Data Oriented Programming

Unlearning objects

Yehonathan Sharvit

MEAP

MANNING

MEAP Edition
Manning Early Access Program

Data-Oriented Programming

Unlearning objects
Version 2

welcome

Thank you for purchasing the MEAP for *Data-Oriented Programming*.

The book is written for developers having experience in a **high level programming language**. It could be a classic Object Oriented language like Java or C# or a dynamically typed language like JavaScript, Ruby or Python. We assume that you have already built (alone or in a team) a couple of **web systems**, either backend or frontend.

Data Oriented (DO) Programming is a programming paradigm that makes the systems we build **less complex**. The cool thing is that DO is **language agnostic**: it is applicable to any programming language.

I discovered Data Oriented programming ten years ago when I started to code in Clojure. Since then, the quality of my design and my code has increased significantly, and the systems I build in Clojure and in other programming languages are much simpler and much more flexible.

DO is based on **three fundamental principles** that we expose briefly in Chapter 0. The principles might seem basic at first sight, but when you apply them in the context of a production-ready information system, they become very powerful.

Chapter 1 exposes some **common pains** that **Object Oriented developers** experience when they develop a system. Please don't read it as a critic of Object Oriented Programming. The main purpose of Chapter 1 is to motivate you to learn a different programming paradigm.

Starting from Chapter 2, we expose -- one by one -- the three principles of DO and their benefits in the context of a production-ready information system.

In order to make the teachings very concrete, we demonstrate how the principles of DO are translated in code. We have chosen JavaScript as the main language for the **code snippets** of the book, but the ideas are applicable to any programming language. We have chosen JavaScript because it supports both Object Oriented and Functional programming styles and its syntax is easy to read even for folks not familiar with JavaScript.

The book is full of **diagrams** and mind maps that illustrate the ideas.

The teachings of the book are conveyed through a **story** of an Object Oriented programmer who meets a Data Oriented expert and learns from him how DO makes a system less complex and more flexible.

I hope that you find the **conversation** between the developer and the expert fun to read and that it clarifies the teaching in the sense that the questions the developers ask the expert resonate well with the questions you ask yourself during reading.

Each chapter closes with a famous song with **modified lyrics** related to the teachings of the chapter. To best enjoy the modified lyrics, I encourage you to listen to the song on Youtube or Spotify while reading.

I truly believe that Data Oriented Programming will make you a **better developer**, as has been the case for me since I discovered it ten years ago. I look forward to reading any questions or comments you may have along the way on Manning's liveBook Discussion Forum. Your feedback is an invaluable part of making this book the best that it can be.

One last thing, the name of the main character of the book is: You!

-- Yehonathan Sharvit

brief contents

INTRODUCTION

0 *Principles of Data-Oriented programming*

PART 1: FOUNDATIONS

1 *The tendency of Object-Oriented Programming towards increased system complexity*

2 *Reduce system complexity by separating Code from Data*

3 *Manipulate the whole system data with generic functions*

4 *State management with immutable data*

5 *Highly scalable concurrency control*

PART 2: IMPLEMENTATION

6 *Data inspection without reflection*

7 *Polymorphism without Objects*

8 *Persistent data structures kills bugs in the egg*

9 *Lock free atomic data comparison*

10 *Write unit tests without mocking*

11 *Debug your programs without a debugger*

PART 3: EVOLUTION

12 *Write your own data manipulation tool set*

13 *Extend the primitive data types*

14 *Specify the shape of your data*

15 *Write unit tests that cover every possible data combination with property based testing*

APPENDIX

A *Immutable functions (lodash, ramda)*

B *Conflicting keys algorithm*

0 Principles of Data-Oriented Programming

0.1 Introduction

Data-Oriented programming is a programming paradigm aimed to **simplify** the **design** and **implementation** of software systems where **information** is at the center. Instead of designing information systems around entities that combine data and code together (e.g. objects instantiated from classes), DO encourages us to **separate code from data**. Moreover, DO provides guidelines about how to **represent** and **manipulate** data.

The essence of DO is that it treats **data a first class citizen**. As a consequence, in Data Oriented programs, we **manipulate** data with the same **simplicity** as we manipulate numbers or strings in any other programs.

TIP In Data Oriented programming, data is a first class citizen.

Treating data as a first class citizen is made possible by adhering to three **core principles**. This chapter presents at a high level the core principles of Data Oriented (DO) Programming.

The principles of Data Oriented (DO) Programming are:

1. Separate **code** from **data**
2. Represent data entities with **generic** data structures
3. Data is **immutable**

When those 3 principles are combined together, they form a cohesive whole as shown in Figure 0.1, that allows us to treat data as a first class citizen. As a consequence, we **improve our developing experience** and makes the systems we build **easier to understand**.

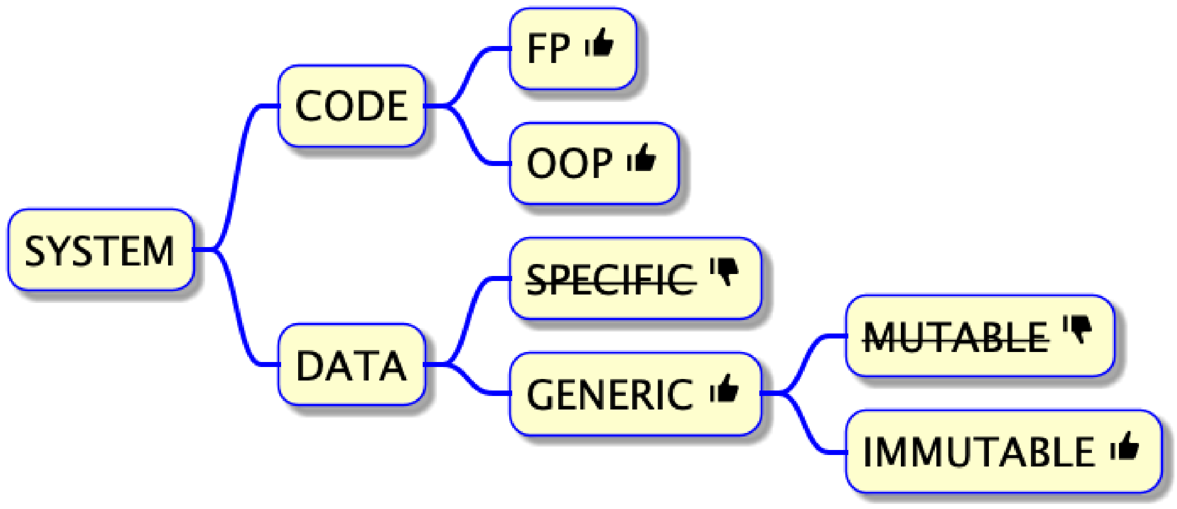


Figure 0.1 The principles of Data Oriented programming

TIP In a Data oriented system, code is separated from data and the data is represented with generic data structures that are immutable.

It is important to understand that DO principles are **language agnostic**: One could **adhere** to them or **break** them in:

- **Object Oriented** (OO) languages: Java, C#, C++…
- **Functional Programming** (FP) languages: Clojure, Ocaml, Haskell…
- Languages that support **both OO and FP**: JavaScript, Python, Ruby…

TIP DO Principles are language agnostic.

WARNING For OO developers, the transition to DO might require more of a mind shift than for FP developers, as DO guides us from the beginning to get rid of the habit of encapsulating data in stateful classes.

In this chapter, we are going to illustrate in a succinct way how those principles could be **applied** or **broken** in **JavaScript**. We chose JavaScript for two reasons:

- JavaScript supports both **FP and OOP**
- The syntax of JavaScript is **easy to read** in the sense that even if you are not familiar with JavaScript, it is possible to read a piece of JavaScript code at a high level as if it were pseudo-code

We will also mention briefly what are the **benefits** that our programs gain when we adhere to each principle and the **price** we have to pay in order to enjoy those benefits.

In this chapter, we illustrate the principles of DO in the context of **simplistic code snippets**.

Throughout the book, we will explore in depth how to apply DO principles in the context of **production information systems**.

0.2 DO Principle #1: Separate code from data

0.2.1 The principle in a nutshell

Principle #1 is a **design** principle that recommends a clear separation between code and data.

NOTE **Principle #1: Separate code from data in a way that the code resides in functions whose behavior does not depend on data that is somehow encapsulated in the function's context.**

This principle might seem like a Functional Programming principle, but in fact Principle #1 is **language agnostic**:

- We can **break** this principle in **FP**, by hiding state in the lexical scope of a function
- We can **adhere** to this principle in **OO** by aggregating the code as methods of a static class

Also, Principle #1 doesn't relate to the way data is represented. This is the theme of Principle #2.

0.2.2 Illustration of Principle #1

Let me illustrate how we can follow this principle or break it on a simplistic program that deals with:

1. An author entity with a `firstName`, a `lastName` and a number of `books` he/she wrote
2. A piece of code that calculates the full name of the author
3. A piece of code that determines if an author is prolific, based on the number of books he/she wrote

As we wrote earlier, Principle #1 is language agnostic: one could adhere to it or break it in FP or OOP languages.

Let's start our exploration of Principle #1 by illustrating first how we could break this principle in OOP.

BREAKING PRINCIPLE #1 IN OOP

We break Principle #1 in OOP, when we write code that **combines data and code together in an object**, like in Listing 0.1.

Listing 0.1 Breaking Principle #1 in OOP

```
class Author {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
    fullName() {
        return this.firstName + " " + this.lastName;
    }
    isProlific() {
        return this.books > 100;
    }
}

var obj = new Author("Isaac", "Asimov", 500); // Isaac Asimov wrote 500 books!
obj.fullName();
```

BREAKING PRINCIPLE #1 IN FP

We could also break this principle **without classes**, in an FP style, by writing code that **hides the data in the lexical scope** of a function, like in Listing 0.2.

Listing 0.2 Breaking Principle #1 in FP

```
function createAuthorObject(firstName, lastName, books) {
    return {
        fullName: function() {
            return firstName + " " + lastName;
        },
        isProlific: function () {
            return books > 100;
        }
    };
}

var obj = createAuthorObject("Isaac", "Asimov", 500); // Isaac Asimov wrote 500 books!
obj.fullName();
```

ADHERING TO PRINCIPLE #1 IN FP

After having seen how this principle could be broken in OOP and FP, let’s see how we could be compliant with this principle.

We are compliant with this principle in a FP style, when we write code that **separates the code from the data**, like in Listing 0.3

Listing 0.3 Following Principle #1 in FP

```
function createAuthorData(firstName, lastName, books) {
    return {
        firstName: firstName,
        lastName: lastName,
        books: books
    };
}

function fullName(data) {
    return data.firstName + " " + data.lastName;
}

function isProlific (data) {
    return data.books > 100;
}

var data = createAuthorData("Isaac", "Asimov", 500); // Isaac Asimov wrote 500 books!
fullName(data);
```

ADHERING TO PRINCIPLE #1 IN OOP

We could be compliant with this principle **even with classes** by writing code where the code lives in **static classes** and the data is stored in **classes with no functions**, like in Listing 0.4.

Listing 0.4 Following Principle #1 in OOP

```
class AuthorData {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}

class NameCalculation {
    static fullName(data) {
        return data.firstName + " " + data.lastName;
    }
}

class AuthorRating {
    static isProlific (data) {
        return data.books > 100;
    }
}

var data = new AuthorData("Isaac", "Asimov", 500); // Isaac Asimov wrote 500 books!
NameCalculation.fullName(data);
```

Now that we have illustrated how one could follow or break Principle #1, both in OOP and FP, let’s explore what benefits Principle #1 brings to our programs.

0.2.3 Benefits of Principle #1

When we are careful to separate code from data, our programs benefit from:

1. Code can be **reused** in different contexts

2. Code can be **tested** in isolation
3. Systems tend to be **less complex**

BENEFIT #1: CODE CAN BE REUSED IN DIFFERENT CONTEXTS

Imagine that we have in our program, beside the author entity, a user entity that has nothing to do with authors but in regard to first name and last name, it has the same data fields as the author entity: `firstName` and `lastName` fields.

The logic of the calculation of the full name is the same for authors and users. However, in the version with `createAuthorObject`, we cannot reuse the code of `fullName` on a user in a *straightforward* way.

One way to achieve code reusability when code and data are mixed is to use OO mechanisms, like **inheritance** or **composition**, to let the `user` and the `author` object use the same `fullName` method. In a simplistic use case it could be fine but on real world systems, the **abundance of classes** (either base classes or composite classes) tends to **increase the complexity** of our systems.

Another way is shown in Listing 0.5: We duplicate the code of `fullName` inside a `createUserObject` function.

Listing 0.5 Duplicating code in OO to avoid inheritance

```
function createAuthorObject(firstName, lastName, books) {
    var data = {firstName: firstName, lastName: lastName, books: books};

    return {
        fullName: function fullName() {
            return data.firstName + " " + data.lastName;
        }
    };
}

function createUserObject(firstName, lastName, email) {
    var data = {firstName: firstName, lastName: lastName, username: username};

    return {
        fullName: function fullName() {
            return data.firstName + " " + data.lastName;
        }
    };
}

var obj = createUserObject("John", "Doe", "john@doe.com");
obj.fullName();
```

In the DO version, where `createAuthorData` and `fullName` are separate, **no modifications to the existing code** (the code that deals with author) are necessary in order to make it available to the user entity. We simply leverage the fact that the data that is relevant to the full name calculation for a user and an author follows the same shape and we call `fullName` on a user data.

With no modifications, the `fullName` function works properly both on author data and on user data, as shown in Listing 0.6.

Listing 0.6 Using the same code on data entities of different types (FP style)

```
function createAuthorData(firstName, lastName, books) {
    return {firstName: firstName, lastName: lastName, books: books};
}

function fullName(data) {
    return data.firstName + " " + data.lastName;
}

function createUserData(firstName, lastName, email) {
    return {firstName: firstName, lastName: lastName, email: email};
}

var authorData = createAuthorData("Isaac", "Asimov", 500);
fullName(authorData);

var userData = createUserData("John", "Doe", "john@doe.com");
fullName(userData);
```

When Principle #1 is applied in OO, we can reuse code in a straightforward way, **even when we use classes**. In statically typed OO languages (like Java or C#), we would have to create a common interface for `AuthorData` and `UserData`, but in a dynamically typed language like JavaScript, it is not required.

The code of `NameCalculation.fullName()` works both with author data and user data, as shown in Listing 0.7.

Listing 0.7 Using the same code on data entities of different types (OOP style)

```
class AuthorData {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}

class NameCalculation {
    static fullName() {
        return data.firstName + " " + data.lastName;
    }
}

class UserData {
    constructor(firstName, lastName, email) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.email = email;
    }
}

var userData = new UserData("John", "Doe", "john@doe.com");
NameCalculation.fullName(userData);

var authorData = new AuthorData("Isaac", "Asimov", 500);
NameCalculation.fullName(authorData);
```

TIP When we separate code from data, it is straightforward to reuse code in different contexts. This benefit is achievable both in FP and in OOP.

BENEFIT #2: CODE CAN BE TESTED IN ISOLATION

Another benefit of separating code from data, which is similar to the previous one, is that we are free to **test code** in an **isolated context**.

When we don't separate code from data, we are forced to instantiate an object in order to test each of its methods. For instance, in order to test the `fullName` code that lives inside the `createAuthorObject` function, we are required to instantiate an author object, as shown in Listing 0.8.

Listing 0.8 Testing code when code and data are mixed requires to instantiate the full object

```
var author =  createAuthorObject("Isaac", "Asimov", 500);

author.fullName() === "Isaac Asimov"
```

In this simplistic scenario, it is not a big pain (only loading unnecessarily the code for `isProlific`), but in a real world situation, instantiating an object might involve lots of unnecessary steps.

In the DO version, where `createAuthorData` and `fullName` are separate, we are free to create the data to be passed to `fullName` as we want and test `fullName` in isolation. An example is shown in Listing 0.9

Listing 0.9 Separating code from data allows us to test code in an isolated context (FP style)

```
fullName({firstName: "Isaac", lastName: "Asimov"}) === "Isaac Asimov"
```

If we choose to use classes, we only need to instantiate a data object. The code for `isProlific` that lives in a separate class than `fullName` doesn't have to be loaded in order to test `fullName`, as shown in Listing 0.10.

Listing 0.10 Separating code from data allows us to test code in an isolated context (OOP style)

```
var data =  new AuthorData("Isaac", "Asimov");

NameCalculation.fullName(data) === "Isaac Asimov"
```

TIP It’s easier to write tests when we separate code from data

BENEFIT #3: SYSTEMS TEND TO BE LESS COMPLEX

The third and last benefit of applying Principle #1 is that systems tend to be less complex.

This benefit is the **deepest** one but also the one that is most **subtle** to explain.

The type of **complexity** I refer to is the one which makes systems **hard to understand** as it is defined in the beautiful paper Out of the Tar Pit. It has nothing to do with the complexity of the resources consumed by a program.

Similarly, when we refer to **simplicity**, we mean "not complex", in other words: easy to understand.

Keep in mind that complexity and simplicity (like hard and easy) are not absolute but **relative concepts**. We can compare the complexity of two systems and argue that system A is more complex (or simpler) than system B.

NOTE Complex in the context of this book means: hard to understand

When code and data reside in separate entities, the system tends to be **easier to understand** for two reasons:

1. The **scope** of a data entity or a code entity is **smaller** than the scope of an entity that combines code and data. Therefore, each entity is easier to understand.
2. Entities of the system are **split into disjoint groups**: code and data. Therefore entities have less relations with other entities.

Let me illustrate this insight on a class diagram of a Library management system, as in Figure 0.2 , where code and data are mixed.

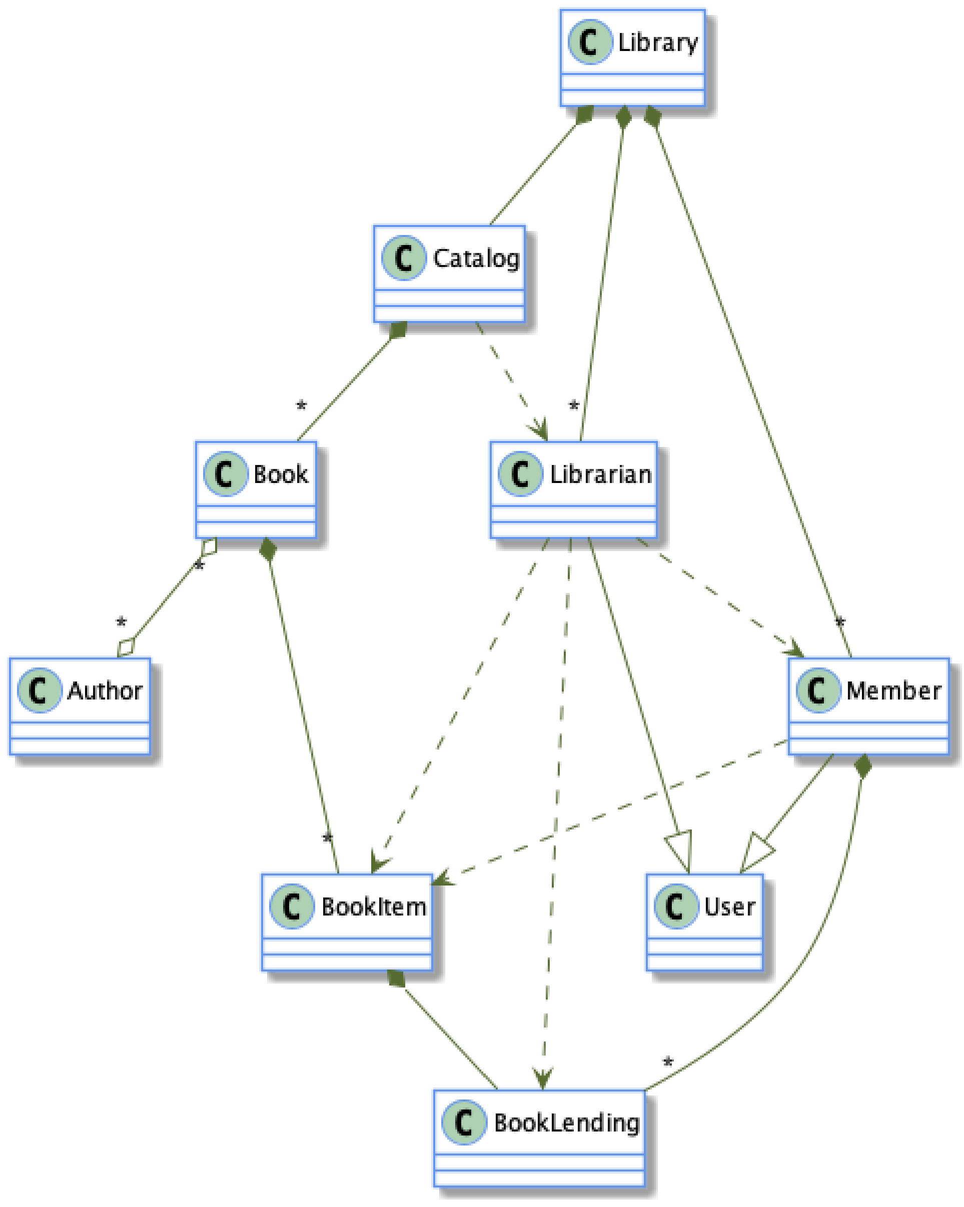


Figure 0.2 A class diagram overview for a Library management system

You don’t need to know the details of the classes of this system to get that this diagram represents a **complex system** in the sense that it is **hard to understand**. The system is hard to understand because there are many **dependencies** between the entities that compose the system.

The **most complex entity** of the system is the `Librarian` entity which is connected via 7 relations to other entities. Some relations are **data relations** (association and composition) and some relations are **code relations** (inheritance and dependency). But the in this design, the `Librarian` entity mixes code and data, therefore it has to be involved in both data and code relations.

Now, if we split each entity of the system in a code entity and a data entity *without making any further modification to the system*, we get the diagram shown in Figure 0.3, that is made of two

disconnected parts:

- The left part is made only of **data entities** and **data relations**: association and composition
- The right part is made only of **code entities** and **code relations**: dependency and inheritance

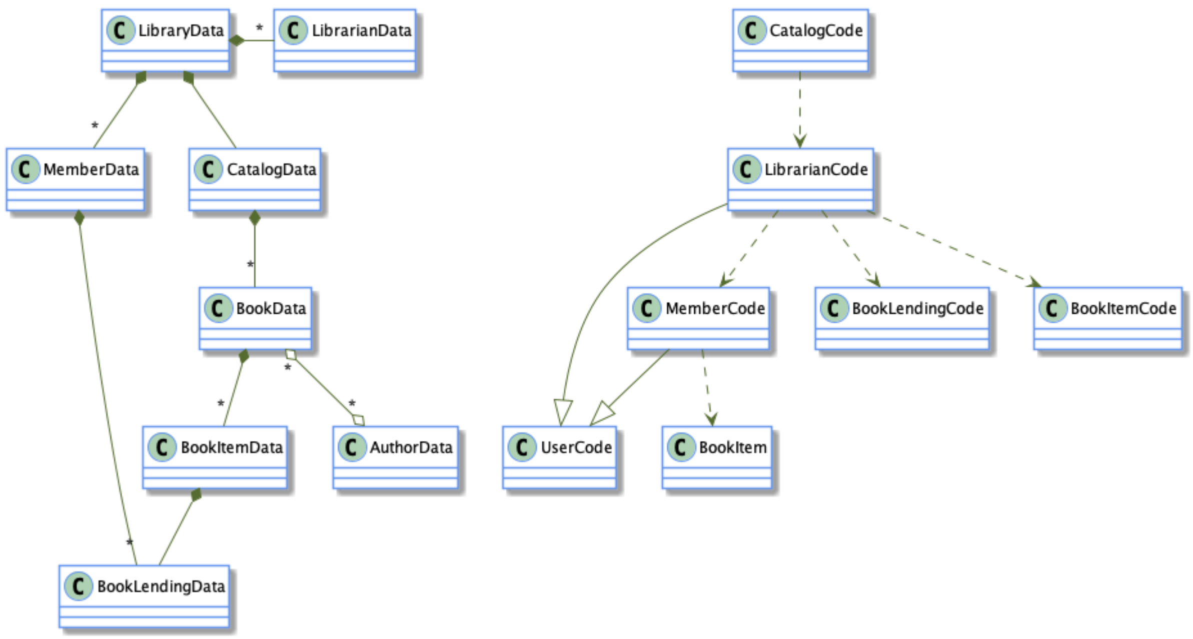


Figure 0.3 A class diagram where every class is split into code and data entities

The new system where code and data are separate is **easier to understand** than the original system where code and data are mixed: we are free to understand the data part of the system on its own and the code part of the system on its own.

TIP **A system made of disconnected parts is less complex than a system made of a single part.**

One could argue that the **complexity** of the original system where code and data are mixed is due to a **bad design** and that an experienced OO developer would have designed a simpler system, leveraging smart **design patterns**. That's true, but in a sense it's **irrelevant**. The point of Principle #1 is that a system made of entities that do not combine code and data *tends* to be simpler that a system made of entities that combine code and data.

It has been said many times that *simplicity is hard*. According to the first principle of DO, simplicity is easier to achieve when we separate code and data.

TIP **Simplicity is easier to achieve when we separate code and data.**

0.2.4 Price for Principle #1

There is no such thing as a free lunch. Applying Principle #1 comes at a price.

The price we have to pay in order to benefit from the separation between code and data is that:

1. There is no **control** on what code access what data
2. No **packaging**
3. Our systems are made from **more entities**

PRICE #1: THERE IS NO CONTROL ON WHAT CODE ACCESS WHAT DATA

When code and data are mixed, one can easily understand what are the pieces of code that access a kind of data.

For example in OO, the data is **encapsulated** in an object. It gives us the guarantee the data is accessible only by the object’s methods.

In DO, data stands on its own. It is **transparent** if you want. As a consequence, it can be accessed by any piece of code.

When we want to refactor the shape of our data, we need to be very careful and make sure that we know all the places in our code that access the data.

Without the application of Principle #3 that enforces **data immutability**, the fact that the data is accessible by any piece of code would be really **unsafe** as it would be very hard to guarantee the **validity** of our data.

TIP	**Data safety is ensured by another principle (Principle #3) that enforces data immutability.**

PRICE #2: NO PACKAGING

One of the benefits of mixing code and data is that when you have an object in hand, it’s a **package** that contains both the code (via methods) and the data (via members).

As a consequence, as a developer it’s really easy to discover what are the various ways to manipulate the data: you look at the methods of the class.

In DO, the code that manipulates the data could be everywhere. For example, `createAuthorData` could be in a file and `fullName` in another file. It makes it difficult for developers to discover that the `fullName` function is available. In some situations, it could lead to **waste of time** and unnecessary **code duplication**.

We will explore throughout the book, various ways to mitigate this drawback.

PRICE #3: OUR SYSTEMS ARE MADE FROM MORE ENTITIES

Let’s do simple arithmetic. Imagine a system made of `N` classes that combine code and data. When you split the system into code entities and data entities, you get a system made of `2N` entities.

This calculation is not accurate, because usually when you separate code and data, the class hierarchy tends to get simpler, as we need less class inheritance and composition. Therefore the number of classes in the resulting system will probably be somewhere between `N` and `2N`.

On one hand, when we adhere to Principle #1, the entities of our system are **simpler**.

On the other hand, we have **more** entities.

This price is mitigated by Principle #2 that guides us to represent our data with generic data structures.

TIP	**When adhering to Principle #1, our system is made of simpler entities but we have more of them.**

0.2.5 Wrapping up

DO guides us to separate **code** from **data**.

In OO languages, we aggregate code in **static classes** and data in **classes with no methods**.

In FP languages, we avoid hiding data in the **lexical scope** of functions.

Separating code from data comes at a price: it reduces the **control** we have on what pieces of code access our data and could cause our systems to be made of more entities.

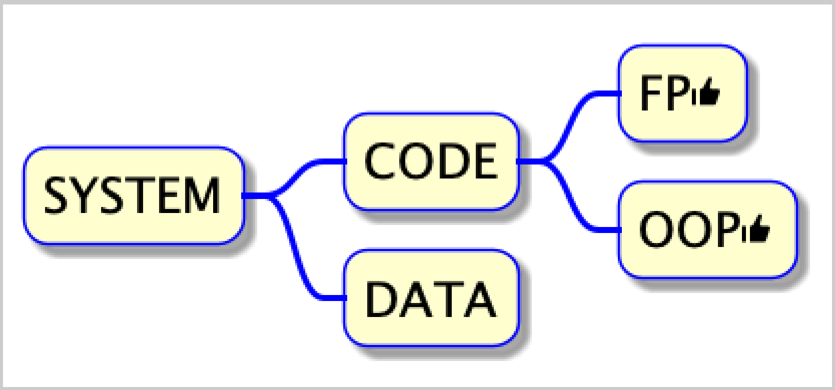
But it worth paying the price because when we adhere to this principle, our code can be reused in different contexts in a **straightforward** way and tested in **isolation**. Moreover, a system made of separate entitites for code and data tends to be **easier** to understand.

After the data has been separated from the code, comes the question of how to **represent the data**. That’s the theme of Principle #2.

SIDEBAR **DO Principle #1: Separate code from data**

The Principle

Separate code from data in a way that the code resides in functions whose behavior does not depend on data that is somehow encapsulated in the function's context.



Benefits

1. Code can be reused in different contexts
2. Code can be tested in isolation
3. Systems tend to be less complex

Price

1. No control on what code access what data
2. No packaging
3. Our systems are made from more entities

0.3 DO Principle #2: Represent data entities with generic data structures

0.3.1 The principle in a nutshell

When we adhere to Principle #1, code is separated from data. DO is not opiniated about the programming constructs to use for organizing the code but it has a lot to say about how the **data** should be **represented**. That's the theme of Principle #2.

NOTE **Principle #2: Represent the data of your application with generic data structures.**

The most common **generic** data structures are **maps** (a.k.a dictionaries) and **arrays**. But it is perfectly fine to use other generic data structures (e.g. sets, lists, queues…).

Principle #2 doesn’t deal with the mutability or the **immutability** of the data. This is the theme of Principle #3: Data is immutable.

0.3.2 Illustration of Principle #2

In DO, we represent our data with **generic data structures** (like maps and arrays) instead of instantiating data via specific classes.

In fact, most of the data entities that appear in a typical application could be represented with **maps** and **arrays**. But there exist other generic data structures (e.g. sets, lists, queues…) that might be required in some use cases.

