Mutual Registration

A Pattern for Ensuring Referential Integrity in Bidirectional Object Relationships

Kevlin Henney

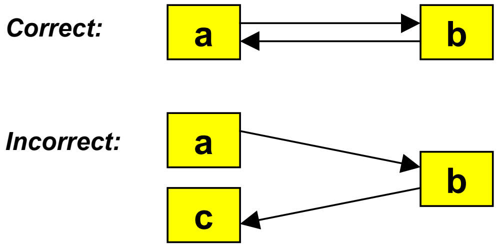
kevlin@acm.org

Context

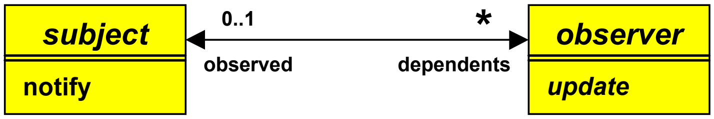
Many object-oriented designs contain bidirectional relationships between objects, where an instance of one class is linked to an instance of another class and vice-versa. These are normally represented in terms of two unidirectional relationships – conventional programming languages provide unidirectional object pointers or references but rarely support any bidirectional constructs.

Problem

When a bidirectional relationship is resolved into two unidirectional relationships, the implicit constraint that the relationship between two objects is a single bidirectional link is lost, i.e. if object *a* points to object *b* then object *b* is guaranteed to point back to object *a*. How can we ensure referential integrity and preserve the relationship's symmetry?



Consider the bidirectional relationship between subjects and observers in many implementations of the OBSERVER pattern for one-to-many event notification [Gamma+1995]. In the following simplified structure, a subject exclusively supplies events to many observers, which can only consume events from one subject at a time:



This leads to the following C++ code (inlined for brevity) for notification and representation:

```
class subject;

class observer
{
public:
    virtual void update() = 0;
    ...
private:
    subject *observed;
};
```

```
class subject
{
public:
    void notify()
    {
        for(iterator current = dependents.begin();
            current != dependents.end();
            ++current)
        {
            (*current)->update();
        }
    }
    ...
private:
    typedef std::set<observer *>::iterator iterator;
    std::set<observer *> dependents;
};
```

An observer is associated explicitly with a subject by calling `attach`, and disassociated by calling `detach`. We can provide these operations on both sides:

```
class observer
{
public:
    void attach(subject *new_observed)
    {
        observed = new_observed;
    }
    void detach()
    {
        observed = 0;
    }
    ...
};
```

```
class subject
{
public:
    void attach(observer *new_dependent)
    {
        if(new_dependent)
        {
            dependents.insert(new_dependent);
        }
    }
    void detach(observer *ex_dependent)
    {
        dependents.erase(ex_dependent);
    }
    ...
};
```

Whilst this makes the implementation of both classes simple, preserving the symmetry of the relationship, the user of the collaboration is left to wire up the relationship correctly:

```
subject  *supplier;
observer *consumer;
...
// make link
supplier->attach(consumer);
consumer->attach(supplier);
...
// break link
```

```
supplier->detach(consumer);
consumer->detach();
```

This approach is easily open to misuse:

```
subject  *supplier[2];
observer *consumer[2];
...
// link not completed for supplier[0] <-> consumer[1]
supplier[0]->attach(consumer[0]);
supplier[0]->attach(consumer[1]);
consumer[0]->attach(supplier[0]);
...
// link not completely broken for supplier[0] <-> consumer[0]
supplier[0]->detach(consumer[0]);
supplier[1]->attach(consumer[0]);
```

It is possible to trap and signal such constraint violations at runtime:

```
class observer
{
public:
    subject *attached() const
    {
        return observed;
    }
    ...
};

class subject
{
public:
    void notify()
    {
        for(iterator current = dependents.begin();
            current != dependents.end();
            ++current)
        {
            if((*current)->attached() != this)
            {
                throw std::logic_error("bad relationship");
            }
            (*current)->update();
        }
    }
    ...
};
```

However, prevention is considered better than cure: the exceptions indicate that the code is incorrect when it is run, but have not prevented incorrect code being written; the horse has already bolted, and is long gone over the horizon by the time the stable door is closed.

