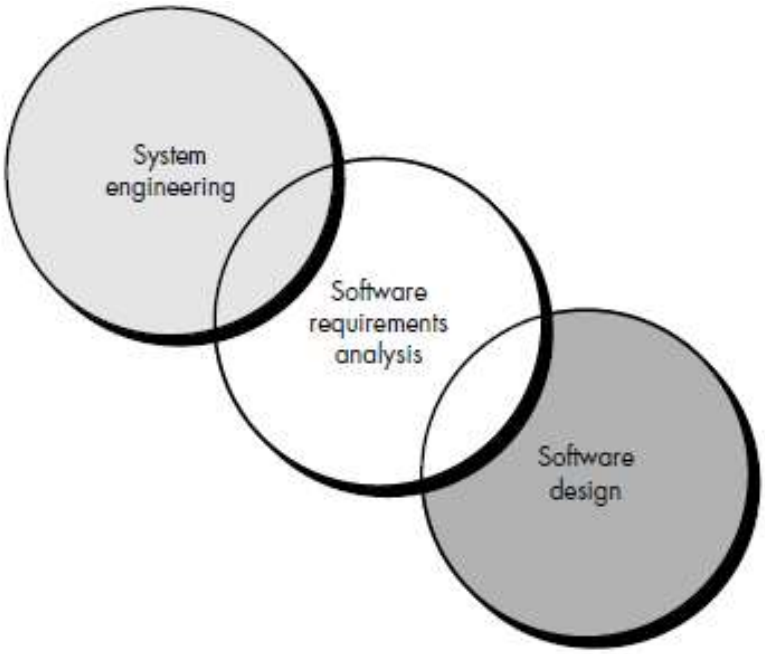
MODULE 3

REQUIREMENTS ENGINEERING

REQUIREMENT ENGINEERING TASKS-INITIATING THE REQUIREMENTS ENGINEERING PROCESS



The outcome of the system engineering process is the specification of a computer basedsystem or product at the different levels .
But the challenge facing system engineers (and software engineers) is profound: How can we ensure that we have specified a system that properly meets the customer's needs and satisfies the customer's expectations? There is no foolproof answer to this difficult question, but a solid requirements engineering process is the best solution we currently have.
Requirements engineering provides the appropriate mechanism for understanding what the customer wants, analyzing need, assessing feasibility, negotiating a reasonable solution, specifying the solution unambiguously, validating the specification, and managing the requirements as they are transformed into an operational system

The requirements engineering process can be described in five distinct steps
• requirements elicitation
• requirements analysis and negotiation
• requirements specification
• system modeling
• requirements validation
• requirements management

REQUIREMENT ELICITATION

It certainly seems simple enough—ask the customer, the users, and others what the objectives for the system or product are, what is to be accomplished, how the system or product fits into the needs of the business, and finally, how the system or product
is to be used on a day-to-day basis. But it isn't simple—it's very hard.
Christel and Kang identify a number of problems that help us understand why requirements elicitation is difficult:
• *Problems of scope*. The boundary of the system is ill-defined or the customers/users specify unnecessary technical detail that may confuse, rather
than clarify, overall system objectives.
• *Problems of understanding.* The customers/users are not completely sure of what is needed, have a poor understanding of the capabilities and limitations of their computing environment, don't have a full understanding of the problem domain, have trouble communicating needs to the system engineer, omit information that is believed to be "obvious," specify requirements that conflict with the needs of other customers/users, or specify requirements that
are ambiguous or untestable.
• *Problems of volatility.* The requirements change over time.
To help overcome these problems, system engineers must approach the requirements gathering activity in an organized manner.
Sommerville and Sawyer suggest a set of detailed guidelines for requirements elicitation, which are summarized in the following steps:
• Assess the business and technical feasibility for the proposed system.
• Identify the people who will help specify requirements and understand their organizational bias.
• Define the technical environment (e.g., computing architecture, operating system, telecommunications needs) into which the system or product will be placed.
• Identify "domain constraints" (i.e., characteristics of the business environment specific to the application domain) that limit the functionality or performance of the system or product to be built.

• Define one or more requirements elicitation methods (e.g., interviews, focus groups, team meetings).
• Solicit participation from many people so that requirements are defined from different points of view; be sure to identify the rationale for each requirement that is recorded.
• Identify ambiguous requirements as candidates for prototyping.
• Create usage scenarios to help customers/users better id identify key requirements.

The work products produced as a consequence of the requirements elicitation activity will vary depending on the size of the system or product to be built. For most systems, the work products include

• A statement of need and feasibility.
• A bounded statement of scope for the system or product.
• A list of customers, users, and other stakeholders who participated in the requirements elicitation activity.
• A description of the system's technical environment.
• A list of requirements (preferably organized by function) and the domain constraints that apply to each.
• A set of usage scenarios that provide insight into the use of the system or product under different operating conditions.
• Any prototypes developed to better define requirements.

Each of these work products is reviewed by all people who have participated in the requirements elicitation

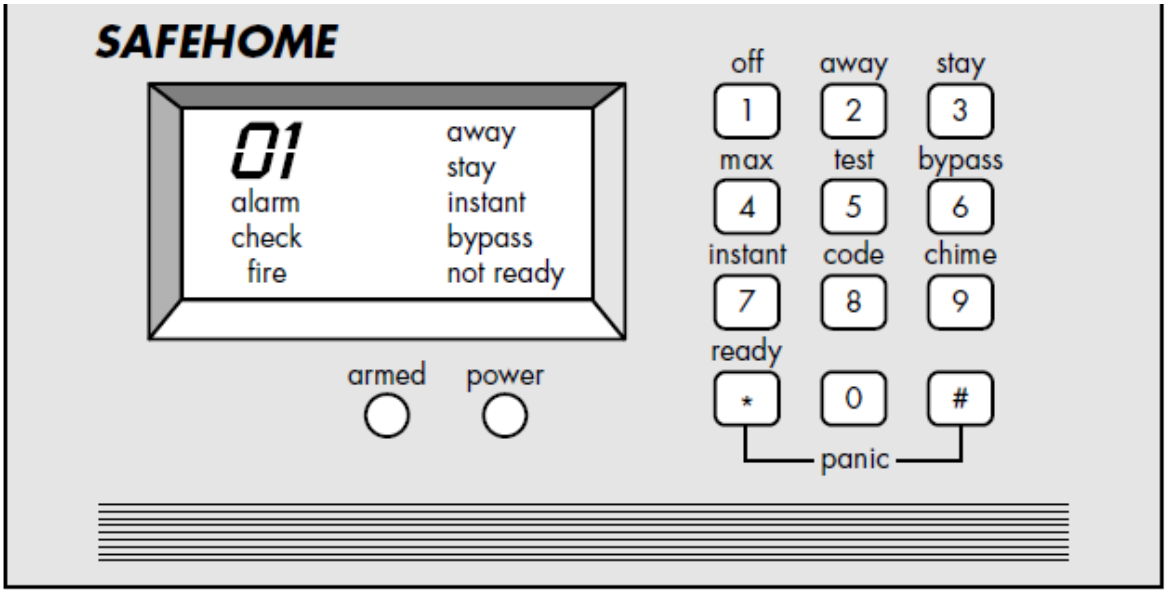
DEVELOPING USE CASES

As requirements are gathered as part of informal meetings, FAST, or QFD, the softwareengineer (analyst) can create a set of scenarios that identify a thread of usage for the system to be constructed. The scenarios, often called *use-cases* [provide a description of how the system will be used.

