# Chapter 5 – Relationships

### Overview

The next logical step when using domain modelling is to model relationships between the Entities in our model.

#### A Code example

Consider how this might be useful when writing code: let’s say that an Invoice can have zero or more Invoice lines, and we wish to calculate the total value of the invoice by totalling the value of each line. We could put a property on the Invoice class such as:

public property decimal TotalValue {

get {

decimal total = 0.0;

this.InvoiceLines.ForEach(line => total += line.TotalValue);

return total;

}

}

The Invoice class now has the ability to give us its total value, and this code is made simple by a relationship that is modelled between Invoice and InvoiceLine (giving us the InvoiceLines property on the Invoice class).

In a similar fashion, given an InvoiceLine we might want to find out some detail about the product that it applies to, such as the description of it to print on the invoice.

String lineDescription = invoiceLine.Product.Description;

Here we have modelled a relationship between InvoiceLine and Product – a “single” relationship, meaning there is only one Product for a given InvoiceLine, and thus the Product property returns the actual Product object which has the Description property on it.

It’s immediately obvious how these constructs make logic code more intuitive as the code becomes more expressive and a closer match to the domain.

#### Relationship Concepts

Relationships can always be expressed in a sentence, and, in fact, if this sentence does not make sense it offers an indication that the relationship does not fit in the Ubiquitous Language of your model. For instance, consider the sentence:

A car is owned by a person.

This denotes a relationship between a car and a person, where the context is that of ownership. This can be modelled in UML as follows:



Note that there can be more than one relationship between two entities. For example, a car can be owned by one person while being driven by another. You can also have an association between an entity and another entity of the same type – for example a person could have siblings (who are also people).

##### Directionality

The sentence above indicates that there is *directionality* in the relationship, as not much can be inferred in terms of the relationship in the other direction (for instance, there is no indication whether more than one car can be owned by a person. We can’t assume things from the real world when doing models otherwise we’d end up with a model bigger than necessary). A relationship that can only be traversed in one direction is *uni-directional*.

A fuller description of the relationship might read:

A car is owned by a person. A person owns zero or more cars.

This leads to a model like this:



This time there is no *directionality*. In other words the relationship is *bi-directional* (note that it could have been modelled as bi-directional before, but if had done that we would have been implying things that were not explicit in the relationship description). For bi-directional relationships it is conventional to drop the arrowheads in keeping with the agile approach (leave unnecessary things out.)

##### Multiplicity

We also know the *multiplicity* of the relationship now. A car can, at any point in time, be owned by one, and only one person (this again suggests that we are not modelling the real world in general, only the scope of the system, and in the system a car always has one owner.) A person, on the other hand, can own zero, one, or any number of cars.

##### Optionality

One more concept when modelling relationships is *optionality*. In our model, because the description says a car must be owned (by not giving the option of zero owners), this is not an optional relationship – a car must have an owner. This is denoted by the number 1 next to the person end of the relationship in the diagram. If this was optional the multiplicity be 0..1. The second half of the relationship description indicates that a person can have zero or more cars, so this is an optional link as denoted by the \* (or sometimes by 0...\*). If a person always owned at least one car the multiplicity indicator would be 1...\*.

##### Transitory and persistent

Relationships, as defined in Habanero and modelled in FireStarter are necessarily *persistent*. The domain model usually does not contain *transitory* relationships, that is, relationships between objects that are not persisted, so this discussion is purely centred on persistent relationships.

#### Relationship Types

##### Association

The simplest relationship type is the *association*, which denotes a loose association between two Entities. The above example is repeated here for convenience:

A car is owned by a person. A person owns zero or more cars.



The example given above between the car and the person is an association: each of the Entities can exist in their own right and the relationship simply links them in some way. A car can then later be delinked from one owner (person) and attached to a different owner. A person, in turn, can have all their cars removed and be left carless, or receive new cars.

