CQRS

Benjamin Rupp, Bachelorand, WS 2014/15

1.1 Idee

Command Query Responsibility Segregation (abgekürzt CQRS) ist ein Architekturmuster, welches die Architektur für ein Informationssystem in zwei Anteile trennt: einen **Lese-** und einen **Schreibanteil** [1].

Dieses Architekturmuster basiert auf Bertrand Meyers Prinzip der **Command Query Segregation** (abgekürzt CQS). Nach Meyer sind Methoden, die eine Änderung bewirken, Befehle (englisch Commands) und Methoden, die einen Rückgabewert haben, Abfragen (englisch Queries). Ein Mischen von beidem führt zu schwer erkennbaren Seiteneffekten, die mit CQS verhindert werden sollen [2]. CQRS wendet das Prinzip von Meyer jedoch nicht auf Methoden oder Objekte, sondern auf eine gesamte – oder zumindest auf Teile einer – Architektur an [3]. Durch diese Trennung lassen sich beide Teile unabhängig voneinander entwickeln und unterschiedlich skalieren [1].

CQRS ist als eine **Verfeinerung** des herkömmlichen Schichtenmodells anzusehen. Dieses enthält in der Datenzugriffsschicht und der Schicht der Geschäftslogik jeweils das Bereitstellen von Daten beziehungsweise die Validierung und Ausführung der Befehle. Das Schichtenmodell selbst trifft jedoch keinerlei Aussage über das Unterteilen von Schichten in nebenläufige Einheiten. Wird eine solche Trennung nicht vorgenommen, verstoßen diese Schichten gegen das **Single Responsibility Principle** (abgekürzt SRP), weil zwei eigentlich völlig unterschiedliche Verantwortlichkeiten in einer einzigen Schicht verwendet werden [4]. CQRS möchte sich mit der angestrebten Aufteilung die Vorteile von SRP zu Nutzen machen.

Command Query Responsibility Segregation ist ein relativ einfaches Architekturmuster. Es bietet jedoch aufgrund seiner besonderen Architektur die Möglichkeit, weitere Konzepte anzuwenden.

1.2 Command Query Segregation

Bertrand Meyers Prinzip der Command Query Segregation beschäftigt sich mit Funktionen und deren Auswirkung auf das System, in welchem sich diese Funktionen befinden. Meyer verwendet dabei den Begriff **Seiteneffekt** (englisch side effect). Seiteneffekte treten demnach dann auf, wenn eine Funktion, von der erwartet wird, dass sie einen Wert zurückliefert, zusätzlich den Zustand des Systems verändert.

Um diese Seiteneffekte klar einzuordnen und im Weiteren zu vermeiden, klassifiziert Meyer die Funktionen einer Klasse wie folgt. Für ihn gibt es:

- Abfragen (englisch Queries) und
- Befehle (englisch Commands).

Befehle sind Funktionen, die Objekte verändern. Sie werden als **Prozeduren** implementiert. Bei ihnen ist klar, dass sie den Zustand des Systems beeinflussen. Abfragen hingegen werden als **Ausdruck** implementiert. Ein Ausdruck kann beispielsweise eine Konstante oder Variable sein. Es ist offensichtlich, dass ein Abfragen einer Konstanten oder Variablen den Zustand des Systems nicht verändert. Ein Ausdruck kann jedoch auch eine Funktion sein, die mit Hilfe eines Algorithmus einen Rückgabewert berechnet. Die Frage nach der Auswirkung auf das System macht bei Abfragen also nur im Zusammenhang mit Funktionen Sinn. Meyer spricht dann von Seiteneffekten, wenn eine solche Funktion zusätzlich zu ihrer eigentlichen Aufgabe – dem Zurückgeben eines Zustands – in unerwarteter Weise Änderungen am System vornimmt [2] [5].

Die Trennung zwischen Abfragen und Befehlen wird bei CQRS von der Funktions- und Objektebene auf die Architekturebene gehoben. So profitiert die gesamte Architektur von dieser klaren Trennung der Verantwortlichkeiten.