To create a use-case, the analyst must first identify the different types of people (or devices) that use the system or product. These *actors* actually represent roles that people (or devices) play as the system operates. Defined somewhat more formally, an actor is anything that communicates with the system or product and that is external to the system itself.

It is important to note that an actor and a user are not the same thing. A typical user may play a number of different roles when using a system, whereas an actor represents a class of external entities (often, but not always, people) that play just one role. As an example, consider a machine operator (a user) who interacts with the control computer for a manufacturing cell that contains a number of robots and numerically controlled machines. After careful review of

requirements, the software for the control computer requires four different modes (roles) for interaction: programming mode, test mode, monitoring mode, and troubleshooting mode. Therefore,
four actors can be defined: programmer, tester, monitor, and troubleshooter. In some cases, the machine operator can play all of these roles. In others, different people may play the role of each actor.
Because requirements elicitation is an evolutionary activity, not all actors are identified during the first iteration. It is possible to identify primary actors during the first iteration and secondary actors as more is learned about the system. Primary actors interact to achieve required system function and derive the intended benefit from the system. They work directly and frequently with the software. Secondary actors support the system so that primary actors can do their work.
Once actors have been identified, use-cases can be developed. The use-case describes the manner in which an actor interacts with the system. Jacobson suggests a number of questions that should be answered by the use-case:
• What main tasks or functions are performed by the actor?
• What system information will the actor acquire, produce, or change?
• Will the actor have to inform the system about changes in the external environment?
A use-case is a scenario that describes how software is to be used in a given situation.
Use-Cases
Use-cases are defined from an actor's pointof view. An actor is arole that people (users) or devices playas they interact with the software.



• What information does the actor desire from the system?
• Does the actor wish to be informed about unexpected changes?
In general, a use-case is simply a written narrative that describes the role of an actor as interaction with the system occurs.
Recalling basic *SafeHome* requirements , we can define three actors:
the homeowner (the user), sensors (devices attached to the system), and the monitoring and response subsystem (the central station that monitors *SafeHome*). For the purposes of this example, we consider only the **homeowner** actor. The homeownerinteracts with the product in a number of different ways:
• enters a password to allow all other interactions
• inquires about the status of a security zone
• inquires about the status of a sensor
• presses the panic button in an emergency
• activates/deactivates the security system
A use-case for *system activation* follows:
1. The homeowner observes a prototype of the *SafeHome* control panel to determine if the system is ready for input. If the system is not ready, the homeowner must physically close windows/doors so that the ready indicator is present. [A *not ready* indicator implies that a sensor is open; i.e., that a door or window is open.]
2. The homeowner uses the keypad to key in a four-digit password. The passwordis compared with the valid password stored in the system. If the passwordis incorrect, the control panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further

action.
3. The homeowner selects and keys in *stay* or *away* to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.
4. When activation occurs, a red alarm light can be observed by the homeowner.
Use-cases for other homeowner interactions would be developed in a similar manner.
It is important to note that each use-case must be reviewed with care. If some element of the interaction is ambiguous, it is likely that a review of the use-case willindicate a problem.
Each use-case provides an unambiguous scenario of interaction between an actor and the software. It can also be used to specify timing requirements or other constraints for the scenario. For example, in the use-case just noted, requirements indicatethat activation occurs 30 seconds after the *stay* or *away* key is hit. This information can be appended to the use-case.
Use-cases describe scenarios that will be perceived differently by different actors.
Wyder [WYD96] suggests that quality function deployment can be used to develop aweighted priority value for each use-case. To accomplish this, use-cases are evaluated from the point of view of all actors defined for the system. A priority value isassigned to each use-case (e.g., a value from 1 to 10) by each of the actors. An average priority is then computed, indicating the perceived importance of each of the usecases.
When an iterative process model is used for software engineering, the priorities can influence which system functionality is delivered first.
BUILDING YHE ANALYSIS MODEL
At a technical level, software engineering begins with a series of modeling tasks that lead to a complete specification of requirements and a comprehensive design representation for the software to be built. The *analysis model*, actually a set of models, is the first technical representation of a system. Over the years many methods have been proposed for analysis modeling. However, two now dominate. The first, *structured analysis,* is a classical modeling method .
Structured analysis is a model building activity. Applying the operational analysis principles we create and partition data, functional,and behavioral models that depict the essence of what must built.
• The products of analysis must be highly maintainable. This applies particularly to the
Target Document [software requirements specifications].

• Problems of size must be dealt with using an effective method of partitioning. The Victorian novel specification is out.
• Graphics have to be used whenever possible.
• We have to differentiate between logical [essential] and physical [implementation] considerations
At the very least, we need . . .
• Something to help us partition our requirements and document that partitioning before specification . . .
• Some means of keeping track of and evaluating interfaces . . .
• New tools to describe logic and policy, something better than narrative text . . .
There is probably no other software engineering method that has generated as much interest, been tried (and often rejected and then tried again) by as many people, provoked as much criticism, and sparked as much controversy. But the method has prospered and has gained a substantial following in the software engineering community.

