Technische Hochschule Rosenheim
Technical University of Applied Sciences

Verteilte Verarbeitung

Kapitel 2.1
Reaktive Systeme

Lernziel

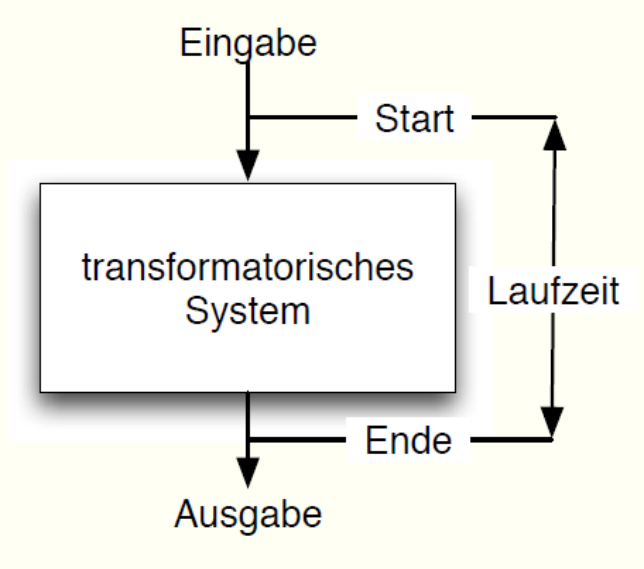
Sie wissen …

- Was ein ***reaktives System*** ist
- Wie man einen endlichen Automaten programmiert
- Welchen Sachverhalt ein Stream abstrahiert

Reaktive und tranformatorische Systeme

Transformatorische Systeme

```
public static void main (
                    String[] args) {
        // Initialisierung
        // Eingabe
        // Verarbeitung
        // Ausgabe
} // Terminiert
```

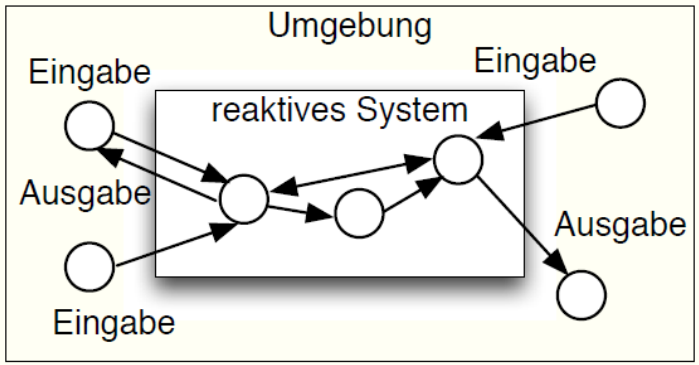


- Transformatorische Systeme ***terminieren***
- Beispiele: Batch-Jobs, Shell-Programme (grep, sed)

Beispiel für ein *Reaktives System*

```
public static void main(String[] args)
{
    // Initialisierung
    while (true) {
        // Eingaben
        // Verarbeitung
        // ggf. Abbruchbedingung
        // Ausgaben
    }
    // Nach Abbruch Ressourcen
    // ggf. freigeben
}
```

Typisch für reaktives System:
Endlosschleife



Reaktive Systeme

A **reactive system** is a computer program that ***continuously*** interacts with its environment.

- Beispiele für Reaktive Systeme
  - Regelungssysteme, Prozesssteuerungen
  - Client ***und*** Server
- Endlosschleife: `while (true) {…}`
  - Embedded Software: System muss auf ***Interrupts*** reagieren können
  - Verarbeitet kontinuierlich Ereignisse (aus der Umgebung)
  - Server/Client: Kann über besonderes Ereignis beendet werden
  - Beispiel: Event-Dispatch-Thread in Java-Swing

Einfache reaktive Systeme =
Mealy-Maschine?

