Technische
Hochschule
**Rosenheim**
Technical University of Applied Sciences

Fakultät für Informatik

Studiengang Informatik

# Serialisierer im Akka Persistence Umfeld

Bachelor Thesis

von

**Maximilian Bundscherer**

Datum der Abgabe: 18.03.2019
Erstprüfer: Prof. Dr. Korbinian Riedhammer
Zweitprüfer: Prof. Dr. Gerd Beneken

Erklärung

Ich versichere, dass ich diese Arbeit selbständig angefertigt, nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Rosenheim, den 18.03.2019

Maximilian Bundscherer

# Abstract

Diese Bachelorarbeit beschäftigt sich mit der Serialisierung bzw. Deserialisierung im Akka Persistence Umfeld. Akka Persistence ist eine Erweiterung für Akka Actors, eine Implementierung des Aktorenmodells. Akka Persistence wird im Bereich Event Sourcing eingesetzt und lässt die Einbindung unterschiedlicher Serialisierer/De-Serialisierer zu.

Um den aktuellen Zustand durch Events im System abbilden und wiederherstellen zu können, benötigt Akka einen kompatiblen und geeigneten Serialisierer/De-Serialisierer. Der verwendete Serialisierer/De-Serialisierer sollte sowohl schnell[1] als auch praxistauglich[2] sein. Daher stellt sich die Frage, welcher Serialisierer/De-Serialisierer geeignet ist.

Um auf die Fragestellung eingehen zu können, werden exemplarisch drei verschiedene Serialisierer/De-Serialisierer miteinander verglichen:

1. Java Serialisierer/De-Serialisierer (Java-Standardserialisierung[3])
2. JSON Serialisierer/De-Serialisierer (Circe[4])
3. Google Protocol Buffers[5] Serialisierer/De-Serialisierer ( ScalaPB[6])

Die Arbeit hat gezeigt, dass die Praxistauglichkeit der Serialisierer/De-Serialisierer stark vom Anwendungsfall abhängig ist. Daher beschäftigt sich diese Bachelorarbeit mit dem generellen Arbeiten mit Serialisierer/De-Serialisierer im Akka Persistence Umfeld; mit dem konkreten Arbeiten mit den oben aufgeführten Serialisierer/De-Serialisierer in diesem Umfeld und zeigt mögliche Bewertungskriterien bezüglich der Praxistauglichkeit auf.

Die Arbeit hat auch gezeigt, dass das Kriterium Schnelligkeit in der Praxis nicht von Relevanz ist.

Schlagworte: Akka, Akka Persistence, Serialisierung, Event Sourcing, Java, Scala, Circe, Google Protocol Buffers, SerDes

1 siehe Definition Schnelligkeit im Abschnitt 1.4
2 siehe Definition Praxistauglichkeit im Abschnitt 1.4
3 siehe `http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_17_010.htm`
4 siehe `https://github.com/circe/circe`
5 siehe `https://developers.google.com/protocol-buffers/`
6 siehe `https://github.com/scalapb/ScalaPB`

# Inhaltsverzeichnis


**1 Einleitung** **1**
- 1.1 Motivation . . . 1
- 1.2 Ziel der Arbeit . . . 2
- 1.3 Vorgehen . . . 2
- 1.4 Definitionen . . . 2
- 1.5 Abgrenzung . . . 3

**2 Beschreibung Umgebung und Technologie** **5**
- 2.1 Aktorenmodell . . . 5
- 2.2 Event Sourcing . . . 7
- 2.3 Akka . . . 8
  - 2.3.1 Akka Actors . . . 9
  - 2.3.2 Akka Persistence . . . 10
- 2.4 Aufbau und Ablauf der Experimente . . . 14
  - 2.4.1 Experiment E1 *Vollständige Umgebung* . . . 20
  - 2.4.2 Experiment E2 *Benchmark Umgebung* . . . 23

**3 Serialisierer/De-Serialisierer im Akka Persistence Umfeld** **27**
- 3.1 Java Serialisierer/De-Serialisierer (Java-Standardserialisierung) . . . 33
- 3.2 JSON Serialisierer/De-Serialisierer (Circe) . . . 35
- 3.3 Google Protocol Buffers Serialisierer/De-Serialisierer (ScalaPB) . . . 37

**4 Fazit** **41**
- 4.1 Ergebnisse aus den Durchführungen der Experimente . . . 41
- 4.2 Schnelligkeit der Serialisierer/De-Serialisierer . . . 43
- 4.3 Praxistauglichkeit der Serialisierer/De-Serialisierer . . . 44

**A Anhänge** **47**
- A.1 Auszüge aus den Eigenschaften der Referenzsysteme . . . 47
- A.2 Versionen der verwendeten Komponenten . . . 48
- A.3 Ergebnisse aus den Durchführungen der Experimente . . . 49

**Literaturverzeichnis** **53**

# Abbildungsverzeichnis

2.1 Exemplarische Darstellung von drei Aktoren in einem Aktorensystem . . . . . 6
2.2 Exemplarische Darstellung des FIFO-Prinzips . . . . . 6
2.3 Exemplarische Darstellung des CQRS-Prinzips . . . . . 7
2.4 Paketübersicht Quellcode der Experimente . . . . . 15
2.5 Sequenzdiagramm von Experiment E1 *Vollständige Umgebung* . . . . . 21
2.6 Klassendiagramm von Experiment E2 *Benchmark Umgebung* . . . . . 25

A.1 Visualisierung der Ergebnisse Durchführung 1 (Referenzsystem R1 *Windows 10*) 49
A.2 Visualisierung der Ergebnisse Durchführung 1 (Referenzsystem R2 *iMac*) . . . . 50
A.3 Visualisierung der Ergebnisse Durchführung 1 (Referenzsystem R3 *AWS EC2*) . . 51
A.4 Visualisierung der Ergebnisse Durchführung 2 (Referenzsystem R3 *AWS EC2*) . . 52

# Tabellenverzeichnis

Auszug der Eigenschaften des Referenzsystem R1 *Windows 10* . . . . . 47
Auszug der Eigenschaften des Referenzsystem R2 *iMac* . . . . . 47
Auszug der Eigenschaften des Referenzsystem R3 *AWS EC2* . . . . . 47
Versionen der verwendeten Komponenten . . . . . 48
Ergebnisse Durchführung 1 (Referenzsystem R1 *Windows 10*) . . . . . 49
Parameter Durchführung 1 (Referenzsystem R1 *Windows 10*) . . . . . 49
Ergebnisse Durchführung 1 (Referenzsystem R2 *iMac*) . . . . . 50
Parameter Durchführung 1 (Referenzsystem R2 *iMac*) . . . . . 50
Ergebnisse Durchführung 1 (Referenzsystem R3 *AWS EC2*) . . . . . 51
Parameter Durchführung 1 (Referenzsystem R3 *AWS EC2*) . . . . . 51
Ergebnisse Durchführung 2 (Referenzsystem R3 *AWS EC2*) . . . . . 52
Parameter Durchführung 2 (Referenzsystem R3 *AWS EC2*) . . . . . 52

## Abkürzungsverzeichnis

**ES** Event Sourcing

**SerDes** Serialisierer/De-Serialisierer

**JSON** JavaScript Object Notation

**CQS** Command-Query-Separation

**CQRS** Command-Query-Responsibility-Segregation

**Protobuf** Protocol Buffers

**JVM** Java Virtual Machine

**JNI** Java Native Interface

**GC** Garbage Collection

**FIFO** First In – First Out

**JOS** Java Object Serialization

# 1 Einleitung

Diese Bachelorarbeit beschäftigt sich mit der Serialisierung bzw. Deserialisierung im Akka Persistence Umfeld. Akka Persistence ist eine Erweiterung für Akka Actors, eine Implementierung des Aktorenmodells. Akka Persistence wird im Bereich Event Sourcing eingesetzt und lässt die Einbindung unterschiedlicher Serialisierer/De-Serialisierer (SerDes) zu.

Um den aktuellen Zustand durch Events im System abbilden und wiederherstellen zu können, benötigt Akka einen kompatiblen und geeigneten SerDes. Der verwendete SerDes sollte sowohl schnell[1] als auch praxistauglich[2] sein. Daher stellt sich die Frage, welcher SerDes geeignet ist.

Um auf die Fragestellung eingehen zu können, werden exemplarisch drei verschiedene SerDes miteinander verglichen:

1. Java SerDes (Java-Standardserialisierung[3]) (siehe Abschnitt 3.1)
2. JSON SerDes (Circe[4]) (siehe Abschnitt 3.2)
3. Google Protocol Buffers[5] SerDes ( ScalaPB[6]) (siehe Abschnitt 3.3)

## 1.1 Motivation

Akka Persistence kann im Bereich Event Sourcing (ES) eingesetzt werden [Akke]. Um den aktuellen Zustand durch Events im System abbilden und wiederherstellen zu können, benötigt das Toolkit einen kompatiblen und geeigneten SerDes [Akkh]. Events werden bei der Implementierung auf eine Klasse abgebildet. Das einzelne Event ist also ein Objekt zur Laufzeit. Um diese Objekte speichern zu können, müssen diese erst auf eine Byte-Folge abgebildet (serialisiert) werden.

Die Struktur eines Events kann sich im Laufe eines Software-Lebenszyklus ändern[7]. Damit muss auch die Klasse in der Implementierung eines Events angepasst werden. Ein geeigneter SerDes sollte die Byte-Folge, trotz Anpassung der Event-Ursprungsklasse, wieder korrekt auf ein verwendbares und gültiges Objekt der modifizierten neuen Event-Klasse abbilden (deserialisieren) können.

1 siehe Definition Schnelligkeit im Abschnitt 1.4
2 siehe Definition Praxistauglichkeit im Abschnitt 1.4
3 siehe `http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_17_010.htm`
4 siehe `https://github.com/circe/circe`
5 siehe `https://developers.google.com/protocol-buffers/`
6 siehe `https://github.com/scalapb/ScalaPB`
7 zum Beispiel durch eine Änderung der Geschäftslogik

## 1.2 Ziel der Arbeit

Ein Entwickler, der mit Akka Persistence arbeitet, muss sich mit der Wahl von einem SerDes für seinen Anwendungsfall auseinandersetzen. Da die Wahl von vielen Kriterien abhängt, soll diese Arbeit dem Leser einen Überblick verschaffen. Anschließend soll der Leser selbstständig entscheiden können, welcher SerDes für seinen Anwendungsfall geeignet ist.

Die Arbeit hat gezeigt, dass die Praxistauglichkeit der SerDes stark vom Anwendungsfall abhängig ist. Daher beschäftigt sich diese Bachelorarbeit mit dem generellen Arbeiten mit SerDes im Akka Persistence Umfeld; mit dem konkreten Arbeiten mit den vorher aufgeführten SerDes in diesem Umfeld und zeigt mögliche Bewertungskriterien bezüglich der Praxistauglichkeit auf.

## 1.3 Vorgehen

Da es viele unterschiedliche Arten und Implementierungen von SerDes gibt, fokussiert sich diese Arbeit auf die Einbindung, Konfiguration und das praktische Arbeiten mit diesen.

Um auf die Fragestellung eingehen zu können, werden exemplarisch drei verschiedene SerDes miteinander verglichen:

- **Standard Java SerDes** als Vertreter der Standardkonfiguration von Akka Persistence. Dieser SerDes sollte nicht in einer Produktivumgebung verwendet werden (mehr dazu im Abschnitt 3.1).
- **JSON SerDes (Circe)** als Vertreter eines SerDes, dessen Format vom Menschen lesbar ist.
- **Google Protocol Buffers SerDes (ScalaPB)** als Vertreter eines SerDes, dessen Format nicht vom Menschen lesbar ist.

Diese Arbeit beschreibt zunächst die nötigen Grundlagen; die verwendete Technologie & Umgebung und stellt die drei SerDes in Form von zwei Experimenten (Experiment E1 *Vollständige Umgebung* und Experiment E2 *Benchmark Umgebung*) gegenüber.

Um einen genaueren Vergleich zu ermöglichen, werden die zwei Experimente mehrmals mit unterschiedlichen Test-Parametern auf drei unterschiedlichen Referenzsystemen (Referenzsystem R1 *Windows 10*, Referenzsystem R2 *iMac* und Referenzsystem R3 *AWS EC2*) für jeden der drei SerDes durchgeführt.

## 1.4 Definitionen

In dieser Arbeit wird auf mehrere Definitionen zurückgegriffen, die nicht allgemein konkret genug sind. Daher werden diese genauer definiert:

- **Schnelligkeit**: In einer messbaren Zeit kann eine fest definierte Menge von Operationen durchgeführt werden. Bei unterschiedlichen Parametern (zum Beispiel unterschiedlichen Implementierungen) kann diese Zeit miteinander verglichen werden.

Ein System ist im Vergleich zu einem anderen System schneller, wenn es die gleiche Menge an Operationen in einer kürzeren Zeit durchführen kann.

- **Praxistauglichkeit**: Unter diesem Begriff versteht der Autor ob eine verwendete Komponente sinnvoll (ohne viele Anpassungen) in der Praxis verwendet werden kann.

Es wird außerdem auf weitere Definitionen zurückgegriffen, die zum besseren Verständnis erklärt werden:

- **Serialisierung** bezeichnet es, ein Objekt in eine Byte-Folge umzuwandeln [Kru09].
- **Deserialisierung** ist die Umkehrung der Serialisierung, bei der aus einer Byte-Folge wieder ein Objekt erzeugt wird.
- **Binärkompatibel** bezeichnet die Eigenschaft, wenn eine Information, die in Binärcode vorliegt, auf einem anderen System ohne erneute Kompilierung interpretierbar ist [BUL].

## 1.5 Abgrenzung

Diese Arbeit beschränkt sich bezüglich Akka auf Akka Actors (Abschnitt 2.3.1) und Akka Persistence (Abschnitt 2.3.2), da diese Komponenten den minimalen Aufbau für eine ES getriebene Software in Akka bilden.

Akka Persistence bietet die Anbindung verschiedener Journal- und Snapshot Storage-Plugins an (mehr dazu im Abschnitt 3). Diese haben einen Einfluss auf die Schnelligkeit des Systems. Alle Experimente verwenden, um einen signifikanten Vergleich sicherzustellen, die gleiche Anbindung. Diese Bachelorarbeit geht nicht auf die unterschiedlichen Plugins ein, da dies nicht zur Beantwortung der Fragestellung beiträgt.

Diese Arbeit geht nicht auf die Versionsunterschiede der verwendeten Software-Komponenten ein. Die Versionen der verwendeten Komponenten können dem Abschnitt A.2 entnommen werden.

# 2 Beschreibung Umgebung und Technologie

Akka implementiert mit Akka Actors das Aktorenmodell. Um den Zustand eines Aktors nach dem Neustart wiederherstellen zu können, wird die Erweiterung Akka Persistence verwendet. Um auf die Fragestellung eingehen zu können, werden in diesem Kapitel erst die dafür nötigen Grundlagen geschaffen.

## 2.1 Aktorenmodell

Das Aktorenmodell ist ein Model aus der Informatik für die nebenläufige Programmierung. Das Programm wird dabei in Aktoren unterteilt. Diese Aktoren werden in einem Aktorensystem verwaltet. Aktoren kommunizieren ausschließlich über unveränderbare Nachrichten. Der Zustand eines Aktors ist von außen nicht direkt sichtbar und kann auch nur über Nachrichten abgefragt und modifiziert werden [Ode08]. Die Abbildung 2.1 zeigt eine exemplarische Darstellung von drei Aktoren in einem Aktorensystem. Das Model wurde 1973 das erste Mal von Carl Hewitt, Peter Bishop und Richard Steiger beschrieben [ACM] und ist bei funktionalen Programmiersprachen wie zum Beispiel Erlang stark verbreitet.

#### Beschreibung eines Aktors

Ein Aktor ist eine kleine Verarbeitungseinheit in einem System, dessen Zustand von außen nicht direkt einsehbar oder veränderbar ist. Um mit einem Aktor interagieren zu können, um zum Beispiel dessen Zustand einsehen oder verändern zu können, wird ausschließlich in Form von unveränderbaren Nachrichten mit diesem kommuniziert. Ein Aktor kann Nachrichten empfangen und selbst versenden. Eingehende Nachrichten werden zunächst in dem Postfach des jeweiligen Aktors hinterlegt.

Der Aktor arbeitet sequentiell die eingegangenen Nachrichten aus seinem Postfach ab. Das Postfach verwaltet die Nachrichten in Form einer Warteschlange. Daher arbeitet ein Aktor nach dem First In – First Out (FIFO)-Prinzip. Bei dem FIFO-Prinzip werden Nachrichten in der Reihenfolge abgearbeitet, in der diese eingegangen sind. Die Abbildung 2.2 visualisiert dieses Prinzip.