THE ELEMENTS OF ANALYSIS MODEL

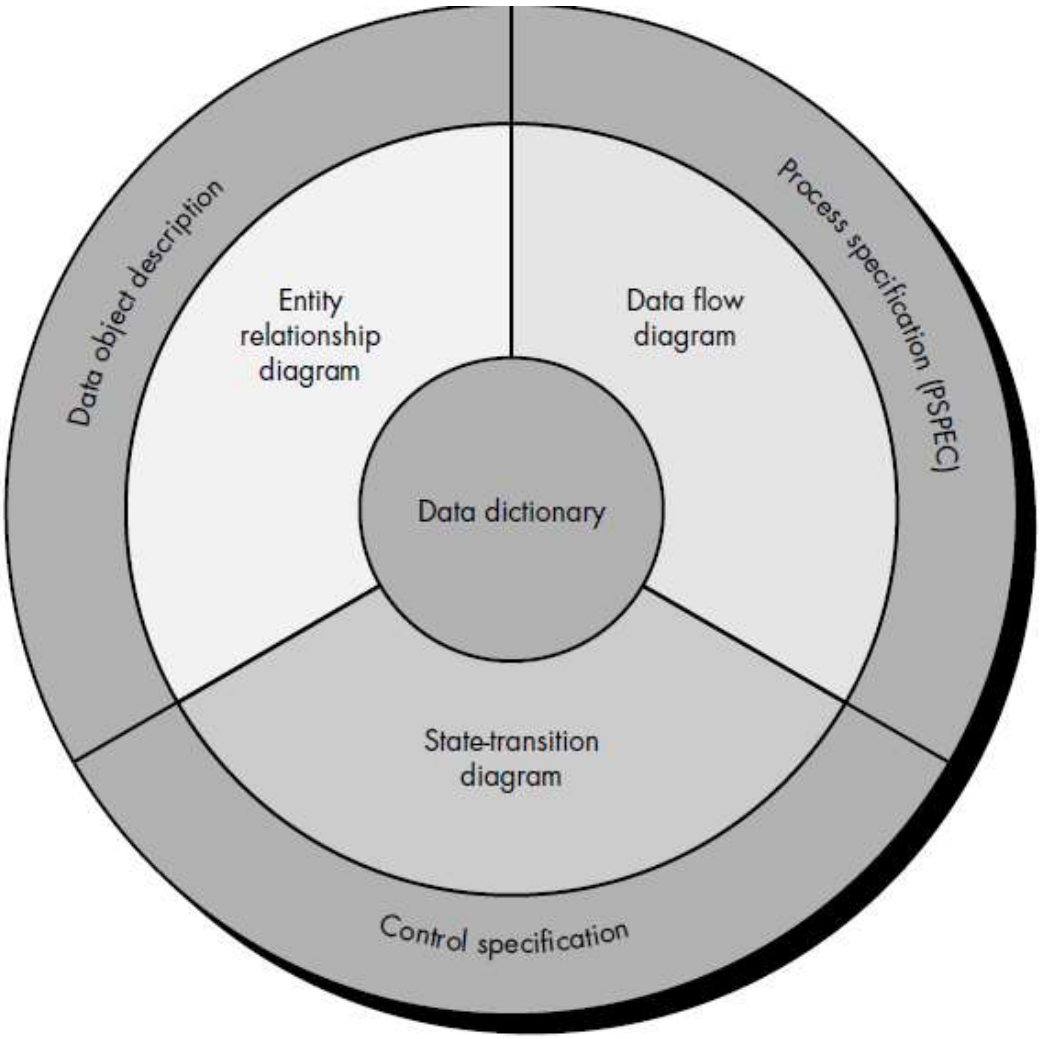
The analysis model must achieve three primary objectives: (1) to describe what the customer requires, (2) to establish a basis for the creation of a software design, and (3) to define a set of requirements that can be validated once the software is built. To accomplish these objectives, the analysis model derived during structured analysis takes the form .

At the core of the model lies the *data dictionary*—a repository that contains descriptions of all data objects consumed or produced by the software. Three different diagrams surround the the core. The *entity relation diagram* (ERD) depicts relationships between data objects. The ERD is the notation that is used to conduct the data modeling activity. The attributes of each data object noted in the ERD can be described using a data object description.

The *data flow diagram* (DFD) serves two purposes: (1) to provide an indication of how data are transformed as they move through the system and (2) to depict the functions (and subfunctions) that transform the data flow. The DFD provides additional information that is used during the analysis of the information domain and serves as a basis for the modeling of function. A description of each function presented in the DFD is contained in a *process specification* (PSPEC).

The *state transition diagram* (STD) indicates how the system behaves as a consequence of external events. To accomplish this, the STD represents the various modesof behavior (called *states*) of the system and the manner in which transitions are made from state to state. The STD serves as the basis for behavioral modeling. Additional information about the control aspects of the software is contained in the *control specification*
(CSPEC).

The analysis model encompasses each of the diagrams, specifications, descriptions,and the dictionary



of

DATA MODELING

Data modeling answers a set of specific questions that are relevant to any data processing

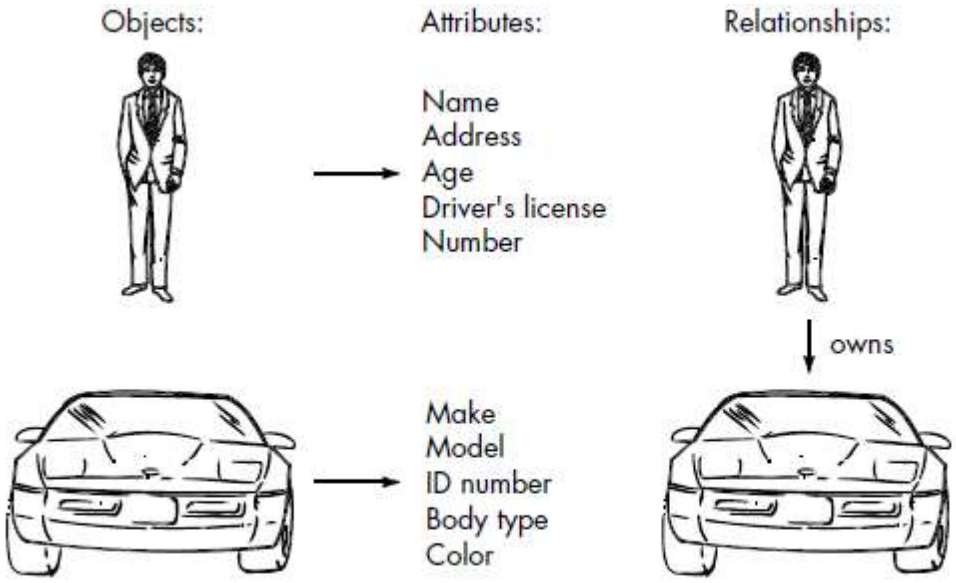
application. What are the primary data objects to be processed by the system? What is the composition of each data object and what attributes describe the object? Where do the the objects currently reside? What are the relationships between each object and other objects? What are the relationships between the objects and the processes that transform them?

To answer these questions, data modeling methods make use of the entity relationship diagram. The ERD, described in detail later in this section, enables a software engineer to identify data objects and their relationships using a graphical notation. In the context of structured analysis, the ERD defines all data that are entered, stored, transformed, and produced within an application.

The entity relationship diagram focuses solely on data (and therefore satisfies the first operational analysis principles), representing a "data network" that exists for a given system. The ERD is especially useful for applications in which data and the relationships
that govern data are complex. Unlike the data flow diagram (discussed in Section 12.4 and used to represent how data are transformed), data modeling considers data independent of the processing that transforms the data.

12.3.1 Data Objects, Attributes, and Relationships

The data model consists of three interrelated pieces of information: the data object, the attributes that describe the data object, and the relationships that connect data objects to one another.

