

# Objektorienterad analys och design med UML



## Workshop 2 – Suggested Solution

Senast ändrad 2019-10-17 09:40 av **Tobias Ohlsson**

I describe my suggested solution for workshop 2 grade 2 below.  
The solution will be opened AFTER the final workshop deadline has passed and submissions have been examined.



### Code and Diagrams

This is not the only way to solve this task and it is also a bit over-engineered in the view. I have implemented the registry in C# as a console application. Code, class diagrams and a selection of sequence diagrams can be found in the zip archive below.

For some reason, the task uses phone number instead of personal number. In some cases, some solutions are really engineered to show some UML notation (the view). The sequence diagrams where reverse engineered and then some details were removed to minimize “noise” and increase understandability

The application is not very well tested and probably contain some bugs.

There is an omission in the class diagram: the View::MainView should have a dependency on Model::MemberRegistry

	MemberZ.zip 	127.71 kB	2013-10-21
---	---	-----------	------------

### Discussion

The process of design and implementation was quite simple. I first decided on the architecture (MVC) and created this structure in a simple class diagram and then coded a simple structure to see that it worked. The model is very simple in this case so it was designed up front before any coding. The view, however, was created through a series of refactorings driven by the need to avoid code duplication when I realized I wanted to divide a large view class into smaller parts. When the code was “done” I did a class diagram and saw a couple of areas for improvement that where addressed with new refactorings. This was quite a continuous design-implement-design-implement cycle. I usually don’t work with interaction diagrams that much unless I’m very unsure and want to test drive my classes on paper.

The architecture is based on MVC where each component was put in its own package (model, view, controller) (this is implemented as namespaces and physical directories in C#). The controller only contains one class (Master) as the application is quite simple. The model contains three classes, again the system is simple (more is needed here for grade 4 and 5). To avoid code duplication and show some more advanced UML the view contains a bit more. Here we find an inheritance hierarchy and a template-class (a class that is parametrized by another type, in this case, an Enum). The view also got the responsibility to keep the state in the user interface (what member and boat are selected). The choice was between the controller or the view for this and actually, both the code and the architecture points to the Controller. Master to have this and not the view (controller gets and sets this all the time). However, as I know that in web applications the view is the place to put stuff like this (using for example hidden variables in a form) I selected the view anyway to keep the application web-friendly and be able to reuse the controller as is if changing to a web-app.

One of the first responsibilities faced is “who should handle a member’s information”, the choice is quite easy and if you look in the domain model the class “member” is a good candidate. Another responsibility assignment you face is “who should handle all the members”. In my domain model, I had no concept for this and was forced to make something up here (PureFabrication).

Another important responsibility is “who should handle a member’s boats” again we can look in the domain model (and actually just listen to the question) and realize that it is most natural if the member objects are assigned the responsibility to know what boats it owns. Sometimes it can be hard to let go of relational-database-thinking. Common mistakes are to let the boat know who owns it or to simply work with ids/keys to connect a member to a boat. These keys can be necessary from a persistence point of view but if so this should be encapsulated in a layer in the model (compare to Data Abstraction Layer) that is responsible for conversion between an OO-model and the persistence model (for example relational). In the “business” layer we want to work with a model that is easy to understand and is suitable for the requirements at hand (for example counting or listing a member’s boats).

One a bit more interesting responsibility is the unique member id. I opted to put this in model.MemberRegistry as this responsibility requires knowledge to all the previously generated ids (for example when the registry is loaded) and as MemberRegistry has access to this information it is

the Information Expert. The generation of this Id is also a business requirement i.e. part of the model.

In the cases, a variable only should be permitted a certain number of values (for example a boat type, menu choices) I opted to use enumerators. The code becomes more clear, and we have type safety. We have a clear way of limiting possible errors which is something you should always strive for. That is why global public interfaces of static operations or global variables are no good if it is not absolutely necessary. In this way possibly the personal number should be its own type and not a simple string.

In the view, three classes were created (they happen to mirror the classes of the model), and as there was some code duplication a base class was created (view.View). To get rid of some duplicated code in handling the menus but yet keep the possibility to use enumerations in the menus a template-class was created (view.ConsoleMenu) that can be used via the template-operation view.View.DoMenu. Template-classes are very powerful but also often hard to understand and maintain. If you are curious you can study how it works in the code.

### Discussion of static attributes and operations

Having static operations is in most cases a sign of a **flawed design** and there are few cases where it is needed. Static attributes are useful in the cases you want all the objects of a certain type to share the same attribute, however, this is also unusual and somewhat of an exception. Having public static operations or attributes in your application makes the application harder to maintain and evolve:

- You cannot use polymorphism if you use static operations. In each spot you call the operation you hard code the class name and it becomes hard to add another implementation without a lot of search and replace.
- The whole application has access to public static parts. It becomes harder for a developer to understand the design and know how he should solve his tasks compared to working using an argument or an attribute.

The [Singleton Designpattern](#) address the first of these problems, but the global access remains. Singleton should therefore only be used when you must guarantee that one and only one instance (object) are to be created from a certain type (class) and this instance should be shared (and as you probably see this is very unusual). The global access in singleton should be avoided, send arguments so that dependencies are clearly shown. [Why Singletons are Controversial](#).

### Discussion of Encapsulation

A common mistake is to add a private variable in the “Registry” containing all the members and then simply return a reference to this list in some “getMembers” operation. In essence, this breaks the encapsulation, and any client class can add, remove and sort members and this will affect the Registry. For example, adding a member with an invalid member id would be easy, and the business rule of unique member ids in the registry is thus invalidated. This also means that exposing the member’s list of boats, in the same way, is not an issue from the business logic aspect, i.e. we have no rules that will make the Member object inconsistent if we add or remove boats directly from the list. However, another issue is that there are multiple ways to do the same thing i.e. adding a member using Registry.addMember(...) or Registry.getMembers().add(new Member(...)), etc. etc. Multiple ways of doing the same thing add confusion and should be avoided. This ties into the Interface Segregation Principle of exposing just the things that clients actually should be able to do and not more. I.e. if we want to be able to iterate the members in the registry then that is what we should expose.

