

# Relationship Manager in Boo

Relationship Manager is a design pattern I wrote a few years ago that I have used in many projects. I have ported it to Boo. See [http://www.atug.com/andypatterns/rm\\_boodotnet.htm](http://www.atug.com/andypatterns/rm_boodotnet.htm) for a web page dedicated to this port and the original [The Relationship Manager Design Pattern](#) paper.

Andy Bulka

<http://www.atug.com/andypatterns>

## Abstract

Relationship manager is 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. It provides a query method. The bottom line: you should **implement all relationship code functionality by calling methods on the central Relationship Manager**. So if you have, for example, a **Person** class and an **Order** class - instead of implementing methods on the Person class e.g. AddOrder(o) and GetOrders() in terms of ArrayLists and whatever, you instead implement those methods by making simple single line calls to a central mediating relationship manager class. Saves a lot of work, especially when you get back pointers on the other side of the relationships that you have to maintain etc.

## What is Relationship Manager good for?

- > I'd appreciate an 'how to use' example
- > to get a quick / better idea on what
- > it's useful for when programming

The basic idea with relationship manager is that it is like a dictionary, which maps relationships between two things, be they object references or strings or whatever. In my examples I usually use strings, though object references are commonly used too, in order to map relationships between object instances.

The benefit over a dictionary is that you can have multiple mappings e.g.

```
a -> 1
a -> 2
a -> 3
```

then you can ask what 'a' points to and get the result

```
[1, 2, 3]
```

you can also ask what is pointing to 3, and get the result

**a**

One common use of this technology is to do all your wiring between objects using a relationship manager, thereby saving yourself having to implement your own one to many lists and having to maintain fiddly backpointer logic etc. So you still have e.g. an AddOrder(o) method on your e.g. 'Person' class...its just that you implement the method using a one line call to the relationship manager - simple! e.g.

```
class Person:
    def AddOrder(o):
        RM.addRelationship(self, o,
'personToOrderRelationship')
    def GetOrders():
        return
RM.findObjectsPointedToByMe(self, 'personToOr
derRelationship')
```

There is a bit more of a tutorial code sample in the post <http://tinyurl.com/9xz5m>

## **Implementation**

The port from python was slow but sure - certainly not an instant conversion, but many things converted without change, especially the core code of my many many unit tests. Some tips I gathered during the conversion can be found at <http://www.atug.com/andypatterns/booDevelTips.htm>.

```
namespace RelationshipManager55
import System

class RM1:
    """
Relationship manager revisited.
Version 1.2
Boo .NET port
```

Sep 2005.

(c) Andy Bulka

<http://www.atug.com/andypatterns>

```

    _____
   |  _ \  ___| |  _  _| |_(_)  ___  _  _  ___|
   |__ ( )_  ___
   | |_) /  _ \ |/_ `  |  ___| |/_  _ \ | ' _ \/_  ___|
   ' _ \ | | ' _ \
   |  _ <  ___/ | ( _ | | | _ | | ( _ ) | | | | \_  _ \ |
   | | | |_) |
   | _ | \_ \ ___| _ | \_ , _ | \_ | _ | \_ ___/ | _ | | _ ___/ _ |
   | _ | _ | . ___/
       | _ |

```

```

    _____
   |  \/_ |  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _  _
   | | \/_ | |/_ `  | ' _ \ / _ `  |/_ `  |/_ `  |/_ `  |/_ `  |/_ `  | | | |
   | | | | ( _ | | | | | ( _ | | ( _ | | ___/ |
   | _ | | _ | \_ , _ | _ | | _ | \_ , _ | \_ , _ | \_ ___| _ |
       | ___/

```

"""

```
private rm as RelationshipManager
```

```
private enforcer as Hash
```

```
def constructor():
```

```
    self.rm = RelationshipManager()
```

```
    self.enforcer = {}
```

```
def ER(relId, cardinality):
```

```
    self.ER(relId, cardinality, "directional")
```

```

def ER(relId, cardinality, directionality):
    # enforceRelationship(id, cardinality,
directionality)
    self.enforcer[relId] = (cardinality,
directionality)

    private def ValidateEnums(cardinality,
directionality):
        assert directionality in ("directional",
"bidirectional")
        assert cardinality in ("onetoone",
"onetomany")

    private def ExtinguishOldFrom(toObj,
relId):
        oldFrom = self.B(toObj, relId)
        self.NR(oldFrom, toObj, relId)
        private def ExtinguishOldTo(fromObj,
relId):
            oldTo = self.P(fromObj, relId)
            self.NR(fromObj, oldTo, relId)

    private def
_RemoveExistingRelationships(fromObj, toObj,
relId):
        if relId in self.enforcer.Keys:
            cardinality, directionality =
self.enforcer[relId]
            self.ValidateEnums(cardinality,
directionality)