##### Aggregation

Often objects are made up of other objects, or need to be treated as a collective. For example, a delivery company might bag a bunch of parcels together into a shipment to go to a certain country – in this case it would make sense that the shipment is made up of one or more parcels, and that a parcel is part of a shipment. This type of relationship is referred to as an *aggregation*.

A shipment consists of one or more packages. A package is either unassigned or assigned to one shipment



Note that while a shipment consists of one or more packages (and a package is thus a part of a shipment), a package can exist without a shipment before it is assigned to a shipment, and can be moved from one shipment to another. This type of qualification shows that indicates that this is an aggregation as opposed to a composition.

##### Composition

When an object is composed of other objects, or it is wholly responsible for interaction with its part objects you have a strong form of aggregation called *composition*. An example is that of an Invoice and its Invoice Lines – an invoice line cannot exist outside of being part of an invoice, and all outside interaction should be done through the Invoice.

An invoice consists of invoice lines. An invoice line cannot exist other than on an invoice.



Composition is sometimes referred to as *strong aggregation* and the two are denoted using the same diagram representation in UML 2.0. The difference between them is subtle, but important. In our examples it is evident that a package can be searched for and interacted with (in a system) on its own. A tracking number, for example might be used to search for the package, check its current status and see what shipment that package is on. In other words, the package has *identity of its own* apart from the shipment that it’s on (indicating aggregation). An invoice line on the other hand only exists as a constituent part of an invoice and does not have identity apart from the invoice that it is on (indicating composition, or strong aggregation).

#### Many to Many Relationships

TODO

### Using Relationships in Habanero

#### Modelling relationships

All relationships in Habanero are modelled as uni-directional relationships. This means that a bi-directional association such as:



Is modelled in Habanero as two uni-directional associations:



1. A car is owned by a person



1. A person owns zero or more cars.

The net effect is exactly the same model, as together the two relationships add up to the original relationship description.

The first, the relationship from Car to Person, will be depicted in code as a property called Owner on the Car class. This property returns an object of type Person:

public Person Owner { get; }

This relationship is what Habanero calls a *single* relationship. These can have a multiplicity zero or one and always return either null or a single object.

The second, the relationship from Person to Car will be depicted in code as a property on the Person class called Cars (this denotes ownership clearly enough, so I’m not going to qualify the name as OwnedCars or something similar). Cars returns a BusinessObjectCollection<Car>, a collection of Car objects:

public BusinessObjectCollection<Car> Cars { get; }

This relationship is what Habanero calls a *multiple* relationship. These can have a multiplicity of any number, and always return a collection.

Thus a many-to-one relationship such as the Car/Owner relationship will consist of one single relationship (Car to Person, called Owner) and one multiple relationship (Person to Car, called Cars).

In Habanero these two relationships, Owner and Cars, are considered *reverse relationships* of each other. When traversing the Owner relationship from Car to Person, the reverse relationship is the Cars relationship. This is important to model as without the link between the two relationships Habanero may not treat them as a proper bi-directional relationship. Linking the two uni-directional relationships makes them behave as one bi-directional relationship.

If the relationship you are modelling is a uni-directional relationship then no reverse relationship will be required.

##### Modelling Aggregation and Composition

The above example involved a pure association relationship. If we take an aggregation or composition relationship, such as our Shipment/Package example, the situation is only slightly different.



When breaking this relationship down into two uni-directional relationships for Habanero, the relationship from Shipment to Package would be modelled as aggregation as normal, but the relationship from Package to Shipment must be modelled as an association (or not at all if you don’t ever need to traverse from a Package to its Shipment). If both are modelled as aggregations Habanero will raise an error when they are used. The diagram is in fact indicating this because the aggregation icon is only connected to Shipment, not to Package as well.

The same thing applies to composition – only one of the two directions must be modelled as composition, the other must always be an association.

##### One to one

Modelling a one to one relationship is similar.

A manager manages a department. A department has one current manager.



