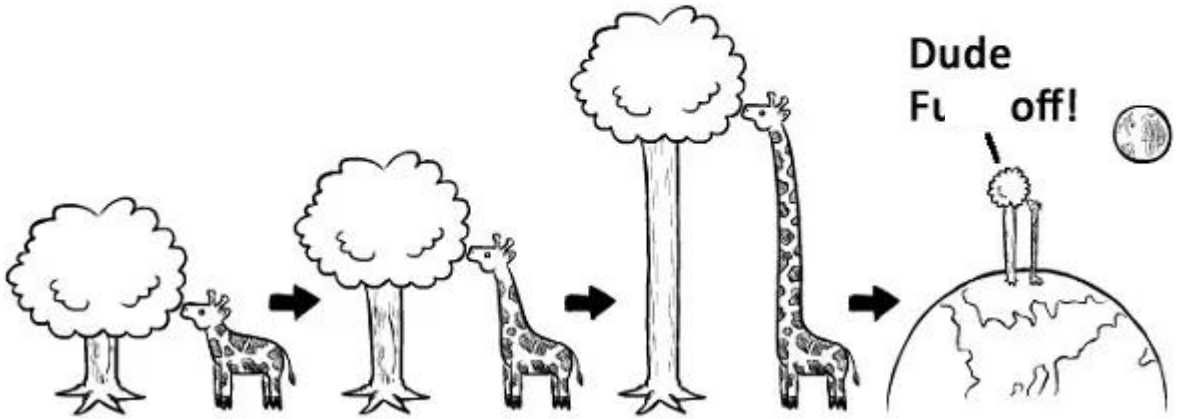


https://www.relativelyinteresting.com/evolution-giraffe-tree-comic/

Emerging Software Design Workshop (Java)

Or… what I've learned so far about Software Design and useful Architecture patterns

Preparation

Import from GitHub: https://github.com/ozihler/esdws-2019

Requirements:

- Java 11+, Angular 8+, NPM 6.4.1+
- Run it:
  - Frontend: Go to package.json and run “start”, or Terminal: cd /frontend, then: npm install, npm run ng serve
  - Backend: run “Application”
- Run backend tests: LibraryTest.java (this is what we will need today)
  - If tests run with gradle in IntelliJ: File > Settings > Build,Execution,Deployment > Gradle > Build and run using IntelliJ IDEA, Run tests using IntelliJ IDEA
- If you have problems running it, here’s the Java 11 only, JUnit 4 version: https://github.com/ozihler/esdws-2019-cmd

Disclaimer

- Some refactorings may be only my personal programming style (which evolves continuously)
- There are almost always multiple, equally correct solutions!
- I cross-referenced what I show here with some LSAs to assure its validity
- The important thing is to discuss different design options together in a fun and respectful way.
- Design is an art and there may be different styles, ideas, and preferences that influence it. This is also why there exist different architectures like Onion, Clean Architecture, or Ports and Adapter, although they basically have the same main idea behind it.

- In the end, discuss it in your team how to do it, write it down, and always stick to it! Nothing worse than multiple different kinds of styles in a project... But at the same time, be open for new and exciting ways to improve your code!

General Idea of Today's Course

- Have Fun!
- Try out ideas you would normally not be able to try out because of time/money restrictions => it does not matter if you finish the exercises or not!
- Maybe you want to do it all alone? in Pairs? Mob-Programming? Your Choice!
- After a while, I will present a possible solution for an exercise. However, you can work at your own pace, and solve whichever exercise you want, in any order… Just check out the respective branch!
  - Use the branch after your exercise to see an example of how it could be done…
- This is the first time I do this course, so don't expect everything to work perfectly…
- Food: You need to go fetch your food yourself...
- Breaks: not specified, but I will present solutions only when most people are here…

What we are going to do today

- Extract domain concepts and logic into domain classes
- Extract and wrap infrastructure logic and presentation logic
- Get to know some background on layered architecture patterns and DDD and how to get there using automated refactoring
- Have an example of how to structure the project as classes and packages towards a »screaming architecture«

Emerging Software Design – Basic Idea

OOP: Bring **state** and **logic** together in meaningful new classes by refactoring

1. Extract some logic as **method**
2. **Wrap state** in **new class**
3. **Move extracted method** into newly created class

6 Step Emerging Design Process

1. **Separate responsibilities within the method**
2. Extract private method
3. Remove direct access to non-required fields of the old class
4. **Wrap target state with domain specific class**
5. **Move method to the target class as public**
6. Simplify parameters to remove unwanted dependencies

Look for nouns in variables & method names: candidates for domain objects!

Online Library! *dev*

Logged in User: AnyUser

Examining the app and code base

Gestörte Kommunikation im amerikanischen Drama Marion Hebach

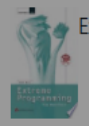
Leidenschaft und Vernunft im Drama des Sturm und Drang Nagla El-Dandoush

Das Science Fiction Jahr 2010 Sascha Mamczak,Wolfga

Refactoring to patterns Joshua Kerievsky

Clean Code Robert C. Martin

Code Complete Steve McConnell



Extreme Programming Kent Beck

OPTIMIZATION METHODS FOR ENGINEERS N.V.S. Raju

ected the following books:

ID	Title	Authors	Type	Days rented
4	Science Fiction in der deutschsprachigen Literatur	Hans-Edwin Friedrich	IMAGE	6
3	Die Monster, die ich rief	Larry Correia	BOTH	4
		nt Beck	IMAGE	5

Your fee

Rental Record for AnyUser
'Science Fiction in der deutschsprachigen Literatur' by 'Hans-Edwin Friedrich' for 6 days: 8.0 $
'Die Monster, die ich rief' by 'Larry Correia' for 4 days: 12.0 $
'Refactoring' by 'Martin Fowler,Kent Beck' for 5 days: 6.5 $
You owe 26.5 $
You earned 4 frequent renter points

