Implementing Associations

Maintaining pointer integrity the easy way

Frank Hellwig
Dr. Dobb's Journal
June 1998

Many object-oriented C++ programs consist of objects that are dynamically linked at run time using pointers. The challenge is to maintain valid pointers as object references change and objects are destroyed. Implementing and using associations is a solution that lets you dynamically create an object model and be assured of valid references throughout your program.

Associations, as described in the Unified Modeling Language (UML) (see Unified Modeling Language for Object Development, Version 1.0, by Grady Booch et al., Rational Software, 1997) are relationships between pairs of objects. Associations can exist between two objects of different classes or between objects of the same class (self associations).

There are two types of associations: by-value or by-reference. A by-value association implies containment, meaning that objects are actual data members of another object. A by-reference association means that an object has pointers to other objects. This article deals with by-reference associations.

The simplest association is when one object has a pointer to another object. This is known as a unidirectional association. If the other object also has a pointer to the first object, the association is a bidirectional association.

Bidirectional associations are common because, frequently, both objects need to know about each other, such as in a parent-child relationship. Having bidirectional associations imposes the additional requirement of referential integrity. When one pointer is updated, the inverse pointer in the other object must also be updated.

Aggregations are a special type of association representing a whole-part relationship. This means that some objects are parts of another object higher up in the object hierarchy. The parts are dependent on the whole for their existence and are automatically destroyed when the aggregate whole is destroyed.

Whole-part relationships can imply ownership (you own your driver's license) or composition (a flower is composed of a stem, leaves, and petals). In either case, there is a dependency between the whole and the part; if the flower dies, then so do the parts. In this article, the term "owns" is used to indicate a dependent whole-part relationship even though other semantics could be applicable.

Associations are called "acquaintance relationships" in Design Patterns: Elements of Reusable Object-Oriented Software, by Erich Gamma et al. (Addison-Wesley, 1995). I first thought of presenting associations as a design pattern. As it turns out, associations are really

an underlying mechanism to help implement other design patterns such as the Composite pattern.

Background

At my previous employer, the Boeing Company, we designed and implemented the mission planning system for the Enhanced Tactical Radar Correlator (ETRAC) system. ETRAC is a deployable system to provide Army field commanders with imagery-based battlefield intelligence from the U-2 spyplane. The mission planning system performs the navigation planning, collection planning, and mission monitoring for ETRAC U-2 operations.

Users interact with the system through a GUI complete with map underlays and various user-interface elements to manipulate the objects in the system. These objects consist of aircraft, sensors, targets, routes, waypoints, missions, plans, and other related objects, and they form a model of the real environment. Each real-world object is encapsulated as a class. While using the mission planner, the user creates, modifies, and deletes objects.

The objects are related by associations and aggregations. A route consists of a series of waypoints. An aircraft has a specific sensor on board. Each mission is associated with a particular route.

Originally, we implemented the system using pointers and lists of pointers to link the objects together. Maintaining pointer integrity as objects were deleted became a virtually impossible housekeeping chore. The complexity increased once we added aggregations. For example, deleting a collection plan caused all of the scenes in that plan to be deleted. But each scene is also owned by a target (you can't have scenes without a target). The pointers to the scenes in either the plan or target had to be updated when the other was deleted.

We developed an association implementation that is useful and expressive. It is based on named sets of objects stored in hash tables. Both the object and its hash table are persistent, making the entire model serializable to a stream.

Associations by Example

Figure 1 shows a class diagram of a simple world. It uses the UML notation, except that multiplicity is shown by a solid dot instead of a text expression such as {0..*}. Each rectangle represents a class with the class name inside the rectangle.

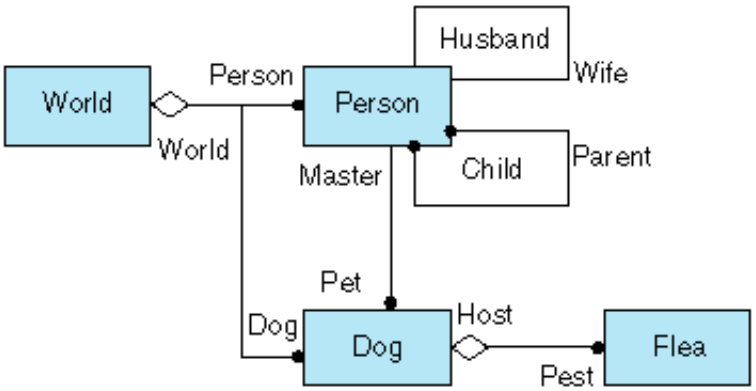

Figure 1: A class diagram for a very simple world.

The class diagram represents an object hierarchy. There is no requirement to model a hierarchy, but by doing so, all objects can be destroyed or saved by destroying or saving the top-level object (an instance of the World class of which there is presumably only one). The world is an aggregation of people and dogs. This is indicated using the diamond next to the aggregate object. The implication is that the people and dogs will cease to exist upon the world's demise.

