CHAPTER 2
What is Data-Centric?

Our position is:

> A data-centric enterprise is one where all application functionality is based on a single, simple, extensible data model.

First, let's make sure we distinguish this from the status quo, which we can describe as an *application-centric* mindset. Very few large enterprises have a single data model. They have one data model per application, and they have thousands of applications (including those they bought and those they built). These models are not simple. In every case we examined, application data models are at least 10 times more complex than they need to be, and the sum total of all application data models is at least 100-1000 times more complex than necessary.

Our measure of complexity is the sum total of all the items in the schema that developers and users must learn in order to master a system. In relational technology this would be the number of classes plus the number of all attributes (columns). In object-oriented systems, it is the number of classes plus the number of attributes. In an XML or json based system it is the number of unique elements and/or keys.

The number of items in the schema directly drives the number of lines of application code that must be written and tested. It also drives the complexity for the end user, as each item, eventually surfaces in forms or reports and the user must master what these mean and how the relate to each other to use the system.

Very few organizations have applications based on an extensible model. Most data models are very rigid. This is why we call them "structured data." We define the structure, typically in a conceptual model, and then convert that structure to a logical model and finally a physical

(database specific) model. All code is written to the model. As a result, extending the model is a big deal. You go back to the conceptual model, make the change, then do a bunch of impact analysis to figure out how much code must change.

An extensible model, by contrast is one that is designed and implemented such that changes can be added to the model even while the application is in use. Later in this book and especially in the two companion books we get into a lot more detail on the techniques that need to be in place to make this possible.

In the data-centric world we are talking about a data model that is primarily about what the data means (that is, the semantics). It is only secondarily, and sometimes locally, about the structure, constraints, and validation to be performed on the data.

Many people think that a model of meaning is "merely" a conceptual model that must be translated into a "logical" model, and finally into a "physical" model, before it can be implemented. Many people think a conceptual model lacks the requisite detail and/or fidelity to support implementation. What we have found over the last decade of implementing these systems is that done well, the semantic (conceptual) data model can be put directly into production. And that it contains all the requisite detail to support the business requirements.

And let's be clear, being data-centric is a matter of degree. It is not binary. A firm is data-centric to the extent (or to the percentage) its application landscape adheres to this goal.

DATA-CENTRIC VS. DATA-DRIVEN

Many firms claim to be, and many firms are, "data-driven." This is not quite the same thing as data-centric. "Data-driven" refers more to the place of data in decision processes. A non-data-driven company relies on human judgement as the justification for decisions. A data-driven company relies on evidence from data.

Data-driven is not the opposite of data-centric. In fact, they are quite compatible, but merely being data-driven does not ensure that you are data-centric. You could drive all your decisions from data sets and still have thousands of non-integrated data sets.

Our position is that data-driven is a valid aspiration, though data-driven does not imply data-centric. Data-driven would benefit greatly from being data-centric as the simplicity and ease of integration make being data-driven easier and more effective.

WE NEED OUR APPLICATIONS TO BE EPHEMERAL

The first corollary to the data-centric position is that applications are ephemeral, and data is the important and enduring asset. Again, this is the opposite of the current status quo. In traditional development, every time you implement a new application, you convert the data to the new applications representation. These application systems are very large capital projects. This causes people to think of them like more traditional capital projects (factories, office buildings, and the like). When you invest $100 Million in a new ERP or CRM system, you are not inclined to think of it as throwaway. But you should. Well, really you shouldn't be spending that kind of money on application systems, but given that you already have, it is time to reframe this as sunk cost.

One of the ways application systems have become entrenched is through the application's relation to the data it manages. The application becomes the gatekeeper to the data. The data is a second-class citizen, and the application is the main thing. In data-centric, the data is permanent and enduring, and applications can come and go.

DATA-CENTRIC IS DESIGNED WITH DATA SHARING IN MIND

The second corollary to the data-centric position is default sharing. The default position for application-centric systems is to assume local self-sufficiency. Most relational database systems base their integrity management on having required foreign key constraints. That is, an ordering system requires that all orders be from valid customers. The way they manage this is to have a local table of valid customers. This is not sharing information. This is local hoarding, made possible by copying customer data from somewhere else. And this copying process is an ongoing systems integration tax. If they were really sharing information, they would just refer to the customers as they existed in another system. Some API-based systems get part of the way there, but there is still tight coupling between the ordering system and the customer system that is hosting the API. This is an improvement but hardly the end game.

