Agile Data

Data Modeling 101

@scottwambler

Home Roles Practices Road maps Resources #AgileDB Contact us

Search

The goals of this article are to overview fundamental data modeling skills that all developers should have, skills that can be applied on both traditional projects that take a serial approach to agile projects that take an evolutionary approach. My personal philosophy is that every IT professional should have a basic understanding of data modeling. They don't need to be experts at data modeling, but they should be prepared to be involved in the creation of such a model, be able to read an existing data model, understand when and when not to create a data model, and appreciate fundamental data design techniques. This article is a brief introduction to these skills. The primary audience for this article is application developers who need to gain an understanding of some of the critical activities performed by an Agile DBA. This understanding should lead to an appreciation of what Agile DBAs do and why they do them, and it should help to bridge the communication gap between these two roles.

Table of Contents

1. What is Data Modeling?

Data modeling is the act of exploring data-oriented structures. Like other modeling artifacts data models can be used for a variety of purposes, from high-level conceptual models to physical data models. From the point of view of an object-oriented developer data modeling is conceptually similar to class modeling. With data modeling you identify entity types whereas with class modeling you identify classes. Data attributes are assigned to entity types just as you would assign attributes and operations to classes. There are associations between entities, similar to the associations between classes – relationships, inheritance, composition, and aggregation are all applicable concepts in data modeling.

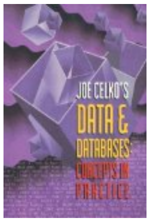
Traditional data modeling is different from class modeling because it focuses solely on data – class models allow you to explore both the behavior and data aspects of your domain, with a data model you can only explore data issues. Because of this focus data modelers have a tendency to be much better at getting the data "right" than object modelers. However, some people will model database methods (stored procedures, stored functions, and triggers) when they are physical data modeling. It depends on the situation of course, but I personally think that this is a good idea and promote the concept in my UML data modeling profile (more on this later).

Although the focus of this article is data modeling, there are often alternatives to data-oriented artifacts (never forget Agile Modeling's Multiple Models principle). For example, when it comes to conceptual modeling ORM diagrams aren't your only option – In addition to LDMs it is quite common for people to create UML class diagrams and even Class Responsibility Collaborator (CRC) cards instead. In fact, my experience is that CRC cards are superior to ORM diagrams because it is very easy to get project stakeholders actively involved in the creation of the model. Instead of a traditional, analyst-led drawing session you can instead facilitate stakeholders through the creation of CRC cards.

1.1 How are Data Models Used in Practice?

Although methodology issues are covered later, we need to discuss how data models can be used in practice to better understand them. You are likely to see three basic styles of data model:

- **Conceptual data models**. These models, sometimes called domain models, are typically used to explore domain concepts with project stakeholders. On Agile teams high-level conceptual models are often created as part of your

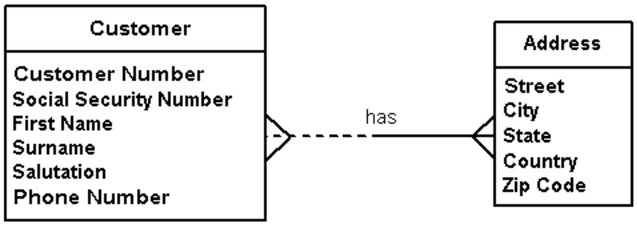


initial requirements envisioning efforts as they are used to explore the high-level static business structures and concepts. On traditional teams conceptual data models are often created as the precursor to LDMs or as alternatives to LDMs.

- **Logical data models (LDMs)**. LDMs are used to explore the domain concepts, and their relationships, of your problem domain. This could be done for the scope of a single project or for your entire enterprise. LDMs depict the logical entity types, typically referred to simply as entity types, the data attributes describing those entities, and the relationships between the entities. LDMs are rarely used on Agile projects although often are on traditional projects (where they rarely seem to add much value in practice).
- **Physical data models (PDMs)**. PDMs are used to design the internal schema of a database, depicting the data tables, the data columns of those tables, and the relationships between the tables. PDMs often prove to be useful on both Agile and traditional projects and as a result the focus of this article is on physical modeling.

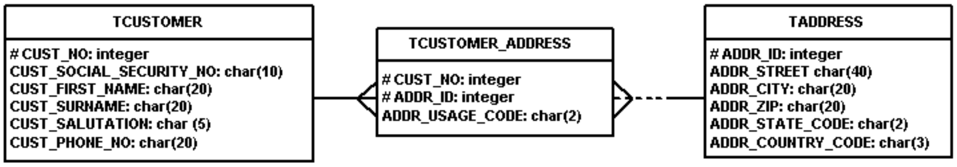
Although LDMs and PDMs sound very similar, and they in fact are, the level of detail that they model can be significantly different. This is because the goals for each diagram is different – you can use an LDM to explore domain concepts with your stakeholders and the PDM to define your database design. Figure 1 presents a simple LDM and Figure 2 a simple PDM, both modeling the concept of customers and addresses as well as the relationship between them. Both diagrams apply the Barker notation, summarized below. Notice how the PDM shows greater detail, including an associative table required to implement the association as well as the keys needed to maintain the relationships. More on these concepts later. PDMs should also reflect your organization's database naming standards, in this case an abbreviation of the entity name is appended to each column name and an abbreviation for "Number" was consistently introduced. A PDM should also indicate the data types for the columns, such as integer and char(5). Although Figure 2 does not show them, lookup tables (also called reference tables or description tables) for how the address is used as well as for states and countries are implied by the attributes *ADDR_USAGE_CODE*, *STATE_CODE*, and *COUNTRY_CODE*.

Figure 1. A simple logical data model.



Copyright 2002-2006 Scott W. Ambler

Figure 2. A simple physical data model.



