HTWK

Hochschule für Technik,
Wirtschaft und Kultur Leipzig

Fakultät Ingenieurwissenschaften

6010 - Praxisprojekt

# Implementierung einer OCPP-Backend-Testumgebung für automatische Intergrationstests einer DC High Power Ladestationen der Plattform Sicharge D

*Author* Ivan Agibalov
*Betreuer* Prof. Dr.-Ing. Andreas Pretschner

26. Juli 2022

# Inhaltsverzeichnis

# Abbildungsverzeichnis



# Tabellenverzeichnis

# 1 Motivation

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

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

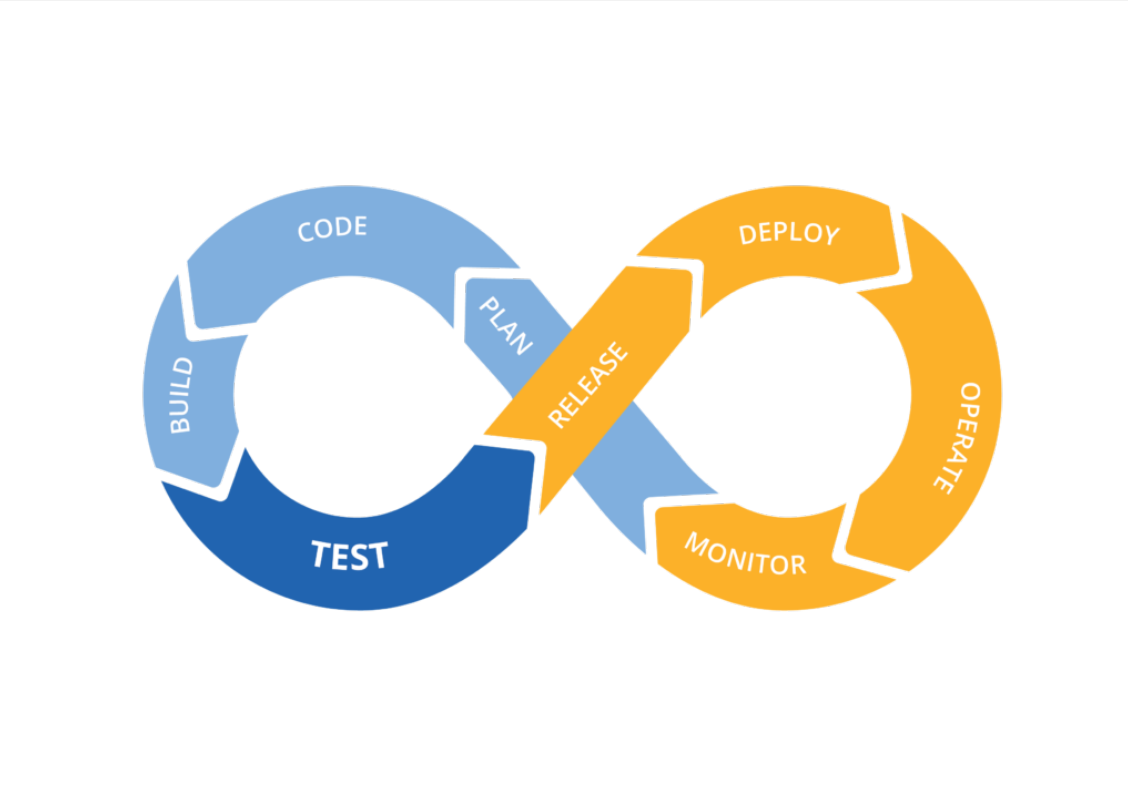



Abbildung 1: CI/CD Pipeline

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

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

Im groben kann man die Entwicklung in 2 Teilen teilen: Bevor die neue Version der Software freigegeben wird und nach der Freigabe der neuen Version. Die Phasen nach der Freigabe der neuen Version lassen sich fast vollständig automatisieren und verbrauchen dementsprechend nicht so viel Ressourcen ab einem gewissen Moment. Das großte Anteil an Ressourcen wird in die ersten 4 Phasen (Plan, Code, Build, Test) verbraucht, denn diese Aufgaben lassen sich sehr schlecht bis zu gar nicht automatisieren. Und wenn man am Anfang des Projektes schlechte Entscheidungen trifft, die das automatisierte Testen erschweren oder durch die schlechte Struktur des Programms das Hinzufügen der neuen Funktionalität deutlich schwieriger macht, wird man deutlich mehr Ressourcen gebrauchen, um das gleiche Ziel zu ereichen als wenn man das nicht gemacht hätte.

