

Masterarbeit

KONZEPT UND PROTOTYPISCHE IMPLEMENTIERUNG VON SELF-HEALING UND FEHLERTOLERANTEN SOFTWARESYSTEMEN UNTER VERWENDUNG DES ACTOR MODELS

Aidar Aibekov

geboren am 08. Juni 1998 in Bischkek, Kirgisistan

Studiengang Informatik

Westsächsische Hochschule Zwickau
Fakultät Physikalische Technik / Informatik
Fachgruppe Informatik

Betreuer, Einrichtung: Prof. Dr. Frank Grimm
Westsächsische Hochschule Zwickau

Abgabetermin: 4. März 2022

Danksagungen

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben. Zuerst gebührt mein Dank Prof. Dr. Frank Grimm, die meine Masterarbeit betreut und begutachtet hat. Für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.

Ich möchte mich auch bei meinen Kommilitonen bedanken, die mich nicht nur bei der Erstellung der Masterarbeit, sondern auch bei den Lehrveranstaltungen und Prüfungen unterstützt haben. Ich möchte mich bei ihnen für die vielen interessanten Diskussionen und Ideen bedanken, die wesentlich zu meinem Studium beigetragen haben.

Abschließend möchte ich mich bei meiner Familie und meinen Freunden bedanken, die mein Studium durch ihre Unterstützung ermöglicht haben und immer ein offenes Ohr für mich hatten.

Vielen Dank!

Eigenständigkeitserklärung

Hiermit versichere ich, Aidar Aibekov, dass ich die vorliegende Masterarbeit mit dem Titel *“Konzept und prototypische Implementierung von Self-healing und fehlertoleranten Softwaresystemen unter Verwendung des Actor Models”* selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe. Diese Arbeit lag in gleicher oder ähnlicher Weise noch keiner Prüfungsbehörde vor und wurde bisher noch nicht veröffentlicht.

Zwickau, den 4. März 2022

Ort, Datum



Unterschrift

Inhaltsverzeichnis

Danksagungen	i
Eigenständigkeitserklärung	ii
Inhaltsverzeichnis	iii
Abbildungsverzeichnis	v
Auflistungsverzeichnis	vi
Abkürzungsverzeichnis	vii
1 Einleitung	1
1.1 Motivation	1
1.2 Zielsetzung und Aufgabenstellung	2
1.3 Aufbau der Arbeit	3
2 Theoretische Grundlagen	4
2.1 Grundlagen von Selbstheilung und Fehlertoleranz	4
2.1.1 Selbstheilende Systeme	5
2.1.2 Fehlertoleranz	6
2.1.3 Ausfall, Fehler, Störung	6
2.1.4 Ist-Zustand von Selbstheilung und Fehlertoleranz in der OOP	8
2.1.5 Fehlerbaumanalyse	9
2.2 Reaktive Anwendungen	12
2.2.1 Fehlertoleranz in reaktiven Systemen	14
2.2.2 Supervision in reaktiven Systemen	14
2.2.3 Patterns für die Fehlerbehandlung	17
2.3 Actor Model	19
2.3.1 Actors	19
2.3.2 Konzepte reaktiver Systeme in Bezug auf das Actor Model	24
2.3.3 Vergleich von Actor Model und objektorientierte Programmierung	24
2.3.4 Selbstheilung und Fehlertoleranz: Actor Supervision	26
2.3.5 Frameworks für Actor Model	31
3 Umsetzung	32

3.1	Konzept	32
3.1.1	Beispielprojekt	34
3.1.2	Systemanforderungen für die Umsetzung	35
3.2	Prototypische Implementierung	36
3.2.1	Geschäftsprozessablauf	36
3.2.2	Implementierung von Actors	39
3.2.3	Implementierung von Actor Supervision	43
3.2.4	Geschäftsprozess-Demonstration	52
3.2.5	Beispiel von Fehlerbaumanalyse in ATM-Software	57
4	Auswertung	60
4.1	Erfüllung von Systemanforderungen	60
4.1.1	Funktionale Anforderungen	60
4.1.2	Nicht funktionale Anforderungen	62
5	Zusammenfassung	63
	Literatur	64

Abbildungsverzeichnis

2.1	<i>Störung, Fehler, Ausfall</i>	8
2.2	<i>Fehlerbaum: Beispieldiagramm</i>	11
2.3	<i>Reaktive Systeme: Hauptprinzipien [Bon+14]</i>	12
2.4	<i>Eskalation von Exceptions in einer blockierenden Architektur [XOO21]</i>	15
2.5	<i>Zuordnung von Anfragen und Exceptions/Störungen/Fehler [KHA17]</i>	16
2.6	<i>Hierarchie der Supervisors</i>	17
2.7	<i>Beziehungen zwischen Actors und Adressen</i>	21
2.8	<i>Kommunikation zwischen Actors</i>	22
2.9	<i>Nachrichtenübermittlung zwischen zwei Actors</i>	23
2.10	<i>Blockierender Methodenaufruf [XOO21]</i>	25
2.11	<i>Nicht blockierende asynchrone Kommunikation [XOO21]</i>	25
2.12	<i>Supervisor mit seinen unterstellten Actors</i>	27
2.13	<i>Ein Beispiel für eine Hierarchie von Actors</i>	28
2.14	<i>Supervisor-Strategien: One-For-One und All-For-One</i>	29
2.15	<i>Das “Circuit Breaker”-Pattern [McK17]</i>	30
3.1	<i>Konzept von selbstheilendes und fehlertolerantes System</i>	33
3.2	<i>Aktivitätsdiagramm die eine Funktion von ATM-Software darstellt</i>	38
3.3	<i>Irgendein Actor sendet eine Nachricht an CardActor</i>	40
3.4	<i>Nachrichtenfluss zwischen AtmActor, CardReaderActor, CardActor</i>	42
3.5	<i>Supervisor-Hierarchie in VLINGO-XOOM Framework [XOO21]</i>	43
3.6	<i>MainSupervisorActor mit andere Actors</i>	44
3.7	<i>One-For-One-Startegie mit Actor-Instanzen</i>	45
3.8	<i>Fehlerbaumanalyse des ATM-Softwares</i>	58

Auflistungsverzeichnis

2.1	Exception Handling in Java	15
3.1	Abhängigkeiten in <i>build.gradle</i>	36
3.2	<i>Card</i> -Schnittstelle	40
3.3	Implementierung von <i>CardActor</i>	41
3.4	Konfiguration für <i>MainSupervisorActor</i>	44
3.5	<i>Supervisor</i> -Schnittstelle	46
3.6	Erstellen von <i>SupervisionStrategy</i>	47
3.7	<i>inform()</i> Methode in <i>MainSupervisorActor</i>	47
3.8	Implementierung von <i>AtmActor</i>	48
3.9	Implementierung von <i>CardReaderActor</i>	51
3.10	Erstellung von <i>World</i>	52
3.11	Prozess ohne <i>NullPointerException</i> Auslösung	53
3.12	Konsolenausgabe von Test <i>checkBankNameTest-withoutException</i>	53
3.13	Prozess mit <i>NullPointerException</i> Auslösung	54
3.14	Konsolenausgabe von Test <i>checkBankNameTest-withException</i>	55

Abkürzungsverzeichnis

SH	Self-Healing
FBA	Fehlerbaumanalyse
FB	Fehlerbaum
FIFO	First In First Out
OOP	Object-Oriented Programming
JVM	Java Virtual Machine
ATM	Automated Teller Machine
PIN	Personal Identification Number
RAM	Random Access Memory

Kapitel 1

Einleitung

1.1 Motivation

Jede Software, die erstellt wird, kann nicht perfekt sein. Es kann eine perfekte Architektur, eine perfekte Auswahl an Werkzeugen und geeignete Entwicklungsmethoden haben, aber früher oder später ist das System dazu bestimmt, auszufallen. Das System wird ausfallen, nicht funktionieren oder in irgendeiner Weise fehlerhaft sein, und die Entwickler, die dafür verantwortlich sind, müssen etwas dagegen tun. [KHA17]

Unternehmen sind zwangsläufig mit Systemausfällen konfrontiert. Unternehmenskritische Anwendungen können fehlerhaft funktionieren oder ganz ausfallen. Dies führt zu eingeschränkter Produktivität, Datenverlusten und verlorenen Geschäftsmöglichkeiten. Die Wiederherstellung nach einem größeren Systemausfall kann außerdem viel Geld kosten, manchmal mehr als die Kosten für das System selbst. [Van19]

Indirekte Verluste, die durch die Nichtverfügbarkeit des Systems verursacht werden, führen auch zum Verlust von zuvor loyalen Kunden an direkte Konkurrenten und können die Marktreputation des Unternehmens schädigen. Am Ende kann dies sogar den Unternehmenswert schmälern und zum Niedergang des Unternehmens beitragen. [Van19]

Wie Murphys erstes Gesetz besagt: [Tea]

“Alles, was schief gehen kann, wird schief gehen.”

Die Lösung des Problems kann in der Entwicklung von selbstheilenden und fehlertoleranten Systemen und Anwendungen liegen. Ein selbstheilendes System kann Fehler in seinem Betrieb erkennen und ohne Eingreifen des Entwicklers bestimmte Maßnahmen ergreifen, wodurch es sich selbst wieder in einen Zustand besserer Funktionalität versetzt. [Ehr+10] [Jia+07]

Warum sollte ein System fehlertolerant sein, und warum sollte der Code mit dem Konzept des Fehlers im Hinterkopf geschrieben werden? Wie in “Akka in Action” geschrieben,

in einer idealen Welt ist das System immer verfügbar und kann garantieren, dass jede Aktion erfolgreich sein wird. Es gibt nur zwei Möglichkeiten, dieses Ideal zu erreichen: die Verwendung von Komponenten, die niemals ausfallen können, oder die Berücksichtigung aller möglichen Ausfälle durch Wiederherstellungsmaßnahmen, die auch den Erfolg garantieren.[RBW15] In der Realität wird das System jedoch einfach unterbrochen, sobald ein Fehler auftritt. Beispielsweise ist die Fehlerbehandlung in objektorientierten Programmiersprachen zu einem integralen Bestandteil dieser Sprachen geworden. [Eck05] Die Fehlerbehandlung beschränkt sich jedoch darauf, den Fehler zu ignorieren und die Entwickler über den Fehler zu informieren. Bei einer reinen Fehlerbehandlung kann das System nicht fehlertolerant sein, da es nicht so funktioniert, wie der Benutzer es erwartet.

Die Idee eines fehlerfreien Systems klingt in der Theorie großartig, aber die Schaffung eines solchen Systems ist eine Herausforderung, und in komplexen Systemen ist es überhaupt nicht möglich. Die Schaffung eines hochverfügbaren und gleichzeitig verteilten Systems ist unmöglich, da jede Komponente ausfallen kann. Ein Beispiel für eine solche Komponente ist ein Netz: Es kann jederzeit ausfallen oder nur teilweise verfügbar sein. Das System kann auch von Drittanbieterdiensten abhängen, die sich möglicherweise nicht korrekt verhalten, nicht richtig funktionieren oder einfach nur sporadisch nicht verfügbar sind. Die Server, auf denen die Software läuft, können ausfallen, nicht mehr verfügbar sein oder sogar ganz ausfallen. Aus diesem Grund gibt es ein “Let-It-Crash”-Pattern, das eine Reihe von Aktionen anbietet, wenn eine Komponente ausfällt. Aber natürlich ist dieses Pattern kein Patentrezept. [RBW15]

1.2 Zielsetzung und Aufgabenstellung

Es gibt einen Ansatz zur Entwicklung von Anwendungen, der reaktiv genannt wird. Eines der Kriterien reaktiver Systeme ist Resilienz. Es bedeutet, dass die Anwendung im Falle eines Fehlers reaktionsfähig bleibt. Die Anwendung kann vorher festgelegte Maßnahmen ergreifen, um wieder in einen “gesunden” Zustand zu kommen. [Bon+14]

Daraus ergibt sich das Hauptziel der Arbeit, Methoden zu finden, um eine Anwendung zu entwickeln, die die Anforderungen an Selbstheilung und Fehlertoleranz erfüllt.

Es gibt drei Ebenen von selbstheilenden Systemen: [Ray20]

- Anwendungsebene
- Systemebene
- Hardware-Ebene

Diese Arbeit konzentriert sich auf die Selbstheilung auf Anwendungsebene, die Designelemente zur Lösung von Problemen beinhaltet. Zum Beispiel ordnen Anwendungen, die das

Akka Framework¹ verwenden, Elemente in einer Hierarchie an und weisen die Probleme eines Actors seinem Supervisor zu. Viele Frameworks, die Actor Model implementieren, ermöglichen Anwendungen, die sich durch ihr Design selbst heilen. [Ver15] [KHA17]

Wie bereits erwähnt, arbeitet das Akka Framework mit Actors. Das Actor Model ist ein konzeptionelles Modell zum Umgang mit nebenläufigen Berechnungen. Es definiert einige allgemeine Regeln dafür, wie sich die Komponenten des Systems verhalten und miteinander interagieren sollen. [DWH18]

Im Gegensatz zur Selbstheilung auf Anwendungsebene ist die Selbstheilung auf Systemebene nicht von einer Programmiersprache oder bestimmten Komponenten abhängig. Vielmehr lässt sie sich verallgemeinern und auf alle Dienste und Anwendungen anwenden, unabhängig von deren internen Komponenten. [Ray20]

Zu den häufigsten Fehlern auf Systemebene gehören Prozessausfälle (die oft durch eine Neuverteilung oder einen Neustart behoben werden) und Probleme mit der Antwortzeit (die oft durch Skalierung und De-Skalierung behoben werden). Selbstheilende Systeme führen Gesundheitsprüfungen für verschiedene Komponenten durch und versuchen automatisch, Korrekturen (z. B. Neuverteilung) vorzunehmen, um den gewünschten Zustand wiederherzustellen. [Ray20]

1.3 Aufbau der Arbeit

In Kapitel 2 werden die Begriffe, der theoretische Grundlagen behandelt, damit der Leser das nötige Wissen für das nächste Kapitel hat und versteht, was ihm bei der Arbeit begegnen wird.

In Kapitel 3 werden die Erkenntnisse aus Kapitel 2 genutzt, um das Beispielprojekt zu konzipieren und zu entwickeln. Es werden die Anforderungen des Beispielprojekts und die Vorgehensweise zur Erreichung des Ziels der Arbeit beschrieben.

In Kapitel 4 werden die Systemanforderungen beschrieben, die auf den Ergebnissen von Kapitel 3 basieren. Das Kapitel beantwortet die Frage: Sind die in Kapitel 3 beschriebenen Anforderungen erfüllt?

Die Arbeit endet in Kapitel 5 mit einer Zusammenfassung.

¹Offizielle Webseite von Akka Framework: <https://akka.io/>

Kapitel 2

Theoretische Grundlagen

Dieses Kapitel ist in mehrere Unterkapitel unterteilt, die den allgemeinen Ist-Zustand bei selbstheilenden und fehlertoleranten Systemen beschreiben. Welche Techniken und Verfahren es bereits gibt und wie sie in der Industrie eingeführt wurden. Wichtige Begriffe wie Selbstheilung und Fehlertoleranz werden definiert. Außerdem werden die Arten von Softwarefehlern beschrieben, das Konzept reaktiver Systeme erläutert, das Konzept des Actor Model beschrieben und Beispiele für seine Anwendung gegeben. Schließlich werden die Vor- und Nachteile dieser Konzepte im Rahmen der Entwicklung selbstheilender und fehlertoleranter Systeme erläutert.

2.1 Grundlagen von Selbstheilung und Fehlertoleranz

Seit den Anfängen der Softwareentwicklung als eigenständige Branche gab es immer Raum für Fehler. Egal, wer die Software geschrieben hat, egal, wie perfekt die Architektur ist, sie wird immer Fehler produzieren oder enthalten. Software ist wie der Mensch anfällig für Fehler, und das ist in Ordnung. Jede Software wird ausfallen. Es wird immer eine Exception geben, die vergessen wurde zu behandeln, oder eine Schleife im Code wird eines der Ergebnisse nicht berücksichtigen. Dies sind alles interne Software-Probleme, die auf irgendeine Weise gelöst werden müssen. Für den Endbenutzer ergeben all diese Fehler keinen Sinn. Das Einzige, was er sehen wird, ist, dass die erwartete Antwort ausbleibt. Für den Nutzer bedeutet dies, dass die Software defekt ist und er sich nach einem anderen Dienst umsehen wird, der ihm die richtige Antwort liefert. Der fehlerhafte Dienst muss auf irgendeine Weise behoben werden, sei es durch einen Spezialisten, der sich um das Problem kümmert, oder durch die Software selbst, die in der Lage ist, sich selbst zu heilen. [KHA17]

Hier stellt sich die Frage, warum nicht gleich eine Software geschaffen wird, die auf kommende Fehler und Ausfälle vorbereitet ist und von sich aus Maßnahmen ergreift. Software,

die in der Lage ist, ihre Fehler zu erkennen und darauf zu reagieren, wird als selbstheilende Software bezeichnet. [KHA17]

2.1.1 Selbstheilende Systeme

Ein selbstheilendes Softwaresystem ist in der Lage, seine Fehler zu analysieren und geeignete Maßnahmen zu deren Behebung zu ergreifen. Ein selbstheilendes System muss sein erwartetes Verhalten kennen, um zu untersuchen, ob sein tatsächliches Verhalten in Abhängigkeit von der Umgebung von dem erwarteten Verhalten abweicht.

Der wissenschaftliche Artikel “Formal Analysis and Verification of Self-Healing Systems” von Hartmut Ehring et al. beschreibt das Begriff der selbstheilenden Systeme wie folgt: [Ehr+10]

“Selbstheilende (SH-)Systeme zeichnen sich durch eine automatische Erkennung von Systemfehlern und Techniken zur Wiederherstellung aus diesen Situationen aus.”

Selbstheilende Systeme sind ein relativ neues Forschungsgebiet, das sich mit der Selbstheilung und Fehlertoleranz dynamischer Systeme befasst. Der Begriff “Selbstheilung” bezieht sich auf die Fähigkeit eines Softwaresystems, mit Fehlern fertig zu werden. Der Rahmen für autonomes Rechnen stützt sich auf biologische Analogien und beschreibt einen autonomen Computer als einen Computer, der sich selbst verwaltet, so wie ein biologischer Organismus sich selbst verwaltet. Ein selbstheilendes Softwaresystem ist ein System, das in der Lage ist, Fehler in den von ihm bereitgestellten Diensten zu erkennen, zu diagnostizieren und zu reparieren (oder zumindest abzumildern). In großen Systemen können viele verschiedene Arten von Fehlern auftreten, und je nach Art des Auftretens gibt es auch viele verschiedene Methoden, um sie zu erkennen. Und die Behebung solcher Fehler ist eine separate Tätigkeit, die viel Mühe und Übung erfordert. [Jia+07]

Folglich muss selbstheilende Software für große Systeme auch in der Lage sein, mehrere Arten von Erkennungs-, Diagnose- und Reparaturmechanismen zu nutzen. Autonome Systeme erweitern den oben erwähnten Begriff der Selbstheilung um die Fähigkeit, sich an Veränderungen in der Umgebung anzupassen, um beispielsweise ihre Leistung und die Verfügbarkeit von Ressourcen aufrechtzuerhalten. [Jia+07]

Bei der Fehlertoleranz für robuste Datenverarbeitung geht es um die Bereitstellung bestimmter Dienste durch sorgfältiges Design, während die Selbstheilung Probleme zur Laufzeit behandelt.

In den meisten Fällen sind Begriffe, die mit dem Ausdruck “Selbst” beginnen, ein Aspekt des Funktionsumfangs eines Softwaresystems. Selbstschutz, Selbstkonfiguration, Selbstheilung und Selbstoptimierung sind ebenfalls in diesem Paket enthalten. [MA08]

2.1.2 Fehlertoleranz

Die Selbstwiederherstellung von Systemen unterscheidet sich vom Begriff der Fehlertoleranz. Fehlertoleranz ist die Eigenschaft eines Software, auch bei Vorliegen von Fehlern korrekt zu arbeiten. Ein fehlertolerantes System ist auf diese Fehler vorbereitet. Ein solches System verursacht nur minimale Schäden, selbst wenn zur Laufzeit Fehler auftreten. [Han07]

Fehlertoleranz kann auf verschiedene Weise erreicht werden. In der Regel wird das System nicht durch das Hinzufügen von mehr Code verkompliziert. Je mehr Code vorhanden ist, desto wahrscheinlicher werden Fehler auftreten. Dieses Prinzip heißt “Keep It Simple”. [Han07]

2.1.3 Ausfall, Fehler, Störung

Störungen in der Software sind auf das Vorhandensein von Fehlern in der Software zurückzuführen. Das Vorhandensein von Softwarefehlern führt jedoch nicht immer zu Systemausfällen. Systemausfälle sind sehr kritisch und kostspielig zu beheben, da sie manchmal auch die Software betreffen und meist die Hardware vollständig beschädigen. Nicht jeder Softwarefehler führt zu solchen Situationen, aber viele. Wenn außerdem die Anforderungen der Benutzer, die die Software erfüllen sollte, nicht erfüllt werden, wird dies ebenfalls als Softwarefehler betrachtet.

Das Buch “Patterns for Fault Tolerance” [Han07] enthält eine kurze Definition dieser Konzepte auf der Grundlage früher wissenschaftlicher Arbeiten:

“Ein Systemausfall liegt vor, wenn die erbrachte Leistung nicht mehr mit der Spezifikation übereinstimmt, wobei letztere eine vereinbarte Beschreibung der erwarteten Funktion und/oder Leistung des Systems ist. Ein Fehler ist derjenige Teil des Systemzustands, der zu einem späteren Ausfall führen kann; ein Fehler, der die Leistung beeinträchtigt, ist ein Hinweis darauf, dass ein Ausfall auftritt oder aufgetreten ist. Die angenommene oder vermutete Ursache eines Fehlers ist eine Störung.”

