Case study

Let's tie all our new object-oriented knowledge together by going through a few iterations of object-oriented design on a somewhat real-world example. The system we'll be modeling is a library catalog. Libraries have been tracking their inventory for centuries, originally using card catalogs, and, more recently, electronic inventories. Modern libraries have web-based catalogs that we can query from our home.

Let's start with an analysis. The local librarian has asked us to write a new card catalog program because their ancient DOS based program is ugly and out of date. That doesn't give us much detail, but before we start asking for more information, let's consider what we already know about library catalogs:

Catalogs contain lists of books. People search them to find books on certain subjects, with specific titles, or by a particular author. Books can be uniquely identified by an **International Standard Book Number** (**ISBN**). Each book has a **Dewey Decimal System** (**DDS**) number assigned to help find it on a particular shelf.

This simple analysis tells us some of the obvious objects in the system. We quickly identify *Book* as the most important object, with several attributes already mentioned, such as author, title, subject, ISBN, and DDS number, and catalog as a sort of manager for books.

We also notice a few other objects that may or may not need to be modeled in the system. For cataloging purposes, all we need to search a book by author is an `author_name` attribute on the book. But authors are also objects, and we might want to store some other data about the author. As we ponder this, we might remember that some books have multiple authors. Suddenly, the idea of having a single `author_name` attribute on objects seems a bit silly. A list of authors associated with each book is clearly a better idea. The relationship between author and book is clearly association, since you would never say "book is an author" (it's not inheritance), and saying "book has an author", though grammatically correct, does not imply that authors are part of books (it's not aggregation). Indeed, any one author may be associated with multiple books.

We should also pay attention to the noun (nouns are always good candidates for objects) *shelf*. Is a shelf an object that needs to be modeled in a cataloging system? How do we identify an individual shelf. What happens if a book is stored at the end of one shelf, and later moved to the beginning of the next shelf because another book was inserted in the previous shelf?

DDS was designed to help locate physical books in a library. As such, storing a DDS attribute with the book should be enough to locate it, regardless of which shelf it is stored on. So we can, at least for the moment, remove shelf from our list of contending objects.

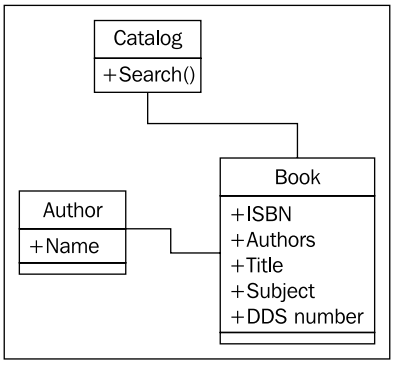
Another questionable object in the system is the user. Do we need to know anything about a specific user? Their name, address, or list of overdue books? So far the librarian has told us only that they want a catalog; they said nothing about tracking subscriptions or overdue notices. In the back of our minds, we also note that authors and users are both specific kinds of people; there might be a useful inheritance relationship here in the future.

For cataloging purposes, we decide we don't need to identify the user, for now. We can assume that a user will be searching the catalog, but we don't have to actively model them in the system, beyond providing an interface that allows them to search.

We have identified a few attributes on the book, but what properties does a catalog have? Does any one library have more than one catalog? Do we need to uniquely identify them? Obviously, the catalog has to have a list of the books it contains, somehow, but this list is probably not part of the public interface.

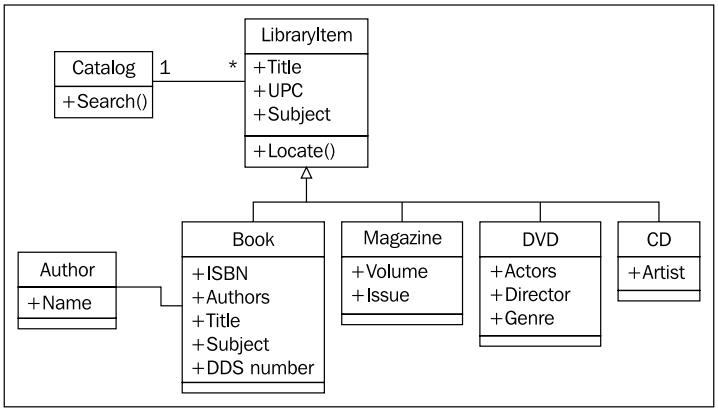
What about behaviors? The catalog clearly needs a search method, possibly separate ones for authors, titles, and subjects. Are there any behaviors on books? Would it need a preview method? Or could preview be identified by a first pages attribute, instead of a method?

