Peer Review for Group OOPSIE

Samuel Kajava, Daniel Rygaard, Pouya Shirin, Emil Svensson

October 2021

1 Design - patterns and principles

1.1 Reusability

The code seems to be reusable in many aspects. A good example of this is the way the frontend has been implemented: the UI elements are split up to its own components, making it possible to reuse the same controller/UI element on different pages within the UI. The classes ViewComponent and JavaFXViewInitializer is a good boilerplate and could be used for other projects that wants to use components together with JavaFX. The same could be said for the class EventList, which is a good start for other projects that needs an event handler.

It could perhaps be possible to improve code reusability by making some of the classes/methods more generic (FileHandler for example). That would allow the codebase to reuse parts of the codebase for other types of file handling than attachments and images.

1.2 Maintability

Maintainable code is code that is easy to modify or extend. In order to follow this they are using a significant amount of interfaces. For example, the interfaces IDatabaseLoader, IDatabaseSaver or IFileHandler are utilised in such a way that they make for a interchangeable save and load system. Therefore, if the data is transferred to another sort of medium, the program can be adapted with not much effort. However, there could be a support for fetching from multiple different databases.

1.3 Modularity

To judge if a code is modular or not, we have to define what modular programming is. A quick search on wikipedia tells us: "Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality".

Taking the model package as an example, we can see that the model classes are separated in their own area of responsibility. User appears to be the main

model, carrying references to the lower level modules. The lower level modules contains the logic and functionality. Then two separate packages, FileHandler and Database, being accessed with their respective Factory classes.

Which results in the project appears to be following modular programming quite well. However a suggestion for improvement would probably include separating each of these modules in their own packages.

1.4 Extensibility

In regards to the Open-closed principle, modules should be open for extension and closed for modification in order to make it easier to extend the project in general. In the way you are using interfaces, I would say you are following this principle quite well. Let's look at IPageNavigator as an example, the interface is small, which is good, and if you wanted to extend the functionality and make another, more specific navigation tool, you could simply implement another interface (that implements IPageNavigator) with the extended functionality.

Another good implementation is the use of the abstract class ViewComponent and its factory. By simply extending the ViewComponent you can create new components without much hassle, which in my opinion is a good solution.

1.5 Design patterns

To achieve the above-mentioned design principles, a number of design patterns have been utilised. The observer pattern has been used to properly implement the MVC structure and isolate the model from the views. The factory pattern is used in multiple packages (which helps out achieving the Open-Closed Principle).

2 Code quality

2.1 Naming

Classes and method names appears to be following the correct naming conventions well. Just by taking a quick glance at the class names gives you a general idea of the class' content, its function and area of responsibility. This results in an easier navigation through the project and the understanding of class interactions.

2.2 Documentation

The code is very well documented with JavaDoc. The IFileHandler-interface is a good example, as I am not familiar with handling files. Despite this unfamiliarity, I am able to grasp what it means, and this is thanks to your documentation. The same can be said for JavaFXViewInitializer, I understand the code and agree with what you have written in the documentation, well done.

2.3 Tests

The test structure is very similar to the main structure, containing test classes for the Model and File Handler package. There's no test for the Database package. Code is well tested.

2.4 Understandability

The project has an easy identifiable MVC structure which you'll see by looking at the project files. The model package is split up in different model classes and the controllers each have a reference to them.

2.5 Improvements

A few things to consider, or maybe discuss:

- Is there any reason to not singleton most of the classes? For example, the Database class; will you be having more than one database? Right now, you can create a Database anywhere since the class is public. (Perhaps you want another Database if you want to load from two data types simultaneously, but that would mean loading two Users? And the Database Factory is made to only utilize one database)
- Same applies to the JSONDatabase classes. But will you ever need more than one JSONDatabaseLoader?
- You seem to be returning whole mutable objects in User (Eventlist, Contactlist and Taghandler). Normally I would be wondering if this was a case of breaking Law of Demeter, but it seems unavoidable?
- One thing I noticed though was the returning of lists. An example is in ContactList. The ContactList.java is a wrapper for contactList, however what use is a wrapper (except for the events) other than hiding the actual list object? Right now getList() is returning the whole list object (which we learnt in OOP is big no no). However the ContactList wrapper gives you the methods to mutate it anyways, so really what is the difference? One advantages that comes to mind is: preventing alias problem. You want to minimize the amount of unexpected issues in your program. Someone modifies the list outside of your wrapper and now notifyObservers won't be called.