10

Modeling the Static/ Data Aspects of the System

Having employed use case analysis techniques in Chapter 9 to round out the SRS requirements specification, we're ready to tackle the next stage of modeling, which is determining how we're going to meet those requirements in an object-oriented fashion.

We learned in Part 1 of the book that objects form the building blocks of an OO system, and that classes are the templates used to define and instantiate objects. An OO model, then, must specify:

- **What classes of objects we are going to need to create and instantiate in order to represent the proper abstraction:** in particular, their attributes, methods, and structural relationships with one another. Because these elements of an object-oriented system, once established, are fairly static – in the same way that a house, once built, has a specific layout, a given number of rooms, a particular roofline, and so forth – we often refer to this process as preparing the **static model**.

  We can certainly change the static structure of a house over time by undertaking remodeling projects, just as we can change the static structure of an OO software system as new requirements emerge by deriving new subclasses, inventing new methods for existing classes, and so forth. However, if a structure – whether a home or a software system – is properly designed from the outset, then the need for such changes should arise relatively infrequently over its lifetime and shouldn't be overly difficult to accommodate.

- **How these objects will need to collaborate in carrying out the overall requirements, or 'mission', of the system:** The ways in which objects interact can change literally from one moment to the next based upon the circumstances that are in effect. One moment, a `Course` object may be registering a `Student` object, and the next, it might be responding to a query by a `Professor` object as to the current student headcount. We refer to the process of detailing object collaborations as preparing the **dynamic model**. Think of this as all of the different day-to-day activities that go on in a home: same structure, different functions.

The static and dynamic models are simply two different sides of the same coin: they jointly comprise the blueprint that we'll work from in implementing an object-oriented Student Registration System (SRS) application in Part 3 of the book.

In this chapter, we'll focus on building the static model for the SRS, leaving a discussion of the dynamic model for Chapter 11. You will learn:

- A technique for identifying the appropriate classes and their attributes.
- How to determine the structural relationships that exist among these classes.
- How to graphically portray this information as a **class diagram** using the UML notation.

Identifying Appropriate Classes

Our first challenge in object modeling is to determine what classes we're going to need as our system building blocks. Unfortunately, the process of class identification is rather 'fuzzy'; it relies heavily on intuition, prior modeling experience, and familiarity with the subject area, or **domain**, of the system to be developed. So, how does an object-modeling novice **ever** get started? One tried and true (but somewhat tedious) procedure for identifying candidate classes is to use the 'hunt and gather' method: that is, to hunt for and gather a list of all nouns/noun phrases from the project documentation set and to then use a process of elimination to whittle this list down into a set of appropriate classes.

In the case of the SRS, our documentation set thus far consists of:

- The requirements specification
- The use case model that we prepared in Chapter 9

Noun Phrase Analysis

Let's perform noun phrase analysis on the requirements specification first, which was originally presented in the Introduction to the book, a copy of which is provided below. We've highlighted all noun phrases:

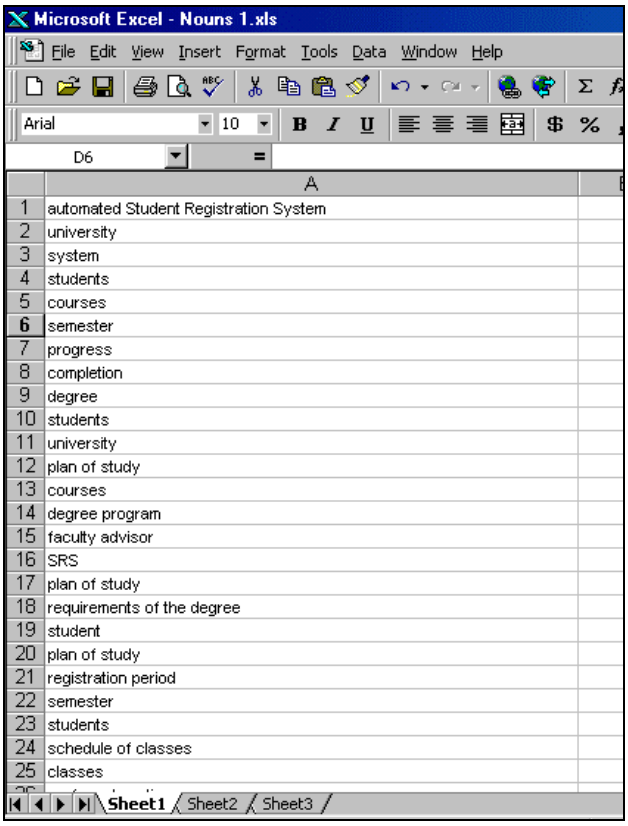
> We have been asked to develop an **automated Student Registration System** (**SRS**) for the **university**. This **system** will enable **students** to register on-line for **courses** each **semester**, as well as tracking their **progress** toward **completion** of their **degree**.
>
> When a **student** first enrolls at the **university**, he/she uses the **SRS** to set forth a **plan of study** as to which **courses** he/she plans on taking to satisfy a particular **degree program**, and chooses a **faculty advisor**. The **SRS** will verify whether or not the proposed **plan of study** satisfies the **requirements of the degree** that the **student** is seeking.
>
> Once a **plan of study** has been established, then, during the **registration period** preceding each **semester**, **students** are able to view the **schedule of classes** on line, and choose whichever **classes** they wish to attend, indicating the **preferred section** (**day of the week** and **time of day**) if the **class** is offered by more than one **professor**. The **SRS** will verify whether or not the **student** has satisfied the necessary **prerequisites** for each **requested course** by referring to the **student**'s on-line **transcript** of **courses completed** and **grades received** (the **student** may review his/her **transcript** on-line at any time).

Assuming that (a) the **prerequisites** for the **requested course(s)** are satisfied, (b) the **course(s)** meet(s) one of the **student**'s **plan of study requirements**, and (c) there is **room** available in each of the **class(es)**, the **student** is enrolled in the **class(es)**.

If (a) and (b) are satisfied, but (c) is not, the **student** is placed on a **first-come, first-served wait list**. If a **class/section that he/she was previously wait-listed for** becomes available (either because some other **student** has dropped the **class** or because the **seating capacity** for the **class** has been increased), the **student** is automatically enrolled in the **waitlisted class**, and an **email message** to that effect is sent to the **student**. It is his/her **responsibility** to drop the **class** if it is no longer desired; otherwise, he/she will be billed for the **course**.

Students may drop a **class** up to the **end** of the **first week of the semester in which the class is being taught**.

A simple spreadsheet serves as an ideal tool for recording our initial findings; just enter noun phrases as a single-column list in the order in which they occur in the specification. Don't worry about trying to eliminate duplicates or consolidating synonyms just yet; we'll do that in a moment. The resultant spreadsheet is shown in part below.



We're working with a very concise requirements specification (approximately 350 words in length), and yet this process is already proving to be very tedious! It would be impossible to carry out an exhaustive noun phrase analysis for anything but a trivially simple specification. If you are faced with a voluminous requirements specification, start by writing an 'executive summary' of no more than a few pages to paraphrase the system's mission, and then use your summary version of the specification as the starting point for your noun survey. Paraphrasing a specification in this fashion provides the added benefit of ensuring that you have read through and *understand* the 'big picture' concerning the system requirements. Of course, you'll need to review your summary narrative with your customers/users to ensure that you've accurately captured all key points.

After you've typed all of the nouns/noun phrases into the spreadsheet, sort the spreadsheet and eliminate duplicates; this includes eliminating plural forms of singular terms (e.g. eliminate 'students' in favor of 'student'). We want all of our class names to be singular in the final analysis, so if any plural forms remain in the list after eliminating duplicates (e.g. 'prerequisites'), make these singular, as well. In so doing, our SRS list shrinks to 38 items in length, as shown below.

	A
1	automated Student Registration System
2	class
3	class that he/she was previously waitlisted for
4	completion
5	course
6	courses completed
7	day of the week
8	degree
9	degree program
10	email message
11	end
12	faculty advisor
13	first week of the semester in which the class is being taught
14	first-come, first-served wait list
15	grades received
16	plan of study
17	plan of study requirements
18	preferred section
19	prerequisites
20	professor
21	progress
22	registration period
23	requested course
24	requirements of the degree
25	responsibility
26	room
27	schedule of classes
28	seating capacity
29	section
30	section that he/she was previously waitlisted for
31	semester
32	SRS
33	student
34	system
35	time of day
36	transcript
37	university
38	waitlisted class

Sheet1 / Sheet2 / Sheet3 /

Remember, we're trying to identify physical or conceptual objects: as stated in Chapter 3, '*something mental or physical toward which thought, feeling, or action is directed*'. Let's now make another pass to eliminate:

- References to the system itself ('automated Student Registration System', 'SRS', 'system').
- References to the university. Because we are building the SRS within the context of a single university, the university in some senses 'sits outside' and 'surrounds' the SRS; we don't need to manipulate information about the university within the SRS, and so we may eliminate the term 'university' from our candidate class list.

Note, however, that if we were building a system that needed to span multiple universities – say, a system that compared graduate programs of study in information technology across the top 100 universities in the country – then we would need to model each university as a separate object, in which case we'd keep 'university' on our candidate class list.

- Other miscellaneous terms which don't seem to fit the definition of an object are 'completion', 'end', 'progress', 'responsibility', 'registration period' and 'requirements of the degree'. Admittedly, some of these are debatable, particularly the last two; to play it safe, you may wish to create a list of rejected terms to be revisited later on in the modeling lifecycle.

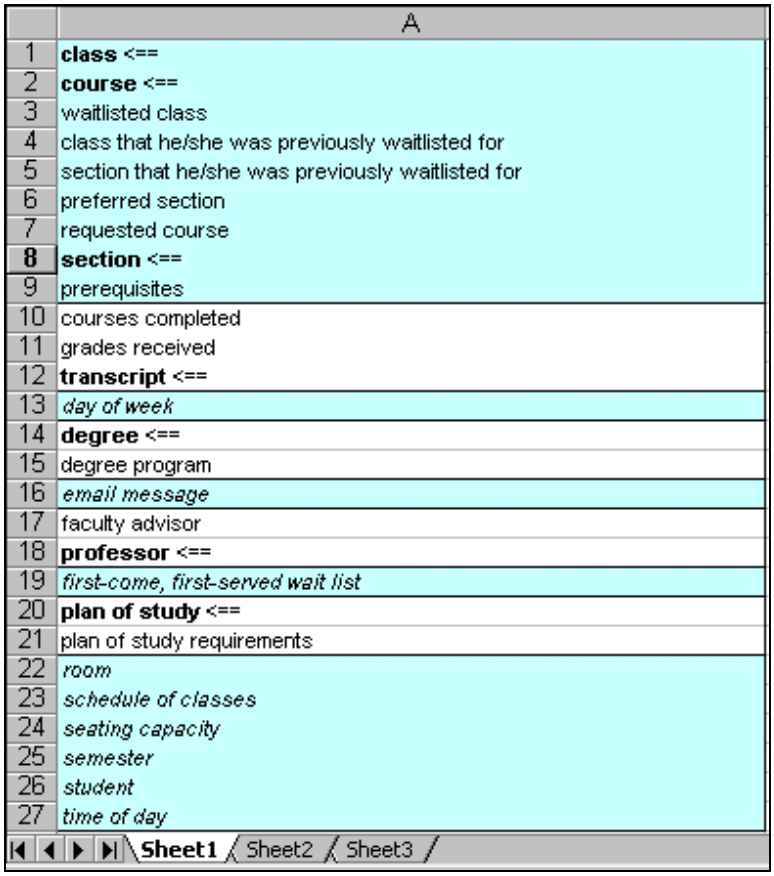
The list shrinks to 27 items as a result, as shown below – it's starting to get manageable now!

	A
1	class
2	class that he/she was previously waitlisted for
3	course
4	courses completed
5	day of week
6	degree
7	degree program
8	email message
9	faculty advisor
10	first-come, first-served wait list
11	grades received
12	plan of study
13	plan of study requirements
14	preferred section
15	prerequisites
16	professor
17	requested course
18	room
19	schedule of classes
20	seating capacity
21	section
22	section that he/she was previously waitlisted for
23	semester
24	student
25	time of day
26	transcript
27	waitlisted class

Sheet1 Sheet2 Sheet3

The next pass is a bit trickier. We need to group apparent synonyms, to choose the one designation from among each group of synonyms that is best suited to serving as a class name. Having a subject matter expert on your modeling team is important for this step, because determining the subtle shades of meaning of some of these terms so as to group them properly is not always easy.

We've grouped together terms which seem to be synonyms in the figure below, **bolding** the term in each synonym group that we are inclined to choose above the rest; italicized words represent those terms for which no synonyms have been identified.



	A
1	**class** <==
2	**course** <==
3	waitlisted class
4	class that he/she was previously waitlisted for
5	section that he/she was previously waitlisted for
6	preferred section
7	requested course
8	**section** <==
9	prerequisites
10	courses completed
11	grades received
12	**transcript** <==
13	*day of week*
14	**degree** <==
15	degree program
16	*email message*
17	faculty advisor
18	**professor** <==
19	*first-come, first-served wait list*
20	**plan of study** <==
21	plan of study requirements
22	*room*
23	*schedule of classes*
24	*seating capacity*
25	*semester*
26	*student*
27	*time of day*

Sheet1 Sheet2 Sheet3

Let's review the rationale for our choices:

- We choose the shorter form of equivalent expressions whenever possible – **degree** instead of degree program and **plan of study** instead of plan of study requirements – to make our model more concise.
- Although they aren't synonyms as such, the notion of a **transcript** implies a record of 'courses completed' and 'grades received', so we'll opt to drop the latter two noun phrases for now.
- When choosing candidate class names, we should avoid choosing nouns that imply **roles** between objects. As we learned in Chapter 5, a role is something that an object belonging to class A possesses by virtue of its relationship to/association with an object belonging to class B. For example, a professor holds the role of 'faculty advisor' when that professor is associated with a student via an 'advises' association. Even if a professor were to lose all of his/her advisees, thus losing the role of faculty advisor, he/she would still be a professor by virtue of being employed by the university – it is inherent in the person's nature relative to the SRS.

