# Chunk Cloud Computing (CCC)

Or

*A summary of the last few years*

Or

*Painful self-reflections from the past*

by Jimmy Nilsson
http://jimmynilsson.com/blog/
http://twitter.com/JimmyNilsson
2009-03-29

*In this article, Jimmy Nilsson describes an architectural style that he has observed slowly growing in popularity over the last few years. He summarizes the style like this:*

*Before: Focus on layering*
*Now: More focus on partitioning*

*Before: Focus on loose coupling*
*Now: A balance (include high cohesion)*

*Before: Huge teams*
*Now: Extremely small teams*

*Before: Integration database*
*Now: Application databases*

Over the last few years, several different trends have been pointing in the same direction for me, and it's high time to sum that up. There's nothing new here, it's just a summary of my thoughts and observations. I felt I just had to paint the picture (even though the background is painfully full of my mistakes). Here goes.

## Little focus on partitioning for years

Even after 16 years I still remember the discussion in [Booch OOAD] about the usage of layering and partitioning (some readers may be happier with the word Module as a synonym to Partition). I created and taught a course on object-oriented analysis and design back then and used that book as the course literature. It was easy to discuss layering since I was exposed to it a lot back then (and later, as I will touch on in the next section), but it was a bit harder to discuss partitioning.

We did use partitioning in the real world projects I was part of before and at that time, but it was hard to demonstrate in small examples, and as I recall it was done because of technical limitations and didn't happen as easily and automatically as layering.

This feeling of under-utilizing partitioning just stayed with me, without me doing much about it.

## Overuse of layering

In my first book [Nilsson NED] from 2001, my fondness for default layering had reached its peak. In Figure 1 you find a simplified version.

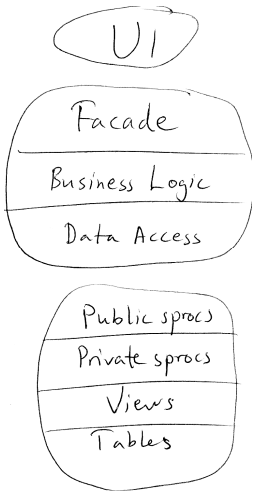


*Figure 1. Typical layering from before (simplified version)*

As you saw in Figure 1, in the mid tier I had some kind of façade layer, some kind of business logic layer and a data access layer. At the data tier I had a public sproc layer, a private sproc layer and sometimes views on top of the tables as well… (the UI was layered too, but you get the point…).

I called this layering scheme my "default architecture", which meant that I commenced new projects with this as a starting point, so to say. Of course its appearance shifted over time, but my point is that I started many new projects with an idea that a rigorous layering scheme was needed.

Some observations:

- Most often, when I had written the database schema, the views, the private sprocs, the public sprocs and the data access layer by hand (to be sure that it was exactly the way I wanted it to be regarding, say, performance), there just wasn't a lot of time to spend on the business layer. Therefore, the business layer was often extremely thin, little more than a pass through layer with some placeholder comments…

- As you saw in Figure 1, there was a lot of protection of the data storage from the UI, very many protective layers. But more often than not, the database schema was actually exposed quite, if not very, openly with the Recordset pattern that was used for transferring the data from the database all the way out to the UI. And

when the UI was done with its changes, the data structure was sent back again, pretty much directly into the database at times… So much for the protection.

- The basis for each and every layer was mostly an idea of "can be good to have" instead of "proved to be needed". A good example of YAGNI violation, eh?

Nowadays, my typical starting layering scheme is very different. It starts out as only a domain model layer that captures the interesting part of the solution, the solution to the business problem.

After that I add layers as necessary, such as a user interface if it's needed. I can actually add that directly on top of the domain model if it makes sense. If not, I add what is needed.

Furthermore, if we want our domain model to be persisted in a relational database, then we add that to the picture. Quite often it works out fairly well with an Object/Relational Mapper as an automatic solution for maybe 80% of the data mapping needs between the domain model and the database. See Figure 2.

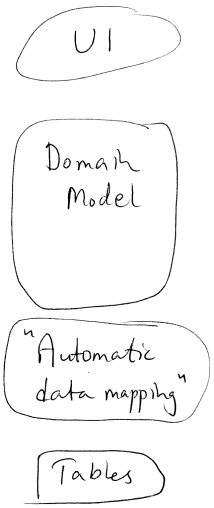


*Figure 2. One example of typical layering nowadays*

I also try to let the database schema be a "result" of how the domain model looks. The database schema will automatically be generated. That means that loads of tedious work can be avoided which will free up time that can be used on the interesting part, the domain model itself. (The story might be the same for the UI.)

And all of a sudden, I don't have such a rigorous layering scheme any longer. Instead, I'm focusing on coming up with suitable partitions. And since the whole solution is split into several smaller partitions, each partition's layering can actually be relaxed simply because of being smaller in size.

