Database Systems
An application-oriented
Approach
2nd Edition
Michael Kifer, Arthur
Bernstein, Philip M.
Lewis

2

The Big Picture

2.1 Case Study: A Student Registration System

CASE STUDY

Your university is interested in implementing a student registration system so that students can register for courses from their home PCs. You have been asked to build a prototype of that system as a project in this course. The registrar has prepared the following preliminary **Statement of Objectives** for the system.

> The objectives of the Student Registration System are to allow students and faculty (as appropriate) to
>
> 1. Authenticate themselves as users of the system
> 2. Register and deregister for courses (offered for the next semester)
> 3. Obtain reports on a particular student's status
> 4. Maintain information about students and courses
> 5. Enter final grades for courses that a student has completed

This brief description is typical of what might be supplied as a starting point for a system implementation project, but it is not specific or detailed enough to serve as the basis for the project's design and coding phases. We will be developing the student registration scenario throughout this book and will be using it to illustrate the various concepts in databases and transaction processing.

Our next step is to meet with the registrar, faculty, and students to expand this brief description into a formal Requirements Document for the system. We will discuss the Requirements Document in Chapter 14, which we expect you to read at appropriate times as you proceed through the rest of the book. In this chapter, we will take a closer look at some of the underlying concepts of databases and transaction processing that are needed for that system.

The following sections provide a brief overview of these concepts. Although we will revisit these concepts in a more detailed fashion in subsequent chapters, an overview will help you see the big picture and will set the stage for better understanding of the following chapters.

2.2 Introduction to Relational Databases

A database is at the heart of most transaction processing systems. At every instant of time, the database must contain an accurate description—often the only one—of the real-world enterprise the transaction processing system is modeling. For example, in the Student Registration System the database is the only source of information about which students have registered for each course.

Relations and tuples. We are particularly interested in databases that use the **relational model** [Codd 1970, 1990], in which data is stored in **tables**. The Student Registration System, for example, might include the STUDENT table, shown in Figure 2.1. A table contains a set of **rows**. In the figure, each row contains information about one student. Each **column** of the table describes the student in a particular way. In the example, the columns are `Id`, `Name`, `Address`, and `Status`. Each column has an associated type, called its **domain**, from which the value in a particular row for that column is drawn. For example, the domain for `Id` is integer and the domain for `Name` is string.

This database model is called "relational" because it is based on the mathematical concept of a relation. A **mathematical relation** captures the notion that elements of different sets are related to one another. For example, `John Doe`, an element of the set of all humans, is related to `123 Main St.`, an element of the set of all addresses, and to `111111111`, an element of the set of all Ids. A relation is a set of **tuples**. Following the example of the table STUDENT, we might define a relation called STUDENT containing the tuple ⟨`111111111, John Doe, 123 Main St., Freshman`⟩. The STUDENT relation presumably contains a tuple describing every student.

We can view a relation as a predicate. A **predicate** is a declarative statement that is either true or false depending on the values of its arguments—for example, the predicate "It rained in Detroit on date X" is either true or false depending on the value chosen for the argument X. When we view a relation as a predicate, the arguments of the predicate correspond to the elements of a tuple, and the predicate is defined to be true for arguments $a_1, \ldots, a_n$ exactly when the tuple $(a_1, \ldots, a_n)$ is in the relation. For instance, we might define the predicate STUDENT

Id	Name	Address	Status
111111111	John Doe	123 Main St.	Freshman
666666666	Joseph Public	666 Hollow Rd.	Sophomore
111223344	Mary Smith	1 Lake St.	Freshman
987654321	Bart Simpson	Fox 5 TV	Senior
023456789	Homer Simpson	Fox 5 TV	Senior
123454321	Joe Blow	6 Yard Ct.	Junior

FIGURE 2.1 The table STUDENT. Each row describes a single student.

with arguments `Id`, `Name`, `Address`, and `Status`. Then we can say that the predicate STUDENT ⟨`111111111, John Doe, 123 Main St., Freshman`⟩ is true, because the tuple ⟨`111111111, John Doe, 123 Main St., Freshman`⟩ is in the table STUDENT shown in Figure 2.1.

The correspondence between tables and relations should now be clear: the tuples of a relation correspond to the rows of a table, and the column names of a table are the names of the **attributes** of the relation. Thus, the rows of the STUDENT table can be viewed as enumerating the set of all 4-tuples (tuples with four attributes of the appropriate types) that satisfy the STUDENT relation (i.e., the Id, Name, Address, and Status of a student).

Operations on tables are mathematically defined. In real applications, tables can become quite large—a STUDENT table for our university would contain over 15 thousand rows, and each row would likely contain much more information about each student than is shown here. In addition to the STUDENT table, the complete database for the Student Registration System at our university would contain a number of other tables, each with a large number of rows, containing information about other aspects of student registration. For example, a TRANSCRIPT table might contain a row for each course that every student has ever taken. Hence, the databases for most applications contain a large amount of information and are generally held in mass storage.

In most applications, the database is under the control of a database management system (DBMS), which is supplied by a commercial vendor. When an application wants to perform an operation on the database, it does so by making a request to the DBMS. A typical operation might extract some information from the rows of one or more tables, modify some rows, or add or delete rows. For example, when a new student is admitted to the university, a row is added to the STUDENT table.

In addition to the fact that tables in the database can be modeled by mathematical relations, operations on the tables can also be modeled as mathematical operations on the corresponding relations. Thus, a particular unary operation might take a table, T, as an argument and produce a result table containing a subset of the rows of T. For example, an instructor might want to display the roster of students registered for a course. Such a request might involve scanning the TRANSCRIPT table, locating the rows corresponding to the course, and returning them to the application. A particular binary operation might take two tables as arguments and construct a new table containing the union of the rows of the argument tables. A complex query against a database might be equivalent to an expression involving many such relational operations involving many tables.

Because of this mathematical description, relational operations can be precisely defined and their mathematical properties, such as commutativity and associativity, can be proven. As we shall see, this mathematical description has important practical implications. Commercial DBMSs contain a **query optimizer** module that converts queries into expressions involving relational operations and then uses these mathematical properties to simplify those expressions and thus optimize query execution.

SQL: Basic SELECT statement. An application describes the access that it wants the DBMS to perform on its behalf in a language supported by the DBMS. We are particularly interested in SQL, the most commonly used database language, which provides facilities for accessing a relational database and is supported by almost all commercial DBMSs.

The basic structure of the SQL statements for manipulating data is straightforward and easy to understand. Each statement takes one or more tables as arguments and produces a table as a result. For example, to find the name of the student whose Id is 987654321, we might use the statement

```
SELECT   Name                                          2.1
FROM     STUDENT
WHERE    Id = '987654321'
```

More precisely, this statement asks the DBMS to extract from the table named in the FROM clause—that is, the table STUDENT—all rows satisfying the condition in the WHERE clause—that is, all rows whose Id column has value 987654321—and then from each such row to delete all columns except those named in the SELECT clause—that is, Name. The resulting rows are placed in a result table produced by the statement. In this case, because Ids are unique, at most one row of STUDENT can satisfy the condition, and so the result of the statement is a table with one column and at most one row.

Thus, the FROM clause identifies the table to be used as input, the WHERE clause identifies the rows of that table from which the answer is to be generated, and the SELECT clause identifies the columns of those rows that are to be output in the result table.

The result table generated by this example contains only one column and at most one row. As a somewhat more complex example, the statement

```
SELECT   Id, Name                                      2.2
FROM     STUDENT
WHERE    Status = 'senior'
```

returns a result table (shown in Figure 2.2) containing two columns and multiple rows: the Ids and names of all seniors. If we want to produce a table containing all the columns of STUDENT but describing only seniors, we use the statement

```
SELECT   *
FROM     STUDENT
WHERE    Status = 'senior'
```

Id	Name
987654321	Bart Simpson
023456789	Homer Simpson

FIGURE 2.2 The database table returned by the SQL SELECT statement (2.2).

The asterisk is simply shorthand that allows us to avoid listing the names of all the columns of STUDENT.

In some situations the user is interested not in outputting a result table but in information *about* the result table. An example is the statement

```
SELECT   COUNT(*)
FROM     STUDENT
WHERE    Status = 'senior'
```

which returns the number of rows in the result table (i.e., the number of seniors). COUNT is referred to as an **aggregate** function because it produces a value that is a function of all the rows in the result table. Note that when an aggregate is used, the SELECT statement produces a single value instead of a table.