Die grundlegenden Begriffe, die es zu unterscheiden gilt, sind Ausfall, Fehler und Störung.

Ausfall

Ein Ausfall ist ein Ereignis, das eintritt, wenn ein Service von der korrekten Wartung abweicht. Und die Spezifikation selbst, die die korrekte Wartung des Systems beschreibt, muss streng dokumentiert werden. Ohne diese Dokumentation ist es unmöglich nachzuvollziehen, ob der Programmablauf von der Planung abgewichen ist und somit ein Systemausfall eingetreten ist. [Han07]

Jedes Mal, wenn ein System etwas tut, das für den Benutzer oder den Systementwickler unerwartet ist, wird es versagen. Daher ist es sehr wichtig, die korrekte Funktionsweise

des Systems selbst zu beschreiben, bevor das System in Betrieb genommen wird, damit der tatsächliche Zustand mit dem idealen Zustand verglichen werden kann. [Han07]

Fehler

Ein Fehler ist eine Abweichung zwischen dem richtigen Zustand eines Services und dem falschen Zustand. Fehler können zu einem Ausfall führen. [Han07]

Das Buch “Patterns for Fault Tolerance” [Han07] enthält Beispiele für häufige Fehler, wie z. B:

- Timing oder “Race Condition”
- Endlosschleifen
- Protokollfehler: Unerwartete Nachrichten, die an andere Teile des Systems gesendet werden, Nachrichten, die zu unpassenden Zeiten oder in falscher Reihenfolge gesendet werden
- Unstimmigkeit der Daten
- Unfähig, Überlastbedingungen zu bewältigen: Das System ist nicht in der Lage, die Arbeitslast zu bewältigen [Han07]

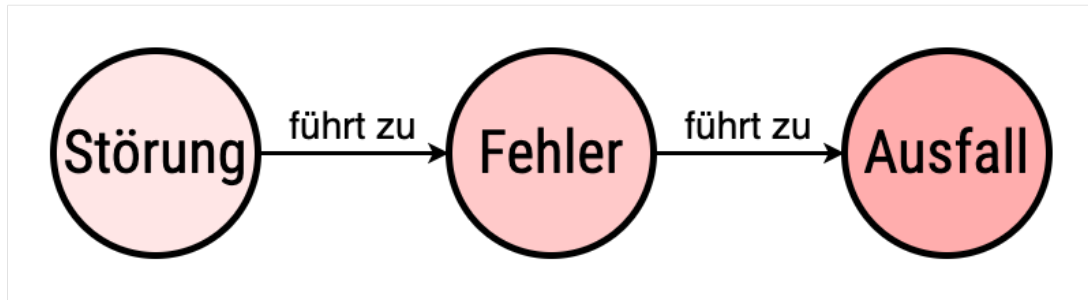
Störung

Störung ist die Ursache für einen Fehler. Zuerst gibt es irgendeine Art von Störung im System, die dann einen Fehler erzeugt, und in der Folge führt der Fehler zu Ausfällen. Bei fehlertoleranten Systemen lohnt es sich, auf der untersten Ebene anzusetzen, d. h. bei Störungen, damit diese nicht zu katastrophalen Folgen führen. [Han07]

Es gibt viele Dinge, die zu einem Fehler führen können. Eine falsch geschriebene Methode, ein vergessener *break*-Operator, fehlende Klammern, wo sie notwendig sind, und so weiter. All diese Kleinigkeiten verhindern, dass das Programm richtig funktioniert. Programmierer sind es gewohnt, das Wort “Bug” umgangssprachlich zu verwenden. [Han07]

Es ist in Ordnung, dass solche Störungen auftreten, weil das System von Menschen geschrieben wird und diese Fehler machen können. Es ist nicht in Ordnung, dass zu viele dieser Ausfälle auftreten. Dies kann verursacht werden durch: [Han07]

- Falsche Spezifikation der Anforderungen
- Falsches Design
- Fehler im Programmcode [Han07]

Abbildung 2.1: *Störung, Fehler, Ausfall*

Fehler und Störungen können ebenfalls in zwei Arten unterteilt werden: Hardware und Software. Im Folgenden wird nur der Softwareteil betrachtet und analysiert. Denn Fehler in der Hardware sind in den meisten Fällen physikalische Fehler. Hardwarekomponenten fallen in der Regel aufgrund von Verschleiß aus. Es kann auch Konstruktionsfehler bei der Hardware geben, aber in der Regel überwiegen physische Fehler. [Han07]

Fehler und Störungen auf der Softwareebene bedingen eine Selbstheilung auf der Anwendungsebene. Einige Softwarefehler, wie z. B. Prozessfehler, Probleme mit der Reaktionszeit oder der Verfügbarkeit von Komponenten, erfordern eine Selbstheilung auf Systemebene. Die Selbstheilung auf Systemebene hängt nicht von einer Programmiersprache, einem Framework oder bestimmten Komponenten ab. Vielmehr kann sie verallgemeinert und auf alle Dienste und Anwendungen angewandt werden, unabhängig von deren internen Komponenten. [Ray20]

Diese Arbeit konzentriert sich auf die Selbstheilung auf der Anwendungsebene, die Gestaltungselemente zur Problemlösung umfasst.

2.1.4 Ist-Zustand von Selbstheilung und Fehlertoleranz in der OOP

In der objektorientierten Programmierung werden Selbstheilung und Fehlertoleranz nur durch die Exception Handling behandelt. [Eck05]

Die Programmiersprache Java verfügt beispielsweise über einen verbesserten Mechanismus zur Fehlerbehebung. Durch die Bereitstellung eines konsistenten Mechanismus für die Behandlung von Exceptions bietet Java den Komponenten eine zuverlässige Möglichkeit, die User über Fehler zu informieren. Dieser Mechanismus wurde in der Erwartung geschaffen, dass große und robuste Programme geschrieben werden können und die Sicherheit besteht, dass das Programm frei von nicht behebbaren Fehlern ist. [Eck05]

Jede von einem Programm erzeugte Exception geht davon aus, dass sie behandelt wird. Die Behandlung von Exceptions in Java wird von einem Exception-Handler übernommen. Es ist wichtig, eine Exception von einem “normalen” Fehler zu unterscheiden. Während ein “normalen” Fehler im aktuellen Kontext behoben werden kann, reichen die Informationen

im aktuellen Kontext nicht aus, um eine Exception zu lösen, und das Programm kann nicht fortfahren. [Eck05]

2.1.5 Fehlerbaumanalyse

Die Fehlerbaumanalyse (FBA) ist eine der bekanntesten Techniken, die in einer Vielzahl von Branchen eingesetzt wird. FBA ist ein grafisches Werkzeug zur Untersuchung von Fehlerursachen auf Systemebene, das die Ausbreitung von Fehlern in einem System simuliert, d.h. wie Komponentenausfälle zu Systemausfällen führen. Aufgrund von Redundanz und Bestandsverwaltung führen nicht alle Komponentenausfälle zu einem Systemausfall. Die FBA prüft, ob das Systemdesign zuverlässig genug ist. Es bietet Methoden und Werkzeuge zur Berechnung einer breiten Palette von Eigenschaften und Metriken. [RS15]

Die FBA kann für alle Arten von Risikobewertungsprozessen auf Systemebene verwendet werden. Der Zweck der FBA besteht darin, die Ursache oder mehrere Ursachen für einen Systemausfall effektiv zu ermitteln und die Risiken zu verringern, bevor er eintritt. Es ist ein effektives Werkzeug für komplexe Systeme, das den logischen Pfad der Problemerkennung visuell darstellt. Darüber hinaus kann mit dieser Analyse die Effizienz des Systems gesteigert werden. [RS15]

Zeichnen von Fehlerbäumen: Ereignisse und Gates

Die Fehlerbaum (FB) ist ein gerichteter azyklischer Graph, der aus zwei Arten von Knoten besteht: Ereignisse und Gates. Ein Ereignis ist ein Vorfall in einem System, in der Regel ein Ausfall eines Teilsystems bis hin zu einer einzelnen Komponente. Es gibt verschiedene Arten von Ereignissen, von denen die drei häufigsten im Folgenden beschrieben werden: [RS15]

- **Top-Ereignis** - Das Ereignis an der Spitze des Baumes, das Gegenstand der gesamten Analyse ist. Dies ist in der Regel das Ereignis, das eine Unterbrechung des gesamten Systems verursacht. Im Diagramm ist er als Rechteck mit einem Pfeil zum Ausgang dargestellt.
- **Basis-Ereignis** - Diejenige, die ganz unten im Baum steht und die Ursache für das oberste Ereignis ist. Das zugrunde liegende Ereignis ist ein endliches Ereignis und hat keinen Ausgang. Sie ist in der Abbildung als Kreis dargestellt.
- **Zwischenereignis** - Ein Ereignis, das zwischen dem Top-Ereignis und dem Basis-Ereignis eintritt. Sie hat sowohl einen Eingang als auch einen Ausgang. Das Zwischenereignis wird durch das Top-Ereignis oder anderen Zwischenereignis ausgelöst und löst schließlich das Basisereignis aus. Im Diagramm wird es als Rechteck mit Pfeilen für Eingang und Ausgang dargestellt. [RS15]

Es gibt auch andere Arten von Ereignissen, die jedoch weniger häufig vorkommen als die oben beschriebenen:

- Unentwickeltes Ereignis
- Transfer in und Transfer out
- Bedingtes Ereignis
- Haus Veranstaltung

Gates stellen dar, wie sich Ausfälle im System ausbreiten, d. h. wie Ausfälle in Teilsystemen zu einem Systemausfall führen können. Jedes Gate hat einen Ausgang und einen oder mehrere Eingänge. Die beiden am häufigsten verwendeten Gates im Fehlerbaum sind UND und ODER. Als Beispiel werden zwei grundlegende Ereignisse betrachtet, die zu einem Top-Ereignis führen können. Wenn das Auftreten eines der beiden Basisereignisse zum Top-Ereignis führt, werden diese Basisereignisse mit einem ODER-Gate verbunden. Andernfalls, wenn beide Basisereignisse eintreten müssen, damit das Top-Ereignis eintreten kann, werden sie mit einem UND-Gate verbunden. [RS15]

Insgesamt gibt es neben OR und AND noch weitere Gates dieser Art:

- Priority AND Gate
- XOR Gate
- Inhibit Gate

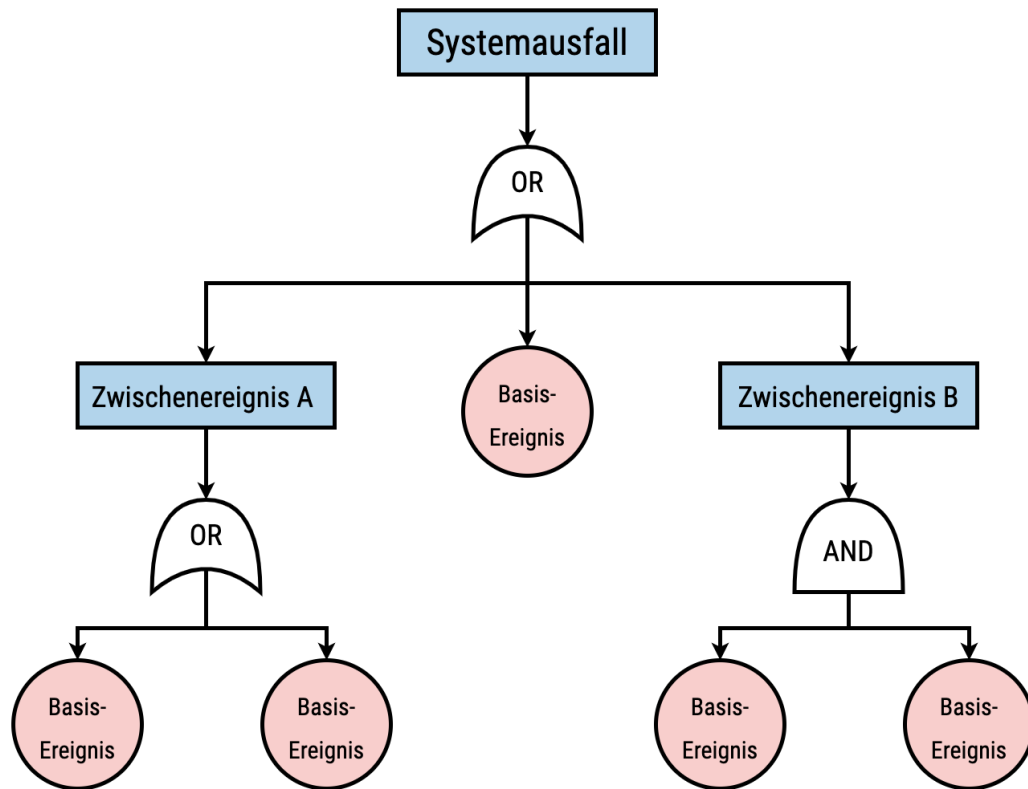


Abbildung 2.2: Fehlerbaum: Beispieldiagramm

Die Abbildung 2.2 zeigt ein Beispiel für einen FB. Es gibt ein Top-Ereignis, was bedeutet, dass etwas schief gelaufen ist. Danach kommt das Tor OR, das drei weitere Wege hat. Entweder endet die FBA sofort mit dem Basisereignis, oder sie führt zu einem der Zwischenereignisse. Wenn das Zwischenereignis A ausgelöst wird, folgt ein weiteres ODER-Gate, das zu einem der Basisereignisse führt. Im Fall von Zwischenereignis B folgt auf das mittlere Ereignis ein UND-Gatter. Dies bedeutet, dass die nächsten beiden Basisereignisse eintreten müssen.

Die Vorteile der Fehlerbaumanalyse:

- Der Fehlerbaum stellt die Analyse visuell dar, um dem Entwickler zu helfen, die Ursache des Ereignisses in einer logischen Weise zu bearbeiten, die zu dem Fehler führt
- Hervorhebung der kritischen Komponenten im Zusammenhang mit Systemausfällen
- Bietet eine effektive Methode zur Analyse des Systems
- Im Gegensatz zu anderen Analysemethoden wird auch das menschliche Versagen in die Analyse einbezogen
- Hilft bei der Priorisierung von Maßnahmen zur Lösung des Problems

- Bietet qualitative und quantitative Analysen [Hes14]

Die Nachteile der Fehlerbaumanalyse:

- Der größte Nachteil ist, dass nur ein Top-Ereignis berücksichtigt wird
- Bei der Analyse großer Systeme sind zu viele Gates und Ereignisse zu berücksichtigen
- Häufige Gründe für Ausfälle sind nicht immer offensichtlich
- Schwierige Berücksichtigung von Zeit und anderen Verzögerungsfaktoren
- Für das Verständnis von Logikgattern werden erfahrene Spezialisten benötigt [Hes14]

2.2 Reaktive Anwendungen

Um ein hohes Niveau an Selbstheilung und Fehlertoleranz zu erreichen, sollten die Systeme reaktiv aufgebaut sein. Die Grundsätze reaktiver Systeme beruhen auf Zuverlässigkeit, Widerstandsfähigkeit und der Minimierung großer Fehler und Ausfälle. [DWH18]

Reaktive Systeme sind ein architektonischer Stil, ein asynchrones Programmierparadigma, das es mehreren separaten Anwendungen ermöglicht, zu einem kohärenten Ganzen zu verschmelzen, auf die Umgebung zu reagieren und sich gegenseitig zu beobachten. Solche Systeme werden als reaktiv bezeichnet, weil sie reaktionsfähig, belastbar und flexibel sind und auf dem Austausch von Nachrichten basieren. Durch die Einhaltung aller oben genannten Grundsätze können Anwendungen flexibel, skalierbar und lose gekoppelt sein. Ein weiterer wichtiger Faktor ist, dass reaktive Anwendungen fehlerresistenter sind und mit Ausnahmesituationen richtig umgehen, um katastrophale Folgen zu vermeiden. [Bon+14]

Reaktive Systeme beruhen auf 4 Hauptprinzipien, die im 2014 veröffentlichten “The Reactive Manifesto” [Bon+14] beschrieben sind. Diese Grundsätze sind:

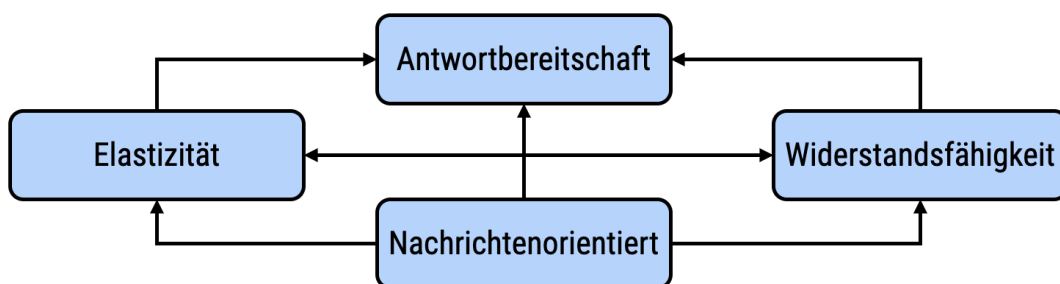


Abbildung 2.3: *Reaktive Systeme: Hauptprinzipien* [Bon+14]

- Antwortbereit - Das System sollte so schnell wie möglich auf Benutzeraktionen und -anfragen reagieren. Das bedeutet auch, dass Probleme schnell erkannt und effizient behoben werden können. Reaktionsfähige Systeme konzentrieren sich auf schnelle und einheitliche Reaktionszeiten, um eine gleichbleibende Servicequalität zu gewährleisten. Jede Software ist bestrebt, reaktionsschnell zu sein und dadurch das

Vertrauen der Endbenutzer zu gewinnen. Bei reaktionsfähigen Systemen müssen die folgenden Grundsätze berücksichtigt werden, um Reaktionsfähigkeit zu erreichen. [Bon+14]

- Widerstandsfähig - Das System bleibt auch bei Störungen reaktionsfähig. Eine der wichtigsten Eigenschaften reaktiver Anwendungen ist die Fehlertoleranz und Fehlerbehandlung. Es gibt viele Möglichkeiten, Anwendungstoleranz zu erreichen, in reaktiven Systemen wird dies durch Replikation, Eingrenzung, Isolierung und Delegation erreicht. Jede Komponente behandelt ihre Ausfälle und Fehler unabhängig und isoliert von anderen Teilen des Systems. Die Wiederherstellung kann auch an eine andere externe Komponente delegiert werden, wodurch das Problem auf einer höheren Ebene gelöst wird. Wenn das Problem nicht gelöst wird oder sich verzögert, kann eine Kopie der Komponente erstellt und dem Benutzer zur Verfügung gestellt werden, damit er nicht zu lange warten muss. [Bon+14]

Die Widerstandsfähigkeit muss auch bei den komplexesten Anwendungen gewährleistet sein. Ohne einen guten Ausfallsicherheitsmechanismus ist selbst das robusteste System von einem Ausfall bedroht. Zu den Aspekten der Ausfallsicherheit gehören Leistung, Ausdauer und Sicherheit. Auch die Isolation spielt eine große Rolle bei der Resilienz. Der Ausfall einer isolierten Komponente beeinträchtigt nicht die Reaktionsfähigkeit des gesamten Systems und gibt der ausgefallenen Komponente die Möglichkeit, sich zu erholen. [Bon+14]

- Elastizität - Das System reagiert auf wechselnde Arbeitsbelastungen. Stabilität und Belastbarkeit müssen zusammen existieren, um ein reibungsloses Funktionieren des Systems zu gewährleisten. Reaktive Systeme müssen in der Lage sein, sich an äußere Umstände und Veränderungen anzupassen, um der auf sie einwirkenden Belastung standhalten zu können. Dies geschieht durch die Erhöhung oder Verringerung der Ressourcen, die zur Bedienung der Inputs bereitgestellt werden. Ein skalierbares System kann bei Bedarf problemlos aufgerüstet werden, um die Reaktionsfähigkeit unter verschiedenen Lastbedingungen zu gewährleisten. Die Lastverteilung kann durch die Replikation von Komponenten und die Verteilung von Eingabedaten zwischen ihnen erreicht werden. [Bon+14]
- Nachrichtenorientiert - Das Herzstück reaktiver Systeme ist die asynchrone Nachrichtenübermittlung zwischen Komponenten, die freie Kommunikation, Isolierung und Standorttransparenz ermöglicht. [Bon+14] Eine nachrichtengesteuerte Anwendung kann ereignisgesteuert, actor-gesteuert oder eine Kombination aus beidem sein. Ein ereignisgesteuertes System basiert auf Ereignissen, die von einem oder mehreren Beobachtern überwacht werden müssen. Dies unterscheidet sich von der imperativen Programmierung dadurch, dass die aufrufende Komponente des Systems nicht

blockiert und auf eine Antwort der aufgerufenen Komponente wartet. Alle Ereignisse haben keinen expliziten Empfänger mit ihrer Adresse, sondern werden von den Komponenten verfolgt, die an dem Ereignis interessiert sind. [DWH18]

Ein Actororientiertes Nachrichtensystem ist eine Art Erweiterung der Nachrichtenarchitektur, bei der Nachrichten an den Empfänger, einen Actor, gesendet werden. Das Actor Model und die Messaging-Architektur werden im nächsten Kapitel ausführlicher beschrieben. [DWH18]

2.2.1 Fehlertoleranz in reaktiven Systemen

In reaktiven Systemen muss auf jede Anfrage innerhalb einer bestimmten Zeit eine Antwort erfolgen. Dies ist wichtig für die Gestaltung reaktionsfähiger und widerstandsfähiger Systeme. Aber selbst bei einer einwandfreien Ausführung eines Entwurfs werden früher oder später unerwartete Dinge passieren. [KHA17]

Kein System ist vor Fehlern gefeit. Software, Hardware und sogar die an der Entwicklung beteiligten Personen können versagen. Die Frage ist also nicht, ob sie scheitern wird, sondern wie und wie oft sie scheitern wird. Ausfälle lassen sich nicht vermeiden, es gilt, mit ihnen fertig zu werden und Fehlertoleranz anzustreben. [KHA17]

Es gibt eine Strategie, um so viele Ausfallszenarien wie möglich zu planen und die Reaktion der Software so anzupassen, dass ein stabiler Systembetrieb wiederhergestellt wird und der Benutzer das System weiterhin nutzen kann. Bei reaktiven Systemen müssen jedoch auch Ausfälle berücksichtigt werden, die zu Beginn nicht vorhergesehen wurden. Das System muss mit jeder Art von Fehler fertig werden, zumindest aber versuchen, ihn zu beseitigen und die ursprüngliche Form wiederherzustellen. [KHA17]

Es gibt viele Praktiken, eine der häufigsten ist die Replikation von Komponenten. Jede kritische Funktion ist mindestens zweimal im System vorhanden. Fällt eine Komponente aus, wird sie durch eine Kopie ersetzt, wodurch der Betrieb des Programms normalisiert wird. Wichtig ist dabei, dass diese Komponenten voneinander isoliert sind, so dass der Ausfall einer Komponente keine Auswirkungen auf ihre Nachbildung hat. [KHA17]

Auch die Wiederherstellung einer ausgefallenen Komponente sollte nicht von der Komponente selbst, sondern von einer anderen Komponente, idealerweise einer höheren Schicht, übernommen werden. Dies wird als Delegation der Einziehungsbefugnis bezeichnet. [KHA17]

2.2.2 Supervision in reaktiven Systemen

Bei der Softwareentwicklung kann und wird es zu Ausnahmesituationen kommen, in denen einem Programm die Informationen fehlen, um weiterzumachen. In höheren Programmiersprachen werden diese Situationen als Exceptions bezeichnet. Der Dienst löst diese Exceptions aus und benachrichtigt den Benutzer. Das Problem ist jedoch, dass der Nutzer keine Ahnung hat, was er mit diesen Informationen anfangen soll. Anstatt den Benutzer

beim Auftreten einer Exception nur zu warnen, muss die Situation korrekt gehandhabt werden. [KHA17]

Beispiel: Es gibt eine in der Programmiersprache Java geschriebene Methode auf der Auflistung 2.1, die theoretisch mehrere Arten von Exceptions auslösen kann. Seien es *ArithmeticException*, *NullPointerException*, *StackOverflowException*. In Java behandelt der Dienst, der diese Methode aufruft, mögliche Ausnahmen selbst.

