Relationship Manager

Graeme Geldenhuys

2009-07-10

In this article we are going to look at the problem surrounding object oriented programming and object relationships. We will look at the traditional way of coding those relationships and their potential problems. In this article I will show you an alternative solution on how we can solve the various types of relationships like one-to-one, one-to-many and many-to-many, all using design patterns. In the end we will end up with a new design pattern called the Relationship Manager. 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.

So what is the problem?

When you are given the task to implement the business objects of some project, you will definitely face the problem of coding up the relationships between classes. These relationships can vary in complexity – from simple ones like one-to-many, to more complex relationships like many-to-many.

To try and explain the relationship complexities a bit easier, I will be using a simple example of a music catalogue program. In this example you have three entities to deal with: the Album, the Artist and the Tracks you will find on the album. By only using these three simple entities we can already see a few different relationships than need implementing.

In Figure 1 on page 2 we can see a UML diagram showing a one-to-one relationship between the Album class and the Artist class. Objects can refer to each other directly via object references. To save these objects to a database, it is vital to save the object references as well. We can not however save these object references directly in a database, because they are specific to the running instance of the application.

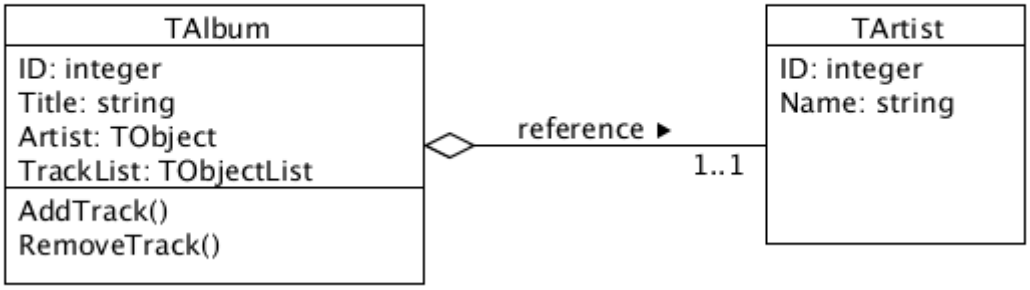

Figure 1: Example class diagram of a one-to-one relationship.

The solution is to use a *foreign key* in the database table. So to save the relationship between these two classes, we need to store the ID field of the Artist instance in the same record in the database table of the related Album. So as you can see from Figure 2 on page 2 our association goes from the *Albums* table to the *Artists* table.

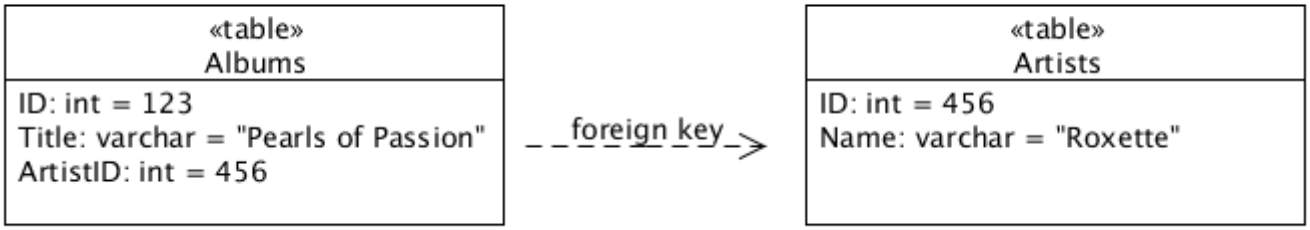

Figure 2: Database layout with foreign key for one-to-one relationship.

In our music catalogue program we also find a more complicated relation. A one-to-many relationship between the Album and the Tracks. In Figure 3 on page 3 we can see the UML diagram depicting the object relationship between the Album and the Track objects. You many notice that the UML diagram looks very similar to the one from Figure 1 on page 2, but the object implementation is quite different. In this case we can't simply use a object reference, because we need to track multiple `TTrack` object instances. So in the `TAlbum` class we need to implement some or other collection to store the Track references. Here the developer can choose various methods of implementing a collection. In terms of Object Pascal, the developer can use a Dynamic Array, TObjectList, TList etc.

