

1. Einführung
2. Spring-Batch Infrastruktur
3. Spring-Batch Repository
4. Job Ausführung
5. Chunkverarbeitung und Fehlerbehandlung
6. Konfiguration
7. Parallelisierung
8. Operations

# Batchverarbeitung

## Was versteht man darunter?

# Herkunft

- Historisch: Stapelverarbeitung (Lochkarten)
- Windows: .BAT Dateien
- Großer Stil: Operator verwaltet Jobs im Rechenzentrum
- Batchverarbeitung vs. Dialogverarbeitung vs. Messaging

# Wodurch zeichnet sich Batchverarbeitung aus?

Batch-Programme ...

- laufen ohne Benutzerinteraktion
- verarbeiten eine vorgegebene Datenmenge
- enden nachdem alle Daten verarbeitet wurden
- eignen sich nur, wenn Verzögerung akzeptabel ist

# Antorderungen

Batch-Programme ...

- müssen große Datenmengen verarbeiten können
- müssen ausreichende Performance liefern
- müssen mit Fehlern klarkommen
  - defekte Eingangsdaten
  - (temporäre) Ausfälle
- müssen Wideranlauf unterstützen
- müssen überwacht werden können

# Spring-Batch

**Hungry?** Goes great with grilled asparagus, ham and cheese panini, veggie kabobs, or bacon-wrapped scallops. Or enjoy it with fresh berries.



**MOTHER'S BREWING COMPANY**

**SPRING BATCH**

-A PROTOTYPE ALE-

BREWED AND BOTTLED BY  
MOTHER'S BREWING CO.  
SPRINGFIELD, MO.

5.8% ALC. BY VOL.  
58 IBU  
12 FL OZ.

PLEASE RECYCLE IA OR DE MA NY CT ME VT DC OK 5¢  
DEP MI 10¢ DEP CA REDEMPTION VALUE OK + COL.

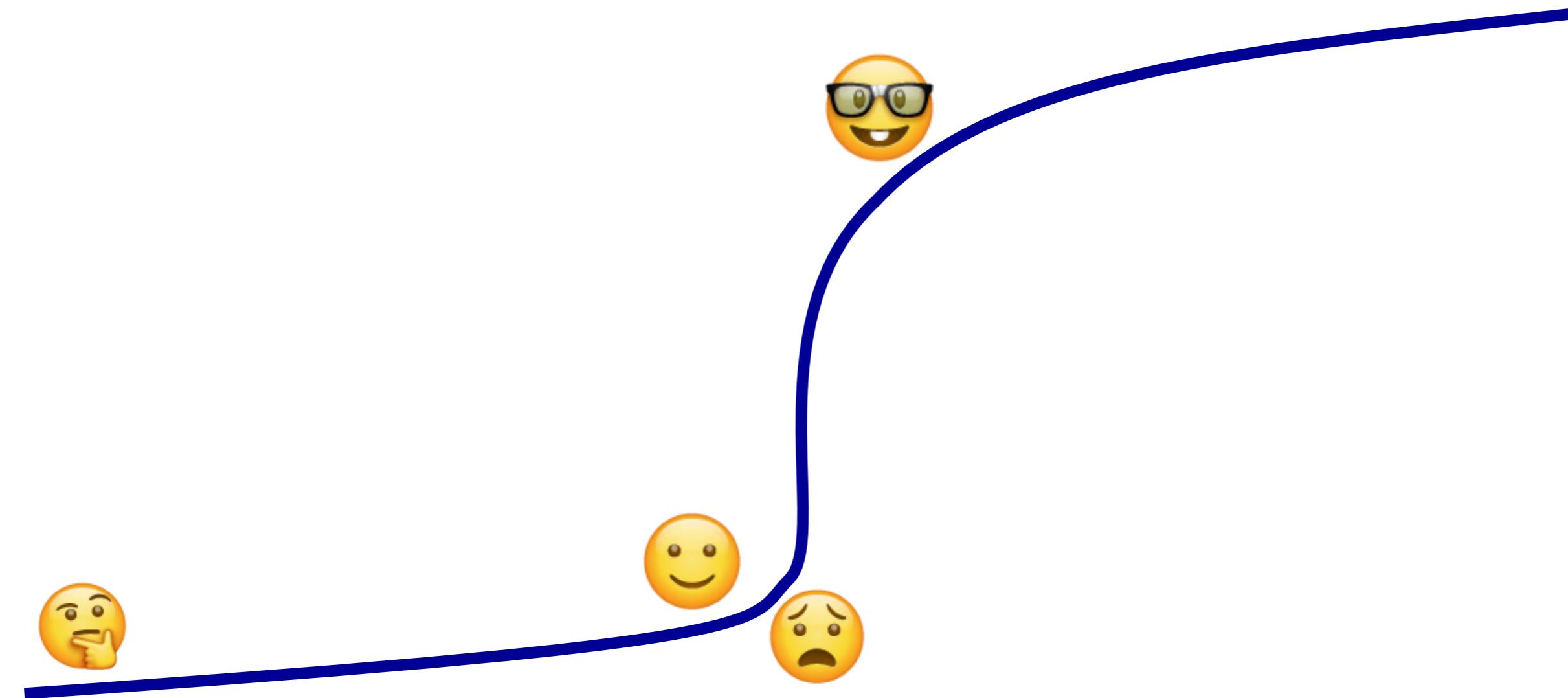
8 51666003085

ADULTS ONLY

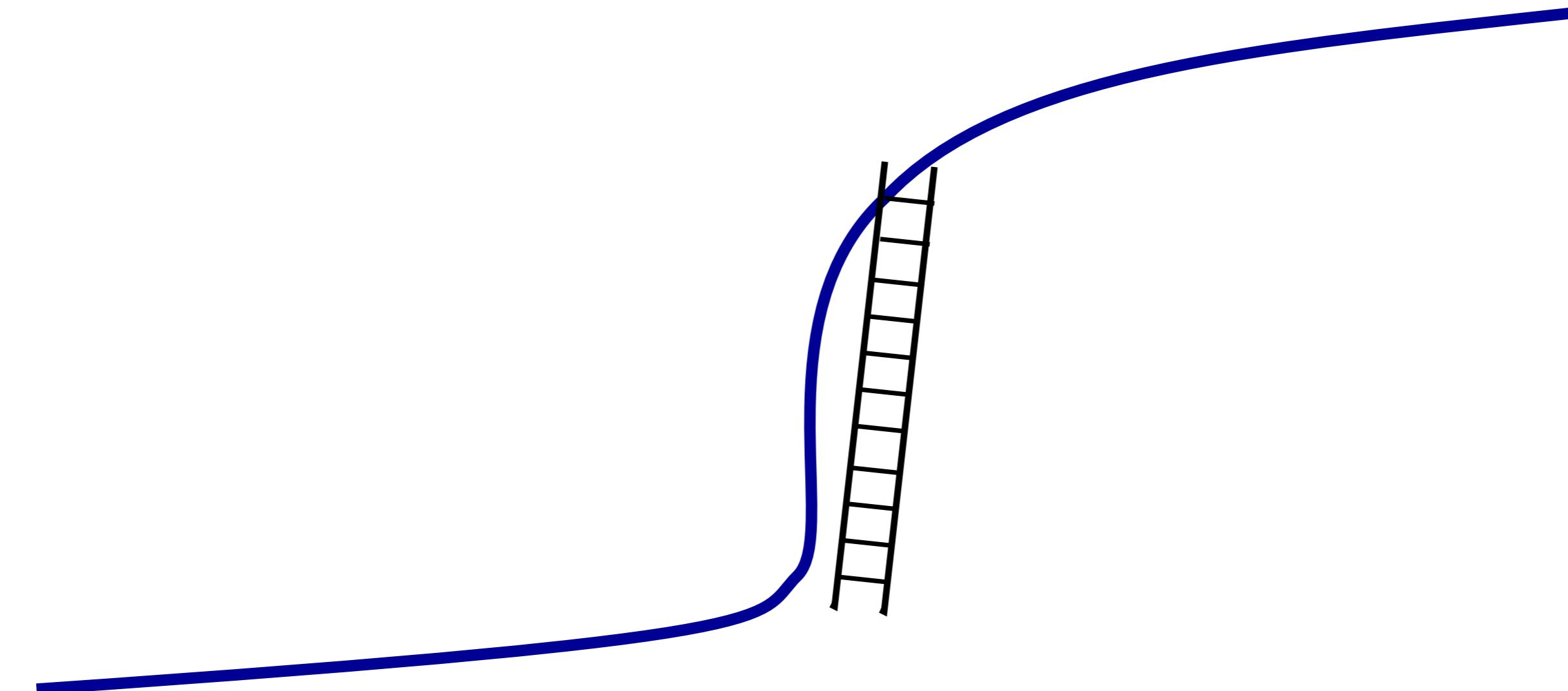
# Historie

- 2007: Code-Beitrag von Accenture
- 2008: Spring-Batch-1.0.0.FINAL (Java 4)
- 2009: 2.0.0.RELEASE (Java 5, Generics, Flows, Remote Chunking, Partitioning)
- 2014: 3.0.0.RELEASE (JSR-352, Spring 4, Java 8)
- 2017: 4.0.0.RELEASE (Spring 5, benötigt Java 8, Builder für Reader/Writer)