Two uni-directional single relationships that are linked correctly will model this relationship. There is an extra consideration to note – it is important that only one of these relationships is configured to be the one that owns the foreign key. If both are considered to have the foreign key then Habanero will throw an error when the manager of a department is set.

#### Relationship Collection behaviour

In Habanero, the first choice to make when modelling is whether to use a single or a multiple relationships. This is not the only item required in order to fully describe the multiplicity of the relationship, but is the most fundamental of the properties of a relationship as it determines whether it points to a single object or a collection of objects, affecting the resultant client code.

Taking the above Owner/Car relationship as an example, the Owner relationship is modelled as a Single relationship, while the Cars relationship is a Multiple relationship (hence the Owner property returning a single Person object and the Cars property returning a collection of Car objects.)

##### Adding/Assigning

With this bi-directional association properly modelled, if we have a Person called bob, we can add a Car called toyotaCorolla to bob’s collection of cars in one of two ways:

bob.Cars.Add(toyotaCorolla);

or:

toyotaCorolla.Owner = bob;

These are, in fact, equivalent statements in Habanero, and both have the same effect on the object model in memory.

In the one-to-one Manager/Department example, if bill is the manager of a department called marketing then these two statements are equivalent (once again assuming the relationship is properly modelled with the reverse relationships configured):

bill.CurrentDepartment = marketing;

or:

marketing.CurrentManager = bill;

##### Removing/Unassigning

In a similar way, if we have a Person called bob who has a Car called toyotaCorolla in his collection of Cars, we can remove it in one of two ways:

bob.Cars.Remove(toyotaCorolla);

or;

toyotaCorolla.Owner = null;

These again have the same affect as each other. You can think of them being the exact same operation, just two ways to express them.

If we wanted to replace bob’s toyotaCorolla with a shiny new toyotaLexus we could do it in two equivalent ways:

bob.Cars.Remove(toyotaCorolla);  
bob.Cars.Add(toyotaLexus);

or:

toyotaCorolla.Owner = null;  
toyotaLexus.Owner = bob;

If bob sold his toyotaCorolla on to jim, we can indicate this as follows:

bob.Cars.Remove(toyotaCorolla);  
jim.Cars.Add(toyotaCorolla);

or:

toyotaCorolla.Owner = jim;

Finally, if bob sold his toyotaCorolla to jim and replaced it with a shiny new toyotaLexus:

bob.Cars.Remove(toyotaCorolla);  
jim.Cars.Add(toyotaCorolla);  
bob.Cars.Add(toyotaLexus);

or:

toyotaCorolla.Owner = jim;  
toyotaLexus.Owner = bob;

It is often simpler to use the single side of the relationship as it easier to read and understand, but it’s up to you what to use as they all accomplish exactly the same thing.

In the one-to-one example of Manager/Department we can remove a manager (bill) from a department (marketing), and vice versa, as follows:

bill.CurrentDepartment = null;

or:

marketing.CurrentManager = null;

##### Creating

For this we will use the example of an Invoice and Invoice Lines as it seems more intuitive to create an Invoice Line through an Invoice than create a Car through a Person.

If we have an invoice called invoice1000 and which to create a lines on it, we could write the following:

InvoiceLine line = invoice1000.InvoiceLines.CreateBusinessObject();

In Habanero it is also allowed to create an Invoice Line and add it to an Invoice, for convenience. To do this you could write:

InvoiceLine line = new InvoiceLine();  
 invoice1000.InvoiceLines.Add(line);

Again, these two forms are equivalent. The Add method realises the Invoice Line is new and tracks it as a created object as if it were created through InvoiceLines.CreateBusinessObject().

In the case of a one-to-one relationship the first form is not applicable, and the second form becomes:

Person bill = new Person();  
 marketing.CurrentDepartment = bill;

##### Marking for deletion

Using the example of an Invoice and Invoice Lines again, if we wish to delete line1 of an invoice called invoice1000 we could write:

invoice1000.InvoiceLines.MarkForDelete(line1);

Or we could write:

line1.MarkForDelete();

In the case of a one-to-one relationship the first form becomes:

marketing.CurrentManager.MarkForDelete();

The second form remains the same.

