Domain Dynamics – The Microservices Achilles heel

Einar Landre & Jørn Ølmheim
Statoil

Introduction

In this paper we will share some experience from implementing early service architecture back in 2002-2004 using Enterprise Java.

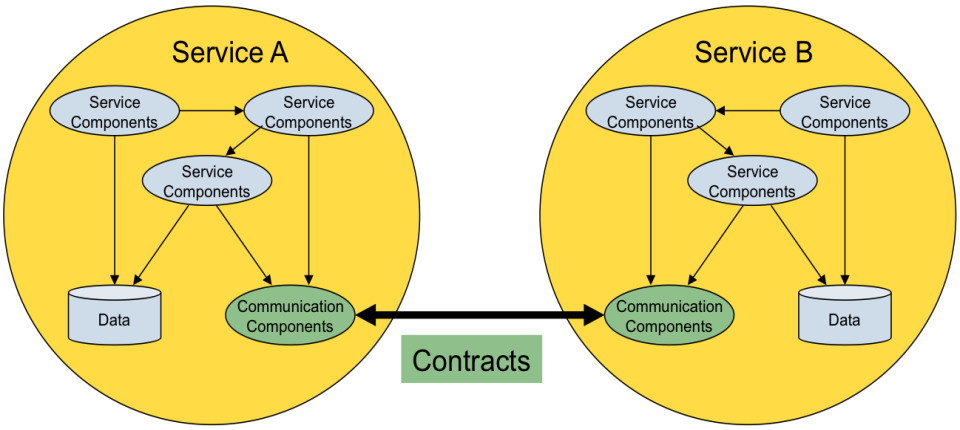
Based on our experience we find Microservices [1] to represent a major step in the right direction, but we think that success depends on replacing the data centric design paradigm used for most enterprise systems with a design paradigm addressing the dynamic properties of the domain.

The Delta Architecture

At the turn of the century Dutch retailer Albert Heijns developed their Delta architecture to support their differentiation strategy. The Delta architecture consisted of a set of both business and technical implementation principles:

- Asynchronous
- Plug & play
- Distribution
- Usage of layers
- Data encapsulation
- Contracting

A service was defined as a software component that supported a business process or area, and added value in a production line. The service model was depicted in the following manner.

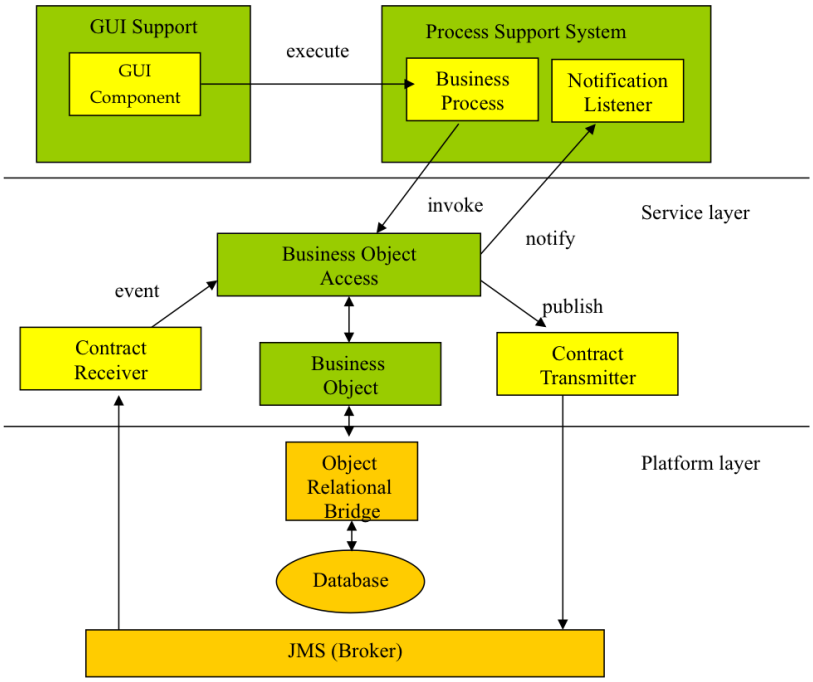


The Delta service model was heavily inspired by multi-agent designs, where services were seen as a network of collaborating agents, each responsible for defined tasks or processes.

Notes on realisation

The two main implementation constraints enforced was data encapsulation and layering, leading to each service to have a private database combined with Enterprise Java using IBM WebSphere (We used JBOSS for development). Late 2003 we had this diagram produced to illustrate the frameworks and architectural building blocks used.