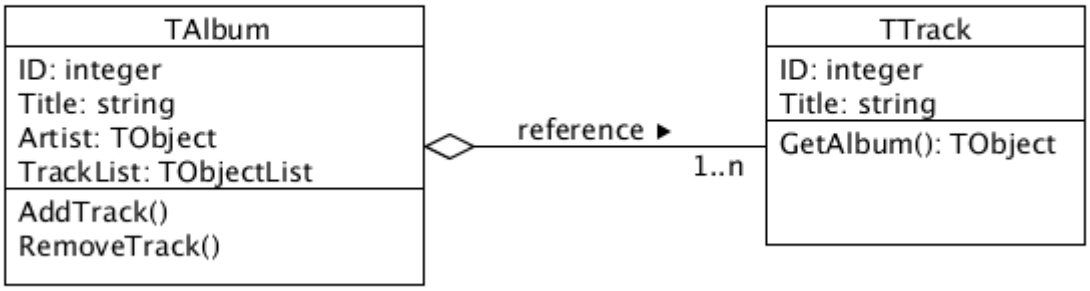

Figure 3: Example class diagram of a one-to-many relationship.

As before, the developer now faces the problem of saving this collection to the database. We will again use a foreign key field to solve the problem, but in this case we need to reverse the direction of the association in the database tables. Our association will go from the *Tracks* table to the *Albums* table.

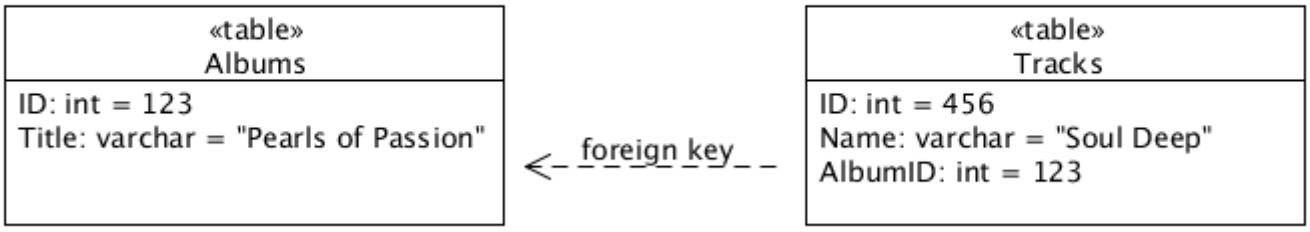

Figure 4: Mapping a collection using a reverse foreign key.

There are many other obstacles at play here. The developer has now solve the problem of saving the collection to the database. But what happens if the user of the music catalogue application amends the relationship after it has already been saved in the database? For example, the user deletes a Track from the collection and assigns it to a collection of different Album. Or the title of one of the Tracks have changed. Possible solutions might be:

- Delete all the Tracks related to a Album in the database, and then save the collection from the application instance back to the database.
- Implement some form of diff comparison against each Track from the application instance to what is currently stored in the database.
- Implement some form of back pointers

The second option is probably the easiest to implement for the developer, but it is definitely not the most efficient.

As you can see, what started as a fairly simple music catalogue program suddenly became a lot more complicated due to the various relationship types, and how we need to save and restore those relationships to a database. Not to mention that each developer might implement the class collections in a different manner - using a TObjectList or Array etc.

The developer might use some object persistence framework like tiOPF[1] to standardise on the collection implementation and help with the persisting of the data, but it's not always sunshine and roses. In the case of tiOPF, if the developer used the *auto-mapping* feature where the tiOPF generates the SQL statements automatically, the developer might run into some problems, because the auto-mapping in tiOPF doesn't support many-to-many relationships very well. So they will have to opt for using tiOPF's *hard-coded visitors* to solve the problem.

Such inter-dependencies between classes and a tight coupling to the collection implementations also makes in much harder to unit test your classes.

So what can we do?