At the next level is the aggregation of Flea objects. This is an example of aggregation being used to model dependency. The fleas depend on the dog for their existence; a dog is not made of fleas.

Aggregations are transitive. When the World object is deleted, all people and dogs are automatically deleted, which causes all fleas to be deleted.

Each line connecting two classes is an association. The endpoints of each association are called "roles." The name at each end of an association is the role name. The role name next to a class indicates how the other objects view and refer to instances of the class. Role names can be the same as the class name, or they can indicate other associations, such as husband-wife or child-parent (which are both self associations).

Role names must be unique within a class. A dog cannot refer to the world as its master because that name is already used in the master-pet association.

Better than Pointers

Why not implement associations as pointer data members? To answer this, consider a simple association in the example model: the master-pet association between a person and his or her dog.

This is a one-to-many relationship since each person can have more than one dog, but each dog only has one master. (I'll ignore the case where a dog belongs to a family.) To implement this using pointers, I'll put a pointer to the dog's master in the Dog class and a list of dog pointers in the Person class.

Being good C++ programmers, we make the pointer and pointer list private data members, and write accessor and mutator functions. Example 1(a) partially declares the Person and Dog classes. Assume there is a Vector collection class that is initially empty.

```
class Dog;
class Person {
public:
  void addPet(Dog* d)
    { pets_.add(d); }
private:
  Vector pets_;
};
class Dog {
public:
  Dog() : master_(0) { }
  void setMaster(Person* p)
    { master_ = p; }
private:
  Person* master_;
};
```

Example 1(a): Partial declaration of the Person and Dog classes.

On the surface, Example 1(a) seems like a reasonable C++ approach. The data members are private, are initialized properly, and are set only via mutator functions. Example 1(b) creates and associates a master and his dog.

```
Person* bill = new Person();
Dog* fido = new Dog();
bill->addPet(fido);
fido->setMaster(bill);
```

Example 1(b): Creating and associating a dog and its master.

The problem is that setting up a link requires two function calls -- addPet() and setMaster(). This has maintenance implications. Imagine having to worry about always calling the correct pair of functions on corresponding objects every time a link is created or deleted.

An alternative is to only call setMaster(), and have setMaster() call addPet(); see Example 1(c). But addPet() is still a public function and could be called separately, thereby establishing only half of the link. In this case, Martin Fowler (Analysis Patterns: Reusable Object Models, Addison-Wesley, 1997) mentions that the C++ friend construct can be used. The Dog class could be declared a friendclass of the Person class. This way, addPet() can be made private and thereby accessible only by the two classes.

```
Dog::setMaster(Person* p)
```

```
{
  master_ = p;
  master_->addPet(this);
}
```

Example 1(c): Calling addPet() from within setMaster().

But this solution is worse than the previous. It violates encapsulation and makes maintenance very difficult. You can't determine how the private data members of a class are being used or modified if another class has direct access. It's possible to make only `setMaster()` a *friend* function, but I doubt that this makes maintenance any easier.

Requirements

We faced the same situation in our mission planning system as the one just described, only on a larger scale. While designing our association implementation, we decided on some key requirements for reliable and useful associations.

- Associating object A with object B must automatically associate object B with object A. Similarly, dissociating object A from B must automatically dissociate object B from object A.
- Destroying object A must automatically remove all associations A has with any other objects, and those objects must no longer be able to reference A.
- Destroying aggregate object A must automatically destroy all of A's parts. Destruction of an objects' parts is transitive. Nonhierarchical aggregations must be supported. There are two types: multiple aggregations (A and B own C) and cyclical (A owns B owns C owns A).
- The associations must be persistent so that they can be stored, thus preserving the state of the application.

Our implementation of associations uses the Rogue Wave Tools.h++ foundation class library. (The Tools.h++ library provides more than 130 C++ components such as string, collection, date and time, internationalization, and streaming classes, plus an interface to the Standard C++ Library.) This is just one option. Any class library supporting collection classes can be used as a base. We found that Tools.h++ is reliable and well documented. The source code for the AssociationObject and AssociationRole classes is available electronically from DDJ.

Associations are implemented with two classes: AssociationObject and AssociationRole. Any class used in an association must be derived from the AssociationObject class. The AssociationRole is only used within the AssociationObject class.

The AssociationObject Class

AssociationObject is a base class from which all objects using associations are derived. It contains a hash dictionary of roles. The AssociationObject class is derived from

RWCollectable. As a result, it can be used in collections (the AssociationRole), and it can use RWCollectable's persistence mechanism to save the roles.

