Architectural decisions and highlights for the Weekly Restaurant Evaluation example

PV, 2015-12-03

This is a simple document explaining the basic packages of the project. We start with the **view** part, then move to the **business logic** part and end with the **data model** part. Other structures are of course possible; several improvements have been omitted; many chances for extensions are present; and of course, you have to think about this stuff too – hopefully, you will put your hands to work and improve it.

Any bugs are solely my fault. If you find any, plz., let me know.

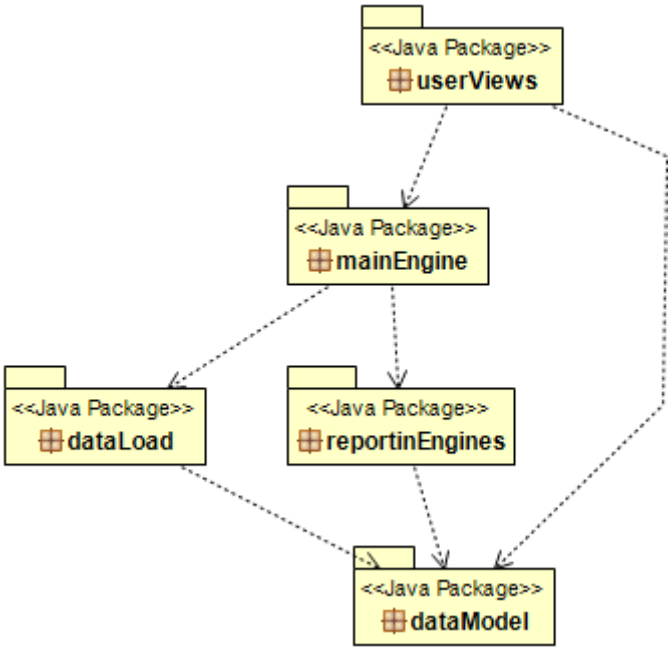

Fig. 1. Package diagram of the Weekly Restaurant Evaluation project

userViews

The simple user interaction class (**SimpleTryUserMain**) comes with a simple **main method** that shows the basic functionality. See that there is a single dependency to the MainEngine and no other dependency (in principle, this cannot happen all the time; sometimes you cannot avoid dependencies to lower-level packages). **See how minimal the task of the client is and how we avoid adding any logic to it** (this is a case with a thin client and a strong back-end application)**.**

The rest of the classes involve a **JavaFX** GUI (Food for thought). The alternative **main method** is at **ApplicationMainGUI** class. See how the simple tasks are part of the start() method without any difference from the simple main method.

For you: if interested, you need to spend some time with JavaFX, a GUI platform of Java. The model used by JavaFX is fairly representative of modern GUI platforms. Try for example, to add a bar chart on the employee evaluations or salaries.

If NOT interested, you can stick to the SimpleTryUSerMain class.

mainEngine

The MainEngine implements all the major use cases of the program: (i) load all data, (ii) compute all stats and (iii) create reports.

Observe how frugal we are with this class. We do not export almost anything else (see next).

I have added a couple of methods to the main engine to allow the JavaFX GUI communicate with it. This is due to the fact that the JavaFX part requires observable lists of dishes and employees, so we populate them.

Observe how we added the JavaFX part as a parameter in the methods **getDishes()** and **getEmployees()**. This is how the engine part is independent from the "upper level" (here: GUI) package: we inject a parameter in the method of the lower-level, depended-upon class (here: main engine). Then, the parameter gets populated inside the method => there is no need for the lower-level class to know who is calling the method, or any other info. Thus, all the dependencies point downwards and no cycles are introduced in the package diagram. The price to pay was the injection of a dependency from the "lower-level" class to the dataModel classes involved in the parameter (here we would not avoid it anyway, but in other cases, it can be a price to pay).

reportingEngines

Part of the business logic with a clear abstract coupling mechanism: an interface + a factory to be reused by the callers.

dataLoad

This is another business-logic package that comes with two "sub-parts". The "upper-level" part involves the implementation of the Full Data Load functionality and is the façade that links the internals of the class to the main engine. This is why the **Full Data Loader** family has (i) an interface + (ii) a factory + (iii) an interface implementation class.

The functionality of loading each of the three files is delegated to a dedicated data loader. Observe the "Template Method" pattern here: there is a generic **Abstract Record Loader** class that performs the same task (open a record file, read line by line, and, for each line, split the line via a tokenizer to an array of strings and pass it to an object creation method, for an object to be created) independently of which file/target class it is addressing. The

specifics are handled by the abstract constructObject() method that is overloaded by the concrete subclasses of AbstractRecordLoader, one per target class. Observe the use of generics here (there are some typing constraints that prevent simpler solutions).

For you: how would you perform abstract coupling here? How can we reduce coupling?

dataModel

Observe the Abstract Class **Employee** along with (a) its subclasses and (b) its factory.

Employee and EmployeeFactory provide a clear case of abstract coupling with the rest of the program (you can observe how we avoid referring to the subclasses at the dataLoad.EmployeeLoader class)

WeeklyStats is a helpful class with static statistics that enables to compute ChefDeCuisine salary and avoid linking the class to the MainEngine at the same time.

Food for thought: how would this task be facilitated if the weekly stats class was not there?

Other material

Input and output files are found in the respective folders. The requirements document is found in the respective folder. Some ObjectAid ucls class diagrams accompany the application.

For you: how would you perform **testing** here? **Documentation**?