HAW Hamburg
Fakultät LS
Life Sciences

Masterarbeit

Weiterentwicklung eines Energiehandels- und Netzstabilitätsdemonstrators auf Blockchainbasis

Bereich: Renewable Energy Systems

Verfasser: Linus Grupp

Matrikelnummer: [redacted]

1. Gutachter: Prof. Dr.-Ing. Volker Skwarek

2. Gutachter: Prof. Dr.-Ing. Cornelia Stübig

vorgelegt am: 24.10.2018

Hiermit versichere ich, Linus Grupp (Matrikelnummer: [redacted]) an Eides statt, dass ich die vorliegende Masterarbeit mit dem Titel „Weiterentwicklung eines Energiehandels- und Netzstabilitätsdemonstrators auf Blockchainbasis“ selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht. Die Arbeit ist noch nicht veröffentlicht oder in anderer Form als Prüfungsleistung vorgelegt worden.

Ich habe die Bedeutung der eidesstattlichen Versicherung und die prüfungsrechtlichen sowie strafrechtlichen Folgen (siehe unten) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

Strafrechtliche Folgen einer eidesstattlichen Versicherung nach § 156 StGB „Falsche Versicherung an Eides Statt“:

> Wer von einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Hamburg, den Unterschrift

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Quellcodeverzeichnis

Abkürzungsverzeichnis

CO_2 Kohlenstoffdioxid

DApp Dezentralisierte Applikation

DLT Distributed Ledger Technology

EEX European Energy Exchange

EOA External Owned Account

EPEX European Power Exchange

EVM Ethereum Virtual Machine

GB Gigabyte

Hz Hertz

ID Identifikationsnummer

IKT Informations- und Kommunikationstechnologie

IoT Internet of Things

kbyte Kilobyte

kV Kilovolt

kW Kilowatt

KWK Kraft-Wärme-Kopplung

MW Megawatt

OTC Over the Counter

P2P Peer-to-Peer

P2X Power-to-X

PoS Proof-of-Stake

PoW Proof-of-Work

PV Photovoltaik

RC V1 Registry Contract Version 1

RC V2 Registry Contract Version 2

SC Smart Contract

TWh Terawattstunde

USD US Dollar

V Volt

1 Einleitung

Elektrizität ist heutzutage notwendiges Fundament technischen Fortschritts in Deutschland und anderen fortgeschrittenen Volkswirtschaften und für den Alltag zur Selbstverständlichkeit geworden. Dabei stellt die Stromversorgung eines der komplexesten Systeme der Infrastruktur dar. Mit den gegenwärtigen Veränderungen im Stromsektor bezüglich erneuerbarer Energien, Dezentralisierung, Sektorkopplung und Digitalisierung wird die Komplexität weiter zunehmen (Aichele und Doleski, 2014, S. 4; Neugebauer, 2018, S. 349-353). Die Energiewende und dabei insbesondere die Förderung erneuerbarer Energien führt zu einem höheren Anteil volatiler Erzeugung. Das durch zentrale Großkraftwerke gekennzeichnete System mit klarem Top-Bottom Fluss weicht einer dezentralen Versorgung mit bidirektionalem Fluss, wovon besonders die niederen Spannungsebenen betroffen sind. Global wird der Stromverbrauch voraussichtlich bis 2040 um 30 % steigen, was die Bedeutung einer sicheren und effizienten Versorgung zusätzlich erhöht (International Energy Agency, 2017, S. 1).

Durch diese Veränderungen ist mit einer gesteigerten Koordination und Kommunikation im Stromsystem zu rechnen. Die Digitalisierung stellt dabei einen entscheidenden Prozess dar, weil sie einen schnellen sowie automatisierten Umgang großer Datenmengen verspricht (BDEW, 2018). Mit zunehmender Demokratisierung des Strommarktes werden Verbraucher künftig stärker eingebunden und könnten die Möglichkeit erhalten, aktiv am Stromhandel teilzunehmen (IqtiyaniIlham u. a., 2017, S. 783). Microgrids, Peer-to-Peer (P2P) Energieaustausch sowie Bereitstellung von Regelenergie durch den Zusammenschluss zu virtuellen Kraftwerken sind Konzepte, die vermehrt Aufmerksamkeit erhalten.

Für die steigende Anzahl an Prosumern (sowohl Konsumenten als auch Produzenten) ist es derzeit nicht rentabel geringe Strommengen über den konventionellen Markt zu handeln (Merz, 2016, S. 18-19). Die Blockchain-Technologie ist eine

Plattform, welche vielversprechende Eigenschaften aufweist, um diese Aufgabe zu übernehmen (siehe Abschnitt 4.3).

Noch ist unklar, ob Blockchain am Geeignetsten ist, um P2P-Handel im Energiesektor zu realisieren oder andere Optionen bessere Eigenschaften mitbringen, siehe Amato u. a., 2015; Zhang, Wu, Long u. a., 2017; Zhang, Wu, Zhou u. a., 2018.

Die Blockchain-Technologie besitzt das Potential sowohl Energiehandel über einen festgelegten Marktmechanismus als auch die finanzielle Abwicklung in einem Schritt zu ermöglichen. Eine Stärke der Blockchain, nämlich Unveränderbarkeit, stellt jedoch gleichzeitig auch eine Herausforderung dar. Programme in der Blockchain können nicht mehr verändert werden, sodass Fehlerbehebung oder Funktionalitätenerweiterung nicht ohne Weiteres möglich sind. Ziel dieser Arbeit ist die Entwicklung eines Entwurfsmusters für die Ethereum Blockchain, welches genau dies ermöglicht:

Ein Netzwerk aus miteinander interagierenden Blockchain Programmen, welches jedes Programm aktualisierbar (updatable) macht und gleichzeitig die Interoperabilität der Programme aufrecht erhält.

Zunächst wird in Kapitel 2 auf das konventionelle Stromsystem mit getrenntem physikalischen und finanziellem Stromhandel eingegangen. Daraufhin erfolgt in Kapitel 3 eine Erläuterung der gegenwärtigen Veränderungen im Stromsektor und deren treibenden Kräfte. Anschließend wird in Kapitel 4 die Blockchain-Technologie beschrieben. Danach wird in Kapitel 5 speziell auf die Ethereum Blockchain eingegangen. Im Anschluss wird in Kapitel 6 detailliert die Aktualisierbarkeit von Programmen in der Ethereum Blockchain erörtert. In Kapitel 7 folgt die Ausarbeitung des Entwurfsmusters. In Kapitel 8 findet eine kritische Auseinandersetzung des entwickelten Systems statt. Abschließend wird in Kapitel 9 kurz die Arbeit zusammengefasst und danach mögliche Erweiterungen des Entwurfsmusters vorgestellt.

2 Aufbau des konventionellen Stromsystems

Die Bildung des deutschen Stromsystems geht zurück auf Entdeckung der Elektrizität im 19. Jahrhundert. Anfänglich wurden dezentral Gleich- und Wechselstromgeneratoren privat betrieben und hauptsächlich zu Beleuchtungszwecken eingesetzt. Das öffentliche Interesse an der Elektrifizierung sorgte für den Bau großer Kraftwerke, welche hauptsächlich oder ausschließlich staatlich finanziert wurden. Die Liberalisierung der Strommärkte in den 1990er Jahren inklusive der Privatisierung der Stromproduktion in Deutschland sorgte dafür, dass bis vor einigen Jahren 4 Stromkonzerne rund 80 % des Stroms lieferten. Damit einhergehend beinhaltete die Liberalisierung das sogenannte „Unbundling“, also die Trennung von Stromerzeugung, -transport und -vertrieb. Die erhofften Effekte waren verstärkter Wettbewerb und eine Senkung der Strompreise für den Kunden (Deutsche Energie-Agentur, 2018). Die Übertragung von Strom in den Netzen wurde monopolistisch gehalten und wird von Aufsichtsbehörden reguliert. Erzeugung, Handel sowie Vertrieb wurden dem Wettbewerb ausgesetzt, welche von Kartellbehörden überwacht werden (Bundesnetzagentur, 2017a, S. 61).

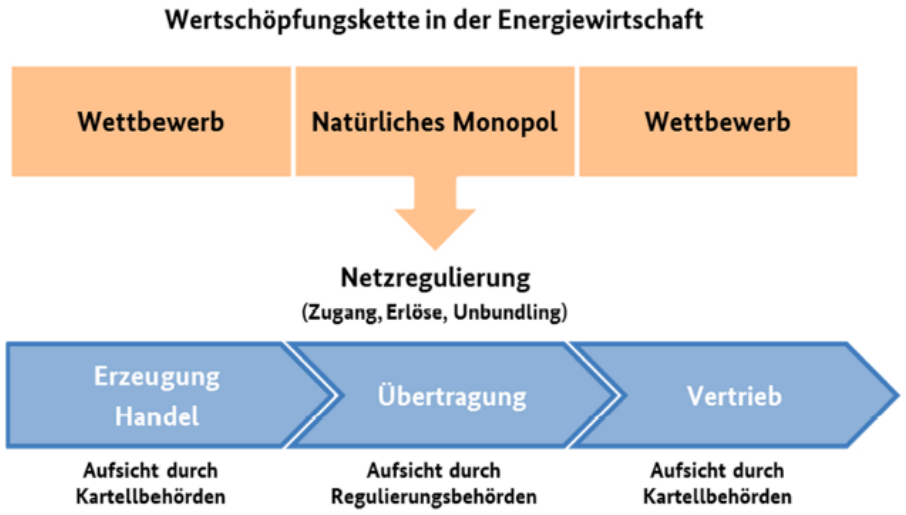

Abbildung 2.1: Wertschöpfungskette in der Energiewirtschaft (Bundesnetzagentur, 2017a, S. 59)

2.1 Aufbau und Betrieb des Stromnetzes

Das europäische Stromverbundnetz wird mit Wechselspannung bei einer Nennfrequenz von 50 Hz betrieben (Synwoldt, 2016, S. 67). Der Stromfluss des konventionellen Systems erfolgt Top-Bottom unidirektional von den hohen Spannungsebenen zu den niedrigeren Spannungsebenen. Die Netze werden in 4 Spannungsebenen unterteilt (Praktiknjo, 2013, S. 9-10; Schwab, 2009, S. 397-398):

- Höchstspannungsnetze bzw. Transportnetze (380/220 kV)
- Hochspannungsnetze bzw. Übertragungsnetze (110 kV)
- Mittelspannungsnetze (30/20/10 kV)
- Niederspannungsnetze (400/230 V)

Eine Verbindung der verschiedenen Spannungsebenen erfolgt über Transformatoren in Umspannwerken und Ortsnetzstationen (Konstantin, 2017, S. 315). Generell sind Transportverluste bei hohen Spannungen geringer. Daher speisen Kraftwerke weitab von Verbraucherzentren falls möglich in hohe Spannungsebenen ein (Zahoransky und Allelein, 2013, S. 392). Verwaltet und gesteuert werden die Netze

von Netzbetreibern. Dabei wird zwischen Übertragungsnetzbetreibern, zuständig für die oberen Spannungsebenen, und Verteilnetzbetreibern, welche für die unteren Spannungsnetze verantwortlich sind, unterschieden. In Deutschland gibt es 4 Übertragungsnetzbetreiber, welche neben dem überregionalen Stromtransport in verschiedenen Regelzonen für ein Gleichgewicht und Funktionalität im Netz zu sorgen haben (Graeber, 2014, S. 4-6). Die Aufrechterhaltung eines zuverlässigen Versorgungsbetriebs erfordert den Einsatz sogenannter Systemdienstleistungen. Diese beinhalten beispielsweise Frequenz- und Spannungshaltung, Störungsbeseitigung sowie Netzausbau (Bundesnetzagentur, 2017a, S. 62).

Das Einstellen eines Gleichgewichts im Netz erfolgt über die Einteilung einer Regelzone in Bilanzkreise. Ein Bilanzkreis besteht aus einem oder mehreren Netznutzern (Kraftwerke, Stromhändler und Lieferanten) und deren Einspeisungs- und Entnahmestellen. Ein Bilanzkreisverantwortlicher muss dem zuständigen Netzbetreiber im Voraus viertelstündliche Fahrpläne übermitteln, welche eine ausgeglichene Bilanz aufweisen. Diese Fahrpläne beruhen auf lastbeeinflussenden Faktoren wie beispielsweise historischer Daten für Typtage und Wettervorhersagen (Konstantin, 2017, S. 333). Die zeitliche Auflösung von 15 Minuten kann den realen Last- und Erzeugungsverlauf mit kontinuierlichen Änderungen nicht abbilden, wodurch Unregelmäßigkeiten enstehen können. Des Weiteren können plötzliche Ausfälle von Erzeugungseinheiten oder falsch prognostiziertes Lastverhalten für Abweichungen zu den Fahrplänen sorgen, was sich in Differenzen zwischen Erzeugung und Verbrauch widerspiegelt. Zum Ausgleich der unplanmäßigen Differenzen müssen die Übertragungsnetzbetreiber mit zusätzlich vorgehaltener Regelleistung eingreifen. Je nach Situationsbedarf kann diese sowohl positiv (Abschalten von Verbrauchern oder Zuschalten von Erzeugern) als auch negativ (Abschalten von Erzeugern oder Zuschalten von Verbrauchern) sein. Regelleistung wird in drei Bereiche mit unterschiedlichen Anforderungen aufgeteilt. Die höchsten Anforderungen hat die Primärregelleistung, welche innerhalb von 30 Sekunden eine Mindestleistung von 1 MW bis zu 15 Minuten vollautomatisch bereitstellen muss. Sekundärregel- und Minutenreserveleistung haben deutlich längere Aktivierungszeiten von 5 Minuten bzw. 15 Minuten, gleichzeitig jedoch auch eine Mindestleistung von 5 MW (Häfner, 2017, S. 6-7; Bundesnetzagentur, 2018). Die Bereitstellung von Regelleistung erfolgt über Ausschreibungen der Übertragungsnetzbetreiber. Die Vor-

haltung von Regelleistung schränkt Kraftwerksbetreiber ein. Bei einem Zuschlag einer Ausschreibung erhält ein Kraftwerksbetreiber deshalb einen Leistungspreis als Entschädigung. Bei tatsächlichem Bezug von Regelleistung erhalten die Kraftwerksbetreiber zusätzlich den Arbeitspreis, welcher in der Ausschreibung festgelegt wurde (Graeber, 2014, S. 8).

2.2 Strommarkt

Der Handel mit Strom unterscheidet sich vom Handel anderer Märkte aufgrund dreier Eigenschaften von Elektrizität - Homogenität, Immaterialität und Nicht-Speicherbarkeit. Der Transport kann ausschließlich netzgebunden erfolgen, was eine notwendige Infrastruktur voraussetzt. Diese wird von den im vorigen Abschnitt beschriebenen Netzbetreibern zur Verfügung gestellt und ausgebaut (Graeber, 2014, S. 4). In Europa erfolgt der Stromhandel über die European Energy Exchange (EEX) mit Sitz in Leipzig. Dort können börslicher Handel am Spotmarkt oder Terminmarkt sowie nicht börslicher bilateraler Handel über sogenannte Over the Counter (OTC) Geschäfte abgewickelt werden, welche ebenfalls in die Kategorie Spotmarkt fallen.

Im Spotmarkt des European Power Exchange (EPEX) fallen Handel und physische Stromlieferung zeitlich nah zusammen. Dort können stundenweise oder blockweise konstante Leistungen über Kontrakte erworben und verkauft werden. Stundenkontrakte erfolgen über einen Auktionshandel, die einen Tag vor der physischen Erfüllung zustandekommen (day ahead). Im kontinuierlichen Handel (intraday) können kurzfristig Überschüsse verkauft oder Strommengen eingekauft werden.

Im Terminmarkt liegen Vertragsabschluss und Lieferung mindestens eine Woche bis zu Jahren auseinander. Die Abwicklung von Future-Kontrakten ermöglicht eine Absicherung gegen erwartete schwankende Preise. Zudem können beispielsweise Strommengen bei Erwartung sinkender Preise im Voraus verkauft werden und später zu niedrigeren Preisen wieder eingekauft werden.

Da OTC Geschäfte außerbörslich durchgeführt werden, birgen diese eine gewisse Intransparenz bei gleichzeitig höherer Flexibilität der Vertragsgestaltung (Konstantin, 2017, S. 432-449). Stand 2015 betrugen die außerbörslich gehandelten OTC Strommengen 80 % der gesamten gehandelten Strommengen (Synwoldt, 2016, S.

351). Dies hängt damit zusammen, dass der Spotmarkt kurzfristiger und flexibler agiert als der Terminmarkt und dadurch eher mit einem sich stetig ändernden Netzzustand, Lastschwankungen oder verstärkt volatiler Einspeisung aufgrund erneuerbarer Energien korreliert. Des Weiteren wird Strom in Deutschland durchschnittlich etwa 10 mal gehandelt, bevor er letztendlich physisch geliefert wird (Krieger und Nickel, 2012, S. 22). Die minimal angebotene Leistung betrug 2016 0,2 MW, demzufolge scheinen Anbieter kleinerer Leistung den Strommarkt nicht sinnvoll nutzen zu können, da die Transaktionskosten im Verhältnis zur dargebrachten Leistung zu hoch sind (Merz, 2016, S. 18; Voshmgir, 2016, S. 24).

3 Transformation des Stromsektors

Obwohl mittlerweile eine komplette Durchdringung von Elektrizität in der Gesellschaft vorliegt, wurden im letzten Jahrhundert kaum Änderungen bezüglich Stromtransport/-verteilung sowie zum Einstellen eines Gleichgewichts zwischen Erzeugung und Verbrauch vorgenommen (Kok, 2013, S. 29-30). Die gegenwärtigen Veränderungen erfordern nun Anpassungen in diesen Bereichen.

Der grundlegende Wandel von fossilen Erzeugern hin zu erneuerbaren Erzeugern sorgt beispielsweise nicht nur für den weiteren Ausbau der Netzinfrastruktur, um die Energie bedarfsgerecht zu verteilen. Er erfordert außerdem einen erhöhten Datenaustausch zwischen Erzeugern, Verbrauchern und Verteilern, da Erzeugung aus Windkraft und Photovoltaik naturgemäß ihre Einspeisung nicht beliebig anpassen kann.

Grundsätzlich können vier Ursachen für die fortwährenden Änderungen im Stromsystem gesehen werden, auf die nun konkret in den folgenden Unterkapiteln eingegangen wird.

3.1 Erneuerbare Energien

Zu den erneuerbaren Energien zählen Windkraft, Wasserkraft, Photovoltaik (PV), Geothermie und Biomasse. Im Gegensatz zu konventionellen Kraftwerken, deren Betrieb Rohstoffe wie beispielsweise Erdgas oder Kohle verbrauchen, stehen erneuerbaren Energien kostenlose Energieträger in Form von Wind, Wasserkraft, Solarstrahlung und Wärme zur Verfügung (Synwoldt, 2016, S. 22-23). Eine Ausnahme ist die Biomasse, welche ebenso einen Rohstoffverbrauch hat. Sie wird dennoch als erneuerbar angesehen, da Biomasse emissionstechnisch so viel CO_2 freigibt wie zuvor beim Anbau aufgenommen wurde (Zichy u. a., 2011, S. 1).

Unterstützt durch die Energiewende und die damit politisch gesetzten Ziele

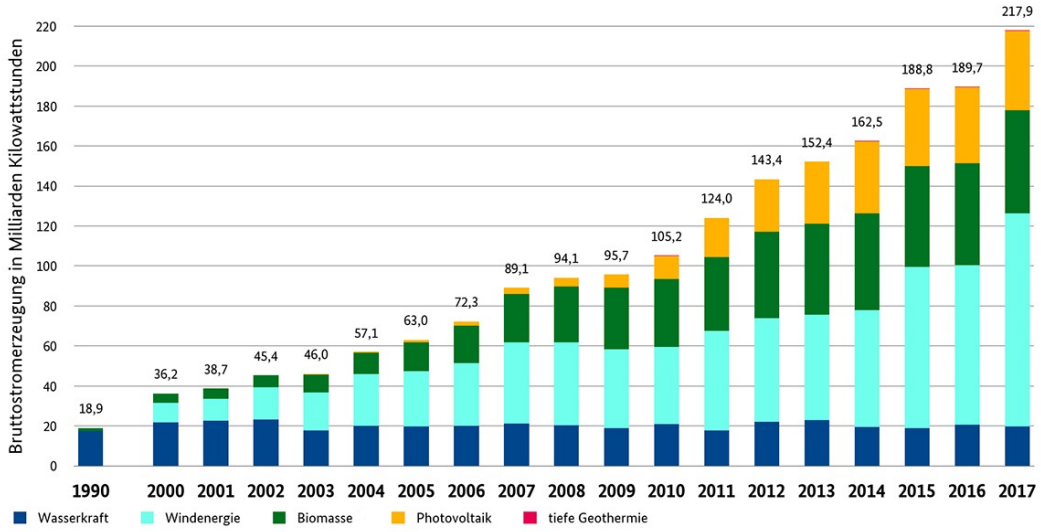

Abbildung 3.1: Entwicklung erneuerbarer Stromerzeugung (Bundesministerium für Wirtschaft und Energie, 2018)

nimmt die Erzeugung erneuerbarer Energien in Deutschland seit vielen Jahren stetig zu, wie in Abbildung 3.1 zu sehen ist. So hat sich die Stromerzeugung aus erneuerbaren Energien von 2005 bis 2015 etwa verdreifacht und lag 2017 bei knapp 218 TWh. Stagniert die Produktion bei Wasserkraft und Biomasse seit einigen Jahren, nehmen Windkraft und Photovoltaik weiterhin zu. Die steigende Bedeutung der erneuerbaren Energien im Stromsystem hat mehrere Gründe:

Zum einen sind fossile Energieträger, auf deren Basis die meisten zentralen Kraftwerke laufen, als Ressource begrenzt verfügbar. Erschwerend für Deutschland kommt hinzu, dass etwa 70 % des Primärenergiebedarfs mithilfe von Importen aus anderen Ländern gedeckt wird (Wilke, 2013). Zudem wurde 2011 nach dem Reaktorunglück im japanischen Fukushima der vollständige Kernenergieausstieg bis zum Jahr 2022 beschlossen. Die deutschen Kernkraftwerke werden also Schritt für Schritt stillgelegt. Das Fehlen dieser Kapazitäten muss demzufolge mit anderen Erzeugern kompensiert werden. Um die geplanten Emissionsziele zu erreichen ist die mittel- und langfristige Kompensation von Kernkraft durch fossile Kraftwerke ungeeignet. Eine logische Schlussfolgerung ist der weitere Ausbau erneuerbarer Energien.

Ein anderer Grund ist die Preisentwicklung für erneuerbare Energien. Von 2010 bis 2016 sind die Kosten für PV-Anlagen um 70 %, für Windkraftanlagen um 25 % und für Batterien um 40 % gesunken (International Energy Agency, 2017, S. 1). Der bisherige Trend verdeutlicht, dass die Preise in Zukunft sinken werden und erneuerbare Energien somit lukrativer werden (Kost u. a., 2018, S. 24).

Der wichtigste Grund für den Umbau der Stromversorgung ist jedoch die Begrenzung des anthropogenen Treibhauseffekts bzw. des Klimawandels. Kraftwerke basierend auf der Verbrennung fossiler Energieträger sorgen für Treibhausgasemissionen, welche den Klimawandel begünstigen und so eine nachhaltige Energieversorgung hinsichtlich kommender Generationen unmöglich machen (Eiselt, 2012, S. 8-9).

Die zunehmende Einspeisung aus Windkraft und Photovoltaik macht es schwieriger für die Netzbetreiber das Gleichgewicht im Netz zu halten. Die Wetterabhängigkeit von Windkraft und PV hat zur Folge, dass die Erzeugung aus diesen Quellen nur begrenzt prognostiziert werden kann. Dies kann zu unvorhergesehenen Schwankungen führen, sodass es trotz des Merit-Order-Effekts[1] bei wetterbedingten Erzeugungsspitzen im Notfall zur Abregelung oder Abschaltung von erneuerbaren Energieanlagen kommen kann, um die Versorgungssicherheit aufrecht zu erhalten (Scholz u. a., 2012, S. 20; Ministerium für Energiewende, Landwirtschaft, Umwelt, Natur und Digitalisierung, 2016, S. 6; Bundesnetzagentur, 2017b, S. 24-25).

Aufgrund der unsicheren Prognostizierbarkeit von erneuerbaren Energien können einzelne Anlagen nicht am Regelenergie-Markt teilnehmen. Eine Teilnahme über die Strombörse kann erreicht werden, indem sich mehrere Anlagen zu virtuellen Kraftwerken zusammenschließen (Bundesnetzagentur, 2017a, S. 71; Willi Horenkamp u. a., 2007, S. 45). Dazu müssen jedoch genug Erzeuger in einem Bilanzkreis gefunden werden. Zudem fallen bei der Abwicklung über die Strombörse Gebühren an (EEX, 2018).

[1] Der Merit-Order-Effekt besagt, dass erneuerbare Energien Vorrang bei der Einspeisung ins Netz gegenüber anderer Kraftwerkstypen haben. Das Kriterium für den Vorrang stellen die Grenzkosten der jeweiligen Anlage dar. Für erneuerbare Energien werden Grenzkosten nahe null angenommen, sodass sie Priorität erhalten (Doleski, 2017, S. 121).