Copyright 2002-2006 Scott W. Ambler

An important observation about Figures 1 and 2 is that I'm not slavishly following Barker's approach to naming relationships. For example, between *Customer* and *Address* there really should be two names "Each CUSTOMER may be located in one or more ADDRESSES" and "Each ADDRESS may be the site of one or more CUSTOMERS". Although these names explicitly define the relationship I personally think that they're visual noise that clutter the diagram. I prefer simple names such as "has" and then trust my readers to interpret the name in each direction. I'll only add more information where it's needed, in this case I think that it isn't. However, a significant advantage of describing the names the way that Barker suggests is that it's a good test to see if you actually understand the relationship – if you can't name it then you likely don't understand it.

Data models can be used effectively at both the enterprise level and on projects. Enterprise architects will often create one or more high-level LDMs that depict the data structures that support your enterprise, models typically referred to as enterprise data models or enterprise information models. An enterprise data model is one of several views that your organization's enterprise architects may choose to maintain and support – other views may explore your network/hardware infrastructure, your organization structure, your software infrastructure, and your business processes (to name a few). Enterprise data models provide information that a project team can use both as a set of constraints as well as important insights into the structure of their system.

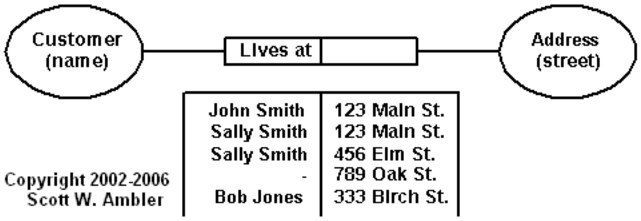
Project teams will typically create LDMs as a primary analysis artifact when their implementation environment is predominantly procedural in nature, for example they are using structured COBOL as an implementation language. LDMs are also a good choice when a project is data-oriented in nature, perhaps a data warehouse or reporting system is being developed (having said that, experience seems to show that usage-centered approaches appear to work even better). However LDMs are often a poor choice when a project team is using object-oriented or component-based technologies because the developers would rather work with UML diagrams or when the project is not data-oriented in nature. As Agile Modeling advises, apply the right artifact(s) for the job. Or, as your grandfather likely advised you, use the right tool for the job. It's important to note that traditional approaches to Master Data Management (MDM) will often motivate the creation and maintenance of detailed LDMs, an effort that is rarely justifiable in practice when you consider the total cost of ownership (TCO) when calculating the return on investment (ROI) of those sorts of efforts.

When a relational database is used for data storage project teams are best advised to create a PDMs to model its internal schema. My experience is that a PDM is often one of the critical design artifacts for business application development projects.

2.2. What About Conceptual Models?

Halpin (2001) points out that many data professionals prefer to create an Object-Role Model (ORM), an example is depicted in Figure 3, instead of an LDM for a conceptual model. The advantage is that the notation is very simple, something your project stakeholders can quickly grasp, although the disadvantage is that the models become large very quickly. ORMs enable you to first explore actual data examples instead of simply jumping to a potentially incorrect abstraction – for example Figure 3 examines the relationship between customers and addresses in detail.   For more information about ORM, visit www.orm.net.

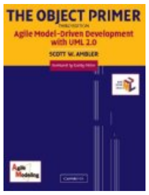
Figure 3. A simple Object-Role Model.



John Smith	123 Main St.
Sally Smith	123 Main St.
Sally Smith	456 Elm St.
-	789 Oak St.
Bob Jones	333 Birch St.

Copyright 2002-2006 Scott W. Ambler

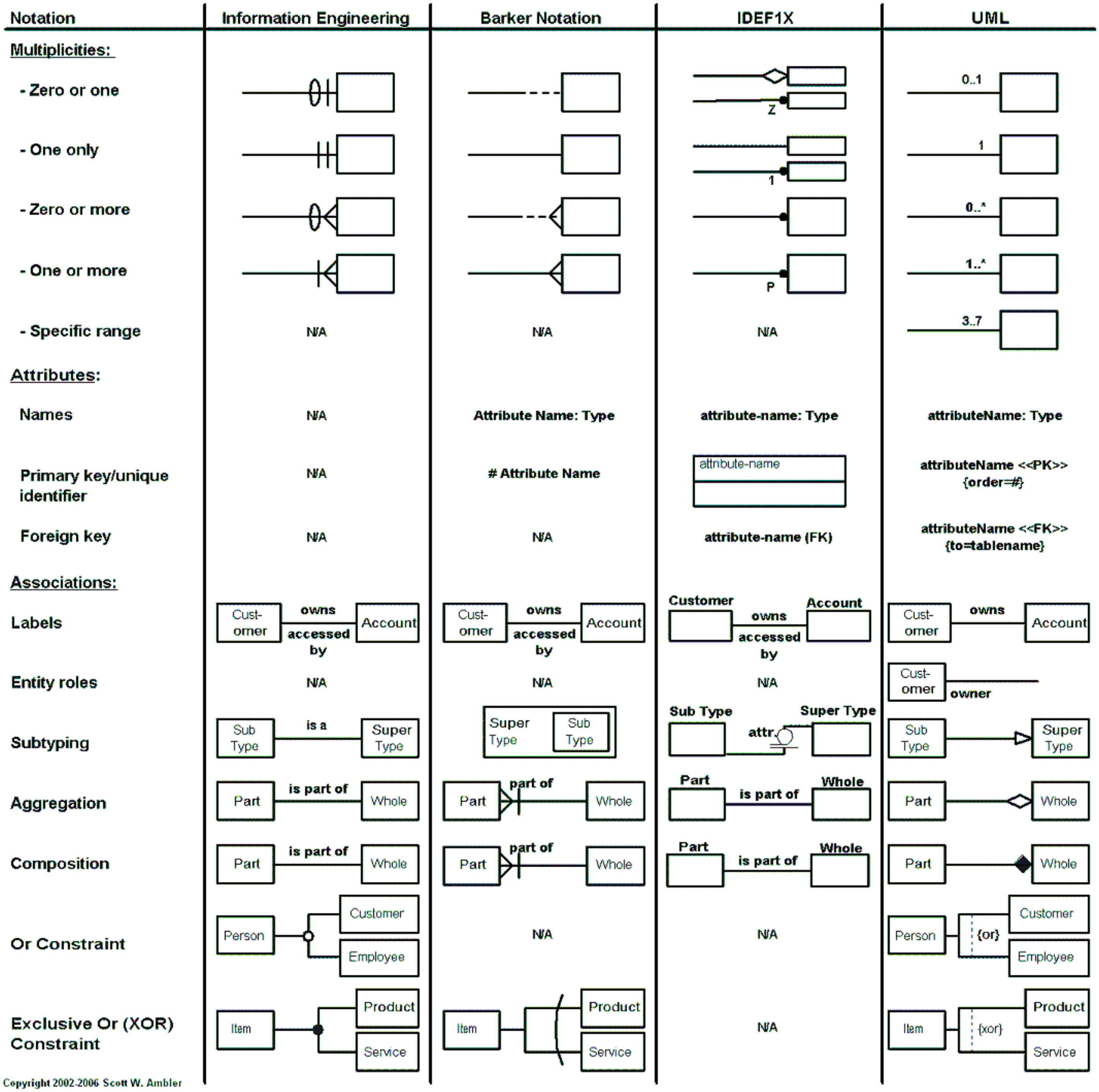
My experience is that people will capture information in the best place that they know. As a result I typically discard ORMs after I'm finished with them. I sometimes user ORMs to explore the domain with project stakeholders but later replace them with a more traditional artifact such as an LDM, a class diagram, or even a PDM. As a generalizing specialist, someone with one or more specialties who also strives to gain general skills and knowledge, this is an easy decision for me to make; I know that this information that I've just "discarded" will be captured in another artifact – a model, the tests, or even the code – that I understand. A specialist who only understands a limited number of artifacts and therefore "hands-off" their work to other specialists doesn't have this as an option. Not only are they tempted to keep the artifacts that they create but also to invest even more time to enhance the artifacts. Generalizing specialists are more likely than specialists to travel light.



