Peer Review of Workshop 2 on Carolina Skov Pedersens group.

By Daniel Svensson, ds222ey.

I'm am using the e-book version of Larman's *Applying Uml and Patterns* which, annoyingly enough doesn't have pagination corresponding to the printed book. Therefore I include both the e-book (pdf) page number and the section or figure where the reference can be found.

Test the runnable version of the application in a realistic way. Note any problems/bugs.

Crashes if non-numeral is entered where numeral is required. Crashes if entered personal id is too short. Crashed if n.n is entered instead of n,n for boat length. No boats in view specific member. Possible to register boat without type. Edit boat type results in numeral instead of boat type. Personal number can be entered (and saved as) either xxxxxx-xxxx or xxxxxxxxxx which looks a bit confusing.

Does the implementation and diagrams conform (do they show the same thing)? Are there any missing relations? Relations in the wrong direction? Wrong relations? Correct UML notation?

From looking at the class diagram it's not clear what the arrows mean. At first glance one can assume that the pink/purple arrows are associations, the green ones inheritance (super-/subclass) and grey ones dependencies but this is uncertain because it doesn't follow notation standards [1. p 397, fig 16.9, p 567 fig 23.2]. Neither role names or multiplicity are present although they are present in the individual class definitions.

Isn't the relationship between Controller and Member an association (controller->member)?
In the implementation Controller has dependencies to both MemberController and BoatController which are missing in the class diagram. MemberController might also have a dependecy (I'm not sure if this qualifies as a dependency) to Member which is not present in the CD. BoatConsole has a dependency to Boat not shown in CD (uses BoatType and Length present in Boat). MemberConsole has dependencies to both Member and Boat shown as associations in the CD. Boat editor has a dependency to Boat not shown in the CD. RegisterEditor has a dependency to Boat not shown in the CD.

Architecture

Is there a model view separation?

Yes.

Is the model coupled to the user interface?

Models has no dependencies to views.

Is the model specialized for a certain kind of IU?

Generally, no. Models simply return values as they are. It is possible to argue some form of specializaton due to the fact that the model expects index numbers for boats to be “one too high” (the zero-index problem). Because there is in fact a Controller layer, perhaps this calculation should be done there since the Controller knows the view and hence knows what values to expect. Also, the Member model stores the personalIdentificationNumber as a string instead of a long which can be viewed as model conforming to view rather than the opposite.

Are there domain rules in the UI?

No

Is the requirement of a unique member id correctly done?

Yes. Not sure what correctly means but it works.

What is the quality of the implementation/source code?

Good, descriptive method names which supports a low representational gap [1, p228, fig 9.6]. MemberRegister in the Controller and subsequent functions regarding the same are dead code. Perhaps the idea was to use this in some way but that would result in some form of code duplication, or functional duplication. E.g. when user wants to list members the list of members

are first aquired by start() in Controller and then again, later in the sequence, by MemberController.

Naming of the Editors (RegisterEditor, MemberEditor, BoatEditor) are somewhat poor choices since doesn't clearly show the fact that RegisterEditor is the parent/superclass [1, p695, sect 32.2]. The superclass' name should be appended to the subclass' name [1, p568, sect 23.2]. In this case that would mean be a superclass named Editor and subsequent subclasses called MemberEditor and so on.

What is the quality of the design?

Objects are connected using associations and not with keys/ids.

Yes, associations.

Is GRASP used correctly?

Some classes have good GRASP implementation. However the Editor classes are somewhat uncertain in their responsibilities. RegisterEditor has the MemberRegister variable and because of this has the knowledge about Member(s). But still the MemberEditor has a lot of the logic for actually adding, deleting and editing members. Of course, the MemberEditor, by being a subclass to RegisterEditor, also has access to the MemberRegister. However this is further confused by the fact that RegisterEditor also is directly responsible for updating the persistence register. The only real reason for the subclass-superclass relationship between MemberEditor and BoatEditor on the one hand and RegisterEditor on the other seems to be easy access to the MemberRegister variable and one or two methods. The result is a strong form of coupling which should be considered carefully before use [1, p446, sect 17.12].

Classes have high cohesion and are not too large or have too much responsibility.

See above.

Classes have low coupling and are not too connected to other entities

See above. The controller Controller also has a MemberRegister instance variable which is referencing the same MemberRegister in RegisterEditor by asking the MemberEditor for it. Is this really necessary when Controller also has an association to MemberEditor and by extension

access the same information twice. This results in an example of unnecessary coupling, especially so since the only use of the association to MemberEditor is to fill the Controllers instance variable MemberRegister with Members [see 1, pp444-447, sect 17.12].

Just from looking at the class diagram there seems to be a lot of associations going on. Is it really necessary, for example, for the BoatConsole to be associated to BoatEditor, MemberEditor, BoatConsole and Console and in addition having a dependency to Boat? I get that the responsibilities in this way are distributed but it seems to be adding a lot of coupling.

Avoid the use of static variables or operations as well as global variables

No static or global elements.

Avoid hidden dependencies

No hidden dependencies that I have noticed.

Information should be encapsulated.

Good information encapsulation. Getters and setters instead of direct access.

Inspired from the Domain Model?

Yes

Primitive data types that should really be classes (painted types)

No

As a developer would the diagrams help you and why/why not?

To some extent they would help. The notation in the class diagram is somewhat confusing however. The sequence diagrams doesn't start with the specific call for the individual functions in the application. Instead they are started with the start() function to the Controller. I realize that the start() is the starting point in a sequence which leads to the Controller telling the Console to display the menu and returning user input which translates into "view member" or "register member" and so on. This adds a lot of clutter before getting to the actual sequence. It would be

easier to understand if the Controller got a generic “Do this” message, for example “Register member”, from some :Actor even though such a method doesn’t exist. This is the real starting point for the Controller telling the model to register a new member for example. The actual sequence from start to menu to getting user menu input could, if necessary at all, better be shown in it’s own diagram. That is, of course, my opinion or rather interpretation of the diagram. But it might be of value to note that Larman states that the interaction doesn’t have to start with a system operation message [1, p486, sect 18.4].

What are the strong points of the design/implementation, what do you think is really good and why?

In summary: Generally good descriptive naming of classes and methods. The model and the view are separated. The controllers seems to have muddied the waters some though. But the view is definitely separated from the Model. The Model doesn’t seem to be specialized for the UI either which is good. The information in the models is encapsulated by getters and setters, with the exception for the list of Boat(s) in Member (List<Boat> is public). However, perhaps it’s not necessary for all properties to have getters and setters. The list of boats, I assume it was meant to be private since it has a getter and setter, might not need a setter since the list is set and filled at Member creation and there is, at this iteration, not really a need to set it again later.

What are the weaknesses of the design/implementation, what do you think should be changed and why?

In summary: UML notation for the class diagram; Pink, grey and green for association, dependency and superclass/subclass are, to my knowledge not standard.. Unnecessary clutter in the sequence diagram, starts “too early”. Runnable crashes when getting unexpected input. Also some problem with Boat registration and editing. Problems with cohesion, coupling and responsibilities. All of this is covered in the review.

Do you think the design/implementation has passed the grade 2 criteria?

No, not in my opinion.

References

1. Larman, C, Larman C., Applying UML and Patterns: An introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, 2004, ISBN: 0131489062. E-bok: https://aanimesh.files.wordpress.com/2013/09/applying-uml-and-patterns-3rd.pdf