Bachelorarbeit

Indentifikation von fehlerbehafteten Daten zur energiebewußten Programmierung

René Frank

Philipps-Universität Marburg
FB Mathematik & Informatik

frank@mathematik.uni-marburg.de

25. Januar 2013

Accuracy-Aware Data Processing

- steigende Energiekosten
- immer höhere Rechenleistungen
- wachsende Zahl an Online-Services

Accuracy-Aware Data Processing

- fehlertoleranz ausnutzen
- Fehlerfreiheit und Genauigkeit reduzieren

Quelle Bild 1: http://www.dreamstime.com/ - Zugriff: 15.10.12
Quelle Bild 2: http://gsablogs.gsa.gov/gsablog/files/2012/09/Cloud-computing-9-25-12b.jpg - Zugriff: 15.10.12

Motivation/Zielsetzung

Fehlerinjektion für Datenströme

- Fehlerinjektion minimal-invasiv
- Nur einen minimalen Overhead erzeugen
- Alle Arten von Datenströmen sollten berücksichtigt werden
- Eine dynamische Regulierung der Fehlerwerte zur Laufzeit

Fehlerinjektion für Datenströme

- Fehlerinjektion minimal-invasiv
- Nur einen minimalen Overhead erzeugen
- Alle Arten von Datenströmen sollten berücksichtigt werden
- Eine dynamische Regulierung der Fehlerwerte zur Laufzeit

Fehlerinjektion für Datenströme

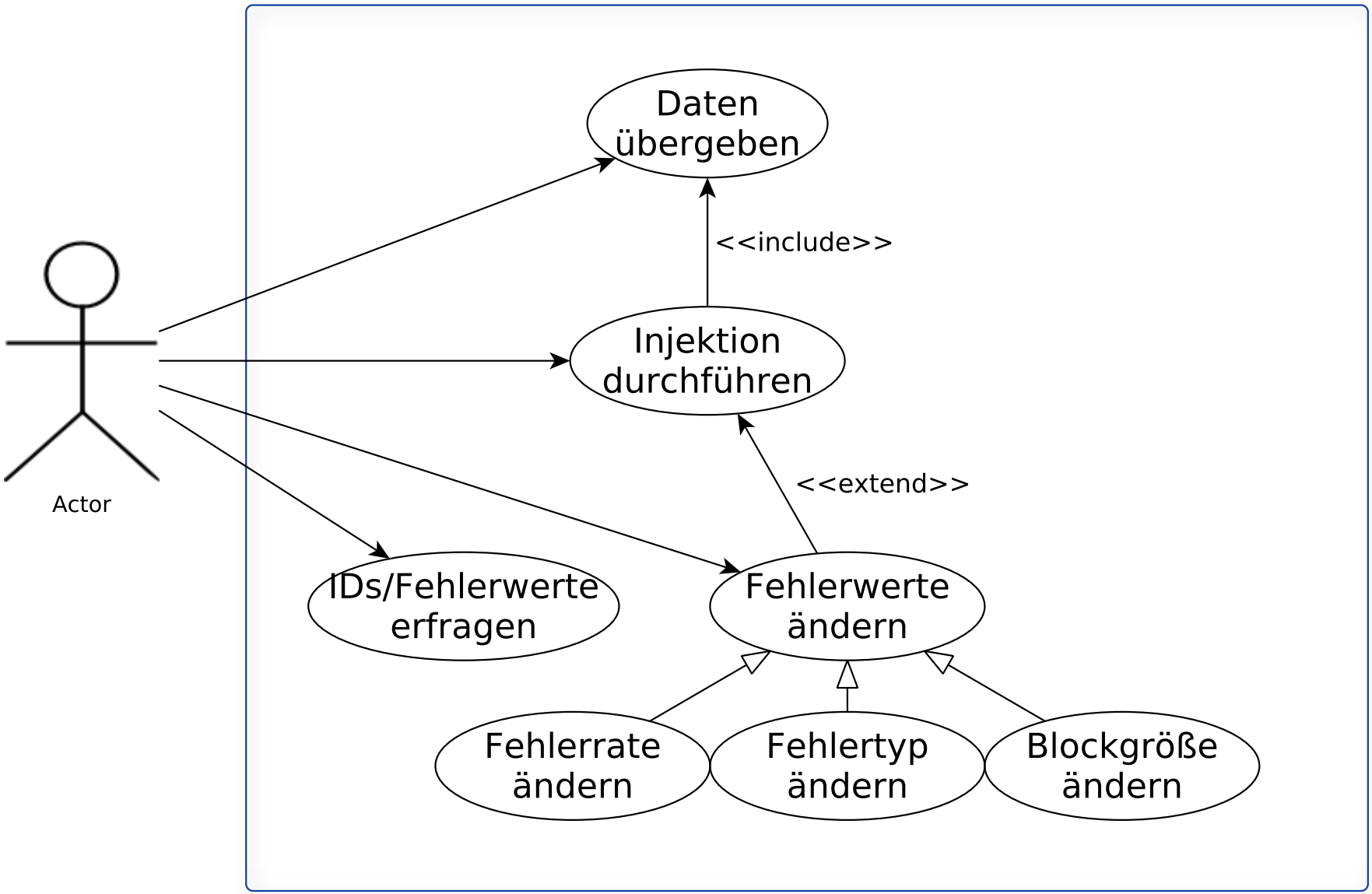
- Fehlerinjektion minimal-invasiv
- Nur einen minimalen Overhead erzeugen
- Alle Arten von Datenströmen sollten berücksichtigt werden
- Eine dynamische Regulierung der Fehlerwerte zur Laufzeit

