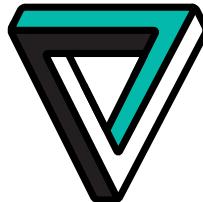


# Fachhochschule Aachen

- University of Applied Sciences -

## Bachelorarbeit

Integration einer Blockchain in Geschäftsprozesse der  
Fachhochschule Aachen



---

Kai Dinghofer

Matrikelnummer: XXXXXXXX

14. Juni 2018

Erstprüfer: Prof. Dr. rer. nat. Heinrich Faßbender

Zweitprüfer: Torben Hensgens, M. Eng.

# **Erklärung**

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe.

Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

.....  
Aachen, den 14. Juni 2018

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Ziele . . . . .	1
1.3 Konventionen . . . . .	2
1.3.1 Formatierung . . . . .	2
1.3.2 Quellen . . . . .	2
1.3.3 Weiteres . . . . .	3
1.4 Überblick . . . . .	3
<b>2 Verwendete Technologien</b>	<b>4</b>
2.1 Die Blockchain: Ethereum . . . . .	4
2.1.1 Hintergrund . . . . .	5
2.1.2 Modell: Globaler Transaktions-Automat . . . . .	6
2.1.3 Proof of Work: Mining . . . . .	8
2.1.4 Ethereum Spezifisches . . . . .	14
2.1.4.1 Währung . . . . .	14
2.1.4.2 Accounts . . . . .	15
2.1.4.3 Transaktionen und Nachrichten . . . . .	16
2.1.4.4 Ethereum Virtual Machine . . . . .	19
2.1.4.5 Weitere Anmerkungen . . . . .	23
2.1.4.6 Zusammenfassung . . . . .	24
2.2 Camunda BPM . . . . .	25
2.2.1 Modeler . . . . .	26
2.2.2 Platform . . . . .	27
2.3 Vagrant . . . . .	28
2.4 Spring Boot . . . . .	29
2.5 Docker . . . . .	29

## Inhaltsverzeichnis

---

2.6	Hardware . . . . .	30
2.6.1	Raspberry Pi . . . . .	30
2.6.2	RFID und NFC . . . . .	31
<b>3</b>	<b>Analyse und Konzeption . . . . .</b>	<b>33</b>
3.1	Analyse: Einsatzgebiet FH Aachen . . . . .	33
3.1.1	WLAN Netz . . . . .	33
3.1.2	Prüfungsverwaltungssystem . . . . .	34
3.1.2.1	Sicherheitsanalyse . . . . .	34
3.1.2.2	Datenhaltung . . . . .	35
3.1.3	Studierendenausweis . . . . .	36
3.1.4	Geschäftsprozess „Klausur durchführen“ . . . . .	38
3.1.4.1	Prozessüberblick in drei Zonen . . . . .	40
3.1.4.2	Nachteile und Verbesserungsvorschläge . . . . .	41
3.2	Zielsetzung und Anforderungen . . . . .	42
3.2.1	Notwendige Anforderungen . . . . .	43
3.2.1.1	Sicherheit . . . . .	43
3.2.1.2	Mengengerüst, Skalierbarkeit und Effizienz . . . . .	44
3.2.1.3	Erweiterbarkeit . . . . .	45
3.2.1.4	Verwendung offener Standards . . . . .	45
3.2.2	Empfohlene/Optionale Anforderungen . . . . .	46
3.2.2.1	Verwendung bestehender Standards . . . . .	46
3.2.2.2	Einfachheit der Bedienung . . . . .	46
3.2.2.3	Dokumentation . . . . .	47
3.2.2.4	Funktional: Interne Währung . . . . .	47
3.3	Konzeption: Wahl der Technologien . . . . .	47
3.3.1	Blockchains im Vergleich . . . . .	47
3.3.1.1	Auswahlkriterien . . . . .	47
3.3.1.2	Vergleich . . . . .	49
3.3.1.3	Ethereum . . . . .	53
3.3.2	Camunda . . . . .	54
3.3.3	Vagrant . . . . .	55
3.3.3.1	Entwicklungsbetriebssystem . . . . .	55
3.3.4	Spring Boot . . . . .	56

## Inhaltsverzeichnis

---

3.3.5	Docker . . . . .	56
3.3.6	Hardware . . . . .	57
3.3.6.1	Raspberry Pi . . . . .	58
3.3.6.2	NFC . . . . .	58
<b>4</b>	<b>Die FHACChain</b>	<b>63</b>
4.1	Kernkomponenten . . . . .	64
4.1.1	Systemumgebung . . . . .	66
4.1.2	Ethereum Test-Netzwerk via Docker . . . . .	69
4.1.3	Backends . . . . .	70
4.1.3.1	Smart Contracts . . . . .	70
4.1.3.2	REST Service . . . . .	77
4.1.4	Frontends . . . . .	78
4.1.4.1	Raspberry Pi NFC Terminal . . . . .	79
4.1.4.2	Administrator Web UI . . . . .	84
4.2	Integration . . . . .	84
4.2.1	„Hallo FHACChain“-Prozess: Erste Anbindung . . . . .	85
4.2.2	Web3-Connector . . . . .	87
4.2.2.1	Verwendungsmöglichkeiten . . . . .	88
4.2.2.2	Optimierter „Hallo FHACChain“-Prozess . . . . .	90
4.3	Optimierte Geschäftsprozesse . . . . .	91
4.3.1	Klausur durchführen . . . . .	92
4.3.2	Bachelorarbeit schreiben . . . . .	95
4.3.2.1	Vergleich mit Prüfungsordnung . . . . .	97
4.3.2.2	Vergleich mit Datenbank . . . . .	98
4.3.2.3	Zusammenfassung . . . . .	99
<b>5</b>	<b>Fazit</b>	<b>100</b>
5.1	Pro und Contra . . . . .	100
5.2	Ergebnis . . . . .	101
5.3	Ausblick . . . . .	101
5.3.1	Pilotprojekt: FH Aachen Testnetzwerk . . . . .	102
5.3.2	Prüfungsamt 3.0 . . . . .	103

*Inhaltsverzeichnis*

---

5.3.3 Gesch鋐tsprozess: Klausurbenotung durchf黨ren . . . . .	103
5.3.3.1 Rechtliche Anforderungen . . . . .	103
5.3.3.2 Soll-Gesch鋐tsprozess . . . . .	104

<b>A Anhang</b>	<b>118</b>
-----------------	------------

# Abkürzungsverzeichnis

<b>ETH</b>	Ether, die intrinsische Währung von Ethereum
<b>DAO</b>	Decentralized Autonomous Organization
<b>EOA</b>	Externally Owned Account
<b>CA</b>	Contract account (sofern nicht lokal anders definiert)
<b>ROM</b>	Read-Only Memory, dt. Nur-Lese-Speicher
<b>RAM</b>	Random-Access Memory, dt. Direktzugriffspeicher
<b>BPM</b>	Business Process Management
<b>OMG</b>	Object Management Group
<b>BPMN</b>	Business Process Model and Notation
<b>DMN</b>	Decision Model and Notation
<b>XML</b>	Extensible Markup Language
<b>UI</b>	User Interface
<b>WYSIWYG</b>	What You See Is What You Get
<b>REST</b>	Representational State Transfer
<b>API</b>	Application Programming Interface
<b>JVM</b>	Java Virtual Machine
<b>MB</b>	Megabyte
<b>GNU</b>	GNU's Not Unix ( <i>rekursives Akronym</i> )
<b>VM</b>	Virtuelle Maschine
<b>SoC</b>	System on a Chip
<b>GPIO</b>	General Purpose Input/Output
<b>CPU</b>	Central Processing Unit
<b>USB</b>	Universal Serial Bus

## *Inhaltsverzeichnis*

---

<b>HDMI</b>	High-Definition Multimedia Interface
<b>USB-OTG</b>	USB On-The-Go
<b>RFID</b>	Radio-Frequency Identification
<b>UID</b>	Unique Identifier Number
<b>ID</b>	Identifier Number
<b>NFC</b>	Near Field Communication
<b>OOP</b>	Objektorientierte Programmierung
<b>JEE</b>	Java Enterprise Edition
<b>POJO</b>	Plain Old Java Objects
<b>REST</b>	Representational State Transfer
<b>WLAN</b>	Wireless Local Area Network
<b>IT</b>	Informationstechnik
<b>ECTS</b>	European Credit Transfer System
<b>IC</b>	Integrated Circuit
<b>ISO</b>	International Organization for Standardization
<b>IEC</b>	International Electrotechnical Commission
<b>DoS</b>	Denial of Service
<b>SWE</b>	Software Engineering
<b>SQL</b>	Structured Query Language
<b>HATEOAS</b>	Hypermedia as the Engine of Application State
<b>LTS</b>	Long-Term Support
<b>EEPROM</b>	Electrically Erasable Programmable Read-Only Memory
<b>SSH</b>	Secure Shell
<b>WAR</b>	Web Application Archive
<b>UML</b>	Unified Modeling Language
<b>JSON</b>	JavaScript Object Notation
<b>RPC</b>	Remote Procedure Call
<b>LED</b>	Light-Emitting Diode
<b>SOAP</b>	Simple Object Access Protocol (früher)

*Inhaltsverzeichnis*

---

# Abbildungsverzeichnis

2.1	Darstellung der Blockchain als (unendlicher) Automat mit $n \in \mathbb{N}$ Zuständen.	8
2.2	Blöcke mit Transaktionen und Metadaten.	9
2.3	Baumstruktur mit mehreren Forks/Blockchains.	13
2.4	Die zwei Arten von Accounts.	16
2.5	Alle drei Kommunikationsmöglichkeiten.	17
2.6	Der Camunda Stack.	26
2.7	Raspberry $\pi_0$ v1.3	31
3.1	eduroam-Statistik. Stand: 04. Juni 2018.	34
3.2	Bildschirmausschnitt des Notenspiegels im <i>QIS HIS</i> .	36
3.3	Scan der FH Karte.	37
3.4	Der Ist-Geschäftsprozess „Klausur durchführen“.	39
4.1	Der <i>FHACChain</i> Technologie-Stack.	65
4.2	Die <i>FHACChain</i> VM.	68
4.3	Die Smart Contracts der <i>FHACChain</i> dargestellt als UML-Diagramm.	71
4.4	Das <i>FHACChain Terminal</i> und zwei NFC Tags.	79
4.5	Use Case Diagramm des <i>FHACChain Terminals</i> .	80
4.6	Dump eines Tags mit dem <i>FHACard Tool</i> .	81
4.7	Exemplarischer „Hallo FHACChain“-Geschäftsprozess.	85
4.8	Verwendung des <i>Web3-Connectors</i> .	88
4.9	Optimierter „Hallo FHACChain“-Geschäftsprozess mittels <i>Web3-Connector</i> .	90
4.10	Der verbesserte Geschäftsprozess „Klausur durchführen“.	93
4.11	Ausschnitt: „Bachelorarbeit schreiben“-Prozess ohne Contract-Anbindung.	96
4.12	Ausschnitt: „Bachelorarbeit schreiben“-Prozess mit neuer <i>ExaminationService</i> -Anbindung.	96
5.1	Der Vorschlag zum Soll-Geschäftsprozess „Klausurbenotung durchführen“.	105

# 1 Einleitung

## 1.1 Motivation

Der abstrakte Begriff der Blockchain gewinnt sowohl in der Industrie<sup>1</sup> als auch bei Privatanwendern<sup>2</sup> rapide an Bedeutung.

Daher ist es sehr wichtig, diese Technologie-Errungenschaft nicht unbeachtet zu lassen. Um eine neue Technologie praxisnah zu evaluieren, ist es äußerst sinnvoll, diese in ein bereits existierendes System zu integrieren. Auf diese Weise lässt sich ein potenzieller Mehrwert durch den Einsatz der Blockchain Technologie mit wissenschaftlichen Mitteln nicht nur abschätzen, sondern sogar nachweisen. Im Kontext des Studiums gibt es für die Integration kein besser geeignetes System als die Geschäftsprozesse der Fachhochschule Aachen selbst.

Inwiefern ist es möglich, eine Blockchain und die Geschäftsprozesse an der FH Aachen zu vereinen? Welche Vor- und Nachteile bietet der Einsatz der Technologie?

Blockchain bedeutet Einführung neuester Konzepte – vollständige Dezentralisierung. In besonderer Weise hat mich daher die Frage herausgefordert, welcher Grad der Dezentralisierung bei einer überwiegend zentralen Einrichtung wie der FH Aachen de facto möglich ist. Diese spannende Frage lässt sich nur beantworten, wenn unter möglichst realen Bedingungen gearbeitet wird. Es folgt die Zielsetzung, mit deren Umsetzung sich die Frage am Ende dieser Bachelorarbeit beantworten lässt.

## 1.2 Ziele

Die Ziele der Bachelorarbeit lassen sich wie folgt zusammenfassen:

---

<sup>1</sup>siehe auch [Cas17](#), für Anstieg der Blockchain-Jobangebote um 631% seit November 2015.

<sup>2</sup>siehe auch [Goo18](#), für Suchanfragen-Statistik via [Google Trends](#) von Januar 2012 bis Januar 2018.

1. Eine Blockchain gezielt in ausgewählte Geschäftsprozesse der FH Aachen integrieren, um mithilfe von Digitalisierung und Automatisierung einen Mehrwert durch gezielte Verbesserung zu erhalten.
2. Die notwendige Software-Infrastruktur zur Nutzung einer Blockchain bereitstellen, um das Fundament für ein mögliches Community-Projekt zu schaffen, das von allen Interessenten erweitert werden kann.

Der Umsetzung dieser Ziele wird möglichst nachvollziehbar und reproduzierbar nachgegangen. Das Gesamtergebnis des praktischen Teils dieser Ausarbeitung – die *FHACChain* – ist daher dank dem Einsatz moderner Technologien auch auf dem Rechner des Lesers ausführbar: Eine Grundlage, die sowohl von Studierenden, als auch von anderen Mitgliedern der Fachhochschule Aachen getestet werden kann, vereinfacht schließlich nachhaltig den Einstieg in die komplexe Thematik der Blockchain.

## 1.3 Konventionen

### 1.3.1 Formatierung

Fachbegriffe werden bei der Einführung erstmalig *kursiv* geschrieben. Genauso wird Kursivschrift für das die spezielle *Hevorheben* eines Sachverhalts verwendet. Produktnamen und Standards werden *abgeschrägt* formatiert. Die Neigung ist hierbei etwas geringer als bei der Kursivschrift.

Im Fließtext werden Wörter nur in seltenen Fällen **fett** geschrieben. Dies sind Ausnahmen, da es sonst die Wichtigkeit abschwächt, den gegenteiligen Effekt erzielt und nicht förderlich für den Lesefluss ist.

Codebeispiele verwenden immer eine Konstantenschrift.

Allgemein wurde sich bei der Farbgebung an das *Corporate Design* der FH Aachen orientiert.

### 1.3.2 Quellen

Quellenangaben sind direkt im Text vermerkt. Dazu wird bei indirekten Angaben die Abkürzung „vgl.“ verwendet. Bei Angaben zu weiterführenden Informationen wird hingegen – sofern möglich – eine Fußnote und der Verweis auf die ergänzende Quelle mit

„siehe auch“ angegeben. Bei Abbildungen ist die Quellenangabe (falls nicht selbst erstellt) immer rechts unter dem Bild zu finden.

### 1.3.3 Weiteres

Zusätzlich werden, immer da wo es sich anbietet, anschauliche Beispiele und Exkurse in Form von farbcodierten Boxen dargelegt, die für das Verständnis unterstützend sind. Rein optionale Beispiele sind mit einem vorangestellten Stern (\*) gekennzeichnet.

Falls „zum Schreibzeitpunkt“, „derzeitig“ oder ein gleichbedeutendes Wort zum Einsatz kommt, so ist es mit „zum Schreibzeitpunkt dieser Bachelorarbeit vom 6. April 2018 bis zum 14. Juni 2018“ gleichzusetzen.

## 1.4 Überblick

Zu Beginn werden im Kapitel 2 die Technologien erklärt, die im Praxisprojekt und in dieser Bachelorarbeit eingesetzt werden. Begriffe, die stark aufeinander aufbauen, werden ausführlicher beleuchtet. Leser, die sich bereits tiefgehender mit *Ethereum* auseinandergesetzt haben und sich mit der Software *Camunda BPM* auskennen, können dieses Kapitel überspringen.

Im nächsten Kapitel 3 wird zunächst die Zielrichtung festgelegt. Weiterhin wird mit einer Analyse des Einsatzgebietes „Fachhochschule Aachen“ ermittelt, wie die Ausgangssituation ist. Daraus lassen sich die Anforderungen an das zu entwickelnde System ableiten, welche dargelegt werden. Gegen Ende des Kapitels werden die in Kapitel 2 erklärten verwendeten Technologien evaluiert und es wird erörtert, aus welchen Gründen sich für die jeweilige Technologie entschieden wurde.

Kapitel 4 präsentiert das Ergebnis der Analyse und der Konzeption inklusive aller optimierten ausführbaren Geschäftsprozesse: Die *FHACChain* ist der Kern dieser Bachelorarbeit und das Ergebnis des Praxisprojekts.

Im finalen Kapitel 5 werden Pro und Contra des Projekts gegenübergestellt und ein Fazit gezogen. Ferner werden im Ausblick weitere Chancen, Konzepte und Ideen betrachtet, die nicht vollständig in der vorliegenden Arbeit umgesetzt werden konnten.

## 2 Verwendete Technologien

Die eingesetzten *Software*-Technologien werden zunächst – nach ihrer Relevanz absteigend sortiert – erklärt. Im direkten Anschluss werden die verwendeten *Hardware*-Technologien gesondert dargestellt.

### 2.1 Die Blockchain: Ethereum

Im Rahmen dieser Arbeit wird die Blockchain Lösung *Ethereum* verwendet, speziell das besondere Feature der *Smart Contracts*.

Um zu verstehen, was ein Smart Contract ist, ist es jedoch zunächst notwendig, die allgemeinen Paradigmen einer Blockchain zu verstehen. Wenn dabei unbestimmt von „einer Blockchain“ geschrieben wird, bezieht sich die Verallgemeinerung, solange sie nicht im Verlauf konkretisiert wird, auf die *Bitcoin* und *Ethereum* Blockchain. Aufgrund der extremen Vielfalt werden andere Implementierungen in diesem Kapitel nicht in Betracht gezogen.

Angefangen wird mit der allgemeinen Definition einer Blockchain.

#### Definition 2.1: Blockchain

Eine Blockchain ist ein digitales Buchführungssystem, dessen immer steigende Anzahl von Einträgen (⇒ Blöcken, die Transaktionen enthalten) mithilfe kryptografischer Verfahren sicher verkettet ist [vgl. Inv18] [vgl. Eco15].

Mit den Erkenntnissen aus den nachfolgenden Sktionen wird dieses Verständnis des Begriffs spezifiziert. Alle Definitionen behalten stets ihre Gültigkeit bei.

### 2.1.1 Hintergrund

Bitcoin ist die erste Umsetzung einer Blockchain. Die Erfindung wurde von 2008 bis 2009 von Satoshi Nakamoto (der reale Name des Autors ist bis heute unbekannt) in Form eines Whitepapers veröffentlicht und entwickelt [vgl. [Wal11](#)].

Nakamoto ist es gelungen, Technologien aus verschiedenen Bereichen in einer ausgeklügelten Art und Weise zu kombinieren um eine Peer-to-Peer Onlinewährung zu erschaffen. Anstatt auf eine zentrale Einheit angewiesen zu sein, werden Transaktionen dezentral auf einer Menge von *Knoten* - also den Teilnehmern des Netzwerks - abgewickelt [vgl. [Nak08](#), S. 1].

Doch warum sollte eine Währung und die dahinter zugrunde liegende Buchhaltung überhaupt dezentral sein? Für einige Menschen ist dieser Gedanke abwegig [vgl. [Sti18](#)]. Schließlich funktioniert das Bankensystem in den meisten Fällen zuverlässig.

Die Motivation hinter Bitcoin ist Folgende: Wenn ein Transfer von Geld stattfindet, so muss gewährleistet sein, dass das Finanzinstitut diesen korrekt und fehlerfrei abwickelt. Die Kunden sind dadurch auf eine zentrale dritte Partei (beispielsweise eine Bank) angewiesen, damit Geld transferiert werden kann. Bitcoin versucht dieses Problem zu lösen, indem der Vermittler eliminiert wird: Die Zahlungsabwicklung basiert auf kryptografischen Beweisen und nicht auf dem Vertrauen gegenüber einer dritten Partei.

Dabei wird das Vertrauen jedoch nicht *abgeschafft*, sondern auf die Mehrheit der Knoten des Netzwerks *ausgelagert*. Denn sofern die Anzahl der ehrlichen Teilnehmer überwiegt, so gilt auch das Netzwerk als sicher. Die logische Schlussfolgerung ist, dass ein Teilnehmer als Einzelperson betrachtet nicht als vertrauenswürdig eingestuft werden darf [vgl. [Nak08](#)].

Die Hauptmerkmale Bitcoins wurden im November 2013 mit der Veröffentlichung des *Ethereum Whitepapers* vom Mitbegründer des Bitcoin Magazins und Entwickler Vitalik Buterin aufgegriffen und erweitert [vgl. [Eth18a](#)]:

Der Grundgedanke ist es, auf Basis der Blockchain Technologie eine dezentrale Plattform für Applikationen mit einer integrierten Turing-vollständigen Programmiersprache zur Verfügung zu stellen. Dazu gibt es eine eigene von innen herkommende Währung für Transaktionen jeglicher Art namens *Ether*. Die in der Blockchain abgelegten Programme werden bei Ethereum *Smart Contracts* (dt. „intelligente Verträge“) genannt und können von den Benutzern hinzugefügt werden. Entgegen der Namensgebung beschränken

sich die Programmiermöglichkeiten jedoch nicht nur auf Geschäftslogik vertraglicher Natur [vgl. [But17a](#)]. Aufgrund des generalisierten Aufbaus ist das Ökosystem von Ethereum sehr gut erweiterbar [vgl. [Woo18](#)]. Populäre Anwendungsbeispiele für Smart Contracts sind Token-Systeme (z.B. untergeordnete Währungen), Identitätsmanagement, dezentralisierte Dateispeicher und dezentrale autonome Organisationen [vgl. [But17a](#)].

### 2.1.2 Modell: Globaler Transaktions-Automat

Eine Blockchain ist aus Sicht der theoretischen Informatik ein globaler Automat, der Transaktionen ausführt.

Die Ausführung von Transaktionen in der Blockchain lässt sich daher auch nach Wood<sup>1</sup> formal als Zustandstransition beschreiben [vgl. [Woo18](#), Seite 2]:

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T) \quad (2.1)$$

Dabei ist  $\sigma_{t+1}$  der neue Zustand nach Ausführung der Transitionsfunktion  $\Upsilon$ , welche auf  $\sigma_t$  (den vorherigen Zustand) und  $T$  - die Transaktion welche ausgeführt werden soll - angewandt wird.

Zur vereinfachten Erklärung wird die obige formale Definition der Übergangsfunktion anschaulicher wie folgt dargestellt [vgl. [But17a](#)]:

---

```
APPLY(S, TX) → S' or ERROR
```

---

Nachfolgend wird der Anwendungsfall eines simplen Zahlungssystems betrachtet, welches zum Beispiel bei Blockchains mit eigener Währung zum Einsatz kommt. Dabei ist der Zustand  $S$  der Eigentumsstatus *aller* verfügbaren Münzen beziehungsweise *Token*. Exemplarisch<sup>2</sup> gibt es in einem möglichst simpel gehaltenem System 2 Teilnehmer: Alice und Bob. Alice ist im Besitz von 50 ETH und Bob im Besitz von 100 ETH:

---

```
S = {Alice: 50 ETH, Bob: 100 ETH}
```

---

Soll beispielsweise die nachfolgende Transaktion TX ausgeführt werden:

---

```
TX = "Send 25 ETH von Alice zu Bob"
```

---

<sup>1</sup>Dr. Gavin Wood, Autor des im lokalen Kapitel vielzitierten *Ethereum Yellowpapers*, der Spezifikation Ethereums.

<sup>2</sup>Das Beispiel orientiert sich an dem Beispiel aus [[But17a](#), „Bitcoin As A State Transition System“].

## 2 Verwendete Technologien

---

So lassen sich die Werte einsetzen und `APPLY` kann erfolgreich ausgeführt werden:

```
APPLY({Alice: 50 ETH, Bob: 100 ETH},  
       "Sende 25 ETH von Alice zu Bob")  
       → {Alice: 25 ETH, Bob: 125 ETH} = S'
```

---

Im neuen Zustand (der abgeleitete Zustand  $S'$ ) hat Bob 125 ETH und Alice 25 ETH, nämlich genau 25 ETH weniger als zuvor.

Zu einem Fehler führt hingegen der Versuch, eine Transaktion mit unzureichendem Guthaben auszuführen:

```
APPLY({Alice: 50 ETH, Bob: 100 ETH},  
       "Sende 51 ETH von Alice zu Bob")  
       → ERROR
```

---

Zusammengefasst werden folgende Aktionen von  $\text{APPLY}(S, \text{ TX}) \rightarrow S'$  beziehungsweise der Funktion  $\Upsilon$  durchgeführt [vgl. [But17a](#)]:

1. Überprüfe, ob das zu versendende Guthaben der Transaktion  $\text{TX}$  in  $S$  im Account des Absenders vorhanden ist. Falls nicht, gib einen Fehler zurück.
2. Überprüfe, ob die kryptografische *digitale Signatur*<sup>3</sup> aus  $\text{TX}$  vom Eigentümer des zu versendenden Guthabens stammt. Falls nicht, gib einen Fehler zurück.
3. Speziell für den Anwendungsfall des Zahlungssystems: Subtrahiere das zu versendende Guthaben vom Account des Absenders, addiere es zum Account des Empfängers. Gib den neuen Zustand als  $S'$  zurück.

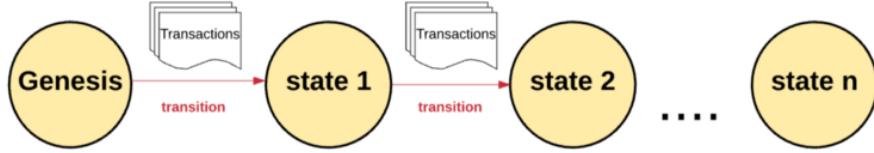
Die Definition [2.1](#) der Blockchain lässt sich nun spezifizieren.

### Definition 2.2: Blockchain (erweitert)

Eine Blockchain ist ein kryptografisch sicherer, globaler Einzelinstanz-Automat ( $\Rightarrow$  *Singleton*) für Transaktionen mit unter einer Menge von Knoten geteiltem Zustand [vgl. [Woo18](#), Seite 1] [vgl. [Kas17](#)].

---

<sup>3</sup>siehe auch [[MOV01](#), Kapitel 11] und auch [[Eck14](#), S. 398 ff. (deutschsprachig)]



**Abbildung 2.1:** Darstellung der Blockchain als (unendlicher) Automat mit  $n \in \mathbb{N}$  Zuständen.

QUELLE: [Kas17]

Der *Genesis* Zustand ist der *Startzustand* des Automaten. Zu diesem Zeitpunkt haben in der Blockchain noch keine Transaktionen stattgefunden. Wie in der Abbildung 2.1 erkennbar ist, wird bei jeder Transition eine Serie von Transaktionen ausgeführt. Dazu wird auf alle Transaktionen ( $T$ ) die Transaktions-Übergangsfunktion  $\Upsilon$  (siehe Gleichung (2.1)) angewandt. Folglich müssen alle Transaktionen gültig sein, damit der Übergang in einen neuen Zustand stattfinden kann.

Weiterhin wird die Serie von Transaktionen in einem *Block* zusammengefasst, wodurch jeder Zustand in Abbildung 2.1 ( $state_1, \dots, state_n$ ) als ein Block interpretiert werden kann.

Schließlich ergibt sich auf „Block-Level“ erweitert und abstrahiert folgende formale Definition [vgl. Woo18]:

$$\sigma_{t+1} \equiv \Pi(\sigma_t, B) \text{ mit } B \equiv (H, (T_0, T_1, \dots)) \quad (2.2)$$

Neben den einzelnen Transaktionen  $T_n$  enthält der Block  $B$  wichtige Metadaten wie beispielsweise den kryptografischen Beweis im Header  $H$ .

Im folgenden Abschnitt wird die Block-Übergangsfunktion  $\Pi$  und damit auch der Algorithmus, welcher hinter dem kryptografischen Beweis steht, genauer behandelt.

### 2.1.3 Proof of Work: Mining

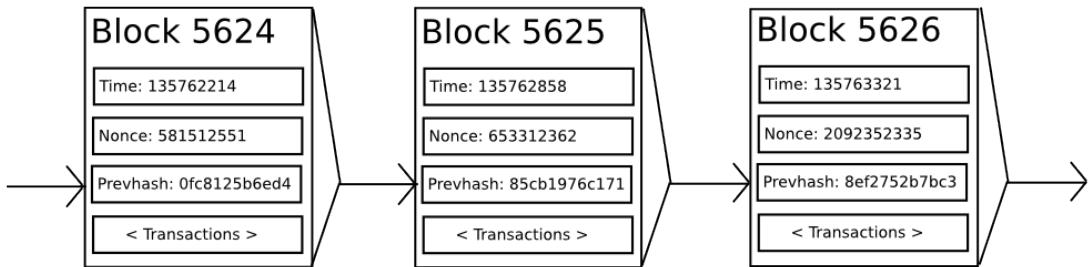
Bei einem vertrauenswürdigen *zentralem* Service gäbe es keinen Grund, das nachfolgend beschriebene Verfahren zu verwenden. Ein zentralisiertes System wird strikt nach Vorschrift implementiert und entspricht im Optimalfall vollständig den Anforderungen.

In einem *dezentralisierten* verteilten System wie der Blockchain muss hingegen sicher gestellt werden, dass es eine Einigkeit zwischen den Knoten des Netzwerks gibt: Die Teilnehmer müssen einen Konsens finden [vgl. [But17a](#), Mining].

In der vorherigen Sektion wurde mit der Transaktions-Übergangsfunktion  $\Upsilon$  sichergestellt, dass nur gültige Transaktionen ausgeführt werden. Dies ist die atomare, lokale Sicht auf eine einzelne Transaktion. Globaler betrachtet wird zusätzlich dazu ein Mechanismus benötigt, um die *Reihenfolge* der Transaktionen festzuhalten, mit der alle Knoten übereinstimmen.

Wie zuvor beschrieben werden Transaktionen in einem Block zusammengefasst. Dieser fungiert als eine Art *unveränderlicher Schnapschuss* der *neuen* Transaktionen, um den Zustand zu einem spezifiziertem Zeitpunkt festzuhalten. Der letzte Block entspricht dabei dem neusten Zustand der Blockchain (vgl. Abb. 2.1) und vice versa.

Im Rahmen der Konsensfindung konzentriert sich nun eine Teilmenge der Knoten auf die konstante Produktion neuer valider Blöcke [vgl. [But17a](#), Mining]. Dieser Prozess wird *Mining* genannt. Die Knoten, welche diese Arbeit verrichten, heißen analog zu Minenarbeitern *Miner* [vgl. [Nak08](#), Kapitel 6].



**Abbildung 2.2:** Blöcke mit Transaktionen und Metadaten.

QUELLE: [[But17a](#)]

Abb. 2.2 zeigt einen exemplarischen Ausschnitt, in dem die Blöcke als eine *einfach-verkettete Liste* dargestellt sind. Ein Block enthält relevante Metadaten im Header  $H$ : Den Zeitstempel des Blocks *Time*, die *Nonce* und den Hash des vorherigen Blocks *Prevhash* (previous Hash, dt. „Hash des Vorgängers“), durch den die Verkettung der Blöcke zu Stande kommt [vgl. [But17a](#), Mining]. Es wäre nicht unüblich, die Pfeilrichtung - anders

als im Bild - umzukehren, um zu verdeutlichen, dass der Hash des vorherigen Blocks referenziert wird.

Damit der Automat in Abbildung 2.1 in den nächsten Zustand  $\sigma_{t+1}$  übergeht, wird zur Validierung des Blocks die Funktion  $\Pi$  mit dem nachstehenden Algorithmus in Pseudo-code ausgeführt:

```

1 def apply_block(new_block: Block) -> State:
2     if new_block.previous_block is None \
3         or not new_block.previous_block.is_valid():
4         return None # FAIL
5     if new_block.time <= new_block.previous_block.time \
6         or (time() - new_block.time) >= TIMING_WINDOW:
7         return None # FAIL
8     if not new_block.check_pow():
9         return None # FAIL
10    # Start at Genesis state and validate by applying ALL tx:
11    new_state = GENESIS_STATE
12    for block in (all_blocks + [new_block]):
13        for tx in block.transactions:
14            try:
15                new_state = apply_transaction(new_state, tx)
16            except:
17                return None # FAIL
18    all_blocks.append(new_block)
19    return new_state # PASS

```

Es gibt vier relevante Überprüfungs-Teilbedingungen [vgl. But17a, Mining]:

1. **Zeilen 2 bis 4:** Validiere, dass der im Block  $B_n$  referenzierte *Vorgänger-Block*  $B_{n-1}$  existiert und gültig ist.
2. **Zeilen 5 bis 7:** Validiere, dass der Zeitstempel von  $B_n$  größer als der Zeitstempel von  $B_{n-1}$  ist und nicht mehr als ein bestimmtes Zeitfenster in der Zukunft liegt. Konkrete Beispiele: Bei Bitcoin ist dieses Zeitfenster 120 Minuten, bei Ethereum 15 Minuten [vgl. But17a, Blockchain and Mining].
3. **Zeilen 8 und 9:** Validiere den *Proof of Work* (dt. „Beweis der Arbeit“) des Blocks. Diese Bedingung wird im weiteren Textfluss detaillierter dargelegt.
4. **Zeilen 10 bis 19:** Führe beginnend vom Startzustand - dem GENESIS\_STATE - alle Transaktionen in allen Blöcken aus, einschließlich des hinzuzufügenden Blocks,

um den *finalen Zustand* der Blockchain zu erhalten. Falls ein Fehler bei einem einzigen Aufruf der Transaktions-Übergangsfunktion  $\Upsilon$  (Zeile 15) auftritt, wird der Vorgang abgebrochen. Bei Erfolg wird der neue potenzielle Block hinzugefügt und der neue Zustand zurückgegeben.

Der obige Algorithmus findet sich essenziell in jeder Blockchain-basierten Implementation mit *Proof of Work* wieder (beispielsweise sowohl in der Bitcoin als auch in der Ethereum Blockchain). Dennoch sollte er als eine eingeschränkte Sicht auf die Realität betrachtet werden; als ein Fundament des allgemeinen Blockchain-Modells, das je nach Schwerpunkt angepasst und erweitert werden kann.

Die dritte Bedingung eines gültigen Proof of Works ist besonders relevant für die Sicherheit - genauer für die Integrität - des Blockchain Netzwerks. Sie verhindert effektiv das doppelte Ausgeben von gleichen Token (im Englischen *Double-spending* genannt): Die Miner müssen einen *kryptografischen Hashwert* bereitstellen ( $\Rightarrow$  Beweis für erledigte Arbeit), der eine bestimmte vom Protokoll abhängige Bedingung erfüllt.

### Exkurs: Kryptografische Hashfunktion

Gegeben sei eine Hashfunktion  $h(x) = y$  mit beliebigem Eingabewert  $x$  und Ausgabewert  $y$  mit der festen Länge  $n$  ( $\Rightarrow$  Komprimierung).<sup>a</sup> Sie gilt nur dann als eine *kryptografische* Hashfunktion, wenn folgende Eigenschaften [vgl. [MOV01](#), S. 321-324] erfüllt werden:

1. **Einwegfunktion:** Mit einem gegebenen Hashwert  $y$  darf es nicht möglich sein, ein  $x$  zu finden, so dass gilt:  $y = h(x)$ . Die Funktion ist somit eine Einwegfunktion, da zwar mit  $x$  das Resultat  $y$  erzeugt werden kann, sich dieses jedoch nicht wieder umkehren lässt, um den Eingabewert wiederherzustellen. Mathematisch betrachtet gibt es keine Umkehrfunktion zu  $h$ ; somit ist die Abbildung auch *nicht* bijektiv.<sup>b</sup>
2. **Kollisionsresistenz:** Bei zwei Eingabewerten  $x_1$  und  $x_2$ , dabei  $x_1 \neq x_2$  darf nicht gelten, dass  $h(x_1) = h(x_2)$ . Der Hashwert darf für zwei unterschiedliche Eingabewerte niemals gleich sein, da dies eine Kollision wäre.

<sup>a</sup>Die Funktion ist nicht injektiv, da die Eingabewert- *mächtiger* als die Ausgabewert-Menge ist.

<sup>b</sup>siehe auch [Hoe14](#), S. 34 f., Def. 1.2.6 (injektiv, surjektiv u. bijektiv) u. 1.2.8 (Umkehrfunktion).

**Fallbeispiel Bitcoin:** Es wird die weit verbreitete kryptografische Hashfunktion *SHA-256* verwendet, um den Proof of Work zu erzeugen. Die Bedingung: Miner müssen einen Hashwert für den zu erstellenden Block bereitstellen, der kleiner als eine dynamisch

angepasste Zahl ist. Diese Zahl wird auch Schwierigkeitsgrad genannt, da sie die Menge der gültigen Hashwerte eingrenzt [vgl. Nak08, Kapitel 4]. Der Schwierigkeitsgrad wird vom Protokoll regelmäßig so angepasst, dass im Schnitt alle 10 Minuten ein neuer Block erstellt wird [vgl. But17a, Mining] [vgl. Nak08, Kapitel 7].

Da *SHA-256* eine Einwegfunktion ist (siehe Exkurs), gibt es keinerlei Möglichkeit, die Bedingung gezielt zu erfüllen; die Mächtigkeit der Eingabewerte lässt sich nicht eingrenzen. Von den Minern wird stattdessen - mit hohem Rechenaufwand - *Trial-and-Error* angewandt. Sie probieren alle Möglichkeiten aus, bis ein gültiger Hash gefunden wurde. Dazu wird die Nonce (siehe Abb. 2.2), welche unter anderem als Eingabewert verwendet wird, ständig erhöht und überprüft, ob die Bedingung für den Ausgabewert, den Hashwert, erfüllt ist.

