# Implementierung von Event-Storming-Modellen mit Axon

*Sven-Torben Janus, Conciso GmbH*

*Seit einigen Jahren erfreut sich Domain-driven Design (DDD) großer Beliebtheit. Damit einhergehend rücken leichtgewichtige Modellierungsmethoden in den Fokus. Eine dieser Methoden ist Event Storming. Es hilft Teams, ein gemeinschaftliches, interdisziplinäres Verständnis des Problem- und Lösungsraums zu entwickeln, das sich in Form vieler Post-its an einer Wand manifestiert. Aber wie überführt man eine Menge von Post-its in ausführbaren Code? Ein patternbasierter Ansatz kann hier helfen.*

Anderthalb Jahrzehnte nach dem Erscheinen von Eric Evans „Big Blue Book" *[1]* ist Domain-driven Design (DDD) präsenter als je zuvor. Dies ist nicht zuletzt auf den anhaltenden Microservice-Hype zurückzuführen. Die Zerlegung großer Systeme in viele kleinere Teile erfordert in der Regel eine strikte Trennung von Funktionalitäten unter fachlichen Gesichtspunkten. Insbesondere die strategischen Patterns des DDD werden daher vielfach für entsprechend notwendige Designentscheidungen angeführt. Die Anwendung dieser Patterns erfordert jedoch immer ein gutes Verständnis des fachlichen Problem- und Lösungsraums. Leichtgewichtige Modellierungsmethoden rücken dabei immer stärker in den Fokus.

Wer schon einmal gesehen hat, wie Kollegen diskutieren und währenddessen eine Wand mit orangenen Post-its tapezieren, hat mit hoher Wahrscheinlichkeit eine Event Storming *[2]* Session beobachtet. Mit Event Storming hat sich eine Workshop-Methode etabliert, die es den beteiligten Personen – von Fachexperten, strategischen Entscheidern über User Experience Designer, Product Owner bis hin zu Entwicklern – ermöglicht, genau dieses Verständnis interdisziplinär und gemeinschaftlich zu erarbeiten. Das gemeinsame Verständnis spiegelt sich dabei in Form eines Domänenmodells wider. Diese können je nach Ausrichtung und Ziel des Workshops von grob granularen „Big Picture"-Modellen bis hin zu detaillierten, implementierungsnahen Modellen variieren. Aber wie kann man behaupten, dass eine Menge von Post-its implementierungsnah sei? Wie kann man Post-its in ausführbaren Code überführen?

Aus anderen Bereichen der Modellierung werden hierzu häufig patternbasierte Ansätze genutzt. Das Überführen von formalen Modellen, wie zum Beispiel UML, in eine objektorientierte Programmiersprache oder ein relationales Datenbankmodell funktioniert im Wesentlichen patternbasiert. Entsprechend sind auch (Meta-)Pattern im Sinne der „Gang of Four" *[3]* recht eindeutig in Code überführ- und ausführbar.

Event Storming kennt im Kern ebenfalls Patterns. Anhand einiger ausgewählter wird nachfolgend auf Basis des Axon Frameworks *[4]* gezeigt, wie sie zur Implementierung von Event-Storming-Modellen

Abbildung 1: Event Storming – Big Picture

nutzbar sind. Ziel dabei soll sein, das gemeinsame Verständnis im Sinne einer Ubiquitous Language [5] möglichst eins zu eins in Code zu überführen.

## Event Storming

Beim Event Storming startet die Modellierung in der Regel mit dem Aufschreiben von Schlüsselereignissen der Fachdomäne auf orangefarbige Post-its. Die Ereignisse sind dabei in zeitlicher Reihenfolge sortiert. Ziel ist es vor allem, den Prozess und den gesamten Kontext des Problems und nicht jedes Detail zu modellieren.