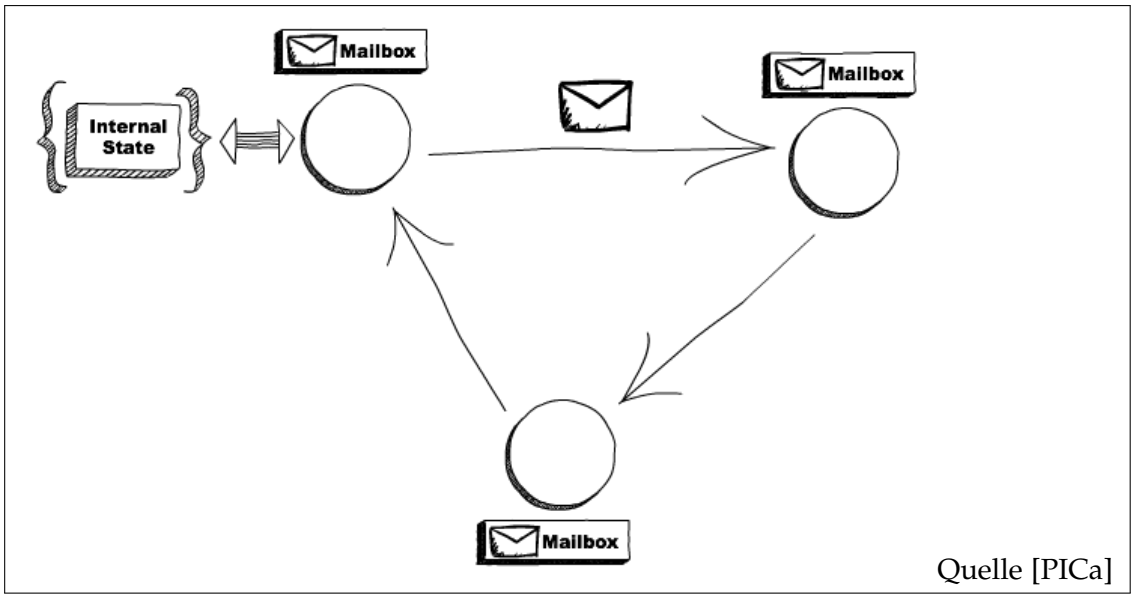


**Abbildung 2.1** Exemplarische Darstellung von drei Aktoren in einem Aktorensystem

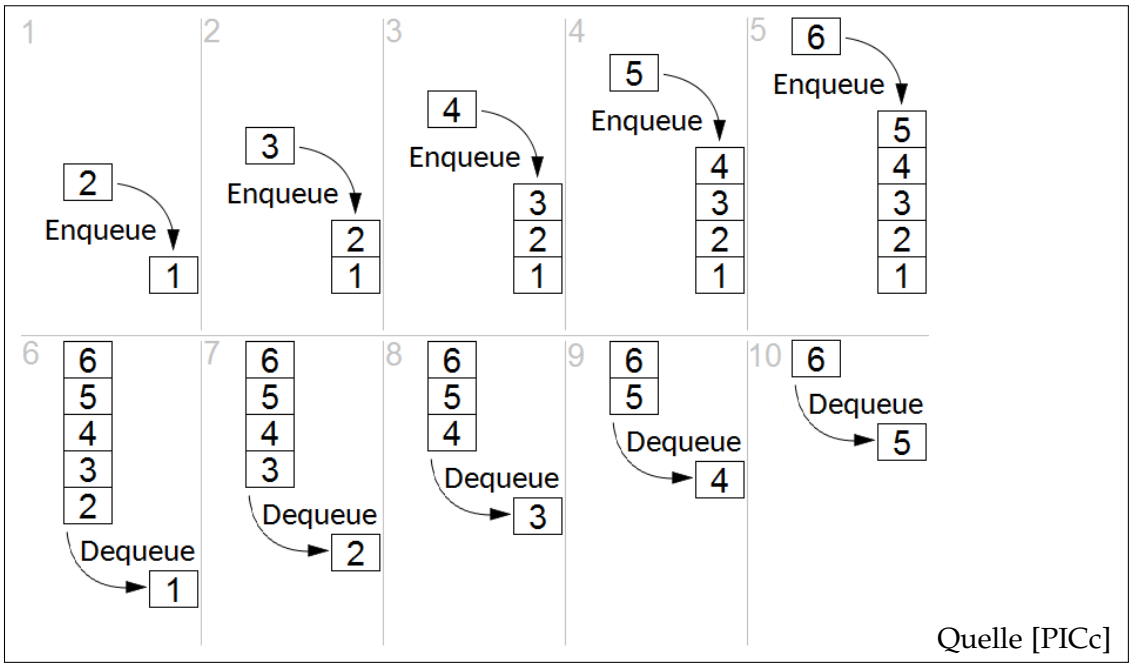


**Abbildung 2.2** Exemplarische Darstellung des FIFO-Prinzips

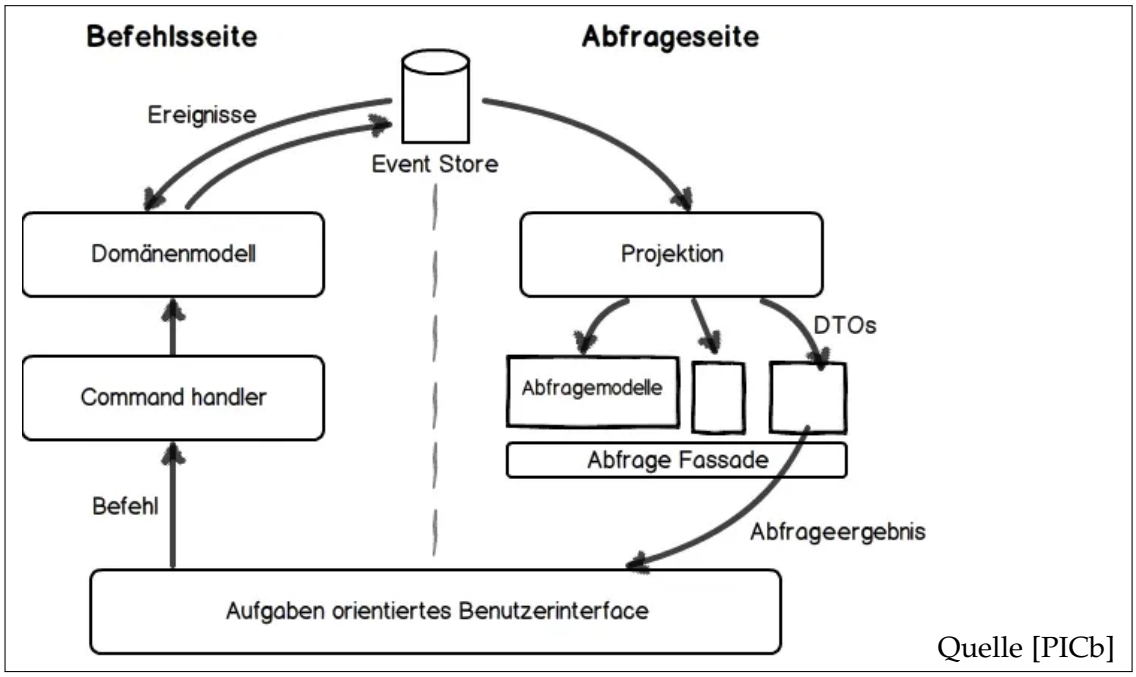


**Abbildung 2.3** Exemplarische Darstellung des CQRS-Prinzips

## 2.2 Event Sourcing

Beim ES werden alle Veränderungen des Zustands eines Systems in Form von Events abgebildet [Pac18]. Durch diese Architekturentscheidung ist es möglich, das komplette System zu jedem Zeitpunkt wiederherstellen zu können. Das unterstützt nicht nur bei Fehlersuche, sondern ermöglicht es auch, besser zu verstehen wie mit dem System gearbeitet wird. Eine Software basierend auf dieser Architektur wird üblicherweise nach dem Command-Query-Responsibility-Segregation (CQRS)-Prinzip (eine Variante des Command-Query-Separation (CQS)-Prinzips) implementiert [Pac18].

#### Beschreibung des Command-Query-Separation-Prinzips

Bei diesem Prinzip wird zwischen zwei verschiedenen Methoden unterschieden:

- **Queries (Abfragen)**: Eine Abfrage liefert Daten zurück und verändert nicht den Zustand [MAR].
- **Commands (Kommandos)**: Ein Kommando verändert den Zustand und liefert keine Daten zurück [MAR].

#### Events im Command-Query-Responsibility-Segregation-Prinzip

Nach dem Erhalt eines Kommandos wird dieses zuerst gegen den aktuellen System-Zustand validiert und bei erfolgreicher Validierung als Event gekennzeichnet und z.B.: in den Event Store, zum Beispiel einer Datenbank, abgebildet [CQR]. Dieses Konzept wird in Abbildung 2.3 (Befehlsseite) visualisiert. Erst nach einer erfolgreichen Validierung wird die Zustandsänderung durchgeführt.

Eine solche Validierung kann exemplarisch folgende Punkte beinhalten:

- Darf der Kommandoersteller diese Aktion durchführen?
- Wird nach der Verarbeitung des Events der Zustand für den Anwendungsfall ungültig?

Ein Kommando (bzw. Event) enthält typischerweise folgende Daten:

- Art des Events (zum Beispiel `Verkaufen`)
- Nutzdaten des Events (zum Beispiel `Fünf Autos`)
- Zeitstempel (zum Beispiel `01. Januar 1980`)

Durch die Speicherung von Events und durch die anschließende chronologische Abarbeitung, ist es möglich jeden durchlaufenden Zustand wiederherzustellen [MIC].

## 2.3 Akka

Akka ist ein Open-Source Toolkit für die Erstellung von parallelisierten, verteilten, ausfallsicheren und nachrichtengesteuerten Anwendungen in Scala und Java [Akkf]. Akka implementiert mit Akka Actors das Aktorenmodell (siehe Abschnitt 2.1) und mit Akka Persistence den ES-Ansatz (siehe Abschnitt 2.2). Akka ist für den Einsatz innerhalb der Java Virtual Machine (JVM)[1] konzipiert und implementiert. Folgende Komponenten sind Bestandteil von Akka [Akkd]:

- **Akka Actors** (wird in dieser Arbeit behandelt)
- Akka Streams
- Akka Http
- Akka Cluster
- Cluster Sharding
- Distributed Data
- **Akka Persistence** (wird in dieser Arbeit behandelt)
- Alpakka
- Akka gRPC
- Commercial Addons
- Akka Management

Wie bereits im Abschnitt 1.5 erwähnt, beschränkt sich diese Arbeit auf Akka Actors und Akka Persistence, da diese Komponenten den minimalen Aufbau für eine ES getriebene Software in Akka bilden.

1 `https://docs.oracle.com/javase/6/docs/technotes/guides/vm/index.html?intcmp=3170`

### 2.3.1 Akka Actors

Akka Actors ist eine Implementierung des Aktorenmodells (siehe Abschnitt 2.1). An der folgenden exemplarischen Implementierung wird verdeutlicht, wie man mit einem Akka Aktor arbeitet. Der `ExampleActor` in dieser Implementierung besitzt einen Zustand in Form eines ganzzahligen Werts (`Int`).

Zuerst werden alle Nachrichten deklariert:

```
object ExampleActor {

  case object  Increment
  case object  Decrement

  case object  WhatIsYourResult
  case class   MyResultIs(value: Int)

}
```

- Nachrichten vom Typ `Increment` können dazu benutzt werden, um den internen Zustand des Aktors um den Wert 1 zu erhöhen. Diese Nachrichten sind **Kommandos**.
- Nachrichten vom Typ `Decrement` können dazu benutzt werden, um den internen Zustand des Aktors um den Wert 1 zu verringern. Diese Nachrichten sind **Kommandos**.
- Nachrichten vom Typ `WhatIsYourResult` können dazu benutzt werden, um über den internen Zustand des Aktors Auskunft zu erhalten. Diese Nachrichten sind **Abfragen**. Der Aktor antwortet auf diese Nachrichten mit einer Nachricht vom Typ `MyResultIs`.

Der `ExampleActor` wird anschließend wie folgt implementiert:

```
import akka.actor._

class ExampleActor extends Actor {

  import ExampleActor._

  var state: Int = 0

  override def receive: Receive = {

    case Increment =>

      state = state + 1

    case Decrement =>

      state = state - 1

    case WhatIsYourResult =>
```

```
        sender ! MyResultIs(value = state)
    }
}
```

- Die Variable `state` beschreibt den Zustand des Aktors in Form eines ganzzahligen Werts (`Int`). Der Wert beträgt bei der Initialisierung des Aktors `0`.
- Über die Funktion `receive` erhält der Aktor die eingehenden Nachrichten (Kommandos oder Abfragen). Mit Hilfe einer Fallunterscheidung wird anschließend unterschieden, um welche Nachrichten es sich handelt.
- Bei der Abfrage `WhatIsYourResult` wird der Versender der Nachricht (`sender`) über den aktuellen Zustand des Aktors in Form einer Nachricht (`MyResultIs(value = state)`) informiert. Der Aktor verfolgt nun nicht weiter die versendete Nachricht und wartet keine Empfangsbestätigung vom `sender` ab. Dieses Verhalten wird in der Informatik als Fire-and-Forget bezeichnet und wird an dieser Stelle mit dem Operator `!` gekennzeichnet.

Der Zustand wird nicht gesichert und geht damit nach dem Beenden des Aktors unwiderruflich verloren. Ein Akka Persistence Aktor kann hingegen seinen Zustand speichern und wiederherstellen (siehe Abschnitt 2.3.2).

### 2.3.2 Akka Persistence

Um den Zustand eines Akka Aktors nach dem Beenden wiederherstellen zu können, kann Akka Persistence verwendet werden [Akkg]. Akka Persistence ist eine Erweiterung für Akka, die es ermöglicht, den Zustand von Akka Persistence Aktoren mithilfe der Speicherung und Verwaltung von Events und optionalen Momentaufnahmen (Snapshots), auch nach dem Beenden der Aktoren wiederherzustellen zu können. Damit ist Akka Persistence eine Implementierung des ES-Ansatzes [Akke].

Ein Akka Persistence Aktor verhält sich von außen betrachtet wie ein Akka Aktor. Alle Änderungen des Zustands werden auf Events und Snapshots abgebildet. Events werden durch Akka Persistence im **Journal** verwaltet, was im ES-Ansatz den Event Store darstellt (siehe Abschnitt 2.2). Snapshots werden im **Snapshot Storage** verwaltet. Die optionalen Snapshots dienen nur der Geschwindigkeitsoptimierung bei der Wiederherstellung des letzten Zustands eines Akka Persistence Aktors.

Wenn ein Akka Persistence Aktor startet, befindet sich dieser erst im `receiveRecover`-Modus, also im Wiederherstellungsmodus. In diesem Modus stellt der Aktor zuerst einen vergangenen Zustand über den letzten verfügbaren Snapshot wieder her. Anschließend stellt der Aktor über die Events ab diesem Snapshot seinen letzten gültigen Zustand her. Die Zustellung des Snapshots und der Events erfolgt in der chronologisch korrekten Reihenfolge. Wenn der Zustand beispielsweise über zehn Events abgebildet wurde und ein Snapshot bis zum sechsten Event vorliegt, wird dem Aktor erst dieser Snapshot zugestellt und anschließend die vier Events ab diesem Snapshot.

Nach der Wiederherstellung befindet sich der Aktor im `receiveCommand`-Modus, also dem normalen Betriebsmodus. In diesem Modus ist der Aktor wieder von außen erreichbar und verarbeitet aktiv Nachrichten aus seinem Postfach.

An der folgenden exemplarischen Implementierung wird verdeutlicht, wie man mit einem Akka Persistence Aktor arbeitet. Der `PersistentExampleActor` in dieser Implementierung besitzt einen Zustand in Form eines ganzzahligen Werts (`Int`). In dem vorliegenden Beispiel wird der `ExampleActor` aus dem Abschnitt 2.3.1 zu einem Akka Persistence Aktor.

Zuerst werden wieder alle Nachrichten deklariert:

```
object PersistentExampleActor {

  sealed trait Evt

  case object Increment extends Evt
  case object Decrement extends Evt

  case object WhatIsYourResult
  case class  MyResultIs(value: Int)

}
```

- Nachrichten vom Typ `Increment` können dazu benutzt werden, um den internen Zustand des Aktors um den Wert 1 zu erhöhen. Diese Nachrichten erben von `Evt`, da diese Kommandos später als Events verwendet werden.
- Nachrichten vom Typ `Decrement` können dazu benutzt werden, um den internen Zustand des Aktors um den Wert 1 zu verringern. Diese Nachrichten erben von `Evt`, da diese Kommandos später als Events verwendet werden.
- Nachrichten vom Typ `WhatIsYourResult` können dazu benutzt werden, um über den internen Zustand des Aktors Auskunft zu erhalten. Diese Nachrichten sind Abfragen. Der Aktor antwortet auf diese Nachrichten mit einer Nachricht vom Typ `MyResultIs`.

Der `PersistentExampleActor` wird anschließend wie folgt implementiert:

```
import akka.actor._
import akka.persistence._

class PersistentExampleActor extends PersistentActor {

  import PersistentExampleActor._

  var state: Int = 0

  override def persistenceId: String = "persistentExampleActorId"
```

- Die Variable `state` beschreibt den Zustand des Aktors in Form eines ganzzahligen Werts (`Int`). Der Wert beträgt bei der Initialisierung des Aktors `0`.
- Die Variable `persistenceId` definiert die ID des Aktors. Diese ID wird später für die Zuordnung im Journal und im Snapshot Storage genutzt (mehr dazu im Abschnitt 3).

```
override def receiveCommand: Receive = {

  case Increment =>

    persist(Increment) { evt => updateState(evt) }

  case Decrement =>

    persist(Decrement) { evt => updateState(evt) }

  case WhatIsYourResult =>

    sender ! MyResultIs(value = state)

}
```

- Über die Funktion `receiveCommand` erhält der Aktor die eingehenden Nachrichten (Kommandos oder Abfragen) im `receiveCommand`-Modus, also dem normalen Betriebsmodus. Mit Hilfe einer Fallunterscheidung wird anschließend unterschieden, um welche Nachrichten es sich handelt.
- Über den Aufruf von `persist(cmd) { evt => updateState(evt) }` wird aus dem Kommando (`cmd`) ein Event (`evt`). Dieses Event wird im Journal abgelegt und anschließend von der Funktion `updateState(evt: Evt)` verarbeitet.

```
private def updateState(evt: Evt): Unit = evt match {

  case Increment =>

    state = state + 1
    snapshot()

  case Decrement =>

    state = state - 1
    snapshot()

}
```

Über die Funktion `updateState(evt: Evt)` wird eine Zustandsänderung nach einer Fallunterscheidung durchgeführt und anschließend die Funktion `snapshot` angesprochen. Diese Funktion wird sowohl im normalen Betriebsmodus als auch im Wiederherstellungsmodus genutzt.

```
  private def snapshot(): Unit = {

    if (
      !recoveryRunning &&
      lastSequenceNr % 5 == 0 &&
      lastSequenceNr != 0
    )  saveSnapshot(state)

  }
```

Über die Funktion `snapshot` wird vom Zustand des Aktors ein Snapshot angefertigt und über den Aufruf von `saveSnapshot(state)` im Snapshot Storage abgelegt. Die Funktion `snapshot` überprüft vor dieser Prozedur, ob der Aktor nicht im Wiederherstellungsmodus läuft, ob es sich nicht um die erste zugestellte Nachricht handelt und, dass nur jede fünfte Nachricht zu einem Snapshot führt.

```
  override def receiveRecover: Receive = {

    case SnapshotOffer(_, snapshot: Int) =>

      state = snapshot

    case evt: Evt =>

      updateState(evt)

  }
}
```

Über die Funktion `receiveRecover` erhält der Aktor die eingehenden Nachrichten (Snapshots oder Events) im `receiveRecover`-Modus, also dem Wiederherstellungsmodus. Mit Hilfe einer Fallunterscheidung wird anschließend unterschieden, um welche Nachrichten (Snapshots oder Events) es sich handelt:

- Beim Empfangen eines Snapshots (`case SnapshotOffer(_, snapshot: Int)`) wird der Zustand des Aktors auf den Snapshot gesetzt.
- Beim Empfangen von Events (`case evt: Evt`) werden diese wieder durch `updateState` verarbeitet.