#### Mapping to a database model

In order to make the relationship you have modelled persistent to a relational database, you need to also indicate what fields model the relationship at the database level. The above bi-directional relationship between Car and Person might be modelled in the database as follows:



Thus the Owner relationship between Car and Person would be a single relationship which relates the PersonID field on Car to the PersonID field on Person. Meanwhile the Cars relationship between Person and Car would be a multiple relationship which relates the PersonID field on Person to the PersonID field on Car. By indicating to Habanero what fields in the database model the relationship, it will ensure that the fields are always updated to the correct values when persons or cars are saved.

Habanero supports multi-field keys, so we could have the same domain layer relationship mapping to a set of tables that look like this:



This can be done by linking the relationships on multiple properties (OwnerSurname to Surname and OwnerFirstName to FirstName).

With these properties configured, when an object’s relationship is changed in the domain layer, that change will be persisted to the database when the object or relationship is persisted. So, for example, consider an Invoice Line that is created in the following manner:

InvoiceLine line = invoice.InvoiceLines.CreateBusinessObject();

The created invoice line will be persisted when the Invoice is persisted, and the fields in the database that relate the InvoiceLine table to the Invoice table will be set correctly.

In the same way if a Car’s owner is changed in code the relevant properties are immediately updated, ready to be persisted when the car (or the owner) is persisted.

#### Modelling Options

Each relationship modelled in Habanero/FireStarter requires certain fields:

* Name: the name of the relationship. A property with this name will be generated on the class this relationship belongs to.
* Related class and related assembly: together these fields fully determine the related .NET entity type
* Type: this can be single or multiple, indicating the multiplicity of the relationship
* Related properties: these are pairs of property names that link a property of this class with a property of the related class, and are used in persistence and loading of the related objects.

A few more fields warrant some description too:

* Relationship type: this can be association, aggregation or composition. When modelling a bi-directional relationship only one of the directions should be modelled as an aggregation or composition – the reverse relationship should be left as association. The default relationship type is association.
* Reverse relationship: this is the name of the relationship that is the reverse relationship of this one (the other half of a bi-directional relationship). It is recommended to model this for all bi-directional relationships; Habanero is able to figure out the reverse relationship if you don’t model it, but it can be mistaken if there is more than one relationship between two entities that are related on the same fields.
* Delete action: this option determines what happens to related objects when this relationship’s owning object is deleted. The default option is Prevent. Note that not all options are available for all relationship types (see Behaviour). The possible options are:
  + Delete Related – delete all related objects as part of the same transaction
  + Dereference Related – clear the foreign keys of all related objects (set them to null) as part of the same transaction
  + Prevent – if any related objects exist disallow the delete process to continue
  + Do nothing – don’t do anything in particular. This option is available to support circular relationships and to allow the option of using the underlying database’s “cascade delete” options.
* Order by: an order by clause used to sort of the related objects as they are loaded. This only applies to multiple relationships
* Owning BO has foreign key: This only applies in a single to single (that is, a one to one) relationship, and is used to indicate which of these relationships holds the foreign key to the other (the default is true, so this should be set to false for the relationship that does not have the foreign key). Note that this field is very important in the case of a one to one as without it Habanero with throw an error when you try to set an object via the relationship.

#### Behaviour of different relationship types

Each of the different relationship types has a slightly different set of rules governing it reflecting the difference between the natures of each relationship.

##### Associations

The following rules apply to associations in Habanero. These rules are described in the context of the above Car and Person example to make them more readable.

