

BATCH API

INHALTSVERZEICHNIS

1. Was ist Batch API?

- Warum ist Batch API wichtig?
- Anwendungsfälle und Beispiele aus der Praxis

2. Übersicht der Batch Architektur

- Was ist ein Job?
- Was ist ein Step?
- Was ist ein Job Operator?
- Was ist ein Job Repository?
- Was ist ein Item Reader?

INHALTSVERZEICHNIS

2. Übersicht der Batch Architektur

- Was ist ein Item Processor?
- Was ist ein Item Writer?
- Workflow in Batch Processing

3. Batch Job Design

- Herangehensweise zur Erstellung eines Batch Jobs
- Definition von Inputs, Verarbeitungsschritten und Outputs
- Identifikation der Anforderungen und Ziele eines Batch Jobs

INHALTSVERZEICHNIS

4. Batch Job Scheduling und Ausführung

- Scheduling Strategien
- Übergangselemente
- Verarbeitungsmodelle

5. Job Specification Language (=JSL)

- Job Syntax
- Step Syntax

INHALTSVERZEICHNIS

6. Interfaces

- BatchContexts
- Chunk Interfaces
- Batchlet

7. WildFly Server mit CDI Beispiel

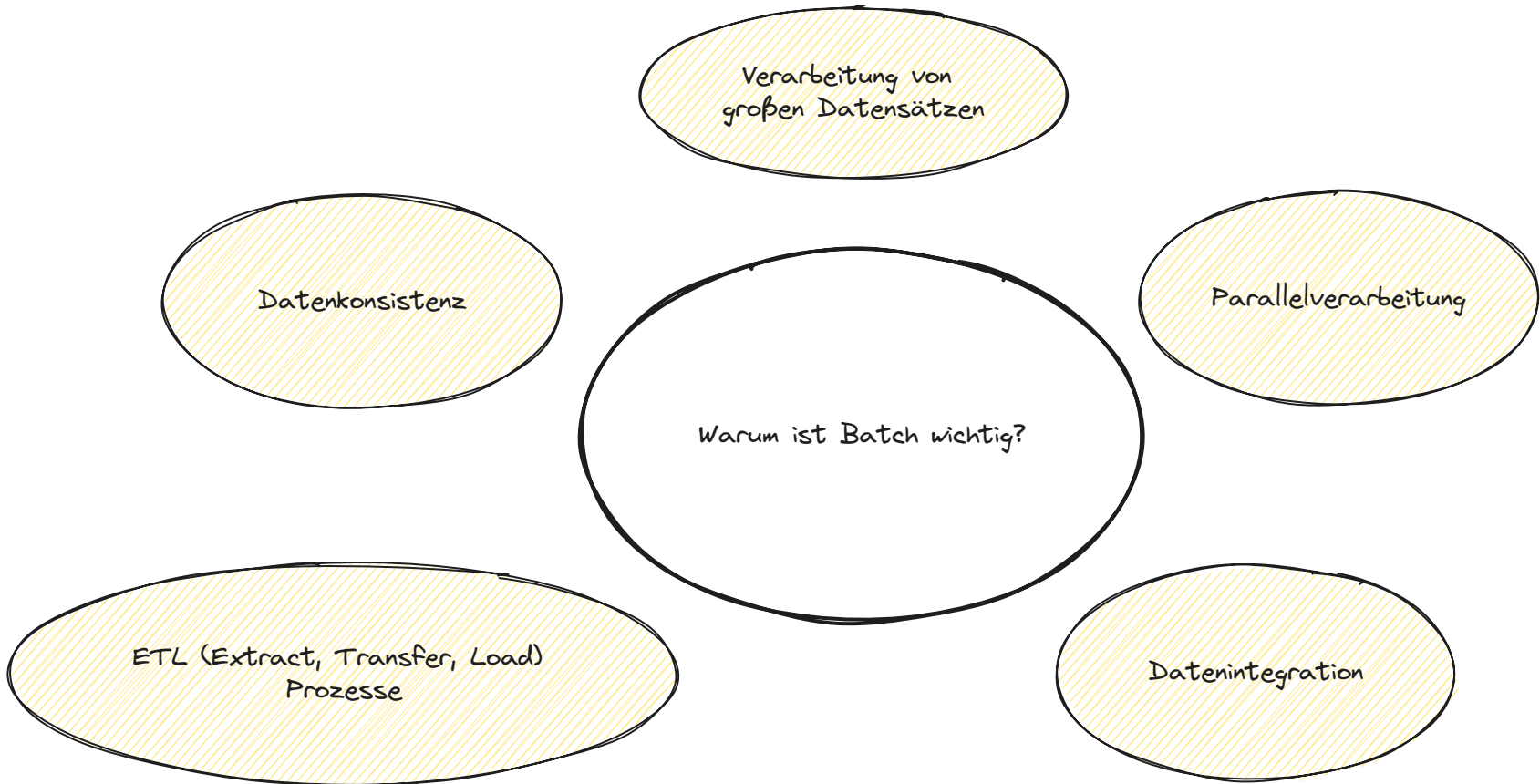
1. WAS IST DIE BATCH API?

Batch API ist eine Software Lösung, die das Verarbeiten von großen Datenmengen in sogenannten "Chunks" oder "Batches" ermöglicht.

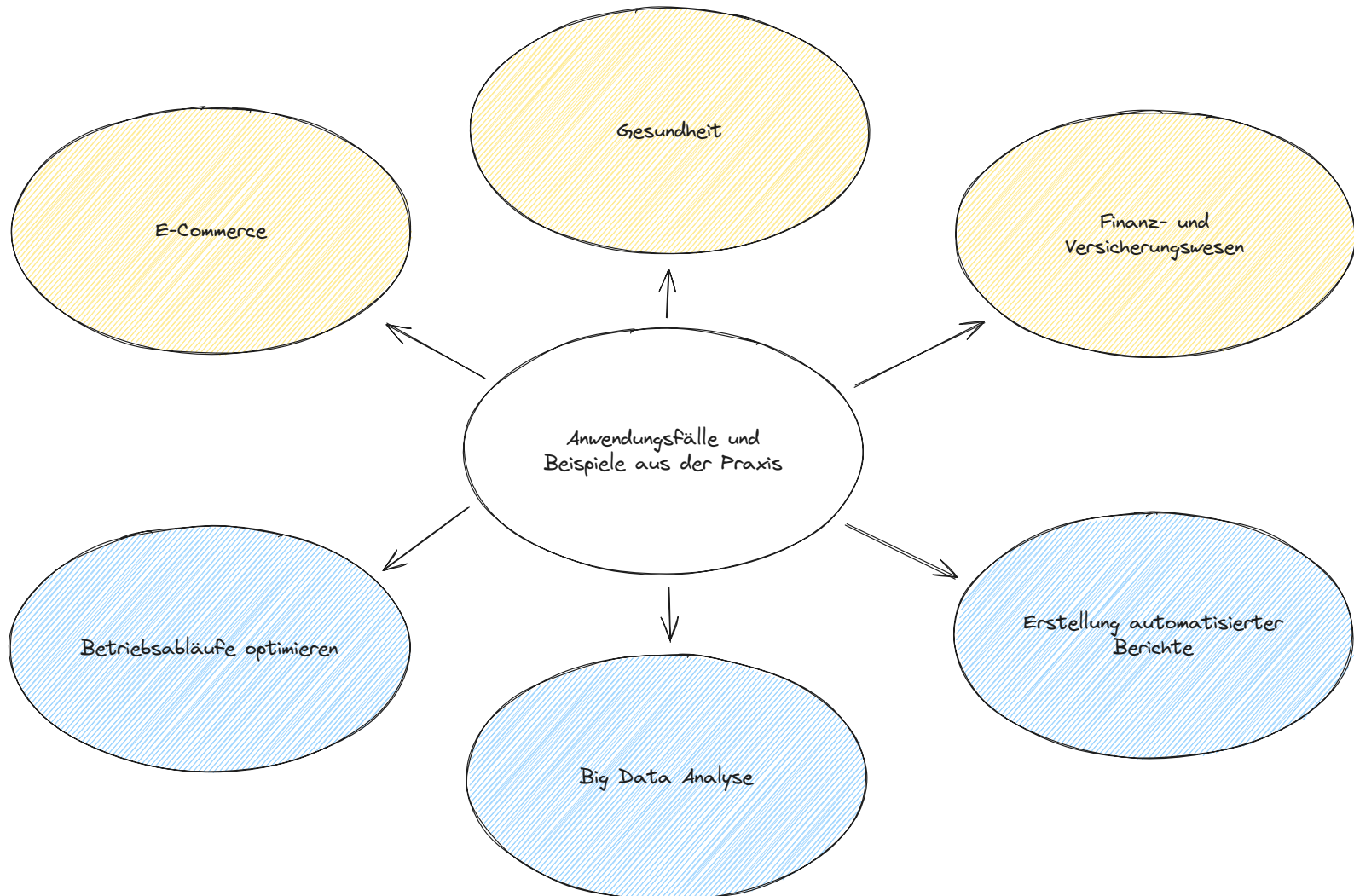
1. WAS IST DIE BATCH API?

- Jakarta EE, damals Java EE
- Jakarta Batch wird in Implementierungen wie JBeret von Wildfly, Batchee und Spring Batch genutzt

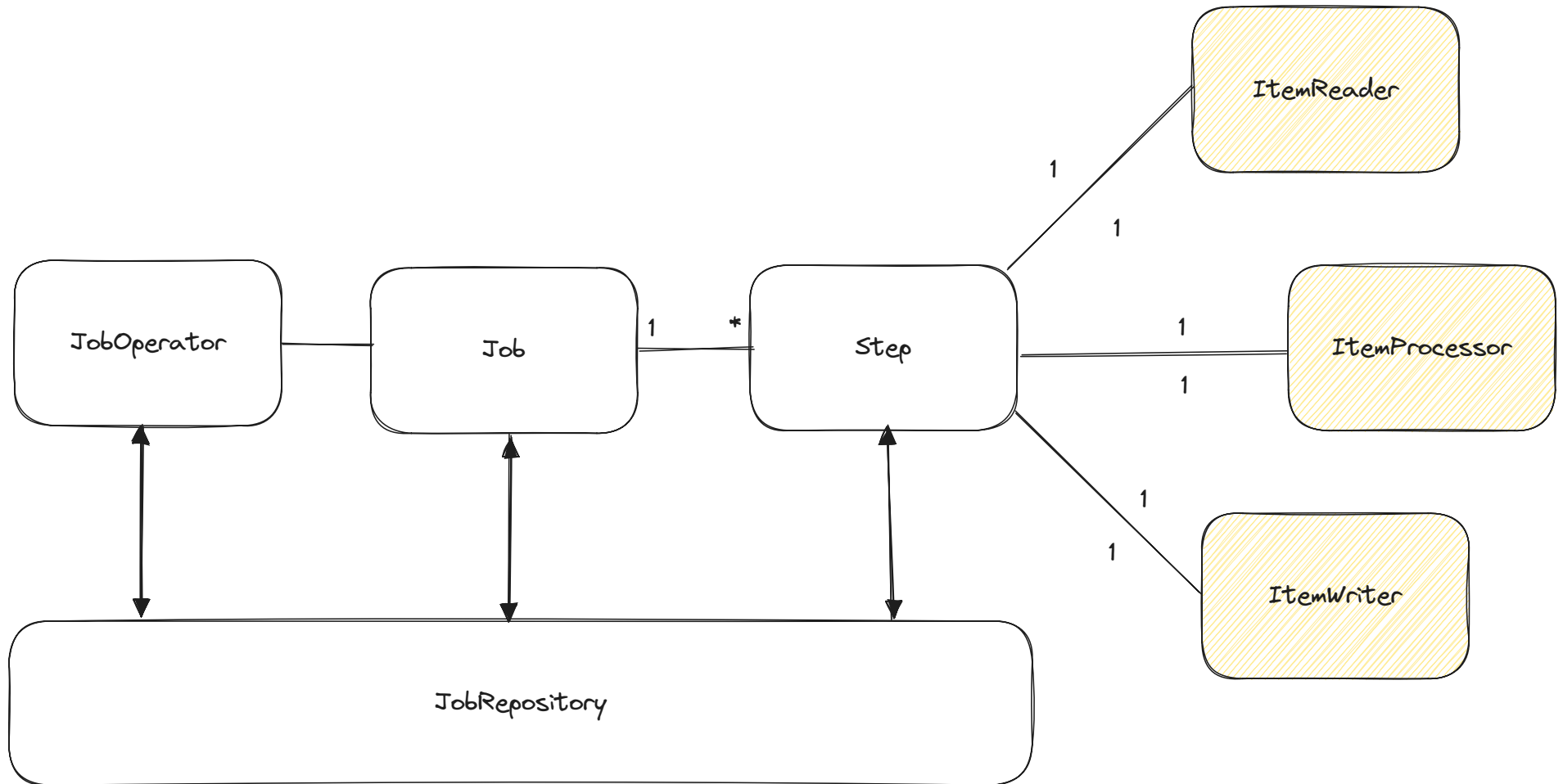
1.1 WARUM IST BATCH API WICHTIG?



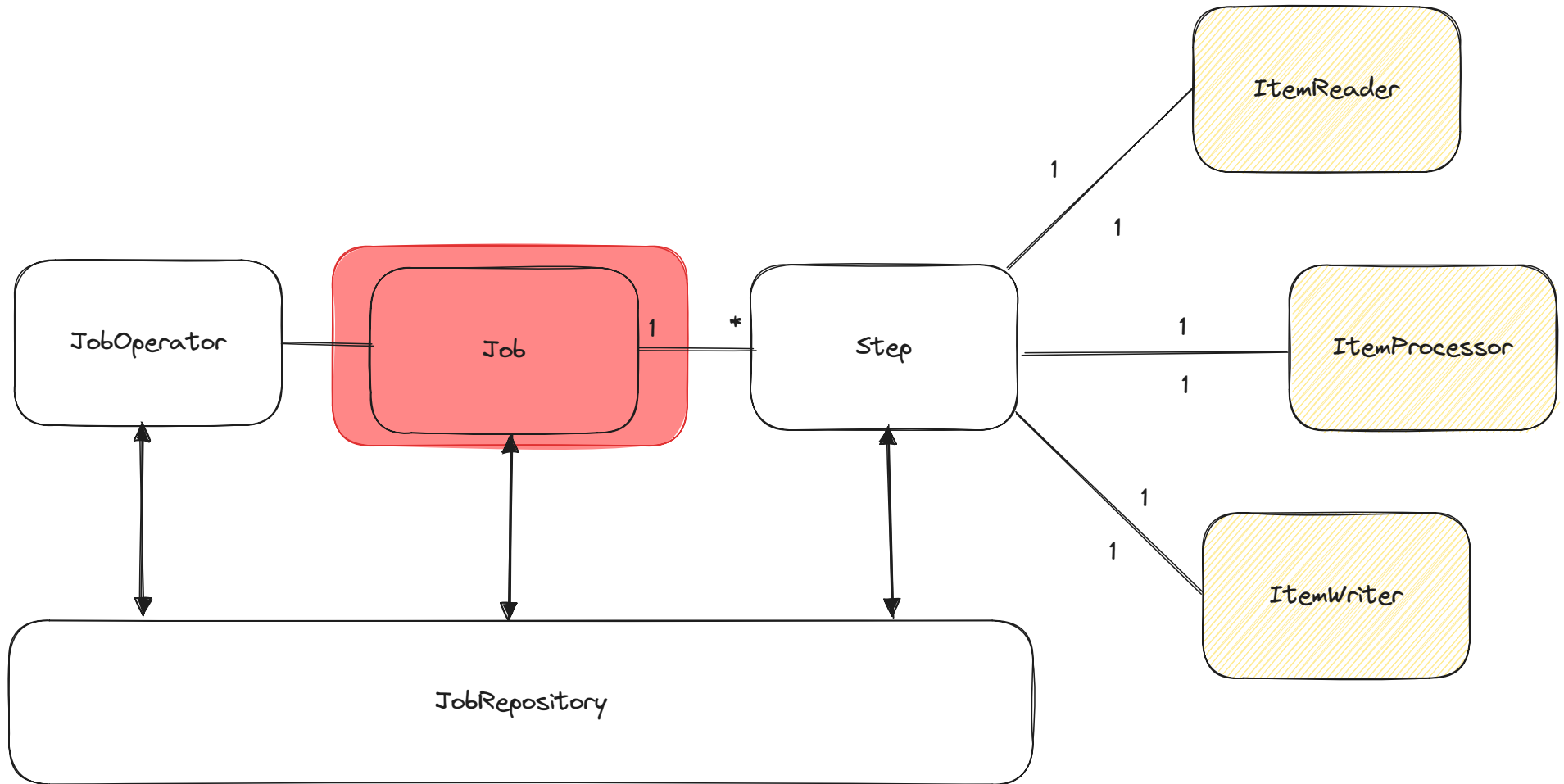
1.2. ANWENDUNGSFÄLLE UND BEISPIELE AUS DER PRAXIS