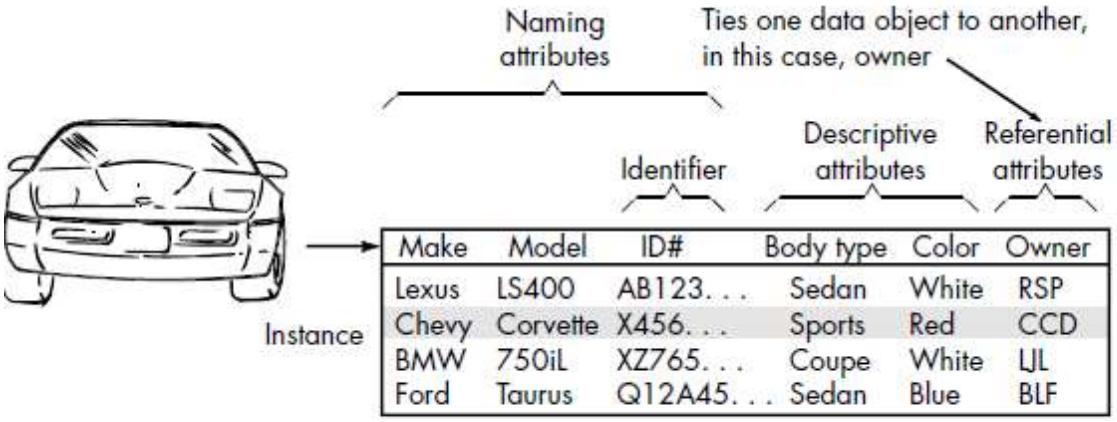


Data objects. A *data object* is a representation of almost any composite information that must be understood by software. By *composite information*, we mean something that has a number of different properties or attributes. Therefore, width (a single value) would not be a valid data object, but dimensions (incorporating height, width, and depth) could be defined as an object.

A data object can be an external entity (e.g., anything that produces or consumes information), a thing (e.g., a report or a display), an occurrence (e.g., a telephone call) or event (e.g., an alarm), a role (e.g., salesperson), an organizational unit (e.g., accounting department), a place (e.g., a warehouse), or a structure (e.g., a file). For example, a person or a car can be viewed as a data object in the sense that either can be defined in terms of a set of attributes. The data object description incorporates the data object and all of its attributes.

Data objects (represented in bold) are related to one another. For example, **person** can *own* **car**, where the relationship *own* connotes a specific "connection" between **person** and **car**. The relationships are always defined by the context of the problem that is being analyzed.

A data object encapsulates data only—there is no reference within a data object to operations that act on the data.1 Therefore, the data object can be represented as a table The headings in the table reflect attributes of the object. In this case, a car is defined in terms of make, model, ID number, body type, color and owner. The body of the table represents specific instances of the data object. For example, a Chevy Corvette is an instance of the data object **car.**



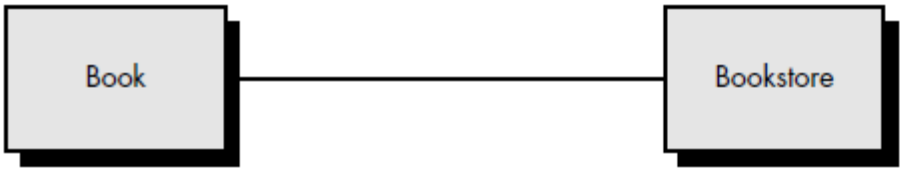
Attributes. Attributes define the properties of a data object and take on one of three different characteristics. They can be used to (1) name an instance of the data object, (2) describe the instance, or (3) make reference to another instance in another table. In addition, one or more of the attributes must be defined as an *identifier*—that is, the identifier attribute becomes a "key" when we want to find an instance of the data object. In some cases, values for the identifier(s) are unique, although this is not a requirement. Referring to the data object **car**, a reasonable identifier might be the ID number.

The set of attributes that is appropriate for a given data object is determined through an understanding of the problem context. The attributes for **car** might serve well for an application that would be used by a Department of Motor Vehicles, but these attributes would be useless for an automobile company that needs manufacturing control software. In the latter case, the attributes for **car** might also include ID number, body type and color, but many additional attributes (e.g., interior code, drive train type, trim package designator, transmission type) would have to be added to make car a meaningful object in the manufacturing control context.

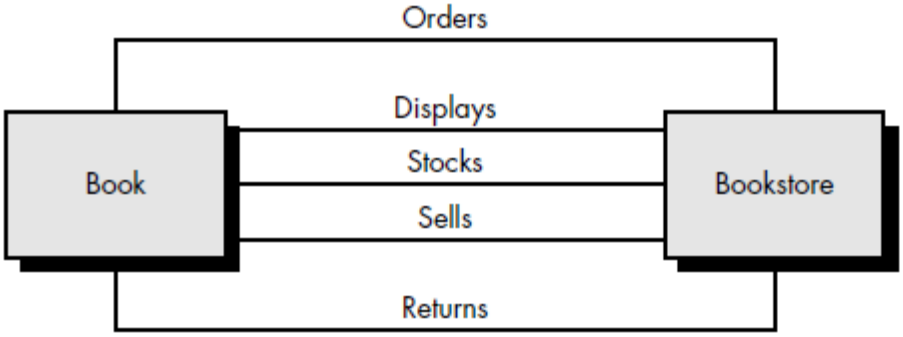
Relationships. Data objects are connected to one another in different ways. Consider two data objects, **book** and **bookstore**. These objects can be represented using the simple notation illustrated in Figure 12.4a. A connection is established between **book** and **bookstore** because the two objects are related. But what are the relationships? To determine the answer, we must understand the role of books and bookstores within the context of the software to be built. We can define a set of object/relationship pairs that define the relevant relationships. For example,

• A bookstore orders books.

• A bookstore displays books.
• A bookstore stocks books.
• A bookstore sells books.
• A bookstore returns books.



(a) A basic connection between objects



(b) Relationships between objects

The relationships *orders, displays, stocks, sells,* and *returns* define the relevant connections between **book** and **bookstore**.
It is important to note that object/relationship pairs are bidirectional. That is, they can be read in either direction. A bookstore orders books or books are ordered by a bookstore.

Cardinality and Modality

The elements of data modeling—data objects, attributes, and relationships— provide the basis for understanding the information domain of a problem. However, additional information related to these basic elements must also be understood.
We have defined a set of objects and represented the object/relationship pairs that bind them. But a simple pair that states: **object X** *relates* to **object Y** does not provide enough information for software engineering purposes. We must understand how many occurrences of **object X** are related to how many occurrences of **object Y.** This leads to a data modeling concept called *cardinality.*
Cardinality. The data model must be capable of representing the number of occurrences objects in a given relationship. Tillmann defines the *cardinality* of an object/relationship pair in the following manner:
Cardinality is the specification of the number of occurrences of one [object] that can be related to the number of occurrences of another [object]. Cardinality is usually expressed as simply 'one' or 'many.' For example, a husband can have only one wife (in most cultures),

while a parent can have many children. Taking into consideration all combinations of 'one' and 'many,' two [objects] can be related as

• One-to-one (l:l)—An occurrence of [object] 'A' can relate to one and only one occurrence of [object] 'B,' and an occurrence of 'B' can relate to only one occurrence of 'A.'

• One-to-many (l:N)—One occurrence of [object] 'A' can relate to one or many occurrences of [object] 'B,' but an occurrence of 'B' can relate to only one occurrence of 'A.'