3.2 Dezentralisierung

Mit dem Ausbau der erneuerbaren Energien geht der Aspekt einer dezentralen Erzeugung einher. Dezentrale Erzeugung bezeichnet die Installation kleinerer Erzeugungsanlagen im kW bis MW Bereich, welche sich über große Flächen verteilen. In der Regel sind diese Anlagen nah am Verbraucher orientiert. Dezentrale Erzeugung erfolgt mehrheitlich über erneuerbare Energien. Alleine deshalb werden große, verteilte Flächen benötigt, da die Energiedichten erneuerbarer Quellen wesentlich kleiner sind als die fossiler Energieträger (Jahn u. a., 2017, S. 17). Es gibt jedoch auch Windparks oder Photovoltaikkraftwerke, welche die Leistungen zentraler Kraftwerke erreichen und somit ebenso eher zentral als dezentral anzusehen sind. Dieselgeneratoren, Kraft-Wärme-Kopplungs-Anlagen (KWK-Anlagen), erneuerbare Energien Systeme und Brennstoffzellen sind eingesetzte Technologien für dezentrale Erzeugung (Rezaee Jordehi, 2016, S. 893-894). Diverse Vorteile gegenüber zentralen Großkraftwerken lassen darauf schließen, dass dezentrale Erzeugung zukünftig eine größere Bedeutung im Stromsystem zukommt:

Der Aufbau dezentraler Anlagen ist normalerweise in unter 6 Monaten möglich, wohingegen zentrale Kraftwerke bis zu 2 Jahre Installationszeit benötigen (Davis, 2002, S. 68). Dabei sollte berücksichtigt werden, dass zentrale Kraftwerke bei Fertigstellung wesentlich größere Leistungen bereitstellen als dezentrale Anlagen, wodurch dieser Vorteil relativiert wird.

Des Weiteren sorgen dezentrale Anlagen für eine höhere Diversität der Energieversorgung. Die großflächige Verteilung erneuerbarer kleinerer Systeme ist versorgungssicherer als ein zentrierter Kraftwerkspark, da die klimatischen Bedingungen so nicht von wenigen Standorten abhängen und das Risiko auf viele Anlagen verteilt wird.

Bei thermischen Kraftwerken fällt permanent Abwärme an. Ein Wärmetransport über weite Strecken erweist sich jedoch aufgrund hoher Verluste als wenig sinnvoll. Im Gegensatz dazu können KWK-Anlagen gleichzeitig Strom und Wärme lokal bereitstellen. Dies spiegelt sich in einer signifikanten Wirkungsgradsteigerung wieder. Moderne Dampfkraftwerke erreichen Wirkungsgrade von ca. 45 % zur Stromerzeugung. Demgegenüber erreichen KWK-Anlagen Gesamtwirkungsgrade von 60 % bei der Strom-und Wärmeerzeugung (Zahoransky und Allelein, 2013, S. 18). Zudem

werden Stromtransportverluste bei verbrauchernahen dezentralen Anlagen minimiert, welche in Deutschland bei etwa 7 % liegen (Statistisches Bundesamt, 2018).

Die Verbrauchernähe sorgt für einen weiteren positiven Effekt. Der produzierte Strom muss nicht über die Übertragungsnetze transportiert werden und entlastet somit das öffentliche Netz. Außerdem eignen sich dezentrale Anlagen gut zur Stromversorgung in ländlichen Gebieten mit schwacher Infrastruktur, da so lange und teure Leitungen zur Anbindung ans Netz vermieden werden (Rezaee Jordehi, 2016, S. 894). Bei teilweiser Autarkie einer bestimmten Region mithilfe dezentraler Anlagen können diese als Microgrids[2] angesehen werden.

Die Zunahme dezentral eingespeisten Stroms hat gleichwohl auch neue Herausforderungen zur Folge. Das klassische Modell mit Top-Bottom Fluss wird vermehrt von Einspeisungen in die niederen Spannungsebenen abgelöst. Dafür ist dieses System jedoch nicht ausgelegt. Es kann beispielsweise bei zu hoher Einspeisung in der Niederspannungsebene zu Rückflüssen in die oberen Spannungsebenen, also einem bidirektionalen Stromfluss, kommen (Crastan und Westermann, 2018, S. 514). Außerdem ist das Niederspannungsnetz kaum mit Sensorik und Aktorik ausgerüstet, da dies bisher nicht notwendig war. Die Folge ist, dass Verteilnetzbetreiber teilweise nicht so viel Kontrolle über ihr System haben, wie sie bräuchten, um es optimal betreiben zu können (Mayer und Dänekas, 2013, S. 78). Die starke Zunahme der Akteure im Stromsystem bedarf eines höheren Informations- und Kommunikationsflusses, worauf in Abschnitt 3.4 eingegangen wird.

3.3 Sektorkopplung

Sowohl aus ökonomischen als auch ökologischen Gesichtspunkten weist die Sektorkopplung, also die Vernetzung der Energienetze Strom, Mobilität, Wärme und Industrie zu integrativen Hybridnetzen, einen vielversprechenden Ansatz auf (Tichler und Moser, 2017, S. 222). Wie auch bei den anderen Veränderungen ist das oberste Ziel die Emissionsreduzierung von klimaschädlichen Treibhausgasen. Dies kann durch die Substitution fossiler Energieträger abseits vom Stromsektor durch

[2]Ein Microgrid ist eine inselbetriebene, dezentrale elektrische Versorgung eines Areals, welche oft mithilfe eines Zusammenspiels aus Erzeugungsanlagen und Speichern realisiert wird (Kroposki u. a., 2008, S. 1).

erneuerbare Energien erreicht werden, was einer Elektrifizierung der anderen Energiesektoren entspricht. Die Vorteile dieses Vorgehens sind:

Dekarbonisierung: Die Ausweitung erneuerbarer Energien auf alle Energiesektoren führt zur Senkung von CO_2-Emissionen. Vor wenigen Jahren liefen im Mobilitätssektor über 90 % und im Wärme-/Kältesektor knapp 90 % der Fahrzeuge bzw. Anlagen auf Grundlage fossiler Brennstoffe (Umweltbundesamt, 2018; BEE, 2018).

Effizienzsteigerung: Eine sinnvolle Kopplung der Sektoren mithilfe moderner Technologien wie Wärmepumpen, KWK-Anlagen sowie Elektroautos verspricht eine Senkung des Gesamtenergieverbrauchs. Wärmepumpen können je nach Typ Jahresarbeitszahlen[3] von ungefähr 2 – 4,5 erreichen (Schuberth, 2018). Auf den Wirkungsgrad von KWK-Anlagen wurde bereits im vorigen Abschnitt 3.2 eingegangen. Konventionelle Verbrennungsmotoren von Kraftfahrzeugen erreichen Wirkungsgrade zwischen 25 % und 45 %, während Elektromotoren Wirkungsgrade von 85 – 95 % aufweisen. Nicht mit eingerechnet ist dabei, mit welchem Wirkungsgrad der im Elektroauto genutzte Strom erzeugt wurde. Nur bei ausschließlich erneuerbarer Stromversorgung sind niedrige Wirkungsgrade nicht klimawirksam (Schmied und Wüthrich, 2015, S. 22).

Flexibilität: Die größten Anteile an erneuerbarer Stromerzeugung haben Windkraft und Photovoltaik. Mit ihrer schwankenden Erzeugung sind zeitlich verschiebbare Lasten optimal, um ein Gleichgewicht im Netz aufrecht zu erhalten. Flexible Lasten wie Wärmespeicher und Power-to-X Technologien (P2X) können als funktionale Speicher genutzt werden, um Überschüsse aufzufangen. Die Investitionskosten von Stromspeichern sind in etwa 100 mal höher als die von Wärmespeichern, somit bietet ein Wärmespeicher eine kostengünstige Option (Lund u. a., 2016, S. 9). Auch wenn solche Maßnahmen bei Rückvestromung mit Wirkungsgradeinbußen einhergehen, sind sie immer noch besser, als alternativ Anlagen abzuregeln oder Strom mit negativen Preisen veräußern zu wollen (Heimberger u. a., 2017, S. 230). Zudem muss nicht notwendigerweise eine Rückverstromung erfolgen. Im Idealfall sind flexible Speicher bereits vorhanden und müssen nicht neu aufgestellt

[3] Die Jahresarbeitszahl einer Wärmepumpe errechnet sich aus der bereitgestellten Wärmeenergie im Verhältnis zur verbrauchten Strommenge innerhalb eines Jahres (Kaltschmitt u. a., 2006, S. 404).

werden. Ein Beispiel wäre ein großes Kühlhaus mit hoher thermischer Kapazität, welches relativ variable Verbrauchszeiten gewährleistet (Sterner und Stadler, 2014, S. 587).

Durch den Trend zur Sektorkopplung kann davon ausgegangen werden, dass die Netzkapazitäten zunehmend stärker belastet werden, wenn zusätzliche Energiesektoren mit Strom versorgt werden müssen. Die geplante Erhöhung der Elektromobilität könnte beispielsweise für abendliche Probleme der Versorgungssicherheit verantwortlich werden, wenn nach der Arbeit die Autos geladen werden (Bundesregierung, 2009, S. 44-47; Wyman, 2018; Hajek, 2018). Dem könnte mit Netzausbau entgegnet werden, welcher jedoch kostspielig und langwierig ist. Andere Gegenmaßnahmen könnten ein preislicher Anreiz für eine zeitliche Verschiebung zur Vermeidung von Lastspitzen und die bessere Ausnutzung lokaler Angebote zur Entlastung der Netze sein.

3.4 Digitalisierung

Digitalisierung hat Einfluss auf viele wirtschaftliche und gesellschaftliche Bereiche. Sie ist dafür bekannt, Wertschöpfungsstufen zu verändern und Prozesse teilweise oder komplett zu übernehmen bzw. überflüssig zu machen (Wittpahl, 2017, S. 149). Eine übliche Konsequenz von Digitalisierung ist die Disintermediation, also der Wegfall von Vermittlern zwischen Akteuren in einem Wirtschaftssystem.

Auch im Stromsystem wird Digitalisierung mittlerweile als unverzichtbar angesehen, um die Herausforderungen aus Dezentralität und volatiler Erzeugung im bestehenden Netz zu integrieren. Stichwort ist hier die Informations-und Kommunikationstechnologie (IKT). Es wird davon ausgegangen, dass 2020 etwa 80 Milliarden Geräte miteinander vernetzt sind und somit dem Internet der Dinge (IoT) angehören (Vogel u. a., 2018, S. 49). Die Vernetzung einer Vielzahl von Geräten im Stromnetz ermöglicht eine automatisierte und effiziente Steuerung (Doleski, 2017, S. 286). Der stetige Ausbau kleinerer Erzeugungsanlagen, welche in das Mittel- und Niederspannungsnetz einspeisen, erfordert die Erfassung und Verarbeitung immer größerer Datenströme. Ein erster Schritt ist die Einführung von Smart Metern für Verbraucher. Diese intelligenten Messsysteme erfassen im Viertelstunden-Takt Verbrauchs- und Erzeugungsdaten und können diese weiterleiten (Fiedler u. a.,

2016, S. 7). Sie verfügen mindestens über die drei Funktionen Messen, Datenspeicherung und Kommunikation (Schellong, 2016, S. 235). Nicht nur für die Person, bei der Smart Meter installiert sind, sondern auch für Netzbetreiber sind solche Daten interessant und können das Netzmanagement verbessern, indem schneller und auf Grundlage besserer Daten als Standardlastprofile agiert und reagiert werden kann (Doleski, 2017, S. 606-607). Unterstützend wirkt das Gesetz zur Digitalisierung der Energiewende, welches 2016 verabschiedet wurde. Es reguliert die Bewirtschaftung von Smart Grids und beinhaltet zukünftige Messstellensysteme sowie Datenkommunikation (Holstenkamp und Radtke, 2018, S. 98).

Die steigende Digitalisierung birgt jedoch auch Risiken. Datenmanipulation, Datenmissbrauch/-diebstahl und Hackerangriffe stellen Gefahren eines vernetzten Energiesystems dar. Die Versorgungssicherheit der Stromversorgung als kritische Infrastruktur sollte im Zuge der Veränderungen nicht nachlassen (Doleski, 2017, S. 301-305).

Die Blockchain-Technologie, welche im folgenden Kapitel erläutert wird, könnte eine bedeutsame Rolle im Energiemarkt einnehmen und den Prozess der Digitalisierung vorantreiben.

4 Blockchain

Blockchain stellt eine Distributed Ledger Technology (DLT) dar. Sie kann als dezentrales Transaktionsregister verstanden werden, von der jeder vollwertige Netzwerkteilnehmer (Node) eine identische, redundante Kopie lokal abgespeichert hat. Jeder Teilnehmer im Netzwerk kann Transaktionen auf der Blockchain durchführen, welche in einem Block gesammelt werden und von bestimmten Netzwerkteilnehmern, den Minern, verifiziert an die Blockkette gehangen werden (Redlich u. a., 2018, S. 100-101).

Die erste Anwendung von Blockchain-Technologie war 2008 mit Einführung der Kryptowährung Bitcoin. Die Vorteile, dass mithilfe von Blockchain Transaktionen ohne Intermediär ablaufen können und ein „double-spending“[1] verhindert wird, sprachen für den Einsatz der Technologie (Nakamoto, 2008, S. 1; Perez-Sola u. a., 2017, S. 1-2). Was die Blockchain so interessant für Bereiche abseits von Kryptowährungen macht, ist die sichere Durchführbarkeit von Transaktionen mit digital abbildbaren Werten ohne Intermediär (Froystad und Holm, 2016, S. 7). Andere potentielle Anwendungsgebiete werden auch abseits von monetären Transaktionen gesehen. So kann eine Blockchain beispielsweise bei der Identifizierung (Führerscheine, Personalausweis, Reisepass) oder als digitaler Schlüsselersatz (Haus, Hotel, Auto) fungieren (Swan, 2015, S. 10).

4.1 Blockchain-Technologie

Eine Blockchain besteht aus miteinander verbundenen Blöcken. Diese Blöcke enthalten hauptsächlich Daten zu Transaktionen, die in einem bestimmten Zeitraum

[1] Double-spending ist das mehrmalige Ausgeben desselben digitalen Werts (Chohan, 2017, S. 1).

durchgeführt wurden. Darüber hinaus sind darin ein Hash[2] dieses Blocks, der Hash des vorherigen Blocks sowie ein Zeitstempel enthalten (Gupta, 2017, S. 13-14). Menschliche Teilnehmer nehmen im Netzwerk über Wallets (digitale Geldbörse) oder External Owned Accounts (EOA) teil. Diese verfügen über einen privaten und einen öffentlichen Schlüssel. Der öffentliche Schlüssel dient dazu, den Node im Netzwerk zu repräsentieren. Der private Schlüssel ist für die Verifizierung von Transaktionen notwendig (Röder, 2016, S. 14).

Da es verschiedene Blockchain-Kategorien für verschiedene Zwecke gibt, unterscheiden diese sich mehr oder weniger bei den Schritten der Transaktionen. Im Folgenden werden allgemeine Schritte erklärt, die bei Transaktionen einer Blockchain auftreten:

1) Der erste Schritt besteht aus einer Transaktionsformulierung. Ein Sender und ein Empfänger beschließen eine Transaktion durchzuführen. Der Sender veröffentlicht eine Nachricht in das Netzwerk, die den öffentlichen Schlüssel des Empfängers, den Transaktionsgegenstand und eine kryptographische Signatur als Hauptelemente enthält, welche die Echtheit des Senders sicherstellt.

2) Daraufhin erhält das Netzwerk die Nachricht, entschlüsselt die digitale Signatur und authentifiziert die Echtheit der Nachricht. Die Transaktion wird dann in einen Pool von auszuführenden Transaktionen aufgenommen.

3) Der dritte Schritt besteht aus der Blockerstellung. Der Pool an auszuführenden Transaktionen wird versucht von Minern schnellstmöglich zusammenzufassen. Der schnellste Miner veröffentlicht den Block zur Validierung im Netzwerk.

4) Validierungsteilnehmer erhalten den neuen Block und bestätigen ihn in einem Prozess, der die Zustimmung der Mehrheit des Netzwerks benötigt. So wird ein Konsens im Netzwerk erreicht. Dieser Validierungsprozess ist von Blockchain zu Blockchain unterschiedlich, hat jedoch das Ziel, Betrug bei Transaktionen zu verhindern.

5) Sobald der neue Block verifiziert wurde, wird er an den vorherigen Block angekettet und der neue Zustand der Blockchain wird nach und nach im Netzwerk von jedem Teilnehmer aktualisiert (Froystad und Holm, 2016, S. 11).

[2]Ein Hash-Wert ist eine einzigartige Abfolge aus Zahlen und Buchstaben, welche jeden Block eindeutig identifizierbar macht (Voshmgir, 2016, S. 13).

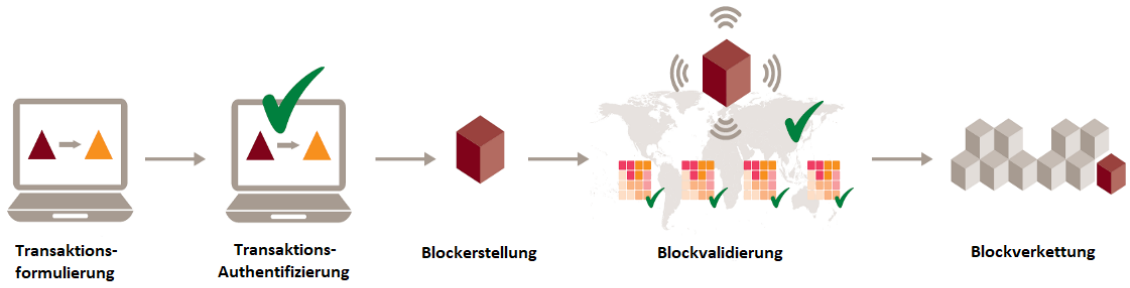

Abbildung 4.1: Transaktionsprozess in der Blockchain
(abgewandelt aus Hasse u. a., 2016, S. 5)

4.2 Blockchain Kategorien

Es gibt unterschiedliche Ansätze einer Blockchain. Im Grundansatz ist die Blockchain öffentlich, also für jeden zugänglich. Nach der Entwicklung der Kryptowährung Bitcoin und ihrer dazugehörigen Blockchain sind jedoch eine zweite und dritte Form der Blockchain erschienen, welche zugangsbeschränkt und somit nur für bestimmte Personen nutzbar sind, siehe Abbildung 4.2. Es kann zwischen öffentlich/privat sowohl bei der Teilnahme einer Blockchain als auch bei dem Validierungsprozess unterschieden werden (Schlatt u. a., 2016, S. 12).

Public: In einer öffentlichen Blockchain kann jeder teilnehmen. Es gelten keine Zugangsbeschränkungen. Jeder hat Zugriff auf die Daten der Blockchain, das heißt jede getätigte Transaktion ist einsehbar und jeder darf neue Transaktionen vorschlagen/durchführen. Die Teilnehmer bleiben dabei dennoch anonym, da nur deren öffentliche Schlüssel in Transaktionen sichtbar sind. In einem solchen System wird Einigkeit/Konsens normalerweise über einen Proof-of-Work (PoW)[3] erreicht, da dieser Konsensmechanismus unter anderem gut erforscht und verstanden ist (Schlatt u. a., 2016, S. 11-12; Chen u. a., 2018, S. 122). Bevor ein neuer Block an die Kette gehängt wird, muss ein recht komplexes Rechenrätsel gelöst werden. Der schnellste Miner bzw. die schnellsten Miner veröffentlichen den neuen Block

[3]Dieser Konsensmechanismus erfordert einen gewissen Rechenaufwand der Miner. Ein Hashwert wird dabei so lange verändert, bis er mit einem bestimmten Muster übereinstimmt (Schlatt u. a., 2016, S. 10).

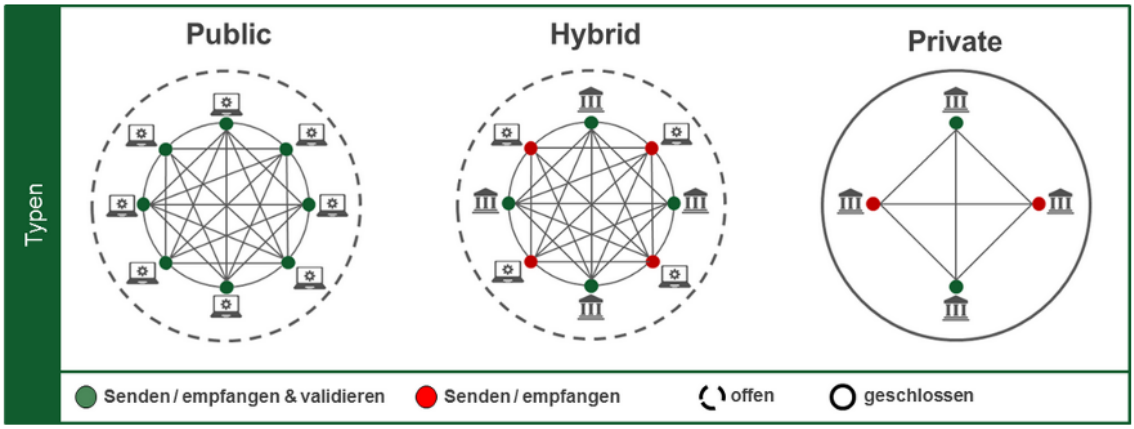

Abbildung 4.2: Blockchain-Kategorien
(Plazibat, 2016)

und alle Nodes aktualisieren ihre Blockchain (Gupta, 2017, S. 17). Für öffentliche Blockchains ist dieses Konzept sinnvoll, da ein Hacker die vereinte Rechenleistung des Netzwerks übertreffen muss, um einen Block zu manipulieren und schneller bei der Lösung des Rätsels für die Validierung zu sein. Ein solcher Rätselalgorithmus ist dementsprechend essentiell für die Sicherheit einer Blockchain. Er muss einen gewissen Grad an Komplexität aufweisen, um eine Herausforderung darzustellen, gleichzeitig jedoch nicht zu schwierig sein, da die Lösung des Problems Rechenleistung und Elektrizität erfordert (Peters und Panayi, 2015, S. 5-6).

Private: Im Gegensatz zu einer öffentlichen Blockchain sind private Blockchains teilnahmeseitig nur bestimmten, von einer höheren Instanz genehmigten Teilnehmern zugänglich. Dadurch geht eine gewisse Dezentralisierung verloren. Die Anwendung einer solchen Blockchain ist jedoch unter bestimmten Voraussetzungen sinnvoll, beispielsweise wenn die Nutzung der Blockchain nur Personen innerhalb eines Unternehmens oder mehrerer Unternehmen gestattet sein soll. Der Konsens wird in der Regel nicht durch PoW erreicht, da dies zu aufwändig ist und durch die Zugangsbeschränkung mehr oder weniger hinfällig wird. Konsens kann unter anderem durch Proof-of-Stake (PoS)[4] erreicht werden (Gupta, 2017, S. 16). Vor-

[4]Die Blockchain wird durch Netzknoten aktualisiert, welche einen großen Anteil an Werten der Blockchain halten. Dadurch soll ein Anreiz geschaffen werden, um das System korrekt zu halten (Schlatt u. a., 2016, S. 12).

teile einer privaten Blockchain sind niedrigere Energiekosten für Transaktionen, höherer Datenschutz und ein schnellerer Validierungsprozess (Froystad und Holm, 2016, S. 14). Beispiele für private Blockchains sind Eris, Hyperledger und Ripple (Peters und Panayi, 2015, S. 7).

Hybrid: Es ist außerdem möglich, Mischformen von öffentlichen und privaten Blockchains aufzusetzen. Sie kann beispielsweise öffentlich zugänglich sein, der Validierungsprozess eines Blocks obliegt jedoch nur bestimmten Teilnehmern. Diese Blockchainform wird auch als „Konsortium Blockchain“ bezeichnet (Voshmgir, 2016, S. 16).

4.3 Eigenschaften von Blockchain-Technologie

Blockchain-Technologie besitzt mehrere Eigenschaften, die sie interessant für den Einsatz im Stromsektor machen. Bei der Nutzung von Smart Contracts erhöht sich der Automatisierungsgrad, was wiederum die Geschwindigkeit verbessert und für eine zuverlässige Abwicklung sorgt (Peter u. a., 2017, S. 24).

Ein weiterer Faktor sind Transaktionskosten. Das Transaktionsvolumen im Spotmarkt nimmt seit einiger Zeit stetig zu (Statista, 2018; Gobmaier, 2018). Da bei Blockchain-Transaktionen kein Intermediär benötigt wird, fallen dessen Gebühren weg und einzig die Rechenleistung der Miner muss aufgewendet werden (Jacob u. a., 2015, S. 1403).

Durch die Kryptographie, Konsensbildung und die Immutabilität der gespeicherten Blöcke ist das System sicher. Ein Hacker oder Angreifer müsste über 50 % des Netzwerks kontrollieren, um verifizierte Änderungen der Daten vorzunehmen (Schlatt u. a., 2016, S. 36). Des Weiteren stellt Blockchain-Technologie Transparenz bei gleichzeitiger relativer Anonymität bereit. Netzwerkteilnehmer können die Transaktionen in den Blöcken einsehen, was insbesondere für Verteilnetzbetreiber interessant sein könnte, um einen Überblick auf der unteren Spannungsebene zu bekommen. Die in den Transaktionen aufgeführten Nodes werden nur mit ihren öffentlichen Schlüsseln dargestellt, was keine Rückschlüsse auf die Personen zulässt (Kounelis u. a., 2017, S. 3; Röder, 2016, S. 14).

Bei zunehmender Dezentralisierung der Stromversorgung ist es naheliegend, den Handel ebenso zu dezentralisieren und DLT zu benutzen. Daten über Transaktio-

nen werden lokal bei Nodes im Netzwerk gespeichert und aktualisiert (Niranjanamurthy u. a., 2018, S. 1-2).

