CHAPTER 10

Modeling the Static/Data Aspects of the System

Having employed use case analysis techniques in Chapter 9 to round out the Student Registration System (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 saw 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 the following:

- ***What types of objects we're 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 OO 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 object-oriented "blueprint" that we'll work from in implementing an object-oriented Student Registration System 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'll 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 UML notation

Identifying Appropriate Classes

Our first challenge in object modeling is to determine what classes of objects 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 following:

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

Noun Phrase Analysis

Let's perform noun phrase analysis on the SRS requirements specification first, which was originally presented in the Introduction to the book, a copy of which is provided in the following sidebar. All noun phrases have been **bolded**.

HIGHLIGHTING NOUN PHRASES IN THE SRS SPECIFICATION

We have been asked to develop an ***automated Student Registration System*** (***SRS***) for the ***university***. This ***system*** will enable ***students*** to register online for ***courses*** each ***semester***, as well as track 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*** online, 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 online ***transcript*** of ***courses completed*** and ***grades received*** (the ***student*** may review his/her ***transcript*** online at any time).

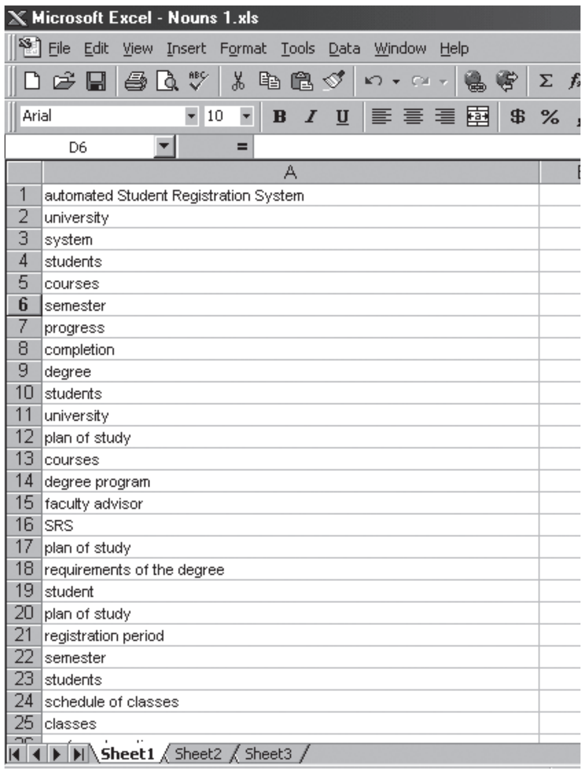
Assuming that (a) the ***prerequisites*** for the ***requested course(s)*** are satisfied, (b) the ***course(s)*** meet(s) one of the ***student***'s ***plan of study requirements***, and (c) there is ***room*** available in each of the ***class(es)***, the ***student*** is enrolled in the ***class(es)***.

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 waitlisted 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 the student's ***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; we 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 in Figure 10-1.

