5

Structural Modeling[1]

A **structural model** is a formal way of representing the objects that are used and created by a business system. It illustrates people, places, things, or events about which information is captured and how they are related to each other. The structural model is drawn using an iterative process in which the model becomes more detailed and less conceptual over time. In analysis, analysts draw a **conceptual model**, which shows the logical organization of the objects without indicating how the objects are stored, created, or manipulated. Because this model is free from any implementation or technical detail the analysts can focus more easily on matching the model to the real business requirements of the system.

In design, analysts evolve the conceptual structural model into a design model that reflects how the objects will be organized in databases and files. A common design model is the **entity relationship diagram (ERD)** which you are familiar with from your database class. UML structural models are more expressive than ERDs because they contain both the structure of objects as well as their behavior.

Introduction to data modeling

The UML language is designed to express the static as well as the dynamic aspects of a system. The static part is represented mainly by the class diagram and defined by the classes, attributes of classes, and the relationships among classes. This chapter focuses on the data modeling aspect of the class diagram.

CLASSES

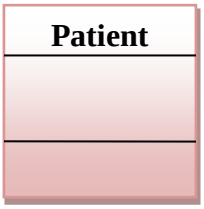
In data modeling terms, a **class** is a person, place, thing, event, or concept in the user environment about which the organization wishes to maintain data. This is derived from the notion of an entity type in the entity-relationship model. Thus a data-intensive class is also called an entity class, and its instances are called entity objects. Some examples of entity classes follow:

Person: Employee, Student, Patient
Place: State, Region, Country, Branch
Thing: Machine, Building, Automobile, Product
Event: Sale, Registration, Renewal
Concept: Account, Course

[1] This text is written by Diane Lending. Parts are adapted from George, Batra, Valacich, and Hoffer: *Object-Oriented Systems Analysis and Design, Second Edition*, Prentice Hall, 2007 and Dennis, Wixom, and Tegarden: *Systems Analysis & Design with UML Version 2.0: An Object Oriented Approach, Third Edition,* Wiley, 2009.

The notation for a class is a rectangle which is partitioned further into three rectangles, one for the name of the class, the second for the attributes, and the third for the operations (Figure 5.1). This chapter will not consider the third rectangle, which is for the operations of a class. Refer to Figure 5-8 on page 210 in your textbook for the full syntax of a class diagram.

Figure 5.1 Class Patient



In later chapters, you will be introduced to other types of classes besides entity classes.

Object

An **object** is an instance of a class; it includes data and behavior. As mentioned earlier, this chapter will focus on the data part of objects. Thus, Maria and David may be objects of the class Patient and have data characteristics such as name and address. The class might have thousands of patient objects. A generic object of the class Patient can be expressed as: Patient.

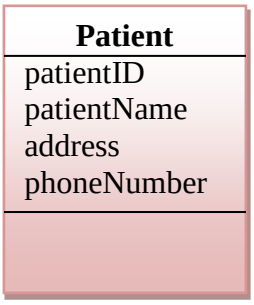
ATTRIBUTES OF CLASSES

A class has a set of attributes associated with it. An **attribute** is a property or characteristic of a class that is of interest to the organization. Following are some typical classes and their associated attributes:

Patient: patientId, patientName, address, phoneNumber
Automobile: vehicleId, color, weight, horsepower
Employee: employeeId, employeeName, address, jobTitle

We use nouns with lowercase letters in naming an attribute, (e.g., address); if an attribute is composed of two words, the second word starts with the first letter in uppercase (e.g., patientName.) See Figure 5.2 which shows the attributes of the class Patient.

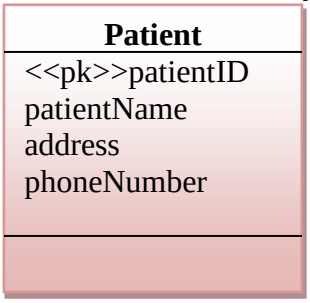
Figure 5.2 The Class Patient and its Attributes



Identifier

An **identifier** is an attribute or a combination of attributes that has been selected to be used as the unique characteristic for a class. The values of the identifier allow objects to be distinguished from one another. An identifier is also called a **primary key.** Figure 5.3 shows the identifier of Patient as patientID by using the stereotype <<pk>>, where "pk" stands for primary key. A **stereotype** is a construct that extends the UML vocabulary. Because primary key has no standard UML symbol, the designer can create one using a stereotype.