5 Ethereum Blockchain

Nach Bitcoin stellt Ethereum die nächstgrößte Kryptowährung Ether hinsichtlich der Marktkapitalisierung dar (CoinMarketCap, 2018). Ein wesentlicher Unterschied zu Bitcoin ist, dass Ethereum neben dem Austausch von Ether die Ausführung sogenannter Smart Contracts ermöglicht. Smart Contracts sind Programme, welche in einer eigenen Ausführungsumgebung, der Ethereum Virtual Machine (EVM), aufgerufen werden können (Bergstra und Burgess, 2018, S. 29). Mit Unterstützung der Smart Contracts ist das Ziel von Ethereum ein System zu entwickeln, in dem Individuen und Organisationen sicher und ohne gegenseitiges Vertrauen miteinander interagieren können (Wood, 2018, S. 1). Konsens wird in Ethereum über PoW erreicht (Wood, 2018, S. 14). Sowohl EOAs als auch Smart Contracts besitzen einzigartige Adressen, die sie im Netzwerk eindeutig bestimmen. Mithilfe dieser Adressen können Teilnehmer im System miteinander interagieren, wobei Unterschiede zwischen einer Transaktion und einer Nachricht gemacht werden. Eine Transaktion ist eine Nachricht eines EOA an einen anderen Account und enthält unter anderem eine Signatur des Senders und einen Wert in Wei[1], welcher der Nachricht beigefügt ist. Eine Nachricht hingegen kann als Funktionsaufruf eines Smart Contracts von einem anderen Smart Contract verstanden werden (Ethereum community, 2018a). Der „World state“ zeigt den aktuellen Zustand der Blockchain an und entspricht einem mapping[2] aus Adressen und Account Zuständen. Ein Account Zustand beinhaltet unter anderem die 'balance', also den digitalen Kontostand in Wei dieses Accounts. Innerhalb eines Blockzyklus werden Veränderungen dieser Zustände durch Transaktionen vorgenommen. Nachdem ein neuer

[1] Wei ist die kleinste Währungseinheit im Ethereum Netzwerk, ein Ether entspricht 10^{18} Wei (Buterin, 2017, S. 30).

[2] Ein mapping ist ein Schlüssel-Wertepaar. Beispielsweise kann ein Datentyp address als Schlüssel fungieren. Mit bestimmten Adressen können dadurch bestimmte Werte anderer Datentypen wie integer verbunden und abgerufen werden (Ethereum community, 2018n).

Block erstellt und vom Netzwerk verifiziert wurde, steht ein neuer World State fest (Buterin, 2017, S. 3).

5.1 Smart Contracts

Smart Contracts sind Programme in der Ethereum Blockchain, welche Funktionen zur Verfügung stellen, die von EOAs und anderen Smart Contracts genutzt werden können. Dies kann beispielsweise die Ausführung einer ganz bestimmten Aktion sein, wenn gewisse Voraussetzungen eintreten. Smart Contracts lassen sich in mehreren Programmiersprachen erstellen, wobei Solidity die bekannteste ist (Ethereum community, 2018l). Durch die Turing-Vollständigkeit[3] von Smart Contracts in der EVM können jegliche Berechnungen von diesen Programmen ausgeführt werden (Chandra u. a., 2017, S. 1). Sobald Smart Contracts in die Blockchainumgebung eingebracht wurden, kann der Code nicht mehr verändert werden (Xu u. a., 2018, S. 6).

Ein Smart Contract und dessen Funktionen können analog zu Klassen und Methoden anderer objektorientierter Programmiersprachen gesehen werden (Sommerville, 2011, S. 131; Frantz und Nowostawski, 2016, S. 2). Die Daten und Funktionen eines Smart Contracts können mit unterschiedlichen Nutzbarkeiten (in der Dokumentation als 'Visibility' gekennzeichnet) versehen werden, welche nun erläutert werden.

external: Externe Funktionen können von anderen Smart Contracts und über Transaktionen aufgerufen werden. Variablen können nicht als extern definiert werden.

public: Öffentliche Funktionen können sowohl extern als auch intern aufgerufen werden. Öffentliche Variablen erhalten automatisch eine get-Funktion.

internal: Interne Funktionen und Variablen können nur innerhalb des Smart Contracts, in dem sie programmiert wurden, genutzt werden.

[3] Turing-Vollständigkeit beschreibt die Lösung jeglicher mathematischer Probleme durch Algorithmen (Vossen und Witt, 2016, S. 291-292).

private: Auf private Funktionen und Variablen können andere Smart Contracts nicht zugreifen.

Auch wenn der Zugriff auf Daten und Funktionen durch entsprechende Nutzbarkeiten für andere Smart Contracts beschränkt werden kann, sind die Inhalte eines Smart Contracts dennoch für jeden außerhalb der Blockchain einsehbar (Ethereum community, 2018b).

Variablen können auch mit dem Zusatz 'constant' definiert werden. Dadurch wird der Wert dieser Variable unveränderbar und muss direkt bei der Initiierung gesetzt werden (Ethereum community, 2018e).

In Solidity können verschiedene Datentypen erstellt werden, welche teilweise auch in anderen Programmiersprachen auftauchen. Ein bool ist beispielsweise ein Datentyp, welcher nur die Werte true und false annehmen kann. Ein spezifischer Datentyp ist address. Eine Variable vom Typ address besteht aus einem Hexadezimalwert mit vielen Ziffern, welche als einzigartige Identifikationsnummer (ID) und Zugriffspunkt von EOAs oder Smart Contracts dient (Ethereum community, 2018o).

Bei dem Einstellen eines Smart Contracts in die Blockchain gibt es die Möglichkeit eine spezielle Funktion, den *constructor*, auszuführen. Diese Funktion wird bei der Erstellung des Smart Contracts einmalig ausgeführt. Dies kann beispielsweise dazu genutzt werden, eine 'owner' Adressvariable entsprechend 'msg.sender'[4] zu setzen, welche im späteren Zyklus des Smart Contracts bestimmte Privilegien bei Funktionsausführungen erhält (Ethereum community, 2018c).

Eine weitere spezielle Funktion ist die *fallback function*. Sie hat keinen Namen, keine Input Parameter oder Rückgabewerte. Diese Funktion wird ausgeführt, wenn ein Aufruf eines Smart Contracts ohne Aufruf einer bestimmten Funktion erfolgt. Sie wird außerdem ausgeführt, wenn Ether ohne zusätzliche Daten an den Smart Contract geschickt wird (Ethereum community, 2018d).

Ein Smart Contract muss gewisse Inhalte aufweisen, damit er auf andere Smart Contracts zugreifen kann. Es müssen Interfaces (Schnittstellen) programmiert wer-

[4]Der Befehl 'msg.sender' gibt die Adresse des EOA oder Smart Contracts zurück, welche für die Ausführung einer Funktion verantwortlich ist (Ethereum community, 2018o).

den, welche die Funktionen der aufzurufenden Smart Contracts widerspiegeln. Dabei müssen der Funktionsname sowie die Input und Output Parameter der Funktionen definiert werden. Was innerhalb dieser Funktion passiert, muss nur in den aufgerufenen Smart Contracts programmiert sein. Mithilfe der Adresse des aufzurufenden Smart Contracts können alle in dem Interface definierten Funktionen entweder über 'call' oder 'delegatecall' von einem anderen Smart Contract aufgerufen werden, sofern die Nutzbarkeit der aufzurufenden Funktion dies zulässt. Der Unterschied zwischen den Befehlen 'call' und 'delegatecall' liegt darin, wessen Daten und Speicher für die Ausführung der Funktion genutzt werden. Ruft ein Contract A Funktionen von Contract B auf, wird per 'call' der Speicher von Contract B beansprucht. Beim Befehl 'delegatecall' würde der Speicher von Contract A genutzt werden (Stack Overflow, 2016a).

Ein Beispiel einer solchen Funktionssignatur ist im Code Interface dargestellt. Das Interface gehört zu einem Contract 'Callee'. Diese Bezeichnung muss nicht den Namen des aufzurufenden Contracts haben sondern kann beliebig gewählt werden. Über diese Bezeichnung kann eine Instanz des aufzurufenden Contracts gesetzt werden, beispielsweise über eine Zeile 'Callee c;'. Die Namen der aufzurufenden Funktionen müssen jedoch zwischen dem Interface und dem Callee Contract übereinstimmen. Auch die Input und Output Parameter sollten von den Datentypen und der Anzahl übereinstimmen. Es ist zwar teilweise möglich mit nicht passenden Input oder Output Parametern dennoch Funktionen aufzurufen, doch das Ergebnis könnte unkontrollierbar sein. Beispielsweise kann im Callee Contract eine Funktion einen Datentyp address als Output Datentyp definieren und ein anderer Contract diese Funktion mit einem Output Datentyp integer im Interface aufrufen. Ein address Datentyp ist ein Hexadezimalwert. Dadurch würde im aufrufenden Contract die Adresse in einen Dezimalwert umgewandelt und ausgegeben werden. Dies wurde mit den Testcodes Caller und Upgrade überprüft.

```
1 contract Callee {
2     function setint(int);
3     function getint() constant returns(int);
4     function getstring() returns(string);
5 }
```

Code 5.1: Solidity Beispiel Interface eines Smart Contracts

5.2 Herausforderungen und Limitationen

Trotz der vielen Möglichkeiten der Ethereum Blockchain und ihren Smart Contracts bestehen einige Hürden und Grenzen. An mehreren dieser Probleme wird bereits nach Verbesserungen gesucht. Ethereum entwickelt sich stetig weiter, was sich manchmal in sogenannten Hard Forks widerspiegelt. Eine Hard Fork ist ein Upgrade des Ethereum Protokolls, welches Verbesserungen des Systems bringen soll und keine Abwärtskompatibilität aufweist. Dabei können Nodes diese neuen Softwareversionen installieren, sind jedoch auch nicht dazu gezwungen. Es kann passieren, dass sich die Blockchain aufteilt und fortan bei einer Hard Fork zwei Blockchains parallel mit unterschiedlichen Versionen laufen, wenn ein gewisser Teil der Nodes auf das Upgrade verzichtet (Etherchain, 2018; Kiffer u. a., 2017, S. 1-2). Auf bestehende Herausforderungen in der Ethereum Blockchain wird nun konkret eingegangen.

5.2.1 Skalierbarkeit

Ein allgemeines Problem von öffentlichen Blockchains ist die Skalierbarkeit. Mit steigender Anzahl an Teilnehmern im Netzwerk steigt tendenziell auch die Anzahl an Transaktionen (Peters und Panayi, 2015, S. 7). In dem PoW gesteuerten Ethereum Netzwerk muss jeder vollwertige Node jede einzelne Transaktion sukzessiv durchführen, nachdem ein Konsens erreicht wurde (Buterin, 2015, S. 1; Vukolić, 2017, S. 1). Durch dieses Vorgehen wird die Sicherheit des Netzwerks garantiert, gleichzeitig aber auch die Geschwindigkeit reduziert (Peter u. a., 2017, S. 46; Scherer, 2017, S. 21). Die Folgen sind die Durchführung von maximal 20 Transaktionen pro Sekunde, Blockzeiten von etwa 14 Sekunden und Blockgrößen von etwa 20 kbyte (Peck, 2017, S. 60; Etherscan, 2018b; Etherscan, 2018a). Demgegenüber schafft beispielsweise VISA etwa 50.000 Kreditkarten Transaktionen pro Sekunde (Hasse u. a., 2016, S. 11).

Zur Verbesserung der Geschwindigkeit wird an verschiedenen Ideen gearbeitet. Eine Überlegung ist die Umstellung des Konsensmechanismus von PoW zu PoS, was auch unter 'Casper' bekannt ist. Dadurch wird der Mining-Prozess zum größten Teil in die Hände der Teilnehmer gelegt, welche große finanzielle Anteile im

System halten und somit ein starkes Interesse daran haben, dass das System sicher bleibt. Dies verringert den Rechenaufwand und den Energieverbrauch des Mining-Prozesses. Es ist jedoch nicht klar, wie sicher die Umstellung auf PoS wäre (Buterin und Griffith, 2017, S. 1; Peter u. a., 2017, S. 46).

Als weitere Option wird das sogenannte 'sharding' diskutiert. Sharding beinhaltet das Aufteilen eines Prozesses in mehrere Unterprozesse, welche von unterschiedlichen Parteien durchgeführt werden können und in Summe den Gesamtprozess ausführen. Das würde bedeuten, dass die Ausführung von Smart Contracts im Netzwerk aufgeteilt wird und nur bestimmte Nodes bestimmte Transaktionen validieren müssten. Im Zusammenhang mit Ethereum wird dabei auch die Aufteilung des Ledgers in Betracht gezogen, sodass nicht jeder Node die gesamte Blockchain gespeichert haben muss. Offene Fragen sind dabei nach welchem Muster das Ledger aufgespalten wird und wie eine Interoperabilität aufgeteilter Nodes bei gleichzeitiger Beibehaltung des shardings erreicht wird (Bergstra und Burgess, 2018, S. 8; Buterin, 2016, S. 21-29).

5.2.2 Gas

Die Ausführung von Transaktionen und Smart Contract Funktionen ist für Teilnehmer nicht kostenlos. Jede Aktion, die in einem Block festgehalten wird, kostet eine Gebühr. Diese Beiträge werden auch als Gas Kosten bezeichnet. Gas ist eine interne Währung speziell zum Begleichen dieser Gebühren und wird über Ether bezahlt (Rosenberger, 2018, S. 54). Durch diesen Mechanismus entsteht ein Anreiz für Miner Transaktionen zu validieren und Smart Contracts in der EVM auszuführen, da diese danach die Gebühren erhalten. Die Ausführung von Smart Contract Code und generell Änderungen in der Blockchain sind mit festgelegten Gas Verbräuchen verbunden (Wood, 2018, S. 7). Jeder Transaktion ist ein Gas Limit und ein Gas Price vom ursprünglichen EOA beigefügt. Das Gas Limit gibt an, wieviel Gas der EOA bereit ist für die Ausführung der Transaktion zu zahlen. In Verbindung mit dem angegebenen Gas Price können Miner sich entscheiden, ob sie die Transaktion validieren wollen. Sollte die Ausführung mehr Gas benötigen als im Gas Limit festgelegt, wird die Transaktion rückgängig gemacht. Der Miner erhält dennoch das Gas, um für seinen bis dahin getätigten Rechenaufwand entschädigt zu werden. So

wird verhindert, dass Teilnehmer Ressourcen der Miner für sinnlose Transaktionen verschwenden, ohne dafür bezahlen zu müssen (Destefanis u. a., 2018, S. 21). Ein Nachteil dieses Mechanismus ist, dass komplexe Algorithmen in Smart Contracts teurer in ihrer Ausführung sind. Die durchschnittliche Transaktionsgebühr beträgt zurzeit etwa 0.15 USD (BitInfoCharts, 2018b). Die Grundgebühr jeder Transaktion ist 21.000 Gas. Zudem ist es vergleichsweise teuer den langfristigen Speicher, den storage der Blockchain, zu beanspruchen. Einen 32 byte großen Speicherplatz von Null auf Nicht-Null zu ändern kostet 20.000 Gas. Einen bereits belegten storage Speicherplatz auf einen Wert ungleich 0 zu verändern kostet 5.000 Gas (Wood, 2018, S. 25). Der Gas Price mit einer mittleren Validierungszeit von etwa 510 Sekunden lag am 17.10.18 bei 2 Gwei, was 2x10^{-9} Ether entspricht (Etherscan, 2018c). Bei einem gegenwärtigen Kurs von 181 €/Ether kostet die erstmalige storage Speicherung von 1 kbyte also (BitInfoCharts, 2018a):

$$\frac{1024}{32\ byte}\, byte \cdot 20000\, Gas \cdot 2 \cdot 10^{-9}\, \frac{Ether}{Gas} \cdot 181\, \frac{€}{Ether} = 0,23\, €$$

Im Vergleich dazu liegen Preise für Speicher in einer Cloud bei einigen Cent pro Gigabyte (GB) und Monat (Mazrekaj u. a., 2016, S. 84-85).

5.2.3 Komplexität

Die Ausführung von Smart Contracts in Ethereum ist oftmals mit monetären Werten verbunden. Daher ist die sichere Programmierung von Smart Contracts von großer Wichtigkeit. Es wurden bereits mehrmals Schwachstellen in Smart Contracts von Angreifern ausgenutzt und Ether im Wert von mehreren Millionen Euro entwendet (Tsankov u. a., 2018, S. 1). Die Interaktion mit Smart Contracts sollte demnach möglichst nur erfolgen, wenn ein gewisses Maß an Grundwissen bezüglich der Funktionen vorhanden ist. Doch bei komplexen Smart Contracts oder einem Netzwerk aus interagierenden Smart Contracts dürfte es für unbedarfte Teilnehmer unmöglich sein nachzuvollziehen, wie sie funktionieren (Frantz und Nowostawski, 2016, S. 1).

5.2.4 Kommunikation Off-Chain

Die Ethereum Blockchain ist ein abgeschlossenes System, in dem Teilnehmer untereinander Daten und Werte austauschen können. Die Kommunikation mit der Außenwelt ist nicht ohne Weiteres möglich. Externe Daten können nur über Transaktionen in die Blockchain gelangen. Sollte es für eine Anwendung beispielsweise notwendig sein bestimmte Preise in gewissen Intervallen abzurufen, stellt dies ein Problem dar. Ein Lösungsansatz sind Oracles. Ein Oracle ist eine Schnittstelle zwischen der Außenwelt und der Blockchain. Über diesen Kommunikationskanal können Daten außerhalb der Blockchain eingelesen werden. Durch die Transaktionsgebundenheit von Daten erfolgt die Aufnahme der Informationen jedoch asynchron. Ein Oracle Contract kann die aufgenommenen Daten an relevante andere Smart Contracts weiterleiten, stellt hierbei aber eine zentrale Instanz dar. Der Adresse, welche die Daten in den Oracle Contract einbringt, muss vom Netzwerk vertraut werden. Sollten über ein Oracle falsche Daten ins System gebracht werden, stellt dies ein Risiko dar (Wohrer und Zdun, 2018a, S. 4).

5.2.5 Wartung und Updates

Smart Contract Code in der Blockchain kann nicht mehr verändert werden. Dies schafft Sicherheit für Teilnehmer, welche mit Smart Contracts interagieren wollen. Es ist sicher, dass ein Aufruf über eine Adresse einen nicht modifizierbaren Code ausführt. Die einzige Möglichkeit, einen Smart Contract anzupassen und zu verändern besteht darin, einen neuen Smart Contract zu programmieren und ins System einzubringen. Dies ist insbesondere dann sinnvoll, wenn Fehler in einem Smart Contract festgestellt werden. Doch andere Contracts, welche mit dem fehlerbehafteten Smart Contract in diesem Netzwerk interagieren, müssen nach einem Update die neue Adresse des Smart Contracts mitgeteilt bekommen. In einem Netzwerk bestehend aus wenigen Smart Contracts könnte manuell in jedem von einem Update abhängigen Smart Contract die neue Adresse gesetzt werden. Doch je mehr Smart Contracts in einem solchen System vorhanden sind, desto aufwändiger und ineffizienter wird dieser Vorgang. Daher wäre ein weitgehend automatisiertes System für das Management von Smart Contracts eine bevorzugte Lösung (Wohrer und Zdun, 2018a, S. 6; Xu u. a., 2018, S. 23).

6 Updatable Smart Contracts

Eine der Stärken von Blockchain-Technologie, die Immutabilität bzw. Unveränderbarkeit, bringt also auch Nachteile mit sich, wie in Unterabschnitt 5.2.5 erwähnt wurde. Für eine Dezentralisierte Applikation (DApp) oder ein Netzwerk aus Smart Contracts könnte es sich als problematisch herausstellen, wenn kein Updatemechanismus im System implementiert ist. Ohne Updates ist die Flexibilität und Wartung des Systems beschränkt.

Neben der Möglichkeit eines Updates könnte es zudem erforderlich sein, neue Funktionen in einem Smart Contract einzuführen oder bestehende Inputs oder Outputs von Funktionen zu modifizieren. Eine potentielle Upgradability ist jedoch nicht Teil dieser Arbeit, dazu mehr in Kapitel 9.

6.1 Konzepte für Updatable Smart Contracts

Grundsätzlich gibt es 3 verschiedene Ansätze, wie ein Smart Contract Update ermöglicht werden kann. Auf jede Form wird nun konkret eingegangen. Diese Darstellung erhebt jedoch keinen Anspruch auf Vollständigkeit. Es ist durchaus möglich, dass es noch andere Formen von Updatable Smart Contracts gibt, welche nicht berücksichtigt wurden.

Die Ansätze unterscheiden sich hinsichtlich Komplexität und bringen verschiedene Vor- und Nachteile mit sich. Der optimale Updatemechanismus dürfte sich von Anwendungsfall zu Anwendungsfall unterscheiden.

6.1.1 Trennung von Daten und Logik

Die Trennung von Daten und Logik stellt einen relativ simplen Ansatz zur Updatability von Smart Contracts dar. Es werden 2 Contracts eingesetzt:

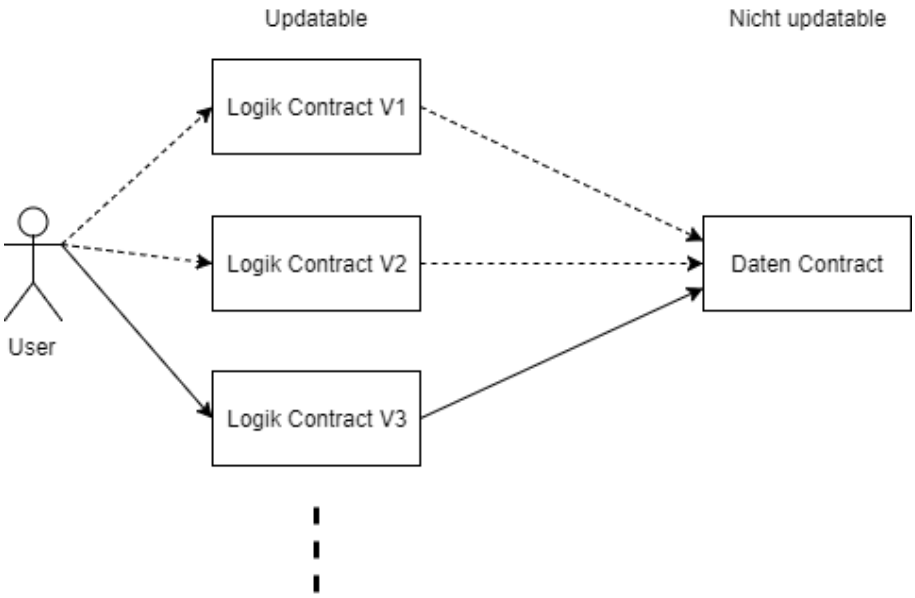

Abbildung 6.1: Konzept der Trennung von Daten und Logik

Einer beinhaltet nur die Daten, also Variablen und mappings. Diese können über get- und set-Funktionen abgerufen und verändert werden. Der andere Contract stellt beispielsweise Funktionen mit Algorithmen bereit, um die Daten des anderen Contracts zu verändern. Dieser Logik Contract ist derjenige, welcher gegebenenfalls Updates erhält. So wird sichergestellt, dass bei einem Update die Daten, also der storage des Daten Contracts, weiterhin vorliegen (Wiesner, 2017). Zur Sicherheit sollte nur der Logik Contract Zugriff auf den Daten Contract erhalten, da sonst eventuell unerwünschte Änderungen im Daten Contract von Usern vorgenommen werden könnten. In diesem Konzept erfolgt ein Zugriff des Logik Contracts auf den Daten Contract über den Befehl 'call'. Wie in Abschnitt 5.1 beschrieben wird dabei der Speicher des aufgerufenen Contracts genutzt, welcher über die Algorithmen des Logik Contracts verändert wird. Im Falle eines Updates des Logik Contracts muss das Zugriffsrecht der set-Funktionen ebenfalls aktualisiert werden, damit die neue Adresse des Logik Contracts die Daten des anderen Contracts verändern kann (Tanner, 2018).

Abbildung 6.1 zeigt den schematischen Aufbau eines solchen Ansatzes. Wenn das System wie oben beschrieben implementiert wird, kann ein User nur über den

Logik Contract Änderungen des Speichers im Daten Contract vornehmen. Es stellt kein Risiko dar die Get-Funktionen des Daten Contracts von allen abrufbar zu implementieren, da diese einen read-only Aufruf darstellen und keine Änderungen vornehmen. Die gepunkteten Pfeile stellen veraltete Zugriffe von alten Versionen des Logik Contracts dar. Version 3 (V3) ist die aktuellste Version und demzufolge die einzige, welche Änderungen im Daten Contract vornehmen kann.

Es wäre auch möglich, den Daten Contract zu aktualisieren. Dies wird erforderlich, wenn neue Variablen eingeführt werden müssten. Dann sollte es in den Logik Contracts eine set-Funktion zum Setzen der Adressvariable des Daten Contracts geben, welche neu gesetzt werden kann. Zudem wird ein Datentransfer vom Speicher des alten Daten Contracts in den neuen notwendig. Im *constructor* des neuen Daten Contracts könnten alle get-Funktionen des alten Daten Contracts aufgerufen werden und so der eigene Speicher entsprechend der erhaltenen Werte gesetzt werden. In diesem Fall wäre die Trennung in Daten Contract und Logik Contract mehr oder weniger hinfällig. Diese Trennung wird genau deswegen vorgenommen, weil nur der eine Teil die Möglichkeit von Updates erhalten soll und der andere Teil voraussichtlich kein Update benötigt.

