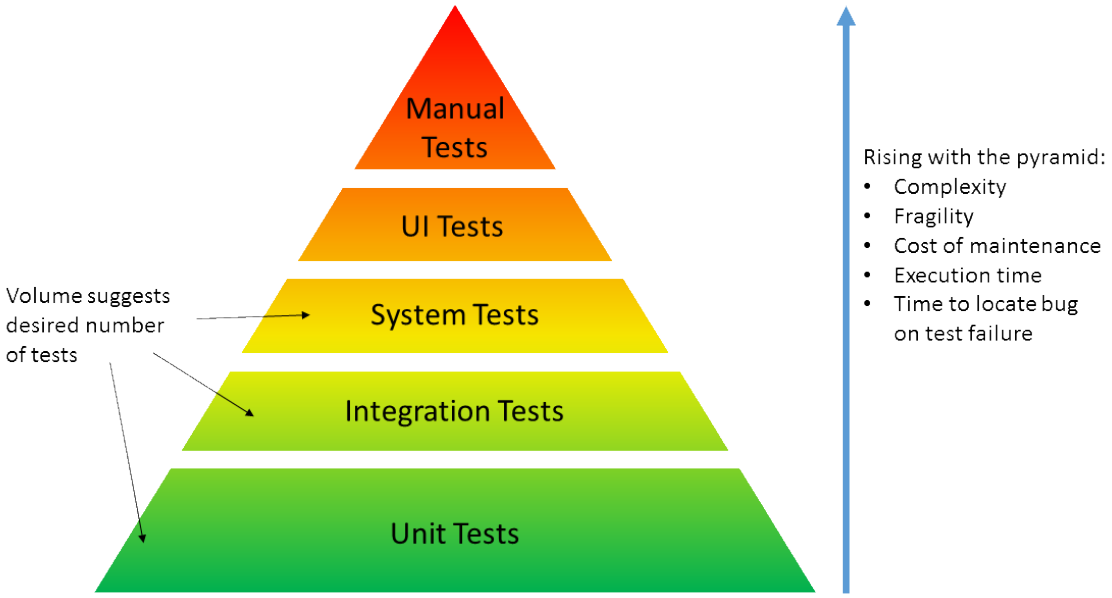


Abbildung 5: Caption written below figure [5]

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

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

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 diese 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 komplete 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 es aber nur die Ausgaben an Benutzeroberflächen getestet werden und alle anderen internen Abläufe ignoriert werdern, ist die Anzahl an Tests sehr wenig.

2.3.6 Manual Tests

some Info about manual tests

2.4 SOLID

some info about solid

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, ist es ein Signal das Modul in zwei Module aufzuteilen.

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

2.4.2 Open–closed principle

OCP

2.4.3 Liskov substitution principle

LSP

2.4.4 Interface segregation principle

ISP

2.4.5 Dependency inversion principle

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 abhängen. Beide sollten von Abstraktionen abhängen.

Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

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

Abbildung 6: Schlechte laut DIP Abhängigkeit [6]

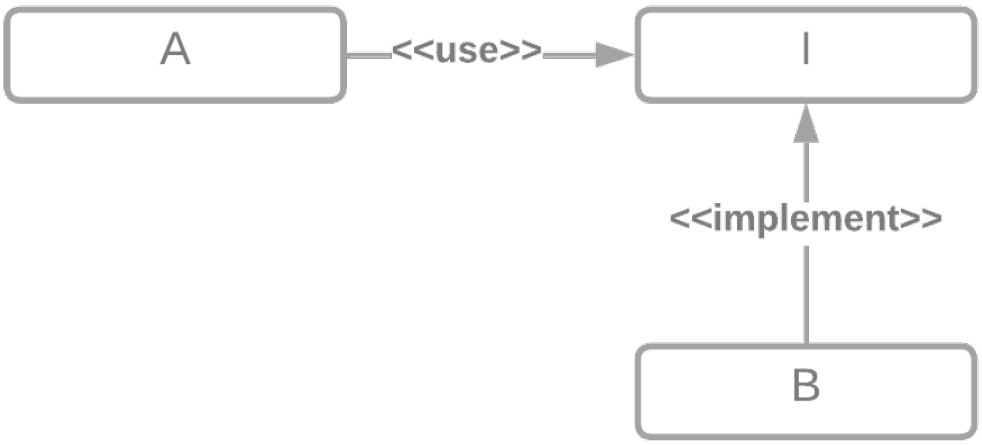


Abbildung 7: Gute laut DIP Abhängigkeit [7]

2.5 GRASP

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

2.6 OOP Principles

some info aboout oop principles

2.6.1 Abstraction

some about abstractions

2.6.2 Encapsulation

some abput Encapsulation

2.6.3 Inheritance

some about Inheritance

2.6.4 Polymorphism

some about Polymorphism

2.7 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) Das Prinzip entkoppelt die Teile der Anwendung voneinander, indem die Reaktion nicht mehr vom Erstllen gebunden ist.

Beispiel für IoC ist die Reaktion auf ein Ereignis mittels einem Callback. Man schreibt eine Methode(Callback), die an eine externe Stelle übergeben wird und beim Geschehen des Ereignisses zum späteren Zeitpunkt ausgeführt wird. Man hat dabei keine Kontrolle darüber, wann das Ereignis geschieht und wie die Zuordnung des Callbacks passiert.

Ein weiteres Beispiel von IoC ist “Dependency Injection”

2.8 Dependency Injection

In der Abbildung 8 sieht man ein Beispiel von einer Anwendung, die die von der Konsole ankommenden Zahlen quadriert und das Ergebnis zurückgibt.

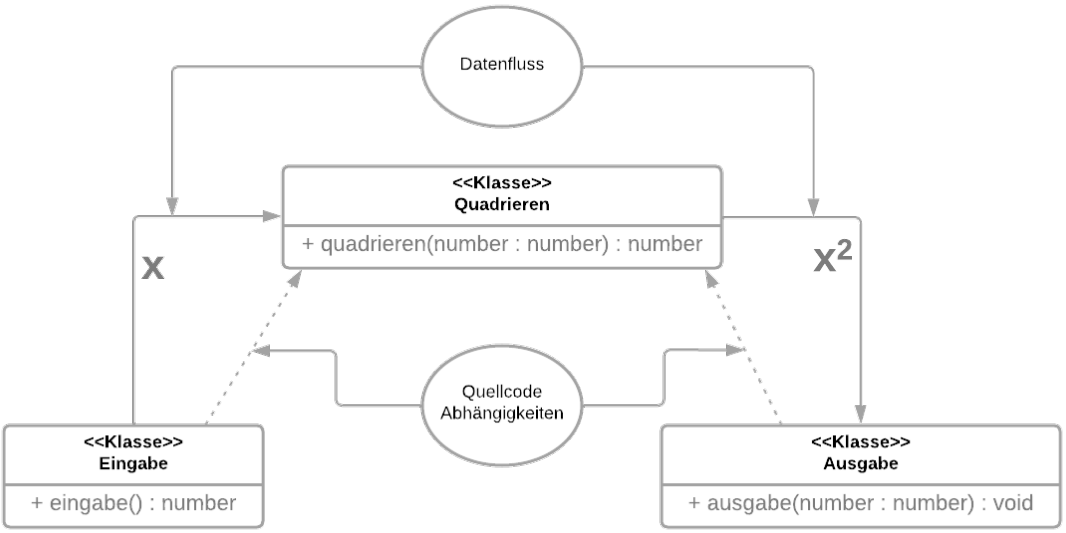


Abbildung 8: Datenfluss und Quellcode Abhängigkeiten

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

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

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

Die Lösung sieht so aus:

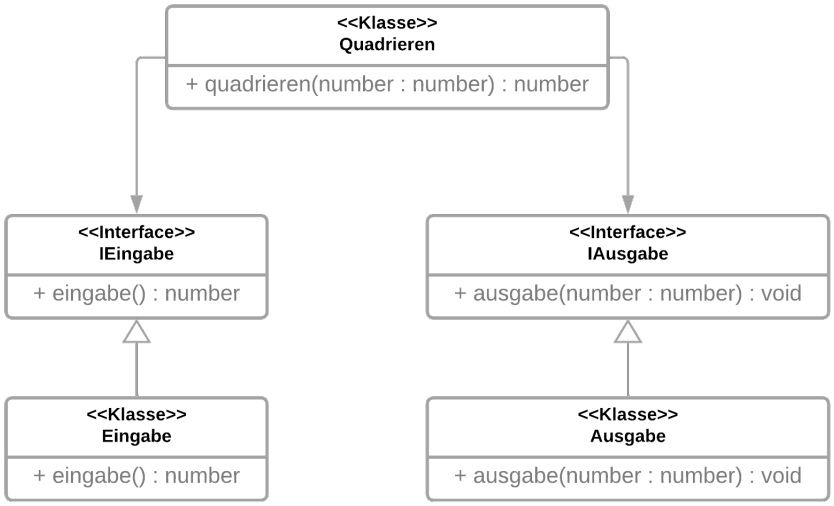


Abbildung 9: Entkopplung der Abhängigkeiten

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

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

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

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

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

3 Software Architektur

Es existiert keine einheitliche Definition einer Software Architektur. Verschiedene Authoren definieren es auch 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 meiner Meinung nach müssen beide 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 inverstierter 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 von nicht Softwareentwicklern definiert und beziehen sich auf nicht Informatikgebiete (z.B. Banksoftware oder eine Smartphone Anwendung). Das Erfüllen von diesen Anforderungen ist das Ziel von jedem Softwareprojekt. Jedoch bei der Umsetzung entstehen viele Herausforderungen, die für die Außenstehenden 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 der 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 (Aufsetzung) der Software.
- Maintenance (Unterstützung) der Software.

Deployment der Software beinhaltet die Kosten die durch das Aufsetzen der neuen Version der Software entstehen.
Maintenance der Software beinhaltet die Kosten, die nach dem Beenden der Entwicklung bei kleineren Erweiterungen und Änderungen des Systems entstehen.

3.1.2 Technische Schulden

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

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

Die technischen Schulden entstehen dadurch, dass bei der Entwicklung eines Teiles des Systems wurde von den Entwicklungsteam weniger Zeit 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.

Die Softwareentwickler können die ankommenden Aufgaben erledigen

- man hat bereits Vorgaben wie die Kommunikationswege zwischen den Modulen ist
- wie die Module benannt werden sollen
- an welchen Stellen das Modul in das System hinzugefügt werden soll
- die Menge an durch den SZufallëntstehenden 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 ableiten.

Daher ist es vom 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 die unterschiedliche Menge an Zeit inverstiert wurde.

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

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

- 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 sind. Mit Hilfe folgender Kriterien können einfacher Entscheidung getroffen werden:

- Wann soll die MVP[1] 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.

[1] Minimum viable product

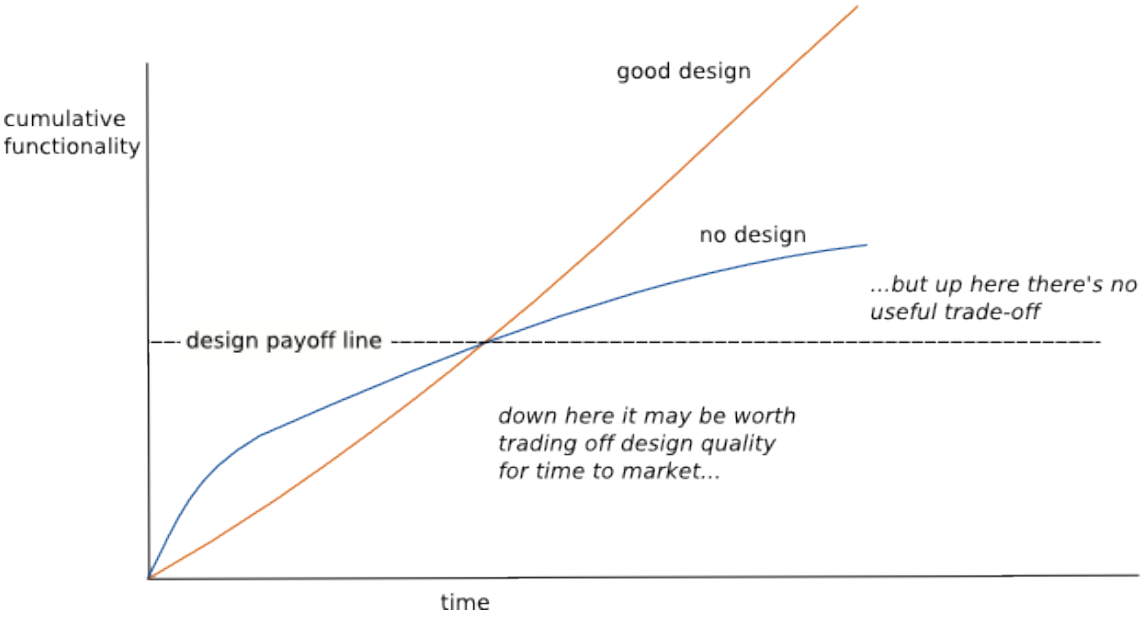


Abbildung 10: Vergleich einer guten und einer schlechten Softwarearchitektur [8]

Diesen Verlauf ist beim Projektbegin zu beachten. Es ist nicht vom Vorteil bei einem Projekt, für das man nur eine Woche Zeit hat, eine gute Architektur zu implementieren, die ohne jeglichen Funktionalitäten mehrere Wochen gebrauchen wird.

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

3.1.4 Testbarkeit der Software ist wichtig

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

[8] 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.