The WHERE clause is the most interesting component of the SELECT statement; it contains a general condition that is evaluated over each row of the table named in the FROM clause. Column values from the row are substituted into the condition, yielding an expression that has either a true or a false value. If the condition evaluates to true, the row is retained for processing by the SELECT clause and then stored in the result table. Hence, the WHERE clause acts as a filter.

Conditions can be much more complex than we have seen so far: A condition can be a Boolean combination of terms. If we want the result table to contain information describing seniors whose Ids are in a particular range, for example, we might use

```
WHERE  Status = 'senior' AND Id > '888888888'
```

OR and NOT can also be used. Furthermore, a number of predicates are provided in the language for expressing particular relationships. For example, the IN predicate tests set membership.

```
WHERE  Status IN ('freshman', 'sophomore')
```

Additional aggregates and predicates and the full complexity of the WHERE clause are discussed in Chapter 5.

Multi-table SELECT statements. The result table can contain information extracted from several base tables. Thus, if we have a table TRANSCRIPT with columns StudId, CrsCode, Semester, and Grade, the statement

```
SELECT   Name, CrsCode, Grade
FROM     STUDENT, TRANSCRIPT
WHERE    StudId = Id AND Status = 'senior'
```

can be used to form a result table in which each row contains the name of a senior, a particular course she took, and the grade she received.

The first thing to note is that the attribute values in the result table come from different base tables: Name comes from STUDENT; CrsCode and Grade come from TRANSCRIPT. As in the previous examples, the FROM clause produces a table whose rows are input to the WHERE clause. In this case the table is the Cartesian product of the tables listed in the FROM clause: a row of this table is the concatenation of a row of STUDENT and a row of TRANSCRIPT. Many of these rows make no sense. For example, Bart Simpson's row in STUDENT is not related to a row in TRANSCRIPT describing a course that Bart did not take. The first conjunct of the WHERE clause ensures that the rows of TRANSCRIPT for a particular student are associated with the appropriate row of STUDENT by matching the Id values of the rows of the two tables. For example, if TRANSCRIPT has a row ⟨987654321, CS305, F1995, C⟩, it will match only Bart Simpson's row in STUDENT, producing the row ⟨Bart Simpson, CS305, C⟩ in the result table.

Query optimization. One very important feature of SQL is that the programmer does not have to specify the algorithm the DBMS should use to satisfy a particular query. For example, tables are frequently defined to include auxiliary data structures, called **indices**, which make it possible to locate particular rows without using lengthy searches through the entire table. Thus, an index on the Id column of the STUDENT table might contain a list of pairs ⟨*Id, pointer*⟩ where the pointer points to the row of the table containing the corresponding Id. If such an index were present, the DBMS would automatically use it to find the row that satisfies the query (2.1). If the table also had an index on the column Status, the DBMS would use that index to find the rows that satisfy the query (2.2). If this second index did not exist, the DBMS would automatically use some other method to satisfy (2.2)—for example, it might look at every row in the table in order to locate all rows having the value senior in the Status column. The programmer does not specify what method to use—just the condition the desired result table must satisfy.

In addition to selecting appropriate indices to use, the query optimizer uses the properties of the relational operations to further improve the efficiency with which a query can be processed—again, without any intervention by the programmer. Nevertheless, programmers should have some understanding of the strategies the DBMS uses to satisfy queries so they can design the database tables, indices, and

SQL statements in such a way that they will be executed in an efficient manner consistent with the requirements of the application.

Changing the contents of tables. The following examples illustrate the SQL statements for modifying the contents of a table. The statement

```
UPDATE   STUDENT
SET      Status = 'sophomore'
WHERE    Id = '111111111'
```

updates the STUDENT table to make John Doe a sophomore. The statement

```
INSERT
INTO     STUDENT (Id, Name, Address, Status)
VALUES   ('999999999', 'Winston Churchill', '10 Downing St',
         'senior')
```

inserts a new row for Winston Churchill in the STUDENT table. The statement

```
DELETE
FROM     STUDENT
WHERE    Id = '111111111'
```

deletes the row for John Doe from the STUDENT table. Again, the details of how these operations are to be performed need not be specified by the programmer.

Creating tables and specifying constraints. Before you can store data in a table, the table structure must be created. For instance, the STUDENT table could have been created with the SQL statement

```
CREATE TABLE  STUDENT(
Id            INTEGER,
Name          CHAR(20),
Address       CHAR(50),                                    2.3
Status        CHAR(10),
PRIMARY KEY(Id) )
```

where we have declared the name of each column and the domain (type) of the data that can be stored in that column. We have also declared the Id column to be a **primary key** to the table, which means that each row of the table must have a unique value in that column and the DBMS will (most probably) automatically construct an index on that column. The DBMS will enforce this uniqueness constraint by not allowing any INSERT or UPDATE statement to produce a row with a value in the Id column that duplicates a value of Id in another row. This requirement is an

example of an **integrity constraint** (sometimes called a **consistency constraint**)—an application-based restriction on the values that can appear as entries in the database. We discuss integrity constraints in more detail in the next section.

We have given simple examples of each statement type to highlight the conceptual simplicity of the basic ideas underlying SQL, but be aware that the complete language has many subtleties. Each statement type has a large number of options that allow very complex queries and updates. For this reason, mastery of SQL requires significant effort. We continue our discussion of relational databases and SQL in Chapter 3.

2.3 What Makes a Program a Transaction—The ACID Properties

In many applications, a database is used to model the state of some real-world enterprise. In such applications, a transaction is a program that interacts with that database so as to maintain the correspondence between the state of the enterprise and the state of the database. In particular, a transaction might update the database to reflect the occurrence of a real-world event that affects the enterprise state. An example is a deposit transaction at a bank. The event is that the customer gives the teller the cash and a deposit slip. The transaction updates the customer's account information in the database to reflect the deposit.

Transactions, however, are not just ordinary programs. Requirements are placed on them, particularly on the way they are executed, that go beyond what is normally expected of regular programs. These requirements are enforced by the DBMS and the TP monitor.

Consistency. A transaction must access and update the database in such a way that it preserves all database integrity constraints. Every real-world enterprise is organized in accordance with certain rules that restrict the possible states of the enterprise. For example, the number of students registered for a course cannot exceed the number of seats in the room assigned to the course. When such a rule exists, the possible states of the database are similarly restricted.

The restrictions are stated as integrity constraints. The integrity constraint corresponding to the above rule asserts that the value of the database item that records the number of course registrants must not exceed the value of the item that records the room size. Thus, when the registration transaction completes, the database must satisfy this integrity constraint (assuming that the constraint was satisfied when the transaction started).

Although we have not yet designed the database for the Student Registration System, we can make some assumptions about the data that will be stored and postulate some additional integrity constraints:

- *IC0.* The database contains the Id of each student. These Ids must be unique.
- *IC1.* The database contains a list of prerequisites for each course and, for each student, a list of completed courses. A student cannot register for a course without having taken all prerequisite courses.

- *IC2.* The database contains the maximum number of students allowed to take each course and the number of students who are currently registered for each course. The number of students registered for each course cannot be greater than the maximum number allowed for that course.
- *IC3.* It might be possible to determine the number of students registered for (or enrolled in) a particular course from the database in two ways: the number is stored as a count in the information describing the course, and it can be calculated from the information describing each student by counting the number of student records that indicate that the student is registered for (or enrolled in) the course. These two determinations must yield the same result.

In addition to maintaining the integrity constraints, each transaction must update the database in such a way that the new database state reflects the state of the real-world enterprise that it models. If John Doe registers for CS305, but the registration transaction records Mary Smith as the new student in the class, the integrity constraints will be satisfied but the new state will be incorrect. Hence, consistency has two dimensions.

> ***Consistency.*** The transaction designer can assume that when execution of the transaction is initiated, the database is in a state in which all integrity constraints are satisfied and, in addition, the database correctly models the current state of the enterprise. The designer has the responsibility of ensuring that when execution has completed, the database is once again in a state in which all integrity constraints are satisfied and, in addition, that the new state reflects the transformation described in the transaction's specification (in other words, that the database still correctly models the state of the enterprise).