## 2.4 Aufbau und Ablauf der Experimente

Um die Schnelligkeit und Praxistauglichkeit (siehe Definitionen im Abschnitt 1.4) der SerDes zu eruieren, wurden zwei Experimente konzipiert:

- **Experiment E1** ***Vollständige Umgebung***: Dieses Experiment wurde entwickelt, um die ausgewählten SerDes in einem vollständigen Umfeld zu testen. Ein vollständiges Umfeld stellt einen minimalen Akka Persistence Aufbau mit einem Test-Aktorensystem und mit einem Test-Aktor dar. Das Experiment und das vollständige Umfeld werden im Abschnitt 2.4.1 beschrieben.
- **Experiment E2** ***Benchmark Umgebung***: Dieses Experiment testet isoliert die ausgewählten SerDes bezüglich Geschwindigkeit über ScalaMeter. Das Experiment wird im Abschnitt 2.4.2 beschrieben.

#### Beschreibung der verwendeten externen Komponenten

In den Experimenten wurden verschiedene externe Komponenten verwendet, die nun zum besseren Verständnis erläutert werden:

- **Java JDK** ist eine Sammlung von Programmierwerkzeugen und Programmbibliotheken, um Anwendungen mit der Programmiersprache Java entwickeln zu können [Jav].
- **Scala** ist eine funktionale und objektorientierte Programmiersprache für die JVM [SCAb].
- **SBT** ist ein Build-Werkzeug [SCAf].
- **Akka Actors** ist eine Implementierung des Aktorenmodells für die JVM (siehe Abschnitt 2.3.1) [Akkd].
- **Akka Persistence** ist eine Implementierung des ES-Ansatzes für Akka Actors (siehe Abschnitt 2.3.2) [Akkd].
- **Circe** ist eine JavaScript Object Notation (JSON)-Implementierung für Scala (siehe Abschnitt 3.2) [GITa].
- **LevelDB JNI** ist ein Java Native Interface (JNI)[2] für die Datenbank LevelDB[3] [GITb].
- **Port of LevelDB to Java** ist eine Portierung der Datenbank LevelDB zu Java [GITc].
- **ScalaMeter** ist ein Microbenchmarking- und Regressionstestframework für die JVM und die Programmiersprache Scala (siehe Abschnitt 2.4.2) [GITd].
- **ScalaPB** ist ein Protocol Buffers (Protobuf)-Compiler für Scala (siehe Abschnitt 3.3) [GITe].

Die Versionen der verwendeten Komponenten werden im Abschnitt A.2 dokumentiert.

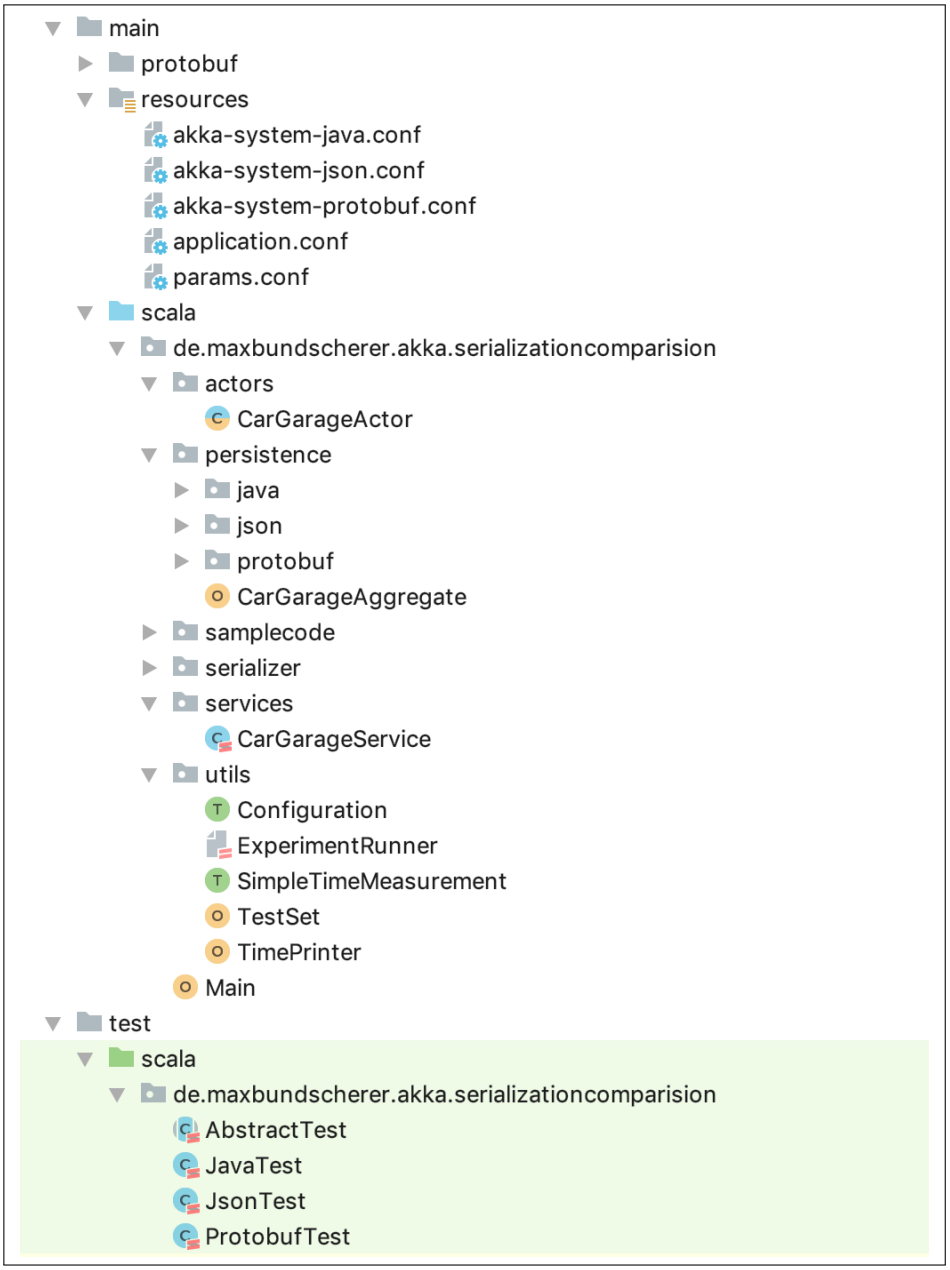


**Abbildung 2.4** Paketübersicht Quellcode der Experimente

### Aufbau des Quellcodes

Die Experimente wurden in Scala implementiert. Der vollständige Quellcode und die dazugehörige Dokumentation, ist dem Github-Projekt `maxbundscherer/akka-serialization-comparision`[4] zu entnehmen.

Da eine Darstellung des gesamten Quellcodes in Form eines Klassendiagramms über beide Experimente zu unübersichtlich wird, wird im Laufe der Arbeit das Experiment E1 *Vollständige Umgebung* als Sequenzdiagramm (Abschnitt 2.4.1) und das Experiment E2 *Benchmark Umgebung* als Klassendiagramm (Abschnitt 2.4.2) visualisiert.

Der Abbildung 2.4 ist der Aufbau des Quellcodes zu entnehmen. Um diesen genauer zu verstehen, werden nun die einzelnen Pakete beschrieben:

- `main/protobuf/`: Aus `.proto`-Dateien in diesem Paket generiert der Protobuf-Compiler ScalaPB von den Protobuf-Beschreibungen den Scala-Quellcode (mehr dazu im Abschnitt 3.3). Dieser Compiler wird als SBT Plugin eingebunden und kompiliert vor dem eigentlichen kompilieren der Scala-Klassen.
- `main/resources/`: In diesem Paket sind die Konfigurationen (`application.conf` und `akka-system-*.conf`) und die Test-Parameter (`params.conf`) hinterlegt. Die verschiedenen SerDes werden über die jeweiligen Konfigurationen (zum Beispiel `akka-system-java.conf`) eingebunden. Diese Konfigurationen überschreiben die Grundkonfiguration (`application.conf`). Um einen realistischen Ablauf zu ermöglichen, wurden nur für die Experimente notwendigen Parameter manuell konfiguriert.
- `main/scala/[...]/`: In diesem Paket befindet sich der Programmeinstiegspunkt für Experiment E1 *Vollständige Umgebung*. Dieses wird im Abschnitt 2.4.1 genauer spezifiziert.
- `main/scala/[...]/actors/`: In diesem Paket befindet sich der Test-Aktor für das Experiment E1 *Vollständige Umgebung*.
- `main/scala/[...]/persistence/`: In diesem Paket werden die Nachrichten, Kommandos und Events deklariert. Generierte Scala-Klassen von ScalaPB werden hier abgelegt. Klassen aus diesem Paket sind für die verschiedenen SerDes von Relevanz.
- `main/scala/[...]/samplecode/`: In diesem Paket befindet sich Beispiels-Quellcode für diese Arbeit. Dieses Paket ist für die Experimente nicht von Relevanz.
- `main/scala/[...]/serializer/`: In diesem Paket werden die unterschiedlichen SerDes eingebunden und angesprochen.
- `main/scala/[...]/services/`: In diesem Paket befindet sich eine Abstraktionssicht, um typsicher mit dem Test-Aktor interagieren zu können. Typsicherheit reduziert unter anderem unerwünschtes oder fehlerhaftes Programmverhalten [UNI].

2 `https://docs.oracle.com/javase/8/docs/technotes/guides/jni/`
3 `https://github.com/google/leveldb`
4 `https://github.com/maxbundscherer/akka-serialization-comparision`

- `main/scala/[...]/utils/`: In diesem Paket befinden sich verschiedene hilfreiche Implementierungen (zum Beispiel die Zeitmessung und Testdatengenerierung).
- `test/scala/[...]/`: In diesem Paket befindet sich der Programmeinstiegspunkt für Experiment E2 *Benchmark Umgebung*. Dieses wird im Abschnitt 2.4.2 genauer spezifiziert.

**Hinweis**: In der Aufführung ist `[...]` durch `de/maxbundscherer/akka/serializationcomparision` zu ersetzen.

#### Besonderheiten in Scala

Die Experimente (Experiment E1 *Vollständige Umgebung* und Experiment E2 *Benchmark Umgebung*) wurden in Scala implementiert. Der Java-SerDes und andere verwendete Komponenten sind in Java implementiert. Es ist möglich, diese Java-Komponenten innerhalb von Scala zu benutzen, da Scala-Bytecode mit Java-Bytecode kompatibel ist. Als Bytecode wird eine Sammlung von Befehlen für eine virtuelle Maschine bezeichnet. Programmiersprachen wie Java und Scala werden nicht zu einem direkten Maschinencode kompiliert, sondern zu einem Zwischencode (genannt Bytecode) [TEC]. Java- und Scala-Bytecodes sind innerhalb der JVM lauffähig und miteinander kompatibel. Dadurch können Scala-Komponenten von Java-Komponenten benutzt werden und andersrum [SCAa]:

```
object Example {

  val javaUUID : java.util.UUID  = java.util.UUID.randomUUID()
  val scalaString : String     = javaUUID.toString

}
```

In der Programmiersprache Scala sind `Traits` vertreten. Diese sind ähnlich zu einem `Interface` in Java 8 [SCAg]. `Traits` selbst können nicht instanziiert werden, besitzen keinen Konstruktor, können aber Implementierungen und Daten enthalten. Eine Scala-Klasse kann von mehreren `Traits` erweitert werden, aber nicht von mehreren abstrakten Scala-Klassen. `Traits` können dazu benutzt werden, um Daten zwischen Scala-Klassen auszutauschen. Diesen Mechanismus verwendet der `trait Configuration` im `utils`-Paket, um Konfigurationen Entwicklern zugänglich zu machen:

```
trait Configuration {
  object Config {
    val testConfig = "testValue"
  }
}

object Example extends Configuration {
  val myConfig = Config.testConfig
}
```

In Scala ist es mit dem Schlüsselwort `implicit` möglich, Klassen ohne direkt sichtbaren Quellcode zu erweitern [ALV]. Dieser Mechanismus ähnelt den Erweiterungsmethoden[5] aus der Programmiersprache C#. In der Scala-Klasse `JsonSerializer` wird dieser Mechanismus verwendet, um die Klasse `AddCarEvtDb` um die Methode `.asJson` zu erweitern. Dies ist möglich, da die Klasse `JsonSerializer` das Paket `io.circe.syntax._` im Quellcode importiert und die Klasse `AddCarEvtDb` von der Klasse `AnyVal` erbt:

```
package io.circe

/**
* This package provides syntax via enrichment classes.
*/
package object syntax {
  implicit final class EncoderOps[A](val wrappedEncodeable: A) extends
     AnyVal {
    final def asJson(implicit encoder: Encoder[A]): Json = ???
  }
  implicit final class StringOps(val value: String) extends AnyVal {
    final def :=[A: Encoder](a: A): (String, Json) = ???
  }
}
```

Durch das Importieren von diesem `Package object` ist es möglich, bei Objekten der Klasse `AddCarEvtDb` die Methode `.asJson` benutzen zu können:

```
val value: AddCarEvtDb = ???
value.asJson
```

**Hinweis:** Einige Punkte wurden in diesem Beispiel nicht implementiert, wie man an `???` erkennen kann, da diese nicht zum aktuellen Verständnis beitragen würden.

#### Gemeinsame Testdaten für die Experimente

Die Testdaten werden vor den Durchläufen automatisch generiert. Um einen sinnvollen Vergleich sicherzustellen, werden die Testdaten beim Start eines Experiments generiert und alle Läufe laufen mit den gleichen Testdaten ab (mehr dazu im Abschnitt 4.1).

Es wird zwischen zwei Arten von Testdatenklassen unterschieden:

```
case class Car(
  id: Int,
  horsepower: Int,
  name: String
)
```

und

5 `https://docs.microsoft.com/de-de/dotnet/csharp/programming-guide/classes-and-structs/extension-methods`

```
case class ComplexCar(
  id: Int,
  horsepower: Int,
  name: String,
  fuelConsumption: Float,
  dieselEngine: Boolean,
  seatAdjustment: Boolean,
  fuelTank: Double,
  brakingDistance: Double,
  notes: String
)
```

Testobjekte aus der Klasse `Car` eigenen sich, um die unterschiedlichen SerDes bei einer geringen Last zu testen. Testobjekte aus der Klasse `ComplexCar` eigenen sich, um die unterschiedlichen SerDes bei einer höheren Last zu testen.

Die Generierung der Testdaten lässt sich mit den folgenden Parametern beeinflussen:

- `numberOfTestCars`: Dieser Wert bestimmt die Anzahl der Testdaten (zum Beispiel `10000`)
- `carNameStringMaxLength`: Dieser Wert bestimmt die maximale Zeichenanzahl des Attributs `name` bei Objekten aus der Testklasse `Car` und `ComplexCar` (zum Beispiel `200`).
- `complexCarNotesStringMaxLength`: Dieser Wert bestimmt die maximale Zeichenanzahl des Attributs `notes` bei Objekten aus der Testklasse `ComplexCar` (zum Beispiel `900`).

Die Parameter befinden sich in der Konfigurationsdatei[6] unter dem Abschnitt `testSet`.

#### Wichtige Hinweise zu der Durchführung der Experimente

Um die Experimente selbst durchführen zu können, wird SBT benötigt. SBT sollte vor den Experimenten anders als Standard parametrisiert werden, um mehr Ressourcen (zum Beispiel Arbeitsspeicher) auf dem Zielsystem nutzen zu können.

Dies ist über eine Umgebungsvariable mit dem Befehl `export SBT_OPTS="-Xms1G -Xmx8G"` möglich:

- Der Parameter `Xms` gibt die initial allokierte Heap-Größe an.
- Der Parameter `Xmx` gibt die maximal allokierbare Heap-Größe an.

Mit folgenden Befehlen können die Experimente gestartet werden:

- Experiment E1 *Vollständige Umgebung*: `sbt clean run`
- Experiment E2 *Benchmark Umgebung*: `sbt clean test`

6 https://github.com/maxbundscherer/akka-serialization-comparision/blob/master/src/main/resources/params.conf

- Starten beider Experimente (sequentiell): `sbt mixedMode`
- Starten beider Experimente (sequentiell) und die Ausgabe protokollieren: `./autoRunner.sh`

#### Messschwankungen

Messschwankungen können zum Beispiel durch nicht vorhersehbare Durchlaufzeiten von Garbage Collection (GC), verschiedene automatische Optimierungsmaßnahmen (z.B.: adaptive Zwischenspeicherung von Befehlen u. Daten und eine sich selbst optimierende Sprungvorhersage) und unterschiedliche Festplattenzugriffszeiten entstehen.