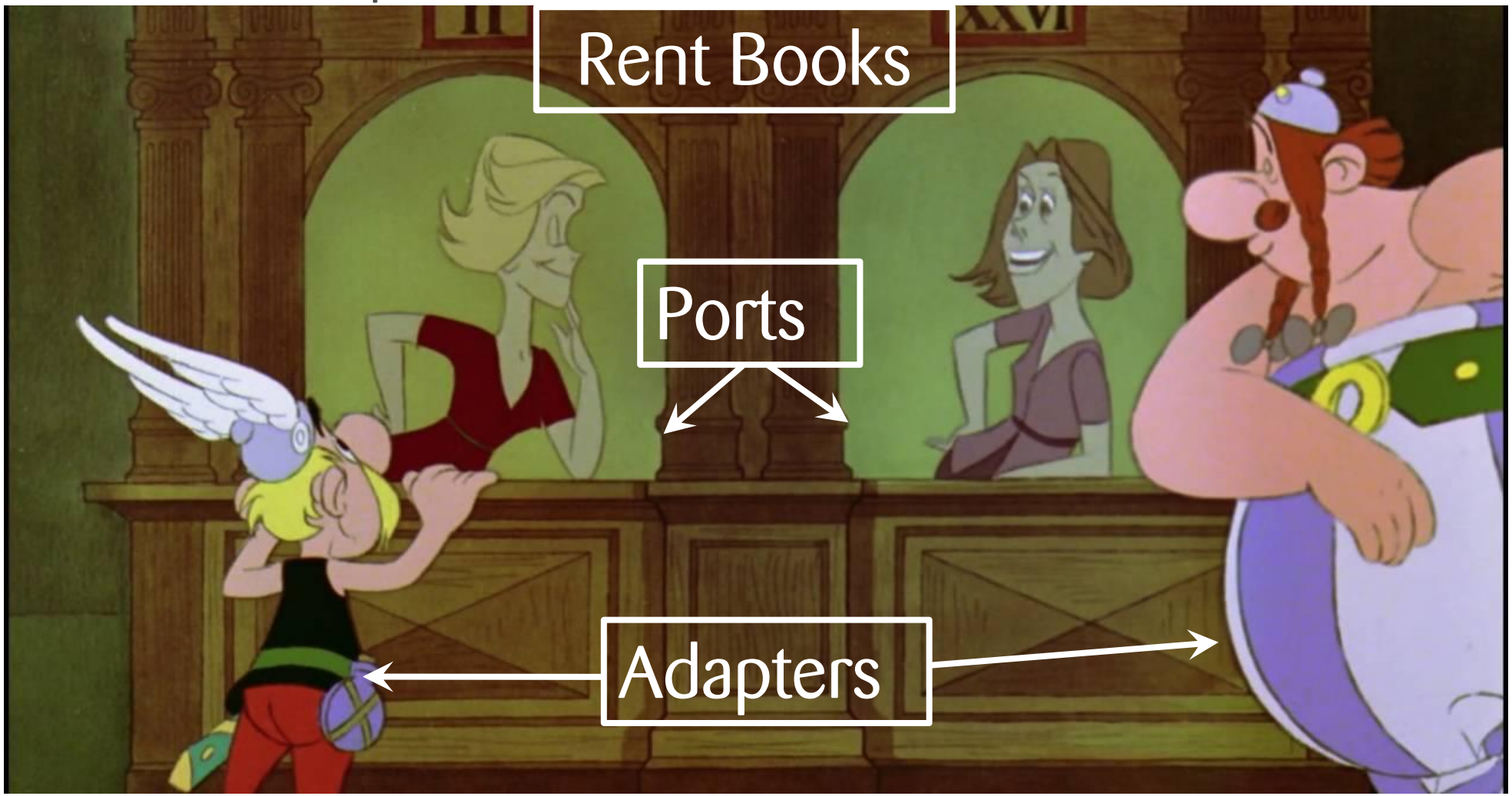
✔ Accept ✖ Reject

Ports and Adapters

https://www.memedroid.com/memes/detail/368783

Architecture: Ports and Adapters

https://i.ytimg.com/vi/wAoUNTRFgvM/maxresdefault.jpg

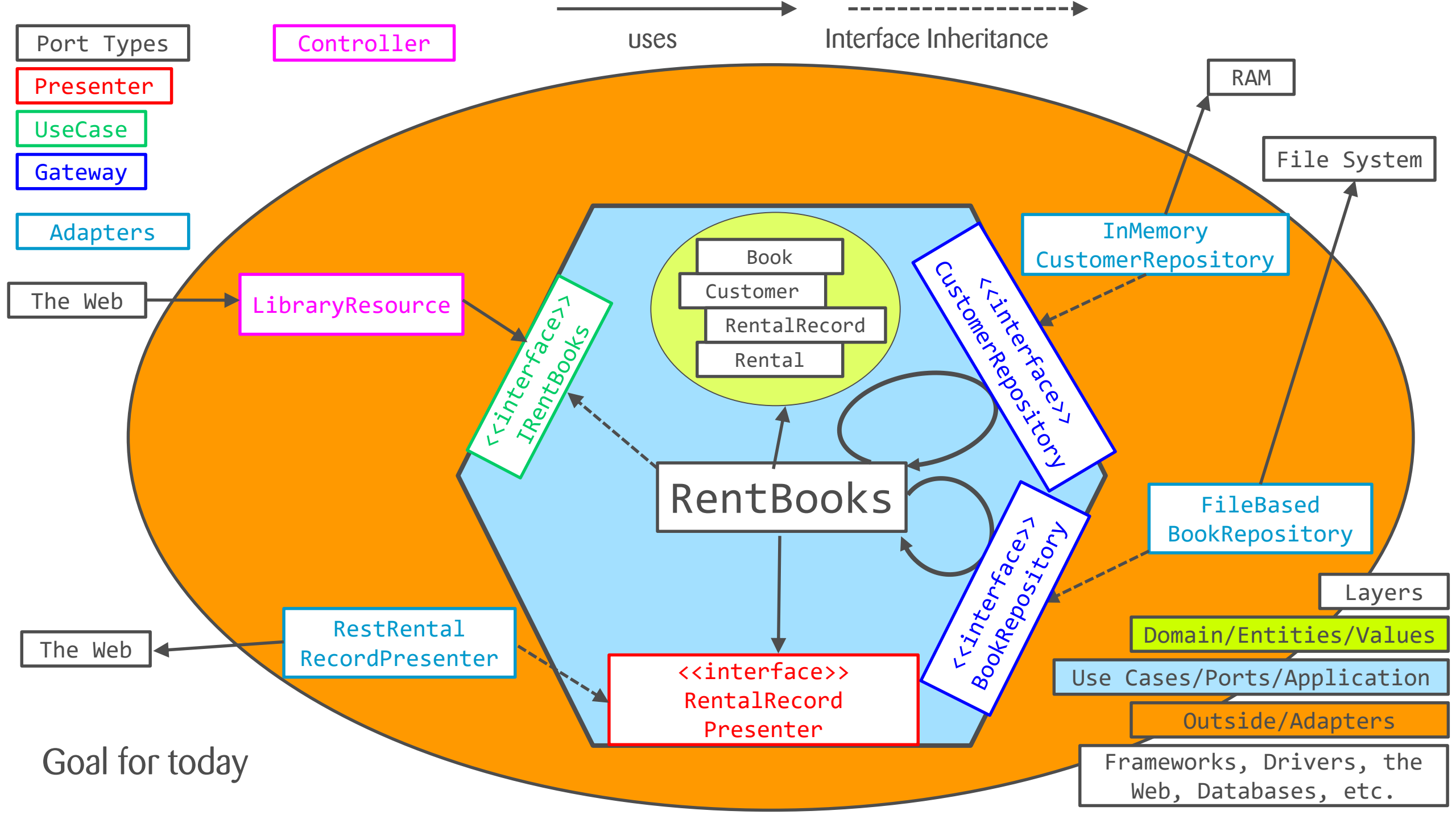


How does the input/output look like? Who decides?

Do we know *how* the output is produced? Who decides?

Do we know *how many different* actions each adapter can perform?

Who *depends* on whom?

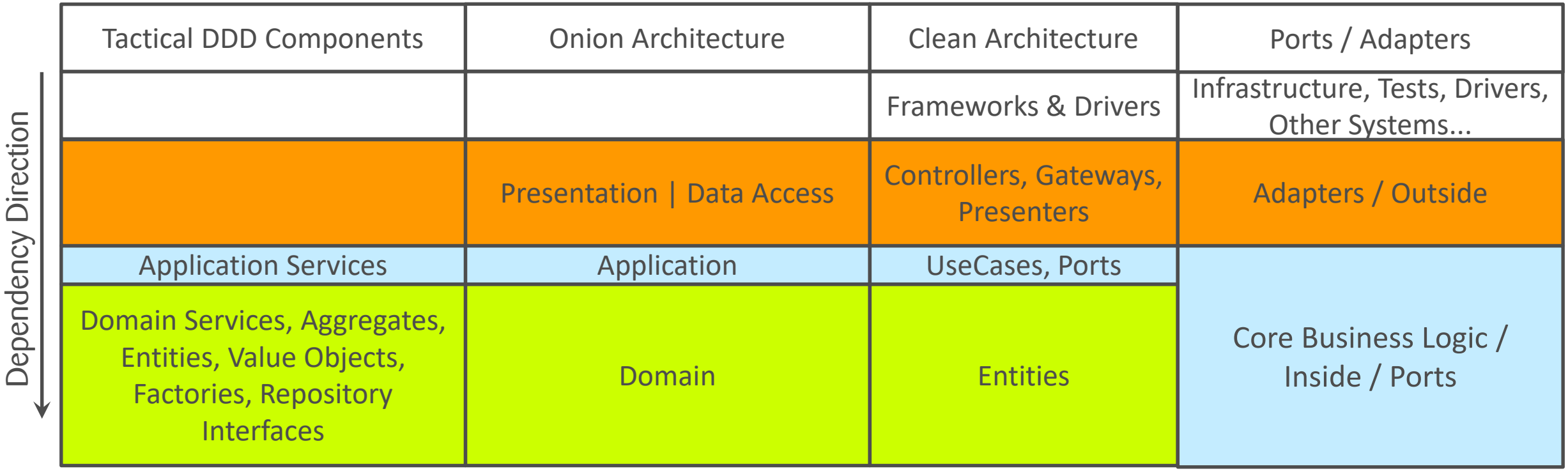

Port Types
Controller
uses
Interface Inheritance
Presenter
UseCase
Gateway
Adapters
RAM
File System
The Web
LibraryResource
<<interface>>
IRentBooks
Book
Customer
RentalRecord
Rental
<<interface>>
CustomerRepository
InMemory
CustomerRepository
RentBooks
<<interface>>
BookRepository
FileBased
BookRepository
The Web
RestRental
RecordPresenter
<<interface>>
RentalRecord
Presenter
Layers
Domain/Entities/Values
Use Cases/Ports/Application
Outside/Adapters
Frameworks, Drivers, the
Web, Databases, etc.
Goal for today

Clean Architecture: **Controllers, Presenters, Gateways** and Ports and Adapters

- Controller: holds a reference to the use case as an interface (Input Port) and formats framework specific input requests to use-case-specific input requests.
- The controller only holds an interface to the use case (Interface Segregation Principle: the Use Case class could have other exposed methods that are irrelevant to the controller, so the controller doesn't need to know of it)
- Gateways: typically interfaces defining CRUD data operations (external systems, databases) (I/O Port). E.g. `Customer findByUsername(String)`. In DDD, these correspond to repositories. Gateway adapters are the actual implementations, e.g. SQL, FS, MongoDB, InMemory...
- Presenters: interface to format Use Case results to output data the view can understand (view model). NOT the same as responses of a use case. Implementations (Adpaters) of these interfaces will even format Java Dates to String representations, so that the view only needs to display it.

Clean Architecture: Use Case (Interactor), DDD: Services, Onion: Application Layer

- A Use Case (Interactor) controls the »dance of entities«. Entities should not leak out of the use case
- Onion Architecture calls this »application layer«, as it's application specific code.
- In DDD, a similar concept to a use case is a Service, but services appear to be broader and tend contain more functionality.
- Use cases are specific to the application. Entities may be used in different use cases.
- Example: RentBooks
- Fetch data through repositories, create entities, orchestrate entity interactions
- Procedural in nature
- No dependencies to UI, DB, or Frameworks
- Input/Output: simple request/response data structures, no dependencies to entities or frameworks! (mostly primitives or *value primitives* (e.g. wrapper classes for »BookId«))

