Collaboration

"The objects within a program must collaborate; otherwise, the program would consist of only one big object that does everything."

-- Rebecca Wirfs-Brock, et. al.,
Designing Object-Oriented Software,
Prentice Hall, 1990

INTRODUCTION

Collaboration, to my mind, is not discussed enough. It is one of the essential elements of object-oriented analysis and design. As Booch says:

> *"Equally important [as inheritance] is the invention of societies of objects that responsibly collaborate with one another. ... These societies form what I call the mechanisms of a system, and thus represent strategic architectural decisions because they transcend individual classes." [The C++ Journal, Vol. 2, NO. 1 1992, "Interview with Grady Booch"]*

In this article we will talk about what collaboarations are and why they are so important. We will discuss how collaborations are unearthed through analysis of the problem domain, and how they are designed into the application. We will also discuss the C++ "friend" mechanism, and how it aids the design of collaborations.

Some of the examples in this article use a variation of the Booch Notation for describing analysis and design decisions. Where necessary I will digress to explain the notation.

WHAT IS COLLABORATION?

A collaboration occurs every time two or more objects interact. A collaboration can be as simple as one object sending one message to another object. Or it can be a as complex as dozens of objects exchanging messages. In fact, an entire application is really a single gigantic collaboration involving all of the objects within it.

An object-oriented application can be broken down into a set of many different behaviors. Each such behavior is implemented by a distinct collaboration between the objects of the appliation. Every collaboration, no matter how small or large, always implements a behavior of the application that contains it.

Imagine an object-oriented application as a network of objects connected by relationships. Collaborations are the patterns of messages that play through that network in pursuit of a particular behavior. A collaboration can be viewed as an algorithm which spans this network, using many different objects and methods. The algorithm is distributed across the network of objects, and so does not exist in any one place.

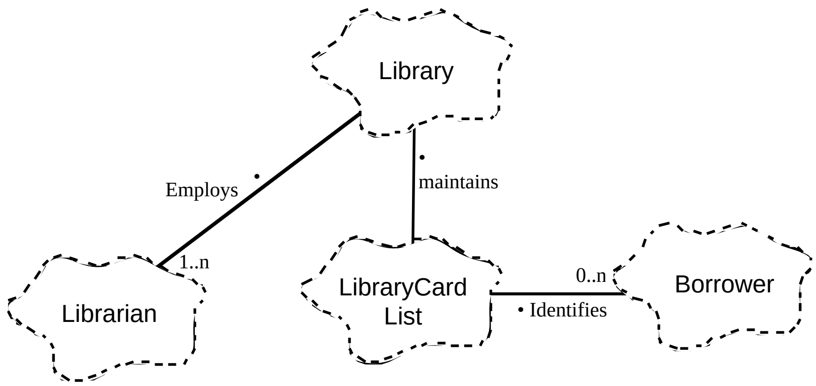
This is in distinct contrast to the behaviors of a class. All behaviors pertinent to a class are methods of that class. They exist in one place. But an object-oriented application is made up of many such classes. Its behaviors are a synthesis of the individual class behaviors. So the application's behaviors are distributed through the classes as collaborations.

This identification with the behaviors of the application gives collaborations a very central role in the analysis and design of object-oriented programs. It is these behaviors, after all, that we are trying to achieve. If the collaborations which implement them are not properly designed, then the application will be inaccurate or brittle.

IDENTIFYING COLLABORATIONS

Collaborations are typically unearthed during the analysis of the problem domain. The first step in this process is to discover the primary classes and their relationships. These are arranged into a model of the static structure of the application. To test this structure, behavioral scenarios are examined. In each scenario we ask which objects will be present, and how they will respond to one particular event. We then attempt to figure out which messages are sent between the objects in order to handle the event. It is within these scenarios that the first hints of collaboration are to be found.