Um diese Schwankungen zu reduzieren, werden die Läufe mehrmals auf unterschiedlichen Referenzsystemen (Referenzsystem R1 *Windows 10*, Referenzsystem R2 *iMac* und Referenzsystem R2 *iMac*) durchgeführt.

### 2.4.1 Experiment E1 *Vollständige Umgebung*

Dieses Experiment läuft über sogenannte `ExperimentRunner` ab. Jeder SerDes wird über einen eigenen `ExperimentRunner` getestet. Die `ExperimentRunner` laufen nicht parallel, sondern sequentiell, und unterscheiden sich nur über die Verwendung von unterschiedlichen SerDes. Um einen aussagekräftigen Vergleich sicherzustellen, werden auf den Umgebungen immer dieselben Operationen mit den gleichen Testdaten und den gleichen Testparametern durchgeführt. Die benötigte Durchlaufzeit wird während des Experiments gemessen und anschließend ausgegeben. Bei der Durchführung dieses Experiments entstehen starke Messschwankungen (siehe Abschnitt 2.4). Die genaue `ExperimentRunner`-Implementierung lässt sich dem Github-Projekt `maxbundscherer/akka-serialization-comparision`[7] entnehmen.

Die Abbildung 2.5 stellt den Ablauf eines `ExperimentRunners` als abstraktes nicht-vollständiges Sequenzdiagramm dar:

- **Teildurchlauf: Abfragen aller Autos** wird dazu benutzt, um über den internen Zustand des Test-Aktors Auskunft zu erhalten. Um diesen Zustand übermitteln zu können, muss der Aktor erst gestartet und vollständig wiederhergestellt werden. **Daher wird dieser Mechanismus benutzt, um einen Start und eine Wiederherstellung des Aktors zu erzwingen.**
- **Teildurchlauf: Anlegen aller Autos** wird dazu benutzt, um den internen Zustand des Test-Aktors zu modifizieren. Dieser Teildurchlauf legt eine über Parameter definierte Anzahl von Objekten aus der Klasse `Car` und `ComplexCar` an. Die Erstellung erfolgt über die Events `AddCarEvt` und `AddComplexCarEvt`. Diese Events werden vor der eigentlichen Zustandsänderung serialisiert. **Daher wird dieser Teildurchlauf genutzt, um die Serialisierung zu testen.**

7 `https://github.com/maxbundscherer/akka-serialization-comparision/blob/master/src/main/scala/de/maxbundscherer/akka/serializationcomparision/utils/ExperimentRunner.scala`

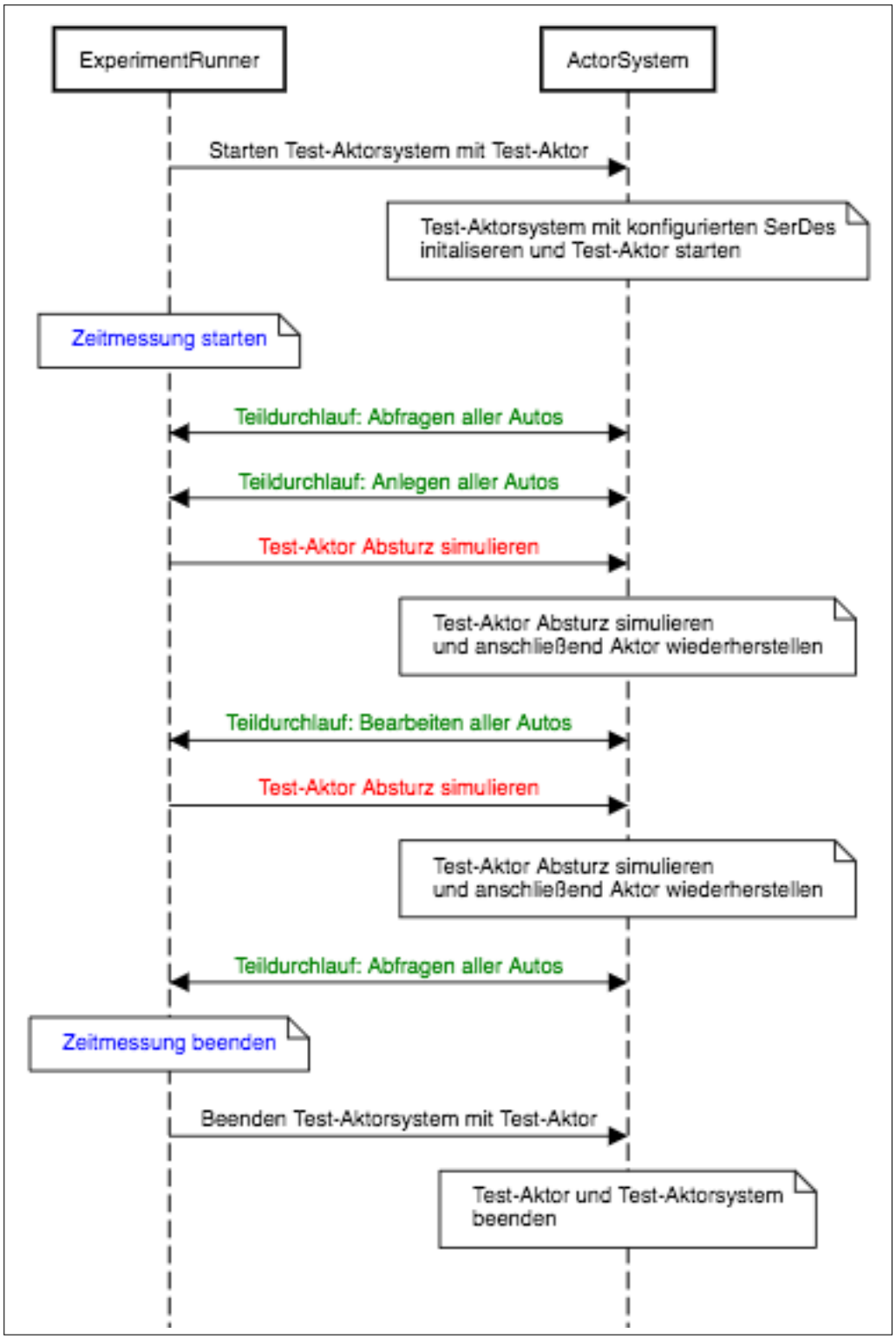


**Abbildung 2.5** Sequenzdiagramm von Experiment E1 *Vollständige Umgebung*

- **Test-Aktor Absturz simulieren** wird dazu benutzt, um den Test-Aktor abstürzen zu lassen. Nach Eingang des Kommandos `SimulateCrashCmd` wird auf dem Aktor eine Ausnahme vom Typ `RuntimeException` ausgelöst. Über diese `RuntimeException` stürzt der Aktor ab, wird anschließend neu gestartet und stellt sich wieder her. Bei der Wiederherstellung müssen alle verarbeitenden Events und Snapshots wieder deserialisiert werden. **Daher wird dieser Mechanismus genutzt, um die Deserialisierung zu testen.**

- **Teildurchlauf: Bearbeiten aller Autos** wird dazu benutzt, um den internen Zustand des Test-Aktors zu modifizieren. Dieser Teildurchlauf bearbeitet eine über Parameter definierte Anzahl von Objekten aus der Klasse `Car` und `ComplexCar`. Die Bearbeitung erfolgt über die Events `UpdateCarEvt` und `UpdateComplexCarEvt`. Diese Events werden vor der eigentlichen Zustandsänderung serialisiert. **Daher wird dieser Teildurchlauf genutzt, um die Serialisierung zu testen.**

#### Parameter dieses Experiments

Dieses Experiment lässt sich mit folgenden Parametern konfigurieren:

- `timeoutInSeconds`: Dieser Wert bestimmt die maximale Zeit in Sekunden, die gewartet wird, bis der Test-Aktor antwortet (zum Beispiel `6000`). Der Wert sollte nicht zu gering gewählt werden, da die Wiederherstellung des Aktors je nach Parameter mehr Zeit in Anspruch nimmt.

- `actorSnapshotInterval`: Dieser Wert bestimmt, in welchem Nachrichten-Intervall der Test-Aktor Snapshots von seinem Zustand macht (mehr dazu im Abschnitt 2.3.2) (zum Beispiel `10000`).

- `numberOfAdds`: Dieser Wert bestimmt die Anzahl von Objekten aus der Klasse `Car` und `ComplexCar`, die beim *Teildurchlauf: Anlegen aller Autos* angelegt werden (zum Beispiel `1000`).

- `numberOfUpdates`: Dieser Wert bestimmt die Anzahl von Objekten aus der Klasse `Car` und `ComplexCar`, die beim *Teildurchlauf: Bearbeiten aller Autos* bearbeitet werden (zum Beispiel `100000`).

- `testCar`: Dieser Wert bestimmt, ob mit Objekten aus der Klasse `Car` beim *Teildurchlauf: Anlegen aller Autos* und *Teildurchlauf: Bearbeiten aller Autos* getestet wird (zum Beispiel `true`).

- `testComplexCar`: Dieser Wert bestimmt, ob mit Objekten aus der Klasse `ComplexCar` beim *Teildurchlauf: Anlegen aller Autos* und *Teildurchlauf: Bearbeiten aller Autos* getestet wird (zum Beispiel `false`).

- `waitForProfilerEnter`: Dieser Wert definiert, ob bei Programmstart auf eine Benutzereingabe (zum Beispiel Entertaste wird gedrückt) gewartet wird (zum Beispiel `false`). Das Warten auf eine Benutzereingabe kann sinnvoll sein, wenn zum Beispiel ein Profiler eingebunden wird.

Die Parameter befinden sich in der Konfigurationsdatei[8] unter dem Abschnitt `experimentMode`.

### 2.4.2 Experiment E2 *Benchmark Umgebung*

Dieses Experiment führt über ScalaMeter eine Geschwindigkeitsmessung durch und testet isoliert die SerDes mit Objekten der Klasse `AddCarEvt` und `AddComplexCarEvt`. Die gemessene Zeit wird anschließend in Millisekunden ausgegeben.

ScalaMeter unterstützt mehrere Arten von Tests. In diesem Experiment wird ein Test mit einfacher und lokaler Zeitmessung durchgeführt. Das Framework bietet die Möglichkeit eigenständig Testdaten zu generieren [Scad]. Da diese Möglichkeit aber den Testfall nicht sinnvoll abbilden kann, wird in diesem Experiment auf eigene generierte Testdaten zurückgegriffen (mehr dazu im Abschnitt 2.4).

Um über ScalaMeter eine Zeitmessung durchführen zu können, muss zunächst ein Generator für die Testdaten angeben werden [Scae] [Scad]. In diesem Fall wurde ein Generator für ganzzahlige Werte (`Int`) gewählt. Dieser wird mit folgenden Eigenschaften parametrisiert:

- `axisName`: Dieser Wert definiert den Namen des Generators (zum Beispiel `MyGenerator`).
- `from`: Dieser Wert definiert den Startwert des generierten globalen Bereichs (zum Beispiel `100`).
- `upto`: Dieser Wert definiert den Endwert des generierten globalen Bereichs (zum Beispiel `1000`).
- `hop`: Dieser Wert definiert die Sprünge zwischen den einzelnen Bereichen (zum Beispiel `100`).

Wenn die Parameter wie folgt belegt werden:

- `from`: 100
- `upto`: 300
- `hop`: 100

werden drei einzelne Bereiche (`0-100`; `0-200` und `0-300`) generiert.

ScalaMeter führt für jeden Bereich einen einzelnen Durchlauf durch (im vorherigen Beispiel also drei Durchläufe) und übergibt die Werte der einzelnen Bereiche dem einzelnen Durchlauf. Diese Eigenschaft ist für den Ablauf des Experiments nicht zielführend, da nur eine bestimmte Anzahl von Testobjekten getestet werden soll. Daher wurden bei der Durchführung der Experimente die Parameter `from`, `upto` und `hop` auf den gleichen Wert gesetzt, um nur einen Durchlauf mit nur einem Bereich zu erzwingen.

8 `https://github.com/maxbundscherer/akka-serialization-comparision/blob/master/src/main/resources/params.conf`

**Beispiel**: `from = upto = hop = 300` führt dazu, dass mit 300 Objekten aus den vorher generierten Testdaten (siehe Abschnitt 2.4) getestet und die Zeit gemessen wird. Um das zu ermöglichen, wird für jeden ganzzahligen Wert aus diesem generierten Bereich ein Testobjekt aus den Testdaten genommen. Der ganzzahlige Wert dient dabei als Index für den Zugriff auf die generierten Testdaten.

Für jedes Testobjekt (Objekte der Klasse `AddCarEvt` oder `AddComplexCarEvt`) wird eine Serialisierung zu einer Byte-Folge durchgeführt. Anschließend eine Deserialisierung von der Byte-Folge zurück zum Testobjekt.

Jeder Durchlauf wird von ScalaMeter standardmäßig 36-mal mit einer Zeitmessung durchgeführt. Anschließend wird das Ergebnis gemittelt und ausgegeben. Dies dient dazu, um ein genaueres Messergebnis zu erhalten. Die ausgebenden Werte liegen in Millisekunden vor [Scae].

Das Framework führt vor jedem Durchlauf `warmup runs` durch, bis ein stabiler Messzustand (mit dem Namen `Steady-state`) erreicht wird [Scac]. Erst wenn dieser Zustand erreicht wird, werden die eigentlichen Test-Durchläufe durchgeführt. Dies dient auch dazu, die Messschwankungen (siehe Abschnitt 2.4) zu reduzieren.

#### Implementierung dieses Experiments

Das nicht-vollständige Klassendiagramm (Abbildung 2.6) zeigt den relevanten Teil der Implementierung:

- Jeder SerDes wird durch einen einzelnen Test (`JavaTest`, `JsonTest` und `ProtobufTest`) getestet.
- Jeder dieser einzelnen Tests erbt von der abstrakten Klasse `AbstractTest`, in der die eigentlichen Tests implementiert sind.
- Die einzelnen Tests unterscheiden sich nur in der Verwendung unterschiedlicher SerDes.
- Die Funktion `triggerSingleSerializeAndDeserialize(i: Int)` wird für jeden ganzzahligen Wert aus dem von ScalaMeter generierten Testbereich (zum Beispiel `0-100`) aufgerufen. Diese Funktion holt sich zunächst über den Wert des Parameters `i` als Index ein Objekt aus den Testdatenobjekten. Dieses Objekt wird anschließend (über die Funktion `serializeAddCarEvt(value: AddCarEvt)` bzw. `serializeAddComplexCarEvt(value: AddComplexCarEvt)`) serialisiert und abschließend (über die Funktion `deserializeAddCarEvt(value: AddCarEvt)` bzw. `deserializeAddComplexCarEvt(value: AddComplexCarEvt)`) deserialisiert.

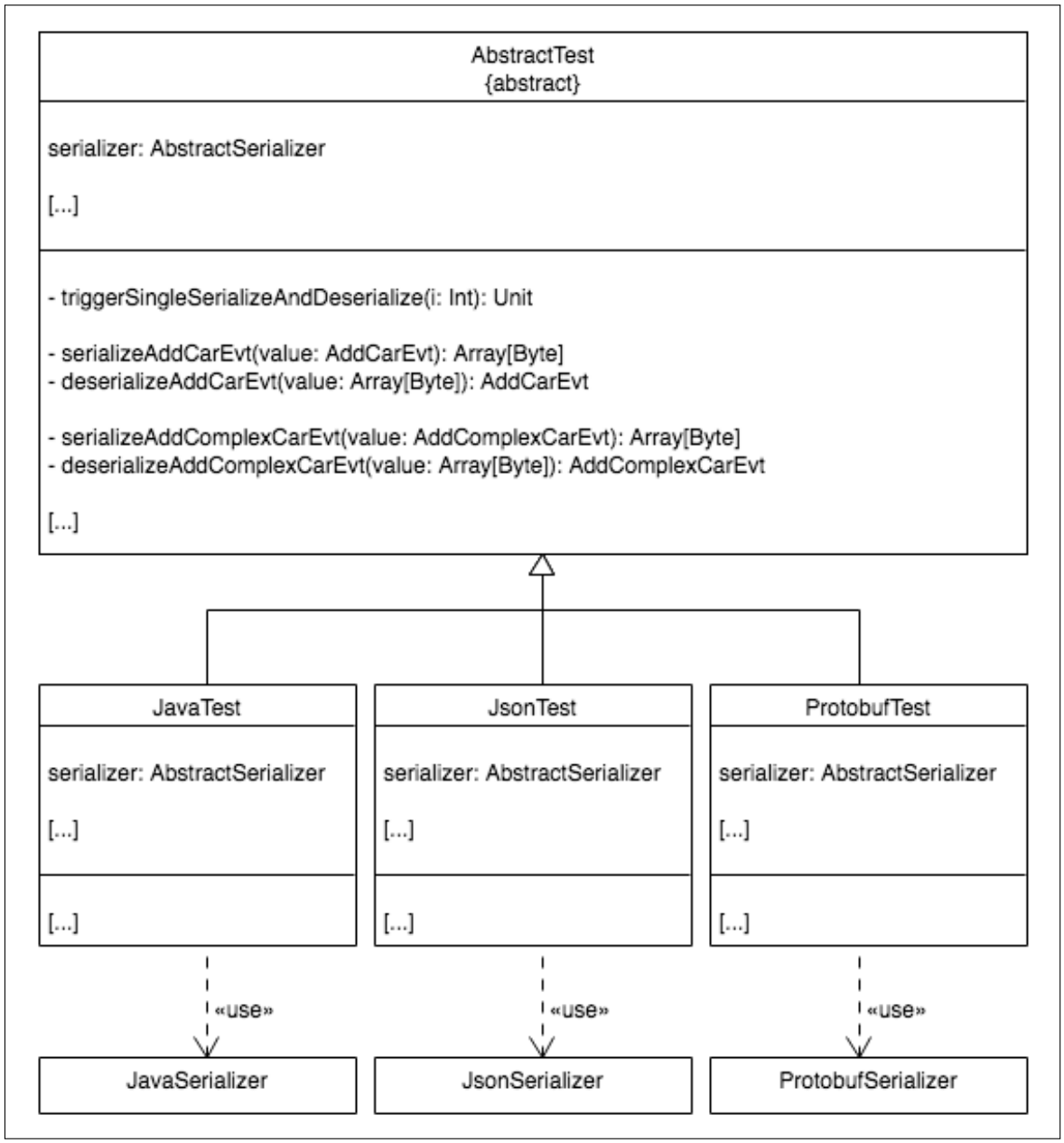


**Abbildung 2.6** Klassendiagramm von Experiment E2 *Benchmark Umgebung*

#### Parameter dieses Experiments

Dieses Experiment lässt sich mit folgenden Parametern konfigurieren:

- `numberOfSingleTests`: Dieser Wert bestimmt die Anzahl der getesteten Objekte aus den Testdaten (zum Beispiel `100`). Der Wert sollte höher als die Anzahl der Testdaten sein.
- `testCar`: Dieser Wert bestimmt, ob mit Objekten aus der Klasse `AddCarEvt` getestet wird (zum Beispiel `true`).
- `testComplexCar`: Dieser Wert bestimmt, ob mit Objekten aus der Klasse `AddComplexCarEvt` getestet wird (zum Beispiel `false`).