1.3 Schichtenmodell

Die folgende Abbildung zeigt die Trennung von Abfragen und Befehlen in einer Architektur und stellt sie dem herkömmlichen Schichtenmodell gegenüber:

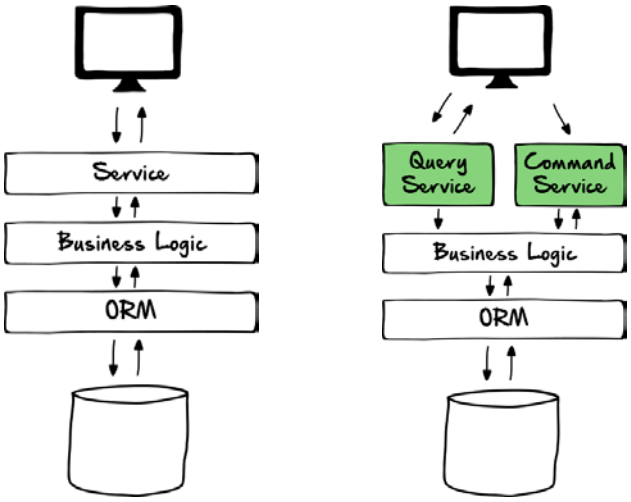

Abbildung 1 - Schichtenmodell ohne und mit CQRS-Muster

Im klassischen Schichtenmodell werden Abfragen und Befehle über ein und dieselbe Schicht koordiniert. Dadurch werden eigentlich unabhängige Verantwortlichkeiten vermischt, wenn man diese Schicht nicht in Teilschichten trennt. Keine Teilschichten für die einzelnen Belange einzuführen, führt zu einer Verletzung des Single Responsibility Principle.

Das Single Responsibility Principle wurde Ende der 1990er Jahre von Robert C. Martin verfasst. Martin bringt darin die Ideen von David L. Parnas („On the Criteria To Be Used in Decomposing Systems into Modules"[1]) und Edsger W. Dijkstra („On the role of scientific thought"[2]) zusammen [6]. Demnach darf ein Softwaremodul nur genau

[1] eine von David L. Parnas 1972 veröffentlichte Publikation über die Unterteilung und Wiederverwendung von Softwaremodulen

[2] eine von Edsger W. Dijkstra 1974 veröffentlichte Publikation, aus der das Prinzip Separation of Concerns (abgekürzt SoC) hervorgeht

einen Grund für Änderungen haben [6]. Was das bedeutet, beschreibt Martin, wie folgt:

> *»Gather together the things that change for the same reasons. Separate those things that change for different reasons.« [6]*

Befolgt man SRP, so muss die Software folglich so aufgeteilt werden, dass die Teile eines Moduls nicht aus verschiedenen Gründen geändert werden müssen. Diesbezüglich stellt sich jedoch die Frage, was der Grund für eine solche Änderung ist. Martin unterscheidet hier zwischen einem **Grund für Änderungen** und dem Begriff der **Verantwortlichkeit**. Geht es nach ihm, muss jedes Softwaremodul einen klaren Verantwortlichen im Unternehmen haben. Dieser Verantwortliche soll möglichst eine Person oder eine kleine, funktional zusammengehörige Gruppe sein. Der Grund für Änderungen sind – so Martin weiter – letztendlich eben diese Personen. Weil sie es sind, die Änderungen in ihren Teilen des Programms vornehmen, sollen sie möglichst nicht vom Programmcode anderer Programmierer verwirrt werden [6].

