HOCHSCHULE LANDSHUT
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

Fakultät Informatik

Masterarbeit

Reactive Design Patterns

Alexander Vetter

Betreuerin: Prof. Dr. Gudrun Schiedermeier

ERKLÄRUNG ZUR MASTERARBEIT

Vetter, Alexander

Hochschule Landshut

Fakultät Informatik

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

2016–02–17 .....

Datum *Unterschrift*

Abstract

Die vorliegende Arbeit beschäftigt sich mit Design Patterns, die die Eigenschaften des Reactive Manifestos erfüllen. Ziel der Arbeit wird es sein, Design Patterns zu identifizieren, die bei der Entwicklung von reaktiven Systemen von Nutzen sein können. Zu Beginn werden neu entstandene Anforderungen an Software Systeme erläutert, die die Motivation hinter dem Reactive Manifestos erklären. Für den Hauptteil werden die Grundlagen zu funktionaler Programmierung sowie den genauen Eigenschaften reaktiver Systeme dargelegt. Im Hauptteil werden dann eine Auswahl an Design Patterns genauer beleuchtet und festgestellt, ob diese die Eigenschaften des Reactive Manifestos erfüllen. Dazu werden dann Code Beispiele folgen, die den Umgang mit den Patterns in der jeweiligen Sprache darstellen sollen. Zum Schluss werden die Ergebnisse zusammen gefasst und ein Ausblick auf die Verwendung von reaktiven Systemen in moderner Softwarearchitektur gegeben.

Abstract

To be done.

Inhaltsverzeichnis

1 Einführung

Software wird entwickelt um Probleme zu lösen und Aufgaben automatisiert zu erledigen, ohne dabei menschliche Fehler zu machen. Deshalb ist es nicht verwunderlich, dass man an Software die Anforderung stellt, in angemessener Zeit auf Eingaben zu reagieren. Hierfür gibt es den Begriff *responsiveness* (dt. Antwortbereitschaft) [Kuh15, S. 18].
Ein System welches auf externe Ereignisse bzw. Eingaben antwortet (reagiert), nennt man reaktives System. David Harel und Amir Pnueli unterschieden in ihrer Arbeit „On the Development of Reactive Systems“ (1985) zwischen reaktiven und transformativen Systemen [HP85]. Sie definierten reaktive Systeme wie folgt:

> Reactive systems [...] are repeatedly prompted by the outside world and their role is to continuously respond to external inputs [HP85].

Transformative Systeme hingegen berechnen Ergebnisse einmalig auf Basis bestimmter Eingabewerte (z.B. Compiler) [Car14, S. 2] [Wie03]. Bei reaktiven Systemen handelt es sich üblicherweise um durchgehend laufende Anwendungen wie zum Beispiel eine Web Applikation.
In dieser Arbeit wird die Entwicklung von reaktiven Software Systeme behandelt, bei welchen die erwähnte *responsiveness*, sowie weitere Anforderungen von großer Bedeutung sind.

1.1 Wandel der Anforderungen

Die heutigen Anforderungen an Software unterliegen dem rapiden Wachstum der Nutzung von Software im technischen wie auch im sozialen Sinne. Der Wandel hin zu einer digitalen Gesellschaft und der weltweiten Vernetzung durch das Internet machen Software allgegenwertig. Die Art und Weise wie moderne Systeme implementiert werden, unterliegt ebenfalls diesem Wandel. Um den heutigen Ansprüchen zu genügen, müssen diese robuster, skalierbarer und anpassungsfähiger sein, als es früher der Fall war [BFKT14].

Früher wurden mithilfe von Software überwiegend mathematische Probleme gelöst. Heutzutage sind die Aufgaben viel komplexer, nicht im Bezug auf mathematische Probleme sondern vielmehr im Bezug auf der Menge der Daten [Kuh15, S. 18]. Zum Beispiel muss der Kurznachrichten Dienst Twitter täglich über 500 Millionen Tweets[1] verarbeiten und über 300 Millionen Nutzern[2] auf der ganzen Welt zur Verfügung stellen. Man spricht hier von dem sogenannten *Web-scale*. Das Internet ermöglicht es global erreichbare Software System bereitzustellen. Dies führt dazu, dass diese Systeme theoretisch mit ca. 3 Milliarden Nutzern[3] zurechtkommen müssen und aufgrund der Zeitzonen keine Wartungszeiträume existieren. Die Architekturansätze solch skalierbarer Systeme halten auch Einzug in Enterprise Applikationen, da diese mit immer größer werdenden Datenmengen zurechtkommen müssen.