Coupling the two classes more tightly can better enforce the object relationship constraints:

```
class observer
{
    friend class subject;
    ... // no attach or detach functions
};

class subject
{
public:
    void attach(observer *new_dependent)
    {
        if(new_dependent && new_dependent->observed != this)
        {
            if(new_dependent->observed)
            {
                new_dependent->observed->detach(new_dependent);
            }
            dependents.insert(new_dependent);
            new_dependent->observed = this;
        }
    }
    void detach(observer *ex_dependent)
    {
        if(ex_dependent && ex_dependent->observed == this)
        {
            dependents.erase(ex_dependent);
            ex_dependent->observed = 0;
        }
    }
    ...
};
```

This correctly abstracts the required control flow, but at the cost of increasing the structural dependency between the two classes. It also introduces an asymmetry that could be addressed by an alternative loosening of encapsulation and shift in dependencies to global functions:

```
class observer
{
    friend void attach(subject *, observer *);
    friend void detach(subject *, observer *);
    ...
};

class subject
{
    friend void attach(subject *, observer *);
    friend void detach(subject *, observer *);
    ...
};
```

Without breaking encapsulation by exposing class implementation or requiring the programmer to follow particular steps when using the classes, it is not immediately obvious how to provide a consistent and symmetric approach for ensuring referential integrity when making and breaking links.

Forces

There is a minimum complexity required of the code and control flow to make and break bidirectional relationships safely and correctly. Where should this complexity lie? For the two classes in the relationship a simpler implementation will mean that the management responsibility lies with the users, increasing the code size and complexity of the usage code. Placing it within either one or both classes in the relationship simplifies usage and prevents incorrect usage, but places a greater burden on the author of the classes.

When one pointer is changed in one object, the pointer in the other must also be changed to maintain the semantics of a bidirectional link. The representation of associations is typically private, and so one object cannot directly modify the link implementation in another object if it needs to be changed.

This can lead to the link representation in each object being managed through attribute-like operations on the interface of each object, i.e. *setters* and *getters*. Thus the relationship is not treated as a whole, forcing the user to perform all of the bookkeeping tasks associated with relationship management. Also the use of attribute-style programming does not translate well to concurrent environments, raising thread-safety issues.

The conflict between encapsulation of representation and encapsulation of control flow leads to unsafe classes and verbose usage: whilst the classes do not preclude correct usage, they do not exclude incorrect usage. Runtime detection can flag incorrect usage, albeit after the fact.

Control flow can be better encapsulated by increasing the dependency of one class on the representation of the other. However, this reduces data encapsulation and breaks the symmetry of the relationship. By reducing data encapsulation further it is possible to reintroduce symmetry with global relationship management functions.

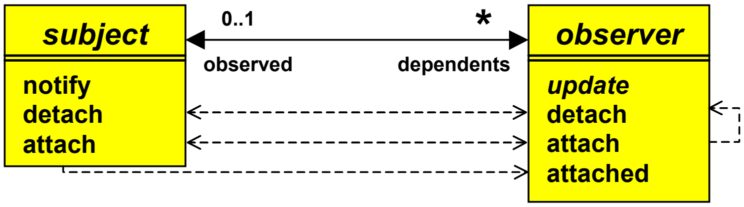
Objects that represent the link itself can be used to factor out the code managing the relationship between two other objects. However, this recursively encounters the same context and problem again: there is a bidirectional between each object and the link object.

Thus, exposing the implementation of the association – for instance, by using C++'s `friend` mechanism – or requiring the programmer to make and break associations following particular steps – clearly error prone – are unsatisfactory approaches.