Let’s look at the same simplistic example as the one used to illustrate Principle #1: the data that represents an author.

An author is a data entity with a `firstName`, a `lastName` and a number of `books` he/she wrote.

We break Principle #2 when we use a **specific class** to represent an author, like in Listing 0.11.

Listing 0.11 Breaking Principle #2 in OOP

```
class AuthorData {
    constructor(firstName, lastName, books) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.books = books;
    }
}
```

We are **compliant** with Principle #2 when we use a map—which is a **generic** data structure—to represent an author, like in Listing 0.12.

Listing 0.12 Following Principle #2 in OOP

```
function createAuthorData(firstName, lastName, books) {
    var data = new Map;
    data.firstName = firstName;
    data.lastName = lastName;
    data.books = books;
    return data;
}
```

In a language like JavaScript, a map could be instantiated also via a **data literal**>>, which is a bit more convenient. An example is shown in Listing 0.13.

Listing 0.13 Following Principle #2 with map literals

```
function createAuthorData(firstName, lastName, books) {
    return {
        firstName: firstName,
        lastName: lastName,
        books: books
    };
}
```

0.3.3 Benefits of Principle #2

When we use **generic data structures** to represent our data, our programs benefit from:

- Leverage **generic** functions that are not limited to our specific use case
- **Flexible** data model

LEVERAGE FUNCTIONS THAT ARE NOT LIMITED TO OUR SPECIFIC USE CASE

There is a famous quote by Alan Perlis that summarizes this benefit very well:

It is better to have 100 functions operate on one data structure than 10 functions on 10 data structures.

– Alan Perlis

When we use **generic** data structures to represent entities, we have the privilege to manipulate the entities with the **rich set of functions** available on those data structures natively in our programming language in addition to the ones provided by third party libraries.

For instance, JavaScript natively provides some basic functions on maps and arrays and third party libraries like lodash extend the functionality with even more functions.

As an example, when an author is represented as a map, we can **serialize** it into JSON for free, using `JSON.stringify()` which is part of JavaScript, as shown in Listing 0.14.

Listing 0.14 Data serialization comes for free when we adhere to Principle #2

```
var data = createAuthorData("Isaac", "Asimov", 500);
JSON.stringify(data);
```

And if we want to serialize the author data without the number of books, we can use lodash's `_.pick()` function to create an object with a subset of keys. An example is shown in Listing 0.15.

Listing 0.15 Manipulating data with generic functions

```
var data = createAuthorData("Isaac", "Asimov", 500);
var dataWithoutBooks = _.pick(data, ["firstName", "lastName"]);
JSON.stringify(dataWithoutBooks);
```

TIP When you adhere to Principle #2, a rich set of functionalities is available to manipulate your data entities.

FLEXIBLE DATA MODEL

When we use **generic** data structures, our data model is **flexible** in the sense that our data is not forced to adhere to a specific shape. As a consequence, we are **free** to create data with **no predefined shape**. And we are free to **modify** the shape of our data.

In classical OO—*when we don't adhere to Principle #2*—each piece of data is instantiated via a class and must follow a rigid data shape. As a consequence, even when a slightly different data shape is needed, we have to define a new class.

Take for example a class `AuthorData` that represents an author entity that made of 3 fields: `firstName`, `lastName` and `books`. Suppose that you want to add a field `fullName` with the full name of the author.

When you don't adhere to Principle #2, you have to define a new class `AuthorDataWithFullName`.

However when you use generic data structures, you are free to add (or remove) fields to a map *on the fly*, like in Listing 0.16.

Listing 0.16 Adding a field on the fly

```
var data = createAuthorData("Isaac", "Asimov", 500);
data.fullName = "Isaac Asimov";
```

TIP Working with a flexible data model is particularly useful in applications where the shape of the data tends to be very dynamic (e.g. web apps and web services).

In Chapter 3, we will explore in detail the benefits of a flexible data model in the context of a real world application.

0.3.4 Price for Principle #2

There is no such thing as a free lunch. Applying Principle #2 comes at a price.

The price we have to pay when we represent data entities with generic data structures is:

- Slight **Performance** hit
- Data shape needs to be documented **manually**

- No **compile** time check that the data is valid

PRICE #1: PERFORMANCE HIT

When we use **specific** classes to instantiate data, retrieving the value of a class member is super fast. The reason is that the compiler knows upfront how the data is going to look like and it can do all kinds of optimizations.

With **generic** data structures, it is harder to optimize. As a consequence, retrieving the value associated to a key in a map is a bit slower that retrieving the value of a **class** member. Similarly setting the value of an arbitrary key in a map is a bit slower that setting the value of a class member.

TIP **Retrieving and storing the value associated to an arbitrary key from a map is a bit slower than with a class member.**

In most programming languages, this performance hit is not significant, but it is something to keep in mind.

PRICE #2: DATA SHAPE NEEDS TO BE DOCUMENTED MANUALLY

When an object is instantiated from a class, the information of the **data shape** is in the class definition. It is helpful for developers and for IDEs (think about auto-completion features).

TIP **When we use generic data structures to store data, the shape of the data needs to be documented manually.**

Even when we are disciplined enough and we document our code, it may occur that we modify slightly the shape of an entity and we forget to update the documentation.

In that case, we have to explore the code in order to figure out what is the real shape of our data.

In Part 3 of the book, we will explore how DO addresses this issue.

PRICE #3: NO COMPILE TIME CHECK THAT THE DATA IS VALID

Take a look again at the `fullName` function that we created during our exploration of Principle #1:

Listing 0.17 A function that receives the data it manipulates as an argument

```
function fullName(data) {
    return data.firstName + " " + data.lastName;
}
```

When we pass to `fullName` a piece of **data that doesn't conform** to the shape `fullName` expects, an error occurs at **runtime**.

For example, we could **mistype** the field that stores the first name (`fistName` instead of `firstName`), and instead of a compile time error or an exception, we get a weird result where the `firstName` is omitted from the result:

Listing 0.18 Weird behavior when data doesn't conform to the expected shape

```
fullName({fistName: "Issac", lastName: "Asimov"}); // it returns "undefined Asimov"
```

When data is instantiated only via classes with rigid data shape, this type of error is caught at compile time.

TIP **When data is represented with generic data structures, data shape errors are caught only at runtime.**

0.3.5 Wrapping up

DO guides us to use **generic** data structures to represent our data.

This might cause a (small) **performance** hit and forces us to document **manually** the shape of our data as we cannot rely on the compiler to statically validate it.

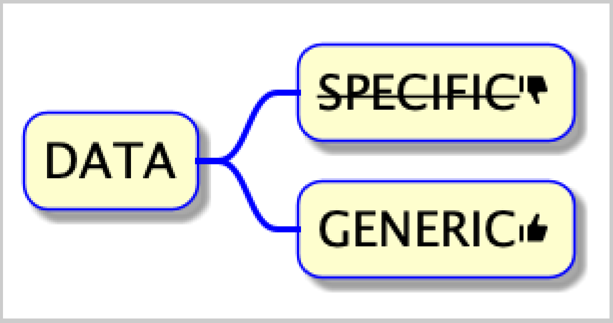
But it worth it because when we adhere to this principle, we can **manipulate** the data entities with a **rich** set of generic functions (provided by the language and by third party libraries) and our data model is **flexible**.

At this point the data could be either **mutable** or **immutable**. The next principle will guide us towards immutability.

SIDEBAR **DO Principle #2: Represent data entities with generic data structures**

The Principle

Represent the data of your application using generic data structures (mostly maps and arrays).



Benefits

- Leverage generic functions that are not limited to our specific use case
- Flexible data model

Price

- Performance hit
- Data shape needs to be documented manually
- No compile time check that the data is valid

0.4 DO Principle #3: Data is immutable

0.4.1 The principle in a nutshell

We are now at a point where our **data** is separated from our **code** and our data is represented with **generic** data structures. Now comes the question of **managing changes** in our data.

DO is very strict about that and doesn't allow any mutations to the data.

NOTE **Principle #3: Data is immutable.**

In DO, we manage changes in our data by creating **new versions** of the data.

Also, we are allowed to **change the reference** of a variable, so that it refers to a new version of the data. What must never change is the value of the data itself.

0.4.2 Illustration of Principle #3

Think about the number 42. What happens to 42 when you add 1 to it? Does it become 43?

No! 42 stays 42 forever!!!

Now put 42 inside an object {num: 42}. What happens to the object when you add 1 to 42? Does it become 43?

It depends on the programming language:

- In Clojure, a programming language that embraces **data immutability**, 42 stays 42 forever, no matter what.
- In many programming languages, 42 becomes 43.

For instance, in JavaScript, mutating the field of a map referred by two variables has an impact on both variables, as shown in Listing 0.19.

Listing 0.19 Mutating data referred by two variables impact both variables

```
var myData = {num: 42};
var yourData = myData;

yourData.num = yourData.num + 1;
```

Now, myData.num equals 43!

According to DO, data should never change. Instead of mutating data, we create a **new version** of it.

A **naive** (and inefficient) way to create a new version of a data is to **clone** it before modifying it.

For instance, in Listing 0.20 there is a function that changes the value of a field inside an object, by cloning the object via Object.assign provided natively by JavaScript. Now, when we call changeValue on myData, myData is not affected: myData.num remains 42. That's the essence of data immutability.

Listing 0.20 Data immutability via cloning

```
function changeValue(obj, k, v) {
    var res = Object.assign({}, obj);
    res[k] = v;
    return res;
}

var myData = {num: 42};
var yourData = changeValue(myData, "num", myData.num + 1);
```

Embracing immutability in an **efficient** way requires a third party library like Immutable.js that provides an efficient implementation of **persistent data structures**.

In most programming languages, there exist libraries that provide an efficient implementation of persistent data structures.

With `Immutable.js`, we don't use JavaScript native maps and arrays but immutable maps and arrays instantiated via `Immutable.Map` and `Immutable.List`. In order to access the element of a map, we use the `get` method and we create a new version of the map where one field is modified, with the `set` method:

Listing 0.21 Creating and manipulating immutable data efficiently with a third-party library

```
var myData = Immutable.Map({num: 42})
var yourData = myData.set("num", 43);
```

`yourData.get("num")` is `43` but `myData.get("num")` remains `42`.

TIP **When data is immutable, instead of mutating data, we create a new version of it.**

0.4.3 Benefits of Principle #3

When we constrain our programs to **never mutate data**, our programs benefit from:

- Data **access** to all with **serenity**
- Code behavior is **predictable**
- **Equality check** is fast
- **Concurrency safety** for free

BENEFIT #1: DATA ACCESS TO ALL WITH SERENITY

According to Principle #1: Separate code from data, data access is **transparent**: Any function is allowed to access any piece of data. Without data **immutability**, we would need to be very **careful** each time we pass data as an argument to a function. We would need to either make sure the function doesn't mutate the data or clone the data before we pass it to the function.

When we adhere to data immutability, none of this is required.

TIP **When data is immutable, we can pass data to any function with serenity, because data never changes.**

BENEFIT #2: CODE BEHAVIOR IS PREDICTABLE

Let me illustrate what I mean by **predictable** by giving first an example of an unpredictable piece of code that doesn't adhere to data immutability.

Please take a look at the following piece of **asynchronous** code in JavaScript.

Listing 0.22 When data is mutable the behavior of asynchronous code is not predictable

```
var myData = {num: 42};
setTimeout(function(data){
    console.log(data.num);
}, 1000, myData)
```

The value of `data.num` inside the timeout callback is not predictable. It depends whether or not the data is modified by another piece of code, during the 1000ms of the timeout.

However, if you constrain yourself to **data immutability**, you are guaranteed that data never changes and you can predict that `data.num` is `42` inside the callback!

TIP When data is immutable, the behavior of code that manipulates data is predictable

BENEFIT #3: EQUALITY CHECK IS FAST

In a UI framework like `React.js`, we frequently check what portion of the "UI data" has been modified since the previous rendering cycle. Portions that didn’t change are not rendered again.

In fact, in a typical **frontend application**, most of the UI data is left **unchanged** between subsequent rendering cycles. In a React application that doesn’t adhere to data immutability, we have no other choice that checking every (nested) part of the UI data.

However in a React application that follows data immutability, we can optimize the comparison of the data for the case where data is not modified. Indeed, when the object address is the same, then we know for sure that the data did not change. Comparing object addresses is much faster than comparing all the fields.

TIP When data is immutable, we benefit from fast equality check by comparing data by reference.

We will see in Chapter 5 how we leverage fast equality check in order to reconcile between **concurrent mutations** in a **highly scalable** production system.

BENEFIT #4: CONCURRENCY SAFETY FOR FREE

In a **multi threaded** environment, we usually use **concurrency safety mechanisms** (e.g. mutexes) to make sure the data is not modified by thread `A` while we access it in thread `B`.

In addition to the slight **performance hit** they cause, concurrency safety mechanisms is a **burden for our minds** as it makes code writing and reading much more difficult.

TIP When we adhere to data immutability, no concurrency mechanism is required: the data you have in hand never changes!

0.4.4 Price for Principle #3

There is no such thing as a free lunch. Applying Principle #3 comes at a price:

- **Performance** hit
- Need a **library** for persistent data structures

PRICE #1: PERFORMANCE HIT

As we mentioned earlier, there exist implementations of persistent data structures in most programming languages. But even the most efficient implementation is be a bit slower than the in-place mutation of the data.

In most applications, the **performance hit** involved in usage of immutable data structures, is not significant. But it is something to keep in mind.

PRICE #2: NEED A LIBRARY FOR PERSISTENT DATA STRUCTURES

In a language like Clojure, the **native** data structures of the language are immutable. However, in most programming languages, adhering to data immutability requires the inclusion a **third party library** that provides an implementation of persistent data structures.

The fact that the data structures are not native to the language means that it is difficult (if not impossible) to **enforce** the usage of immutable data across the board.

Also, when you integrate with other **third party libraries** (e.g. a chart library), you need first to **convert** your persistent data structure into a equivalent native data structure.

0.4.5 Wrapping up

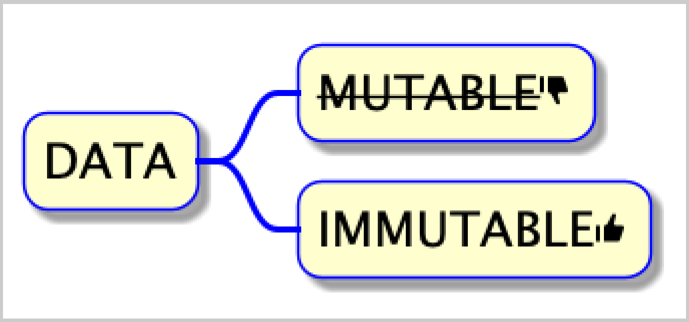
DO considers data as a value that never changes. When you adhere to this principle, your code is **predictable** even in a multi threaded environment and equality check is **fast**.

However, it requires a non negligible mind shift and in most programming languages, you need a **third party library** that provides an efficient implementation of **persistent data structures**.

SIDEBAR **DO Principle #3: Data is immutable**

The Principle

Data is immutable.



Benefits

- Data access to all with serenity
- Code behavior is predictable
- Equality check is fast
- Concurrency safety for free

Price

- Performance hit
- Need a library for persistent data structures

0.5 Conclusion

Data Oriented programming **simplifies** the design and implementation of information systems by treating **data** as a **first class citizen**. This is made possible by adhering to 3 language agnostic core principles:

1. Separate **code** from **data**
2. Represent entities with **generic** data structures
3. Data is **immutable**

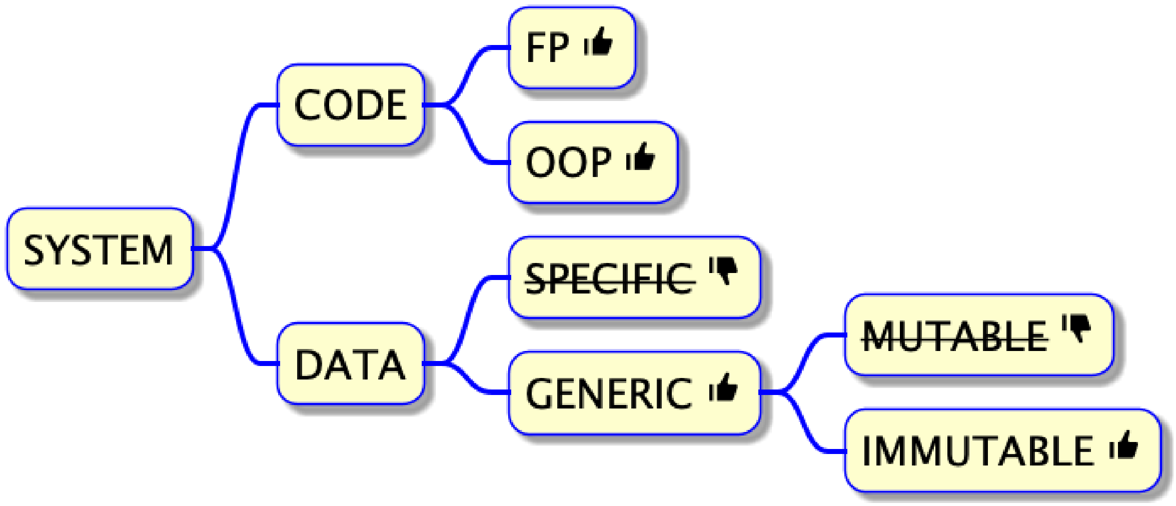


Figure 0.4 The principles of Data Oriented programming

In this chapter we have **illustrated** how each principle can be applied both in **FP** and **OO** languages. We have mentioned at a high level what are the **benefits** of each principle and the **price** it costs to adhere to it.

Throughout the book, we will explore those principles in detail and illustrate how we apply them as a whole in **information systems**.

1 The tendency of Object Oriented Programming towards increased system complexity

1.1 Introduction

In this chapter, we explore the **tendency** of OO systems to be **complex**.

This complexity is not related to the syntax or the semantics of a specific OO language. It is something that is inherent to OO's fundamental insight that programs should be composed from **objects** that consist of some **state** together with **methods** for accessing and manipulating that state.

In this chapter, we illustrate how some fundamental **aspects** of OO tend to increase the **complexity** of a system.

Over the years, OO ecosystems have **alleviated** this complexity increase by adding **new features** to the language (e.g. anonymous classes and anonymous functions) and by developing **frameworks** that hide part of this complexity by providing a simpler interface to the developers (e.g. Spring and Jackson in Java). Internally, they rely on advanced features of the language (like reflection and custom annotations).

This chapter is not meant to be read as a **critics** of OO programming. Its purpose is to raise **awareness** of the **tendency towards increased complexity** of OO as a programming paradigm and to motivate you to discover a **different programming paradigm** where the system complexity tends to be reduced present, namely **Data Oriented programming**.

As we mentioned in Chapter 0, DO principles are language agnostic: if one choose to build a OO system that adheres to DO principles, the system will be less **complex**.

1.2 OO design: classic or classical?

1.2.1 Meeting with a customer

It’s Monday morning 9:00 AM, you seat at a coffee shop with Nancy, a potential customer, that needs you to build a new **library management system**.

YOU: What’s a library management system in your mind?

NANCY: It’s a system that handles **housekeeping functions of a library**, mainly around the book collection and the library members.

YOU: Could you be a little bit more precise?

NANCY: Sure

Nancy grabs the napkin under her coffee mug and she writes down a couple of bullet points on the napkin:

SIDEBAR The requirements for the library management system

- Two kinds of users: library members and librarians
- Users log in to the system via email and password.
- Members can borrow books
- Members and librarians can search books by title or by author
- Librarians can block and unblock members (e.g. when they are late in returning a book)
- Librarians can list the books currently lent by a member
- There could be several copies of a book

YOU: "Well, that’s pretty clear."

NANCY: When will you be able to deliver it?

YOU: If you give me a down payment today, I should be able to deliver it by next Wednesday.

NANCY: Fantastic! I’ll make you a bitcoin transfer later today.

1.2.2 The design phase

You get back to your office with Nancy’s napkin in your pocket.

Before rushing to your laptop to code the system, you grab a sheet of paper—much bigger than the napkin—and you prepare yourself to draw the **UML class diagram** of the system.

You are an OO programmer. For you there is no question: Everything is an object and every object is made from a class.

Here are the main classes that you identify for the library management system:

SIDEBAR **The main classes of the library management system**

- `Library`: **The central part for which the system is designed**
- `Book`: **A book**
- `BookItem`: **A book can have multiple copies, each copy is considered as a book item**
- `BookLending`: **When a book is lent, a book lending object is created**
- `Member`: **A member of the library**
- `Librarian`: **A librarian**
- `User`: **A base class for** `Librarian` **and** `Member`
- `Catalog`: **Contains list of books**
- `Author`: **A book author**

That was the easy part. Now comes the difficult part: the **relationships between the classes**.

After two hours or so, you come up with a first draft of a **design** for the library management system. It looks like the diagram shown on Figure 1.1.

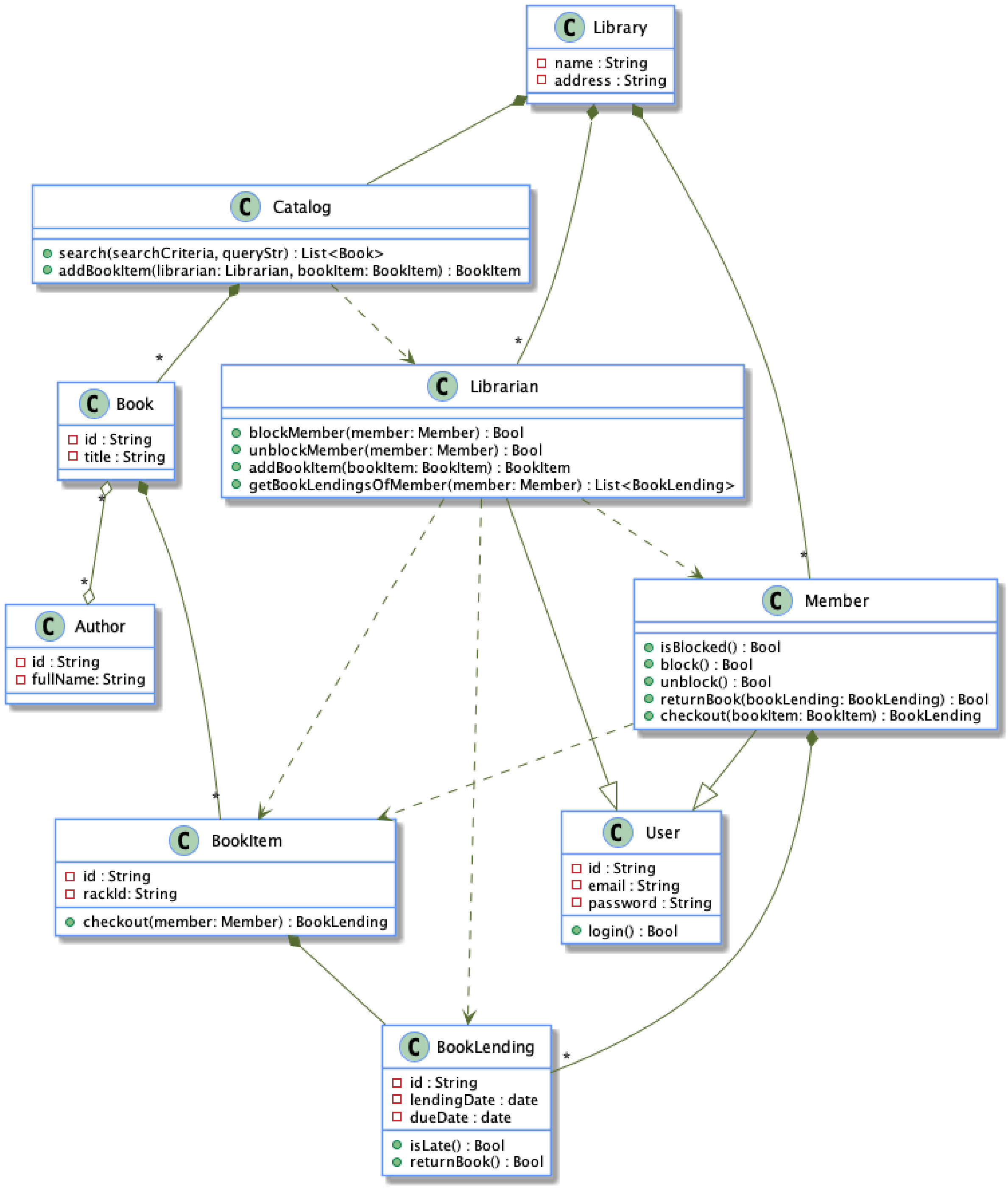

Library
name : String
address : String
Catalog
search(searchCriteria, queryStr) : List<Book>
addBookItem(librarian: Librarian, bookItem: BookItem) : BookItem
Book
id : String
title : String
Librarian
blockMember(member: Member) : Bool
unblockMember(member: Member) : Bool
addBookItem(bookItem: BookItem) : BookItem
getBookLendingsOfMember(member: Member) : List<BookLending>
Author
id : String
fullName: String
Member
isBlocked() : Bool
block() : Bool
unblock() : Bool
returnBook(bookLending: BookLending) : Bool
checkout(bookItem: BookItem) : BookLending
BookItem
id : String
rackId: String
checkout(member: Member) : BookLending
User
id : String
email : String
password : String
login() : Bool
BookLending
id : String
lendingDate : date
dueDate : date
isLate() : Bool
returnBook() : Bool

Figure 1.1 A class diagram for a Library management system

This design is meant to be very naive and by no means it pretends to cover all the features of the system.

WARNING The design presented here doesn't pretend to be the smartest OO design: experienced OO developers would probably leverage a couple of design patterns and suggest a much better design.

This design serves two purposes:

1. For you - *the developer* - it is rich enough to start coding
2. For me - *the author of the book* - it is rich enough to illustrate the complexity of a typical OO system

Anyway, you feel proud of yourself and of the design you produced. You definitely deserve a cup of coffee. Near the coffee machine, you meet Dave, a **junior software developer** that you appreciate.

YOU: Hey Dave! How are you doing?

DAVE: Trying to fix a bug in my code: I cannot understand why the state of my objects always change! You?

YOU: I have just finished the design of a system for a new customer.

DAVE: Cool! Can you show me your design?

YOU: Sure.

1.2.3 UML 101

Dave follows you to your desk and you show him your piece of art: the UML diagram for the library management system in Figure 1.1.

Dave seems really excited.

DAVE: Wow! Such a detailed class diagram.

YOU: Yeah. It’s pretty neat.

DAVE: The thing is that I can never remember the **meaning of each arrow**.

YOU: There are 4 types of arrows in my class diagram: **composition**, **association**, **inheritance** and **usage**.

DAVE: Whats the difference between composition and association?

You google "composition vs association" and you read loudly to Dave:

YOU: Its all about whether the objects can live one without each other: with **composition**, when one object dies, the other one dies also, while in an **association** relation, each object has an independent life.

TIP In a composition relation, when one object dies, the other one dies also, while in an association relation, each object has an independent life cycle.

In the class diagram, there are two kinds of **composition** relation, symbolized by an arrow with a **plain diamond** at one edge, and an optional **star** at the other edge:

1. A `Library` owns a `Catalog`: That's a **one-to-one composition** relation: if a `Library` object dies, then its `Catalog` object dies with it.
2. A `Library` owns many `Members`: That's a **one-to-many composition** relation: if a `Library` object dies, then all its `Member` objects die with it.

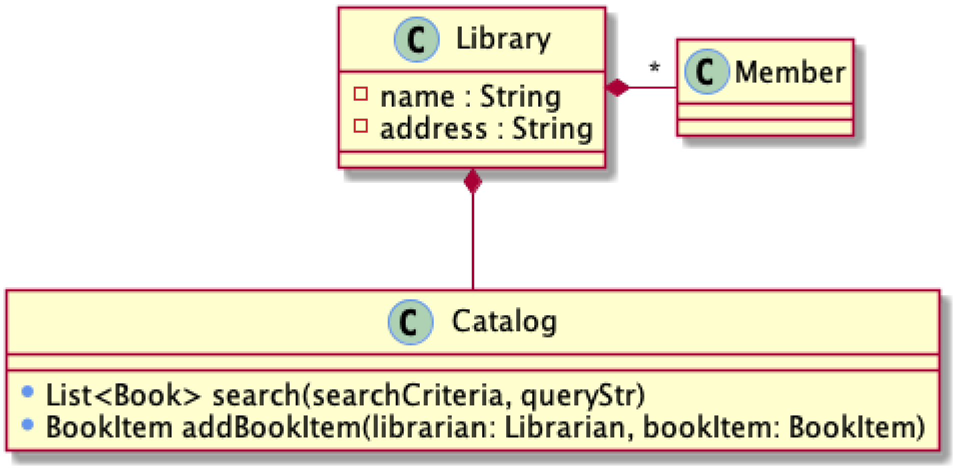

C Library
name : String
address : String
*
C Member
C Catalog
List<Book> search(searchCriteria, queryStr)
BookItem addBookItem(librarian: Librarian, bookItem: BookItem)

Figure 1.2 Two kinds of composition: one-to-one and one-to-many. In both cases, when an object dies, the composed object dies with it.

TIP A composition relation is represented by a plain diamond at one edge and an optional star at the other edge.

DAVE: Do you have association relations in your diagram?

YOU: Take a look at the arrow between `Book` and `Author`. It has an **empty diamond** and a **star** at both edges: it's a **many to many association** relation.

A book can be written by **multiple** authors and an author can write **multiple** books. Moreover, `Book` and `Author` objects can live independently: the relation between books and authors is a **many-to-many association** relation.

TIP A many-to-many association relation is represented by an empty diamond and a star at both edges.

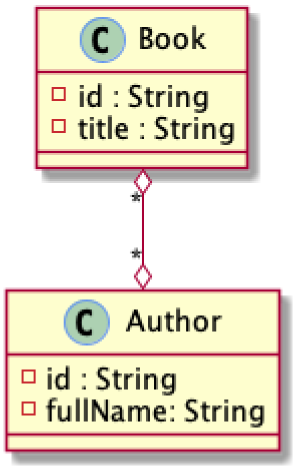


Figure 1.3 Many to many association relation: each object lives independently

DAVE: I see also many **dashed arrows** in your diagram.

YOU: Dashed arrows are for **usage** relations: when a class uses a method of another class. Consider for example, at the `Librarian::blockMember()` method. It calls `Member::block()`.

