Hans Karlsen

MDriven The book

Doing effective Business – by taking control of Information

Hans Karlsen, Stockholm, Sweden
2016-01-23

What is MDriven

MDriven is software modeling, prototyping and execution made easy. You do not need to know Java or c# to fully test your ideas for advanced information handling. You just need to learn how to model and you will get instant gratification. If you already know c# or Java you will be amazed on how quickly you can verify your ideas with MDriven compared to alternative prototyping schemes.

Furthermore MDriven can take you all the way to a finished system – in the cloud or on your server – with web based user interface or/and with a rich client application.

MDriven uses full-fledged relational databases as information storage and will scale with your needs.

MDriven is totally transparent in architecture and you will never lose control of your data.

With MDriven you can continuously progress systems easily making you truly agile.

MDriven is like having a software developer made of software – that does what you say really fast.

If you used to use excel to handle information, MDriven will strike as much more type safe, precise and efficient way to get things done fast and to have things stay done.

If you used to write requirements for software with text, images and mockups, MDriven can change the way you interact with stakeholders and developers – removing a lot of confusion and speeding up the process.

Modeling will make you think further and faster about information.

MDriven is free to use for up to 50 information classes – enough for even non trivial systems.

Introduction

As I have been working as a software developer and software architect in Sweden for the last 20 years I have seen a lot. I am not going to bore you with my historic reflections at all.

That was my first attempt on an introduction for this book. Then I came to realize that most things we have done with MDriven is very much anchored in the historic reflections of things we have experienced in the past. So I guess what I need to say is more like this: Having worked many years as a software architect I have seen a lot of things that change – but also very much that sort of stays the same over time – a stable core.

What is the stable core? The need to store and retrieve the information we are dealing with. The need to somehow display it for users and handle users need to update it according to the rules we want to enforce. This is the technology of any application or system and is what every system developer needs to deliver – using technology modern at the time of implementation. I will refer to this as system modernity.

Furthermore we have the need to understand the business information in the system and how the rules that governs the information's evolution and consistency protection works. This part I will refer to as the system gist.

On the other hand we all know that in order to make our software sell – to an external market or to in-house users - it must be perceived as modern and cool, but in the software business modern and cool change every year or so, at least on the surface.

So mixing the system gist down into a format that is modern now but that will be obsolete in a couple of years seems like something one should avoid.

It would be better if the system gist somehow could be separated from the currently modern implementation strategy – which we know for sure will be less modern as time goes by.

This book is on how I suggest that we deal with system gist and system modernity.

Praise to UML

UML from Wikipedia: "The Unified Modeling Language (UML) is a general-purpose modeling language in the field of software engineering, which is designed to provide a standard way to visualize the design of a system.

It was created and developed by Grady Booch, Ivar Jacobson and James Rumbaugh at Rational Software during 1994–95, with further development led by them through 1996."

To fully convey my appreciation for UML I must explain how I look on the world.

Below I define three areas that will help to explain my reasoning.

Fashion: fashion is what goes together with what and if it is hot and sexy in the manner that people somehow crave it without further need to understand it.

Modernity: I attribute modernity to ways to solve known problems. Tools and strategies have a modernity aspect. A more modern tool is not necessarily better – but it often is since the new tool has had the advantage of creation in a world that has more knowledge than the world that created the old tool.

System gist: a system in this context is anything that combines a series of ideas and actions in order to produce value. The gist of the system is the system description stripped from everything that is either fashion or modernity as described above.

Let me exemplify these definitions. A car manufacturer is very much reliant on fashion. How the lines of the car body appeals to the target audience is very important but almost totally based on feelings and soft aspects that are hard to measure. The modernity aspect of car manufacturing is important for the manufacturing process. What tools to use. How to apply industrial robots. What third party systems to include like anti-lock braking and airbags. The system gist is captured in the design phase of the car construction process. It involves all the inherited knowledge about what is important for cars in general and also some new things that are important for this car model in particular.