Der Vorteil eines solchen Systems ist die Übersichtlichkeit, da nur 2 Contracts bestehen. Alte Versionen des Logik Contracts können über einen 'selfdestruct' Befehl deaktiviert werden, falls dies erwünscht ist (Ethereum community, 2018q). Es sollte bei einem Update jedoch ohnehin sichergestellt werden, dass nur die aktuellste Logik Version Zugriff auf set-Funktionen im Daten Contract hat. Des Weiteren wird durch ein solches Konzept verhindert, dass nach jedem Update die Daten in die neue Version übergeben werden müssen, was je nach Menge der Daten viel Gas kosten könnte (Wohrer und Zdun, 2018a, S. 6).

Die Nachteile sind, dass die Änderung komplexer Datenstrukturen wie struct[1] nur aufwändig über den Logik Contract möglich sind. Zudem stellt dieser Ansatz nur einen Updatemechanismus für einen der beiden Smart Contracts dar. Ein miteinander interagierendes Netzwerk aus Smart Contracts ist in diesem Konzept nicht vorgesehen. Ein weiterer Nachteil ist, dass externe Aufrufe kostenintensiver

[1]Eine struct ist ein Datentyp, welcher andere Datentypen enthalten kann. Beispielsweise könnte eine struct 'Mensch' mit mehreren Datentypen zu Eigenschaften und Attributen eines Menschen wie Alter, Größe und Gewicht erstellt werden (Ethereum community, 2018m).

hinsichtlich Gas sind als interne Aufrufe. Jede Änderung im Daten Contract über den Logik Contract kostet demzufolge im Vergleich zur Nicht-Anwendung dieses Konzepts etwas mehr Gas (Gupta, 2018).

6.1.2 Proxy Contract

Die Einführung eines Proxy Contracts ist ähnlich wie der Ansatz zur Trennung von Daten und Logik. Eine Proxy, also ein Stellvertreterobjekt, dient in einer Blockchainumgebung dazu, Aufrufe weiterzuleiten. Die Weiterleitung erfolgt über den Befehl 'delegatecall'. Ein Problem dabei ist, dass mit diesem Befehl Rückgabewerte nur über assembly Code möglich sind, was die Verständlichkeit des Codes erschwert und die Komplexität erhöht (Daonomic, 2018; Ethereum community, 2018k).

Abbildung 6.2 zeigt den grundlegenden Aufbau eines solchen Systems. In einer Aufruf-Kette wird zuerst ein Caller Contract vom User aufgerufen. Dieser wird über den Proxy Contract an den Logik Contract weitergeleitet. Es wird der Speicher des ursprünglichen Caller Contracts im Kontext des Logik Contracts beansprucht, da die Weiterleitung über 'delegatecall' erfolgt. Es ist auch möglich di-

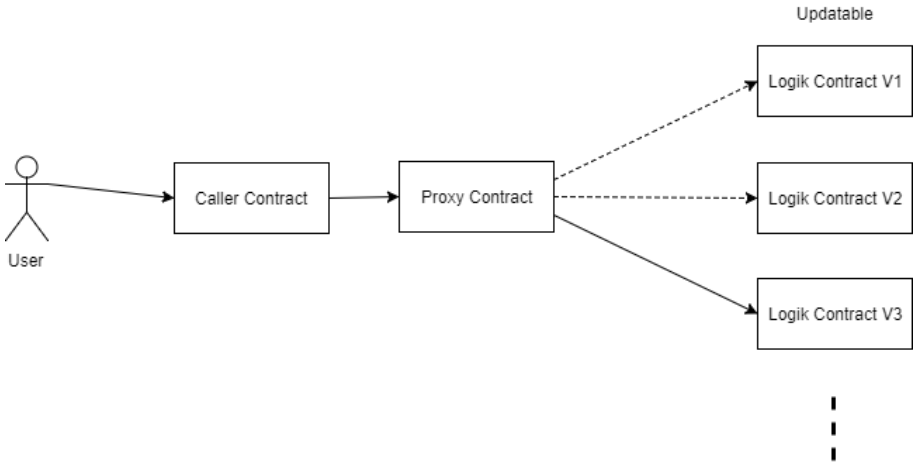

Abbildung 6.2: Konzept eines Proxy Contracts

rekt den Proxy Contract aufzurufen und dessen Speicher für die Ausführung einer

Funktion zu nutzen. Dann entspräche der Proxy Contract sowohl einem Stellvertreterobjekt als auch einem Daten Contract (Rauf, 2018).

Der Proxy Contract Ansatz und verschiedene Ausprägungen davon wurden von einem Entwicklerteam programmiert und getestet (Nadolinski, 2018; Zeppelin, 2018).

Wie bei der Trennung von Daten und Logik erhöht die Einführung eines Proxy Contracts die Gas Kosten, da externe Aufrufe zusätzliche Kosten verursachen. Ein User, der mit einem updatable Logik Contract über einen Proxy Contract interagieren will, sollte sich vorher intensiv mit dem Logik Contract auseinandersetzen. Durch den Befehl 'delegatecall' wird die Macht über den Speicher des Caller Contracts an den Logik Contract weitergegeben, sodass dieser Änderungen vornehmen könnte, welche gegebenenfalls unerwünscht sein können.

Auch bei diesem Ansatz wird jedoch nur die Updatability eines einzelnen Smart Contracts berücksichtigt.

6.1.3 Registry Contract

Im Ethereum Netzwerk kann ein Registry Contract dazu genutzt werden, eine Adressen-Verwaltung für Smart Contracts aufzubauen. Nach dem Einsetzen eines Smart Contracts in der Blockchain kann dieser über seine Adresse in einem Registry Contract angemeldet werden. Sollte ein Update erforderlich sein, kann nach dem Einfügen der Update-Version in die Blockchain die Adresse im Registry Contract aktualisiert werden. Die Interaktion mit anderen Smart Contracts erfolgt über ein Abrufen der aktuellen Adresse im Registry Contract. Das Grundprinzip ist in Abbildung 6.3 dargestellt (Xu u. a., 2018, S. 23-24).

1) Ein User ruft im ersten Schritt eine Funktion in einem Caller Contract auf, welche wiederum einen anderen Smart Contract aufrufen will.

2) Vor dem Aufruf des Callee Contracts wird der Registry Contract nach der aktuellen Adresse des Callee Contracts abgefragt.

3) Zum Schluss erfolgt der Aufruf des Callee Contracts vom Caller Contract.

Dabei muss ein Mechanismus eingebaut werden, über den Smart Contracts die Adressen anderer Smart Contracts abonnieren können, was ein Publish/Subscribe

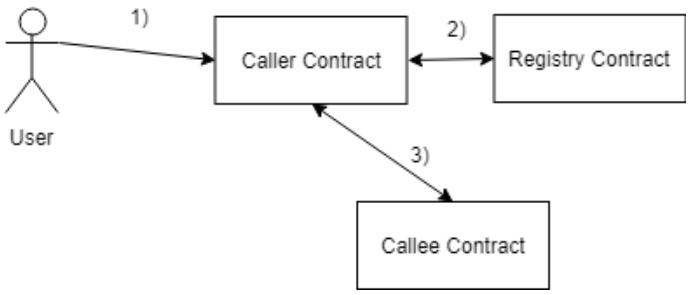

Abbildung 6.3: Konzept eines Registry Contracts

Muster darstellt. Es muss einen eindeutigen Schlüssel geben, über den die Adresse eines Smart Contracts aufgerufen werden kann.

Die Möglichkeit zur Verwaltung der Adressen vieler Smart Contracts stellt das geeignetste Konzept für das Management eines Netzwerks aus Smart Contracts dar. Aus diesem Grund wird das Entwurfsmuster auf diesem Konzept aufgebaut.

7 Entwurfsmuster des Registry Contracts

Zu Beginn des Kapitels werden die funktionalen Anforderungen an das System festgehalten. Anschließend erfolgt eine weitergehende Detaillierung, welche mit Diagrammen untermauert wird. Alle Funktionalitäten wurden in der browser-basierten Entwicklungsumgebung 'Remix' getestet (Remix, 2018). Am Ende des Kapitels wird der umgesetzte Code des Entwurfsmusters vorgestellt und erläutert.

7.1 Funktionale Anforderungen

In diesem Abschnitt werden Anforderungen an das System gestellt und Funktionalitäten erörtert, welche das Entwurfsmuster bereitstellen sollte. Auf Grundlage dieser Anforderungen erfolgt eine weitergehende Konzeptionierung des Entwurfsmusters. Die Anforderungen werden mit IDs versehen, auf die im späteren Teil dieses Kapitels verwiesen wird. Tabelle 7.1 gibt einen Überblick über die gestellten Anforderungen.

Tabelle 7.1: Anforderungen an das Entwurfsmuster

ID	Bezeichnung	Beschreibung
A1	Zugriff auf Smart Contracts und dazugehörige Registerattribute über Namen	Adressen sind lange Hexadezimalwerte, was sie für menschliche User unhandlich macht. Es wäre angenehmer, wenn über Namen auf Smart Contracts zugegriffen und Daten zu einem Smart Contract im Registry Contract abgerufen werden können.
A2	Registrierung von Smart Contracts und deren Updateversionen	Jeder Smart Contract kann im Registry Contract über seine Adresse registriert werden. Dadurch erhalten andere Smart Contracts Zugang zu diesem. Bei Einführung eines Updates wird ein Eintrag der neuen Adresse im Registry Contract angelegt.
A3	Automatisierung	Das ganze System soll möglichst automatisiert und mit möglichst wenig Funktionsaufrufen von Usern arbeiten.
A4	Sicherheit	Ein Registereintrag eines Smart Contracts muss mit einem „Besitzer“ verknüpft werden, sodass nur dieser Besitzer Updateversionen eines Smart Contracts registrieren darf. Dieses Zugriffsrecht kann auch an andere EOAs oder Smart Contracts abgegeben werden.
A5	Updatability des Registry Contracts	Der Registry Contract soll selber ebenso updatable sein. Wichtig ist dabei, dass bei einem Update das Register der Vorgängerversion übernommen wird. Ebenso muss berücksichtigt werden, dass registrierte Smart Contracts die neue Adresse der Updateversion mitgeteilt bekommen.
A6	Permanente Interoperabilität registrierter Smart Contracts	Der Registry Contract stellt eine Funktion zur Verfügung, die das Abrufen der aktuellen Adresse eines Smart Contracts ermöglicht. Dadurch wird sichergestellt, dass Smart Contracts immer mit der aktuellen Version eines anderen Smart Contracts interagieren.

7.2 Funktionenbeschreibung

Das System besteht aus drei Komponenten, welche unterschiedliche Aufgaben in dem Entwurfsmuster haben:

1. User
2. Registrierte Smart Contracts
3. Registry Contract

Die essentielle Komponente stellt der Registry Contract dar, welcher für die Erfüllung der Anforderungen verantwortlich ist. Dies wird durch die Einführung entsprechender Datentypen und Funktionen bewerkstelligt. Abbildung 7.1 zeigt ein Klassendiagramm des Systems.

User

Der User ist immer der Ursprung einer Änderung im System. Im Normalfall wird es sich dabei um einen EOA, also einen Menschen, handeln. Es kann sich jedoch auch um einen Smart Contract handeln, welche in Ethereum die gleichen Möglichkeiten haben wie EOAs. User programmieren und registrieren Smart Contracts und deren Updateversionen. Sie können mit allen Smart Contracts im System interagieren, wobei Grenzen wie das Gas Limit bestehen. Bei der Programmierung von Smart Contracts müssen gewisse Strukturen implementiert werden, damit diese im Registry Contract registriert werden können und auf andere Smart Contracts zugegriffen werden kann. Darauf wird genauer im folgenden Abschnitt eingegangen.

Registrierte Smart Contracts

Der obere Teil zeigt die im Registry Contract registrierten Smart Contracts. Jeder Smart Contract, der im Registry Contract angemeldet wird, muss zumindest die Funktion 'setnewRCAddress' mit einer Adresse als Input Parameter beinhalten. Diese Grundstruktur ist Voraussetzung zur Registrierung. Die blauen Inhalte müssen nur Smart Contracts aufweisen, welche auf mindestens einen anderen Smart

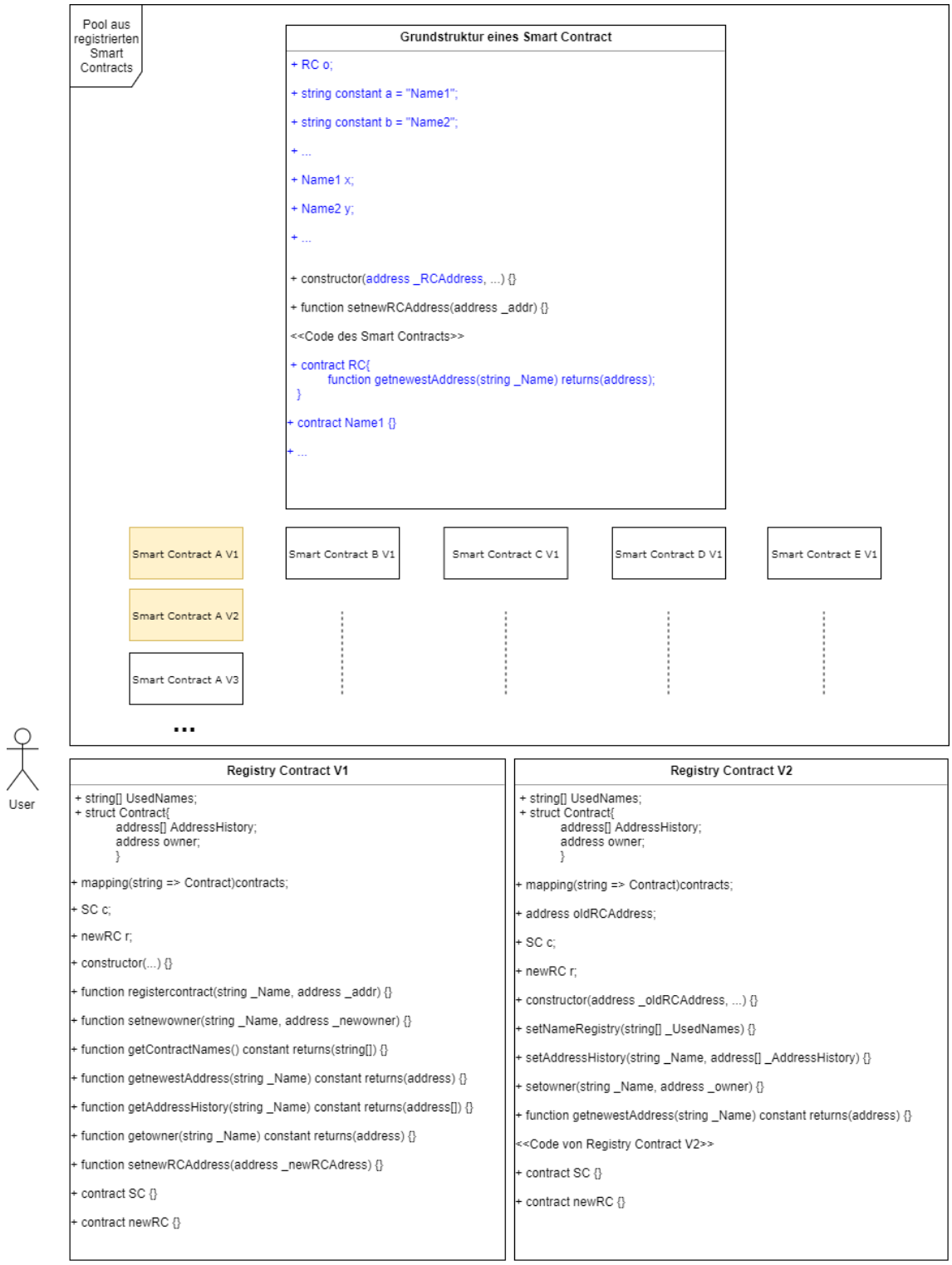

Abbildung 7.1: Klassendiagramm des Entwurfsmusters

Contract im System zugreifen (über 'call' oder 'delegatecall'). Ein Smart Contract, der nicht auf andere zugreift, muss die Adressen anderer Smart Contracts nicht abfragen. Demzufolge muss ein unabhängiger Smart Contract auch nicht die Adresse des Registry Contracts speichern und aktualisieren können, da er nie auf das Register zugreifen muss.

Im unteren Teil der Grundstruktur eines Smart Contracts sind in blau die Interfaces 'contract RC' und 'contract Name1'. Dort müssen Funktionssignaturen von aufzurufenden Smart Contracts definiert werden. Von diesen Interfaces können Instanzen wie ganz oben 'RC o;' und weiter unten 'Name1 x;' sowie 'Name2 y;' initiiert werden. Diesen Instanzen können Adressen zugeordnet werden, sodass mit diesen Instanzen Funktionen anderer Smart Contracts aufgerufen werden können, welche in den Funktionssignaturen definiert wurden.

Laut Anforderung A1 werden Adressen registrierter Smart Contracts im Registry Contract über Namen abgerufen. Daher sollten in der Grundstruktur strings als Konstanten mit den Namen von Smart Contracts deklariert werden, unter denen diese Smart Contracts im Registry Contract angemeldet wurden.

Im *constructor* ist die Adresse des Registry Contracts ein Input Parameter, wobei noch andere Input Parameter festgelegt werden können, die nichts mit dem Registry Contract zu tun haben.

Erfolgt ein Aufruf eines anderen Smart Contracts, so wird zuerst die aktuelle Adresse dieses Smart Contracts nach dem Pull-Prinzip (dazu mehr im nächsten Abschnitt) aus dem Registry Contract abgerufen, danach erfolgt der Aufruf des anderen Smart Contracts. Es erfolgen also insgesamt zwei externe Aufrufe auf andere Smart Contracts.

Ältere Versionen von registrierten Smart Contracts werden im System nicht mehr genutzt. Diese sind gelb dargestellt und könnten über den Befehl 'selfdestruct' deaktiviert werden, ohne dass das System zusammenbrechen würde.

Registry Contract

Der Registry Contract fungiert in diesem System nach dem Entwurfsmuster 'Beobachter'. Die registrierten Smart Contracts entsprechen beobachtbaren Objekten. Bei einer Änderung eines Smart Contracts, also der Registrierung einer Updateversion, erhalten von diesem Smart Contract abhängige Smart Contracts

eine Information dieser Änderung in Form der aktualisierten Adresse. Es gibt zwei Möglichkeiten, diese Änderungsinformation mitzuteilen:

Eine Möglichkeit wäre die Anwendung des Push-Prinzips. Im Falle einer Änderung benachrichtigt der Beobachter aktiv alle Smart Contracts, die von dem geänderten Smart Contract abhängig sind.

Die andere Option ist das Pull-Prinzip. Dabei müssen die Smart Contracts eigenständig die Änderungsinformation beim Beobachter abrufen und verarbeiten (Goll, 2014, S. 163-173).

Dieses Entwurfsmuster wird nach dem Pull-Prinzip arbeiten. Bei einer Anwendung des Push-Prinzips würde mit zunehmender Menge an registrierten Smart Contracts der Aufwand dieses Prinzips ebenso zunehmen. Bei jeder Registrierung einer Updateversion eines Smart Contracts müsste der Registry Contract jeden Registereintrag der Smart Contracts durchgehen und überprüfen, ob diese von dem Update Smart Contract abhängig sind. Falls ja müsste ein 'call' erfolgen, der die Adressvariable in den Smart Contracts aktualisiert. Dies könnte bei einem großen System hohe Gas Kosten verursachen.

Um Anforderung A1 zu erfüllen, muss in dem Registry Contract ein Array aus Strings 'UsedNames' angelegt werden, welches als Namensregister der Smart Contracts fungiert. Ein Registereintrag eines Smart Contracts könnte über eine struct 'Contract' angelegt, welche ein Array aus Adressen 'AddressHistory' und eine Adresse 'owner' beinhaltet. Über ein mapping zwischen einem string und der struct 'Contract' ist es möglich, einen Registereintrag eines Smart Contracts anzulegen, welcher über einen Namen abrufbar ist. Außerdem müssen Instanzen von aufzurufenden Smart Contracts initiiert werden, was mit 'SC c;' und 'newRC r;' erfolgt. Diese Instanzen ermöglichen Aufrufe von Funktionen der Contracts, die unten im Interface des Registry Contracts definiert werden.

Der kommende Teil befasst sich mit den einzelnen Funktionen, die der Registry Contract aufweisen muss. Nach der Nennung des Funktionsnamens sind in runden Klammern die Input Parameter und gegebenenfalls danach noch über die Kennzeichnung 'returns' in runden Klammern der Output-Typ definiert. Jede Funktion wird zudem mit den Anforderungen versehen, zu deren Erfüllung sie beitragen.

function registercontract (string _Name, address _addr): A1, A2, A4

Über diese Funktion können Smart Contracts im Registry Contract angemeldet werden. Bei einer Registrierung von Version 1 müssen drei Variablen gesetzt werden: Die Adresse des Smart Contracts _addr, die Adresse des 'owner' und der string _Name. Handelt es sich um eine Updateversion muss eine Überprüfung erfolgen, ob der Aufrufer dieser Funktion der 'owner' des Smart Contracts ist. Danach kann die Updateadresse in die Adress-Historie gesetzt werden.

function setnewowner (string _Name, address _newowner): A4

Es sollte für 'owner' möglich sein, das Registrierungsrecht für Updateversionen an andere Adressen abzugeben, was in dieser Funktion möglich sein soll. _Name gibt dabei an, von welchem Smart Contract das Recht abgetreten werden soll und _newowner erhält dieses Recht.

function getContractNames () returns (string[]):

Mit dieser Funktion können User sich informieren, welche Namen für Smart Contracts schon in Benutzung sind und dementsprechend nicht für die Registrierung eigener Smart Contracts zur Verfügung stehen.

function getnewestAddress (string _Name) returns (address): A1, A3, A6

'getnewestAddress' ist die Funktion, die registrierte Smart Contracts aufrufen, um an die aktuellen Adressen anderer Smart Contracts zu kommen. Entsprechend des Input Parameters _Name wird die Adresse der zuletzt registrierten Version zurückgegeben.

function getAddressHistory (string _Name) returns (address[]):

Über diese Funktion können alle Adressen abgerufen werden, die je von einem Smart Contract _Name registriert wurden.

function getowner (string _Name) returns (address):

Mit dieser Funktion kann überprüft werden, wer momentan 'owner' eines Smart Contracts _Name ist und somit das Recht hat, Update Adressen zu registrieren.

function setnewRCAddress (address _newRCAddress): A3, A5, A6

Diese Funktion wird ausgeführt, wenn ein neuer Registry Contract eingeführt wird. Dazu muss das Register an den neuen Registry Contract transferiert werden. Außerdem muss allen aktuellen Versionen registrierter Smart Contracts die Adresse des neuen Registry Contracts mitgeteilt werden.

Damit der Registry Contract mit anderen Smart Contracts interagieren kann, müssen Interfaces der aufzurufenden Smart Contracts angelegt werden. Der Registry Contract muss sowohl registrierte Smart Contracts als auch seine Nachfolgerversion in der Funktion 'setnewRCAddress' aufrufen. Daher müssen für diese beiden Smart Contract Typen Interfaces angelegt werden, was mit 'contract SC ' für registrierte Smart Contracts sowie 'contract newRC ' für den Update Registry Contract gekennzeichnet ist.

Die Nachfolgeversion eines Registry Contracts (V2) braucht ein paar Zusätze, damit das System stabil bleibt. Der *constructor* eines Update Registry Contracts hat die Adresse des Vorgänger Registry Contracts als Input Parameter. Dort sollte eine Adressvariable gesetzt werden, damit nur diese Adresse die Funktionen 'setNameRegistry', 'setAddressHistory' und 'setowner ausführen kann, welche für den Registertransfer zuständig sind.

Außerdem muss weiterhin die Funktion 'getnewestAddress' mit genau dem gleichen Namen und dem gleichen Input Parameter vorhanden sein. Andernfalls könnten die registrierten Smart Contracts nicht mehr auf das Register zugreifen und die aktuellen Adressen abrufen. Unten im Registry Contract V2 sind die gleichen Interfaces wie bei Version 1 definiert, damit die Updatability des Registry Contracts weiterhin gegeben ist.

7.3 Zeitlicher Ablauf des Entwurfsmusters

Zur besseren Verständlichkeit und Nachvollziehbarkeit ist in Abbildung 7.3 ein Sequenzdiagramm dargestellt, welches nun Schritt für Schritt durchgegangen und erörtert wird.

6 Komponenten sind in dem Diagramm vertreten: Der User, der Registry Contract in erster Version (RC V1), ein zu registrierender Smart Contract 'Contract

X' in erster und zweiter Version, ein anderer zu registrierender Smart Contract 'Contract Y' und der Registry Contract in zweiter Version (RC V2).

Der 'Contract X' ist als „Kein Caller" deklariert. Damit ist gemeint, dass dieser Smart Contract keine Aufrufe anderer Smart Contracts durchführt und demzufolge nicht von anderen Smart Contracts abhängig ist. Demgegenüber ist der 'Contract Y' als „Caller" deklariert, womit gemeint ist, dass er mit anderen Smart Contracts im System interagiert und von ihnen abhängig ist.

Der erste Schritt ist das Deployment des Registry Contracts V1, also das Einfügen des Smart Contracts in die Blockchain. Dabei wird der *constructor* ausgeführt, in dem der Registry Contract V1 einen Registereintrag für sich selbst anlegt.