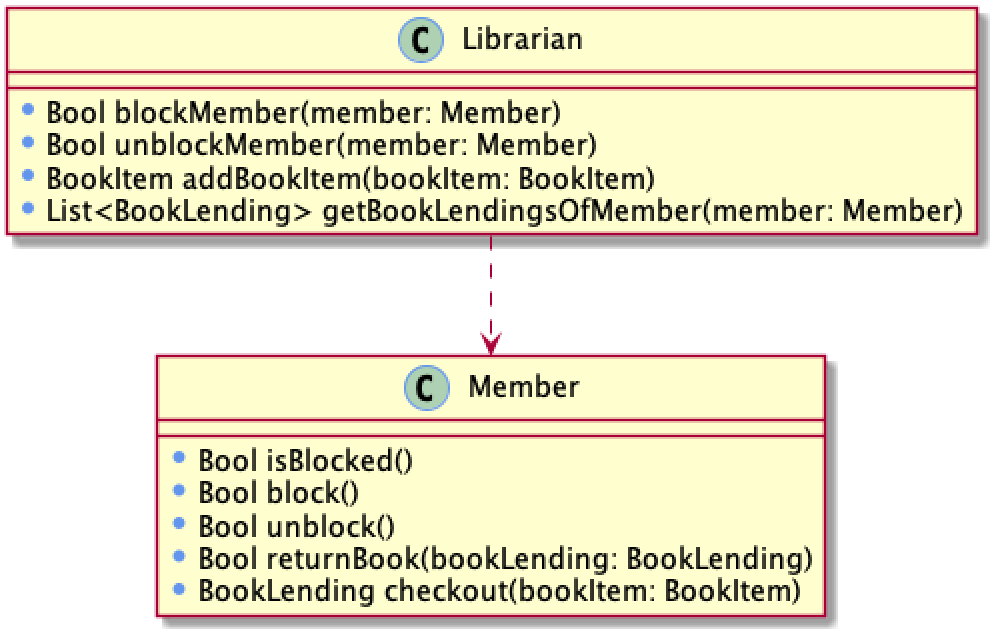

Librarian
Bool blockMember(member: Member)
Bool unblockMember(member: Member)
BookItem addBookItem(bookItem: BookItem)
List<BookLending> getBookLendingsOfMember(member: Member)
Member
Bool isBlocked()
Bool block()
Bool unblock()
Bool returnBook(bookLending: BookLending)
BookLending checkout(bookItem: BookItem)

Figure 1.4 Usage relation: a class uses a method of another class

TIP	Dashed arrows are for usage relations: for instance, when a class uses a method of another class.

DAVE: I see. And I guess that **plain arrows with empty triangle**—like the one between `Member` and `User`—represent **inheritance**.

YOU: Absolutely.

TIP Plain arrows with empty triangle represents class inheritance.

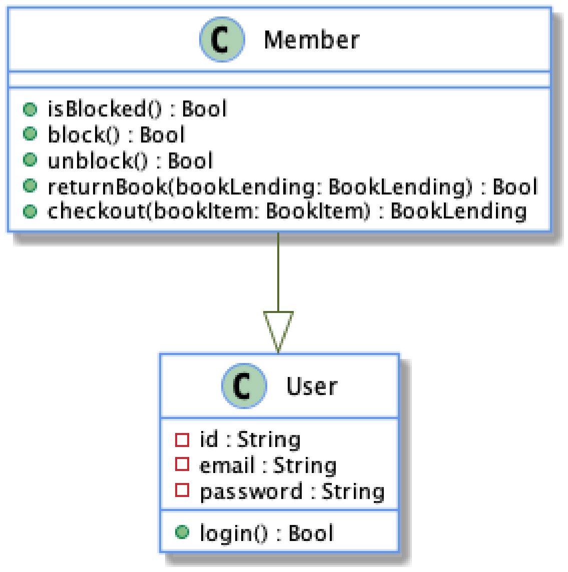


Figure 1.5 Inheritance relation: a class derives from another class

1.2.4 Explaining each piece of the class diagram

DAVE: Thank you for this short UML course. Now I understand the **meaning** of each kind of **arrow** in your diagram.

YOU: My pleasure.

DAVE: What class should I look at first?

YOU: I think you should start from `Library`.

THE LIBRARY CLASS

The `Library` is the root class of the system.

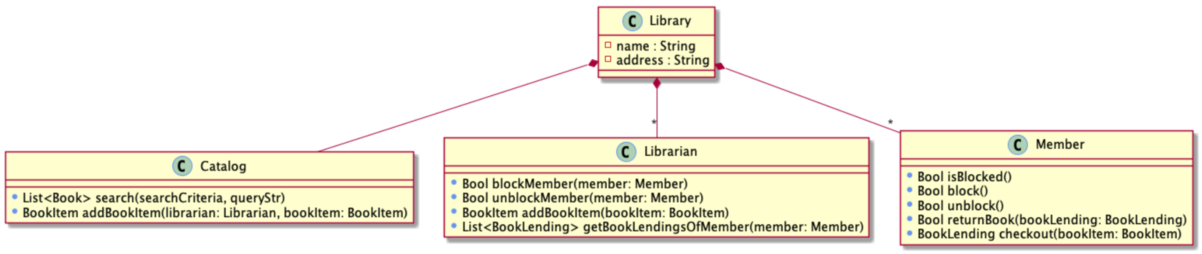

C Library
name : String
address : String
*
*
C Catalog
List<Book> search(searchCriteria, queryStr)
BookItem addBookItem(librarian: Librarian, bookItem: BookItem)
C Librarian
Bool blockMember(member: Member)
Bool unblockMember(member: Member)
BookItem addBookItem(bookItem: BookItem)
List<BookLending> getBookLendingsOfMember(member: Member)
C Member
Bool isBlocked()
Bool block()
Bool unblock()
Bool returnBook(bookLending: BookLending)
BookLending checkout(bookItem: BookItem)

Figure 1.6 The Library class

In terms of **code**, a `Library` object does nothing on its own, it delegates everything to objects it owns.

In terms of **data**, a `Library` object owns:

1. Multiple `Member` objects
2. Multiple `Librarian` objects
3. A single `Catalog` object

LIBRARIAN, MEMBER AND USER CLASSES

`Librarian` and `Member` who both derive from `User`.

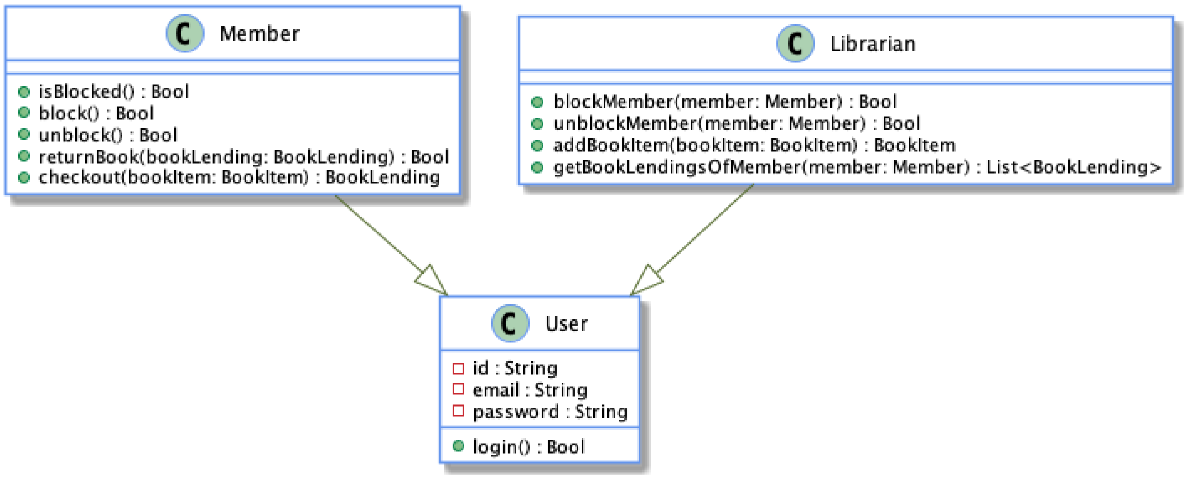

Member
isBlocked() : Bool
block() : Bool
unblock() : Bool
returnBook(bookLending: BookLending) : Bool
checkout(bookItem: BookItem) : BookLending
Librarian
blockMember(member: Member) : Bool
unblockMember(member: Member) : Bool
addBookItem(bookItem: BookItem) : BookItem
getBookLendingsOfMember(member: Member) : List<BookLending>
User
id : String
email : String
password : String
login() : Bool

Figure 1.7 Librarian and Member derive from User

The `User` class represents a user of the library.

1. In terms of **data members**, it sticks to the bare minimum: it has a `id`, `email` and `password` (no with security and encryption for now).
2. In terms of **code**, it can login via `login()`

The `Member` class represents a member of the library.

1. It **inherits** from `User`
2. In terms of data members, it has nothing more than `User`
3. In terms of code, it can:
   A. Checkout a book via `checkout()`
   B. Return a book via `returnBook()`
   C. Block itself via `block()`
   D. Unblock itself via `unblock()`
   E. Answer if it is blocked via `isBlocked()`
4. It owns multiple `BookLending` objects
5. It uses `BookItem` in order to implement `checkout()`

The `Librarian` class represents a librarian.

1. It derives from `User`
2. In terms of data members, it has nothing more than `User`
3. In terms of code, it can:
   A. Block and unblock a `Member`
   B. List the book lendings of a member via `getBookLendings()`
   C. Add book items to the library via `addBookItem()`
4. It uses `Member` in order to implement `blockMember()`, `unblockMember()` and `getBookLendings()`
5. It uses `BookItem` in order to implement `checkout()`
6. It uses `BookLending` in order to implement `getBookLendings()`

THE CATALOG CLASS

The `Catalog` class is responsible for the management of the books.

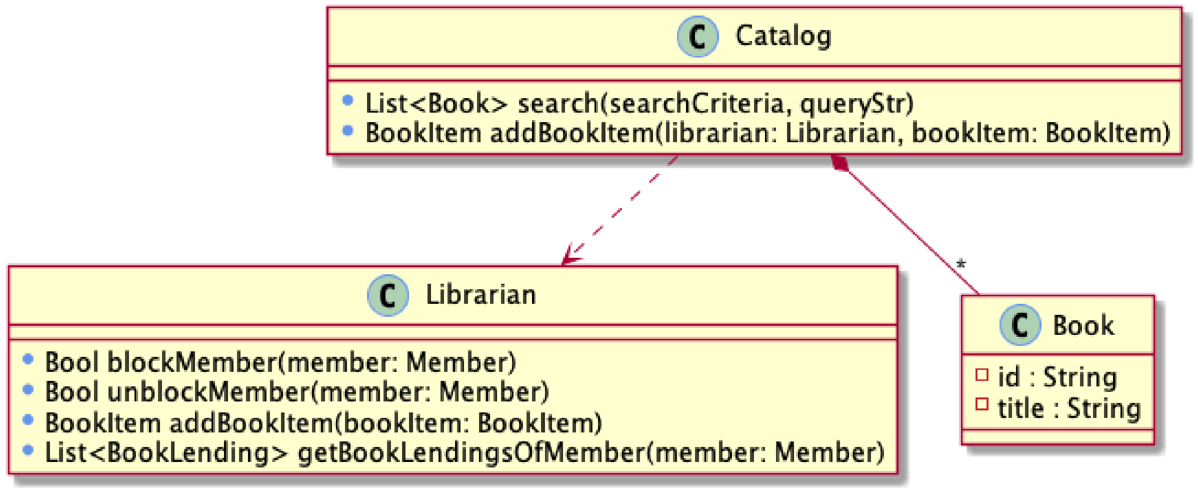

C Catalog
List<Book> search(searchCriteria, queryStr)
BookItem addBookItem(librarian: Librarian, bookItem: BookItem)
C Librarian
Bool blockMember(member: Member)
Bool unblockMember(member: Member)
BookItem addBookItem(bookItem: BookItem)
List<BookLending> getBookLendingsOfMember(member: Member)
*
C Book
id : String
title : String

Figure 1.8 The Catalog class

In terms of **code**, a `Catalog` object can:

1. Search books via `search()`
2. Add book items to the library via `addBookItem()`
   A. It uses `Librarian` in order to implement `addBookItem`

In terms of **data**, a `Catalog` owns:

1. Multiple `Book` objects

THE BOOK CLASS

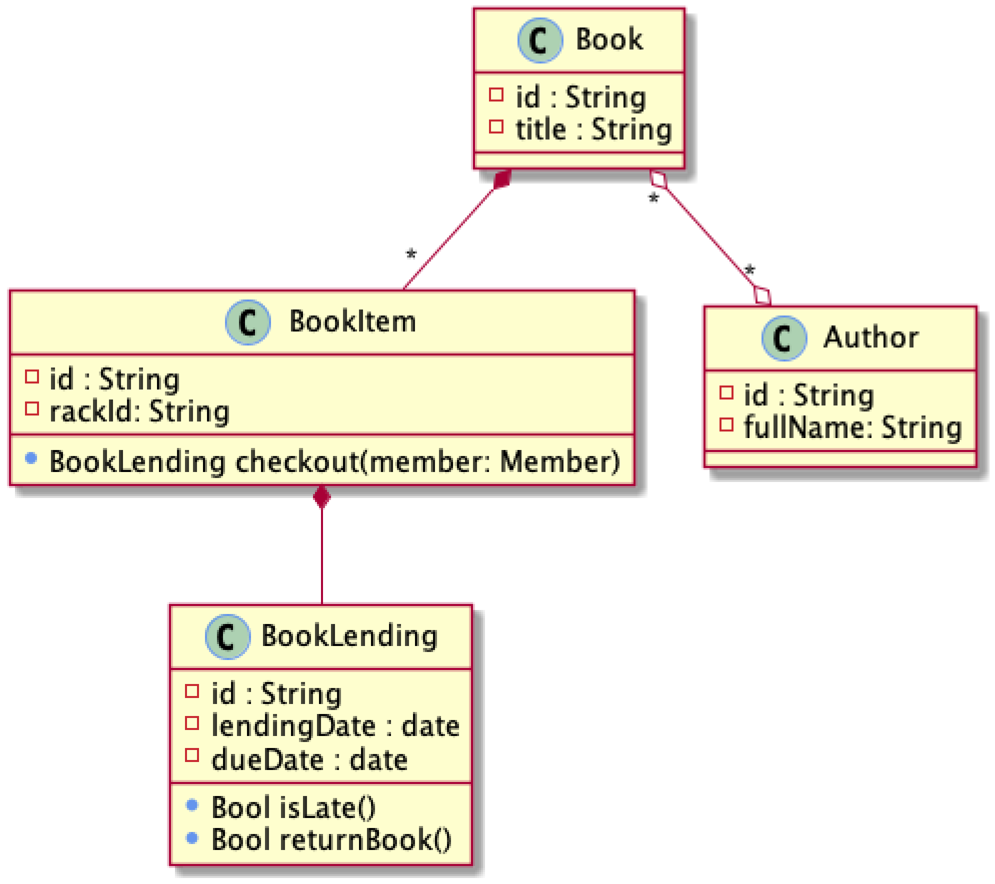


Figure 1.9 The Book class

In terms of **data** a `Book` object:

1. In terms data members, we stick to the bare minimum: it has a `id`, and a `title`
2. It is associated with multiple `Author` objects (A book might have multiple authors)
3. It owns multiple `BookItem` objects, one for each copy of the book

THE BOOKITEM CLASS

The `BookItem` class represents a book copy. A book could have many copies.

In terms of **data** a `BookItem` object:

1. In terms data members, we stick to the bare minimum: it has a `id`, and a `rackId` (for its physical location in the library)
2. It owns multiple `BookLending` objects, one for each time the book is lent

In terms of **code**:

1. It can be checked out via `checkout()`

1.2.5 The implementation phase

After this detailed investigation of your diagram, Dave compliments you.

DAVE: Wow! That's amazing.

YOU: Thank you.

DAVE: I didn't know people were really spending time to write down their design in such details, before coding.

YOU: I always do that. It saves me lot of time during the coding phase.

DAVE: When will you start coding?

YOU: When I finish my coffee.

You look at your coffee mug and it is full (and cold). You were so excited to show your class diagram to Dave that you forgot to drink your coffee.

1.3 Sources of complexity

Before you start coding, and while you prepare yourself another cup of coffee, I would like to **challenge** your design. It might look beautiful and clear on the paper but I am going to claim that this design is **too complex**.

It's not that you picked the wrong classes or that you misunderstood the relationships between the classes. It's much **deeper**. It's about the programming paradigm you chose to implement the system. It's about the Object Oriented paradigm. It's about the **tendency** of OO to increase the **complexity** of a system.

TIP	OO has a tendency to create complex systems.

Like we mentioned in Chapter 0, the type of **complexity** I refer to is the one which makes systems **hard to understand** as it is defined in the beautiful paper Out of the Tar Pit. It has nothing to do with the complexity of the resources consumed by a program.

Similarly, when I refer to **simplicity**, I mean *not complex*, in other words: *easy to understand*.

Keep in mind that complexity and simplicity (like hard and easy) are not absolute but **relative concepts**. We can compare the complexity of two systems and argue that system A is more complex (or simpler) than system B.

NOTE **Complex in the context of this book means: hard to understand**

As we mentioned in the introduction of this chapter, there are many ways in OO to alleviate the complexity. The purpose of this book is not be a critics of OO. The purpose is to present a **programming paradigm** called Data Oriented Programming (DO), that **tends** to build systems that are less **complex**. In fact, the DO paradigm is **compatible** with OO and if one choose to build a OO system that adheres to DO principles, the system will be less **complex**.

TIP **DO is compatible with OO.**

According to DO, the main **sources of complexity** of your system—and of many OO systems—are:

1. Code and data are **mixed**
2. Objects are **mutable**
3. Data is **locked** in objects as members
4. Code is **locked** into classes as methods

In the remaining sections of this chapter, we are going to illustrate each of the above aspects—summarized in Table 1.1—in the context of the library management system and explain in what sense it is a source of complexity.

Table 1.1 Aspects of Object Oriented programming and their impact on increased system complexity

Aspect	Impact on increased complexity
Code and data are mixed	Classes tend to be involved in many relations
Objects are mutable	Extra thinking when reading code
Objects are mutable	Explicit synchronization on multi-threaded environments
Data is locked in objects	Data serialization is not trivial
Code is locked in classes	Class hierarchies are complex

1.4 When code and data are mixed, classes tend to be involved in many relations

One way to assess the **complexity** of a class diagram is to look only at the entities and their **relationships** (ignoring members and methods) as in Figure 0.2.

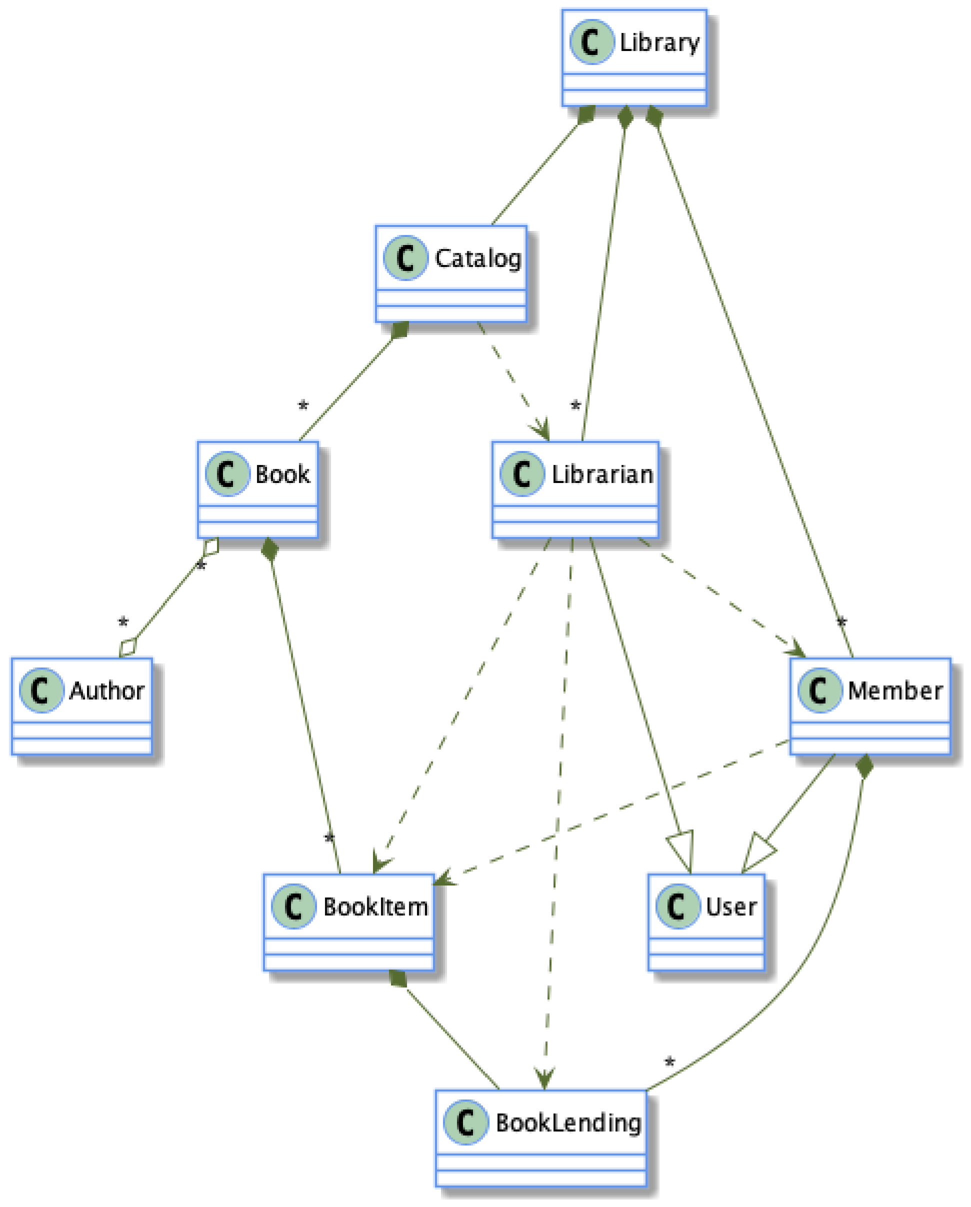


Figure 1.10 A class diagram overview for a Library management system

When we **design** a system, we have to define the **relationships** between different pieces of **code** and **data**: that’s unavoidable.

TIP	**In OO, code and data are mixed together in classes, data as members and code as methods.**

From a **system analysis** perspective, the fact that code and data are mixed together makes the system complex in the sense that entities tend to be **involved** in many **relationships**.

In Figure 1.11, we take a closer look at the `Member` class. `Member` is involved 5 relations: 2 **data** relations and 3 **code** relations.

Data relations:

1. `Library` *has* many `Members`
2. `Member` *has* many `BookLendings`

Code relations:

1. `Member` *extends* `User`
2. `Librarian` *uses* `Member`
3. `Member` *uses* `BookItem`

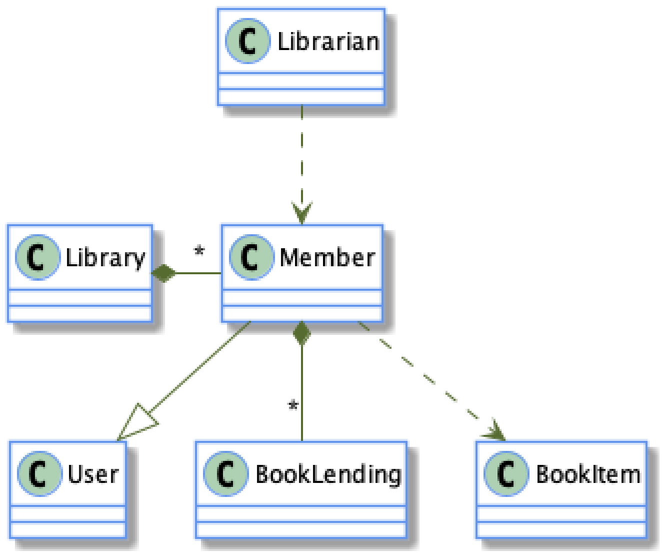


Figure 1.11 The Member is involved in 5 relations

Imagine for a moment that we were able somehow to **split** the `Member` class into two **separate** entitites:

- `MemberCode` for the **code**
- `MemberData` for the **data**

Instead of a `Member` class with 5 relations, we would have the diagram shown Figure 1.12, with:

- A `MemberCode` entity with 3 relations
- A `MemberData` entity with 2 relations

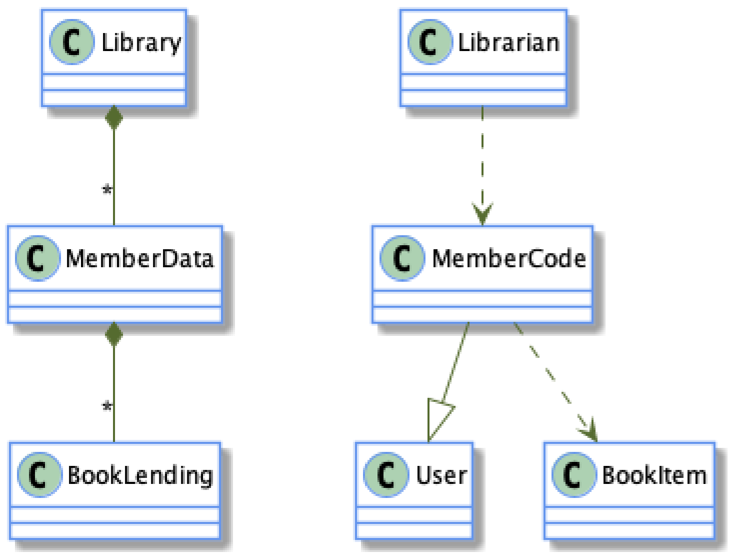


Figure 1.12 A class diagram where Member is split into code and data entities

The class diagram where `Member` is split into `MemberCode` and `MemberData` is made of two disconnected parts, where each part is easier to understand than the original diagram.

Now, let’s **split every class of our original class diagram into code and data entities.** The resulting diagram is shown in Figure 0.3: Now, the system is made of two **disconnected** parts:

1. A part that involves **only code** entities
2. A part that involves **only data** entities

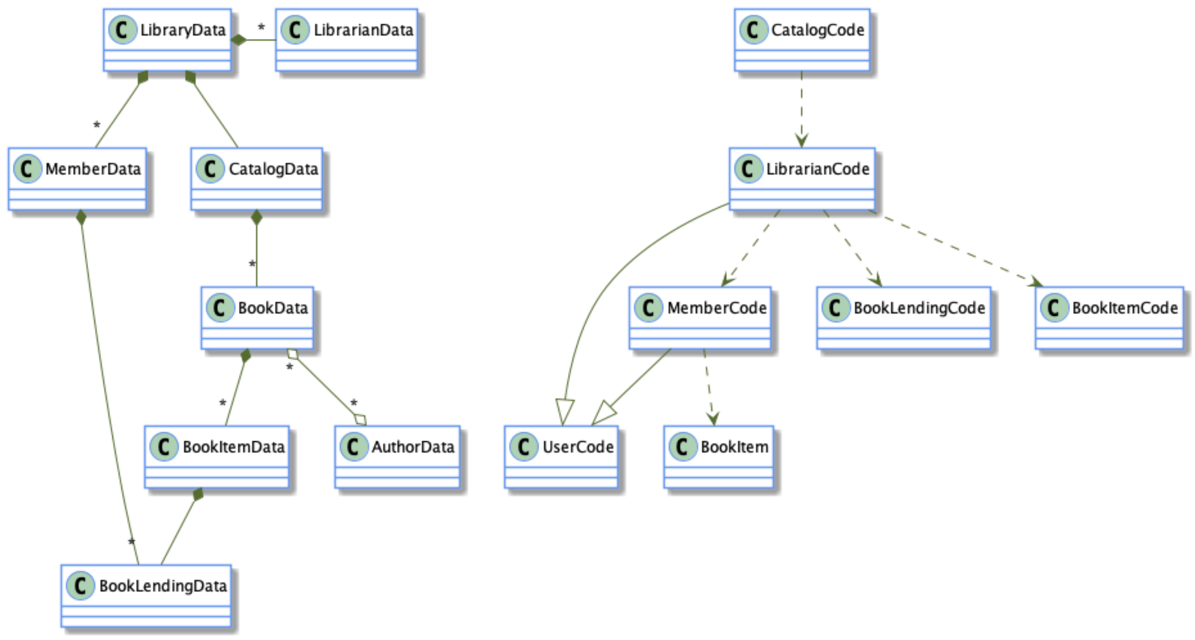


Figure 1.13 A class diagram where every class is split into code and data entities

TIP A system where every class is split into code and data is made of two disconnected parts where each part is simpler than a system where code and data are mixed.

The resulting system—made of two **disconnected** sub-systems—is **easier to understand** than the original system. The fact that the two sub-systems are **disconnected** means that each sub-system can be **understood separately**. We can first understand the **data** part of the system and then the **code** part of the system (or the opposite).

The resulting system not simpler by *accident*, it is a *logical consequence* of separating **code** from **data**.

TIP A system made of disconnected simple parts is less complex than a system made of a single complex part.

1.5 When objects are mutable, understanding code requires extra thinking

You might be a bit tired after the system-level analysis that we presented in the previous section.

Let's get refreshed and look at some code.

Please take a look at the code shown in Listing 1.1: we get the blocked status of a member and we display it twice.

If I tell you that when I called `displayBlockedStatusTwice`, the program displayed `true` on the first `console.log()` call, can you tell me what the program displayed on the second `console.log()` call?

Listing 1.1 Really simple code

```
class Member {
    isBlocked = false;

    displayBlockedStatusTwice() {
        var isBlocked = this.isBlocked;
        console.log(isBlocked);
        console.log(isBlocked);
    }
}

var member = new Member();
member.displayBlockedStatusTwice();
```

"Of course, it displayed `true` again", you tell me.

And you are right.

Now, please take a look at a slightly different pseudocode as shown in Listing 1.2: here we display twice the blocked status of a member without assigning a variable.

Same question as before: If I tell you that when I called `displayBlockedStatusTwice`, the program displayed `true` on the first `consol.log()` call, can you tell me what the program displayed on the second `consol.log()` call?