Durch die Modularisierung des SRP wird der Programmcode einer Software in eigenständige Teile (Module) aufgeteilt, die jeweils eine einzige Aufgabe erfüllen und einen einzigen Verantwortlichen haben. Dieser Gedanke kann auf das Schichtenmodell und im Weiteren auf das CQRS-Muster übertragen werden:

- Das Schichtenmodell vereint sowohl die **Befehls- als auch die Abfrageseite in einer einzigen Schicht**. Diese Schicht enthält damit Softwareteile, die eigentlich vollkommen unabhängig voneinander geändert werden könnten (ein Ändern der Abfrageschnittstelle sollte beispielsweise keine Auswirkungen auf die Befehlsseite haben). Damit verstößt eine solche Architektur laut Martin, wenn man sie nicht verfeinert und in unterschiedliche Verantwortlichkeiten aufteilt, gegen das SRP.
- Mit CQRS werden diese unterschiedlichen **Verantwortlichkeiten voneinander losgelöst**. Änderungen können dort – und nur dort – vorgenommen werden, wo sie tatsächlich gebraucht werden. Das SRP ist auf der Ebene der Schnittstellen zum User Interface (abgekürzt UI) erfüllt.

Die Trennung bei CQRS, wie sie in Abbildung 1 dargestellt wird, hat einen weiteren Vorteil, der wiederum dem SRP entgegenkommt. Durch das Auftrennen der beiden Verantwortlichkeiten in separate Module können diese auch von unterschiedlichen Entwicklerteams bearbeitet werden. Zudem können Befehls- und Abfrageseite auf unterschiedlichen Servern eingerichtet und **asymmetrisch skaliert** werden. Beispielsweise kann die Abfrage-Schnittstelle mehrfach angeboten werden, wenn sie häufiger angefragt wird als die Befehls-Schnittstelle. Diese Situation entspricht eher der Realität, weil ein synchrones Skalieren der beiden Teile nur in seltenen Fällen notwendig ist [1].

1.4 Datenhaltung

Die Datenhaltung ist bei verteilten Systemen, welche von mehreren Anwendern parallel genutzt werden können, besonders zu betrachten. Hier müssen Entscheidungen bezüglich der **Verfügbarkeit** (englisch availability) und **Konsistenz** (englisch consistency) der im System gespeicherten Daten getroffen werden, ohne dabei die **Skalierbarkeit** (englisch scalability) zu vernachlässigen [7].

Verfügbarkeit ist dann gegeben, wenn eine Operation auf eine gewünschte Weise antwortet [7]. Konsistenz der Daten bezeichnet einen Zustand, in welchem die Daten für alle Klienten (englisch clients) gleich sind [8].

1.4.1 Eventual Consistency

Eventual Consistency garantiert, dass es, wenn keine neuen Änderungen vorgenommen werden, einen Zeitpunkt gibt, an dem sich alle Kopien der Persistenz in einem konsistenten Zustand befinden [9]. Deshalb können auch kurze Zeit nach der Änderung eines Datensatzes in einer der Kopien der Persistenz noch veraltete Daten ausgegeben werden, weil eine dieser Kopien diese Änderung noch nicht erhalten hat. Die Zeit, bis sichergestellt ist, dass die aktualisierten Daten zurückgegeben werden – also jene Zeit, in der die Daten inkonsistent sind – nennt man **Inconsistency Window**.

1.4.2 CQRS mit getrennter Datenhaltung

Da das CQRS-Muster für Systeme ausgelegt ist, die wesentlich mehr Anfragen als Befehle zu verarbeiten haben, stellt diese Architektur hohe Anforderungen an die Verfügbarkeit. Es ist deshalb von Vorteil, nicht nur die nach Abfragen und Befehlen getrennte Geschäftslogik zu duplizieren. Auch eine Vervielfältigung der Datenhaltung kann zur erforderlichen Verfügbarkeit beitragen. Der Speicher der Befehlsseite muss in der Lage sein, die vorgenommenen Änderungen auch der anderen Seite bekannt zu machen. Hierfür eignet sich beispielsweise ein Event Store. Mehr dazu in Kapitel 0.

Setzt man die Trennung, welche das CQRS-Muster in der Geschäftslogik vornimmt, auch in der Datenhaltung fort, so erlangt man einige Vorteile. Zum einen lässt sich damit – analog zur Trennung und Vervielfältigung der Geschäftslogik der Abfrageseite – auch der Speicher auf der Abfrageseite beliebig duplizieren, um eine höhere Performanz und Verfügbarkeit zu erreichen. Dies ist möglich, weil auf die Datenhaltung der Abfrageseite nur lesend zugegriffen wird [10, p. 266].