Die Parameter befinden sich in der Konfigurationsdatei[9] unter dem Abschnitt `benchmarkMode`.

9 `https://github.com/maxbundscherer/akka-serialization-comparision/blob/master/src/main/resources/params.conf`

# 3 Serialisierer/De-Serialisierer im Akka Persistence Umfeld

Dieses Kapitel setzt voraus, dass sich der Leser mit Akka Actors (Abschnitt 2.3.1) und Akka Persistence (Abschnitt 2.3.2) auseinandergesetzt hat:

- **Events** werden durch Akka Persistence im **Journal** verwaltet.
- **Snapshots** werden durch Akka Persistence im **Snapshot Storage** verwaltet.

Events und Snapshots sind zur Laufzeit Objekte. Um diese Objekte im Journal oder Snapshot Storage abbilden zu können, müssen diese erst über einen SerDes zu einer Byte-Folge serialisiert werden. Um ein Event oder einen Snapshot wieder als Objekt zur Laufzeit benutzen zu können, muss dieses erst von einer Byte-Folge über einen SerDes deserialisiert werden.

#### Verwaltung von Events im Journal

Das Journal wird in Akka Persistence in Form eines Plugins eingebunden [Akka]. Mehr dazu im Abschnitt 3. Um ein Event im Journal verwalten zu können, werden mindestens folgende Informationen benötigt:

- `persistenceId`: Dieser Wert wird für die Zuordnung der Events für einen Aktor benutzt und sollte daher nicht doppelt vorkommen (zum Beispiel `customer-account-123`). **Der Wert muss im Aktor vom Entwickler gesetzt werden (siehe Abschnitt 2.3.2).**
- `sequenceNumber`: Dieser ganzzahlige fortlaufende Wert definiert die Reihenfolge der Events (zum Beispiel `0`). Durch diese Information wird sichergestellt, dass der Aktor bei der Wiederherstellung die Events chronologisch korrekt abarbeitet (siehe Abschnitt 2.3.2). **Dieser Wert wird von Akka Persistence automatisch gesetzt.**
- `manifest`: Dieser Wert repräsentiert die Klassenzuordnung der Events als Zeichenkette (zum Beispiel `AddCarEvt`). Über diese Information kann der SerDes unterscheiden, um welche Klasse es sich bei dem Event beim Deserialisieren handelt. **Dieser Wert wird vom SerDes beim Serialisieren gesetzt** (mehr dazu im Abschnitt 3).
- `payload`: Dieser Wert repräsentiert das serialisierte Event als Byte-Folge. **Der Wert wird vom SerDes beim Serialisieren gesetzt.**

#### Verwaltung von Snapshots im Snapshot Storage

Der Snapshot Storage wird in Akka Persistence als Plugin eingebunden [Akka]. Mehr dazu im Abschnitt 3. Um einen Snapshot im Snapshot Storage verwalten zu können, werden mindestens folgende Informationen benötigt:

- `persistenceId`: Dieser Wert wird für die Zuordnung der Snapshots für einen Aktor benutzt und sollte daher nicht doppelt vorkommen (zum Beispiel `customer-account-123`). **Der Wert muss im Aktor vom Entwickler gesetzt werden (siehe Abschnitt 2.3.2).**
- `payload`: Dieser Wert repräsentiert den serialisierten Snapshot als Byte-Folge. **Der Wert wird vom SerDes beim Serialisieren gesetzt.**

**Hinweis:** Eine Information über die Klassenzuordnung (über zum Beispiel die Information `manifest` im Journal) ist an dieser Stelle nicht notwendig, da Akka Persistence nur eine Scala-Klasse als Snapshot-Klasse akzeptiert.

#### Unterschiedliche Arten von Serialisierer/De-Serialisierer

Es ist möglich zwei Arten von SerDes in Akka einzubinden [Akkh]:

- `Serializer`: Diese Art von SerDes setzt die Information `manifest` im Journal automatisch. Das Attribut wird hierbei auf den Klassennamen des Events gesetzt.
- `SerializerWithStringManifest`: Diese Art von SerDes setzt die Information `manifest` im Journal nicht automatisch und muss daher vom Entwickler definiert werden. Diese Art von SerDes wird im Laufe der Arbeit behandelt.

#### Definieren eigener Serialisierer/De-Serialisierer

In Akka ist es möglich einen eigenen SerDes einzubinden. Dies ist über die Implementierung einer eigenen Klasse (hier `SampleSerDes`) möglich, die von der abstrakten Klasse `Serializer` oder `SerializerWithStringManifest` erbt [Akkc]:

```
package de.mb.akka.serializationcomparision.serializer

class SampleSerDes extends SerializerWithStringManifest {

  override def identifier: Int = ???

  override def manifest(o: AnyRef): String = ???

  override def toBinary(o: AnyRef): Array[Byte] = ???

  override def fromBinary(bytes: Array[Byte], manifest: String): AnyRef
     = ???

}
```

**Hinweis:** Einige Punkte wurden in diesem Beispiel nicht implementiert, wie man an `???` erkennen kann, da diese nicht zum aktuellen Verständnis beitragen würden.

- Der Wert `identifier` definiert die intern verwendete ID für Akka und sollte nicht doppelt vorkommen.
- Die Methode `manifest(o: AnyRef): String` wird von Akka Persistence benutzt, um die Information über die Klassenzuordnung beim Serialisieren auf die Information `manifest` abbilden zu können. Über diese Information kann der SerDes unterscheiden, um welche Klasse es sich beim Deserialisieren handelt. Daher sollte die gleiche Logik in der Funktion `toBinary(o: AnyRef): Array[Byte]` und `fromBinary(bytes: Array[Byte], manifest: String): AnyRef` implementiert werden.
- Die Methode `toBinary(o: AnyRef): Array[Byte]` wird von Akka Persistence benutzt, um ein Objekt (`o: AnyRef`) auf eine Byte-Folge (`Array[Byte]`) abbilden zu können. Diese Byte-Folge wird auf die Information `payload` abgebildet.
- Die Methode `fromBinary(bytes: Array[Byte], manifest: String): AnyRef)` wird von Akka Persistence benutzt, um eine Byte-Folge (`bytes: Array[Byte]`) über die Information über die Klassenzuordnung (`manifest: String`) durchführen zu können.
- Dieser Mechanismus wird auch genutzt, um die unterschiedlichen SerDes aus dieser Arbeit einbinden zu können (die Referenzimplementierung kann dem Abschnitt 2.4 entnommen werden).
- Nach Implementierung dieser Klasse muss dieser SerDes noch in der Konfiguration angegeben werden.

#### Konfiguration: Einbindung der Serialisierer/De-Serialisierer

Standardmäßig verwendet Akka die Java-Standardserialisierung [Akkh]. Diese wird aber nicht empfohlen, da diese zwischen unterschiedlichen Java-Versionen und Systemen nicht binärkompatibel ist [Akkb].

Wenn die Java-Standardserialisierung ohne Warnung beim Programmstart verwenden werden soll, kann in der Konfigurationsdatei (`src/main/resources/application.conf`) die Konfiguration `akka.actor.warn-about-java-serializer-usage` auf den Wert `false` gesetzt werden.

Die Einbindung eines SerDes erfolgt ebenfalls in der Konfigurationsdatei und sieht wie folgt aus:

```
akka.actor {

  serializers {
    mySerDes =
        "de.mb.akka.serializationcomparision.serializer.SampleSerDes"
  }

  serialization-bindings {
    "de.sample.Evt" = mySerDes
  }

}
```

- Über die Zuweisung von `mySerDes` wird ein SerDes eingebunden. Der Wert definiert den Klassenpfad der verwendeten Klasse, die von der abstrakten Klasse `Serializer` oder `SerializerWithStringManifest` erben muss.
- Über die Zuweisung von `"de.sample.Evt" = mySerDes` wird dieser SerDes für Objekte vom Typ `de.sample.Evt` verwendet.

#### Konfiguration: Journal- und Snapshot Storage-Plugin

Das Journal und der Snapshot Storage werden als Plugins in Akka eingebunden [Akka]. Es existieren verschiedene Plugins, die das Journal oder den Snapshot Storage auf verschiedene Arten von Datenbanken abbilden (zum Beispiel auf eine relationale Datenbank, auf eine NoSQL-Datenbank oder eine In-Memory-Datenbank). Auf die unterschiedlichen Plugins wird an dieser Stelle (wie bereits im Abschnitt 1.5 beschrieben) nicht eingegangen, da dies nicht zum aktuellen Verständnis beitragen würde.

Die Einbindung der Plugins erfolgt ebenfalls in der Konfigurationsdatei und sieht exemplarisch wie folgt aus:

```
akka.persistence {
  journal.plugin   = "akka.persistence.journal.leveldb"
  snapshot-store.plugin = "akka.persistence.snapshot-store.local"
}
```

- Für das Journal-Plugin wurde eine LevelDB-Portierung verwendet (mehr dazu im Abschnitt 2.4).
- Für das Snapshot Storage-Plugin wurde ein Plugin verwendet, dass die Snapshots im lokalen Dateisystem verwaltet.
- Für die Entwicklung ist diese Konfiguration ausreichend. Für eine Produktivumgebung sollte diese aber nicht verwendet werden, weil die Speicherung der Daten nur temporär erfolgt und die Daten beim Beenden des Programms bereinigt und damit gelöscht werden.

#### Nicht direkt kompatible Deserialisierung über einen Umweg

Manchen SerDes bereitet es Probleme, eine Byte-Folge nach Modifikation der Ursprungsklasse nach Serialisierung wieder direkt Deserialisieren zu können. Ein Beispiel hierfür findet man im Abschnitt 3.2. Um eine serialisierte Byte-Folge trotzdem noch verwenden zu können, kann sich mit der folgenden exemplarischen Implementierung beholfen werden:

**Hinweis**: Die Ursprungsklasse (z.B.: `CarV1`) wird dabei nicht verändert, stattdessen wird eine komplett neue Klasse (z.B.: `CarV2`) mit den gewünschten Eigenschaften und ein unidirektionaler Konverter (von der alten auf die neue Klasse) implementiert.

Das folgende Quellcode-Beispiel zeigt exemplarisch die Klasse `CarV1`:

```
case class CarV1(
  title: String,
  horsePower: Int,
  color: Int
)
```

Das folgende Quellcode-Beispiel zeigt exemplarisch die Klasse `CarV2`, bei der das Attribut `color` nun keine ganze Zahl mehr repräsentiert, sondern eine Zeichenkette:

```
case class CarV2(
  title: String,
  horsePower: Int,
  color: String
)
```

Das folgende Quellcode-Beispiel zeigt exemplarisch eine Funktion, die ein Objekt der Klasse `CarV1` zu einem Objekt der Klasse `CarV2` konvertiert:

```
def convertCar(value: CarV1): CarV2 = {
  CarV2 (
  title = value.title,
  horsePower = value.horsePower,
  color = value.color match { case 0 => "red" case _ => "blue" }
  )
}
```

Nun muss diese Logik noch in Akka Persistence eingebunden werden. Das folgende Quellcode-Beispiel zeigt exemplarisch eine Möglichkeit dafür:

```
class ConverterExample extends SerializerWithStringManifest {

  import ConverterExample._

  override def identifier: Int = 9001

  override def manifest(o: AnyRef): String = o match {
    case CarV1 => "CarV1"
    case CarV2 => "CarV2"
  }

  override def toBinary(o: AnyRef): Array[Byte] = o match {
    case obj: CarV1 => ???
    case obj: CarV2 => ???
  }

  override def fromBinary(bytes: Array[Byte], manifest: String): AnyRef
      = manifest match {

    case "CarV1" =>

      val obj: CarV1 = ???
      convertCar(obj)

    case "CarV2" =>

      ???
  }

}
```

**Hinweis:** Einige Punkte (Serialisierungs- und Deserialisierungs-Logik) wurden in diesem Beispiel nicht implementiert, wie man an `???` erkennen kann, da diese nicht zum aktuellen Verständnis beitragen würden.

Durch diese Implementierung wird ein Objekt der Klasse `CarV1` automatisch nach der Deserialisierung zu einem Objekt der Klasse `CarV2` konvertiert. Akka kann nun mit diesen Objekten arbeiten, ohne dass sich der Entwickler weiter damit beschäftigen muss.

## 3.1 Java Serialisierer/De-Serialisierer (Java-Standardserialisierung)

Standardmäßig verwendet Akka die Java-Standardserialisierung. Diese sollte aber nicht verwendet werden, wie bereits im Abschnitt 3 beschrieben. Die Java-Standardserialisierung kann auch innerhalb von Scala verwendet werden, wie bereits im Abschnitt 2.4 ersichtlich wird.

### Verwendung

Diese Arbeit geht nicht auf die Logik bzw. Implementierung dieses SerDes ein, wie bereits im Abschnitt 1.5 beschrieben. In Java lassen sich Objekte über verschiedene Ansätze serialisieren:

- **Standardserialisierung**: Die Objektstruktur und Zustände werden in ein binäres Format abgebildet. Dieses Verfahren wird auch als Java Object Serialization (JOS) bezeichnet [OPE].
- **XML-Serialisierung über JavaBeans Persistence**: Das Objekt wird auf ein XML-Format[1] abgebildet. Nur Java-Beans-Komponenten können mit diesem Verfahren serialisiert und deserialisiert werden [OPE].
- **XML-Abbildung über JAXB**: Die Objektstruktur und Zustände werden über JAXB[2] auf ein XML-Format abgebildet [OPE].

**Hinweis**: Diese Arbeit beschäftigt sich mit der Standardserialisierung.

Die **Serialisierung** erfolgt über die Klasse `ObjectOutputStream` und die Methode `writeObject`, wie in dem folgenden Beispiel demonstriert wird:

```
private def toJavaByteArray(o: java.io.Serializable): Array[Byte] = {

  val byteArrayOutputStream : ByteArrayOutputStream = new
     ByteArrayOutputStream

  val objectOutputStream : ObjectOutputStream = new
     ObjectOutputStream(byteArrayOutputStream)

  objectOutputStream.writeObject(o)

  objectOutputStream.close()
  byteArrayOutputStream.close()

  byteArrayOutputStream.toByteArray
}
```

1 https://wiki.selfhtml.org/wiki/XML
2 https://docs.oracle.com/javase/8/docs/technotes/guides/xml/jaxb/index.html

Die **Deserialisierung** erfolgt über die Klasse `ObjectInputStream` und die Methode `readObject`, wie in dem folgenden Beispiel demonstriert wird:

```
private def fromJavaByteArray[ObjectType](bytes: Array[Byte]) :
    ObjectType = {

  val byteArrayInputStream : ByteArrayInputStream = new
      ByteArrayInputStream(bytes)

  val objectInputStream : ObjectInputStream = new
      ObjectInputStream(byteArrayInputStream)

  val ans: ObjectType =
      objectInputStream.readObject().asInstanceOf[ObjectType]

  objectInputStream.close()
  byteArrayInputStream.close()

  ans
}
```

Dieser SerDes kann nur Klassen serialisieren bzw. deserialisieren, die von der Schnittstelle `Serializable` erben. Falls dies nicht der Fall ist, tritt eine Ausnahme vom Typ `NotSerializableException` auf. Diese Schnittstelle enthält keine Methoden und ist somit nur eine Markierungsschnittstelle[3] [OPE].

#### Ein Beispiel

Exemplarisch wird durch diese Form der Serialisierung aus dem Scala-Objekt

```
Car(id = 0, name = "BMW F30", horsePower = 200)
```

diese Byte-Folge:

```
¬í sr Jde.maxbundscherer.akka.serializationcomparision.samplecode.JavaExample$Card  ;2Ø l  I
horsePowerI idL namet Ljava/lang/String;xp    t BMW F30
```

**Hinweis**: Um diese Byte-Folge darzustellen, wurde diese in eine Zeichenkette konvertiert. Nicht druckbare Zeichen werden nicht abgebildet.

#### Hinweise

- Zum Deserialisieren benötige Klassen müssen unter dem gleichen Klassenpfad vorliegen wie bei der Serialisierung [JAX]. Dies kann durch manuelles Setzen des Attributs `serialVersionUUID` innerhalb der Klassen umgangen werden [Ull14].
- Falls das Attribut `serialVersionUUID` nicht vom Entwickler gesetzt wurde, wird dieses automatisch über das Java-Dienstprogramm `serialver`[4], auf Basis der Klassendefinition, berechnet [Ull14]. Eine Änderung der Klassendefinition kann also zu einer veränderten `serialVersionUUID` führen.