Für den immensen Rechenaufwand werden die Miner entlohnt: Erstens mit einem fixen Betrag der intrinsischen Währung, auch *Kryptowährung* genannt (konkret bei Bitcoin 12.5 BTC, bei Ethereum zum Schreibzeitpunkt 3 ETH [vgl. Bit18c]). Zweitens bekommen die Miner üblicherweise die Transaktionsgebühren, welche ein Teil der Transaktionen sind [vgl. But17a, Mining]. Um die Relevanz des Minings hervorzuheben, wird nachfolgendes Szenario eines potenziellen (imaginären) Angreifers geschildert.

### Beispiel 2.1: Szenario eines Double-spending Angriffs

Bob ist im Besitz von 500 ETH. Er kauft sich mit diesen 500 ETH einen Lamborghini. Nachdem die Händlerin Alice die Zahlung durch die Transaktion  $T_1$  entgegengenommen hat, gibt sie Bob den Autoschlüssel und die Besitzurkunde.

Noch kurz bevor Bob im Lamborghini sitzt um davonzurasen, sendet er mutwillig mit seiner modifizierten Ethereum Wallet App<sup>a</sup> (dt. „Geldbörse“) eine zweite Transaktion  $T_2$  mit 500 ETH an seinen eigenen zweiten Ethereum Account. Sein bösartiger Plan: Lasse das Netzwerk glauben, dass  $T_2$  in Wirklichkeit die erste Transaktion war.

Woran Bob nicht gedacht hat: Die Händlerin Alice hat eine bestimmte Anzahl von Blöcken ( $\Rightarrow$  in diesem Kontext auch *Bestätigungen* genannt) abgewartet bevor sie Bob den Schlüssel übergeben hat. Dadurch ist die Reihenfolge der Transaktionen bereits unveränderlichpersistiert! Die zweite Transaktion  $T_2$  wird schlichtweg von den Minern verworfen.

---

<sup>a</sup>I.d. R. haben Blockchain Wallets eine lokale Überprüfung, ob ausreichend Guthaben vorhanden ist (vgl. mit *Client-Side Validation*), die jedoch von Bob absichtlich entfernt wurde.

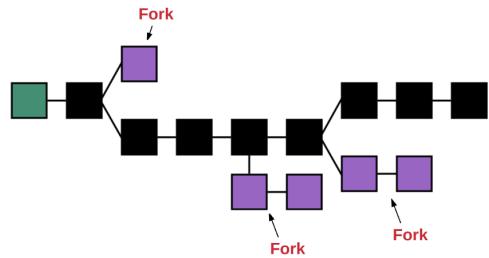
Da eine Blockchain laut Definition 2.2 ein globaler Singleton-Automat ist, gibt es auch nur genau eine globale Wahrheit.

Damit ein Angreifer erfolgreich sein kann, muss er dafür sorgen, dass seine Lüge für die Mehrheit der anderen Knoten legitim ist und damit als globale Wahrheit akzeptiert wird [vgl. But17a, Mining]. Da er die Block-Übergangsfunktion  $\Pi$ , einschließlich  $\Upsilon$ , inklusive der kryptografischen Beweise bei den anderen Knoten ohne Rechnerzugriff nicht modifizieren kann, bleibt als einzige Schwachstelle die Reihenfolge der Transaktionen übrig (siehe Beispiel 2.1).

Bei einem derartigen Angriff (oder genereller immer dann wenn Knoten sich nicht einig sind - also auch bei einer Protokolländerung) entsteht ein *Fork* (englisch für „Gabelung“) [vgl. Woo18, Kapitel 2].

In der Abbildung 2.3 sind Forks gekennzeichnet. Das grüne Kästchen ist der Block Nummer 0, nämlich der Genesis Block. Die schwarzen Kästchen zeigen den längsten *Teilbaum* mit der höchsten Anzahl an Blöcken (von der grünen Wurzel bis zum letzten schwarzen Blatt bzw. Block). Dieser Teilbaum wird als globale Wahrheit anerkannt, da die Erstellung der Blöcke am meisten Rechenleistung benötigt hat und auch *kanonische Blockchain* genannt [vgl. Kas17].

Am Ende gilt: Die angreifende Partei muss - um sicher erfolgreich zu sein - den Großteil der Rechenleistung aller Knoten besitzen.<sup>4</sup> Jeder Block ist eine zusätzliche Sicherheitsschicht; je mehr Blöcke nach dem Zeitpunkt einer Transaktion erstellt wurden, desto geringer ist die Erfolgswahrscheinlichkeit des Angreifers [vgl. But17a, Mining].



**Abbildung 2.3:** Baumstruktur mit mehreren Forks/Blockchains.

QUELLE: [Kas17]

---

<sup>4</sup>siehe auch Lea18, 51% Attack.

### 2.1.4 Ethereum Spezifisches

Die allgemeinen Blockchain Paradigmen wurden erklärt. In diesem Teil geht es um die Besonderheiten von Ethereum.

Die Philosophie der Entwicklung Ethereums setzt sich aus den folgenden Grundsätzen zusammen: Einfachheit des Protokolls, Universalität durch Turing-vollständige Smart Contracts, Modularität der Komponenten, Agilität und Freiheit durch fehlende regulatorische Vorschriften [vgl. [But17a](#), Kapiel „Philosophy“].

Diese Ansicht macht es nicht zuletzt möglich, die Ethereum Blockchain ständig durch Änderungen des Protokolls zu aktualisieren und zu verbessern ohne dabei langfristig betrachtet den Konsens unter den Knoten zu verlieren. So hat das Ethereum Netzwerk seit Mitte 2015 sieben Hardforks über- bzw. bestanden und sich somit bewährt [vgl. [Bit18b](#)]. Die mächtigen Smart Contracts, welche der größte konzeptionelle Unterschied zu Bitcoin sind, werden in den folgenden Abschnitten detaillierter beschrieben.

#### 2.1.4.1 Währung

Wie schon zuvor erwähnt, hat Ethereum - so wie Bitcoin auch - eine eigene, von innen herkommende, also *intrinsische* Währung [vgl. [Ree14](#)] namens *Ether*, abgekürzt mit ETH. Die kleinstmögliche Einheit heißt *Wei*, wobei ein Ether einer Trillion Wei entspricht, nämlich  $10^{18}$  Wei [vgl. [Woo18](#), S. 2, 2.1] [vgl. [But17a](#), Currency and Issuance]:

Multiplikator	Einheit	SI-Präfix-Ether	Zielgruppe/Verwendungszweck
$10^0$	Wei	a Ether	Technisch, Spezifikation
$10^9$	Shannon	n Ether	Technisch, z.B. Gas <sup>5</sup>
$10^{12}$	Szabo	μ Ether	Technisch, z.B. Fee
$10^{15}$	Finney	m Ether	Anwender, kl. Transaktionen <sup>6</sup>
$10^{18}$	<b>Ether</b>	—	Anwender, allg. Transaktionen

Bei den Einheiten besteht durchaus eine (gewollte) Ähnlichkeit zu physischem Geld, vergleichbar mit „Euro“ und „Cent“ als Unterwährung. Ebenfalls wurden Subeinheiten Ethers als Analogie zum amerikanischen Dollar nach populären Persönlichkeiten der

<sup>5</sup>der Begriff wird später erklärt.

<sup>6</sup> auch *Mikrotransaktionen* oder *Micropayment* (dt. „Kleinbetragzahlung“) genannt.

Kryptowährungs-Szene benannt. Zum Beispiel nach *Wei Dai*, dem Autor eines frühen Konzepts für anonyme elektronische Währungen, das auf kryptografischen Rätseln basiert oder *Nick Szabo*, dem Erfinder des Konzepts von Smart Contracts [vgl. But17a, History].

Neben dem Transfer von Werten wird die Währung auch zum Kauf von Rechenleistung für das Ausführen von Smart Contracts verwendet, da die Miner in Form der Transaktionsgebühr (im Englischen „Fee“) für ihre Arbeit entlohnt werden, damit es für sie wirtschaftlich bleibt.

Zusätzlich erhalten Miner wie in Abschnitt 2.1.3 beschrieben „aus dem Nichts“ einen fixen Betrag wenn ein Block produziert wird - auch *Block Reward* genannt. Dies ist der einzige Mechanismus zur Erstellung neuer Token und er wurde Mitte 2017 von 5 ETH auf 3 ETH reduziert. Unter anderem wurde dadurch die Inflationsrate verringert und der Weg für eine bedeutsame Protokolländerung geebnet, auf die später im Abschnitt 2.1.4.5 Bezug genommen wird [vgl. SB17].

### 2.1.4.2 Accounts

Ethereums globaler Zustand ist ähnlich wie im Beispiel des Zahlungssystems aus Abschnitt 2.1.2 durch Accounts definiert. Diese haben ihren eigenen internen Zustand und können untereinander kommunizieren (siehe Abschnitt 2.1.4.3).

Vom Endanwender werden Accounts in der Regel mithilfe einer sogenannten *Wallet-Applikation* verwaltet. Eine Wallet (dt. „Geldbörse“) ist eine Ansammlung von Accounts. Bei jedem Neuanlegen einer Wallet wird dazu ein möglichst zufälliger *Private Key* als positive 32 bit-Ganzzahl generiert [vgl. Woo18, S. 24]. Der Private Key sollte vom Anwender stets geheim gehalten werden, da sie den Besitzer dazu ermächtigen, Ether zu transferieren. Accounts haben zur eindeutigen Identifizierung eine öffentliche *Adresse*, die aus 20 Bytes besteht [vgl. Woo18, S. 3, 4.1].

Nach Wood entsteht die Adresse  $A$  aus den letzten 160 Bits vom KEC (*SHA-3 Keccak*) Hashwert des *Public Keys*, welcher wiederum aus dem zuvor zufällig erzeugten *Private Key*  $p_r$  berechnet wird [Woo18, S. 24]:  $A(p_r) = \mathcal{B}_{96..255}(\text{KEC}(\text{ECDSAPUBKEY}(p_r)))$

Im Kontrast zum Private Key  $p_r$  ist die *öffentliche* Adresse  $A(p_r)$  dazu konzipiert, ohne Risiko weitergegeben zu werden. Etwa, wenn einer anderen Person Ether überwiesen werden soll (wie zum Beispiel in Abschnitt 2.1.2 dargestellt).

### Beispiel 2.2: Ethereum Adresse

**Beispiel einer Adresse:** 0xc0ffee059173696e6cfa2c8fb9881126512f2fad

**Adresse mit Checksumme:** 0xC0fFEe059173696E6cFa2C8fB9881126512f2fAd

Die Adresse wird regulär in Hexadezimal-Darstellung mit 0x-Präfix angegeben. Abzüglich des Präfixes sind es 40 Zahlen, da eine hexadezimale Zahl von 0<sub>16</sub> bis F<sub>16</sub> 4 Bits umfasst, was genau einem halben Byte (Fachausdruck: *Nibble*) entspricht.

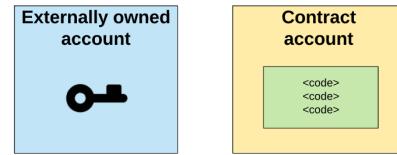
Seit Anfang 2016 wird sie standardmäßig mit einer eingebauten Checksumme ausgegeben, die darauf basiert, dass vorhandene Buchstaben (a bis f) zum Teil groß geschrieben werden, wenn der Hashwert der Adresse eine bestimmte Bedingung erfüllt.<sup>a</sup>

Für den Endanwender sorgt diese vergleichsweise simple Maßnahme dafür, dass die Wahrscheinlichkeit eines nicht entdeckten Tippfehlers auf **0,0247%** reduziert wird - ohne die Zeichenanzahl der Adresse zu verlängern. Die Überprüfung wird inzwischen von allen relevanten Clients bzw. Wallets vorgenommen (Stand: Mitte 2017) [vgl. [But16](#)].

<sup>a</sup>Für die genaue Spezifikation siehe [\[But16\]](#) inklusive voll funktionsfähigem Beispiel in Python.

Es gibt zwei Arten von Accounts [vgl. [But17a](#)]:

1. **Externally owned account (EOA):** Accounts, welche den (meist menschlichen) Besitzer des privaten Schlüssels dazu berechtigen, Transaktionen zu senden; sie werden durch den *Private Key* kontrolliert.
2. **Contract Account (CA):** Accounts in Form von Smart Contracts. Sie werden durch ihre Implementation, den Code, kontrolliert.



**Abbildung 2.4:** Die zwei Arten von Accounts.

QUELLE: [\[Kas17\]](#)

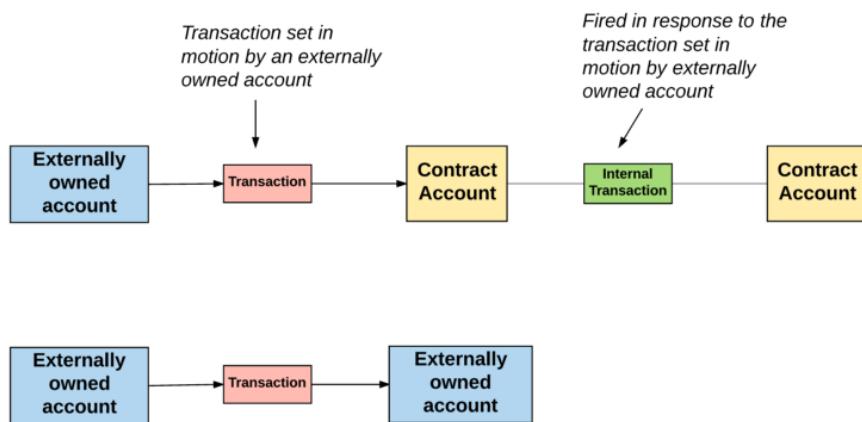
#### 2.1.4.3 Transaktionen und Nachrichten

Dieser Abschnitt befasst sich mit den Kommunikationsmöglichkeiten zur Interaktion mit dem globalen, aktuellen Zustand Ethereums. Unter den beschriebenen zwei Account Arten gibt es folgende Kommunikationsmöglichkeiten (EOA steht für Externally Owned Account bzw. externe Akteure, CA für Contract Account) [vgl. [Woo18](#), S. 4, 4.2] [vgl. [But17a](#), Messages and Transactions]:

1. EOA → CA: Der Smart Contract wird durch die eingehende Transaktion vom EOA aktiviert, sein Code ausgeführt. Wichtig: Der Pfeil ist gerichtet (unidirektional),

das heißt: CA kann keine Transaktion zu EOA von alleine senden.

2.  $CA_1 \leftrightarrow CA_2$ : Smart Contracts können, sofern zuerst durch eine Transaktion aktiviert (siehe 1.), mithilfe von Nachrichten (englisch *Messages*) untereinander kommunizieren. Dies erfolgt zum Beispiel wenn  $CA_1$  eine externe Funktion von  $CA_2$  aufruft.
3.  $EOA_1 \leftrightarrow EO A_2$ : Werttransfer zweier externer Akteure. Transaktion der Währung Ether von  $EOA_1$  zu  $EOA_2$  oder umgekehrt.



**Abbildung 2.5:** Alle drei Kommunikationsmöglichkeiten.

QUELLE: [Kas17]

Abb. 2.5 zeigt im oberen Teil exemplarisch den Nachrichtenfluss von Möglichkeit 1 und 2. Ein EOA aktiviert einen CA zuerst mittels einer Transaktion (oberer roter Kasten) um auf eine Funktion zuzugreifen. Da der Smart Contract beispielsweise Funktionalität aus einem anderen Contract benötigt, resultiert die Transaktion schließlich in einer Nachricht an einen anderen, weiteren CA. In der Abbildung wird die *Message* bzw. der *Message Call* in Form eines grünen Kastens dargestellt und aufgrund seiner Natur auch „interne Transaktion“ genannt.

Im unteren Teil der Abbildung wird Möglichkeit 3 gezeigt, speziell wie ein Werttransfer zweier externer Akteure stattfindet.

Es lässt sich folgende Definition ableiten:

### Definition 2.3: Smart Contract

Smart Contracts sind in der Ethereum Welt lebende autonome Agenten, deren Arbeit - direkt oder indirekt - ausschließlich durch externe Akteure angestoßen wird [vgl. [But17a](#), Ethereum Accounts].

Es ergeben sich zwei verschiedene Arten von Transaktionen, die durch einen externen Aktor ausgelöst werden: *Message Call Transactions* und *Contract-creating Transactions*. Eine *Message* unter zwei CA (siehe Abb. [2.5](#)) ist der Message Call Transaction untergeordnet. Insbesondere wird eine Message nicht serialisiert, da sie nur in der Laufzeitumgebung, der *Ethereum Virtual Machine*, benötigt wird [vgl. [But17a](#), Messages]. Beide Transaktionsarten nehmen u.a. die folgenden Felder an [vgl. [Woo18](#), S. 4, 4.2], welche kategorisch eingeordnet und ins Deutsche übersetzt wurden:

- **Kryptowährungs-notwendig:** Die *Empfänger-Adresse* (20 Bytes), den zu sendenden *Wert* (Integer, in Wei), die *digitale Signatur* des externen Aktors.
- **Smart Contract-relevant:** Das *Datenfeld* (Byte-Array), den *Gas-Preis* (Integer, in Wei) und das *Gas-Limit* (Integer).

Die Gas-Felder bestimmen schlussendlich, wie viel Wei ein externer Aktor maximal für die beanspruchten Ressourcen an die Miner zahlen möchte - jedoch nicht nur beim Aufruf von Smart Contracts sondern auch als Gebühr für den Werttransfer zweier EOAs.

Ferner gilt nur für die Contract-creating Transaktion: Sie enthält ein zusätzliches *Initialisierungsfeld* mit dem Bytecode des zu erstellenden Smart Contracts (siehe nächster Abschnitt). Zudem ist die *Empfänger-Adresse* nicht spezifiziert (leer); schließlich ist das Ziel dieser Transaktion, einen neuen Smart Contract zu erstellen. Daher wird das leere Feld wie ein „Broadcast“ an das gesamte Netzwerk von Knoten ohne festen Empfänger interpretiert.

Eine erhebliche Rolle nimmt das Feld der *digitalen Signatur* ein: Sie beschreibt erstens den Absender und stellt zweitens kryptografisch sicher, dass eine Transaktion nur ausgeführt wird, wenn sie vom Besitzer des *Private Key* genehmigt ( $\Rightarrow$  unterschrieben) wurde.

Zum Abschluss dieses Abschnitts folgt die Definition einer Transaktion:

### Definition 2.4: Transaktion

Eine Transaktion ist eine von einem externen Aktor digital signierte Anweisung, welche die Brücke zwischen der Außenwelt und dem globalen Zustand der Ethereum Welt bildet [vgl. [Woo18](#)] [vgl. [Kas17](#)].

#### 2.1.4.4 Ethereum Virtual Machine

In dieser Sektion geht es um das „Herz“ Ethereums, die Ethereum Virtual Machine (kurz: EVM). Die EVM ist eine Laufzeitumgebung, welche die Ausführung des *Bytecodes* von Smart Contracts ermöglicht [vgl. [But17a](#)].

Entwickler schreiben zuerst Code für Smart Contracts in einer speziell dafür vorgesehnen - menschlich lesbaren - höheren Programmiersprache (zum Beispiel in dieser Arbeit *Solidity*). Nach Fertigstellung wird dieser zu einem Bytecode, konkret dem *EVM-Code*, kompiliert. Nachdem der kompilierte Smart Contract mittels einer Contract-creating Transaction migriert wurde (auch *Deployment* genannt), kann dieser von den Knoten des Netzwerks in der EVM ausgeführt werden [vgl. [But17a](#)].

Jedes Byte des Bytecodes ( $\Rightarrow$  eine Serie von Bytes) ist einem *Operationscode* (auch *Instruktion* genannt) zugeordnet, der eine *Operation* ausführt. Die EVM ist eine *stack-basierte*<sup>7</sup> virtuelle Maschine, das heißt, dass die Operationen nach *LIFO-Art* (Last In – First Out) abgearbeitet werden. Der Stack ist limitiert auf 1024 Elemente und hat eine Wortgröße von 256 Bit pro Element [vgl. [Woo18](#), S. 11].

---

<sup>7</sup>siehe auch [But17b](#), für Hintergrund und Historie hinter dieser Entscheidung.

### Beispiel 2.3: Beispiel zur Stack-basierten Architektur

Gegeben sei der folgende Code (Solidity-Ausschnitt):  $42 == 1;$   
 Nach der Kompilierung entsteht daraus der Bytecode:  $0x6001602a14$   
 Dies entspricht den Operationscodes<sup>a</sup>:  $PUSH1\ 0x1;\ PUSH1\ 0x2A;\ EQ;$

Der Bytecode (in Hexadezimal-Darstellung) hat eine Größe von 5 Bytes.

Dabei entspricht  $0x60$  der `PUSH1` Operation,  $0x14$  der `EQ` Operation.

Die Funktion des Codes ist trivial: Vergleiche die Ganzzahl 42 mit der Ganzzahl 1 auf Gleichheit. Dazu werden die beiden Operanden 1 (=  $0x01$ ) und 42 (=  $0x2A$ ) zunächst mit dem `PUSH1` Befehl auf den Stack gelegt.

Danach wird mittels `EQ` Operator überprüft, ob die beiden Elemente gleich sind. Der `EQ` Operator liest immer das erste und das zweite Element des Stacks (das erste Element ist das oberste Element) [vgl. [Woo18](#), S. 29-33].

<sup>a</sup>Zur besseren Lesbarkeit wurden die Semikolons als Trennzeichen hinzugefügt. Das Beispiel wurde mithilfe der IDE „Remix“ erstellt, die u.a. Bytecode anzeigt [siehe auch [Eth18c](#)].

Jede Ausführung einer Instruktion – also jede Operation – kostet *Gas*. Jede Instruktion ist einem fixen Gaswert zugeordnet, der je nach Beanspruchung von Rechner-Ressourcen höher oder niedriger ausfällt. Die fixen Gas-Kosten pro Instruktion sind im *Ethereum Yellowpaper* spezifiziert.<sup>8</sup> Beispielsweise würde die `PUSH1` Operation aus dem Beispiel [2.3](#) konstante 3 Gas verbrauchen. Gas ist nicht zu verwechseln mit der intrinsischen Währung Ether; daher auch die Einführung eines anderen Begriffs [vgl. [Woo18](#)]. Ferner dient das Gas dazu, die Kosten für Instruktionen von der intrinsischen Währung zu entkoppeln.

Der Absender einer Transaktion spezifiziert (siehe Abschnitt [2.1.4.3](#)) mit dem *Gas-Preis* wie viel *Wei*<sup>9</sup> der Währung Ether (siehe Abschnitt [2.1.4.1](#)) er für *ein* Gas ausgibt. Mit dem *Gas-Limit* gibt er an, wie viel Gas er maximal ausgeben möchte. Die maximal auszugebende Gebühr in Ether (englisch „Fee“) für eine Transaktion errechnet sich infolgedessen wie folgt:  $\text{GASPRICE} * \text{GASLIMIT}$ . Die Miner favorisieren dabei in der Regel die Transaktionen von denen sie am meisten profitieren; jene, die den höchsten Gas-Preis offerieren. Natürlicherweise fungiert dieses Gebührenschema auch als defensiver Mechanismus gegen *Denial-of-Service* Attacken, da es für Angreifer schnell unwirtschaftlich

<sup>8</sup>siehe auch [Woo18](#).

<sup>9</sup>oft auch in der Subwährung *Shannon* angegeben

wird, das ganze Netzwerk mutwillig auszulasten (zum Beispiel mit einer „unendlichen“ Schleife) [But17a].

Es muss nicht nur für Rechenleistung gezahlt werden, sondern auch für beanspruchten Speicherplatz. Neben dem Stack gibt es in der EVM folgende **Typen von Datenspeichern** [vgl. Woo18, S. 11, 9.1]:

- **Volatiler Hauptspeicher:** Ein „unendlich großer“ volatiler (flüchtiger) Speicher, der intern als ein erweiterbarer Byte-Array realisiert ist.
- **Nicht-volatiler Speicher:** Zur permanenten Speicherung von Daten in Smart Contracts ( $\Rightarrow$  Persistenz) verfügt die EVM über einen nichtflüchtigen *Key-Value* Speicher, der Teil des Systemzustands Ethereums ist. In diesem *assoziativen Datenfeld* <sup>10</sup> werden Werte (*Values*) mit Schlüsseln (*Keys*) adressiert.
- **Read-Only Memory:** Der EVM-Code des Smart Contracts selbst ist in einem virtuellen Festwertspeicher (ROM) abgelegt. Dieser kann *nicht* manipuliert werden und auf ihn kann nur mit speziellen Instruktionen *lesend* zugegriffen werden. Aufgrund dieser besonderen Eigenschaft ist die EVM keine Maschine nach der klassischen *Von-Neumann-Architektur*, da Programmcode bei diesem Referenzmodell in einem Schreib-Lese-Speicher (englisch „Read-Write Memory“) gespeichert wird.

**Der Prozessor der EVM:** Nach Buterin lässt sich der Zustand der EVM zur *Ausführungszeit* als ein Tupel beschreiben [Kapitel „Code Execution“ But17a]:

(block\_state, transaction, message, code, memory, stack, pc, gas)

Die EVM ist eine vollständig deterministische Maschine [vgl. Woo18, S. 15], d.h. die Ausführung der Smart Contracts durch den *Prozessor* muss deterministisch erfolgen, da jeder Knoten mit dem gleichen Tupel an Eingabewerten auch den gleichen Tupel an Ausgabewerten erhalten (berechnen) muss. Wenn das Gegenteil der Fall wäre, könnten die Netzwerkteilnehmer keinen Konsens finden, da jeder Knoten einen eigenen Fork produzieren würde (siehe Abschnitt 2.1, insbesondere Abb. 2.3).

---

<sup>10</sup>in einigen Programmiersprachen auch bekannt als Dictionary, Hashmap oder Hashtable

### \* Beispiel 2.4: Zeit und Zufall in Smart Contracts

Das Abrufen von **Zeit** ist *nicht* deterministisch. Dies heißt jedoch nicht, dass es unmöglich ist, auf eine *ungefähre* Zeit in einem Smart Contract zuzugreifen. Der Konsens unter den Knoten – und damit auch der Determinismus – entsteht bei der Zeit dadurch, dass auf ein spezielles *statisches* Zeit-Feld des Blocks zugegriffen werden kann (in Solidity `block.timestamp` genannt) [vgl. Eth18h]. Bei der Produktion eines neuen Blocks trägt der Miner *seine* Zeit ein; sie ist also bis zu einem bestimmten Grad vom Miner beeinflussbar (siehe Code in Abschnitt 2.1.3, insb. Bed. Nr. 2).

Noch weniger trivial ist es, Pseudo-**Zufallszahlen** in einem Smart Contract zu generieren: Jeder Knoten müsste die gleiche „Zufallszahl“ berechnen, was schlichtweg gegen die Definition einer zufälligen Zahl ist. Allerdings gibt es dennoch praxistaugliche Möglichkeiten: Jeder Block verfügt über einen Proof-of-Work *Block-Hashwert* (siehe Abschnitt 2.1.3). Die Eigenschaft, dass zum Sendezeitpunkt der Transaktion der Hashwert noch nicht bekannt ist, lässt sich ausnutzen. Da der Wert durch eine Hashfunktion ( $\Rightarrow$  Einwegfunktion) mittels *Trial-and-Error* entsteht, kann dieser nicht vorhersagt werden; er hat aufgrund dessen gute Eigenschaften für eine Pseudo-Zufallszahl. Aus Sicherheitsgründen werden in der Praxis die letzten 256 Block-Hashwerte mit einbezogen, damit eine bösartige Mining-Partei das Ergebnis nicht beeinflussen kann [vgl. Woo18, S. 15].

Der Prozessor führt den Bytecode aus algorithmischer Sicht wie folgt aus: Zu Beginn werden der *Program Counter* (pc, dt. „Befehlszähler“) mit 0, der Stack und alle weiteren Speichertypen initialisiert. Die Instruktion des Bytecodes wird an der Stelle, welche im Program Counter steht, eingelesen. Dann wird die Instruktion bzw. Operation abgearbeitet und der Program Counter um 1 inkrementiert. Das Gas, welches die Operation benötigt, wird verbraucht. Die Ausführung der Operationen läuft so lange, bis das Programm entweder normal terminiert oder ein Ausnahmezustand bzw. eine *Exception* eintritt. Wenn ein Ausnahmezustand eingetreten ist, so werden alle Änderungen am Zustand rückgängig gemacht (im Englischen auch *Revert* genannt) [But17a].

Beispiele für Ausnahmezustände sind: invalide Instruktionen, Stack-Unterläufe oder die *Out-of-Gas Exception* [vgl. Woo18, 11 ff.]. Letztere tritt auf, wenn für eine weitere Ausführung kein Gas mehr zur Verfügung steht. Das *verbrauchte* Gas wird allerdings logischerweise nicht wieder zurückgegeben; es liegt vielmehr in der Verantwortung des Absenders der Transaktion, das Gas-Limit angemessen zu setzen. Bei vertrauenswürdigen Smart Contracts bietet es sich jedoch durchaus an, das *Gas-Limit* großzügig anzusetzen, da nicht verbrauchtes Gas in Form von *Ether* zum Originalpreis erstattet wird [vgl.

[Woo18, S. 7].

Die EVM selbst ist streng genommen keine zu 100% Turing-vollständige Maschine, da ihre Berechnung durch das Gebührenschema, also den Gasverbrauch, künstlich eingeschränkt wird [vgl. Woo18, S. 11, 9.0].

### 2.1.4.5 Weitere Anmerkungen

In diesem Abschnitt geht es um sinnvolle Ergänzungen, die es nicht in die anderen Abschnitte geschafft haben, nichtsdestotrotz aber eine Relevanz für die folgenden Kapitel haben.

**Proof of Stake:** Wie am Ende von Abschnitt 2.1.4.1 erwähnt wird, gibt es eine (zum Schreibzeitpunkt) bevorstehende Protokolländerung. Diese Änderung ist die Aktualisierung des Protokolls vom Mining-basierten *Proof of Work* nach *Proof of Stake*. Zusammengefasst handelt es sich dabei um einen anderen Algorithmus zur *Konsensfindung*, der in verbesserter Sicherheit, verringertem Zentralisierungs-Risiko und mehr Energieeffizienz resultiert. Statt kryptografische Rätsel zu lösen gibt es Transaktions-validierende Wähler, deren Stimmrecht umso mehr ins Gewicht fällt, je mehr sie an Ether hinterlegt haben (englisch *Stake*) [Eth18g].

**Scheduling von Smart Contracts:** In Abschnitt 2.1.4.3 wird beschrieben, dass Smart Contracts immer einen externen Aktor benötigen, um ausgeführt zu werden. Dennoch ist es möglich, mithilfe eines speziell dafür programmierten Smart Contracts *Scheduling* (d.h. zeitgesteuertes Ausführen) zu betreiben. Das Projekt heißt *Ethereum Alarm Clock* und ist im einfachen Sinne ein „Marktplatz“ für das Ausführen von Transaktionen zu einem bestimmten Zeitpunkt durch externe Akteure. Die Voraussetzung für die korrekte Funktionsweise ist jedoch, dass es genug externe Akteure gibt, die motiviert sind, Transaktionen zu fremden Smart Contracts senden [Ala18].

**Data Feeds:** Ein Smart Contract bzw. ein *Contract Account* kann nur auf den internen globalen Zustand Ethereums zugreifen, nicht aber auf die Außenwelt (das Internet). Im ersten Moment scheint dies für die Entwickler einschränkend zu sein, da Daten wie z.B. die Wertstellung von Ether in Euro nicht aktiv ermittelt werden können. Aller-

dings besteht die Möglichkeit, Smart Contracts stets mit aktuellen Daten von Außen zu versorgen. Um dies zu erreichen werden regelmäßig *Message Call* Transaktionen ausgeführt, die den Zustand des Smart Contracts aktualisieren und „füttern“. Dazu bietet es sich an, einen Server zu betreiben, der ständig die benötigten Daten aus dem Internet herunterlädt um sie an den Smart Contract weiterzugeben [vgl. [Woo18](#), S. 15].<sup>11</sup>.

### 2.1.4.6 Zusammenfassung

Zurück zum Ursprung – nämlich zur allgemeinen Transaktions-Übergangsfunktion einer Blockchain nach Wood (Gleichung (2.1) aus Abschnitt 2.1.2):

$$\sigma_{t+1} \equiv \Upsilon(\sigma_t, T)$$

Mit den dazu gewonnenen Erkenntnissen lässt sich nun sagen, dass speziell bei Ethereum die Funktion  $\Upsilon$  sehr mächtig ist; sie ist für die Ausführung der Transaktion  $T$  in der EVM zuständig. Nichtsdestoweniger ist es der Zustand  $\sigma$ , der in der Lage ist, das Resultat der Transaktionsausführung festzuhalten, damit weitere Transitionen überhaupt erst möglich sind [vgl. [Woo18](#), S. 2].

Statt lediglich ein dezentrales Zahlungssystem wie beispielsweise Bitcoin zu sein (siehe Abschnitt 2.1.2), so ist Ethereum dank der Smart Contracts letztendlich eine „Plattform für dezentralisierte Applikationen der nächsten Generation“ [[But17a](#)].

---

<sup>11</sup>Die dadurch entstehende Zentralisierung sollte beachtet werden.

## 2.2 Camunda BPM

Camunda BPM ist ein *Open-Source Softwarestack* (eine Ansammlung von Softwarekomponenten), der es ermöglicht, *Geschäftsprozesse* bzw. *Workflows* zu modellieren und auszuführen [vgl. [Cam18d](#)].

Der Fokus wird dabei stark auf die Umsetzung folgender Standards der Object Management Group (OMG) gerichtet, die ein nichtkommerzielles Konsortium für industrielle Standards ist [vgl. [OMG11](#), S. xxiii]:

- **BPMN 2.0 - Business Process Model and Notation:**

Das primäre Ziel dieser Notation ist es, jedem Beteiligtem eines Geschäftsprozesses eine verständliche Darstellung dieses Prozesses zu vermitteln. Dabei soll mit Hilfe von Einheitlichkeit und Konsistenz „die Lücke zwischen Prozess-Design und Prozess-Implementierung“ geschlossen werden [[OMG11](#), S. 1, übersetzt]. Dies führt auch zu weniger Komplikationen bei der Kommunikation: Beispielsweise haben sowohl der Business Analyst, der den Prozess modelliert, als auch der Entwickler, der diesen implementiert, einen gemeinsamen Nenner. Letztendlich ist dieser Nenner eine standardisierte *Sammlung von Diagrammelementen*, die möglichst in allen relevanten Abteilungen des Prozesses bekannt sein sollte [vgl. [OMG11](#)].

- **DMN 1.1 - Decision Model and Notation:**

DMN kann - separat standardisiert und unabhängig einsetzbar - als Kompagnon von BPMN betrachtet werden, der es allen zuständigen Personen ermöglicht, ihren Input zu *wiederholbaren geschäftlichen Entscheidungen* beizusteuern. Im XML-Format definiert ist es insbesondere ein Ziel der Notation, auch organisationsübergreifend *austauschbar* zu sein. Ermöglicht wird dies, da die Entscheidungs-Modelle von den Geschäftsprozessen (z.B. in BPMN vorliegend) losgelöst sind, aber trotzdem miteinander interagieren können [vgl. [OMG16](#), S. 9].

Von Camunda wird der umfangreiche BPMN 2.0 Standard teilweise umgesetzt [vgl. [Gei+15](#)]. Der DMN 1.1 Standard wird hingegen vollständig implementiert [vgl. [DMN18](#)].

Die Softwarekomponenten des Camunda Stacks (siehe Abb. 2.6) lassen in drei Phasen einsortieren:

1. **Modellieren:** Mithilfe des *Modeler* den Geschäftsprozess modellieren.
2. **Ausführen:** Das Modell mit der *Workflow Engine* und der *Decision Engine* ausführen und testen. Dazu können vom Anwender die Web-Frontends *Tasklist*, *Cockpit* und *Admin* genutzt werden.
3. **Verbessern:** Nach der Ausführung auf Basis der Testergebnisse<sup>12</sup> das Modell verbessern. Danach wieder bei Phase 1 anfangen und die nächste Iteration starten.

Im Folgenden werden der Modeler und die Komponenten der zweiten Phase (als *Camunda Platform* zusammengefasst) genauer beschrieben.

### 2.2.1 Modeler

BPMN-Workflows und DMN-konforme Entscheidungstabellen können dank der einfach zu bedienenden Software *Camunda Modeler* grafisch modelliert werden. Der Modeler folgt dabei dem *WYSIWYG-Prinzip* („What You See Is What You Get“), wodurch auch modelliert werden kann ohne das dahinter stehende XML-Format direkt zu modifizieren (oder zu verstehen). Technischere Anwender können jedoch – sofern nötig – ebenso direkt die XML-basierte BPMN-Datei bearbeiten [Cam18g].

Der Modeler basiert auf einer JavaScript Implementierung namens *bpmn-js*, die auch ohne Installation online getestet werden kann.<sup>13</sup>

Aufgrund der modularen Struktur ist es möglich, Plugins zu entwickeln und zu verwenden. So existiert beispielsweise ein *Token Simulation Plugin* mit dem die *Syntax* (Anordnung der Diagrammelemente) und teilweise die *Semantik* (Bedeutung der Dia-

<sup>12</sup>und/oder mittels Monitoring durch *Camunda Optimize*, das jedoch nur in der kommerziellen Variante *Camunda Enterprise* verfügbar ist, die im weiteren Verlauf nicht behandelt wird.

<sup>13</sup>siehe auch [BPM18](#).



**Abbildung 2.6:** Der Camunda Stack.

QUELLE: Screenshot [Cam18d]

grammelemente) des Modells getestet werden können.<sup>14</sup>

Nach Abschluss der Prozess-Modellierung und -Entwicklung wird der Prozess typischerweise der Camunda Plattform bereitgestellt, die im nächsten Abschnitt beschrieben wird. Diese Bereitstellung wird in der Fachsprache auch *Deployment* genannt.