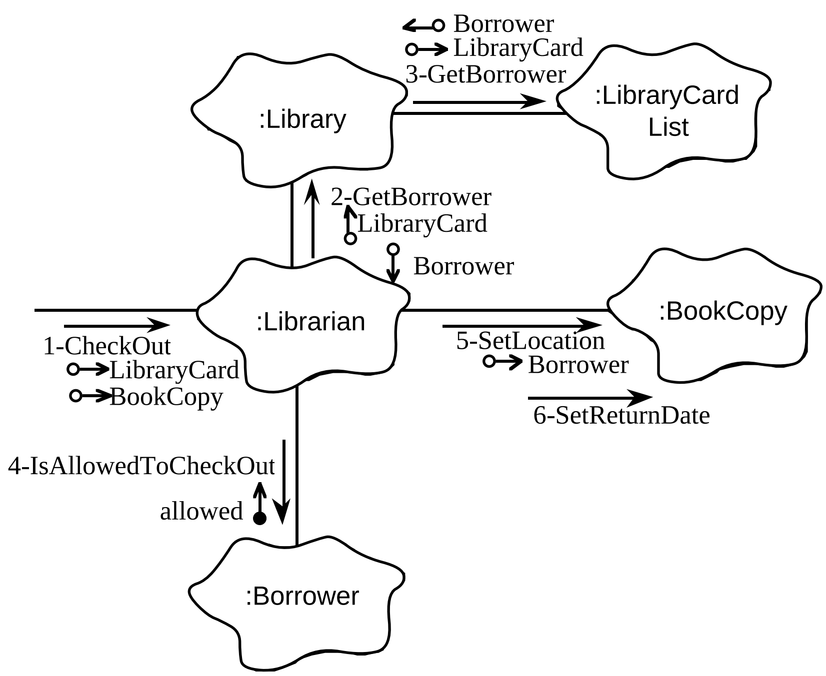
For example, consider an application to automate a public library. The analysis of such an application might yeild the following static model. This model is by no means complete, it simply shows a few of the classes in the problem domain.



This diagram is called a class diagram. It is typical of those produced during object-oriented analysis. It is similar to an entity relationship diagram (ERD), except that it uses Booch symbols. It shows the classes in the model, and the static relationships between those classes.

In this case we see that the `Library` employs some number of `Librarians`. It also maintains a list of all the library cards which identify the `Borrower`s that the `Library` is willing to loan books to.

Lets examine the behavioral scenario related to borrowing a book from the library. A Borrower takes a book up to a Librarian and presents his or her library card with a request to check the book out. The librarian enters the book id and library card number into a terminal. This creates an event from which we can trace out the flow of messages through the system.



This diagram is called an object diagram. It shows the objects that we expect to participate in the behavior, and shows the messages and data that flow between those objects. Note that each message is numbered in the sequence that it occurs.

We have shown the initial event as the `CheckOut` message which is sent to the `Librarian` object (message #1). The message includes the `BookCopy`, which is an object which represents a particular copy of a book. The message also contains the `LibraryCard` of the `Borrower`. The `Librarian` asks the `Library` to look up the `Borrower` from the `LibraryCard` (#2), The `Library` in turn asks the `LibraryCardList` for the same information (#3).

Once in possession of the `Borrower`, the `Librarian` checks its status `(#4),` to see if it is allowed to check out any books. In this example, the `Borrower` *is* allowed to check out books, so the `Location` of the book is set to the `Borrower (#5)`, and the appropriate return date is set (#6).

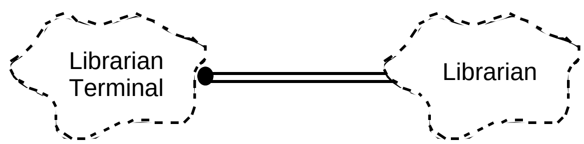
This behavioral scenario is a first step towards identifying the collaboration for checking a book out of the library. Its purpose, at this stage, is to prove that the static model is capable of supporting the behavior. But is also gives us a very good idea of the methods that the classes will need in order to properly collaborate.

Every behavior of the application should be modeled in this way. From this work a set of behavioral scenarios is generated. Each of these is an early representation of the collaborations within the application.