As we will see later in this book, it is now possible to have a single instantiation of each of your key data types—not a "golden source" that is copied and restructured to the various application consumers, but a single copy that can be used in place.

IS DATA-CENTRIC EVEN POSSIBLE?

Most experienced developers, after reading the above, will explain to you why this is impossible. Based on their experience, it is impossible. Most of them have grown up with traditional development approaches. They have learned how to build traditional standalone

applications. They know how applications based on relational systems work. They will use this experience to explain to you why this is impossible. They will tell you they tried this before, and it didn't work.

Further, they have no idea how a much simpler model could recreate all the distinctions needed in a complex business application. There is no such thing as an extensible data model in traditional practice.

You need to be sympathetic and recognize that based on their experience, extensive though it might be, they are right. As far as they are concerned, it is impossible.

But someone's opinion that something is impossible is not the same as it not being possible. In the late 1400s, most Europeans thought that the world was flat and sailing west to get to the far east was futile. In a similar vein, in 1900 most people were convinced that heavier than air flight was impossible.

The advantage we have relative to the pre-Columbians, and the pre-Wrights is that we are already post-Columbus and post-Wrights. These ideas are both theoretically correct and have already been proved.

THE DATA-CENTRIC VISION

To fix your wagon to something like this, we need to make a few aspects of the end game much clearer. We earlier said the core of this was the idea of a single, simple, extensible data model. Let's drill in on this a bit deeper.

Simple

> *Everything should be made as simple as possible, but not simpler.*[1]
>
> Albert Einstein

We have observed that most enterprises have at their core a conceptual model that consists of a few hundred concepts. By concepts we mean the sum total of the classes (e.g., tables and entitles) plus the properties (e.g., columns and attributes) . These are the distinctions with which programmers work. Because they are displayed in forms and reports, they are also the distinctions with which end users must deal.

---

[1] This is Roger Sessions "simplification" of Einstein's original "It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience." From "On the Method of Theoretical Physics," the Herbert Spencer Lecture, Oxford, June 10, 1933.

Let's put this in contrast to what is typically implemented in a large enterprise. Very few applications have models that are simple. A typical application has thousands of concepts. Some Enterprise Resource Planning systems (ERPs), such as SAP or Oracle's ERPs, have hundreds of thousands of concepts, as do Electronic Medical Record systems (EMRs). Individual systems are already many orders of magnitude more complex than what is needed to manage the entire enterprise. When you combine many of these complex systems together, the sum total is staggeringly complex. We thought we hit a high-water mark when we discovered a firm that had 100 million concepts in their implemented systems, but the early results from another firm predict that they may hit the 1 billion mark. This is 1 billion concepts, not 1 billion records—obviously, they have many billions or trillions of records. But the fact that they have come up with 1 billion ways to type or categorize such records indicates just how far we have come from the ideal.

We were at a client recently, and Jans Aasman made an observation that was so profound, it is still ringing in my ears. Jans is CEO of Franz Inc.,[2] the provider of a leading semantic graph database. Jans has had a front row seat and participated in many projects that ushered in the exact kind of change we're referring to in this book. He said, and I may be paraphrasing a bit:

> *I was at a prospect firm recently, in the healthcare space. One of their analysts told me in a rather matter of fact way that they had data on patients in over 4,000 tables in their various systems. This is the end result of forty years of relational thinking: it didn't occur to them that this was wrong. This arrangement essentially guarantees that you will never have a unified view of your patient.*
>
> Jans Aasman

We outline in this book how to reverse this. One of the key enablers is simplicity.

It is easy to come up with simple models. People do it all the time. However, usually when people oversimplify, the complexity goes somewhere else. Usually that somewhere else is in application code. Another important resting place for complexity is manual procedures. It is possible (and is actually the norm) to construct procedural workarounds to handle the cases not handled by a system. People often build rule-based systems on top of other systems to handle some of the distinctions omitted from the system.

It is possible to make a data model overly complex. It is possible to over simplify it. Is there a goldilocks zone where the model has "just the right" amount of complexity? We are finding that there is such a zone. We are finding it to be far smaller than even we would have thought when we started on this journey over a decade ago. There are techniques, which we will get to

---

[2] http://www.franz.com.

later, to keep a lot of the fine-grained distinctions from polluting the design space of your core model.

We have been finding that optimal levels of complexity are in the many hundreds of concepts, 300-500 seeming to be a reasonable target.

