Using Data Modeler

Rational Rose® 2001

VERSION: 2001.03.00

PART NUMBER: 800-023926-000

support@rational.com
http://www.rational.com

Rational®
the e-development company™

Copyright Notice

Copyright © 2000 Rational Software Corporation. All rights reserved.

THIS DOCUMENT IS PROTECTED BY COPYRIGHT AND CONTAINS INFORMATION PROPRIETARY TO RATIONAL. ANY COPYING, ADAPTATION, DISTRIBUTION, OR PUBLIC DISPLAY OF THIS DOCUMENT WITHOUT THE EXPRESS WRITTEN CONSENT OF RATIONAL IS STRICTLY PROHIBITED. THE RECEIPT OR POSSESSION OF THIS DOCUMENT DOES NOT CONVEY ANY RIGHTS TO REPRODUCE OR DISTRIBUTE ITS CONTENTS, OR TO MANUFACTURE, USE, OR SELL ANYTHING THAT IT MAY DESCRIBE, IN WHOLE OR IN PART, WITHOUT THE SPECIFIC WRITTEN CONSENT OF RATIONAL.

U.S. Government Rights Notice

U.S. GOVERMENT RIGHTS. Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational License Agreement and in DFARS 227.7202-1(a) and 227.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct 1988), FAR 12.212(a) 1995, FAR 52.227-19, or FAR 52.227-14, as applicable.

Trademark Notice

Rational, the Rational logo, and Rational Rose are trademarks or registered trademarks of Rational Software Corporation in the United States and in other countries.

Visual C++, Visual Basic, and SQL Server are trademarks or registered trademarks of the Microsoft Corporation. Java is a trademark of Sun Microsystems Inc. DB2 is a trademark of the IBM Corporation. All other names are used for identification purposes only and are trademarks or registered trademarks of their respective companies. Portions of Rational Rose include source code from Compaq Computer Corporation; Copyright 2000 Compaq Computer Corporation.

U.S. Registered Patent Nos. 5,193,180 and 5,335,344 and 5,535,329. Licensed under Sun Microsystems Inc.'s U.S. Pat. No. 5,404,499. Other U.S. and foreign patents pending. Printed in the U.S.A.

Contents

Preface xi
Audience xi
Other Resources xi
Contacting Rational Technical Publications xii
Contacting Rational Technical Support xii
1 Introduction: Unifying the Team 1
Team Roles 1
Business Analyst Role 1
Application Designer Role 1
Database Designer Role 2
Role Dependencies 2
The Data Modeler Solution 3
2 UML and Data Modeling 5
Contents 5
UML Introduction 5
Why UML for Data Modeling? 5
Data Modeling Profile Added to UML 5
Advantages of the UML Data Modeling Profile 6
Advantages of Rose UML and Data Modeling 7
The Data Model Diagram 7
Reusing Data Modeling Elements 8
Data Modeler's Transformation and Engineering Features 9
3 Logical Data Modeling 11
Contents 11
Introduction 11
Using Rose for Logical Data Modeling 11
Class Diagram 11
Standardized Notation 12
Mapping Capabilities 12
Mapping an Object Model to a Data Model 13
Mapping Components 13
Mapping Operations 13
Mapping Packages to Schemas 14
Mapping Classes to Tables 14

Mapping Attributes to Columns 14
Mapping Composite Aggregations to Identifying Relationships 17
Mapping Aggregations and Associations to Non-Identifying Relationships 18
Mapping Association Classes to Intersection Tables 20
Mapping Qualified Associations to Intersection Tables. 21
Mapping Inheritance 22
Transforming the Object Model to the Data Model 24
Why Transform 24
The Transformation Process. 24

4 Physical Data Modeling 29

Contents 29
Introduction 29
Data Models 29
Building a New Data Model. 30
Create a Database 30
Create a Schema 30
Create a Data Model Diagram 30
Create Domains 30
Create Tables 31
Create Columns 32
Create Constraints 33
Create Relationships 35
Define Referential Integrity 40
Create Custom Triggers 41
Create Stored Procedures 42
Reverse Engineering to Create a Data Model 44
Reverse Engineering Wizard 44
Reverse Engineering DB2 Databases or DDL 44
Reverse Engineering Oracle Databases or DDL 44
Reverse Engineering SQL Server Databases or DDL 45
Reverse Engineering Sybase Databases or DDL. 45
After Reverse Engineering 45
After Building the Data Model 45

5 Mapping the Physical Data Model. 47

Contents 47
Introduction 47
Mapping the Data Model to an Object Model 47
Mapping Schemas to Packages 47

Mapping Domains to Attribute Types 48
Mapping Tables to Classes 48
Mapping Columns to Attributes 49
Mapping Enumerated Check Constraints to Classes. 49
Mapping Identifying Relationships to Composite Aggregations 50
Mapping Non-Identifying Relationships to Associations 51
Mapping Intersection Tables 52
Mapping Supertype/Subtype Structures to Inheritance Structures 55
Transforming a Data Model to an Object Model 56
Why Transform 56
The Transformation Process 56
6 Implementing the Physical Data Model. 59
Contents 59
Introduction. 59
Forward Engineering a Data Model 59
Forward Engineering Wizard 59
Forward Engineering to the ANSI SQL 92 DDL 60
Forward Engineering to a DB2 Database or DDL 61
Forward Engineering to an Oracle Database or DDL. 61
Forward Engineering to a SQL Server Database or DDL 61
Forward Engineering to a Sybase Database or DDL. 62
Comparing and Synchronizing a Data Model 62
Synchronization Wizard 63
A Appendix 65
UML Data Modeling Profile. 65
B Appendix 67
Object to Data Model Data Type Mapping 67
C Appendix 71
Data to Object Model Data Type Mapping 71
D Appendix 79
Database Connections 79
DB2 79
Oracle. 80
SQL Server 80
Sybase 81

Index 83

Figures

Figure 1 A Data Model Diagram in Rose 8
Figure 2 A Class Diagram in Rose. 12
Figure 3 Classes Map to Tables. 14
Figure 4 Attributes Map to Columns 15
Figure 5 ID-based Columns. 16
Figure 6 Domain Columns 17
Figure 7 Composite Aggregations Map to Identifying Relationships 18
Figure 8 Associations Map to Non-Identifying Relationships. 19
Figure 9 Many-to-Many Associations Map to Intersection Tables 20
Figure 10 Association Classes Map to Intersection Tables 21
Figure 11 Qualified Associations Map to Intersection Tables 22
Figure 12 Inheritance Maps to Separate Tables 23
Figure 13 Transform Object Model to Data Model Dialog Box. 25
Figure 14 A Domain. 31
Figure 15 A Table. 32
Figure 16 An Identifying Relationship 35
Figure 17 A Non-Identifying Relationship 36
Figure 18 A Role 38
Figure 19 An Intersection Table 39
Figure 20 A Self-Referencing Relationship 40
Figure 21 Domain Columns Map to Attributes. 48
Figure 22 Tables Map to Classes. 49
Figure 23 Columns Map to Attributes 49
Figure 24 Enumerated Check Constraints Map to Classes with <<ENUM>> 50
Figure 25 Identifying Relationships Map to Composite Aggregations 51
Figure 26 Non-Identifying Relationships Map to Associations. 52
Figure 27 Intersection Tables Map to Many-to-Many Associations 53
Figure 28 Intersection Tables Map to Qualified Associations 54
Figure 29 Intersection Tables Map to Association Classes 55
Figure 30 Transform Data Model to Object Model Dialog Box. 57

Tables

Table 1 Cardinalities for Foreign Key Constraints. 37
Table 2 UML Data Modeling Profile Stereotypes 65
Table 3 Analysis Object to Data Model Data Type Mapping 68
Table 4 Java Object to Data Model Data Type Mapping. 69
Table 5 Visual Basic Object to Data Model Data Type Mapping 70
Table 6 SQL 92 Data Model to Object Model Data Type Mapping. 72
Table 7 DB2 Data to Object Model Data Type Mapping. 73
Table 8 Oracle Data to Object Model Data Type Mapping 74
Table 9 SQL Server Data to Object Model Data Type Mapping 75
Table 10 Sybase Data to Object Data Type Mapping. 77

Preface

Rational Rose®, hereafter referred to as Rose, is a comprehensive, integrated programming environment that supports the development of complex software systems. This manual presents the concepts needed to use specific functionalities of Rose in a data modeling environment.

Audience

This manual is intended for:

- Database developers and administrators
- Software system architects
- Software engineers and programmers
- Anyone who makes design, architecture, configuration management, and testing decisions

This manual assumes you are familiar with database modeling concepts and the life-cycle of a software development project.

Other Resources

- Online Help is available for Rational Suite.

  From a Suite tool, select an option from the **Help** menu.

- All manuals are available online, either in HTML or PDF format. The online manuals are on the Rational Solutions for Windows Online Documentation CD.
- For more information on training opportunities, see the Rational University Web site: http://www.rational.com/university.

Contacting Rational Technical Publications

To send feedback about documentation for Rational products, please send e-mail to our technical publications department at techpubs@rational.com.

Contacting Rational Technical Support

If you have questions about installing, using, or maintaining this product, contact Rational Technical Support as follows:

Your Location	Telephone	Facsimile	E-mail
North America	(800) 433-5444 (toll free) (408) 863-4000 Cupertino, CA	(781) 676-2460 Lexington, MA	support@rational.com
Europe, Middle East, Africa	+31 (0) 20-4546-200 Netherlands	+31 (0) 20-4545-201 Netherlands	support@europe.rational.com
Asia Pacific	+61-2-9419-0111 Australia	+61-2-9419-0123 Australia	support@apac.rational.com

Note: When you contact Rational Technical Support, please be prepared to supply the following information:

- Your name, telephone number, and company name
- Your computer's make and model
- Your operating system and version number
- Product release number and serial number
- Your case ID number (if you are following up on a previously-reported problem)

Introduction: Unifying the Team

1

What truly makes a team is a group of people working together to accomplish a common overall goal. Sometimes the team consists of only two people, other times it consists of hundreds, but regardless of the number, the team must be unified in its efforts. A development team is a specialized team in which the various members hold different responsibilities in the development life cycle. In solving business problems, there are three distinct roles in a development team—the business analyst, the application designer, and the database designer.

Team Roles

For the scope of this book, the role of business analyst is performed by business-process analysts, systems analysts, and those who capture and review requirements for a business process. The role of application designer is performed by application developers, software engineers, and those who design and review application designs. The role of database designer is performed by data developers, database administrators (DBAs), data analysts, and those who design and review relational databases.

Business Analyst Role

In the development process, the business analyst's responsibility is to interview the end-user or client to understand the overall business problem, analyze the business as it currently is, identify the defects in the current processes that are causing the overall business problem, and model the business as it could be. The business analyst captures the end-users requirements and the process improvements in business models. Business models divide the process into components of events, people or things, and order the process using use-case, sequence, and activity models.

Application Designer Role

The responsibility of the application designer is to generate code and build an application based on the business analyst's business models. The application designer uses object-oriented conceptual models and creates classes from the use cases, organizing the classes into an object model structure using class models. Then the

application designer assigns the class model structure to a component and generates code, building an application. The application designer is also responsible for creating classes that can access the data in the database.

Database Designer Role

The responsibility of the database designer is to provide a storage container for all data applying to the built application, and provide a method to maintain the data structure's integrity, based on the business analyst's business models. The database designer uses logical and physical data models; creates schemas, tables, and other database elements from the use cases or classes; and organizes the schemas, tables, and database elements into a data model structure. Then, using the data model structure, the database designer selects a database management system (DBMS) and generates a data-defined language (DDL) script, building a database. The database designer must also ensure the application designer can map the application classes to the correct tables to access the data for the application.

Role Dependencies

Due to their differences in responsibilities and knowledge, each team member solves a business problem using a different method. The business analyst solves the problem by revising the processes; the application designer solves the problem by generating code; and the database designer solves the problem by controlling the database and data accepted into it. For any one of these methods to solve the overall business problem, the other two methods must be included in the planning.

A dependency exists between each of the different roles to solve the overall business problem. Modeling the business processes alone, does not physically solve the business problem. Generating code is useless without the database to host the data. Data is meaningless without an application to access it. Each role is dependent on the other; no one role can completely solve the business problem. The application designer and database designer need the business models with the modeled requirements. The business analyst needs to confirm that the database design meets all the required business rules. If the application designer needs an additional field, the database designer needs to illustrate what impact such a change would have on the database structure.

These dependencies are especially evident in iterative development, where change is constant and communication between the teams is essential. However, when each of these roles is using a different notation, communication is difficult. A unified language between the teams can reduce miscommunication and development time while improving quality.

The Data Modeler Solution

Rose unified the roles of business analyst and application designer in the previous releases of the Rose software. With the addition of Data Modeler, Rose unifies all three roles of the development team, allowing them to communicate freely through their individual models using the common language of UML.

UML and Data Modeling

2

Contents

This chapter is organized as follows:

- *UML Introduction* on page 5
- *Why UML for Data Modeling?* on page 5
- *Advantages of Rose UML and Data Modeling* on page 7

UML Introduction

Teams need one tool, one methodology, and one notation. Since its inception, the Unified Modeling Language (UML) has been unifying members of a development team for faster and higher quality applications. But UML allowed only business analysts and application designers to communicate with each other, database designers were excluded because they used a different kind of notation. Rose adds a UML profile to accommodate entity/relationship (E/R) notation with the addition of Data Modeler, allowing the database designer to communicate with the business analyst and application designer, making the UML a truly unifying language.

Why UML for Data Modeling?

The UML offers a standard notation very similar to Peter Chen's E/R notation. Like Chen's E/R the UML is based on building structures using entities that relate to one another. By adding the data modeling profile, UML's ability to model an entire system is increased, allowing you to model not only logical models, but physical data models too, mapping your applications and your databases.

Data Modeling Profile Added to UML

A UML profile is an Object Management Group (OMG) approved method of adding to UML for a specific subject, without altering the UML metamodel. UML profiles added to UML use customized stereotypes and tagged values based on their subject's concepts and terminology.

The UML data modeling profile allows you to model databases based on data modeling stereotypes added to existing UML structures. Stereotypes added to UML structures such as components, packages, and classes allow you to model databases, schemas, and tables. Stereotypes added to UML associations allow you to model relationships. Strong relationships are modeled with the addition of stereotypes to composite aggregations. Finally, stereotypes added to UML operations allow you to model primary key constraints, foreign key constraints, unique constraints, and additional database concepts such as check constraints, triggers, and stored procedures. Refer to *Appendix A* for a listing of database concepts and the UML data modeling profile stereotypes.

Advantages of the UML Data Modeling Profile

The UML data modeling profile has four distinct advantages consisting primarily of its compatibility with business modeling and applications. First, the UML focuses on the overall architecture of the system allowing you to model high-level business processes, applications, and their implementation.

Second, the UML separates logical and physical design, making mapping your database to your application, and change management easier. When you separate the physical design from the logical design, you can customize the physical design to accommodate your specific database management system (DBMS) and appropriate levels of normalization, while the logical design remains a high-level design appropriate for an overall view of your database or application. The separate designs also allow you to see the effect changes have on the design before the changes are committed. The change may be applied easily to the logical design, but specific DBMS structures may restrict the same change when it is applied in the physical model; therefore, the change would not be acceptable for the physical design.

Third, the UML addresses behavior modeling, allowing you to model operations and constraints, including business rules.

Finally, the UML is compatible with object-oriented notation. It is the difference in the object-oriented notation for some applications and E/R notation for databases that hinders communication between the database designer and the other team members. Database designers are isolated, often excluded from the communication of crucial design decisions, and not able to communicate to other team members design decisions that they themselves make. The UML eliminates this communication failure by allowing all the team members to communicate their design decisions in the same notation. This UML data modeling profile enables Rose to create a UML-based data model.

Advantages of Rose UML and Data Modeling

Rose UML offers distinct advantages when data modeling. These advantages are the introduction of the data model diagram to Rose's set of UML diagrams, reusability of data modeling elements, and Data Modeler's engineering capabilities.

The Data Model Diagram

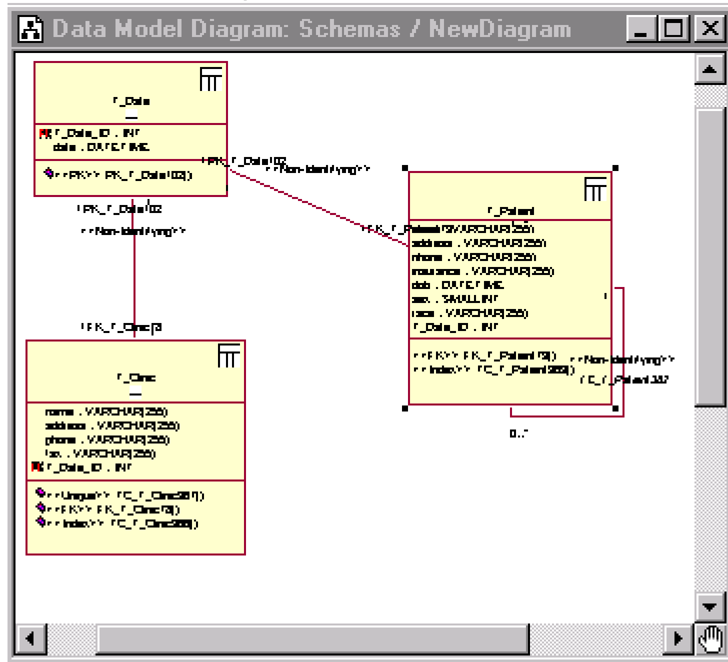
For each type of development model—that is business models, application models, and data models—Rose uses a specific type of diagram.

- Business models—Use case diagram, activity diagram, sequence diagram
- Application models or logical data models— Class diagram
- Physical data models—Data model diagram

An advantage the data model diagram offers database designers is the ability to model using terms and structures already familiar to them, such as columns, tables, and relationships. Also, the data model diagram completes Rose's set of diagrams to model the whole system, closing the chasm in development between database designers and application developers.