Die nächste Aktion ist das Deployment von Contract X V1. Contract X greift auf keine anderen Smart Contracts (SC) zu und braucht deshalb auch nicht die Adresse des Registry Contracts als Variable im *constructor* zu setzen. Es können jedoch beliebige Input Parameter genutzt werden, falls dies so programmiert wurde.

Nach dem Deployment wird der Contract X V1 von dem Ersteller bzw. demjenigen, der die Kontrolle über diesen Smart Contract im Registry Contract haben soll, über den Funktionsaufruf 'registercontract' mit den Input Parametern string _Name und address _addr registriert. _Name ist der Name, unter dem andere Smart Contracts auf dessen Adresse im Registry Contract zugreifen können. _addr ist die Adresse des zu registrierenden Contracts, in diesem Fall also die Adresse von Contract X V1.

Es wäre auch möglich, dass sich ein Smart Contract in seinem *constructor* über einen Aufruf der Funktion 'registercontract' im Registry Contract selber registriert. Das würde den Automatisierungsgrad des Systems erhöhen, da ein 'owner' nach dem Einsetzen eines Smart Contracts nicht noch die Registrierung des Smart Contracts eigenständig vornehmen müsste. Die Frage ist nur, wie dann die Adresse 'owner' festgelegt wird. Dies über einen zusätzlichen Input Parameter zu machen wäre möglich. Dann besteht jedoch das Problem, dass es im Falle eines Updates keine sichere Überprüfung gibt, ob wirklich der 'owner' den Update Smart Contract registrieren lassen will. Ein böswilliger User könnte als Input Parameter die Adresse eines 'owner' einsetzen und Updates registrieren, die gar nicht vom 'owner' stammen. Diese registrierten Adressen könnten zu nicht vorhandenen oder falschen Smart Contracts führen, sodass das System instabil werden würde. Es gibt

den Befehl 'tx.origin', welcher die Adresse des ursprünglichen EOA beim Aufruf einer Funktion repräsentiert. Dazu ein kurzes Beispiel:

Ein EOA ruft eine Funktion in Contract A auf, welche wiederum Contract B aufruft und dieser ruft Contract C auf - EOA $\rightarrow$ A $\rightarrow$ B $\rightarrow$ C. In dieser Aufrufkette ist in allen Contracts A, B und C 'tx.origin' die Adresse des EOA. Demgegenüber gibt der Befehl 'msg.sender' die Adresse des vorigen Aufrufers zurück, was sowohl ein Smart Contract als auch ein EOA sein kann. 'tx.origin' ist immer ein EOA (Ethereum community, 2018p). Es ist leider nicht sicher, den Befehl 'tx.origin' zur Authentifizierung zu benutzen (Ethereum community, 2018j). Falls ein bösartiger User es hinbekommt, dass ein 'owner' eines registrierten Smart Contracts Ether an einen bestimmten „Angreifer" Smart Contract schickt, könnte dieser Angreifer so programmiert sein, dass die *fallback function* ausgelöst wird und einen Update Smart Contract im Registry Contract anmeldet. Im Registry Contract würde 'tx.origin' der 'owner' sein und ein falscher Smart Contract wäre im System. Dieser Vorgang ist in Abbildung 7.2 dargestellt. Außerdem wird abgeraten diesen Befehl zu benutzen, da er womöglich abgeschafft wird (Stack Overflow, 2016b). Daher wird in diesem Entwurfsmuster der Befehl 'msg.sender' zur Authentifizierung verwendet, was zwar den Automatisierungsgrad verringert, aber gleichzeitig sicher ist.

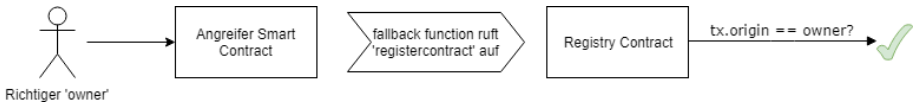

Abbildung 7.2: Authentifizierungsproblem von tx.origin

Über die Funktion 'setnewowner' kann der aktuelle 'owner' des Contracts _Name einen neuen owner über den Input Parameter address _newowner festlegen.

Die folgenden Funktionsaufrufe von 'getContractNames', 'getnewestAddress', 'getAddressHistory' sowie 'getowner', welche von einem User aufgerufen werden, dienen höchstens dazu, einen Überblick zu erhalten. Sie stellen read-only Aufrufe dar, welche keine Änderungen im Registry Contract hervorrufen. Ein User kann beispielsweise über 'getContractNames' nachschauen, welche Namen schon in Benutzung sind und nicht mehr von ihm zur Registrierung ausgewählt werden können.

Das Deployment von Contract X V2 ist zunächst völlig losgelöst vom Registry Contract. Wie bei dessen Vorgängerversion greift dieser Contract nicht auf andere Smart Contracts zu und muss dementsprechend nicht die Adresse des Registry Contracts als Input Parameter für den *constructor* haben. Sollte es irgendwann erforderlich sein, dass ein als „Kein Caller“ deklarierter Contract doch noch auf andere Smart Contracts zugreifen muss, kann dies problemlos realisiert werden. Wie in Abbildung 7.1 zu sehen muss nur die Grundstruktur entsprechend den blau markierten Inhalten angepasst werden.

Von dem aktuellen 'owner' des Contracts X V1 kann Contract X V2 nun über die Funktion 'registercontract' im Registry Contract registriert werden. Die Adresse von Contract X V2 wird dabei in der AddressHistory am Ende des Arrays eingefügt.

Die folgende Aktion ist das Deployment von Contract Y V1. Es handelt sich dabei um einen „Caller“, daher wird dort im *constructor* als Input Parameter die Adresse des Registry Contracts benötigt. Dadurch kann ausschließlich dem Registry Contract das Zugriffsrecht auf die Funktion 'setnewRCAddress' gegeben werden. Außerdem kann dadurch schon vor der Registrierung auf die Funktion 'getnewestAddress' des Registry Contracts zugegriffen werden und Funktionsaufrufe auf andere Smart Contracts durchgeführt werden.

Im Anschluss erfolgt die Registrierung des Contracts Y V1 im Registry Contract wie zuvor über einen _Name und dessen Adresse _addr von dem User, welcher Kontrolle über den Contract Y haben soll.

Nun wird über den Contract Y V1 ein Funktionsaufruf eines anderen Smart Contracts, beispielsweise Contract X V2, gestartet. Dabei wird als erstes die Funktion 'getnewestAddress' mit dem Input Parameter string _Name des registrierten Contracts, der aufgerufen werden soll, im Registry Contract aufgerufen. Diese gibt die aktuelle Adresse des Contracts _Name zurück, sodass Contract Y V1 auf die richtige Version zugreift. Damit der Aufruf erfolgreich ist müssen das Interface und die richtigen Input Parameter des aufzurufenden Contracts im „Caller“ Contract programmiert sein.

Danach erfolgt das Deployment eines Update Registry Contract V2. Dieser braucht neben möglichen anderen Input Parametern die Adresse des vorigen Registry Contracts. Damit wird im *constructor* eine Adressvariable gesetzt. Diese

Adresse erhält als einzige Zugriff auf die Funktionen 'setNameRegistry', 'setAddressHistory' sowie 'setowner', um einen Registertransfer durchzuführen.

Nach dem Deployment des Registry Contract V2 wird vom 'owner' des Registry Contracts V1 die Funktion 'setnewRCAddress' mit der Adresse des Registry Contracts V2 als Input Parameter aufgerufen. Dieser Funktionsaufruf sorgt zum einen für eine Aktualisierung der Registry Contract Adresse in jeder aktuellen registrierten Version von Smart Contracts, indem der Registry Contract V1 deren Funktion 'setnewRCAddress' aufruft. Zum anderen wird das Register über Funktionsaufrufe von 'setNameRegistry', 'setAddressHistory' sowie 'setowner' im Registry Contract V2 transferiert. Abschließend deaktiviert sich der Registry Contract V1 über den Befehl 'selfdestruct'.

Das Sequenzdiagramm stellt nur ein Beispiel dar, um die Abläufe in dem Entwurfsmuster zu verdeutlichen. Eigentlich sollte das System im Idealfall aus einem großen Pool aus Smart Contracts bestehen, in dem viel Dynamik aus Funktionsaufrufen anderer Smart Contracts, Smart Contract Registrierung und Updates herrscht.

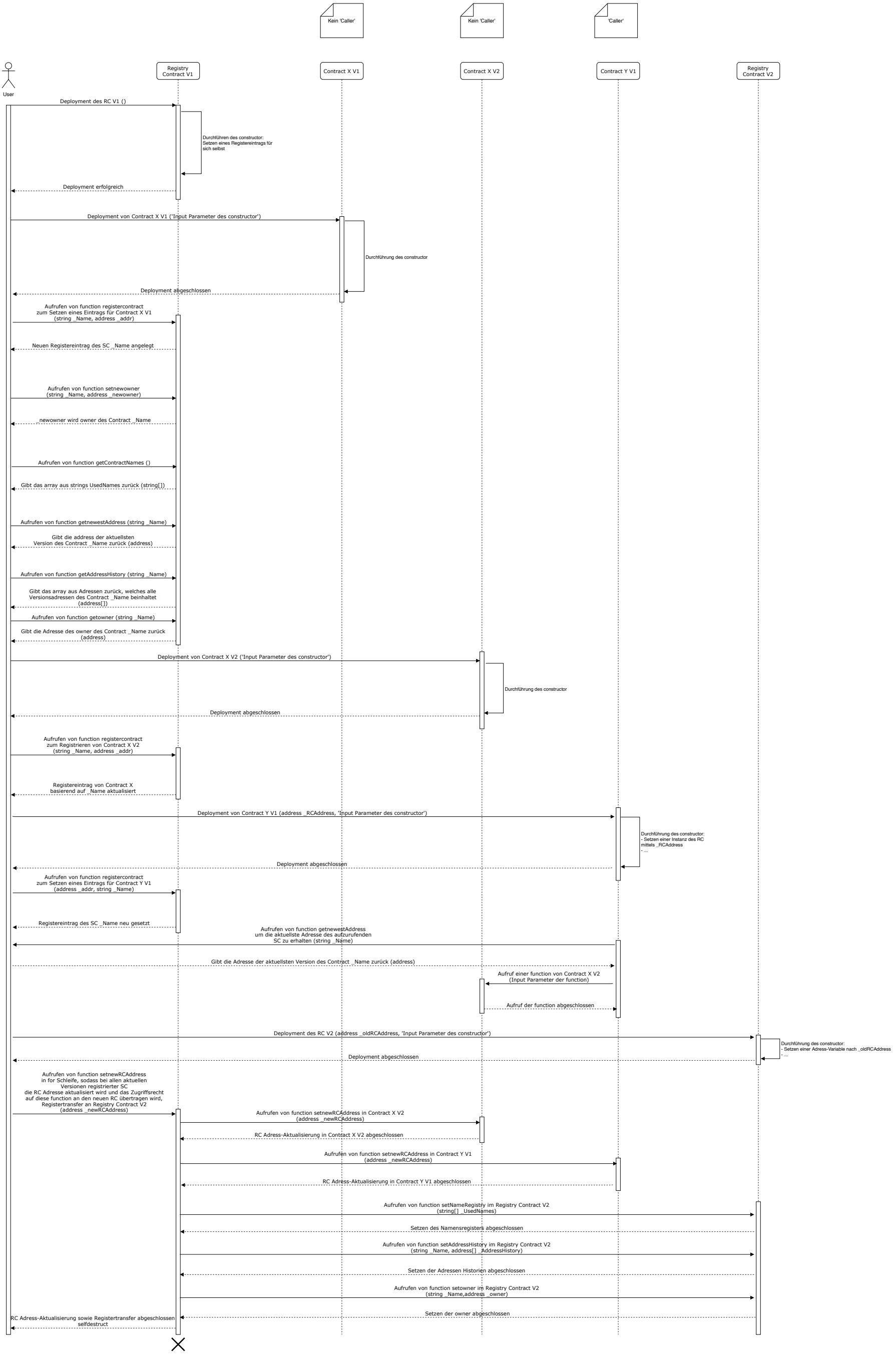

Abbildung 7.3: Sequenzdiagramm des Entwurfsmusters

7.4 Umsetzung des Entwurfsmusters

Ein grundsätzliches Problem bei der Umsetzung besteht darin, dass aktuell keine 2-dimensionalen Arrays unterstützt werden. Da ein string in Solidity schon als Array angesehen wird, ist es nicht möglich ein Array aus Strings anzulegen, wie es für das Namensregister (string[] UsedNames) vorgesehen war (Ethereum community, 2018g; Stack Overflow, 2017b). Es gibt jedoch einen Zusatz, der im Code eingefügt werden kann, sodass es möglich wird, Arrays aus strings zu erstellen. Die Zeile 'pragma experimental ABIEncoderV2;' löst dieses Problem. Die darin enthaltenen Features sind aber noch nicht auf der Blockchain vorhanden, weshalb die reale Umsetzung in Ethereum noch nicht möglich ist. Bei Verzicht der Einbettung eines Arrays aus strings 'UsedNames' funktioniert der Registry Contract dennoch. Dadurch würde nur die Übersichtlichkeit etwas verringert, da User nicht direkt überprüfen können, welche Namen schon in Benutzung sind. Das Abrufen von Adressen über Namen ist auch ohne 'UsedNames' über das mapping möglich, wie in einem Code eines Registry Contract Fragments getestet wurde.

Die Umsetzung erfolgte entsprechend dem in Abschnitt 7.2 vorgestellten Klassendiagramm und den Funktionenbeschreibungen.

Um Anforderung A1 zu erfüllen, wird in dem Registry Contract ein Array aus Strings 'UsedNames' angelegt. Ein Registereintrag eines Smart Contracts wird über eine struct 'Contract' angelegt, welche ein Array aus Adressen 'AddressHistory' und eine Adresse 'owner' beinhaltet. Über ein mapping zwischen einem string und der struct 'Contract' ist es nun möglich, einen Registereintrag eines Smart Contracts anzulegen, welcher über einen Namen abrufbar ist.

Im Registry Contract muss ein Interface der registrierten Smart Contracts angelegt werden, da dieser deren Funktion 'setnewRCAddress' aufrufen können muss. Die Instanz dazu ist als 'SC c;' gekennzeichnet. Des Weiteren muss ein Interface und eine Instanz 'newRC r;' für einen potentiellen Update Registry Contract festgelegt werden, damit ein Registertransfer erfolgen kann. Dazu werden die Funktionen 'setNameRegistry', 'setAddressHistory' sowie 'setowner' als Funktionssignaturen definiert.

Im *constructor* des Registry Contracts legt dieser einen Registereintrag für sich selbst an, was im Klassendiagramm hartkodiert über den Namen „RC“ passiert.

Alternativ könnte dies auch über einen Input Parameter vom Typ string erfolgen.

Der kommende Teil befasst sich mit den einzelnen Funktionen, die der umgesetzte Registry Contract aufweist. Nach der Nennung des Funktionsnamens sind in runden Klammern die Input Parameter und gegebenenfalls danach noch über die Kennzeichnung 'returns' in runden Klammern der Output-Typ. Anschließend folgt eine Erläuterung, was konkret innerhalb dieser Funktion passiert.

function registercontract (string _Name, address _addr):

Diese Funktion dient sowohl der erstmaligen Registrierung als auch der Registrierung von Updateversionen. Zunächst wird über die Adresse _addr getestet, ob der zu registrierende Smart Contract die Funktion 'setnewRCAddress' mit einer Adresse als Input Parameter beinhaltet. Dies ist notwendig, da sonst bei einem Update des Registry Contracts eine Schleife abbrechen würde, die den registrierten Smart Contracts die neue Registry Contract Adresse mitteilt (siehe setnewRCAddress). Bei der Überprüfung, ob die Funktion 'setnewRCAddress' vorhanden ist, hat der registrierende EOA keine Vorgaben, was innerhalb dieser Funktion passiert. Dadurch, dass bei einer Transaktion oder einer Aufrufkette der ursprünglich initierende EOA für das Bezahlen der Gas Kosten belangt wird, besteht kein Risiko dieses Überprüfungsmechanismus. Sollte diese Funktion hohe Gas Kosten verursachen, muss der registrierende EOA diese bezahlen (Ethereum community, 2018h; Stack Overflow, 2016c). Danach wird über den Input Parameter _Name geprüft, ob dieser Name schon in dem Array 'UsedNames' vorhanden ist. Ist dies nicht der Fall, liegt eine Registrierung von Version 1 eines Smart Contracts vor. Der _Name wird in das Array 'UsedNames' eingetragen, die Adresse _addr wird als erster Eintrag in die 'AddressHistory' angelegt und der 'owner' wird über den Befehl 'msg.sender' festgehalten. Wenn der _Name schon in 'UsedNames' vorhanden ist, soll eine Updateversion registriert werden. Es wird überprüft, ob der 'msg.sender' der 'owner' dieses Smart Contracts ist und anschließend die Adresse _addr in die 'AddressHistory' eingetragen.

Bei der Registrierung einer Updateversion eines Smart Contracts sollten die 'owner' berücksichtigen, dass gegebenenfalls ein Transfer der Daten des alten Smart Contracts durchgeführt werden muss.

function setnewowner (string _Name, address _newowner):

Anforderung A4 erfordert die Übertragung des Rechts zur Registrierung von Update-Adressen. Diese Funktion überprüft, ob der Aufrufer dieser Funktion der 'owner' des Smart Contracts _Name ist und gibt dann seine Updaterechte an die Adresse _newowner ab.

function getContractNames () returns (string[]):

'getContractNames' gibt das Array 'UsedNames' zurück. Diese Funktion ermöglicht es Usern nachzugucken, welche Namen schon im Register in Benutzung sind.

function getnewestAddress (string _Name) returns (address):

Durch diese Funktion wird es anderen Smart Contracts ermöglicht, das Pull-Prinzip anzuwenden. Der _Name unter dem ein Smart Contract abgespeichert ist ermöglicht den Zugang zu dessen Adresse. Bevor ein 'call' oder 'delegatecall' auf einen anderen Smart Contract durchgeführt wird, holt sich der aufrufende Contract die aktuelle Adresse des aufzurufenden Smart Contracts aus dem Registry Contract mittels dieser Funktion. Die aktuelle Adresse ist der letzte Eintrag im Array 'AddressHistory'.

function getAddressHistory (string _Name) returns (address[]):

Der Aufruf dieser Funktion verschafft einen Überblick über die Adressen aller Versionen, wodurch alte Versionen eines Smart Contracts potentiell zugreifbar bleiben.

Alten Versionen wird jedoch bei einem Update des Registry Contracts nicht die neue Adresse des Registry Contracts mitgeteilt. Dadurch werden Funktionsaufrufe an externe Contracts nicht mehr funktionieren, da das Pull-Prinzip bei einem Registry Contract angewendet werden würde, welcher den Befehl 'selfdestruct' ausgeführt hat.

Es wäre möglich allen je registrierten Smart Contracts die Adresse im Falle eines Updates mitzuteilen. Dies würde aber die vermutlich ohnehin schon hohen Gas Kosten noch deutlich erhöhen. Außerdem wäre es ungünstig, wenn eine alte Version eines registrierten Contracts den Befehl 'selfdestruct' ausgeführt hat. Dies würde einen Funktionsabbruch von 'setnewRCAddress' zur Folge haben, wodurch

keinem Contract eine neue Registry Contract Adresse mitgeteilt werden könnte und somit kein Update des Registry Contracts zustande käme.

function getowner (string _Name) returns (address):

Mit dieser Funktion kann überprüft werden, wer momentan 'owner' eines Smart Contracts _Name ist und somit das Recht hat, Update Adressen zu registrieren.

function setnewRCAddress (address _newRCAddress):

Nachdem ein Update Registry Contract eingesetzt wurde, muss das Register transferiert werden. Den Registertransfer an dieser Stelle durchzuführen stellt sicher, dass alle registrierten Smart Contracts die neue Adresse mitgeteilt bekommen, da nach Beendigung dieser Funktion kein Smart Contract mehr im alten Registry Contract angemeldet werden kann. Zunächst wird die Funktion 'setNameRegistry' im neuen Registry Contract aufgerufen, welcher das Array aus Strings der registrierten Contractnamen übergibt. In einer for-Schleife, welche alle registrierten Namen durchgeht, werden nun zwei Schritte erledigt:

Zum einen werden dem neuen Registry Contract die mit den Namen verknüpften Adress-Historien sowie deren owner über die Funktionen 'setAddressHistory' und 'setowner' übergeben.

Zum anderen wird den aktuellen Versionen aller registrierten Smart Contracts die neue Adresse des Registry Contracts mitgeteilt werden, damit diese weiterhin die Funktion 'getnewestAddress' von dem neuen Registry Contract nutzen können. Am Ende dieser Funktion erfolgt ein 'selfdestruct' Befehl, sodass der alte Registry Contract deaktiviert wird.

```
1  pragma solidity ^0.4.25;
2  pragma experimental ABIEncoderV2;   //enables dynamic sized arrays as return type,
       or structs as return type
3
4  contract RC_V1 {
5
6      string[] UsedNames;          // array of strings for the names of the
           registered contracts
7
8      struct Contract {               // creating a struct for registered contracts,
           which contains their address history and their owner
9          address[] AddressHistory;
```

```
10          address owner;
11      }
12
13      newRC r;            // initiate instance of new RC
14
15      SC c;               // initiate instance of contract SC
16
17      mapping(string => Contract) contracts;      // linking the struct with string,
            so that the info of the struct can be accessed or changed via a string
            name
18
19      constructor() {                 // setting an entry of its own (RC_V1)
20          UsedNames.push("RC");
21          contracts["RC"].owner = msg.sender;
22          contracts["RC"].AddressHistory.push(address(this));
23      }
24
25      function registercontract(string _Name, address _addr) {  // function for
            registering a new contract or an update of an already registered contract
26
27
28          c = SC(_addr);                                      // It has to be
                checked, whether the registered Contract contains the function
                setnewRCAddress. If not it won't be registered. If this function does
                not exist in the contract, then the system could crash if an update RC
                 would be deployed and the function setnewRCAddress is invoked.
29          c.setnewRCAddress(this);
30
31
32
33
34
35          for(uint i = 0; i < UsedNames.length; i++) {                    // Is
                _Name already used?
36
37
38              if(keccak256(UsedNames[i]) == keccak256(_Name)) {       // if yes,
                    then this is an update (keccak256 needs less gas than character
                    comparison)
39
40                  if(msg.sender == contracts[_Name].owner) {          // if the
                        sender is the owner of the contract, push new update address
                        in AddressHistory
41                      contracts[_Name].AddressHistory.push(_addr);
42                  }
43
44                  else {
45
46                      revert('You are not the owner!');
47
```

```
48                 }
49
50                 break;
51             }
52
53         }
54
55         if(i == UsedNames.length) {                              // if _Name is not
               used
56
57             UsedNames.push(_Name);                               // create a new
                   entry for a contract
58             contracts[_Name].owner = msg.sender;
59             contracts[_Name].AddressHistory.push(_addr);
60
61         }
62
63     }
64
65     function setnewowner(string _Name, address _newowner) {
66
67         if(msg.sender == contracts[_Name].owner) {               // if the sender
               is the owner of the contract, set owner = _newowner
68             contracts[_Name].owner = _newowner;
69         }
70
71         else {
72
73             revert('You are not the owner!');
74
75         }
76
77     }
78
79     function getContractNames() constant returns(string[]) {    // returns the
           array of strings with all used names for the contracts
80
81         return UsedNames;
82
83     }
84
85     function getnewestAddress(string _Name) constant returns(address) {
                           // returns the latest address of the contract _Name
86
87         return contracts[_Name].AddressHistory[contracts[_Name].AddressHistory.
               length-1];
88
89     }
90
91     function getAddressHistory(string _Name) constant returns(address[]) {  //
```

```
        returns all addresses in an array of the contract _Name
92
93          return contracts[_Name].AddressHistory;
94
95      }
96
97      function getowner(string _Name) constant returns(address) {     // returns the
            owner of the contract _Name
98
99          return contracts[_Name].owner;
100
101     }
102
103     function setnewRCAddress(address _newRCAddress) {     // call every newest
            version of each contract which ever registered and update the RC address
            in these contracts, transfer the Register to the new RC
104
105         if(msg.sender == contracts["RC"].owner) {
106
107             r = newRC(_newRCAddress);                        // set address of new
                    RC
108
109             r.setNameRegistry(UsedNames);                // transfer UsedNames to
                    new RC
110
111             contracts["RC"].AddressHistory.push(_newRCAddress);     // push
                    address of new RC into its registry
112
113             r.setAddressHistory(UsedNames[0], contracts[UsedNames[0]].
                    AddressHistory); // transfer RC AddressHistory
114
115             r.setowner(UsedNames[0], contracts[UsedNames[0]].owner);
                                    // transfer RC owner
116
117             for(uint i = 1; i < UsedNames.length; i++) {    // start with second
                    name entry at index 1, because the first is the RC (function would
                     call itself and fail)
118
119                 c = SC(getnewestAddress(UsedNames[i]));
120
121                 c.setnewRCAddress(_newRCAddress);         // update RC Address in
                        current version of registered Contract
122
123                 r.setAddressHistory(UsedNames[i], contracts[UsedNames[i]].
                        AddressHistory);  // transfer AddressHistory to new RC
124
125                 r.setowner(UsedNames[i], contracts[UsedNames[i]].owner);
                                        // transfer owner to new RC
126             }
127
```

```
128             selfdestruct(contracts["RC"].owner);            // after updating all
                    RC addresses in the registered contracts, selfdestruct and send
                    all remaining ether to the owner of the RC
129
130         }
131
132         else {
133
134             revert('You are not the owner of the RC!');
135
136         }
137
138     }
139
140
141
142 }
143
144 contract SC {                                   // interface of registered
        contracts, so that the RC address can be updated in the registered contracts
145
146     function setnewRCAddress(address _addr);
147
148 }
149
150 contract newRC {
151     function setNameRegistry(string[] _UsedNames);          // interface of the
            new RC, so that this RC can transfer its register in the function
            setnewRCAddress to the new RC
152     function setAddressHistory(string _Name, address[] _AddressHistory);
153     function setowner(string _Name, address _owner);
154 }
```

Code 7.1: Solidity Code des Registry Contract Version 1

Zum finalen Test diente ein Programm bestehend aus 3 Contracts, welche 'C', 'D' und 'E' heißen (Stack Overflow, 2016a). Die Contracts wurden dahingehend angepasst, dass sie im Registry Contract System genutzt werden können, beispielsweise durch das Einfügen der Funktion 'setnewRCAddress'.