SQL provides some support for the transaction designer in maintaining consistency. When the database is being designed, the database designer can specify certain types of integrity constraints and include them within the statements that declare the format of the various tables in the database. The primary key constraint of the SQL statement (2.3) is an example of this. Later, as each transaction is executed, the DBMS automatically checks that each specified constraint is not violated and prevents completion of any transaction that would cause a constraint violation.

Atomicity. In addition to the transaction designer's responsibility for consistency, the TP monitor must provide certain guarantees concerning the manner in which transactions are executed. One such condition is atomicity.

> ***Atomicity.*** The system must ensure that the transaction either runs to completion or, if it does not complete, has no effect at all (as if it had never been started).

In the Student Registration System, either a student has registered for a course or he has not registered for a course. Partial registration makes no sense and might leave the database in an inconsistent state. For example, as indicated by constraint IC3, two items of information in the database must be updated when a student registers.

If a registration transaction were to have a partial execution in which one update completed but the system crashed before the second update could be executed, the resulting database would be inconsistent.

When a transaction has successfully completed, we say that it has **committed**. If the transaction does not successfully complete, we say that it has **aborted** and the TP monitor has the responsibility of ensuring that whatever partial changes the transaction has made to the database are undone, or **rolled back**. **Atomic execution** means that every transaction either commits or aborts.

Notice that ordinary programs do not necessarily have the property of atomicity. For example, if the system were to crash while a program that was updating a file was executing, the file could be left in a partially updated state when the system recovered.

Durability. A second requirement of the transaction processing system is that it does not lose information.

> ***Durability.*** The system must ensure that once the transaction commits, its effects remain in the database even if the computer, or the medium on which the database is stored, subsequently crashes.

For example, if you successfully register for a course, you expect the system to remember that you are registered even if it later crashes. Notice that ordinary programs do not necessarily have the property of durability either. For example, if a media failure occurs after a program that has updated a file has completed, the file might be restored to a state that does not include the update.

Isolation. In discussing consistency, we concentrated on the effect of a single transaction. We next examine the effect of executing a set of transactions. We say that a set of transactions is executed sequentially, or **serially**, if one transaction in the set is executed to completion before another is started. The good news about serial execution is that if all transactions are consistent and the database is initially in a consistent state, serial execution maintains consistency. When the first transaction in the set starts, the database is in a consistent state and, since the transaction is consistent, the database will be consistent when the transaction completes. Because the database is consistent when the second transaction starts, it too will perform correctly and the argument will repeat.

Serial execution is adequate for applications that have modest performance requirements. However, many applications have strict requirements on response time and throughput, and often the only way to meet the requirements is to process transactions concurrently. Modern computing systems are capable of servicing more than one transaction simultaneously, and we refer to this mode of execution as **concurrent**. Concurrent execution is appropriate in a transaction processing system serving many users. In this case, there will be many active, partially completed transactions at any given time.

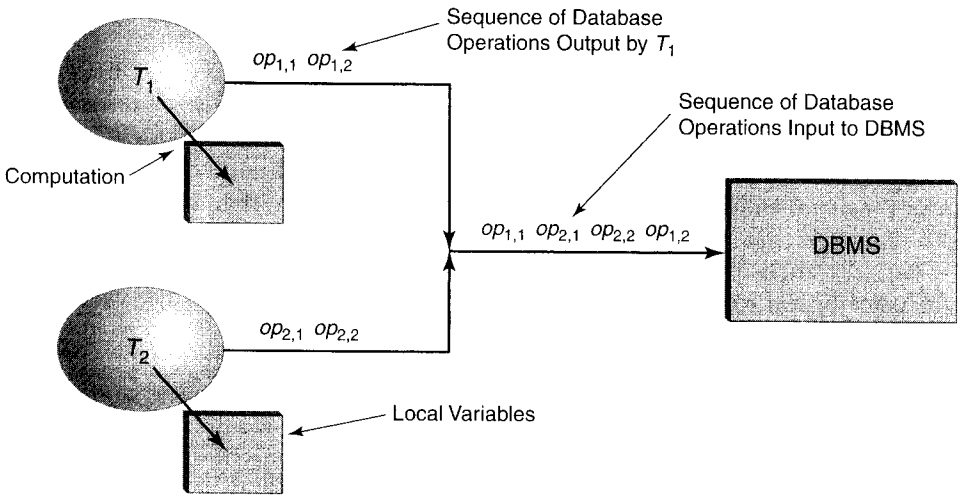

FIGURE 2.3 The database operations output by two transactions in a concurrent schedule might be interleaved in time. (Note that the figure should be interpreted as meaning that $op_{1,1}$ arrives first at the DBMS, followed by $op_{2,1}$, etc.)

In concurrent execution, the database operations of different transactions are effectively interleaved in time, a situation shown in Figure 2.3. Transaction T_1 alternately computes using its local variables and sends requests to the database system to transfer data between the database and its local variables. The requests are made in the sequence $op_{1,1}$, $op_{1,2}$. We refer to that sequence as a **transaction schedule**. T_2 performs its computation in a similar way. Because the execution of the two transactions is not synchronized, the sequence of operations arriving at the database, called a **schedule**, is an arbitrary merge of the two transaction schedules. The schedule in the figure is $op_{1,1}$, $op_{2,1}$, $op_{2,2}$, $op_{1,2}$.

When transactions are executed concurrently, the consistency of each transaction is not sufficient to guarantee that the database that exists after both have completed correctly reflects the state of the enterprise. For example, suppose that T_1 and T_2 are two instances of the registration transaction invoked by two students who want to register for the same course. A possible schedule of these transactions is shown in Figure 2.4, where time progresses from left to right and the notation $r(cur\_reg : n)$ means that a transaction has read the database object *cur_reg*, which

FIGURE 2.4 A schedule in which two registration transactions are not isolated from each other.

T_1 : $r(cur\_reg: 29)$			$w(cur\_reg: 30)$
T_2 :		$r(cur\_reg: 29)$		$w(cur\_reg: 30)$

records the number of current registrants, and the value n has been returned. A similar notation is used for $w(cur\_reg : n)$. The figure shows only the accesses[1] to *cur_reg*.

Assume that the maximum number of students allowed to register is 30 and the current number is 29. In its first step, each of the two transactions will read this value and store it in its local variable, and both will decide that there is room in the course. In its second step, each will increment its private copy of the number of current registrants; hence, both will calculate the value 30. In their write operations, both will write that same value, 30, into *cur_reg*.

Both transactions complete successfully, but the number of current registrants is incorrectly recorded as 30 when it is actually 31 (even though the maximum allowable number is 30). This is an example of what is often referred to as a **lost update** because one of the increments has been lost. The resulting database does not reflect the real-world state, and integrity constraint IC2 has been violated. By contrast, if the transactions had executed sequentially, T_1 would have completed before T_2 was allowed to start. Hence, T_2 would find the course full and would not register the student.

As this example demonstrates, we must specify some restriction on concurrent execution that is guaranteed to maintain the consistency of the database and the correspondence between the enterprise state and the database state. One such restriction that is obviously sufficient follows.

> ***Isolation.*** Even though transactions are executed concurrently, the overall effect of the schedule must be the same as if the transactions had executed serially in some order.

It should be evident that if the transactions are consistent and if the overall effect of a concurrent schedule is the same as that of some serial schedule, the concurrent schedule will maintain consistency. Concurrent schedules that satisfy this condition are called **serializable.**

As was the case with atomicity and durability, ordinary programs do not necessarily have the property of isolation. For example, if programs that update a common set of files are executed concurrently, updates might be interleaved and produce an outcome that is quite different from that obtained if they had been executed in any serial order. That result might be totally unacceptable.

ACID properties. The features that distinguish transactions from ordinary programs are frequently referred to by the acronym **ACID** [Haerder and Reuter 1983]:

- ***Atomic.*** Each transaction is executed completely or not at all.
- ***Consistent.*** Each transaction maintains database consistency.
- ***Isolated.*** The concurrent execution of a set of transactions has the same effect as some serial execution of that set.

[1] In a relational database, r and w represent SELECT and UPDATE statements.

- ***Durable.*** The effects of committed transactions are permanently recorded in the database.

When a transaction processing system supports the ACID properties, the database maintains a consistent and up-to-date model of the real world and the transactions supply responses to users that are always correct and up to date.

BIBLIOGRAPHIC NOTES

The relational model for databases was introduced in [Codd 1970, 1990]. The SQL language is described by the various SQL standards, such as [SQL 1992]. The term "ACID" was coined by [Haerder and Reuter 1983], but the individual components of ACID were introduced in earlier papers—for example, [Gray et al. 1976] and [Eswaran et al. 1976].