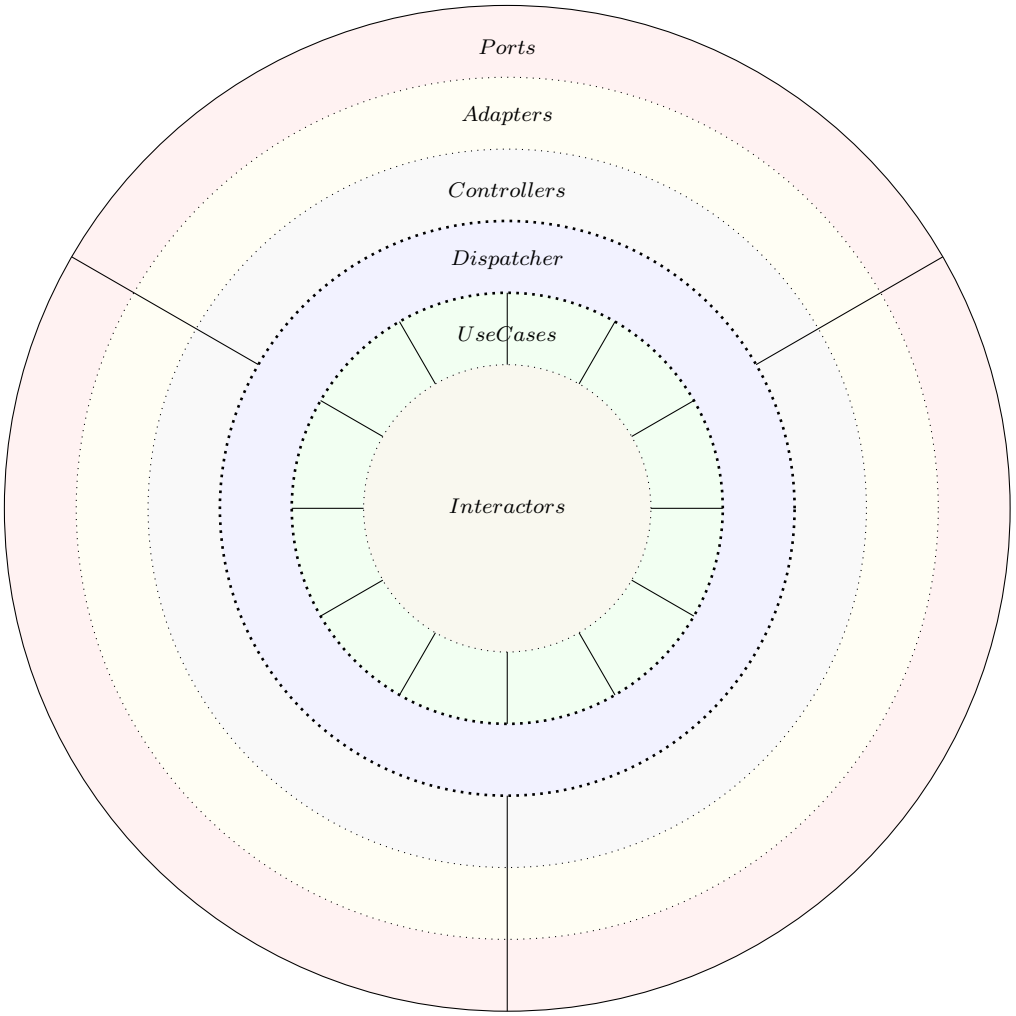


Abbildung 11: Darstellung der umgesetzten Architektur als **Clean Architecture** mit 7 Schichten. Die Linien repräsentieren die Grenzen zwischen den einzelnen Teilen des Programms [9]

Beschreibung der Darstellung:

Jede Komponente bringt in das gesamt Programm folgende Teile:

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

- UseCase - definiert den Ablauf an Tätigkeiten (Interactoren) beim Geschehen eines Ereignisses
- Interactors - Hülle für alle definierten Tätigkeiten im Controller

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 ist schneller als mit reelen Schnittstellen und man hat das gleiche Ergebnis bezüglich des Verhaltens des Programms.
- Die Innere Logik ist nicht an Schnittstellen gebunden, somit können alle Schnittstellen mit wenig Aufwand getauscht werden.

3.2.1.1 Port-Adapter-Controller

Am nähesten zu der **PAC**[10] Struktrur ist das Pattern **MVP**[11]. Dabei die Aufgaben von **View** entsprechen den Aufgaben von **Port**. Die Aufgaben von **Presenter** entsprechen den Aufgaben **Adapter**. Und die Aufgaben von dem Rest des Programms inklusiv **Controller** entspechen den Aufgaben von **Model**.

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

Die Aufgabe von drei Komponenten ist die Steurung von einem Aufgabengebiet (z.B. HTTP Schnittstelle). Der Grund für das Aufteilen in 3 verschiedene Teile (Adapter, Port und Controller) ist **SRP** (Kapitel 2.4.1). Somit hat jedes Teil eine Aufgabe und lässt sich besser unterstützen.

[10] Port-Adapter-Controller
[11] Model-View-Presenter

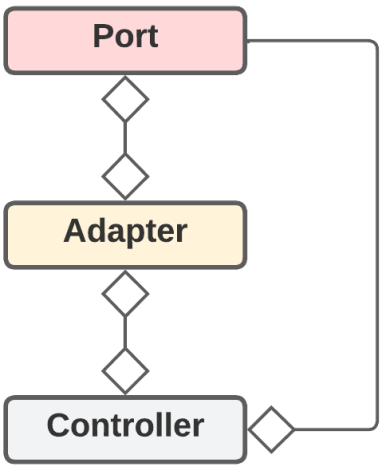


Abbildung 12: Objektendiagramm PAC [12]

3.2.1.2 Controller-Dispatcher-UseCase-Interactor

In dem Teil wird die komplete innere Logik der Application beschrieben. Dafür ist nur **Controller** notwendig. D.h. beim Geschehen eines Ereignisses wird eine (oder mehrere) Method(en) in deb 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 wiederspricht 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. **Contoller** besitzt eine weitere Verantwortlichkeit die sich in ein anderes Teil verschieben lässt. Dieses Teil heißt **Dispatcher** und seine Aufgabe ist alle Abonnenten **UseCase**s 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.

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

Wenn alles zusammengeführt wird, entsteht folgendes Objektendiamm:

[12] Eigene Quelle

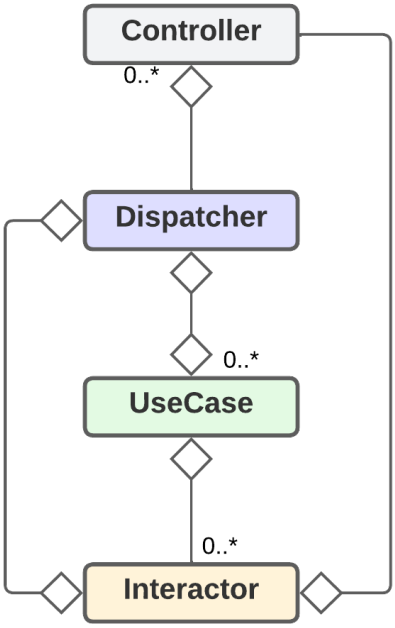


Abbildung 13: Objektendiagramm Controller-Dispatcher-UseCase-Interactor

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

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

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

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

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

Ein möglicher Ablauf wäre:

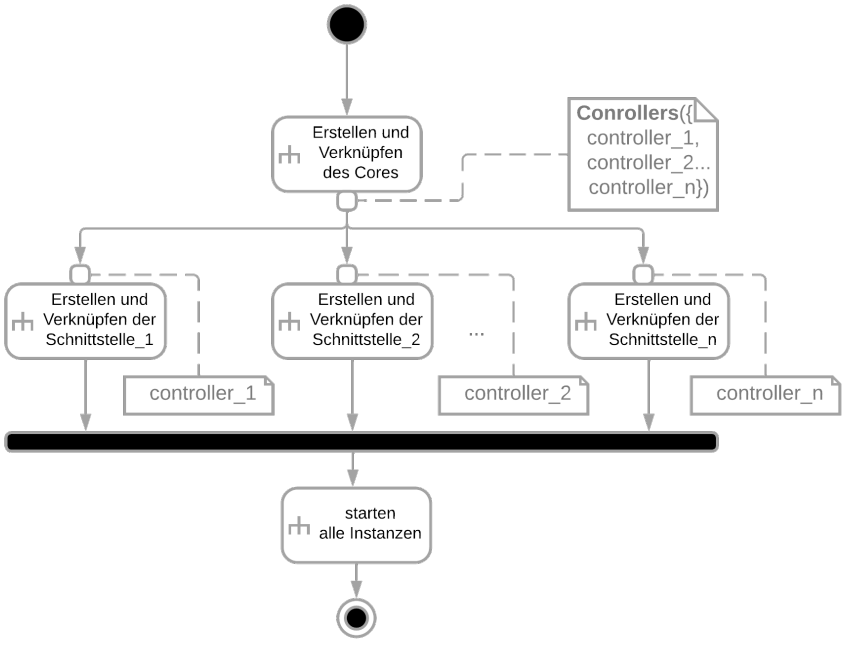


Abbildung 14: Ablaufiagramm Erstellen der Struktur [13]

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 man die Konstruktoren ändert.

Es gibt folgende Möglichkeiten das Problem zu lösen:

- Globale Objekte bzw. Instanzen (z.B. OOP Design Pattern **Singleton** 2.1.6)
- Initialisiereung von den Entsprechenden Instanzen und Übergeben in dem Konstruktor (OOP Sprachen) oder mittels einer Settermethode.

Bei der ersten Implementierung hat man das Problem, dass die benutzten Utility Controllers nicht ersetzbar sind. D.h. man kann dann alle Module sehr schwer mit Unittests abdecken, denn es werden immer auch die Utility Controllers mitgetestet. Ein weiteres Problem besteht darin, dass alle Utility Controllers eine Infrastruktur benutzen. (z.B. Logs in dem Datenbank speichern). Jeder Test wird dadurch deutlich länger laufen. 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 übergeben oder mittels einer Settermethode der Instanz. Die erste Möglichkeit führt zu einer längeren Parameterliste, die die Lesbarkeit des Codes erschwert. Die zweite Möglichkeitführt dazu, dass der Aufruf der Methode von dem Softwareentwickler vergessen werden kann.

Das Problem lässt sich zum Beispiel durch OOP design pattern **Factory** (Kapitel 2.1.7), 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.

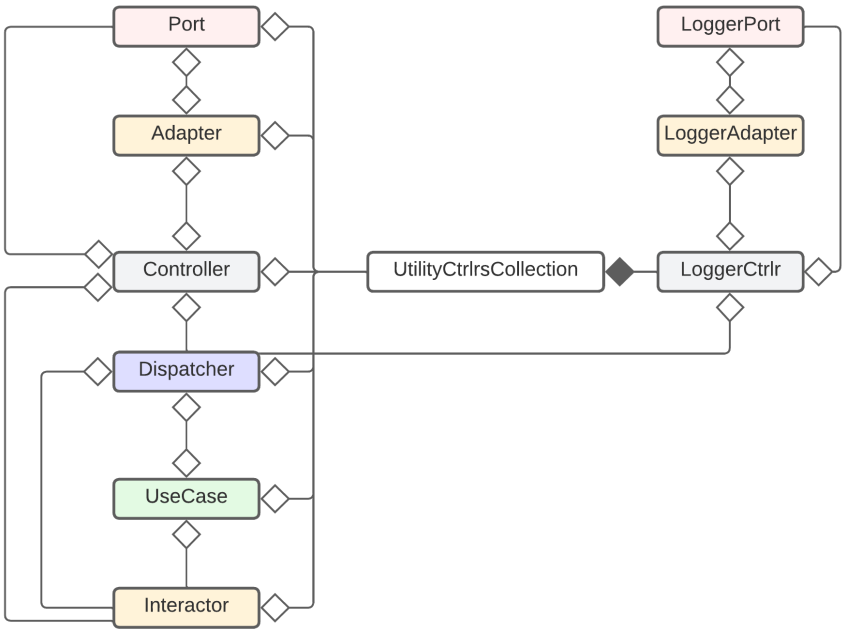


Abbildung 15: Objektendiagramm mit Utility Controllers [14]

3.2.1.5 Verbindung der einzelnen Schichten miteinander und die Testbarkeit

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

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

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

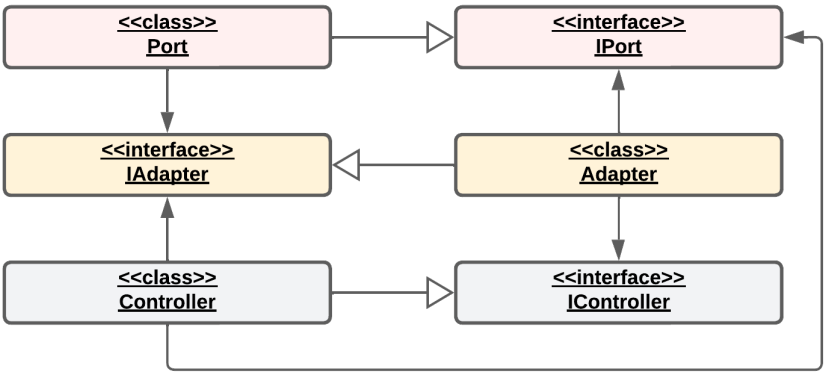


Abbildung 16: Klassendiagramm Port-Adapter-Controller [15]

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

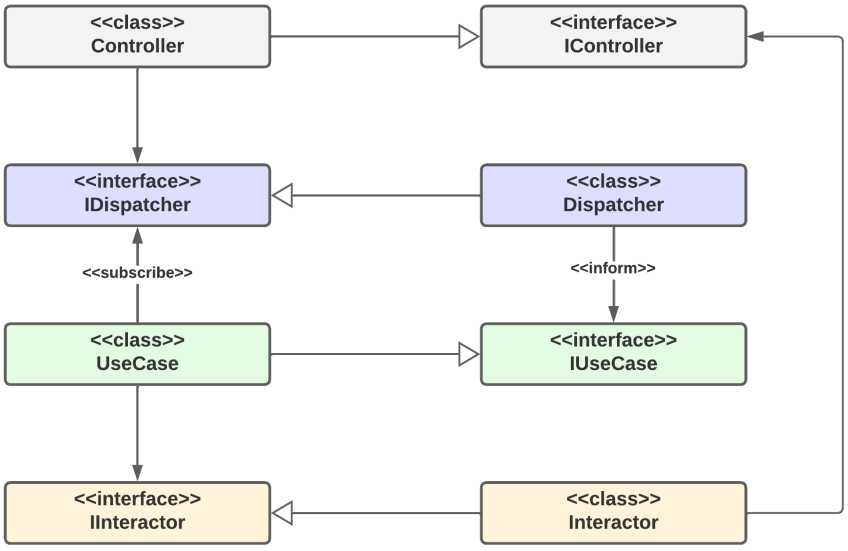


Abbildung 17: Klassendiagramm Controller-Dispatcher-UseCase-Interactor [16]

