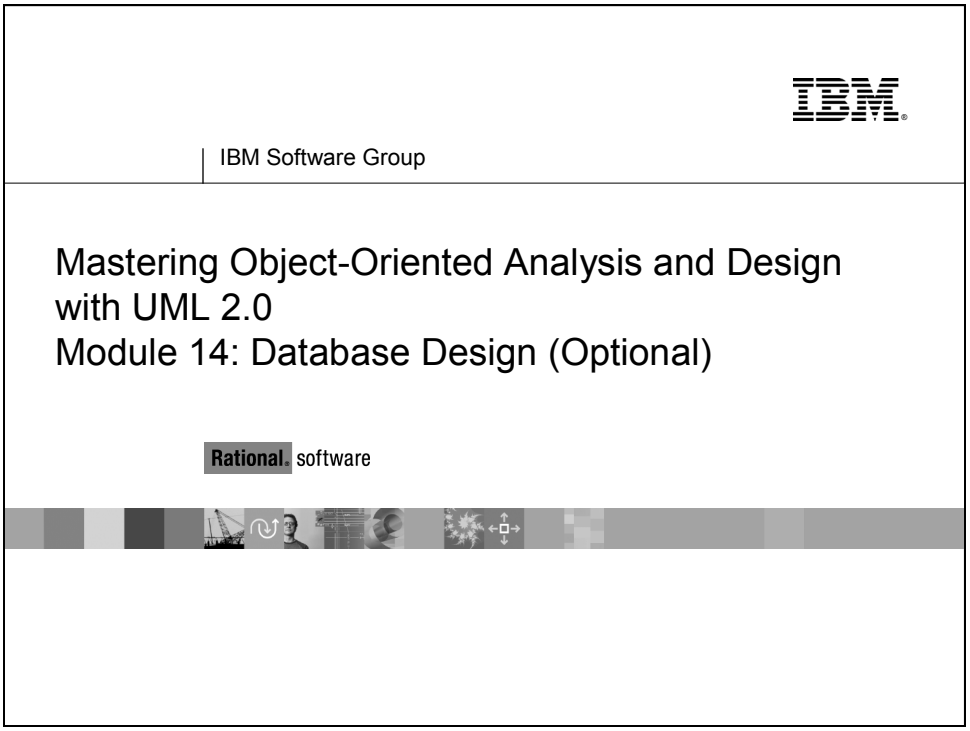
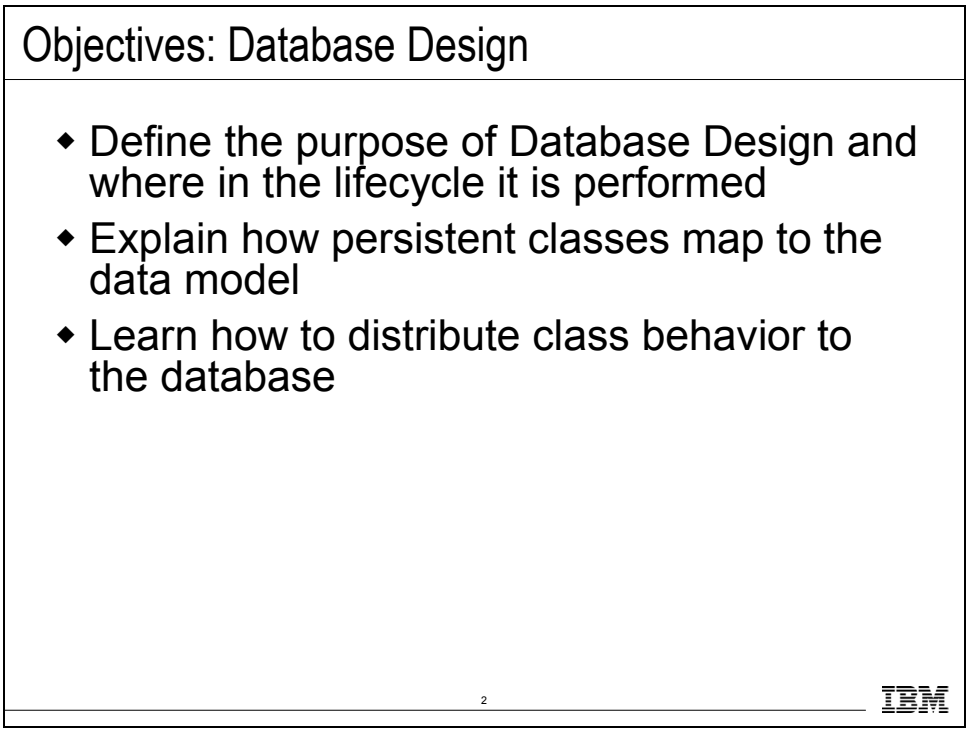
▸ ▸ ▸ Module 14
Database Design



Topics

Database Design Overview 14-4
Relational Databases and Object Orientation 14-7
Object-Relational Framework: Characteristics 14-11
What Are Stored Procedures? 14-19
Review 14-23

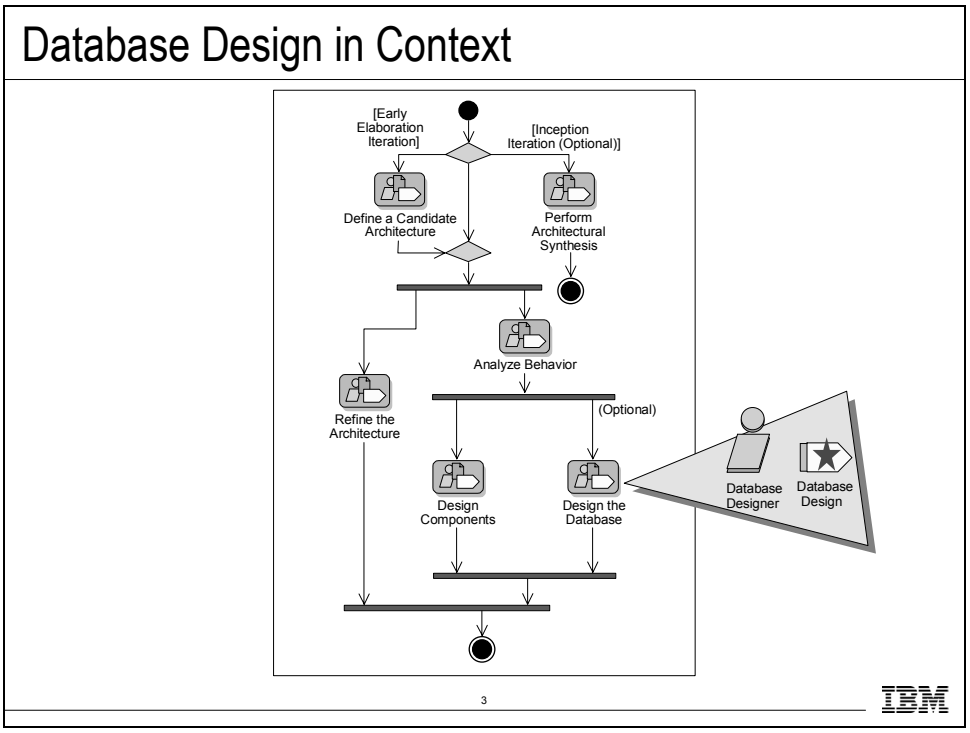
Objectives: Database Design



Persistent classes need to be related to tables in the data model. This step will ensure that there is a defined relationship between the design and data model.

