Implementing Associations

Frank G. Hellwig*

January 14, 1997

Introduction

Do you have a C++ program where you have to handle many objects that are linked together with pointers and are dynamically created and deleted as the application is used? If so, and if you've been frustrated because it's next to impossible to maintain valid pointers, then you need associations. In this article, I will explain what associations are, why they are better than using pointers directly, and describe an implementation using the Rogue Wave Tools.h++ foundation class library.

Associations, as described in the Unified Modeling Language (UML)[1], are relationships between pairs of objects. Associations can be 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 a called a unidirectional association. If the other object also has a pointer to the first object, the association is called a bidirectional association.

Aggregations are a special type of association representing a whole-part relationship. The aggregate object (whole) is made of or owns the *subobjects* (parts). The subobjects are automatically destroyed when the aggregate object is destroyed.

Associations are called acquaintance relationships in Design Patterns[2]. Indeed, 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

Over the past year, our software team has designed and developed 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.

The user interacts with the system through a graphical user interface complete with map underlays and many forms to manipulate the

*Frank Hellwig is a Senior Software Engineer at ARGOSystems, Inc., A subsidiary of The Boeing Company and can be reached at *frank.hellwig@boeing.com*. He specializes in mission planning applications. His main interest is providing C++ solutions to help engineers work smarter and more efficiently.

objects in the system. These objects form a model of the real environment. This model consists of aircraft, sensors, targets, routes, waypoints, missions, plans, and other related objects. 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. This proved to be a disaster. Maintaining pointer integrity as objects were deleted became a virtually impossible housekeeping chore. Sure, we encapsulated all object references in member functions but each time a new association was added, the ripple effect was massive. The complexity increased once we added aggregations. Deleting a collection plan caused all 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.

At this point, I started researching associations both on the web and in books. The material either discussed associations from a high level or suggested implementations such as pointers or hash tables. I did not find a complete implementation.

Necessity is the mother of invention. We have, after many iterations, developed an implementation that is useful, safe, expressive, yet does not force a particular style of programming. It is based on named sets of objects stored in hash tables. Each object's hash table is persisted along with the object making the entire model serializable to a stream. The motivation for writing this article is to provide a detailed working implementation of associations.

Associations by Example

Figure 1 shows an object model of a very simple world. It uses the UML notation except that multiplicity is shown by a solid circle instead of a text expression. Each rectangle represents a class with the class name inside the rectangle.

The object model

The sample model represents an object hierarchy. Models don't have to be a hierarchy but it helps in deletion and persistence as will be explained later. The World class, of which there is presumably one instance, is the root object. It is an aggregation of two types of subobjects: people (instances of the Person class) and dogs. The diamond indicates aggregation. The direction of the aggregation implies that the people and dogs will cease to exist upon the world's demise.

At the next level is the aggregation of flea subobjects. This is an example of where aggregation is only used to model dependency. The fleas depend on the dog for their existence; a dog is not made of fleas.

Aggregation is transitive. When the world object is deleted, all people and dogs are automatically deleted which then causes all fleas to be deleted.

Roles

Each line connecting two classes is an association. The endpoints of each association are called *roles*. The word at each end of an association is the *role name*.

The terms *target* and *client* are used when referring to classes or objects in an associa-

tion. The role name indicates how the client views the target and is placed next to the target class in the object diagram. Role names can be the same as the target 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.

The association between dogs and fleas is unidirectional as indicated by the lack of a role name on the dog end of the association. Unidirectional roles in a by-reference association typically indicate that the first object has a pointer to the second but not the other way around. In our implementation, all associations are bidirectional although we provide convenience functions to model unidirectional associations. The semantics of the dog-flea association are that a flea doesn't need to know on which dog it lives.

Better Than Pointers

So why not implement associations as pointer data members? To answer this, let's 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. (We'll ignore the case where a dog belongs to a family.) To implement this using pointers, we'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. The following code fragments partially declare the Person and Dog classes. Assume there is a Vector collection class which is initially empty.

```
class Dog;

class Person {
public:
    void insertPet(Dog* d)
        { pets.insert(d); }
private:
    Vector pets;
};

class Dog {
public:
    Dog() : master(0) { }
    void setMaster(Person* p)
        { master = p; }
private:
    Person* master;
};
```

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

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

The problem here is that setting up a link requires two function calls. 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 `insertPet()`.

```
Dog::setMaster(Person* p)
{
    master = p;
    master->insertPet(this);
}
```

But `insertPet()` is still a public function and could be called separately thereby establishing only one half of the link. In this case, Fowler[3, page 274] mentions that the C++ friend construct can be used. The Dog class could be declared a friend class of the Person class. This way, `insertPet()` can be made private and thereby accessible only by the two classes.

