Lufthansa Industry Solutions

Bachelor Thesis

Analysis of the Advantages and Disadvantages of the Command Query Responsibility Segregation Pattern

for attainment of the academic degree of

Bachelor Of Science

presented by

Martin Eckardt
Löwenstraße 12
20251 Hamburg

Student number: 5322

Lufthansa Industry Solutions
martin.eckardt@lhind.dlh.de

FH Nordakademie

Supervisor:	Dr. Lars Braubach
Date of issue:	January 18, 2016
Date of submission:	February 23, 2016

Table of Contents

List of Figures

List of Code Fragments

List of Appendices

List of Abbreviations

BC	Bounded Context
BDD	Behavior Driven Development
BPMN	Business Process Model and Notation
CQRS	Command Query Responsibility Segregation
CQS	Command-Query Separation
DAO	Data Access Object
DDD	Domain Driven Design
DTO	Data transfer object
ES	Event Sourcing
ORM	Object Relational Mapping
UI	User interface
UUID	Universally Unique Identifier

1 Introduction

The architectural pattern called Command Query Responsibility Segregation (CQRS) emerged in the early 2000’s in the .NET field and was first formulated by Greg Young.[1] Since it was presented in the architecture guidance “Exploring CQRS and Event Sourcing”[2] by Microsoft's Pattern & Practices Group in 2012 the interest increased steadily.

CQRS states a simple idea: Divide the back end into two parts. One that is responsible for processing Commands and ensuring consistency (Command Side) and another that is responsible for responding to queries (Query Side). There are different levels of CQRS. In this thesis the more complex one, where Events are the form of communication which is used by the Command Side to inform the Query Side about changes, is evaluated.[3]

Next to Greg Young, Dino Esposito and Udi Dahan are driving forces that contribute in the further development and publication of CQRS. Even though they all are from the .NET languages the pattern can be implemented in Java as well.

1.1 Objective and Organization of the Study

Lufthansa Industry Solutions is an IT service company for process consulting and system integration. This wholly-owned subsidiary of Lufthansa Group supports companies in selected industries with the digitization of their business processes. On a basis of well-founded knowledge of industry, Lufthansa Industry Solutions develops and operates tailored state-of-the-art solutions. The Norderstedt-based company employs around 1,000 members of staff at several branch offices in Germany, Switzerland and the USA. Its customer base includes more than 150 companies, ranging from SMEs to DAX corporations.[4]

When developing software for the customers a variety of architectural patterns are used. In order to being able to choose the most appropriate pattern newly emerging patterns are observed and evaluated. Therefore, this thesis aims at providing an introduction into CQRS for systems architects and developers. Furthermore, it aims at supporting decision-makers by showing the strengths and states characteristics of systems that profit most from the implementation of CQRS.

In the first part of the thesis the building blocks for a CQRS application are explained. Then an analysis of advantages and disadvantages is conducted using selected criteria. Finally, a list of characteristics of good candidates is presented.

---

[1] Heimeshoff, Jander 2013, p. 4

[2] Betts et al. 2012

[3] Betts et al. 2012, p. 229

[4] Lufthansa Industry Solutions 2016

1.2 Accompanying code examples

This thesis is accompanied by a sample application called “Aviation Management System” which can be found in the **appendix A-1**. It implements the CQRS pattern and was developed using the cqrs-4-java[5] library which is providing base classes for CQRS and was written by the author’s co-worker Michael Schnell.

The application provides functionality to schedule, reschedule and cancel flights. The users may register passengers and make reservations on a flight. If a flight is fully booked and additional reservations are made they are put on a waitlist. They will be confirmed if a reservation is cancelled. Furthermore, the system communicates with external payment systems. No user interface is provided for the application. The functionality is demonstrated solely using unit tests.

The domain of the sample application was chosen in order to demonstrate the basic building blocks and concepts of CQRS and Event Sourcing. It is suitable for that because it fulfills the following criteria:

- It contains several logical units and can therefore be divided into subsystems communicating with each other.
- It includes not only simple business rule, such as rescheduling a flight, but also complex ones, such as cancelling a reservation.
- It is collaborative, meaning that multiple actors operate simultaneously on shared data.
- The system is accessed globally
- Different representations of the same data for different user groups are required

[5] Schnell 2007

2 Theoretical Background

CQRS depends on a variety of other concepts. These are not necessary but usually the combination of these with the CQRS approach brings great benefit to the developed system.

2.1 Domain Driven Design

Domain Driven Design (DDD) is an approach to developing complex software systems with frequently changing business rules, which are expected to last for a long time. In order to give advice about how to successfully master these complex software projects Eric Evans describes a set of techniques in his book “Domain Driven Design – Tackling Complexity in the Heart of Software“. Domain in this context describes the sphere of knowledge about a certain subject.

“Some experts consider the DDD approach to be an essential prerequisite for implementing the CQRS pattern. However, many people can point to projects where they have seen real benefits from implementing the CQRS pattern while not using the DDD approach for the domain analysis and model design. In summary, the DDD approach is not a prerequisite for implementing the CQRS pattern, but in practice they do often go together.”[6]

In the following subchapters some of the concepts and techniques that are relevant to CQRS are described. For additional information on these and DDD in general refer to the books of Eric Evans and Vaugh Vernon.

2.1.1 Ubiquitous Language

In the Ubiquitous Language all the “key domain-related terms”[7] are described. The goal is to encourage the common understanding of the domain for the domain experts and the developers. All project members should commit to using the same language in order to avoid translations and misunderstandings.[8] An example for the definition of a Ubiquitous Language may be found in **appendix A-2**.

The Domain Model is a good starting point for the language. It contains the names of the most common classes and operations.[9] Furthermore, it can be supplemented with technical and business terms. The language is not a static model. When used extensively in meetings, writing, code, documentation and speech the weaknesses of the language will be uncovered. This leads to a continuous adaption of the language and therefore to refactoring of the code and the documentation.[10]

---

[6] Betts et al. 2012, pp. 220–221
[7] Betts et al. 2012, p. 15
[8] Avram, Marinescu 2006, p. 14
[9] Evans 2004, p. 26
[10] Evans 2004, p. 26, 2011, p. 10

2.1.2 Domain Model

The Domain Model is a conceptual model that contains data structures and business logic of a domain. Only the relevant real-world aspects of the domain are abstracted into the model. It may be used to solve problems of the domain.[11] "Isolating the domain implementation is a prerequisite for a domain-driven design."[12] Evans suggests the following building blocks to model the domain:

2.1.2.1 Entities

Objects that are defined by their identity are called Entities. Even though the attributes of an Entity may change it keeps his identity. Two instances of an Entity can be matched even if their attributes differ from another. On the other hand two Entities can be differentiated even though they have the same attributes.[13] Most practically Entities hold some kind of immutable identifier, like an UUID[14]. The importance of matching and differing Entities increases in distributed systems.

Buildings are an example for Entities. They can be identified by their address. Even though some attributes, like the inhabitants or the color of the facade, may change the building keeps his identity.

2.1.2.2 Value Objects

Value Objects do not have conceptual identity[15] and are defined only in terms of their value[16]. Rather than describing a specific element they describe the nature of things. Only the attributes of the Value Object concern us. Since the value defines their identity they are immutable. If the value changes a new Value Object is created and replaces the old object.[17]

The introduction of Value Objects may enhance clarity. In our domain we will model the capacity of a flight as a Value Object. The value range is from including 1 to 300. In contrast to saving the capacity as an integer, the Value Object bundles the validation and error handling, makes the API more readable and increases testability.[18]

---

[11] Evans 2004, p. 60
[12] Evans 2004, p. 56
[13] Evans 2004, p. 66, 2011, p. 18
[14] See glossary
[15] Evans 2004, p. 70
[16] Heimeshoff, Jander 2013, p. 2
[17] Evans 2004, pp. 70–71
[18] Johnsson 2009, at 24:10

Other good candidates for Value Objects are strings with format limitations, such as names, identifier[19] or ZIP codes, as well as integers with limitations[20], such as percentages or quantities greater than zero.[21]

Another motivation for creating Value Objects is the optimization of performance in terms of “copying, sharing, and immutability”[22]. If two objects share the same Value Object they can hold references to the same object.

2.1.2.3 Aggregates

With many users working on the same set of data it may be difficult to avoid conflicts. “The problem is acute in a system with concurrent access to the same objects by multiple clients. With many users consulting and updating different objects in the system we have to prevent simultaneous changes to interdependent objects.”[23] To face this challenge one needs to identify objects that are affected in transactions and separate them from the ones that are independent.

Aggregates are a cluster of related elements, such as Entities and Value Objects.

Every Aggregate has an Aggregate Root. It is a specific Entity of the Aggregate, which is the only one that is accessible from outside of the Aggregate. All other objects in the Aggregate can only be accessed through the Aggregate Root. These objects “have local identity , but it only needs to be unique within the Aggregate, since no outside object can ever see it out of the context of the Root Entity”[24].

Aggregates are clearly separated by consistency boundaries. Invariants within these boundaries are defined for the Aggregate as a whole. The Root is responsible for the enforcement of these. [25]

2.1.2.4 Repositories

Instances of an Aggregate may be stored in a Repository. Therefore, Repositories provide the CRUD operations to create, read, update and delete instances by their Aggregate ID.[26] Only Aggregates have Repositories. Each is used exclusively for a single Aggregate type.[27] Eric Evans calls them an “illusion of an in-memory collection of all objects of that type”.[28]

---

[19] See FlightId class
[20] See Capacity class
[21] Johnsson 2009, at 27:00
[22] Evans 2004, p. 72
[23] Evans 2004, p. 89
[24] Evans 2004, p. 90; Betts et al. 2012, p. 215
[25] Betts et al. 2012, p. 215
[26] See appendix A-4
[27] Vernon 2013, p. 401
[28] Evans 2004, p. 151

The pattern is similar to the Data Access Objects pattern. While the DAO pattern is very database-centric, Repositories rather focus on the domain.

2.1.3 Bounded Contexts

Maintaining a single Domain Model for a large system may not be practical.[29] Subsystems are used by different user groups having different requirements. Therefore, the system should be divided in several Bounded Contexts.[30] Each Bounded Context has its own Domain Model and Ubiquitous Language or respectively a dialect of the language. They only apply within in the Bounded Context.[31] Furthermore, they have their own architecture using appropriate technologies.

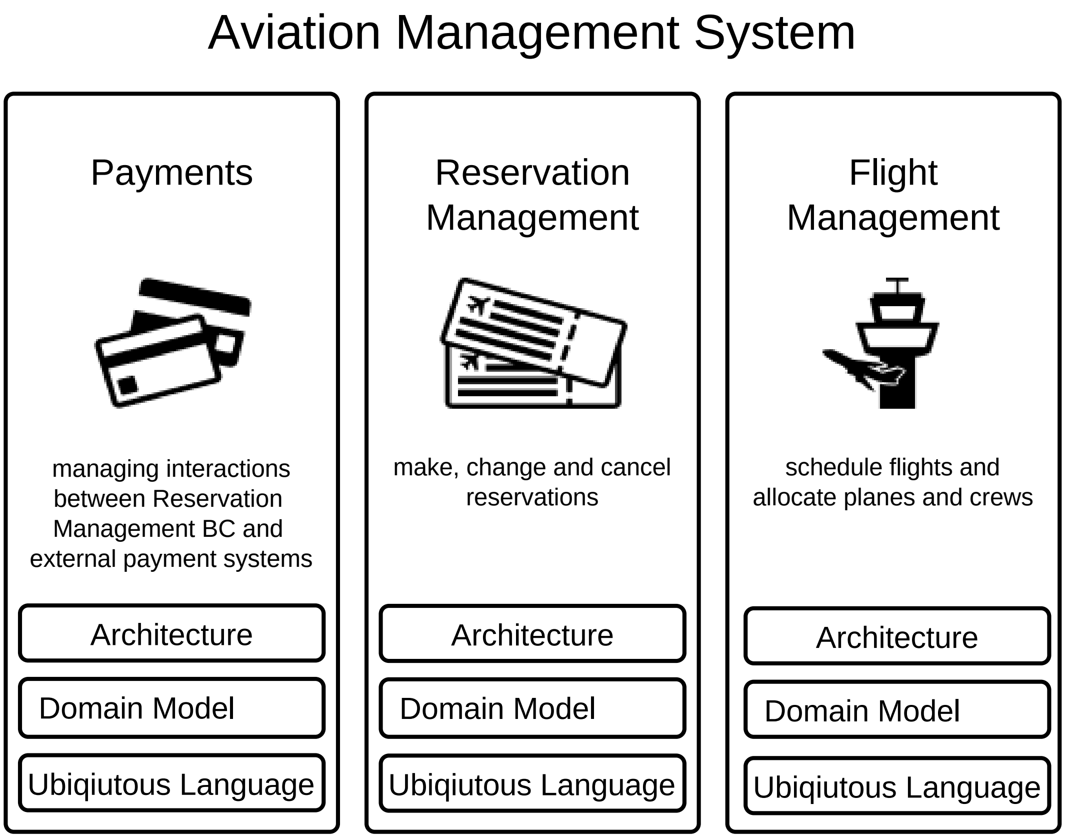

Aviation Management System
Payments
managing interactions between Reservation Management BC and external payment systems
Architecture
Domain Model
Ubiqiutous Language
Reservation Management
make, change and cancel reservations
Architecture
Domain Model
Ubiqiutous Language
Flight Management
schedule flights and allocate planes and crews
Architecture
Domain Model
Ubiqiutous Language

Figure 1: Overview over the Bounded Contexts of the Aviation Management System

The sample system is divided in three subsystems as shown in figure 1. The Reservation Management BC is implemented using CQRS in the sample. For a list of the supported features see **appendix A-3**. The other subsystems are only used to demonstrate the communication between several Bounded Contexts.

[29] Evans 2011, p. 8; Betts et al. 2012, p. 215

[30] Note that this concept differs from the concept of splitting a model in modules. Modules are used to organize a single model.

[31] Betts et al. 2012, p. 8, 2012, p. 215

This division of the system reduces the complexity and simplifies the team cooperation and communication.[32] Furthermore, it allows using appropriate technical architecture in the BC.[33] For some contexts a layered architecture with the CRUD style may be appropriate. In others CQRS may be the better fit.[34]

Some Entities may exist in multiple BCs. However, they may expose different properties since only a subset of them is relevant to the current BC.[35] The flight Entity may expose a list of confirmed reservations and its waitlist in the Reservation Management BC. In the Flight Management BC it may expose the list of assigned captains and flight attendants. This different understanding of the same word needs to be well documented in the Ubiquitous Language.

In large systems with many Bounded Contexts a Context Map that shows the relationships of the BCs clarifies the boundaries between them. The Messages they send and received are visualized.

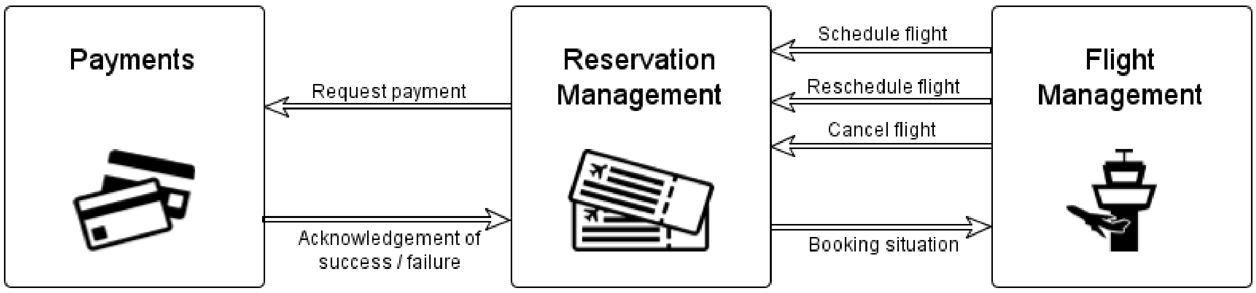


Figure 2: Context Map of the Aviation Management System

While the Flight Management BC is responsible for scheduling the flights and allocating planes and crews it informs the Reservation Management BC about new or changed flights. Additionally, it queries the current booking situation in order to provide enough resources on the flight. The Reservation Management BC allows the users to register passengers and make or cancel reservations. Furthermore, it requests the payment from the Payments BC which includes the logic to communicate with different payment providers, such as credit card companies. They may either be confirmed or rejected.