The questions in the preceding discussion are all part of the object-oriented analysis phase. But intermixed with the questions, we have already identified a few key objects that are part of the design. Indeed, what you have just seen is several micro-iterations between analysis and design. Likely, these iterations would all occur in an initial meeting with the librarian. Before this meeting, however, we can already sketch out a most basic design for the objects we have concretely identified:



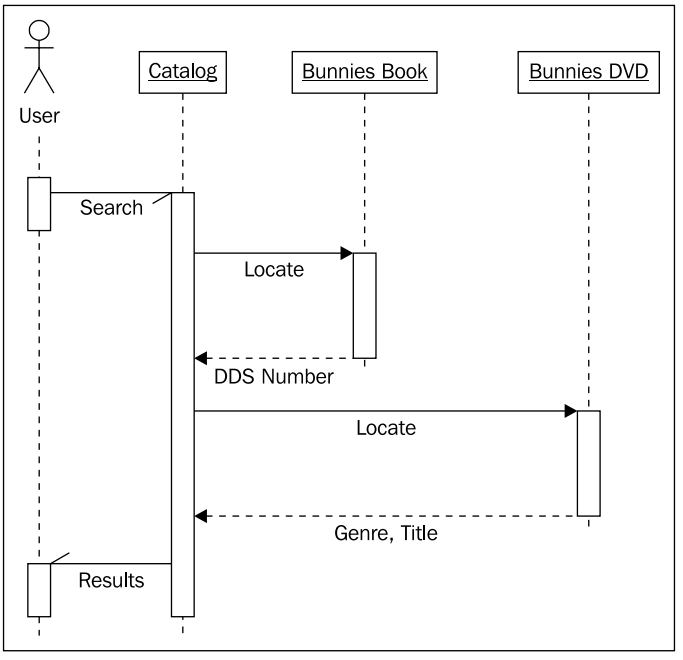
Armed with this basic diagram and a pencil to interactively improve it, we meet up with the librarian. They tell us that this is a good start, but libraries don't serve only books, they also have DVDs, magazines, and CDs, none of which have an ISBN or DDS number. All of these types of items can be uniquely identified by a UPC number, though. We remind the librarian that they have to find the items on the shelf, and these items probably aren't organized by UPC. The librarian explains that each type is organized in a different way. The CDs are mostly audio books and they only have a couple dozen in stock, so they are organized by the author's last name. DVDs are divided into genre and further organized by title. Magazines are organized by title and then refined by volume and issue number. Books are, as we had guessed, organized by DDS number.

With no previous object-oriented design experience, we might consider adding separate lists of DVDs, CDs, magazines, and books to our catalog, and search each one in turn. The trouble is, except for certain extended attributes, and identifying the physical location of the item, these items all behave in much the same. This is a job for inheritance! We quickly update our UML diagram:



The librarian understands the gist of our sketched diagram, but is a bit confused by the **locate** functionality. We explain using a specific use case where the user is searching for the word "bunnies". The user first sends a search request to the catalog. The catalog queries its internal list of items and finds a book and a DVD with "bunnies" in the title. At this point, the catalog doesn't care if it is holding a DVD, book, CD or magazine; all items are the same, as far as the catalog is concerned. But the user wants to know how to find the physical items, so the catalog would be remiss if it simply returned a list of titles. So it calls the **locate** method on the two items it has uncovered. The book's **locate** method returns a DDS number that can be used to find the shelf holding the book. The DVD is located by returning the genre and title of the DVD. The user can then visit the DVD section, find the section containing that genre, and find the specific DVD as sorted by title.

