CPS211 Lecture: Representing Associations in Java; Collection

Last revised July 24, 2008

Objectives:

1. To show how associations can be represented by references
2. To show how associations can be represented by collections

I. Implementing Associations using Java References

A. Of course, the associations that are identified during the design phase will eventually have to be implemented in the Java code implementing the various classes. This typically takes the following form:

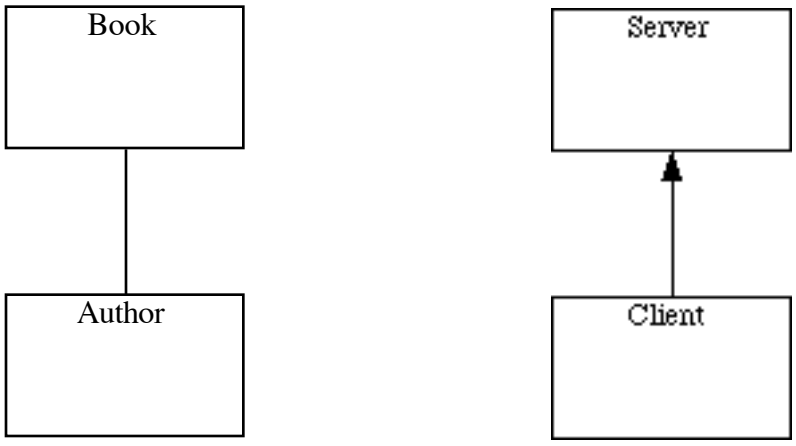
1. For each different association that relates objects of a given class to other objects, there will be a field in each object containing a link to the appropriate object(s).

a) If the association is bidirectional, *each* participating class will need such a field.

b) If the association is unidirectional, only the class whose objects need to know about their partner(s) will have such a field.

EXAMPLE:

Consider two cases that we looked at earlier:



In the first case, each Book object will need a field linking to the associated Author object(s), and each Author object will need a field linking to the associated Book object(s).

In the second each Client object will need a field linking to the associated Server object(s), but *not* vice-versa.

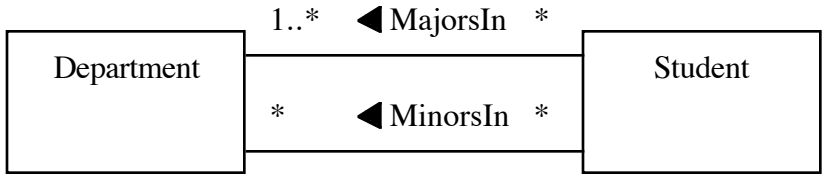
(This reduction in information that needs to be maintained is why we consider the possibility of unidirectional associations.)

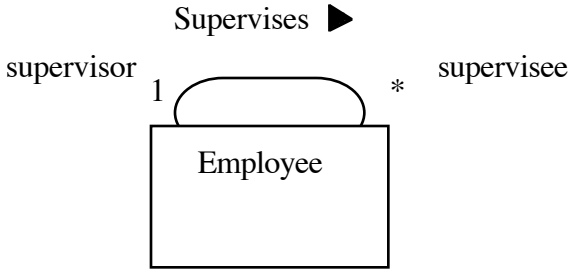
2. Frequently, the field name will be derived from the name of the association, or from the role names, if such are present. If not, the name will often come from the name of the class at the other end.

   EXAMPLE: A Book object may contain a field called authors, and an Author object may contain a field called books.

   EXAMPLE: A Client object may contain a field called server. The Server object, however, would not contain a field called client.

   EXAMPLE: Consider two cases we looked at earlier





In the first case, a Student object might have fields called majors and minors. A Department object might likewise have fields called majors and minors. Or the fields might be named majorsIn/minorsIn in the Student and still probably majors and minors in the Course.

In the second case, an Employee object might have fields called supervisor and supervisees. (Note the plural in the case of the latter name - the role is supervisee, but one supervisor can supervise multiple people.) Or the fields might be named supervises and supervisedBy.

B. There are a variety of different implementation approaches that can be used to actually realize the links.

1. If a given object can relate to only one other object in a given association (there is a "1" at the other end of the link), the easiest approach is to use a Java reference to the other object.

2. If the multiplicity is "0..1", the same strategy can be used, with the reference being null if there is no related object for this association.

3. If the multiplicity is some fixed, small integer, or is limited by some small fixed integer, then multiple fields can be used, or a field whose value is an array.