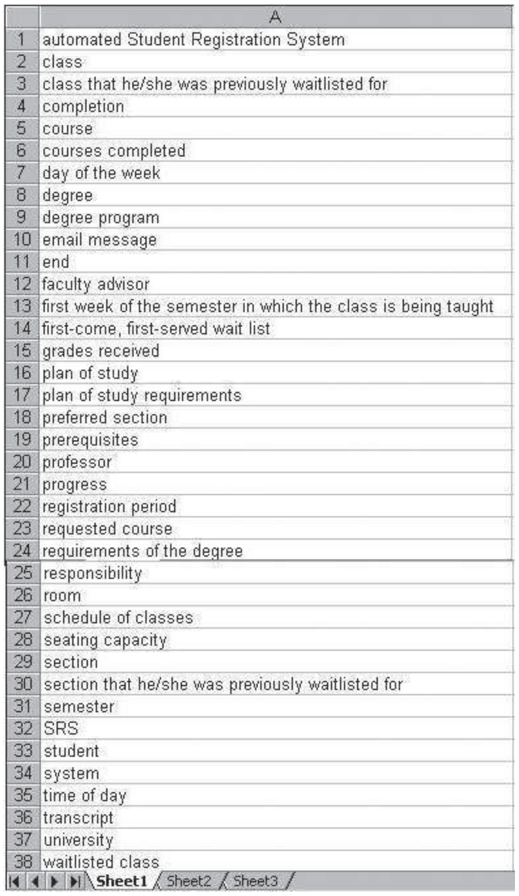

Microsoft Excel - Nouns 1.xls
File Edit View Insert Format Tools Data Window Help
Arial
10
D6
=
Sheet1
Sheet2
Sheet3

	A
1	automated Student Registration System
2	university
3	system
4	students
5	courses
6	semester
7	progress
8	completion
9	degree
10	students
11	university
12	plan of study
13	courses
14	degree program
15	faculty advisor
16	SRS
17	plan of study
18	requirements of the degree
19	student
20	plan of study
21	registration period
22	semester
23	students
24	schedule of classes
25	classes

Figure 10-1. *Noun phrases found in the SRS specification*

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're 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 the system requirements and understand the "big picture." Of course, you'll need to review your summary narrative with your customers/future users to ensure that you've accurately captured all key points.

After we've typed all of the nouns/noun phrases into the spreadsheet, we 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"), we make these singular, as well. In so doing, our SRS list shrinks to 38 items in length, as shown in Figure 10-2.



	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

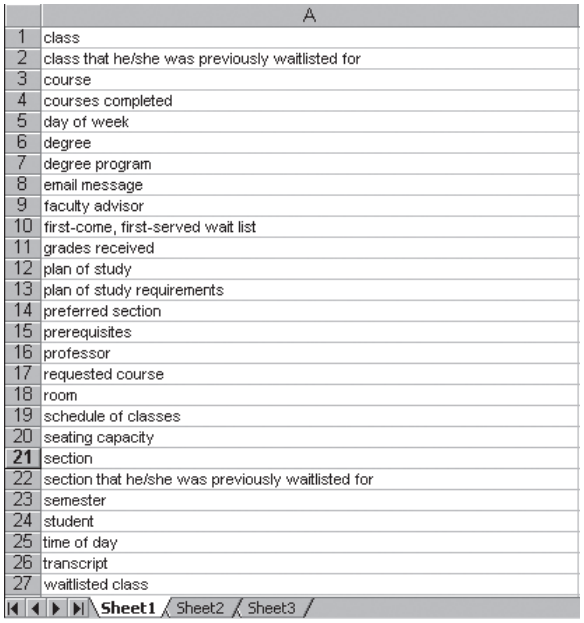
Figure 10-2. *Removing duplicates streamlines the noun phrase list.*

Remember, we're trying to identify both physical and 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 the following:

- References to the system itself (“automated Student Registration System,” “SRS,” “system”).
- References to the university. Because we’re 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 indeed need to model each university as a separate object, in which case we’d keep “university” on our candidate class list.
- Other miscellaneous terms that 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 life cycle.

The list shrinks to 27 items as a result, as shown in Figure 10-3—it’s starting to become manageable now!