Figure 5.3 The identifier shown by the stereotype <<pk>>

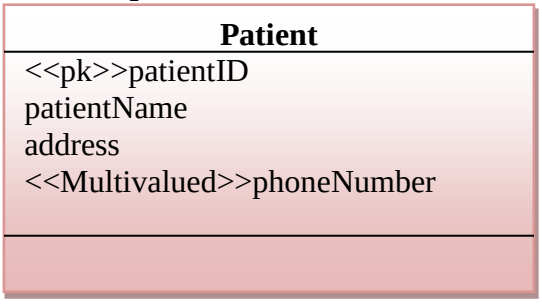


Multivalued Attributes

A multivalued attribute can take on more than one value for each object. Suppose that a patient can have several phone numbers. Say, for example, the patient Mary has two phone numbers, Rick has three, and David has none. A patient can have none, one, or several phone numbers. The attribute phone number is multivalued because for a given object, the attribute can have more than one value.

During conceptual design, there are two common approaches to handle multivalued attributes. The first approach is to use the stereotype <<Multivalued>> before the attribute, see Figure 5.4. Note that object-relational DBMSs can implement multivalued attributes directly.

Figure 5.4 Attribute phoneNumber shown as Multivalued



The other approach is to create a separate class for the multivalued attribute and use an association relationship to link it with the main class. Relationships will be introduced soon. This approach also handles several attributes that repeat together, called a repeating group. For example, dependent name, age, and relation to patient (spouse, child, etc.) are multivalued attributes about a patient and these attributes repeat together. We can show this using a class called Dependent and a relationship between Dependent and Patient.

Composite Attribute

The attribute address in the class Patient can be designed as an atomic attribute; however, it will make querying difficult on a part of the address like City. The address typically has a value such "1220 Hillcrest Avenue, Harrisonburg, VA 22801" where the value of City is embedded in a larger string. Address can be split into four attributes: Street, City, Sate and Zip code. However, because the four attributes almost always go together, it is better to design it as a composite attribute consisting of Street, City, State and Zip Code. To accomplish this, the composite needs to be defined as a class. Thus, we define a class Address (Figure 5.5), which can then be shown as a type in the definition of class Patient. The other attributes are shown as typical atomic attribute types. Types are generally not included in the first iteration of structural modeling.

Figure 5.5a Composite Attribute Address shown as a Class

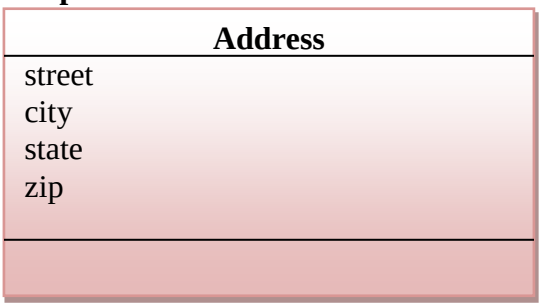
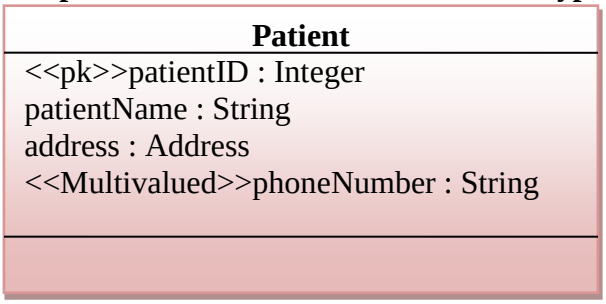


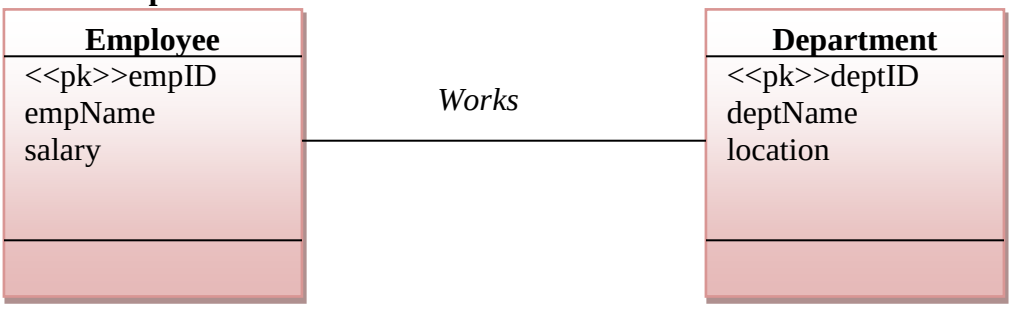
Figure 5.5b Composite Attribute Address shown as Type Address



