Reviewed by: Sing Trinh dt222cc@student.lnu.se

- For Mathias Lundberg (ml223nw):
- https://github.com/ml223nw/1dv607-workshops/tree/master/Workshop%202

Intro

Some feedback can be wrong as I do not exactly know if I'm correct or not. Lack of references (no access to the course book atm). The application did not crash and handled exceptions well enough but could use some improvements ofc.

Is the Architecture ok?

The Architecture needs a bit of improvement. There is MVC separation but there is some violation like Controller having model responsiblities like adding a new member to the list, which should be handled in a model class. I believe the controller was not supposed to alter the domain but use the models to alter the domains.

Otherwise the architecture is good.

The unique member id are being randomised with no need to input a memberId during registration. (I wanted to have some kind of auto memberId but did not implement it). The Id length/size might be a bit overkill. Perhaps use memberId as the input instead of fixed row number. The member on "row" 7 might not be the same member another day.

What is the quality of the implementation/source code?

All the CRUD functionalities are included and the code is easy to read and understand. There's no comments so it's up the the reader to interpret the code, misunderstanding can be a thing.

Methods are named well, easy to understand what the methods do. The naming of parameters/arguments are fine: camelCasing. The naming of methods also fine, PascalCasing. No small letters are being used, good, which can in some case make it harder to understand (if you have a couple of different letters).

Naming conventions: (https://msdn.microsoft.com/en-us/library/ms229002(v=vs.110).aspx)

- Naming of public and private members/fields: camelCasing w/o underscore and PascalCasing (followed but not consistent, Controller.cs, DAL.cs)

I see some dead code (the "internal" stuff):

- Controller.cs line 20: Model.Member Member
- Controller.cs line 32: Model.Boat Boat
- Boat.cs line 59: View.Menu Menu

Error handling regarding input is handled well, no crashes. No typographical errors :D

Bool variable for Do-while cases not necessary, do{} while (true), works fine (have return; instead of break; to exit).

MenuView.ClearConsole() could be applied inside the MenuView.MenuShow() and the other similar methods, to make the code even more clean.

No duplication of code I think, nice. Methods on line 292 and 300 in Controller could be combine with a bool useReadLine as a parameter to pick readline or readkey.

The model throws exceptions(so does mine) and are not specialized to return strings.

What is the quality of the design? Is it Object Oriented?

Objects are connected using associations instead of using keys/ids, good.

Do more Object Oriented Programming. Instead of sending the different data: example Controller line 239 ChangeBoat and Menu line 89 ShowBoat. Send the boat object instead of the type and length which can be accessed in Menu, example:

```
public void ShowBoat(Model.Boat b) //or boat, in this case b seemed more appropriate
{
    Console.WriteLine("\n\nType: {0} | Length: {1}\n", b.Type, b.Length);
}
```

There is no hidden dependencies and no use of static variables or operations (I think) as well as no global variables, great.

Information should be encapsulated (OOP-Encapsulation). Lots of methods in Controller being public instead of private. Also Member.cs: line 85. Which means we can do stuff from outside the intended place. Think public when being accessed from outside the class and private when being only used withing the class.

Primitive data types that should really be classes (painted types) like List<model.Member> listOfMembers to a MemberList.cs. With methods like AddMemberToList() etc (can be wrong about this ofc).

I believe stuff like handling input is a view responsibility and should not exist in the Controller. Like _menuView.GetIntFromUser() or something similar. Can be wrong about this though. Think about it, the view presents info to the user and relay input from the user to the controller which then does something with that input.

Controller seems to have model responsibilities, with adding/removing member to the list, editing member. Repeat the way you did adding boat (which was using the model to alter the domain). So some refactoring to do.

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

The class diagram can be helpful to some degree. We have a Controller, DAL, Member, Boat and Menu and the associations between them (note: I do not know how much the diagram was supposed to cover, MVC or not). The class diagram probably shows the same thing as the implementation.

Just seeing the classes and I kinda get an idea of what this is about. The associations are not so helpful with the confusing association name like "Member" as the class name and also "Member" as the association name.

There's a bit lack of information, some kind of prefix would work, m.Member, c.Controller, v.Menu.

There is no attributes which helps in some cases with working with the associations. Also to help the viewer to understand what the class is for, which can be helpful when starting to get into the implementation.

The association between Member and Boat are missing (I think)because the Member has a list of boats. The association between Controller and Boat is not neccessary as the Controller uses Member to add boats into that member's boatlist.

Some sequence diagrams would be helpful. Alternative paths are mentioned, which is good. There's a clean sequence and easy understandable, how this specific method would run. There's some more complex which needs source code to better understand it.

The sequence ChangeBoat seems way more complex that it needs to be. Some text are clipped off which isnt that good. Few details for ChangeMember are missing (does not show the same as the implementation) the changing name/ssn is missing (after the getting userinputs).

Improvement (optional): "foreach member in members" instead of "foreach (members)".

Do something with clear() as that is not important enough to be in the diagram, have it moved/included in the different display methods instead to make the diagram and code more "clean". <<return>> is optional and in this case not so useful. Consider not having them for methods with no return values.

Perhaps "string name = GetNameFromUserInput()" instead of "HandleMemberInputName" (optional? for me it becomes more "helpful" which is kinda the purpose of the diagrams, this means some changes to the implementation). It seems that the diagrams where more modeled after the implementation than the other way around.

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

The mentioned good qualities from above like, organized code and not to complex to understand the code. Not so hard figuring out what classes has what responsibilities. Code isnt duplicated to much.

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

With the mentioned negative notations from above. Perhaps more work could be put into the diagrams early on to shape the implementation better. Then ofc make changes to them if it didnt work as intended. Do you think your diagrams have what it needs to be able to relay "good" information and provide some better understanding.

Else it's pretty solid.

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

I believe it is sufficient, there's areas that needs some improvements, regarding code qualities, Object Oriented Programming and the diagrams. With a bit more work with tweaking and refactoring stuff, Mathias should be able to pass this workshop.

And once again excuse me for not being able to provide more references. Most of this feedback can be considered as opinions rather than facts :D so disregard some notation if you think that I'm at fault/wrong. Just adding my point of view for your work :D