In dieser Arbeit wird es darum gehen, welche Entscheidungen am Beispiel eines OCPP Servers man bereits in der Phase “Plan” und “Code” treffen kann, um die Gesamtqualität der Software zu erhöhen, die die Anzahl an benötigten menschlichen Ressourcen reduziert.

# 2 Introduction

SomeIntroducation

## 2.1 OOP Design Patterns

some Introduction

### 2.1.1 Observer

Das OOP Design Pattern **Observer** ermöglicht dynamische Verbindungen zwischen den einzelnen Objecten 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 definiert werden.

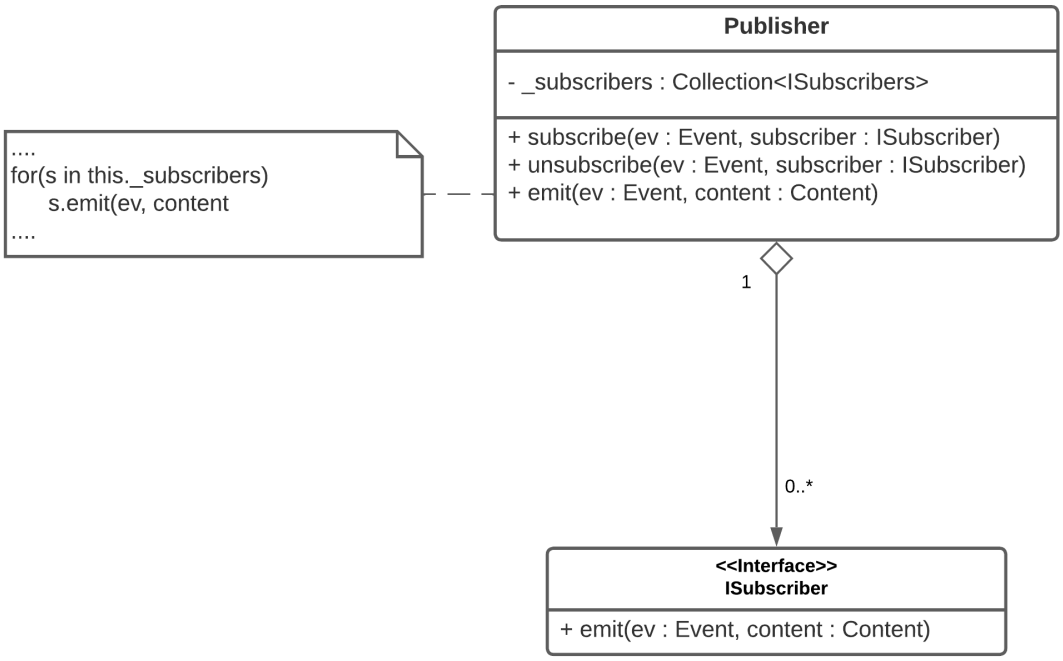



Abbildung 2: Klassendiagrammm Observer

Source: Eigene Quelle

### 2.1.2 Proxy

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

### 2.1.3 TemplateMethod

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

### 2.1.4 Builder

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

### 2.1.5 Facade

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

## 2.2 CleanArchitecture

some info about clean architecture

## 2.3 Software Testing

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

Um das Risiko auf defekte zu minimieren, können 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 hier kurz beschrieben werden.