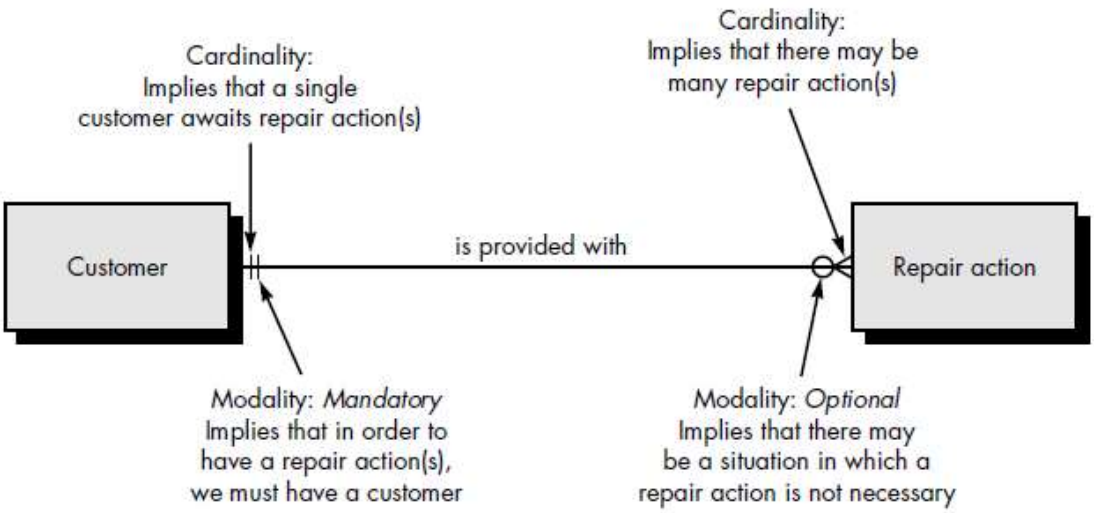
For example, a mother can have many children, but a child can have only one mother.

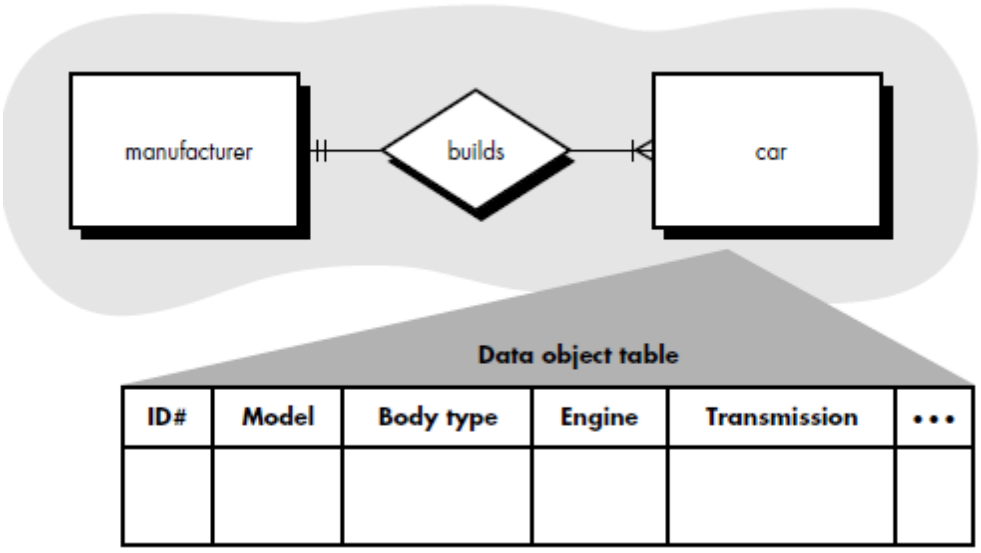
• Many-to-many (M:N)—An occurrence of [object] 'A' can relate to one or more occurrences of 'B,' while an occurrence of 'B' can relate to one or more occurrences of 'A.'

For example, an uncle can have many nephews, while a nephew can have many uncles.

Cardinality defines "the maximum number of objects that can participate in a relationship" [TIL93]. It does not, however, provide an indication of whether or not a particular data object must participate in the relationship. To specify this information, the data model adds modality to the object/relationship pair.

Modality. The *modality* of a relationship is 0 if there is no explicit need for the relationship to occur or the relationship is optional. The modality is 1 if an occurrence of the relationship is mandatory. To illustrate, consider software that is used by a local telephone company to process requests for field service. A customer indicates that there is a problem. If the problem is diagnosed as relatively simple, a single repair action occurs. However, if the problem is complex, multiple repair actions may be required. Figure 12.5 illustrates the relationship, cardinality, and modality between the data objects **customer** and **repair action.**





Referring to the figure, a one to many cardinality relationship is established. That is, a single customer can be provided with zero or many repair actions. The symbols on the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar indicates one and the three-pronged fork indicates many. Modality is indicated by the symbols that are further away from the data object rectangles. The second vertical bar on the left indicates that there must be a customer for a repair action to occur. The circle on the right indicates that there may be no repair action required for the type of problem reported by the customer.

Entity/Relationship Diagrams

The object/relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the *entity/relationship diagram*. The ERD was originally proposed by Peter Chen for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

. Data objects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality

The relationship between the data objects **car** and **manufacturer** would be represented

One manufacturer builds one or many cars. Given

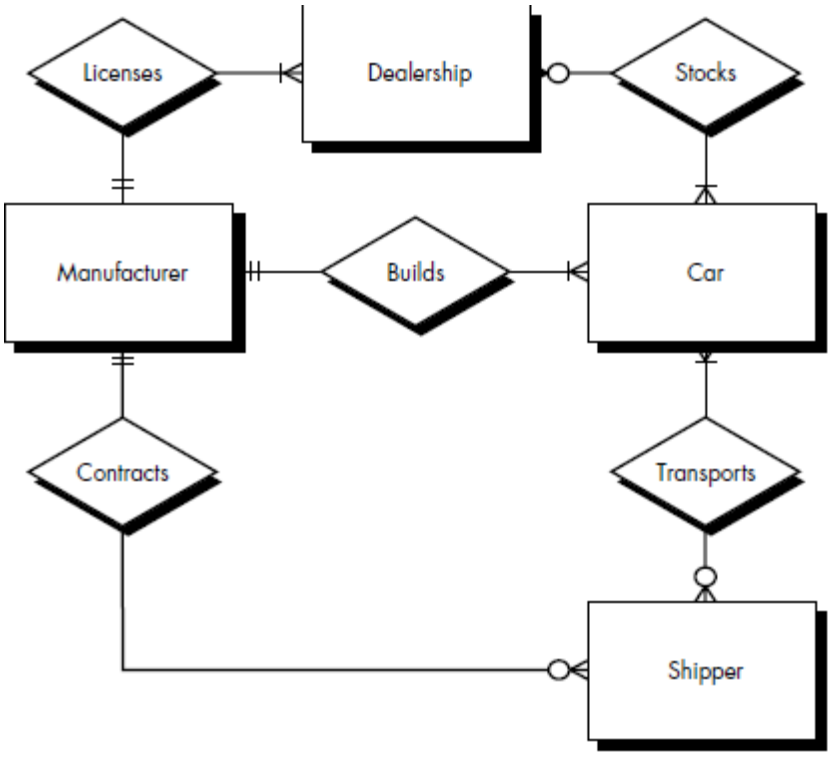
Referring to the figure, a one to many cardinality relationship is established. That is, a single customer can be provided with zero or many repair actions. The symbols on the relationship connection closest to the data object rectangles indicate cardinality. The vertical bar indicates one and the three-pronged fork indicates many. Modality is indicated by the symbols that are further away from the data object rectangles.