EXERCISES

2.1 Given the relation MARRIED that consists of tuples of the form $\langle a, b \rangle$, where a is the husband and b is the wife, the relation BROTHER that has tuples of the form $\langle c, d \rangle$, where c is the brother of d, and the relation SIBLING, which has tuples of the form $\langle e, f \rangle$, where e and f are siblings, describe how you would define the relation BROTHER-IN-LAW, where tuples have the form $\langle x, y \rangle$ with x being the brother-in-law of y.

2.2 Design the following two tables (in addition to that in Figure 2.1) that might be used in the Student Registration System. Note that the same student Id might appear in many rows of each of these tables.

a. A table implementing the relation COURSESREGISTEREDFOR, relating a student's Id and the identifying numbers of the courses for which she is registered
b. A table implementing the relation COURSESTAKEN, relating a student's Id, the identifying numbers of the courses he has taken, and the grade received in each course

Specify the predicate corresponding to each of these tables.

2.3 Write an SQL statement that

a. Returns the Ids of all seniors in the table STUDENT
b. Deletes all seniors from STUDENT
c. Promotes all juniors in the table STUDENT to seniors

2.4 Write an SQL statement that creates the TRANSCRIPT table.

2.5 Using the TRANSCRIPT table, write an SQL statement that

a. Deregisters the student with `Id = 123456789` from the course `CS305` for the fall of 2001
b. Changes to an A the grade assigned to the student with `Id = 123456789` for the course `CS305` taken in the fall of 2000
c. Returns the Id of all students who took `CS305` in the fall of 2000

2.6 Write an SQL statement that returns the names (not the Ids) of all students who received an A in CS305 in the fall of 2000.

2.7 State whether or not each of the following statements could be an integrity constraint of a checking account database for a banking application. Give reasons for your answers.

a. The value stored in the `balance` column of an account is greater than or equal to $0.
b. The value stored in the `balance` column of an account is greater than it was last week at this time.
c. The value stored in the `balance` column of an account is $128.32.
d. The value stored in the `balance` column of an account is a decimal number with two digits following the decimal point.
e. The `social_security_number` column of an account is defined and contains a nine-digit number.
f. The value stored in the `check_credit_in_use` column of an account is less than or equal to the value stored in the `total_approved_check_credit` column. (These columns have their obvious meanings.)

2.8 State five integrity constraints, other than those given in the text, for the database in the Student Registration System.

2.9 Give an example in the Student Registration System where the database satisfies the integrity constraints IC0–IC3 but its state does not reflect the state of the real world.

2.10 State five (possible) integrity constraints for the database in an airline reservation system.

2.11 A reservation transaction in an airline reservation system makes a reservation on a flight, reserves a seat on the plane, issues a ticket, and debits the appropriate credit card account. Assume that one of the integrity constraints of the reservation database is that the number of reservations on each flight does not exceed the number of seats on the plane. (Of course, many airlines purposely over-book and so do not use this integrity constraint.) Explain how transactions running on this system might violate

a. Atomicity
b. Consistency
c. Isolation
d. Durability

2.12 Describe informally in what ways the following events differ from or are similar to transactions with respect to atomicity and durability.

a. A telephone call from a pay phone (Consider line busy, no answer, and wrong number situations. When does this transaction "commit?")
b. A wedding ceremony (Suppose that the groom refuses to say "I do." When does this transaction "commit?")
c. The purchase of a house (Suppose that, after a purchase agreement is signed, the buyer is unable to obtain a mortgage. Suppose that the buyer backs out during the closing. Suppose that two years later the buyer does not make the mortgage payments and the bank forecloses.)
d. A baseball game (Suppose that it rains.)

2.13 Assume that, in addition to storing the grade a student has received in every course he has completed, the system stores the student's cumulative GPA. Describe an integrity constraint that relates this information. Describe how the constraint would be violated if the transaction that records a new grade were not atomic.

2.14 Explain how a lost update could occur if, under the circumstances of the previous problem, two transactions that were recording grades for a particular student (in different courses) were run concurrently.

14

Requirements and Specifications

The implementation of the Student Registration System is proceeding on schedule. A team consisting of faculty, students, and representatives of the registrar has met several times with an analyst and has refined the informal Statement of Objectives given in Section 2.1 into a formal Requirements Document, which we reproduce in Section 14.2. Now it is time to complete the remaining parts of the project. We have already jumped ahead and implemented the database design part of the project in Chapters 4 and 6. In this chapter and in Chapter 15 we review the entire software engineering process in more detail. Our main interest is in the software engineering issues involved in the design and implementation of transactions and databases.

14.1 Software Engineering Methodology

The implementation of a transaction processing system is a significant engineering endeavor. The project must complete on time and on budget, and, when operational, the system must meet its requirements and operate efficiently and reliably. The documentation and coding for the project must be such that the system can be maintained and enhanced over its lifetime. Most important, it must meet the needs of its users.

Many years' experience with both successful and unsuccessful software projects has given rise to a number of procedures and methodologies generally agreed to be "good engineering practice." Many books and entire courses are devoted to software engineering. Here we sketch one approach and apply it to the Student Registration System.

In particular we talk about what software engineers call the **Waterfall model**, in which the project is divided into separate phases: requirements, specification, design, code, and test. And after the system is delivered, there is a final phase: maintenance. Similar phases exist in the development of most large engineering systems—for example a commercial airliner. In this chapter we talk about the requirements and specification phases, and in Chapter 15 we discuss the remaining phases.

Requirements Document. Projects usually begin with an informal Statement of Objectives as given at the beginning of Section 2.1. The next step is for the customers and users of the system, perhaps with the help of a system analyst, to expand these objectives into a formal **Requirements Document** for the system as given in Section 14.2. The Requirements Document describes in some detail what the system is supposed to do, not how it will do it. In many contexts, the Requirements Document is a Request for Proposals (RFP) to the implementors, describing what the customer wants them to build.

Specification Document. The implementation group analyzes the Requirements Document in detail and produces a **Specification Document**, which is an expanded version of the Requirements Document that describes in still more detail what the system will do. In many contexts, the Specification Document is a contract proposal, describing exactly what the implementation group intends to build. The description is so precise that the User Manual and the Specification Document can be written and published at the same time. The following examples illustrate the different levels of detail in the requirements and specification documents.

- In the Requirements Document, the set of user interactions with the system is listed, together with what each interaction is intended to do. In the Specification Document, the forms associated with each interaction are specified, together with exactly what happens when each button is pressed and each menu item is accessed.
- The Requirements Document lists the information that must be contained in the system. The Specification Document includes the domains of all items of information.

14.1.1 UML Use Cases

The Requirements Document specifies what the system is supposed to do from the user's point of view. Specifically, it specifies the user interactions with the system.

A common way to describe user interactions with the system is as a set of *use cases*. A software engineering text might define a **use case** as a sequence of actions that are performed to produce an observable result of benefit to one or more users (called **actors**). For example, in the Requirements Document, we have a use case called *Registration* in which a student registers for a course. Analysts often develop use cases by asking potential users of the system, "How do you accomplish such and such?". Thus we might ask a student, "How do you register for a course?" and then ask the registrar, "What are some situations in which the registration should not succeed?". Their responses might be the basis for developing the Registration use case, in which a student is the actor. Such a use case might be described as follows:

Registration.
Purpose. Register a student in a course to be taught next semester.
Actor. A student.
Input. A course number.

Result. The student is registered for the course, and an appropriate message is displayed.
Exception. The registration shall not be successful for any of the following reasons, which shall be contained in the output.

A. There exists a prerequisite course that the student is not currently enrolled in or has not completed with a grade of at least C.

B. And so on. (The complete list of exceptions is in the Requirements Document.)

Different software engineering texts use different formats for describing use cases. We use a particular format that seems appropriate for this application. Other formats might include *Preconditions*—what must be true before the use case starts; for example a precondition for the Registration use case might be that the Authentication use case for that actor has been successful; or *Actions*—for example, the student first does this, then that happens, then the student does this, etc.

Note that we describe user interactions using use cases rather than transactions because at this stage we do not yet know how many transactions will be required to implement each use case—that is part of the design.

