Relationship Manager

-Andy Bulka, August 2001
abulka@netspace.net.au

Abstract

A central mediating class which records all the one-to-one, one-to-many and many-to-many relationships between a group of selected classes.

Classes that use a Relationship Manager to implement their relationship properties and methods have a consistent metaphor and trivial implementation code (one line calls). Traditional techniques of implementing relationships are varied and flexible but often require a reasonable amount of non-trivial code which can be tricky to get working correctly and are almost always a pain to maintain due to the detailed coding and coupling between classes involved.

Context

You have designed your application's business/domain classes (e.g. `Customer, Order, Supplier`) and are faced with the job of implementing all the required relationships. e.g. one-to-one, one-to-many, many-to-many, back-pointers etc. (but not inheritance relationships). You have a large number of classes to implement. The project will evolve over a long period of time and be maintained by many programmers.

Typical implementations of relationship properties and methods (e.g. `Customer.AddOrder(o), Order.GetCustomer()` etc.) involve adding code and private data structures (e.g. pointers, lists/vectors) into each class involved in the relationship. When a relationship changes, all relevant objects that have a record of that relationship must be notified and modified so that their private collections/vector/lists are correctly updated. All this extra code chips away at the cohesiveness of your classes, and couples your classes together in spaghetti fashion.

Some relationships can be tricky to implement. For example a two-way relationship is best implemented by choosing one object to be the *leader* and another as *follower*. The follower object's implementation should only be coded using calls to leader object methods. (**Mutual Friends** pattern, also described as a refactoring in *("Change Unidirectional Association to Bidirectional" Fowler, Refactoring p.197).*

Other relationships may involve a lot of complexity. A relationship between multiple objects with rules and constraints on the relationship may call for a reification and encapsulation of the relationship into a **Relationship Object** *("Basic Relationship Patterns" J. Noble, PLoPD 4).*

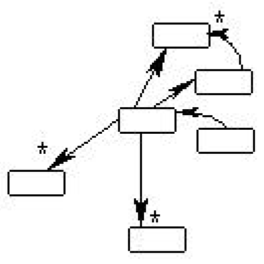

Figure 1. How to implement a rat's nest of relationships?

Problem

How do you robustly implement the required relationships between your classes with as little code as possible - minimising the number of wiring idioms you need to learn?

How do you avoid tightly coupled classes full of code dedicated to maintaining relationships - tedious code which reduces the cohesiveness of your classes?

How do you ensure that your relationship related code is exceptionally easy to read, modify and maintain by existing and especially new programmers joining the project?

Forces

Key forces

- **Necessity:** Relationships e.g. aggregation and composition are primal object oriented concepts that require implementation in all software development projects.
- **Hard Work:** Coding relationship logic into each business object class is often tedious and error-prone work. Mastering a variety of wiring idioms and patterns may be necessary to achieve a robust implementation.
- **Coupling and Maintainability:** Traditional relationship code involves the implementation being spread out amongst many business classes (esp. back–pointers, and deletion notifications etc), which is more difficult to track, maintain and keep in synch than mediated approaches.

Other forces

- **Readabilty:** Detailed relationship related code and coding idioms are harder to read than say, single line function calls.
- **Robustness:** Creating and changing traditional relationship code involves detailed changes to multiple classes, which is error-prone work - robustness is compromised unless strict testing regimes/suites are adopted.
- **Persistence:** It is difficult to persist relationships between classes when relationships are implemented as low level pointers and collections etc. Persisting relationships between business classes is usually needed. Relational database solutions require persistence layers. Object oriented database (OODMS) solutions are complete solutions, but can be expensive to purchase.
- **Scale:** A large number of classes in which to implement relationship code requires more programming hours. Using a central Relationship Manager significantly reduces programming hours. If the number of classes is small, then staying with non-mediated traditional approaches avoids having to learn the Relationship Manager paradigm. Traditional approaches are usually more cpu efficient than a mediated Relationship Manager approach.
- **Tradition:** Coding the relationship related pointers and collections directly into business classes are standard programming practices. Unusual solutions (e.g. like using a mediator like Relationship Manager object to manage relationships) may encounter resistance from experienced programmers. Programmers will need education as to the benefits and consequences of any alternative approach.

Solution

Use a **Mediator** object - a central relationship manager to record all the one-to-one, one-to-many and many-to-many relationships between a group of selected classes. Provide a query method on this mediating Relationship Manager. Implement all relationship code functionality by calling methods on the central Relationship Manager.

