Alex Lawrence

Implementing DDD, CQRS and Event Sourcing

Implementing DDD, CQRS and Event Sourcing

Alex Lawrence

This book is for sale at http://leanpub.com/implementing-ddd-cqrs-and-event-sourcing

This version was published on 2021-04-15

Tweet This Book!

Please help Alex Lawrence by spreading the word about this book on Twitter!

The suggested hashtag for this book is #ddd-cqrs-es.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#ddd-cqrs-es

This book is dedicated to my wonderful wife and kids. I love you.

Contents

Preface

This book explains and illustrates how to implement Domain-Driven Design, Command Query Responsibility Segregation and Event Sourcing. The goal is to build software that is behavior-rich, event-based, problem-centric, reactive, scalable and well-designed. **Domain-Driven Design** is a concept that focuses on the problem space and its associated knowledge areas. **Command Query Responsibility Segregation** separates a software into a write side and a read side. **Event Sourcing** is an architectural pattern that represents state as a sequence of immutable events. The concepts are explained in theory and put into practice with standalone examples and a Sample Application. This is done without third-party technologies. The book comes with a source code bundle and supports interactive execution. All code is written in **JavaScript** and uses **Node.js** as backend runtime.

Version & Feedback

Book version: **1.3.3**

If you have any questions, suggestions or problems to report, please write an e-mail or visit the book's Leanpub Forum. The contact e-mail address is mail@alex-lawrence.com and the Leanpub forum can be found here. All feedback is highly appreciated.

Style of this book

The primary focus of this book is the application and the implementation of concepts. Therefore, the purely theoretical parts are generally concise. The covered topics are illustrated extensively with a large amount of examples and code. Selected conceptual parts are also discussed in greater detail. **Apart from Node.js and JavaScript, the book's main content does not utilize or explain specific frameworks or technologies. For functionalities that require persistence or inter-process communication, exemplary implementations are provided that directly work with the filesystem**. This includes Repositories, the Event Store, Read Model stores and a remote event distribution. The goal is to convey a deeper understanding of the according concepts. For production purposes, these implementations can be replaced with suitable technologies. This procedure is exemplified in Appendix B.

Why Node.js and JavaScript?

There are multiple reasons for using Node.js as runtime platform and JavaScript as programming language. One is my personal long-term experience with them. Also, two projects in which I applied CQRS and Event Sourcing made use of both technologies. Another reason for JavaScript is that it is a very widespread language. Even more, its syntax can look similar to other popular languages, such as Java, especially when using classes. Furthermore, this book uses JavaScript for the frontend and shares some code with the backend. One specific reason for Node.js is its simplicity for certain use cases, such as when operating an HTTP server. Finally, JavaScript helps to keep the code examples concise through specific language features such as arrow functions or destructuring assignments.

Programming paradigms

The content and the code examples in this book compromise a combination of imperative, declarative, object-oriented and functional programming. However, the majority of the implementations apply the object-oriented paradigm. Also, classes are used extensively as well as private fields and private methods for strong encapsulation. Still, over the course of the book, the domain-related implementations transition towards a more functional style. Also, certain infrastructural functionalities apply selected principles of it wherever useful. As the changes come naturally together with introducing specific concepts, there is no need for upfront knowledge in Functional Programming.

Type checking and validation

The source code provided with this book only makes selective use of type checking and data validation. Since JavaScript is a dynamically typed language, type checks can only happen at runtime. Also, the available technical possibilities are rather limited. In contrast, data validation itself is not fundamentally different from most other languages. However, due to the lack of static types, it can be more cumbersome. This book uses both mechanisms only for specific examples and for Sample Application parts where data integrity is crucial. This means, verification checks are primarily implemented for domain-related components, for Domain Events as well as for Commands and Queries. Note that this approach does not imply any opinion about the general importance of type checking and data validation.

Required technical knowledge

Working through the book requires advanced knowledge in JavaScript and in Node.js. Many of the provided implementations use language features introduced with more re-

cent ECMAScript standards. The following lists show which newer language constructs, which JavaScript APIs and which Node.js APIs are assumed to be known. Every entry is accompanied by a link, pointing to either a documentation page or a tutorial. Generally, it is not necessary to read through all of them. Many functionalities can be understood from the context they are used in. The only exceptions are **Promises** and **async/await**, which provide an alternative to callbacks for asynchronous programming. As those concepts heavily influence the way of writing code, getting familiar with them should be done upfront.

Newer JavaScript language constructs:

- Arrow functions
- Async functions
- Await operator
- Classes, specifically private class fields
- const, let
- Destructuring assignment
- JavaScript Modules
- Shorthand property names
- Spread Syntax
- Template literals
- Throwing and handling Errors

Used JavaScript APIs:

- Array functions such as `map()`, `reduce()`, `forEach()`, `includes()`, `filter()` and `flat()`
- Console
- JSON
- Map
- Object functions such as `defineProperty()`, `defineProperties()`, `assign()` and `freeze()`
- Promises
- Proxy
- URLSearchParams

Used Node.js APIs:

- crypto.createHash()
- fs.watch

- many of the standard operations of fs.promises
- http functions such as `createServer()`, `get()`, `post()`
- various path functions
- Readable Streams and Transform Streams
- url.parse()
- child_process.spawn()

Executing the code

This book provides an easy possibility to execute most of the shown code examples when reading on a computer. The included bundle contains a Code Playground than can execute Node.js code and display its output. Almost every implementation in the book is accompanied by a hyperlink labeled as "run code" or "run code usage". Clicking such a link opens a web page with an editor and an output window. Please note that **this utility executes Node.js on your local machine**. Therefore, you are responsible for verifying the safety of each program. The only requirement is to have a recent version of Node.js installed. Running the playground is done by executing `npm start` inside the bundle root directory. The software starts an HTTP server on port 8080.

.data directory

Starting from Chapter 9, many code examples and Sample Application implementations save data to the filesystem. Every created file and subdirectory is put within a directory called ".data", which itself is placed at the bundle root. This data container can be deleted at any time without influencing the functionality of associated code.

The complete source code that is bundled with the book is licensed under the **MIT License**. This means, you are free to re-use every contained functionality for your own projects. However, please refrain from publishing the complete bundle, as it represents an essential part of the work for this book. Also, note that the code is primarily for illustration purposes. While the implementations fulfill the respective functional requirements, they are not necessarily ready to be used in production environments. Amongst other things, the code may miss essential validations and can be vulnerable to security-related attacks.

About the author

I am a software developer with knowledge and experience in architecture, automation, backend, frontend, operations, teaching, technical leadership and testing. Since 2007, my professional focus lies on full-stack web development. In most projects, I use JavaScript as language and Node.js as backend runtime. Wherever useful, I apply selected parts of DDD. The architectural patterns I am most interested in are Event-driven Architecture, CQRS and Event Sourcing. For the frontend, I personally favor to use native technologies, such as Web Components. Professionally, I also work with various libraries and tools. Most recently, I started learning Rust as new programming language and picked up selected concepts of Functional Programming. Since many years, I am a strong supporter for Free/Libre Open Source Software.