Auflistung 2.1: Exception Handling in Java

```
try {
    einigeObjekt.einigeMethode();
} catch (ArithmeticException exception) {
    // Exception Handling
} catch (NullPointerException exception) {
    // Exception Handling
} catch (StackOverflowError exception) {
    // Exception Handling
}
```

In einer solchen blockierenden Architektur 2.4 wird die Exception immer weiter nach oben verschoben, bis es den Benutzer erreicht. Es blockiert das gesamte System. [KHA17]

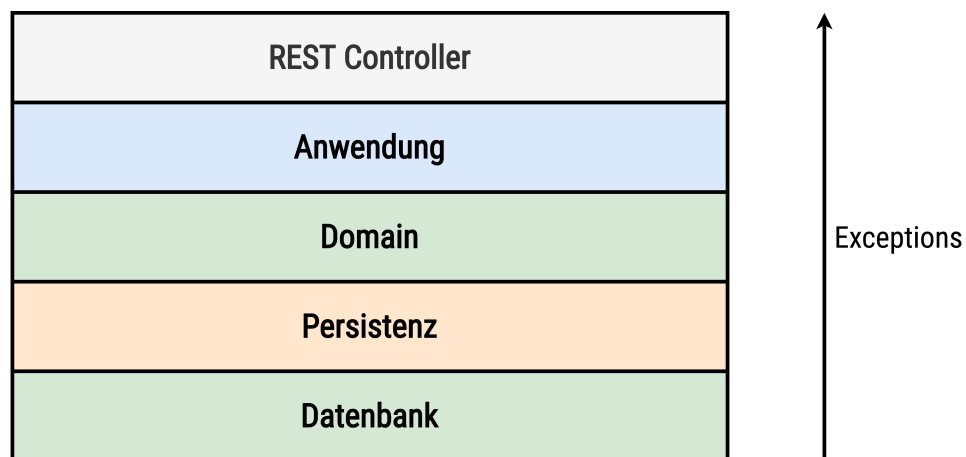


Abbildung 2.4: Eskalation von Exceptions in einer blockierenden Architektur [XOO21]

Die Abbildung 2.4 zeigt, wie Exceptions, die in einem beliebigen Teil der Anwendung aufgetreten sind, bis zum Endbenutzer gelangen. Dies blockiert das gesamte System und macht es funktionsunfähig.

Im Gegensatz zu dieser Architektur steht eine reaktive Architektur, bei der es Services gibt, die andere Services verwalten. In reaktiven Systemen wird diese Beziehung durch

den Begriff Supervisor dargestellt. Die Supervisor ist für die Aufrechterhaltung des Services verantwortlich und verwaltet den gesamten Lebenszyklus des Moduls unter ihrer Kontrolle. [KHA17]

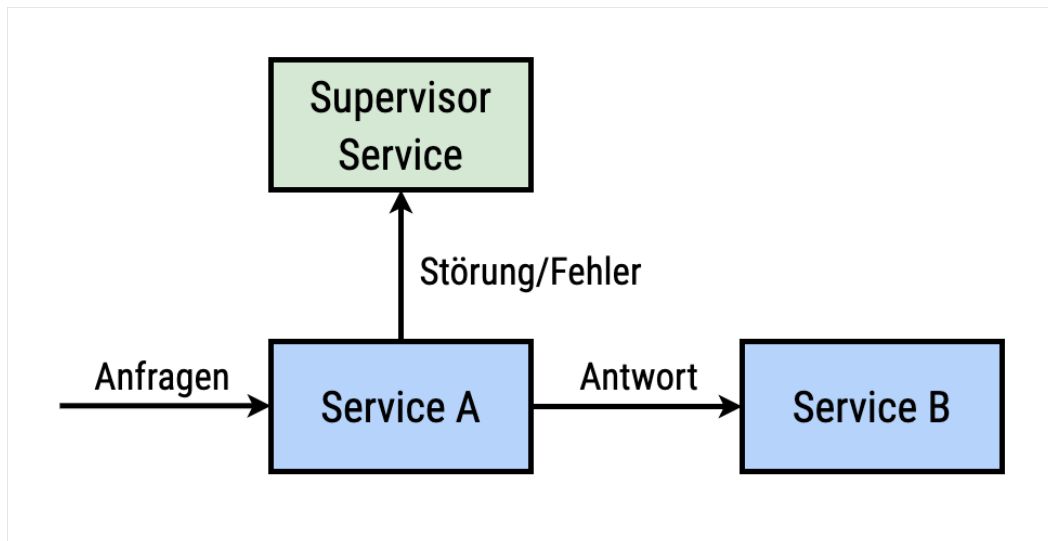


Abbildung 2.5: Zuordnung von Anfragen und Exceptions/Störungen/Fehler [KHA17]

Das Diagramm 2.5 zeigt, dass eine Anfrage, die bei einem bestimmten Service eingeht, auf zwei Arten weitergeleitet werden kann. Entweder bearbeitet der Service die Anfrage erfolgreich und sendet die Antwort an den Benutzer, oder es tritt eine Exception ein und sie wird an den für diesen Service zuständigen Supervisor weitergeleitet. Dadurch werden die Geschäftslogik und die spezielle Ausnahmebehandlung getrennt. Es ist auch möglich, diese Ausnahmebehandlung universell zu gestalten und eine gemeinsame Strategie für mehrere Services zu implementieren. [KHA17]

Die Supervisor hat die volle Befugnis, den Lebenszyklus der von ihr beaufsichtigten Services zu verwalten. Zu seinen Aufgaben gehören: [KHA17]

- Service starten
- Service stoppen
- Service neu starten
- Eskalieren

Eskalieren bedeutet, die Behandlung der Ausnahme an eine höhere Ebene weiterzuleiten, d. h. an den Vorgesetzten, der den Vorgesetzten kontrolliert. [KHA17]

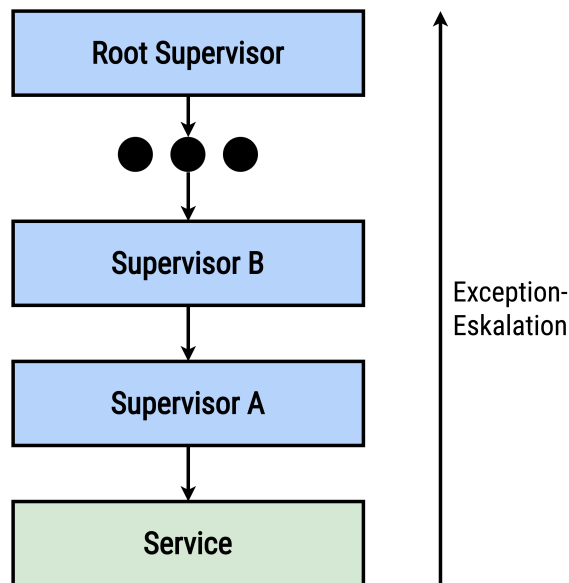


Abbildung 2.6: *Hierarchie der Supervisors*

Wenn der Service eine Exception auslöst, kümmert sich Supervisor A um die Exception. Wenn Supervisor A keine Lösung finden konnte oder nicht genügend Rechte, Befugnisse oder Informationen hatte, um das Problem zu lösen, gibt er das Problem an eine höhere Ebene weiter, z.B. Supervisor B. Wie in Abbildung 2.6 dargestellt, kann die Exception die höchste Ebene erreichen, nämlich den Root Supervisor, der für das gesamte System verantwortlich ist.

Dies führt zu einer hierarchischen Komponentenstruktur, bei der die übergeordnete Komponente für die Leistung ihrer Nachkommen verantwortlich ist. Die Abbildung 2.6 zeigt eine abstrakte Hierarchie der Supervisors. Die konkrete Hierarchie unterscheidet sich von Implementierung zu Implementierung.

2.2.3 Patterns für die Fehlerbehandlung

Die Probleme, die bei der Entwicklung und Implementierung von Software auftreten, existieren nicht in einem Vakuum. Sie wurden bereits erforscht und viele von ihnen wurden bereits gelöst und dokumentiert. Ein Pattern ist eine Dokumentation, eine Beschreibung einer Lösung, die sich in vielen Projekten mehrfach wiederholt. Softwareentwickler passen diese Lösungen an ihre Bedürfnisse, Situationen und Projekte an. [Han07]

“Simple Component”-Pattern

Software besteht aus vielen Komponenten, und es ist die Aufgabe des Entwicklers, diese Komponenten unter Einhaltung des Patterns “Simple Component” zu implementie-

ren. Dies ist das grundlegendste Prinzip bei der Entwicklung fehlertoleranter Systeme. [KHA17]

Das Buch “Reactive Design Patterns” [KHA17] beschreibt den Zweck dieses Patterns:

“Das Ziel des “Simple Component” Patterns ist es, das Prinzip der einzigen Verantwortlichkeit umzusetzen.”

Das bedeutet, dass die Komponenten des Systems so gestaltet sein müssen, dass sie eine bestimmte Aufgabe lösen. Jede Komponente sollte nur eine Aufgabe erfüllen. Große Komponenten sollten in viele kleine Komponenten aufgeteilt werden, um so Einfachheit und schwachen Zusammenhalt zu erreichen. [KHA17]

Dieses Pattern löst das Problem der Systemkomplexität und befreit von der Notwendigkeit, das System als Ganzes zu betrachten und sich auf seine kleineren Komponenten zu konzentrieren. [KHA17]

Um dieses Pattern anwenden zu können, muss das System analysiert und die Komponenten identifiziert werden, die in kleinere Teile aufgeteilt werden sollen. Dieses Pattern wird rekursiv angewendet, bis klar ist, dass alle Komponenten einfach genug sind, um ein einziges Problem zu lösen. [KHA17]

Es lohnt sich auch nicht, Komponenten in einen Zustand zu versetzen, in dem sie zu trivial sind, ohne einen guten Grund für ihre Existenz zu haben. [KHA17]

“Let-It-Crash”-Pattern

Es gibt so viele Arten von Fehlern, von denen einige durch die Ausarbeitung von Strategien für die Wiederherstellung und die Ergreifung geeigneter Maßnahmen bewältigt werden können. Aber es gibt auch solche Fehler, bei denen es nicht klar ist, wie sie zu behandeln sind. [KHA17]

Dies geschieht, weil das System mit Fehlern nicht richtig umgehen kann, weil es nicht weiß, wie es sie behandeln soll. [KHA17]

Das Pattern hilft dabei, sich beim Schreiben von Software auf den “Happy Path” zu konzentrieren. Auch bei diesem Pattern verbringen die Entwicklern mehr Zeit mit dem Schreiben von Code, der etwas bewirkt, als mit Code, der sich auf die Fehlerbehandlung konzentriert. [KHA17]

Um ein Komponenten neu zu starten, muss zunächst festgestellt werden, ob ein Fehler vorliegt. Zu diesem Zweck gibt es zwei Methoden, die mit der allgemeinen Methode “Let-It-Crash” zusammenhängen: [KHA17]

- “The Heartbeat”-Pattern
- “The Proactive Failure Signal”-Pattern

Sie setzen gegensätzliche Strategien zur Erkennung von Fehlern in Komponenten ein. “The Heartbeat“-Pattern stellt den Supervisor vor die Aufgabe, einen Fehler in einer untergeordneten Komponente zu identifizieren. Aber woher weiß der Supervisor, ob es einen Fehler in der ihm unterstellten Komponente gibt? [KHA17]

Zu diesem Zweck sendet der Vorgesetzte gelegentlich Anfragen an die ihm unterstellte Komponente, um sich zu vergewissern, dass sie sich in einem gesunden Zustand befindet. Er kann die benötigten Daten abfragen und sich so vergewissern, dass die Komponente ordnungsgemäß funktioniert. [KHA17]

Das Pattern “The Proactive Failure Signal“ hingegen überträgt dem Komponent selbst die Aufgabe, den Fehler zu identifizieren. Mit anderen Worten, die ausgefallene Komponente diagnostiziert ihren eigenen Ausfall und sendet eine Anfrage an ihren Supervisor über ihren Status. Der Supervisor ergreift dann bestimmte Maßnahmen, um den Fehler zu beheben. [KHA17]

2.3 Actor Model

Das Actor Model ist ein mathematisches Modell des parallelen Rechnens. Im Actor Model ist der Actor die grundlegende Einheit der Berechnung, und alle sind Actors. [DWH18]

Dieses Modell wurde erstmals 1973 eingeführt und sein Autor ist Carl Hewitt. [DWH18] Er ist auch für die Erfindung der Programmiersprache Planner bekannt. Planner hat die Entwicklung sowohl der logischen Programmierung als auch der objektorientierten Programmierung beeinflusst. Seine Veröffentlichungen umfassen auch Beiträge in den Bereichen offene Informationssysteme, Organisations- und Multiagentensysteme, logische Programmierung, parallele Programmiersprachen, direkte Ausgabe, Client Cloud Computing. [Hew22]

Actors sind ein alternativer Ansatz für die Organisation von Parallelrechnern und verteilten Systemen. Das Actor Model ermöglicht die effiziente Nutzung von Ressourcen und die Ausführung von Aufgaben mit einem hohen Grad an Parallelität. Das Konzept selbst ist einfach zu verstehen und ermöglicht so die Entwicklung und Implementierung eleganter, hocheffizienter, reaktiver Anwendungen, die sich im Falle von Fehlern selbst heilen. [DWH18]

2.3.1 Actors

Ein Actor ist eine primitive grundlegende Recheneinheit, die Nachrichten empfängt und auf der Grundlage dieser Nachrichten primitive Berechnungen durchführt. [Ver15] Es gibt nur drei Dinge, die ein Actor macht:

- Verarbeitung von etwas
- Datenspeicherung

- Kommunikation mit anderen Actors [Ver15]

Die Actors sind voneinander isoliert und haben einen internen Zustand, der von außen nicht verändert werden kann. Actors sind einfach, und es ist leicht, Tausende oder sogar Millionen von ihnen zu erstellen, da sie weniger Ressourcen benötigen als Streams. [Ver15]

Wie der Autor des Actor Models selbst, Carl Hewitt, sagte: [jas12]

“Ein Actor ist kein Actor. Actors kommen in Systemen.”

Die Actors sind voneinander isoliert und haben keinen gemeinsamen Speicher. Sie haben einen Zustand, den sie aber nur ändern können, wenn sie eine Nachricht erhalten. [Ver15]

Da reaktive Systeme auf dem Austausch von Nachrichten beruhen, besteht die einzige Form des Informationsaustauschs zwischen den Actors in Nachrichten. Ein Actor empfängt eine Nachricht und tut etwas als Reaktion darauf. Dies kann z. B. die Durchführung von Berechnungen, die Aktualisierung des internen Zustands, das Senden neuer Nachrichten oder das Auslösen einer Ein/Ausgabe sein. [DWH18]

Jeder Actor hat seine eigene Mailbox, die einer Nachrichtenwarteschlange ähnelt. Die Nachrichten werden in den Postfächern der Actors gespeichert, bis sie bearbeitet werden. Sobald die Actors erstellt sind, warten sie auf den Eingang von Nachrichten. Wenn Sie 2 oder mehr Nachrichten an einen Actor senden, verarbeitet dieser sie nacheinander in FIFO-Reihenfolge (first in, first out). Wenn Nachrichten parallel verarbeitet werden sollen, sind mehrere Actors zu erstellen und jedem Actor eine Nachricht zu übergeben. [Ver15]

Nachrichten sind einfache, unveränderliche Datenstrukturen, die leicht über ein Netz versendet werden können. [Ver15]

Konzeptionell kann ein Actor immer nur 1 Nachricht gleichzeitig verarbeiten. Die Actors sind getrennt, sie arbeiten asynchron und müssen nicht auf eine Antwort eines anderen Actors warten. [Ver15]

Wenn ein Actor eine Nachricht erhält, kann er eine der folgenden Aktionen durchführen: [Ver15] [jas12]

- Einen neuen Actor erstellen
- Eine Nachricht an andere Actors senden
- Festlegen, wie die folgende Nachricht zu behandeln ist [Ver15]

Einen neuen Actor erstellen

Jeder Actor hat die Möglichkeit, andere Actors zu schaffen. [Ver15]

Eine Nachricht an andere Actors senden

Actors haben Adressen, so dass ein Actor eine Nachricht an einen anderen Actor senden kann, wenn er dessen Adresse kennt. Ein Actor kann nur mit Actors kommunizieren, deren Adressen er hat. Ein Actor hat die Adressen der Actors, die er selbst erstellt hat, und kann weitere Adressen aus der empfangenen Nachricht erhalten. Ein Actor kann viele Adressen haben. Es ist zu beachten, dass eine Adresse nicht gleichbedeutend mit einer Identität ist, was also nicht bedeutet, dass zwei Actors mit der gleichen Identität die gleiche Adresse haben. Das heißt, mehrere Actors können dieselbe Adresse haben und ein Actor kann mehrere Adressen haben. Hier gibt es eine Analogie zu den relationalen Datenbanken mit Many-To-Many Beziehung zwischen Datentabellen wie zwischen Actors und Adressen. [jas12]

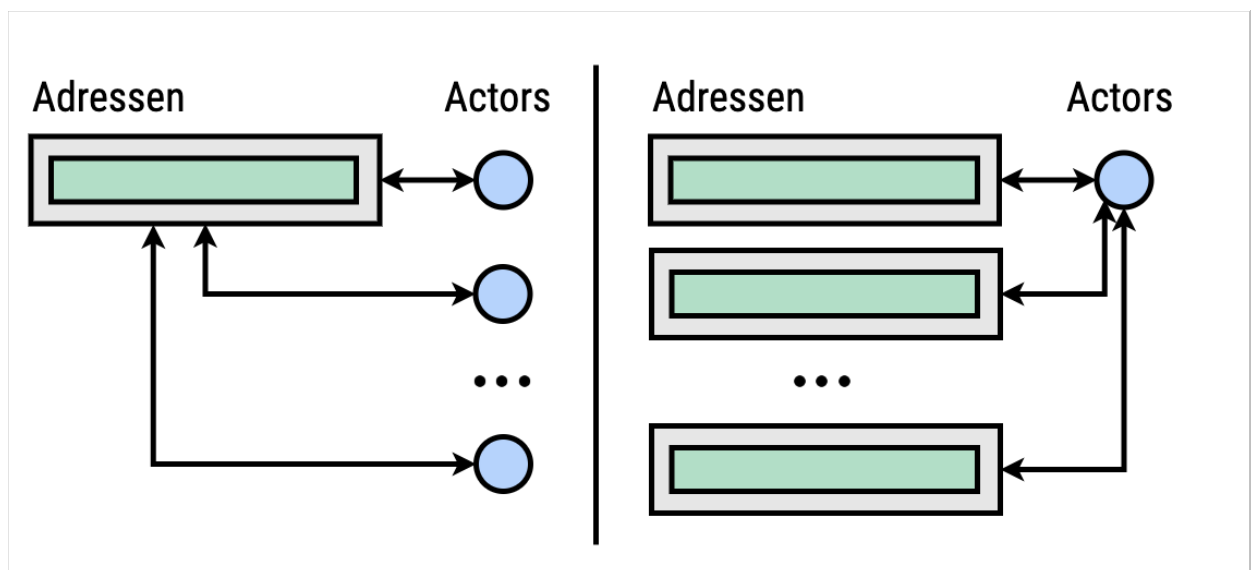


Abbildung 2.7: *Beziehungen zwischen Actors und Adressen*

Wie in der Abbildung 2.7 zu sehen ist, kann ein Actor eine oder mehrere Adressen haben. Eine Adresse kann auch wiederum einem oder mehreren Actors zugeordnet werden.

