GEDOPLAN

Java Batch – Der Standard für's Stapeln

Berlin Expert Days

18.09.2015

Dirk Weil, GEDOPLAN GmbH

GEDOPLAN

Dirk Weil

- GEDOPLAN GmbH, Bielefeld
  - GEDOPLAN IT Consulting
    Konzeption und
    Realisierung von IT-Lösungen
  - GEDOPLAN IT Training
    Seminare in Berlin, Bielefeld, on-site
- Java EE seit 1998
- Vorträge
- Veröffentlichungen

GEDOPLAN

Batch

- Was?
  - Stapelverarbeitung
  - "Menge von Daten, die vom Computer i. A. ohne Eingriff des Benutzers der Reihe nach abgearbeitet wird" (Wikipedia)
- Wozu?
  - Verlagerung / Optimierung von Ausführungszeiten
  - Ausnutzung freier Ressourcen
  - Verwaltung umfangreicher, wiederholter Tätigkeiten
- Wie?*
  - Großrechner
  - Spring Batch
  - Scheduler mit individueller Programmierung

* bisher

GEDOPLAN

Java Batch

- Batch Applications for the Java platform
- JSR-352

- basiert stark auf Spring Batch
- Bestandteil von Java EE 7
  - WildFly 8: JBeret
  - GlassFish 4: jsr352-RI

- auch in SE-Umgebung nutzbar

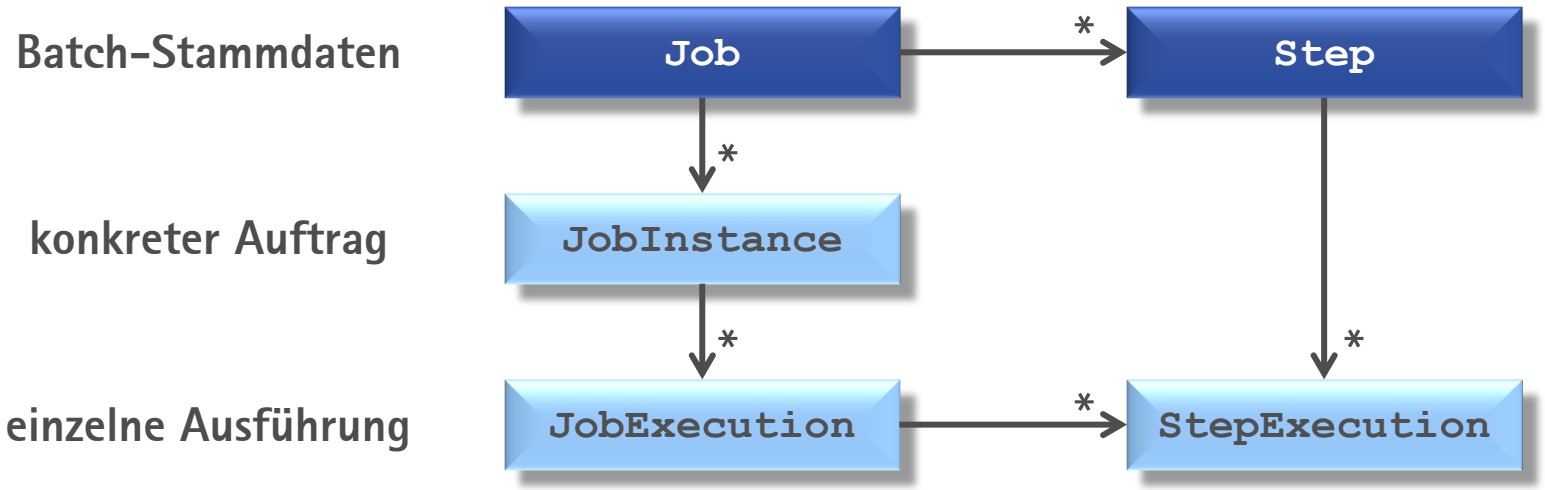
GEDOPLAN

Java Batch

- Features:
  - Task und Chunks
  - Exception Handling
  - Checkpoints
  - Start, Restart, Abort
  - sequenzielle und parallele Ausführung
  - Job Repository
  - Job Control

