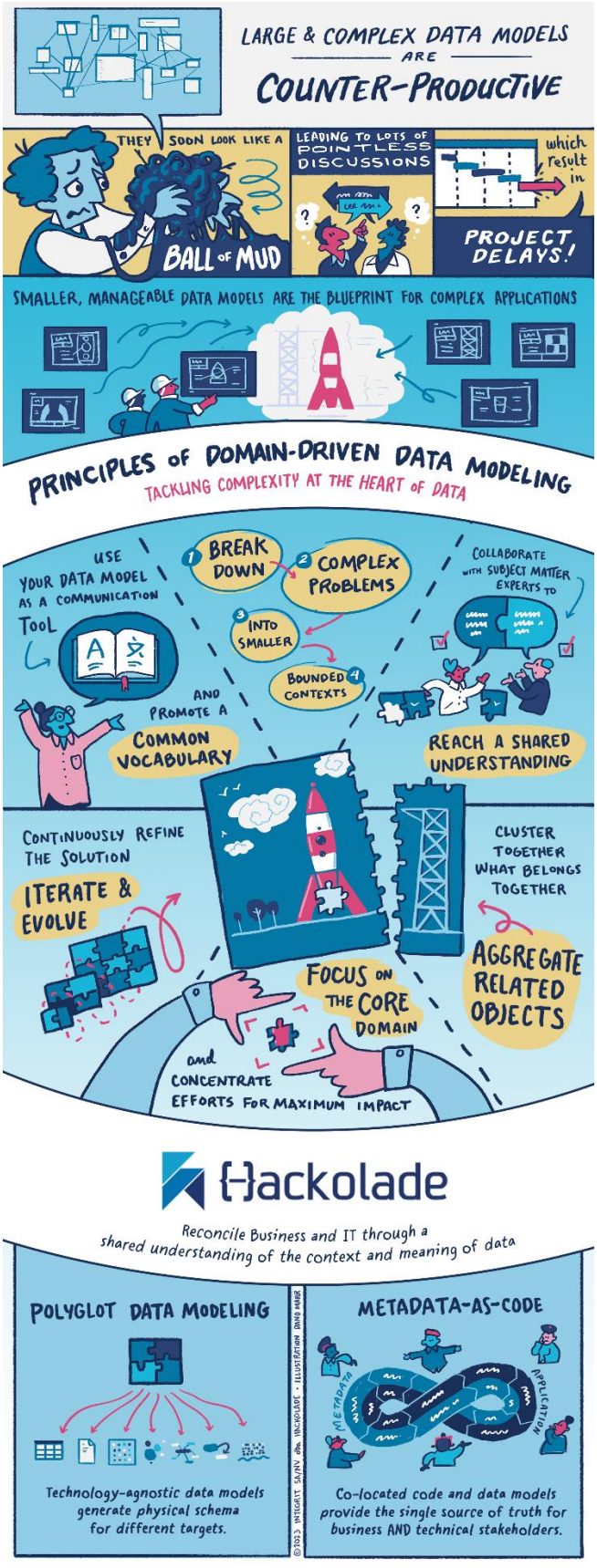


Hackolade

Principles of Domain-Driven Data Modeling

Tackling complexity at the heart of your data

Table of Contents

(This eBook is greatly inspired by the principles of Domain-Driven Design, the Agile Manifesto, and The Lean Startup)

Introduction

There is quite a bit of debate, to say the least, about the industry appeal for Data Modeling in general, and Enterprise Data Models in particular. Some say that data modeling is dead. Some say that it is on life support. At the same time, everyone in data realizes that some sort of data modeling (or at least schema design ahead of writing code) is a critical success factor for a successful software development project. These data modelers would argue that it is even more important these days than it was before.

Our take at Hackolade is that ***traditional*** data modeling must evolve to remain relevant. There are a number of reasons why traditional data modeling got a bad reputation. If data modeling takes too long, gets in the way of getting things done, adds more complexity than needed, or is done for the sake of data modeling alone (instead of for the purpose of achieving the delivery of great software), then no wonder that some people will try to avoid it.

And they would be right, except that avoiding it all together goes to the other extreme, and that doesn't get us to a good place either. The authors of the Agile Manifesto were right on target when they explained: "We want to restore a balance. We embrace modeling, but not in order to file some diagram in a dusty corporate repository. We embrace documentation, but not hundreds of pages of never-maintained and rarely-used tomes."

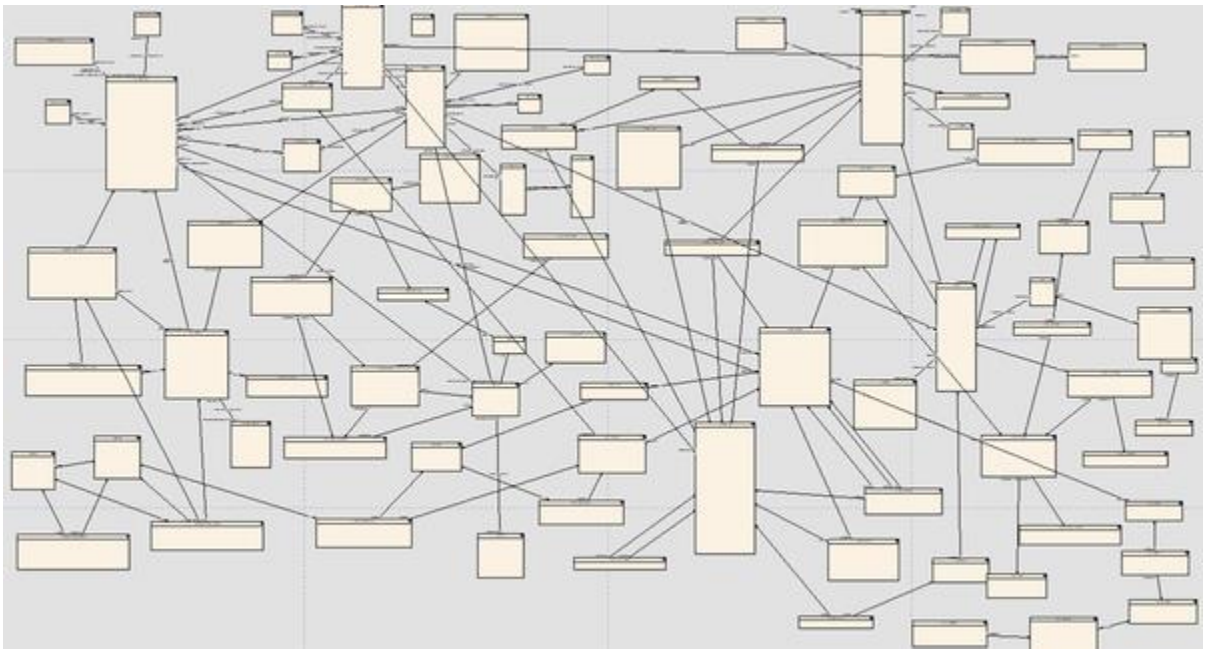
At Hackolade, we think that there's a pragmatic, modern way to perform just enough data modeling to reach the strategic objectives of organizations, and not too much that it would become counter-productive. Such balance can be achieved using our next-gen Hackolade Studio data modeling tool, and using an approach that we call "Domain-Driven Data Modeling".