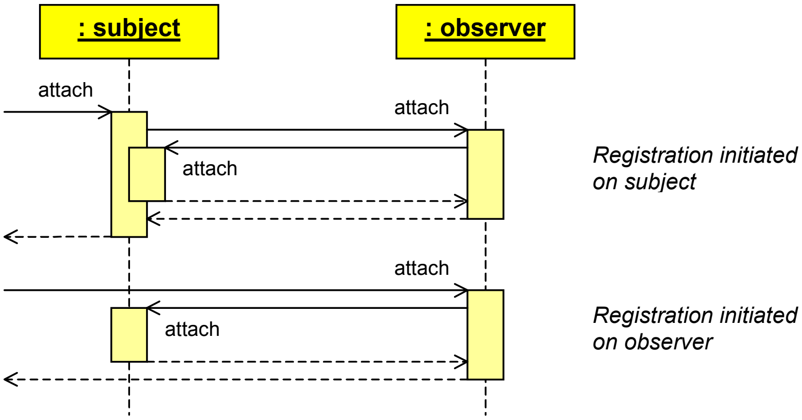
Solution

In the public interface for each class provide operations for removing an existing link and adding a new link, and one of the classes must provide a link query operation. Use mutual calls, possibly resulting in mutual recursion, to implement the registration and deregistration of links.

The operations and call dependencies are illustrated as follows for the motivating example:



Here is a simplified view of the dynamics of attachment of an object of one type to one of the other from each side:


: subject
: observer
attach
attach
attach
Registration initiated on subject
attach
attach
Registration initiated on observer

The following code illustrates the interfaces and control flow in more detail. For brevity, non-throwing container insertions are assumed:

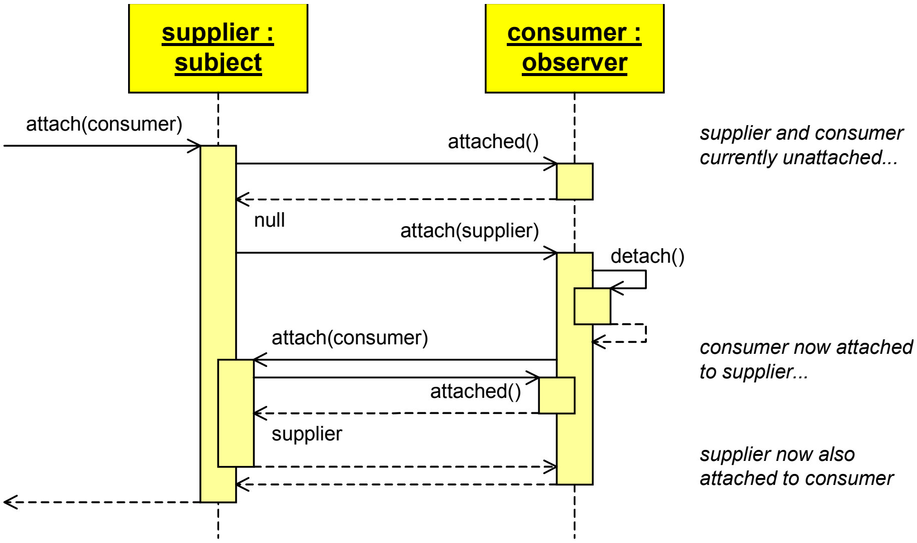
```
class observer
{
public:
    void attach(subject *new_observed)
    {
        if(new_observed != observed)
        {
            detach();
            if(new_observed)
            {
                observed = new_observed;
                observed->attach(this);
            }
        }
    }
    void detach()
    {
        if(observed)
        {
            subject *old_observed = observed;
            observed = 0;
            old_observed->detach(this);
        }
    }
    subject *attached() const
    {
        return observed;
    }
    ...
};

class subject
{
public:
    void attach(observer *new_dependent)
    {
        if(new_dependent)
        {
            if(new_dependent->attached() == this)
            {
                dependents.insert(new_dependent);
            }
            else
            {
                new_dependent->attach(this);
            }
        }
    }
    void detach(observer *ex_dependent)
    {
        if(ex_dependent)
        {
            if(ex_dependent->attached() == this)
            {
                ex_dependent->detach();
            }
            else
            {
                dependents.erase(ex_dependent);
            }
        }
    }
    ...
};
```

To illustrate the usage and the sequence of actions, consider the following scenario: an observer, `consumer`, which is currently not attached to a subject, `supplier`, now registers with that subject. Registration is initiated by a call to the subject:

```
subject  *supplier;
observer *consumer;
...
supplier->attach(consumer);
```

This plays out as follows:



Consequences

A mutually recursive solution maintains the integrity of bidirectional relationships between objects and preserves the symmetry of such relationships, i.e. it can be made or broken at either end, without exposing the representation of the link on either side. The use of MUTUAL REGISTRATION prevents the possibility of invalid states between valid objects – users are simply not given the opportunity to incorrectly establish invalid object relationships – thereby eliminating a particular class of bug.

