: Database Design

Introduction

This topic exposes the student to aspects on designing and implementing a database. It takes the learner through the process of modelling data for a database and the phases involved in designing an effective database. It extends to explain the tools and software applications that a designer may use when practically designing the logical and physical models of a database.

Objectives

By the end of this topic you should be able to:

i. Describe the database design process
ii. Describe the data modelling process
iii. Explain the activities involved in the design process
iv. Analyze database management software

Topic 2 Notes

2.1 Data Modeling

What is data modeling?

A data model is a simply a diagram that describes the most important "things" in your business environment from a data-centric point of view.

When data modeling, you are telling the RDBMS the following:

i. what elements of the data you will store
ii. how large each element can be
iii. what kind of information each element can contain
iv. what elements may be left blank
v. which elements are constrained to a fixed range
vi. whether and how various tables are to be linked

Process

Identify Entity Types

An entity type, also simply called entity represents a collection of similar objects. An entity type could represent a collection of people, places, things, events, or concepts. Examples of entities in an order entry system would include Customer, Address, Order, Item, and Tax. If you were class modeling you would expect to discover classes with the exact same names. However, the difference between a class and an entity type is that classes have both data and behavior whereas entity types just have data.

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

Identify Attributes

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

Attributes should also be cohesive from the point of view of your domain, something that is often a judgment call. Getting the level of detail right can have a significant impact on your development and maintenance efforts.

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

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

Apply Data Naming Conventions
Your organization should have standards and guidelines applicable to data modeling, something you should be able to obtain from your enterprise administrators (if they don't exist you should lobby to have some put in place). These guidelines should include naming conventions for both logical and physical modeling, the logical naming conventions should be focused on human readability whereas the physical naming conventions will reflect technical considerations.

Normalize to Reduce Data Redundancy
Data normalization is a process in which data attributes within a data model are organized to increase the cohesion of entity types. In other words, the goal of data normalization is to reduce and even eliminate data redundancy, an important consideration for application developers because it is incredibly difficult to stores objects in a relational database that maintains the same information in several places.

Logical versus physical models
Logical data models — logical models capture general information about entities and relationships and are used for communication with business users.
LDMs are used to explore the domain concepts, and their relationships, of your problem domain. This could be done for the scope of a single project or for your entire enterprise. LDMs depict the logical entity types, typically referred to simply as entity types, the data attributes describing those entities, and the relationships between the entities.

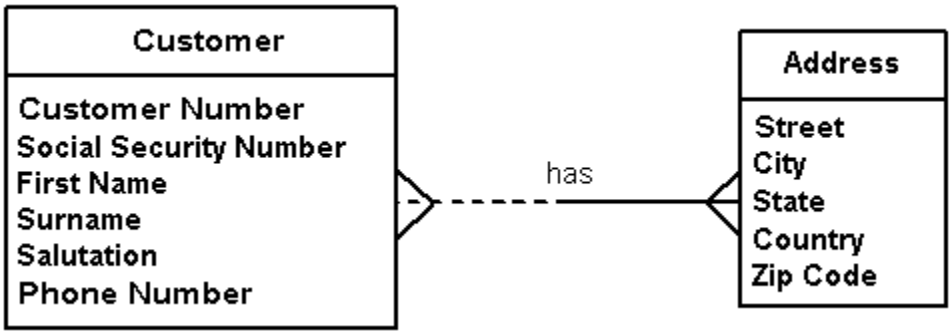

Fig 2.1 Customer Address Relationship

Physical data models

PDMs are used to design the internal schema of a database, depicting the data tables, the data columns of those tables, and the relationships between the tables. PDMs often prove to be useful on both Agile and traditional projects. physical models serve as a precise specification for the implemented system. As a consequence, the models must take into account the technology used to store the data. For example, a given logical data model translates into very different physical data models depending on whether the target technology is a file-based system, a relational database, or an object-oriented database.

Normally, you start with a high-level logical model and refine with the help of users over several iterations. Once you are happy with the logical model, you transform it into a physical model and hand it to a **database administrator** (DBA) for implementation as a database.