[1] Twitter. Anzahl der täglichen Tweets auf Twitter vom Februar 2010 bis Oktober 2013 (in Millionen). http://de.statista.com/statistik/daten/studie/237226/umfrage/wachstum-von-twitter-nach-anzahl-der-taeglichen-tweets/ (zugegriffen am 09. Januar 2016).

[2] Twitter. Anzahl der monatlich aktiven Nutzer von Twitter weltweit vom 1. Quartal 2010 bis zum 3. Quartal 2015 (in Millionen). http://de.statista.com/statistik/daten/studie/232401/umfrage/monatlich-aktive-nutzer-von-twitter-weltweit-zeitreihe/ (zugegriffen am 09. Januar 2016).

[3] Internet Live Stats. Anzahl der Internetnutzer weltweit von 1997 bis 2014 (in Millionen). http://de.statista.com/statistik/daten/studie/186370/umfrage/anzahl-der-internetnutzer-weltweit-zeitreihe/ (zugegriffen am 16. Februar 2016).

Neben der Skalierbarkeit unterliegt auch die Antwortzeit dem Wandel der Anforderungen. Der Benutzer von heute ist es nicht mehr gewohnt lange auf Ergebnisse zu warten. Laut einer Studie von 2011 würden 16 % der Befragten eine aufgerufene Webseite wieder verlassen, wenn die Ladezeit mehr als drei Sekunden beträgt. Weitere 10 % bei einer Ladezeit mehr als vier Sekunden[4]. Man kann davon ausgehen, dass Nutzer heute weitaus kürzere Ladezeit erwarten und fordern.

Eine weitere Entwicklung ist bei den Prozessoren zu beobachten. Die Prozessorhersteller sind bei den Frequenzen bzw. Taktraten an Grenzen gestoßen, weshalb man seit einigen Jahren auf Multi-core Architekturen setzt. Für Software Entwickler bedeutet das, um Anwendungen schneller zu machen, müssen diese auf die Multicore Prozessoren hin optimiert werden [But14, S. 15]. Anwendungen müssen ihre Aufgaben und Teilaufgaben nebenläufig und parallel ausführen, um die Prozessoren optimal auszulasten. Man ist auf einmal mit ähnlichen Problemen konfrontiert wie bei verteilten Systemen, da man auch bei lokaler Interprozesskommunikation mit Latenzen rechnen muss.

[4]Wie lange sind Sie bereit zu warten bis eine mobile Website auf Ihrem Smartphone vollständig geladen hat?. http://de.statista.com/statistik/daten/studie/202650/umfrage/wartezeit-bis-zum-verlassen-einer-mobilen-website/ (zugegriffen am 09. Januar 2016).

1.2 Reaktives Manifest

Um den erwähnten Anforderungen gerecht zu werden, muss sich die moderne Software Architektur anpassen. Bonér et al. postulieren in dem „Reactive Manifesto“, reaktive Systeme währen skalierbarer, einfacher weiterzuentwickeln und zuverlässiger [BFKT14]. Aufgrund der im Manifest definierten Eigenschaften bzw. Anforderungen sollen reaktive Systeme den heutigen Herausforderungen besser gewachsen sein. Entspricht die Architektur einer Software dem Reactive Manifesto, gewährleistet man ein System welches resilient (dt. widerstandfähig) sowie elastic (dt. elastisch) ist und deshalb auch immer responsive (dt. antwortbereit). Um das zuereichen muss ein System messagen driven (dt. nachrichtenorientiert) sein [Ver16, S. 5] [BFKT14].
Man bezeichnet diese Anwendungen dann als reaktive Anwendungen im Sinne des Manifests [BFKT14].
Das Reactive Manifesto beschreibt keine neuen Konzepte oder Paradigmen, vielmehr formalisiert es Begriffe und bietet Technologie-übergreifende Definitionen für reaktive Systeme.
Die Arbeit soll die Frage beantworten, welche Konzepte und Design Patterns helfen reaktive Systeme zu entwicklen.