This activity assumes the use of a Relational Database (RDBMS).

Database Design in Context



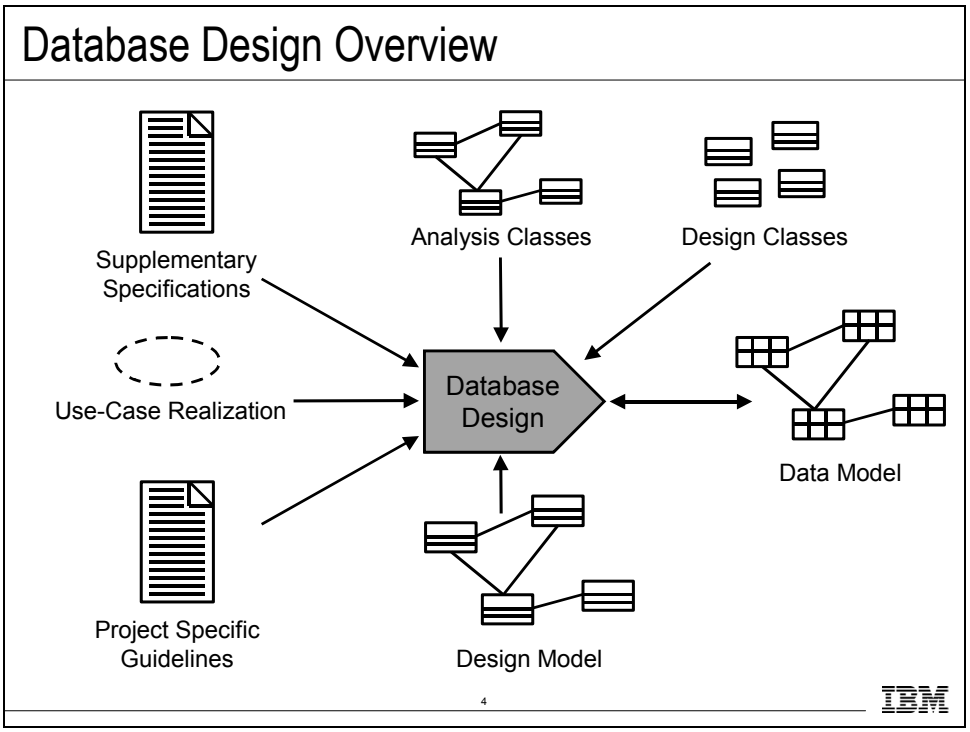
As you may recall, the above diagram illustrates the workflow that we are using in this course. It is a tailored version of the Analysis and Design core workflow of the Rational Unified Process.

The purpose of this workflow detail is to:

- Identify the persistent classes in the design.
- Design appropriate database structures to store the persistent classes.
- Define mechanisms and strategies for storing and retrieving persistent data to meet the system's performance criteria.

The designers responsible for persistent classes need to have an understanding of persistence in general and the persistence mechanisms in specific. Their primary responsibility is to ensure that persistent classes are identified and that these classes use the persistence mechanisms in an appropriate manner. The database designer needs to understand the persistent classes in the design model and so must have a working understanding of object-oriented design and implementation techniques. The database designer also needs a strong background in database concurrency and distribution issues.

Database Design Overview



Database Design is performed for each persistent design class.

Purpose:

- To ensure that persistent data is stored consistently and efficiently.
- To define behavior that must be implemented in the database.

Input Artifacts:

- Supplementary Specifications
- Use-Case Realization
- Project Specific Guidelines
- Data Model
- Analysis Class
- Design Class
- Design Model

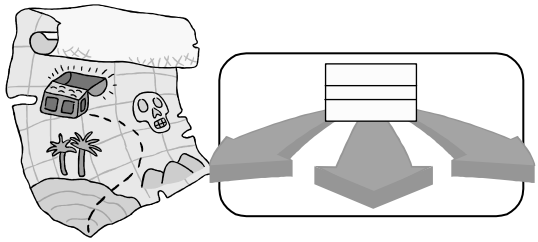
Resulting Artifacts:

- Data Model

Database Design Steps

Database Design Steps

- Map persistent design classes to the data model
- Distribute class behavior to the database



5

IBM

This slide shows the major steps of the **Database Design** activity.

The purpose of **Database Design** is to ensure that persistent data is stored consistently and efficiently, and to define behavior that must be implemented in the database.

There are other steps in **Database Design**, but they are outside the scope of this course. The two steps are the only ones that apply to transitioning persistent design elements to the Data Model.

Database Design Steps

Database Design Steps

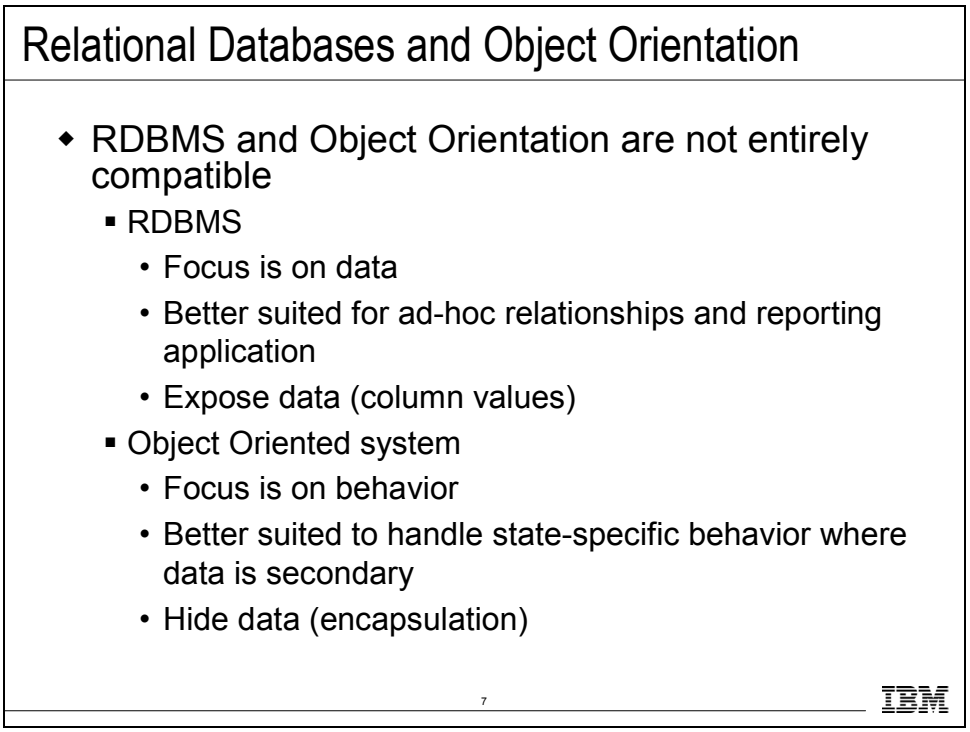
- ☆ Map persistent design classes to the data model
- Distribute class behavior to the database

6

IBM

In this step, our purpose is to create define and refine the Data Model to support storage and retrieval of persistent classes. This is done only after the design of the persistent classes has stabilized.