Die Actors interagieren miteinander, indem sie Nachrichten austauschen. Die Richtungspfeile 2.8 zeigen den Weg der Nachrichtenübertragung an.

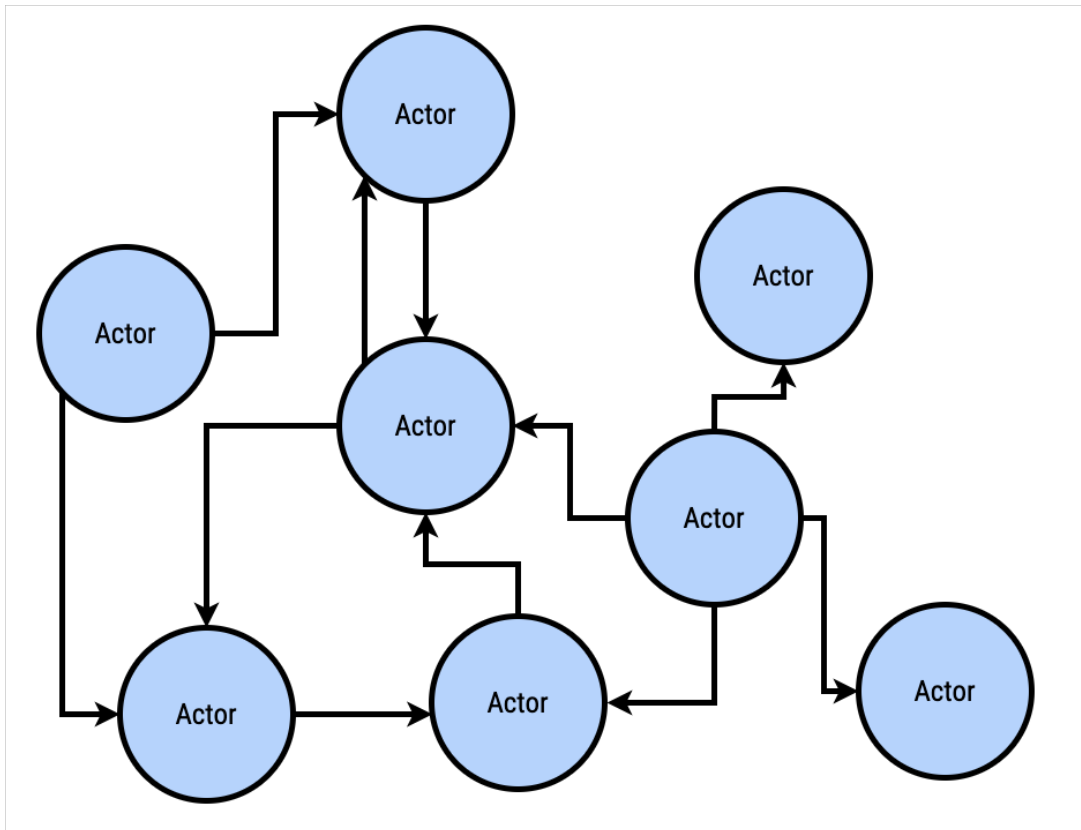


Abbildung 2.8: *Kommunikation zwischen Actors*

Die Actors können lokal oder entfernt auf einem anderen Rechner arbeiten. Dies ist für das System völlig transparent, da die Actors über Adressen kommunizieren, die lokal oder entfernt sein können. [Ver15]

Nachdem ein Actor eine Nachricht gesendet hat, kann er seine Arbeit fortsetzen, ohne blockiert zu werden. Auf diese Weise kann er mehr Aufgaben in der gleichen Zeit erledigen. [Ver15]

Festlegen, wie die folgende Nachricht zu behandeln ist

Schließlich kann ein Actor seinen eigenen Zustand speichern und festlegen, wie der nächste Zustand zu behandeln ist. [jas12] [Ver15] Ein Actor hat beispielsweise ein Feld, in dem der Gesamtsaldo eines Kontos gespeichert wird. Wenn dieser Actor die Nachricht “N-Betrag zum Konto hinzufügen” erhält, aktualisiert er seinen Status, indem er den neuen Betrag zum bereits bestehenden Gesamtsaldo hinzufügt. Das bedeutet, dass der Zustand des Actors bei der nächsten Nachricht anders sein wird.

Verarbeitung von Nachrichten

Wie interagieren die Actors, indem sie sich gegenseitig Nachrichten schicken? Es ist wichtig zu verstehen, dass die Übermittlung von Nachrichten zwischen Actors asynchron erfolgt. [DWH18] Dies bedeutet, dass ein Actor, der eine Nachricht an einen anderen Actor ge-

sendet hat, nicht auf eine Antwort wartet und mit der Bearbeitung anderer Nachrichten fortfährt, die von demselben Actor empfangen wurden. Wenn ein Actor jedoch eine Antwort auf seine Nachricht erhalten muss, wartet er einfach auf eine Antwortnachricht mit dem Ergebnis der Verarbeitung seiner Nachricht. Der empfangende Actor muss seinerseits eine Nachricht mit den Ergebnissen senden. [Ver15] Die Interaktion zwischen den Actors ist in dem folgenden Diagramm 2.9 dargestellt.

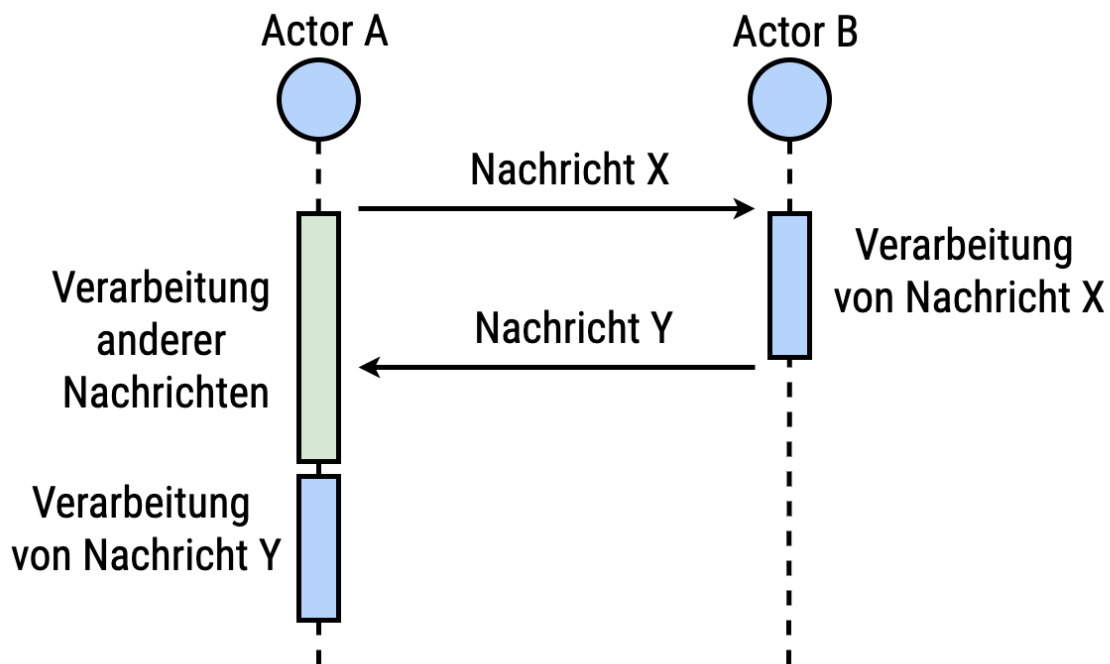


Abbildung 2.9: *Nachrichtenübermittlung zwischen zwei Actors*

Diagramm 2.9 zeigt Nachrichtenübermittlung zwischen zwei Actors. Es zeigt, wie Actor A die Nachricht X an Actor B sendet. Während Actor B diese Nachricht bearbeitet, kann Actor A die anderen Nachrichten, die er erhalten hat, bearbeiten. Wenn Actor B die Verarbeitung von Nachricht X abgeschlossen hat, sendet er Nachricht Y mit seinen Ergebnissen an Actor A. Actor A wiederum fährt mit der Verarbeitung von Nachricht Y fort, sobald er die zuvor erhaltenen Nachrichten verarbeitet hat. Wie bereits beschrieben, entspricht dies einer FIFO-Verarbeitung.

Der Vorteil dieses asynchronen Nachrichtenaustauschs besteht darin, dass keiner der Actors seine Arbeit blockiert, während er auf die Fertigstellung der anderen Komponente wartet. Das heißt, verschiedene Actors arbeiten parallel zueinander, so dass das Actor-System so viele Nachrichten gleichzeitig verarbeiten kann, wie die Hardware unterstützt. [Ver15]

Jeder Actor hat eine Mailbox, in der Nachrichten empfangen werden. Dort werden die Nachrichten gespeichert und warten auf ihre Bearbeitung. [DWH18]

Im Konzept des Actor Model befinden sich die Actors nicht in einem Vakuum, sondern in einem Actor System. Das Actor System ist für den Prozess verantwortlich, bei dem sich die Actors gegenseitig asynchrone Nachrichten schicken. Sie weist auch die Systemressourcen zu, die zur Unterstützung der Actorsumgebung erforderlich sind. [Ver15]

2.3.2 Konzepte reaktiver Systeme in Bezug auf das Actor Model

Das Actor Model erfüllt eindeutig alle Anforderungen an reaktive Systeme. Wie im “The Reactive Manifesto” [Bon+14] beschrieben, muss das Actor Model widerstandsfähig, elastisch, reaktionsfähig und nachrichtengesteuert sein. Die Grundlage für alles ist die asynchrone Kommunikation in Form von Nachrichten zwischen Actors.

Antwortbereit

Das Actor Model reagiert flexibel und zeitnah auf die Anforderungen der Nutzer. [EM17] Mit den Mechanismen für Disaster Recovery, Skalierbarkeit und Multithreading sorgt das Actor Model für schnelle Reaktionszeiten auf Benutzeranfragen. [Ver15]

Widerstandsfähigkeit

Mit seinem speziellen Mechanismus zur Fehlerbehandlung hat sich das Actor Model als ein sehr robustes Modell erwiesen. [Ver15] Wie im Unterkapitel “Selbstheilung und Fehlertoleranz: Actor Supervision” 2.3.4 beschrieben wird, kann das Actor Model mit Fehlern perfekt umgehen und sehr effektiv reagieren.

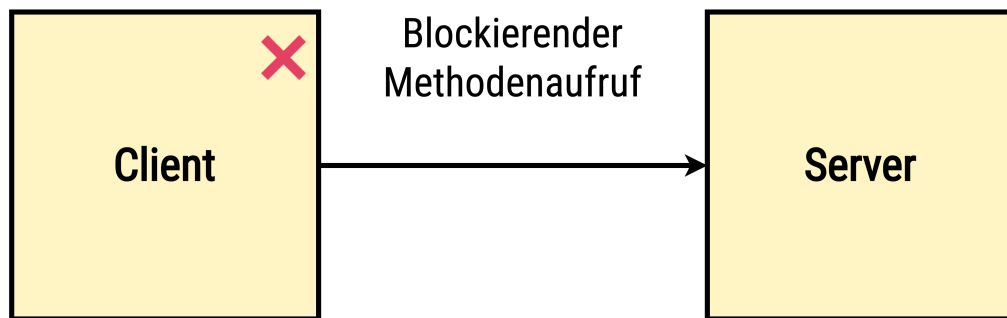
Elastizität

Ein weiteres Merkmal des Actor Model ist die einfache und praktische Skalierbarkeit der Systemkomponenten. Wie bereits beschrieben, kann jeder Actor andere Actors erstellen und ihre Arbeit an diese delegieren. Durch diese Replikation der Komponenten werden die Elastizität und der reibungslose Betrieb unter hoher Belastung erreicht. [Ver15]

Durch eine effiziente Ressourcenzuweisung können Actors auch in verschiedenen Actor Systemen, auf verschiedenen JVMs oder auf verschiedenen Rechnern existieren und dennoch ohne Unterbrechung miteinander kommunizieren. [Ver15]

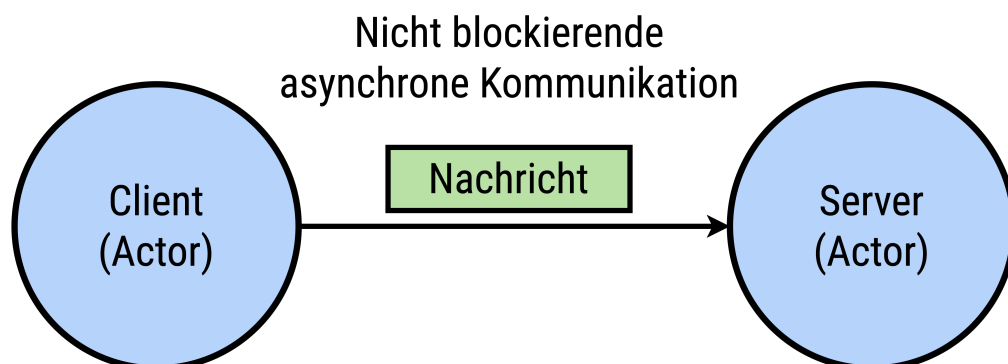
2.3.3 Vergleich von Actor Model und objektorientierte Programmierung

Beim Entwickeln objektorientierter Systeme mit den Programmiersprachen Java oder C++ wird das Blocking-Paradigma angewendet. [XOO21]

Abbildung 2.10: *Blockierender Methodenaufruf* [XOO21]

Wie in der Abbildung 2.10 dargestellt, ruft der Client eine Server-Methode auf, wird selbst blockiert und wartet auf eine Antwort. Während er wartet, kann er keine anderen Anfragen bearbeiten.

Im Gegensatz dazu ist beim Actor Model nichts blockiert. Da die Kommunikation zwischen den Komponenten ausschließlich auf asynchronem Messaging basiert, kann der Sender andere Dinge erledigen, ohne auf eine Antwort des Empfängers zu warten.

Abbildung 2.11: *Nicht blockierende asynchrone Kommunikation* [XOO21]

Dies ist einer der großen Unterschiede zwischen dem Actor Model und der Programmierung mit Objekten. Während in der OOP “alle sind Objekte” ist, ist im Actor Model ‘alle sind Actors’. In dem Buch “Reactive Messaging Patterns” zitiert der Autor die Entwickler des OOP-Stils Alan Kay mit den Worten: [Ver15]

“Das Actor Model hat mehr von dem beibehalten, was ich für die guten Eigenschaften der Objektidee hielt.”

Es gibt eine große Anzahl von Unterschieden zwischen dem Actor Model und der OOP. In der Tabelle 2.1 sind die wichtigsten davon dargestellt.

Actor Model	Objekt-orientierte Programmierung
Alle sind Actors	Alle sind Objekte
Die Kommunikation erfolgt nur asynchron	Eine Mischung aus synchroner und asynchroner Kommunikation
Das Actor Model löst dieses Problem durch die Verwendung des “share nothing”-Modells, so dass die Parallelität nicht beeinflusst wird und kein Sperrmechanismus erforderlich ist	Moderne OOP-Anwendungen berücksichtigen die Parallelität nicht. Folglich ist es sehr einfach, eine “Race Condition” einzuführen, da eine gemeinsame Bedingung verwendet wird
Dank der Supervision-Mechanismen blockieren Fehler nicht das gesamte System	Fehler bei der Anwendung blockieren das gesamte System, bis sie behoben sind

Tabelle 2.1: Grundlegende Unterschiede zwischen Actor Model und OOP [Ver15] [DWH18] [Eck05]

Ein grundlegender Unterschied war das Vorhandensein von Selbstheilungsmechanismen durch die Schaffung von Supervisoren mit wirksamen Strategien. Aufgrund dieser und anderer Vorteile des Actor Model gegenüber anderen Paradigmen wurde entschieden, das Actor Model in dieser Arbeit zu verwenden.

2.3.4 Selbstheilung und Fehlertoleranz: Actor Supervision

Da das Actor Model ein Konzept für reaktive Systeme ist, enthält es ein Konzept wie das der Supervision. Supervision wurde im Unterkapitel “Supervision in reaktiven Systemen” 2.2.2 behandelt, aber hier wird im Detail besprochen, wie sie im Rahmen des Actor Models funktioniert. [KHA17]

Wie bereits beschrieben, können Actors andere Actors schaffen. In diesem Fall entsteht zwischen ihnen eine Beziehung zwischen Vorgesetzten und Arbeitnehmern. Der erstellende Actor wird zum Vorgesetzten des erstellten Actors. [KHA17]

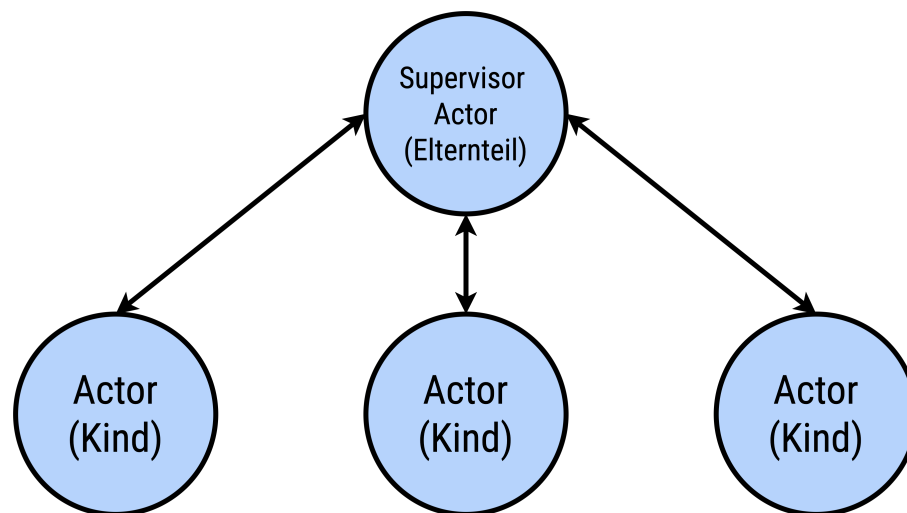


Abbildung 2.12: *Supervisor mit seinen unterstellten Actors*

Der Supervisor kann die Arbeit an seine Untergebenen delegieren. Auch alle Untergebenen melden ihre Probleme an ihren Supervisor. Das Actor Model verwendet das “The Proactive Failure Signal”-Pattern 2.2.3. Actors, die auf ein Problem stoßen, müssen es selbst ihrem Vorgesetzten melden, damit dieser geeignete Maßnahmen ergreifen kann. [DWH18]

Wie im Unterkapitel “Supervision in reaktiven Systemen” 2.2.2 beschrieben, hat ein Actor mehrere Szenarien, in denen er handeln kann. Im Actor Model sind diese Aktionen identisch:

- Den Actor starten
- Den Actor stoppen
- Den Actor neu starten
- Eskalation zum Supervisor [DWH18]

Aber die Supervisor ist nicht auf diese Maßnahmen beschränkt, sie sind nur grundlegend. Der Supervisor kann eine beliebige Anzahl von maßgeschneiderten Strategien erstellen, um den in Schwierigkeiten geratenen Actor wiederherzustellen. [McK17]

Im Actor System hat jeder Actor einen Supervisor und jeder Actor kann ein Supervisor sein. Das Ergebnis ist eine einheitliche Hierarchie von Actors, an deren Spitze der Root-Actor steht. Wenn ein Fehler aus irgendeinem Grund den Root-Actor erreicht, wendet er die Strategie des Neustarts des Actors an. [McK17]

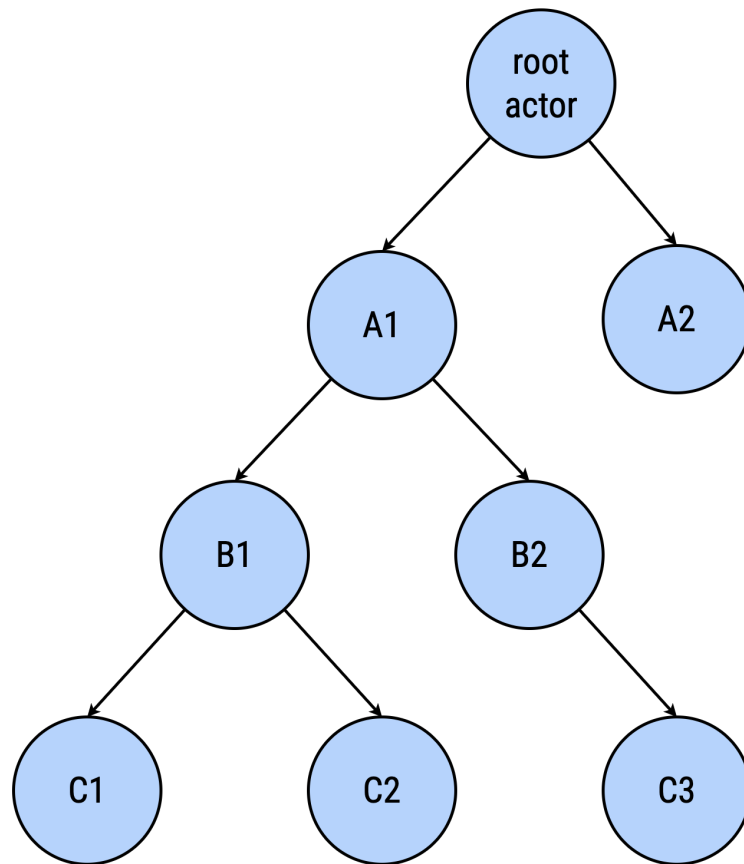


Abbildung 2.13: *Ein Beispiel für eine Hierarchie von Actors*

Das Diagramm 2.13 zeigt, wie der Root-Actor ganz oben im Diagramm steht und alle anderen Actors aus ihm gebildet werden. Durch die Schaffung einer Eltern-Kind-Beziehung bilden sie eine große Hierarchie von Actors. [McK17]

Es gibt nur zwei eingebaute Strategien in der Supervisor: [Ver15]

- One-For-One-Strategie
- All-For-One-Strategie

Diese Strategien unterscheiden sich nur darin, inwieweit sich die Fehlerkorrekturmaßnahmen der Supervisor auswirken werden.

