Philipps Universität Marburg

FACHBEREICH MATHEMATIK UND INFORMATIK

Arbeitsgruppe Verteilte Systeme

# Identifikation von fehlerbehafteten Daten zur energiebewußten Programmierung

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von

René Frank

| | |
|---|---|
| Gutachter: | Prof. Dr. Bernd Freisleben |
| Betreuer: | Dipl.-Inf. Pablo Graubner |
| Ausgabedatum: | 24.04.2012 |
| Abgabedatum: | 13.08.2012 |

Philipps-Universitt Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße
35032 Marburg

# Zusammenfassung

In vielen Konzernen nehmen die Kosten des rapide wachsenden Energieverbrauchs einen immer höheren Stellenwert ein. Gerade große Rechenzentren benötigen stetig höhere Rechenleistungen, um die wachsende Zahl an neuen Online-Services bereitstellen zu können. Mit den zusätzlich stetig steigenden Energiekosten verstärkt sich immer mehr die Notwendigkeit, in diesem Bereich Einsparungsmöglichkeiten vorzunehmen. Die Forcierung entsprechend energieeffizienter Technologien ist der Schlüssel zum Erfolg. Diese Bachelorarbeit baut auf dem Paradigma *Accuracy Awareness*, zur energieeffizienten Verarbeitung von “Big Data” in Rechenzentren, auf. *Accuracy Awareness* bedient sich der Eigenschaft vieler Anwendungen, fehlertolerant bzw. unempfindlich gegenüber niedrigen Fehlerraten zu sein. Handelsübliche Hardware verwendet jedoch einen hohen Anteil ihres Energieverbrauchs zur Garantie von Fehlerfreiheit. Durch das Anpassen der Fehlerfreiheit auf ein akkurates Maß, sollen an diesem Punkt Einsparungen erzielt werden. In der erstellten Arbeit wurde zu diesem Zweck ein Tool zur Fehlerinjektion für Datenströme geschrieben. Die Anwendung erlaubt es Streams, die fehlerbelastet sein dürfen, um eine spezielle Fehler-Annotation zu erweitern. In die eingelesenen Daten des markierten Streams werden durch die Annotation und diverser Injektionsstrategien Fehler eingestreut und somit die Daten manipuliert. Ein wesentlicher Bestandteil der Arbeit war die dynamische Regulierbarkeit der Fehlerinjektion zur Laufzeit.

# Danksagung

An erster Stelle möchte ich mich bei Herrn Prof. Dr. B. Freisleben für die Möglichkeit bedanken, diese Arbeit in seiner Arbeitsgruppe durchzuführen.

Besonders möchte ich mich bei Pablo Graubner für die investierte Zeit und die gute Betreuung, sowohl auf menschlicher als auch auf fachlicher Ebene, die er mir zu teil werden ließ herzlich bedanken.

Danken möchte ich auch meinen Freunden, die immer an meiner Seite stehen und mir bei der Arbeit mit Verständnis und guten Worten geholfen haben.

Der größte Dank gilt allerdings meiner Familie, die mich in meiner Ausbildung sowohl in der Schule, als auch im Studium immer tatkräfig unterstützt hat und auf die ich mich immer verlassen kann. Ohne sie wäre diese Arbeit sicher nicht zustande gekommen.

# Inhaltsverzeichnis

**ZUSAMMENFASSUNG** I

**DANKSAGUNG** II

**INHALTSVERZEICHNIS** IV

**1 Einleitung** 1

**2 Grundlagen** 3
- 2.1 Accuracy Awareness Data Processing . . . 3
- 2.2 Serialisierung . . . 4
- 2.3 RMI . . . 5
- 2.4 JMX . . . 6
- 2.5 Jmxtools . . . 8
- 2.6 Javassist . . . 8
- 2.7 HotSwapper . . . 9

**3 Verwandte Arbeiten** 11
- 3.1 EnerJ . . . 11
- 3.2 Green-Framework . . . 12
- 3.3 Probabilistic Accuracy Bounds . . . 13
- 3.4 Loop Perforation . . . 13

**4 Design** 15
- 4.1 Motivation und Zielsetzung . . . 15
- 4.2 Benutzerinteraktion . . . 16
- 4.3 Klassendesign & -abhängigkeiten . . . 17
  - 4.3.1 StreamProcessor . . . 18
  - 4.3.2 Fehlerwerte . . . 19
  - 4.3.3 Injektions-Strategie . . . 20
  - 4.3.4 Dynamische Konfiguration . . . 23
  - 4.3.5 JMX Agent . . . 24

4.3.6 Ausnahmebehandlungen . . . . . 24
4.4 Verschiedene Clients . . . . . 26
4.5 Sequenzieller Ablauf . . . . . 27

**5 Implementierung** **29**
5.1 Fehlerwerte . . . . . 29
5.2 Klasse StreamProcessor . . . . . 31
5.3 Klasse AddRuntimeAnnotation . . . . . 33
5.4 Context . . . . . 37
5.5 Fehler Injektion . . . . . 38
5.6 JMX Implementierung . . . . . 39
5.6.1 Controller/MainServer . . . . . 39
5.6.2 Client Implementierung . . . . . 41

**6 Evaluation/Messungen** **44**
6.1 Testaufbau . . . . . 44
6.2 Lese- und Schreiboperatinen . . . . . 45
6.3 Dynamische Konfiguration der Fehlerwerte . . . . . 45
6.4 Injektionsstategien . . . . . 46
6.5 Branch Prediction . . . . . 52
6.6 Zusammenfassung . . . . . 52

**7 Fazit** **54**

**ABBILDUNGSVERZEICHNIS** **I**

**LISTINGS** **II**

**TABELLENVERZEICHNIS** **III**

**LITERATURVERZEICHNIS** **IV**

**ERKLÄRUNG** **IV**

Kapitel 1

# 1 Einleitung

In vielen großen Unternehmen nimmt das Thema Energieverbrauch einen immer höheren Stellenwert ein. Besonders der rapide Anstieg des Energiebedarfs von Rechenzentren und Serverräumen in den letzten Jahren stellt ein erhebliches Kostenproblem dar. Nicht zu vergessen sind die negativen Einflüsse auf die Umwelt[1], welche ein hoher Energieverbrauch zusätzlich mit sich bringt. Dennoch benötigen gerade große Rechenzentren stetig höhere Rechenleistungen, um die wachsende Zahl an neuen Online-Services bereitzustellen. Allein Google verbraucht 0,01 Prozent des gesamten weltweiten Stromverbrauchs und knapp ein Prozent des Strombedarfs aller weltweiten Rechenzentren [**?** ].
Durch den Einsatz moderner Technologien und neuer Verfahren ist es heute möglich den Energiebedarf eines Rechenzentrums deutlich zu reduzieren. Daraus resultieren wichtige Vorteile für Unternehmen und Gesellschaft:

- Die Betriebskosten eines Rechenzentrums sinken erheblich
- Strom- und Kühlleistungsbedarf sinkt
  - falls diese bereits im Grenzbereich liefen, können somit teure Neuanschaffungen vermieden werden
- positive Umweltauswirkungen

Dem Thema Energieeffizienz kommt heute demnach eine hohe Bedeutung zu [**?** ].
Diese Bachelorarbeit baut auf dem Paradigma *Accuracy Awareness*, zur energieeffizienten Verarbeitung von “Big Data” in Rechenzentren, auf. Das hierfür entwickelte Programm hat die Aufgabe Daten über Streams einzulesen und über deren Bytecode Fehlerwerte einzustreuen. Für dieses Verfahren, im weiteren Verlauf der Arbeit als Fehlerinjektion bezeichnet,

[1] bspw. CO2-Emission

sind verschiedene Fehlertypen mit unterschiedenlichen Injektionsalgorithmen definiert worden. Es wurde zusätzlich die Integration neuer Algorithmen bei den Designentscheidungen berücksichtigt. Die Fehlerinjektion konnte zunächst nur zur Compilezeit festgelegt werden. Im zweiten Schritt wurde die Anwendung mittels eines Agenten dynamisch regulierbar gemacht. Diese Bachelorarbeit gliedert sich in fünf Kapitel. Zuerst werden alle wesentlichen Grundlagen die zum besseren Verständnis der Arbeit hilfreich sind, sowie das Paradigma *Accuracy Awareness*, näher beleuchtet. Im Anschluss werden verwandte Arbeiten aus diesem Bereich vorgestellt, die verschiedenen Ansätze diskutiert und mit dem entwickelten Programm dieser Arbeit verglichen. Den Hauptteil bilden die Kapitel Design und Implementierung. Im Designkapitel werden das gewählte Programmdesign, die Architektur sowie der allgemeine Funktionsablauf der Anwendung detailliert beschrieben. Anhand verschiedener Diagramme soll die Funktionsweise verdeutlicht und die Abhängigkeiten der einzelnen Java-Klassen aufgezeigt werden. Auf die Imlementierung dieser Designentscheidungen wird im darauffolgenden Kapitel eingegangen. Die wichtigsten Codestellen und verwendeten Muster sind in diesem Abschnitt beschrieben. Den Abschluss bildet die Evaluation, bei der anhand einer Reihe aufgestellter Messungen, die Programmeigenschaften analysiert und die Verwendbarkeit der Anwendung diskutiert wird.

Kapitel 2

# 2 Grundlagen

In diesem Kapitel sind alle wesentlichen Elemente, die für die Erstellung der Anwendung verwendet wurden, kurz beschrieben. Zunächst wird das Paradigma *Accuracy Awareness* näher erläutert und dessen Bezug zu dieser Arbeit aufgezeigt. Im Anschluss wird ein Überblick über die verwendeten Technologien gegeben, welcher zu einem besseren Verständnis der übrigen Kapitel helfen soll.

## 2.1 Accuracy Awareness Data Processing

Durch die zunehmende Nutzung von Internet, Telekommunikationsdienstleistungen und IT-Netzwerken ist die Anzahl von Servern und deren Stromverbrauch in den letzten Jahren enorm angestiegen. Für eine energieeffiziente Verarbeitung von großen Datamengen in Rechenzentren, wurde deshalb mit *Accuracy Awareness* ein neues Paradigma integriert. In der heutigen Zeit verwendet gewöhliche Hardware einen hohen Anteil ihres Energieverbrauchs für die Garantie von Fehlerfreiheit. Mit dem Paradigma *Accuracy Awareness* wird versucht, an genau diesem Punkt Einsparungen zu erzielen. Es wird sich dabei zunutze gemacht, dass wichtige Klassen von Anwendungen fehlertolerant arbeiten bzw. gegenüber niedrigen Fehlerraten unempfindlich sind. Daraus resultiert die Möglichkeit Energie zu sparen, indem die Genauigkeit und Fehlerfreiheit auf ein akkurates Maß reduziert wird. Die Voraussetzung dafür ist die Unterscheidung zwischen fehlerfreien und fehlerbehafteten Daten. Dies ist vorallem wichtig, da beispielsweise Steuerinformationen[1] für einen korrekten Programmablauf nicht fehlerbehaftet sein dürfen. Andere Daten können hingegen in gewissen Grenzen Fehler enthalten, ohne die Berechnung zu gefährden.
In Abbildung 2.1 ist der Aufbau einer Software-Architektur, unter Verwendung des Paradigams,

[1] Programmcode, Datei-Header

dargestellt. Die Architektur besteht aus den vier Ebenen Job/Task, Middleware, Betriebssystem und Hardware.

- Die Job/Task-Ebene ist beispielsweise durch fehlerbehaftete Datentypen, fehlerbehaftetes lesen und schreiben, als auch fehlerbehaftete Kommunikation beschrieben. Auf dieser Ebene sind fehlertolerante Algorithmen implementiert. Diese haben die Eigenschaft, dass sie in ihren Ausführungen durch Datenverluste oder fehlerbehafteten Daten nicht beeinflusst werden.

- Die Ebene Middleware stellt Schnittstellen für eine verteilte Speicherung, Kommunikation und die Nutzung fehlerbehafteter Daten zur Verfügung.
  Das für diese Bachelorarbeit erstellte Programm arbeitet auf genau dieser Middleware Ebene. Dabei sind speziell die beiden Bereiche fehlerbehaftetes Dateisystem und fehlerbehaftetes Netzwerkprotokoll betroffen.

- Die Betriebssysstem-Ebene ist für die Unterscheidung zwischen fehlerbehafteten Speicherseiten sowie die Netzwerk-Kommunikation verantwortlich.

- Auf der Hardware-Ebene wird zwischen fehlerfreien und fehlerbehafteten Daten in den Bereichen CPU/Memory, Speicher und Netzwerk unterschieden. Damit die Hardware diese Unterscheidung treffen kann, müssen an ihr geringfügige Modifiktationen vorgenommen werden.

Durch *Accuracy Awareness* können Rechenzentren dauerhaft von ca. 10% bis 50% ihres Stromverbrauchs einsparen und damit zusätzlich einen wertvollen Beitrag für den Umweltschutz leisten [**?** ].

## 2.2 Serialisierung

Die Objektserialisierung bietet die Möglichkeit, Objekte auch ohne Datenbank persistent zu sichern. Dabei wird ein Objektzustand in einen Bytestrom eingepackt. Dieser kann nun problemlos über ein Netzwerk transferiert und auf der anderen Seite wieder ausgepackt werden. Es wird hierbei eine Kopie des aktuellen Objektes zurückgelesen. Um ein Objekt serialisierbar zu machen, muss lediglich das leere Interface $\mathrm{java.io.Serializable}$ implementiert werden. Die serialisierten Objekte erhalten eine eindeutige Versionsnummer. Trägt ein Objekt eine falsche Versionsnummer kann es nicht deserialisiert werden. Ist keine Nummer angegeben, so benutzt Java einen Hashwert, der sich unter anderem aus den Namen der Klassenattribute errechnet. Für die Deserialisierung wird der Datenstrom eingelesen und ein Java Objekt mit dem gespeicherten Zustand erzeugt [**?** ].

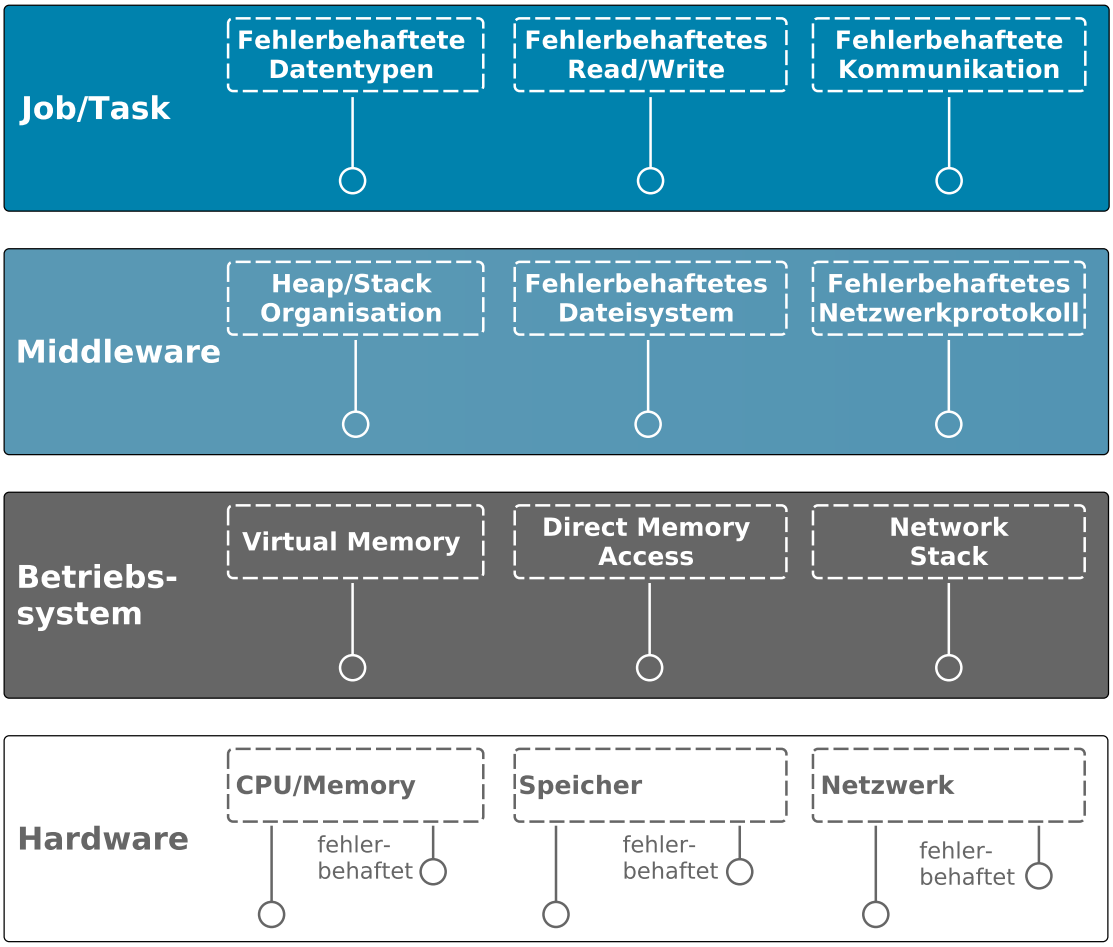



Abbildung 2.1: Software-Architektur [**?** ]

## 2.3 RMI

RMI[2] bezeichnet einen Mechanismus zur sogenannten verteilten Anwendungsprogrammierung. Vereinfacht ausgedrückt ermöglicht es den Zugriff auf Methoden eines entfernten Java-Objekts, welches auf einer anderen Virtuellen Machine und bei Bedarf auf einem anderen vernetzten Rechner läuft. Die Details dieses Netzwerkverbundes bleiben dem Aufrufer verborgen, so dass die Arbeit mit verteilten Objekten sich im Prinzip kaum von der mit lokalen Objekten unterscheidet. In Abbildung 2.2 ist diese Funktionsweise grafisch dargestellt. Auf der Client-Seite nutzt Java sogenannte Stubs zur Kapselung der Daten, welche über das Netzwerk verschickt werden sollen. Ein Stub implementiert das Remote-Interface und dient dem Client als Platzhalter für das Remote-Objekt. Der Stub kommuniziert über die Netzwerkverbindung mit dem Skeleton auf der Serverseite. Skeletons rufen daraufhin die gewünschte Methode auf,

[2]Remote Method Invocation

übergeben die Parameter und liefern das Resultat an den Client zurück. Sollen komplette Objekt übermitttelt werden, ist die Objekt Serialisierung erforderlich. Objekte, die als Parameter für RMI-Aufrufe dienen, müssen das Interface Serializable aus dem Package java.io implementieren. Die eigentliche Verbindungsaufnahme geschieht über die Serveradresse und einen Bezeichner. Der Bezeichner ist notwendig damit der Namensdienst auf dem Server eine Referenz auf das entfernte Objekt zurückliefern kann. Damit dies funktioniert muss sich das entfernte Objekt zuvor am Server, unter diesem Namen, beim Namensdienst registriert haben. Der RMI-Namensdienst wird über statische Methoden der Klasse java.rmi.Naming angesprochen [**?** ].