Ever since the age of 10, I have been interested in computers. Experimenting with BASIC on a C64, programming with Turbo Pascal and failing to self-teach C++ accompanied my early adolescence. Only much later when studying Informatics, I deep-dived into the theory and learned many other programming languages. My first job introduced me to advanced concepts and patterns, including CQRS and Event Sourcing. At some point, I quit my job and became a freelancer. Simultaneously, I launched a mouse tracking product that naturally applied Event Sourcing. Also, I joined a startup developing a collaborative software with strong focus on DDD, CQRS and Event Sourcing in Node.js. In this project, I had the chance to deepen my theoretical knowledge and apply many concepts practically.

Formatting and highlighting

The book uses certain formatting and highlighting to either categorize or emphasize information. The names of acknowledged concepts are always capitalized, such as Entity or Domain Model. Words written in **bold** either indicate that a term is defined and/or explained, or simply emphasize an important text passage. References to code constructs such as classes, functions, keywords or variables are displayed in `monospaced` font. Complementary content is placed into separately positioned paragraphs, visually highlighted with an icon. Typically, this is an “i” enclosed in a circle. Wherever it makes sense, lists and tables are used to display information. Apart from plain text, this book includes simple drawings and numerous code examples. The displayed code is not always complete. Rather, it focuses on illustrating specific facts.

Complementary content

This is an example paragraph of complementary information. It is not required to be read in order to understand the main content.

Chapter summary

The chapter order in this book is determined by what makes most sense with regard to building the Sample Application. As a consequence, the topics are laid out in a way they can build upon each other. Amongst other things, this also reduces the likelihood of referring to terms before they are explained. Every chapter follows the same structure. The main part explains the respective concepts and illustrates them either with drawings or code examples. At the end of each chapter, the discussed concepts are applied to the Sample Application. This is done by describing the working steps to take and by showing the according drawings and code. An exception to this format is Chapter 5, which does not contain a Sample Application section.

Chapter 1

Chapter 1 introduces the concept **Domains** and defines it in the context of DDD as problem-specific knowledge fields. The explanation incorporates the importance of always focusing on the original problems to solve. Different categories of knowledge areas and their significance are compared to each other. This is followed by the definition of **Domain Experts**, which represent the primary source of relevant information in each project. For an adequate subdivision of knowledge areas, the concept **Subdomains** is introduced. This is accompanied by a description of the different types **Core Domain**, **Supporting Subdomain** and **Generic Subdomain**. A brief illustration on how to identify Subdomains provides practical guidance. At the end of the chapter, the Domains and the Subdomains for the Sample Application are identified.

Chapter 2

Chapter 2 covers the concept **Domain Models**, which are sets of knowledge abstractions focused on solving specific problems. The chapter starts with describing their typical structure and components, and explains why such abstractions should incorporate verbs, adjectives and adverbs. Also, the relation to Domain Expert knowledge is clarified. As next

step, the concept **Ubiquitous Language** is introduced, which promotes a linguistic system to unify communication and eliminate translation. Afterwards, different representation possibilities for Domain Models are described and compared to each other, such as drawings and code experiments. This is followed by a section on **Domain Modeling**, which is the process of creating structured knowledge abstractions. As last step, the Sample Application Domain Model is created and expressed in multiple different ways.

Chapter 3

Chapter 3 focuses on the concept **Bounded Contexts**, which represent conceptional boundaries for the applicability of Domain Models. First, the concept is differentiated from Domains and Domain Models. Then, multiple context sizes and their implications are compared to each other. This includes large unified interpretations and smaller boundaries that align with Subdomains. Afterwards, the relation between a Bounded Context and a Ubiquitous Language is clarified. Also, it is explained why contexts are first and foremost of conceptual nature, but still commonly align with technological structures. The relationship and integration patterns **Open Host Service**, **Anti-Corruption Layer** and **Customer-Supplier** are discussed briefly. This is followed by the concept **Context Map** for visualizing conceptional boundaries. Finally, the chapter illustrates the definition of Bounded Contexts for the Sample Application.

Chapter 4

Chapter 4 deals with **Software Architecture**, which is a high-level structural definition of a software and its parts. Since this topic is very broad, the focus is narrowed down to common architectural patterns that fit well with DDD. The chapter starts with describing the typical parts of a software, consisting of **Domain**, **Infrastructure**, **Application** and **User Interface**. This is followed by describing and comparing the patterns **Layered Architecture** and **Onion Architecture**, which share many fundamental principles. Afterwards, it is explained and justified what approach the book and its implementations follow. Also, the section includes an explanation on how to invert dependencies between software layers by using abstractions. The chapter ends with defining the Software Architecture and the resulting directory layout for the Sample Application.

Chapter 5

Chapter 5 describes and illustrates selected concepts for a high code quality with focus on Domain Model implementations. First, it explains the importance of establishing a

binding between the Domain Model and its implementation, together with other artifacts. Afterwards, refactoring is described as functional refinement in the context of DDD and is differentiated from pure technical improving. Next, code readability is discussed and its characteristics are broken down into quantifiable aspects. The importance of combining related state and behavior is emphasized and exemplified by creating meaningful units with high cohesion. This is followed by an explanation on how to deal with dependencies and how to apply **Dependency Inversion**. The final concept of the chapter is **Command-Query-Separation**, which promotes a separation of state modifications and computations.

Chapter 6

Chapter 6 provides a number of tactical patterns as essential building blocks for a Domain Model implementation. First, **Value Objects** are introduced as a way to design a descriptive part of a Domain without a conceptual identity. This is accompanied by an illustration of the key characteristics Conceptual Whole, Equality by Values and Immutability. Afterwards, **Entities** are described as possibility to design unique, distinguishable and changeable Domain Model components. This includes an explanation of identities and how to generate reliable identifiers using **UUID**. Next, **Domain Services** are discussed for expressing stateless computations and encapsulating external dependencies through abstractions. Afterwards, **Invariants** are presented as a way to transactionally ensure consistent state. Finally, all patterns are applied to create a useful first Sample Application Domain Model implementation.

Chapter 7

Chapter 7 presents the concept of **Domain Events**, which represent structured knowledge of specific meaningful occurrences in a Domain. The chapter starts with explaining their relation to Event-Driven Architecture. This is followed by describing common naming conventions for creating expressive event types based on a Ubiquitous Language. Afterwards, the standard structure and content of events are discussed, including a differentiation between specialized values and generic information. Next, it is described how the distribution and processing of Domain Events work. This includes the implementation of an in-memory **Message Bus** and an **Event Bus**. Also, important challenges of a distribution process are discussed briefly. At the end of the chapter, Domain Event notifications are used to integrate different context implementations of the Sample Application.

Chapter 8

Chapter 8 introduces **Aggregates** to establish transactional consistency boundaries, which are essential for concurrency and persistence. The first section contains an explanation of **transactions** and a breakdown of their characteristics Atomicity, Consistency, Isolation and Durability. This is followed by discussing the structural possibilities of Aggregates, the **Aggregate Root** and the management of individual component access. Afterwards, the principle of **Concurrency** is explained and illustrated in detail, together with Concurrency Control. Then, the most important Aggregate design considerations are described, which focus on component associations, invariants and optimal size. **Eventual Consistency** is introduced as mechanism to synchronize distributed information across consistency boundaries in a non-transactional fashion. Finally, the Sample Application is analyzed and refactored in order to achieve a useful Aggregate design.