   EXAMPLE: Suppose we assume that a given student can have at most three majors and at most two minors. Then we might include fields like the following in a Student object:

```
Department major1, major2, major3;
Department minor1, minor2;
```

   or

```
Department [] major;
Department [] minor;
```

   (Of course, there are dangers if you cannot be sure that the upper limit is definite. However, three majors is probably enough for anyone!)

4. When (the upper end of) the multiplicity range is "*" (or some large integer), the first approach won't work, and the second is tricky unless you know when the object is created how many other objects it will be related to (since arrays in Java are created with a fixed size). A more flexible approach results from using Collections, which we will discuss next.

II. Collections

A. The Collections facility was added to Java as a part of JDK 1.2

1. A Collection is a group of objects that supports operations like:

   a) Adding objects to the Collection.

   b) Removing an object from the Collection.

   c) Accessing individual objects in the Collection.

   (Note that an array can be thought of as a very simple and limited form of Collection, but doesn't offer the full elegance of the Collections facility in the Java library)

2. Java Collections are of three basic types:

   a) Sequences are collections in which the contents are regarded as having some sequential order. (Note: the Java library calls these "Lists")

      (1) If a collection is a sequence, it is legitimate to ask questions like "what is the first object in the sequence?" or "what is the last object?" or "what is the ith object?". (Provided the collection is non-empty, in the first two cases - or has at least i+1 elements, in the last case - since elements are numbered starting at 0 - so to get, say, item 2 the collection must contain at least three elements.)

      (2) It is also legitimate to make requests like "add this object at the very front" or "add this object at the very end" or "add this object in position i". (Provided the collection has at least i elements in the last case.)

      (3) Finally, it is legitimate to make requests like "remove the first object" or "remove the last object" or "remove the ith object". (Provided the collection is non-empty, in the first two cases - or has at least i+1 elements, in the last case.)

   b) Sets are collections in which a given object may appear at most once, and there is no ordering.

      (1) If a collection is a set, it is legitimate to ask the question "is this particular object in the collection?" (yes or no).

(2) It is also legitimate to make the request "add this object to the collection". (Provided it's not already there.)

(3) Finally, it is legitimate to make the request "remove this object from the collection". (Provided it is in the collection to begin with.)

c) *Maps* are collections of key-value pairs. (Actually, Java Maps are not technically Collections due to some implementation issues, but it is common to speak of them as "small c" collections.

(1) Maps are often used for qualified associations, with the qualifier serving as the key, and the associated object the value

EXAMPLE: The qualified association between a college and its students could be represented by a map (stored in the college object) consisting of pairs where the student id is the key and the corresponding student object is the value.

(2) If a collection is a map, it is legitimate to ask questions like "what value - if any - is associated with the following key?" or "does this map contain the following key?".

(3) It is also legitimate to make the request "put the following key-value pair in the map". This can have one of two effects:

(a) If the key was not in the map to begin with, it is added with the specified value.

(b) If it was in the map, but with a different value, the old value is removed and the new value is associated with the key.

(4) Finally, it is legitimate to make the request "remove the following key from the map". If the key was in the map, it and its associated value are removed; if not, nothing happens.

d) For all types of Collections, it is possible to create an Iterator object that makes it possible to access each item in the collection once.

(1) For sequences, the order in which items are accessed by an Iterator is the sequential order first, second ...

(2) For sets and maps, the order is implementation-determined.

(3) In the case of maps, the Iterator is actually obtained from either its set of keys or its set of values.

3. The Java Collections library contains two or more implementations for each of the different types of collection. The different implementations of a given type of Collection have the same behavior, but have different performance characteristics.

   a) For List (Sequence), the Java library supplies:

      (1) LinkedList - good if the list changes size (grows or shrinks) frequently), good for accessing either end of the list, but slower when accessing items in the middle of the list

      (2) ArrayList - good if accessing elements by specific position, but slower for adds and removes.

   b) For Set, the Java library supplies:

      (1) HashSet - more efficient in most cases

      (2) TreeSet - an iterator will access the elements of the set in a specific order based on their value (e.g. Strings would be kept in alphabetical order.)

   c) For Map, the Java library supplies:

      (1) HashMap - more efficient in most cases

      (2) TreeMap - an iterator obtained from the key set will access the elements of the map in key order.