```

```

    if cardinality == "onetoone":
        ExtinguishOldFrom(toObj, relId)
        ExtinguishOldTo(fromObj, relId)
    elif cardinality == "onetomany": # and
directionality == "directional":
        ExtinguishOldFrom(toObj, relId)

def R(fromObj, toObj, relId):
    # addRelationship(f, t, id)
    self._RemoveExistingRelationships(fromObj,
toObj, relId)
    self.rm.AddRelationship(fromObj, toObj,
relId)

    if relId in self.enforcer.Keys:
        cardinality, directionality =
self.enforcer[relId]
        self.ValidateEnums(cardinality,
directionality)
        if directionality == "bidirectional":
            self.rm.AddRelationship(toObj, fromObj,
relId)

def P(fromObj, relId):
    # findObjectPointedToByMe(fromMe, id,
cast)
    return self.rm.FindObject(fromObj, null,
relId)

def B(toObj, relId):
    # findObjectPointingToMe(toMe, id cast)

```

```
    return self.rm.FindObject(null, toObj,
relId)

    def PS(fromObj, relId):
        # findObjectsPointedToByMe(fromMe, id,
cast)
        return self.rm.FindObjects(fromObj, null,
relId)

    def BS(toObj, relId):
        # findObjectsPointingToMe(toMe, id, cast)
        return self.rm.FindObjects(null, toObj,
relId)

    def NR(fromObj, toObj, relId):
        # removeRelationship(f, t, id)
        self.rm.RemoveRelationships(fromObj,
toObj, relId)

        if relId in self.enforcer.Keys:
            cardinality, directionality =
self.enforcer[relId]
            self.ValidateEnums(cardinality,
directionality)
```

```

if directionality == "bidirectional":
    self.rm.RemoveRelationships(toObj,
fromObj, relId)

```

## API and documentation

### Relationship Manager API

Returns	Function Name	Short-hand
void	addRelationship(f, t, id)	R(f,t)
void	removeRelationship(f, t, id)	NR(f,t)
IList	findObjectsPointedToByMe(f, id)	PS(f)
IList	findObjectsPointingToMe(t, id)	BS(t)
void	EnforceRelationship(id, cardinality, bidirectionality)	ER(id, c, bi)
Object	findObjectPointedToByMe(fromMe, id, cast)	P(f)
Object	findObjectPointingToMe(toMe, id cast)	B(t)
void	removeAllRelationshipsInvolving(o, id)	NRS(o)

Enforcing relationships e.g.

```
ER("xtoy", "onetoone", "directional")
```

registers the relationship as being one to many and directional, so that e.g. when you add a second relationship between the same two objects the first relationship is automatically removed - ensuring the relationship is always one to one. Alternatively, you could raise an exception.

The findObject methods only find one object (even though many more may be there), and cast it to the appropriate type. This is a commonly used convenience method.

## Supporting implementation code

The private guts of the relationship manager now follows. Note you don't need to use the methods of this class, in