### 2.2.2 Platform

Nachdem der Geschäftsprozess *deployed* wurde, also die Deployment-Phase abgeschlossen ist, kann er von den „Engines“ (respektive *BPMN Workflow Engine* und *DMN Decision Engine*) ausgeführt werden. Diese beiden Engines (dt. „Motoren“) sind als Prozess-ausführende Einheit der *Kern* der Camunda Platform. Die Camunda Platform (im Nachfolgenden auch deutsch „Plattform“ genannt) stellt dem Prozessentwickler dabei verschiedene Möglichkeiten bereit, um den Kern anzusprechen; so kann sowohl via REST als auch mit einer Java API auf ihn zugegriffen werden. Der in jeder JVM ausführbare Kern ist mit einer Größe von unter 5 MB schlank und arbeitet performant genug, um auch in großen Unternehmen eingesetzt zu werden [vgl. [Cam18h](#)].

Im oberen Teil der Abb. [2.6](#) befinden sich die verschiedenen Webapplikationen bzw. Frontends, die für die Endbenutzer gedacht sind:

Nach dem Start des Prozesses können mit der *Tasklist* Aufgaben (englisch *Tasks*) abgearbeitet werden. Im BPMN-Modell sind die für die Aufgaben zuständigen Personen (englisch *Assignees*) und die dazugehörigen Formulare definiert [vgl. [Cam18c](#)].

Für das technische Monitoring der Prozesse stellt die Plattform das *Cockpit* zur Verfügung. Wichtig ist diese Komponente, um sich einen Überblick über laufende Prozessinstanzen zu verschaffen und um Fehler während der Prozessausführung (zum Beispiel durch eine fehlgeschlagene *Service Task*) gezielt zu identifizieren [vgl. [Cam18b](#)].

Die *Admin* Applikation ist eine Benutzerverwaltung, mit der Administratoren kontrollieren können, *wer* sich anmelden darf. Zudem kann konfiguriert werden, *wie* und auf *was* (welche Ressourcen) der Benutzer zugreifen darf [vgl. [Cam18a](#)].

Als Ganzes betrachtet gibt es die Plattform in verschiedenen Ausführungen, auch *Distributionen* genannt [[Cam18f](#)]. Im Rahmen dieser Arbeit wird die *Full Distribution* auf Basis des Webservers *Apache Tomcat* verwendet.

---

<sup>14</sup>siehe auch [Fro17](#).

## 2.3 Vagrant

Vagrant ist eine Open-Source Software, um möglichst reproduzierbar Entwicklungssysteme zu erzeugen, die isoliert vom Hostrechner in einer virtuellen Maschine (Gastsystem) laufen [vgl. Has13, S. 1 f.]. Das Tool agiert als Wrapper um bestehende Virtualisierungslösungen, bei Vagrant auch *Provider* genannt. Seit Version 1.1 steht es dem Benutzer frei, die Software zur Virtualisierung selbst zu wählen; vorher wurde nur das weit verbreitete *VirtualBox* unterstützt [vgl. Has13, S. 5 u. 13]. Die eigentliche Installation und Konfiguration der Softwarekomponenten erfolgt mittels Skripten (z.B. Shell-Skripte), welche die zur Automatisierung notwendigen Schritte ausführen (auf Englisch *Provisioning* genannt) [vgl. Has13, 1 ff.]. Um den Nutzen von Vagrant besser nachvollziehen zu können, folgt das Beispiel 2.5, bei dem Vagrant *nicht* zum Einsatz kommt.

### Beispiel 2.5: Szenario einer Installationsanleitung

Bob hat eine Anleitung geschrieben, die beschreibt, wie Camunda BPM inklusive aller Abhängigkeiten auf einem Linux System installiert wird.

Alice hält sich strikt an die Anleitung. Nach dem Download und Entpacken der Camunda Distribution versucht sie die benötigte Java Laufzeitumgebung zu installieren. Doch der von Bob dokumentierte Befehl scheitert mit der Meldung „apt-get: command not found“. Die Problematik: Bob hat die Anleitung unter seiner *Debian* Distribution getestet und angenommen, dass die Befehle auch auf anderen GNU/Linux-basierten Systemen funktionieren. Alice verwendet jedoch die *openSUSE* Distribution, bei der standardmäßig ein gänzlich anderer Paketmanager zum Einsatz kommt [ope16].

Vagrant hat das Ziel, die klassische Installationsanleitung zu ersetzen. Aufgrund der Vielfalt an Betriebssystemen und Konfigurationsparametern sind Anleitungen oftmals fehleranfällig, nicht zuletzt da sie manuell ausgeführt werden. Mit Vagrant müssen Entwickler normalerweise nur einen einzigen Befehl (`vagrant up`) ausführen, damit die folgenden Schritte automatisch durchgeführt werden [vgl. Has13, S. 1]:

- 1. Erstellung der VM:** Die virtuelle Maschine wird mit einem konfigurierbarem Betriebssystem und weiteren Parametern (wie Netzwerkverbindungen) erstellt.
- 2. Konfiguration der VM:** Unter anderem wird hier die Verbindung zwischen Host- und Gastsystem für geteilte Ordner und Netzwerkverbindungen hergestellt.
- 3. Hochfahren:** Die Maschine wird gestartet.

4. **Provisioning:** Die benötigten Softwarekomponenten werden installiert und konfiguriert, so dass das System für den vorgesehenen Einsatzzweck bereit ist.

Zusammengefasst ermöglicht es die Software, eine vollständige Systemumgebung aufzusetzen, die für ein spezifiziertes Betriebssystem alle benötigten Softwarekomponenten vorkonfiguriert beinhaltet.

## 2.4 Spring Boot

Spring Boot ist eine auf dem *Spring Framework* basierende Java-Lösung, die dem Entwickler alltägliche Konfigurationsarbeiten abnimmt, um möglichst schnell ein fertiges Softwareprodukt ausliefern zu können. Dabei ist das Spring Framework eine Alternative zur *Java Enterprise Edition* (JEE), die eine andere Philosophie verfolgt: Statt stärker an die Implementierung von Interfaces gebunden zu sein, werden häufig POJOs (*Plain Old Java Objects*) eingesetzt, die mit der eingesetzten Technologie maximal nur *loose gekoppelt* sind. Ermöglicht wird dies durch die sinnvolle Kombination von *Dependency Injection* und *Annotationen* [vgl. [Wal15](#), Abschnitt 1.1].<sup>15</sup>

Von *Spring Boot* werden typische Funktionalitäten (z.B. ein REST-Webservice oder eine Datenbankanbindung) in Form von sogenannten *Startern* zusammengefasst, die alle benötigten Abhängigkeiten mitbringen. Die Starter werden dabei automatisch (vor-)konfiguriert, so dass der Entwickler nur in spezielleren Fällen die Softwarekomponenten der Starter konfigurieren muss [vgl. [Wal15](#), Abschnitt 1.1.2]. Dieses Softwaredesign-Paradigma wird auch *Convention over Configuration* (dt. „Konvention vor Konfiguration“) genannt. Die notwendige Bedingung für den Erfolg des Paradigmas ist eine vollständige Dokumentation, welche die Konventionen bzw. Regeln offenlegt.<sup>16</sup>

## 2.5 Docker

Bei Docker handelt es sich um eine im März 2013 veröffentlichte Technologie, mit der Applikationen in sogenannten *Containern* ausgeführt werden können, die alle eigenen relevanten Abhängigkeiten mitbringen. Container unterscheiden sich von virtuellen

---

<sup>15</sup>siehe auch [Spr18b](#), für Beispielcode eines POJOs mit Annotationen.

<sup>16</sup>siehe auch [Spr18a](#), für die offizielle Spring Boot Anleitung mit 398 Seiten.

Maschinen, da sie weniger isoliert vom Host-Betriebssystem sind – sie teilen sich den Betriebssystem-Kernel und sind leichtgewichtiger. Prozesse in Containern laufen daher ohne den *Overhead* eines kompletten zusätzlichen Betriebssystems. Aus Performance-Sicht sind die Prozesse nahezu gleichzusetzen mit der nativen Ausführung, wodurch sie gute Skalierungseigenschaften vorweisen [vgl. [Mou15](#), Kapitel 1].

## 2.6 Hardware

### 2.6.1 Raspberry Pi

Der Raspberry Pi ist eine Serie von Computern im Kreditkartenformat, bei der alle Hardwareteile – bis auf das Netzteil – auf einer einzigen Platine positioniert sind (auch *Einplatinencomputer* genannt). Möglich wird dies u.a. durch Einsatz eines *System on a Chip* (SoC), der alle notwendigen elektronischen Komponenten (wie Hauptprozessor, Grafikeinheit und Arbeitsspeicher) vereint [vgl. [Pi18a](#)]. Zudem verfügen Raspberry Pis über GPIO-Pins, die genutzt werden, um zusätzliche Hardware anzuschließen.

Die *Raspberry Pi Foundation* ist eine im Jahr 2008 gegründete britische Wohltätigkeitsorganisation, die es sich zum Ziel gemacht hat, möglichst kostengünstig vergleichsweise leistungsfähige Hardware an Menschen rund um den Globus verfügbar zu machen [vgl. [Pi18b](#)]. Dabei steht das pädagogische Ziel „Learning through making“ im Fokus [[Pi18b](#), S. 5 ff.]. Die breite Verfügbarkeit spiegelt sich in den Verkaufszahlen wieder: Seit der ersten Raspberry Pi Version im Jahr 2012 [vgl. [Pi18b](#), S. 3] wurden bis März 2018 rund 19 Millionen Einheiten verkauft [vgl. [Upt18](#)].

Jedoch gibt es nicht nur im Hobbybereich Anwendungen für den Einplatinencomputer, da für industrielle Einsatzzwecke und Anforderungen das *Compute Unit* angeboten wird. Mit diesem Modell ist der Übergang vom *Prototypen* zum fertigen *Produkt* im Ökosystem Raspberry Pi mit wenig Komplikationen möglich.

Im Rahmen dieser Arbeit kommt mit dem *Raspberry Pi Zero* das günstigste (mit einem Nettopreis von 5 USD gelistete) Modell der Serie zum Einsatz, das folgende technische Eigenschaften hat [Pi18a]:

- **SoC:** Broadcom BCM2835
- **CPU:** 1000 MHz, Single Core
- **RAM:** 512 MB
- **Ports:** 1 USB-OTG, 1 mini-HDMI <sup>17</sup>
- **Empfohlener Energiebedarf:** 1.2 A
- **Maße:** 65 mm \* 30 mm \* 5.4 mm
- **Gewicht:** 9 g



Abbildung 2.7: Raspberry  $\pi_0$  v1.3

QUELLE: [Amo16]

### 2.6.2 RFID und NFC

Radio-Frequency Identification (RFID) ist eine Technik, die ein elektromagnetisches Feld aufbaut, um den Informationsaustausch zwischen einem *Transponder* und einem *Lesegerät*<sup>18</sup> möglich zu machen. Der Transponder ist entweder *aktiv* (mit eigener integrierter Stromquelle, etwa einem Akku) oder *passiv*. Er verfügt über ein Speicherelement, in der Regel mit weniger als 1 kB Speicherplatz [vgl. JCI14, S. 13]. Bei passiven Transpondern (z.B. ein sogenannter *Tag* in Kartenform) wird das vom Lesegerät aufgebaute elektromagnetische Feld als Stromquelle genutzt, denn zur Übertragung von Daten wird nur wenig Energie benötigt. Aktive Transponder kommen hingegen meist dann zum Einsatz, wenn eine größere Reichweite benötigt wird. Mit RFID werden nur geringe Datenmengen übertragen, oftmals auch nur eine UID (*Unique Identifier Number*), die speziell zur Identifizierung der Karte verwendet werden kann. Diese eindeutige ID wird für bestimmte Einsatzgebiete in einer Datenbank referenziert, um zusätzliche Informationen zum Inhaber der Karte abzuspeichern [vgl. JCI14, S. 11 u. 13]. Exemplarisch könnte die extern gespeicherte zusätzliche Information bei einem Zahlungssystem das verbleibende Guthaben oder bei einem Alarmsystem der vollständige Name des Karteninhabers sein.

<sup>17</sup>Seit Version 1.3 verfügt der Raspberry Pi Zero zusätzlich auch über einen speziellen Kamera-Port.

<sup>18</sup>Im Folgenden wird generell nicht ausgeschlossen, dass das Lesegerät auch über eine Schreibfunktion verfügt. Für den Leseffluss ist es allerdings vorteilhafter, nur von einem Lesegerät zu sprechen.

Near Field Communication (NFC) ist eine Sammlung weit verbreiteter Standards, die beschreibt, *wie* die Kommunikation zwischen dem Transponder (bzw. Tag) und dem Lesegerät abläuft. Die Standards können daher auch als *Kommunikationsprotokolle* betrachtet werden, die RFID als Basis nehmen und spezifizieren [vgl. [JCI14](#), S. 14 ff.].

### Beispiel 2.6: Analogie aus der OOP – Abgrenzung RFID/NFC

Aus Sicht der objektorientierten Programmierung kann RFID als Basisklasse *eines* NFC-Standards ansehen werden: Die NFC-Klasse ist von RFID abgeleitet und bietet mehr Funktionalität – sie ist spezifischer und eine Erweiterung; so werden z.B. Kommunikationsmöglichkeiten ergänzt, etwa das Auslesen und Schreiben von *verschlüsselten* Daten.

Genauso wie RFID hat auch NFC einen aktiven und einen passiven Kommunikationsmodus, wobei die Reichweite (der Abstand zwischen beiden Geräten) mit 10 cm oder weniger bei beiden Modi bewusst gering gehalten ist [vgl. [JCI14](#), S. 12 u. 14]. Passiv wird beispielsweise auf eine Smart Card zugegriffen, da sie über keine Stromquelle verfügt. Wohingegen aktive Übertragung zustande kommt, wenn etwa zwei Smartphones mit NFC-Funktion Daten austauschen [vgl. [JCI14](#), S. 14].

NFC findet in vielen Bereichen Anwendung, sowohl im Verbrauchermarkt als auch in der Industrie: So wird die Technologie in Form von Smart Cards (bspw. Kreditkarten) zum kontaktlosen Zahlen, zum Identitätsnachweis (z.B. in Hochschulen oder Firmen) und auch zum Management von Inventar in Unternehmen eingesetzt [vgl. [JCI14](#), S. 20].

# 3 Analyse und Konzeption

Eine Blockchain Technologie soll gezielt in Geschäftsprozesse der Fachhochschule Aachen integriert werden.

Außerdem soll das zu entwickelnde System an der FH Aachen auch von Studierenden und FH-Mitarbeitern aller Art *potenziell* verwendet werden können, damit es sich als Open-Source *Community Projekt* entfalten kann.

Daher ist es zunächst notwendig, das Einsatzgebiet „Fachhochschule Aachen“ zu analysieren, um daraus die Anforderungen abzuleiten, die für die nachfolgende Konzeption notwendig sind. Ohne die vorherige Analyse des Einsatzgebietes bzw. ohne einen Bezug zur FH Aachen wäre die Aussagekraft der Anforderungen deutlich geringer.

Besonders berücksichtigt werden Aspekte, die für den Geschäftsprozess „Klausur durchführen“ relevant sind, mit einem Schwerpunkt auf IT-Sicherheit. Dieses Prozess-Exemplar wird im Laufe der Arbeit digitalisiert und mithilfe von Blockchain Technologie optimiert.

## 3.1 Analyse: Einsatzgebiet FH Aachen

Die auf moderne Lehre fokussierte FH Aachen hat über 14.000 Studierende (aus mehr als 80 Studiengängen), circa 250 Professorinnen und Professoren zuzüglich 900 Mitarbeiter(innen). Ein essenzielles Ziel sei es, lokal bis global mit anderen Einrichtungen der Forschung und Lehre in Verbindung zu stehen, beispielsweise sowohl durch Kooperation mit internationalen Partnerhochschulen [vgl. [FHA18a](#)].

### 3.1.1 WLAN Netz

Das eduroam-Projekt ermöglicht den WLAN Zugang an der FH Aachen. Eine für das Ermitteln des Mengengerüsts relevante Kennzahl ist die Anzahl gleichzeitiger Geräte

bzw. FH-Nutzer aus der öffentlich zugängigen eduroam-Statistik der Fachhochschule.

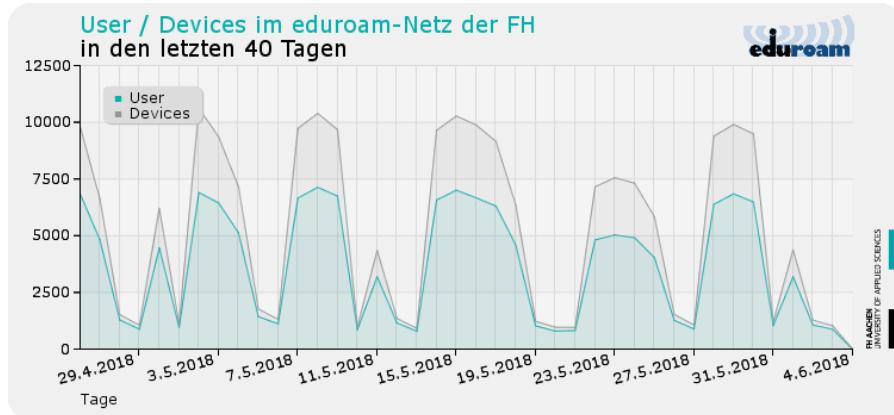


Abbildung 3.1: eduroam-Statistik. Stand: 04. Juni 2018.

QUELLE: [FHA18b]

Wie man aus der Statistik 3.1 ablesen kann, gab es vom 29.04.18 bis zum 04.06.18 zu Hochzeiten durchschnittlich mehr als 10.000 gleichzeitig verbundene Geräte und daneben circa 6.000 parallel angemeldete FH-interne Benutzer. Diese Zahl wird später für die Mengengerüst Anforderung benötigt. Ausgehend davon, dass alle Angehörigen der Fachhochschule sich – sobald sie vor Ort sind – beim eduroam anmelden, so sind (bezogen auf Abschnitt 3.1) maximal rund 40% aller FH-Mitglieder gleichzeitig vor Ort.

#### 3.1.2 Prüfungsverwaltungssystem

Prüfungen können von Studierenden sowohl offline, d.h. lokal beim Prüfungsamt selbst, als auch online, mithilfe des *HIS QIS*, verwaltet werden. Prüfungen können an- und abgemeldet und der Notenspiegel kann eingesehen werden.

##### 3.1.2.1 Sicherheitsanalyse

**Offline:** Noten werden in der Regel zuerst vor dem Prüfungsamt ausgehängt, bevor sie in das *HIS QIS* eingetragen werden. Es ist zu beachten, dass diese Offline-Veröffentlichung nicht *anonym*, sondern *pseudonym* ist; jede Matrikelnummer wird einer Note zugewiesen (Matrikelnummer → Note), d.h. wenn ein potenzieller Angreifer den Realnamen zu einer Matrikelnummer kennt, so ist es auch wahrscheinlich, dass er die Noten des angegriffenen

Studenten in Erfahrung bringen kann (Realname → Matrikelnummer → Note). Dieser Angriff könnte bei jedem Notenaushang beliebig oft wiederholt werden, so dass aus langfristiger Sicht der gesamte Notenspiegel eines oder mehrerer Studierenden anhand öffentlich verfügbarer Daten gesammelt werden kann.<sup>1</sup>

#### \* Beispiel 3.1: IT-Sicherheit – Angriffsphasen

Die *Black-Hat-Hackerin* mit dem Pseudonym „CrackerAlice“ hat das Ziel, alle Noten des Studenten Bob in Erfahrung zu bringen, um ihn damit zu erpressen und Lösegeld zu fordern. Ihr Vorgehen:

**Pre-Attack:** „CrackerAlice“ macht Bob’s Adresse mittels Besuch seiner Internetpräsenz ausfindig (auch *Footprinting* genannt). Sie nutzt die Social Engineering Technik *Dumpster Diving*, durchwühlt die Altpapiertonne und findet schließlich ein Dokument der Fachhochschule mit Bob’s Matrikelnummer.

**Attack:** Die Hackerin schreibt Bob, dass „seine Noten ab jetzt immer veröffentlicht werden“, wenn er nicht „einen Betrag von 2 ETH auf ihren Account sendet“.

**Post-Attack:** Die Angreiferin hat den Betrag erhalten und verwischt ihre Spuren, indem sie die 2 ETH auf eine große Anzahl von Adressen aufteilt, damit sie schwerer nachzuverfolgen ist.

**Online:** Falls ein Angreifer mittels *Identitätsdiebstahl* den Benutzernamen und das Passwort eines Studenten in Erfahrung bringt, so kann er nicht nur wie beim Offline-Angriff die Noten einsehen, sondern auch beispielsweise Prüfungen an- und abmelden.

Die Anmeldung über das *HIS QIS* erfolgt über eine verschlüsselte Verbindung, daher sind die Erfolgsaussichten gering, mittels *Netzwerk-Sniffing* an die Login-Daten zu gelangen.

Es gilt zwar der Grundsatz, dass alle Systeme möglicherweise Sicherheitslücken haben können (Programmierfehler lassen sich nicht ausschließen), aber auch hier ist der Erfolg eines *Social Engineering* Angriffs am wahrscheinlichsten, beispielsweise mithilfe einer *Phishing-Website*.

#### 3.1.2.2 Datenhaltung

Da es sich beim *QIS HIS* um ein aus Entwicklersicht nicht öffentlich dokumentiertes System handelt, folgen in im weiteren Text Annahmen, die sich nicht in Gänze verifizieren lassen (vergleichbar mit einer *Black Box*). Aus technischer Sicht besteht eine hohe

<sup>1</sup>siehe auch [Eck14](#), S. 11 f. für Definition von „Anonymisierung“ und „Pseudonymisierung“.

### 3 Analyse und Konzeption

---

Wahrscheinlichkeit, dass zur Speicherung der Daten eine zentrale Datenbank verwendet wird, auf die das Prüfungsamt direkten Zugriff hat.

Rückschlüsse auf die Datenstruktur und die *Entitäten* der Datenbank lassen sich mit einer genaueren Untersuchung der Benutzeroberfläche ziehen:

Prüfung	Prüfungstext	Semester	Prüfungsdatum	Note	Lstg.Pkte.	Status	Versuch
1000	Abschluss Sem 1+2 IBA	SoSe 16	.2016	60,0	bestanden	1	
8997	Summe Module Gesamt AIT	WiSe 17/18	.2018	150,0	bestanden	1	
51101	Höhere Mathematik 1	WiSe 14/15	.2015	8,0	bestanden	1	
51104	Grundlagen der Informatik und höhere Programmiersprache	WiSe 14/15	.2015	13,0	bestanden	1	

**Abbildung 3.2:** Bildschirmausschnitt des Notenspiegels im *QIS HIS*.

Abb. 3.2 zeigt einen anonymisierten Auszug der Notenspiegel-Ansicht. In der ersten Spalte „Prüfung“ ist der *Primärschlüssel* der Entität *Prüfung* vermerkt. Als Besonderheit ist erkennbar, dass nicht nur Prüfungen im eigentlichen Sinne in der Datenstruktur vermerkt sind, sondern auch die Ansammlung mehrerer Module, bspw. „Summe Module Gesamt AIT“. Dies könnte dafür sprechen, dass es für derartige „Spezialfälle“ keine eigene Entität gibt.

Weiterhin enthält die Tabelle: Das „Semester“ als *Zeichenkette*, das „Prüfungsdatum“ in Form eines *Zeitstempels*, die „Note“ als optionale<sup>2</sup> *Gleitkommazahl* zwischen 1.0 und 5.0, das mit der Note direkt im Bezug stehende Feld „Status“ (wahrscheinlich ein *Enum*), die „Leistungspunkte“ (auch *ECTS* genannt) als *Ganzzahl* und abschließend die Anzahl der Versuche als *Ganzzahl* von 1 bis maximal 3.

#### 3.1.3 Studierendenausweis

An der FH Aachen wird ein elektronischer Studierendenausweis verwendet, auch *FH Karte* genannt. Für den Akteur „Student“ bietet das Kartensystem folgende *Anwendungsfälle* (engl. *Use Cases*):

1. **Identität nachweisen:** Studierende der Fachhochschule Aachen können sich als Student der Einrichtung identifizieren und identifizieren lassen, z.B. von einem FH-Mitarbeiter bei einer Prüfung in Kombination mit einem gültigen Lichtbildausweis. Die Überprüfung erfolgt manuell, das bedeutet nicht unter Zuhilfenahme von Elektronik (nicht digital).

---

<sup>2</sup>Optional, da es bspw. Softskills gibt, deren Prüfungsform nicht benotet wird (z.B. Vorträge).

2. **Zahlung durchführen:** Sowohl in Mensen/Cafeterien als auch in der Bibliothek kann mit der FH Karte bezahlt werden. Zum Ausleihen von Büchern wird ein Passwort benötigt, in der Mensa zum Kauf von Speisen allerdings nicht.
3. **Geld einzahlen:** An speziellen Automaten kann Bargeld eingezahlt werden, damit der Anwendungsfall „Zahlung durchführen“ realisiert werden kann.

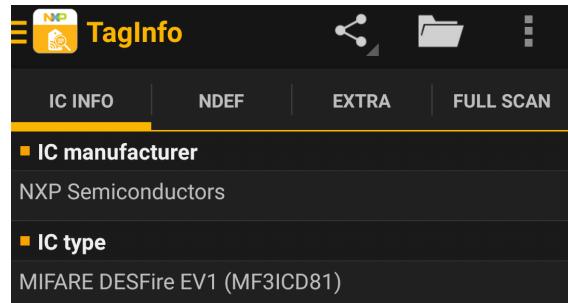
Zusatzinformation: Für Mitarbeiter(innen) der Fachhochschule wurde die FH Karte ebenfalls eingeführt. Sie bietet dem Mitarbeiter die gleichen Anwendungsfälle.

Es lag die Vermutung nahe, dass es sich bei der FH Karte um einen NFC-kompatiblen Tag handeln könnte.

Um dem nachzugehen, wurde die FH Karte mithilfe eines NFC-fähigem Smartphones und der Applikation „NFC TagInfo by NXP“<sup>3</sup> untersucht. Das Ergebnis: Die FH Karte nutzt wie in Abb. 3.3 erkennbar ist tatsächlich NFC. Spezifischer wird ein IC (dt. „integrierter Schaltkreis“) vom Typen *MIFARE DESFire EV1* des Herstellers NXP Semiconductors verwendet.

Diese NFC-kompatible Smart Card verfügt über einen eigenen Mikroprozessor mit Hardware-Sicherheitselement, das eine verschlüsselte Datenübertragung ermöglicht [vgl. NXP15, S. 1 u. 5]. Weiterhin zeigt eine detailliertere Analyse (in der Applikation im „Extra“ oder „Full Scan“ Menü erreichbar), dass die Karte mit dem ISO/IEC 14443A Standard konform ist und einen Speicher von 8 kB hat, der jedoch aus Sicherheitsgründen nicht ohne Weiteres auslesbar ist. Aus der Perspektive der IT-Sicherheit ist ein Identitätsdiebstahl durch das absichtliche Kopieren einer FH Karte daher unwahrscheinlich, denn ein Hacker müsste dazu eine nicht veröffentlichte Sicherheitslücke (engl. *Zero-Day Exploit*) anwenden. Hinzu kommt, dass ein solcher Identitätsdiebstahl limitiert würde, nämlich dadurch, dass Studierende z.B. bei einer Prüfung zum Identitätsnachweis zusätzlich einen Lichtbildausweis vorlegen müssen (s.o., Use Case „Identität nachweisen“).

Nach einer Internetrecherche konnten die Ergebnisse der Analyse durch das Dokument „Rechtliche Hinweise zur FH Karte“ verifiziert werden. Unter anderem wird darin auch



**Abbildung 3.3:** Scan der FH Karte.

---

<sup>3</sup>Separat im Google Play Store erhältlich.

aufgezählt, welche Werte die Karte speichert (z.B. die Seriennummer, Bibliotheksnummer, den Immatrikulationsstatus und Daten zur Bezahlfunktion) [vgl. LK13].

#### 3.1.4 Geschäftsprozess „Klausur durchführen“

Auf der nächsten Seite folgt seitwärts dargestellt der Ist-Geschäftsprozess „Klausur durchführen“, wie er derzeitig an der FH Aachen ausgeübt wird. In diesem Prozess kommen sowohl das *HIS QIS* (siehe Abschnitt 3.1.2) als auch die FH Karte (siehe Abschnitt 3.1.3, Use Case Nr. 1, „Identität nachweisen“) zum Einsatz.

##### Farblegende:

- *Endevents* sind bei Erfolg des Prozesses grün und bei einem Abbruch durch nicht erfüllte Bedingungen rot eingefärbt (d.h. der Prozess konnte nicht den längstmöglichen Pfad durchlaufen).
- Gelborange eingefärbt sind *Tasks*, die Optimierungsbedarf haben.

### 3 Analyse und Konzeption

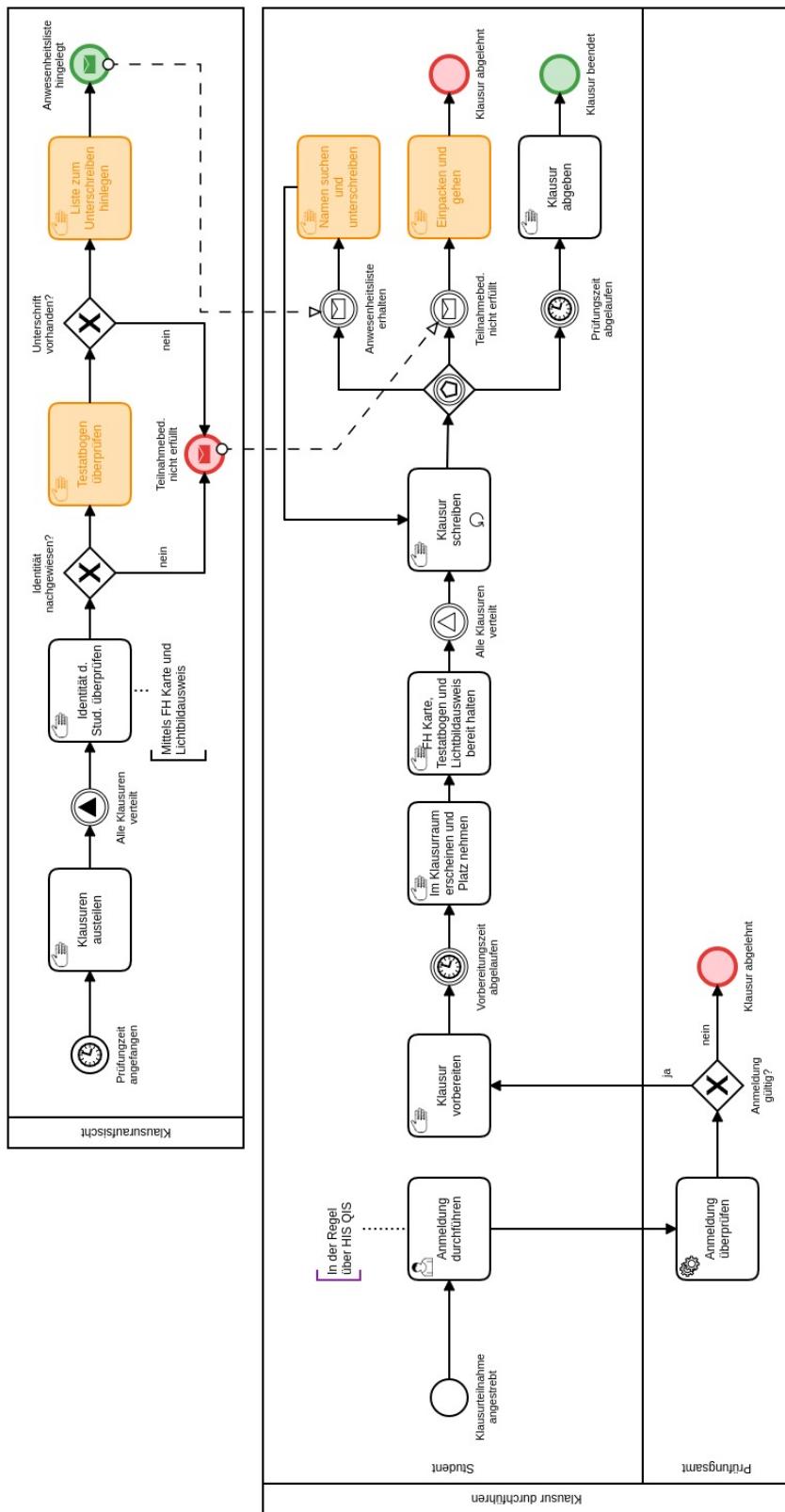


Abbildung 3.4: Der Ist-Geschäftsprozess „Klausur durchführen“.

ERSTELLT VIA: Camunda Modeler v1.11.3 (Linux)

Wie in Abbildung 3.4 dargestellt ist, gibt es drei Teilnehmer (in Lanes eingeordnet, von oben nach unten): *Klausuraufsicht*, *Student* und *Prüfungsamt*. Die Klausuraufsicht arbeitet separat in ihrem eigenen Prozesskontext bzw. *Pool*, während *Student* und *Prüfungsamt* direkt in einem Pool interagieren können. Der Kontrollfluss ist von der Klausuraufsicht zum Studenten *Event-basiert*. Dies ist eine Design-Entscheidung um die *parallele* Ausführung der beiden Prozesse zu verdeutlichen.

#### 3.1.4.1 Prozessüberblick in drei Zonen

1. **Start vom Pool „Klausur durchführen“, aus Sicht des Studierenden:** Der Student meldet sich zunächst mithilfe des *HIS QIS* an. Nachdem er sich *erfolgreich* angemeldet hat, bereitet er die Prüfung vor. Nach Ablauf der Vorbereitungszeit nimmt er im Klausurraum Platz und legt drei wichtige Dokumente bereit: Die *FH Karte*, den *Testatbogen* und einen gültigen *Lichtbildausweis*. Schließlich wartet er darauf, dass alle Klausuren verteilt sind. Die Fortsetzung folgt in Zone 3.
2. **Start vom Pool der „Klausuraufsicht“:** Kurz vor Beginn der Prüfung teilt die Aufsicht alle Klausuren an die Studierenden aus. Dann überprüft sie mithilfe der Dokumente *FH Karte* und *Lichtbildausweis*, ob die Identität des jeweiligen Studenten korrekt ist, wozu folgende Checkliste abgearbeitet wird:
  - Immatrikulationsstatus, Vollständigkeit und Authentizität: Liegen die beiden Dokumente FH Karte und Lichtbildausweis vor? Sind sie gültig?
  - Gleichheit und Integrität: Ist der vollständige Name auf der FH Karte, dem Lichtbildausweis und dem *unterschriebenen* Deckblatt gleich? Gegebenenfalls auch: Stimmt die Matrikelnummer der FH Karte mit dem Deckblatt überein?
  - Identitätsnachweis: Ist die auf dem Lichtbildausweis abgebildete Person der Student, d.h. gibt es eine Übereinstimmung mit dem Bild?

Sofern eine einzige Frage mit „Nein“ beantwortet wird, so wird dem Studenten mitgeteilt, dass er die Teilnahmebedingungen nicht erfüllt. In der Folge wird er angewiesen, den Raum zu verlassen. Andernfalls, wenn alle Fragen mit „Ja“ beantwortet werden, geht es weiter mit der Überprüfung des Testatbogens, auf dem die Unterschrift zu eventuell erforderlichen Praktika vorhanden und der eingetragene Name korrekt sein muss. Falls dies nicht der Fall ist, so hat der Student auch hier keine Zulassung, um an der Prüfung teilzunehmen. Ansonsten bekommt der Stu-

dent eine Anwesenheitsliste hingelegt, auf der unterschrieben werden muss. Der Prozess endet nur in diesem Fall *erfolgreich*.

3. **Fortsetzung von Zone 1:** Die Studenten erhalten, nachdem alle Klausuren von der Aufsicht ausgeteilt wurden, ein Signal, dass sie mit der Bearbeitung anfangen dürfen. Schließlich wird der jeweilige Student, sobald die Klausuraufsicht die Überprüfung vorgenommen hat, benachrichtigt (analog zum Prozessverlauf von Zone 2). Der Student muss dann seine Arbeit unterbrechen. Im positiven Fall erhält er die Liste, sucht seinen Namen darauf und unterschreibt. Danach setzt seine Arbeit fort, bis die Prüfungszeit abgelaufen ist. Im negativen Fall hat der Student die Teilnahmebedingungen nicht erfüllt, muss seine Sachen packen und den Raum verlassen.

#### 3.1.4.2 Nachteile und Verbesserungsvorschläge

Sowohl die textuelle Beschreibung als auch die Grafik zeigen, dass Optimierungsbedarf besteht:

1. **Nachteil:** Lediglich die Prüfungsanmeldung erfolgt digital – alle anderen Tasks im Prozess sind *manuell*. Einige der manuellen Tasks ab Zone 2 sind fehleranfällig und sie erfordern menschliche Arbeit.

**Verbesserung:** Während die Überprüfung der Identität nur schwer<sup>4</sup> automatisiert werden kann, so können allerdings beiden Tasks „Testatbogen überprüfen“ und „Namen suchen und unterschreiben“ digitalisiert werden:

Studierende verfügen bereits über eine FH Karte zum Identitätsnachweis. Es bietet sich nur an, diese auch zum digitalen Signieren von Dokumenten zu verwenden und die Use Cases aus Abschnitt 3.1.3 zu erweitern. Damit könnten sowohl die Überprüfung der Praktikumszulassung (Speichern des Bogens mit Unterschriften in digitaler Form) als auch das Unterschreiben auf der *Anwesenheitsliste* (direkte Erstellung digitaler Signaturen) automatisiert werden. Positiver Nebeneffekt: Der Testatbogen müsste auch nicht mehr mitgebracht werden, so dass dieser auch nicht mehr vom Studenten vergessen werden kann.