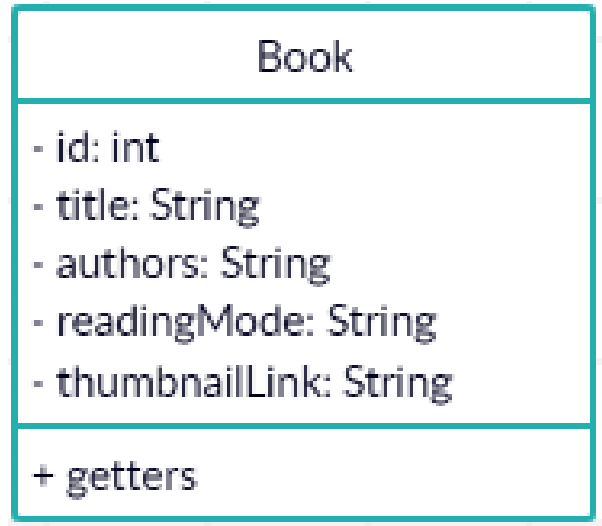

Tactical DDD Components
Onion Architecture
Clean Architecture
Ports / Adapters
Frameworks & Drivers
Infrastructure, Tests, Drivers, Other Systems...
Presentation | Data Access
Controllers, Gateways, Presenters
Adapters / Outside
Application Services
Application
UseCases, Ports
Domain Services, Aggregates, Entities, Value Objects, Factories, Repository Interfaces
Domain
Entities
Core Business Logic / Inside / Ports
Dependency Direction

E1: Introduce Book Entity (master branch)

- Introduce Book.java in `Library#calculateFee` with the following constructor:

```
Book(int id, String title, String authors, String readingMode, String thumbnailLink)
```

- Use Book wherever the primitive String[] book array was used (implement getters)
- Try to use **Parallel Change** to replace
  - String[] book --> Book book
  - List<String[]> books --> List<Book> books
- Do not move any more logic into Book yet
- Add Book to new package »**library.domain.entities**«



If you finish early, try introducing value objects for all primitives where possible

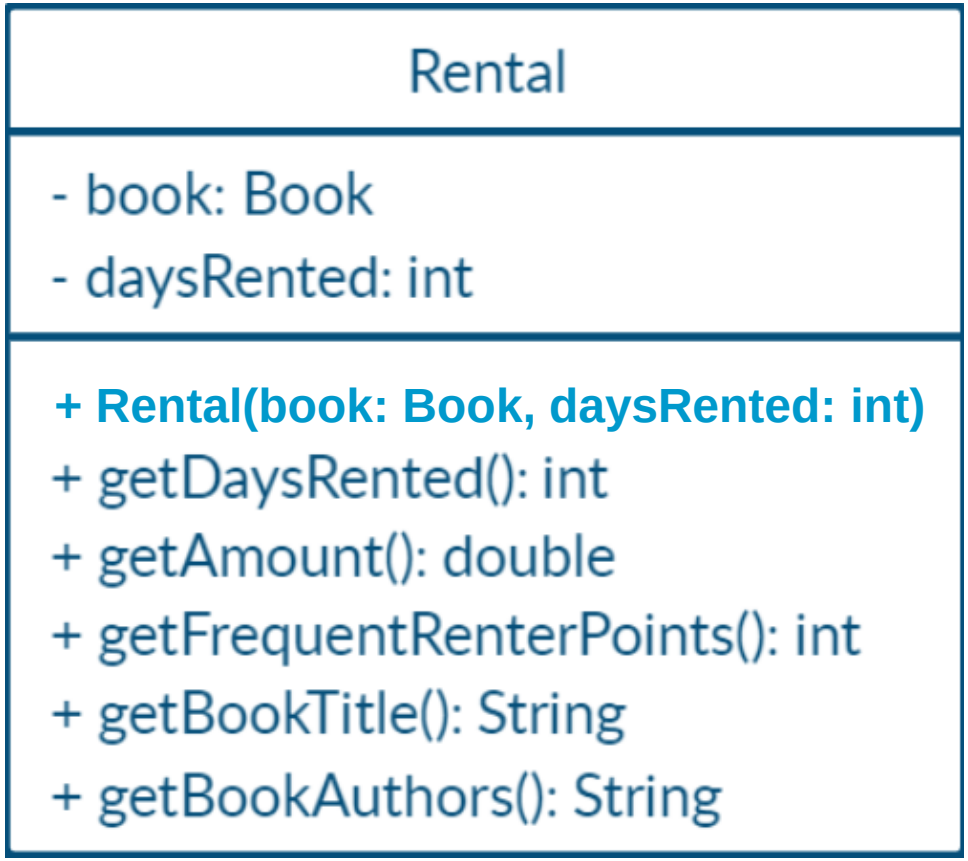
Parallel Change

1. Create **new structure** in parallel to existing one
2. Step-by-step **add write accesses** to new structure next to **every write access** to old structure (**state** of new structure becomes the same as the one of the old structure)
3. Step-by-step **switch read accesses** from old to new structure
4. Step-by-step **remove write accesses** to old structure
5. Remove old structure

Run tests whenever it makes sense!

E2: Introduce `Rental` (Branch E1-2)

- Replace String[] rental with Rental rental
- `Rental` should get the following constructor:
  - `public Rental(Book book, int daysRented)`
- Extract and move the following logic into Rental
  - `public int getDaysRented()`
  - `public double getAmount()`
  - `public int getFrequentRenterPoints()`
  - `public String getBookTitle()`
  - `Public String getBookAuthors()`



- Use **Parallel Change** to replace the old Rental Array with the new Rental Object
- Move the Rental class to package »**library.domain.values**«

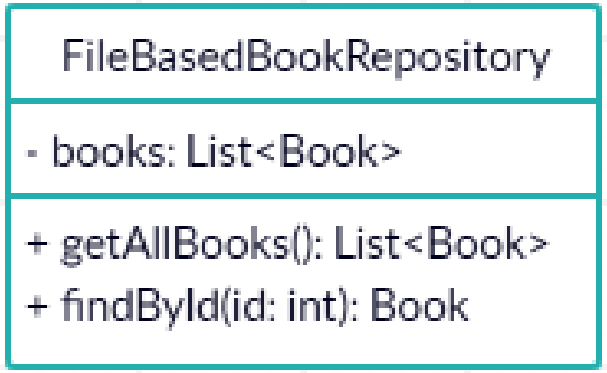
If you finish early, try introducing value objects for all primitives where possible