fact you don't even need to know about this class - just use the above API instead. (Though you can use this class directly if you wish to, its a bit more low level but quite usable):

```
def GetRelations():
def SetRelations(listofrelationship tuples):
def AddRelationship(From, To):
def AddRelationship(From, To, RelId):
def RemoveRelationships(From, To, RelId):
def FindRelIdsBetween(From, To) as IList:
def DoesRelIdExistBetween(From, To) as bool:
def DoesRelIdExistBetween(From, To, RelId) as bool:
def FindObjects(From, To) as IList:
def FindObjects(From, To, RelId) as IList:
def Clear():
def FindObject(From, To):
def FindObject(From, To, RelId) as object:
```

```
namespace RelationshipManager55
```

```
import System
```

```
//import copy
```

```
import System.Collections # for IList
```

```
class EfficientRelationshipManager1:
```

```
#region Bigcomment
```

```
/*
```

```
e.g.
```

```
relations {
```

```
  from1 : {to1:[rel1]}
```

```
  from2 : {to5:[rel1,rel2], to6:[rel1]}
```

```
}
```

```
inverseRelations {
```

```
  same as above except meaning is reversed.
```

```
}
```

```
Real Sample:
```

```
-----
```

```

relations = {
    "from1" : {"to1":["rel1"]},
    "from2" : {"to5":["rel1","rel2"],
"to6":["rel1"]}
    }
print relations
print relations.Keys

*/
#endregion

public Relations = {}
public InverseOfRelations = {}

def constructor():
    pass

def GetRelations():
    result = []
    //fromobj as duck      // maybe need later?
    //relId as string     // maybe need later?
    todict as Hash       // new
    for fromobj in self.Relations.Keys:
        todict = self.Relations[fromobj]
        for toobj in todict.Keys:
            //relationsList as IList = todict[toobj]
            //for relId in relationsList:
            for relId in todict[toobj]:
                result.Add((fromobj, toobj, relId))
    return result

```

```

def SetRelations(listofrelationship tuples):
    for r as IList in
listofrelationship tuples:
        self.AddRelationship(r[0], r[1], r[2])

//Relationships = property(GetRelations,
SetRelations) # ANDY
Relationships as IList:
    get:
        return self.GetRelations()
    set:
        self.SetRelations(value)

private def AddEntry(relationshipsDict as Hash,
From, To, RelId):
    if From not in relationshipsDict:
        relationshipsDict[From] = {}
    if To not in relationshipsDict[From] as Hash:
        //relationshipsDict[From][To] = []
        entry as Hash = relationshipsDict[From]
        entry[To] = []
    entry1 as Hash = relationshipsDict[From]
    entry2 as IList = entry1[To]
    if RelId not in entry2:
        //if RelId not in relationshipsDict[From][To]:
        //relationshipsDict[From][To].append(RelId)
        entry2.Add(RelId)

def AddRelationship(From, To):
    self.AddRelationship(From, To, 1)

```

```
def AddRelationship(From, To, RelId):
    AddEntry(self.Relations, From, To, RelId)
    AddEntry(self.InverseOfRelations, To,
From, RelId)
```

```
private def _ZapRelationId(rdict as Hash,
From, To, RelId):
    assert _HavespecifiedallParams(From, To,
RelId)
    relList as IList = (rdict[From] as
Hash)[To]
    if RelId in relList:
        relList.Remove(RelId)
    if relList == []: # no more
relationships, so remove the entire mapping
//del rdict[From][To]
(rdict[From] as Hash).Remove(To)
```

```
private def ZapRelId(From, To, RelId):
    _ZapRelationId(self.Relations, From,
To, RelId)
    _ZapRelationId(self.InverseOfRelations,
To, From, RelId)
```

```
private def _HavespecifiedallParams(From,
To, RelId):
    return (From != null and To != null and
RelId != null)
```

```

private def
_NumberOfNonWildcardParamsSupplied(From, To,
RelId):
    numberOfNoneParams = 0
    if From == null:
        numberOfNoneParams+=1
    if To == null:
        numberOfNoneParams+=1
    if RelId == null:
        numberOfNoneParams+=1
    return numberOfNoneParams

def RemoveRelationships(From, To, RelId):
    /*
    Specifying None as a parameter means 'any'

                                i.e. match with anything
    */
    lzt as IList

    if
_NumberOfNonWildcardParamsSupplied(From, To,
RelId) > 1:
        raise "RuntimeError " + "Only one
parameter can be left as None"

    if _HavespecifiedallParams(From, To,
RelId):
        if self.DoesRelIdExistBetween(From, To,
RelId): # returns T/F
            ZapRelId(From, To, RelId)

```

```

else:
    if From==null:
        lzt = self.FindObjects(null, To, RelId)
# result is list of From objects
        for obj in lzt:
            ZapRelId(obj, To, RelId)
    elif To==null:
        lzt = self.FindObjects(From, null,
RelId) # result is list of To objects
        for obj in lzt:
            ZapRelId(From, obj, RelId)
    elif RelId==null:
        lzt = self.FindRelIdsBetween(From, To) #
result is RelIds
        for relid in lzt:
            ZapRelId(From, To, relid)
    else:
        raise "impossible else"

private def _FindRels(subdict as Hash,
RelId) as Boo.Lang.List:
    resultlist = []
    for k in subdict.Keys:
        v as IList = subdict[k]
        if ((RelId in v) or (RelId == null)):
            resultlist.Add(k)
    return resultlist

public def FindRelIdsBetween(From, To) as
IList:

```

```
# Find all RelId's between blah and blah
# Used to be FindObjects() where From=blah
To=blah RelId=None
subdict as Hash
subdict = self.Relations[From] or {}
relationIdsList as Boo.Lang.List =
subdict[To] or []
return relationIdsList[:] # return the
entire list of relationship ids between
these two.
```

```
public def DoesRelIdExistBetween(From, To)
as bool:
```

```
    self.DoesRelIdExistBetween(From, To, 1)
```

```
public def DoesRelIdExistBetween(From, To,
RelId) as bool:
```

```
    # T/F does this specific relationship
exist
```

```
    # Used to be FindObjects() where From=blah
To=blah RelId=blah
```

```
    subdict as Hash
```

```
    subdict = self.Relations[From] or {}
```

```
    relationIdsList as IList= subdict[To] or
[]
```

```
    return RelId in relationIdsList # return
T/F
```

```
def FindObjects(From, To) as IList:
```

```
    return self.FindObjects(From, To, 1)
```

```

def FindObjects(From, To, RelId) as IList:
    /*
    Specifying None as a parameter means 'any'
    Can specify
        # 'From' is None - use normal relations
dictionary
        From=None To=blah RelId=blah anyone
pointing to 'To' of specific RelId
        From=None To=blah RelId=None anyone
pointing to 'To'

        # 'To' is None - use inverse relations
dictionary
        From=blah To=None RelId=blah anyone
'From' points to, of specific RelId
        From=blah To=None RelId=None anyone
'From' points to
    */
    if From!=null and To!=null:
        if RelId == null:
            raise "Deprecated in boo version. Use
FindRelIdsBetween(From, To) instead"
        else:
            raise "Deprecated in boo version. Use
DoesRelIdExistBetween(From, To, RelId)
instead"

    if From==null and To==null:
        raise "RuntimeError " + "Either 'From' or
'To' has to be specified"

```

```

    //havespecifiedallParams = lambda :
(From<>None and To<>None and RelId<>None)
    //resultlist as IList
    resultlist = []
    subdict as Hash

    if From==null:
        //subdict =
self.InverseOfRelations.get(To, {})
        subdict = self.InverseOfRelations[To] or
{}

        //resultlist = [ k for k, v in
subdict.iteritems() if (RelId in v or RelId
== None)]
        resultlist = self._FindRels(subdict,
RelId)

    elif To==null:
        # returns a list of all the matching tos
        //subdict = self.Relations.get(From, {})
        subdict = self.Relations[From] or {}

        //resultlist = [ k for k, v in
subdict.iteritems() if (RelId in v or RelId
== None)]
        resultlist = self._FindRels(subdict,
RelId)

    else:

```

```

    raise "RuntimeError!! " + "Either 'From'
or 'To' has to be specified"
    /*
    # Both 'To' & 'From' specified, use any
e.g. use normal relations dictionary
    From=blah To=blah RelId=None    all RelId's
between blah and blah
    From=blah To=blah RelId=blah    T/F does
this specific relationship exist

    //subdict = self.Relations.get(From, {})
    subdict = self.Relations[From] or {}
    //relationIdsList = subdict.get(To, [])
    relationIdsList as IList= subdict[To] or
[]
    if RelId==null:
        resultlist = relationIdsList # return
the entire list of relationship ids between
these two.
    else:
        return RelId in relationIdsList #
return T/F
    */

    //return copy.copy(resultlist)
    return resultlist[:]

def Clear():
    self.Relations.Clear()
    self.InverseOfRelations.Clear()

```

```
def FindObject(From, To):  
    return self.FindObject(From, To, 1)  
  
def FindObject(From, To, RelId) as object:  
    lzt as IList  
    lzt = self.FindObjects(From, To, RelId)  
    if lzt.Count > 0:  
        return lzt[0]  
    else:  
        return null
```

```
# -----
```

```
class
```

```
RelationshipManager(EfficientRelationshipMan  
ager1):  
    pass
```

Andy Bulka <http://www.atug.com/andypatterns>