Chapter 7

Database Design Models—
the UML Profile for Database Design

In this chapter, we look at the process of moving from the logical design models into the database-specific designs and what happens once we get there. Also included in this chapter is the new UML Profile for Database Design created by Rational Software Corporation for use when designing a database. We look at the stereotypes used in the UML for database-specific needs, how they are implemented, and how EAB Healthcare has used the UML and specifically the Profile to model the company’s database needs.

The Workflow

The use of modeling to design databases far exceeds the use of modeling for applications and is generally done within most organizations with which we have worked and visited. The issue is that modeling a database is generally just that—modeling the database tables, columns, and relationships but not the entire database design. In this book, we look at many other parts of database design that can be modeled and how using the UML helps to model the entire database design. One part of database design that has not been covered very well in the past by the UML is the actual modeling of the database. Designers have primarily focused on using the UML for object-oriented application design and not for database design. With the use of the Profile for Database Design, this has helped open the UML to database design and database designers to the UML. Some tools like Rational Rose automate many of the stereotypes that we
discuss so that the database designer does not need to understand the entire UML or even the stereotypes; designers can just take advantage of the UML without having to be experts.

We will model the database so that we can visualize and understand how the tables are structured and how they relate to other tables and views. Using the UML for the database design, we will also model constraints, triggers, schemas, indexes, stored procedures, and more. In the next chapter we will cover how to model the storage of the data, including tablespaces, databases, and partitioning.

Having a model of the database is important for many users. The database designer uses the data model in order to ensure database rules, including normalization, key migration, and others. The application developer can use the data model, especially if designed in the UML and mapped to the application model, to understand how the application accesses the database. This will help when the application developer is charged with building the data access software. A user interface (UI) developer also takes advantage of the data model to be sure that the UI picks up the available columns and uses the proper lengths, precision, and scale when designing the look and feel of the application. The analysts and end users (customers) use the data model to ensure that the data they believe is needed is captured in the database, in the correct format, to ensure fewer iterations.

During database design we begin to look at the specifics of the database so that we can decide what is being built and how to optimize it fully. This includes the type of server, operating system, database management software (DBMS) and version, and possibly other considerations. The selection of the DBMS will affect how optimization of the database is accomplished once the physical model is designed. Different DBMS systems consider different options for optimization, and the database administrator (DBA) works differently with their schemas depending on the DBMS chosen. For example, DB2 and Oracle are very sophisticated when it comes to storage, while SQL Server has been made more simplistic and doesn’t offer nearly as many options, requiring fewer decisions to be made. Because of the different storage mechanisms used, the indexing specifically is treated differently, but there are many other options to consider, such as view support, triggers, stored procedure languages, and more.

**Working as a Team**

We have already discussed many ways that using the UML to do all of your modeling in one language is helpful in bringing the many teams involved in the development process together, but using the UML for modeling the database itself is helpful in its own right. The database is generally built by a team of data
analysts, database designers, and database administrators along with others, and there must be a way for them to work together while building the database. The database teams themselves need to work together as a team, sharing artifacts, corporate and group standards, templates, diagrams, domains, and descriptions.

The standard entity-relationship (ER) notations do support the needs of the database team at this level, but the UML supports what ER notations support and beyond. The UML was built with ER in mind. ER modeling has existed for a long time, and when Booch, Rumbaugh, and Jacobson created the UML, they built it as a superset of ER notations. Since their intentions were somewhat different from those of database designers, supporting object-oriented development and not database design, some key needs of the database team were left out. With the addition of the UML Profile for Database Design, the UML supports the business models, the requirements models, and the logical and physical application and data models all in one language.

UML Packages

The constructs of the UML, beginning with the concept of packages, encourage sharing of information. By using packages a modeler can logically group information and break up the model artifacts in different ways for different uses. Often teams break up the model into standards packages and packages that different people or teams work on for different parts of the project, different phases (for example, development, testing, and production), and many other groupings that you can use to help make your job and communication among your team members easier (see Figures 7–1 and 7–2).
Packages can include different types of UML objects as well. A package can contain tables and classes, so that you can see visually and organizationally which tables map to which classes. Some modelers have logical design and database design packages so that they can easily see what is used during each phase of the project. Packages can be used as a good way to organize the models for version control, based on the package rather than on individual tables, for example. This way when you start using the model, you know exactly which elements are needed based on the package with which you begin. Having standard elements, such as domains, tables, classes, use cases, and others, stored in standards packages gives a way to understand the standards that can be used in new and existing models. Also, if you are using some sort of configuration management, you can version packages and be sure that they are maintained and not changed on a whim. Packages should be used to store standard modeling elements. The UML supports a derived relationship, which can be created from those standards to the elements that use them, helping to ensure that the elements are updated by changing the standard package. Some tools even support the changes automatically based on the association between the standard elements and the elements to which they are related.