Chapter 9

Chapter 9 describes the concept of **Repositories** for enabling persistence in a meaningful way to a Domain Model. First, the chapter underlines the emphasis on the Domain Model as opposed to technological aspects. Then, a basic Repository functionality is illustrated that consists of saving, loading and custom querying. Also, the influences of persistence support on a Domain Model expression are exemplified. The next part explains **Optimistic Concurrency** and promotes version numbers as implementation. Furthermore, it describes automatic retries and conflict resolution for concurrency conflicts. Afterwards, the chapter focuses on the interaction with Domain Events and provides different publishing approaches. One promotes extending Repositories and the other one recommends separating concerns. Lastly, the Sample Application is refactored for persistence support together with reliable Domain Event publishing.

Chapter 10

Chapter 10 illustrates **Application Services**, which are the responsible part for executing use cases, managing transactions and handling security. The first section describes and compares two approaches for their overall design and implementation. This is followed by an explanation of different possible service scenarios, ranging from simple Entity modifications to specialized stateless computations. Afterwards, the topics transactions and consistency are explained with regard to Application Services and complemented with a detailed example. The subsequent section discusses cross-cutting concerns and utilizes the design patterns **Decorator** and **Middleware** for generic solution approaches. Next, the security-related

concepts **Authentication** and **Authorization** are explained briefly and illustrated with individual examples. The chapter ends with implementing the Application Services for the Sample Application together with exemplary authentication and authorization.

Chapter 11

Chapter 11 explains **Command Query Responsibility Segregation**, which separates a software into a write side and a read side. The chapter starts with an architectural overview to explain the high-level picture as well as the interactions between individual parts. The following section introduces and illustrates the two Domain Layer concepts **Write Model** and **Read Model**. This is followed by a comparison of different approaches for the synchronization of Read Model data. As next step, the message types **Commands** and **Queries** are explained together with recommendations on their naming and structure. Afterwards, **Command Handlers** and **Query Handlers** are presented as a specialized form of Application Services. Finally, all covered concepts are combined and applied in order to create a Sample Application implementation with CQRS.

Chapter 12

Chapter 12 covers the pattern **Event Sourcing**, where state is represented as a sequence of immutable change events. As first part, the chapter provides an architectural overview and describes the overall flow. This is complemented with a clarification on the relation to Domain Events and recommendations on event anatomy. Afterwards, event-sourced Write Models are explained and two different implementation approaches are compared. Then, the concept **Event Store** is introduced for the persistence of event-sourced state. This is followed by a section on **Read Model projections**, which are responsible for computing derived read data. Next, the concept of Event Sourcing is differentiated from Event-Driven Architecture and their combination is explained. The chapter ends with transforming the existing Sample Application code into an implementation with Event Sourcing.

Chapter 13

Chapter 13 describes how to split a software into separate executable programs that operate as autonomous and self-contained runtime units. The chapter starts with an explanation of program layout possibilities. This includes a division of architectural layers as well as the alignment with context implementations. Next, the two main types of context-level communication are described and compared to each other. This is followed by a section on **remote**

use case execution, which includes an HTTP interface factory as example implementation. Afterwards, the idea of **remote event distribution** is discussed and complemented with a filesystem-based implementation. As last step, the chapter separates the Sample Application implementation into multiple programs. This incorporates the introduction of a proxy server that provides a unified HTTP interface to multiple endpoints.

Chapter 14

Chapter 14 focuses on the **User Interface** part in the context of web-based software. As first step, the chapter explains how to serve files to browsers and introduces an HTTP file server. Then, the concept of a **Task-based UI** is introduced and compared to a CRUD-based approach. This is followed by explaining and illustrating the **Optimistic UI** pattern, which can improve user experience through optimistic success indication. Afterwards, the concept of **Reactive Read Models** is described, which helps to build reactive User Interfaces. This includes the introduction of the **Server-Sent Events** web standard. Also, the notion of components and their composition is explained together with the **Custom Elements** technology. The chapter ends with illustrating the implementation of the User Interface for the Sample Application.

Appendix A

Appendix A explains the use of static types and their potential benefits with TypeScript as example. The first part focuses on Value Objects, Entities and Services. This is followed by describing how static types affect the definition and the usage of events. Next, the implications on event-sourced Write Models are discussed. Afterwards, the concept Dependency Injection is revisited. Then, the effects of static types on Commands and Queries are explained. The appendix ends with summarizing the most important aspects of a TypeScript implementation for the Sample Application.

Appendix B

Appendix B deals with using existing technologies for production scenarios. The first part discusses identifier generation. This is followed by a section on **Containerization** and **Orchestration** using Docker and Docker Compose. Next, the Event Store is discussed with the example of EventStoreDB. Then, Read Model stores are covered with Redis as exemplary technology. Afterwards, the Message Bus functionality is exemplified with RabbitMQ. Then, a static file server and a proxy functionality are illustrated with NGINX. Finally, the introduced technologies are used for the Sample Application implementation.

Chapter 1: Domains

Generally speaking, a **Domain** is an area of expertise. Put into more abstract terms, it is a conceptual compound of cohesive knowledge. In the context of Domain-Driven Design (DDD), the term stands for the topical area in which a software operates in. [Vernon, p. 43] describes it as "what an organization does and the world it does it in". As alternative, [Evans, p. 2] defines it as "the subject area to which the user applies a program". Another way to think of it, is as the knowledge space around the problems a software is designed to solve. The exact definition of the term can vary slightly, depending on its context. Regardless of such detailed differences, it is always important to identify the distinct knowledge areas a software operates in.

Technological and business domains

Independent of individual experience and progression, the profession of a software developer demands to acquire and apply technological knowledge. This may include topics such as Boolean Algebra, Control Structures, Object-oriented Programming, Processes and Threading, Networking and many more. While these concepts are typically illustrated with concrete examples, they themselves are purely technical and applicable to different situations. Therefore, they can be gathered in a common Domain called "Informatics" (or "Computer Science"). When attempting to solve a non-technological problem with software, the crucial knowledge does not originate from this particular Domain. Rather, it is to be found in the knowledge area that is specific to the original problem itself.

