

Design: Architecture and Methodology

OBJECTIVES

- Understand the differences between architectural and detailed design.
- Gain knowledge of common software architectural styles, tactics, and reference architectures.
- Gain knowledge of basic techniques for detailed design, including functional decomposition, relational database design, and object-oriented design.
- Understand the basic issues involved in user-interface design.

7.1 Introduction to Design

Once the requirements of a project are understood, the transformation of requirements into a design begins. This is a difficult step that involves the transformation of a set of intangibles (the requirements) into another set of intangibles (the design). Software design deals with how the software is to be structured—that is, what its components are and how these components are related to each other. For large systems, it usually makes sense to divide the design phases into two parts:

- **Architectural design phase:** This is a high-level overview of the system. The main components are listed, as well as properties external to the components and relationships among components. The functional and nonfunctional requirements along with technical considerations provide most of the drive for the architecture.
- **Detailed design phase:** Components are decomposed to a much finer level of detail. The architecture and the functional requirements drive this phase. The architecture provides general guidance and all functional requirements have to be addressed by at least one module in the detailed design.

Architectural design phase The period during which the high-level overview of the system is developed.

Detailed design phase The phase in which the architectural components are decomposed to a much finer level of detail.

Figure 7.1 illustrates the relationships among requirements, architecture, and detailed design. Ideally there is a one-to-one mapping between each functional requirement and a module in the detailed design; the influence of the requirements on the architecture is illustrated with the wide arrow; notice in this case the most influential requirements may be nonfunctional requirements such as performance and maintainability. The architecture drives the detailed design, with the mapping being ideally from one architectural component to several detailed modules.

The architecture drives the detailed design, with the mapping being ideally from one architectural component to several detailed modules.

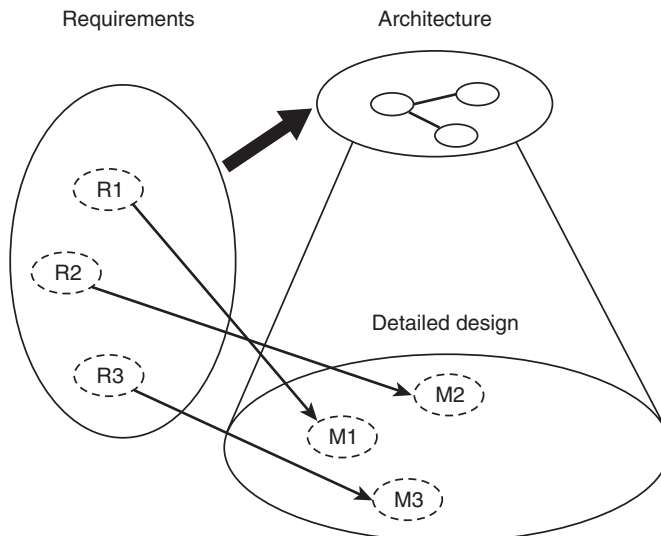


Figure 7.1 The relationships among requirements, architecture, and detailed design.

Smaller systems may get away with not having an explicit architecture, although it is useful in almost all cases. In traditional software processes, the ideal is for the design to be created and documented up to the lowest level of detail possible, with the programmers doing mainly translation of that design into actual code. Agile methodologies and the actual processes followed in many companies, especially for smaller systems, give the programmer a much more important role in the detailed design. In many Agile methodologies, the programmer ends up doing the actual detailed design.

There are many different ways of specifying a design. Given that for most people, pictorial representations of information are useful, it is desirable for a design notation to be graphical. Although many different notations have been proposed, in the last few years the Unified Modeling Language (UML) has gained widespread popularity and is the de facto standard, at least for **object-oriented (OO) design**. This chapter is not dedicated to OO design, but a brief discussion of OO and UML is presented in Section 7.3.3. Here, we will simply state that OO design is a technique that models a design with classes, their relationships, and the interactions among them.

Object-oriented design A technique that models a design with classes, their relationships, and the interactions among them.

7.2 Architectural Design

7.2.1 What Is Software Architecture?

The software architecture of a system specifies its basic structure. In many ways, it is design created at a high level of abstraction. Bass, Clements, and Kazman (2003) define software architecture as follows:

The software architecture of a program or computing system is the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them.

There are several important points to note about the architecture of a system:

- Every system has an architecture. Whether you make it explicit or not, whether you document it or not, the system has an architecture.
- There could be more than one structure. For large systems, and even many small ones, there is more than one important way the system is structured. We need to be aware of all those structures, and document them with several views.
- Architecture deals with properties external to each module. At the architecture level, we should think about the important modules and how they interact with other modules. The focus is on the interfaces among modules rather than details concerning the internals of each module.

7.2.2 Views and Viewpoints

An important concept in architectural design, and design in general, is the fact that a system has many different structures (that is, many different ways of being structured) and in order to get the complete picture you need to look at many of those structures.

A view is a representation of a system structure. Although in most situations we can use view and structure interchangeably, keep in mind the structure exists whether you

represent it or not, and the view only depicts the structure. This is a distinction similar to the one for a photograph and its subject.

In a seminal paper that later became one of the foundations of the Rational Unified Process (RUP), Kruchten (1995) proposed having four architectural views to represent the requirements for a system and to unify it, plus use cases (which he called *scenarios* in his paper):

- **Logical view:** Represents the object-oriented decomposition of a system—that is, the classes and the relationships among them. In OO, a class is a conceptual element derived and conceived from the requirements. The interaction and the relationships among the classes are often also derived from the business or workflow expressed in the requirements. The notation used is basically that of a UML class diagram.
- **Process view:** Represents the run-time components (processes) and how they communicate with each other.
- **Subsystem decomposition view:** Represents the modules and subsystems, joined with export and import relationships.
- **Physical architecture view:** Represents the mapping of the software to the hardware. This assumes a system that runs on a network of computers and depicts which processes, tasks, and objects are mapped to which nodes.

Bass, Clements, and Kazman (2003) provide examples of more views, and classify them into three categories:

- **Module views:** Represent elements in static software modules and subsystems. Views of these types include the following:
 - Module decomposition views, which represent a part-of hierarchy of modules and submodules;
 - Uses view, which depicts how modules depend on each other; and
 - Class generalization views, representing the inheritance hierarchy of classes. Information represented by UML's class diagrams and Kruchten's logical view will also fall under this category.
- **Run-time views:** Represent the running structure of the program; also called component-and-connectors views. They indicate how executing modules or processes communicate with each other. Views here could be depicted with different graphical diagrams such as communicating process diagrams, client-server diagrams, and concurrency diagrams.
- **Allocation views:** Represent the mapping of software modules to other systems. Typical views of this type include deployment views, which represent the mapping of modules to hardware structures; implementation views, which map the modules to actual source files; and work assignment views, which show the person or team responsible for each module.

An important point to realize is that different views are useful to different stakeholders. For example, an implementation view, showing which modules are implemented on which files, is useful mostly to implementors, whereas a class diagram is useful for many more kinds of stakeholders.

When you are designing an architecture, it is important to keep in mind that there are many different views of it that may be useful; of course there is limited time, so you need to think hard about which views to produce.

7.2.3 Meta-Architectural Knowledge: Styles, Patterns, Tactics, and Reference Architectures

Although many systems have been developed with many different architectures, several architectures share common characteristics at many levels. Software engineers have been comparing system architectures and describing their similarities and differences for quite a long time. Much of this knowledge, sometimes called meta-architectural, has been codified in different ways to provide easier ways of comparing and choosing architectures, and to provide a starting point when creating an architecture.

The software architecture community has codified this kind of knowledge mainly in three different ways:

- Architectural styles or patterns
- Architectural tactics
- Reference architectures

Meta-architectural knowledge serves two main purposes. First, it can be used as a starting point for a particular system's architecture, saving some work and providing guidance for the final architecture. Second, it is an effective communication mechanism for providing a quick idea of the high-level structure of a system. When written in a format similar to design patterns, architectural styles are also called architectural patterns.

Architectural Styles or Patterns Software architectural styles or patterns are akin to styles in physical building architecture. Just as many buildings share a common style that many people can recognize—from Gothic to Southern U.S. antebellum—many system architectures also have a recognizable style.

Among the most common architectural styles are the following:

- *Pipes-and-filters*: A style widely used for Unix scripts, and for signal processing applications. It consists of a series of processes connected by “pipes.” The output of a process serves as the input of the next one; processes do not need to wait until the previous process finishes but can start processing input as soon as some of the input is available. Most of the time the topology is linear, but occasionally there could be forks. Although the most popular application of this style is in combining Unix commands, it is also the conceptual model for many audio and video processing applications. **Figure 7.2** shows a screen shot of *gst-editor*, an editor for the GStreamer multimedia framework.
- *Event driven*: A style in which system components react to externally generated events and communicate with other components through events. Modern graphical user interface (GUI) libraries and the programs that use them are organized with this style at some level. Many distributed systems use this style as well, as it allows for decoupling of the components and easy reorganization of the system.

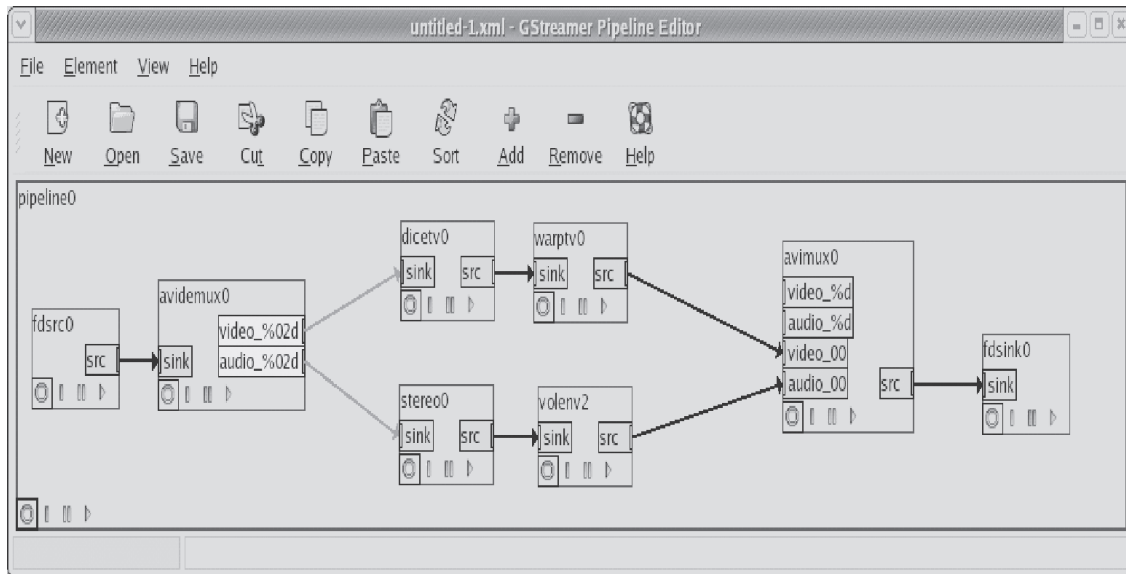


Figure 7.2 Screen shot of the GStreamer editor. A video file is read from a file, then demultiplexed into audio and video streams. Both streams are passed through two different filters, then combined again and saved to another file.