Listing 1.2 Apparently simple code

```
class Member {
    isBlocked = false;

    displayBlockedStatusTwice() {
        console.log(this.isBlocked);
        console.log(this.isBlocked);
    }
}

var member = new Member();
member.displayBlockedStatusTwice();
```

The correct answer is: in a **single threaded** environment, it displays `true` while on a **multi threaded** environment it's **unpredictable**.

Indeed, in a multi threaded environment, between the two `console.log()` calls, there could be a **context switch** and the **state** of the object could be changed (e.g. a librarian unblocked the member).

Actually, as we showed in Chapter 0, with a slight modification, the same kind of code unpredictability could occur even in a **single threaded** environment like JavaScript, when a data is modified via asynchronous code.

The difference between the two code snippets is that:

- In the first snippet, we access twice a boolean value which is a **primitive** value
- In the second snippet, we access twice a **member** of an object

TIP When data is mutable, code is unpredictable.

This **unpredictable** behavior of the second snippet is one of the annoying consequences of the fact that in OO, unlike primitive types who are usually **immutable**, object members are **mutable** .

One way to solve this problem in OO is to protect sensitive code with **concurrency safety** mechanism like **mutexes**, but it introduces issues on its own like a performance hit and a risk of deadlocks.

We will see later in the book that DO treats every piece of data in the same way: both primitive types and collection types are **immutable** values. This "value treatment for all citizens" brings to DO developers' minds a lot of **serenity**. As a consequence, more cells of DO developers' minds are available to handle the interesting pieces of the applications they build.

TIP Data immutability brings to DO developers' minds a lot of serenity.

1.6 When data is locked in objects as members, data serialization is not trivial

Now, you are really tired and you fall asleep at your desk…

You have a **dream** about Nancy, your customer.

In this dream, Nancy asks you to make the library management system accessible via a REST API using **JSON** as a transport layer.

You need to implement a `/search` **endpoint** that receives a query in JSON format and return results in JSON format.

An **input** example of the `/search` endpoint is shown in Listing 1.3.

Listing 1.3 A JSON input of the `/search` endpoint

```
{
    "searchCriteria": "author",
    "query": "albert"
}
```

An **output** example of the `/search` endpoint is shown in Listing 1.4.

Listing 1.4 A JSON output of the `/search` endpoint

```
[
    {
        "title": "The world as I see it",
        "authors": [
            {
                "fullName": "Albert Einstein"
            }
        ]
    },
    {
        "title": "The Stranger",
        "authors": [
            {
                "fullName": "Albert Camus"
            }
        ]
    }
]
```

You would probably implement the `/search` endpoint by creating three classes similarly to what is shown in Figure 1.14 (Not surprising: everything in OO has to be wrapped in a class. Right?):

1. `SearchController` that is responsible for **handling** the query
2. `SearchQuery` that **converts** the JSON query string into **data**
3. `SearchResult` that **converts** the search result **data** into a JSON string

The `SearchController` would have a single `handle` method with the following flow:

1. Create a `SearchQuery` object from the JSON query string
2. Retrieve `searchCriteria` and `queryStr` from the `SearchQuery` object
3. Call the `search` method of the `catalog:Catalog` with `searchCriteria` and `queryStr` and receives `books:List<Book>`
4. Create a `SearchResult` object with `books`
5. Convert the `SearchResult` object to a `JSON` string

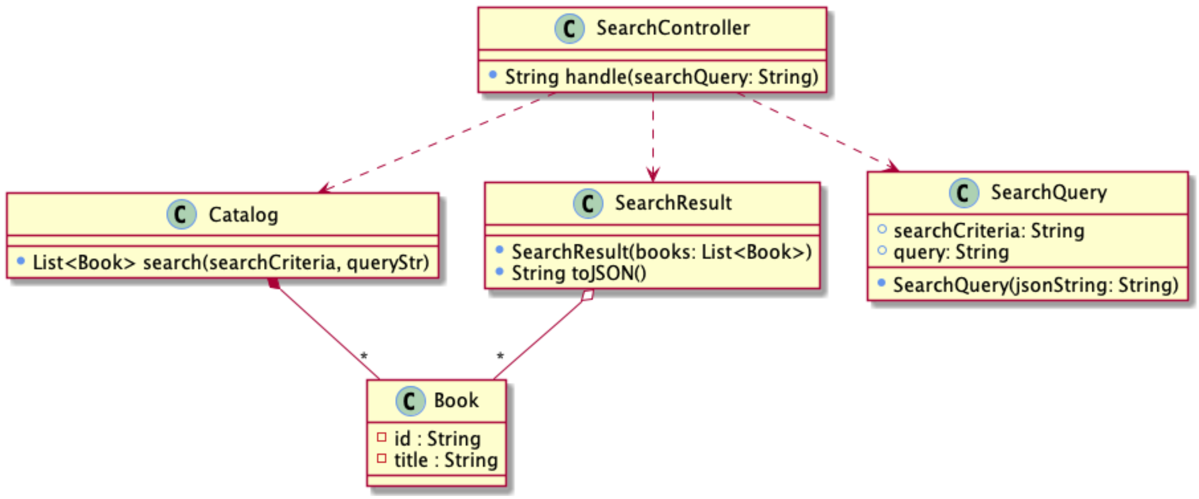

SearchController
String handle(searchQuery: String)
Catalog
List<Book> search(searchCriteria, queryStr)
SearchResult
SearchResult(books: List<Book>)
String toJSON()
SearchQuery
searchCriteria: String
query: String
SearchQuery(jsonString: String)
Book
id : String
title : String
*
*

Figure 1.14 A class diagram where every class is split into code and data entities

What about other endpoints, for instance allowing librarians to add book items through `/add-book-item`?

Well, you would have to **repeat the exact same process** and create 3 classes:

1. `AddBookItemController` that is responsible for **handling** the query
2. `BookItemQuery` that **converts** the JSON query string into **data**
3. `BookItemResult` that **converts** the search result **data** into a JSON string

The code that deals with JSON **deserialization** that you wrote previously in `SearchQuery` would have to be **rewritten** in `BookItemQuery`. Same thing for the code that deals with JSON **serialization** that you wrote previously in `SearchResult`: it would have to be **rewritten** in `BookItemResult`.

The bad news is that you would have to repeat the same process for every endpoint of the system. Each time you encounter a new kind of JSON input or input, you have to create a new class and write code.

Suddenly, you wake up and realize that Nancy never asked for JSON. All of the above was a dream, a really bad dream…

TIP	In OO, data serialization is a nightmare

It's quite **frustrating** that handling JSON **serialization** and **deserialization** in OO requires to add so many classes and to write so much code again and again!

The frustration gets bigger when you consider that serializing a search query, a book item query or any query is quite **similar**. It comes down to:

1. Go over data fields
2. Concatenate the name of the data fields and the value of the data field (separated by a comma)

Why such a simple thing is so hard to achieve in OO?

The thing is that in OO, data has to follow a rigid shape (defined in classes), which means that data is locked in members. There is no way to access data generically.

TIP	In OO, data is locked in classes as members

We will **refine** later what we mean by a **generic** access to the data and we will see how DO provides a generic way to handle JSON **serialization** and **deserialization**. Until then, you will have to continue suffering. But at least you are **aware** of this suffering and you know that this suffering is **avoidable**.

WARNING	Most OO programming languages alleviate a bit the difficulty involved the conversion from and to JSON. It either involves reflection (which is definitely a complex thing) or code verbosity.

1.7 When code is locked into classes, class hierarchies are complex

One way to avoid writing the same code twice in OO involves class **inheritance**. Indeed, when every requirement of the system is **known up front**, you design your class hierarchy is such a way that classes with common behavior derive from a base class.

An example of this pattern is shown in Figure 1.15, that focuses in the part of our class diagram that deals with members and librarians. Both `Librarians` and `Members` need the ability to login and they **inherit** this ability form the `User` class.

So far so good.

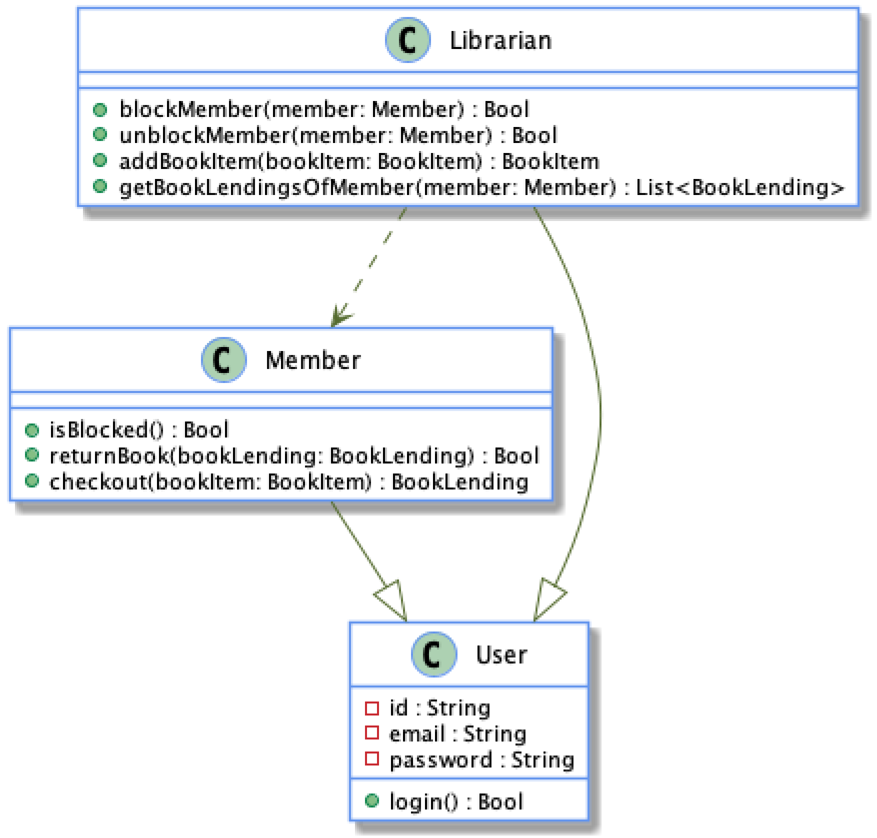

Librarian
blockMember(member: Member) : Bool
unblockMember(member: Member) : Bool
addBookItem(bookItem: BookItem) : BookItem
getBookLendingsOfMember(member: Member) : List<BookLending>
Member
isBlocked() : Bool
returnBook(bookLending: BookLending) : Bool
checkout(bookItem: BookItem) : BookLending
User
id : String
email : String
password : String
login() : Bool

Figure 1.15 The part of the class diagram that deals with members and librarians

But when requirements to the system are added **after the system is implemented** that's a completely different story.

It's Monday 11:00 AM, **two days** are left before the **deadline** (which is on Wednesday midnight) and Nancy put your on an urgent phone call.

You are not sure if it's dream or reality. You pinch yourself and you feel the jolt. It's definitely **reality**!

NANCY: How is the project doing?

YOU: Fine, Nancy. We are **on schedule** for the deadline. Running our last round of **regression tests**.

NANCY: Fantastic! It means we have time for adding a **tiny** feature to the system. Right?

YOU: Depends what you mean by *tiny*.

NANCY: We need to add **VIP members** to the system.

YOU: What do you mean by VIP members?

NANCY: VIP members are members that are allowed to **add by themselves** book items to the library.

YOU: Hmm…

NANCY: What?

YOU: That's not a tiny change!

NANCY: Why?

I am asking you the same question Nancy asked: Why adding **VIP members** to your system is not a **tiny** task?

After all, you **already** have written the code that allows librarians to add book items to the library: it's in `Librarian::addBookItem()`.

What prevents you from reusing this code for VIP members?

The reason is that in OO, the code is locked into classes as methods.

TIP **In OO, code is locked into classes.**

Let's see how you would probably handle this last minute request from your customer.

VIP members are members that are allowed to add by themselves book items to the library.

Let's decompose the customer requirements into two pieces:

1. VIP members are members
2. VIP members are allowed to add by themselves book items to the library

For sure, you need a new class `VIPMember`.

For requirement #1, it sounds reasonable to make `VIPMember` derive from `Member`.

However, handling requirement #2 is more complex. We cannot make `VIPMember` derive from `Librarian` because the relationship between `VIPMember` and `Librarian` is not linear:

1. On one hand, VIP members are **like librarians** as they are **allowed** to add book items
2. On the other hand, VIP members are **not like librarians** as they are **not allowed** to block members or to list the book lendings of a member

The problem is that the code that adds book items is locked in the `Librarian` class. There is no way for `VIPMember` class to use this code.

One possible solution that makes the code of `Librarian::addBookItem()` available to both `Librarian` and `VIPMember`, is shown in Figure 1.16. Here are the changes to the previous class diagram:

1. A base class `UserWithBookItemRight` that extends `User`
2. Move `addBookItem()` from `Librarian` to `UserWithBookItemRight`
3. Both `VIPMember` and `Librarian` extend `UserWithBookItemRight`

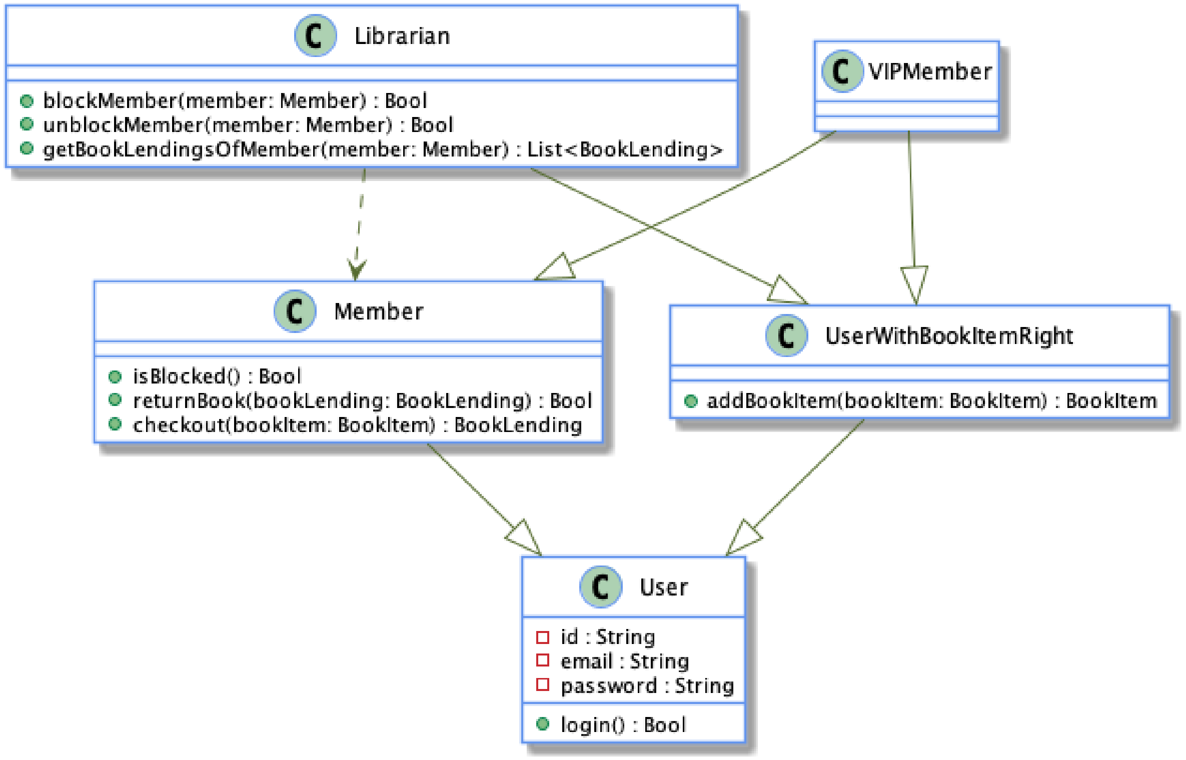

C Librarian
blockMember(member: Member) : Bool
unblockMember(member: Member) : Bool
getBookLendingsOfMember(member: Member) : List<BookLending>
C VIPMember
C Member
isBlocked() : Bool
returnBook(bookLending: BookLending) : Bool
checkout(bookItem: BookItem) : BookLending
C UserWithBookItemRight
addBookItem(bookItem: BookItem) : BookItem
C User
id : String
email : String
password : String
login() : Bool

Figure 1.16 A class diagram for a system with VIP members

That was **tough** but you were able to handle it **on time** (thanks to a white night in front of your laptop). You were even able to include new tests to the system and running again the regression tests.

You were so excited that you didn’t pay attention to the **diamond** `VIPMember` introduced in your class diagram, (`VIPMember` extends both `Member` and `UserWithBookItemRight` who both extend `User`)

We are Wednesday morning 10:00 AM, 14 hours before the deadline and you call Nancy to tell her the good news:

YOU: We were able to add VIP members to the system on time, Nancy.

Fantastic! I told you it was a **tiny** feature.

YOU: Hmm…

NANCY: Look, I was to call you anyway. I just finished a meeting with my **business partner** and we realized that we need another **tiny** feature before the launch. Will you be able to handle it before the deadline?

YOU: Again, it depends what you mean by *tiny*.

NANCY: We need to add **Super members** to the system.

YOU: What do you mean by Super members?

NANCY: Super members are members that are allowed to **block** and **unblock** members

YOU: Hmm…

NANCY: What?

YOU: That's not a tiny change!

NANCY: Why?

Like with VIP members, adding Super members to the system requires **changes to your class hierarchy**. A possible solution is shown in Figure 1.17.

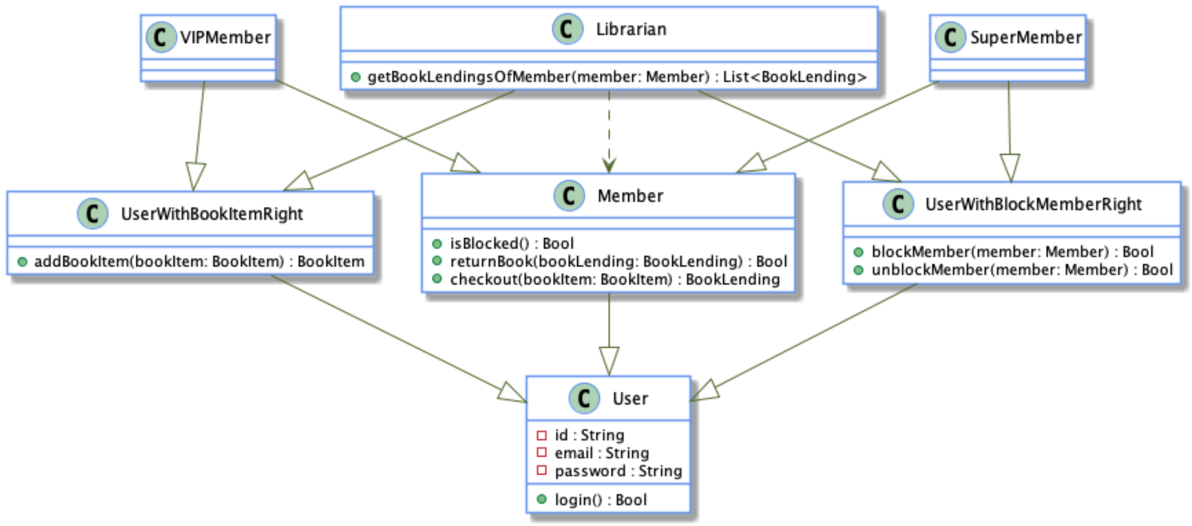

C VIPMember
C Librarian
getBookLendingsOfMember(member: Member) : List<BookLending>
C SuperMember
C UserWithBookItemRight
addBookItem(bookItem: BookItem) : BookItem
C Member
isBlocked() : Bool
returnBook(bookLending: BookLending) : Bool
checkout(bookItem: BookItem) : BookLending
C UserWithBlockMemberRight
blockMember(member: Member) : Bool
unblockMember(member: Member) : Bool
C User
id : String
email : String
password : String
login() : Bool

Figure 1.17 A class diagram for a system with Super members and VIP members

The addition of Super members made the system too complex. You suddenly noticed that you had 3 diamonds in your class diagram: not gemstones but 3 **Deadly Diamonds of Death**!

You tried to avoid the diamonds by transforming the `User` class into an interface and using

Composition over Inheritance Design Pattern.

But with the stress of the deadline coming, you were not able to use all the cells of your brain.

In fact, this complexity prevented you from delivering the system before the deadline. You tell yourself that you should have used composition instead of class inheritance. But it's too late now.

TIP **In OO, prefer composition over class inheritance.**

You call Nancy in order to explain her the situation at 10:00 PM, two hours before the deadline:

YOU: Look Nancy, we really did our best, but we will not be able to add **Super members** to the system before the deadline

NANCY: No worries, my business partner and I decided to **postpone** the launch.

YOU: Phew!

NANCY: Do you think that if we add other tiny features later, you'd be able to handle them on time?

YOU: Yes

NANCY: How could it be?

YOU: We are going to **refactor** the system from Object Oriented to **Data Oriented**.

NANCY: What is Data Oriented?

YOU: It is a **magic sauce** that allows developers to write code for **changing** requirements **faster** !

TIP **DO is a magic sauce that allows developers to write code for changing requirements faster!**

After reading this book, you will belong to the community of happy developers who know the recipe of DO magic sauce.

1.8 Wrapping up

In this chapter, we have explored the **tendency** of OO to increase system complexity, in the sense that OO systems tend to be **hard to understand**. The root cause of the complexity increase is related to the mixing of **code and data together** into objects.

We illustrated how some fundamental **aspects** of OO tend to increase the **complexity** of OO systems.

Aspects of Object Oriented programming and their impact on increased system complexity

Aspect	Impact on increased complexity
Code and data are mixed	Classes tend to be involved in many relations
Objects are mutable	Extra thinking when reading code
Objects are mutable	Explicit synchronization on multi-threaded environments
Data is locked in objects	Data serialization is not trivial
Code is locked in classes	Class hierarchies are complex

It is possible to deal with this complexity with **smart design patterns** and **advanced features** of the language. This book proposes to deal with this complexity by adhering to **Data Oriented programming**, a paradigm that could be implemented both in OO and FP.

2 Reduce system complexity by separating Code from Data

2.1 Introduction

As we mentioned in Chapter 0, the big insight of Data Oriented Programming (DO) is that we can **decrease the complexity** of our systems by **separating code from data**. Indeed, when code is separated from data, our systems are made of two main pieces that can be **thought separately**: Data entities and Code modules.

This chapter is a deep dive in the first principle of Data Oriented Programming:

NOTE **Principle #1: Separate code from data in a way that the code resides in functions whose behavior does not depend on data that is somehow encapsulated in the function's context.**

We **illustrate** the separation between code and data in the context of the Library Management system that we introduced in Chapter 1 and we unveil the **benefits** that this separation brings to the system:

1. The system is **simple**: it is easy to understand
2. The system is **flexible**: quite often, it requires no design changes to adapt to changing requirements

We show how to:

1. **Design a system** where code and data are separate
2. **Write code** that respects the separation between code and data.

This chapter focuses on the **design of the code part** of a system where code and data are separate. In Chapter 3, we will focus on the **design of the data part** of the system. As we progress in the book, we will discover other benefits of separating code from data.

2.2 The two parts of a DO system

A quick research among your friends regarding a DO expert to teach you DO lead to Joe, a 40-year old developer that used to be a Java developer for many years and moved to Clojure 7 years ago.

You decide to hire Joe for a 1 one 1 workshop in your office.

When you tell Joe about the Library management system you built (Chapter 1) and the details of the struggle you had to adapt to changing requirements, he is not surprised.

Joe tells you that the systems he and his team have build in Clojure over the last 7 years are **less complex** and **more flexible** than the systems he used to build in Java. The main cause of this benefits is that the systems he built were following principles of Data Oriented Programming.

YOU: What makes DO systems **less complex** and **more flexible**?

JOE: The first **insight** of DO is about the relationships between **code** and **data**.

YOU: You mean the **encapsulation** of data in objects?

JOE: Actually, DO is against encapsulation.

YOU: Why is that? I thought encapsulation was a positive programming paradigm.

JOE: Data encapsulation has its merits and drawbacks: Think about the way you designed the Library Management System (in Chapter 1). According to DO, the main cause of the **complexity** of systems and their **lack of flexibility** is because code and data are mixed together (in objects).

TIP **DO is against data encapsulation.**

YOU: Does it mean that in order to adhere to DO, I need to get rid of OO and learn a Functional programming language?

JOE: No. DO principles are **language agnostic**: they can be applied both in OO and FP languages.

YOU: Cool! I was afraid that you were going to teach me about monads, algebraic data types and high order functions.

JOE: None of this is required in DO.

TIP **DO Principles are language agnostic.**

YOU: How does the separation between code and data look like in DO?

JOE: Data is represented by **data entities** that hold members only. Code is aggregated into **modules** where all the functions are **stateless**.

YOU: What do you mean by stateless functions?

JOE: Instead of having the state encapsulated in the object, the data entity is passed as an argument.

YOU: I don't get that.

JOE: Let me make it visual.

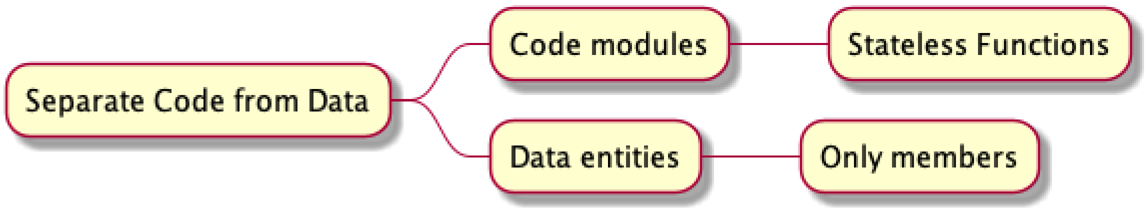


Figure 2.1 The separation between code and data

YOU: It's still not clear

JOE: It will become clearer when I show you how it looks like in the context of your library management system.

YOU: OK. Shall we start we code or with data?

JOE: Well, it's **Data** oriented programming. Let's start with Data!

2.3 Data entities

In DO, we start the design process by discovering the data entities of our system.

JOE: What are the **data entities** of your system?

YOU: What do you mean by *data entities*?

JOE: I mean the parts of your system that hold **information**.

NOTE Data entities are the parts of your system that hold information

YOU: Well, it's a library management system, so for sure we have **books** and **members**.

JOE: Of course. But there are more: One way to discover the data entities of a system is to look for **nouns** and **noun phrases** in the requirements of the system.

You look at Nancy's requirement napkin and you highlight the **nouns** and **noun phrases** that seem to represent data entities of the system:

SIDEBAR **Highlighting terms in the requirements that correspond to data entities**

- Two kinds of users: library members and librarians
- Users log in to the system via email and password.
- Members can borrow books
- Members and librarians can search books by title or by author
- Librarians can block and unblock members (e.g. when they are late in returning a book)
- Librarians can list the books currently lent by a member
- There could be several copies of a book

JOE: Excellent. Can you see a natural way to group the entities?

YOU: Not sure, but it seems to me that *users*, *members* and *librarians* form a group while *books*, *authors* and *book copies* form another group.

JOE: Sounds good to me. How would you call each group?

YOU: User management for the first group and **Catalog** for the second group.

SIDEBAR **The data entities of the system organized in a nested list**

- The catalog data
  - Data about books
  - Data about authors
  - Data about book items
  - Data about book lendings
- The user management data
  - Data about users
  - Data about members
  - Data about librarians

YOU: I am not sure about the relationships between books and authors: should it be association or composition?

JOE: Don't worry too much about the details for the moment. We will refine our data entities design later (Chapter 3). For now, let’s visualize the two groups in a mind map.

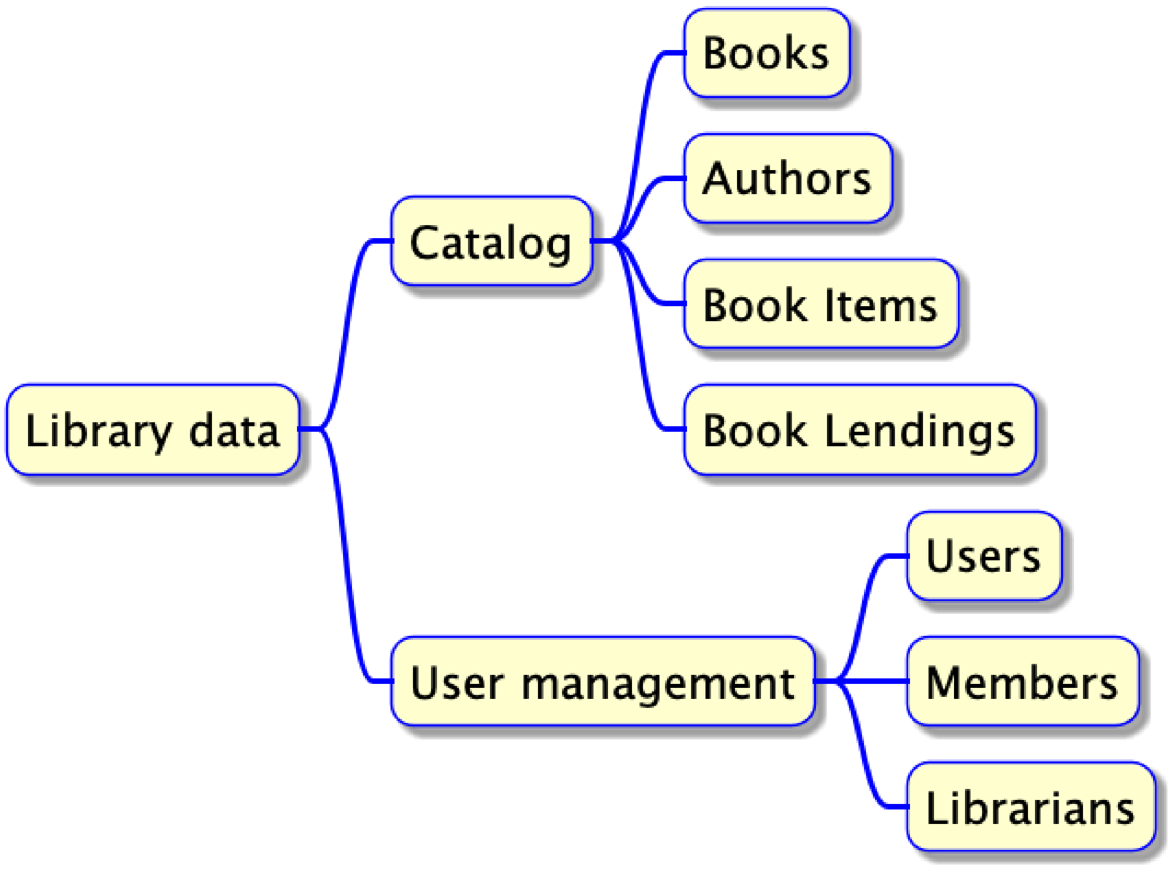


Figure 2.2 The data entities of the system organized in a mind map

The most precise way to visualize the data entities of a DO system is to draw a data entity diagram with different arrows for association and composition. We will come back to data entity diagram in Chapter 3.