RELATIONSHIPS AMONG CLASSES

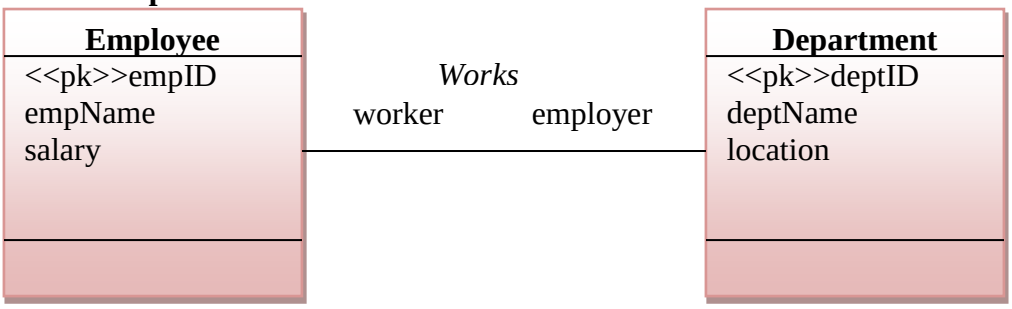
Relationships are the glue that holds together the elements of a class diagram. A **relationship** is the semantic connection between **objects** of one or more classes. A relationship can result because of an event that has occurred or because of some natural linkage between entity instances. Relationships generally are labeled with verb phrases. For example, we might be interested in knowing which employees belong to a certain department. This leads to a relationship (called works) between the Employee and Department classes as shown in Figure 5.6.

Figure 5.6 Relationship between two classes



The *works* relationship can be used to determine the department of a given employee. Conversely, employees who belong to a particular department also can be determined. The relationship can be optionally qualified by the roles of each class in the relationship. A **role** provides a name to identify a relationship end within a relationship. For example, the class Department can be qualified by the role Employer, and the class Employee by the role Worker (Figure 5.7.) However this can clutter the class diagram and often is not used unless needed for clarification.

Figure 5.7 Relationship *Works* shown with Roles



A relationship generally is described by its name, degree, and multiplicity (which is comprised of minimum and maximum cardinality.) Next, the degree and multiplicity characteristics will be detailed.

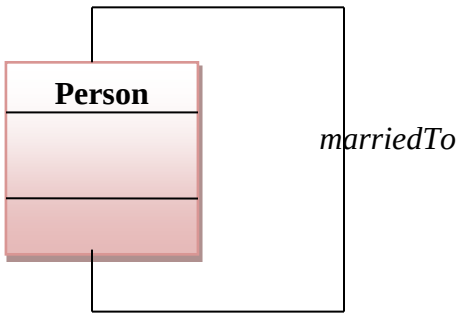
Degree of a Relationship

The **degree** of a relationship is the number of classes that participate in that relationship. Thus, the relationship *Works* illustrated previously is of degree two, because it has two classes: Employee and Department. Relationships of degree two (binary) are the most common. Relationships of degree one (unary) and degree three (ternary) are encountered occasionally. Higher-degree relationships are possible, but they rarely are encountered in practice.

Unary Relationship

Also called a recursive relationship, a **unary** relationship is a relationship between the objects of one class. An example is shown in Figure 5.8. The relationship *marriedTo* is between two different objects of the class Person. Be careful not to interpret this as one person is married to himself. Since relationships are between **objects** of a class, this is a relationship between two different objects of the same class.

Figure 5.8 Unary Relationship



Binary Relationship

A **binary** relationship is a relationship between objects of two classes and is the most common type of relationship encountered in data modeling. Figure 5.6 shows the *works* relationship between Employee and Department.