- **Client-server:** A style showing a clear demarcation between clients and servers, which reside on different nodes in a network. Components interact through basic networking protocols or through remote procedure calls (RPC). Usually there will be many clients accessing the same server. **Figure 7.3(a)** shows a client-server architecture, with several clients accessing the same server; Figure 7.3(b) shows a more complicated version, with several different kinds of servers.

The client-server architecture was heavily influenced by various hardware changes and hardware cost. First, with less powerful terminals or client boxes, many of the processing resided on the server boxes. As the client machines improved in power and dropped in price, more functions were placed in the clients. An interesting side note on this development of placing more functions on the clients or personal desktops is that it created the need to support these clients. A whole profession called IT desktop support became a necessity as a result of the powerful client desktops.

- **Model-view-controller (MVC):** A popular way of organizing GUI programs that need to display several different views of data. The main idea is to separate the data from the display. In the original version a controller took care of translating user input, such as mouse movements and clicks or keystrokes, into appropriate messages for the view. Because modern GUI libraries usually do this, there is no need for a separate controller class. Modern variations of this pattern use just the model and view classes. The model is responsible for storing the data and for notifying the views whenever the data change. The views register with the model, can modify the model, and respond to changes in the model by redrawing themselves. **Figure 7.4** shows a simplified dia-

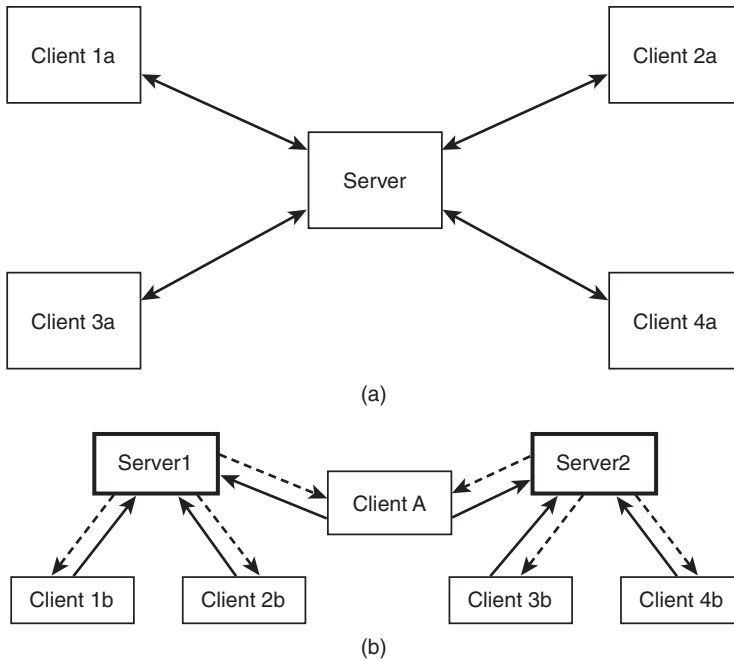


Figure 7.3 (a) Client-server style with one server and many clients. (b) Client-server style with several servers.

gram of an MVC architecture. MVC can be implemented with a client-server architecture with both the view and part of the controller residing on the client box while the model portion resides on the server.

- **Layered:** A style in which components are grouped into layers, and the components communicate only with other components in the layers immediately above and below their own layer. When the layered architecture is combined with a client-server architecture and the layers may reside in different computers, they are usually called tiers rather than layers. **Figure 7.5** depicts a common layered system, with the Java API

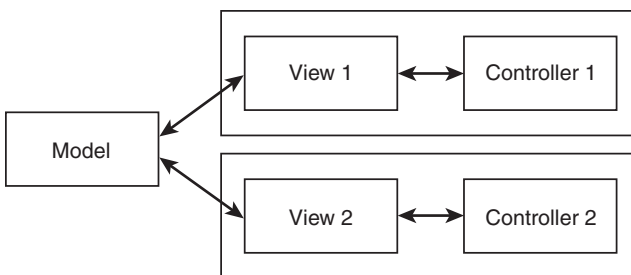


Figure 7.4 Model-view-controller style.

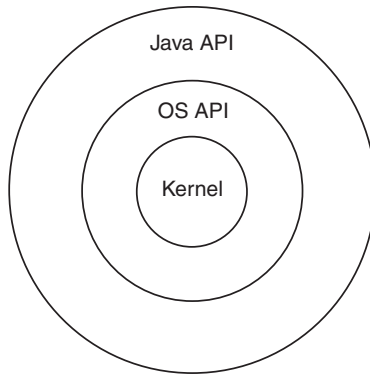


Figure 7.5 A common layered system, with the Java API implementation calling operation system functions, which in turn communicate with kernel functions.

implementation calling operating system functions, which in turn communicate with kernel functions. The Java API calls functions directly through the operating system API, not through the kernel. While layered architecture keeps the components themselves focused on specific tasks and facilitates the detection of problems, it sometimes presents a performance problem in terms of the number of layers a message may have to travel through before being processed.

- **Database centric:** A style in which a central database and separate programs access the database. The programs communicate only through the database, not directly among themselves. A big advantage of this style is that it introduces a layer of abstraction for the database, which is usually called a database management system (DBMS). A modern DBMS can guarantee many user-defined constraints on any data entered in the database, which allows the programs to assume those constraints; this leads to a relatively coupled system of multiple programs to the database. Rather than building one huge system, you can build several smaller programs. By far the most popular DB technology is that of relational databases. In fact, in most cases, database centric really means relational database centric. Later in this chapter we discuss relational database technologies and database design in more depth.

The database-centric style is commonly combined with the client-server style, as illustrated in **Figure 7.6**. There is a central database server, running a database management system (DBMS) that is accessible over the network. Programs running on client machines interact with the database server. In the traditional configuration, called two tier, the clients interact directly with the database.

- **Three tier:** A variation on the database-centric and client-server approaches that adds a middle tier between the clients and servers, implementing much of the business logic. The clients cannot access the database directly and have to go through the middle tier. This way, the business logic gets implemented in one place, simplifying the system.

For many systems, the business logic is very hard to express with just the kinds of constraints supported by relational DBMS. Although the basic technologies for relational databases are standardized, some more-advanced features, such as stored

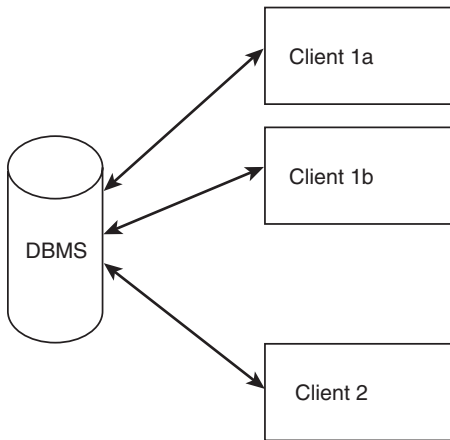


Figure 7.6 The database-centric style. Typically, the clients communicate directly with the database.

procedures and triggers, are not. These features would be needed to implement much of the business logic required for many systems, but implementing them inside the DBMS would mean that we have to keep using that specific DBMS or we would need to port all those triggers and stored procedures to a new DBMS.

The three-tier style is often used as a model for web-based applications. The client machines access an application server through a web browser; the application server implements the business logic and communicates with the database. This kind of architecture can also be viewed as a variation of the MVC architecture, with the database being the model, and the application server implementing the controller and generating the views that will be displayed by the clients with a web browser.

Three-tier architecture may be extended to n -tier with additional middle tier servers. **Figure 7.7** illustrates the three-tier architectural style. **Figure 7.8** illustrates a

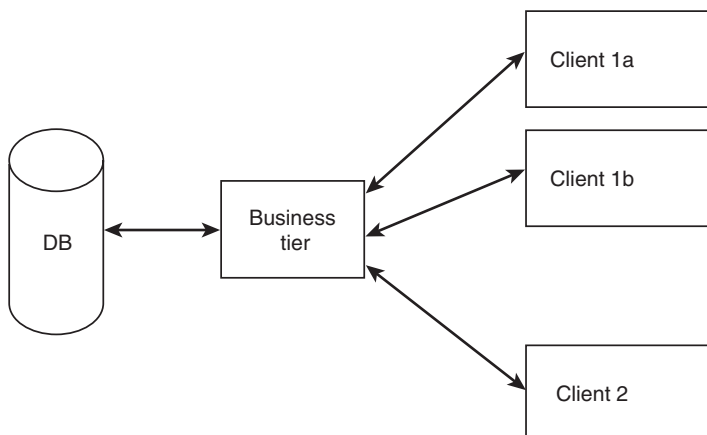


Figure 7.7 A three-tier style, in which clients do not connect directly to the database.