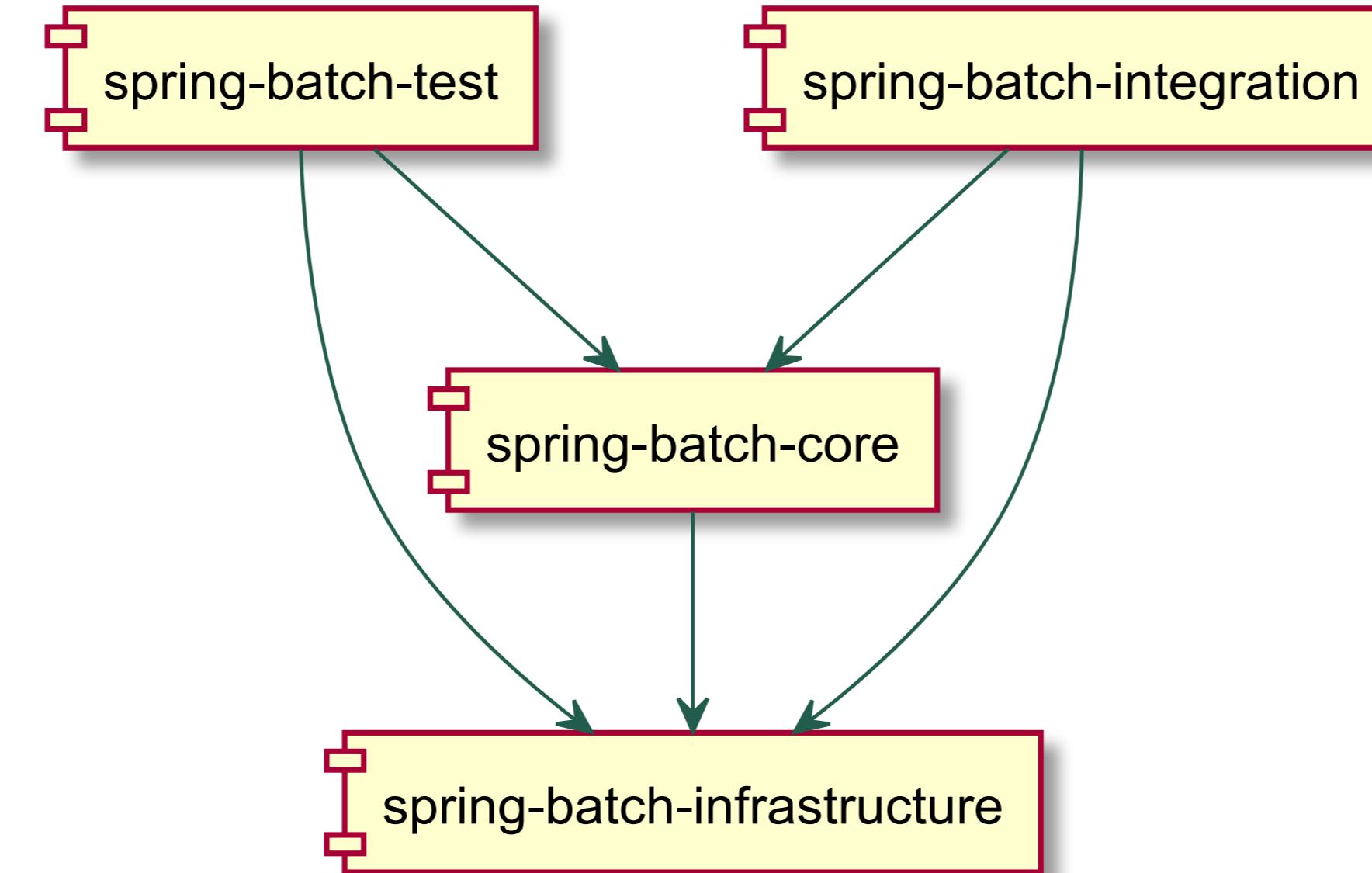
# Lernkurve



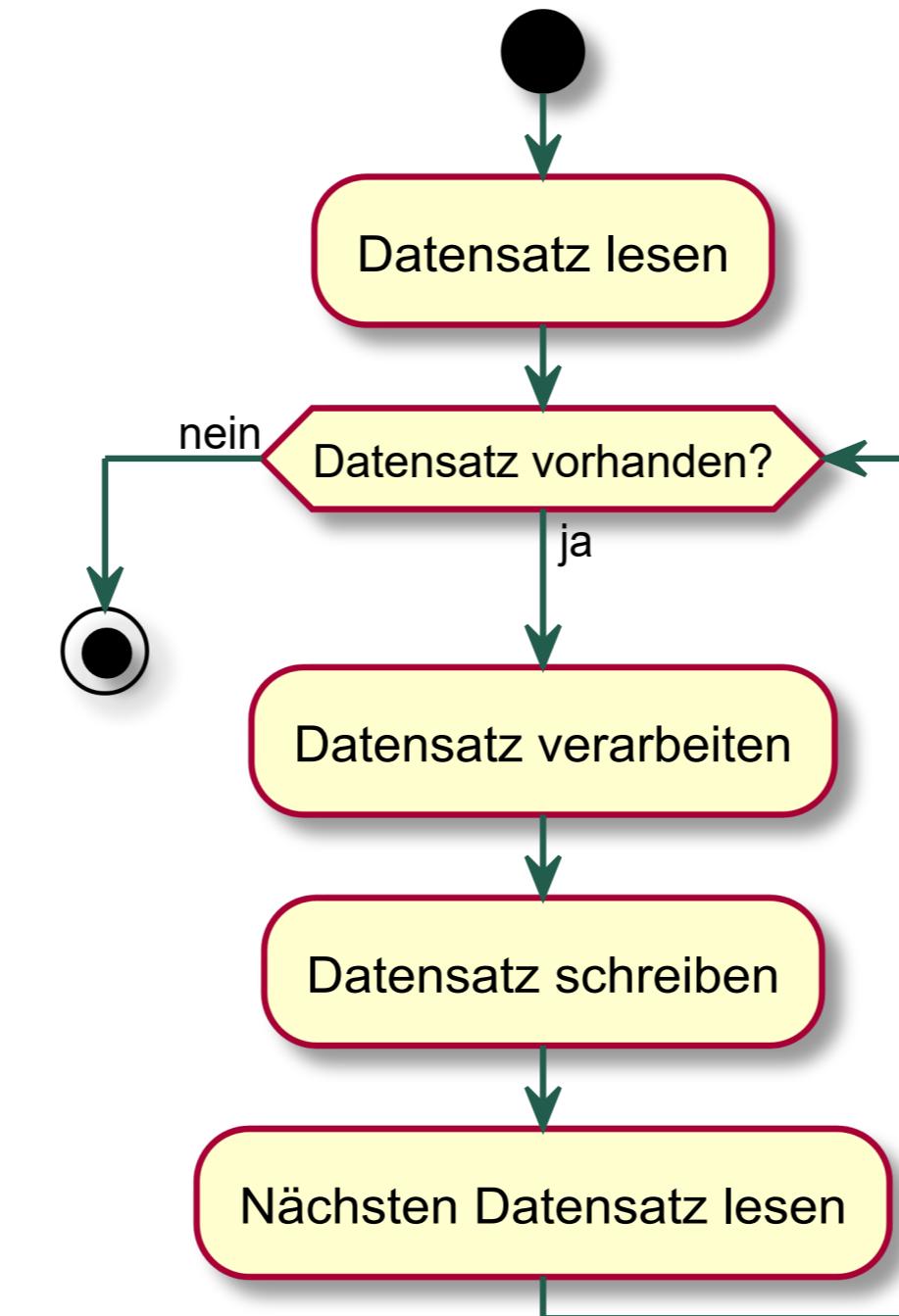
# Ziel für heute...



# Architektur (JARs)



# Typischer Ablauf eines Batch-Jobs

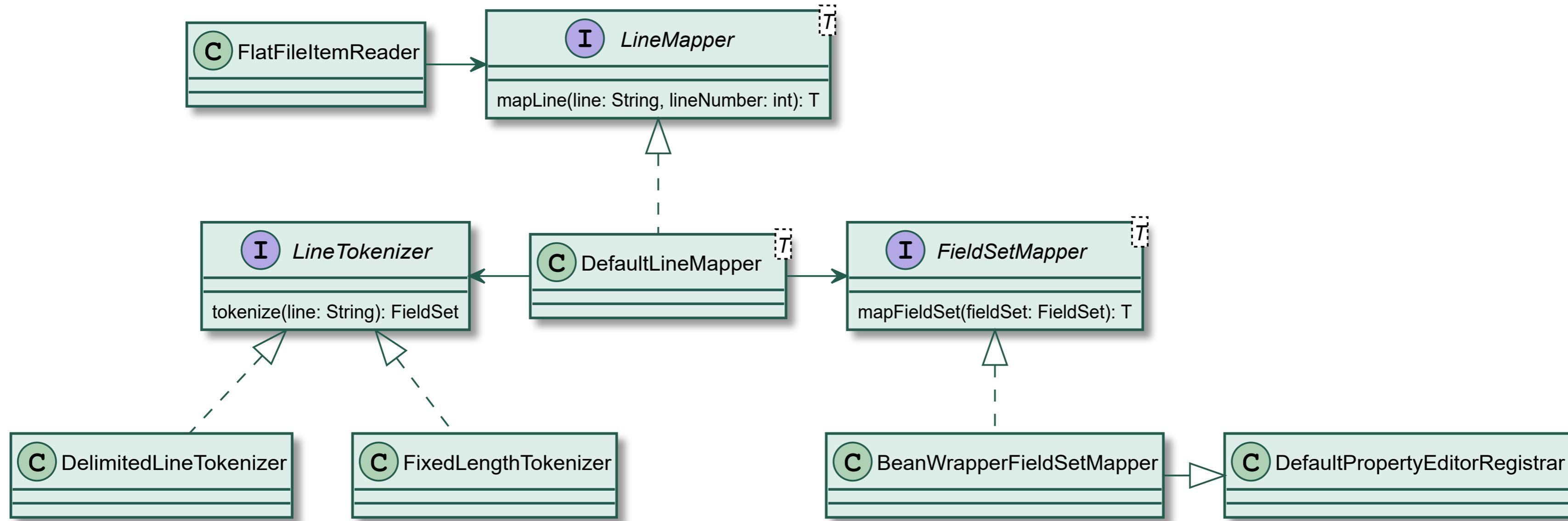


# Spring-Batch Infrastruktur

## Reader und Writer

- Flat Files (delimited, fixed size)
- XML/JSON Files (große Dateien, StAX+OMX fragmentweise)
- Multi-File Input
- Datenbanken (JDBC, JPA, Cursor-based/Paging, NoSQL, LDAP)
- Messaging (JMS, AMQP)
- Diverse Wrapper und Adapter (`IItemReader`, `SynchronizedItemStreamReader`, JSR-352)

# FlatFileItemReader



# Beispiel: CSV Datei einlesen

*sample-data.csv*

```
Vorname; Name  
Anna; Gramm  
Franz; Brandwein  
Izmir; Egal
```

*Person.java*

```
@Data  
public class Person {  
    private String firstName;  
    private String lastName;  
}
```

# FlatFileItemReader erzeugen

## FlatFileItemReaderTest.java

```
private FlatFileItemReader<Person> createPersonReader() {  
    FlatFileItemReader<Person> reader = new FlatFileItemReaderBuilder<Person>()  
        .name("personReader") // alternativ: .saveState(false)  
        .resource(new ClassPathResource("sample-data.csv"))  
        .delimited().delimiter(";")  
        .names(new String[] { "firstName", "lastName" }) // LineTokenizer  
        .fieldSetMapper(new BeanWrapperFieldSetMapper<Person>() {  
            {  
                setTargetType(Person.class);  
            }  
        })  
        .linesToSkip(1)  
        .build();  
    return reader;  
}
```

Seit Spring-Batch 4.x gibt es Builder für alle Reader und Writer

# FlatFileItemReader verwenden

```
private List<Person> readPersons(ItemStreamReader<Person> reader) throws Exception {  
    List<Person> persons = new ArrayList<>();  
    Person person;  
    ExecutionContext ec = new ExecutionContext(); // Damit speichert der Reader seinen Fortschritt  
    reader.open(ec);  
    while ((person = reader.read()) != null) {  
        persons.add(person);  
    }  
    reader.close();  
    return persons;  
}
```