Listing One is the public interface to the AssociationObject class. The associateWith() function associates this object with another object using the specified role. The inverse role name is specified so that the inverse link in the specified object can be established. The associateWithPart()function does the same thing, except it establishes this object as the whole and the specified object as the part. The specified object will be destroyed when this object is destroyed.

```
class AssociationObject : public RWCollectable
{
  RWDECLARE_COLLECTABLE(AssociationObject)
public:
  AssociationObject();
  ~AssociationObject();
  void associateWith(AssociationObject *obj,
                     const RWCString &name,
                     const RWCString &inverseName);
  void associateWithPart(AssociationObject *obj,
                         const RWCString &name,
                         const RWCString &inverseName);
  void dissociateFrom(AssociationObject *obj, const RWCString &name);
  void dissociate(const RWCString &name);
  AssociationObject *getObject(const RWCString &name) const;
  RWOrdered getObjects(const RWCString &name) const;
  RWOrdered getObjectsSorted(const RWCString &name) const;
  void restoreGuts(RWvistream &);
  void saveGuts(RWvostream &) const;
};
```

Listing One: The AssociationObject public interface.

The dissociateFrom() function undoes an association to a specific object. The dissociate()function dissociates all objects associated with this object in the specified role.

There are three functions to get the objects in a role. The getObject() function just gets the first object in a role. This is for one-to-one or many-to-one relationships. The getObjects() function gets all of the objects in the role. The getObjectsSorted() function sorts the objects based on the RWCollectable's virtual compareTo() function.

In these functions, the role name can be a compound name. Dots are used to separate roles. For example, world->getObjects("dog.flea"); gets the fleas for all of the dogs in the world. The role names "dog" and "flea" are recursively searched for objects in each role. The objects in the last role are returned.

The saveGuts() and restoreGuts() functions are standard Rogue Wave class functions that

keep the object persistent to and from a stream.

The AssociationRole Class

The AssociationRole class is used only by the AssociationObject class. An AssociationRole is an identity set of object pointers and is stored in a hash dictionary in the AssociationObject. Listing Two is the declaration of the AssociationRole class.

```
class AssociationRole : public RWIdentitySet
{
  RWDECLARE_COLLECTABLE(AssociationRole)
public:
  // Default constructor for persistence.
  AssociationRole() {}
  AssociationRole(const RWCString &inverseName, RWBoolean isAggregate);
  RWCString getInverseName() const
  {
    return inverseName_;
  }
  RWBoolean isAggregate() const
  {
    return isAggregate_;
  }
  void restoreGuts(RWvistream &);
  void saveGuts(RWvostream &) const;
private:
  RWCString inverseName_;
  RWBoolean isAggregate_;
};
```

Listing Two: The AssociationRole declaration.

Each AssociationRole has an inverse name. The inverse name is the role name of the opposite role in the association. The inverse name is maintained in the role so that both ends of the association are removed when two objects are dissociated, thereby preserving referential integrity.

The aggregate flag indicates that the role contains object parts that must be destroyed when the client object is destroyed. For example, a World object contains a "person" role. This role contains pointers to Person objects. The inverse name is "world" since that is how Person objects refer to the World object. Finally, the aggregate flag is set because the Person objects in the role are an aggregation of parts owned by the World object.

Multiplicity

If you examine the UML, you will find an explanation of the multiplicity of a role. For example, the multiplicity of a role can be 0..1 (optional), 0..* (many), or 1..3, 5..7 (specified ranges).

We did not implement multiplicity because it is difficult to enforce. If a role's multiplicity is 2..6, then having zero objects is invalid. This means that at least two objects must be supplied in one step. This is hard to do since the associateWith() function works with single object pointers, not with sets of objects.

Additionally, if the role contains two objects, then the deletion of either one of the two objects must be prevented. Since the delete operator is valid for any dynamically created object, some other delete function must be provided that handles multiplicity checking. We simply didn't want to make our implementation that complex.

We therefore regard any role as having a multiplicity of zero or more, and leave enforcement of more restrictive situations to member functions of derived classes. For example, if a person can have no more than three pets, we would write the addPet() function as in Example 2.

```
void Person::addPet(Dog *dog)
{
  int ndogs = getObjects("dog").entries();
  if (ndogs > 3)
  {
    cerr << "Too many dogs." << endl;
    return;
  }
  associateWith(dog, "dog", "person");
}
```

Example 2: The addPet() function restricting the number of pets to three.