2.3. Common Data Modeling Notations

Figure 4 presents a summary of the syntax of four common data modeling notations: Information Engineering (IE), Barker, IDEF1X, and the Unified Modeling Language (UML). This diagram isn't meant to be comprehensive, instead its goal is to provide a basic overview. Furthermore, for the sake of brevity I wasn't able to depict the highly-detailed approach to relationship naming that Barker suggests. Although I provide a brief description of each notation in Table 1 I highly suggest David Hay's paper A Comparison of Data Modeling Techniques as he goes into greater detail than I do.

Figure 4. Comparing the syntax of common data modeling notations.



Notation	Information Engineering	Barker Notation	IDEF1X	UML
Multiplicities:				
- Zero or one			Z	0..1
- One only			1	1
- Zero or more				0..*
- One or more			P	1..*
- Specific range	N/A	N/A	N/A	3..7
Attributes:				
Names	N/A	Attribute Name: Type	attribute-name: Type	attributeName: Type
Primary key/unique identifier	N/A	# Attribute Name	attribute-name	attributeName <<PK>> {order=#}
Foreign key	N/A	N/A	attribute-name (FK)	attributeName <<FK>> {to=tablename}
Associations:				
Labels	Cust-omer owns accessed by Account	Cust-omer owns accessed by Account	Customer owns accessed by Account	Cust-omer owns Account
Entity roles	N/A	N/A	N/A	Cust-omer owner
Subtyping	Sub Type is a Super Type	Super Type Sub Type	Sub Type attr. Super Type	Sub Type Super Type
Aggregation	Part is part of Whole	Part part of Whole	Part is part of Whole	Part Whole
Composition	Part is part of Whole	Part part of Whole	Part is part of Whole	Part Whole
Or Constraint	Person Customer Employee	N/A	N/A	Person {or} Customer Employee
Exclusive Or (XOR) Constraint	Item Product Service	Item Product Service	N/A	Item {xor} Product Service

Copyright 2002-2006 Scott W. Ambler

Table 1. Discussing common data modeling notations.

Notation	Comments
IE	The IE notation (Finkelstein 1989) is simple and easy to read, and is well suited for high-level logical and enterprise data modeling. The only drawback of this notation, arguably an advantage, is that it does not support the identification of attributes of an entity. The assumption is that the attributes will be modeled with another diagram or simply described in the supporting documentation.
Barker	The Barker notation is one of the more popular ones, it is supported by Oracle’s toolset, and is well suited for all types of data models. It’s approach to subtyping can become clunky with hierarchies that go several levels deep.
IDEF1X	This notation is overly complex. It was originally intended for physical modeling but has been misapplied for logical modeling

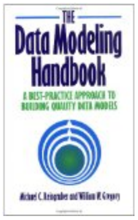
	as well. Although popular within some U.S. government agencies, particularly the Department of Defense (DoD), this notation has been all but abandoned by everyone else. Avoid it if you can.
UML	This is not an official data modeling notation (yet). Although several suggestions for a data modeling profile for the UML exist, none are complete and more importantly are not “official" UML yet. However, the Object Management Group (OMG) in December 2005 announced an RFP for data-oriented models.

3. How to Model Data

It is critical for an application developer to have a grasp of the fundamentals of data modeling so they can not only read data models but also work effectively with Agile DBAs who are responsible for the data-oriented aspects of your project. Your goal reading this section is not to learn how to become a data modeler, instead it is simply to gain an appreciation of what is involved.

The following tasks are performed in an iterative manner:

- Identify entity types
- Identify attributes
- Apply naming conventions
- Identify relationships
- Apply data model patterns
- Assign keys
- Normalize to reduce data redundancy
- Denormalize to improve performance



Very good practical books about data modeling include Joe Celko’s Data & Databases and Data Modeling for Information Professionals as they both focus on practical issues with data modeling. The Data Modeling Handbook and Data Model Patterns are both excellent resources once you’ve mastered the fundamentals. An Introduction to Database Systems is a good academic treatise for anyone wishing to become a data specialist.