⇒ Reader und Writer lassen sich auch ohne Spring-Batch-Framework einsetzen

# Spring-Batch Repository

## Wozu ein Repository?

Note MapJobRepository kann als Platzhalter verwendet werden

# Spring-Batch Repository

## Wozu ein Repository?

- Protokollierung

Note MapJobRepository kann als Platzhalter verwendet werden

# Spring-Batch Repository

## Wozu ein Repository?

- Protokollierung
- Wiederanlauf

Note MapJobRepository kann als Platzhalter verwendet werden

# Spring-Batch Repository

## Wozu ein Repository?

- Protokollierung
- Wiederanlauf
- Überwachung

Note MapJobRepository kann als Platzhalter verwendet werden

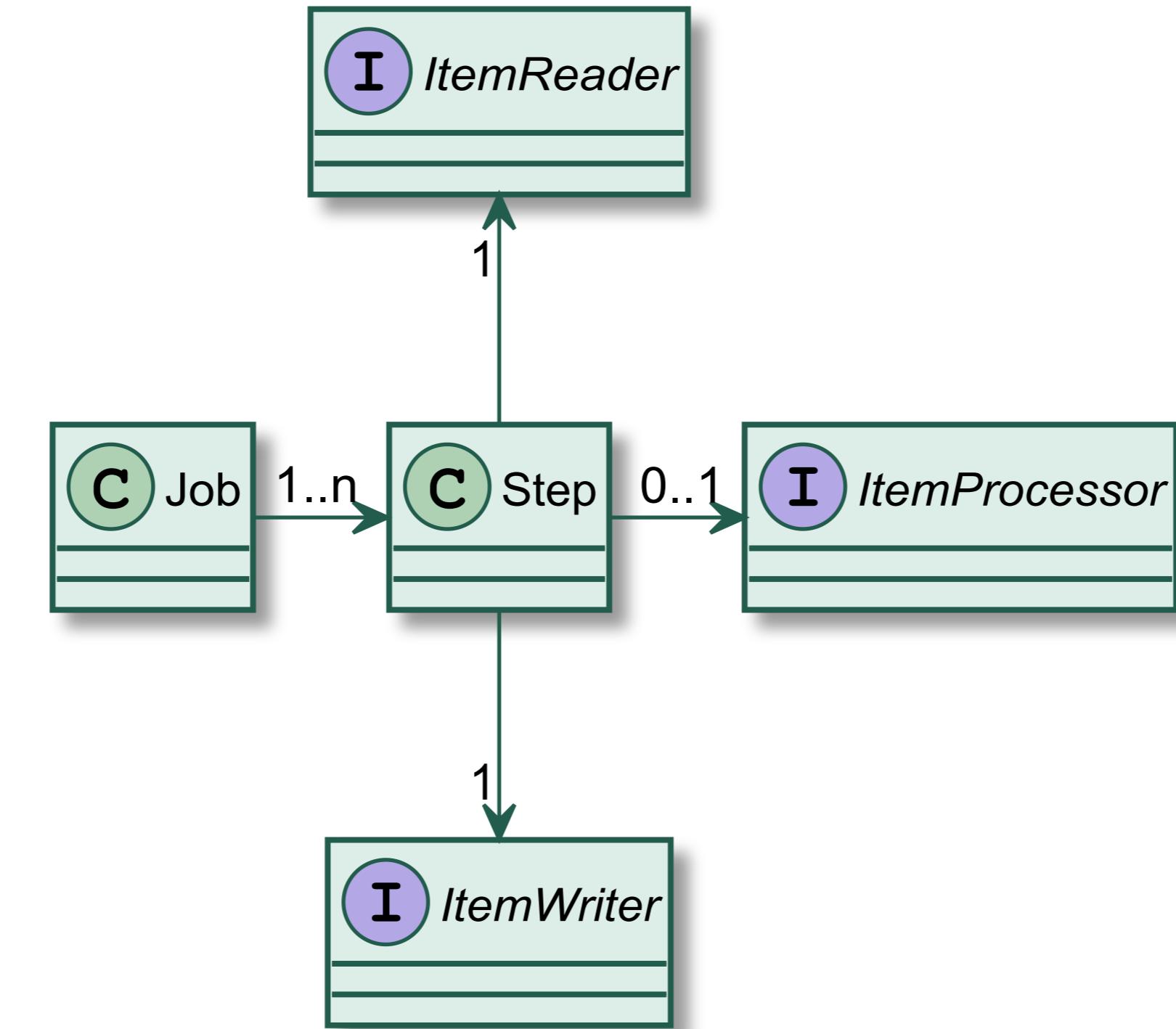
# Spring-Batch Repository

## Wozu ein Repository?

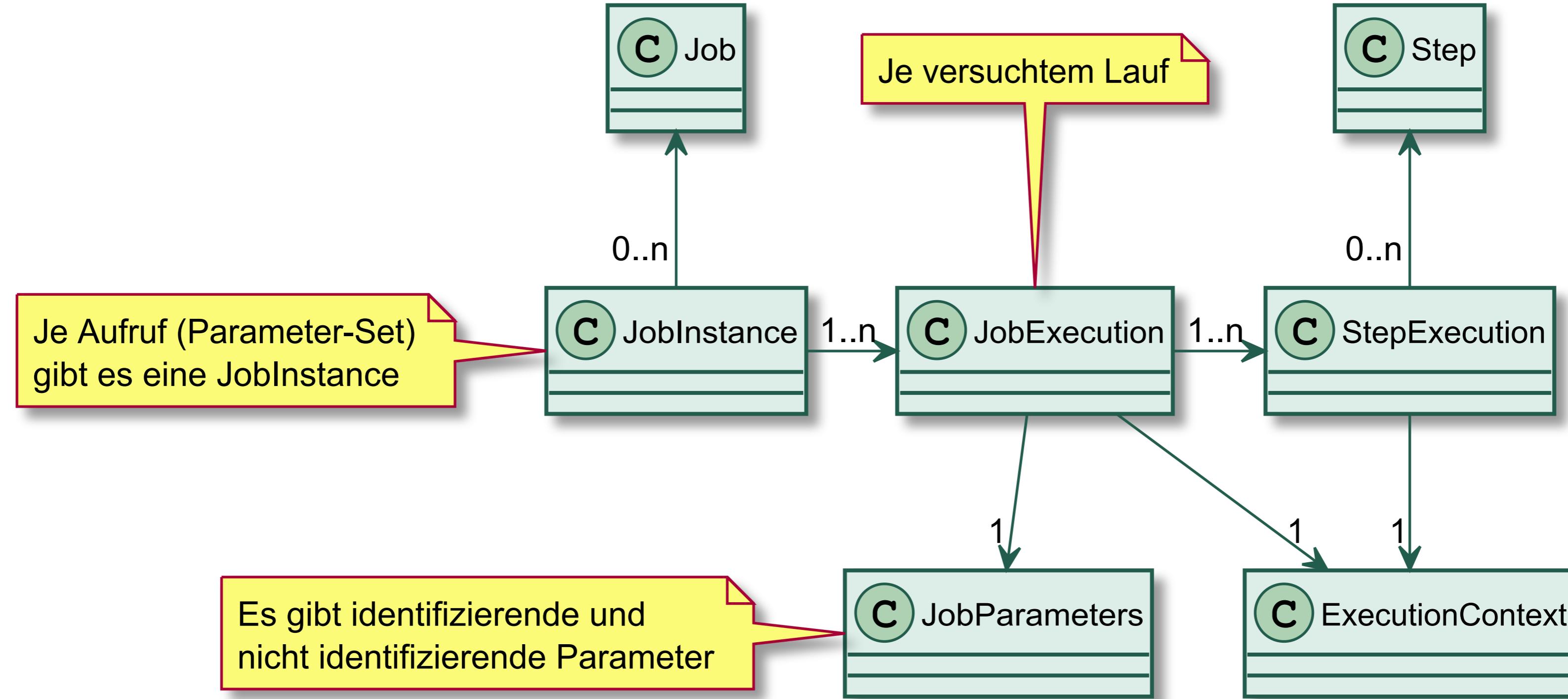
- Protokollierung
- Wiederanlauf
- Überwachung
- Steuerung