3 https://www.it-visions.de/glossar/alle/3113/Marker_Interface.aspx
4 https://docs.oracle.com/javase/7/docs/technotes/tools/solaris/serialver.html

- Die `serialVersionUUID` wird vor dem Deserialisieren mit der Klasse abgeglichen. Falls der Wert nicht übereinstimmt, wird die Deserialisierung abgebrochen und es tritt eine Ausnahme auf.
- Die Serialisierung bzw. Deserialisierung ist abhängig vom verwendeten Softwarestand. Eine Deserialisierung kann bei unterschiedlichen Softwareständen zu Problemen führen [JAX]. Da in der Praxis unterschiedliche Softwarestände (zum Beispiel nach der Aktualisierung der JVM) vorkommen, ist diese Form der Serialisierung bzw. Deserialisierung nicht in einer Produktivumgebung praxistauglich.
- Da Akka standardmäßig die Java-Standardserialisierung verwendet, ist keine zusätzliche Konfiguration oder Einbindung nötig. Es ist darauf zu achten, dass die zu verarbeitenden Klassen von der Schnittstelle `Serializable` erben. `Case-Klassen` in Scala erfüllen diese Anforderungen automatisch und können daher verwendet werden.

## 3.2 JSON Serialisierer/De-Serialisierer (Circe)

JSON ist ein Datenaustauschformat, dass für den Menschen einfach les- und schreibbar ist. Für Maschinen ist dieses Format einfach zu parsen und zu generieren, da die Analyse der Datenstruktur nicht aufwendig ist [JSO].

Es basiert auf einer Untermenge der Programmiersprache JavaScript. JSON ist unabhängig von der verwendeten Programmiersprache. Dadurch findet dieses Format eine hohe Verbreitung in der Speicherung von Daten und im Datenaustausch [JSO].

### Das Format JSON verglichen mit Scala

JSON wurde im Jahr 1999 in `ECMA-262 dritte Edition` standardisiert und sieht exemplarisch wie folgt aus:

```
{
  "age" : 5,
  "name" : "MyData",
  "tags" : [ "tag0", "tag1", "tagN" ]
}
```

Dieses Beispiel zeigt die beiden Strukturen, auf welchen JSON basiert (vgl. [JSO]), auf:

- **Name/Wert Paare**: In Scala würde diese Struktur zum Beispiel als `Map` repräsentiert werden.
- **Eine geordnete Liste von Werten**: In Scala würde diese Struktur zum Beispiel als `Array` oder `List` repräsentiert werden.

### Implementierung durch Circe

Diese Arbeit geht nicht auf die Logik bzw. Implementierung dieses SerDes ein, wie bereits im Abschnitt 1.5 beschrieben. Die reine Serialisierungs- und Deserialisierungs-Logik wird durch die Bibliothek Circe[5] implementiert (mehr dazu im Abschnitt 2.4) und über einen eigenen SerDes mithilfe der Klasse `JsonSerializer` eingebunden (mehr dazu im Abschnitt 3). Circe wurde aus folgenden Gründen als Vertreter eines JSON-SerDes gewählt:

- Hohe Anzahl an beteiligten Entwicklern (über 145) (vgl. [GITa])
- Hohe Anzahl an Quellcode-Beiträgen (über 1.700) (vgl. [GITa])
- Viele und einschlägige Projekte verwenden diese Implementierung (vgl. [GITa])
- Direkt in Scala ohne Adapter verwendbar

### Ein Beispiel

Exemplarisch wird durch diese Form der Serialisierung aus dem Scala-Objekt

```
Car(id = 0, name = "BMW F30", horsePower = 200)
```

folgende JSON-Ausgabe:

```
{
  "id" : 0,
  "name" : "BMW F30",
  "horsePower" : 200
}
```

### Hinweise

- Circe serialisiert ein Objekt abstrahiert zu einem Objekt der Klasse `JSON`. Diese Klasse bietet die Möglichkeit über die Methode `toString(): String` die eigentliche JSON-Ausgabe als Zeichenkette zu repräsentieren. Diese Zeichenkette kann durch, die in Java bereitgestellte Methode `byte[] getBytes(String charsetName)` zu einer Byte-Folge über einen definierten Zeichensatz konvertiert werden. Bei der Deserialisierung sollte die Umkehrung mit dem gleichen Zeichensatz durchgeführt werden, um Kompatibilitätsprobleme zu vermeiden.
- Circe bereitet es Probleme, die JSON-Ausgabe nach Modifikation der Ursprungsklasse nach Serialisierung wieder interpretieren zu können. Das ist daraus resultierend, dass JSON geparst wird und zum Beispiel eine Änderung des Namens eines Attributs nicht erkennt. Abhilfe bei dieser Problemstellung schafft der Umweg aus Abschnitt 3.

5 siehe `https://github.com/circe/circe`

## 3.3 Google Protocol Buffers Serialisierer/De-Serialisierer (ScalaPB)

Protobuf ist ein binäres Datenformat mit einer eigenen Schnittstellen-Beschreibungssprache, entwickelt von der Firma Google. Protobuf ist unabhängig von der verwendeten Programmiersprache. Derzeit werden offiziell folgende Sprache unterstützt [GOO]:

- Java (damit auch in Scala - siehe Abschnitt 2.4)
- Python
- Objective-C
- C++
- Dart (erst ab Protobuf-Version 3)
- Go (erst ab Protobuf-Version 3)
- Ruby (erst ab Protobuf-Version 3)
- C# (erst ab Protobuf-Version 3)

#### Verwendung

1. Bei der Verwendung von Protobuf wird zunächst die Struktur der Daten definiert. Diese Definition wird als Schnittstellen-Beschreibung genutzt.
2. Diese Schnittstellen-Beschreibung wird von einem Protobuf-Compiler, zum Beispiel in Form einer Klasse, in eine Zielprogrammiersprache übersetzt.
3. Dem Entwickler stehen nun durch die generierten Klassen, Methoden zum Serialisieren und Deserialisieren zur Verfügung.

#### Schnittstellen-Beschreibung und Scala

Die Struktur der Daten wird innerhalb einer `.proto`-Datei definiert. Eine Definition wird hier exemplarisch aufgezeigt:

```
message CarDb {
  int32 id = 1;
  int32 horsepower = 2;
  string name = 3;
}
```

- `message CarDb`: Definiert den Klassennamen in Scala (`case class CarDB(...)`).
- `int32 id = 1`: Deklariert ein Attribut in Scala (`id: Int`). `int32` gibt hierbei den Datentyp des Attributs an. `= 1` weist dieses Attribut einer Protobuf-Feldnummer zu. Diese Nummer sollte innerhalb einer Klasse nicht doppelt vorkommen und wird für die interne Verarbeitung beim Serialisieren und Deserialisieren benötigt.

### Implementierung durch ScalaPB

Diese Arbeit geht nicht auf die Logik bzw. Implementierung dieses SerDes ein, wie bereits im Abschnitt 1.5 beschrieben. Diese Arbeit verwendet den Protobuf-Compiler ScalaPB[6]. ScalaPB wurde aus folgenden Gründen als Vertreter eines Protobuf-Compilers für Scala gewählt:

- Die Recherche nach einem Protobuf-Compiler für Scala lieferte nur zwei sinnvolle Ergebnisse (ScalaPB und protoless[7]). Die Entwicklung von ScalaPB hat im Vergleich zu protoless eine wesentlich höhere Beteiligung (42 Entwickler zu einem Entwickler).
- ScalaPB lässt sich direkt in SBT als Plugin einbinden und wird vor dem Kompilieren des Scala-Quellcodes angestoßen.

ScalaPB stellt nach dem Kompilieren Scala-Klassen mit diversen Methoden für die Serialisierung und Deserialisierung bereit:

- `.toByteArray`: Diese Methode serialisiert ein Objekt zu einer Byte-Folge.
- `.parseFrom`: Diese Methode deserialisiert eine Byte-Folge zu einem Objekt. Bei dieser Methode handelt es sich nicht um eine statische Methode, d.h. es muss erst ein konkretes Objekt erzeugt werden, um diese zu verwenden. Alle Attribute dieses Objekts werden mit `null` initialisiert, was für einen Scala-Entwickler unüblich ist. Erst durch den Aufruf der Methode wird das Objekt mit sinnvollen Daten befüllt.
- `.getField`: Diese Methode deserialisiert ein einzelnes Attribut aus einer Byte-Folge.

**Hinweis**: Die Aufzählung ist nicht vollständig.

### Ein Beispiel

Exemplarisch wird durch die Protobuf-Serialisierung aus dem Scala-Objekt

```
CarDb(id = 0, name = "BMW F30", horsepower = 200)
```

diese Byte-Folge:

```
BMW F30
```

**Hinweis**: Um diese Byte-Folge darzustellen, wurde diese in eine Zeichenkette konvertiert. Nicht druckbare Zeichen werden nicht abgebildet.

6 siehe `https://github.com/scalapb/ScalaPB`
7 siehe `https://github.com/julien-lafont/protoless`

#### Hinweise

- Die durch dieses Verfahren serialisierten Objekte können auch nach einer Veränderung der Protobuf-Beschreibung wieder deserialisiert werden (mehr dazu im Abschnitt 3.3).
- Diese Arbeit geht auf `proto3` ein und nicht auf den Vorgänger (`proto2`) (mehr dazu im Abschnitt 1.5). Das Schlüsselwort `required` steht in `proto3` nicht mehr zur Verfügung. Dadurch kann in Scala nicht mehr direkt ein optionales Attribut (zum Beispiel `Option[Int]`) abgebildet werden.
- Die grundlegende Protobuf-Beschreibungssprache kann durch zusätzliche Bibliotheken erweitert werden. So ist es zum Beispiel möglich, über die Bibliothek `google/protobuf/wrappers.proto` ein optionales Attribut (über `google.protobuf.Int32Value` auf `Option[Int]`) abbilden zu können. Falls das Attribut bei einer in der Vergangenheit serialisierten Byte-Folge nicht abgebildet wurde, wird das Attribut nicht mit dem Wert `None` deserialisiert, sondern mit dem Wert `null`.
- Klassen, die durch ScalaPB erzeugt worden sind, bieten passende Schnittstellen an, die mit der Akka-internen Protobuf-Integration kompatibel sind. Daher reicht es aus, wenn ein SerDes unter `akka.remote.serialization.ProtobufSerializer` bekannt gemacht und den zu verarbeitenden Klassen zugewiesen wird (siehe Abschnitt 3).
- Akka selbst verwendet Protobuf, um Nachrichten zwischen Aktoren serialisieren bzw. deserialisieren zu können [Akkh]. Wenn eine Nachrichten nicht als Protobuf kompatible Klasse vorliegt, muss diese erst zu einer Byte-Folge serialisiert und in ein Protobuf kompatibles Objekt übertragen werden. Dieser Schritt kann eingespart werden, wenn Klassen aus ScalaPB verwendet werden.

#### Abwärtskompatible Deserialisierung in Protobuf

Die durch Protobuf serialisierten Objekte können auch nach einer Veränderung der Protobuf-Beschreibung wieder deserialisiert werden. In der Praxis kann dies zu Konflikten (zum Beispiel eine Änderung des Datentyps eines Attributs von `Float` auf `Double`) führen, da die Byte-Folge nicht mehr korrekt deserialisiert werden kann. Um das Risiko eines Konflikts zu reduzieren, können folgende Empfehlungen beachtet werden:

- Protobuf-Feldnummern dürfen innerhalb einer Klasse nicht mehrfach vergeben werden.
- Der Datentyp eines Attributs sollte nicht nachträglich geändert werden. Falls das Attribut bei einer in der Vergangenheit serialisierten Byte-Folge anders abgebildet worden ist, scheitert die Deserialisierung.
- Bei Entfernung eines Attributs aus einer Klasse sollte dieses nicht gelöscht, sondern nur auskommentiert werden. Dadurch wird die Gefahr einer doppelten und damit falschen Feldnummern-Zuweisung durch zum Beispiel einen anderen Entwickler reduziert.
- Es sollten keine Protobuf-Attribute (trotz gleicher Protobuf-Feldnummer) umbenannt werden, da diese sonst beim Deserialisieren nicht immer richtig interpretiert werden.

- Bei der Verwendung des Schlüsselworts `repeated` (in Scala als `Vector` abgebildet) ist Vorsicht geboten. Falls das Attribut bei einer in der Vergangenheit serialisierten Byte-Folge nicht abgebildet worden ist, wird das Attribut nicht mit dem Wert `Vector.empty` deserialisiert, sondern mit dem Wert `null`.
- Bei der Verwendung von Attributen vom Typ `Boolean` ist ebenfalls Vorsicht geboten. Falls das Attribut bei einer in der Vergangenheit serialisierten Byte-Folge nicht abgebildet worden ist, wird das Attribut nicht mit dem Wert `null` deserialisiert, sondern mit dem Wert `false`.

# 4 Fazit

Um die Schnelligkeit und Praxistauglichkeit (siehe Definitionen im Abschnitt 1.4) der SerDes zu eruieren, wurden zwei Experimente konzipiert:

- **Experiment E1** ***Vollständige Umgebung***: Dieses Experiment wurde entwickelt, um die ausgewählten SerDes in einem vollständigen Umfeld zu testen. Ein vollständiges Umfeld stellt einen minimalen Akka Persistence Aufbau mit einem Test-Aktorensystem und mit einem Test-Aktor dar. Das Experiment und das vollständige Umfeld werden im Abschnitt 2.4.1 beschrieben.
- **Experiment E2** ***Benchmark Umgebung***: Dieses Experiment testet isoliert die ausgewählten SerDes bezüglich Geschwindigkeit über ScalaMeter. Das Experiment wird im Abschnitt 2.4.2 beschrieben.

Diese Experimente wurden auf drei verschiedenen Referenzsystemen durchgeführt:

- **Referenzsystem R1** ***Windows 10***
- **Referenzsystem R2** ***iMac***
- **Referenzsystem R3** ***AWS EC2***

Auszüge der Eigenschaften der Referenzsysteme können dem Abschnitt A.1 entnommen werden.

## 4.1 Ergebnisse aus den Durchführungen der Experimente

Es wurden folgende Durchführungen durchgeführt:

1. **D1 von E1 und E2** auf Referenzsystem R1 *Windows 10*
2. **D1 von E1 und E2** auf Referenzsystem R2 *iMac*
3. **D1 von E1 und E2** auf Referenzsystem R3 *AWS EC2*
4. **D2 von E1 und E2** auf Referenzsystem R3 *AWS EC2*

Alle gesetzten Test-Parameter, sowie die vollständigen Ergebnisse und Visualisierungen können dem Abschnitt A.3 entnommen werden.

**Hinweis**: Da die Testdaten vor jedem Lauf (vor der Durchführung von E1 und E2) automatisch generiert werden (mehr dazu im Abschnitt 2.4), ist ein globaler Vergleich nicht möglich.

#### D1 von E1 und E2 auf Referenzsystem R1 *Windows 10*

Die Ergebnisse werden in Abbildung A.1 visualisiert. Bei dieser Durchführung wurden E1 und E2 **3-mal** getestet:

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| **Avg.** | **3200s** | **3227s** | **3044s** | **891ms** | **243ms** | **20ms** |
| Med. | 3187s | 3191s | 3059s | 889ms | 247ms | 20ms |

Um mehr Ressourcen (zum Beispiel Arbeitsspeicher) auf dem Zielsystem nutzen zu können (mehr dazu im Abschnitt 2.4), wurde SBT wie folgt parametrisiert:
`export SBT_OPTS="-Xms4G -Xmx10G".`

- Die einzelnen Ergebnisse aus E1 unterliegen starken Messschwankungen (mehr dazu im Abschnitt 2.4). Ein Vergleich ist bei dieser Durchführung nicht aussagekräftig. Eine Vergrößerung der Testmenge war auf diesem Referenzsystem nicht möglich, da nicht genug Ressourcen (Arbeitsspeicher) zur Verfügung standen.
- Die Ergebnisse aus E2 sind aussagekräftig und ermöglichen es, die verschiedenen SerDes bezüglich Schnelligkeit gegenüberzustellen.

#### D1 von E1 und E2 auf Referenzsystem R2 *iMac*

Die Ergebnisse werden in Abbildung A.2 visualisiert. Bei dieser Durchführung wurden E1 und E2 **12-mal** getestet:

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| **Avg.** | **2225s** | **2703s** | **2270s** | **292ms** | **163ms** | **22ms** |
| Med. | 2229s | 2696s | 2270s | 292ms | 162ms | 19ms |

Um mehr Ressourcen (zum Beispiel Arbeitsspeicher) auf dem Zielsystem nutzen zu können (mehr dazu im Abschnitt 2.4), wurde SBT wie folgt parametrisiert:
`export SBT_OPTS="-Xms4G -Xmx13G".`

- Die einzelnen Ergebnisse aus E1 unterliegen starken Messschwankungen (mehr dazu im Abschnitt 2.4). Ein Vergleich ist bei dieser Durchführung nicht aussagekräftig. Eine Vergrößerung der Testmenge war auf diesem Referenzsystem nicht möglich, da nicht genug Ressourcen (Arbeitsspeicher) zur Verfügung standen.
- Die Ergebnisse aus E2 sind aussagekräftig und ermöglichen es, die verschiedenen SerDes bezüglich Schnelligkeit gegenüberzustellen.

#### D1 von E1 und E2 auf Referenzsystem R3 *AWS EC2*

Die Ergebnisse werden in Abbildung A.3 visualisiert. Bei dieser Durchführung wurden E1 und E2 **2-mal** getestet:

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| **Avg.** | **2902s** | **2013s** | **2765s** | **768ms** | **236ms** | **31ms** |
| Med. | 2902s | 2013s | 2765s | 768ms | 236ms | 31ms |

Um mehr Ressourcen (zum Beispiel Arbeitsspeicher) auf dem Zielsystem nutzen zu können (mehr dazu im Abschnitt 2.4), wurde SBT wie folgt parametrisiert:
`export SBT_OPTS="-Xms4G -Xmx14G".`

- Die einzelnen Ergebnisse aus E1 unterliegen starken Messschwankungen (mehr dazu im Abschnitt 2.4). Ein Vergleich ist bei dieser Durchführung nicht aussagekräftig. Eine Vergrößerung der Testmenge war auf diesem Referenzsystem möglich (siehe D2).
- Die Ergebnisse aus E2 sind aussagekräftig und ermöglichen es, die verschiedenen SerDes bezüglich Schnelligkeit gegenüberzustellen.