Stehen die wesentlichen Ereignisse einmal fest, stellt sich die Frage, durch welche Aktionen die Ereignisse ausgelöst werden. Um diese Frage zu beantworten, müssen die Teilnehmer einen rückwärtsgewandten Blick einnehmen. Aktionen werden im Event Storming durch sogenannte „Kommandos" auf blauen Post-its modelliert. Oftmals sind sie durch einen Benutzer initiiert. Sie können allerdings auch durch externe Systeme oder Richtlinien ausgelöst werden. Damit ein Benutzer ein Kommando auslösen beziehungsweise eine Entscheidung treffen kann, benötigt er entweder Informationen aus der realen Welt oder über den Zustand (grüne Post-its) der Anwendung. Letzterer kann aus den in der Domäne aufgetretenen Ereignissen abgeleitet werden.

Ist der wesentliche Prozess einmal definiert und klar, können vor allem die Entwickler in ein implementierungsnäheres Design einsteigen. Hierzu werden Aggregate (breite gelbe Post-its) und externe Systeme (breite pinke Post-its) identifiziert, die Verantwortlichkei-

Abbildung 2: Beispielprozess: Kunde wählt Produkte aus

ten für die Verarbeitung von Kommandos übernehmen. Im Zusammenhang ergibt sich somit ein Bild wie in *Abbildung 1* dargestellt.

## Ein Modellbeispiel

Zur Veranschaulichung der Implementierung eines durch Event Storming entstandenen Modells soll ein klassisches Shop-Beispiel dienen. Zu Beginn des betrachteten Prozesses steht ein Kunde, der einen Shop betritt. Immer, wenn der Kunde den Shop betritt, soll ihm ein Warenkorb bereitgestellt werden, in den er Produkte legen kann *(siehe Abbildung 2)*.

Nachdem der Kunde einige Produkte ausgewählt hat, bekommt er seinen Warenkorb, den er anschließend bestellen kann, inklusive Gesamtpreis und möglicher Rabatte angezeigt. Daraufhin wird eine Bestellung erzeugt und das System quittiert dies mit dem Ereignis, dass der Warenkorb bestellt wurde *(Abbildung 3)*.

Im Anschluss an die Bestellung erfolgen einige automatische Schritte. Per Richtlinie wird festgelegt, dass bei einem Versand außerhalb der EU die Waren zum Zollverfahren angemeldet werden müssen. Gleichzeitig werden immer die Versandart bestimmt sowie eine Rechnung gestellt *(siehe Abbildung 4)*.

Nach der Rechnungsstellung kann der Kunde seine Rechnung bezahlen. Das Rechnungssystem quittiert dies mit einem entsprechenden Ereignis. Durch die Bezahlung der Rechnung wird die Ware automatisch zum Versand freigegeben *(Abbildung 5)*.