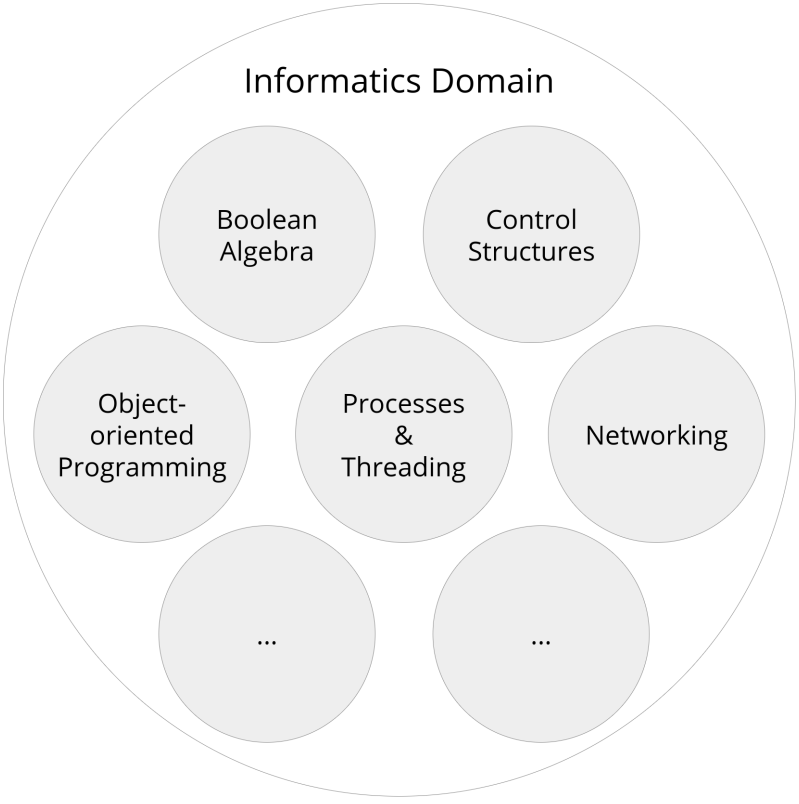

Figure 1.1: Informatics Domain

Business Domain problems cannot be solved adequately with solutions that exclusively belong to technological Domains. One exception is when the business itself is of technological nature, such as when developing a code analysis tool. Without applying the specialized knowledge around a problem, it is virtually impossible to build an adequate software solution. Therefore, it is necessary for involved participants and entities to understand the business Domains they are operating in. This applies to every project, regardless of its size and complexity. There are times when developers get involved in purely technical areas, such as when implementing communication protocols or configuring infrastructure. These fields of knowledge must not be confused with the actual Domain of a company or a product.

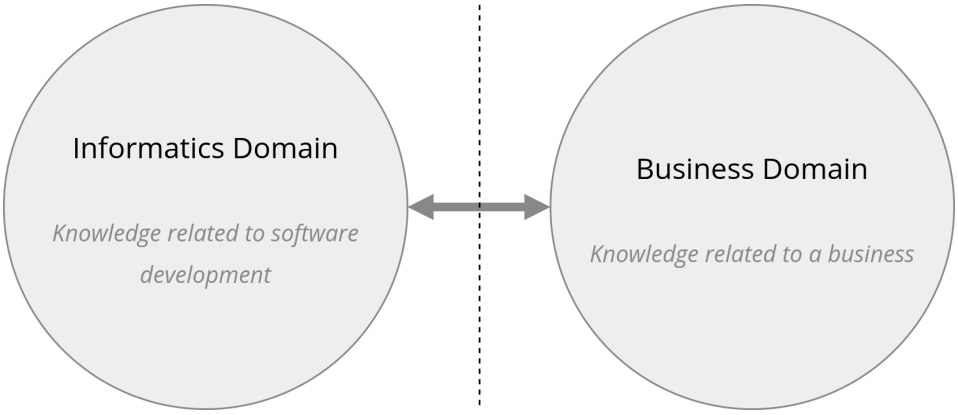

Figure 1.2: Technological vs business Domains

Example: User Experience testing

As example, consider building a software for qualitative user experience testing of websites. The goal is for customers to identify areas and functionalities on their website that cause user frustration. For this purpose, a JavaScript snippet is provided that records user interaction data and sends it to a server. The received information is matched against patterns that indicate user frustration, and the results are presented to the customer. For example, repeated clicks at random positions may be categorized as so-called "rage clicks". Building such a software requires knowledge in many purely technological areas, such as DOM Events and HTTP. However, the User Experience part is the relevant business Domain. Without understanding and applying its specialized knowledge, it is almost impossible for the software to succeed.

Domain Experts

Domain Experts are the primary source of specialized knowledge that enables to create an adequate software-based solution to a problem. For every project, there should be at least one person with this role. If this is not the case, the relevant Domain knowledge can be acquired by individuals, who then become the experts. In general, Domain Experts should be able to provide helpful answers to most of the arising questions. However, they cannot be expected to know "everything" about a respective Domain. Also, the knowledge may be shared across multiple persons, each with their own specialty. Furthermore, some facts may simply be unknown. In such situations, it makes sense to put effort into discovering the unknown parts and gaining new insights.

Relation to other roles

Typically, a Domain Expert is a secondary role on top of the existing ones inside a software project. Theoretically, the role can be taken on by anyone, be it developers, designers, product managers or agile coaches. Still, it fits best with project members that work closely with the actual customer. Most commonly, this is either a product manager or a product owner. Depending on the project, even the customer themselves can be a Domain Expert. However, such a constellation requires to always strictly distinguish between relevant knowledge and unprocessed feature inquiries. While a customer often understands the Domain in question well, the actual functionalities of a software are best defined by the project team.

Domain Experts in small projects

For small projects, it may be that you as a developer consider yourself the Domain Expert. While this is possible, the setup demands some precaution. Being the developer, the Domain Expert and maybe even the client simultaneously can be problematic. Requirements can become fuzzy or even get abandoned. Make sure to always focus on the actual problem and its solution.

Subdomains

Subdomains are distinguishable knowledge areas that are part of a larger compound. In theory, almost every Domain can be understood as a collection of individual self-contained parts. Likewise, multiple Domains can be grouped together with others into an overarching one. This is mainly a matter of perspective and scope. Amongst other things, it also illustrates the ambiguity of the term “Domain”. On the one hand, it can stand for the whole of something. On the other hand, it can describe a subordinate part. For software projects, the overall Domain may not be a universally meaningful knowledge area. Rather, it can be a collection of parts that may even be unrelated. There are three types of Subdomains, of which each is described as a following subsection.

Chapter 6: Value Objects, Entities and Services

The implementation of a Domain Model is an expression of abstractions as source code, targeted to solve specific problems. If this part does not manage to fulfill its duties appropriately, the whole surrounding project is likely to fail its purpose. Therefore, it should be considered the heart of every software. DDD provides a number of tactical patterns that guide the code design of the Domain layer. These patterns can be applied independently of the concept DDD as a whole. At a minimum, a Domain Model implementation should consist of a combination of Value Objects and Entities. On top of that, stateless functionalities can be designed as Domain Services. Applying these tactical patterns helps to express individual Domain Model concepts adequately.

Type checking and validation

As explained in the Preface, the code for this book makes only limited use of type checking and data validation. In this chapter, both of the mechanisms are used sporadically for protecting invariants and ensuring integrity in Domain Model implementations.

Value Objects

The implementation of a Domain Model component is typically either done as Value Object or as Entity. **Value Objects** quantify or describe an aspect of a Domain without conceptual identity. They are defined by their attributes. [Evans, p. 98] describes them as elements "that we care about only for what they are, not who or which they are". For example, two date objects are considered the same as long as their day, month and year match. According to [Vernon, p. 219], such objects are "easier to create, test, use, optimize and maintain". Therefore, they should be used extensively. Value Objects must provide a few key characteristics. The most important ones are explained in the following subsections. This is preceded by the introduction of an overarching example topic.

Value Objects versus data

Value Objects may have functions and carry out domain-specific behavior. They are first class citizens of a Domain Model implementation and must not be confused with plain data structures. The subsection on Immutability provides an example to illustrate such meaningful behavior.

Example: Floor planning software

Consider implementing selected parts of the Domain Model for a floor planning software. Its purpose is to enable virtual room layout plannings. For the example, two selected concepts of the Domain Model are implemented. One is the furniture component, which consists of a type, a width and a length. The type is represented as a simple string. In contrast, both the width and the length are so-called measurements, which is the second relevant Domain Model part. The measurement concept is defined as a combination of magnitude and unit. While the magnitude is a plain number, the unit is an element out of a list of fixed values. In the following subsections, both components are implemented in different ways to illustrate the respective Value Object characteristics.