Complex data models are counter-productive

A “code-first” approach refers to the practice of developers to immediately start writing code as soon as an idea is expressed. It may make sense for quick-and-dirty prototyping, but it does not work for applications with any kind of complexity. It greatly lowers time-to-market and total cost-of-ownership to think and design first, before coding.

At the other extreme, some organizations can get paralyzed by over-thinking and creating monolithic Enterprise Data Models that slow down execution.

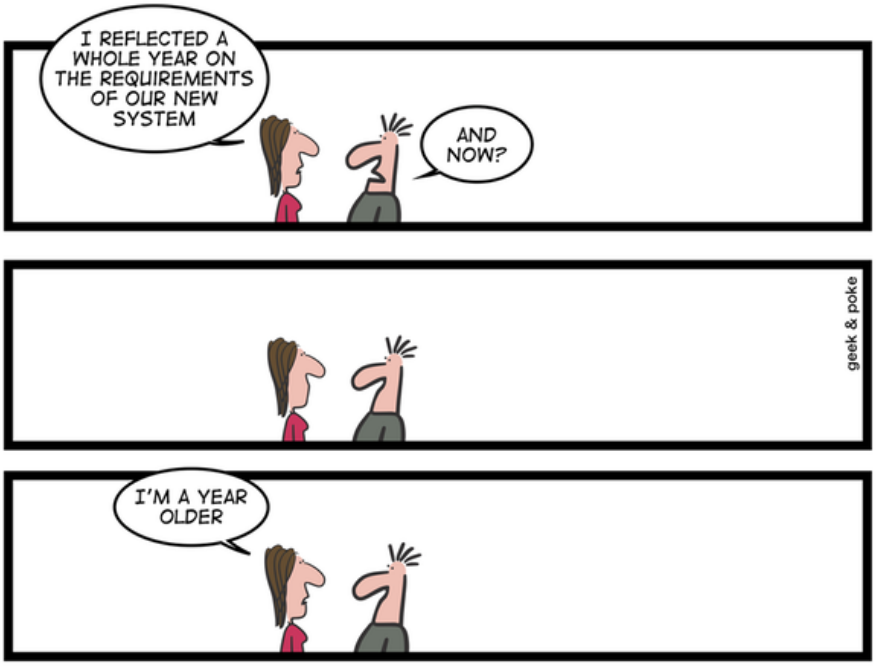
They say that a picture is worth a thousand words. That's only true when the picture is an abstraction of reality that actually facilitates understanding...



Enterprise Data Model

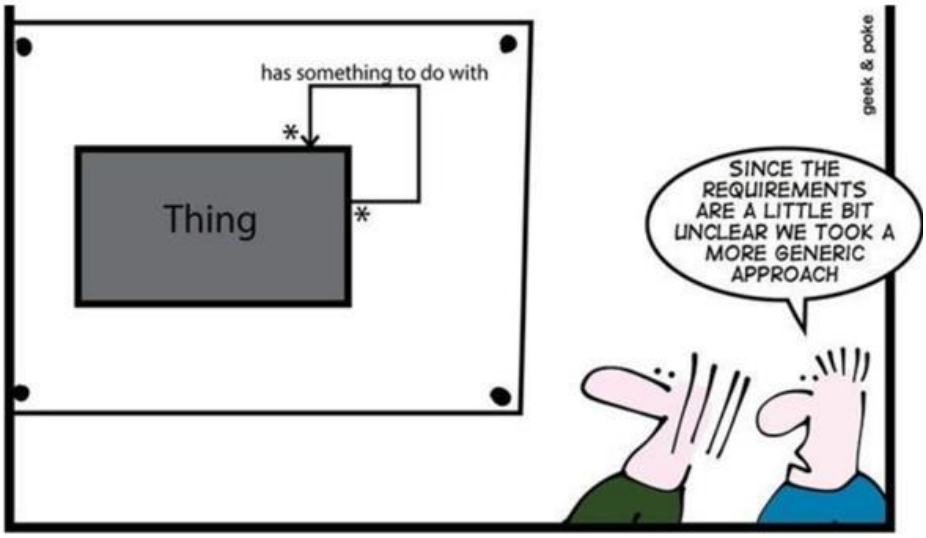
The above data model would clearly not achieve the clarification objective... The authors may think that it makes them look smart, but it is actually counter-productive! Business users won't understand it. Technical users (including developers) will wonder how that's ever going to solve their challenges. And frankly, even data modelers will be challenged to understand a model with hundreds of entities.

Enterprise Data Modeling initiatives often fail to deliver on time, if at all. By the time the EDM is completed, it is already obsolete.



Business Requirements

Then there's the issue of endless semantic discussions about whether to use the term customer, client, person, party, or whatever abstraction is supposed to be generic enough that it fails to describe the actual "thing" in terms that the stakeholders actually understand.



Abstraction

Abstractions are good… to a point. Until they become counter-productive. It is hard enough to not lose anything in the translation of business requirements into technical specifications, so it is critical to use the terms that are familiar to business users when conducting the business, so the meaning is intuitively obvious to the casual observer.

Some will argue that there might be an issue down the road, or that another division in the company might be using a different term for the same thing. Let's use common sense here, and remain pragmatic. Applications evolve over time, so terms can evolve too. By wanting to make things perfectly abstract from the get-go, there's only one thing that can occur: delays in delivering the project, with no guarantees of a better result.

Finally, one has to wonder why a domain expert of an organization's core business would start their work with an industry data model?

Industry Data Models

Maybe an industry data model is useful for someone who's new to an industry and wants an overview of unfamiliar concepts. Maybe there are some descriptions that could serve as a basis for further discussions. Maybe it could help validate that nothing has been overlooked. Maybe the models provides information about non-core aspects of the business.

That can very well be true, but surely, an in-house domain expert knows more about the core business than the industry data model might reveal, right?

