Peer review for dv222bk by kk222hk

Runnable files
Compiles and runs with no problems. Tried all the use cases and it didn't crash. Some usability problems, like deleting members without confirmation, but that wasn't necessary for the workshop.

Diagram and code correlation
In the class diagram, it's a bit hard to see, but it looks like there's no dependency between User and Member? User gets Member objects as parameters in methods, eg. MemberResponse(), and calls Member methods like GetBoatList(). That's a form of dependency/coupling according to Larman in chapter 17.12 : ”A TypeX object calls on services in a TypeY object”.

Otherwise the class diagram is great and shows the important operations and attributes. It's good that you used – and + to show visibility, as Larman did in figure 16.1. I think it's easier to see which operations are supposed to be used inside a class and which are to be used outside of it.

In the sequence diagram ”Display Specific Member”, I wouldn't have put the Console.WriteLine in there. I don't think calls to an API like that needs to be shown since it makes the diagram unneccesarily complicated.

Overall great diagrams, very difficult to find something to complain about!

Architecture
Great Model-View separation. Tried to find business rules in view and vice versa but didn't find anything.

Models can be reused for other interfaces (WPF, asp.net..) since they aren't connected to System.Console.

Regarding encapsulation, the MemberList is used in a state that can be manipulated in the Console. If you return a read-only wrapper like IEnumerable<Member> from GetMembers(), the view can only read info from the member list, and it's impossible to accidentally break the read-only relation from view to model.

The persistance is tied to the MemberList class. Take a look at chapter 25.3 in the Larman book, where he uses an example of the GRASP concept ”Indirection” to seperate the persistant storage (file storage in your case) and the domain concept (member list). This decreases coupling (another GRASP concept, chapter 17.12). This gives the advantage that you can switch your storage to a database or XML file without changing the MemberList class.

Continuing with GRASP, Information Expert and Creator (Larman chapter 17.11 and 17.10) are handled well. When you add members, you create the objects in the MemberList class. That class contains, records and closely uses Member objects, which are three out of four reasons for it to create members according to Larman.

For cohesion, Larman states in chapter 17.14 that ”A class with low cohesion does many unrelated things or does too much work”. For your controller, you've chosen a controller that represents an actor, and the ”User” actor does quite a lot of things in this system, leading to a rather large controller (~350 lines) that might be thougt of as having low cohesion since it's doing a lot of things. Maybe split it up into smaller controllers for the use cases? Look at chapter 17.13, where Larman points out that a controller can represent a use case scenario.

The model classes on the other hand are very cohesive. It's good that there is a separate BoatList class. It increases the cohesion of Member class, since the Member class doesn't have to deal with the boats and is therefore more focused and dedicated.

I tried scanning your code for static variables/operations or hidden dependencies but didn't find anything. All dependencies are documented in the class diagram, and nothing is static. Couldn't find any painted types either, and you've used associations in member variables well.

Code standard
In the Console class, there is some duplication between DisplayCompactMemberList() and DisplayVerboseMemberList(). There's ~20 lines of code that do basically the same thing, with a boolean and print message being the only difference. Maybe a point for refactoring? The menu choices also appear in a lot of places. ”q to go back”, ”e to edit”, etc.

Naming is good, the methods have self-documenting names. Eg. AddMember(), has an obvious purpose. Great!

Conclusion
To sum up, here's what I percieved as strong points:
- Naming
- Documentation
- Model/view seperation
- Diagram/code correlation
- UML notation
- Cohesive model classes
- Documented dependencies
- No static stuff

Suggested improvements:
- Show User → Member dependeny in class diagram
- No Console.WriteLine() calls in sequence diagram
- Encapsulating the MemberList to prevent breaking model->view readonly
- Indirection between MemberList and persistance. DAL-class?
- Split up the controller in use cases
- Minor refactoring in Console class

References

1. Larman C., Applying UML and Patterns 3rd Ed, 2005, ISBN: 0131489062