Note MapJobRepository kann als Platzhalter verwendet werden

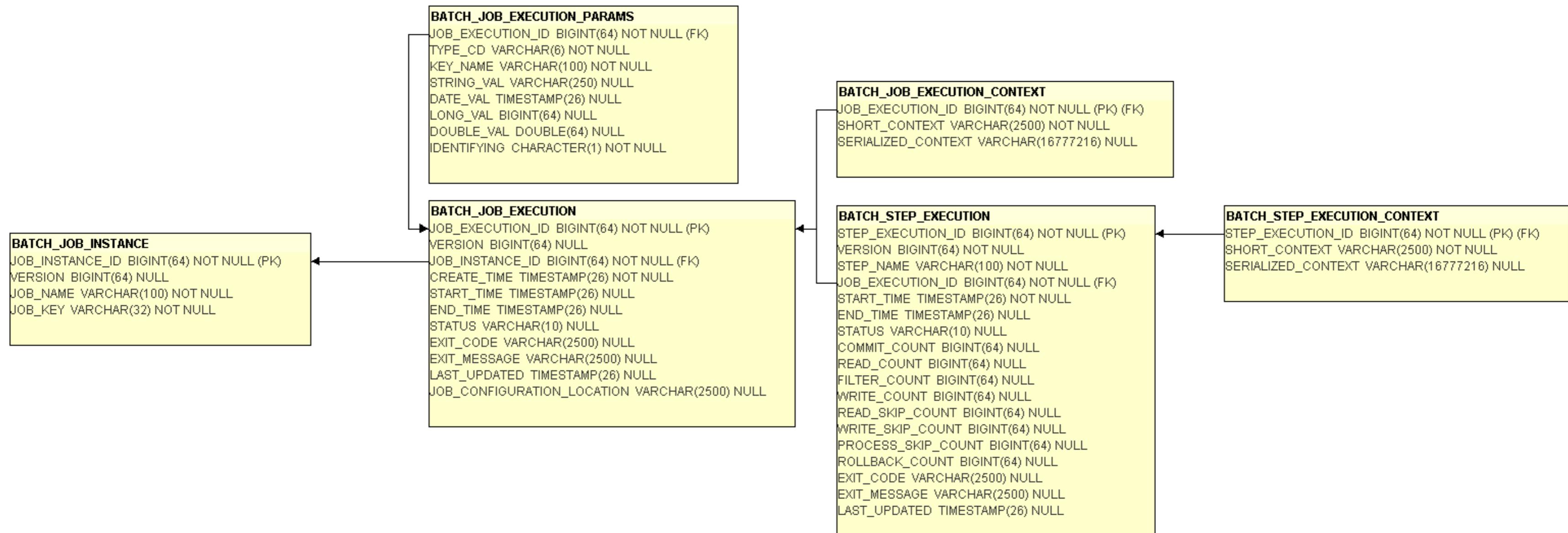
# Aufbau eines Batch-Jobs



# Lautzeit-Datenmodell

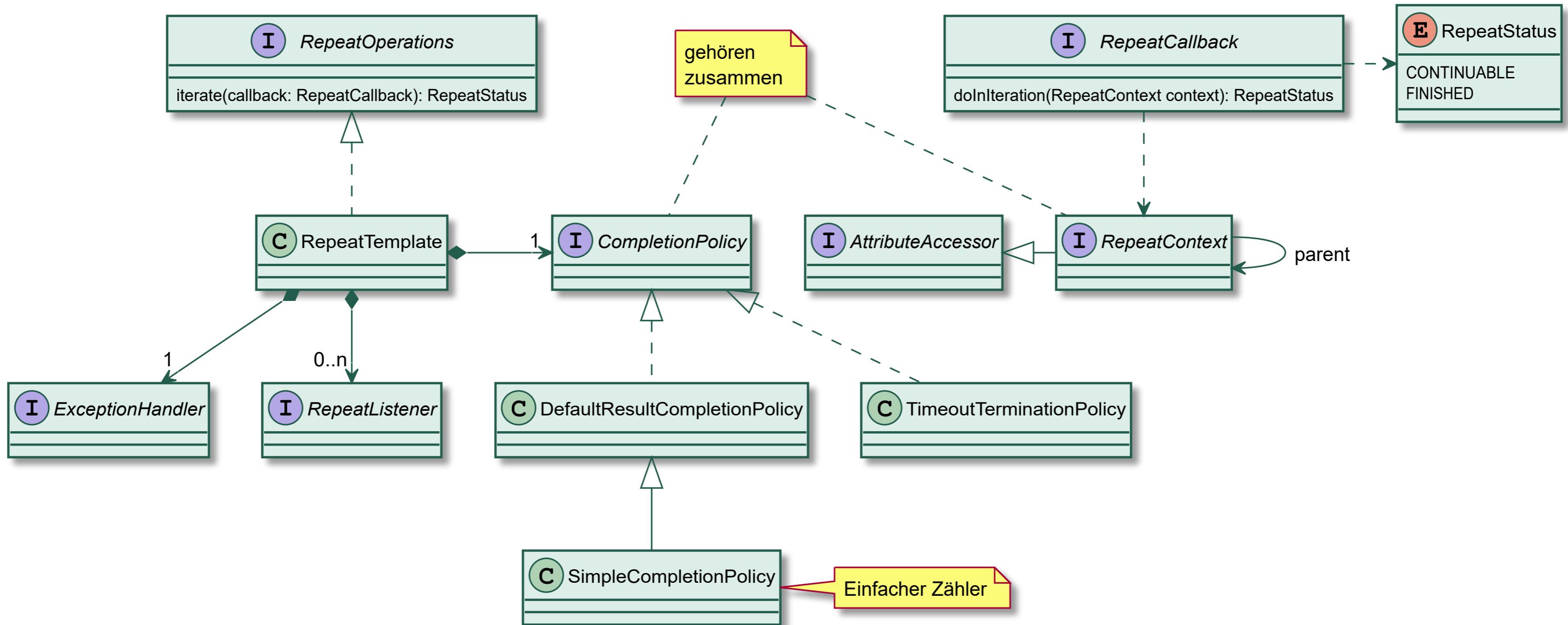


# Datenbank-Schema

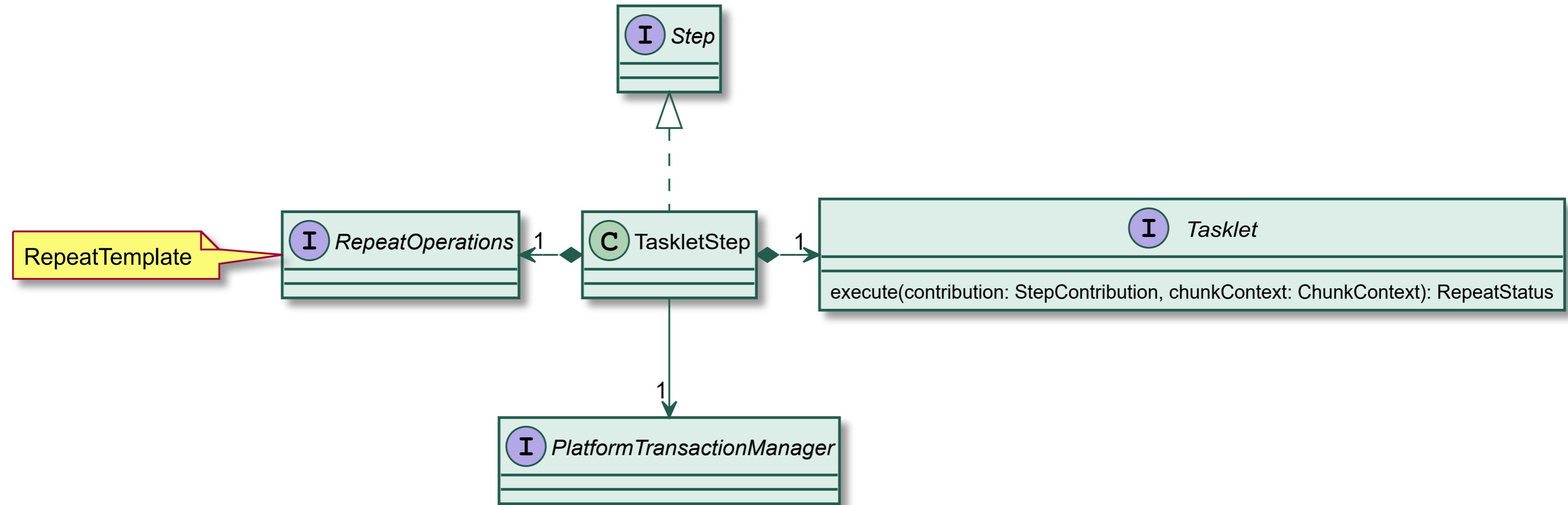


# Job Ausführung

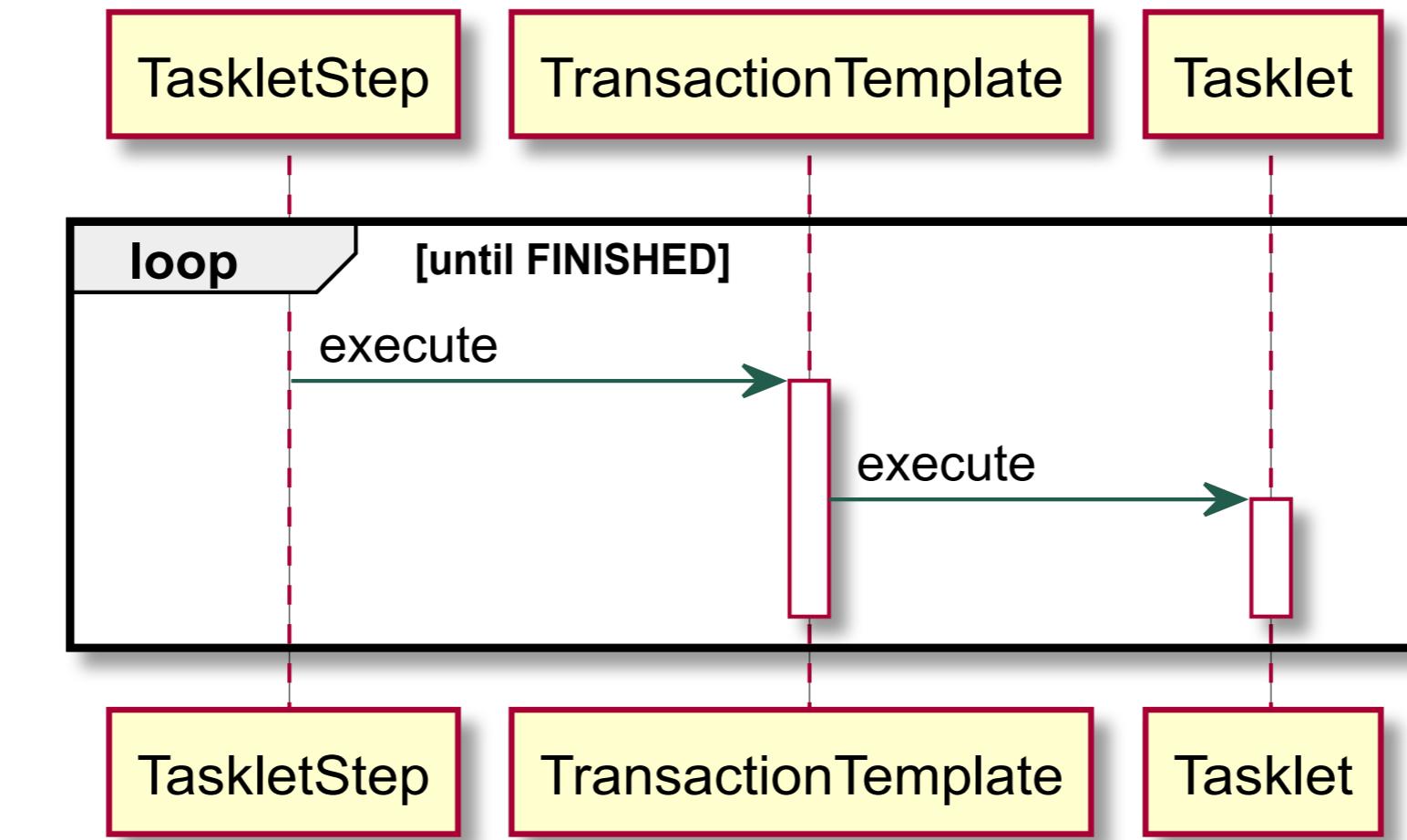
## RepeateTemplate – Ein Framework für eine Schleife?



# TaskletStep



# Tasklet ausführen



# Hello-World Job mit einem TaskletStep

## HelloJobConfig.java

```
@Bean
Job helloJob() {
    var tasklet = new Tasklet() {

        @Override
        public RepeatStatus execute(StepContribution contribution, ChunkContext chunkContext) {
            System.out.println("Hello World!");
            return RepeatStatus.FINISHED;
        }
    };
    TaskletStep step = stepBuilderFactory
        .get("helloStep")
        .tasklet(tasklet)
        .build();
    Job job = jobBuilderFactory
        .get("helloJob")
        .start(step)
        .build();
    return job;
}
```

# Rahmen für Job-Konfiguration

## HelloJobConfig.java

```
@Configuration  
@EnableBatchProcessing  
public class HelloJobConfig {  
    @Autowired  
    private StepBuilderFactory stepBuilderFactory;  
    @Autowired  
    private JobBuilderFactory jobBuilderFactory;  
}
```

# Job austuhren (Unit-test)

```
@SpringJUnitConfig(HelloJobConfig.class)
public class HelloJobTest {

    @Autowired
    private JobLauncher jobLauncher;

    @Autowired
    private Job helloJob;

    @Test
    void testHello() throws JobParametersInvalidException, JobExecutionAlreadyRunningException,
        JobRestartException, JobInstanceAlreadyCompleteException {
        jobLauncher.run(helloJob, new JobParameters());
    }
}
```

# Job austuhren (CommandLineJobRunner)

```
<configuration>
    <mainClass>org.springframework.batch.core.launch.support.CommandLineJobRunner</mainClass>
    <arguments>
        <argument>com.anderscore.goldschmiede.springbatch.samples.exec.HelloJobConfig</argument>
        <argument>helloJob</argument>
    </arguments>
</configuration>
```

# Job manuell zusammenstellen

```
@Test
public void testManualJobConfig() throws Exception {
    // prepare infrastructure
    var jobRepositoryFactoryBean = new MapJobRepositoryFactoryBean();
    var jobRepository = jobRepositoryFactoryBean.getObject();
    var transactionManager = new ResourcelessTransactionManager();
    jobBuilderFactory = new JobBuilderFactory(jobRepository);
    stepBuilderFactory = new StepBuilderFactory(jobRepository, transactionManager);

    // create the job
    Job job = helloJob();

    // create a launcher
    var launcher = new SimpleJobLauncher();
    launcher.setJobRepository(jobRepository);
    launcher.afterPropertiesSet();

    // launch the job
    var params = new JobParameters();
    JobExecution execution = launcher.run(job, params);

    // make sure job finished successfully
    assertThat(execution.getAllFailureExceptions()).isEmpty();
    assertThat(execution.getExitStatus()).isEqualTo(ExitStatus.COMPLETED);
}
```

# Chunkverarbeitung und Fehlerbehandlung

## Wozu Chunkverarbeitung?

# Chunkverarbeitung und Fehlerbehandlung

## Wozu Chunkverarbeitung?

- Transaktion zu groß (wenn der ganze Job in einer Transaktion läuft)
  - Rollback Segment läuft über
  - Bei Fehler wird alles zurückgerollt

# Chunkverarbeitung und Fehlerbehandlung

## Wozu Chunkverarbeitung?

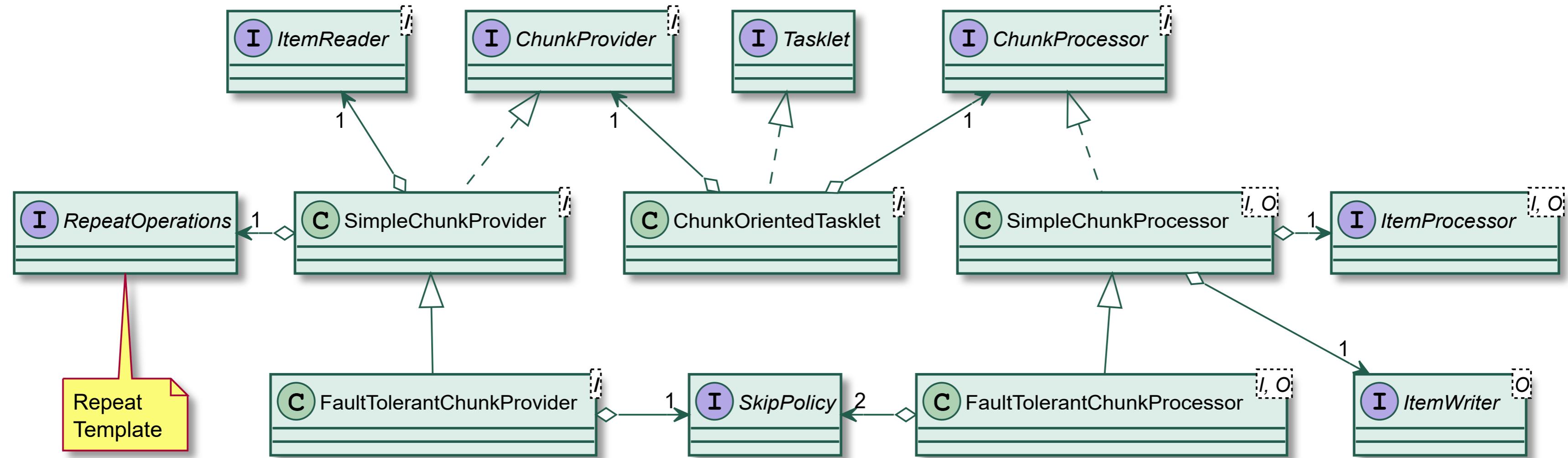
- Transaktion zu groß (wenn der ganze Job in einer Transaktion läuft)
  - Rollback Segment läuft über
  - Bei Fehler wird alles zurückgerollt
- Transaktionen zu klein (wenn jeder Datensatz in einer Transaktion verarbeitet wird)
  - Performance
  - Spring-Batch aktualisiert Status mit jeder Transaktion

# Chunkverarbeitung und Fehlerbehandlung

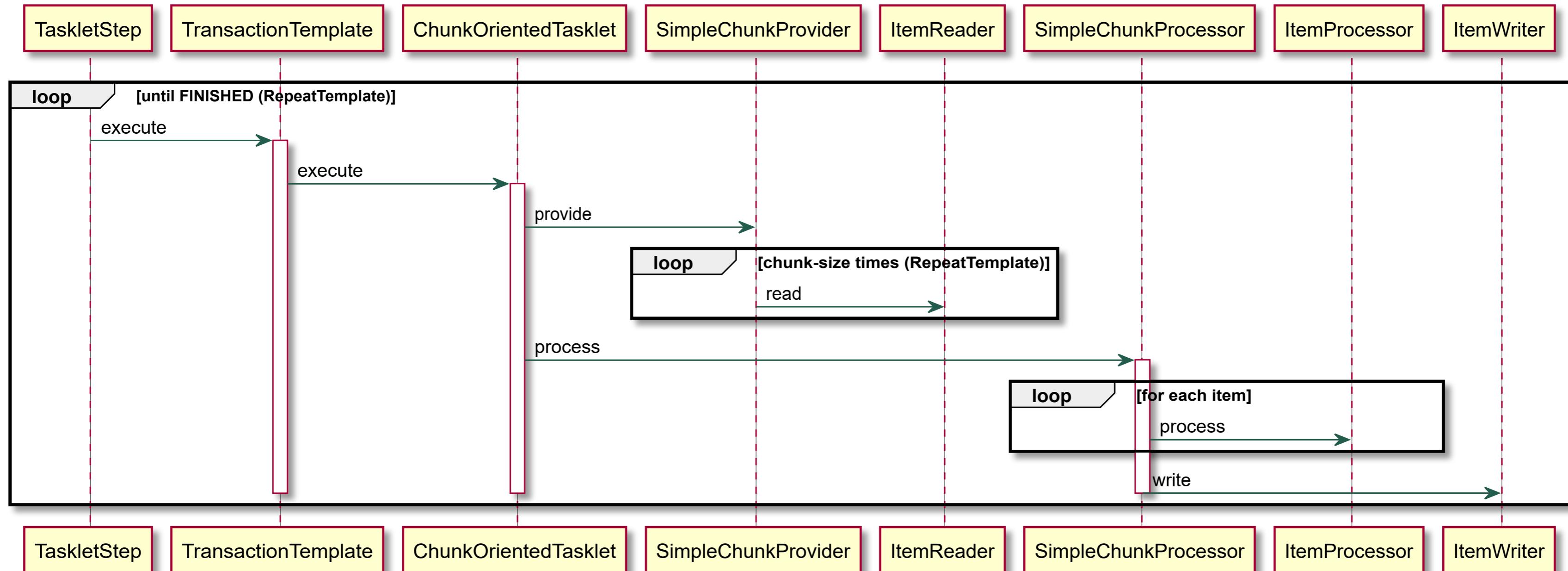
## Wozu Chunkverarbeitung?