Relationship Manager is an implementation choice

The decision to use a Relationship Manager to implement your business object relationships does not affect the design of the interfaces of your business classes. Your classes look and work the same e.g. `c = Customer();  o = Order();  c.AddOrder(o);`

It's only when *implementing* a property or method that entails getting, manipulating or creating a relationship that you *implement* this functionality by calling methods on the central Relationship Manager, rather than attempting to define the attributes and code required for maintaining these relationships within business classes themselves.

The client code doesn't care how methods on the business classes are implemented!

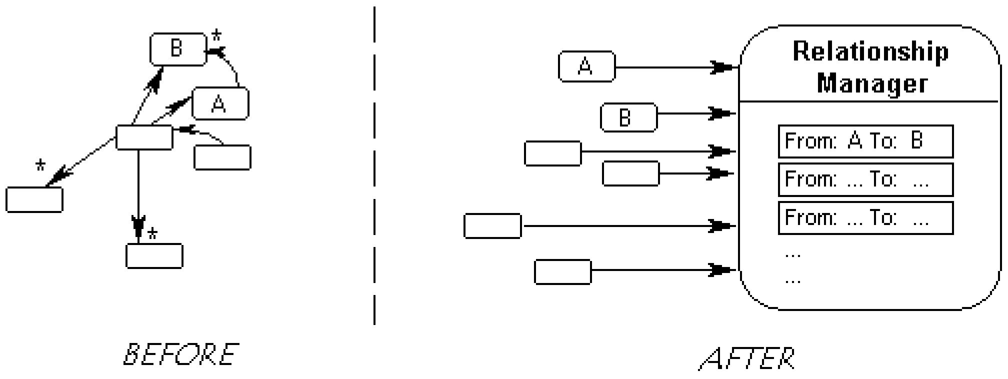

Figure 2. Implementing a rat's nest of relationships versus a simpler approach.

The implementation of the Relationship Manager Class need not be complex – a working implementation taking only 32 lines of code is show below (*see implementation*). Alternatively you can use a third party OODBMS (object oriented database) as a Relationship Manager.

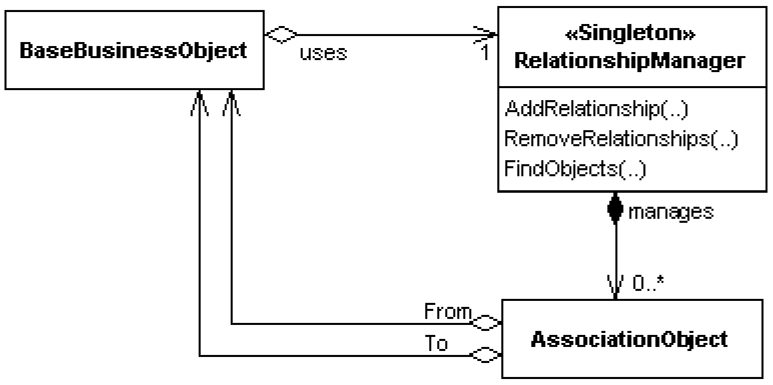

Figure 3. Relationship Manager Pattern

The Relationship Manager provides a generic set of methods to add, remove and query the relationships it manages. Each relationship between objects should be represented by an ***Association Object*** or equivalent data structure. Queries to the Relationship Manager (by the classes involved in the relationships) return lists of objects.

Steps to build:

In a nutshell (bold indicates the participants):

1. Define an **Association Object** class or other equivalent data-structure, with the attributes `from`, `to` and `relationshipId`.
2. Define a **Relationship Manager** class with the methods `AddRelationship`, `RemoveRelationship`, `FindObjects`. Each call to `AddRelationship` results in an association object being created and added to the Relationship Managers's list.
3. *[optional]* Define a constructor in your **BaseBusinessObject** class that sets up a pointer to the Relationship Manager. This way each business object is set up knowing where its Relationship Manager lives.
4. Implement all your **Business Object** relationship logic using calls to the relationship manager.