Conceptual Whole

A Value Object is a collection of coherent attributes and related functionality that form a Conceptual Whole. This means, it is a self-contained structure that captures some descriptive aspect of a Domain. The consequential boundary makes it possible to give an intention-revealing name based on the respective Ubiquitous Language. In most scenarios, attributes that belong to a common conceptual unit should not be placed separately on their own. Doing so fails to capture the idea of the Domain Model. At the same time, the attributes must not be embedded into larger structures that are focused on other concepts. Otherwise, the code establishes incorrect boundaries that do not align with the actual abstractions. Related attributes and their associated actions should be combined into meaningful Value Objects.

The first example shows standalone attributes that together represent a single furniture instance (run code):

Floor Planning: Standalone attributes

```
const furnitureType = 'desk';
const furnitureWidth = 100, widthUnit = 'cm';
const furnitureLength = 60, lengthUnit = 'cm';

console.log(`dimensions of ${furnitureType} furniture:`);
console.log(`${furnitureWidth}${widthUnit} x ${furnitureLength}${lengthUnit}`);
```

The second example provides a class that expresses the furniture concept as Value Object type (run code):

Floor Planning: Furniture Value Object

```
class Furniture {

  type; length; lengthUnit; width; widthUnit;

  constructor({type, length, lengthUnit, width, widthUnit}) {
    Object.assign(this, {type, length, lengthUnit, width, widthUnit});
  }

  toString() {
    return `${this.type}, ${this.width}${this.widthUnit} wide`
      + `, ${this.length}${this.lengthUnit} long`;
  }

}

const desk = new Furniture(
  {type: 'desk', length: 60, lengthUnit: 'cm', width: 100, widthUnit: 'cm'});
console.log(desk.toString());
```

The first approach using standalone attributes does not express the domain-specific concepts furniture and measurement in a meaningful way. There is no structural relationship or enclosing boundary across the individual values. Not even their variable naming unambiguously states that there is a conceptual connection between them. In contrast, the second implementation correctly captures the furniture concept as the class `Furniture`. The code defines a structure that encloses related aspects. However, it mixes multiple concepts into a single unit and therefore establishes an incorrect boundary. This is because the furniture component must not be concerned with measurement details. Rather, they must

be encapsulated into a separate Value Object type. Embedding them directly makes it less clear what the intrinsic attributes of furniture instances are.

The next example implements both the furniture and the measurement concept as Value Object type (run code):

Floor Planning: Measurement Value Object

```
class Measurement {

  magnitude; unit;

  constructor({magnitude, unit}) {
    Object.assign(this, {magnitude, unit});
  }

  toString() {
    return `${this.magnitude}${this.unit}`;
  }

}

class Furniture {

  type; length; width;

  constructor({type, length, width}) {
    Object.assign(this, {type, length, width});
  }

  toString() {
    return `${this.type}, ${this.width} wide, ${this.length} long`;
  }

}

const width = new Measurement({magnitude: 100, unit: 'cm'});
const length = new Measurement({magnitude: 60, unit: 'cm'});
const desk = new Furniture({type: 'desk', length, width});
console.log(desk.toString());
```

The class `Measurement` accurately expresses the Domain Model definition of a measurement as a dedicated Value Object type. Instead of maintaining individual attributes, the component

`Furniture` uses measurement objects for both the width and the height. This allows the class to focus on its own specific attributes. Aspects that are only relevant to details of measurement components do not affect the state of furniture. For example, consider changing a desk's width from 100 centimeters to 120 centimeters. With the previous approach, this requires multiple attributes inside the furniture to change. When using separate self-contained measurement Value Objects, the state of the enclosing furniture remains largely unaffected. The measurement of 100 centimeters is simply replaced by a new one representing 120 centimeters.

Equality by Values

Two Value Objects of the same type are considered equal when all their corresponding attributes have equal values. Consequently, it is also irrelevant whether they are represented by the same reference object. This characteristic is closely tied to the absence of a conceptual identity. Compare this to primitive types such as strings or numbers. The character sequence "test" is always equal to "test", independent of the technical handling of strings. The number 42 is always equal to 42, even if both values originate from different memory addresses. It is a matter of equality and not identity. The same applies to Value Objects. However, other than primitives, they consist of multiple parts. Therefore, all their attributes need to match for Value Objects to count as equal.

The following code example implements Equality by Value for the previously introduced measurement component (run code):

Measurement: Equality by Values

```
class Measurement {

  magnitude; unit;

  constructor({magnitude, unit}) {
    Object.assign(this, {magnitude, unit});
  }

  equals(measurement) {
    return measurement instanceof Measurement &&
      this.magnitude === measurement.magnitude && this.unit === measurement.unit;
  }

}
```

```
const desk1Height = new Measurement({magnitude: 60, unit: 'cm'});
const desk2Height = new Measurement({magnitude: 60, unit: 'cm'});
console.log(`are the heights equal? ${desk1Height.equals(desk2Height)}`);
```

The updated implementation of the class `Measurement` provides the operation `equals()` to check whether two Value Objects are equal. As argument, it expects another measurement to compare to. First, the operation verifies that the provided argument is an actual measurement. As described in the beginning of the book, type checking in JavaScript is limited in its possibilities. Depending on the respective context, one particular mechanism can be most useful. Here, the example code uses the `instanceof` keyword to verify a correct constructor. In case of success, the attributes for unit and magnitude are tested for value equality. Finally, the operation returns either `true` or `false`. The usage code demonstrates that two different reference objects are considered equal when both their unit and magnitude match.

Immutability

Value Objects must be immutable. This means that their attributes are not allowed to change after an initial assignment. Consequently, all necessary values must be provided at construction time. The main motivation for this is best understood when comparing it to mathematics. A number never changes and always represents one specific numerical value. An addition of two numbers refers to a third one, but does not cause any modification. Algebraic variables can change the values they refer to, but the values themselves remain constant. The same logic applies to Value Objects. This has some useful advantages. For one, an object that is initially valid always remains valid. There is no later verification or validation required. Secondly, working with an immutable object cannot cause side effects.

The next implementation shows an approach for enforcing Immutability on the example measurement Value Object (run code):

Measurement: Immutability

```
const Measurement = function({magnitude, unit}) {
  Object.freeze(Object.assign(this, {magnitude, unit}));
};

const deskWidth = new Measurement({magnitude: 100, unit: 'cm'});
try {
  deskWidth.magnitude = 200;
} catch (error) {
  console.log(error.message);
}
```

The implementation of the component `Measurement` represents an immutable Value Object type without dedicated behavior. For this particular example, the component is implemented as plain constructor function. Upon instantiation, both the values for magnitude and unit are assigned to the according attributes. Afterwards, the operation `Object.freeze()` is executed with the newly created instance as argument. This closes the measurement for any kind of future modification, including value assignments. In terms of Immutability, the same result can be achieved with the functions `Object.defineProperty()` or `Object.defineProperties()`. Even more, when only parts of an object should be immutable, those operations must be used. The exemplary usage demonstrates that any attempt to modify an attribute of a frozen object results in an exception.