Relational Databases and Object Orientation

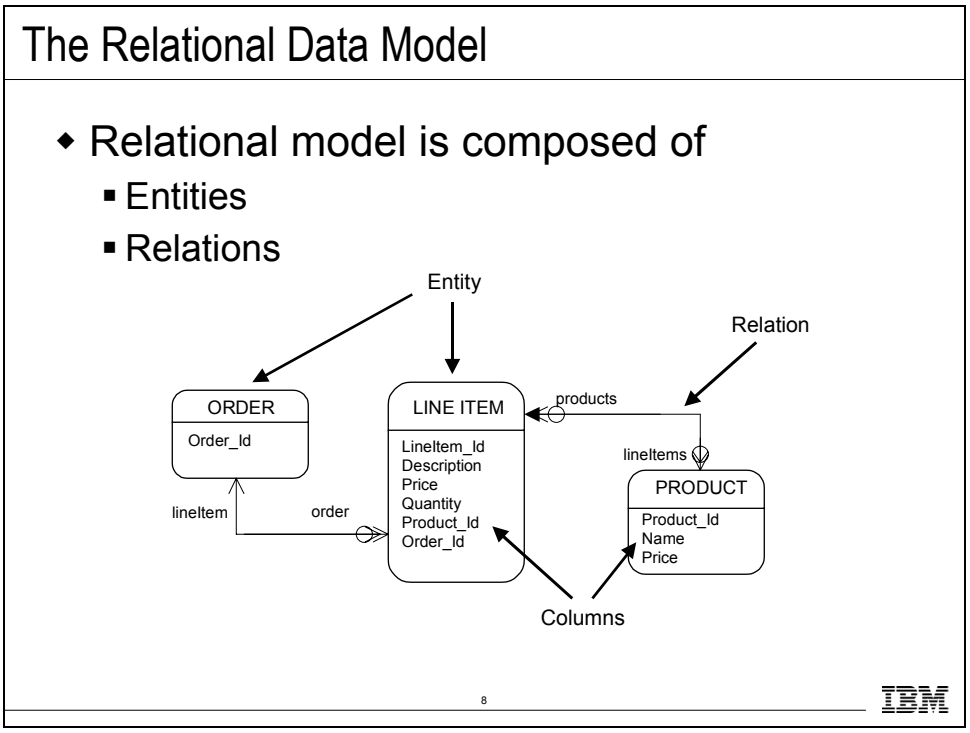


Relational databases and object orientation are not entirely compatible. They represent two different views of the world: In an RDBMS, all you see is data; in an object-oriented system, all you see is behavior. It is not that one perspective is better than the other, but they are different. The object-oriented model tends to work well for systems with complex behavior and state-specific behavior in which data is secondary, or systems in which data is accessed navigationally in a natural hierarchy (for example, bills of materials). The RDBMS model is well suited to reporting applications and systems in which the relationships are dynamic or ad hoc.

The real fact of the matter is that a lot of information is stored in relational databases, and if object-oriented applications want access to that data, they need to be able to read and write to an RDBMS. In addition, object-oriented systems often need to share data with non-object-oriented systems. It is natural, therefore, to use an RDBMS as the sharing mechanism.

While object-oriented and relational design share some common characteristics (an object's attributes are conceptually similar to an entity's columns), fundamental differences make seamless integration a challenge. The fundamental difference is that data models expose data (through column values) while object models hide data (encapsulating it behind its public interfaces).

The Relational Data Model



The relational model is composed of entities and relations. An entity may be a physical table or a logical projection of several tables, also known as a view. The figure above illustrates LINEITEM and PRODUCT tables and the various relationships between them.

An entity has columns. Each column is identified by a name and a type. In the figure above, the LINEITEM entity has the columns LineItem_Id (the primary key), Description, Price, Quantity, Product_Id and Order_Id (the latter two are foreign keys that link the LINEITEM entity to the ORDER and PRODUCT entities).

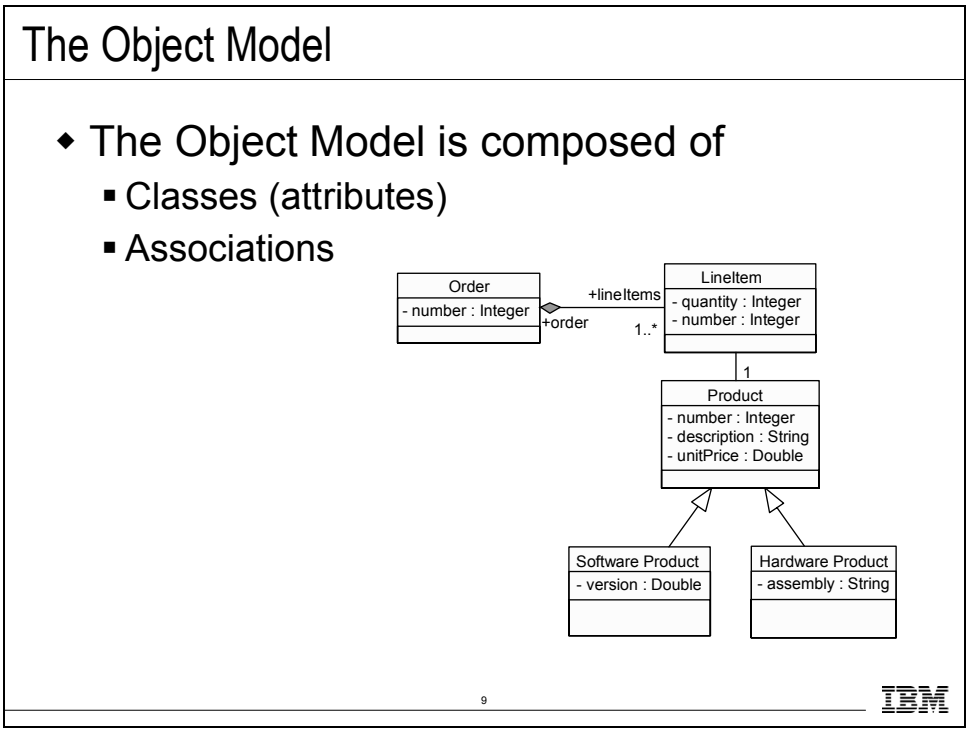
An entity has records or rows. Each row represents a unique set of information that typically represents an object's persistent data.

Each entity has one or more primary keys. The primary keys uniquely identifiy each record (for example, Id is the primary key for LINEITEM table).