The use of a Relationship Manager is a private implementation issue for the classes involved. The Relationship Manager is only accessed by the classes involved in the relationships. The interfaces on your business classes remains the same as your UML design. You end up with the following architecture:

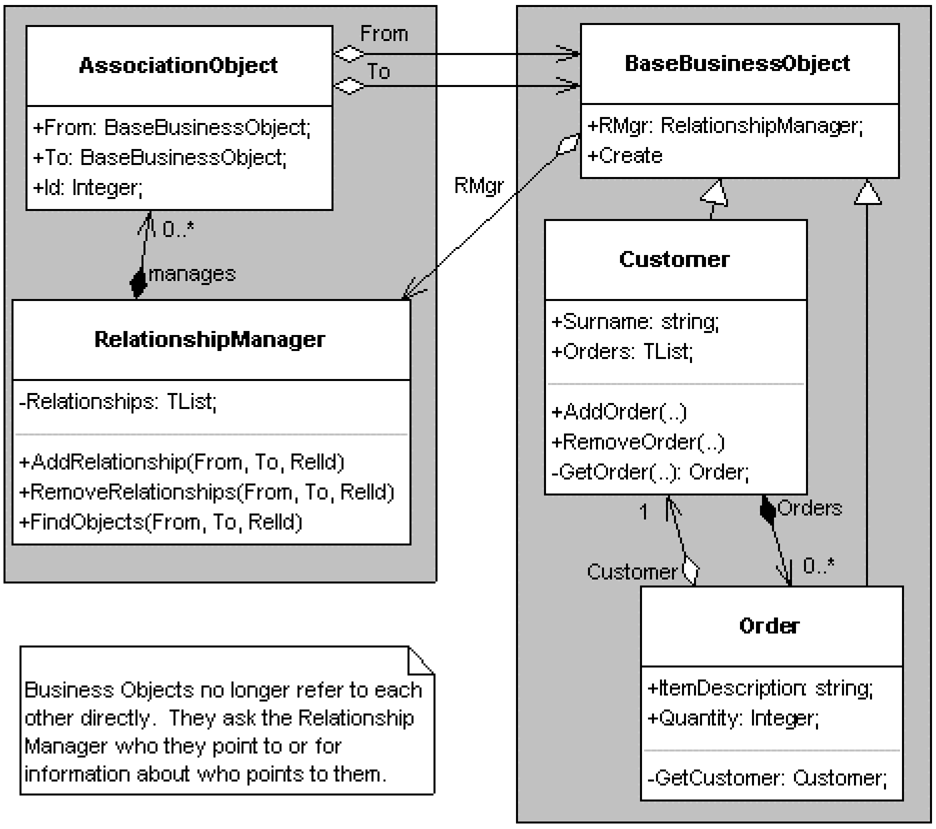
AssociationObject
+From: BaseBusinessObject;
+To: BaseBusinessObject;
+Id: Integer;
From
To
BaseBusinessObject
+RMgr: RelationshipManager;
+Create
RMgr
0..*
manages
RelationshipManager
-Relationships: TList;
+AddRelationship(From, To, RelId)
+RemoveRelationships(From, To, RelId)
+FindObjects(From, To, RelId)
Customer
+Surname: string;
+Orders: TList;
+AddOrder(..)
+RemoveOrder(..)
-GetOrder(..): Order;
1
Orders
Customer
0..*
Order
+ItemDescription: string;
+Quantity: Integer;
-GetCustomer: Customer;
Business Objects no longer refer to each other directly. They ask the Relationship Manager who they point to or for information about who points to them.

Figure 4. An architecture using the Solution.
Business Classes (on the right) are using a Relationship Manager (on the left).

The big benefit of this solution

Your classes should now implement relationships with simple one line calls e.g.

```
class Customer:
    def AddOrder(self, order):
        RelMgr.AddRelationship( From=self, To=order,
                                              RelId=CUSTORD )
```

Consequences and Limitations

Resolution of Forces

Implementing each business class property or method dealing with relationships is now easily done using single-line code calls to the methods of the Relationship Manager. Such code is easy to construct, maintain and read *(see definition of classes* `Customer` *and* `Order`*, below.)*.

There is a greatly reduced need to track and synchronise intricate code changes across multiple classes because classes are no longer as tightly coupled. Classes no longer contain the direct references and detailed code dedicated to maintaining relationships inside themselves. Maintenance is therefore easier. It is also easier and quicker to build a robust implementation.

Tricky mappings like back-pointers in two way relationships can now simply be derived by the Relationship Manager (since it knows every detail about who points to who) rather than needing to be coded explicitly using idioms like Mutual Friend.

