cases you may want to use the natural key instead of a surrogate key such as a UUID or POID.

How can you be effective at assigning keys? Consider the following tips:

1. **Avoid "smart" keys**. A "smart" key is one that contains one or more subparts which provide meaning. For example the first two digits of an U.S. zip code indicate the state that the zip code is in. The first problem with smart keys is that have business meaning. The second problem is that their use often becomes convoluted over time. For example some large states have several codes, California has zip codes beginning with 90 and 91, making queries based on state codes more complex. Third, they often increase the chance that the strategy will need to be expanded. Considering that zip codes are nine digits in length (the following four digits are used at the discretion of owners of buildings uniquely identified by zip codes) it's far less likely that you'd run out of nine-digit numbers before running out of two digit codes assigned to individual states.

2. **Consider assigning natural keys for simple "look up" tables**. A "look up" table is one that is used to relate codes to detailed information. For example, you might have a look up table listing color codes to the names of colors. For example the code 127 represents "Tulip Yellow". Simple look up tables typically consist of a code column and a description/name column whereas complex look up tables consist of a code column and several informational columns.

3. **Natural keys don't always work for "look up" tables**. Another example of a look up table is one that contains a row for each state, province, or territory in North America. For example there would be a row for California, a US state, and for Ontario, a Canadian province. The primary goal of this table is to provide an official list of these geographical entities, a list that is reasonably static over time (the last change to it would have been in the late 1990s when the Northwest Territories, a territory of Canada, was split into Nunavut and Northwest Territories). A valid natural key for this table would be the state code, a unique two character code – e.g. CA for California and ON for Ontario. Unfortunately this approach doesn't work because Canadian government decided to keep the same state code, NW, for the two territories.

4. **Your applications must still support "natural key searches"**. If you choose to take a surrogate key approach to your database design you mustn't forget that your applications must still support searches on the domain columns that still uniquely identify rows. For example, your *Customer* table may have a *Customer_POID* column used as a surrogate key as well as a *Customer_Number* column and a *Social_Security_Number* column. You would likely need to support searches based on both the customer number and the social security number. Searching is discussed in detail in Finding Objects in a Relational Database.

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 data entities. 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 data entities 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 data entity. For example, if all of your data entities 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)	A data entity is in 1NF when it contains no repeating groups of data.
Second normal form (2NF)	A data entity 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)	A data entity is in 3NF when it is in 2NF and when all of its attributes are directly dependent on the primary key.

3.7.1 First Normal Form (1NF)

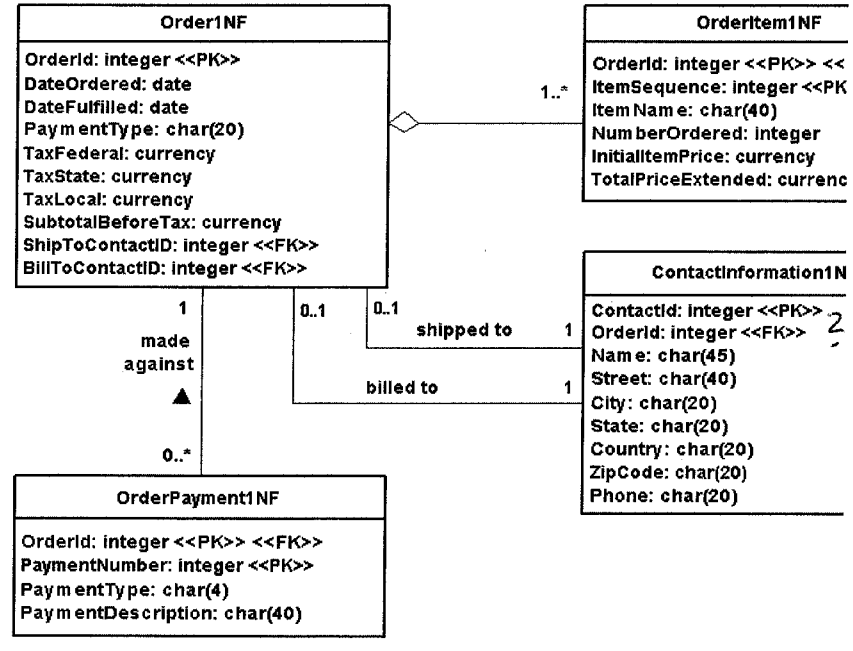
Let's consider an example. A data entity is in first normal form (1NF) when it contains no repeating groups of data. For example, in Figure 7 you see that there are several repeating attributes in the data *Order0NF* table – the ordered item information repeats nine times and the contact information is repeated twice, once for shipping information and once for billing information. Although this initial version of orders could work, what happens when an order has more than nine order items? Do you create additional order records for them? What about the vast majority of orders that only have one or two items? Do we really want to waste all that storage space in the database for the empty fields? Likely not. Furthermore, do you want to write the code required to process the nine copies of item information, even if it is only to marshal it back and forth between the appropriate number of objects. Once again, likely not.