Now we have an object model with half the classes being friends to the other half. This solution is worse than the previous. It violates encapsulation making 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 claim that this doesn't make maintenence any easier. Besides, using the friend mechanism establishes a close relationship between seemingly unrelated classes when the goal is to model the real world.

Requirements

The situation just described was what we faced in our mission planning system only on a grand scale. While designing our association implementation, we decided on some key requirements for reliable and useful associations.

1. Associating object A with object B must automatically associate object B with object A.

2. Deleting object A must automatically remove all associations A has with any other objects and those objects must no longer be able to reference A.

3. Deleting aggregate object A must automatically delete all of A's subobjects. Deletion of aggregate objects is transitive. Non-hierarchical aggregations must be supported. There are two types: multiple aggregations (A and B own C) and cyclical (A owns B owns C owns A).

4. The associations must be made persistable so that they can be stored thus preserving the state of the application.

5. The code to create the roles used in associations must be able to be localized to one routine so that the code can be validated against the object model diagram.

Implementation

Our implementation of associations uses the Rogue Wave Tools.h++ foundation class library. This is not the only implementation option. Any class hierarchy supporting collection classes can be used as a base. We found that Tools.h++ is expresive, reliable, and well-documented. The Rogue Wave classes are prefixed by the letters RW.

Associations are implemented with three classes:

- AssociationRole
- AssociationObject
- AssociationTable

Figure 2 is an inheritance diagram of these classes. For clarity, only the public and protected member functions are shown. Figure 3 shows the key elements of each class and how they interact. In the following class descriptions, a subset of the member functions relevant to associations are described. There are additional functions particular to the Rogue Wave implementation.

The AssociationRole class

The AssociationRole class is used only by the AssociationObject and AssociationTable

classes. An AssociationRole is a named set of object pointers and is stored in the client class of the association. The AssociationRole class is derived from the RWCollectableString class. The RWCollectableString data is the target name of the role and is used as the lookup key.

Each AssociationRole also has a client name. The client name is the target name of the opposite role in the association. Both the target and client names are contained in the role for the following reasons:

- When associating two objects, the client name is used to look up the inverse role in the target object pointing back to the client. This is how both sides of the association are automatically established satisfying requirement one.

- When dissociating two objects (either explicitly or at object destruction), the client name is again used to look up the inverse role and remove the pointer to the client object from the inverse target role. This satisfies requirement two.

The aggregate flag indicates that the role contains objects that must be deleted when the client object is deleted. The section *Handling Aggregations* addresses requirement three.

For example, a World object contains a "person" target role. This role contains pointers to Person objects. The client name is "world" since that is how the Person class viewes the world class. Finally, the aggregate flag is set because the Person objects in the role are an aggregation owned by the aggregate World object. (The terminology can get confusing.)

The AssociationObject class

The AssociationObject is a base class from which all objects using associations are derived. It contains a hash table of roles called the *role table*. The AssociationObject class is derived from RWCollectable for two reasons.

- An AssociationObject can be used in collections.

- The persistence mechanism provided by RWCollectable is used to store the role table to a stream (requirement four).

The AssociationObject class provides five functions; one to create the roles, two to manage associations, and two to query the associations. These are the five functions:

- `createRoles(`*classNam*e`)` – Copies the roles from the AssociationTable to the AssociationObject's role table.

- `associateWith(`*object, targetName*`)` – Associates self with the specified object using the named role.

- `dissociateFrom(`*object, targetName*`)` – Removes the association between self and the specified object. The target role must be specified because self could be associated with the object with more than one role.

- `getAssociation(`*targetName, objects*`)` – Gets the objects in the role specified by the target name.

- `associationEntries(`*targetName*`)` – Counts the objects in the target role. This is a convenience function so that a separate vector doesn't have to be declared.

In the last two functions, the target name can be a compound role name to traverse more than one level of classes. For example, the

following gets the fleas for all of the dogs in the world.

```
world->getAssociation("dog.fleas",
                      objects);
```

The role names "dog" and "flea" are recursively passed to `getAssociation()` which searches the role table for the appropriate roles. The role table must be initialized in the constructor of a class derived from AssociationObject. This is done by calling `createRoles(`*className*`)` in the derived class constructor. The class name is used to copy the roles for that class from the AssociationTable's role dictionary to the AssociationObject's role table.

The AssociationTable class

The AssociationTable is a static class that stores all of the associations in the object model. Each association or aggregation is stored as a pair of roles in a hash dictionary called the *role dictionary.* The role pairs are created using one of following functions.