The interesting thing from the perspective of human cognition is that there is a real difference between 10,000 concepts and 400 concepts. It takes years, maybe an entire career, to really master 10,000 concepts. But 400 concepts are within reach of most motivated participants. As one of our clients said recently: "This is at a level that a motivated analyst could internalize it over a weekend." This is especially true when half of those 400 concepts are very known and already understood and agreed upon (including the concepts of "Person," "Unit of Measure," and "Geographic Location").

We usually start with a model of an enterprise that consists of a few hundred concepts. From that we specialize a particular subdomain (see extensibility below) for a given application. Our unstated goal is to add fewer than 25% new concepts for each subdomain. What that does is recognize that the sub domain is literally a subdomain and not something entirely new.

> An ontology is a conceptual model that captures formal definitions of classes and properties. A formal definition is a machine process-able set of rules that can be used to infer such things as class membership or the existence of property assertions.

In one case, we built a first cut enterprise ontology (a core conceptual model of the firm), which consisted of about 400 concepts. This was a company that made electrical devices. We created a sub domain model for their product catalog, specification, and configuration management. We then converted the data from their existing product catalog to the product ontology. When we went back and reviewed what we had done we noticed that their existing catalog management system (just one of hundreds or thousands of systems they had) had 700 tables and 7000 attributes, or 7,700 concepts in total. The part of their new model that we populated with the data from this system consisted of 46 classes and 36 properties (later in the book we will get to how these kinds of models can have fewer properties than classes, which is literally impossible in traditional systems). The fascinating thing was that as we began building product selection and configuration management systems on this new model, we proved that the new model had not dropped any of the fidelity of the original model. We had all the distinctions with about 1% of the complexity: (46+36)/(700+7000).

This is important at two levels:

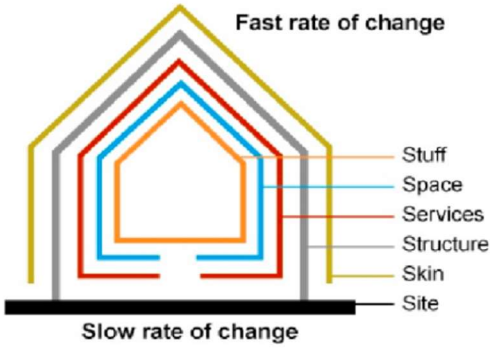
1. Everything you do with this model (write code against it, write rules in it, use it for big data analytics) is simpler, cheaper, and more reliable.

2. It shifts dealing with the data from the province of the expert (the person who understands the 7,700 concepts and how they relate to each other) to the true owners and consumers of the data.

EVOLVE-ABLE

"Extensible" in the definition really plays out two ways: evolvable and specialize-able.

A system is evolvable if it can gracefully adapt. Stuart Brand wrote a charming book called How Buildings Learn,[3] in which he made an observation and built a conceptual model. The observation was that some buildings gracefully change as new tenants change the building for new purposes. And some buildings don't adapt well. He adopted a model called "pace layering" to explain this.



[4]

A building is a set of "shells" that want to change at different rates. If the architecture of the building allows them to change independent of each other, the building can evolve.

A data-centric architecture, done well, has these same properties. The foundation of a data-centric architecture is what Marshall Cline and Mike Girou called "Enduring Business Themes."[5] There are core concepts in enterprises that remain stable over decades or longer. If we base our model on these, we have stability. If we layer on the things that change more rapidly over time, and allow them to change without disrupting the core, we can evolve in place.

Imagine an inventory control system. If we decide that we would like to age inventory in the way that accountants age receivables (not completely fabricated; I had a client do this), an

---

[3] How Buildings Learn, Stuart Brand 1995 https://amzn.to/2vaudlR.

[4] We may want to redraw this picture from https://bit.ly/2V9ABsb.

[5] Enduring Business Themes, Marshall Cline, Mike Girou https://bit.ly/2V7Dwlp.

evolve-able system would be able to implement this without a major data conversion project (it is harder than it sounds).

SPECIALIZE-ABLE

Specialize-ability is the other side of the extensibility coin. Not only do we want the model to evolve over time to keep pace with overall demands, we would like it to be locally different.

With traditional approaches, if one department needs some extra features in an application system, they generally have two choices:

1. They can convince the rest of the firm to accept the changes to incorporate the features, or
2. Build their own system.