A Relationship Manager can easily be enhanced to persist the associations objects that it manages, thus resolving the force of persistence; this force may not be resolved adequately unless the objects involved in the relationship are also persisted, along with their relationships. (*See 'Persistence' in the Discussion section for more discussion on this*).

Coupling is greatly reduced, but replaced with a coupling to a central mediator. Business classes are not tightly coupled to each other any more, but are coupled to their Relationship Manager. The overall coupling effect may seem to be the same (giving up one coupling for another) but it is in fact reduced, since multiple pointers/couplings are traded in for just one coupling to the Relationship Manager. Also, a coupling to the one, same thing is easier to maintain than coupling to a diversity of other classes. Business Classes are more cohesive since they don't have the detailed, messy relationship implementation code in them any more.

The Relationship Manager paradigm is consistent with the insights of relational database technology and OODBMS (object database) technology viz. that we are modelling relationships and the majority of relationships can simply be expressed as simple mappings plus an appropriate query language/method(s). It could be argued that traditional coding approaches not only are re-inventing the wheel, but are ignoring it.

Limitations

The paradigm shift to using a mediated solution may be an obstacle to adoption - programmers will need education as to the benefits and consequences of such a approach. Traditional coding approaches are also usually more cpu efficient than a mediated Relationship Manager approach, which may or may not be an issue depending on the context and the size of the project. (*See performance section in the implementation section for more discussion on this*).

For implementing complex, constrained relationships involving more than a couple of classes, a Relationship Object pattern is still a good choice, since Relationship Manager has a general interface and does not constrain the relationships it manages. (*There is more discussion on the difference between Relationship Manager and Relationship Object in the related patterns section.*)

Implementation

Working Example Implementation of Business Classes

The following code is an example of how to write your business classes in terms of a Relationship Manager. Notice that the implementation of each Class relationship method is a simple *one-line call* to the Relationship Manager. The code is written in the pseudo-code-like, syntactically minimalist but straightforward language Python: (See *Appendix* for easy tips on reading Python)

```
RM = RelationshipManager() # Create the RelationshipManager instance

class BaseBusinessObject:
    def __init__(self):  # Constructor ensures each BO knows it’s RM
        self.RelMgr = RM
```

In the code above, each business object knows which Relationship Manager to use though a pointer in the base class, which is initialised in the base business class constructor. To simplify this implementation, you could instead hard-code a static reference to a global variable (or Singleton) referring to the Relationship Manager where needed. This is a valid solution since you are unlikely to want to dynamically switch relationship managers at run-time.

In the code below, which shows the implementation of example business classes that use a Relationship Manger, the relationship id `101` is an arbitrary value which, in a real project, should be replaced with a named constant e.g. `CUSTORDER = 101.`

```
class Customer(BaseBusinessObject):
    def AddOrder(self, order):
        self.RelMgr.AddRelationship(From=self, To=order, RelId=101)
    def RemoveOrder(self, order):
        self.RelMgr.RemoveRelationships(From=self, To=order,
                                                      RelId=101)
    def Orders(self):
        return self.RelMgr.FindObjects(From=self, RelId=101)


class Order(BaseBusinessObject):
    def GetCustomer(self):
        return self.RelMgr.FindObject(To=self, RelId=101)
```

In the above code, a client call to the customer `Orders()` method will query the Relationship Manager for a list of objects it points to by calling `RelMgr.FindObjects(…)`. The `RelId` restricts the search to just those objects involved in the ‘101’ relationship as opposed to say, the ‘102’ relationship etc.

Working Implementation of a RelationshipManager Class

```
class RelationshipManager:
  def __init__(self):      # Constructor
      self.Relationships = []
  def AddRelationship(self, From, To, RelId=1):
      if not self.FindObjects(From, To, RelId):
        self.Relationships.append( (From, To, RelId) ) # assoc obj
  def RemoveRelationships(self, From, To, RelId=1):
      if not From or not To:
          return
      lzt = self.FindObjects(From, To, RelId)
      if lzt:
          for association in lzt:
              self.Relationships.remove(association)
  def FindObjects(self, From=None, To=None, RelId=1):
      resultlist = []
      match = lambda obj,list,index : obj==list[index] or obj==None
      for association in self.Relationships:
        if match(From,association,0) and match(To,association,1)
                                          and RelId==association[2]:
              if From==None:
                  resultlist.append(association[0])
              elif To==None:
                  resultlist.append(association[1])
              else:
                  resultlist.append(association)
     return resultlist
  def FindObject(self, From=None, To=None, RelId=1):
      lzt = self.FindObjects(From, To, RelId)
      if lzt:
        return lzt[0]
      else:
        return None
```