GEDOPLAN

Job Configuration & Runtime



GEDOPLAN

Task-orientierte Steps

- Batchlet
  - i. A. kleine Aufgabenstellung / kurze Ausführung
  - z. B. Initialisierung, Cleanup

```
@Named
public class HelloWorldBatchlet extends AbstractBatchlet {
  public String process() throws Exception {
    System.out.println("Hello, batch world!");
```

GEDOPLAN

Job Descriptor

- **META-INF/batch-jobs/*jobname*.xml**

```
<job id="demoJob1" >
  <step id="step1">
    <batchlet ref="helloWorldBatchlet" />
  </step>
</job>
```

GEDOPLAN

Job-Start

- `JobOperator` erlaubt Zugriff auf das Job Repository

```
JobOperator jobOperator = BatchRuntime.getJobOperator();

long executionId = jobOperator.start("demoJob1", new Properties());

JobInstance jobInstance = jobOperator.getJobInstance(executionId);
JobExecution jobExecution = jobOperator.getJobExecution(executionId);
```

GEDOPLAN

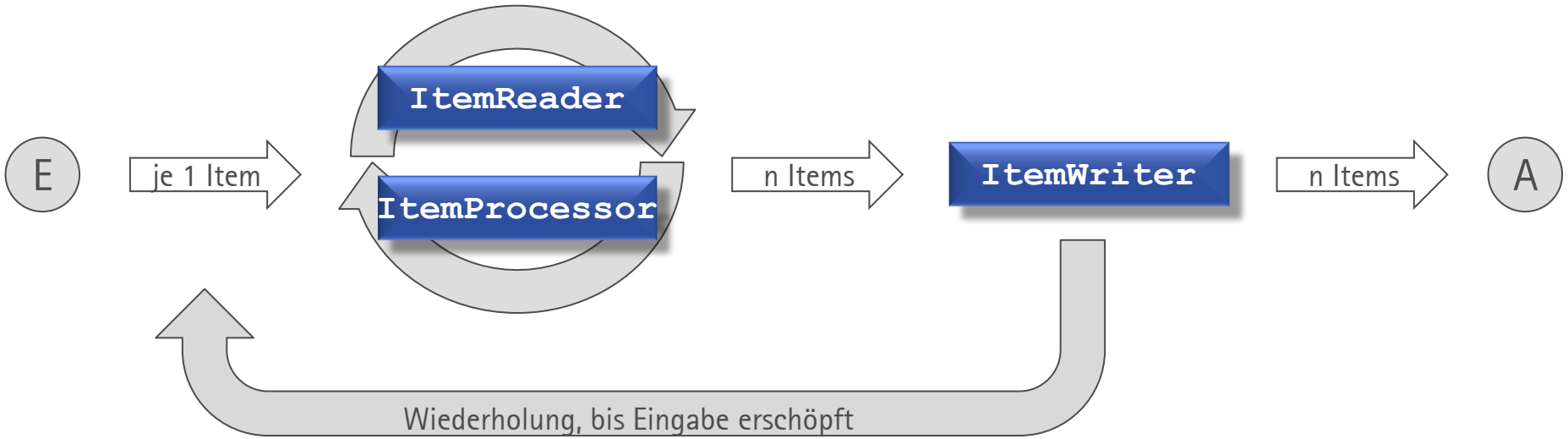
Chunk-orientierte Steps

- ~EVA
  - ItemReader liest Eingabe
  - ItemProcessor verarbeitet Daten
  - ItemWriter schreibt Ausgabe
  - Chunk Size bestimmt Anzahl gemeinsam verarbeiteter Items

GEDOPLAN

Chunk-orientierte Steps

- Chunk Size bestimmt Anzahl gemeinsam verarbeiteter Items



GEDOPLAN

Job Descriptor

- **META-INF/batch-jobs/*jobname*.xml**

```
<job id="demoJob3" >
 version="1.0">
  <step id="init" next="import">
    <batchlet ref="countryCleanBatchlet" />
  </step>
  <step id="import">
    <chunk item-count="3">
      <reader ref="countryItemReader" />
      <processor ref="countryItemProcessor" />
      <writer ref="countryItemWriter" />
    </chunk>
  </step>
</job>
```