2. **Nachteil:** Parallelisierung ist nicht immer von Vorteil – dass die Klausuraufsicht

---

<sup>4</sup>Mit einer intelligenten Bilderkennungssoftware, die mit den Bildern des Studenten angelernt wird, wäre es in der Theorie möglich, jedoch erfordert es dennoch in Einzelfällen (z.B. bei einer fehlerhaften Erkennung) die manuelle Überprüfung, womit der Mehrwert wieder (wesentlich) abnimmt.

parallel zum Schreibvorgang des Studenten Überprüfungen vornimmt, ist hier ein *Störfaktor*. Es ist entscheidend, bei einem Prozess, in dem alle Beteiligten fokussiert sein müssen, derartige Negativ-Faktoren zu eliminieren.

**Verbesserung:** Die Überprüfungen sollten im besten Fall bereits *vor* und nicht *während* der Klausur stattfinden, da Studierende ihre Arbeit sonst unterbrechen müssen. Dazu bedarf es einer generellen Restrukturierung des Prozesses.

3. **Nachteil:** Des Weiteren führt die oben genannte Parallelisierung auch dazu, dass ein weiterer Störfaktor während des Schreibvorgangs auftritt: Ein Student muss seine Sachen einpacken und den Raum verlassen, falls er die Teilnahmebedingungen nicht erfüllt (Task „Einpacken und gehen“). Dies ist zwar nicht der anzunehmende Regelfall, aber dennoch unerwünscht, da es Geräusche verursacht, die es zu vermeiden gilt.

**Verbesserung:** Die Parallelisierung eliminieren (siehe Punkt 2).

## 3.2 Zielsetzung und Anforderungen

**Allgemein:** Es soll ein System auf Basis einer Blockchain Technologie entwickelt werden, das sich nahtlos in die Geschäftsprozesse der FH Aachen einbinden und sich erweitern lässt. Zusätzlich soll es sich u.a. für Lehrzwecke von Studierenden nutzen lassen.

**Speziell (Anwendungsfall des Systems):** Das System soll dazu in der Lage sein, den Geschäftsprozess „Klausur durchführen“ zu optimieren, indem die drei Ideen zur Verbesserung in Abschnitt 3.1.4.2 vollständig umgesetzt werden. Anhand dieses Spezialfalls lässt sich die Leistungsfähigkeit des Systems in der Praxis testen und messen; es ist dadurch „greifbarer“ und weniger abstrakt.

Das *FH-interne* System richtet sich an zwei Zielgruppen, im Folgenden auch generalisiert *System-Benutzer* genannt:

1. **Endanwender bzw. Endbenutzer:** *Teilnehmer* der in dieser Arbeit enthaltenen Geschäftsprozesse sind auch mit Endanwendern gleichzusetzen. Zudem wird jede Person, die ein *Frontend* verwendet, als Endanwender bezeichnet. Teilnehmer können sowohl Studierende als auch FH-Mitarbeiter(innen) sein.

**2. Entwickler:** Diese Zielgruppe soll die Möglichkeit haben, sich am Ausbau und an der Verbesserung des Systems zu beteiligen. Entwickler können sowohl FH-Mitarbeiter(innen) als auch Studierende (z.B. zu Lehrzwecken) sein.

In den nächsten Abschnitten werden die *allgemeinen* Anforderungen beschrieben, damit das System in der Lage ist, die obenstehenden Ziele zu erfüllen.

#### 3.2.1 Notwendige Anforderungen

##### 3.2.1.1 Sicherheit

Das System muss das gleiche Level an Sicherheit bieten wie bereits existierende Lösungen der Fachhochschule. Die Analyse des bereits existierenden Prüfungsverwaltungssystems in Abschnitt 3.1.2.1 hat gezeigt, dass die Notenbekanntgabe nicht anonym erfolgt, sondern pseudonym, da die Studenten mit ihrer Matrikelnummer einer Note zugewiesen werden. Diese *Pseudonymität* muss mindestens gewahrt werden, besser wäre jedoch noch eine Hochstufung auf *Anonymität* (optional).

Die allgemeinen Sicherheitsanforderungen lassen sich mit den häufig aufkommenden *IT-SchutzzieLEN* äußerst prägnant zusammenfassen [vgl. auch Eck14, S. 7 - 13]:

Schutzziel	Bedeutung	Realisierung
<b>Vertraulichkeit</b>	Schutz vertraulicher Informationen vor Unbefugten	Lese-Limitation
<b>Integrität</b>	Schutz vor Datenmanipulation durch Unbefugte	Schreib-Limitation
<b>Verfügbarkeit</b>	Gewährleistung ständiger Verfügbarkeit f. Befugte	DoS-Schutz
<b>Authentizität</b>	Sicherstellung der Echtheit von Informationen <sup>5</sup>	Authentifikation <sup>6</sup>
<b>Verbindlichkeit</b>	Nicht-Abstreitbarkeit autorisierter Aktionen	Digitale Signaturen

**Tabelle 3.1:** Fünf bekannte IT-Schutzziele inkl. der ersten drei Grundziele.

In der Tabelle 3.1 sind *Befugte* mit „autorisierten Personen“ bzw. „Subjekten mit ausreichenden Rechten“ gleichzusetzen (das Gegenteil: *Unbefugte*).

Die drei ersten Schutzziele spiegeln die im englischsprachigen Raum bekannte **CIA-Triade** (**C**onfidentiality, **I**ntegrity und **A**vailability) wider, ein IT-Sicherheitsmodell, das speziell in Organisationen Anwendung findet und sich daher auch sehr gut für die FH Aachen eignet [vgl. Rou14].

Während die ersten drei Schutzziele eher unabhängig voneinander zu betrachten sind,

<sup>5</sup>Informationen können auch Personen sein, die als solche abgespeichert sind, etwa in einer Datenbank.

<sup>6</sup>Zum Beispiel durch ein *User Account-System*

so nehmen die beiden erweiterten letzten Schutzziele eine gesonderte Rolle ein: Die *Authentizität* ist meist eine Voraussetzung für die *Verbindlichkeit*. Wenn eine Person als echt identifiziert wurde, so ist dies auch der erste Schritt zur *Verbindlichkeit*.

Diese ist besonders für rechtliche Angelegenheiten relevant, da etwa mit der Realisierung einer *digitalen Signatur* nicht abgestritten werden kann, dass eine bestimmte (autorisierte) Aktion durchgeführt wurde.

#### **Exkurs: Rechtlicher Kontext zu digitalen Signaturen (Verbindlichkeit)**

Mitte 2016 hat sich die europäische Verordnung namens *eIDAS* für *elektronische Signaturen*<sup>a</sup> durchgesetzt [vgl. [Bor16](#)]. Diese bestimmt, welche Anforderungen die digitale Signatur erfüllen muss. Unter den drei verschiedenen Arten gilt nur die *qualifizierte elektronische Signatur* als Äquivalent zur handschriftlichen Signatur (auch vor Gericht). Das Zertifikat zu dieser Signatur-Art muss an die Person von einem sogenannten *Vertrauensdiensteanbieter* ausgestellt werden, der bestimmte Auflagen erfüllen muss [vgl. [Doc18](#)].

**Anwendung auf die FH Aachen:** Es wäre mit einem großen Aufwand verbunden, wenn die Fachhochschule extra zu einer Zertifizierungsstelle werden müsste, um das (digitale) Schutzziel der *Verbindlichkeit* zu erfüllen. Dies ist jedoch nicht notwendig, da spätestens seit *eIDAS* auch die *fortgeschrittene digitale Signatur* als Beweismittel vor Gericht zugelassen ist und nicht abgelehnt werden darf – auch wenn sie *nicht* die Rechtswirksamkeit der handschriftlichen Signatur ersetzt [vgl. [Uni14](#)]. Empfohlen wird diese Art bei Dokumenttypen, die mit „mittleren rechtlichen Risiken“ verbunden sind [[Doc18](#)]. Im späteren Verlauf der Arbeit wird erklärt, welche Aspekte realisiert werden müssen, um fortgeschrittene digitalen Signaturen an der FH Aachen einzusetzen.<sup>b</sup>

<sup>a</sup>In der Verordnung wird statt *digital* durchgehend der Begriff *elektronisch* verwendet.

<sup>b</sup>siehe auch [Uni14](#), Art. 26 (für Leser, die sich die Anforderungen bereits durchlesen möchten.)

#### **3.2.1.2 Mengengerüst, Skalierbarkeit und Effizienz**

Anwendung auf die Analyse des Einsatzgebietes (siehe Abschnitt [3.1](#)):

1. **Mengengerüst:** Das System muss *belastbar* genug sein.

Unter der Schätzung, dass zu Hochzeiten 50% aller eduroam Nutzer das System gleichzeitig nutzen wollen (siehe Abschnitt [3.1.1](#)), so muss es 3000 Nutzern gleichzeitig die volle Systemfunktionalität bieten.

2. **Skalierbarkeit:** Weiterhin muss das System *horizontal skalierbar* sein.

Zum Beispiel, falls es in Zukunft auch an den Partnerhochschulen (siehe Ab-

schnitt 3.1) eingesetzt oder die FH weiter ausgebaut würde. Horizontal skalierbar heißt, dass das System leistungsfähiger wird, indem neue Knoten zum (internen) Netzwerk hinzugefügt werden. Dies impliziert, dass neue Knoten (Rechner) nicht zu teuer in der Anschaffung sein dürfen; daraus resultiert schließlich die nächste Anforderung.

3. **Effizienz:** Das System muss so *effizient* arbeiten, dass es auch auf vergleichsweise kostengünstigerer Hardware ausgeführt werden kann. Nur dadurch kann die Umsetzung der zweiten Anforderung der *horizontalen Skalierbarkeit* gewährleistet werden.

#### 3.2.1.3 Erweiterbarkeit

Entwickler sollen allgemein die Möglichkeit haben, sich auf möglichst einfache Art und Weise an der Weiterentwicklung des Systems zu beteiligen.

Zielführend ist es daher, dass sich der Programmcode an etablierte Konventionen hält (Stichwort „Clean Code“) und auf bewährte Paradigmen der objektorientierten Programmierung (wie *Datenkapselung*, *Vererbung*, *Abstraktion* und *Design-Patterns*) zurückgriffen wird.

Zudem sollte die Konfiguration der Komponenten stets über spezielle Konfigurationsdateien erfolgen, so dass der Quelltext ausnahmslos abgeschottet von der Konfiguration ist – in anderen Worten: Keine hartkodierten Konfigurationsparameter.

#### 3.2.1.4 Verwendung offener Standards

Alle einzusetzenden Hardware-Lösungen müssen auf offenen Standards basieren, die ausführlich dokumentiert sind. Alle einzusetzenden Softwaretechnologien müssen Open-Source sein.

Diese Anforderung trägt zur Umsetzung der Erweiterbarkeit (siehe 3.2.1.3) bei, da gut dokumentierte Open-Source Software diese Eigenschaft bereits mitbringt. Zudem können sowohl bei der Verwendung von Standards als auch bei Open-Source Produkten Sicherheitslücken leichter aufgedeckt werden und das Risiko von Zero-Day Exploits nimmt in der Regel ab (siehe 3.2.1.1).

## 3.2.2 Empfohlene/Optionale Anforderungen

### 3.2.2.1 Verwendung bestehender Standards

Damit eine *reale* Integration des Systems in die FH Aachen zu einem späteren Zeitpunkt stattfinden könnte, wäre es hilfreich, wenn auf die bereits bestehenden technologischen Standards des Einsatzgebietes „Fachhochschule Aachen“ (siehe Abschnitt 3.1) zurückgegriffen oder sich an bereits vorhandenen Projekten orientiert wird.

So hat eine (Black Box-) Analyse des Prüfungsverwaltungssystems *QIS HIS* im Abschnitt 3.1.2.2 Rückschlüsse über die wahrscheinlich eingesetzte Datenhaltung erlaubt, an die sich (z.B. bei der Implementation des „Klausur durchführen“-Geschäftsprozesses) orientiert werden sollte.

Weiterhin hat eine Untersuchung der FH Karte in Abschnitt 3.1.3 gezeigt, dass der NFC-kompatible Standard *ISO/IEC 14443A* verwendet wird. Nach Möglichkeit sollte dieser weiterverwendet werden.

### 3.2.2.2 Einfachheit der Bedienung

Das System sollte möglichst einfach und intuitiv – im besten Fall auch ohne ein Handbuch – zu bedienen sein.

Endbenutzer sollte nicht wissen müssen, wie sie die Technologien des Systems bedienen; im Optimalfall sollte beispielsweise gar nicht erst offensichtlich sein, dass eine Blockchain Technologie zum Einsatz kommt. Aufgrund ihrer Komplexität muss sichergestellt werden, dass sie ausreichend *abstrahiert* wird.

Daher ist es hilfreich, die zu verwendende Blockchain Lösung als ein *Backend* zu kategorisieren und die Anwendung, das vom Endbenutzer gesehen und bedient wird, als ein *Frontend*.

#### \* Beispiel 3.2: Vergleich mit Betriebssystem

Bei einem Betriebssystem muss der Endbenutzer zwar wissen, wie er Anwendungen startet und bedient. Allerdings liegt es nicht in seiner Verantwortung, zu verstehen, wie der Maschinencode des Programms vom Prozessor ausgeführt wird (vgl. auch mit Ähnlichkeiten zur Ethereum Virtual Machine in Abschnitt 2.1.4.4).

#### 3.2.2.3 Dokumentation

Es sollte eine verständliche, nach Möglichkeit englischsprachige *Systemdokumentation* geschrieben werden, damit das System weiterentwickelt werden kann (wie in Abschnitt 3.2.1.3 beschrieben, Anforderung der „Erweiterbarkeit“). Eine mögliche *Projektdokumentation* liegt in Kapitel 4 vor. Letzten Endes ist es ein Ziel der vorliegenden Arbeit, auch das allgemeine Vorgehen zu beschreiben, womit sie auch zum Teil als eine *Prozessdokumentation* angesehen werden kann.

#### 3.2.2.4 Funktional: Interne Währung

Im Abschnitt 3.1.3 der FH Karte wird der Use Case „Zahlung durchführen“ erklärt. Es würde sich eignen, wenn diese Funktionalität auch von einer Blockchain Lösung abgedeckt würde. Die notwendige Bedingung dafür ist, dass eine eigene FH-Aachen-interne Währung eingeführt werden müsste (etwa ein Token namens „FHACoin“). Damit die Anforderung „Einfachheit der Bedienung“ weiterhin abgedeckt ist, sollten alle Frontends stets den Eurokurs anzeigen, damit ständige Konvertierungen entfallen (die für den Endbenutzer aufwendig sind).

## 3.3 Konzeption: Wahl der Technologien

In diesem Abschnitt wird anhand der Anforderungen konzipiert, welche Technologien bei dem System zum Einsatz kommen. Dabei ist die Reihenfolge die gleiche wie in Kapitel 2, also nach Relevanz absteigend sortiert.

### 3.3.1 Blockchains im Vergleich

#### 3.3.1.1 Auswahlkriterien

Es wurde eine Blockchain Technologie gesucht, die nach Möglichkeit alle in Abschnitt 3.2 notwendigen Anforderungen abdeckt (mit besonderem Augenmerk auf die in Anforderung **Sicherheit** erklärten Schutzziele). Zusätzlich muss es in der Lage sein, *Turing-vollständige* Smart Contracts auszuführen.

Das Attribut wird an dieser Stelle hervorgehoben, da es etwa mit *Bitcoin Script* eine

### 3 Analyse und Konzeption

Möglichkeit gibt, Transaktions-basierte Skripte mit der populären *Bitcoin Blockchain* auszuführen, die jedoch nicht Turing-vollständig sind. Die daraus resultierenden *dezentralisierten Applikationen* unterstützen keine Schleifen [vgl. Bit18a]. Diese Limitation sollte die gesuchte Technologie nicht haben, da es die Entwicklung stark einschränkt; Wiederholungen entstehen bei Bitcoin Skript durch die Wiederholung von *Instruktionen*, wodurch die Anforderung **Erweiterbarkeit** ab einem bestimmten Komplexitätsgrad nicht mehr ohne viel Aufwand möglich ist – die Wartbarkeit der Programme würde schwierig.

Interessanterweise erfüllen Blockchain Technologien allgemein 4 der 5 Schutzziele bereits auf natürliche Weise (siehe 3.2.1.1):

Schutzziel	Realisierung	Ref.-Abschnitt
<b>Vertraulichkeit</b>	–	–
<b>Integrität</b>	Unveränderlichkeit d. Blöcke ⊃ Transaktionen	2.1, 2.1.3
<b>Verfügbarkeit</b>	DoS-Schutz durch (redundante) Knoten <sup>7</sup>	2.1.1, 2.1.3
<b>Authentizität</b>	Benutzer haben <i>Accounts</i> mit <i>Private Keys...</i>	2.1.2, 2.1.4.2
<b>Verbindlichkeit</b>	↳ für Transaktions-basierte <i>digitale Signaturen</i> .	2.1.2, 2.1.4.3

**Tabelle 3.2:** Anwendung der Schutzziele aus 3.1 auf die Blockchain.

Tabelle 3.2 legt jeweils das Schutzziel mit der Realisierung und einem Verweis (Referenz) auf den Abschnitt, der die Realisierung im Kapitel 2 beschreibt, dar. Das Schutzziel der **Vertraulichkeit** ist das Einzige, das nicht durch die allgemeingültigen Grundsätze einer Blockchain abgedeckt wird. Aufgrund dessen wird zwischen zwei Arten von Blockchains unterschieden [siehe auch But15]:

#### Definition 3.1: Blockchain Arten – *private* und *public*

**Private (permissioned):** Nur ausgewählte Teilnehmer dürfen auf das Netzwerk auf eine bestimmte Art und Weise zugreifen; so kann etwa festgelegt werden, wer *lesend* auf eine Ressource (z.B. einen Smart Contract) zugreifen darf [vgl. jpm18].

**Public (permissionless):** Jeder Netzwerkteilnehmer kann sowohl *schreibend* als auch *lesend* auf die Blockchain zugreifen – d.h. jeder kann Mining ausüben,<sup>a</sup>, Transaktionen senden und bereits stattgefundene Transaktionen nachvollziehen.

<sup>a</sup>Oder genereller: Jeder Netzwerkteilnehmer kann an der Konsensfindung teilnehmen, z.B. im Hinblick auf *Proof of Stake*, siehe auch Abschnitt 2.1.4.5 und vgl. mit Abschnitt 2.1.3

<sup>7</sup>Knoten synchronisieren sich und haben jeweils eine Kopie der Blockchain.

Vertraulichkeit kann demzufolge durch eine *private* Blockchain erreicht werden. Diese Blockchain-Art findet typischerweise im Business-Bereich Anwendung. In Unternehmen können auch die Mechanismen zur Konsensfindung abweichen oder angepasst werden: Statt generell keinem *einzelnen* Knoten zu vertrauen wie es bei der *public* Blockchain der Fall ist, so lässt sich dort die Anzahl Transaktions-validierende Knoten einschränken – Knoten, denen generell (mehr) vertraut werden kann, da sie z.B. nur Organisationsintern eingesetzt werden. Das erhöhte Vertrauen gegenüber mancher Knoten resultiert auf der positiven Seite in einer höheren Performance, jedoch zwangsläufig auf der negativen Seite wieder automatisch in einer erhöhten *Zentralisierung*. Die größte mögliche Dezentralisierung kann daher nur mit einer transparenten *public* Blockchain erreicht werden, die jedoch das Schutzziel der Vertraulichkeit nicht direkt unterstützt [vgl. But15].

#### \* Beispiel 3.3: Rückblick – Schwerpunkt einer Blockchain

Die oben genannte Problematik wurde bereits indirekt in Abschnitt 2.1.3 bei der Erklärung des Block-Übergangsalgorithmus in Pseudocode vorweggenommen:

Der Algorithmus ist „[...] ein Fundament des allgemeinen Blockchain-Modells, das je nach Schwerpunkt angepasst und erweitert werden kann.“ (siehe S. 11)

In der folgenden Sektion werden aus Gründen, die sich später herausstellen werden, sowohl *private permissioned* als auch *public permissionless* Blockchain Technologien betrachtet.

#### 3.3.1.2 Vergleich

Bei der Recherche nach Blockchain Technologien, die Smart Contracts für dezentralisierte Applikationen unterstützen, ist aufgefallen, dass es insbesondere auf Seite der *public Blockchains* zwar viele Versprechen gibt, diese jedoch nicht immer schon umgesetzt sind – oder gar produktiv einsetzbar sind. Daher wurde zusätzlich zu den oben genannten Kriterien beschlossen, nur Blockchain Technologien einzubeziehen, die mindestens ein halbes Jahr vor dem Beginn der Bachelorarbeit bereits eine stabile lauffähige Version haben. Das heißt, dass nach Technologien gefiltert wurde, die spätestens am 5. Oktober 2017 über eine als *stable* gekennzeichnete Version verfügten. Auf diese Weise wird verhindert, dass fertige Lösungen mit „halbfertigen“ Lösungen verglichen werden (vorausgesetzt, Entwickler wenden etwa das *Semantic Versioning* zu Genüge an)<sup>8</sup>. Es wurden

<sup>8</sup>siehe auch Pre18.

drei entsprechende Technologien für dezentralisierte Applikationen ermittelt [vgl. VS17]: *Hyperledger Fabric*, *Ethereum* und *Corda*.

**Corda:** Nach genauer Studie des technischen Whitepapers von *Corda* hat sich herausgestellt, dass es sich dabei *nicht* um eine Lösung handelt, die eine Blockchain verwendet [vgl. Hea16, S. 5]. Stattdessen ist es eine *dezentralisierte Datenbank* mit SQL-Zugriffsmöglichkeiten. Code ist zwar durch Bitcoin und Ethereum inspiriert, aber das Projekt geht anderen Konzepten nach [vgl. Hea16, S. 5 - 36]. Diese Konzepte wurden speziell auf Finanzapplikationen ausgerichtet; zum Beispiel gibt es „Notar“-Services (i.d.R. in geringer Zahl), die Transaktionen verifizieren und finalisieren [vgl. Hea16, S. 29 ff.]. Wie auch bei *public* und *private* Blockchains (siehe Def. 3.1) gilt hier die gleiche Problematik: Eine vorher bekannte Anzahl von Notaren kann zwar sehr performant Transaktionen unterschreiben ( $\Rightarrow$  Skalierbarkeit), jedoch wird das System dadurch auch wieder zentralisierter – möglicherweise auch unsicherer, da nur eine kleine Menge von Knoten (*Notaren*) kompromittiert werden muss, um beispielsweise den in Beispiel 2.1 beschriebenen Angriff des *Double-spendings* durchzuführen. Aufgrund des geringen Grads der Dezentralisierung und der zum Teil stark abweichenden Grundlagen (*Corda* ist nicht als Blockchain-Lösung zu klassifizieren) scheidet die Technologie als möglicher Kandidat aus. Übrig bleiben damit *Hyperledger Fabric* und *Ethereum*.

**Hyperledger Fabric versus Ethereum:** Für einen Überblick der beiden Kandidaten wurden Daten aus verschiedenen Quellen zusammengetragen [vgl. auch VS17]:

Aspekt	Hyperledger Fabric	Ethereum
Organisation (Gründung)	Linux Foundation (2000)	Ethereum Foundation (2014)
Scope	Cross-Industry	Globale Dezentralisierung
Fokus	Modularisierung	Generalisierung
Art	<i>Private</i> permissioned	<i>Public</i> permissionless
Konsensfindung	Diverse, auswählbar	Proof of Work (Mining)
Smart Contracts	Ja, z.B. in Go, Java	Ja, z.B. in Solidity, Viper
Währung	–	Ether & Gas-Mechanismus
Erste Produktiv-Version <sup>9</sup>	11.07.2017	14.03.2016
Enterprise Allianz <sup>10</sup>	190+	500+
GitHub Repositories <sup>11</sup>	2.039	14.989
StackExchange Threads <sup>12</sup>	4.772	16.360

**Tabelle 3.3:** Gegenüberstellung: Hyperledger Fabric und Ethereum

Die Tabelle 3.3 ist in drei Sektionen unterteilt, von oben nach unten: Allgemeine Informationen, Features und Statistiken.

Hyperledger ist ein Projekt der Linux Foundation, das aus mehreren autonom agierenden Blockchain-Frameworks besteht: Eines davon, *Hyperledger Fabric* (kurz *Fabric*), wurde am 11. Juli 2017 in Version 1.0.0 veröffentlicht, die als *stable* und „produktionsfähige“ *private* Blockchain gekennzeichnet wurde. Die Ethereum Foundation hat bereits Anfang 2016 mit „Homestead“ eine stabile Version veröffentlicht. Da Ethereum eine *public* Blockchain ist, kam diese auch direkt global zum Einsatz und der Client konnte von einer Vielzahl von Teilnehmern rund um den Globus erfolgreich öffentlich ausgeführt werden.

Bei Fabric liegt der Software-architektonische Fokus auf Modularisierung: Die Komponenten sind so designed, dass sie „plug-and-play“ ausgetauscht werden können, wie unter anderem auch der Mechanismus zur Konsensfindung [vgl. Hyp18b].

Bei Ethereum ist die Konsensfindung hingegen eher fest in den Clients verdrahtet: Der *Proof of Work*-Mechanismus (Mining) kann nur mit einer Änderung des Protokolls – einem Fork – erreicht werden (siehe Ende von Abschnitt 2.1.4.1). Dafür ist die Plattform jedoch im hohen Maße generisch implementiert und erweiterbar.

Während Fabric für Smart Contracts (bei Hyperledger *Chaincode* genannt) auch bekannte Sprachen wie Go und Java unterstützt [vgl. VS17], so gibt es bei Ethereum speziellere Programmiersprachen wie *Solidity* und *Viper*, die auf den Anwendungsfall der Smart Contracts spezialisiert sind.

Ethereum benötigt als public Blockchain im Gegensatz zu Fabric eine intrinsische Währung. *Ether* wird unter anderem dazu eingesetzt, um das Schutzziel der *Verfügbarkeit* umsetzen zu können, denn der Gas-Mechanismus ist ein natürlicher DoS-Schutz (siehe Abschnitt 2.1.4.4, mittig).

Der unterste Tabellenteil spiegelt Indikatoren zur Aktivität der Entwicklung wider, die insbesondere ein Maßstab für die Anforderung *Erweiterbarkeit* sind.

Sowohl das Hyperledger Projekt als auch Ethereum haben eine Enterprise Allianz gebildet; einen Zusammenschluss mehrerer – zum Teil äußerst bekannter – Unternehmen aus verschiedensten Bereichen, die die jeweilige Technologie testen und verbessern. Bei-

<sup>9</sup>Hyperledger: [vgl. Hyp18a, S. v1.0.0] u. [vgl. Hyp18b, S. 15], Ethereum: [vgl. Eth16b, „Homestead“]

<sup>10</sup>[vgl. Hyp18b, Stand: Februar 2018] und [vgl. Eth18b, Stand: Mai 2018]

<sup>11</sup>„Hyperledger“ [vgl. Git18b, Stand: 06.06.2018], „Ethereum“ [vgl. Git18a, Stand: 06.06.2018]

<sup>12</sup>„Hyperledger“ [vgl. Sta18b, Stand: 06.06.2018], „Ethereum“ [vgl. Sta18a, Stand: 06.06.2018]

de Projekte haben somit auch eine Ausrichtung auf die Industrie, auch wenn diese bei Hyperledger stärker ausfällt. Hier wurde von Anfang an die Absicht verfolgt, branchenübergreifend (*Cross-Industry*) und kooperativ über Firmen verteilt eine Blockchain zu verwenden [vgl. Hyp18b]. Gerade deshalb ist die Zahl der *Enterprise Ethereum Alliance* Mitglieder beachtlich, die mit 500 Mitgliedern mehr als doppelt groß ist wie der Zusammenschluss von Unternehmen bei Hyperledger.<sup>13</sup>

Ohne Entwickler für dezentrale Applikationen kann eine Blockchain Plattform nicht bestehen. Die Entwicklungsaktivität ist daher ein nicht zu unterschätzender ausschlaggebender Faktor. Zur Analyse dieser Aktivität wurde folgender Ansatz verfolgt: Es wurde bei *GitHub*, einem der größten Hosting-Anbieter für *Git*-Projekte und bei *StackExchange*, einer populären Plattform, die u.a. für Fragestellungen rund um Softwareentwicklung genutzt wird, jeweils nach „Hyperledger“ und nach „Ethereum“ gesucht. Die Suche hat ergeben, dass es bei GitHub mit rund 15.000 Projekten die 7.5-fache Menge an Repositories gibt, die „Ethereum“ erwähnen (was häufig auch ein Zeichen dafür ist, dass das Projekt auf der Ethereum Technologie basiert oder etwa eine dezentralisierte Applikation für Ethereum ist). Bei StackExchange sind es rund 3.5 mal mehr Fragestellungen für „Ethereum“-bezogene Probleme als bei „Hyperledger“. Die Entwickler-Community scheint demzufolge öffentlich bedeutend mehr Ethereum- als Hyperledger-Projekte zu publizieren, jedoch verhältnismäßig weniger Fragen über die Technologie zu stellen.

#### Beispiel 3.4: Zusatzinterpretation

Dass es nicht auch die rund 7.5-fache Menge an Fragen zu Ethereum gibt, kann auch anderweitig positiv für Ethereum ausgelegt werden, da es bspw. für eine ausführlichere oder verständlichere Dokumentation oder gegen die Einfachheit von Hyperledger sprechen könnte, wenn das Verhältnis zwischen GitHub (GH) Repositories und StackExchange (SE) Fragen wie folgt betrachtet wird:

$$\frac{\text{Entwicklung}_{\text{GH}}}{\text{Frageanzahl}_{\text{SE}}} = \text{Effizienz}$$

Bei Ethereum ist der Quotient etwa 0.9, bei Hyperledger (Fabric) nur rund 0.4. Je höher der Wert, desto mehr Produktivität wird bei der Entwicklung erwartet.

Es bleibt zu sagen, dass Ethereum sichtlich den Vorreiter-Status trägt, was sich bspw. auch darin bemerkbar macht, dass das Hyperledger Projekt auch ein Framework namens

---

<sup>13</sup>Die Hyperledger- und Ethereum-Allianz-Mitglieder überschneiden sich zu einem gewissen Anteil.

Burrow umfasst, das die Ethereum Virtual Machine (zu einem bestimmten Anteil) nach der Spezifikation implementiert [vgl. Hyp18b].

#### 3.3.1.3 Ethereum

Die Entscheidung ist auf Ethereum gefallen, da überlegt wurde, das einzige Problem, das gegen die Technologie gesprochen hat, anders anzugehen: Die *Vertraulichkeit*. Sie kann, wie aus Anforderung **Sicherheit** hervorgeht, vorerst auch durch *Pseudonymisierung* erfolgen. Die Idee ist daher gewesen, im Rahmen dieser Arbeit nicht alle (auch vertraulichen Daten) in der Blockchain zu speichern, da dies ohnehin bezüglich des Speicherbedarfs ineffizient wäre (Anforderung **Mengengerüst, Skalierbarkeit und Effizienz**). Stattdessen werden nur Informationen in der Blockchain abgespeichert, die für die Ausführung der Smart Contracts notwendig sind (z.B. allgemein IDs bzw. Primär- und Fremdschlüssel oder spezieller: Matrikelnummern von Studierenden) – letztendlich sind diese Daten als *Metadaten* zu typisieren, da sie aus Sicht des Endanwenders Daten sind, die andere Daten beschreiben, jedoch für ihn eine geringe(re) Aussagekraft haben. Alle für den Endanwender relevanten „menschlich lesbaren“ und vertraulichen Daten werden außerhalb der Blockchain persistiert und etwa mithilfe eines REST-Webservices abgerufen (z.B. der reale Name eines Studenten oder der Name eines Moduls).

Dennoch ist mit Ethereum das vollständige Erreichen des Schutzzieles der Vertraulichkeit mittels *Anonymisierung* nicht ausgeschlossen. Es gibt – dank der Generalisierung Ethernets, die zu einer verbesserten Erweiterbarkeit führt – bereits eine Spezialisierung Ethernets namens *Quorum*, die exakt dieser Problematik angeht: Mit *Quorum* können Leserechte auf Smart Contract Ebene eingeschränkt werden, damit nur bestimmte Parteien auf sie zugreifen können.<sup>14</sup> Zudem ist es möglich (wie auch bei Hyperledger) andere effizientere Mechanismen zur Konsensfindung zu nutzen [vgl. jpm18].

Auf diese Weise könnte, falls an der FH Aachen notwendig, eine spätere *Migration* zu Quorum erfolgen, ohne dass die bereits geschriebenen Smart Contracts verworfen werden müssen. Bis zu diesem Zeitpunkt sollte jedoch zuerst versucht werden, den oben beschriebenen Ansatz der Pseudonymisierung zu verwenden, der letztendlich zu einer höheren Transparenz hinsichtlich der Geschäftslogik führt. Dadurch, dass die Geschäftslogik bereits in Smart Contracts öffentlich dokumentiert ist, könnten einige Abläufe an

---

<sup>14</sup>Die Limitierung von *Schreibrechten* bei Smart Contracts ist auch ohne Quorum möglich, nämlich mithilfe des Codes selbst; doch dazu später mehr in Kapitel 4.

der Fachhochschule schließlich auch – zumindest für die Zielgruppe der Entwickler – deutlicher werden.

Es sei noch anzumerken, dass das Ethereum Netzwerk ausschließlich FH-intern aufgebaut werden muss, so dass dritte Parteien von außerhalb keinen Zugriff auf die Blockchain haben. Zum Abschluss ist die Erfüllung der Anforderung [Funktional: Interne Währung](#) mit Ethereum – ohne eine eigene Implementation eines Tokens (d.h. ohne Mehraufwand) – möglich, da mit *Ether* bereits eine Blockchain-interne Währung existiert.

#### 3.3.2 Camunda

Eine Geschäftsprozessmanagement-Software wurde gesucht, damit die drei Phasen *Modellieren*, *Ausführen* und *Verbessern* (wie in Abschnitt [2.2](#) beschrieben) für den exemplarischen Prozess „Klausur durchführen“ durchlaufen werden können. Wichtig war dabei, dass es sich um eine erweiterbare (Anforderung [Erweiterbarkeit](#)) Open-Source Engine handelt, die einen möglichst hohen Anteil der *BPMN*- und *DMN*-Standards umsetzt (Anforderung [Verwendung offener Standards](#)).

Beim Vergleich der drei Open-Source Engines *jBPM*, *Activiti* und *Camunda BPM* (ein Fork von *Activiti*) führt letztere bezüglich der *BPMN*-Konformität. Keine der Engines ist vollständig *BPMN*-konform, aber mit einer Feature-Abdeckung von 63% ist Camunda vor seinem Stammpunkt *Activiti* (56%) positioniert, da mehr *Event*-Typen gemäß Standard realisiert werden: Zum Beispiel das *Link-Event* oder die Anzahl der umgesetzten Funktionen des *Timer-Events* [vgl. [Gei+15](#)].

Von Camunda wird der *DMN* Standard komplett abgedeckt, womit damit nach Definition der OMG [vgl. [OMG16](#)] das sogenannte „Conformance Level 3“ erfüllt wird [vgl. [DMN18](#)].

Abschließend wurde Camunda auch aufgrund der modularen Struktur ausgewählt, wie sie in Abschnitt [2.2](#) beschrieben wird, was für die Anforderung [Erweiterbarkeit](#) von Vorteil ist. Im Endeffekt kommt auch die Leichtgewichtigkeit der performanten Engine der Anforderung [Mengengerüst, Skalierbarkeit und Effizienz](#) zugute (siehe Anfang Abschnitt [2.2.2](#)).

#### 3.3.3 Vagrant

Es wurde eine Software gesucht, mit der Entwickler eine vollständige Systemumgebung aufsetzen können, *ohne* dabei eine Anleitung manuell zu befolgen. Dies dient der Anforderung **Erweiterbarkeit**, da auch die Einfachheit der Entwicklung signifikant zunimmt – schließlich soll es auch mehr Interessierte dazu bringen, das Projekt auszuprobieren. Eine mehrere Seiten lange manuelle Installationsanleitung wirkt abschreckend und ist fehleranfällig(er). Wenn hingegen nur wenige Befehle ausgeführt werden müssen, ist zu erwarten, dass sich mehr Entwickler beteiligen, wodurch das Ziel des Community-Projekts einen Schritt näher gebracht wird.

Die Wahl ist auf *Vagrant* gefallen, um die Installationsanleitungs-Automatisierung in einer *VM* mit nur einem einzigen Befehl zu erreichen. Der VM-Aspekt ist wichtig, um die mögliche Varianz unter den Systemen zu eliminieren (siehe rückblickend Beispiel 2.5). Als *Provider* wird *VirtualBox* eingesetzt, da es sich als Open-Source Projekt bewährt hat – auch an der FH Aachen selbst, so kommt es in diversen Praktika zum Einsatz, was letztendlich für die Umsetzung der Anforderung **Verwendung bestehender Standards** spricht. Weiterhin erleichtert es die Anforderung **Dokumentation**, da die Installationsprozedur nicht mehr dokumentiert werden muss.

##### 3.3.3.1 Entwicklungsbetriebssystem

Als Basis-Betriebssystem für Vagrant wurde *Ubuntu 18.04 LTS (Bionic Beaver)*<sup>15</sup> erwählt, da ein breites Spektrum an aktuellen notwendigen Paketen bereits *out of the box* verfügbar ist und zudem als *Debian* Derivat eine ähnliche Open-Source-Philosophie vertreten wird (siehe Anforderung **Verwendung offener Standards**). Der Release Zyklus einer Ubuntu *Long-Term Support* Version ist ebenfalls gut zu handhaben, da von Canonical Ltd. (dem Unternehmen hinter Ubuntu) nach Erscheinen kostenlos 5 Jahre Support für kritische Updates gewährleistet wird, wodurch die Administrationskosten (damit ist auch Zeit gemeint) limitiert werden. Die genannte Ubuntu Version wird bis Anfang 2023 mit kritischen Updates inkl. Sicherheitsupdates versorgt [vgl. Can18], wodurch auch die Realisierung der Schutzziele aus Anforderung **Sicherheit** seitens des Entwicklungssystems bis zu diesem Zeitpunkt gewährleistet ist.

---

<sup>15</sup>Die Version wurde im Laufe der Arbeit von *Ubuntu 16.04 LTS* auf die Genannte aktualisiert.