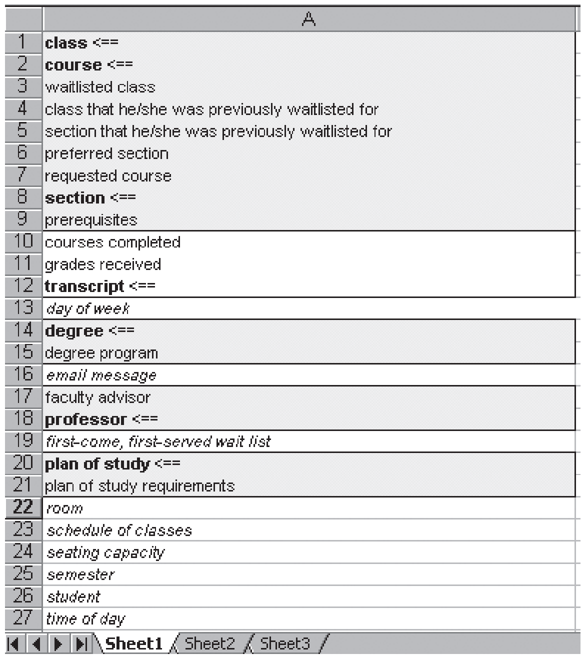


	A
1	class
2	class that he/she was previously waitlisted for
3	course
4	courses completed
5	day of week
6	degree
7	degree program
8	email message
9	faculty advisor
10	first-come, first-served wait list
11	grades received
12	plan of study
13	plan of study requirements
14	preferred section
15	prerequisites
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

Figure 10-3. *Further streamlining the SRS noun phrase list*

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 serve 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 isn’t always easy.

We group together terms that seem to be synonyms, as shown in Figure 10-4, **bolding** the term in each synonym group that we're 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*

Figure 10-4. *Grouping synonyms*

Let's now 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 drop the latter two noun phrases for now.

When choosing candidate class names, we should avoid choosing nouns that imply **roles** between objects. As you 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 or her advisees, thus losing the role of faculty advisor, he or she would still be a professor by virtue of being employed by the university—it's inherent in the person's nature relative to the SRS.

If a professor were to lose his or her job with the university, one might argue that he or she is no longer a professor; but then, this person would have no dealings with the SRS, either, so it's 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've 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 `Course`s.
- "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're 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 Beginning Math is being offered this coming Spring semester as three classes/sections:
  - Section 1, which meets Tuesdays from 4:00 to 6:00 p.m.
  - Section 2, which meets Wednesdays from 6:00 to 8:00 p.m.
  - Section 3, which meets Thursdays from 3:00 to 5:00 p.m.

  There is thus 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 initial inclination would be to keep "class" and discard "section," but in order to avoid confusion when referring to a class named `Class` (!) we'll opt for "section" instead.

Refining the Candidate Class List

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—I'll 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 these questions:

- Can we think of any ***attributes*** for this class?
- Can we 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:

```
public 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:

```
public 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.,
                            // "Innovation Hall Room 333"
  // 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 also 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 weren’t 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 life cycle of a modeling exercise between being classes on their own versus 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 preceding candidate class list, we’ll opt to treat 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.
- “Time of day” will be incorporated as a `String` attribute of the `Section` class.

When we’re first modeling an application, we want to focus exclusively on functional requirements to 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 the associated student ID number. We’ll 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 earlier—“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 pop-up message that appears on the screen while using an application. An email message gets sent out of the SRS system, and after it’s read by the recipient, we have no control over whether the email is retained or deleted. It’s 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 don't 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 domain 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 or she registers in a given semester. We'll omit `ScheduleOfClasses` from our candidate class list for now, but we 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 preceding 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 1; because of its importance, it is repeated again in the following sidebar.

THE CHALLENGES OF OBJECT MODELING

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.

Finally, 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 isn't "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`
- `PlanOfStudy`
- `Professor`
- `Section`
- `Student`
- `Transcript`
- `WaitList`

Let's examine `WaitList` 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 you'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 `WaitList` 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`
- `PlanOfStudy`
- `Professor`
- `Section`
- `Student`
- `Transcript`

Revisiting the Use Cases

One more thing that we need to do before we deem our candidate class list good to go is to revisit our use cases—in particular, the actors—to see if any of ***these*** 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 on to 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 on to the SRS, might he or she need to manipulate information about faculty? Yes; when a student selects an advisor, for example, he or 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***; 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 on to 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 on to 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 automated 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`
- `PlanOfStudy`
- `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'll 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, or relationships with other classes to warrant the creation of a common base class.
- In addition, as 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 illustrated in this chapter thus far, fraught with uncertainty. For this reason, it's 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's impossible to define precisely what makes one concept significant and another less so. I'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 explore later in this chapter as well as in Chapter 11.