Second example: A surgeon. Modernity provides important tools to diagnose a patient, like MRI. It also provides ever better tools in fixing what is wrong, like with minimally invasive surgery. System gist is for the surgeon the knowledge on what and where to cut and how and why organs act as they do. It is important for a surgeon to be able to draw the intuitive line between gist and modernity. Having the latest tools will not be enough if you are not educated in what to look for and how to act

upon what you find. As a patient you will want a surgeon that masters both gist and modernity, and does not neglect one for the benefit of the other. When it comes to fashion it is important for the plastic surgeons – but they too must have the main focus on gist and modernity or you will end up with a defect system. I know little about the field of surgery but I am sure that if I knew more I would also see aspects of fashion in appendix removal procedures. It might be how hi to cut and how to stitch the wound that has no immediate support in current science but feels right or looks good.

Where is the software industry in this spectrum? The software industry differs from the two examples given by not having ONE fixed or slow evolving system gist. Software industry is actually about producing new systems and as such new areas that have gist, modernity and fashion of their own. The software industry is one Meta level up compared to surgery or car manufacturing. It is in this way not just a human activity like surgery but an activity of activities. This is what makes software development a field that will leave no other area of human activity untouched.

Hundreds of new unique software systems are finalized every day and they will resemble each other when it comes to modernity and fashion. What makes them each unique is mostly their system gist.

Look at what all the apps on mobile devices share: their execution environment, their use of the network, the interaction patterns and widgets. Modernity and fashion is a time marker that makes it easy to guess the time of construction for a particular software system.

The modernity aspect of software development is very important to be able to produce a well behaving system. The fashion aspect of system development is very important to attract users and make the system intuitive to use. These two areas, reused over and over, also evolve at a rather high pace. Refined strategies – or as I call them – modernity aspects - on how to build software systems is a topic discussed endlessly in the developer community.

It is easy for software developers to completely get lost in the modernity aspects. As they do they will leave less room in their minds for system gist. When we get software developers that move cross field and mostly work with modernity and fashion, is anyone taking care of the gist?

What sets one software system apart from another is mainly its gist not its modernity.

UML is all about system gist. This is why we need UML or something that solves the same problems. We use UML to describe system gist in an easy, clear and partly visual format without any possibility for alternate interpretations. UML is the most prominent way to handle system gist.

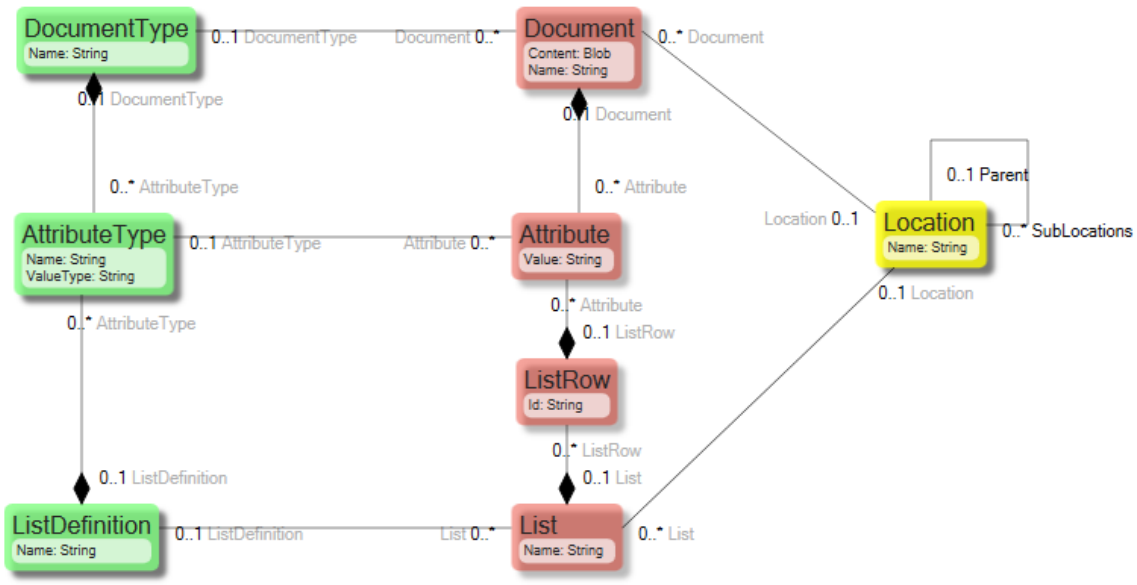
For a software engineer it is important to be able to quickly place arguments on design decisions in the correct category. If it is about system gist then there will be facts to research in the domain of the system in order to make the best decisions. Is it about modernity – then it is important to analyze best practices from the development communities and consider the pace and gain of change to see which path to take. Is it about fashion – stand back and let the end users or market decide. Take a vote if they do not reach a consensus.