3.1 Identify Entity Types

An entity type, also simply called entity (not exactly accurate terminology, but very common in practice), is similar conceptually to object-orientation’s concept of a class – an entity type represents a collection of similar objects. An entity type could represent a collection of people, places, things, events, or concepts. Examples of entities in an order entry system would include *Customer*, *Address*, *Order*, *Item*, and *Tax*. If you were class modeling you would expect to discover classes with the exact same names. However, the difference between a class and an entity type is that classes have both data and behavior whereas entity types just have data.

Ideally an entity should be normal, the data modeling world’s version of cohesive. A normal entity depicts one concept, just like a cohesive class models one concept. For example, customer and order are clearly two different concepts; therefore it makes sense to model them as separate entities.

3.2 Identify Attributes

Each entity type will have one or more data attributes. For example, in Figure 1 you saw that the *Customer* entity has attributes such as *First Name* and *Surname* and in Figure 2 that the *TCUSTOMER* table had corresponding data columns *CUST_FIRST_NAME* and *CUST_SURNAME* (a column is the implementation of a data attribute within a relational database).

Attributes should also be cohesive from the point of view of your domain, something that is often a judgment call. – in Figure 1 we decided that we wanted to model the fact that people had both first and last names instead of just a name (e.g. “Scott" and “Ambler" vs. “Scott Ambler") whereas we did not distinguish between the sections of an American zip code (e.g. 90210-1234-5678). Getting the level of detail right can have a significant impact on your development and maintenance efforts. Refactoring a single data column into several columns can be difficult, database refactoring is described in detail in Database Refactoring, although over-specifying an attribute (e.g. having three attributes for zip code when you only needed one) can result in overbuilding your system and hence you incur greater development and maintenance costs than you actually needed.

3.3 Apply Data Naming Conventions

Your organization should have standards and guidelines applicable to data modeling, something you should be able to obtain from your enterprise administrators (if they don't exist you should lobby to have some put in place). These guidelines should include naming conventions for both logical and physical modeling, the logical naming conventions should be focused on human readability whereas the physical naming conventions will reflect technical considerations. You can clearly see that different naming conventions were applied in Figures 1 and 2.

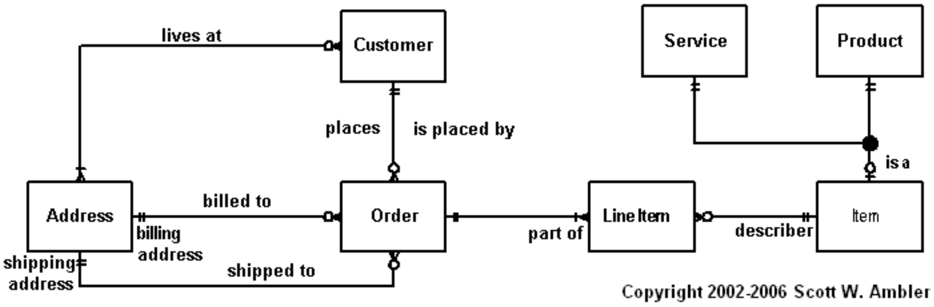
As you saw in Introduction to Agile Modeling, AM includes the Apply Modeling Standards practice. The basic idea is that developers should agree to and follow a common set of modeling standards on a software project. Just like there is value in following common coding conventions, clean code that follows your chosen coding guidelines is easier to understand and evolve than code that doesn't, there is similar value in following common modeling conventions.

3.4 Identify Relationships

In the real world entities have relationships with other entities. For example, customers PLACE orders, customers LIVE AT addresses, and line items ARE PART OF orders. Place, live at, and are part of are all terms that define relationships between entities. The relationships between entities are conceptually identical to the relationships (associations) between objects.

Figure 5 depicts a partial LDM for an online ordering system. The first thing to notice is the various styles applied to relationship names and roles – different relationships require different approaches. For example the relationship between *Customer* and *Order* has two names, *places* and *is placed by*, whereas the relationship between *Customer* and *Address* has one. In this example having a second name on the relationship, the idea being that you want to specify how to read the relationship in each direction, is redundant – you're better off to find a clear wording for a single relationship name, decreasing the clutter on your diagram. Similarly you will often find that by specifying the roles that an entity plays in a relationship will often negate the need to give the relationship a name (although some CASE tools may inadvertently force you to do this). For example the role of *billing address* and the label *billed to* are clearly redundant, you really only need one. For example the role *part of* that *Line Item* has in its relationship with *Order* is sufficiently obvious without a relationship name.

Figure 5. A logical data model (Information Engineering notation).

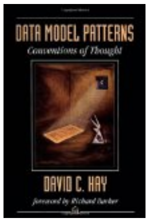


You also need to identify the cardinality and optionality of a relationship (the UML combines the concepts of optionality and cardinality into the single concept of multiplicity). Cardinality represents the concept of "how many" whereas optionality represents the concept of "whether you must have something." For example, it is not enough to know that customers place orders. How many orders can a customer place? None, one, or several? Furthermore, relationships are two-way streets: not only do customers place orders, but orders are placed by customers. This leads to questions like: how many customers can be enrolled in any given order and is it possible to have an order with no customer involved? Figure 5 shows that customers place zero or more orders and that any given order is placed by one customer and one customer only. It also shows that a customer lives at one or more addresses and that any given address has zero or more customers living at it.

Although the UML distinguishes between different types of relationships – associations, inheritance, aggregation, composition, and dependency – data modelers often aren't as concerned with this issue as much as object modelers are. Subtyping, one application of inheritance, is often found in data models, an example of which is the *is a* relationship between *Item* and it's two "sub entities" *Service* and *Product*. Aggregation and composition are much less common and typically must be implied from the data model, as you see with the *part of* role that *Line Item* takes with *Order*. UML dependencies are typically a software construct and therefore wouldn't appear on a data model, unless of course it was a very highly detailed physical model that showed how views, triggers, or stored procedures depended on other aspects of the database schema.

