Peer Review of nb222gp

Review written by ro222cm (ro222cm@student.lnu.se)

Running the Program

Even though I'm using Linux, the supplied .exe file was easy to run using the Wine Windows emulator. As far as I can tell from running the program multiple times and reading the sources, it meets the requirements for grade 2.

The menus are well designed and user friendly and the program works with correct input.

Architecture and Design

The information is encapsulated with private member variables available to other classes through get and set methods.

The design is clearly inspired by the domain model and I recognize the important concepts of the domain by looking at the class diagram. The class diagram is well structured, easy to understand and uses correct UML notation. The same goes for the sequence diagrams. However, the workshop requirement states that the sequence diagrams should cover one input requirement (create member, register boat or change information) and one output requirement (list members or look at member info). I'm not sure that ”general control flow” is an output requirement, but I think it's a good diagram nontheless.

I'm not familiar with C# so it's possible that I misunderstand something, but it looks like all of the classes in the implementation are collected in one namespace (BoatClubRegistry). If the classes in the implementation had been assigned to different packages/namespaces it would have been easier to identify the different layers in the architecture (”view”, ”model”). According to Larman [1, p. 204] a layer is a grouping of classes and packages that has a strong relationship with each other. The layers could then be illustrated in the class diagram by using UML package notation [1, p. 201].

Concerning the GRASP patterns I have noticed the following things:

- The implementation follows the *creator* pattern described in Larman [1, p. 282] which states that if class B ”contains” or compositely aggregates instances of class A, then class B should be responsible for creating instances of class A. The class Member aggregates instances of class Boat in the implementation and the class MemberRegistry aggregates Members. As far as I can tell, this implementation follows the creator pattern throughout. One example is that instances of Boat are created in the Member class like this:

```
public void addBoat(BoatType type, int length)
  {
      _boats.Add(new Boat(type, length));
  }
```

- The implementation also uses the *controller* pattern where the class Controller is the one handling the messages from the UI layer [1, p.287]. The separation between the UI layer and the rest of the application is really good in the implementation. There's nothing in the Controller class that deals with how something is presented to the user. The controller calls methods in the UI layer and the UI layer responds with the values the user has entered. Also, there's no application logic in UI layer. I believe that it would be easy to switch to another type of user interface without making changes to the other layers.
- The implementation follows the *low coupling* pattern which means that responsibilities are assigned to classes so that there's no unnecessary coupling between classes [1, p. 285]. Boat doesn't depend on any of the other classes. Member depends on Boat, and MemberRegistry depends on Member, which is correct according to the creator pattern. Overall this indicates that the implementation follows the *high cohesion* pattern as well, where the classes are focused and support low coupling [1, p. 291].

Overall I think it's a really good design with model-view-controller separation. None of the classes in the model layer depends on anything in the view. I do however think that the classes should be separated into namespaces.

Implementation

I'm not familiar with C# and what would be considered good C# coding standards, but I have compared the source code with the style rules in Kernighan & Pike [2] (which isn't language specific) and found that the implementation meets all of the following criteria:

- *Descriptive names* for classes, methods and variables[2, p. 3].
- *Consistent names* that follows the same convention throughout [2, p. 4].
- *Method names* based on verbs.

According to Kernighan & Pike [2, p. 23] comments should add what is not immediately obvious from the code and the best comments aid the understanding of a program by briefly pointing out important details. There is no point in comments that report self-evident information. Although there's not a single comment in the sources to aid the reader, this implementation has descriptive names and a clear and logical structure which has enabled me to understand the program without any prior knowledge of C#. A few more comments wouldn't hurt though.

Overall Impression

A good overall design with properly designed classes with encapsulated information. Most of the classes could easily be reused in another project.

I'm particularly impressed with the separation of the user interface and the controller in the implementation.

I do however suggest that classes with similair responsibilities should be collected in packages/namespaces and that the class diagram should be updated to reflect this according to the principle of *separation of concerns* [1, s. 204].

Sources

1. Larman C., Applying UML and Patterns 3rd Ed, 2005, ISBN: 0131489062
2. Kernighan B. W. & Pike R., The Practice of Programming, 1999, ISBN: 020161586X