GEDOPLAN

Item Reader

```
@Named
public class CountryItemReader extends AbstractItemReader {

  private BufferedReader reader;

  public void open(Serializable checkpoint) throws IOException {
    this.reader = …;
  }

  public Object readItem() throws IOException {
    return this.reader.readLine();
  }
```

GEDOPLAN

Item Processor

```
@Named
public class CountryItemProcessor implements ItemProcessor {

  public Object processItem(Object item) throws Exception {
    String[] attributes = ((String) item).split("\\t");
    String isoCode = attributes[0];
    String name = empty2Null(attributes[1];

    Country country = new Country(isoCode, name, …);
    return country;
  }
```

GEDOPLAN

Item Writer

```
@Named
public class CountryItemWriter extends AbstractItemWriter {

  @Inject
  CountryRepository countryRepository;

  @Override
  public void writeItems(List<Object> items) {
    for (Object item : items) {
      this.countryRepository.persist((Country) item);
    }
  }
```

GEDOPLAN

Job und Step Status

- Step endet mit Status
  - explizit gesetzt
  - durch Exception

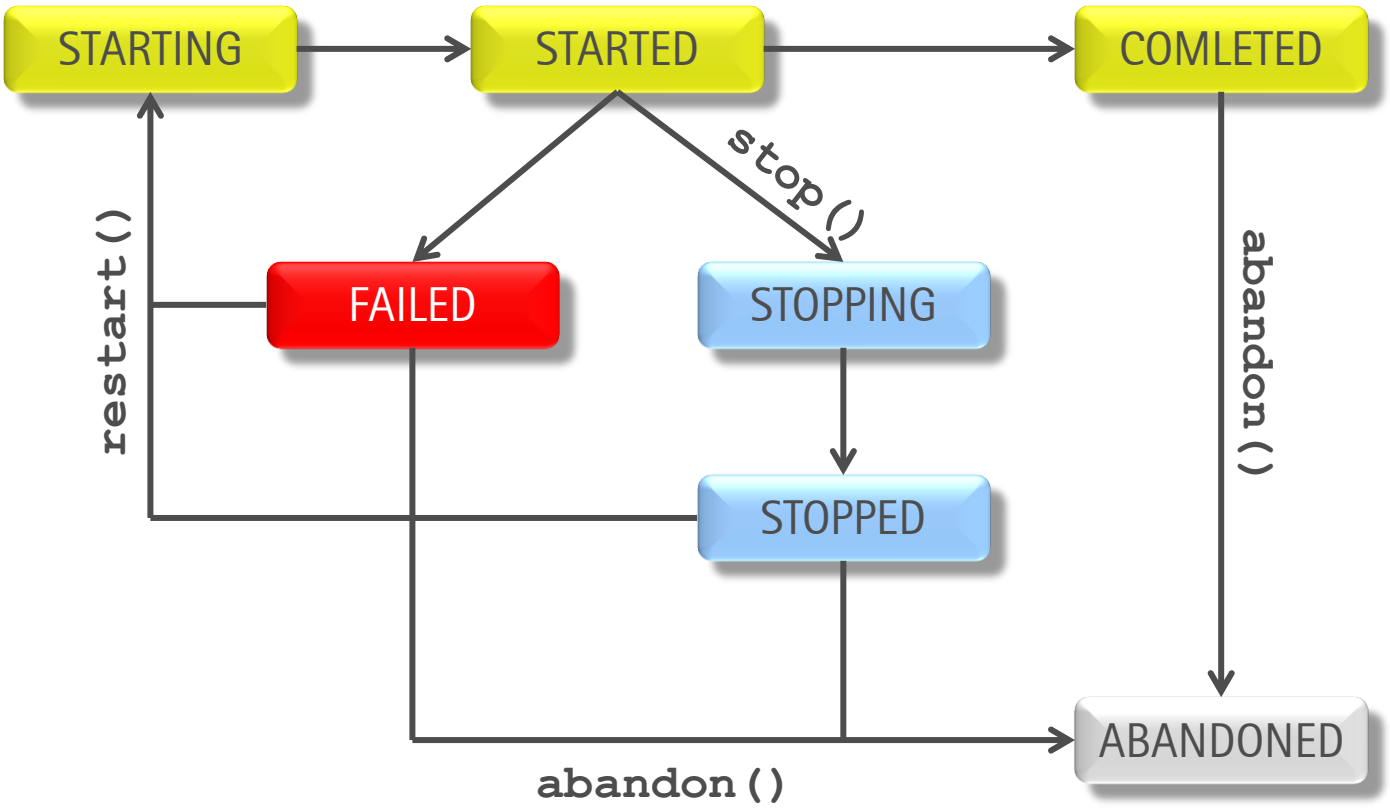
```
ABANDONED
COMPLETED
FAILED
STARTED
STARTING
STOPPED
STOPPING
```

- erzeugt ebenfalls Batch Status

- zusätzlicher String kann als Exit Status erzeugt werden

GEDOPLAN

Job Control



GEDOPLAN

Exception Handling

- Exceptions führen per Default zu FAILED
- Andere Behandlung pro Chunk:
  - Retryable Exceptions führen zur Wiederholung
  - Skippable Exceptions werden ignoriert

```
<step id="import">
  <chunk item-count="3" retry-limit="2">