As we explain, we sketch a UML **sequence diagram** explaining how the various objects are communicating:

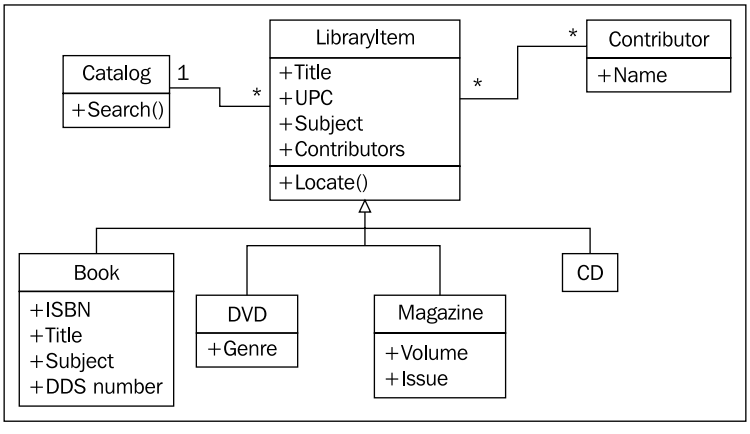


Where class diagrams describe the relationships between classes, sequence diagrams describe specific sequences of messages passed between objects. The dashed line hanging from each object is a **lifeline** describing the lifetime of the object. The wider boxes on each lifeline represent active processing in that object (where there's no box, the object is basically sitting idle, waiting for something to happen). The horizontal arrows between the lifelines indicate specific messages. The solid arrows represent methods being called, while the dashed arrows with solid heads represent the method return values. The half arrowheads indicate asynchronous messages sent to or from an object. An asynchronous message typically means the first object calls a method on the second object which returns immediately. After some processing, the second object calls a method on the first object to give it a value. This is in contrast to normal method calls, which do the processing in the method, and return a value immediately.

Sequence diagrams, like all UML diagrams, are best used when they are needed. There is no point in drawing a UML diagram for the sake of drawing a diagram. But when you need to communicate a series of interactions between two objects, the sequence diagram is a very useful tool.

Unfortunately, our class diagram so far is still a messy design. We notice that actors on DVDs and artists on CDs are all types of people, but are being treated differently from the book authors. The librarian also reminds us that most of their CDs are audio books, which have authors instead of artists.

How can we deal with different kinds of people that contribute to a title? An obvious implementation is to create a `Person` class with the person's name and other relevant details and then create subclasses of this for the artists, authors, and actors. But is inheritance really necessary here? For searching and cataloging purposes, we don't really care that acting and writing are two very different activities. If we were doing an economic simulation, it would make sense to give separate actor and author classes different `calculate_income` and `perform_job` methods, but for cataloging purposes, it is probably enough to know how the person contributed to the item. We recognize that all items have one or more **Contributor** objects, so we move the author relationship from the book to its parent class:

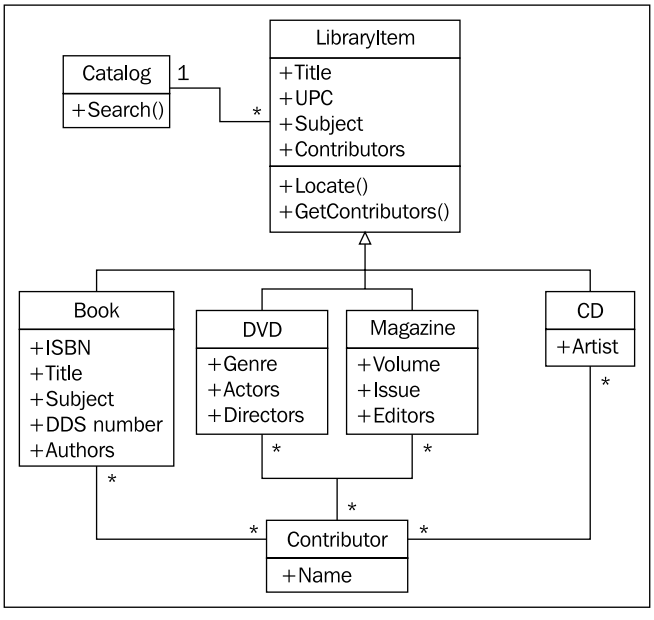


The multiplicity of the **Contributor/LibraryItem** relationship is **many-to-many**, as indicated by the ***** at each end of the relationship. Any one library item might have more than one contributor (for example, several actors and a director on DVD). And many authors write many books, so they would be attached to multiple library items.