## Technical team split

Something that occurred to me when writing the previous section was the discussion we had at the Software Architecture Workshop in Lillehammer, Norway, in 2005 [Fowler LayeringPrinciples] about how to split a big team into several smaller ones.

Both before and after that event, I had split teams after both technical and functional aspects several times. I remembered especially one project where the functional split was a failure when it came to one participant. That developer didn't have much prior experience and struggled with more or less all of the different aspects...

In another project with other developers, a technical split seemed to be very successful, even though the developers had a great deal of experience. But even in that project, there were quite a lot of situations where a functional split would have been more productive, for example, having to involve fewer developers to finish a quick little change that spanned both parts, so to say.

To conclude all this, I prefer to have a functional split. Then of course people will have expertise in certain parts, but for most parts they can run as fast as possible without having to sync and wait for other developers to do their stuff on the same story.

## Enterprise Domain Model

A couple of years ago it was quite popular to try to capture *the* data model of the whole enterprise. The idea was that if the data model could be caught and described once, lots of business value could be crafted out of that complete data model. So the business people received a message along these lines:
*"Leave us alone for two years now and then we'll get back to you with a defined model and everything you need and want will be extremely productive to create."*

As I understand it, "Enterprise Data Model" failed big time. I'm sure there were many reasons for that, but perhaps the following were among the most important:

- Such a data model became enormous, even for a moderate sized company.

- The data model tried to describe a moving target statically. It's also the case that a task like that becomes much harder when the moving target is enormous rather than if it's small.

- The big model was more or less generic and therefore context free.

I'm personally a strong believer in taking on big tasks piecemeal, and that context is king.

As I said, I think attempts to create the Enterprise Data Model were often considered failures. What's slightly ironic is that I find projects that almost try to repeat those attempts, but this time with an Enterprise *Domain* Model. The arguments are similar, and I think the result and the reasons for the result are similar too…

In fairness, it's far more common to see situations when it's not really an Enterprise Domain Model that has been tried, but rather a single large monolithic Domain Model. Furthermore, teams often see the need for splitting a big Domain Model, but find it hard to find a satisfactory solution.

Anyway, I strongly recommend that you should partition the domain model when it starts getting large. And don't forget that a model has context!

Interestingly enough, I've heard about several very big SOA projects which seem to try to create something like Enterprise *Document* Model as the first action in the project. Will it work out better than the Enterprise Data/Domain Model? I put my money on it not being the best approach in this case either.

## Integration database

Not everybody creates huge monolithic domain models, but they can achieve good sized, isolated pieces instead. But quite often in those cases, the database isn't partitioned. Instead, the different domain models are put on top of a single database, an integration database, see Figure 3.

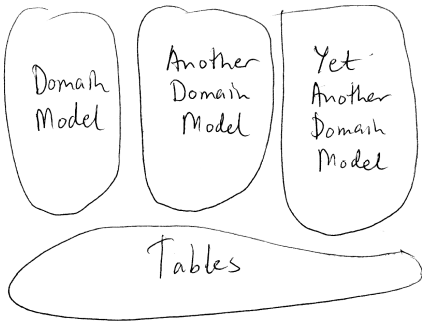


*Figure 3. Partitioned domain models, using an integration database*

Having several domain models on top of a single database is of course only one situation of integration database. It's just the same when you have several isolated applications sharing the same database. Let's just conclude that it's an extremely common situation.

Unfortunately, one of the drawbacks this creates is a great burden on maintenance. All changes that affect the database schema have to be synched between all consumers of that database schema. The more consumers there are, the worse. The result is that the number of changes is most often kept to a minimum, which in its turn probably means that far less business value is squeezed out of the involved code bases than is possible.

I would go as far as saying that nowadays I see integration database as an anti pattern. OK, I said it. Ah, it feels good.

So, what's the alternative? I think that the partitioning you decided on for the domain model should follow along into the database as well so that the database is partitioned in

the same way. This is called Application database [Fowler ApplicationDatabase] instead of Integration database [Fowler IntegrationDatabase], see Figure 4.

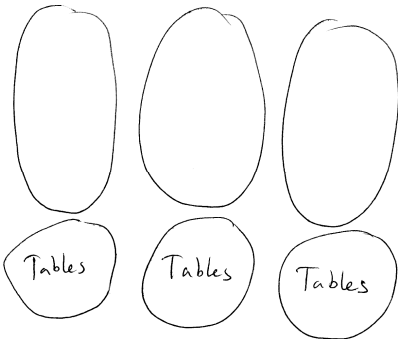


*Figure 4. Partitioned domain models, using application databases*

The way to communicate between the partitions is to use services on top of the domain models. Shortcuts to the application databases of the other domain models are not allowed. This implies that it should be possible (and typical) to effectively deploy different application databases on separate servers (or in the cloud).

