Exercise 1: Refactoring and Design Principles

16.10.2025

Presentation of results in next lab /
Submission in AULIS until 23.10.25, 9:45am

Assignment 1: Refactoring

This assignment is about refactoring, i.e. the re-structuring of an application in order to improve its design and code quality while preserving its functionality. Many refactorings are supported by modern IDEs such as IntelliJ IDEA or Eclipse.

The Object of Refactoring

The *OrderSystem* program facilitates ordering of up to 5 products (e.g. “notebook”) and up to 5 services (e.g. “cleaning”). While products have a unit price and a quantity, services are billed by the number of employed persons and price per hour. After finishing the order process, the application outputs the selected items (ordered by price) and the resulting sum.

Installation and Start

- Download the project from AULIS.
- Since the program is created with IntelliJ, importing it into IntelliJ should be straightaway. You may of course use other IDEs such as Eclipse (please agree on *one* IDE within your working group).
- Make sure you can start the program and make yourself acquainted with its use.

Program Analysis

Have a close look at the source code and try to identify existing deficiencies: Which drawbacks and weaknesses does the current solution have? (Don’t read the following subtasks for now, but think yourselves!)

Subtask 1.1: Basic Improvements

First, your task is to improve the code's basic structure and its readability:

- Introduce packages such as *model*, *frontend*, *data*, *utility*, *service*, *database*, *ui*, … (your choice!)
- Currently, the application stops when the order has been finished. Allow to place another order after finishing an order (and another and another…).
- Whenever you find flaws in the code's syntactic structure or naming or …, improve it!

Subtask 1.2: Management of an Order's Items

So far, up to 5 products and up to 5 services may be ordered at once. The selected products and / or services are held by the `OrderService` instance. The next goal is to be able to manage an arbitrary number of products and / or services in one order (i.e. no limitation to max. 5 items):

- Introduce a class `Order` holding the selected products and / or services.
- Which Java data types (instead of the currently employed array) are particularly suitable for managing the selected products / services in the order?
- How can you solve the problem of sorting the products and services by price better (or simpler) than with the currently implemented BubbleSort?
- Have a look at the Java Collection API!

Change the program accordingly and test your solution.

Subtask 1.3: Unified Handling of Products and Services

So far, the application does not make use of the structural similarities of products and services: it implements two arrays (or whatever data structure you replaced the arrays with), two sorting methods, and uses two similar loops when finishing an order. The goal of the subtask is for the `OrderService` to be able to handle products and services just as orderable *items*.

Some advice on this subtask:

- Introduce a joint supertype (e.g. `Item`) for products and services. Think about inheritance and polymorphism. Is an abstract class or an interface adequate?
- Manage both the selected products and services in *one* shared data structure of your `Order` class. Sorting this data structure may mix products and services (unlike in the original version).
- The `OrderService`'s `finishOrder()` method shall set the checkout date and time in the `Order` instance, moreover the `Order` class shall be equipped with a getter for the lump sum. After printing the order, a new order will be created, i.e. a new "session" will be started.

Apply the required changes to the application.

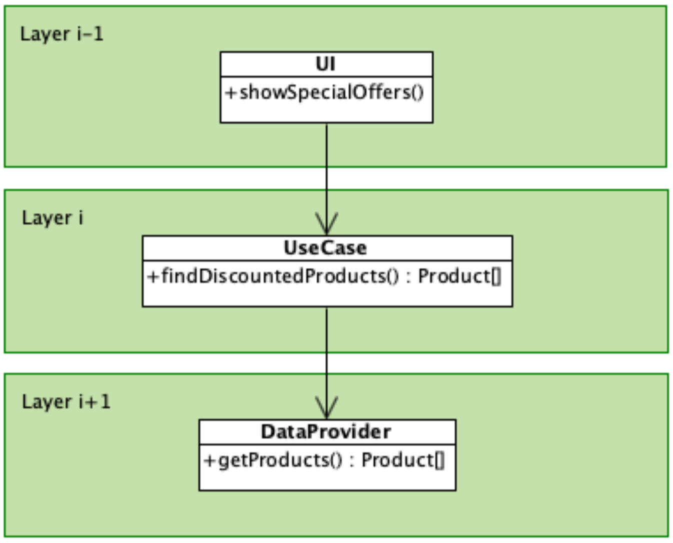
Test your refactored application.

Is the resulting `OrderService` considerably shorter and more readable? Is it more flexible?

Note: *You may work on these three subtasks in one development project, i.e. you don't have to present the results of these subtasks in individual projects.*

Assignment 2: Dependency Inversion Principle

This assignment is about the dependency inversion principle (DIP). Consider the following layered architecture, where a `UI` is (among others) able to show special offers. In order to do so, it asks the `UseCase` for discounted products, which again retrieves products from a `DataProvider` and checks for discounts.



In this layered architecture, higher-level modules depend on lower-level modules. Moreover, we can state that the `DataProvider` (be it a local database or some provider that retrieves data from a network) is a detail. The relationship between `UseCase` and `DataProvider` therefore violates the dependency inversion principle:

> a. *High-level modules should not depend on low-level modules. Both should depend on abstractions.*
>
> b. *Abstractions should not depend on details. Details should depend on abstractions.*
>
> [Robert C. Martin, Agile Software Development, 2003]

In the following subtasks, we focus on the two lower layers *i* and *i+1* only.

Subtask 2.1: Dependency Inversion between two Layers

Invert the dependency structure between `UseCase` and `DataProvider` such that the dependency inversion principle is respected. Of course, the interaction between both classes outlined in the introduction should still be supported, i.e. the `UseCase` should be able to retrieve product data from the `DataProvider`. Your result should be a class diagram that clearly shows relationships (dependencies) and the assignment of classes to the two layers and additionally a short textual explanation of how your design is supposed to work.

Subtask 2.2: Updates from lower Layers

Let's assume that the `DataProvider` is retrieving and saving data from and to a network. At times, the `DataProvider` receives updated product data. Because – in the long term – we would like to refresh our UI, we want our `UseCase` instance to be notified of updated product data.

Extend your class diagram from subtask 2.1 and make the `DataProvider` *additionally* send updated product data to the `UseCase` instance through a new method *UseCase::updateProducts(products: Product[])*. Important: of course, we don't want the `DataProvider` to directly depend on the `UseCase`. Details should depend on abstractions... Again, provide a short explanation of the interaction between the participating classes / objects.

Hint: For modelling purposes, you may use Visual Paradigm (cf. slides *MKSS_00 – Organisation.pdf*).