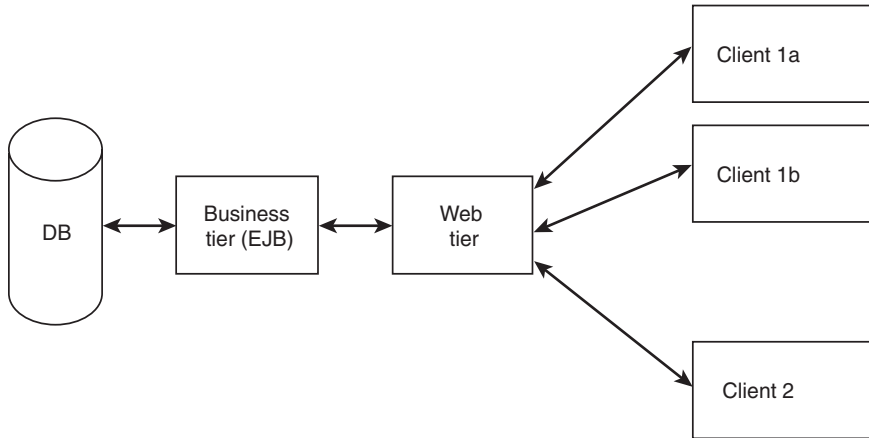


Figure 7.8 The J2EE reference architecture.

specific four-tier architecture, the J2EE reference architecture, which will be discussed later in this section.

Notice that for simplicity, these diagrams illustrate only one application. There may be several applications accessing the same database, in which case we can also view each application as a separate component within the database-centric architecture of the whole system.

Architectural Tactics Different from styles and patterns, architectural tactics solve smaller problems and do not directly affect the overall structure of the system. They are designed to solve very specific problems for the various architectural styles.

For example, assume that we have a three-tier distributed system, and we want to increase its reliability. Specifically, we are worried about the possibility of some component failing without anybody noticing until something goes really wrong. Thus we try to improve on fault detection. We decide on two possible tactics; in both cases we have another component that is responsible for detecting failures.

The first tactic is to have each component send a message to the fault detector at prescribed intervals. The fault detector knows when it should receive those messages and produces a notification if it does not receive the message within the appropriate time. In distributed systems this is commonly known as a heartbeat.

The second tactic is to have the fault detector send a message to the other component and wait for a response. If it does not receive the response, it knows an error has occurred. This tactic is known as a ping/echo in Internet applications.

Although each specific tactic is applicable only to a limited number of problems, knowing the right tactics can save much time for the software architect. Bass, Clements, and Kazman (2003) provide a small catalog of tactics in their book *Software Architecture in Practice*.

Reference Architectures A third category of meta-architectural knowledge is that of reference architectures—full-fledged architectures that serve as templates for a whole

class of systems. Taylor, Medvidovic, and Dashofy (2009) define reference architecture as “the set of principal design decisions that are simultaneously applicable to multiple related systems, typically within an application domain, with explicitly defined points of variation.” Design decisions include all aspects of design, including structure, functional behavior, component interactions, nonfunctional properties, and even some implementation decisions.

7.3 Detailed Design

The architectural design of a system, together with the requirements, need to be refined to produce a detailed design. The development process used determines the level of detail to which the design is decomposed and the level of formality of its documentation. If the design is carried out to the finest level of detail, the implementation task is an almost one-to-one mapping of that design to the implementation language; but often the design is not specified to its finest level, leaving some detailed design tasks to be done in the implementation phase.

7.3.1 Functional Decomposition

Functional decomposition is used mostly in structured programming, but some of the ideas can be used with other programming paradigms. The basic idea is to decompose a function or module into smaller modules, which will be composed together to form the bigger module. Traditionally the modules are other systems or procedures that are called from the main module.

When using object-oriented programming languages, this technique can be used to do the initial decomposition of a system into modules by functionality, or to decompose methods that are particularly hard to implement. Although object-oriented systems and languages receive most of the attention today, there are still many systems that are developed with procedural techniques. In fact, many small web-based applications can be modeled this way, with the system decomposed into functional modules and each module corresponding to one or a few related webpages.

We will illustrate this technique with an example. Suppose you are designing a system for managing course registration and enrollment. The requirements specify four tasks that need to be done: (1) modify and delete students from the database; (2) modify and delete courses from the database; (3) add, modify, and delete sections for a given course; and (4) register and drop students from a section.

Doing a functional decomposition of this system, you would decompose the main module into four submodules for dealing with students, courses, sections, and registration. The first three modules would be decomposed further into modules for adding, modifying, and deleting, while the fourth module would be decomposed into two modules for registering or dropping a student from a section.

The usual process is to produce module decomposition diagrams, where the modules are represented by rectangles, and there is some form of standardized numbering system, with numbers assigned to each module according to their level. The important characteristic of the numbering scheme is that each module gets assigned a unique number, and it is easy to see the level of the module and who its parent is.

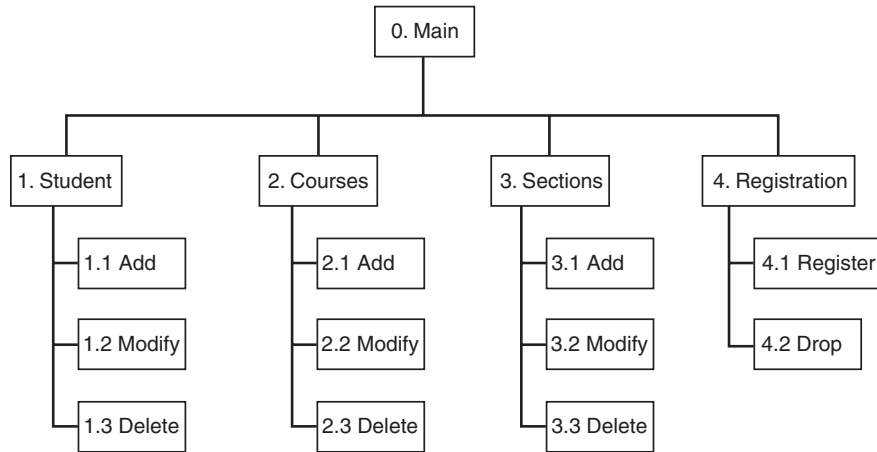


Figure 7.9 A module decomposition diagram for a student registration system.

A module decomposition diagram is presented in **Figure 7.9**. It depicts a simple system partitioning based on externally perceived functionality. Notice that there are usually several different ways to partition the system. For example, we could also partition the same system as shown in **Figure 7.10**, abstracting all the database operations into one specific database module.

In Figure 7.9, the Add, Modify, and Delete functions for the three entities—Students, Courses, and Sections—are uniquely tied to the three respective entities. In Figure 7.10, the design focuses on potential reuse of the three functions. The Add, Modify, and Delete functions are grouped under database common services for potential reuse and multiple

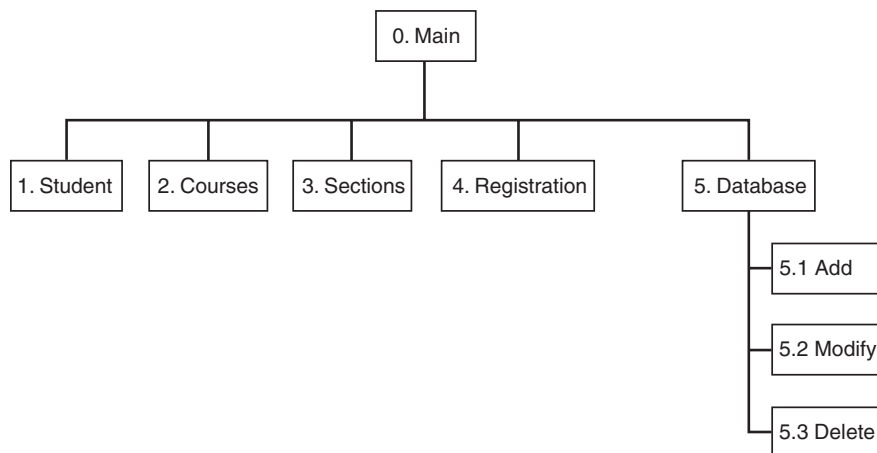


Figure 7.10 An alternative module decomposition with database operations in its own module.

usages. In design, we should not only be concerned with the actual structures but also ask which one is the preferred alternative. Note that in Figure 7.9, the individual entities and their functions serve a single purpose and individually appear very cohesive, while in Figure 7.10 the database common services serve multiple purposes and provide some reuse. However, with common services for reuse, there is a certain amount of coupling introduced among the entities. Which design is a better alternative is a complex topic. This issue of assessing “good” design will be discussed extensively in Chapter 8 where the notion of cohesion and coupling will be better defined and clarified.

7.3.2 Relational Database Design

All reasonably sized applications must handle a large amount of information. Today, we can assume the existence of a convenient database that can be incorporated into the design of any system for information storage and retrieval. We will include a database in our discussion of designing software systems, but not enter into extensive analysis of database technologies.

Most database and business applications use relational database technologies. First proposed by E. F. Codd of IBM in the late 1960s, relational databases are grounded on mathematical concepts of sets and relations and are relatively simple to use and understand. They are also relatively simple to implement and can be implemented efficiently, which has led to their popularity. See the Suggested Readings section for more information on relational databases.

In a relational database, information is stored in tables, also called *relations*. They are two-dimensional sets of data, with rows (also called *tuples*) and columns (also called *attributes*). In the simplest case, each row corresponds to an object or entity in the real world, and columns correspond to attributes of those entities. Relational database theory requires that a set of attributes is identified as the primary key of a table, but most implementations do not have this requirement.

Database design concentrates only on how to represent the data required for the program and how to store it efficiently on a relational database. It can be divided into four phases:

- **Data modeling:** This usually entails creating an entity-relationship (ER) model of the data. The ER model may have been created in the requirements analysis step, as discussed in Chapter 6, but it may still need to be extended and refined.
- **Logical database design:** Taking as input a detailed ER model, a normalized relational schema is produced. The relational schema is a set of tables together with foreign key relationships.
- **Physical database design:** In this step, the main decisions include what data type to use for each attribute and what indexes to create. Sometimes the logical schema is transformed for efficiency (but this should be done with extreme care). The output is a detailed set of structured query language (SQL) statements that implement the logical schema. Sometimes decisions may be made about more low-level issues, such as which relations are stored on which hard drives, although most of the time these decisions are made during deployment and maintenance.