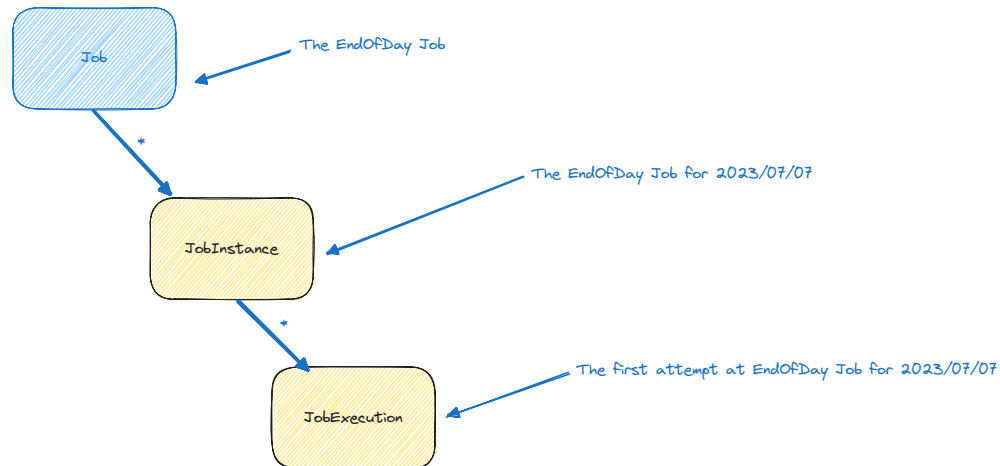
2. ÜBERSICHT DER BATCH ARCHITEKTUR



2.1. WAS IST EIN JOB?

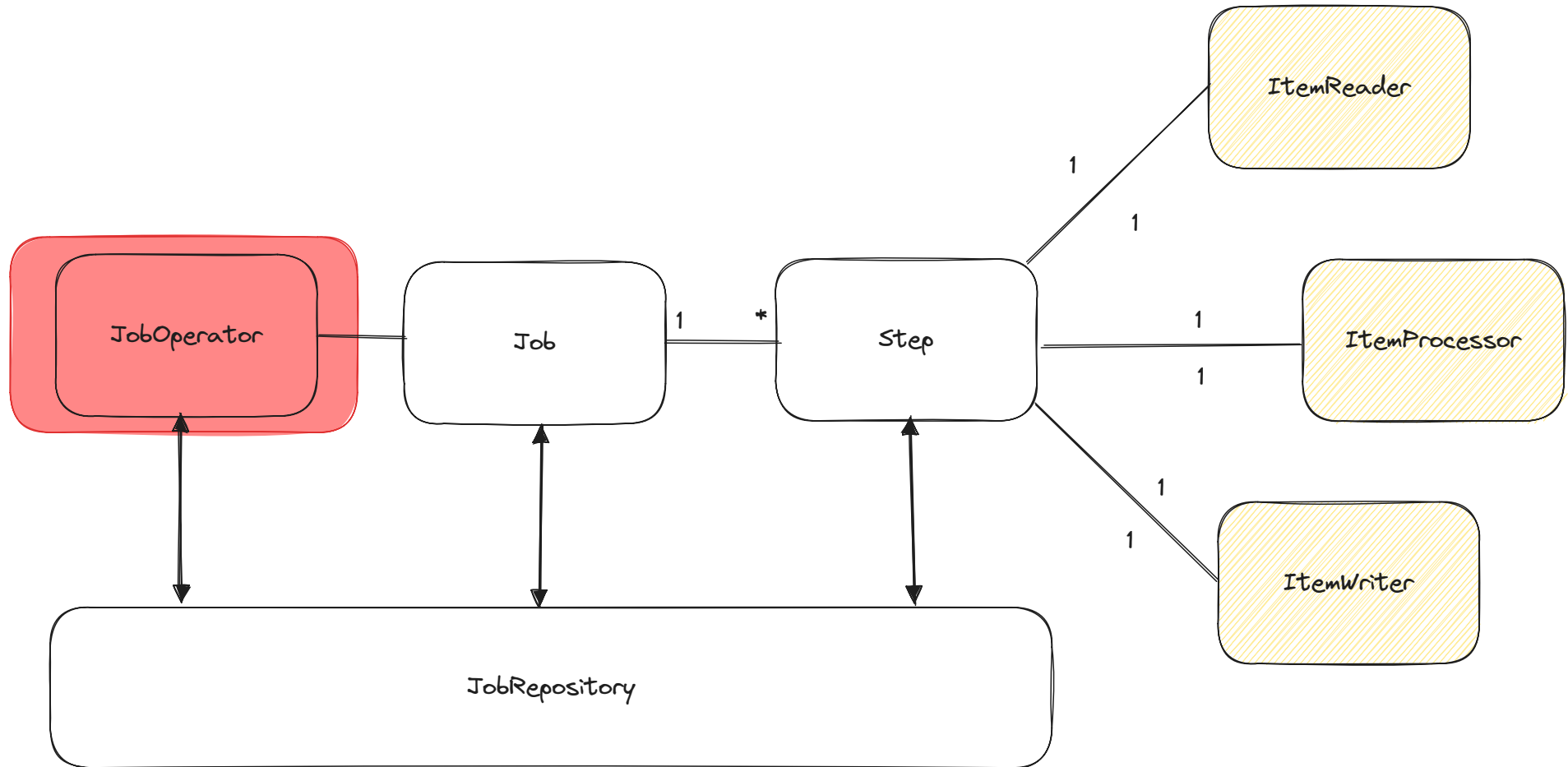


2.1. WAS IST EIN JOB?



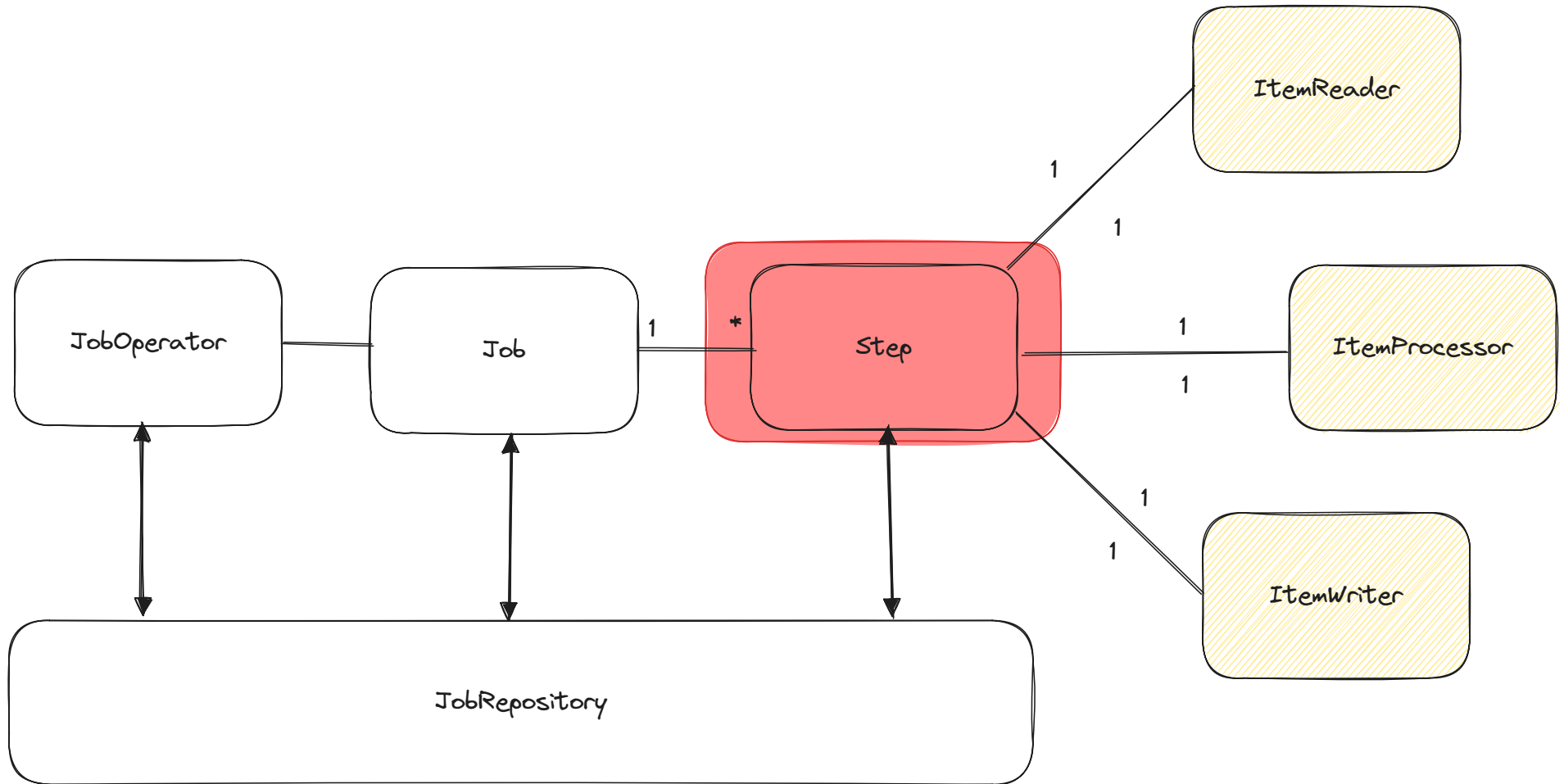
- ein Job besteht aus:
 - Job Instance: ist ein Identifikator, um mehrere Jobsauführungen mit verschiedenen Parametern zu verwalten
 - Job Execution: ein einzelner Versuch einen Job auszuführen

2.2. WAS IST EIN JOB OPERATOR?

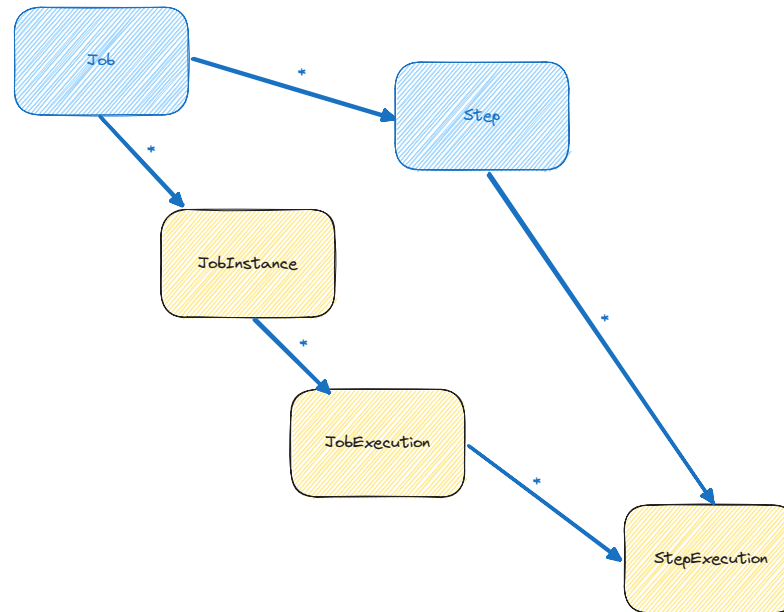


- ist ein Interface und bietet die Schnittstelle zur Verwaltung der Jobs

2.3. WAS IST EIN STEP?



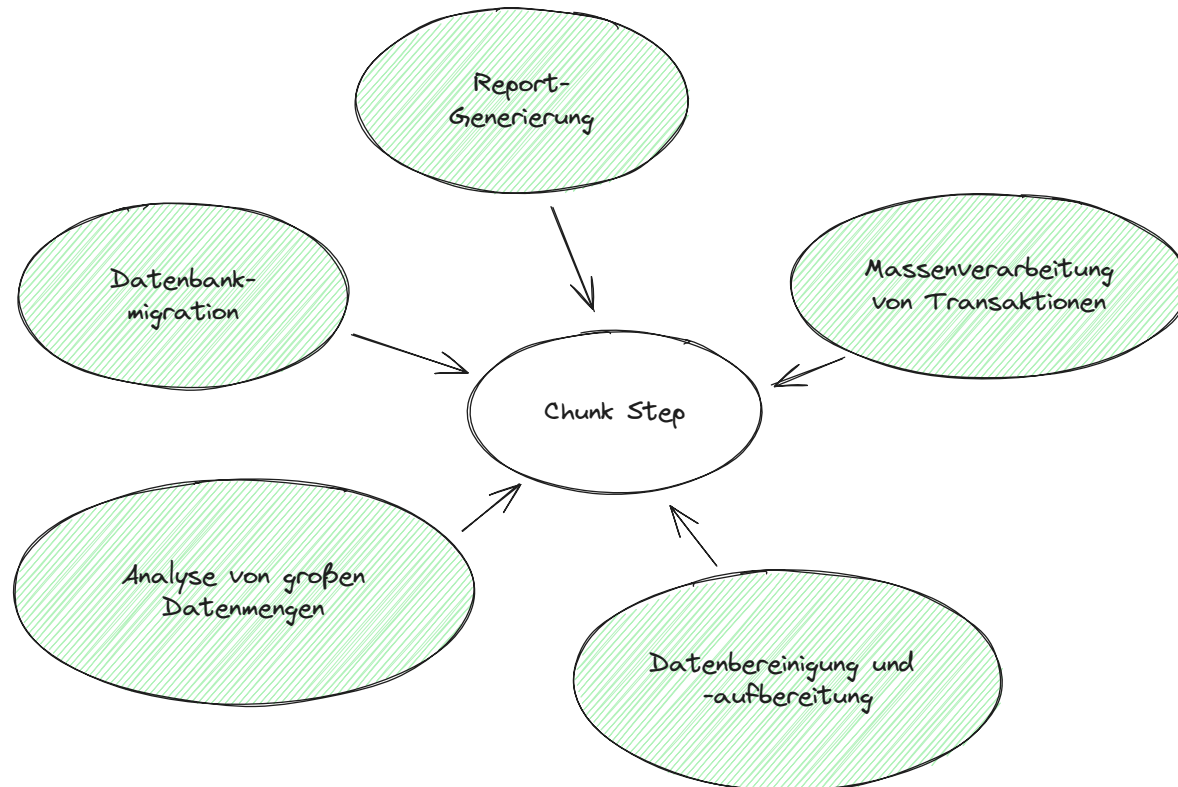
2.3. WAS IST EIN STEP?



- repräsentiert eine einzelne Arbeitseinheit in einem Batch Job
- ein Step besteht aus:
 - Step Execution: ein einzelner Versuch einen Step auszuführen

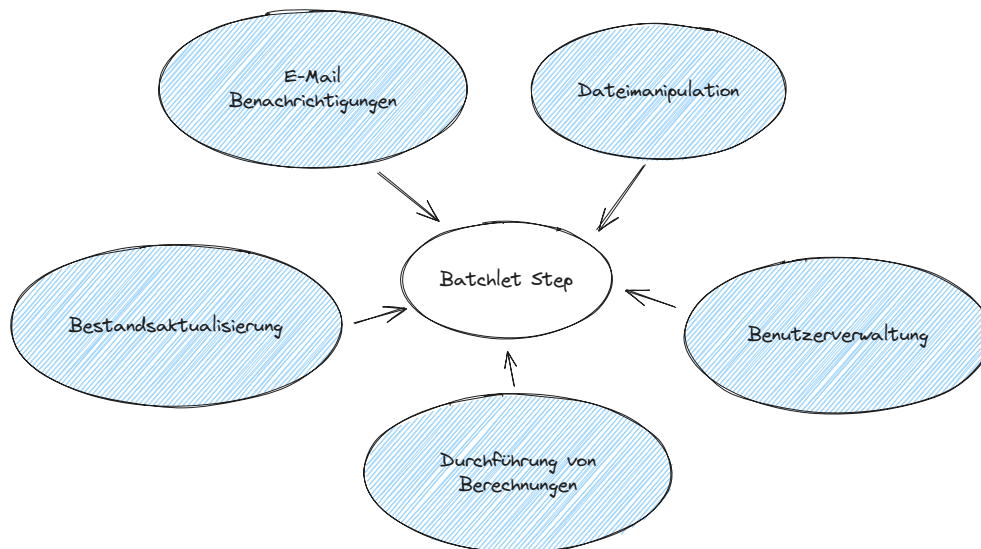
2.3.1 CHUNK STEP

- wird genutzt, um große Datenmengen zu lesen, verarbeiten und in einen Datenspeicher zu schreiben



2.3.2 BATCHLET STEP

- wird genutzt, um einfache, kurzlebige und nicht-transaktionelle Aufgaben und Operationen durchzuführen
- ist eine eigenständige Verarbeitungseinheit



BEISPIEL: BATCHLET

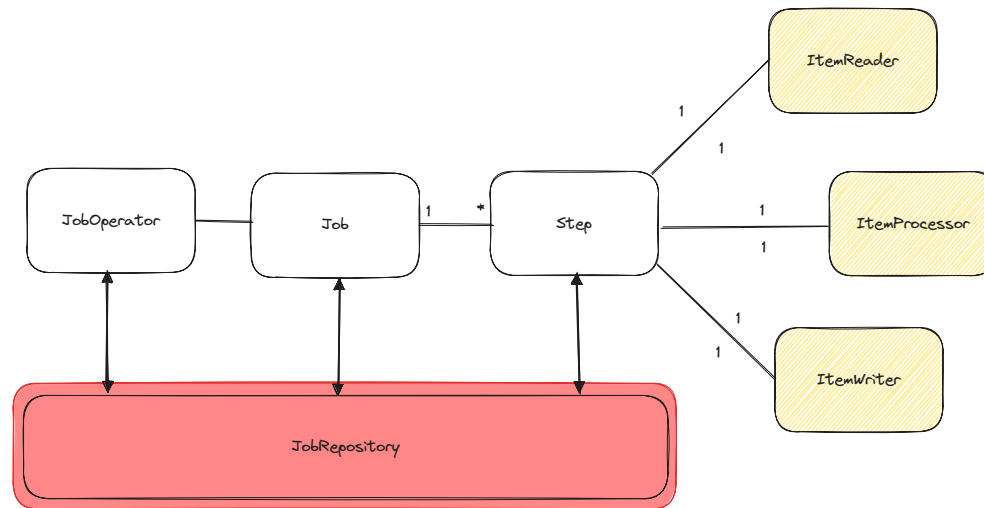
BATCHLET AUFGABE:

- Versuchen Sie die neue CSV mit den Kundendaten in die H2 Datenbank zu laden

VORGABEN:

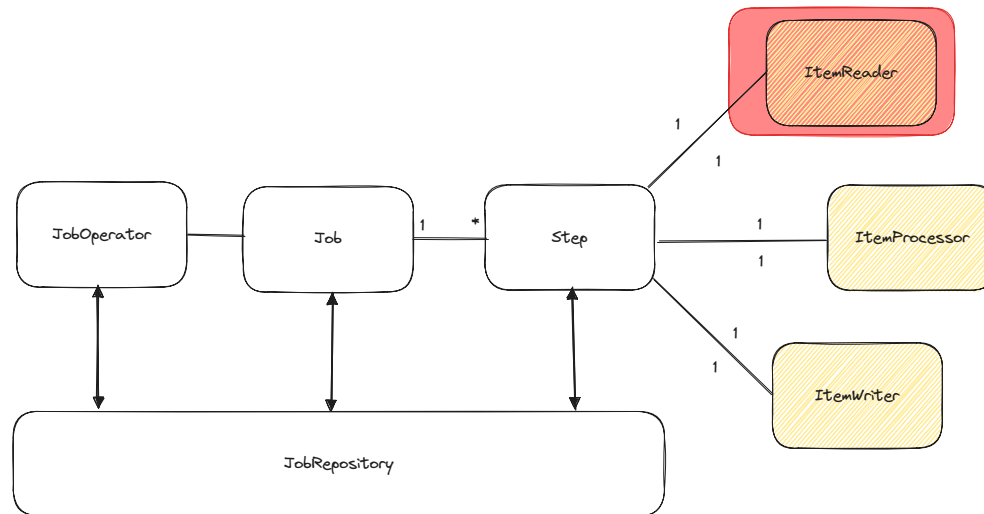
- h2 Datenbank starten:
 - .m2 repository finden
 - in den Ordner "repository" navigieren
 - ins Verzeichnis /com/h2database/h2/1.4.200 gehen
 - java -jar h2-1.4.200.jar ausführen
- um auf die h2 Datenbank zuzugreifen, muss man unter localhost:8080 folgende URL benutzen:
jdbc:h2:~/databases/customers

2.4 WAS IST EIN JOB REPOSITORY?



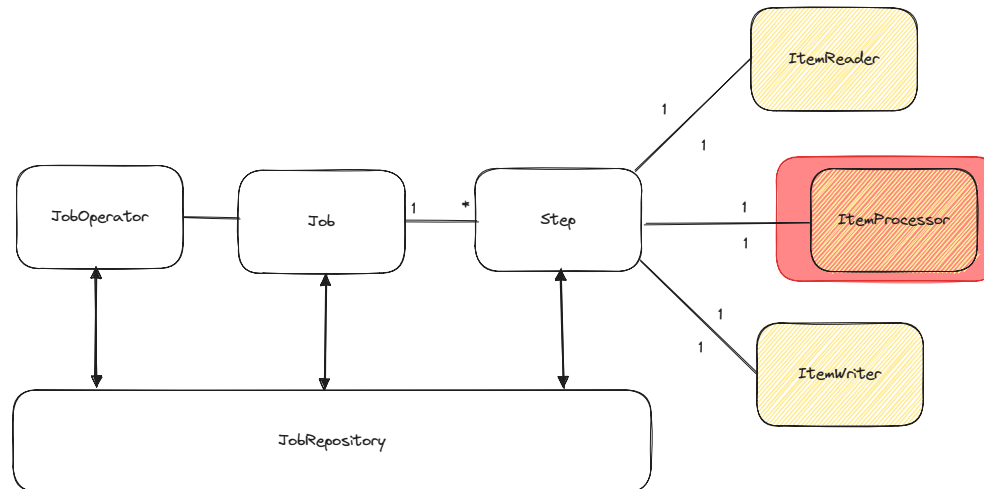
- beinhaltet die Informationen über die Jobs, die derzeit laufen oder gelaufen sind wie beispielsweise Job und Step Execution Historie oder Zustandsverfolgung

2.5. WAS IST EIN ITEM READER?



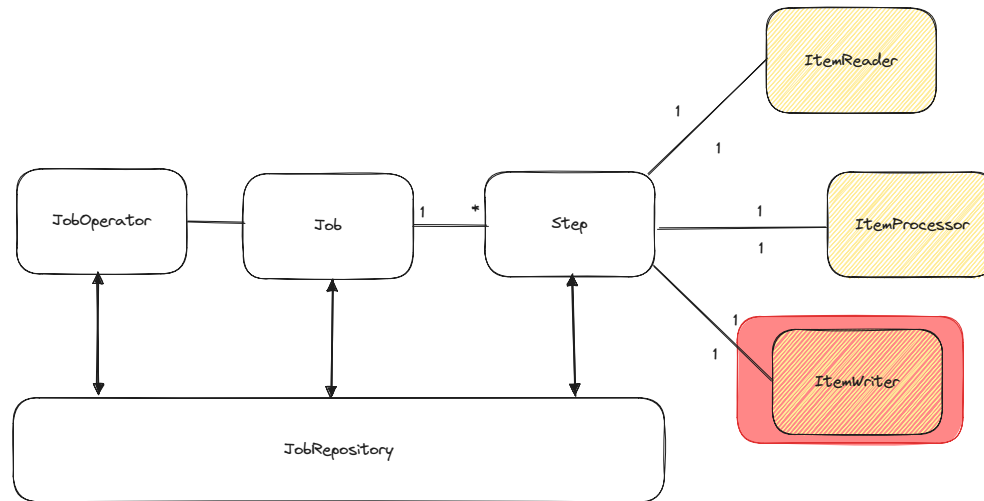
- ist dafür verantwortlich Daten aus Datenquellen wie z.B. einer Datei, Datenbank oder einer Message Queue zu lesen

2.6. WAS IST EIN ITEM PROCESSOR?



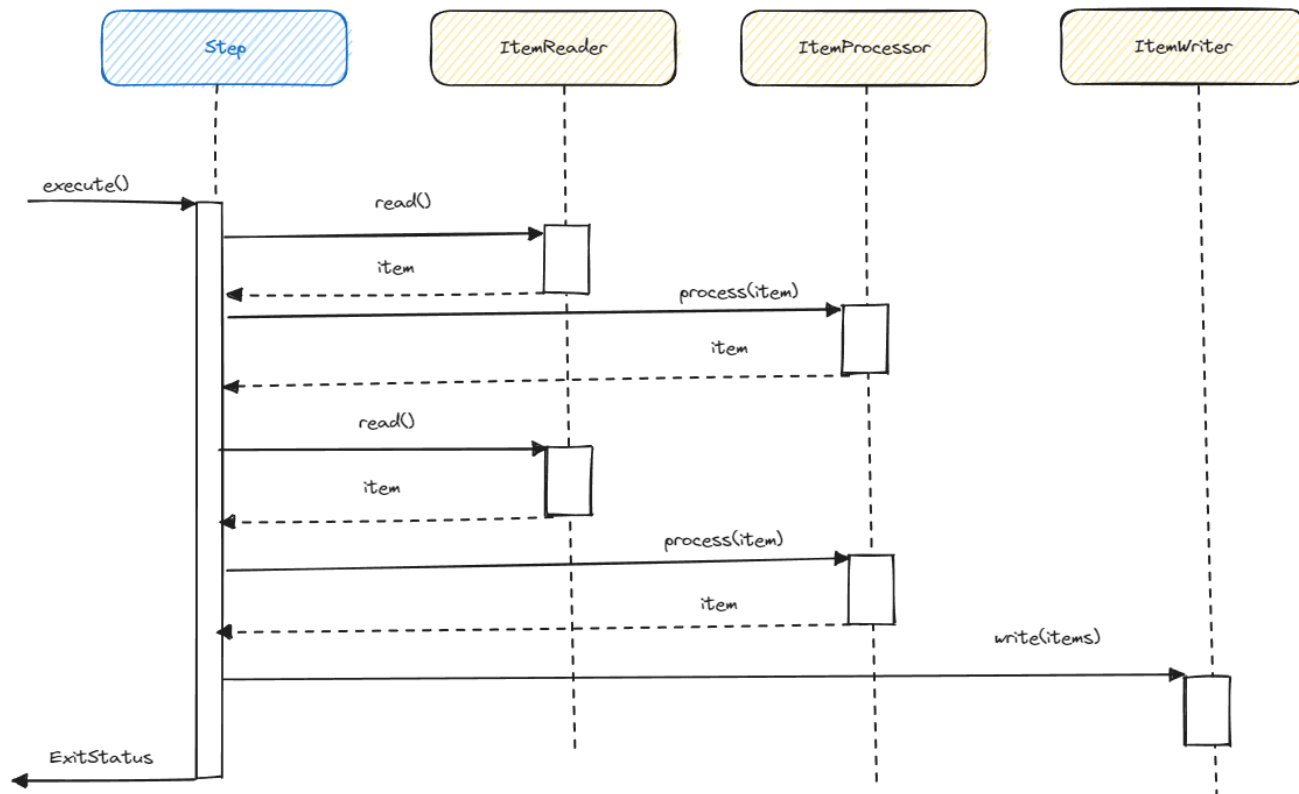
- ist eine Komponente, die die Daten des Item Readers Element für Element verarbeitet und an den Item Writer übergibt

2.7. WAS IST EIN ITEM WRITER?

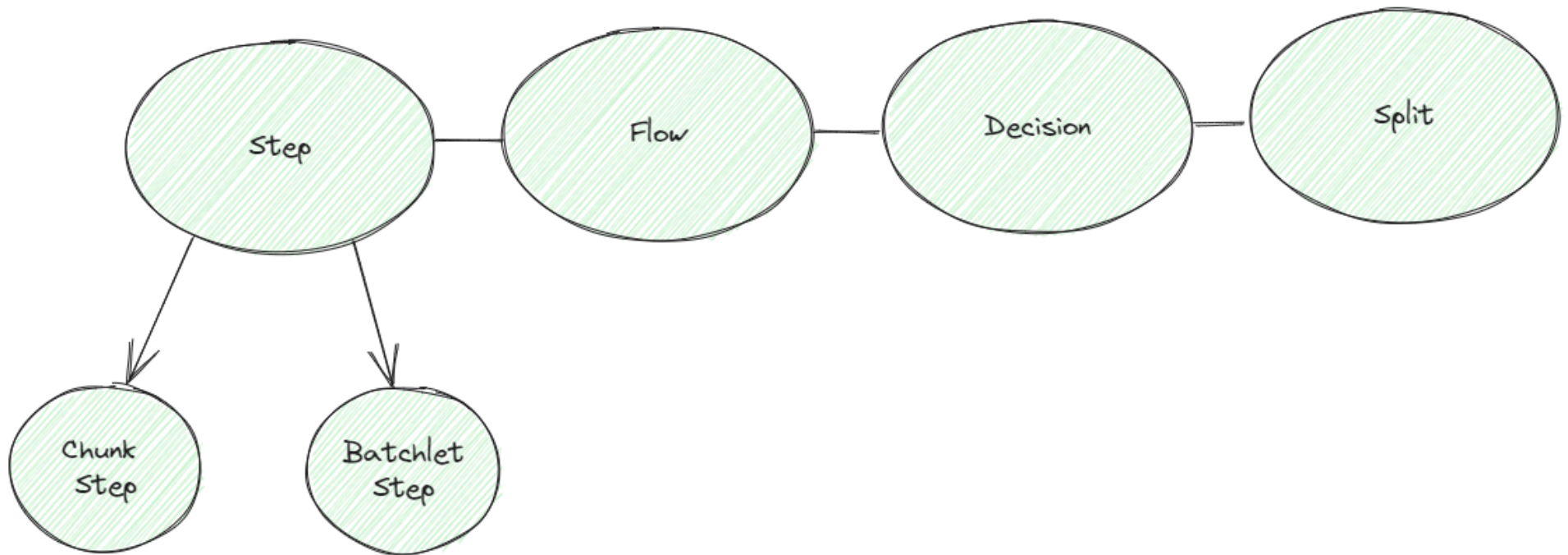


- ist verantwortlich dafür die verarbeiteten Daten des Item Processors in einen Datenspeicher zu schreiben wie z.B. Datenbank, Datei oder Message Queue

2.8. WORKFLOW IM BATCH PROCESSING SYSTEM



2.9 STEP-ARTEN



BEISPIEL: CHUNK

CHUNK AUFGABE:

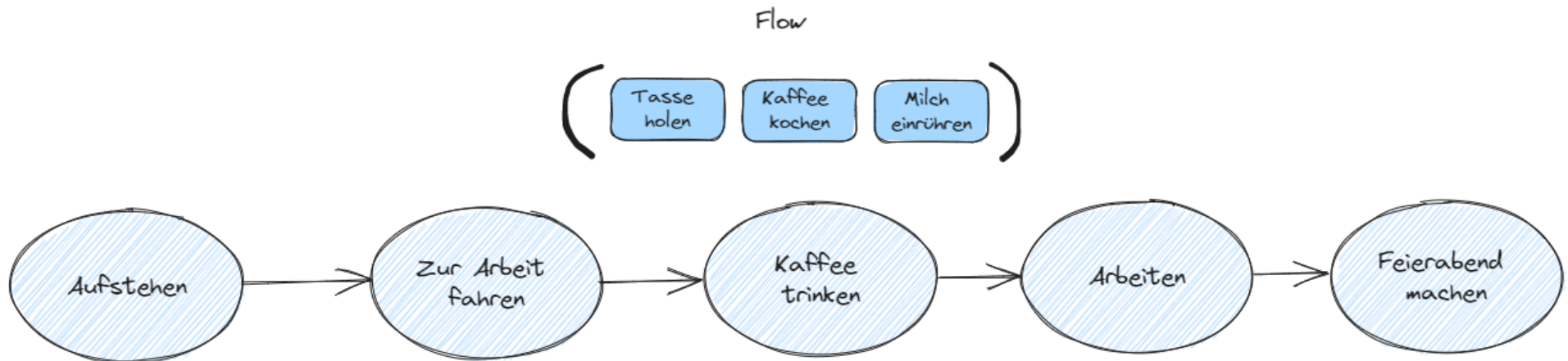
- Chunkverarbeitung für die neue csv-Datei mit den Kundendaten implementieren

2.9.1 FLOW

- definiert eine Folge von Ausführungen, die zusammen als Einheit ausgeführt werden
- können auch zu den Elementen next, stop, fail oder end übergehen



2.9.1 FLOW

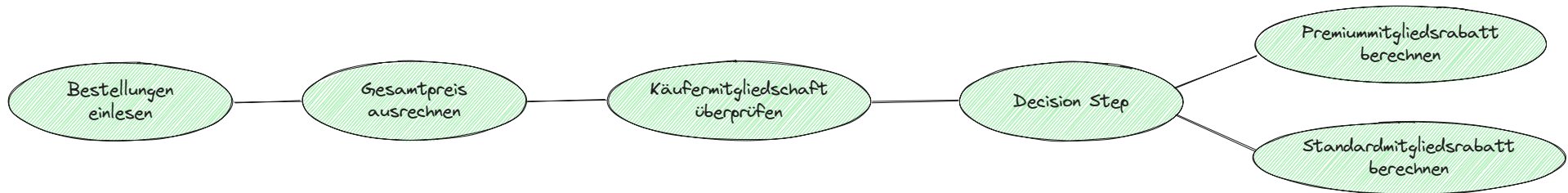


BEISPIEL: FLOW

FLOW AUFGABE:

- Neue Spalte einfügen mit dem Namen 'membership'
- im Processor soll ermittelt werden, ob der Kunde schon länger als 7 Jahre bei uns Kunde ist
- falls ja, soll in der Membership Spalte 'Premium' gesetzt werden
- falls nicht, soll in der Membership Spalte 'Basic' stehen
- im Writer die SQL Query anpassen
- Customer Klasse anpassen
- xml strukturieren

2.9.2 DECISION



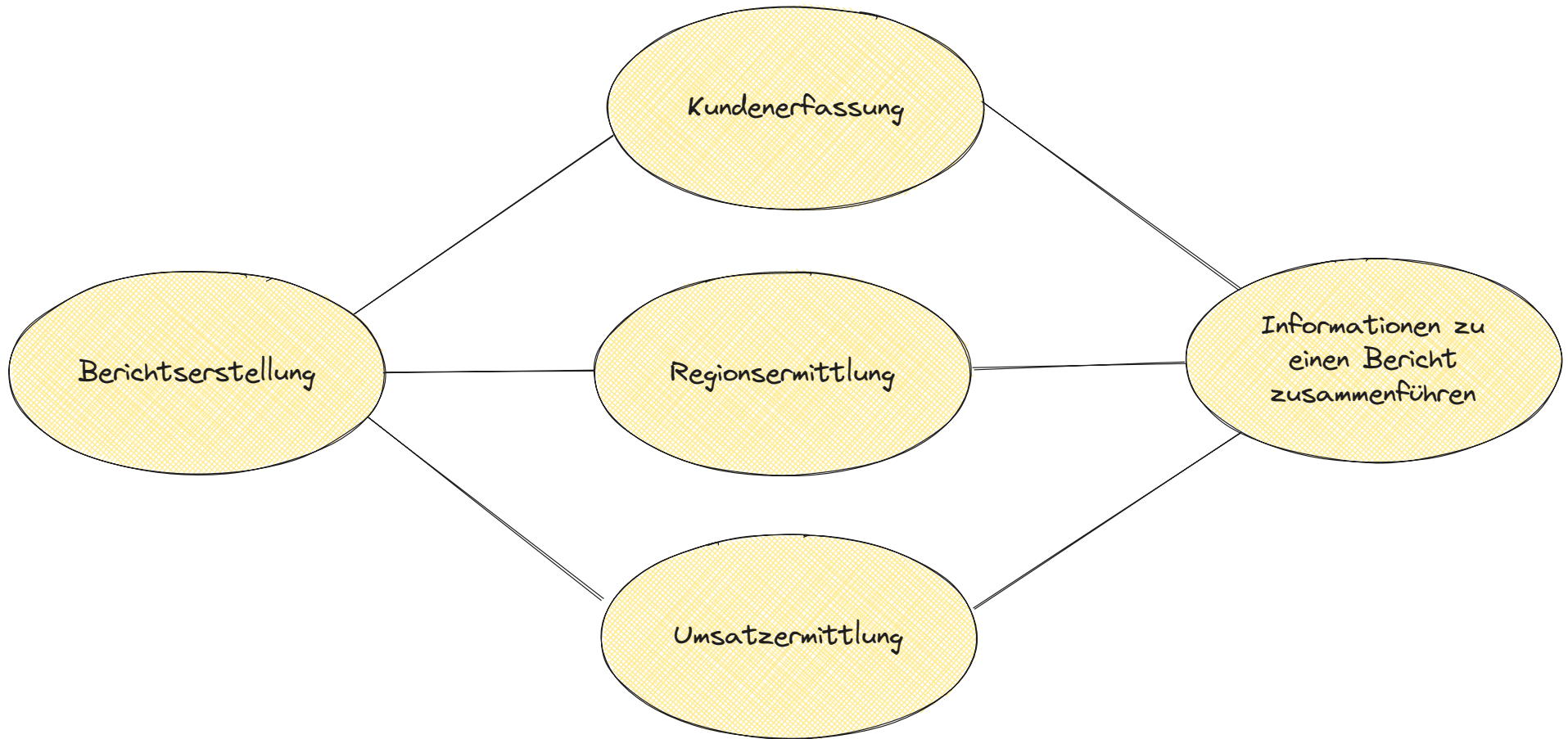
- bietet die Möglichkeit die Reihenfolge von Steps, Flows und Splits zu bestimmen
- mithilfe der Elemente next, stop, fail oder end kann der nächste Flow, Step, Decision oder Split bestimmt werden

