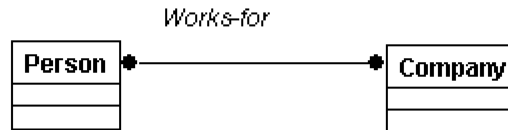
Implementing Reusable Binary Associations in C++
Terris Linenbach January 1996 (started October 1995)
terris@terris.com
Revision 295

Introduction: A Parable

Consider the following diagram:



This diagram denotes, clearly and concisely, that one person might work for many companies. Contractors and consultants typically work for several employers. This diagram also denotes that one company has many employees. Nothing could be simpler. Any talented programmer could code this relationship between people and companies.

But these facts weren't obvious when coding started at the ACME Coding Company. Their product, a personal tax accounting package, assumed that one person worked for at most one company at any given time. This model fit everyone at ACME, and no one questioned it.

After the product shipped, a loyal customer purchased the product. The customer was a software systems design consultant. He quickly found that he could not enter both of his employers into the product – it only allowed him to enter one. The product was free of obvious glitches. It didn't crash. It sported a friendly, fun, and attractive user interface. But the software was practically useless.

The designers and programmers at ACME were experienced, intelligent, and capable. The design wasn't flawed; it was just incomplete. In fact, not one shred of design documentation at ACME stated "one person, one company." This statement was hidden and scattered throughout thousands of lines of source code. The bug was eventually fixed, but at a large and unexpected cost. This left almost everyone on the team wondering, "How could this fiasco have been avoided?"

Objectives

Failure to understand relationships between objects can have a dire effect on a software project. Software that doesn't correctly model the real world will have incomplete or incorrect functionality. Modifying a program that assumes a "one to one" relationship to support a "one to many" relationship can be very time consuming. These types of unpleasant surprises are all too common in today's object-oriented programming shops.

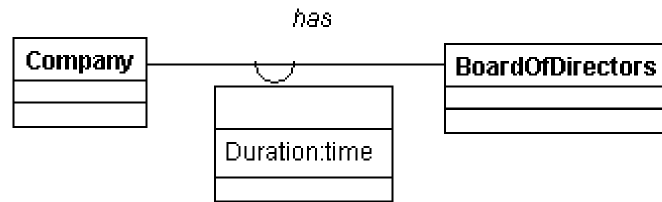
This paper has three objectives. The first goal is to encourage designers and programmers to think hard about object relationships before writing a single line of code. Secondly, this paper provides an alternative to the "pointer," which is the most common method of linking objects in C++. Finally, this paper demonstrates how to use the Object Modeling Technique (OMT) to document designs.

OMT Associations

Relationships between objects are known as *associations* in the OMT. Associations that relate two classes are known as *binary associations*. Three, four, and even more classes can participate in associations. However, binary associations are the easiest to conceptualize, and they tend to be the most common.

Associations have additional semantics: *multiplicity*, *bi-directionality*, *link objects*, and *link attributes*. This paper cannot cover all aspects of associations; instead, see chapter 3 of Object Oriented Modeling and Design.[1] The next few paragraphs will cover some of the basics.

Associations can express one-to-one, one-to-many, and many-to-many relationships between classes; this is known as multiplicity.[2] Consider a simple one-to-one relationship between a company and its board of directors, where the board has performed its duties for a particular duration:



A company and a board of directors participate in a simple association called "has", as in "a company has a board of directors." In this example, a path exists from a particular corporation to its board of directors. Likewise, a path exists from a board its company. The association is called bi-directional since both participants can locate each other.

The links between objects can themselves be modeled and implemented as separate objects. For example, if the company Acme Paint has five board members, then five link objects would associate the company with its board members.

Link objects can have their own attributes. Link attributes describe aspects of the association itself. Without the association, the attributes are nearly meaningless. In the above example, "duration" is an attribute - it describes the amount of time that the board has performed its duties. This paper refers to link attributes as *association properties*.

Implementation

The degree of effort required to implement associations depends on the target language. For example, SQL requires a properly prepared join between tables. In C++, programmers typically implement associations using pointers.

A pointer-based implementation of the above example might look like:

```
class Company {
public:
  string name;
};

class Board {
public:
  Company *pCorporation;
};
```

There is a flaw in this implementation - can you see it? Although a board object can locate its company, the company object cannot locate its board. This is a common oversight. The fact may well be that the existing code base does not need to find a board of directors object given a company object. In many cases, this is just a coincidence and is not a true reflection of the project's needs. Some programmers take the point of view "this will get us by - we can add the other pointer later when we need it." However, adding the pointer later means that the class implementation will change, and depending on the code base, client code may need to be changed. Programmers should not allow existing code to bias class implementations. Large systems should implement associations correctly up front; otherwise, the hasty decision to implement half of the association may result in unplanned effort.