- *Deployment and maintenance*: The final details are ironed out, including where the relations are stored. Not only do the details of the DBMS software used need to be known, but also the specific hardware the system is going to be deployed on needs to be specified. As the system is used, some of these decisions, along with issues dealing with physical design, such as index creation, may be modified to improve performance, reduce space, or reflect changes in hardware or system usage.

Data Modeling During this phase, a detailed and complete ER model is created. Documentation for the ER model includes an ER diagram, along with annotations for information not reflected on the diagram. The best practices suggest the creation of a data dictionary containing all information pertaining to each attribute.

The requirements documentation may already include an ER diagram that would need to be refined, or a diagram would need to be created from scratch if one does not exist. An ER diagram contains three main types of objects: (1) entities (represented by rectangles), (2) attributes (represented by ovals), and (3) relationships (represented by diamonds). Entities represent objects or things in the real world (or, more precisely, in our mental model of the real world). They have attributes and are related to other entities by relationships. Relationships may also have attributes. In an ER diagram we usually represent entity and relationship types, rather than particular instances of entities or relationships.

Entity types represent the kinds of things we are modeling. Entities have attributes and one or more attributes are marked as the identifier, which will allow us to distinguish among entities of the same type. Weak entities do not have an identifier and are dependent on another entity for their identity.

Attributes are classified as simple or composite and as single-valued or multivalued. Simple attributes are those that do not need to be subdivided further (that is, those that your DBMS supports as primitives), while composite attributes are formed of several parts. For example, if we represent the full name of a person as a string, we consider it a simple attribute. If we divided into first, middle, and last names (as is commonly done in the United States), then it is a composite attribute. Each of the parts (first, middle, last) in this case would be strings, but in other cases they could be composed of several parts themselves.

Most attributes have at most one value. For example, you have one full name (you can divide it into pieces of course), one date of birth, and so on. However, for some other attributes, the same entity can have several values at the same time. Prime examples are email addresses and phone numbers; the same person can have many email addresses or many phone numbers. We call these kinds of attributes multivalued attributes and represent them with a double oval.

It is important to not confuse multivalued attributes with composite attributes. Composite attributes have different parts, while multivalued attributes have a set of values for one entity. It is also important to keep in mind that databases usually keep a snapshot in time of the values rather than full historical information. Basically, the fact that you could change your name does not make it a multivalued attribute; at any given time, you have only one official full name.

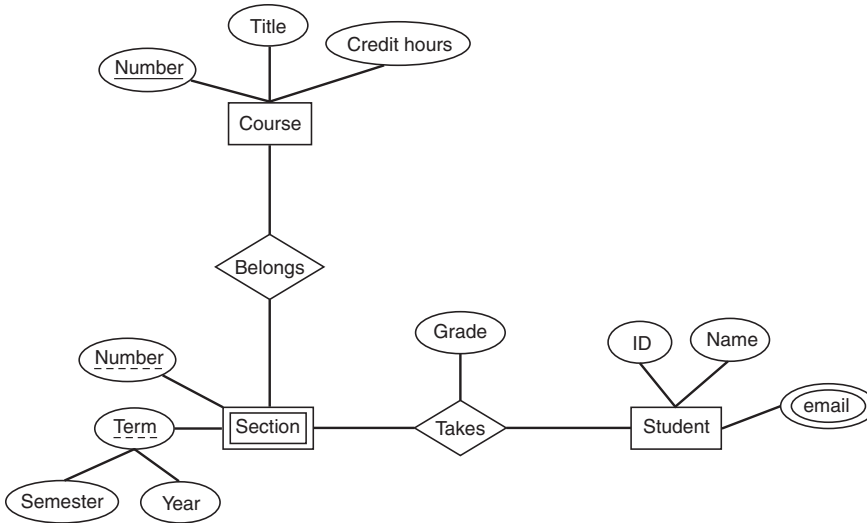


Figure 7.11 An entity-relationship diagram for courses, sections, and students in a course registration database.

Relationships specify associations among two or more entities. Relationships are classified by their cardinality (the number of entities that participate in a relationship), and modality. These concepts were also presented in some detail in Chapter 6.

Figure 7.11 shows an ER diagram for a portion of a course registration database. It represents courses, sections, and students. A section is a weak entity, because it needs the course number (which is an attribute of the course, not the section) to be uniquely identified. A student enrolls in a section, and, at the end of the term, is assigned a grade for the section. Notice that the grade is an attribute of the relationship, not of any of the entities. A student gets a grade for a particular section.

Logical Database Design Logical database design implies transforming the detailed ER diagram into a set of tables, together with foreign key relationships. We can formalize the process as follows:

1. *Transform entities:* Create a table for the entity, with all its simple and single-valued attributes. For composite attributes, use only the simple parts, with an appropriate naming convention. Weak entities have a primary key formed with the primary key of the identifying relationship and their discriminator. For example, the course and section entities would be transformed as shown in **Figure 7.12**.
2. *Creation of new tables:* For each independent multivalued attribute, create a new table containing as attributes the primary key of the entity and the multivalued attribute. If the attribute is composite, then use the simple parts only. The primary key of this relation is formed with all attributes. For example, the student entity has a multivalued attribute, email. Because the same student can have more than

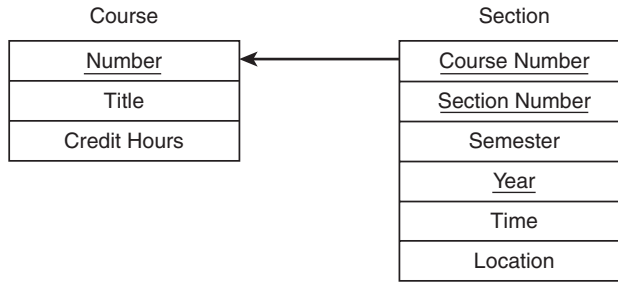


Figure 7.12 A relational schema diagram for course and section.

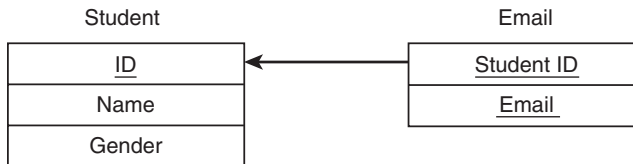


Figure 7.13 A relational schema diagram for students and email.

one email, we need to create one table for the student, and another one for those emails. The relational schema would resemble that of **Figure 7.13**.

3. *Transform relationships:* Notice that identifying relationships for weak entities have already been transformed in Step 1. The transformation to be done here depends on the kind of relationship.
 - a. *One-to-many or many-to-one:* You do not need to create a new table, unless the relationship has attributes. Just add a foreign key reference on the table corresponding to the entity next to the “many” side—that is, the entity that can be related to just one entity. Note that this is what we did for Section in Figure 7.12, although that was for a weak entity.
 - b. *One-to-one:* Follow the same process used with one-to-many, but you have a choice now. You can put the foreign key reference on either side. As a rule of thumb, choose the entity that always participates in the relationship to minimize nulls or the entity that you expect will have the fewest instances.
 - c. *Many-to-many:* Create a new table, with foreign key references to the participating entities, plus any attributes of the relationship. The primary key is the union of the attributes in the primary keys of the participating entities. For example, the Takes relationship in **Figure 7.14** would need to be mapped to a new table. Figure 7.14 shows the mapping, along with the two tables it references, Section and Student.

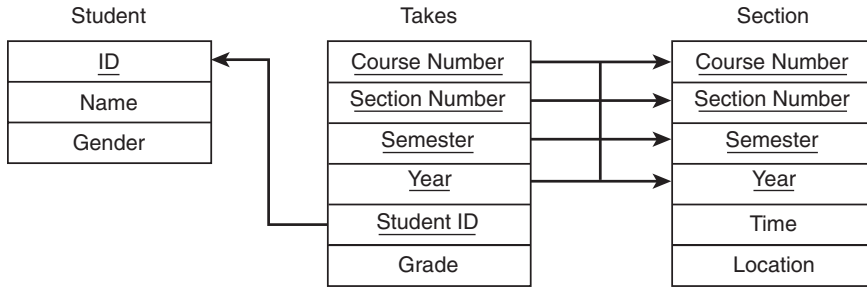


Figure 7.14 A relational schema diagram representing a many-to-many relationship.

Notice that in Figure 7.14 the arrows from Takes to Section are joined, denoting that the foreign key relationship is for the four attributes together rather than for each attribute individually. Also, many people would prefer to add a new identifier to the section table to avoid having composite foreign key references; in that case, the ER diagram would need to be updated.

- d. *Ternary relationships:* As with many-to-many relationships, you need to create a new table, containing foreign key references to the participating entities in addition to any attributes the relationship may have.

Following this process, and assuming your ER diagram was normalized, you will always achieve normalization—a situation where each row in a table represents just one simple fact rather than several. Alternatively, we can say that each table contains information pertaining to only one entity or relationship. Normalization helps ensure that information is stored only once, minimizing redundancy. There is a mathematical theory of normalization on relational databases that, for space reasons, we cannot cover in this text. See the Suggested Readings section at the end of the chapter or your favorite database textbook for additional coverage of this topic.

Physical Database Design During physical database design, the following decisions are made:

- *What data types to assign to each attribute:* Depending on which data types are supported by the DBMS, make sure that all possible data values are represented and that an eye is kept on performance. Most relational DBMS support fixed and varying size characters, fixed-precision numbers and dates, with possibly others such as IEEE floating point numbers, and integers stored in binary representation. A common issue is how to encode certain attributes; for example, you could store the gender of students in many different ways. It may be a string (“male” or “female”), a Boolean, or an integer. A common technique is to create a smaller encoding such as “M” or “F” and a new table that allows transforming that encoding to the label we want (e.g., male/female). Other issues such as encryption or compression of certain attributes may also arise.

- *What indexes to create:* Indexes consume space, but greatly increase search performance. By default, the primary key of each table is indexed to facilitate constraint checking and assuming the majority of joins would use the primary key. If you know that entities will be searched by specific fields, you may want to create other indexes.
- *Denormalization:* Sometimes, usually for performance reasons, tables will be denormalized; that is, information will be added to the table that does not really belong there or can be obtained from other tables, introducing redundancy. This should be done as a last resort, because it may confuse developers.