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)
NumberOrdered1: integer
InitialItemPrice1: currency
TotalPriceExtended1: currency
ItemName2: char(40)
NumberOrdered2: integer
InitialItemPrice2: currency
TotalPriceExtended2: currency
. . .
ItemName9: char(40)

Figure 8 presents a reworked data schema where the order schema is put in first normal form. The introduction of the *OrderItem1NF* table enables us to have as many, or as few, order items associated with an order, increasing the flexibility of our schema while reducing storage requirements for small orders (the majority of our business). The *ContactInformation1NF* table offers a similar benefit, when an order is shipped and billed to the same person (once again the majority of cases) we could use the same contact information record in the database to reduce data redundancy. *OrderPayment1NF* was introduced to enable customers to make several payments against an order – *Order0NF* could accept up to two payments, the type being something like "MC" and the description "MasterCard Payment", although with the new approach far more than two payments could be supported. Multiple payments are accepted only when the total of an order is large enough that a customer must pay via more than one approach, perhaps paying some by check and some by credit card.

Figure 8. An Order Data Schema in 1NF (UML Notation).



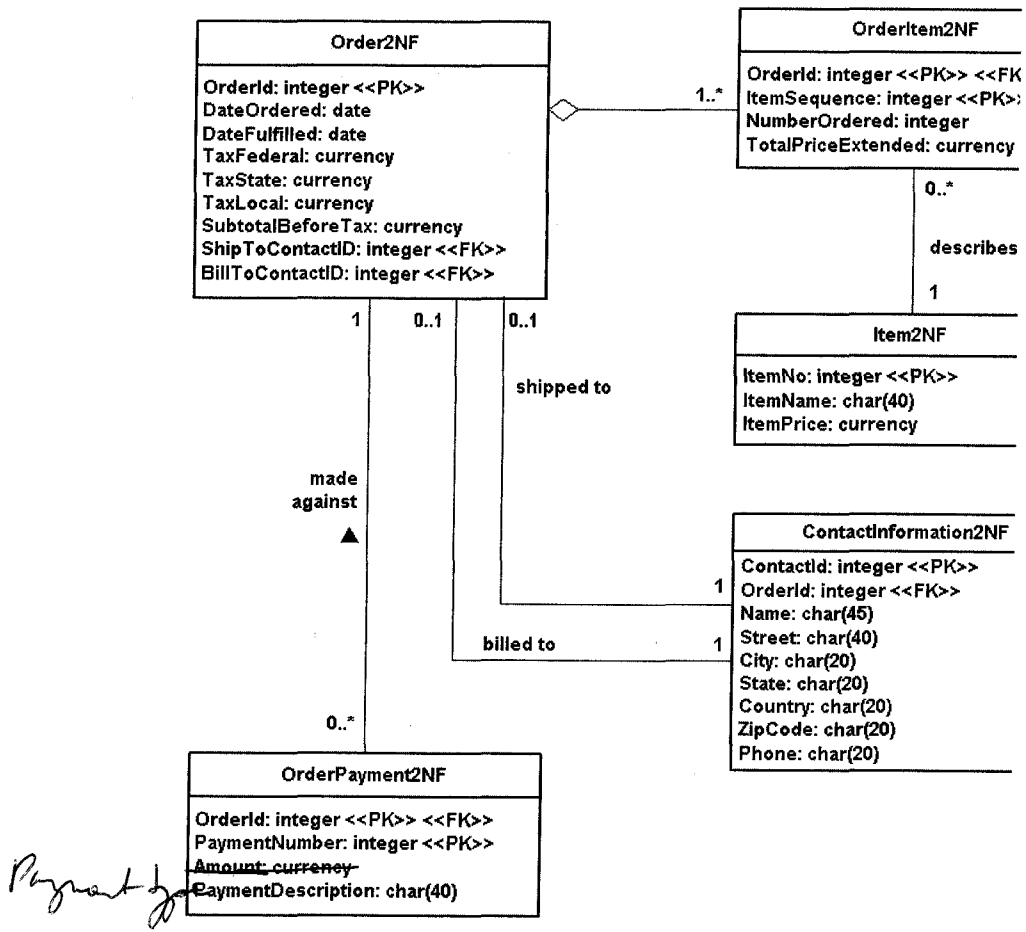
An important thing to notice is the application of primary and foreign keys in the new solution. *Order1NF* has kept *OrderID*, the original key of *Order0NF*, as its primary key. To maintain the relationship back to *Order1NF*, the *OrderItem1NF* table includes the *OrderID* column within its schema, which is why it has the tagged value of {ForeignKey}. When a new table is introduced into a schema, in this case *OrderItem1NF*, as the result of first normalization efforts it is common to use the primary key of the original table (*Order0NF*) as part of the primary key of the new table. Because *OrderID* is not unique for order items, you can have several order items on an order, the column *ItemSequence* was added to form a composite primary key for the *OrderItem1NF* table. A