Fehlerinjektion für Datenströme

- Fehlerinjektion minimal-invasiv
- Nur einen minimalen Overhead erzeugen
- Alle Arten von Datenströmen sollten berücksichtigt werden
- Eine dynamische Regulierung der Fehlerwerte zur Laufzeit

Design/Implementierung

Funktionen


Daten
übergeben
<<include>>
Injektion
durchführen
<<extend>>
Actor
IDs/Fehlerwerte
erfragen
Fehlerwerte
ändern
Fehlerrate
ändern
Fehlertyp
ändern
Blockgröße
ändern

Wichtige Klassen

- StreamProcesser
  - Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy
  - Oberklasse der Injektionsstrategien
- AddRunTimeAnnotation
  - Dynamische Modifikation der Annotationen
- Controller
  - Steurerung aller Funktionen

StreamProcessor

Aufgaben

- Daten fehlerbehafteter Streams einlesen
  - Fehlermarkierung durch Annotationen

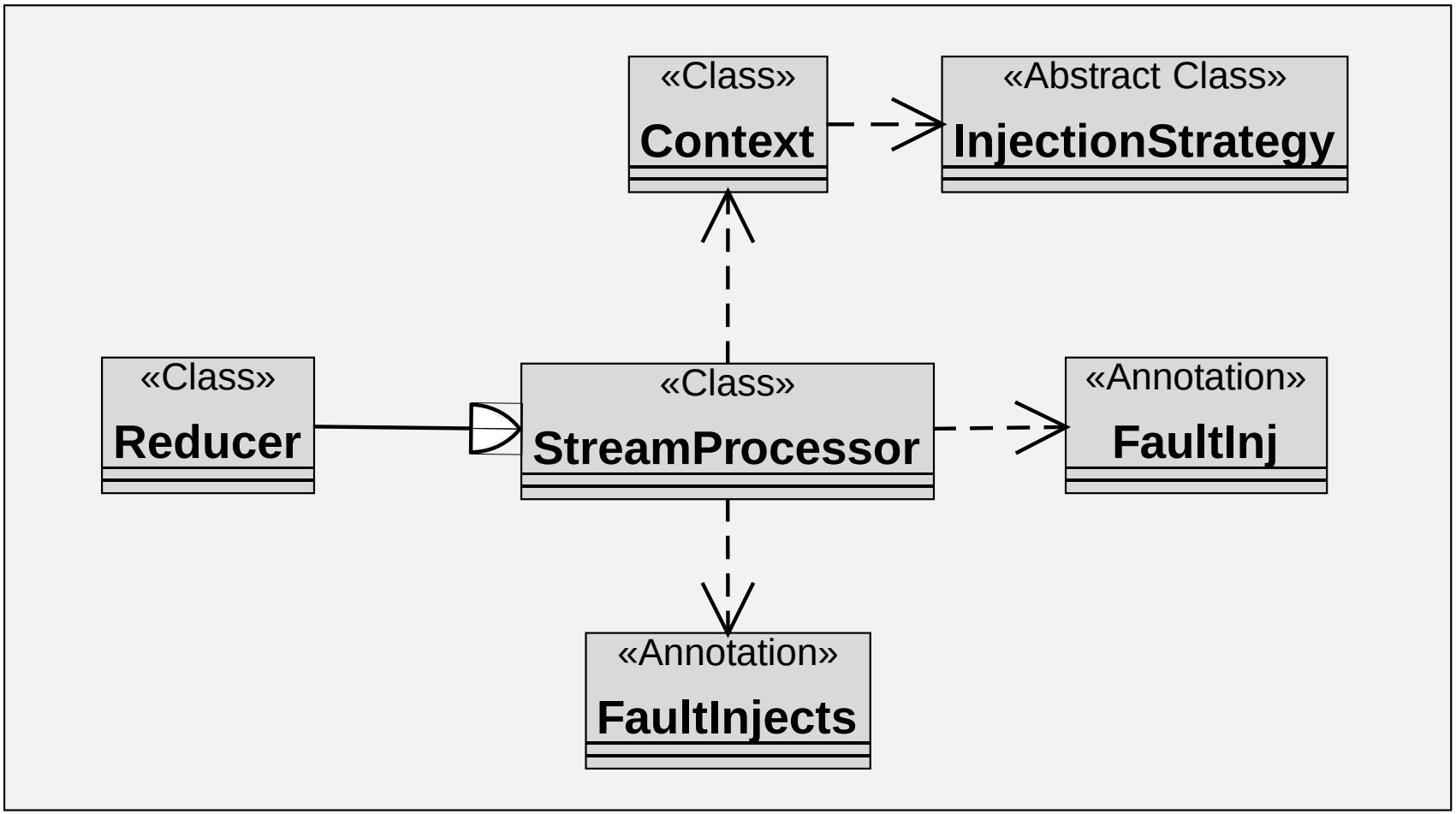
```
1     public @interface FaultInj {
2         String id();
3         String type() default "NONE";
4         double rate() default 0.0;
5         long blocksize() default 1024;
6     }
```