On very rare occasions, you may decide to combine the information on two tables. If the tables have a one-to-one relationship, this may make sense. We recommend checking the ER diagram. If the entities are conceptually different, keep them on separate tables.

Deployment and Maintenance During deployment, the final decisions are made. Specifying the characteristics of the hardware the database will reside on and deciding which tables go on which files or hard drives are some of the decisions that will be made.

While you are using the system, the usage profile may change or some performance bottlenecks may become apparent. In many cases, performance can be improved by altering the data type of some attribute or, more commonly, by adding or deleting an index, without affecting the programs that access the database in any way. Of course, if the changes affect the programs, then you have to do program maintenance.

7.3.3 Object-Oriented Design and UML

Many modern software systems are developed using object-oriented techniques. The requirements are usually expressed mainly with use cases. Also, preliminary class diagrams may have been produced in the requirements analysis step.

Most documentation for object-oriented projects is presented in the form of UML diagrams. The UML is a graphic design notation, standardized by the Object Management Group (OMG), with wide industry and academic support. We will use UML in this section, but we will not try to cover UML in detail. For additional information on UML, see the Suggested Readings section at the end of the chapter.

During the design step, you will decide on exactly which classes to create, documenting each one with a class diagram. You may need to refine your use cases, and produce other diagrams to document the behavior of your objects.

Use Cases and Use-Case Diagrams A use-case diagram is produced during the requirements phase, depicting the main use cases for the system, as we saw in Chapter 6 where use cases were introduced. Each individual use case is documented to some extent.

Figure 7.15 shows a use-case diagram for a course registration system. It depicts two actors, the Student and the Registrar. Students participate in two use cases, Register for section and Choose section. The registrar participates in four cases, Register for section, Add course, Add section, and Add student.

Each individual use case needs to be documented further. During the requirements phase, essential use cases are commonly developed. During the design phase, they will

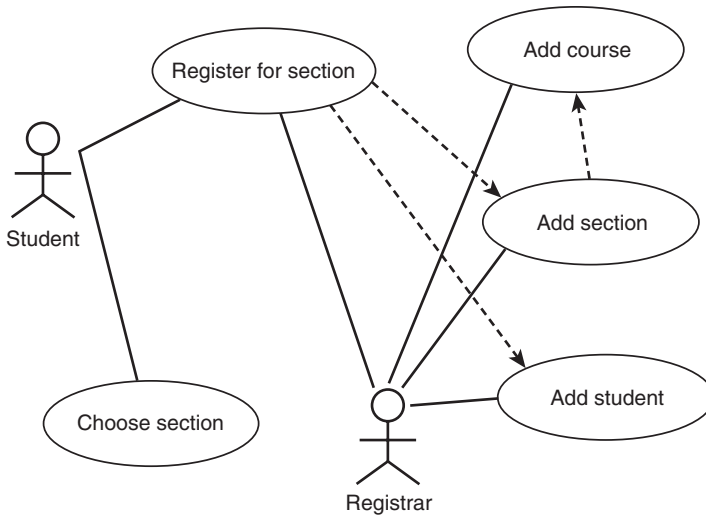


Figure 7.15 A use-case diagram for a course registration system.

need to be refined into system use cases. Essential use cases provide less detail and do not provide any details about the system; they mainly describe what the actor is supposed to do and what it tries to achieve. System use cases refine the essential case, adding detailed information about how the system achieves those goals. The details may be placed in a separate box as part of the use case diagram. The content of the box may include the following:

Essential Use-Case Documentation

Name: Register for section

Preconditions: Student is registered, section exists

Postconditions: Student will be enrolled in section (if space is available, etc.)

Basic Course of Action

1. A student wants to enroll in a section (usually after having chosen the section).
2. Student logs on to system.
3. Student specifies which sections he/she wants to register for.
4. System verifies space is available in the section, that the student has the prerequisites, and that there is no scheduling conflict.
5. If there is no problem, the student is enrolled in all the requested sections. If there is any problem, the student is notified and given the chance to modify his or her choices.

A system use case will be defined at a much finer level of detail.

Class Design and Class Diagrams One of the most important issues in detail design is designing classes as well as UML class diagrams to represent the design. In this section we will explain the basic concepts of class design and how to document them with UML. There are several basic concepts related to object orientation:

- Objects represent entities in the real world. This is similar to the concept of entity instances.
- Objects are organized into classes, which serve a similar purpose as “typing” an entity. Classes serve to group objects with similar structure and also as a template for creating new objects. Thus, a class is an abstraction of a set of similar objects. Classes are a central concept in most object-oriented languages. In our course registration example Student may be a Class, with Joe Smith as a specific student instance of that Class.
- Objects are associated with attributes, also called properties, similarly to the ER model. Each student object, such as Joe Smith, has a set of specific attribute values or data values associated with him or her. For instance, the values for address, gender, or age are associated with each specific object.
- In contrast with ER models, objects do not contain just data. They are also associated with methods, which are modules of executable code. The class, Student, may include functions or methods such as a set-birth-date method, which initializes a student’s birth-date data attribute.

An important concept in class design is encapsulation. In an object, both data and methods are included. Expressing whether both, either, or neither data/methods can be publicly accessible is an important feature. In UML, publicly accessible methods and attributes are sometimes marked with a plus sign, and private methods, which are those that are only accessible to the class but not to other classes, with a minus sign.

In UML, classes are represented by rectangles divided into three areas: (1) name of the class, (2) the attributes, and (3) the methods. When you implement the class, it is common to never make an attribute public, but rather to create accessor methods (getX, setX); it is also common practice to not show those on the diagram but to have a convention about which ones get created.

In a UML diagram, a Student class with two attributes and two methods would be represented as shown in **Figure 7.16**.

Another important factor is that of an association between two objects, a concept similar to an ER relationship, with two important differences. Associations are always binary, and they cannot have attributes of their own. In UML, associations are shown by

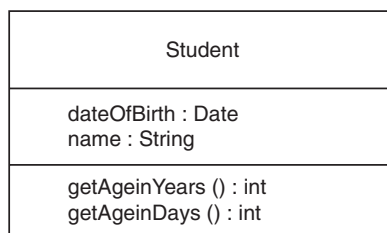


Figure 7.16 A UML class diagram for class Student.

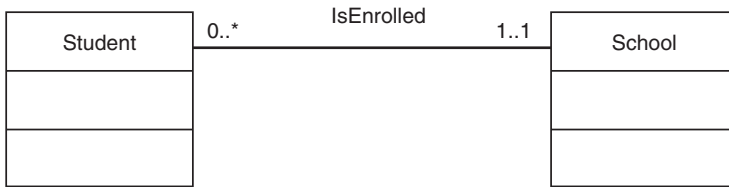


Figure 7.17 A UML representation of association.

lines among classes and associations may or may not be available from both sides. We call this property *navigability*; if the association is navigable from both sides, we call it *bidirectional*; if it is not, we call it *unidirectional*.

Figure 7.17 is a simplified class diagram that shows a student being associated with a school with the *IsEnrolled* association. The association is bidirectional—that is, it can be navigated from both sides. We also show the allowed cardinalities. A student has to be enrolled in only one school. This would be the case with elementary schools, for example. A school can have zero or more students.

A special kind of association is that of aggregation, which corresponds with the “part-of” association. Notice this may be a real-world part-of (say an engine is part of a car) or one that only makes sense in the computer (an address is part of a student). A particularly strong version of aggregation is composition, in which the subordinate object cannot participate in any other association and is the sole responsibility of the containing class; basically, the contained class is just like an attribute of the containing class. In UML, aggregation is represented as an association with a diamond, and composition has the diamond drawn in black.

For example, assume we are modeling students and we want to represent their addresses as complex objects rather than just as one string. We can view the address as being part of the student object. If we know we will not share the address objects (that is, even if two real-world students live in the same place, in our program their corresponding objects will be assigned different address objects with the same attributes rather than the same address object), then we can represent this as composition, as in **Figure 7.18**.

Another central concept to OO design is class inheritance. When a class inherits from another class, it automatically gets all its attributes and methods. If class A inherits from class B, we call class A the superclass and class B the subclass. Although

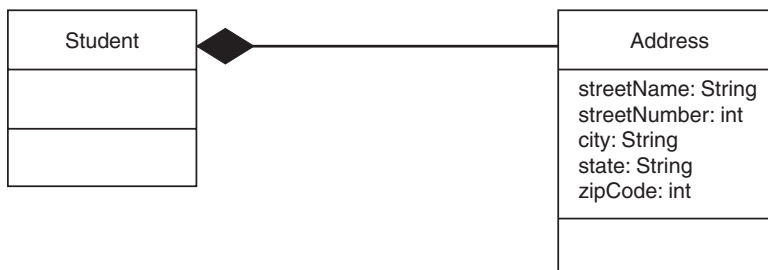


Figure 7.18 UML representation of composition.

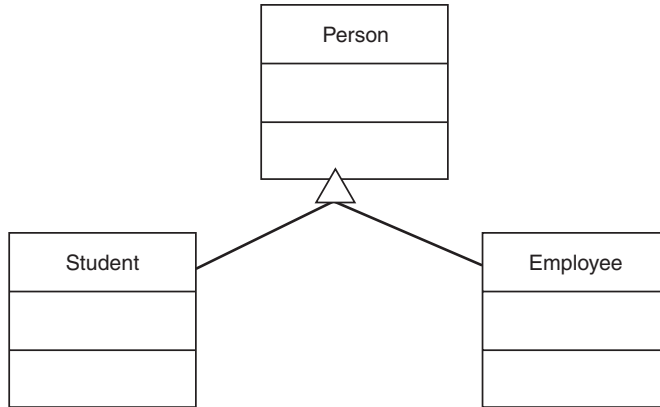


Figure 7.19 UML representation of inheritance.

the subclass can override any method it inherits from the superclass, changing its behavior, the intent is to mostly add additional methods. **Figure 7.19** shows a simplified UML class diagram illustrating inheritance. We have a *Person* class, with two subclasses, *Student* and *Employee*. Inheritance relationship is an “is-a” relationship between the subclass and the superclass. Thus, we would like to preserve as much of the superclass as possible during inheritance.