Domain-Driven Design

Eric Evans is the author of the book "Domain-Driven Design: Tackling Complexity in the Heart of Software" published in 2003, which is considered one of the most influential works on Domain-Driven Design (DDD). In a nutshell, its principles include:

- bounded context: managing the complexity of the system by breaking it down into smaller, more manageable pieces. This is done by defining a boundary around each specific domain of the software system. Each bounded context has its own model and language that is appropriate for that context.
- domain model: using a conceptual model of the domain that represents the important entities, their relationships, and the behaviors of the domain.
- ubiquitous language: establishing a common vocabulary used by all stakeholders of a project, and reflecting the concepts and terms that are relevant to the business.
- context mapping: defining and managing the interactions and relationships between different bounded contexts to ensure consistency between the different models and facilitate communication between teams.
- aggregates: identifying clusters of related objects, and treating each of them as a single unit of change.
- continuous refinement: an iterative process with continuous refinement of the domain model as new insights and requirements are discovered. The domain model should evolve and improve over time based on feedback from stakeholders and users.

These principles are striking by their common sense when dealing with application development, and are also applicable to enhance data modeling. Some data modeling traditionalists have expressed reservations about DDD. For every methodology and technology, there are of course examples of misinterpretations and misguided efforts. But applied with clairvoyance and experience, they lead to great success.

Domain-Driven Data Modeling

We see DDD principles as directly applicable to data modeling to further enhance its relevance, rather than as an alternative methodology.

With Domain-Driven Data Modeling (DDDM) the main principles are:

- focus on your core ("domain and sub-domains")
- break down complex problems into smaller ones ("bounded context")
- use data modeling as a communication tool ("ubiquitous language")
- keep together what belongs together ("aggregates")
- reach a shared understanding between business and tech ("collaboration of domain experts and developers")
- iterate and evolve ("continuous refinement")

Domain-Driven Data Modeling Principles

Focus on the core domain

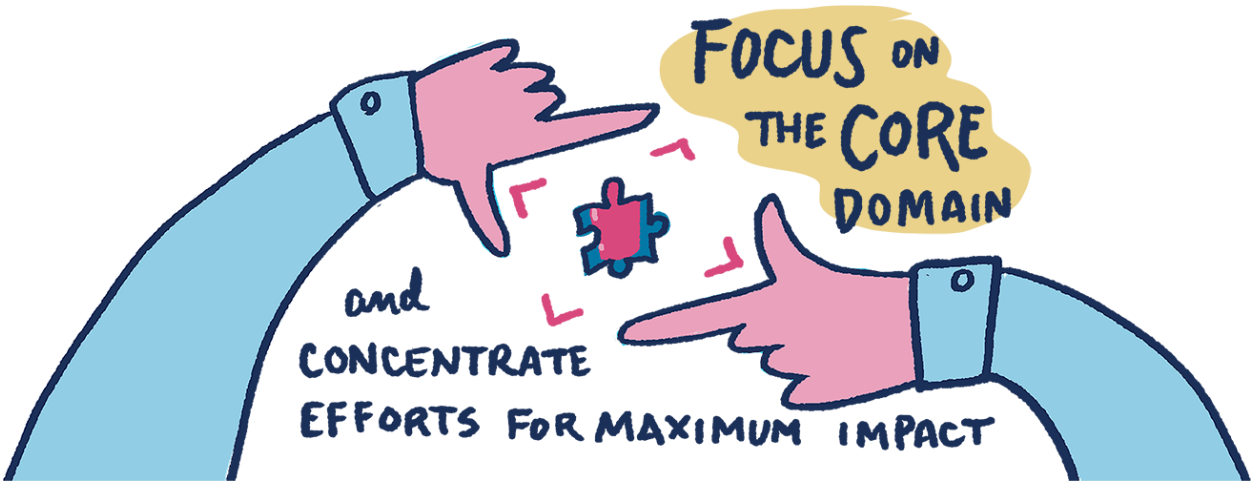
The term "domain" may itself be subject to interpretation. The domain is defined here as a sphere of knowledge or activity that a software system is designed to support, and that its corresponding data model is supposed to represent. It denotes the business problem or opportunity that the data model is intended to address. The domain is the core of that business problem or opportunity, and it defines the key concepts, behaviors, and relationships that are relevant in that specific context.

Defining the domain helps ensure that the software system is designed around the needs of the business and that the models and language used in the system are closely aligned with the business problem. By focusing on the domain, developers can create software systems that are easier to understand, maintain, and extend over time.

A subdomain is a part of the overall domain that represents a distinct area of the business problem or opportunity. Subdomains are used to break down complex problems into smaller, more manageable parts, and to help organize the development of the software system around the business problem.

There are three types of subdomains in DDD:

1. **Core** subdomains: represent the most important parts of the business problem or opportunity. They are the areas of the business that provide the most competitive advantage or differentiation, and are the focus of most of the effort.
2. **Supporting** subdomains: represent the parts of the business that are necessary for the core subdomains to function, but that are not themselves the main focus of the project. Supporting subdomains may include things like billing, customer management, or reporting, for which there probably exist conformed dimensions in the organization.
3. **Generic** subdomains: represent parts of the business that are common across many different industries, and that are not specific to the business problem or opportunity being addressed. Examples of generic subdomains might include things like definitions of time, address, and other common dimensions.



The main focus must be on core subdomains, while supporting subdomains can be delegated, and generic subdomains can often be outsourced. Each subdomain has its own models, language, and bounded contexts, and may be developed by a separate team within the development organization. By breaking down the domain into smaller, more manageable parts, you can reduce complexity and increase the effectiveness of the effort.

Break down complex problems into smaller ones ("bounded context")

In DDD, Eric Evans talks about the concept of a "big ball of mud", which refers to a monolithic system that has become overly complex, difficult to understand, and resistant to change, all things that can occur when there is no clear architecture or design guiding the development process.

The same seems to apply to data modeling. Enterprise Data Models are designed to provide a comprehensive and standardized view of an organization's data assets and how they relate to one another. As a result they run the risk of being too complex, hard to maintain, and hard for stakeholders to understand. It can be challenging to establish clear ownership and governance of an EDM, especially in larger organizations with distributed teams and data sources.

It may be tempting to try to do a monolithic EDM for the entire enterprise. It sounds like a big challenge, and particularly if it is a prerequisite, it surely slows down the delivery of a small enhancement. Politics start playing a role which does not help anyone.

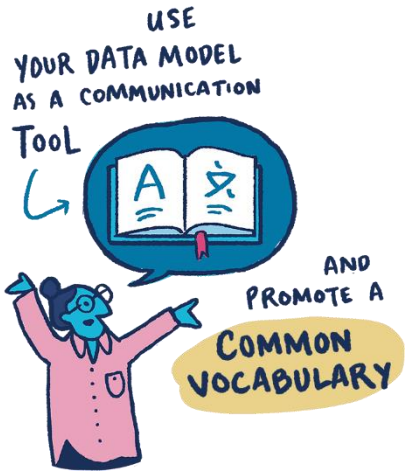
A "bounded context" defines the boundaries of a subdomain within which the data model applies. It is applicable to Data Modeling in the sense that it is useful to break down complex EDMs into smaller, more manageable models that correspond to each core subdomain. This simplifies the modeling process and makes it easier to develop and maintain models that are closely aligned with the business domain.