If a professor were to lose his/her job with the university, one might argue that he/she is no longer a professor; but then, this person would have no dealings with the SRS, either, so it is a moot point.

For this reason, we prefer 'Professor' to 'Faculty Advisor' as a candidate class name, but make a mental note to ourselves that faculty advisor would make a good potential association when we get to considering such things later on.

- Regarding the notion of a course, we see that we have collected numerous noun phrases that all refer to a course in one form or another: 'class', 'course', 'preferred section', 'requested course', 'section', 'prerequisite', 'waitlisted class', 'class that they were previously waitlisted for', 'section that they were previously waitlisted for'. Within this grouping, several roles are implied:
  - 'waitlisted class' in its several different forms implies a role in an association between a Student and a Course
  - 'prerequisite' implies a role in an association between two Courses
  - 'requested course' implies a role in an association between a Student and a Course
  - 'preferred section' implies a role in an association between a Student and a Course

Eliminating all of these role designations, we are left with only three terms: 'class', 'course', and 'section'. Before we hastily eliminate all but one of these as synonyms, let's think carefully about what real-world concepts we're trying to represent.

- The notion that we typically associate with the term 'course' is that of a semester-long series of lectures, assignments, exams, etc. that all relate to a particular subject area, and which are a unit of education toward earning a degree. For example, Beginning Math is a course.

- The terms 'class' and 'section', on the other hand, generally refer to the offering of a particular course in a given semester on a given day of the week and at a given time of day. For example, the course Math 101 is being offered this coming Spring semester as three classes/sections:
  - Section 1, which meets Tuesdays from 4 – 6 PM
  - Section 2, which meets Wednesdays from 6 – 8 PM
  - Section 3, which meets Thursdays from 3 – 5 PM

- There is a one-to-many association between Course and Class/Section. The same course is offered potentially many times in a given semester and over many semesters during the 'lifetime' of the course.

Therefore, 'course' and 'class/section' truly represent different abstractions, and we'll keep **both** concepts in our candidate class list. Since 'class' and 'section' appear to be synonyms, however, we need to choose one term and discard the other. Our inclination would be to keep 'class' and discard 'section', but in order to avoid confusion when referring to a class called Class (!) we'll opt for 'section' instead.

To make matters worse, there is also a built in Java class called `Class`! We'll see it in use in Chapter 13.

A list of candidate classes has begun to emerge from the fog! Here is our remaining 'short list' (please disregard the trailing symbols (*, +) for the moment – we will explain their significance shortly):

- course
- day of week*
- degree*
- email message+
- plan of study
- professor
- room*
- schedule of classes+
- seating capacity*
- section
- semester*
- student
- time of day*
- transcript
- (first-come, first-served) wait list

Not all of these will necessarily survive to the final model, however, as we're going to scrutinize each one very closely before deeming it worthy of implementation as a class. One classic test for determining whether or not an item can stand on its own as a class is to ask the questions:

- Can I think of any attributes for this class?
- Can I think of any services that would be expected of objects belonging to this class?

One example is the term 'room': we could invent a `Room` class as follows:

```
class Room {
  // Attributes.
  int roomNo;
  String building;
  int seatingCapacity;
  // etc.
}
```

or we could simply represent a room location as a `String` attribute of the `Section` class:

```
class Section {
  // Attributes
  Course offeringOf;
  String semester;
  char dayOfWeek;  // 'M', 'T', 'W', 'R', 'F'
  String timeOfDay;
  String classroomLocation; // building name and room name:  e.g.,
                            //  "Government Hall Room 105"
  // etc.
}
```

Which approach to representing a room is preferred? It all depends on whether or not a room needs to be a focal point of our application. If the SRS were meant to do 'double duty' as a Classroom Scheduling System, then we may indeed wish to instantiate `Room` objects so as to be able to ask them to perform such services as printing out their weekly usage schedules or telling us their seating capacities. However, since these services were not mentioned as requirements in the SRS specification, we'll opt for making a room designation a simple `String` attribute of the `Section` class. We reserve the right, however, to change our minds about this later on; it's not unusual for some items to 'flip flop' over the lifecycle of a modeling exercise between being classes on their own vs. being represented as simple attributes of other classes.

Following a similar train of thought for all of the items marked with an asterisk (*) in the candidate class list above, we'll opt to reflect them all as attributes rather than making them classes of their own:

- 'day of week' will be incorporated as either a `String` or `char` attribute of the `Section` class;
- 'degree' will be incorporated as a `String` attribute of the `Student` class;
- 'seating capacity' will be incorporated as an `int` attribute of the `Section` class;
- 'semester' will be incorporated as a `String` attribute of the `Section` class; and
- 'time of day' will be incorporated as a `String` attribute of the `Section` class.

When we are first modeling an application, we want to focus exclusively on functional requirements at the exclusion of technical requirements, as defined in Chapter 9; this means that we need to avoid getting into the technical details of how the system is going to function behind the scenes. Ideally, we want to focus solely on what are known as **domain classes** – that is, abstractions that an end user will recognize, and which represent 'real world' entities – and to avoid introducing any extra classes that are used solely as behind-the-scenes 'scaffolding' to hold the application together, known alternatively as **implementation classes** or **solution space classes**. Examples of the latter would be the creation of a collection object to organize and maintain references to all of the Professor objects in the system, or the use of a dictionary to provide a way to quickly find a particular Student object based on his/her student ID number. We will talk more about solution space objects in Part 3 of the book; for the time being, the items flagged with a plus sign (+) in the candidate class list above – email message, schedule of classes – seem arguably more like implementation classes than domain classes:

- An email message is typically a **transient** piece of data, not unlike a popup message that appears on the screen while using an application: it gets sent **out** of the SRS system, and after it is read by the recipient, we have no control over whether the email is retained or deleted. It is unlikely that the SRS is going to archive copies of all email messages that have been sent – there certainly was no requirement to do so – so we won't worry about modeling them as objects at this stage in our analysis.

Email messages will resurface in Chapter 11, when we talk about the behaviors of the SRS application, because **sending** an email message is definitely an important **behavior**; but, emails do not constitute an important **structural** piece of the application, so we don't want to introduce a class for them at this stage in the modeling process. When we actually get to programming the system, we might indeed create an `EmailMessage` class in Java, but it needn't be modeled as a domain class. (If, on the other hand, we were modeling an email messaging system in anticipation of building one, then `EmailMessage` would indeed be a key class in our model.)

- We could go either way with the schedule of classes – include it as a candidate class, or drop it from our list. The schedule of classes, as a single object, may not be something that the user will manipulate directly, but there will be some notion behind the scenes of a schedule of classes **collection** controlling which Section objects should be presented to the user as a GUI pick list when he/she registers in a given semester. We'll omit Schedule of Classes from our candidate class list for now, but can certainly revisit our decision as the model evolves.

Determining whether or not a class constitutes a domain class instead of an implementation class is admittedly a gray area, and either of the above candidate class 'rejects' could be successfully argued into or out of the list of core domain classes for the SRS. In fact, this entire exercise of identifying classes hopefully illustrates a concept that was first introduced in Chapter 2; because of its importance, we'll repeat it again:

> '[…] developing an appropriate model for a software system is perhaps the most difficult aspect of software engineering, because:
>
> ***There are an unlimited number of possibilities.*** Abstraction is to a certain extent in the eye of the beholder: several different observers working independently are almost guaranteed to arrive at different models. Whose is the best? Passionate arguments have ensued!
>
> To further complicate matters, ***there is virtually never only one 'best' or 'correct' model***, only 'better' or 'worse' models relative to the problem to be solved. The same situation can be modeled in a variety of different, equally valid ways. […]
>
> […] There is no 'acid test' to determine if a model has adequately captured all of a user's requirements.'

As we continue along with our SRS modeling exercise, and particularly as we move from modeling to implementation in Part 3 of the book, we'll have many opportunities to rethink the decisions that we've made here. The key point to remember is that the model is not 'cast in stone' until we actually begin programming, and even then, if we've used objects wisely, the model can be fairly painlessly modified to handle most new requirements. Think of a model as being formed out of modeling clay: we'll continue to reshape it over the course of the analysis and design phases of our project until we're satisfied with the result.

Meanwhile, back to the task of coming up with a list of candidate classes for the SRS. The terms that have survived our latest round of scrutiny are as follows:

- Course
- Plan of Study
- Professor
- Section
- Student
- Transcript
- Wait List

Let's examine 'Wait List' one last time. There is indeed a requirement for the SRS to maintain a student's position on a first-come, first-served wait list. But, it turns out that this requirement can actually be handled through a combination of an association between the `Student` and `Section` classes, plus something known as an **association class** which we'll learn about later in this chapter. This would not be immediately obvious to a beginning modeler, and so we'd fully expect that the Wait List class might make the final cut as a suggested SRS class. But, we're going to assume that we have an experienced object modeler on the team, who convinces us to eliminate the class; we'll see that this was a suitable move when we complete the SRS class diagram at the end of the chapter.

So, we'll settle on the following list of classes, based on our noun phrase analysis of the SRS specification:

- Course
- Plan of Study
- Professor
- Section
- Student
- Transcript

Revisiting the Use Cases

One more thing that we need to do before we deem our class list good to go is to revisit our use cases – in particular, the actors – to see if any of them ought to be added as classes. You may recall that we identified seven potential actors for the SRS in Chapter 9:

- Student
- Faculty
- Department Chair

- Registrar
- Billing System
- Admissions System
- Classroom Scheduling System

Do any of *these* deserve to be modeled as classes in the SRS? Here's how to make that determination: if any user associated with any actor type 'A' is going to need to manipulate (access or modify) information concerning an actor type 'B' when 'A' is logged onto the SRS, then 'B' needs to be included as a class in our model. This is best illustrated with a few examples.

- When a student logs onto the SRS, might he or she need to manipulate information about faculty? Yes; when a student selects an advisor, for example, he/she might need to view information about a variety of faculty members in order to choose an appropriate advisor. So, the Faculty actor role must be represented as a class in the SRS; and, indeed, we have already designated a `Professor` class, so we're covered there. But, student users are not concerned with Department Chairs per se.

- Following the same logic, we'd need to represent the Student actor role as a class because when professors log onto the SRS, they will be manipulating `Student` objects when printing out a course roster or assigning grades to students, for example. Since Student already appears in our candidate class list, we're covered there, as well.

- When **any** of the actors – faculty, students, the Registrar, the Billing System, the Admissions System, or the Classroom Scheduling System – access the SRS, will there be a need for any of them to manipulate information about the Registrar? No, at least not according to the SRS requirements that we've seen so far. Therefore, we needn't model the Registrar actor role as a class.

- The same holds true for the Billing, Admissions, and Classroom Scheduling Systems: they require 'behind the scenes' access to information managed by the SRS, but nobody logging onto the SRS expects to be able to manipulate any of these three systems directly, so they needn't be represented by domain classes in the SRS.

  *Again, when we get to **implementing** the SRS in code, we may indeed find it appropriate to create 'solution space' Java classes to represent interfaces to these other systems; but, such classes don't belong in a **domain** model of the SRS.*

Therefore, our proposed candidate class list remains unchanged after revisiting all actor roles:

- Course
- Plan of Study
- Professor
- Section
- Student
- Transcript

Is this a perfect list? No – there is no such thing! In fact, before all is said and done, the list may – and in fact probably will – evolve in the following ways:

- We may add classes later on: terms we eliminated from the specification, or terms that don't even appear in the specification, but which we will unearth through continued investigation.
- We may see an opportunity to generalize – that is, we may see enough commonality between two or more classes' respective attributes, methods, and/or relationships with other classes to warrant the creation of a common superclass.
- In addition, as we mentioned earlier, we may rethink our decisions regarding representing some concepts as simple attributes (semester, room, etc.) instead of as full-blown classes, and vice versa.

The development of a candidate class list is, as we've tried to illustrate, fraught with uncertainty. For this reason, **it is important to have someone experienced with object modeling available to your team when embarking on your first object modeling effort.** Most experienced modelers don't use the rote method of noun phrase analysis to derive a candidate class list; such folks can pretty much review a specification and directly pick out significant classes, in the same way that a professional jeweler can easily choose a genuine diamond from among a pile of fake gemstones. Nevertheless, what does 'significant' really mean? That's where the 'fuzziness' comes in! It is impossible to define precisely what makes one concept significant and another less so. We've tried to illustrate some rules of thumb by working through the SRS example, but you ultimately need a qualified mentor to guide you until you develop – and trust – your own intuitive sense for such things.

The bottom line, however, is that even expert modelers can't really confirm the appropriateness of a given candidate class until they see its proposed use in the full context of a class diagram that also reflects associations, attributes, and methods, which we'll do later in this chapter.