- Transaktion zu groß (wenn der ganze Job in einer Transaktion läuft)
  - Rollback Segment läuft über
  - Bei Fehler wird alles zurückgerollt
- Transaktionen zu klein (wenn jeder Datensatz in einer Transaktion verarbeitet wird)
  - Performance
  - Spring-Batch aktualisiert Status mit jeder Transaktion
- ⇒ Lösung: Daten in Chunks verarbeiten

# ChunkOriented Tasklet



# Chunkverarbeitung



# Reader/Processor/Writer

- Reader: Rückgabewert `null` signalisiert Ende
- Processor: Rückgabewert `null` überspringt Datensatz (Filter)
- Verhalten von Spring-Batch im Ausnahmefall wird über Exceptions gesteuert

# Welche Fehler gibt es?

Und wie soll damit umgegangen werden?

- Datensatz nicht lesen
- Infrastrukturproblem
- Out of Memory
- Inkonsistente Daten

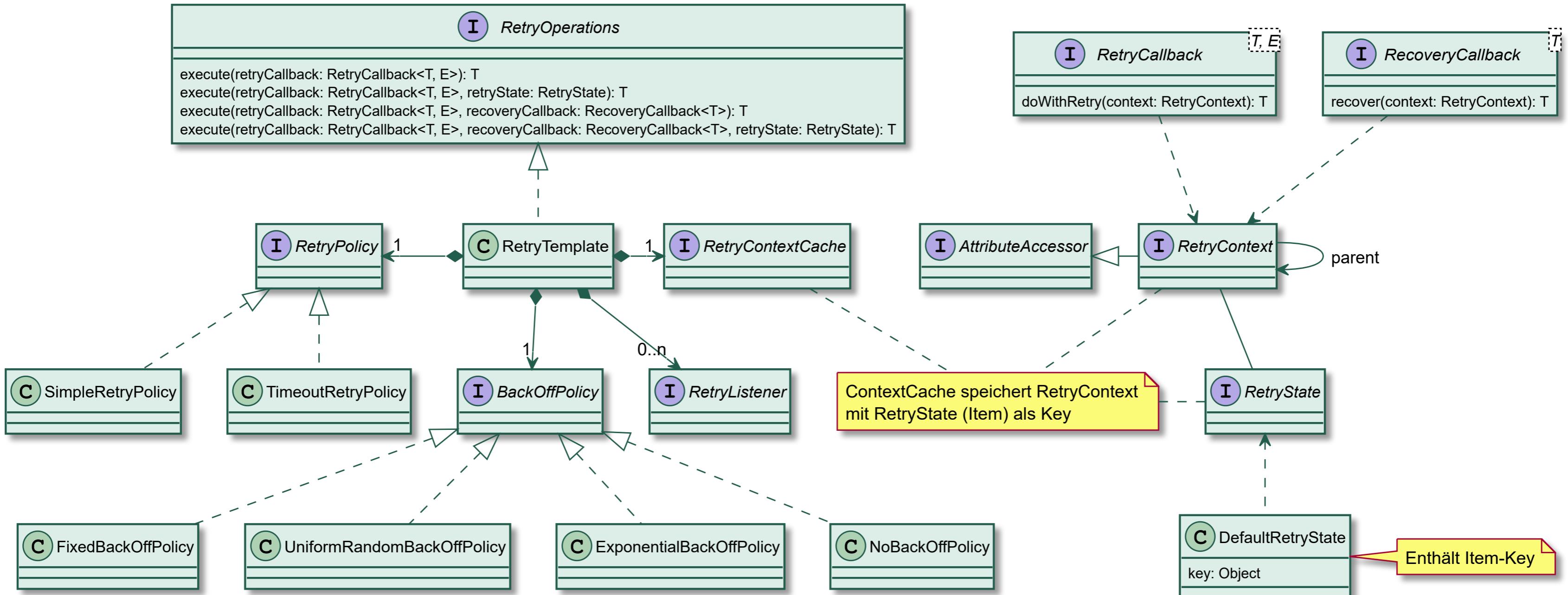
# Fehler und wünschenswertes Fehlerverhalten

Datensatz überbringen	Erneut versuchen	Job abbrechen
Programmierfehler	Verbindungsunterbrechung	Systemausfall
Ungültiges Zeichen in Datensatz	Neustart eines Servers	Fehlende Berechtigung
Datei defekt	Datensatz gesperrt	Nicht berücksichtigte Exception

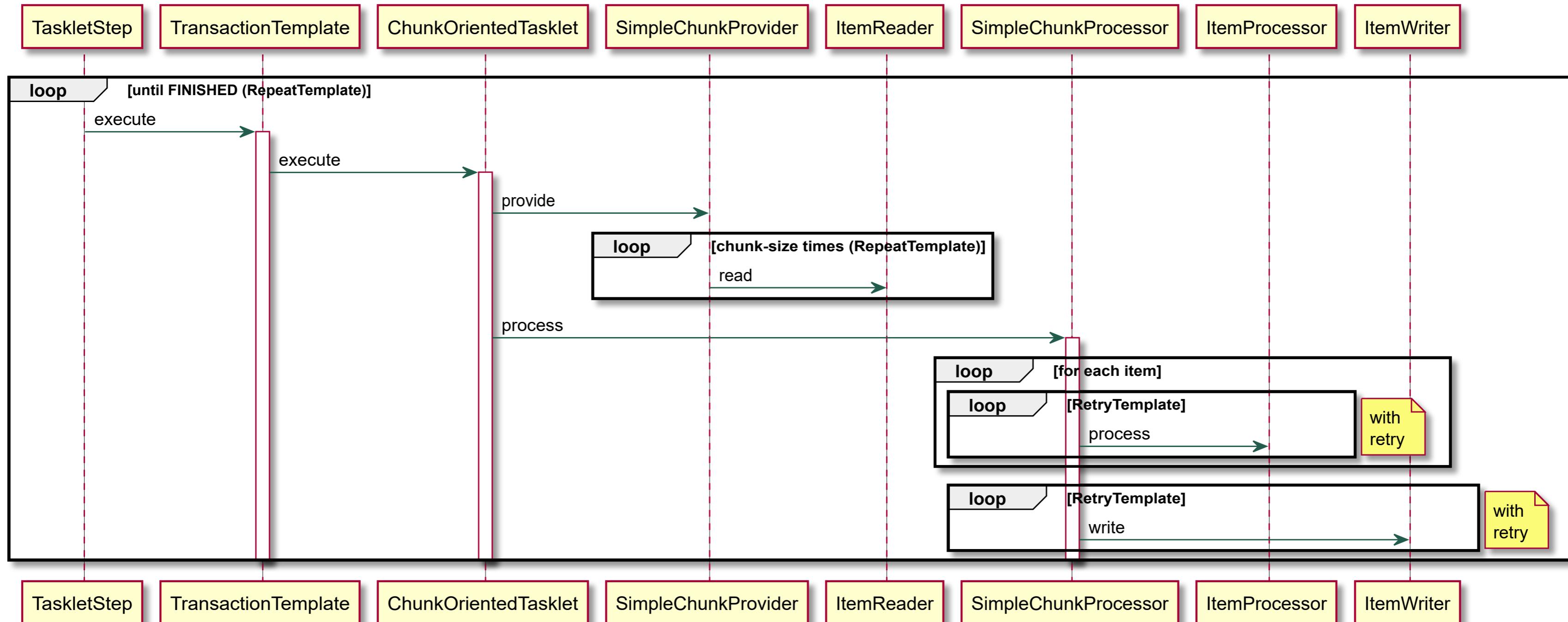
# Überspringen von Datensätzen

Demo...

# Retry Template – Noch ein Framework für eine Schleife...



# Chunkverarbeitung mit Retry



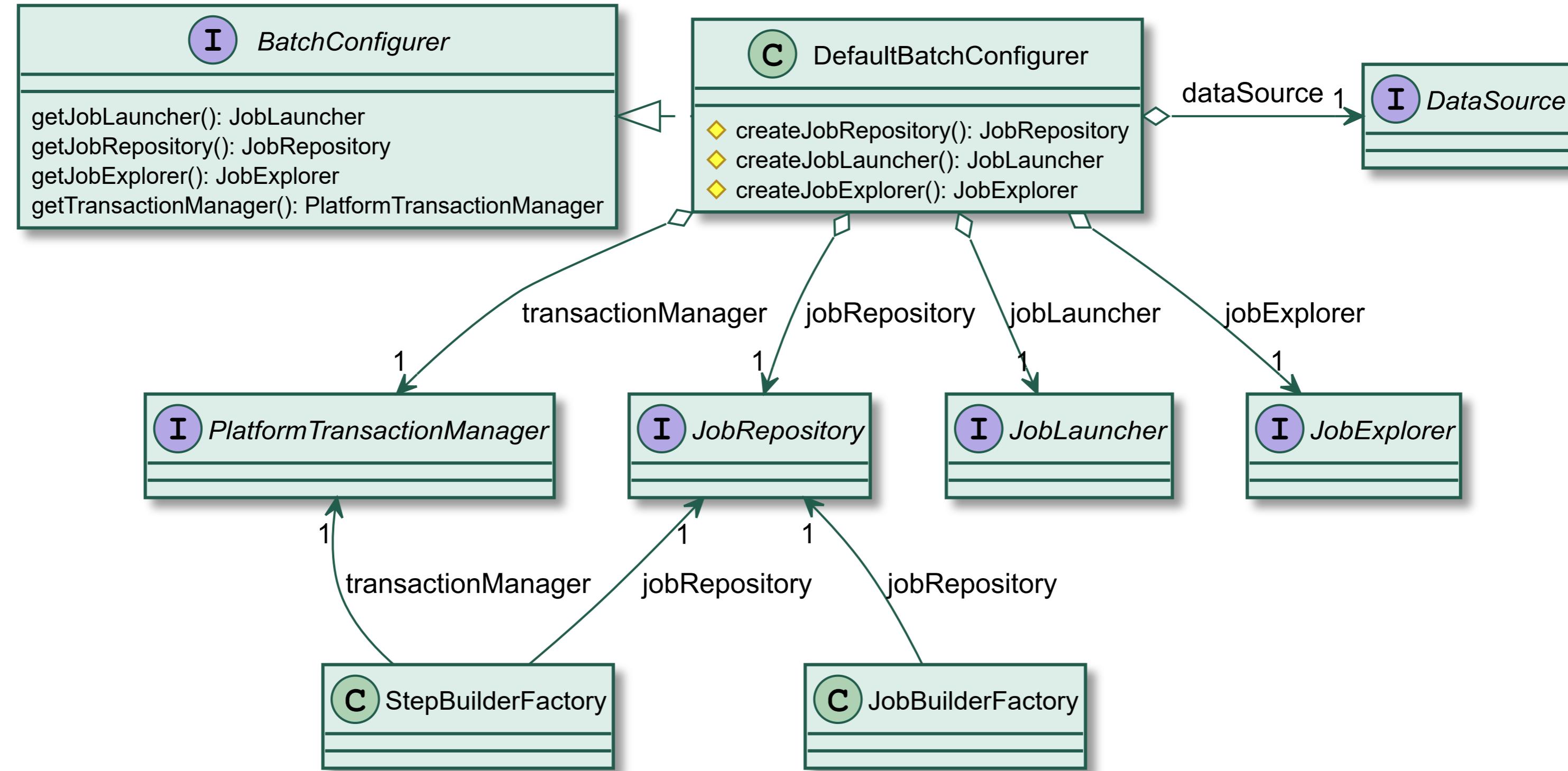
# Job-Konfiguration

## Konfiguration eines Batch-Jobs (Java-Config)

```
@Configuration  
@EnableBatchProcessing  
public class RetryJobConfig extends DefaultBatchConfigurer { (1)  
    @Autowired  
    private StepBuilderFactory stepBuilderFactory;  
    @Autowired  
    private JobBuilderFactory jobBuilderFactory;  
}
```