Support for relations is vendor-specific. The example illustrates the logical model and the relation between the PRODUCT and LINEITEM tables. In the physical model relations are typically implemented using foreign key and primary key references. If one entity relates to another, it will contain columns that are foreign keys. Foreign key columns contain data that can relate specific records in one entity to another.

Relations have multiplicity (also known as cardinality). Common cardinalities are one to one (1:1), one to many (1:m), many to one (m:1), and many to many (m:n). In the example, LINEITEM has a 1:1 relationship with PRODUCT, while PRODUCT has a 0:m relationship with LINEITEM.

The Object Model



An Object Model contains, among other things, classes. Classes define the structure and behavior of a set of objects, sometimes called object instances. The structure is represented as attributes (data values) and associations (relationships between classes). The figure in the slide illustrates a simple class diagram, showing only attributes (data) of the classes.

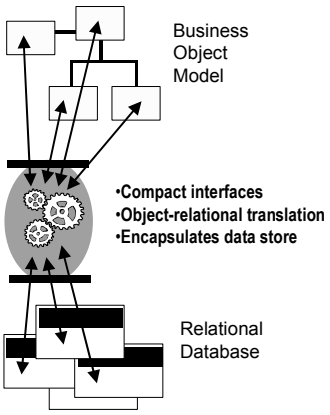
An Order has a number (the Order Number), and an association to one or more (1..*) LineItems. Each LineItem has a quantity (the quantity ordered).

The Object Model supports inheritance. A class can inherit data and behavior from another class (for example, SoftwareProduct and HardwareProduct inherit attributes and methods from Product class).

Persistence Frameworks

Persistence Frameworks

- The challenge:
  - Changes should not break the model
- The solution: An object-relational framework that
  - Encapsulates the physical data store
  - Provides object translation services
- The importance of the framework
  - 30% of development time is spent in accessing an RDBMS
  - Maintenance can be 60% of total cost



10

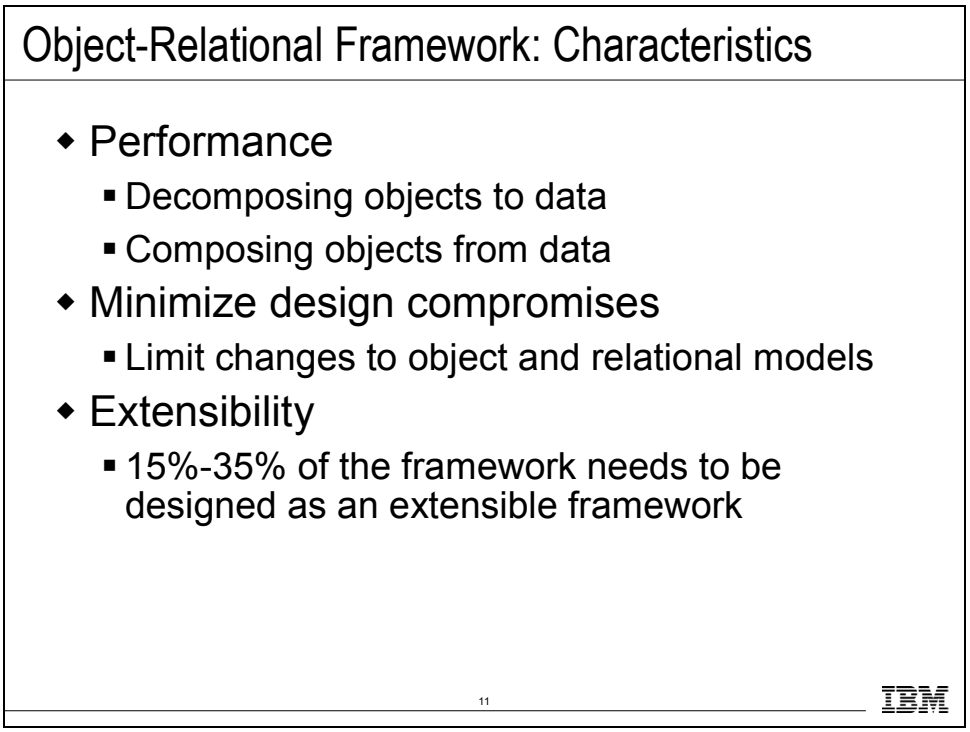
IBM

The majority of business applications use relational technology as a physical data store. The challenge facing object-oriented application developers is to sufficiently separate and encapsulate the relational database so that changes in the Data Model do not "break" the Object Model, and vice versa. Many solutions exist which let applications directly access relational data. The challenge is in achieving a seamless integration between the Object Model and the Data Model.

Database APIs come in standard flavors (for example, Microsoft's Open Data Base Connectivity API, or ODBC) and are proprietary (native bindings to specific databases). The APIs provide data manipulation language (DML) pass-through services that allow applications to access raw relational data. In object-oriented applications, the data must undergo object-relational translation prior to being used by the application. This requires a considerable amount of application code to translate raw database API results into application objects. The role of the object-relational framework is to generically encapsulate the physical data store and to provide appropriate object translation services.

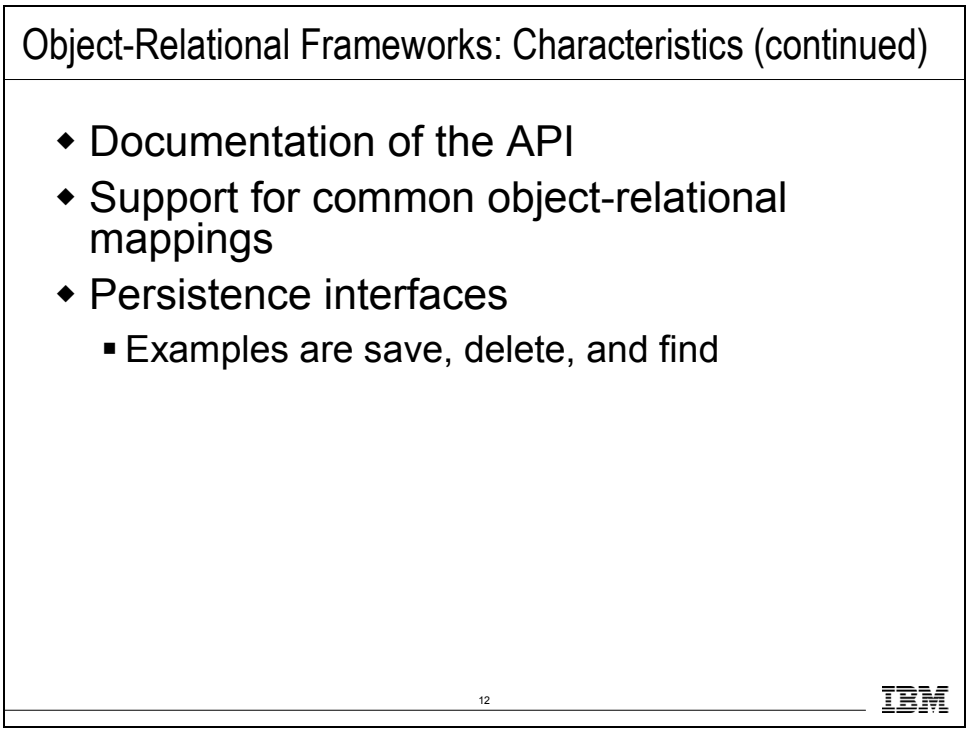
Application developers spend over 30% of their time implementing relational database access in object-oriented applications. If the object-relational interface is not correctly implemented, the investment is lost. Implementing an object-relational framework captures this investment. The object-relational framework can be reused in subsequent applications, reducing the object-relational implementation cost to less than 10% of the total implementation costs. The most important cost to consider when implementing any system is maintenance. Over 60% of the total costs of a system over its entire life cycle can be attributed to maintenance. A poorly implemented object-relational system is both a technical and financial maintenance nightmare.