There are at least four more problems with pointers:

1. In a one-to-one association, both objects must point to each other. Pointers do not enforce this constraint. During runtime, this rule is easy to defeat, leading to hidden bugs.
2. Dangling pointers can crop up easily at run time. Pointers can point to deleted objects. Sometimes, pointers should be NULL when they aren't (these bugs are difficult bugs to track down).
3. The forward-and-backward-pointer management problem is even more difficult in one-to-many associations.
4. Associations can have properties. These properties should be shared by both the forward and backward pointers in an association. This requires due diligence on the programmer's part.

Sometimes pointers fit the job perfectly. If you're sure pointers will work for your job, use them. Pointers are by far the easiest to implement. It's not always easy to determine whether pointers will work in the long run. The best way to make a decision is to consider the alternatives. The next few paragraphs will discuss some of them.

Criteria For Robust Association Implementations

A good association implementation should make the programmer's job easy. It should be easy to build and debug. It should work reliably as objects come and go, as their relationships change during program execution.

Specifically, a good implementation should:
- Provide type safety.
- Maintain forward and backward pointers, automatically.
- Support one-to-zero-or-one, one-to-many, and many-to-many relationships.
- Allow one *class* to be associated with many classes. For example, class A can participate in a one-to-one association with class B and participate in a one-to-many association with class C.
- Support *ordered* one-to-many and many-to-many associations (the order of the objects on the "many" side is an important design detail).
- Allow links to have their own attributes.
- Work with object-oriented databases.

Some Solutions

David Papurt's article "Automating Association Implementation in C++" describes a template-based approach that uses inheritance to express associations between objects.[3] In his design, a parent class provides functionality for linking to another object, locating a linked object, and removing a link.

```
template <class parentClass, class linkClass>
class Association :  public parentClass {
  linkClass *pLinkedObject;

public:

  void linkToObject( linkClass * );
  linkClass *getAssociatedObject();
};
```

The class CompanyBase provides all company-oriented functionality, without any association details:

```
class CompanyBase {
  public:
    string name;
};
```

Finally, the Company class mixes CompanyBase with association functionality:

```
class Company : public Association< CompanyBase, BoardOfDirectors > {
};
```

This approach works well for simple object models. However, since inheritance is used, this approach is cumbersome when one class participates in many associations with two or more classes; for example, if a board of directors participates in a one-to-one relationship with a company but also participates in an one-to-many relationship with company executives. The article also does not directly solve one-to-many or many-to-many relationships (he mentions that the approach can be used to implement such associations, but doesn't demonstrate how). Finally, association properties and association objects are not discussed.

Another approach is described in chapter 15 of Object-Oriented Analysis and Design.[4] Private methods, such as add_item() and remove_item(), are defined on the classes that participate in an association. These functions maintain the backward and forward pointers. However, code reuse is not optimal and the chapter glosses over the details.

My Approach

This article presents my approach via OMT object model and dynamic model diagrams. A sample implementation in portable C++ will be presented, along with examples that use it.

There are several points worth mentioning about my approach:

- It satisfies the criteria listed above.
- Code reuse is achieved using templates.
- Programmers can specify the container class for one-to-many and many-to-many relationships. There is no reliance on a particular container implementation.
- In order to avoid name collision, the class names start with AS.
- All of the code in this article is available at http://devguy.com/assoc. The source code is freely distributable.
- Link attributes implemented as follows. Attributes have their own classes (or C++ structures). A link points to an object that holds the attributes. The attribute object can be deleted automatically when the association is broken, or it can be up to the program to delete it.