Producing a Data Dictionary

Early on in our analysis efforts, it is important that we clarify and begin to document our use of terminology. A **data dictionary** is ideal for this purpose. For each candidate class, the data dictionary should include a simple definition of what this item means in the context of the model/system as a whole; include an example if it helps to illustrate the definition.

Here is our complete SRS data dictionary so far:

> **Course**: a semester-long series of lectures, assignments, exams, etc. that all relate to a particular subject area, and which are typically associated with a particular number of credit hours; a unit of study toward a degree. For example Beginning Objects is a required **course** for the Master of Science Degree in Information Systems Technology.
>
> **Plan of Study**: a list of the **courses** that a student intends to take to fulfill the **course** requirements for a particular degree.

Professor: a member of the faculty who teaches **sections** and/or advises **students**.

Section: the offering of a particular **course** during a particular semester on a particular day of the week and at a particular time of day (for example, **course** 'Beginning Objects' as taught in the Spring 2001 semester on Mondays from 1:00 – 3:00 PM).

Student: a person who is currently enrolled at the university and who is eligible to register for one or more **sections**.

Transcript: a record of all of the **courses** taken to date by a particular **student** at this university, including which semester each **course** was taken in, the grade received, and the credits granted for the **course**, as well as reflecting an overall total number of credits earned and the **student's** grade point average (GPA).

Note that it is permissible, and in fact encouraged, for the definition of one term to include one or more of the other terms; when we do so, we highlight the latter in **bold text**.

The data dictionary joins the set of other SRS narrative documents as a subsequent source of information about the model. As our model evolves, we will expand the dictionary to include definitions of attributes, associations, and methods.

It is a good idea to also include the dictionary definition of a class as a header comment in the Java code representing that class. Make sure to keep this in-line documentation in synch with the external dictionary definition, however.

Determining Associations Between Classes

Once we've settled on an initial candidate class list, the next step is to determine how these classes are interrelated. To do this, we go back to our narrative documentation set (which has grown to consist of the SRS requirements specification, use cases, and data dictionary) and study verb phrases this time. Our goal in looking at verb phrases is to choose those that suggest structural relationships, as were defined in Chapter 5 – associations, aggregations, and inheritance – but to eliminate or ignore those that represent (transient) actions or behaviors. (We'll focus on behaviors, but from the standpoint of use cases, in Chapter 11.)

- For example, the specification states that a student 'chooses a faculty advisor'. This is indeed an action, but the result of this action is a lasting structural relationship between a professor and a student, which can be modeled via the association 'a Professor *advises* a Student'.
- As a student's advisor, a professor also meets with the student, answers the student's questions, recommends courses for the student to take, approves their plan of study, etc. - these are behaviors on the part of a professor acting in the role of an advisor, but do not directly result in any new relationships being formed between objects.

Let's try the verb phrase analysis approach on the requirements specification. We've highlighted all relevant verb phrases below (note that we omitted such obviously irrelevant verb phrases as 'We have been asked to develop an automated SRS …'):

> We have been asked to develop an automated Student Registration System (SRS) for the university. This system will **enable students to register** on-line **for courses** each semester, as well as **tracking their progress toward completion of their degree**.
>
> When a student first **enrolls at the university**, he/she uses the SRS to **set forth a plan of study** as to which **courses he/she plans on taking** to **satisfy a particular degree program**, and **chooses a faculty advisor**. The SRS will **verify whether or not the proposed plan of study satisfies the requirements of the degree that the student is seeking**.
>
> Once a **plan of study has been established**, then, during the registration period preceding each semester, students are able to **view the schedule of classes** on line, and **choose whichever classes he/she wishes to attend**, **indicating the preferred section** (day of the week and time of day) if the **class is offered by more than one professor**. The SRS will **verify whether or not the student has satisfied the necessary prerequisites** for each requested course by **referring to the student's on-line transcript** of courses completed and grades received (the **student may review his/her transcript** on-line at any time).
>
> Assuming that (a) the **prerequisites for the requested course(s) are satisfied**, (b) the **course(s) meet(s) one of the student's plan of study requirements**, and (c) **there is room available** in each of the class(es), the **student is enrolled in the class(es).**
>
> If (a) and (b) are satisfied, but (c) is not, the **student is placed on a first-come, first-served wait list**. If a **class/section that he/she was previously wait-listed for becomes available** (either because some other **student has dropped the class** or because the **seating capacity for the class has been increased**), the **student is automatically enrolled in the waitlisted class**, and an **email message** to that effect **is sent** to the student. It is his/her responsibility to **drop the class** if it is no longer desired; otherwise, **he/she will be billed for the course**.
>
> **Students may drop a class** up to the end of the first week of the semester in which the **class is being taught**.

Let's scrutinize a few of these:

- 'students […] register […] for courses': although the act of registering is a behavior, the end result is that a static relationship is created between a Student and a Section, as represented by the association 'a Student *registers* for a Section'. (Note that the specification mentions registering for 'courses', not 'sections', but as we stated in our data dictionary, a Student registers for concrete Sections as embodiments of Courses. Keep in mind when reviewing a specification that the English language is imprecise, and that as a result we have to read between the lines as to what the author really meant in every case. (If we are going to be the ones to write the specification, here is an incentive to keep the language as clear and concise as possible!)

- '[students track] their progress toward completion of their degree': again, this is a behavior, but it nonetheless implies a structural relationship between a Student and a Degree. However, recall that we didn't elect to represent Degree as a class – we opted to reflect it as a simple `String` attribute of the `Student` class – and so this suggested relationship is immaterial with respect to the candidate class list that we've developed.
- 'student first enrolls at the university': this is a behavior that results in a static relationship between a Student and the University; but, we deemed the notion of 'university' to be external to the system and so chose not to create a University class in our model. So, we disregard this verb phrase, as well.
- '[student] sets forth a plan of study': this is a behavior that results in the static relationship 'a Student *pursues/observes* a Plan of Study'.
- 'students are able to view the schedule of classes on-line': this is strictly a transient behavior of the SRS; no lasting relationship results from this action, so we disregard this verb phrase.

and so on.

Another complementary technique for both determining and recording what the relationships between classes should be is to create an *n* x *n* **association matrix**, where *n* represents the number of candidate classes that we've identified. Label the rows and the columns with the names of the classes, as shown for the empty matrix below.

	Section	**Course**	**Plan of Study**	**Professor**	**Student**	**Transcript**
Section						
Course						
Plan of Study						
Professor						
Student						
Transcript						

Then, complete the matrix as follows:

- In each cell of the matrix, list all of the associations that you can identify between the class named at the head of the row and the class named at the head of the column. For example, in the cell highlighted in the diagram on the next page at the intersection of the Student 'row' and the Section 'column', we have listed three potential associations:
  - A Student is waitlisted for a Section.
  - A Student is registered for a Section (this could be alternatively phrased as 'a Student is currently attending a Section').
  - A Student has previously taken a Section: this third association is important if we plan on maintaining a history of all of the classes that a student has ever taken in their career as a student, which we must do if we are to prepare a student's transcript on-line.

    (As it turns out, we'll be able to get by with a single association that does 'double duty' for the latter two of these, as we'll see later on in this chapter.)

- Mark a cell with an '✗' if there are no known relationships between the classes in question, or if the potential relationships between the classes are irrelevant. For example, we've marked the cells representing the intersection between Professor and Course with an '✗', even though there is an association possible – 'a Professor *is qualified to teach* a Course' – because it isn't relevant to the mission of the SRS.
- We mentioned in Chapter 4 that all associations are inherently bidirectional. This implies that if a cell in row *j*, column *k* indicates one or more associations, then the cell in row *k*, column *j* should reflect the reciprocal of these relationships. E.g. since the intersection of the Plan of Study 'row' and the Course 'column' indicates that 'a Plan of Study *calls for* a Course', then the intersection of the Course 'row' and the Plan of Study 'column' must indicate that 'a Course *is called for by* a Plan of Study'.

  It's not always practical to state the reciprocal of an association; for example, our association matrix shows that 'a Student *plans to take* a Course', but to try to state its reciprocal – 'a Course *is planned to be taken* by a Student' – is quite awkward. In such cases where a reciprocal association would be awkward to phrase, simply indicate its presence with a '✓'.

	Section	**Course**	**Plan of Study**	**Professor**	**Student**	**Transcript**
Section	✗	instance of	✗	is taught by	✓	included in
Course	✓	prerequisite for	is called for by	✗	✓	✗
Plan of Study	✗	calls for	✗	✗	observed by	✗
Professor	teaches	✗	✗	✗	advises; teaches	✗
Student	registered for; waitlisted for; has previously taken	plans to take	observes	is advised by; studies under	✗	owns
Transcript	includes	✗	✗	✗	belongs to	✗

We'll be portraying these associations in graphical form shortly! For now, we'd want to go back and extend our data dictionary to explain what each of these associations means; here's one such example:

'**calls for** (a Plan of Study calls for a Course): In order to demonstrate that a **student** will satisfy the requirements for his/her chosen degree program, the **student** must formulate a **plan of study**. This **plan of study** lays out all of the **courses** that a **student** intends to take, and possibly specifies in which semester the **student** hopes to complete each **course**.'

Identifying Attributes

To determine what the attributes for each of our domain classes should be, we make yet another pass through the requirements specification looking for clues. We already stumbled upon a few attributes earlier, when we weeded out some nouns/noun phrases from our candidate class list:

- For the `Section` class, we identified 'day of week', 'room', 'seating capacity', 'semester', and 'time of day' as attributes
- For the `Student` class, we identified 'degree' as an attribute

We can also bring any prior knowledge that we have about the domain into play when assigning attributes to classes. Our knowledge of the way that universities operate, for example, suggests that all Students will need some sort of student ID number as an attribute, even though this is not mentioned anywhere in the SRS specification. We can't be sure whether this particular university assigns an arbitrary student ID number, or whether the default is to use a student's social security number (SSN) as his/her ID; these are details that we'd have to go back to our end users for clarification on.

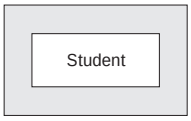
Finally, we can also look at how similar information has been represented in existing legacy systems for clues as to what a class's attributes should be. For example, if a Student Billing System already exists at the university based on a relational database design, we might wish to study the structure of the relational database table housing Student information. The columns that have been provided in that table – name, address, birthdate, etc. – are logical attribute choices.

UML Notation: Modeling the Static Aspects of an Abstraction

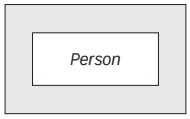
Now that we have a much better understanding about the static aspects of our model, we're ready to portray these in graphical fashion to complement the narrative documentation that we've developed for the SRS. We'll be using the Unified Modeling Language (UML) to produce a **class diagram**; here are the rules for how various aspects of the model are to be portrayed.

Classes, Attributes, and Methods

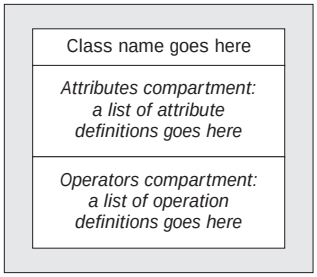
We represent classes as rectangles. When we first conceive of a class – before we know what any of its attributes or methods are going to be – we simply place the class name in the rectangle, for example:



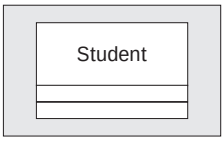
An *abstract* class is denoted by presenting the name in italics, for example:



When we are ready to reflect the attributes and/or methods of a class, we divide the class rectangle into three **compartments** – the class name compartment, the attributes compartment, and the operations compartment – as shown below. Note that we prefer the nomenclature of 'operations' versus 'methods' to reinforce the notion that the diagram that we are producing is intended to be programming language independent.

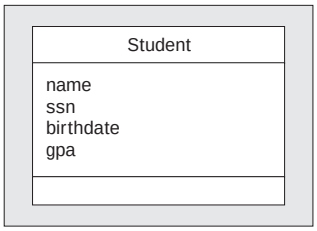


Some CASE tools automatically portray all three compartments when a class is first created, even if we haven't specified any attributes or methods yet:

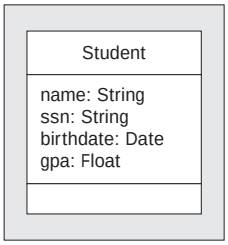


As we begin to identify what the attributes and/or operations need to be for a particular class, we can add these to the diagram in as much or as little detail as we care to.

- We may choose simply to list attribute names:



or we may specify their names along with their types:

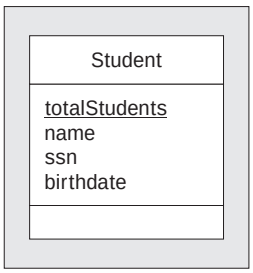


We may even wish to specify an initial starting value for an attribute, as in:

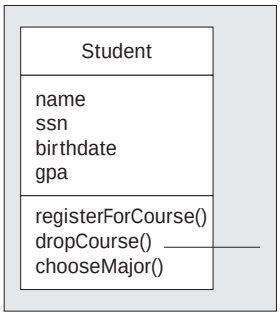
```
gpa : Float = 0.0
```

although this is less commonplace.