### 3.3.4 Spring Boot

Ein REST-Service wird benötigt, um nach der Idee aus Abschnitt 3.3.1.3 alle ergänzenden Informationen zu einem Smart Contract außerhalb der Blockchain abzuspeichern und abzufragen. Um diesen REST-Webservice zu implementieren, der möglichst schnell produktiv einsatzbereit sein soll, wurde unter der Vielfalt der Technologien in diesem Bereich *Spring Boot* erwählt.

Dies liegt zum einen an den im Abschnitt 2.4 beschriebenen *Startern*, zum anderen allerdings auch an der generellen Philosophie *Convention over Configuration*: Spring Boot ermöglicht es mit verhältnismäßig wenig Aufwand, das Level 3 (HATEOAS) des *Richardson Maturity Model* für REST umzusetzen [siehe auch Fow10]. Das Modell klassifiziert REST nach 3 Leveln, anhand derer sich prinzipiell die Qualität eines REST-Services messen lässt. Beziiglich der Anforderungen wird durch die *Hypermedia Controls* (HATEOAS) ein hohes Maß an selbstdokumentierenden Schnittstellen erreicht (Anforderung Dokumentation), da bei einem Aufruf einer Ressource immer auch die kontextabhängigen weiteren Möglichkeiten ausgeliefert werden. Zudem dient Level 3 – aufgrund der hierarchischen Modell-Struktur, die auch unterliegende Level mit einschließt – der Anforderung Erweiterbarkeit. Eine höhere Qualität des REST-Services wird erreicht, da erstens auf den HTTP-Standard gesetzt wird, zweitens Ressourcen verwendet und drittens HTTP-Verben (semantisch) korrekt eingesetzt werden (respektive Level 0 bis 2) [vgl. Fow10]. Betrachtet man das Modell als eine Art Standard, so ist dies abschließend auch für die Anforderung Verwendung offener Standards von Bedeutung.

### 3.3.5 Docker

Zu Test-, Entwicklungs- und Lehrzwecken sollte es im Sinne der Anforderung Erweiterbarkeit ein *Ethereum Testnetzwerk* geben, in dem auch die Kommunikation unter den verschiedenen Knoten analysierbar ist.

Docker schien für diesen Einsatzzweck sehr angemessen, da die Knoten – in isolierten Containern verpackt – auf einfache Weise hoch skaliert werden können.

Obwohl Docker wie Vagrant auf einer Virtualisierungsschicht basiert, so ist der Anwendungsfall der jeweiligen Technologie bei dem zu konzipierenden System doch ein anderer. Docker ist zwar leichtgewichtiger, aber hat dadurch, dass es an den Betriebssystem-

Kernel gebunden ist, eine stärkere Kopplung zum Host (siehe Abschnitt 2.5). Diese Kopplung entfällt bei Vagrant, da es sich um eine vollständige Virtualisierung handelt, bei der das Gast-Betriebssystem gänzlich abgeschottet wird. Daher wurde entschieden, dass Vagrant für das (virtuelle) Entwicklungssystem auf Basis von Ubuntu (siehe Abschnitt 3.3.3.1) zum Einsatz kommt und Docker die darin laufende *Container-Technologie* ist. Dadurch können Inkompatibilitätsprobleme auf Systemen von unterschiedlichen Entwicklern verringert und die Container können so zielgerichtet für eine einzige Linux Distribution implementiert werden, ohne dass Sonderfälle in Betracht gezogen werden müssen.

#### 3.3.6 Hardware

An dieser Stelle wird zuerst erklärt, warum zusätzliche Hardware notwendig ist. Danach werden die beiden eingesetzten Hardwaretechnologien anhand der Anforderungen begründet.

Die Untersuchung des Ist-Prozesses „Klausur durchführen“ (Abschnitt 3.1.4.2) hat ergeben, dass es sich eignen würde, wenn digitale Signaturen zum Einsatz kommen (1. Verbesserungsvorschlag). Die Idee ist, dass der Student dazu seine FH-Karte nutzen kann, um digitale Signaturen vorzunehmen und (im Blockchain-Kontext) Transaktionen zu tätigen. Damit dies funktionieren kann, muss der Student (oder die Aufsicht) die Karte auf einen (nach Möglichkeit portablen) NFC-Reader platzieren, auf dem dann die Identität des Studenten abgespeichert ist.

Da Ethereum zum Einsatz kommt, muss also der in Abschnitt 2.1.4.2 beschriebene *externe Account* eines Studenten inkl. *Private Key* in einem bestimmten (wie folgt spezifiziertem) Format auf der NFC Karte abgelegt werden. Der NFC-Reader wird durch den im nächsten Abschnitt beschriebenen als *Controller* funktionierenden Raspberry Pi angesprochen, auf dem auch die Applikation läuft, um die digitale Signatur vorzunehmen und die Transaktionen an das interne Ethereum Netz der Fachhochschule abzuschicken. Die Kombination dieser beiden Geräte wird in der vorliegenden Arbeit *Terminal* genannt.

#### \* Beispiel 3.5: Gründe für den Namen „Terminal“

Der Endbenutzer nimmt eine Eingabe vor, z.B. indem er die NFC Karte auf das Lesegerät legt. Der Controller  $\pi$  (Raspberry Pi) kommuniziert dazu mit dem Lesegerät, damit eine Ausgabe erfolgen kann, z.B. eine Transaktion:  $\text{Input}_{\text{NFC}} \rightarrow \text{Controller}_\pi \rightarrow \text{Output}_{\text{TX}}$ . Zusätzlich zum NFC Lesegerät soll das Terminal über eine Benutzeroberfläche verfügen, auf die sich das Schema der Eingabe → Ausgabe ebenso anwenden lässt.

#### 3.3.6.1 Raspberry Pi

Für die Realisierung der Anforderung **Mengengerüst, Skalierbarkeit und Effizienz** spielt es eine große Rolle, kostengünstige Hardware zu verwenden, damit das Ziel der horizontalen Skalierbarkeit erreicht werden kann. Der Raspberry Pi Zero, der in diesem Projekt zum Einsatz kommt, kann zwar mit einer CPU von 1 GHz nicht als leistungsfähig bezeichnet werden, aber als besonders kostengünstig (siehe Abschnitt 2.6.1). Gerade diese beiden Eigenschaften machen ihn zum optimalen Testgerät: Oft fallen vermeidbare Bottlenecks bzw. Performance-Probleme in der Softwareprogrammierung erst auf, nachdem sie auf einem weniger leistungsstarken Gerät getestet werden. Ebenfalls von Vorteil ist, dass es ein *Compute Unit* gibt, das hilft, das Proof of Concept zu einem produktiv einsetzbaren Projekt zu transformieren.

Schlussendlich erfüllt der Einplatinencomputer auch die Anforderung **Verwendung offener Standards**, da es sich um gut dokumentierte Hardware handelt.

#### 3.3.6.2 NFC

Dass ein NFC-kompatibler Tag als *Identitätsmanagement* zum Einsatz kommt, hat einige **Vorteile**:

1. Anforderung **Verwendung bestehender Standards**: FH-Mitglieder besitzen bereits eine NFC Karte, die *FH Karte*...
2. ↴ ...somit wissen sie bereits, wie die Technologie verwendet wird (Umsetzung der Anforderung **Dokumentation** fällt leichter).
3. Anforderung **Sicherheit**: Der Endbenutzer wäre einem erhöhten Identitätsdiebstahl-Risiko ausgesetzt, wenn er seinen Private Key kennen müsste und diesen möglicherweise unverschlüsselt in einer nicht vertrauenswürdigen Umgebung ablegen würde.

### 3 Analyse und Konzeption

---

Dieses Problem wird umgangen, da der Private Key in verschlüsselter Form auf der NFC Karte abgespeichert werden soll.<sup>16</sup>

4. Anforderung **Einfachheit der Bedienung** (Abstraktion): Der Endbenutzer muss nicht wissen, *wie* oder *dass* intern eine Transaktion an ein Netz aus Ethereum Knoten getätigt wird; er muss lediglich wissen, wie das Terminal (als Frontend) bedient wird.

Zusammengefasst ist NFC eine gute Ergänzung zu Ethereum, da beide Technologien in Kooperation ein breites Spektrum an Anforderungen der FH Aachen umsetzen können.

In der vorliegenden Arbeit wird ein Lesegerät für NFC Tags des *ISO/IEC 14443A* Standards verwendet, da die FH Aachen mit der FH Karte – ein *MIFARE DESFire EV1* Tag – bereits auf diesen Standard setzt (siehe Anforderung **Verwendung bestehender Standards** und Abschnitt 3.1.3):

- **NFC Reader:** *MFRC522* – Standard performance MIFARE frontend<sup>17</sup>
- **NFC Tag:** *MIFARE Classic EV1 1K* – Mainstream contactless smart card IC<sup>18</sup>

Aufgrund der breiten Verfügbarkeit wurde im Rahmen des Proof of Concepts ein anderer Tag-Typ als der von der FH Karte verwendete *MIFARE DESFire EV1* Tag verwendet, speziell der *MIFARE Classic* Tag (der jedoch auch den *ISO/IEC 14443A* Standard implementiert). Im Folgenden wird das Speicherformat für diesen Tag spezifiziert, um Ethereum Accounts darauf ablegen zu können.

#### Spezifikation und Konzeption des eigenen Karten/Tag-Formats:

Der EEPROM eines *MIFARE Classic* Tags hat 16 Sektoren mit jeweils 4 Blöcken. Jeder Block hat eine Größe von 16 Byte [vgl. [NXP16a](#)]. Insgesamt hat die Karte damit eine Größe von 1024 Byte. Dies ist völlig ausreichend für die Abspeicherung aller notwendigen Werte eines Ethereum Accounts (siehe Abschnitt 2.1.4.2):

- $A(p_r)$ : Die öffentliche Ethereum Adresse des Private Keys  $p_r$ .
- $EC(p_r)$ : Der mit einer Funktion EC (Encrypt) verschlüsselte Private Key  $p_r$ .

Der Private Key  $p_r$  wird in keinem Fall unverschlüsselt abgelegt.

---

<sup>16</sup>Falls erfahrenere Benutzer über ihren Private Key verfügen möchten, so können sie diesen beispielsweise auf einer speziell dafür vorgesehenen Webseite *auf eigene Verantwortung* anfordern.

<sup>17</sup>siehe auch [NXP16b](#), „Standard performance MIFARE and NTAG frontend“.

<sup>18</sup>siehe auch [NXP16a](#), „Mainstream contactless smart card IC for fast and easy solution development“.

Stattdessen wird mit der Funktion **EC** eine symmetrische Verschlüsselung angewandt, die wie folgt definiert wird:

$$\text{EC}(p_r) \equiv \text{ENCRYPT}(p_r, \text{MASTERPW} + \text{SALT}(A(p_r))) \quad (3.1)$$

Dabei sei **ENCRYPT** eine – beliebige, den Anforderungen genügende – Funktion zur *symmetrischen* Verschlüsselung, die als ersten Parameter den zu verschlüsselnden Private Key  $p_r$  annimmt und als zweiten Parameter das Passwort.

Das **MASTERPW** (Master-Passwort) wird auf dem Controller, dem Raspberry Pi, gespeichert. Der Speicher des Raspberry Pis darf maximal für Administratoren lesbar sein. Mit der Funktion **SALT** wird ein FH-Mitglied spezifisches Salt erzeugt, das von der Ethereum Adresse  $A(p_r)$  abhängig ist. Dieses Salt wird mit dem Master-Passwort konkateniert und als Passwort für die Funktion verwendet. Das Salt soll verhindern, dass ein Angreifer, der das **MASTERPW** kennt, *alle* Tags (bzw. FH Karten) entschlüsseln kann. Es wird z.B. in einem unabhängigem Service abgelegt, der nur vom Terminal verwendet werden kann.

Die Daten werden wie folgt den jeweiligen Blöcken zugeordnet:

$\downarrow$ Sektor/Block $\rightarrow$	<b>0</b>	<b>1</b>	<b>2</b>	<b>3</b>
<b>0</b>	MF – BLOCK	Metadata	Reserved	TRAILER
<b>1</b>	$A(p_r)_{0..15}$	$A(p_r)_{16..19}$	$\emptyset$	TRAILER
<b>2</b>	$\text{EC}(p_r)_{0..15}$	$\text{EC}(p_r)_{16..31}$	$\text{EC}(p_r)_{32..47}$	TRAILER
<b>3-15</b>	...	...	...	TRAILER

**Tabelle 3.4:** Format-Spezifikation des *MIFARE Classic* Tags mit Ethereum Account.

Alle in Tabelle 3.4 großgeschriebenen Felder sind durch den Hersteller vorgegebene Spezialfelder, in die üblicherweise keine frei wählbaren Daten abgelegt werden (können). Alle anderen Felder sind für Nutzdaten verfügbar.

Der **MF – BLOCK** (Manufacturer-Block) in Sektor 0 Block 0 ist unveränderlich (*read-only*) und wird bei der Produktion des Tags initial beschrieben. Er enthält neben anderen herstellerspezifischen Daten auch die 4 Byte große eindeutige ID, die *UID* des Tags, die vom Hersteller einmalig vergeben wird [vgl. [NXP16a](#), S. 1]. Im **Metadata** Feld werden folgende Werte abgelegt: Eine Kennzeichnung (dass es sich um eine Karte der FH Aachen handelt), die Version des Tags (beispielsweise wichtig, wenn das o.g. Format geändert wird) und weitere Optionen. Das nächste Feld ist aus Erweiterungsgründen reserviert. Der **TRAILER** am Ende Sektors kann nicht mit Nutzdaten gefüllt werden; er beinhaltet

u.a. die *Access Bits*, die festlegen, welche Blöcke in dem vorliegenden Sektor beschrieben werden dürfen. Außerdem beinhaltet er zwei *Secret Keys*, welche die Authentifikation ermöglichen – der NFC Reader muss sich zuerst bei dem Tag „anmelden“, damit der Tag Informationen ausliefert. Der IC verfügt dazu über einen proprietären *CRYPTO1* genannten Mechanismus zur Verschlüsselung und Authentifizierung der Daten auf dem Tag [vgl. [NXP16a](#), 5 ff.].

Weiter geht es mit den eigentlichen Nutzdaten des Tags: Die ersten beiden Blöcke von Sektor 1 enthalten die Ethereum Adresse  $A(p_r)$  ohne die eingebaute Checksumme (siehe Beispiel [2.2](#)), da dadurch Platz gespart werden kann und die Adresse auf diese Weise mit lediglich 20 Byte in nur einen einzigen Sektor passt.

Der verschlüsselte Private Key  $EC(p_r)$  erstreckt sich über die verbleibenden Blöcke, falls notwendig bis zum letzten Nutzdaten-Block des Speichers (Sektor 15 Block 2). Die symmetrische Verschlüsselungsfunktion ist hierbei so zu wählen, dass die Ausgabe nie größer als die verbleibenden 42 Nutzdaten-Blöcke (672 Byte) sind.

#### Exkurs: Sicherheitslücken in *MIFARE Classic*

Im späteren Verlauf des Projekts hat sich herausgestellt, dass *MIFARE Classic* Tags gravierende Sicherheitslücken aufweisen, die nicht im Datenblatt des Herstellers dokumentiert sind [vgl. [NXP16a](#)]:

**Erstens – CRYPTO1.** Die Funktion zur Authentifizierung und Verschlüsselung ist seit 2009 vollumfänglich mittels eines *Brute-Force*-ähnlichen Ansatzes angreifbar. In der Regel dauert es rund eine Stunde, den Inhalt der Karte vollständig zu kopieren (engl. *full Dump*), selbst wenn alle Sektoren „verschlüsselt“ sind [vgl. [Alm14](#)].

**Zweitens – UID.** Die vermeintlich eindeutige ID des Tags ist zwar vom Hersteller als eindeutig vorgesehen. Jedoch hat dies andere Hersteller (z.B. aus dem asiatischen Raum) nicht daran gehindert, Tags herzustellen, bei denen auch der erste Block beschrieben werden kann (entweder mittels *Backdoor*-Kommando oder noch einfacher als reguläre Schreiboperation). Im ersten Block ist auch die *UID* abgespeichert, die dann frei gewählt werden kann [vgl. auch [Alm14](#)]. Dieser Angriff konnte auch selbst getestet und nachvollzogen werden.

Durch Ausnutzen beider Lücken ist es möglich, vollständige Tag-Replikate anzufertigen.

Aufgrund der später aufgefallenen Sicherheitslücken darf der im Projekt verwendete *MIFARE Classic* Tag *nicht* produktiv eingesetzt werden (siehe Exkurs für Details), da er gegen die Sicherheitsanforderungen verstößt. Stattdessen sollte im wirklichen Produk-

### *3 Analyse und Konzeption*

---

tiveinsatz nach Möglichkeit die *FH Karte* weiterverwendet werden, da diese auf den derzeitig sicheren *MIFARE DESFire EV1* Tag setzt. In der Theorie sollte eine derartige Migration möglich sein, da jener mehrere Speicherbereiche besitzt und mit 8 kB auch über ausreichend Speicherplatz verfügt [vgl. [NXP15](#)] (siehe Abschnitt [3.1.3](#)). Dies wäre ohnehin die beste Lösung, da dann keine neuen zusätzlichen Karte eingeführt werden müsste.

# 4 Die FHACChain

Die *FHACChain* (Kofferwort aus den Begriffen FHAC<sup>1</sup>, und *Blockchain*) ist das Gesamtergebnis des Praxisprojekts und damit auch der vorliegenden Arbeit. Es handelt sich um einen Technologie-Stack, der die Integration von Ethereum in die Geschäftsprozesse der Fachhochschule Aachen – im Sinne der in Abschnitt 3.2 dargelegten Anforderungen – ermöglicht.

In diesem Kapitel wird zu Beginn ein Überblick der *FHACChain* Kernkomponenten gezeigt. Darauffolgend wird die Funktionalität der einzelnen Komponenten im Detail beschrieben.

Im Anschluss wird dann eine erste minimale Integration vorgestellt, die sich jedoch auch auf die folgenden Geschäftsprozesse anwenden lässt.

Der erste optimierte Geschäftsprozess, der vorgestellt wird, ist „Prüfung durchführen“. Er wurde bereits als Ist-Prozess in Abschnitt 3.1.4 beschrieben, im folgenden Abschnitt 4.3.1 wird er mit Ethereum verbunden und mit dieser Technologie werden in einem Zug die Verbesserungen aus Abschnitt 3.1.4.2 realisiert.

Danach folgt der Prozess „Bachelorarbeit schreiben“, der bereits zuvor im Fach Geschäftsprozessmanagement als Soll-Prozess fertiggestellt wurde. In dieser Bachelorarbeit wird er wiederverwendet, um darzustellen, welche Auswirkungen die Integration einer Blockchain auf bestehende (funktionsfähige) Prozesse hat; es wird verglichen, welche Änderungen bezüglich der Entwicklung durch die dezentralen Eigenschaften Ethernets entstehen.

Alle Prozesse sind mit Camunda auch vom Leser ausführbar.

---

<sup>1</sup>Stilisierte Abkürzung zu „Fachhochschule Aachen“.

## 4.1 Kernkomponenten

Der *FHACChain* Technologie-Stack lässt sich architektonisch in drei Bereiche unterteilen:

- **Frontend:** Technologien, die der Endbenutzer verwendet, etwa um Eingaben zu tätigen und/oder visuelle Ausgaben zu erhalten. Diese verfügen typischerweise über ein UI.
- **Backend:** Technologien, die im Hintergrund arbeiten und deren *interne* Vorgänge für den Endbenutzer nicht sichtbar sein sollten. Sie enthalten die Geschäftslogik der *FHACChain*.
- **Middleware-Schicht:** Enthält Software, die *Frontend* und *Backend* verbindet und als Vermittler agiert.

Die bereits dargelegten verwendeten Technologien lassen sich demzufolge wie folgt einordnen:

Schicht	Technologien	System	Ref.-Konzeption
Frontend	Raspberry Pi mit NFC, Ethereum	Terminal	<a href="#">3.3.1.3</a> , <a href="#">3.3.6</a>
Backend	Ethereum, Spring Boot	Full Server	<a href="#">3.3.1.3</a> , <a href="#">3.3.4</a>
Middleware	Camunda BPM	Full Server	<a href="#">3.3.2</a>

**Tabelle 4.1:** Zuordnung der *FHACChain* Technologien zur jeweiligen Schicht.

Die Tabelle 4.1 zeigt die Zuordnung der Technologien und neben einer Referenz zum Kapitel 3 auch das System, auf dem die Technologien zum Einsatz kommen.

Neben den beiden Systemen *Terminal* und *Full Server* gibt es ein in der Tabelle nicht aufkommendes *Node-only* System, das exemplarisch existiert, jedoch nicht im Proof of Concept zum Einsatz kommt. Es handelt sich dabei (konträr zum *Full Server*) um ein System, auf dem als einzige Anwendung ein *Ethereum Full Node* läuft.

Ein Ethereum *Full Node* synchronisiert die gesamte Ethereum Blockchain mit anderen Knoten (mehr zu den verschiedenen Full Node-Typen im folgenden Abschnitt 4.1.2). Im Gegensatz dazu lädt ein *Light Node* nur – für die sichere Ausführung von Transaktionen – benötigte Zustände der Blockchain herunter. Dieser Knoten-Typ benötigt prinzipiell weniger Rechner-Ressourcen, insbesondere weniger Speicherplatz [vgl. [Eth18f](#)]. Aus diesem Grund soll er auf dem *Terminal* System zum Einsatz kommen.

Auf der nächsten Seite wird der Technologie-Stack als Überblick in Form einer Abbildung seitwärts dargestellt, der alle in Tabelle 4.1 genannten Technologien beinhaltet.

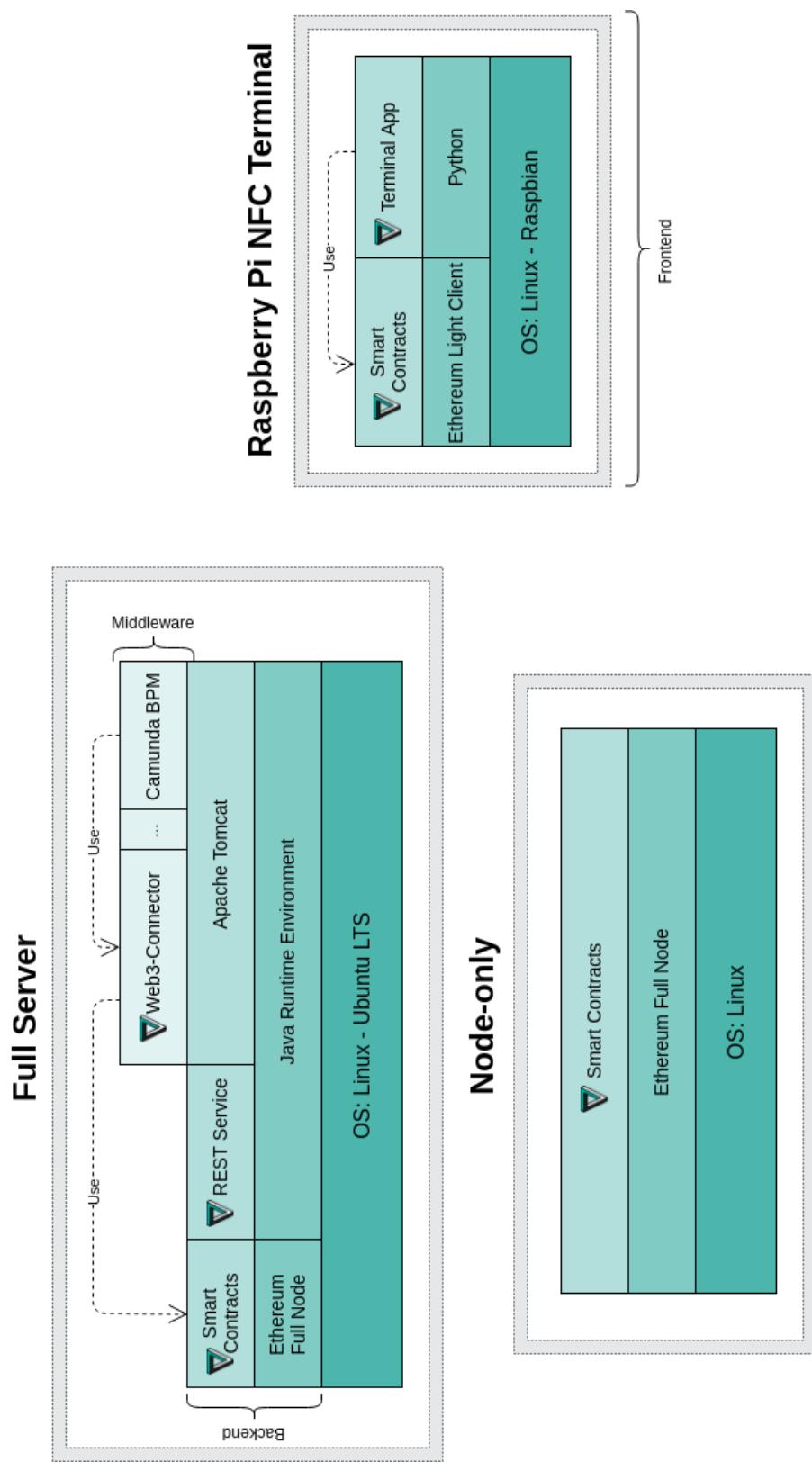


Abbildung 4.1: Der FHACChain Technologie-Stack.

ERSTEILLT VIA: draw.io v8.7.6

Alle in der Abb. 4.1 mit dem -Symbol (das *FHACChain*-Logo) markierten Elemente des Stacks sind Kernkomponenten des Proof of Concepts, die u.a. in den folgenden Abschnitten im Detail vorgestellt werden.

Es sollte noch erwähnt werden, dass dies keine klassische Client-Server Architektur ist. Der *Full Server* wird lediglich als solcher bezeichnet, da er auch einen *Apache Tomcat* Webserver und einen *Spring Boot REST Service* beinhaltet. Ohne diese beiden Komponenten könnte man jedoch alle Systeme als *Clients* bezeichnen; dies ist ein wichtiger Aspekt, der sich anhand der Dezentralisierung begründen lässt.

### Tutorial 4.1: Speichermedium

Interaktive Tutorials finden sich in Boxen wie der Vorliegenden wieder.

Auf der letzten Seite der Arbeit findet sich ein Speichermedium, auf dem die *FHACChain* Distribution als ZIP-Datei liegt. Bevor die Tutorials befolgt werden können, sollte diese auf einen PC kopiert werden.

Für alle folgenden Befehle ist unter Windows die Verwendung der Git-Bash und unter Linux die Verwendung einer Bash empfohlen.

#### 4.1.1 Systemumgebung

Die Systemumgebung des *Full Servers* aus Abb. 4.1 kann mit *Vagrant* als eine virtuelle Maschine aufgesetzt werden. Wie in 3.3.3.1 konzipiert wurde, wird als Betriebssystem-Basis ein *Ubuntu LTS* eingesetzt.

Sobald der Befehl `vagrant up` ausgeführt wird, werden folgende Schritte automatisiert (vgl. mit Abschnitt 2.3):

- 1. Erstellung der VM:** Die *Ubuntu 18.04 LTS* Basis wird heruntergeladen und entpackt. Dann wird die Maschine mittels des Providers *VirtualBox* erstellt und 4 CPU Kerne und 2048 GB RAM zugewiesen.<sup>2</sup>
- 2. Konfiguration der VM:** Die VM bekommt den Hostnamen „fhachain“ und die IP-Adresse 192.168.42.42 in einem privaten – vom Host erreichbaren – Netzwerk zugewiesen. Zudem wird in der VM der Ordner `/vagrant` erstellt und mit dem Host geteilt.

<sup>2</sup>Die Werte können in der *Vagrantfile* in Form von Konstanten konfiguriert werden. Die Mindestanforderungen der VM sind wie folgt festgelegt: 2 CPU Kerne, 2048 GB RAM.

3. **Hochfahren:** *Ubuntu 18.04 LTS* wird gestartet. Es wird so lange gewartet, bis alle vom Betriebssystem benötigten Programme hochgefahren sind.
4. **Provisioning:** Das *FHACChain* Installations-Shellskript wird ausgeführt:
  - Die Paketquellen und die Pakete des Betriebssystems werden aktualisiert.
  - Es werden Anpassungen an *Ubuntu 18.04 LTS* vorgenommen (wie etwa ein neuer Benutzer und das Hinzufügen eines Banners, das installierte Versionen anzeigt).
  - Alle Softwarekomponenten aus Abb. 4.1 werden *installiert, konfiguriert* und *gestartet*.
  - Smart Contracts werden zur Entwicklungs-Blockchain deployed und mit Beispieldaten befüllt.

### ▼ Tutorial 4.2: Vagrant – Aufsetzen der Systemumgebung

**Voraussetzungen:** Vagrant, VirtualBox und eine aktive Internetverbindung.

**Aufsetzen:** Im Ordner `fhachain/vm` liegen die erforderlichen Daten. Indem der Befehl `vagrant up` in diesem Verzeichnis ausgeführt wird, werden alle notwendigen Schritte (s.o.) zum Aufsetzen und Hochfahren der VM mittels Vagrant durchgeführt. Das Provisioning dauert etwa 5 bis 10 Minuten. Jeder Installationsschritt wird im Terminal ausgegeben.

**SSH:** Nachdem die VM aufgesetzt wurde, kann mittels `vagrant ssh` eine Secure Socket Verbindung aufgebaut werden. Das Banner aus Abb. 4.2 erscheint. Alle von nun an eingegebenen Befehle werden in der VM, dem Gastsystem, ausgeführt (erkennbar an „`vagrant@fhachain: $`“ vor der Befehlseingabe). Damit verständlich ist, dass das Terminal der VM gemeint ist, wird dieses fortlaufend immer *VM-Terminal* genannt.

**Herunterfahren:** Mit `vagrant halt` wird die VM regulär heruntergefahren.

**Löschen:** Sollte die VM nicht mehr benötigt werden, so lässt sie sich rückstandslos mit `vagrant destroy` wieder entfernen.

Der Zyklus kann beliebig oft wiederholt werden. Falls sie doch wieder benötigt wird, kann sie wieder wie oben beschrieben aufgesetzt werden.

In der VM wird standardmäßig aus Effizienzgründen (und da es ein Entwickler- und kein Produktivsystem ist) *kein* vollständiges Netzwerk aus Ethereum Knoten aufgebaut. Stattdessen wird eine Software namens *Ganache CLI* installiert, die im Wesentlichen eine JS-basierte Ethereum Entwicklungs-Blockchain ist. Diese enthält beispielhaft fünf Ethereum Accounts, die zur Entwicklung genutzt werden können. Die Smart Contracts, die beim *Provisioning* deployed werden, haben bei jedem Neuaufladen der VM die gleiche Adresse, wodurch manuelle Änderungen bei jeder Smart Contract Anbindung entfallen.

Die zum Einsatz kommende *Camunda BPM Tomcat Distribution* wird vom Installationsskript dahingehend modifiziert, dass ein Deployment von Webapplikationen einschließlich der Geschäftsprozesse im *WAR-Format* über einen dafür vorgesehenen *Tomcat Manager Script User* möglich ist. Zusätzlich zu dieser Änderung wird ein eigens implementierter Camunda Connector namens *Web3-Connector* zu Apache Tomcat hinzugefügt (mehr dazu im folgenden Abschnitt 4.2.2). Das *Camunda Cockpit* wird mittels JavaScript-only Plugin-Mechanismus um ein *Admin UI* für Smart Contracts erweitert (mehr im Abschnitt 4.1.4.2).

Alle Komponenten werden mittels *systemd* gestartet und in Form von Services verwaltet.

### ▼ Tutorial 4.3: Logs und Prozessverwaltung

Die folgenden Services existieren: `camunda`, `ganache`, `fhachain-rest`

**Logs:** Log-Ausgaben können im VM-Terminal mit `journalctl -u ${SERVICE}` angesehen werden. Mit angehängtem `-f` (für „follow“) werden die Logs live aktualisiert und neue Einträge automatisch zum Standard Output des Terminals hinzugefügt.

**Starten, stoppen oder Neustarten:** Dies ist mit `sudo systemctl {start|stop|restart} ${SERVICE}` möglich.

**Beispiel:** Camunda BPM neustarten: `sudo systemctl restart camunda`

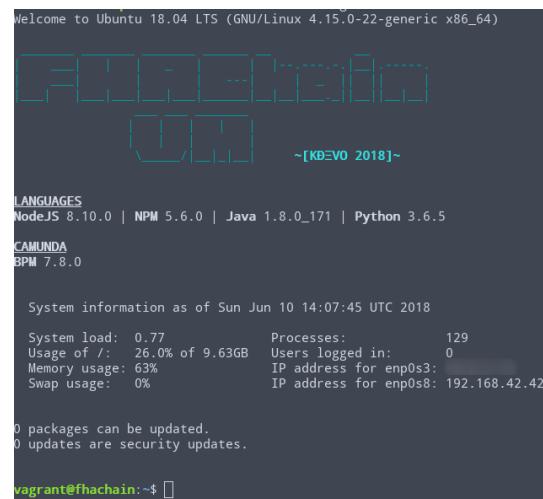


Abbildung 4.2: Die FHACChain VM.

### 4.1.2 Ethereum Test-Netzwerk via Docker

Um möglichst viel über die Kommunikation der Knoten in einem Ethereum Netzwerk in Erfahrung zu bringen und da mehr Kontrolle besteht, wenn nicht eines der vorhandenen öffentlichen Test-Netzwerke verwendet wird, wurde die Entscheidung getroffen, zu Beginn ein eigenes Netzwerk aufzubauen. Da dieses Netz intern ist, lässt es sich auch einfacher auf die FH Aachen übertragen, bei der schließlich auch kein *öffentliches* Test-Netzwerk verwendet werden sollte (siehe Abschnitt 3.3.1.3).

In einem Ethereum Netzwerk unterscheidet man zwischen folgenden Knoten-Typen (*Full Nodes*) [vgl. Eth17]:

1. **Bootnode:** Ein spezieller leichtgewichtiger Ethereum Knoten welcher lediglich das *Ethereum Node Discovery* Protokoll implementiert und *ausschließlich* für die Vermittlung der anderen Knoten zuständig ist.
2. **Node:** Der Knoten synchronisiert die Blockchain: Es wird überprüft, ob die empfangenen Blöcke valide sind - sollte dies der Fall sein, so werden sie an andere Knoten via Peer-to-Peer (außer an *Bootnodes*) weitergegeben.
3. **Node aktiviertem Mining:** Dieser Knoten hat die selbe Funktionalität wie der reguläre Knoten, übt jedoch Mining aus (siehe Abschnitt 2.1.3).

Zu beachten ist, dass die jeweiligen Nachfolger der Knoten die Funktionalität ihrer Vorgänger *erweitern* (vergleichbar mit einer Hierarchie). Speziell bedeutet dies, dass jeder *Node* auch die Funktionalitäten des *Bootnodes* mitbringt. Selbiges gilt für den *Node mit Mining-Funktionalität*.

Die Konstellation der Knoten-Typen lässt sich auch in der *Docker Compose* Datei wiedererkennen:

**Listing 4.1: docker-compose.yml der FHACChain**

```

1 version: '2'
2 services:
3   bootnode:
4     build: images/bootnode
5     # [...]
6     volumes:
7       - /etc/localtime:/etc/localtime:ro
8     environment:
9       - BOOTNODE_PORT=${BOOTNODE_PORT}

```

```
10 node:
11   build: images/node
12   links:
13   - bootnode
14   environment:
15   - ENABLE_MINING=false
16   # [...]
17 miningnode:
18   build: images/node
19   links:
20   - bootnode
21   environment:
22   - ENABLE_MINING=true
23   # [...]
```

Docker ist für den Anwendungsfall besonders gut geeignet, da sich die Anzahl und Art der Knoten mithilfe eines einzigen Befehls skalieren lässt (hier 1 Bootnode, 2 Nodes und 2 Mining-Nodes):

---

```
sudo docker-compose scale bootnode=1 node=2 miningnode=2
```

---

### 4.1.3 Backends

#### 4.1.3.1 Smart Contracts

Alle Smart Contracts im Projekt wurden in *Solidity* implementiert und in den folgenden Abschnitten auch kurz *Contracts* genannt. Eine Übersicht aller implementierten FHACChain Smart Contracts bietet das folgende UML-Diagramm im Querformat, das auch im späteren Verlauf noch als Referenz eingesetzt wird:

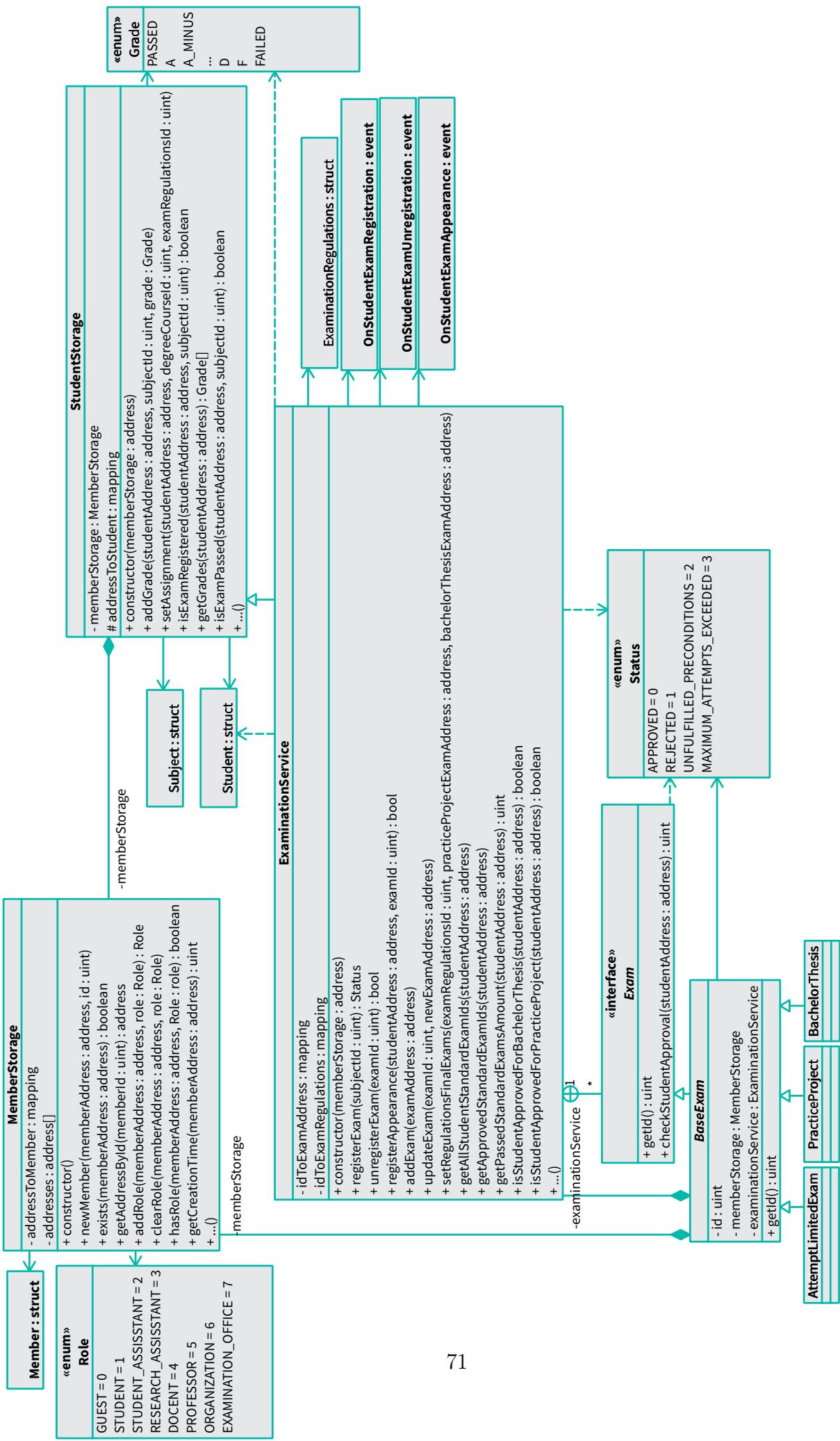


Abbildung 4.3: Die Smart Contracts der FHACChain dargestellt als UML-Diagramm.