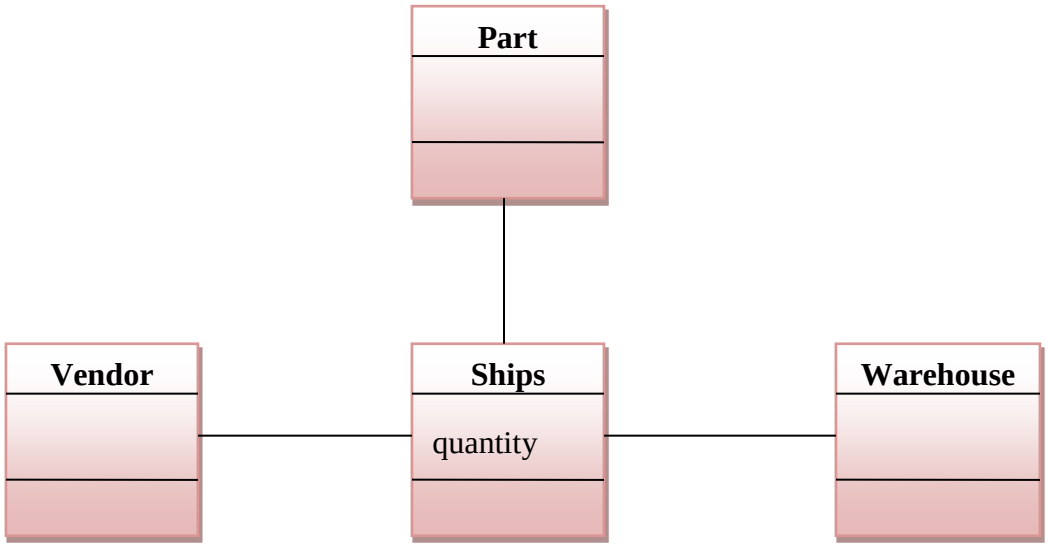
Ternary Relationship

A **ternary** relationship is a simultaneous relationship among objects of three classes. The most convenient representation of a ternary (or higher-degree) relationship is to show it as a class and relate it to the involved classes. In the example shown in Figure 5.9, the relationship *ships* tracks the quantity of a given part shipped by a particular vendor to a selected warehouse.

Note that a ternary relationship is not the same as three binary relationships. For example, *quantity* is an attribute of the *ships* relationship. The attribute quantity cannot be

associated properly with any of the three possible binary relationships among the three entity types (such as that between Part and Vendor), because quantity is the amount of a particular Part shipped from a particular Vendor to a particular Warehouse.

Figure 5.9 Ternary Relationship *Ships* Shown as a Class

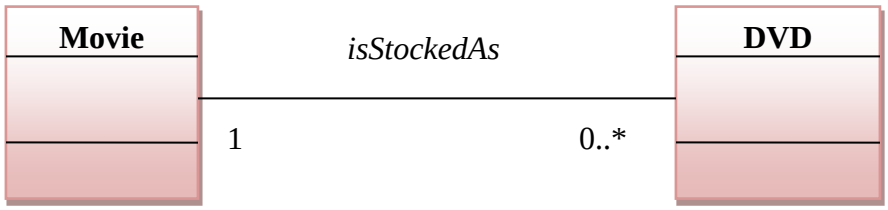


Multiplicity in Relationships

Suppose that two classes, A and B, are connected by a relationship. The **cardinality** of a class A in relationship is the number of objects of class A that can (or must) be associated with each object of class B. The minimum cardinality of class A is the minimum number of objects of class A that may be associated with each object of class B. The maximum cardinality of a class A is the maximum number of objects of class A that may be associated with each object of class B. **Multiplicity** of a class A is composed of cardinalities that specify the range of the number of objects of class A that can (or must) be associated with each object of class B. For example, consider the relationship *isStockedAs* among DVDs and Movies shown in Figure 5.10.

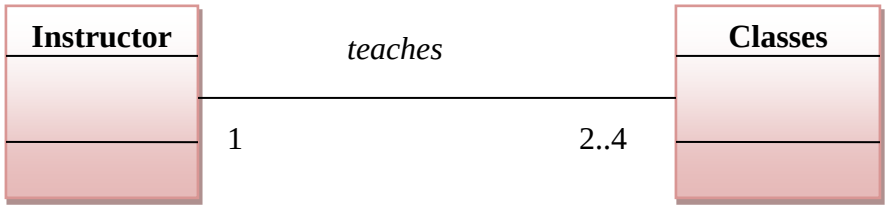
A video store can stock more than one DVD of a given movie. Thus, the maximum cardinality of DVD is "many," which is denoted by *. It is also possible that the store may not have a single DVD of a particular movie in stock. Thus, the minimum cardinality of DVD is 0. The multiplicity of DVD is 0..*. Suppose a DVD holds one and only one movie. The minimum as well as maximum cardinality of movie is 1, and the multiplicity is also 1.