Producing a Data Dictionary

Early on in our analysis efforts, it's 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.

The following sidebar shows our complete SRS data dictionary so far.

THE SRS DATA DICTIONARY, TAKE 1: CLASS DEFINITIONS

- **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.
- **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** 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 2005 semester on Mondays from 1:00 to 3:00 p.m.).
- **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 a reflection of an overall total number of credits earned and the **student's** grade point average (GPA).

Note that it's 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'll expand the dictionary to include definitions of attributes, associations, and methods.

It's 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 inline documentation in sync 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 the student's plan of study, etc.—these are behaviors on the part of a professor acting in the role of an advisor, but don't directly result in any new links being formed between objects.

Let's try the verb phrase analysis approach on the requirements specification. All relevant verb phrases are highlighted in the sidebar that follows (note that I omitted such obviously irrelevant verb phrases as "We've been asked to develop an automated SRS . . .").

HIGHLIGHTING VERB PHRASES IN THE SRS SPECIFICATION

We have been asked to develop an automated Student Registration System (SRS) for the university. This system will **enable students to register** online **for courses** each semester, as well as **track 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, a student is able to **view the schedule of classes** online, 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 online transcript** of courses completed and grades received (the **student may review his/her transcript** online 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 waitlisted 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 the student's 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 natural language is often 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're 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 online"***: 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.

Association Matrices

Another complementary technique for both determining and recording what the relationships between classes should be is to create an ***n*** × ***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 represented by Table 10-1.

Table 10-1. *An "Empty" Association Matrix for the SRS*

	`Section`	`Course`	`PlanOfStudy`	`Professor`	`Student`	`Transcript`
`Section`						
`Course`						
`PlanOfStudy`						
`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 in Table 10-2 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 his or her career as a student, which we must do if we are to prepare a student's transcript online. (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 you'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, in Table 10-2 the cells representing the intersection between `Professor` and `Course` are marked 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.

As mentioned in Chapter 4, 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. For example, since the intersection of the `PlanOfStudy` "row" and the `Course` "column" indicates that "a `PlanOfStudy` *calls for* a `Course`," then the intersection of the `Course` "row" and the `PlanOfStudy` "column" must indicate that "a `Course` *is called for by* a `PlanOfStudy`."

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 trying 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 the symbol ✔.

Table 10-2. *Our Completed Association Matrix*

	`Section`	`Course`	`PlanOfStudy`	`Professor`	`Student`	`Transcript`
`Section`	×	*instance of*	×	*is taught by*	✔	*included in*
`Course`	✔	*prerequisite for*	*is called for by*	×	✔	×
`PlanOfStudy`	×	*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, let's go back and extend our data dictionary to explain what each of these associations means. The following sidebar shows one such example.

ADDITIONS TO THE SRS DATA DICTIONARY

Calls for (a Plan of Study calls for a Course): In order to demonstrate that a **student** will satisfy the requirements for his or 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 isn't mentioned anywhere in the SRS specification. We can't be sure whether this particular university assigns an arbitrary student ID number, or whether the policy is to use a student's Social Security number (SSN) as his or her ID; these are details that we'd have to go back to our 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 UML to produce a **class diagram.** Here are the rules for how various aspects of the model are to be portrayed.

Classes, Attributes, and Operations

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, as illustrated in Figure 10-5.

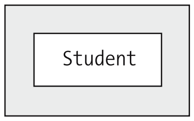


Figure 10-5. *UML depiction of the* `Student` *class*

An ***abstract*** class is denoted by presenting the class name in *italics*, as shown in Figure 10-6.

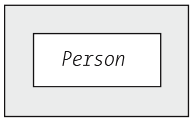


Figure 10-6. *UML depiction of an abstract class*

When we're ready to reflect the attributes and operations of a class, we divide the class rectangle into three **compartments**—the class name compartment, the attributes compartment, and the operations compartment—as shown in Figure 10-7. Note that UML favors the nomenclature of "operations" versus "methods" to reinforce the notion that the diagram is intended to be programming language independent.