Die Contracts können in UML Notation auch als Klassen dargestellt werden. Es gibt drei Haupt-Contracts, die relevant für die folgenden optimierten Geschäftsprozesse sind:

- **MemberStorage:** Dieser Smart Contract speichert die Mitglieder der Fachhochschule Aachen, wie konzipiert (siehe Abschnitt 3.3.1.3) mit Pseudonymen. Speziell bedeutet dies, dass als ein *primäres* Identifizierungsmerkmal die Ethereum Adresse des Mitglieds verwendet wird. Diese Adresse wird mithilfe eines assoziativen Datenfeld (ein *Mapping*) zum jeweiligen *Member* zugeordnet. In der *Member* Struktur werden u.a. auch die Rollen (*Roles*) des Mitglieds abgespeichert und eine weitere ID. Letztere wird als Möglichkeit genutzt, um etwa bei einem *Member* mit der Rolle *Student* die Matrikelnummer abzuspeichern. Bei anderen Mitgliedern wurde davon ausgegangen, dass diese auch eine eindeutige Kennnummer haben, die bei den Services der Fachhochschule im Hintergrund verwendet wird – Entitäten relationaler Datenbanken brauchen schließlich Primärschlüssel.<sup>3</sup> Auf diese Weise wird eine Kompatibilität zu den bestehenden Services der Fachhochschule Aachen hergestellt.
- **StudentStorage:** Ein weiterer Speicher, der aber die spezielle Mitgliedsform der Studenten persistiert. Der *StudentStorage* Contract benötigt den globalen *MemberStorage*-Contract, der im Konstruktor in Form einer Contract Adresse entgegengenommen wird. Auch hier ist die führende ID die Ethereum Adresse des Mitglieds, so dass ergänzende Informationen zum Studenten als generelles FH-Mitglied mithilfe des *MemberStorage*-Contracts abgefragt werden können. Die *Student*-Datenstruktur beinhaltet die Fachrichtung des Studierenden, die Prüfungsordnung und die Fächer (*Subject*-Struktur, enthält u.a. auch die Noten des Studenten).
- **ExaminationService:** Der *StudentStorage*-Contract wird *erweitert* (Vererbung unter Contracts ist in Solidity möglich). Der *ExaminationService* ist – anders als die beiden Vorgänger – nicht lediglich eine Persistenz-Schicht, sondern kann sowohl von FH-Mitarbeitern als auch von Studierenden für diverse *Prüfungsangelegenheiten* verwendet werden. So offeriert der Contract für Studierende etwa die Funktionalität des An- und Abmelden von Prüfungen. Berechtigte Personen können jedoch auch neue Prüfungen (Contracts die das *Exam*-Interface implementieren) hinzufügen. Dies ist eine besondere Funktion, deren Leistungsfähigkeit im späteren Abschnitt 4.3.2 demonstriert wird.

---

<sup>3</sup>Daher heißt dieses Feld in der Implementierung auch generisch `id` und nicht etwa `studentNumber`.

Solidity ist zwar an bekannte höhere Programmiersprachen wie C++, Python und JavaScript angelehnt [vgl. Eth18d, S. 1], jedoch gibt es eine Vielzahl nicht offensichtlicher Besonderheiten, die beispielhaft anhand eines Auszugs des *MemberStorage*-Contracts im Detail vorgestellt werden.

**Listing 4.2: fhachain/smart-contracts/contracts/MemberStorage.sol**

```

1 pragma solidity ^0.4.23;
2 import "./util.sol";
3 contract MemberStorage is mortalAndOwned {
4     mapping(address => Member) private addressToMember;
5     address[] public addresses;
6     enum Role {
7         GUEST, STUDENT, /*...*/ PROFESSOR //... see UML diagram
8     }
9     uint8 constant ROLE_SIZE = 8;
10
11    struct Member {
12        uint id; // Should be UNIQUE!
13        mapping(uint8 => bool) roleToExistingFlag; // (Un)used roles
14        uint creationTime;
15        string freeField;
16    }
17
18    function newMember(address memberAddress, uint id,
19                      string freeField) public onlyByOwner
20    {
21        require(id != 0); // ID 0 is not allowed
22        require(!exists(memberAddress));
23
24        addressToMember[memberAddress].freeField = freeField;
25        addressToMember[memberAddress].id = id;
26        addressToMember[memberAddress].creationTime = now;
27        allAddresses.push(memberAddress);
28    } // [...]
29 }
```

Es ist erkenntlich, dass die Sprache statisch typisiert ist.

Zeile 4 beschreibt das mapping von Ethereum Adressen auf Mitglieder, welches bereits oben beschrieben wurde. Die Sichtbarkeit steht, anders als etwa in Java, hinter dem

Datentypen. Die dazugehörige Mitglieder-Datenstruktur ist als struct in Zeile 11 bis 16 definiert.

Zusätzlich wird in Zeile 5 ein Adressen-Array deklariert, welcher notwendig ist, da es nicht möglich ist, durch den mapping Datentyp zu iterieren. Folgender Grund ist dafür verantwortlich: Der eigentliche Key eines Mappings (in diesem Fall eine address) ist nicht im Klartext abgespeichert wird, sondern vorher irreversibel mit der kryptografischen Hashfunktion *SHA3 Keccak* geshashed [vgl. Eth18d, S. 54 f.]. Dadurch entsteht zwar eine Komprimierung (vorausgesetzt der Key ist kürzer als die Länge des Hashwertes von 256 bit,, was auch der Wortgröße aus Abschnitt 2.1.4.4 entspricht), jedoch könnte es für den Entwickler ungewohnt sein – verglichen mit der *HashMap* oder dem *Dictionary* aus anderen Programmiersprachen. Beide Variablen aus Zeile 4 und 5 sind *Zustandsvariablen* und werden damit im nicht-volatilen Speicher (*storage* genannt) der EVM gespeichert (siehe Abschnitt 2.1.4.4) [vgl. Eth18d, S. 39 ff.].

Weiterhin ist das `addresses` Feld `public`. Dies ist nicht etwa ein Bruch objektorientierter Konventionen, sondern: Das `public` Keyword sorgt dafür, dass der Solidity Compiler *Getter-Funktionen* automatisch generiert, jedoch keine *Setter* [vgl. Eth18d, S. 72 f.], damit andere Smart Contracts *lesend* darauf zugreifen können. Aus diesem Grund wurde das Feld im UML-Diagramm (Abb. 4.3) auch nicht als `public` mit einem „+“ gekennzeichnet, da es zu Missverständnissen führen könnte, d.h. es ist nicht im klassischen Sinne der OOP „öffentlich“.

#### Beispiel 4.1: Die „Glasbox Ethereum“

Auch die Sichtbarkeit `private` bedeutet nicht, dass es für niemanden möglich ist, den Inhalt einer Variable zu lesen. Die Definition 3.1 gilt weiterhin: Ethereum ist eine vollständig transparente *public permissionless* Blockchain, d.h. externe Akteure können stets *alles* lesen – wie eine „Glasbox“. Nur die Smart Contracts, die Agenten, welche in der Box „leben“, müssen sich an das Regelwerk (das Protokoll) halten.<sup>a</sup>

---

<sup>a</sup>Dies deckt sich ebenso mit Definition 2.3.

Zeile 9: Obwohl die Konstante für die Größe des Enums<sup>4</sup> ebenfalls eine *Zustandsvariable* ist, so kostet sie keinen Speicherplatz, da nach dem Kompilieren alle `ROLE_SIZE` Vorkommnisse durch den Wert 8 ersetzt werden [vgl. Eth18d, S. 75].

---

<sup>4</sup>Die Konstante wird benötigt, da sich die Enum-Größe nicht anderweitig bestimmen lässt.

Die Funktion `newMember` ab Zeile 18 wird zum Erstellen neuer FH-Mitglieder verwendet; mit einer Ethereum Adresse, der oben zuvor beschriebenen ID (die etwa die Matrikelnummer eines Studenten ist) und einer `freeField` Zeichenkette, die frei beliegbar ist. Die Zeichenkette wurde insbesondere zum Testen verwendet, sie kann jedoch auch als eine Art „Beschreibung“ des Mitglieds dienlich sein, auch wenn in Betracht gezogen werden muss, dass die Informationen FH-weit (öffentlich) lesbar sind (Anforderung Sicherheit).

In der ersten Zeile des Rumpfs (21) gibt es zunächst das `require` Keyword, welches einen Ausnahmestatus der EVM (*Revert*, siehe Abschnitt 2.1.4.4) auslöst, falls die innen stehende Bedingung nicht erfüllt ist [vgl. Eth18d, S. 59]. Dass keine IDs mit dem Wert 0 erlaubt sind hat folgenden Grund: Alle Werte sind in Solidity mit Defaultwerten vorbelegt (z.B.: `bool` hat den Wert `false` und `uint` den Wert 0). Es wurde – ohne ein neues Feld in der Member-Struktur einzuführen – nach einer Möglichkeit gesucht, bereits existierende Mitglieder zu erkennen. Falls ein Mitglied mit der ID 0 erstellt würde, so wäre diese Möglichkeit ausgeschlossen. Weiter in Zeile 22 wird dieser Mechanismus angewandt; `exists` überprüft, ob es in dem Mapping unter der angegebenen Adresse bereits ein Mitglied existiert (der Wert wäre dann ungleich 0):

```
function exists(address memberAddress) public view returns (bool) {
    return addressToMember[memberAddress].id != 0;
}
```

---

Die Funktion hat den `view`-Modifikator. Dieser erfordert, dass die Funktion keine Zustandsänderung vornimmt (wie ein „Versprechen“) [Eth18d, S. 76].

Alle `getter`-Funktionen, die keine Seiteneffekte haben, lassen sich demnach auch als `view` deklarieren.

Weiter mit dem eigentlichen Listing 4.2 – in den verbleibenden Zeilen 24 bis 26 werden (nach den beiden Überprüfungen durch `require`) die Mitgliedsdaten eingetragen. Zudem wird in Zeile 26 der Block-Zeitstempel<sup>5</sup> gesetzt, damit approximiert werden kann, wann das Mitglied erstellt wurde.

In der letzten Zeile der Funktion wird die Adresse an das Ende des dynamisch wachsenden Arrays angehängt.

---

<sup>5</sup>Siehe auch rückblickend Abschnitt 2.1.4.4, ggf. auch das optionale Beispiel 2.4.

Es bleibt noch die Frage, wie erreicht wird, dass *nur autorisierte* Mitglieder mit der soeben vorgestellten Funktion neue Mitglieder hinzufügen können. Erreicht wird dies mit einem zusätzlichen Smart Contract, der im Listing 4.2 in Zeile 2 importiert wird:

**Listing 4.3: fhachain/smart-contracts/contracts/util.sol**

```

1 pragma solidity ^0.4.23;
2 contract mortalAndOwned {
3   address owner;
4
5   constructor() public { owner = msg.sender; }
6
7   modifier onlyByOwner {
8     require(msg.sender == owner);
9     _; // <-NOTE: Restricted code will go here.
10  }
11
12  function kill() public onlyByOwner { selfdestruct(owner); }
13 }

```

Der Contract erfüllt zwei Funktionen. Erstens bietet er einen sogenannten *modifier* an, der in Listing 4.2 Zeile 19 am Ende verwendet wird. Zweitens macht er den Contract, der von ihm erbt, „sterblich“ (engl. *mortal*), d.h. er bekommt eine Art Destruktor.

Der Konstruktor in Zeile 5 stellt sicher, dass der *owner* (dt. Eigentümer) des Contracts als Zustandsvariable gespeichert wird. Sobald ein Smart Contract mit einer *Contract-creating Transaction* erstellt wird (siehe Abschnitt 2.1.4.3), so wird dieser Konstruktor aufgerufen. Das spezielle Feld *msg.sender* beinhaltet den Urheber der Transaktion.

Der erwähnte *onlyByOwner*-Modifier findet sich in den Zeilen 7 bis 10 wieder. Sobald er wie in Listing 4.2 Zeile 19 verwendet wird, stellt er sicher, dass die Funktion lediglich vom Eigentümer aufgerufen werden kann (Zeile 8). Der Rumpf der Funktion, die mit *onlyByOwner* eingeschränkt wird, folgt „virtuell“ nach dem Unterstrich „\_“ in Zeile 9, der eine Art Markierungszeichen darstellt [vgl. Eth18d, S. 75].

Die Funktion *kill* in Zeile 12 ermöglicht es, die Ressourcen wieder freizugeben, die in der Blockchain durch den Contract Code belegt werden. Dies ist eine Art Empfehlung an die Ethereum Knoten, die jedoch nicht verbindlich ist [vgl. Eth18d, S. 21]. Schlussendlich bleibt zu sagen, dass es fatal wäre, wenn die Funktion von jedem Teilnehmer aufgerufen werden könnte, daher trägt sie ebenfalls den *onlyByOwner*-Modifier.

#### 4.1.3.2 REST Service

Der Spring Boot REST Service ist ausschließlich zum Ausliefern zusätzlicher Informationen nach der Konzeption aus Abschnitt 3.3.4 zuständig. Er bietet die folgenden Ressourcen an (der Basis-Kontextpfad ist /api/v1):

- /general/info: Generelle Informationen über den Service, einschließlich Systemzeit und App- sowie API-Version.
- /subjects: Die Ressource ist für das Ausliefern Fächern (engl. „Subjects“) zuständig. Das Datenobjekt beinhaltet die gleiche ID, welche auch im Smart Contract definiert ist, damit zusätzliche Informationen zu Fächern über die ID angefordert werden können. Diese sind unter anderem der vollständige Name des Fachs und eine Abkürzung.
- /files: Diese Schnittstelle handhabt das Hochladen, Herunterladen und Auflisten von Dateien.<sup>6</sup>

#### ▼ Tutorial 4.4: Zugriff auf den REST Service vom Host

Außerhalb der VM ist der Service unter <http://192.168.42.42:8081/> erreichbar.

**Beispiel:** Um alle verfügbaren (exemplarischen) Fächer aufzulisten, kann <http://192.168.42.42:8081/api/v1/subjects> aufgerufen werden. Weitere Möglichkeiten der Kommunikation sind im \_links Feld der JSON-Ausgabe zu finden.

Bei der Implementation war insbesondere die Kombination aus den Startern spring-boot-starter-hateoas, spring-boot-starter-data-rest und spring-boot-starter-ata-jpa hilfreich, da sie es ermöglichen, mit nur wenigen Zeilen Code einen REST-Service mit integrierter (H2-)Datenbank zu entwickeln. So wird die /subjects Ressource lediglich als ein Interface angegeben:

#### Listing 4.4: fhachain/rest/src/.../service/SubjectRepository.java

```

1 public interface SubjectRepository extends
2   PagingAndSortingRepository<Subject, Long> {
3   List<Subject> findByName(@Param("name") String name);
4   List<Subject> findByShortName(@Param("shortName") String shortName);
5   List<Subject> findByExaminationOrderId(@Param("id")
6                                         Integer examinationOrderId); }
```

<sup>6</sup>Wird derzeit nicht verwendet, aber im [Ausblick](#) referenziert.

Spring Boot implementiert diese Methoden des Interfaces zum Durchsuchen von Fächern selbstständig. Auf diese Weise konnte ein prototypischer Java REST Service mit wenig zeitlichem Aufwand entstehen, der nach Möglichkeit in Zukunft weiter ausgebaut werden kann.

#### 4.1.4 Frontends

Die Frontends müssen mit den zuvor erklärten Backends kommunizieren können. Für den REST Service gibt es dazu breit verfügbare *HTTP*-Clients.

Hinsichtlich der für diese Arbeit relevanteren Smart Contracts, die in Abschnitt 4.1 auch als *Backends* klassifiziert wurden, funktioniert die Anbindung wie folgt:

Zur Kommunikation mit Smart Contracts wird sich mit einem Ethereum Knoten verbunden. Dieser bietet dazu eine *JSON-RPC* API an, die mit passenden Clients angesprochen wird [vgl. Eth18e]. Die Clients sind dabei Proxies, welche die Ethereum API implementieren. Sie sind für eine Vielzahl von Programmiersprachen verfügbar.

Als vollständige Einheit<sup>7</sup> betrachtet werden derartig zu Stande kommende Apps auch als *DApps* bezeichnet, eine Abkürzung für „dezentralisierte Applikationen“ [vgl. Eth16a]. Die Idee sei es, dass der nächste Meilenstein des Internets, das Web 3.0, ein dezentralisiertes Internet – mit einer *DApp* für jeden Anwendungsfall – ist [vgl. Eth16a]. Aus diesem Grund hat der Großteil der Bibliotheken, mit denen auf die *JSON-RPC* Schnittstelle zugegriffen wird, den Präfix „web3...“.

Zusätzlich dazu die benötigen Smart Contracts selbst auch eine Schnittstelle, die beschreibt, wie auf den Contract zugegriffen werden kann – sowohl intern als auch extern. Diese Schnittstellenbeschreibung heißt *Application Binary Interface* (kurz ABI) und enthält die Signaturen der Funktionen eines Smart Contracts, die dann auch *ABI-encoded* genannt werden [vgl. Eth18d, 129 f.]. Der folgende Text beschränkt sich auf Funktionsaufrufe von außerhalb, etwa der Zugriff durch ein Frontend, dem das ABI der benötigten Funktionen typischerweise bekannt sein muss. Dazu kommt das sogenannte JSON-ABI Format zum Einsatz [vgl. Eth18d, 134 f.], das beim Kompilieren mit dem Solidity Compiler bei Bedarf ausgegeben wird. Es wird dann als JSON-Datei auf Frontend-Seite abgespeichert.

---

<sup>7</sup>die Kombination aus Backend (⇒ Smart Contract) und Frontend

### Beispiel 4.2: JSON-ABI zu `newMember`

Die Funktion `newMember` aus Listing 4.2 wird JSON-ABI-encoded wie folgt beschrieben:

**Listing 4.5: fhachain/terminal/resources/contracts/.**

```
{
  "name": "newMember", "type": "function", // ...
  "inputs": [ {"name": "memberAddress", "type": "address"} ,
    { "name": "id", "type": "uint256"} ,
    { "name": "freeField", "type": "string"} ],
  "outputs": []
}
```

#### 4.1.4.1 Raspberry Pi NFC Terminal

Die Abb. 4.4 zeigt das Haupt-UI des Projekts, das *FHACChain Terminal*. Die Objekte des Bilds sind nummeriert:

1. Zwei *MIFARE Classic NFC* Tags; einer in Kartenformat, einer in Schlüsselanhänger-Form (siehe Konzeption, Abschnitt 3.3.6.2).
  2. Das NFC-Lesegerät, das mit dem Raspberry Pi kommuniziert (siehe Abschnitt 3.3.6.2).
  3. Der *Raspberry Pi Zero v1.3*, der alle Eingaben und Ausgaben behandelt. Das Lesegerät ist mit den GPIO-Pins des Einplatinencomputers verbunden. Zusätzlich wurde ein LCD-Bildschirm mit kapazitiven Buttons angeschlossen, damit eine Interaktion mit dem Endanwender stattfinden kann.
- Auf dem Bildschirm ist das Hauptmenü erkennbar, auf das zugegriffen werden kann, sobald sich ein Anwender mit dem NFC Tag authentifiziert hat. Die



LEDs rechts vom LCD-Bildschirm (3.1) sind als Ganzes betrachtet die sogenannte *Session-Bar*, die anzeigen, wie lange der Benutzer noch authentifiziert ist, falls die Karte vom Lesegerät entfernt wird. Auf dem Bild ist es an der Zeit, dass der Nutzer den NFC Tag wieder auf das Lesegerät legt, da nur noch etwa ein Viertel der Zeit verfügbar ist, ehe ein *Session-Timeout* eintritt. In diesem Fall würde das Hauptmenü wieder gesperrt und der Benutzer müsste sich neu authentifizieren. Insgesamt ist es wichtig, dass der Raspberry Pi in seiner Rolle als zentraler Controller *fehlertolerant* arbeitet, da z.B. Abstürze dazu führen könnten, dass private Informationen des Nutzers preisgegeben werden, falls es zu einem Standbild des LCD-Bildschirms kommt. Aus diesem Grund gibt es für das Menü, welches nach der Authentifizierung aufgebaut wird, eine Art Selbstzerstörungsmechanismus, damit *Information Leakage* (engl. für „Informationsleck“) begrenzt wird.

Das Terminal ist für alle FH-Mitglieder vorgesehen.

Es unterstützt die speziellen (abgeleiteten) Akteure *Student* und *Klausuraufsicht*, für die folgende Anwendungsfälle vorgesehen sind:

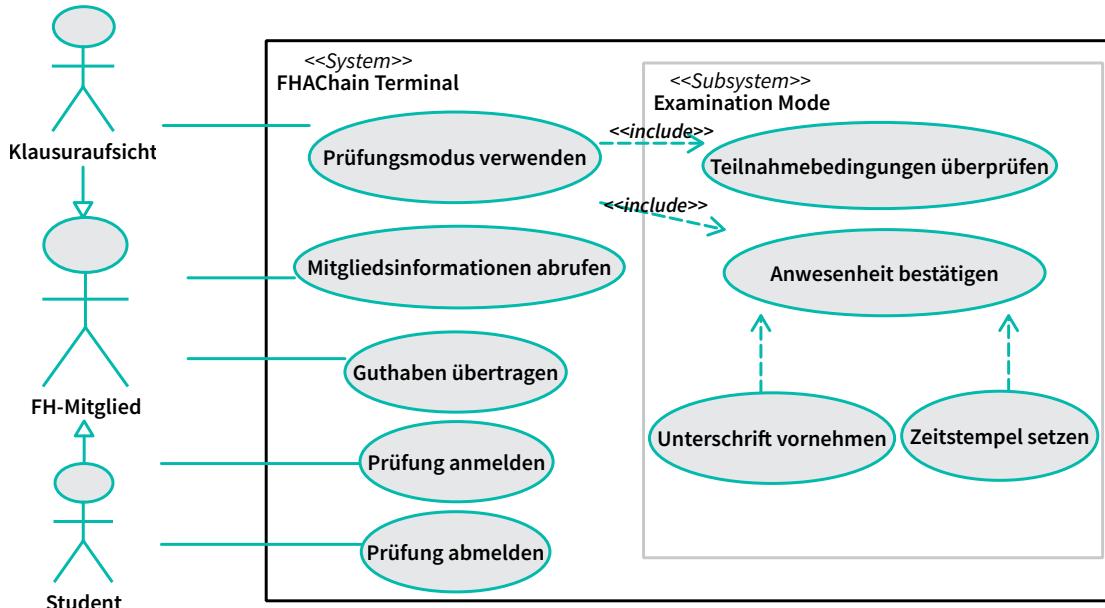


Abbildung 4.5: Use Case Diagramm des *FHACChain* Terminals.

ERSTELLT VIA: Umbrello UML Modeler v2.24.3 (Linux)

Das Terminal greift lesend auf den *MemberStorage* Contract zu und hat eine starke

Kopplung an den *ExaminationService*-Contract, auf den sowohl lesend (z.B. beim Abfragen der Prüfungen, die angemeldet werden können) als auch schreibend (z.B. beim Anmelden einer Prüfung) zugegriffen wird (siehe Abb. 4.3).

Aus Abb. 4.5 kann man ableiten, dass nur die Klausuraufsicht autorisiert ist, den Prüfungsmodus zu verwenden. Der *MemberStorage*-Contract verfügt (wie u.a. im UML-Diagramm dargestellt) dazu über Rollen, die den Mitgliedern zugewiesen werden können. Nur wenn das authentifizierte Mitglied über eine der Rollen DOCENT, PROFESSOR oder EXAMINATION\_OFFICE verfügt, kann auf den speziellen Modus des Terminals zugegriffen werden. Der Modus ist für den noch folgenden Geschäftsprozess in Abschnitt 4.3.1 relevant.

Das NFC-Format aus Abschnitt 3.3.6.2 wurde strikt nach der Spezifikation implementiert, so dass der Speicherinhalt des Tags dem der Tabelle 3.4 entspricht. Damit auch neue Tags produziert werden können, gibt es für Administratoren zusätzlich die Möglichkeit, das sogenannte *FHACard Tool* zu nutzen. Das konsolen-basierte Programm wird auch verwendet, um den gesamten Speicherinhalt eines Tags auszugeben:

```

SECTOR: 0
0 @ 0x00 [D]: 60 BA A1 E0 9B 88 04 00 C2 05 00 00 00 00 00 13 | `°;à Å
1 @ 0x01 [D]: AC 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ~
2 @ 0x02 [D]: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
3 @ 0x03 [T]: 00 00 00 00 00 00 78 77 88 C1 00 00 00 00 00 00 | xw Å

SECTOR: 1
0 @ 0x04 [D]: 23 09 0A 27 F0 27 16 DB 82 BD 30 34 B9 FA 85 A1 | #      \n'ð'0 %0
1 @ 0x05 [D]: 5A E4 22 2C 00 00 00 00 00 00 00 00 00 00 00 00 | Zä",
2 @ 0x06 [D]: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
3 @ 0x07 [T]: 00 00 00 00 00 00 7F 07 88 40 00 00 00 00 00 00 | @ @

SECTOR: 2
0 @ 0x08 [D]: 67 41 41 41 41 41 42 61 79 35 5F 6C 53 35 59 50 | gAAAAABay5_lS5YP
1 @ 0x09 [D]: 77 52 64 30 73 6F 31 45 73 51 78 64 58 67 52 36 | wRd0solEsQxdXgR6
2 @ 0x0A [D]: 7A 77 6D 4A 72 55 39 30 57 5F 65 2D 50 4B 42 73 | zwmJrU90W_e-PKRs

```

**Abbildung 4.6:** Dump eines Tags mit dem *FHACard Tool*.

Abb. 4.6 zeigt von oben nach unten den Inhalt der Sektoren 0 bis 2. Jeder Block wird sowohl links in Hexadezimaldarstellung als auch auf der rechten Seite in *ASCII* (nach dem |-Trennzeichen) ausgegeben. Die hellblauen Markierungen sind die FH-spezifischen Daten des Tags (siehe Abschnitt 3.3.6.2):

1. **Sektor 0 Block 1:** Metadaten-Block, der als Identifizierungsmerkmal mit 0xAC

anfängt und die Version des Kartenformats enthält (0x01).

2. **Sektor 1 Block 0 bis 1:** Die FH-Mitglied Ethereum Adresse  $A(p_r)_{0..19}$ , welche mit 0x23 anfängt und mit 0x2C endet (0x2309...222C).
3. **Sektor 2 Block 0 bis n:** Der verschlüsselte Private Key  $EC(p_r)_{0..x}$  mit  $x < 896$  (restliche zur Verfügung stehende Bytes).

Sowohl das *FHACChain Terminal* als auch das *FHACard Tool* wurden aufgrund der guten hardwarenahen Unterstützung in der Programmiersprache *Python* implementiert.

Es existiert bereits eine Python-Bibliothek namens *web3.py*, die zur Anbindung des Terminals an die Smart Contracts verwendet werden konnte.

Die Smart Contracts werden mithilfe der Library wie folgt instanziert:<sup>8</sup>

**Listing 4.6: fhachain/terminal/src/core/services/fhachain web3.py**

```
1 contract = self._w3_eth.contract(
2     address=self.normalize_address(contract_interface['address']),
3     abi=contract_interface['abi'],
4     ContractFactoryClass=ConciseContract)
```

Während das `contract_interface` von einer JSON-Datei eingelesen wird, die neben dem *ABI* auch die Adresse des Smart Contracts enthält. Zuvor hat demnach ein Deployment mittels einer *Contract-creating Transaction* stattgefunden (siehe Abschnitt 2.1.4.3). Unter der Annahme, dass es sich bei dem instanzierten `contract` um den *MemberStorage*-Contract handelt, können Informationen über ein Mitglied (mit einem gegebenen NFC Tag *fhacard*) wie folgt erhalten werden – siehe auch UML-Diagramm 4.3:

**Listing 4.7: fhachain/terminal/src/core/services/fhachain web3.py**

```
1 def get_account_info(self, fhacard: FHACard):
2     adr = self.normalize_address(fhacard.address)
3     fh_id = self.contract.getId(adr)
4     roles = [list(AccountInfo.Role)[idx]
5             for idx, s in enumerate(self.contract.getRoleStates(adr)) if s]
6     description = self.contract.getFreeField(adr)
7     return AccountInfo(address, fh_id, roles, description)
```

Wie beispielsweise in Zeile 3 erkennbar, kann der *MemberStorage*-Contract wie ein reguläres Objekt behandelt werden. Da Python eine dynamisch-typisierte Sprache ist, besteht

<sup>8</sup>Dies ist nicht zu verwechseln mit dem Deployment eines Smart Contracts: Die Instanziierung erfolgt an dieser Stelle, damit auf die erforderlichen Funktionen zugegriffen werden kann.

auch keine Notwendigkeit ein Interface o.Ä. zu implementieren. Wie in Listing 4.6 beschrieben ist, reicht es vollkommen aus, die Contract-Adresse und das *ABI* anzugeben, um Funktionen eines Smart Contracts aufzurufen.

Es kostet kein Ether, wie in der obigen Funktion lediglich *lesend* auf den Smart Contract zuzugreifen (daher wird auch nicht etwa der Private Key des `fhabard` Tags angegeben). Da mit `web3.py` Contracts wie reguläre Objekte angesprochen werden bleibt in dem Listing 4.7 als einzige Auffälligkeit das Konstrukt in Zeile 4 und 5. Dies ist eine sogenannte *List Comprehension*, die in kompakter Darstellung eine Liste aus Rollen zusammenstellt (das *Role*-Enum des UML-Diagramms 4.3).

Um verstehen zu können, warum das Konstrukt in dieser Form eingesetzt wird, ist es notwendig, das Solidity-Gegenstück – nämlich die Implementation der Funktion im Smart Contract – zu betrachten:

**Listing 4.8: fhachain/smart-contracts/contracts/MemberStorage.sol**

```

1 function getRoleStates(address memberAddress) public view returns (bool[])
2 {
3     bool[] memory roleStates = new bool[](ROLE_SIZE);
4     for (uint8 i = 0; i < ROLE_SIZE; i++) {
5         roleStates[i] = hasRole(memberAddress, Role(i));
6     }
7     return roleStates;
8 }
```

Auch zum Verständnis beitragend ist Listing 4.2, speziell die Zeilen 9 und 13. In Listing 4.8 Zeile 4 wird zunächst ein Array im flüchtigen Speicher der EVM angelegt. Die Größe `ROLE_SIZE` dieses Arrays ist in Solidity (prinzipbedingt), wenn er einmal alloziert wurde, nicht mehr änderbar [vgl. Eth18d]. Aufgrund dessen wurde die Entscheidung getroffen, einen Array mit booleschen Werten zurückzugeben, der immer eine Größe von 8 hat (die Rollenzahl `ROLE_SIZE`).

Zurück zum Python Quelltext, Listing 4.7. Auf dieser Seite wird mit dem zuvor angesprochenen *List Comprehension*-Konstrukt das Mapping auf den jeweiligen `AccountInfo.Role` Wert vorgenommen. In Solidity werden im ABI (auch in der zum Schreibzeitpunkt neusten Version 0.4.25) keine Enums unterstützt [vgl. Eth18d, S. 176]. Daher sind die Konstanten des Solidity-Enums (in Listing 4.2 Zeile 6 bis 8) im Endeffekt im Python-Enum `AccountInfo.Role` auf Seite des *FHACChain Terminals* dupliziert.

#### 4.1.4.2 Administrator Web UI

Als zusätzliche Benutzeroberfläche zur Demonstration der Vielfältigkeit an Anbindungs möglichkeiten für Smart Contracts wurde das *Administrator Web UI* entwickelt, welches ein in der Programmiersprache *TypeScript* geschriebenes *Angular* Webprojekt ist. Es nutzt *web3.js*, die JavaScript API Ethereums.<sup>9</sup> Das UI erfüllt nach derzeitigem Stand lediglich eine minimale Funktionalität: Es zeigt die Mitglieder der Fachhochschule Aachen im *MemberStorage* Smart Contract an, indem zuerst der aus Listing 4.2 bekannte *addresses* Array iteriert wird, um dann aus dem *addressToMember* Mapping die in der *Member*-Datenstruktur gespeicherten Informationen zu erhalten (siehe auch Abb. 4.3).

Da des am ehesten für Administratoren und Geschäftsprozess-Entwickler der *FHACChain* relevant ist, wurde es im Rahmen der in Abschnitt 4.1.1 vorgenommenen Installation in das *Camunda Cockpit* in Form eines Plugins integriert.

Es erfordert einen Web3-fähigen Browser (z.B. *Mist* oder ein Standard-Browser mit dem *MetaMask*-Plugin), damit ein sogenannter *Web3-Provider* injiziert und in der Angular Applikation auf ihn zugegriffen werden kann. Falls dieser Provider nicht verfügbar ist, so gibt es zwar einen *Fallback* auf <http://localhost:7545>, jedoch muss dann auch auf genau diesem Port ein Ethereum Knoten laufen, damit hinter der Fassade mittels *JSON-RPC* mit ihm kommuniziert werden kann.

## 4.2 Integration

In dieser Sektion geht es darum, wie die Geschäftsprozesse an Ethereum angebunden werden können. Dazu wird zunächst ein Weg mittels einer Kombination aus *Code Generation* und eines *Camunda Java Delegates* vorgestellt. Danach wird gezeigt, wie dies ohne den zusätzlichen Schritt erfolgen kann – nämlich, indem lediglich der Camunda Modeler genutzt wird.

---

<sup>9</sup>Da TypeScript zu JavaScript kompiliert wird, können auch JS-Bibliotheken eingebunden werden.

#### 4.2.1 „Hallo FHACChain“-Prozess: Erste Anbindung

Zu Beginn wird folgender Geschäftsprozess betrachtet, mit dem Mitgliederinformationen aus dem *MemberStorage-Contract* angezeigt werden sollen:

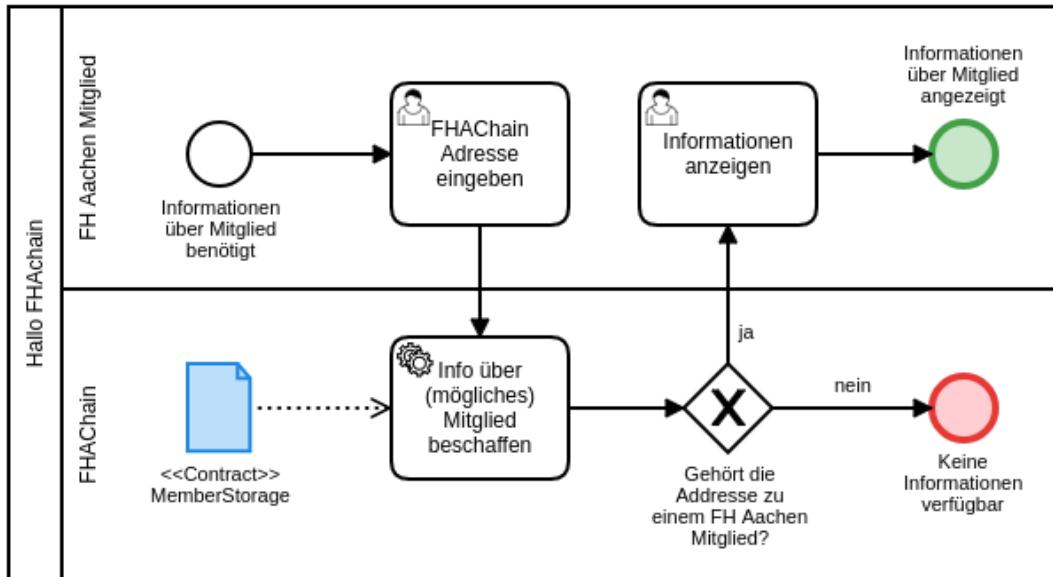


Abbildung 4.7: Exemplarischer „Hallo FHACChain“-Geschäftsprozess.