When preparing a design, note that in this example we can discover inheritance relationships through generalization or specialization. In generalization, we discover the *Student* and *Employee* classes, and later realize they share some characteristics and decide to create a common class, *Person*. In specialization, we first discover the *Person* class and later realize there are two special subtypes, *Student* and *Employee*. But in the diagrams only the inheritance relationship is shown. The diagram does not show how we came to discover this relationship. Generalization is a design technique closely related to abstraction where we simplify the design by keeping only the essentials and delaying the considerations of detail until a later time.

We should only create subclasses when there is a need to incorporate additional behavior or attributes. For example, although we may be tempted to create four additional subclasses for *Student*—namely *Freshman*, *Sophomore*, *Junior*, and *Senior*—it is probably unnecessary, as just adding a new property will allow us to discriminate among those, and there are no differences in the behavior or data that we model.

State Modeling In many cases objects can be in different states, and it is important to model those states and how they are allowed to change. This information is represented

State transition diagram A diagram representing information concerning the states of an object and the allowed state transitions.

with **state transition diagrams**. For example, in our student domain, we start considering students when they are accepted into the university. After they enroll in their first class, they become active students. If they fail to enroll for a certain number of semesters, they become inactive students. Students may be expelled or graduate and become alumni. **Figure 7.20** shows a state transition diagram representing this situation.

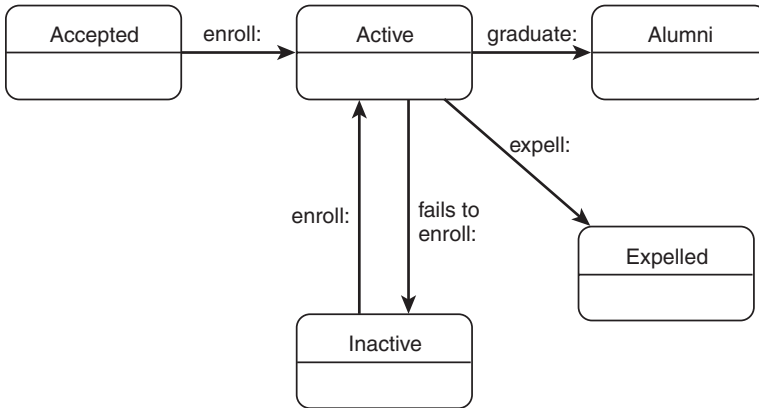


Figure 7.20 A state transition diagram.

The event-driven system mentioned earlier may be readily modeled with a state transition diagram. As external events occur, the system reacts to the events and changes states. Even though a state transition diagram is part of UML, it has been in use for modeling the system states by early computer scientists, automata theorists, and software engineers for many years.

Interactions Among Classes Designing classes and their relationships provides only the static structure of the design. The interactions among the classes to collectively accomplish some task also need to be designed. In UML these interactions are usually illustrated through **UML sequence or communication diagrams**. Communication diagrams were called collaboration diagrams in UML 1, and are still called collaboration diagrams by many authors.

UML sequence diagrams Diagrams that illustrate the flow of messages from one object to another and the sequence in which those messages are processed.

The arrows in **Figure 7.21** illustrate the flow of messages, starting from the top and flowing from left to right. The returning messages are shown with dashed lines.

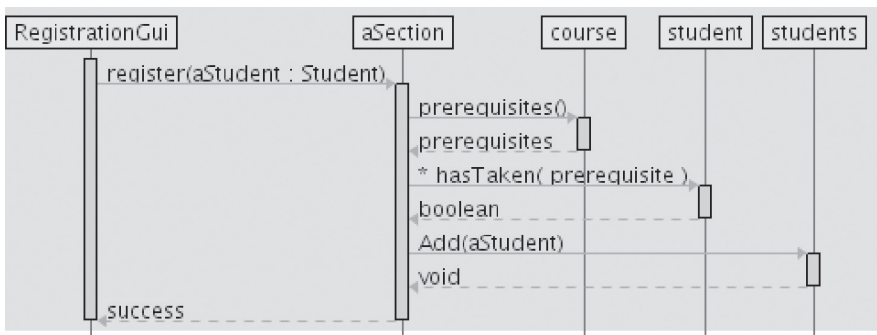


Figure 7.21 A UML sequence diagram.

7.3.4 User-Interface Design

The user interface (UI) is the part of the software most visible to the user and is one of the most important to get right. In many situations a prototype of the user interface is developed as part of the requirements analysis, as explained in Chapter 6. This prototype serves as a validation test. The client or user will confirm the prototype as the correct system or point out the defect(s). This activity also enables early testing. A big problem with user-interface design is that it is very different from programming and is very difficult for many software engineers. UI design is based on psychology, cognitive science, aesthetics, and art. Most software engineers are not well trained in any of these areas. Also, most software engineers are not typical users of software. Many software engineers tend to be very familiar with computing systems and tend to favor certain types of thinking that are not shared by the population at large. As assumed in the popular stereotype, many software engineers are “geeks” who use computers in different ways than do “ordinary” people. That makes designing good UI for the general population a difficult task.

It is better in most circumstances to leave UI design to specialists with training and skills more appropriate to this task. However, it is still important for software engineers to have a basic understanding of the issues. In many cases, it is not possible to have a separate person do the UI design, and it falls upon software developers or systems analysts to do the job. There are two main issues with user-interface design:

- The flow of interactions with the program
- The “look and feel” of the interface

The looks are not as important as the flow. We can easily design bad user interfaces and make them look pretty. Interaction design deals with the flow of interactions with the program.

Flow of Interactions in the Interface The user of a system has specific goals to be achieved in the system. These goals are directly associated with the use cases and the

sequence diagrams designed for the system. Chapter 6, “Requirements Engineering,” shows a Shipping clerk needing to process a shipping item list and create the shipping labels in Figure 6.5 Use-case notation in UML. Figure 7.15 shows a use-case diagram for a course registration system. Consider the actor:Student needing to choose and register for the section.

These are the goals of the actor:Student in their usage of the system. Figure 7.21 shows a UML sequence diagram for the detail design of registration for the system. We can see the inner design of the solution for register(aStudent: Student).

The possible registration screens are prototyped in **low fidelity** (hand drawn) or in **high fidelity** (done with a variety of screen design software like Visual Basic). **Figure 7.22** shows an example of low-fidelity prototypes for the course registration. The “look and feel” of the interface is explored with these hand-drawn screens.

Def Low fidelity prototype is a simple mockup sketch of the target product.

Def High fidelity prototype is a detailed mockup resembling and behaving close to the final product.

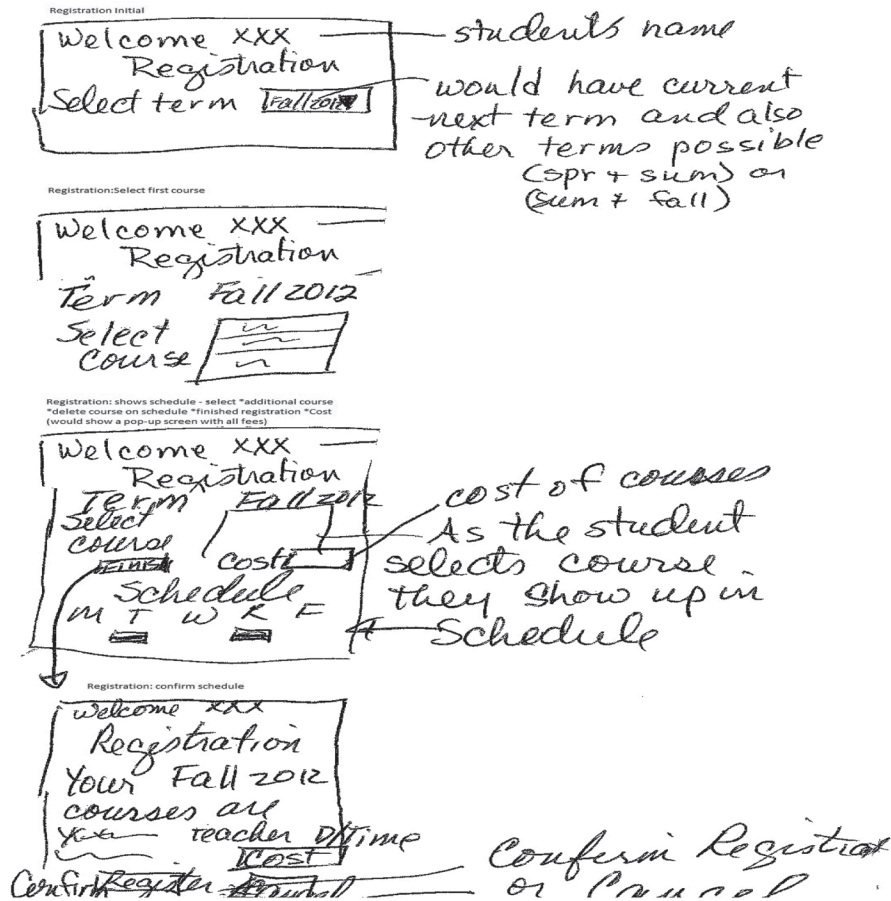


Figure 7.22 Low-fidelity prototypes of registration screens.

The low-fidelity prototype activity suggested the following four screens:

1. Registration: Initial—Select term
2. Registration: Term—Select first course
3. Registration: Desired schedule—Select additional course/delete course on schedule/finish registration/cost (would show a pop-up screen with all fees)
4. Registration: Confirm schedule

Figures 7.23 to 7.25 are the high-fidelity prototypes developed using Visual Basic. The look and feel of the interface continues to develop.

The possible registration screens developed are considered with each of the use cases for the system. The flows of interactions in the interface are shown to the client or user

Welcome UserName

Registration

School term to register SPR 2013 ▼

Help Cancel

Figure 7.23 High-fidelity prototype of registration: Initial screen.

Welcome aStudent

Registration

Desired School term to register - Spring 2012

Select course to add ALL Courses ▼

Add Course Cancel Help

Figure 7.24 High-fidelity prototype of registration: Term—Select first course.