In the above code, the Relationship Manager uses a list of tuples in the attribute `self.Relationships` to store each relationship. A tuple is just a cheap object – we could have used a proper object with 'from' and 'to' attributes etc. as in the class diagram *fig 3 or 4*. However, since such an association object does not have behaviour, it might as well be implemented cheaply as a minor data structure (e.g. a C++ *struct* or Delphi *record*), in this case a tuple. A tuple is just a list of values e.g. `(23315, 45422, 101)` that can be treated as an object instance, in Python. Each `association` object is a tuple, the 'from' part of the tuple is accessed with the expression `association[0]` and the 'to' part is accessed with `association[1] etc.`

This test code creates a customer and an order and establishes a relationship between them

```
c = Customer()
o = Order()
c.AddOrder(o)

assert(o.getCustomer == c)
assert( o in c.Orders() )
assert( len(c.Orders()) == 1)
```

Implementation Discussion

Queries to Relationship Manager

This particular implementation of a Relationship Manager has a query method `FindObjects()`, which relies on callers leaving one of the 'from' or 'to' parameters empty (`None/Nil/NULL)`. The empty parameter indicates *what is being searched for*. The result is the list of objects found.

For example `FindObjects(objA, None)` will return a list of objects pointed to by objA whilst `FindObjects(None, objA)` will return a list of objects pointing at objA.

Alternative implementations of Relationship Manager may choose to have more explicit interrogatatory methods e.g. `FindObjectsPointedToBy(obj)` and `FindObjectsPointingAt(obj),` or might even use a pure text SQL-like syntax.

Relationship Id

A relationship ID represents the 'name' of the relationship. A relationship ID is an attribute of each association object, used to differentiate between the multiple association objects that would exist between the same two objects when there are multiple relationship pointers between the same two objects. A `RelId` of 101 might stand for the 'owns' relationship and the `RelId` of 102 might stand for the 'parent' relationship between the same two objects. If there is only going to be one type of relationship between two objects then the relationship ID can be omitted. *The Relationship Id is a parameter in all methods of Relationship Manager.*

Relationship Direction

If you tell Relationship Manager that A points to B, then Relationship Manager automatically knows that B is pointed to by A – this is a derived mapping. If you want to simulate the precision of the directionality of the relationships in your UML design (some are one way only, some two way) then you could pass another parameter, being the direction of the relationship. Then if you ask for a backpointer relationship that hasn't been explicitly registered as a two-way relationship, then Relationship Manager will not derive/deduce it, but return `null`. *If required, Relationship Direction would be a parameter in all methods of Relationship Manager.*

Relationship Visibility

There may be a requirement for some relationships not to be visible to everyone using the Relationship Manager. Domain objects would use the Relationship Manager to map and represent both their private and public, and possibly other relationships. Further levels of security (e.g. which objects are allowed to query for what relationship information) could also be implemented and supported by a Relationship Manager.

Association Objects

The programmer is free to implement a Relationship Manager's Association Objects in a variety of ways – it is a private implementation detail. Business Objects (or whatever classes are using the Relationship Manager) do not know about Association Objects – Business Objects only talk to and know about the services of the Relationship Manager.

Type loss using root class references

Having association objects' `from/to` pointers and parameters in the Relationship Manager API restricted to descendants of a base business object class may be too limiting for your needs. Consider implementing the from/to pointers in terms of interfaces, or perhaps as pointers to the root of your implementation language class hierarchy e.g. Java's `Object` or Delphi's `TObject`. These techniques will allow a wider range of classes to be registered with the Relationship Manager.

Type loss using root class or base class references is not a problem peculiar to Relationship Manager. The collection objects (vector/list utility classes) in most languages store references to objects as pointers to the root class (unless you use something like C++ templates). Most languages offer a facility to safely determine and cast back an object's type. In Python, types are associated with objects not variables, which makes things even easier and such castings are not necessary.