Just as software frameworks – like Entity Framework, Hibernate or the like, aims to help developers with modernity issues there are manufacturers of generic software systems that aim to solve everything. These are Meta systems where you somehow can describe your system gist and then you would be done.

Since a meta system like this is a holy grail – the search for it engage many software companies. Of course many claims are made that the grail is found. But is it really? And is there really such a thing as the holy grail in the first place?

SharePoint is a tool that I have seen used like this on a number of occasions. As SharePoint may have an appealing modernity and may be fashionable in certain populations it is an easy sell if it also lends itself to handle ANY system gist.

As SharePoint is a software system it has a system gist of its own. One that may be described with UML.



As you see the gist of SharePoint is tiny. In fact the gist of a Meta systems often are tiny.

Alan Turing thought us in 1936 that one can build universal machines – a machine that can simulate all other machines. Universality of machines is reached very fast as Stephen Wolfram showed us in his book “A new kind of science (NKS)”-2002 where he suggest that a cellular automate of only two state and three colors is universal. Alex Smith later proved this in 2007.

In light of this it is not surprising that the system gist of SharePoint is Universal – so it can be used to implement any and all systems conceivable. This may seem fantastic – but just because something is possible does not necessarily make it a good idea. It may for example be possible to build anything by gluing grains of sands together under a microscope – but not practical or economical defensible.

Folding down the system gist into ListDefinitions in SharePoint is not the best way to treat the gist. In that format it is not easily evolved and maintained. It is however possible, I do not question that.

My opinion is that the best way currently available to describe system gist is UML using the language of the domain. Refrain from building Meta systems.

Accept that each area of human activity has its own gist that deserves its own UML description. When each area has its own gist clearly described in UML it is easy to maintain and evolve. Free from modernity and fashion issues.

I wish all developers would be aware of the three different areas of gist, modernity and fashion. It is my belief that we limit our ability to develop everything due to lack of focus and lack of discussion on system gist. Having a language for the gist opens up for discussions and thinking that helps development in all areas of human activity.

What if UML was forbidden?

If UML or other structured ways to define system gist was forbidden – what would happen then. Maybe UML need not be forbidden – the effect would be the same if UML simply was not used.

Well the system gist still exists – even if it is not explicitly documented. It must be extractable from the source code of any running system since the system is a transformation of the system gist any how that gist was captured in the first place.

Maybe the system gist is held in documents that outline requirements or prose text that describe scenarios that the system should solve or support with. Maybe the developers were obligated to write other prose documents that act as the documentation of the produced software.

To document software is boring compared to coding for any developer. Code can be compiled and type checked so that bugs in it can be removed. Code can also be executed and further tested to ensure that the idea we wanted to cover really is covered. Documentation does not work that way.

Most likely any existing documentation is filled with bugs beyond belief since the process of verifying lacks compilation and testing. Since developers suspect this, developers seldom trust and seldom consult documentation for existing software systems. Instead they have a tendency to go to code. If the only option is prose documents they are probably correct. Even if the developer finds information in the code – it is important to remember that the code is just the original developer's interpretation of the requirement – and this need not be equal to the requirement. Software tools and libraries may follow other rules – but when an experienced developer is confronted with a software system he or she very seldom expect that the documentation is complete or correct.

Prose documentation of system-gist is by the reasons stated above almost never used for anything except making the stakeholders feel a bit safe – like life jackets on an airliner – that fly over land – it was never intended to help anyone – but the stakeholders want it so we provide it.