The data model diagram visually represents the physical data model, so to work in your physical data model you must create a new or activate an existing data model diagram. Rose provides this additional diagram to reduce the confusion between object model and data model items, and to support features unique only to Data Modeler such as supported DBMSs, key migration, and schema migration.

Figure 1 A Data Model Diagram in Rose



Reusing Data Modeling Elements

Another advantage of using Rose is it enables you to reuse your modeling elements to apply to other modeling needs. You can do this by using *frameworks*. Frameworks are files that act as templates for other models. You can create a framework based on your entire .mdl file using the Frameworks Wizard. Because frameworks work as a template, you can create standard frameworks used for your business. Frameworks are especially useful when the model contains domains, because domains can enforce your business standards at the column level. It is the combination of frameworks containing standard entities and domains that act as a foundation to enforce specific standards of your business. For example when modeling the business process of a clinic you need certain entities like patient, physician and clinic, but you will also need to specify that each patient's first name cannot exceed 20 characters and each physician must have a social security number of numeric value and exactly 9 digits in length. Creating a framework that contains these entities and domains that support these standards will allow you to reuse these standards repeatedly.

Data Modeler's Transformation and Engineering Features

With the UML data modeling profile, Data Modeler can map from the data model diagram known as the *data model* to the class diagram known as the *object model* and vice versa. This mapping enables Data Modeler's transformation and engineering features to create a round-trip engineering effect.

Transformation Between the Object Model and Data Model

The relationship between logical classes and physical tables provides the basis for the mapping between a logical data model and a physical data model. This mapping automatically occurs when you use the Transform Object Model to Data Model or Transform Data Model to Object Model features. These features map the classes and tables in a one-to-one mapping, with the exception of denormalization issues and DBMS restrictions.

Forward and Reverse Engineering the Data Model

The relationship between logical classes and physical tables can also act as a mapping between a DBMS and an object-oriented language like Java or C++, using forward engineering or reverse engineering features. When you reverse engineer a DBMS schema to a data model and then transform that data model to an object model, the object model transforms to the Analysis language. By reassigning the language of your object model from Analysis to Java or C++, Rose maps your DBMS database to an application model that can be used to build an application. The same process applies to mapping the application to the database— the object model that built the application using a C++ language, can be reassigned to the Analysis language. Then that object model can be transformed to a data model, and that data model can be forward engineered to create a schema in a DBMS.

Comparing and Synchronizing the Data Model

If you are working with a legacy DBMS database and application, you can still use these round-trip engineering features. However, instead of forward engineering to a DBMS, you can use Data Modeler's Compare and Synchronization feature to update your existing DBMS database, synchronizing your DBMS database with the transformed object model that built the application.

All of Data Modeler's features working together create a round-trip engineering effect with the mapping of logical classes to physical tables being the key to mapping the database to the application.

Logical Data Modeling 3

Contents

This chapter is organized as follows:

- *Introduction* on page 11
- *Using Rose for Logical Data Modeling* on page 11
- *Mapping an Object Model to a Data Model* on page 13
- *Transforming the Object Model to the Data Model* on page 24

Introduction

Logical data modeling is an essential step in modeling a database. The logical data model gives an overall view of the captured business requirements as they pertain to data entities. You can use Rose for logical data modeling and customize your logical data model to transform to a physical data model.

Using Rose for Logical Data Modeling

The previous chapter discussed the advantages of using Rose and the UML in general for data modeling. This chapter discusses the advantages of using Rose and the UML for logical data modeling. These advantages are Rose's ability to graphically depict a logical data model using the class diagram, and Rose's standardized notation and mapping capabilities.

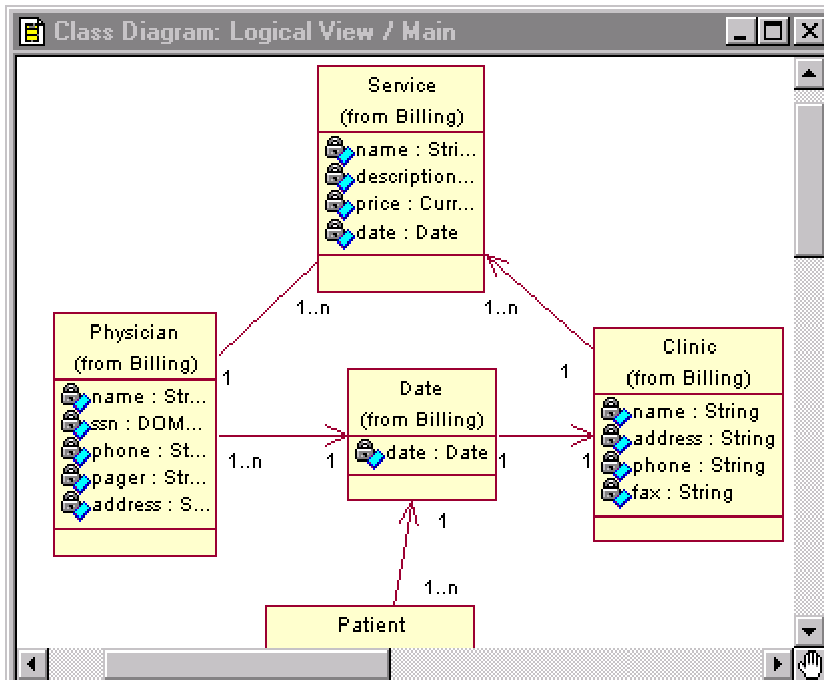
Class Diagram

The class diagram can graphically depict a logical data model because it uses a structure similar to a logical data model. When you create a logical data model, you start by identifying high-level entities. The class diagram also identifies high-level entities. In UML terminology these entities are called *classes*. Rose allows you to assign the stereotype <<entity>> to these classes for further distinction.

The next step is to assign attributes to these entities to identify them to the system. This is the same step you take when modeling in the class diagram; you assign attributes to the classes.

The final step that both the logical data model and the class diagram share is to relate these entities or classes to other entities or classes using *associations*.

Figure 2 A Class Diagram in Rose



Standardized Notation

Another advantage is Rose uses the common notation of the UML. As was discussed in *Chapter 2, UML and Data Modeling*, the UML data modeling profile added stereotypes to UML elements relating to data modeling terminology. These same UML elements are used in the class diagram and with these stereotypes you can understand the mapping from the logical data model to the physical data model.

Mapping Capabilities

In Rose data modeling terminology, a class diagram is also known as an object model. Object models serve two purposes. A class diagram or object model can act as a logical data model, but an object model can also act as a model to capture a conceptual

view of an application. An object model is necessary if you want to map an application to a database. It is the object model mapping to the physical data model that is the basis for mapping the application to the database.

Mapping an Object Model to a Data Model

Rose allows you to take a step beyond identifying high-level entities, attributes, and associations. It allows you to create a robust logical data model that can map more precisely to a physical database. Customizing the object model for your database helps to manage change, thereby decreasing the impact a change of requirements can have on the existing model. If the object model maps to the data model, and changes or enhancements are made to the object model, the same changes or enhancements can be applied to the data model. You can model your object model specifically to map to a database by modeling your object model elements—with the exception of components and operations—to map to data model elements.

Mapping Components

Components represent the actual application language. In Rose terminology, a component represents a software module (source code, binary code, executable, or DDL) with a well-defined interface. According to the UML Data Modeling profile, a component maps to a database, but this mapping is only for reference purposes; in actual object model to data model transformation, components are ignored.

Although Rose gives you the option of several component languages to assign to your logical package, the Data Modeler add-in is compatible with only three of those component languages—Java, Visual Basic, and Analysis. The classes that you want to map to tables in the data model must use one of these three component languages to be transformed to a data model. If you want to use a component language not compatible with Data Modeler, it is recommended you create a separate object model using your desired component language, and map that model to an Analysis object model that is used for transformation purposes.

Mapping Operations

According to the UML Data Modeling profile, operations map to various constraints; however, just like components, operations are ignored in the transformation process. Operations are the behavior of a class, and can be useful to database designers because they can be used as a basis for identifying index items, possible triggers, and other constraints.

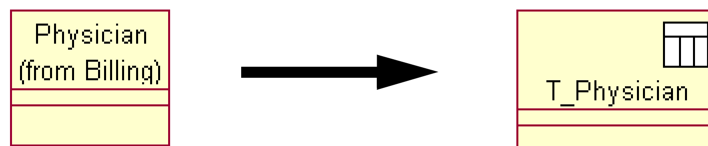
Mapping Packages to Schemas

In a class diagram, a *logical package* is an optional element, but when using Data Modeler, a logical package is required. A logical package is considered to be the primary container of your object model, so it is the level at which Data Modeler initiates the transformation process. You must group your classes in a logical package to transform them to a data model. Data Modeler allows you to transform one logical package at a time and generates one schema for each logical package transformed.

Mapping Classes to Tables

Classes are high-level entities that can have two states of existence—transient and persistent. It is the persistent classes that map to physical tables, because persistent classes can work as persistent data storage, existing even after the application has completed its process. All persistent classes can map to tables using a one-to-one mapping, unless you are using an inheritance structure in your model, then the mapping could be a one-to-many mapping.

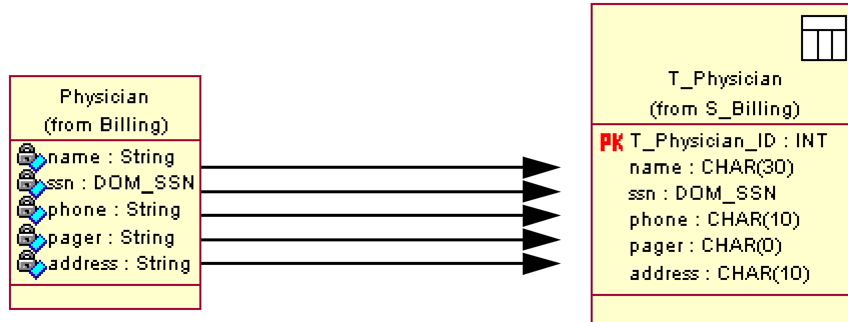
Figure 3 Classes Map to Tables



Mapping Attributes to Columns

Persistent attributes map to columns in a one-to-one mapping. In your model, you may have multiple attributes that map to one column, but in the transformation, Data Modeler transforms one-to-one for every attribute. Data Modeler ignores non-persistent attributes like derived values.

Figure 4 Attributes Map to Columns



Rose allows you to customize your attribute mapping by mapping specific attributes to speciality columns like primary key columns, ID-based columns, or domain columns, and to map attribute types to specific DBMS data types.

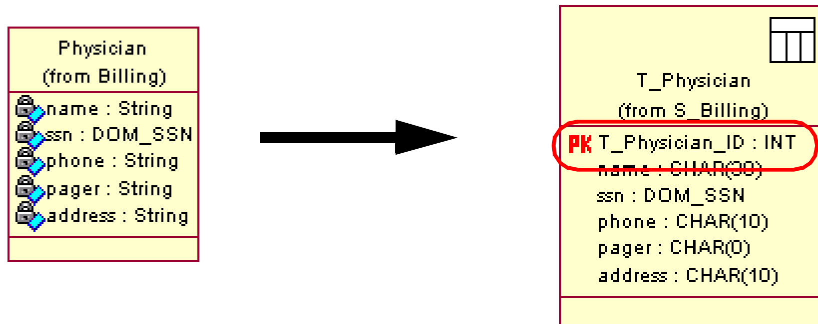
Primary Keys

You can map an attribute to be a primary key column by assigning the attribute to be a *candidate key*. A candidate key is an attribute tagged as part of object identity, which Data Modeler transforms to a primary key in a table during the object to data model transformation process. You can assign one or more attributes to be candidate keys. If you assign more than one candidate key to a class, those attributes will transform to a composite primary key in the data model.

ID-based Columns

If you do not designate a candidate key in your parent classes, Data Modeler will automatically generate a primary key for each parent class when you transform your object model to a data model. Data Modeler generates this key by adding an additional column to the table (called an *ID-based column*) and assigns it as a primary key. An ID-based column uses unique system-generated values.

Figure 5 ID-based Columns



ID-based Key vs. Candidate Key

An ID-based key is considered to have distinct advantages over a candidate key, when choosing a unique identifier for a table. One of these advantages is an ID-based key's ability to maintain a constant size, because it is a system-generated value.

Another advantage is that an ID-based key uses one column, whereas a candidate key may need to use multiple columns to uniquely identify the table. Using a single column results in a simpler, cleaner database design, because in relationships only one column not multiple columns migrates as a foreign key.

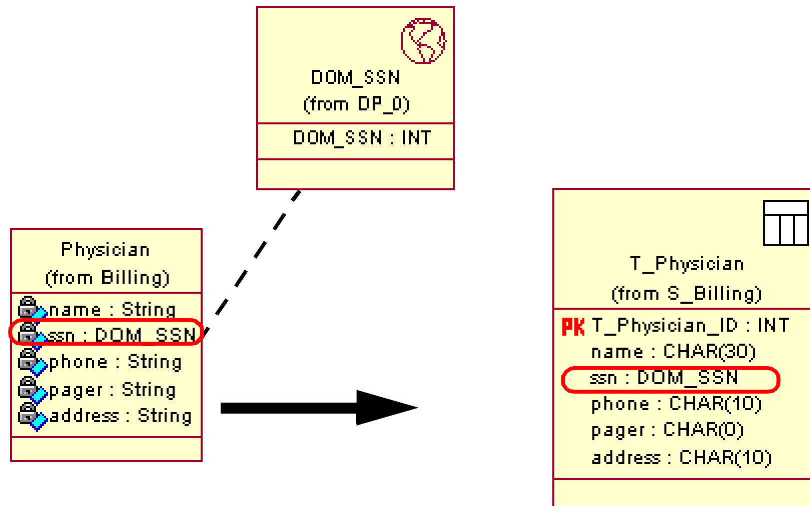
Using multiple columns also makes it more difficult to ensure unique entries in the table. For example, in the T_Physician table you may use name and address as candidate keys, but you may encounter duplications if one physician has the same name as another physician who lives at the same address, such as would occur in a mother-daughter situation. To avoid this you may use ssn as the candidate key, but then there is a greater likelihood of this information being entered incorrectly. A system-generated ID-based key reduces these problems.

Domain Columns

You can map attributes to *domains* if you already have a domain defined in your data model. Refer to *Chapter 4, Physical Data Modeling* for more information on creating domains in the data model. Domains can act as a user-defined data type corresponding to a specific DBMS language. It is important that you assign your domain to the same DBMS language that you will use for your data model, so your domain will be compatible with your database.

You map attributes to domains by setting the attribute type to the name of your domain. In the transformation process, the attribute transforms to a column using the domain name as the data type, thereby using the domain's defined data type and settings.

Figure 6 Domain Columns



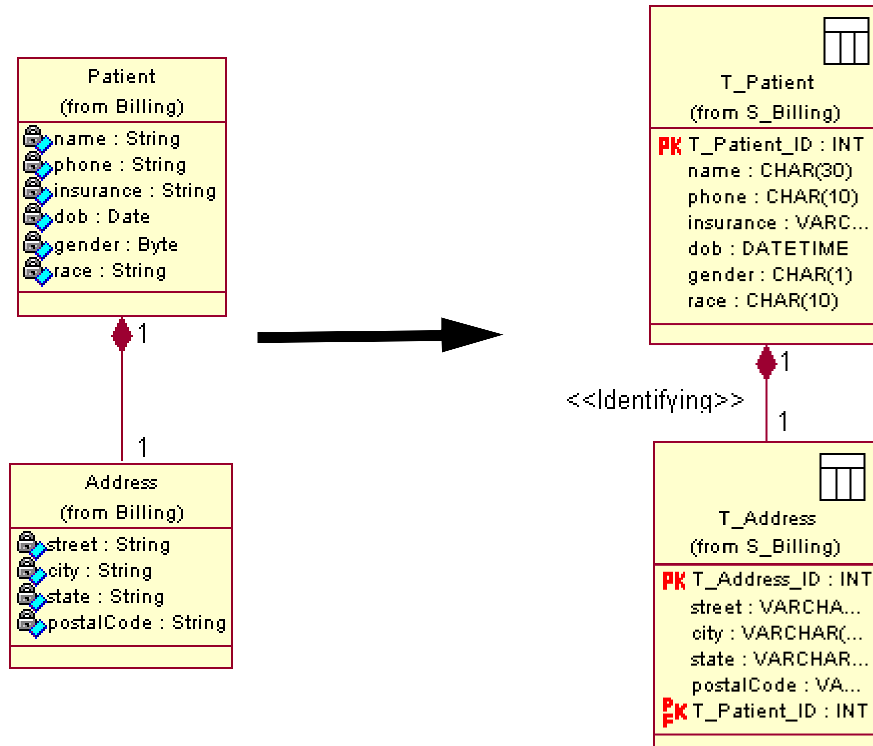
Data Type

Data Modeler automatically maps your attribute type to an appropriate column data type of your DBMS or the ANSI SQL 92 standard. If you want your attribute to transform to a particular data type in the data model, you need to designate the correct attribute type that maps to your desired data type. Refer to *Appendix B* for a listing of the data type mapping. Furthermore, if you want to specify a default value for your column, you can specify it in the attribute's initial value which maps to a column's default value.

Mapping Composite Aggregations to Identifying Relationships

Aggregations by value, known as ***composite aggregations***, map to identifying relationships in the data model. Composite aggregations consist of a whole and a part, indicating a "strong" relationship, wherein the part cannot exist without the whole. You use a composite aggregation when you have one instance of a parent class that owns a dependent class. The dependent class is defined as the part and must be accompanied by a whole or parent class. If the parent class is deleted its composite parts must be deleted also, therefore the parent class must use the multiplicity of 1.