1. DefaultBatchConfigurer bietet *create*-Methoden für JobRepository, JobLauncher, TransactionManager etc.

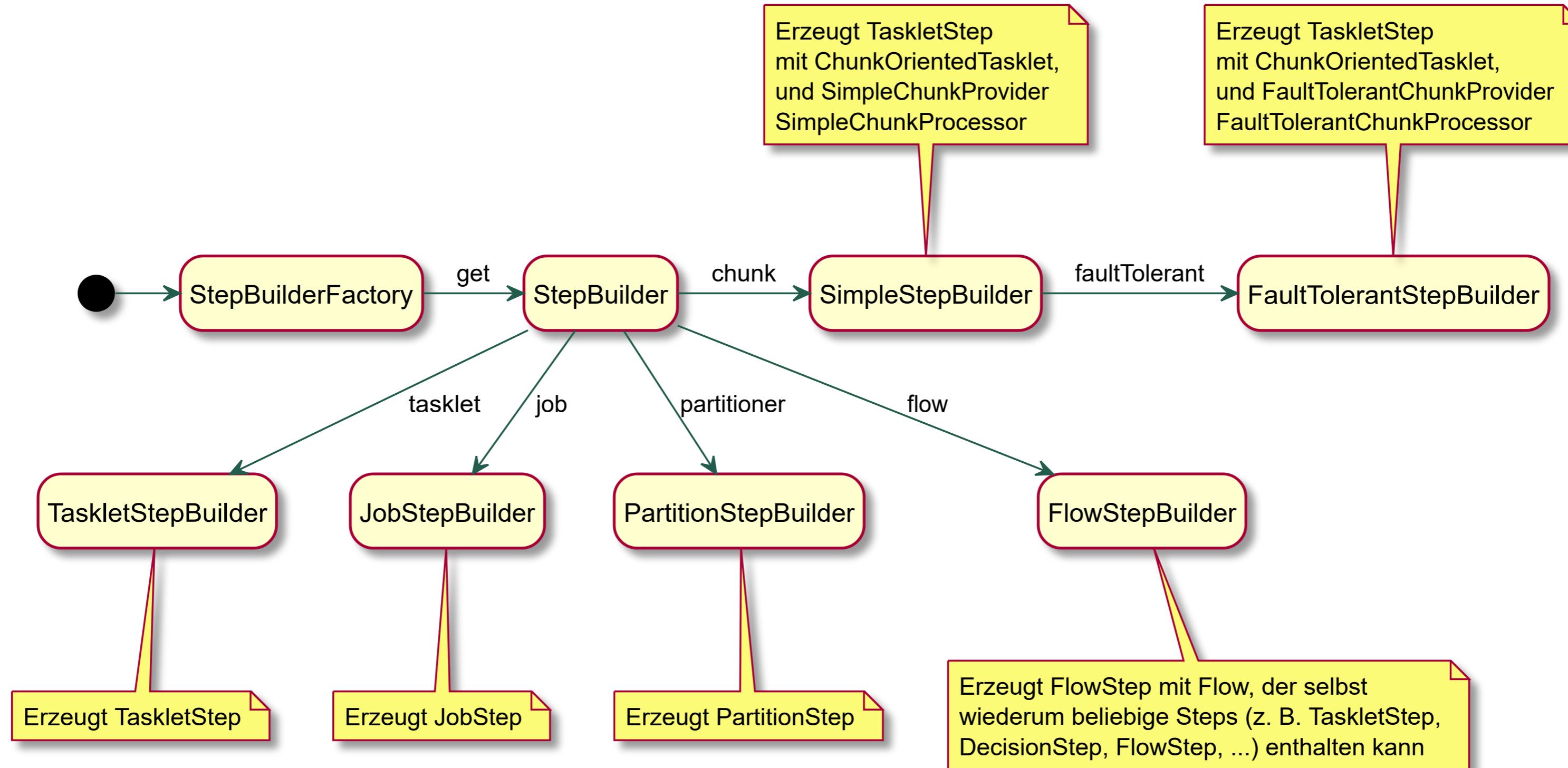
# Komponenten zur Job-Konfiguration



# Konfiguration eines Steps

```
TaskletStep step = stepBuilderFactory
    .get("retryStep") // StepBuilder
    .<Integer, String>chunk(4) // SimpleStepBuilder
    .reader(reader()).processor(processor()).writer(writer())
    .faultTolerant().retryLimit(7).retry(RetryException.class) // FaultTolerantStepBuilder
    .build();
```

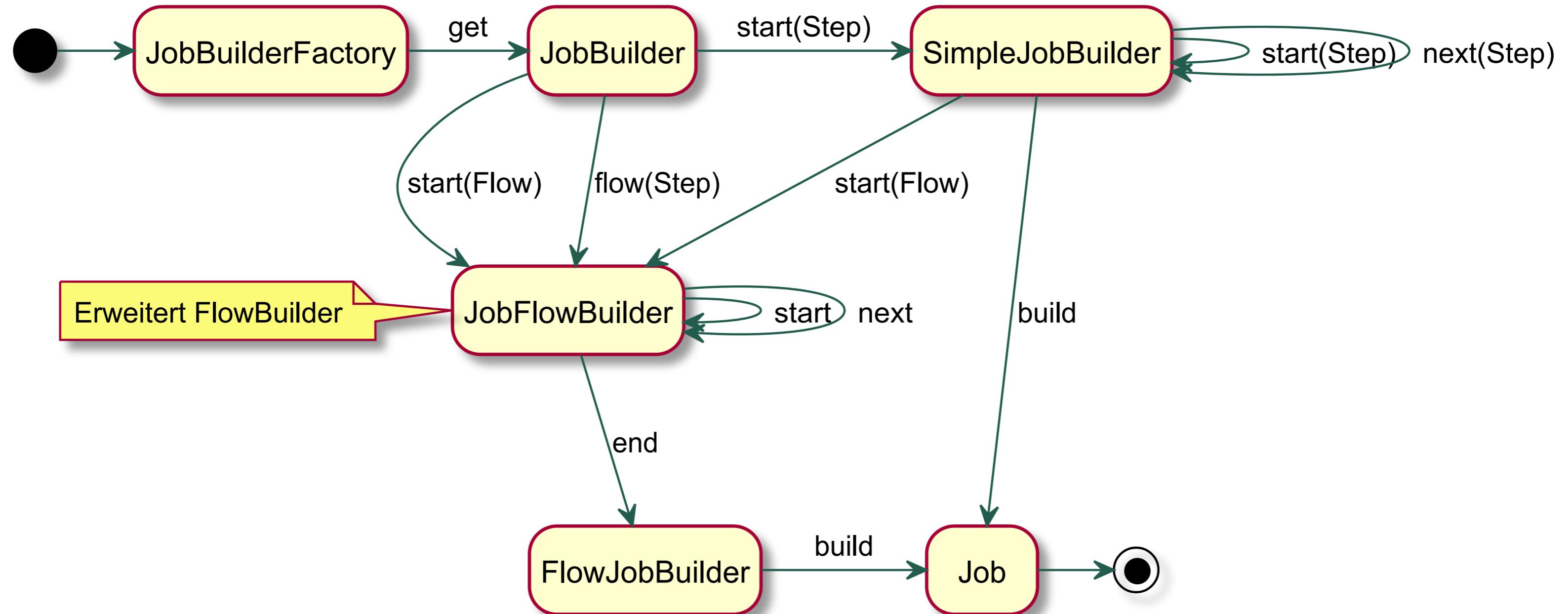
# StepBuilder



# Konfiguration eines Jobs

```
Job job = jobBuilderFactory  
    .get("retryJob")  
    .start(step)  
    .build();
```

# JobBuilder



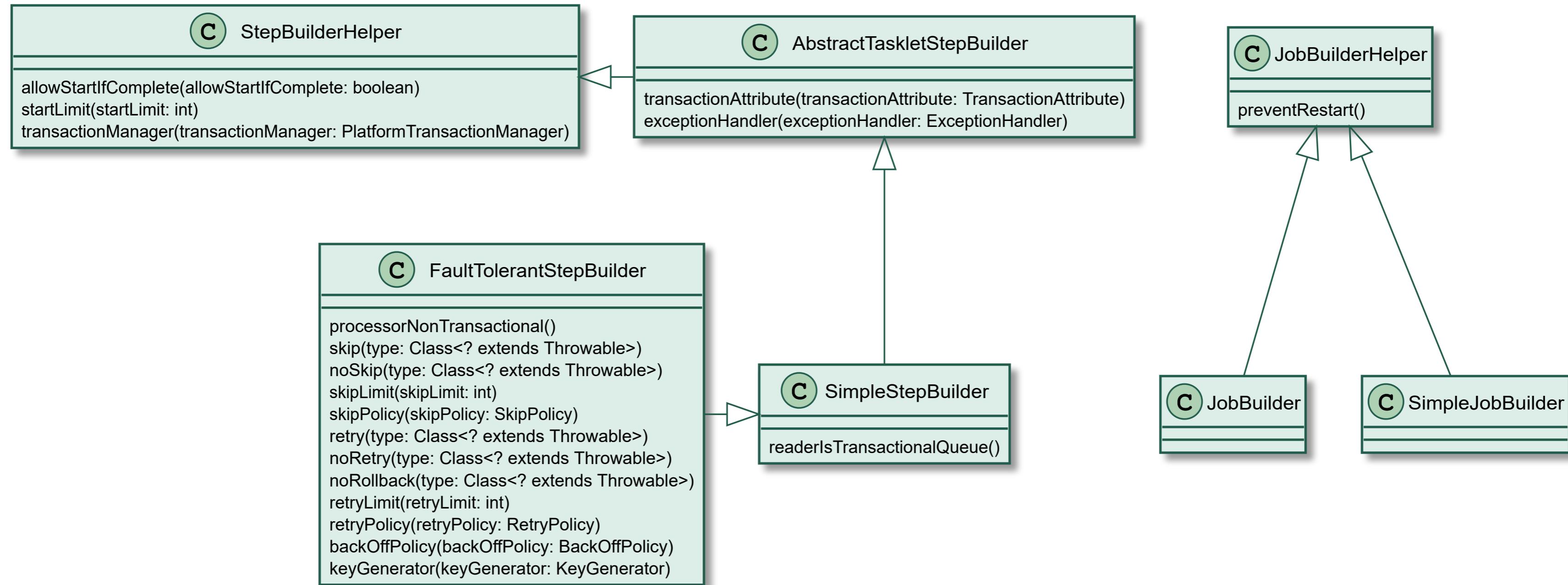
# Flows

## Job mit Flow

```
Step step3 = createTaskletStep("step 3", ExitStatus.COMPLETED);
Step step3x = createTaskletStep("step 3x");
Step step4 = createTaskletStep("step 4");
Flow flow1 = new FlowBuilder<Flow>("flow1")
    .from(createTaskletStep("step 2a 1"))
    .next(createTaskletStep("step 2a 2"))
    .build();
Flow flow2 = new FlowBuilder<Flow>("flow2")
    .from(createTaskletStep("step 2b"))
    .build();
Job job = jobBuilderFactory
    .get("flowJob")
    .start(createTaskletStep("step 1"))
    .split(executor()).add(flow1, flow2)
    .next(step3)
    .on(ExitStatus.COMPLETED.getExitCode()).to(step4)
    .from(step3).on(ExitStatus.FAILED.getExitCode())
    .to(step3x).next(step4)
    .end()
    .build();
```