BEISPIEL: DECISION

DECISION AUFGABE:

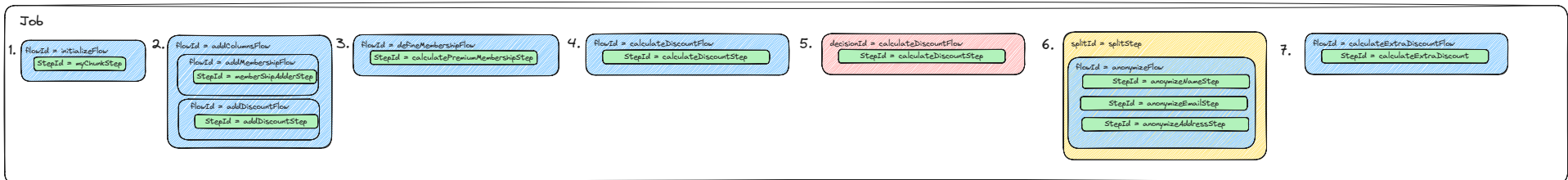
1. nach XML werden zuerst die zwei neuen Spalten hinzugefügt
2. danach werden die defineMembershipFlow und der calculateDiscountFlow definiert
3. Basic 10% und Premium 20%
4. hier wird der decider eingestellt
5. im DBCalculateExtraltemProcessor wird ein zusätzlicher Rabatt berechnet, falls der Kunde aus München kommt, wenn der Decider den BatchStatus 'CALCULATE_EXTRA_DISCOUNT' zurückgibt

2.9.2 SPLIT



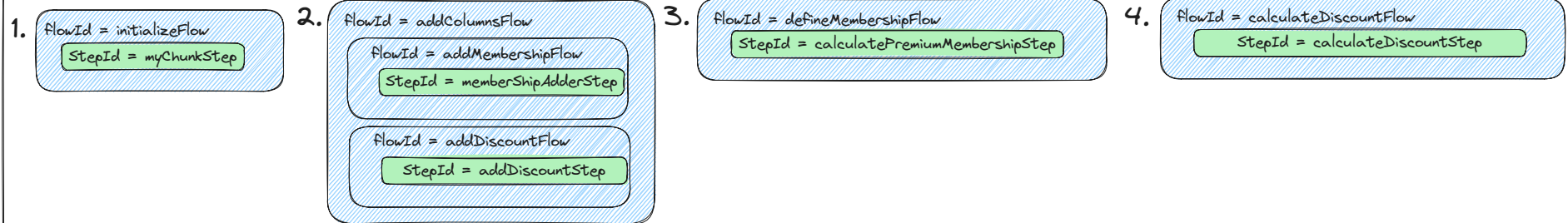
- definiert eine Gruppe von Flows, die gleichzeitig ausgeführt werden

BEISPIEL: STEP-FLOW-DECISION-SPLIT



BEISPIEL: STEP-FLOW-DECISION-SPLIT

Job



BEISPIEL: STEP-FLOW-DECISION-SPLIT

Job

5.

decisionId = calculateDiscountFlow
StepId = calculateDiscountStep

6.

splitId = splitStep

flowId = anonymizeFlow

StepId = anonymizedNameStep

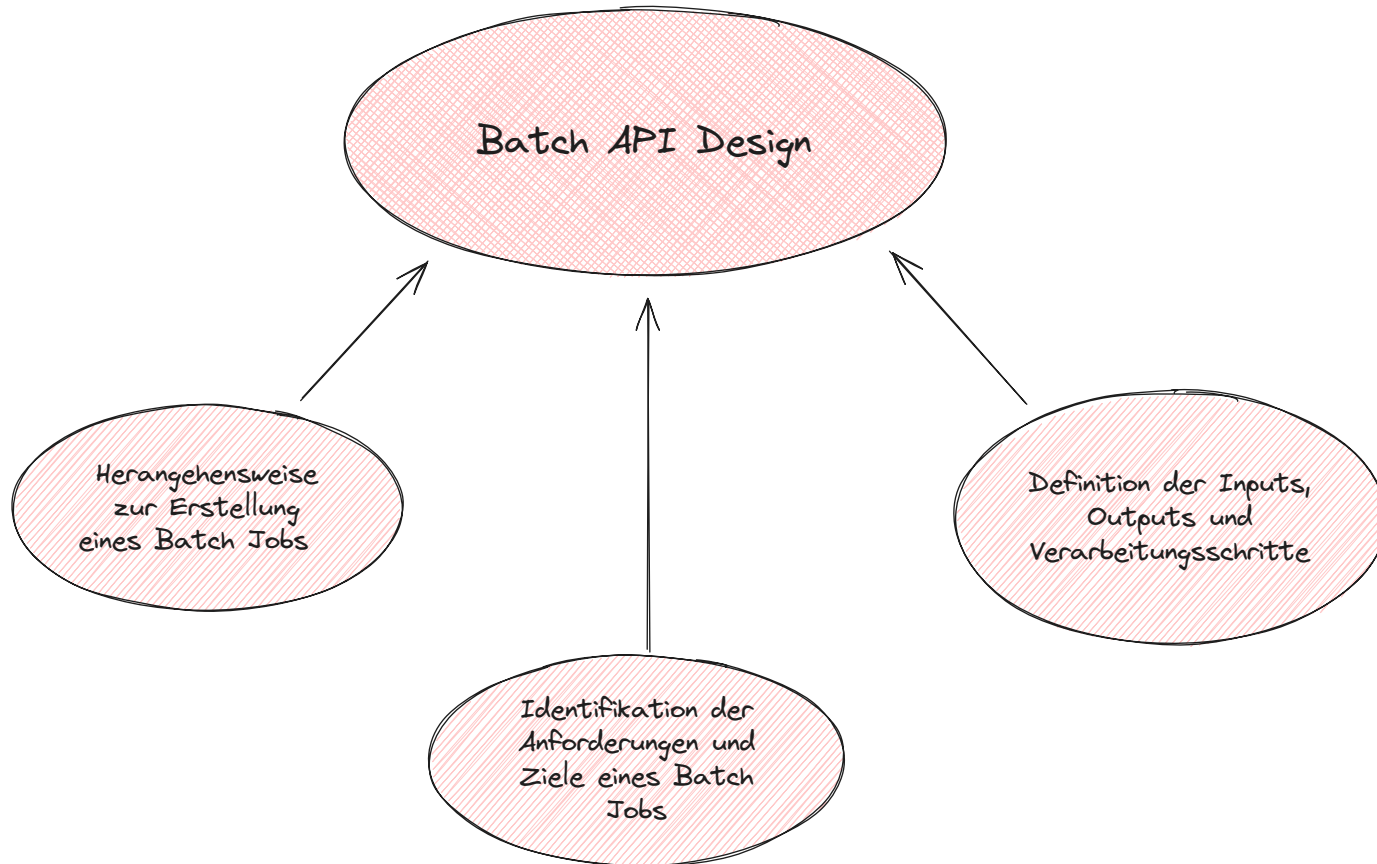
StepId = anonymizeEmailStep

StepId = anonymizeAddressStep

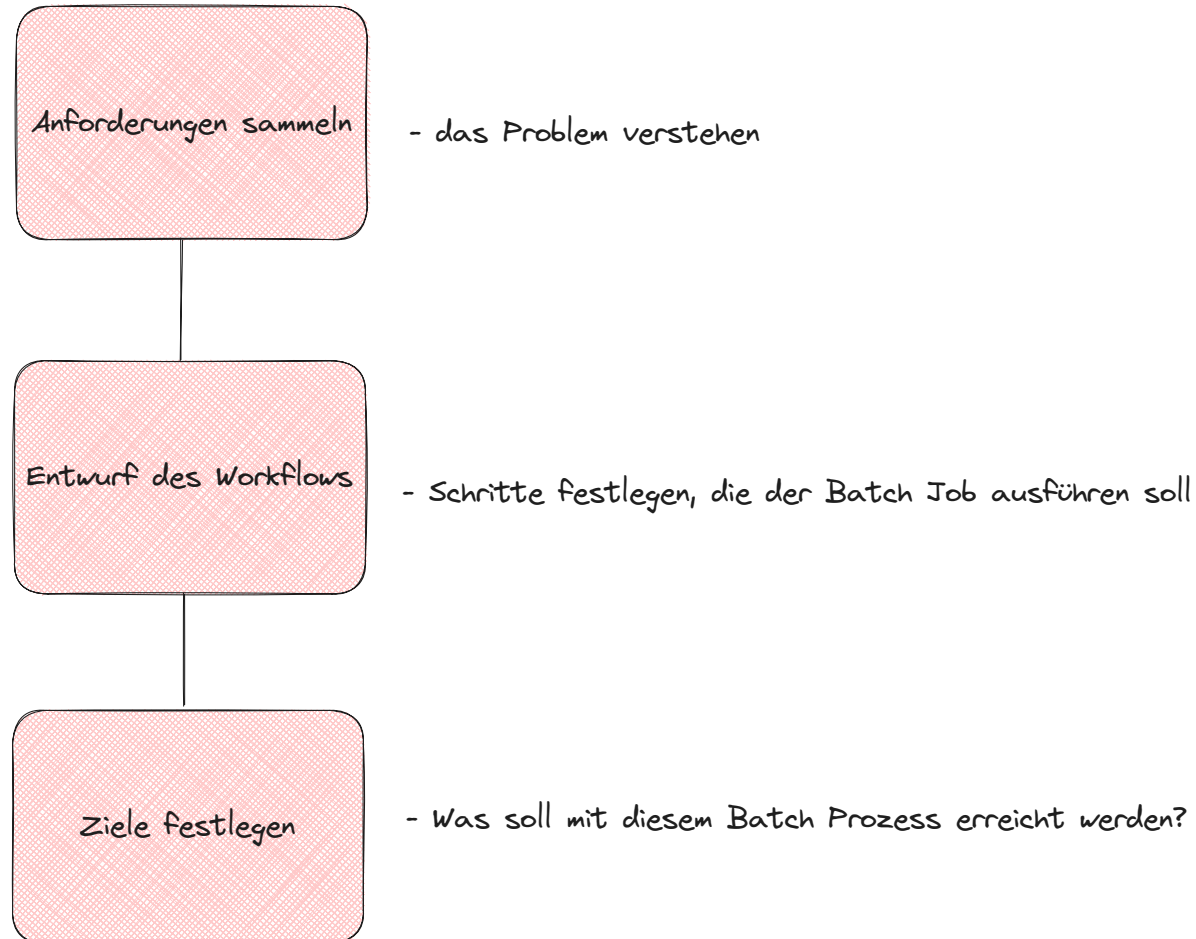
7.

flowId = calculateExtraDiscountFlow
StepId = calculateExtraDiscount

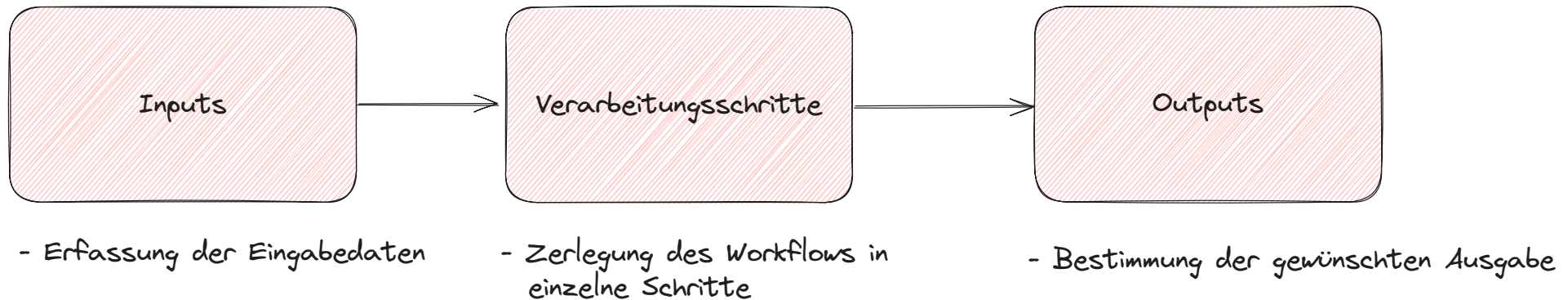
3. BATCH JOB DESIGN



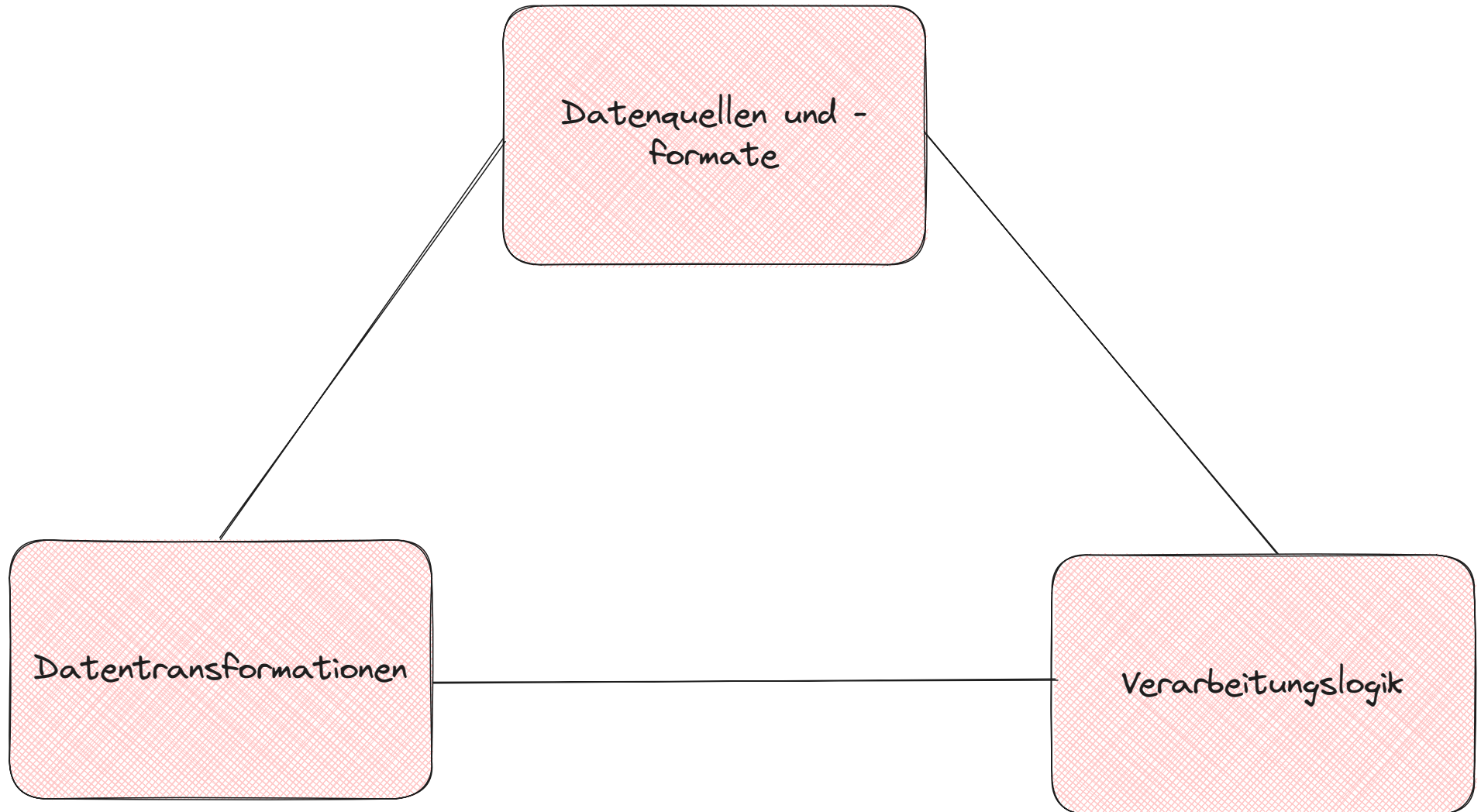
3.1. HERANGEHENSWEISE ZUR ERSTELLUNG EINES BATCH JOBS



