Emerging Software Design Workshop (Java)

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

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 LSA's to assure it's correctness
- 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!

Emerging Software Design – Basic Idea

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

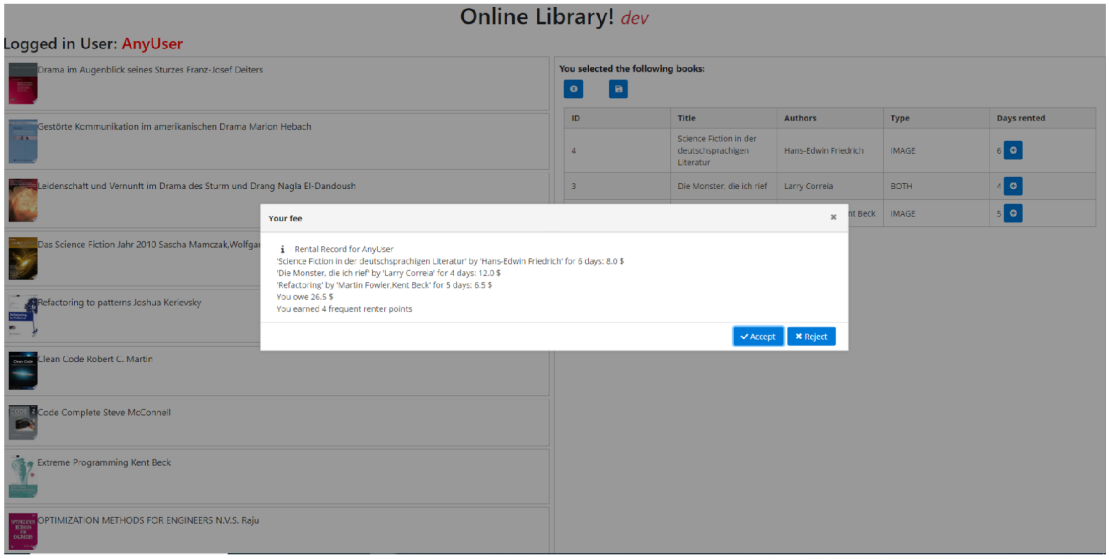
1. Extract some logic as **method**
2. **Wrap state** in **new class**
3. **Move extracted method** into newly created class

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!

Examining the app and code base



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 get there using automated refactoring
- Have an example of how to structure the project as classes and packages towards a »screaming architecture«

E1: Introduce Book Entity (15')

- Introduce Book.java in `Library#calculateFee` with the following constructor:
- `Book(int key, String title, String authors, String readingMode, String link)`
- 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**«

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

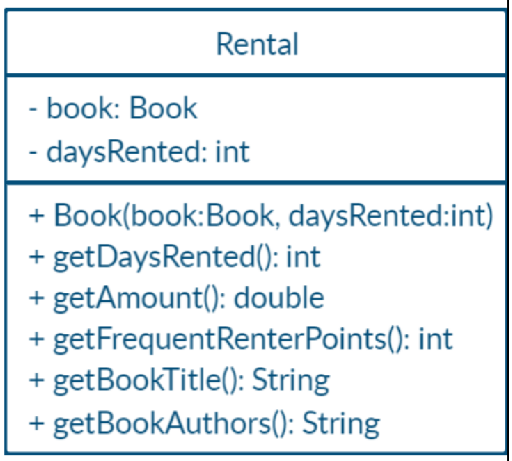
Run tests whenever it makes sense!

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

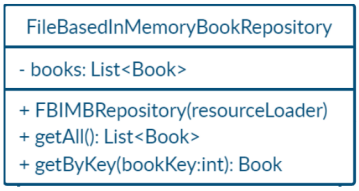
E2: Introduce `Rental` Value Object (20')

- 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**
- Add Rental to package »**domain.values**«



E3: Introduce FileBasedBookRepository (20')

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



- `FileBasedBookRepository` should **load the file in its constructor**
- Use `FileBasedBookRepository` also for `Library#getBooks()` (without altering getBooks() return type: convert books to String[] again!)
- Initialise the repo in Library's constructor
- Move FileBasedBookRepository to package »**library.adapters.file_persistance**«

E4: Split the while loop (15')

- Into 4 loops
- Extract the following methods for the 3 additional loops:
  - `double getTotalAmount(List<Rental> rentals)`
  - `int getFrequentRenterPoints(List<Rental> rentals)`
  - `String toDocument(List<Rental> rentals)`
- Don't extract any methods for creating a Rental yet

Technique: Loop Splitting

- 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**
- **Create a List** in the first loop and iterate over it in all the following loops
- 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: RentBookRequest (15')

