# Guided Project, Part 3.

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 implement the use cases that you designed in previous parts.

STEP12. Create your Maven project in IntelliJ containing 1 module to start with: core.

STEP13. Implement the entities, the interactor and the services you need for the interactor. Don't forget to create an abstract Entity class containing an Identifier (UUID in Java) that all your entities will implement. Don't create any gateway for now (no interface for your services).

STEP14. Test your use case by putting the main() method in the Interactor itself.

(At some point, you will probably find that you will need to handle different entities in your repositories. So it's a very good idea to use generic objects in your repository implementation. The generic objects will all have to extend your abstract Entity.)

STEP15. Now work on your datatransferobjects package: the Request and its corresponding concrete class, the Response and its corresponding concrete class, the IRequestHandler (implemented in the interactor)

STEP16. Create a second module in your IntelliJ project, called app. The app module will depend on core.

STEP17. In app, create a package named controller, you will put your controller classes in it. Each controller is specific for a use case. It will typically have a run() method which will loop waiting for the user to enter commands at the console. When a command comes in, it will:

1. create the request message from user input

2. instantiate the corresponding use case Interactor injecting the gateways (the concrete implementations of the gateways that are necessary for the use case can be stored as private attributes of the controller and instantiated in the constructor of the controller);

3. ask the interactor to do its job (handle the request), retrieve the response

4. instantiate a presenter, ask it to handle the response -> if you don't have a presenter yet, because we do it in a later step, just display the response to the console.

Move your main() method to the controller now and test your use case. main() will now just call run().

Optionally at step 2, you can use factories to instantiate the services. Put the factories package in app

STEP18. Now create a 3rd module called services and move the implementation of your services in it (repository, authentication, logging…), each service in a different package. Add the gateways interfaces at this point into the core module and adapt your Interactor to use those interfaces instead of concrete classes.

STEP19. Put the correct dependencies between your modules in Maven (look at the example in the lecture).

STEP20. Go back to the app module and continue with the user interaction part. Create your Presenter in the presenter package, use it in the controller.

STEP21. Create your view in a new view package, use it in the presenter to display the data formatted by the presenter.

STEP22. Invert the dependencies between view, presenter and controller by using interfaces.

STEP23. Move the main method out of the controller to a Main class at the root of the app module.

Your main will:

Instantiate the controller and call the method run() on the instance.

Test.

You're DONE !

For a second use case, you will need a second interactor, new data transfer objects and a second controller. How to deal with the user input at the console when you have two controllers? Think about using a Façade.