Figure 5.10 Multiplicity of a Relationship



Minimum and maximum cardinalities can also be specific numbers rather than 0,1, or *. Such a situation is referred to as a multiplicity with fixed cardinality (see Figure 5.11.) If the multiplicity is 0..* or 1..*, then the multiplicity is often referred to as "many."

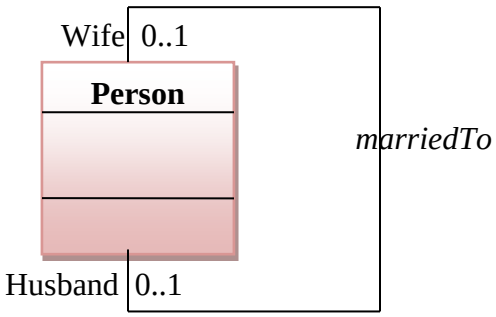
Figure 5.11 Multiplicity with a Fixed Cardinality



Multiplicity in a Unary Relationship

In a unary relationship, it is difficult to assign multiplicity because both ends of the relationship connect to the same class. To distinguish the participants in the relationship, the notion of roles can be used. For example in Figure 5.12, the relationship *marriedTo* has been enhanced to include roles and their multiplicity.

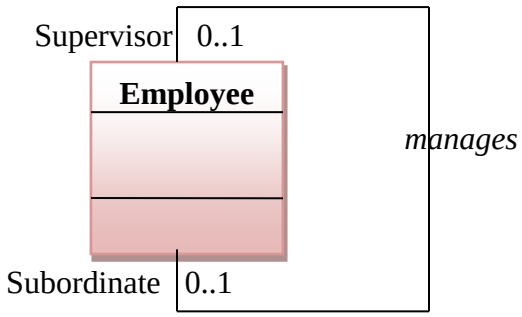
Figure 5.12 Unary Relationship



Let's consider the *manages* relationship illustrated in Figure 5.13. An employee can manage other employees. The multiplicity of the Subordinate role is 0..*, and the multiplicity of the Supervisor role is 0..1. Note that the minimum cardinality of Supervisor is zero because the

top boss does not have a supervisor. Further, the minimum cardinality of Subordinate is also zero because an employee might not have a subordinate.

Figure 5.13 Unary Relationship

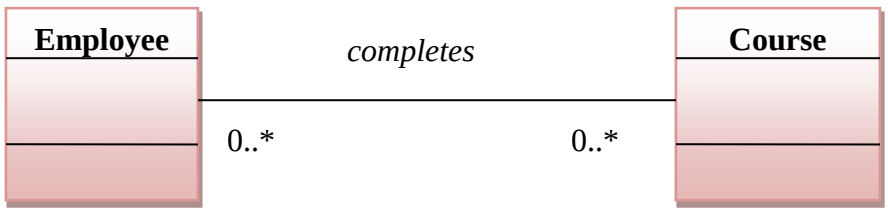


Association

An **association** is a relationship where no object is subordinate to others. The association is the most common type of relationship. For example, the relationship between a movie and a DVD (Figure 5.10) and the relationship between an employee and a department (Figure 5.6) are associations. These two relationships are 1-to-many. The *marriedTo* relationship in Figure 5.11 is also an association, which is unary and 1-to-1.

Attributes can be associated with a many-to-many relationship. For example, suppose that there is a relationship between employees and courses as shown in Figure 5.13.

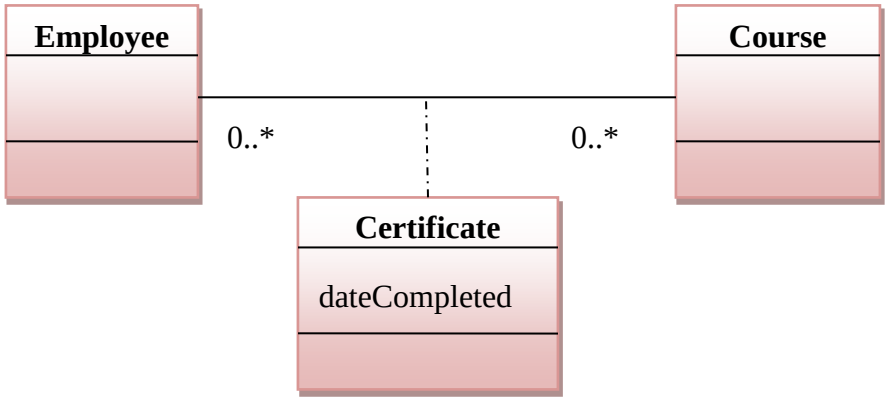
Figure 5.13 A many-to-many relationship



The company may want to record the date (month and year) when an employee completes the course. DateCompleted is not an attribute of Employee since the employee may finish different Courses on different dates. Similarly dateCompleted is not an attribute of Course since different Employees may finish the course on different dates. Instead dateCompleted is a property of the relationship *completes*. This many-to-many relationship is behaving like a class; that is, it has attributes.