As you might have guessed, if you were reading my previous articles, there is always some design pattern(s) we can apply to help solve our programming obstacles. This is where the *Relationship Manager* design pattern steps in to save the day. The Relationship Manager is simply a central *mediating class*[2] that handles all one-to-one, one-to-many and many-to-many relationships between classes.

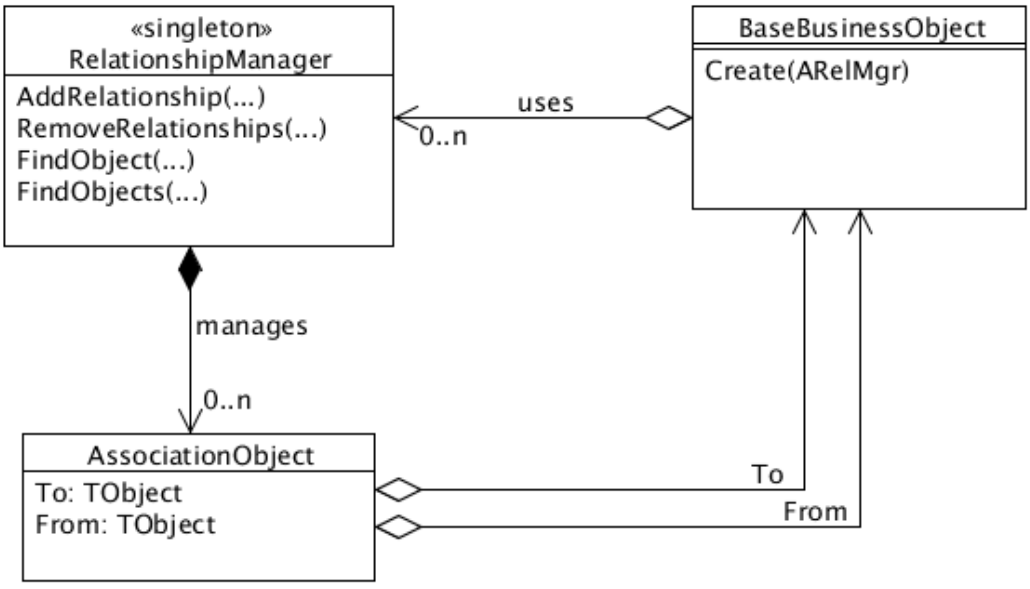

Figure 5: UML diagram of the Relationship Manager.

The traditional approach of creating a class relationship is to write non-trivial code which often causes unnecessary coupling between classes. These hard coded dependencies also make unit testing of those classes overly complicated. The Relationship Manager allows you to setup relationships with a single line of code and keeps all involved classes loosely coupled.

It is important to note that by using the Relationship Manager, it does not mean your class interface has to change. You can still continue using method calls like `Album.AddTrack(ATrack)` or `LAlbum := Track.GetAlbum`. The Relationship Manager simply makes the implementation of such method calls much simpler, and normally reduces them to only one-liner implementations.