**UML Diagrams**

The database design model represents the physical database design as it will be implemented in the DBMS. The database design is modeled in a new diagram type, called the *database diagram*, provided by the UML Profile for Database Design. Another advantage to the UML is that diagrams do not have to be typed; this means you can have elements of many different types on a diagram,
or you can stereotype the diagram to a specific type and allow only those types of elements. You will often want a diagram that shows tables and to what tablespaces they belong. Having them both on the same diagram allows for the ability to understand which tables are in which tablespaces and whether they are partitioned across multiple tablespaces as well.

A typed diagram also can be helpful. Having a diagram with only specific items allowed makes it easier for the modelers to see which elements can be used on the diagram. Some tools supply typed diagrams that give you only the elements that apply to that diagram according to the UML. For example, a diagram typed as «Database Diagram» would include tables, relationships, views, stored procedures, and domains on that particular diagram but would not allow for things like classes, objects, and other nondatabase elements.

**Database Design**

Designing the database through data modeling in the UML gives you the ability to capture many more items on the diagram visually than with traditional ER notations. You can model elements like domains, stored procedures, triggers, and constraints as well as the traditional tables, columns, and relationships. There are several reasons to model the database, including creation of a good design, enforcement of referential integrity, management of standards reuse, and communication of the database structures. With the models used for these many areas and more, it is a tremendous advantage to describe visually on the diagram as many things as possible.

Database design is an iterative process; using models makes the constant change easier to manage and understand, allowing you to make changes to the models prior to code generation so you can understand what the implications are based on each change and analyze whether the change should be made or something else should be chosen. For example, a change to one relationship or key column could affect several tables based on that one single change. Because of key migration and the cascading of changes based on the key migration, one small change can cause data loss and incompatibility.

In a perfect world, there would always be what is called model-driven development. In model-driven development, all changes are made to the model prior to being implemented directly in the database. By using model-driven development, you can eliminate most surprises. You can see what the changes will affect prior to making the changes in the database directly. The problem is in reality; say the DBA gets a call in the middle of the night about a problem in the database or a call in the afternoon about something that isn’t working or data that is missing. What does the DBA do? Go to the model, make the change, and then implement it? That’s what we would like to see, but that isn’t always
possible. The DBA has to get the job done and doesn’t have the time or desire to work in the model first, so he or she just changes the code and worries about the model later. This can cause problems that won’t be uncovered until later when the database and model are resynchronized, but this is something that really can’t be avoided.

The Case Study Status

Now that we have completed, at least for now, the logical design model for EAB Healthcare, we can move forward to the database design model and create a database design for the MDS in the UML. First it is important to understand the UML Profile for Database Design (see the “Concepts” section in this chapter). The Profile uses stereotypes and tagged values for all the information needed to describe the structures of the database and its elements. The Profile also uses constraints to enforce database design conformance. This includes many-to-many relationships not allowed on the database design, limiting them to the logical design.

EAB takes care of many residents, and since many of those residents receive government aid like Medicare and Medicaid, it is very important to keep a good record of all information on patients. If the residents’ records are not well organized and easily queried, the government can fine EAB or even shut down the company. EAB’s records are often scrutinized, so the creation of this new system will help satisfy many of the government’s needs and protect EAB in the future.

The Concepts

The following UML, object-oriented, and other concepts are cited in this chapter. For a more complete discussion and more rigorous definitions of UML concepts, refer to Booch et al. [1999]. In this chapter we also cover many concepts that are not discussed in UML books and specifications that currently exist; these are part of the information in the UML Profile for Database Design. A profile is an extension to the UML that keeps the UML metamodel intact. The Profile for Database Design adds stereotypes and tagged values that are attached to the stereotypes but does not change the underlying metamodel for the UML. The Profile also includes some icons to more easily visualize the database elements that are created and rules to enforce about the creation of a relational database design.