Performance

One of the frequently asked questions a programmer will ask about a Relationship Manager solution is – isn't the central Relationship Manager a performance bottleneck? Yes, the extra function calls and level of indirection involved in the use of a Relationship Manager might reduce application performance in some larger or specialised applications. Here are some ideas for optimising Relationship Manager performance in these cases:

The most important decision will be where to store the association objects and how to scan through them quickly. The simple linear list look-up illustrated in the Python example, above, should be enhanced with a more elaborate scheme when performance becomes an issue. Passing say, a relevant class name as an extra parameter to all calls to Relationship Manager might allow Relationship Manager to hash into a list of lists, thereby reducing the search space. *If required, a Performance Parameter would be a parameter in all methods of Relationship Manager.*

You might consider implementing Relationship Manager as an object-oriented database (OODBMS), which has been designed with the relevant technology to optimise queries.

Relationship Managers for multi threaded systems would have synchronisation issues to contend with. Use the multi-threading and synchronisation techniques found in more complex mediator solutions like Reactor and Broker. OODBMS's (object databases) in addition, usually have commit/rollback facilities. *(see Related Patterns section, below).*

You can add caching to the implementation of business objects to avoid having to query the Relationship Manager each and every time a relationship reference is needed. Caching must be carefully done, since you might be relying on a possible out of date situation. But when appropriate notification protocols are designed (e.g. each business object is designed to accept a notification from the Relationship Manager to clear it's caches), such a caching scheme will give performance equal to traditional spaghetti implementations. The downside to caching is that you are introducing complexity again into the business object, however a simple caching scheme, strategically used, will pay off bigtime. The extra memory involved in caching is not that significant, since you would only be storing references to objects, not duplicating the objects themselves.

The Relationship Manager is usually a Singleton, but one could easily have multiple relationship managers as required, which would increase performance. If you had separate sets of domain objects that had no need to refer to each other, then multiple Relationship Managers would be OK, since there would be no requirement (or ability, without resorting to direct pointer references) to refer to objects outside the confines of a particular Relationship Manager.

Deletion Notification Issues

In general, the Relationship Manager should be notified whenever an object is deleted, since it needs to remove all relationships (association objects) involving that object from its mapping database/list. Each business class destructor should notify the Relationship Manager of the deletion, or at least remove any mappings it is involved with. You could add a special method to Relationship Manager e.g `RM.Deleting(anObject)` to handle any such notifications.

An alternative approach to deletion notification in classes that form a composite structure, might involve creating multiple Relationship Managers, each servicing a small number of classes, and being owned by the root class of the composite tree. When the root class is deleted, not only are the owned objects deleted, but the Relationship Manager is also cleanly deleted, without any notifications required. This approach assumes that none of the deleted classes are themselves referred to by other objects or by other Relationship Managers. Also, at this level of granularity, the Relationship Manager is more like a Relationship Object (*see related patterns section*).

Discussion

Paradigms

The Relationship Manager paradigm is consistent with the insights of Relational Database technology and Object Database technology whose focus is on modelling relationships – the majority, if not all of which can be expressed as simple mappings plus an appropriate query language/method(s). It could be argued that traditional coding techniques and idioms (pointers, leader/follower backpointers, collection objects etc.) are not only re-inventing the wheel, but ignoring it.

Of course it may not be appropriate to implement all relationships between classes in the mediated way - the forces in a particular context may still encourage traditional solutions. However the rise in popularity of object oriented databases (which have similarities to the relationship manager paradigm – *see related patterns section*) is evidence that mediated, relational database-like solutions are becoming increasingly important.

Many programmers design object oriented models which then map, via a persistence layer to traditional relational database tables. This approach usually does not leverage the power of the relational database technology – using databases as a mere storage medium, and still coding the relationships between classes in memory the traditional way (using pointers and lists of references etc.). Some other approaches *do* leverage relational database technology to perform queries, however the resulting recordsets require transformation into lists of objects that the application running in memory can actually use. In contrast, queries to a Relationship Manager always return an object or a list of objects - which is ideal.

Persistence

A Relationship Manager can easily be enhanced to persist the association objects it manages. Relationship Manager's association data is flat (just a list of association objects) no matter what the semantics of the relationship between business objects involved are, which makes implementing the persistence of the association objects straightforward and non recursive.