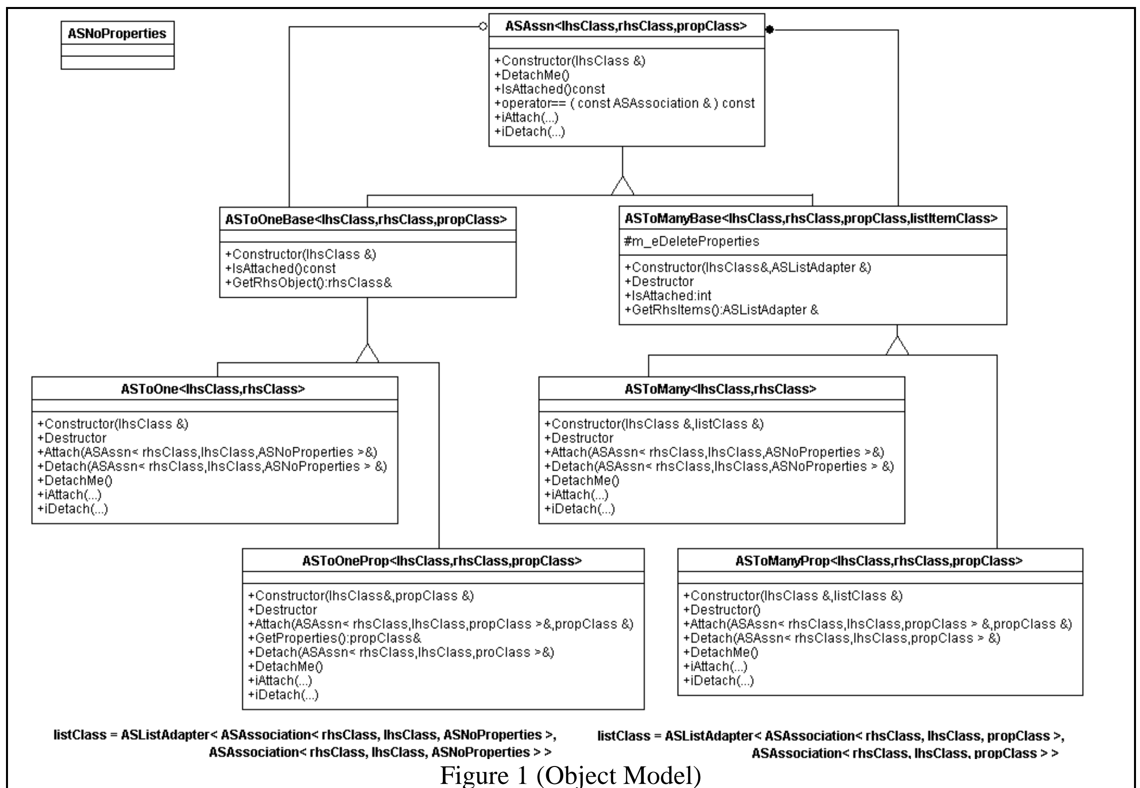
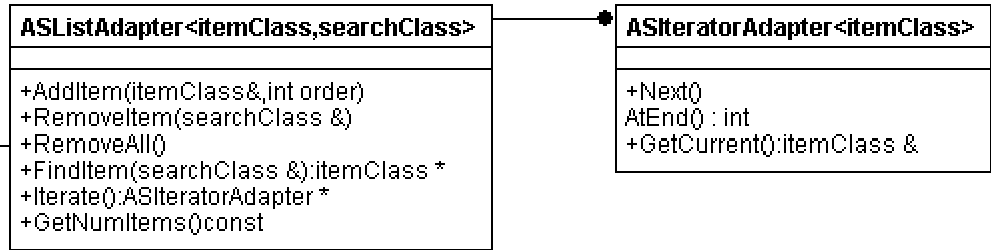


Figure 1 (Object Model)

These are both abstract base classes.

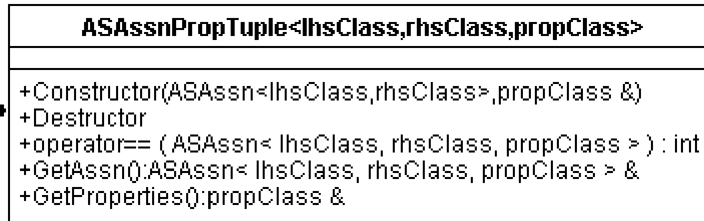


Notes:

1. Modifying the list while an iterator is active invalidates the iterator.

2. The list stores pointers to objects. Objects are not deleted automatically.

3. FindItem() returns NULL if the item could not be found.



ASAssnPropTuple objects are stored in the list when associations have properties.

Figure 2 (Object Model)