[32] Avram, Marinescu 2006, p. 69
[33] Betts et al. 2012, p. 219
[34] Betts et al. 2012, p. 217
[35] Betts et al. 2012, p. 218

2.2 Command-Query Separation (CQS)

“Asking a question should not change the answer.”

– Bertrand Meyer (Meyer 1997, p. 751)

In 1997 Bertrand Meyer introduced the idea of Command-Query-Separation in his book “Object-oriented software construction”. It states that a function should be either a Command or a query, but never both.[36]

A Command is expected to alter the state of an object.[37] It does not return any data. The only response may be an exception stating why the Command failed to execute.

In contrary, a Query returns data but does not change the object.[38] Queries do not have side effects and are therefore idempotent.[39] That means if you execute a Query multiple times without issuing a Command in between the result will not change.

The main motivation is avoiding side effects and making the functions referentially transparent.[40] “From the Command-Query Separation principle follows a style of design that yields simple and readable software, and tremendously helps reliability, reusability and extendibility.” [41]

2.3 Message-based Systems

In message-based systems the components communicate through Messages contained in message objects. They have a unique identifier and a timestamp of the time they have been created.[42] Therefore, two instances of a message are recognized by the system components as the same.[43] In the sample implementation Universal Unique Identifiers[44] are used. It has the benefit that a message can be created by the client without requesting an available ID from the back end. Clients that may be disconnected for a period of time are still able to create messages.

Message objects are DTO’s and therefore they do not implement any behavior. The purpose is simply the transfer of data. They are technical constructs.[45] All the business logic is implemented in the back end. To send Messages between distributed system components

[36] Esposito 2015c, p. 10; Meyer 1997, p. 751
[37] Meyer 1997, p. 751
[38] Betts et al. 2012, p. 223
[39] Meyer 1997, p. 752; Young 2014a, at 48:10
[40] Meyer 1997, p. 750
[41] Meyer 1997, p. 752
[42] Esposito 2015b, p. 13
[43] Evans 2011, p. 21
[44] See glossary
[45] Johnsson 2009, at 46:30

serialization and deserialization as xml or json should be supported by the clients and the back end.

DDD suggests modeling the behavior rather than the data of the business domain.[46] In CQRS this suggestion is applied by the use of two types of messages: Event Messages and Command Messages. Every Message is associated with an Aggregate and contains the identifier of the Aggregate as well as the identifier of the Entities within the Aggregate that are involved.[47]

2.3.1 Command Messages

A Command Message contains the request to perform a specific action to the back end. Greg Young compares them with “serializable method calls”[48]. Commands can be issued by users, other system components or by autonomous services that asynchronously interact with the system. They do not return a value. A feedback if the operation was executed successfully or the reason why the Command was not able to be processed may be returned.[49]

Command objects are “data transfer objects, which encapsulate a Command passed to the business logic and all of the data required to execute it”.[50] Their name is always in the imperative tense.[51]

```
public class RescheduleFlightCommand extends AbstractCommand {
  private FlightId flightId;
  private LocalDateTime newDepartureTime;
  private Duration newDuration;

  public RescheduleFlightCommand(FlightId flightId,
      LocalDateTime newDepartureTime, Duration newDuration) {
    super();
    Contract.requireArgNotNull("FLIGHT_ID", flightId);
    Contract.requireArgNotNull("newDepartureTime", newDepartureTime);
    Contract.requireArgNotNull("newDuration", newDuration);
    this.flightId = flightId;
    this.newDepartureTime = newDepartureTime;
    this.newDuration = newDuration;
  }
}
```

Code fragment 1: RescheduleFlightCommand

The back end performs the appropriate action when the message is received.[52] In some cases there may be a permission of a person required to execute a Command. Therefore, the Commands can be stored until they are confirmed as permissible.

[46] Betts et al. 2012, p. 228
[47] Evans 2011, p. 21; Betts et al. 2012, p. 245
[48] Young 2010, p. 11; Esposito 2015d, p. 10
[49] Esposito 2015d, p. 10
[50] Heimeshoff, Jander 2013, p. 2
[51] Betts et al. 2012, p. 14; Esposito 2015d, p. 10; Young 2010, p. 11

2.3.2 Event Messages

An activity in the domain is modeled as an Event Message which states that something has happened in the system typically as a result of the execution of a Command.[53] They should be named accordingly in the past tense.[54] Usually an Event describes a state change. [55]

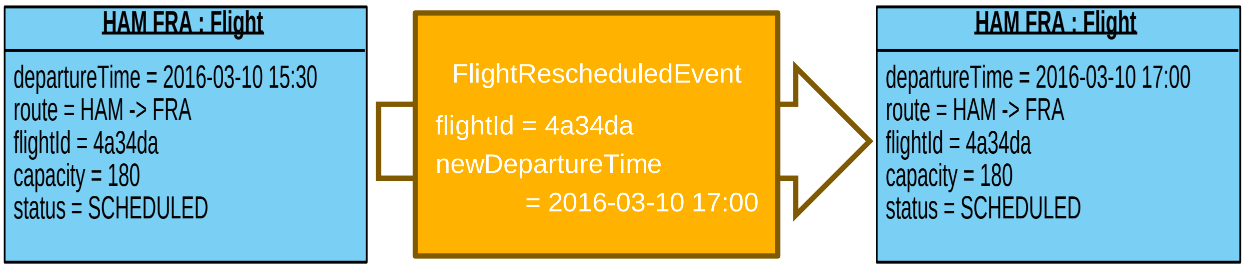


Figure 3: State change of the Flight Entity modelled as FlightRescheduledEvent

Events are associated with Aggregates. Therefore, they hold the identifier of the Aggregate the state change is about.[56] In the example of rescheduling a flight this is the flightId.

An Event "primarily encompasses the reason for it and secondarily the data pertaining to the change of state"[57]. Therefore, the properties contain all the information about change. We may think of it as a notification to other systems or other components of the same system. These components may not only be in the same, but also in other Bounded Contexts.[58] An Event can be processed by multiple receivers in several systems differently.[59] Events are one-way, asynchronous messages. A reply to the sender is not required.[60]

Since Events happened in the past they are immutable and cannot be deleted once they have been published.[61] If an Event Message states a change that needs to be reversed or is incorrect "it is necessary to model a delete explicitly as a new transaction"[62]. To reverse the effect of a ReservationConfirmedEvent one has to issue a CancelReservationCommand which will result in a ConfirmedReservationCancelled Event.

[52] Chatterjee 2004, section "Command Message"; Esposito 2015d, p. 10

[53] Vernon 2013, p. 161; Evans 2011, p. 20; Betts et al. 2012, p. 14

[54] Betts et al. 2012, p. 96; Esposito 2015d, p. 10

[55] Hakim 2012, p. 10

[56] Young 2010, p. 32

[57] Heimeshoff, Jander 2013, p. 2

[58] Betts et al. 2012, p. 286

[59] Betts et al. 2012, p. 229, 2012, p. 235; Chatterjee 2004, section "Event Message"; Young 2014a, at 10:05

[60] Chatterjee 2004, section "Event Message"; Betts et al. 2012, p. 259

[61] Young 2014a, at 11:40; Heimeshoff, Jander 2013, p. 2; Evans 2011, p. 21

[62] Young 2010, p. 31

2.3.3 Design of messages

Messages should be designed to capture the business intent.[63] To give an example we will look at the use case of changing the address of a customer. In a CRUD-Application we could simply create an UpdateAddressCommand. However, in some domains the reason why the address changed may be relevant to the business. Reasons for changing an address may be a misspelling or the fact that the customer actually moved. In this case more concrete Commands like a CorrectCustomerAddressCommand and a RelocateCustomerCommand should be used.[64] The chosen names are part of the **Ubiquitous Language**.[65] Consequently, they should be explicitly modeled in the Domain Model.

If designed properly the Messages define the interface of the system components very clearly and support the separation of concerns.

2.4 Event Sourcing

“There are times when we don't just want to see where we are, we also want to know how we got there.”

– Martin Fowler (Fowler 2005)

The general concept of Event Sourcing is saving the state of an object as a series of Events.[66] These Events representing state changes of Aggregates are captured in an append-only Event Stream.[67]

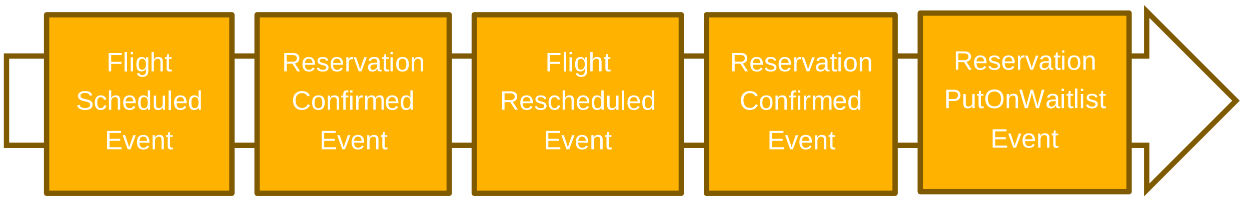

Flight Scheduled Event
Reservation Confirmed Event
Flight Rescheduled Event
Reservation Confirmed Event
Reservation PutOnWaitlist Event

Figure 4: Event Stream of a Flight Aggregate

Many businesses are naturally event sourced. A first example is the balance of a bank account. A system will most likely not only store the current balance, but all transactions, such as money transfers, cash deposits and withdrawals. The balance at any point can be derived from the transactions in the past.[68]

[63] Betts et al. 2012, p. 256
[64] Young 2010, p. 13; Betts et al. 2012, p. 257
[65] Young 2010, p. 25
[66] Heimeshoff, Jander 2013, p. 2; Young 2014a, at 8:20
[67] Vernon 2013, p. 539
[68] Young 2014a, at 41:40; Esposito 2015e, p. 10

To separate concerns within the pattern and provide exchangeability the responsibilities have been divided to the following components:

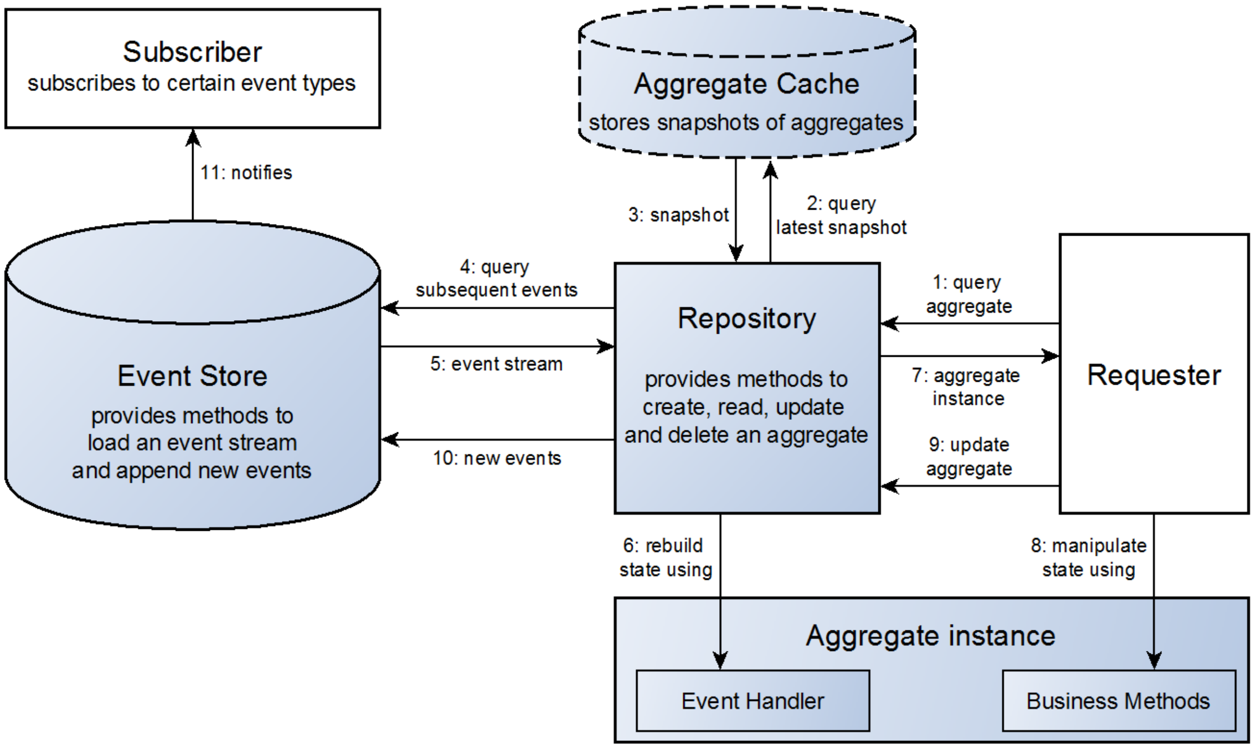


Figure 5: Communication Diagram of the ES Components

2.4.1 Business Methods of Aggregates

The state of an Aggregate is mutated by appending new Events to its Event Stream.[69] These Events are created by the Aggregate itself when a Business Method is called.

```
public class Flight extends AbstractAggregateRoot<FlightId> {

  public void reschedule(LocalDateTime newDepartureTime, Duration newDuration)
      throws DateInPastException {
    validateDepartureTime(newDepartureTime);

    apply(new FlightRescheduledEvent(flightId, newDepartureTime, newDuration));
  }
}
```

Code fragment 2: Reschedule Method resulting in a FlightRescheduledEvent

Every call of a Business Method results either in at least one Event that describes the state change or an exception that describes why it failed.[70] More complex Business Methods may result in several Events.[71]

[69] Vernon 2013, p. 539

2.4.2 Aggregate Event Handler

Every state of an Aggregate may be reconstructed using its Event Stream.[72] Therefore, the Aggregates have Event Handlers to apply the state change the Events represent (6). Since Events happened in the past Event Handlers cannot fail.

```
public class Flight extends AbstractAggregateRoot<FlightId> {

  @ApplyEvent
  private void handle(FlightRescheduledEvent event) {
    this.departureTime = event.getNewDepartureTime();
    this.duration = event.getNewDuration();
  }
}
```

Code fragment 3: Event Handler of Flight Aggregate

2.4.3 Event Store

The mechanisms to query the Event Stream for an Aggregate (4) and append new Events to the Event Stream (10) are bundled in the Event Store. Furthermore, it publishes the new Events to the subscribers (11).[73]

As previously mentioned, Events cannot be deleted. This means the Event Stream is an “Append-Only Immutable Model”.[74] Therefore, write once read many (WORM) data storage devices may be used. They are not only cheaper but also may enhance security, since the data cannot be manipulated.[75]

The Event Store may be implemented SQL, an ORM framework such as hibernate or any other database technology.[76] There are several commercial and open source implementations, such as Event Store[77] with Greg Young as the lead architect, available.

2.4.4 Repository Implementation

The Repository implementation of the DDD is the interface to the components creating, reading, updating and deleting Aggregates.