In der *User Task* „FHACChain Adresse eingeben“ kann ein Mitglied der Fachhochschule im *Task Form* eine Ethereum Adresse eingeben. Daraufhin werden Informationen über das Mitglied aus dem *MemberStorage-Contract* beschafft und angezeigt (die ID und eine Beschreibung, aus dem `freeField` – siehe auch Listing 4.2, Zeile 15 u. 24), falls es sich bei der eingegebenen Adresse um eine FH-Mitglieds Adresse handelt.

Die *Service Task* wurde mithilfe eines *Java Delegates* implementiert, das folgende Kernfunktionalität hat:

**Listing 4.9: fhachain/business-proc\*/exec\*.../MemberStorageDelegate.java**

```

1 boolean isMember = contract.exists(memberAddress).send();
2 if (isMember) {
3     String id = contract.getId(memberAddress).send().toString();
4     exec.setVariable("memberId", id);
5     String freeField = contract.getFreeField(memberAddress).send();
6     delegateExecution.setVariable("memberFreeField", freeField);
7 }
8 delegateExecution.setVariable("isMember", isMember);

```

---

Hier wird die Java-Variante von Web3 verwendet, *web3j*, um die ermittelten Informationen in Form von Execution-Variablen zu setzen, damit sie später im HTML-Form der *User Task* „Informationen anzeigen“ dargestellt werden können.

Anders als in der Python Anbindung des *FHACChain Terminals* (siehe 4.1.4.1) ist hier ein zusätzlicher Schritt notwendig gewesen, um die Funktionen des *MemberStorage*-Contracts gleichwertig<sup>10</sup> zugreifen zu können: Mit einem zusätzlichen separaten erhältlichen Kommandozeilen-Tool wurde der clientseitige Java-Code zum Contract als eine Wrapper-Klasse automatisch generiert, indem (lediglich) das ABI als Datei angegeben werden musste. Daraus wurden Funktionen in folgender Form generiert:

**Listing 4.10: fhachain/business-proc\*/exec\*.../MemberStorage.java**

```

1 public RemoteCall<Boolean> exists(String memberAddress) {
2     final Function function = new Function("exists", Arrays.<Type>asList(
3         new org.web3j.abi.datatypes.Address(memberAddress)),
4         Arrays.<TypeReference<?>>asList(new TypeReference<Bool>() {}));
5     return executeRemoteCallSingleValueReturn(function, Boolean.class);
6 }

```

---

Auf die Definition der `exists`-Methode aus Listing 4.11 wird in Listing 4.9 in Zeile 1 zugegriffen.

Dass der Schritt der *Code Generation* überhaupt notwendig ist, lässt sich letzten Endes darauf zurückführen, dass Java – anders als Python – eine *statisch-typisierte* Programmiersprache ist. Da bei jeder Änderung der Funktionssignatur eines Smart Contracts auch immer wieder das Kommandozeilen-Tool von *web3j* *manuell* aufgerufen werden müsste, wurde entschieden, diesen Schritt nach Möglichkeit entfallen zu lassen – schließlich ließe sich dadurch die Prozessentwicklung vereinfachen, die im besten Fall genauso

---

<sup>10</sup>Im Sinne des Komforts bei der Entwicklung „gleichwertig“.

rationalisiert werden sollte, wie die Geschäftsprozesse selbst. Dieser Schritt wird im nächsten Abschnitt beschrieben.

### 4.2.2 Web3-Connector

Der Web3-Connector ist die einzige noch übrige Komponente der *FHACChain*, welche zwar der Abbildung des Technologie-Stacks 4.1 vorkommt, jedoch noch nicht weiter beschrieben wurde.

Es handelt sich hierbei um einen im Rahmen dieser Arbeit implementierten *Custom Camunda Connector*, der ähnlich wie die von *Camunda BPM* bereitgestellten Konnektoren für *SOAP* (*soap-http-connector*) und *REST* (*http-connector*) aufgebaut ist, welche praktisch nur wenige bis keine Programmierkenntnisse erfordern. Aus Sicht der Entwicklung des *Web3-Connector* wird das Interface

`org.camunda.connect.spi.Connector` implementiert. Die Klasse `Web3Connector` wird dann mithilfe des von Camunda verwendeten *Java ServiceLoader*-Mechanismus (der meist bei Java Applikationen eingesetzt wird, die Plugins verwenden) erkannt, so dass er der Engine als zusätzlicher Camunda Connector zur Verfügung steht – alle Konnektoren lassen sich etwa in einem Delegate wie folgt zur Laufzeit abfragen [vgl. Cam18e]:

```
import org.camunda.connect.Connectors;  
// ...  
System.out.println(Connectors.getAvailableConnectors());
```

Für die Contract-Anbindung als solche wird – wie auch schon im vorigen Abschnitt – die Java Bibliothek *web3j* verwendet, jedoch ist dies für den Endanwender nicht unbedingt erkennbar. Dass wie in Abschnitt 4.2.1 beschrieben keine Code Generation mehr benötigt wird, wurde wie folgt gelöst:

1. Statt mithilfe des gesamten ABIs eine Klasse zu erzeugen, wird von nun an die Sicht auf einzelne Funktionen des Smart Contracts beschränkt – es werden nur die Funktionen betrachtet, die auch tatsächlich im Prozess (bzw. Modell) verwendet werden.
2. Die Smart Contract Funktionen (siehe Listing 4.11, Zeile 2 bis 4) werden nicht mittels Java, sondern vielmehr via *Groovy* zur Laufzeit anstatt zur Übersetzungszeit instanziiert. Insbesondere ist dies zielführend, da dadurch nur der *Camunda*

Modeler verwendet werden kann und keine Delegates mehr nötig sind.

#### 4.2.2.1 Verwendungsmöglichkeiten

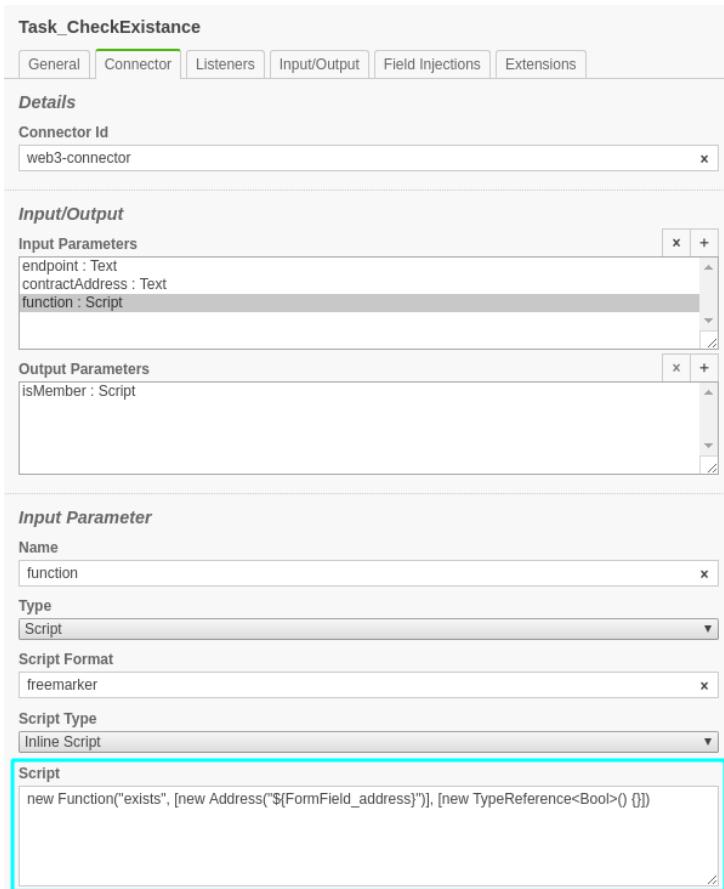


Abbildung 4.8: Verwendung des Web3-Connectors.

gene *Groovy-Shell* aufgerufen, die die Funktion dynamisch instanziert. Wie man im hellblauen Bereich erkennen kann, wird lediglich die Signatur der Smart Contract Funktion `exists` angegeben (speziell der Name, die Eingabeparameter und der in diesem Fall boolesche Rückgabewert). Zusätzlich zu den standardmäßigen *Contract Calls* (`contract-call`) unterstützt der Connector noch folgende Funktionen, die im `type` Input Parameter angegeben werden können.<sup>11</sup>

<sup>11</sup> Alle möglichen Parameter Eingabe- und Ausgabewerte sind im Ordner/Paket `fhachain/web3-connector/src/main/java/.../web3/request/parameters` in

Der Custom Connector lässt sich – wie in Abb. 4.8 gezeigt wird – im Camunda Modeler konfigurieren bzw. verwenden. Es lassen sich bei den *Input Parametern* die benötigten Felder ablesen, um den sogenannten *Contract Call* durchführen zu können. Ein Contract Call ist ein Funktionsaufruf, um ausschließlich lesend auf einen Smart Contract zuzugreifen. Im hellblau markierten Bereich ist das Feld der `function` in der Detailansicht – das Skript ist für Camunda zwar ein *Freemarker*-Skript, damit zusätzlichliche Execution-Variablen substituiert werden können. Intern (im Connector) wird der String allerdings wie ein *Groovy* Skript behandelt und eine ei-

- **contract-call-wait:** Sorgt dafür, dass der Connector wiederholt eine angegebene Contract-Funktion aufruft und überprüft, ob dieser eine angegebene `waitCondition` (ein weiteres Groovy-Script) erfüllt – so kann beispielsweise darauf gewartet werden, dass ein bestimmtes Mitglied erstellt wird. Sollte die Bedingung `true` zurückgeben, so wird weiter gewartet, andernfalls wird die Schleife verlassen und der Rückgabewert der ausgeführten Funktion zurückgegeben.
- **contract-event-watch:** Dies ist die Alternative zu dem obigen Ansatz, für die folgendes Hintergrundwissen nötig ist: In der Ethereum Blockchain gibt es eine *Log*-Datenstruktur, zu der in Solidity neue Einträge mittels *Events* hinzugefügt werden können. Dadurch ist es möglich, dass ohne viel Ressourcen-Bedarf auf Ereignisse eines Smart Contracts gefiltert (und gewartet) werden kann [vgl. Eth18d, S. 21]. Dies ist effizienter als der obige Mechanismus, setzt aber voraus, dass die Events im Solidity-Code ausgelöst werden.<sup>12</sup> Im UML-Diagramm aus Abb. 4.3 sind alle möglichen Events rechts neben dem *ExaminationService*-Contract gelistet, da sie alle von ihm emittiert werden.

Falls die in Abb. 4.8 dargestellte Verwendung nicht erwünscht ist, so lässt sich der *Web3Connector* auch regulär mittels Java-Code nutzen, beispielsweise wie folgt:

**Listing 4.11: /fhachain/web3\*/src/test/.../integr\*/TestContractEventWatch**

```

1 public void shouldWatchContractEvent() {
2     ConnectorResponse response = new Web3Connector(Web3Connector.ID) .
3         createRequest()
4         .endpoint(Setup.ENDPOINT)
5         .asContractEventWatch()
6         .contractAddress(Setup.Contract.EXAMINATION_SERVICE.getAddress())
7         .event(new EventFilter("OnStudentExamRegistration",
8             Arrays.<TypeReference<?>>asList(new TypeReference<Address>() {}),
9             Arrays.<TypeReference<?>>asList(new TypeReference<Uint256>() {}),
10            Collections.singletonList(
11                new Address(Setup.FHACChainMember.S2.getAddress())))
12        .execute(); //...
13        Assert.assertEquals(
14            response.getOutputs().get(0).toString().toLowerCase(),
15            Setup.FHACChainMember.S2.getAddress().toLowerCase());
16    }

```

Code-Form dokumentiert.

<sup>12</sup>Der obige Ansatz hat diese Abhängigkeit zum Code nicht.

Dieser Code-Ausschnitt demonstriert, wie der `contract-event-watch`-Type genutzt werden kann, um auf das Eintreten eines mit einem Solidity emittierten Events namens `OnStudentExamRegistration` zu warten. Dieses und weitere Events sind für die optimierte Variante des Geschäftsprozesses „Klausur durchführen“ in Abschnitt 4.3.1 noch relevant. In Solidity wird es wie folgt etwa in der Funktion `registerExam` des `ExaminationService`-Contract ausgelöst:

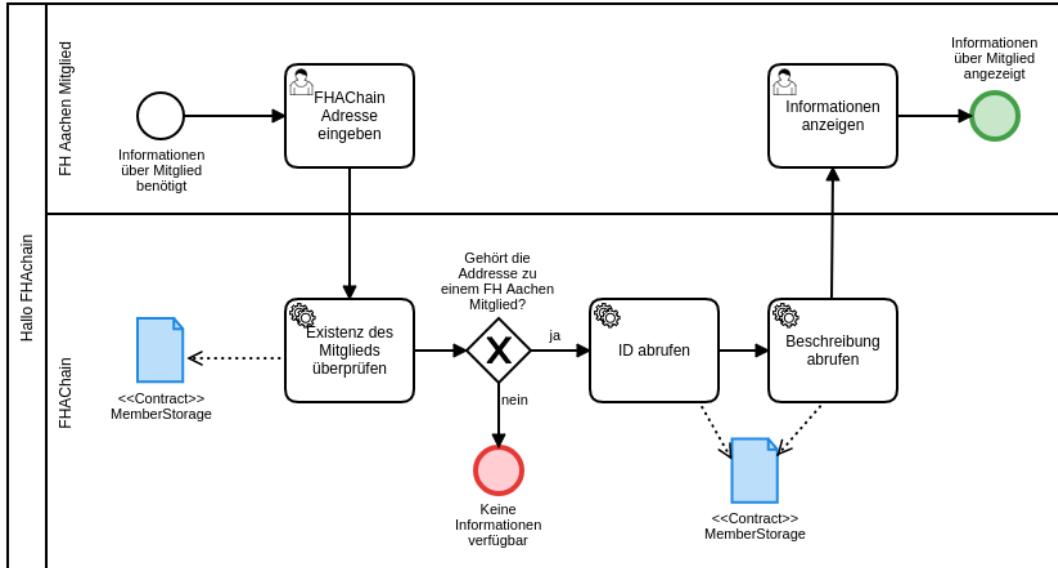
**Listing 4.12: fhachain/smart-contracts/contr\*/ExaminationService.sol**

```
emit OnStudentExamRegistration(msg.sender, examId);
```

Dementsprechend wird in der dahinter liegenden Log-Datenstruktur die Absender-Adresse (die einem Studenten gehören muss, denn nur Studierende dürfen sich für Prüfungen anmelden) und die Prüfungs-ID abgspeichert.

#### 4.2.2.2 Optimierter „Hallo FHACChain“-Prozess

Optimiert mit dem Web3-Connector sieht die `MemberStorage`-Contract-Integration des Prozesses aus Abschnitt 4.2.1 wie folgt aus:



**Abbildung 4.9:** Optimierter „Hallo FHACChain“-Geschäftsprozess mittels Web3-Connector.

Wie in Abb. 4.9 erkennbar ist, sind die Tasks im Vergleich zu Abb. 4.7 feiner, d.h. die Granularität der Tasks hat zugenommen. Dadurch, dass der Web3-Connector verwendet wird, wird der Prozess-Entwickler dazu forciert, mehrere Funktionalitäten nicht mehr in einem Delegate zu kapseln (wie in der Service Task „Info über (mögliches) Mitglied beschaffen“, in der sowohl der Mitgliedsstatus als auch Informationen abgefragt werden). Der Vorteil: Der *Flow of Control* wird klarer – im Prozess der Abb. 4.9 ist nämlich direkt ersichtlich, dass zuerst die Existenz des Mitglieds überprüft wird und danach erst die zusätzlichen Informationen abgerufen werden, respektive die ID und die Beschreibung. Dennoch soll dies nicht heißen, dass es per se immer negativ ist, mit einem Delegate mehrere Funktionen gleichzeitig auszuführen; im Gegenteil: In einigen Fällen ist es sogar im Sinne der Abstraktion notwendig. Für den minimalen Prozess oben ist es aber vorteilhaft, die Teilschritte (Tasks) klar und deutlich zu modellieren.

### ▼ Tutorial 4.5: „Hallo FHACChain“-Prozess testen

Die genannten Prozesse lassen sich wie folgt testen:

**1. Prozess Deployment durchführen:**

Im VM-Terminal `/vagrant/scenarios/checkout.sh hello` aufrufen und einen kleinen Moment warten. Das Checkout-Skript kommt auch in den weiteren Tutorials zur Automatisierung zum Einsatz.

**2. Prozess starten und Aufgaben bearbeiten:**

Bei der Camunda Tasklist-Applikation (<http://192.168.42.42:8181/camunda/app/tasklist/>) mittels Benutzer `demo`, Passwort `demo` einloggen.<sup>a</sup> Dann in der Aktionsleiste oben auf „Start process“ klicken und den Prozess „FHACChain Greeter ...“ auswählen.

In der Tasklist können links unter „All Tasks“ die Eingaben vorgenommen werden.

<sup>a</sup>Die Login-Daten sind für alle weiteren Tutorials gleich.

## 4.3 Optimierte Geschäftsprozesse

In dieser Sektion werden Prozesse vorgestellt, die im Rahmen dieser Bachelorarbeit mithilfe der Smart Contracts optimiert wurden. Für alle Zugriffe auf die Contracts wird stets der zuvor vorgestellte Web3-Connector eingesetzt.

### 4.3.1 Klausur durchführen

Der Prozess wurde als Ist-Prozess in Abschnitt 3.1.4 vorgestellt, auf der folgenden Seite (Abb. 4.10) ist die verbesserte Version, die das *FHACChain Terminal* verwendet. Das Terminal wird vor der eigentlich Klausurdurchführung (wie in den Use Cases aus Abschnitt 4.1.4.1 definiert) zum Anmelden der Prüfung von Studierenden verwendet – dabei werden im Prozess ausschließlich gültige Anmeldungen erfasst, da der Smart Contract in einem Zug bei der Anmeldung in der Funktion `registerExam` die Teilnahmebedingungen überprüft (bspw., ob das zum Fach dazugehörige Praktikum bestanden wurde). Unmittelbar vor der Klausur, beim Einlass in den Raum, kann das Subsystem (der „Prüfungsmodus“) des Terminals aus Abb. 4.6 von der Klausuraufsicht verwendet werden, um als eine Art Einlasskontrolle zu agieren. In der Task „Identität mit Ausweis und FH-Karte abgleichen“ wird eine (manuelle<sup>13</sup>) Identitätskontrolle vorgenommen. Danach wird die Karte auf das Terminal gelegt und die Aufsicht wartet auf ein „gültiges Ticket“, das letztendlich das Event *OnStudentExamAppearance* ist. Dieses Event wird nur ausgelöst, wenn die Teilnahmebedingungen weiterhin erfüllt sind (d.h. auch, dass sich der Student vorher nicht abgemeldet haben darf). Vor dem Auslösen des Events wird die Anwesenheit von der Aufsicht bestätigt; sie sendet mit ihrem eigenen Account<sup>14</sup> die Transaktion `registerAppearance` (wodurch auch implizit der Use Case „Unterschrift vornehmen“ zu Stande kommt). In dieser Funktion wird auch ein Zeitstempel gesetzt, damit kryptografisch sicher auch später nachvollzogen werden kann, wann der Student zur Klausur erschienen ist. Im weiteren sequentiellen Prozessverlauf wird nach der Klausur-Austeilung schließlich die Klausur geschrieben und nach Ablauf der Zeit abgegeben. Sie wird allerdings nur von der Aufsicht erfolgreich angenommen, wenn das Deckblatt auch den Namen des Studierenden trägt (und nicht etwa einen anderen).

Es wurde eine Möglichkeit gesucht, über das *FHACChain Terminal* mit dem Prozess zu kommunizieren – mit den in Abschnitt 4.2.2.1 beschriebenen Events wurde ein solcher Mechanismus gefunden. Ein Teil der Geschäftslogik ist nicht im Geschäftsprozess selbst, sondern im *FHACChain Terminal* bzw. vielmehr im Smart Contract *ExaminationService*.

---

<sup>13</sup>Zur besseren Ausführbarkeit des Prozesses wurde eine anschaulichere User Task verwendet.

<sup>14</sup>Sie musste sich vorher, um den Prüfungsmodus zu betreten, schließlich mithilfe des NFC-Tags anmelden.

## 4 Die FHACChain

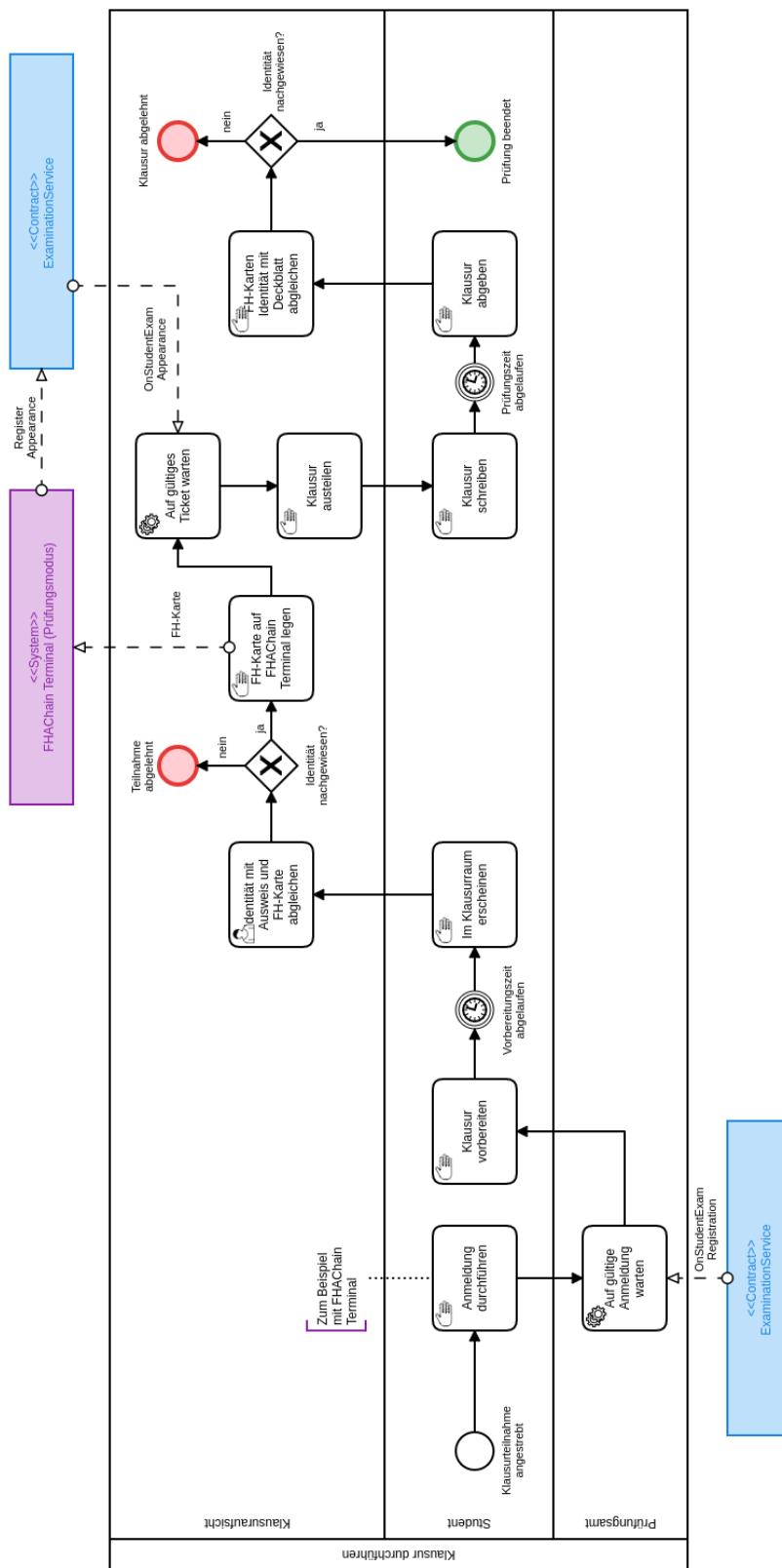


Abbildung 4.10: Der verbesserte Geschäftsprozess „Klausur durchführen“.

ERSTELLT VIA: Camunda Modeler v1.11.3 (Linux)

Alle in Abschnitt 3.1.4.2 dargelegten Verbesserungen wurden im ausführbaren Prozess der Abb. 4.10 vorgenommen:

1. Alle vier Tasks, die realistisch betrachtet digitalisiert werden konnten, wurden automatisiert oder entfallen anderweitig. Es hat eine Reduktion der manuellen Tasks von 11 auf 7 stattgefunden:
  - Es wird kein Testatbogen mehr benötigt, lediglich die FH-Karte und ein Lichtbildausweis zum Identitätsabgleich. Die manuelle Task „Testatbogen überprüfen“ entfällt und wird vom Terminal vorgenommen.
  - Es muss keine Liste mehr herumgereicht werden („Liste zum Unterschreiben hinlegen“)...
  - ↴ ...daher entfällt für den Studenten auch das „Namen suchen und unterschreiben“.
  - Ferner entfällt das „Einpacken und gehen“ während der Klausur. Nicht zugelassene Studierende müssen den Raum vor dem Schreibvorgang der Klausur bereits verlassen.
2. Der Prozess wurde so modelliert, dass er vollständig sequenziell abläuft. Der Student muss nicht mehr während der Klausur den „Namen suchen und unterschreiben“. Im Gegensatz dazu kann er mit voller Konzentration unterbrechungsfrei die Klausur schreiben.
3. Der Störfaktor der Geräusche durch die manuelle Task „Einpacken und gehen“ entfällt aufgrund der fehlenden Parallelisierung.

Zusammengefasst konnte der Prozess durch die Mittel, die durch die Technologien Blockchain und NFC zur Verfügung stehen, deutlich effizienter werden (ca. 36% weniger manuelle Tasks). Ferner wurden alle störenden Faktoren eliminiert und auch die Parallelisierung sowie die zum Teil fehleranfällige Überprüfung des Testatbogens aufgehoben. Für die Technologie Ethereum sprechen in diesem Prozess die Verbesserungen bezüglich der Sicherheit (siehe Abschnitt 3.3.1.3 und die Anforderung Sicherheit).

#### ▼ Tutorial 4.6: Prozess testen (ohne FHACChain-Terminal)

Der Prozess lässt sich auch ohne die für das Terminal benötigte Hardware testen:

##### 1. Prozess Deployment durchführen:

Im VM-Terminal `/vagrant/scenarios/checkout.sh exam` aufrufen und einen kleinen Moment warten.

##### 2. Prozess starten und Aufgaben bearbeiten:

Bei der Camunda Tasklist-Applikation (<http://192.168.42.42:8181/camunda/app/tasklist/>) anmelden. Dann in der Aktionsleiste oben auf „Start process“ klicken und den Prozess „Exam“ auswählen.

Ohne das *FHACChain-Terminal* wird die Anmeldung (auf die der Prozess nun wartet) wie folgt vorgenommen:

- Öffnen der *Truffle*-Entwicklerkonsole: `cd /opt/fhachain-contracts/ && truffle console`
- Folgende 3 JavaScript Befehle zum Anmelden eines Studenten eingeben (Achtung, die letzten beiden Zeilen ist ein Einzeiler):

```
1 var accounts;
2 web3.eth.getAccounts(function(err, r) { accounts = r; });
3 ExaminationService.deployed().then(function(instance){return
    instance.registerMeForExam(51104, {from:accounts[3]}); })
```

##### 3. Nach etwa 15 bis 30 Sekunden sollte der Prozess das Event erkennen und starten.

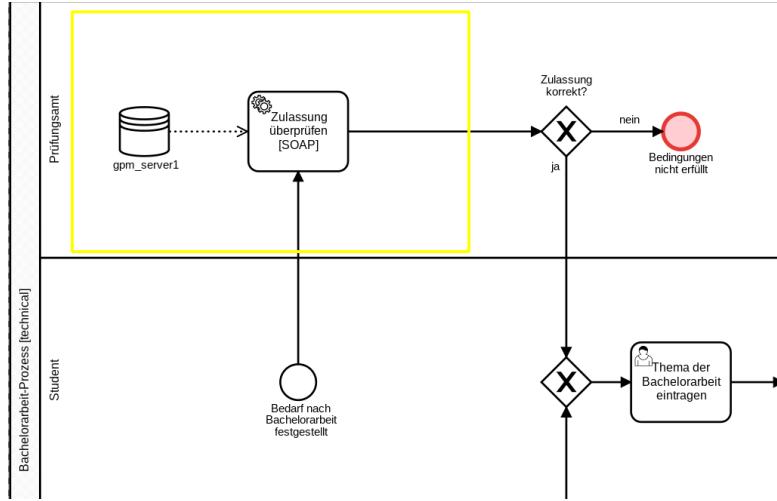
Da es eine leichte Verzögerung aufgrund der „Vorbereitungszeit“ gibt, dauert es etwas, bis die erste Aufgabe in der Tasklist erscheint.

### 4.3.2 Bachelorarbeit schreiben

Das Beispiel in diesem Abschnitt zeigt, wie in einen bereits existierenden ausführbaren Geschäftsprozess eine Blockchain-Integration vorgenommen werden kann.

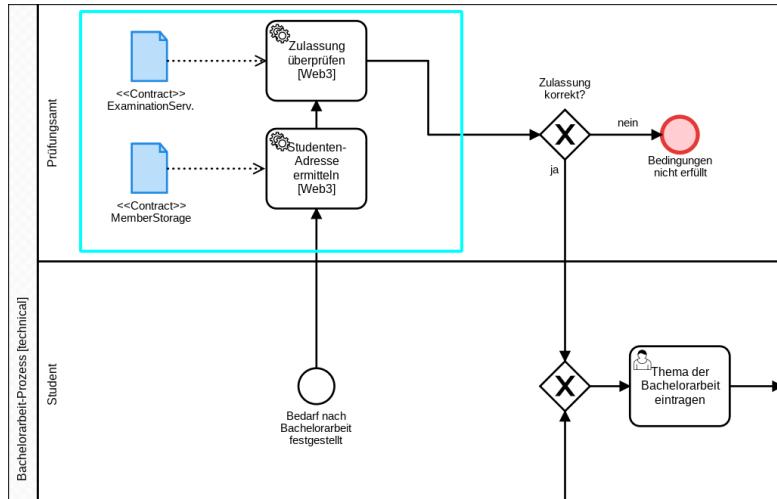
Betrachtet wird ein Ausschnitt des „Bachelorarbeit schreiben“-Prozesses. Zuvor (Abb. 4.11) wurde die Zulassungsüberprüfung zur Bachelorarbeit mithilfe eines SOAP-Services vorgenommen (gelb umrandet), der jedoch durch eine Anbindung an den bekannten *ExaminationService* ersetzt werden soll.

#### 4 Die FHACChain



**Abbildung 4.11:** Ausschnitt: „Bachelorarbeit schreiben“-Prozess ohne Contract-Anbindung.

Die erfolgreiche Migration wird in Abb. 4.12 gezeigt (hellblaue Umrandung):



**Abbildung 4.12:** Ausschnitt: „Bachelorarbeit schreiben“-Prozess mit neuer *ExaminationService*-Anbindung.

Damit die Integration funktionieren konnte, wurde – wie im UML-Diagramm in Abb. 4.3 sichtbar – ein *BachelorThesis*-Contract erstellt, der das *Exam*-Interface implementiert. Ein Smart Contract ist nach den Grundsätzen einer Blockchain nach dem Deployment nicht mehr änderbar. Im *ExaminationService*-Contract wurde aber ein Pattern verfolgt, mit dem der Contract *upgradeable* (dt. „erweiterungsfähig“) ist, da zu jeder Zeit neue

Contracts zu ihm hinzugefügt werden können, die das *Exam*-Interface implementieren. Die Contracts werden schließlich in der `idToExamAddress`-Statusvariablen persistiert. Die *Exam*-Adressen können dabei explizit mittels *Cast* zu einem *Exam*-Interface umgewandelt werden, damit auf die Funktionen des Interfaces zugegriffen und die Zulassung überprüft werden kann.

### ▼ Tutorial 4.7: Prozess testen

Der dazugehörige vollständige Prozess kann wie folgt ausprobiert werden:

1. **Prozess Deployment durchführen:**

Im VM-Terminal `/vagrant/scenarios/checkout.sh thesis` aufrufen und einen kleinen Moment warten.

2. Nachdem der Prozess gebaut und deployed wurde, wird ein README angezeigt, das alle weiteren Schritte enthält.

#### 4.3.2.1 Vergleich mit Prüfungsordnung

Die Prüfungsordnung (exemplarisch für den Studiengang Informatik) legt in § 17 fest, welche Vorbedingungen erfüllt sein müssen, damit die Bachelorarbeit angemeldet werden kann [FH 13, S. 5]:

- (1) Zur Bachelorarbeit wird zugelassen, wer alle Prüfungen bis auf maximal zwei erbracht hat und das Praxisprojekt erfolgreich absolviert hat. Beim Studiengang Informatik mit Praxissemester ist zusätzlich zur Zulassung das bescheinigte Praxissemester gemäß § 13 Absatz 9 erforderlich.

Der dazugehörige speziell für diese Prüfungsordnung implementierte *BachelorThesis*-Contract für den Studiengang ohne Praxissemester:

#### Listing 4.13: fhachain/smart-contracts/contracts/Exams.sol

```
1 contract BachelorThesis is BaseExam {  
2     constructor(address examinationServiceAddress, uint examId)  
3         public BaseExam(examinationServiceAddress, examId) {}  
4  
5     function checkStudentApproval(address studentAddress)  
6         external view  
7         returns (uint8)  
8     {
```

```

9   // A student has a max. of 2 attempts for the Bachelor Thesis
10  if (examinationService.getAttempts(studentAddress, id) >= 2) {
11      return uint8(Status.MAXIMUM_ATTEMPTS_EXCEEDED);
12  }
13  // Check if student passed Practice Project
14  if (examinationService.isExamPassed(studentAddress, examinationService
15 .getPracticeProjectId(studentAddress))) {
16      return uint8(Status.UNFULFILLED_PRECONDITIONS);
17  }
18  uint examAmount = examinationService.getAllStudentStandardExamIds(
19 studentAddress).length;
20  // Check if student has passed all exams except a maximum amount of 2:
21  if (examinationService.getPassedStandardExamsAmount(studentAddress) >=
22 examAmount - 2) {
23      return uint8(Status.UNFULFILLED_PRECONDITIONS);
24  }
25  return uint8(Status.APPROVED);
26 }
```

---

Der Vergleich legt dar, dass sich Smart Contracts sehr nah an bestehende Vereinbarungen formulieren lassen, sofern ein gewisses Maß an Abstraktion erreicht ist. Dadurch kann etwa verhindert werden, dass eine Regel, wie sie in der Prüfungsordnung formuliert ist, nur zum Teil implementiert wird. Entwickler von *Exam*-basierten Contracts können sich also bei der Implementation sehr nah an bestehende Prüfungsordnungen halten.

#### 4.3.2.2 Vergleich mit Datenbank

Der Contract Listing 4.14 lässt sich auch mit dem zuvor eingesetzten SOAP-Service aus Abb. 4.11 vergleichen, der hauptsächlich die folgende Geschäftslogik hat:

**Listing 4.14: fhachain/smart-contracts/contracts/Exams.sol**

```

1 Connection con = DriverManager.getConnection("jdbc:mysql://...", props);
2 PreparedStatement checkStatement = con.prepareStatement(
3   "SELECT COUNT(subject_id), (SELECT COUNT(id) FROM subject) " +
4   "FROM student AS st JOIN written_exam AS ex ON st.id = ex.student_id " +
5   "WHERE passed_practice_project = 1 AND grade != '5.0' AND student_id=?");
6 checkStatement.setLong(1, studentId);
7 ResultSet valuePair = checkStatement.executeQuery();
```

```
8 valuePair.next();  
9 return valuePair.getInt(2) >= valuePair.getInt(1) - 2;
```

Aus dem Listing 4.14 ist ersichtlich, dass zuvor eine Datenbank angebunden wurde. Diese wird nach der Integration des Contracts nicht mehr benötigt.

Dass die Smart Contracts zu einem Netzwerk aus Knoten deployed werden, hat zudem folgenden Vorteil, der sich besonders hier herausstellt: Falls an einer anderen Stelle eine ähnliche Funktionalität bzw. Geschäftslogik wie in Listing 4.14 benötigt würde, so müsste entweder a) die Logik dupliziert werden oder b) der Code in eine Bibliothek verpackt werden, die dann z.B. mit *Maven* eingebunden werden kann.

Beide Schritte entfallen durch den Einsatz des *ExaminationService*-Contracts, da etwa mithilfe des *Web3Connectors* (wie in Abb. 4.8 gezeigt) lediglich mit einer Adresse und der Funktionssignatur auf die Geschäftslogik zugegriffen werden kann.

### 4.3.2.3 Zusammenfassung

Für den Einsatz von Ethereum spricht hinsichtlich des Prozesses, dass durch die Verwendung eines *upgradeable* Contracts eine hohe Erweiterbarkeit garantiert wird: Es wäre beispielsweise möglich, dass ein Dozent speziellere Bedingungen für eine Klausur in Form eines Contracts selbst formuliert, falls sie nicht im Standardset der *Exam*-Implementationen bereits vorhanden sind.

Die dezentrale Natur könnte demzufolge auch dafür sorgen, dass sich die Verantwortlichkeiten verschieben und diese auch dezentraler werden.

Weiterhin konnte durch den Vergleich mit der Prüfungsordnung in Abschnitt 4.3.2.1 dargelegt werden, dass sich Smart Contracts durchaus nah an realen Verträgen formulieren lassen, was zu weniger Programmierfehlern führen könnte.

Zudem hat Abschnitt 4.3.2.2 gezeigt, dass es Vorteile hat, eine Möglichkeit zu haben, die Geschäftslogik dezentral abzulegen.