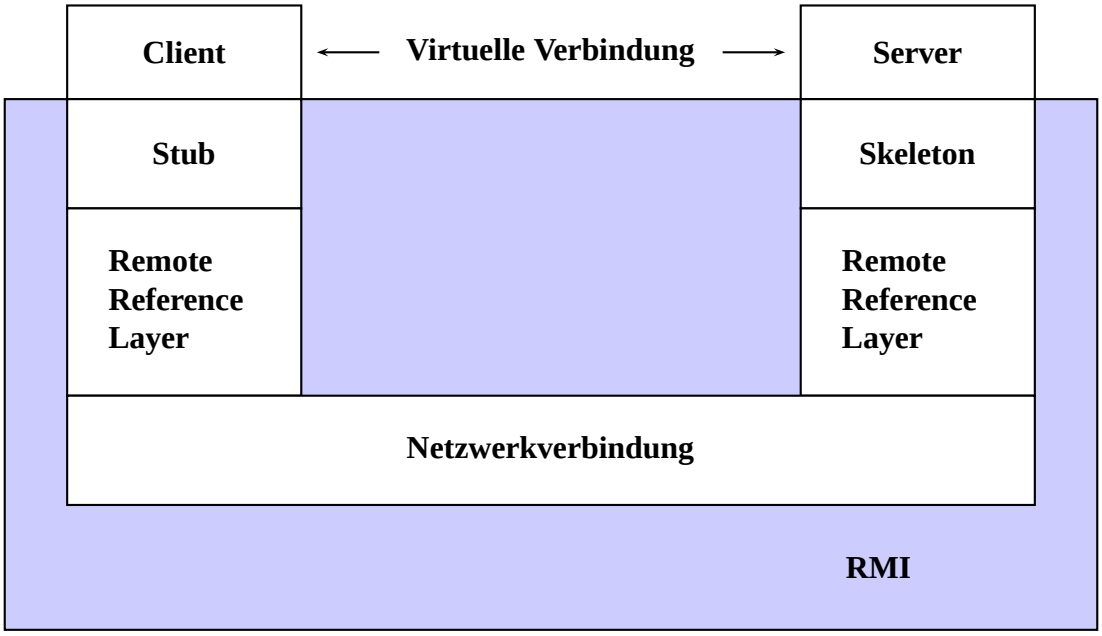



Abbildung 2.2: RMI Kommunikations-Architektur [**?** ]

## 2.4 JMX

Mittels JMX[3] ist es gelungen, Java Anwendungen und Dienste über einen standardisierten Weg zu verwalten und zu überwachen. Die damit entwickelte Spezifikation beschreibt eine Architektur, eine API, Design Patterns, verschiedene Verwaltungsdienste für Anwendungen sowie Monitoring Dienste für Java. Durch die Unterstützung von Adaptern und Konnektoren ermöglicht JMX die Kommunikation zwischen verschiedenen JVM[4]. Eine Anwendung kann somit beispielsweise über einen einfach implementierten HTTP-Adapter, mittels Webbrowser gesteuert werden. In Abbildung 2.3 sind die verschiedenen Ebenen der JMX-Architektur

[3] Java Management Extensions
[4] Java Virtual Machine

dargestellt [**?** ].
Die zu überwachenden Resourcen befinden sich auf der Ebene Instrumentation Level. Sie werden als MBeans[5] repräsentiert. MBeans gibt es in verschiedenen Variationen, wie beispielsweise Standard oder Dynamic MBeans. Damit der Client Zugriff auf die Resourcen bekommt, sind sogennante Agents notwendig. Diese befinden sich auf dem Agent Level[6] und sprechen die Resourcen direkt an. Diese Ebene wird deshalb auch als Mittelsmann zwischen Anwendung und den MBeans bezeichnet. Das Distributed Services Level dient als Schnittstelle, um den Zugriff der Management Applikationen auf die Agents, innerhalb des Servers zu ermöglichen. Die verbindung wird über sogenannte Connectors oder Adaptors aufgebaut. Der Connector liefert den vollen Zugang zu der MBeanServer API und kann via RMI, JMS die Verbindung herstellen. Ein Adaptor adaptiert die API auf ein anderes Protokoll wie SNMP oder auf webbasierte Oberflächen wie HTML/HTTP, WML/HTTP [**?** ].

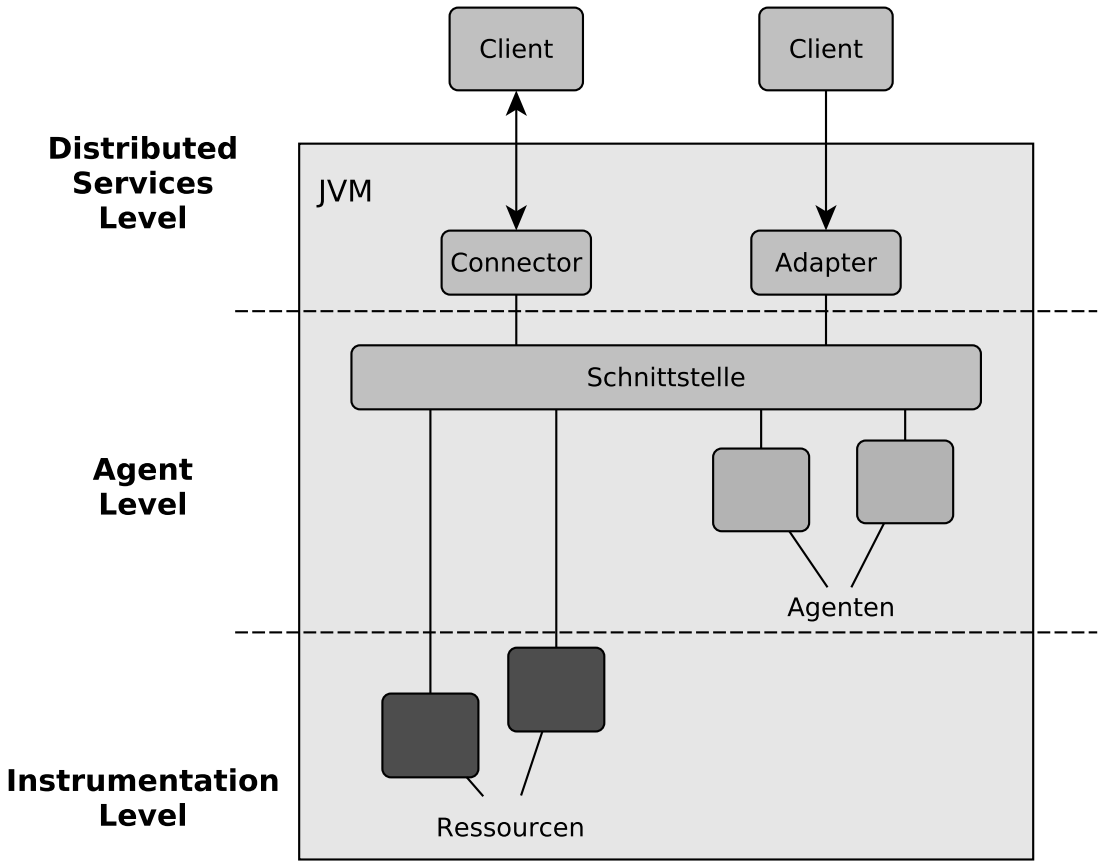



Abbildung 2.3: JMX-Architektur [**?** ]

[5] Managed Bean
[6] auch MBeanServer

## 2.5 Jmxtools

Die Java library Jmxtools stellt eine große Auswahl an nützlicher Zusatzfunktionalität für die JMX Implementierung bereit. In dieser Arbeit wurde die Library für die Implementierung eines HTML-Clients verwendet. JMXtools bietet zu diesem Zweck einen $\mathrm{HtmlAdaptorServer}$ an. Dieser ermöglicht es einen Zugang auf die Agent-View über einen Webbrowser zu erstellen. Anschließend können die verwalteten Ressourcen über den Webbrowser verwendet werden.

## 2.6 Javassist

Javassist[7] ist eine Java library mit deren Hilfe sich der Bytecode einer Java Anwendung leicht manipulieren oder neuer Bytecode erstellen lässt. Diese Eigenschaft war für die dynamische Modifikation der Fehlerwerte eine wichtige Voraussetzung.
Eine Java Klasse ist, sobald sie in die JVM geladen wurde, nicht mehr ohne weiteres veränderbar. Für diese Bachelorarbeit war es jedoch notwendig die Parameter einer Annotation zur Laufzeit modifizierbar zu halten. Da Annotationen im ursprünglichen Sinne, nur für den Compiler vorgesehen und deshalb auch nur zur Compilezeit veränderbar sind, musste auf Bytecode-Ebene gearbeitet werden. Javassist stellt hierfür die folgenden Operationen bereit.
Die wichtigste Klasse ist der $\mathrm{classPool}$, indem die zu manipulierenden Klassen eingefügt werden müssen. Anders als der JVM classloader, werden geladene Klassen durch die Javaassist API als Daten verwendbar gehalten. Klassen die in den $\mathrm{classPool}$ aufgenommen werden, sind Instanzen von $\mathrm{javassist.CtClass}$. Eine $\mathrm{CtClass}$ bietet zusätzlich zu den normalen Methoden der Class Datei die Möglichkeit neue Methoden, Felder und Konstruktoren der Klasse hinzuzufügen, sowie Klassenname, Superklasse und Interfaces abzuändern. Felder, Methoden und Konstruktoren werden durch $\mathrm{javassist.CtField}$, $\mathrm{javassist.CtMethod}$, und $\mathrm{javassist.CtConstructor}$ dargestellt, welche Methoden für ihre jeweiligen Modifikationen mit sich führen [**?** ]. Der allgemeine Modifikationsablauf ist in Abbildung 2.4 dargestellt. Hierbei wird die Ausgangsklasse über die Bytecode-Manipulation um neue Aspekte, in Abbildung 2.4 die um $\mathrm{if}$-Anweisung und die $\mathrm{@override}$ Annotation, erweitert. Die neu geladene Klasse enthält im Anschluss beide Elemente.
Javassist bietet damit ausreichend Funktionionalität um die gewünschte dynamische Regulierung umsetzen zu können, ohne wesentliche Auswirkungen auf die Laufzeit der Anwendung zu haben. In Kapitel **??** Evaluation, wurde diese Eigenschaft anhand einer Messung belegt.

[7] Java programming assistant: http://www.csg.ci.i.u-tokyo.ac.jp/∼chiba/javassist/

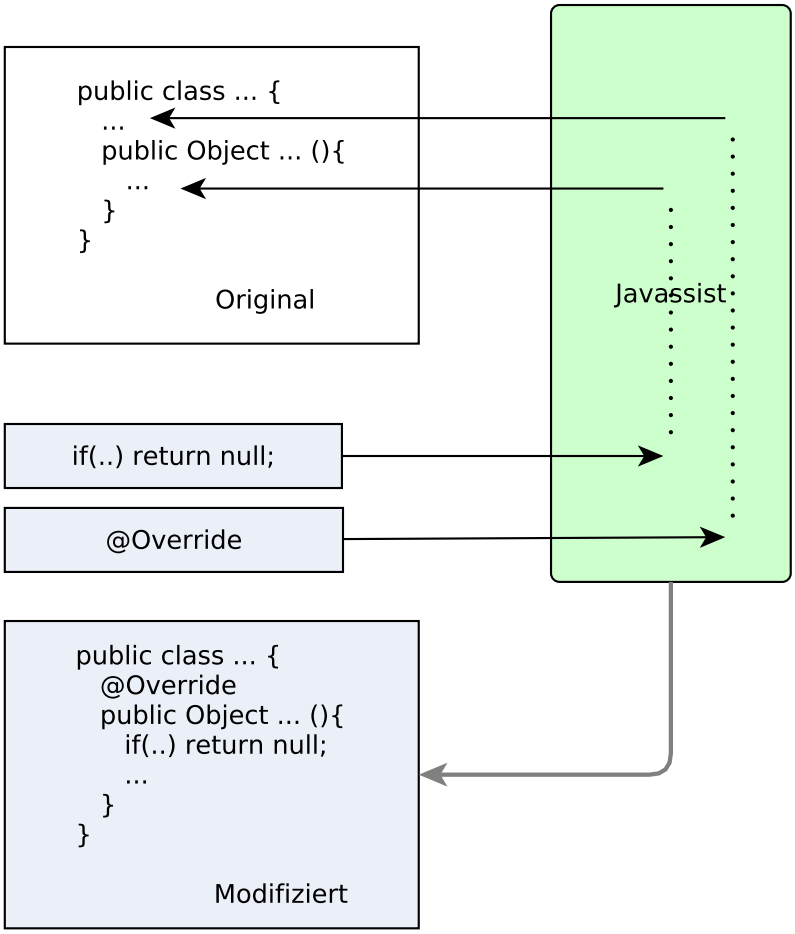



Abbildung 2.4: Code-Manipulation mit Javassist [**?** ]

## 2.7 HotSwapper

Wie bereits beschrieben ist es durch javassist möglich Klassen während der Laufzeit zu modifizieren und mit den neu erstellten Instanzen zu arbeiten. Sind allerdings bereits Instanzen der jeweiligen Klasse geladen, erhält man die Fehlermeldung: *duplicate class definition*. Um die durch javassist modifizierten Klassen respektive deren modifierzierten Bytecode für alle involvierten Objekte verwendbar zu machen, wurde javassist.util.HotSwapper[8] in die Implementierung eingebunden. Der Hotswapper benötigt zusätzlich die Einbindung der tools.jar library in den BuildPath und diverse VM-Argumente, die im Abschnitt Konfiguration im Kapitel Design erläutert werden. Die Funktionsweise der Klasse HotSwapper ist in Abbildung 2.5 grafisch dargestellt. Der neue Bytecode wird über den Debugger in die Class Datei des jeweiligen Objektes eingefügt und diese anschließend neu geladen. Für den Ladevorgang stellt

[8]http://www.csg.ci.i.u-tokyo.ac.jp/∼chiba/javassist/html/javassist/util/HotSwapper.html

die Klasse HotSwapper die Funktion reload bereit. Nach dem Update des Bytecodes und dem Ladevorgang, können alle betroffenen Klassen den neuen Code, beim Methodenaufruf ausführen.

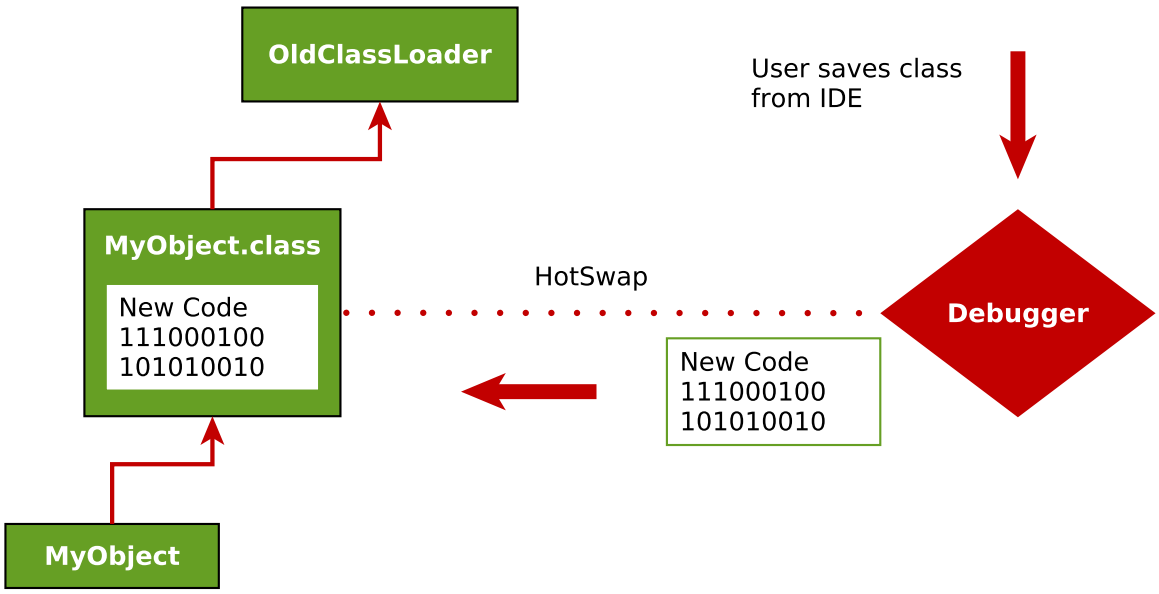



Abbildung 2.5: Funktionsweise HotSwapper [**?** ]

Kapitel 3

# 3 Verwandte Arbeiten

In der Informatik spielt das Thema Energieverbrauch eine besonders große Rolle. Es existieren daher bereits viele Ansätze und neue Verfahren zum effizienten Umgang mit der knappen Ressource Strom. Die in diesem Kapitel diskutierten Ansätze fussen auf vergangenen Studien die zeigten, dass viele Programme fehlertolerant gegenüber geringfügigen Fehlermengen sind [**?** ]. Verschiedene Anwendungen machen sich dieses Verhalten zunutze und erzielen mit der Differenzierung zwischen fehlerbehafteten und fehlerfreien Daten erstaunliche Ergebnisse.
Das entwickelte Programm dieser Bachelorarbeit, soll bei der Identifikation von fehlerbehafteten Daten eingesetzt werden. Konkret bearbeitet die Anwendung Datenströme, indem sie in deren Bytecode Fehlerwerte injiziert. Die Injektion geschieht über die Markierung der fehlerbehafteten Streams durch eine bestimmte Annotation. Das Setzen der Fehlerwerte ist zur Laufzeit dynamisch möglich, einzige Voraussetzung ist die gobale Deklaration der Streams. Die verwendete Annotation enthält Parameter wie Fehlerrate, die Blockgröße und den Fehlertyp. Die Datensätze können somit auf ein akkurates Maß mit Fehlern bzw. Datenverlusten injiziert werden. Letztendlich sollen dadurch Einsparungen bei der weiteren Bearbeitung erzielt und trotzdem das gewünschte Resultat geliefert werden. Solche Fehlerinjektionen wurden bereits in verschiedenen Anwendungen auf unterschiedlicher Weise eingeführt.

## 3.1 EnerJ

Eine dieser Anwendungen, mit vergleichbarer Funktionalität, ist EnerJ das eine Erweiterung der Programmiersprache Java bietet. EnerJ nutzt für die Unterscheidung zwischen fehlerfreien und fehlerbehafteten Elementen eindeutige Annotationen, welche als type-qualifier eingesetzt werden. Fehlerfreie Datentypen werden in EnerJ durch eine @Precise Annotation gekennzeichnet. Eine Annotierung präziser Datentypen ist allerdings optional, da der Default-Wert

@Precise ist. Die Kennzeichnung fehlerbehafteter Datentypen geschieht über eine @Approx Annotation. Auch in dieser Bachelorarbeit wurden Annotationen zur Fehlermarkierung eingesetzt. Ein wesentlicher Unterschied liegt in der Verwendung der Annotationen. Während EnerJ allgemeine Datentypen als approximiert oder exakt deklariert, sind die Annotationen dieser Bachelorarbeit, auschließlich auf eine variable Fehlerinjektion in Datenströmen ausgelegt.
Die Korrektheit der Daten kann in EnerJ, nur bei den präzisen/fehlerfreien Daten garantiert werden. Approximierte Daten haben diese Garantie nicht. Dies resultiert aus dem Grundsatz das ein Datentyp entweder präzise oder approximiert sein kann. Es gibt keine verschiedenen Ebenen der Approximation, so das auch keine Garantie bzgl. der Fehlergrenzen gegeben werden kann [**?** ]. In dieser Bachelorarbeit wurden dagegen verschiedene Ebenen eingeführt. Die Fehlerwerte der $\mathrm{FaultInj}$ Annotation können beliebig gewählt und auch zur Laufzeit angepasst werden.
Für die Implementierung von EnerJ wurde auf das Checker Framework von Papi et al. [**?** ] zurückgegriffen. Dieses erweitert das Java Typ-System und macht es um einiges vielseitiger. Es besteht aus Compiler Plug-ins sogenannten “checkers” und ist flexibel um weitere Compiler Plug-ins erweiterbar. Das Framework bringt einen besonderen Vorteil für die Fehlermarkierung mit sich. Indem es auch Annotationen für lokal definierte Variablen erlaubt, beitet es einen großen Handlungsspielraum. Lokale Annotationen sind urspünglich nur zur Auswertung für den Compiler vorgesehen und finden aus diesem Grund in meiner Implementierung keine Anwendung. Auf die Verwendung des Checker Frameworks habe ich jedoch bewusst verzichtet, da die Verwendung dieser Technologie eine erhebliche Laufzeitverschlechterung mit sich bringt.