The UML Profile for Database Design

In this chapter, we cover the first part of the Profile, how to model the database structures. In the next chapter, we will cover some of the more physical data-
base elements involved in storage modeling. The database is created with tables, columns, and relationships. There are several elements that extend the database, like triggers, stored procedures, constraints, user-defined types (domains), views, and others. The Profile covers how to model all of these elements and where in the model they are defined.

**Diagram Elements**

The following diagram elements are described below. Figure 7–3 shows their associated icons.

**Table**—a grouping of information in a database about the same subject, made up of columns

**Column**—a component of a table that holds a single attribute of the table

**Primary key**—the candidate key that is chosen to identify rows in a table

**Foreign key**—a column or set of columns within a table that map to the primary key of another table

**Identifying relationship**—a relationship between two tables in which the child table must coexist with the parent table

**Non-identifying relationship**—a relationship between two tables in which each table can exist independently of the other

**View**—a virtual table that, from the user's perspective, behaves exactly like a typical table but has no independent existence of its own

**Stored procedure**—an independent procedural function that typically executes on the server

**Domains**—the valid set of values for an attribute or column

A table is a container composed of columns to store and organize the data in the system. A table is modeled as a class with the stereotype of table viewed as <<Table>>. The stereotype of table on class automatically causes all of the attributes on the class to become columns and automatically stereotypes the attribute as <<Column>>.

A column can be designated as a key or non-key column. A key column can be primary, foreign, or a combination of both primary and foreign. A primary key column is a column (or group of columns) that uniquely identifies its table or the row within its table. A foreign key is a column that was a primary key in a parent table that migrates to the child table and identifies the relationship between the tables. The foreign key can participate as a key or non-key column within the child table. The type of foreign key depends on the type of relationship involved.
between the two tables. An identifying relationship between tables makes the foreign key part of the primary key in the child table; a non-identifying relationship makes the foreign key a non-primary key column. There are icons that help to visualize the tables, columns, and type of keys (see Figure 7–4).

There are two basic types of relationships when modeling the database: identifying (also known as mandatory) and non-identifying. The identifying relationship means that the child table cannot exist without the parent table. An example of an identifying relationship is that between Order and Customer; without a customer, the order doesn’t exist. A non-identifying relationship occurs when the child table can live on its own. An example of a non-identifying relationship is that between Customer and Employee. A customer can exist without an employee, but often when an employee makes the sale that employee is assigned
to the customer. With new technologies of the Internet and older technologies of catalogs, you may not have an employee who sold the product; the customer may have purchased it without employee assistance. Therefore it is not mandatory that a customer exist only when associated to an employee. The identifying relationship is created with a stereotype of <<Identifying>> on a composite aggregation, and a non-identifying relationship is created with a stereotype of <<Non-identifying>> on an association.

When creating relationships, there is a need to define the relationship with cardinality. Cardinality is a numerical range defined on the relationship on how many times the relationship can occur. An example using cardinality is one customer can place one or more orders. The UML shows the cardinality directly on the relationship. For this example it would look like a 1 on the customer end of the relationship and 1..* on the order end of the relationship. Because it is a 1 and not 0..1 on the parent end of the relationship, this means that the relationship cannot be null and that each customer must have at least one order. You can also define roles on a relationship. A role describes the relationship textually. For this same example, the roles would be read as a customer places one or more orders, with places being the role on the customer (see Figure 7–5).

Database views are defined as a database component that behaves exactly like a table but has no independent existence of its own. A view is also known as a virtual table. Views may have different rules depending on the database server. For example, in Oracle a view can be updated, whereas in SQL Server Version 7 it cannot. A view is defined in the UML as a class with the stereotype of <<view>>. The view can be derived from one or more tables or views. A
view’s relationship to its parent tables and views is modeled with a dependency with the stereotype of <<Derived>> (see Figure 7–6).