By also owning the *objects* involved in the relationships it manages, the Relationship Manager becomes a self-contained system, where all objects and their relationships are known and accounted for. This eases the task of persistence, since persisting relationships separately from objects may create reference resolution difficulties. Further, if the Relationship

Manager is implemented as a Component or 'Bean' capable of composition and recursive streaming (e.g. a JavaBean or Delphi TComponent) then your whole business object model and its relationships can be persisted easily with a Serialization style method call e.g. `RMgr.saveToFilestream('c:\mymodel.out')`. For this to work, the Relationship Manager, the Business Objects and the Association Objects must all be components (e.g. descendants of `TComponent`) and owned by the Relationship Manager component, the way a GUI form owns the component widgets living on it. OODBMS (object database) style relationship managers provide both relationship management and object persistence, though usually not of a serialized nature as just described.

Advanced Discussion

If the use of mediators is appealing to you then consider that in order to represent both the basic relationships *between* Business Objects (inward/yin?) and the observer relationships between Business Objects and *other application objects e.g. GUI widgets*, (outward/yang?) you need both a Relationship Manager and a Change Manager. Together, this pair might be said to comprise two 'centres' of an application architecture.

Interestingly, when you implement them in detail, *relationships* and *observer relationships* are similar in quite a few ways. They are both mappings between objects. Also, basic relationships often require an aspect of 'notification' in them e.g. a deletion notification message is sent to a business object when something it points to has been deleted – this corresponds exactly to the way an observer object gets notified of an event occurring in the subject. Finally, the many observers of a single subject is of course a many-to-one relationship. One might therefore speculate that aspects of Relationship Manager and Change Manager are echoed and contained in the other – as the Chinese say, there is a little yin in yang and vice versa.

Related Patterns

Relationship Manager is an instance of the **Mediator pattern -** it promotes loose coupling by keeping objects from referring to each other explicitly. A mediator encapsulates all interconnections, acting as the hub of communication. Relationship Manager is a closely related but distinct pattern because it specifically deals with those forces relating to programmers implementing relationship methods in business objects.

Change Manager *(GOF, p. 299)* is also an instance of the Mediator pattern. It deals specifically with the implementation of Observer or Publisher/Subscriber relationships, whilst the Relationship Manager deals with the encapsulation of general relationships (one-to-one, one-to-many etc.).

Other more complex mediator patterns like **Broker** *(POSA p. 121)* and **Reactor** *(PLOPD1 p.544)* encapsulate more complex behaviour and relationships and include sophisticated event notifications systems.

A **Relational Database** is also a mediator of relationships, however suffers from an impedance mismatch with objects living in memory (recordsets returned by queries need to be transformed into lists of objects) and thus relational databases are not readily suited to implementing the day to day relationships between business objects – though they could certainly be used if the appropriate 'impedance translation code' was written (either by hand or automatically e.g. by an object to relational mapper tool). Note also that Relationship Manager performance issues (*see implementation performance section*) are similar to those relational database designers face when optimising their query engines.

An **Object-oriented database management system** (ODBMS) treats objects as first class citizens, storing objects not tables. ODBMS queries return lists of objects - not table recordsets. Relationship Manager is like an ODBMS, in that they both store relationships between objects. However Relationship Manager doesn't require that you store the objects themselves, like an ODBMS does. ODBMS systems also offer a range of transaction control and robustness functionality that pure Relationship Manager does not. You could certainly implement Relationship Manager using an OODBMS, and Relationship Manager could certainly benefit from the speed and robustness features of an OODBMS.

The **Relationship Object** and the **Collection Object** *(Noble PLOPD4, "Basic Relationship Patterns", p 80)* use an instance of each type for *each* relationship or for each one-to-many relationship respectively. The solution of Relationship Manager goes further and models *all* (or many) relationships in one central Relationship Manager. Whilst both patterns intend to help ease implementation of relationships between classes there are several differences in intent, forces and implementation:

Relationship Manager	**Relationship Object**
Intent: Intent is to provide a global dictionary of relationship mappings between all or many classes	Intent is to represent a single, possibly complex and constrained relationship involving two or more classes
Interface: Interface to a Relationship Manager is generic – simply a pair of add/remove relationship methods and a single query method which returns a list of objects e.g. `FindObjects(from, to, relId) : TList`	Interface for a Relationship Object is flexible, and depends on the situation and the classes involved. Method names might be customised to suit the context. Note that Collection Objects (which are Relationship Objects) typically *will* have similar interfaces e.g. Java `vector`, Delphi `TList` all have similar methods relating to add/remove/find/indexing/pack/sort etc.
Scope of implementation: No complexity encapsulated by a Relationship Manager. No 'relationship constraints' are maintained or enforced. It just stores mapping pairs and responds to generic queries	Encapsulates complexity – one of the intents of Relationship Object is to encapsulate the rules and constraints of a possibly complex relationship amongst two or more classes
Cardinality: One Relationship Manager for all relationships. Usually a global singleton offering generic mapping services. Note: Multiple RMs can be used to service different varieties of classes, if there are efficiency or conceptual separation concerns.	One Relationship Object per relationship: A Relationship Object needs to be created for each relationship (that you want to represent in this way). Each Relationship Object is usually owned by the class it is serving. Alternatively the interface of the Relationship Object may be merged with an existing business class interface.