3.5 Apply Data Model Patterns

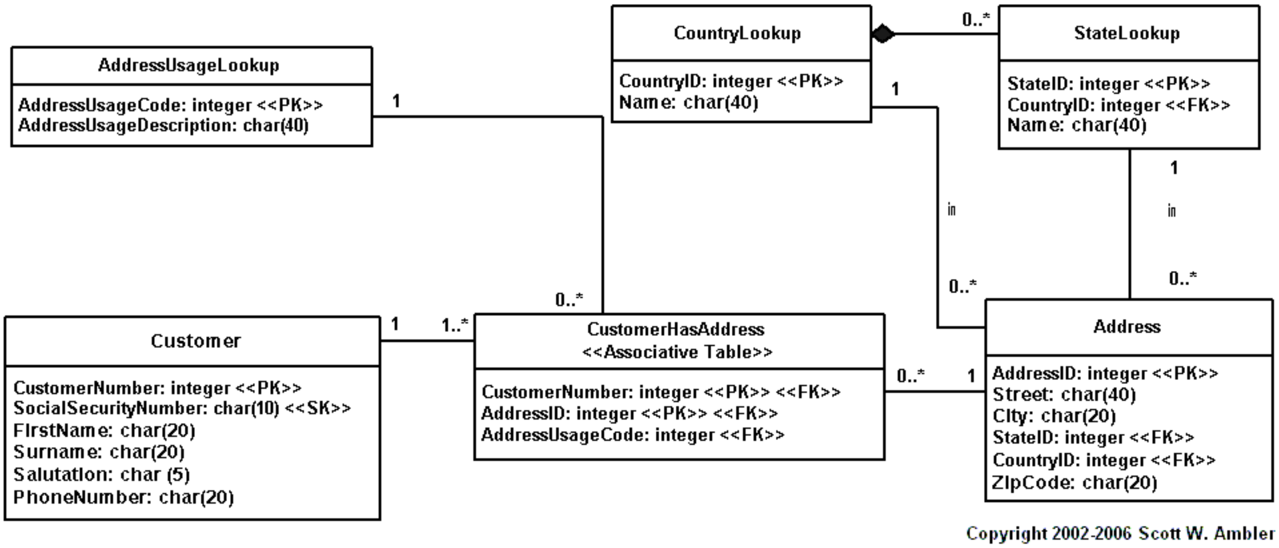
Some data modelers will apply common data model patterns, David Hay's book Data Model Patterns is the best reference on the subject, just as object-oriented developers will apply analysis patterns (Fowler 1997; Ambler 1997) and design patterns (Gamma et al. 1995). Data model patterns are conceptually closest to analysis patterns because they describe solutions to common domain issues. Hay's book is a very good reference for anyone involved in analysis-level modeling, even when you're taking an object approach instead of a data approach because his patterns model business structures from a wide variety of business domains.



3.6 Assign Keys

There are two fundamental strategies for assigning keys to tables. First, you could assign a natural key which is one or more existing data attributes that are unique to the business concept. The *Customer* table of Figure 6 there was two candidate keys, in this case *CustomerNumber* and *SocialSecurityNumber*. Second, you could introduce a new column, called a surrogate key, which is a key that has no business meaning. An example of which is the *AddressID* column of the *Address* table in Figure 6. Addresses don't have an "easy" natural key because you would need to use all of the columns of the *Address* table to form a key for itself (you might be able to get away with just the combination of *Street* and *ZipCode* depending on your problem domain), therefore introducing a surrogate key is a much better option in this case.

Figure 6. Customer and Address revisited (UML notation).


AddressUsageLookup
AddressUsageCode: integer <<PK>>
AddressUsageDescription: char(40)
CountryLookup
CountryID: integer <<PK>>
Name: char(40)
StateLookup
StateID: integer <<PK>>
CountryID: integer <<FK>>
Name: char(40)
Customer
CustomerNumber: integer <<PK>>
SocialSecurityNumber: char(10) <<SK>>
FirstName: char(20)
Surname: char(20)
Salutation: char (5)
PhoneNumber: char(20)
CustomerHasAddress
<<Associative Table>>
CustomerNumber: integer <<PK>> <<FK>>
AddressID: integer <<PK>> <<FK>>
AddressUsageCode: integer <<FK>>
Address
AddressID: integer <<PK>>
Street: char(40)
City: char(20)
StateID: integer <<FK>>
CountryID: integer <<FK>>
ZipCode: char(20)
1
0..*
1
1..*
0..*
1
in
0..*
1
in
0..*
0..*
1
Copyright 2002-2006 Scott W. Ambler

Let's consider Figure 6 in more detail. Figure 6 presents an alternative design to that presented in Figure 2, a different naming convention was adopted and the model itself is more extensive. In Figure 6 the *Customer* table has the *CustomerNumber* column as its primary key and *SocialSecurityNumber* as an alternate key. This indicates that the preferred way to access customer information is through the value of a person's customer number although your software can get at the same information if it has the person's social security number. The *CustomerHasAddress* table has a composite primary key, the combination of *CustomerNumber* and *AddressID*. A foreign key is one or more attributes in an entity type that represents a key, either primary or secondary, in another entity type. Foreign keys are used to maintain relationships between rows. For example, the relationships between rows in the *CustomerHasAddress* table and the *Customer* table is maintained by the *CustomerNumber* column within the *CustomerHasAddress* table. The interesting thing about the *CustomerNumber* column is the fact that it is part of the primary key for *CustomerHasAddress* as well as the foreign key to the *Customer* table. Similarly, the *AddressID* column is part of the primary key of *CustomerHasAddress* as well as a foreign key to the *Address* table to maintain the relationship with rows of *Address*.