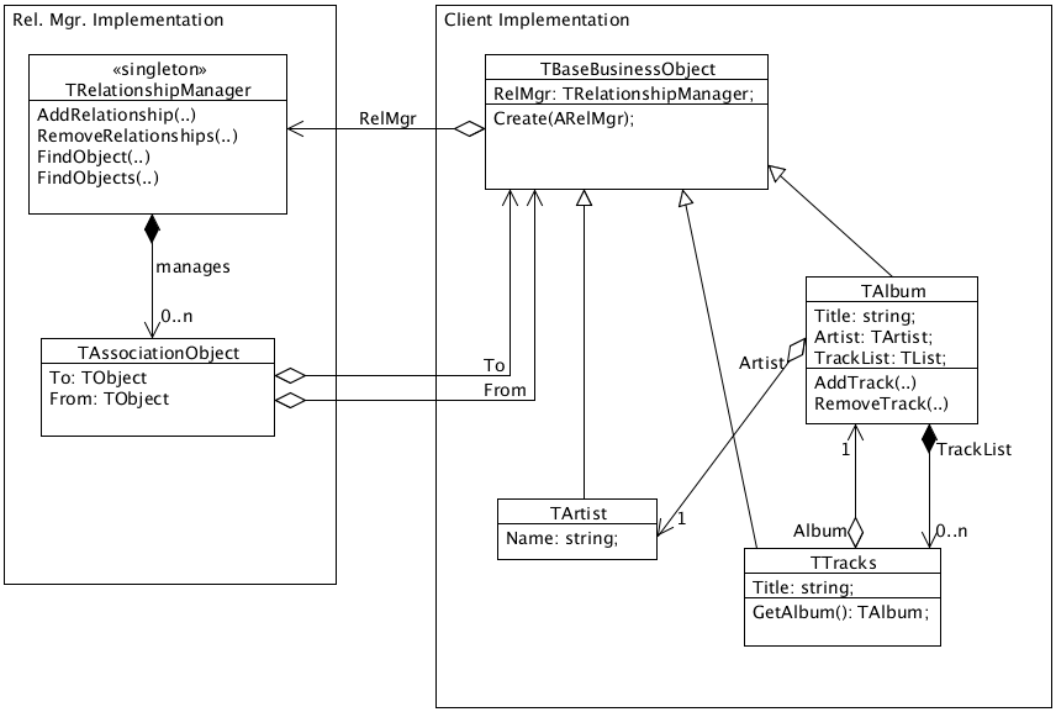

Figure 6: Our Music Catalogue program shown as a UML diagram.

The implementation

The Relationship Manager is actually a very easy pattern to implement. I have included two implementations of the Relationship Manager for this article. The one shown here in this text is a simplified version – yet it works perfectly for in-memory relationships that do not need to be stored to a database. I will leave that for you to implement as an exercise. The second implementation is a more advanced and complete one. It uses the tiOPF framework to help out with the persistence of the relationships to a database. The second implementation also relies on the base classes, `TtiObject` and `TtiObjectL-`

ist, to help simplify the implementation and add some extra behaviour as a bonus.

Due to space constraints, I will only be showing the simplified implementation in this text, but both implementations and their unit tests will be available on the accompanied Toolbox magazine DVD.

Enough talk about theory – lets see some code!

```
TRelationshipManager = class(TObject)
private
  // Internal list to store association objects
  FList: TObjectList;
protected
  // only for unit testing purposes
  property List: TObjectList Read FList;
public
  constructor Create;
  destructor Destroy; override;
  procedure AddRelationship(pFrom, pTo: TObject;
      pRelId: string);
  procedure RemoveRelationships(pFrom, pTo: TObject;
      pRelId: string);
  function FindObjects(pFrom: TObject = nil;
      pTo: TObject = nil;
      pRelId: string = ''): TObjectList;
  function FindObject(pFrom: TObject = nil;
      pTo: TObject = nil;
      pRelId: string = ''): TObject;
end;
```

Above you can see the interface declaration of the `TRelationshipManager` class. The Relationship Manager would normally be accessed via a *Singleton*[3], but this is not required. Sometimes it is even desirable to have a few instances of the Relationship Manager – handling various groups of relationships per instance.

As you can see from the class declaration above, it has an internal list that stores the relationships. Each relationship is represented by a data only class `TAssociationObject`, of which I will talk about in a minute.

Then we have two procedures called `AddRelationship()` and `RemoveRelationships()`, which does exactly as their names describe. Respectively they add new relationships or remove relationships from the Relationship Manager.

Then we also have two methods, `FindObject()` and `FindObjects()`, which the client code can use to query the Relationship Manager.

For all these methods the `pFrom` and `pTo` parameters work in a similar fashion. The parameters describe the direction of the relationship, or what list of objects it needs to find and return. In the latter case where you are searching for objects, you would normally leave one of the parameters empty. The empty (nil) parameter indicates to the Relationship Manager what it must search for.

The `pRelID` parameters is a string value, normally a constant definition, which defines a specific relationship. Think of it as the *name* of the relationship. Each type of relationship needs to have its own unique name. This relationship ID will be stored in the

`TAssociationObject` I mentioned earlier. This is how the Relationship Manager can distinguish between various relationships, and ignore all the irrelevant relationships when it needs to do a search. As an example, the value "album_tracks" might define the relationship regarding what Track objects belong to what Album. If you preferred, you can just as easily change the type of the relationship ID to *Integer* instead of *String*. There is no rule that it must be implemented as a *String* type.