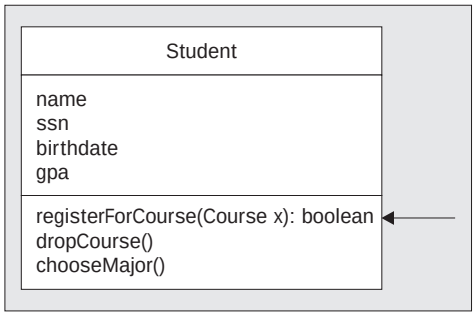
- Static attributes are identified as such by underlining them:



- We may choose simply to list method names (typically with the suffix '()' attached to imply their nature as methods) in the operations compartment of a class rectangle:



or we may optionally choose to use an expanded form of operation definition as we have for the `registerForCourse()` method below:



Note that the formal syntax for operation specifications in a UML class diagram:

methodName(*optional argument list*) : *returnType*

differs from the syntax that we are used to seeing for Java method signatures:

returnType methodName(*optional argument list*);

Here is a situation where using a CASE tool may restrict your flexibility. The rationale for making these operation definitions generic instead of representing them as language specific method signatures is so that the same model may be rendered in any of a variety of target programming languages. It can be argued, however, that there is nothing inherently better or clearer about the first form versus the second. Therefore, if you know that you are going to be programming in Java, it might make sense to reflect standard Java method signatures in your class diagram, if your object modeling tool will accommodate this.

It is often impractical to show all of the attributes and methods of every class in a class diagram, because the diagram will get so cluttered that it will lose its 'punch' as a communications tool. Consider the data dictionary to be the official, complete source of information concerning the model, and only reflect in the diagram those attributes and methods that are particularly important in describing the mission of each class. In particular, 'get' and 'set' methods are implied for all attributes, and should not be explicitly shown.

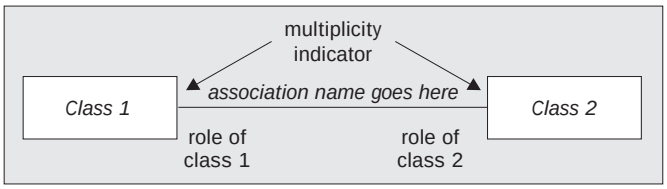
Also, just because the attribute or operation compartment of a class is empty, don't assume that there are no features of that type associated with a class; it may simply mean that the model is still evolving.

Relationships Between Classes

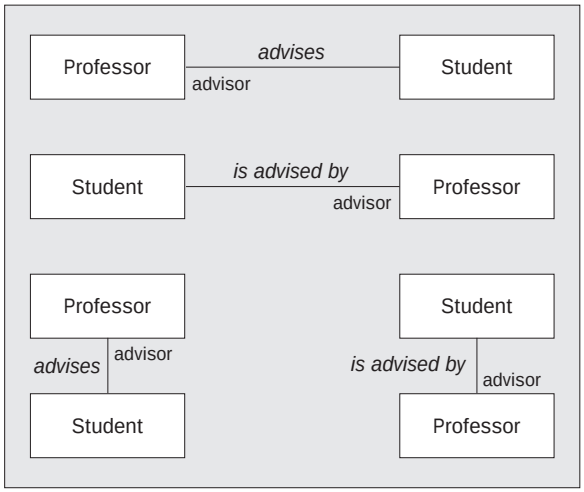
In Chapter 4, we defined several different types of structural relationship that may exist between classes – associations, aggregations (a specific type of association), and inheritance. Let's explore how each of these relationship types is represented graphically.

- Binary associations – in other words relationships between two different classes – are indicated by drawing a line between the rectangles representing the participating classes, and labeling the line with the name of the association. Role names can be reflected at either end of the association line if they add value to the model, but should otherwise be omitted.

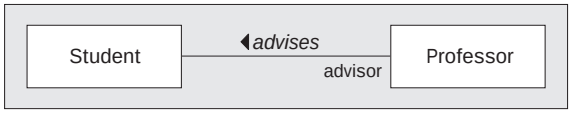
  We also mark each end of the line with the appropriate **multiplicity indicator**, to reflect whether the relationship is one-to-one, one-to-many, or many-to-many; we'll talk about how to do this a bit later in the chapter.



All associations are assumed to be bidirectional at this stage in the modeling effort, and it doesn't matter in which order the participating classes are arranged in a class diagram. So, to depict the association 'a Professor advises a Student', the following graphical notations are all considered equivalent:



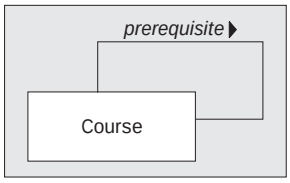
With OMT, the precursor notation to UML, we were instructed to label associations so that their names made sense when reading a diagram from left to right, top to bottom. There was thus an incentive to arrange classes in our diagram in whatever way would make association names less 'awkward'. In the examples above, placing the `Professor` class above or to the left of `Student` simplifies the association name. Achieving an optimal placement of classes for purposes of simplifying all of the association names in a diagram is often not possible in an elaborate diagram, however. Therefore, UML has introduced the simple convention of using a small arrowhead (▸) to reflect the direction in which the association name is to be interpreted, giving us a lot more freedom in how we place our class rectangles in a diagram:



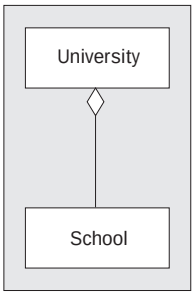
With UML, no matter how the above two rectangles are situated, we can still always label the association 'advises'.

It is easy to get caught up in the trap of trying to make diagrams 'perfect' in terms of how classes are positioned, to minimize crossed lines, etc. Try to resist the urge to do so early on, because the diagram will inevitably get changed many times before the modeling effort is finished.

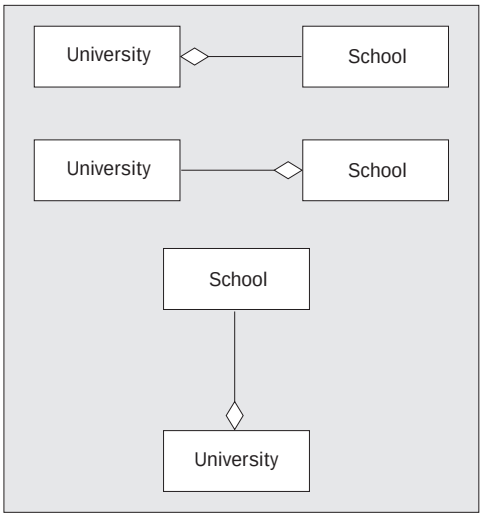
- Unary (reflexive) associations – i.e. relationships between two different objects belonging to the same class – are drawn with an association line that loops back to the same class rectangle from which it originates. For example, to depict the association 'a Course *is a prerequisite for* a (different) Course', we'd use the notation shown below:



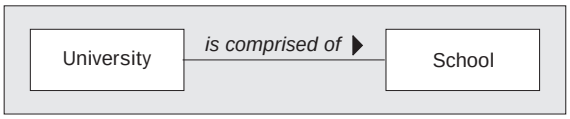
- Aggregation, which as we learned in Chapter 5 is really just an association that happens to imply containment, is differentiated from a 'normal' association by placing a diamond at the end of the association line that touches the 'containing' class. For example, to portray the fact that a University is comprised of Schools – the School of Engineering, School of Law, School of Medicine, etc. – we'd use the following notation:



An aggregation relationship can actually be oriented in any direction, as long as the diamond is properly anchored on the 'containing' class:



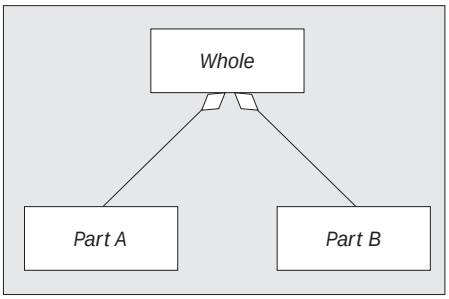
As we mentioned when we first introduced aggregation in Chapter 5, however, you can get by without ever using aggregation! To represent the above concept, we could have just created a simple association between the University and School classes, and labeled it 'is comprised of':



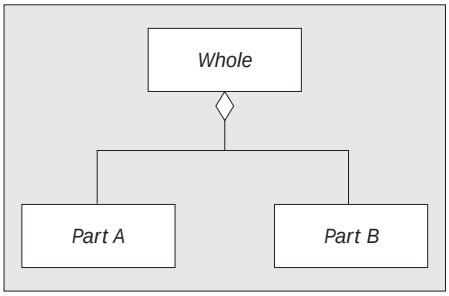
The decision of whether to use aggregation vs. plain association is subtle, because it turns out that both can be rendered in code in essentially the same way, as we'll see in Part 3 of the book.

Unlike association lines, which should always be labeled with the name of the association that they represent, aggregation lines are typically not labeled, since an aggregation by definition implies containment. However, if you wish to optionally label an aggregation line with a phrase such as 'consists of', 'is comprised of', 'contains', etc. you may certainly do so.

When two or more different classes represent 'parts' of some other 'whole', each 'part' is involved in a separate aggregation with the 'whole', as shown below:

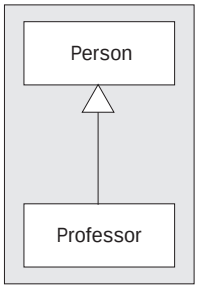


However, we often join such aggregation lines into a single structure that looks something like an organization chart, as follows:



Doing so is not meant to imply anything about the relationship of *Part A* to *Part B*; it is simply a way to clean up the diagram.

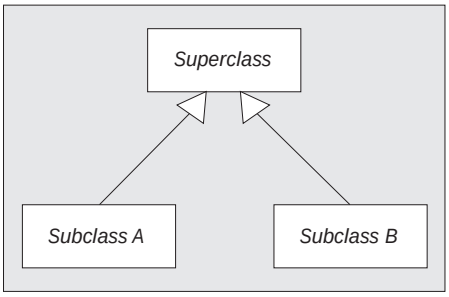
- Inheritance (generalization/specialization) is illustrated by connecting a subclass to its parent class with a line, and then marking the line with a triangle that touches the base class.



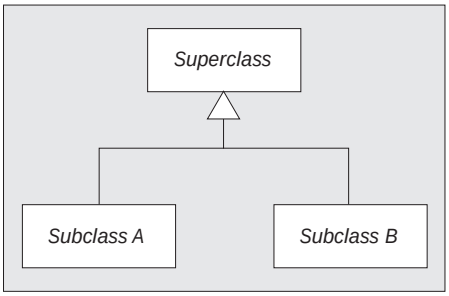
As with aggregation, the classes involved in an inheritance relationship can be portrayed with any orientation, as long as the triangle points to the superclass.

Unlike association lines, which must always be labeled, and aggregation lines, which needn't be labeled (but can be if you desire), inheritance lines should **not** be labeled, as they unambiguously represent the 'is a' relationship.

As with aggregation, when two or more different classes represent subclasses of the same parent class, each subclass is involved in a separate inheritance relationship with the parent:



but we often join the inheritance lines into a single structure, as follows:



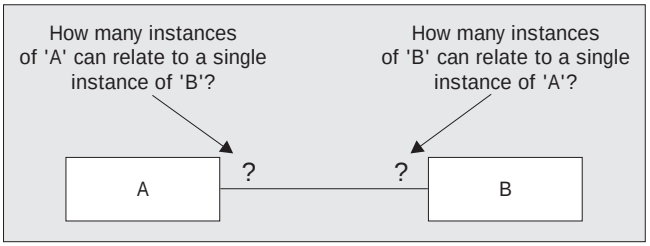
Doing so is not meant to imply anything different about the relationship of *Subclass A* to *Subclass B* as compared with the previous depiction – these classes are considered to be sibling classes with a common parent class in both cases. It is simply a way to clean up the diagram.

Reflecting Multiplicity

We learned in Chapter 5 that for a given association type X between classes A and B, the term 'multiplicity' refers to the number of instances of objects of type A that must/may be associated with a given instance of type B, and vice versa. When preparing a class diagram, we mark each end of an association line to indicate what its multiplicity should be from the perspective of an object belonging to the class at the other end of the line: in other words,

- We mark the number of instances of 'B' that can relate to a single instance of 'A' at *B*'s end of the line
- We mark the number of instances of 'A' that can relate to a single instance of 'B' at *A*'s end of the line

This is depicted in the figure below.



By way of review, given a single object belonging to class 'A', there are four different scenarios for how object(s) of type 'B' may be related to it:

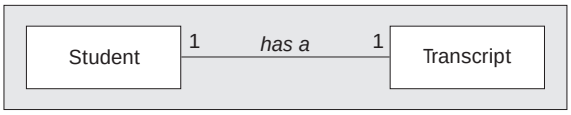
- The 'A' type object may be related to **exactly one** instance of a 'B' type object (**mandatory**), as in the situation 'a Student (A) has a Transcript (B)'.
- The 'A' type object may be related to **at most one** instance of a 'B' type object (**optional**), as in the situation 'a Professor (A) optionally chairs a Department (B)'.
- The 'A' type object may be related to **one or more** instances of a 'B' type object (**mandatory**), as in the situation 'a Department (A) employs many (one or more) Professors (B)'.
- The 'A' type object may be related to **zero or more** instances of a 'B' type object (**optional**), as in the situation 'a Student (A) is attending many (zero or more) Sections (B)'. (At our hypothetical university, a Student is permitted to take a semester off.)