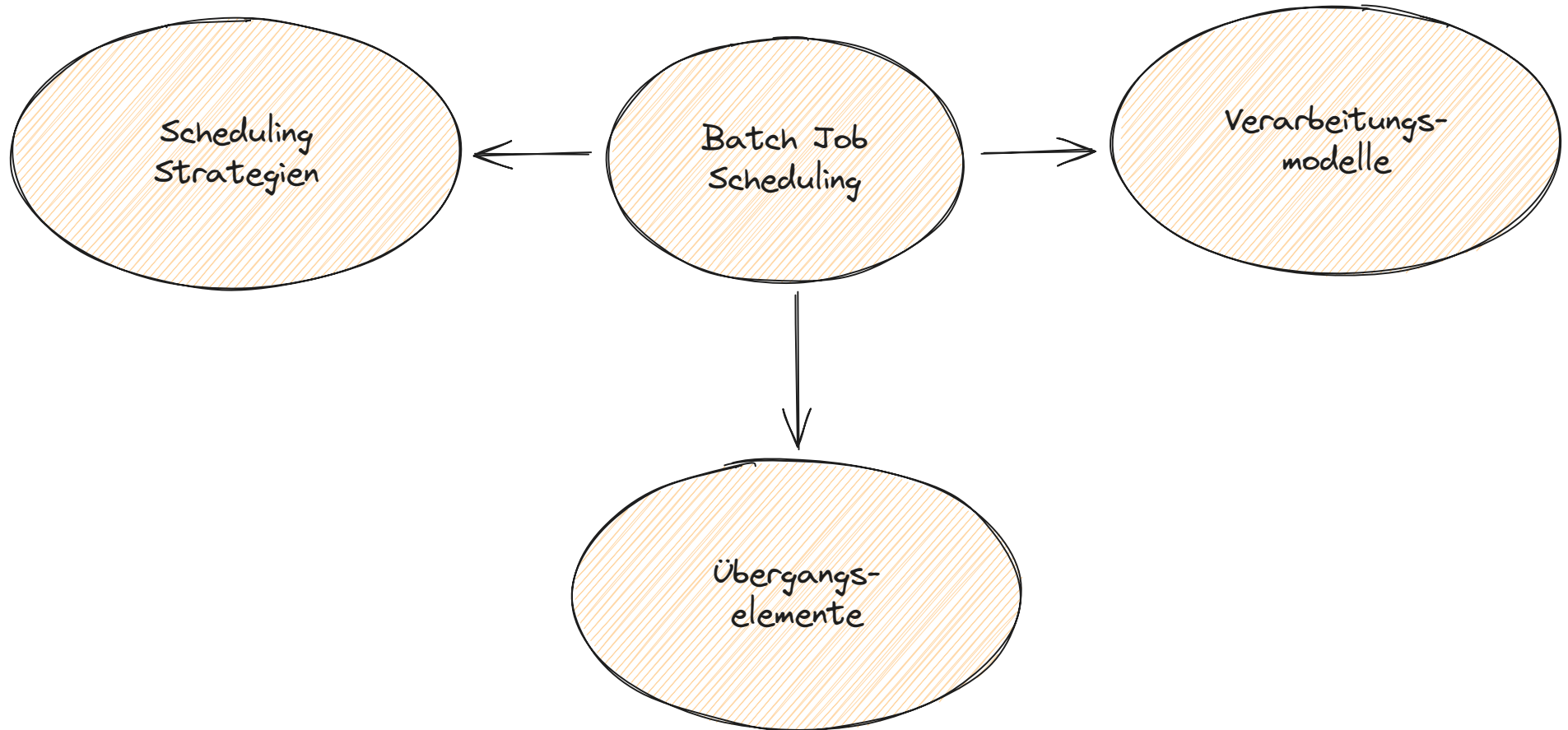
3.2. DEFINITION VON INPUTS, VERARBEITUNGSSCHRITTEN UND OUTPUTS



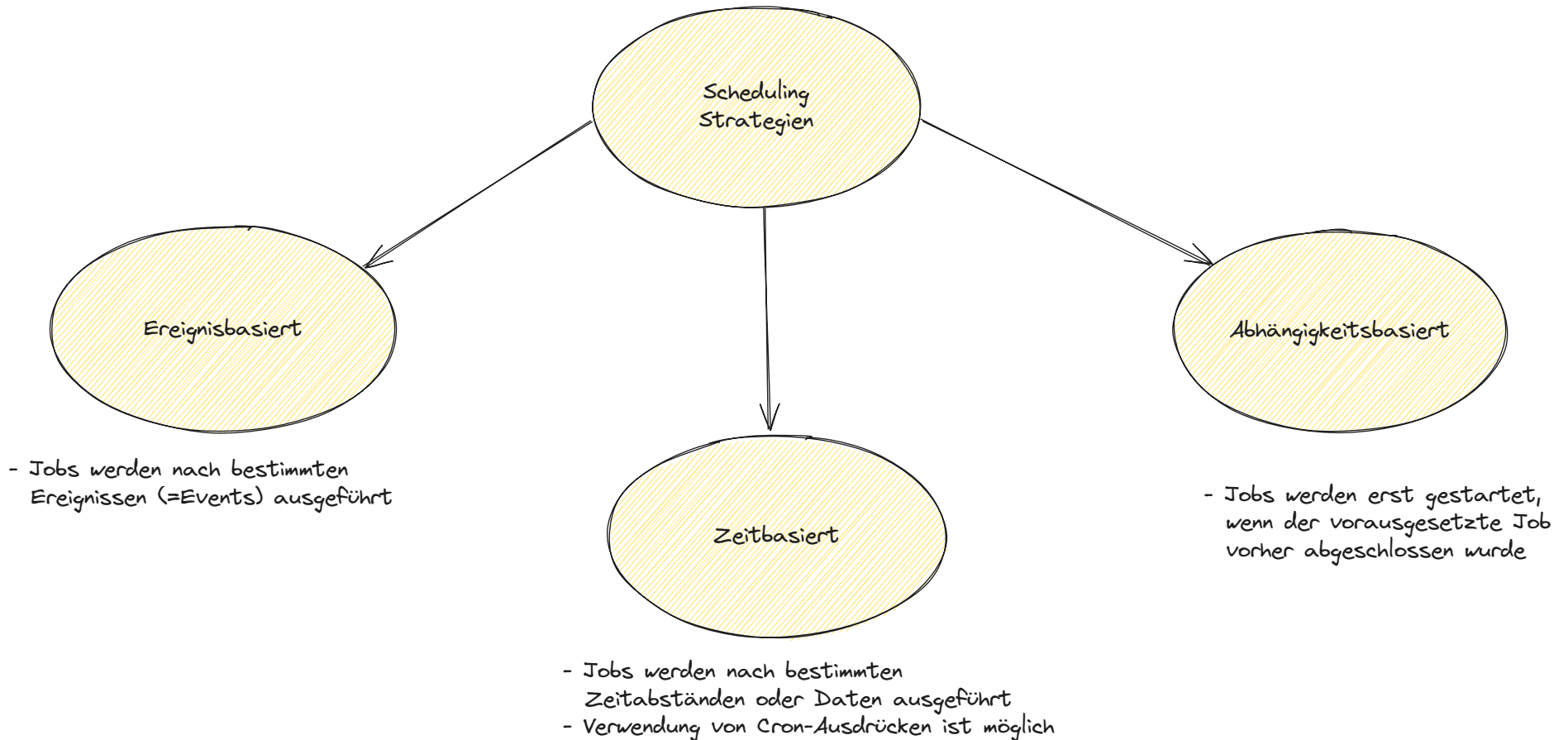
3.3. IDENTIFIKATION DER ANFORDERUNGEN UND ZIELE EINES BATCH JOBS



4. BATCH JOB SCHEDULING UND AUSFÜHRUNG



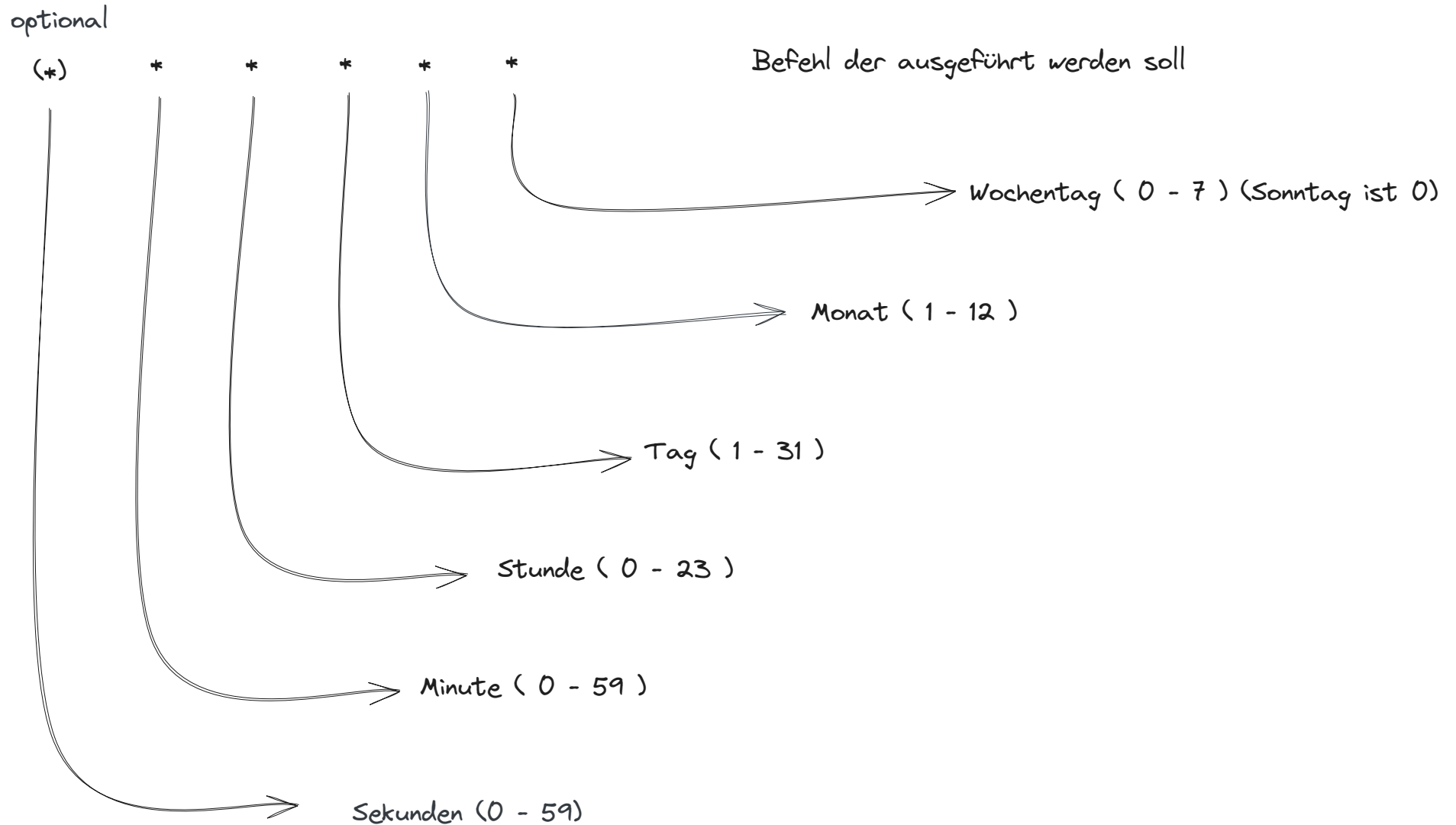
4.1. SCHEDULING STRATEGIEN



4.1.1. CRON SYNTAX

- ist ein zeitbasierter Job-Scheduler in Unix oder Unix-ähnlichen Betriebssystemen
- dient zur Automatisierung von Befehlen und Shell-Skripten
- für die Ausführung in Schritten verwendet man das "/"
- und für Bereiche verwenden wir "-"

4.1.1. CRON SYNTAX



4.1.1. CRON SYNTAX

0	0	2	*	*	*	jeden Tag um 2 Uhr
0	30	20	*	*	1-5	von Montag bis Freitag täglich um 20:30 Uhr
0	*	*	15,30	*	*	jeden 15. und jeden 30. Tag im Monat
0	10-20	2,5	*	*	*	jede 10. und jede 20. Minute in jeder 2. und 5. Stunde
0	*/15	2-6	*	*	*	alle 15 Minuten von 2 bis 6 Uhr morgens
0	*	*	*	4-5	0,6	jeden Samstag und Sonntag im April und im Mai

4.1.2. EINSCHUB: WILDFLY



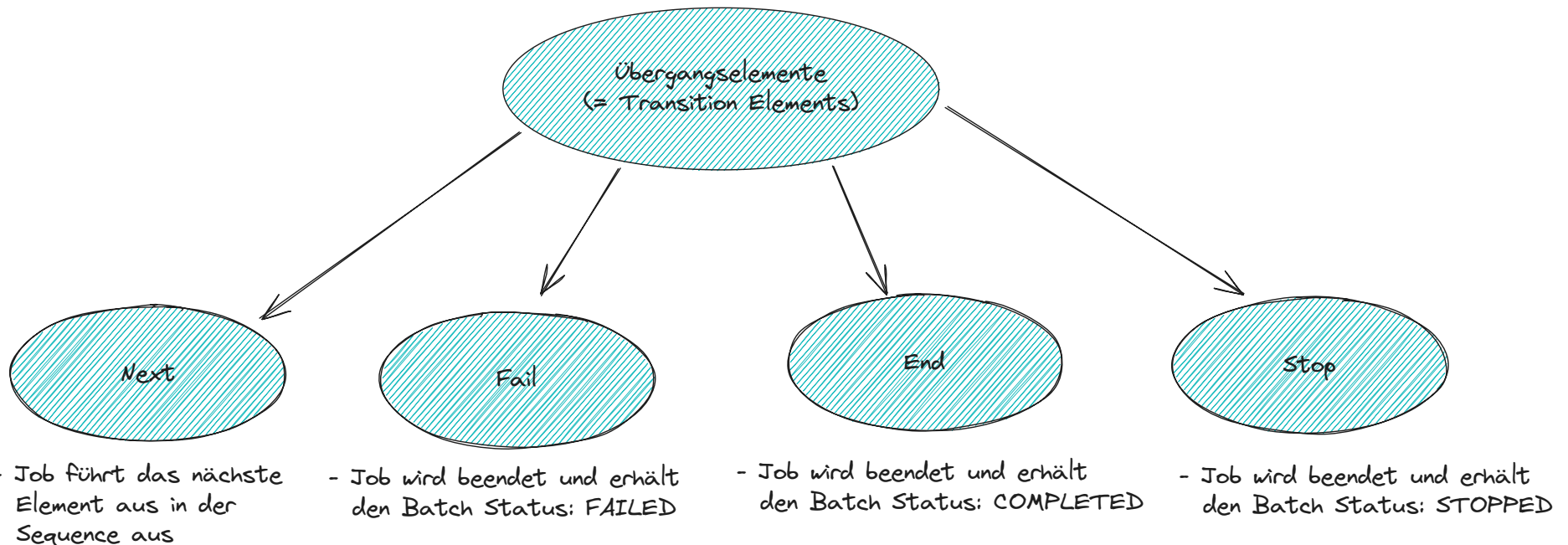
- Anwendungsserver nach dem Jakarta-EE-Standard
- Teil der JBoss Middleware Frameworks
- plattformunabhängig

BEISPIEL: CRON SYNTAX

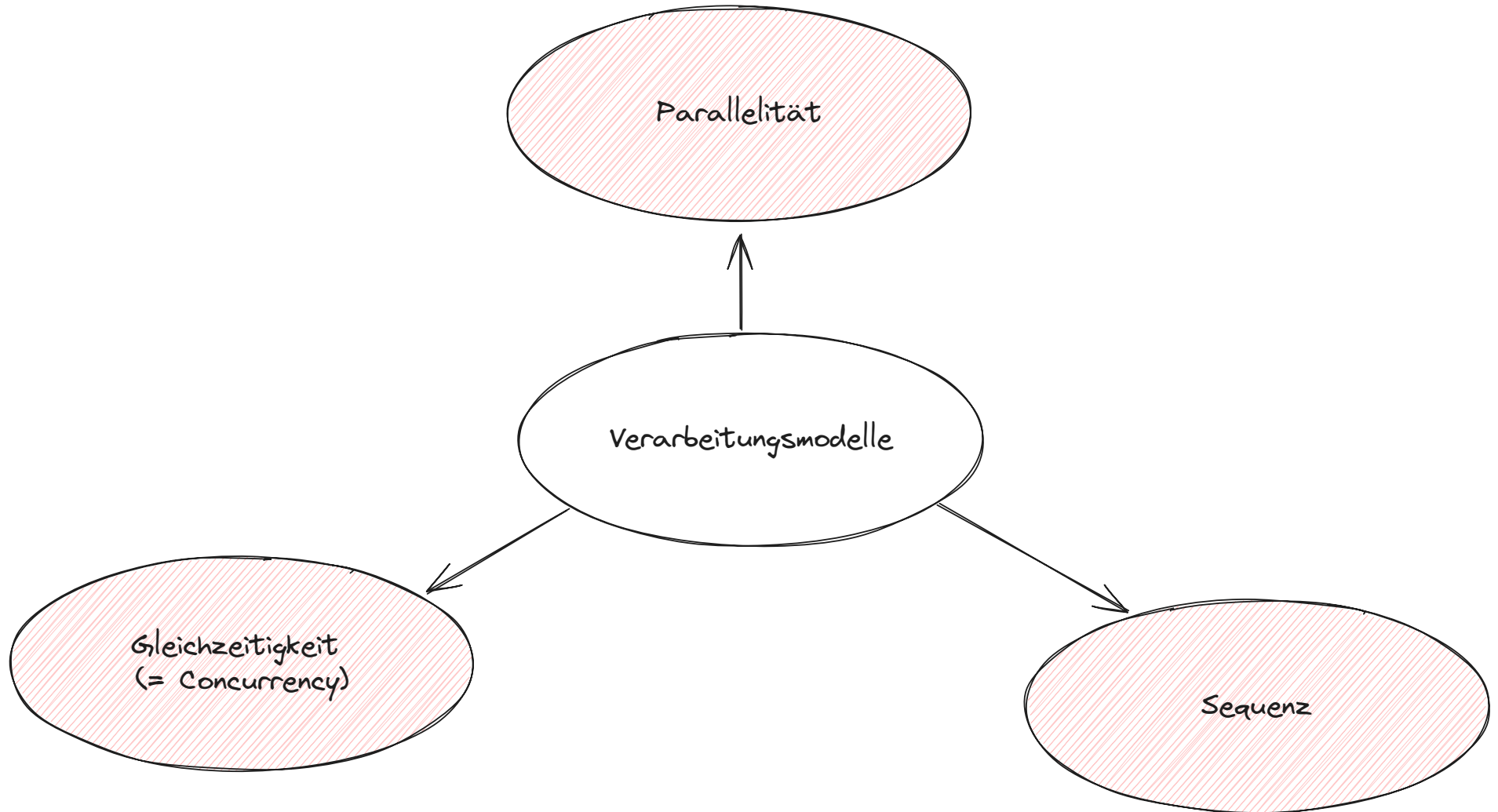
CRON AUFGABE:

- Konfigurieren Sie den CronJob so, dass er genau heute am 15. April am Montag zur aktuellen Stunde alle 5 Sekunden ausgeführt wird

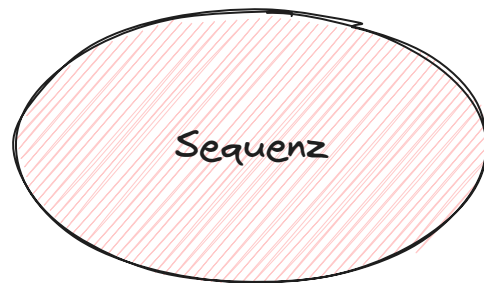
4.2. ÜBERGANGSELEMENTE



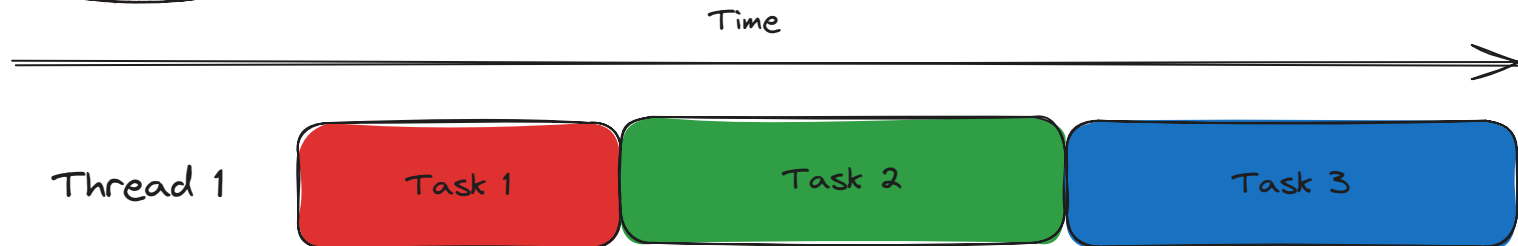
4.3. VERARBEITUNGSMODELLE



4.3.1. SEQUENZ



- mehrere Aufgaben werden nacheinander Schritt für Schritt bearbeitet



BEISPIEL: STEP SEQUENCE CODE

batchProcessingJob

myChunkStep

```
reader = MyItemReader  
processor = MyItemProcessor  
writer = MyItemWriter
```

weaponManipulationEnhancerStep

```
reader = DBItemReader  
processor = ItemProcessorWeaponManipulator  
writer = DBUpdateItemWriter
```

updatePowerLevelOfEnhancedHeroesStep

```
reader = DBFindByColumnValueItemReader  
processor = UpdatePowerLevelItemProcessor  
writer = DBUpdateItemWriter
```

STEP SEQUENCE CODE AUFGABE:

- Erstelle drei Steps:
 - myChunkStep: aus csv-Datei lesen und in die Datenbank schreiben
 - münchenZipReplacerStep: aus Datenbank mit SQL Abfrage lesen, Münchner PLZ mit "77777" ersetzen und geänderte Datensätze in DB updaten
 - hansReplacerStep: aus Datenbank nur Kunden auslesen, die "Hans" heißen, "Hans" mit "JOHANNES" ersetzen, geänderte Datensätze in DB updaten und in der Abfrage für 'firstName' und 'Hans' @BatchProperty verwenden

STEP SEQUENCE CODE AUFGABE

batchProcessingJob

myChunkStep

```
reader = MyItemReader  
processor = MyItemProcessor  
writer = MyItemWriter
```

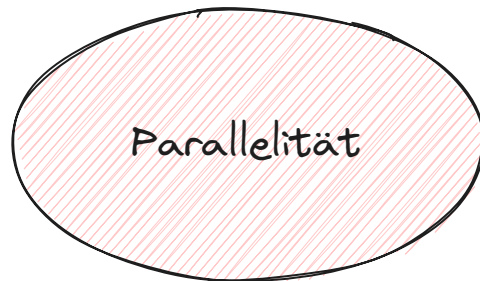
münchenZIPReplacerStep

```
reader = DBItemReader  
processor = ItemProcessorMünchenZIPReplacer  
writer = DBUpdateItemWriter
```

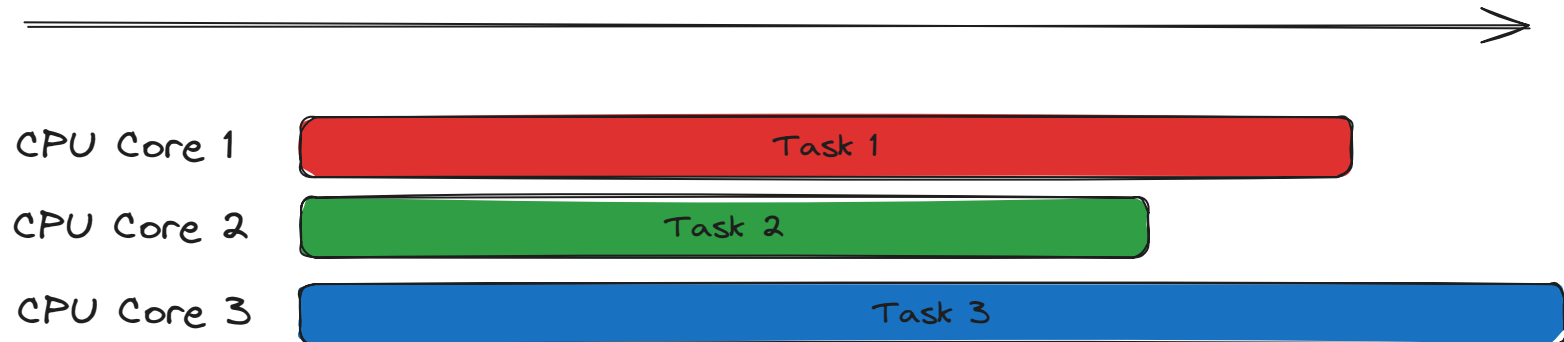
hansReplacerStep

```
reader = DBFindByColumnValueItemReader  
processor = ItemProcessorHansReplacer  
writer = DBUpdateItemWriter
```

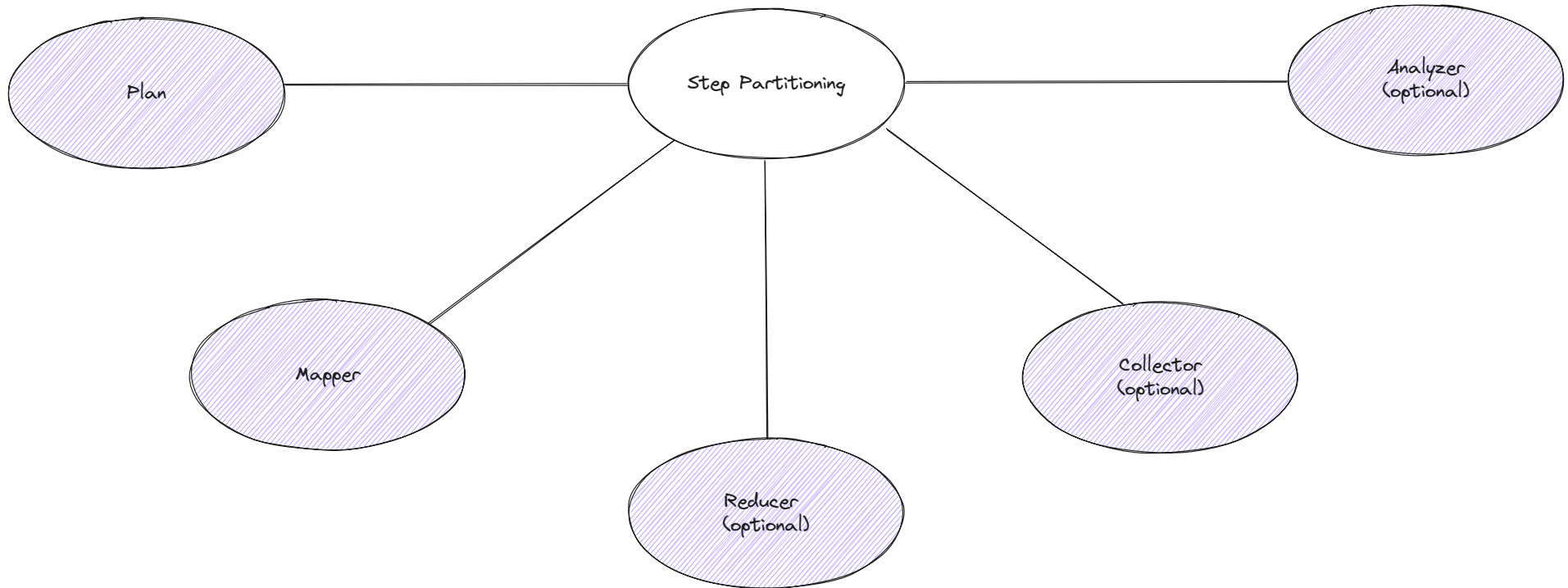
4.3.2. PARALLELITÄT



- mehrere Aufgaben werden zur selben Zeit bearbeitet

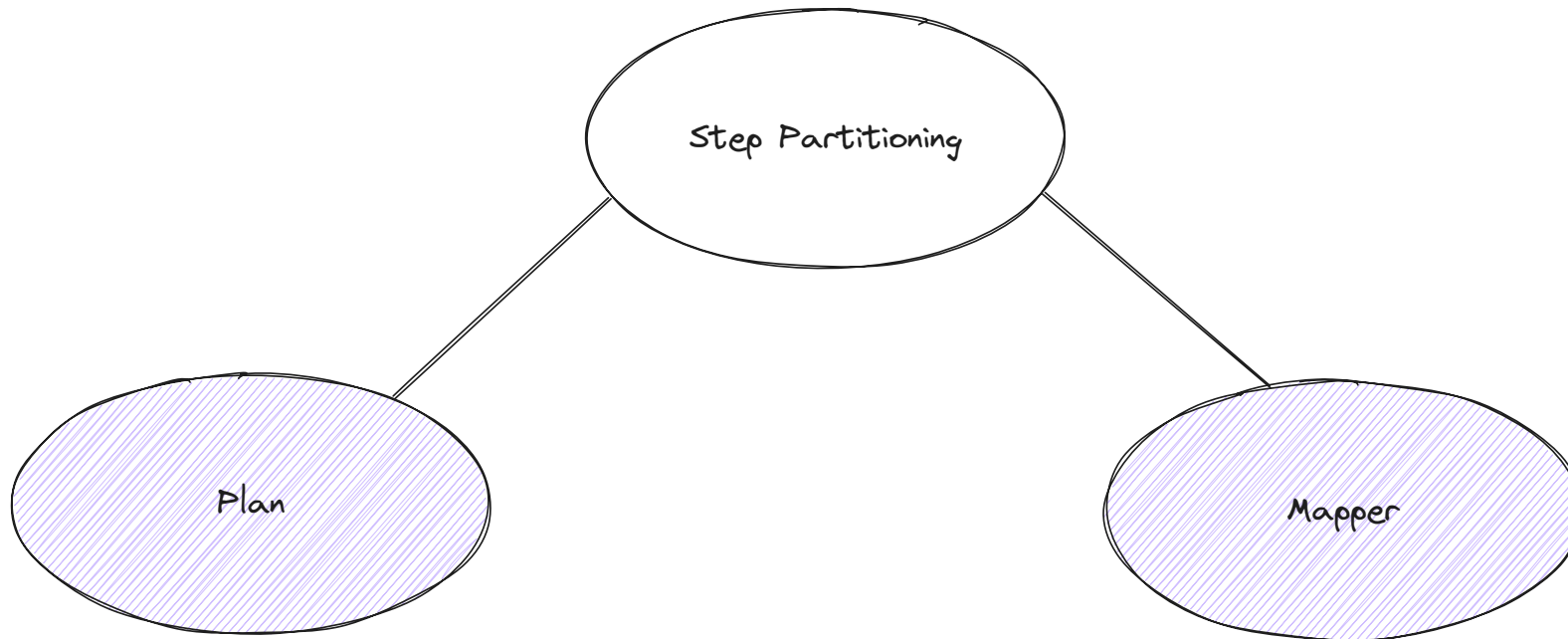


4.3.2.1. STEP PARTITIONING



- Technik, um eine große Datenmenge in kleinere Teile (=Partitionen) aufzuspalten und sie mithilfe von mehreren Threads gleichzeitig zu verarbeiten
- sorgt für eine schnellere Bearbeitungszeit

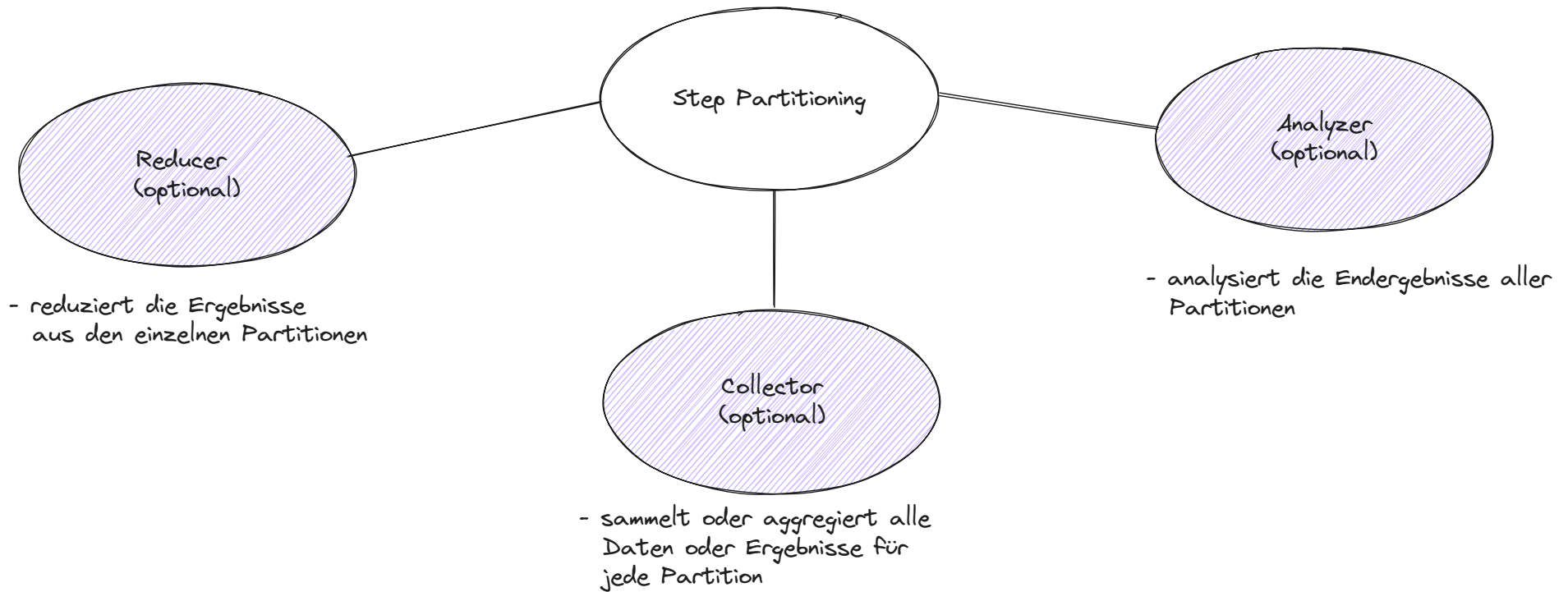
4.3.2.1 STEP PARTITIONING (NOTWENDIG)



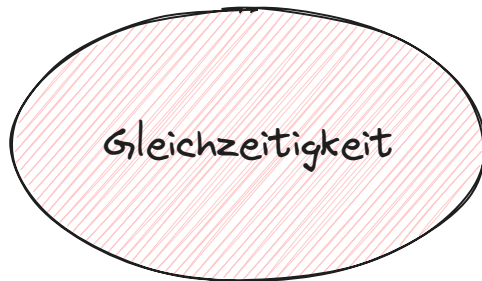
- kapselt die Konfiguration für jede Partition des Steps