This little change, while it looks a bit cleaner and simpler has lost some vital information. We can still tell who contributed to a specific library item, but we don't know how they contributed. Were they the director or an actor? Did they write the audio book, or were they the voice that narrated the book?

It would be nice if we could just add a `contributor_type` attribute on the **Contributor** class, but this will fall apart when dealing with multi-talented people who have both authored books and directed movies.

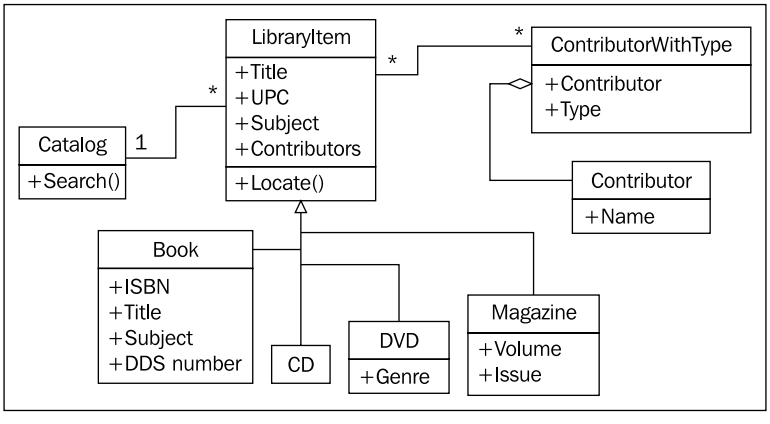
One option is to add attributes to each of our **LibraryItem** subclasses that hold the information we need, such as **Author** on **Book**, or **Artist** on **CD**, and then make the relationship to those properties all point to the **Contributor** class. The problem with this is that we lose a lot of polymorphic elegance. If we want to list the contributors to an item, we have to look for specific attributes on that item, such as **Authors** or **Actors**. We can alleviate this by adding a **GetContributors** method on the **LibraryItem** class that subclasses can override. Then the catalog never has to know what attributes the objects are querying; we've abstracted the public interface:



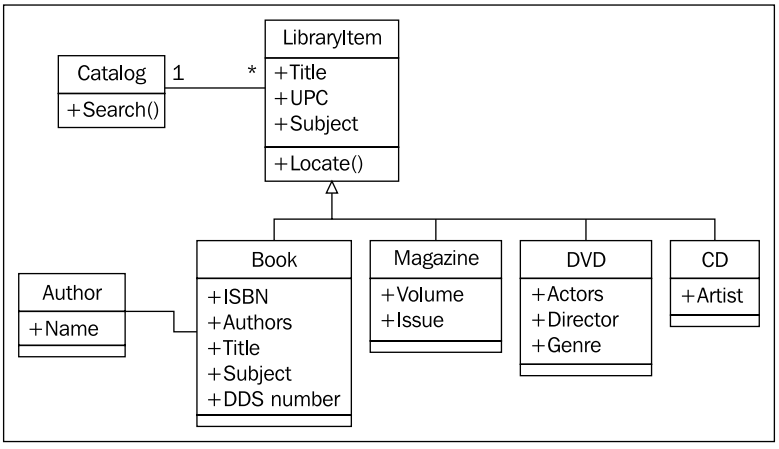
Just looking at this class diagram, it feels like we are doing something wrong. It is bulky and fragile. It may do everything we need, but it feels like it will be hard to maintain or extend. There are too many relationships, and too many classes would be affected by modifications to any one class. It looks like spaghetti and meatballs.