Although the "natural vs. surrogate" debate is one of the great religious issues within the data community, the fact is that neither strategy is perfect and you'll discover that in practice (as we see in Figure 6) sometimes it makes sense to use natural keys and sometimes it makes sense to use surrogate keys. In Choosing a Primary Key: Natural or Surrogate? I describe the relevant issues in detail.

3.7 Normalize to Reduce Data Redundancy

Data normalization is a process in which data attributes within a data model are organized to increase the cohesion of entity types. In other words, the goal of data normalization is to reduce and even eliminate data redundancy, an important consideration for application developers because it is incredibly difficult to stores objects in a relational database that maintains the same information in several places. Table 2 summarizes the three most common normalization rules describing how to put entity types into a series of increasing levels of normalization. Higher levels of data normalization (Date 2000) are beyond the scope of this book. With respect to terminology, a data schema is considered to be at the level of normalization of its least normalized entity type. For example, if all of your entity types are at second normal form (2NF) or higher then we say that your data schema is at 2NF.

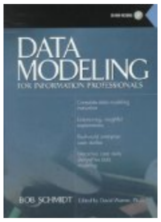


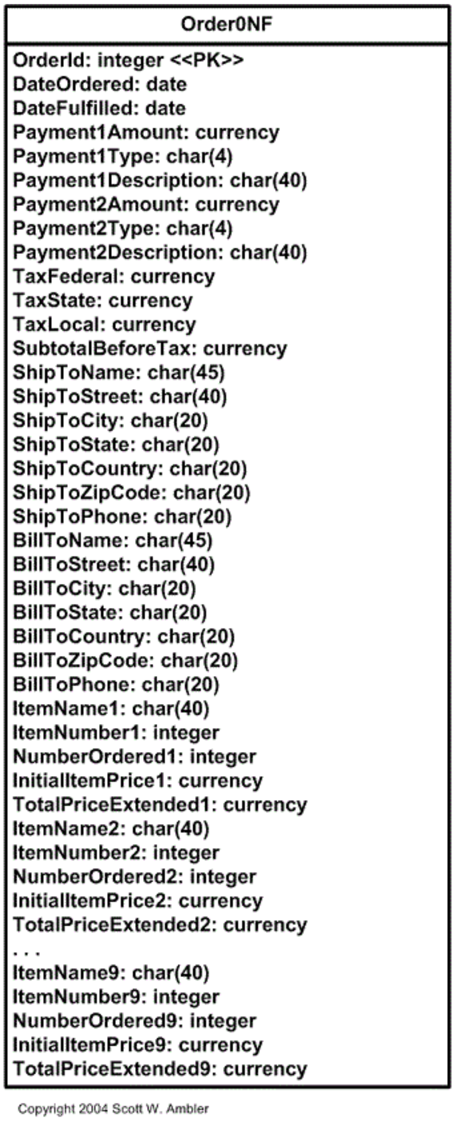
Table 2. Data Normalization Rules.

Level	Rule
First normal form (1NF)	An entity type is in 1NF when it contains no repeating groups of data.
Second normal form (2NF)	An entity type is in 2NF when it is in 1NF and when all of its non-key attributes are fully dependent on its primary key.
Third normal form (3NF)	An entity type is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the primary key.

Figure 7 depicts a database schema in ONF whereas Figure 8 depicts a normalized schema in 3NF. Read the Introduction to Data Normalization essay for details.

Why data normalization? The advantage of having a highly normalized data schema is that information is stored in one place and one place only, reducing the possibility of inconsistent data. Furthermore, highly-normalized data schemas in general are closer conceptually to object-oriented schemas because the object-oriented goals of promoting high cohesion and loose coupling between classes results in similar solutions (at least from a data point of view). This generally makes it easier to map your objects to your data schema. Unfortunately, normalization usually comes at a performance cost. With the data schema of Figure 7 all the data for a single order is stored in one row (assuming orders of up to nine order items), making it very easy to access. With the data schema of Figure 7 you could quickly determine the total amount of an order by reading the single row from the *Order0NF* table. To do so with the data schema of Figure 8 you would need to read data from a row in the *Order* table, data from all the rows from the *OrderItem* table for that order and data from the corresponding rows in the *Item* table for each order item. For this query, the data schema of Figure 7 very likely provides better performance.