- dient zur Berechnung der Anzahl von Partitionen und Threads

4.3.2.1 STEP PARTITIONING (OPTIONAL)



4.3.3. GLEICHZEITIGKEIT (= CONCURRENCY)

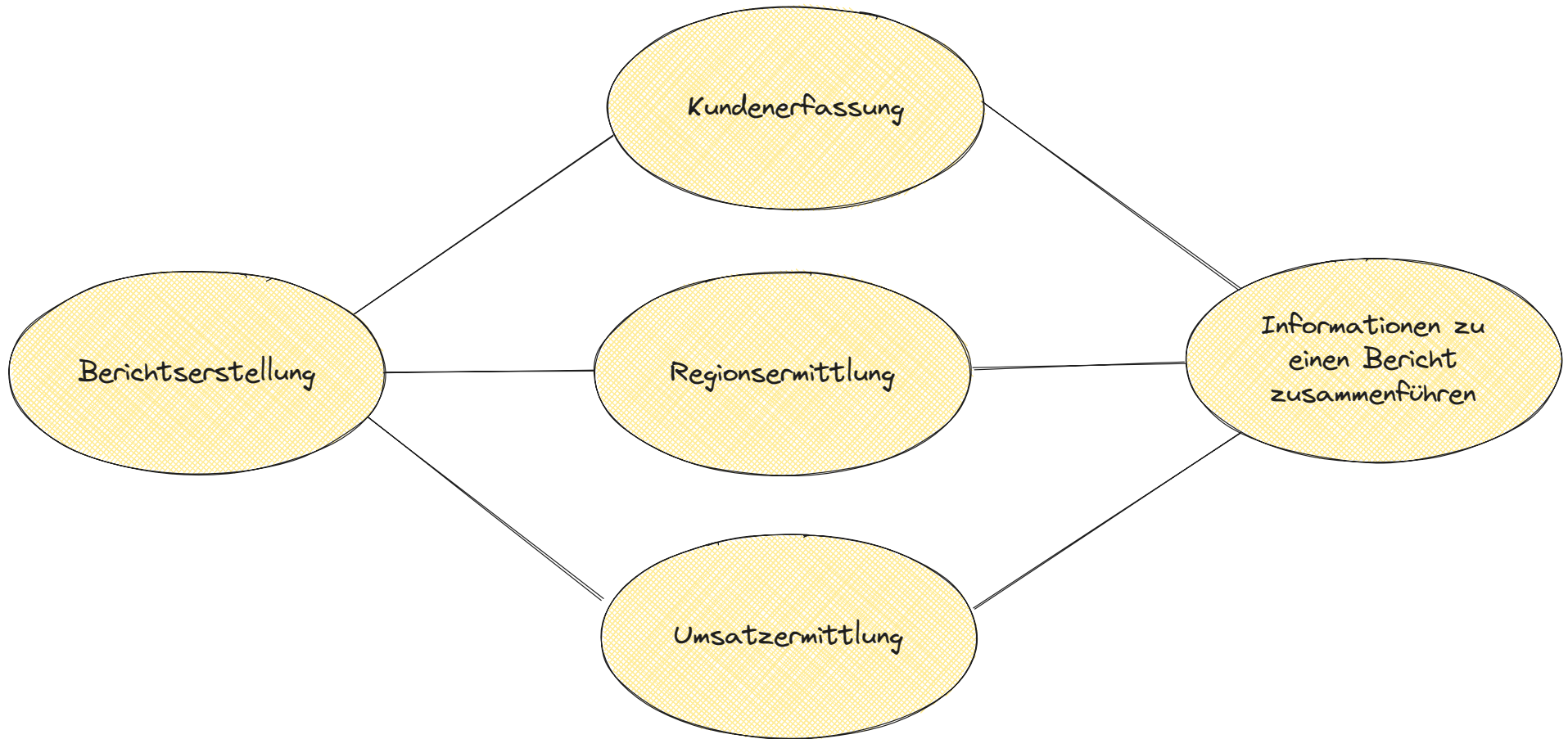


- mehrere Aufgaben koexistieren gleichzeitig und werden effektiv verarbeitet

Thread 1

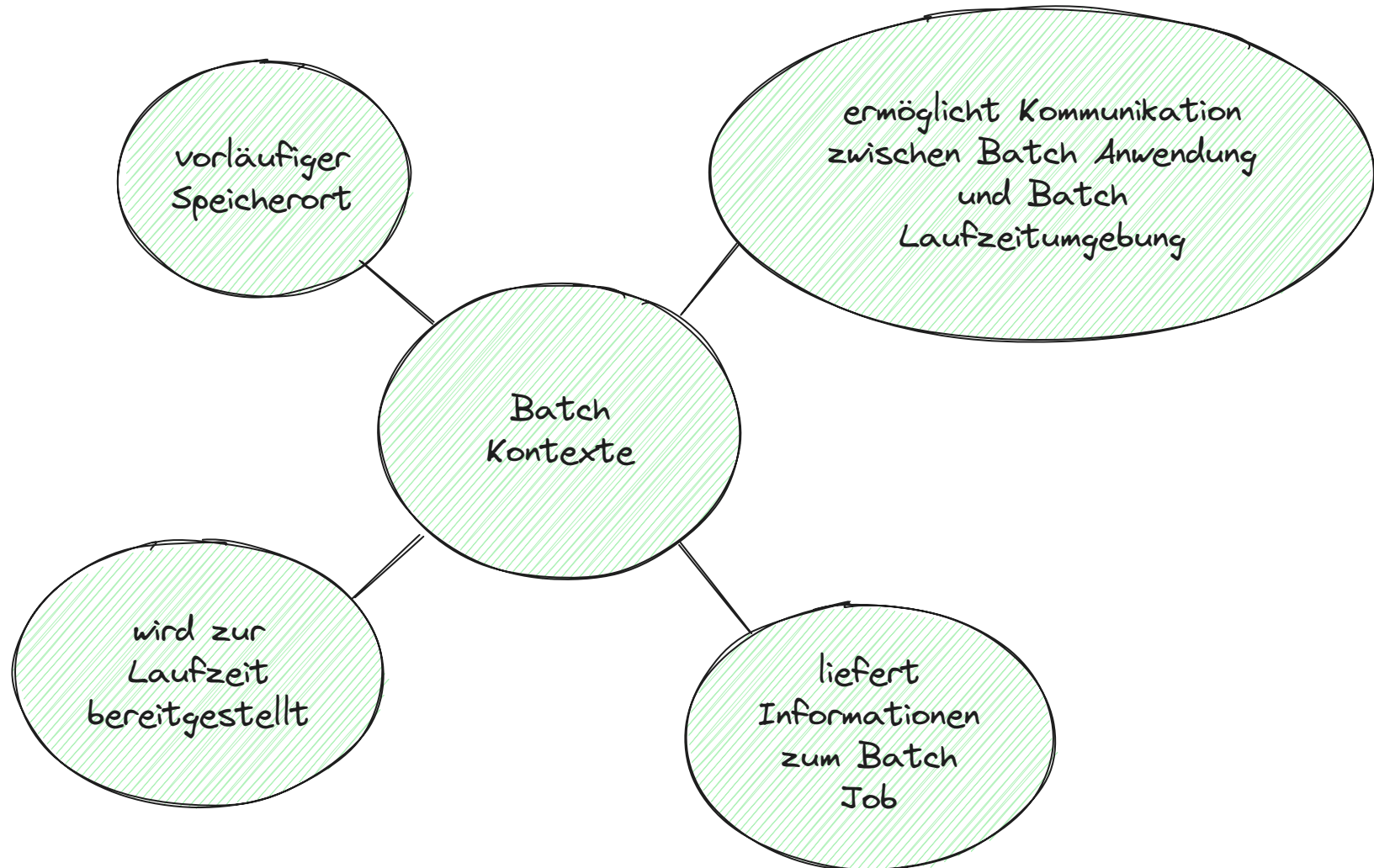


4.3.3.1. SPLIT



5. INTERFACES

5.1. BATCH CONTEXTS



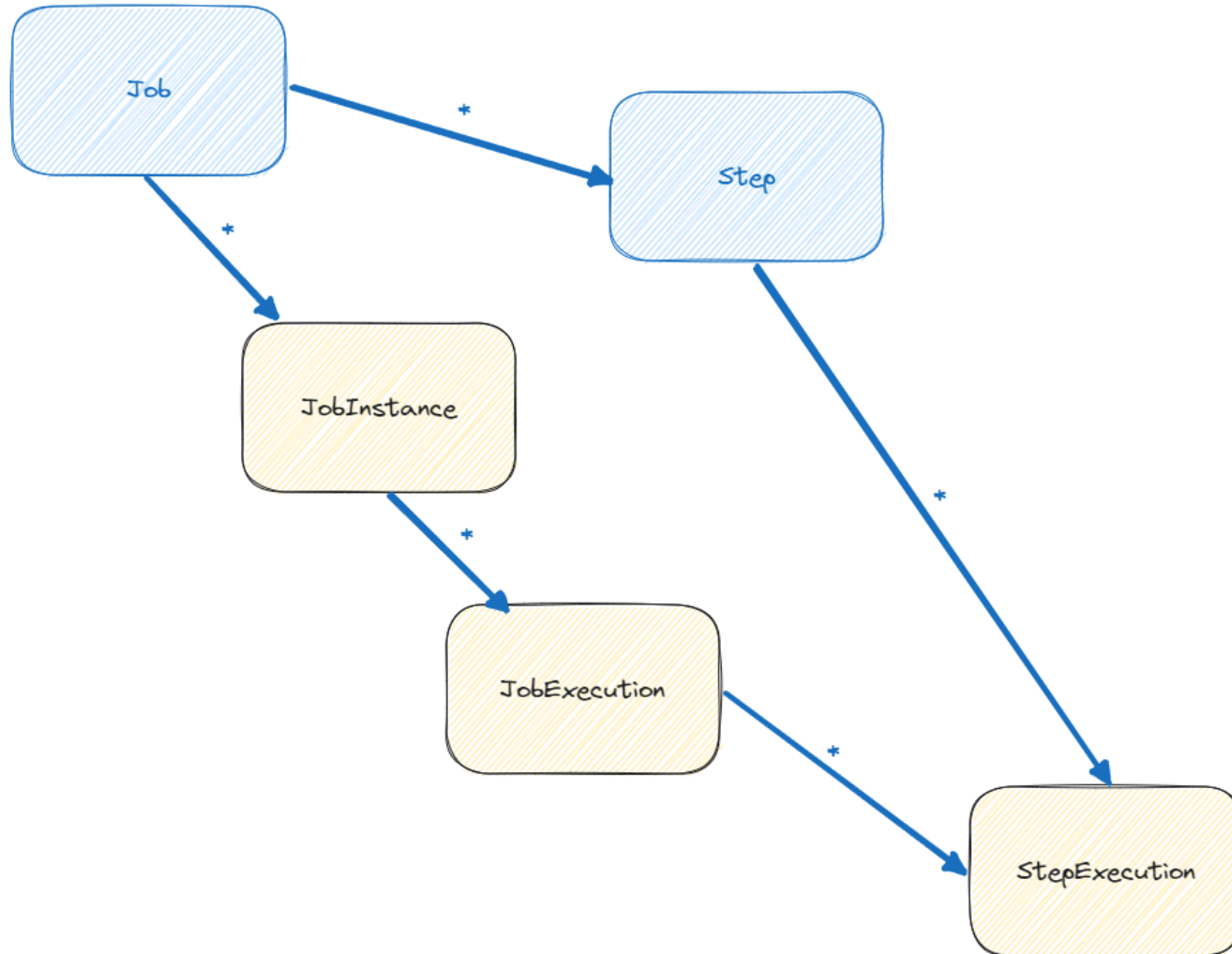
5.1.1 STEP CONTEXT

```
public interface StepContext {  
    public String getStepName();  
    public Object getTransientUserData();  
    public void setTransientUserData(Object data);  
    public long getStepExecutionId();  
    public Properties getProperties();  
    public Serializable getPersistentUserData();  
    public void setPersistentUserData(Serializable data);  
    public BatchStatus getBatchStatus();  
    public String getExitStatus();  
    public void setExitStatus(String status);  
    public Exception getException();  
    public Metric[] getMetrics();  
}
```

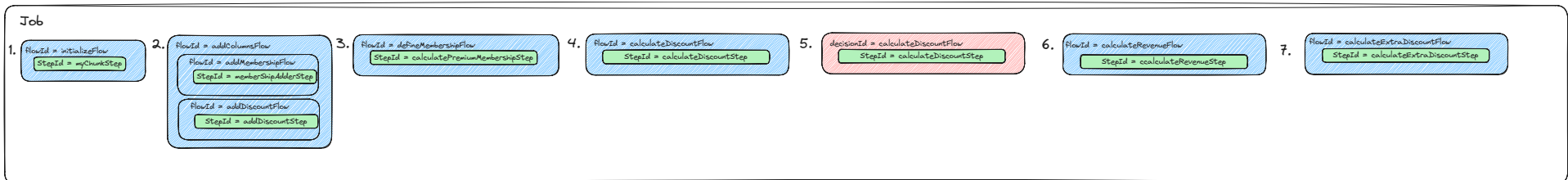
5.1.2. JOB CONTEXT

```
public interface JobContext {  
    public String getJobName();  
    public Object getTransientUserData();  
    public void setTransientUserData(Object data);  
    public Properties getProperties();  
    public BatchStatus getBatchStatus();  
    public String getExitStatus();  
    public void setExitStatus(String status);  
    public long getInstanceId();  
    public long getExecutionId();  
}
```

5.1.2. JOB CONTEXT



BEISPIEL: JOB UND STEPCONTEXT



BEISPIEL: JOB UND STEPCONTEXT

Job

5.

decisionId = calculateDiscountFlow

StepId = calculateDiscountStep

6.

flowId = calculateRevenueFlow

StepId = calculateRevenueStep

7.

flowId = calculateExtraDiscountFlow

StepId = calculateExtraDiscount

ÜBUNG: JOB UND STEPCONTEXT NUTZEN

Job

1.

flowId = initializeFlow

StepId = myChunkStep

2.

flowId = weaponManipulationEnhancerFlow

StepId = weaponManipulationEnhancerStep

3.

flowId = updatePowerLevelOfEnhancedHeroesFlow

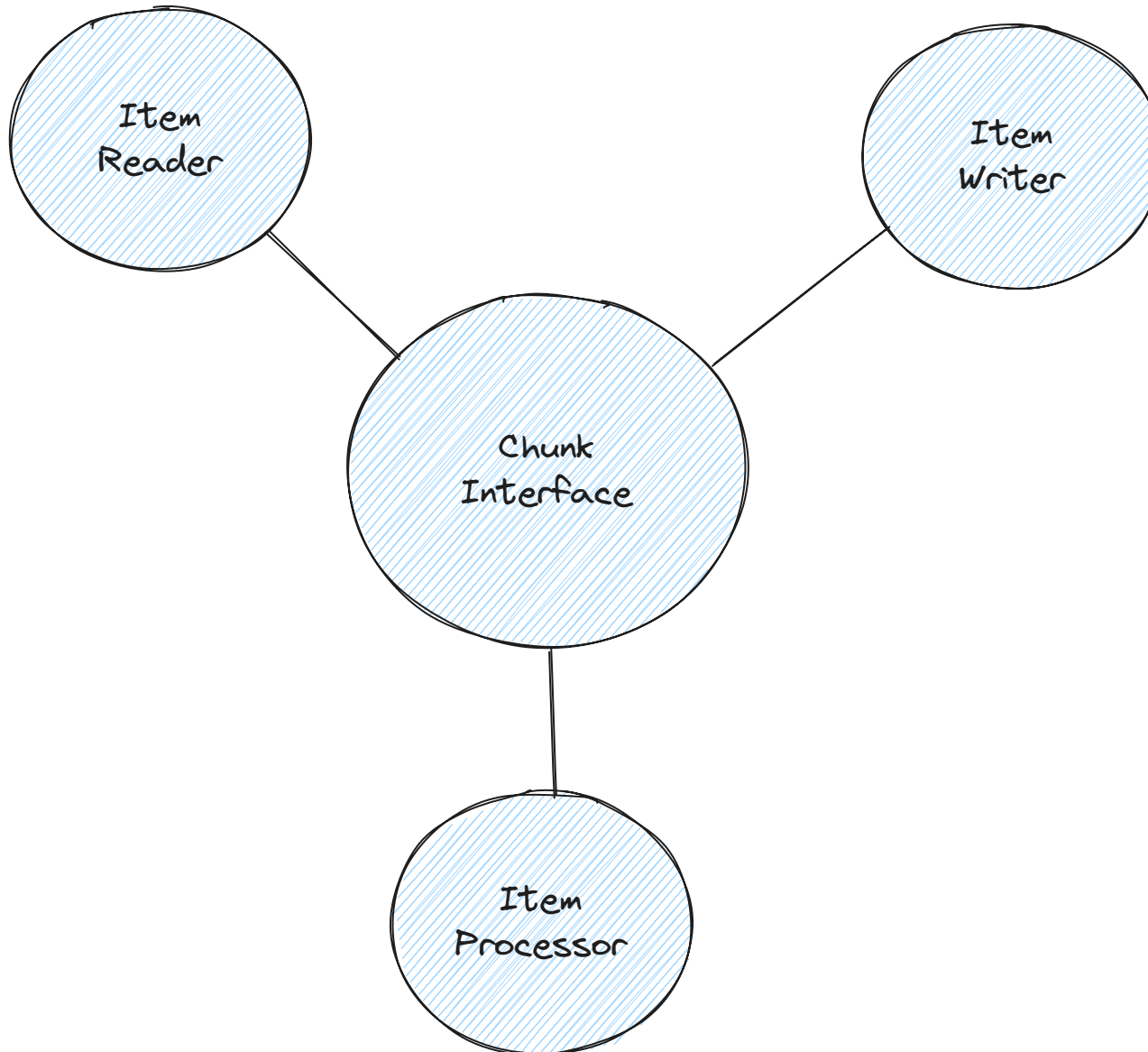
StepId = updatePowerLevelOfEnhancedHeroesFlow

ÜBUNG: JOB UND STEPCONTEXT NUTZEN

Aufgabe:

- in UpdatePowerLevelItemProcessor die Anzahl der enhancedHeroes und avengers ohne Duplikate in StepContext abspeichern
- in ItemProcessorWeaponManipulationEnhancer die einzelnen Avengers als Liste im JobContext abspeichern
- die Zahlen und einzelnen Avengers im DBUpdateWriter ausgeben
- Extra Frage -> Wieso werden die Heroes zwei Mal geprinted?