*Abbildung 3: Beispielprozess: Kunde bestellt Waren*

*Abbildung 4: Beispielprozess: Rechnungsstellung, Versandart und Zollanmeldung*

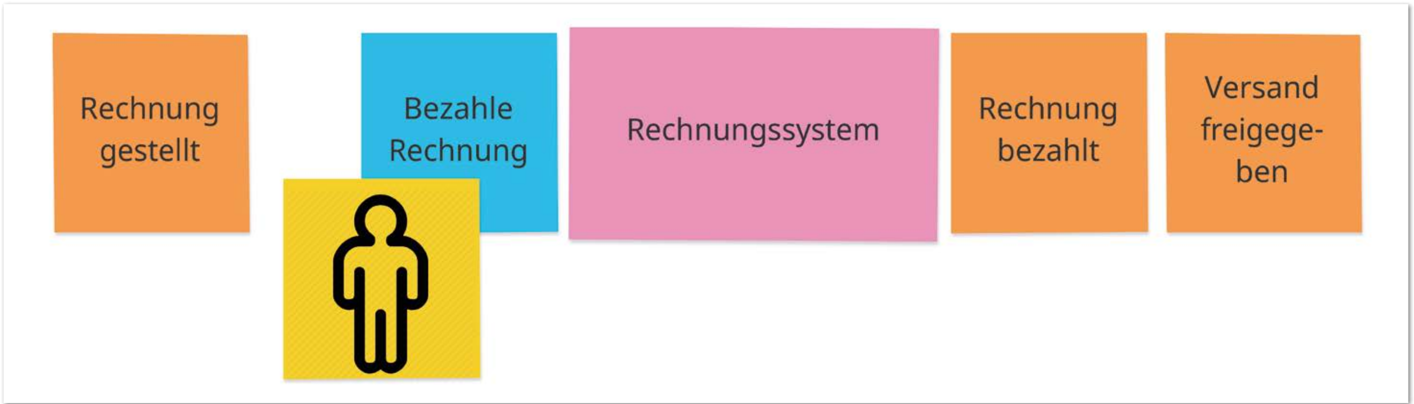


*Abbildung 5: Beispielprozess: Kunde bezahlt Rechnung*

*Abbildung 6: Beispielprozess: Warenversand*

Wenn der Versand freigegeben ist und die Versandart bestimmt wurde, wird die Ware versendet *(siehe Abbildung 6)*.

## Das Axon Framework

Für die Implementierung kommt das Axon Framework zum Einsatz. Axon ist ein auf Java basierendes Command Query Responsibility Segregation (CQRS) *[6]* Framework. CQRS trennt lesende Anfragen, die den Zustand der Anwendung nicht verändern, von schreibenden, zustandsverändernden Aktionen. Im Kern seiner Architektur folgt Axon dabei dem Zyklus in *Abbildung 1*. Es stellt einige Building Blocks bereit, die eine Implementierung des Zyklus recht einfach gestalten. Über sogenannte „Command Handler" können Kommandos eines Benutzers ausgeführt werden. Die Command Handler agieren auf dem Domänenmodell, das im Kern aus Aggregaten besteht. Aggregate können mittels Repositories gespeichert werden. Dabei unterstützt Axon neben dem Speichern des aktuellen Zustands eines Aggregats auch die Speicherung der Ereignisse, die zu diesem Zustand geführt haben. Axon kombiniert an dieser Stelle CQRS mit Event Sourcing *[7]*. Ereignisse werden zugleich auf einen Event Bus publiziert. Von dort können sie weitere Aktionen auf den Aggregaten auslösen. Gleichzeitig können sie jedoch auch mit Event Handlern verarbeitet werden. Diese Event Handler erlauben es, weitere Modelle, die für lesende Anfragen optimiert sind, zu persistieren. Diese wiederum können durch Query Handler abgefragt und somit Informationen auf der UI angezeigt werden.

Mit diesen Building Blocks bietet Axon alle Voraussetzungen, um gängige Patterns aus Event-Storming-Modellen zu implementieren. Die am häufigsten anzutreffenden sind neben vielen weiteren vermutlich die nachfolgenden drei *(siehe Abbildung 7)*.

## Pattern: Automation

Automation ist eines der einfachsten Patterns im Event Storming. Das Pattern besagt, dass bei Eintritt eines Ereignisses automatisch ein weiteres Ereignis oder Kommando ausgelöst wird. Dieses Pattern lässt sich im Beispielprozess gut erkennen. Der Versand