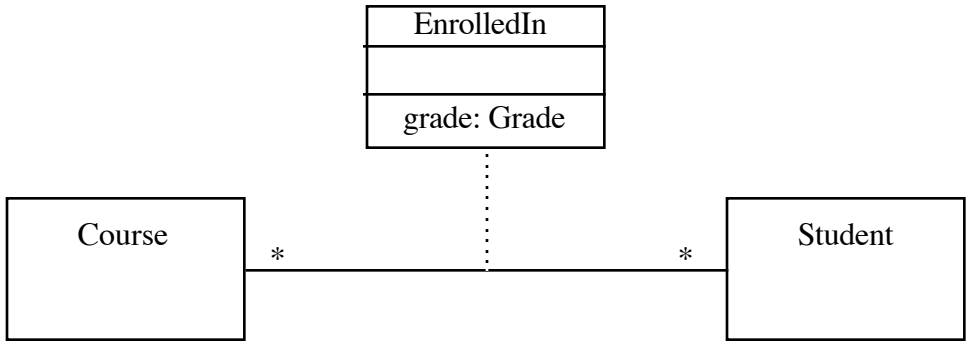
4. We will devote a lab to working with Java Collections

B. We noted earlier that if an association has attributes associated with the association itself (not just the participating objects), an association class can be used. In this case:

1. Each participating object contains a reference to the association class object.

2. The association class object contains references to each of the related objects.

   EXAMPLE:




```
class Course {
     ...
     (Some sort of collection of references
to
       EnrolledIn objects) enrolledIn;
     ...
  }

class Student {
     ...
     (Some sort of collection of references
to
       EnrolledIn objects) enrolledIn;
     ...
  }

class EnrolledIn {
     ...
     Course course;
     Student student;
     Grade grade;
     ...
  }
```

C. Now, let's think about the various associations in the Video Store problem and how they might be represented by Java collections: For now, let's restrict ourselves to a subset of the requirements, assuming only movies (not games) are being rented. It will also prove helpful to assume that there is a singleton object (perhaps of a class called Store) which represents the Store and "owns" all the other objects.

1. First though, we need to consider an interesting (and actually quite tricky) issue that arises in connection with rentable items.

a) Typically, a video store owns multiple copies of popular movies.

(1) Presumably, we need a separate object for each copy, since each can be rented to a different customer, be due on a different day, etc..

(2) At the same time, we want to associate all kinds of information with a copy - its title, its actors, its director ... etc - but storing all this information multiple times (once for each copy) is problematic

(a) Wasteful of space

(b) Makes extra work when a new copy comes in

(c) Suppose we needed to correct a piece of information - e.g. maybe we had recorded a wrong actor. We would need to make this change in <u>each</u> copy.

(3) Moreover, when we accept a reservation for a particular movie, we don't want to associate the reservation with a <u>specific</u> copy - we want to associate it with the movie itself. Why?

ASK

b) There is a very problematic way to handle a case like this:

(1) Create a separate <u>class</u> for each title. Thus, we would have a class Shrek3, a class BourneUltimatum ...

(a) The various items of information we would want to record about a movie could be represented as <u>static</u>

fields of the appropriate class - e.g. we would have something like

```
class BourneUltimatum {
   static String leadActor="Matt Damon";
   ...
```

(b) A copy would be represented by an object of the appropriate class. Thus, if we had 10 copies of The Bourne Ulitimatum, each would be represented by an object of class BourneUltimatum - 10 in all.

(2) Why is this very problematic?

ASK

(a) A serious problem is that we have to create a new class every time a new movie comes out. This means that the software would have to be revised and recompiled about once a week!

(b) A similar problem is that any change to information stored about a movie would necessitate revising and recompiling the software. While this may seem relatively improbable in this case, it could be a serious problem in other, similar cases.

c) A much better way is to make use of a design pattern, which is, in essence, a good solution to a commonly-occurring tricky problem. (We will talk about design patterns later in the course.) In this case, we want to use a pattern known as the Abstraction-Occurrence pattern.

(1) In essence, what we want to do is to use two classes to represent movies.

(a) One class - which we might call something like `Movie` - represents the abstraction.

i) There would be, then one object of class `Movie` for Shrek 3, another for the Bourne Ultimatum ....

ii) This one object holds all the information that pertains to all copies of the movie - its actors, director ...

iii) When a new movie comes out, we create a new object of this class to represent it.