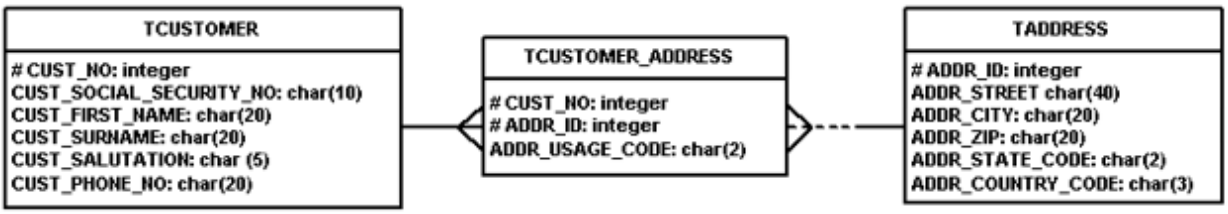

Fig 2.2 Physical Model

2.2 Database Design

There is a common belief that database design is done by the database administrator (DBA), while program design is done by programmers. While this may seem like a natural division of labor, it is actually quite unfortunate. Applications are made up of two equal and interrelated parts: data and processes. Neither has any purpose without the other and weaknesses in the design of either (or their intimate relationship) will be reflected in the entire system.

Design of the database and design of the total application require parallel efforts with frequent validations of each against the other. A tightly integrated and effective delineation/description of program modules can be rendered useless if the data needed by those modules is not structured for convenient access. Even worse is the discovery, as an application nears its final integration, that some data needed for a particular function isn't available or that some portion of the database has no means of being populated or maintained.

The solution is for the entire development team, analysts, programmers, and DBAs to work together with the system's eventual users to mutually discover both functional and data requirements and coordinate their implementation. DBAs need to gain an appreciation for modern programming constructs, while programmers need to understand the tenets of good database design.

Some parts of application development (systems engineering) are methodical and almost scientific. There are straightforward ways of performing certain common tasks that can be readily adopted to solve specific problems. Database design tends to have more guidelines and ideas than absolute rules. What rules do exist are always presented in the context of when they should be broken.

The only absolute rule of database design is that there are always alternatives. That is not to say that some alternatives are not better than others; clearly there are wrong designs. The point is that the relative merits of design alternatives are only evaluated in the context of how the database will be used. These "how" issues are the province of the process designers and programmers. No DBA can develop an optimal database design without intimate knowledge of the way all of the programs will need to access the data.

Even with knowledge of the programs' data needs, it is still necessary to evaluate alternatives in light of multiple often-conflicting priorities. Performance is generally one of the priorities, but then it is necessary to determine which of the programs' performance needs are most critical. Modifying the database design to optimize a particular access will inevitably make some other accesses less efficient.

2.2.1 Database Design Phases

Database design is not a single effort. Typically, the design of the database goes through multiple high-level phases. At each step it is necessary to review the current state of the database design and compare it to the current state of the processing design. At each phase the comparison becomes more detailed and exacting.

Conceptual Database Design

There is no significant reason to separate conceptual design from logical design. Some designers or methodologies move some preliminary activities into this phase, such as identifying key business drivers and high-level purposes of an application (and its database). The output of such a preliminary phase would typically be a few paragraphs (or, at most, pages) that describe the opportunity for improving a business process or provide a new service with an estimate of potential benefits and a statement of the scope of the effort. This can provide a useful perspective to set boundaries on the rest of the design processes to prevent expanding beyond what can be reasonably accomplished.

Logical Database Design

From a list of requirements (as they are discovered and documented), the team initially identifies a long list of data elements that they know must be included in the database. They additionally sort the data elements into groupings, informally at first and then with more rigor, around the entities of importance to the organization.

Logical design is performed without concern to the specific database management system that will eventually host the application's data. The logical database design for a system will look identical whether the eventual system will be hosted on DB2 on the mainframe or on SQL Server on a Windows workstation. That isn't to say that the eventual systems will be identical — it's just to say that those differences don't appear during logical design.

During the logical design phase the development team determines what data needs to be acquired, stored, and used by a particular application. Simultaneously, the team identifies what functional processes need to be performed by the application. As each model is refined, it is compared to the other to ensure that needed data for each process is included in the data model and that each element of data included in the data model has some identified processes that will create, use, and (usually) modify or purge the data.