This will avoid the "big ball of mud" syndrome for data modeling. You will be able to deliver early and often as the project evolves, and iterate before moving to the next one. Inevitably, new discoveries along the way might trigger changes in earlier models -- that's OK, we're agile, and we understand that nothing is set in stone forever. At least things get done in the meantime.

Note that we're not suggesting that you should ignore relevant context or overall understanding of the whole company. We're only saying that we all know what happens when scope grows: delays. If you're doing it for the right reasons, that's great! But let's make sure that it is critical to the core effort.

Nobody is suggesting that the definition of a bounded context should be so strict that you'll create a system that makes no sense. Use your best collective judgement to correctly define the scope.

Data Modeling is a communication tool ("ubiquitous language")



As in all aspects of life, communication is critical to avoid misunderstanding. One of the purposes of data modeling is to create a shared understanding of the business problem or opportunity through its abstraction and representation in the form of an Entity-Relationship Diagram (ERD).

Sounds great, in theory. But misunderstandings can easily appear. Developers often prefer vocabulary relating to technical solutions and architectural implementations. Data modelers may be more inclined to use abstract and conceptual terms. Domain experts and business users may prefer to use language that is more familiar and understandable to them. Misalignment of these approaches slows down the modeling process and results in a model that could fail to meet the needs of the business.

Similarly, there can be misunderstandings between developers and data modelers when it comes to defining requirements and designing data structures. Developers may feel that data modelers are too focused on theoretical considerations and not enough on practical implementation, while data modelers may feel that developers are not taking the long-term needs of the organization into consideration.

Developing software, and managing data more generally, is a people business. Emotional intelligence, empathy, open-mindedness, and experience on all sides can only help. But it often all comes down to vocabulary. Speaking a common, ubiquitous language helps people work together.

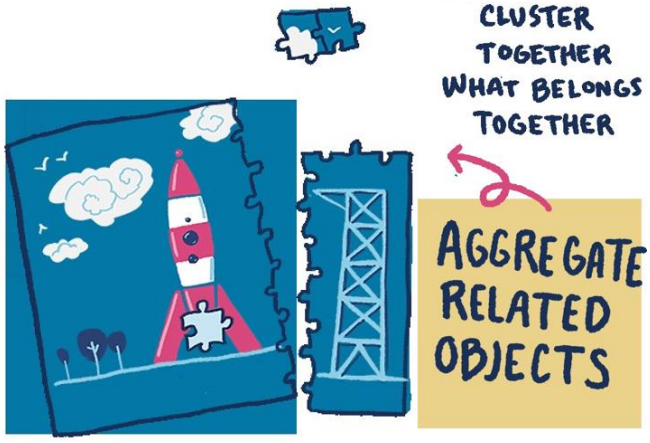
Overall, effective communication and collaboration between the different stakeholders (data modelers, subject matter experts, business users, and developers) is key to creating a data model that is accurate, relevant, and meets the needs of the organization.

The exercise should involve using language and concepts that are more accessible to the business users, as well as providing clear explanations and examples of more technical terms. It can also be helpful to establish clear guidelines and standards for

the language and terminology used in the data model. This helps ensure that everyone involved in the modeling process is using consistent and clear language, and can minimize confusion and semantic arguments.

It also means that developers should use, ubiquitously, the same vocabulary in the application code, database entities and attributes, and in APIs. It makes it so much easier later on to debug issues and discuss enhancements.

Keep together what belongs together ("aggregates")



With traditional logical data modeling, the aim is to create a technology-agnostic model documenting the business rules. All that works well enough using the famous rules of database normalization -- as long as your target technologies are relational only.

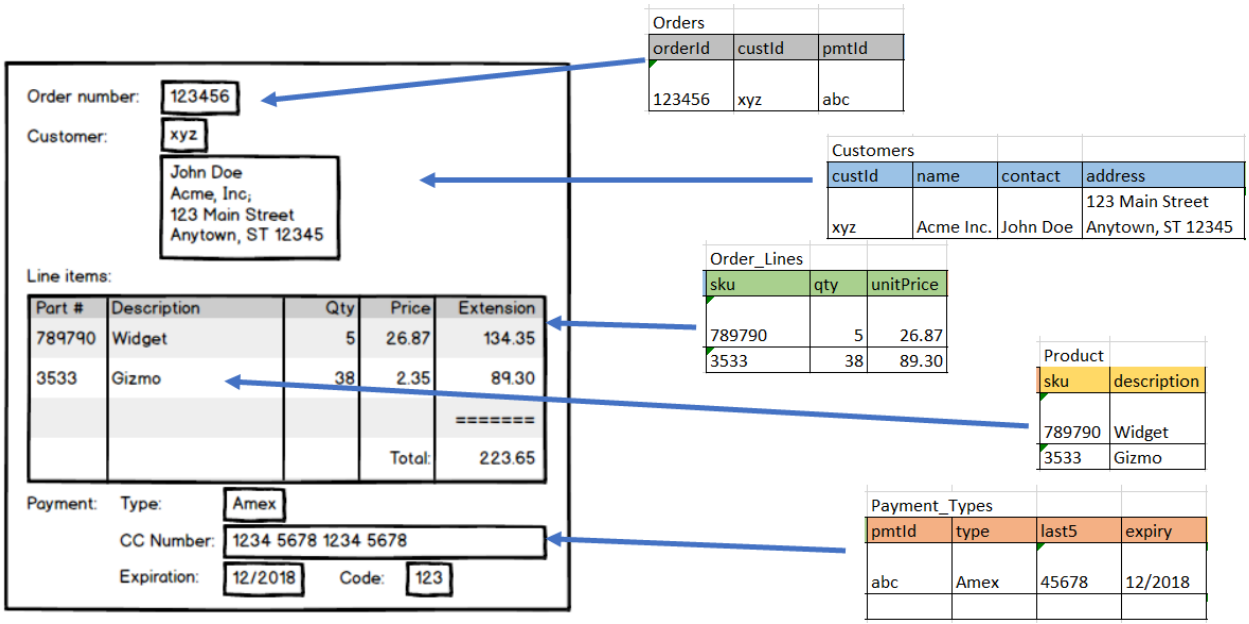
But as soon as we're having to deal with NoSQL or with APIs and modern storage formats, it turns out that logical models which respect third normal form for many-to-many relationships with junction tables are not "technology-agnostic" after all. In a NoSQL document database, you would embed information and not use any junction table. Similarly, supertypes/subtypes can be handled in NoSQL via polymorphism instead of inheritance tables, etc.