Automat mit Ausgabe: Mealy oder Moore-Maschine

- Außenwelt hat ***Ereignisse*** (z.B. elektrische Signale wie Tastendruck, eine Nachricht über ein Netzwerk, …) diese sind das ***Eingabealphabet Σ des Automaten***
- Die Maschine produziert eine ***Ausgabe*** (z.B. Elektrische Signale wie einen High-Pegel an einem Ausgang, oder eine Nachricht, die über Netzwerk gesendet wird), dies ist das ***Ausgabealphabet Ω*** des Automaten
- Die Maschine hat interne ***Zustände*** (z.B. einen Speicher) dieser wird dargestellt über die ***Menge der Zustände Q***
- Das Verhalten der Maschine wird dargestellt über zwei Funktionen:
  - Übergangsfunktion $\delta: Q \times \Sigma \rightarrow Q$
  - Ausgabefunktion $\lambda: Q \times \Sigma \rightarrow \Omega$
- Einfache Verhaltensmodelle sind damit möglich (für einfache diskrete Steuerungen)

Warum Mealy-Maschinen?

- = ***Vollständige*** Verhaltensbeschreibung
  - Wenn Übergangsfunktion wirklich: **δ: Q x Σ → Q**
  - Simulation möglich
  - Beweise über Eigenschaften möglich („Model Checking“)
- Leicht zu implementieren in verschiedenen Varianten (bei uns mit Tabelle, siehe unten)

Implementierung der Übertragungsfunktion

Implementierungs-Varianten z.B.:

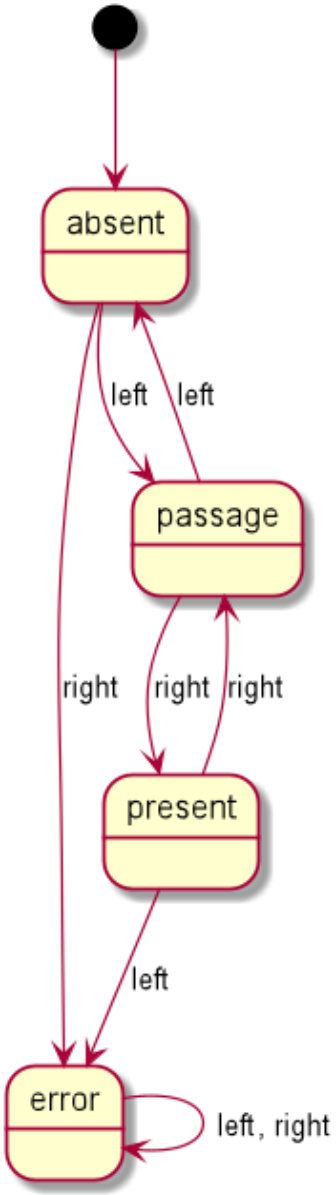
1. Großes ***switch / case*** (für jeden Zustand), dann switch/case für jedes Zeichen des Eingabealphabetes

2. ***Tabelle*** (Eingabealphabet x Zustände), Tabelleneinträge sind dann etwa Ausgabealphabet und Folgezustand

3. Für jeden Zustand eigene Klasse abgeleitet von State, mit jeweils Methode „State nextState (InputSymbol s)“

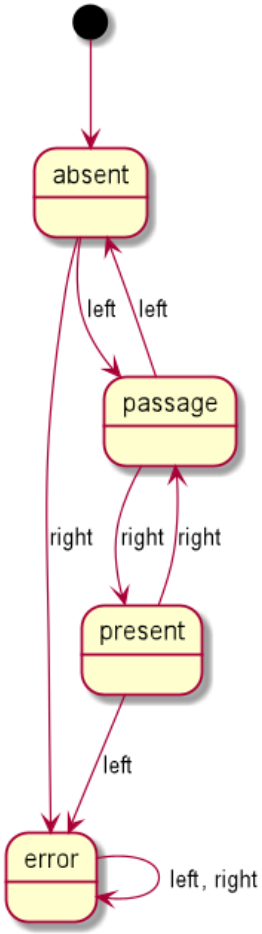
Zutrittskontrollsystem
Automat als Tabelle und als Grafik
(keine Mealy-Maschine)

	ABSENT, PRESENT, PASSAGE, ERROR
LEFT:	PASSAGE, ERROR, ABSENT, ERROR
RIGHT:	ERROR, PASSAGE, PRESENT, ERROR



Für Mealy-Maschine: Zusätzliche Tabelle mit Ausgabesymbolen!