For all of the data elements collected in this phase, it is necessary to begin documenting some basic characteristics of the data. You need to know where the data comes from (if it isn't being created by the new application), and you will want to gather some preliminary estimates of the type of data:

i. Is it strictly numeric? Will it always be integers or allow decimal values?
ii. Is the data item a date? If so, what granularity is required . . . to the day? to the minute? to the millisecond?
iii. Will the data be alphanumeric or will it need to include other characters?
iv. What if a needed element isn't character data but an image or some other unstructured object? If so, how large do you expect each instance to be?
v. Is there a defined domain (range or list of allowable values) associated with the data? Does the domain relate to other data expected to be in your system? Does the domain have an external definition rule (such as spatial coordinates limited to valid latitude and longitude pairs)?
vi. What is the expected lifetime of this particular data element? Is it transitory or does it continue to exist for a predictable period of time?

Gathering an initial shopping list of data elements is a necessary but relatively casual process during the early parts of logical system design. Some discipline, however, is soon needed. The primary means of organizing this rapidly growing list of data elements is *normalization* and is traditionally documented using *Entity-Relationship Diagram* (ERD) techniques.

When developers and DBAs are working effectively together, the ERDs may be supplemented with use case diagrams as well. Examples of both of these techniques are shown in the discussion of database design tools later in this excerpt.

Physical Design

The logical database design phase delivered a consistent structure of the data without regard to the eventual deployment within a specific database environment. It is possible to directly translate the logical design into a set of tables and build a database. It is usually appropriate, though, to take

advantage of the specific capabilities (and limitations) of the selected relational database management system (RDBMS).

Some shops have declared that they want to maintain RDBMS neutrality and therefore will not utilize any of the extended capabilities offered by any single RDBMS. This argument is naïve in at least three aspects.

i. First is the assumption that the "least common denominator" capabilities of all RDBMSs will be sufficient to satisfy the application's needs. The standard SQL language is the basis of all current relational database systems, but SQL does not include specifications for much of the necessary infrastructure of an actual database. Additionally, there have been several revisions of the SQL standard as well as different levels of compliance by commercial products for each revision. Two RDBMSs may both be SQL-compliant but incompatible because of the different subsets of functionality that they implement. Discovering what the true "lowest common denominator" between every SQL database would be more work than just choosing one and exploiting it.
ii. Second is the assumption that by avoiding special features, the resulting design will actually be portable across RDBMSs without changes. There are so many operational and administrative differences between different databases that the SQL commonality isn't enough to facilitate simple transfer of data and application from one RDBMS to another.
iii. The argument for this decision is commonly expressed as, "We decided to purchase Oracle because our study indicates that it is the best product on the market today. However, if another product is better three years from now, we want to be able to move our application to that new platform." Do you see the fundamental flaw in this logic? What led this organization to determine that Oracle was the best product available today? Inevitably it was the extended set of features differentiating it from competitors that influenced this decision and justified the expense of licensing Oracle. Ignoring all of those additional capabilities means the organization pretty much wasted the months of evaluation time as well as the licensing costs.

In any event, if your organization wants to build a system that will run unchanged on either Oracle or MySQL, you should just choose MySQL and save some money rather than lobotomize Oracle so that it will behave like MySQL.

So, logically, the next step must be to translate the generic logical design into something that is specific to the implementation using a particular RDBMS. For this article's purpose, that RDBMS will obviously be Oracle. How do you represent the logical table structure into the most reliable, best performing, most scalable, and highest concurrency database that can be built?

It is during this phase that knowledge of available RDBMS features and experience with the requirements, usage, and limits of each comes into play. Physical database design is primarily an evaluation of trade-offs. For any given logical design, we guarantee that there are at least a hundred possible physical designs to implement it. We'll further guarantee that there will be significant differences in how those alternative physical designs behave, operate, and perform. There is usually not one absolutely correct alternative in that list, but some will fit the total set of

requirements better than the others. Sorting through the alternatives by evaluating the compromises imposed by the conflicting requirements is the heart of the art of database design.

Practical Design

In real systems development projects, there is a final phase in which the physical design is actually tested and various accommodations, minor or otherwise, are made in response to the discoveries made during testing. Even the most experienced database designers will not be able to fully anticipate the effects of the actual programs running with actual data volumes.