Sample Program

Listing Three is a C++ program that uses the AssociationObject class to implement an object model according to the class diagram in Figure 1.

```
/* assoctst.cc  */


#include accocobj.h
#include <rw/rstream.h>


class World : public AssociationObject
{
};


class Person : public AssociationObject
```

```
{
public:
  Person(World *world)
  {
    world->associateWithPart(
        this, "person", "world");
  }
};


class Dog : public AssociationObject
{
public:
  Dog(World *world)
  {
    world->associateWithPart(
        this, "dog", "world");
  }
};


class Flea : public AssociationObject
{
public:
  Flea(Dog *dog)
  {
    dog->associateWithPart(this, "pest", "host");
  }
};


void main()
{
  // Create the objects.
  World *world = new World();
  Person *dick = new Person(world);
  Person *jane = new Person(world);
  Person *tiffany = new Person(world);


  Dog *spot = new Dog(world);


  new Flea(spot);
  new Flea(spot);
  new Flea(spot);
```

```
  // Create plain associations.
  dick->associateWith(jane, "wife", "husband");
  tiffany->associateWith(dick, "parent", "child");
  tiffany->associateWith(jane, "parent", "child");
  tiffany->associateWith(spot, "pet", "master");


  // Get Tiffany's parents.
  RWOrdered parents = tiffany->getObjects("parent");


  // Count the fleas in the world.
  cout << "There are "
       << world->getObjects("dog.pest").entries()
       << " fleas in the world."
       << endl;


  // Delete the object hierarchy.
  delete world;
}
```

Listing Three: Sample program that implements an object model.

Each class (except the top-level World class) gets its parent passed to it in the constructor where the association is created. An alternate implementation is to follow the Composite pattern (described by Gamma et al.), and provide functions such as addPerson() to the World class.

In main(), the classes are created, then the plain (nonaggregate) associations are created. The getObjects() function then is used to both get and count objects. Finally, the entire hierarchy is destroyed by simply calling delete on the top-level object.

Handling Aggregations

I've said very little about aggregations except that each role has an aggregate flag. The trick with aggregations is to destroy all aggregate objects without destroying the same object twice. Although none exist in Figure 1, strange aggregations are possible. An aggregation could be a cycle where object A owns object B, B owns C, and C owns A. This is rare. More common are multiple aggregations where one object is owned by more than one parent. (Recall that a scene is owned by both a collection plan and a target.)

Deleting an aggregate object follows a two-phase model similar to the destruction model for widgets in the X toolkit or in a mark-and-sweep garbage collector.

Phase one consists of marking objects for destruction. The first step is to mark self as being destroyed followed by marking all subobjects in aggregate roles as being destroyed. Each

marked object is added to an identity set of parts to destroy. The marking process continues recursively for each subobject not already marked for destruction. Phase two consists of destroying all objects in the identity set of parts to destroy.

Because the order in which subobjects are destroyed is nondeterministic, objects that are marked for destruction are no longer considered members of an association.

For example, consider deleting a Dog object, which thereby deletes the fleas associated with the dog. Say there's a function in Dog called computeInfestationFactor(). This function is called from the Flea constructor and destructor to compute some value whenever a flea is created or destroyed.

What happens when a Dog is deleted? C++ calls the destructors in a derived-to-base-class order; the Dog destructor is called followed by the AssociationObject destructor. It is in the AssociationObjectdestructor, not the Dog destructor, that subobjects are deleted. The Flea destructor must not be able to get a pointer to the Dog object, because such a pointer would not point to an object of type Dog. Calling getObject() with "dog" would return an object of type AssociationObject that could not be cast to a Dog object.

Since roles are needed during phase one, they are not destroyed until the parts are destroyed. Therefore, getting a pointer to an invalid Dog object must be prevented. This is done by not returning objects marked for destruction in any of the getObject() variants.

Persistence

Example 3 saves the object model to a file. The use of a binary output stream is specific to Rogue Wave's Tools.h++. (The Tools.h++ manual goes into depth about the Rogue Wave persistence mechanism.) But this example shows just how easy it is to save the entire object model, whether it consists of five or 500 objects.

```
ofstream f("world");
RWbostream strm(f);
strm << world;
```

Example 3: Making the object stream persistent.

If you examine the source code for the AssociationRole and AssociationObject, you will see two functions: saveGuts() and restoreGuts(). They call the base class version of the function and then save or restore the data members.

The AssociationObject saves and restores the role table. Each AssociationRole saves and restores its data members including the object set. Saving and restoring the object set then saves and restores each AssociationObject. The Rogue Wave persistence mechanism assures that objects are only stored once; a reference to the same object just stores an index to the stored object.

The cascade effect can be used to its best advantage if the object model is constructed as a

hierarchy with one root object that can be saved, restored, and deleted.

Conclusion

Any application with interactions between dynamic objects uses associations. The question is, do you use pointers and manage them on an individual basis or do you adopt a general-purpose solution? Using direct pointers and providing explicit functions makes the code easier to read since much less is hidden, but the potential for invalid pointers and the associated segmentation faults are the downside. Using associations, as described here, solves the problem of pointer maintenance by providing a reliable substructure that is common to all classes in an application.

Acknowledgments

I would like to thank and acknowledge my coworkers Jeff Travisano and Dave Worthington for reviewing this article.