## 3.2 Green-Framework

Ein weiterer ähnlicher Ansatz findet sich im Green-Framework von Woongki Baek und Trishul M. Chilimbi. Diese Anwendung bietet ein flexibles Framework, welches versucht unötige Genauigkeiten durch Approximationen zu kompensieren. Das Framework fokusiert sich dabei allerdings auf Approximationen für Schleifen und Funktionen, wodurch es sich von meiner Implementierung bereits abgrenzt. Jedoch bietet es ebenfalls verschiedene Ebenen der Fehlerinjektion. Der Entwickler hat die Möglichkeit einen maximalen Verlust bezüglich der Anforderungen/QoS[1] anzugeben. Das Green-Framework liefert im Anschluss eine Auswertung, inwieweit die Anwendung diesen Anforderung genügen wird. Um approximierte Funktionen nutzen zu können, muss der Entwickler diese zusätzlich bereitstellen. Schleifen können beispielsweise durch weniger Iterationen approximiert werden. Am Ende erstellt der Green

[1] Quality of Service

Compiler, auf Grundlage eines generierten QoS-Models mit dem urspünglichen Programm, eine energieeffizientere Anwendung mit den approximierten Funktionen [**?** ].

## 3.3 Probabilistic Accuracy Bounds

Die Arbeit von Martin Rinard, “Probabilistic Accuracy Bounds for Fault-Tolerant Computations that Discard Tasks” beschäftigt sich ebenfalls mit einer Methode die es ermöglicht, Berechnungen mit Datenfehlern auszuführen und dennoch einen vernünftigen Output zu erzeugen.
Die Berechnungen werden bei diesem Ansatz in einzelne Tasks eingeteilt. Bei der Ausführung werden Tasks, bei denen Fehler bzw. Störungen vorhanden waren, einfach verworfen. Die fehlerfreien Tasks setzen die Berechnung bis zum Ende fort. Das Modell verwendet als Grundlage zufällige Ausführungen eines Programms mit unterschliedlichen Task-Fehlerraten. Anhand dieser Ausführungen wird ein quantitatives, probabilistisches Modell erstellt. Das Modell charakterisiert die Verzerrung des Outputs. Durch die somit definierten Grenzen, leistet die Anwendung trotz Fehlern einen qualitativen Output. Weiterhin ist die Erstellung eines Timing-Modells[2] vorgesehen. Die Kombination der beiden Modelle erlaubt es die Genauigkeit der Ausführung zu reduzieren. Die Fehlerinjektion wird bei diesem Ansatz durch eine Injektion mit fehlerhaften Tasks vogenommen. Dadurch lässt sich Ausführungszeit sparen, während die Abweichung der Berechung innerhalb akzeptabler Grenzen liegt [**?** ].

## 3.4 Loop Perforation

Eine weitere Technik, um die Genauigkeit einer Anwendung zu reduzieren und damit einen Leistungszuwachs zu erzielen, ist die “loop perforation” von Stelios Sidiroglou-Douskos et al. Diese Anwendung versucht analog zum Green-Framework, eine bessere Performance durch effizientere Berechnungen in Schleifen zu erzielen. Im kern wird versucht die Anzahl der Iterationen von Schleifen zu reduzieren, um somit einen geringeren Rechenaufwand zu erhalten. Dieser Vorgang wird “Perforierung” genannt. Das Verfahren filtert für die Unterscheidung von fehlerfreien und fehlerbehafteten Daten zunächst die kritischen Schleifen heraus, dieser Vorgang nennt sich “Criticality Testing”. Alle kritischen Schleifen, sind Schleifen die durch die Perforierung einen Schaden verursachen. Alle nicht kritischen Schleifen können durch die Perforierung einen erheblichen Effizientzanstieg, bei einer weiterhin akzeptablen Genauigkeit erzielen. Die zweite Phase ist der “Perforation Space Exploration” Algorithmus. Alle nicht kritischen Schleifen werden in dieser Phase verarbeitet, indem versucht wird sie

[2] Ausführungszeit als Funktion über Task Fehlerraten

optimal zu kombinieren. Dadurch entsteht eine Menge von Pareto-optimalen[3] Varianten mit bestmöglicher Perfomance, bei einem bestimmten Genauigkeitsverlust [**?** ]. Die Anwendung fokusiert sich im Gegensatz zu meiner Implementierung auf Schleifen. Die Fehlerinjektion beginnt beim Criticality Testing und wird anschließend algorithmisch optimiert. Die Optimierung wird grundsätzlich durch eine Reduzierung von Schleifeniterationen vorgenommen. Die Anwendung dieser Bachelorarbeit bietet bezieht sich ausschließlich auf Datenströme. Zusätzlich wird vielseitigerer Injektionsmechanismus mit diversen Fehlertypen und variabel anpassbarer Fehlerrate angeboten.

[3] Zustand, in dem ein Individuum nicht besser gestellt werden kann

Kapitel 4

# 4 Design

Das Kapitel Design beschreibt zunächst die Problemstellung der zugrundeliegenden Arbeit. Zu diesem Zweck wird die bestehende Ausgangssituation aufgezeigt sowie eine kurze Einführung hinsichtlich der Funktionsweise der Anwendung gegeben. Ein Anwendungsfall beschreibt infolge einer dargestellten Benutzerinteraktion die zur Verfügung gestellten Operationen. Im Hauptteil werden die wichtigsten Klassen bezüglich deren Abhängigkeiten und Funktionsweisen vorgestellt. Für die Visualisierung dieser Eigenschaften wurde die graphische Modellierungssprache UML eingesetzt. Es werden allgemeine Designentscheidungen diskutiert und abschließend ein Gesamtüberblick anhand eines sequentiellen Programmablaufs gegeben.

## 4.1 Motivation und Zielsetzung

Es wurde bereits in den vorherigen Kapiteln erwähnt, dass Energie in der Zukunft eine immer knapper werdende Ressource sein wird. Gerade in der Informatik steigt der Energiekonsum aufgrund neuer Technologien und Online-Services stetig an [**?** ]. Das für diese Bachelorarbeit entwickelte Programm ist Bestandteil einer Forschung, welche versucht dieses Problem zu lösen, indem es mit Hilfe des Paradigmas Accuracy Aware eine energiebewusste Programmierung ermöglicht.
Die Zielsetzungen waren:

- die Fehlerinjektion minimal-invasiv, d.h. mit möglichst wenig Änderungen im Code vornehmen zu müssen.
- Nur einen minimalen Overhead erzeugen, um die Verwendung für große verteilte Systeme (z.B. Hadoop), bei denen neben der Fehlerinjektion zugleich Energiemessungen vorgenommen werden können, zu ermöglichen.

- Alle Arten von Datenströmen, wie FileIO oder Network, sollten berücksichtigt werden, weil für “Accuracy Awareness” sowohl modifizierte Disk- als auch Network-Hardware simuliert werden soll.

- Eine dynamische Regulierung der Fehlerwerte zur Laufzeit. Dem Client sollte es somit möglich sein Fehlerwerte, welche vorher im Programmcode festgelegt wurden, im laufenden Programm modifizieren zu können.

Das Programm verwendet zunächst gewöhnliche Streams um Daten einzulesen. Diese werden anschließend “verpackt” und über verschiedene Logiken mit Fehlerwerten injiziert. Danach werden die injizierten Daten für die weitere Verwendung ausgegeben. Die Fehlerinjektion wird über eine Markierung der betroffenen Streams durch Java Annotations[1] im Quelltext eingeleitet. Jeder markierte Stream wird somit als fehlerbehaftet gekennzeichnet und bekommt von Beginn an Fehlerwerte zugewiesen. Für die Markierungen wurde eine selbstdefinierte Annotation mit dem Namen $\mathrm{FaultInj}$ verwendet. Die Fehlerwerte sind durch die vier Parameter ID, Fehlerrate, Fehlertyp und die Größe eines fehlerbehafteten Datenblocks definiert. Ziel der Fehlerinjektion ist es, die eingelesen Daten mittels diesen konkreten Fehlerwerten und diversen Injektionsstrategien manipulieren zu können. Für die Fehlerinjektion war es zusätzlich notwendig, auch mehrere Markierungen/Annotationen mit verschiedenen Werten für einen Stream definieren zu können.

## 4.2 Benutzerinteraktion

Die Benutzerinteraktion wurde sehr übersichtlich gestaltet, um den Client eine möglichst kompakte, intuitive Bedienbarkeit zur Verfügung zu stellen. Für den Anwender wurde deshalb eine Schnittstelle mit dem MBean Server kreiert (Siehe dazu Abschnitt **??**), welche ausschließlich vier unterschiedliche Funktionalitäten anbietet.
In Abbildung **??** ist diese Benutzerinteraktion als Anwendungsfalldiagramm bzw. Nutzfalldiagramm dargestellt. Die ersten beiden Operationen der Anwendung mit ähnlicher Funktionalität, geben alle Fehlerwerte respektive die ID’s der annotierten Streams zurück. Dies ist insbesondere von Vorteil, wenn später die Fehlerwerte geändert werden sollen und die eindeutige ID des Streams nicht bekannt ist.
Um die Fehlerwerte der einzelnen Streams ändern zu können, gibt es eine Methode die in Abbildung **??** unter “Fehlerwerte ändern” zu finden ist. Die Änderungsmöglichkeiten umfassen die Fehlerrate, Fehlertyp und die größe des Datenblocks. Die ID dient zur Zuordnung der Fehlerwerte und kann ausschließlich im Programmcode verändert werden.
Die wichtigste Funktion hat den Namen $\mathrm{runInjection}$ und ist in Abbildung **??** als “Injektion

[1] Metadaten

durchführen” dargestellt. Für diese Funktion lassen sich zwei Varianten auswählen. Die erste Variante ist für eine Dateiinjektion definiert und benötigt lediglich den Dateipfad. Die zweite Variante verlangt einen Java $\mathrm{InputStream}$ um allgemeine Datenströme zu verarbeiten. Nach dem Laden der Daten wird durch diese Funktion die eigentliche Fehlerinjektion und die Datenausgabe veranlasst.
Für die Datenausgabe sind ebenfalls zwei Wahlmöglichkeiten vorhanden, deren genaue Differenzierung im weiteren Verlauf dieses Kapitels erläutert wird.

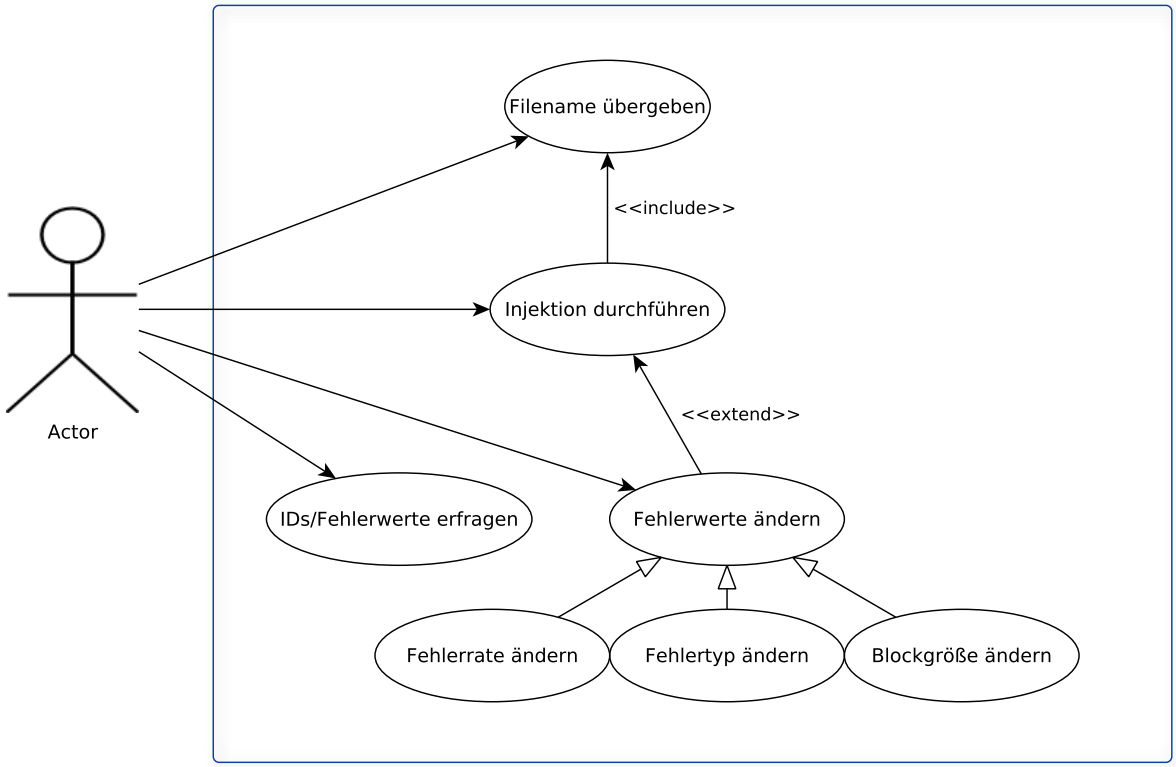



Abbildung 4.1: Anwendungsfalldiagramm

## 4.3 Klassendesign & -abhängigkeiten

Die Klassenhierarchie der zugrunde liegenden Anwendung stützt sich auf den folgenden vier Säulen, welche das Grundgerüst bzw. die Hauptfunktionalität der Applikation bilden. Ein wesentliches Ziel war es, die Anforderungen der Modularität zu erfüllen. Die Anwendung sollte für eventuelle Erweiterungen möglichst flexibel und gut strukturiert bleiben.

- StreamProcesser : Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy : Oberklasse der Injektionsstrategien

- AddRunTimeAnnotation : Dynamische Modifikation der Annotationen
- Controller : Steuerung aller Funktionen

### 4.3.1 StreamProcessor

Der StreamProcessor ist das Herzstück der Anwendung. Alle Daten die es zu manipulieren gilt, werden an diese Klasse übergeben. Die für das Einlesen der Daten notwenigen Streams sind in der Klassendefinition als globale Variablen vorgesehen. Dies bedeutet auch, dass die Fehlermarkierungen durch die FaultInj bzw. mehrere Markierungen durch eine FaultInjects Annotation, ausschließlich global vorgenommen werden können.
Der StreamProcessor zerlegt einen Datenstream in einzelne Bytes, um sie anschließend der Klasse Context zu übergeben. Die Klasse Context verwaltet die Daten zusammen mit deren Fehlerwerte, falls jene definiert wurden. Die Implementierung erlaubt es an dieser Stelle auch mehrere Datensätze verschiedener Streams durch ein Context Objekt verwalten zu lassen. Für jeden Datensatz wird eine eindeutige ID zur Unterscheidung festgelegt. Einzige Prämisse für die Nutzung dieser Datenverwaltung ist die Serialisierbarkeit der Daten, um einen Netzwerktransfer zu ermöglichen.
Nach dem Einlesevorgang wird über den StreamProcessor die Fehlerinjektion veranlasst. Bereits während des Einlesens wurden für jeden Datensatz, anhand der ID des markierten Streams, die Fehlerwerte registriert. Der StreamProcessor startet nun die Fehlerinjektion über seine Instanz der Klasse Context. Diese bestimmt anhand des Fehlertyps die notwendige Injektions-Strategie. Ist für einen Datensatz kein Strategie vorhanden bleibt dieser unberührt und eine Fehlermeldung wird ausgegeben. Für die konkrete Fehlerinjektion wurde ein Strategy Pattern in leicht abgewandelter Form verwendet.
Die aktuelle Implementierung sieht nach der Fehlerinjektion die Datenausgabe in eine Datei vor. Zu diesem Zweck wurde eine innere Klasse Reducer implementiert. Sie ist für die Erstellung der neuen Dateien zuständig und bietet hierfür zwei verschiedene Ausgabevarianten an. Per Default wird ein FileChannel für die Ausgabe verwendet. Durch eine zusätzliche Funktion kann der Ausgabestream vom Client, zu einem ObjectOutputStream geändert werden. Dadurch wird eine persistente Objektspeicherung ermöglicht. Nach diesem Schritt wird durch alle Datensätze iteriert und die Daten in die neuen Dateien geschrieben. Diese Struktur ist in Abbildung **??** als UML-Klassendiagramm dargestellt.

### Exkurs: Klasse Context

Bisher wurde häufig die Klasse Context erwähnt, welche die eingelesenen Daten verwaltet und Fehler in die enthaltenen Datensätze einbaut. Um diese Aufgabe zu erfüllen, muss die Klasse Context zunächst für jeden Datensatz die erforderliche Injektionslogik bestimmen.

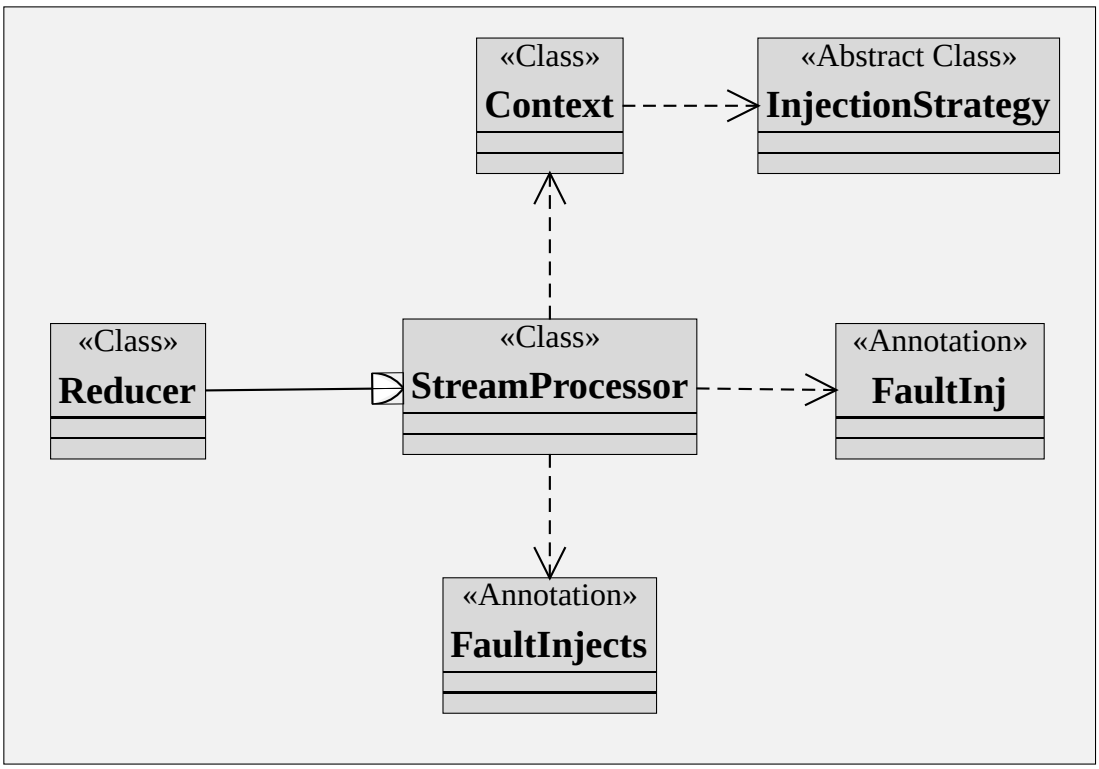



Abbildung 4.2: UML StreamProcessor

Die Injektionslogiken wurden durch eine Form des Strategy Patterns [**?** ] in die Implementierung integriert.
Die Klasse Context trifft die Auswahl der konkreten Injektionsstrategie anhand des Fehlertyps des jeweiligen Datensatzes. Eine abstrakte Klasse repräsentiert die allgemeinste Strategie. Sie kann flexibel von den beschriebenen Logiken in den folgenden Abschnitten erweitert/überschrieben werden. Die verschiedenen Strategien sind als einzelne Module definiert, die problemlos durch neue Klassen ergänzt werden können. Wenn die richtige Strategie gefunden wurde, kann die Injektion mit den entsprechenden Fehlerwerten gestartet werden.