Some of our design decisions were good, others not that good. One of the good one was to use JMS messaging wrapping it in a pluggable message-handling framework decoupling message processing from message transport. Message handlers where loaded dynamically based on message type.



Another was to implement domain (business) objects as POJOs and to use an Object Relational Mapper. No handwritten SQL was the mantra. The business object access mechanism was typical a façade of some sort.

On the negative side, business objects was processed by CRUD based scripts, though with some exceptions. One of those was an event mechanism solving assortment hierarchy state. It took us 6 weeks to figure out and contained less than 100 lines of Java code.

The user interface was enforced to be web based. Here we had many challenges and ended up with a custom-built framework extending Struts. We were early adopters of web technology for data intensive user interfaces. Today it would be fair to say that the technologies were not mature enough for our needs.

A final remark is that we think the industry derailed around the turning of the century. We had object-oriented languages such as Java, without proper understanding of how to build object-oriented systems. Object oriented thinking was lost in vendors focus on big-iron infrastructure such as Enterprise Java Beans.

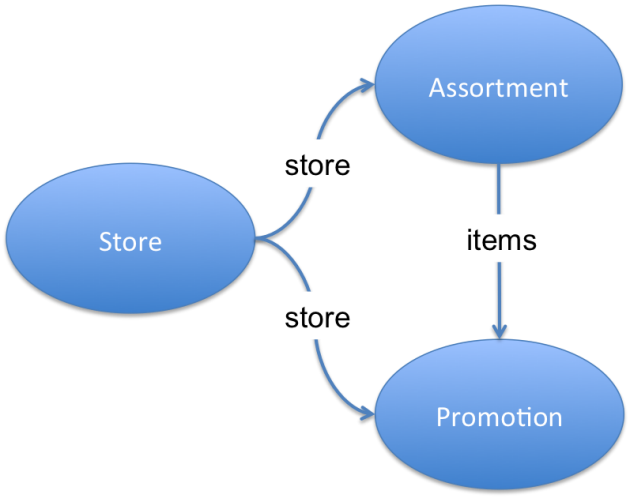
Finally we will write some words about the domain and how it was addressed. Remember that almost all developers where new to the domain. The retailing domain can be seen as a collection of three main objects: Store, Assortment Hierarchy and Item.

Of these, Item is the most complex. A can of beer and a six-pack of beer are both items in their own right that might have different price policies. We started with Store because it was the simplest and left the most complex, Item to the end. That was a big mistake. We should have started with Item, the most complex one and seen that in relationship with Store and the Assortment Hierarchy.

Lessons learned

Screwing up the domain model

In terms of domain driven design [2], each of the Delta services are bounded contexts even tough we did not think about it that way at the time. The figure below illustrates how three of the services where tied together using contracts.



Both Assortment and Promotion need stores to be able to select what items to sell and what promotion to run.

In addition Promotion need items (the things to sell) to be able to build its campaigns.

This is the point where we made the biggest design error:

We copied the internal storage model of stores and items from their originating services to the subscribing services, a pattern we repeated all over the place.

I have over the years asked myself the question; what forces made us think this was a good idea. After years of thinking and experiencing the state of other legacy systems from the 1990ties we think this boils down to three things:

1. The dynamics of the domain was not understood at the time the system wide design decisions where made.
2. We modelled with objects using an object-oriented programming language, but our modelling approach was data-centric.
3. A misguided attempt to reuse domain concepts and models across services.

A service encapsulate a process, it takes some input and create some output. In the case of stores, input includes location, size and type of store. Outputs are store related events, like the opening of a new store, changes to opening hours and changes to format (type of store). To create these events the store service encapsulates data and rules.

The challenge is that when the dynamic properties of the domain are not fully understood, we easily start out with what is perceived as the easy part, the structure of the domain. In this case the structure of how to represent the store model. Then we copied that data model into all other services needing stores.

The same mistake was done with the assortment hierarchy and the item structure. Then we dumbed down the domain events as CRUD operations. The domain objects was treated as records, not objects.

This approach is what we think of as data-centric modelling. Instead of modelling domain dynamics in the vertical axis, we begin modelling structures horizontally, adding dynamics as we get insight. The result is a “global” data model that is manipulated using CRUD operations packaged in scripts growing from day to day as domain insight increases and new requirements are added. This is probably the reason we still find the 10.000LOC methods in object-oriented languages such as Java. Instead of modelling domain behaviour as classes and small functions we type up long scripts containing the mother of all “switch” statements.