# 5 Fazit

## 5.1 Pro und Contra

Die Verwendung der FHACChain Kernkomponenten hat zusammengefasst folgende Vorteile und Nachteile:

Pro	Contra
Alle 4 von 5 Schutzziele außer Vertraulichkeit konnten vollständig umgesetzt werden.	Die Vertraulichkeit ist definitiv von Bedeutung. Die in dieser Arbeit vorgenommene Speicherung unter Pseudonymen ist wahrscheinlich nicht ausreichend für einen Produktiveinsatz an der FH Aachen.
Ein hoher Grad der Erweiterbarkeit wurde erreicht: FH-Mitglieder können etwa das bestehende Ökosystem der Smart Contracts problemlos erweitern, damit immer mehr neue dezentralisierte Applikationen erschaffen werden können.	Abschnitt 4.1.3.1 hat gezeigt, dass es einige Besonderheiten bei der Programmierung mit Solidity gibt, die nicht offensichtlich sind und zu Programmierfehlern führen könnten.
Es entsteht mehr Transparenz bezüglich der Geschäftslogik bei der Verwendung von Smart Contracts.	Programmierfehler haben bei Smart Contracts eine weitaus fatalere Wirkung.
Smart Contracts lassen sich nah an realen Verträgen formulieren.	Die Anschaffung neuer Hardware wäre für ein Produktivsystem notwendig (Kostenaufwand). Zudem ist das Mining ineffizient und es verursacht hohe Stromkosten.
Es gibt gute Möglichkeiten der horizontalen Skalierung.	Der Verlust eines Private Keys ist unumkehrbar und stellt ein (permanentes) Identitätsdiebstahl-Risiko dar.
Studenten und andere FH-Mitglieder können Praxiserfahrungen sammeln, indem sie selbst dezentralisierte Applikationen schreiben.	

**Tabelle 5.1:** Pro und Contra der Blockchain-Integration mithilfe der *FHACChain*.

## 5.2 Ergebnis

Zusammenfassend lässt sich sagen, dass sich die Blockchain-Integration mithilfe der *FHACChain* gut für die Geschäftsprozesse der Fachhochschule Aachen eignet, auch wenn Abstriche bei der Dezentralisierung gemacht werden müssen, damit sie real eingesetzt werden kann – solange jedoch nur gezielt an bestimmten Stellen zentralisiert wird, gibt es dennoch Vorteile, die nicht unbeachtet gelassen werden sollten. Die Komplexität der Technologie ist möglicherweise ihr größter Nachteil – damit sie in Prozesse integriert werden kann, ist es notwendig, zunächst ein Grundverständnis aufzubauen; schließlich ist der architektonische Wandel von „Client-Server“ nach „Client-Client“ im Bezug auf das Ausführen von Programmen gänzlich neu.

Letztendlich zeigt die Technologie ihre (Sicherheits-relevanten) Stärken insbesondere in Abschnitt 4.3.1, im „Klausur durchführen“-Prozess. Hier wird sie sinnvoll mit NFC kombiniert, um Transaktionen abzuschließen, die digital signiert sind und im Nachhinein nicht mehr verfälscht werden können. Zu diesem Prozess gibt es im Ausblick in Abschnitt 5.3.3 eine Fortsetzung.

Weiterhin konnten in Abschnitt 4.3.2 die Möglichkeiten zur Erweiterbarkeit von Smart Contracts demonstriert werden, die sich nicht zuletzt auch sehr nah an realen Verträgen formulieren lassen.

Die Anwendungsmöglichkeiten sind mit Ethereum unbegrenzt; ob sich das Projekt *FHACChain* jedoch auch als Community-Projekt durchsetzen und für Lehrzwecke eingesetzt werden kann, wird nur die Zukunft und die Praxis zeigen – an der Fachhochschule Aachen, der University of Applied Sciences.

## 5.3 Ausblick

In diesem finalen Abschnitt finden sich Projektideen und Konzepte, die mit dem in dieser Arbeit zur Verfügung gestellten *FHACChain* Technologie-Stack in Zukunft realisiert werden können.

### 5.3.1 Pilotprojekt: FH Aachen Testnetzwerk

Es gibt zwar mit dem *FHACChain* Technologiestack (siehe Abschnitt 4.1) bereits die *Software-Infrastruktur* für eine FH-interne Ethereum Blockchain. Allerdings gibt es keine *Hardware-Infrastruktur*, damit auf einem real existierenden Knotennetzwerk getestet und entwickelt werden kann. Dieses Netzwerk lässt sich im Optimalfall von allen FH-Mitgliedern nutzen – niemand sollte ausgeschlossen werden, da dies der dezentralen Natur zu Gute kommt.

Zum Aufbau eines realen und offiziellen Testnetzwerks innerhalb der Fachhochschule Aachen ist es unvermeidlich, folgende Punkte zu klären:

- Notwendig: Die Fachhochschule Aachen um Erlaubnis bitten.
- Das Erstellen eines notwendigen IT-Sicherheitskonzepts speziell für die hauseigene Ethereum Blockchain, in Kooperation mit evtl. Sicherheitsbeauftragten der FH:
  - Scope festlegen: Fachbereichs-übergreifend, Fachbereich-intern oder (sogar) von Außen erreichbar?<sup>1</sup>
  - Tiefgreifende Analyse, ob die Nutzung einer *private permissioned* Blockchain erforderlich ist. In diesem Fall müsste etwa die Spezialform Ethereums *Quorum* eingesetzt werden (Abschnitt 3.3.1.3). Als Folge müsste die Software-Infrastruktur der *FHACChain* zu *Quorum* migriert werden.
  - Firewall-Anforderungen und -Spezifikation festlegen.
  - Bei der zuständigen Stelle nachfragen, ob es möglich ist, die bereits existierende *FH Karte* zum Speichern von Ethereum Accounts zu verwenden (siehe auch Abschnitt 3.3.6 und Abschnitt 4.1.4.1).
- Ermitteln der Hardwareanforderungen (siehe auch Abschnitt 3.2.1.2):
  - Wie viele (Mining-) Knoten sind mindestens nötig, damit die Schutzziele aus Abschnitt 3.2.1.1 realisiert werden können?
  - Welche Mindestanforderungen werden an die Hardware gestellt?
  - Berechnung der Kosten für die Anschaffung und für den Betrieb (d.h. Energiebedarf mit einkalkulieren).

---

<sup>1</sup>Eventuelle Kommunikation mit Partnerhochschulen beachten und festlegen, siehe Abschnitt 3.1.

### 5.3.2 Prüfungsamt 3.0

Eine neue Generation des Prüfungsamts als *DApp* könnte das *QIS HIS* (Abschnitt 3.1.2) vollständig ersetzen.

Zur Realisierung können die bereits implementierten Smart Contracts (Abschnitt 4.1.3.1) als Backend verwendet werden, so dass (größtenteils) lediglich das UI (Frontend) implementiert werden muss. Zum Anmelden und Abmelden von Prüfungen gibt es zwar bereits das *FHACChain Terminal* (Abschnitt 4.1.4.1), allerdings wäre es praxistauglicher, wenn Studierende auch mobil – ohne ein Terminal aufzusuchen zu müssen – Prüfungen verwalten und auch Noten einsehen können.

Als Basis für die Anbindung an die Smart Contracts könnte dazu das in Abschnitt 4.1.4.2 vorgestellte Frontend verwendet werden, so dass eine neue dezentralisierte Applikation mithilfe der FHACChain entstehen könnte – das Prüfungsamt 3.0.

Es wäre sogar denkbar, dass Studierende eigene Applikationen rund um die Basis des *ExaminationService-Contracts* schreiben; etwa ein Benachrichtigungsdienst, der neue Noten auf einem Smartphone bekannt gibt.

### 5.3.3 Geschäftsprozess: Klausurbenotung durchführen

Der Geschäftsprozess „Klausurbenotung durchführen“ ist der Nachfolger des „Klausurdurchführen“-Prozesses aus Abschnitt 4.3.1. Im Ist-Prozess von „Klausurbenotung durchführen“ wird eine Vielzahl von handschriftlichen Unterschriften benötigt.

In diesem Abschnitt wird gezeigt, wie sich die Unterschriften *auf Dokumenten* digitalisieren lassen – eine Möglichkeit, wie die dazugehörige Anwesenheitsliste selbst digitalisiert werden kann, wurde ja bereits u.a. in Abschnitt 4.3.1 dargestellt.

#### 5.3.3.1 Rechtliche Anforderungen

Artikel 26 der EU-Verordnung Nr. 910/2014 legt die Anforderungen an die fortgeschrittenen elektronische Signatur (die auch vor Gericht als Beweismittel eingesetzt werden kann, siehe Exkurs Abschnitt 3.2.1.1) wie folgt fest [Uni14]:

- a) Sie ist eindeutig dem Unterzeichner zugeordnet.
- b) Sie ermöglicht die Identifizierung des Unterzeichners.
- c) Sie wird unter Verwendung elektronischer Signaturerstellungsdaten erstellt, die

der Unterzeichner mit einem hohen Maß an Vertrauen unter seiner alleinigen Kontrolle verwenden kann.

d) Sie ist so mit den auf diese Weise unterzeichneten Daten verbunden, dass eine nachträgliche Veränderung der Daten erkannt werden kann.

Die Punkte a, b und d treffen auch schon nach jetzigem Stand der *FHACChain* zu (siehe Abschnitt 3.3.1).

Lediglich Punkt c ist aufgrund der Sicherheitslücken des in diesem Projekt eingesetzten *MIFARE Classic* NFC Tags nicht möglich. Stattdessen würde es sich eignen, die vorhandene *FH Karte* zu verwenden und diese für das Abwickeln von Transaktionen mit einem PIN zu schützen, damit die „alleinige Kontrolle“ gewährleistet ist (siehe auch Abschnitt 3.3.6.2).

### 5.3.3.2 Soll-Geschäftsprozess

Auf der folgenden Seite wird gezeigt, wie ein derartiger Prozess funktionieren könnte. Die Essenz der Idee ist es, einen neuen Smart Contract namens *DocumentStorage* zum Repertoire der *FHACChain* hinzuzufügen, der ein Mapping von FH-Mitglieder Ethereum Adressen zu Hashwerten beinhaltet. Immer wenn ein Dozent etwa eine Notenliste publizieren möchte, so trägt er mit einer Transaktion die Prüfsumme des Dokuments in das *DocumentStorage* ein; es wäre zu ineffizient, das gesamte Dokument in der Blockchain abzulegen. Zur Speicherung des Dateiinhalts kann z.B. die /files-Schnittstelle des REST Service aus Abschnitt 4.1.3.2 verwendet werden.

Falls im Nachhinein die Echtheit eines Dokuments überprüft werden soll, so würde zuerst der Hashwert der Datei berechnet werden, um dann mithilfe dieses Wertes das *DocumentStorage* zu durchsuchen und den Urheber des Dokuments zu finden (falls das Dokument echt ist).

Das Konzept ließe sich ebenfalls auf andere Prozesse, in denen andere Dokumente signiert werden müssen, übertragen. Daher wäre es sinnvoll, den Smart Contract möglichst generisch zu implementieren. Im Bezug auf die Notenbekanntgabe wäre es vorteilhaft, diesen direkt – wie in Abb. 5.1 dargestellt – an den bestehenden *ExaminationService* anzukoppeln, so dass Studierende die Note unmittelbar danach einsehen können (siehe auch Abschnitt 5.3.2).

## 5 Fazit

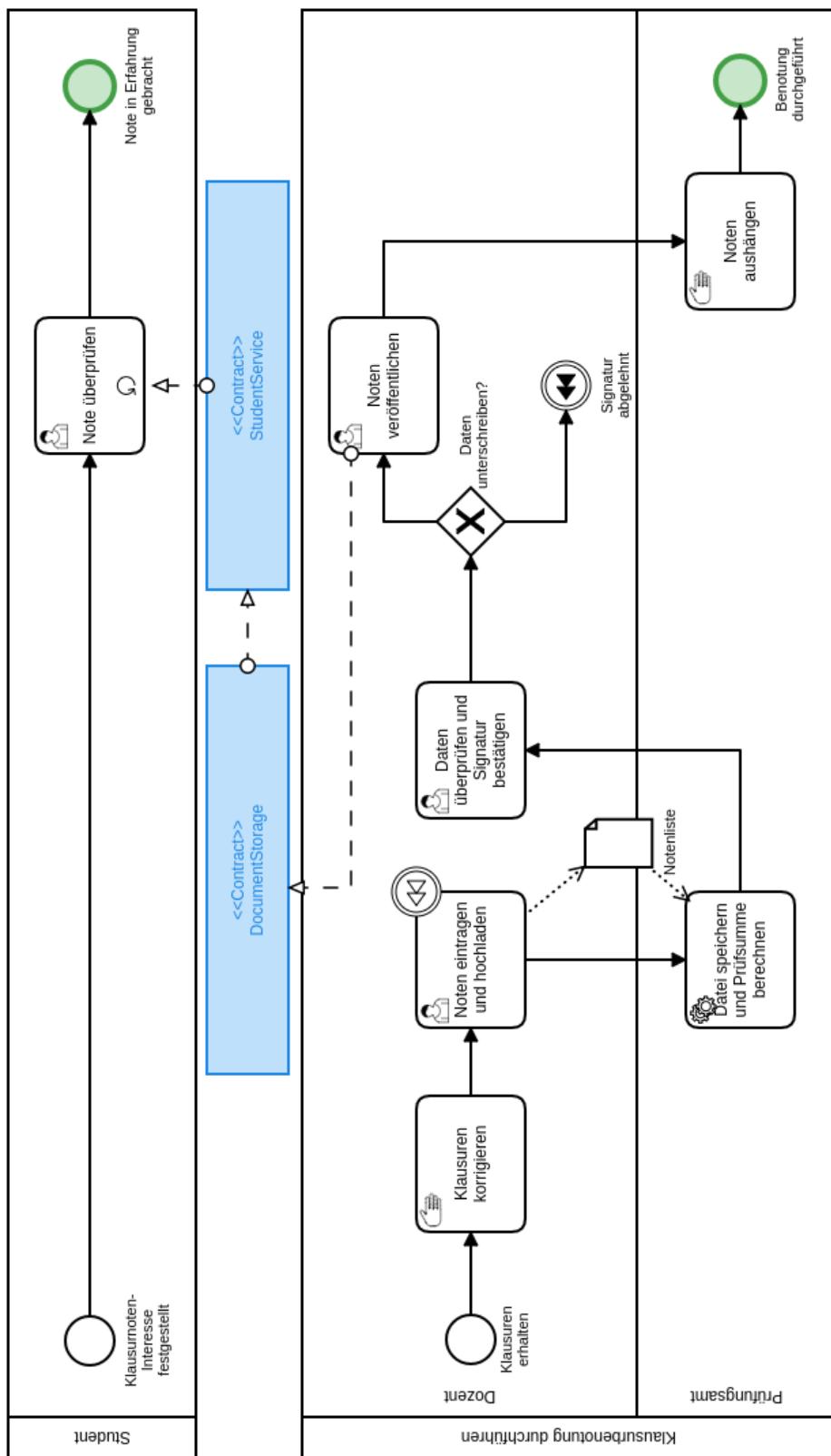


Abbildung 5.1: Der Vorschlag zum Soll-Geschäftsprozess „Klausurenbenotung durchführen“.

ERSTELLT VIA: Camunda Modeler v1.11.3 (Linux)

# Literatur

## Technologische Berichte

- [But17a] Vitalik Buterin. *A Next-Generation Smart Contract and Decentralized Application Platform*. Techn. Ber. Ethereum Foundation, 18. Sep. 2017.  
URL: <https://github.com/ethereum/wiki/wiki/White-Paper> (besucht am 18.05.2018).
- [Hea16] Mike Hearn. *Corda: A distributed ledger*. Version 0.5. Techn. Ber. R3, 29. Nov. 2016, S. 56.  
URL: [https://docs.corda.net/\\_static/corda-technical-whitepaper.pdf](https://docs.corda.net/_static/corda-technical-whitepaper.pdf) (besucht am 07.06.2018).
- [Nak08] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Techn. Ber. Bitcoin.org, Nov. 2008.  
URL: <https://bitcoin.org/bitcoin.pdf> (besucht am 17.05.2018).
- [VS17] Martin Valenta und Philipp Sandner.  
*Comparison of Ethereum, Hyperledger Fabric and Corda*. Forschungsber. Frankfurt School, Juni 2017, S. 8.  
URL: [http://explore-ip.com/2017\\_Comparison-of-Ethereum-Hyperledger-Corda.pdf](http://explore-ip.com/2017_Comparison-of-Ethereum-Hyperledger-Corda.pdf) (besucht am 06.06.2018).
- [Woo18] Dr. Gavin Wood.  
*Ethereum: A Secure Decentralised Generalised Transaction Ledger*. Techn. Ber. Parity.io, 4. Mai 2018.  
URL: <https://ethereum.github.io/yellowpaper/paper.pdf> (besucht am 18.05.2018).

## Standards und Verordnungen

- [FH 13] Der Rektor der FH Aachen. *FH-Mitteilungen 22. August 2013 Nr. 97 / 2013. Prüfungsordnung für den Bachelorstudiengang „Informatik“ und für den Bachelorstudiengang „Informatik mit Praxissemester“ im Fachbereich Elektrotechnik und Informationstechnik an der Fachhochschule Aachen.* Hrsg. von Silvia Crummenerl Dezernat Z. FH Aachen. 2013.  
URL: [https://www.fh-aachen.de/downloads/fh-mitteilungen/pruefungsordnungen/elektrotechnik-und-informationstechnik/bachelor/informatik-bis-ws1415/?no\\_cache=1&download=2013\\_97\\_PO\\_B\\_Inf\\_1f\\_StBabWS1314.pdf&did=1441](https://www.fh-aachen.de/downloads/fh-mitteilungen/pruefungsordnungen/elektrotechnik-und-informationstechnik/bachelor/informatik-bis-ws1415/?no_cache=1&download=2013_97_PO_B_Inf_1f_StBabWS1314.pdf&did=1441) (besucht am 13.06.2018).
- [NXP15] NXP.com. *MF3ICDx 21/41/81. MIFARE DESFire EV1 contactless multi-application IC.* NXP Semiconductors. 9. Dez. 2015.  
URL: [https://www.nxp.com/docs/en/data-sheet/MF3ICDX21\\_41\\_81\\_SDS.pdf](https://www.nxp.com/docs/en/data-sheet/MF3ICDX21_41_81_SDS.pdf) (besucht am 30.05.2018).
- [NXP16a] NXP.com. *MF1S50YYX V1. MIFARE Classic EV1 1K - Mainstream contactless smart card IC for fast and easy solution development.* Version Rev. 3.2, 23 May 2018. Product data sheet. NXP Semiconductors. 27. Apr. 2016. URL: [https://www.nxp.com/docs/en/data-sheet/MF1S50YYX\\_V1.pdf](https://www.nxp.com/docs/en/data-sheet/MF1S50YYX_V1.pdf) (besucht am 08.06.2018).
- [NXP16b] NXP.com.  
*MFRC522. Standard performance MIFARE and NTAG frontend.* Version Rev. 3.9. Product data sheet. NXP Semiconductors. 27. Apr. 2016.  
URL:  
<https://www.nxp.com/docs/en/data-sheet/MFRC522.pdf> (besucht am 30.05.2018).
- [OMG11] OMG. *Business Process Model and Notation (BPMN).* Object Management Group. 3. Jan. 2011. URL: <https://www.omg.org/spec/BPMN/2.0/> (besucht am 28.05.2018).
- [OMG16] OMG. *Decision Model and Notation (DMN) V1.1.* Object Management Group. 1. Juni 2016. URL: <https://www.omg.org/spec/DMN/1.1/> (besucht am 27.05.2018).

- [Uni14] Europäische Union. *Verordnung (EU) Nr. 910/2014 des Europäischen Parlaments und des Rates über elektronische Identifizierung und Vertrauensdienste für elektronische Transaktionen im Binnenmarkt und zur Aufhebung der Richtlinie 1999/93/EG*. vom 23. Juli 2014. Deutsch. Artikel 26. EU. 28. Aug. 2014.  
URL: <https://eur-lex.europa.eu/legal-content/de/TXT/?uri=CELEX:32014R0910> (besucht am 05.06.2018). Amtsblatt.

## Bücher

- [Eck14] Claudia Eckert. *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. 2014, S. 1004. ISBN: 9783486859164.  
URL: <https://books.google.de/books?id=u3zpBQAAQBAJ>.
- [Has13] Mitchell Hashimoto. *Vagrant: Up and Running*. 1. Aufl. O'Reilly Media, Inc., Juni 2013, S. 155. ISBN: 978-1-449-33583-0.  
URL: <https://www.safaribooksonline.com/library/view/vagrant-up-and/9781449336103/> (besucht am 28.05.2018).
- [Hoe14] Georg Hoever. *Höhere Mathematik kompakt*. 2. Aufl. Springer Spektrum, Mai 2014. 245 S. ISBN: 978-3-662-43995-1.  
DOI: [10.1007/978-3-662-43995-1](https://doi.org/10.1007/978-3-662-43995-1).  
URL: <https://www.springer.com/de/book/9783662439944>.
- [JCI14] Brian Jepson, Don Coleman und Tom Igoe. *Beginning NFC*. 1. Aufl. O'Reilly Media, Inc., 14. Jan. 2014, S. 246. ISBN: 9781449324124.  
URL: <https://www.safaribooksonline.com/library/view/beginning-nfc/9781449324094/> (besucht am 29.05.2018).
- [Mou15] Adrian Mouat. *Using Docker*. 22. Dez. 2015, S. 358. ISBN: 978-1491915769.  
URL: <https://www.safaribooksonline.com/library/view/using-docker/9781491915752/> (besucht am 28.05.2018).

## KONFERENZEN

---

- [MOV01] Alfred J. Menezes, Paul C. Van Oorschot und Scott A. Vanstone. *Handbook of Applied Cryptography*. 5. Aufl. CRC Press, Aug. 2001. 816 S. ISBN: 0-8493-8523-7.  
URL: <http://cacr.uwaterloo.ca/hac/> (besucht am 19.05.2018).
- [Wal15] Craig Walls. *Spring Boot in Action*. Manning Publications Co., Dez. 2015, S. 264. ISBN: 9781617292545. URL:  
<https://www.manning.com/books/spring-boot-in-action>  
(besucht am 30.05.2018).

## Konferenzen

- [Alm14] Márcio Almeida, Hrsg. *Hacking Mifare Classic Cards*. Regional Summit. São Paulo: black hat, 2014.  
URL: <https://www.blackhat.com/docs/sp-14/materials/arsenal/sp-14-Almeida-Hacking-MIFARE-Classic-Cards-Slides.pdf> (besucht am 08.06.2018).
- [But16] *EIP 55: Mixed-case checksum address encoding*. 14. Jan. 2016.  
URL: <https://eips.ethereum.org/EIPS/eip-55> (besucht am 24.05.2018).
- [Gei+15] *BPMN Conformance in Open Source Engines*. (3. Apr. 2015). University of Bamberg. San Francisco Bay, CA, USA: IEEE, 2015.  
ISBN: 978-1-4799-8356-8. DOI: [10.1109/SOSE.2015.22](https://doi.org/10.1109/SOSE.2015.22).  
URL: <https://pdfs.semanticscholar.org/4d62/14123bcd82ad437eaeb76279ad58fff97f00.pdf> (besucht am 27.05.2018).
- [Hyp18b] Hyperledger.org, Hrsg. *Hyperledger Overview 2018-02*. Hyperledger.org, Feb. 2018.  
URL: [https://www.hyperledger.org/wp-content/uploads/2018/02/Hyperledger-Overview\\_February-2018-2.pdf](https://www.hyperledger.org/wp-content/uploads/2018/02/Hyperledger-Overview_February-2018-2.pdf) (besucht am 07.06.2018).

- [Pi18b] *Raspberry Pi Foundation: Strategy 2018 - 2020.* RASPBERRY PI FOUNDATION - UK REGISTERED CHARITY 1129409. 2018.  
URL: <https://static.raspberrypi.org/files/about/RaspberryPiFoundationStrategy2018%E2%80%932020.pdf> (besucht am 29. 05. 2018).
- [SB17] *EIP 649: Metropolis Difficulty Bomb Delay and Block Reward Reduction.* 21. Juni 2017. URL: <https://eips.ethereum.org/EIPS/eip-649> (besucht am 24. 05. 2018).

## Dokumentation und Anleitungen

- [Ala18] Alarm-Clock. *Ethereum Alarm Clock 1.0.0 documentation.* 2018.  
URL: <http://docs.ethereum-alarm-clock.com/en/latest/introduction.html> (besucht am 26. 05. 2018).
- [Bit18a] Bitcoin-Wiki-Community. *Script.* 26. Mai 2018. URL:  
<https://en.bitcoin.it/wiki/Script> (besucht am 06. 06. 2018).
- [Cam18e] Camunda. *Extending Connect.* 2018.  
URL: <https://docs.camunda.org/manual/7.7/reference/connect/extending-connect/> (besucht am 12. 06. 2018).
- [Cam18f] Camunda. *Install Camunda BPM.* Camunda Services GmbH. 2018.  
URL: <https://docs.camunda.org/manual/7.8/installation/> (besucht am 28. 05. 2018).
- [Eth16a] Ethereum. *Web 3, A platform for decentralized apps.* 2016. URL:  
<http://ethdocs.org/en/latest/introduction/web3.html> (besucht am 12. 06. 2018).
- [Eth17] Ethereum-Community. *Setting up private network or local cluster.* 27. März 2017. URL:  
<https://github.com/ethereum/go-ethereum/wiki/Setting-up-private-network-or-local-cluster> (besucht am 18. 05. 2018).

- [Eth18d] Ethereum. *Solidity Documentation. Release 0.4.25.* 8. Juni 2018.  
URL: <https://media.readthedocs.org/pdf/solidity/develop/solidity.pdf> (besucht am 11.06.2018).
- [Eth18e] Ethereum-Community. *JSON RPC API.* 7. Juni 2018.  
URL: <https://github.com/ethereum/wiki/wiki/JSON-RPC> (besucht am 11.06.2018).
- [Eth18f] Ethereum-Community. *Light client protocol.* Mai 2018.  
URL: <https://github.com/ethereum/wiki/wiki/Light-client-protocol> (besucht am 10.06.2018).
- [Eth18g] Ethereum-Community. *Proof of Stake FAQ.* 10. Mai 2018.  
URL: <https://github.com/ethereum/wiki/wiki/Proof-of-Stake-FAQ> (besucht am 26.05.2018).
- [Eth18h] Ethereum-Community.  
*Units and Globally Available Variables - Solidity 0.4.24 documentation.*  
e67f0147. Ethereum Foundation, 2018.  
URL: <http://solidity.readthedocs.io/en/v0.4.24/units-and-global-variables.html> (besucht am 02.06.2018).
- [ope16] openSUSE-Community. *Package management.* 27. Nov. 2016.  
URL: [https://en.opensuse.org/Package\\_management](https://en.opensuse.org/Package_management) (besucht am 28.05.2018).
- [Pi18a] Raspberry Pi. *Raspberry Pi FAQs.* 2018.  
URL: <https://www.raspberrypi.org/help/faqs/> (besucht am 29.05.2018).
- [Spr18a] Spring.io. *Spring Boot Reference Guide. Version: 2.0.2.RELEASE.* 2018,  
S. 398. URL: [https://docs.spring.io/spring-boot-docs/current/reference/pdf/spring-boot-reference.pdf](https://docs.spring.io/spring-boot/docs/current/reference/pdf/spring-boot-reference.pdf) (besucht am 30.05.2018).
- [Spr18b] Spring.io. *Understanding POJOs.* 2018.  
URL: <https://spring.io/understanding/POJO> (besucht am 30.05.2018).

## Artikel

- [Bor16] Detlef Borchers. „Elektronische Signaturverordnung eIDAS ist gestartet – wie geht es weiter?“ In: *heise Online* (2. Juli 2016). URL: <https://www.heise.de/newsticker/meldung/Elektronische-Signaturverordnung-eIDAS-ist-gestartet-wie-geht-es-weiter-3252396.html> (besucht am 05.06.2018).
- [But15] Vitalik Buterin. „On Public and Private Blockchains“. In: *Ethereum-Blog* (7. Aug. 2015). URL: <https://blog.ethereum.org/2015/08/07/on-public-and-private-blockchains/> (besucht am 06.06.2018).
- [But17b] Vitalik Buterin. „A Prehistory of the Ethereum Protocol“. In: *Vitalik Buterin’s website* (14. Sep. 2017). URL: <https://vitalik.ca/general/2017/09/14/prehistory.html> (besucht am 04.06.2018).
- [Cas17] Michael del Castillo. „Blockchain’s Boom Year: Job Market Grows 200 Percent“. In: *CoinDesk* (13. Dez. 2017). URL: <https://www.coindesk.com/blockchains-big-year-competitive-job-market-grows-200/> (besucht am 17.05.2018).
- [Eco15] Economist-Staff. „The great chain of being sure about things“. In: *The Economist* (31. Okt. 2015). URL: <https://www.economist.com/briefing/2015/10/31/the-great-chain-of-being-sure-about-things> (besucht am 18.05.2018).
- [Fow10] Martin Fowler. „Richardson Maturity Model. steps toward the glory of REST“. In: *MartinFowler.com* (18. März 2010). URL: <https://martinfowler.com/articles/richardsonMaturityModel.html> (besucht am 06.06.2018).
- [Kas17] Preethi Kasireddy. „How does Ethereum work, anyway?“ In: *Medium* (27. Juli 2017).

## INTERNET

---

- URL: <https://medium.com/@preethikasireddy/how-does-ethereum-work-anyway-22d1df506369> (besucht am 18.05.2018).
- [Ree14] Mark Rees. „You Say Bitcoin Has No Intrinsic Value? Twenty-two Reasons to Think Again.“ In: *Bitcoin Magazine* (5. Juli 2014).  
URL: <https://bitcoinmagazine.com/articles/you-say-bitcoin-has-no-intrinsic-value-twenty-two-reasons-to-think-again-1399454061/> (besucht am 23.05.2018).
- [Sti18] Kai Stinchcombe.  
„Blockchain is not only crappy technology but a bad vision for the future“. In: *Medium* (5. Apr. 2018).  
URL: <https://medium.com/@kaistinchcombe/c1ca122efdec> (besucht am 17.05.2018).
- [Upt18] Eben Upton. „Raspberry Pi 3 Model B+“. In: *Raspberry Pi Blog* (14. März 2018).  
URL: <https://www.raspberrypi.org/blog/raspberry-pi-3-model-bplus-sale-now-35/> (besucht am 29.05.2018).
- [Wal11] Benjamin Wallace. „The Rise and Fall of Bitcoin“. In: *Wired* (23. Nov. 2011).  
URL: [https://www.wired.com/2011/11/mf\\_bitcoin/](https://www.wired.com/2011/11/mf_bitcoin/) (besucht am 18.05.2018).

## Internet

- [Bit18b] Bitfly. *Ethereum Hard Forks*. Collection of all historic hard forks.  
Bitfly GmbH. 22. Mai 2018.  
URL: <https://www.etherchain.org/hardForks> (besucht am 22.05.2018).
- [BPM18] BPMN.io. *Web-based tooling for BPMN, DMN and CMMN*.  
Camunda Services GmbH. 2018.  
URL: <https://bpmn.io/> (besucht am 27.05.2018).

## INTERNET

---

- [Cam18a] Camunda. *Camunda Admin*. Camunda Services GmbH. 2018.  
URL: <https://camunda.com/products/admin/> (besucht am 28.05.2018).
- [Cam18b] Camunda. *Camunda Cockpit*. Camunda Services GmbH. 2018.  
URL: <https://camunda.com/products/cockpit/> (besucht am 28.05.2018).
- [Cam18c] Camunda. *Camunda Tasklist*. Camunda Services GmbH. 2018.  
URL: <https://camunda.com/products/tasklist/> (besucht am 28.05.2018).
- [Cam18d] Camunda. *Der Camunda Stack im Überblick*. Camunda Services GmbH. 2018. URL: <https://camunda.com/de/products/> (besucht am 28.05.2018).
- [Cam18g] Camunda. *Modeler*. Camunda Services GmbH. 2018.  
URL: <https://camunda.com/products/modeler/> (besucht am 28.05.2018).
- [Cam18h] Camunda. *The Camunda BPM Workflow Engine*.  
Camunda Services GmbH. 2018.  
URL: <https://camunda.com/products/bpmn-engine/> (besucht am 28.05.2018).
- [Can18] Canonical. *Ubuntu release end of life*. Canonical Ltd. 2018.  
URL: <https://www.ubuntu.com/info/release-end-of-life> (besucht am 06.06.2018).
- [Doc18] DocuSign. *eIDAS & die elektronische Signatur. Die eIDAS-Verordnung und der rechtliche Rahmen für die elektronische Signatur*. DocuSign, Inc. 2018.  
URL: <https://www.docusign.de/eidas> (besucht am 05.06.2018).
- [Eth16b] Ethereum-Community. *The Homestead Release*. Ethereum Foundation. 2016. URL:  
<http://www.ethdocs.org/en/latest/introduction/the-homestead-release.html> (besucht am 07.06.2018).

- [Eth18a] Ethereum. *About the Ethereum Foundation*. 2018.  
URL: <https://www.ethereum.org/foundation> (besucht am 17.05.2018).
- [Eth18b] Ethereum. *Enterprise Ethereum Alliance*. Ethereum Foundation. Juni 2018.  
URL: <https://entethalliance.org/media-coverage/> (besucht am 07.06.2018).
- [Eth18c] Ethereum. *Remix - Solidity IDE*. Ethereum Foundation. 26. Mai 2018.  
URL: <https://remix.ethereum.org/> (besucht am 26.05.2018).
- [FHA18a] FH-Aachen. *Kruzprofil der Hochschule*. FH Aachen. 2018.  
URL: <https://www.fh-aachen.de/pressestelle/kurzprofil-der-hochschule/> (besucht am 31.05.2018).
- [FHA18b] FH-Aachen. *WLAN - eduroam*. FH Aachen. 4. Juni 2018.  
URL: <https://www.fh-aachen.de/hochschule/datenverarbeitungszentrale/netzanbindung/wlan/> (besucht am 05.06.2018).
- [Fro17] Philipp Fromme. *Token Simulation for the Camunda Modeler. Token simulation as a plugin for the Camunda Modeler*. Camunda Services GmbH. 21. Sep. 2017.  
URL: <https://github.com/philippfromme/bpmn-js-token-simulation-plugin> (besucht am 28.05.2018).
- [Inv18] Investopedia. *Blockchain*. 18. Mai 2018. URL:  
<https://www.investopedia.com/terms/b/blockchain.asp>.
- [jpm18] jpmorganchase. *Quorum. A permissioned implementation of Ethereum supporting data privacy* <https://www.jpmorgan.com/quorum>. JPMorgan Chase und Co. 4. Juni 2018.  
URL: <https://github.com/jpmorganchase/quorum> (besucht am 06.06.2018).
- [Lea18] LearnCryptography.com. *51 Percent Attack*. 21. Mai 2018.  
URL: <https://learncryptography.com/cryptocurrency/51-attack> (besucht am 21.05.2018).

- [LK13] Thomas Lex und Volker Krüger. *Rechtliche Hinweise zur FH Karte*. FH Aachen. Dez. 2013.  
URL: [https://www.fh-aachen.de/fileadmin/org/org\\_dezernat\\_2/chipkarte/Rechtliche\\_Hinweise\\_FH\\_Karte.pdf](https://www.fh-aachen.de/fileadmin/org/org_dezernat_2/chipkarte/Rechtliche_Hinweise_FH_Karte.pdf) (besucht am 31.05.2018).
- [Pre18] Tom Preston-Werner. *Semantic Versioning 2.0.0*. 2018.  
URL: <https://semver.org/> (besucht am 06.06.2018).
- [Rou14] Margaret Rouse.  
*Definition - Confidentiality, integrity, and availability (CIA triad)*.  
Hrsg. von Matthew Haughn und Stan Gibilisco. Nov. 2014. URL: <https://whatis.techtarget.com/definition/Confidentiality-integrity-and-availability-CIA>.

## Statistiken

- [Bit18c] Bitinfocharts.com. *Price stats and information*. 20. Mai 2018. URL: <https://bitinfocharts.com/ethereum> (besucht am 20.05.2018).
- [DMN18] DMN-TCK. *DMN TCK results overview for Camunda 7.8.0*.  
Hrsg. von Camunda. 27. Mai 2018. URL: [https://dmn-tck.github.io/tck/overview\\_Camunda%20BPM\\_7.8.0.html](https://dmn-tck.github.io/tck/overview_Camunda%20BPM_7.8.0.html) (besucht am 27.05.2018).
- [Git18a] GitHub. *GitHub API: Search „ethereum“*. Hrsg. von GitHub. 6. Juni 2018.  
URL: <https://web.archive.org/web/20180606212100/https://api.github.com/search/repositories?q=ethereum> (besucht am 06.06.2018).
- [Git18b] GitHub. *GitHub API: Search „hyperledger“*. Hrsg. von GitHub.  
6. Juni 2018.  
URL: <https://web.archive.org/web/20180606214119/https://api.github.com/search/repositories?q=hyperledger> (besucht am 06.06.2018).

- [Goo18] Google. *Google Trend „Blockchain“*. 17. Mai 2018. URL: <https://trends.google.com/trends/explore?date=2012-01-01%202018-01-01&q=blockchain> (besucht am 27.05.2018).
- [Hyp18a] Hyperledger. *Releases*. März 2018. URL: <https://github.com/hyperledger/fabric/releases?after=v1.0.1> (besucht am 07.06.2018).
- [Sta18a] StackExchange. *Stack Exchange Search Results „ethereum“*. Hrsg. von StackExchange. 6. Juni 2018. URL: <https://web.archive.org/web/20180606211220/https://stackexchange.com/search?q=ethereum> (besucht am 06.06.2018).
- [Sta18b] StackExchange. *Stack Exchange Search Results „hyperledger“*. Hrsg. von StackExchange. 6. Juni 2018. URL: <https://web.archive.org/web/20180606214312/https://stackexchange.com/search?q=hyperledger> (besucht am 06.06.2018).

## Reine Bildquellen

- [Amo16] Evan Amos, Hrsg. *Raspberry Pi Zero Image*. 13. Nov. 2016. URL: <https://en.wikipedia.org/wiki/File:Raspberry-Pi-Zero-FL.jpg> (besucht am 29.05.2018).

# A Anhang

Auf dieser letzten Seite befindet sich ein 16 GB großes Speichermedium mit der *FHACChain Distribution* inkl. Bachelorarbeit-Erweiterung, erstellt am 21. Juli 2018, zu Händen des Erst- und Zweitprüfers.



v1.0.0