### 4.3.2 Fehlerwerte

Fehlerwerte sind im Programmcode als Annotationen darstellt. Ein Stream kann dabei auch mit mehreren Annotationen versehen werden. Der Fehlerwert besteht immer aus den folgenden vier Komponenten:

- ID: Die ID ist ein eindeutiger Wert zur Unterscheidung bzw. Identifikation der einzelnen Streams. Soll ein Fehlerwert dynamisch verändert werden, wird dieser über die ID angesprochen. Sie wird dementsprechend in der Implementierung gesetzt und kann während der Laufzeit nicht verändert werden.

- Type: Der Fehlertyp wird über den Parameter $\mathrm{type}$ angesprochen. Er repräsentiert die Fehlerlogik die angewendet werden soll. Als mögliche Typen stehen in dieser Version der Anwendung RANDOM, LOSS, ZERO, BITFLIP, BITFLIPB, NONE zu Verfügung. Auf deren genauere Bedeutung wird in den folgenden Abschnitten der konkreten Logiken eingegangen.

- Rate: Die Fehlerrate ist eine Gleitkommazahl doppelter Genauigkeit und liegt zwischen 0 und 1. Sie gibt die Wahrscheinlichkeit der Injektion eines Blocks oder Bits an. Die Werte stehen für 0% bis 100% und fließen in die Berechnung einer Zufallsfunktion ein. Diese erstellt auf Grundlage der statischen Methode $\mathrm{random}$ aus der Klasse $\mathrm{Math}$ einen Zufallswert und setzt diesen in Relation zu der Fehlerrate. Das Ergbnis der Berechnung liefert die Entscheidung über die Injektion eines Blocks oder Bits, abhängig von der konkreten Strategie. Bei einer Fehlerrate von 1 würden demnach alle Blöcke oder Bits injiziert werden.

- Blocksize: Die Blockgröße bestimmt die Anzahl der zu injizierenden Bytes, die in einen Block gepackt werden sollen. Bis auf die einfache Bitflip Strategie sind alle Injektionsstrategien auf den Blocksize Parameter angewiesen. Die eben beschriebene Zufallsfunktion arbeitet für diese Logiken auf der Blockebene. Das heißt die Zufallsentscheidung der Injektion wird nicht für einzelne Bytes, sondern für den gesamten Block getroffen.

### 4.3.3 Injektions-Strategie

Die abstrakte Klasse $\mathrm{InjectionStrategy}$ wurde als Oberklasse für alle konkreten Strategien definiert. Sie enthält keine eigene Injektionslogik, fordert aber die Implementierung einer Injektionslogik von ihren Unterklassen. Sie stellt ebenso die Zufallsfunktion als auch die Fehlerwerte für ihre Unterklassen zur Verfügung. In Abbildung **??** wurde dieses Muster, zuzügliche der Verbindung zur Klasse $\mathrm{Context}$, modelliert.

#### 4.3.3.1 Strategie Bitflip

Die erste Strategie trägt den Namen Bitflip und ist in der Regel das aufwendigste Verfahren. Grund dafür ist die individuelle Verwendung der Zufallsfunktion für jedes einzelne Bit. Auch die Datenmanipulation muss, auf Grundlage des Resultats dieser Funktion entsprechend häufig ausgeführt werden. Wie der Name Bitflip schon verrät, werden bei dieser Strategie Bits lediglich umgedreht[2]. Im folgenden Beispiel wurden durch die Zufallsfunktion, die Bits an

[2] flip: 0 zu 1 und 1 zu 0

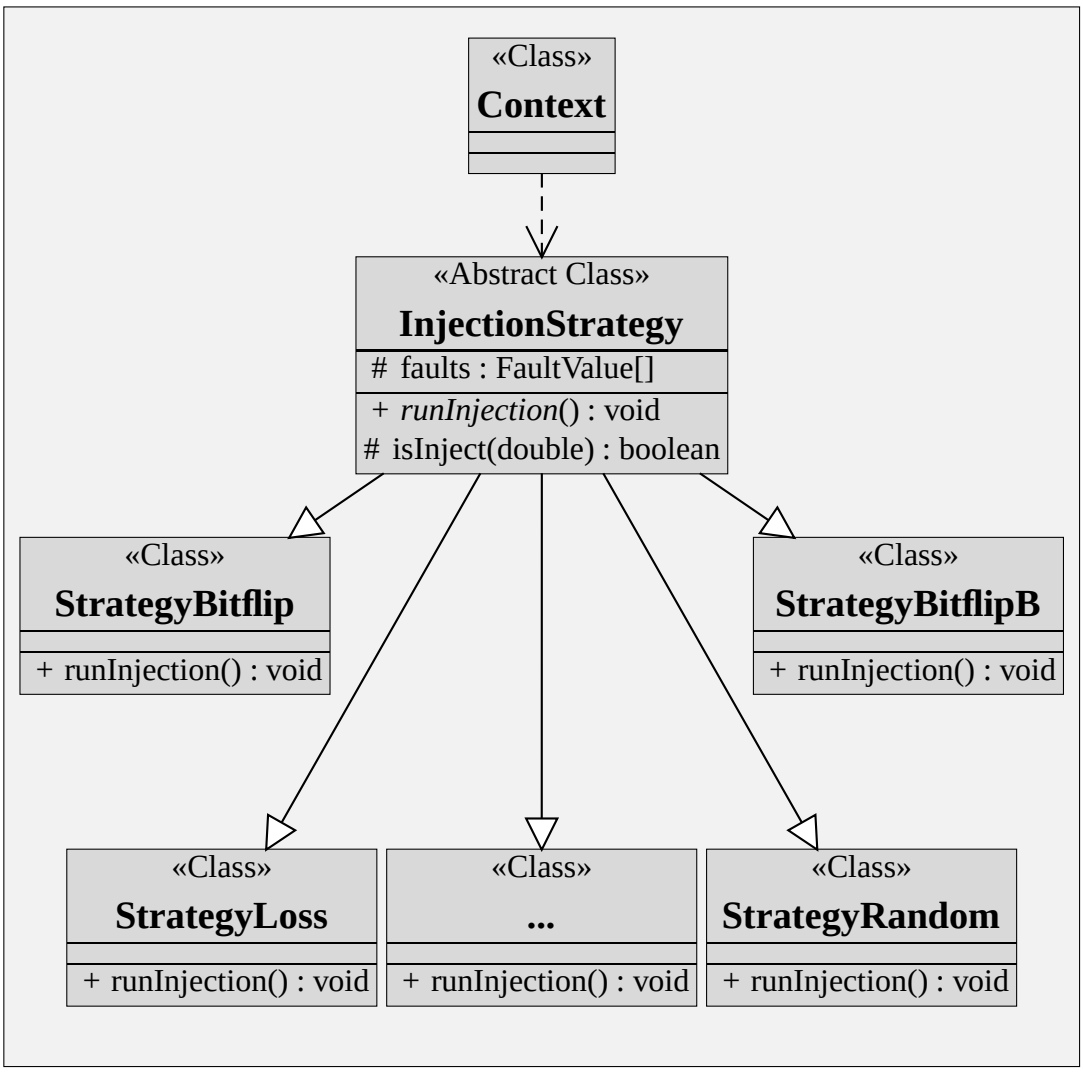
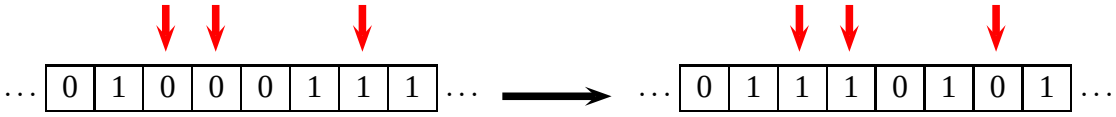



Abbildung 4.3: UML Strategy Pattern

den Positionen 3,4 und 7 ausgewählt und entsprechend injiziert.



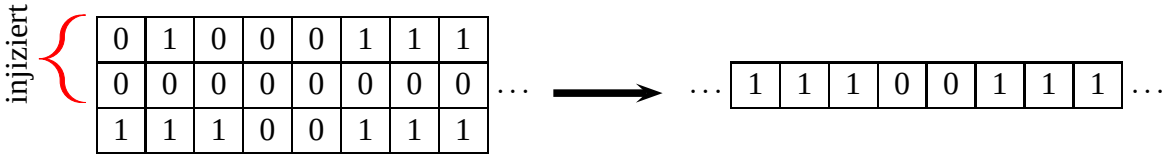

#### 4.3.3.2 Strategie BitflipB

Diese Strategie funktioniert analog zum normalen Bitflip. Einzige Ausnahmen sind die Bearbeitung ganzer Bytes statt einzelner Bits sowie die Möglichkeit der Blockverarbeitung. Die Zufallsfunktion entscheidet diesmal individuell für jeden Block über dessen Injektion. Wird ein Block zur Injektion ausgewählt, werden sämtliche Bits der darin enthaltenen Bytes gedreht.

Wird der Block nicht ausgewählt, bleiben die Bytes komplett erhalten.


#### 4.3.3.3 Strategie Loss

Loss ist eine Strategie die das Löschen von ganzen Byte-Blöcken ermöglicht. Wird ein Block im Datensatz zur Injektion ausgewählt, werden sämtliche Bytes des Blocks entfernt. Im Beispiel wird ein Block mit einer Größe von zwei aus dem Datensatz ausgewählt und beide Bytes entfernt.




#### 4.3.3.4 Strategie Random

Die Strategie Random verfügt über einen weiteren Zufallsgenerator. Dieser erstellt für die injizierten Blöcke zufällige Bytes. Die alten Bytes des jeweiligen Blocks werden im Anschluss durch die neu generierten Bytes ersetzt.


#### 4.3.3.5 StrategyZero

Diese Strategie funktioniert analog zur Random Strategie. Es werden diesmal allerdings Null-Bytes, statt zufällige Bytes für die Ersetzungen im injizierten Block verwendet. Das folgende Beispiel zeigt ein Byte das bei einer Blockgröße von eins ausgewählt und durch die Zero Strategie injiziert wurde.


#### 4.3.3.6 Strategie None

Soll keine Injektion ausgeführt werden, besteht die Möglichkeit die Fehlerrate auf 0 oder den Fehlertyp auf NONE zu setzen. Beide Varianten liefern als Resultat die Ausgangsdaten zurück. Einfaches Entfernen des Fehlertyps reicht nicht aus und würde zu einer Fehlermeldung führen.

### 4.3.4 Dynamische Konfiguration

Die Klasse AddRunTimeAnnotation ist für die dynamische Anpassung der Fehlerwerte zur Laufzeit verantwortlich. Annotationen sind ursprünglich nur Metainformationen, die im Quelltext eines Programmes notiert werden und zusätzlich semantische Informationen bereit stellen. Problematisch an der Verwendung von Annotationen ist die Tatsache, dass sich diese nicht ohne weiteres dynamisch verändern lassen. Aus diesem Grund wurden verschiedene Lösungswege getestet und schließlich eine Manipulation auf Bytecode-Ebene durchgeführt. Zu diesem Zweck wurde sich die Funktionalität der Bibliotheken Javassist und Tools zunutze gemacht, welche im Kapitel Grundlagen bzw. deren genaue Konfiguration im Abschnitt Voraussetzungen/Libraries näher erläutert werden.
Bei der Umsetzung der dynamischen Regulierung der Fehlerwerte wurde in der Klasse AddRunTimeAnnotation eine statische Methode zur Modifikation des StreamProcessor's geschrieben. Aufgerufen wird die Methode direkt über den Controller. Sie verändert bei ihrer Ausführung den Class-File des StreamProcessor's, lädt ihn in die JVM und liefert im Anschluss eine Instanz dieser Klasse an den Controller zurück. Die Beziehungen der einzelnen Klassen sind in Abbildung **??** grafisch dargestellt.

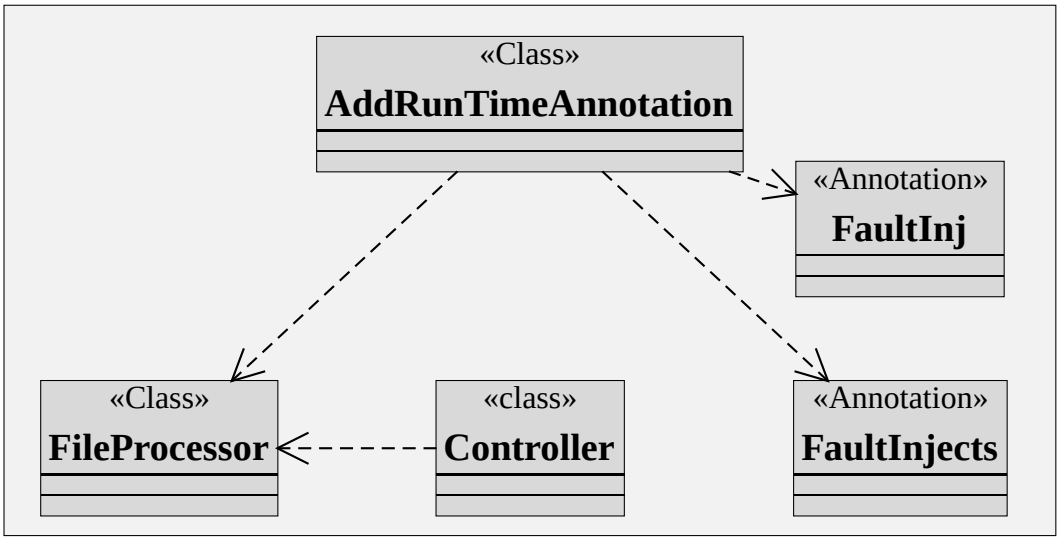



Abbildung 4.4: UML Dynamische Konfiguration

#### 4.3.4.1 Voraussetzungen/Libraries/Konfiguration

Für die Änderungen am Bytecode wurden die Bibliotheken Javassist und Tools verwendet.

1. Javassist: Durch Javassist (Java Programming Assistant) ist eine einfache Möglichkeit der Bytecode Manipulation gegeben. Diese Bibliothek erlaubt es Klassen zu verändern,

welche bereits von der JVM geladen wurden. Die Annotationen des StreamProcessor konnten dadurch mit neuen Annotationen überschrieben werden.

2. Tools: Bereits ab Java 1.4 wird die Tools Library im Ordner lib der Java Version mitgeliefert. Sie wurde im Build Path eingebunden, um die Klasse HotSwapper aus Javassist verwenden zu können. Dies war für den Reload der modifizierten Class-Datei in die JVM notwendig. Bei der Programmausführung müssen folgende VM-Argumente angegeben werden.
   - Java 1.4:
     -Xdebug -Xrunjdwp:transport=dt_ socket, server=y, suspend=n, address=8000
   - ab Java 5:
     -agentlib:jdwp=transport=dt_ socket, server=y, suspend=n, address=8000

### 4.3.5 JMX Agent

Für den Zugriff auf dem Controller und dessen Funktionalitäten wurde ein JMX Agent verwendet. Dieser bietet die Möglichkeit Komponenten zur Laufzeit zu managen, eine standardisierte Schnittstelle zum Application Server zu haben oder die Anwendungsfunktionalität leichter administrierbar zu machen.
Für die Verwendung des JMX Agent musste ein management Interface erstellt werden. Das management Interface stellt dem Client ausgewählte Funktionalitäten der gemanagten Komponente zur Verfügung. In dieser Anwendung erfüllt das Interface MBeanController aus Abbildung **??** diesen Aufgabenteil. Die durch den JMX Agent verwaltete Klasse ist der Controller, welcher das Interface MBeanController implementieren muss. Insgesamt werden in dieser Anwendung sieben Methoden des Controllers über den MBeanController bereitgestellt. In Abbildung **??** ist diese Klassenhierarchie mit den Funktionalitäten dargestellt. Für das Design des management Interface verlangte der JMX Agent besondere Namenskonventionen. Es wird grundsätzlich zwischen Attributen und Methoden unterschieden. Um ein Attribute zu setzen oder abzufragen, müssen Methoden mit dem Präfix *set-* bzw. mit dem Präfix *get-* im Controller erstellt und im Interface angegeben werden. Alle weiteren Methoden werden als normale Operationen angesehen [**?** ].

### 4.3.6 Ausnahmebehandlungen

Um mögliche Fehleingaben abzufangen, wurden drei eigene Exceptions definiert. Die Ausnahmebehandlungen werden durch folgende Punkte ausgelöst:

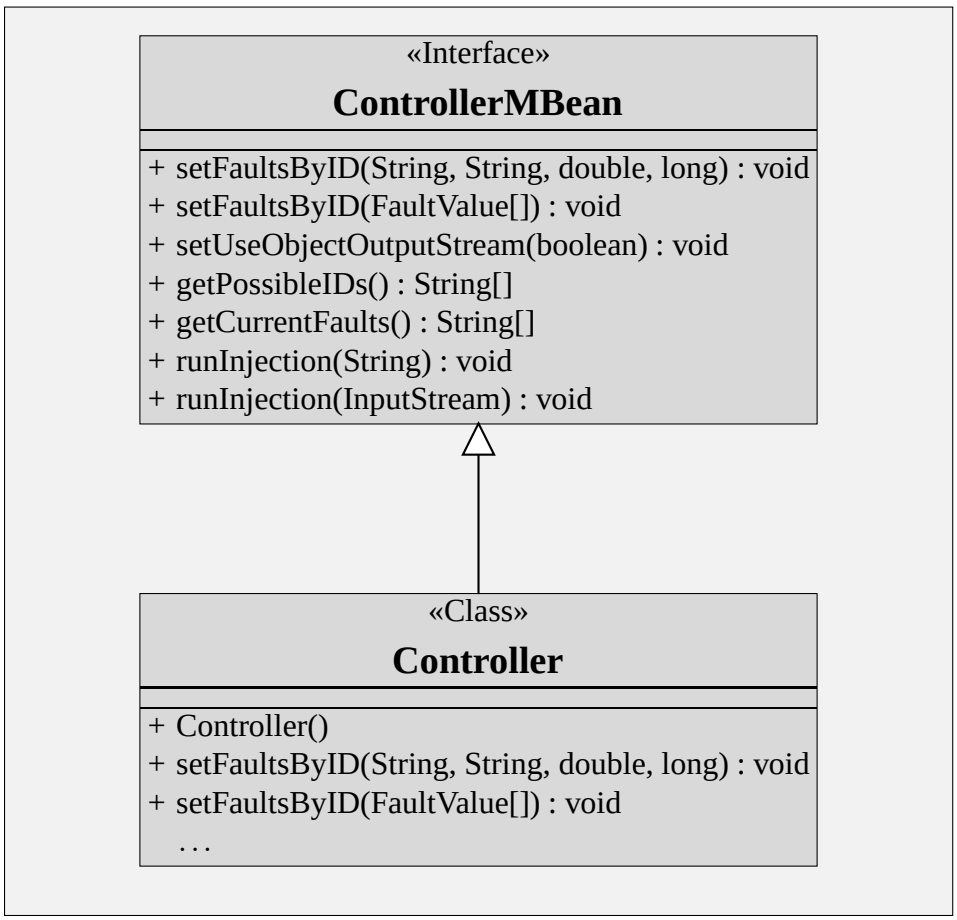



Abbildung 4.5: MBean Interface und verwaltete Controller Klasse

1. unbekannte ID : Falls die übergebene ID keinen Stream zuzuordnen ist, kann auch keine Annotation ersetzt werden. In diesem Fall wird eine NoSuchIDException geworfen

2. unbekannter Typ : Falls die gewählte Fehlerstrategie nicht existiert, wird eine NoSuchTypeException geworfen.