Welcome aStudent

Registration

Desired School term to register - Spring 2012

Desired Schedule:
SWE 2313 Intro to Software Engineering Delete course

Add another course Confirm Schedule Cancel Help

Figure 7.25 High-fidelity prototype of registration: Desired schedule.

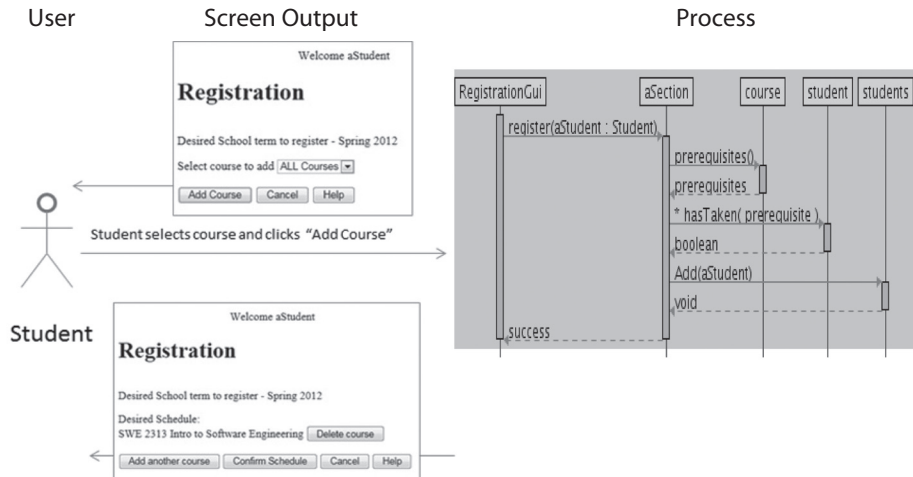


Figure 7.26 Flow of interaction.

for approval. **Figure 7.26** shows navigation of the possible screens for the Student user to select his or her initial course and then decide to add more courses, confirm schedule, and so on. Note the three columns in Figure 7.26—the User (student) on the left, screen outputs and user inputs in the center, and the inner system process (the sequence diagram for this use case) on the right.

The flow of all possible interactions—which include the user input, the screens, and the process—is considered and added as seen in **Figure 7.27**. Considerations for all

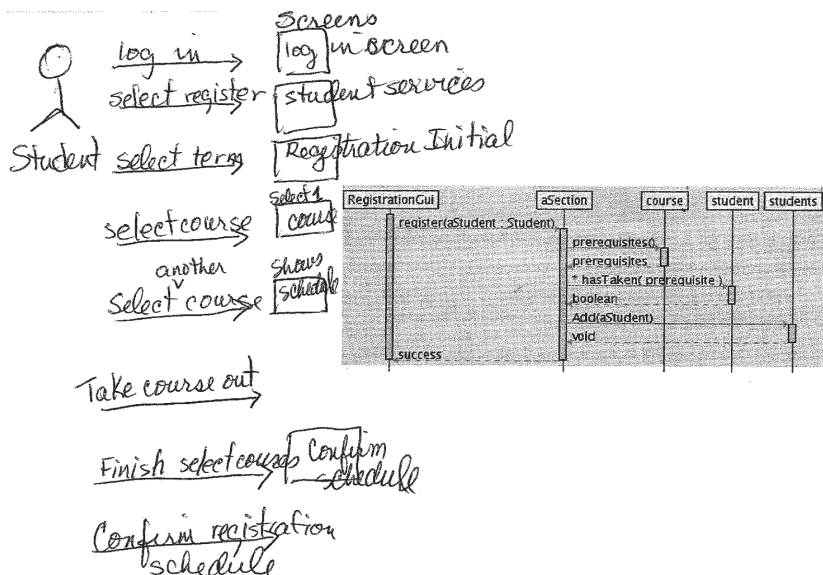


Figure 7.27 User input; screen output; process.

possible user expectations including the instructions, directions, feedback, confirmations, and help are taken into account in the development of the user-interface design.

Cognitive Models Humans think in specific stages. Norman (1988) studied the psychology of everyday actions and developed a model with seven different stages. Users will (1) form a goal; (2) form an intention; (3) specify an action; and (4) execute the action. After the user executes the action, the feedback from the system is critical for the user's understanding of the system. The user will (5) perceive the system state (feedback); (6) interpret the feedback; and (7) evaluate. If, at the last stage, the users evaluates the mode as "intuitive," they will continue with the next cycle toward their goal.

The GOMS (goals, operators, methods, and selection) model is a classical model of user interaction that involves identifying, for a particular kind of users, their goals, the basic operators your program provides, the methods, that is the sequence of operations your user can use to achieve its goals, and Selection rules, which specify which Methods to apply for achieving a particular goal when several are available. See Card, Moran, and Newell in the Suggested Reading section for more on the GOMS model.

When designing a user-centered task-oriented system you need to know the goals of your users, study their expectations about the actions in the goal, and provide the appropriate operators and methods, making sure selection rules are clear. These goals should roughly map into the use cases or scenarios. Feedback to each of the actions within the user's goals is vital to successful usage of the system. For example, a system with a button that does not give any visibility change when pressing the button- will have a hesitation in the user's usage of the button.

In the interface, every button and menu option is an operator, and methods will roughly correspond to coarser-grained UI elements, such as dialogs or wizards. The designer needs to provide all the required operators and methods, because creating new ones is more difficult and normal users will not be capable of combining them in meaningful ways even though there are macros and similar facilities.

Other Issues

- *Different kinds of users:* Often the interfaces that appeal to one kind of user are not appropriate for others. It is good practice to provide alternatives within the interface; for example, most GUI programs also provide for keyboard shortcuts. Many times features that make an interface easier to learn or use for the casual user will get in the way of expert users.
- *Heuristics for good user interface design:* There are many good heuristics for user interface design. The main heuristic is consistency throughout your program and with your platform and similar programs. Other heuristics include putting the user in control, reducing user's memory load, and making the system status visible.
- *User interface guidelines:* Almost all GUI platforms, such as Apple's operating systems, Microsoft Windows, GNOME, and KDE, provide user interface guidelines. These are much more detailed than the heuristics and provide information about which controls to use, what menu items have to exist, and many other detailed issues. Following the platform's user interface guidelines will make all programs more consistent with each other.

- *Multicultural issues:* Creating a program that will be usable for people in many different countries and cultures is a great challenge. The creation of a program version for a specific group of users based on language or country is called *localization*. Colors and icons have different meanings for different cultures. Translating messages from one language to another is a difficult task. In many cases a localized version needs to be created not just for each language but also for each country that uses that language, as the words and expressions used change from one to another. There are many programming libraries available for dealing with some of the internationalization issues. Internationalization and localization issues will only become more relevant as world globalization continues. Designing a program that is usable by people of many cultures and countries will open up many markets and may be required for some systems.
- *Metaphors:* Many user interfaces are based on denotation to known objects. Most file management and operating system GUIs are based on the desktop metaphor. Most word processing programs try to utilize a paper document metaphor. An appropriate metaphor can facilitate program learning and transfer of real-world skills. However, in some situations, the real-world system is different from the metaphor and the users need to be made aware of those differences.
- *Multiplatform software:* Software engineers use different software platforms. Most users are very attached to their platform and will not change it just to run other programs. In many cases, the newly developed software needs to run in several different platforms and must integrate well with each one. The main problem with multiplatform software is consistency. A decision must be made about whether the software must be consistent across all platforms or whether, instead, it must integrate fully with each platform and follow that specific platform's guidelines.
- *Accessibility:* The software should be made as accessible as possible in order to be used by as many people as possible. Some people cannot see well or distinguish between certain colors, and some cannot operate a normal keyboard or mouse.
- *Multimedia interfaces:* Graphics and text are not the only ways to provide information. It is currently possible to use sound and, in some cases, tactile feedback to convey information. There is even a device to produce smells. How to take advantage of these output devices to make a better user interface will be a challenge in the years to come.

7.3.5 Some Further Design Concerns

Most commercial applications have three main components: user interface, application logic, and data. In the case of web applications, user interfaces are displayed via a browser, the data is usually stored in a relational database, and the application logic is written in either a programming or script language. This is the MVC architectural style. Earlier we mentioned that design decisions may even include implementation concerns. When object-oriented (OO) design is chosen, the entities in the graphical user interface must be mapped to the object defined in the OO programming language. Similarly, the object in the OO programming language needs to be mapped to the relational database

table. The constructs in the user interface, in the programming objects, and in the relational database tables are different and do not necessarily match.

Here we will briefly discuss one mapping problem called *object-relational impedance mismatch*. Several issues arise when a relational database system is utilized by an OO programming style design. Difficulties are encountered when classes are mapped onto relational database tables and class attributes are mapped onto table columns.

One issue is that objects have an identity, and rows in a relational database are considered just values. While this is important, it can prove difficult at times. However, the issue can be ameliorated by adding a specific field—an object id—to each row in a relational database. One would still need to keep track of whether there is more than one copy of the same database object in memory.

Another issue is the difference in the support of data typing. The relational model prohibits the by-reference or pointer type, but an OO language supports the by-reference type. The way string data type and collations are supported also differ between relational database systems and OO programming languages.

Additional problems stem from the fact that relational databases deal with sets of rows, while objects reference each other in complex structures. When bringing an object from the database into main memory, it is not clear whether to bring the other related objects into memory as well. In extreme situations, one can conceptually require that the entire database be brought into memory.

While one may argue that these are implementation issues, they certainly need to be considered at detail design time. See Heinckiens (1998) in the Suggested Readings section for further discussions on database applications and impedance mismatch.

In the next chapter we will discuss some ways of evaluating different designs, the parameters and metrics used for evaluation, and some guidance for good design.

7.4 HTML-Script-SQL Design Example

In this section we will delve deeper into some details of designing a web application though a simple example utilizing HTML, PHP, and SQL. Note that this specific combination of tools may change, and one needs to make sure that the chosen tools will interact together.