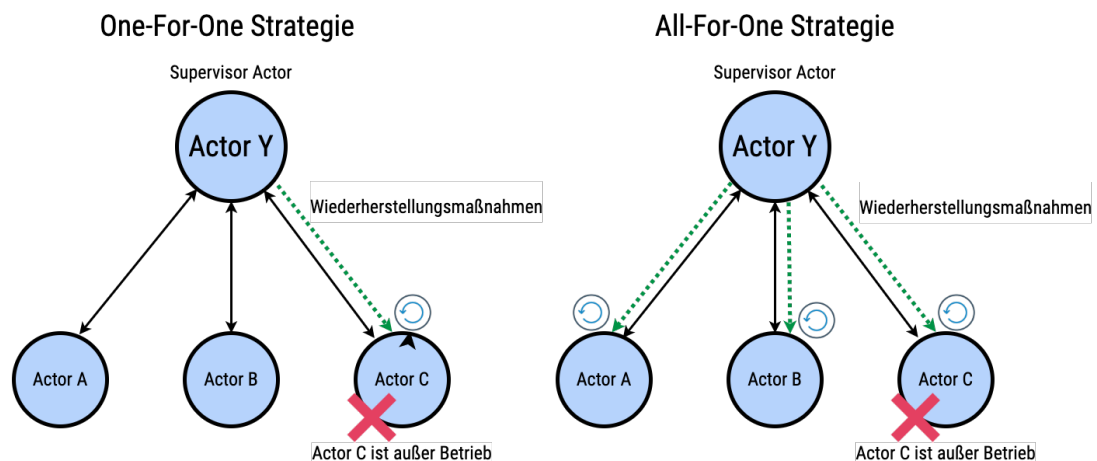


Abbildung 2.14: Supervisor-Strategien: One-For-One und All-For-One

Die One-For-One-Strategie bedeutet, dass Maßnahmen zur Wiederherstellung eines gesunden Zustands eines Actors nur für den ausgefallenen Actor gelten. Diese Aktionen haben keine Auswirkungen auf die anderen untergeordneten Actors. Diese Strategie ist die Standardstrategie. [Ver15]

Die All-For-One-Strategie ist das Gegenteil der One-For-One-Strategie und bedeutet, dass alle Maßnahmen gleichermaßen für den gescheiterten Actor und für alle anderen Child-Actors gelten. [Ver15]

Actor Monitoring

Es gibt auch ein Aufsichtsmodell im System, das keine Verbindung zwischen Vorgesetzten und Arbeitnehmern vorsieht. Dies bedeutet, dass ein bestimmter Actor nicht nur von seinem Elternteil, sondern auch von einem anderen Actor überwacht werden kann, wenn die Architektur von Anfang an so aufgebaut ist. [McK17]

Wenn ein Actor seine Arbeit beendet, sendet er eine Nachricht an den übergeordneten Actor, um ihn über seinen Zustand zu informieren. Der überwachende Actor wird seinerseits tätig, wenn er diese Nachricht erhält. [McK17]

Diese Vorgehensweise hilft dabei, die Zustände der Actors zu verfolgen, die für den Betrieb der größeren Teile des Systems wichtig sind. Wenn ein Actor in einer großen Systemkette ausfällt, wird das gesamte System versagen. Um dies zu verhindern, muss das System überwacht werden, um einen gesunden Gesamtzustand zu erhalten. [McK17]

Wenn ein Actor ausfällt, können die anderen Actors, die von ihm abhängen, ihre Aufgaben nicht mehr erfüllen. Wenn ein Actor ausfällt, wird eine große Anzahl von Anfragen an die

ausgefallene Komponente gestellt, und diese erhöhte Last kann den Wiederherstellungsprozess behindern. [McK17]

Das “Circuit Breaker”-Pattern kann angewendet werden, um dieses Problem zu lösen. Die Grundidee des Patterns besteht darin, eine Schicht, den so genannten Circuit Breaker, zwischen die Anfragen und den Actor zu schalten. Das heißt, alle Anfragen werden über sie laufen. Aber hier hat der Leistungsschalter zwei Zustände - offen und geschlossen. Im Normalfall, d. h. bei normalem Betrieb des Systems, ist der Leistungsschalter geschlossen und ermöglicht es, dass Nachrichten den Actor erreichen. Wenn der Actor jedoch ausfällt, wechselt der Zustand des Unterbrechers auf “offen” und alle empfangenen Nachrichten werden zurückgewiesen. [McK17]

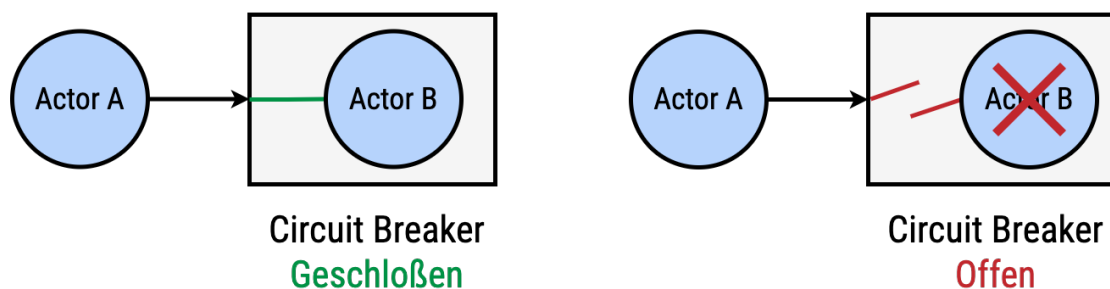


Abbildung 2.15: Das “Circuit Breaker”-Pattern [McK17]

Während der Circuit Breaker alle eingehenden Nachrichten aufhält, werden Wiederherstellungsmaßnahmen durchgeführt. Sobald der Fehler behoben und der Actor wieder in Betrieb ist, wechselt der Circuit Breaker in den Zustand “geschlossen” und verhindert nicht, dass Nachrichten an den Actor übertragen werden. [McK17]

Vor- und Nachteile von Actor Model

Vorteile:

- Asynchrone Kommunikation zwischen Actors
- Einfache Skalierbarkeit
- Eingebauter Selbstheilungsmechanismus und Fehlertoleranz
- Verteilung der Knotenpunkte unabhängig vom Standort
- Kein gemeinsamer Zustandsspeicher

Nachteile:

- Verteilung der Geschäftslogik auf eine große Anzahl von Actors
- Nachrichtenüberlauf in den Mailboxen der Actors

2.3.5 Frameworks für Actor Model

Es gibt verschiedene Möglichkeiten der Programmierung mit Hilfe des Actor Models. Entweder implementieren eine Programmiersprache direkt das Actor Model, wie z.B.: ActorScript, Smalltalk, Erlang, Scala, Scratch. [Ver15]

Die zweite Möglichkeit besteht darin, ein spezifisches Framework anzuwenden, das das Actor Model implementiert. Dies ermöglicht die Verwendung einer der am weitesten verbreiteten Programmiersprachen - Java. Es gibt viele verschiedene Frameworks, die das Konzept des Actor Models mehr oder weniger stark umsetzen. [Fra]

Sie unterscheiden sich in vielerlei Hinsicht: Einige von ihnen implementieren das Actor Model auf der Grundlage der JVM, andere auf der Grundlage der Programmiersprachen Python, Go, C++ und so weiter. [Fra]

Zwei auf der JVM basierende Frameworks heben sich von der Masse ab. Diese sind Akka² und VLINGO XOOM³. Die verbreitetste davon [Fra] ist das Akka-Framework. Die offizielle Akka-Website [Akk] bietet diese Definition des Frameworks:

“Akka ist ein Toolkit zur Erstellung hochgradig nebenläufiger, verteilter und robuster nachrichtengesteuerter Anwendungen für Java und Scala.”

In Bezug auf Akka ist VLINGO XOOM ein junges Projekt, das über eine große Anzahl von Funktionen verfügt, einschließlich der Implementierung des Actor Models mit dem XOOM Actors-Toolkit. [Xoo]

Auf der offiziellen Dokumentationsseite wird das Instrument auf diese Weise beschrieben: [Xoo]

“XOOM Actors ist die reaktive Basis-Laufzeitumgebung, die der gesamten Plattform zugrunde liegt. Es unterstützt hohe Skalierbarkeit mit vollständig asynchronem, typsicherem Messaging. Um dies zu erreichen, ist unser Toolkit eine Implementierung des Actor Models, und alle primären Plattformkomponenten bauen auf unserer Actor-Model-Implementierung auf.”

In vielerlei Hinsicht sind sich die beiden Frameworks ähnlich, da sie beide das Actor Model vollständig implementieren. Sie unterscheiden sich hauptsächlich in der Syntax und der Komplexität der Programmierung.

Nach der Analyse der vorhandenen Dokumentation wurde beschlossen, sich für das VLINGO XOOM-Framework zu entscheiden und es auszuprobieren. Es ist relativ einfach zu programmieren und der Einstieg in die Programmierung mit dem Actor Model ist klar.

²Offizielle Webseite von Akka Framework: <https://akka.io/>

³Offizielle Webseite von VLINGO XOOM Framework: <https://vlingo.io/>

Kapitel 3

Umsetzung

In diesem Kapitel wird das Konzept eines selbstheilenden und fehlertoleranten Systems beschrieben. Die Einführung des Konzepts ist wichtig für das weitere erfolgreiche Verständnis der Implementierung. Das nächste Unterkapitel nach dem Konzept ist die prototypische Implementierung unter Verwendung des Actor Models und der im Kapitel: Theoretische Grundlagen 2 beschriebenen Themen.

3.1 Konzept

Wie im theoretischen Teil beschrieben, gibt es bei Software immer Raum für Fehler. Egal wie perfekt die Software geschrieben ist, sie wird trotzdem fehlschlagen. Es geht also nicht darum, perfekte Software ohne Fehler zu schreiben, sondern darum, wie man mit Fehlern umgeht, sich selbst heilt und fehlertolerant ist.

In diesem Konzept eines selbstheilenden und fehlertoleranten Systems wird die Software in erster Linie reaktiv sein. Durch die Anwendung der Grundsätze reaktiver Systeme kann ein hohes Maß an Selbstheilung und Fehlertoleranz erreicht werden. Das Resilienzprinzip ist insbesondere dafür verantwortlich, dass das System immer verfügbar ist und Störungen effizient behandelt.

Außerdem vermeidet das Elastizitätsprinzip durch automatische Skalierung Fehler aufgrund von Systemüberlastungen.

Es kann eine große Anzahl von Implementierungen reaktiver Systeme geben. Aber in dem Konzept wurde das Actor Model als eine der besten Implementierungen reaktiver Systeme verwendet. Durch die Selbstheilungsmechanismen der Actors und die Anwendung von Resilienzmustern kann das Ziel dieser Arbeit erreicht werden.

Die wichtigste Idee und Instrument zur Selbstheilung in diesem Konzept ist der Supervisor-Mechanismus. Das heißt, wenn ein Actor in irgendeiner Weise versagt, muss die Supervisor

geeignete Maßnahmen ergreifen, um den Actor wieder in einen gesunden Zustand zu versetzen.

Die Patterns “Simple Component” und “Let-It-Crash” werden ebenfalls in dem Konzept verwendet. Die einfache Komponente ist besonders wichtig, weil sie die Komplexität des Systems vermeidet und es ermöglicht, die Actors so atomar wie möglich zu gestalten.

Da es unmöglich ist, alle Fehler vorherzusehen, es sollte das Pattern “Let-It-Crash” anwenden.

Ein vollständiges Bild des Konzepts ist in der folgenden Abbildung 3.1 zu sehen. Es zeigt abstrakt auf, welche Techniken und Methoden verwendet werden sollen. Die Supervisors werden aktiv als Selbstheilungsmechanismus eingesetzt. Darüber hinaus werden die Patterns “Simple Component”, “Let-It-Crash” und “Circuit Breaker” verwendet, um ein bestimmtes Niveau der Fehlertoleranz zu erreichen.

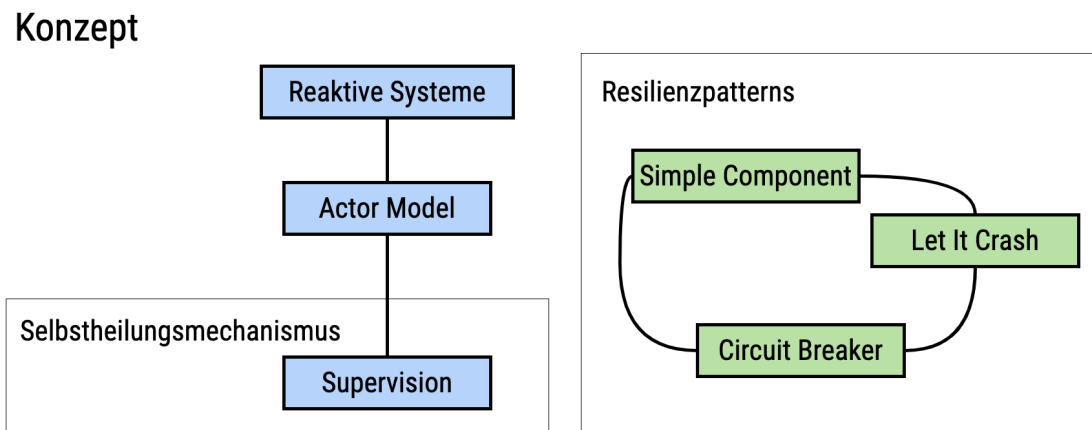


Abbildung 3.1: *Konzept von selbstheilendes und fehlertolerantes System*

Bei der Programmierung muss ein “defensiver” Programmierstil verwendet werden, um fehlertoleranten Code zu schreiben. Zu einem “defensiven” Programmierstil gehört es, sich möglicher Fehler bewusst zu sein und bereit zu sein, sie zu beheben. Beim Schreiben von Code muss sich der Programmierer jedes Mal fragen: Was kann hier schief gehen? Welche Fehler können auftreten? Wie kann dieser Code fehlerfrei sein? [Han07]

Es gibt so viele Patterns, um Fehlertoleranz zu erreichen. Nur drei von ihnen werden in das Konzept aufgenommen, da die Verwendung einer großen Anzahl von Patterns die Komplexität des Projekts erhöhen würde. Dies sind die grundlegendsten Patterns, die in jeder Software implementiert werden müssen, die Fehlertoleranz erreichen wollen. Um es einfach und gleichzeitig fehlertolerant zu halten, wird sich das Konzept daher auf einen defensiven Programmierstil und drei Patterns konzentrieren. Wenn das Projekt entwickelt,

wird es natürlich notwendig sein, weitere Patterns zu implementieren. Innerhalb dieses Konzepts können weitere Patterns implementiert werden, wodurch sich das Niveau der Fehlertoleranz der Software erhöht.

3.1.1 Beispielprojekt

Um das Konzept zu realisieren, wird ein Beispielprojekt benötigt, in dem die Software erstellt werden soll. Als erstes muss das Geschäftsfeld gewählt werden, für das die Software erstellt werden soll.

Skalierbare, fehlertolerante, zeitnah reagierende, an variable Arbeitslasten anpassbare und vor allem selbstheilende Software wird in jedem Bereich nützlich sein. In Bereichen wie Spiel, elektronischer Handel, Telekommunikation, Banken, elektronischer Zahlungsverkehr, Medienanbieter usw. besteht jedoch ein großer Bedarf an solcher Software. Die Erstellung einer solchen Software ist durch die Anwendung reaktiver Systeme und des oben beschriebenen Konzepts möglich. [Opr21]

Schließlich wurde entschieden, sich auf das Banken zu konzentrieren, wo die Kosten von Fehlern besonders hoch sind. In der Welt der Banktransaktionen spielen Fehler und die Folgen dieser Fehler eine große Rolle. Ein Systemausfall kann zu zahlreichen Ausfällen der Bank selbst oder der zugehörigen Software führen. Da eine solche Software direkt mit den Finanzen und den sensorischen Daten der Nutzer zu tun hat, ist es von entscheidender Bedeutung, dass das System fehlertolerant ist und sich selbst heilen kann.

Eine Software für eine große Bank hätte eine komplexe Architektur und eine große Anzahl von Microservices⁴. Daher wurde in diesem Prototyp die ATM-Software verwendet, um Selbstheilung und Fehlertoleranz auf Anwendungsebene zu zeigen. Anhand dieses Beispiels kann das Konzept veranschaulicht und das Ziel der gesamten Arbeit erreicht werden.

Der Begriff ATM steht für “Automated Teller Machine” (Geldautomat). Es ist ein Gerät, mit dem Karteninhaber jeder Bank verschiedene Transaktionen durchführen können. Der Geldautomat verwendet ein Tastenfeld und eine Plastikkarte, auf deren Rückseite sich ein Magnetfeld befindet, in das alle erforderlichen Informationen eingegeben werden. In Kombination mit Hard- und Software liest der Geldautomat diese Informationen und bietet dem Benutzer seine Dienste an. [Kag21]

Die Funktionalität eines Geldautomaten unterscheidet sich von Modell zu Modell. Die einfachsten ermöglichen, Kontoinformationen einzusehen und Bargeld von Ihrem Konto abzuheben. Es gibt auch Modelle, die komplexere Vorgänge ermöglichen, z. B. Überweisungen von Konto zu Konto, Änderung des Pincodes und so weiter. [Kag21]

⁴Microservices - ist ein Architekturstil, der eine Anwendung als Sammlung von Diensten strukturiert. Offizielle Website: <https://microservices.io/>

Jeder Geldautomat ist mit einer Software ausgestattet, die die Transaktionen innerhalb des Geldautomaten verwaltet und mit dem Benutzer und der Bank interagiert. Wenn der Benutzer eine Karte und den Pin-Code für diese Karte eingibt, stellt die Geldautomaten-Software eine Anfrage an die Bank des Kunden. [Kag21]

Fordert der Benutzer Bargeld an, hebt die Geldautomatensoftware es vom Konto des Benutzers ab. Sobald die Bank die Abhebung genehmigt hat, sendet sie eine Art Meldung, die anzeigt, dass sie befugt ist, das Geld an den Nutzer auszuzahlen. An diesem Punkt ist die Software mit ihrer Arbeit fertig. [Kag21]

Ausgehend von der oben beschriebenen Funktionalität der ATM-Software wurde die Idee einer ATM-Management- und Prozessautomatisierungssoftware als Beispielprojekt übernommen.

3.1.2 Systemanforderungen für die Umsetzung

Um zu wissen, ob die Ziele des Konzepts erreicht wurden, müssen die Anforderungen an das System definiert werden. Die Anforderungen werden in zwei Arten unterteilt: funktionale und nicht-funktionale. Im Folgenden sind die Anforderungen für eine prototypische Implementierung aufgeführt.

Funktionale Anforderungen

- Das System sollte selbst mit Fehlern umgehen
- Das System sollte auf viele Standardfehler vorbereitet sein und über Strategien zur Wiederherstellung verfügen
- Das System sollte in einfache Komponenten mit jeweils nur einer Zuständigkeit unterteilt werden
- Das System sollte kritische Ausfälle vermeiden, indem es schnell und zeitnah auf Fehler reagiert
- Das System sollte die Patterns “Simple Component”, “Let-It-Crash” und “Circuit Breaker” implementieren
- Das System sollte den PIN-Code der Karte überprüfen
- Das System sollte den Namen der Bank von der Karte prüfen, ob die Karte von diesem Geldautomaten bedient wird
- Das System sollte in der Lage sein, Geld von der Bank anzufordern

Nicht funktionale Anforderungen

- Das System sollte selbstheilend sein
- Das System sollte fehlertolerant sein

Im nächsten Unterkapitel wird beschrieben, wie diese Anforderungen erfüllt wurden.

3.2 Prototypische Implementierung

In diesem Unterkapitel wird eine prototypische Implementierung eines Beispielprojekts vorgestellt.

Als Architekturstil werden reaktive Systeme und das Actor Model verwendet. Die Programmierung mit dem Actor Model erfordert die Verwendung eines spezifischen Frameworks und einer Programmiersprache. Aufgrund persönlicher Präferenzen und des Vergleichs im Kapitel “Frameworks für Actor Model” 2.3.5 wurde entschieden, den folgenden Stack von Technologien und Tools zu verwenden:

- Programmiersprache: Java 11
- Framework: VLINGO XOOM
- Ein Instrument zur Automatisierung der Projektzusammenstellung: Gradle
- Entwicklungsumgebung: IntelliJ IDEA

Um mit der Umsetzung des Projekts beginnen zu können, musste die Arbeitsumgebung eingerichtet und alle erforderlichen Abhängigkeiten beseitigt werden. Zunächst sollten das leeres Gradle-Projekt mit der Entwicklungsumgebung IntelliJ IDEA erstellt werden. Da bei der Entwicklung das Build-Tool Gradle verwendet wurde, wird im Folgenden Code aus der Datei “build.gradle” semantisch dargestellt.