### 2.3.1 Testing Pyramide

Some text to testing pyramid

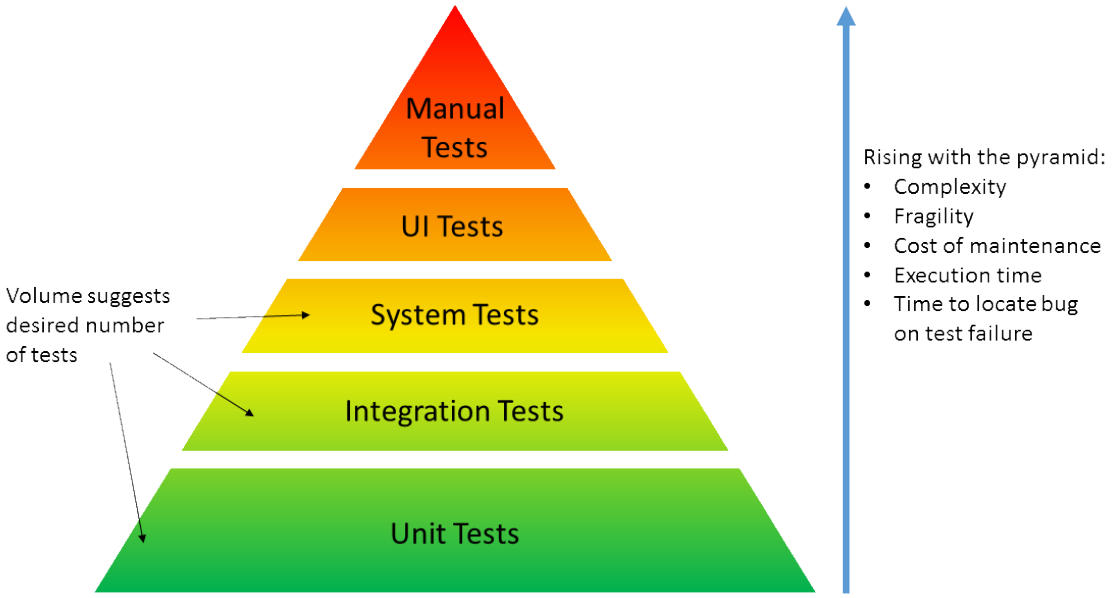



Abbildung 3: Caption written below figure.

Source: https://www.cqse.eu/de/news/blog/junit3-migration/

### 2.3.2 Unit Tests

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

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

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

### 2.3.3 Integration Tests

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

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

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

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

### 2.3.4 SystemTests

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

#### 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 laufende Runtime, um den User zu simulieren, ein UI Test wird nur mittels der Ausgabe an der Oberfläsche validiert.

#### 2.3.6 Manual Tests

some Info about manual tests

### 2.4 SOLID

some info about solid

#### 2.4.1 single-responsibility principle

SRP

#### 2.4.2 open–closed principle

OCP

#### 2.4.3 Liskov substitution principle

LSP

#### 2.4.4 interface segregation principle

ISP

#### 2.4.5 dependency inversion principle

DIP

### 2.5 GRASP

grasp

### 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

# 3 Software Architektur

## 3.1 Was ist eine Software Architektur

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

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

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

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

In dem 2. Teil des Kapitels, wird die Architecture des OCPP Backend Serves beschrieben

- in welche Teile das Programm aufgeteilt wurde
- Testbarkeit und Erweitbarkeit einzelner Teile des Programms
- wie die einzelnen Teile miteinander kommunizieren

## 3.2 Ziele der Software Architektur

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

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

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

Beispiele für solche Entscheidungen wären:

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

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

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

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

## 3.3 Technische Schulden

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

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

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

- Unittests
- Dokumentieren
- Code Review

Beispiele für Teschnise Schulden wären:

- Alte Funktionalitäten funktionieren nach der Änderung nicht mehr
- Aufdeckung eines Bugs erst nach einer gewissen Zeit in Produktionsversion der Software
- Das 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 Dokumentation geschrieben werden, mit weniger Dokumentation findet man schneller die gesuchten Informationen.

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

Daher es ist vom Vorteil bevor man mit der Umsetzung des Softwaresystems anfängt, die oben gennanten Aufgaben zu lösen. Die erforderliche Zeit um die Änderungen von oben genannten Eigenschaften nimmt mit der Lebenszeit der Software zu.

Somit lassen sich die vorhandenen Ressourcen effizienter eingesetzt werden.

## 3.4 Qualität und Kosten der Software

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

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

- Bugs können schneller lokalisiert und beseitigt werden
- Neue Funktionalitäten können mit weniger Aufwand umgesetzt werden
- Die Änderungen der Funktionalitäten können schneller umgesetzt werden
- Die Wahrscheinlichkeit alte Funktionalitäten zu zerstören weringert sich
- Die Einarbeitungszeit von neuen Teammitgliedern verkürzt sich

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

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

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

- Wann soll spätestens die erste Version da sein
- Wie viele Ressourcen man zur Verfügung hat
- Wie wahrescheinlich sind die Änderungen und Erweiterungen der Software
- Wie kritisch verschieden Probleme und Ausfälle der Software sind

Auf dem unterer Darstellung sieht man, dass auf langere Distanz eine gute Softwarearchitektur deutlich mehr Funktionalitäten besitzt als eine Software mit schlechter Architektur. Jedoch es gibt einen Zeitinterval, in dem die schlectere Software besser da steht.