A commonly perceived advantage for integration databases is that of reporting. Another way of dealing with reporting is to see it as another application, with its own storage. The data is collected from the other partitions. One approach is to see each partition as an event sourced application [Fowler Events], and just tap those even sources of interesting events for the reporting application.

By the way, it occurred to me that a piece [Nilsson Bricks] that I initially wrote for ADDDP [Nilsson ADDDP], (but that I decided to rip out of the book) described another take on application databases. However, in this case the focus I had when I wrote the description was on performance, not maintainability. In the text I suggested how you could use the knowledge of the different characteristics of the different parts of the system to your advantage.

By the way again, be sure to read Greg Young's very interesting writings about DDDD [Greg Young] that can be considered as being in this category too. He suggests a strict partitioning between reads and writes, and thereby achieves, for example, very good scalability.

## Integration UI

The argumentation is actually similar, although maybe not as strong, when it comes to the UI as with the integration database vs application database. Instead of creating one UI on top of several services, why not let each service have its own UI and then integrate the different UIs with a UI container framework instead? See Figure 5.

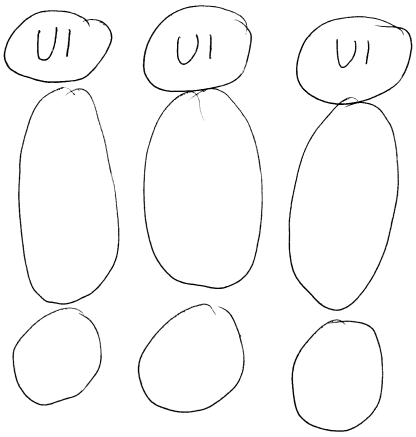

*Figure 5. Partitioned UIs too*

All of a sudden, a developer in a certain partition can take care of the whole story, from the UI all the way to the storage, without having to sync with other teams. The possibility for real productivity is huge.

## Huge teams

Quite often, people come to us and complain about that they can't get their big team of, for example, 100 developers to be as productive as they want them to be.

Each and every time, I come to the same conclusion. Don't! 100 developers in a single team, that's just scary and the risk of failure is enormous. Even if you add 20 more! ;-)

Almost 30 years after it was written, it seems as if surprisingly few people have read The Mythical Man-Month [Brooks MMM]. That's sad. (The idea of the title, The Mythical Man-Month, is that adding more developers to a project running behind schedule will make it even later. The book is full of other important ideas too.)

So, assume for argument's sake that you do have a problem of enormous size and you actually do need 100 developers. Be extremely wary of splitting the big team into several smaller, as isolated as possible, teams, so that each can run at full speed.

Obviously, however, watch out that complexity isn't just moved somewhere else when you split the big team into, for example, 100 new teams… Just because it's probably better to have 10-20 teams than 1 in this example, it's probably not the case that it's even better to have 100. Keep balance in mind.

## SOA and loose coupling

Talking of balance, a couple of years ago I had a really hard time understanding some of the loudest SOA-messages around and I wrote a couple of blog posts [Nilsson SOA-Qs] asking for explanations.

I just didn't get the message about extreme fine granularity, for example. Why not think of services as you think of Bounded Contexts [Evans DDD] instead of having services that are sooo small that they are just a single line of code…

Jim Webber discussed it in an article [Webber Anemic Service Model] and I think he has a very good point there, that the focus on loose coupling had become the only thing that mattered, without pairing it with its typical friend high cohesion. That's probably what created such a strange message, one that I just didn't get.

So, I think of the partitions I've discussed above as "SOA done right".
:-)

## Enforcing one style

When first writing that header, I initially wrote "Enforcing one *religion*", but then perhaps Google would direct a lot of people here who would be very disappointed in the content of the article. What I'm talking about here with style/religion is that there are quite a few of us who think, like and talk about TDD, DDD, BDD, patterns, refactoring, clean code and so on. But there are far more developers out there who don't agree at all that that is *the* way.

One solution that is sometimes used is to force all developers to use one and the same style, no matter what. What happens then? I guess the following is quite common:

- Some developers are very unproductive.
- The same developers are very unhappy.
- The whole team gathers around a style that is the lowest common denominator.
- And therefore the developers who like the style that was enforced in the first place also become unproductive and unhappy.

I definitely do see merits in consistency, but not at all costs. I actually think it can often be beneficial to let different partitions be developed with different styles. Differences in skill sets between different developers is one reason. Another, and often overlooked reason, is that different partitions will, of course, be different and therefore different styles are more or less beneficial. For example, some partitions will most certainly not benefit much from a domain model approach, and then it's OK not to go that route. It's actually recommended not to, of course!
:-)

Saying that, even if you are very relaxed regarding the inside of the partitions, you can (and probably should) set strict requirements on the outside view, as far as automatic tests go, for example.