When an Aggregate is queried from the Repository (1) it loads the Event Stream from the Event Store (4) and constructs the Aggregate by invoking the Aggregate Event Handlers in order (6). It

[70] Vernon 2013, p. 161
[71] See cancelReservation() method of Flight class and figure 13
[72] Young 2014c, at 2:30
[73] Young 2010, p. 41, 2010, p. 46
[74] Young 2014c, at 6:20; Young 2014a, at 42:33
[75] Young 2014a, at 31:20
[76] Esposito 2015b, p. 12
[77] See https://geteventstore.com/

may offer the option to query certain versions of an Aggregate. In this case not all Events of the Aggregate are loaded and applied. When the Aggregate is persisted with the `update()` method (9), the newly created Events that describe the state changes are appended to the Event Stream in the Event Store (10).[78]

2.4.5 Aggregate Cache

If an Aggregate is likely to accumulate a large number of Events over time, it would be not very efficient to query and replay all Events each time it is queried. The optional concept of the Aggregate Cache suggests saving a snapshot of the state of an Aggregate after a specific number of Events. Since the Events are immutable a snapshot will always be valid.[79]

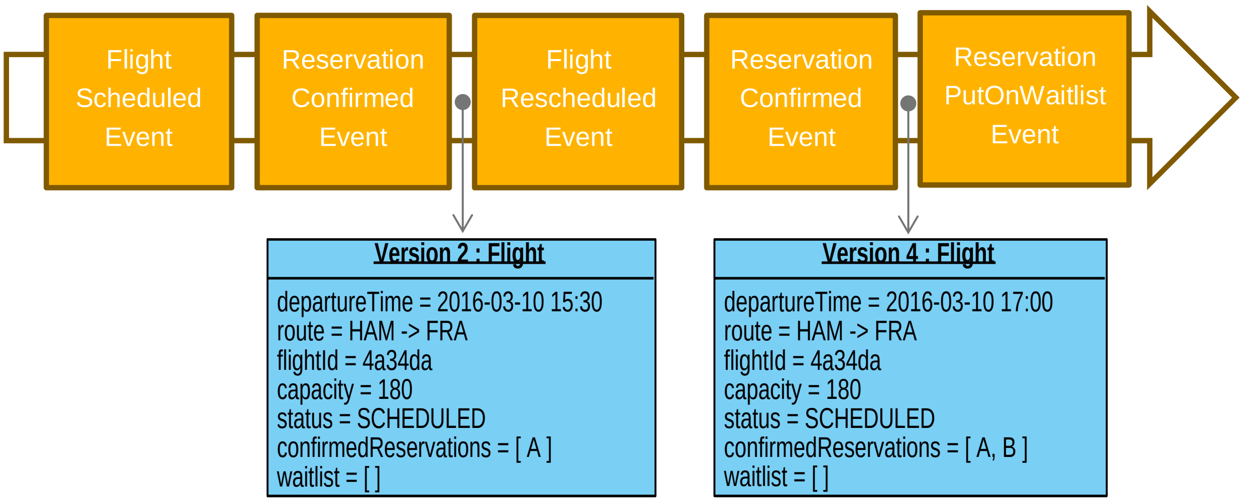


Figure 6: Event Stream with Rolling Snapshots

When the Aggregate is queried from the Repository, the most recent snapshot is retrieved from the Aggregate Cache (2). Instead of all Events only the subsequent Events are loaded from the Event Store (4) and applied to the latest snapshot (6).[80]

The process handling the creation of snapshots runs asynchronously in the background.[81] The snapshot itself contains the identifier of the Aggregate, the version as well as the serialized Aggregate instance.[82] The ideal number of Events between two snapshots is individual for every system and depends on the ratio between reads and writes.

[78] Vernon 2013, pp. 160–162
[79] Young 2014a, at 36:40
[80] Betts et al. 2012, p. 166; Young 2014a, at 36:40
[81] Young 2010, p. 35
[82] Vernon 2013, p. 161; Young 2010, p. 44

2.5 Command Query Responsibility Segregation (CQRS)

Based on the Command-Query Separation principle for functions Greg Young described the architecture precept of CQRS. It suggests splitting the system in two parts: the Command and the Query Side. In some literature they are called write and read site. Each side has its own Domain Model and uses patterns and technologies that suit the needs best. The two sides may even run on two distinct tiers and be optimized separately without affecting each other.[83]

CQRS is not intended for use as the top-level architecture of a system. The pattern should be rather applied to subsystems which gain benefit from the separation.[84]

2.5.1 Overview

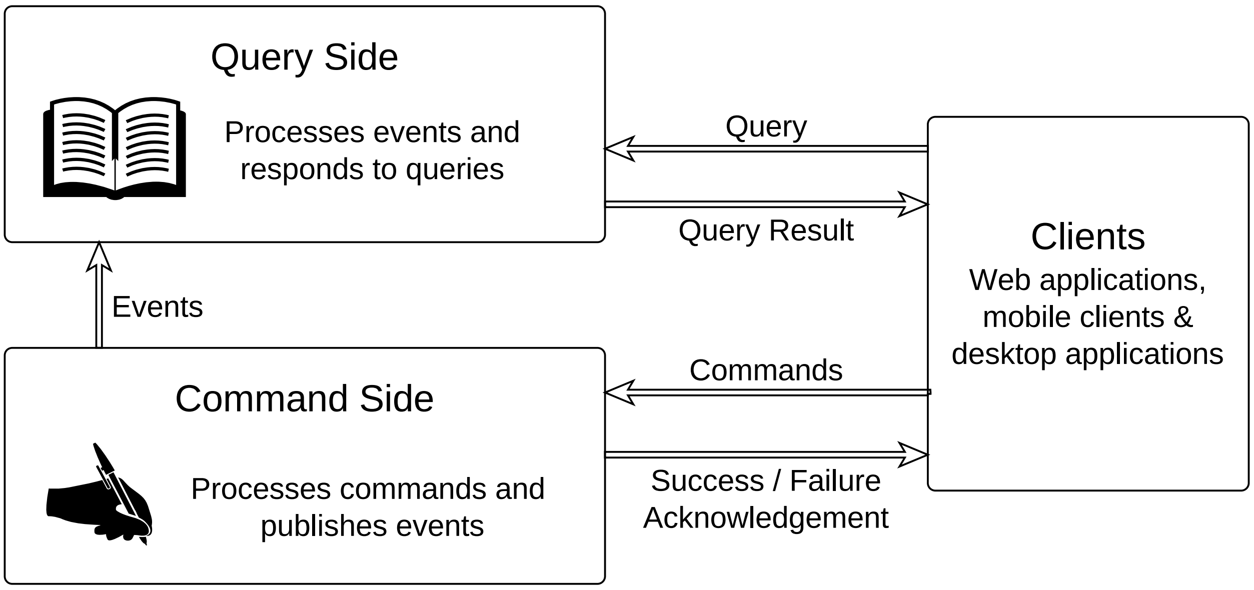

Query Side
Processes events and responds to queries
Query
Query Result
Clients
Web applications, mobile clients & desktop applications
Events
Commands
Command Side
Processes commands and publishes events
Success / Failure Acknowledgement

Figure 7: Overview over CQRS

The system is structured in three parts. The clients on the far right may be web applications, mobile clients or desktop applications.[85] They send Commands to the Command side. In our example implementation a MakeReservationCommand could be issued to make a reservation for a passenger on a flight.

2.5.2 Command Side

The Command Side processes the Command. If no error occurs it acknowledges the successful execution to the client and publishes one or more Events to the Query Side. If an error occurred, it acknowledges the failure to the client.

[83] Esposito 2015c, p. 10
[84] Betts et al. 2012, p. 211, 224; Heimeshoff, Jander 2013, p. 3; Esposito 2015c, p. 14; Fowler 2011
[85] Vernon 2013, p. 140

In the sample implementation we use the Event Sourcing pattern described earlier for the Command side. Therefore, an Event Store and a Repository are implemented.

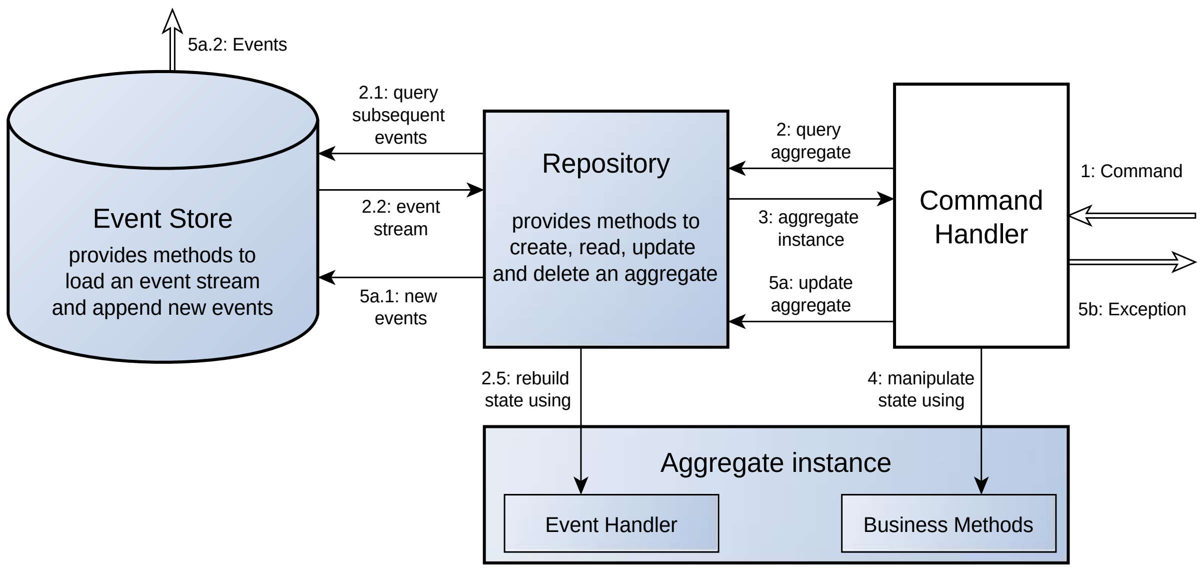


Figure 8: Overview over the Command Side

2.5.2.1 Command Handler

The Command Handlers build a thin interface layer that is responsible for executing the Commands. Furthermore, they are responsible for authorization, logging and exception handling.[86] Exceptions may occur when retrieving from (2) or persisting (5a) an Aggregate instance to the Repository or when executing the Business Method of the Aggregate (4). If an exception occurs the Command Handler acknowledges the failure to the client.

[86] Heimeshoff, Jander 2013, p. 2

```java
public class FlightCommandHandler extends AbstractCommandHandler {

  private FlightRepository flightRepository;

  public void handle(RescheduleFlightCommand cmd) throws DateInPastException {
    // Check Authorization
    checkAuthorization(cmd);

    // Retrieving Aggregate instance from Repository
    Flight flight = flightRepository.read(cmd.getFlightId());

    // Invoking Business Method of Aggregate
    flight.reschedule(cmd.getNewDepartureTime(), cmd.getNewDuration());

    // Persisting Aggregate instance
    flightRepository.update(flight, null);
  }
}
```

Code fragment 4: Excerpt from the FlightCommandHandler

When a Command is received, the Command Handler loads the required Aggregate from the Repository, invokes the method and updates the Aggregate if no exception occurred.

With this implementation the entire mapping from Commands to the methods invoked on the Write Model is collected in one place. Every Aggregate has its own Command Handler.

2.5.2.2 Write Model

The Command Side has a single normalized model called the Write Model which maps the business logic to a program structure. It is frequently in the shape of an object-oriented Domain Model.[87] However, it may also be build using a different programming paradigm, such as functional programming.

[87] Heimeshoff, Jander 2013, p. 1

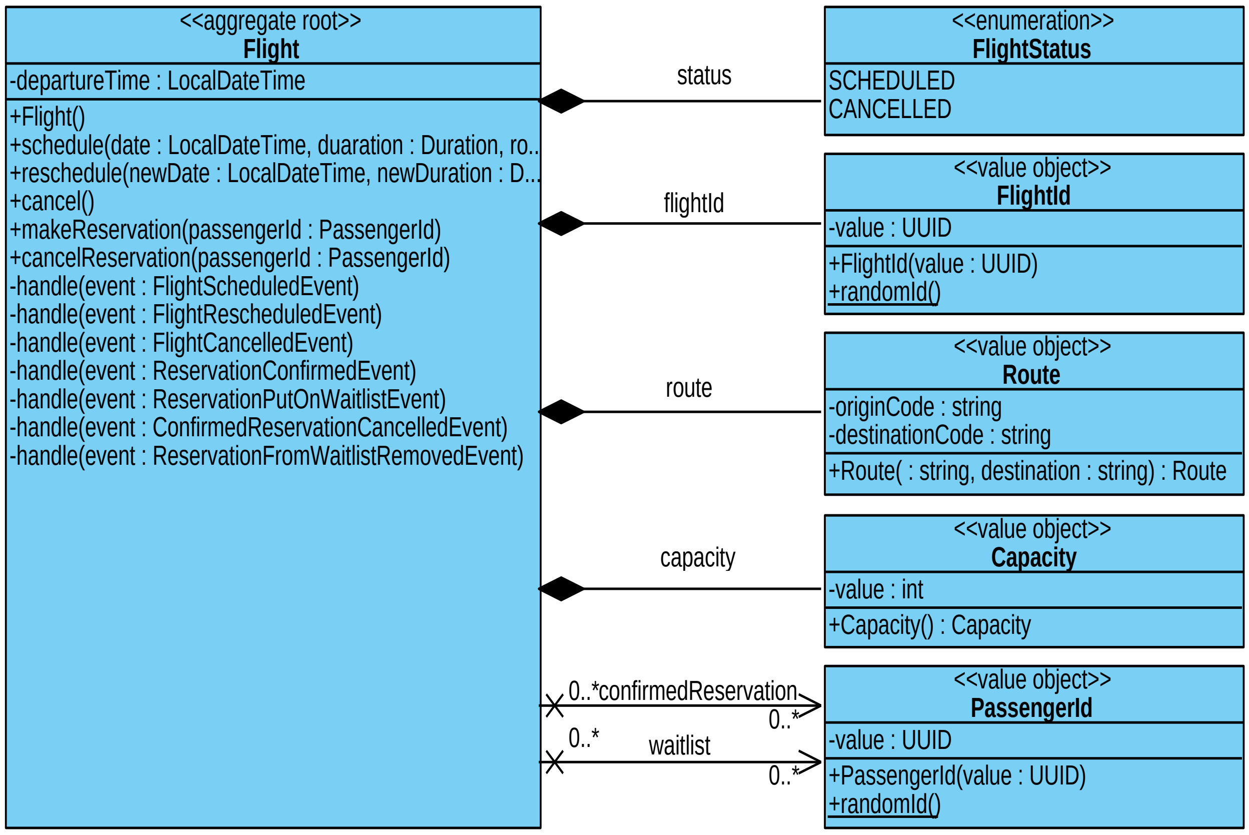

<<aggregate root>>
Flight
-departureTime : LocalDateTime
+Flight()
+schedule(date : LocalDateTime, duaration : Duration, ro...
+reschedule(newDate : LocalDateTime, newDuration : D...
+cancel()
+makeReservation(passengerId : PassengerId)
+cancelReservation(passengerId : PassengerId)
-handle(event : FlightScheduledEvent)
-handle(event : FlightRescheduledEvent)
-handle(event : FlightCancelledEvent)
-handle(event : ReservationConfirmedEvent)
-handle(event : ReservationPutOnWaitlistEvent)
-handle(event : ConfirmedReservationCancelledEvent)
-handle(event : ReservationFromWaitlistRemovedEvent)
status
flightId
route
capacity
0..*confirmedReservation
0..*
0..*
waitlist
0..*
<<enumeration>>
FlightStatus
SCHEDULED
CANCELLED
<<value object>>
FlightId
-value : UUID
+FlightId(value : UUID)
+randomId()
<<value object>>
Route
-originCode : string
-destinationCode : string
+Route(: string, destination : string) : Route
<<value object>>
Capacity
-value : int
+Capacity() : Capacity
<<value object>>
PassengerId
-value : UUID
+PassengerId(value : UUID)
+randomId()

Figure 9: Class Diagram of the Flight Aggregate in the Write Model

Our Write Model contains two Aggregates: The Passenger and the Flight Aggregate. Both their Aggregate Roots are named like the Aggregate. The Flight Aggregate contains Value Objects and two collections containing the identifier of passenger Aggregate instances. The first collection contains the identifier of the passengers with a confirmed reservation. The second collection contains the identifier of the passengers that are on the waitlist.