Auflistung 3.1: Abhängigkeiten in *build.gradle*

```
dependencies {  
    compile 'io.vlingo.xoom:xoom-actors:1.8.0'  
    compile 'junit:junit:4.12'  
  
    testCompile group: 'junit', name: 'junit', version: '4.12'  
  
    implementation group: 'org.slf4j', name: 'slf4j-api', version: '2.0.0-alpha1'  
    testImplementation group: 'org.slf4j', name: 'slf4j-simple', version: '2.0.0-alpha1'  
}
```

Wie in der Auflistung 3.1 zu finden ist, wird das jar vlingo.xoom:xoom-actors Version 1.8.0 als Abhängigkeiten bezeichnet. Außerdem die JUnit-Testbibliothek Version 4.12 und das Tool slf4j Version 2.0.0-alpha1.

3.2.1 Geschäftsprozessablauf

Wie in der Projektbeschreibung 3.1.1 beschrieben, ist die ATM-Software in der Lage, bestimmte Operationen durchzuführen, wie z. B.

- Karteninformationen an den User ausgeben
- Bargeld an den User ausgeben
- Bargeld bei der Bank anfordern

Diese Funktionen sollten im Projekt mit Hilfe der oben genannten Technologien implementiert werden.

Ein Beispiel ist der Fall, dass ein Benutzer seine Karte in einen Geldautomaten einführt und damit Prozesse in der Software auslöst. Dies ist einer der einfachsten Mechanismen innerhalb des Systems. Sie wird als Aktivitätsdiagramm dargestellt:

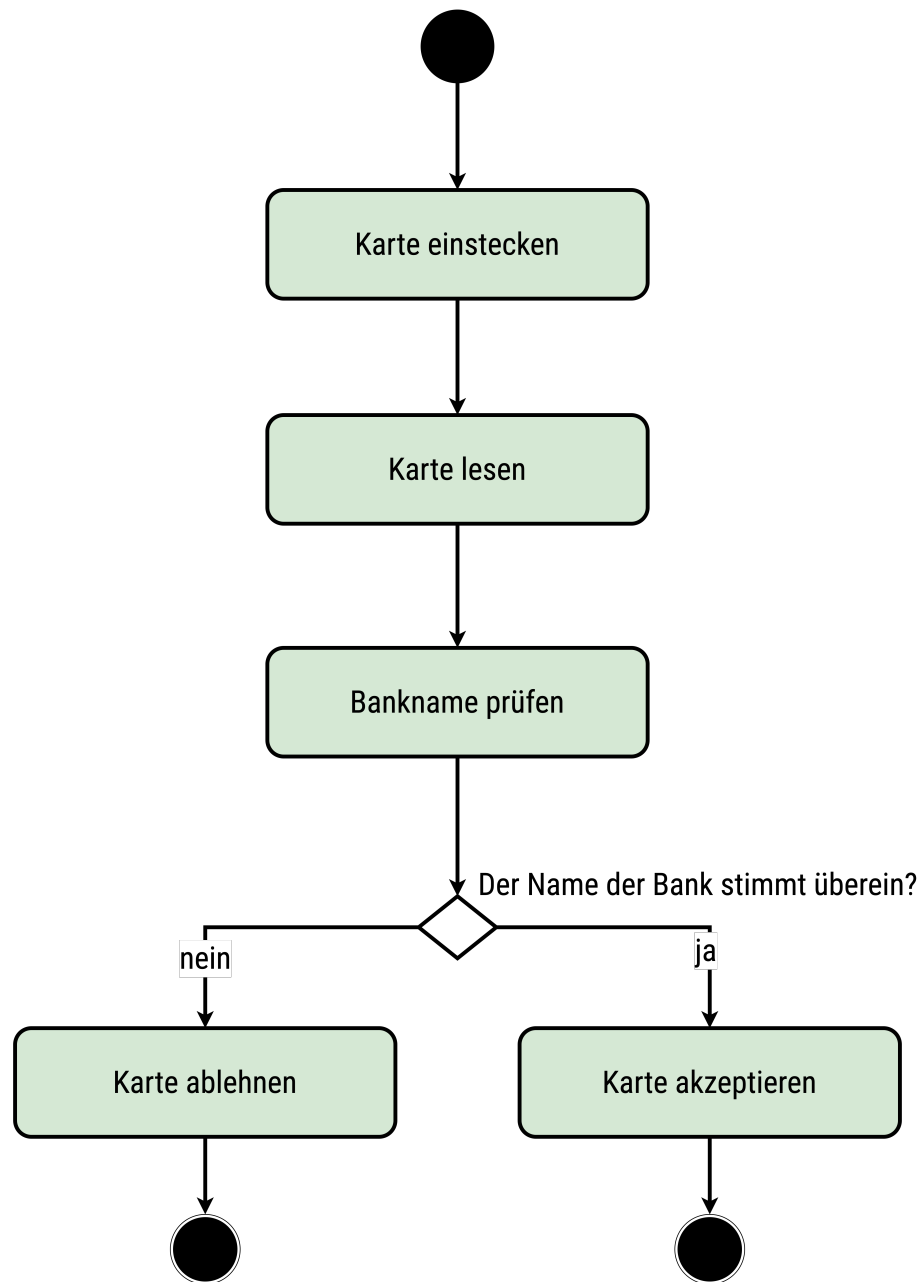


Abbildung 3.2: *Aktivitätsdiagramm die eine Funktion von ATM-Software darstellt*

Das Diagramm 3.2 zeigt, wo der Mechanismus zum Erhalt der Karte beginnt. Der Benutzer steckt die Karte in den Geldautomaten und dann beginnt die Software mit ihrer Arbeit. Der Geldautomat liest die Karte, extrahiert die erforderlichen Informationen und überprüft dann den Namen der Bank auf der Karte. Es geht darum, dass, wenn eine nicht autorisierte Bankkarte in den Geldautomaten eingeführt wird, der weitere Betrieb eingestellt werden sollte. Deshalb folgt auf “Bankname prüfen” die Frage: Der Name der Bank stimmt überein? Wenn der Name der Bank in der Liste der bedienten Banken enthalten

ist, wird die Karte akzeptiert, wenn nicht, wird es abgelehnt. Dieses Diagramm hat also einen Eingang und zwei Ausgänge.

Auf der Grundlage solcher Diagramme können die atomaren Komponenten des Systems, nämlich die Actors, definiert werden. Das bedeutet, dass bei Anwendung des “Simple Component”-Pattern die Komponenten in viele kleine Komponenten mit einer Verantwortung aufgeteilt werden sollten.

Aus dem Diagramm 3.2 sind folgende Actors zu erkennen:

- Karte einstecken - Um diese Aktion zu starten, wird *AtmActor* als Einstiegspunkt verwendet.
- Karte lesen - *CardReaderActor* wird die Karte lesen. Der *CardActor* wird auch benötigt, um Daten zu speichern und einige Operationen durchzuführen. Dort werden alle notwendigen Informationen über die Karte gespeichert
- Bankname prüfen - *CardReaderActor* führt auch diesen Prozess durch
- Karte ablehnen - *AtmActor* ist für diesen Prozess verantwortlich
- Karte akzeptieren - *AtmActor* ist auch für diesen Prozess verantwortlich

Infolgedessen werden die folgenden Actors zur Umsetzung des Prototyps erstellt:

- *AtmActor*
- *CardActor*
- *CardReaderActor*
- *BankActor* - wird in anderen Mechanismen und Funktionen verwendet

3.2.2 Implementierung von Actors

Actors sind unabhängige Komponenten des Systems, sie sind im Wesentlichen Objekte, aber sie können Nachrichten von anderen Actors empfangen und Nachrichten an andere Actors senden, wobei sie ihren Zustand ändern.

Wie in der objektorientierten Programmierung die gewünschte Anzahl von Objekten, Instanzen einer Klasse, erstellt werden kann, so kann auch im Actor Model die gewünschte Anzahl von Actor-Instanzen erstellt werden. Actor-Instanzen haben keine synchronisierten Zustände, jede Instanz hat ihren eigenen Zustand.

Ein Beispiel dafür ist der *CardActor*. Die Klasse *CardActor* implementiert die Schnittstelle *Card*. Die *Card*-Schnittstelle sieht wie folgt aus:

Auflistung 3.2: *Card*-Schnittstelle

```
public interface Card extends Stoppable {  
    void sendBankName(final Atm atm, final CardReader cardReader);  
    void sendPin(String typedPin, final Atm atm, final CardReader cardReader);  
}
```

Die *Card*-Schnittstelle beschreibt, dass der *CardActor* zwei Arten von Nachrichten annehmen kann. *sendBankName()* und *sendPin()* sind eigentlich Methoden der Klasse *CardActor*. Daher ist der Mechanismus zum Schreiben des Codes ähnlich dem objektorientierten Stil. Um eine Nachricht an den *CardActor*-Actor zu senden, rufen die anderen Actors nur eine seiner Methoden auf.

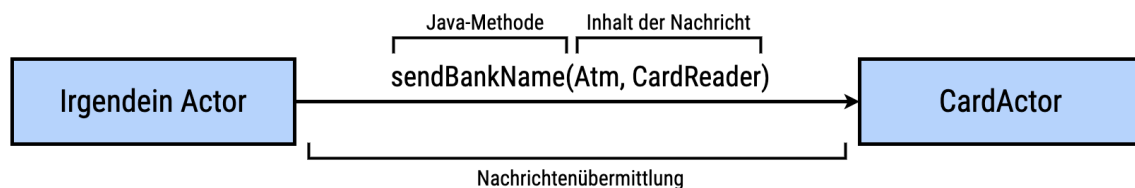


Abbildung 3.3: *Irgendein Actor sendet eine Nachricht an CardActor*

Eigentlich ist *sendBankName()* eine Java-Methode, denn der Code ist in der Programmiersprache Java geschrieben. Aber auch jede Methode bei einem Actor zeigt an, dass der Actor Nachrichten empfangen kann. Das heißt, die Methode ist der Weg für das Senden von Nachrichten an den Actor. Abbildung 3.3 zeigt zum Beispiel den Weg der Nachrichtenübertragung. Die Methode *sendBankName()* hat zwei Klassen in ihren Parametern: *Atm* und *CardReader*. *CardActor* benötigt diese Daten, um seine Aufgabe zu erfüllen. Das heißt, in den Parametern der Methode gibt *CardActor* an, was er zu empfangen erwartet. Folglich ist die Methode eine Möglichkeit, eine Nachricht an den Actor zu senden, und die Parameter sind der Inhalt der Nachricht.

Der Empfang dieser Nachricht durch *CardActor* ist nur die erste Schritt. *CardActor* muss in der Lage sein, die empfangene Nachricht zu verarbeiten. Die eigentliche Verarbeitung findet im Hauptteil der Methode *sendBankName()* statt. Wie in der Auflistung 3.3 dargestellt, hat *CardActor* seinen eigenen Zustand. Es hat zwei Felder: *pin* und *bankName*, und speichern die PIN-Code und den Banknamen.

Auflistung 3.3: Implementierung von *CardActor*

```
public class CardActor extends Actor implements Card {
    private final String pin;
    private final String bankName;
    private final Card self;

    public CardActor(final String pin, final String bankName) {
        this.self = selfAs(Card.class);
        this.pin = pin;
        this.bankName = bankName;
    }

    @Override
    public void sendBankName(final Atm atm, final CardReader cardReader) {
        logger().info("Sending bank name to CardReader");

        cardReader.checkBankName(atm, self, bankName);
    }

    @Override
    public void sendPin(String typedPin, final Atm atm, final CardReader cardReader)
    {
        logger().info("Sending pin to CardReader");

        cardReader.checkPin(atm, self, pin, typedPin);
    }
}
```

Das Verhalten der Methode *sendBankName()* ist einfach. Wenn die Nachricht “Bankname senden” ankommt, protokolliert der *CardActor* zunächst und ruft dann die *CardReader*-Methode auf, d. h. er sendet an *CardActor* eine Nachricht.

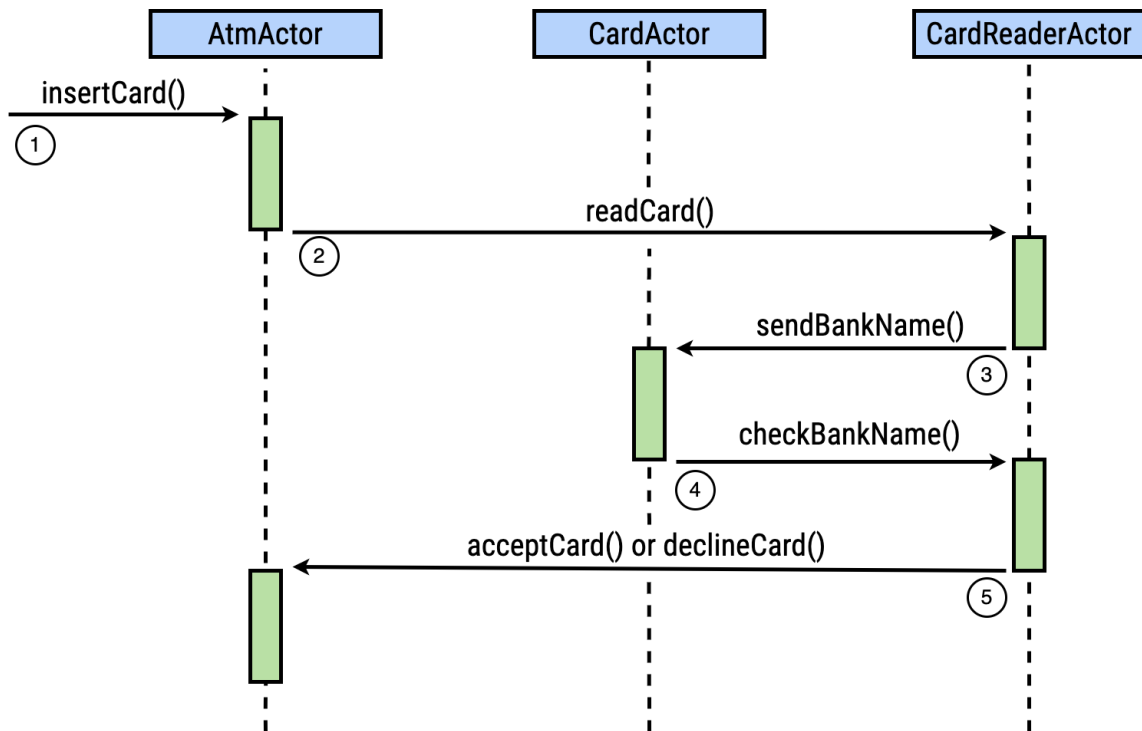


Abbildung 3.4: Nachrichtenfluss zwischen *AtmActor*, *CardReaderActor*, *CardActor*

Die Abbildung 3.4 zeigt den Nachrichtenfluss zwischen *AtmActor*, *CardActor* und *CardReaderActor*:

1. Am Anfang erhält *AtmActor* die Nachricht *insertCard()*, dass eine Karte eingesteckt wurde
2. Danach sendet *AtmActor* seinerseits eine Nachricht *readCard()* an *CardReaderActor*, um die Karte zu lesen
3. Anschließend wird die Nachricht verarbeitet und die Nachricht *sendBankName()* an *CardActor* gesendet
4. *CardActor* verarbeitet diese Nachricht und sendet eine weitere *checkBankName()*-Nachricht zurück an *CardReaderActor*
5. Der *CardReaderActor* entscheidet dann, ob er die Karte akzeptiert oder ablehnt und sendet die entsprechende Nachricht an *AtmActor*

Die gestrichelte Linie in der Abbildung 3.4 zeigt die Bereitschaft an, weitere Nachrichten zu verarbeiten.

Das VLINGO XOOM-Framework [XOO21] vereinfacht die Interaktion zwischen Actors, indem es sie in analoge Java-Objekte verwandelt. Der Programmierstil ist ähnlich wie bei

der objektorientierten Programmierung in dem Punkt, wo Methoden aufgerufen werden. Der Unterschied ist jedoch, dass in VLINGO XOOM alle Methoden keinen Rückgabewert haben, da die Kommunikation im Actor Model asynchron erfolgt.

3.2.3 Implementierung von Actor Supervision

Die Abbildung 3.4 zeigt das korrekte Verhalten des Systems, aber in jeder Phase kann ein Fehler auftreten und die weitere Arbeit wird unterbrochen. Da jeder Schritt von Actor durchgeführt wird, müssen diese über Selbstheilungsmechanismen verfügen. Wie im Kapitel 2.3.4 “Selbstheilung und Fehlertoleranz: Actor Supervision” beschrieben, werden Fehler in den Actors von einem anderen Actor behandelt, der für diese verantwortlich ist.

Im Actor-System hat jeder Actor seinen Standard-Supervisor. VLINGO XOOM bietet standardmäßig eine Hierarchie von Supervisor-Actors. Es gibt zwei Standard-Supervisors: [XOO21]

- *PrivateRootActor* - Ein grundlegender Supervisor, der das System vor einem katastrophalen Ausfall schützt
- *PublicRootActor* - Supervisor Actor, von dem alle Supervisoren im System erben, außer *PrivateRootActor* [XOO21]

Ihr Verhalten ist auf statische Fehlerbehebungsstrategien beschränkt. Es ist selbstverständlich, dass die Erstellung eigener Supervisor mehr Flexibilität und spezielles Verhalten bietet. [XOO21]

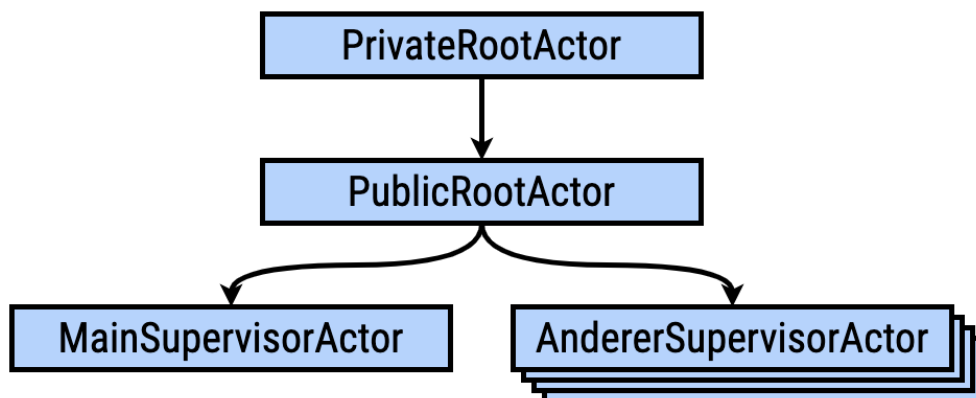


Abbildung 3.5: *Supervisor-Hierarchie in VLINGO-XOOM Framework* [XOO21]

Wie die Abbildung 3.5 zeigt, steht *PrivateRootActor* als Basis-Supervisor und dann kommt *PublicRootActor*. Alle im Projekt erstellten Actors werden standardmäßig dem *PublicRootActor* zugewiesen, z. B. *MainSupervisorActor*.

Das ATM-Projekt hat einen einzigen Supervisor für alle aufgeführten Actors. Er wird als *MainSupervisorActor* benannt. Es hat Strategien, wie sie auf bestimmte Fehler der untergeordneten Actors reagieren wird.

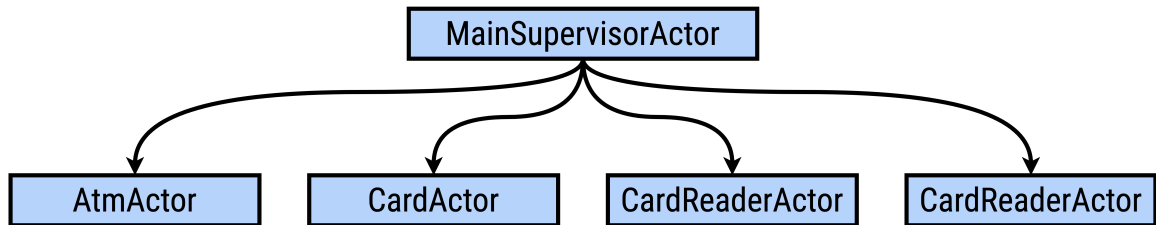


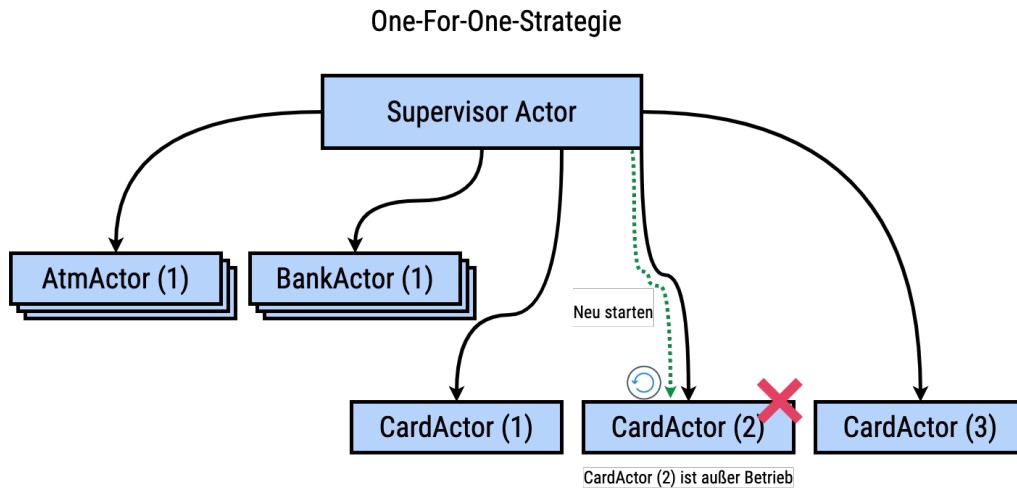
Abbildung 3.6: *MainSupervisorActor* mit andere Actors

Die Abbildung 3.6 zeigt, dass der *MainSupervisorActor* alle Actors im System kontrolliert. Wenn einem neu erstellten Actor kein Supervisor zugewiesen ist, wird er standardmäßig *PublicRootActor* zugewiesen.