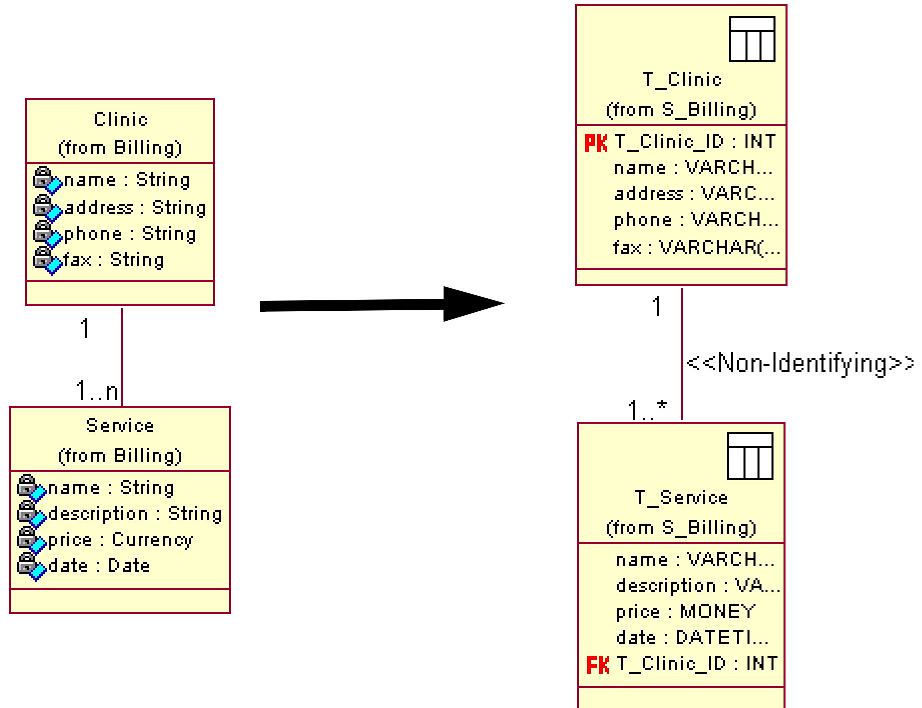
Figure 7 Composite Aggregations Map to Identifying Relationships



Mapping Aggregations and Associations to Non-Identifying Relationships

Aggregations by reference (known as *aggregations*) and associations map to non-identifying relationships, with the exception of many-to-many associations. Both of these join two classes without using a “strong” relationship. You use an aggregation when you have multiple instances of a parent class owning a dependent class. You use an association when you have classes that are independent of each other. An aggregate or association can be mandatory, where a parent class is required, by using a multiplicity of 1 or 1..n. An association can also be optional, where a parent class is not required, by using a multiplicity of 0..n.

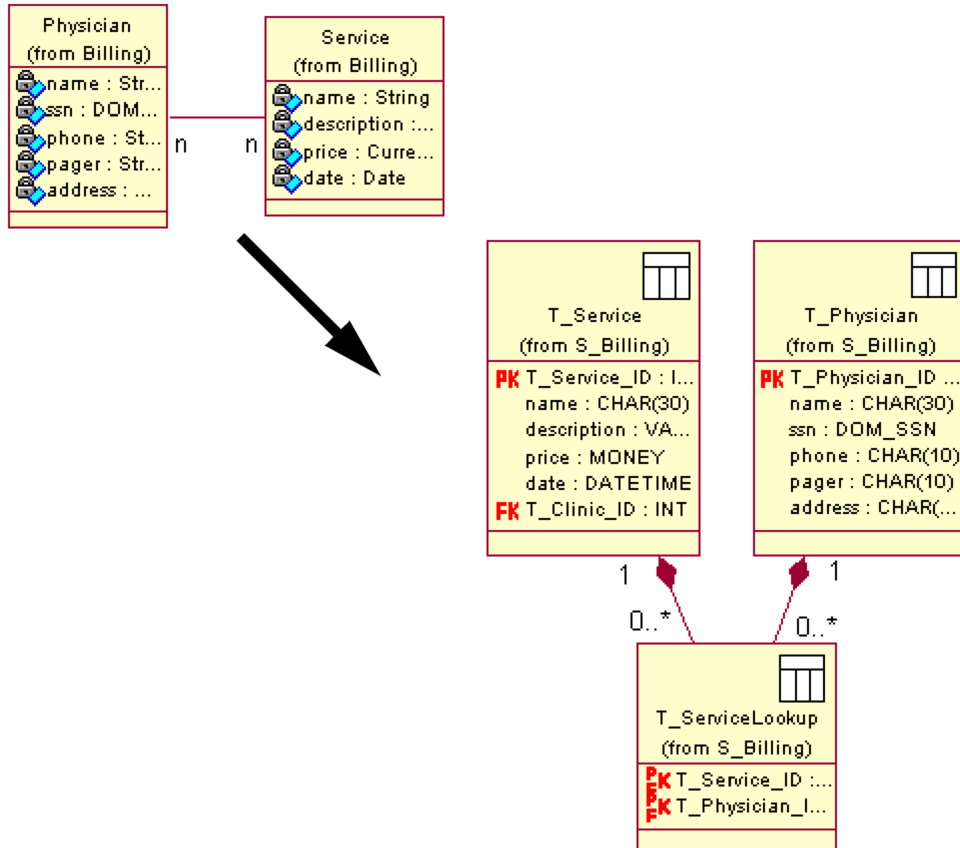
Figure 8 Associations Map to Non-Identifying Relationships



Many-to-Many Associations

Many-to-many associations map to *intersection table structures* where all the columns of the intersection table are primary/foreign keys. In the transformation process, Data Modeler reads the two classes in the many-to-many relationship and creates two separate tables, then it joins these two tables with two identifying relationships to a system-generated table called an intersection table. As part of generating the identifying relationships, the two tables' primary keys migrate to the intersection table as primary/foreign keys.

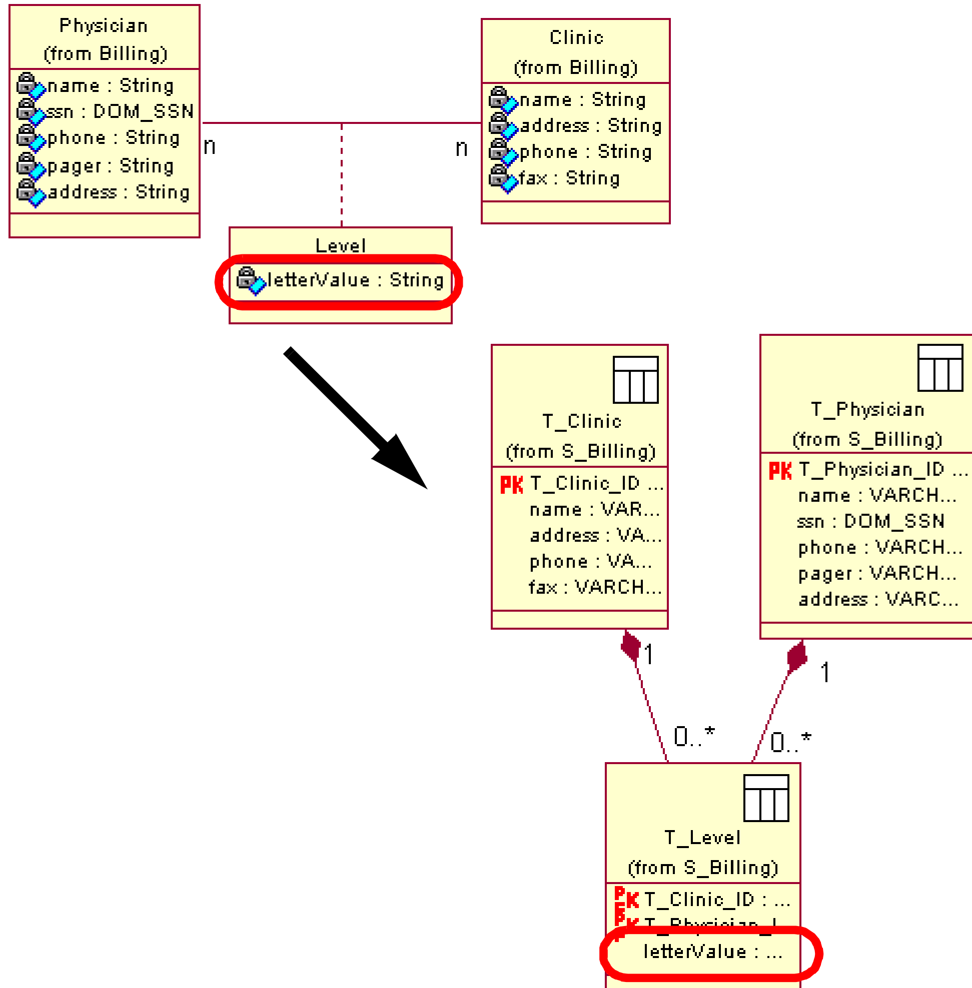
Figure 9 Many-to-Many Associations Map to Intersection Tables



Mapping Association Classes to Intersection Tables

An *association class* that links to a many-to-many association maps to an intersection table structure where the intersection table contains one or more additional columns that are not primary/foreign keys. An association class is an additional class attached to an association that can store properties and operations shared by the two individual classes of the association. The reason the classes share these properties and operations is because they cannot be stored in either of the classes. Data Modeler transforms association classes and their properties, but association class operations are ignored by Data Modeler. In the object to data model transformation, Data Modeler transforms the two classes to tables and joins them to an intersection table with two identifying relationships, migrating the primary keys of each individual table to the intersection table as primary/foreign keys. Then Data Modeler adds the attributes of the association class as columns in the intersection table.

Figure 10 Association Classes Map to Intersection Tables

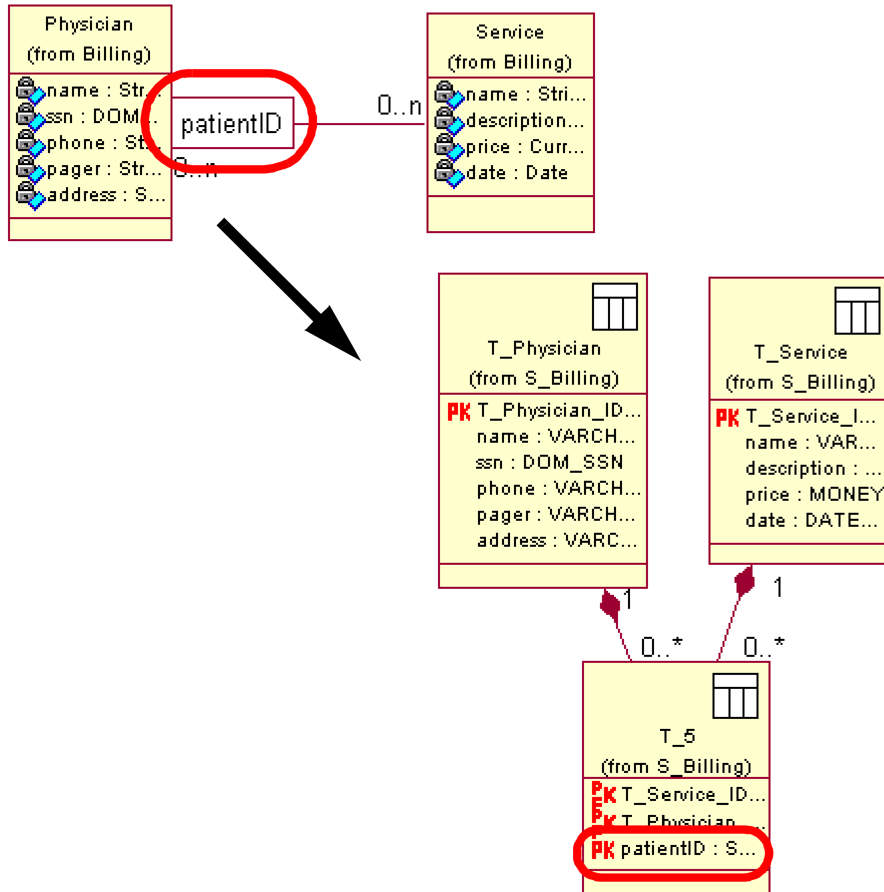


Mapping Qualified Associations to Intersection Tables

A *qualified association* maps to an intersection table that contains an additional primary key. Qualified associations are many-to-many associations that use qualifiers. A qualifier is an attribute applied to one side of an association that works as an identifier to return a specific set of objects at the opposite end of the association. Similar to the transformation process of association classes, the two classes in a qualified association transform to individual classes and are joined with identifying relationships to the intersection table. The qualifier of the qualified association is transformed to a

primary key in the intersection table. The intersection table then contains the primary/foreign keys of the two tables, and the additional primary key column generated by the transformation of the qualifier.

Figure 11 Qualified Associations Map to Intersection Tables

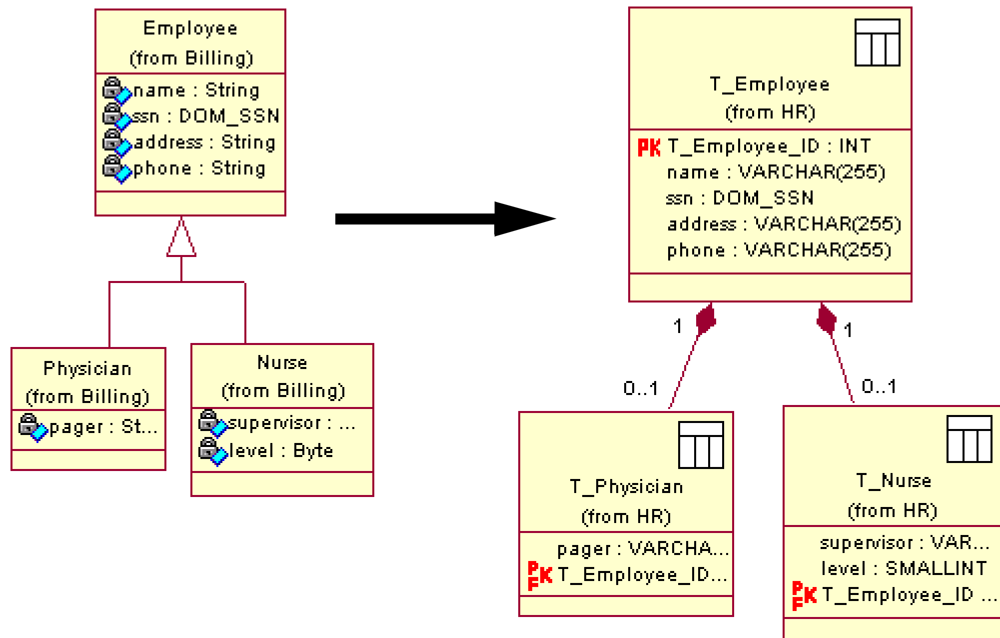


Mapping Inheritance

Inheritance structures or, in UML terminology, *generalization* structures can map to separate tables or one table; however, Data Modeler only transforms inheritance structures to separate tables. Inheritance structures associate a general parent class with more specific child classes. The child classes share the same attributes of the parent class and add additional attributes that affect only their specific class.

When Data Modeler transforms an inheritance structure to separate tables, one table is created for each participating class. The parent table is joined to each of the child tables with a zero-to-one or one identifying relationship. Therefore, each child table contains a primary/foreign key related to the parent table's primary key.

Figure 12 Inheritance Maps to Separate Tables



You can manually map an inheritance structure to one class after you transform your object model to a data model. You do this by deleting the two identifying relationships between the tables; this deletes the primary/foreign keys in your tables. Then, you move the individual columns from the two child tables to the intersection table. Finally, you set the child classes in your object model to transient.

Important: If you manually map your inheritance structure to one table, you will have to repeat this process each time you transform your object model to the data model or vice versa. If you do not manually remap your models each time you transform, you may encounter conflicts with your application and DBMS mapping.

Transforming the Object Model to the Data Model

After customizing your object model to map to a data model, you can transform your object model to generate a data model that reflects the mapping you specified. You transform the object model to a data model for various reasons, but regardless of the reason the process is the same.

Why Transform

You can transform an object model to generate a new DBMS database, or to synchronize the object model and the data model, sharing information with data designers.

Transformation is a necessary step in generating a new database that supports the application. When you transform the object model, a data model is generated. This data model can then be forward engineered to create a DBMS database.

Also, when you transform the object model to a data model you are synchronizing the models, because each of the generated elements in the data model map to one or more elements in the object model. This synchronization creates consistency in the system and allows for the sharing of information between the application designers and the database designers through the object model.

The database designer is dependent on the object model of the application, because it is through this structure that the data is accessed. Therefore the transformation of the object model to the data model can be the means of communicating application changes that may affect the DBMS database. The application designer can share information, especially changes, by transforming the changed object model to a data model. The database designer can review the changes and see the impact such changes would have on the DBMS database without updating the database. You can share the object model changes with the database designer by using the Data Modeler Transform Object Model to Data Model feature.

The Transformation Process

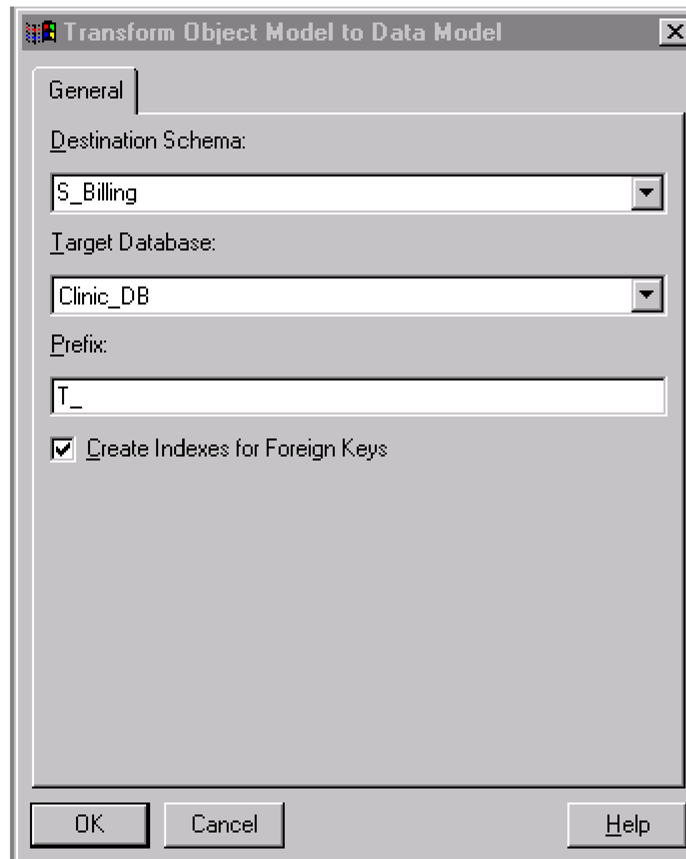
All the object model elements mentioned in the mapping section of this chapter transform to the data model items they are mapped to, except components and operations.

