Miserable software architectures

Jess Nielsen

Trapeze Group Europe A/S, Søren Frichs Vej 38K, DK-8230 Åbyhøj, Denmark

Every software architect and most developers know the importance of separation of concerns by encapsulation. A layered architecture is often used when satisfying such design goals (among others). All software projects that have a dedicated software architect often have a well-defined architecture when the development of the product starts, but at the end various changes that are contradictions to the architecture might exist for several reasons.

TRADITIONAL LAYERED ARCHITECTURE

When software architecture for a given product has been defined we often end up with a beautiful layered model, which can vary a bit depending on the age of the software product.

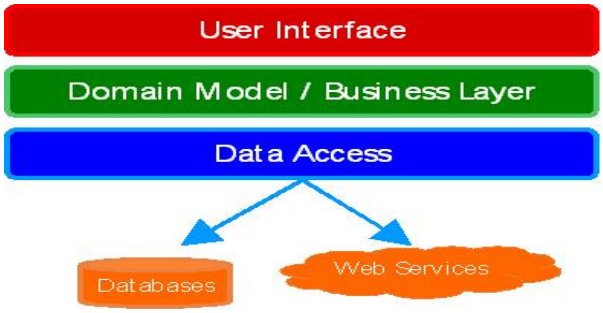

Figure 1: Layered architecture

Elderly layered models have often both the domain model and the business logic located in the same layer. That means that it ends up with very large domain objects that both are containing domain data and business logic. In the illustration above (figure 1) the domain model / business layer represent that concrete layer.

Dealing with large domain objects makes it more difficult to transport domain data between layers and the presentation of the domain data in the user interface will also be very difficult.

The user interface contains graphical objects such as combo boxes, list boxes, tables and edit-fields etc. These objects often have to contain an enormous amount of data. When using large domain objects it will turn the graphical user interface into an elephant that it will be working very slowly.

DOMAIN-DRIVEN ARCHITECTURE

To solve this problem among others the software architecture has turned into a so-called domain-driven architecture [4], which is also appropriated when the domain is not trivial. The primary goal of this architecture is to separate *domain data* and *business logic*. In this way the domain objects contains only the domain data and all business logic will be placed in a dedicated business layer instead.

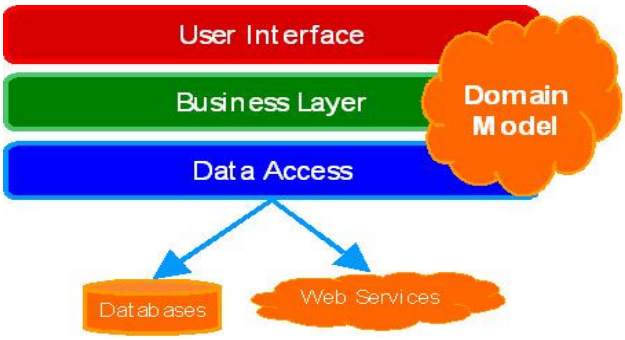

Figure 2: Domain-driven architecture

This gives us some very lightweight domain objects, which easily can be transferred between the layers and it will have a higher coherence with the graphical objects in the user interface layer as well.

We have now defined the software architecture for the product. When the detailed design has been accomplished we are ready to start the development.

AN ECONOMICAL PERSPECTIVE

During the development, which often are the most time-expensive phase a lot of other factors comes in. Even that the software architect has tried to satisfy them all it will not be possible to satisfy them all completely and it will often be necessary to make some tradeoffs.

All software products always depend on the economical aspects.

The product has to be completed to a certain date and only a limited number of working hours are allowed for the development of the product. Few working hours spent is equal to low costs that give often-higher benefit.

To achieve this, a number of work-a-rounds or architectural shortcuts often exist and some of them are in direct contradiction with the software architecture strategies and should not be used at all.

Three general architectural shortcuts, that all should be avoided, is listed below. These shortcuts are dealing with compromises on functional requirements, non-functional requirements or even both.

1. Reduce the functionality is one way of reducing the working hours, but this will of course end up with a product that does not match the functional requirements for the product. It will be very difficult to do this in practice because of the legal binds.

2. Cutting down the architecture using a virtual chainsaw i.e. by reducing the number of layers is another way of doing this. It should be obvious that this is of course in contradiction with the software architecture strategies.

3. Remaining the software architecture as is and instead uses small hacks where possible to cut down the use of man hours is another way. All kind of hacks is in contradiction with software architecture strategies and it becomes very difficult to maintain the product afterwards.

Generally all kind of architectural shortcuts should be avoided when doing software development, because they will always be a contradiction to the defined software architecture.

It will often end up with a software architecture, where the layers no longer are well-defined and purely encapsulated.

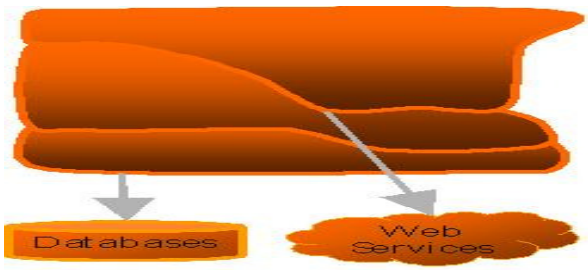

Figure 3: Miserable architecture

One of the major disadvantages is that it makes it difficult to change the implementation of a layer due to changed services provided by the lower layers and even more difficult to maintain afterwards.

"If you think good architecture is expensive, try bad architecture" - Brian Foote and Joseph Yoder

Today, hacks and quick patches are unfortunately often used as shortcuts when doing software development as concluded in [2]. This has been confirmed by the research that clarifies the premises of the design decisions as described in [1].

Finally I can strongly recommend documenting each major design decisions that have impact at either the quality attributes or the functional requirements. The primary purpose of this is to document any tradeoffs that have an impact on the architecture by affecting the general quality attributes and the functional requirements.

In this way, it is possible to clarify the architectural impact of any architectural shortcuts that are violating the principles depicted by these major design decisions. The design decisions can be described by using Meta models such as REMAP[1] and DRL[2] as outlined in [3].

Jess Nielsen is a senior developer at Trapeze Group Europe A/S. His research interests include autonomic systems, software architectures and cryptographic. He received his master's degree in IT (Software Development) from Aarhus University, Denmark.
Contact him at jessn@bluebottle.com.

REFERENCES

[1] Architectural documentation and evaluation of a system to transfer invoices
Nielsen, J.
University of Aarhus

[2] Defence of bachelor degree
Nielsen, J.
University of Aarhus
http://jess.heidrun.dk/cv/bachdef.pdf

[3] Architecture decisions: demystifying architecture
Tyree, J.; Akerman, A.
IEEE Software Magazine. Mar/Apr 2005

[4] Wikipedia article: Domain-driven design
http://en.wikipedia.org/wiki/Domain-driven_design
January, 2nd 2012

[1] REMAP is an abbreviation for "Representation and maintenance of process knowledge".
[2] DRL is an abbreviation for "Decision Representation Language".