Peer review for lm222ix by kk222hk

Runnable files
I ran the JAR file and it compiles and runs with no problems. Did all the use cases and it didn't crash. Nothing to complain about.

Diagrams and code correlation
It's good that the class diagram is showing visibility with ”-” and ”+” before the operations and attributes. That makes it much easier to see what is intended to be used outside of the class, and what is not.

There seems to be some undocumented dependencies and associations. You have a controller member variable in your view, called ”clubRegistry”. That is an association between the UI class and the ClubRegistry class, but it's not documented in the class diagram. There are also dependencies between the UI and the Member class, since the UI creates and edits members, which is a dependency according to Larman in chapter 16.11. (”sending a message to a supplier”).

There is a dependency documented from Member to Boat, but I can't find it in the code? The Member class seems complemetely free from dependencies by looking at the code.

In the sequence diagrams, what does ”User” represent? A physical user? I don't find a User class. That should probably not be in the sequence diagrams, since the entities in sequence diagrams are usually instances of classes (Larman chapter 15.4). Same goes for Database, I'm not finding a class with that name.

Architecture
When using the class UserRegistry as both a data access class and a controller, the GRASP principle of high cohesion (Larman, chapter 17.14) is violated. Larman describes cohesion as ”a measure of how strongly related and focused the responsibilities of an element are”. The UserRegistry has two pretty heavy responsibilites, to control the models and views, and to deal with the database.

Separating these responsibilites into a DAL class and a dedicated controller would achieve higher cohesion using the indirection principle in GRASP (chapter 25.3 in Larman). It would mean that you could change the storage solution (eg. switch database or change to XML files) without changing anything in the controller. It would also make it easier to reuse the database handling class, and to debug it in case something goes wrong.

As you noted in the code comments, the view is dependent on both the model and the controller. As Larman states in chapter 13.7: ”Do not put application logic (such as a tax calculation) in the UI object methods”. It seems like most of your application logic (editing members, adding boats etc.) is happening in the view class. This makes it highly coupled to your controller and models, which means that it's not possible to change the view to another view with a different user interface. This removes the point of using MVC, and increases coupling between classes which Larman states is bad in chapter 17.12, ”A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable”

The models are well encapsulated though, and does not contain any view logic. The only case might be the ToString() methods.

A member is connected to his/her boats using id's. When you add a boat, it is saved directly in the database with a foreign key to the member. A better solution would be to have a list of boats in the member class, or a separate BoatList class. This way, the entities would be connected with associations instead, which is closer to the domain model. That lowers the semantic gap between the domain model and the implementation and makes it much easier to extend, debug and understand the system since you won't need to mess with the database if you want to change how the member listing works.

Code standard
Naming is good. The names of operations are self-documenting since they describe what they do, e.g. ”DeleteMember()”, ”AddBoats()”.

There are some duplication that can be refactored. In the controller, createMemberTable() and createBoatTable() share a lot of identical code. Maybe refactor them into a createTable() method and pass the SQL query as a parameter?

Conclusion
This is the strong points in my opinion:
- Good UML notation in class diagram (visibility for instance)
- Good, self-explanatory code standard
- Database handling seems good

Suggested points of improvements:
- MVC structure. What part of the M, V and C are allowed to to have dependencies between eachother? What is the purpose of a controller and a view? Where should the application logic be?
- Separating the controller from the database access
- Only using real, existing classes in sequence diagrams
- Show hidden depencencies in the class diagram
- Associations instead of foreign keys/id's
- Some refactoring to remove duplication

As a developer would the diagrams help you and why/why not?
Yes, the diagrams made it easier for me to grasp how the system works. The marked visibility made it easier to see what is private and what is public.

Do you think it passes grade 2?
It's object oriented and passes the requirements, so yes. The only problem for passing is the model-view separation, I don't know how important that is for the grade.

References

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