DESIGNING COLLABORATIONS

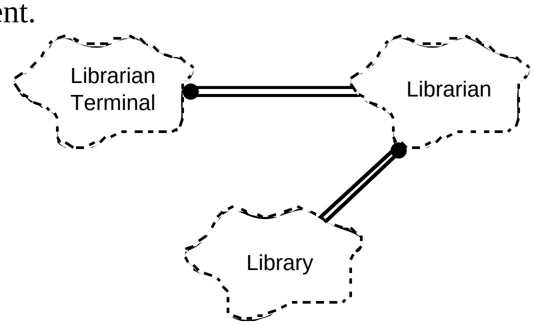
Identification is not enough. By analyzing the problem domain we have compiled a list of proto-collaborations. Now we need to design the detailed structure of the application so that the collaboration can be supported. This involves replacing the weak relationships in the analysis model, with strong OOD relationships such as inheritance (IsA), containment (HasA) and usage relationships. This is done by inspecting the behavioral scenario to see how the messages flow.

For example, the first message in the library collaboration comes to the `Librarian` from the outside. This implies some kind of `LibrarianTerminal` object which knows about the `Librarian`.



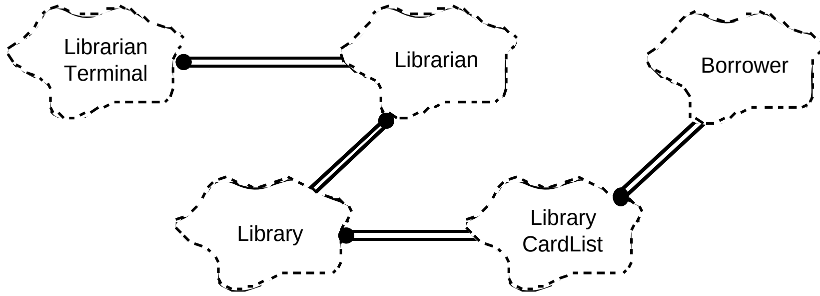
The black ball and double line represents a containment (HasA) relationship. The class `LibrarianTerminal` contains a `Librarian`. This relationship means that the `LibrarianTerminal` has intrinsic knowledge of the `Librarian`. This is important if the `LibrarianTerminal` is to send a message to the `Librarian`.

The second message in the collaboration is between the `Librarian` and the `Library`. Since none of the data currently flowing in the collaboration has identified a particular `Library` object, the `Librarian` must has intrinsic knowledge of the `Library`. Once again, this implies containment.

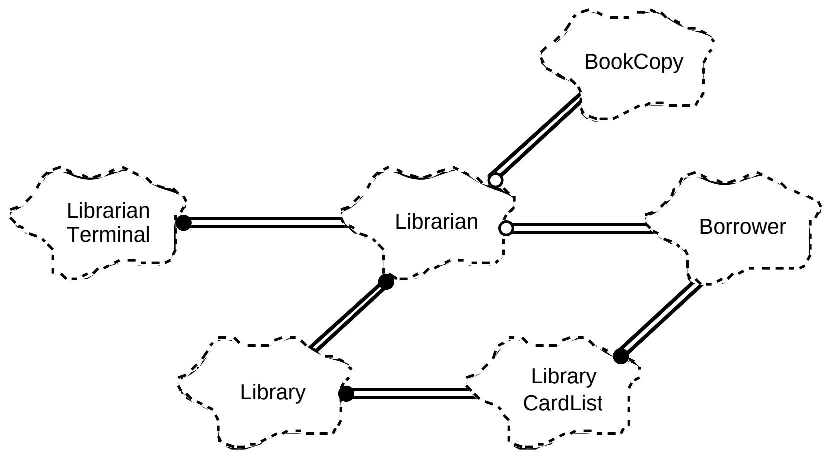


Note that this relationship seems to go the "wrong" direction when compared to the analysis model. In the analysis model the `Library` *employed* the `Librarian`. However, in this design, the `Librarian` contains the `Library`. Although the analysis model makes perfect sense by itself, it does not support the needed collaboration at the detailed level. Thus, the direction of the relationship must changed to support the collaboration.

Message number 3 is sent from the Library to the LibraryCardList. Again, intrinsic knowledge is needed, again implying containment. Moreover, we know from the analysis model that the LibraryCardList identifies all the Borrowers. This too implies containment.