Auflistung 3.4: Konfiguration für *MainSupervisorActor*

```
Configuration configuration =  
    Configuration  
        .define()  
        .with(CommonSupervisorsPlugin.CommonSupervisorsPluginConfiguration  
            .define()  
            .supervisor("default", "mainSupervisor", Atm.class,  
                MainSupervisorActor.class)  
            .supervisor("default", "mainSupervisor", CardReader.class,  
                MainSupervisorActor.class)  
            .supervisor("default", "mainSupervisor", Bank.class,  
                MainSupervisorActor.class)  
            .supervisor("default", "mainSupervisor", Card.class,  
                MainSupervisorActor.class)  
        );
```

Die Auflistung 3.4 zeigt, wie der *MainSupervisorActor* als Supervisor für den *AtmActor*, *CardReaderActor*, *BankActor*, *CardActor* zugewiesen wird. Das heißt, wenn eine Instanz dieser Actors eine Exception auslöst, wird diese von der Laufzeit erkannt und dann sollte der *MainSupervisorActor* darauf reagieren. Dadurch wird die fehlgeschlagene Actor-Instanz unterbrochen und die Verarbeitung anderer Nachrichten gestoppt, bis die Exception behandelt worden ist. Die anderen Instanzen desselben Actors funktionieren weiterhin, da sie ihre Zustände nicht synchronisieren.

Abbildung 3.7: *One-For-One-Strategie mit Actor-Instanzen*

Zum Beispiel hat jeder Actor mehrere Instanzen, wie in Abbildung 3.7 dargestellt. *CardActor* hat 3 Instanzen. Wenn *CardActor (2)* ausfällt, sind die Wiederherstellungsmechanismen nur an diese Instanz gerichtet. *CardActor (1)* und *CardActor(3)* funktionieren weiterhin wie gewohnt. Das ist die One-for-One-Strategie. Bei der All-For-One-Strategie sind die Wiederherstellungsmechanismen an jede der Instanzen gerichtet, d. h. *CardActor(1)*, *CardActor(2)*, *CardActor(3)*.

VLINGO XOOM bietet die folgenden Aktionen zur Wiederherstellung des Actors an:

- Wiederaufnahme des Betriebs des Actors ab seiner nächsten Nachricht
- Neustart des Actors
- Den Actor stoppen, was bedeutet, dass auch alle seine untergeordneten Actor gestoppt werden
- Eskalation der Wiederherstellung auf eine Ebene höher [XOO21]

Um die oben genannten Aktionen ausführen zu können, muss der *MainSupervisorActor* die Schnittstelle *Supervisor* mit allen ihren Methoden implementieren.

Auflistung 3.5: *Supervisor*-Schnittstelle

```
public interface Supervisor {
    void inform(Throwable var1, Supervised var2);

    SupervisionStrategy supervisionStrategy();

    default Supervisor supervisor() {
        return new Supervisor() {
            public void inform(Throwable throwable, Supervised supervised) {
                SupervisionStrategy strategy = DefaultSupervisor.
                    DefaultSupervisionStrategy;
                supervised.restartWithin(strategy.period(), strategy.intensity(),
                    strategy.scope());
            }

            public SupervisionStrategy supervisionStrategy() {
                return DefaultSupervisor.DefaultSupervisionStrategy;
            }
        };
    }
}
```

- *inform()* ist eine Methode, die zwei Werte als Parameter erhält. Die erste ist *Throwable*, eine Exception, die den Supervisor-Mechanismus auslöst. Der zweite ist *Supervised*, ein Actor, mit dem der Zustand des ausgefallenen Actors verwaltet werden kann
- *supervisorStrategy()* ist eine Methode, die SupervisionStrategy mit bestimmten Parametern im Rückgabetyt empfängt:
 - *intensity*
 - *period*
 - *scope*
- *supervisor()* gibt mit dem Schlüsselwort *this* den aktuellen Supervisor, d.h. sich selbst, zurück. [XOO21]

Wenn entschieden wird, die Neustart-Aktion zu verwenden, kann die Supervisor dies eine bestimmte Anzahl von Malen innerhalb einer bestimmten Zeitspanne tun.

- *intensity* ist eine Anzahl der Male, die der Actor neu gestartet werden muss
- *period* ist ein Zeitintervall in Millisekunden, in dem der Actor neu gestartet werden soll
- *scope* ist eine Strategie, entweder One-For-One oder All-For-One

Auflistung 3.6: Erstellen von *SupervisionStrategy*

```
@Override
public SupervisionStrategy supervisionStrategy () {
    return new SupervisionStrategy() {
        @Override
        public int intensity() {
            return 5;
        }

        @Override
        public long period() {
            return 2000;
        }

        @Override
        public Scope scope() {
            return Scope.One;
        }
    };
}
```

Wie in der Auflistung 3.6 dargestellt, hat der *MainSupervisorActor* die folgenden Werte. *intensity()* - 5, *period()* - 2 Sekunden und *scope()* - One-For-One-Strategie. Anhand dieser Werte wird der Supervisor Maßnahmen für den ausgefallenen Actor ergreifen. Eine weitere Methode, die der Supervisor anwenden muss, ist *inform()*.

Auflistung 3.7: *inform()* Methode in *MainSupervisorActor*

```
@Override
public void inform(Throwable throwable, Supervised supervised) {
    if (throwable.getClass().equals(NullPointerException.class)) {
        supervised.restartWithin(supervisionStrategy().period(),
            supervisionStrategy().intensity(), supervisionStrategy().scope());
    } else if (throwable.getClass().equals(ArithmeticException.class)) {
        supervised.resume();
    } else if (throwable.getClass().equals(IllegalArgumentException.class)) {
        supervised.stop(supervisionStrategy().scope());
    } else {
        supervised.restartWithin(supervisionStrategy().period(),
            supervisionStrategy().intensity(), supervisionStrategy().scope());
    }
}
```

Für die verschiedenen Arten von Exceptions sind unterschiedliche Maßnahmen vorgesehen. Wie in der Auflistung 3.7 gezeigt, wird der *MainSupervisorActor* bei einer *NullPointerException* den Actor mit den Parametern von *SupervisionStrategy* neu starten. Handelt es sich bei der Exception um eine *ArithmeticException*, verwendet der Supervisor die Aktion *Resume*. Bei *IllegalArgumentException* wird die *Stop*-Aktion gewählt und bei allen anderen neu startet.

Wenn *AtmActor* zum Beispiel einen Fehler auslöst, wird *MainSupervisorActor* auf der Grundlage der Strategien in der *inform()*-Methode Maßnahmen durchführen.

In der Klasse *AtmActor* wurde speziell eine Methode erstellt, die eine *NullPointerException* auslöst. Dies ist die Methode *insertCardWithNullPointerException()*.

Auflistung 3.8: Implementierung von *AtmActor*

```
public class AtmActor extends Actor implements Atm {
    private final Atm self;
    private boolean cardAccepted;
    private final TestUntil until;
    private boolean isPinCorrect;
    private Long money;

    public AtmActor(final TestUntil until) {
        this.until = until;
        this.cardAccepted = false;
        this.isPinCorrect = false;
        this.self = selfAs(Atm.class);
        this.money = 0L;
    }

    @Override
    public void insertCard(final CardReader cardReader, final Card card) {
        logger().info("Inserting Card");
        logger().info("money = " + this.money);

        cardReader.readCard(self, card);
    }

    @Override
    public void insertCardWithNullPointer(final CardReader cardReader, final Card
        card) {
        logger().info("Inserting Card");
        logger().info("money = " + this.money);
        this.money = money + 200;
        logger().info("money = " + this.money);
        this.money = null;
    }
}
```

```
        cardReader.readCard(self, card);
    }

    @Override
    public void acceptCard(final CardReader cardReader, final Card card) {
        cardAccepted = true;
        logger().info("Card is accepted: " + cardAccepted);
    }

    @Override
    public void declineCard(final CardReader cardReader, final Card card) {
        cardAccepted = false;
        logger().info("Card is not accepted: " + cardAccepted);
    }

    @Override
    public void verifyPin(String typedPin, final Card card, final CardReader
        cardReader) {
        logger().info("Verifying PIN");
        card.sendPin(typedPin, self, cardReader);
    }

    @Override
    public void acceptPin(final CardReader cardReader, final Card card) {
        logger().info("Accepting PIN");
        isPinCorrect = true;
        logger().info("isPinCorrect " + isPinCorrect);
    }

    @Override
    public void declinePin(final CardReader cardReader, final Card card) {
        logger().info("Declining PIN");
        isPinCorrect = false;
        logger().info("isPinCorrect is " + isPinCorrect);
        self.stop();
    }

    @Override
    public void affordMoney(final Bank bank, long money) {
        logger().info("Affording money");

        bank.sendMoney(self, money);
    }

    @Override
    public void declinedAffordMoney() {
```

```
        logger().info("Declined affording money, balance is " + this.money);
    }

    @Override
    public void addMoney(long money) {
        logger().info("Adding extra money to ATM money, before adding balance is " +
            this.money);
        this.money += money;
        logger().info("After adding balance is " + this.money);
    }

    @Override
    protected void afterStop() {
        logger().info("playground.AtmActor " + address() + " just stopped!");
        until.happened();
        super.afterStop();
    }

    @Override
    protected void afterRestart(final Throwable reason) {
        logger().info("playground.AtmActor " + address() + " just restarted!");
        money = 0L;
        until.happened();
        super.afterRestart(reason);
    }
}
```

Die anderen Methoden werden auch in anderen Verfahren eingesetzt. Anhand der Auflistung 3.8 lässt sich feststellen, dass *AtmActor* die folgenden Dinge tun kann:

1. *insertCard()*
2. *insertCardWithNullPointer()*
3. *acceptCard()*
4. *declineCard()*
5. *verifyPin()*
6. *acceptPin()*
7. *declinePin()*
8. *affordMoney()*
9. *declinedAffordMoney()*
10. *addMoney()*

Jeder Actor ist in der Lage, seinen Zustand zu ändern und damit seinen Lebenszyklus zu steuern. VLINGO XOOM bietet dazu die folgenden Methoden an: [XOO21]

- *beforeStart()* wird aufgerufen, bevor der Actor gestartet wird
- *afterStop()* wird aufgerufen, nachdem der Actor gestoppt wurde
- *beforeRestart(final Throwable reason)* wird aufgerufen, bevor der Actor neu gestartet wird
- *afterRestart(final Throwable reason)* wird aufgerufen, nachdem der Actor neu gestartet wurde
- *beforeResume(final Throwable reason)* wird aufgerufen, vor der Wiederaufnahme des Actors

AtmActor hat zwei Methoden, *afterStop()* und *afterRestart()*. Sie werden nach den entsprechenden Maßnahmen der Supervisor oder des Actors selbst aufgerufen. In Fällen, in denen ein Actor einen Fehler ausgelöst hat und funktionsunfähig geworden ist, bieten die oben genannten Lebenszyklusmethoden die Möglichkeit, das Problem, das die Exception verursacht hat, zu beheben und sich selbst neu zu initialisieren, bevor auf die nachfolgenden Nachrichten reagiert wird. [XOO21]

Wie in Abbildung 3.4 dargestellt, beginnt alles mit der Methode *insertCard()*. Anschließend wird die Methode *readCard()* der Klasse *CardReaderActor* gestartet.

Auflistung 3.9: Implementierung von *CardReaderActor*

```
public class CardReaderActor extends Actor implements CardReader{
    private final CardReader self;
    private final String bankName;

    public CardReaderActor(String bankName) {
        this.self = selfAs(CardReader.class);
        this.bankName = bankName;
    }

    @Override
    public void readCard(final Atm atm, final Card card) {
        logger().info("Reading card inside of CardReader");

        card.sendBankName(atm, self);
    }

    @Override
    public void checkBankName(final Atm atm, final Card card, String bankName) {
        logger().info("Checking name of the bank on Card in CardReader");
    }
}
```

```
        if (this.bankName.equals(bankName)) {
            logger().info("Bank names matched");
            atm.acceptCard(self, card);
        } else {
            logger().info("Bank names did not match");
            atm.declineCard(self, card);
        }
    }
}

@Override
public void checkPin(final Atm atm, final Card card, String pin, String typedPin
) {
    logger().info("Checking pin on Card in CardReader");
    if (pin.equals(typedPin)) {
        logger().info("Pins matched");
        atm.acceptPin(self, card);
    } else {
        logger().info("Pins did not match");
        atm.declinePin(self, card);
    }
}
}
```

CardReaderActor sendet eine Nachricht an *CardActor* und erhält dann eine *checkBankName()* Nachricht. Der Body dieser Methode entscheidet, ob der Name der Bank auf der Karte mit dem Namen der Bank im Geldautomaten übereinstimmt. Nachdem er dies überprüft hat, sendet er eine entsprechende Nachricht an *AtmActor*. Damit ist der in der Abbildung 3.4 dargestellte Vorgang beendet. Das nächste Unterkapitel wird zeigen, wie diese Actors miteinander interagieren.

3.2.4 Geschäftsprozess-Demonstration

Um Tests zu schreiben, ist es notwendig, einen Raum zu schaffen, in dem die Actor miteinander interagieren können. Dazu ist es erforderlich, eine *World* zu erstellen. [XOO21]

Auflistung 3.10: Erstellung von *World*

```
final World world = World.start("actors", configuration);
```

Als Parameter für die Methode *start()* wurde es *actors* angegeben. Dieser Wert muss mit dem Namen des Verzeichnisses übereinstimmen, in dem alle Actor-Klassen gespeichert sind. Auch *configuration* ist das Feld, das in diesem Auflistung 3.4 erstellt wurde.

Auflistung 3.11: Prozess ohne *NullPointerException* Auslösung

```
@Test
public void checkBankNameTest_withoutException() {
    final World world = World.start("actors", configuration);
    final TestUntil until = TestUntil.happenings(1);
    final Atm atm = world.actorFor(Atm.class, AtmActor.class, until);
    final CardReader cardReader = world.actorFor(CardReader.class, CardReaderActor.
        class, "Money Bank");
    final Card card = world.actorFor(Card.class, CardActor.class, "1111", "Money
        Bank");

    atm.insertCard(cardReader, card);

    until.completes();

    world.terminate();
}
```

In der Auflistung 3.11 ist eine Testmethode dargestellt, die zeigt, wie der in Abbildung 3.4 beschriebene Geschäftsprozess funktioniert.

Zu Beginn werden Actors erstellt: *AtmActor*, *CardReaderActor* und *CardActor*. Für jede von ihnen werden die Argumente benötigt, die in ihren Konstruktoren angegeben sind.

Der Geschäftsprozess wird mit der Methode *insertCard()* gestartet und die Actors beginnen mit dem Austausch von Nachrichten. Während der Kommunikation wurden keine Fehler festgestellt und der Vorgang wurde erfolgreich abgeschlossen. Das Folgende wurde in die Konsole geschrieben:

Auflistung 3.12: Konsolenausgabe von Test *checkBankNameTest-withoutException*

```
[pool-2-thread-6] INFO actors.AtmActor - Inserting Card
[pool-2-thread-6] INFO actors.AtmActor - money = 0
[pool-2-thread-6] INFO actors.CardReaderActor - Reading card inside of CardReader
[pool-2-thread-4] INFO actors.CardActor - Sending bank name to CardReader
[pool-2-thread-3] INFO actors.CardReaderActor - Checking name of the bank on Card in
    CardReader
[pool-2-thread-3] INFO actors.CardReaderActor - Bank names matched
[pool-2-thread-5] INFO actors.AtmActor - Card is accepted: true
```

Wie in dieser Auflistung 3.12 zu sehen ist, sind die Banknamen von *CardReaderActor* und *CardActor* identisch - "Money Bank". Daher hat *AtmActor* eine Nachricht erhalten, dass die Karte akzeptiert wurde, und das Feld *cardAccepted* ist *true*.

Die Auflistung 3.12 zeigt, wie dieser Geschäftsprozess funktionieren sollte. Bei der Ausführung können jedoch Fehler auftreten. Actors, die Fehler gemacht haben, wenden sich an den Supervisor, der seinerseits Maßnahmen ergreift.

Die folgende Auflistung 3.13 zeigt eine leicht veränderte Version des Auflistung 3.11. *insertCard()* wurde durch *insertCardWithNullPointerException()* ersetzt, so dass *AtmActor* zur Laufzeit eine *NullPointerException* auslöst.

Auflistung 3.13: Prozess mit *NullPointerException* Auslösung

```
@Test
public void checkBankNameTest_withException() {
    final World world = World.start("actors", configuration);
    final TestUntil until = TestUntil.happenings(1);
    final Atm atm = world.actorFor(Atm.class, AtmActor.class, until);
    final CardReader cardReader = world.actorFor(CardReader.class, CardReaderActor.
        class, "Money Bank");
    final Bank bank = world.actorFor(Bank.class, BankActor.class, 20000);
    final Card card = world.actorFor(Card.class, CardActor.class, "1111", "Money
        Bank");

    atm.insertCardWithNullPointerException(cardReader, card);
    atm.affordMoney(bank, 500);
    pauseThisThreadForSeconds(5);

    atm.verifyPin("1111", card, cardReader);
    pauseThisThreadForSeconds(5);

    atm.affordMoney(bank, 213);
    pauseThisThreadForSeconds(5);

    atm.verifyPin("1121", card, cardReader);
    pauseThisThreadForSeconds(5);

    until.completes();

    world.terminate();
}
```

In der Auflistung 3.13 wurden *World*, *TestUntil*, *AtmActor*, *CardReaderActor*, *BankActor*, *CardActor* erstellt. Anschließend wird die Methode *insertCardWithNullPointerException()* aufgerufen, die später eine *NullPointerException* verursacht. Direkt danach wird die Methode *affordMoney()* aufgerufen, die tatsächlich eine *NullPointerException* auslöst. Der *AtmActor* verwendet die Methode *affordMoney()*, um die *BankActor* um Geld zu bitten. Nachdem der Fehler aufgetreten ist, wird die Methode *verifyPin()* mit dem richtigen PIN-Code auf-

gerufen, um sicherzustellen, dass der Selbstheilungsmechanismus im *MainSupervisorActor* funktioniert hat. Die Methode *affordMoney()* wird dann erneut aufgerufen und es wird erwartet, dass das Programm nicht erneut eine *NullPointerException* auslöst. Der Test wird durch den Aufruf der Methode *verifyPin()* mit einem bereits falschen PIN-Code beendet, so dass das Test abbricht.

Das Ergebnis dieses Tests wurde in der folgende Auflistung 3.14 dargestellt.

Auflistung 3.14: Konsolenausgabe von Test *checkBankNameTest-withException*

```
[pool-2-thread-5] INFO actors.AtmActor - Inserting Card
[pool-2-thread-5] INFO actors.AtmActor - money = 0
[pool-2-thread-5] INFO actors.AtmActor - money = 200
[pool-2-thread-5] INFO actors.CardReaderActor - Reading card inside of CardReader
[pool-2-thread-5] INFO actors.AtmActor - Affording money
[pool-2-thread-5] INFO actors.CardActor - Sending bank name to CardReader
[pool-2-thread-5] INFO actors.BankActor - Sending money to ATM
[pool-2-thread-5] INFO actors.CardReaderActor - Checking name of the bank on Card in
    CardReader
[pool-2-thread-5] INFO actors.CardReaderActor - Bank names matched
[pool-2-thread-4] INFO actors.AtmActor - Card is accepted: true
[pool-2-thread-2] INFO actors.BankActor - APPROVED: 500 will be sent to ATM
[pool-2-thread-3] INFO actors.AtmActor - Adding extra money to ATM money, before
    adding balance is null
[pool-2-thread-5] ERROR actors.AtmActor - Default before restart recovery after:
    null
java.lang.NullPointerException
    at actors.AtmActor.addMoney(AtmActor.java:88)
    at actors.Atm__Proxy.lambda$addMoney$70c2d385$1(Atm__Proxy.java:130)
    at io.vlingo.xoom.actors.LocalMessage.internalDeliver(LocalMessage.java:121)
    at io.vlingo.xoom.actors.LocalMessage.deliver(LocalMessage.java:53)
    at io.vlingo.xoom.actors.plugin.mailbox.concurrentqueue.
        ConcurrentQueueMailbox.run(ConcurrentQueueMailbox.java:108)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(
        ThreadPoolExecutor.java:1128)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(
        ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:834)
[pool-2-thread-2] INFO actors.AtmActor - playground.AtmActor Address[id=18, name=(
    none)] just stopped!
[pool-2-thread-3] INFO actors.AtmActor - playground.AtmActor Address[id=18, name=(
    none)] just restarted!
[pool-2-thread-3] ERROR actors.AtmActor - Default after restart recovery after: null
java.lang.NullPointerException
    at actors.AtmActor.addMoney(AtmActor.java:88)
    at actors.Atm__Proxy.lambda$addMoney$70c2d385$1(Atm__Proxy.java:130)
    at io.vlingo.xoom.actors.LocalMessage.internalDeliver(LocalMessage.java:121)
```

```
at io.vlingo.xoom.actors.LocalMessage.deliver(LocalMessage.java:53)
at io.vlingo.xoom.actors.plugin.mailbox.concurrentqueue.
    ConcurrentQueueMailbox.run(ConcurrentQueueMailbox.java:108)
at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(
    ThreadPoolExecutor.java:1128)
at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(
    ThreadPoolExecutor.java:628)
at java.base/java.lang.Thread.run(Thread.java:834)
[pool-2-thread-3] ERROR actors.AtmActor - Default before resume recovery after: null
java.lang.NullPointerException
    at actors.AtmActor.addMoney(AtmActor.java:88)
    at actors.Atm__Proxy.lambda$addMoney$70c2d385$1(Atm__Proxy.java:130)
    at io.vlingo.xoom.actors.LocalMessage.internalDeliver(LocalMessage.java:121)
    at io.vlingo.xoom.actors.LocalMessage.deliver(LocalMessage.java:53)
    at io.vlingo.xoom.actors.plugin.mailbox.concurrentqueue.
        ConcurrentQueueMailbox.run(ConcurrentQueueMailbox.java:108)
    at java.base/java.util.concurrent.ThreadPoolExecutor.runWorker(
        ThreadPoolExecutor.java:1128)
    at java.base/java.util.concurrent.ThreadPoolExecutor$Worker.run(
        ThreadPoolExecutor.java:628)
    at java.base/java.lang.Thread.run(Thread.java:834)
[pool-2-thread-4] INFO actors.AtmActor - Verifying PIN
[pool-2-thread-5] INFO actors.CardActor - Sending pin to CardReader
[pool-2-thread-3] INFO actors.CardReaderActor - Checking pin on Card in CardReader
[pool-2-thread-3] INFO actors.CardReaderActor - Pins matched
[pool-2-thread-2] INFO actors.AtmActor - Accepting PIN
[pool-2-thread-2] INFO actors.AtmActor - isPinCorrect true
[pool-2-thread-3] INFO actors.AtmActor - Affording money
[pool-2-thread-3] INFO actors.BankActor - Sending money to ATM
[pool-2-thread-3] INFO actors.BankActor - APPROVED: 213 will be sent to ATM
[pool-2-thread-3] INFO actors.AtmActor - Adding extra money to ATM money, before
    adding balance is 0
[pool-2-thread-3] INFO actors.AtmActor - After adding balance is 213
[pool-2-thread-1] INFO actors.AtmActor - Verifying PIN
[pool-2-thread-1] INFO actors.CardActor - Sending pin to CardReader
[pool-2-thread-1] INFO actors.CardReaderActor - Checking pin on Card in CardReader
[pool-2-thread-1] INFO actors.CardReaderActor - Pins did not match
[pool-2-thread-1] INFO actors.AtmActor - Declining PIN
[pool-2-thread-4] INFO actors.AtmActor - isPinCorrect is false
[pool-2-thread-4] INFO actors.AtmActor - playground.AtmActor Address[id=18, name=(
    none)] just stopped!
```

Der Test beginnt mit der Operation “Karte einstecken”. In der Methode *insertCardWithNullPointer()* gibt es eine Zeile, die dem Feld *money* den Wert *null* gibt (Auflistung 3.8). Das Geldfeld wird verwendet, wenn die Bank versucht, das Guthaben am Geldauto-

maten zu aktualisieren. Und wenn *AtmActor* die Nachricht “add null amount to balance” erhält, wird ein Fehler erzeugt, weil das Feld “money” *null* ist.