2 Grundlagen und Begriffsbestimmung

In diesem Teil der Arbeit geht es um die Grundlagen und Begriffsbestimmung auf diese dann im Hauptteil aufgebaut werden. Zu Beginn werden die genauen Eigenschaften reaktiver Systeme definiert. Später werden grundlegende Programmierkonzept für reaktive Systeme erklärt.

2.1 Eigenschaften reaktiver Systeme

Reaktive Anwendungen haben die Eigenschaften folgendes zu leisten bzw. foldenge Punkte zu erfüllen [Kuh15, S. 19ff] [Ver16, S. 6].
Eine reaktive Anwendung muss. . .

1. . . . **auf Nutzer oder Komponenten reagieren**. Die Applikation erfüllt die geforderte Antwortzeit und übertrifft diese eventuell sogar.
2. . . . **auf Fehler reagieren**. Die Software ist von Grund auf widerständfähig gegenüber Fehlerzuständen. Die Wiederherstellung des Normalzustand erfolgt automatisch.
3. . . . **auf variable Belastung reagieren**. Das System ist automatisch in der Lage dynamische Skalierung durchzuführen.
4. . . . **auf Nachrichten reagieren**. Das System verwendet asynchrone Nachrichtenübermittlung zwischen den Komponenten und ist somit nachrichtenorientiert.

2.1.1 Elastic

Eine reaktive Anwendung muss auf variable Belastung ebenso variable bzw. dynamisch reagieren können. Wird Beispielsweise ein Online Shop in einer Fernsehwerbung oder von einem bekannten Blog erwähnt, müssen kurzfristig sehr viele Anfragen angemessen und zufriedenstellend verarbeitet werden [Kuh15, S. 39].
Um die gewünschten und geforderten Antwortzeiten einzuhalten, muss das System skalieren. Traditionellerweise ist hiermit „Scaling up“ gemeint. Aufgrund der heutigen Anforderungen stößt man schnell an die Grenzen eines einzelen Hosts. Dementsprechend muss das System nicht nur vertikal sondern auch horizontal skalieren. Dazu muss das System auf mehrere Nodes verteilt werden [Ver16, S. 7]. Es ist deshalb wichtig, die Teilaufgaben eines Systems zu identifizieren und auf einzelne Komponenten aufzuteilen, die dann wie bereits erwähnt auf einzelne Nodes verteilt werden können [Kuh15, S. 40].
Das Reactive Manifesto fordert zudem auch, das System so zu gestalten, dass es „Scaling down“ fähig ist. Das bedeutet ungenutze und nicht benötigte Ressourcen müssen wieder freigegeben werden. Dadurch kann man ein hochskalierbares reaktives System kosteneffizient betreiben. Im Reactive Manifesto hat man sich auf den Begriff *elastic* geeinigt, um deutlich zu machen, dass man in beide Richtungen skalieren kann. Bei reaktiven Applikationen kann spricht man deshalb von elastischer bzw. dymanischer Skalierung [Ver16, S. 8].
Für die Software bedeutet dies, dass einzelne Nodes jederzeit hinzugefügt und entfernt werden können. Als Folge dessen muss das System *location transparent* sein. Insofern dürfen dessen Komponenten und deren Funktionen nicht abhängig von einem Host sein [Ver16, S. 8].

2.1.2 Resilient

Eine Software *resilient* (dt. widerstandsfähig) zu entwickeln bedeutet nicht, dass die Software fehlerfrei ist. Es bedeutet, dass die Software sich von einem Fehlerzustand erholen kann [Ver16, S. 6].
Man versucht bei dem Entwurf der Software Fehler von vornherein zu bedenken und mit ihnen sinnvoll umzugehen. Folgendes Zitat von Jonas Bonér macht deutlich, wie wichtig die Widerstandsfähigkeit einer Software ist.