In DDDM with aggregates, we make a conscious decision to identify clusters of related objects and treat each of them as a single unit of change. With aggregates it is easier to enforce consistency and integrity within a domain.

And as it turns out, this is beneficial also on the front of object-oriented programming and efficient use of NoSQL technologies. Let's explore this.

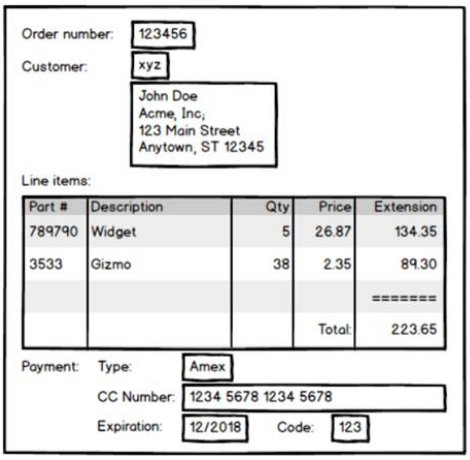
Object impedance mismatch

Impedance mismatch is a fancy term used to describe the problem that arises when the way data is represented in a database or storage system does not match the way it is used in the application code. In other words, when it is normalized, and thus results in inefficiencies and complexities in the application code when retrieving or manipulating data.



Normalization

In contrast, NoSQL document databases allow for more flexible data models, including nested objects and arrays, which can better match the needs of some applications, and the life of developers.




```
{
    "orderId": 123456,
    "customer": {
        "custId": "xyz",
        "name": "Acme Inc.",
        "contact": "John Doe",
        "address": "123 Main Street, Anytown, ST 12345"
    },
    "line_items": [
        { "sku": "789790", "description": "Widget", "qty": 5, "unitPrice": 26.87 },
        { "sku": "3533", "description": "Gizmo", "qty": 38, "unitPrice": 89.30 }
    ],
    "payment": { "type": "Amex", "last5": "45678", "expiry": "12/2018" }
}
```

Denormalization

In DDD, aggregates are a way of grouping related objects and business logic together into cohesive units. Using aggregates can help solve the impedance mismatch problem by providing a way to map complex and dynamic domain objects to more flexible database schemas, particularly in the context of NoSQL document databases with nested objects.

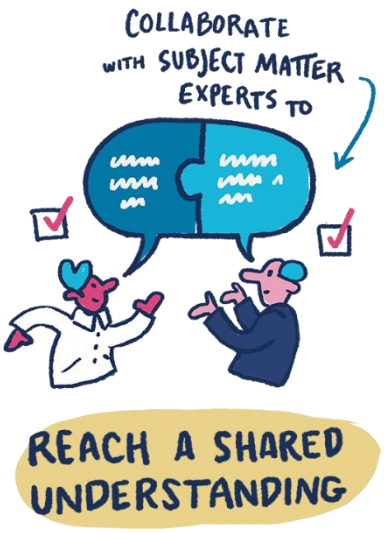
Query-driven data modeling

Query-driven database design is an approach to database schema design that focuses on optimizing database queries and data retrieval for a specific application use case, rather than on an application-agnostic structure of the underlying data.

Aggregates can help with query-driven database design by providing a way to group related data together in a way that supports efficient queries and data retrieval. By grouping related data into aggregates, we can reduce the number of database queries needed to retrieve data for a particular operation, and improve the performance of the application.

By using aggregates, we can improve the performance and maintainability of the application, while also ensuring that the data model is closely aligned with the application's domain. And making life simpler for developers by eliminating the impedance mismatch discussed above.

Reach a shared understanding



If tech and business don't share the same understanding on the meaning and context of data, organizations are exposed to great risks of misguided business decisions, poor data quality, inaccurate or incomplete data, inefficient processes, and missed opportunities for innovation and growth. The lack of collaboration will also be a source of frustrations, delays, missed deadlines and budget targets, and more undesirable consequences. No fun.

The main challenge in achieving a shared understanding on data is the difference in language, knowledge, and priorities. Business stakeholders may not have a deep understanding of the technical aspects of data, while tech stakeholders may not fully appreciate the business context and objectives.

Modern data modeling is a valuable tool to mitigate these risks and challenges, by providing a common language, clarity, visualization, an iterative process, and validation. As long as it is done with pragmatism and applying the principles detailed above, an Entity-Relationship Diagram can serve as an enabler for collaboration and engage in a fruitful dialog between analysts, architects, designers, developers, and DBAs.

A data model describes the business. A data model is the blueprint of the application. Such a map helps evaluate design options beforehand, think through the implications of different alternatives, and recognize potential hurdles before committing sizable amounts of development effort. A data model helps plan ahead, in order to minimize later rework. It helps ensure that all stakeholders have a clear and accurate understanding of the data requirements and can work together to build effective software systems

This alignment between business and tech increases efficiency, improves decision-making, and reaches better outcomes for the business.

Iterate and evolve

The adage "Rome wasn't built in a day" attests to the need for time to create remarkable things. We could add that they did not start with a single, complete blueprint before laying the first brick.

Deep insights and breakthroughs only happen after living with the problem through many iterations.

The same way that software development is being built incrementally nowadays, rather than in big bulky deliveries, it makes sense to approach data modeling in a mindset of continuous evolutions. Each iteration builds on the previous one. Each iteration typically involves a small set of improvements, which are validated and refined before being incorporated into the next iteration. Each iteration is in sync with the application development evolutions and follows the same lifecycle.

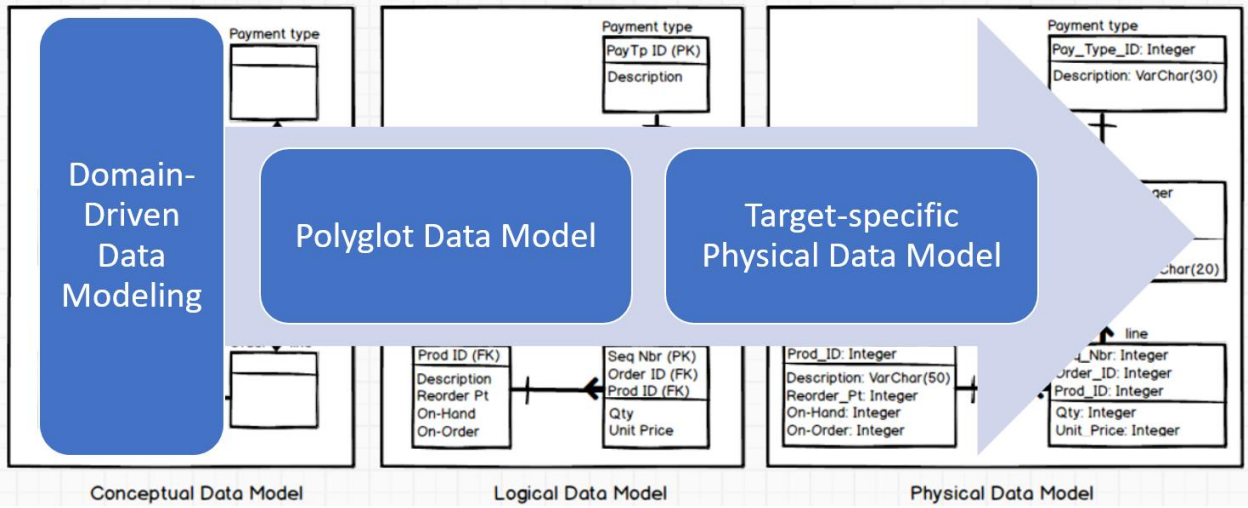
Lifecycle of Domain-Driven Data Modeling

We just covered the principles of DDDM and how they differ from the traditional data modeling approach used for decades.