The bat-and-ball delegation structure leads to slightly more complex code than either a correctly used *getter/setter* approach or an encapsulation compromising structure. Thus the implementation requires more care. However, the complexity is on the inside of the participating classes rather than the outside, and there is a higher degree of independence between the two classes.

Duplication of usage code is reduced compared to a *getter/setter* approach. The nature of the collaboration makes the control flow ideal for refactoring into base classes, into a link object class, or into appropriate template classes or functions.

The overall decision complexity is slightly higher than in other cases, but does not significantly impair performance when compared to the more expensive actions being taken in managing the relationship. The detail of the control flow is simplified considerably when there are particular constraints on the relationship. For instance, when a relationship must be explicitly broken before a new one can be created – such as with monogamous marriage and divorce.

If the registration and deregistration functions can be overridden, there is the possibility that they may not be overridden using the same control flow model, which could lead to errors. In C++, care must also be taken if the registration and deregistration functions are themselves polymorphic or call other functions on themselves that are polymorphic, as this will not have the desired effect if the functions are called from within constructors and destructors of a base class but are overridden in a derived class.

These functions can be made thread safe by ensuring that each registration and deregistration function is implemented as a critical region. However, because of the control flow structure the synchronisation primitives used must be reentrant with respect to the same thread, i.e. reentrant mutexes will work but semaphores will not.

References and Notes

[Gamma+1995] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

As it is discussed in this book, the OBSERVER pattern implementation includes a mention of dangling references from observers to subjects when subjects are deleted. The suggested solution is to notify observers on subject destruction. The sample code also shows the observer detaching itself from the subject on destruction. Observers do not change their subject in their lifetime, i.e. the subject is set and attached to in the observer constructor. However, there is nothing in the sample code or pattern description that deals with referential integrity should the observer be detached from the subject interface.

There are echoes of CHAIN OF RESPONSIBILITY in MUTUAL REGISTRATION. However, the intent of CHAIN OF RESPONSIBILITY is quite different: it distributes responsibility for handling a request into a linked sequence of objects, so that the delegation structure finds the appropriate handler. The focus of MUTUAL REGISTRATION is to ensure referential integrity of bidirectional object relationships symmetrically. With MUTUAL REGISTRATION only specific – rather than general – requests are of interest, i.e. registration and deregistration. CHAIN OF RESPONSIBILITY introduces an object structure, along with an execution structure, to solve a problem, whereas MUTUAL REGISTRATION deals with supporting an existing object structure with an execution model.

We can consider the cyclic relationship between the two parties involved in MUTUAL REGISTRATION to be a closed, fixed length chain with requests for registration and deregistration being passed recursively around it. However, where CHAIN OF RESPONSIBILITY locates a singler handler for a request, the act of registration or deregistration in MUTUAL REGISTRATION requires both parties (or all three, if call context counts differently) to perform significant actions, i.e. MUTUAL REGISTRATION splits a sequence of actions between two parties rather than locating a single site for the sequence. Unlike CHAIN OF RESPONSIBILITY, there is no requirement that the interface of either party conforms to the interface of the other.

[Henney1997] Kevlin Henney, "Beyond the Gang of Four", *Software Design Masterclass: Getting the Best out of Patterns*, Unicom, 1997.

MUTUAL REGISTRATION was previously documented in a shorter form in this talk.

Acknowledgements

I would like to thank Paul Dyson for shepherding this paper, and Charles Weir, Mark Radford and Frank Buschmann for their helpful and detailed comments.