Since Event Sourcing is used as the data storage mechanism for the Command Side the Aggregate Root contains not only Business Methods resulting in Events but also Event Handlers to rebuild the state.

The model contains exclusively the core business logic. Read operations with query logic are not implemented.

2.5.2.3 Workflows and Sagas

Complex processes that involve multiple Aggregates are modeled in Workflows, which are also called Sagas. In the Workflow the Message flow is defined. The Process Manager is responsible for calling the Business Methods of the involved Aggregates, as well as receiving Events.[88] It does not implement any business logic, since it belongs in the according Aggregate.

[88] Betts et al. 2012, pp. 288–289

Workflows are implemented in Process Managers. They are only aware of the Messages they send and receive. They have a set of rules describing which Messages to send next when a certain Message is received. Workflows may be linear or as shown in the following example they have multiple possible outcomes.

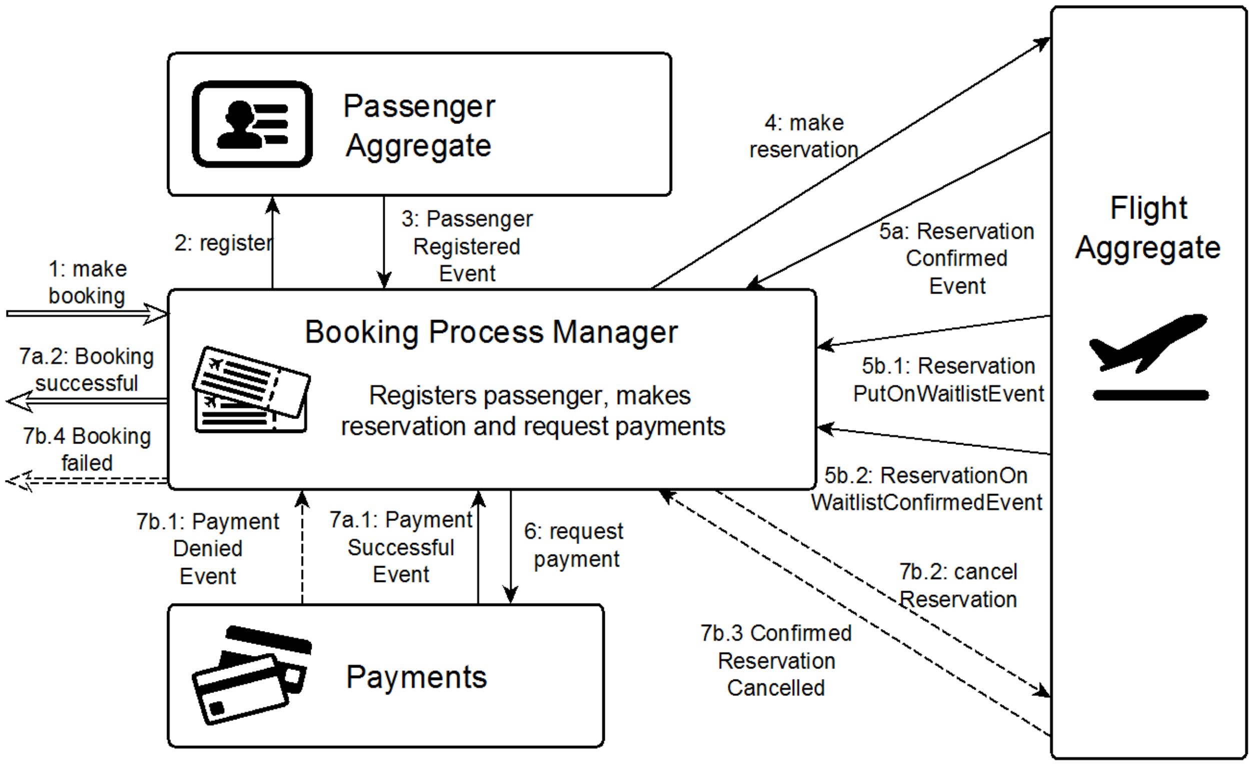

Passenger Aggregate
Flight Aggregate
Booking Process Manager
Registers passenger, makes reservation and request payments
Payments
1: make booking
2: register
3: Passenger Registered Event
4: make reservation
5a: Reservation Confirmed Event
5b.1: Reservation PutOnWaitlistEvent
5b.2: ReservationOn WaitlistConfirmedEvent
6: request payment
7a.1: Payment Successful Event
7a.2: Booking successful
7b.1: Payment Denied Event
7b.2: cancel Reservation
7b.3 Confirmed Reservation Cancelled
7b.4 Booking failed

Figure 10: Overview of the MakeBooking Process

Figure 10 shows the process of making a booking. This process includes registering the passenger, making the reservation, arrange the payment and confirming the success or failure. After the process is triggered (1), the passenger is registered (2). Then the reservation on the desired flight is made (4). If the reservation is confirmed right away (5a), the payment is requested (6). If there are not enough seats available the flight will publish will a ReservationPutOnWaitlistEvent (5b.1). In this case the Process Manager simply waits and the process is on hold. If the reservation is confirmed Eventually, because another reservation was cancelled, the flight publishes a ReservationOnWaitlistConfirmedEvent (5b.2). The process continues with the payment. If the payment is successful (7a.1), the booking is confirmed (7a.2). If the payment was denied (7b.1) the reservation is cancelled (7b.2) and the failure is acknowledged to the customer (7b.4).

2.5.3 Query Side

The Query Side has two responsibilities. Firstly, it consumes the Events published by the Command Side and builds Query Models from them. Secondly, it answers the Queries issued by the clients. Since Queries may be of very different nature, multiple Projections with different

Query Models may be useful. For different use cases there may be different representations of an Aggregate.[89] Therefore, each Projection has its own Query Model.

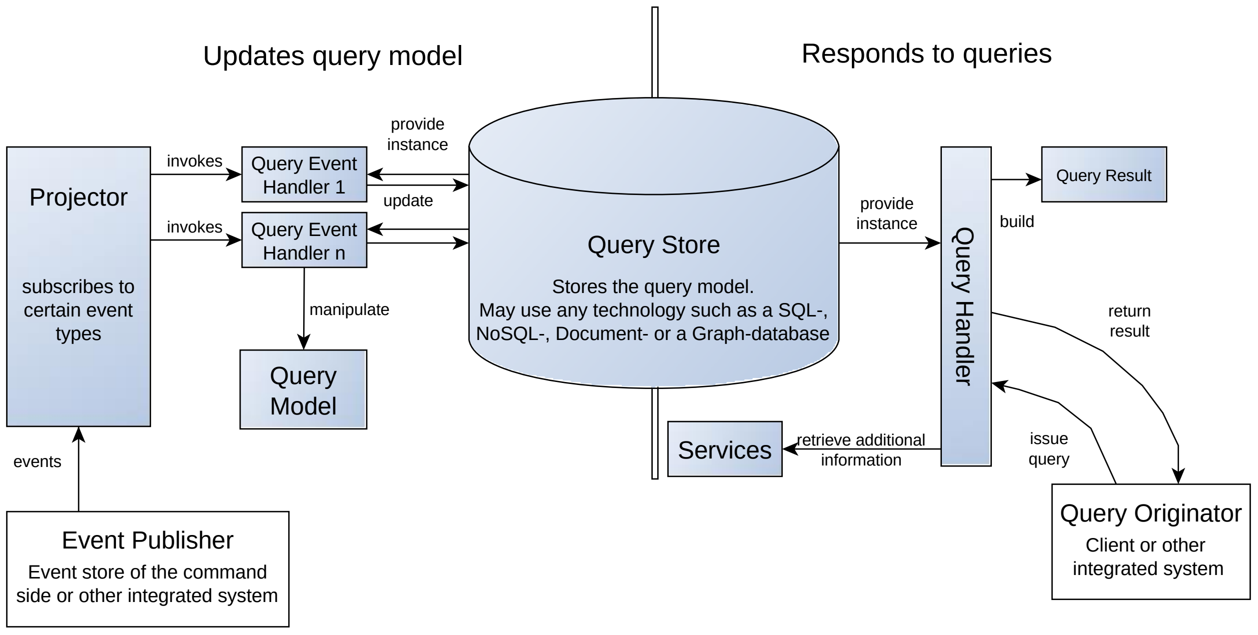


Figure 11: Overview over the Query Side

2.5.3.1 Projections

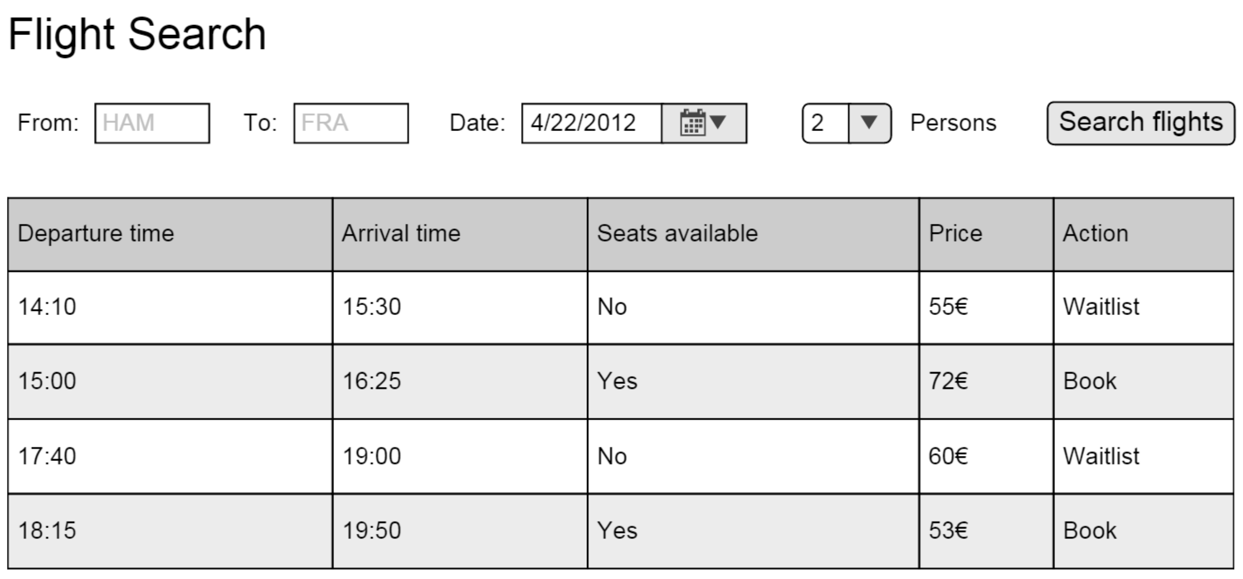
Projections are optimized to serve a set of related query operations. Their models are not built from the Write Model but from the Events. The Query Side may have several Projections each interpreting the same Events differently.[90] They create a transient state from an Event Stream to answer a specific question. Every Projection may use an appropriate technology to store the Query Model. This may be a SQL-database or a graph database, such as neo4j.[91]

In our example implementation we will offer a feature in the client to search for flights. He may search for flights from an origin to a destination on a specific date. If a flight has been cancelled, it is not shown. For each flight it is indicated if sufficient seats are available on the flight. Additionally the price of the flight is displayed.

[89] Betts et al. 2012, p. 248

[90] Young 2010, p. 30; Betts et al. 2012, p. 241; See specification of a very different projection in appendix A-7 and the implementation in the accompanying code example.

[91] Young 2014a, at 20:20


Flight Search

From: HAM To: FRA Date: 4/22/2012 2 Persons Search flights

Departure time	Arrival time	Seats available	Price	Action
14:10	15:30	No	55€	Waitlist
15:00	16:25	Yes	72€	Book
17:40	19:00	No	60€	Waitlist
18:15	19:50	Yes	53€	Book

Figure 12: Mockup of the Flight Search

2.5.3.2 Query Models and Query Results

The Query Model is a denormalized data model that is structured to serve the Queries. The data structure is modelled in order to serve the Query without the need of transformation.[92] The Query Models may differ very strongly from another to serve the queries best. To demonstrate the different natures of Query Models two additional specifications of Projections with Query Models may be found in the appendices **A-6** and **A-7**.

In order to answer the Flight Search Queries the Projection needs to store a record of all the flights with the number of available seats. To avoid confusion with the Entity of the Write Model the Entity is named QueryFlight. Even though they are similar, they differ in some aspects:

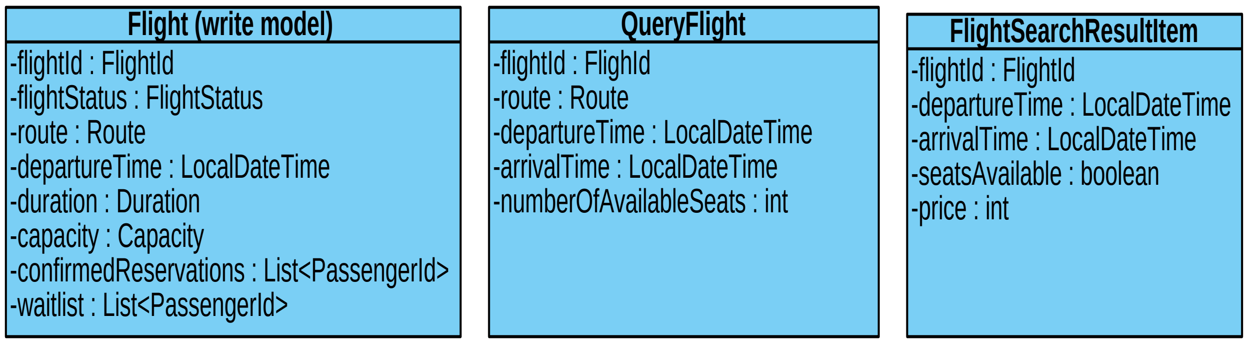

Flight (write model)
-flightId : FlightId
-flightStatus : FlightStatus
-route : Route
-departureTime : LocalDateTime
-duration : Duration
-capacity : Capacity
-confirmedReservations : List<PassengerId>
-waitlist : List<PassengerId>
QueryFlight
-flightId : FlighId
-route : Route
-departureTime : LocalDateTime
-arrivalTime : LocalDateTime
-numberOfAvailableSeats : int
FlightSearchResultItem
-flightId : FlightId
-departureTime : LocalDateTime
-arrivalTime : LocalDateTime
-seatsAvailable : boolean
-price : int

Figure 13: Comparison of Flight Representations

All classes are representing a flight. However, they expose the properties differently. The flight Entity in the Write Model contains all properties to ensure consistency. The QueryFlight stores the arrival time directly instead of the departure time and the duration. This way no calculation is

[92] Vernon 2013, p. 140

necessary when the flight is queried. Furthermore, it contains the number of available seats on the flight. This number is initially the capacity of the flight and adjusted every time a booking has been confirmed or cancelled.

In some cases the Query Model may be returned directly.[93] In this case a conversion to a Query Result is necessary because the model contains sensitive information about the number of available seats that should not be made publicly available. As shown in figure 9 the Query Result should only contain a Boolean value indicating if enough seats are available for the amount queried. Furthermore, the search result is not required to store information about the route since the flights are queried by the route and therefore the route is already known to the client. Additionally, it contains the price of the booking which is retrieved from an external service.