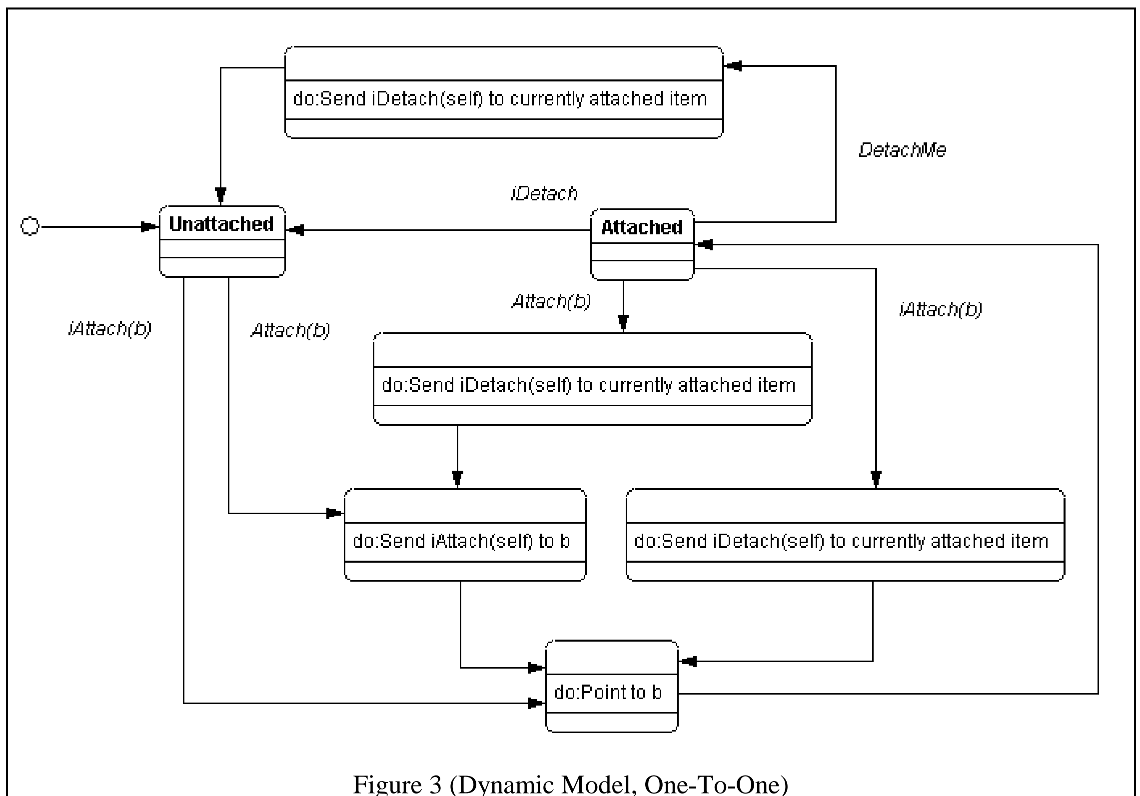


Figure 3 (Dynamic Model, One-To-One)

LHS and RHS

This paper uses the terms "left-hand-side" (LHS) and "right-hand-side" (RHS) to identify the objects in an association. The left-hand-side is the point of reference - there is one and only one object on the left-hand-side of an association. The right-hand-side of the association is what the left-hand-side object "sees" as it "looks out" through the association, as if it were a sailor looking through a periscope. In a one-to-many relationship, the left-hand-side object sees many objects through the periscope. In a one-to-one relationship, the left-hand-side sees only one object.

Pseudo-Link Objects

This article implements *pseudo-link* objects. They are not link objects in the sense of Rumbaugh et. al.,[5] because one object in this design only tells half of the story; two pseudo-link objects are needed to express one link. The rest of this paper will use the term "association object", when it means "*pseudo*-link" object.

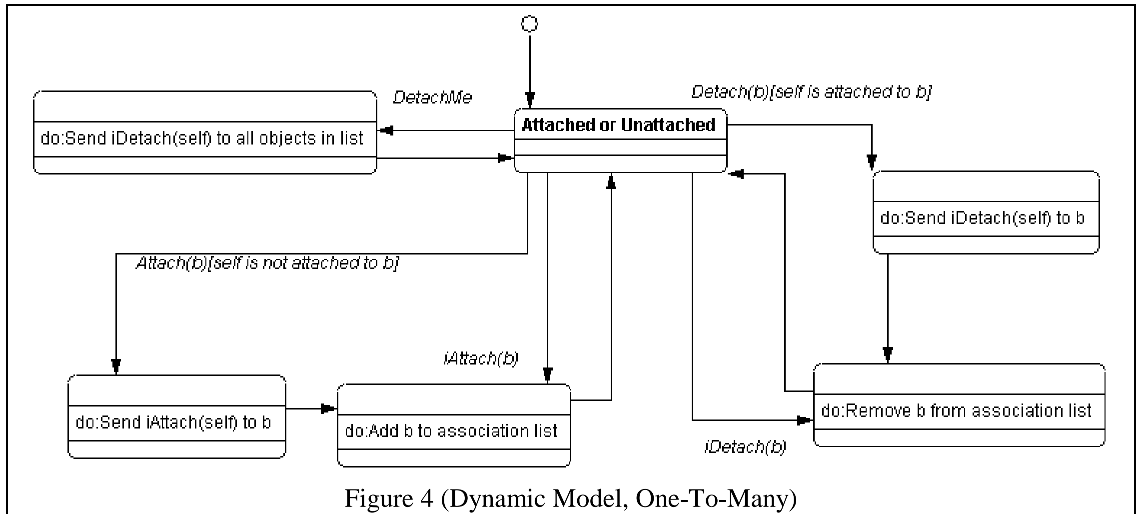


Figure 4 (Dynamic Model, One-To-Many)

Object Model Diagrams

The object model diagrams are shown in figures 1 and 2. ASToOne, ASToOneProp, ASToMany, and ASToManyProp are all classes of association objects. The classes have the following methods:

Method	ASToOne / ASToOneProp	ASToMany / ASToManyProp
Attach(object)	Attaches the LHS object to the passed-in object. If either object is already attached, the previous attachments are revoked.	Attaches the LHS object to the passed-in object.
DetachMe()	Detaches the LHS and RHS objects.	Detaches the LHS object from all of the objects it is attached to
Detach(object)	<<Not present>>	Detaches the LHS object from a specific RHS object.
GetLhsObject()	Returns the left-hand-side object.	Returns the left-hand-side object.
GetRhsObject()	Returns the object that is associated with the LHS object.	<<Not present>>

DetachMe() is called on destruction to minimize "dangling pointers."

ASToOneBase and ASToManyBase[6] classes are indirectly associated. ASToOneBase has a one-to-zero-or-one relationship with ASAssn and ASToManyBase has a one-to-many relationship with ASAssn. This means that:

1. One ASToOneBase object can have a one-to-zero-or-one relationship with an ASToOneBase object - this is how one-to-one relationships are implemented.
2. One ASToOneBase object can have a one-to-zero-or-one relationship with an ASToManyBase object - this is how one-to-many relationships are implemented.
3. One ASToManyBase object can have a one-to-many relationship with ASToOneBase objects - this is how one-to-many relationships are implemented.
4. One ASToManyBase object can have a one-to-many relationship with ASToManyBase objects - this is how many-to-many relationships are implemented.

Dynamic Model Diagrams

The dynamic models are shown in figures 3 and 4. The design consists of two dynamic models, one for one-to-one relationships and another for one-to-many relationships. The dynamic models show the behavior of the association objects. The dynamic models demonstrate how the "forward" and "backward" pointers are maintained.

Figure 3 shows the behavior for the one-to-one case. First, the object (call it A) starts in the "unattached" state. Association objects are linked by sending one of the objects an "Attach" event. Sending the object an "Attach" event results in two things: first, B is sent an "iAttach" event; then, A points to B. This ensures that A points to B and vice-versa. The events are prefixed with an "i" because they are used internally within the dynamic model – these events are not accepted from "outside" objects.

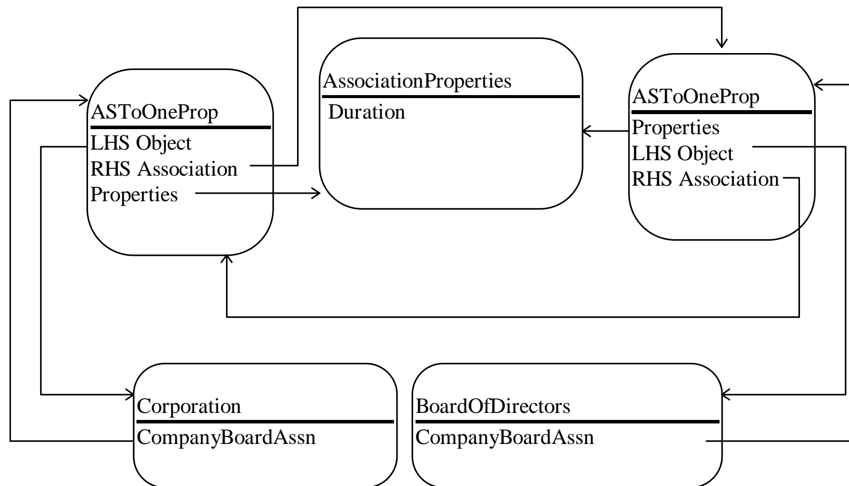
When A receives a "DetachMe" event, B is sent an "iDetach" message. This ensures that A and B are no longer linked.

Finally, consider the case where A is attached to B, and A receives another "Attach" event. Since A is a one-to-one association object, A's LHS object cannot be linked to more than one object. So, A sends B an "iDetach" message.

The one-to-many dynamic model (figure 4) is similar to figure 3. The "Attach" and "Detach" events have the same "message forwarding" behavior. "Attach" sends an iAttach event and "Detach" sends an iDetach event.

<u>One To One</u>

Refer to the company/board of directors object model diagram in the introduction. Both sides of the association are implemented with "ASToOneProp" data members. An instance diagram of the participants follows:



This is an instance diagram of the association implementation. This type of diagram should not be specified in a real design. Lines with arrows represent pointers.

The Corporation class has its own ASToOneProp object that contains:

- A pointer to the corporation, as the LHS (left hand side) pointer
- A pointer to the board of director's association object, as the RHS (right hand side) pointer
- A pointer to the association properties

The BoardOfDirectors class has its own ASToOneProp object that contains:

- A pointer to the board, as the LHS pointer
- A pointer to the corporation's association object, as the RHS pointer
- A pointer to the association properties

The RHS pointers point to Association objects, not to "real" objects. This allows both sides of an association to be updated in one *Attach* or *Detach* operation. A valid question to ask is: *why are two*

association objects needed and not just one? The answer is that association objects are not shared on both sides of the relationship because their classes may differ (as in the case of one-to-many relationships – this is discussed later).