An **associative class** is a many-to-many association that the data modeler chooses to model as a class. Since we use nouns to indicate classes, we rename the association from a verb to a noun. Figure 5.14 illustrates the employee-course example with an associative class. This new class may be involved in associations with other classes as needed.

Figure 5.14 Example of an Associative Class



While an associative class is similar to that of an intersection entity in an ERD, it is not required to appear on all many-to-many relationships. In UML modeling, the associative class is only shown when there are attributes of the relationship.

Aggregation Relationship

If two classes are bound by a whole-part relationship, the association is called an **aggregation.** Hence, aggregation is known as a "part-of" relationship. For example, a computer is made up of parts. When do we use aggregation rather than an association? For example, is an order made up of line items, or is a team made up of players? To determine whether a relationship is an aggregation, Quatrani (1999) suggests the following three guidelines:

1. Is the phrase *part of* used to describe the relationships? For example is a player part of a team?
2. Are some operations on the whole automatically applied to its parts? For example, if an order is deleted, it automatically leads to deletion of all line items in the order.
3. Is there intrinsic asymmetry to the relationships where one class is subordinate to others?

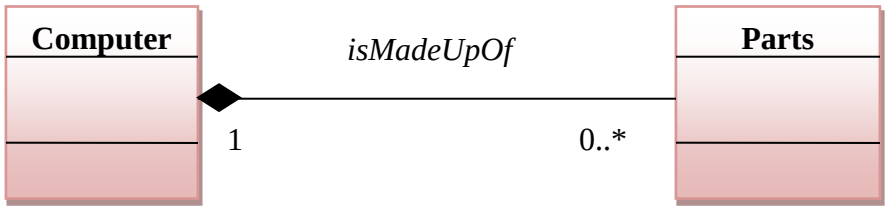
The systems analyst should also consider the phrases *made up of, consists of,* or *element of* in case *part of* does not seem to be a natural way to describe the relationship. Consider how the two items are retrieved. If one is retrieved, is it likely that the other will be retrieved too?

When you retrieve an order, you typically retrieve its line items, too? In order to add a new object, is it necessary to ensure that an object of the related class already has been created? Before you add a part on a bike, you need to ensure that the bike object has been created. Is one object dependent on another, or can it exist independently of the other? A subassembly for a bike can exist independently of the bike for a while, it eventually becomes part of the bike. Does the primary key of one class depend on the other class's primary key? The primary key of order line items depends on the primary key of the order.

The answers to these questions determine whether a relationship can be considered an aggregation. The aggregation is strongly bound or loosely bound, depending on the number of criteria satisfied by the relationship. The recognition of a relationship at the analysis state alerts the designer to consider the storage, addition, deletion, modification, and retrieval of related objects.

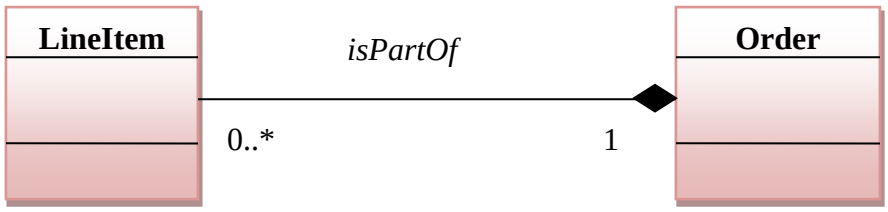
A type of strong aggregation of a relationship is called **composition.** In this case, the part is strongly owned by the composite and might not be part of any other composite. Thus, composition is a form of aggregation with strong ownership and a coincident lifetime of the part with the whole. Typically the parts exist only as long as the whole exists. If the whole is deleted, the parts are deleted as well. For example, if a computer is destroyed, the hard drive is also considered destroyed. We would model the relationship between the computer and its parts as a composition relationship as shown in Figure 5.15. In this type of relationship, a deletion will cascade deletions. In the model, a solid diamond shows composition.