different approach to keys was taken with the *ContactInformation1NF* table. The column *ContactID*, a surrogate key that has no business meaning, was made the primary key – *OrderID* is needed as a foreign key to maintain the relationship back to *Order1NF*. A good rule of thumb is that if the tables are highly related to one another, there is an aggregation relationship between *Order1NF* and *OrderItem1NF*, then it is likely that it makes sense to include the primary key of the original table as part of the primary key of the new table. If the two tables are not as strongly related, there is merely a relationship between *Order1NF* and *ContactInformation1NF*, then a surrogate key may make more sense: however, because each row of *ContactInformation1NF* is associated with only one row of *Order1NF*, keeping *OrderID* as the key would be a valid (and easier) approach to take.

3.7.2 Second Normal Form (2NF)

Although the solution presented in Figure 8 is improved over that of Figure 7, it can be normalized further. Figure 9 presents the data schema of Figure 8 in second normal form (2NF). A data entity is in second normal form (2NF) when it is in 1NF and when every non-key attribute, any attribute that is not part of the primary key, is fully dependent on the primary key. This was definitely not the case with the *OrderItem1NF* table, therefore we need to introduce the new table *Item2NF*. The problem with *OrderItem1NF* is that item information, such as the name and price of an item, do not depend upon an order for that item. For example, if Hal Jordan orders three widgets and Oliver Queen orders five widgets, the facts that the item is called a "widget" and that the unit price is $19.95 is constant. This information depends on the concept of an item, not the concept of an order for an item, and therefore should not be stored in the order items table – therefore the *Item2NF* table was introduced. *OrderItem2NF* retained the *TotalPriceExtended* column, a calculated value that is the number of items ordered multiplied by the price of the item. The value of the *SubtotalBeforeTax* column within the *Order2NF* table is the total of the values of the total price extended for each of its order items.