If your team did a good job with physical design, then the database that you create and load is likely to meet the design requirements well. However, many of the requirements that are obtained early in the process may get revised as programs are built and users are able to do their first hands-on testing of the application. Sometimes the estimates of how many users, how many rows, how many transactions, and so on turn out to be inaccurate. As this new knowledge is acquired, it is important to revisit the physical design and potentially reconsider some of the alternatives set aside earlier.

In many cases, this ongoing practical design process can be transparent to the application programs. The Oracle DBA can generally change indexing, partitioning, and virtual private database policies; add views and materialized views; and move a LOB (large object, either of character or binary data) to out-of-line storage without having to modify actual SQL. However, learning late in the development cycle that what you thought was a one-to-one relationship is actually one-to-many can have significant ripples throughout the design.

The key point of considering this as an ongoing part of the development cycle is that the earlier you recognize the need for adapting your database design, the easier it will be to fully investigate alternatives, make the necessary changes, and fully test the results. Discovering these needed changes the day after you put the database and application into production will be several orders of magnitude more painful.

2.2.2 Database Design Tools

Using a tool to support the documentation, diagramming, and iterative evolution of data models is strongly advised. Simple databases with only a few entities may not require a modeling tool, but "real" systems with dozens of entities, hundreds of data elements, and large development teams will need the capabilities of such a tool.

Over the past 20 years or so, data modeling using entity-relationship diagrams, as originally proposed by Peter Chen, has become a standardized methodology with excellent support from several vendors' modeling tools. The most common of these tools (although not necessarily the most powerful) is ERwin Data Modeler, now provided by Computer Associates. Oracle Corporation's tool in this space, Oracle Designer, has unfortunately been somewhat lost in the shadow of its newer sister product, JDeveloper. Both are bundled into the Oracle Developer Suite along with Oracle's other development tools. If your shop acquires JDeveloper for building, debugging, and deploying Java and PL/SQL modules, be aware that you have also already acquired

an excellent data modeling tool (along with Oracle Warehouse Builder for designing data warehouses and their requisite data extraction and transformation processes).

Most of these tools do a reasonable job of aiding the team during logical design, since, by definition, this is a generic process that isn't tied to a specific database management system's particular capabilities and extensions. It is during the later stages of physical (and practical) design that it is important to have a tool that is capable of fully exploiting all of the capabilities of the RDBMS. In the case of designing a database specifically for Oracle, you'll want a tool that can let you define and build partitioned tables, index-organized tables, sequence number generators, global temporary tables, bitmap indexes, materialized views, and so on.

With a tool that isn't aware of the full capabilities of the specific RDBMS, the development team must manually modify and maintain the database definitions generated by the tool. In the long run, this manual effort will present a lot of additional work and risk.

2.2.2.1 Object-Oriented Design Tools

Developer frameworks, such as J2EE Entity Beans, can automatically generate a database object (and the necessary access routines) that will meet the narrow needs of the particular bean to persist its state. It is dangerous to depend upon such default database objects, however. In many cases the generated object will not properly exploit the underlying database's features for performance and scalability. Further, the administration of a database with such unmodeled objects will be unnecessarily difficult and potentially lead to problems in production.

This is not to say that developers shouldn't use the capability when prototyping their program modules. Just consider the generated objects as an input to the actual database design process and not a substitute for it. Several narrowly defined objects may be better modeled as part of a larger picture, tied to more general business requirements, and thereby become more resilient to future application enhancements.

2.2.3 Database Design Techniques

The preceding section on design tools mentioned entity-relationship modeling as a key technique used in the development and discussion of database designs. Unified Modeling Language is another set of popular techniques that can also aid in this process.

2.2.3.1 Entity-Relationship Modeling

Entity-Relationship Diagrams (ERDs) attempt to accomplish two things early in the design of a database. First it documents the entities of interest to the enterprise. Entities may represent things (such as a customer, an employee, or a product) or an event (like a registration, a shipment, or an order). Entities have identifying characteristics and additional descriptive characteristics. For instance, an employee has an employee number, a name, a home address, a work location, a supervisor, a hire date, a pay rate, and so on. One of the challenges during design will be sorting through all of the available data "thingies" to figure out which are entities, which are identifiers of entity instances, and which are descriptive of particular instances of an entity.

