Peer Review of group 13 - Lagersystem

Henrik Lagergren, Markus Pettersson, Aron Sjöberg,
Ellen Widerstrand, Robert Zetterlund
26/10/18
v.1.1

1 Peer review

This is a peer review of group 13's project *Lagersystem*. Aspects such as how the code is structured as well as if it is easy to understand has been considered, among other things.

1.1 Maintainability

1.1.1 Code documentation

As it stands, the codebase is not very well documented. Only classes in the View package, besides two classes from other packages, contains any sort of JavaDoc. However, much of this documentation is redundant, as it covers things as getters/setters and not actual business logic. A lot of code throughout the codebase is commented out without any explanation as to why.

The classes *Admin*, *Writer* and *Reader* are identical, resulting in redundant inheritance, however, as noted in group 13's SDD, this is the foundation of an inheritence-based design based upon the following inheritance: *Admin* <: *Writer* <: *Reader* <: User. Refactoring core functionality in the User class will transfer to the subtypes, in some sense allowing for self maintaing code. However, this tight coupling makes it hard to refactor logic without introducing unintended side effects. It is mentioned that they strive for LSP, Liskov substitution principle, which is conceptually correct, but different roles shouldn't be inherited, they should be delegated [1]. Therefore it could be argued that they should avoid using inheritance.

The views of the application is initialized using a method *start*, meaning no use of a interface markup language, resulting in a lot of code configuring the layout. This approach makes modifying the views difficult, meaning it reduces maintainability.

1.1.2 Names used

The names of the classes are fairly satisfactory, because of the functionality being based in the real-world. For example, the *Order* class is supposed to represent a real world order and the class does encapsulate what can be expected of an order pretty well, e.g. an order having an order number and data on which that order was placed.

1.2 Design Pattern implementation

The SDD states that the project is using a Factory pattern. The purpose of a factory pattern is to be able to avoid dependencies on specific implementations and instead rely on abstractions [2]. More specifically it should work as a facade to initialize objects of a certain family or sort, which allows for full control of what is shown to the rest of the program as well as decreasing dependencies. As stated in the SDD, the project is trying to implement a Factory Pattern. The implementation however, does not fit the description of the pattern. The only Factory class in the project is the CMSFactory and it does not provide any functionality for instantiating objects. It instead seems like it holds/handles all the observers and controls when their "onAction-methods" are called. This class name makes it more difficult to understand the code.

The Observer pattern implementation is also a bit difficult to get an overview of. The pattern usually indicates that there are two interfaces, an Observer-interface and an Observable-interface[3]. The implementation here only has an Observer interface and the CMSFactory sort of serves as the observable object for all the observers in the project. This makes it difficult to see what observers are called at what time, which could lead to unpredictable behaviour. A possible solution would be to create an Observable interface and have the desired classes in the Model-package implement it. This way an observer could subscribe to a more specific part in the model, which would make the program easier to understand and easier to Maintain/extend.

The code does have an MVC structure and the model is isolated from the other parts. The only way the model can communicate with other parts is via *observer-pattern*.

1.3 Tests

The Model-package is the only package which actually has been tested, with a method-coverage of 71%. The downside is that several of the written tests are for getters/setters, which might be unnecessary. Tests should aim to evaluate behaviour of the code, so instead of write tests for getters/setters should the focus be on real functionality. Tests for both the controller-package and view-package have opportunities for development with a coverage of 0%.

1.4 Security problems

Considering the program is an inventory managing system, where data should be immutable most of the time, variables that are "passed-by-reference" should

be sparingly sent around, but it is not. For example, the class *Order* has the String OrderNr which is fetched using:

```
public String getOrderNr() {
   return orderNr;
}
```

Improvements could be made by ensuring immutability of an *Order* object by returning a copy of the variable. However, it is deemed unsafe to keep this implementation.

The application begins with a login-screen which only can get passed with a correct username and password. However, at the moment, it is possible to login with any random name and password. As a user you have the power to delete articles from the inventory. This is a security issue which does not make the application particularly safe to use.

References

[1] Alex Gerdes. "Principles of subclasses". [Online]. Available: http://www.cse.chalmers.se/edu/course/TDA552/files/lectures/tda552$_lecture2-2.pdf, retrieved : 2018 - 10 - 23.$

[2] Tutorialspoint. "Factory Pattern". [Online]. Available: https://www.tutorialspoint.com/design$_pattern/factory_pattern.htm, retrieved :$ $2018 - 10 - 25.$

[3] Alex Gerdes. "Observer Pattern och MVC". [Online]. Available: http://www.cse.chalmers.se/edu/course/TDA552/files/lectures/tda552$_lecture5-2.pdf, retrieved : 2018 - 10 - 25.$