Even though components map to a database, components themselves do not transform to the data model. In fact, in place of a component, you must create a new database before transforming your object model to a data model. When you create a new database, you can specify a database name and assign the database target to a supported DBMS or the ANSI SQL 92 standard. Refer to *Chapter 4 Physical Data Modeling* for information on creating a new database.

Transform Object Model to Data Model Dialog Box

The process of transforming an object model begins from the **Transform Object Model to Data Model** dialog box. From this dialog box, you can select or specify a schema name for your data model, the database name you will be using, prefixes for your tables, and whether or not you want to create indexes for your foreign keys.

Figure 13 Transform Object Model to Data Model Dialog Box



Destination Schema

If you select an existing schema name, that existing schema will be overwritten by the new schema generated from your object model. If you specify a new schema name, Data Modeler will generate a new schema using that name without overwriting any existing schema you may have assigned to your database.

Target Database

When you specify a **Target**, which is the database name, you must select a name from the list. If you leave the Target blank, your specification dialog boxes will be unavailable to you after the transformation is complete.

Prefixes

You can distinguish your new tables from the classes in your object model by assigning a prefix to them. It is recommended you use a prefix. The prefix will appear before each of your tables' names and will help you avoid drawing relationships between a table and a class or vice versa, making your object model or data model invalid.

Indexes for Foreign Keys

As an additional convenience, you can specify to have an index built for every foreign key in your new data model. When you click **OK** on this dialog box, the actual transformation process begins.

Transforming Packages, Classes, and Associations

In the actual transformation process, Data Modeler generates a schema or executes a destructive rewrite using the schema name you designated in the **Transform Object Model to Data Model** dialog box. Then Data Modeler reads the object model package looking for persistent classes and relationships between persistent classes. These classes are transformed to tables in the data model.

At this point, Data Modeler looks for any attributes assigned as candidate keys in the persistent classes. If there are attributes assigned as candidate keys, each of these keys are transformed to primary keys and are added to the table's primary key constraint with unique indexes being created for each constraint. If none of the attributes are assigned as a candidate key, Data Modeler adds an ID-based column to the parent tables and assigns that column as the primary key for the table. This process ensures that all the parent tables have a primary key, so all relationships will be valid. Thereafter, all other attributes are added to the table as columns.

Once the tables are established, Data Modeler transforms the object model associations to data model relationships. Each of the relationships receives the appropriate cardinality and roles as designated in the object model. Also, each of the relationships migrates a foreign key to the child table based on the primary key in the parent table. If you specified to have an index built for each foreign key, these indexes are generated after the foreign key migrates from the parent to the child table. This also includes any special structures like those that transform to intersection tables.

The entire processing time of the transformation depends on the number of classes to be transformed in your object model.

After the Transformation Process

When the transformation process is complete you can make any changes necessary in the data model to map the object and data model more precisely. You can also see the object model to data model mapping using the *Mapped From* property in the data model's Table or Column Specifications.

Mapped From

When you transform an object model or a data model the Mapped From text box is automatically populated with the name of the referencing object model element. This is a protected field so you cannot control it manually. When you are working with a data model you can know which class or attribute a table or column maps to, even if the class has been renamed, by reviewing the Mapped From text box for your particular table or column. This property is also helpful because mapping object model elements to data model elements is not always a one-to-one mapping. You can have instances where your data model is denormalized and a class and a table loses their one-to-one mapping, because of the movement of columns between tables. In an instance like this, the Mapped From text box can help you track the source of an entity regardless of the changes made to it.

Physical Data Modeling 4

Contents

This chapter is organized as follows:

- *Introduction* on page 29
- *Data Models* on page 29
- *Building a New Data Model* on page 30
- *Reverse Engineering to Create a Data Model* on page 44
- *After Building the Data Model* on page 45

Introduction

Physical data modeling is the next step in modeling a database. The physical data model uses the logical data model's captured requirements, and applies them to specific database management system (DBMS) languages. Physical data models also capture the lower-level detail of a DBMS database. Rose Data Modeler calls this physical data model, the "data model."

Data Models

When working with Data Modeler, the physical data model is known as a data model and is graphically represented in the data model diagram. The data model diagram is customized from other UML diagrams allowing the database designer to work with already familiar concepts and terminology.

Using Rose you can create a data model in a variety of ways. The previous chapter explained how to create a data model by transforming an object model, but you can also create a data model by building a new model, or by reverse engineering a database or DDL file.

Building a New Data Model

You can build a new data model by manually creating the elements of a data model. This section describes the process of building a data model using Data Modeler.

Create a Database

A *database* is the implementation component for a data model. Each database is assigned to a *target*. The target refers to the actual ANSI SQL 92 standard or supported DBMS and DBMS version you want to use. You can specify your target using the Database Specification. The default target is ANSI SQL 92. When you designate a target only the elements supported by your designated DBMS version or ANSI SQL 92 standard are supported in your data model.

Also when you create a database, a Schemas folder is automatically created for you in your Rose browser, so you can store your schemas. Each database can contain one or more schemas.

Create a Schema

A *schema* is the primary container for a data model and contains all the tables of the data model. Data Modeler requires a schema for the data model to exist. Therefore, all elements in a data model, with the exception of domains are required to be assigned to a schema.

Before you create tables for your schema, you should assign your schema to a database. When you assign your schema to a database, only the elements supported by your database's designated DBMS version or ANSI SQL 92 standard are supported. If you do not assign your schema to a database, your schema will use the ANSI SQL 92 standard as a default.

Create a Data Model Diagram

It is necessary for you to create a data model diagram if you want to create relationships, because you must draw your relationships on the data model diagram. When you create a data model diagram and enable it, Data Modeler's customized tool set and the corresponding menu commands are made available to you. Refer to *Chapter 2, Data Modeling and UML* for more information on data model diagrams.

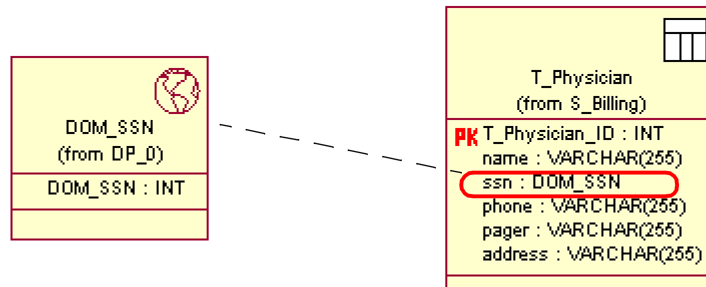
Create Domains

Domains act as a template for columns you use frequently and can be applied to columns and attributes alike as a customized data type. For example, when modeling tables for employees, the social security number will always be needed. Instead of

recreating the ssn column settings repeatedly, you can create a domain that contains the social security number settings and assign that domain as the data type for the column. Refer to Figure 14 on page 31.

Creating a domain is optional and is dependent upon the domain support for your DBMS. When you create a domain, you must first create a domain package as a container for your domain. You can assign your domain package to a specific DBMS. Then you can create your domain and assign your domain to the domain package. Assigning your domain to a domain package means your domain will only support the data types supported by the domain package's DBMS. You can also tag your domain as server-generated for forward engineering purposes.

Figure 14 A Domain

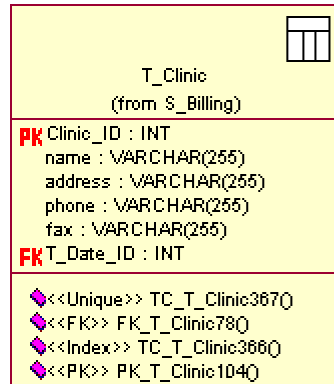


Create Tables

A table identifies an entity. Each table can contain columns, constraints, triggers, and indexes. Tables are joined together through relationships. A table can belong to only one schema, however a schema may contain zero or more tables. You can also indicate the tablespace name for your table.

When you create a table, your table name must be unique within the schema to which it belongs, and must meet your specified DBMS's naming requirements. It is recommended you use a prefix to distinguish the tables in your data model from the classes in your object model.

Figure 15 A Table



Create Columns

A column defines the characteristics of a table. The tables are linked by their columns. Column names must be unique within the table and column values are controlled by constraints. You can specify a column type, data type, precision, length, scale (if required), and whether the column is part of a key constraint and is nullable.

Column Types

Each column is assigned a column type. Data Modeler supports two column types—data columns and computed columns.

Data Column

A data column stores any data information except derived values. You can assign your data column to a data type supported by your target DBMS, or to an existing domain. If you specify a data type, your data type may require you to enter a length or precision and an accompanying scale. You can assign a default value to your data column. You can also assign constraints to your data column, specifying if it is a primary key, unique key, and whether NULL values are accepted. Data columns also support SQL Server's Identity property and DB2's ForBitData.

Computed Column

A computed column uses a SQL expression to derive and store its values. Because they are derived, and do not use unique values, computed columns cannot be assigned as a primary key or unique key. As part of optimization efforts to increase query speed, you can create an index on a computed column.

Create Constraints

There are two types of constraints—*key constraints* and *check constraints*. Key constraints restrict data in a column or group of columns to uniquely identify a row and enforce referential integrity within a relationship. Check constraints restrict the adding to or modifying of a table's data to enforce business rules.

Key Constraints

There are three types of key constraints—primary key, unique, and foreign key. However, also included as a type of key constraint is indexes. Indexes are included as a type of key constraint because they relate directly to the other key constraints, and use a key column list. Each table can have only one primary key constraint and can contain zero or more unique and foreign key constraints. To enforce the key constraints, the database server creates a unique index.

Primary Key Constraint

If you assign a primary key for one of your columns, a primary key constraint is automatically created for you consisting of that column and any other columns you assign as a primary key. Primary key constraints do not allow any two rows of a table to have the same non-NULL values in any primary key column, and do not allow any primary keys to have NULL values. Primary keys control the characteristics of all corresponding foreign keys and can identify the parent table in a relationship.

Primary key constraints can consist of one primary key or a composite primary key. The composite primary key can consist of primary keys and/or primary/foreign keys. Primary/foreign keys are embedded keys that enforce the referential integrity of a relationship, and therefore cannot have NULL values. Primary/foreign keys are created through identifying relationships, so you cannot manually control a primary/foreign key. Refer to *Embedded Primary/Foreign Key* on page 35 for more information on primary/foreign keys.

Unique Constraint

Unique constraints also known as alternate constraints consist of one or more unique columns. Unique constraints do not allow any two rows of a table to have the same non-NULL values in any unique constraint columns. NULL values are not allowed for this constraint, with the exception of the SQL Server DBMS. SQL Server allows NULL values for unique constraints.

Foreign Key Constraint

Foreign key constraints consist of one or more foreign keys. Foreign keys are read-only with the exception of the foreign key name and default value. Each foreign key is generated by creating a relationship between two tables, thereby migrating the primary or unique key from the parent table. Any changes made to the parent table's primary or unique keys cascade to the foreign keys in the child table. NULL and unique constraints on foreign keys are controlled by the relationship's cardinality. Refer to *Cardinality on page 37* for more information.

Index

Indexes consist of a key list of columns that provide quick access to any given value by searching only that key list of columns. Each index must have a unique name. You can cluster your index, however Data Modeler only allows one clustered index per table. Clustering an index increases the efficiency of an index by physically storing the index with the data. You should always use a clustered index when you are creating an index for a child table that participates in an identifying relationship.

You can also specify a fill factor/percent free for your index. The fill factor/percent free specifies the percentage of rows each index page can contain. Specifying a low fill factor allows for flexibility in your index. Specifying a high fill factor allows for little change to the records in the index.

Creating Key Constraints

You can create key constraints by specifying the name of the key constraint, and what type of constraint it is—either primary key, unique, or an index. You can also select columns to include in your key constraint. If you are creating an index you can specify if you want the index to be unique; the key constraints that automatically generate their indexes are set as unique indexes.

Check Constraints

A check constraint restricts actions to a table's data, by using SQL predicate expressions. If the SQL expression returns false when it is executed, the table's data is not altered. You can create a check constraint for tables or domains using the **Check Constraints** tab on the Table or Domain Specification. You can specify a check constraint name and a SQL expression, and if you are using Oracle as your target DBMS you can also specify *deferrable* or *non-deferrable*.

Deferrable vs. Non-deferrable (Oracle only)

Check constraints for the Oracle DBMS can be identified as non-deferrable or deferrable. Non-deferrable check constraints verify the validity of an action at the end of the SQL statement. Deferrable check constraints verify the validity of an action either at the end of the statement using Initially Immediate, or at the end of the transaction before it is committed using Initially Deferred.

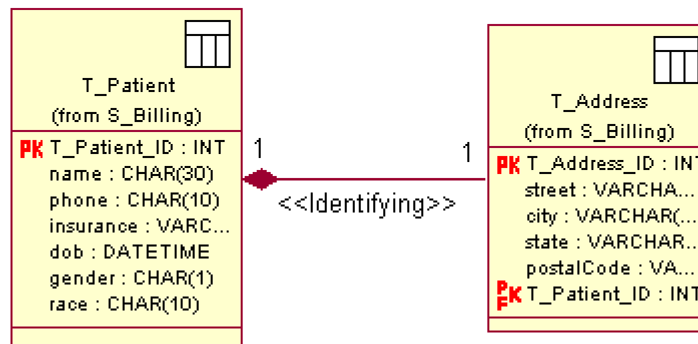
Create Relationships

Relationships relate tables to one another in a data model using two types of relationships—*identifying* and *non-identifying*. You can also define the cardinality and roles for a relationship, and you can use them in different relationship structures.

Identifying Relationships

An identifying relationship specifies that a child table cannot exist without the parent table. When you use an identifying relationship, the primary key of the parent table migrates to the child table as a foreign key. The foreign key is embedded in the child table's existing primary key constraint as a primary/foreign key. If the child table does not have an existing primary key constraint, the migrating foreign key is assigned as a primary/foreign key creating both a foreign key and a primary key constraint.

Figure 16 An Identifying Relationship



Embedded Primary/Foreign Key

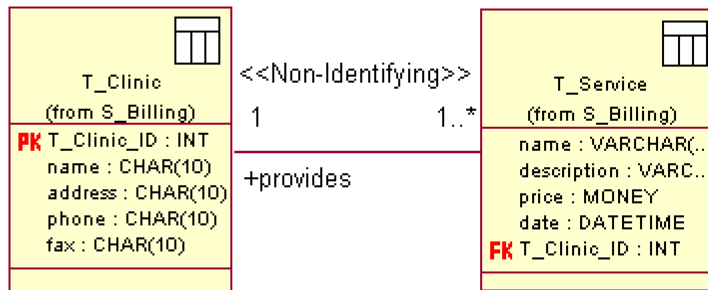
When the foreign key is embedded, it appears as a primary/foreign key in the table. Embedding the foreign key in the primary key enforces the referential integrity of the relationship. This embedding prevents orphan records in the child table by requiring the deletion of the record in the child table first, before deleting the record in the parent table. It also prevents you from reassigning the primary key to another column

in the parent table, because such a reassignment would create orphan records in the child table. It is the embedded primary/foreign key that distinguishes the relationship as an identifying relationship as opposed to it being a non-identifying relationship.

Non-Identifying Relationships

Non-identifying relationships are relationships in which there is no interdependency between child and parent tables, hence the foreign key is not embedded in the child table's primary key constraint. There are two kinds of non-identifying relationships—*optional* and *mandatory.* In an optional non-identifying relationship the parent table is not required, therefore the parent table of the relationship uses the cardinality of 0..1. In a mandatory non-identifying relationship, a parent table is required and uses the cardinality of 1 or 1..n.

Figure 17 A Non-Identifying Relationship



Mandatory Non-Identifying Relationships vs. Identifying Relationships

Because both mandatory non-identifying relationships and identifying relationships require a parent table, it may be difficult to know when to use them. When making this decision, consider first if the tables have a dependency of part-to-whole, where the part would have no meaning without the whole. If they do, then you should use an identifying relationship. For instance in the Clinic model, the Address table has addresses, but no names of patients who live at those addresses; therefore the addresses themselves have no meaning, they must be related to a patient's name.

Another important factor to consider is whether the foreign key can be nullable. In a mandatory non-identifying relationship the foreign key can be null, because it is not part of the primary key constraint. This allows for more flexibility in the relationship.

Cardinality

Cardinality is the minimum and maximum number of possible instantiations of a relationship between two tables. Cardinality is used to enforce referential integrity. If a table has a cardinality of 1, then that signifies the table must exist in the relationship. This cardinality is especially important for parent tables to prevent orphan records in the child tables.

Cardinality can also determine if a foreign key is unique and can be nullable. It is the foreign key's ability to be NULL that can determine if you should use an identifying or non-identifying relationship. If the parent table has a cardinality of one or more the foreign key cannot be NULL. Below is a table specifying the necessary cardinalities to make a foreign key nullable and/or unique.