- als Byte-Liste zur Datenmanipulation an Klasse Context weitergeben
- manipulierte Daten holen und in entsprechender Form ausgeben

Fehlermarkierung

- id muss eindeutig sein
- type muss vorhanden sein
- rate liegt zwischen 0 und 1

```
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    ...
```


«Class»
Context
«Abstract Class»
InjectionStrategy
«Class»
Reducer
«Class»
StreamProcessor
«Annotation»
FaultInj
«Annotation»
FaultInjects

Wichtige Klassen

- StreamProcesser
  - Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy
  - Oberklasse der Injektionsstrategien
- AddRunTimeAnnotation
  - Dynamische Modifikation der Annotationen
- Controller
  - Steurerung aller Funktionen

Wichtige Klassen

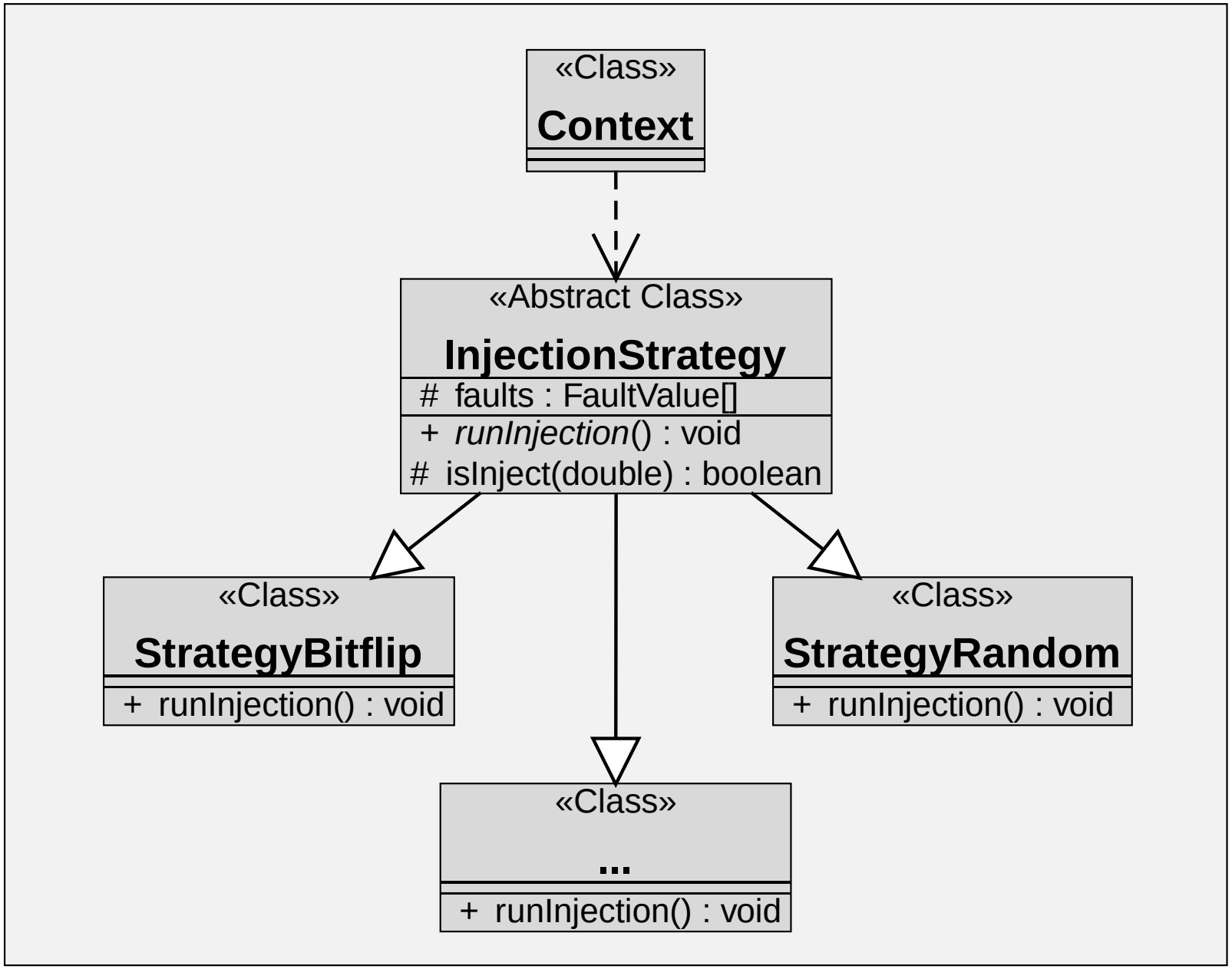
- StreamProcesser
  - Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy
  - Oberklasse der Injektionsstrategien
- AddRunTimeAnnotation
  - Dynamische Modifikation der Annotationen
- Controller
  - Steurerung aller Funktionen

InjectionStrategy

- abstrakte Oberklasse aller Strategien
- keine eigene Injektionslogik $\rightarrow$ abstrakte Methode
- Zufallsfunktion zur Fehlerinjektion
  - entscheidet anhand Fehlerrate über Injektion

Strategien

- *Bitflip*
- *BitflipB*
- *Random*
- *Loss*
- *Zero*


«Class»
Context
«Abstract Class»
InjectionStrategy
faults : FaultValue[]
+ runInjection() : void
# isInject(double) : boolean
«Class»
StrategyBitflip
+ runInjection() : void
«Class»
StrategyRandom
+ runInjection() : void
«Class»
...
+ runInjection() : void

Strategieauswahl

```
1  InjectionStrategy getStrategy(List data, FaultValue fault){
2
3    if(fault.getType().equals(FaultType.ZERO.name()))
4      return new StrategyZero(data,fault);
5    else if(fault.getType().equals(FaultType.RANDOM.name()))
6      return new StrategyRandom(data, fault);
7      ...
8
9    throw new NoSuchTypeException(fault.getType());
10 }
```