Figure 5.15 A Composition Relationship



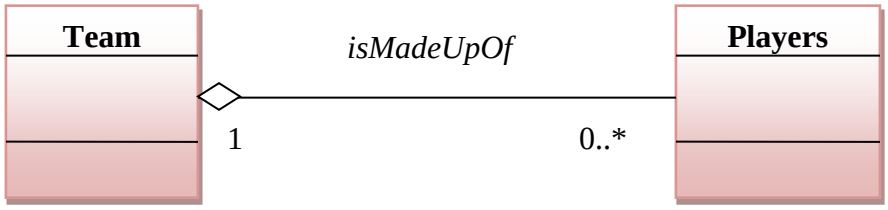
Another common example of composition is the relationship between a transaction and its line items. For example, a line item is part of an order as shown in Figure 5.16. The line items have no meaning if the order is deleted.

Figure 5.16 Order and Line Items as a Composition Relationship



An aggregation may be loosely bound. Consider a team that is made up of (or consists of) players as shown in Figure 5.17. If the team is dissolved, information about players might or might not be maintained. If the players leave, a team might or might not be dissolved. The deletion semantics will depend on the situation. In the data model, the diamond is not filled in to indicate a loosely-bound aggregation.

Figure 5.17 A Loosely-Bound Aggregation



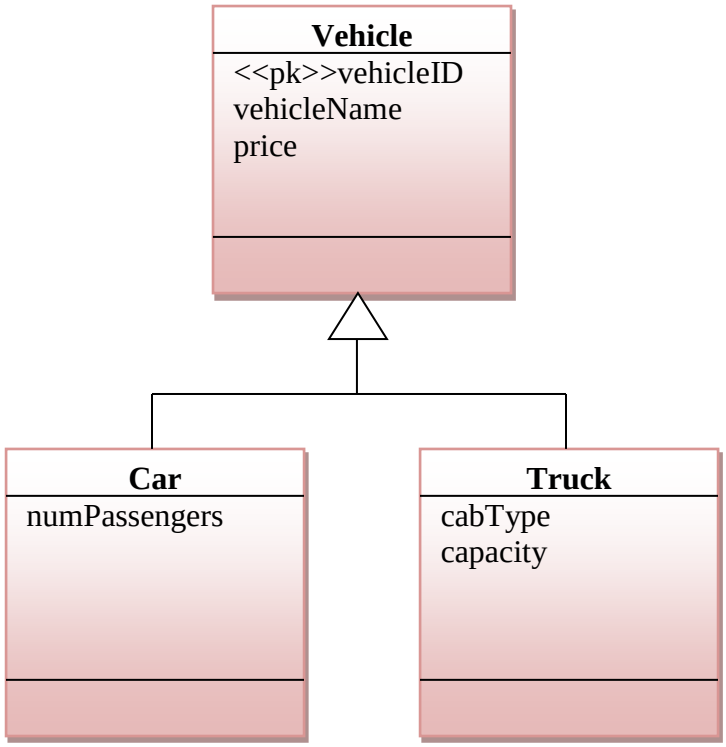
By indicating a relationship as an aggregation, the issues regarding object deletion and creation semantics can be considered more closely. Thus, the information collected during the analysis phase can be helpful to a designer in a later phase.

Generalization

The **generalization** relationship also is called a supertype/subtype (or superclass/subclass) relationship and is based upon the inheritance concept. **Inheritance** defines a relationship among classes where one class shares the structure or behavior of another class. This relationship allows one to model a general class (called the supertype) and then to subdivide it into several specialized classes (called subtypes). The subtypes inherit attributes and operations from the supertype. Inheritance is called an "is-a" relationship.

As illustrated in Figure 5.18, a car "is-a" vehicle and a truck "is-a" vehicle. Note that the subclass Car inherits all attributes of Vehicle and has an additional attribute numPassengers. The subclass Truck inherits all attributes of Vehicle and has two additional attributes: cabType and capacity. Vehicle is the more general type, and Car and Truck are the more specialized types distinguished by vehicleType.

Figure 5.18 Generalization Applied to Vehicle Supertype



GATHERING INFORMATION FOR CONCEPTUAL STRUCTURAL MODELING

Systems analysts usually do conceptual data modeling at the same time as they do other requirements analysis. They can use methods such as interviewing, questionnaires, and JAD sessions to collect information for conceptual data modeling. In addition, they develop a data model for the *as-is* system. This *as-is* model is essential for planning the conversion of the *as-is* database into the *to-be* database. In addition, it provides a good starting place for understanding the new system's data requirements.