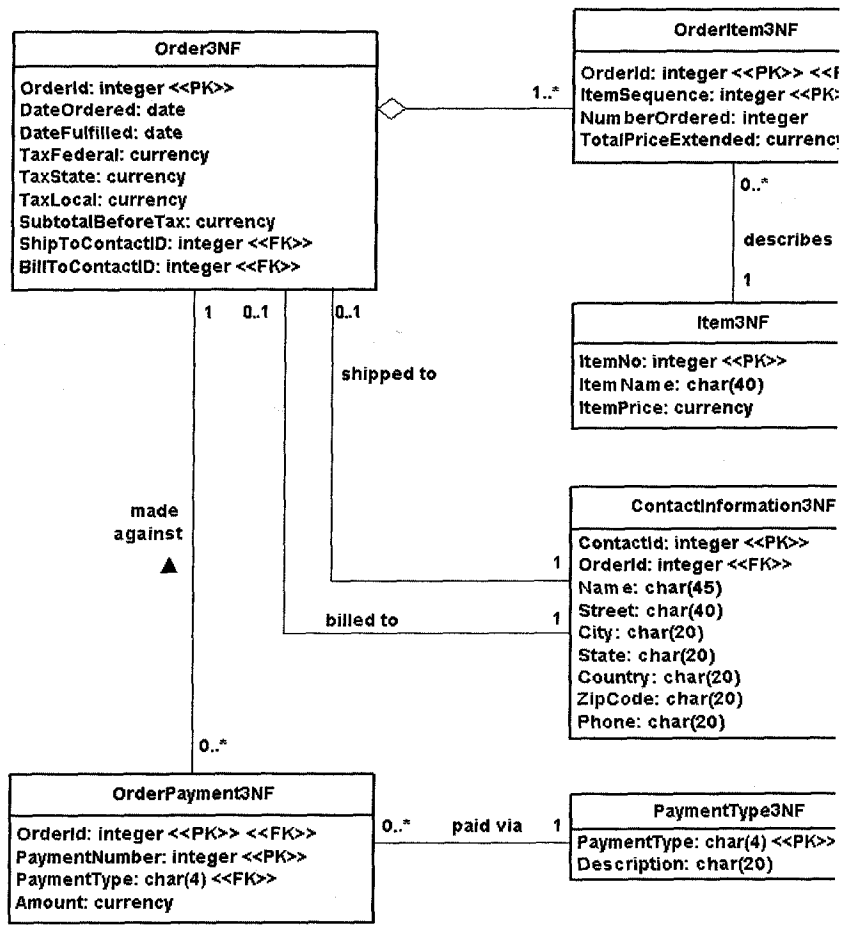
Figure 9. An Order in 2NF (UML Notation).



3.7.3 Third Normal Form (3NF)

A data entity is in *third normal form (3NF)* when it is in 2NF and when all of its attributes are directly dependent on the primary key. A better way to word this rule might be that the attributes of a data entity must depend on all portions of the primary key, therefore 3NF is only an issue only for tables with composite keys. In this case there is a problem with the *OrderPayment2NF* table, the payment type description (such as "Mastercard" or "Check") depends only on the payment type, not on the combination of the order id and the payment type. To resolve this problem the *PaymentType3NF* table was introduced in Figure 10, containing a description of the payment type as well as a unique identifier for each payment type.