Now that we've explored inheritance as an option, and found it wanting, we might look back at our previous composition-based diagram, where **Contributor** was attached directly to **LibraryItem**. With some thought, we can see that we actually only need to add one more relationship to a brand-new class to identify the type of contributor. This is an important step in object-oriented design. We are now adding a class to the design that is intended to *support* the other objects, rather than modeling any part of the initial requirements. We are **refactoring** the design to facilitate the objects in the system, rather than objects in real life. Refactoring is an essential process in the maintenance of a program or design. The goal of refactoring is to improve the design by moving code around, removing duplicate code or complex relationships in favor of simpler, more elegant designs.

This new class is composed of a **Contributor** and an extra attribute identifying the type of contribution the person has made to the given **LibraryItem**. There can be many such contributions to a particular **LibraryItem**, and one contributor can contribute in the same way to different items. The diagram communicates this design very well:



At first, this composition relationship looks less natural than the inheritance-based relationships. But it has the advantage of allowing us to add new types of contributions without adding a new class to the design. Inheritance is most useful when the subclasses have some kind of specialization. Specialization is creating or changing attributes or behaviors on the subclass to make it somehow different from the parent class. It seems silly to create a bunch of empty classes solely for identifying different types of objects (this attitude is less prevalent among Java and other "everything is an object" programmers, but it is common among more practical Python designers). If we look at the inheritance version of the diagram, we can see a bunch of subclasses that don't actually do anything:



Sometimes it is important to recognize when not to use object-oriented principles. This example of when not to use inheritance is a good reminder that objects are just tools, and not rules.

Exercises

This is a practical book, not a textbook. As such, I'm not about to assign you a bunch of fake object-oriented analysis problems to create designs for. Instead, I want to give you some things to think about that you can apply to your own projects. If you have previous object-oriented experience, you won't need to put much effort into these. But they are useful mental exercises if you've been using Python for a while but never really cared about all that class stuff.

First, think about a recent programming project you've completed. Identify the most prominent object in the design. Try to think of as many attributes for this object as possible. Did it have: Color? Weight? Size? Profit? Cost? Name? ID number? Price? Style? Think about the attribute types. Were they primitives or classes? Were some of those attributes actually behaviors in disguise? Sometimes what looks like data is actually calculated from other data on the object, and you can use a method to do those calculations. What other methods or behaviors did the object have? What objects called those methods. What kinds of relationships did they have to this object?

Now think about an upcoming project. It doesn't matter what the project is; it might be a fun free-time project or a multi-million dollar contract. It doesn't have to be a complete application; it could just be one subsystem. Perform a basic object-oriented analysis. Identify the requirements and the interacting objects. Sketch out a class diagram featuring the very highest level of abstraction on that system. Identify the major interacting objects. Identify minor supporting objects. Go into detail for the attributes and methods of some of the most interesting ones. Take different objects to different levels of abstraction. Look for places you can use inheritance or composition. Look for places you should avoid inheritance.

The goal is not to design a system (although you're certainly welcome to do so if inclination meets both ambition and available time). The goal is to think about object-oriented designs. Focusing on projects that you have worked on or are expecting to work on in the future simply makes it real.

Now visit your favorite search engine and look up some tutorials on UML. There are dozens, so find the one that suits your preferred method of study. Sketch some class diagrams or a sequence diagram for the objects you identified earlier. Don't get too hung up on memorizing the syntax (after all, if it is important, you can always look it up again), just get a feel for the language. Something will stay lodged in your brain, and it can make communicating a bit easier if you can quickly sketch a diagram for your next OOP discussion.

Summary

In this chapter, we took a whirlwind tour through the terminology of the object-oriented paradigm, focusing on object-oriented design. We learned how to separate different objects into a taxonomy of different classes and to describe the attributes and behaviors of those objects via the class interface. In particular, we covered:

- Classes and objects
- Abstraction, encapsulation, and information hiding
- Designing a public interface
- Object relations: association, composition, and inheritance
- Basic UML syntax for fun and communication

In the next chapter, we'll explore how to implement classes and methods in Python.