Automaten
Standard Variante: Mit Tabelle
Verwenden wir hier



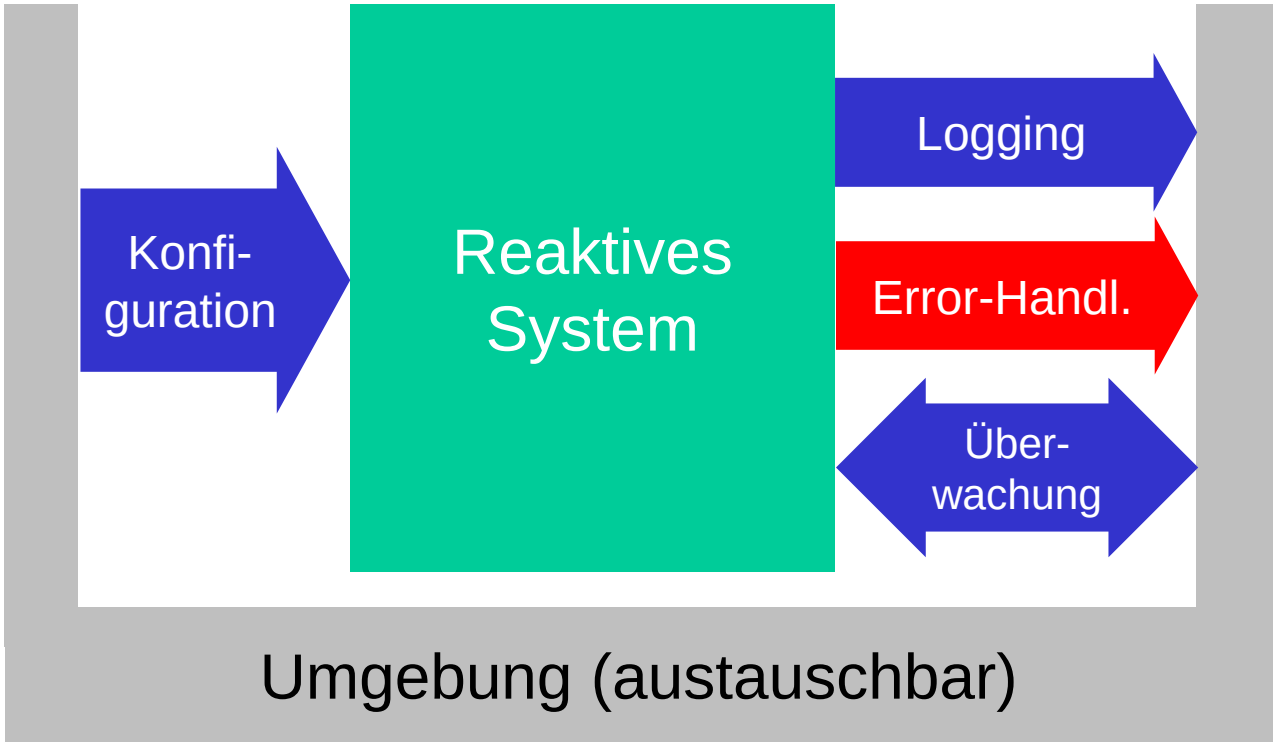

```
public class Empoyee {
    enum State  {ABSENT, PRESENT, PASSAGE, ERROR}
    enum Symbol {LEFT, RIGHT}

    private State[][] transitionTable = {
       {State.PASSAGE, State.ERROR, State.ABSENT, State.ERROR },
       {State. ERROR, State.PASSAGE, State.PRESENT, State.ERROR}
    };
    private State currentState = State.ABSENT;


    public void transition(Symbol symbol) {
        currentState =
             transitionTable [symbol.ordinal()]
                             [currentState.ordinal()];
    }
```

Zustandsüberg. Tabelle

Startzustand/ laufender Zust.