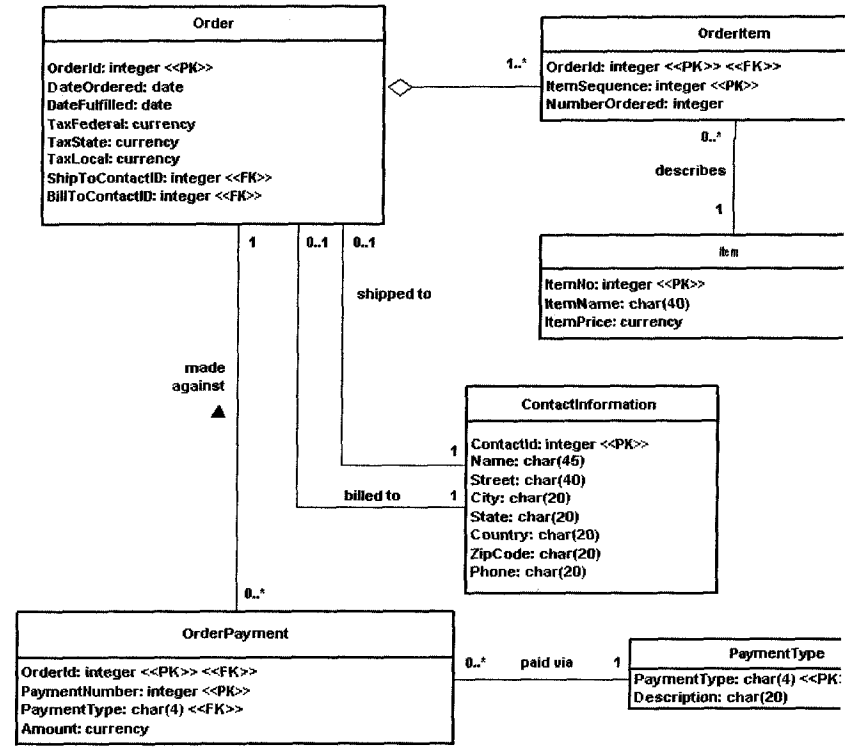
Figure 10. An Order in 3NF (UML Notation).



3.7.4 Beyond 3NF

The data schema of Figure 10 can still be improved upon, at least from the point of view of data redundancy, by removing attributes that can be calculated/derived from other ones. In this case we could remove the *SubtotalBeforeTax* column within the *Order3NF* table and the *TotalPriceExtended* column of *OrderItem3NF*, as you see in Figure 11.

Figure 11. An Order Without Calculated Values (UML Notation).

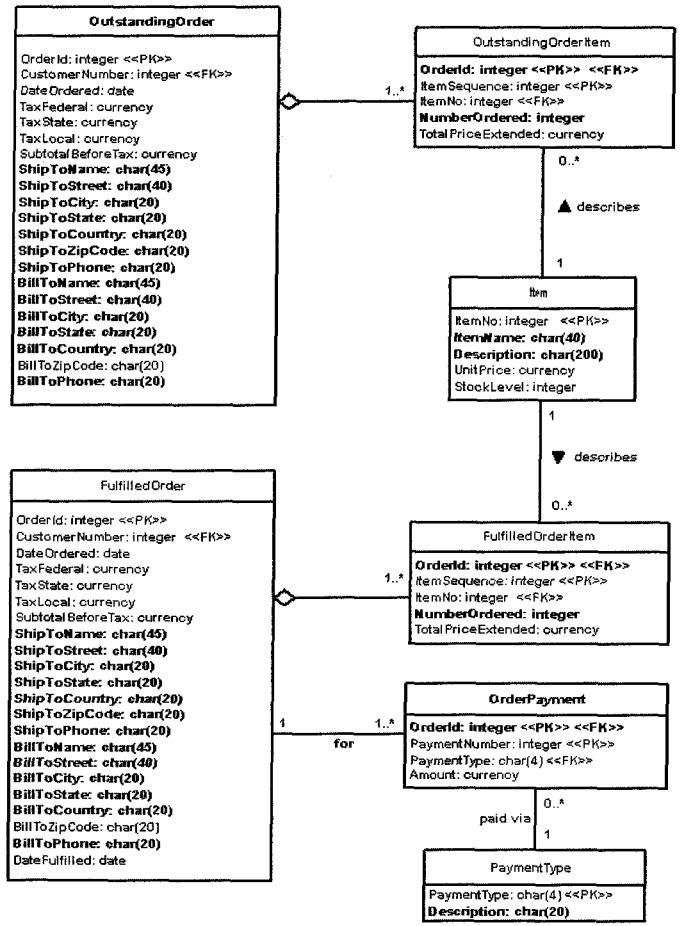


Why data normalization? The advantage of having a highly normalized data schema is that information is stored in one place and one place only, reducing the possibility of inconsistent data. Furthermore, highly-normalized data schemas in general are closer conceptually to object-oriented schemas because 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 11 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.

