## Design Patterns

During the development of the program we were disappointed with the amount of design patterns we had applied to our solution. Early on, in the elaboration phase, we identified the need for a composite design pattern to represent a hierarchy of folders and documents. After that problem was solved, we never found the need to use other known patterns, because we never arrived upon another serious problem regarding software design during development.

We were familiar with the design patterns in advanced and knew that their use does not necessarily increase code quality if there is no need for their implementation. We wanted to avoid over-complicating our design, and kept ourselves from needlessly making use of patterns we did not need. A pattern was only considered to be used if we could predict a potential problem with our current solution in the future.

However, the need for design patterns rarely appeared. We were both worried but mostly relieved about the assumption that our design was simple enough to not warrant the need for other design patterns than the Composite Pattern, but near the final parts of the project we realized that the C# language/.NET framework had been providing solutions already implemented and ready to use, to many of the common software design problems encountered elsewhere.

As a consequence of this, our design workflow in our project, except the core class library, changed to more of a solution-finding process than a problem-solving process.

This is better explained as a development process where we found ourselves more often thinking:

“What solutions does the .NET-framework provide for achieving our goal?”,   
rather than   
“How are we going to design to avoid problems in the future?”

Many decisions on our software design were made from knowing that we had the convenience of the availability of the .NET-framework features. This convenience allowed us to abstract away from many common problematic software design areas that required a lot of careful design, such as graphical interfaces, networking and even more common situations such as collection iterators.

Despite the convenience of the .NET-framework, we still believe that being familiar to common design patterns and design pattern guidelines in general has had a positive influence in the way that we have designed our system, and also in the way that we have used the features of the .NET-framework.

### Design Patterns used

The following is a short description of the design patterns we have implemented ourselves, as well as the design patterns provided by the .NET-framework and C# that we have identified and used:

Self-implemented patterns used

* Composite

To allow for operations that affected a whole hierarchy of folders and documents, we decided to take inspiration from the composite pattern. At the time of implementation, we decided we would not need to be able to do method calls that ran down the hierarchy by itself, so our composite interface does not expose methods that are able to do so.

Provided design patterns used

* Composite

Also, worthy of mention, the Composite pattern is used in the .NET-framework implementation of WinForms and WebForms. All controls provide a diverse amount of different functionality, but they all retain child management features.

* Iterator

The Iterator pattern allows staple functionality for using collections. As collections are a core functionality of the .NET-framework, we make use of iterators very often throughout our project.

* Observer

The WinForms and WebForms make great use of the observer pattern, using events and delegates to make publishers and subscribers. User interfaces are a common area where the observer pattern sees great use.

* Proxy

Using WCF services to make the server/client implementation, the local client trying to contact the service is provided with a proxy object where it can perform method calls as if the local object was the remote object on the server itself, without the program being much aware of what object is actually residing behind their common interface. This is a good example of using the Proxy pattern to abstract away from the worries of what to do with accessing an object across a network (in this case, a remote procedure call).

(We have only included the pre-provided patterns that we feel we have used substantially. This list does not cover all of the common design patterns implemented into the C# language and .NET-Framework.)