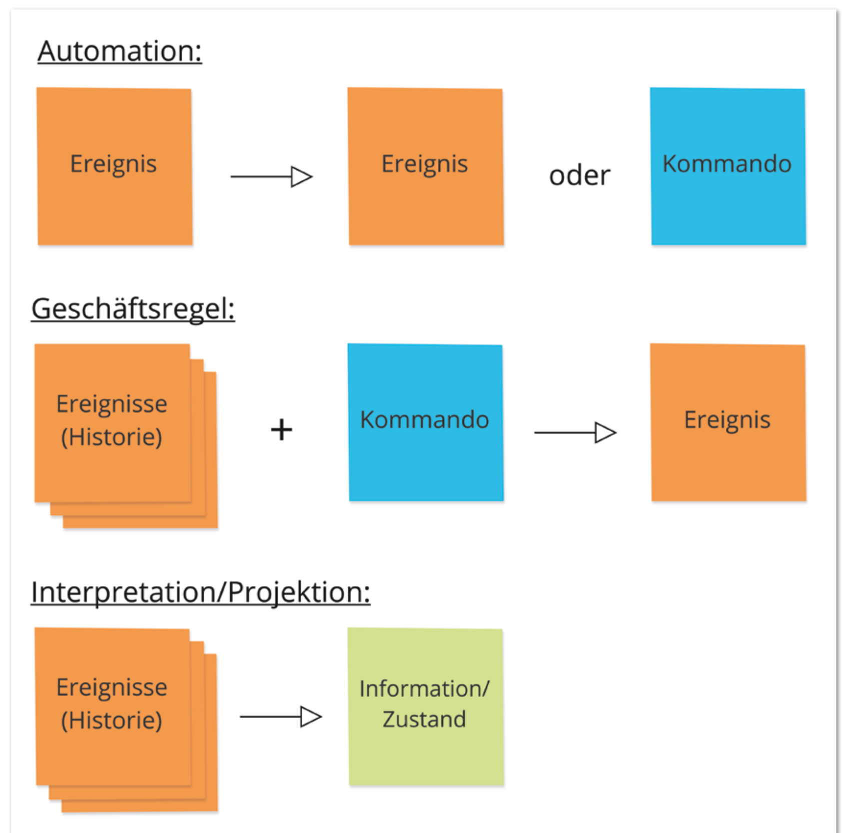


*Abbildung 7: Auswahl häufig auftretender Patterns beim Event Storming*

von Waren wird automatisch freigegeben (Ereignis), wenn die Rechnung bezahlt wurde (Ereignis).

In Axon lässt sich dieses Pattern mit einem einfachen Event Handler in Kombination mit Axons Event Bus implementieren. Mit Event Handlern kann auf Ereignisse reagiert werden. Der Event Bus erlaubt das Versenden weiterer Ereignisse. Für einen solchen Event Handler muss lediglich eine Methode mit @EventHandler annotiert werden. Sie nimmt als Parameter ein Ereignis vom Typ „RechnungBezahlt" entgegen. Der Typ lässt sich einfach als POJO implementieren und stellt ereignisbezogene Informationen bereit. In diesem Fall enthält das Ereignis die „WarenkorbId", um mitzuteilen, für welchen Warenkorb die Rechnung bezahlt wurde. Um zu signalisieren, dass automatisch der Versand freigegeben wurde, kann einfach ein entsprechendes Ereignis mithilfe des Event Bus publiziert werden *(siehe Listing 1)*.

Anstelle eines Ereignisses kann bei der Automation auf ein Ereignis auch ein Kommando folgen. Zum Beispiel soll einem Kunden, wann immer er den Shop betreten hat (Ereignis), automatisch ein Warenkorb bereitgestellt werden (Kommando). Das Pattern lässt sich mit Axon ebenfalls recht einfach implementieren *(siehe Listing 2)*.

Auch an dieser Stelle muss einfach nur ein Event Handler wie zuvor implementiert werden. Die dazugehörige Methode nimmt als Parameter ein Ereignis vom Typ „KundeHatShopBetreten" entgegen. Anstelle in der Methode ein weiteres Ereignis zu publizieren, kann Axon über das sogenannte „Command Gateway" aber auch ein Kommando auslösen. Das Gateway lässt sich wie auch der Event Bus einfach injizieren. Mit der Methode „sendAndWait" können ein Kommando ausgelöst und auf dessen Verarbeitung gewartet werden. Es stehen allerdings auch Methoden zur asynchronen Verarbeitung bereit. Auch Kommandos sind einfache POJOs und enthalten die Informationen, die benötigt werden, um ein Kommando auszuführen. In diesem Falle die „KundenId", um zu signalisieren, für welchen Kunden ein Warenkorb bereitgestellt werden soll.

```
class Shop {

  @Inject
  EventBus eventBus;

  @EventHandler
  void on(RechnungBezahlt event) {
      eventBus.publish(GenericEventMessage.asEventMessage(new
VersandFreigegeben(event.getWarenkorbId())));
  }

}
```