- `createAssociation(`*className1, className2, roleName1, roleName2*`)` – Creates two roles so that object of class 1 can be associated with objects of class 2.
- `createAggregation(`*className1, className2, roleName1, roleName2*`)` – Creates an association but marks role 2 as an aggregate role.

The keys in the role dictionary are the class names. The value stored under each class name is a set of roles for that class. The role created with *roleName1* is inserted in the set stored under *className2* and the role created with *roleName2* is inserted in the set stored under *className1.* Now the function call to create an association or aggregation matches the object model diagram.

```
createAggregation("World", "Person",
                  "world", "person");
```

For unidirectional associations or aggregation, two other forms of these functions are provided that do not require the roleName1 parameter. In reality, they are just convenience functions that automatically create *roleName1* from *roleName2* by prefixing it with a tilde (~). For example, the client name for the "pest" role in Dog is "~pest."

An AssociationTable provides a single data structure which can be initialized in a single routine. This satisfies requirement 5.

Our original implementation did not use an AssociationTable and we were forced to specify the inverse name in every association. If the role did not exist, it was created on the fly. In case of an error, the wrong role would be created. The only way to check that the code matched the design was to search for all association creation calls and verify that they matched and that they correctly implemented the design.

Omissions

If you examine the UML, there are two association attributes that we decided not to implement: multiplicity and OR-associations.

Multiplicity

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 when the role is created. Coupling the creation of roles to the creation of objects would violate requirement five. 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. Not allowing the delete operator to be used was not an option. Our implementation therefore regards any role as having a multiplicity of zero or more and leaves enforcement of more restrictive situations to member functions of derived classes.

OR-Associations

OR-associations are associations where the same class can play one of two roles. For example, the parent-child association can be split into the role pairs parent-son and parent-daughter.

We did not implement OR-associations because using them makes the code more complex. We use an association table to store all role pairs. Only the target role name is specified when associating two objects because the client name is available from the table. With OR-associations, the inverse role must always be specified. In the Person constructor, you couldn't just write

```
associateWith(mother, "parent");
associateWith(father, "parent");
```

Instead, you would have to write

```
associateWith(mother, "parent",
      "son");
associateWith(father, "parent",
      "son");
```

OR-associations can also mask a bad design. Instead of a parent-son, parent-daughter OR-association, the gender distinction is much better modeled by having a gender value in the Person class and sticking with the original parent-child association.

Associations In Action

This example is a simple C++ program that uses the association classes to implement the object model in Figure 1. The file is partitioned into sections with an explanation preceding each section.

Header files

First, the header files for the AssociationObject and AssociationTable are included. The AssociationRole header file should not be included because it is used only by the AssociationObject and AssociationTable classes. The file `rw/sortvec.h` declares the RWSortedVector class and the file `rw/rstream.h` is a portable include file for streams (i.e. cout).

```
#include "assocobj.h"
#include "assoctbl.h"
#include <rw/sortvec.h>
#include <rw/rstream.h>
```

Class declaration

Next, all the classes are declared. Each class must call `createRoles()` in the constructor to copy the roles from the AssociationTable to the internal role table in the AssociationObject. In a real application, the class name strings would be `#defines`.

I chose to pass the aggregate object to the subobject in the Person, Dog, and Flea constructors. This is a reasonable approach because the aggregate object must exist before the subobject can be created.

```
class World :
  public AssociationObject
{
public:
  World() { createRoles("World"); }
};
```

```
class Person :
  public AssociationObject
{
public:
  Person(World* world)
  { createRoles("Person");
    associateWith(world, "world"); }
};

class Dog :
  public AssociationObject
{
public:
  Dog(World* world)
  { createRoles("Dog");
    associateWith(world, "world"); }
};
```

The Flea is a special case because of the unidirectional aggregation so the call to `associateWith()` is done from the dog's perspective.

```
class Flea :
  public AssociationObject
{
public:
  Flea(Dog* dog)
  { createRoles("Flea");
    dog->associateWith(
      this, "pest"); }
};
```

AssociationTable initialization

The AssociationTable is initialized in one function. Note how we can take Figure 1 and match it up exactly to the code to make sure that all associations and aggregations are created.

```
void
initAssociationTable()
{
  typedef AssociationTable AT;

  AT::createAggregation("World",
    "Person", "world", "person");

  AT::createAggregation("World",
    "Dog", "world", "dog");

  AT::createAssociation("Person",
    "Dog", "master", "pet");

  AT::createAssociation("Person",
    "Person", "husband", "wife");

  AT::createAssociation("Person",
    "Person", "parent", "child");

  AT::createAggregation("Dog",
    "Flea", "pest");
}
```

main()

The `main()` function starts off by calling `initAssociationTable()` and creating the objects.

```
int
main(int argc, char* argv[])
{
  initAssociationTable();

  // Create the objects.
  World* world = new World();

  Person* dick = new Person(world);
  Person* jane = new Person(world);
  Person* mary = new Person(world);

  Dog* spot = new Dog(world);

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

Next, the objects are associated with each other through nonaggregate (plain) associations.

Creating the plain associations is done outside the class. This prevents additional checks (a dog cannot have two masters) that could otherwise be done in member functions. A safe approach would be to call a `setMaster()` function which would then call `associateWith()`. In our mission planning system, we made the `associateWith()` and `dissociateFrom()` functions protected instead of public to enforce the use of member functions.