Another one of the many advantages of using the UML for your database design is the ability to model standard elements that are not generally modeled in traditional ER notations but have great value in being modeled. One such element in the database design is the stored procedures. A stored procedure can be defined in different ways, again based on the database server. The stored proce-

Figure 7–5  Employee, Customer, and Order tables and their relationships
A stored procedure can be defined as a procedure within the database, from an external file or as a function. Some databases support the concept of a stored procedure package or container. A stored procedure container is a grouping of one or multiple stored procedures within a group and currently is supported in the Oracle 8i database. The container is modeled as a class with the stereotype of "<<SP Container>>". Within a "<<SP Container>>" are stored procedures, which are modeled as operations on the container with the stereotype of "<<SP>>". Another advantage of showing the procedures on the diagram is that you can visualize what the parameters are and the procedures dependency on tables within the database (see Figure 7-7).

When building a database design, it is important to enforce standards and enable ways to reuse elements as much as possible. The use of domains, also

![Diagram of database design elements](ch07.qxd_6/20/01_11:16_AM_Page_129)

**Figure 7-6**  Model elements for a view and its relationships to parent tables
known as user-defined datatypes, gives the modelers the power to reuse elements. A domain has all of the properties of a column except the column’s name. The domain is assigned to one or more columns and the column inherits all of the domain’s properties. If you change the properties of the domain, it will change the properties of all columns that inherit from that domain. A domain can be used as a standard to be reused within one or several models. A domain is a class that has the stereotype of \texttt{<<Domain>>}. A domain has attributes, which pertain to a column’s property. A composite domain can also be used, which is a class that has the stereotype of \texttt{<<Domain>>} but has multiple attributes. When a domain has multiple attributes and it is assigned to a column in a table, the additional attributes become columns as well. An example of a composite domain would be address. The domain address has attributes of street1, street2, city, state, country, and postal code. A table for customer has a column called address with the domain of address assigned to it. When the table is expanded for code generation, it will contain all of the columns of the
address domain. This will enable a standard address to be used everywhere in the model.

**Table and Column Elements**

A table can contain many items in addition to the columns that have already been described. In the same way, columns contain many elements that have to be described as well as some that appear directly on the diagram. The elements of tables and columns that appear in the diagram are

- **Constraints**—a rule that limits the value of or actions on the specified data field
- **Key**—a constraint that defines a type of key and its column(s)
- **Check**—a constraint that defines rules against the database
- **Unique**—a constraint that defines a column or set of columns as containing unique data
- **Trigger**—a code that tells the database what actions to perform
- **Index**—a file that enables faster data access
- **Datatype**—a type whose values have no identity
- **Precision**—the maximum number of digits allowed for a numeric data item
- **Scale**—the number of digits in the fractional part of a numeric data item
- **Length**—the maximum number of letters allowed for a character data item
- **Null**—a column that is not mandated to contain data
- **Not null**—a column that must contain data

Other elements that are part of the tables and columns as tagged values but may not appear in the diagram themselves are

- **Owner**—the creator of the database or specific element of the database
- **Comment**—a description in a database
- **Identity**—a column that has data automatically added by the database, generally for primary key columns

The UML supports the concept of tagged values. They can differ based on the stereotype of an element and the type of element. The stereotype inherits its own tagged values and can include some or all of the tagged values on the base element prior to the stereotype. As always, some of these tagged values vary depending on the database server, but within this book, we try to stay generic and stick to standard ANSI SQL rules.
There are three basic types of constraints and they can be assigned to a column, but generally they are owned by the table itself. The key constraint defines the primary and foreign keys for the table and can contain one or more columns. There are two different types of key constraints: primary key and foreign key. They are modeled as operations on a table with the stereotypes of <<PK>> and <<FK>> (see Figure 7–8). A check constraint is a rule on a column or table that can consist of calculations, a valid set of values, or a range of values. A check constraint is modeled as an operation on a table with a stereotype of <<Check>>. The last type of constraint is a unique constraint. A unique constraint designates that the data in a column or set of columns data must be unique. If a set of columns is grouped within a single unique constraint, the data when combined for the entire set of data must be unique, but the data within a single one of those columns are not required to be unique. A unique constraint is modeled as an operation on a table with the stereotype of <<Unique>>.