Historically there hasn't been a good way to share most of the data and most of the functionality. Object-Oriented design was meant to accomplish this, but in practice it wasn't flexible enough.

Going back to our inventory control example: imagine that one department decided they wanted to store perishable items; produce, for instance. They would like to extend the model to include "good until" dates and perhaps base their cycle counting on this information. But they don't want to re-implement all the ordering, stock keeping, issuing, and receiving functionality.

A system that will allow them to do this is specialize-able. All the core functionality should still work with the produce. We should be able to value all the inventory, regardless.

SINGLE BUT FEDERATED

In our definition, we said there would be a single data model. There are two qualifications to that. The first is that it may be locally extended, as in the previous example. But what makes it data-centric is that they are extensions off of a core rather than new models.

The other qualification is that while there is a single model, all the data need not be in a single database. For most firms this is impractical. Luckily, it is not necessary to get the data-centric approach to work.

The single core model provides a vocabulary to execute queries. To the extent the specialized models are derivative from the core, then a query using the core vocabulary will pick up the

specializations. Again, to our inventory example, if we query for inventory, we will get widgets as well as watermelons, even if we didn't know ahead of time what a watermelon was. We will get the "good until date" of the watermelon, even if we didn't ask for it.

There is a second aspect to the implementation of the data-centric approach, which is the ability to federate a query over many repositories or databases. Traditional relational databases have been tuned in such a way that it is very difficult to execute a single query across more than one database. One of the key aspects of a data-centric architecture is its support of federated queries. We will go into this in more detail in The Data-centric Architecture. Briefly, a federated query is one where you write one query, and it is distributed to multiple databases, each of which solves a portion of the problem, and then the results are recombined.

These federated queries can include databases that were loaded directly into the data-centric model, as well as existing legacy databases that have been mapped to the data-centric model.

ENTERPRISE APP STORE

The data-centric approach makes possible an "enterprise app store." Anyone who has enjoyed the app store (which is to say everyone) might well wonder: "why can't I have this for my enterprise?" The double-barreled short answer is integration and functionality. That is, the end consumer app store "works" because of the low bar on data integration. For the most part there is very little integration between apps, as they are mostly unaware of each other. What little integration exists is done by the user. You might load your contacts into an app, you might build an "IFTTT" (IF This Then That) app to synch up some of your data, but really this is a personal ad hoc bit of integration and not enterprise integration. And the functionality in the app store is pretty lightweight.

You can't really manage inventory, process orders, settle trades, underwrite insurance, adjudicate claims, or any of the many other complex functions that an enterprise routinely performs on a lightweight app store type app.

We believe the application ecosystem of the future will look and act a lot more like the AppStore of today, but there will be two major changes

The first is that in the enterprise app store all the apps will be built to the shared data model. Everything the app knows will come from the data model, and the result of everything the app does will be returned to the shared model. You might have an app that does inventory cycle counting. What it knows about inventory, location, quantity on hand and recent transactions, it gets from the shared model and the data bound to the shared model. The app supplies some

algorithms, perhaps to suggest how often to count which items, or maybe one that suggests counting when the system thinks the item is out of stock. This is both the easiest time to count (either there are zero or you find a few) and it is the time to count when it will make the most difference (the difference between having 11 or 12 items on hand doesn't make a lot of difference, but if you discover you have one item on hand when you thought you had zero, you can continue to sell the item.

The important thing is that the results of the count (that it was taken, who took it, what the count was, as well as any adjustments in quantity on hand if there were any) are put back in the shared data, using the shared concepts, in a way that any other application can take advantage of them.

This is a huge difference. It means that if you find or build a better cycle count app, you can just start using it. No data conversion. No other apps that must change. With current technology, usually if you want better cycle counting you end up implementing another giant monolithic application.

The other advantage is not every department has to use the same cycle counting app. Maybe the department with large outdoor inventory wants their counts optimized around travel time. This environment means that everyone doesn't have to use the exact same application, they merely have to conform to the same model and the same rules.

The other thing that will be different between the enterprise app store and the current consumer app store is the economics.

When the app store was launched, many veterans scoffed at the idea of 99 cent applications. It is now an $80 Billion industry.

The enterprise app store apps will not be 99 cents. In most cases they will have to be bespoke built to the enterprise's core model. So, the universe of buyers will not be large enough for any one to make money at 99 cents.