Use cases are part of the *Unified Modeling Language* (or UML). The UML is a graphical language for modeling the static and dynamic behavior of a system. It provides a standard set of diagrams, each of which models a different aspect of the system's behavior. Because these diagrams are graphical, they are particularly appropriate for communicating information between the customer and the implementation group and between different members of the implementation group. Also, because the UML has become a widely adopted standard, UML diagrams can be used to communicate with other people not directly involved in the project, perhaps consultants invited in for a project review.

UML diagrams are one of the sources on which the requirements, specification, and design documents are based. In particular they are used to capture and display certain key aspects of the system behavior needed for these documents. Use cases and, as we shall see, use case diagrams are a part of the UML that deals with formulating requirements. In Section 14.4, we discuss the use of UML sequence diagrams for formulating specifications. In Chapter 4 we discuss the use of UML class diagrams for database design, and in Section 15.1.2 we discuss the use of UML state diagrams for describing the dynamic behavior of objects as part of the design process.

Although use cases are not inherently graphic in nature, the UML provides a graphic way to display the use cases in an application: the **use case diagram**. Figure 14.1 shows a use case diagram for all the use cases in the Student Registration System (as described in Section 14.2). Each actor in the use case is represented as a labeled stick figure, and each use case is represented as a labeled oval. Arrows connect each actor with the use cases in which the actor participates. Such a use case diagram might be developed while interviewing various potential users of the system and then discussed with these users to ensure that the set of use cases is complete and that the final system will satisfy their needs and meet their goals

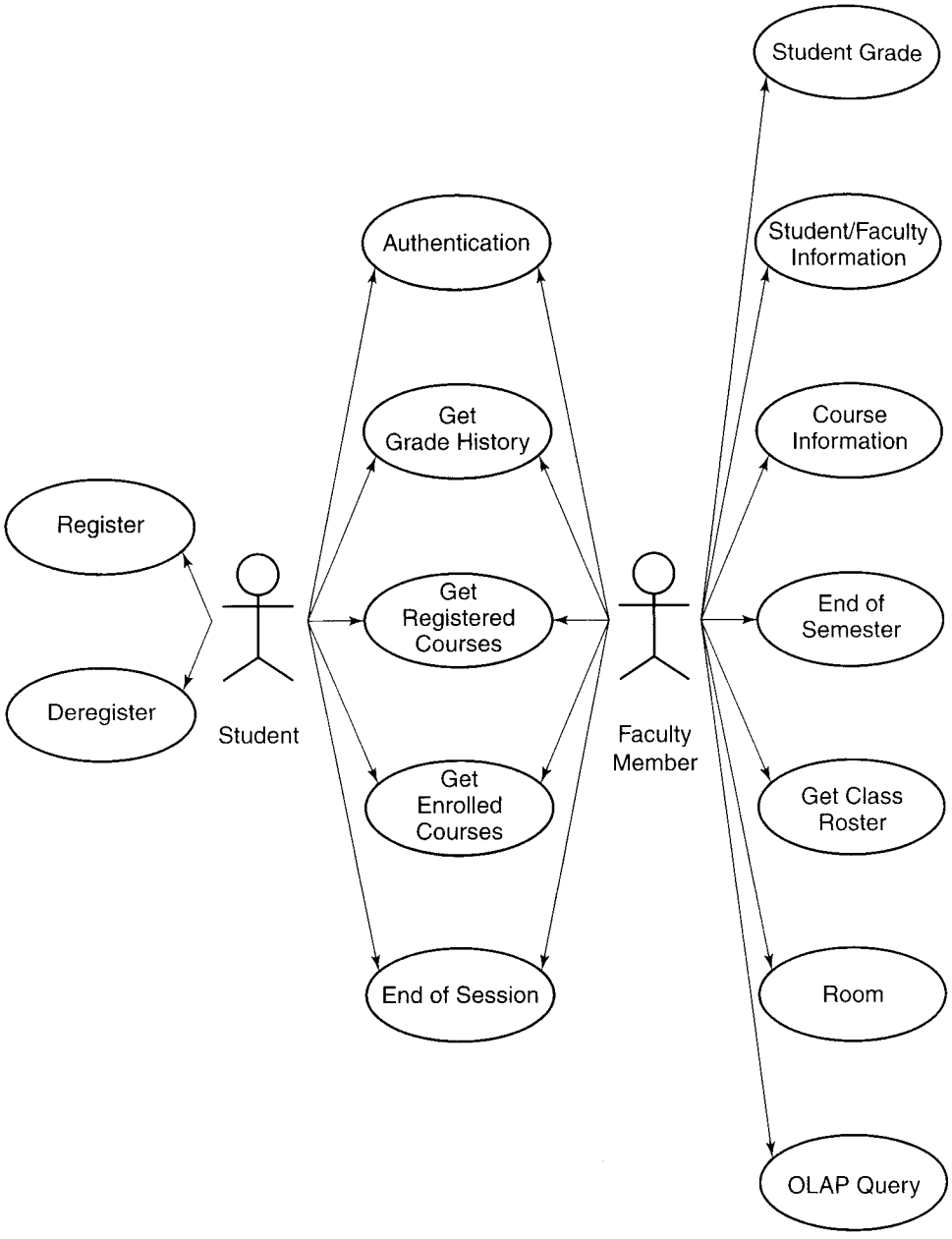

FIGURE 14.1 A use case diagram for the Student Registration System.

for the system. Use case diagrams provide a visually clear model for displaying the requirements of an application and might be included as part of the Requirements Document.

14.2 The Requirements Document for the Student Registration System

CASE STUDY

I. Introduction

The objectives of the Student Registration System are to allow students and faculty (as appropriate) to

A. Authenticate themselves as users of the system

B. Register and deregister for courses (offered for the next semester)

C. Obtain reports on a particular student's status

D. Maintain information about students and courses

E. Enter final grades for courses that a student has completed

In this document, the term "enrolled" refers to courses a student is currently taking and the terms "registered" and "deregistered" refer to courses to be taken or dropped by the student in the following semester.

II. Related Documents

A. *Statement of Objectives* of the Student Registration System (including date and version number)

B. This university's *Undergraduate Bulletin* (including date)

III. Information to Be Contained in the System

The information to be stored in the system includes four major categories of data: personal information about students and faculty members, academic records of students, teaching records of faculty members, and information about courses and course offerings. Information about classrooms and other auxiliary data is also stored in the system.

A. **Personal records.** The system shall contain a name, an Id number, and a password for each student and faculty member allowed to use the system.[1] The password and the Id authenticates users. Id numbers are unique. It is assumed

[1] Note that the requirements are numbered so that they can be referred to in later documents, such as the Test Plan, which must test that the system meets every one of its requirements. Also, requirements that are stated using words such as "shall" and "must" are mandatory. Words such as "should" and "can" do not connote a mandatory requirement and should be avoided unless the requirement is optional. For example, in one of the earliest recorded Requirements Documents (even then the requirements were numbered), the commandment is "Thou shalt not kill," not "Thou should not kill."

that at least one faculty member has been initialized as a valid user at startup time.

B. **Academic records**. The system shall contain the academic record of each student.
   1. Each course the student has completed, the semester the student took the course, and the grade the student received (all grades are in the set {A, B, C, D, F, I})
   2. Each course for which the student is enrolled this semester
   3. Each course for which the student has registered for next semester

C. **Course information**. The system shall contain information about the courses offered, and for each course the system shall contain
   1. The course name, the course number (must be unique), the department offering the course, the textbook, and the credit hours
   2. Whether the course is offered in spring, fall, or both
   3. The prerequisite courses (there can be an arbitrary number of prerequisites for each course)
   4. The maximum allowed enrollment, the number of students who are enrolled (unspecified if the course is not offered this semester), and the number of students who have registered (unspecified if the course is not offered next semester)
   5. If the course is offered this semester, the days and times at which it is offered; if the course is offered next semester, the days and times at which it will be offered. The possible values shall be selected from a fixed list of weekly slots (e.g., MWF10).
   6. The Id of the instructor teaching the course this semester and next semester (the Id is unspecified if the course is not offered in the specified semester; it must be specified before the start of the semester in which the course is offered)
   7. The classroom assignment of the course for this semester and next semester (the classroom assignment is unspecified if the course is not offered in the specified semester; it must be specified before the start of the semester in which the course is offered)

   All course information shall be consistent with the *Undergraduate Bulletin*.