2.5.3.3 Query Event Handler

Each Query Event Handler handles a single Event type for the Projection. They are invoked when an Event of the type is received. The handler loads the Query Model from the Query Store and invokes the according methods to update the Query Model.

The update of the Query Model implements the push model. Whenever an Event is published by the Command Side all Projections are notified and update immediately.[94]

2.5.3.4 Query Handler

The Query Handlers are the interface to the client which is responsible for authorization, validation of the query parameters, and returning the result.[95] The query parameters may be handed in separately or captured in an object similar to Commands. In this case they could be persisted to log the issued queries.

Furthermore, they transform the Query Model to the Query Result. The Query Results are simple DTO’s without any logic. They may hide information of the Query Model, such as the number of available seats in the example, or add additional information that cannot be retrieved from the model, such as the price for the flight. In some cases this transformation may not be necessary since the Query Model has the necessary structure and may be returned directly.

[93] See the second example of an Airport Schedule projection in appendix A-7 and the accompanying code example.

[94] Heimeshoff, Jander 2013, p. 2

[95] Heimeshoff, Jander 2013, p. 2

3 Analysis of Advantages and Disadvantages

In this chapter the effect of implementing the CQRS patter on the design, the development, the testability and the operational management of the systems is evaluated. Furthermore, the entry barriers, costs and the value to the business are analyzed.

The following sub categories have been selected in close cooperation with the authors training organization. The ISO/IEC 25010 norm on “System and software quality models” served as an inspiration.

3.1 System Design

CQRS has great impact on the overall system design. Firstly, it structures the components very clearly and separates their concerns. In the following subchapters some greater impacts on the system are described.

3.1.1 Scalability

Due to the strict separation of the Command and the Query Side they may be scaled independently.[96] In many enterprise systems, the number of Queries usually outweighs the number of Commands.[97] The Query Side may be duplicated to handle a large amount of Queries. A load balancer distributes the workload across the multiple instances.[98]

3.1.2 Distributability

Since the components are clearly separated they may be distributed independently. Therefore, the Command and the Query Side may run on distinct physical servers.[99] To support faster querying several instances of the read models may be distributed globally to provide a close by server for the users of each region.[100] Furthermore, other components such as the Event Store may be separated and distributed.

3.1.3 Interoperability

Interoperability describes the degree to which multiple systems are able to exchange information.[101]

[96] Fowler 2011
[97] Betts et al. 2012, p. 232; Young 2010, p. 19
[98] Vernon 2013, p. 162; Young 2014a, at 53:45
[99] Esposito 2015d, p. 10
[100] Young 2014a, at 54:00
[101] ISO/IEC 25010, § 4.2.3.2

By defining the Commands and Events explicitly as interfaces of the Command and Query Side the integration of other subsystems should be done using these. Accordingly, processes in the systems may be triggered by sending Commands to the Command side. Therefore, the Command and Query Handler may be implemented as an REST-API.

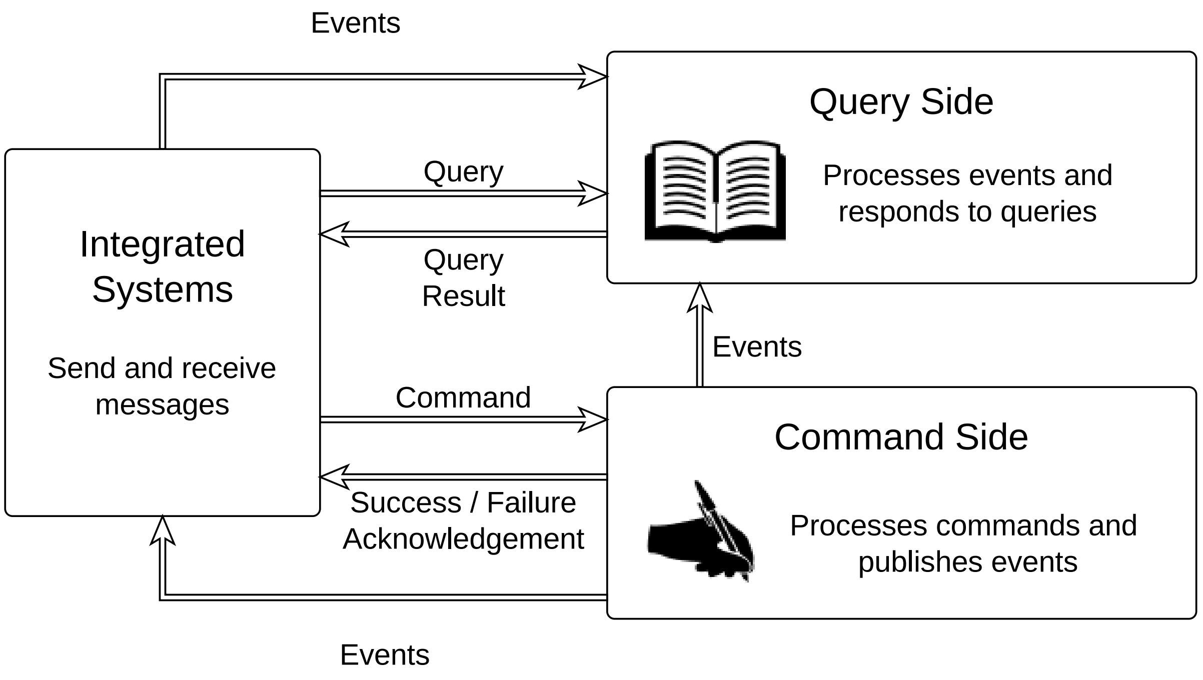

Events
Query Side
Processes events and responds to queries
Integrated Systems
Send and receive messages
Query
Query Result
Events
Command
Command Side
Success / Failure Acknowledgement
Processes commands and publishes events
Events

Figure 14: Integration of other Systems

The Events provide a useful medium of communication with other integrated system. They may be used to trigger processes in other systems that have subscribed to certain Event types.[102] Therefore, they can be posted to an REST-API of the subscriber. This is also known as the push model where the interested party is notified instead of querying for news (pull).[103] Furthermore, they may send their own Events to the Query Side.

Alternatively, a Projection with a Query Model may be implemented that collects the desired data and is queried by the interested party when needed.

3.1.4 Performance

The performance has to be examined in two dimensions: the read and the write performance.

The read performance is vastly increased compared to systems with a fully normalized data model. Whenever an Event is published all the concerning Projections are updated. This means

[102] Betts et al. 2012, p. 241
[103] Young 2010, p. 53

the Query Models are updated asynchronously.[104] Consequently, the result can be retrieved without the execution of long-running calculations and queries when it is queried.[105]

However, the additional time for loading the Events and recreating the Aggregate states from the Events has to be taken in consideration for the write performance.[106] The Aggregate Cache providing snapshots is a good optimization for systems with a large number of Events per Aggregate.[107]

In conclusion, the CQRS pattern makes sense for systems with more reads than writes in the perspective of performance. As mentioned earlier, this is the case for the vast majority of systems.

3.1.5 Low Coupling

Since the interfaces of the components are clearly defined by the Messages they are sending and that they are able the handle they can be reused independently. This is one of the main benefits of message-based systems.[108]

Additionally, the components of the system are internally structured very clearly and the concerns are well separated. For example the Command Handler does not know about Event Sourcing since all the functionality is implemented in the Repository and the Aggregates. This guarantees the exchangeability of components.

3.1.6 Availability

Availability describes the degree of which a system "is operational and accessible when required for use".[109]

CQRS enables the system to have multiple instances of the Query and the Command Side in operation simultaneously. Therefore, if a single instance breaks down the system can still operate normally by compensating the breakdown with the operating instances.

3.1.7 Consistency

Since the Write Model is a single normalized model it is always consistent. However, the Query Models are only eventually consistent due to the asynchronous updates.[110]

---

[104] Vernon 2013, p. 146
[105] Esposito 2015e, p. 13
[106] Betts et al. 2012, p. 242
[107] Young 2010, p. 46
[108] Hendrickson et al. 2005, p. 1
[109] ISO/IEC 25010, § 4.2.5.2
[110] Vernon 2013, p. 146

Eventual consistency means that if additional updates are made to an Aggregate the Query Models will be eventually consistent. This is a weak model. Especially in very collaborative domains it is never possible to rule out that the displayed data is stale.[111] However, Greg Young points out that most distributed systems are already eventual consistent.[112]

Especially directly after submitting a Command and viewing the data afterwards the data will most probably not be in the state after execution of the Command. Therefore, Udi Dahan suggests updating the data according to the issued Command in the client.[113]

3.1.8 Offline Use

Even though clients may be disconnected temporarily from other nodes of the system, they are still able to work with the data in their cache. All Commands that are issued by the user are stored while the client is disconnected, then transmitted to the Command Side and executed. If the Command fails the client notifies the user.

3.1.9 Conflict Detection and Resolution

Concurrent modifications of Aggregates are simple to detect. A Command does not only contain the identifier but also the version of the Aggregate that is displayed to the user at the time of the issuance. Therefore, the Command Handler can see if Command was issued based in the latest state. After the detection a set of rules determines if a conflict exists. There are basically three scenarios:

Firstly, there is a clear conflict present. In the sample implementation a user can make a reservation. In the meantime the flight that was selected by the user has been rescheduled. When the MakeReservationCommand is received, the system can detect that a FlightRescheduledEvent had occurred in the meantime. The user is informed and asked to either make the reservation anyways or pick a new flight.

Secondly, there may be situations where Commands do not conflict with certain changes in the meantime. An example is the CancelReservationCommand. Even if the flight has been rescheduled while the user was seeing the flight itinerary and deciding to cancel the flight it probably has no effect on his decision. Therefore, the cancellation can be processed anyways.

Finally, there are some cases the decision if two Commands are conflicting depends on the specific situation. If two users book the same flight simultaneously and the Flight Search Projection stated that the seats are available, it depends on the fact if sufficient seats for both of them are available.

[111] Bailis, Ghodsi 2013
[112] Young 2014a, at 51:45
[113] Vernon 2013, p. 146

3.1.10 Security

Security as defined by ISO/IEC 25010 means that the “systems protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization”[114].

CQRS system are particular capable to implement authorization functionality. Since all communication to external components goes through the Command and Query Handler authorization is their responsibility. On the lowest level they may check if the originator of the message is allowed to issue the Message. This way a set of Commands and queries can be made explicitly available to certain user groups.

Furthermore, the Message may be validated depending on the originator. An example for the sample application would be that some user groups are allowed to reschedule the flight only in a time frame of 2 hours and others without limitations.

In certain cases it may be necessary to limit the number of Commands of a type a user can issue within a certain amount of time.

Additionally, security includes the protection and information of data that is transmitted between the system components.[115] This quality may be fulfilled by implementing encryption and decryption of the Messages.

3.1.11 Use of the appropriate Technology

The components of the system communicate using Messages. Therefore, the only requirement to them is the ability to serialize these Messages to the format that is understood by the other components (e.g. xml or JSON). This allows the system architect to select appropriate technologies for each component. This may include different programming languages, database systems or use of hardware.[116]

Especially on the Query Side different types of data storage may be suitable for each of the Query Models. These include the system memory for frequently used data sets, document databases for lists and detailed records, as well as relational and OLAP databases for large data sets serving ad hoc queries.[117]

3.2 Development

Following the CQRS principles has impact on the development of the system. In the following subchapters the opportunities that arise because of the CQRS pattern are described.

---

[114] ISO/IEC 25010, § 4.2.6

[115] ISO/IEC 25010, § 4.2.6 Note 1 to entry

[116] Hendrickson et al. 2005, p. 1; Heimeshoff, Jander 2013, p. 2; Young 2014a, 55:40

[117] Esposito 2015d, p. 13

3.2.1 Specification

The specification of the business logic is a key challenge developing software. Good ways to communicate with domain experts are process modeling tools. In the CQRS context the Business Process Model and Notation (BPMN)[118] proved to be very suitable.

The focus of the specification is on the behavior, which is expressed in the Ubiquitous Language throughout the process models. The design of the Messages is a central step of the system design.[119]

Since BPMN supports throwing and catching Events[120], the behavior of the Write Model may be visualized using very clear elements. The Events created by the Business Methods of Aggregate may be modelled as intermediate or end throwing Events, as shown in the example in figure 14. Additionally, exceptions can be modeled as errors[121] and Commands as messages[122].

When specifying the business rules and the processes, the Events and Commands may be extracted from the BPMN process diagram.

[118] BPMN 2.0.2
[119] Vernon 2013, p. 540
[120] See BPMN 2.0.2, § 8.4.5
[121] See BPMN 2.0.2, § 8.4.3
[122] See BPMN 2.0.2, § 8.4.11

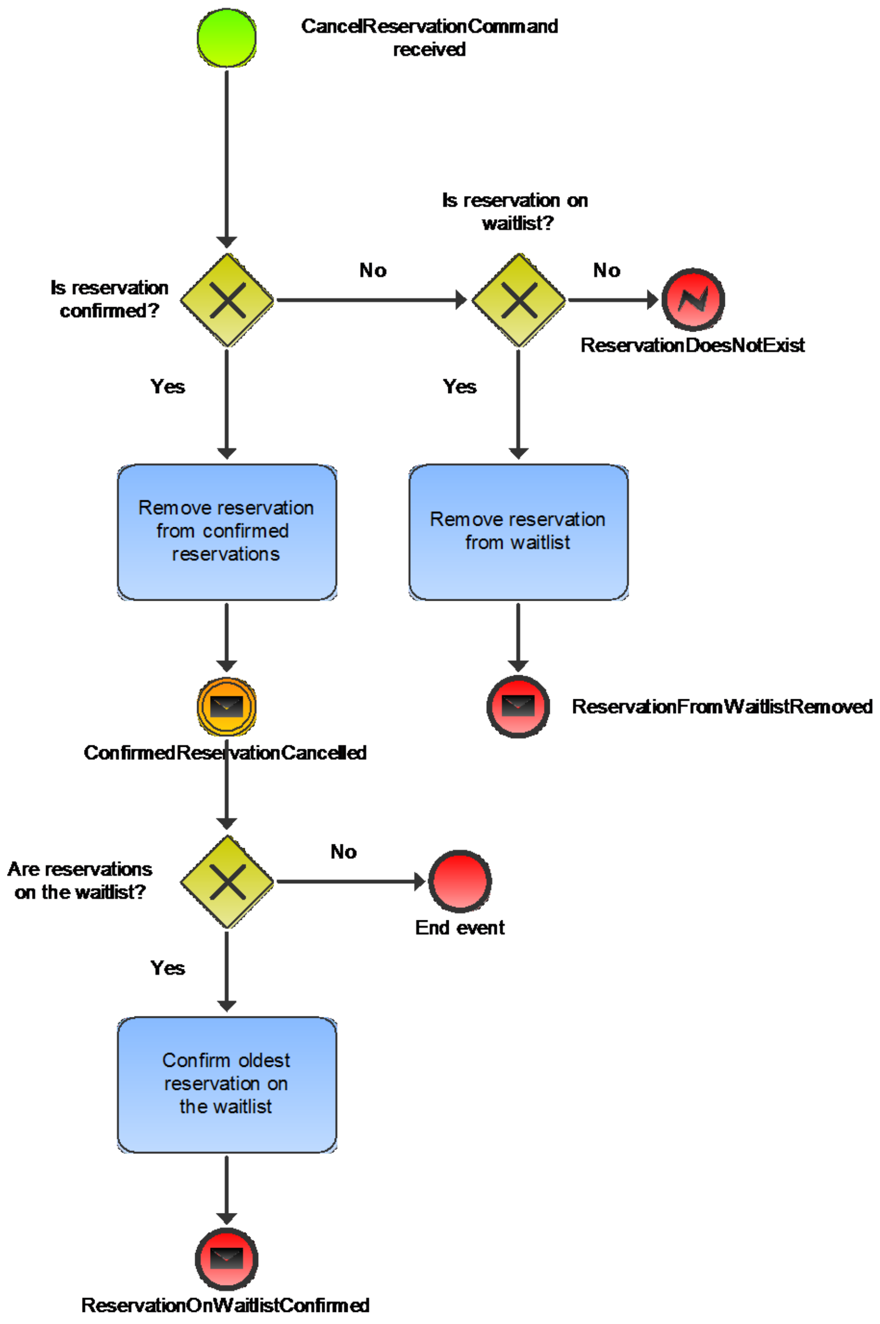

CancelReservationCommand received
Is reservation confirmed?
No
Is reservation on waitlist?
No
ReservationDoesNotExist
Yes
Yes
Remove reservation from confirmed reservations
Remove reservation from waitlist
ConfirmedReservationCancelled
ReservationFromWaitlistRemoved
Are reservations on the waitlist?
No
End event
Yes
Confirm oldest reservation on the waitlist
ReservationOnWaitlistConfirmed