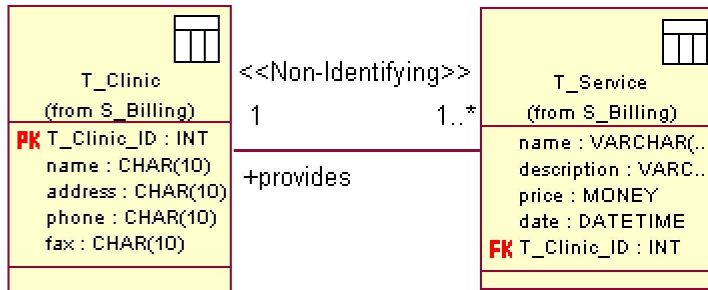
Table 1 Cardinalities for Foreign Key Constraints

Required Constraints	Parent Table Cardinality	Child Table Cardinality
Foreign key is nullable and unique	0..1	0..1 or 1
Foreign key is nullable and not unique	0..1	0..* or 1..*
Foreign key is not nullable and not unique	1	1..* or 0..*
Foreign key is not nullable, but is unique	1	0..1 or 1

Roles

Roles in a relationship explain how the table acts in the relationship, giving meaning to cardinality. You can apply roles to either one table or both tables in a relationship. Roles combined with cardinality create a grammatical statement of what is occurring in the relationship. So as each parent table in a relationship acts as a noun, the role acts as a verb and the child table acts as a direct object, and vice versa. For example in Figure 18 on page 38 the role of the relationship between a clinic and the services it provides is stated: One clinic provides one or more services.

Figure 18 A Role



This helps when trying to transform business requirements to modeled elements in the data model, and communicates to the business analyst that the business requirements have been met.

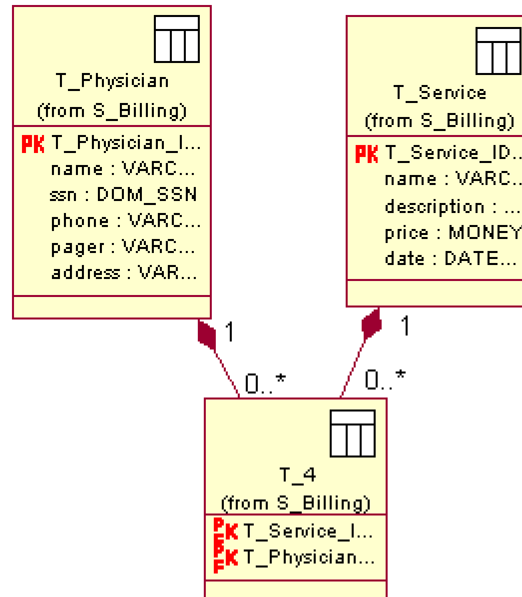
Relationship Structures

Using identifying and non-identifying relationships you can create different relationship structures. One of these structures is an intersection table. The other structure is a self-referencing structure.

Intersection Tables

Although, an *intersection table* is by definition a table, the specific relationship structure it participates in is what really defines it. An intersection table is a relationship structure in which one child table is related to two parent tables with two 1:n identifying relationships. When such a structure is created the child table contains primary/foreign keys corresponding to the parent tables' primary keys, therefore any updates made to the child table will affect both parent tables. This kind of relationship structure can be used for supertype/subtype structures.

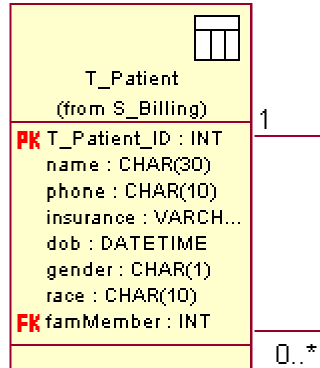
Figure 19 An Intersection Table



Self-referencing

Another relationship structure is a *self-referencing structure*. A self-referencing structure uses only one table, relating the table to itself with a non-identifying relationship. You use a self-referencing structure when you have an instance of a table that must be related with another instance of the same table. This kind of relationship structure can be used for recursive relationships.

Figure 20 A Self-Referencing Relationship



Define Referential Integrity

Referential integrity ensures the integrity of your data when you update or delete the parent table of a relationship. It does this by applying specific actions to its corresponding child tables or by preventing the parent table itself from being updated or deleted.

Data Modeler offers two methods to support referential integrity actions: *declarative referential integrity* (DRI) and *system-generated referential integrity* (RI) triggers. With these two methods you can perform the following actions:

- Cascade – deletes/updates all associated children
- Restrict – prevents deletion/update of the parent
- Set NULL – sets all child foreign keys to NULL
- No Action – no action taken
- Set Default – sets all child foreign keys to a default value

Declarative Referential Integrity

DRI specifies the referential integrity action as part of the foreign key clause when the data model is forward engineered. Although it is considered the most efficient method and easiest method, DRI is DBMS dependent and not all DRI actions are supported for each DBMS.

System-generated Referential Integrity Triggers

System-generated RI triggers specify the referential integrity actions by generating system triggers. System-generated RI triggers are better supported by each DBMS. As an additional integrity measure you can specify Child Restrict to prevent the insertion of orphan records.

Create Custom Triggers

Custom triggers execute a set of SQL statements when you update, insert, or delete data in your database. You can use a trigger to enforce business rules for multiple tables, because triggers can prevent specific data modifications. It is important to use triggers as a method of enforcing business rules, because it ensures that the application designer and the database designer complete the same business processes using the same logic.

Trigger Events

When creating a trigger you must decide on an event or combination of events which will fire the trigger. A *trigger event* is a specific action such as update, insert, and delete that modifies the data.

Additional Trigger Settings for DB2 and Oracle

Along with trigger events Data Modeler supports additional trigger settings available only to DB2 and Oracle DBMSs. These settings are trigger type, granularity, referencing, and using the WhenClause.

Trigger Type

Data Modeler defines a *trigger type* as the determination of when the trigger statement is verified before the trigger event, after the trigger event, or if the trigger is fired instead of the event occurring.

If the trigger statement is verified before the trigger event, the trigger can verify if the condition of the modification is appropriate for the database, before the modification occurs. For example, if you are using a before trigger and attempt to insert a new patient record that does not meet the specified requirements in the SQL statement, the insert action is rejected, and the database is not modified.

If the trigger statement is verified after the trigger event, the trigger can verify if the condition of the modification is appropriate for the database after the modification occurs. Using the after trigger type you can determine a level of granularity and reference alias names, so you can see the results the modification would make to your data before committing it. This is important for the application designer, who can create a call that points to the old database if an error occurs, or to the new database if no error occurs.

If the trigger is fired instead of the trigger event occurring, the proposed modification does not occur. In place of the proposed modification, the trigger fires, creating its own modification.

Trigger Granularity

Trigger *granularity* determines how often the trigger is executed—after each row or after each statement. This distinction is important if you have a situation in which the modification condition would be considered invalid, unless first some other column or row was modified, for example updating a child table would require the parent table to be updated first, so you would want to execute only after each statement instead of each row. Whereas granularity of each row is a more detailed level and will verify the SQL conditions more often. It is the granularity that determines your level for trigger referencing whether by table or by row.

Trigger Referencing

Trigger referencing allows you to assign an alias name to your table or row before it is modified, or to your table or row after its modification. If you are using Oracle as your target DBMS, you can assign an alias name for the old row and the modified row. If you are using DB2 as your target DBMS, you can assign an alias name for the old table or row and the modified table or row. Such referencing gives you two views of the same table or row, which can be used by multiple applications depending on your business processes and development decisions.

Trigger WhenClause

A *WhenClause* is an additional SQL statement indicating a search condition that can further filter data modified in the database by the trigger. You can write this SQL search condition statement to return true, meaning only if the statement returns true will the modification occur. If the statement returns false, the modification will not occur.

Create Stored Procedures

In Data Modeler terms, a *stored procedure* can be a stored procedure or a stored function. Stored procedures reduce the occurrences of the same SQL statement in your model, and allow you to call these statements from the application with one name as opposed to coding the statements at the application level.

When you create a stored procedure, a stored procedure container is automatically created for you, indicating to which schema you assigned your stored procedure.

Language

When determining which language to use to write your stored procedure code, you should consider if you want your stored procedure to be internal or external. Internal stored procedures are supported by the database, and their code can only be written

in SQL. External stored procedures are supported by the application, and you can choose from Java and C code languages. When you use external stored procedures you can assign them an external name consisting of the path or library name.

Parameters

Parameters are the variables of the procedure or function. Each parameter can be assigned a data type, direction, and default value. The direction of your parameter depends on if your parameter is an input value, an output value, or both.

Stored Procedures

You use a stored procedure when you have a routine that will not return a value. For example you can create a stored procedure that adds new patient records when a new patient ID is created.

Parameter Styles for DB2 Stored Procedures

You can select the parameter style DB2Dari as a convention for sending parameters to and receiving values from a procedure that conforms to the C language.

Stored Functions

You use a *stored function* when your routine returns a NULL or Not NULL value. You can specify a specific data type for the return value including that data type's length or precision, and scale.

You can also determine whether or not you want the function called when the function parameters return a NULL value. If you do not want the function called when a NULL value is returned, use Return Null. If you want the function to be called regardless of a NULL value, use Call Procedure.

Not Deterministic

When a function is Not Deterministic the function depends on a state of values that affect the results, so the results are not always the same. Not Deterministic functions are used when you have a result value and a possible other value in addition to the result value.

Parameter Styles for DB2 Stored Functions

A parameter style is the convention for sending parameters to and receiving values from a function that conforms to a specific stored function language. Data Modeler allows you to select the DB2 parameter style that relates to your stored function

language. You can select DB2GNRL for functions that conform to C language calling and linking, or you can select DB2SQL for functions that are defined as methods in a Java class.

Reverse Engineering to Create a Data Model

You can create a physical data model by reverse engineering an existing DBMS database or DDL file. Before reverse engineering, you must ensure your existing database meets all your specific DBMS's requirements and is a stable structure. Data Modeler will not repair incorrect structures. This is especially important in table name length and column name length.

Reverse Engineering Wizard

You reverse engineer your DBMS database or DDL file by using the Reverse Engineering Wizard. The wizard guides you through and initiates the reverse engineering process, offering you the option to include indexes, triggers, and stored procedures in your new schema. It also guides you through your database connectivity. Refer to *Appendix D* for information on connecting to your database.

If the wizard encounters any error during the reverse engineering process, Data Modeler logs the error in the Rose Log and continues the process. When the wizard informs you that the process is complete you need to review the Rose Log for any errors that may have occurred.

Reverse Engineering DB2 Databases or DDL

When you reverse engineer a DB2 database or DDL file, Data Modeler reads the system catalog, and creates a new Data Modeler schema using the database name as the schema name. Then Data Modeler recreates your DBMS database elements in your new schema. All of your database comments are recreated in the data model as documentation. Each table is recreated in the data model schema including the table's columns, constraints, and appropriate data type settings. All distinct data types are recreated as domains in the data model. The domain name is the name of the distinct data type. If you are using DB2 MVS version 6.x, routines are recreated in the data model as stored procedures, with all arguments becoming parameters.

Reverse Engineering Oracle Databases or DDL

When you reverse engineer an Oracle database or DDL file, Data Modeler reads the user catalog, and creates a new Data Modeler schema database name as the schema name. Then Data Modeler recreates your DBMS database elements in your new schema. All of your database comments are recreated in the data model as

documentation. Each table is recreated in the data model schema including the table's columns, constraints, and appropriate data type settings. All user defined data types are recreated as domains in the data model. The domain name is name of the user-defined data type. Database procedures are recreated as stored procedures.

Reverse Engineering SQL Server Databases or DDL

When you reverse engineer a SQL Server database or DDL file, Data Modeler reads the system catalog, and creates a new Data Modeler schema using the database name as the schema name. Then Data Modeler recreates your DBMS database elements in your new schema. All of your database comments are recreated in the data model as documentation. Each table is recreated in the data model schema including the table's columns, constraints, and appropriate data type settings. All user-defined data types are recreated as domains in the data model. The domain name is the name of the user-defined data type. Your rules are recreated as check constraints in the data model.

Reverse Engineering Sybase Databases or DDL

When you reverse engineer a Sybase database or DDL file, Data Modeler reads the system catalog, and creates a new Data Modeler schema using the database name as the schema name. Then Data Modeler recreates your DBMS database elements in your new schema. All of your database comments are recreated in the data model as documentation. Each table is recreated in the data model schema including the table's columns, constraints, and appropriate data type settings.

After Reverse Engineering

When the reverse engineering process is complete and you have reviewed the Rose Log for errors, you can view your data model. Although your data model was populated with your database or DDL schema you can only view that model graphically by creating a data model diagram. Once you create your data model diagram, you can either drag your data model elements from the logical view on to the diagram or you can add your tables using the **Query** menu on the menu bar.

After Building the Data Model

When you completed the creation of your data model you can map and transform this model to an object model to build an application from the data model's design or you can implement this model by forward engineering it to generate the DDL code and/or DBMS database.

Mapping the Physical Data Model 5

Contents

This chapter is organized as follows:

- *Introduction* on page 47
- *Mapping the Data Model to an Object Model* on page 47
- *Transforming a Data Model to an Object Model* on page 56

Introduction

If you created your physical data model manually or reverse engineered an existing database to create a physical data model, you may want to map and transform your physical data model to the object model. Then your object model can act either as a robust logical data model that is synchronized with your physical data model, or it can act as the structure for your application model, so you can build your application using the same logic for modeling business processes as you use in your database.

Mapping the Data Model to an Object Model

Chapter 3 discussed Rose's ability to map the object model to the data model, but you can also map the data model to the object model. This mapping allows you to transform your data model to an object model, using the names of the data model elements for the names of the object model elements. You can map all data model elements to an object model, with the exception of stored procedures, triggers, key constraints, and check constraints that do not use enumerated clauses.

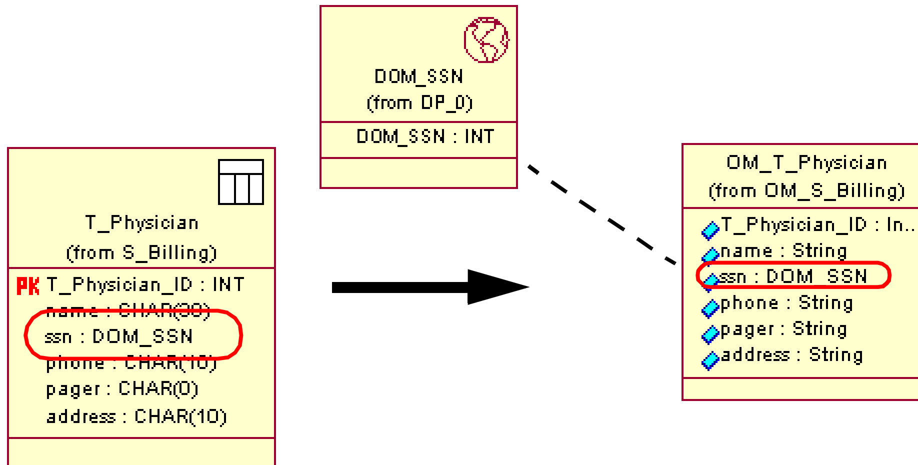
Mapping Schemas to Packages

Each data model schema maps to a logical package. Data Modeler allows you to transform one schema at a time, and generates one package for each schema transformed, using the schema name for the package name.

Mapping Domains to Attribute Types

Domains themselves do not map to an element in the object model. However, in the transformation process, a column that uses a domain as a data type will transform to an attribute, with an attribute type that references the domain name. For example, the column ssn uses the domain DOM_SSN. When it transforms to the object model, it transforms to an attribute ssn that uses DOM_SSN as its attribute type. Refer to Figure 21 on page 48.

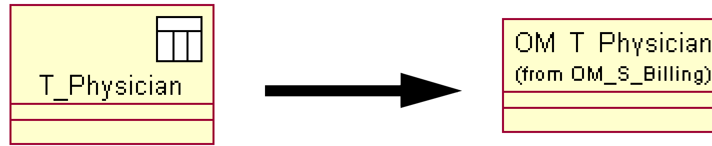
Figure 21 Domain Columns Map to Attributes



Mapping Tables to Classes

All tables map one-to-one to persistent classes in an object model, with the exception of intersection tables. Intersection tables map to special association structures. The classes can be used to build a model that can map directly to an application.

Figure 22 Tables Map to Classes

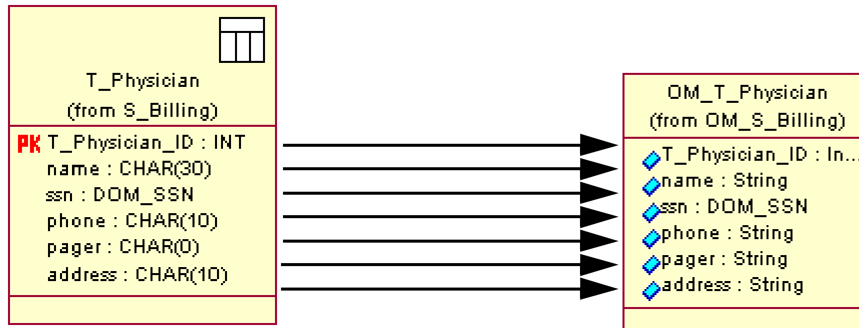


Mapping Columns to Attributes

Columns map to attributes in a one-to-one mapping with the exception of computed columns and foreign key columns. Column data types map to appropriate object model attribute types. Refer to *Appendix C* for more information on data type to attribute type mapping. Column default values map to an attribute's initial value. Primary key columns can map to attributes tagged as part of object identity.

However in the transformation process, other specialty columns are ignored. Specialty columns such as computed columns are ignored because they use derived values. Foreign key columns are ignored because they act as pointers in a relationship referring to an existing column in another table, therefore they are redundant and unnecessary.

Figure 23 Columns Map to Attributes



Mapping Enumerated Check Constraints to Classes

Check constraints that contain enumeration clauses such as "in" statements map to classes with stereotype <<ENUM>>. This provides a more robust logical model that closely resembles the physical model, giving more information on business rules. Business analysts and application designers can know the only acceptable values for specific columns in the logical data model, not just in the physical data model.