Zum anderen können die beiden voneinander getrennten Datenhaltungen für ihre Aufgaben optimiert werden [11, pp. 225-226]. So ist es möglich, die Befehlsseite als normalisierte, auf Änderungen optimierte Datenhaltung zu implementieren. Die Abfrageseite hingegen kann denormalisiert angelegt werden, um die Daten so abzuspeichern, wie sie später verwendet werden. Damit lassen sich aufwendige Operationen wie beispielsweise Joins vermeiden [10, p. 266].

1.5 Event Sourcing

> *»Event Sourcing is a way of persisting your application's state by storing the history that determines the current state of your application.« [12, p. 236]*

Bei Event Sourcing werden Zustandsänderungen der Anwendung durch Ereignisse (englisch events) ausgelöst [13]. Diese Ereignisse werden in der Reihenfolge ihres Auftretens in der Datenhaltung – einem sogenannten **Event Store** – gespeichert. Bei

Event Sourcing werden statt Objekten lediglich die Änderungen an diesen Objekten in Form von Events persistent abgelegt. Anhand dieser chronologisch gespeicherten Ereignisse können Objekte in jeder ihrer historischen Versionen geladen werden. Darüber hinaus geht die Absicht einer Zustandsänderung nicht verloren, weil sie in Form der Events mit abgespeichert wird.

1.5.1 Nutzerinteraktion

Die Zustandsänderungen des Systems, die bei Event Sourcing gespeichert werden sollen, werden vom Anwender ausgelöst. Die Software stellt ein User Interface zur Verfügung, mit dessen Hilfe der Nutzer seine Aufgaben lösen kann. Häufig werden hierfür **CRUD[3]-Anwendungen** eingesetzt. Dort bilden die Buttons des UI die CRUD-Logik der Persistenz ab. Das User Interface ist deshalb weniger auf die Aktionen des Nutzers angepasst [14, p. 99].

Der Nachteil bei CRUD-Anwendungen ist, dass die eigentliche Absicht des Nutzers verloren geht [15, p. 10], da lediglich die Änderung und nicht deren Grund an das System übermittelt werden. Da die Nutzerabsicht durchaus von Interesse sein kann, muss beim Softwareentwurf darüber entschieden werden, ob deren Verlust – durch das Einsetzen eines CRUD-UI – toleriert werden kann.

Als Alternative bietet sich das Konzept der **Task-based User Interfaces** an. Bei diesen stehen die Aufgaben des Nutzers und damit die Geschäftsprozesse im Vordergrund. Das UI soll so gestaltet werden, dass der Anwender durch seinen zu bearbeitenden Prozess geleitet wird [15, p. 14]:

> *»The basic idea behind a Task Based […] UI is that its important to figure out how the users want to use the software and to make it guide them through those processes.«*

Abbildung 2 zeigt sowohl einen CRUD als auch einen Task-based UI-Screen im Vergleich:

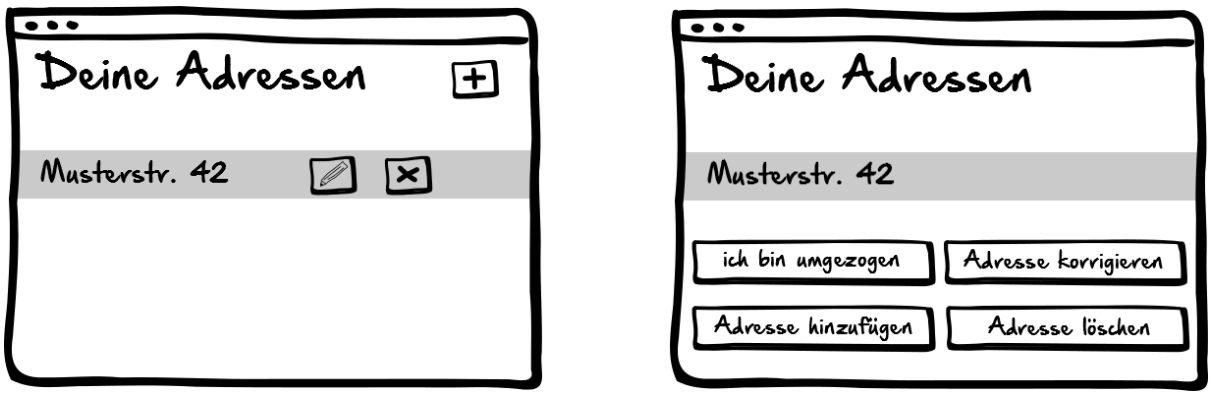

Abbildung 2 - Beispiel für ein CRUD (links) und Task-based User Interfaces (rechts)

Task-based User Interfaces eignen sich besser als CRUD-UIs für den Einsatz von Event Sourcing, da die Zustandsänderungen des Systems schon in Form von Nachrichten vorliegen.

[3] CRUD steht für create, read, update und delete

1.5.2 Messaging

Die Kommunikation innerhalb eines Systems basiert bei Event Sourcing auf Nachrichten. Alan Kay[4] sieht die Kommunikation zwischen Softwaremodulen als den Schlüssel für skalierbare Systeme:

> *»The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.« [16]*

Kay sprach schon in seinem ursprünglichen Konzept [17] der objektorientierten Programmierung davon, dass Objekte nur über Nachrichten kommunizieren sollten. Ziel war es, Objekte als biologische Zellen zu betrachten, deren Kommunikation ausschließlich über Nachrichten funktioniert. Kay räumt später ein, dass es eine Weile dauerte, bis man herausfand, wie man dieses Prinzip effizient auf die Software anwenden konnte.

1.5.2.1 Ereignisse und Befehle

In Zusammenhang mit Event Sourcing und später auch CQRS werden zwei Arten von Nachrichten näher betrachtet [10, p. 268]: **Ereignisse** (englisch events) und **Befehle** (englisch Commands).

Ereignisse haben folgende Eigenschaften [12, p. 235]:

- Ereignisse treten in der Vergangenheit auf.
- Ereignisse sind unveränderlich.
- Ereignisse sind unidirektional und können mehrere Empfänger haben.

Befehle hingegen zeichnen sich durch diese Merkmale aus [10, p. 253]:

- Befehle werden im Imperativ formuliert.
- Befehle sind Anfragen an das System, um eine Aufgabe oder Aktion durchzuführen.
- Befehle werden meist nur einmalig und von nur einem Empfänger ausgeführt.

Mit Befehlen werden Aktionen im System ausgelöst. Events berichten über Dinge, die schon geschehen sind [11, pp. 228-229].

Ereignisse können nicht verändert oder rückgängig gemacht werden, da sie eine durchgeführte Aktion wiedergeben [12, p. 235]. Stattdessen können zusätzliche Ereignisse vorherige Ereignisse negieren. Zum Beispiel das Ereignis: „Die Reservierung wurde widerrufen."

[4] Alan Kay ist Erfinder der Programmiersprache Smalltalk und prägte darüber hinaus den Begriff der objektorientierten Programmierung. [18]

1.5.3 Event Store

Um die Ereignisse bei Event Sourcing persistent abzulegen, wird ein sogenannter **Event Store** verwendet [12, pp. 245-246]. Dieser dient dazu, Ereignisse zu speichern und – beim Wiederherstellen eines Aggregatzustands – bestimmte Ereignisströme zu laden. Ein solcher Event Store kann durch eine relationale, aber auch durch eine NoSQL-Datenbank oder irgendeine andere Form von Persistenz realisiert werden[5]. Da Ereignisse bei Event Sourcing unveränderlich sind – also nur gespeichert, jedoch nicht verändert werden – handelt es sich bei einem Event Store um einen Speicher, der – neben dem Auslesen – mit einfachen und schnellen Einfüge-Operationen auskommt. Der Event Store gilt als **Single Source Of Truth**[6], weil ein System damit – außer den Ereignissen im Event Store – keine weitere Persistenz benötigt [12, p. 240].