Figure 15: BPMN Process Diagram of the CancelReservation Process created with ARIS

3.2.2 Complexity

Designing and implementing not only the business logic but also the query logic in complex areas of the domain can lead to a devious Domain Model. Usually an operation requires either the business logic or the query logic. If more issues like multiple user access, shared data, transactions, performance as well as staleness of the data are addressed in a single model the complexity grows to a degree where it becomes unmanageable. The separation of the two models reduces the complexity and enables the developer to address the issues where they occur.[123]

“Like many patterns, you can view the CQRS pattern as a mechanism for shifting some of the complexity inherent in your domain into something that is well known, well understood, and that offers a standard approach to solving certain categories of problems.”[124]

3.2.3 Multiple Teams

The introduction of different development teams is not only possible but also useful when developing a system using the CQRS pattern. There are basically three areas of the system: The client, the Command Side and the Query Side. This fundamental separation simplifies the assignment of the developers to the team based on skills and experience.[125] The isolation of the teams decreases the communication and therefore reduces the required time.[126]

The skills required for the development of the client are the depending on the used technology. If multiple clients are planned further separation is possible. Due to the low complexity of the Query Models they are a good fit for unexperienced developers. Experienced developers which have extensive background knowledge about the domain should be assigned to the team of the Command side.

If a single team reaches a size that is too big it should be evaluated if the Bounded Context, which CQRS is applied to, is too big and can be divided in two or more distinct contexts.

3.2.4 Outsourcing

Since the complexity of the Query Models is low Greg Young sees them as a good candidate for outsourcing. The Events which the models are built from and the DTO’s that are the return value of the Query are very clear defined.[127] Two examples for the specification of Query Models are given in the appendices **A-6** and **A-7**.

The outsourcing may have several motivations. On the one hand it may be a goal to save resources. On the other hand it may be efficient to outsource complex Query Models where certain technologies, such as a graph database, may be useful to experts of the field.

3.2.5 Programming Language

The CQRS pattern is generally independent of a programing language. Since the components are independent, different programming languages may be used throughout the system. Due to

---

[123] Betts et al. 2012, p. 231
[124] Betts et al. 2012, p. 231
[125] Heimeshoff, Jander 2013, p. 4
[126] Young 2010, p. 55
[127] Young 2010, p. 55

the fact that the pattern originated from the C# environment the community is most active and the training materials as well as sample implementations are best elaborated. Since the pattern gained more attention the java community adapts the concepts.

Next to the two big object-oriented languages, there is also a tutorial for a implementation with the functional programming language F#.[128]

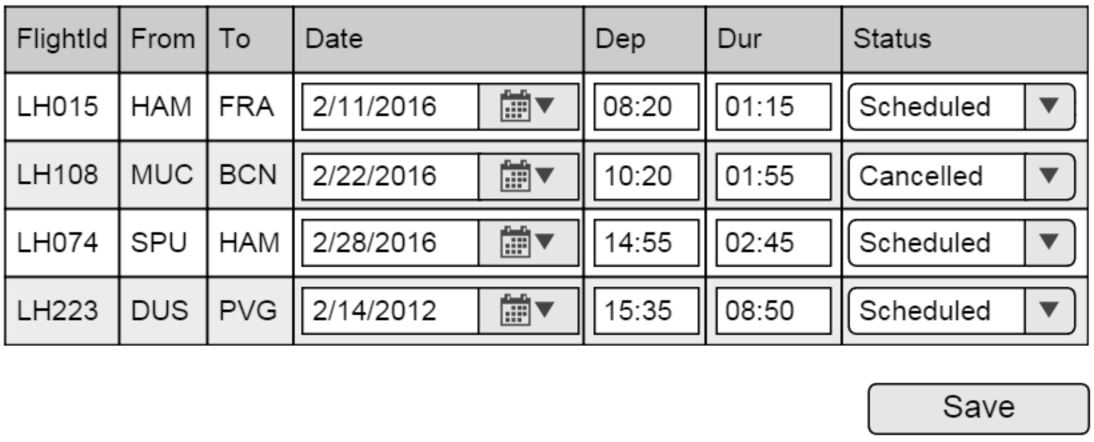
3.2.6 Tools & Frameworks

In the sample implementation the library cqrs-4-java of the author's colleague Michael Schnell was used. In the process of the thesis it was revised in order to comply with the latest definition of best practices. However, it still has some issues and only provides the base classes. Most urgent is that it does not support the push model for the Query Side. Therefore, the Events have to be queried from the Event Store regularly.

Alternatively, there is the open-source AxonFramework[129] for Java that is currently in active development by Trifork. It is licensed under the Apache 2 license and provides the basic building blocks for a CQRS application such as Aggregates, Repositories, as well as Event Stores and Event buses.[130]

3.2.7 Usability of Clients

In a CRUD application the client interacts with the back end by posting DTO's back and forth. This results in a very technical shaped user interface that tends to be very data-centric.[131]



FlightId	From	To	Date	Dep	Dur	Status
LH015	HAM	FRA	2/11/2016	08:20	01:15	Scheduled
LH108	MUC	BCN	2/22/2016	10:20	01:55	Cancelled
LH074	SPU	HAM	2/28/2016	14:55	02:45	Scheduled
LH223	DUS	PVG	2/14/2012	15:35	08:50	Scheduled

Figure 16: Example of a CRUD User Interface for editing flights

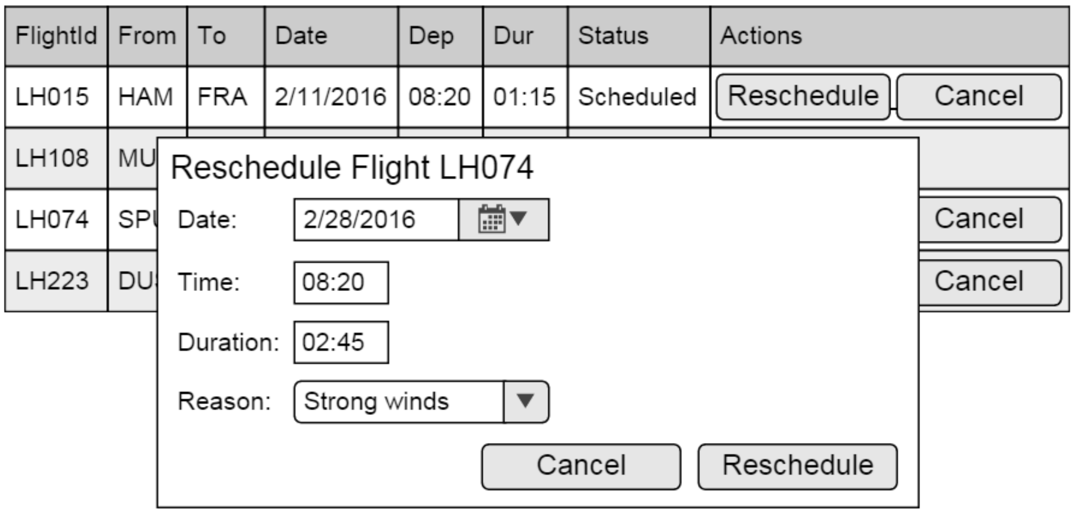
The goal of a task-based UI is to guide the user through the process in the application.[132] Instead of data-centric they are workflow oriented.

[128] See Gorodinski 2012

[129] See www.axonframework.org/

[130] Trifork 2016, § 1.1.2

[131] Betts et al. 2012, p. 232



FlightId	From	To	Date	Dep	Dur	Status	Actions
LH015	HAM	FRA	2/11/2016	08:20	01:15	Scheduled	Reschedule Cancel
LH108	MU						
LH074	SP						Cancel
LH223	DU						Cancel

Reschedule Flight LH074

Date:	2/28/2016
Time:	08:20
Duration:	02:45
Reason:	Strong winds

Figure 17: Example of a task-based UI for editing flights[133]

Instead if freely editing the Aggregates the user is confronted with prompts in a task-based UI.[134] These prompts are closely related to the Commands.[135] In our given example the task is started with the button “Reschedule” triggers a RescheduleFlightCommand with the entered departure time and duration. This new style is especially clear when complex processes with multiple steps need to be realized. Furthermore, it enables the designer to catch the intent of the user. It is not only about changing the state anymore but also about the reason why.

In most simple domains the effort of designing a task-based UI outweighs the benefit. However, in complex domains it may increase to productivity of the end users heavily.[136]

3.3 Testability

The testability describes the degree with which tests can be performed in order to determine whether test criteria are met. [137]

3.3.1 Unit Testing of Business Logic

The business logic is located in the Aggregates and they may be tested independently of other components. To test the business logic it can be simply tested if the Business Method has resulted in the expected Event.[138] The test is structured in three parts:

[132] Young 2010, p. 14
[133] Inspired by Figure 3 in Heimeshoff, Jander 2013, p. 2
[134] Heimeshoff, Jander 2013, p. 2
[135] Betts et al. 2012, p. 232
[136] Betts et al. 2012, p. 272
[137] ISO/IEC 25010, § 4.2.7.5
[138] Betts et al. 2012, p. 241

```java
public class FlightTest {

  @Test
  public void testMakeReservationSuccessWaitlist() throws DateInPastException,
      FlightCancelledException {
    // Setup
    Flight flight = new Flight();
    flight.schedule(FLIGHT_ID, DATE, ROUTE, CAPACITY);
    flight.makeReservation(PASSENGER_ID_1);
    flight.makeReservation(PASSENGER_ID_2);
    flight.markChangesAsCommitted();

    // Test
    flight.makeReservation(PASSENGER_ID_3);

    // Verify
    ReservationAccommodatedOnWaitlistEvent event =
      (ReservationAccommodatedOnWaitlistEvent)
      flight.getUncommittedChanges().get(0);

    assertThat(event.getEntityId()).isEqualTo(FLIGHT_ID);
    assertThat(event.getPassengerId()).isEqualTo(PASSENGER_ID_3);
  }

}
```

Code fragment 5: Unit Test of the Waitlist Business Logic

Firstly, the test is prepared. Therefore, the instance of the Aggregate is created and put in the desired state. In this example a flight is scheduled with a capacity of 2 passengers. Furthermore, two reservations are made on the flight. At the end of the preparation the `markChangesAsCommitted()` method of the Aggregate is called in order to empty the list of uncommitted changes.

In the second step the tested method is called. If the Aggregate raises an Event but does not implement an Event Handler, the test will fail here. In this example an additional reservation is made.

In the last step the result of the method is verified. Therefore, the last Event from the uncommitted changes is queried and casted to the expected Event type. If an Event with an Event type other than the expected (e.g. ReservationConfirmedEvent) has been published the test fails with a ClassCastException. Finally the properties of the Event are verified.

An example for testing of exception rising of an Aggregate may be found in the **appendix A-8**.

The Event Handlers are tested separately in order to avoid code repetition if multiple methods publish the same Events.

```
public class FlightEventHandlerTest{

  // static variables and setup omitted

  @Test
  public void testHandleRescheduleFlightEvent() {
    // Prepare
    LocalDateTime newDepartureTime = LocalDateTime.of(2021, 05, 05, 10, 00);
    Duration newDuration = Duration.ofHours(3);
    FlightRescheduledEvent event = new FlightRescheduledEvent(FLIGHT_ID,
        newDepartureTime, newDuration);

    // test
    flight.loadFromHistory(event);

    // verify
    assertThat(flight.getDepartureTime()).isEqualTo(newDepartureTime);
    assertThat(flight.getDuration()).isEqualTo(newDuration);
  }
}
```

Code fragment 6: Unit Test of the RescheduleFlightEventHandler

In conclusion, the testability of the business logic is given. It can be tested independently from all other components and follows a clear structure. If the business rules are modelled using BPMN the test cases may be derived from the process diagram. For each path in the diagram an own test case should be implemented. This way full test coverage of the business rules is accomplished.

3.3.2 Unit Testing of Query Logic

Similar to the business logic the query logic may be tested isolated from other components. The Query Event Handler of a Projection can be tested as shown in code fragment 7. The test follows the earlier described three step structure.

```
public class ConfirmedReservationCancelledEventHandlerTest {

  @Test
  public void testHandle() throws InvalidAttributeValueException {
    // Prepare
    QueryFlight testFlight = new QueryFlight(DummyData.FLIGHT_ID, new Route(
        "HAM", "FRA"), DummyData.DEPARTURE_TIME, DummyData.ARRIVAL_TIME,
        DummyData.CAPACITY.getValue());
    dao.insertFlight(testFlight);

    ConfirmedReservationCancelledEvent event
      = new ConfirmedReservationCancelledEvent(DummyData.FLIGHT_ID,
            DummyData.PASSENGER_ID_1);

    // Test
    handler.handle(event);

    // Verify
    QueryFlight flight = dao.findById(DummyData.FLIGHT_ID);
    assertThat(flight.getNumberOfAvailableSeats()).isEqualTo(3);
  }

}
```

Code fragment 7: Unit Test of the ConfirmedReservationCancelledEventHandler

3.3.3 Behavior Testing

The focus on behavior allows using techniques developed for Behavior-Driven-Development. Given-When-Then-Tests using the Gherkin syntax[139] are structured in three parts like the tests described earlier.

```
Scenario: Place passenger on waitlist if capacity is reached
      Given a 2 hour long flight from HAM to FRA has been scheduled on 2/22/2016
            at 15:20 with a capacity of 2 passengers
        And a reservation has been made on the flight
        And a reservation has been made on the flight
      When  a reservation for the passenger with the ID 4a14a1 has been made
      Then  the passenger with id 4a14a1 is on the waitlist of the flight
```

Code fragment 8: Given-When-Then-Test of the Waitlist Business Logic

Each statement represents a bunch of operations. The test is converted from the Gherkin syntax to code and can be executed. This enables domain experts without programming skills to write unit test for business rules.

