Guided Project, Part 2.

The project will be ideally implemented using the IntelliJ or Eclipse environment: Java with a Maven project and JUnit for unit tests. You will use Git and Gitlab (https://gitlab.esiea.fr with the Azure AD login) and work in pairs.
You will work in pairs or in trios, but each member of the team is expected to create minimum 3 use cases at the end of the project, on different branches of your project; branches will be merged at the end and so you will deliver minimum 6 working or 9 working if in trios, tested use cases in one program.
Beware that you are not allowed to write code until STEP 12. With STEP 1 to STEP 11 you are DESIGNING your application. Starting with STEP 12, you are IMPLEMENTING it. During the implementation, you can update your design.
All the diagrams (use case, class, package, sequence, component) and the use case description tables will be committed in git as well, in a folder named docs.

With the next steps you will put in place a proper design for the use cases that you have described in the previous part.

STEP 2. Let's now start to put in place the design that allows you to implement the use case. Start by identifying the entities that your use case will have to handle. These will constitute your Model. Draw them as simple boxes.

STEP 3. Now we go up to the Use case layer (your Interactor). Follow these rules when designing the Interactor classes belonging to this package:

• they encapsulate the implementation of your use cases (one class per use case)

• they handle the flow of data from the entities and achieve the goals of the use case

• they have no dependencies and are totally isolated from use interface, database or frameworks • If the use case changes, they will change The Interactor will implement an interface (say RequestHandler) allowing to hide the implementation details of the Interactor from the Controller. This leads to a loosely coupled interaction between the Controller and the Interactor. That interface can contain a single method which, when implemented in the Interactor, will do all the work described in your use case. For instance: Response handle (Request message) The interface imposes a Request object as parameter and a Response class as return value. For your Interactor, these will be specific to your use case. The Request and the Response can be abstractions; they are named data transfer objects, together with the classes implementing them. The concrete classes are usually simple classes similar to entities. The classes implementing the Request for your use case contain any data necessary for your use case, coming as user input. The classes implementing the Response contain any information needed as feedback to the user, as a result of the execution of the use case.

STEP 4. Your interactor will use some “services” to implement the use case. The services used by the interactor are referred to as gateways and are injected into the interactor by dependency injection. Interactors only know what behavior these gateways offer by their interface definition. They have no idea how they do their work because those details are encapsulated in another layer which the Interactor knows nothing about: this is the database layer. Now design your gateways, which are basically the interfaces defining the services needed by the interactor to implement the use case. Examples: repositories to persist your entities, authentication services, logging services… STEP 5. Let’s go into the most external layer of the Clean Architecture diagram now and design the classes offering the services for your Interactor (repositories, authentication, logging etc). These classes will implement the interfaces defined in the gateways package.

STEP 6. It’s time to review your packages. At this point, make sure that you have: repositories, entities, usecases, datatransferobjects. You can draw a first package diagram.

STEP 7. Now that you have the design of the Interactor and all the services it needs to implement a use case, let’s deal with the user interaction. We will introduce a controller for our use case only, who has several responsibilities:

• wait and retrieve the user input from the console

• instantiate the Interactor and the necessary data transfer objects (so the Request)

• use the Interactor to handle the use case based on the request coming from the user

• instantiate a presenter to show the result of the use case (the Response) to the user

Your controller will inject in the Interactor the services it needs to perform the job.

STEP 8. We still have to design a presenter, whose responsibility is to prepare the information as we need it to be shown to the user, then use an instance of a View to display it.

Don’t forget to inverse the dependency between your controller and the presenter by using an interface.

STEP 9. The view is the class that finally displays to the user the result of the use case. The view has no intelligence, and it simply writes at the console what the presenter prepared.

As always, you must inverse the dependency between the presenter and the view by using an interface.

STEP 10. It’s time to review your packages. At this point, you should add to your package diagram three packages: controller, presenter and view. Draw the arrows representing the dependencies between your packages.

STEP 11. Suppose that each of your package represents a component. Make sure that there are no cycles in the dependency graph. Compute the stability and the abstractness of each package (see lecture).

At this point, your “box and arrows” design showing classes and interfaces should be ready. You can start detailing the methods of your classes in a class diagram and also, draw your components diagram.