T7 - Application Development

T-DEV-700

Cash Manager

A payment service with a mobile terminal

1.5.0

EPITECH.

Cash Manager

binary name: cashmanager_*$AcademicYear*_*$GroupNumber*.zip
delivery method: Moodle
language: whatever works

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.
- All the bonus files (including a potential specific Makefile) should be in a directory named *bonus*.

The aim of this project is to build a distant payment system that can receive and execute orders issued by a terminal app on your phone.

The product in itself is very simple; here we insist on Quality. The project will be quite a journey in learning many aspects of application development: Oriented Object Programming, Mobile Apps, Design Pattern, Code Coverage and more generally developing **robust, reliable, multiple-uses** code.

Tools and Techniques

This project was designed to be realized in Java and Kotlin. Theses technologies are suggested, not mandatory

How to Work

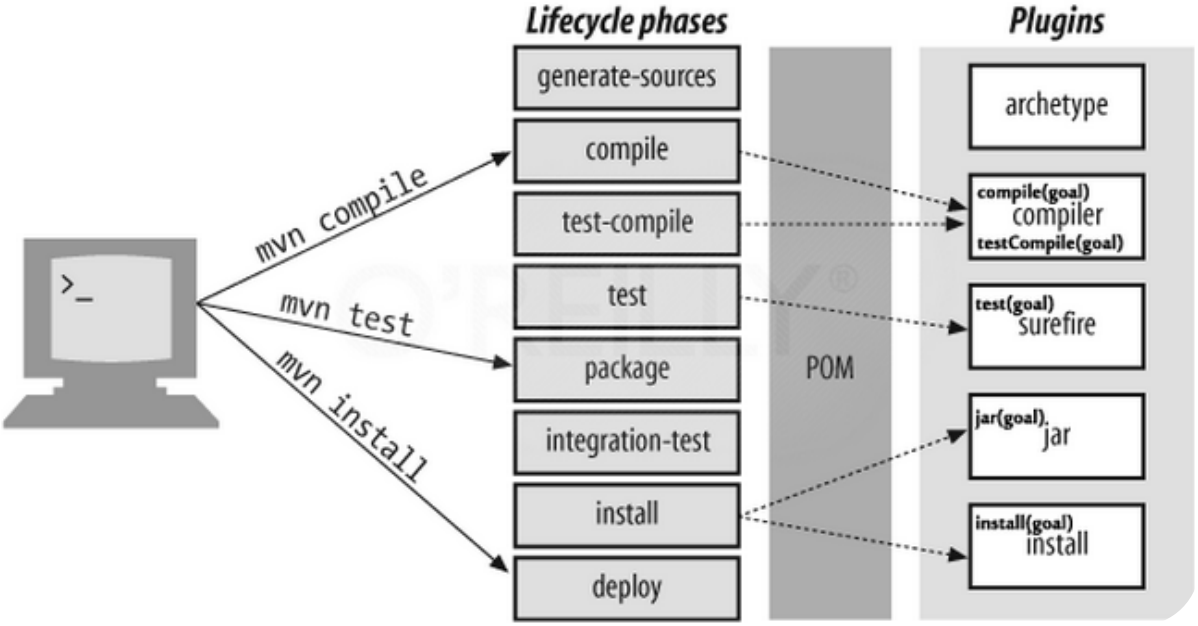
An idustrial project is not a single-use script. It is a whole ecosystem that is designed to ensure readability, robustness and portability. As a famous Epitech director is used to saying:

> *I would always prefer to re-write a program from scratch than go through someone else's code, unless, of course, it's written in Java.*
>
> *Axelle Z.,*

Java is not mandatory, but you should choose languages and writing practices with a single thing in mind: "any of my colleagues should be able to get into my code in no time."
For that reason we **strongly recommend** you follow the following steps:

- embrace classical design patterns
- implement a **P**roject **O**bject **M**odel, like Maven
- build a code coverage thanks to **U**nit **T**ests
- provide extensive functional and technical documentation

Project modeling is the basis of a development cycle. It handles the various steps of the making of a program, in a way that can easily be handled to another programmer on another device.



Project Construction

As part of this project, you will have to develop several parts, more or less independent, which rely on different technologies. The use of **Docker Compose** will help to homogenize the construction of such a project.

You will have to make a **docker-compose.yml** file at the root of your project, which will describe the different docker services used. This file must include *at least* the following two docker services:

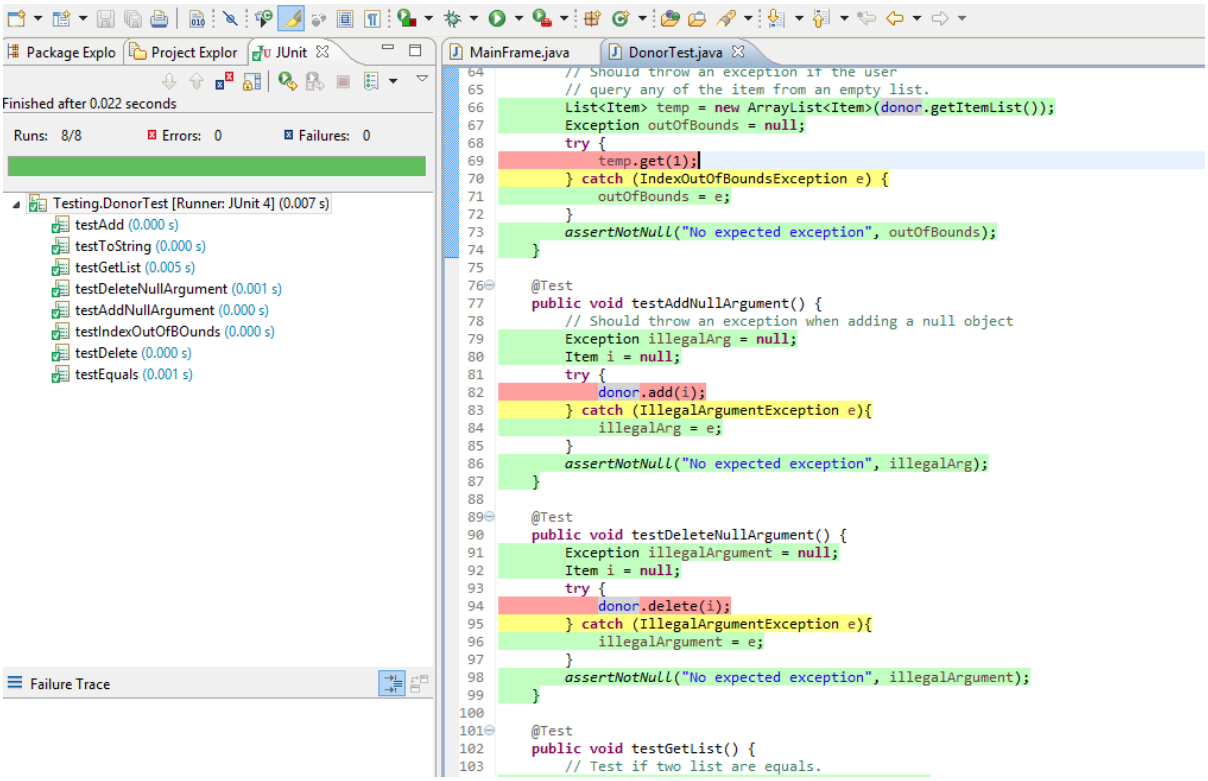
- A *server* service to launch the server on port 8080
- A *mobile* service to build the terminal. The APK file must be build at this moment.

Validation of the integrity of your images will be done when launching the docker-compose up command. The following points should be respected :

- The services *mobile* and *server* will share a common volume
- The *server* service will run by exposing the port 8080
- `http://localhost:8080/client.apk` to provide the terminal application

Code coverage

You probably already have some standards about code quality, this is great practice. Here we insist on unit tests, and the integrated tools Java provides.



The point of unit tests is to check every possible behavior of elementary pieces of code given various inputs. A bundle of unit tests for a given method, class, package, is a way of securing this part of the code's reliability. With a good coverage, you are sure to be able to re-use parts of the code, and to detect regression problems (i.e. when changing your code in a place has unexpected impact on other parts).

Java, and some other languages, offer you integrated tools to deal with unit tests. Typical solution is to mirror every class with a test class that check and ensure correct behavior.

Design patterns

In the industry, the focus is to rely on robust solutions that can be easily understood by partners.
However imaginative and fancy it is, a piece of code only you can understand is little worth. In addition, many unique problems rely at their core to a single pattern that has probably been encountered thousands of times in the community. This factorization (identifying a recurring problem and adopting the best solu-

tion to our knowledge) is called a design pattern.

We ask you to identify design patterns that are relevant to your project and choose the ones you find suitable. Your documentation is expected to reflect this choice explicitly and the organization of your code to follow it.

Specifications

The terminal

In this project you have to make a **mobile app**, which will be your terminal. Users will use the terminal by querying the server. Your terminal is supposed to let you add articles to a cart, then proceed to payment. The terminal **must** provide a way to configure the network location and the password of the server with a setting view.
The payment process must allow credit card and cheque payment. Before any payment, the terminal must be connected to the payment server, then scanning can proceed and finally payment attempts. The payment server can allow or refuse the payment, for instance if security codes do not match.

You must be compliant with the material which is composed of **four screens**:

- a setting screen allowing the user to connect the terminal to the server (including login/password input fields)
- a screen displaying the cash register, where the user adds articles and the bill is updated in real time
- a screen displaying the bill's total (no specific action but proceed to payment)
- a screen providing the user payment. It allows the user to scan a card (**NFC Reader**) or a cheque (**QR Code scanner**) and displays the card payment status: pending authorization, payment accepted, payment refused.

The bank server

On the other side, you are expected to build a server application that will handle requests from the terminal. This application will:

- receive requests
- accept or refuse authentication based on stored data (any authentication method will do)
- fetch data from the bank account (any storage method will do)
- accept or refuse the payment, based on credit card/check information and money reserves

- notify the mobile app in return
- update the user's account according to the transaction

Providing an admin interface for the server is not required. On the other hand, some settings should be configurable on a file (number of successive wrong card or cheque, maximum cost of a transaction, number of transactions per unit of time, etc.)

Delivery

The delivery consists in a **docker-compose.yml** file which contains the services to compile, test, package and deploy your server and your mobile app.
Your programs will be **duly documented**. In addition, you **must provide** the following documentation:

- **Software Architecture Specification**. Since the functional specifications are given, we now need UML diagrams that describes your understanding of the specifications and the solution you intend to implement. Here we only ask you to deliver a *class* diagram, though you may feel useful to use *use-case* and *activity* diagrams.
- **Software Qualification**. In addition to the unit tests, which are included in the project, you must describe all the case studies (functional tests) you intend to conduct in order to meet the client's demands. As a bonus you can automate that procedure on an automated testing platform of your choice.

Bonuses

You can improve this project in many ways, including:

- adding a real authentication protocol with security checks
- building an admin view for the server (either directly with a java graphical lib or as additional features for the android app that are available only for admin people)
- conduct a full functional testing process through an external platform according to tour qualification specs
- adding formal specification documents, including various uml diagrams