[139] For an introduction to Given-When-Then-Tests refer to Cucumber 2014 and Fowler 2013. For deeper insights into Behavior Driven Development refer to Smart 2014

3.3.4 Smoke Testing

Furthermore, Greg Young points out opportunities for smoke tests. Smoke tests cover the main functionality of a system in order to verify the crucial functionalities without taking finer details in consideration.[140] He suggests running all or a subset of Commands that occurred in the systems and compare the result to the one of the previous version of the system. [141]

3.4 Operational Management

3.4.1 Modifiability

Changes on the Query Side may be made being confident that they do not have any impact on the business logic lying on the Command side. Therefore, Projections can be altered or extended without much risk at any time. If necessary it is rebuild from the beginning of the system.[142]

Compared to systems without CQRS the Domain Model on the Command Side is much simpler since it does not implement any additional read logic. The model concerns only with the core business logic. Therefore, the business logic is much easier to change.[143] This may involve changing Business Methods or the data structure of Aggregates. Furthermore, the message-based architecture makes it easy to modify complex workflows with different components involved.[144]

Since the Event types represent the behavior they change less often than data structures.[145] However, if it is necessary to change an Event type this causes problems. The Events of this type that occurred in the past are stored in the Event Store in the old version. There are three ways to handle this. The Aggregate may provide an individual Event Handler for each version of an Event type. This option should only be considered as a short-term solution since it pollutes the Aggregates with legacy Event Handlers. An alternative is to provide a component in the Event Store that translates the Events to the newest version. If the old Events are only queried infrequently because an Aggregate Cache is implemented this is a good solution. The last option is to translate all past Events in the Event Store to the current version. This approach is the cleanest but can be very expensive if the number of Events is very large.[146]

In conclusion, the distinct models enable the developers to react on changing business requirements without much risk on the Query Side and at a low cost on the Command Side.

---

[140] McKay 2015, p. 55
[141] Young 2014a, at 26:00
[142] Heimeshoff, Jander 2013, p. 2; Betts et al. 2012, p. 231
[143] Betts et al. 2012, p. 231
[144] Esposito 2015d, p. 13
[145] Young 2014a, at 10:25
[146] Betts et al. 2012, p. 128

3.4.2 Extensibility

Extensibility of the system in this context means supporting a new set of Queries or Commands, or adding functionality to existing Commands and Queries.

In order to support new queries, simply a new Projection is added. This is not affecting any business logic or the other queries. The Events to build the read model can be used out-of-the-box.

Existing queries may be enhanced by adding a query parameter or extending the Query Result. If the Query Model sufficient to serve the query the implementation is trivial. However, if changes to read model are required the Event Handlers have to be updated. Afterwards, the Query Model may be rebuilt using the historic Events.[147]

For new Commands the according Command Handler need to be implemented.[148] Due to their simple structure this is also trivial. If new business logic needs to be added this is done in the Aggregates. The effort largely depends on the quality of the Write Model.

3.4.3 Bug Tracking and Troubleshooting

Localizing a bug is much easier in CQRS systems. When a user issues a bug report the developers can look at the Events in the Event Store.[149] They captured everything that has happened to the model.[150]

Furthermore, the exact same situation can be staged in a test environment. The developer may debug the code step by step by replaying an Event at a time. The only information that is required is when the error occurred.[151]

3.4.4 Recoverability

Recoverability describes the degree of the system’s ability to recover data, lost due to an interruption or a failure, and to reestablish the desired state.[152]

In case of a system crash the Query Models on the Query Side may be simply rebuild using the Event log of the Event Store at any time.[153] The same applies to the Aggregate Cache. The only critical component is the Event Store. However, if the Commands are stored they may be reissued to rebuild the Event Store.[154]

[147] Heimeshoff, Jander 2013, p. 2; Betts et al. 2012, p. 231
[148] Betts et al. 2012, p. 231
[149] Young 2014a, at 20:35
[150] Vernon 2013, p. 162
[151] Betts et al. 2012, p. 241
[152] ISO/IEC 25010, § 4.2.5.4
[153] Heimeshoff, Jander 2013, p. 2
[154] Gamma et al. 1995, p. 266

3.4.5 Parallelization

The independence of the components allows running several versions of the software at the same time. Since they share the same Event Store Commands are effective and the queries are up to date. In order to support several versions running at the same time a Message Vus is necessary that allocates the Messages to the appropriate receiver. Running several versions of the system may have different motivations:

Firstly, it allows backwards compatibility of Commands and queries. The Message Bus decides using the specified version in the message which instance of the Command or Query Side processes it.

Secondly, it allows migrating to a new version of the system without downtime. The new version may be set up while the old version is still running. When the new version is fully available and tested the Message Bus is configured to direct all new incoming Messages to the new version.

Finally, it allows testing new or adapted features with exclusive user groups. Therefore, Messages of certain user groups are directed to another version of the system.

3.5 Entry Barriers

In this subchapter the entry barriers are examined.

3.5.1 Availability of trained Staff

The number of experienced developers is limited.[155] If a project is realized using the CQRS pattern the company will most likely have to train unexperienced developers and system architects.

3.5.2 Availability of Training Material and Workshops

In the recent years several training materials in different forms have been released. The most detailed and elaborated is the free guidance “Exploring CQRS and Event Sourcing”[156] by Microsoft's Pattern & Practices Group. Additionally, there is a Pluralsight course by Dino Esposito[157] and an online video class by Greg Young[158]. Greg Young also gives classes in person. As a general introduction to the topic there are numerous talks at conferences documented.[159]

---

[155] Vernon 2013, p. 540
[156] Betts et al. 2012
[157] Esposito 2015a
[158] Young 2013
[159] Young 2014a, 2014b, 2014c

However, there is no organization that has taken the responsibility to define best practices and provides various examples for different applications like Eric Evan's "Domain Driven Design Community"[160] does it for DDD.

3.6 Costs

There are different opinions about the cost benefit ratio of implementing CQRS. However, simple and static Bounded Contexts are less likely to warrant the up-front investment in detailed analysis, modeling, and complex implementation.[161]

The modifiability especially in domains with frequently changing business rules decreases the costs for development and maintenance in the long term.[162]

[160] See **www.dddcommunity.org/**
[161] Betts et al. 2012, p. 234
[162] Betts et al. 2012, p. 232

4 Characteristics of Systems that Benefit from CQRS

This chapter is an approach to give guidance to system architects on characteristics of good candidates for the CQRS pattern. The following characteristic of a system can point to an advantage that may be gained by using the CQRS pattern:

4.1 Competitive Advantage

The design of a system following the CQRS and Event Sourcing principles brings a high level of effort with it. It requires a deep understanding of the business domain and is therefore only worth for complex domain where competitive advantage can be derived from.[163]

4.2 Imbalance between the Number of Reads and Writes

In many business systems the number of reads outweighs the number of writes.[164] The segregation of the responsible components allows independent optimization. This may involve scaling up by duplicating the Query Side and distributing it geographically or using different database technologies.[165] Furthermore, the read performance can be vastly increased compared to normalized models.

4.3 Legal Requirement of an Audit Trail

The immutable Events provision the full history of state changes of the system. Therefore, they provide a detailed audit trail.[166] This enables the domain experts to track them and tell exactly why the system is in the current state. In some cases this may even be a legal requirement. [167]

4.4 Business value of state changes

Event Sourcing delivers an alternative approach to business intelligence. The history of state changes enables the developers to answer hypothetical questions.[168] The value of ES exceeds the value of simple database logging where the information about the reason of the change is lost. The business value of the information about change that is preserved in the Events can be highly valuable.[169] Even if the value of the information may be unknown at the time of

[163] Vernon 2013, p. 540
[164] Young 2014a, at 50:50
[165] Betts et al. 2012, p. 226; Young 2014a, at 53:40; Fowler 2011
[166] Betts et al. 2012, p. 241; Vernon 2013, p. 541
[167] Vernon 2013, p. 162
[168] Vernon 2013, p. 162
[169] Young 2010, p. 28

implementation it may be relevant at a later point in time. Reports evaluating the Events of the past can be developed.[170] Event Sourcing is a “good lossless, transactional model”.[171]

4.5 Highly collaborative Systems with stale Data

Collaboration means that multiple users or systems work on the same set of data. With the enhanced techniques for conflict detection and resolution described earlier the pattern can provide a significant advantage over the traditional CRUD approach.[172]

When multiple user work on the same set of data in a collaborative environment the data in the cache of the UI shown to a user may have been changed by another user. This means the data is stale. Therefore, when a user makes a decision the system has to check if it was made on outdated information. With CQRS it is possible to detect these conflicts and implement rules about which actions are conflicting and which action do not interfere.[173]

[170] Betts et al. 2012, p. 241; Young 2014a, at 17:00
[171] Young 2014a, at 42:30; Vernon 2013, p. 540
[172] Betts et al. 2012, p. 233
[173] Dahan 2009, p. 1

5 Conclusion

The study was set out to explore the advantages and disadvantages of the Command Query Responsibility Segregation pattern and has identified its effect on the system design, the development, the testability and the operational management as well as its entry barriers. Prior to the study no guidance providing such a wide analysis was available on the subject. Furthermore, the study sought to give a list of characteristics of systems that benefit most from the application of the CQRS and Event Sourcing principles.

The analysis has shown that systems built following the CQRS and Event Sourcing principles are highly scalable, distributable and increase the read performance. Furthermore, they hold a well-defined interface for other integrated systems to use. One of the key strengths is its techniques for conflict detection and resolving.

Additionally, it was shown that CQRS works well with process modeling tolls due to the fact that the behavior is modelled. The separation reduces complexity and suggests dividing the work among teams that have developers assigned based on their skills. The costs may be reduced by outsourcing query components.

The division of the system into two parts where the Command Side contains complex business logic to ensure consistency and the Query Side much simpler read operations allows optimization for individual requirements and makes the system more maintainable and flexible.[174]

The building blocks of Domain Driven Design and CQRS provide isolated testability. Furthermore, parallelization allows smoke tests in a productive-like environment.

In conclusion, CQRS should be applied to complex, collaborative Bounded Contexts with frequently changing business rules and where the business gains significant competitive advantage from.[175]

5.1 Critical Evaluation

During writing this process various limitations became apparent to the author. When analyzing the advantages and disadvantages the wealth of information varied strongly from criteria. While the most common advantages were widely discussed and multiple sources from authors with different background could be compared there are very little to no statements to criteria such as testability. There are especially few detailed records of weaknesses of and issues occurring while implementing CQRS.

[174] Betts et al. 2012, p. 266
[175] Betts et al. 2012, p. 232

Furthermore, there is not a unified perception of the pattern, since it is not fully defined yet. This is caused by the fact that the attention just recently emerged and the lack of an organization that has taken the responsibility to define best practices.

An assessment of more complex criteria such as security, scalability and distributability requires a more elaborated sample implementation than the one provided.

5.2 Further Research

With the many building blocks like Events, Command, Command Handlers and Query Handler of CQRS and Event Sourcing the software accumulates a high number of classes. A major part of them is pre-defined and reoccurring within the same type. Therefore, it could be evaluated how development could be optimized using Domain Specific Languages.

Furthermore, it could be studied if and how the process diagrams used for specification may be transformed into code of Events, Business Methods and workflows.

Publication Bibliography

Avram, Abel; Marinescu, Floyd (2006): Domain-Driven Design Quickly. [a summary of Eric Evans' Domain-Driven Design]. [S.l.]: C4Media (InfoQ : Enterprise software development series).

Bailis, Peter; Ghodsi, Ali (2013): Eventual Consistency Today. Limitations, Extensions, and Beyond. In *ACM Queue* 11 (3), p. 20. DOI: 10.1145/2460276.2462076.

Betts, Dominic; Domínguez, Julián; Melnik, Grigori; Simonazzi, Fernando; Mani, Subramanian (2012): Exploring CQRS and Event Sourcing. A journey into high scalability, availability, and maintainability with Windows Azure: Microsoft Developer Guidance. Available online at https://www.microsoft.com/en-us/download/details.aspx?id=34774, checked on 1/24/2016.

BPMN 2.0.2, 2013: Business Process Model and Notation. Available online at http://www.omg.org/spec/BPMN/2.0.2/, checked on 2/21/2016.

Chatterjee, Soumen (2004): Messaging Patterns in Service-Oriented Architecture, Part 1. In *MSDN Magazine* (4). Available online at https://msdn.microsoft.com/en-us/library/aa480027.aspx, checked on 1/23/2016.

Cucumber (2014): Given When Then. Available online at https://github.com/cucumber/cucumber/wiki/Given-When-Then, checked on 2/9/2016.

Dahan, Udi (2009): Clarified CQRS. Available online at http://udidahan.com/2009/12/09/clarified-cqrs/, checked on 1/23/2016.

Esposito, Dino (2015a): Modern Software Architecture: Domain Models, CQRS, and Event Sourcing: Pluralsight. Available online at https://www.pluralsight.com/courses/modern-software-architecture-domain-models-cqrs-Event-sourcing.

Esposito, Dino (2015b): Event Sourcing for the Common Application. Cutting Edge. In *MSDN Magazine* 30 (3), pp. 10–13. Available online at https://msdn.microsoft.com/en-us/magazine/mt267548.aspx, checked on 2/18/2016.

Esposito, Dino (2015c): CQRS for the Common Application. Cutting Edge. In *MSDN Magazine* 30 (6), pp. 10–14. Available online at https://msdn.microsoft.com/en-us/magazine/mt147237, checked on 1/19/2016.

Esposito, Dino (2015d): CQRS and Message-Based Applications. Cutting Edge. In *MSDN Magazine* 30 (7), pp. 10–13. Available online at https://msdn.microsoft.com/en-us/magazine/mt267548.aspx, checked on 1/19/2016.

Esposito, Dino (2015e): CQRS and Events: A Powerful Duo. Cutting Edge. In *MSDN Magazine* 30 (8), pp. 10–13. Available online at https://msdn.microsoft.com/en-us/magazine/mt267548.aspx, checked on 2/22/2016.

Evans, Eric (2004): Domain-driven design. Tackling complexity in the heart of software. Boston, Mass.: Addison-Wesley.

Evans, Eric (2011): Domain-Driven Design Reference. Definitions and Pattern Summaries. Domain Language, Inc. Available online at https://domainlanguage.com/ddd/patterns/DDD_Reference_2011-01-31.pdf, checked on 1/25/2015.

Fowler, Martin (2005): Event Sourcing. Capture all changes to an application state as a sequence of Events. ThoughtWorks. Available online at http://martinfowler.com/eaaDev/EventSourcing.html.

Fowler, Martin (2011): CQRS. ThoughtWorks. Available online at http://martinfowler.com/bliki/CQRS.html, checked on 2/19/2016.

Fowler, Martin (2013): GivenWhenThen. ThoughtWorks. Available online at http://martinfowler.com/bliki/GivenWhenThen.html, checked on 2/9/2016.

Gamma, Erich; Helm, Richard; Johnson, Ralph (1995): Design patterns. Elements of reusable object-oriented software. Reading, Mass., Wokingham: Addison-Wesley (Addison-Wesley professional computing series).

Gorodinski, Lev (2012): Domain-Driven Design, Event Sourcing and CQRS with F# and EventStore. Available online at https://www.youtube.com/watch?v=MHvr71T_LZw.

Hakim, Kamil (2012): Correctness for CQRS Systems. Elicitation and validation. Master Thesis. KTH Royal Institute of Technology, Stockholm, Sweden. School of Computer Science and Communication. Available online at https://www.nada.kth.se/utbildning/grukth/exjobb/rapportlistor/2012/rapporter12/hakim_kamil_12072.pdf, checked on 2/1/2016.

Heimeshoff, Marco; Jander, Philip (2013): CQRS – a new architecture precept based on segregation of Commands and queries. In *The H Developer*. Available online at http://www.h-online.com/developer/features/CQRS-an-architecture-precept-based-on-segregation-of-Commands-and-queries-1803276.html, checked on 1/23/2016.

Hendrickson, Scott A.; Dashofy, Eric M.; Taylor, Richard N. (2005): An (Architecture-centric) Approach for Tracing, Organizing, and Understanding Events in Event-based Software Architectures. Institute for Software Research, University of California, Irvine. Available online at http://www.ics.uci.edu/~shendric/publications/C3-Event-comprehension.pdf, checked on 1/23/2016.

Johnsson, Dan Bergh (2009): Power Use of Value Objects in DDD. San Francisco: QCon. Available online at http://www.infoq.com/presentations/Value-Objects-Dan-Bergh-Johnsson, checked on 2/2/2016.