Es müssen keine Interfaces für Funktionsaufrufe untereinander implementiert werden, weil alle 3 Contracts in einer Datei sind. Nur das Interface des Registry Contracts 'Contract RC' ist notwendig, damit die Contracts nach dem Pull-Prinzip die aktuellen Adressen aus dem Registry Contract abfragen können. Vor jedem externen Aufruf einer Funktion holt sich der Aufrufer die aktuelle Adresse des aufzurufenden Contracts aus dem Registry Contract mittels 'getnewestAddress'.

Contract 'E' initiiert die Variablen uint 'n' und address 'sender'. Die einzige Funktion ist 'setN', welche die Variable 'n' entsprechend dem Input Parameter _n und die Variable 'sender' entsprechend 'msg.sender' setzt. Dieser Contract greift auf keine anderen Contracts zu und muss dementsprechend auch nicht die Adresse des Registry Contracts kennen. Es wurde noch eine leere Funktionshülse 'setnewRCAddress' eingefügt, damit eine Registrierung erfolgen kann.

Contract 'D' initiiert die gleichen Variablen wie 'E'. Contract 'D' greift auf Contract 'E' zu und muss deswegen Instanzen für den Registry Contract 'RC o;' und für Contract 'E' 'E e;' setzen. Im *constructor* wird die Instanz des Registry Contracts über den Input Parameter _RCAddress gesetzt.

Die drei Funktionen 'callSetN', 'callcodeSetN' und 'delegatecallSetN' rufen alle die Funktion 'setN' von Contract 'E' auf, jedoch über unterschiedliche Methoden. Je nachdem welche Methode angewandt wird, erfolgt entweder die Nutzung des Speichers von Contract 'D' ('callSetN') oder von Contract 'E' ('callcodeSetN' und 'delegatecallSetN').

Die Funktion 'setnewRCAddress' kann nur der Registry Contract aufrufen, welcher im Falle eines Updates des Registry Contracts dessen neue Adresse übergibt.

Contract 'C' greift auf Contract 'D' zu, deswegen müssen für den Registry Contract 'RC o;' und Contract 'D' 'D d;' Instanzen gesetzt werden. Im *constructor* wird die Instanz des Registry Contracts über den Input Parameter _RCAddress gesetzt.

Die Funktion 'foo' führt einen 'call' auf die Funktion 'delegatecallSetN' von Contract 'D' aus. Dadurch wird eine Aufrufkette ausgelöst, welche letztendlich die Variable 'n' von Contract 'D' über eine Funktion von Contract 'E' entsprechend dem ursprünglichen Input Parameter _n von Contract 'C' setzt.

Die 3 Contracts arbeiten nur mit dem Typ integer, doch bei entsprechender Programmierung wird eine solche Interoperabilität mit allen anderen Datentypen ebenso möglich sein. Außerdem ist die Handhabung 2-dimensionaler Arrays nur dank des Zusatzes 'pragma experimental ABIEncoderV2;' im Code möglich.

```
pragma solidity ^0.4.18;
contract D {
  uint public n;
  address public sender;
  RC o;
  E e;

    constructor(address _RCAddress) {  // setting the RC Address

        o = RC(_RCAddress);

    }

  function callSetN(uint _n) public {
      e = E(o.getnewestAddress("E"));
    e.call(bytes4(keccak256("setN(uint256)")), _n); // E's storage is set, D is
        not modified
  }

  function callcodeSetN(uint _n) public {
      e = E(o.getnewestAddress("E"));
    e.callcode(bytes4(keccak256("setN(uint256)")), _n); // D's storage is set, E
        is not modified
  }

  function delegatecallSetN(uint _n) public {
      e = E(o.getnewestAddress("E"));
    e.delegatecall(bytes4(keccak256("setN(uint256)")), _n); // D's storage is set,
         E is not modified
  }

    function setnewRCAddress(address _newRCAddress) {    // update the RC Address,
         but only if msg.sender is the RC

        if(msg.sender == address(o)) {

            o = RC(_newRCAddress);

        }

        else{

            revert('No access allowed! msg.sender not RC');

        }

    }

}

```

```
47 contract E {
48   uint public n;
49   address public sender;
50
51   function setN(uint _n) public {
52     n = _n;
53     sender = msg.sender;
54     // msg.sender is D if invoked by D's callcodeSetN. None of E's storage is
       updated
55     // msg.sender is C if invoked by C.foo(). None of E's storage is updated
56
57     // the value of "this" is D, when invoked by either D's callcodeSetN or C.foo
       ()
58   }
59   function setnewRCAddress(address _newRCAddress) {}
60
61 }
62
63 contract C {
64
65     RC o;
66     D d;
67
68     constructor(address _RCAddress) {  // setting the RC Address
69
70         o = RC(_RCAddress);
71
72     }
73
74     function foo(uint _n) public {
75         d = D(o.getnewestAddress("D"));
76         d.delegatecallSetN(_n);
77     }
78
79     function setnewRCAddress(address _newRCAddress) {     // update the RC Address
       , but only if msg.sender is the RC
80
81         if(msg.sender == address(o)) {
82
83             o = RC(_newRCAddress);
84
85         }
86
87         else{
88
89             revert('No access allowed! msg.sender not RC');
90
91         }
92
93     }
```

```
94  }
95
96  contract RC{                        // interface of the RC, so that contracts can
        pull the newest address of the the contract they want to interact with
97
98      function getnewestAddress(string _Name) returns(address);
99
100 }
```

Code 7.2: Solidity Code der finalen Test Smart Contracts

Ein Gerüst des Registry Contract als Updateversion wurde ebenfalls nach dem in Abschnitt 7.2 vorgestellten Klassendiagramm und den Funktionenbeschreibungen erstellt. Er enthält die gleichen Funktionen wie die Vorgängerversion, nur dass noch die Funktionen 'setNameRegistry', 'setAddressHistory' und 'setowner' für den Registertransfer enthalten sind. Vorher wird im *constructor* mit dem Input Parameter _oldRCAddress das Zugriffsrecht auf die set-Funktionen an diese Adresse erteilt. Der Aufruf der Funktion 'setnewRCAddress' im alten Registry Contract sorgt für eine erfolgreiche Übertragung des gesamten Registers und anschließender Interaktion registrierter Smart Contracts mit dem neuen Registry Contract. Dieses Gerüst an sich hat gegenüber der ersten Version keinerlei Vorteile. Es müssten noch neue Features eingebaut werden, damit diese Version eine Verbesserung im Vergleich zur vorigen Version darstellt. Bei einem großen Register kann davon ausgegangen werden, dass solch ein Registertransfer viel Gas kostet. Daher sollte vor der Einführung eines Update Registry Contracts sorgfältig überlegt werden, ob das Update notwendig ist.

```
1   pragma solidity ^0.4.25;
2   pragma experimental ABIEncoderV2;   //enables dynamic sized arrays as return type,
        or structs as return type
3
4   contract RC_V2 {
5
6       string[] UsedNames;         // array of strings for the names of the
            registered contracts
7
8       struct Contract {               // creating a struct for registered contracts,
            which contains their address history and their owner
9           address[] AddressHistory;
10          address owner;
11      }
12
```

```
13      SC c;                   // initiate instance of contract SC
14
15      newRC r;            // initiate instance of new RC
16
17      address oldRCAddress;           // address variable for oldRC
18
19      mapping(string => Contract) contracts;      // linking the struct with string,
            so that the info of the struct can be accessed or changed via a string
            name
20
21      constructor(address _oldRCAddress) {        // set address variable of old RC,
            so that only this address can access the functions for the transfer of
            the register
22
23          oldRCAddress = _oldRCAddress;
24
25      }
26
27      function setNameRegistry(string[] _UsedNames) {     // set the Name registry
28
29          if(msg.sender == oldRCAddress) {
30
31              UsedNames = _UsedNames;
32          }
33
34          else {
35
36              revert('You are not the owner of the RC!');
37
38          }
39      }
40
41      function setAddressHistory(string _Name, address[] _AddressHistory) {   // set
            AddressHistory Registry from old RC
42
43          if(msg.sender == oldRCAddress) {
44
45              contracts[_Name].AddressHistory = _AddressHistory;
46
47          }
48
49          else {
50
51              revert('You are not the owner of the RC!');
52
53          }
54
55      }
56
57      function setowner(string _Name, address _owner) {          // set owner of
```

```
        each registered Contract
58
59          if(msg.sender == oldRCAddress) {
60
61          contracts[_Name].owner = _owner;
62
63          }
64
65          else {
66
67              revert('You are not the owner of the RC!');
68
69          }
70
71      }
72
73      function registercontract(string _Name, address _addr) {    // function for
            registering a new contract or an update of an already registered contract
74
75
76          c = SC(_addr);                                  // It has to be checked,
              whether the registered Contract contains the function setnewRCAddress.
               If not it won't be registered. If this function does not exist in the
               contract, then the system could crash if an update RC would be
              deployed and the function setnewRCAddress is invoked.
77          c.setnewRCAddress(this);
78
79
80
81          for(uint i = 0; i < UsedNames.length; i++) {            // Is _Name
              already used?
82
83              if(keccak256(UsedNames[i]) == keccak256(_Name)) {       // if yes,
                  then this is an update (keccak256 needs less gas than character
                  comparison)
84
85                  if(msg.sender == contracts[_Name].owner) {          // if the
                      sender is the owner of the contract, push new update address
                      in AddressHistory
86                      contracts[_Name].AddressHistory.push(_addr);
87                  }
88
89                  else {
90
91                      revert('You are not the owner!');
92
93                  }
94
95                  break;
96              }
```

```
97
98          }
99
100         if(i == UsedNames.length) {                         // if _Name is not
                used
101
102             UsedNames.push(_Name);                          // create a new
                    entry for a contract
103             contracts[_Name].owner = msg.sender;
104             contracts[_Name].AddressHistory.push(_addr);
105
106         }
107
108     }
109
110     function setnewowner(string _Name, address _newowner) {
111
112         if(msg.sender == contracts[_Name].owner) {          // if the sender
                is the owner of the contract, set owner = _newowner
113             contracts[_Name].owner = _newowner;
114         }
115
116         else {
117
118             revert('You are not the owner!');
119
120         }
121
122     }
123
124     function getContractNames() constant returns(string[]) {    // returns the
            array of strings with all used names for the contracts
125
126         return UsedNames;
127
128     }
129
130     function getnewestAddress(string _Name) constant returns(address) {
                        // returns the latest address of the contract _Name
131
132         return contracts[_Name].AddressHistory[contracts[_Name].AddressHistory.
                length-1];
133
134     }
135
136     function getAddressHistory(string _Name) constant returns(address[]) {  //
            returns all addresses in an array of the contract _Name
137
138         return contracts[_Name].AddressHistory;
139
```

```
140     }
141
142     function getowner(string _Name) constant returns(address) {     // returns the
            owner of the contract _Name
143
144         return contracts[_Name].owner;
145
146     }
147
148     function setnewRCAddress(address _newRCAddress) {     // call every newest
            version of each contract which ever registered and update the RC address
            in these contracts, transfer the Register to the new RC
149
150         if(msg.sender == contracts["RC"].owner) {
151
152             r = newRC(_newRCAddress);                        // set address of new
                    RC
153
154             r.setNameRegistry(UsedNames);                // transfer UsedNames to
                    new RC
155
156             contracts["RC"].AddressHistory.push(_newRCAddress);     // push
                    address of new RC into its registry
157
158             r.setAddressHistory(UsedNames[0], contracts[UsedNames[0]].
                    AddressHistory); // transfer RC AddressHistory
159
160             r.setowner(UsedNames[0], contracts[UsedNames[0]].owner);
                                    // transfer RC owner
161
162             for(uint i = 1; i < UsedNames.length; i++) {    // start with second
                    name entry at index 1, because the first is the RC (function would
                     call itself and fail)
163
164                 c = SC(getnewestAddress(UsedNames[i]));
165
166                 c.setnewRCAddress(_newRCAddress);         // update RC Address in
                        current version of registered Contract
167
168                 r.setAddressHistory(UsedNames[i], contracts[UsedNames[i]].
                        AddressHistory);  // transfer AddressHistory to new RC
169
170                 r.setowner(UsedNames[i], contracts[UsedNames[i]].owner);
                                        // transfer owner to new RC
171             }
172
173             selfdestruct(contracts["RC"].owner);            // after updating all
                    RC addresses in the registered contracts, selfdestruct and send
                    all remaining ether to the owner of the RC
174
```

```
175         }
176 
177         else {
178 
179             revert('You are not the owner of the RC!');
180 
181         }
182 
183     }
184 
185 }
186 
187 contract newRC {
188     function setNameRegistry(string[] _UsedNames);          // interface of the
        new RC, so that this RC can transfer its register in the function
        setnewRCAddress to the new RC
189     function setAddressHistory(string _Name, address[] _AddressHistory);
190     function setowner(string _Name, address _owner);
191 }
192 
193 contract SC {                                   // interface of registered
    contracts, so that the RC address can be updated in the registered contracts
194 
195     function setnewRCAddress(address _addr);
196 
197 }
```

Code 7.3: Solidity Code des Registry Contract Version 2

8 Analyse

Es wurde gezeigt, dass das Entwurfsmuster entsprechend den Anforderungen funktioniert. Es kann ein Netzwerk aus miteinander interagierenden, updatable Smart Contracts mithilfe des Registry Contracts realisiert werden. Dennoch gibt es noch viele Möglichkeiten zur Verbesserung. In diesem Kapitel erfolgt eine kritische Auseinandersetzung mit der Konzeption des erstellten Entwurfsmusters und möglichen Ansätzen zur Optimierung.

8.1 Namensregister

Das Namensregister 'UsedNames' kann noch nicht in der realen Blockchainumgebung umgesetzt werden, da das Namensregister als 2-dimensionales Array nur bei Tests in Remix verfügbar ist. Doch auch ohne das Namensregister wären Aufrufe über Namen dank eines mappings möglich, was mithilfe eines Testprogramms überprüft wurde (siehe Registry Contract ohne Namensregister). Ohne das Namensregister ist es umständlicher zu überprüfen, welche Namen schon in Benutzung sind. Dies müsste indirekt über einen Aufruf der Funktion 'getowner' erfolgen. Mit dem Input Parameter string _Name müsste 'getowner' aufgerufen werden und falls der Rückgabewert eine leere Adresse (eine 0 als Hexadezimalwert) ist, liegt der _Name noch nicht im Register vor.

8.2 Gas Kosten

Wie auch bei den in Abschnitt 6.1 vorgestellten Konzepten wird eine Updatability Funktionsaufrufe kostspieliger machen. Vor jedem Aufruf einer externen Funktion wird zunächst die aktuelle Adresse aus dem Registry Contract abgefragt, auch

wenn das eventuell nicht notwendig ist. Der aufrufende Contract könnte bereits die aktuelle Adresse aus einem früheren Aufruf haben. Zwischen dem ersten Aufruf und beispielsweise dem fünften Aufruf könnte kein Update des aufzurufenden Contracts registriert worden sein und die Adresse wurde schon beim ersten Aufruf abgefragt, was die restlichen vier Aufrufe überflüssig macht. So könnten theoretisch viele vermeidbare Gas Kosten entstehen. Es wäre von Vorteil, wenn nur eine Abfrage der Adresse erfolgt, falls im aufrufenden Contract nicht die aktuelle Adresse vorliegt.

Über die Einführung eines bool 'live' sowie einer Funktion 'contractlives' zur Überprüfung des Werts von 'live' in registrierten Smart Contracts wäre es möglich nur Abfragen nach Adressen erfolgen zu lassen, wenn dies notwendig ist. Dazu würden Smart Contracts vor jedem externen Aufruf zunächst beim aufzurufenden Contract die Funktion 'contractlives' aufrufen um zu überprüfen, ob der Contract noch die aktuelle Version darstellt. Es könnte folgendermaßen ablaufen:

Kein Pull erforderlich

1) Contract A ruft bei Contract B V2 die Funktion 'contractlives' auf
2) Contract B V2 gibt den bool 'live'=true zurück
3) Contract A ruft eine oder mehrere Funktionen von Contract B V2 auf

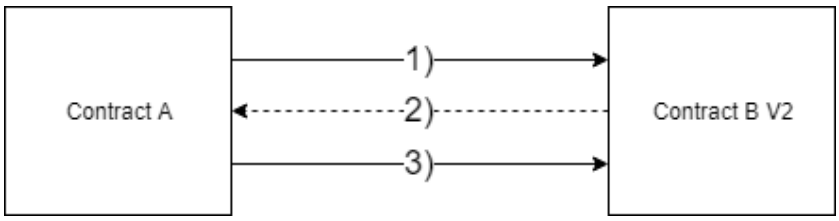

Abbildung 8.1: Chronologischer Ablauf ohne Pull

Pull erforderlich

1) Contract B V3 wird im Registry Contract registriert
2) Der Registry Contract ruft in Contract B V2 eine Funktion auf, welche den bool 'live' auf false setzt

3) Contract A ruft bei Contract B V2 die Funktion 'contractlives' auf

4) Contract B V2 gibt den bool 'live'=false zurück

5) Contract A ruft im Registry Contract die Funktion 'getnwestAddress' mit dem Namen von Contract B als Input Parameter auf, welche ihm die Adresse von Contract B V3 zurückgibt

6) Contract A setzt bei sich neue Adresse von Contract B V3

7) Contract A ruft eine oder mehrere Funktionen von Contract B V3 auf

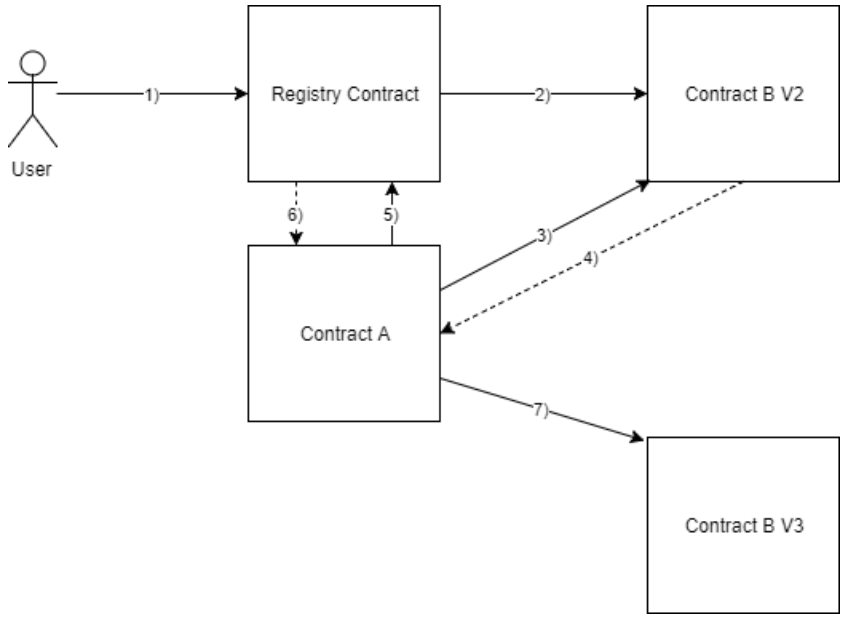

Abbildung 8.2: Chronologischer Ablauf mit Pull

Es ist fragwürdig, ob diese Änderung eine Verbesserung zum jetzigen Entwurfsmuster darstellt, da in jedem Fall ein externer Aufruf vor dem eigentlich auszuführenden externen Aufruf erfolgt. Im jetzigen Entwurfsmuster wird 'getnewestAddress' aufgerufen und eine Adresse im aufrufenden Contract entsprechend des Rückgabewerts gesetzt. In der eben genannten Alternative wird 'contractlives' aufgerufen und der Rückgabewert überprüft. Des Weiteren erfordert dieser Ablauf weitere Elemente in der Grundstruktur eines zu registrierenden Smart Contracts,

welche beim Funktionsaufruf von 'registercontract' wie mit 'setnewRCAddress' auf Existenz überprüft werden müssten.

Im Laufe der Arbeit wurde mithilfe des Codes von CheckMortal in Remix getestet, ob ein Smart Contract noch Werte zurückgeben kann, nachdem der Befehl 'selfdestruct' ausgeführt wurde. Dann wäre das eben vorgestellte Vorgehen mit per 'selfdestruct' deaktivierten Smart Contracts realisierbar. Ein deaktivierter Smart Contract kann jedoch keine Werte mehr zurückgeben.

Auch die Veröffentlichung eines 'event'[1] hilft in diesem Zusammenhang nicht, da Smart Contracts nicht auf die Daten von Events zugreifen können (Ethereum community, 2018f; Ethereum community, 2018i).

Das Push-Prinzip wäre eine Alternative, wurde jedoch in Abschnitt 7.2 verworfen. Die Frage ist, welches Prinzip letztendlich mehr Gas Kosten verursacht. Das Pull-Prinzip sorgt bei jedem externen Aufruf einer Funktion für kleine extra Gas Kosten, welche sich aufsummieren. Das Push-Prinzip hingegen würde im Falle der Registrierung eines Updates punktuell hohe Gas Kosten verursachen, da im ganzen Register überprüft werden müsste, ob andere Smart Contracts von dem Update abhängen und denen, wo dies der Fall ist, die neue Adresse mitteilen. Es müsste für das Push-Prinzip auch eine Art Publish/Subscribe Mechanismus geben, über den Smart Contracts bestimmen, von welchen anderen Smart Contracts sie immer die aktuelle Adresse haben wollen. Dies könnte über ein zusätzliches Array aus Strings in der struct 'Contract' gelöst werden, in dem die Namen der abonnierten anderen Smart Contracts gespeichert werden. Dort könnten auch Namen nachträglich hinzugefügt oder gelöscht werden. Welches Prinzip effizienter ist, müsste durch Tests überprüft werden.

Allgemein besteht im System mit Sicherheit noch viel Optimierungsbedarf bei den Gas Kosten. Insbesondere die Funktion 'setnewRCAddress' wird bei einem System mit vielen registrierten Smart Contracts hohe Gas Kosten verursachen. Wie in Unterabschnitt 5.2.2 dargestellt wird jede Aktualisierung der Registry Con-

[1] Ein Event ist ein Ereignis, welches von Smart Contracts ausgelöst werden kann. Darin können auch Parameter eingebettet werden. Die Daten eines Events werden in der Blockchain gespeichert und enthalten die Adresse des erstellenden Smart Contract. Über einen Ethereum Client können externe Applikationen Events von Smart Contracts abonnieren und darauf reagieren (Ethereum community, 2018f). Ein Beispiel eines Events wäre, wenn ein EOA einen bestimmten Mindestbetrag an Ether an einen Smart Contract schickt und so ein Event auslöst.

tract Adresse in den registrierten Smart Contracts 5.000 Gas kosten, den externen Aufruf noch nicht eingerechnet. Dazu kommt der Registertransfer, welcher mit 20.000 Gas pro neu gesetztem Wert veranschlagt wird und die Grundgebühr von 21.000 Gas für eine Transaktion. Es muss auch die Nutzbarkeit der Variablen und Funktionen des Registry Contracts berücksichtigt werden, welche momentan per Standardwert public ist. Dies auf public zu belassen dürfte jedoch kein Problem darstellen, da die Funktionen, bei denen das Zugriffsrecht eine Rolle spielt, eine Authentifizierungsüberprüfung implementiert haben[2]. Das System wird über die Zeit sukzessiv mehr Speicher belegen, je mehr Smart Contracts sich registrieren.

Bei einem Registertransfer für das Update des Registry Contracts könnten die Funktionen 'setAddressHistory' und 'setowner' zu einer Funktion zusammengefasst werden. Die mit dem Input Parameter string _Name über ein mapping verbundenen Attribute 'AddressHistory' sowie 'owner' könnten in einer Funktion vom alten Registry Contract übergeben und im neuen Registry Contract gesetzt werden. Dies würde die Gas Kosten verringern, da statt zwei externen Aufrufen nur einer erfolgen müsste.

8.3 Publish/Subscribe

User, die Smart Contracts im System anmelden wollen, müssen im Vorfeld die Adresse des Registry Contract wissen. Zusätzlich müssen die Namen anderer Smart Contracts, unter denen diese im Registry Contract angemeldet wurden und mit denen ein zu registrierender Smart Contract interagieren will bekannt sein, damit über diese Namen deren Adressen abgerufen werden können. Eine Lösung wäre es, eine Website oder einen Informationsdienst Off-Chain zu haben, die bzw. der Daten zum Registry Contract bereitstellt. Es könnte außerdem ein 'event' eingebaut werden, falls ein Update des Registry Contracts erfolgt, sodass direkt die neue Adresse veröffentlicht wird. Ein 'event' könnte gleichzeitig beim eben genannten Informationsdienst zur direkten Aktualisierung der Registry Contract Adresse Off-Chain genutzt werden, da ein 'event' und dessen Information für die Außenwelt zugänglich sind (Wohrer und Zdun, 2018b, S. 4). Im Informationsdienst sollten

[2]Gemeint sind die Funktionen 'registercontract', 'setnewowner' sowie 'setnewRCAddress'.

die 'owner' registrierter Smart Contracts Beschreibungen und Anleitungen zu ihren Smart Contracts bereitstellen, damit andere User einen Überblick über das System erhalten können. Dort könnten beispielsweise Funktionsbeschreibungen, der Name unter dem ein Smart Contract registriert wurde, die Versionen und die Abhängigkeiten zu anderen Smart Contracts dargestellt werden.