```
dick->associateWith(jane, "wife");
mary->associateWith(dick, "parent");
mary->associateWith(jane, "parent");
mary->associateWith(spot, "pet");
```

Objects are queried for information about our model.

```
// Get Mary's parents.
RWSortedVector parents;

mary->getAssociation(
  "parent", parents);

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

The programmer decides whether to get the association objects sorted or unsorted when calling `getAssociation()`. The second parameter in this function is a reference of type RWCollection, an abstract type. The association objects are inserted in the collection. The result is either sorted or unsorted depending on what type of collection is provided.

For example, we may or may not want to get the people in the world in alphabetical order. Sorting all the people takes time. If we are computing the average age, then there is no reason to sort the objects. On the other hand, if we are printing a list, then we certainly would want them in alphabetical order.

Since each object, except the world, is part of an aggregation, the whole object hierarchy can be reliably deleted with one call.

```
  delete world;

  return 0;
}
```

Handling Aggregations

I've said very little about aggregations except than each role has an aggregate flag. The trick with aggregations is to delete all aggregate objects without deleting the same object twice. Although not part of the example model in Figure 1, strange aggregations are possible. An aggregation could be a cycle where object A owns object B, B owns C, and C then owns A. This is rare. More common is 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.)

Two-phase model

Deleting an aggregate object follows a two-phase model similar to the destruction model for widgets in the X toolkit.

Phase one consists of marking objects for deletion. The first step is to mark self as deleted followed by marking all subobjects in aggregate roles as deleted. Each marked object is added to an identity set of objects marked for deletion. The marking process continues recursively for each subobject not already marked for deletion.

Phase two consists of deleting all objects in the identity set of objects marked for deletion.

Objects marked for deletion

Because the order in which subobjects are deleted is nondeterministic, objects that are marked for deletion are no longer considered members of an association. Consider deleting a dog object thereby deleting the fleas associated with the dog.

Let's say that 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? The Dog destructor is called followed by the AssociationObject destructor. It is in the AssociationObject destructor, 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 `getAssociation()` with "dog" would return an object of type AssociationObject.

Since roles are needed during phase one, they are not deleted until after the dependent subobjects are destroyed. Therefore, getting a pointer to an invalid Dog object must be prevented. This is done by not returning objects marked for deletion in `getAssociation()` or `associationEntries()`.

Persistence

The following three lines of code persist the object model to a file.

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

The use of a binary output stream is particular to Rogue Wave. The Tools.h++ manual[4] goes into depth about the Rogue Wave persistence mechanism. But this does show just how easy it is to save the entire object model; whether it consists of five or 500 objects.

If you examine the source code for the AssociationRole and AssociationObject in Figures 4 and 6, 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.

Source Code

Figures 4–9 are the source code listing for the files implementing the association classes.

- Figure 4: assocrol.h (AssociationRole class declaration).
- Figure 5: assocrol.cpp (AssociationRole member functions).
- Figure 6: assocobj.h (AssociationObject class declaration).
- Figure 7: assocobj.cpp (AssociationObject member functions).
- Figure 8: assoctbl.h (AssociationTable class declaration).
- Figure 9: assoctbl.cpp (AssociationTable member functions).

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? The former may seem adequate at first, but once the object model grows, along with the number of segmentation faults, you will want something better. The implementation presented in this article solves the problems of pointer maintenance by providing a reliable substructure that is common to all classes in an application.

References

[1] Grady Booch, Ivar Jacobson, James Rumbaugh. *Unified Modeling Language for Object–Development, Version 0.8 with Version 0.91 Addendum.* Rational Software Corporation, 1996.

[2] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns.* Addison-Wesley Publishing Company, Inc, 1995.

[3] Martin Fowler. *Analysis Patterns – Reusable Object Models.* Addison-Wesley Publishing Company, Inc, 1997.

[4] Thomas Keffer. *Tools.h++ Introduction and Reference Manual, Version 6.* Rogue Wave Software, Inc., 1995.