But we expect that most enterprise app store apps, will be a few pages of code and could be built in a few days. You might have them built by your in-house agile teams, or an industry of firms that are good at making apps and aligning them to your model may spring up. Most enterprises will need several thousand of these apps. If they cost $1,000 each (which is what I think the market would like settle in to) you would have all the application functionality of your firm built for a few million dollars.

More importantly any one of them could be swapped out for $1,000 or so.

We believe that the end game of the data-centric approach will be a viable market for low-cost small-scale bits of functionality, coordinated through a shared model.

INCLUDES ALL TYPES OF DATA

The final thing, at this first cursory level, is how data-centric solves the data polyglot problem. Right now, we have these types of data:

- *Structured data*—data in relational databases.
- *Semi-structured data*—data in XML, json, html, or other structures that are not rigid tabular structures but have some structural clues.
- *Network structured*—social media and other data expressed in graphs.
- *Unstructured data*—documents, notes, memos, and even audio / video.

For the longest time, integrating these types of data was the holy grail: an imaginary ambitious thing that would probably never happen. It is now routinely do-able. In chapters 8 and 9 we will explore the standards that support bringing various types of data into a single representation. For now, be aware that the integration of structure, semi-structured, and unstructured data is a solved problem.

THE ECONOMICS OF THE END GAME

Right now, in large enterprises (in the private sector or government), the cost to get a major system implemented is in the tens to hundreds of millions of dollars. The cost of keeping all the implemented systems stitched together is even greater.

I'm going to ask you to imagine an end state that may sound a bit utopian at the moment, but I will flesh out the details in this and the companion books. In this end state, you have a single core model for all the concepts shared across your enterprise. As we suggested earlier, the model consists of 400 concepts that everyone at your firm understands and agrees with.

The model is extended into a dozen sub-domains that cover in more detail each of the more specific and specialized areas of your business. Each extension adds dozens of more specialized concepts and a host of fine-grained "taxonomic" distinctions.

> Taxonomy: Taxonomies are ways of organizing or structuring information. They are often thought of as being hierarchical, but our experience suggests that there are many useful taxonomies that are flat. The more important distinction for our purposes is that taxonomies

have few complex or custom relationships. Concepts in your core model (e.g., “customer,” “purchase order,” or “treatment plan,”) have many complex relationships to other concepts. The taxonomic distinctions to use to further refine the characterization of your concepts (e.g., “VIP,” “domestic,” or “chronic”) are simple tags. They have few, if any, relations to each other and can be treated as stand-alone tags used to categorize other things.

There is a library of shared functional routines that provides IT functions in the application-specific data model and programming language that, traditionally, are coded over and over again to satisfy the common function. These shared functions do things like:

- Layout fields on a screen
- Detect the change to a data field
- Forward a message, such as an email, in response to a stimulus
- Provide mathematical functions, such as net present value and internal rate of return
- Geospatial calculations, such as area or distance between two points

This library is finite in size. Depending on your industry and requirements, it might be as small as a few hundred functions. Each of these functions is tied to concepts in your core model. For instance, the geospatial functions are tied to place-related concepts. These functions are coded and deployed in a way that is language and operating system independent.

There exists an environment for bringing data and functions together declaratively, that is, there is little need for new code.

In this world, we will have forgotten why we even needed the monolithic application. As we contemplate the need for new functionality or new data structures, we will just add them. My observation is that the impetus for most new application projects is that there is a small number of functional deficits with the current systems, and these deficits are deemed to be too hard to implement in the existing system.

In this new world, the new requirements will be added to the ecosystem incrementally. We will cease thinking of application projects.

The economics of this are staggering. High risk multi-million-dollar applications will be replaced with small incremental improvements. Integration projects, which require small armies to maintain, will be replaced with information that holds its own integration.

Summary of what is data-centric

There is a whole new world opening up for sponsors of enterprise information systems. By flipping the prevailing paradigm on its head, we can see how enterprise information could be differently harvested, organized, and consumed.

This change is not a technological change. It is a change about how we choose to organize, describe, and deploy information systems. This new way of envisioning systems puts data at the center. Applications (think app store-sized applications) will come and go. They will conform to the data structure that exists rather than impose their own. Everything they do will be reduced to data and be self-describable, and in such a way that if the application were to vanish tomorrow, the data would be sufficient to carry on.

This arrangement has many fortuitous benefits, one of which is the ability to not only integrate all the structured data that exists in a firm, but also to take this to the semi-structured and unstructured data of the firm.

This is the vision. What comes next is how to get there and some examples of firms that have already achieved it.