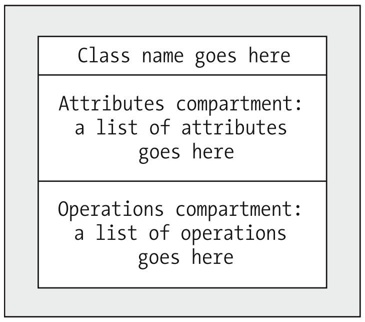

Class name goes here
Attributes compartment: a list of attributes goes here
Operations compartment: a list of operations goes here

Figure 10-7. *Class rectangles are divided into three compartments.*

Some CASE tools automatically portray all three (empty) compartments when a class is first created, even if we haven't specified any attributes or operations yet, as shown in Figure 10-8.

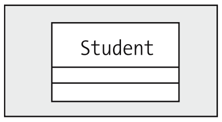


Figure 10-8. *Alternative UML class depiction as rendered by some CASE tools*

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 (see Figure 10-9), or we may specify their names along with their types (see Figure 10-10).

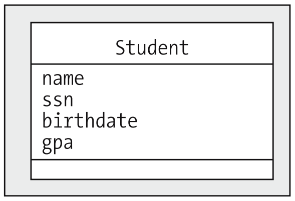


Figure 10-9. *Sometimes just attribute names are presented.*

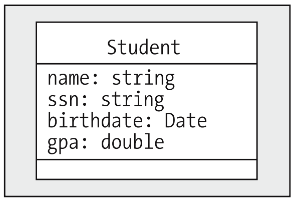


Figure 10-10. *Sometimes both attribute names and types are shown.*

We may even wish to specify an initial starting value for an attribute, as in `gpa : double = 0.0`, although this is less common.

Static attributes are identified as such by underlining their names (see Figure 10-11).

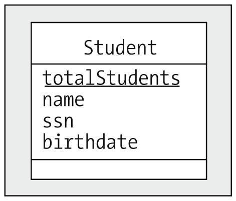


Figure 10-11. *Identifying static attributes by underlining*

We may choose simply to list operation names in the operations compartment of a class rectangle, as shown in Figure 10-12, or we may optionally choose to use an expanded form of operation definition, as we have for the `registerForCourse` operation in Figure 10-13.

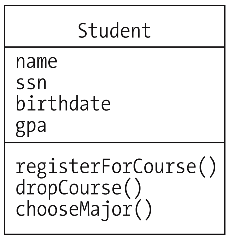


Figure 10-12. *Sometimes just method names are presented.*

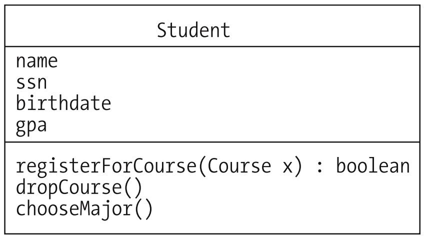


Figure 10-13. *Sometimes argument signatures and return types are also reflected.*

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

`[`*`visibility`*`]` *`name`* `[(`*`parameter list`*`)] [:` *`return type`*`]`

for example,

`registerForCourse(Course x) : boolean`

differs from the syntax that we're used to seeing for Java method headers:

`returnType methodName``(`*`parameter list`*`)`

for example,

`boolean registerForCourse(Course x)`

Note in particular that the UML refers to the combination of operation name, parameters, and return type as the **operation signature**, but that in Java the return type is part of the method header but ***not*** part of the ***method signature***.

The rationale for making these operation signatures generic versus language specific 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're going to be programming in Java, it might make sense to reflect standard Java method headers in your class diagram, if your object modeling tool will accommodate this.

It's often impractical to show all of the attributes and operations 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 reflect in the diagram only those attributes and operations that are particularly important in describing the mission of each class. In particular, "get" and "set" operations are implied for all attributes, and shouldn't be explicitly shown.