Below is the implementation section of our Relationship Manager showing the class, `TAssociationObject`. As you can see, it is a pure data object – no data manipulation happens in it. We could very easily have implemented this as a record structure as well. This class is defined in the implementation section, so it is hidden from any other client code. The reason for this is simply because it should not be used by any other client code. It forms part of the internal workings of the Relationship Manager.

```
Type
  { This object store the relationship between two classes }
  TAssociationObject = class(TObject)
  private
    FFromObj: TObject;
    FRelID: string;
    FToObj: TObject;
  public
    constructor Create(const pFrom: TObject;
        const pTo: TObject; pRelID: string); reintroduce;
    property FromObj: TObject read FFromObj write FFromObj;
    property ToObj: TObject read FToObj write FToObj;
    property RelID: string read FRelID write FRelID;
  end;

{ TAssociationObject }

constructor TAssociationObject.Create(const pFrom: TObject;
    const pTo: TObject; pRelID: string);
begin
  inherited Create;
  { only the object references gets stored }
  FFromObj := pFrom;
  FToObj   := pTo;
  FRelID   := pRelId;
end;
```

And finally we get to the `TRelationshipManager` implementation! As you read through the code, you will notice that the implementation is quite easy to follow – one of the easier design patterns to implement. I have placed comments in the code to help explain certain areas.

```
{ TRelationshipManager }

constructor TRelationshipManager.Create;
begin
  inherited Create;
  FList := TObjectList.Create;
end;
```

```
destructor TRelationshipManager.Destroy;
begin
  FList.Free;
  inherited Destroy;
end;

procedure TRelationshipManager.AddRelationship(pFrom,
    pTo: TObject; pRelID: string);
var
  lst: TObjectList;
begin
  lst := FindObjects(pFrom, pTo, pRelID);
  if not Assigned(lst) then
    FList.Add(TAssociationObject.Create(pFrom, pTo, pRelID))
  else
    lst.Free;
end;

{ Based on which parameter (pFrom or pTo) has been
  assigned, we can determine how to find objects. }
function TRelationshipManager.FindObjects(pFrom: TObject;
    pTo: TObject; pRelID: string): TObjectList;
var
  i: integer;
  lAssc: TAssociationObject;
begin
  if (pFrom = nil) and (pTo = nil) then
    raise EEmptyParameters.Create('You can''t have both ' +
      'parameters nil');
  Result := TObjectList.Create;
  // loop through all association objects
  for i := 0 to FList.Count - 1 do
  begin
    lAssc := TassociationObject(FList.Items[i]);
    // Are we working with the correct relationship?
    if (lAssc.RelID = pRelId) then
    begin
      // will return a list of objects pointing at pTo
      if not Assigned(pFrom) then
      begin
        if (lAssc.ToObj = pTo) then
          Result.Add(lAssc.FromObj);
      end
      // and this will do the reverse lookup
      else if not Assigned(pTo) then
      begin
        if (lAssc.FromObj = pFrom) then
          Result.Add(lAssc.ToObj);
      end
      else
      begin
        // used when adding/removing relationships
        if (lAssc.FromObj = pFrom) and
           (lAssc.ToObj = pTo) then
          Result.Add(lAssc);
      end;
    end;
  end;  { for }
  { Nothing was matched so free the empty list }
  if Result.Count = 0 then
  begin
```

```
    Result.Free;
    Result := nil;
  end;
end;

function TRelationshipManager.FindObject(pFrom: TObject;
    pTo: TObject; pRelId: string): TObject;
var
  lst: TObjectList;
begin
  Result := nil;
  lst    := FindObjects(pFrom, pTo, pRelId);
  if Assigned(lst) then
    // Because lst is a TObjectList which manages its items
    // memory, we have to extract it before it gets freed
    // with the list.
    Result := lst.Extract(lst.First);
  lst.Free;
end;

procedure TRelationshipManager.RemoveRelationships(pFrom,
    pTo: TObject; pRelID: string);
var
  lst: TObjectList;
  i: integer;
begin
  Assert(pFrom <> nil);
  Assert(pTo <> nil);
  lst := FindObjects(pFrom, pTo, pRelID);
  if Assigned(lst) then
  begin
    for i := lst.Count - 1 to 0 do
      Flist.Extract(TAssociationObject(lst.Items[i]));
    TAssociationObject(lst.Items[i]).Free;
    lst.Free;
  end;
end;
```