D. **Teaching information**. The system shall contain a record of all courses that have been taught, including the semester in which they were taught and the Id of the instructor.

E. **Classroom information**. The system shall contain a list of classroom identifiers and the corresponding number of seats. A classroom identifier is a unique three-digit integer.

F. **Auxiliary information**. The system shall contain the identity of the current semester (e.g., F2004).

CASE STUDY

IV. Integrity Constraints

The database shall satisfy the following integrity constraints.

A. Id numbers are unique.

B. If in item III.B.2 (or III.B.3) a student is listed as enrolled (or registered) for a course, that course must be indicated in item III.C.2 as offered this semester (or next semester).

C. In item III.C.4, the number of students registered or enrolled in a course cannot be larger than the maximum enrollment.

D. The count of students enrolled (or registered) in a course in item III.B.2 (or III.B.3) must equal the current enrollment (or registration) indicated in item III.C.4.

E. An instructor cannot be assigned to two courses taught at the same time in the same semester.

F. Two courses cannot be taught in the same room at the same time in a given semester.

G. If a student is enrolled in a course, the corresponding record must indicate that the student has completed all prerequisite courses with a grade of at least C.

H. A student cannot be registered (or enrolled) in two courses taught at the same hour.

I. A student cannot be registered for more than 20 credits in a given semester.

J. The room assigned to a course must have at least as many seats as the maximum allowed enrollment for the course.

K. Once a letter grade of A, B, C, D, or F has been assigned for a course, that grade cannot later be changed to an I.[2]

V. Use Cases

Use cases are performed during *sessions*. A session starts when a user executes an Authentication use case and ends when a user executes an End of Session use case. During a session, a user can execute one or more use cases. The use case diagram for these use cases is shown in Figure 14.1 (which is assumed to be part of the Requirements Document).

A. **Authentication.**
Purpose. Identify the actor and determine whether she is a student or faculty member. Subsequent use cases in the same session depend on this distinction.
Actor. A student or a faculty member.
Input. The actor's Id number and password.

[2] This is an example of a *dynamic* integrity constraint, which limits the allowable changes to the state of a database, in contrast to a *static* integrity constraint, which limits the allowable states of the database. We discuss dynamic integrity constraints in Section 3.2.2.

Result. The actor is authenticated and can perform other use cases she is authorized to perform.
Exception. If the actor enters an incorrect Id or password, authentication does not occur and the actor is given another chance to enter an Id and password.

B. **Registration**.
Purpose. Register a student in a course to be taught next semester.
Actor. A student.
Input. A course number.
Result. The student is registered for the course, and an appropriate message is displayed.
Exception. The registration shall not be successful for any of the following reasons, which shall be contained in the output:

1. There exists a prerequisite course that the student is not currently enrolled in or has not completed with a grade of at least C.
2. The number of students registered for the course would exceed the allowed maximum.
3. The initiator of the use case is not a student.
4. The student has registered for another course scheduled at the same time.
5. The student is enrolled in the course or has taken the course and has received a grade of C or better.
6. The course is not offered next semester.
7. The student is already registered for the course.
8. The student would be taking more than 20 credits if the registration were to succeed.

C. **Deregistration**.
Purpose. Deregister the student from a course to be taught next semester for which that student previously registered.
Actor. A student.
Input. A course number.
Result. The student is no longer registered for the course, and an appropriate message is displayed.
Exception. If the student is not registered for the course, the deregistration shall be unsuccessful.

D. **Get Grade History**.
Purpose. Produce a report describing the grade history of a student for each semester in which he has completed courses.
Actor. A student or a faculty member.
Input. A student Id number. If a student is executing the use case, the number need not be entered because the student can request only his or her own report, and the Id of the invoker has been determined as part of authentication.
Result. The report shall include

1. Current semester
2. Student name and Id number

CASE STUDY

3. List of courses completed with grade and instructor grouped by semester
4. Semester GPA and total number of credits for each semester in which the student has completed courses
5. Cumulative GPA and total number of credits of all courses completed so far

Exception. If a faculty member is executing the use case and an invalid student Id is input, no report shall be produced.

E. **Get Registered Courses.**
Purpose. Produce a report listing the courses for which a particular student has registered for the next semester.
Actor. A student or a faculty member.
Input. A student Id number. If a student is executing the use case, the number need not be entered because the student can request only his or her own report, and the Id of the invoker has been determined as part of authentication.
Result. The report shall include

1. Student's name and Id number
2. Course number and credit hours
3. Time schedule for every course
4. Classroom assignment (if available)
5. Instructor (if available)

Exception. If a faculty member is executing the use case and an invalid student Id is input, no report shall be produced.

F. **Get Enrolled Courses.**
Purpose. Produce a report listing the courses in which a particular student is enrolled this semester.
Actor. A student or a faculty member.
Input. A student Id number. If a student is executing the use case, the number need not be entered because the student can request only his or her own report, and the Id of the invoker has been determined as part of authentication.
Result. The report shall include

1. Student's name and Id number
2. Course number and credit hours
3. Time schedule for every course
4. Classroom assignment
5. Instructor

Exception. If a faculty member is executing the use case and an invalid student Id is input, no report shall be produced.

G. **Student Grade.**
Purpose. Assign or change a grade for a course a student has completed.
Actor. The faculty member who taught the course.
Input. A student Id number, a course number, a semester, and a grade.

Result.

1. The student shall no longer be shown as enrolled in that course, but shall be shown as having completed that course.
2. If the course is a prerequisite for some course in the following semester for which the student is currently registered and if the grade is less than C, the student shall be deregistered from that course.

Exception. The grade shall not be assigned or changed if

1. The invoker is not the faculty member who taught the course in the semester indicated.
2. The student Id number is invalid.
3. The student is not currently enrolled in the course or did not take the course in a previous semester.
4. The use case would change a grade (A, B, C, D, F) to an I.

H. **Student/Faculty Information.**
Purpose. Add, delete, or edit an entry specified in item III.A.
Actor. a faculty member.[3]
Input. If an entry is to be added, the name, Id number, faculty/student status, and password must be supplied. If an entry is to be deleted or edited, the Id number must be provided as well as any fields to be changed.
Result. The specified information is added, edited, or deleted.

I. **Course Information.**
Purpose. Display or edit the information describing an existing course (item III.C) or enter information describing a new course.
Actor. A student or a faculty member.
Input. A course number.
Result. The requested information is displayed and can be edited. A faculty member can change any characteristic of a course but cannot delete the course. Students shall be allowed only to display (not to enter or edit) information about a course.
Exception. If the edited course information would violate any integrity constraint, no update shall take place.

J. **End of Semester.**
Purpose. Update the database to reflect the end of the semester.
Actor. A faculty member.
Result.

1. The identity of the current semester, as specified in item III.F, shall be advanced.
2. For each student, an I grade shall be assigned for all courses in which that student is currently enrolled and for which no grade has yet been assigned.

[3] In a real system, this information would be controlled by a database administrator using a special set of transactions. In this project, we assume for simplicity that the database has been initialized with at least one faculty member's name, Id, and password.

CASE STUDY

3. Each student shall be indicated as enrolled in those courses for which the database previously indicated that the student was registered.
4. For each course listed in item III.C.4, the number of students enrolled shall be set equal to the number registered, and the number registered shall be set to 0.

Exception. If a course is scheduled to be taught next semester to which an instructor or classroom has not yet been assigned, the semester shall not be updated, no database changes shall be made, and an appropriate message shall be displayed.

K. **Get Class Roster.**
Purpose. Produce a list of the names and Id numbers of students currently enrolled in or registered for a course.
Actor. A faculty member.
Input. A course number and an indication of whether an enrollment or a registration list is requested.
Result. The requested class roster is displayed.
Exception. If an enrollment list for a course not currently being taught or a registration list for a course not to be taught next semester is requested, no display is returned.

L. **Room.**
Purpose. Display the size (i.e., number of seats) of an existing classroom (item III.E) or enter the identifier and size of a new classroom.
Actor. A faculty member.
Input. A classroom identifier and the number of seats (if a new classroom is to be entered) or just the identifier (if the size of an existing classroom is requested).
Result. The requested information is displayed or the identity and size of the new classroom is stored in the database.
Exception. If an existing classroom size is requested and the specified classroom identifier is incorrect, an appropriate error message is displayed.