Entities & Value Objects in a nutshell

- An Entity is an object with an identity. The Book »Clean Code« is not the same as book »Clean Architecture«
- A Value Object has no identity. 1 Dollar = 1 Dollar. Although the object may be different, the identity of the two objects is still *equal.* Any ID is a value object. Values are typically immutable, see String
- It often depends on the context if an object is treated as entity or as value object
- In an all encompasing OOP system, there are no primitive types anymore, so all Strings, integers, etc. are at least value objects or entities.
- So a book constructor looks like this:
  - `Book(Key key, Title title, Authors authors, ReadingMode readingMode, Link link)`
- See e.g. DDD for more details. All Exercises have a sister branch with «-with-value-objects« where there are now primitive types anymore

E3: Introduce FileBasedBookRepository (Branch E1-3)

- Extract **file reading and parsing** logic for Book into a new `BookRepository` class
- `FileBasedBookRepository` should get the following two methods:
  - `public Collection<Movie> getAllBooks()`
  - `public Book findById(int id)`



- `FileBasedBookRepository` should **load the file in its constructor**
- Optional:
  - Use `FileBasedBookRepository` also for `Library#getBooks()` (without altering getBooks() return type: convert books to String[] again!)
  - Initialise the repo in Library's constructor
  - Use **Parallel Change** if you have to replace an old with a new structure and run tests after each change
- Move FileBasedBookRepository to package »**library.adapters.file_persistence**«

E4: Split the while loop (Branch E1-4)

- Into 4 loops
  - first loop creates rentals and stores it in List<Rental> rentals
  - the next 3 loops iterate over this rentals list and perform their respective action
- Extract the following methods for the 3 additional loops:
  - `double getTotalAmount(List<Rental> rentals)`
  - `int getFrequentRenterPoints(List<Rental> rentals)`
  - `String format(List<Rental> rentals)`
- Don't extract any methods for creating a Rental yet
- Hint: in IntelliJ, enter »iter + enter« to create a loop for an existing collection in the current scope
- Try to move variables close to where they are used, or inline them if they are only used once (call the method directly)

If you finish early, try introducing value objects for all primitives where possible

Technique: Loop Splitting

1. If the body of a loop contains **several responsibilities** that should be separated, the loop needs to be **duplicated** so that **each loop has only a single responsibility**

2. **Create a List** in the first loop and iterate over it in all the following loops

3. This approach of duplicating the control structure is also feasible for other control structures (e.g. if, switch) with several responsibilities in the body

E5: Introduce RentBookRequest (Branch E1-5)

- Introduce a Data Transfer Object (DTO) for one »rent book« request that wraps bookId and daysRented
- Replace all accesses to the variables with accesses to rentBookRequest.getBookId() and rentBookRequest.getDaysRented()
- Add all »rent book« requests to a List<RentBookRequest> rentBookRequests and iterate over it to create rentals (**loop splitting**)
- Additionally, extract two methods for the two loops (but don't move them yet into any class):
  - getRentBookRequests(rentBooksRequestData: List<String>)
  - getRentals(rentBookRequests: List<RentBookRequest>)
- move RentBookRequest to package »**library.application.use_cases.rent_books.ports**«

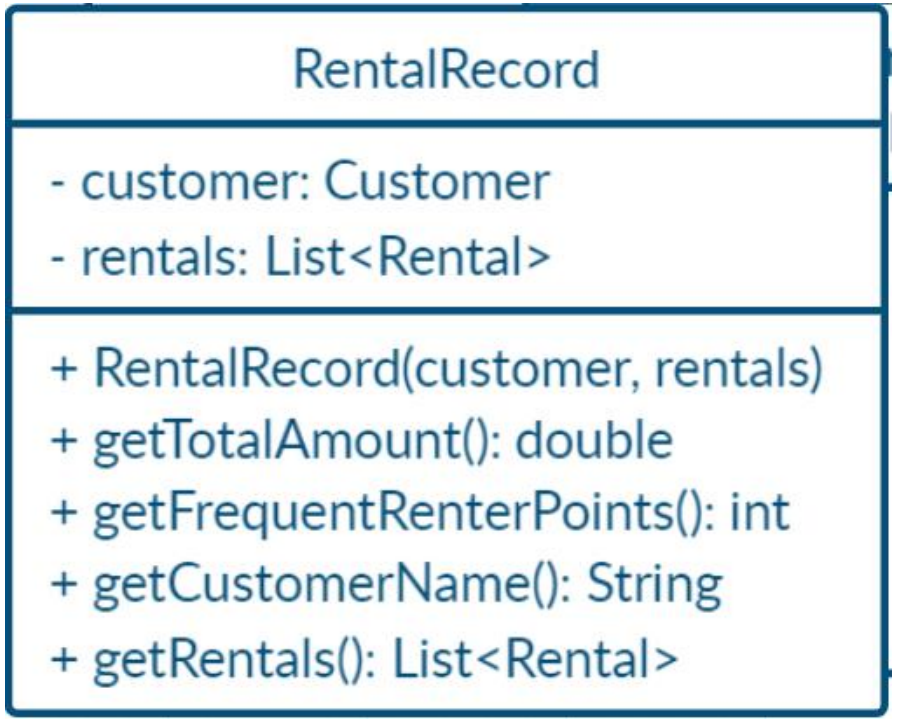
If you finish early, try introducing value objects for all primitives where possible

E6: Introduce `RentalRecord` (Branch E1-6)

- With member variables `List<Rental>` rentals and `Customer customer`
- And methods
  - `public List<Rental> getRentals()`
  - `public String getCustomerName()`
  - `public double getTotalAmount()`
  - `public int getFrequentRenterPoints()`
- Replace all access to rental/customer variables with calls to these methods

Add RentalRecord to package »**domain.values**«

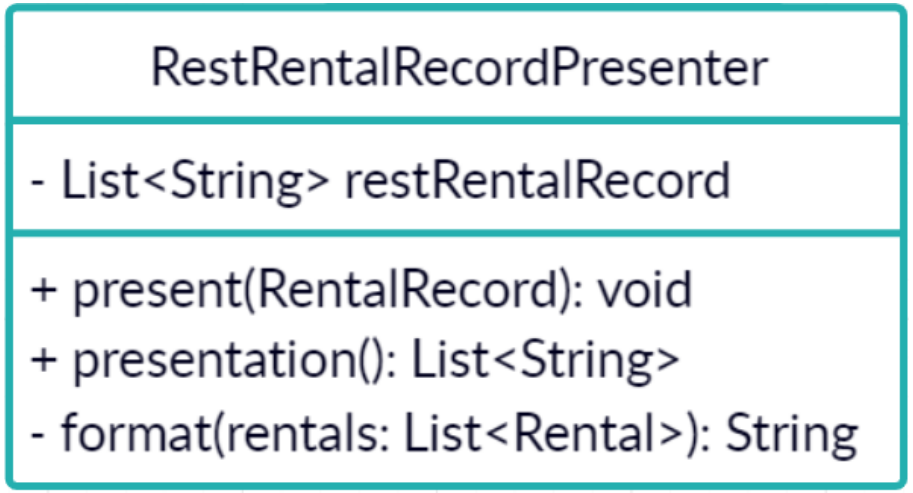
Rental creation should stay in LibraryResource, only move the generated rentals List to RentalRecord