The second vertical bar on the left indicates that there must be a customer for a repair action to occur. The circle on the right indicates that there may be no repair action required for the type of problem reported by the customer.

The object/relationship pair is the cornerstone of the data model. These pairs can be represented graphically using the *entity/relationship diagram*. The ERD was originally proposed by Peter Chen [CHE77] for the design of relational database systems and has been extended by others. A set of primary components are identified for the ERD: data objects, attributes, relationships, and various type indicators. The primary purpose of the ERD is to represent data objects and their relationships.

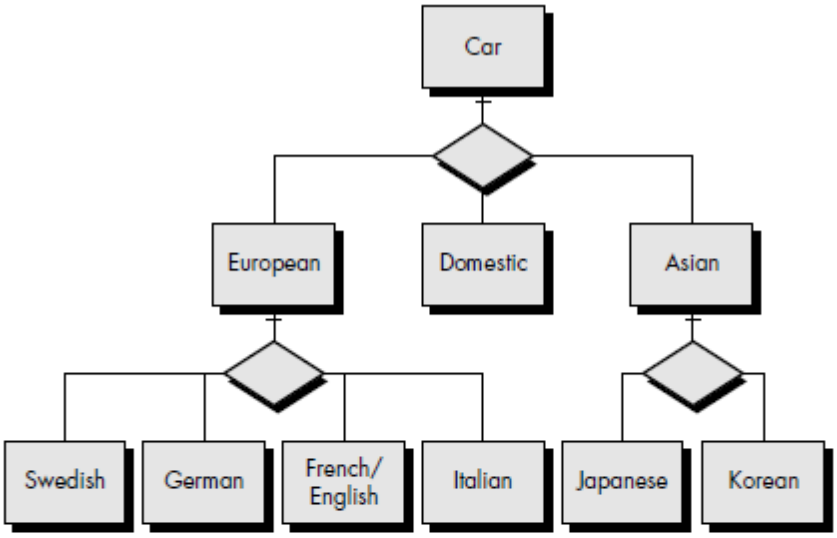
Dataobjects are represented by a labeled rectangle. Relationships are indicated with a labeled line connecting objects. In some variations of the ERD, the connecting line contains a diamond that is labeled with the relationship. Connections between data objects and relationships are established using a variety of special symbols that indicate cardinality and modality

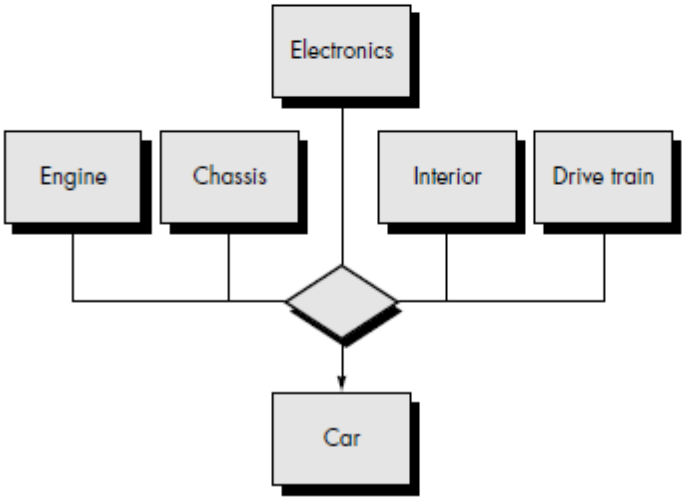
The relationship between the data objects **car** and **manufacturer** would be represented as shown in Figure . One manufacturer builds one or many cars. Given



the context implied by the ERD, the specification of the data object **car** (data object

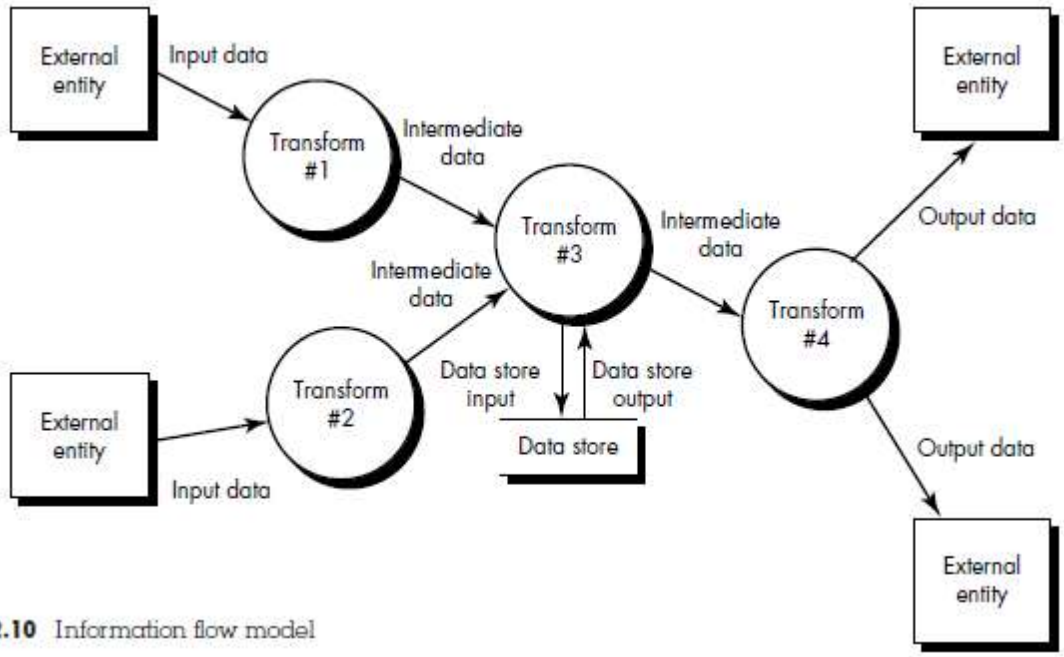
table in Figure) would be radically different from the earlier specification (Figure). By examining the symbols at the end of the connection line between objects, it can be seen that the modality of both occurrences is mandatory (the vertical lines).
Expanding the model, we represent a grossly oversimplified ERD (Figure) of the distribution element of the automobile business. New data objects, **shipper** and **dealership,** are introduced. In addition, new relationships—*transports, contracts, licenses,* and *stocks*—indicate how the data objects shown in the figure associate with one another. Tables for each of the data objects contained in the ERD would have to be developed according to the rules introduced earlier in this chapter.
In addition to the basic ERD notation introduced in Figures , the analyst can represent *data object type hierarchies.* In many instances, a data object may actually represent a class or category of information. For example, the data object **car** can be categorized as domestic, European, or Asian. The ERD notation shown in Figure represents this categorization in the form of a hierarchy [ROS85].
ERD notation also provides a mechanism that represents the associativity between objects. An *associative data object* is represented as shown in . In the figure, each of the data objects that model the individual subsystems is associated with the data object **car.**





FUNCTIONAL MODELING

Information is transformed as it flows through a computer-based system. The system accepts input in a variety of forms; applies hardware, software, and human elements to transform it; and produces output in a variety of forms. Input may be a control.