3. ungültige Fehlerrate : Die Fehlerrate liegt zwischen 0 und 1. Falls der übergebene Wert außerhalb dieser Grenzen liegt, kann die Berechnung nicht ausgeführt werden und eine RateOutOfBoundsException wird geworfen.

Der Controller bekommt somit die Fehlermeldung nach dem Setzen der Werte direkt von der Klasse AddRuntimeAnnotation übergeben. Der Client wird anschließend vom Controller informiert und kann entsprechend neue Werte angeben.

## 4.4 Verschiedene Clients

Um einen Client für diese Anwendung zu schreiben existieren viele verschiedene Möglichkeiten. Die folgenden vier Varianten sollen die Vielfältigkeit und Flexibilität der Implementierung zeigen.
Jedem Client dieser Anwendung werden ausschließlich die vorher festgelegten Funktionalitäten des Controllers zur Verfügung gestellt. Diese Auswahl wird über eine Schnittstelle mit dem Namen MBeanController angeboten. Um einen Zugriff auf die bereitgestellten Operationen zu erhalten, muss sich ein Client zunächst am MBean-Server anmelden. Eine Anmeldung kann lokal oder über die Remote Method Invocation geschehen. Nach der Anmeldung sind die spezifizierten Methoden über den MBeanController ausführbar.

- JConsole: Die einfachste Variante eine Verbindung aufzubauen bietet die JConsole. Dies ist ein grafisches Monitoring-Tool welches es ermöglicht, lokale und auch entfernte Anwendungen zu überwachen. Es bietet zusätzlich eine Vielzahl an nützlichen Messwerten wie Speicherverbrauch oder CPU-Auslastung.

- Shell-Script: Eine andere Möglichkeit, um beispielsweise einen konsolenbasierten Client zu schrieben, ist durch die Verwendung von jmxterm[3] gegeben. Dafür ist lediglich ein einfaches Skript für den Verbindungsaufbau zum MBeanServer und der Übergabe einer Ausführungsliste notwendig. Die gewünschten Interaktionen werden unter Berücksichtigung einer gewissen Syntax in diese Liste[4] geschrieben und dem Skript als Parameter übergeben. Ein konkretes Beispiel wird im Kapitel Implementierung aufgezeigt.

- Webbrowser: Für die Agent-View über einen Webbrowser bietet sich die Nutzung von JMXTools an. Neben dem MBeanServer muss die durch die Library bereitgestellte Klasse HtmlAdaptorServer gestartet werden. Dieser wird zusätzlich ein Host und ein Port übergeben. Mittels Webbrowser kann nun über den Host und den Port eine Verbindung zum MBeanServer aufgebaut werden. Anschließend stehen die MBeans zur Auswahl bereit. Die Operationen des gewählten MBeans lassen sich nun über den Webbrowser steuern.

- Java-Client: Java-Clients lassen sich ebenfalls über die ControllerMBean Schnittstelle sehr leicht integrieren. Die Verbindung zum MBeanServer wird über RMI oder lokal aufgebaut. Ist die Verbindung erfolgreich können Variablen über Attribut-Objekte gesetzt oder erfragt werden. Um die Methoden auszuführen ist ein Connection-Objekt erforderlich.

[3] http://wiki.cyclopsgroup.org/jmxterm
[4] einfache Textdatei

## 4.5 Sequenzieller Ablauf

Den zeitlichen Ablauf der Interaktion zwischen den Hauptklassen zeigt das Sequenzdiagramm aus Abbildung **??**. Für die Nutzung des Programms ist die Anmeldung am MBean-Server als erster Schritt obligatorisch. Dieser stellt die Schnittstelle zum Controller bereit, welcher alle erforderlichen Operationen für den Client verfügbar macht. Im Sequenzdiagramm aus Abbildung **??** ist zur Vereinfachung der Grafik die Verbindung zwischen Client und Controller direkt eingezeichnet worden.
Nach der erfolgreichen Anmeldung am Server steht es dem Client frei welche Operation er ausführen möchte. Auf der Zeitlinie aus Abbildung **??** folgt als erster Schritt eine Fehlerwertabfrage, um anhand der ID's neue Werte setzen zu können.
Im Anschluss wird die Default Fehlerinjektion vom Client gestartet. Der Controller gibt den Aufruf zunächst an den StreamProcessor weiter. Der StreamProcessor liest die Daten ein und übergibt sie als Byteliste an ein Context-Objekt. Die Klasse Context wählt anhand der Fehlerwerte, in diesem Fall die Default Fehlerwerte im Quellcode, die erforderliche Strategie aus. Die Bytes werden mit Fehlern injiziert und zurück an den StreamProcessor geschickt. Der Injektionsaufruf des Controllers respektive des Clients wird nun vom StreamProcessor durch das Schreiben der Daten in eine vorher angelegte Ausgabedatei zum Abschluss gebracht.
Eine Fehlerinjektion mit Streamwechsel ist in Abbildung **??** unter "RunInjection mit Streamwechsel" dargestellt. Zuerst wird ein anderer Ausgabestream gewählt. Diese Operation kann zeitlich auch später folgen, muss jedoch vor dem Injektionsaufruf ausgeführt werden. Danach wurden die Fehlerwerte neu gesetzt. Das Setzen der Fehlerwerte kann zu verschiedenen Ausnahmebehandlungen führen. Im Diagramm wurde eine falsche ID übergeben. Der Controller bekommt daraufhin bei der Modifikation der Annotationen eine Fehlermeldung übergeben, welche an den Client weitergereicht wird. Im Falle einer korrekten Übergabe der Fehlerwerte erhält der Controller einen modifizierten StreamProcessor. Anschließend kann die Injektion vom Client veranlasst werden. Der Controller ruft an dieser Stelle die entsprechende Methode des modifizierten StreamProcessor auf. Die weitere Verarbeitung geschieht ananlog zur Default Injektion.

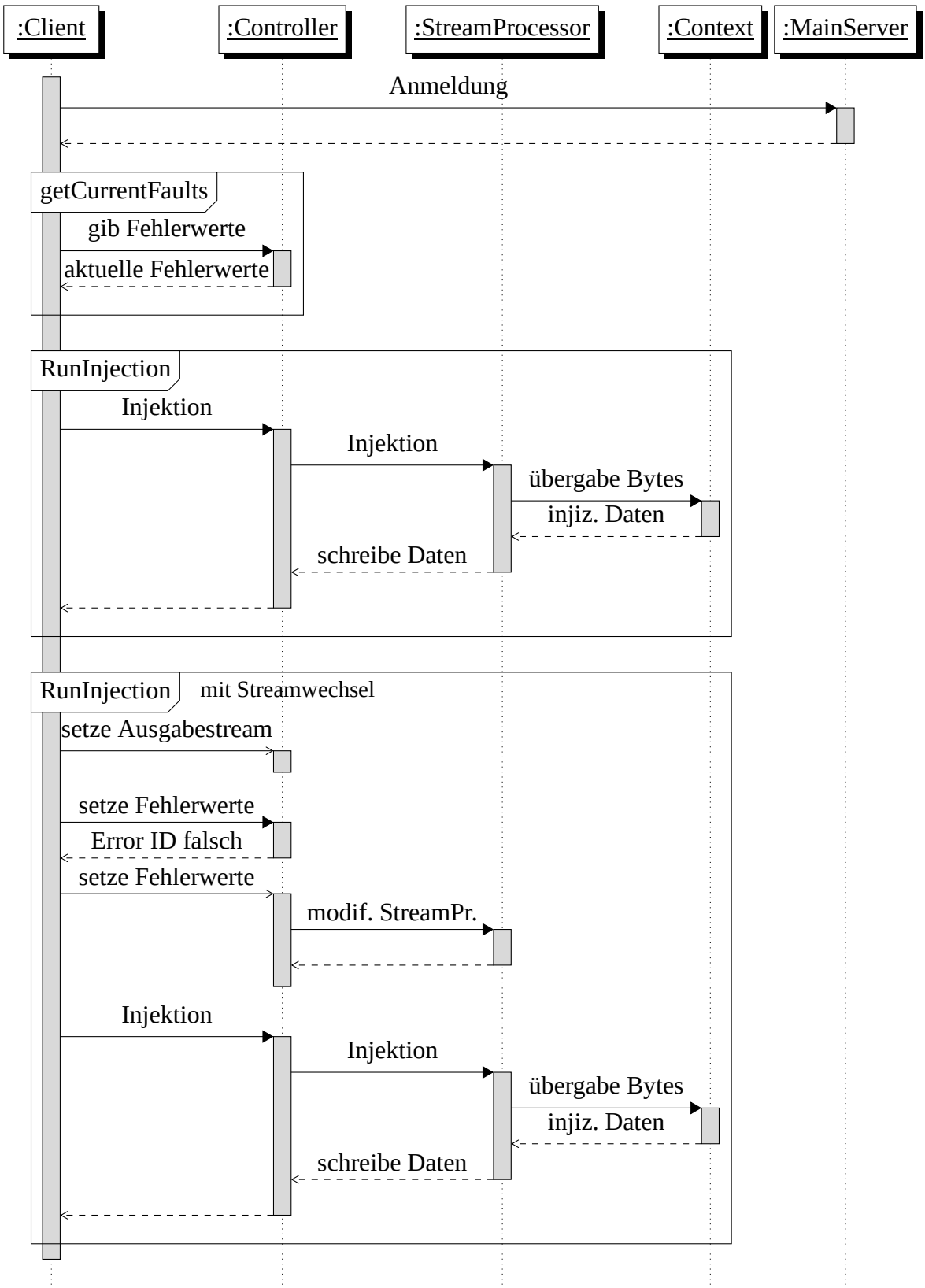



Abbildung 4.6: Sequenzdiagramm

Kapitel 5

# 5 Implementierung

Das folgende Kapitel zeigt die Implementierungsdetails der im Design-Kapitel erläuterten Komponenten. Es werden hierfür die wichtigsten Funktionen beschrieben und zum besseren Verständnis der Code anhand von kurzen Ausschnitten aufgezeigt.
Die einzelnen Klassen wurden nach ihrem Aufgabengebiet in verschiedene Java-Pakete eingeteilt. Zunächst wird das Paket faults betrachtet. Dessen Inhalte sind alle Klassen die Fehlerwerte repräsentieren und daher von den folgenden Klassen verwendet werden.
Das Paket processor bildet den Hauptteil des Kapites. Es beinhaltet die Klassen Context, StreamProcessor, also auch die Klasse AddRuntimeAnnotation. Alle drei Klassen sind eng aneinander gekoppelt und für die Hauptfunktionalität der Anwendung zuständig.
Die einzelnen Injektions-Logiken wurden in das Paket strategies gepackt. In diesem Abschitt steht das verwendete Pattern, welches für die Implementierung der Injektion verwendet wurde, im Fokus. Die vorhandenen Strategien werden nur kurz, anhand einer ausgewählten Strategie exemplarisch vorgestellt.
Die Implementierungen des Controllers und des MainServers sind Bestandteil des Paketes jmx. In diesem Abschnitt wird die verwendete JMX-Technologie detailliert beschrieben und anschließend dessen Verwendung anhand verschiedener JMX-Client Implementierungen aufgezeigt.

## 5.1 Fehlerwerte

Die Fehlerwerte der zu injizierenden Streams sind im Paket faults erstellt worden. Dieses Paket besteht aus einer Klasse, zwei Annotationen und einer Enumeration. In der weiteren Implementierung werden diese Konstrukte an vielen Stellen verwendet und stehen insbesondere bei der dynamischen Anpassung der Fehlerwerte im Fokus.

1. FaultInj : Die Annotation FaultInj wurde eigens definiert, um die zu injizierenden Streams als fehlerbehaftet kennzeichnen zu können. Sie besteht aus den Werten id, type, rate und blocksize. In Listing **??** ist zu erkennen, dass alle Variablen bis auf die id einen Default-Wert besitzen. Eine Fehlermarkierung kann somit wie in Listing **??** vorgenommen werden. In diesem Fall ist zwar eine Markierung vorhanden, es werden aber vorerst keine Fehlerwerte eingestreut. Aufgrund der Eindeutigkeit der ID besitzt sie keinen Default-Wert. Der Entwickler wird somit explizit durch den Compiler zur Angabe einer ID gezwungen.

```
1    @Documented
2    @Target(java.lang.annotation.ElementType.FIELD)
3    @Retention(java.lang.annotation.RetentionPolicy.RUNTIME )
4
5    public @interface FaultInj {
6        String id();
7        String type() default "NONE";
8        double rate() default 0.0;
9        long blocksize() default 1024;
10   }
```

Listing 5.1: Annotation FaultInj

```
1    @FaultInj(id = "id01")
2    private InputStream stream;
```

Listing 5.2: Annotation Default-Werte

2. FaultInjects : Für Streams, die mehr als einen Fehlerwert erhalten sollen, wurde eine weitere Annotation definiert. Da mehrere Annotationen für eine Variable nicht zulässig sind, wurde dieses Verhalten über einen Umweg erzielt. Die FaultInject-Annotation aus Listing **??** besitzt einen Array vom Typ der ursprünglichen FaultInj-Annotation. Der Stream wird nun einfach mit der FaultInject-Annotation markiert und die gewünschten FaultInj-Annotationen dem Array übergeben.

```
1    @Documented
2    @Target(java.lang.annotation.ElementType.FIELD)
3    @Retention(java.lang.annotation.RetentionPolicy.RUNTIME )
4
5    public @interface FaultInjects {
6        FaultInj[] value();
7    }
```

Listing 5.3: Annotation FaultInjects

3. FaultType : Für die angebotenen Fehlertypen wurde ein Enum mit dem Namen FaultType erstellt (Listing **??**). Die Fehlertypen beziehen sich, wie im Design Kapitel erläutert, auf die Injektions-Strategie. Soll die Anwendung um eine Fehlerstrategie erweitert werden, kann diese über das Enum FaultType dem Client zugänglich gemacht werden. Ausschließlich hier definierte Typen werden von der Anwendung akzeptiert.

```
1     public enum FaultType {
2         RANDOM, LOSS, ZERO, BITFLIP, BITFLIPB, NONE
3     }
```

Listing 5.4: Enum FaultType

4. FaultValue: Die Klasse FaultValue dient als Wrapper-Klasse[1] für die Parameter einer Annotation. Die Klasse erleichtert somit die Datenübergabe an andere Klassen sowie das Auslesen der jeweiligen Fehlerparameter.

## 5.2 Klasse StreamProcessor

Der StreamProcessor ist das Zentrum der Anwendung. Über diese Klasse werden zu Beginn die Daten aus der Datei oder dem Stream gelesen. Die Daten werden anschließend als einzelne Bytes an die Klasse Context übergeben. Dieses Vorgehen ist durch Listing **??** umgesetzt worden. Für das Einlesen der Daten wurde ein InputStream verwendet. Dieser Stream

```
1     private void loadStream(InputStream src, Context<Byte> context) {
2
3         stream = src;
4         context.write(0L,readStream(stream));
5
6         setFaultsToContext(context);
7         //setFaultsByIDToContext(context,
8         //  new String[]{"OutputBAOS","OutputBAOS2"},0L);
9     }
10
11    private List<Byte> readStream(InputStream src) {
12        List<Byte> list = new ArrayList<Byte>();
13        int i;
14        while((i=src.read())!=-1)
15            list.add((byte)i);
16        return list;
17    }
```

Listing 5.5: StreamProcessor Einlesen der Daten

[1] Aufname von Daten in ein Objekt

wird von FileIO, Sockets, etc. unterstützt. Der fehlerbehaftete Stream muss global deklariert sein, damit er annotiert/injiziert werden kann. Die beiden Möglichkeiten der Annotierung sind durch Listing **??** gegeben.
Im nächsten Schritt wird der InputStream durch die Methode readStream in eine Byte-Liste ausgelesen. Aufgerufen wurde readStream durch die Methode loadStream, welche die erstellte Ergebnisliste bzw. den Datensatz mit einer dazugehörigen ID an das Context Objekt übergibt. Im zweiten Schritt werden die Fehlerwerte durch die Methode setFaultsByIDToContext unter Verwendung von Java Reflection [2] an das Contex Objekt übergeben. Die Methode benötigt dafür die Fehler-ID's und die ID des betroffenen Datensatzes. Das Laden der Dat-

```
1     @FaultInj(id="OutputTEST", type="LOSS", rate=1, blocksize=1024)
2     private InputStream test;
3
4     // ================================================
5
6     @FaultInjects({
7         @FaultInj(id="OutputBAOS", type="ZERO", rate=0.01, blocksize=8),
8         @FaultInj(id="OutputBAOS2", type="LOSS", rate=0.001, blocksize=8)
9     })
10    private InputStream fis;
11
12    ...
```

Listing 5.6: StreamProcessor Fehlermarkierung

en wird beim Aufruf der Methode processData aus Listing **??** veranlasst. Nach dem Laden folgt die Methode injectFaults der Klasse Context. Diese sorgt intern dafür, dass die verwalteten Daten mit den beigefügten Fehlerwerten injiziert werden.
Für das Auslesen der manipulierten Daten aus dem Context Objekt, sowie das Schreiben in eine neue Datei wurde die innere Klasse Reducer implementiert. Der Reducer erstellt zunächst den Stream für die Ausgabe und bereitet eine neue Datei vor. Danach wird die Methode reduce aufgerufen um durch den jeweiligen Datensatz zu iterieren und die Bytes durch den gewählten Stream in die neue Datei zu schreiben. Das Auslesen der FaultValues ist durch die Implementierung aus Listing **??** realisiert worden. Diese Methode wird ebenfalls im Controller/MBeanController verwendet, um auf Anfrage des Clients die Fehlerwerte liefern. Die Funktion arbeitet mit der bereits erwähnten Java Reflection Technologie um die injizierten Felder zu ermitteln. Es werden alle globalen Variablen durchgegangen und überprüft, ob die FaultInjects Annotation präsent ist. Ist dies der Fall, werden alle FaultInj Annotationen heraus gefiltert und deren Fehlerwerte als FaultValue verpackt zu einer Liste beigefügt. Für den Fall, dass statt einer FaultInjects Annotation nur eine einzelne FaultInj Annotation vorhan-

[2] http://docs.oracle.com/javase/tutorial/reflect/index.html - Zugriff am 21. Juli 2012

```
1   public void processData(String filename) {
2       startprocessData(new FileInputStream(filename),filename);
3   }
4
5   public void processData(InputStream src) {
6       startprocessData(src,"");
7   }
8
9   private void startprocessData(InputStream src ,String filename) ...
10      Context<Byte> ctx = new Context<Byte>();
11
12      loadStream(src, ctx);
13
14      ctx.injectFaults();
15
16      Reducer reducer = new Reducer();
17      for (long i = 0; i < ctx.getEntries().size(); i++) {
18          reducer.setFileName(filename);
19          reducer.reduce(ctx.getEntries().get(i));
20      }
21  }
```