Figure 24 Enumerated Check Constraints Map to Classes with <<ENUM>>

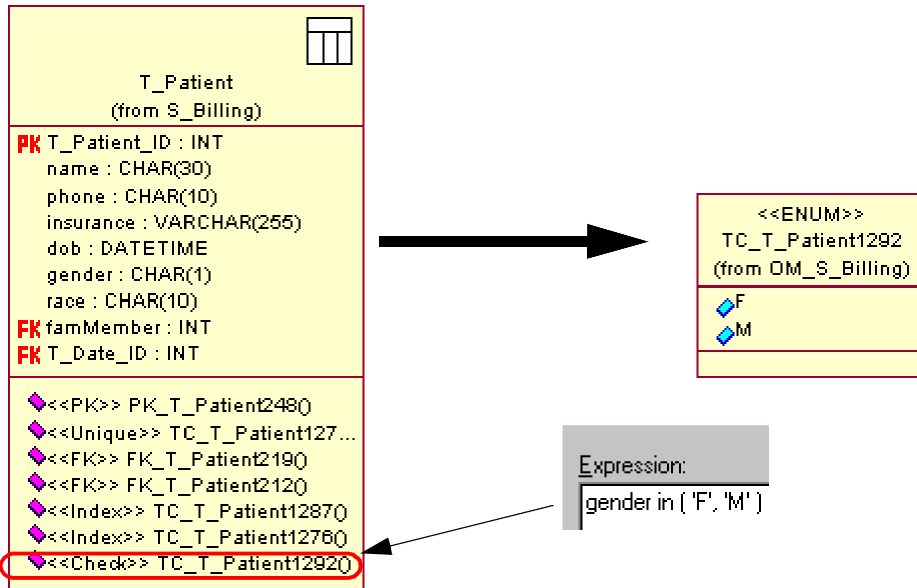
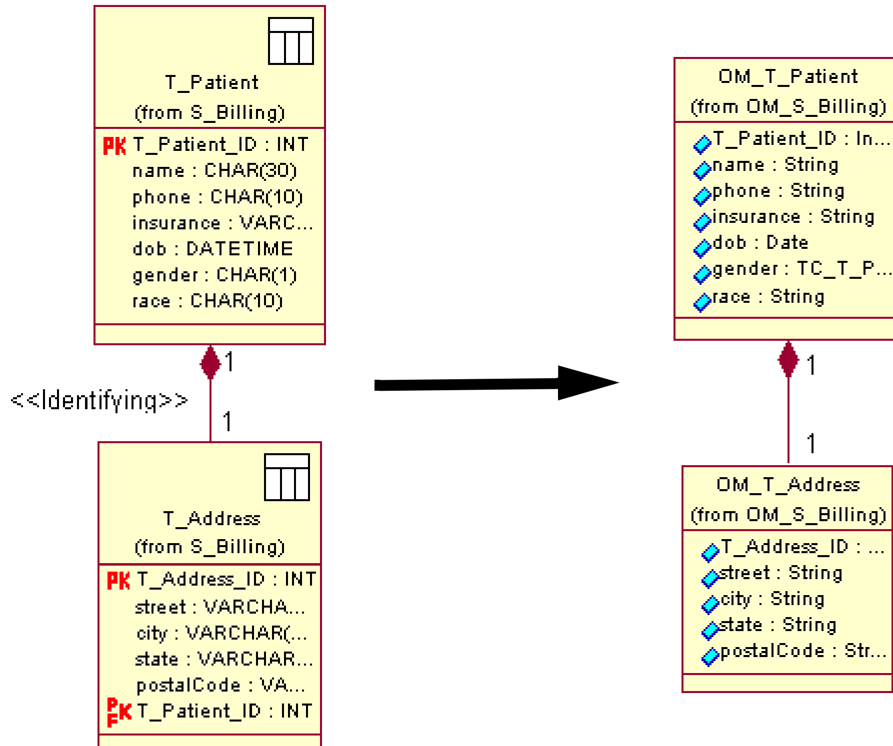


Figure 21 on page 48 demonstrates this with the enumerated check constraint for gender. The new class uses the name of the constraint, and creates attributes for each item in the enumeration clause.

Mapping Identifying Relationships to Composite Aggregations

Identifying relationships map to composite aggregations, because composite aggregations imply you cannot have a part without a corresponding whole. Identifying relationships also support this implication through the embedded foreign key (also called the primary/foreign key), because modifications cannot be made to a child table with a primary/foreign key without first making that modification to the parent table. Therefore, the child table (the part) could never have part of a record that is not already contained in the parent table (the whole).

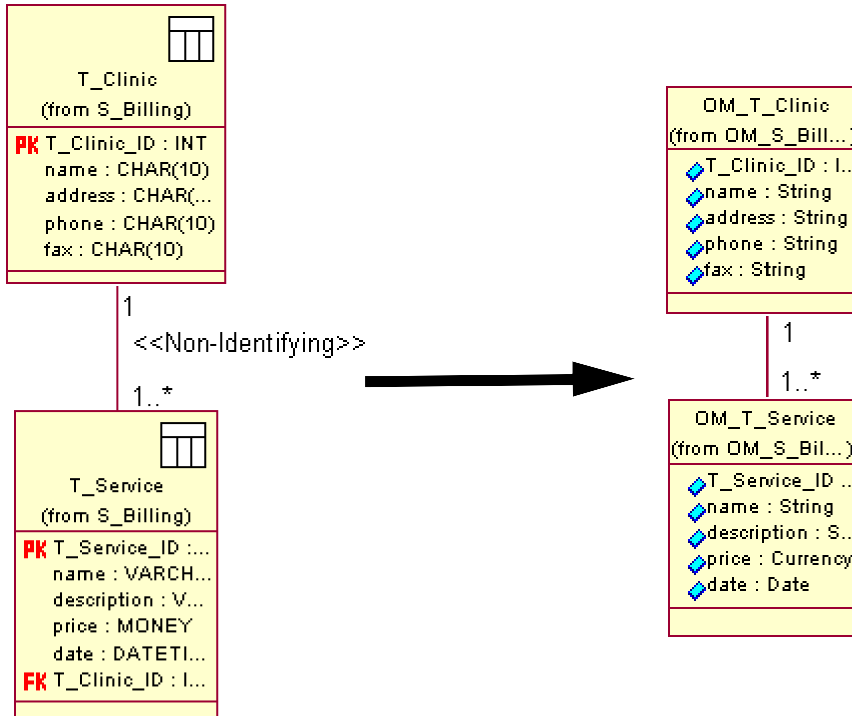
Figure 25 Identifying Relationships Map to Composite Aggregations



Mapping Non-Identifying Relationships to Associations

Non-identifying relationships map to associations, because associations join two classes that are independent of each other, or use a “weak” relationship. The weak relationship is indicated by the foreign key not being embedded in the primary key constraint.

Figure 26 Non-Identifying Relationships Map to Associations



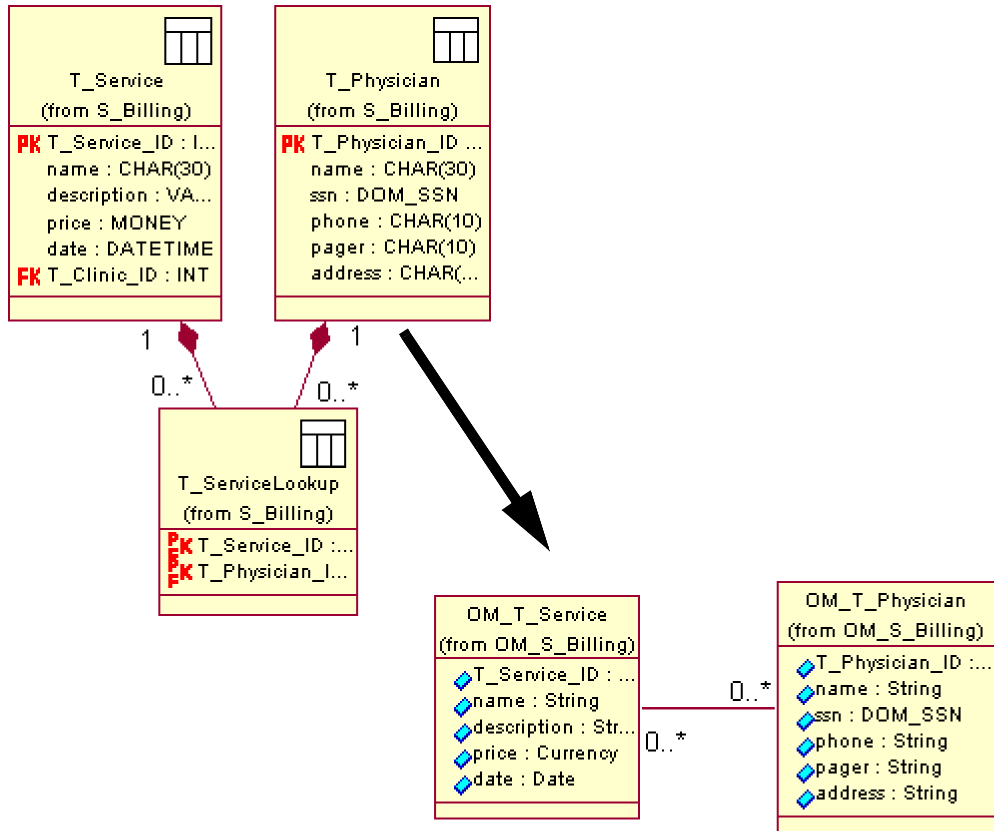
Mapping Intersection Tables

Intersection tables are difficult to map because depending on the structure of the intersection table itself, it can map up to three different object model elements—many-to-many associations, qualified associations, and association classes. An intersection can map to any one of these or all of them.

Many-to-Many Associations

An intersection table maps to a many-to-many association in an object model when all the columns in the intersection table itself are primary/foreign keys. This means all the data contained in the intersection table pertains to either one or the other table in the relationship. Therefore, the attributes that map to the columns containing this data must be in one or the other class in the association, but cannot be contained in both classes.

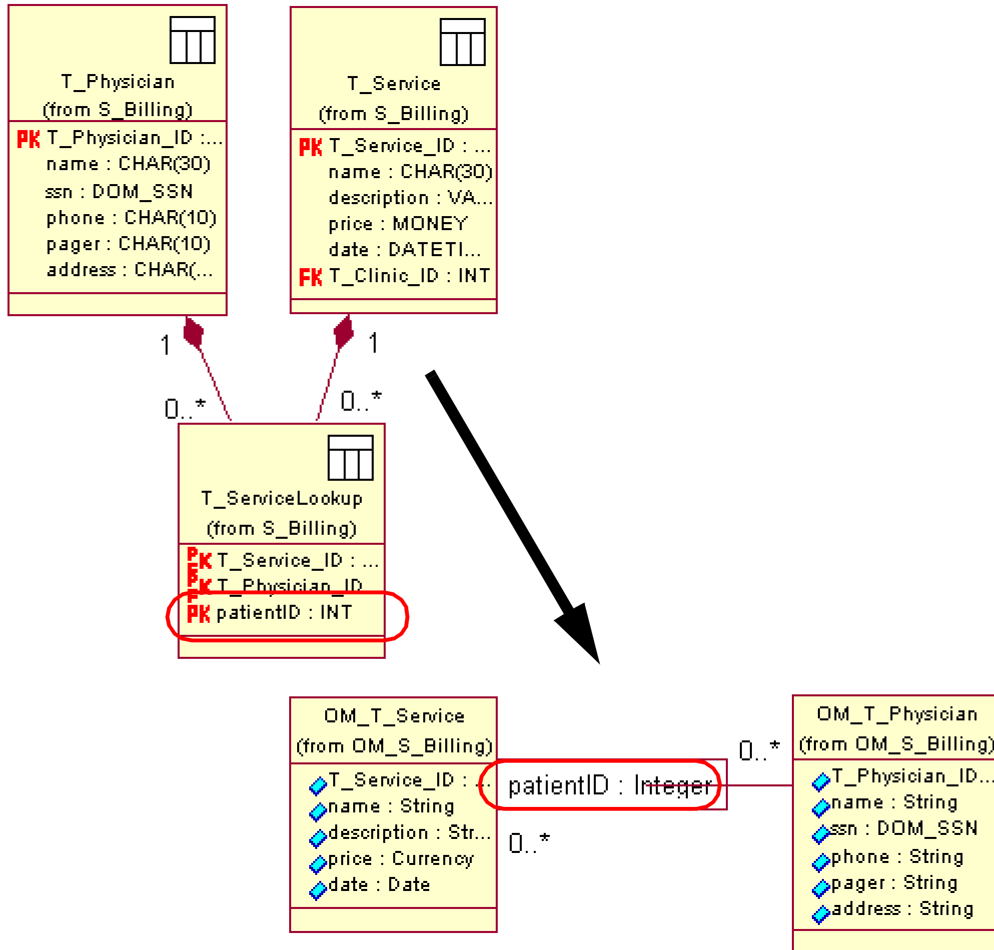
Figure 27 Intersection Tables Map to Many-to-Many Associations



Qualified Associations

An intersection table also maps to a qualified association of a many-to-many relationship when the intersection table contains an additional primary key. This additional primary key is specifically for the intersection table and is not received through an identifying relationship, meaning this key is not a primary/foreign key. Since the intersection table itself maps to the relationship between the two classes, the association "owns" the additional primary key column. The primary key column acts as a filter for the association, because neither class can own nor share this key. It filters out the possible instances that do not contain this primary key information.

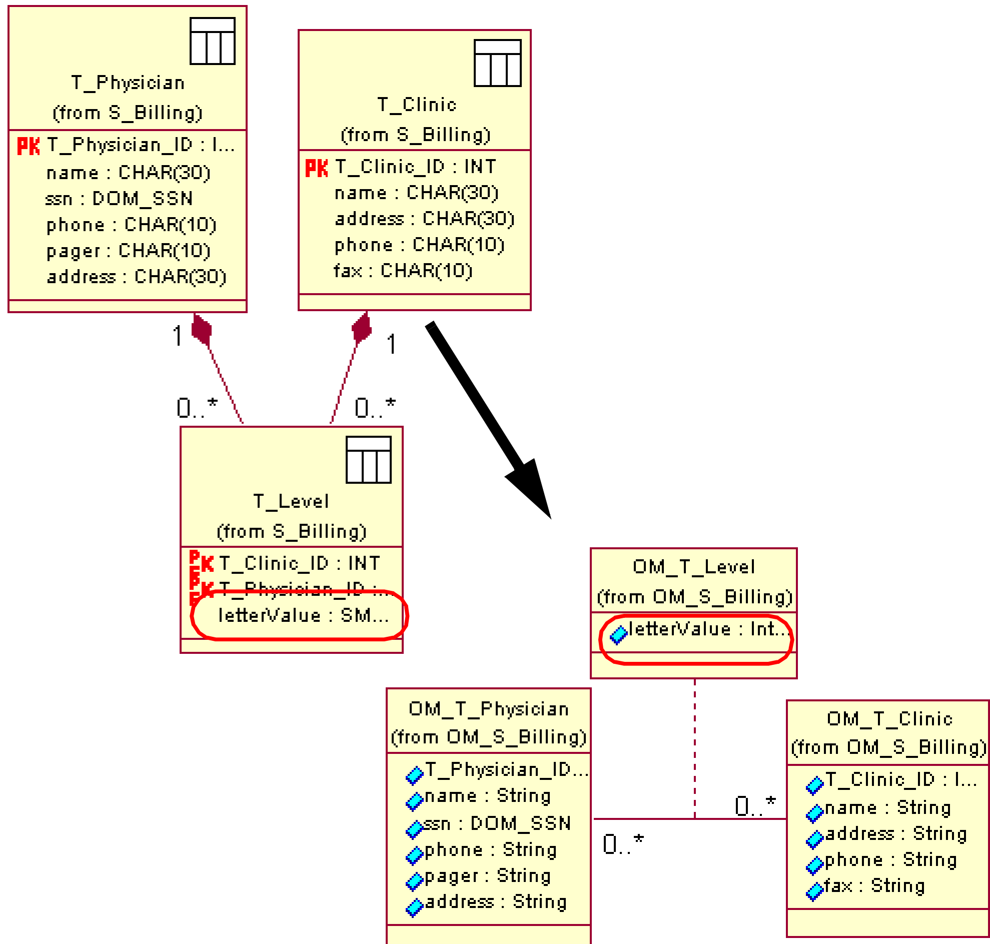
Figure 28 Intersection Tables Map to Qualified Associations



Association Classes

An intersection table also maps to an association class of a many-to-many relationship when the intersection table contains an additional column that is not a primary key, foreign key, or primary/foreign key. This additional column identifies the intersection table outside of the relationship, so that neither table participating in the relationship can own the column, yet both share the column. The association class uses the name of the intersection table as its name.

Figure 29 Intersection Tables Map to Association Classes



Mapping Supertype/Subtype Structures to Inheritance Structures

Supertype/subtype structures can map to inheritance structures, this is also known as a generalization. However in the transformation process, Data Modeler transforms supertype/subtype structures as separate classes. After the transformation process you can manually delete the composite relationships and replace them with the generalization structure also known as the inheritance tree.

Transforming a Data Model to an Object Model

After customizing or reviewing your data model to ensure you will receive the desired results in your object model, you can transform your data model to an object model.

Why Transform

As Chapter 3 explained, transforming is helpful to synchronize your physical data and logical data models and can be a means of communication between the development team members.

You can also transform to build an application from your database structure. When you transform to the object model, you can take that object model and assign its classes or entire package to an appropriate component language. Then you can create the additional boundary classes and interfaces you need to generate the application code in that component language.