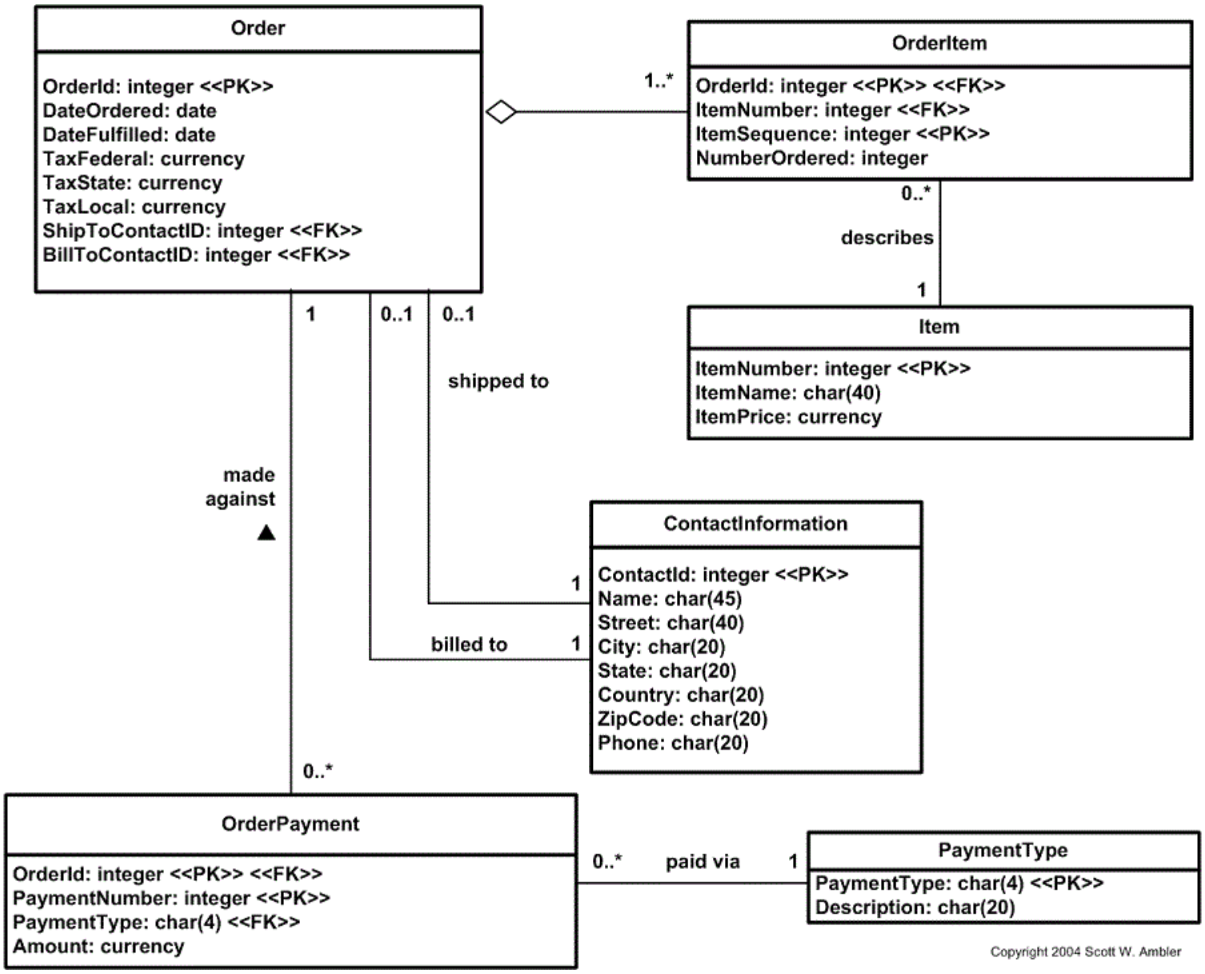
Figure 7. An Initial Data Schema for Order (UML Notation).



Order0NF

OrderId: integer <<PK>>
DateOrdered: date
DateFulfilled: date
Payment1Amount: currency
Payment1Type: char(4)
Payment1Description: char(40)
Payment2Amount: currency
Payment2Type: char(4)
Payment2Description: char(40)
TaxFederal: currency
TaxState: currency
TaxLocal: currency
SubtotalBeforeTax: currency
ShipToName: char(45)
ShipToStreet: char(40)
ShipToCity: char(20)
ShipToState: char(20)
ShipToCountry: char(20)
ShipToZipCode: char(20)
ShipToPhone: char(20)
BillToName: char(45)
BillToStreet: char(40)
BillToCity: char(20)
BillToState: char(20)
BillToCountry: char(20)
BillToZipCode: char(20)
BillToPhone: char(20)
ItemName1: char(40)
ItemNumber1: integer
NumberOrdered1: integer
InitialItemPrice1: currency
TotalPriceExtended1: currency
ItemName2: char(40)
ItemNumber2: integer
NumberOrdered2: integer
InitialItemPrice2: currency
TotalPriceExtended2: currency
. . .
ItemName9: char(40)
ItemNumber9: integer
NumberOrdered9: integer
InitialItemPrice9: currency
TotalPriceExtended9: currency

Figure 8. A normalized schema in 3NF (UML Notation).

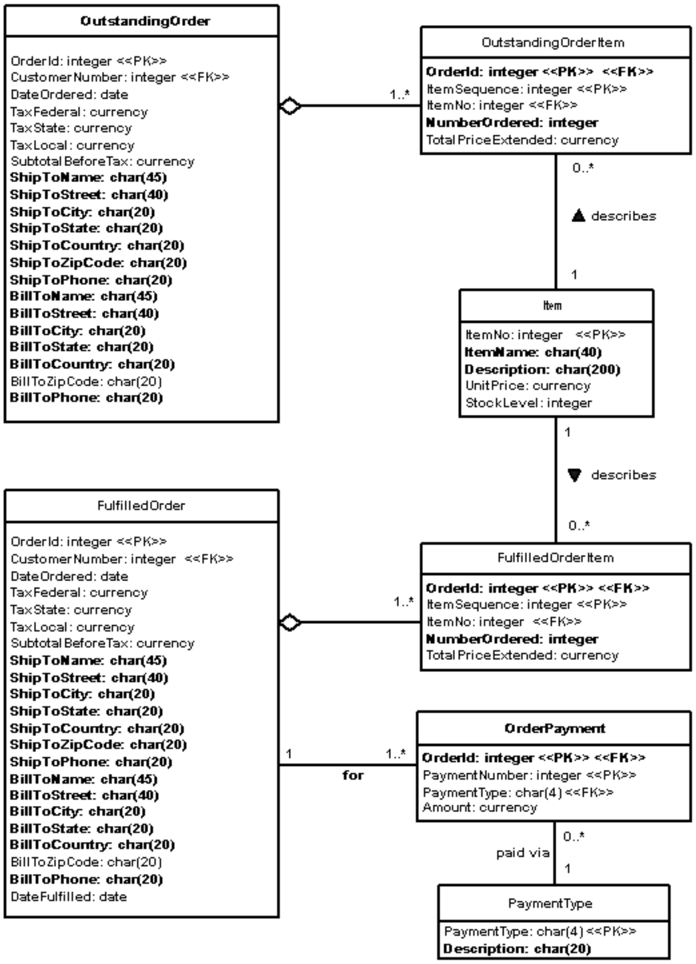


In class modeling, there is a similar concept called Class Normalization although that is beyond the scope of this article.