Zustandsüberg. Funktion

Logging und Konfiguration



Logging – Was ist das?

Logging is the process of recording events, with a computer program usually an application software in a certain scope in order to provide an audit trail that can be used ***to understand the activity of the system and to diagnose problems***. [after execution].

[Wikipedia]

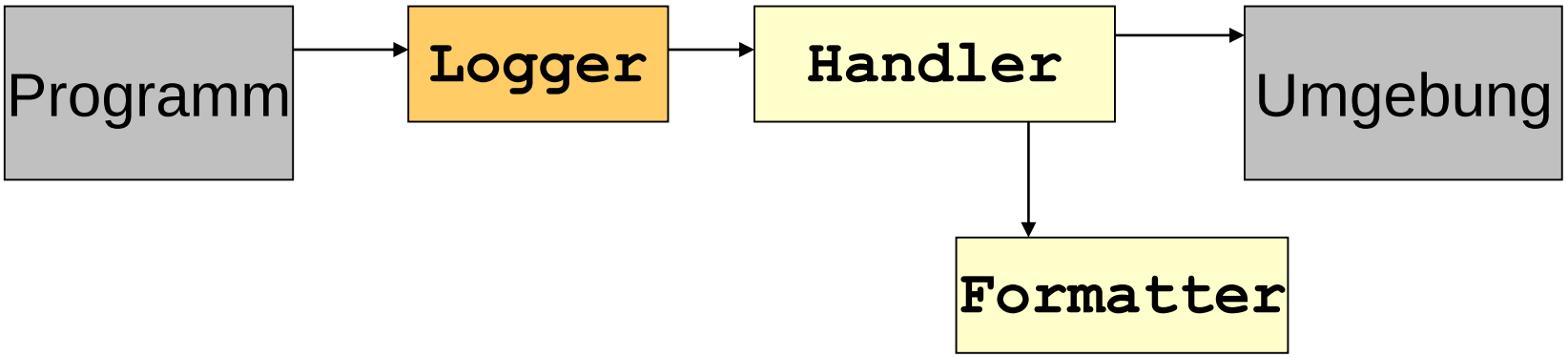
- Betreiber muss wissen,
  - Wie das System konfiguriert ist
  - In welcher Umgebung es läuft
  - Was seit start des Systems passiert ist
  - Ob Fehler aufgetreten sind …

Tracing und Logging: Implementierung

- **Böse**: `System.err.println`(....)
  - Nicht abschaltbar
  - Nicht steuerbar
- Log4J (www.apache.org) mit **`Logger`** – Klasse
  - Verschiedene Level (debug bis fatal)
  - Verschiedene „Kategorien“
  - Konfigurierbar / Steuerbar
- `java.util.logging.Logger` (seit Jdk 1.4)
  - Wie Log4j

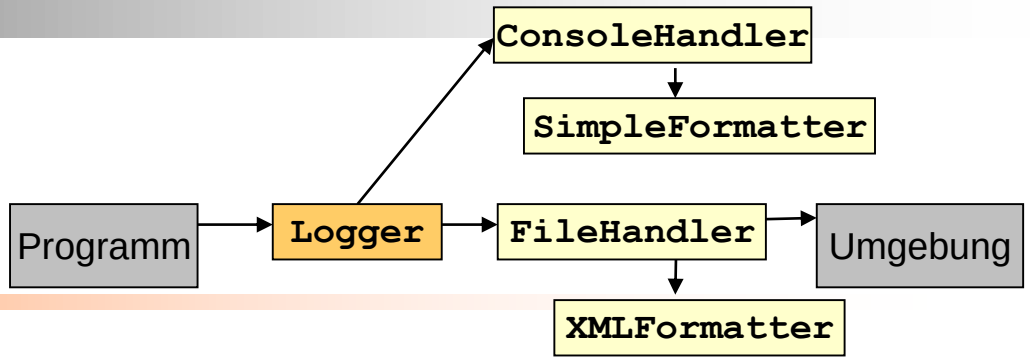
Beispiel: Java Logging Framework

- Seit JDK 1.4 enthalten



- ***Konfigurierbar*** über Handler, Filter und Formatter
  - ***Handler***: `StreamHandler, ConsoleHandler, FileHandler, SocketHandler, MemoryHandler`
  - ***Formatter***: `SimpleFormatter, XMLFormatter`