Listing 1

```
class Shop {

  @Inject
  CommandGateway commandGateway;

  @EventHandler
  void on(KundeHatShopBetreten event) {
      UUID warenkorbId = commandGateway.sendAndWait(new
StelleWarenkorbBereit (event.getKundenId()));
      // …
  }

}
```

Listing 2

## Pattern: Geschäftsregel

Kommandos sind in der Regel auch Auslöser einer Geschäftsregel. Eine Geschäftsregel besteht immer aus einer Reihe von Ereignissen, der sogenannten „Historie", und einem auslösenden Kommando. Als Ergebnis der Geschäftsregel wird ein weiteres Ereignis signalisiert.

Wie Kommandos durch das Command Gateway ausgelöst werden können, wurde bereits gezeigt. Zur Verarbeitung von Kommandos

```
public class Warenkorb {

  @AggregateIdentifier
  private UUID id;
  private UUID kundenId;

}

public class WarenkorbHandler {

  @Inject
  Repository<Warenkorb> warenkoerbe;

  @CommandHandler
  public UUID handle(StelleWarenkorbBereit command) throws Exception {
      UUID warenkorbId = UUID.randomUUID();
      Aggregate<Warenkorb> warenkorb = warenkoerbe.newInstance(() -> new Warenkorb(warenkorbId, command.getKunde-
nId()));
      return (UUID) warenkorb.identifier();
  }

  @CommandHandler
  public void handle(LegeProduktInWarenkorb command) {
      Aggregate<Warenkorb> warenkorb = warenkoerbe.load(command.getWarenkorbId().toString());
      // ...
  }

}
```

Listing 3

```
public class Warenkorb {

  @AggregateIdentifier
  private UUID id;
  private UUID kundenId;

  @CommandHandler
  public Warenkorb(StelleWarenkorbBereit command) {
     this(command.getWarenkorbId(), command.getKundenId());
  }

  @CommandHandler
  public void handle(LegeProduktInWarenkorb command) {
     if (command.getAnzahl() > 0) {
        // …
     } else {
        throw new IllegalArgumentException("Anzahl
muss größer 0 sein.");
     }
  }
}
```

Listing 4

```
public interface LegeProduktInWarenkorb {

  @TargetAggregateIdentifier
  UUID getWarenkorbId();

  UUID getProduktId();

  int getAnzahl();

}
```

Listing 5

ebenfalls nur eine Methode annotiert werden – diesmal mittels @CommandHandler. Die Methode nimmt als Parameter ein Kommando vom Typ „StelleWarenkorbBereit" entgegen *(siehe Listing 3)*.
Die Methode implementiert die Logik, die zur Ausführung des Kommandos notwendig ist. Im Beispiel werden eine „WarenkorbId" erzeugt und ein Warenkorb in Form eines Aggregats mit dieser ID angelegt. Durch die Annotation `@AggregateIdentifier` wird dem Framework signalisiert, welcher Wert als Identifier des Aggregats (hier des Warenkorbs) dient. Die eigentliche Anlage des Aggregats erfolgt über Axons Repository Interface. Das Repository stellt Methoden zum Speichern und Laden bereit.

Bei dieser Implementierungsvariante fällt auf, dass recht viel Boiler-Plate-Code für den Zugriff auf das Repository geschrieben werden muss. Die zweite Möglichkeit der Implementierung ist deutlich weniger geschwätzig. Wann immer ein Aggregat erzeugt werden soll, kann nämlich auch ein Konstruktor als Command Handler genutzt werden *(siehe Listing 4)*.