Further Discussion and Improvement Ideas

Here are a few ideas that can be implemented in the future. Some comments also relate to the slightly more advanced implementation available on the Toolbox DVD. Also remember that the more advanced version is used with the tiOPF Framework, but can just as easily be applied to any other type of persistence framework.

- As I mentioned earlier, the Relationship ID is currently defined as a String type. There is no set rule that it must be a string type. It can very easily be changed to a Integer, GUID or whatever data type you preferred. The more advanced implementation uses an Integer type.
- There may come a time where you would like to restrict what users are allowed to see – what relationships relate to them. Many applications include some form of user security levels in their software – limiting the user to what they are allowed to see. For this reason you might want to incorporate that security into the

Relationship Manager as well. Limiting what relationships a specific user may query – based on their security level.

- Creating new business object classes at the Database and Object Persistence Framework visitor level is very flexible. You just need to take care of the non-object properties, the Relationship Manager will do the rest when it comes to (any kind of) object relations.
- Traversing complex object structures is really easy, no need to take care of back-pointers to parents etc. One-to-one, one-to-many and many-to-many relationships are supported in the same way.

We can't always have everything working perfectly – what fun would that be? There are some possible problems – or rather things that could be improved on. The implementations I made available to you are not 100% perfect in design. There are a few things that could be improved on, but remember this is only related to the implementations on the DVD. Your own implementations might not have these issues at all.

- When the RM is used for multiple complex relation types, it can quickly become the main bottleneck in an application due to the great number of association objects. This could probably be improved on by making giving the relationship manager slightly more information, which it could then use to create some form of a hash list. The other alternative is to use multiple instances of the relationship manager for various parts of your application.
- The Relationship Manager might require the lists related to the association objects to be read into memory before the Relationship Manager gets populated. This might mean those lists must be globally accessible. The `TListStrategy` used in the advanced Relationship Manager is an attempt to mitigate this.
- Sometimes temporary variables must be used to avoid two calls to the Relationship Manager – which would mean you are actually doing two object lookups. For example, the following code will trigger two calls to the Relationship Manager – and two searches through the relationship list:

```
if Assigned(Order.Customer) then
  Display(Order.Customer.Name);
```

To avoid the double call we must use a temporary variable:

```
tmpCustomer := Order.Customer;
if Assigned(tmpCustomer) then
  Display(tmpCustomer.Name);
```

Conclusion

The relationship manager pattern might not be for everybody. The most probable reason might be that it is too different to what you are used to. I have found the Relationship Manager very handy in certain projects – especially ones with complex relationships. So there is definitely a use for the Relationship Manager.

I also mentioned earlier about unit tests. I have included all the test suites in the accompanied code as console based applications. There are test suites for both implementations (easy and advanced). I would also like to take this time to mention that I followed the programming technique called *Test-Driven Development*, while I implemented the Relationship Manager. I highly recommend you give it a try. You can read more about it in this book: "*Test-Driven Development by Example*"[4] or Google for more information.

References on the following page...

[1] Toolbox 2'2007, Seite 26: Das Persistenzframework tiOPF by Michael Van Canneyt.
tiOPF homepage is freely available at http://www.tiopf.com
[2] Eric Gamma, Richard Helm, Ralph Johnson & John Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley 1994. ISBN 0-201-63361-2, Seite 273.
[3] Toolbox 6'1998, Seite 58: Das Singleton-Pattern in C++ by Marian Heddesheimer.
Toolbox 5'2008: Patterns in Pascal: bessere Programmpflege mit Simple Factories
[4] Kent Beck: Test-Driven Development: by Example. Addison-Wesley 2003. ISBN 0-321-14653-0