TIP **Discover the data entities of your system and group them into high level groups, either as a nested list or as a mind map.**

We will get deeper into the design and the representation of data entities in Chapter 3. For now, let’s simplify and say that the data of our library system is made of two high level groups: **User Management** and **Catalog**.

2.4 Code modules

The second step of the design process in DO, is to define the **code modules** of the system.

JOE: Now that you have identified the data entities of your system and group them into high level groups, it’s time to think about the **code part** of your system.

YOU: What do you mean by *code part*?

JOE: One way to think about it is to identity the **functionalities of your system**.

You look again at Nancy's requirement napkin and this time you highlight the **verb phrases** that represent functionalities of the system:

SIDEBAR Highlighting terms in the requirements that correspond to functionalities

- Two kinds of users: library members and librarians
- Users log into the system via email and password.
- Members can borrow books
- Members and librarians can search books by title or by author
- Librarians can block and unblock members (e.g. when they are late in returning a book)
- Librarians can list the books currently lent by a member
- There could be several copies of a book

In addition to that, it is obvious that members can also return a book. Moreover, there should be a way to detect whether a user is a librarian or not.

Your write down a list of the functionalities of the system.

SIDEBAR The functionalities of the system

1. Search a book
2. Add a book item
3. Block a member
4. Unblock a member
5. Login a user into the system
6. List the books currently lent by a member
7. Borrow a book
8. Return a book
9. Check whether a user is a librarian

JOE: Excellent! Now, tell me what functionalities need to be **exposed to the outside world**?

YOU: What do you mean by *exposed to the outside world*?

JOE: Imagine that the library management system were exposing an API over HTTP: what would be the endpoints of the API?

YOU: I see. All the functionalities them beside checking if a user is a librarian should need to be exposed.

JOE: Perfect, now give to each exposed functionality a short name and gather them together in a module box called `Library`

It takes you less than a minute: Figure 2.3 shows the module box that contains the exposed functions of the Library.

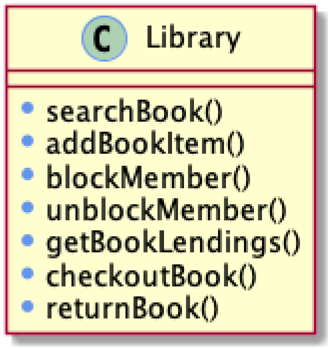


Figure 2.3 The Library module contains the exposed functions of the Library management system

TIP **The first step in designing the code part of a DO system is to aggregate the exposed functions in a single module.**

JOE: Beautiful. You just created your first **code module**.

YOU: To me it looks like a class: What's the difference between a *module* and a *class*?

JOE: A **module** is an aggregation of functions. In OO, a module is represented by a class but in other programming languages, it might be a *package* or a *namespace*.

YOU: I see.

JOE: The important thing about DO code modules is that they contain only **stateless functions**.

YOU: You mean like **static methods** in Java?

JOE: Exactly!

YOU: So how the functions know on what piece of information they operate?

JOE: We pass it as the first argument to the function.

YOU: I don't understand. Could you give me an example?

Joe takes a look at the list of functions of the `Library` module in Figure 2.3.

JOE: Let's take for example `getBookLendings()`: in classic OO, what would be its arguments?

YOU: A librarian id and a member id.

JOE: In classic OO, `getBookLendings` would be a method of a `Library` class that receives two arguments: `librarianId` and `memberId`

YOU: Yeap.

JOE: Now comes the subtle part: in DO, `getBookLendings` is part of the library module and it receives the `LibraryData` as the first argument, in addition to the other arguments.

YOU: Could you show me what you mean?

JOE: Sure.

Joe gets closer to your keyboard and start typing.

That's how a class method looks like in OO:

```
class Library {
    libraryData // state of the object

    getBookLendings(userId, memberId) {
        // accesses library data via this.libraryData
    }
}
```

The method accesses the **state** of the object—in our case the library data— via `this.libraryData`. The object's state is an **implicit argument** to the object's methods.

TIP **In classic OO, the state of the object is an implicit argument to the methods of the object.**

In DO, the signature of `getBookLendings` would look like this:

```
class Library {
    static getBookLendings(libraryData, userId, memberId) {
    }
}
```

The **state** of the library is stored in `libraryData` that is managed outside the `Library` class and `LibraryData` is passed to the `getBookLendings` **static** method as an **explicit argument**.

TIP **In DO, functions of a code module are stateless: they receive the data they manipulate as an explicit argument, usually the first argument.**

The same rule applies to the other functions of the library module. All of them are stateless: they receive the library data as first argument.

IMPORTANT **A module is an aggregation of functions. In DO, the module functions are stateless.**

You apply this rule and you refine the design of the library module by including the details about functions' arguments.

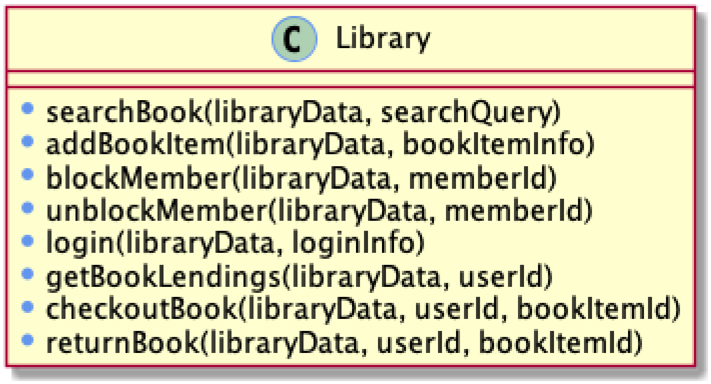

Library
searchBook(libraryData, searchQuery)
addBookItem(libraryData, bookItemInfo)
blockMember(libraryData, memberId)
unblockMember(libraryData, memberId)
login(libraryData, loginInfo)
getBookLendings(libraryData, userId)
checkoutBook(libraryData, userId, bookItemId)
returnBook(libraryData, userId, bookItemId)

Figure 2.4 The Library module with the function arguments

JOE: Perfect. Now, we are ready to design at a high level our system.

YOU: What's a **high level design in DO**?

JOE: The **definition** of modules and the **interaction** between them.

YOU: I see. Is there any guideline to help me define the modules?

JOE: Definitely. The **high level modules** of the system correspond to the **high level data entities**.

YOU: You mean the data entities that appear in the data mind map?

JOE: Exactly!

You look again at the data mind map in Figure 2.5 and you focus on the high level data entities: **Library**, **Catalog** and **User management**.

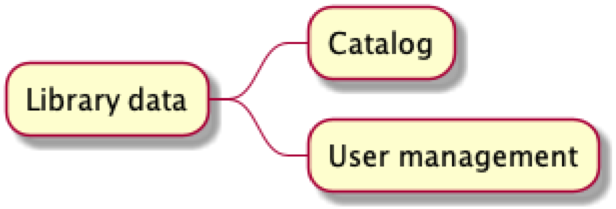


Figure 2.5 A mindmap of high level data entities of the Library management system

It means that in the system, beside the `Library` module, we have two high level modules:

1. `Catalog` module that deals with catalog data
2. `UserManagement` module that deals with user management data

Then you draw the high level design of library management system, by adding `Catalog` and `UserManagement` modules:

- Functions of `Catalog` receive `catalogData` as first argument
- Functions of `UserManagement` receive `userManagementData` as first argument

Here is the diagram:

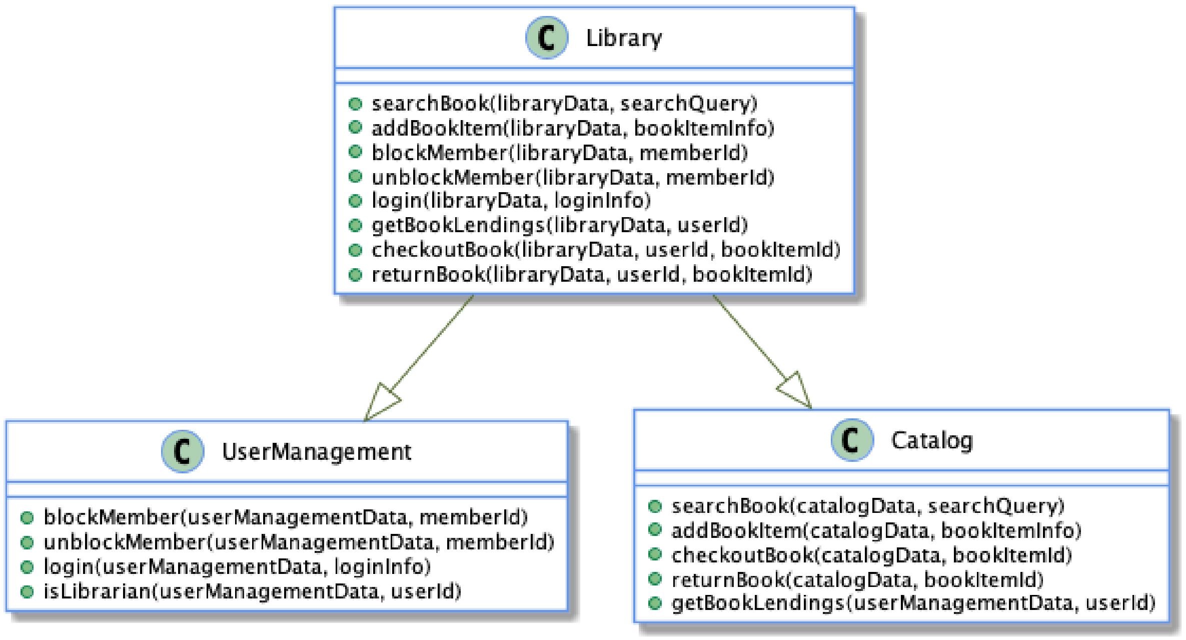

Library
searchBook(libraryData, searchQuery)
addBookItem(libraryData, bookItemInfo)
blockMember(libraryData, memberId)
unblockMember(libraryData, memberId)
login(libraryData, loginInfo)
getBookLendings(libraryData, userId)
checkoutBook(libraryData, userId, bookItemId)
returnBook(libraryData, userId, bookItemId)
UserManagement
blockMember(userManagementData, memberId)
unblockMember(userManagementData, memberId)
login(userManagementData, loginInfo)
isLibrarian(userManagementData, userId)
Catalog
searchBook(catalogData, searchQuery)
addBookItem(catalogData, bookItemInfo)
checkoutBook(catalogData, bookItemId)
returnBook(catalogData, bookItemId)
getBookLendings(userManagementData, userId)

Figure 2.6 The modules of the Library management system with the function arguments

It might not yet be clear for you how the data entities get passed between modules. For the moment, you can think of `libraryData` as a class with two members:

- `catalog` that holds the catalog data
- `userManagement` that holds the user management data

The functions of `Library` share a common pattern:

1. They receive `libraryData` as an argument
2. They pass `libraryData.catalog` to functions of `Catalog`
3. They pass `libraryData.userManagement` to functions of `UserManagement`

Later on, in this chapter, we will see the code for some functions of the `Library` module.

TIP	**The high level modules of a DO system correspond to the high level data entities.**

2.5 DO systems are easy to understand

You take a look at the two diagrams that represent the high level design of your system:

1. The data entities in the data mind map from Figure 2.7
2. The code modules in the module diagram from Figure 2.8

A bit perplexed, you ask Joe:

YOU: I am not sure that this system is better than a classic OO system, where **objects encapsulate data**.

JOE: The main benefit of a DO system over a classic OO systems is that it is **easier to understand**.

YOU: What makes it easier to understand?

JOE: The fact that the system is split clearly in code modules and data entities.

YOU: I don’t get you.

JOE: When you try to understand the data entities of the system, you don’t have to think about the details of the code that manipulates the data entities.

YOU: You mean that when I look at the data mind map of my library management system, I am able to understand it on its own?

JOE: Exactly. And similarly, when you try to understand the code modules of the system, you don’t have to think about the details of the data entities manipulated by the code. There is a clear **separation of concerns** between the code and the data.

You look again at the data mind map in Figure 2.7, and you get kind of a Aha moment:

Data lives on its own!

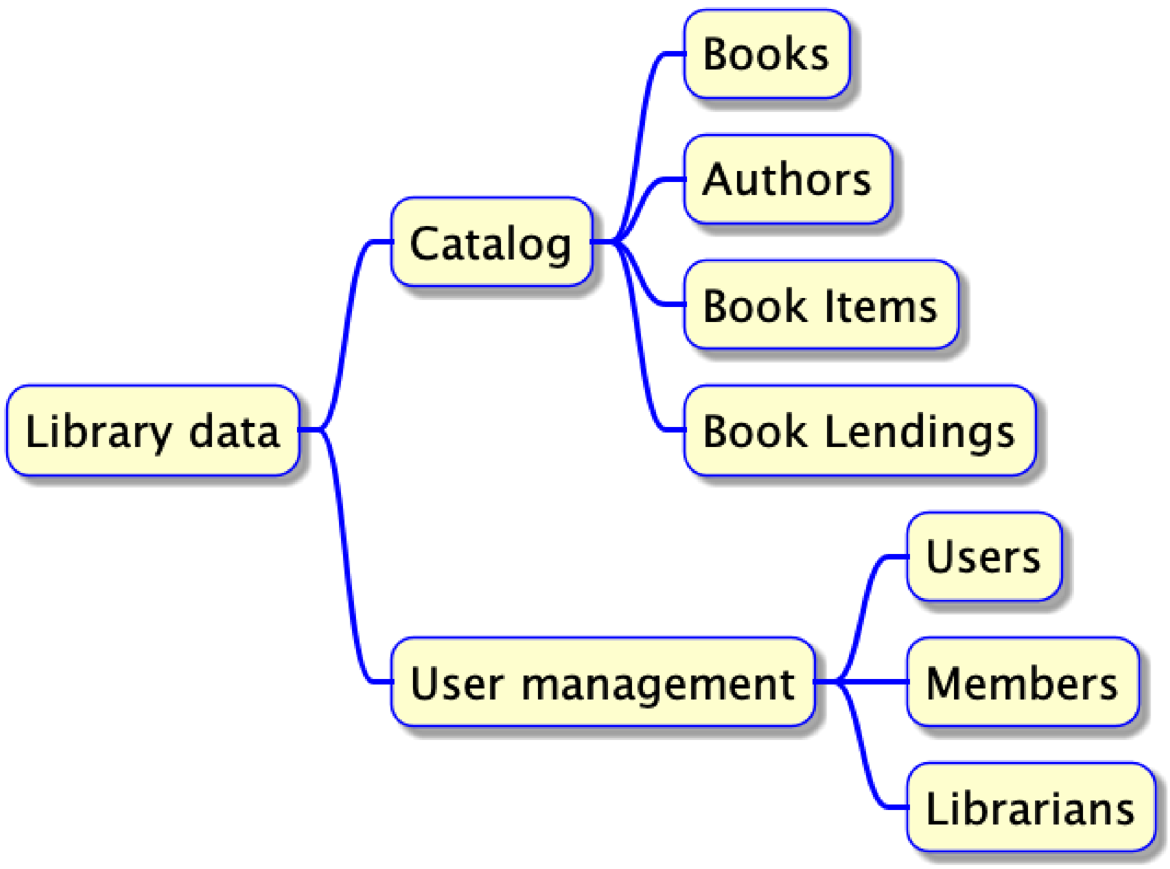


Figure 2.7 A data mindmap of the Library management system

IMPORTANT A DO system is easier to understand because the system is split in two parts: data entities and code modules.

Now you look at the module diagram in Figure 2.8 and you feel a bit confused:

- On one hand, the module diagram looks **similar** to the class diagrams from **classic OO**: boxes for classes and arrows for relations between classes.
- On the other hand, the code module diagram looks much **simpler** than the class diagrams from **classic OO**, but you cannot explain why.

You ask Joe for a clarification.

YOU: The module diagram seems much simpler that the class diagrams I am used to in OO. I feel it but I cannot put words on it.

JOE: The reason is that module diagrams have **constraints**.

YOU: What kind of constraints?

JOE: Constraints on the functions as we saw before: All the functions are static (stateless). But also **constraints on the relations between the modules**.

TIP	All the functions in a DO module are stateless.

YOU: Could you explain that?

JOE: There is a single kind of relation between DO modules: the **usage relation**. A module uses code from another module. No **association**, no **composition** and no **inheritance** between modules. That's what make a DO module diagram easy to understand.

YOU: I understand why there is no association and no composition between DO modules: after all, association and composition are **data relations**. But why no **inheritance** relation? Does it mean that in DO is against **polymorphism**?

JOE: That's a great question. The quick answer is that in DO, we achieve polymorphism with a different mechanism than class inheritance. We will talk about it later (in Chapter 5).

YOU: Now, you triggered my curiosity: I was quite sure that inheritance was the only way to achieve polymorphism.

TIP	The only kind of relation between DO modules is: usage relation.

You look again at the module diagram in Figure 2.8 and now you not only feel that this diagram is simpler than classic OO class diagrams, you understand why it is simpler: All the functions are static and all the relation between modules are of type usage.

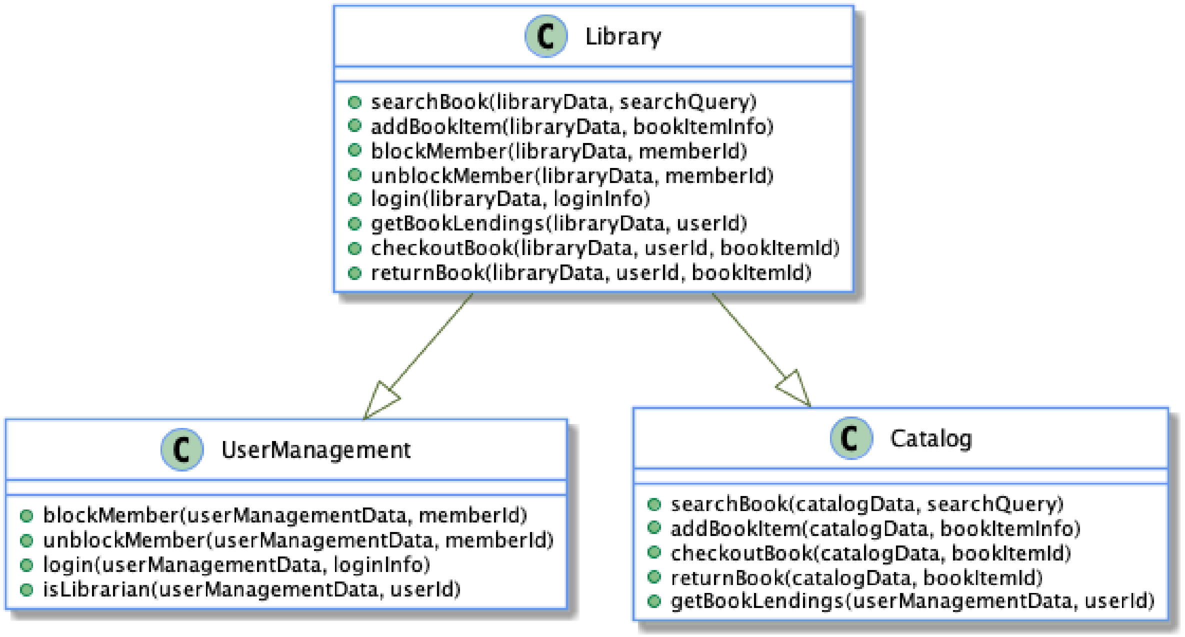

C Library
searchBook(libraryData, searchQuery)
addBookItem(libraryData, bookItemInfo)
blockMember(libraryData, memberId)
unblockMember(libraryData, memberId)
login(libraryData, loginInfo)
getBookLendings(libraryData, userId)
checkoutBook(libraryData, userId, bookItemId)
returnBook(libraryData, userId, bookItemId)
C UserManagement
blockMember(userManagementData, memberId)
unblockMember(userManagementData, memberId)
login(userManagementData, loginInfo)
isLibrarian(userManagementData, userId)
C Catalog
searchBook(catalogData, searchQuery)
addBookItem(catalogData, bookItemInfo)
checkoutBook(catalogData, bookItemId)
returnBook(catalogData, bookItemId)
getBookLendings(userManagementData, userId)

Figure 2.8 The modules of the Library management system with the function arguments

Table 2.1 What makes each part of a DO system easy to understand

System part	Constraint on entities	Constraints on relations
Data entities	Members only (no code)	Association and Composition
Code modules	Stateless functions (no members)	Usage (no inheritance)

TIP Each part of a DO system is easy to understand, because it has constraints.

2.6 DO systems are flexible

YOU: I get that the sharp separation between code and data makes DO systems easier to understand than classic OO systems. But what about adapting to changes in requirements?

JOE: Another benefit of DO systems is that it is **easy to adapt** them to changing requirements.

YOU: I remember that when Nancy asked me to add *Super Members* and *VIP Members* to the system, it was hard to adapt my OO system: I had to introduce a few base classes and the **class hierarchy became really complex**.

JOE: I know exactly what you are talking about. I experienced the same kind of struggle when I was a OO developer. Tell me what were the changes in the requirements for *Super Members* and *VIP Members* and I am quite sure that you will see by yourself that it is easy to adapt your DO system.

SIDEBAR The requirements for Super Members and VIP Members

1. Super Members are members that are allowed to list the book lendings of other members
2. VIP Members are members that are allowed to add book items to the library

You open your IDE and you start to code the `getBookLendings` function of the Library module, first without addressing the requirements for Super Members. You remember what Joe told you about module functions in DO:

1. Functions are **stateless**
2. Functions receive the **data** they manipulate as **first argument**

In terms of functionalities, `getBookLendings` have two parts:

1. Check that the user is a librarian
2. Retrieve the book lendings from the catalog

Basically, the code of `getBookLendings` have two parts:

1. Call `isLibrarian()` function from the `UserManagement` module and pass it the `UserManagementData`
2. Call `getBookLendings()` function from the `Catalog` module and pass it the `CatalogData`

Here is the code for `Library.getBookLendings()`:

Listing 2.1 Getting the book lendings of a member

```
class Library {
    static getBookLendings(libraryData, userId, memberId) {
        if(UserManagement.isLibrarian(libraryData.userManagement, userId)) {
            return Catalog.getBookLendings(libraryData.catalog, memberId);
        } else {
            throw "Not allowed to get book lendings";   ❶
        }
    }
}

class UserManagement {
    static isLibrarian(userManagementData, userId) {
        // will be implemented later   ❷
    }
}

class Catalog {
    static getBookLendings(catalogData, memberId) {
        // will be implemented later   ❸
    }
}
```

❶ There are other ways to manage errors

❷ In Chapter 3, we will see how to manage permissions with generic data collections

❸ In Chapter 3, we will see how to query data with generic data collections

It's your first piece of DO code: passing around all those data objects `libraryData`, `libraryData.userManagement` and `libraryData.catalog` feels a bit awkward. But you made it.

Joe looks at your code and seems satisfied.

JOE: How would you adapt your code to adapt to Super Members?

YOU: I would add a function `isSuperMember` to the `UserManagement` module and call it from `Library.getBookLendings`

JOE: Exactly! It's as simple as that.

You type this piece of code on your laptop:

Listing 2.2 Allowing Super Members to get the book lendings of a member

```
class Library {
    static getBookLendings(libraryData, userId, memberId) {
        if(Usermanagement.isLibrarian(libraryData.userManagement, userId) ||
           Usermanagement.isSuperMember(libraryData.userManagement, userId)) {
            return Catalog.getBookLendings(libraryData.catalog, memberId);
        } else {
            throw "Not allowed to get book lendings";   ❶
        }
    }
}

class UserManagement {
    static isLibrarian(userManagementData, userId) {
        // will be implemented later   ❷
    }
    static isSuperMember(userManagementData, userId) {
        // will be implemented later   ❷
    }
}

class Catalog {
    static getBookLendings(catalogData, memberId) {
        // will be implemented later   ❸
    }
}
```

❶ There are other ways to manage errors

❷ In Chapter 3, we will see how to manage permissions with generic data collections

❸ In Chapter 3, we will see how to query data with generic data collections

Now, the awkward feeling caused by passing around all those data objects is dominated by a feeling of relief: Adapting to this change in requirement takes only a few lines of code and require **no changes in the system design**.

Once again, Joe seems satisfied.

TIP **DO systems are flexible. Quite often, they adapt to changing requirements without changing the system design.**

You prepare yourself a cup of coffee, and you start coding the `addBookItem()` code.

You look at the signature of `Library.addBookItem()` in Listing 2.3 and it is not clear to you what is the meaning of the third argument `bookItemInfo`. You ask Joe for a clarification.

Listing 2.3 The signature of `Library.addBookItem`

```
class Library {
    static addBookItem(libraryData, userId, bookItemInfo) {
    }
}
```

YOU: What is `booItemInfo`?

JOE: Let’s call it the book item information and imagine we have a way to **represent this information** in a **data entity** named `bookItemInfo`.

YOU: You mean an object?

JOE: For now, it’s ok to think about `bookItemInfo` as an object. Later on (in Chapter 3), I will show you how to **we represent data** in DO.

Beside this subtlety about how the book item info is represented by `bookItemInfo`, the code for `Library.addBookItem()` in Listing 2.4 is quite similar to the code you wrote for `Library.getBookLendings()` in Listing 2.2. Once again, you are amazed by the fact that adding support for VIP Members requires **no design change**.

Listing 2.4 Allowing VIP Members to add a book item to the library

```
class Library {
    static addBookItem(libraryData, userId, bookItemData) {
        if(UserManagement.isLibrarian(libraryData.userManagement, userId) ||
           UserManagement.isVIPMember(libraryData.userManagement, userId)) {
            return Catalog.addBookItem(libraryData.catalog, bookItemData);
        } else {
            throw "Not allowed to add a book item";   ❶
        }
    }
}

class UserManagement {
    static isLibrarian(userManagementData, userId) {
        // will be implemented later   ❷
    }
    static isVIPMember(userManagementData, userId) {
        // will be implemented later   ❷
    }
}

class Catalog {
    static addBookItem(catalogData, memberId) {
        // will be implemented later   ❸
    }
}
```

❶ There are other ways to manage errors

❷ In Chapter 3, we will see how to manage permissions with generic data collections

❸ In Chapter 4, we will see how to manage state of the system with immutable data

YOU: It required a big **mindset shift** for me to learn how to separate code from data.

JOE: What was the most challenging part for your mind?

YOU: The fact that data is not encapsulated in objects.

JOE: It was the same for me when I switched from OO to DO.

YOU: Will there be other mindset shifts in my journey into DO?

JOE: There will be two more mindset shifts but I think that they will be less challenging than separating code from data.

YOU: What will it be about?

JOE: Representing data entities with **generic data structures** (Chapter 3) and constraining ourselves to **immutable data objects** (Chapter 4).

But before that you and Joe go to a lunch at *Simple*, a nice small restaurant near your office.

2.7 Wrapping up

In this chapter, we have illustrated DO Principle #1 about the separation between code from data:

NOTE **Principle #1: Separate code from data in a way that the code resides in functions whose behavior does not depend on data that is somehow encapsulated in the function's context.**

It required quite a big mindset shift to learn that in DO:

- Code is **separated** from data
- Code is aggregated in **modules**
- Data is aggregated in **data entities**
- Code is made of **stateless** functions
- Functions receive **data as first argument**

We illustrated how to apply this principle in a OO language.

A consequence of this separation is that:

- We have the **freedom** to design code and data in isolation
- Module diagrams are **simple**: it's only about usage (no inheritance)
- Data entities diagram are **simple**: it's only about association and composition

The details of Principle #1 are summarized in this mindmap:

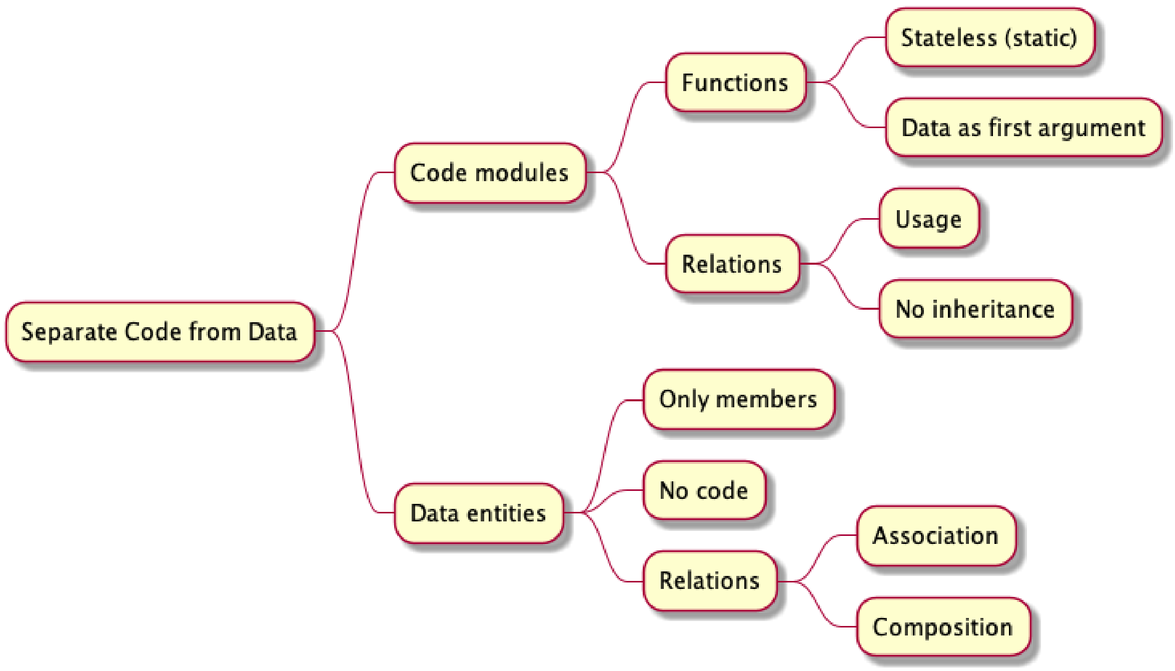


Figure 2.9 The summary of Principle #1: Separate code from data

Overall, the DO systems are **simpler** (easier to understand) than classic OO systems and **more flexible** (easier to adapt to changing requirements).