With UML notation, multiplicity markings are as follows:

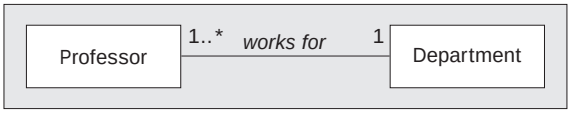
- 'Exactly one' is represented by the notation '1'.
- 'At most one' is represented by the notation '0..1', which is alternatively read as 'zero or one'.
- 'One or more' is represented by the notation '1..*'.
- 'Zero or more' is represented by the notation '0..*'.
- We use the notation '*' when we know that the multiplicity should be 'many' but we are not certain (or we don't care to specify) whether it should be 'zero or more' or 'one or more'.
- It is even possible to represent an arbitrary range of explicit numerical values *x..y*, such as using '3..7' to indicate, for example, that 'a Department employs no fewer than three, and no more than seven, Professors'.

Here are some UML examples:

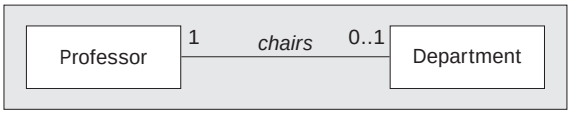
- 'A Student has exactly one Transcript, and a Transcript belongs to exactly one Student.'



- 'A Professor works for exactly one Department, but a Department has many (one or more) Professors as employees.'



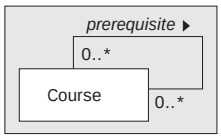
- 'A Professor optionally chairs at most one Department, while a Department has exactly one Professor in the role of chairman.'



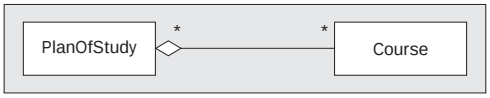
- 'A Student attends many (zero or more) Sections, and a Section is attended by many (zero or more) Students.'

A Section which continues to have zero students signed up to attend will most likely be cancelled; nonetheless, there is a period of time after a Section is first made available for enrollment via the SRS that it will have zero Students enrolled.

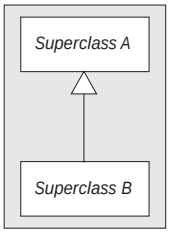
- 'A Course is a prerequisite for many (zero or more) Courses, and a Course can have many (zero or more) prerequisite Courses.'



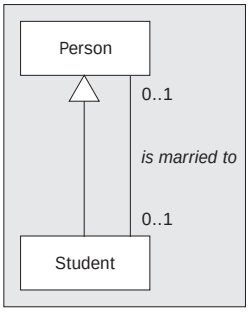
We reflect multiplicity on aggregations as well as on simple associations. For example, the following UML notation would be interpreted as follows: 'A (Student's) Plan of Study is comprised of many Courses; any given Course can be included in many different (Students') Plans of Study.'



It makes no sense to reflect multiplicity on inheritance relationships, however, because as we discussed in Chapter 4, inheritance implies a relationship between classes, but not between objects. That is, the notation:



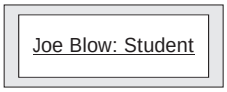
implies that any object belonging to *Subclass B* is also simultaneously an instance of *Superclass A* by virtue of the 'is a' relationship. If we wanted to illustrate some sort of relationship between different objects of types 'A' and 'B', e.g. 'a Person is married to a Student', we'd need to introduce a separate association between these classes independent of their inheritance relationship, as follows:



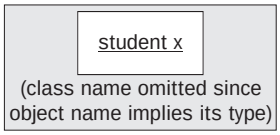
Object Diagrams

When describing how objects can interact, we sometimes find it helpful to sketch out a scenario of specific objects and their linkages, and for that we create an **object diagram**. An instance, or object, looks much the same as a class in UML notation, the main differences being that:

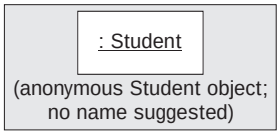
- We typically provide both the name of the object and its type, separated by a colon. We underline the text to emphasize that this is an object, not a class.



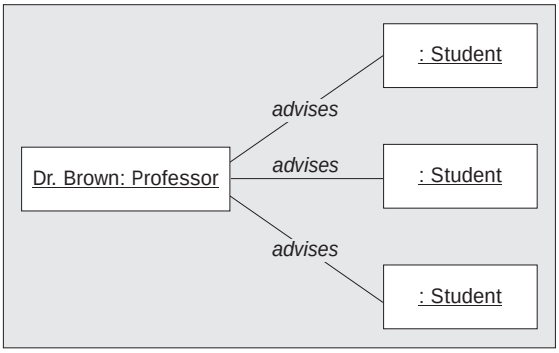
- The object's type may be omitted if it is obvious from the object's name; for example, the name 'student x' implies that the object in question belongs to the Student class.



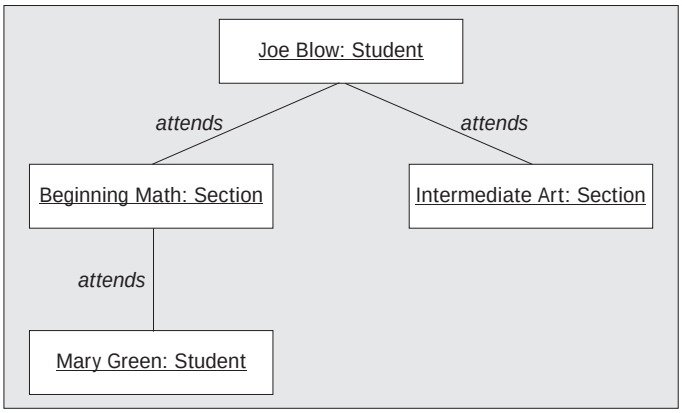
- Alternatively, the object's name may be omitted if we want to refer to a 'generic' object of a given type; such an object is known as an **anonymous object**. Note that we must precede the class name with a colon (:) in such a situation.



Therefore, if we wanted to indicate that Dr. Brown, a Professor, is the advisor for three students, we could create the following object diagram:

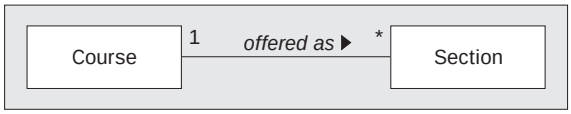


or, to reflect that a Student by the name of Joe Blow is attending two classes this semester, one of which is also attended by a Student named Mary Green:



Associations as Attributes

Given the following diagram of the association 'a Course is offered as a Section':



we see that a `Course` object can be related to many different `Section` objects, but that any one `Section` object can only be related to a single `Course` object. What does it mean for two objects to be related? It means that they maintain 'handles' on one another so that they can easily find one another to communicate and collaborate, a concept that we talked about in detail in Chapter 4. If we were to sketch out the attributes of the `Course` and `Section` classes based solely on the above diagram, we'd need to allow for these handles as reference variable attributes, as follows:

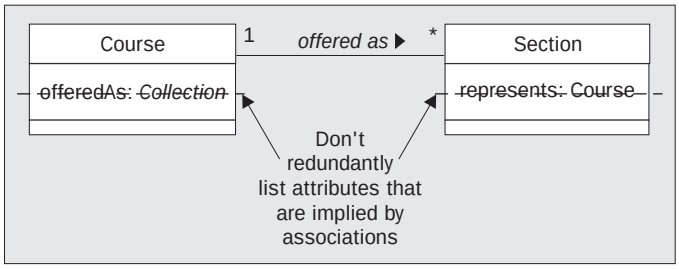
```
class Section {
  // Attributes.
  private Course represents;   // A 'handle' on a single related Course
                               // object.

  // etc.
}

class Course {
  // Attributes.
  private Collection offeredAs;        // A collection of related Section
                                       // object 'handles'.
  // etc.
}
```

So we see that the presence of an association between two classes A and B in a class diagram implies that class A *potentially* has an attribute declared to be either:

- A reference to a single object of type B
- A collection of references to many objects of type B

depending on the multiplicity involved, and vice versa. We say 'potentially' because, when we get to the point of actually programming this application, we may or may not wish to code this relationship bidirectionally, even though at the analysis stage all associations are presumed to be bidirectional. We'll talk about the pros and cons of doing so in Chapter 14.

Because the presence of an association line implies attributes as handles in both related classes, it is inappropriate to additionally list such attributes in the attribute compartment of the respective classes.

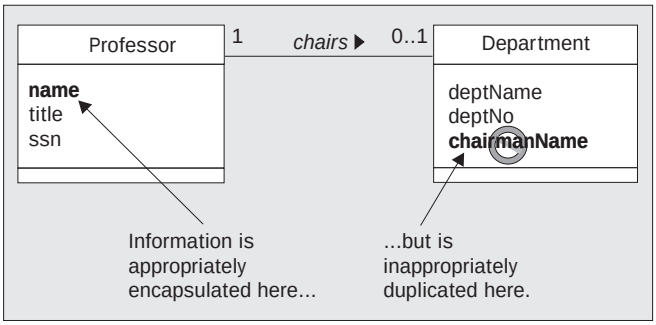


This is a mistake commonly made by beginners. The biggest resultant problem with doing so arises when using the code generation capability of a CASE tool: if the attribute is listed explicitly in a class's attributes compartment, and also implied by an association, it may appear in the generated code twice:

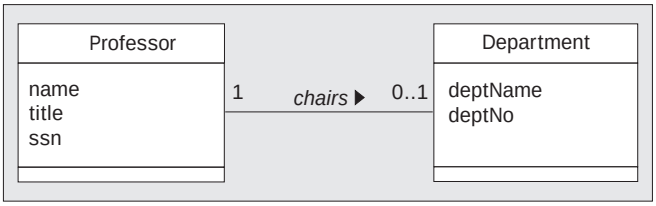
```
class Course {
  Collection offeredAs;   // by virtue of an explicit attribute
  Collection offered_as;  // by virtue of the association
  // etc.
}
```

Information 'Flows' Along the Association 'Pipeline'

Beginning modelers also tend to make the mistake of introducing undesired redundancy when it comes to attributes in general. In the following association diagram, we see that the '`name`' attribute of the `Professor` class is inappropriately mirrored by the '`chairmanName`' attribute of the `Department` class.



While it is true that a `Department` object needs to know the name of the `Professor` object that chairs that `Department`, it is inappropriate to explicitly create a `chairmanName` attribute to reflect this information. Because the `Department` object maintains a reference to its associated `Professor` object as an attribute, the `Department` has ready access to this information any time it needs it, simply by invoking the `Professor` object's `getName()` method. This piece of information is rightfully encapsulated in the `Professor` class, where it belongs, and should not be duplicated anywhere else. A corrected version of the preceding diagram is shown below, with the redundancy eliminated.

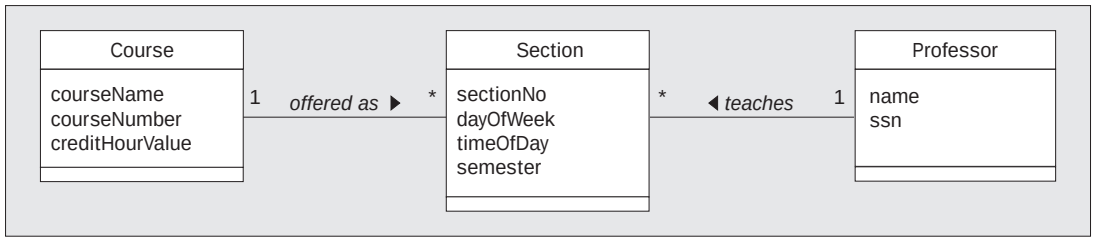


In essence, whenever we see an association/aggregation line in a diagram, we can think of this as a conceptual 'pipeline' across which information can 'flow' between related objects as needed.

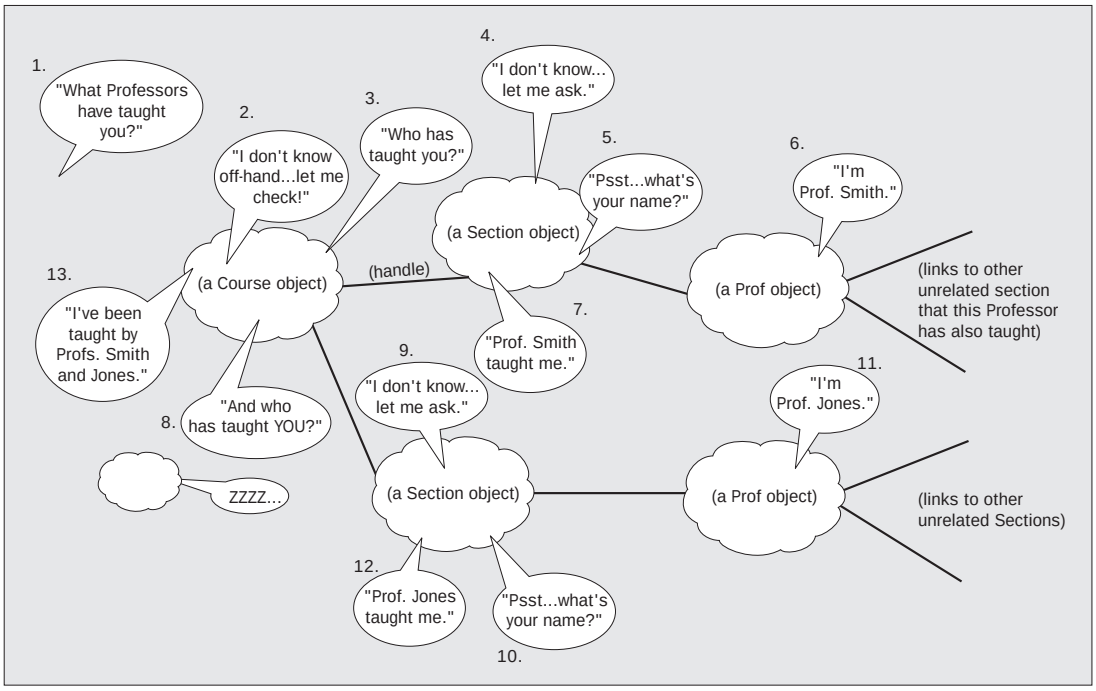
> *At the analysis stage, we don't worry about the visibility (public, private) of attributes, or of the directionality of associations; we'll assume that the values of all of the attributes reflected in a diagram are obtainable by calling the appropriate 'get' methods on an object.*

Sometimes, this 'pipeline' extends across multiple objects, as illustrated by the next example.