Wichtige Klassen

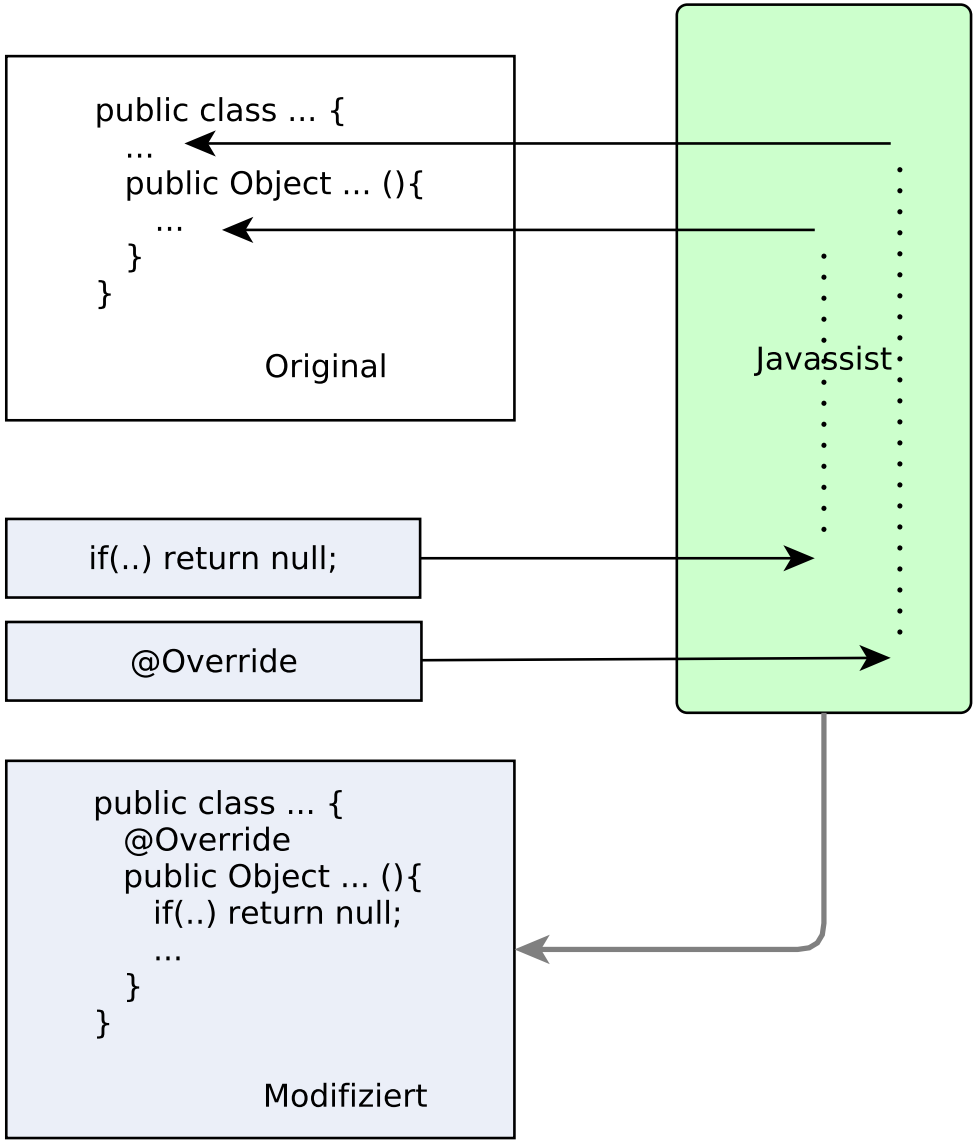
- StreamProcesser
  - Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy
  - Oberklasse der Injektionsstrategien
- AddRunTimeAnnotation
  - Dynamische Modifikation der Annotationen
- Controller
  - Steurerung aller Funktionen

Wichtige Klassen

- StreamProcesser
  - Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy
  - Oberklasse der Injektionsstrategien
- AddRunTimeAnnotation
  - Dynamische Modifikation der Annotationen
- Controller
  - Steurerung aller Funktionen

AddRunTimeAnnotation

- Library Javassist
- Manipulation der Annotationen
- reload der Class-Datei



Quelle: http://www.sciencedirect.com/science/article/pii/S0164121211000501 - Zugriff: 15.10.12

Bestimmung der zu manipulierenden Streams

- Ct = comile-time
- cc vom Typ CtClass

```
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         ...
```

Anlegen neuer Annotationen

- Unterscheidung FaultInj und FaultInjects
- anlegen der entsprechenden Annotation in Annotation-Array

```
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  }
```

Hinzufügen der Fehlerwerte zu FaultInjects

- *fis* = Array mit Fehlerwerte eines markierten Streams

```
1   for (int x = 0; x < len; x++) {
2
3     if (fis[x].id().equals(faults[i].getId())) {
4       subAnnotations[x] = new Annotation(
5         "faults.FaultInj", constpool);
6         subAnnotations[x].addMemberValue(
7             "id",
8             new StringMemberValue(faults[i].getId(),
9                 ccFile.getConstPool()));
10        subAnnotations[x].addMemberValue(
11            "rate",
12            new DoubleMemberValue(faults[i].getRate(),
13                ccFile.getConstPool()));
14        ...
15    ...
```