If you finish early, try introducing value objects for all primitives where possible

E7: Presentation Logic: `RestRentalRecordPresenter` (E1-7)

- extract a method `present(RentalRecord rentalRecord)` and move it to RestRentalRecordPresenter, which formats the rental record to a List.of(String).
- Store this in a field called **List<String> restRentalRecord**
- Add method **List<String> presentation()** that returns restRentalRecord and use it as a return value in Library#calculateFee
- also move **`format(rentals)`** to RestRentalRecordPresenter
- move this class to package »**library.adapters.rest**«



If you finish early, try introducing value objects for all primitives where possible

E8: Extract Use Case Interactor: `RentBooks (E1-8)`

- `RentBooks.executeWith(String customerName, List<String> rentalRequest, RestRentalRecordsPresenter presenter): void`
  - This method should fetch the customer, create the rentals, build a rental record, and pass it to the restRentalRecordsPresenter
- Move it to package «**library.application.use_cases.rent_books**«

If you finish early, try introducing value objects for all primitives where possible

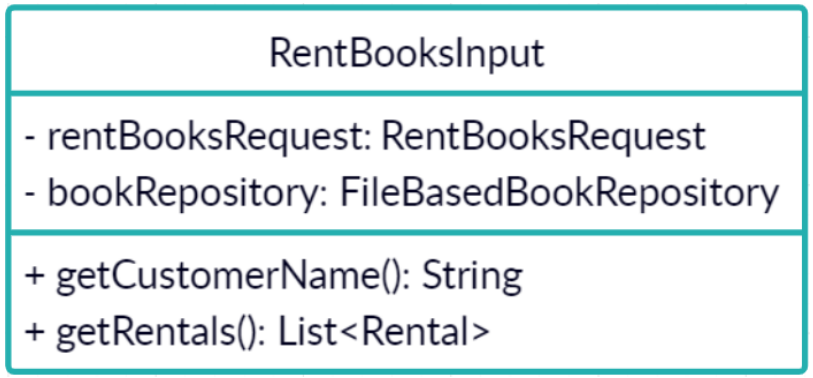
E9: Extract RentBooksRequest (E1-9)

- RentBook**s**Request(customerName, List<RentBookRequest>)
- This class only has getters and wraps the request parameters that belong together
- use it as input for RentBooks.with(RentBooksRequest, RestRentalRecordPresenter)
- Move it to package »**library.application.use_cases.rent_books.ports**«
- Try to use automatic refactorings »**introduce Parameter Object**« or »**extract parameter**« on method RentBooks.executeWith. Which works better?

If you finish early, try introducing value objects for all primitives where possible

E10: Extract RentBooksInput (E1-10)

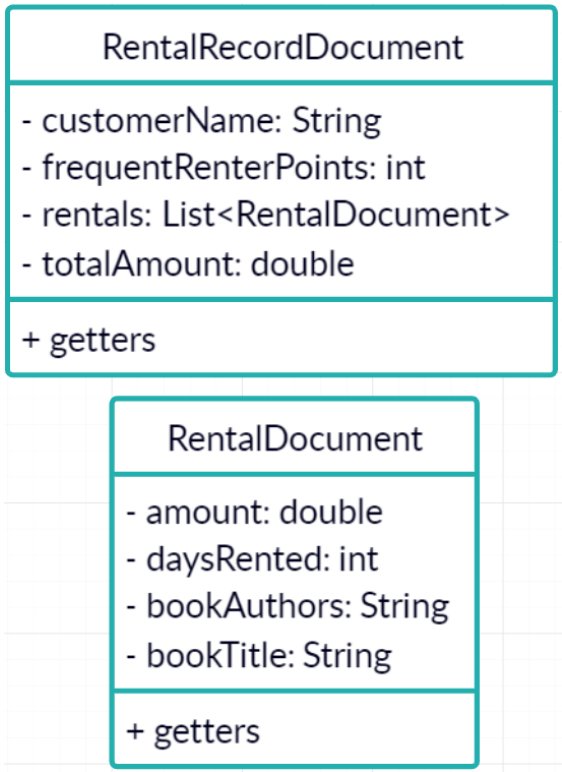
- input to RentBooks.executeWith(RentBooksInput, RestRentalRecordPresenter)
- RentBooksInput wraps RentBooksRequest and creates and provides rental entities to the use case (like a factory) → thus, RentBooksInput needs a bookRepository for creating rentals
- Try to use »Parameter Object« or »extract parameter (ctrl-alt-p)«
- move RentBooks.getRentals(List) to RentBooksInput (and remove the list parameter)
- extract method getCustomerName() and move it to RentBooksInput so that there is no get-chain anymore
  - input.getCustomerName() instead of input.getRentBooksRequest().getCustomerName()
- Finally, remove the unused bookRepository from RentBooks Use Case
- Move it to package »**library.application.use_cases.rent_books.ports**«



If you finish early, try introducing value objects for all primitives where possible

E11: Introduce RentalRecordDocument and RentalDocument (E1-11)

- That holds double totalAmount, int frequentRenterPoints, String customerName, and rentals
- These classes represent the presentable information that the use case produces and do not contain entities like Rentals anymore
- Rentals becomes a List<RentalDocument>, where each RentalDocument has double amount, String bookAuthors, String bookTitle, and int daysRented
- Use RentalDocument as input for **RestRentalRecordPresenter.present()** instead of the Rental Entity: present(RentalRecordDocument)
- Both RentalRecordDocument and RentalDocument are simple data structures with only getters and no logic
- **Creation of these documents is inside RentalRecord and Rental, respectively:**
  - **RentalRecord.asDocument() / Rental.asDocument() create a RentalRecordDocument / RentalDocument, respectively, create new RentalRecordDocuments/RentalDocuments**
- Move RentalRecordDocument/RentalDocument to »**library.domain.values**«