Domain-Driven Data Modeling Principles

They should now be orchestrated and deployed using the following steps:

- start with requirements,
- define the bounded context of the domain,
- develop a high-level model while assembling the shared vocabulary,
- design screens and reports, draw business logic flowcharts
- analyze and quantify workloads and access patterns
- create a polyglot data model
- derive target-specific physical models
- generate schemas
- integrate with lifecycle of application changes through metadata-as-code
- publish to data catalogs
- iterate and evolve



Domain-Driven Data Modeling Lifecycle

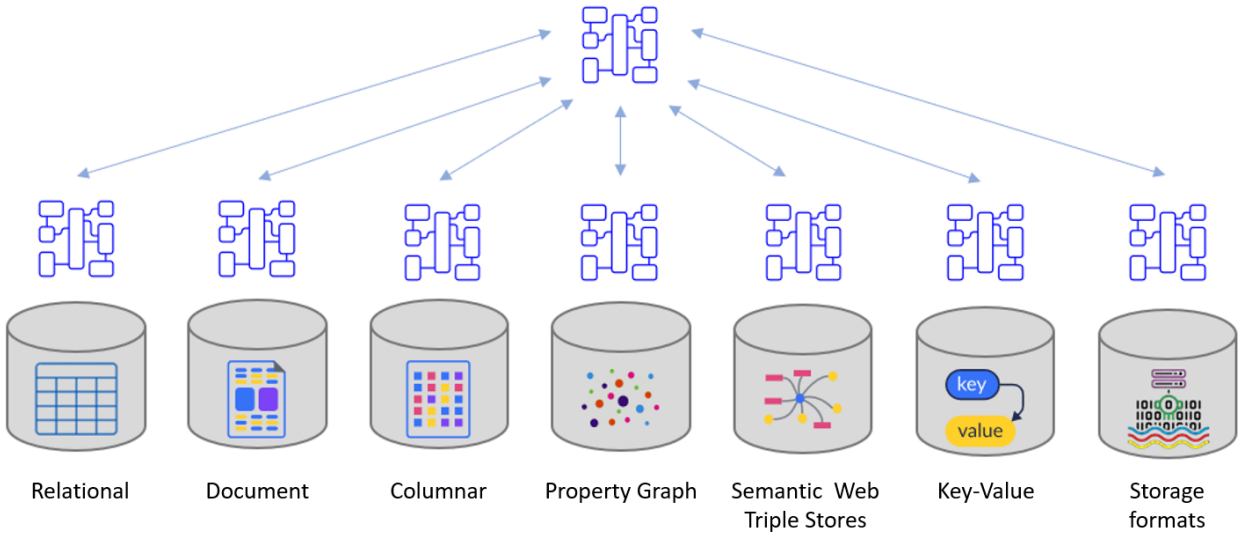
The 2 steps you might be less familiar with are those involving Polyglot Data Modeling and Metadata-as-Code, as they're specific to using Hackolade Studio.

Polyglot Data Modeling

We highlighted earlier how traditional logical data models have characteristics that are too constraining for the modern data stack: they're not as technology-agnostic as originally thought, they can only be normalized, and they can only use scalar data types.

In contrast, Hackolade Studio's Polyglot Data Models may be denormalized and polymorphic, may use complex data types, and are truly technology-agnostic. They also can display a graph view to render a conceptual model that is friendly to business users.

A polyglot model allows you to define data structures once, then instantiate them in any of the physical targets supported by the tool:



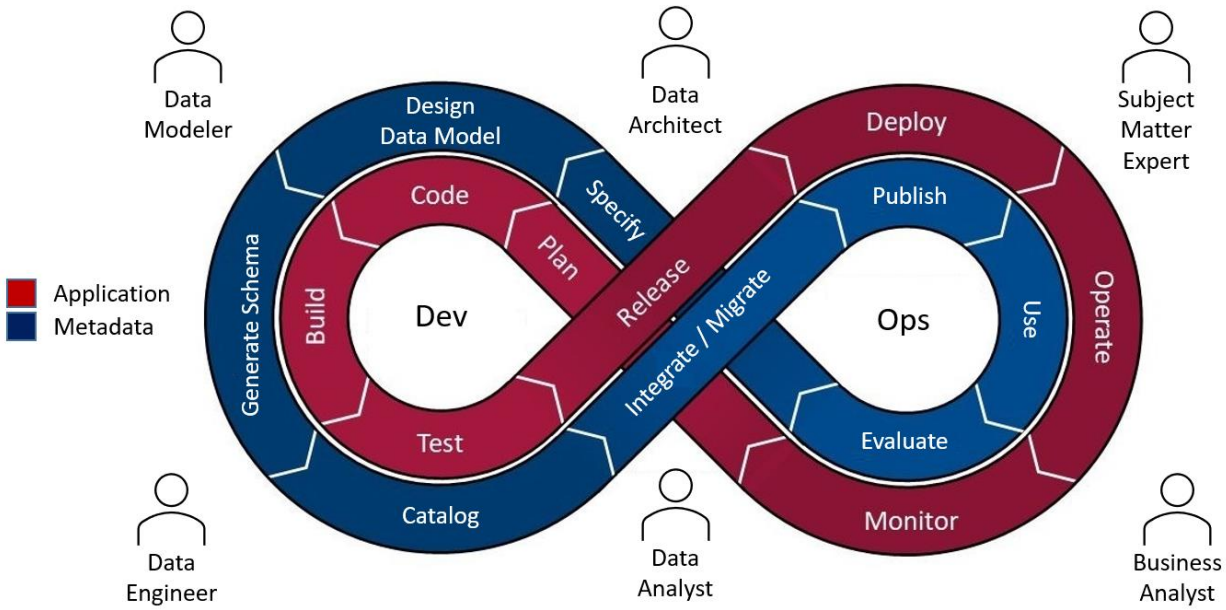
Polyglot Data Modeling

Metadata-as-Code

Hackolade Studio provides a native integration to Git repositories and platforms such as GitHub, GitLab, Bitbucket and Azure DevOps Repos. The primary benefits include collaboration, versioning, branching, conflict resolution, peer review workflows, change tracking and traceability.