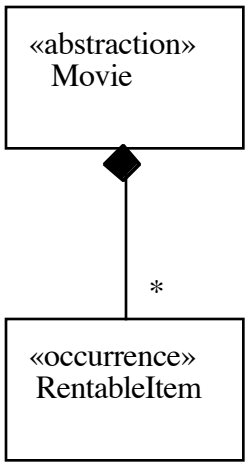
(b) A second class - which we might call `RentableItem` - represents the occurrence.

i) There would be one object of this class for each physical copy of a movie that we own - thus, if we had 10 copies of the Bourne Ultimatum, there would be 10 objects of class `RentableItem` representing them.

ii) Each object of class `RentableItem` would be associated with the object of class `Movie` to which it pertains.

(c) If the store owned 10,000 DVD's in all, representing 3000 different movies, there would, in total, be 10,000 objects of class `RentableItem`, and 3000 objects of class `Movie`.

(d) We might show this in UML as follows:



i) Composition is appropriate here, because,

(1) At least as far as the store is concerned, a movie is composed of its copies

(2) Each copy is associated with one and only one movie

(3) A copy of one movie can never be associated with a different movie

ii) A multiplicity of * is appropriate on the RentableItem end, because a given Movie can have an arbitrary number of copies. (We might even have 0 for a short time, if our only copy is lost and we're waiting for a reorder to come in)

iii) Bidirectional navigability is appropriate, because me want to be able to answer both of the following kinds of question:

(1) What copies of "The Bourne Ultimatum" do we own?

(2) What is the title of the Movie that rentable item 1234 is a copy of?

2. Given this, we want to have an association between the store and the movies it owns. Clearly this is 1..*.

a) What navigability is appropriate here?

ASK

b) What type of collection is appropriate in the Store object to hold references to Movie objects for this case?

ASK

Map, keyed on something like title of the movie. The Store object would have a declaration like

```
Map < Movie > movies;
```

But nothing would be needed in the Movie object, since we have no need to navigate from a Movie to the (one and only) Store

3. Again, we need an association between the store and the rentable items it owns as well. Clearly, this is also 1..* When a customer wants to rent an item, the ID number on the item is

used to find the appropriate object to associate with the customer.

a) What navigability is appropriate here?

ASK

b) What type of collection is appropriate in the Store object to hold references to RentableItem objects for this case?

ASK

Map, keyed on the item's ID. The Store object would have a declaration like

```
Map < RentableItem > items;
```

But nothing would be needed in the object, since we have no need to navigate from an item to the (one and only) Store

4. The association between the store and its customers is clearly 1:*. When a customer wants to rent one or more items, he/she presents a card, and the ID number on the card is used to access the stored information on the customer.

a) What navigability is appropriate here?

ASK

b) What type of collection is appropriate in the Store object to hold references to Customer objects for this case?

ASK

Map, keyed on the customer's ID. The Store object would have a declaration like

```
Map < Customer > customers;
```

But nothing would be needed in the Customer object, since we have no need to navigate from a Customer to the (one and only) Store

5. We have already said that we need an association between a customer and the specific items the customer has out.

a) What is the multiplicity here?

   0..1 : * - an item is either out to 1 customer, or it's on the shelf; but a customer can have any number of items out

b) In this case, we probably want bidirectional navigability, so we can answer questions like

   (1) What items does customer 1234 have out?

   (2) What customer rented item 5678 (that was just returned)

c) Observe that the act of checking out an item involves attributes - e.g. date due. We can represent this either by using an association class object, or by recording the date due as an attribute of the copy (since it can only be checked out to one person at a time)

   ASK class for how each approach would be set up in terms of Java collections

6. We also need an association between a customer and the movies he/she has reservations for.

   a) What is the multiplicity here?

      ASK

      It must be * .. * - any number of customers can have a reservation for a given movie, and a customer can have reservations for multiple movies.

   b) In this case, we probably want bidirectional navigability, so we can answer questions like

      (1) Who has reservations for "The Bourne Ultimatum"?

      (2) What reservations does customer 1234 have?

   c) Presumably, we want to keep track of reservations on a first-come first-served basis. This being the case, what sort of collection is needed in a Movie object to keep track of the customers who have a reservation for the Movie?

      ASK

7. What would we need to do to this structure to also handle games?

   ASK

   a) Create a new base class (Title?) with subclasses Movie and Game.

   b) Copies and reservations are now associated with a Title - the mechanics are the same for either Movies or Games. That's all we need to do!