Another way I have come across to protect and keep the knowhow that is the system gist is what I would describe as "invest in the team".

Let the skilled and motivated software developers solve things with their talent and memory. Let the collective team memory be your knowhow and documentation of the system gist.

This is the common way for most businesses that produce software that I have come across. I argue that this is not a strategy. It is an abdication of ownership and control. Developers might not see this as a problem at first since they take pride in the trust management place in them.

Still I have seen many cases where this strategy over the years create a chasm between the ones that know (developers) and the ones that make decisions (management) – it usually ends with a collapse that benefits no one. Since developers now are left on their own to decide where the resources should go – into system gist or into modernity or maybe into fashion. They will soon lose the managements trust since management lacks control and does not understand why the developers chose to prioritize the way they do.

Nevertheless this is commonly how small and midsized businesses handle their software investments today. Let the code speak for itself – there is no other representation of the ideas within the software than the code itself. The very same code that is strongly flavored by the team members take on what is modern and what is fashionable this year – or was – last year.

Luckily UML is NOT forbidden

For prose document writers and for nothing but code cowboys there is a missing link – a way to describe system gist separated from the flavor of the year implementation method in a format that is not as flimsy and open to interpretation as prose documents.

What I propose here and what many has proposed before me is that we can document with models. The models are more descriptive than prose text and leave less room for alternate interpretations. At the same time models are less complex and easier to read than code written in the flavor and style of the year.

The model is then home of the system gist. Here it can be understood, discussed, criticized and evolved long before it has taken the expensive form of implementation code.

What is somewhat new in what I say is that Models can be compiled and executed just like code. We can then focus on using the currently modern technique to execute the model. This way the modeled solution can have a much longer lifespan than the user interface or delivery method that are heavily subjected to modernity and fashion.

Models can cover the uniqueness and true solution of your software.

A model executer brings your model to life in a specific environment. When one model executer goes out of style and something new is requested by the market – we do not rewrite the system gist – we create a new model executer – and feed it the very same model.

MDriven is the latest model executer we have done – but we have been involved in doing many. A model that was executed with Delphi 1999 (BoldSoft) can be executed with c# MVC5 or with WPF (CapableObjects) today. It was executed with Windows Forms in 2003 (Borland, Embarcadero), with Silverlight in 2007, with ASP.NET in 2005. No doubt will there be new model executers when modernity so requires – and sure enough MDriven Turnkey that brings any system gist to AngularJS is currently available.

One key to a good investment in software is to avoid entangling things that change for different reasons and with different intervals and speed. Your system gist change and evolve along with the business it supports. The modernity of the solution changes by market forces no one can control but everyone must adapt to.

I propose that we should keep these two areas apart so that they do not get confused as being the one and same problem.

Model driven development is by me defined as: develop system gist in its own machine readable format. Build a software machine that turns the system gist into a complete software system and fulfills the required modernity aspects. Giving such a machine a descriptive name: ModernGistExecuter – we at CapableObjects call our implementation MDriven Framework.

What is not to like?

Having worked with model driven development for the last 20 years I have many times been surprised on how much resistance we have met. It is not like other developers are indifferent or do not care. Many do care – but the reactions are surprisingly often skeptical and negative.

We have tried to take this as an indication that we are on to something relevant and beautiful. Big and disruptive shifts never come without agony and pain. The suggestion is that it is a defense mechanism that kicks in. People are afraid that model driven development will change their current set up – and resistance to change is natural and triggers unconsciously.

I do not believe in the anti-change theory. I think it is a simple case of not correctly separating the different issues at hand: modernity versus gist.

Developers know that everything changes. Experienced developers have been left stranded with abandoned techniques and products thru out their careers. It is not a good feeling and it is not good for business or the credibility of the developers. No one likes to be forced into change by external influence, but a product that has lost new development and support must be replaced. This has led most developers into a minimalistic approach when it comes to using products. Minimalistic or gigantic – trust only the really big companies in software like Microsoft or Oracle – and things that are transparent like open source and simple tools.