> Without resilience, nothing else matters. If your beautiful, production-grade, elastic, scalable, highly concurrent, non-blocking, asynchronous, highly responsive and performant application isn't running, then you're back to square one. It starts and ends with resilience. [Bonér, Jonas; 2015]

Im Grunde ist diese Aussage trivial. Eine Software die nicht läuft, ist unbrauchbar — egal wie komplex und durchdacht die Architektur auch sein mag.
Es ist aber nicht nur die eigene Software die betroffen sein kann. Andere externe Softwarekomponenten von denen man abhängt oder auch die Hardware kann im laufenden Betrieb Probleme bereiten [Kuh15, S. 33].
Den Schluss, den man daraus ziehen sollte, lautet deshalb nicht, ob ein Fehler auftritt sondern viel mehr wann und wie häufig das passiert. Für den Benutzer ist es nebensächlich warum ein interner Fehler aufgetreten ist. Die Anwendung wird in diesem Moment nicht das tun, was der Benutzer von ihr erwartet [Kuh15, S. 33].

Im Reactive Manifesto hat man für dieses Problem bzw. die Eigenschaft ganz bewusst den Begriff *resilience* und nicht *reliability* (dt. Ausfallsicherheit) gewählt. Man möchte deutlich machen, dass es nahezu unmöglich ist, ein ausfallsicheres System zu schaffen. Deshalb setzt auf widerstandsfähige Systeme, welche mit Fehlerzustanden umgehen können und vorallem sich von dieses wieder erholen können. Folglich ist ein reaktives System nicht nur *fault tolerant* (dt. Fehler tolerant) sondern kann sich auch von einem Fehlerzustand selbstständig wieder erholen [Kuh15, S. 34].

Um die Eigenschaft *resilience* zu erfüllen, müssen das System in Komponenten aufgeteilt und anschließend verteilen (engl. distribute) werden. Zusätzlich ist es nötwendig die verteilen Komponenten von einander abzuschotten (engl. compartmentalize) [Kuh15, S. 34] [Ver16, S. 7].
Aufgrund der verteilten und abgeschottenten Komponenten können Fehler isoliert werden. Hierfür führt man das Prinzip der *supervisor* ein. Nach- bzw. untergeordnete Komponenten informieren ihren *supervisor* im Falle eines Fehlers. Dieser hat nun die Möglichkeit die Subkomponente z.B. neuzustarten oder eine erneute Anfrage zustellen.

2.1.3 Responsive

Eine reaktive Anwendung muss zu jederzeit auf jede Anfragen reagieren. Das heißt die Anwendung ist jederzeit *responsive* (dt. antwortbereit). Anfragen können nicht nur durch einen Benutzer ausgelöst werden, sondern können auch von anderen Diensten oder Komponenten initiert werden. Als Client einer reaktiven Anwendung muss man sich drauf verlassen können, dass eine Antwort in einem festgelegten Zeitraum eintrifft. Das bedeutet es müssen *timeouts* festgelegt werden, nachdem eine Anfrage für fehlerhaft erklärt wird.
(Außerdem muss eine reaktive Anwendung Anfragen nebenläufig und parallel bearbeiten.) Ist eine Anwendung nicht *elastic* (siehe 2.1.1) und/oder nicht *resilient* (siehe 2.1.2) kann sie auch nicht *responsive* sein. Eine reaktive Anwendung muss unter Last skalieren, um die geforderte maximale Antwortzeit einzuhalten oder zu vermeiden, dass das System gar ganz ausfällt. Fallen Komponenten aus, z.B. durch einen unvorhergesehenen Hardwarefehler, könnten diese aufgrund der *location transparency* auf einem anderen Node neugestartet werden.

2.1.4 Message-driven

Um die bereits erwähnten Eigenschaft *elasticity*, *resilience* sowie die daraus folgende *responsiveness* zu erfüllen, müssen reaktive Anwendung *message-driven* sein.
Ist ein System *message-driven* erfolgt die Kommunikation ausschließlich über asynchronen Nachrichtenaustausch (engl. Message passing) zwischen den Komponenten, wodurch eine strikte Abgrenzung der Komponenten erfolgt. Durch die Abstraktion der Kommunikation wird eine lose Kopplung zwischen den Komponenten sichergestellt. Desweiteren werden die Komponenten von einander isoliert, wodurch es möglich wird Fehler als Nachrichten zu delegieren (siehe 2.1.2). Durch den explizite Nachrichtenaustausch kann über *message queues* und *flow control* die Last verteilt und kontrolliert werden. Auch die gewünschte *location transparency* wird durch die Entkopplung über den asynchronen Nachrichtenaustauch ermöglicht [Bon15b].