Requirements determination methods for conceptual data modeling must include questions and investigations that take a data focus rather than a process, logic, or user interface focus. The types of questions that may be asked are shown in Table 5.1.

In addition, analysts study documents to help develop the structural models. There are three common types of documents to study: use case descriptions, documents and reports from the *as-is* system, and patterns.

Use Case Descriptions The analyst starts by reviewing the use-case descriptions and the use-case diagrams. This analysis is called textual analysis. The text in the descriptions is examined to identify potential objects, attributes, operations, and relationships. The nouns in the use case suggest possible classes, whereas the verbs suggest possible operations. Table 5.2 shows guidelines for textual analysis.

Documents The reports, forms, and other documents from the *as-is* system also provide rich sources for possible classes and attributes. For example, a form that a patient fills out when registering at the dentist's office, suggests that Patient might be a class and the data items filled out could be attributes of the patient. The form also shows that the patient may have two insurance policies suggesting that Insurance Policy might be a class and the possible relationship between Patient and Policy.

Patterns A pattern is a useful group of collaborating classes that provide a solution to a common business problem. Because patterns provide a solution to commonly occurring problems, they are reusable. You can find patterns in books and on the web that can be adapted for use in many situations.

Table 5.1 Questions to consider when developing data models

Category of Questions	Information to Derive from Systems Users and Business Managers
Classes	What are the subjects/objects of the business? What types of people, places, things, and materials are used or interact in this business? How many instances of each object may exist?
Attributes	What data needs to be maintained about the objects? What characteristics describe each object? On what basis are objects referenced, selected, qualified, sorted, and characterized? Can a property of an object have multiple values? Is the property mandatory or optional for a given object?
Identifier	What unique characteristic(s) distinguishes each object from other objects of the same type? Could this distinguishing feature change over time or is it permanent?
Association, Aggregation, and Composition	What relationships exist between objects? Is a relationship of the type: part-whole? If yes, can one exist without the other? How many objects are involved in a relationship? Is the relationship between objects of the same class? What is the cardinality of an object participating in a relationship?
Generalization	Is one object "a kind of" another? Do objects form a hierarchy from more general to more specialized?
Time Dimensions	Over what period of time are you interested in these data? If characteristics of an object change over time, must you know the obsolete values? What is the unit of time?
Integrity Rules	Are values for data characteristics limited in any way? Can the data characteristics take non-null values only?
Security Controls	Who can create data? Who can retrieve data? Who can update data? Who can delete data? Are there time intervals when the data cannot be accessed?

Source: This table is from George, Batra, Valacich, and Hoffer: *Object-Oriented Systems Analysis and Design*, Second Edition, Prentice Hall, 2007.

Table 5.2 Textual Analysis Guidelines

Summary of guidelines that may be useful in examining use-case descriptions and other documents
• **A common or improper noun implies a class of objects** • **A proper noun or direct reference implies an instance of a class** • **A collective noun implies a class of objects made up of groups of instances of another class** • **An adjective implies an attribute of an object** • **A doing verb implies an operation** • **A being verb implies a classification relationship between an object and its class** • **A having verb implies an aggregation or association relationship** • **A transitive verb implies an operation** • **An intransitive verb implies an exception** • **A predicate or descriptive verb phrase implies an operation** • **An adverb implies an attribute of a relationship or an operation**
Source: This table is from Dennis, Wixom, Tegarden, *Systems Analysis & Design with UML Version 2.0: An Object Oriented Approach, Third Edition*, Wiley, 2009.

Dentist Office Class Diagram Example

In an interview with the staff at the dentist office, you find that there are three types of employees: dentists, hygienists, and administrative staff. You inquire and find that patients make appointments with a dental team composed of a dentist and a hygienist. Each calendar day is divided into hour blocks. The dental team is assigned to a number of blocks. When a patient calls, the patient is scheduled for one or more blocks with the appropriate dental team.

You ask about attributes and find that they want to keep similar information for all employees such as name, address, phone, birth date, age, and salary. They keep all the same information about patients except salary but also include employer and insurance carrier.

From this you conclude that the best way to model all the various people in the system is as a generalization relationship. Employees and Patients are kinds of Person. Dentists, hygienists and administrative staff are kinds of Employees. A dental team is composed of dentists and hygienists.

A calendar day is composed of appointment blocks. This is a composition because the appointments cannot exist without the calendar day. The patient will make appointments and the Dental team will be assigned to appointments.

The resulting model is shown in Figure 5.19.

Figure 5.19: Data Model of Dentist Office