Object-Relational Framework: Characteristics



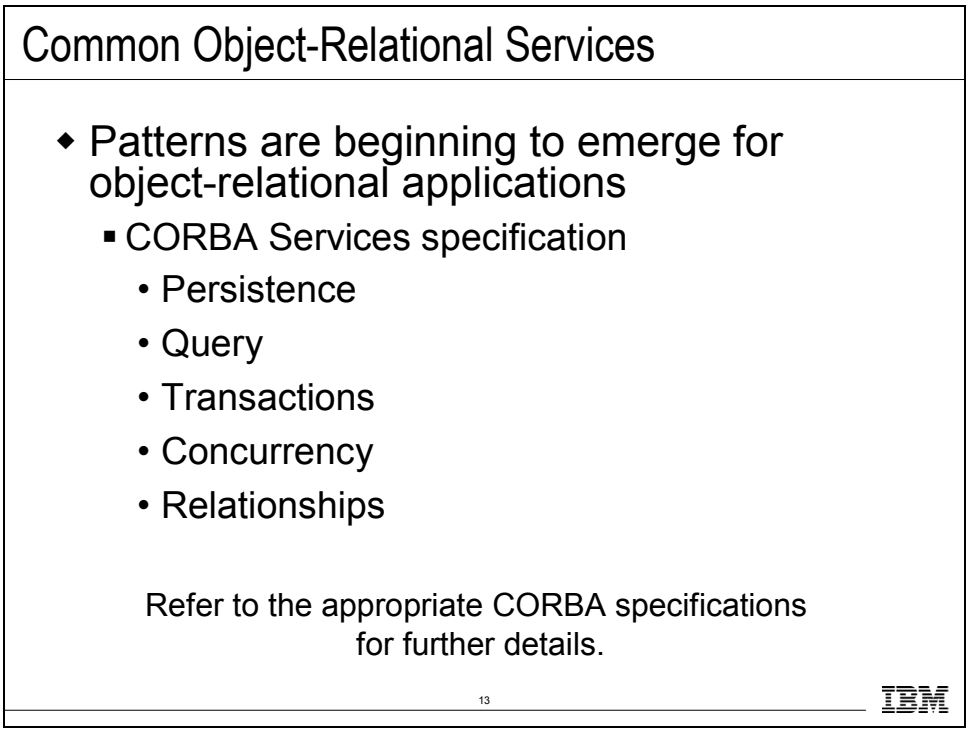
- **Performance**: Close consideration must be given to decomposing objects into data and composing objects from data. In systems where data through-put is high and critical, this is often the Achilles heel of an inadequately designed access layer.
- **Minimize design compromises**: A familiar pattern to object technologists who have built systems that use relational databases is to adjust the object model to facilitate storage into relational systems and to alter the relational model for easier storage of objects. While minor adjustments are often needed, a well designed access layer minimizes both object and relational model design degradation.
- **Extensibility**: The Access layer is a white-box framework that allows application developers to extend the framework for additional functions. Typically, an Access layer will support 65-85% of an application's data storage requirements without extension. If the access layer is not designed as an extensible framework, achieving the last 15-35% of an application's data storage requirements can be very difficult and costly.

Object-Relational Frameworks: Characteristics (continued)



- **Documentation**: The Access layer is both a black-box component and a white-box framework. The API of the black-box component must be clearly defined, well documented, and easily understood. As previously mentioned, the Access layer is designed to be extended. An extensible framework must be very thoroughly documented. Classes intended to be subclassed, must be identified. The characteristics of each relevant class's protocol must be specified (for example, public, private, protected, final). Moreover, a substantial portion of the access layer framework's design must be exposed and documented to facilitate extensibility.
- **Support for common object-relational mappings**: An Access layer should provide support for some basic object-relational mappings without the need for extension.
- **Persistence Interfaces**: In an object-oriented application, the business model for an object application captures semantic knowledge of the problem domain. Developers should manipulate and interact with objects without having to worry too much about the data storage and retrieval details. A well-defined subset of persistent interfaces (save, delete, find) should be provided to application developers.

Common Object-Relational Services



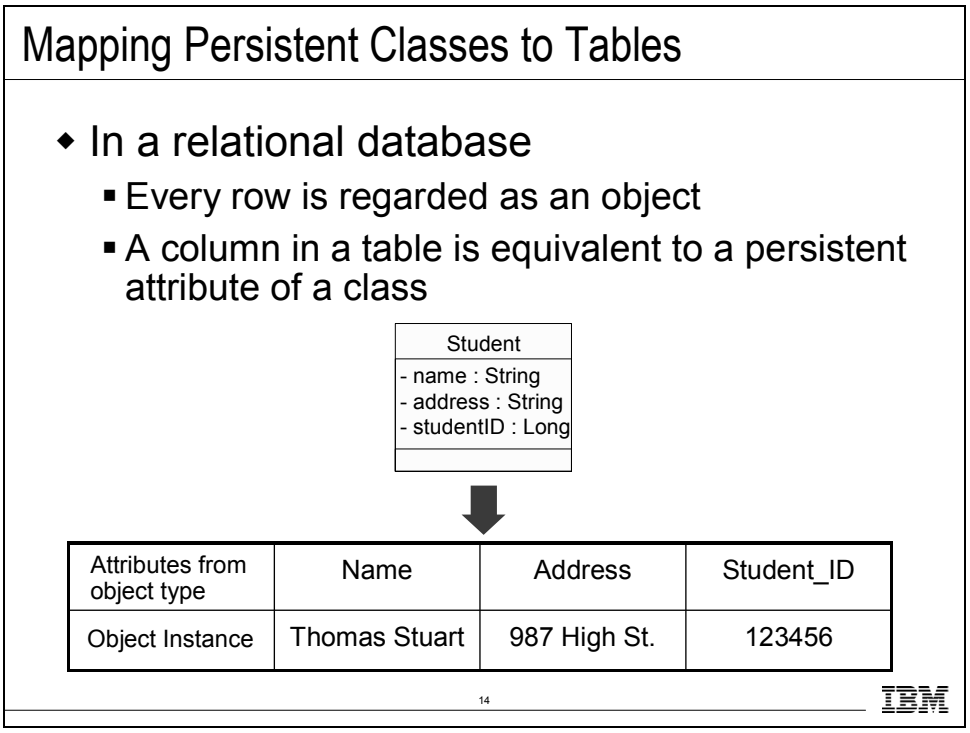
Common patterns are emerging for object-relational applications. IT professionals who have repeatedly crossed the chasm are beginning to understand and recognize certain structures and behaviors which successful object-relational applications exhibit. These structures and behaviors have been formalized by the high-level CORBA Services specifications (which apply equally well to COM/DCOM-based systems).