Laden der Klasse mit neuen Annotationen

- ctClass zu normaler Class Datei umwandeln
- javassist.util.HotSwapper $\rightarrow$ Bytecode reload

```
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();
```

Wichtige Klassen

- StreamProcesser
  - Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy
  - Oberklasse der Injektionsstrategien
- AddRunTimeAnnotation
  - Dynamische Modifikation der Annotationen
- Controller
  - Steurerung aller Funktionen

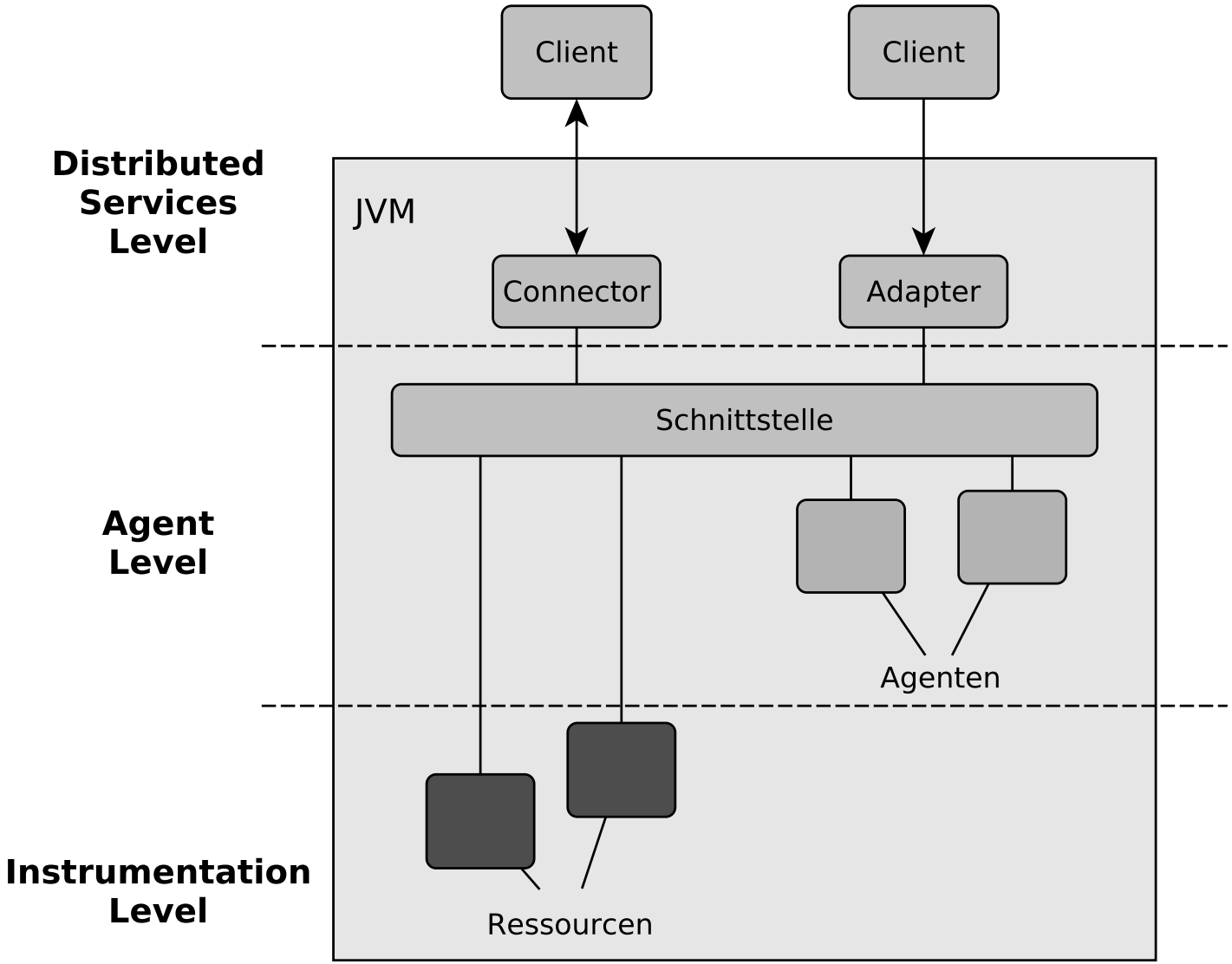
Wichtige Klassen

- StreamProcesser
  - Einlesen und Ausgabe der Daten, Start der Fehlerinjektion
- InjectionStrategy
  - Oberklasse der Injektionsstrategien
- AddRunTimeAnnotation
  - Dynamische Modifikation der Annotationen
- Controller
  - Steuerung aller Funktionen