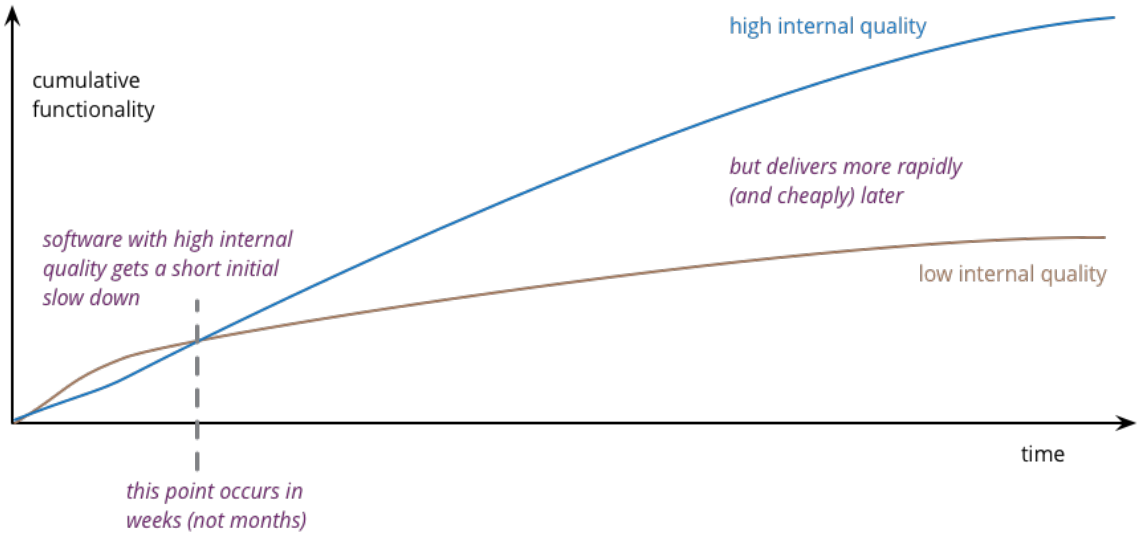



Abbildung 4: Vergleich einer guten und einer schlecten Softwarearchitektur

Source: https://martinfowler.com/articles/is-quality-worth-cost.html

Diese Eigenschaft muss man immer beim Projektbegin beachten, denn es wäre sehr Zeitaufwändig für ein Studiumprojekt, für das man evtl. nur 1 Woche Zeit komplexe Architektur zu implementieren, die ohne jeglichen Funktionalitäten mehrere Wochen gebrauchen wird.

Wenn man aber ein Projekt hat, das regelmäßig weiterentwickeln wird, dass man gleich eine gute Architektur umzusetzt.

## 3.5 Wichtigkeit der Testbarkeit der Software

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

## 3.6 Technische Umsetzung der Software Architektur

### 3.6.1 Auswahl der Datenbank

### 3.6.2 Dependency Rule

### 3.6.3 Unterschied zu Layered Architektur

### 3.6.4 Implementierung der Testbarkeit

- Humble Objects

### 3.6.5 Dependency Injection

In dem unteren Diagramm sieht man ein Beispiel von einer Anwendug, die die von der konsole ankommenden Zahlen quadriert und das Ergebnis zurückgibt.