In addition to documenting entities, an ERD documents the relationship between entities. From the list of example entities in the previous paragraph, you might conclude that there is a relationship between a customer and a product. It might be direct (a CUSTOMER owns a PRODUCT) or it might be more complex set of relationships:

i. each CUSTOMER may place one or more ORDERs
ii. each ORDER must include one or more PRODUCTs
iii. each SHIPMENT must include at least one PRODUCT to one and only one CUSTOMER
iv. each ORDER may be fulfilled with one or more SHIPMENTs
v. a SHIPMENT may include parts of one or more ORDERs for one and only CUSTOMER and so on

This list is an example of the relationship rules that might exist for a company. Another company with the same entities may have very different relationships and rules. For instance, an alternative relationship might be that "a SHIPMENT fulfills one and only one ORDER". Sorting through these relationships and the business rules that define them is the essence of designing a database that will meet the requirements of a specific business. Getting one rule wrong during logical design can lead to a database that will not be able to support necessary business processes.

Relationships between entities are indicated by lines connecting them. Figure 2.1 illustrates an ERD with eight entities and nine relationships.

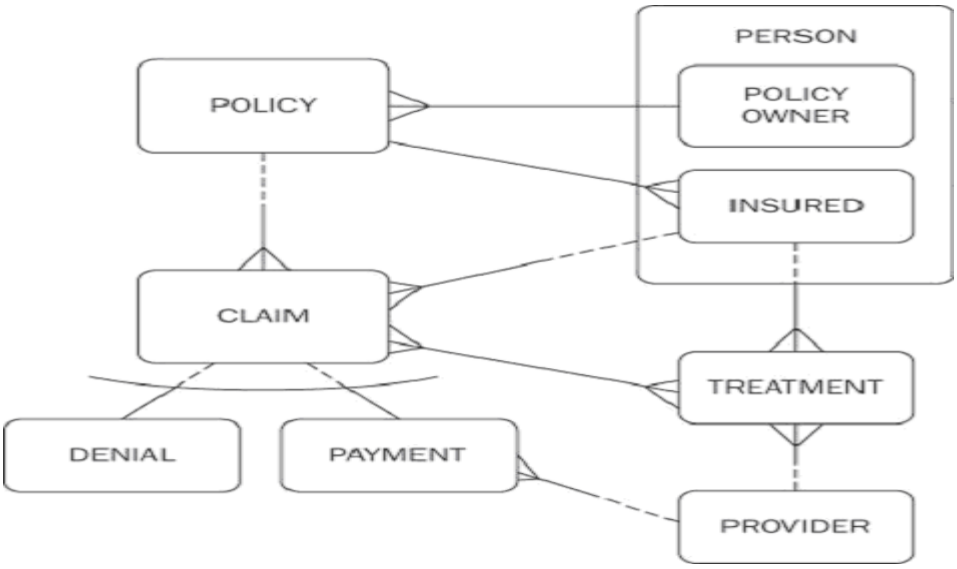

Figure 2.3: A simplified example of an ERD for the medical insurance claims.

While we might have shown INSURED and POLICY_OWNER as completely distinct entities, this model assumes that there are many characteristics of the two that overlap, so they have been shown as two subtypes of the PERSON entity.

The second aspect of an ERD is to document the relationships between entities, as shown by the connecting lines. One line is drawn for each relationship; every entity should be expected to have

at least one relationship to some other entity, but not all entities are necessarily related to every other entity. It is possible for two entities to have more than one relationship.

Further, it should be recognized that every relationship exists in two directions. If entity A has a relationship to entity B, then entity B has a relationship to entity A. A single line defines this bidirectional relationship, but it is useful to define the relationship in both directions. Many methodologies and tools are not strict in this regard, but it helps when reading an ERD to be able to traverse the relationships without having to translate on the fly. In Figure 1, the relationship between POLICY and CLAIM entities can be stated in two ways: (1) A POLICY may have one or more CLAIMs submitted and (2) A CLAIM must be submitted against exactly one POLICY.