Chapter 9: Repositories

Repositories enable persistence in a meaningful way with regard to the concepts of a Domain Model. Their design goal is to emphasize domain-related aspects as opposed to technological concerns. Through expressive interfaces, the focus lies on components and collections rather than columns and tables, or files and directories. This requires all infrastructural details to be encapsulated inside their implementations. Meaningful Domain behavior is expressed through explicit queries instead of technical search capabilities. Typically, Repositories and Aggregates share a one-to-one relationship, where the root Entities are the directly accessed components. Regardless of the respective alignment, each transactional consistency boundary must have its isolated realm of persistence. Repositories are typically also capable of providing concurrency control, for example through an optimistic locking mechanism.

One-to-one alignment with Aggregates

This chapter exclusively focuses on one Repository per Aggregate type. While there are other valid constellations, this scenario is the most common one. [Vernon, p. 401] also states that there is typically "a one-to-one relationship between an Aggregate type and a Repository".

Domain Model emphasis

The design goal of Repositories is to put emphasis on the Domain Model instead of technological aspects. Generally, this pattern must not be confused with Data Access Objects (DAO) or Object Relational Mapping (ORM). Despite their common purpose of persistence, the latter concepts can distort the design and the implementation of a Domain Model. Repositories promote a strict and clean separation. Every interface must only reflect existing domain-specific concepts or, alternatively, introduce new ones. Consequently, the used terminology must apply the respective Ubiquitous Language. In contrast, an implementation is allowed to contain technological details, as long as they are encapsulated. Designing a Repository with focus on the Domain Model also avoids extraneous functionalities. Furthermore, providing meaningful querying capabilities prevents Domain concerns from leaking into consumer code.

Use cases for technical interfaces

There are scenarios where it can be useful to provide technical and parametrizable query functionalities. However, the need for this must emerge from the Domain Model. For example, e-mail clients commonly provide the ability to search items based on technical and combinable criteria. In this case, the functionality is specific to the problem space and its Domain.

Design and implementation

The general design recommendation for Repositories is to provide a technology-agnostic interface. On top of that, there are more specific guidelines. [Evans, p. 151] suggests to "provide the illusion of an in-memory collection [..] with more elaborate querying capability". [Vernon, p. 402] describes two specific design approaches. One is called "collection-oriented" and is equivalent to what Evans describes. The idea is to mimic the appearance and the functionality of collections. Despite possible advantages, the approach requires automatic change tracking, which can cause implicit behavior. The second design is called "persistence-oriented". With this approach, a Repository may also resemble a collection, but clearly conveys its persistence-specific purpose. While this demands consumers to be aware of persistence, it is more explicit and intention-revealing. The examples in this book exclusively use persistence-oriented Repositories.

Basic functionality

The basic functionality of a Repository is to save and to load individual Aggregate instances through their root Entity. For most components, the save operation is identical and stores a single element. What the querying part consists of, depends on the Domain Model requirements. One essential functionality is to load an object via its identifier. Apart from this, there are endless possibilities. Queries may also respond with computational results such as counts. Repositories should always explicitly convert objects into data and vice versa to differentiate between object representation and persisted information. Generally, it is not advisable to save elements as-is. At a minimum, a conversion ensures data integrity. In JavaScript, Repository interfaces should always be asynchronous, even if the underlying persistence mechanism is synchronous.

What about data deletion?

Apart from saving and loading data, another essential functionality is to delete existing records. This aspect is excluded from the book due to multiple reasons. For one, it causes additional complexity. Secondly, data deletion plays a subordinate role when applying Event Sourcing, which is covered later in the book.

The first code example provides a function to save a file atomically, as explained in the previous chapter (run code usage):

Filesystem: Write file helper function

```
const writeFileAtomically = async (filePath, content) => {
  const tempPath = `${filePath}-${generateId()}.tmp`;
  await writeFile(tempPath, content);
  await rename(tempPath, filePath);
};
```

The second code shows a simple filesystem-based Repository class:

Repository: Filesystem Repository

```
class FilesystemRepository {

  storageDirectory; #convertToData; #convertToEntity;

  constructor({storageDirectory, convertToData, convertToEntity}) {
    mkdirSync(storageDirectory, {recursive: true});
    Object.defineProperty(
      this, 'storageDirectory', {value: storageDirectory, writable: false});
    this.#convertToData = convertToData;
    this.#convertToEntity = convertToEntity;
  }

  async save(entity) {
    const data = this.#convertToData(entity);
    await writeFileAtomically(this.getFilePath(entity.id), JSON.stringify(data));
  }

  load(id) {
    return readFile(this.getFilePath(id))
      .then(buffer => this.#convertToEntity(JSON.parse(buffer.toString())));
  }
```

```
  getFilePath(id) {
    if (!id) throw new Error('invalid identifier');
    return `${this.storageDirectory}/${id}.json`;
  }

}
```

The class `FilesystemRepository` implements a minimal Repository interface and persists Entities as JSON strings in the filesystem. Its constructor expects three arguments. The parameter `storageDirectory` determines which directory is used as storage location. How to convert an Entity into data and vice versa is controlled with the function parameters `convertToData` and `convertToEntity`. Persisting an object is achieved with the `save()` command. This operation converts an element into data, creates a JSON string and invokes the function `writeFileAtomically()`. The respective `id` attribute is used as filename. Loading an Entity is done via the query `load()`. For this operation, the JSON string is read from a file, de-serialized and transformed into an Entity. Through the configurable converters, the Repository class can be used for arbitrary components.

Chapter 11: Command Query Responsibility Segregation

Command Query Responsibility Segregation (CQRS) is an architectural pattern that separates a software into a write side and a read side. Typically, use cases that perform write operations differ in requirements and technical characteristics from the ones that primarily read information. These differences can include associated data and its structure, requirements for availability, consistency and performance, and even underlying code complexity. Still, many systems treat both use case types in a very similar way. They are processed by the same set of components and share a single data storage. Applying CQRS helps to clearly differentiate between write and read concerns and enables to evolve and scale them independently. Furthermore, it promotes to isolate the more critical write part of a Domain Model implementation.

Relation between CQRS and CQS

The architectural pattern CQRS is partially related to the design pattern CQS. To some extent, CQRS is CQS on a higher abstraction level. Initially, it was even referred to by the same name. The introduction of the term "Command Query Responsibility Segregation" aimed to clearly differentiate between the two concepts.

Architectural overview

CQRS divides a software into a write side and a read side. Typically, the pattern is applied to a selected area, but not to a complete system. For DDD-based software, this can map to a context implementation. The concept itself does not dictate specific technological decisions, but only implies that write and read concerns are somehow separated. Still, in many cases the two sides employ individual data storages and establish a synchronization mechanism. On top of that, they can apply custom concepts, run in different processes and even use distinct technologies. While a one-to-one alignment of both sides can be useful, it is not mandatory.

One write side may affect multiple read aspects and one query functionality can aggregate information from several write sides.

Separate data storages

This chapter focuses on explaining and illustrating CQRS with separate data storages for the write side and the read side. While it is possible to use the same storage, employing separate ones makes most sense when working towards Event Sourcing.