Java Logging Framework




```
public class LoggerDemo {
   public static void main(String[] args) {
        Logger logger =
                Logger.getLogger("de.thro");
        FileHandler fh =
                new FileHandler("WichtigeInfos.log");
        fh.setFormatter(new SimpleFormatter());


        Logger.getLogger("").addHandler(fh);
        logger.setLevel(Level.ALL);

        logger.severe("Ein Schwerwiegender Fehler");
        logger.warning("Eine Warning");
        logger.info("Eine Info über den Zustand des Programms");
        logger.config("Eine Info über die Konfiguration");
    }
}
```

Twelve Factor App (Heroku)

Die Konfiguration in Umgebungsvariablen ablegen

- Die *Konfiguration* einer App ist alles, was sich wahrscheinlich zwischen den Deploys ändert (Staging, Produktion, Entwicklungsumgebungen, usw.). Dies umfasst:
  - Resource-Handles für Datenbanken und andere unterstützende Dienste
  - Credentials für externe Dienste wie Amazon S3 oder Twitter
  - Direkt vom Deploy abhängige Werte wie der kanonische Hostname für den Deploy
- Manchmal speichern Apps die Konfiguration als Konstanten im Code. Dies ist eine Verletzung der zwölf Faktoren. Sie fordern **strikte Trennung der Konfiguration vom Code**. Die Konfiguration ändert sich deutlich von Deploy zu Deploy, ganz im Gegensatz zu Code.

Optionen für unser System

- Umgebungsvariablen des Betriebssystems (der Shell)

```
Map<String, String> env = System.getenv();
System.getenv("ANT_HOME");
```

- Java Properties (java … -D …)

```
Properties props = System.getProperties();
System.getProperty("file.separator");
```

- Kommandozeilen Parameter
- Konfigurations-Datei
- Idee: Interface **`IConfiguration`** mit allen Parametern, Implementierung über Dependency Injection

Konfiguration über Kommandozeilen Parameter mit Apache commons.cli

Kommandozeilen Parameter: p,u

Hat der Parameter einen zus. Wert?

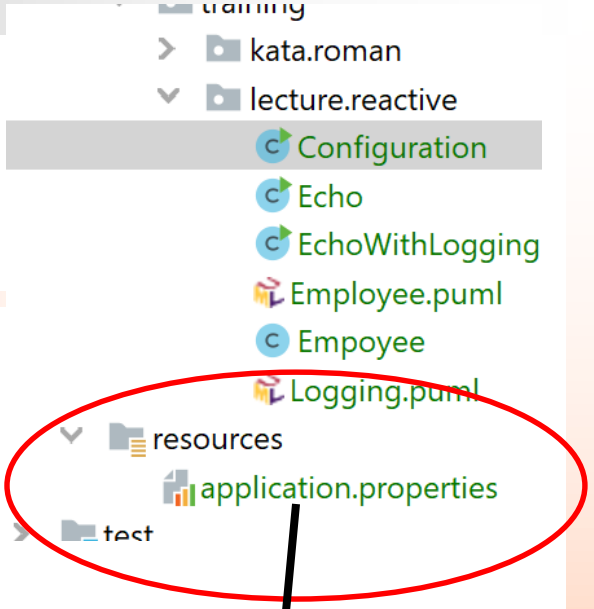
Erklärung

```
Options options = new Options();
options.addOption(new Option("u", "user", true, "username"));
options.addOption(new Option("p", "password", true, "password"));

CommandLineParser parser = new DefaultParser();
CommandLine cmd = parser.parse(options, args);
```