I do realize that this violates the principle of collective code ownership, since people will stay within their partitions. *Within* each partition, there will absolutely be collective code ownership (if that's part of the style that is used in that particular partition).

Oh, and within a single partition, I do think you should enforce one style.

## Different change speed

It might be obvious to you, but I'd like to point out that the change speed for different partitions will probably differ greatly. I usually joke about a sentence commonly heard early on in the project room of database driven projects:
*"We have to set the database schema so we can get started working!"*
I've said it myself so I'm not trying to poke fun at someone else.

Even though I'm a bit hesitant to think so, perhaps some partitions can work out well just like that, setting the database schema and then not change it later on.

At the same time, other partitions will use the typical style of DDD projects:
*"Well, we don't know very much now, but let's start with our current understanding and change the domain model every day when we learn more, which we will!"*

Now both styles can co-exist happily. As a matter of fact, they could have before too I guess, but now we have made the boundaries explicit and the chance of success has increased.

## What about the "name"?

I think someone said that everything worth discussing must be given a name, and I wouldn't want to disqualify this architectural style from discussions just because it has no name.

Is "Chunk Cloud Computing" a good name? That's up to debate of course, as always. For now, I'd just like to tell you the origins of the name.

This year, at the annual European Software Architecture Workshop, Christoffer Skjoldborg proposed a topic with the name of "Chunk Cloud". "Chunk" as in a small piece (partition), "cloud" after the current hype. Then he described the topic in quite a similar way to how I see one standard approach of architecting applications nowadays. (That is what I have tried to explain above. I was particularly inspired by Chris in the section "Enforcing one style".) Chris was maybe a bit more extreme in his description, but I think it was largely to get his points across.

It was Nicklas Andersson (http://nickeandersson.blogs.com), at the same workshop, who added the suffix of "Computing", probably because CCC looks cool.
:-)

## A best practice?

I don't know about you, but when people say "best practice", an alarm bell goes off. It's probably because of all the discussions about the Dreyfus model [Dreyfus model of skill acquisition] and the trouble of having context-ignorant "best practices" and so on…

I don't see Chunk Cloud Computing as a best practice. I just see it as an architecture style that I quite often find suitable. But, as always, it all depends.

## New problems?

Absolutely! But don't you find old problems boring anyway?
:-)
I do know about lots of new problems in this style, and there are probably some that will crop up over time too (as always). I find that the ones most people first bring to the table are the one of having the same data at several places and the risk of data inconsistency over several chunks. Those problems have to be dealt with, absolutely, but I'll save this for another article.

Of course one problem that arises the first few times is "how to", not least how to partition the domain model into several small chunks. Let's see when I find some time to write about that too.

## New possibilities not mentioned above?

Absolutely again! But again, I'd like to come back to those in yet another article. I will probably write that one before the one about the new problems. I'm just more into happy stories.
:-)
Stay tuned!

## More observations pointing in this direction? Away from it?

I guess it's safe to say that there are observations (and opinions) in both directions, and I'm looking forward to reading about your thoughts!

## References

[Booch OOAD] Booch, Grady; Object-Oriented Analysis and Design with Applications; 1994

[Brooks MMM] Brooks, Fredrerick P; The Mythical Man-Month;

[Dreyfus model of skill acquisition]
http://en.wikipedia.org/wiki/Dreyfus_model_of_skill_acquisition

[Evans DDD] Evans, Eric; Domain-Driven Design; 2003

[Fowler ApplicationDatabase] Fowler, Martin;
http://martinfowler.com/bliki/ApplicationDatabase.html

[Fowler Events] Fowler, Martin; ]http://martinfowler.com/eaaDev/EventNarrative.html

[Fowler IntegrationDatabase] Fowler, Martin;
http://martinfowler.com/bliki/IntegrationDatabase.html

[Fowler LayeringPrinciples] Fowler, Martin;
http://martinfowler.com/bliki/LayeringPrinciples.html

[Greg Young] http://codebetter.com/blogs/gregyoung

[Nilsson ADDDP] Nilsson, Jimmy; Applying Domain-Driven Design and Patterns; 2006

[Nilsson Bricks] Nilsson, Jimmy; http://www.jnsk.se/weblog/posts/bricks.htm

[Nilsson NED] Nilsson, Jimmy; .NET Enterprise Design; 2001

[Nilsson SOA-Qs] Nilsson, Jimmy; http://www.jnsk.se/weblog/posts/soa-QSemantics.htm

[Webber Anemic Service Model] Webber, Jim;
http://jim.webber.name/2008/04/19/30b4f0e9-f67a-4310-bf38-ca0a3423206e.aspx