Listing 5.7: StreamProcessor Einlesen/Injektion/Ausgabe

den ist, wird im else-Zweig die Verarbeitung analog für die einzelne FaultInj durchgeführt.

```
1   public String[] getAnnotatedValues(){
2       ...
3       for (Field f : fields) {
4           if (f.isAnnotationPresent(FaultInjects.class)) {
5               ...
6               for(FaultInj fi : fis){
7                   list.add(new FaultValue(fi.id(), fi.type(), fi.rate(),
                        fi.blocksize()).toString());
8               }
9           }
10          else{
11              if (f.isAnnotationPresent(FaultInj.class)) {
12                  ...
13              }
14      ...
```

Listing 5.8: StreamProcessor Fehlerwertrückgabe

## 5.3 Klasse AddRuntimeAnnotation

Für die dynamische Konfiguration der Fehlerwerte wurde die Klasse AddRuntimeAnnotation geschrieben. Sie enthält eine statische Methode addFaultInjAnnotationToMethod die für die Umsetztung der Modifikation zuständig ist. Bei ihrer Ausführung verändert sie die Class Datei des StreamProcessors und liefert als Resultat eine Instanz des modifizierten Objektes zurück. Die Methode wurde so konzipiert, dass mehrere Annotationen mit einem Aufruf geändert werden können. Für die Modifikation nutzt die Methode verschiedene Komponenten aus der Library javassist.
Eine in der Implementierung aus Listing **??** benutzte Komponente ist die Klasse Javassist.CtClass[3], welche die Class Datei des StreamProcessors repräsentiert. Um eine solche CtClass erstellen und später modifizieren zu können wurde vorher ein ClassPool Objekt erstellt. Der ClassPool liest mittels des Schlüsselworts get die gewünschte Class Datei und konstruiert aus dieser eine CtClass. Die CtClass kann nun auf Bytecode-Ebene angepasst werden. Um den Bytecode des StreamProcessors updaten zu können, wurde ein Objekt vom Typ HotSwapper erstellt. Dieses benötigt lediglich einen Port für das Laden der neuen Class Datei[4]. Die Umsetzung der Erstellung dieser Komponenten wird in Listing **??** gezeigt.

```
1     // pool creation
2     private static ClassPool pool = ClassPool.getDefault();
3     private static CtClass cc;
4     ...
5     // reload classfile
6     private static HotSwapper hs;
7
8     public static void initHotSwapper(String port) ...{
9         hs = new HotSwapper(port);
10    }
11    ...
12    private static StreamProcessor addFaultInjAnnotationToMethod(
13            String className, FaultValue[] faults) {
14        ...
15        // extracting the class
16        cc = pool.getCtClass(className);
17        ...
18    }
```

Listing 5.9: Initialisierung HotSwapper & CtClass

Als Parameter bekommt die Methode addFaultInjAnnotationToMethod die neuen Fehlerwerte anhand eines FaultValue Array übergeben. Mittels der ID des jeweiligen FaultValues kann, wie in Listing **??** zu sehen, das dazugehörige Feld in der Klasse StreamProcessor bes-

[3] compile-time class
[4] inter-thread communication

timmt werden. Die ermittelten Felder bekommen für den Manipulationsvorgang eine Instanz der Klasse Javassist.CtField zugewiesen.

```
1     // looking for the field to apply the annotation on
2     CtField injectFieldDescriptor[] = new CtField[faults.length];
3
4     for (int i = 0; i < faults.length; i++)
5         ...
6         injectFieldDescriptor[i] = cc
7             .getDeclaredField(getFieldName(faults[i].getId()));
8         ...
```

Listing 5.10: Bestimmung injizierter Felder

Um neue Felder, Methoden, Annotationen etc. hinzufügen zu können wurde ein ClassFile Objekt auf den CtClass Class-File referenziert. Zusätzlich wurde ein ConstPool[5] Objekt mit Verweis auf diesen Class-File erstellt (Listing **??**). Für die neuen Annotationen wurden zwei javassist.Annotation Objekte definiert. Eines für die FaultInjects und das andere Objekt für die FaultInj Annotationen. Ein AnnotationsAttribute Objekt ist nach der Erstellung der Annotationen für die Übergabe an das entsprechende ctField notwendig.

```
1     // create the annotation
2     ClassFile ccFile = cc.getClassFile();
3     ConstPool constpool = ccFile.getConstPool();
4
5     AnnotationsAttribute[] attr = new AnnotationsAttribute[faults.length];
6     Annotation[] annotations = new Annotation[faults.length];
7     Annotation[] subAnnotations;
```

Listing 5.11: Neue Annotationen

Für die Erstellung der neuen Annotationen wurde zunächst durch die vorher bestimmten CtFelder iteriert. Es wurde die alte Annotation bestimmt und eine neue Annotation dieses Typs (FaultInjects oder FaultInj) angelegt (Listing **??**). Die erstellten Annotationen wurden gleichzeitig zur Speicherung, der constant-pool-table beigefügt.
Im if-Zweig, also falls ein Feld mit einer FaultInjects-Annotation präsent ist, wird ein AnnotationMemberValue Array für die Unterannotationen verwendet. Es werden zunächst alle Unterannotationen durchlaufen und nach der passenden ID gesucht. Falls die ID gefunden wurde, können die neuen Fehlerwerte aus dem FaultValue Array geschrieben werden. Für alle anderen Unterannotationen wurden die alten Werte erneut verwendet. Nach jedem Durchlauf wird die erstellte Unterannotation dem AnnotationMemberValue Array übergeben. Dieser

[5] constant pool table

```
1  for (int i = 0; i < faults.length; i++) {
2
3    if (injectFieldDescriptor[i].hasAnnotation(FaultInjects.class)) {
4        annotations[i] = new Annotation(
5                "faults.FaultInjects", constpool);
6        ...
7    } else {
8        annotations[i] = new Annotation("faults.FaultInj",
9                constpool);
10       ...
11   }
12 }
```

Listing 5.12: Zuweisung des Annotationstyps

repräsentiert die FaultInjects-Annotation und kann im Anschluss komplett an das Annotation Array übergeben werden (Listing **??**).
Im else-Zweig, falls also ein Feld mit einer FaultInj-Annotation präsent ist, werden lediglich die neuen Werte in die Annotation aufgenommen. Da im anfangs erstellten CtField Array nur Felder enthalten sein können die geändert werden sollen, muss hier auch keine weitere Überprüfung durchgeführt werden.
Zum Schluss kommt das AnnotationsAttribute Objekt zum Einsatz. Jede Annotation wird diesem Objekt übergeben und die Sichtbarkeit zur Laufzeit gesetzt. Dann kann dieses dem entsprechenden CtField übergeben werden.
Nachdem die Klasse mit den neuen Annotationen versehen wurde, muss der Bytecode neu geladen werden. Hierfür wird das CtClass Objekt in eine Class Datei transformiert und anschließend der Bytecode ausgelesen. Die Klasse javassist.HotSwapper lädt nun anstelle des ursprünglichen, den modifizierten Bytecode erneut in die JVM (Listing **??**).

## 5.4 Context

Die Klasse Context ist für die Datenverwaltung/-manipulation verantwortlich. In Listing **??** ist die Methode write zu sehen die vom StreamProcessor zur Übergabe der Datensätze aufgerufen wird. Als Prämisse dafür gilt die Serialisierbarkeit der Daten. Zur Erfassung wurde eine HashMap gewählt, über die eine einfache Verwaltung der Datensätze anhand der ID möglich ist.
Um die Daten mit Fehlern zu injizieren, wurde die Methode getStrategy aus Listing **??** erstellt. Die Methode bekommt einen Datensatz mit dem dazugehörigen Fehlerwert als Parameter übergeben. Anhand des Fehlertyps wird die zugrundeliegende Strategie ausgewählt. Neue Strategien müssen in diese Methode integriert werden.

```
1    AnnotationMemberValue[] elements = new AnnotationMemberValue[len];
2
3    for (int x = 0; x < len; x++) {
4
5        if (fis[x].id().equals(faults[i].getId())) {
6            subAnnotations[x] = new Annotation(
7                    "faults.FaultInj", constpool);
8            subAnnotations[x].addMemberValue(
9                    "id",
10                   new StringMemberValue(faults[i].getId(), ccFile
11                       .getConstPool()));
12           ...
13
14       } else {
15           subAnnotations[x] = new Annotation(
16                   "faults.FaultInj", constpool);
17           subAnnotations[x].addMemberValue(
18                   "id",
19                   new StringMemberValue(fis[x].id(), ccFile
20                           .getConstPool()));
21           ...
22       }
23       elements[x] = new AnnotationMemberValue(subAnnotations[x],
24               ccFile.getConstPool());
25   }
26
27   ArrayMemberValue arrMemberValue = new ArrayMemberValue(
28           ccFile.getConstPool());
29   arrMemberValue.setValue(elements);
30   annotations[i].addMemberValue("value", arrMemberValue);
```

Listing 5.13: Setzen der Fehlerwerte

```
1  // transform the ctClass to java class
2  dynamiqueBeanClass = Class.forName(className);
3  classFile = cc.toBytecode();
4  hs.reload(className, classFile);
5
6  // instanciating the updated class
7  imgP = (StreamProcessor) dynamiqueBeanClass.newInstance();
```

Listing 5.14: Laden des Bytecodes

Die gewählte Strategie kann nun in der Methode injectFaults für die Dateninjektion verwendet werden (Listing **??**). Die Variable registeredFaultInjectors ist eine Liste, bestehend aus Tupeln von IDs[6] und dazugehörigen FaultValues. Im Gegensatz zu den Datensätzen der HashMap sind bei dieser Liste die IDs nicht eindeutig, da jeder Datensatz mehrere FaultVal-

[6] Referenz auf Datensätze

```
1   public class Context<E extends Serializable> {
2   ...
3       public void write(Long key, List<E> data)  {
4
5           if (outHashtable.containsKey(key)) {
6               outHashtable.get(key).addAll(data);
7           } else {
8               outHashtable.put(key, data);
9           }
10          return;
11  }
12  ...
```

Listing 5.15: Context: Einlesen der Datensätze

```
1   InjectionStrategy getStrategy(List data, FaultValue fault) ....{
2       if(fault.getType().equalsIgnoreCase(FaultType.ZERO.name()))
3           return new StrategyZero(data,fault);
4       else if(fault.getType().equalsIgnoreCase(FaultType.RANDOM.name()))
5           return new StrategyRandom(data, fault);
6       else if(fault.getType().equalsIgnoreCase(FaultType.LOSS.name()))
7           return new StrategyLoss(data, fault);
8       ...
9   }
```

Listing 5.16: Context: Strategieauswahl

ues haben kann. Die gewählte Strategie bekommt diese Werte übergeben und ruft ihrerseits die Methode runInjection auf.

```
1   ...
2   for(Tupel<Long,FaultValue> tplInj : registeredFaultInjectors) {
3       if(tplInj.getFst()<len){
4           outHashtable.put(tplInj.getFst(), (List<E>)
5           this.getStrategy(outHashtable.get(tplInj.getFst()), tplInj.
                getSnd()).runInjection());
6       }
7   }
8   ...
```

Listing 5.17: Context: Injektionsausführung

## 5.5 Fehler Injektion

Die Implementierung der Fehlerinjektionen wurde mittels einer abgewandelten Form des Strategy Patterns realisiert. Die abstrakte Klasse InjectionStrategy dient als Oberklasse aller konkreten

Strategien. Sie stellt für ihre Unterklassen die beiden Funktionen aus Listing **??** bereit. Die Methode isInjected ist für jede Strategie gleich und wird daher bereits mit Funktionalität an die Unterklassen vererbt. isInjected ist für den Funken Zufall in den einzelnen Injektionen verantwortlich. Anhand der Fehlerrate, die zwischen 0 und 1 liegen muss und einem Zufallswert der ebenfalls zwischen 0 und <1 liegt, wird entschieden ob ein Bit bzw. Block injiziert werden soll.
Die zweite Methode ist mit dem Schlüsselwort abstract deklariert. Die Implementierung der eigentlichen Algorithmen finden sich erst in der Methode runInjection der jeweiligen Unterklassen wieder.
Verwendet wird dieses Muster in der bereits beschriebenen Klasse Context durch die Methode getInjectionStrategy. Diese Methode erzeugt eine Member-Variable aus der abstrakten Klasse InjectionStrategy, die mit einer Referenz auf die gewünschte Logik belegt ist. Somit kann eine konkrete Strategie beliebig eingebunden als auch ausgetauscht werden. Die Implementierung ist dadurch sehr leicht erweiterbar. Insgesamt benötigen neue Strategien folgende Anpassungen/Erweiterungen:

- einen eigenen Typ im Enum FaultType
- müssen von der aktrakten Klasse InjectionStrategy erben und dessen abstrakte Methode runInjection implementieren.
- müssen in die Methode getStrategy der Klasse Context integriert werden

```
1     ...
2     public abstract List<Byte> runInjection();
3
4     protected static boolean isInjected (double rate) ...
5         if(rate < 0 || rate > 1){
6             ...
7             throw new RateOutOfBoundsException(msg);
8         }
9         return Math.random() < rate;
10    }
```

Listing 5.18: InjectionStrategy

## 5.6 JMX Implementierung

### 5.6.1 Controller/MainServer

Als Schnittstelle zwischen Benutzer und Anwendung wurde die Klasse Controller eingeführt. Sie wird vom Anwender über den sogenannten MBean-Server angesprochen. Die Klasse Con-

troller ist in der Anwendung für die Steuerung der Fehlerinjektion und weiteren Funktionalitäten verantwortlich. Der Client bekommt die für ihn vorgesehene Funktionalität durch das Interface ControllerMBean aus Listing **??**. Das Interface stellt alle nach außen sichtbaren Methoden bereit und muss vom Controller implementiert werden. Der Controller ist somit über das Interface ansprechbar, alle weiteren Implementierungsdetails bleiben nach außen verborgen.

```
1    public interface ControllerMBean {
2
3        public void setFilename(String filename) throws ...;
4        public void setFaultsByID(...) throws ...;
5        ...
6        public void runInjection() throws ...
7    }
```

Listing 5.19: ControllerMBean

Die Implementierung des MainServer ist in Listing **??** dargestellt. Um die Java Management Extensions nutzen zu können, muss zunächst ein MBeanServer Objekt durch den Aufruf der Methode getPlatformMBeanServer der Klasse java.lang.management.ManagementFactory erstellt werden. Danach wird ein ObjectName für das zu erstellende Controller MBean definiert. Dieses besteht aus einer Domain und einer Liste von key-properties. Die Domain ist das package in dem sich das zu erstellende MBean befindet. In Listing **??** ist dies das package jmx. Die key-property spezifiziert den Typen des Objektes, in diesem Fall der Typ Controller. Im Anschluss wird eine Objektinstanz des Controllers erzeugt. Dieses kann nun als MBean mit dem ObjectName Objekt am MBeanServer registriert werden. Nach dem Starten der Klasse MainServer wird auf die Aufrufe der im ControllerMBean definierten Methoden gewartet, welche dann über den Controller ausgeführt werden.
Für die Möglichkeit einer zusätzlichen RMI Connection wurde die Implementierung aus Listing **??** verwendet. Falls keine Parameter mitgegeben wurden, wird die Klasse Mainserver wie bisher gestartet. Andernfalls existiert die Klasse JMXServiceURL aus javax.management.remote, der sich Hostname und Port als Parameter übergeben lassen. Die somit erstellte URL wird mit dem MBean Server über einen JMXConnectorServer verknüpft.

### 5.6.2 Client Implementierung

JMX-Clients gibt es bereits in vielen Variationen. Sehr beliebt sind die JConsole, Java-, Shell- oder HTML-Clients[7]. Im Folgenden werden ein Shell-Skript und eine Java Client Implementierung vorgestellt.

[7] http://www.servletsuite.com/jmx/jmx-html.htm - Zugriff am 09. September 2012

```
    // Get the Platform MBean Server
    MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();

    // Construct the ObjectName for the Controller MBean we will register
    ObjectName mbeanName = new ObjectName("jmx:type=Controller");

    // Create the Controller MBean
    Controller mbean = new Controller();

    // Register the Controller MBean
    mbs.registerMBean(mbean, mbeanName);
    ...
    // Wait forever
    System.out.println("Server is running now");
    Thread.sleep(Long.MAX_VALUE);
}
```

Listing 5.20: MainServer

```
    if (args.length >= 2) {
        LocateRegistry.createRegistry(Integer.parseInt(args[1]));

        JMXServiceURL url = new JMXServiceURL(
                "service:jmx:rmi:///jndi/rmi://" + args[0] + ":" + args[1]
                        + "/server");

        JMXConnectorServer cs = JMXConnectorServerFactory
                .newJMXConnectorServer(url, null, mbs);
        ...
        cs.start();
    }
```

Listing 5.21: MainServer RMI

#### 5.6.2.1 Shell-Skript

Für das Bash-Script wurde die jmxterm-1.0 library verwendet. Diese bietet eine intuitive Handhabung und ermöglicht einen einfachen Verbindungsaufbau. Der implementierte Client besteht aus einer bash-Datei für den Verbindungsaufbau und dem Starten der jmxterm.jar mit einer Ausführungsliste. Für eine RMI-Connection genügt der Aufruf der jmxterm library mit der entsprechenden Service URL. Die lokale Verbindung aus Listing **??** muss zunächst den erforderlichen MainServer Process[8] bestimmen und diesen anschließend mit dem Schlüsselwort open der Ausführungsliste beifügen. Der open Befehl öffnet entweder eine neue JMX-Session oder gibt die aktuelle Verbindung wieder. Eine Ausführungsliste ist eine zusätzliche

[8] jps: Java Virtual Machine Process Status Tool

Text-Datei, die Anweisungen für den direkten Verbindungsaufbau zum MainServer und alle gewünschten Methodenaufrufe erfasst.

```
#!/bin/bash
if [ $# -ge 2 ]; then
    # rmi connection
    java -jar jmxterm-1.0-alpha-4-uber.jar -l service:jmx:rmi:/// ...
else
    echo "for rmi type : ./run.sh <host> <port>"
    p=$(jps | grep MainServer | cut -d ' ' -f 1)

    if [ -z "$p" ]
    then
        exit 0
    fi
    ...
    sed -i '1i open '"$p" script.txt
    java -jar jmxterm-1.0-alpha-4-uber.jar -n -i script.txt -o output.txt
    ...
fi
```

```
domain de.umr.emu.jmx
bean de.umr.emu.jmx:type=Controller
get Annotations
run runInjection ""
get PossibleIDs
run setFaultsByID "OutputBAOS" "ZERO" 0.05 1024
get Annotations
run runInjection "/home/.../Tyson.jpg"
quit
```

Listing 5.22: Bash-File und Ausführungsliste

#### 5.6.2.2 Java Client

Die Erstellung eines Java Clients verläuft relativ analog wie das eben beschriebene Beispiel. Zunächst wird wieder die JMX Verbindung per RMI oder Lokal aufgebaut. Danach wird ein MBeanServerConnection Objekt erstellt über dessen alle Methodenaufrufe ausgeführt werden. Sollen beispielsweise ein FaultValue gesetzt werden, wird die Methode setFaultValue in ein Attribute Objekt verpackt und anschließend der MBeanServerConnection mittels dessen Methode setAttribute übergeben. Getter-Methoden wie getCurrentFaults können analog mit getAttribute ausgewertet werden. Handelt es sich bei einer Methode nicht um ein Attribute[9]

[9] getter-/setter-Methoden mit nur einem Parameter

wird die Methode invoke der MBeanServerConnection verwendet. Diese bekommt als Parameter das ObjectName Objekt, den Methodennamen und dessen Parameterwerte, als auch die Methodensignatur.

```
    ...
    // Get an MBeanServerConnection
    MBeanServerConnection mbsc = jmxc.getMBeanServerConnection();
    ObjectName mbeanName = new ObjectName("jmx:type=Controller");

    Attribute att = new Attribute("UseObjectOutputStream", true);

    // Invoke the runInjection op
    String[] sig = {"java.lang.String"};
    Object[] opArgs = {"/home/../Tyson.jpg"};

    mbsc.setAttribute(mbeanName, att);
    mbsc.setAttribute(mbeanName, re);
    ... mbsc.getAttribute(mbeanName, "CurrentFaults");
    ...

    mbsc.invoke(mbeanName, "runInjection", opArgs, sig);
```

Listing 5.23: Java Client

Kapitel 6

# 6 Evaluation/Messungen

Im Kapitel Evalution/Messungen soll das in dieser Arbeit implementierte Programm analysiert und bewertet werden. Gemessen wurden das Einlesen und Schreiben der Daten, die dynamische Konfiguration der Fehlerwerte, sowie der Injektionsvorgang mit den verschiedenen Injektionsstrategien. Im Fokus der Messungen stand primär die Laufzeit der Fehlerinjektionen, ebenso wurden CPU-Auslastung und Speicherverbrauch untersucht.