Note that there is only one instance that holds the association properties. This ensures consistency and may conserve memory.

Sample Code

One-to-zero-or-one relationships are fairly easy to implement using the template classes. Two template classes exist for this purpose, ASToOne and ASToOneProp. ASToOne accepts two parameters, which are the classes that are involved in the association. ASToOneProp accepts an additional parameter, which is the class that contains the association properties.

```
class Company;
class BoardOfDirectors;
class CompanyBoardProperties;

// This is the company
class Company {
public:
  Company() : CompanyBoardAssn ( *this ) {}

  // Company Name
  string name;

  // Link Object (to a board of directors object)
  ASToOneProp< Company, BoardOfDirectors, CompanyBoardProperties >
     CompanyBoardAssn;
};

class BoardOfDirectors {
public:
  BoardOfDirectors() : CompanyBoardAssn ( *this ) {}
  // Data member: This is a link object that links to a company object
  ASToOneProp<BoardOfDirectors, Company, CompanyBoardProperties>
      CompanyBoardAssn;
};

// These are the properties
class CompanyBoardProperties {
public:
  CompanyBoardProperties( const long duration )
    : m_duration( duration ) {}

  // How long the board has served
  long m_duration;
};
```

The following code segment links a company to a board of directors:

```
Company company;
BoardOfDirectors board;

// Attach the board and the company - the board has served for 5 months
company.CompanyBoardAssn.Attach( board.CompanyBoardAssn,
  *new CompanyBoardProperties( 5 ) );
```

By default, the properties object is deleted when the link is broken. This can be overridden by constructing `CompanyBoardAssn with` `ASDP_DO_NOT_DELETE_PROPERTIES` (as the last parameter).

A board can locate its company:

```
board.CompanyBoardAssn.GetRhsObject()
```

And a company can locate its board:

```
company.CompanyBoardAssn.GetRhsObject()
```

The properties for the association can be located starting from the company object:

```
company.CompanyBoardAssn.GetProperties().m_duration
```

and also starting from the board of directors object:

```
board.CompanyBoardAssn.GetProperties().m_duration
```

Violating Encapsulation

Notice that the Attach() method (above) accepts an ASAssn object. The code that calls Attach() accesses a data member in "board."[7] Clearly, this violates encapsulation. All other association implementations have this encapsulation problem. Some object-oriented practitioners even argue against associations entirely because of the encapsulation problem.[8] Creating association object "getters" on company and board is a trivial task, but this is not a complete solution. The objects are tightly bound by a member name in either case.

Other proposed solutions only refer to entire objects, as in `Link( objectA, objectB )`,[9] so they don't need to access data members; however, this violates one of our design goals, which is to allow one class to participate in many associations with other classes.

The encapsulation problem is not as detrimental as it seems. First, associations can be implemented without adding data members to a class, because ASAssn objects can be declared anywhere in a program. The following section ("external link objects") describes how.

Secondly, subclassing can be used to isolate classes. First, a stand-alone class is built, free from any associations. Then, a subclass is created that has association objects as data members. Any class that is reusable without a particular association should be implemented in this manner. The following example demonstrates this approach:

```
Class CompanyPure {
public:
  string name;
};

class BoardOfDirectorsPure {
};

class Company : public CompanyPure {
public:
  Company() : CompanyBoardAssn ( *this ) {}

  ASToOneProp< Company, BoardOfDirectors, CompanyBoardProperties >
      CompanyBoardAssn; // You can use any name you want
};

class BoardOfDirectors : public BoardOfDirectorsPure {
public:
  BoardOfDirectors() : CompanyBoardAssn ( *this ) {}

  ASToOneProp<BoardOfDirectors, Company, CompanyBoardProperties>
     CompanyBoardAssn; // You use any name you want
}
```

External Link Objects

Let's address the encapsulation problem in a different way. Instead of adding data members to the Company and BoardOfDirectors classes, use the fact that link objects can be instantiated almost anywhere in a program. Here is an example:

```
class Company {
public:
  string name;
};

class BoardOfDirectors {
};
```

Declare local variables for the company and board objects:

```
Company company;
BoardOfDirectors board;
```

Declare an association object for the company called company_link:

```
ASToOneProp< Company,
             BoardOfDirectors,
             CompanyBoardProperties
           > company_link(company);
```

Declare an association object for the board called board_link:

```
ASToOneProp< BoardOfDirectors,
             Company,
             CompanyBoardProperties
           > board_link(board);
```