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 12 looks nothing like the normalized schema of Figure 11. 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 12. 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 says, if it ain't broke don't fix it.

4. Evolutionary Modeling

One of the fundamentals of agile software development is that you should take an iterative and incremental, often referred to as an evolutionary, approach to development. Within an evolutionary approach to development your models, including data-oriented ones, are developed over time. There is no "requirements phase" or "design phase", instead modeling is performed as needed throughout your project in a continuous manner.

Unfortunately many existing data professionals believe that you need to get your data models "mostly right" reasonably early in a project. This misconception is often the result of:

- **Prevailing organizational culture**. Many IT organizations are still following a traditional, near serial software process that includes a "big design up front (BDUF)" approach to modeling. Because they haven't made the shift to agile software development yet they haven't come to the realization that they need to change their mindset. An interesting observations is that slow moving organizations want to freeze everything and that faster organizations don't – perhaps the reason why your organization takes so long to get anything done is your penchant for BDUF?

- **Prevailing professional culture**. The data community is just beginning to assess agile techniques and has not yet had a chance to absorb the evolutionary mindset of agile developers. Agile software development is new, for the most part it's coming out of the object community, and until this book very little attention has been agile database techniques. Worse yet, many within the data community are still struggling with object-orientation (having missed the boat in the early 1990s), let alone agility.

- **Lack of experience with evolutionary techniques**. Many data professionals haven't had the opportunity to try an evolutionary approach, and because they haven't seen it with their own eyes they are justifiably skeptical (but that doesn't mean you should go into a state of denial either). If you don't yet have experience with evolutionary data modeling you can at least read about the experiences of others. Fowler and Sadalage (2003) describe their efforts on a multi-

site, 100+ person project that followed a collection of techniques very similar to those described in this book.

- **Prescriptive processes**. Many organizations have well-defined, prescriptive processes in place that make it difficult to change your data models once they've been accepted. The need to review and baseline models dramatically slow you down, and the need to "accept" a model indicates a command and control mindset this is very likely hampering your efforts. Agile Modeling (AM)'s practices of Model With Others, Active Stakeholder Participation, and Collective Ownership go a long way to removing the need for reviews. When all you know is a prescriptive process it's very difficult to imagine that another, significantly faster and more way is possible.

- **Lack of supporting tools**. Tools are generally behind methods, although with the open source movement and recent consolidations between development tool vendors we're starting to see very good progress. Tools, Sandboxes, and Scripts describes the current state of tools for evolutionary data-oriented development.

So how do you take an evolutionary approach to data modeling? Figure 13 depicts a high-level process diagram to evolutionary development that includes several data-oriented activities. First, notice how the arrows are two-way, implying that you iterate back and forth between activities. Second, notice how there is no starting point. Although you may choose to start with your enterprise model, then do some conceptual modeling, then let your conceptual model drive your object and data schemas this doesn't have to be the case. Depending on the nature of your project you could start with a project-level conceptual model (you may not have an enterprise model) or you may start first with traditional object modeling activities such as use case modeling. It doesn't really matter because agile software developers will iterate to another activity as required. Third, notice how I use the term "enterprise structural modeling" and not "enterprise data modeling" – many organizations are choosing to use UML class models or even UML component models (Herzum and & Sims 2000; Atkinson et. al. 2002) instead of data models for structural modeling. The same fundamental goals are being achieved, albeit by different means. Fourth, I've combined the notions of conceptual and domain modeling in one as they're often commingled anyway (if they're done at all).

Figure 13. Evolutionary development.