But the additional advantage is in the fact that we use Git as a repository technology instead of proprietary servers provided by traditional data modeling tools. These servers require costly licenses, infrastructure, installation, maintenance, backups, upgrades, etc…

Furthermore, we're now able to co-locate data models and their schema artifacts with application code, so they can follow the lifecycle of application changes.



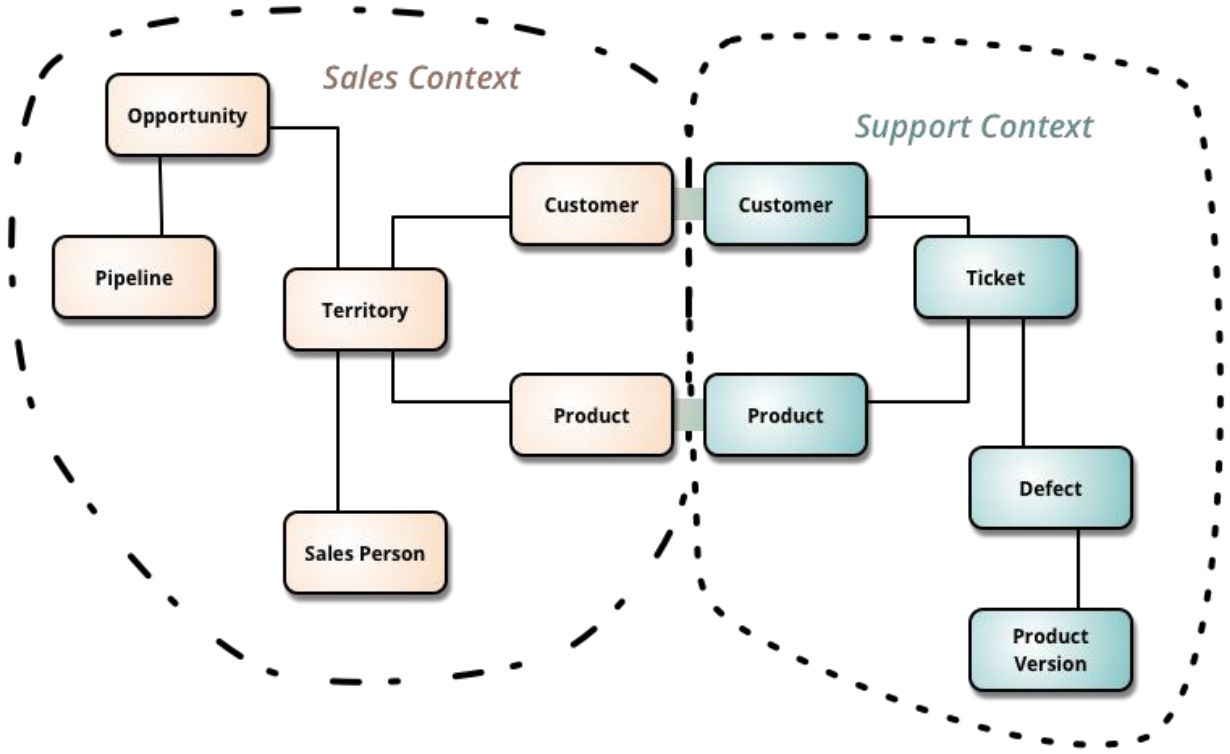
Metadata-as-Code Infinite Loop

And last but not least, our Metadata-as-Code approach provides a single source-of-truth for business and technical stakeholders as the data model in the Git repo is the source of the technical schemas used by applications, databases, and APIs at the same time as it is the source for the business-facing data dictionaries. This architecture contributes to the shared understanding of data by all the stakeholders.

Frequently Asked Questions

What if a concept is shared across 2 or more bounded contexts?

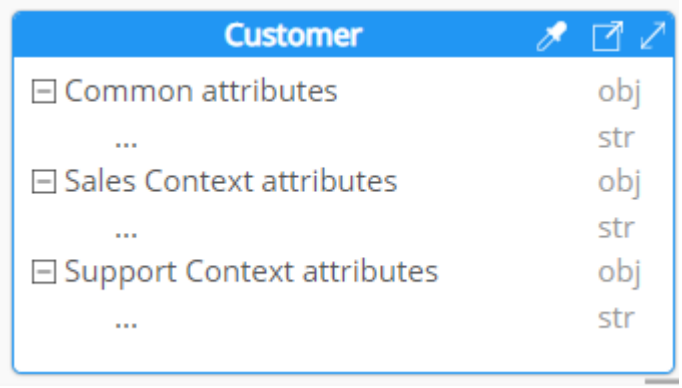
In his article on the subject, Martin Fowler created this useful image to describe the situation:



Bounded contexts

Inevitably, some concepts sit on the boundary and are shared between bounded contexts. Should you still have a single data model per bounded context, or are you then forced to have a single model covering multiple bounded contexts?

The answer is very clear, from our perspective. Having a single model covering multiple bounded contexts would defeat the purpose of having bounded contexts! You should therefore continue to have a single model per bounded context. And each model should reference an external definition of each common concept. In other words using the example above, you should have a definition of the Customer concept. The Customer definition contains a set of common attributes, a set of attributes related only to the Sales Context, and another set of attributes related only to the support context. With JSON, it is easy to group attributes:



Customer concept definition

It is easy, in the Sales Context model, to make an external reference to the Customer definition, and pick the relevant attributes.

What is the best way to map DDDM terms with Hackolade Studio concepts?

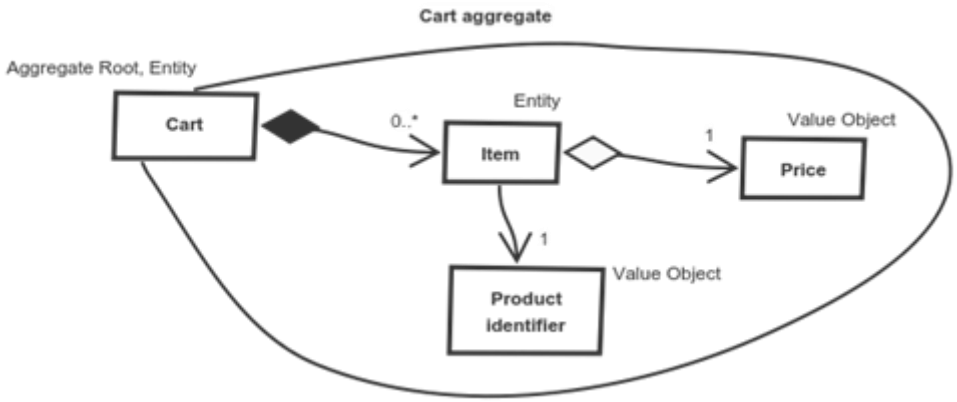
Vocabulary originally introduced by DDD matches directly with concepts in Hackolade Studio polyglot and physical models.