Succeeding with Microservices

Even instrumented with the sound service oriented principles defined by the Delta architecture we ended up building data centric monoliths. Our concern is that Microservices can end in the same tar pit, unless the root cause leading to tightly coupled monoliths is addressed and eliminated.

Acknowledging that poor software practice is not only about one thing, we dare claim that the missing understanding of domain dynamics is one of the main caveats, and therefore one of the biggest threats against the Microservice architecture.

We must avoid moving into a data-centric modelling paradigm where the domain model dynamics is dumbed down to CRUD operations wrapped in endless scripts. The question is how to do this?

Capturing the Domain

The father of object-oriented programming, late professor Kristen Nygaard always talked about objects as entities that encapsulated state and behaviour, entities that responded on requests based on their state and as entities that mirrored the real-world.

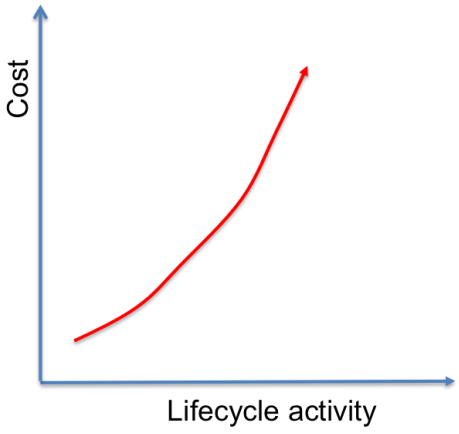
In one of his last talks he stated the following: “Teaching object-oriented programming must begin with exposing students to complex situations that only may be mastered by understanding and using object-oriented concepts”. Pointing at the fact that object-oriented concepts is about comprehending and organizing complexity. He then illustrated this through Café Objecta [3].

Back in the early 1990ties the industry experienced an emerging understanding of the importance of formalizing object-oriented modelling and thinking and we got what became to be known as Object Oriented Analysis and Design [4].

The key here is that object-oriented thinking is about capturing the behaviour and structure of the domain. The same can be claimed about Microservices. They are primarily about domain behaviour and structure in terms of what services do we need. The tricky part is domain behaviour.

Requirements elicitation and analysis

Domain behaviour is defined by system requirements. A poor understanding of system behaviour during implementation indicates that the system requirements are not understood.



Since we know that the cost of fixing a defect grows almost exponentially over the products lifecycle phases, our claim is that the requirements engineering practice need to be rethought for Microservices to succeed.

In agile methodologies functional requirements are defined by small stories. Cranking out good object models and code structure from a collection of small functional stories on the fly is not easy.

Therefore we suggest it is time to revisit some of the proven object-oriented analysis technics and methods established decades ago and supported by modelling languages such as UML.

Object-Oriented Analysis

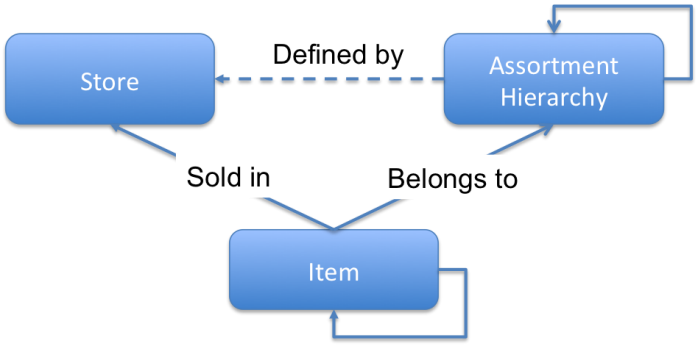
The primary tasks in object-oriented analysis are:

- Find the objects
- Organize objects
- Describe how objects interact
- Define the behaviour of objects
- Define the internals of objects
- Document requirements objects must satisfy

Bruce Powel Douglas in Doing Hard Time [4] provides a structured approach for how this can be done for real time systems. These techniques are applicable for less time critical domains as well. For domain behaviour we find the following tools:

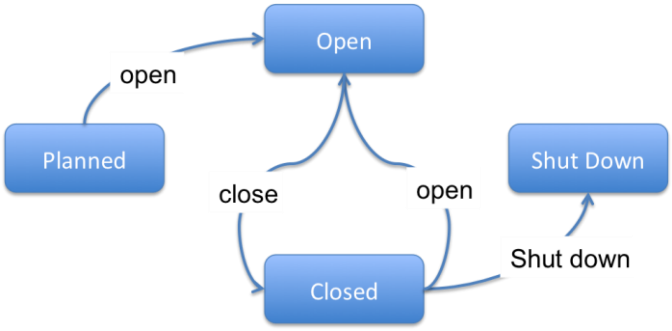
1. Use cases or other informal textual descriptions of how system functions are performed.
2. State charts capturing object states, events and state transitions.
3. Timing diagrams (for time sensitive domains).
4. Activity diagrams.