2.1.5 Zusammenhang

2.2 Parallelität und Nebenläufigkeit

2.3 Functional programming

2.3.1 First-class functions

2.3.2 Immutable State

2.4 Reactive programming

3 Hauptteil

3.1 Reactive Design Patterns

3.1.1 Actor Model

3.1.2 Reactive Streams

3.1.3 Circuit Breaker

3.2 Implementierung

3.2.1 Akka (Scala)

3.2.2 NodeJS (JavaScript)

3.3 Ergebnisse

3.3.1 Nutzen der Patterns

3.3.2 Einfachheit der Implementierung

3.3.3 Unterstützung durch Frameworks

4 Schluss

4.1 Reaktive Systeme

4.1.1 Microservices

4.1.2 12-Factor Applications

4.2 Zusammenfassung

Literaturverzeichnis

[BFKT14] Bonér, Jonas ; Farley, Dave ; Kuhn, Roland ; Thompson, Martin: *The Reactive Manifesto*. http://www.reactivemanifesto.org. Version: September 2014

[Bla14] Blalock, Micah: *The Virtuous Developers Guide to Reactive Programming*. https://www.credera.com/blog/technology-insights/open-source-technology-insights/virtuous-developers-guide-reactive-programming/. Version: Juli 2014

[Bon15a] Bonér, Jonas: *Reactive Revealed 3/3: Resiliency, Failures vs Errors, Isolation, Delegation and Replication*. https://www.youtube.com/watch?v=JvbUF33sKf8. Version: 2015

[Bon15b] Bonér, Jonas: *Reactive Trends on the JVM*. https://dzone.com/articles/reactive-trends-on-the-jvm. Version: Oktober 2015

[But14] Butcher, Paul: *Seven concurrency models in seven weeks: when threads unravel*. Dallas, Texas : Pragmatic Bookshelf, 2014 (The pragmatic programmers). – ISBN 978–1–937785–65–9

[Car14] Carkci, Matt: *Dataflow and reactive programming systems: a practical guide*. CreateSpace Independent Publishing Platform, 2014. – ISBN 978–1–4974–2244–5

[HP85] Harel, D. ; Pnueli, A.: On the development of reactive systems. Version: 1985. http://dl.acm.org/citation.cfm?id=101969.101990.

In: Apt, Krzysztof R. (Hrsg.): *Logics and Models of Concurrent Systems*. New York, NY, USA : Springer-Verlag New York, Inc., 1985. – ISBN 0–387–15181–8, 477–498

[Kla15] Klang, Viktor: *Reactive Revealed 2/3: Elasticity, Scalability and Location Transparency in Reactive systems*. `https://www.youtube.com/watch?v=ryIAibBibQI`. Version: 2015

[Kuh14] Kuhn, Roland: *33rd Degree 2014 - Go Reactive: Blueprint for Future Applications - Roland Kuhn*. `https://www.youtube.com/watch?v=IGW5VcnJLuU`. Version: 2014

[Kuh15] Kuhn, Roland: *Reactive design patterns*. O'Reilly Media, 2015. – ISBN 978–1–61729–180–7

[Mal15] Malawski, Konrad: *Reactive Revealed 1/3: Async NIO, Back-pressure and Message-driven vs Event-driven*. `https://www.youtube.com/watch?v=fNEZtx1VVAk`. Version: 2015

[Ver16] Vernon, Vaughn: *Reactive messaging patterns with Actor model: application and integration patterns in Scala and Akka*. New York : Addison-Wesley, 2016. – ISBN 978–0–13–384683–6

[Wam14] Wampler, Dean: *React 2014 : Dean Wampler - Reactive Design & Language Paradigms*. `https://www.youtube.com/watch?v=4L3cYhfSUZs`. Version: 2014

[Wie03] Wieringa, Roel: *Design Methods for Reactive Systems (Slides)*. `http://booksite.elsevier.com/9781558607552/slides/slides.pdf`. Version: 2003

Abbildungsverzeichnis

Tabellenverzeichnis