3.8 Denormalize to Improve Performance

Normalized data schemas, when put into production, often suffer from performance problems. This makes sense – the rules of data normalization focus on reducing data redundancy, not on improving performance of data access. An important part of data modeling is to denormalize portions of your data schema to improve database access times. For example, the data model of Figure 9 looks nothing like the normalized schema of Figure 8. To understand why the differences between the schemas exist you must consider the performance needs of the application. The primary goal of this system is to process new orders from online customers as quickly as possible. To do this customers need to be able to search for items and add them to their order quickly, remove items from their order if need be, then have their final order totaled and recorded quickly. The secondary goal of the system is to the process, ship, and bill the orders afterwards.

Figure 9. A Denormalized Order Data Schema (UML notation).



To denormalize the data schema the following decisions were made:

1. To support quick searching of item information the *Item* table was left alone.
2. To support the addition and removal of order items to an order the concept of an *OrderItem* table was kept, albeit split in two to support outstanding orders and fulfilled orders. New order items can easily be inserted into the *OutstandingOrderItem* table, or removed from it, as needed.
3. To support order processing the *Order* and *OrderItem* tables were reworked into pairs to handle outstanding and fulfilled orders respectively. Basic order information is first stored in the *OutstandingOrder* and *OutstandingOrderItem* tables and then when the order has been shipped and paid for the data is then removed from those tables and copied into the *FulfilledOrder* and *FulfilledOrderItem* tables respectively. Data access time to the two tables for outstanding orders is reduced because only the active orders are being stored there. On average an order may be outstanding for a couple of days, whereas for financial reporting reasons may be stored in the fulfilled order tables for several years until archived. There is a performance penalty under this scheme because of the need to delete outstanding orders and then resave them as fulfilled orders, clearly something that would need to be processed as a transaction.
4. The contact information for the person(s) the order is being shipped and billed to was also denormalized back into the *Order* table, reducing the time it takes to write an order to the database because there is now one write instead of two or three. The retrieval and deletion times for that data would also be similarly improved.

Note that if your initial, normalized data design meets the performance needs of your application then it is fine as is. Denormalization should be resorted to only when performance testing shows that you have a problem with your objects and subsequent profiling reveals that you need to improve database access time. As my grandfather said, if it ain't broke don't fix it.

5. Evolutionary/Agile Data Modeling

Evolutionary data modeling is data modeling performed in an iterative and incremental manner. The article Evolutionary Development explores evolutionary software development in greater detail. Agile data modeling is evolutionary data modeling done in a collaborative manner. The article Agile Data Modeling: From Domain Modeling to Physical Modeling works through a case study which shows how to take an agile approach to data modeling.

Although you wouldn't think it, data modeling can be one of the most challenging tasks that an Agile DBA can be involved with on an agile software development project. Your approach to data modeling will often be at the center of any controversy between the agile software developers and the traditional data professionals within your organization. Agile software developers will lean towards an evolutionary approach where data modeling is just one of many activities whereas traditional data professionals will often lean towards a big design up front (BDUF) approach where data models are the primary artifacts, if not THE artifacts. This problem results from a combination of the cultural impedance mismatch, a misguided need to enforce the "one truth", and “normal" political maneuvering within your organization. As a result Agile DBAs often find that navigating the political waters is an important part of their data modeling efforts.

6. How to Become Better At Modeling Data

How do you improve your data modeling skills? Practice, practice, practice. Whenever you get a chance you should work closely with Agile DBAs, volunteer to model data with them, and ask them questions as the work progresses. Agile DBAs will be following the AM practice Model With Others so should welcome the assistance as well as the questions – one of the best ways to really learn your craft is to have someone as “why are you doing it that way". You should be able to learn physical data modeling skills from Agile DBAs, and often logical data modeling skills as well.

Similarly you should take the opportunity to work with the enterprise architects within your organization. As you saw in Agile Enterprise Architecture they should be taking an active role on your project, mentoring your project team in the enterprise architecture (if any), mentoring you in modeling and architectural skills, and aiding in your team’s modeling and development efforts. Once again, volunteer to work with them and ask questions when you are doing so. Enterprise architects will be able to teach you conceptual and logical data modeling skills as well as instill an appreciation for enterprise issues.

You also need to do some reading. Although this article is a good start it is only a brief introduction. The best approach is to simply ask the Agile DBAs that you work with what they think you should read.

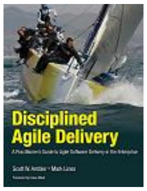
My final word of advice is that it is critical for application developers to understand and appreciate the fundamentals of data modeling. This is a valuable skill to have and has been since the 1970s. It also provides a common framework within which you can work with Agile DBAs, and may even prove to be the initial skill that enables you to make a career transition into becoming a full-fledged Agile DBA.

Share with friends: Tweet LinkedIn Facebook StumbleUpon Digg Baidu Google +

Let Us Help

We actively work with clients around the world to improve their information technology (IT) practices, typically in the role of mentor/coach, team lead, or trainer. A full description of what we do, and how to contact us, can be found at Scott Ambler + Associates.

Recommended Reading


Disciplined Agile Delivery
A Practitioner's Guide to Agile Software Delivery in the Enterprise
Scott W. Ambler • Mark Lines

This book, Disciplined Agile Delivery: A Practitioner's Guide to Agile Software Delivery in the Enterprise describes the Disciplined Agile Delivery (DAD) process decision framework. The DAD framework is a people-first, learning-oriented hybrid agile approach to IT solution delivery. It has a risk-value delivery lifecycle, is goal-driven, is enterprise aware, and provides the foundation for scaling agile. This book is particularly important for anyone who wants to understand how agile works from end-to-end within an enterprise setting. Data professionals will find it interesting because it shows how agile modeling and agile database techniques fit into the overall solution delivery process. Enterprise professionals will find it interesting beause it explicitly promotes the idea that disciplined agile teams should be enterprise aware and therefore work closely with enterprise teams. Existing agile developers will find it interesting because it shows how to extend Scrum-based and Kanban-based strategies to provide a coherent, end-to-end streamlined delivery process.

I also maintain an agile database books page which overviews many books you will find interesting.