Object Design Document - Yellow Project

William Diedrichsen Marstrand, Dennis Thinh Tan Nguyen,
Jacob Mullit Møiniche, Thor Valentin Olesen

16. december 2015

Indhold

1 Introduction

This object design document has been written to provide the reader with an insight to our design choices throughout the project. To support this, class diagrams will be used to highlight where design patterns have been used. Finally, this is related back to the SOLID principles. The class diagrams are used to showcase our familarity with the techniques elaborated in OOSE throughout the course. Note that this document only focuses on important classes and provide a complete detailed description only of those. The purpose of the UML techniques is to better convey our design choices and final code structure.

1.1 Interface documentation guidelines

To improve communication and understanding between the developers, the group has defined a set of documentation guidelines that must be followed strictly in order to have a consistent codebase structure and design.

Base Guidelines

- Every class are named with singular nouns.
- Every class and methods are to be documented before they are implemented. The documentation must be precise in regarding for what the given class or method is responsible for. This will enable developers to be more effective in their work because they do not have to spend time on analyzing poorly written documentation or the implementation itself.
- Every method must be defined in a way that they only have one responsibility. Long methods should be refactored into helping methods or moved to other appropriate sections of the system.
- Every non-trivial parameters and returns are to be documented.
- Every Method signatures must be written with camel-case and they must follow C# formation standards. This improves readability for other c# developers that are to read the code.
- Error status must be returned as exceptions and propagated to appropriate layers. One should avoid catching general exception since this may mask problems that have nothing to do with the system. Instead of just catching a concrete 'Exception', one must catch specific exceptions that may be relevant to process such as "OutOfMemoryException"that would preferably be propagated to the user.
- One should use generic classes where to seem to be fit. For an example, one should utilize an interface such as IList¡object¿ instead of a concrete implementation such as 'List¡object¿'. This makes it easier to refactor the code since it is not tightly coupled to some concrete class.

- Dependency injection must be considered when creating new classes and methods. System modularity will make it easier to extend or refactor the system.

It should be noted that the group had decided to follow these base guidelines as much as possible. However, it should be considered that exceptions may occur and some rules may be broken. These exceptions must be brought up into descussion or reconsidered before proceeding with the given activiety.

2 Design Decisions

3 Object design choices

3.1 Facade Pattern: MainHandler Facade

Multiple handler classes were made with their own well defined area of responsibility in an attempt to uphold the Single Responsibility principle. This resulted in several handlers and thus a complex system of handlers which we considered a problem, since it made the understanding of the system more complicated than if the functionality was gathered one place. In this case a Facade Pattern seemed useful, since it would make it possible to wrap the complex set of handlers and offer a simple way to access the required handler functionality using a MainHandler Facade as a facilitator. The use of the facade pattern in this case is not strictly as the general guidelines describe, since no interface is provided for the handlers to implement, but the principle of having one facilitating class providing the functionality of the subsystem classes is respected. Another important point was the way a MainHandler facade could decouple the subsystem of handlers from the using classes. Even though the limited functionality provided through the facade would mean some what less flexibility than if all handlers were used, the conclusion was made that based on the just mentioned pros, the use of a facade would be primarily beneficial.

The repository design pattern has been used as a layer in the code to separate the business code in our Application Logic and the database. The business logic accesses data from data stored in the Storage package where the AutoSysDbModel represents a local database in SQL Server 2014. This has been done to isolate the data layer to support unit testing and support swapping out the database technology. The code maintainability and readability is increased by separating business logic from data or service access logic. The entity-related repositories in the Storage pack separate the logic that retrieves the data and maps it to the stored entity model from the business logic by using the AutoMapper framework. The repositories are used to perform basic CRUD operations on entities that are required for permanent storage in the database. The repositories can query the data source layer (database, either local production database in SQL Server 2014 or in-memory database using fake connection in the Effort Framework for integration tests). The repositories do not map the

data from the stored model entities in the Storage package but AutoMapper is used to map the data source to a business logic entity in the StorageAdapter classes in the ApplicationLogic. These adapter classes are likewise used to store persist changes in the business entity to an Entity Framework database. Thus, the repository separates our business logic from the interactions with the underlying database.

4 Repository Pattern

4.1 Benefits

Using the Repository design pattern has centralised our data logic and helped create a flexible architecture that can be adapted as the overall design of the application evolves. Without the repositories we would have to access data sprinkled throughout our codebase. By way of example, updating a table would require to find every spot in our code that relies on it and update all of them. With the repository pattern we only need to update a single class and the rest of the code is unaffected. This corresponds well to the Single Responsibility Principle stating that each thing should have a single reason for change. The repository helps us isolate changes and limit the scope of an entity model to a single level of abstraction. Ultimately, increasing readability and maintainability.

4.2 Cons

The only downside is the fact that our lacking knowledge on the domain objects has forced us to refactor the data representation of all stored model entities numerous times. As opposed to having a clear understanding of the business logic and mapping these objects correctly to the permanent storage from the start. Also, we struggled choosing whether the repositories should be async or not. Consequently, we chose async assuming the program could possibly run with a hosted database. In that case, async maks it possible to use CPU resources that are otherwise not usd while threads are waiting for an answer from database queries. Finally, we did not have a clear opinion on when to dispose the DbContext used to interact with our database. Initially, we disposed the context for each database method call in the repositories. Instead, we now implement IDisposable throughout all interfaces between the WebApi and the Storage layer to dispose after each client request is invoked and completed instead.

4.3 Test Reflection

All the functionalities have been tested by mocking an interface of the AutoSysDbModel instead of using a test stub. We should have started on the integration tests earlier to validate that our repositories and the stored model entities and their relationships are actually compatible with a database. Initially, we tested if the logic in the repositories are called but not if the entities were actually

stored correctly. For this purpose, we used the Effort framework to create an in-memory database on a fake connection and validate the CRUD operations on a real database while checking if the entities were properly interrelated with the foreign key and data attribute functionalities supported by Entity Framework.

4.4 Conclusion

We have performed numerous design choices in the actual implementation of the repository pattern. Mostly, the challenges considered have been whether we should use a generic repository or several concrete entity-related repositories and mapping the entities. We have chosen latter to achieve Separation of concerns and reduce unnecessary unit testing complexity. As a result, we end up with more repository classes but also a significant readability and maintainability improvement and easier unit testing. The repository methods only need to change if the flow of data access changes and it is not impacted by the code in the business logic. Thus, working with a repository built on SOLID adds a lot more speed to future addition than without proper separation.