M. **OLAP Query.**
Purpose. Allow the user to input an arbitrary query from the screen.
Actor. A faculty member (who, it is assumed, knows the database schema).
Input. A query in the form of a single SELECT statement.
Result. The table produced by the query is output on the screen with attribute names (where possible) heading each column.
Exception. If a statement other than a SELECT is input or if the statement is incorrect, an appropriate error message is returned.

N. **End of Session.**
Purpose. End the session.
Actor. A student or a faculty member.
Input. The actor clicks the logout button.
Result. Any subsequent use cases with the system require a new authentication.

VI. System Issues

A. The system shall be implemented as a client/server system. The client computer shall be a PC on which the application programs will execute.

B. The user interface shall be graphical and easy to use by students and faculty with little or no training.

C. The database can be any SQL database that executes on an available server computer and provides a transactional interface (in other words, it can perform the commit and abort operations).

VII. Deliverables

A. A Specification Document that describes in detail the sequence of events (input/output) that occurs for each use case, including
   1. The forms and controls to be used
   2. The effect of using each control on each form, including any new forms that are displayed as a result of each possible action
   3. The errors for which the system checks and the error messages that are output
   4. Integrity constraints

B. A Design Document that describes in detail
   1. An entity-relationship (E-R) diagram that describes the system
   2. The declaration of all database elements (including tables, domains, and assertions)
   3. The decomposition of each use case into transactions and procedures
   4. The behavior of each transaction and procedure

C. A Test Plan describing how the system will be tested, including how each of the numbered requirements and specifications will be tested

D. A demonstration of the completed system (including running the tests in the Test Plan)

E. Fully documented code for the system

F. A User Manual with separate sections for student and faculty

G. Version 2 of the Specification Document, the Design Document, and the Test Plan, describing the as-built system

CASE STUDY

14.3 Requirements Analysis—New Issues

The next phase of the project is to analyze the Requirements Document and produce a formal Specification Document. Experience has shown that, no matter how carefully the Requirements Document is written, when the implementation team analyzes the requirements in order to prepare the Specification Document, a number

of new issues will be identified. Parts of the Requirements Document will be found to be inconsistent or incomplete, and questions will be raised about the desired behavior of the system in certain previously unforeseen situations. The implementation team customarily presents these issues to the customer, who resolves them in a written document. The resolved issues then become part of a revised version of the Requirements Document and part of the initial version of the Specification Document. This entire scenario underscores the difficulties in precisely specifying the desired behavior of a proposed system.

When the Requirements Document given in Section 14.2 was analyzed by our local implementation team, a number of issues were identified. Below we present some of these together with their resolution. Your local implementation team is likely to discover other issues.

Issue 1. What if, during the Course Information use case, an attempt is made to add a new prerequisite for a course such that the prerequisites form a cycle? For example, course A is a prerequisite for course B, B is a prerequisite for course C, and C is a prerequisite for A. In other words, course A is a prerequisite for itself.

Resolution. A new database integrity constraint must be added to deal with this situation: there must not be a cycle of prerequisites. Any transaction that implements the Course Information use case shall check for this condition, and, if it exists, the prerequisite shall not be added and an appropriate message shall be presented to the user. (The check for circularity in the general case is not simple. We might, however, require that the prerequisite for a course have a lower number than the course itself. Such a requirement eliminates the possibility of circularity.)

Issue 2. What if, during the Course Information use case, an attempt is made to add a new prerequisite for a course, and a student who does not have that prerequisite is already registered for that course?

Resolution. A new prerequisite for a course does not apply to the offering of the course (if any) in the following semester.

Issue 3. What if, during the Course Information use case, the maximum number of students allowed in a course is reduced to a value that is less than the number of students who have already registered for the course?

Resolution. Room rescheduling is a fact of academic life, so this use case must be allowed. However, the appropriate number of students must be deregistered from the course to bring the total number of registered students to the new maximum. The students shall be deregistered in the reverse order that they were registered. All deregistered students shall be notified in writing.

Issue 4. What if, during the Course Information use case, an attempt is made to change the day and/or time a course is offered?

Resolution. A change in day and/or time does not apply to the offering of the course (if any) in the following semester.

Issue 5. What if, during the Course Information use case, a course is canceled?

Resolution. Cancellation applies to the next semester. Students registered for the next semester shall be deregistered and notified in writing.

Issue 6. What if, during a Student/Faculty Information use case, an attempt is made to change a student's Id number?

Resolution. An Id number (in contrast to a name or password) is permanently associated with an individual, so an attempt to change it makes no sense, except if it was originally entered in error. As a result, a change in Id number is allowed only if there is no other information relevant to the student in the system.

Issue 7. Several of the use cases, such as Get Grade History, Get Registered Courses, and Get Enrolled Courses, produce reports that describe the state of the database at a particular instant. Should those reports include the date and time they were produced?

Resolution. Yes, all such reports shall include the date and time.

Issue 8. Should the information in items III.D and III.F contain the year as well as the semester?

Resolution. Yes, and the End of Semester use case shall appropriately update the year. This information shall be initialized at startup time.

Issue 9. How many digits should be used to indicate the year in the system?

Resolution. Four.

CASE STUDY

14.4 Specifying the Student Registration System

A Specification Document contains a complete description of what the system is supposed to do from the viewpoint of its end users—it is an expanded version of the Requirements Document. For a transaction processing system, the Specification Document should include

- The integrity constraints of the enterprise
- A complete description of the user interface
  - A picture of every form with every control specified
  - A description of what happens when each control is used, including
    - What application procedure is executed
    - What changes occur in the form or what new form is displayed
    - What error situations can occur and what happens in each such situation
- A description of each use case, including
  - The information input by the user and what events cause the use case to be executed

- A textual description of what the use case does (for example, "the student is registered for the course")
- A list of conditions under which the use case succeeds or fails, and what happens in each case

The Specification Document might also contain other information related to project planning (such as schedules, milestones, deliverables, cost information, etc.), information related to system issues (such as software and hardware on which the system must run), and any time or memory constraints. The Table of Contents for the Specification Document for the Student Registration System has the following sections:

I. Introduction
II. Related Documents
III. Forms and Use Cases
IV. Project Plans

A. **Milestones**

B. **Deliverables**

Note the relationship between the contents of the requirements and specification documents.

14.4.1 UML Sequence Diagrams

Part of the plan for developing a Specification Document from a Requirements Document might be to expand each use case into a UML sequence diagram. A **sequence diagram** is a graphic display of the temporal order of the interactions between the actors in a use case and the other modules in the system.

Figure 14.2 is a sequence diagram for the Authentication use case. The actors in the use case (in the figure, a student or a faculty member) and the pertinent modules in the system (in the figure, the Web server and the database) are labelled at the top of the diagram. Time moves downward through the diagram, and the vertical line descending from each actor and module show its lifetime during the use case. The boxes on the vertical line show when that actor (or module) is active in the use case. The horizontal lines show particular actions taken by an actor or module. Thus, the sequence diagram starts with the student or faculty member typing in the URL of the Student Registration System, after which the Web server displays the Welcome Form (Figure 14.3). Note the notation used for conditional actions: [*status = student*] *Display Student Options Form.* This means that if the status returned by the Authentication interaction is "student," the Web server displays the Student Options Form.

Note that Specification III.A in the Specification Document given in Section 14.5 is an English-language description of the sequence diagram in Figure 14.2.