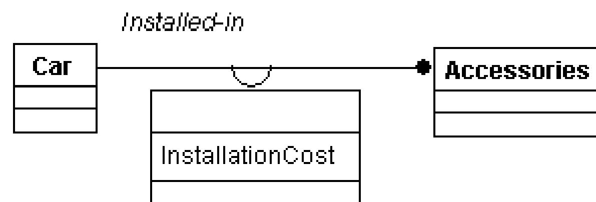
The company and board are linked via the statement:

```
company_link.Attach(board_link, *new Properties(1));
```

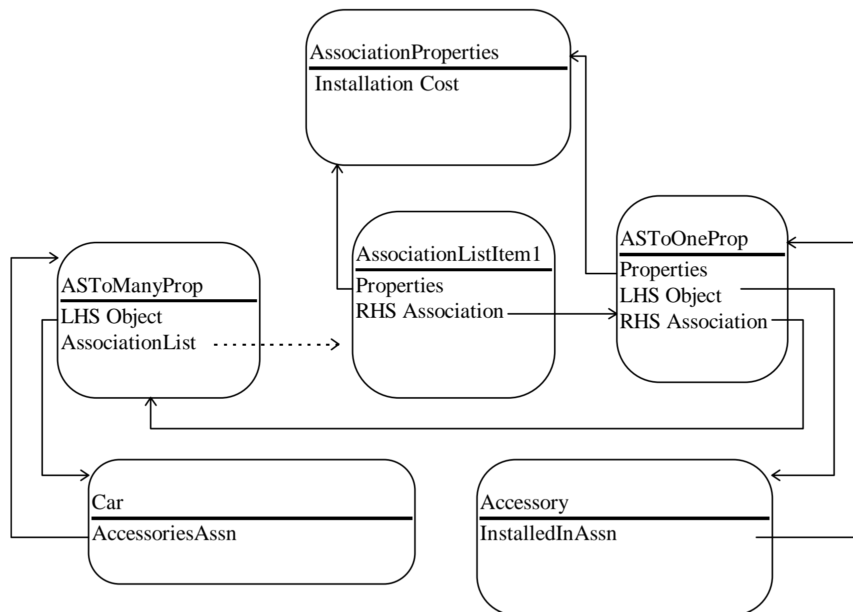
Link objects can be stored in other useful ways, such as in arrays or in a “master class” that holds all of the links in the system. External link objects have some important benefits. First, the classes that participate in an association can be decoupled, possibly leading to greater maintainability and reusability. External link objects also solve the encapsulation problem. However, external link objects come at a price: the application must to keep track of them. Class data members, on the other hand, are easier to locate.

One To Many

Consider a simple one-to-one relationship between a car and its many accessories:



The “car” side of the implementation uses an association of type “ASToManyProp” and the “accessory” side of the implementation uses an object of type “ASToOneProp”:



This is an instance diagram of the association implementation. This type of diagram should not be specified in a real design. Lines with arrrows represent pointers.

The Car object has its own ASToManyProp object that contains:

- A pointer to the car, as the LHS pointer
- A list of objects, each of which contain:
    - A pointer to the accessory's association object
    - A pointer to the properties for the association

The Accessory object has its own ASToOneProp object that contains:

- A pointer to the accessory, as the LHS pointer
- A pointer to the car's association object, as the RHS pointer
- A pointer to the association properties

Sample Code

Implementing one-to-many associations requires more work than implementing one-to-one associations. First, list and iterator classes must be implemented. The list class inherits from ASListAdapter and the iterator class inherits from ASIteratorAdapter.[11]

> Adapter - Adapters define a new interface for an existing class.[10] ASListAdapter and ASIteratorAdapter have pointers to "real" objects that implement the methods defined in the adapter interface. The objects can be STL vectors or any other type of container – you decide.

Lists contain different types of items, depending on whether the association has properties. In the ASToMany case, the list contains ASAssn objects. In the ASToManyProp case, the list contains ASAssnPropTuple objects. Each ASAssnPropTuple (figure 2) object contains a pointer to an ASAssn object and a pointer to an association properties object. The same list and iterator implementations can be to store both types of objects.[12]

The sample code implements two classes, ListTempl and IteratorTempl.[13] The following is an example of the car and accessory classes:

```
class Car;
class Accessory;
class CarAccessoryProperties;

typedef AS_LIST_PROP( ListTempl,
   Car, Accessory, CarAccessoryProperties ) List;
```

```
class Car {
public:
  Car() : m_hasAccessories( *this, *new List ) {}

  // The accessories associated with the car
  ASToManyProp< Car, Accessory, CarAccessoryProperties >
   m_hasAccessories;
};

class Accessory {
public:
  Accessory() : m_installedIn( *this ) {}
  // The car associated with this accessory
  ASToOneProp<Accessory, Car, CarAccessoryProperties> m_installedIn;
};

// The associaton properties
class CarAccessoryProperties {
public:
  CarAccessoryProperties( long cost )
    : m_installationCost( cost ) {}

  long m_installationCost;
};
```