#### D2 von E1 und E2 auf Referenzsystem R3 *AWS EC2*

Die Ergebnisse werden in Abbildung A.4 visualisiert. Bei dieser Durchführung wurden E1 und E2 **2-mal** getestet:

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| **Avg.** | **28282s** | **23486s** | **20238s** | **3861ms** | **1085ms** | **156ms** |
| Med. | 28282s | 23486s | 20238s | 3861ms | 1085ms | 156ms |

Um mehr Ressourcen (zum Beispiel Arbeitsspeicher) auf dem Zielsystem nutzen zu können (mehr dazu im Abschnitt 2.4), wurde SBT wie folgt parametrisiert:
`export SBT_OPTS="-Xms4G -Xmx14G".`

- Durch die Vergrößerung der Testmenge war es möglich, die Messschwankungen zu reduzieren. Die Ergebnisse aus E1 sind daher aussagekräftig und ermöglichen es, die verschiedenen SerDes bezüglich Schnelligkeit gegenüberzustellen.
- Die Ergebnisse aus E2 sind aussagekräftig und ermöglichen es, die verschiedenen SerDes bezüglich Schnelligkeit gegenüberzustellen.

## 4.2 Schnelligkeit der Serialisierer/De-Serialisierer

Bereits der Abschnitt 4.1 zeigte, dass es nicht einfach ist, aussagekräftige Ergebnisse zu erhalten:

| Durchführung | Referenzsystem | E1 aussagekräftig | E2 aussagekräftig |
|---|---|---|---|
| D1 | Referenzsystem R1 *Windows 10* | nein | ja |
| D1 | Referenzsystem R2 *iMac* | nein | ja |
| D1 | Referenzsystem R3 *AWS EC2* | nein | ja |
| D2 | Referenzsystem R3 *AWS EC2* | ja | ja |

Erst durch eine starke Vergrößerung der Testmenge mit

- `numberOfUpdates = 1000000` (D1 auf Referenzsystem R3 *AWS EC2*) auf
- `numberOfUpdates = 5000000` (D2 auf Referenzsystem R3 *AWS EC2*)

konnte auch ein aussagekräftiges Ergebnis von der Durchführung von E1 ermittelt werden.

Die Durchführungen von E2 lieferten immer aussagekräftige Ergebnisse. Wenn man die einzelnen SerDes isoliert miteinander vergleicht, ist es sinnvoll diese bezüglich Schnelligkeit gegenüberzustellen.

#### Schlussfolgernd

- Da viele Faktoren zu starken Messschwankungen (siehe Abschnitt 2.4) führen, ist es nicht einfach, die gewählten SerDes bezüglich Schnelligkeit miteinander zu vergleichen.
- Erst durch eine starke Vergrößerung der Testmenge konnte ein aussagekräftiges Ergebnis provoziert werden. Eine solch große sequentielle Verarbeitungsmenge ist in der Praxis selten vertreten. Die reine Serialisierung und Deserialisierung nimmt in einer Akka Persistence Umgebung nur wenig Laufzeit in Anspruch. Daher ist es sinnvoll, das gesamte System bezüglich Schnelligkeit erst an anderen Stellen (zum Beispiel durch das Austauschen von Sortieralgorithmen oder das Einbauen von schnellerer Hardware) zu optimieren.
- Soll das System aber an dieser Stelle optimiert werden, können die SerDes isoliert (wie in E2 gezeigt) miteinander bezüglich Geschwindigkeit verglichen werden.

## 4.3 Praxistauglichkeit der Serialisierer/De-Serialisierer

Praxistauglichkeit[1] ist nicht direkt messbar und hängt stark vom Anwendungsfall ab. Diese Arbeit geht von einem einfachen Anwendungsfall aus:

- Das System verwaltet Daten konsistent.
- Das System sollte auch nach Neustart seinen letzten Zustand vollständig wiederherstellen können.
- Das System wird im Laufe des Softwarelebenszyklus modifiziert. Daten, die nicht mehr zu dem neuen System passen, müssen aber dennoch korrekt verarbeitet werden.

Aus diesem Anwendungsfall lassen sich exemplarisch folgende Anforderungen an den SerDes ableiten:

- **Anforderung 1**: Korrektes Deserialisieren von Byte-Folgen auch nach Veränderung der Ursprungsklasse nach dem Serialisieren;
- **Anforderung 2**: Geringer Konfigurationsaufwand für einen Entwickler;

1 siehe Definition im Abschnitt 1.4

- **Anforderung 3**: Geringer Aufwand bei der Fehlersuche für einen Entwickler.

Durch diese Anforderungen lassen sich die SerDes in diesem Umfeld miteinander vergleichen.

#### Vergleich nach Anforderung 1

- **Java-Standardserialisierung**: Diese Form der Serialisierung/Deserialisierung **ist nicht praxistauglich**, da diese zum Beispiel keine korrekte Deserialisierung auch nach Veränderung der Ursprungsklasse nach dem Serialisieren garantieren kann (mehr dazu im Abschnitt 3.1). Dies ist zwar über einen Umweg (mehr dazu im Abschnitt 3) möglich, wirkt sich aber negativ auf die Anforderung 2 und 3 aus.
- **JSON-Serialisierung durch Circe**: Diese Form der Serialisierung/Deserialisierung **ist nicht praxistauglich**, da diese zum Beispiel keine korrekte Deserialisierung auch nach Veränderung der Ursprungsklasse nach dem Serialisieren garantieren kann (mehr dazu im Abschnitt 3.2). Dies ist zwar über einen Umweg (mehr dazu im Abschnitt 3) möglich, wirkt sich aber negativ auf die Anforderung 2 und 3 aus.
- **Protobuf-Serialisierung durch ScalaPB**: Diese Form der Serialisierung/Deserialisierung **ist praxistauglich**, da diese zum Beispiel eine korrekte Deserialisierung auch nach Veränderung der Ursprungsklasse nach dem Serialisieren garantieren kann. Es gelten jedoch Einschränkungen, die im Abschnitt 3.3 beschrieben werden.

Bei dieser Gegenüberstellung siegt die **Protobuf-Serialisierung durch ScalaPB**.

#### Vergleich nach Anforderung 2

- **Java-Standardserialisierung**: Diese Form der Serialisierung/Deserialisierung muss nicht explizit konfiguriert werden, da diese standardmäßig verwendet wird (mehr dazu im Abschnitt 3). Um eine korrekte Deserialisierung auch nach Veränderung der Ursprungsklasse nach dem Serialisieren ermöglichen zu können, muss sich mit einem Umweg (siehe Abschnitt 3) beholfen werden.
- **JSON-Serialisierung durch Circe**: Diese Form der Serialisierung/Deserialisierung kann nicht direkt verwendet werden und muss über einen eigenen SerDes in Form einer Klasse eingebunden und in der Konfigurationsdatei angegeben werden (mehr dazu im Abschnitt 3). Um eine korrekte Deserialisierung auch nach Veränderung der Ursprungsklasse nach dem Serialisieren ermöglichen zu können, muss sich mit einem Umweg (siehe Abschnitt 3) beholfen werden.
- **Protobuf-Serialisierung durch ScalaPB**: Diese Form der Serialisierung/Deserialisierung wird in Akka Persistence automatisch unterstützt, da Akka selbst Protobuf benutzt, um Nachrichten zwischen Aktoren serialisieren bzw. deserialisieren zu können. Daher ist die Konfiguration nicht aufwändig (mehr dazu im Abschnitt 3.3).

Bei dieser Gegenüberstellung siegt die **Protobuf-Serialisierung durch ScalaPB**.

#### Vergleich nach Anforderung 3

- **Java-Standardserialisierung**: Das Produkt dieser Serialisierung ist nicht vom Menschen lesbar. Fehler, die durch diese Serialisierung entstehen, können nicht einfach gefunden und behoben werden.
- **JSON-Serialisierung durch Circe**: Das Produkt dieser Serialisierung ist vom Menschen lesbar. Fehler, die durch diese Serialisierung entstehen, können einfach gefunden und behoben werden.
- **Protobuf-Serialisierung durch ScalaPB**: Das Produkt dieser Serialisierung ist nicht vom Menschen lesbar. Fehler, die durch diese Serialisierung entstehen, können nicht einfach gefunden und behoben werden.

Bei dieser Gegenüberstellung siegt die **JSON-Serialisierung durch Circe**.

#### Schlussfolgernd

- Die Praxistauglichkeit ist nicht direkt messbar und hängt stark vom Anwendungsfall ab.
- Erst durch den Anwendungsfall und die daraus abgeleiteten Anforderungen an den SerDes, können diese mit einander verglichen werden.
- Die Java-Standardserialisierung eignet sich gut für die lokale Entwicklung, da nichts extra konfiguriert werden muss. Diese Form der Serialisierung bzw. Deserialisierung sollte nicht in einer Produktivumgebung verwendet werden.
- Die JSON-Serialisierung durch Circe eignet sich für die lokale Entwicklung, da das Produkt der Serialisierung vom Menschen lesbar ist. Das kann bei der Fehlersuche helfen. Diese Form der Serialisierung bzw. Deserialisierung kann in einer Produktivumgebung verwendet werden.
- Die Protobuf-Serialisierung durch ScalaPB eignet sich, auf Grund zahlreicher Vorteile (mehr dazu im Abschnitt 3.3), für eine Produktivumgebung.

# A Anhänge

## A.1 Auszüge aus den Eigenschaften der Referenzsysteme

#### Auszug der Eigenschaften des Referenzsystem R1 *Windows 10*

| Name der Eigenschaft | Wert der Eigenschaft |
|---|---|
| Betriebssystem | Windows 10 Pro |
| Architektur: | 64-Bit |
| Prozessorbezeichnung: | Intel(R) Core(TM) i7-2600 3.40 GHz |
| Prozessor Anzahl der Kerne: | 4 |
| Arbeitsspeicher: | 12 GB |
| Festplatte Art: | HDD |

#### Auszug der Eigenschaften des Referenzsystem R2 *iMac*

| Name der Eigenschaft | Wert der Eigenschaft |
|---|---|
| Bezeichnung: | iMac (21,5", Ende 2013) |
| Betriebssystem | macOS Sierra Version 10.12.6 |
| Architektur: | 64-Bit |
| Prozessorbezeichnung: | 2,7 GHz Intel Core i5 |
| Prozessor Anzahl der Kerne: | 4 |
| Arbeitsspeicher: | 16 GB 1600 MHz DDR3 |
| Festplatte Art: | SSD |

#### Auszug der Eigenschaften des Referenzsystem R3 *AWS EC2*

| Name der Eigenschaft | Wert der Eigenschaft |
|---|---|
| EC2-Image-Bezeichnung: | ami-0bdf93799014acdc4 |
| EC2-Instanz-Bezeichnung: | t2.xlarge |
| Betriebssystem: | Ubuntu Server 18.04 LTS (HVM) |
| Architektur: | 64-Bit |
| Prozessor Anzahl der Kerne: | 4 |
| Arbeitsspeicher: | 16 GB |
| Festplatte Art: | SSD |

## A.2 Versionen der verwendeten Komponenten

| Bezeichnung | Version |
|---|---|
| Java JDK[1] | 8 |
| Scala[2] | 2.12.7 |
| Sbt[3] | 1.2.6 |
| Akka Actors[4] | 2.5.18 |
| Akka Persistence[5] | 2.5.18 |
| Circe[6] | 0.10.0 |
| LevelDB JNI[7] | 1.8 |
| Port of LevelDB to Java[8] | 0.7 |
| ScalaMeter[9] | 0.8.2 |
| ScalaPB[10] | 0.8.1 |

[1] `https://www.oracle.com/technetwork/java/javase/`
[2] `https://www.scala-lang.org/`
[3] `https://www.scala-sbt.org/`
[4] `https://doc.akka.io/docs/akka/2.5/actors.html`
[5] `https://doc.akka.io/docs/akka/2.5/persistence.html`
[6] `https://circe.github.io/circe/`
[7] `https://github.com/fusesource/leveldbjni`
[8] `https://github.com/dain/leveldb`
[9] `https://scalameter.github.io/`
[10] `https://github.com/scalapb/ScalaPB`

Die verwendenden Komponenten werden im Abschnitt 2.4 beschrieben.

## A.3 Ergebnisse aus den Durchführungen der Experimente

### Ergebnisse aus den Experimenten Referenzsystem R1 *Windows 10*

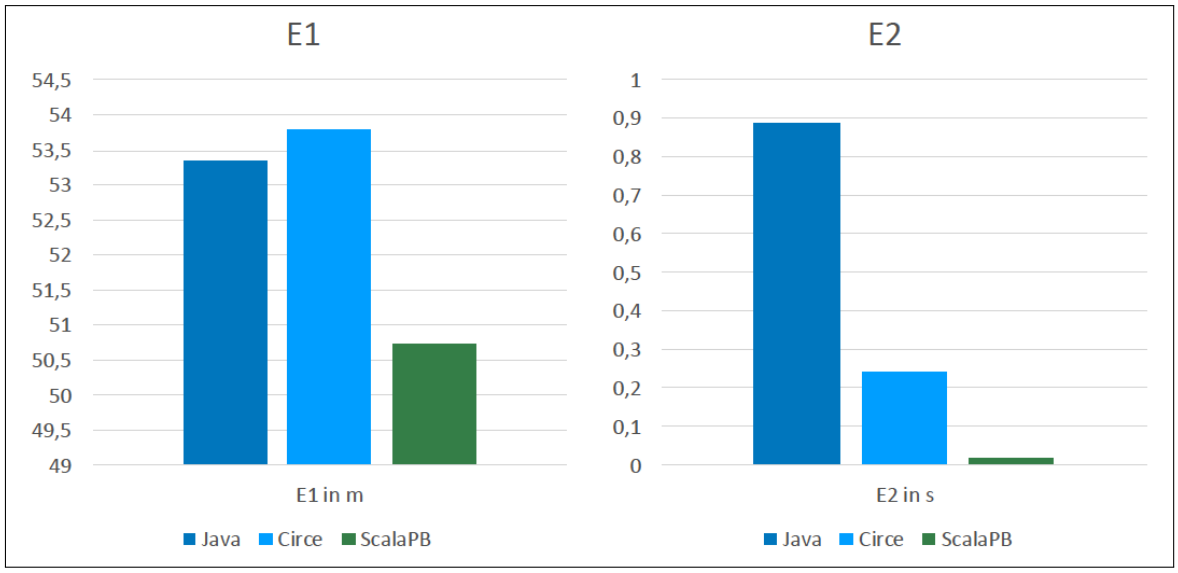


**Abbildung A.1** Visualisierung der Ergebnisse Durchführung 1 (Referenzsystem R1 *Windows 10*)

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| 02.02.2019 | 3164s | 3191s | 3059s | 889ms | 265ms | 20ms |
| 02.02.2019 | 3250s | 3330s | 3072s | 900ms | 216ms | 19ms |
| 03.02.2019 | 3187s | 3160s | 3001s | 883ms | 247ms | 20ms |
| **Avg.** | **3200s** | **3227s** | **3044s** | **891ms** | **243ms** | **20ms** |
| Med. | 3187s | 3191s | 3059s | 889ms | 247ms | 20ms |

Die Daten aus der Tabelle werden in der Abbildung A.1 visualisiert. Die Zahlen wurden kaufmännisch ohne Nachkommastellen gerundet. Die Testparameter lauteten:

| Konfiguration | Name | Wert |
|---|---|---|
| testSet | numberOfTestCars | 100000 |
| testSet | carNameStringMaxLength | 200 |
| testSet | complexCarNotesStringMaxLength | 900 |
| experimentMode | timeoutInSeconds | 60000 |
| experimentMode | actorSnapshotInterval | 100000 |
| experimentMode | numberOfAdds | 100000 |
| experimentMode | numberOfUpdates | 5000000 |
| experimentMode | testCar | wahr |
| experimentMode | testComplexCar | wahr |
| experimentMode | waitForProfilerEnter | falsch |
| benchmarkMode | numberOfSingleTests | 10000 |
| benchmarkMode | testCar | wahr |
| benchmarkMode | testComplexCar | wahr |