8.4 Grundstruktur registrierter Smart Contracts

Mit der Einführung eines bool 'needRCAddress' in der struct 'Contract' des Regsitry Contracts könnte verhindert werden, dass alle zu registrierenden Smart Contracts die Funktion 'setnewRCAddress' wie in der Grundstruktur in Abbildung 7.1 haben müssten. Smart Contracts, die nicht auf andere Smart Contracts zugreifen, brauchen nicht die Adresse des Registry Contracts und müssten dementsprechend auch nicht dessen Adresse bei einem Update mitgeteilt bekommen. Wenn dieser bool auf false gesetzt ist, würde also bei einem Aufruf von 'setnewRCAddress' im Registry Contract dieser registrierte Contract nicht aufgerufen werden, um die neue Adresse des Registry Contract zu erhalten. Sollte ein Update registriert werden, welches in seinen Vorgängerversionen diesen bool auf false hatte, nun aber doch auf andere Smart Contracts zugreift, kann dies bei der Registrierung berücksichtigt werden. Der bool wäre ein Input Parameter bei der Registrierung und falls dieser true ist, wird überprüft, ob 'setnewRCAddress' im zu registrierenden Contract vorhanden ist. So wird sichergestellt, dass das System stabil bleibt und es bei einem Update des Registry Contract nicht zum Abbruch der Funktion 'setnewRCAddress' kommt, weil ein Contract diese Funktion nicht besitzt. Eine Änderung des bool 'needRCAddress' von true auf false wäre auch möglich, falls ein Smart Contract ab einem Update nicht mehr auf andere Smart Contracts zugreift.

In der Grundstruktur von Smart Contracts könnte eine Funktion 'destroyme' eingeführt werden, welche den Befehl 'selfdestruct' bei Aufruf ausführt. So könnte automatisiert eine alte Version deaktiviert werden, wenn eine Updateversion im Registry Contract angemeldet wird. Damit würde sichergestellt werden, dass nur noch mit der aktuellen Version gearbeitet werden kann. Das Zugriffsrecht auf diese Funktion müsste an den Registry Contract gegeben werden, welches im *constructor* gesetzt werden könnte. So wird gleichzeitig dem Registry Contract mehr Einfluss

im System gegeben und das Vertrauen der User noch mehr beansprucht. Des Weiteren muss wie bei 'setnewRCAddress' überprüft werden, ob diese Funktion bei der Registrierung vorhanden ist. Bei der Anmeldung eines Updates würde ein Aufruf der Funktion 'destroyme' bei der Vorgängerversion erfolgen. Ist diese Funktion nicht vorhanden, hätte dies einen Funktionsabbruch von 'registercontract' zur Folge und das Update würde nicht angemeldet werden. Es wäre kontraproduktiv, wenn direkt bei der Anmeldung im Registry Contract ein 'selfdestruct' Befehl des zu registrierenden Contracts durchgeführt werden würde. Daher müsste ein Mechanismus eingebaut werden, beispielsweise über einen Zähler, welcher den Befehl 'selfdestruct' in der Funktion 'destroyme' nur ausführt, wenn diese das zweite Mal aufgerufen wird. Eine unkompliziertere aber bei Durchführung auch aufwändigere Variante wäre, den 'owner' der registierten Contracts es selber zu überlassen, ob sie ihre alten Versionen manuell deaktivieren oder sie im System lassen.

8.5 Abwärtskompatibilität

Eine Limitierung des Entwurfsmuster ist, dass ein Update nicht beliebige neue Features und Änderungen beinhalten kann. In Abschnitt 5.1 wurde darauf eingegangen, welche Voraussetzungen Smart Contracts benötigen, um mit anderen Smart Contracts zu interagieren. Ein Update darf keine neuen Input oder Output Parameter bei bestehenden Funktionen einführen, da sonst diese Funktionen von anderen Smart Contracts entweder gar nicht mehr genutzt werden können oder unerwartete Rückgabewerte liefern. Zudem dürfen sich die Namen der Funktionen nicht verändern. Die Algorithmen innerhalb von Funktionen können hingegen beliebig verändert und angepasst werden. Es könnten auch völlig neue Funktionen eingebaut werden. Diese können dann nur nicht extern von anderen Smart Contract genutzt werden, da einerseits das Interface dafür nicht vorhanden ist und andererseits der Aufruf für diese Funktionen nicht programmiert ist. Demzufolge müssen Update Smart Contracts eine Abwärtskompatibilität aufweisen.

8.6 Schwachstellen

Durch die Updatability wird die Immutabilität der Blockchain umgangen. Der Registry Contract stellt eine zentrale Instanz in diesem System dar, was kritisch betrachtet werden muss. Beim jetzigen Stand des Entwurfsmusters sind User des Registry Contracts darauf angewiesen, dass der 'owner' des Registry Contracts nur sinnvolle Updates des Registry Contracts einführt. Der Registry Contract stellt also einen Single Point of Failure dar, weil er eine essentielle Komponente ist, ohne die das System nicht funktioniert (Neugebauer, 2018, S. 355-356). Doch nicht nur der 'owner' des Registry Contract hat in diesem System Macht. Auch 'owner' registrierter Contracts könnten das Entwurfsmuster missbrauchen. Wenn viele Smart Contracts mit einem anderen registrierten Smart Contract interagieren, könnte der 'owner' ein Update einfügen, welches nichts mit der Vorgängerversion zu tun hat. Dieses Update könnte im schlimmsten Fall Ether aus anderen Smart Contracts des Systems ziehen. Die Betriebssicherheit des Systems ist ebenfalls durch User gefährdet. Es könnte beispielsweise ein Update Contract registriert werden, der nicht funktioniert. Dann sind alle Smart Contracts, die von diesem abhängig sind, ebenso zumindest teilweise nicht mehr funktionsfähig. Wenn von diesen wiederum andere abhängig sind, könnte das eine Kettenreaktion auslösen, welche große Teile des Systems lahmlegt. Dies würde auch passieren, wenn ein 'owner' den Befehl 'selfdestruct' in seinem registrierten Smart Contract implementiert und bei seiner aktuellen Version ausführt. Stand jetzt sind User gezwungen, immer auf die neueste Version eines Contracts zuzugreifen, auch wenn das eventuell nicht erwünscht ist. Es müsste eine Art Wahl im System eingeführt werden, damit User Vertrauen zu dem System haben. Nur wenn Parteien einem Update zustimmen, welche von diesem Update betroffen wären, sollte es im Registry Contract aufgenommen werden. Eine Demokratisierung erscheint unerlässlich, um ein sicheres System zu etablieren. Auf der Ethereum Homepage werden Beispiele solcher Maßnahmen vorgestellt (Ethereum Foundation, 2018). Bei der Registrierung eines Smart Contracts könnte der 'owner' automatisch ein Wahlrecht bei der Einführung eines Update Registry Contracts erhalten. Es könnte auch so aufgebaut werden, dass der Einfluss der Stimme auf eine Wahl proportional zu der Anzahl an registrierten Smart Contracts ist. Ein Nachteil dabei ist, dass die Einführung von Updates verlang-

samt wird. Es sollte ein verhältnismäßiger Mittelweg zwischen Wahlperiode eines Updates und der Verzögerung dieses Updates gewählt werden.

Eine andere Möglichkeit für User dem System zu schaden besteht in der Registrierung vieler sinnloser Contracts. Die Gas Kosten für die Registrierung von Smart Contracts sind vermutlich nicht hoch genug, um User von einer Spam-Registrierung abzuhalten. Dadurch könnten viele potentielle Namen für andere User blockiert werden, da aktuell kein Mechanismus implementiert ist, welcher einen Namenstransfer oder eine Namenslöschung einer Smart Contract Reihe ermöglicht. Außerdem würde das System künstlich mit sinnlosen Inhalten vergrößert werden, was insbesondere bei einem Registertransfer zu einem Update Registry Contract unnötige Kosten verursacht. Es könnten Moderatoren-Accounts über Adressen eingeführt werden, welche das System überwachen und bei einer Störung dieser Art regulierend einschreiten. Doch so wird wieder etwas mehr Zentralität in das System gebracht und User müssten diesen Moderatoren vertrauen, dass diese nicht deren registrierte Smart Contracts aus dem Registry Contract entfernen. Auch hier könnte wieder eine Wahl stattfinden, bevor ein Smart Contract aus dem Registry Contract entfernt wird. Dann müsste festgelegt werden, wer ein Wahlrecht in dieser Hinsicht erhält.

Es gibt für User eine Möglichkeit Updates des Registry Contracts zu verhindern. Ein User könnte einen Smart Contract registrieren, welcher so programmiert ist, dass er den Befehl 'selfdestruct' ausführen kann. Wenn der 'selfdestruct' Befehl ausgeführt wurde und dieser Contract eine aktuelle Version eines registrierten Smart Contracts darstellt, kann die Funktion 'setnewRCAddress' des Registry Contracts bei diesem Smart Contract die Funktion 'setnewRCAddress' nicht mehr aufrufen und die Funktion würde abbrechen, sodass keinem Smart Contract die neue Registry Contract Adresse mitgeteilt werden kann. Wie bereits erwähnt und im Code von CheckMortal getestet, kann der Registry Contract vorher auch nicht überprüfen, ob ein Smart Contract den Befehl 'selfdestruct' ausgeführt hat, da dieser keine Funktionsaufrufe mehr zulässt.

Das Einfügen eines Smart Contracts in die Blockchain und das Anmelden im Registry Contract sind im Entwurfsmuster zwei separate Schritte. Handelt es sich um eine erstmalige Registrierung eines Smart Contracts, sollte diese direkt nach dem Einfügen des Contracts in die Blockchain erfolgen. Ein anderer User könnte

den Smart Contract registrieren und als 'owner' gespeichert werden, wenn sich zwischen diesen Schritten zu viel Zeit gelassen wird und ein anderer User auf den Smart Contract aufmerksam wird. Das ist zwar recht unwahrscheinlich, sollte jedoch vermieden werden.

9 Zusammenfassung und Ausblick

Der Stromsektor erlebt momentan einen grundlegenden strukturellen Wandel. Einer der wichtigsten Gründe liegt in der Begrenzung des anthropogenen Treibhauseffekts. Erneuerbare Energien, Dezentralisierung, Sektorkopplung und Digitalisierung sind die resultierenden Veränderungen. Damit ein permanentes Gleichgewicht von Erzeugung und Verbrauch bei zunehmend volatiler Einspeisung gewährleistet ist, muss sich die Kommunikation, Koordination und Automatisierung im System erhöhen (Appelrath u. a., 2012, S. 12). Dezentrale regenerative Erzeugung führt zu einer steigenden Anzahl an Prosumern. Der lukrative Energiehandel bleibt Erzeugern kleinerer Anlagen aufgrund zu hoher Transaktionskosten über beispielsweise die Börse verwehrt (Holstenkamp und Radtke, 2018, S. 110-111; Merz, 2016, S. 18-19).

Die Blockchain Technologie könnte einen P2P-Energiehandel ohne Mittelsmänner ermöglichen. Transparenz, Sicherheit und vergleichsweise geringe Transaktionskosten sind Eigenschaften, welche sich als vorteilhaft für diese Rolle erweisen (Swan, 2015, S. 30; Peters und Panayi, 2015, S. 1; Pop u. a., 2018, S. 3). Der Handel würde über Smart Contracts erfolgen, welche Programme in der Blockchain darstellen. Diese autonomen und unveränderbaren Programme können von EOAs genutzt und die Funktionalitäten entsprechend ihres Codes ausgeführt werden (Mylrea und Gourisetti, 2017, S. 3). Die Unveränderbarkeit macht jedoch die Fehlerbehebung und Funktionalitätenerweiterung unmöglich (Wohrer und Zdun, 2018b, S. 2). Die einzige Möglichkeit besteht im Einführen eines neuen Smart Contracts, welcher nur über eine neue Adresse aufgerufen werden kann.

Für die Erstellung eines interagierenden, updatable Smart Contract Systems in Ethereum ist eine Adressen-Verwaltung unerlässlich, da nur über Adressen auf Smart Contracts zugegriffen werden kann. Das Entwurfsmuster eines Registry Contracts für das Anmelden von Smart Contracts und das Abrufen von Smart

Contract Adressen ist geeignet, ein solches System zu realisieren. Das Abrufen von Adressen über Namen macht das System für menschliche User angenehmer, da andernfalls die direkte Auseinandersetzung mit Adressen über lange Hexadezimalwerte erfolgen müsste. Auch der Registry Contract selber kann Updates erhalten, was jedoch kostspielig hinsichtlich Gas Kosten sein dürfte, da das Register der alten Version transferiert und den registrierten Smart Contracts die neue Adresse mitgeteilt werden muss.

Auch wenn die grundlegenden Funktionalitäten eines updatable Smart Contract Systems vorhanden sind, konnten im vorigen Kapitel bereits einige Risiken seitens der 'owner' und Potentiale zur Verbesserung ausgemacht werden.

Interessant in diesem Zusammenhang dürften Hybridkonzepte aus den in Abschnitt 6.1 vorgestellten Konzepten sein. Zur Vermeidung eines teuren Registertransfers bei jedem Update des Registry Contracts könnte eine Trennung von Daten und Logik vorgenommen werden. So blieben die Daten des Registers bei einem Update unangetastet und der vermutlich hohe Gas Kosten verursachende Transfer der Daten zu der Updateversion wird überflüssig. Die Mitteilung der neuen Logik Adresse des Registry Contracts an die registrierten Smart Contracts müsste dennoch erfolgen. Zur Sicherheit sollte der Daten Contract trotzdem so programmiert werden, dass das Register transferiert werden kann. Die Einführung neuer Variablen könnte notwendig werden. Die Option eines Updates macht das System flexibler. Bei einem Update des Daten Contracts müsste nur dem aktuellen Logik Contract die neue Adresse mitgeteilt werden. Registrierte Smart Contracts kommen ausschließlich über den Logik Contract mit dem Daten Contract in Kontakt und müssen demzufolge auch nicht die Adresse des Daten Contracts speichern. In einer solchen Umsetzung würde der Registry Contract in zwei Komponenten, einen Logik Contract und einen Daten Contract, aufgespalten werden, wie in Abbildung 6.1 dargestellt.

Zur stärkeren Automatisierung des Systems sollte untersucht werden, ob das Einfügen eines Smart Contracts in die Blockchain und die Registrierung des Smart Contracts im Registry Contract in einem Schritt erfolgen kann. In Abschnitt 7.3 wurde die automatische Anmeldung im *constructor* diskutiert, doch da die Authentifizierung über 'tx.origin' nicht sicher ist, wurde dieser Ansatz verworfen. Eventuell ist es möglich in der Funktion 'registercontract' den Code eines zu re-

gistrierenden Contracts als Input Parameter zu setzen, sodass das Einfügen eines Smart Contracts vom Registry Contract übernommen wird. Es ist möglich einen sogenannten Factory Contract zu erstellen, welcher mehrere gleichartige Smart Contracts erstellt und in die Blockchain einfügt (Xu u. a., 2018, S. 28-29; Stack Overflow, 2017a). Doch dabei können nur gleichartige Smart Contracts mithilfe von Code erstellt werden, welcher im Vorhinein bekannt sein muss. Für das System wäre es erforderlich unbekannten Code durch den Registry Contract in die Blockchain einzufügen. Wenn dies möglich ist, wäre das ein erstrebenswertes Ziel, da so Usern ein Schritt erspart bleibt. Ein Entwicklerteam hat sich damit bereits auseinandergesetzt und Ansätze veröffentlicht (Aragon Project Blog, 2017).

Eine Upgradability[1] ist in diesem Entwurfsmuster nicht implementiert. Eine automatisierte Upgradability wird nicht realisierbar sein, da dafür der Code von anderen Smart Contracts, die auf das Upgrade zugreifen, verändert werden muss. Dies wurde mithilfe der Testcodes eines Caller Contracts und eines Upgrade Contracts getestet. Ein Schritt in Richtung Upgradability könnte vielleicht über updatable Interface Contracts erreicht werden. Zu jedem registrierten Smart Contract könnte ein Interface Contract registriert werden, welcher ausschließlich das Interface dieses Smart Contracts beinhaltet. Sollte es möglich sein, über die Adressen der Interface Contracts deren Inhalte in andere Smart Contracts zu importieren, könnten dynamische Interfaces für abhängige Smart Contracts zur Verfügung gestellt werden. Bei einem Upgrade könnte der 'owner' ebenfalls ein Update des Interface Contracts im Registry Contract einbinden, sodass automatisch das Interface des Upgrades in die davon abhängigen Smart Contracts aufgenommen wird. Dennoch müsste danach noch jeder von dem Upgrade abhängige Smart Contract ebenfalls ein Upgrade erhalten, um die neuen oder veränderten Funktionalitäten vollends nutzen zu können. Ob Interfaces aus anderen Smart Contracts überhaupt dynamisch importiert werden können ist jedoch nicht klar und müsste getestet werden.

[1] Unter Upgradability wird in dieser Arbeit im Zusammenhang mit der Ethereum Blockchain folgendes verstanden: Die Einführung neuer Input und/oder Output Parameter in bestehenden Funktionen, die Namensänderung bestehender Funktionen oder die Einführung neuer Funktionen bei gleichzeitiger Interoperabilität von anderen Smart Contracts zu dem Upgrade.

Anhang

Die folgenden Testprogramme wurden eigenständig erstellt und die Tests, auf die in der Arbeit hingewiesen wurde, in der Online Umgebung Remix durchgeführt (Remix, 2018).

```
1  pragma solidity ^0.4.25;
2
3  contract CheckMortal {
4
5      address owner;                      // address variable for the owner
6
7      event destroyed(bool dead);         // event showing that this contract
           selfdestructed
8
9      bool live = true;                   // bool showing whether this contract is
           still alive
10
11     constructor() {                     // set owner to msg.sender for the
           selfdestruct command
12         owner = msg.sender;
13     }
14
15     function destroyme() {              // sets the bool live to false, emits an
           event and selfdestructs itself, return all remaining ether to owner
16
17         live = false;
18         emit destroyed(true);
19         selfdestruct(owner);
20
21     }
22
23     function contractlives() constant returns(bool) {   // function returns the
           bool live which says, whether this contract has already selfdestructed or
           not
24
25         return live;
26
27     }
```

```
}
```

Code 1: Solidity Testcode von CheckMortal

```
pragma solidity ^0.4.25;


contract Registry {

    struct Contract {               // creating a struct for registered contracts, which contains their address history and their owner
        address[] AddressHistory;
        address owner;
    }

    mapping(string => Contract) contracts;      // linking the struct with string, so that the info of the struct can be accessed or changed via a string name

    constructor() {                 // setting an entry of its own (Registry)

        contracts["RC"].owner = msg.sender;
        contracts["RC"].AddressHistory.push(address(this));
    }

    function registercontract(string _Name, address _addr) {  // function for registering a new contract or an update of an already registered contract

        contracts[_Name].owner = msg.sender;
        contracts[_Name].AddressHistory.push(_addr);

    }

    function getnewestAddress(string _Name) constant returns(address) {
                        // returns the latest address of the contract _Name

        return contracts[_Name].AddressHistory[contracts[_Name].AddressHistory.length-1];

    }

    function getAddressHistory(string _Name) constant returns(address[]) {  // returns all addresses in an array of the contract _Name

        return contracts[_Name].AddressHistory;

    }

```

```
38      function getowner(string _Name) constant returns(address) {     // returns the
            owner of the contract _Name
39
40          return contracts[_Name].owner;
41
42      }
43
44  }
```

Code 2: Solidity Testcode eines Registry Contracts ohne Namensregister

```
1   pragma solidity ^0.4.25;
2
3   contract Caller {
4
5       Upgrade c;
6
7       function setnewCalleeAddress(address _addr) {   // set new Upgrade address
8
9           c = Upgrade(_addr);
10      }
11
12      function testsetint(int _a) {                   // calls the function setint
            of Upgrade
13
14          c.setint(_a);
15
16      }
17
18      function testgetint() returns(int) {            // calls the function getint
            of Upgrade
19
20          return c.getint();
21
22      }
23
24      function testincrement() {                      // calls the function
            increment of Upgrade
25
26          c.increment();
27
28      }
29
30  }
31
32  contract Upgrade {                                  // Define function names, input
        parameters and outputs of the called contract so that this contract can call
        the other contract by its function names.
```

```
33      function setint(int _a);                    // Interface is created, otherwise
            they cannot interact.
34      function increment();                       // The name of this instance does
            not matter! Only the function names and their inputs and outputs have to
            match the called contract!
35      function getint() constant returns(int);
36  }
```

Code 3: Solidity Testcode von einem Upgrade abhängigen Smart Contract

```
1   pragma solidity ^0.4.25;
2
3   contract Upgrade {
4
5       int a;
6       int b;
7
8       function setint(int _a, int _b) {           // set a and b to input
            parameters
9
10          a = _a;
11          b = _b;
12
13      }
14
15      function getint() constant returns(int, address) {  // return int a and the
            address of this contract
16
17          return (a,address(this));
18
19      }
20
21      function incrementboth() {                      // increment a and b
22
23          a = a+1;
24          b = b+1;
25
26      }
27
28      function decrement() {                      // decrement a and b
29
30          a = a-1;
31          b = b-1;
32
33      }
34
35  }
```

Code 4: Solidity Testcode eines Upgrade Smart Contracts

Quellenverzeichnis

Aichele, C. und O. D. Doleski, Hrsg. (2014). *Smart Market: vom Smart Grid zum intelligenten Energiemarkt*. Wiesbaden: Springer Vieweg. ISBN: 978-3-658-02777-3 978-3-658-02778-0.

Amato, A. u. a. (2015). "Design and evaluation of P2P overlays for energy negotiation in smart micro-grid". In: *Computer Standards & Interfaces* 44. ISSN: 09205489. DOI: 10.1016/j.csi.2015.04.004.

Appelrath, H.-J., H. Kagermann und C. Mayer, Hrsg. (2012). *Future energy grid: Migrationspfade ins Internet der Energie*. Acatech-Studie. Berlin: Springer. ISBN: 978-3-642-27863-1 978-3-642-27864-8.

Aragon Project Blog (2017). *Advanced Solidity code deployment techniques*. Aragon Project Blog. URL: http://blog.aragon.org/advanced-solidity-code-deployment-techniques-dc032665f434/ (besucht am 15. 10. 2018).

BDEW (2018). *Was bedeutet die Digitalisierung für die Energiewirtschaft?* URL: http://www.bdew.de/energie/digitalisierung/was-bedeutet-der-trend-der-digitalisierung-fuer-die-energiewirtschaft/ (besucht am 11. 10. 2018).

BEE (2018). *Effizient Erneuerbar: Was JETZT zum Gelingen einer Erneuerbaren Wärmewende getan werden muss*. URL: https://www.bee-ev.de/unsere-positionen/waerme/ (besucht am 15. 10. 2018).

Bergstra, J. A. und M. Burgess (2018). *Blockchain Technology and its Applications A Promise Theory view - V0.11*.

BitInfoCharts (2018a). *Cryptocurrency prices for today in real time*. BitInfoCharts. URL: https://bitinfocharts.com/cryptocurrency-prices/#EUR (besucht am 17. 10. 2018).

– (2018b). *Ethereum / Ether (ETH) statistics - Price, Blocks Count, Difficulty, Hashrate, Value*. BitInfoCharts. URL: https://bitinfocharts.com/ethereum/ (besucht am 17. 10. 2018).

Bundesministerium für Wirtschaft und Energie (2018). *Entwicklung der Stromerzeugung aus erneuerbaren Energien in Deutschland*. URL: https://www.bmwi.de/Redaktion/DE/Infografiken/Energie/entwicklung-stromerzeugung-erneuerbare-energien-deutschland.html (besucht am 15. 10. 2018).

Bundesnetzagentur (2018). *Regelenergie*. URL: https://www.bundesnetzagentur.de/DE/Sachgebiete/ElektrizitaetundGas/Unternehmen_Institutionen/Versorgungssicherheit/Engpassmanagement/Regelenergie/regelenergie-node.html (besucht am 15. 10. 2018).

– (2017a). *Digitale Transformation in den Netzsektoren*.

– (2017b). *Quartalsbericht zu Netz- und Systemsicherheitsmaßnahmen: Viertes Quartal und Gesamtjahr 2016*.

Bundesregierung (2009). *Nationaler Entwicklungsplan Elektromobilität*.

Buterin, V. (2015). *Notes on Scalable Blockchain Protocols (version 0.3.2)*.

– (2016). *Ethereum: Platform Review*.

– (2017). *A next Generation Smart Contract & Decentralized Application Platform*.

Buterin, V. und V. Griffith (2017). *Casper the Friendly Finality Gadget*.

Chandra, P. u. a. (2017). *Implementing Mechanisms as Smart Contracts on Blockchains*.

Chen, L. u. a. (2018). "Protecting Early Stage Proof-of-Work Based Public Blockchain". In: *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. Luxembourg: IEEE. ISBN: 978-1-5386-6553-4. DOI: 10.1109/DSN-W.2018.00050.