    <reader ref="countryItemReader" />
    <processor ref="countryItemProcessor" />
    <writer ref="countryItemWriter" />
    <retryable-exception-classes>
      <include class="de.gedoplan.….SomeException" />
    </retryable-exception-classes>
```

GEDOPLAN

Checkpoints

- Item Reader und Writer können Checkpoints liefern
- beliebiges Objekt (→ frei programmierbar)
- letzter Checkpoint wird bei Restart wieder angeliefert
- ermöglicht Wiederaufsetzen nach Fehler

- Reader und Writer haben unabhängige Checkpoints
- häufig: Zusammenführung in Checkpoint Algorithm
- kann/sollte transaktionales Verhalten unterstützen

GEDOPLAN

Step-Reihenfolge

- Job beginnt mit erstem Step
- Jeder Step kann Folge-Steps deklarieren
  - unterschiedliche je nach Status

```
<job …>
  <step id="first" next="second">
    <batchlet …/>
  </step>

  <step id="second">
    <batchlet …/>
    <next on="Batchlet failure" to="third"/>
  </step>

  <step id="third">
```

GEDOPLAN

Step-Reihenfolge

- Decision, Transition ≈ `switch`, `case`

```
<job …>
  <step … next="decider">
    …
  </step>

  <decision id="decider" ref="SomeDecider" />
    <next on="good" to="…" />
    <fail on="bad" exit-status="Mist!" />
    <end on="finished" exit-status="Fertig!" />
```

```
@Named
public class SomeDecider implements Decider {
  public String decide(StepExecution[] executions) {
    … return … ? "good" : "bad";
```

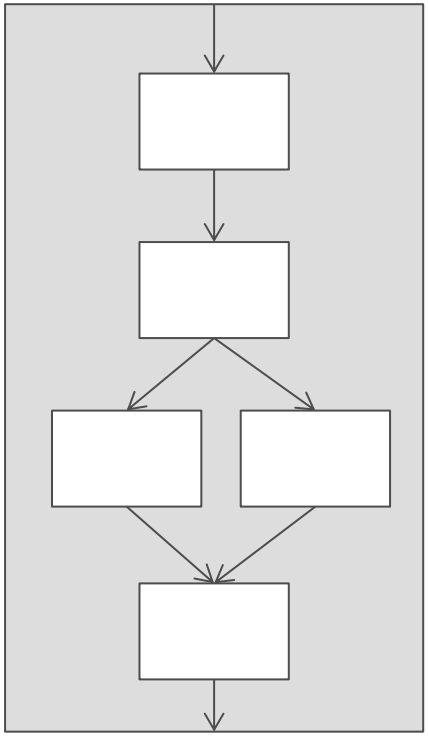
(auch für Flows und Splits)

GEDOPLAN

Flows

- Folge von Steps *

```
<flow id="…" next="…">
  <step id="…" …/>
  <step id="…" …/>
```