### Ergebnisse aus den Experimenten Referenzsystem R2 *iMac*

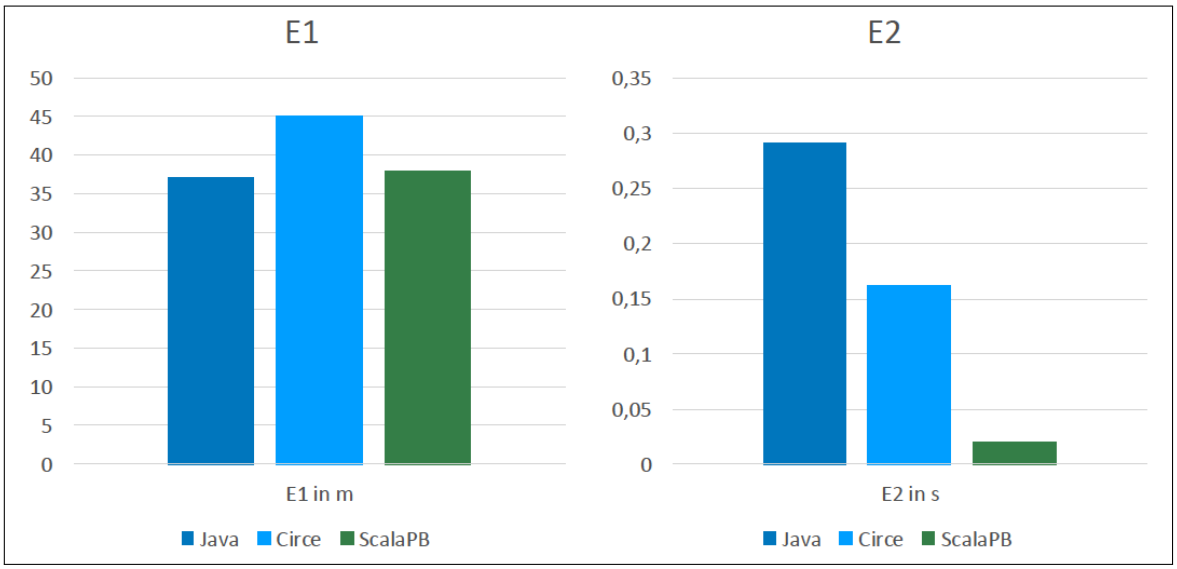


**Abbildung A.2** Visualisierung der Ergebnisse Durchführung 1 (Referenzsystem R2 *iMac*)

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| 28.01.2019 | 2245s | 2752s | 2325s | 290ms | 159ms | 18ms |
| 28.01.2019 | 2193s | 2696s | 2320s | 295ms | 161ms | 19ms |
| 29.01.2019 | 2217s | 2728s | 2277s | 296ms | 168ms | 34ms |
| 01.02.2019 | 2287s | 2741s | 2266s | 294ms | 163ms | 19ms |
| 02.02.2019 | 2218s | 2659s | 2200s | 298ms | 162ms | 21ms |
| 02.02.2019 | 2189s | 2764s | 2319s | 289ms | 163ms | 18ms |
| 02.02.2019 | 2240s | 2695s | 2274s | 292ms | 164ms | 18ms |
| 02.02.2019 | 2244s | 2624s | 2230s | 291ms | 171ms | 18ms |
| 03.02.2019 | 2245s | 2694s | 2293s | 292ms | 165ms | 19ms |
| 03.02.2019 | 2240s | 2688s | 2249s | 296ms | 159ms | 35ms |
| 03.02.2019 | 2196s | 2701s | 2259s | 285ms | 162ms | 19ms |
| 04.02.2019 | 2189s | 2694s | 2232s | 290ms | 155ms | 19ms |
| **Avg.** | **2225s** | **2703s** | **2270s** | **292ms** | **163ms** | **22ms** |
| Med. | 2229s | 2696s | 2270s | 292ms | 162ms | 19ms |

Die Daten aus der Tabelle werden in der Abbildung A.2 visualisiert. Die Zahlen wurden kaufmännisch ohne Nachkommastellen gerundet. Die Testparameter lauteten:

| Konfiguration | Name | Wert |
|---|---|---|
| testSet | numberOfTestCars | 100000 |
| testSet | carNameStringMaxLength | 200 |
| testSet | complexCarNotesStringMaxLength | 900 |
| experimentMode | timeoutInSeconds | 60000 |
| experimentMode | actorSnapshotInterval | 100000 |
| experimentMode | numberOfAdds | 100000 |
| experimentMode | numberOfUpdates | 5000000 |

| | | |
|---|---|---|
| experimentMode | testCar | wahr |
| experimentMode | testComplexCar | wahr |
| experimentMode | waitForProfilerEnter | falsch |
| benchmarkMode | numberOfSingleTests | 10000 |
| benchmarkMode | testCar | wahr |
| benchmarkMode | testComplexCar | wahr |

### Ergebnisse aus den Experimenten Referenzsystem R3 *AWS EC2*

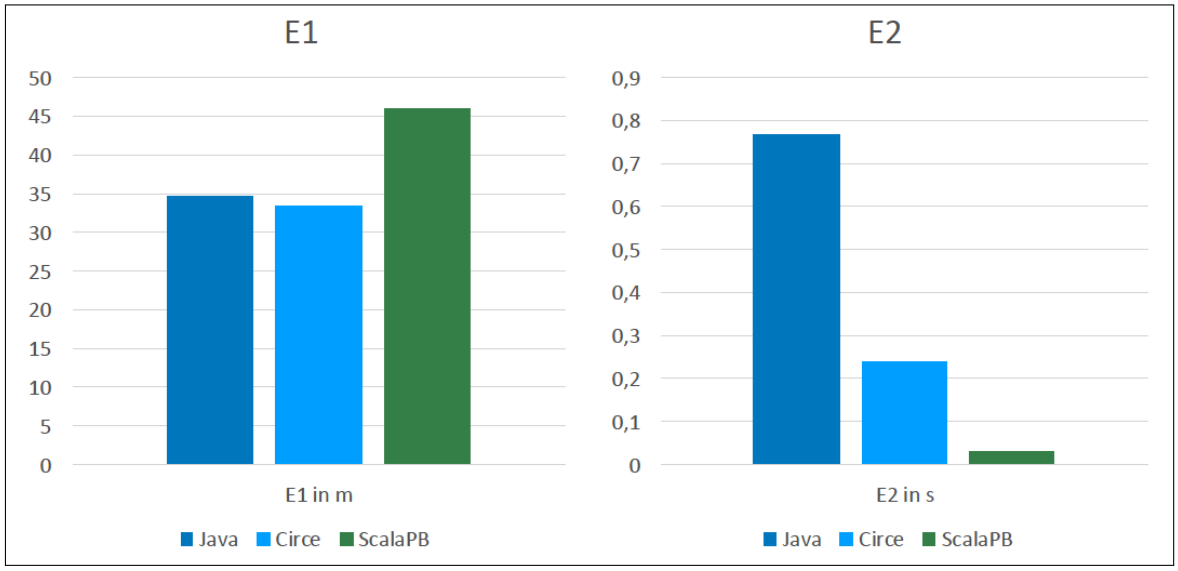


**Abbildung A.3** Visualisierung der Ergebnisse Durchführung 1 (Referenzsystem R3 *AWS EC2*)

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| 04.01.2019 | 2083s | 1958s | 1802s | 779ms | 219ms | 31ms |
| 04.01.2019 | 2100s | 2067s | 3727s | 756ms | 253ms | 31ms |
| **Avg.** | **2902s** | **2013s** | **2765s** | **768ms** | **236ms** | **31ms** |
| Med. | 2902s | 2013s | 2765s | 768ms | 236ms | 31ms |

Die Daten aus der Tabelle werden in der Abbildung A.3 visualisiert. Die Zahlen wurden kaufmännisch ohne Nachkommastellen gerundet. Die Testparameter lauteten:

| Konfiguration | Name | Wert |
|---|---|---|
| testSet | numberOfTestCars | 100000 |
| testSet | carNameStringMaxLength | 200 |
| testSet | complexCarNotesStringMaxLength | 900 |
| experimentMode | timeoutInSeconds | 60000 |
| experimentMode | actorSnapshotInterval | 100000 |
| experimentMode | numberOfAdds | 100000 |
| experimentMode | numberOfUpdates | 1000000 |
| experimentMode | testCar | wahr |
| experimentMode | testComplexCar | wahr |

| | | |
|---|---|---|
| experimentMode | waitForProfilerEnter | falsch |
| benchmarkMode | numberOfSingleTests | 10000 |
| benchmarkMode | testCar | wahr |
| benchmarkMode | testComplexCar | wahr |

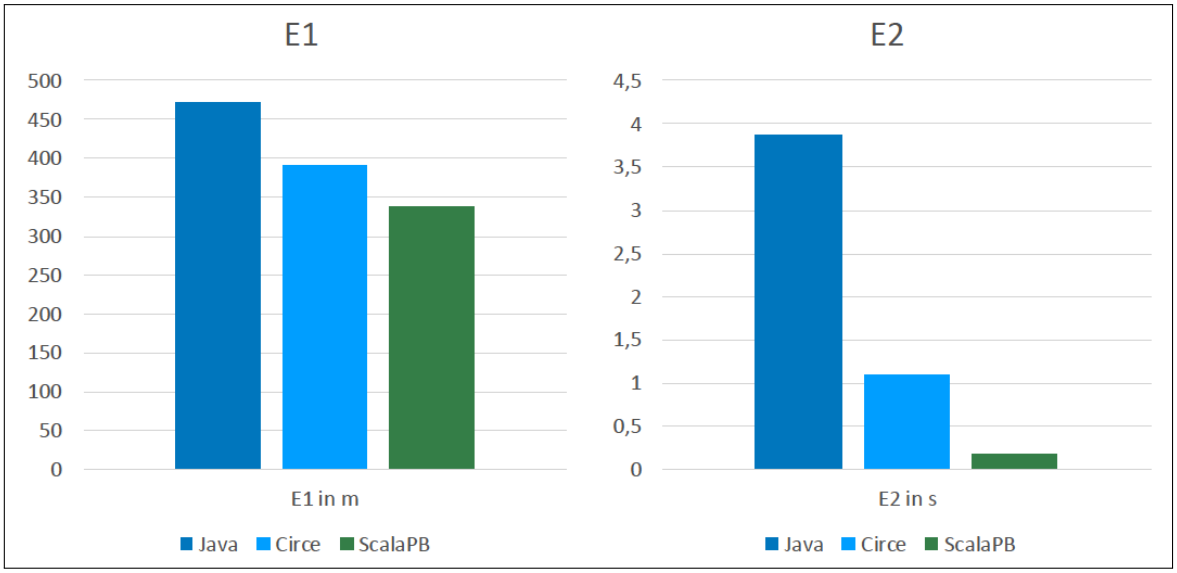


**Abbildung A.4** Visualisierung der Ergebnisse Durchführung 2 (Referenzsystem R3 *AWS EC2*)

| Datum | E1 Java | E1 Circe | E1 ScalaPB | E2 Java | E2 Circe | E2 ScalaPB |
|---|---|---|---|---|---|---|
| 05.01.2019 | 28280s | 23415s | 20257s | 3940ms | 1108ms | 152ms |
| 06.01.2019 | 28283s | 23557s | 20219s | 3781ms | 1062ms | 160ms |
| **Avg.** | **28282s** | **23486s** | **20238s** | **3861ms** | **1085ms** | **156ms** |
| Med. | 28282s | 23486s | 20238s | 3861ms | 1085ms | 156ms |

Die Daten aus der Tabelle werden in der Abbildung A.4 visualisiert. Die Zahlen wurden kaufmännisch ohne Nachkommastellen gerundet. Die Testparameter lauteten:

| Konfiguration | Name | Wert |
|---|---|---|
| testSet | numberOfTestCars | 100000 |
| testSet | carNameStringMaxLength | 200 |
| testSet | complexCarNotesStringMaxLength | 900 |
| experimentMode | timeoutInSeconds | 60000 |
| experimentMode | actorSnapshotInterval | 100000 |
| experimentMode | numberOfAdds | 100000 |
| experimentMode | numberOfUpdates | 5000000 |
| experimentMode | testCar | wahr |
| experimentMode | testComplexCar | wahr |
| experimentMode | waitForProfilerEnter | falsch |
| benchmarkMode | numberOfSingleTests | 50000 |
| benchmarkMode | testCar | wahr |
| benchmarkMode | testComplexCar | wahr |

# Literaturverzeichnis


[ACM] The ACM Digital Library - A universal modular ACTOR formalism for artificial intelligence. https://dl.acm.org/citation.cfm?id=1624804. Abgerufen am 28.02.2019.

[Akka] Akka.io (2018): Offizielle Dokumentation - Multiple persistence plugin configurations. https://doc.akka.io/docs/akka/2.5/persistence.html#multiple-persistence-plugin-configurations. Abgerufen am 08.03.2019.

[Akkb] Akka.io (2018): Offizielle Dokumentation - A Word About Java Serialization. https://doc.akka.io/docs/akka/2.5.4/java/serialization.html#a-word-about-java-serialization. Abgerufen am 08.03.2019.

[Akkc] Akka.io (2018): Offizielle Dokumentation - Ausschnitt Serialisierung - Creating new Serializers. https://doc.akka.io/docs/akka/2.5/serialization.html#customization. Abgerufen am 08.03.2019.

[Akkd] Akka.io (2018): Offizielle Dokumentation. https://akka.io/docs. Abgerufen am 09.11.2018.

[Akke] Akka.io (2018): Offizielle Dokumentation - Ausschnitt Akka Persistence mit Event Sourcing. https://doc.akka.io/docs/akka/current/persistence.html?language=scala#event-sourcing. Abgerufen am 20.12.2018.

[Akkf] Akka.io (2018): Offizieller Internetauftritt. https://akka.io. Abgerufen am 09.11.2018.

[Akkg] Akka.io (2018): Offizielle Dokumentation - Ausschnitt Akka Persistence. https://doc.akka.io/docs/akka/current/persistence.html#introduction. Abgerufen am 20.12.2018.

[Akkh] Akka.io (2018): Offizielle Dokumentation - Ausschnitt Serialisierung. https://doc.akka.io/docs/akka/current/serialization.html?language=scala#introduction. Abgerufen am 26.02.2019.

[ALV] Alvin Alexander - Scala: How to add new methods to existing classes. https://alvinalexander.com/scala/scala-how-to-add-new-methods-to-existing-classes. Abgerufen am 04.03.2019.

[BUL] Bullhost - Definition bzw. Erklaerung: Binaerkompatibel. https://www.bullhost.de/b/binaerkompatibel.html. Abgerufen am 28.02.2019.

[CQR] CQRS - Provided by Edument - Entire FAQ. http://www.cqrs.nu/faq. Abgerufen am 28.02.2019.

[GITa] GitHub-Seite von Circe. https://github.com/circe/circe. Abgerufen am 01.01.2019.

[GITb] GitHub-Seite von LevelDB JNI. https://github.com/fusesource/leveldbjni. Abgerufen am 01.01.2019.

[GITc] GitHub-Seite von Port of LevelDB to Java. https://github.com/dain/leveldb. Abgerufen am 01.01.2019.

[GITd] GitHub-Seite von ScalaMeter. https://scalameter.github.io. Abgerufen am 01.01.2019.

[GITe] GitHub-Seite von ScalaPB. https://github.com/scalapb/ScalaPB. Abgerufen am 01.01.2019.

[GOO] Google - Protocol Buffers. https://developers.google.com/protocol-buffers/. Abgerufen am 09.03.2019.

[Jav] Java SE Development Kit 8 Download Seite. https://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html. Abgerufen am 01.01.2019.

[JAX] Jaxenter - Java Serialisierung. https://jaxenter.de/aus-der-java-trickkiste-java-serialisierung-wann-passt-sie-wann-nicht-35558. Abgerufen am 09.03.2019.

[JSO] JSON - Einführung in JSON. https://www.json.org/json-de.html. Abgerufen am 09.03.2019.

[Kru09] G. Krueger. Serialisierung. In *Handbuch der Java-Programmierung (ISBN: 978-3-8273-2874-8,3-8273-2373-8)*, S. 963. Addison-Wesley, 2009.

[MAR] Martin Fowler - CommandQuerySeparation. https://martinfowler.com/bliki/CommandQuerySeparation.html. Abgerufen am 28.02.2019.

[MIC] Microsoft - Pattern: Event sourcing. https://microservices.io/patterns/data/event-sourcing.html. Abgerufen am 28.02.2019.

[Ode08] M. Odersky. Good actors style. In *Programming in Scala (ISBN: 0-9815316-0-1,978-0-9815316-0-1)*, S. 597. Artima Press, 2008.

[OPE] Rheinwerk Openbook - Java ist auch eine Insel - Persistente Objekte und Serialisierung. http://openbook.rheinwerk-verlag.de/javainsel9/javainsel_17_010.htm#mjfbe8cb1105d7dfaf6adbc23f31c81b93. Abgerufen am 09.03.2019.

[Pac18] V. F. Pacheco. Understanding event sourcing. In *Microservice patterns and best practices (ISBN: 978-1-78847-120-6,1-78847-120-2)*, S. 115. Packt Publishing, 2018.

[PICa] Brianstorti - The actor model in 10 minutes - Bild Exemplarische Darstellung Aktorenmodell. https://www.brianstorti.com/the-actor-model/. Abgerufen am 28.02.2019.

[PICb] Heise Developer - Bild Exemplarische Darstellung CQRS. https://heise.cloudimg.io/width/610/q80.png-lossy-80.webp-lossy-80.foil1/_www-heise-de_/developer/imgs/06/9/7/9/0/2/0/abb2-8f91b55dc4f69adb.png. Abgerufen am 11.03.2019.

[PICc] Wikimedia Commons - Bild Exemplarische Darstellung FIFO. https://commons.wikimedia.org/wiki/File:Fifo_queue.png. Abgerufen am 28.02.2019.

[SCAa] Offizielle Seite der Programmiersprache Scala - Seamless integration with Java. https://www.scala-lang.org/old/node/25. Abgerufen am 04.03.2019.

[SCAb] Offizielle Seite der Programmiersprache Scala. https://www.scala-lang.org. Abgerufen am 01.01.2019.

[Scac] Offizielle Dokumentation von ScalaMeter - Executors. https://scalameter.github.io/home/gettingstarted/0.5/executors/. Abgerufen am 08.03.2019.

[Scad] Offizielle Dokumentation von ScalaMeter - Generators. http://scalameter.github.io/home/gettingstarted/0.7/generators/index.html. Abgerufen am 08.03.2019.

[Scae] Offizielle Dokumentation von ScalaMeter - Simple benchmark. http://scalameter.github.io/home/gettingstarted/0.7/simplemicrobenchmark/index.html. Abgerufen am 08.03.2019.

[SCAf] Offizielle Seite des Build-Werkzeugs SBT. https://www.scala-sbt.org. Abgerufen am 01.01.2019.

[SCAg] Offizielle Seite der Programmiersprache Scala - Traits. https://docs.scala-lang.org/tour/traits.html. Abgerufen am 04.03.2019.

[TEC] Techopedia - Bytecode. https://www.techopedia.com/definition/3760/bytecode. Abgerufen am 04.03.2019.

[Ull14] C. Ullenboom. Die eigene SUID. In *Java SE 8 Standard-Bibliothek (ISBN: 978-3-8362-2874-9)*, S. 658–659. Galileo Computing, 2014.

[UNI] Informatik UNI Hamburg - Typisierung. https://wr.informatik.uni-hamburg.de/_media/teaching/sommersemester_2018/ep-18-schnieders-typisierung-praesentation.pdf. Abgerufen am 04.03.2019.