5.2. CHUNK INTERFACES



5.2.1. ITEM READER

```
public interface ItemReader {  
    public void open(Serializable checkpoint) throws Exception;  
    public void close() throws Exception;  
    public Object readItem() throws Exception;  
    public Serializable checkpointInfo() throws Exception; }  
}
```

5.2.2. ITEM PROCESSOR

```
public interface ItemProcessor {  
    public Object processItem(Object item) throws Exception;  
}
```

5.2.3. ITEM WRITER

```
public interface ItemWriter {  
    public void open(Serializable checkpoint) throws Exception;  
    public void close() throws Exception;  
    public void writeItems(List<Object> items) throws Exception;  
    public Serializable checkpointInfo() throws Exception;  
}
```

5.3. BATCHLET INTERFACE

```
public interface Batchlet {  
    public String process() throws Exception;  
    public void stop() throws Exception;  
}
```

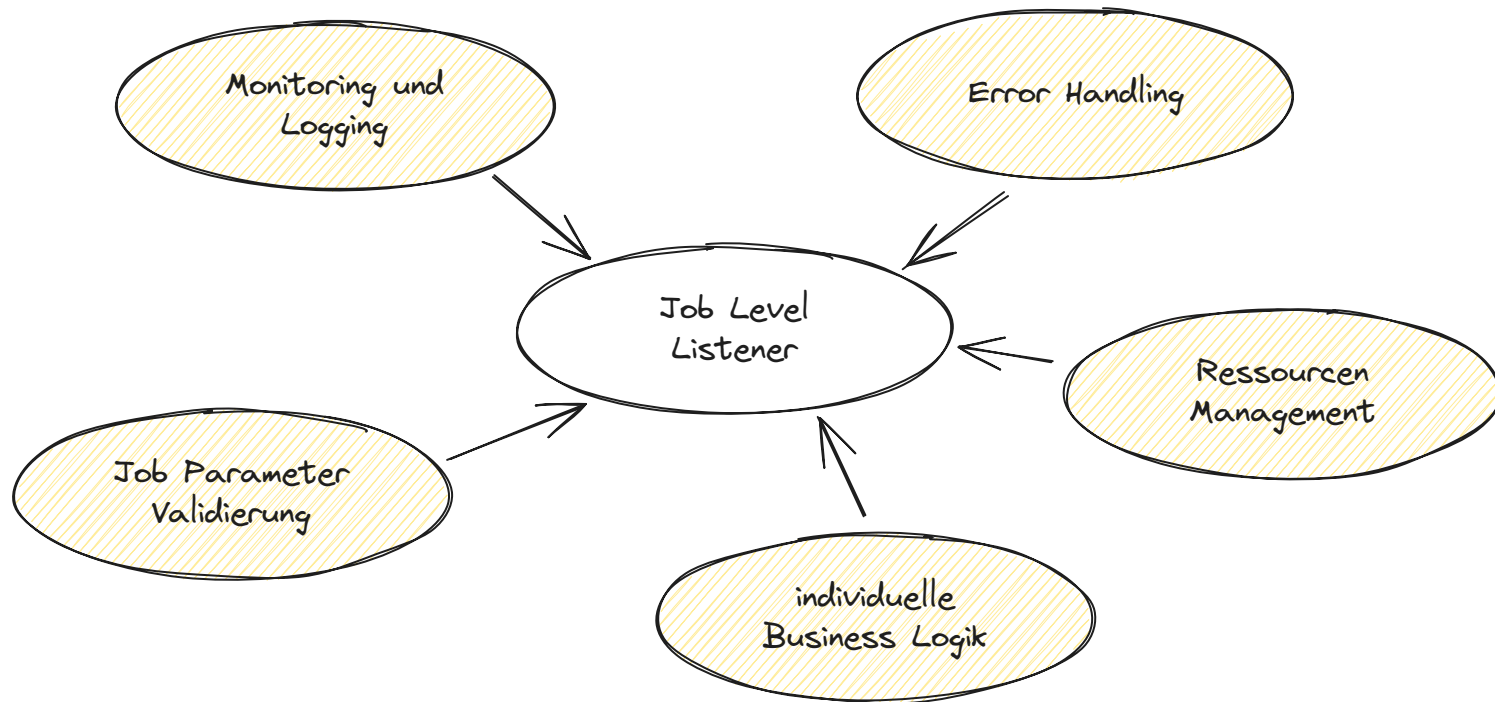
6. JOB SPECIFICATION LANGUAGE (= JSL)

- spezifiziert einen Job, seine Schritte und steuert seine Ausführung
- JSL für Jakarta Batch ist in XML implementiert und wird daher auch als "Job XML" genannt

6.1. JOB SYNTAX

```
<job id="name" restartable="true|false">
```

6.1.1. JOB LEVEL LISTENER



6.1.1. JOB LEVEL LISTENER

- werden genutzt, um auf den Lifecycle eines Batch Jobs zu reagieren und ihn zu beobachten
- ist der einzige Listener, der auf Jobebene agiert
- mehrere Listener können auf einen Job konfiguriert werden
- geordnete Reihenfolge nicht garantiert

```
<listeners>
  <listener ref="name">
    ...
</listeners>
```

6.1.2. JOB LEVEL LISTENER PROPERTIES

- wird genutzt, um Property Werte an den Job Listener weiterzugeben
- es kann eine beliebige Anzahl an Properties gesetzt werden

```
<properties>  
  <property name="property-name" value="name-value"></property>  
</properties>
```

6.2. STEP SYNTAX

- Steps können nicht nur nach einem Step ausgeführt werden, sondern nach einem Flow, einer Decision oder einem Split
- besitzt auch Step Level Listener und Step Level Properties

```
<step id="name"  
  start-limit="integer"  
  allow-start-if-complete="true|false"  
  next="flow-id|step-id|split-id|decision-id">
```

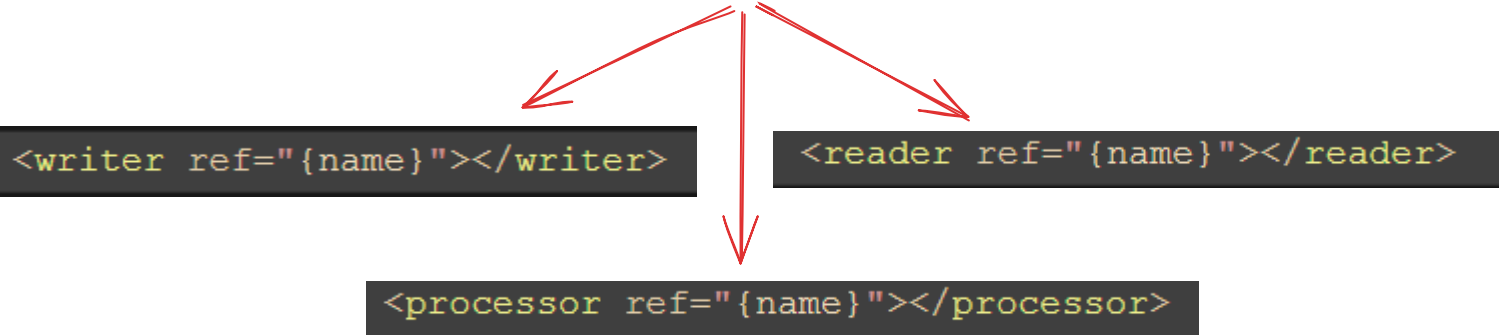
6.2.1. CHUNK STEP

```
<chunk checkpoint-policy="item|custom"  
  item-count="value"  
  time-limit="value"  
  skip-limit="value"  
  retry-limit="value"></chunk>
```

BEISPIEL: CHECKPOINT-POLICY

6.2.1.1. READER, PROCESSOR UND WRITER ELEMENT

- gibt den Namen des Batch Artifacts an



```
<writer ref="{name}"></writer>
```

```
<reader ref="{name}"></reader>
```

```
<processor ref="{name}"></processor>
```

6.2.1.2. READER, PROCESSOR UND WRITER PROPERTY

```
<properties>  
  <property name="property-name" value="name-value"></property>  
</properties>
```

6.2.1.3. CHUNK EXCEPTION HANDLING

- wenn während der Batch Laufzeit eine Exception geworfen wird, endet der Job mit dem Batch Status "FAILED"
- Reader, Processor und Writer können so konfiguriert werden, dass nach diesen Exceptions der Vorgang entweder neu gestartet oder übersprungen wird

6.2.1.4. SKIPPABLE EXCEPTION

```
<skippable-exception-classes>  
  <include class="java.lang.Exception"></include>  
  <exclude class="java.io.FileNotFoundException"></exclude>  
</skippable-exception-classes>
```

6.2.1.5. RETRYABLE EXCEPTION

```
<retryable-exception-classes>  
  <include class="java.io.IOException"></include>  
  <exclude class="java.io.FileNotFoundException"></exclude>  
</retryable-exception-classes>
```

6.2.1.6. NO-ROLLBACK EXCEPTION

```
<no-rollback-exception-classes>  
  <include class="class name"></include>  
  <exclude class="class name"></exclude>  
</no-rollback-exception-classes>
```

6.2.2. BATCHLET

```
<batchlet ref="name"></batchlet>
```

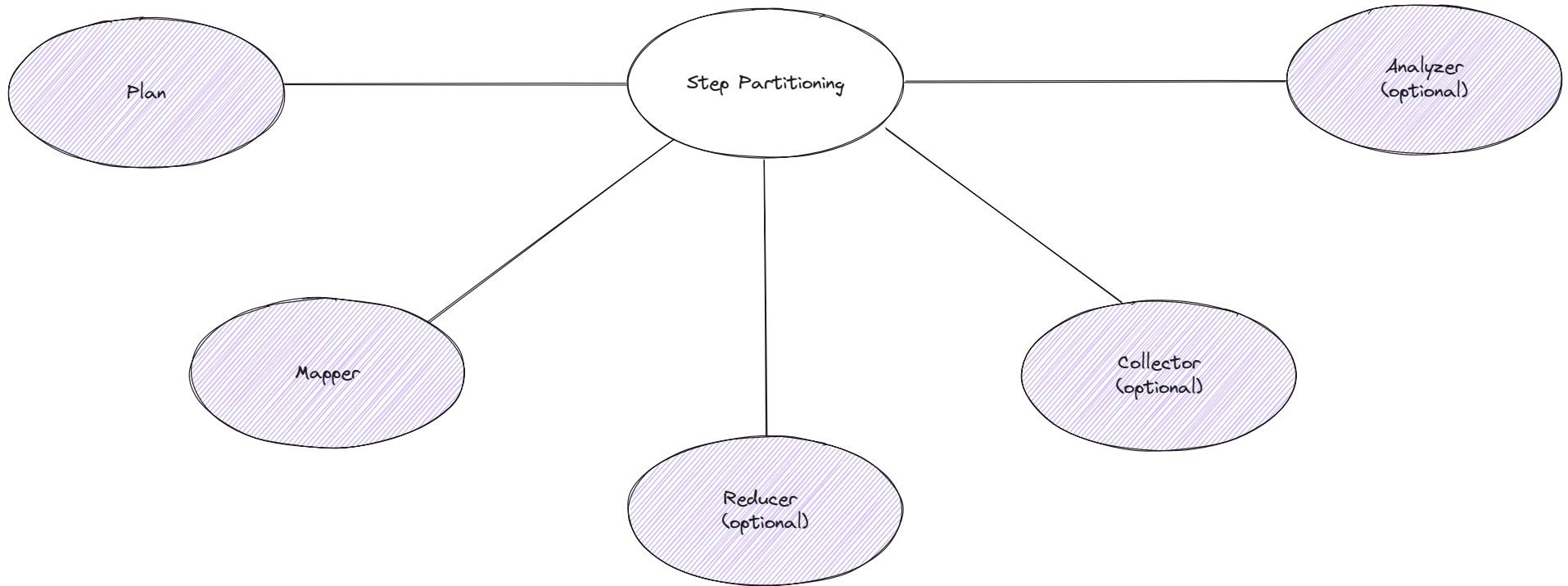
```
<properties>  
  <property name="property-name" value="name-value"></property>  
</properties>
```

6.2.3. STEP SEQUENCE

- falls die Reihenfolge der Steps wichtig ist, sollten sie sequenziell implementiert werden
- auch möglich für Flows, Splits oder Decisions

```
<next on="exit status" to="id" ></next>
```

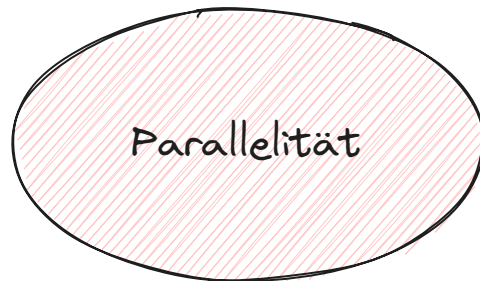
6.2.4. STEP PARTITIONING



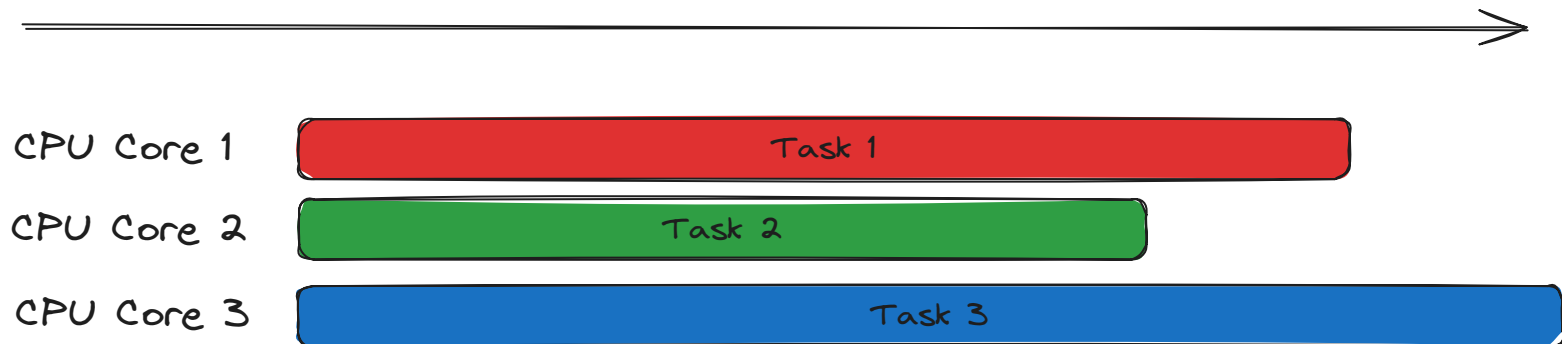
6.2.4. STEP PARTITIONING

```
<step id="Step1">  
  <chunk ...></chunk> or <batchlet ... ></batchlet>  
  <partition ...></partition>  
</step>
```

RÜCKBLICK: PARALLELITÄT



- mehrere Aufgaben werden zur selben Zeit bearbeitet



6.2.4.1. PARTITION PLAN

```
<step id="Step1">  
  <chunk ...></chunk>  
  <partition>  
    <plan partitions="3" threads="2"></plan>  
  </partition>  
</step>
```

6.2.4.2. PARTITION PROPERTY

```
<properties partition="{partition-number}">  
  <property name="{property-name}" value="{name-value}"></prop  
</properties>
```

6.2.4.2. PARTITION MAPPER

```
<partition>  
  <mapper ref="MyStepPartitioner"></mapper>  
</partition>
```

6.2.4.3. PARTITION COLLECTOR

```
<partition>  
  <collector ref="MyStepCollector"></collector>  
</partition>
```

6.2.4.4. PARTITION ANALYZER

```
<partition>  
  <analyzer ref="MyStepAnalyzer"></analyzer>  
</partition>
```

6.2.4.5. PARTITION REDUCER

```
<partition>  
  <reducer ref="MyStepPartitionReducer"></reducer>  
</partition>
```

BEISPIEL: STEP PARTITIONING

AUFGABE: STEP PARTITIONING

- Collector: übergibt den `StepContext.getPersistentData` an den Analyzer
- Analyzer: `StepContext injecten` -> wie viele Kunden leben in München
- Reducer: Daten aus `StepContext` sollen von einer `HashMap` zu einer Liste verarbeitet werden, wobei wir nur Kunden haben wollen, die "Schwarz" als Nachnamen haben
- `MyItemProcessor`: alle Münchner in die `HashMap` laden und in `StepContext` speichern
- `batchProcessing.xml`: `partition` implementieren
- Wie viele Kunden haben wir, die in München wohnen und "Schwarz" als Nachnamen haben
- Im Analyzer angeben wie viele Münchner wir haben
- Alle Kunden in `beforePartitionCompletion` im Reducer ausgeben lassen

6.2.5. FLOW

```
<flow id="{name}" next="{flow-id|step-id|split-id|decision-id}"  
  <step> ... </step>  
</flow>
```

6.2.6 DECISION

```
<decision id="{name}" ref="{ref-name}">
```

6.2.7. SPLIT

```
<split id="{name}" next="{flow-id|step-id|split-id|decision-id}">  
  <flow> ... </flow>  
  ...  
</split>
```

7. WILDFLY SERVER MIT CDI BEISPIEL

AUFGABE: WILDFLY SERVER MIT CDI

- Chunk Batchverarbeitung mit customer.csv möglich machen
- auf Wildfly deployen
- über H2Console auf Datenbank zugreifen

Quellenangaben:

- <https://jakarta.ee/specifications/batch/2.1/>
- <https://docs.wildfly.org/34/>