E12: Extract Ports (Interfaces) for Use Case, Gateways, and Presenters (E1-12)

- **RentalRecordsPresenter.present(RentalRecordDocument)** from **RestRentalRecordsPresenter**
  - move RentalRecordsPresenter output port to package »**library.application.outbound_ports.presentation**«
- **IRentBooks.executeWith(RentBooksInput, RentalRecordPresenter)** from **RentBooks**
  - move IRentBooks input port to package »**library.use_cases.ports**«
- **CustomerRepository.findByUsername(username)** from **InMemoryCustomerRepository**
  - move InMemoryCustomerRepository adapter to package »**library.adapters.in_memory_persistance**«
  - move CustomerRepository output port to package »**library.application.outbound_ports_persistance**«
- **BookRepository.findById(id)** and **BookRepository.getAllBooks()** from **FileBasedBookRepository**
  - move BookRepository output port to package »**library.application.outbound_ports.persistance**
- Use »**extract interface**« automatic refactoring if possible
- Clean Up: Move Customer to **library.domain.entities**, LibraryResource to **library.adapters.rest**

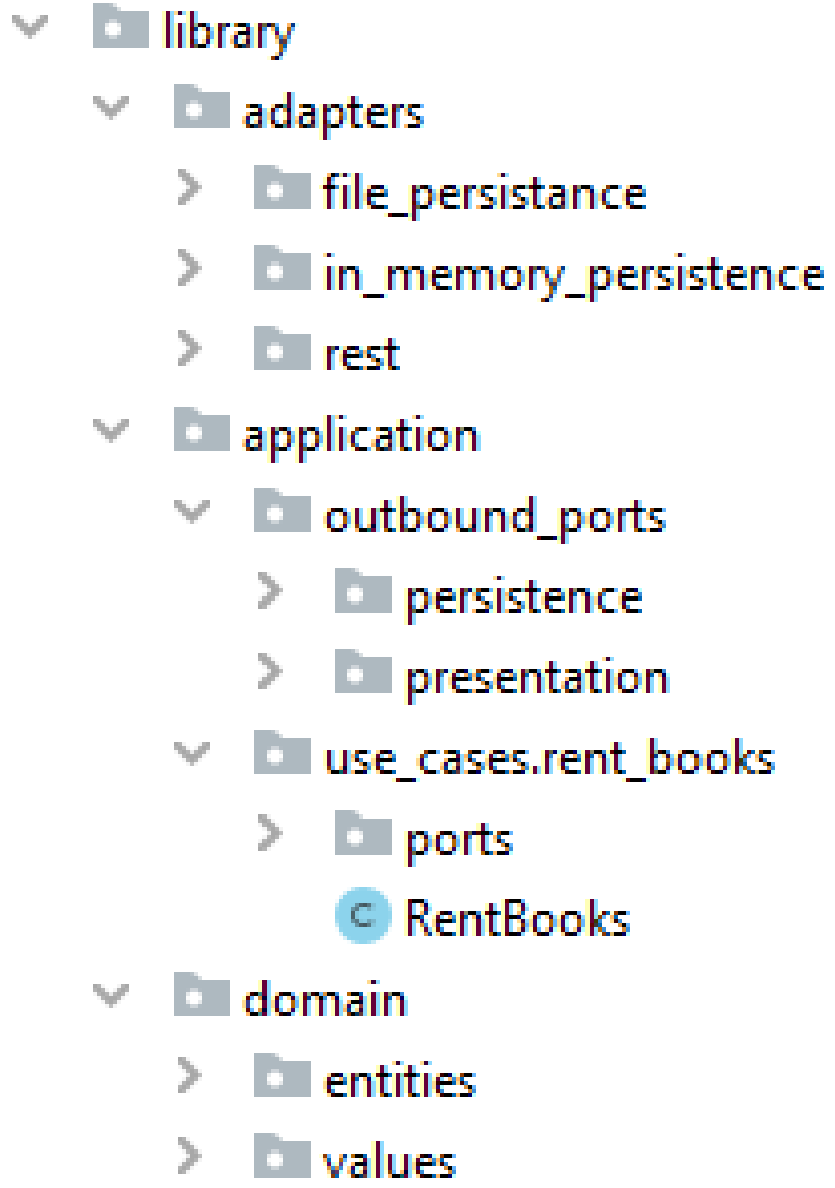
Some Discussion

- There are other small things that can still be improved →Try it out if you have time. Also see E1-13 introduce strategy, some pages further on
- Add a controller-adapter that calls the use case and annotate it with e.g. @Service to make it injectable by Spring (using @Autowired) (see E1-14 for solution)
- Add @Repository to InMemoryCustomerRepository and FileBasedBookRepository to make it injectable by Spring (using @Autowired) (see E1-14 for solution)
- BookRepository port interface could be split into two interfaces (Interface Segregation Principle)
- The interfaces abstract what goes in and out of the use case
  - Dependency Inversion: Outside depends on the use case, not the other way round
  - Interface Segregation Principle: The use case implementation could expose many more methods that the corresponding adapter does not need to care about
  - TESTING is simplified!
    - The first adapter to any use case interface should be a test case
    - Outbound interfaces are easily replaced by mocks or test objects
    - The core business domain can be tested extensively without the need of any special framework!
- The use case is independent of any framework! We could simply add another layer for use case specific operations (e.g. @Service, @Repository, etc.) --> it's a detail!