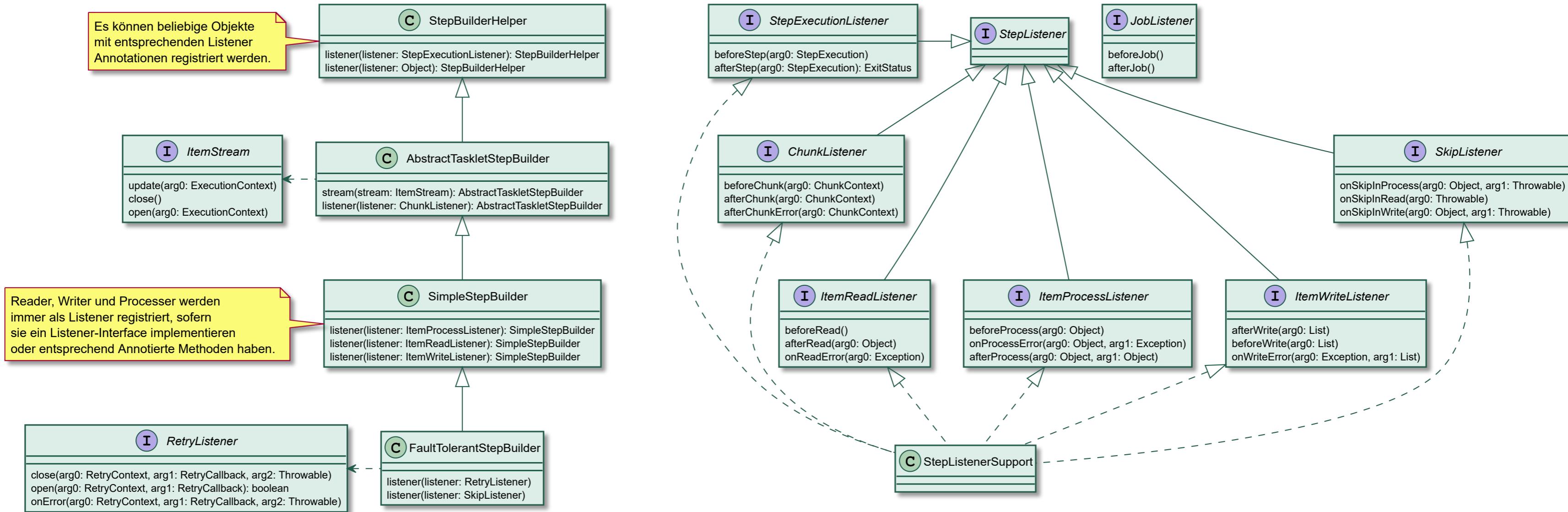
# Transaktionen und Wiederaufnahme

## Konfiguration des Transaktions- und Wiederauflaufverhaltens



# Listener

## Listener Interfaces und deren Registrierung



# Scope

## @StepScope und @JobScope

- Erzeugt Bean für Lebenszyklus eines Steps bzw. eines Jobs
- Ermöglicht *late binding* über `@Value ("#{jobParameters[input.file.name]}")`,  
`@Value ("#{jobExecutionContext[key]}")` oder  
`@Value ("#{stepExecutionContext[minValue]}")`
- `@StepScope` bei Reader, Processor oder Writer in Kombination mit Partitionierung sinnvoll

# Parallelisierung

## Vorüberlegungen zur Parallelisierung

# Parallelisierung

## Vorüberlegungen zur Parallelisierung

- Muss das sein?
  - → Lieber erst mal die Chunkgröße justieren

# Parallelisierung

## Vorüberlegungen zur Parallelisierung

- Muss das sein?
  - → Lieber erst mal die Chunkgröße justieren
- Ihr seid nicht allein!

# Parallelisierung

## Vorüberlegungen zur Parallelisierung

- Muss das sein?
  - → Lieber erst mal die Chunkgröße justieren
- Ihr seid nicht allein!
- Sinnvoll bei langsamer I/O
  - Archivsystem
  - Webservice

# Möglichkeiten der Parallelisierung

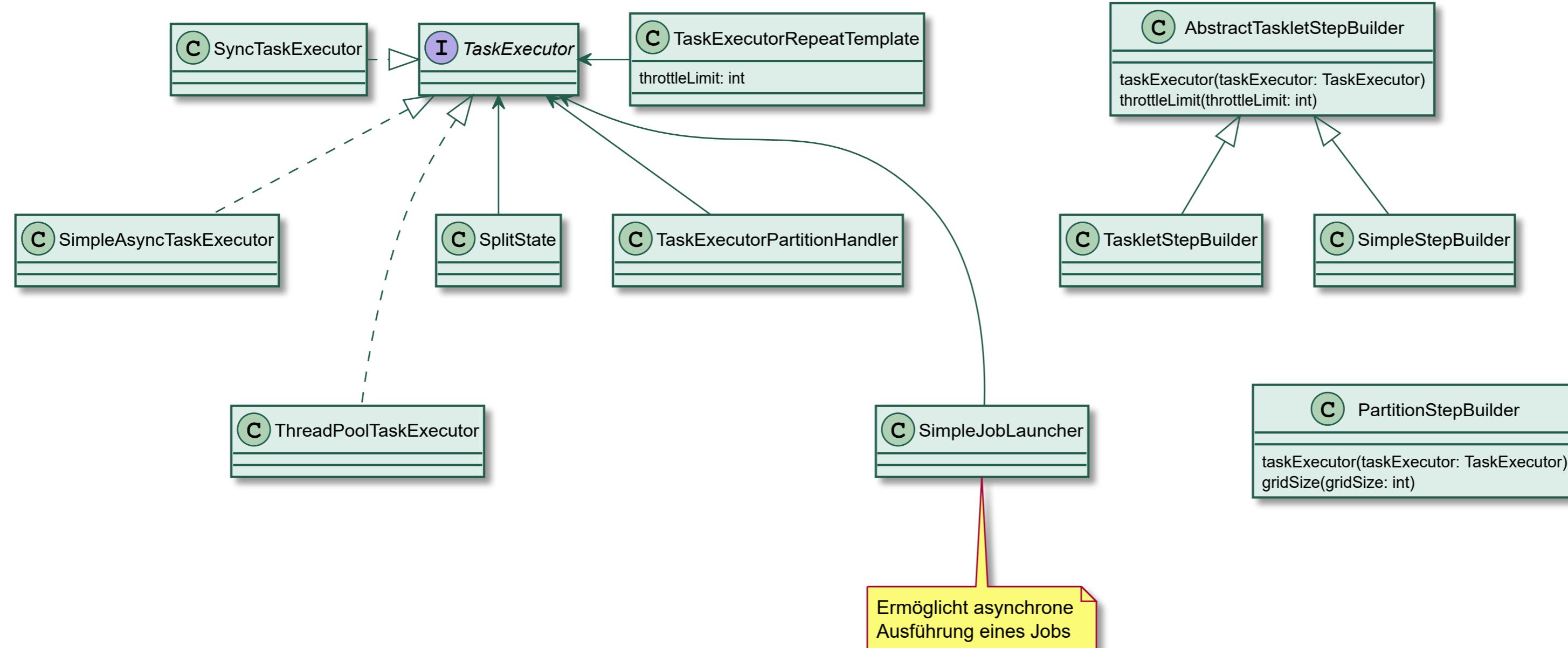
- Einen Step mit mehreren Threads ausführen
- Mehrere Steps gleichzeitig ausführen (Flows)
- Partitionierung
- Remote Chunking

# Step mit mehreren Threads ausführen

```
TaskletStep step = stepBuilderFactory
    .get("parallelStep")
    .<Integer, String>chunk(5)
    .reader(reader()).processor(processor()).writer(writer())
    .taskExecutor(taskExecutor()) // Ausführung mit mehreren Threads
    .throttleLimit(4) // begrenzt Anzahl der Threads (default: 4)
    .build();
```

- Reader, Processor und Writer müssen **thread-save** sein
- Chunk-size bestimmt Synchronisationsaufwand

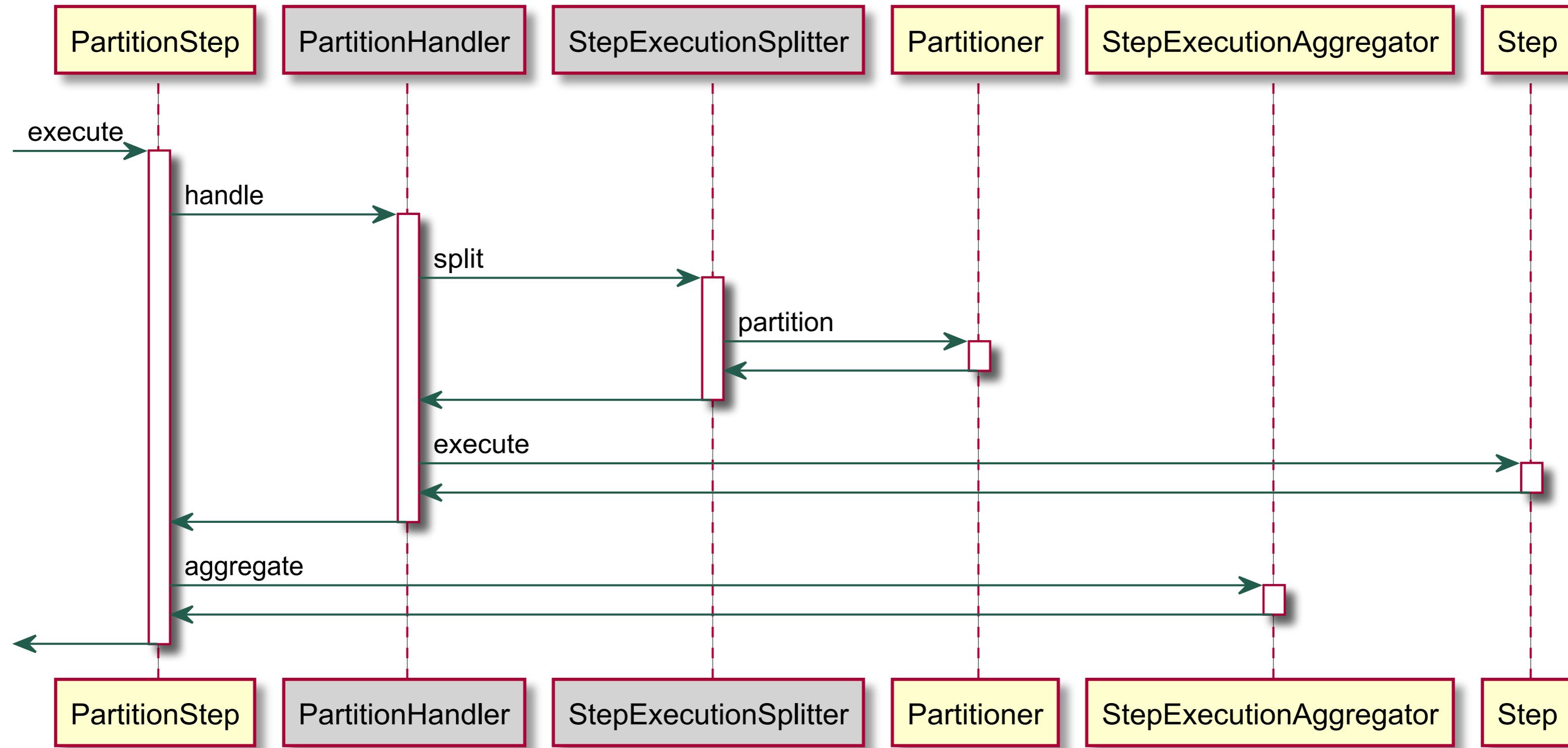
# TaskExecutor in Spring-Batch



# Partitionierung

- Aufteilung der Datenmenge eines Steps auf mehrere Steps
- Sinnvolle Kriterien zur Aufteilung notwendig
- Ermöglicht auch Remote-Verarbeitung
- Auch ohne Partitionierung in Kombination mit @StepScope sinnvoll