Here, we have a diagram involving three classes:



Let's say that someone wishes to obtain a list of all of the Professors who have ever taught the Course entitled 'Beginning Objects'. Because each `Course` object maintains a handle on all of its `Section` objects, past and present, the `Course` object representing 'Beginning Objects' can ask each of its `Section` objects the name of the `Professor` who previously taught, or is currently teaching, that `Section`. The `Section` objects, in turn, each maintain a handle on the `Professor` object who taught/teaches the `Section`, and can use the `Professor` object's `getName()` method to retrieve the name. So, information flows along the association 'pipeline' from the `Professor` objects to their associated `Section` objects and from there back to the `Course` object that we started with.

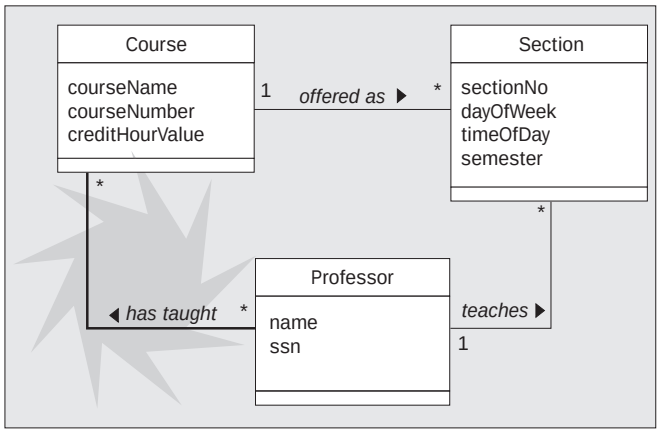


We'll learn a formal, UML-appropriate way to analyze and depict such 'object conversations' in Chapter 11.

We've modeled these three classes' attributes in code below, highlighting all of the association-driven attributes:

```
class Course {
  // Attributes.
  private Collection offeredAs;       // a collection of Section object
                                      // 'handles'
  private String courseName;
  private int courseNumber;
  private float creditHourValue;
  // etc.
}

class Section {
  // Attributes.
  private Course represents;   // a 'handle' on the related Course
                               // object
  private int sectionNo;
  private String dayOfWeek;
  private String timeOfDay;
  private String semester;
  private Professor taughtBy;  // a 'handle' on the related Prof. object

  // etc.
}

class Professor {
  private Collection sectionsTaught; // a collection of Section obj.
                                     // 'handles'
  private String name;
  private String ssn;

  // etc.
}
```

If we knew that the `Course` class was going to regularly need to know who all the Professors were that ever taught the course, we might decide to introduce the redundant association 'a Professor has taught a Course' into our diagram, as illustrated below:



This has the advantage of improving the speed with which a `Course` object can determine who has ever taught it. `Course` objects can now talk directly to `Professor` objects without using `Section` objects as go-betweens – but the cost of this performance improvement is that we've just introduced additional complexity to our application, reflected by the highlighted additions to the code below:

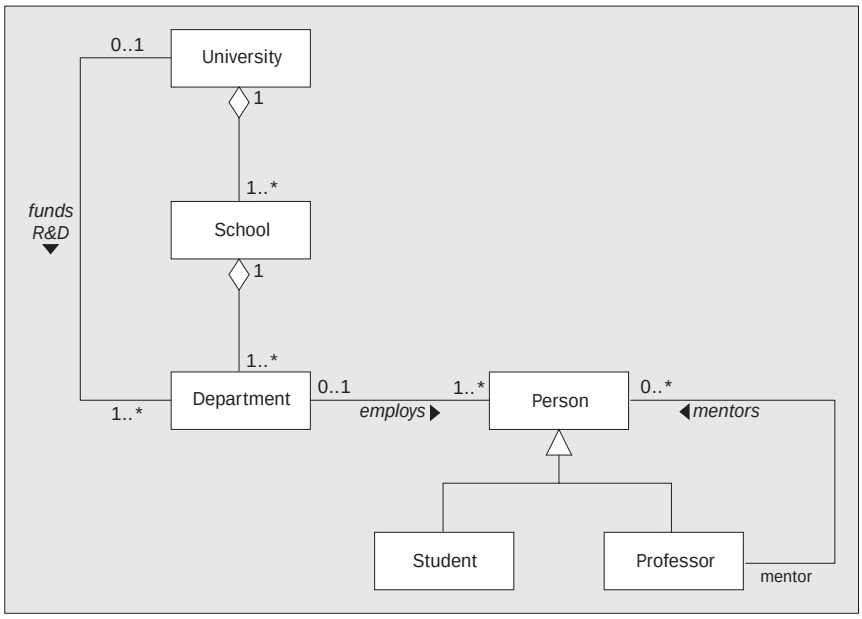
```
class Course {
  // Attributes.
  private Collection offeredAs;      // a collection of Section object
                                     // 'handles'
  private String courseName;
  private int courseNumber;
  private float creditHourValue;
  private Collection professors;    // a collection of Professor obj.
                                    // 'handles'
  // etc.
}

class Section {
  // Attributes.
  private Course represents;   // a 'handle' on the related Course
                               // object
  private int sectionNo;
  private String dayOfWeek;
  private String timeOfDay;
  private String semester;
  private Professor taughtBy;  // a 'handle' on the related Prof. object

  // etc.
}

class Professor {
  private Collection coursesTaught;    // a collection of Course obj.
                                       //'handles'
  private Collection sectionsTaught;   // a collection of Section obj.
                                       // 'handles'
  private String name;
  private String ssn;

  // etc.
}
```

By adding the redundant association, we now have extra work to do in terms of maintaining referential integrity: that is, if a different Professor is assigned to teach a particular Section, we have two links to update rather than one: the link between the Professor and the Section, and the link between the Professor and the Course.

We'll talk more in Part 3 of the book about the implications, from a coding standpoint, of making such tradeoffs. The bottom line, however, is that deciding which associations to include, and which to eliminate as derivable from others, is similar to the decision of which web pages you create a bookmark for in your web browser: you bookmark those that you visit frequently, and type out the URL long-hand for those that you only occasionally need to access. The same is true for object linkages: the decisions of which to implement depends on which 'communication pathways' through the application we're going to want to use most frequently. We'll get a much better sense of what these communication patterns are when we move on to modeling behaviors in Chapter 11.

'Mixing and Matching' Relationship Notations

It is possible to intertwine the various relationship types in some rather sophisticated ways. To appreciate this fact, let's study the following model to see what it is telling us:



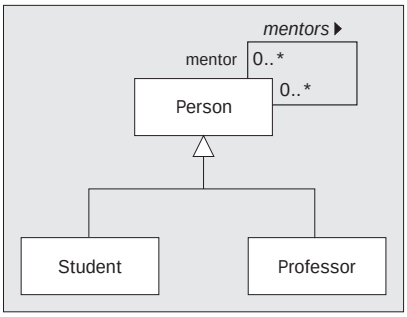
- First of all, we see some familiar uses of aggregation and inheritance.
  - The use of aggregation in the upper left hand corner of the diagram – a two-tier aggregation – communicates the facts that a University is comprised of one or more Schools, and that a School is comprised of one or more Departments, but that any one Department is only associated with a single School and any one School is only associated with a single University.
  - The use of inheritance in the lower right hand corner of the diagram indicates that Person is the common superclass for both Student and Professor. Alternatively, stated another way: that a Student is a Person, and a Professor is a Person.
- The first interesting use of the notation that we observe is that an association can be used to relate classes at differing levels in an aggregation, as in the use of the *funds R&D (Research & Development)* association used to relate the University and Department classes. This indicates that the University funds one or more Departments for research and development purposes, but that a given Department may or may not be funded for R&D.
- Next, we note the use of the *employs* association to relate the Department and Person classes, indicating that a Department employs one or more Persons, but that a given Person may work for only one Department, if indeed they work for any Department at all.

  Because Person is a superclass to both the Student and Professor subclasses, then by virtue of the 'is a' relationship, anything we can say about a Person must also be true of its subclasses.

Therefore, a given Student may optionally work for one Department, perhaps as a teaching assistant, and a given Professor may optionally work for one Department. Thus, associations/aggregations that a superclass participates in are inherited by its subclasses. (This makes sense, because we now know that associations are really rendered as attributes.)

- Also, because we can deduce (via the aggregation relationship) which School and University a given Department belongs to, the fact that a Person works for a given Department also implies which School and University the Person works for.
- Finally, we note that an association can be used to relate classes at differing levels in an inheritance hierarchy, as in the use of the **mentors** association to relate the Person and Professor classes. Here, we are stating that a Professor optionally mentors many Persons – Students and/or Professors – and conversely that a Person – either a Student or a Professor – is mentored by optionally many Professors. We label the end of the association line closest to the Professor class with the role designation 'mentor' to emphasize that Professors are mentors at the University, but that Persons in general (i.e. Students) are not.

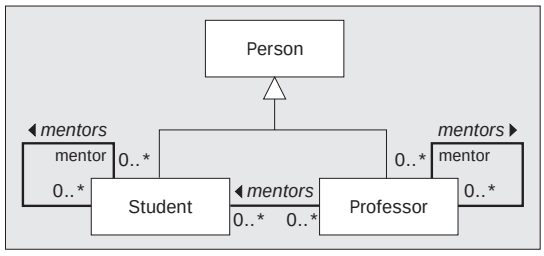
  What if we instead wanted to reflect the fact that both Students and Professors may hold the role of mentor? We could substitute a unary/reflexive association on the Person class, as follows:



  which, by virtue of inheritance, actually implies four relationship possibilities:

  - A Professor mentoring a Student
  - A Professor mentoring another Professor
  - A Student mentoring another Student
  - A Student mentoring a Professor (which is not very likely!)

  If we wanted to reflect that only the first three of these are possible, we'd have to resort to the rather more complex version shown below, where the three relationships of interest are all reflected as separate association lines (two reflexive, one binary):



As cumbersome as it is to change the diagram to reflect these refinements in our understanding, it would be orders of magnitude more painful to change in the software once the application had been coded.

Association Classes

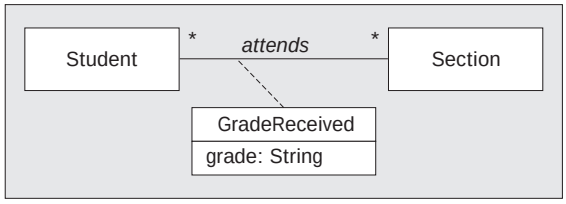
We sometimes find ourselves in a situation where we identify an attribute that is critical to our model, but which doesn't seem to nicely fit into any one class. As an example, let's revisit the association 'a Student attends a Section'. (Note that we are using the 'generic' **many** multiplicity adornment this time, a single asterisk (*), at each end of the association line.)

At the end of every semester, a student receives a letter grade for every section that he/she attended during that semester. We decide that the grade should be represented as a `String` attribute (e.g. 'A-', 'C+'). However, where does the 'grade' attribute belong?

- It's not an attribute of the `Student` class, because a student doesn't get a single overall grade for all of his/her coursework, but rather a different grade for each course attended.
- It's not an attribute of the `Section` class, either, because not all students attending a section typically receive the same letter grade.

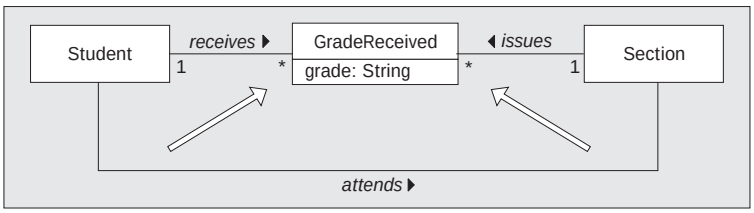
If we think about this situation for a moment, we realize that the grade is actually an attribute of the *pairing* of a given `Student` object with a given `Section`; that is, it is an attribute of the **link** that exists between these two objects.

With UML, we create a separate class, known as an **association class**, to house the attribute(s) belonging to the link between objects, and attach it with a dashed line to the association line as shown below.



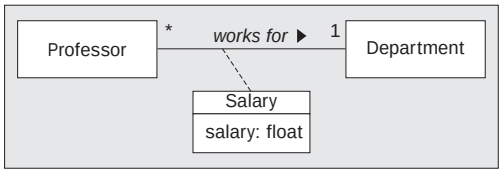
Any time you see an association class in a class diagram, realize that there is an alternative equivalent way to represent the same situation **without** using an association class.

- In the case of a many-to-many association involving an association class, you may split the many-to-many association into two one-to-many associations, inserting what was formerly the association class as a 'normal' class between the other two classes. Doing this for the preceding *attends* association, we wind up with the following equivalent alternative:

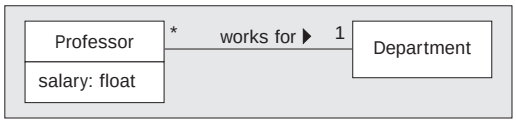


One important point to note is that the 'many' ends of these two new associations reside with the newly inserted class, because a Student *receives* many grades and a Section *issues* many grades.