3 Manipulate the whole system data with generic functions

3.1 Introduction

Now that we have separated code from data, let’s talk about data on its own.

Given a **system data model** designed as a **rigid** class hierarchy in OO, DO prescribes that we represent our data model as a **flexible** combination of **maps** and **collections** where we can access each piece of information via an **information path**.

This chapter is a deep dive in DO Principle #2:

NOTE **Represent data entities with generic data structures**

We increase system **flexibility** when we represent **records as string maps** and not as objects instantiated from classes. This **liberates** data from the rigidity of a class-based system. Data becomes **first class citizens** powered by **generic** functions to add, remove or rename fields.

The dependency between the code that manipulates data, and the data, is a **weak dependency**. The only thing that matters are the names of the fields we want to manipulate.

In this chapter, we’ll deal only with data query. We’ll discuss managing changes in system state in Chapter 4.

3.2 Design a data model

During lunch at *Simple*, you and Joe don’t talk about programming. Instead, you try to get to know Joe on a personal level and talk about family, hobbies and health. You find out that Joe is married with two kids, and that he meditates daily.

The food is good! And somehow, it helps you digest the DO material you ingested in the morning!

As soon as you're back at the office, you ask Joe about the next step in your journey into DO, which is about the **data model** and **data representation**…

JOE: When we get to the **design** of the data part of our system, we can design it in **isolation**.

YOU: What do you mean by *isolation*?

JOE: I mean that you don't have to bother with code. Only data.

YOU: Yes, I remember you telling me it's a key aspect that makes a DO system simpler than OO. After all, **separation of concerns** is a design principle I am used to in OO.

JOE: Indeed.

YOU: And when we think about data, the only relations we have to think about are **association** and **composition**.

JOE: Correct.

YOU: Will the data model design be significantly different than the data model I'm used to designing as an OO developer?

JOE: Not so much.

YOU: OK. Let me draw a data entity diagram.

You take a look at the data mind map that you drew in Chapter 2:

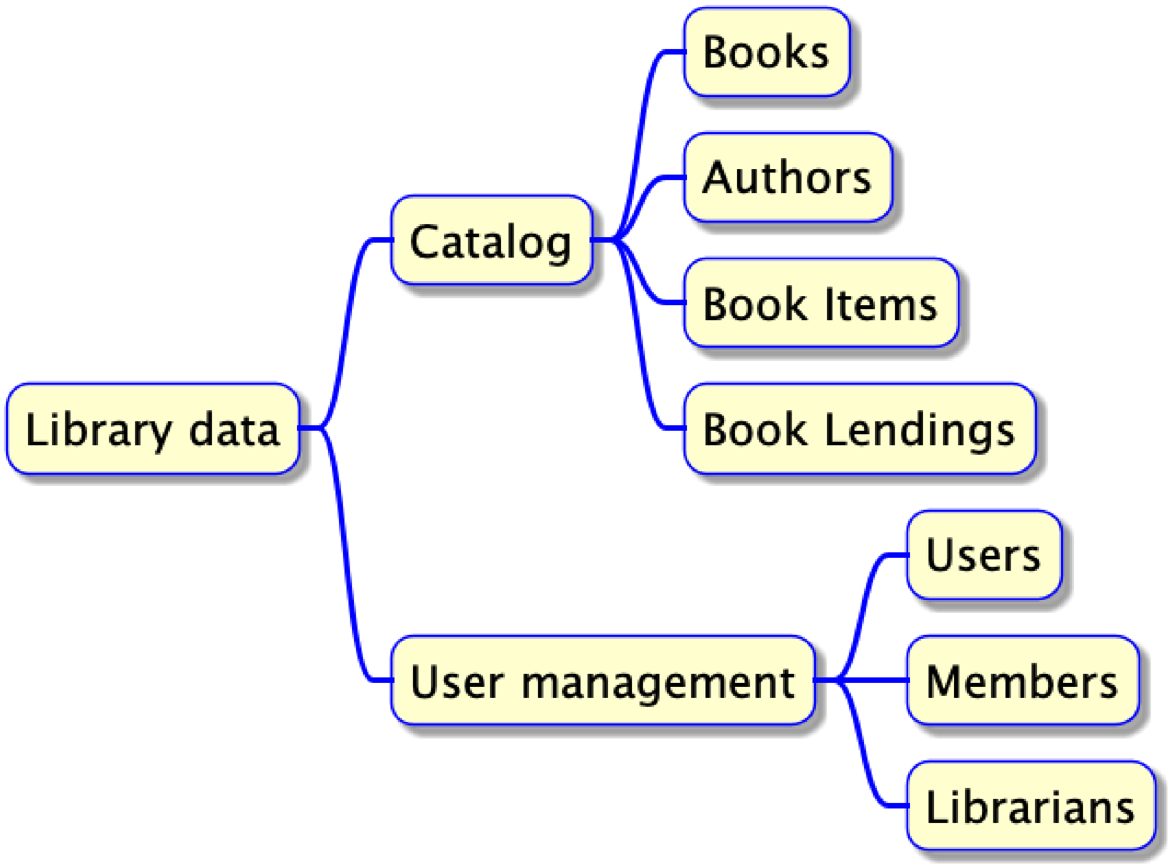


Figure 3.1 A data mindmap of the Library management system

You **refine the details** of the fields of each data entity and the kind of relationships between entities, and the result is this data entity diagram:

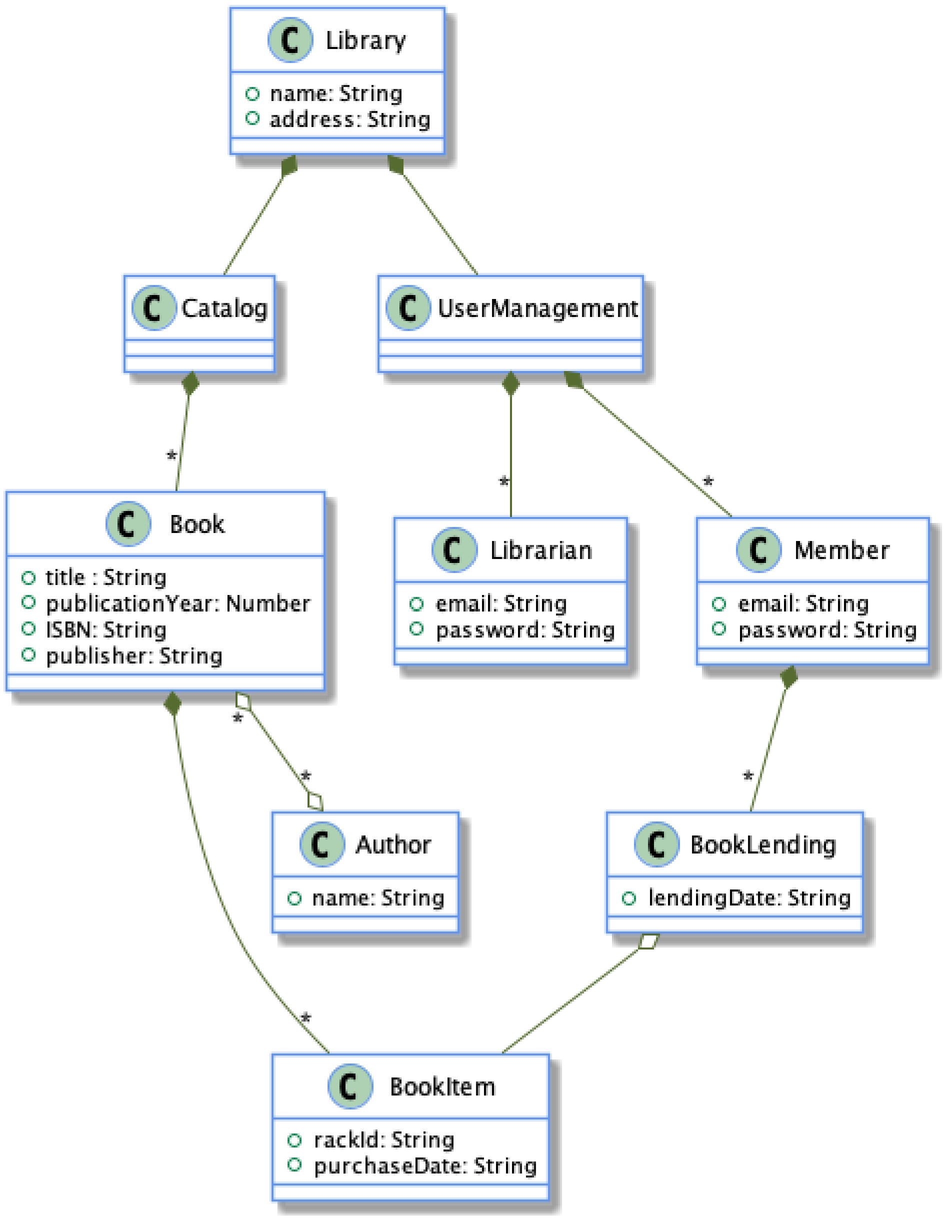


Figure 3.2 A data model of the Library management system

JOE: The next step is to be more **explicit** about the relations between entities.

YOU: What do you mean?

JOE: For example, in your entity diagram, `Book` and `Author` are connected by a many-to-many association relation. How is this **relation** going to be **represented** in your program?

YOU: In the `Book` entity, there will be a **collection** of author IDs, and in the `Author` entity, there will be a **collection** of book IDs.

JOE: Sounds good. And what will be the book ID?

YOU: The book ISBN.[1]

JOE: And where will you hold the index that will enable you to retrieve a `Book` from its ISBN?

YOU: In the `Catalog`. The catalog holds a `bookByISBN` index.

JOE: What about author ID?

YOU: Author ID is the author name, in lower case, and with dashes instead of white spaces (assuming that we don't have two authors with the same name).

JOE: And I guess that you also hold the author index in the `Catalog`?

YOU: Exactly!

JOE: Excellent. You've been 100% explicit about the relation between `Book` and `Author`. I'll ask you to do the same with the other relations of the system.

It's quite easy for you. You did that so many times as an OO developer. Here's the detailed entity diagram of your system:

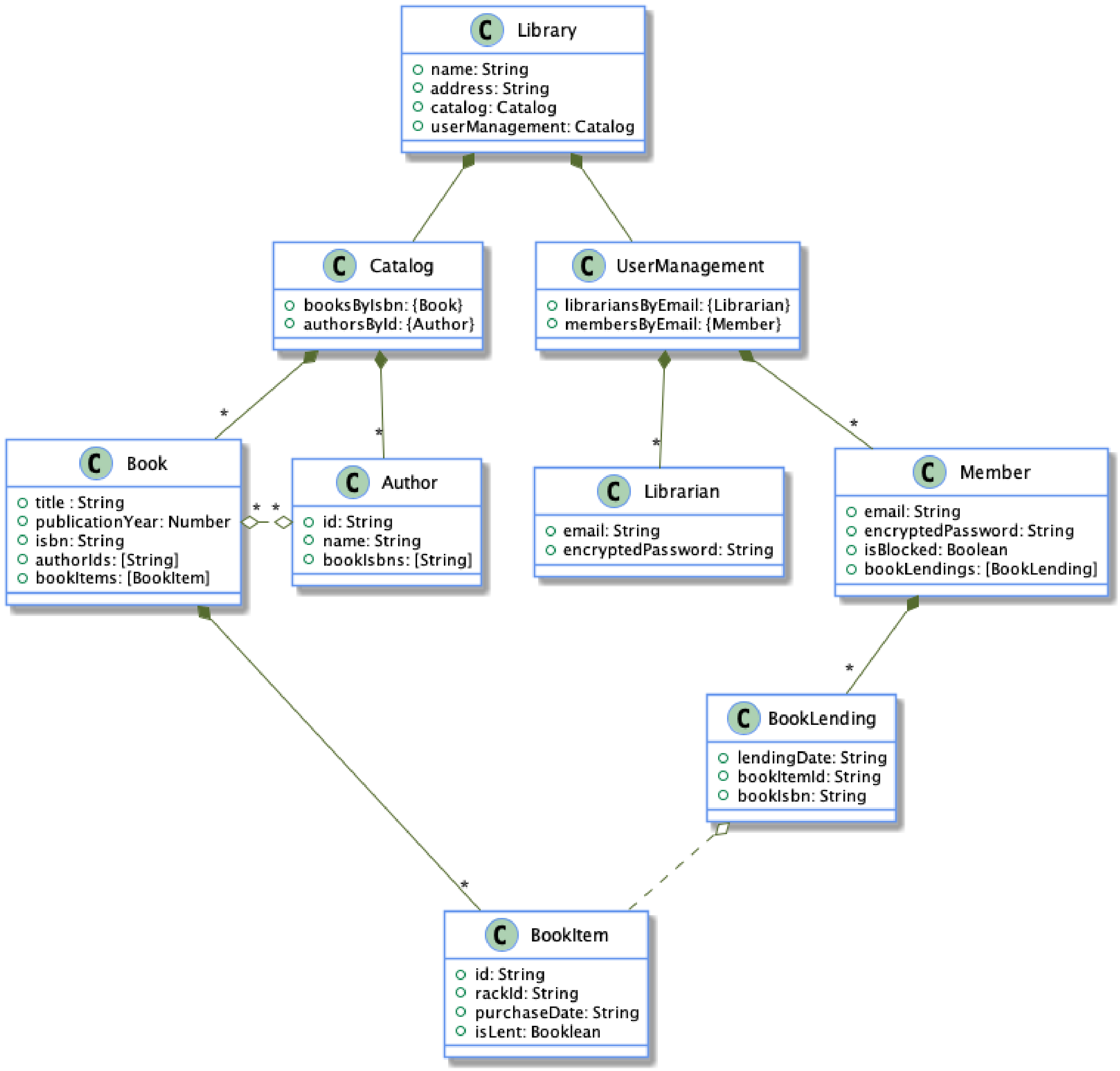


Figure 3.3 Library management relations model. Dashed lines (e.g., between Book and Author) denotes indirect relations. [String] denotes a collection of strings. {Book} denotes an index of Books.

The `Catalog` entity contains two **indexes**:

1. `booksByIsbn`: The keys are book ISBNs and the values are `Book` entities. Its type is noted as `{Book}`
2. `authorsById`: The keys are author IDs and the values are `Author` entities. Its type is noted as `{Author}`

Inside a `Book` entity, we have `authors`, which is a **collection** of author IDs of type `[String]`. Inside an `Author` entity, we have `books`, which is a **collection** of book IDs of type `[String]`.

NOTE **Notation for collection and index types: A collection of `Strings` is noted as `[String]`. An index of `Books` is noted as `{Book}`. In the context of a data model, the index keys are always strings.**

There is a **dashed line** between `Book` and `Author`, which means that the relation between `Book` and `Author` is **indirect**. To access the collection of `Author` entities from a `Book` entity, we'll use the `authorById` index defined in the `Catalog` entity.

JOE: I like your **data entity diagram**.

YOU: Thank you.

JOE: Can you tell me what the three kinds of **data aggregations** are in your diagram (and in fact in any data entity diagram)?

YOU: Let me see… We have **collections**, like `authors` in `Book`. We have **indexes**, like `booksByIsbn` in `Catalog`. I can't find the third one.

JOE: The third kind of data aggregation is what we've called until now an "entity" (like `Library`, `Catalog`, `Book`, etc…). The common term for "entity" in computer science is **record**.

NOTE **Record: A record is a data structure that groups together related data items. It's a collection of fields, possibly of different data types.**

YOU: Is it correct to say that a data entity diagram consists only of records, collections and indexes?

JOE: That's correct. Can you make a similar statement about the relations between entities?

YOU: The relations in a data entity diagram are either **composition** (solid line with full diamond) or **association** (dashed line with empty diamond). Both types of relations can be either 1-to-1, 1-to-many or many-to-many.

JOE: Excellent!

TIP **A data entity diagram consists of records whose values are either primitives, collections or indexes. The relation between records is either composition or association.**

3.3 Represent records as maps

So far, we've illustrated the benefits we gain from the **separation between Code and Data** at a high system level. There's a **separation of concerns** between code and data, and each part has clear constraints:

1. **Code** consists of **static functions** that receive data as an **explicit** argument
2. **Data** entities are modeled as **records**, and the relations between records are represented

by **collections** and **indexes**

Now comes the question of the **representation of the data**.

While DO has nothing special to say about collections and indexes, it's strongly opinionated about the **representation of records**. It applies to every programming language, dynamically- or statically-typed, Object-Oriented or Functional, it doesn't matter. In DO, records should be represented by **generic data structures** such as maps. Let's see how and why…

YOU: I'm really curious to know how we represent collections, indexes and records in DO.

JOE: Let's start with **collections**. DO is not opiniated about the representation of collections. They can be linked lists, arrays, vectors, sets or other collections best suited for the use case.

YOU: It's like in OO.

JOE: Right. For now, to keep things simple, we'll use arrays to represent collections.

YOU: What about **indexes**?

JOE: Indexes are represented as **homogeneous maps** with string keys.

YOU: What do you mean by an *homogeneous* map?

JOE: I mean that all the values of the map are of the same kind. For example, in a `Book` index, all the values are `Book`, in an author index, all the values are `Author`, etc…

YOU: Again, it's like in OO.

NOTE **A homogeneous map is a map where all the values are of the same type. A heterogeneous map is a map where the values are of different types.**

JOE: Now, here's the big surprise. In DO, **records are represented as maps**; more precisely, **heterogeneous maps** with string keys.

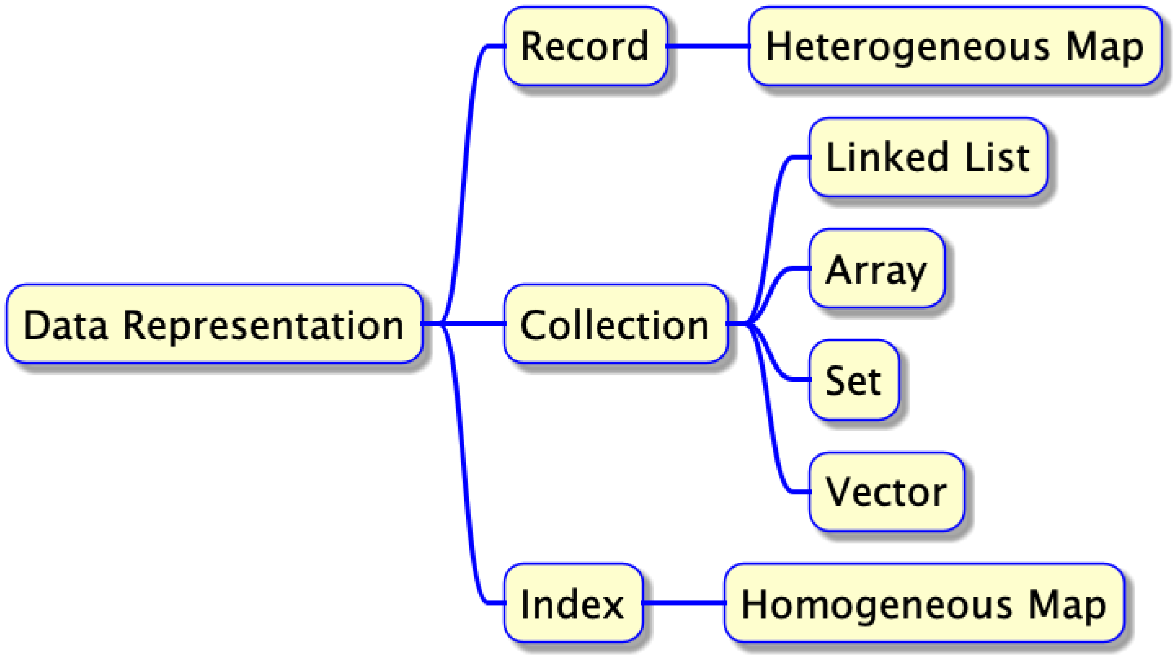


Figure 3.4 The building blocks of data representation

You stay silent for a while. You're shocked to hear that one can represent the data entities of a system as a generic data structure, where the field names and value types are not specified in a class.

Then you ask Joe:

YOU: What are the benefits of this folly?!

JOE: **Flexibility** and **genericity**.

YOU: Could you explain, please?

JOE: I'll explain in a moment, but before that, I'd like to show you how an instance of a record in a DO system looks like.

YOU: OK.

JOE: Let's take as an example, "Watchmen" by Alan Moore and Dave Gibbons, which is my favorite graphic novel. This masterpiece was published in 1987. I'm going to assume that there are two copies of this book in the library, both located on a rack whose ID is `rack-17`, and that one of the two copies is currently out. Here's how I'd represent the `Book` record for "Watchmen" in DO.

Joe comes closer to your laptop, opens a text editor (not an IDE!) and starts typing…

Listing 3.1 An instance of a `Book` record represented as a map

```
{
    "isbn": "978-1779501127",
    "title": "Watchmen",
    "publicationYear": 1987,
    "authors": ["alan-moore", "dave-gibbons"],
    "bookItems": [
        {
            "id": "book-item-1",
            "rackId": "rack-17",
            "isLent": true
        },
        {
            "id": "book-item-2",
            "rackId": "rack-17",
            "isLent": false
        }
    ]
}
```

You look at the laptop screen and ask Joe:

YOU: How am I supposed to instantiate the `Book` record for "Watchmen" **programmatically**?

JOE: It depends on the facilities that your programming language offers to instantiate maps. With dynamic languages like JavaScript, Ruby or Python, it’s straightforward because we can leverage **literals** for maps and arrays.

Listing 3.2 Creating an instance of a `Book` record represented as a map in JavaScript

```
var watchmenBook = {
    "isbn": "978-1779501127",
    "title": "Watchmen",
    "publicationYear": 1987,
    "authors": ["alan-moore", "dave-gibbons"],
    "bookItems": [
        {
            "id": "book-item-1",
            "rackId": "rack-17",
            "isLent": true
        },
        {
            "id": "book-item-2",
            "rackId": "rack-17",
            "isLent": false
        }
    ]
}
```

YOU: And if I’m in Java?

JOE: It’s a bit more tedious, but still doable with the immutable `Map` and `List` static factory methods.[2]:

Listing 3.3 Creating an instance of a `Book` record represented as a map in Java

```
Map watchmen = Map.of(
                      "isbn", "978-1779501127",
                      "title", "Watchmen",
                      "publicationYear", 1987,
                      "authors", List.of("alan-moore", "dave-gibbons"),
                      "bookItems", List.of(
                                           Map.of(
                                                  "id", "book-item-1",
                                                  "rackId", "rack-17",
                                                  "isLent", true
                                                  ),
                                           Map.of (
                                                   "id", "book-item-2",
                                                   "rackId", "rack-17",
                                                   "isLent", false
                                                   )
                                           )
                      );
```

TIP In DO, we represent a record as a heterogeneous map with string keys.

YOU: I’d definitely prefer to instantiate a `Book` record out of a `Book` and a `BookItem` class.

You open your JavaScript IDE and you start typing…

Listing 3.4 Creating an instance of a `Book` record represented as an instance of a `Book` class in JavaScript

```
class Book {
    isbn;
    title;
    publicationYear;
    authors;
    bookItems;
    constructor(isbn, title, publicationYear, authors, bookItems) {
        this.isbn = isbn;
        this.title = title;
        this.publicationYear = publicationYear;
        this.authors = authors;
        this.bookItems = bookItems;
    }
}

class BookItem {
    id;
    rackId;
    isLent;
    constructor(id, rackId, isLent) {
        this.id = id;
        this.rackId = rackId;
        this.isLent = isLent;
    }
}

var watchmenBook = new Book("978-1779501127",
                            "Watchmen",
                            1987,
                            ["alan-moore", "dave-gibbons"],
                            [new BookItem("book-item-1", "rack-17", true),
                             new BookItem("book-item-2", "rack-17", false)]);
```

JOE: Why do you prefer **classes over maps** for representing records?

YOU: It makes the data shape of the record **part of my program**. As a result, the IDE can auto-complete field names, and errors are caught at compile time.

JOE: Fair enough. Would you let me show you some drawbacks of this approach?

YOU: Sure.

JOE: Imagine that you want to display the information about a book in the context of search results. In that case, instead of author IDs, you want to display author names and you don't need the book item information. How would you handle that?

YOU: I'd create a class `BookInSearchResults` without a `bookItems` member, and with an `authorNames` member instead of the `authorIds` member of the `Book` class. Also, I would need to write a copy constructor that receives a `Book` object.

JOE:The fact that in classic OO, data is instantiated only via classes brings safety. But this safety comes at the cost of flexibility.

YOU: How can it be different?

TIP There's a trade-off between flexibility and safety in a data model.

JOE: In the DO approach, where records are represented as maps, we don't need to create a class for each variation of the data. We're free to add, remove and rename record fields dynamically. Our data model is **flexible**.

YOU: Interesting!

TIP In DO, the data model is flexible. We're free to add, remove and rename record fields dynamically, at runtime.

JOE: Now, let me talk about **genericity**: How would you serialize to JSON the content of a `Book` object?

TIP In DO, records are manipulated with generic functions.

YOU: Oh no! I had a nightmare about JSON serialization when I was developing the first version of the Library Management system (see Chapter 1).

JOE: Well, in DO, serializing a record to JSON is super easy.

YOU: Does it involve *reflection* to go over the fields of the record?

JOE: Not at all! Remember that in DO, a record is nothing more than data. We can write a generic JSON serialization function that works with any record. It can be a `Book`, an `Author`, a `BookItem`, or anything else.

YOU: Amazing!

TIP In DO, you get JSON serialization for free.

JOE: Actually, as I'll show you in a moment, lots of data manipulation stuff can be done using generic functions.

YOU: Are the generic functions part of the language?

JOE: It depends on the functions and on the language. For example, JavaScript provides a JSON serialization function called `JSON.stringify()` out of the box, but none for omitting multiple keys or for renaming keys.

YOU: That’s annoying.

JOE: Not so much. There are third-party libraries that provide data-manipulation facilities. A popular data-manipulation library in the JavaScript ecosystem is Lodash.[3]

YOU: And in Java?

JOE: There exist ports of Lodash to Java,[4] to C#,[5] to Python,[6] and to Ruby.[7]

YOU: Cool!

JOE: Actually, Lodash and its **rich set of data manipulation functions** can be ported to any language! That’s why it’s so beneficial to represent records as maps!

TIP DO compromises on data safety to gain flexibility and genericity.

Table 3.1 Tradeoff between safety, flexibility and genericity

	OO	DO
Safety	high	low
Flexibility	low	high
Genericity	low	high

3.4 Manipulate data with generic functions

JOE: Now, let me show you how we manipulate data in DO with generic functions.

YOU: Yes, I’m quite curious to see how you’ll implement the search functionality of the Library Management system.

JOE: OK. First, let’s instantiate, according to your data model from Figure 3.3, a `Catalog` record for the catalog data of a library, where we have a single book, "Watchmen":

Listing 3.5 A `Catalog` record

```
var catalogData = {
    "booksByIsbn": {
        "978-1779501127": {
            "isbn": "978-1779501127",
            "title": "Watchmen",
            "publicationYear": 1987,
            "authorIds": ["alan-moore", "dave-gibbons"],
            "bookItems": [
                {
                    "id": "book-item-1",
                    "rackId": "rack-17",
                    "isLent": true
                },
                {
                    "id": "book-item-2",
                    "rackId": "rack-17",
                    "isLent": false
                }
            ]
        }
    },
    "authorsById": {
        "alan-moore": {
            "name": "Alan Moore",
            "bookIsbns": ["978-1779501127"]
        },
        "dave-gibbons": {
            "name": "Dave Gibbons",
            "bookIsbns": ["978-1779501127"]
        }
    }
}
```

YOU: I see the two indexes we talked about, `booksByIsbn` and `authorsById`. How do you differentiate a record from an index in DO?

JOE: In an entity diagram, there's a clear distinction between records and indexes. But in our code, both are plain data.

YOU: I guess that's why this approach is called **Data**-Oriented Programming.

JOE: Notice how straightforward it is to visualize any part of the system data inside a program. The reason is that **data is represented as data**!

YOU: It sounds like a lapalissade.[8]

TIP In DO, data is represented as data.

JOE: Indeed, it's obvious, but usually in OO, data is represented by objects, which makes it more challenging to visualize data inside a program.

TIP In DO, we can visualize any part of the system data.

YOU: How would you retrieve the title of a specific book from the catalog data?

JOE: That's a great question. In fact, in a DO system, every piece of information has a *path* from which we can retrieve the information.

YOU: I don't get that.

JOE: For example, the path to the title of the "Watchmen" book in the catalog is: `["booksByIsbn", "978-1779501127", "title"]`.

YOU: So what?

JOE: Once we have the path of a piece of information, we can retrieve the information with Lodash's `_.get()` function:

Listing 3.6 Retrieving the title of a book from its path

```
_.get(catalogData, ["booksByIsbn", "978-1779501127", "title"])
```

YOU: Does it work smoothly in a **statically-typed** language like Java?

JOE: It depends whether you need only to pass the value around or to concretely access the value.

YOU: I don't follow.

JOE: Imagine that once you get the title of a book, you want to convert the string into an upper-case string. Then you need to do a static cast to `String`.

Listing 3.7 Casting a field value to a string, in order to manipulate it as a string

```
((String)watchmen.get("title")).toUpperCase()
```

YOU: It makes sense. The values of the map are of different types. Thus, the compiler declares it as a `Map<String,Object>`. The information of the type of the field is lost.

JOE: It's a bit annoying, but quite often, the code just passes the data around. So we don't have to deal too much with static casting.

TIP **In statically-typed languages, we sometimes need to statically cast the field values.**

YOU: What about **performance**?

JOE: In most programming languages, maps are quite effective. Accessing a field in a map is slightly slower than accessing a class member. Usually, this is not significant.