The CORBA service specifications that are useful to consider for object-relational mapping are:

- **Persistence:** How objects use a secondary storage medium to maintain their state across discrete sessions.
- **Query:** Allows applications to interrogate and retrieve objects based on a variety of criteria. The basic query operations provided by an object-relational mapping framework are “find” and “find unique.”
- **Transactions:** Enable the definition of an atomic unit of work. In database terminology, it means that the system must be able to apply a set of changes to the database, or it must ensure that none of the changes are applied.
- **Concurrency:** When an object is accessed simultaneously by many users, the system must provide a mechanism to ensure that modifications to the object in the persistent store occur in a predictable and controlled manner.
- **Relationships:** When an object is stored, retrieved, transacted, or queried consideration must be given to its related objects.

Refer to the appropriate CORBA specifications for further details.

Mapping Persistent Classes to Tables

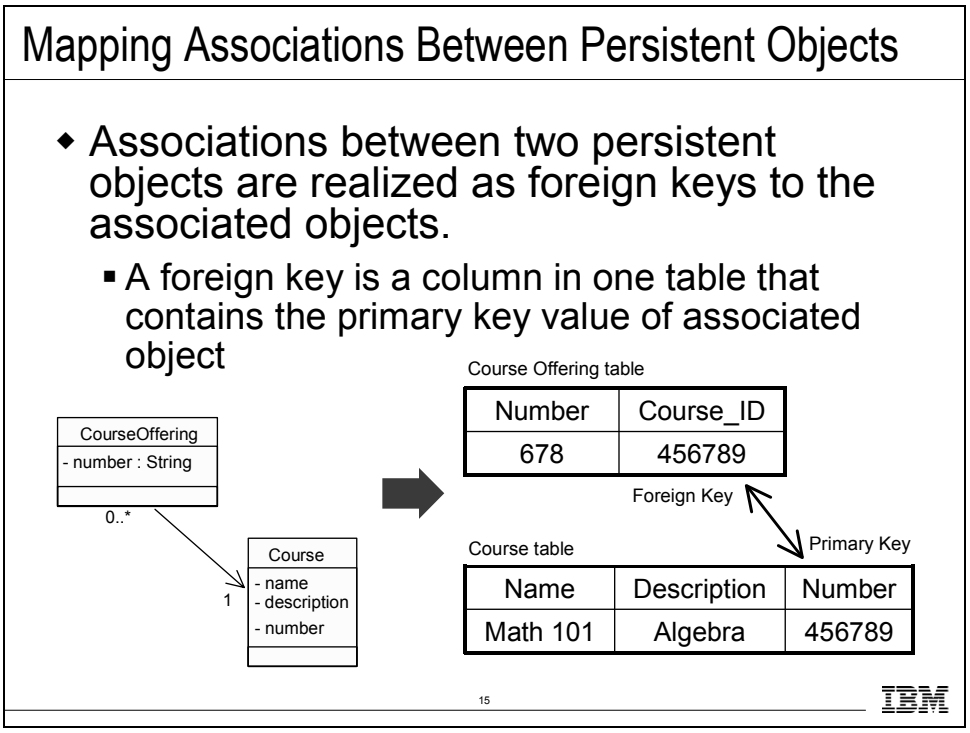


The persistent classes in the Design Model represent the information the system must store. Conceptually, these classes might resemble a relational design (for example, the classes in the Design Model might be reflected in some fashion as entities in the relational schema). As we move from elaboration into construction, however, the goals of the Design Model and the Relational Data Model diverge. The objective of relational database development is to normalize data, whereas the goal of the Design Model is to encapsulate increasingly complex behavior. The divergence of these two perspectives — data and behavior — leads to the need for mapping between related elements in the two models.

In a relational database written in third normal form, every row in the tables — every "tuple" — is regarded as an object. A column in a table is equivalent to a persistent attribute of a class (keep in mind that a persistent class may have transient attributes). So, in the simple case where we have no associations to other classes, the mapping between the two worlds is simple. The data type of the attribute corresponds to one of the allowable data types for columns.

In the example in this slide, the class Student when modeled in the RDBMS would translate to a table called Student, with the columns Name, Address, and Student_ID.

Mapping Associations Between Persistent Objects

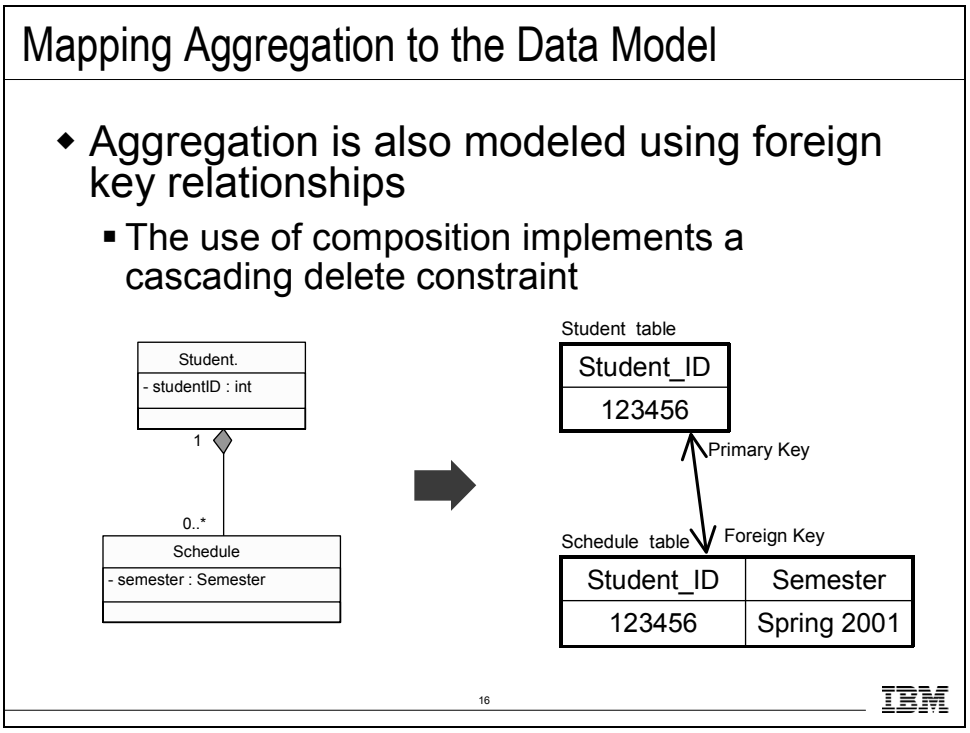


Associations between two persistent objects are realized as **foreign** keys to the associated objects. A **foreign key** is a column in one table that contains the **primary key** value of the associated object.