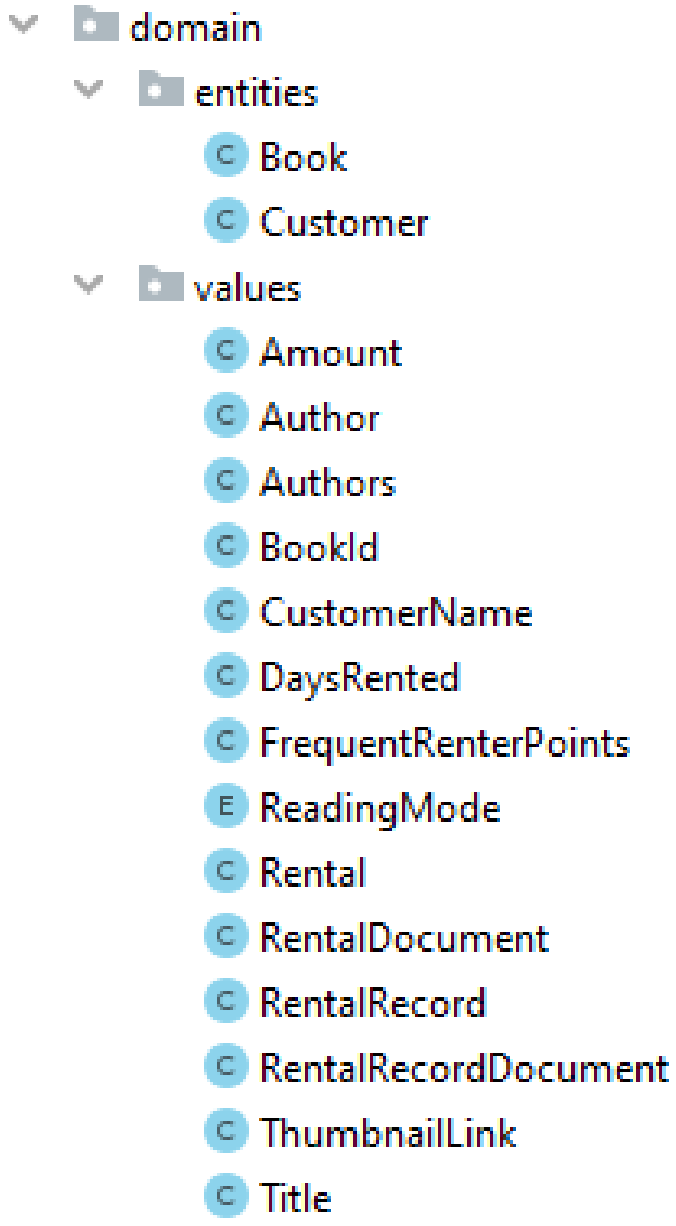
Testing the Use Case with a Unit Test Example

```java
public class RentBooksTest {
    @Test
    public void testRentBooksExecuteWith() {
        new RentBooks(name -> new Customer(name))
                .executeWith(testInput(), rentalRecord -> assertResults(rentalRecord));
    }

    private RentBooksInput testInput() {
        RentBookRequest rentBookRequest = new RentBookRequest( bookId: 1,  daysRented: 5);
        RentBooksRequest rentBooksRequest = new RentBooksRequest( customerName: "userX", List.of(rentBookRequest));
        return new RentBooksInput(bookId -> testBook(bookId), rentBooksRequest);
    }

    private Book testBook(int bookId) {
        return new Book(bookId,  title: "Hello",  authors: "World",  readingMode: "BOTH",  thumbnailLink: "link");
    }

    private void assertResults(RentalRecordDocument rentalRecord) {
        assertRentalRecord(rentalRecord);
        assertRentals(rentalRecord.getRentals());
    }

    private void assertRentalRecord(RentalRecordDocument rentalRecord) {
        assertEquals(testInput().getCustomerName(), rentalRecord.getCustomerName());
        assertEquals(testInput().getRentals().get(0).getFrequentRenterPoints(), rentalRecord.getFrequentRenterPoints(),  delta: 0.001);
        assertEquals(testInput().getRentals().get(0).getAmount(), rentalRecord.getTotalAmount(),  delta: 0.001);
    }

    private void assertRentals(List<RentalDocument> rentals) {
        assertEquals( expected: 1, rentals.size());
        assertEquals(testBook( bookId: 1).getTitle(), rentals.get(0).getBookTitle());
        assertEquals(testBook( bookId: 1).getAuthors(), rentals.get(0).getBookAuthors());
        assertEquals(testInput().getRentals().get(0).getDaysRented(), rentals.get(0).getDaysRented());
        assertEquals(testInput().getRentals().get(0).getAmount(), rentals.get(0).getAmount(),  delta: 0.001);
    }
}
```

Possible Package Structure - »Screaming« Architecture



With Value Objects



BONUS E1-13 - Introduce Polymorphism for Book Reading Mode

- Total Amount is calculated based on the Reading Mode
- A switch statement is not OOP!
- Replace it by introducing a strategy for each type that calculates the total amount based on its type

Technique: Introduce Polymorphism

1. Encapsulate constructor with static factory method
2. Create empty subclasses with the same constructor signature like the parent class, simply passing all parameters via super(…)
3. Instantiate subclasses instead of parent class inside factory method
   - Copy-paste the switch case statement from one of the methods into the factory method and return the corresponding subclass for each case
4. Declare parent class as abstract
5. Use **Duplicate & Reduce** to …
   1. … move logic from the parent class into the subclasses
   2. … and simplify that logic to only perform the subclass's responsibilities
   - CPA (Copy-Paste-Adapt) or "Push members down" Refactoring + Adapt

Duplicate & Reduce

Technique to split methods containing multiple responsibilities needed by different callers

1. Duplicate the method by:
   - Copy-Paste (plus rename) or
   - “Push members down” to all subclasses or
   - “Inline method implementation” on caller side

2. Reduce the original and copy to what is needed for the specific caller or in the given scope

What techniques we have learned today

- Replace temporary variable with query/extract method
- Introduce parameter object/extract parameter
- Parallel Change
- Narrow Change
- Loop Splitting
- Duplicate & Reduce
- Separating responsibilities and concerns
- Emerging inheritance via Introduce Polymorphism
- Emerging reasonable compositions
- Code generations, refactorings, IDE suggestions
- The basics Ports/Adapters, Clean Architecture, Onion and some DDD

Last Page…

- I hope you had fun and gained some new or different insights today!
- You can contact me e.g. via oliver.zihler@zuehlke.com
- See also Alistair Cockburn in the Hexagon (1-3, that's how I learned about Ports/Adapters): https://www.youtube.com/watch?v=th4AgBcrEHA
- Books: Clean Code, Clean Architecture, Clean Coder, DDD, Implementing DDD, DDD Quickly, DDD Distilled, Principles, Patterns, and Practices in DDD, Refactoring (old and new), Refactoring to Patterns
- Courses on Refactoring and Code Smells: https://www.industriallogic.com/
- Use the Spring Boot/Angular Project as a template for your own project if you like!