Message number 4 represents the `Librarian` interrogating the `Borrower` about its ability to borrow books. Intrinic knowledge is not implied since the `Borrower` was returned to the `Librarian` through message number 2 and 3. Thus we say that the `Librarian` *uses* the `Borrower`, but does not contain it. The *using* relationship, represented by the double line and white ball, implies that the used object is somehow made available to the user via the user's interface. By the same reasoning, messages 5 and 6 imply that the Librarian *uses* the class BookCopy, since it finds out about the BookCopy from the LibrarianTerminal in message #1.



This design of the classes within the library model now fully supports the check-out collaboration. Similar exercises need to occur for each of the collaborations unearthed through the analysis.

Notice that the static model of the analysis was used in the creation of our collaboration, and that the collaboration was then used to refine the static model. This oscillation between the static and dynamic models is typical and essential. We only showed one small oscillation, but in a real analysis and design, the oscillations would continue many more times before the design was considered sufficiently refined. Each change to the static model sheds new light on the dynamics of the collaborations. Each refinement made to the collaborations may expose deficiencies in the static model.

TYPES OF COLLABORATION

We can classify the ways in which classes collaborate into 4 broad categories. Each of these categories has to do with the relationships between the collaborating classes. The differences between these 4 classifications has to do with the intimacy of the collaboration. Some collaborations take place strictly through their public interfaces, and are therefore not very intimate. Other collaborations require closer coupling between the participants.

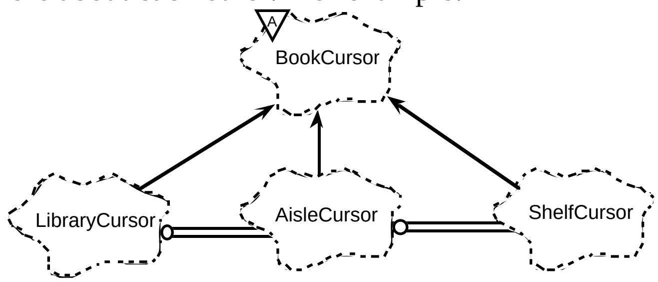
•Peer- to-Peer collaborations

All the collaborations that we have studied so far have been of the Peer-to-Peer variety. Peer-to-Peer collaborations occur when two unrelated classes exchange messages. This is the most common form of collaboration.

Typically, peer-to-peer collaborations are not intimate; i.e. the collaborators do not depend upon special knowledge of each other. In C++, they are seldom declared as friends. This is not a hard and fast rule however. Sometimes intimacy is indicated. Containers and iterators are an example of peer-to-peer collaborators which are generally intimate and require friendship.

•Sibling Collaborations

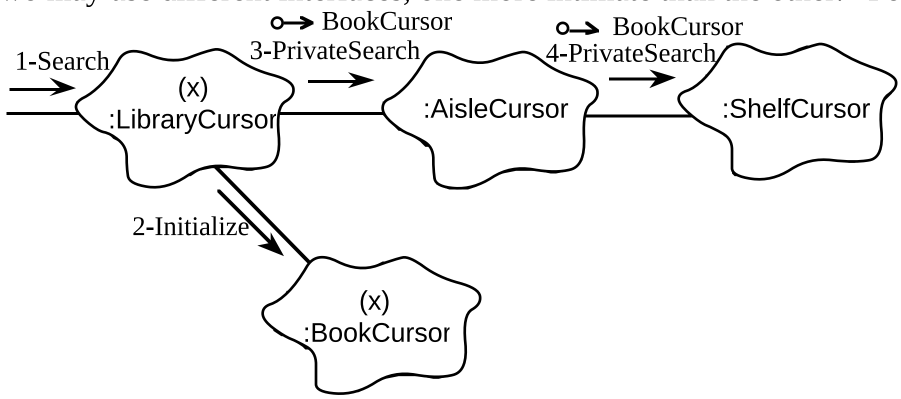
A Sibling collaboration occurs when two or more classes, derived from a common base, exchange messages. Often such collaborations are more intimate than the Peer-to-Peer variety, since the objects know more about each other. For example:



Here we see three classes derived from the same base class; they are siblings. The `BookCursor` base class is abstract, which is signified by the triangular icon. `BookCursor` represents the set of classes which search the library for books. The three siblings represent different scopes in which such searches can occur. You can search an entire shelf with `ShelfCursor`, an entire aisle with `AisleCursor` and the whole library with `LibraryCursor`.