One car is declared along with three accessories:

```
Car car;

Accessory acc1;
Accessory acc2;
Accessory acc3;
```

The car and accessories are then linked together:

```
car.m_hasAccessories.Attach( acc1.m_installedIn,
  *new CarAccessoryProperties( 500 ) );

acc2.m_installedIn.Attach( car.m_hasAccessories,
    *new CarAccessoryProperties( 1000 ) );

acc3.m_installedIn.Attach( car.m_hasAccessories,
  *new CarAccessoryProperties( 7000 ) );
```

Given a car object, the following code steps through all of its accessories:

```
typedef AS_ITER_ADAPTER_PROP( Car, Accessory,
   CarAccessoryProperties ) Iterator;

// Create an iterator so we can go throu all of the accessories
Iterator *pIterator = car.m_hasAccessories.GetRhsObjects().Iterate();

// Print out the code of the accessories
for ( ; !pIterator -> AtEnd(); pIterator -> Next() ) {
  printf( "Cost %d\n",
          pIterator->GetCurrent().GetProperties()
             .m_installationCost );
}
delete pIterator;
```

Iterate() returns a pointer to an object that must be deleted by the caller. The following code demonstrates how access the properties, given an accessory object:

```
acc1.m_installedIn.GetProperties().m_installationCost
```

Code that iterates through a list of associations depends upon whether associations have properties.

For example, without properties, the code is:

```
for ( ; !pIterator -> AtEnd(); pIterator -> Next() ) {
  printf( "Many link:  %d\n", pIterator->GetCurrent().i );
}
```

And with properties:

```
for ( ; !pIterator -> AtEnd(); pIterator -> Next() ) {
  printf( "Many link:  %d\n",
          pIterator->GetCurrent().GetLhsObject().i );
}
```

The difference is that the call to `GetLhsObject` is missing in the first block.

Ordered associations can be implemented in two ways. First, the list itself can be ordered: items can be traversed in the order that they were added to the list; or, a list can be built to sort items based on a key generated from RHS objects. Secondly, traversal order can be specified explicitly when items are attached – the "iOrder" parameter sent to the Attach() method can specify relative order.

Many to Many

One-To-One and One-To-Many relationships have been discussed. Many-to-many relationships can also be implemented. The difference is be that ASToMany (or ASToManyProp) objects appear as data members in both classes.

Evaluation

This approach can help projects avoid bugs that are difficult to detect and resolve. However, there are some drawbacks. C++ templates introduce their own problems. Vendor support varies and cryptic compiler errors make debugging difficult. The syntax may turn off C++ veterans as well as newcomers. The "code bloat" caused by the templates will make the program's binary image larger. Finally, the implementation uses considerably more memory than would a traditional pointer-based implementation, because C++ objects are used to represent links.

Conclusion

Understanding the relationships between objecs is crucial to any software project. Implementing these relationships is not trivial. Programmers should take the time to design a standard approach for implementing associations; otherwise, individual programmers may write their own solutions that will require separate debugging and documentation effort. The approach presented in this article can be modified to suit a particular project's needs.

---

[1] Rumbaugh, James, Blaha, Michael, Premerlani, William, Eddy, Frederick, and Lorensen, William, Object-Oriented Modeling and Design (Englewood Cliffs, New Jersey: Prentice Hall, 1991) 27-38.
[2] Other types of relationships exist, but this paper only covers one-to-zero-or-one and one-to-many.
[3] David Papurt, "Automating Association Implementation in C++," Dr. Dobb's Journal October 1995: 18-25.
[4] Rumbaugh, et al., Object-Oriented Modeling and Design 312-315.
[5] Rumbaugh, et al., Object-Oriented Modeling and Design 33.
[6] The ASToOneBase and ASToManyBase classes exist for memory use optimization. When an association doesn't have properties, one pointer's-worth of memory is saved per ASAssn object.
[7] ASAssn objects can be public data members, because the association is a fundamental aspect of the object model (it's not an implementation detail). If the members are made private or protected, the classes in the association can be friends.
[8] Rumbaugh, et al., Object-Oriented Modeling and Design 31.
[9] Rumbaugh, et al., Object-Oriented Modeling and Design 312-315.
[10] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John, Design Patterns (Menlo Park, California: Addison-Wesley Publishing Company, 1995) 139-150.

[11] A sample list can be found in list.h in the distribution.
[12] See test.cpp in the distribution.
[13] See list.h in the distribution.