Controller

- Steuert alle Funktionalitäten
- Verwaltung der Funktionalitäten durch JMX-Agent
  - Komponenten zur Laufzeit managen
  - Anwendungsfunktionalität leichter administrierbar
- Schnittstelle MBeanController
  - liefert dem Client Zugriff auf gegebene Funktionen

Controller/MBean

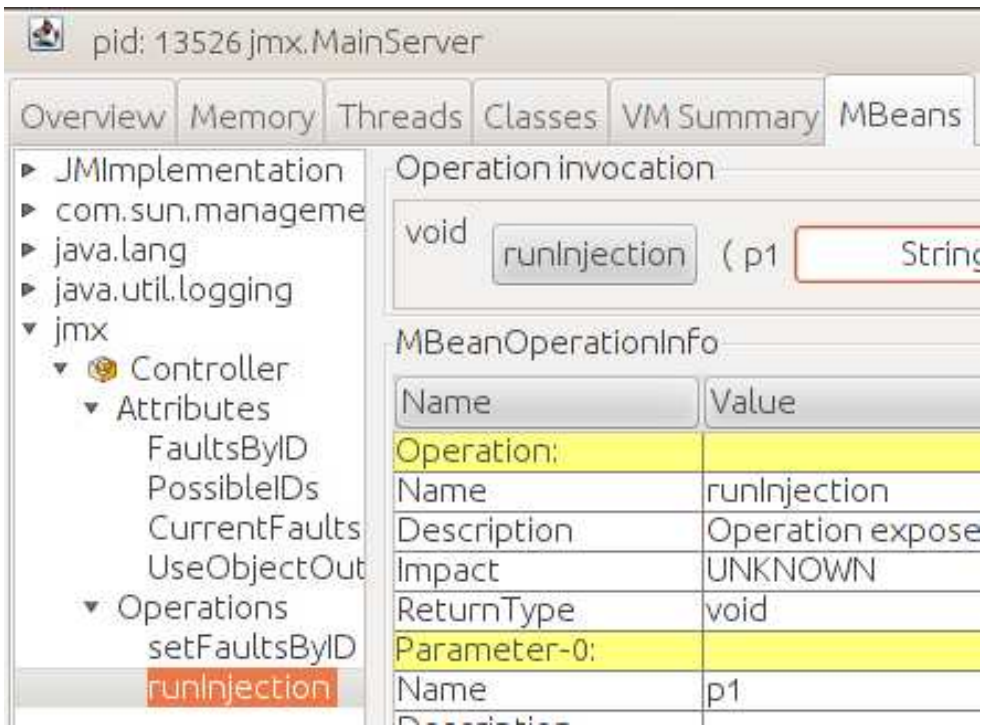


Quelle: http://de.wikipedia.org/wiki/Datei:Jmx_architektur.png - Zugriff: 15.10.12

Schnittstelle zum MBeanServer

- bereitgestellte Funktionalitäten

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



Verbindungsaufbau (1)

```
1     if (args.length >= 2) {
2         LocateRegistry.createRegistry(Integer.parseInt(args
              [1]));
3
4         JMXServiceURL url = new JMXServiceURL(
5                 "service:jmx:rmi:///jndi/rmi://" + args[0] +
                      ":" + args[1]
6                         + "/server");
7
8         JMXConnectorServer cs = JMXConnectorServerFactory
9                 .newJMXConnectorServer(url, null, mbs);
10        ...
11        cs.start();
12    }
```

Verbindungsaufbau (2)

```
1     // Get the Platform MBean Server
2     MBeanServer mbs = ManagementFactory.
          getPlatformMBeanServer();
3
4     // Construct the ObjectName for the Controller MBean we
          will register
5     ObjectName mbeanName = new ObjectName("jmx:type=
          Controller");
6
7     // Create the Controller MBean
8     Controller mbean = new Controller();
9
10    // Register the Controller MBean
11    mbs.registerMBean(mbean, mbeanName);
12    ...
13    // Wait forever
14    System.out.println("Server is running now");
15    Thread.sleep(Long.MAX_VALUE);
16 }
```