## 6.1 Testaufbau

Für die Messungen der Funktionen wurde der folgende Testaufbau umgesetzt: Softwareseitig wurde das System Ubuntu 12.04 mit der Kernel Version 3.2.0-26-generic-pae eingesetzt. Die verwendete Javassist Library ist in Version 3.12.0.GA in die Anwendung integriert worden. Auf der Hardwareseite wurde ein Intel Core i7-2630QM Prozessor mit 8 $\times$ 2.00GHz und einem Arbeitsspeicher von 7.8 GiB verwendet.
Die konkreten Messergebisse konnten mit Hilfe des Tools VisualVM Version 1.3.4[1] bestimmt werden. Das Tool zeichnet die Aktivitäten eines gewählten Prozesses auf und gibt diese als Diagramm im Reiter Monitor wieder. Alle Laufzeitmessungen wurden direkt im Java Code integriert. Als Testinput dient eine 51.2MB große, durch den Output von /dev/zero[2], generierte Datei. Die getesteten Methoden wurden nicht separat sondern, wie in der Anwendung auch, über das $\mathrm{Controller}$-Objekt aufgerufen. Die Messung der Strategien wurden jeweils für die Blockgrößen 1, 8, 128, 1024 und 2048 mit den Fehlerraten 1%, 5%, 10%, 50% und 100% durchgeführt.

[1] http://visualvm.java.net/relnotes.html - Zugriff 16.09.12
[2] Unix virtuelle Gerätedatei, liefert Null-Bytes

## 6.2 Lese- und Schreiboperatinen

Für die Analyse der Einleseoperation ist die Method loadStream evaluiert worden. Die folgenden Messwerte aus Tabelle **??** beziehen sich auf das Einlesen der 51.2 MB Datei und der Datenübergabe an das Context Objekt.
Die Laufzeitmessungen der Methode loadStream ergaben einen Mittelwert von 2650 Millisekunden. Versuche mit anderen Java Streams konnten keine nennenswerten Verbesserungen erzielen. Die CPU-Auslastung der Methode stieg bei den Messungen durchschnittlich auf 20% bis 25% an. Der Speicherverbrauch lag bei den knapp unter 600 MB. Beim Schreiben der Daten durch die beiden unterschiedlichen Streams war der Laufzeitunterschied signifikanter. Die reduce Methode wurde mit dem ObjectOutputStream für serialisierbare Objekte und dem FileChannel getestet. Die Umsetzung mit dem FileChannel hat eine durchschnittliche Laufzeit von ca. 242 ms. Zum Schreiben der Daten als persistentes Objekt in Form der kompletten Java Liste bedarf es hingegen 113,88 Sekunden.

| loadStream | Laufzeit | CPU-Last | Speicher |
|---|---|---|---|
| | 2650 ms | 20% bis 25% | <600 MB |

Tabelle 6.1: Messwerte Einlesen und Datenübergabe

## 6.3 Dynamische Konfiguration der Fehlerwerte

Der folgende Test bezieht sich ausschließlich auf die Funktionalität der dynamischen Fehlerregulierung bzw. der Manipulierung der Annotationen zur Laufzeit. Für die Messungen wurden einem aktiven Stream der Klasse StreamProcessor 50 FaultInj Annotationen beigefügt. Im Testlauf wurden alle 50 Annotationen mit neuen Werten belegt und die Klasse StreamProcessor neu geladen.
Die Laufzeitmessung ergab einen Durschnittswert von 1.01 Sekunden. Für den Normalfall einer einzelnen Fehlermodifikation liegt der Wert bei 220 ms. Demnach wird die meiste Zeit für den Reload des modifizierten StreamProcessor's verwendet. Die CPU-Auslastung und der Speicherverbrauch sind in Abbildung **??** grafisch dargestellt. Bei der CPU-Auslastung konnte ein kontinuierlicher Anstieg bis auf knapp 20% ermittelt werden. Der Speicherverbrauch stieg während der Methodenausführung von 10 auf 15 MB nur geringfügig an.

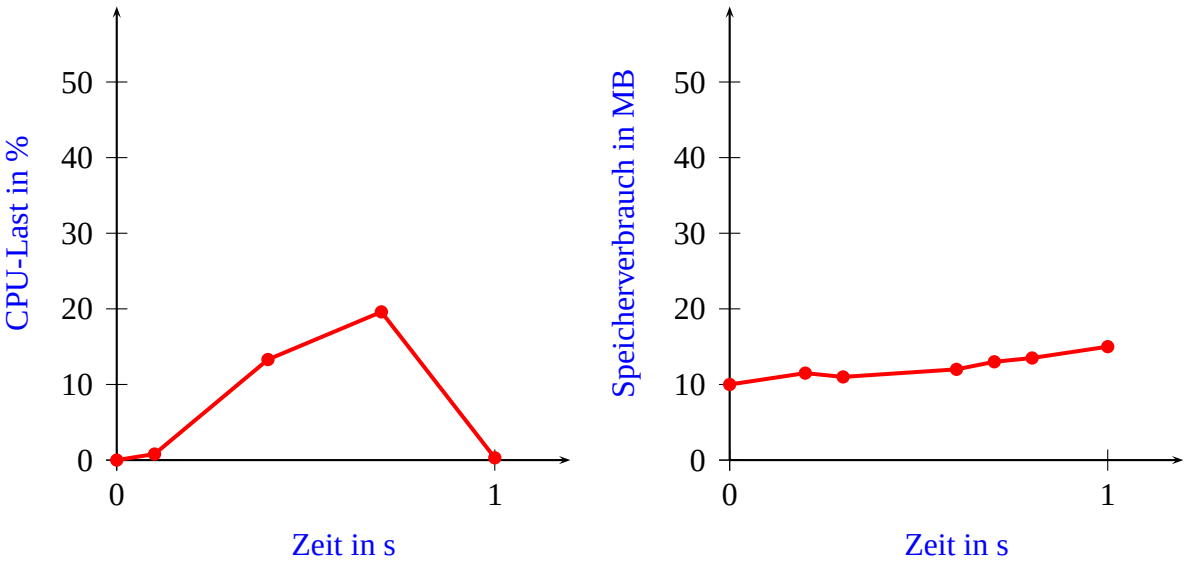



Abbildung 6.1: Messung dynamische Fehlerregulierung

## 6.4 Injektionsstategien

### Strategie Random

Die Random Strategie hat von allen Block-Strategien die längste Laufzeit. Dies resultiert aus der zusätzlichen Generierung von zufälligen Bytes.
In Abbildung **??** ist zu sehen wie die Laufzeit bei einer Blockvergrößerung sehr stark abnimmt, im Vergleich zu den übrigen Strategien. Je größer der Block, um so seltener muss die Zufallsgenerierung aufgerufen werden. Sie kann in diesem Fall bereits die Bytes für den ganzen Block in einer Interation generieren. Bei einer Blockgröße von 1 ruft die Funktion für jedes zu injizierende Byte die Zufallsgenereriung erneut auf, was zu einem Anstieg der Laufzeit führt.
Entsprechend verhält es sich auch mit der Fehlerrate, wie die Werte aus Tabelle **??** verdeutlichen. Bei einer steigenden Fehlerrate verlängert sich die Laufzeit, da mehr Bytes injiziert werden müssen und entsprechend häufig eine Zufallsgenerierung verlangt wird. Dementsprechend müssen bei einer Fehlerrate von 1.0 für jeden Block neue Zufallsbytes erstellt werden, was zu einem deutlichen Mehraufwand führt.

| RANDOM | Block | Rate: 0.01 | Rate: 0.05 | Rate: 0.1 | Rate: 0.5 | Rate: 1.0 |
|---|---|---|---|---|---|---|
| | 1 | 2.84 s | 2.91 s | 3.00 s | 3.85 s | 4.37 s |
| | 8 | 0.85 s | 0.86 s | 0.89 s | 1.33 s | 1.69 s |
| | 128 | 0.58 s | 0.59 s | 0.65 s | 0.96 s | 1.30 s |
| | 1024 | 0.54 s | 0.58 s | 0.62 s | 0.91 s | 1.28 s |
| | 2048 | 0.53 s | 0.57 s | 0.61 s | 0.90 s | 1.26 s |

Tabelle 6.2: Messwerte Random Strategie

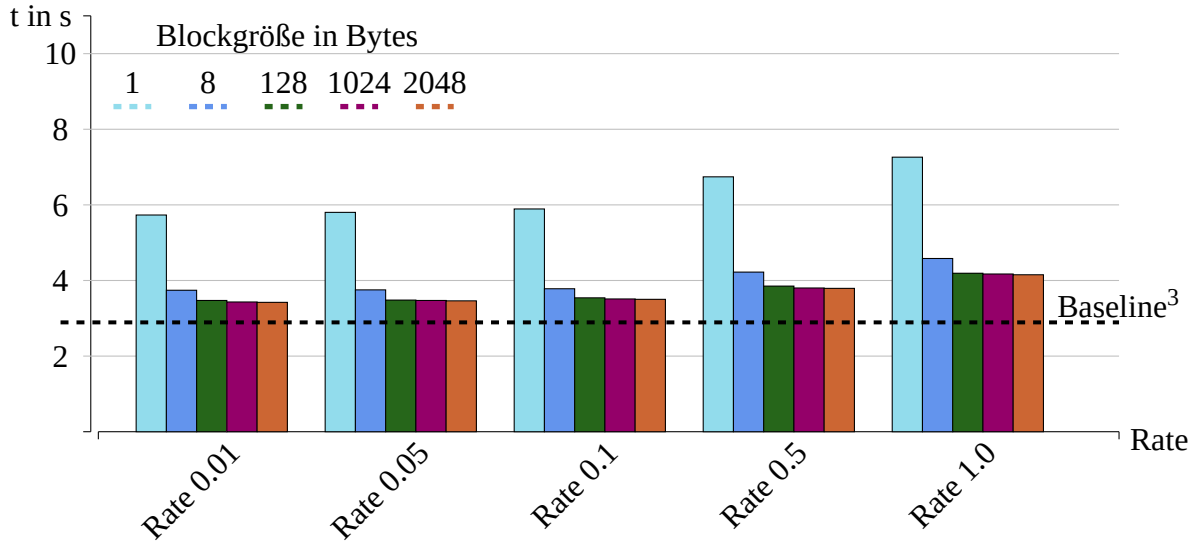



Abbildung 6.2: Injektionsstrategie Random mit Lese- und Scheiboperationen

## Strategie Loss

Die Loss Strategie wurde anfangs durch eine einfache Löschoperation in der Liste realisiert. Diese Variante stellte sich allerdings als sehr ineffizient heraus, da die interne Umstruktierung durch das Löschen die Anwendung erheblich verlangsamt. Die Messung für die 51,2 MB Testdatei wurde nach etwas 5 Minuten abgebrochen. Eine 1 MB Datei konnte innerhalb von 61,2 Sekunden injiziert werden.
Um das Löschen der Elemente zu umgehen wurde ein zweiter Ansatz gewählt, bei dem alle nicht injizierten Bytes einer neuen Liste angehangen werden. In der Auswertung aus Tabelle **??** spiegelt sich diese Art der Implementierung wieder. Im Gegensatz zu allen anderen Strategien verlaufen die Balken im Diagramm aus Abbildung **??** absteigend, je größer die Fehlerrate gewählt wurde. Dies resultiert aus der Tatsache, dass die Methode die injizierten Bytes einfach ignoriert und nur alle nicht injizierten Bytes in die Liste aufnimmt. Eine Ausnahme bildet die

[3] Ausführung ohne Fehlerinjektion

Blockgöße 1, bei der sich das überspringen der injizierten Bytes erst ab einem Wert größer als 0.5 lohnt. Eine detaillierte Begründung dieses Phänomens gibt es im Abschnitt **??** über Branch Prediction.

| LOSS | Block | Rate: 0.01 | Rate: 0.05 | Rate: 0.1 | Rate: 0.5 | Rate: 1.0 |
|---|---|---|---|---|---|---|
| | 1 | 3.44 s | 3.47 s | 3.56 s | 3.86 s | 2.79 s |
| | 8 | 1.52 s | 1.47 s | 1.45 s | 1.27 s | 0.86 s |
| | 128 | 1.17 s | 1.15 s | 1.13 s | 0.89 s | 0.60 s |
| | 1024 | 1.16 s | 1.13 s | 1.12 s | 0.88 s | 0.57 s |
| | 2048 | 1.14 s | 1.12 s | 1.11 s | 0.88 s | 0.57 s |

Tabelle 6.3: Messwerte Loss Strategie

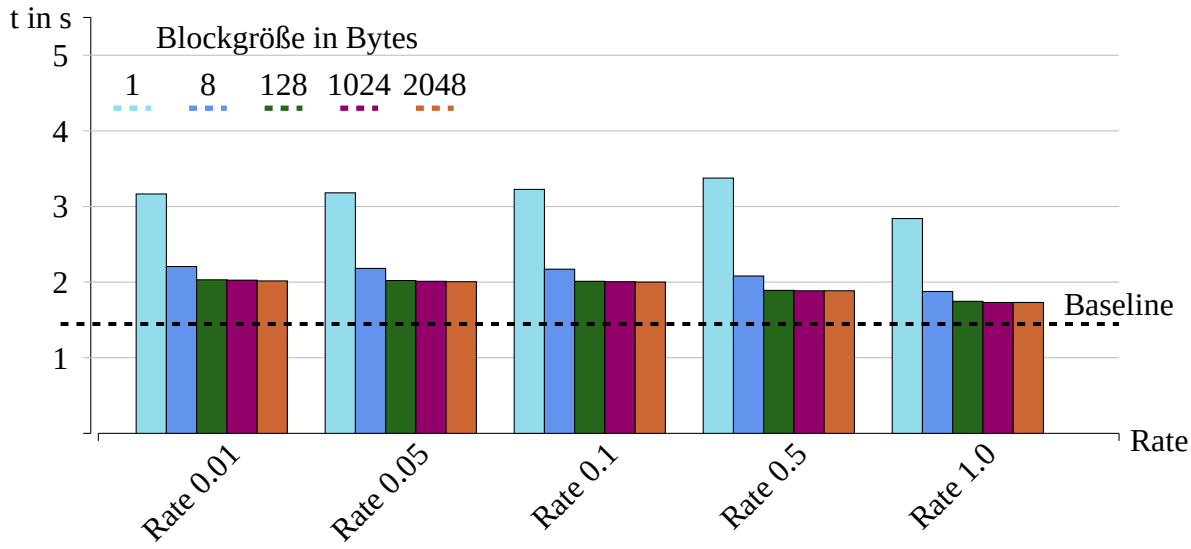



Abbildung 6.3: Injektionsstrategie Loss mit Lese- und Schreiboperationen

## Strategie Zero

Bei der Zero Strategie zeigt sich ein ähnlicher Verlauf, wie bei der Random Strategie. Durch eine zunehmende Blockgröße sinkt die Laufzeit und bei zunehmender Fehlerrate steigt diese wieder. Dennoch ist der Unterschied zwischen Random und Zero, speziell bei niedrigen Blockgrößen, ziemlich groß, da die Zero Funktion für ihre Injektion lediglich ein Null-Byte setzt und Random ständig neue Werte generieren muss. Einzige Ausnahme aus Tabelle **??** bildet wieder die Blockgröße von 1, bei der ab einer Fehlerrate größer als 0.5 die Laufzeit wieder sinkt (siehe Abschnitt **??**)

| ZERO | Block | Rate: 0.01 | Rate: 0.05 | Rate: 0.1 | Rate: 0.5 | Rate: 1.0 |
|---|---|---|---|---|---|---|
| | 1 | 2.75 s | 2.82 s | 2.87 s | 3.37 s | 3.07 s |
| | 8 | 0.82 s | 0.84 s | 0.86 s | 0.98 s | 1.04 s |
| | 128 | 0.56 s | 0.57 s | 0.57 s | 0.64 s | 0.69 s |
| | 1024 | 0.54 s | 0.56 s | 0.56 s | 0.61 s | 0.67 s |
| | 2048 | 0.54 s | 0.55 s | 0.56 s | 0.61 s | 0.66 s |

Tabelle 6.4: Messwerte Zero Strategie

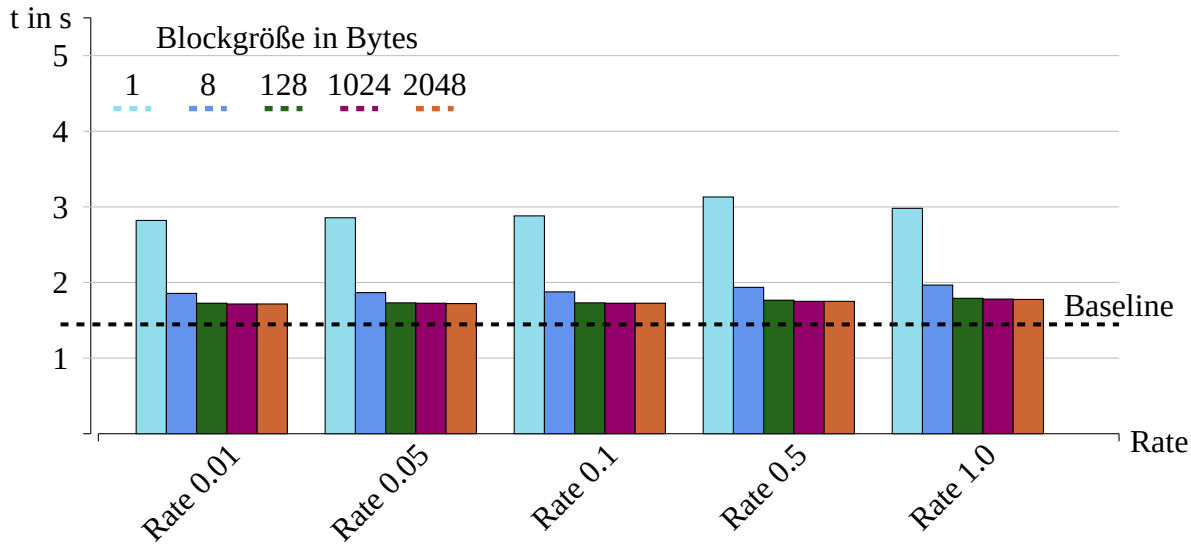



Abbildung 6.4: Injektionsstrategie Zero mit Lese- und Scheiboperationen

## Strategie BitflipB

Die Verteilung der Messerte von BitflipB aus Tabelle **??** ist Analog zu der Zero Strategie. Auffallend ist, dass selbst die konkreten Werte der beiden Strategien, sehr nahe bei einander liegen, wie das Diagramm aus Abbildung **??** verdeutlicht. Dies resultiert aus der Verwendung des gleichen Algorithmus für beide Strategien ohne zusätzliche Methodenaufrufe. Die Zero Strategie tauscht ein Byte durch das Null-Byte und BitflipB durch das geflippte Byte. Ansonten sind beide Strategien identisch.