The problem with traditional development that blends and mixes modernity with system gist is that change hits so hard. Everything must be rewritten once a technology change is required. If the gist changes a lot – rewrite, if the modernity changes a lot – rewrite, if the fashion changes heavily – rewrite. There are more reasons to abandon made investments than what any investor dare to think about. All this because of the mix up of the three different areas.

If we separate the gist from modernity we will find that most changes in technology leave the gist untouched. In fact all different types of gist will be handled by many different approaches and technologies over its lifetime. For this we can plan from the start.

Since few have had the opportunity to try this in real life, few know the benefits it brings.

Separating the gist from modernity protects that part of the system from the IT-wind-of-change that is always blowing at hurricane strength. At the same time free up the modernity area to change without having to change all the gist stuff at the same time. Both areas will win by keeping this separation.

As an information architect or developer of domain work organization, the gist is the most interesting area. It is the theory and motive. This is also were the structural capital of the enterprise

resides. Having this documented in a useful and actively maintained format is very attractive to any business.

For classic software architecture Modernity is extremely important as it ties into the projected lifespan of the system, maintainability, how hard it is to build, usability concerns, security, efficiency and overall investment sanity like “our maintenance burden must not be to disparate”. However for a business wanting to build in house systems in order to become more automated the modernity aspect is still important but they should put system gist in the front seat. It will be good for the whole business to structure their knowledge of whom they are and how they do things.

I agree that if you use no tools to manage your gist it is easier to just mix gist and modernity together and let it stew. Then sit back and wait for the inevitable rewrite need that forces you do it all again. I propose that this is the wrong approach and an approach that is not very smart or efficient.

Looking back on many discussions over the years I now believe that a lot of developers and software architects do not separate the system gist area from modernity with enough clarity. Many software developers also focus mainly on modernity issues – after all as a professional developer this is what you can reuse for different clients that all have different gist. I believe that this limits their ability to produce robust long term systems. I suspect that this is part of the explanation why software development has to low success rate. It can also explain why legacy ERP systems that are so far from modernity that it hurts still have an appeal on the industry.

It is my belief that we must be vigilant to correctly identify modernity and fashion arguments when solving system gist problems. If not, people will wrongfully let modernity or fashion arguments color their gist and by doing so confuse both other developers and stakeholders.

This book is primarily on how to use MDriven to handle and manage gist and secondary on how to use MDriven to help you manage modernity. I will show how the modernity issues are managed with standard techniques in Visual Studio with no limitations or assumptions – so that you are ready for all new modernity requirements that will be sailing up. Lastly this book will cover how the gist and modernity offered by MDriven Framework easily is dressed up with the current fashion.

What is next

I propose and will show examples of how you can maintain the gist and idea of your software in a model – and a how you can apply existing or create your own model execution engines to act as modern and current delivery mechanisms for your solutions.

The steady pace of shifts in the markets perceptions on what is new and cool often paralyze companies from building support systems and with them control their information. The main problem is that most techniques today mix the two different problems of system gist and modernity. Entangling these two problems makes solving them as one hard and risky and something that many businesses will avoid.

To summarize: Model centric – describe the system gist in a model that use a model executer of correct modernity level to bring it to life. The model is then the documentation, the knowhow and the gist of the system– invest heavily in this. Put modernity and fashion in the model executer and into user interface and delivery –invest in this too but keep in mind it may change soon. Make sure

you involve management in the system gist governance so that they are in control and can take informed decisions.

The series of tools and strategies presented here reduce the effort needed for business to take control of and own their information. It does so by introducing a clear separation between system gist and modernity.

It is my belief that companies that own and control their information are better equipped to compete than companies that are clueless to it. A no brainer of course – but still – most companies lack a good and deep understanding of their information. The few that have control often spend too much resource trying to evolve system gist and modernity in one complex and risky process.

It is the aim of the tools and strategies presented in this book to show new ways to produce and maintain domain specific software support systems – and to vastly reduce the costs and increase the quality and speed on how to produce and maintain them. It is not magic – it is just a matter of raising the abstraction level a bit and refrain from entangling problems that are much better solved separately.