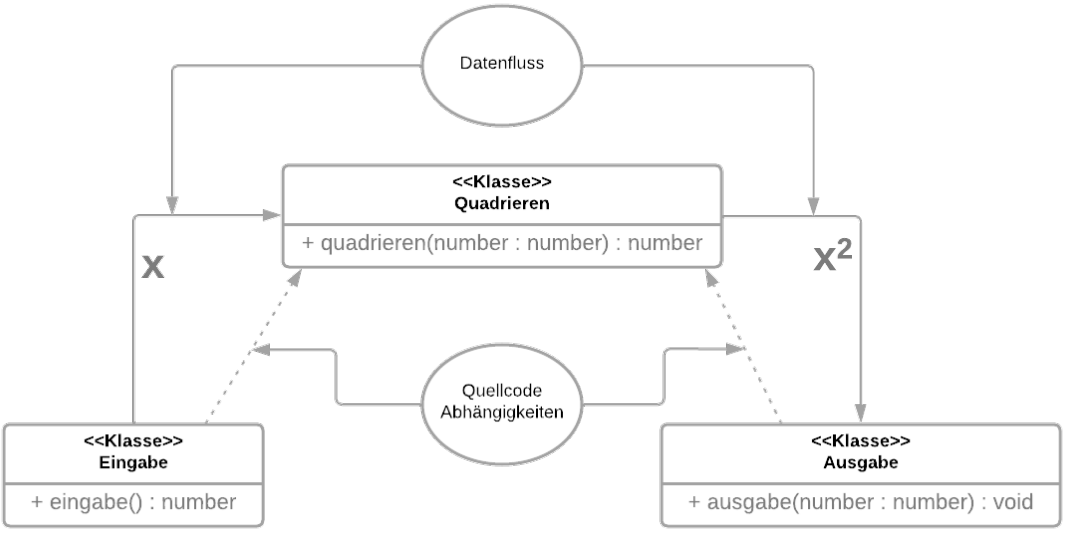



Abbildung 5: Datenfluss und Quellcode Abhängigkeiten

Source: Eigene Quelle

Die Funktion "Quadrierenïst 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 wurde dann so aussehen.

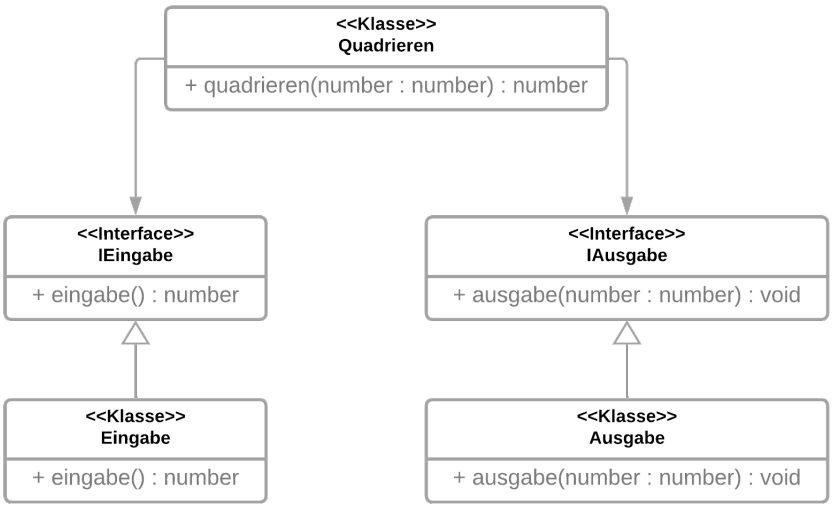



Abbildung 6: Entkopplung der Abhängigkeiten

Source: Eigene Quelle

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

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

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

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

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

# 4 Aufgabenstellung

Zu Begin der Praxisphase war die Entwicklung und Testen an vielen Stellen an eine externe Schnittstelle gebunden. Um diese Abhängigkeit zu reduzieren und die Möglichkeit zu bekommen eigene Funktionalitäten hinzuzufügen, bekammm 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 gebraucht 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 Conclusion

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

# Bibliography


[GGS82] U. Ghia, K. N. Ghia und C. T. Shin. „High-Re Solutions for Incompressible Flow Using the Navier-Stokes Equations and a Multigrid Method“. In: *Journal of Computational Physics* 48 (1982), S. 387–411.

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

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

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

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

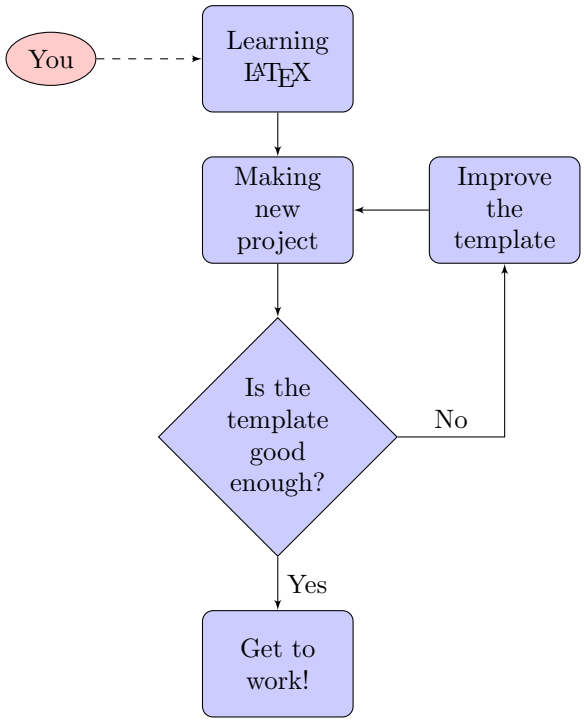
[NTN] Department of Marine Technology NTNU. *IMT Software Wiki - LaTeX*. URL: https://www.ntnu.no/wiki/display/imtsoftware/LaTeX (besucht am 15. Sep. 2020).