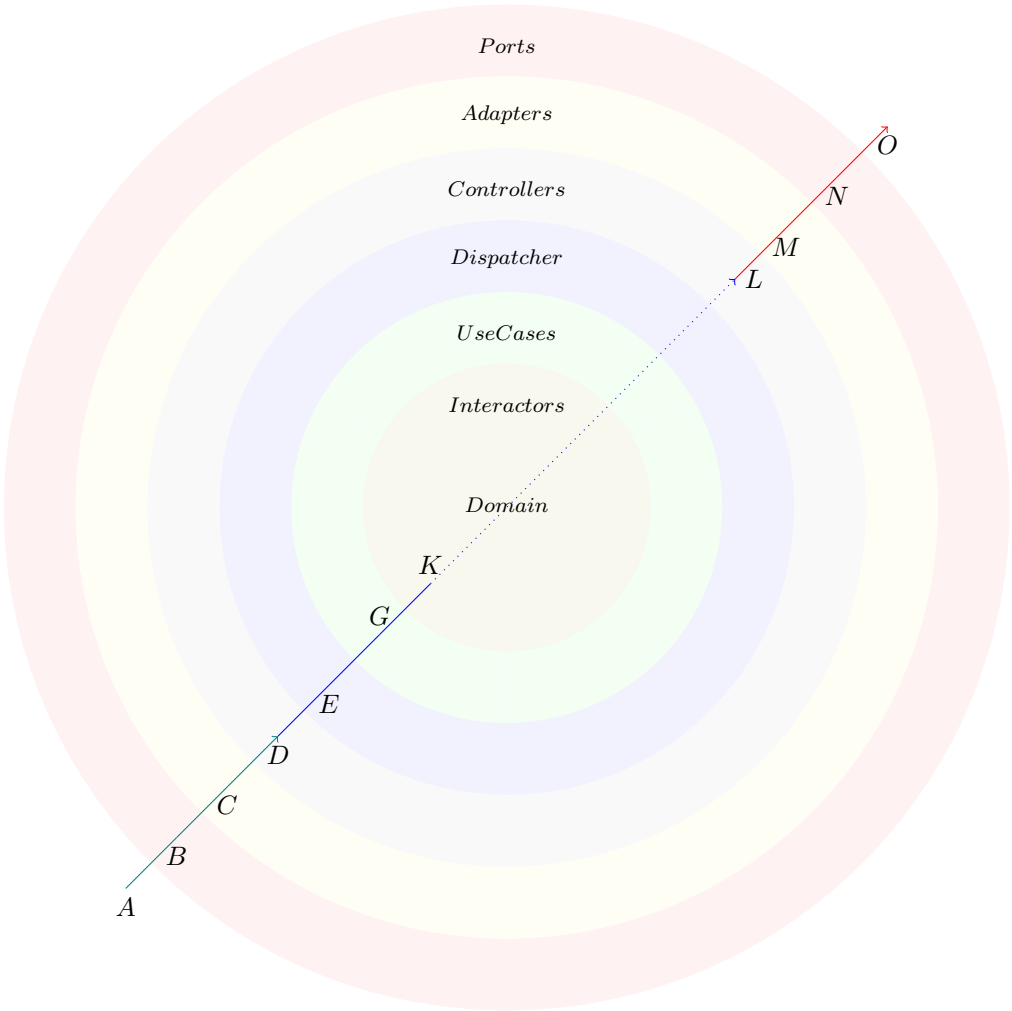
Im Klassendiagramm 16 und 17 sind alle Klassen miteinander über ein Interface verbunden. Dies ermöglicht leichte und schnelle Ersetzbarkeit der Schichten und somit lässt sich jeder Zustand der Umgebung um einer Schicht simulieren. Das ist Voraussetzung für Testbarkeit jeder einzelnen Schicht unabhängig 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. (blau) Controller löst ein Ereignis im Dispatcher aus.
2. (rot) Controller spricht sein Port an(z.B. Speichern der Daten in der Datenbank oder OCPP Antwort abzuschicken)
3. (grün) Das Programm wird von einem externen System angesprochen (z.B. Ladesäule schickt eine OCPP Nachricht an den Server)

Wenn das geschehen ist, sieht der Datenfluss so aus:



——— *Teile des Programms wissen nichts voneinander*

Abbildung 18: Darstellung der umgesetzten Architektur als **Clean Architecture** mit sieben Schichten. Pfeile repräsentieren mögliche Datenflüsse [17]

Darstellung des Datenflusses **1** als **Sequencediagram**:

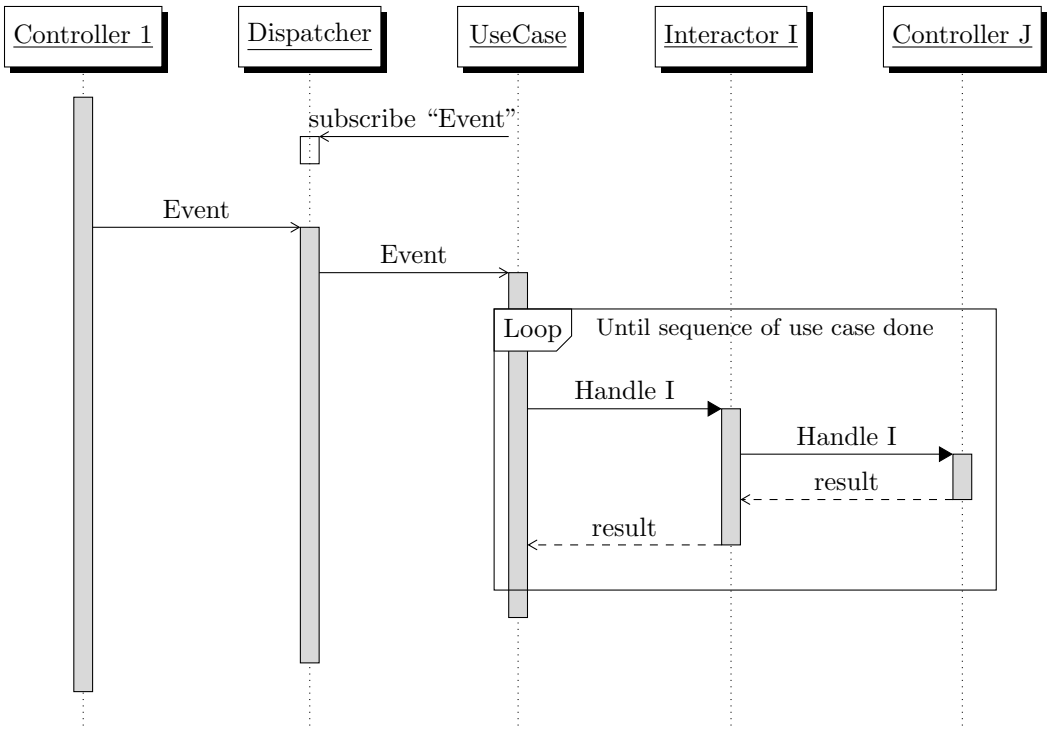


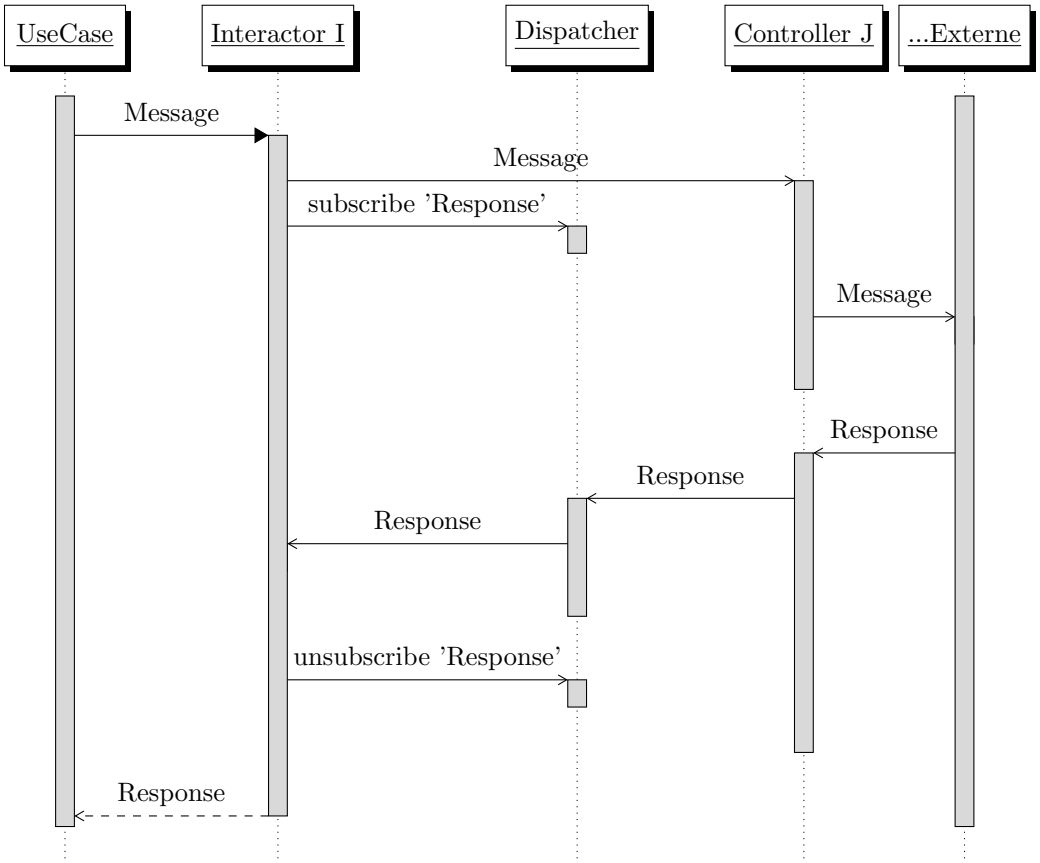
Abbildung 19: Sequencediagramm vom Datenfluss “1” Blau

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

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

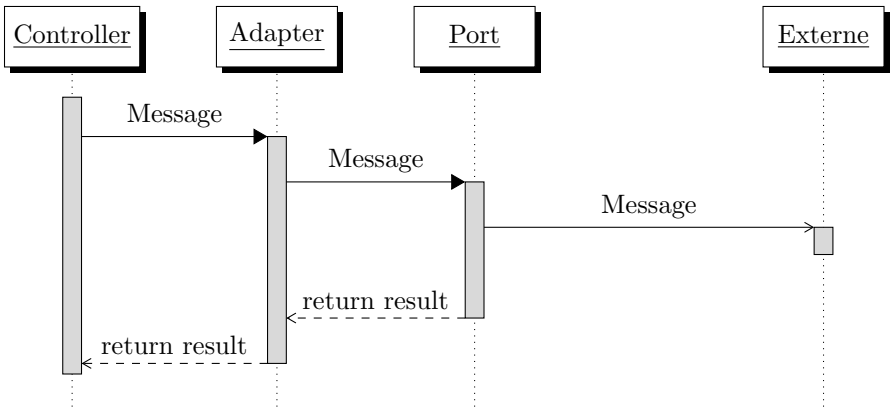
1. synchron - der Rückgabewert ist das Ergebnis der aufgerufenen Methode
2. asynchron - auf die dazugehörige Antwort vom Port wird gewartet(z.B. auf OCPP Response warten, wenn man ein OCPP Request abschickt)

Darstellung der 2. Möglichkeit:



Der Rückgabewert wird beim synchronen Funktionsaufruf zurückgegeben, wie in der Abbildung 19 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**, die alle Abonnierten darüber informiert, unteranderem auch den **Interactor**. **Der Interactor** gibt diese Antwort als Rückgabewert der Funktionsaufruf

Darstellung des Datenflusses “2” als “sequencediagram”:



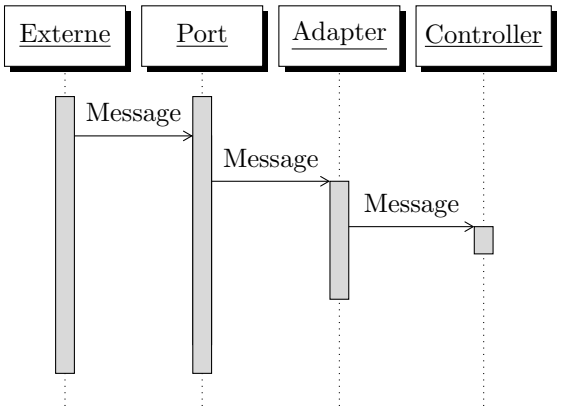
Darstellung der Datentransformation:
Controller - Adapter: Alle Informationen werden als Objekte übergeben, die im Domain definiert werden müssen.

```
    OCPP20Message({
        destination: {
            chargerId : "some_unique_charger_id"
        },
        message : {
            name : "BootNotification",
            type : "Response",
            payload : BootNotification({
                currentTime : Date(Thu Jul 28 2022 14:26:49 GMT+020
                    0),
                interval : 30,
                status : "Rejected"
            })
        }
    })
```

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

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

Darstellung des Datenflusses “3” als “sequencediagram”:


Externe
Port
Adapter
Controller
Message
Message
Message

3.2.2.1 Logging

Ein wichtiger Bestandteil jeder Software ist das Logging von unterschiedlichen Ereignissen in der Software. Das Ziel vom Logging ist später von den Softwareentwicklern verschiede 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 untere Abbildung 20 zeigt den kompleten 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 lässen sich die Fehler auf die Komponente genau lokalisieren, d.h. man kann einen entsprechenden Unittest schreiben, der diesen Fehler abdeckt, bzw. einen Integrationtests, da man auch den Ablauf des Ereignisses kennt.