TIP **No significant performance hit by accessing a field in a map instead of a class member.**

YOU: Let's get back to this idea of information path. In OO also, I'd be able to access the title of the "Watchmen" book with `catalogData.booksByIsbn["978-1779501127"].title`. Class members for record fields and strings for index keys.

JOE: There's a fundamental difference. When records are represented as maps, the information can be retrieved via its path using a **generic** function like `_.get()`. But when records are represented as objects, you need to write **specific** code for each type of information path.

YOU: What do you mean by *specific* code? What's specific in `catalogData.booksByIsbn["978-1779501127"].title`?

JOE: In a **statically-typed** language like Java, to write this piece of code, you need to import the class definitions for `Catalog` and `Book`.

YOU: And in a **dynamically-typed** language like JavaScript?

JOE: Even in JavaScript, when you represent records with objects instantiated from classes, you cannot easily write a function that receives a path as an argument and display the information that corresponds to this path. You would have to write **specific code** for each kind of path. You'd access class members with dot notation and map fields with bracket notation.

YOU: Would you say that in DO, the **information path** is a **first-class citizen**?

JOE: Absolutely! The information path is a first-class citizen. It can be stored in a variable and passed as an argument to a function.

TIP **In DO, you can retrieve every piece of information via a path and a generic function.**

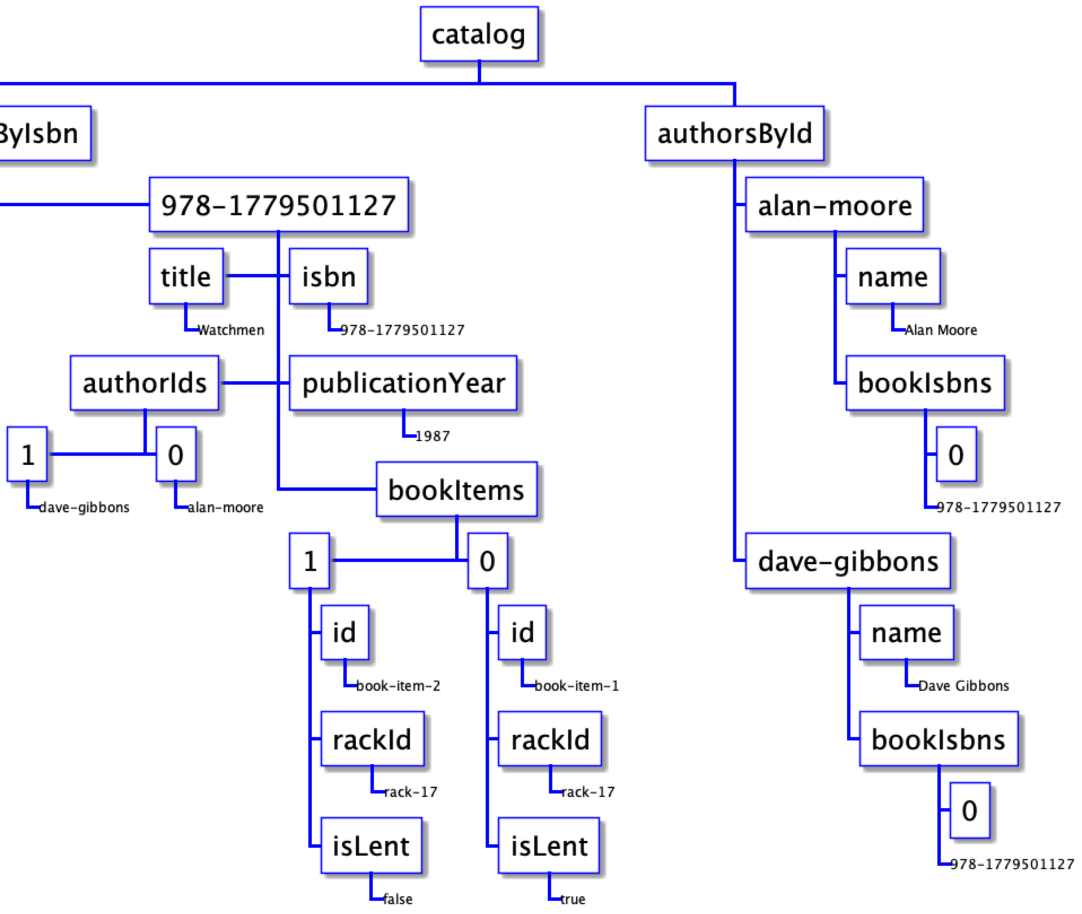


Figure 3.5 The catalog data as a tree. Each piece of information is accessible via a path made of strings and integers. For example, the path of Alan Moore's first book of is ["catalog", "authorsById", "alan-moore", "bookIsbns", 0].

3.5 Calculate search results

YOU: I am starting to feel the **power of expression** of DO.

JOE: Wait. It's just the beginning. Let me show you how simple it is to write code that retrieves the book information and displays it in search results. Can you tell me exactly what information has to appear in search results?

YOU: In the context of search results, the book information should contain `isbn`, `title` and `authorNames`.

JOE: Can you try to write down how a `BookInfo` record would look like for "Watchmen"?

YOU: Sure, here you go…

Listing 3.8 A `BookInfo` record for Watchmen in the context of search result

```
{
    "title": "Watchmen",
    "isbn": "978-1779501127",
    "authorNames": [
        "Alan Moore",
        "Dave Gibbons",
    ]
}
```

JOE: Now, I'm going to show you, step by step, how to write a function that returns search results matching a title in JSON format, using **generic data manipulation functions** from Lodash.

YOU: Cool!

JOE: Let's start with an `authorNames()` function that calculates the author names of a `Book` record by looking at the `authorsById` index. The information path for the name of an author is `["authorsById", authorId, "name"]`.

Listing 3.9 Calculating the author names of a book

```
function authorNames(catalogData, book) {
    var authorIds = _.get(book, "authorIds");
    var names = _.map(authorIds, function(authorId) {   ❶
        return _.get(catalogData, ["authorsById", authorId, "name"]);
    });
    return names;
}
```

❶ can be done with `.forEach()` instead of `.map()`

YOU: What's this `_.map()` function? It smells like Functional Programming stuff! You promised me I don't have to learn FP to implement DO!

JOE: You can use Lodash's `_.forEach()` if you like.

YOU: Yes, I like that. What's next?

JOE: Now, we need a `bookInfo` function that converts a `Book` record into a `BookInfo` record.

Listing 3.10 Converting a `Book` record into a `BookInfo` record

```
function bookInfo(catalogData, book) {
    var bookInfo =  {
        "title": _.get(book, "title"),
        "isbn": _.get(book, "isbn"),
        "authorNames": authorNames(catalogData, book)
    };
    return bookInfo;   ❶
}
```

❶ No need to create a class for `bookInfo`

YOU: Looking at the code, I see that a `BookInfo` record has three fields: `title`, `isbn` and `authorNames`. Is there a way to get this information without looking at the code?

JOE: You can either add it to the data entity diagram or write it in the documentation of the `bookInfo` function, or both.

YOU: I have to get used to the idea that in DO, the record field information is not part of the program.

JOE: Indeed, it's not part of the program, but it gives us a lot of flexibility.

YOU: Is there any way for me to have my cake and eat it, too?!

JOE: Yes. In Part 3, I'll show you how to make record field information as part of a DO program.

YOU: Sounds intriguing!

JOE: Now, we have all the pieces in place to write our `searchBooksByTitle` function that returns book information about the books that match the query. First, we find the `Book` records that match the query (with `_.filter()`), and then we transform each `Book` record into a `BookInfo` record (with `_.map()` and `bookInfo()`). Here's the code:

Listing 3.11 Searching books that match a query

```
function searchBooksByTitle(catalogData, query) {
    var allBooks = _.get(catalogData, "booksByIsbn");
    var matchingBooks = _.filter(allBooks, function(book) {   ❶   ❷
        return _.get(book, "title").includes(query);
    });

    var bookInfos = _.map(matchingBooks, function(book) {   ❷
        return bookInfo(catalogData, book);
    });
    return bookInfos;
}

searchBooksByTitle(catalogData, "Watchmen");
```

❶ when you pass a map to `_.filter()`, it goes over the values of the map

❷ can be done with `_.forEach()`

YOU: It's a bit weird to me that to access a the title of a book record, you write `_.get(book, "title")`. I'd expect it to be `book.title` in dot notation, or `book["title"]` in bracket notation!

JOE: Remember that `book` is a record that's not represented as an object. It's a map. Indeed, in

JavaScript, you can write `_.get(book, "title")`, `book.title` or `book["title"]`. But I prefer to use Lodash's `_.get()`. In some languages, the dot and the bracket notations might not work on maps.

YOU: Are we done with the search implementation?

JOE: Almost. The `searchBooksByTitle` function we wrote is part of the `Catalog` module, and it returns a collection of records. We have to write a function that's part of the `Library` module and that returns a JSON string.

YOU: You told me earlier that **JSON serialization** was straightforward in DO.

JOE: Right. Here's the code for `searchBooksByTitleJSON()`. It retrieves the `Catalog` record, passes it to `searchBooksByTitle()`, and converts the results to JSON with `JSON.stringify()` (that's part of JavaScript).

Listing 3.12 Searching books in a library as JSON

```
function searchBooksByTitleJSON(libraryData, query) {
    var results = searchBooksByTitle(_.get(libraryData, "catalog"), query);
    var resultsJSON = JSON.stringify(results);
    return resultsJSON;
}
```

YOU: How are we going to combine the four functions that we have written so far?

JOE: The functions `authorNames`, `bookInfo` and `searchBooksByTitle` go into the `Catalog` module, and `searchBooksByTitleJSON` goes into the `Library` module.

You look at the resulting code of the two modules, quite amazed by the **conciseness** of the code.

Listing 3.13 Calculating search results. The code is split in two modules: `Library` and `Catalog`.

```
class Catalog {
    static authorNames(catalogData, book) {
        var authorIds = _.get(book, "authorIds");
        var names = _.map(authorIds, function(authorId) {   ❶
            return _.get(catalogData, ["authorsById", authorId, "name"]);
        });
        return names;
    }

    static bookInfo(catalogData, book) {
        var bookInfo =  {
            "title": _.get(book, "title"),
            "isbn": _.get(book, "isbn"),
            "authorNames": Catalog.authorNames(catalogData, book)
        };   ❷
        return bookInfo;
    }

    static searchBooksByTitle(catalogData, query) {
        var allBooks = _.get(catalogData, "booksByIsbn");
        var matchingBooks = _.filter(allBooks, function(book) {   ❶   ❸
            return _.get(book, "title").includes(query);
        });
        var bookInfos = _.map(matchingBooks, function(book) {   ❶
            return Catalog.bookInfo(catalogData, book);
        });
        return bookInfos;
    }
}

class Library {
    static searchBooksByTitleJSON(libraryData, query) {
        var catalogData = _.get(libraryData, "catalog");
        var results = Catalog.searchBooksByTitle(catalogData, query);
        var resultsJSON = JSON.stringify(results);   ❹
        return resultsJSON;
    }
}
```

❶ can be done with `_.forEach()`

❷ no need to create a class for `bookInfo`

❸ when `_.filter()` is passed a map, it goes over the values of the map

❹ converts data to JSON (part of JavaScript)

YOU: Let’s check whether the code works as expected.

JOE: Sure. For that, we need to create a `Library` record that contains our `Catalog` record.

Listing 3.14 The library data (without user management data)

```
var libraryData = {
    "name": "The smallest library on earth",
    "address": "Here and now",
    "catalog": {
        "booksByIsbn": {
            "978-1779501127": {
                "isbn": "978-1779501127",
                "title": "Watchmen",
                "publicationYear": 1987,
                "authorIds": ["alan-moore",
                              "dave-gibbons"],
                "bookItems": [
                    {
                        "id": "book-item-1",
                        "rackId": "rack-17",
                        "isLent": true
                    },
                    {
                        "id": "book-item-2",
                        "rackId": "rack-17",
                        "isLent": false
                    }
                ]
            }
        },
        "authorsById": {
            "alan-moore": {
                "name": "Alan Moore",
                "bookIsbns": ["978-1779501127"]
            },
            "dave-gibbons": {
                "name": "Dave Gibbons",
                "bookIsbns": ["978-1779501127"]
            }
        }
    },
    "userManagement": {
        // omitted for now
    }
};
```

YOU: Let’s search for books with titles that match `"Watchmen"`.

Listing 3.15 Search results in JSON

```
Library.searchBooksByTitleJSON(libraryData, "Watchmen");
// returns "[{\"title\":\"Watchmen\",\"isbn\":\"978-1779501127\",\"authorNames\":[\"Alan Moore\",
//              \"Dave Gibbons\"]}]"
```

You look again at the source code from Listing 3.12… After a few seconds, you feel like you’re in an Aha! moment.

YOU: The important thing is not that the code is **concise**, but that the code contains **no abstractions**. It’s just data manipulation!

Joe responds with a smile that says, "You got it, my friend!"

JOE: It reminds me what my first meditation teacher told me 10 years ago: Meditation guides the mind to grasp the reality as it is, without the abstractions created by our thoughts.

TIP **In DO, many parts of our code base tend to be just about data manipulation with no abstractions.**

3.6 Handle records of different types

We've seen how DO enables us to treat records as **first class citizens** that can be manipulated in a **flexible** way using **generic** functions. But if a record is nothing more than an aggregation of fields, how do we know what the **type** of the record is?

DO has a surprising answer to this question.

YOU: I have a question. If a record is nothing more than a *map*, how do you know the **type of the record**?

JOE: That's a great question with a **surprising** answer.

YOU: I'm curious.

JOE: Most of the time, there's **no need to know the type of the record**.

YOU: What do you mean?

JOE: I mean that what matter most are the values of the fields. For example, take a look at the `Catalog.authorNames()` source code in Listing 3.15. It operates on a `Book` record, but the only thing that matters is the value of the `authorIds` field.

Doubtful, you look at the source code of `Catalog.authorNames`.

Listing 3.16 Calculating the author names of a book

```
function authorNames(catalogData, book) {
    var authorIds = _.get(book, "authorIds");
    var names = _.map(authorIds, function(authorId) {   ❶
        return _.get(catalogData, ["authorsById", authorId, "name"]);
    });
    return names;
}
```

❶ can be done with `.forEach()` instead of `.map()`

YOU: What about **differentiating** between various user types like `Member` vs `Librarian`? I mean, they both have `email` and `encryptedPassword`. How do you know if a record represents a `Member` or a `Librarian`?

JOE: You check if the record is found in the `librariansByEmail` index or in the `membersByEmail` index of the `Catalog`.

YOU: Could you be more specific?

JOE: Sure. Let me write down how the user management data of our tiny library might look like, assuming that we have one librarian and one member. To keep things simple, I am encrypting passwords through naive base-64 encoding.

Listing 3.17 A UserManagement record

```
var userManagementData = {
    "librarians": {
        "franck@gmail.com" : {
            "email": "franck@gmail.com",
            "encryptedPassword": "bXlwYXNzd29yZA=="   ❶
        }
    },
    "members": {
        "samantha@gmail.com": {
            "email": "samantha@gmail.com",
            "encryptedPassword": "c2VjcmV0",   ❷
            "isBlocked": false,
            "bookLendings": [
                {
                    "bookItemId": "book-item-1",
                    "bookIsbn": "978-1779501127",
                    "lendingDate": "2020-04-23"
                }
            ]
        }
    }
}
```

❶ base-64 encoding of "mypassword"

❷ base-64 encoding of "secret"

TIP **Most of the time, there’s no need to know what the type of a record is.**

YOU: I remember that in Chapter 2, you told me you’ll show me the code for `UserManagement.isLibrarian()` function in Chapter 3.

JOE: So here we are in Chapter 3, and I’m going to fulfill my promise:

Listing 3.18 Checking if a user is a librarian

```
function isLibrarian(userManagement, email) {
    return _.has(_.get(userManagement, "librariansByEmail"), email);
}
```

YOU: OK. You simply check if the `librariansByEmail` map contains the `email` field.

JOE: Yep.

YOU: Would you use the same pattern to check if a member is a Super member or a VIP member?

JOE: We can, indeed, have `SuperMembersByEmail` and `VIPMembersByEmail` indexes. But there's a better way.

YOU: How?

JOE: When a member is a VIP member, we add a field, `isVIP` with the value `true`, to its record. To check if a member is a VIP member, check whether the `isVIP` field is set to `true` in the member record:

Listing 3.19 Checking if a member is a VIP member

```
function isVIPMember(userManagement, email) {
    return _.get(userManagement, ["membersByEmail", email, "isVIP"]) == true;
}
```

YOU: I see that you access the `isVIP` field via its information path: `["membersByEmail", email, "isVIP"]`.

JOE: Yes. I think it makes the code crystal clear.

YOU: Agree. And I guess that we can do the same and have an `isSuper` field set to `true` when a member is a Super member?

JOE: Yes. Just like this:

Listing 3.20 The code of `UserManagement` module

```
class UserManagement {
    isVIPMember(userManagement, email) {
        return _.get(userManagement, ["membersByEmail", email, "isVIP"]) == true;
    }

    isSuperMember(userManagement, email) {
        return _.get(userManagement, ["membersByEmail", email, "isSuper"]) == true;
    }
}
```

You look at the `UserManagement` module code for a couple of seconds, and suddenly an idea comes to you…

YOU: Why not have a `type` field in member record, whose value would be either `VIP` or `Super`?

JOE: I assume that, according to the product requirements, a member can be both a VIP and a Super member.

YOU: Hmm… We can have a `types` field that will be a collection of either `VIP` or `Super`.

JOE: In some situations, having a `types` field is helpful, but I find it simpler to have a *boolean* field for each feature that the record supports.

YOU: Is there a name for fields like `isVIP` and `isSuper`?

JOE: I call them **feature fields**.

TIP **Instead of maintaining type information about a record, use a feature field (e.g., `isVIP`).**

YOU: Can we use feature fields to differentiate between librarians and members?

JOE: You mean having an `isLibrarian` and an `isMember` field?

YOU: Yes, and having a common `User` record type for both librarians and members.

JOE: We can, but I think it's simpler to have different record types for librarians and members: `Librarian` for librarians, and `Member` for members.

YOU: Why?

JOE: Because there's a clear distinction between librarians and members in terms of data. For example, members have book lendings but librarians don't.

YOU: I agree. Now, we need to mention the two `Member` feature fields in our entity diagram:

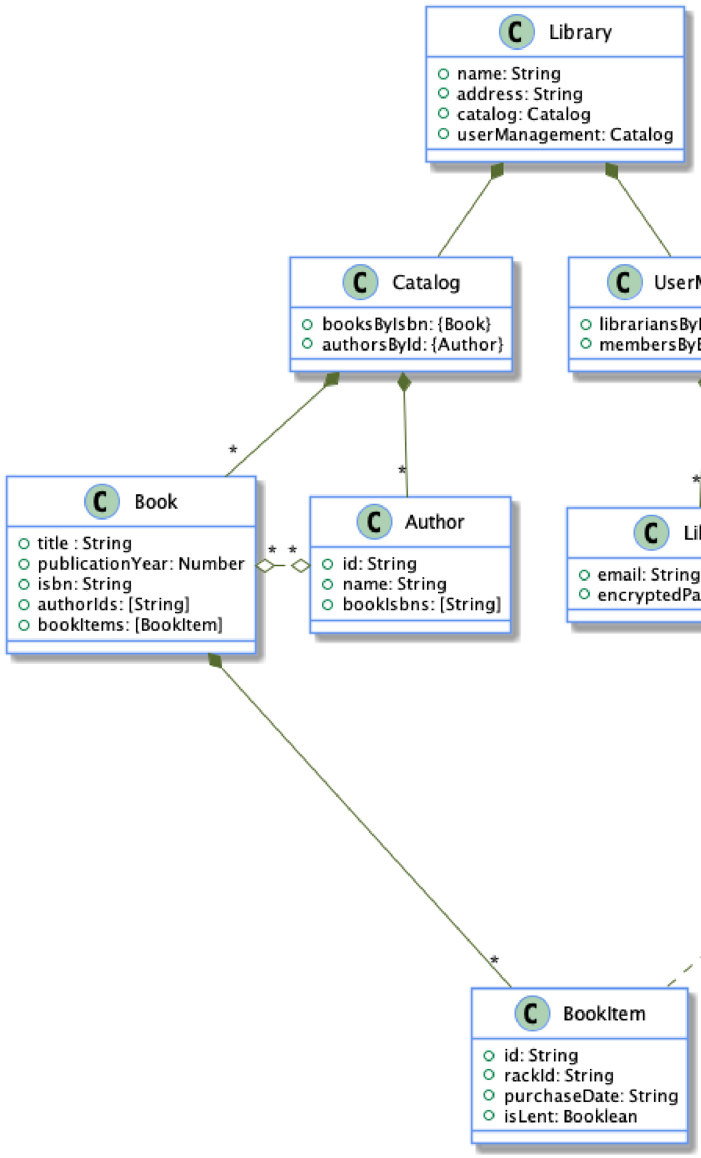


Figure 3.6 Library management data model, with Member feature fields isVIP and isSuper.

JOE: Do you like the data model that we have designed together?

YOU: I find it quite **simple** and **clear**.

JOE: That’s the main goal of DO.

YOU: Also, I’m pleasantly surprised how easy it is to adapt to changing requirements both in terms of code and data model.

JOE: I suppose you’re also happy to get rid of complex **class hierarchy diagrams**.

YOU: Absolutely! Feature fields feel much simpler to deal with than class inheritance.

JOE: Avoiding inheritance in records keeps our **data model simple**.

YOU: Are there more benefits of representing records with maps?

JOE: Yes. We can quite easily deal with advanced data inspection stuff. I’ll tell you more about that in Chapter 5.

YOU: Why not in the next Chapter?

JOE: Because I have something more fundamental to tell you about.

YOU: What’s that?

JOE: How to manage state in DO without mutating the data.

3.7 Wrapping up

In this chapter, we explored the benefits of **representing records with string maps**.

The data part of our system is **flexible**, and each piece of information is accessible via its **information path**. We manipulate data with **generic functions**, which are provided either by the language itself or by third-party libraries like Lodash. As an example, you get **JSON serialization** for free.

On one hand, we’ve lost the safety of accessing record fields via members defined at compile time. On the other hand, we’ve **liberated** data from the limitation of classes and objects. Data is represented as data!

When data is not represented by objects, we’re free to **visualize** every part of the system.

Instead of maintaining **type information** about a record, we use a **feature field**.

In a DO system, the dependency between code and data is **weak**. It’s all about record field names. Weak dependency makes it is **easier to adapt** to changing requirements.

4 State management with immutable data

4.1 Introduction

So far we have seen how DO deals with requests that **query information** about the system, via **generic** functions that access the system data, represented as a **hash map**.

In this chapter and the following one, we illustrate how DO deals with **mutations**, i.e. requests that **change** the **system state**. Instead of updating the state in place, we maintain **multiple versions** of the system data. At a specific point in time, the system state refers to a specific version of the system data.

The maintenance of multiple versions of the system data requires the data to be immutable. This is made **efficient** both in terms of **computation** and **memory** via a technique called **Structural Sharing**, where parts of the data that are common between two versions are shared instead of being copied.

In DO, a mutation is split into two distinct phases:

1. In the **Calculation phase**, we **compute** the next version of the system data.
2. In the **Commit phase**, we **move forward** the system state so that it **refers** to the version of the system data computed by the Calculation phase.

This distinction between Calculation and Commit phases allows us to reduce the part of our system that is stateful to its bare minimum. Only the code of the Commit phase is **stateful**, while the code in the Calculation phase of a mutation is **stateless** and made of **generic** functions similar to the code of a query.

The implementation of the Commit phase is common to all the mutations. As a consequence, inside the Commit phase, we have the ability to ensure that the state always refers to a **valid version** of the system data.

Another benefit of this state management approach is that we can keep track of **the history of previous versions** of the system data. If needed, restoring the system to a previous state is straightforward.

Table 4.1 The two phases of a mutation

Phase	Responsibility	State	Implementation
Calculation	Compute next version of system data	Stateless	Specific
Commit	Move forward the system state	Stateful	Common

In the present chapter, we assume that no mutations occur concurrently in our system. In the next chapter, we will deal with concurrency control.

4.2 Multiple versions of the system data

During the coffee break, you and Joe go for a walk around the block and this time the discussion turns around version control systems. You discuss about how git keeps track of the whole commit history and how easy and fast it is to restore the code to a previous commit. You discuss also about commit hooks that allows to validate the code before it is committed.

JOE: So far we have seen how in DO, we manage **queries** that retrieve information from the system. Now I am going to show you how we manage **mutations**. By a mutation, I mean an operation that changes the state of the system.

NOTE A mutation is an operation that changes the state of the system.

YOU: Is there a fundamental difference between queries and mutations in DO? After all, the whole state of the system is represented as a hash map. I could easily write code that modifies part of the hash map. It would be similar to the code that retrieves information from the hash map.

JOE: You could mutate the data in place, but then it would be challenging to make sure that the code of a mutation doesn't put the system into an invalid date. Also you would lose track of previous versions of the system state.

YOU: I see. So how do you handle mutations in DO?

JOE: We adopt a **multi-version state** approach, similar to what a version control like git does. We manage different versions of the system data. At a specific point in time, the state of the system refers to a version of the system data. After a mutation is executed, we move forward the reference.

YOU: I am confused: is the system state mutable or immutable?

JOE: The data is **immutable** but the state reference is **mutable**.

TIP The data is immutable but the state reference is mutable.

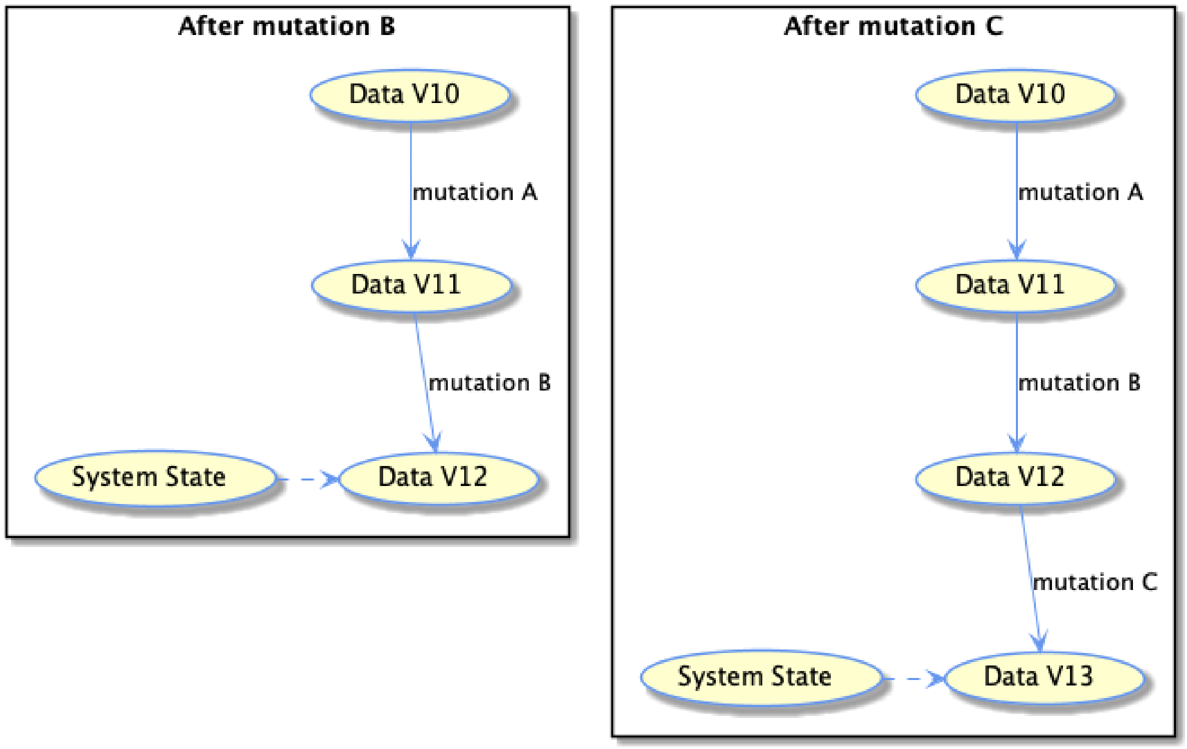


Figure 4.1 After mutation B is executed, the system state refers to Data V12. After mutation C is executed, the system state refers to Data V13.

YOU: Does it mean that before the code of a mutation runs, we make a copy of the system data?

JOE: No. That would be very inefficient, as we would have to do a deep copy of the data.

YOU: So how does it work?

JOE: It works by using a technique called **structural sharing**, where most of the data between subsequent versions of the state is **shared** instead of being copied. This technique allows to efficiently create new versions of the system data, both in terms of **memory** and **computation**.

YOU: I am intrigued.

TIP With structural sharing, it is efficient (in terms of memory and computation) to create new versions of data.

JOE: I’ll explain you in details how **structural sharing** works in a moment.

You take another look at the diagram in Figure 4.1 that illustrates how the system state refers to a version of the system data and suddenly a question emerges in your mind.

YOU: Are the **previous versions** of the system data kept?

JOE: In a simple application, previous versions are automatically removed by the **garbage collector**. But in some cases, we maintain **historical references** to previous versions of the data.

YOU: What kind of cases?

JOE: For example, we can allow **time travel** in our system. Like in git, we can move back the system to a previous version of the state very easily.

YOU: Now, I understand what you meant by: *The data is immutable but the state reference is mutable.*

4.3 Structural sharing

As we mentioned in the previous section, structural sharing allows to **efficiently** create new versions of immutable data. In DO, we leverage **structural sharing** in the **Calculation** phase of a **mutation** to compute the next state of the system based of the current state of the system. Inside the calculation phase, we don't have to deal with state management: this is delayed to the Commit phase. As a consequence, the code involved in the calculation phase of a mutation is **stateless** and is as simple as the code of a query.

YOU: I am really intrigued by this efficient way to create new version of data. How does it work?

JOE: Let's take a simple example from our library system. Imagine that you want to modify the value of a field in a book in the catalog, for instance the publication year of Watchmen. Can you tell me what is the information path for Watchmen publication year?