Parameter Werte auslesen

```
System.out.println("User = " + cmd.getOptionValue("u"));
System.out.println("Password = " + cmd.getOptionValue("p"));
```

Properties - Datei



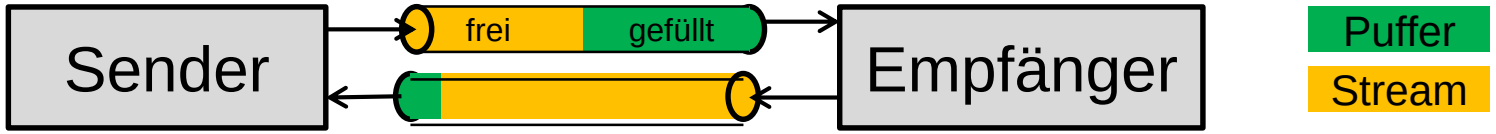

```
Properties properties = new Properties();
InputStream propertyFile = getClass()
        .getClassLoader()
        .getResourceAsStream("application.properties");

properties.load(propertyFile);

System.out.println("User = " +
      properties.get("user.name"));
System.out.println("Password = " +
       properties.get("user.password"));
```

```
user.name = Gerd
user.password = nix
```

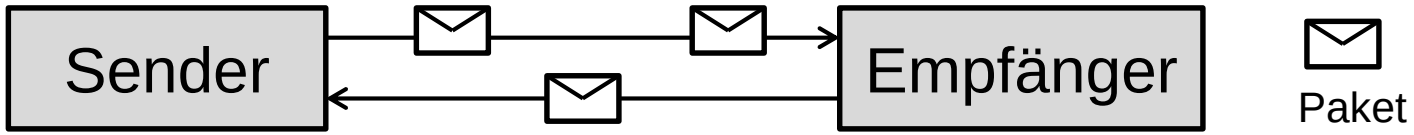
Kommunikation zwischen reaktiven Systemen

Kommunikation zwischen reaktiven Systemen

- = mindestens zwei verschiedene Prozesse (-> vgl. Interprozess-Kommunikation aus BS)
  - Gemeinsamer Speicher (Shared Memory), geht nicht
  - Nachrichtenaustausch (Message Passing)
- Nachrichtenaustausch: Varianten
  - ***Stromorientiert*** (Streams / Pipes)



  - ***Paketorientiert*** (Datagramme / Nachrichten)



Stromorientiert vs. Nachrichtenorientiert

- Stromorientiert bzw. Verbindungsorientiert (wie TCP/IP)
  - ***Feste Verbindung*** zwischen einem Sender und einem Empfänger
  - Unidirektionale / Bidirektionale Ströme, gepuffert / ungepuffert
  - Kommunikation ist seriell (= Strom von Bytes),
    typisch auch für eingebettete Systeme (RS232, USB, …)
  - Zuverlässig, Reihenfolge bleibt erhalten
  - Wie in (alten) Telefonnetzen: GSM, Analoge Modems, …
- Paketorientiert (wie IP bzw. UDP/IP)
  - ***Keine Verbindung***, Möglicherweise „Fire & Forget“
  - Paket wird übertragen bzw. über Netzwerk geroutet
  - Multicast / Broadcast möglich
  - ggf. unzuverlässig, ggf. geht Reihenfolge der Pakete verloren

Stromorientierte Kommunikation
Serielles Lesen und Schreiben von Daten

Idee in vielen Programmiersprachen:
Datenquellen und -ziele einheitlich behandeln

- Datenquelle/Datenziel = ***Strom von Bytes / Zeichen***
- ***Sequenzielles*** Lesen und Schreiben in diese Ströme
- Stream abstrahiert Datei, Hauptspeicher, Konsole, Socket, …
- Je nach Datenquelle/ziel andere ***Stream*** bzw. ***Writer/Reader*** Klasse in Java (Java: im JDK6 > 60 Stream-Klassen!)
- Schön: Filter / Transformation einbaubar