1.5.4 CQRS und Event Sourcing

Große Vorteile bietet Event Sourcing, wenn die CQRS-Architektur eine getrennte Datenhaltung für die Befehls- und Leseseite hat. Abbildung 3 zeigt eine Architektur, die sowohl CQRS als auch Event Sourcing nutzt.

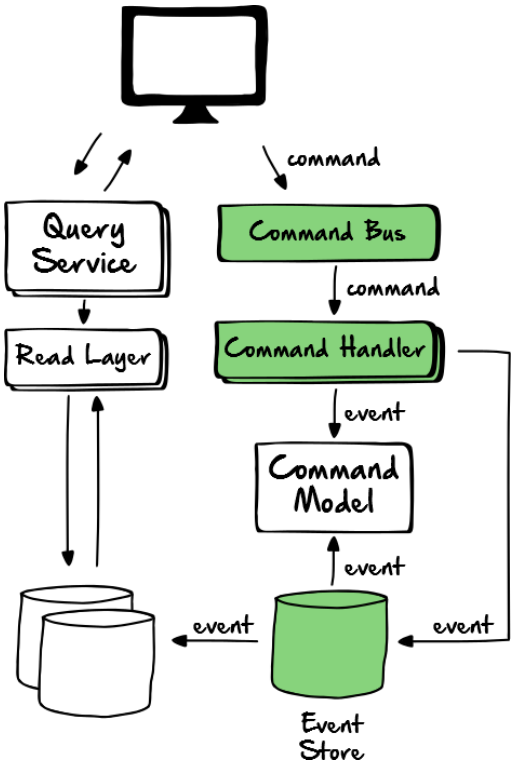

Abbildung 3 - Architektur mit CQRS und Event Sourcing

[5] Wird eine relationale Datenhaltung verwendet, enthält diese nur eine einzige Tabelle, welche die Events chronologisch abspeichert. Diese Art der Persistenz ist nicht zu verwechseln mit einem klassischen CRUD-Datenbankmodell, in welchem Entitäten abgelegt werden.

[6] Single Source of Truth wird im Zusammenhang des DRY-Prinzips verwendet und hat dieselbe Bedeutung.

1.6 Fazit

CQRS ist ein Architekturmuster, welches auf Teile einer Architektur angewandt werden kann. Es ist nicht dafür gedacht, für eine gesamte Architektur angewendet zu werden. Mit CQRS soll die Trennung von Commands und Queries auf Architekturebene ermöglicht werden. Die Trennung der Logik und Datenhaltung sowie Event Sourcing resultieren lediglich aus der mit CQRS erreichten Architektur. Sie sind keinesfalls Bestandteil des CQRS-Musters.

In folgender Tabelle sind die Vor- und Nachteile von CQRS aufgeführt:

Vorteile	Nachteile
● Skalierbarkeit des Systems ● geringere Komplexität der einzelnen Schichten ● Flexibilität	● Umdenken notwendig ● erhöhter Aufwand in Analyse, Design und Implementierung

CQRS ist nicht als Standard-Architekturmuster anzusehen, das für alle Systeme verwendet werden kann. Der Aufwand, welcher mit CQRS verbunden ist, sollte gut mit den Vorteilen abgewogen werden. Hier einige Entscheidungskriterien, die für und gegen den Einsatz von CQRS sprechen:

CQRS	kein CQRS
● … wenn eine **asymmetrische Skalierung** von Anfrage- und Befehlsseite erforderlich ist ● … wenn die Anwendung **skalieren** muss ● … wenn die **Geschäftslogik komplex** ist und Abfragen ineffizient macht ● … wenn **Event Sourcing** Vorteile für die Anwendung bringt	● … wenn es sich um eine einfache **CRUD**-**Anwendung** handelt ● … wenn die Applikation klein ist und **nicht skalieren** muss ● … wenn die Geschäftslogik nicht davon profitiert ● … wenn die Anwendung nicht mit vielen Anfragen umgehen muss

Quellenverzeichnis

[1] G. Young, „CQRS, Task Based UIs, Event Sourcing agh!," CodeBetter.com, 16 2 2010. [Online]. Available: http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/. [Zugriff am 13 8 2014].