Assume we have the above association between Course and CourseOffering. When we map this into relational tables, we get a Course table and a Course Offering table. The Course Offering table has columns for attributes listed, plus an additional Course_ID column that contains foreign-key references to the primary key of associated rows in the Course table. For a given Course Offering, the Course_ID column contains the identifier of the Course with which the Course Offering is associated. Foreign keys allow the RDBMS to **join** related information together.

Note: The Number attribute of the Course class is a string. For optimal performance, it is best to convert this to a number in the database schema (queries and joins work faster on numbers than strings, since number comparisons are faster in general than string comparisons).

Mapping Aggregation to the Data Model



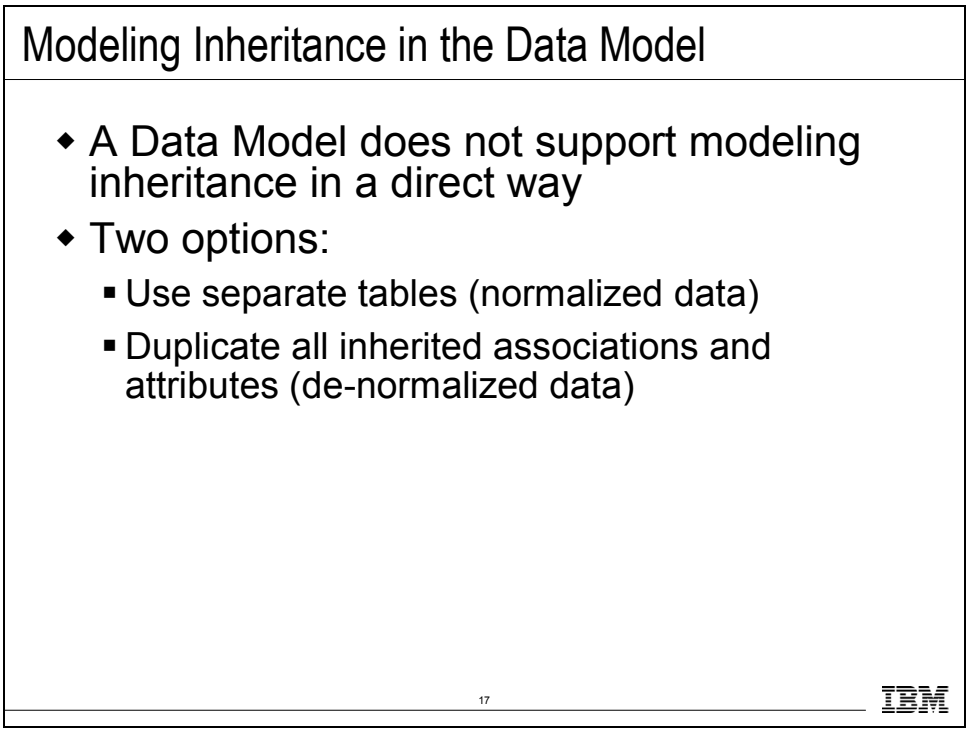
Aggregation is also modeled using foreign key relationships.

Assume we have the above aggregation between Student and Schedule. (Note: This is modeled as a composition, but remember that composition is a nonshared aggregation).

When we map this into relational tables, we get a Student table and a Schedule table. The Schedule table has columns for attributes listed, plus an additional column for Student_ID that contains foreign-key references to associated rows in the Student table. For a given Schedule, the Student_ID column contains the Student_ID of the Student that the Schedule is associated with. Foreign keys allow the RDBMS to **join** related information together.

In addition, to provide referential integrity in the Data Model, we would also want to implement a **cascading delete** constraint, so that whenever the Student is deleted, all of its Schedules are deleted as well.

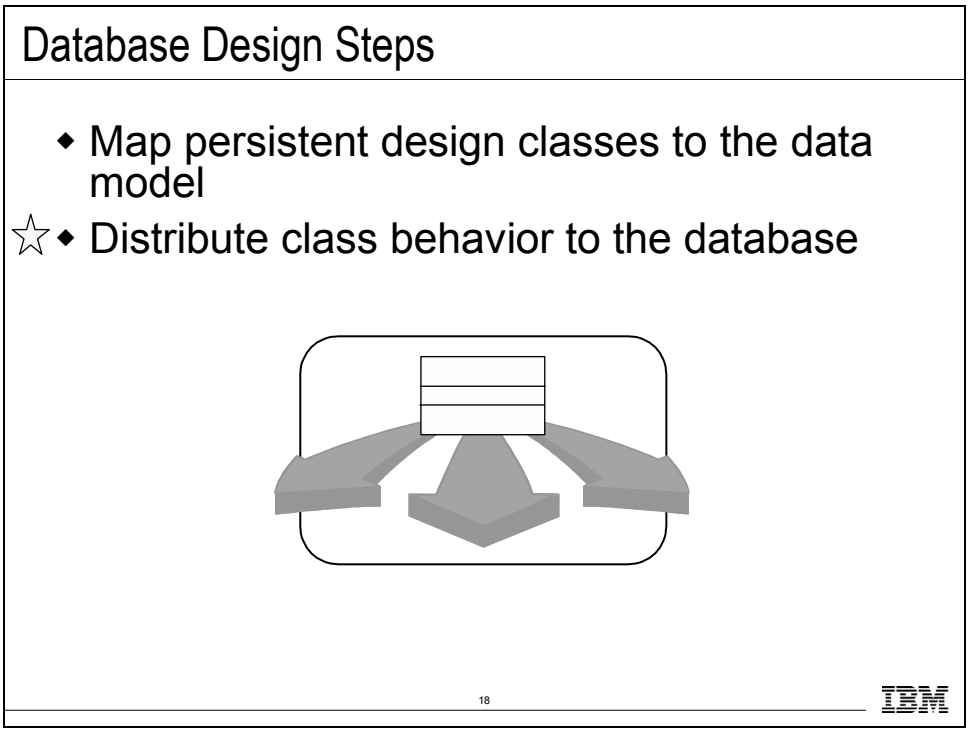
Modeling Inheritance in the Data Model



The standard relational Data Model does not support modeling inheritance associations in a **direct** way. A number of strategies can be used to model inheritance:

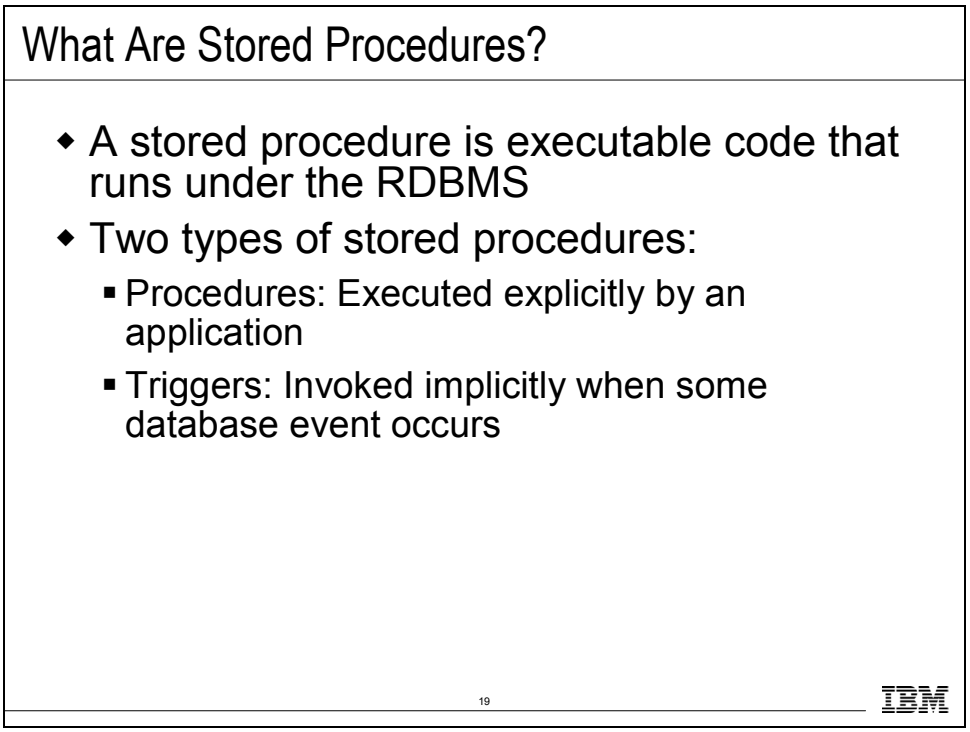
- Use separate tables to represent the super-class and subclass. Have, in the subclass table, a foreign key reference to the super-class table. In order to "instantiate" a subclass object, the two tables would have to be joined together. This approach is conceptually easier and makes changes to the model easier, but it often performs poorly due to the extra work.
- Duplicate all inherited attributes and associations as separate columns in the subclass table. This is similar to **de-normalization** in the standard relational Data Model.

Database Design Steps



In this step, our purpose is to determine the behavior of the class that can be distributed to and implemented by the database. This is done only after the design of the persistent classes has stabilized.

What Are Stored Procedures?



Most databases support a **stored procedure** capability. A stored procedure is executable code that runs within the process space of the database management system. Stored procedures provide the ability to perform database-related actions on the server without having to transfer data across a network. Their judicious use can improve performance of the system.

Stored procedures usually come in two flavors: actual procedures and triggers. Procedures are executed explicitly by an application. They generally have parameters and can provide an explicit return value. Triggers are invoked implicitly when some database event occurs (insert a row, update a row, delete a row, and so on). They have no parameters (because they are invoked implicitly) and do not provide explicit return values.

In database systems that lack constraints, triggers are often used to enforce referential and data integrity. Otherwise, they tend to be used when an event needs to trigger (or cause) another event.

Map Class Behavior to Stored Procedures

Map Class Behavior to Stored Procedures

- Determine if any operations can be implemented as a stored procedure
- Candidates:
  - Operations that deal with persistent data
  - Operations in which a query is involved in a computation
  - Operations that need to access the database to validate data

20

IBM

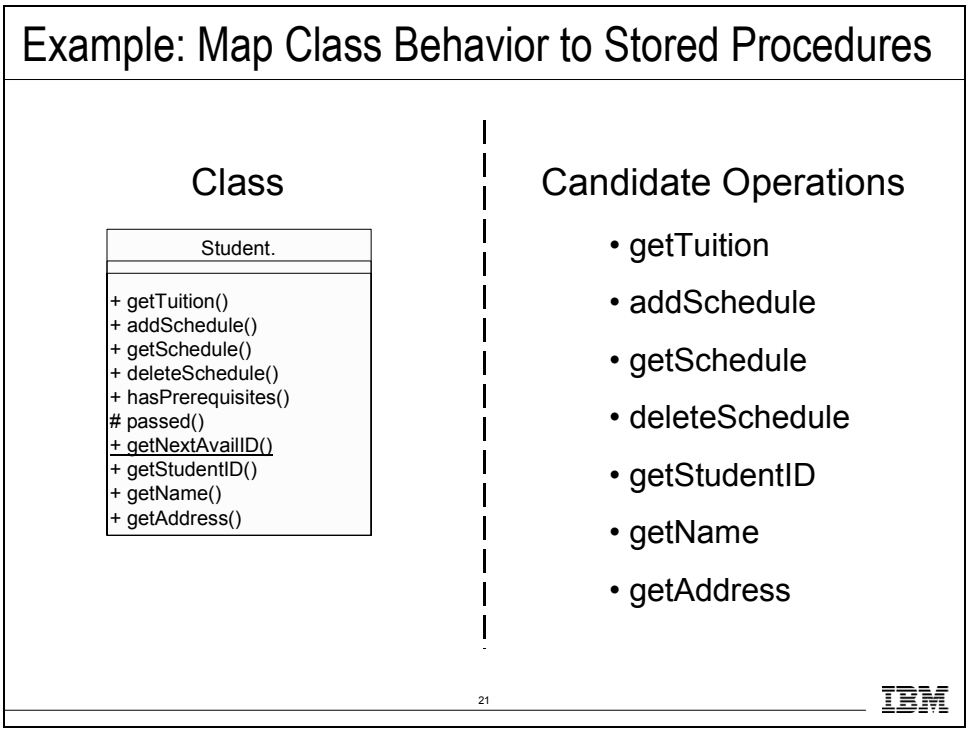
The Design classes should be examined to see if they have operations that should be implemented using a stored procedure or trigger. Candidates include:

- Any operations that primarily deal with persistent data (creating, updating, retrieving, or deleting it).
- Any operations in which a query is involved in a computation (such as calculating the average quantity and value of a product in inventory).
- Operations that need to access the database to validate data.

Remember that improving database performance usually means reducing I/O. As a result, if performing a computation on the DBMS server will reduce the amount of data passed over the network, the computation should probably be performed on the server.

The database designers should work with the designer of the class to discuss how the database can be used to improve performance. The designer will update the operation method to indicate that one or more stored procedures can be or should be used to implement the operation.

Example: Map Class Behavior to Stored Procedures



The class Student has been designed with a number of operations. The designer and database designer should sit down and determine which operations can be implemented by using a stored procedure.

Candidate operations for the Student class include:

- getTuition
- addSchedule
- getSchedule
- deleteSchedule
- getStudentID
- getName
- getAddress

The reason that these operations are considered candidates is that they deal with retrieving, creating, or deleting persistent data.

Checkpoints: Database Design

Checkpoints: Database Design

- Have all persistent classes been mapped to database structures?
- Have stored procedures and triggers been defined?
- Does the persistence mechanism use stored procedures and database triggers consistently?

22

IBM

Review

Review: Database Design

- What is the purpose of the Database Design?
- What comprises a relational data model?
- What are the components of an object model?
- When mapping persistent classes to tables, what is every row in a table regarded as? What is every column equivalent to?
- What are stored procedures?

23

IBM