After a a quick look at the catalog data in Figure 4.2, you answer:

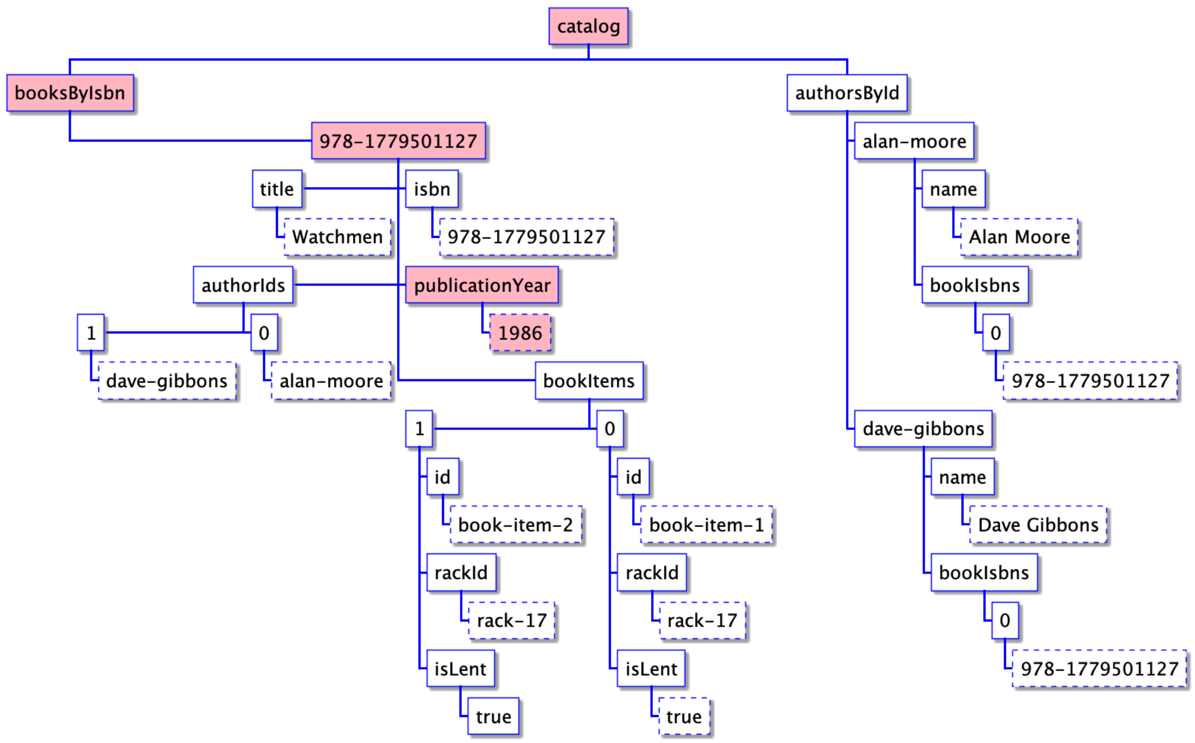


Figure 4.2 An updated version of the library

YOU: The information path for Watchmen publication year is: `["catalog", "booksByIsbn", "978-1779501127", "publicationYear"]`.

JOE: Now, let me show how to use the **immutable function** `_.set()` provided by Lodash.

YOU: What do you mean by an immutable function? When I look at Lodash documentation for `_.set()`,[9] it says that it mutates the object.

JOE: You are right. By default Lodash functions are not immutable. In order to use a immutable version of the functions, we need to use Lodash FP module (Functional Programming), as it is explained in the Lodash FP guide.[10]

YOU: Do the immutable functions have the same signature as the mutable functions?

JOE: By default, the order of the arguments in immutable functions is shuffled. In the Lodash FP guide, they explain how to resolve it: with this piece of code in Listing 4.1 the signature of the immutable functions is exactly the same as the mutable functions.

Listing 4.1 Configuring Lodash so that the immutable functions have the same signature as the mutable functions

```
_ = fp.convert({
    "cap": false,
    "curry": false,
    "fixed": false,
    "immutable": true,
    "rearg": false
});
```

TIP In order to use Lodash immutable functions, we use Lodash FP module and we configure it so that the signature of the immutable functions is the same as in the Lodash documentation web site.

YOU: So basically, I can still rely on Lodash documentation when using immutable versions of the functions.

JOE: Except for the piece in the documentation that says the function mutates the object.

YOU: Of course!

JOE: Now, let me show you how to write code that creates a version of the library data with the **immutable function** `_.set()` provided by Lodash.

Listing 4.2 Creating a version of the library where Watchmen publication year is 1986

```
var nextLibrary = _.set(library, ["catalog", "booksByIsbn",
                                  "978-1779501127", "publicationYear"],
                        1986);
```

NOTE A function is said to be immutable when instead of mutating the data, it creates a new version of the data without changing the data it receives.

YOU: You told me earlier that structural sharing allowed immutable functions to be **efficient** in terms of **memory** and **computation**. Could you tell me what make them efficient?

JOE: With pleasure. But before that you'd have to answer a series of questions. Are you ready?

YOU: Yes.

JOE: What part of the library data is impacted by updating Watchmen publication year: the `UserManagement` or the `Catalog`?

YOU: Only the `Catalog`.

JOE: What part of the `Catalog`?

YOU: Only the `booksByIsbn` index.

JOE: What part of the `booksByIsbn` index?

YOU: Only the `Book` record that holds the information about Watchmen.

JOE: What part of the `Book` record?

YOU: Only the `publicationYear` field.

JOE: When you use an **immutable function** to create a new version of the `Library` where the publication year of Watchmen is set to 1986 (instead of 1987), it creates a fresh `Library` hash map that **recursively** uses the parts of the current `Library` that are **common** between the two versions instead of deeply copying them. This technique is called: **structural sharing**.

YOU: Could you describe me how structural sharing works step by step?

Joe grabs a piece of paper and draws the diagram in Figure 4.3 that illustrates structural sharing.

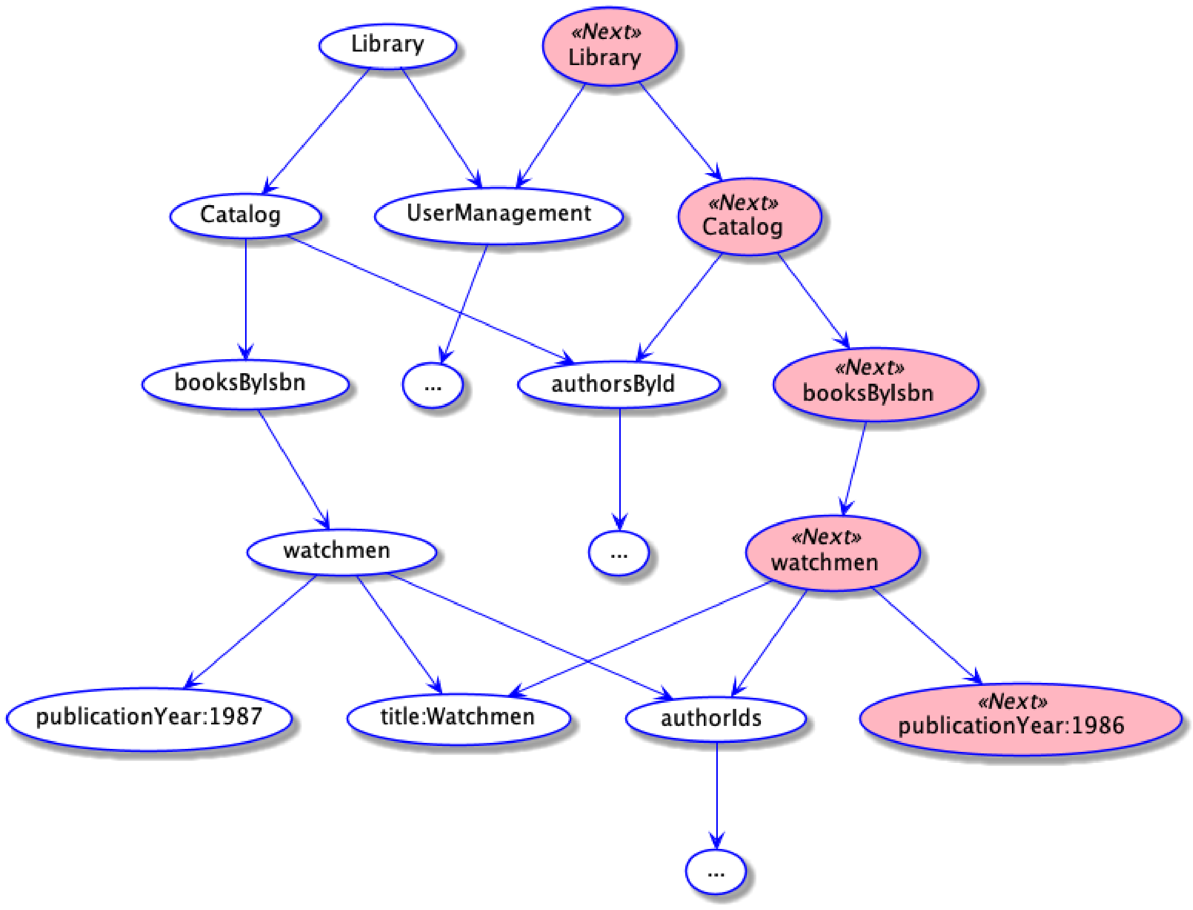


Figure 4.3 Structural sharing provides an efficient way to create a new version of the data: Next Library is recursively made of nodes that uses the parts of Library that are common between the two.

JOE: The next version of the `Library`, uses the same `UserManagement` hash map as the old one. The `Catalog` inside the next `Library` uses the same `authorsById` as the current `Catalog`. The Watchmen `Book` record inside the next `Catalog` uses all the fields of the current `Book` except for the `publicationYear` field.

TIP **Structural sharing provides an efficient way (both memory and computation) to create a new version of the data by recursively sharing the parts that don't need to change.**

YOU: That's very cool!

JOE: Indeed. Now let me show you how to write a mutation for adding a member using immutable functions. Figure 4.4 shows a diagram that illustrates how structural sharing looks like when we add a member.

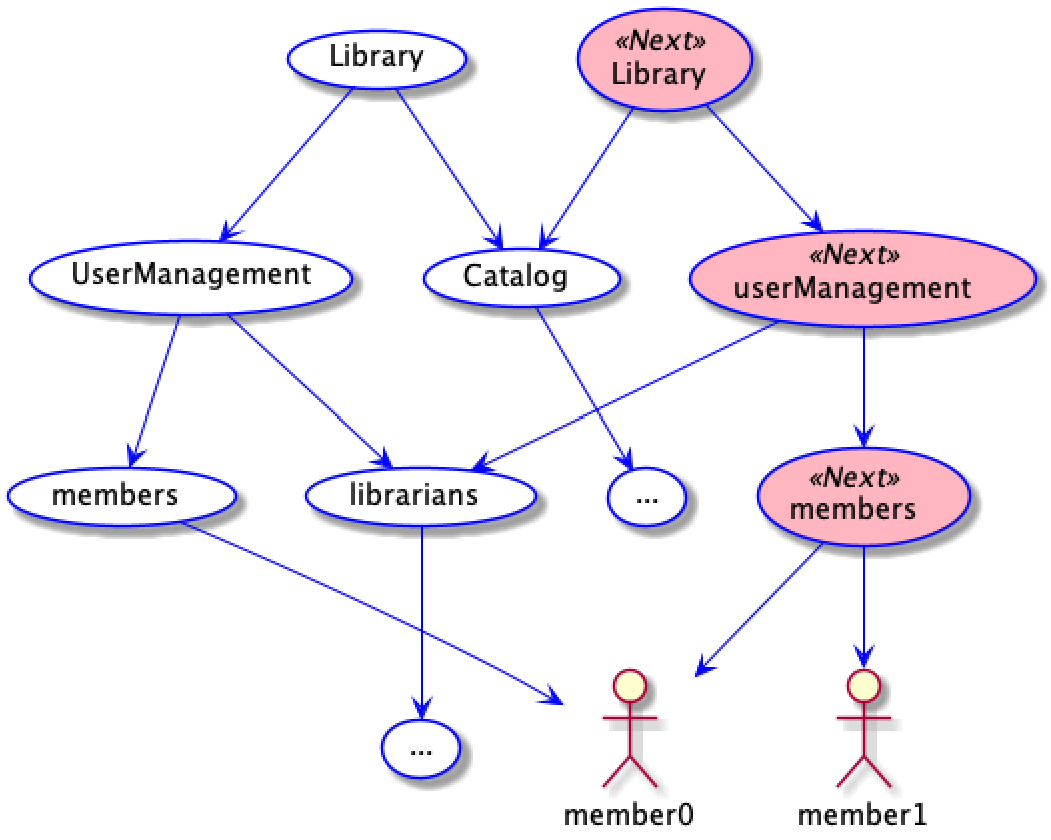


Figure 4.4 Adding a member with structural sharing: most of the data is shared between the two versions

YOU: It's so cool that the `Catalog` and the `librarians` hash maps don't have to be copied!

JOE: In terms of code, we have to write a `Library.addMember()` function that delegates to `UserManagement.addMember()`.

YOU: I guess it is going to be similar to the code we wrote in Chapter 2 to implement the search books query, where `Library.searchBooksByTitleJSON()` delegates to `Catalog.searchBooksByTitle()`.

JOE: Similar in the sense that all the functions are static and they receive the data they manipulate as an argument. But there are two difference: First, A mutation could fail, for instance if the member to be added already exists. Secondly, the code for `Library.addMember()` is a bit more elaborate than the code for `Library.searchBooksByTitleJSON()` as we have to create a new version of the `Library` that refers to the new version of the `UserManagement`. Listing 4.3 shows the code for the mutation that adds a member.

Listing 4.3 The code for the mutation that adds a member

```
UserManagement.addMember = function(userManagement, member) {
    var email = _.get(member, "email");
    var infoPath = ["membersByEmail", email];
    if(_.has(userManagement, infoPath)) {    ❶
        throw "Member already exists.";
    }
    var nextUserManagement =  _.set(userManagement,    ❷
                                    infoPath,
                                    member);
    return nextUserManagement;
}

Library.addMember = function(library, member) {
    var currentUserManagement = _.get(library, "userManagement");
    var nextUserManagement = UserManagement.addMember(currentUserManagement, member);
    var nextLibrary = _.set(library, "userManagement", nextUserManagement);    ❸
    return nextLibrary;
}
```

❶ Check if a member already exists with the same email address

❷ Create a new version of `userManagement` that includes the member

❸ Create a new version of `library` that contains the new version of `userManagement`

YOU: It’s a bit weird to me that immutable functions return an updated version of the data instead of changing it in place.

JOE: It was also weird for me when I first encountered immutable data in Clojure 10 years ago.

YOU: How long did it take you to get used to it?

JOE: A couple of weeks.

4.4 Data safety

YOU: Something is not clear to me regarding this structural sharing stuff. What happens if we write code that modifies the data part that is shared between the two versions of the data? Does the change affect both versions?

JOE: Could you please write a code snippet that illustrates your question?

You start typing on your laptop, and you come up with the code snippet in Listing 4.4 that illustrates your point.

Listing 4.4 A piece of code that modifies a piece of data that is shared between two versions

```
var member = {
    "email": "joe@me.com",
    "password": "secret",
    "isBlocked": true
}

var updatedMember = _.set(member, "password", "hidden");

member["isBlocked"] = false;
```

YOU: My question is: what is the value of `isBlocked` in `updatedMember`?

JOE: The answer is that mutating data via the native hash map setter is **forbidden**. All the data manipulation must be via immutable functions.

WARNING **All data manipulation must be done via immutable functions: It is forbidden to use the native hash map setter.**

YOU: When you say *forbidden* you mean that it's up to the developer to make sure it doesn't happen. Right?

JOE: Exactly.

YOU: Is there a way to protect our system from a developer's mistake?

JOE: Yes, there is a way to ensure the immutability of the data at the level of the data structure. It's called **persistent data structures**.

YOU: Are persistent data stuctures also efficient in terms of memory and computation?

JOE: Actually, the way data is organized inside persistent data structures make them even more efficient than immutable functions.

TIP **Persistent data structures are immutable at the level of the data: There is no way to mutate them (even by mistake).**

YOU: Are there **library** providing persistent data structures?

JOE: Definitely. For example, we have Immutable.js in JavaScript,[11] Paguro in Java,[12] Immutable Collections in C#,[13] Pyrsistent in Python,[14] and Hamster in Ruby.[15]

YOU: So why not using persistent data structures instead of immutable functions?

JOE: The **drawback** of persistent data structures is that they are not native which means that working with them require **conversion** from native to persistent and from persistent to native.

YOU: What approach would you recommend then?

JOE: If you want to play around a bit, then start with immutable functions. But for a production application I'd recommend using persistent data structures.

YOU: So bad native data structures are not persistent!

JOE: That's one of the reasons why I love Clojure: the native data structures of the language are immutable!

4.5 The Commit phase of a mutation

So far we have seen how to implement the **Calculation** phase of a mutation. The Calculation phase is stateless, in the sense that it doesn't make any change to the system. Now, we are going to see how we update the state of the system inside the **Commit** phase.

You take another look at the code for `Library.addMember()` in Listing 4.5 and something bothers you: this function returns a new state of the library that contains an additional member but it doesn't affect the current state of the library!

Listing 4.5 The Calculation phase of a mutation doesn't make any change to the system

```
Library.addMember = function(library, member) {
    var currentUserManagement = _.get(library, "userManagement");
    var nextUserManagement = UserManagement.addMember(currentUserManagement, member);
    var nextLibrary = _.set(library, "userManagement", nextUserManagement);
    return nextLibrary;
};
```

YOU: I see that `Library.addMember()` doesn't change the state of the library. How the library state gets updated then?

JOE: That's an excellent question. `Library.addMember()` deals only with data calculation and is stateless. The state is updated in the **Commit** phase by moving forward the version of the state that the system state refers to.

YOU: What do you mean?

JOE: Here is what happens when we add a member to the system. The **Calculation** phase creates a version of the state that has two members. Before the **Commit** phase, the system state refers to the version of the state with one member. The responsibility of the **Commit** phase is to move the system state forward so that it refers to the version of the state with two members.

TIP **The responsibility of the Commit phase is to move forward the system state to the version of the state returned by the Calculation phase.**

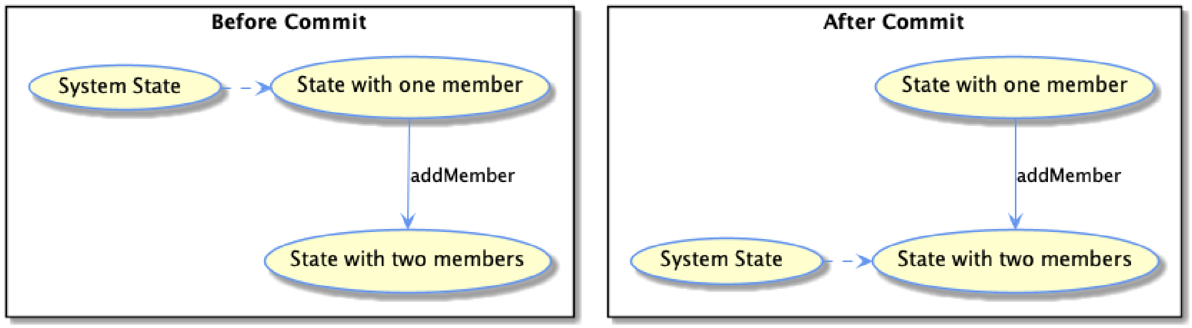


Figure 4.5 The Commit phase moves forward the system state

YOU: How does it look like in terms of code?

JOE: The code is made of 2 classes: `System`, a singleton stateful class that implements the mutations. `SystemState` a singleton stateful class that manages the system state.

YOU: It sounds to me like classic OO.

JOE: Right. This part of the system being stateful is very OO-like.

YOU: I am happy to see that you still find some utility in OO.

JOE: Meditation taught me that *Every piece of our universe has a role to play*.

YOU: Nice! Could you show me some code?

JOE: Sure. Let's start with the `System` class: Listing 4.6 shows the implementation of the `addMember` mutation.

Listing 4.6 The `System` class

```
class System {
    addMember(member) {
        var previous = SystemState.get();
        var next = Library.addMember(previous, member);
        SystemState.commit(previous, next);
    }
}
```

YOU: How does a `SystemState` look like?

JOE: Listing 4.7 shows the code for the `SystemState` class: It is a stateful class!

Listing 4.7 The `SystemState` class

```
class SystemState {
    systemState;

    get() {
        return this.systemState;
    }

    commit(previous, next) {
        this.systemState = next;
    }
}
```

YOU: I don't get the point of the `SystemState`. It's a simple class with a getter and a commit function!

JOE: In a moment, we are going to enrich the code of the `SystemState.commit()` method so that it provides data validation and history tracking. For now, the important thing to notice is that the code of the **Calculation** phase is stateless and it is decoupled from the code of the **Commit** phase which is stateful.

TIP **The Calculation phase is stateless. The Commit phase is stateful.**

4.6 Ensure system state integrity

YOU: Something still bothers me with the way functions manipulate immutable data in the Calculation phase: How do we preserve the **data integrity**?

JOE: What do you mean?

YOU: In OO, the data is manipulated only by methods that belong to the same class as the data. It prevents from other classes to corrupt the inner state of the class.

JOE: Could you give me an example of an invalid state of the library?

YOU: For example imagine that the code of a mutation adds a book item to the book lendings of a member without marking the book item as lent in the catalog. Then the system data would be corrupted.

JOE: In DO, we have the privilege to ensure **data integrity** at the level of the whole system instead of scattering the validation among many classes.

YOU: I don't get that.

JOE: The fact the code for the Commit phase is common to all the mutations allows us to validate the system data in a **central place**: At the beginning of the Commit phase, there is a step that checks (see Listing 4.8) whether the version of the system state to be committed is valid. If the data is invalid, the commit is rejected.

Listing 4.8 Data validation inside the commit phase

```
SystemState.commit = function(previous, next) {
    if(!SystemValidity.validate(previous, next) {
        throw "The system data to be committed is not valid!";
    });
    this.systemData = next;
}
```

YOU: It sounds similar to a commit hook in git.

JOE: I like your analogy!

YOU: Why are you passing to `SystemValidity.validate()` the `previous` in addition to the `next`?

JOE: Because it allows the code of `SystemValidity.validate()` to optimize the validation in terms of computation. For example, we could validate only the part of the data that has changed.

TIP **In DO, we validate the system data as a whole. Data validation is decoupled from data manipulation.**

YOU: How does the code of `SystemValidity.validate()` look like?

JOE: I will you show you in Part 2 how we could for instance make sure that every author id mentioned in a book record is valid. It involves more advanced data manipulation logic.

4.7 Time travel

Another advantage of the **multi-version** state approach with **immutable data** that is manipulated via **structural sharing** is that we can keep track of the **history** of all the versions of the data **without exploding the memory** of our program. It allows us for instance to restore the system back to an earlier state very easily.

YOU: You told me earlier that it was easy to restore the system to a previous state. Could you show me how?

JOE: With pleasure. But before I'd like to make sure you understand why keeping track of all the versions of the data is **efficient** in terms of **memory**.

YOU: I think it's related to the fact that immutable functions use **structural sharing**. And most of the data between subsequent versions of the state is shared.

TIP **Structural sharing allows us to keep many versions of the system state without exploding the memory.**

JOE: Perfect. Now, I am going to show you how simple it is to *undo* a mutation. In order to implement *undo*, our `SystemState` class needs to have two references to the system data: `systemData` references to the current state of the system and `previousSystemData` references to the previous state of the system.

YOU: That makes sense.

JOE: In the Commit phase, we update both `previousSystemData` and `systemData`.

YOU: And what does it take to implement *undo*?

JOE: Undo is achieved by having `systemData` referencing the same version of the system data as `previousSystemData`.

YOU: Could you give me an example?

JOE: To make things simple, I am going to give a number to each version of the system state. It starts at `V0` and each time a mutation is committed the version is incremented: `V1`, `V2`, `V3` etc…

YOU: OK.

JOE: Let's say that currently our system state is at `V12` (see Figure 4.6). In the `SystemState` object, `systemData` refers to `V12` and `previousSystemData` refers to `V11`.

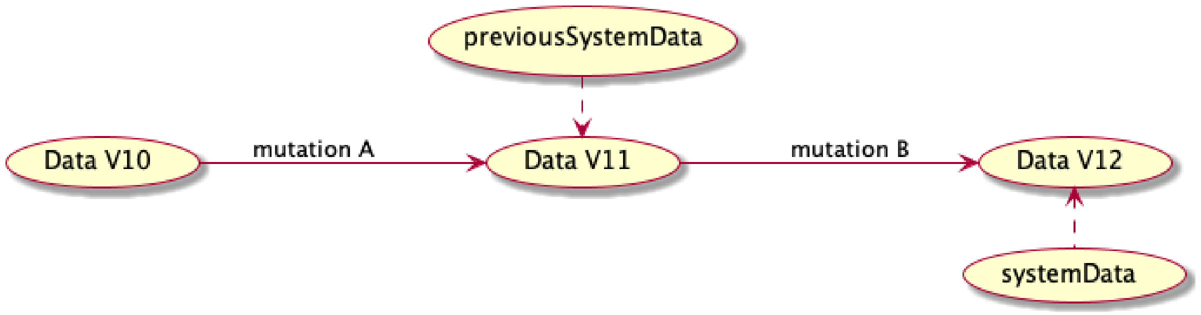


Figure 4.6 When the system state is at V12, systemData refers to V12 and previousSystemData refers to V11

YOU: So far so good.

JOE: Now when a mutation is committed (for instance adding a member), both references move forward: `systemData` refers to `V13` and `previousSystemData` refers to `V12`

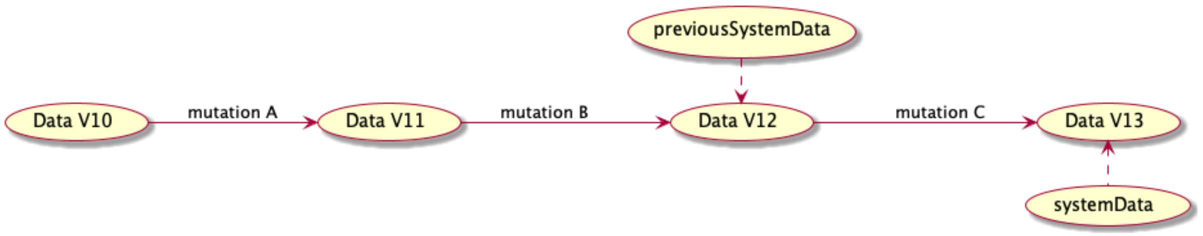


Figure 4.7 When a mutation is committed, systemData refers to V13 and previousSystemData refers to V12

YOU: And I suppose that when we undo the mutation, both references move backward.

JOE: In theory, yes. But in practice, it would require to maintain a stack of all the state references. For now, to simplify things we maintain only a reference to the previous version. As a consequence, when we undo the mutation, both references refer to `V12` as shown in Figure 4.8.

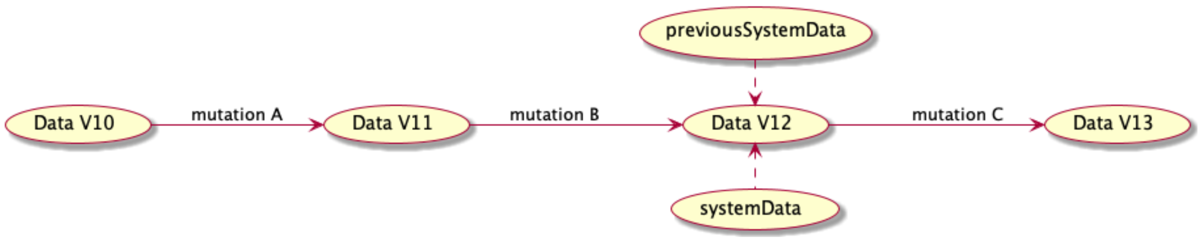


Figure 4.8 When a mutation is undone, both systemData and previousSystemData refer to V12

YOU: Could you show me how to implement this undo mechanism?

JOE: Actually, it takes only a couple of changes to the `SystemState` class. The result is in Listing 4.9.Pay attention to the changes in the `commit()` function: We keep inside `systemDataBeforeUpdate` a reference to the current state of the system. If the validation and the conflict resolution succeed, we update both `previousSystemData` and `systemData`.

Listing 4.9 The `SystemState` class with undo capability

```
class SystemData {
    systemData;
    previousSystemData;

    get() {
        return this.systemData;
    }

    commit(previous, next) {
        var systemDataBeforeUpdate = this.systemData;
        if(!Consistency.validate(previous, next) {
            throw "The system data to be committed is not valid!";
        });
        this.systemData = next;
        this.previousSystemData = systemDataBeforeUpdate;
    }

    undoLastMutation() {
        this.systemData = this.previousSystemData;
    }
}
```

YOU: And I see that implementing `System.undoLastMutation()` is simply a matter of having `systemData` refers the same value as `previousSystemData`.

JOE: As I told you, if we need to allow multiple undos, the code would be a bit more complicated. But you get the idea.

4.8 Wrapping up

In this chapter, we have explored how DO **manages state** via a multi-version approach, where the mutation is split into Calculation and Commit phases.

During the **Calculation phase**, the data is manipulated with **immutable functions** that leverage **structural sharing** to **efficiently** (memory and computation) create a **new version** of the data where the data that is common between the two versions is **shared** instead of being copied.

Moving forward the **state reference** occurs in the **Commit** phase which is the only part of our system that is **stateful**. The fact that the code for the **Commit phase** is common to all the mutations, allows us to **validate** the system state in a central place before we update the state.

Moreover, it is easy and efficient to keep the **history** of the versions of the system data and restoring the system to a previous state is straightforward. As an example, we have seen how to implement *undo* in a DO system.

Notes

1. The International Standard Book Number (ISBN) is a numeric commercial book identifier which is intended to be unique
2. docs.oracle.com/javase/9/core/creating-immutable-lists-sets-and-maps.htm
3. lodash.com/
4. javalibs.com/artifact/com.github.javadev/underscore-lodash
5. www.nuget.org/packages/lodash/
6. github.com/dgilland/pydash
7. rudash-website.now.sh/
8. A lapalissade is an obvious truth — i.e. a truism or tautology — which produces a comical effect
9. lodash.com/
10. github.com/lodash/lodash/wiki/FP-Guide
11. immutable-js.github.io/immutable-js/
12. github.com/GlenKPeterson/Paguro
13. docs.microsoft.com/en-us/archive/msdn-magazine/2017/march/net-framework-immutable-collections
14. github.com/tobgu/pyrsistent
15. github.com/hamstergem/hamster