High-level picture

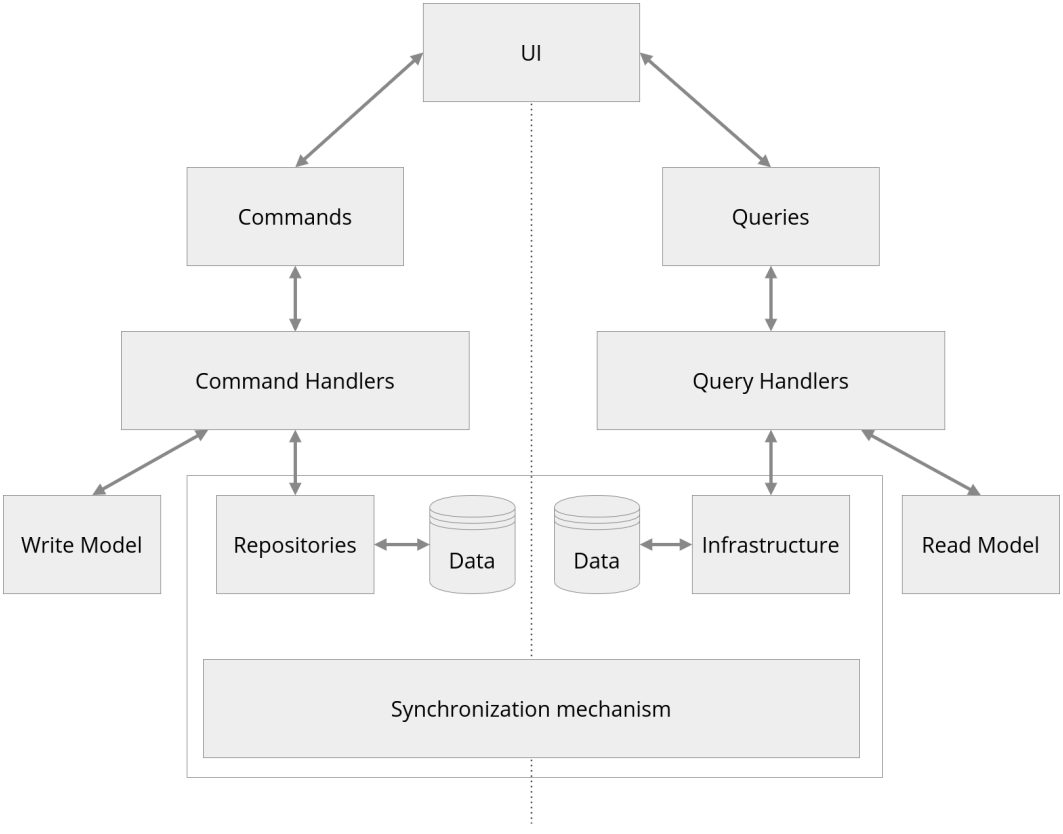

Figure 11.1: CQRS

The User Interface Layer is responsible for translating each meaningful user action into either a Command or a Query. The resulting messages are delivered to their responsible handler, which belongs to the Application Layer. Command Handlers load data via Repositories, construct Write Model components and execute a target action with the provided arguments. When encountering an error, the overall process is aborted. In case of success, the resulting state change is persisted. The synchronization mechanism consumes Write Model state and forwards it to domain-specific transformations that produce Read Model data. This information can be persisted with Repositories, but may also be saved with other storage mechanisms. Query Handlers load data based on given parameters, optionally construct domain-specific components and return a result.

Dependencies between write side and read side

Generally speaking, the two parts of a CQRS-based software should have minimal interdependencies. Naturally, a read side always depends on a write side for deriving its Read Models. In contrast, a write side should not depend on the other part. However, [Young] describes that "there are times where you will have to have the write side query the read side".

Implicit CQRS

Many software projects implicitly apply CQRS without consciously aiming for it. This is because it often makes sense to execute different types of use cases in different ways. As explained earlier, it can be about data structures, consistency models, performance aspects or associated code complexity. Even the sole use of a caching mechanism can be interpreted as some form of CQRS. While Command Handlers operate on current state, the according Query Handlers respond with cached data. Another common scenario is when a software provides reporting functionalities. Typically, this is achieved by exposing preprocessed and aggregated data from a specialized storage. As example, consider a large online software platform that determines its number of users. Most certainly, this is not done by querying a production database.

Write and Read Model

The architectural segregation of write and read concerns typically causes the Domain Model implementation to be split into two parts. [Vernon, p. 139] explains that "we'd normally see

Aggregates with both command and query methods". When applying CQRS, the implementation for these two concerns is separated from each other. All the actions that cause a state to change are placed in the write part. In contrast, the data and structures that are used for querying and displaying information belong to the read side. The associated persistence-related mechanisms are optionally split and direct to individual data storages. Note that the conceptual Domain Model itself is not strictly required to reflect this technological separation. Nevertheless, in most cases it makes sense to account for this aspect.

Different terminologies

The Write Model and the Read Model can alternatively be called Command Model and Query Model. However, the terms Command and Query describe concepts that are part of the Application Layer. As the model implementation belongs to the Domain Layer, the use of distinct terms helps to differentiate between those areas.

Chapter 12: Event Sourcing

Event Sourcing is an architectural pattern where state is represented as a sequence of change events. The events are treated as immutable and get persisted in an append-only log. Any current data is computed on demand and generally considered transient. Capturing state as a set of transitions enables to emphasize intentions and behavior instead of focusing primarily on data structures. According to [Young], another benefit is that "time becomes an important factor" of a system. Yet, the most distinct advantage is that historical event logs allow to derive specialized data for read-related scenarios, even retroactively. Event Sourcing can only be applied adequately together with some form of CQRS. Independent of this constraint, the pattern is commonly used in combination with both Domain Events and Event-Driven Architecture.

Architectural overview

Event Sourcing models state as a series of transitioning events. As with CQRS, the pattern typically targets subordinate areas or specific contexts, but rarely a complete software. Even more, individual parts of a system should not rely on the fact that other areas apply this pattern. Instead, it should be treated as an internal implementation detail. However, it is acceptable to provide query functionalities that return historical or event-related information in a controlled manner. Other than with CQRS, Event Sourcing only affects certain architectural parts. This primarily includes the Write Model implementation, its associated infrastructural functionalities and the synchronization mechanism for Read Model data. In contrast, most parts of the read side, its storage components and the User Interface Layer largely remain unaffected.

Relation between CQRS and Event Sourcing

CQRS was originally meant to be a stepping stone when working towards Event Sourcing. However, the pattern can also be applied independently and its core idea is even found in other concepts. In contrast, Event Sourcing primarily makes sense when a software also separates its write side and read side.