A software project that follows the Model-View-Controller (MVC) architectural style (see Figure 7.4) can be done with three main parts for the detail design:

1. A HyperTextMarkup Language (HTML) interface design used for portraying the “view” and the information flow for the application
2. A scripting language as the engine of the system (we will use PHP) serving as the “controller” for the application
3. An SQL database that stores the information and acts as the “model” for the application

The Web-based database application begins with an interactive interface composed of HTML pages. A study of the structure of an HTML document shows that it really has many parts. Formatting tags, hyperlinks, lists, tables, frames, and Cascading Style Sheets (CSS) are the essential materials needed to create the interactive interface. HTML forms

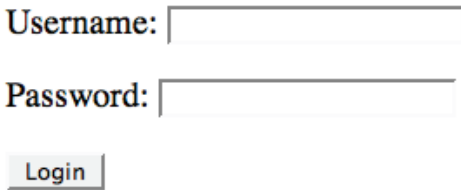
Sample HTML	Visual result (possible)
<pre><form method="GET" action="something.php"> <p> Username: <input type="text" name="username"> </p> <p> Password: <input type="password" name="password"> </p> <input type="submit" value="Login"> </form></pre>	

Figure 7.28 Sample HTML with visual result displayed using Firefox browser on an Apple Mac Pro machine.

allow the web pages to have a wide variety of input fields for the user to input information, selections, and so on. A simple example using an HTML with PHP method is shown in **Figure 7.28**. The visual results shown used Firefox browser on an Apple Mac Pro machine.

Using the GET method of PHP, data in the HTML form is appended to the URL in the action field. Thus the data submitted is visible. We would note that this represents a security risk, but on the other hand it allows one to bookmark the page with the submitted data, or even write an external link to it. By copying the syntax, you can encode data to be sent to a server page in a link, with the data coming from a database, or some other program.

Besides the URL, data sent through a GET request has some size limitation, so when sending big amounts of data you need to use the POST method of PHP.

HTML pages are static documents, so we need a programming language that can generate HTML pages. For this example we use PHP as that programming language. PHP is a scripting and dynamically typed language designed for web development. A good source for information about PHP is www.php.net. A study of variables, printing, strings, arrays, control structures, loops, and functions is the first step if one is not familiar with the PHP scripting language. PHP code can be embedded inside an HTML file and is saved as a .php extension rather than a .html extension. The web server executes the PHP code and embeds its output in the HTML file that is sent to the browser.

The next step to consider in the application design is the database model and database access. For this example, the design and creation of the database would be established. PHP would be used to send SQL commands to the database management system (DBMS), specifically to PostgreSQL. PHP provides an abstract layer for accessing many DBMSs through the same interface (called PEAR DB), but for simplicity we will

cover only the functions that are PostgreSQL specific (if you were to switch to another DBMS you'd probably need to change the first few characters of each function's name). Note that if you use something such as PEAR DB for general interface, there is probably a performance penalty.

The PHP functions we will use for accessing the database include those in **Table 7.1**.

Table 7.1 Sample PHP-DB Access Functionalities

Function	Purpose
<code>pg_connect</code>	Establishes a connection to the database and returns a handle to it.
<code>pg_query</code> , <code>pg_query_params</code>	Executes a query and returns a handle to the result set; notice the query can be an INSERT, UPDATE, or DELETE, in addition to a SELECT. <code>pg_query_params</code> is used for parametrized queries; that is, those for which the final form is obtained by interpolating strings or otherwise incorporating variables into a query string.
<code>pg_numrows</code>	Returns the number of rows in a result set.
<code>pg_fetch object</code>	Returns an object representing a row in a result set.

An example for retrieving all rows from the relational table named "student" in the database named "ok" would have the following lines of PHP code (explanations given here):

```
$conn = pg_connect("host=localhost user=namedbname=ok password=abc");
```

The above line establishes the connection and stores (a handle to) that connection in the `$conn` variable.

```
$query_str="Select * FROM student";
```

This line just initializes a string variable called `$query_str`. Notice that the value of that string variable is a SQL statement, which will be passed to the PostgreSQL database.

```
$res=pg_query($conn, $query_str);
```

The above line actually sends the query to the DBMS (PostgreSQL in this case). It uses the connection already established (`$conn`) and the query stored in `$query_str`, and stores (a handle to) the rows returned by the DBMS as a result of the query in `$str`.

The application will be conceptually organized into pages which serve as screen for the application. It is a good idea to keep each page in its own file. To implement a piece of functionality, one will usually need two components:

1. An HTML form.
2. A PHP page called from that form that uses the input provided by the form. The input is used to query, obtain, save, or pass along using one or more SQL statements.

These separate pages may be related through the usage of links. Many of the pages can include links to each other, and these links may even be generated using PHP. The number of links may be dependent on the information on the database; recall that we can actually encode information in a link by adding a question mark (?) at the end, and then `name=value` pairs. If we define a menu in a frame, that menu can link to several

pages so that the user can keep track of the functionality, while the functionality being currently accessed is displayed in another frame.

Notice that each PHP page will be a separate page and that each request comes as a completely separate request. However, many times, we want to give the user the illusion that they are accessing an application and that the pages know which other pages that particular user has accessed recently. Within web applications, we call this idea of all the recent interactions of a user with a website a *session*. PHP supports keeping track of user sessions. Using session variables in PHP is very easy. For each page, we need to call the function `session_start()`, being careful that this call occurs before any output.

Common web application exercises that have request for accounts, login form, and then process orders can be created using this model. The detail design of each part of the MVC architectural style involve different strengths of a software engineering team. The database experts design the SQL database, the programming experts tackle the PHP code, and the usability experts on the team focus on the user interactions with the HTML pages.

7.5 Summary

In this chapter we have discussed most of the issues with design. We discussed high-level, or architectural, design. We then discussed detailed design, and techniques for functional decomposition, relational database design, object-oriented design, and user interface design.

The design of your software is one of the most important issues of its development. Whether you do a formal, complete design or an informal one, it is always advantageous to think about how you are going to achieve your goals before doing much programming.

In the next chapter we will discuss some ways of evaluating different designs, the parameters and metrics used for evaluation, and some guidance for good design.

7.6 Review Questions

1. Explain the role of requirements in architectural design. Explain the role of requirements in detail design.
2. What does aggregation mean in OO? Give an example.
3. When we employ the technique of generalization in design, what are we doing, and which part of OO design is closely related to this concept?
4. List two differences between the state transition diagram and the sequence diagram.
5. Describe three different views used in architectural design.
6. What is the difference between data modeling and logical database design?
7. Describe the difference between low-fidelity and high-fidelity prototyping in the design of the interface. Choose one and give the reasons why you would show the client this prototype.

8. Explain the three columns in Figure 7.26 labeled User, Screen Output, and Process with regard to design.
9. Choose one of the cognitive models and explain how the model impacts the design of the user interface.
10. Visit a website that is from a different country or culture. Give an example of a multicultural issue you found in the site. Explain how you would propose to re-design, taking into consideration the issue found.

7.7 Exercises

1. Write a command-line program for converting different units of measurement. Begin your program by converting kilograms and pounds and then yards and meters. Discuss your user interface with other students.
2. Write a GUI program for converting different units of measurement. Begin your program by converting kilograms and pounds and then yards and meters. Discuss your user interface with other students.
3. Find a person who speaks your language but comes from a different country or region, and discuss how the vocabulary and expressions you both use are different. On what kinds of words do you find the most differences?
4. For each one of the architectural styles mentioned in this chapter, find one example of a software system that uses it (not mentioned in the chapter).
5. Consider the case of a software system designed to keep track of team rosters and scheduled games for a sports league. Create a UML class diagram representing all the domain classes and a sequence diagram depicting one of the main interactions among those classes.
6. Consider the case of a software system designed to keep track of team rosters and scheduled games for a sports league. Create an ER diagram for this situation and convert the diagram into a relational schema.
7. Consider the case of a software system designed to keep track of team rosters and scheduled games for a sports league. Define the main functionality of the system and create a module decomposition diagram for it.

7.8 Suggested Readings

S. W. Ambler, *The Object Primer: The Application Developer's Guide to Object Orientation*, 2nd ed. (New York: Cambridge University Press, 2001).

L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 2nd ed. (Reading, MA: Addison-Wesley, 2003).

G. Booch, *Object-Oriented Analysis and Design with Applications*, 2nd ed. (Reading, MA: Addison-Wesley, 1994).

S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction* (Mahwah, NJ: Lawrence Erlbaum, 1983).

- P. Chen, "The Entity-Relationship Model—Towards a Unified View of Data," *ACM Transactions on Database Systems*, 1 (March 1976): 9–36.
- E. F. Codd, "A Relational Model of Data for Large Shared Data Banks," *Communications of ACM* 13, no. 6 (June 1970): 377–387.
- C. J. Date, *An Introduction to Database Systems*, 8th ed. (Reading, MA: Addison-Wesley, 2003).
- M. Fowler and K. Scott, *UML Distilled*, 2nd ed. (Reading, MA: Addison-Wesley, 1999).
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Reading, MA: Addison-Wesley, 1995).
- P. M. Heinckens, *Building Scalable Database Applications: Object Oriented Design, Architecture, and Implementation* (Reading, MA: Addison-Wesley, 1998).
- D. Hix and H. R. Hartson, *Developing User Interfaces: Ensuring Usability Through Product and Process* (New York: Wiley, 1993).
- P. Kruchten, "Architectural Blueprints—The 4+1 View Model of Software Architecture," *IEEE Software* (November 1995).
- R. Malveau and T. Mowbray, *Software Architecture Bootcamp* (Upper Saddle River, NJ: Prentice Hall, 2000).
- D. A. Norman, *The Design of Everyday Things* (New York: Doubleday, 1988).
- M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline* (Upper Saddle River, NJ: Prentice Hall, 1996).
- B. Shneiderman and C. Plaisant, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, 4th ed. (Reading, MA: Addison-Wesley, 2005).
- A. Silberschatz, H. F. Korth, and S. Sudarshan, *Database System Concepts*, 4th ed. (New York: McGraw Hill, 2002).
- C. Szyperski, D. Gruntz, and S. Murer, *Component Software—Beyond Object-Oriented Programming* (New York: Addison-Wesley/ACM Press, 2002).
- R. N. Taylor, N. Medvidovic, and E. M. Dashofy, *Software Architecture: Foundations, Theory and Practice* (Hoboken, NJ: John Wiley & Sons, 2009).