[2] B. Meyer, „Commands and queries," in s *Object-Oriented Software Construction*, 2. Edition Hrsg., New Jersey, Prentice Hall PTR, 1997, p. 748.

[3] M. Nijhof, „CQRS à la Greg Young," 12 11 2009. [Online]. Available: http://cre8ivethought.com/blog/2009/11/12/cqrs--la-greg-young. [Zugriff am 22 10 2014].

[4] M. Heimeshoff und P. Jander, „Getrennt sind wir stark," 05 02 2013. [Online]. Available: http://www.heise.de/developer/artikel/CQRS-neues-Architekturprinzip-zur-Trennung-von-Befehlen-und-Abfragen-1797489.html. [Zugriff am 13 08 2014].

[5] B. Meyer, „Forms of side effect," in s *Object-Oriented Software Construction*, 2. Edition Hrsg., New Jersey, Prentice Hall PTR, 1997, pp. 748-749.

[6] R. C. Martin, „The Single Responsibility Principle," 8 5 2014. [Online]. Available: http://blog.8thlight.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html. [Zugriff am 15 8 2014].

[7] D. Pritchett, „BASE: An Acid Alternative," *acm queue,* pp. 48-55, 05/06 2008.

[8] E. Brewer, „CAP Twelve Years Later: How the "Rules" Have Changed," *IEEE Computer,* pp. 23-29, 2 2012.

[9] W. Vogels, „Eventually consistent," *acm queue,* pp. 14-19, 10 2008.

[10] D. Betts, J. Domínguez, G. Melnik, F. Simonazzi und M. Subramanian, „Reference 4: A CQRS and ES Deep Dive," in s *Exploring CQRS and Event Sourcing*, Microsoft, 2012, pp. 247-281.

[11] D. Betts, J. Domínguez, G. Melnik, F. Simonazzi und M. Subramanian, „Reference 2: Introducing the Command Query Responsibility Segregation Pattern," in s *Exploring CQRS and Event Sourcing*, Microsoft, 2012, pp. 223-234.

[12] D. Betts, J. Domínguez, G. Melnik, F. Simonazzi und M. Subramanian, „Reference 3: Introducing Event Sourcing," in s *Exploring CQRS and Event Sourcing*, Microsoft, 2012, pp. 235-246.

[13] M. Fowler, „Event Sourcing," martinfowler.com, 12 12 2005. [Online]. Available: http://martinfowler.com/eaaDev/EventSourcing.html. [Zugriff am 12 9 2014].

[14] D. Betts, J. Domínguez, G. Melnik, F. Simonazzi und M. Subramanian, „Journey 5: Preparing for the V1 Release," in s *Exploring CQRS and Event Sourcing*, Microsoft, 2012, pp. 91-124.

[15] G. Young, „CQRS Documents by Greg Young," [Online]. Available: http://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf. [Zugriff am 26 9 2014].

[16] A. Kay, Squeak-dev Mailing List, 1998.

[17] S. Ram und A. Kay, „Dr. Alan Kay on the Meaning of "Object-Oriented Programming"," userpage.fu-berlin.de, 23 07 2003. [Online]. Available: http://www.purl.org/stefan_ram/pub/doc_kay_oop_en. [Zugriff am 15 9 2014].

[18] „People - Founders - Alan Kay," Viewpoints Research Institute, [Online]. Available: http://www.viewpointsresearch.org/html/people/founders.htm. [Zugriff am 16 9 2014].