Notice that the siblings make use of each other in a directional manner. The `LibraryCursor` uses the `AisleCursor` which in-turn uses the `ShelfCursor`. This makes perfect sense, since searching the library is a matter of searching all the aisles, and searching an aisle is a matter of searching all the shelves within the aisle.

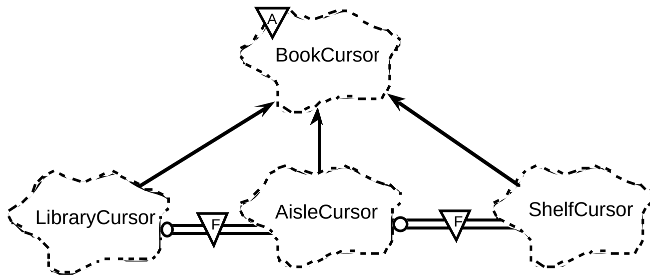
This kind of hierarchical relationship is typical of sibiling collaborations. Each sibling builds on the facilities of the other. However, siblings are often able to deal with peer clients as well. When dealing with peers, the relationship is usually not as intimate as when dealing with a sibling, so the two may use different interfaces, one more intimate than the other. For example:



Here we see a client sending the `Search` message to object `(x):LibraryCursor`. The name of the object is 'x', but the parenthesis indicate that the name is local to this diagram, and not known to the rest of the design. It's kind of like a local variable. Object 'x' responds by sending itself the `Initialize` method, which is handled by the `BookCursor` base class. This method clears a set of counters in the `BookCursor` which keep track of statistics concerning the search.

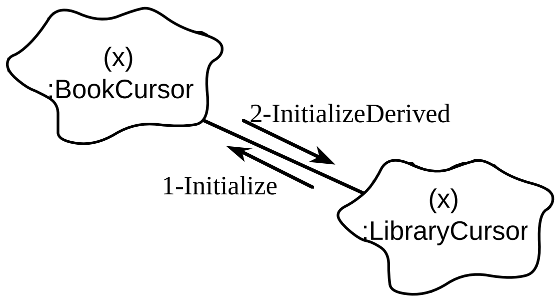
Since each of the siblings must be able to deal directly with clients, they must each respond to the `Search` method by initializing the base class with the `Initialize` method. However, when we are searching the entire library, we want all the statistics gathered in the base class of the `LibraryCursor` object, rather than spread out through a bunch of `AisleCursor` and `ShelfCursor` objects. So the `LibraryCursor` object 'x' tells the `AisleCursor` to use the statistics counters in the base class of 'x'. Moreover, the `AisleCursor` passes this information along to the `ShelfCursor` as well. This information is passed using the `PrivateSearch` method, which is designed for intimate use between siblings, rather than general purpose client access.

Since the classes have a method that they wish to keep private amongst themselves, they should declare the method to be restricted to private access. In order for the siblings to access the methods, they must be friends of each other. Thus we modify the class diagram to show the friendship.



•Base-Derived collaborations

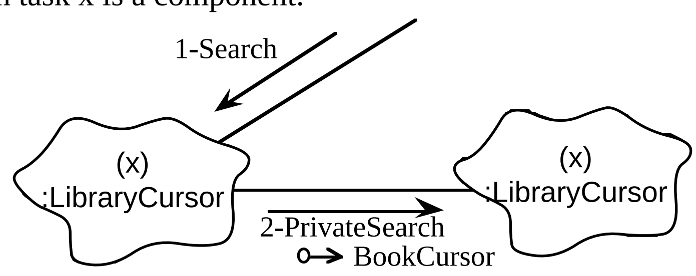
We saw a small example of a Base-Derived collaboration in the previous example. Such collaborations occur when a derived class exchanges messages with its base. Such collaborations are often very intimate; base and derived classes know a lot about each other and can take advantage of that knowledge. Such collaborations typically involve short term violations of class invariants, i.e. they temporarily leave the class in an illegal state between messages. But these invariants are always restored prior to the end of the collaboration.