- If we happen to have an association class for a one-to-many association, as in the *works for* association between Professor and Department:



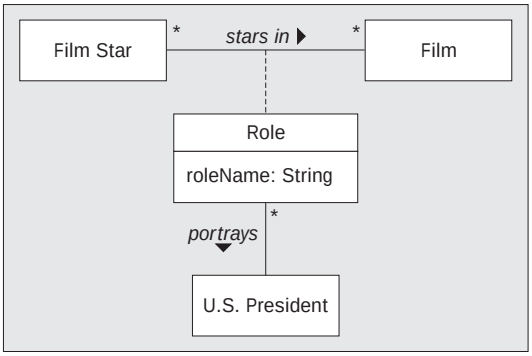
then the association class's attribute(s) can, in theory, be 'folded into' the class at the 'many' end of the association instead, and we can do away with the association class completely:



(With a one-to-one association, we can fold the association class's attributes into either class.)

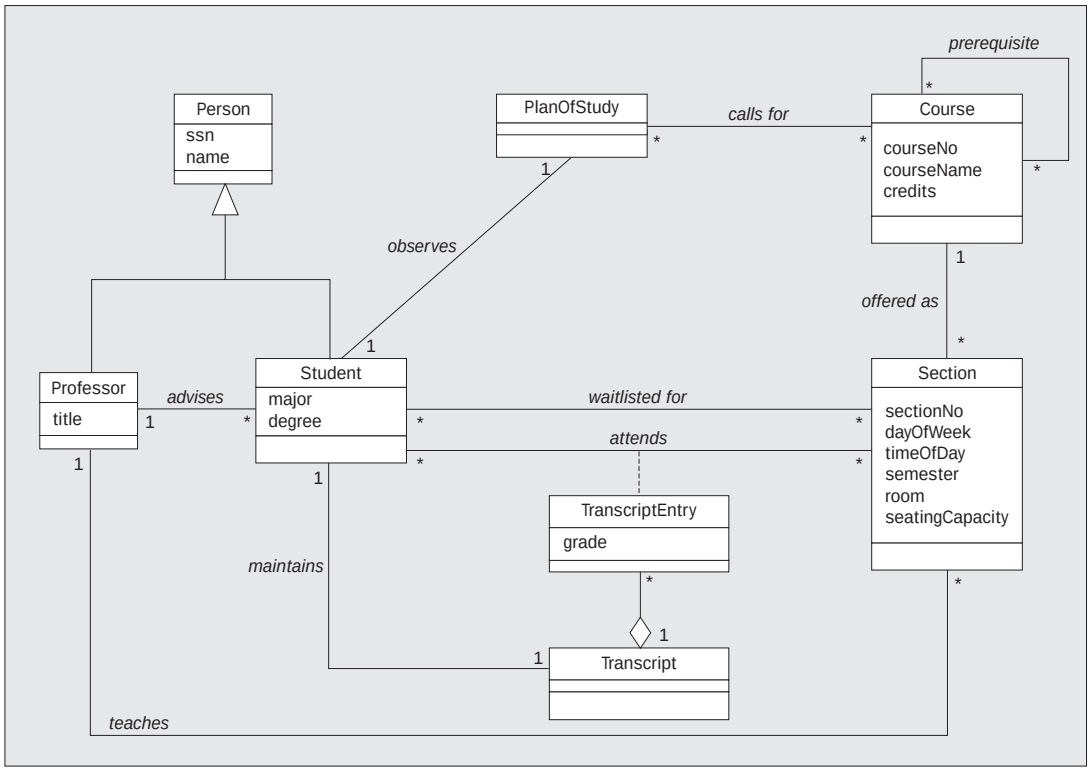
That being said, this practice of folding in association class attributes into one end of a one-to-many or one-to-one association is discouraged, however, because it reduces the amount of information communicated by the model. In the preceding example, the only reason that a Professor has a salary is because he/she works for a Department; knowledge of this 'cause and effect' connection between employment and salary is lost if the association class is eliminated as such from the model.

Note that association classes are 'normal' classes that may themselves participate in relationships with other classes. In the diagram to the right, for example, we show the association class '`Role`' participating in a one-to-many association with the class '`U.S. President`'; an example illustrating this model would be that 'Film Star Anthony Hopkins starred in the movie 'Nixon' in the role of Richard M. Nixon, thus portraying the **real** former U.S. President Richard M. Nixon'.



Our 'Completed' Student Registration System Class Diagram

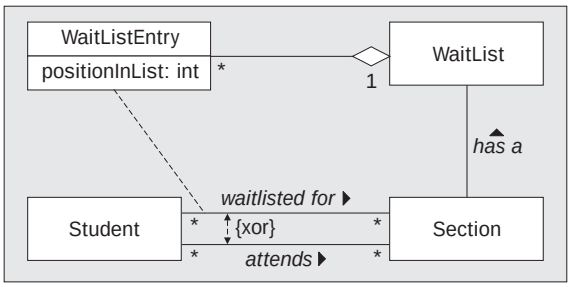
Applying all that we've learned in this chapter about static modeling, we produce the UML class diagram for the SRS shown below. Of course, as we've said repeatedly, this is not the only correct way to model the requirements, nor is it necessarily the 'best' model that we could have produced; but it is an accurate, concise, and correct model of the static aspects of the problem to be automated.



A few things worth noting:

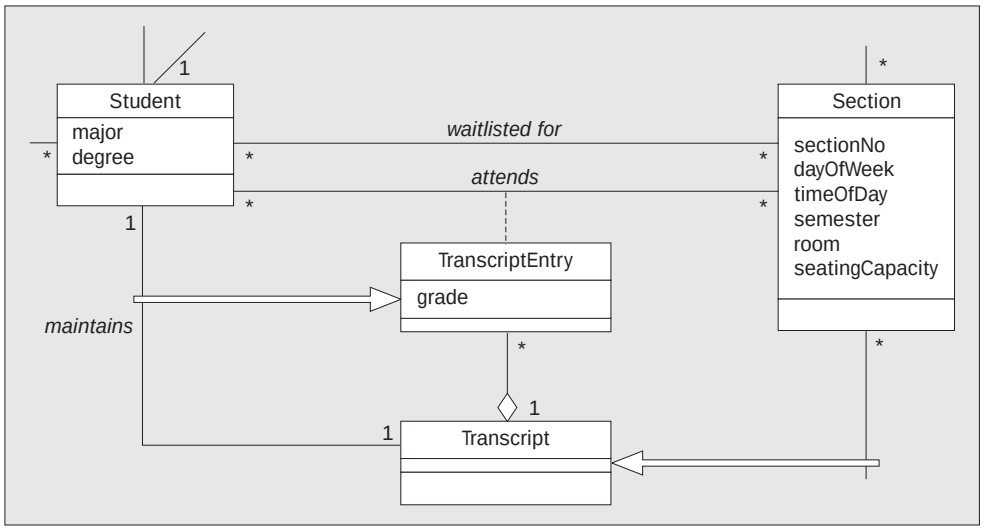
- We opted to use the 'generic' *many* notation (* for UML) rather than specifying 0..* or 1..*; this is often adequate during the initial modeling stages of a project.
- Note that we've reflected two separate many-to-many associations between the Student and Section classes: *waitlisted for* and *attends*. A given Student may be waitlisted for many different Sections, and they may be registered for/attending many other sections. What this model doesn't reflect is the fact that a Student may not simultaneously be attending and waitlisted for the *same* Section. Constraints such as these can be reflected as textual notes on the diagram, enclosed in curly braces, or can be omitted from the diagram but spelled out in the data dictionary. In the following diagram excerpt, we use the annotation '`{ xor }`' to represent an 'exclusive or' situation between the two associations: a Student can either be waitlisted for or attending a Section, but not both.

- As mentioned earlier in this chapter, we are able to get by with a single *attends* association to handle both the Sections that a Student is currently attending, as well as those that they have attended in the past. The date of attendance – past or present – is reflected by the 'semester' attribute of the `Section` class; also, for any courses that are currently in progress, the value of the 'grade' attribute of the `TranscriptEntry` association class would be as of yet undetermined.
- We could have also reflected an association class on the *waitlisted for* association representing a given Student's position in the wait list for a particular Section, and then could have gone on to model the notion of a `WaitList` as an aggregation of `WaitListEntry` objects:



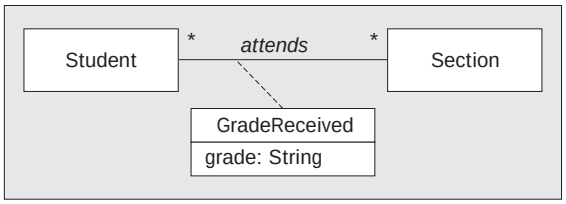
Since we are going to want to use the object model to gain user confirmation that we understand their primary requirements, we needn't clutter the diagram with such behind-the-scenes implementation details just yet.

- We also renamed the association class for the *attends* relationship; it was introduced earlier in this chapter as `GradeReceived`, but is now called `TranscriptEntry`. We've also introduced an aggregation relationship between the `TranscriptEntry` class and another new class called `Transcript`.

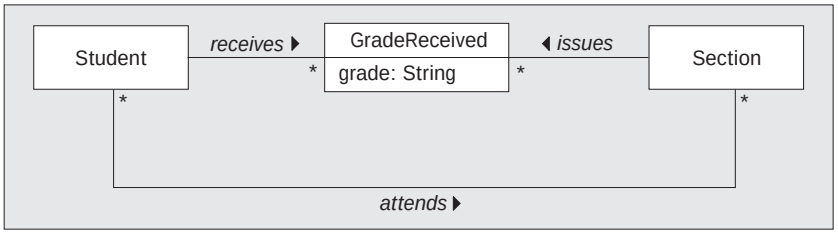


Let's explore how all of this evolved.

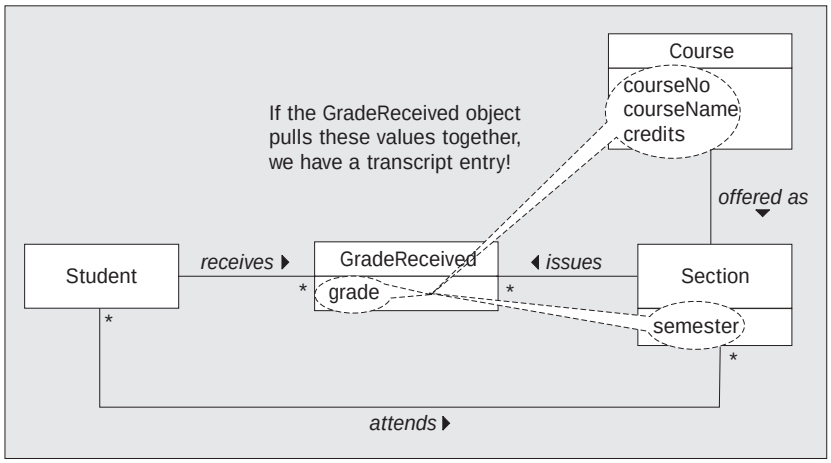
- When we first introduced the *attends* association earlier in this chapter, we portrayed it as follows:



- We then learned that it could equivalently be represented as a pair of one-to-many associations *issues* and *receives*:



- In this alternative form, it is clear that any individual `GradeReceived` object maintains one handle on a `Student` object and another handle on a `Section` object, and can ask either of them for information whenever necessary. The `Section` object, in turn, maintains a handle on the `Course` object that it represents by virtue of the *offered as* association. It is a trivial matter, therefore, for the `GradeReceived` object to request the values of attributes '`semester`', '`courseNo`', '`courseName`', and '`credits`' from the `Section` object (which would in turn have to ask its associated `Course` object for the last three of these four values); this is illustrated conceptually below.



- If the `GradeReceived` object pulls these values together, we have everything that we need for a line item entry on a student's transcript:

Transcript For: Joe Blow — Semester: Spring 2000

Course No.	Credits	Course Name	Grade Received	Credits Earned*
MATH 101	3	Beginning Math	B	9
OBJECTS 101	3	Intro to Objects	A	12
ART 200	3	Clay Modelling	A	12

** 'Credits Earned' is computed by multiplying the credit value of a course - say, 3 - by 4 if the student earned an A grade, 3 if he/she earned a B, and so forth.*

Therefore, we see that renaming the association class from `GradeReceived` to `TranscriptEntry` makes good sense.

- It was then a natural step to aggregate these into a `Transcript` class.

- The diagram is a little 'light' in terms of attributes; we've only reflected those which we'll minimally need when we build an automated SRS in Part 3.

Of course, we need to go back to the data dictionary to capture definitions of all of the new attributes, relationships, and classes that we've identified in putting together this model. Here is our revised SRS data dictionary:

Classes

Course: a semester-long series of lectures, assignments, exams, etc. that all relate to a particular subject area, and which are typically associated with a particular number of credit hours; a unit of study toward a degree. For example, Beginning Objects is a required **course** for the Master of Science Degree in Information Systems Technology.

Person: a human being associated with the university.

PlanOfStudy: a list of the **courses** that a student intends to take to fulfill the **course** requirements for a particular degree.

Professor: a member of the faculty who teaches **sections** and/or advises **students**.