Allerdings fällt auf, dass die Generierung der „WarenkorbId" nun bereits in das Kommando kodiert werden muss, da diese Form von Command Handler keinen Rückgabewert erlaubt. Soll das Aggregat weitere Kommandos ausführen, ist es für den Zugriff auf das Aggregat außerdem notwendig, innerhalb des Kommandos festzulegen, welche Eigenschaft den Identifier des Aggregats definiert. Dies erfolgt, wie in *Listing 5* gezeigt, einfach mit der Annotation `@TargetAggregateIdentifier`.

```
public class Warenkorb {

  @AggregateIdentifier
  private UUID id;
  private UUID kundenId;
  private Map<Produkt, Integer> produkte;

  @CommandHandler
  public Warenkorb(StelleWarenkorbBereit command) {
     this(command.getWarenkorbId(), command.getKundenId());
  }

  public Warenkorb(UUID warenkorbId, UUID kundenId) {
     AggregateLifecycle.apply(new WarenkorbBereitgestellt(warenkorbId, kundenId));
  }

  @CommandHandler
  public void handle(LegeProduktInWarenkorb command) {
     if (command.getAnzahl() > 0) {
        AggregateLifecycle.apply(new ProduktInWarenkorbGelegt(command.getProduktId(), command.getAnzahl()));
     } else {
        throw new IllegalArgumentException("Anzahl muss größer 0 sein.");
     }
  }

  @EventSourcingHandler
  public void on(WarenkorbBereitgestellt event) {
     this.id = event.getWarenkorbId();
     this.kundenId = event.getKundenId();
     this.produkte = new HashMap<>();
  }

  @EventSourcingHandler
  public void on(ProduktInWarenkorbGelegt event) {
     Produkt neuesProdukt = new Produkt(event.getProduktId());
     produkte.compute(neuesProdukt, (produkt, anzahlImWarenkorb) ->
        anzahlImWarenkorb == null ? event.getAnzahl() : anzahlImWarenkorb + event.getAnzahl());
  }

}
```

Listing 6

Nach Verarbeitung des Kommandos muss als Ergebnis der Geschäftsregel wieder ein Ereignis signalisiert werden. Auch hier kennt Axon zwei grundlegende Mechanismen. Zum einen lässt sich auch hier einfach der Event Bus, wie zuvor gezeigt, nutzen. Zum anderen stellt Axon aber auch einen Event-Sourcing-Mechanismus bereit. Für das zuvor gezeigte Beispiel sieht das wie in *Listing 6* beschrieben aus.

Anstatt in einem Command Handler den Zustand eines Aggregats direkt zu verändern, wird im Command Handler nur geprüft, ob das Kommando (z.B. basierend auf dem aktuellen Zustand des Aggregats) angewendet werden kann oder darf. Wenn dies so ist, wird einfach ein Ereignis signalisiert. Diesmal wird dazu jedoch nicht der Event Bus direkt verwendet, sondern die AggregateLifecycle.apply-Methode. Diese bewirkt zum einen, dass das Ereignis mittels Event Bus signalisiert wird, speichert aber zugleich, dem Event-Sourcing-Ansatz entsprechend, das Ereignis in der Historie aller Ereignisse des Aggregats. Dies erlaubt Axon, den Zustand des Aggregats jederzeit aus der Historie der Ereignisse wiederherzustellen. Dazu müssen, ähnlich zu Event Handlern, sogenannte „Event Sourcing Handler" implementiert werden. Auch dies geschieht analog zu Event Handlern einfach per Annotation `@EventSourcingHandler`. Beim Wiederherstellen des Zustands des Aggregats wird Axon die Event Sourcing Handler in der Reihenfolge der Ereignisse wieder aufrufen.

Neben diesen sehr einfachen Geschäftsregeln kommt es in der Praxis natürlich auch vor, dass langlebigere Geschäftsprozesse automatisiert werden müssen. Axon stellt hierzu sogenannte „Sagas" bereit. Eine Saga erlaubt es, eine Reihe lokaler Transaktionen oder in diesem Fall Kommandos auszuführen (und gegebenenfalls zu kompensieren). Aus dem Beispielprozess bietet sich hier eine Folge von Ereignissen und Kommandos an, die zur Abwicklung der Bestellung dienen. Wurde ein Warenkorb bestellt, sollen die Versandart bestimmt und die Rechnung gestellt werden. Wenn die Rechnung bezahlt wurde, wird die Ware automatisch zum Versand freigegeben. Wurden die Ware zum Versand freigegeben und die Versandart bestimmt, soll die Ware versendet werden. Dies lässt sich mit einer Saga wie in *Listing 7* beschrieben abbilden.