Of these tools we suggest to start out with the informal scenarios and from those find the objects and their behaviour. Then, for the main objects, identify states and the events that trigger transition from one state to another and draw a state chart.



To illustrate what we mean about domain models, here a simplified static model of the assortment service, whose job was to manage the assortment hierarchy its items, and to make complete assortments available for various types of store.

From this kind of lightweight domain models we can start explore the internals of each main object. We know Item is a composite and the same we know about Assortment Hierarchy. What operations do we need to build these structures? What are the states and events that driving this model? Bellow an example of how states and events for a store can be modelled (for illustration only).



Think now how we can associate rules to the event that a store changes state from Planned to Open. One constraint could be that the store cant open without having a valid assortment.

The point is to illustrate how events tied to one domain object might drive state changes in other domain objects. Requirements analysis is about capturing these things so they can be implemented. In terms of domain driven design this is all about building the ubiquitous language.

Formalizing the domain

Business domains like retail, drilling and oil trading are characterized by being informal. They are not well defined in mathematical terms. This means that different organisations used more or less the same words, but there are always some nuances that create trouble and misunderstanding.

Developing support software for a domain that is vaguely defined and fuzzy is almost impossible. Domain experts are not used to the rigour software development demands. This adds additional burdens on the development organisation that must formalize the domain. In terms of domain driven design, this is about developing the ubiquitous language. To ease this process analysis patterns can be a useful tool. They provide general blue prints for how to structure common domains concepts [6].

Realisation

With a deeper understanding of the domain dynamics we are much better positioned to organise the domain into a set of cohesive loosely coupled services. Even though we modelled requirements using objects, it does not imply we have to use an object oriented language to program it. (Personally I have written more C code in an object-oriented manner than Java). The point is, object oriented thinking does not require an object-oriented language for realisation.

But what you must do is to find a way to represent events and domain behaviour in terms of state machines. One way is to build an event / state matrix and populate each cell with the address to the actual event handler for that state. Another way is to build it with objects. What's the best thing to do, pick your choice, but do remember that state transitions most likely should be atomic and need protection.

Finally some thoughts on the difference of building new services from scratch versus refactoring a legacy system. Building from scratch means some more up front analysis. It's the cheapest way to figure out how to structure the domain. For each service we need to know functional scope and its main domain objects, their states and their behaviour and their relations.

When refactoring a legacy system, the objects are more or less defined by the data model. The challenge now is to find where to cut the data model and say that these tables belong to service A and these to service B.

The point here is that these types of situations require different takes on the task.

Summary

We have tried to bring forward our concern that without proper domain modelling focused on capturing the dynamics of the domain Microservices will add the complexity of distributed programming, without addressing the weaknesses and complexities coming from a data centric development approach where the domain model is dumbed down to CRUD operations wrapped in monolithic scripts.

Our recommended take on this challenge is to revitalize object oriented analysis and design techniques focused on capture of domain dynamics and find ways to use these in an agile manner. The crux is to balance up-front analysis with refactoring of the code base to ensure a readable and manageable modular code. For this to work analysts and developers need to be trained in object-oriented thinking and modelling techniques.

References

[1] Building Microservices, Designing Fine-Grained Systems (Early Release), Sam Newman, 2014

[2] Domain Driven Design, Tackling the complexity in the heart of software, Eric Evans, 2004.

[3] Kristen Nygaard, 2001, Café Objecta
http://www.bandgap.cs.rice.edu/classes/comp410/S03/Mall%20Sim%20Project/OO-PedagogicalApproach.ppt

[4] Object Oriented Analysis and Design
http://en.wikipedia.org/wiki/Object-oriented_analysis_and_design

[5] Doing Hard Time, Developing Real-Time Systems with UML, Objects, Frameworks and Patterns, Bruce Powel Douglas, 1999.

[6] Analysis Patterns, Reusable Object Models, Martin Fowler, 1997.