# PartitionStep

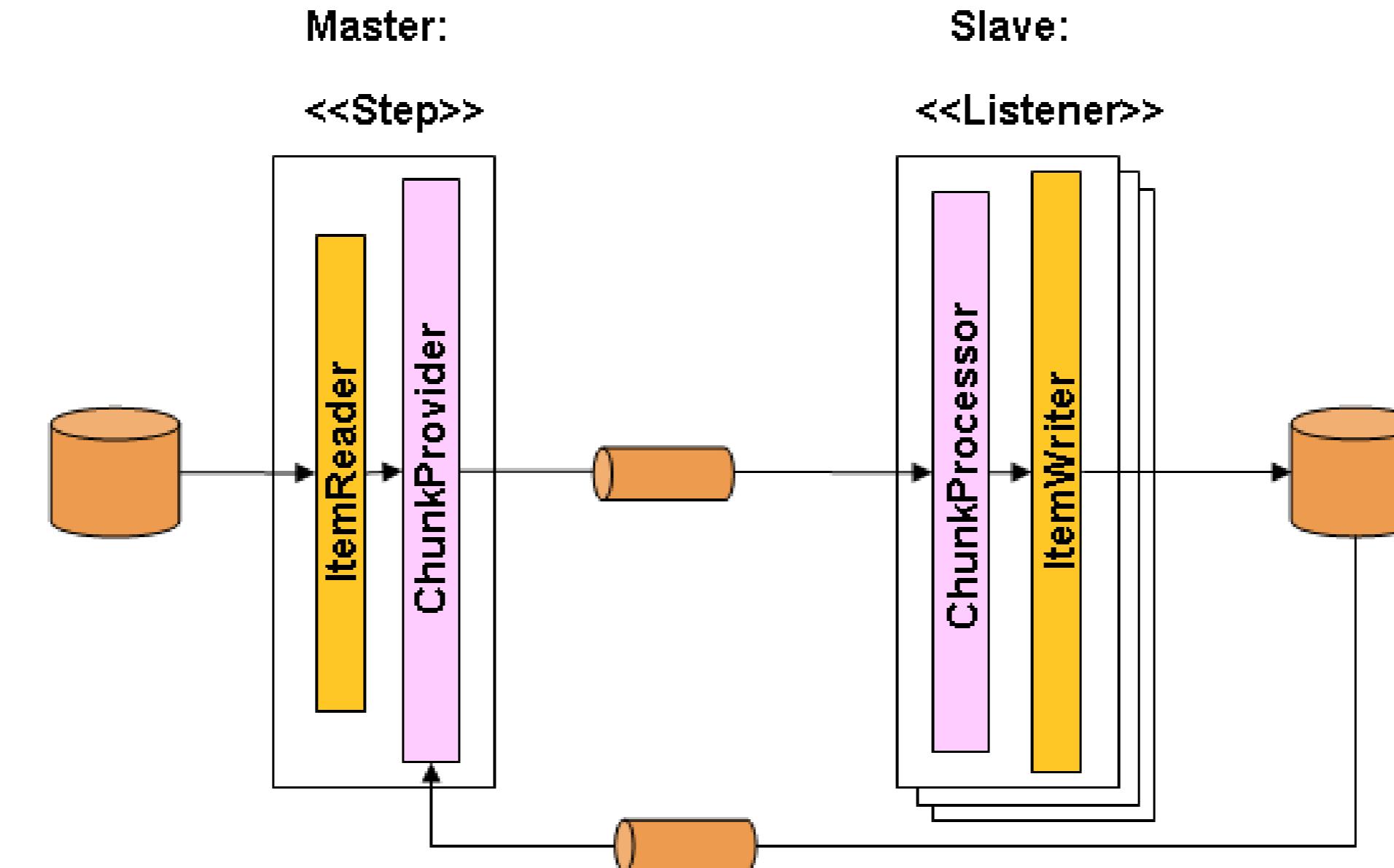


## Beispiel

→ PartitionConfig.java

# Remote Chunking

⇒ Für rechenintensive Steps



# Operations

## Operations Aufgaben

# Operations

## Operations Aufgaben

- Job Starten

# Operations

## Operations Aufgaben

- Job Starten
- Prüfen, ob Job erfolgreich ausgeführt wurde

# Operations

## Operations Aufgaben

- Job Starten
- Prüfen, ob Job erfolgreich ausgeführt wurde
- Fehlerursache nachsehen

# Operations

## Operations Aufgaben

- Job Starten
- Prüfen, ob Job erfolgreich ausgeführt wurde
- Fehlerursache nachsehen
- Statistik erstellen

# Operations

## Operations Aufgaben

- Job Starten
- Prüfen, ob Job erfolgreich ausgeführt wurde
- Fehlerursache nachsehen
- Statistik erstellen
- Job anhalten bzw. forsetzen

# Operations

## Operations Aufgaben

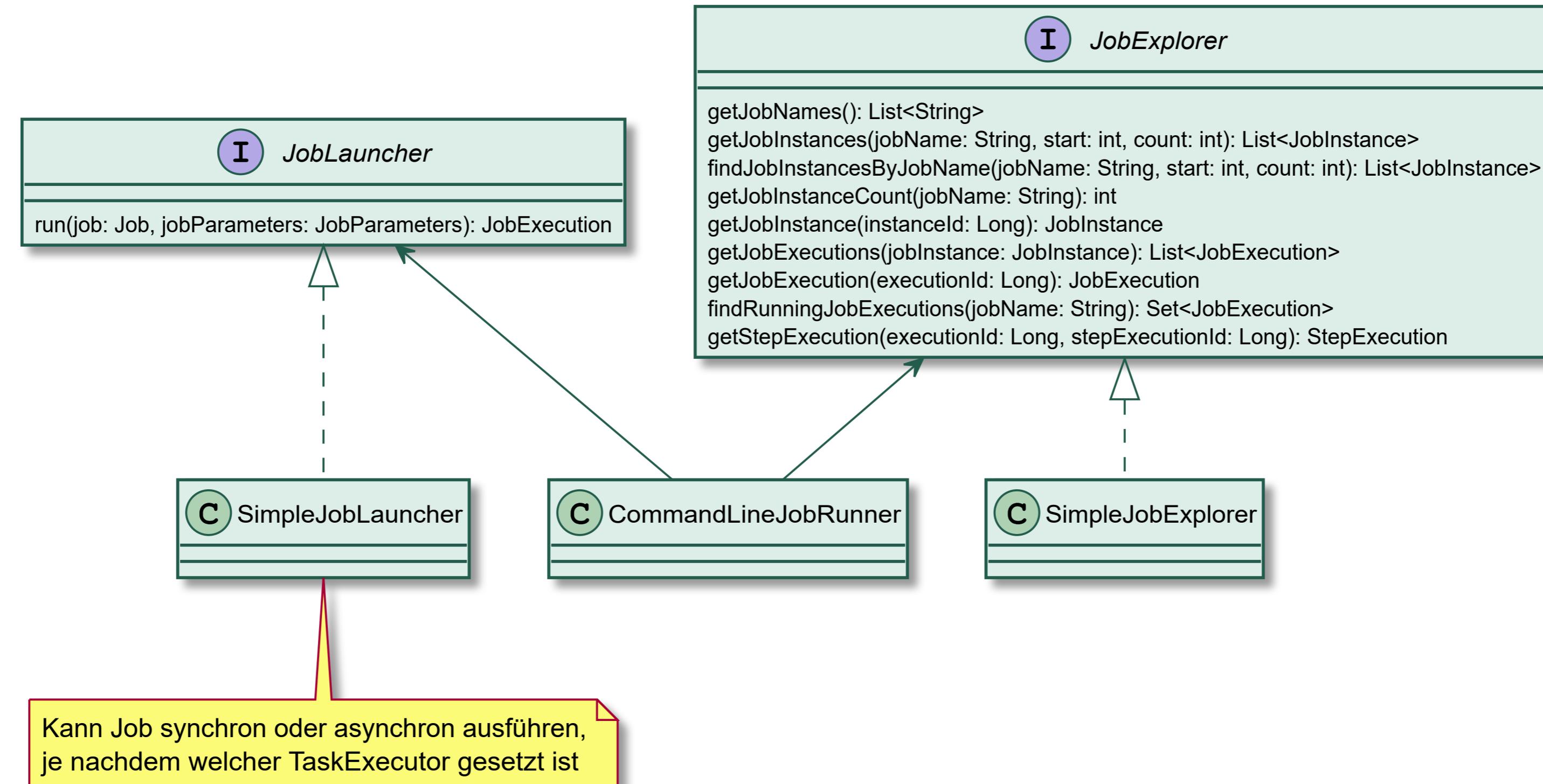
- Job Starten
- Prüfen, ob Job erfolgreich ausgeführt wurde
- Fehlerursache nachsehen
- Statistik erstellen
- Job anhalten bzw. forsetzen
- Job abbrechen

# Operations

## Operations Aufgaben

- Job Starten
- Prüfen, ob Job erfolgreich ausgeführt wurde
- Fehlerursache nachsehen
- Statistik erstellen
- Job anhalten bzw. forsetzen
- Job abbrechen
- Wiederanlauf versuchen

# Operations Interfaces



# Eindeutigen *run.id* Parameter konfigurieren

## *LauncherConfig.java*

```
@Bean  
RunIdIncrementer incrementer() {  
    return new RunIdIncrementer();  
}  
  
@Bean  
Job helloJob() {  
    Job job = jobBuilderFactory  
        .get("helloJob")  
        .start(helloStep())  
        .incrementer(incrementer())  
        .preventRestart()  
        .build();  
    return job;  
}
```

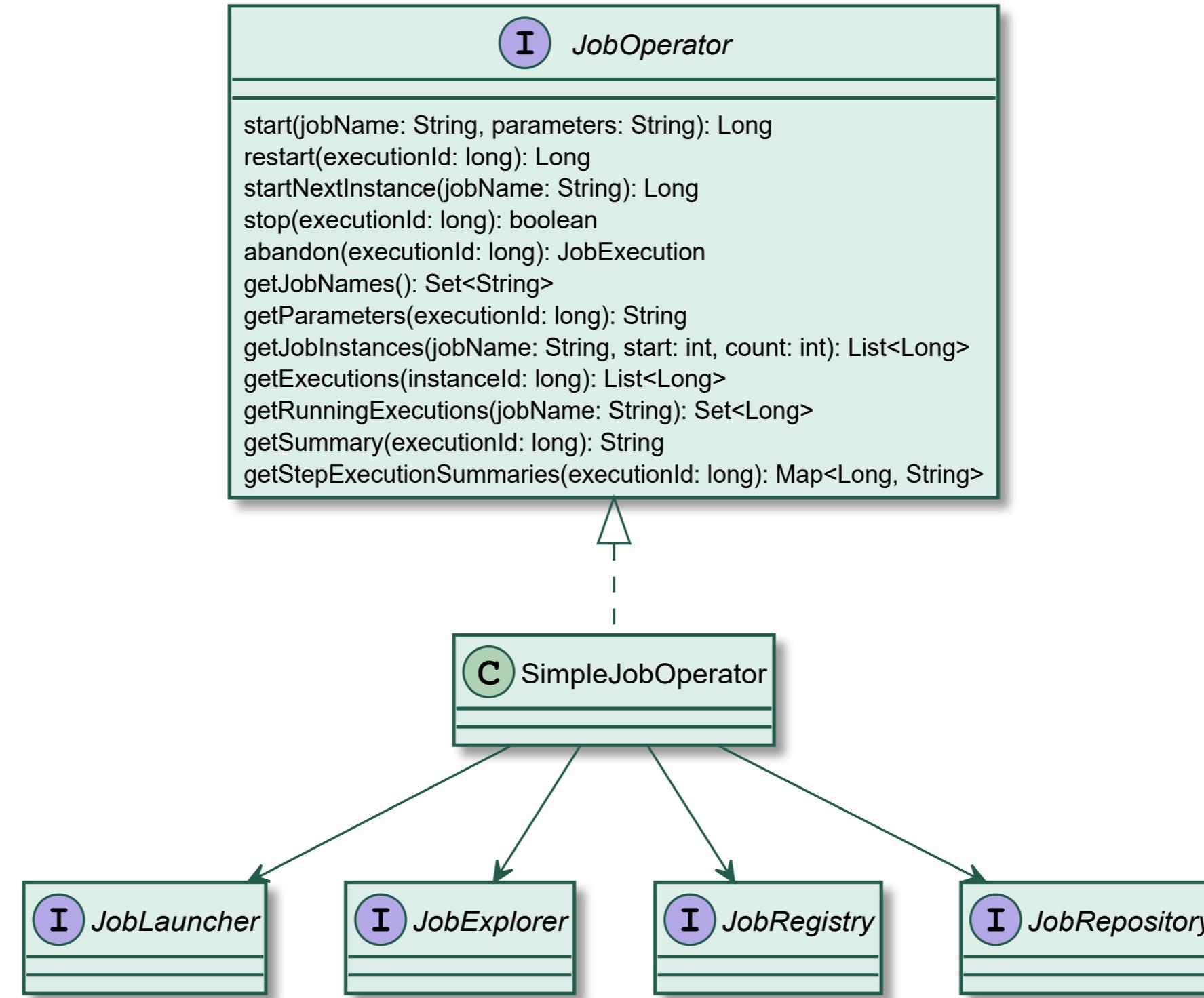
# Mit eindeutiger *run.id* starten

## *LauncherConfig.java*

```
JobParameters jobParameters = new JobParametersBuilder(jobExplorer)
    .getNextJobParameters(helloJob)
    .toJobParameters();
JobExecution jobExecution = jobLauncher.run(helloJob, jobParameters);

System.out.println(jobExecution);
jobExecution.getStepExecutions().forEach(System.out::println);
```

# JobOperator



# JobOperator konfigurieren

## *OperatorConfig.java*

```
@Bean  
SimpleJobOperator jobOperator() {  
    SimpleJobOperator operator = new SimpleJobOperator();  
    operator.setJobLauncher(jobLauncher);  
    operator.setJobExplorer(jobExplorer);  
    operator.setJobRegistry(jobRegistry);  
    operator.setJobRepository(jobRepository);  
    return operator;  
}  
  
@Bean  
JobRegistryBeanPostProcessor jobRegistryBeanPostProcessor() {  
    JobRegistryBeanPostProcessor postProcessor = new JobRegistryBeanPostProcessor();  
    postProcessor.setJobRegistry(jobRegistry);  
    return postProcessor;  
}
```

# JobOperator verwenden

## *OperatorConfig.java*

```
@Autowired  
private JobOperator jobOperator;  
  
@Test  
void testJobOperator() throws Exception {  
    Long executionId = jobOperator.startNextInstance("helloJob");  
    System.out.println(jobOperator.getSummary(executionId));  
    System.out.println(jobOperator.getStepExecutionSummaries(executionId));  
}
```