Sagas nutzen spezielle Event Handler, die mithilfe der `@SagaEventHandler`-Annotation definiert werden. Mit der Annotation `@StartSaga` kann definiert werden, welcher der Event Handler eine neue Instanz der Saga startet. Die Saga wird durch eine mithilfe von associationProperty definierte Property des

```
public class Bestellabwicklung {

  private UUID warenkorbId;
  private boolean rechnungBezahlt = false;
  private boolean versandartBestimmt = false;

  @Inject
  private transient CommandGateway;

  @StartSaga
  @SagaEventHandler(associationProperty = "warenkorbId")
  public void handle(WarenkorbBestellt event) {
     warenkorbId = event.getWarenkorbId();
     String versandnummer = erzeugeVersandnummer();
     String rechnungsnummer = erzeugeRechnungsnummer();
     SagaLifecycle.associateWith("versandnummer", versandnummer);
     SagaLifecycle.associateWith("rechnungsnummer", rechnungsnummer);
     commandGateway.send(new BestimmeVersandart(warenkorbId, versandnummer));
     commandGateway.send(new StelleRechnung(warenkorbId, rechnungsnummer));
  }

  @SagaEventHandler(associationProperty = "versandnummer")
  public void handle(VersandartBestimmt event) {
     versandartBestimmt = true;
     if (rechnungBezahlt) {
        commandGateway.send(new VersendeWare(warenkorbId));
     }
  }

  @SagaEventHandler(associationProperty = "rechnungsnummer")
  public void handle(RechnungBezahlt event) {
     rechnungBezahlt = true;
  if (versandartBestimmt) {
        commandGateway.send(new VersendeWare(warenkorbId));
     }
  }

  @SagaEventHandler(associationProperty = "warenkorbId")
  public void handle(WareVersand event) {
     SagaLifecycle.end();
  }

  // ...

}
```

*Listing 7*

startenden Ereignisses eindeutig identifiziert. Wie im startenden Event Handler und den weiteren `@SagaEventHandler`-Annotationen zu sehen, können aber durch die SagaLifecycle.associateWith-Methode auch andere Properties und deren Werte zur Assoziation von Ereignissen mit einer bestimmten Saga-Instanz herangezogen werden. Zum Auslösen weiterer Kommandos wird auch innerhalb von Sagas einfach das Command Gateway genutzt. Zum Beenden einer Saga steht in SagaLifecycle die „end"-Methode zur Verfügung. Mithilfe des unterstützten Lebenszyklus von Sagas ist es somit möglich, auch langlebige Geschäftsprozesse abzubilden und zu automatisieren.

## Pattern: Interpretation/Projektion

An Stellen, an denen ein Geschäftsprozess manuellen Eingriff beziehungsweise Benutzerinteraktion erfordert, ist es in der Regel notwendig, den Zustand des Systems oder eines Teils des Systems zu visualisieren. Im Beispiel ist das der Fall, wenn ein Kunde seine Rechnung bezahlt oder Waren in den Warenkorb legt. Hier wären beispielsweise die Rechnung oder die Produkte im Warenkorb zu visualisieren. Wie zuvor gesehen, können diese Informationen einfach über Aggregate, die durch Repositories in Axon zugreifbar sind, ermittelt werden.

Manchmal ist der Zugriff auf ein Aggregat jedoch nicht ausreichend. Soll einem Kunden beispielsweise eine Produktempfehlung gemacht werden, die alle Produkte umfasst, die er aus einem Warenkorb entfernt und letztlich noch nie bestellt hat, müssen prinzipiell alle Warenkörbe und Bestellungen durchsucht werden. Als Alternative bietet Axon das Konzept von Query Handlern an. *Listing 8* zeigt eine einfache Implementierung.