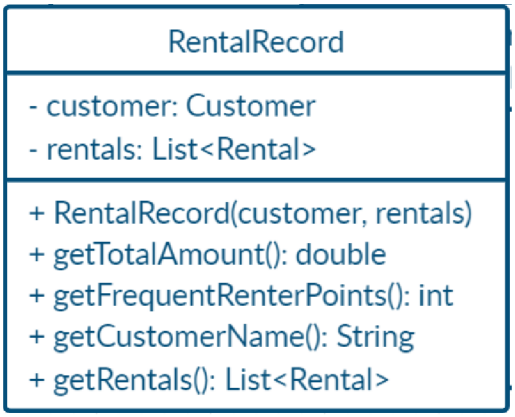
- Introduce a Data Transfer Object (DTO) that wraps bookId and daysRented
- Replace all accesses to the variables with accesses to rentBookRequest.getBookId() and getDaysRented()
- Add all to a List<RentalBookRequest> rentalBookRequests and iterate over it to create rentals (loop splitting)
- move it to package »**library.application.use_cases.rent_books.ports**«

E6: Extract domain concept – create `RentalRecord` (10')

- With member variables `List<Rental>` rentals and `Customer customer`
- And methods
  - `public double getTotalAmount()`
  - `public int getFrequentRenterPoints()`
  - `public String getCustomerName()`
  - `public List<Rental> getRentals()`

Extract the loops into own methods and move them to RentalRecord

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



E7: Extract Presentation Logic – create `RestRentalRecordPresenter`

- add method `present(RentalRecord rentalRecord)` to StringRentalRecordPresenter, which formats the rental record to a List.of(String). Store this in a field called List<String> rentalsResponse
- Add getter for the rentalsResponse and use it in Library#calculateFee
- also move `format(rentals)` to RestRentalRecordPresenter

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, DTOs, no logic!)

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 LSA's to assure it's correctness
- 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!

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 LSA's to assure it's correctness
- 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!

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 LSA's to assure it's correctness
- 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!

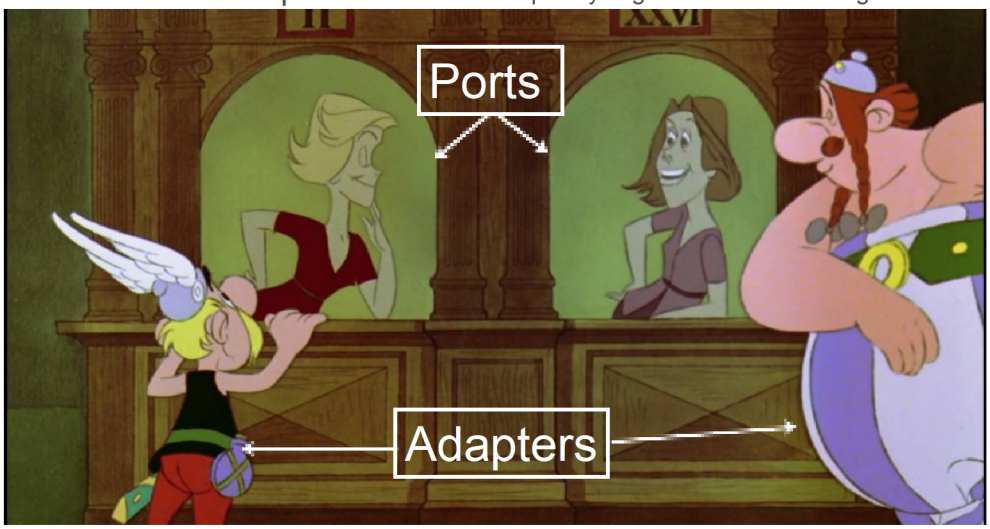
6 Step Emerging Design Process

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!