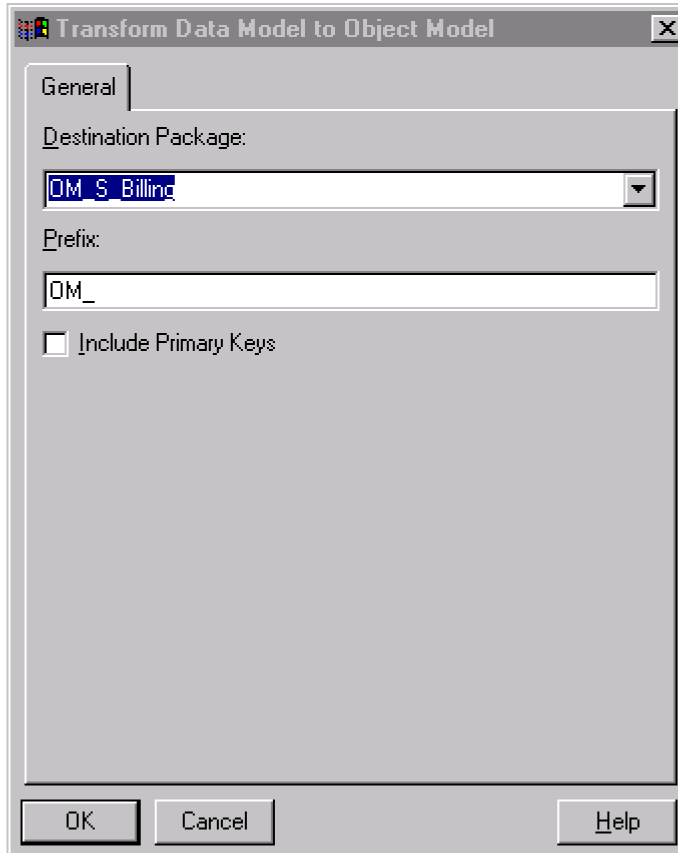
The Transformation Process

All data model elements transform to the object models they are mapped to. In the transformation process, Data Modeler will only transform your data model to the Analysis component language in the object model. If you want to use a different component language you must manually reassign your package and classes to that other language after the transformation process is complete.

Transform Data Model to Object Model Dialog Box

The process of transforming the data model begins in the **Data Model to Object Model Transformation** dialog box. In this dialog box, you can select or specify a schema name for the object model, a prefix for each of your object model classes (it is recommended you use a prefix to distinguish your object model classes from your data model tables), and whether or not you want the primary key columns to be included in the transformation.

Figure 30 Transform Data Model to Object Model Dialog Box



Transforming Data Model Elements

In the actual transformation process, Data Modeler reads the data model schema, and generates a package or executes a destructive rewrite if an existing package has the same name as the schema you are transforming. Then, Data Modeler creates a persistent class for each valid table in the data model with the exception of an intersection table.

At this point all foreign key columns are removed from the classes, and all other columns are transformed to attributes. If you designated in the **Transform Data Model to Object Model** dialog box to include primary key columns, these are also transformed to attributes. All constraints are removed from the attributes with the exception of primary key columns; they are tagged as part of object identity. Also any check constraints that contain enumerated clauses are transformed to classes with the stereotype <<ENUM>>.

Once the classes are established, Data Modeler transforms the relationships to associations or composite aggregations depending on the relationship type, and assigns the appropriate multiplicity and roles as designated in the data model.

After the Transformation Process

When the transformation process is complete, you can make any changes necessary in the object model, such as revising inheritance structures. You can also see the data model to object model mapping using the Mapped From property in the data model's Table or Column Specifications. Refer to *Chapter 3, Logical Data Modeling* for more information on the Mapped From property.

Implementing the Physical Data Model

6

Contents

This chapter is organized as follows:

- *Introduction* on page 59
- *Forward Engineering a Data Model* on page 59
- *Comparing and Synchronizing a Data Model* on page 62

Introduction

The final step in modeling a database is implementing the physical data model. You can implement a physical data model by forward engineering to create a new DDL or database, or you can compare and synchronize your model with an existing DDL or database, implementing only specific table changes.

Forward Engineering a Data Model

Before forward engineering your data model to a database, you must ensure your data model meets all your specific DBMS's requirements, especially for table name and column name length. You must also ensure your data model structure is stable, Data Modeler will not repair incorrect data structures.

Forward Engineering Wizard

You forward engineer your data model to a database or DDL file by using the Forward Engineering Wizard. The wizard guides you through and initiates the forward engineering process. It also guides you through connecting to your DBMS. Refer to *Appendix D* for information on connecting to your DBMS.

If the wizard encounters any error during the forward engineering process, Data Modeler logs the error in the Rose Log and continues the process. When the wizard informs you that the process is complete, you need to review the Rose Log for any errors that may have occurred.

Additionally the wizard offers you the option to include or ignore `CREATE` statements for comments, tables, indexes, stored procedures, and triggers, and also to include or ignore fully qualified names, quoted identifiers, and drop statements. Although all of these are important, fully qualified names, quoted identifiers, and drop statements can have a critical effect on your executable DDL.

Fully Qualified Names

Fully qualified names have a critical effect on your DDL because if you use fully qualified names in your DDL, you restrict flexibility by not allowing the use of other tools to execute your DDL. You should only use fully qualified names if you are going to forward engineer to a database, by executing your generated DDL.

Quoted Identifiers

Quoted identifiers have a critical effect on your DDL, because they allow you to use characters outside of the standard code set such as spaces. They allow you to use the double byte code set (DBCS) as part of your element names, by placing your element names in quotes. This is helpful when you must follow specific naming standards for your business or when you are working with international alphabets.

Drop Statements

Drop statements in conjunction with `CREATE Table` statements have a critical effect on your DDL, because if not specified correctly in your forward engineering they can cause errors or cause a destructive overwrite on your DDL. If you include tables and drop statements, all of your tables are dropped at the beginning of the DDL script and then the `CREATE Table` statements are generated. If you include tables, but do not include drop statements, your `CREATE Table` statements will begin at the final point of your existing DDL script, creating possible errors because you may have duplicate tables in your DDL. If you do not include tables, but include drop statements, all your tables are dropped at the beginning of your DDL script and no `CREATE Table` statements are generated thereafter, causing your DDL to be empty. All the DBMSs support drop statements with the exception of SQL Server and Sybase, which use a conditional drop statement. Refer to *Forward Engineering to a SQL Server Database or DDL* on page 61 or *Forward Engineering to a Sybase Database or DDL* on page 62 for more information on conditional drop statements.

Forward Engineering to the ANSI SQL 92 DDL

When you forward engineer the data model to an ANSI SQL 92 DDL file, Data Modeler reads the data model schema, and generates DDL statements for each modeled element. All of your documentation is recreated as comments. However,

stored procedures and triggers do not forward engineer and are not included in the DDL. A";" acts as your statement delimiter separating the statements in your DDL script.

Forward Engineering to a DB2 Database or DDL

When you forward engineer the data model to a DB2 database or DDL file, Data Modeler reads the data model schema, and generates a system catalog. All of your documentation is recreated as comments in your DBMS database. Each table is recreated in the database or DDL including the table's columns, selected constraints, and data type settings. A";" acts as your statement delimiter separating the statements in your DDL script.

Domains are recreated as distinct data types. If the domain is tagged as server-generated, then when it is forward engineered an additional `CREATE DISTINCT TYPE` statement is included providing information such as the domain name and data type. If the domain is not tagged as server-generated, then when it is forward engineered the domain column in the `CREATE Table` statement will only provide the data type, but not its associated domain name.

Forward Engineering to an Oracle Database or DDL

When you forward engineer the data model to an Oracle database or DDL file, Data Modeler reads the data model schema, and generates a user catalog. Documentation for your tables and columns is recreated as comments in your DBMS database. Each table is recreated in the database or DDL including the table's columns, selected constraints, and data type settings. Clustered indexes are recreated as index organization tables (IOT). You can specify a ";" or a "/" to act as your statement delimiter, separating the statements in your DDL script.

Forward Engineering to a SQL Server Database or DDL

When you forward engineer the data model to a SQL Server database or DDL file, Data Modeler reads the data model schema, and generates a system catalog. None of your documentation forward engineers. However, each table is recreated in the database or DDL including the table's columns, selected constraints, and data type settings. "Go" acts as your statement delimiter separating the statements in your DDL script.

Domains are recreated as user-defined data types. If the domain contains enumerated check constraints or default values, one of those check constraints is forward engineered as a rule, all others remain check constraints. The domain default value becomes the default for that user-defined data type. If the domain is tagged as server-generated, then when it is forward engineered an additional `EXEC`

`sp_addtype` statement is included providing information such as the domain name, data type, and length. If the domain is not tagged server-generated, then when it is forward engineered the domain column in the `CREATE Table` statement will only provide the data type, but not the associated domain name.

If you included drop statements in your forward engineering, they will act as conditional drop statements. If the table exists it is dropped, if it does not exist then no drop statement is created and therefore errors do not occur in your DDL script.

Forward Engineering to a Sybase Database or DDL

When you forward engineer the data model to a Sybase database or DDL file, Data Modeler reads the data model schema, and generates a system catalog. None of your documentation forward engineers. However, each table is recreated in the database or DDL including the table's columns, selected constraints, and data type settings. "Go" acts as your statement delimiter separating the statements in your DDL script.

Domains are recreated as user-defined data types. If the domain contains enumerated check constraints or default values, one of those check constraints is forward engineered as a rule, all others remain check constraints. The domain default value becomes the default for that user-defined data type. If the domain is tagged as server-generated, then when it is forward engineered an additional `EXEC sp_addtype` statement is included providing information such as the domain name, data type, and length. If the domain is not tagged server-generated, then when it is forward engineered the domain column in the `CREATE Table` statement will only provide the data type, but not the associated domain name.

If you included drop statements in your forward engineering, they will act as conditional drop statements. If the table exists it is dropped, if it does not exist then no drop statement is created and therefore errors do not occur in your DDL script.

Comparing and Synchronizing a Data Model

When implementing a data model that corresponds to an existing database, compare and synchronization is an optimal choice, because it updates your database only with the changes or differences per table, unlike forward engineering which makes changes to all tables in the schema. You compare and synchronize the data model with a database or DDL file by using the Data Model to Database Synchronization Wizard.

Synchronization Wizard

The synchronization wizard guides you through and initiates this process, offering you the option to include comments, indexes, triggers, and stored procedures. If you are generating a DDL, you can also include fully qualified names and quoted identifiers. The wizard also guides you through connecting to a DBMS database. Refer to *Appendix D* for information on connecting to your database. The wizard divides this process into two parts—comparison and synchronization.

Comparison Process

In the comparison process, Data Modeler reads the data model schema and the DBMS database. Then, Data Modeler compares these two schemas, visually listing the differences for each database element and each data model element in the synchronization wizard. These two lists are placed side by side, grouped by table, and match the element names for the data model to the element names for the DBMS database. If there is an item in the data model that does not exist in the DBMS database, then the item is placed on a line by itself under the data model list and the other list is left blank, and vice versa. If the difference is about an element in a table such as a column, only that differing column and the table it belongs to are listed, the other elements of the table are not included in the list.

The wizard allows you to set an action for each line item. These actions indicate where an element is updated, in the data model or in the database. If you do not set any actions, then no changes will be made to your database or your data model. If you want to update your database schema with the element in the data model, you can set this action to Export. However, you must consider that any changes made to a database column or any of its characteristics will require you to first drop your original table from the database, which may result in the loss of data. If you want to update your data model schema with the element in the database schema, you can set this action to Import. If you want to delete the element from both the data model and the database schema, you can set this action to Delete. When you set an action it applies to the entire table, but if you do not want specific elements in your table to be synchronized, then you can indicate an Ignore action on those elements.

The synchronization wizard also allows you to save the results of this comparison to a .txt file. You can use this .txt file as it is or you can open it using a word processing or spreadsheet application for clearer formatted reports.

Synchronization Process

After you set the actions for each element, the wizard allows you to review your proposed changes before beginning the actual synchronization process. The wizard also allows you to provide a path for your DDL and gives you the option of executing

your DDL after it is generated. In the synchronization process, Data Modeler reads the synchronization options and updates the data model and DBMS database according to the actions you specified. If there are changes to the database or DDL, a new DDL with only those changes is created and can be executed to update the existing DBMS database.

Appendix A

UML Data Modeling Profile

This table maps database concepts to data modeling stereotypes assigned to existing UML structures and is called the UML Data Modeling Profile.

Table 2 UML Data Modeling Profile Stereotypes

Database Concept	UML Structure	Data Modeling Stereotype
Database	Component	<<Database>>
Schema	Package	<<Schema>>
Table (entity)	Class	<<Table>>
Domain	Class	<<Domain>>
Relationship	Association	<<Non-Identifying>>
Strong Relationship	Composite Aggregate	<<Identifying>>
Index	Operation	<<Index>>
Primary Key Constraint	Operation	<<PK>>
Foreign Key Constraint	Operation	<<FK>>
Unique Constraint	Operation	<<Unique>>
Check Constraint	Operation	<<Check>>
Trigger	Operation	<<Trigger>>
Stored Procedure	Utility Class	<<SP>>

Appendix B

Object to Data Model Data Type Mapping

This appendix contains tables mapping each supported object model language with each of the supported DBMS data types. This is the same mapping Data Modeler uses when transforming an object model to a data model.

The following tables are contained:

- *Analysis Object to Data Model Data Type Mapping* Table 3 on page 68
- *Java Object to Data Model Data Type Mapping* Table 4 on page 69
- *Visual Basic Object to Data Model Data Type Mapping* Table 5 on page 70

Table 3 Analysis Object to Data Model Data Type Mapping

Analysis	ANSI SQL 92	DB2	Oracle	SQL Server	Sybase
STRING	VARCHAR (255)	VARCHAR (255)	VARCHAR (255)	VARCHAR (255)	VARCHAR (255)
INTEGER	INTEGER	INTEGER	NUMBER (10,0)	INT	INT
DOUBLE	DOUBLE PRECISION	DOUBLE	FLOAT	FLOAT	FLOAT
DATE	DATE	TIMESTAMP	DATE	DATAETIME	DATAETIME
BOOLEAN	SMALLINT	SMALLINT	NUMBER (1,0)	BIT	BIT
BYTE	SMALLINT	SMALLINT	NUMBER (3,0)	SMALLINT	SMALLINT
SINGLE	FLOAT	REAL	FLOAT	FLOAT	FLOAT
LONG	INTEGER	BIGINT	NUMBER (20,0)	INT	INT
CURRENCY	DOUBLE PRECISION	DOUBLE	FLOAT	MONEY	MONEY

Table 4 Java Object to Data Model Data Type Mapping

Java	ANSI SQL 92	DB2	Oracle	SQL Server	Sybase
LONG	INTEGER	BIGINT	NUMBER (20,0)	INT	INT
CHAR	CHAR	GRAPHIC (1)	CHAR	CHAR (1)	CHAR (1)
INT	INTEGER	INTEGER	NUMBER (10,0)	INT	INT
SHORT	SMALLINT	SMALLINT	NUMBER (5,0)	SMALLINT	SMALLINT
DOUBLE	FLOAT	DOUBLE	FLOAT)	FLOAT	FLOAT
BYTE	SMALLINT	SMALLINT	NUMBER (3,0)	SMALLINT	SMALLINT
FLOAT	FLOAT	REAL	FLOAT	FLOAT (0)	FLOAT
BOOLEAN	SMALLINT	SMALLINT	NUMBER (1,0)	BIT	BIT
Jjava.util.Date	DATE	TIMESTAMP	DATE	DATETIME	DATETIME
java.lang.String	VARCHAR (255)	VARCHAR (255)	VARCHAR2 (255)	CHAR (255)	CHAR (255)

Table 5 Visual Basic Object to Data Model Data Type Mapping

Visual Basic	ANSI SQL 92	DB2	Oracle	SQL Server	Sybase
STRING	VARCHAR (255)	VARCHAR (255)	VARCHAR (255)	VARCHAR (255)	VARCHAR (255)
INTEGER	INTEGER	INTEGER	NUMBER (10,0)	INT	INT
DOUBLE	DOUBLE PRECISION	DOUBLE	FLOAT	FLOAT	FLOAT (0)
COLLECTION	SMALLINT	SMALLINT	NUMBER (5)	SMALLINT	SMALLINT
DECIMAL	FLOAT	REAL	FLOAT	FLOAT	FLOAT
BYTE	DATE	TIMESTAMP	DATE	DATETIME	DATETIME
SINGLE	FLOAT	REAL	FLOAT	FLOAT	FLOAT
BOOLEAN	SMALLINT	SMALLINT	NUMBER (1,0)	BIT	BIT
BYTE	SMALLINT	SMALLINT	NUMBER (3,0)	SMALLINT	SMALLINT
LONG	INTEGER	BIGINT	NUMBER (20,0)	INT	INT
CURRENCY	DOUBLE PRECISION	DOUBLE	FLOAT	MONEY	MONEY

Appendix C

Data to Object Model Data Type Mapping

This appendix contains tables mapping each supported DBMS's data types with the Analysis, Java, and Visual Basic languages. Data Modeler *only* transforms to the Analysis language. The other languages are for mapping reference only.