# Appendix

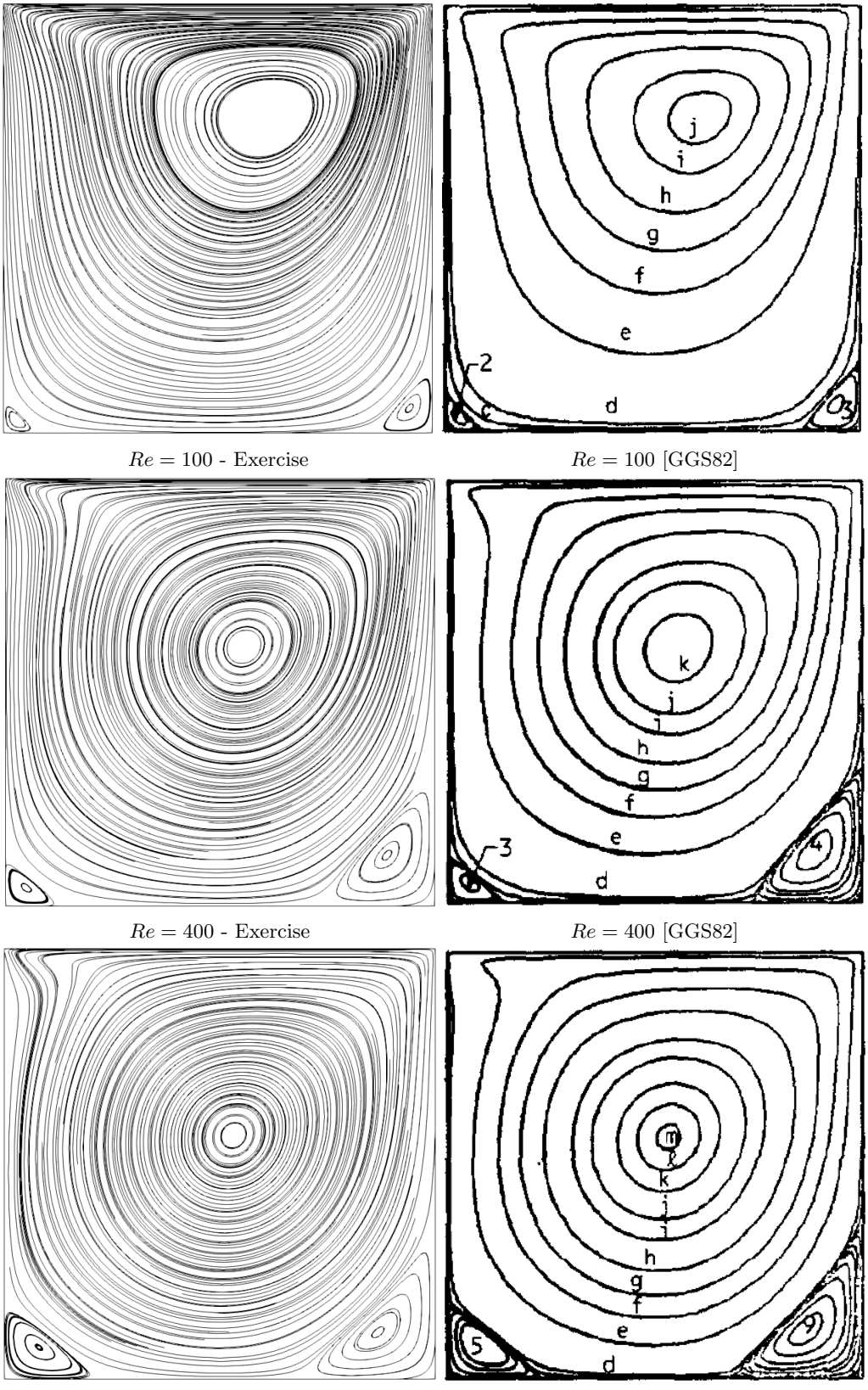
## A Hello World Example

⋮

## B Flow Chart Example



## C Sub-figures Example




$Re = 100$ - Exercise $Re = 100$ [GGS82]

$Re = 400$ - Exercise $Re = 400$ [GGS82]

$Re = 1000$ - Exercise $Re = 1000$ [GGS82]

Abbildung 7: Streamlines for the problem of a lid-driven cavity.