Leach, P.; Mealling, M.; Salz, R. (2005): A Universally Unique IDentifier (UUID) URN Namespace: IETF. Available online at http://tools.ietf.org/pdf/rfc4122.pdf, checked on 1/24/2016.

Lufthansa Industry Solutions (2016): Company Description. Available online at https://lufthansa-industry-solutions.com/company.html, checked on 2/20/2016.

McKay, Judy (2015): Standard Glossary of Terms Used in Software Testing. Version 3.01. International Software Testing Qualifications Board. Available online at http://www.istqb.org/downloads/send/20-istqb-glossary/104-complete-glossary-including-an-introduction-references-trademarks-and-revision-history.html, checked on 2/9/2016.

Meyer, Bertrand (1997): Object-oriented software construction. 2nd ed. Upper Saddle River, N.J.: Prentice Hall PTR; London : Prentice-Hall International.

Schnell, Michael (2007): cqrs-4-java. Command Query Responsibility Segregation for Java. Available online at https://github.com/fuinorg/cqrs-4-java/, checked on 2/21/2016.

Smart, John Ferguson (2014): BDD in action. Behavior-driven development for the whole software lifecycle: Manning Publications Company.

ISO/IEC 25010, 2011: System and software quality models. Available online at https://www.iso.org/obp/ui/#!iso:std:35733:en, checked on 2/20/2016.

Trifork (2016): Axon Framework 2.4 Reference Guide. Available online at http://www.axonframework.org/docs/2.4/, checked on 2/20/2016.

Vernon, Vaughn (2013): Implementing domain-driven design. Harlow: Addison-Wesley.

Young, Greg (2010): CQRS Documents. CQRS Info. Available online at https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf, checked on 1/19/2016.

Young, Greg (2013): Greg Young's CQRS Class. Available online at http://subscriptions.viddler.com/GregYoung, checked on 2/22/2016.

Young, Greg (2014a): CQRS and Event Sourcing: Code on the Beach. Available online at https://www.youtube.com/watch?v=JHGkaShoyNs, checked on 1/19/2016.

Young, Greg (2014b): Event Sourcing: GOTO Conferences. Available online at https://www.youtube.com/watch?v=8JKjvY4etTY, checked on 1/20/2016.

Young, Greg (2014c): Querying Event Streams. London: React. Available online at https://www.youtube.com/watch?v=DWhQggR13u8, checked on 1/21/2016.

Icon Attribution

The icons used in the figures were created by freepik[176] and are published under to Creative Commons BY 3.0 license[177].

[176] See www.freepik.com
[177] See www.creativecommons.org/licenses/by/3.0/

Appendices

A-1 Accompanying Code Examples

The code examples are available on GitHub:

https://github.com/martineckardt/reservation-mgmt.git

System Requirements

The software has been developed using Eclipse Luna Service Release 2. The project requires maven to load its dependencies and java 8.

A-2 Ubiquitous Language of the Reservation Management BC

Artifacts

Flight	A flight meaning a single journey from one airport to another on a specific date. The flight may be rescheduled or canceled. A flight can take as many reservations as its capacity allows. All additional reservations are accommodated on a waitlist.
Reservation	A passenger booked on a specific flight.
Capacity	The maximum number of passengers that can be accommodated on the flight. The value has to be between 1 and 300.
Confirmed reservation	A reservation that has been confirmed and will have a guaranteed seat on the flight.
Waitlist	All reservations that have been issued after the maximum capacity of the flight has been reached. They may be confirmed if a confirmed reservation is cancelled or a passenger does not show up at the boarding time at the gate.
Passenger	The person who is flying including its contact details and if he is vegetarian.
…	

Commands

ScheduleFlightCommand	Schedule a flight at the given date on the given route with the capacity
RescheduleFlightCommand	Change the date of the flight
CancelFlightCommand	Cancel the entire flight. This should only happen in extreme circumstances
MakeReservationCommand	Make a reservation for a flight with a registered passenger
CancelReservationCommand	Cancel a reservation. The reservation may be confirmed or in the waitlist
…	

Events

FlightScheduledEvent	A flight with a date, a route and a capacityhas been scheduled
FlightRescheduledEvent	
FlightCancelledEvent	
ReservationConfirmed	A reservation for a flight has been confirmed
…	

A-3 Product Features of the Reservation Management BC

Flight Scheduling

- Schedule a flight with a departure time, a duration, a route and a capacity
- Origin and destination may not be equal
- Date has to be in the future

Flight Rescheduling

- Date has to be in the future

Flight Cancellation

- A flight may be cancelled

Registration of a Passenger

- A passenger may be registered with his personal information and the information about his preferred language as well as if he is a vegetarian.

Reservation

- One reservation per passenger, otherwise fail
- If a reservation is confirmed, the payment is requested
- If capacity is reached accommodate on a waitlist

Reservation Cancellation

- Fail, if the passenger does not have a reservation
- If passengers are on waitlist confirm one passenger per cancellation

A-4 Simplified Repository Interface of ddd4j

```java
package org.fuin.ddd4j.ddd;

/** Repository that supports CRUD operations for an Aggregate.
 *
 * @param <ID> Type of the Aggregate root identifier.
 * @param <T> Type of the Aggregate.
 */
public interface Repository<ID extends AggregateRootId,
    T extends AggregateRoot<ID>> {

    /** Factory method to create a new Aggregate. */
    public T create();

    /** Reads the latest version of an Aggregate. */
    public T read(ID id);

    /** Reads a given version of an Aggregate. */
    public T read(ID id, int version);

    /**      * Saves the changes on an Aggregate in the Repository. */
    public void update(T Aggregate, MetaData metaData);

    /** Deletes an Aggregate from the Repository. */
    public void delete(ID AggregateId);

}
```

A-5 Overview over the most important CQRS Terms

Command Side

Command Handler *Also: Command Processor*	Interface to the client. Receives Commands and invokes the according Business Methods of Aggregates.
Write model *Also: Command Model*	Normalized model of the Command Side containing the Aggregates. Ensures consistency.
Repository	Provides the CRUD operations to create, read, update and delete Aggregate instances by their ID.
Aggregate Cache	Stores snapshots of the Aggregate instances
Event Store	Stores the Events
Business Methods*	Contain business logic. If a state change occurs it is modelled as an Event. The state change is not applied by the Business Method but by invoking to according Event handler.
Aggregate Event Handler *Also: Event Handler*	Apply the state change represented by an Event to the Aggregate instance

Query Side

Projection	Subscribes to Events and invokes Event Handlers. For each set of related queries a Projection exists.
Query Models *Also: Read models*	Models optimized for the views of the data in the client.
Query Event Handler*	Retrieves instances of the Query Model from the Query Store and manipulates them using the properties of the Event.
Query Store*	Provides CRUD operations to create, read, update and delete instances of the Query Model. Similar to the Repository on the Command Side.
Query Result*	DTO's designed to fit the requirements of the client. May hide

	information of the Query Model.
Query Handler* *Also: Query Processor*	Interface to the client. Receives the Queries and builds the response.

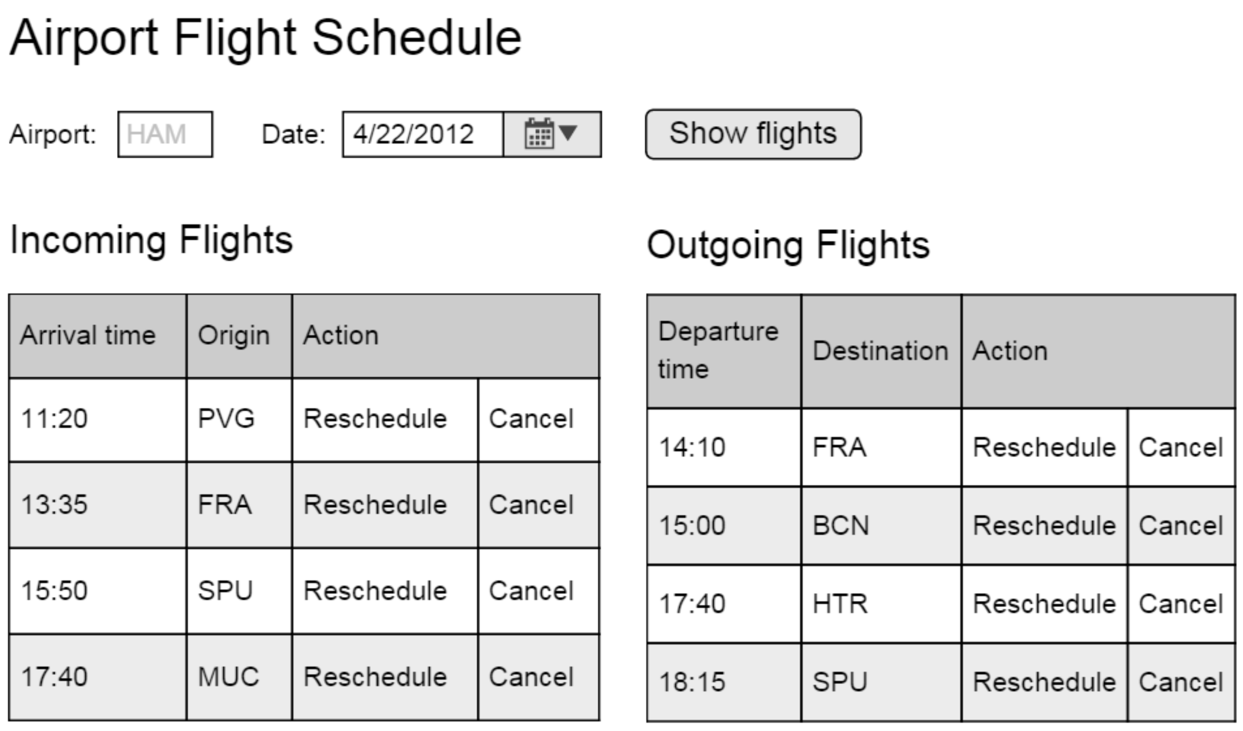
* Term introduced by the author

A-6 Specification of the Airport Flight Schedule Projection

Description

Shows the incoming and outgoing flights for an airport on a specific date. This view is intended for the operator of the airport.

Mockup of the client



Airport Flight Schedule

Airport: HAM Date: 4/22/2012 Show flights

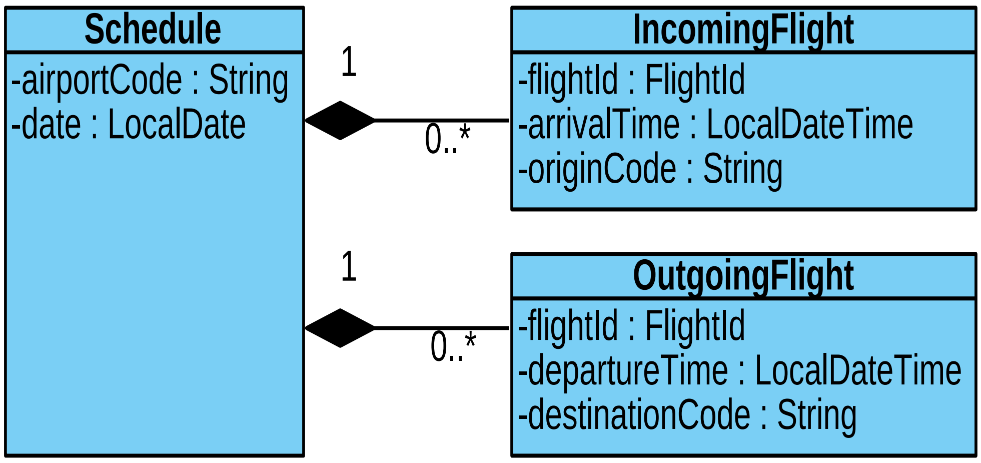
Incoming Flights

Arrival time	Origin	Action	
11:20	PVG	Reschedule	Cancel
13:35	FRA	Reschedule	Cancel
15:50	SPU	Reschedule	Cancel
17:40	MUC	Reschedule	Cancel

Outgoing Flights

Departure time	Destination	Action	
14:10	FRA	Reschedule	Cancel
15:00	BCN	Reschedule	Cancel
17:40	HTR	Reschedule	Cancel
18:15	SPU	Reschedule	Cancel

Query Model / Query Result:



Subscribed Events and behavior

FlightScheduledEvent	Add flight to outgoing flights and incoming flights of the aiport schedules. Be aware that the arrival may be on the next day, if the flight is over night.
FlightCancelled	Delete the flight from all schedules
FlightRescheduledEvent	Update the times. If the dates changed delete from the schedule and add to the schedule of the new date.

Query parameters

airportCode	The code of the airport
date	The date of the desired schedule

Access Restrictions

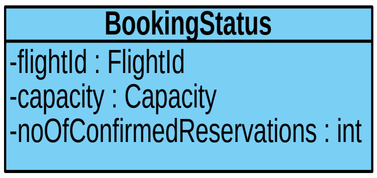
User is required to have role ‘airport operator’.

A-7 Specification of the Booking Status Projection

Description

Shows the booking status of flights including the number of confirmed reservations and the capacity.

Query Model / Query Result:



Subscribed Events and behavior

FlightScheduledEvent	Create an instance of BookingStatus for the flight with the capacity
FlightCancelled	Delete instance of BookingStatus
ReservationConfirmedEvent, ReservationOnWaitlistConfirmed	Add 1 to AmountOfConfirmedReservations
ConfirmedReservationCancelledEvent	Substract 1 to AmountOfConfirmedReservations

Query parameters

flightId (optional)	return BookingStatus for the given flight. If not set, return for all flights

Exceptions

BookingStatusDoesNotExistException	thrown if no booking status for the given flightId exists

A-8 Unit Testing of Exceptions of an Aggregate

```
package org.lhind.reservation_mgmt_example.command.flight;

public class FlightTest {

  private static final FlightId FLIGHT_ID = FlightId
      .valueOf("58e677da-ccbb-47a2-92e3-237af92201d7");
  private static final Route ROUTE = new Route("HAM", "FRA");
  private static final Date DATE = new Date(4106072638000L);
  private static final Capacity CAPACITY = new Capacity(2);

  private static final PassengerId PASSENGER_ID_1 = PassengerId
      .valueOf("f4ed5afc-a88b-4e26-a130-73150a6fa439");

  @Test(expected = FlightCancelledException.class)
  public void testMakeReservationFailsWithFlightCancelledException()
      throws DateInPastException, FlightCancelledException {

    // Setup
    Flight flight = new Flight();
    flight.schedule(FLIGHT_ID, DATE, ROUTE, CAPACITY);
    flight.cancel();
    flight.markChangesAsCommitted();

    // Test
    flight.makeReservation(PASSENGER_ID_1);
  }
}
```

This test case tests if an exception is thrown when a reservation is made on a cancelled flight. Please note that the property `expected` of the test annotation is a feature that requires jUnit 4+.

Glossary

CAP-Theorem

According to the CAP theorem it is impossible for a distributed computer system to provide the following three guarantees simultaneously:

- Consistency (C). All the nodes in the system see the same data at the same time.
- Availability (A). The system can continue to operate even if a node is unavailable.
- Partition tolerance (P). The system continues to operate despite the nodes being unable to communicate.[178]

Universally Unique Identifier (UUID)

In order to identify objects we may allocated IDs to them. IDs need to be unique in their namespace. UUIDs are 128-bit values.[179] Commonly there are displayed in their hexadecimal representation separated by hyphens.

`c76b7bf5-c3b7-4060-85a7-0b6d1a99b4fe`

`Version 4 UUID`

Since the number of unique UUIDs is so high the chance to generate the same UUID twice is one in 17 billion. The main motivation to use UUIDs is that the IDs may be generated by the clients without consulting a centralized authority.[180]

[178] Betts et al. 2012, p. 261
[179] Leach et al. 2005, p. 2
[180] Leach et al. 2005, p. 3