!.10 Information flow model

signal transmitted by a transducer, a series of numbers typed by a human operator, a packet of information transmitted on a network link, or a voluminous data file retrieved from secondary storage. The transform(s) may comprise a single logical comparison, a complex numerical algorithm, or a rule-inference approach of an expert system. Output may light a single LED or produce a 200-page report. In effect, we can create a *flow model* for any computer-based system, regardless of size and complexity.
Structured analysis began as an information flow modeling technique. A computer-based system is represented as an information transform as shown in Figure 12.10. A rectangle is used to represent an *external entity;* that is, a system element (e.g., hardware, a person, another program) or another system that produces information for transformation by the software or receives information produced by the software. A circle (sometimes called a *bubble*) represents a *process* or *transform* that is applied to data (or control) and changes it in some way. An arrow represents one or more *data items* (data objects). All arrows on a data flow diagram should be labeled. The double line represents a data store—stored information that is used by the software. The simplicity of DFD notation is one reason why structured analysis techniques are widely used.
It is important to note that no explicit indication of the sequence of processing or conditional logic is supplied by the diagram. Procedure or sequence may be implicit in the diagram, but explicit logical details are generally delayed until software design. It is important not to confuse a DFD with the flowchart.

Data Flow Diagrams

As information moves through software, it is modified by a series of transformations. A *data flow diagram* is a graphical representation that depicts information flow and the transforms that are applied as data move from input to output. The basic form of a data flow diagram, also known as a *data flow graph* or a *bubble chart,* is illustrated in Figure

The data flow diagram may be used to represent a system or software at any level of abstraction. In fact, DFDs may be partitioned into levels that represent increasing information flow and functional detail. Therefore, the DFD provides a mechanism for functional modeling as well as information flow modeling. In so doing, it satisfies the second operational analysis principle (i.e., creating a functional model) discussed in Chapter 11.

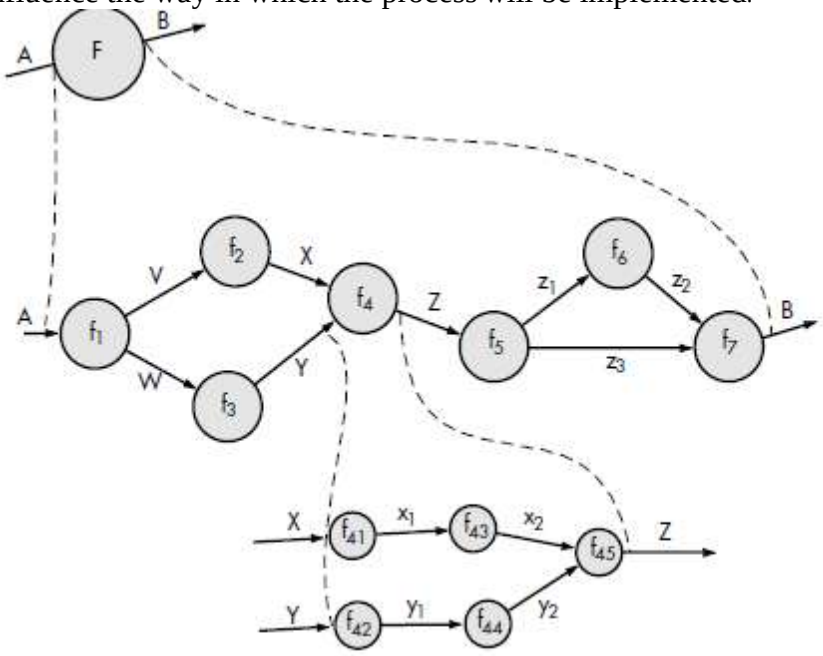
A level 0 DFD, also called a *fundamental system model* or a *context model*, represents the entire software element as a single bubble with input and output data indicated by incoming and outgoing arrows, respectively. Additional processes (bubbles) and information flow paths are represented as the level 0 DFD is partitioned to reveal more detail. For example, a level 1 DFD might contain five or six bubbles with interconnecting arrows. Each of the processes represented at level 1 is a subfunction of the overall system depicted in the context model.

As we noted earlier, each of the bubbles may be refined or layered to depict more detail. Figure illustrates this concept. A fundamental model for system *F* indicates the primary input is *A* and ultimate output is *B*. We refine the *F* model into transforms *f1* to *f7*. Note that *information flow continuity* must be maintained; that is, input and output to each refinement must remain the same. This concept, sometimes called *balancing*, is essential for the development of consistent models. Further refinement of *f4* depicts detail in the form of transforms *f41* to *f45*. Again, the input (*X, Y*) and output (*Z*) remain unchanged.

The basic notation used to develop a DFD is not in itself sufficient to describe requirements for software. For example, an arrow shown in a DFD represents a data object that is input to or output from a process. A data store represents some organized collection of data. But what is the content of the data implied by the arrow or depicted by the store? If the arrow (or the store) represents a collection of objects, what are they? These questions are answered by applying another component of the basic notation for structured analysis—the *data dictionary*. The use of the data dictionary is discussed later in this chapter.

DFD graphical notation must be augmented with descriptive text. A *process specification* (PSPEC) can be used to specify the processing details implied by a bubble within a DFD. The process specification describes the input to a function, the algorithm that is applied to transform the input, and the output that is produced. In addition, the PSPEC indicates restrictions and limitations imposed on the process (function),

performance characteristics that are relevant to the process, and design constraints that may influence the way in which the process will be implemented.



VALIDATION

Software testing is one element of a broader topic that is often referred to as *verification and validation* (V&V). *Verification* refers to the set of activities that ensure that software correctly implements a specific function. *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements. Boehm [BOE81] states this another way:
Verification: "Are we building the product right?"
Validation: "Are we building the right product?"
The definition of V&V encompasses many of the activities that we have referred to as *software quality assurance* (SQA).
Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, database review, algorithm analysis, development testing, qualification testing, and installation testing .