The following tables are contained:

- *ANSI SQL 92 Data to Object Model Data Type Mapping* Table 6 on page 72
- *DB2 Data to Object Model Data Type Mapping* Table 7 on page 73
- *Oracle Data to Object Model Data Type Mapping* Table 8 on page 74
- *SQL Server Data to Object Model Data Type Mapping* Table 9 on page 75
- *Sybase Data to Object Model Data Type Mapping* Table 10 on page 77

Table 6 SQL 92 Data Model to Object Model Data Type Mapping

ANSI SQL 92	Analysis	VB	Java
BIT	BOOLEAN	BOOLEAN	BOOLEAN
BIT VARYING	BYTE	BYTE	BYTE
CHAR	STRING	STRING	STRING
DATE	DATE	DATE	DATE
DECIMAL	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE	BYTE SHORT INTEGER LONG FLOAT DOUBLE
DOUBLE PRECISION	DOUBLE	DOUBLE	DOUBLE
FLOAT	SINGLE DOUBLE	SINGLE DOUBLE	SINGLE DOUBLE
INTEGER	INTEGER	INTEGER	INTEGER
INTERVAL	STRING	STRING	STRING
REAL	SINGLE	SINGLE	FLOAT
SMALLINT	INTEGER	INTEGER	INTEGER
TIME	DATE	DATE	DATE
TIME WITH TIME ZONE	DATE	DATE	DATE
TIMESTAMP	DATE	DATE	DATE
TIMESTAMP WITH TIME ZONE	DATE	DATE	DATE
VARCHAR	STRING	STRING	STRING

Table 7 DB2 Data to Object Model Data Type Mapping

DB2	Analysis	VB	Java
SMALLINT	BYTE OBJECT VARIANT	BYTE OBJECT COLLECTION DECIMAL	BYTE SHORT
INTEGER	INTEGER BOOLEAN	INTEGER BOOLEAN	BOOLEAN INT
BIGINT	LONG	LONG	LONG
REAL	SINGLE	SINGLE	FLOAT
DOUBLE	DOUBLE CURRENCY	DOUBLE	DOUBLE
DECIMAL	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE
CHARACTER	STRING	STRING	STRING
VARCHAR	STRING	STRING	STRING
LONG VARCHAR	STRING	STRING	STRING
GRAPHIC	STRING	STRING	CHAR
VARGRAPHIC	STRING	STRING VARIANT	STRING
LONG VARGRAPHIC	STRING	STRING	STRING
DATE	DATE	DATE	DATE
TIME	DATE	DATE	DATE
TIMESTAMP	DATE	DATE	DATE
BLOB	STRING	STRING	STRING
CLOB	STRING	STRING	STRING
DBCLOB	STRING	STRING	STRING

Table 8 Oracle Data to Object Model Data Type Mapping

Oracle	Analysis	VB	Java
CHAR	STRING	STRING	java.lang.String
VARCHAR2	STRING	STRING	java.lang.String
NCHAR	STRING	STRING	java.lang.String
NVARCHAR2	STRING	STRING	java.lang.String
NUMBER (1,0)	BYTE	BYTE	BYTE
NUMBER (3,0)	BYTE	BYTE	BYTE
NUMBER (5,0)	INTEGER	INTEGER	SHORT
NUMBER (10,0)	LONG	LONG	INT
NUMBER (20,0)	LONG	LONG	LONG
NUMBER (m, n) (n cannot equal zero)	SINGLE DOUBLE	SINGLE DOUBLE	FLOAT DOUBLE
FLOAT (n)	SINGLE DOUBLE	SINGLE DOUBLE	FLOAT DOUBLE
LONG	STRING	STRING	java.lang.String
LONG RAW	STRING	STRING	java.lang.String
RAW	STRING	STRING	java.lang.String
DATE	DATE	DATE	java.util.Date
BLOB	STRING	STRING	java.lang.String
CLOB	STRING	STRING	java.lang.String
NCLOB	STRING	STRING	java.lang.String
BFILE	STRING	STRING	java.lang.String
ROWID	LONG	LONG	LONG

Table 9 SQL Server Data to Object Model Data Type Mapping

SQL Server	Analysis	VB	Java
BIT	BOOLEAN	BOOLEAN	BOOLEAN
INT	INTEGER LONG	INTEGER LONG	INT LONG
SMALLINT	BYTE OBJECT VARIANT	BYTE OBJECT COLLECTION	BYTE SHORT
TINYINT	BYTE	BYTE	BYTE
DECIMAL	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE
NUMERIC	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE	BOOLEAN
REAL	SINGLE	SINGLE	FLOAT
MONEY	CURRENCY	CURRENCY	DOUBLE
SMALLMONEY	CURRENCY	CURRENCY	DOUBLE
DATETIME	DATE	DATE	DATE
SMALLDATETIME	DATE	DATE	DATE
TIMESTAMP	LONG	LONG	LONG
UNIQUEIDENTIFIER	LONG	LONG	LONG
CHAR	STRING	STRING	STRING
VARCHAR	STRING	STRING	STRING
TEXT	STRING	STRING	STRING
BINARY	STRING	STRING	STRING
VARBINARY	STRING	STRING	STRING
IMAGE	STRING	STRING	STRING

SQL 7.0 only Data to Object Attribute Type Mapping

SQL Server	Analysis	VB	Java
NTEXT (ver. 7.0 only)	STRING	STRING	STRING
NVARCHAR (ver. 7.0 only)	STRING	STRING VARIANT	STRING
NCHAR (ver. 7.0 only)	STRING	STRING	CHAR

Table 10 Sybase Data to Object Data Type Mapping

Sybase	Analysis	VB	Java
BIT	BOOLEAN	BOOLEAN	BOOLEAN
INT	INTEGER LONG	INTEGER LONG	INT LONG
SMALLINT	BYTE OBJECT VARIANT	BYTE OBJECT COLLECTION	BYTE SHORT
TINYINT	BYTE	BYTE	BYTE
DECIMAL	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE
NUMERIC	BYTE INTEGER LONG SINGLE DOUBLE	BYTE INTEGER LONG SINGLE DOUBLE	BOOLEAN
REAL	SINGLE	SINGLE	FLOAT
MONEY	CURRENCY	CURRENCY	DOUBLE
SMALLMONEY	CURRENCY	CURRENCY	DOUBLE
DATETIME	DATE	DATE	DATE
SMALLDATETIME	DATE	DATE	DATE
TIMESTAMP	LONG	LONG	LONG
UNIQUEIDENTIFIER	LONG	LONG	LONG
CHAR	STRING	STRING	STRING
VARCHAR	STRING	STRING	STRING
TEXT	STRING	STRING	STRING
BINARY	STRING	STRING	STRING
VARBINARY	STRING	STRING	STRING
IMAGE	STRING	STRING	STRING

Appendix D

Database Connections

This appendix contains information on connecting to your database when using any of Data Modeler's engineering wizards. You can connect to any of the supported databases when you use the reverse engineer, the forward engineer, or the synchronize data model to database wizards. It is important that you have the correct user privileges for each of the engineering processes. When you are reverse engineering a database you only need read privileges for your system or master catalog. If you are forward engineering or synchronizing the data model, you need read/write privileges to the actual schema or database affected.

DB2

Before you connect to a DB2 database you must ensure you have your DB2 client and ODBC driver installed. You can use the IBM DB2 ODBC driver (or other third party DB2 ODBC drivers).

Note: Ensure you use the ODBC driver that matches your version of DB2.

You must specifically configure your ODBC driver for DB2. If your ODBC driver is configured correctly your data source name appears in the **Data Source** drop-down lists in the Data Modeler wizards. If you are configuring your ODBC driver for initial use, test your database connection from the ODBC Data Source Administrator. For all subsequent database connections test your connection in the Data Modeler wizards.

Connecting to DB2 Databases

Data Modeler wizards require the name of the ODBC data source, a user name, and a password. The user name and corresponding password indicate your privileges for reading or using the database. You will only be allowed to select the databases that are authorized by your user name. If you are reverse engineering a DB2 UDB database you must have read privileges to the syscat.* file. If you are reverse engineering a DB2 MVS database you must have read privileges to the sysIBM.* file.

If you are using DB2 7.0, you can use NT authentication for your user name and password. You do this by leaving the user name and password blank. When you use NT authentication Data Modeler queries your operating system for your user identification. This method of single sign on is helpful for database administrators.

Oracle

Before you connect to an Oracle database you must ensure your Oracle client and Oracle Config service is installed. You must configure your Oracle Net8 or Net Easy Config.

Connecting to Oracle Databases

Data Modeler wizards require a service name, a user name, and a password. The service name indicates the Oracle Server name or Oracle Easy Config Service name. The user name and corresponding password indicate your privileges for reading or using the database. You are only allowed to select the databases that are authorized by your user name.

Note: Always test your database connection before continuing in any of the wizards.

SQL Server

Before you connect to a SQL Server database you must ensure you have the SQL Server client. Your OLE DB driver should have been installed with your installation of Rose. You must configure your OLE DB driver specifically for SQL Server.

Connecting to SQL Server Databases

Data Modeler wizards require a server name, database name, a user name, and a password. The server name is the name of the server running SQL Server. If the server name is your local machine you can leave the server name blank, or enter the local machine name.

The user name and its associated password indicate your privileges for reading or using the database. You are only allowed to select the databases that are authorized by your user name. If you have system administrator privileges, you will have access to the master database and it will appear in the drop-down list for **Database Name**. If you do not have system administrator privileges you must enter the database name manually.

You can use NT authentication for your user name and password, if you leave the user name and password blank and your SQL Server's server supports NT authentication. When you use NT authentication Data Modeler queries your operating system for your user identification. This method of single sign on is helpful for database administrators.

Note: Always test your database connection before continuing in any of the wizards.

Sybase

Before you connect to a Sybase database you must ensure you have your Sybase client and ODBC driver installed. You must specifically configure your ODBC driver for Sybase. If your ODBC driver is configured correctly your data source name appears in the **Data Source** drop-down lists in the Data Modeler wizards. If you are configuring your ODBC driver for initial use, test your database connection from the ODBC Data Source Administrator. For all subsequent database connections test your connection in the Data Modeler wizards.

Connecting to Sybase Databases

Data Modeler wizards require a data source, database name, a user name, and a password. The data source is the name of the Sybase data source. The user name and its associated password indicate your privileges for reading or using the database. You are only allowed to select the databases that are authorized by your user name. If you have system administrator privileges, you will have access to the master database and it will appear in the drop-down list for **Database Name**. If you do not have system administrator privileges you must enter the database name manually.

Note: Always test your connection before continuing in any of the wizards.

Index

A

aggregations 17
- mapping to non-identifying relationships 18

ANSI SQL 92
- data models 30
- forward engineering DDL 60

application designer role 1
applications
- communicating changes 24
- shown in object models 12
- working with legacy 9

association classes
- intersection table mappings 54
- mapping to intersection tables 20

associations
- many-to-many 19, 52
- mapping to non-identifying relationships 18
- qualified 21, 53
- transforming to data model relationships 26

attribute types
- mapping to data types 17

attributes
- as qualifiers 21
- assigning to classes 12
- mapping to columns 14
- mapping to domains 16
- transforming as columns 26

B

business analyst role 1
business requirements
- and data entities 11

C

C++ 9
- DB2 stored procedures 43
- parameter styles for stored functions 43
- stored procedures 42

candidate key columns 16
candidate keys
- transforming to primary keys 26

cardinality 37
- foreign keys 37

check constraints 34
- mapping to classes 49

child classes 22
child tables 35
- intersection tables 38
- roles 37

class diagrams
- components 13
- described 11
- mapping to data model diagrams 9

classes 11
- assigning attributes 12
- association 20, 54
- child 22
- dependent 17
- mapping to tables 14
- parent 17, 22
- persistent 14
- transforming to tables 26
- transient 14

columns
- candidate key 16
- computed 32
- creating in data model 32
- data columns 32
- described 32
- domain 16
- ID-based 15
- key constraints 33
- mapping to attributes 13, 49
- primary keys 15

comparing data models 62

described 63
components
described 13
transforming 24
composite aggregations
see also aggregations 17
computed columns 32
constraints
check 33, 34
creating in data models 33
described 33
key 33
unique 33
Create Table statement 60
custom triggers
creating in data models 41
defined 41

D

data
check constraints 34
data columns 32
data model diagrams 7, 29
creating in data models 30
described 7
mapping to class diagrams 9
data model elements
transforming 57
data modeling
logical 11
data modeling profiles 5
data models 29
building new 30
comparing and synchronizing 62
defined 29
forward engineering 59
mapping to applications 12
mapping to object models 9, 13
relationships 30
reverse engineering to create 44
transforming to object models 56
data types
mapping to attributes types 17
stored parameters for stored functions 43
stored procedure parameters 43
database designer role 2
databases
assigning schemas 30
creating in data model 30
default targets 30
modeling 11
specifying for transformation 26
target 30
databases,
working with legacy 9
DB2
forward engineering to databases or DDL 61
parameter styles for stored functions 43
reverse engineering databases or DDL 44
stored procedures 43
DB2 triggers 41
DBMS 6
data types 17, 32
DB2 41, 43, 44, 61
domain columns 16
domains 31
generating new 24
mapping to languages 9
Oracle 41, 44, 61
physical data models 29
referential integrity 40
see also Oracle 35
SQL Server 45, 61
Sybase 45, 62
target databases 30
working with legacy databases 9
DDL
ANSI SQL 92 60
DB2 61
drop statements 60
fully qualified names 60
Oracle 61
quoted identifiers 60
SQL Server 61
Sybase 62
declarative referential integrity 40
deferrable check constraints 35
dependent classes 17

diagrams
  data model 7, 29, 30
  mapping between data model and class 9
domain columns 16
domain packages
  assigning to DBMS 31
domains
  check constraints 34
  creating in data models 30
  data types 30
  DB2 61
  described 30
  mapping to attribute types 48
  mapping to attributes 16
  server-generated 31
  SQL Server 61
  Sybase 62
drop statements 60
  SQL Server 62
  Sybase 62

E

embedded primary/foreign keys 35
entity/relationship notation 5
enumerated check constraints
  mapping to classes 49
events
  triggers 41

F

foreign key columns
  mapping 49
foreign key constraints 34
foreign keys
  cardinality 37
  child tables 35
  embedding 35
  referential integrity 40
  relationships 36
  specifying indexes 26
  transforming data model elements 57
forward engineering
  data models 9
forward engineering data models 59
Forward Engineering Wizard 59
frameworks 8
fully qualified names 60

I

ID-based columns 15
  creating in transformation 26
identifying relationships 35
  mapping to aggregations 17
  mapping to composite aggregations 50
In 12
indexes constraints 34
inheritance
  mapping 22
intersection tables 19, 38
  mapping 52
  mapping to association classes 20
  mapping to association structures 48
  mapping to qualified associations 21

J

Java 9
  parameter styles for stored functions 43
  specifying for components 13
  stored procedures 42
joining tables 19

K

key constraints
  creating in data models 34
  described 33
  foreign key 34
  indexes 33, 34
  primary key 33
keys
  embedding 35

L

logical data modeling 11
logical data models 6
logical packages 14

M

mandatory non-identifying relationships 36
many-to-many associations
  intersection table mappings 52
many-to-many relationships
  association classes 54
Mapped From text box
  using during transformation 27
mapping
  association classes to intersection tables 20
  attributes to domains 16
  components to languages 13
  inheritance 22
  object models to data models 13
  operations 13
  packages 14
  qualified associations to intersection tables 21
modeling databases 11
  physical data modeling 29
models
  synchronizing 24
  transforming 9

N

non-deferrable check constraints 35
non-identifying relationships
  mandatory 36
  mapping to aggregations 18
  mapping to associations 18, 51
  optional 36
  structures 39
Not Deterministic stored functions 43
notations
  entity/relationship 5

O

object models
  components 13
  forward engineering 9
  mapping to data models 9, 13
  representing applications 12
  reverse engineering 9
  transforming to data models 24
operations
  mapping 13
  transforming 24
optional non-identifying relationships 36
Oracle
  check constraints 35
  forward engineering to databases or DDL 61
  reverse engineering databases or DDL 44
Oracle triggers 41
orphan records
  preventing with referential integrity 40

P

packages 14
  transforming 26
parameters
  stored procedures 43
parent classes 17, 22
  aggregations 18
  associations 18
parent tables 35
  non-identifying relationships 36
  referential integrity 40
  relationships 36
  roles 37
persistent classes 14
physical data model
  implementing 59
physical data modeling 29
physical data models 6
  mapping 6
  see also data models 29
physical data models, transforming 6
primary key constraints 33
primary keys 15

and ID-based columns 15
computed columns 32
embedding 35
parent tables 35
qualified associations 53
primary/foreign keys 35, 52
profiles 5
programming languages
C 42, 43
C++ 9
Java 9, 13, 42, 43
SQL 42
Visual Basic 13

Q

qualified associations
intersection table mappings 52, 53
mapping to intersection tables 21
qualifiers 21
quoted identifiers 60

R

recursive relationships
self-referencing structures 39
referential integrity 39
actions 40
declarative (DRI) 40
parent tables 37, 40
system-generated (RI) triggers 40
relationship structures 38
relationships
cardinality 37
creating in data models 35
described 35
foreign keys 36
identifying 35
non-identifying 35, 36
parent tables 36
roles 37
reverse engineering
data models 9
reverse engineering data models 44
Reverse Engineering Wizard 44
roles
tables 37
Rose diagrams 7
rows
and indexes 34

S

schemas
assigning to databases 30
creating in data model 30
default standard 30
defined 30
mapping to packages 14, 47
reverse engineering to create 44
specifying for transformation 25
storage 30
SQL Server
forward engineering to databases or DDL 61
reverse engineering databases or DDL 45
stored procedures 42
stereotypes 5
entity 11
ENUM 49, 57
stored functions
data types 43
DB2 parameter styles 43
not deterministic 43
parameters 43
stored procedures
creating in data models 42
languages 42
parameters 43
stored functions 43
structures
mapping 55
self-referencing 39
supertype/subtype structures
mapping to inheritance structures 55
Sybase
forward engineering to databases or DDL 62
reverse engineering databases or DDL 45
Synchronization Wizard 63

synchronizing
  data models 62
synchronizing data models
  described 63
system triggers
  referential integrity 40
system-generated referential integrity (RI)
  triggers 40

T

tables
  cardinality in relationships 37
  check constraints 34
  creating in data models 31
  described 31
  intersection 19, 52
  mapping to classes 14, 48
  relationships 35
  schemas 31
  specifying prefixes for names 26
tablespace name
  tables 31
tagged values 5
target databases 30
  described 30
team roles
  application designer 1
  business analyst 1
  database designer 2
  dependencies 2
transformation 9
transforming
  associations 26
  classes 26
  object model to data model 24, 25
  packages 26
transforming data models to object models 56
transforming models 9
transient classes 14
trigger
  events 41
  granularity 42
  referencing 42
  types 41
  WhenClauses 42
trigger events
  defined 41
triggers
  creating in data models 41
  custom 41
  referential integrity 40

U

UML 12
  profiles 5
UML (Unified Modeling Language) 5
unique constraints 33

V

Visual Basic
  specifying for components 13

W

WhenClause trigger 42