Wie in der Auflistung 3.14 gezeigt, verfügt der *MainSupervisorActor* über Strategien zur Wiederherstellung, wenn eine Exception auftritt. Wenn eine *NullPointerException* auftritt, startet *MainSupervisorActor* den Actor neu, der die Exception ausgelöst hat, in diesem Fall *AtmActor*. *AtmActor* hat eine Methode *afterRestart()*, die ausgelöst wird, wenn der Actor neu startet. Es gibt eine Zeile, in der dem Feld *money* der Wert 0 zugewiesen wird. Auf diese Weise kann *AtmActor* mit der Bearbeitung anderer Nachrichten fortfahren und es tritt kein Fehler bei *addMoney()* auf. Die Auflistung 3.14 zeigt, dass *AtmActor* nach dem Neustart andere Nachrichten wie *verifyPin()* und *affordMoney* ohne Fehler verarbeitet.

Die Supervisor kann den *AtmActor* zum Beispiel auch stoppen. Dann kann es keine Nachrichten senden und empfangen. Alle anderen Actors, die nicht von *AtmActor* abhängig sind, können jedoch weiterhin funktionieren.

3.2.5 Beispiel von Fehlerbaumanalyse in ATM-Software

Wie im Unterkapitel “Fehlerbaumanalyse” 2.1.5 in den theoretischen Grundlagen beschrieben, ist die Fehlerbaumanalyse eine effektive Methode, um die Ursachen von Fehlern in einem System zu identifizieren. Bei der Anwendung dieser Technik auf eine prototypische Implementierung wurde der folgende Fehlerbaum erstellt:

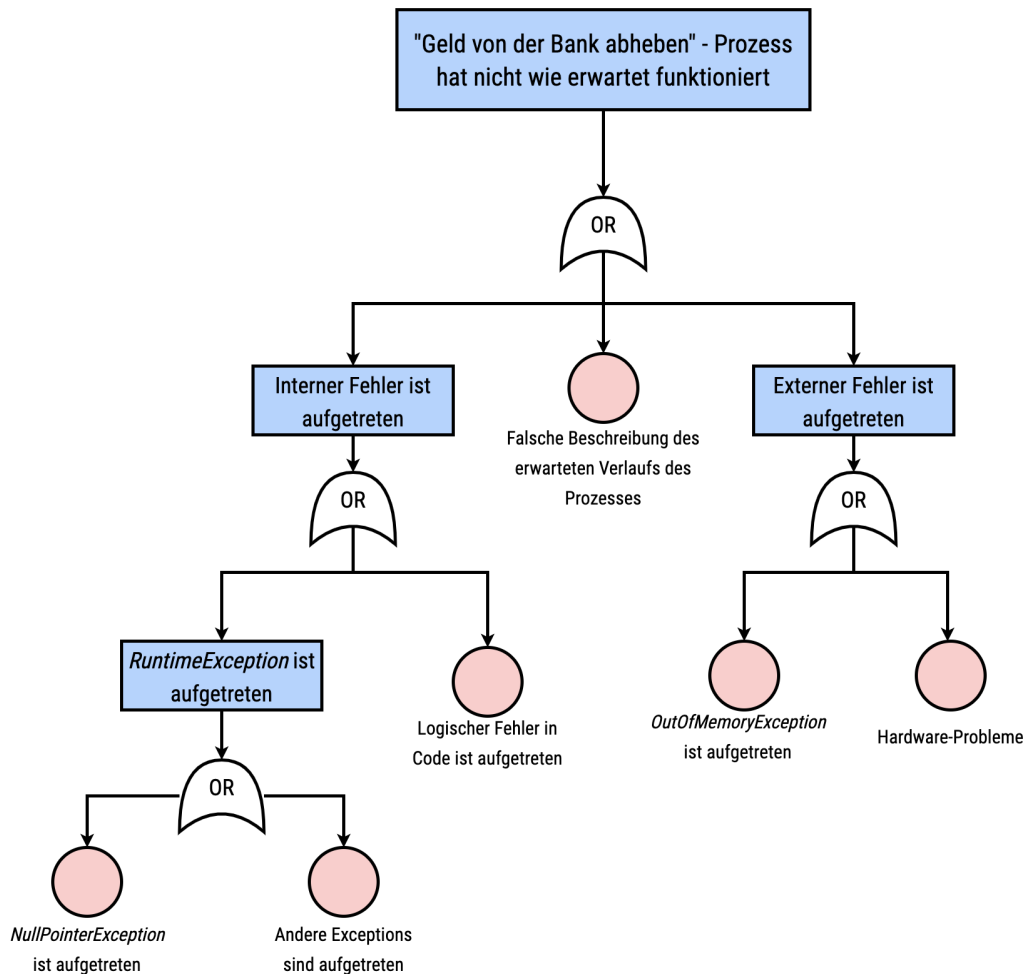


Abbildung 3.8: Fehlerbaumanalyse des ATM-Softwares

An der Spitze des Fehlerbaums steht das Top-Ereignis, das Problem, dessen Ursachen ermittelt werden müssen. In diesem Baum ist das oberste Ereignis "Geld von Der Bank abheben"-Prozess hat nicht wie erwartet funktioniert". Wie in der Auflistung 3.14 gezeigt, stößt *AtmActor* beim Empfang der Nachricht *affordMoney()* auf einen Fehler und bricht mit einem unerwarteten Ergebnis ab.

Um die Ursache zu finden, wurde dieser Baum in der Abbildung 3.8 erstellt. Auf das Top-Ereignis folgt einer von drei Wegen. Entweder liegt das Problem darin, dass der korrekte Prozessablauf "Geld von der Bank abfragen" nicht korrekt dokumentiert wurde, oder es wurde ein externer oder interner Fehler festgestellt.

Ein externer Fehler ist z. B. ein Mangel an RAM oder CPU-Leistung. Es kann sein, dass ein Teil der Hardware ist ausgefallen und funktioniert nicht mehr.

Ein interner Fehler ist softwarebedingt. Es gibt zwei Möglichkeiten: entweder wurde eine *RuntimeException* ausgelöst, oder es gibt logische Fehler im Code, die vom Programmierer programmiert wurden.

Die *RuntimeException* wird weiter unterteilt in die häufigste *NullPointerException* und andere Exceptions, die in diesem Fall weniger wahrscheinlich sind.

Durch die Analyse des Fehlerbaums wurden mögliche Ursachen für das Ereignis “Geld von Der Bank abfragen”-Prozess hat nicht wie erwartet funktioniert” ermittelt. Tatsächlich war der Grund ein logischer Fehler im Code. Es gibt eine Zeile in der Methode *insertCardWithNullPointerException()*, in der das Feld *money* auf *null* gesetzt wurde.

Kapitel 4

Auswertung

Dieses Kapitel besteht aus einem Unterkapitel. Das Unterkapitel “Erfüllung der Systemanforderungen” 4.1 enthält eine Beschreibung der funktionalen und nicht-funktionalen Anforderungen.

4.1 Erfüllung von Systemanforderungen

Zur Beurteilung des Konzepts und der prototypischen Implementierung des Beispielprojekts sollten die im Unterkapitel “Systemanforderungen für die Umsetzung” 3.1.2 beschriebenen Anforderungen berücksichtigt werden. Durch die Analyse und Zusammenfassung der einzelnen Anforderungen wird festgestellt, ob das implementierte System die Anforderungen erfüllt.

4.1.1 Funktionale Anforderungen

Es wurde eine Beschreibung für jede funktionale Anforderung bereitgestellt.

Das System sollte selbst mit Fehlern umgehen.

Das System ist in der Lage, Fehler dank der in den Supervisors vorgeschriebenen Wiederherstellungsmechanismen selbständig zu beheben. Außerdem dank der im Unterkapitel “Implementierung von Actors” vorgestellten Methoden des Lebenszyklus-Managements.

Das System sollte auf viele Standardfehler vorbereitet sein und über Strategien zur Wiederherstellung verfügen.

VLINGO XOOM bietet die Möglichkeit, Supervisors zu erstellen und Strategien zur Behebung von Standardfehlern vorzuschreiben.

Das System sollte in einfache Komponenten mit jeweils nur einer Zuständigkeit unterteilt werden.

Während der Entwurfsphasen des Systems und der verschiedenen Diagramme im Beispiel Aktivitätsdiagramm 3.2 können die einzelnen Komponenten des Systems identifiziert werden. Bei der Umsetzung wurden sechs Actors festgelegt, die in eigener Verantwortung stehen. Durch das Zusammenspiel dieser einfachen Komponenten lassen sich so komplexe Geschäftsprozesse realisieren.

Das System sollte kritische Ausfälle vermeiden, indem es schnell und zeitnah auf Fehler reagiert.

Wie in der Auflistung 3.13 zu sehen ist, war das System in der Lage, selbständig und rechtzeitig auf eine *NullPointerException* zu reagieren und dadurch kritische Fehler zu vermeiden. Auflistung 3.14 zeigt, wie das System nach der Exception wieder in einen gesunden Zustand zurückkehrt.

Das System sollte die Patterns “Simple Component”, “Let-It-Crash” und “Circuit Breaker” implementieren.

Das Pattern “Simple Component” wurde verwendet, um die vorherige Anforderung zu erfüllen.

Das “Let-It-Crash”-Pattern wurde in der *inform()* Methode des *MainSupervisorActor* implementiert. Die Grundidee dieses Patterns ist, dass im Falle eines nicht erwarteten Fehlers die Komponente, die den Fehler verursacht hat, neu gestartet wird.

Das “Circuit Breaker”-Pattern wurde im Rahmen der prototypischen Implementierung nicht umgesetzt. VLINGO XOOM verfügt im Gegensatz zum Akka-Framework nicht über eine Implementierung dieses Patterns. Laut Vaughn Vernon, dem Begründer des VLINGO XOOM-Frameworks und Autor zahlreicher Bücher, ist das “Circuit Breaker”-Pattern in der aktuellen Version des Frameworks nicht implementiert, könnte aber in zukünftigen Versionen erscheinen.⁵

Das System sollte den Pin-Code der Karte überprüfen.

Das Beispielprojekt implementiert den Prozess der Überprüfung des Pin-Codes der Karte.

Das System sollte den Namen der Bank von der Karte prüfen, ob die Karte von diesem Geldautomaten bedient wird.

Ein Beispielprojekt implementiert das Prozess der Namensprüfung der Bank.

⁵Diskussion auf der Plattform Slack Channel: vlingo-xoom-community. Link zur Nachricht: https://vlingo-xoom-community.slack.com/archives/CQ5LNC465/p1643474213776289?thread_ts=1643473388.611089cid=CQ5LNC465

Das System sollte in der Lage sein, Geld von der Bank anzufordern.

Ein Beispielprojekt implementiert den Prozess der Anforderung von Bargeld bei der Bank.

4.1.2 Nicht funktionale Anforderungen

Beide nicht funktionalen Anforderungen wurden mit einer Beschreibung zusammengefasst.

Das System sollte selbstheilend sein.

Das System sollte fehlertolerant sein.

Durch die Verwendung einer reaktiven Systemarchitektur, insbesondere des Actor Models, ist das System selbstheilend und fehlertolerant auf der Anwendungsebene. Wie im Konzept 3.1 beschrieben, kann die Software dank der Selbstheilungsmechanismen des Actor-Systems in kürzester Zeit in einen gesunden Zustand zurückkehren. Fehlertoleranz wird durch die Anwendung eines defensiven Programmierstils und die Anwendung grundlegender Patterns für Fehlertoleranz erreicht.

Das Beispielprojekt zeigt, wie ein System sich selbst wiederherstellen und fehlertolerant sein kann. Daher könnte der Grad der Selbstwiederherstellung und Fehlertoleranz noch höher sein. Offensichtlich ist die implementierte Software nicht in der Lage, den Code während der Laufzeit zu ändern. Für diese Ebene der Selbstheilung sollten andere Instrumente und Verfahren eingesetzt werden.

Kapitel 5

Zusammenfassung

Insgesamt gab die Masterarbeit einen Einblick in die Entwicklung eines effektiven Umgangs mit Fehlern. Ziel der Masterarbeit war es, ein Konzept zu erstellen und einen Prototyp eines selbstheilenden und fehlertoleranten Systems zu implementieren. Um diese Fähigkeiten zu definieren, wurde an den Grundbegriffen “Selbstheilung” und “Fehlertoleranz” untersucht. Wichtige Begriffe wie Störung, Fehler, Ausfall wurden zum besseren Verständnis der Arbeit beschrieben. Der Ansatz der reaktiven Systeme und seine Umsetzung, das Actor Model, wurden ebenfalls berücksichtigt.

In Kapitel 3 wurden die in Kapitel 2 beschriebenen vorläufigen Erkenntnisse angewandt, um ein Konzept und eine prototypische Implementierung eines selbstheilenden und fehlertoleranten Systems zu erstellen. Es wurde ein Beispielprojekt beschrieben, auf dessen Grundlage die Software implementiert wurde. Die Software ist teilweise, insbesondere auf Anwendungsebene, selbstheilend. Außerdem ist die Software durch die Verwendung bestimmter Patterns fehlertolerant. Das Konzept der Anwendung reaktiver Systeme, nämlich das Actor Model, ermöglicht die Selbstheilung von Anwendungen, wie in Kapitel 3 gezeigt wurde. Eine der Ergebnisse der Entwicklung ist die Schaffung eines in der Supervisor vorgeschriebenen Fehlerbehebungsmechanismus. Wie im Kapitel 3.2.4 “Geschäftsprozess-Demonstration” gezeigt, hat die Software auf den Fehler reagiert und konnte Maßnahmen ergreifen, um weiterhin korrekt zu funktionieren.

Schließlich wurde in Kapitel 4 eine Analyse auf der Grundlage der Anforderungen des Beispielprojekts durchgeführt. Auf diese Weise wurde nachgewiesen, dass die implementierte Software auf der Anwendungsebene selbstheilend und bis zu einem gewissen Grad fehlertolerant ist. Aus allem, was in den Kapiteln 3 und 4 beschrieben wurde, wie Software geschaffen werden kann, die sich selbst heilen kann und fehlertolerant ist. Nämlich durch die Anwendung reaktiver Systeme, des Actor Models und Patterns für Fehlertoleranz. Es kann festgestellt werden, dass das Ziel der Arbeit erreicht wurde und dass es noch Raum für weitere Arbeiten und Analysen gibt. Bei der erstellten Software handelt es sich

nicht um eine Komplettlösung. Es bietet nur die Grundlage für weitere Untersuchungen und die Entwicklung von Software mit einem noch höheren Grad an Selbstheilung und Fehlertoleranz.

Literatur

- [Akk] Akka. *Akka Official Website*. URL: <https://akka.io/>. Letzter Besuch: 5.02.2022.
- [Bon+14] Jonas Bonér u. a. *The Reactive Manifesto*. Juli 2014. URL: <https://www.reactivemanifesto.org>. Letzter Besuch: 28.01.2022.
- [DWH18] Duncan DeVore, Sean Walsh und Brian Hanafée. *Reactive Application Development*. 2018.
- [Eck05] Bruce Eckel. *Thinking in Java (4th Edition)*. Prentice Hall PTR: USA, 2005. ISBN: 0131872486.
- [Ehr+10] Hartmut Ehrig u. a. „Formal Analysis and Verification of Self-Healing Systems“. *FASE*. 2010.
- [EM17] Markus Eisele und Hugh McKee. *Building Reactive Systems Using Akka’s Actor Model and Domain-Driven Design*. Okt. 2017. URL: <https://www.infoq.com/articles/Reactive-Systems-Akka-Actors-DomainDrivenDesign/>. Letzter Besuch: 28.01.2022.
- [Fra] Best Frameworks. *Actor Model Frameworks*. URL: <https://bestframeworks.com/projects/actor-model>. Letzter Besuch: 4.02.2022.
- [Han07] Robert Hanmer. *Patterns for Fault Tolerant Software*. Wiley Publishing, 2007. ISBN: 0470319798.
- [Hes14] Ted Hessing. *What is Fault Tree Analysis*. Juli 2014. URL: <https://sixsigmastudyguide.com/fault-tree-analysis/>. Letzter Besuch: 28.01.2022.
- [Hew22] Carl Hewitt. *Carl Hewitt*. 2022. URL: <https://www.linkedin.com/in/carl-hewitt-a0409738/>. Letzter Besuch: 28.01.2022.

- [jas12] jasonofthel33t. *Hewitt, Meijer and Szyperski: The Actor Model (everything you wanted to know...)* 2012. URL: https://www.youtube.com/watch?v=7erJ1DV_Tlo. Letzter Besuch: 28.01.2022.
- [Jia+07] Michael Jiang u. a. „A Modeling Framework for Self-Healing Software Systems“. 2007.
- [Kag21] Julia Kagan. *Automated Teller Machine (ATM)*. Sep. 2021. URL: <https://www.investopedia.com/terms/a/atm.asp>. Letzter Besuch: 28.01.2022.
- [KHA17] Roland Kuhn, Brian Hanafée und Jamie Allen. *Reactive Design Patterns*. 1st. Manning Publications Co.: USA, 2017. ISBN: 1617291803.
- [MA08] Stefania Montani und Cosimo Anglano. „Achieving self-healing in service delivery software systems by means of case-based reasoning“. *Appl. Intell.* 28 (Apr. 2008), S. 139–152. DOI: **10.1007/s10489-007-0047-1**.
- [McK17] Hugh McKee. *Designing Reactive Systems*. 2017.
- [Opr21] Vlad Oprisa. *Building software for the future*. Okt. 2021. URL: <https://eu.rsystems.com/building-software-for-the-future/>. Letzter Besuch: 21.02.2022.
- [Ray20] Shaan Ray. *Self Healing System Concept, Explained*. Juni 2020. URL: <https://hackernoon.com/self-healing-system-concept-explained-ot6r3w8w>. Letzter Besuch: 17.02.2022.
- [RBW15] Raymond Roestenburg, Rob Bakker und Rob Williams. *Akka in Action*. 1st. Manning Publications Co.: USA, 2015. ISBN: 1617291013.
- [RS15] Enno Ruijters und Mariëlle Stoelinga. „Fault tree analysis: A survey of the state-of-the-art in modeling, analysis and tools“. *Computer Science Review* 15-16 (2015), S. 29–62. ISSN: 1574-0137. DOI: <https://doi.org/10.1016/j.cosrev.2015.03.001>. URL: <https://www.sciencedirect.com/science/article/pii/S1574013715000027>.
- [Tea] MindTools Content Team. *Murphy’s Law Expecting the Unexpected*. URL: https://www.mindtools.com/pages/article/newSTR_MurphyLaw.htm. Letzter Besuch: 16.02.2022.
- [Van19] Ranganath Vangavi. *Self-Healing IT Systems: Less Of A Choice, More A Mandatory Requirement*. Apr. 2019. URL: <https://www.mindtree.com/insights/blog/self-healing-it-systems-less-choice-more-mandatory-requirement>. Letzter Besuch: 26.02.2022.

- [Ver15] Vaughn Vernon. *Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka*. 1st. Addison-Wesley Professional, 2015. ISBN: 0133846830.
- [Xoo] Vlingo Xoom. *VLINGO XOOM Official Website*. URL: <https://vlingo.io/>. Letzter Besuch: 5.02.2022.
- [XOO21] VLINGO XOOM. *Actors*. 2021. URL: <https://docs.vlingo.io/xoom-actors>. Letzter Besuch: 28.01.2022.