Chohan, U. W. (2017). "The Double Spending Problem and Cryptocurrencies". In: *SSRN Electronic Journal*. ISSN: 1556-5068. DOI: 10.2139/ssrn.3090174.

CoinMarketCap (2018). *Kryptowährung Marktkapitalisierungen*. URL: https://coinmarketcap.com/de/ (besucht am 15. 10. 2018).

Crastan, V. und D. Westermann, Hrsg. (2018). *Elektrische Energieversorgung. 3: Dynamik, Regelung und Stabilität, Versorgungsqualität, Netzplanung, Betriebsplanung und -führung, Leit- und Informationstechnik, FACTS, HGÜ*. 2., aktualisierte Auflage. Berlin: Springer Vieweg. ISBN: 978-3-662-49020-4 978-3-662-49021-1.

Daonomic (2018). *Upgradeable Ethereum Smart Contracts*. URL: https://medium.com/@daonomic/upgradeable-ethereum-smart-contracts-d036cb373d6 (besucht am 20. 10. 2018).

Davis, M. W. (2002). *Distributed Resource Electric Power Systems Offer Significant Advantages Over Central Station Generation and T & D Power Systems Part II*.

Destefanis, G. u. a. (2018). "Smart contracts vulnerabilities: a call for blockchain software engineering?" In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. Campobasso: IEEE. ISBN: 978-1-5386-5986-1. DOI: 10.1109/IWBOSE.2018.8327567.

Deutsche Energie-Agentur (2018). *Liberalisierung des Strommarktes*. URL: https://www.dena.de/themen-projekte/energiesysteme/strommarkt/ (besucht am 15. 10. 2018).

Doleski, O. D., Hrsg. (2017). *Herausforderung Utility 4.0: wie sich die Energiewirtschaft im Zeitalter der Digitalisierung verändert*. Wiesbaden: Springer Vieweg. ISBN: 978-3-658-15736-4 978-3-658-15737-1.

EEX (2018). *Preisliste*. URL: https://www.eex.com/de/handel/preisliste (besucht am 15. 10. 2018).

Eiselt, J. (2012). *Dezentrale Energiewende: Chancen und Herausforderungen*. Praxis. Wiesbaden: Springer Vieweg. ISBN: 978-3-8348-2461-5 978-3-8348-2462-2.

Etherchain (2018). *Ethereum Hard Fork History - etherchain.org*. URL: https://www.etherchain.org/hardForks (besucht am 16. 10. 2018).

Ethereum community (2018a). *Account Types, Gas, and Transactions — Ethereum Homestead 0.1 documentation*. URL: http://ethdocs.org/en/latest/contracts-and-transactions/account-types-gas-and-transactions.html#what-is-a-transaction (besucht am 15. 10. 2018).

– (2018b). *Contracts — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/contracts.html#visibility-and-getters (besucht am 16. 10. 2018).

– (2018c). *Contracts — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/contracts.html#creating-contracts (besucht am 16. 10. 2018).

Ethereum community (2018d). *Contracts — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/contracts.html#fallback-function (besucht am 16. 10. 2018).
– (2018e). *Contracts — Solidity 0.5.0 documentation*. URL: https://solidity.readthedocs.io/en/latest/contracts.html#constant-state-variables (besucht am 16. 10. 2018).
– (2018f). *Contracts — Solidity 0.5.0 documentation*. URL: https://solidity.readthedocs.io/en/latest/contracts.html#events (besucht am 15. 10. 2018).
– (2018g). *Frequently Asked Questions — Solidity 0.2.0 documentation*. URL: https://solidity-doc-test.readthedocs.io/en/latest/frequently-asked-questions.html#can-a-contract-function-accept-a-two-dimensional-array (besucht am 15. 10. 2018).
– (2018h). *Introduction to Smart Contracts — Solidity 0.4.24 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html#gas (besucht am 15. 10. 2018).
– (2018i). *Introduction to Smart Contracts — Solidity 0.4.24 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.24/introduction-to-smart-contracts.html#logs (besucht am 15. 10. 2018).
– (2018j). *Security Considerations — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/security-considerations.html (besucht am 15. 10. 2018).
– (2018k). *Solidity Assembly — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/assembly.html (besucht am 15. 10. 2018).
– (2018l). *The Ethereum Wiki. Contribute to ethereum/wiki development by creating an account on GitHub*. URL: https://github.com/ethereum/wiki/wiki/Programming-languages-intro (besucht am 15. 10. 2018).
– (2018m). *Types — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/types.html#structs (besucht am 19. 10. 2018).
– (2018n). *Types — Solidity 0.5.0 documentation*. URL: https://solidity.readthedocs.io/en/latest/types.html#mappings (besucht am 19. 10. 2018).
– (2018o). *Units and Globally Available Variables — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/units-and-global-variables.html#block-and-transaction-properties (besucht am 16. 10. 2018).

Ethereum community (2018p). *Units and Globally Available Variables — Solidity 0.4.25 documentation*. URL: https://solidity.readthedocs.io/en/v0.4.25/units-and-global-variables.html (besucht am 15. 10. 2018).

– (2018q). *Units and Globally Available Variables — Solidity 0.5.0 documentation*. URL: https://solidity.readthedocs.io/en/latest/units-and-global-variables.html#contract-related (besucht am 18. 10. 2018).

Ethereum Foundation (2018). *Create a Democracy contract in Ethereum*. URL: https://www.ethereum.org/dao (besucht am 15. 10. 2018).

Etherscan (2018a). *Ethereum Average BlockSize Chart*. URL: https://etherscan.io/chart/blocksize (besucht am 17. 10. 2018).

– (2018b). *Ethereum Average BlockTime Chart*. URL: https://etherscan.io/chart/blocktime (besucht am 17. 10. 2018).

– (2018c). *Ethereum Gas Price Tracker*. URL: https://etherscan.io/gastracker (besucht am 17. 10. 2018).

Fiedler, I., F. Fiedler und L. Ante (2016). *Die Vision eines integrierten Energiemarktes*.

Frantz, C. K. und M. Nowostawski (2016). "From Institutions to Code: Towards Automated Generation of Smart Contracts". In: *2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. Augsburg, Germany: IEEE. ISBN: 978-1-5090-3651-6. DOI: 10.1109/FAS-W.2016.53.

Froystad, P. und J. Holm (2016). *Blockchain: Powering the Internet of Value*.

Gobmaier, T. (2018). *Online-Messung der Netzfrequenz: Aktuelle Informationen*. URL: http://www.netzfrequenzmessung.de/aktuelles.htm (besucht am 15. 10. 2018).

Goll, J. (2014). *Architektur- und Entwurfsmuster der Softwaretechnik: mit lauffähigen Beispielen in Java*. 2., aktualisierte Aufl. Wiesbaden: Springer. ISBN: 978-3-658-05531-8.

Graeber, D. R. (2014). *Handel mit Strom aus erneuerbaren Energien*. Aufl. 2014. essentials. Wiesbaden: Springer Fachmedien Wiesbaden GmbH. ISBN: 978-3-658-05940-8.

Gupta, M. (2017). *Blockchain For Dummies IBM Limited Edition*.

Gupta, M. (17. Sep. 2018). *How to make smart contracts upgradable!* Hacker Noon. URL: https://hackernoon.com/how-to-make-smart-contracts-upgradable-2612e771d5a2 (besucht am 15. 10. 2018).

Häfner, L. (2017). "Demand Side Management: Entscheidungsunterstützungssysteme für die flexible Beschaffung von Energie unter integrierten Chancen- und Risikoaspekten". In: *HMD Praxis der Wirtschaftsinformatik* 55.3. ISSN: 1436-3011, 2198-2775. DOI: 10.1365/s40702-017-0363-9.

Hajek, S. (2018). *Elektromobilität: Hält das Stromnetz dem E-Auto-Boom stand?* URL: https://www.wiwo.de/technologie/mobilitaet/elektromobilitaet-haelt-das-stromnetz-dem-e-auto-boom-stand/20231296.html (besucht am 15. 10. 2018).

Hasse, F. u. a. (2016). *Blockchain – an opportunity for energy producers and consumers?* PricewaterhouseCoopers International Limited.

Heimberger, M. u. a. (2017). "Energieträgerübergreifende Planung und Analyse von Energiesystemen". In: *e & i Elektrotechnik und Informationstechnik* 134.3, S. 229–237. ISSN: 0932-383X, 1613-7620. DOI: 10.1007/s00502-017-0504-4.

Holstenkamp, L. und J. Radtke (2018). *Handbuch Energiewende und Partizipation.* Springer VS Handbuch. Wiesbaden: Springer VS, Springer Fachmedien Wiesbaden. ISBN: 978-3-658-09415-7.

International Energy Agency (2017). *World energy outlook 2017: Executive summary.*

IqtiyaniIlham, N., M. Hasanuzzaman und M. Hosenuzzaman (2017). "European smart grid prospects, policies, and challenges". In: *Renewable and Sustainable Energy Reviews* 67. ISSN: 13640321. DOI: 10.1016/j.rser.2016.09.014.

Jacob, F., J. Mittag und H. Hartenstein (2015). "A Security Analysis of the Emerging P2P-Based Personal Cloud Platform MaidSafe". In: *2015 IEEE Trustcom/BigDataSE/ISPA.* Helsinki, Finland: IEEE. ISBN: 978-1-4673-7952-6. DOI: 10.1109/Trustcom.2015.538.

Jahn, A., G. Rosenkranz und C. Podewils (2017). *Energiewende und Dezentralität. Zu den Grundlagen einer politisierten Debatte.*

Kaltschmitt, M., W. Streicher und A. Wiese, Hrsg. (2006). *Erneuerbare Energien: Systemtechnik, Wirtschaftlichkeit, Umweltaspekte ; mit 83 Tabellen.* 4., aktualisierte, korrigierte und erg. Aufl. Berlin: Springer. ISBN: 978-3-540-28204-4.

Kiffer, L., D. Levin und A. Mislove (2017). "Stick a fork in it: Analyzing the Ethereum network partition". In: *Proceedings of the 16th ACM Workshop on Hot Topics in Networks - HotNets-XVI*. the 16th ACM Workshop. Palo Alto, CA, USA: ACM Press. ISBN: 978-1-4503-5569-8. DOI: 10.1145/3152434.3152449.

Kok, K. (2013). "The PowerMatcher: Smart Coordination for the Smart Electricity Grid". Diss. Technical University of Denmark.

Konstantin, P. (2017). *Praxisbuch Energiewirtschaft: Energieumwandlung, -transport und -beschaffung, Übertragungsnetzausbau und Kernenergieausstieg*. 4., aktualisierte Auflage. VDI-Buch. Berlin: Springer Vieweg. ISBN: 978-3-662-49822-4 978-3-662-49823-1.

Kost, C. u. a. (2018). *Stromgestehungskosten erneuerbare Energien*.

Kounelis, I. u. a. (2017). *Blockchain in energy communities a proof of concept*.

Krieger, S. und M. Nickel (2012). *Wettbewerb 2012: Wo steht der deutsche Energiemarkt?* Bundesverband der Energie- und Wasserwirtschaft.

Kroposki, B., T. Basso und R. DeBlasio (2008). "Microgrid standards and technologies". In: *2008 IEEE Power and Energy Society General Meeting - Conversion and Delivery of Electrical Energy in the 21st Century*. Energy Society General Meeting. Pittsburgh, PA, USA: IEEE. ISBN: 978-1-4244-1905-0. DOI: 10.1109/PES.2008.4596703.

Lund, H. u. a. (2016). "Energy Storage and Smart Energy Systems". In: *International Journal of Sustainable Energy Planning and Management, Vol 11 (2016)*. DOI: 10.5278/ijsepm.2016.11.2.

Mayer, C. und C. Dänekas (2013). "Smart Grids – die Bedeutung der Informatik für die zukünftige Energieversorgung". In: *Informatik-Spektrum* 36.1. ISSN: 0170-6012, 1432-122X. DOI: 10.1007/s00287-012-0636-1.

Mazrekaj, A., I. Shabani und B. Sejdiu (2016). "Pricing Schemes in Cloud Computing: An Overview". In: *International Journal of Advanced Computer Science and Applications* 7.2. ISSN: 21565570, 2158107X. DOI: 10.14569/IJACSA.2016.070211.

Merz, M. (2016). "Einsatzpotenziale der Blockchain im Energiehandel". In: *Blockchain Technology*. Hrsg. von D. Burgwinkel. Berlin, Boston: De Gruyter. ISBN: 978-3-11-048895-1. DOI: 10.1515/9783110488951-003.

Ministerium für Energiewende, Landwirtschaft, Umwelt, Natur und Digitalisierung (2016). *Abregelung von Strom aus Erneuerbaren Energien.pdf*. Kiel.

Mylrea, M. und S. N. G. Gourisetti (2017). "Blockchain: A path to grid modernization and cyber resiliency". In: *2017 North American Power Symposium (NAPS)*. Morgantown, WV: IEEE. ISBN: 978-1-5386-2699-3. DOI: 10.1109/NAPS.2017.8107313.

Nadolinski, E. (2018). *Proxy Patterns*. ZeppelinOS Blog. URL: https://blog.zeppelinos.org/proxy-patterns/ (besucht am 15. 10. 2018).

Nakamoto, S. (2008). *Bitcoin: A Peer-to-Peer Electronic Cash System*.

Neugebauer, R., Hrsg. (2018). *Digitalisierung: Schlüsseltechnologien für Wirtschaft und Gesellschaft*. 1. Auflage. Fraunhofer-Forschungsfokus. Berlin Heidelberg: Springer Vieweg. ISBN: 978-3-662-55889-8 978-3-662-55890-4.

Niranjanamurthy, M., B. N. Nithya und S. Jagannatha (2018). "Analysis of Blockchain technology: pros, cons and SWOT". In: *Cluster Computing*. ISSN: 1386-7857, 1573-7543. DOI: 10.1007/s10586-018-2387-5.

Peck, M. E. (2017). *Do You Need a Blockchain? This chart will tell you if the technology can solve your problem*.

Perez-Sola, C. u. a. (2017). *Double-spending Prevention for Bitcoin zero-confirmation transactions*.

Peter, V. u. a. (2017). *Blockchain in der Energiewirtschaft: Potenziale für Energieversorger*. Bundesverband der Energie- und Wasserwirtschaft.

Peters, G. W. und E. Panayi (2015). *Understanding Modern Banking Ledgers through Blockchain Technologies: Future of Transaction Processing and Smart Contracts on the Internet of Money*.

Plazibat, A. (2016). *Blockchain, mehr als nur ein Hype? – Eine Einführung in die Blockchain*. URL: http://www.ccsourcing.news/blockchain-mehr-als-nur-ein-hype-eine-einfuehrung-in-die-blockchain/ (besucht am 15. 10. 2018).

Pop, C. u. a. (2018). "Blockchain Based Decentralized Management of Demand Response Programs in Smart Energy Grids". In: *Sensors* 18.2. ISSN: 1424-8220. DOI: 10.3390/s18010162.

Praktiknjo, A. (2013). *Sicherheit der Elektrizitätsversorgung: das Spannungsfeld von Wirtschaftlichkeit und Umweltverträglichkeit*. Wiesbaden: Springer. ISBN: 978-3-658-04343-8.

Rauf, Z. (2018). *Upgradeable smart contracts in Ethereum.* URL: https://zohaib.me/upgradeable-smart-contracts-in-ethereum/ (besucht am 20. 10. 2018).

Redlich, T., M. Moritz und J. P. Wulfsberg, Hrsg. (2018). *Interdisziplinäre Perspektiven zur Zukunft der Wertschöpfung.* Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-20264-4 978-3-658-20265-1. DOI: 10.1007/978-3-658-20265-1.

Remix (2018). *Remix - Solidity IDE.* URL: https://remix.ethereum.org/#optimize=false&version=soljson-v0.4.25+commit.59dbf8f1.js (besucht am 15. 10. 2018).

Rezaee Jordehi, A. (2016). "Allocation of distributed generation units in electric power systems: A review". In: *Renewable and Sustainable Energy Reviews* 56. ISSN: 13640321. DOI: 10.1016/j.rser.2015.11.086.

Röder, D. (2016). *Netzwerk des Vertrauens: Blockchain.*

Rosenberger, P. (2018). *Bitcoin und Blockchain: vom Scheitern einer Ideologie und dem Erfolg einer revolutionären Technik.* Berlin, Germany: Springer Vieweg. ISBN: 978-3-662-56087-7 978-3-662-56088-4.

Schellong, W. (2016). *Analyse und Optimierung von Energieverbundsystemen.* 1. Auflage. OCLC: 945131889. Berlin Heidelberg: Springer Vieweg. ISBN: 978-3-662-48527-9 978-3-662-49463-9.

Scherer, M. (2017). "Performance and Scalability of Blockchain Networks and Smart Contracts". Diss. Umea University.

Schlatt, V. u. a. (2016). *Blockchain - Grundlagen, Anwendungen und Potentiale.* Bayreuth: Fraunhofer-Institut für Angewandte Informationstechnik FIT.

Schmied, M. und P. Wüthrich (2015). *Postfossile Energieversorgungsoptionen für einen treibhausgasneutralen Verkehr im Jahr 2050 – eine verkehrsträgerübergreifende Bewertung.*

Scholz, B., V. Rißland und M. Sauer (2012). *Smart Grids in Deutschland: Handlungsfelder für Verteilnetzbetreiber auf dem Weg zu intelligenten Netzen.* Bundesverband der Energie- und Wasserwirtschaft.

Schuberth, J. (2018). *Umgebungswärme und Wärmepumpen.* Umweltbundesamt. URL: https://www.umweltbundesamt.de/themen/klima-energie/erneuerbare-energien/umgebungswaerme-waermepumpen#Effizienz (besucht am 15. 10. 2018).

Schwab, A. J. (2009). *Elektroenergiesysteme: Erzeugung, Transport, Übertragung und Verteilung elektrischer Energie*. 2., aktualisierte Aufl. Berlin: Springer. ISBN: 978-3-540-92226-1 978-3-540-92227-8.

Sommerville, I. (2011). *Software engineering*. 9th ed. Boston: Pearson. ISBN: 978-0-13-703515-1 978-0-13-705346-9.

Stack Overflow (2016a). *contract design - Difference between CALL, CALLCODE and DELEGATECALL*. Ethereum Stack Exchange. URL: https://ethereum.stackexchange.com/questions/3667/difference-between-call-callcode-and-delegatecall (besucht am 15. 10. 2018).

– (2016b). *contract design - How do I make my DAPP SSerenity-Proof?"*. Ethereum Stack Exchange. URL: https://ethereum.stackexchange.com/questions/196/how-do-i-make-my-dapp-serenity-proof (besucht am 15. 10. 2018).

– (2016c). *solidity - Who pays gas when a contract function that creates/calls another contract is called?* Ethereum Stack Exchange. URL: https://ethereum.stackexchange.com/questions/1452/who-pays-gas-when-a-contract-function-that-creates-calls-another-contract-is-cal (besucht am 15. 10. 2018).

– (2017a). *Deploy contract from contract in Solidity*. Ethereum Stack Exchange. URL: https://ethereum.stackexchange.com/questions/13415/deploy-contract-from-contract-in-solidity (besucht am 15. 10. 2018).

– (2017b). *ethereum - String array in solidity*. Stack Overflow. URL: https://stackoverflow.com/questions/42716858/string-array-in-solidity (besucht am 15. 10. 2018).

Statista (2018). *Handelsvolumen an der EEX-Strombörse bis 2017 | Statistik*. Statista. URL: https://de.statista.com/statistik/daten/studie/12486/umfrage/entwicklung-der-eex-handelsvolumina/ (besucht am 15. 10. 2018).

Statistisches Bundesamt (2018). *Wirtschaftsbereiche - Energie - Erzeugung - Statistisches Bundesamt (Destatis)*. URL: https://www.destatis.de/DE/ZahlenFakten/Wirtschaftsbereiche/Energie/Erzeugung/Tabellen/BilanzElektrizitaetsversorgung.html (besucht am 15. 10. 2018).

Sterner, M. und I. Stadler (2014). *Energiespeicher: Bedarf, Technologien, Integration*. Berlin: Springer Vieweg. ISBN: 978-3-642-37379-4 978-3-642-37380-0.

Swan, M. (2015). *Blockchain: blueprint for a new economy*. First edition. Beijing : Sebastopol, CA: O'Reilly. ISBN: 978-1-4919-2049-7.

Synwoldt, C. (2016). *Dezentrale Energieversorgung mit regenerativen Energien*. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-13046-6 978-3-658-13047-3. DOI: 10.1007/978-3-658-13047-3.

Tanner, J. (2018). *Summary of Ethereum Upgradeable Smart Contract Strategies*. Indorse. URL: https://blog.indorse.io/ethereum-upgradeable-smart-contract-strategies-456350d0557c (besucht am 20. 10. 2018).

Tichler, R. und S. Moser (2017). "Systemische Notwendigkeit zur Weiterentwicklung von Hybridnetzen". In: *e & i Elektrotechnik und Informationstechnik* 134.3. ISSN: 0932-383X, 1613-7620. DOI: 10.1007/s00502-017-0499-x.

Tsankov, P. u. a. (2018). *Securify: Practical Security Analysis of Smart Contracts*. arXiv: 1806.01143.

Umweltbundesamt (2018). *Energieverbrauch nach Energieträgern, Sektoren und Anwendungen*. Umweltbundesamt. URL: http://www.umweltbundesamt.de/daten/energie/energieverbrauch-nach-energietraegern-sektoren (besucht am 15. 10. 2018).

Vogel, H.-J., K. Weißer und W. D.Hartmann (2018). *Smart City: Digitalisierung in Stadt und Land*. Wiesbaden: Springer Fachmedien Wiesbaden. ISBN: 978-3-658-19045-3 978-3-658-19046-0. DOI: 10.1007/978-3-658-19046-0.

Voshmgir, S. (2016). *Blockchains, Smart Contracts und das Dezentrale Web*.

Vossen, G. und K.-U. Witt (2016). *Grundkurs Theoretische Informatik: eine anwendungsbezogene Einführung - für Studierende in allen Informatik-Studiengängen*. 6., erweiterte und überarbeitete Auflage. Lehrbuch. Wiesbaden: Springer Vieweg. ISBN: 978-3-8348-1770-9 978-3-8348-2202-4.

Vukolić, M. (2017). "Rethinking Permissioned Blockchains". In: *Proceedings of the ACM Workshop on Blockchain, Cryptocurrencies and Contracts - BCC '17*. the ACM Workshop. Abu Dhabi, United Arab Emirates: ACM Press. ISBN: 978-1-4503-4974-1. DOI: 10.1145/3055518.3055526.

Wiesner, T. (2017). *Upgrade Smart Contracts on Chain*. URL: https://vomtom.at/upgrade-smart-contracts-on-chain/ (besucht am 20. 10. 2018).

Wilke, S. (2013). *Primärenergiegewinnung und -importe*. Umweltbundesamt. URL: https://www.umweltbundesamt.de/daten/energie/primaerenergiegewinnung-importe (besucht am 15. 10. 2018).

Willi Horenkamp u. a. (2007). *Dezentrale Energieversorgung 2020*. Energietechnische Gesellschaft im VDE.

Wittpahl, V., Hrsg. (2017). *Digitalisierung: Bildung, Technik, Innovation: iit-Themenband*. Berlin Heidelberg: Springer Vieweg. ISBN: 978-3-662-52853-2 978-3-662-52854-9.

Wohrer, M. und U. Zdun (2018a). "Design Patterns for Smart Contracts in the Ethereum Ecosystem". In: *2018 IEEE International Conference on Blockchain*. Halifax: IEEE.

– (2018b). "Smart contracts: Security Patterns in the Ethereum Ecosystem and Solidity". In: *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*. Campobasso: IEEE. ISBN: 978-1-5386-5986-1. DOI: 10.1109/IWBOSE.2018.8327565.

Wood, D. G. (2018). *Ethereum: A secure decentralized generalized Transaction Ledger*.

Wyman, O. (2018). *E-Mobilität bedroht stabile Stromversorgung*. URL: https://www.oliverwyman.de/media-center/2018/jan/E-Mobilitaet-bedroht-stabile-Stromversorgung.html (besucht am 15. 10. 2018).

Xu, X. u. a. (2018). *A Pattern Collection for Blockchain-based Applications*.

Zahoransky, R. und H.-J. Allelein, Hrsg. (2013). *Energietechnik: Systeme zur Energieumwandlung ; Kompaktwissen für Studium und Beruf ; mit 46 Tabellen*. 6., überarb. und erw. Aufl. Studium. Wiesbaden: Springer Vieweg. ISBN: 978-3-8348-1869-0 978-3-8348-2279-6.

Zeppelin (2018). *ZeppelinOS*. URL: https://zeppelinos.org (besucht am 15. 10. 2018).

Zhang, C., J. Wu, C. Long u. a. (2017). "Review of Existing Peer-to-Peer Energy Trading Projects". In: *Energy Procedia* 105. ISSN: 18766102. DOI: 10.1016/j.egypro.2017.03.737.

Zhang, C., J. Wu, Y. Zhou u. a. (2018). "Peer-to-Peer energy trading in a Microgrid". In: *Applied Energy* 220. ISSN: 03062619. DOI: 10.1016/j.apenergy.2018.03.010.

Zichy, M. u. a. (2011). *Energie aus Biomasse - ein ethisches Diskussionsmodell: eine Studie des Institutes Technik-Theologie-Naturwissenschaften und des Technologie- und Förderzentrums*. 1. Aufl. Studium. Wiesbaden: Vieweg + Teubner. ISBN: 978-3-8348-1733-4.