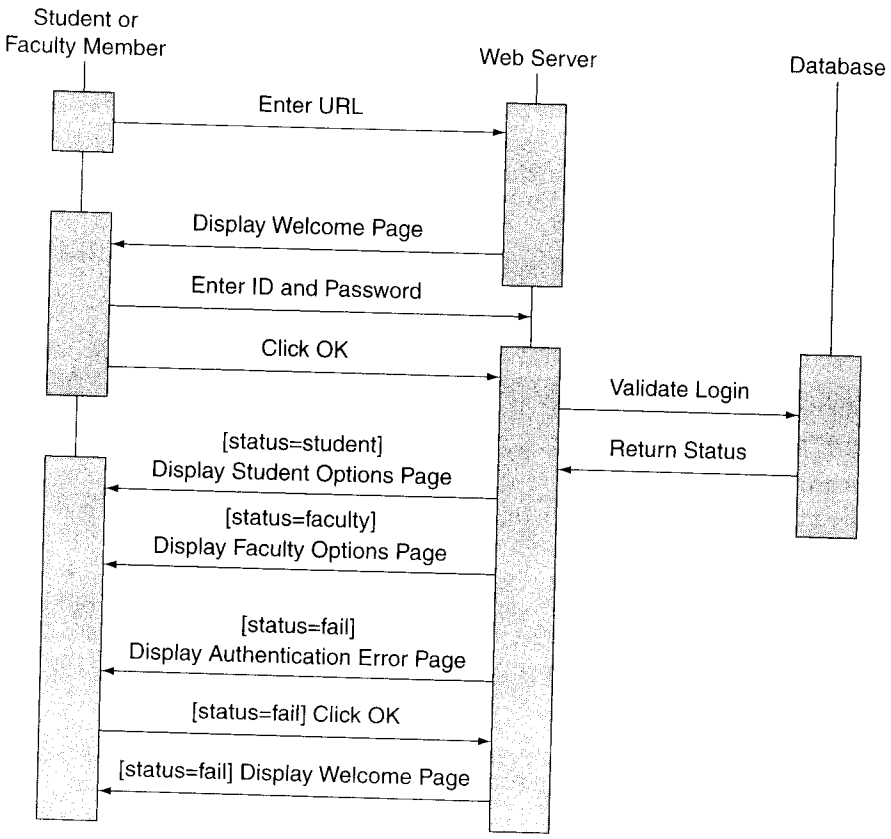

FIGURE 14.2 A sequence diagram for the Authentication use case.

CASE STUDY

In the following section, we present the initial part of Section III of the Specification Document. We give a design for the database in the the Student Registration System in Section 4.8 and the design and part of the code for the Registration Transaction in Section 15.7.

14.5 The Specification Document for the Student Registration System: Section III

Section III of the Specification Document—"Forms and Use Cases"—contains a detailed description of all interactions with users. Its initial part might look like this:

III. Forms and Use Cases

A. When the Student Registration System is entered, Form 1, the Welcome Form (Figure 14.3) is displayed. In Form 1

   1. The Id and Password text boxes are filled in.

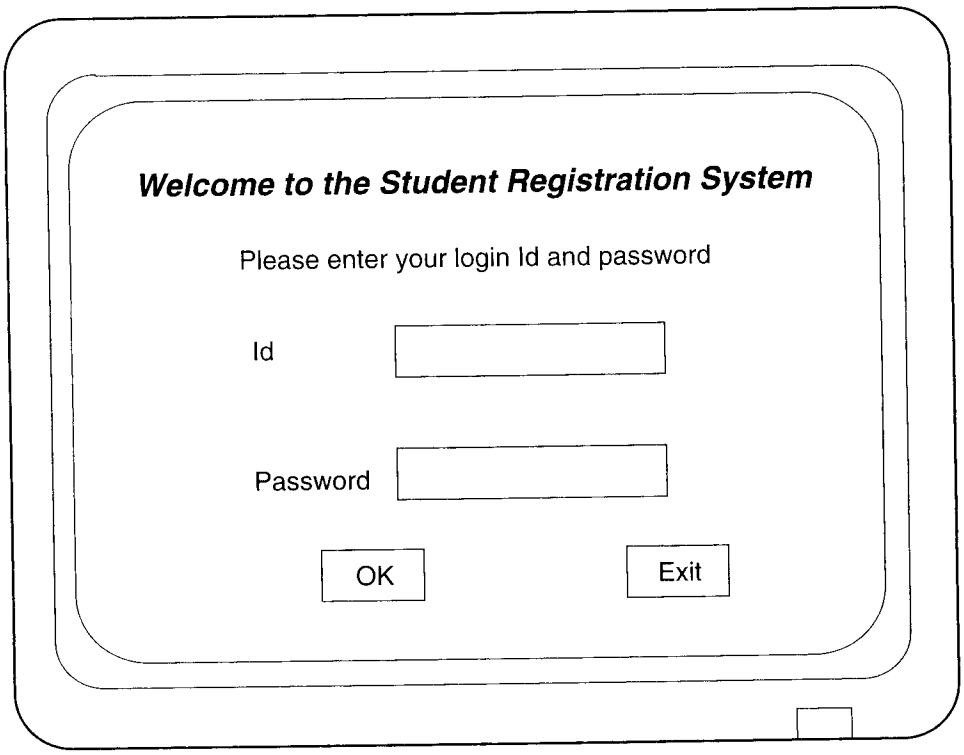

FIGURE 14.3 Introductory form for the Student Registration System.

2. The OK command button is clicked to run the *Authentication* use case.
   a. If the Authentication use case fails, Form 2, the Authentication Error Form, is displayed.[4] In Form 2, clicking the OK command button returns to Form 1.
   b. If the Authentication use case succeeds and the authenticated user is a student, Form 3, the Student Options Form, is displayed (as described in Specification III.B).
   c. If the Authentication use case succeeds and the authenticated user is a faculty member, Form 4, the Faculty Options Form, is displayed.
3. The Exit command button is clicked to display Form 5, the Do You Really Want To Exit Form. In Form 5
   a. The Yes command button is clicked to exit the Student Registration System.
   b. The No command button is clicked to return to Form 1.

[4] We omit the figures for the other forms; they would be included in the actual specification.

CASE STUDY

B. In Form 3, the Student Options Form,
   1. The Course description menu item is selected to run the *Get Course Names* use case, which displays Form 6, the Course Name Form. In Form 6
      a. A course option box is selected.
      b. The OK command button is clicked to run the *Get Course Description* use case, which displays Form 7, the Course Description Form.
      c. The Cancel command button is clicked to return to Form 3.

The remainder of Section III of the Specification Document is similar and is therefore omitted.

14.6 The Next Step in the Software Engineering Process

After the implementation group has expanded the Requirements Document into the Specification Document and the customer has signed off on the Specification Document, the design portion of the project can begin. In contrast with specifications, which describe *what* the system is supposed to do, the design describes *how* the system is to do what it does. We discuss design in Chapter 15 and the specific issues involved in designing databases in Chapters 4 and 6. We gave a complete database design for the Student Registration System in Section 4.8 and the complete design and part of the code for the Registration Transaction in Section 15.7.

One reason so much time and effort is put into producing requirements and specification documents is that experience has shown it to be surprisingly difficult to build a system that actually satisfies the customer's needs. Often the system's requirements are quite complex, and the customer has difficulty articulating his needs in the precise manner needed for programming, or he leaves out important details (such as what is supposed to happen if a course is canceled after a number of students have registered for it) or specifies some feature and then is unhappy with that feature when it is implemented.

The U.S. Department of Defense, which is probably the largest customer for software systems in the world, says that over 56% of all the defects in software systems it contracts for are due to errors in the specifications. It is cheaper and more efficient to work with the customer at the beginning of the project to sharpen and refine the specifications than it is to reimplement the system at the end of the project if it does not meet the customer's needs.

BIBLIOGRAPHIC NOTES

There are many excellent books on software engineering—for example, [Sommerville 2000]; [Pressman 2002]; [Schach 1999]. One of the very few books that address software engineering for database and transaction processing systems is [Blaha and Premerlani 1998].

EXERCISES

14.1 Prepare a Requirements Document for a simple calculator.

14.2 According to the Requirements Document for the Student Registration System, one session can include a number of use cases. Later, during the design, we will decompose each use case into one or more transactions. The ACID properties apply to all transactions, but a session that involves more than one transaction might not be isolated or atomic. For example, the transactions of several sessions might be interleaved. Explain why the decision was made not to require sessions to be isolated and atomic. Why is a session not one long transaction?

14.3 Suppose that the database in the Student Registration System satisfies all the integrity constraints given in Section IV of the Requirements Document Outline (Section 14.2). Is the database necessarily correct? Explain.

14.4 The Requirements Document for the Student Registration System does not address security issues. Prepare a section on security issues, which might be included in a more realistic Requirements Document.

14.5 In the resolution of issue 2 in Section 14.3, the statement was made that new prerequisites do not apply to courses offered in the next semester. How can a Registration Transaction know whether or not a prerequisite is "new"?

14.6 Suppose that the Student Registration System is to be expanded to include graduation clearance. Describe some additional items that must be stored in the database. Describe some additional integrity constraints.

14.7 Prepare a Specification Document for a simple calculator.

14.8 Prepare a Specification Document for the controls of a microwave oven.

14.9 Specify a use case for the Student Registration System that assigns a room to a course to be taught next semester.