Overall flow

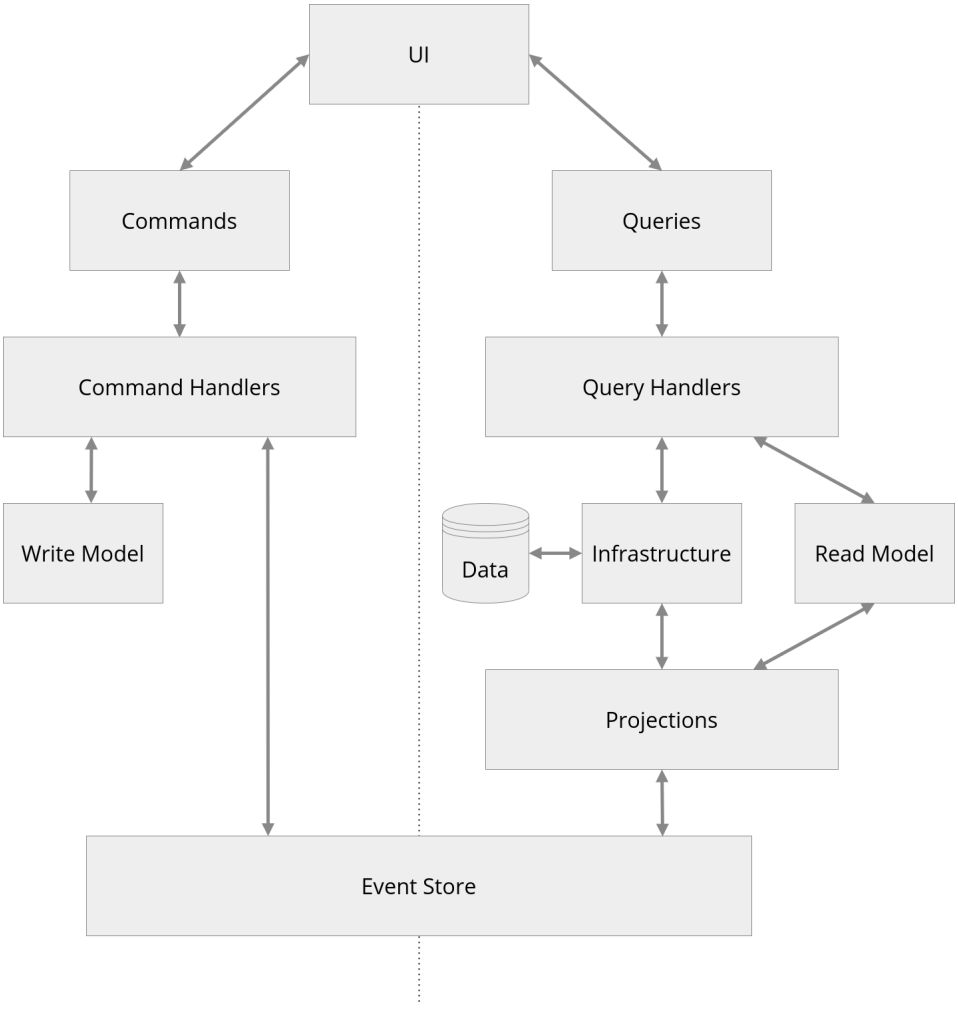

Figure 12.1: Event Sourcing

The write side retrieves events from the Event Store, rebuilds state representations for Write Model components and executes target behavior. All new events are appended to their according stream inside the store. For DDD-based software, the event streams are typically separated by Aggregates. After persisting, all interested subscribers are informed about the state changes. The read side operates Projection components, which subscribe at the Event Store to update Read Model structures. Upon notification, they access data from

their associated storage, optionally construct Read Model components and perform necessary updates. For specialized use cases, the read side may even expose direct event stream access. In terms of CQRS, the Event Store is both the persistence mechanism of the write side and part of the synchronization mechanism.

Event anatomy

The recommendations for Domain Events with regard to types, structure and content largely also apply to Event Sourcing. Every event should have an expressive and unambiguous type name that adequately describes the enclosed change. The term should be a combination of the affected subject and the executed action in past tense. For each event, the contents should consist of both data and metadata, where individual fields are represented as simple attributes. The data part should include the type name, the identity of the affected subject and all changed values. Other than with Domain Events, there should be no derived or extraneous data. The metadata section should contain generic information independent of event types. When using Event Sourcing, this part often includes additional persistence-related fields.

Relation to Domain Events

While Domain Events and Event Sourcing events share similarities, they are conceptually not the same. Domain Events represent meaningful Domain occurrences and can be distributed for both internal and external consumption. They may contain redundant and additional data to satisfy consumer needs. Event Sourcing captures state changes as events and uses them as persistence records. These items may be ill-suited for an outbound distribution. They are rather fine-grained, should only contain changed data and may have a local meaning. Exposing them directly risks to couple external functionalities to internal details. Therefore, there should be a dedicated mechanism that translates them to Domain Events. At a minimum, this functionality filters out items that are considered private. Furthermore, it can perform domain-specific transformations, such as data enrichment.

Every event is a Domain Event

In fact, all events can be considered Domain Events. However, they may differ in their scope of validity. Private events are only meaningful inside their enclosing context. In contrast, public events are part of a larger conceptual boundary. This book only labels items that are made available for external consumption as Domain Events.

Example: Video platform channel subscription

Consider implementing the Event Sourcing events and the Domain Event creation mechanism for channel subscriptions of a video platform. The goal is to be able to inform channel owners whenever a subscription is added or removed. Overall, the possible state transitions for an individual subscription consist of adding it, updating its notification settings and removing it. Both the addition and the removal must be treated as public Domain Events in order to notify channel owners. In contrast, the notification settings update is considered a private detail. Even though this occurrence may also be useful for external consumption, there is initially no need for exposing it. The mechanism for yielding Domain Events should only forward the public event types and filter out all private items.

The following code provides the event definitions together with an operation for yielding Domain Events (run code):

Video platform: Channel subscription events

```
const SubscriptionAddedEvent = createEventType('SubscriptionAdded',
  {subscriptionId: 'string', userId: 'string', channelId: 'string'});

const SubscriptionNotificationsSetEvent = createEventType(
  'SubscriptionNotificationsSet',
  {subscriptionId: 'string', notifications: 'boolean'});

const SubscriptionRemovedEvent = createEventType(
  'SubscriptionRemoved', {subscriptionId: 'string'});

const publicEventTypes = [SubscriptionAddedEvent, SubscriptionRemovedEvent];
const yieldDomainEvents = event => publicEventTypes.some(
  Event => Event.type === event.type) ? [event] : [];

const subscriptionId = generateId();
const userId = generateId();
const channelId = generateId();

const events = [
  new SubscriptionAddedEvent({subscriptionId, userId, channelId}),
  new SubscriptionNotificationsSetEvent({subscriptionId, notifications: true}),
  new SubscriptionRemovedEvent({subscriptionId}),
];

const domainEvents = events.flatMap(yieldDomainEvents);
```

The code starts with creating the event types "SubscriptionAdded", "SubscriptionNotificationsSet" and "SubscriptionRemoved". Next, the array `publicEventTypes` is initialized with all public event types. Then, the function `yieldDomainEvents()` is implemented, which expects an event as argument and returns a list of Domain Events. For each addition or removal of a subscription, it returns the original item. For all other types, an empty array is returned. The actual usage code starts with defining identifiers for a subscription, a user and a channel. Afterwards, three events are instantiated, one for each defined type. The items are passed individually to the function `yieldDomainEvents()` and the combined return value is logged to the console. Executing the code demonstrates that the settings update does not appear as Domain Event.

Specialized Domain Event types

As alternative to the example implementation, consider a separate Domain Event type to represent a channel subscription count change. The data could consist of both the previous and the new subscriber count. For each subscription addition and removal, the event type would be instantiated and forwarded. The denormalized data would free consumers from manually determining the current subscription count.

Event-sourced Write Model

The Write Model implementation for a software based on Event Sourcing must exhibit specific behavior. For one, each Aggregate type must be able to build its required state representation from a list of events. Secondly, all behavioral operations must not directly mutate state, but produce events that represent a respective change. As optional side effect, the information may be used to synchronize its state representation. Finally, all new events must be exposed to the outside for persistence purposes. There are multiple advantages with event-sourced Write Models. According to [Young], modeling in events "forces you to think about behavior". Also, state representations can be structured freely without accommodations for persistence. One potential downside is the additional complexity of an event-sourced design and the resulting implementation.