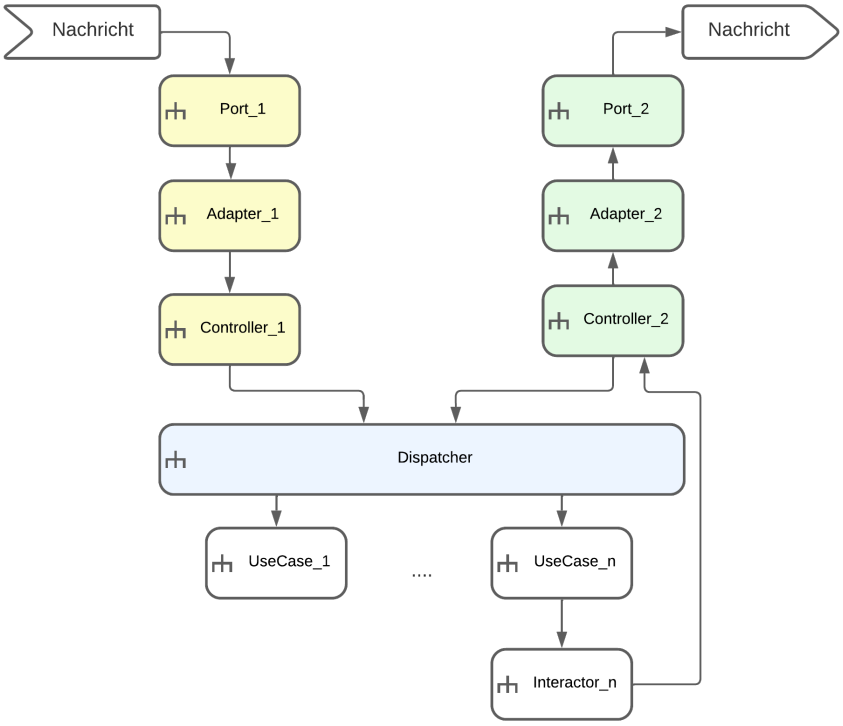


Abbildung 20: Kompletter Datenfluss [18]

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 gestartet wird)

3.2.3.1 Verhalten für ein Ereignis erweitern

Bei diesem Szenario geht man davon aus das ein Ereignis an **Dispatcher** ankommt und entspechend an alle dafür verantwortlichen **UseCases** weiterleitet. Es gibt hier zwei Möglichkeiten:

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

Mit der ersten Möglichkeit hat man das komplete Verhalten für ein Ereignis an einem Ort und mit Unittests abdecken.

Die zweite Möglichkeit streut das Verhalten für ein Ereignis im Projekt, was eine besere Lesbarkeit des jeweiligen Teils erhöht, 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 man eine neue Schnittstelle hinzufügen möchte, 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 bringt 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 3.2.2 folgendeweise vereinfachen:

Abbildung 21: Vereinfachte Darstellung

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

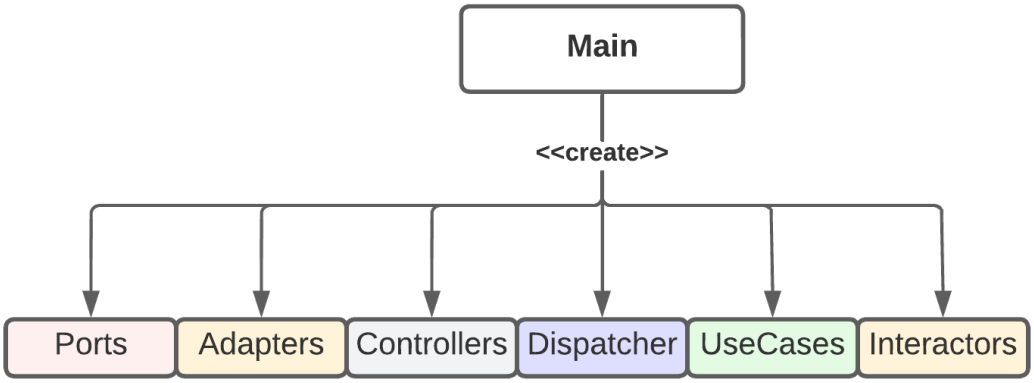


Abbildung 22: Vereinfachte Darstellung einer Standalone Anwendung

Der Datenfluss lässt sich so darstellen:

Abbildung 23: Vereinfachte Darstellung einer Standalone Anwendung

Wenn man es als Komponente in einer anderen Anwendung verwenden möchte, braucht die Struktur eine Fassade (Kapitel 2.1.5), damit man auf wichtige Teile der Komponente zugreifen kann und der Rest verborgen bleibt. Die Fassade baut die gesamte Struktur der Komponente auf.

Eine fertige Komponente, die man in die anderen Anwendungen integrieren kann:

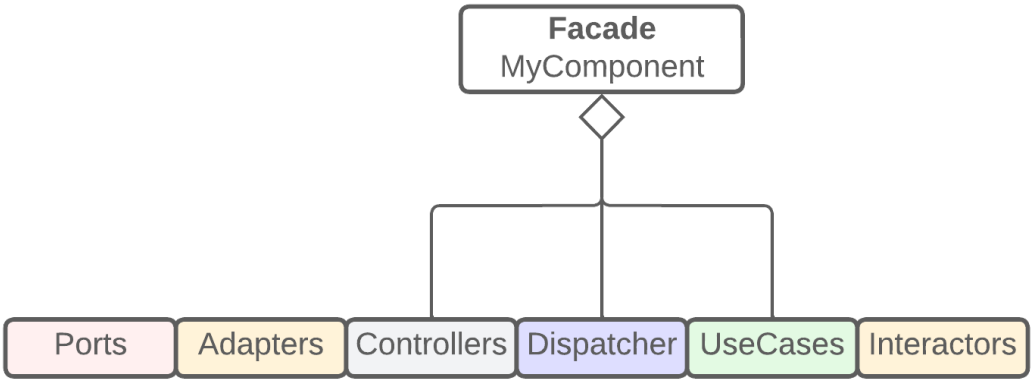


Abbildung 24: Vereinfachte Darstellung der Architektur als Komponente

Laut der Darstellung 24 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:

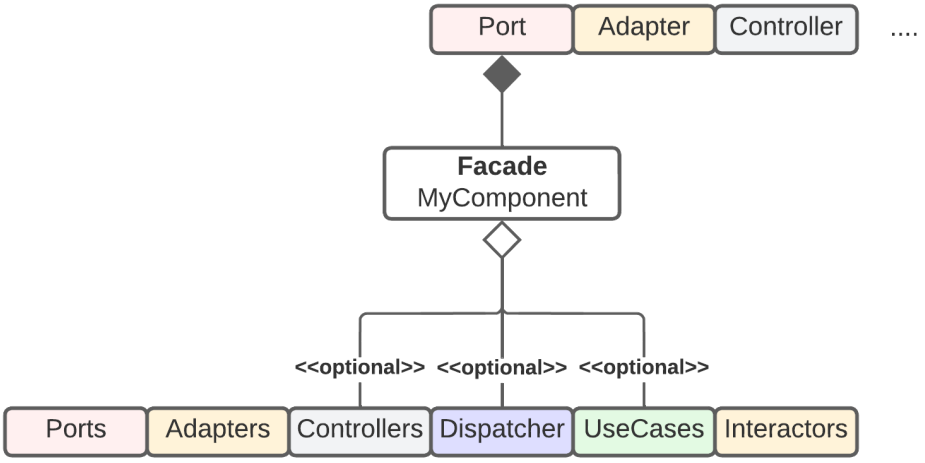


Abbildung 25: Vereinfachte Darstellung einer Standalone Anwendung mit der Komponente

Der Datenfluss in der Anwendung:

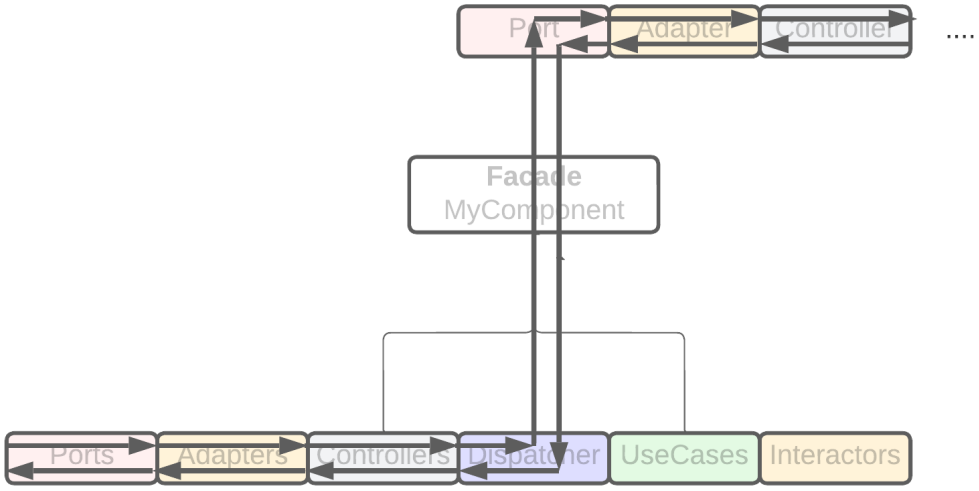


Abbildung 26: Vereinfachte Darstellung des Datenflusses in einer Anwendung mit Komponente

3.3.2 Framework und Bibliothek

mehr dazu kann man bei nachlesen: **siehe kommentar in Latex**

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

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.

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

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 aufruft. (Inversion Of Control)

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

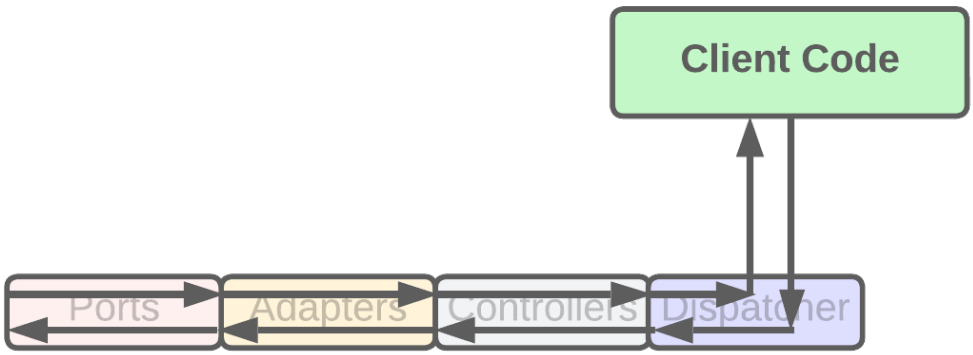


Abbildung 27: Vereinfachte Darstellung des Datenflusses bei einer Bibliothek

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

Abbildung 28: 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

4 Aufgabenstellung

Zu Begin 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, bekamm ich die Aufgabe einen OCPP 1.6. Server zu implementieren.

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

Das sind:

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

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

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

4.1 Anforderungen an den Standalone Server

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

Die Anforderungen an den Server sind:

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

4.2 Anforderungen an das Testframework

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

Die Anforderungen an das Testframework sind:

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

4.3 Anforderungen an ERK Automatisierungstool

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

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

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

Die Anforderungen an das ERK Automatisierungstool sind:

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

5 Lösung der Aufgabe

hier wird die Lösung der Aufgabe beschrieben mit Bildern usw.

6 Gewünschtes Interfaces

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

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

Jeder Ablauf eines Systemtests kann mit folgender Schema beschrieben werden:

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

7 Implementierung

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

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

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

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

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

7.1 Achitecture des Frameworks

Es wurde entschieden das Framework in 7 Abstraktionsschichten aufzuteilen.

7.1.1 Ports

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

In dem Framework wird nur ein Port gebraucht - WebSocket Server

7.1.2 Adapters

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

In dem Framework wird nur einen Adapter gebraucht - OCPP16 Adapter

7.1.3 Controllers

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

In dem Framework werden mehrere Controllers gebraucht:

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

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

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

7.1.4 Dispatcher

Der Dispatcher informiert alle abonnierten UseCases über das eingetretene Event.

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

7.1.5 UseCases

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

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

7.1.6 Interactors

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

Benutzt mittels "Dependency Injection"die Controller.

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

7.1.7 Domain

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

7.2 Zugriff auf das Testframework

IRGENDWAS EINLEITENDES

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

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

7.3 Testbeispiel

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

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

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

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

        // 3. start server
        await testOcppInstanz.start();
        return;
    });
    it("test 1", async () => {
        // 5. wait until test is done
        const res = await testOcppInstanz.waitOfEvent("BootNotification");

        // 4. and 6. set the requirement test is valid, validate the result
        expect(res.name).equal("BootNotification");
    });
    it("test 2", async () => {
        // 5. wait until test is done
        const res = await testOcppInstanz.waitOfNextEvent();

        // 4. and 6. set the requirement test is valid, validate the result
        expect(res.name).equal("Heartbeat");
    });
    after(async () => {
        // 7. stop server
        testOcppInstanz.stop();
        return;
    });
});
```

8 Übersichtsdiagramm

9 Conclusion

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

Bibliography

[Sag93] Carl Sagan. *Brocas brain: reflections on the romance of science*. Presidio Press, 1993.

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

[Fowa] Martin Fowler. URL: https://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf.

[Fowb] Martin Fowler. *Is High Quality Software Worth the Cost?* URL: https://martinfowler.com/articles/is-quality-worth-cost.html.

[NTN] Department of Marine Technology NTNU. *IMT Software Wiki - LaTeX*. URL: https://www.ntnu.no/wiki/display/imtsoftware/LaTeX (besucht am 15. Sep. 2020).

[Rod] Vladislav Rodin. URL: https://tech-de.netlify.app/articles/de521476/index.html.