Although testing plays an extremely important role in V&V, many other activities are also necessary.
Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered. But testing should not be viewed as a safety net. As they say, "You can't test in quality. If it's not there before you begin testing, it won't be there when you're finished testing." Quality is incorporated into software throughout the process of software engineering. Proper application of methods and tools, effective formal technical reviews, and solid management and measurement all lead to quality that is confirmed during testing.
Miller relates software testing to quality assurance by stating that "the underlying motivation of program testing is to affirm software quality with methods that can be economically and effectively applied to both large-scale and small-scale systems."

SOFTWARE REQUIREMENT SPECIFICATION-SRS

An SRS is basically an organization's understanding (in customer or potential client's system requirements and dependencies at particular point in time (usually) prior to any actual design or development work. It's a two-way insurance policy that assures that both the client and the organization understand the other's requirements from that perspective at a given point in time.

The SRS document itself states in precise and explicit language those functions and capabilities a software system (i.e., a software application, an eCommerce Web site, and so on) must provide, as well as states any required constraints by which the system must abide. The SRS also functions as a blueprint for completing a project with as little cost growth as possible. The SRS is often referred to as the "parent" document because all subsequent project management documents, such as design specifications, statements of work, software architecture specifications, testing and validation plans, and documentation plans, are related to it.

It is important to note that an SRS contains functional and nonfunctional requirements only; it doesn't offer design suggestions, possible solutions to technology or business issues, or any other information other than what the development team understands the customer's system requirements to be.
A well-designed, well-written SRS accomplishes four major goals:

- It provides feedback to the customer. An SRS is the customer's assurance that the development organization understands the issues or problems to be solved and the software behavior necessary to address those problems. Therefore, the SRS should be written in natural language (versus a formal

language, explained later in this article), in an unambiguous manner that may also include charts, tables, data flow diagrams, decision tables, and so on.

- It decomposes the problem into component parts. The simple act of writing down software requirements in a well-designed format organizes information, places borders around the problem, solidifies ideas, and helps break down the problem into its component parts in an orderly fashion.
- It serves as an input to the design specification. As mentioned previously, the SRS serves as the parent document to subsequent documents, such as the software design specification and statement of work. Therefore, the SRS must contain sufficient detail in the functional system requirements so that a design solution can be devised.
- It serves as a product validation check. The SRS also serves as the parent document for testing and validation strategies that will be applied to the requirements for verification.

SRSs are typically developed during the first stages of "Requirements Development," which is the initial product development phase in which information is gathered about what requirements are needed--and not. This information-gathering stage can include onsite visits, questionnaires, surveys, interviews, and perhaps a return-on-investment (ROI) analysis or needs analysis of the customer or client's current business environment. The actual specification, then, is written after the requirements have been gathered and analyzed.

Why Technical Writers should be Involved with Software Requirements Specifications?

Unfortunately, much of the time, systems architects and programmers write SRSs with little (if any) help from the technical communications organization. And when that assistance is provided, it's often limited to an edit of the final draft just prior to going out the door. Having technical writers involved throughout the entire SRS development process can offer several benefits:

- Technical writers are skilled information gatherers, ideal for eliciting and articulating customer requirements. The presence of a technical writer on the requirements-gathering team helps balance the type and amount of information extracted from customers, which can help improve the SRS.
- Technical writers can better assess and plan documentation projects and better meet customer document needs. Working on SRSs provides technical writers with an opportunity for learning about customer needs firsthand--early in the product development process.

- Technical writers know how to determine the questions that are of concern to the user or customer regarding ease of use and usability. Technical writers can then take that knowledge and apply it not only to the specification and documentation development, but also to user interface development, to help ensure the UI (User Interface) models the customer requirements.
- Technical writers involved early and often in the process, can become an information resource throughout the process, rather than an information gatherer at the end of the process.

In short, a requirements-gathering team consisting solely of programmers, product marketers, systems analysts/architects, and a project manager runs the risk of creating a specification that may be too heavily loaded with technology-focused or marketing-focused issues. The presence of a technical writer on the team helps place at the core of the project those user or customer requirements that provide more of an overall balance to the design of the SRS, product, and documentation.

What Kind of Information Should an SRS Include?

You probably will be a member of the SRS team (if not, ask to be), which means SRS development will be a collaborative effort for a particular project. In these cases, your company will have developed SRSs before, so you should have examples (and, likely, the company's SRS template) to use. But, let's assume you'll be starting from scratch. Several standards organizations (including the IEEE) have identified nine topics that must be addressed when designing and writing an SRS:

1. Interfaces
2. Functional Capabilities
3. Performance Levels
4. Data Structures/Elements
5. Safety
6. Reliability
7. Security/Privacy
8. Quality
9. Constraints and Limitations

But, how do these general topics translate into an SRS document? What, specifically, does an SRS document include? How is it structured? And how do you get started? An SRS document typically includes four ingredients, as discussed in the following sections:

1. A template
2. A method for identifying requirements and linking sources

3. Business operation rules
4. A traceability matrix

Begin with an SRS Template

The first and biggest step to writing an SRS is to select an existing template that you can fine tune for your organizational needs (if you don't have one already). There's not a "standard specification template" for all projects in all industries because the individual requirements that populate an SRS are unique not only from company to company, but also from project to project within any one company. The key is to select an existing template or specification to begin with, and then adapt it to meet your needs.

In recommending using existing templates, I'm not advocating simply copying a template from available resources and using them as your own; instead, I'm suggesting that you use available templates as guides for developing your own. It would be almost impossible to find a specification or specification template that meets your particular project requirements exactly. But using other templates as guides is how it's recommended in the literature on specification development. Look at what someone else has done, and modify it to fit your project requirements. (See the sidebar called "Resources for Model Templates" at the end of this article for resources that provide sample templates and related information.) Table 1 shows what a basic SRS outline might look like.

1. Introduction
1.1 Purpose
1.2 Document conventions
1.3 Intended audience
1.4 Additional information
1.5 Contact information/SRS team members
1.6 References
2. Overall Description
2.1 Product perspective
2.2 Product functions
2.3 User classes and characteristics
2.4 Operating environment
2.5 User environment
2.6 Design/implementation constraints
2.7 Assumptions and dependencies
3. External Interface Requirements
3.1 User interfaces
3.2 Hardware interfaces

3.3 Software interfaces
3.4 Communication protocols and interfaces
4. System Features
4.1 System feature A
4.1.1 Description and priority
4.1.2 Action/result
4.1.3 Functional requirements
4.2 System feature B
5. Other Nonfunctional Requirements
5.1 Performance requirements
5.2 Safety requirements
5.3 Security requirements
5.4 Software quality attributes
5.5 Project documentation
5.6 User documentation
6. Other Requirements
Appendix A:
Terminology/Glossary/Definitions list
Appendix B: To be determined

Characteristics of a Good SRS:
Before winding up the topic SRS let me list the characteristics of a good SRS.It should be:

- Correct
- Unambiguous
- Complete
- Consistent
- Ranked for importance and/or stability
- Verifiable
- Modifiable
- Traceable