Section: the offering of a particular **course** during a particular semester on a particular day of the week and at a particular time of day (For example, **course** 'Beginning Objects' as taught in the Spring 2001 semester on Mondays from 1:00 – 3:00 PM).

Student: a person who is currently enrolled at the university and who is eligible to register for one or more **sections**.

Transcript: a record of all of the **courses** taken to date by a particular **student** at this university, including which semester each **course** was taken in, the grade received, and the credits granted for the **course**, as well as reflecting an overall total number of credits earned and the **student's** grade point average (GPA).

TranscriptEntry: one line item entry from a **transcript**, reflecting the **course** number and name, semester taken, value in credit hours, and grade received.

Relationships

*advises: a **professor** advises a **student**:* A professor is assigned to oversee a student's academic pursuits for the student's entire academic career, leading up to his/her attainment of a degree. An advisor counsels his/her advisees regarding course selection, professional opportunities, and any academic problems the student might be having.

*attends: a **student** attends a **section**:* A student registers for a section, attends class meetings for a semester, and participates in all assignments and examinations, culminating in the award of a letter grade representing the student's mastery of the subject matter.

*calls for: a **plan of study** calls for a **course**:* A student may only take a course if it is called out by his/her plan of study. The plan of study may be amended, with a student's advisor's approval.

*maintains: a **student** maintains a **transcript**:* Each time a student completes a course, a record of the course and the grade received is added to the student's transcript.

*observes: a **student** observes a **plan of study**:* See notes for the *calls for* association, above.

*offered as: a **course** is offered as a **section**:* The same course can be taught numerous times in a given semester, and of course over numerous semesters for the 'lifetime' of a course – that is, until such time as the subject matter is no longer considered to be of value to the student body, or there is no qualified faculty to teach the course.

*prerequisite: a **course** is a prerequisite for another **course**:* If it is determined that the subject matter of a course 'A' is necessary background to understanding the subject matter of a course 'B', then 'A' is said to be a prerequisite of 'B'. A student typically may not take 'B' unless he/she has either successfully completed 'A', or can otherwise demonstrate mastery of the subject matter of 'A'.

*teaches: a **professor** teaches a **section**:* A professor is responsible for delivering lectures, assigning thoughtful homework assignments, examining students, and otherwise ensuring that a quality treatment of the subject matter of a course is made available to students.

waitlisted for: a student is waitlisted for a section: If a section is 'full' – for example the maximum number of students have signed up for the course based on either the classroom capacity or the student group size deemed effective for teaching – then interested students may be placed on a waitlist, to be given consideration should seats in the course subsequently become available.

(aggregation between Transcript and TranscriptEntry)
(specialization of Person as Professor)
(specialization of Person as Student)

Attributes

Person.ssn: The unique social security number (SSN) assigned to an individual.

Person.name: The person's name, in 'last name, first name' order.

Professor.title: The rank attained by the professor, e.g. 'Adjunct Professor'.

Student.major: A reflection of the department in which a student's primary studies lie, for example Mathematics. (We assume that a student may only designate a single major.)

Student.degree: The degree that a student is pursuing, e.g. Master of Science Degree.

TranscriptEntry.grade: A letter grade of A, B, C, D, or F, with an optional +/- suffix, such as 'A+' or 'C-'.

Course.courseNo: A unique ID assigned to a course, consisting of the department designation plus a unique numeric ID within the department, for example: ‘MATH 101’.

Course.courseName: A full name describing the subject matter of a course, for example 'Beginning Objects'.

Course.credits: The number of units or credit hours a course is worth, roughly equating to the number of hours spent in the classroom in a single week (typically, 3 credits for a full semester lecture course).

Section.sectionNo: A unique number assigned to distinguish one section/offering of a particular course from another offering of the same course in the same semester, for example MATH 101 section no. 1.

Section.dayOfWeek: The day of the week on which the lecture course meets.

Section.timeOfDay: The time (range) during which the course meets, for example 2 – 4 PM.

Section.semester: An indication of the scholastic semester in which a section is offered, for example ‘Spring 2000’.

Section.room: The building and room number where the section will be meeting, for example ‘Government Hall Room 105’.

Section.seatingCapacity: The maximum number of students permitted to register for a section.

Metadata

One question that is often raised by beginning modelers is why we don't use an inheritance relationship to relate the `Course` and `Section` classes, rather than using a simple association as we have chosen to do. On the surface, it does indeed seem tempting to want `Section` to be a subclass of `Course`, because all of the attributes listed for a `Course` – `courseNo`, `courseName`, and `credits` – also pertain to a `Section`; so, why wouldn't we want `Section` to **inherit** these, in the same way that `Student` and `Professor` inherit all of the attributes of `Person`? A simple example should quickly illustrate why inheritance isn't appropriate.

Let's say that, because 'Beginning Object Concepts' is such a popular course, the university is offering three sections of the course for the Spring 2001 semester. So, we instantiate one `Course` object and three `Section` objects. If `Section` were a subclass of `Course`, then all four objects would reflect `courseNo`, `courseName`, and `credits` attributes. Filling in the attribute values for these four objects, as follows:

Attribute Name	Value for the Course Object
`courseName`	**"Beginning Object Concepts"**
`courseNumber`	**"OBJECTS 101"**
`creditValue`	**3**

Attribute Name	Value for Section Object #1	Value for Section Object #2	Value for Section Object #3
`courseName`	**"Beginning Object Concepts"**	**"Beginning Object Concepts"**	**"Beginning Object Concepts"**
`courseNumber`	**"OBJECTS 101"**	**"OBJECTS 101"**	**"OBJECTS 101"**
`creditValue`	**3**	**3**	**3**
`studentsRegistered`	(to be determined)	(to be determined)	(to be determined)
`instructor`	reference to professor X	reference to professor Y	reference to professor Z
`semesterOffered`	Spring 2001	Spring 2001	Spring 2001
`dayOfWeek`	Monday	Tuesday	Thursday
`timeOfDay`	7:00 PM	4:00 PM	6:00 PM
`classroom`	Hall A, Room 123	Hall B, Room 234	Hall A, Room 345

we see that there is quite a bit of repetition in the attribute values across these four objects: we've repeated the same `courseName`, `courseNumber`, and `creditValue` four times! That's because the information contained within a `Course` object is common to, and hence describes, **numerous** `Section` objects. To reduce redundancy and promote encapsulation, we should eliminate inheritance of these attributes, and instead create only one instance of a `Course` object for *n* instances of its related `Section` objects. We can then have each `Section` object maintain a handle on the common `Course` object so as to retrieve these shared values whenever necessary. This is precisely what we have modeled via the one-to-many *offered as* association.

Whenever an instance of some class 'A' encapsulates information that describes numerous instances of some other class 'B' (such as Course does for Section), we refer to the information contained by the 'A' object (Course) as **metadata** relative to the 'B' objects (Sections).

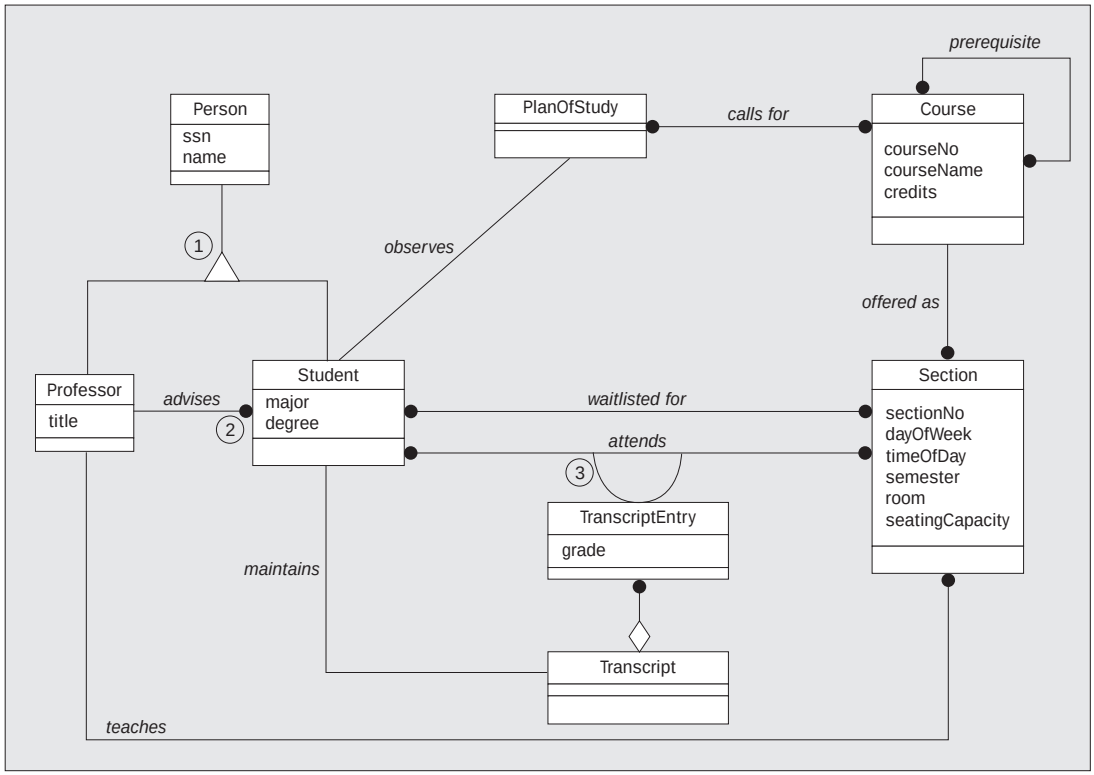
Comparing UML and OMT Notations

We mentioned in the Introduction to the book that UML notation is remarkably similar to OMT (which stands for 'Object Modeling Technique') notation because UML is based to a great extent on OMT. To illustrate how similar the UML and OMT notations are, we present the same model in OMT notation below; note that the only differences are:

1. The change in position of the triangle for the inheritance relationship (it no longer touches the bottom of the superclass).

2. The use of solid circles instead of asterisks to represent the 'many' ends of relationships.

3. The use of a loop instead of a dashed line for the association class (which is called a 'link attribute' in OMT nomenclature).



Summary

Our object model has started to take shape! We have a good idea of what the static structure needs to be for the SRS – the classes, their attributes and relationships with one another – and are able to communicate this knowledge in a concise, graphical form. There are many more embellishments to the UML notation than we haven't covered in this chapter, but we've presented the core concepts that will suffice for most 'industrial strength' modeling projects. Once you've mastered these, you can explore the Recommended Reading section of the book if you'd like to learn more about these notations.

There is an obvious 'hole' in our class diagram, however: all of our classes have empty operations compartments. We'll address this deficiency by learning some complementary modeling techniques for determining the dynamic behavior of our intended system in Chapter 11.

In this chapter, we've learned:

- The noun phrase analysis technique for identifying candidate domain classes.
- The verb phrase analysis technique for determining potential relationships among these classes.
- That coming up with candidate classes is a bit subjective, and hence that we have to remain flexible, and willing to revisit our model, through many iterations until we – and our users – are satisfied with the outcome.
- The importance of producing a data dictionary as part of a project's documentation set.
- How to graphically portray the static structure of our model as a class diagram using UML.
- How important it is to have an experienced object modeling mentor available to a project team.

Exercises

1. Come up with a list of candidate classes for the Conference Room Reservation System (CRRS) case study presented in Appendix B, as well as an association matrix.

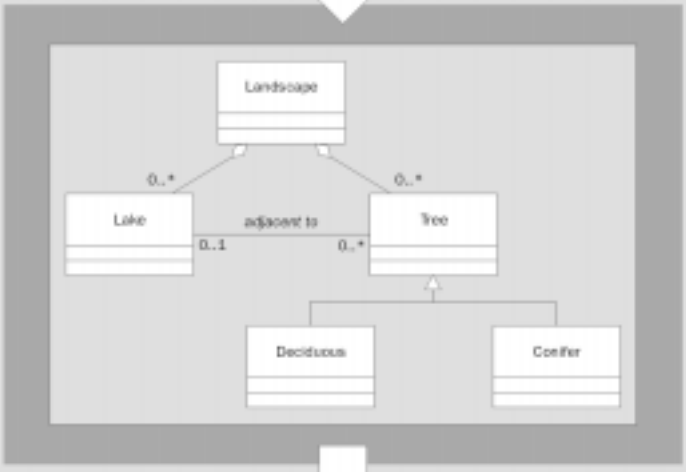
2. Develop a class diagram for the CRRS case study, using UML notation. Reflect all significant attributes and relationships among classes, including the appropriate multiplicity. Ideally, you should use an object modeling software tool if you have one available to you.

3. Prepare a data dictionary for the CRRS, to include definitions of all classes, attributes, and associations.

4. Devise a list of candidate classes for the problem area whose requirements you defined for exercise No. 3 in Chapter 2, as well as an association matrix.

5. Develop a class diagram for the problem area whose requirements you defined for exercise No. 3 in Chapter 2, using UML notation. Reflect all significant attributes and relationships among classes, including the appropriate multiplicity. Ideally, you should use an object modeling software tool if you have one available to you.

6. Prepare a data dictionary for the problem area whose requirements you defined for exercise No. 3 in Chapter 2, to include definitions of all classes, attributes, and associations.




```
public class Tree {
    protected Landscape landscape;
    protected Lake nextTo;

    public void setNextTo(Lake l) {
        nextTo = l;
    }
    public Lake getNextTo() {
        return nextTo;
    }

    public abstract Color getLeafColor();
}
```