Let's review DDD definitions:

- **Domain:** the problem to be addressed with a software effort. It is divided into subdomains:
  - Core domain
  - Supporting subdomain(s)
  - Generic subdomain(s)
- **Bounded context:** a segment of the solution (see image above) that requires:
  - its own ubiquitous language, and where the ubiquitous language is consistent
  - its own architecture
  - a specific responsibility enforced with explicit boundaries
  - ideally covering no more than one subdomain
- **Aggregate:** represents a cohesive unit of related objects that can be treated together for the purpose of data changes. The aggregate ensures that all operations and business rules are applied consistently, and provides a clear boundary for transactional consistency. It consists of three main components:
  - a single root: serves as the entry point and the main point of reference for the aggregate. It is responsible for maintaining the identity of the aggregate as a whole.
  - entities: objects that have their own unique identity and lifecycle within the aggregate. They represent concepts or entities that have distinct attributes and behavior. Entities are typically mutable and can be modified or updated over time.
  - value objects: objects that do not have their own identity but are defined by their attributes or values

It's important to note that while the root entity holds the identity of the aggregate, other entities within the aggregate may have their own local identities relevant only within the scope of the aggregate. These local identities help to establish relationships and maintain consistency within the aggregate.

In the book Implementing Domain-Driven Design, the author Vaugh Vernon shows this diagram to illustrate aggregates:



DDD cart aggregate

In Hackolade Studio, the following hierarchy exists:

- Model: an abstraction using an Entity-Relationship Diagram or a graph to describe and document the system of an organization. Stored as a physical file.
    - Container: generic term which translates, depending on the physical target, to: database, schema, bucket, keyspace, or namespace, etc.
        - Entity: generic term which translates, depending on the physical target, to: table, collection, record, node label, vertex label, class, subclass, request or response resource, etc.
            - Attribute: generic term which translates, depending on the physical target, to: column, field, subject, array; map, list, etc.
        - Relationship: generic term which translates, depending on the physical target, to: foreign-key relationship, reference, edge label, etc.

Each object above, no matter its level in the hierarchy, has properties that are displayed in the Properties Pane.

These DDD terms map to Hackolade Studio concepts:

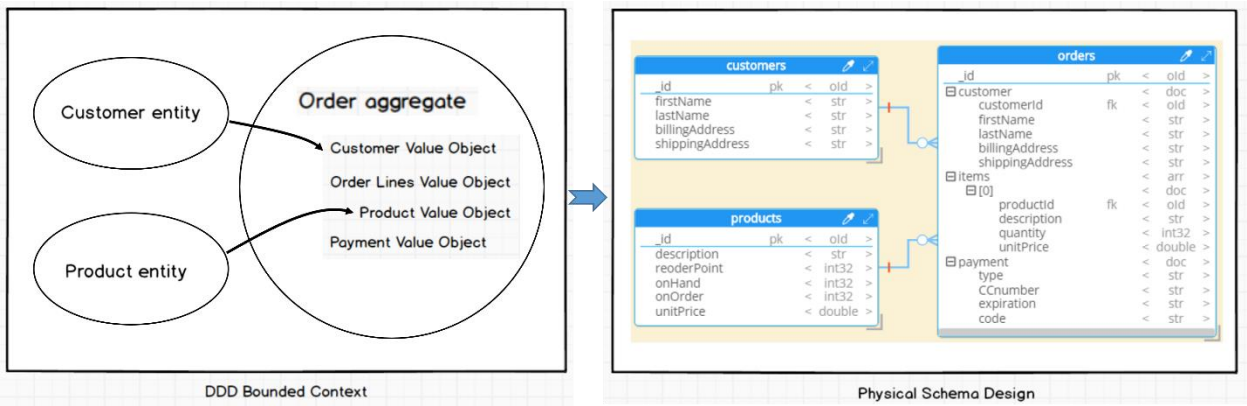
- Should a Hackolade model represent an entire domain, a subdomain, or a bounded context? It depends on complexity, scale, and implementation. Remember that with DDDM, just like with DDD, we're trying to break down complex problems into smaller ones. We're also trying to avoid large enterprise models. Often a solution is to have one model per bounded context, and to have a physical implementation per bounded context, as is often seen with micro-services and event-driven architectures. Remember also that a physical model in Hackolade

Studio can be composed of multiple subsets of different polyglot models.

But let's be flexible and open-minded to pragmatic solutions. If it makes sense to have multiple bounded contexts in the same model, Hackolade Studio will accommodate that. If the model then grows to an unreasonable size, it might be time to split into separate models.

- An aggregate maps directly to a Hackolade Studio entity in a polyglot model.
- DDD entities maps to subobjects in aggregates (called entities in DDD vocabulary) which remain as such in target technologies that support complex data types. For target technologies that only support scalar data types, like RDBMS and SQL-like analytics, aggregates are normalized when derived from polyglot. In other words, entities that are part of aggregates get normalized into separate tables.
- An identity maps directly to a primary key at the root level, and to a foreign key relationship in subobjects.
- Value objects map to attributes.

Here is an illustration of how multiple access patterns for an ordering systems result in a combination of a denormalized entity (orders) and normalized entities (customers and products)



DDD order aggregate mapping

Takeaways

If you read all the way to this point, you might feel that there's a lot to swallow. But in effect, much of it are just subtle nuances of existing methods, fine-tuned to accommodate the modern 21st century data stack, and simplified to eliminate heavy counter-productive practices in favor of a lean and pragmatic approach focused on the nimble delivery of quality software and data that will benefit many stakeholders.

Of course, such an approach will greatly benefit from its own modern toolset. Stakeholders can be better served by a next-gen data modeling tool built from the ground up to support them. Hackolade Studio will allow you to be as rigorous or as reactive as your organization chooses to be. You may follow the Domain Driven Data Modeling process from A thru Z. Or you may start directly with a target-specific physical schema, then iterate. That's up to you.

In practice, many organizations use agile data modeling methodologies that prioritize flexibility and collaboration over creating a fixed and complete model. This approach involves working closely with stakeholders to identify the most critical data assets and relationships and iteratively refining the model over time based on new insights and changing business needs. At Hackolade, we support customers in that journey, and assist with our Domain-Driven Data Modeling process, leveraging Polyglot data modeling outcomes and target-specific physical data models. Such models are considered to be metadata that can be treated as code.