Query Handler sind ähnlich wie Event Handler, lösen aber keine Ereignisse oder Kommandos aus. Vielmehr stellen sie Informationen bereit, die zuvor über Event Handler in Lesemodelle persistiert wurden. Dies folgt einem konsequenten Command-Query-Responsibility-Segregation(CQRS)-Ansatz. Aufwendige Datenbankabfragen sind somit nicht nötig. Stattdessen wird der Zustand aus einer Teilmenge der aufgetretenen Ereignisse projiziert. Die benötigten Informationen können so direkt aus dem Lesemodell abgefragt werden.

## Technische Integration

Es zeigt sich, wie einfach mit einigen Building Blocks eine patternbasierte Überführung eines „Post-it-Modells" in eine ausführbare Implementierung ist. Sicherlich sind hier die wichtigen fachlichen Details wegabstrahiert. Dennoch wird deutlich, wie ein solcher Ansatz mindestens ein schnelles Prototyping des Modells erlaubt. Ein solcher Ansatz erfordert sicherlich noch nicht den Einsatz eines Frameworks. Mit einfachen Java-Board-Mitteln lassen sich solche Building Blocks auch schnell selbst entwickeln *[8]*. Spätestens, wenn man aber eine ernsthafte Implementierung auf diese Weise in Erwägung zieht, kann Axon weitere Stärken ausspielen. Denn auch im Bereich der technischen Integration bietet Axon aufgrund seiner guten Schnittstellenabstraktionen vielfältige Konfigurationsmöglichkeiten und eine gute Integration mit anderen Frameworks und Technologien. AMQP-basierte Messaging-Systeme, aber auch Kafka, können ebenso einfach angebunden werden wie JPA-fähige Datenbanken oder eine MongoDB. Aber auch querschnittliche Themen wie Transaktionssicherheit oder Schemamigration werden durch ein Unit-of-Work-Pattern und Event Upcaster gelöst. Vor allem steht jedoch eine sehr gute Integration mittels Spring und Support von Spring Boot zur Verfügung.

## Fazit

Event Storming ist ein Tool, mit dem es möglich ist, auch implementierungsnahe Domänenmodelle zu erarbeiten. Anhand der vorgestellten, immer wiederkehrenden Patterns können diese einfach in eine Implementierung überführt werden. Dies ist besonders einfach mit Frameworks wie Axon zu realisieren, die dafür notwendige Building Blocks bereitstellen und zugleich technische Aspekte nicht außer Acht lassen. Sie erleichtern es, eine allgegenwärtige Sprache (Ubiquitous Language) zu etablieren, die sich vom gemeinsam entwickelten Domänenmodell während des Event Storming bis in die Implementierung wiederfindet.

## Referenzen


[1] *http://dddcommunity.org/book/evans_2003/*
[2] *https://www.eventstorming.com/*
[3] *http://wiki.c2.com/?GangOfFour*
[4] *https://axoniq.io/*
[5] *https://martinfowler.com/bliki/UbiquitousLanguage.html*
[6] *https://martinfowler.com/bliki/CQRS.html*
[7] *https://martinfowler.com/eaaDev/EventSourcing.html*
[8] *https://conciso.de/blog/2018/05/23/kombination-cqrs-event-sourcing-java-teil3/*


Hinweis: Unter *„http://www.ijug.eu/go/javaaktuell/201903/listings"* finden Sie das fehlende *Listing 8*.

**Sven-Torben Janus**
sven-torben.janus@conciso.de

Sven-Torben Janus arbeitet als Senior Software Architect bei der Conciso GmbH, wo er den Bereich Softwarearchitektur mitverantwortet. Er befürwortet einen agilen und praktikablen Entwurf von Softwarearchitekturen. Sein Unbehagen über technologiegetriebene Designs führte ihn zum Domain-driven Design und zur Gründung des DDD Meetups Rhein/Main. Er twittert unter @sventorben und teilt sein Wissen auf Konferenzen, Artikeln und im Blog auf https://conciso.de/blog/.