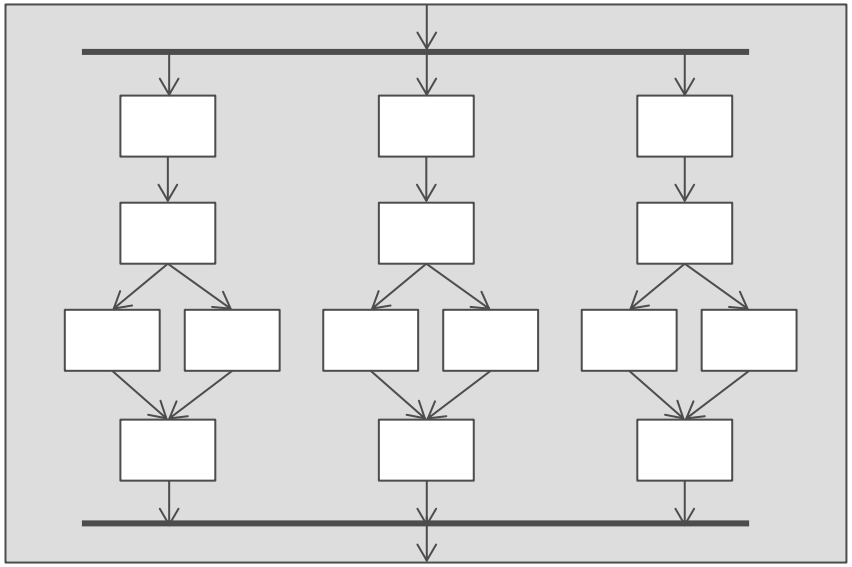
* und Splits, Flows

GEDOPLAN

Parallelisierung

- Split = Mehrere Flows in unterschiedlichen Threads

```
<split id="…" next="…">
  <flow id="…" …/>
  <flow id="…" …/>
  <flow id="…" …/>
```



GEDOPLAN

Parallelisierung

- Partition = Aufteilung der Items eines Steps auf mehrere Threads
  - nach Eingabequellen, Segmenten, Bereichen, ...
  - statisch (im XML), dynamisch (mit PartitionMapper)

```
<job …>
  <step …>
    …
    <partition>
      <plan partitions="3">
        <properties partition="0">
          <property name="firstItem" value="0" />
        </properties>
        <properties partition="1">
          <property name="firstItem" value="100" />
        </properties>
```

GEDOPLAN

Weitere Features

- Properties
  Parameter auf Job- und Step-Ebene

- Listener
  zusätzlicher Code (~Interceptor) auf Job-, Step- oder Step-Teil-Ebene

GEDOPLAN

Nachteile / Merkwürdigkeiten

- Schwach getypte Schnittstellen

```
public interface ItemReader {
  public Object readItem() throws Exception;
public interface ItemProcessor {
  public Object processItem(Object item) throws Exception;
public interface ItemWriter {
  public void writeItems(List<Object> items) throws Exception;
```

```
public interface ItemReader<I> {
  public I readItem() throws Exception;
public interface ItemProcessor<I,O> {
  public O processItem(I item) throws Exception;
public interface ItemWriter<O> {
  public void writeItems(List<O> items) throws Exception;
```

schön wär's ...

GEDOPLAN

Nachteile / Merkwürdigkeiten

- API der Batch Runtime ID-orientiert, nicht Objekt-orientiert
- Standard enthält keine Implementierungen für Standardaufgaben
- Keine Vorgaben / Konfigurationsmöglichkeiten bzgl. Threads
- Keine Vorgaben bzgl. Clustering

GEDOPLAN

Fazit

- JSR 352 erfindet das Rad nicht neu
  - viele De-facto-Standards eingearbeitet
  - enge Anlehnung an Spring Batch
- "endlich Batch-Verarbeitung im EE-Kontext"
- Trotz guter erster Version: Verbesserungen sind an vielen Stellen möglich

GEDOPLAN

More

- http://www.gedoplan-it-training.de
  Seminare in Berlin, Bielefeld, Inhouse
- http://www.gedoplan-it-consulting.de
  Reviews, Coaching, ...

- http://javaeeblog.wordpress.com/
- http://expertenkreisjava.blogspot.de/

- dirk.weil@gedoplan.de
- @dirkweil