### Discussion of connecting objects with keys (ids)

If you are using a relational database, rows in one table are connected to rows in another table using integer keys (primary key, foreign key). Using keys like this throughout your application is not a good idea for several reasons: they are not type safe, you will not get any good error messages, they are not naturally documented like a real class (class name, operation names, argument names). As a developer, you simply do not know what to do when you are working with an id. Compare the following:

```
DoStuffOnMemberBoats(Member a_member) {  
    // I know I get a Member object and can check that class on how to get the boats of the member and how to call Stuff on the boats.  
}  
  
DoStuffOnMemberBoats(a_memberId) {  
    // I have no idea where to begin. I need to find out what a_memberId is by checking everyone who calls the operation and figure out what they send me. Then I probably need to understand the database design and SQL, then I maybe can start doing what I’m supposed to do.  
}
```

Keys also hide dependencies that prompts erroneous decisions in testing, reuse, etc. etc. Keys are a way of connecting rows in tables in relational databases. And this should be encapsulated and not spread out in the whole application, for example in a data abstraction layer (DAL) in the model.

To not “leak” dependencies (to an SQL db for example) throughout the application is one of the advantages with a good object-oriented design (encapsulation). Developers of other parts then do not need to know how/when/where data is stored and the application in turns becomes more secure (anyone/anything cannot talk to the database) and more flexible (we can change the way data is stored without breaking the whole application). If we work in iterations/an agile way only parts of the database are designed and the database will evolve (database refactoring), yet another reason to encapsulate data storage. Having dependencies on id’s, tables and SQL spread all over the applications becomes a nightmare and the smallest change becomes very costly.

## Discussion of Model View Separation Principle Problems

Problems with separation of model and view come in three forms. The first form is a dependency from something in the model to something in the view. This can, for example, be a class or function reference, a well-made class diagram will show this as some dependency from a model class to a view class. However, such dependencies can be hidden, i.e. in a form that is not easily recognized, typically such dependencies rely on information hidden in primitive datatypes (for example integer ids or strings). As such use of primitive datatypes that fill no apparent function should always rise some form of suspicion.

The second form is to have view responsibility in the model. The responsibility of the view is in essence to output stuff to the user and collect the user’s input, commonly output on a screen and input via keyboard and mouse. Having view responsibility in the model then means that the model has taken on this responsibility. You can spot such problems by imagining a scenario where you want to change an aspect of a view (for example changing the language, or the layout in a table etc). If the model then needs to change there is a fair chance that it has taken on a view responsibility. In some cases, there can be direct calls to API functions that only the view should have (e.g. calling a console print function), such dependencies can, of course, be checked for. In other cases the responsibility is more subtle, i.e. the model returns something that is directly formatted to a specific type of view, often in the form of a primitive datatype, e.g. returning a formatted string representing some model object that is then used for output. This type of problem is treacherous as it seems convenient, fast and you can often encapsulate some model classes and then get seemingly fewer dependencies (albeit they are hidden). However, the price for this will be paid when the interface needs changes, and changes in user interfaces is a common situation. With this kind of problems you get lower changeability, i.e. you cannot evolve your user interface as fast as you would like.

The third form is to have model responsibility in the view. The view has taken on some functionality that should really be part of the model, i.e. a business rule is implemented in the view. You can spot such problems by imagining reuse of the model in a different system if some important function is then missing you have a problem with business logic in the view. Another way is to imagine another user interface to perform the feature if a lot of code needs to be duplicated there is also a fair chance that such code should be in the model. From a dependency point of view there could be some un-needed dependencies from view to model, however, there is a fair chance that the number of unique dependencies is the same and only the number of calls (etc.) can be lower. In essence, the model presents a too low level of abstraction to the view. This is probably the trickiest problem as it requires intimate knowledge of the business domain to make the judgment of what actually is a model responsibility. With these types of problems you get lower reusability, i.e. you need to rewrite code that you should have been able to reuse. Another issue in this form of the problem is that you may think (read assume) that the requirement is actually in the model and then create a new view, forget that this needs to be handled in the view, and then a bug will be present. Common issues are authentication and validation code in the view.

## Discussion of Hidden Dependencies

Hidden or implicit dependencies are relations that do exist but are not easily visible using standard techniques. Our common forms of class to class relation are to inherit some class, realize an interface, have an attribute of a class, or simply use another class in an operation. From these, we get the UML relations; generalization/specialization, realization, association, and dependency respectively. Such relations are often easy to spot and using a compiler we can simply change the name of the thing that we have a relation to and the compiler will issue an error. For example, changing the name of Boat to Vessel will issue compiler errors in the Member class as it has relations to the Boat type. In this case, relations are explicit or visible. So a hidden dependency is then a relation that will **not** trigger such errors, instead these will often manifest in the form of bugs. I.e. a working software that has some unwanted behavior. The scenario for finding such problems is to imagine that we change class B and if we need to manually remember that class A then also needs to change we have a problem with a hidden dependency. In some cases we can use language constructs or design so that we can make the dependency explicit, in other cases we will need to rely on testing (preferably automatic) to make sure that such changes are remembered. Common signs of hidden dependencies are hard-coded constants that need to match in several different places, the order of information in arrays/lists that need to be the same in several different places, data in files that need to be in a certain format/order in several different places. The root cause of hidden dependencies is often poor encapsulation and the responsibility has spread out in several different classes.