## Strategie Bitflip

Die Messwerte aus Tabelle **??** zeigen wie aufwendig die normale Bitflip Strategie ist. Grund dafür ist die Bearbeitung jedes einzelnen Bits und der Tatsache, dass keine Blöcke verwendet werden können. Damit leistet diese Strategie einen beachtlichen Mehraufwand gegenüber den

| BITFLIPB | Block | Rate: 0.01 | Rate: 0.05 | Rate: 0.1 | Rate: 0.5 | Rate: 1.0 |
|---|---|---|---|---|---|---|
| | 1 | 2.80 s | 2.79 s | 2.86 s | 3.30 s | 3.02 s |
| | 8 | 0.73 s | 0.76 s | 0.79 s | 1.01 s | 1.18 s |
| | 128 | 0.47 s | 0.48 s | 0.51 s | 0.69 s | 0.88 s |
| | 1024 | 0.45 s | 0.46 s | 0.48 s | 0.65 s | 0.87 s |
| | 2048 | 0.44 s | 0.46 s | 0.48 s | 0.65 s | 0.87 s |

Tabelle 6.5: Messwerte BitflipB Strategie

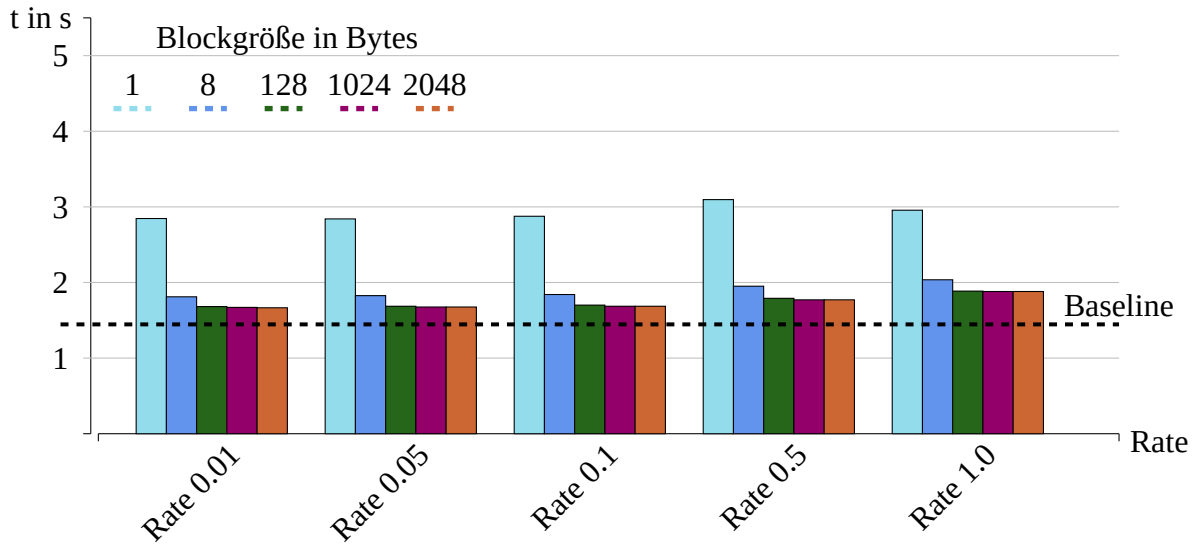



Abbildung 6.5: Injektionsstrategie BitflipB mit Lese- und Scheiboperationen

bisherigen Injektionsverfahren. In Abbildung **??** ist der dieser Overhead im Vergleich zur Baseline sehr gut erkennbar.

| BITFLIP | Block | Rate: 0.01 | Rate: 0.05 | Rate: 0.1 | Rate: 0.5 | Rate: 1.0 |
|---|---|---|---|---|---|---|
| | Kein Block | 18.05 s | 18.60 s | 18.82 s | 21.35 s | 18.67 s |

Tabelle 6.6: Messwerte Bitflip Strategie

## CPU-Auslastung & Speicherverbrauch der Strategien

Die CPU-Auslastung und der Speicherverbrauch wurden bei einer Blöckgröße von 1 gemessen, da diese die längsten Messungen ermöglicht. Bei allen Strategien konnte für die CPU-Auslastung nach dem Lesen der Daten, ein konstanter Wert zwischen 12.4% und 12.8% ermittelt werden. Auch der Speicherverbrauch erreichte einen konstanten Wert von ca. 440 MB. Die einzige

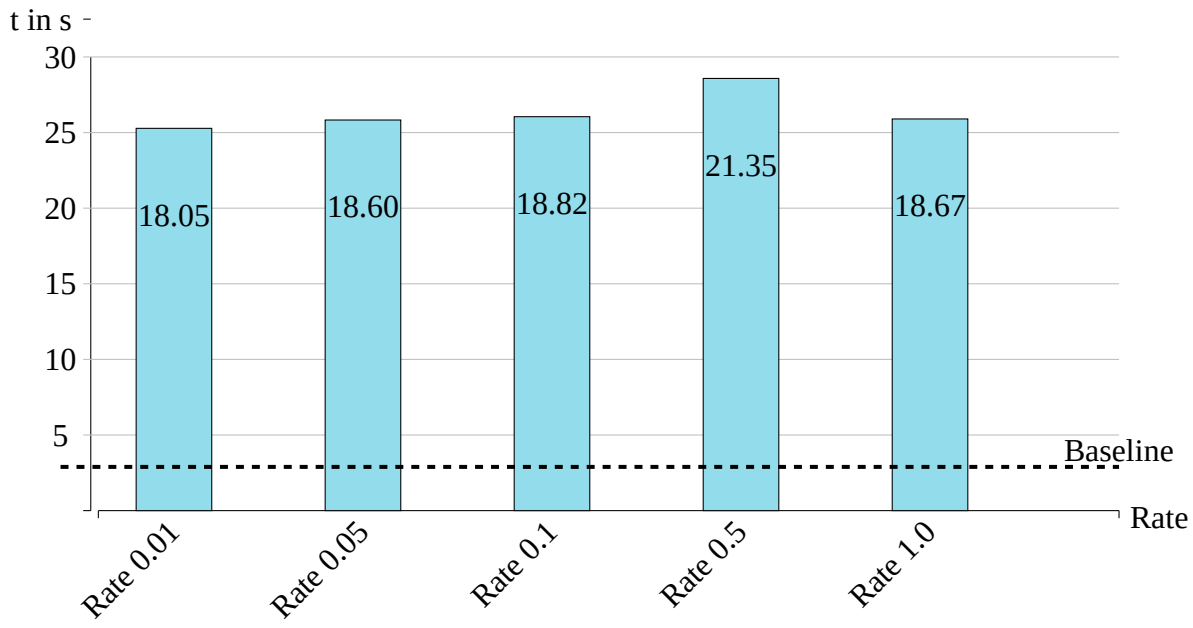



Abbildung 6.6: Injektionsstrategie Bitflip mit Lese- und Scheiboperationen

Ausnahme im Bezug auf den Speicherbereich, bildete die Loss Strategie mit einem Wert von 490 MB. Dies ist dem Mechanismus zur Bestimmung der nicht injizierten Werte geschuldet. Es werden hierfür alle Fehlerwerte in einer weiteren Datenstruktur markiert. Am Ende der Strategie findet die Datenübertragung in die Ergebnisliste statt, was zusätzlich zu einem rapiden Anstieg des Speicherverbrauchs führt. Die Messwerte diesbzgl. sind in Tabelle **??** angegeben. Es ist schnell zu erkennen das der Anstieg am Ende der Methode, abhängig von der Anzahl der verbliebenen Bytes ist. Bei einem Wert von 0.1 werden nur wenige Bytes entfernt, wodurch die Ergebnisliste sehr groß ausfällt. Liegt die Fehlerrate allerdings bei 1.0 werden alle Bytes entfernt, wodurch die Ergebnisliste leer bleibt.

| Loss | Fehlerrate | Speicher zu Beginn (MB) | Speicher bei Übertragung (MB) |
|---|---|---|---|
| | 0.1 | 490 | 675 |
| | 0.3 | 490 | 634 |
| | 0.5 | 490 | 592 |
| | 0.7 | 490 | 552 |
| | 1.0 | 490 | 490 |

Tabelle 6.7: Messwerte Speicherverbrauch Loss

## 6.5 Branch Prediction

Ein branch predictor in einer Computer Architekur, ist eine digitale Schaltung die versucht, den nächsten richtigen Weg in einer Verzweigung zu erraten, bevor dieser sicher bekannt ist. Die Tabelle **??** zeigt exemplarisch die Zero Strategie mit den gemessenen Fehlerwerten von 0.1 bis 1.0, die Anzahl der ausgeführten Injektionen und die jeweilige Laufzeit. Man sieht beispielsweise bei einer Fehlerrate von 0.3, eine deutlich geringere Anzahl an Injektionen, als bei einer Fehlerrate von 1.0. Dennoch ist die Laufzeit bei einer Fehlerrate von 0.3 höher. Genauer lässt sich feststellen das Fehlerwerte die näher bei 0.5 liegen, mit hoher Wahrscheinlichkeit mispredictions also Fehlvorhersagen erzeugen (ungefähr eine pro Durchlauf), die Performance Einbuße nach sich ziehen. Liegt der Wert hingegen nahe bei 0 oder 1 sind die branch predictions oft exakt, was die Performance erheblich beschleunigt [4]. Besonders gut wird dieses Verhalten im folgenden Diagramm "Durschnittliche Laufzeit Zero" verdeutlicht.

| Zero | Fehlerrate | Anzahl Injektionen | Laufzeit in ms |
|---|---|---|---|
| | 0.1 | 5121757 | 2870 |
| | 0.2 | 10238034 | 3079 |
| | 0.3 | 15357873 | 3150 |
| | 0.4 | 20483144 | 3279 |
| | 0.5 | 25598114 | 3372 |
| | 0.6 | 30722540 | 3332 |
| | 0.7 | 35841016 | 3267 |
| | 0.8 | 40959774 | 3207 |
| | 0.9 | 46079605 | 3183 |
| | 1.0 | 51200000 | 3070 |

Tabelle 6.8: Messwerte bzgl. Branch Prediction

## 6.6 Zusammenfassung

Die ausgeführten Tests haben gezeigt, dass die Modifikation der Annotationen, selbst bei einer großen Anzahl neuer Werte, die Laufzeit kaum beeinträchtigt. Die CPU-Auslastung stieg kurzzeitig auf 20% bei einem minimal ansteigenden Speicherverbrauch. Bei den Fehlerinjektionen teilt sich das Feld in drei Kategorien auf. Die Zero, BitflipB und auch die etwas aufwendigere Random Strategie, zeichnen sich durch eine schnelle Laufzeit sowie einen kostanten Speicherverbrauch aus. Die Loss Strategie ist in ihrer Laufzeit unkritisch, verursacht aber einen hohen Speicherverbrauch im letzten Abschnitt ihrer Berechnung. Dies sollte

[4] bei 0 oder 1 keine mispredictions

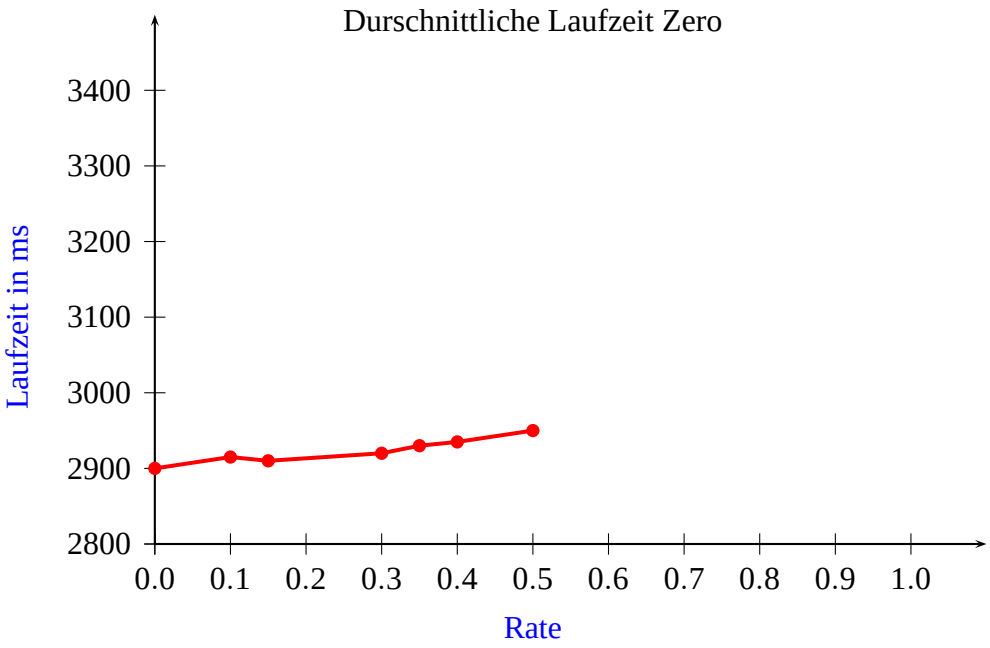



Abbildung 6.7: Messung Branch Prediction

bei ihrer Verwendung berücksichtigt werden. Dagegen benötigt Bitflip für die Ausführung erheblich mehr Zeit, weshalb alternativ die Strategie BitflipB erstellt wurde. Diese Messungen zeigen, dass die Implementierung zur Evaluation großer verteilter Systeme geeignet ist. Der gemessene CPU- und Memory-Overhead ist trotz zu erwartender großer Datenmengen in einem akzeptablen Bereich. Messungen zur Energieeffizenz dieser Systeme sollten, trotz der vorgenommenen Fehlerinjektion, weiterhin aussagekräftig bleiben.

Kapitel 7

# 7 Fazit

Aufgrund der ständig steigenden Anforderungen für Server und Rechenzentren und den damit verbundenen hohen Eniergiekosten, wird es zukünftig immer wichtiger die knappe Ressource Strom effizient nutzen zu können. Es existieren bereits verschiedene Ansätze und neue Verfahren die schon heute einen Schritt in die richtige Richtung machen. Das in dieser Bachelorarbeit erwähnte Paradigma *Accuracy-Aware* ist einer dieser Ansätze, auf dessen Grundlage die beschriebene Applikation beruht. Die Anwendung soll für den Bereich der Identifikation von fehlerbehafteten Daten des Paradigmas eingesetzt werden.
Zu diesem Zweck wurde eine Lösung entwickelt, die es erlaubt, Streams mit speziellen Annotationen zu versehen, um über diese Fehlerwerte in die eingelesen Daten zu streuen. Eine wichtige Eigenschaft der Anwendung sollte die dynamische Regulierbarkeit zur Laufzeit jener Fehlerwerte sein. Die definierten Fehler reichen von diversen Fehlertypen, einer flexiblen Fehlerrate von 0% bis 100% bis zu der Möglichkeit ganze Blöcke in die Injektion einzubeziehen.
Bei der Implementierung wurde versucht möglichst Modular zu arbeiten um eine spätere Erweiterbarkeit des Programms zu gewährleisten. Am deutlichsten wird dies bei der Erstellung neuer Fehlerstrategien sichtbar. Diese können separat als eigene Klasse implementiert und mit wenig Aufwand in die Anwendung eingeflochten werden. Während die Annotationen verwendet wurden um Streams zu markieren die fehlerbehaftet sein können, wurde die JMX Technologie eingesetzt um die Anwendung zu überwachen und dynamisch konfigurierbar zu machen. Durch JMX wird eine einfache Schnittstelle bereitgestellt, über die sich die Anwendung flexibel ausführen respektive in andere Implementierungen integrieren lässt.
Die Verwendbarkeit der implementierten Lösung wurde im Kapitel Evaluation durch verschiedene Messungen analysiert. Dabei wurde der Fokus auf Laufzeit, CPU-Auslastung und Speicherverbrauch gelegt.

# Abbildungsverzeichnis


2.1 AccuracyAwareness: Software-Architektur . . . . . 5
2.2 RMI Kommunikations-Architektur . . . . . 6
2.3 JMX-Architektur . . . . . 7
2.4 Javassist Code-Manipulation . . . . . 9
2.5 Funktionsweise HotSwapper . . . . . 10

4.1 Anwendungsfalldiagramm . . . . . 17
4.2 UML StreamProcessor . . . . . 19
4.3 UML Strategy Pattern . . . . . 21
4.4 UML Dynamische Konfiguration . . . . . 23
4.5 UML MBean Interface . . . . . 25
4.6 Sequenzdiagramm . . . . . 28

6.1 Messung dynamische Fehlerregulierung . . . . . 46
6.2 Messung Random Strategie . . . . . 47
6.3 Messung Loss Strategie . . . . . 48
6.4 Messung Zero Strategie . . . . . 49
6.5 Messung BitflipB Strategie . . . . . 50
6.6 Messung Bitflip Strategie . . . . . 51
6.7 Messung Branch Prediction . . . . . 53

# Listings


5.1 Annotation FaultInj . . . . . 30
5.2 Annotation Default-Werte . . . . . 30
5.3 Annotation FaultInjects . . . . . 30
5.4 Enum FaultType . . . . . 31
5.5 StreamProcessor Einlesen der Daten . . . . . 31
5.6 StreamProcessor Fehlermarkierung . . . . . 32
5.7 StreamProcessor Einlesen/Injektion/Ausgabe . . . . . 33
5.8 StreamProcessor Fehlerwertrückgabe . . . . . 33
5.9 Initialisierung HotSwapper & CtClass . . . . . 34
5.10 Bestimmung injizierter Felder . . . . . 35
5.11 Neue Annotationen . . . . . 35
5.12 Zuweisung des Annotationstyps . . . . . 35
5.13 Setzen der Fehlerwerte . . . . . 36
5.14 Laden des Bytecodes . . . . . 37
5.15 Context: Einlesen der Datensätze . . . . . 37
5.16 Context: Strategieauswahl . . . . . 38
5.17 Context: Injektionsausführung . . . . . 38
5.18 InjectionStrategy . . . . . 39
5.19 ControllerMBean . . . . . 40
5.20 MainServer . . . . . 40
5.21 MainServer RMI . . . . . 41
5.22 Bash-File und Ausführungsliste . . . . . 42
5.23 Java Client . . . . . 43

# Tabellenverzeichnis

6.1 Messwerte Einlesen und Datenübergabe . . . . . 45
6.2 Messwerte Random Strategie . . . . . 47
6.3 Messwerte Loss Strategie . . . . . 48
6.4 Messwerte Zero Strategie . . . . . 49
6.5 Messwerte BitflipB Strategie . . . . . 50
6.6 Messwerte Bitflip Strategie . . . . . 50
6.7 Messwerte Speicherverbrauch Loss . . . . . 51
6.8 Messwerte bzgl. Branch Prediction . . . . . 52

# Erklärung

Ich erkläre hiermit, dass ich diese Bachelorarbeit mit dem Titel *Identifikation von fehlerbehafteten Daten zur energiebewußten Programmierung* selbstständig ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel verfasst habe. Alle den benutzten Quellen wörtlich oder sinngemäß entnommenen Stellen sind als solche einzeln kentlich gemacht.

Diese Arbeit ist bislang keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht worden.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Marburg den 13.08.12

René Frank