There are two types of triggers, one that is modeled and defined on a particular table and another that is part of a relationship to ensure proper referential integrity. This section covers the table-level triggers. They are created to trigger events to happen when something is done to a specific table. The trigger is database code that tells the database what other actions to perform after
certain SQL statements have been executed. The table-level trigger is modeled as an operation on a table with the stereotype of <<Trigger>>. A trigger can have many tagged values that get stored in the model, but they vary significantly based on the database server and version being used. You can create the necessary tagged values as needed based on your particular database needs.

The final operation that is modeled on a table is the index. An index is a pointer used to locate rows in a table rapidly. You can think of an index as the folder tabs in a filing cabinet—an index allows you to file information in a certain place and makes it very quick to find the information you are seeking because it organizes what you need in a single place. An index is modeled as an operation on a table with the stereotype of <<Index>>. There may be some tagged values on an index, for example, an index can be unique and therefore it would contain the tagged value of unique.

Datatypes, precision, scale, length, and nullability are tagged values on columns but may be displayed on the diagram as well. It is quite useful to visualize these tagged values on the diagram so that you can easily understand what the values of a column are. It is just as important to be able to not display these values on the diagram so that when using the diagram for specific needs, it is less cluttered, easier to read, and less technical for some of the nontechnical viewers. There are no specific stereotypes for tagged values, but the list of values available depends on several items, including the stereotype and to what database server the table is attached.

Additional tagged values that may not be as useful on the diagram but are very important to the model itself are owner for the table, comment on all elements of the model, and identity, which is available to many databases as a rule on a column that automatically generates a sequence of unique values for a key column. Different databases call identity something different, but the concept is similar, so we have chosen one word to describe them all. An identity column generally can be an assigned tagged value of only certain datatypes and should be available for use only with those datatypes.

**The Approach**

The typical approach to moving into the actual modeling of the database is to use all of the classes that were already captured and understand how they will become tables. At this point the classes that will be transformed into tables should have been marked with a tagged value of persistent. This tagged value means that the class requires data and that the data for this class persists in the database. Therefore, the database must contain the needed structures to support the class. Having already worked through the requirements and an understanding of what is being built and how to describe it from both a business and
a technical sense, it is much easier to determine what tables to create, the data
and descriptions of those tables, and how to construct them.

Although the database design team has been involved with the previous
requirements gathering and modeling of the needed elements, it is now time to
look at the model from strictly the database point of view. The database design-
ers look at the needs of the database from a more physical view. It is not yet
quite the time to look at specific database server requirements, but we begin to
understand what would make this a good optimized data model. For example,
do all of the tables have unique identifiers? Is referential integrity being en-
forced through key migration? Are there rules within the database, from the
earlier requirements, that may cause a need for constraints and stored proce-
dures? Employ corporate and group standards to ensure that the database
model meets the rules of the organization.

Since we are using the UML, an understanding of how models are built in
the UML and perhaps a tool that supports database design using the UML are
very helpful at this point. You can use the UML and any tools that support the
creation of stereotypes to begin this process, but it is always easier if you al-
ready have the stereotypes and database knowledge so that the tool can get out
of your way and the database designers can just start working on the model.
Knowledge of the entire UML itself is quite helpful at this point as well, allow-
ing the modelers not only to build their database models but also to use the
existing models to understand what has been done and described so that the
modelers can continue to use the requirements and definitions for those re-
quirements that have already been described.

When building a database model, it is beneficial not to concern yourself
with the physical aspects of the database until you are sure that all of the re-
quirements have been captured and modeled as database elements. For this rea-
son, we do not cover the physical storage of the data here, saving example
schemas, tablespaces, databases, and other items until the next chapter. This
chapter focuses on how to build the database model using the UML without a
focus on implementation. However, the database designers begin to think
about the implementation with the creation of views to store data for better
access, constraints to enforce rules that were uncovered during the earlier
processes as well as during database modeling, indexes based on query pat-
terns that may have been uncovered in sequence diagrams, and stored proce-
dures and triggers to enforce rules that have been identified.

The Design

The database design team first focuses on creating tables from existing classes.
This ensures that all of the classes that were created and marked as persistent
Based on the requirements that were gathered are moved forward into the database. Once that has occurred, the team can start looking for ways to optimize the database, beginning with how to handle tables that were created based on inheritance relationships in the class model and classes that took part in many-to-many relationships that have to be split using association tables. From there the database design team will begin to ensure uniqueness of the tables and enforcement of such items as rules using constraints on the database. The database design team at EAB Healthcare doesn’t really work much with triggers and stored procedures. They are handled primarily by the database administrators and will therefore be worked on later, after the database design model has been fairly locked down.