Note: Relationship Object and Relationship Manager can complement each other, working together when complex relationship constraints need to be implemented. The Relationship

Manager can be used for storing all the relationships, and the Relationship Object used to encapsulate any particularly complex and constrained relationships. The Relationship Object would use the 'low level' services of the Relationship Manager, typically calling Relationship Manager as many times as it needs to in order to work out and manage constraints on the relationship it is managing.

It is also worth noting that the implementation of Relationship Object can often be similar to Relationship Manager, except there is no need for a *relationship id* to distinguish between relationships, since each Relationship Object is dedicated to looking after a just one relationship.

Known Uses

'Web Nuggets' is an application that uses both a Relationship Manager and a Change Manager to model complex web searches and display results in a GUI.

A Relationship Manager is used in the modelling of Australia's national electricity grid. The grid is composed of elements like regions, stations, generators, transformers and lines which are all related in various ways to each other.

Global dictionaries are often used to store relationships between objects.

Any Relational Database is a kind of Relationship Manager – it is a central mediator / API for storing and retreiving information about relationships. An OODBMS (object oriented databases) is both a relationship manager and an object manager, with persistence and transactional safety features (rollback/commit etc).

Bibliography

Gamma, Helm, Johnson, Vlissides (GOF), Design Patterns – "Singleton", p. 127

Gamma, Helm, Johnson, Vlissides (GOF), Design Patterns – "Change Manager", p. 299

James Noble, "Basic Relationship Patterns", Pattern Languages of Program Design #4 (PLOPD4), p 73.

Buschmann et. al. "Broker" Pattern Oriented Software Architecture (POSA) p. 121

Schmidt 1995, "Reactor", Pattern Languages of Program Design (PLOPD1) p.544

Fowler "Change Unidirectional Association to Bidirectional" Refactoring p.197

APPENDIX

Easy Tips on Reading Python Code

Python is a simple, straightforward and elegant language. It uses standard conventions of accessing methods and properties and is fully OO. Types are associated with objects not variables, so you don't need to declare variables. Functions are called like `afunction(param1, param2)` and objects are created from classes the same way e.g. `o = MyClass()`. Python is case sensitive.

There are no `begin end` reserved words or `{ }` symbols in Python to indicate code blocks – this is done through indentation. The colon in `if lzt:` simply means 'then'. The idiom of testing objects (rather than expressions) in an if statement makes sense as python treats empty lists, None and 0 as false.

Python understands named parameters e.g. In a method call, `afunction(From=None)` means you are passing None *(null / nil)* as the 'From' parameter, whilst in a method definition `From=None` means that if the caller does not supply this parameter, then it will default to None.

The first argument of all methods defined inside classes must be 'self'. This argument fills the role of the reserved word *this* in C++ or Java or *self* in Delphi. Most languages supply this parameter (a reference to the current instance) implicitly whereas Python makes this concept explicit. At runtime this parameter is supplied by the system, not the caller of the method, thus the `def AddOrder(self, order)` method in reality takes *one* parameter when calling it: `AddOrder( order ).`

The statement `pass` means *do nothing*.

You can return multiple items at once *e.g.* `return (2, 50)` and also assign multiple items at once *e.g.* `x, y, z = 0` or even expressions like `result, status, errmsg = myfunction(1, 90).`

Other class files/modules are imported using `import somefile.` `__init__` methods are simply constructors. Finally, a *lambda* is just a one line function that reads `functionname = lambda paramlist : returnedexpression.`

Both **Python** and JPython (Java Python, now called **Jython**) are open source, free and available from www.python.org

-Andy Bulka
abulka@netspace.net.au
6/32 Loller street,
Brighton VIC 3186, Australia
Ph: +613-9593-1389