Here we see an elaboration of part of the previous example. The `LibraryCursor` object initializes itself by sending itself the `Initialize` message. The `BookCursor` base class handles this message and sends the `InitializeDerived` message back to the derived class (probably via virtual deployment). Thus, the base portion of the class is initialized first, and then the base class initializes the derived class. In between these two messages, the object is in an invalid state, being only partially initialized. Certainly the `InitializeDerived` method should be private and virtual.

•Auto-Collaboration

Auto-collaboration occurs when an object sends a message to itself. This is the most intimate of all collaborations, since the object is generally talking to itself. Such collaboration is typically used to encapsulate portions of the implementation. For example, task x may be a component of many of the methods of class Y. Rather than coding task x in each of these methods, it makes better sense to create a new method which performs task x. Certainly such a method should be kept private, since its function is never meant to appear in isolation from the other methods of which task x is a component.



Here we see a typical case of auto-collaboration. When a `LibraryCursor` object is sent the `Search` method, it invokes the `PrivateSearch` method. The data item sent along is presumably its own base class. Notice how this encapsulates the task of searching within the `PrivateSearch` method. No other method of this class knows the details of a search.

USING FRIENDSHIP IN COLLABORATION

In one of the examples above, we used friendship to aid the collaboration of siblings. Friendship is also sometimes used in peer-to-peer collaborations. In early versions of C++, before the `protected` keyword was added, friendship was also used to support base-derived collaborations. In fact, the proliferation of base classes declaring their derivatives as friends was a principle factor in the decision to add `protected` access to the language.

Friendship allows unrelated classes to participate in intimate collaborations. This is important when several classes are working together to present a single abstraction. As a case in point, take the example of the `LibraryCursor`. This class collaborated with its sibling `AisleCursor` to present a single abstraction: that of searching the entire library for books. This collaboration required that the two classes be friends.

Such multi-class abstractions are an important design technique. There are situations where it is not practical or possible to represent an abstraction as a single class. A good example of this is *iterators.* Container classes and their iterators represent a single abstraction. But there is simply no good way to represent this abstraction as a single class.

Another role of friendship is to prevent private portions of a collaboration from leaking out into the public arena. Again, the `LibraryCursor` class provides us with an example. The `PrivateSearch` method is a dangerous method to make public. It badly violates the invariants of the `BookCursor` abstraction. Friendship allows these dangerous functions to remain private to the abstraction, and to be used by the friends participating in that abstraction.

When many classes collaborate, the use of friendship to solve the problems of access and efficiency will result in classes that are bound tightly to each other. Sometimes they can be so tightly bound that they cannot be separated from each other.

Certainly we want to avoid, at all costs, huge networks of classes which are all friends and which all take great liberties with each others internal parts. Such a perversion could not be called object-oriented. Also, we want to avoid the temptation to use friendship to join two very separate abstractions. If such abstractions need to be joined in some way, the joining should generally be accomplished through their interfaces, or through an intermediary class.

However, when two ore more classes are truly part of the same abstraction, then tight binding and friendship should not be discouraged. As Rumbaugh says: "Some object-oriented authors feel that every piece of information should be attached to a single class, and they argue that associations violate encapsulation of information into classes. We do not agree with this viewpoint. Some information inherently transcends a single class, and the failure to treat associations on an equal footing with classes can lead to programs containing hidden assumptions and dependencies." [*Object Oriented Modeling and Design*, Rumbaugh et. al., Prentice Hall, 1991]

Since friendship can only be given, and cannot be taken, the choice of who to give friendship to becomes a design decision. This means that the class is *designed* to collaborate with certain special friends. The collaborators become members of a team which work more closely together than normal in order to achieve a single end. Thus, encapsulation is not lost, nor even compromised. The "capsule" simply widens to enclose all the friends.

SUMMARY

In this article we have examined collaboration. We have shown that all the behaviors of an application are implemented through collaborations. We have shown how collaborations are first detected in the analysis phase of a project, and how their static and dynamic elements can be expressed using the Booch notation. We have shown how the static and dynamic views can be iterated to provide successive refinement of the application's design. We have discussed the various types of collaborations, and typical situations when they may be used. Finally we have discussed the role of friendship in collaborations.

Collaboration is at the heart of OOA/OOD. The proper design of an object-oriented application depends upon a thorough and detailed understanding of the collaborations which implement its behaviors.