Creating Tables from Classes

EAB has done a good job of capturing the requirements and turning them into classes. Working together, the development, business, and database teams have gone through each of the classes and marked the ones that are needed as persistent. The first job for EAB’s database designers in creating tables from classes is to decide what packages to work with and to begin the transformation to tables.

The database designers for EAB begin to move items into the database, working first with some of the actors that have information required for the MDS. Based on the government requirements, EAB has identified three primary MDSs that EAB has to be concerned with: the Background MDS, the Basic Assessment MDS, and the Full Assessment MDS. Each of these is important and feeds information to the other. When building the database, we want to begin with the lowest common denominator and build from there. This ensures that the basics are covered. As the database designers continue gathering the information that needs to be captured, they can determine whether each item needs its own table, can fit into an existing table, or is already being captured for another purpose.

The database team looks at the requirements for each MDS type to determine what information is needed to complete each type. It is almost like thinking of each MDS as some type of form or screen and considering what information is needed to fill in that screen and what is the best way to organize the information in table and column structures to store the information.

Background Information

The first set of information captured about a patient in the initial MDS comes from the background information obtained at admission. The Background MDS gathers the initial information from an incoming resident as he or she registers at EAB and can be used to fulfill much of the more detailed MDS later.
The background information is used to understand who the resident is and some of his or her habits, problems, and reasons for coming to EAB. Based on the needs of the Background MDS, the information is broken into two categories:

1. Demographic information
2. Customary routine

The database design team creates a set of tables based on the typical information needed for these two categories, gathering the appropriate information from the classes and actors that already exist. The Resident actor fulfills most of the demographic information needed and the customary routine information is gathered from various other classes. Figure 7–9 shows the first database diagram the team creates to support the Background MDS.

For the transformation from the logical design model to the database design model, the team creates and uses some domains, as shown in Figure 7–9. The primary domain is Boolean and is mapped to a CHAR(2). Also attached to that domain is a check constraint that has a valid value of y or n. This helps to enforce that the logical type of Boolean is transformed correctly to a database type of CHAR(2) and has rules to enforce its properties.

**Basic Assessment Tracking**

The next set of information that is tracked and may have a set of circumstances for when it is used to feed the Full Assessment MDS is the Basic Assessment Tracking (BAT). Some of the information for the BAT will come from tables that have already been created in the Background MDS and some will be new. The database designers for EAB create diagrams to demonstrate each type of form. Tables can be included on many diagrams but exist only within one model. A diagram is a basic way to display information visually to make it easier to understand and group in a logical way.

The EAB database designers now focus on the BAT and transform classes to tables that pertain to the BAT. There is really only one class that becomes a table for the BAT, the Basic Assessment MDS class. The other table used in the diagram is the Resident table (see Figure 7–10).

**Full Assessment Data**

The final set of information needed to complete all of the information for the different types of MDSs is the Full Assessment MDS. Just like the previous sets of information, the Full Assessment MDS includes some information already
The EAB database designers begin looking at the information already captured in the logical design that is needed to create the Full Assessment MDS and decide what to transform into the database design as tables and columns. Once the database designers have completed their assessment of the classes to become tables, they create the tables as needed and make sure that column properties are correct based on the business rules already uncovered and the needs of the database design.
The Full Assessment MDS contains many tables, columns, and relationships—everything from the prior MDSs as well as everything from its own MDS—and therefore is quite crowded. The EAB database designers decide to build several diagrams to display the Full Assessment MDS to make it easier to read and understand. One diagram has a complete overview of the entire MDS but contains only table names and relationships (Figure 7–11). One diagram contains just the new tables that are involved in the Full Assessment MDS and not the tables involved in other MDSs (Figure 7–12). The database designers create several other diagrams for different views as well.

The database designers make several changes to the model to help make it more easily read and optimized. First they create alternate keys for the Background MDS, Full Assessment MDS, and Basic Assessment MDS, enabling the designers to migrate the alternate keys rather than all of the primary keys. This is a big change to the model because of all the identifying relationships from the child tables to each of the different types of MDS tables; there would have
Figure 7–11  Complete overview diagram of the entire MDS
### Figure 7–12  Tables that are included only in the Full Assessment MDS
been many foreign keys migrated since each had become a primary and foreign key in the child MDS tables. Using the alternate keys instead of all primary keys means that only one column migrates as a foreign key from the MDS type tables to the MDS table (see Figure 7–13).