Architecture: Ports and Adapters

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

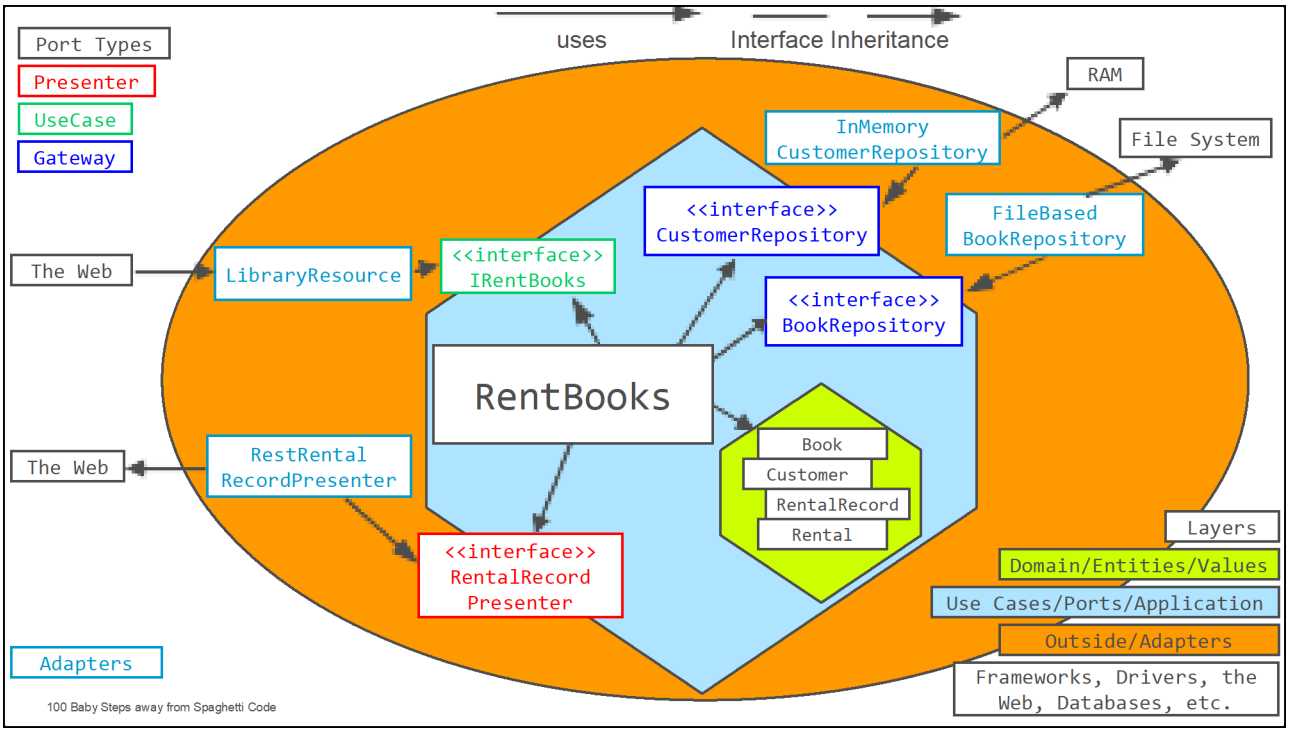


How does the input/output look like? Who decides?

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

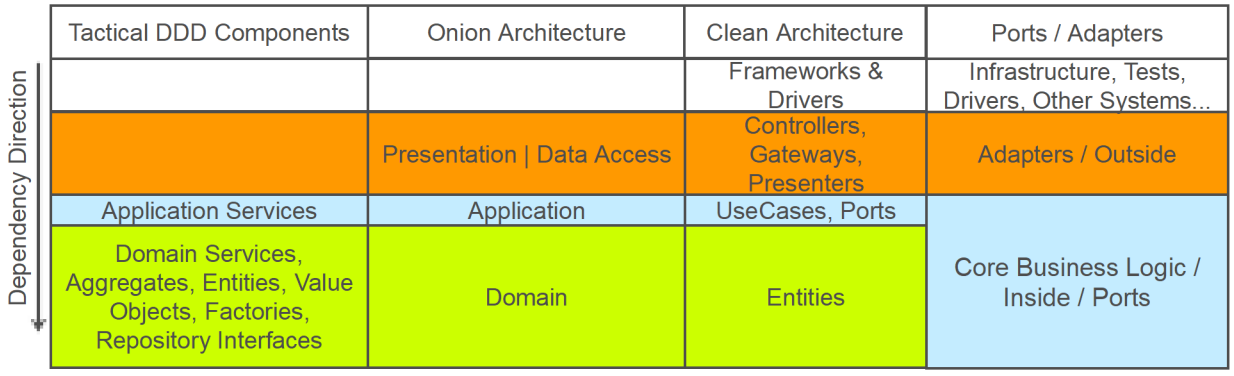
Do we know *how many different* actions each adapter can perform?

Who *depends* on whom?


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

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 LSA's to assure it's correctness
- 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!


Dependency Direction
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

Disclaimer

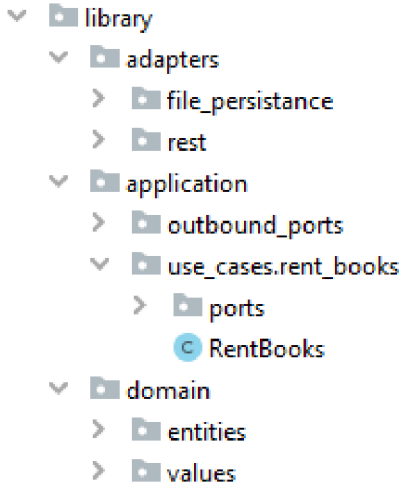
- Some refactorings may be only my personal programming style (which evolves continuously)
- There are almost always multiple, equally correct solutions!
- I cross-referenced what I show here with some LSA's to assure it's correctness
- The important thing is to discuss different design options together 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!

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

Possible Package Structure



E5: RentBookRequest (15')

- Introduce a Data Transfer Object (DTO) that wraps bookId and daysRented
- Replace all accesses to the variables with accesses to rentBookRequest.getBookId() and getDaysRented()
- Add all to a List<RentalBookRequest> rentalBookRequests and iterate over it to create rentals (loop splitting)
- move it to package »**library.application.use_cases.rent_books.ports**«

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 LSA's to assure it's correctness
- 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!

Technique: 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 copy to what is needed for the specific caller or in the given scope

- 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 of common architecture patterns like Onion, Ports/Adapters, Clean Architecture
- Concepts like Entity, Value Object, Aggregates, Factory, Repository, Service, Use Case, etc.

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!