The nature of each relationship may be optional or mandatory in one direction or both. In this statement of the bi-directional relationship between the POLICY and CLAIM entities, the term "MUST" indicates that the relationship is mandatory in that direction (a CLAIM cannot exist without an associated POLICY.) A solid line in the ERD indicates that the relationship, in that direction is mandatory.

An optional relationship exists in the other direction — the term "MAY" informs is that a POLICY is allowed to exist without any associated CLAIMs. In the ERD, a broken line indicates that end of the line to indicate that the relationship from that direction is optional.

The degree of each relationship, additionally, may be a one-to-one correspondence between the entities or, more commonly, a one-to-many relationship. Where many occurrences of an entity may exist for a relationship, a "crow's foot" is used to designate the many end of the one-to-many relationship, such as the fact that multiple CLAIMs are allowed for a POLICY. The written description of the relationship uses the "one or more" phrase to designate this.

Where only one occurrence of an entity may participate in the particular relationship, no crow's foot is used and we'll describe the relationship using "one and only one" or "exactly one" terminology.

Many-to-many relationships do occur in the real world, and during this logical design phase, they may appear in our ERD. Figure 1 showed such a relationship between CLAIM and TREATMENT. Many-to-many relationships, however, present difficulties in creating a physical database. Remember that a relational database makes its connection between tables using data values themselves rather than pointers. We will eventually, in physical design, implement a one-to-many relationship by carrying the key value of the "one" side as a data value stored within each row on the "many" side table. Many-to-many relationships can't be stored that way, so data modelers will resolve the many-to-many relationship into two one-to-many relationships and by creating a new connecting entity to help define the relationship. This issue was ignored in the preliminary example, but Figure 2.2 shows that specific portion of our total ERD with this relationship resolved.

Figure 2.4: Resolving a many-to-many relationship using a connecting entity.

2.2.4 Unified Modeling Language

Application designers have been modeling systems for the 50 years that computerized systems have been developed. Flowcharts (some of us actually used the plastic diagram templates to draw them) were an early means of documenting the processing requirements and how data and, more specifically, control passed from module to module. During the 1970s new structured programming methods improved our ability to model the behavior of COBOL programs.

Along with the adoption of object-oriented design and development techniques in the early 1990s, a new set of similar (but distinct) design methodologies arose. OO developers learned and adopted the Booch, Jacobson (OOSE) and/or Rumbaugh (OMT) methodologies. Each of these methodologies had its strengths and its proponents. While they shared many concepts and goals, there were enough differences to make it challenging to pull together a development team that could agree on a common approach to the design of a new system.

The current solution to this dilemma is the Unified Modeling Language (UML), formally adopted as a standard by the Object Management Group in 1997. Today UML is almost universally accepted as the methodology and documentation techniques for nearly all object-oriented application design.

UML can do a far better job of balancing the focus of design on both process and data than an ERD. UML is, however, very process-oriented in its orientation. This is not a criticism; in fact, it is what makes UML the perfect complement to data-oriented design techniques. Many UML practitioners focus on use case diagrams to document interactions between users (Actors) and processes (Tasks) and the relationships (Composition, Association, Triggers) between tasks. Secondary emphasis is given to documenting Scenarios, Sequence Diagrams, and Narrative Descriptions. (Note that at this point the UML process begins to identify entities and attributes, the building blocks of the database design techniques discussed previously.) All of these aspects of UML are primarily focused on process, with only a secondary acknowledgement of data.

Working through the UML facilities, we finally get to UML class diagrams. We recognize that they closely mimic the diagramming techniques of ERD modeling. It is, however, tempting for many designers to consider their class and object modeling from a limited perspective of the particular use case on which they are focused. Nothing in UML prevents or restricts data definition from being performed from a more global perspective, and in fact, it has to be. Design of database objects requires all of the rigor and emphasis that is given to any particular user interface problem — in fact, more. The effects of an inappropriate database design will have ramifications throughout the entire system, and correction of any data design deficiencies are likely to force major changes within many program modules as well.

The danger of UML modeling is not a weakness of the UML techniques themselves. It is an unfortunate tendency within development teams to focus their attention on the process design techniques and ignore class diagrams.

Revision questions

1. What is the importance of data modelling?
2. What are the types of attributes that may be identified during data modelling?
3. Differentiate between an entity type and entity set
4. What is the significance of an ERD in database design?