The database designers begin to look at the way tables will be queried so that they can include indexes on the tables as needed. There is a lot of information captured in the MDS tables and therefore a lot of data will be required in different queries. The EAB database designers look at each table individually to determine what indexes need to be created. (For this example we look at just the MDS table and uncover the indexes that are needed for it.) The EAB database designers decide to create a few different indexes, one for each of the foreign keys and one composite index that includes both the creationDate and lastUpdate columns (see Figure 7–14). Some databases automatically create indexes for each foreign key, but since EAB has not yet chosen a platform, the database designers include indexes on the model to ensure that the indexes will be created at database generation time.

To ensure that the data collected within the database is correct, the database designers create check constraints on columns and tables. The check constraints are restrictions on the database that are enforced with the implementation of the rule within the constraint. For this example we again focus on only one table, although many tables and columns will have check constraints assigned to them. Since the Resident table has exposure to almost every facet of the database design, we have chosen to work on this table for the example. Some of the constraints can be reused in many different tables as long as the constraints apply to the context. The constraints may be created based on domains that are assigned to the columns within the table, and others are assigned specifically to that column but may be reused on a more global basis.

There are four check constraints on the Resident table: Gender, Birthdate, MedicareResident, and MaritalStatus (see Figure 7–15). Gender is a valid value type of check constraint with the valid values of M or F. Birthdate is a constraint with a rule that enforces that the birthdate must be prior to the admission date. MedicareResident and MaritalStatus are also valid value check constraints. MedicareResident will have the valid Boolean values of Y or N and MaritalStatus values will be M, D, S, or W.

**Database Views**

Database views are a physical implementation of virtual tables used for reasons including performance and security. The database designers at EAB choose to use views so that they can secure information within certain tables that many different people need to access yet make some of the very secure information
Figure 7–13  Background MDS, Full Assessment MDS, Basic Assessment MDS, and MDS tables showing that only the primary keys have migrated and not the primary/foreign keys.
available to only a few people. The other set of views takes information that is often queried from various tables at the same time and puts it together in a view as a virtual table to enhance performance when querying (thus the \texttt{SELECT} statement can query one place and get all needed information). There are many views in the database, but we again focus on only a few.

The view that the EAB database designers start with is on the Resident table and is called the PublicResidentView (see Figure 7–16). This view prevents most users from viewing such information as the patient’s social security number, education level, and marital status. There are many groups in EAB Healthcare that do not need this information (and having this information can possibly cause bias toward the patient), so the view excludes these columns. Another view is based on the Background MDS for the information that is needed most often by the accounts receivable team members. They do not need all the information, only some from many tables within the Background MDS and its related tables that supply information for the Background MDS. Therefore, it is much easier and quicker for the application to query one place to get the needed information. This view is called ARBackgroundView (see Figure 7–17).
Figure 7–15  Resident table showing columns and constraints
Figure 7–16  PublicResidentView as it is associated to the Resident table
Figure 7–17  ARBackgroundView as it is associated to the Resident and BackgroundMDS tables
This chapter reviewed the reasons to build a database design, specifically in the UML, and explored in detail the Profile for Database Design created by Rational Software Corporation, although this information is mainly an introduction. The next chapter goes into more detail on the Profile and how to use the UML for deployment and storage of the database elements. The UML has great benefits for database designers: it brings teams together, exposes elements directly on the model that normally get hidden behind modeled elements as tagged values, and describes the database design in great detail. By leveraging work already done in both the requirements and the logical design phases, the database design team at EAB was able to get a quick start on building tables and columns and on understanding the requirements and business reasons for creating the database elements. The constraints needed and ways the database would be used were quite evident because of the work done up front to uncover the needs of people using the database and how it would be used. Using the UML has been quite valuable to all teams so far and now that the database design is based on the logical design. The application developers are working on the application design at the same time based on the same artifacts. When the time comes, it will be more intuitive for the teams to build the object relational mappings and to create the queries of the database.