1. A Car can exist independently of a Person if they are related by an association relationship.
2. A new (unpersisted) or already persisted Car can be added to the Cars of a Person : person.Cars.Add(car)
3. A Car can be removed from the Cars of a Person : person.Cars.Remove(car)
4. A Car can be created via the Cars of a Person : person.Cars.CreateNewBusinessObject()
5. A Car can be marked for delete via the Cars of a Person : person.Cars.MarkForDelete(car).
6. A Person may be deleted even if it has Cars (unless the Prevent Delete setting is used). If a Person is deleted, the valid strategies for how this delete is applied to its Cars are: Prevent Delete, Dereference or Do Nothing. Cascading of deletes (Delete Related) is not allowed for associations.
7. A Person is considered to be dirty (that is, requiring persistence) if it has added, created, marked for delete or removed Cars
8. A Person is not considered to be dirty if it has any previously persisted Cars that are dirty
9. If a Person is persisted then its Cars relationship will be persisted, but not the Cars themselves. In other words, any created or new Cars will be saved and any marked for delete Cars will be deleted. Added or removed Cars that were previously persisted will only have their relationship properties persisted. If they have other dirty properties these will be left alone.

##### Aggregations

The following rules apply to aggregations in Habanero. These rules are described in the context of the Shipment/Package relationship described above.

1. A Package can exist independently of a Shipment if they are related by an aggregation relationship
2. A new (unpersisted) or already persisted Package can be added to the Packages of a Shipment : shipment.Packages.Add(package)
3. A Package can be removed from the packages of a Shipment : shipment.Packages.Remove(package)
4. A Package can be created via the packages of a Shipment : shipment.Packages.CreateNewBusinessObject()
5. A Package can be marked for delete via the packages of a Shipment : shipment.Packages.MarkForDelete(package).
6. A Shipment may not be deleted if it has Packages (although these may be removed by using an appropriate deletion strategy). The valid strategies for how this delete is applied to its Packages are: Prevent Delete, Dereference (remove), Delete Related, or do nothing.
7. A Shipment is considered dirty (that is, requiring persistence) if it has added, created, marked for delete or removed Packages
8. A Shipment is considered dirty if any of its Packages are dirty.
9. If a Shipment is persisted then all its Packages are persisted

##### Compositions

The following rules apply to composition in Habanero. These rules are described in the context of the Invoice/Invoice Line relationship described above.

1. An Invoice Line cannot exist independently of an Invoice if they are related by a composition relationship
2. A new (unpersisted) Invoice Line can be added to an Invoice: invoice.InvoiceLines.Add(invoiceLine). An already persisted Invoice Line cannot be added to an Invoice.
3. An Invoice Line cannot be removed from an Invoice
4. An Invoice Line can be created via the Invoice Lines of an Invoice: invoice.InvoiceLines.CreateBusinessObject()
5. An Invoice Line can be marked for delete via the Invoice Lines: invoice.InvoiceLines.MarkForDelete(invoiceLine)
6. An Invoice may not be deleted if it has Invoice Lines (although these may be deleted by using an appropriate deletion strategy). The valid strategies for how this delete is applied to its Invoice Lines are: Prevent Delete, Delete Related or Do Nothing.
7. An Invoice is considered to be dirty (that is, requiring persistence) if it has created (or new, added), or marked for delete Invoice Lines.
8. An Invoice is considered to be dirty if any of its Invoice Lines are dirty.
9. If an Invoice is persisted then all of its Invoice Lines are persisted.

##### Summary

The table below compares the different relationship types side by side to highlight their similarities and differences. The table refers to parents and children in a general sense – the parent is the owner of a uni-directional relationships, while the child (or children) is (or are) the related object (or objects). For example, in the Person/Car relationship “a person can own zero or more cars”, the Person is the parent and the Cars are the children. In the Car/Person relationship “a car is owned by one person”, the Car is the parent and the Person is the child.

The essential difference between association and the other two relationship types is what is persisted when you persist the parent object – in an association only added, removed, created and deleted objects are persisted – all other children are not. In aggregation or composition, all children are persisted (if they are dirty/changed).

The essential difference between aggregation and composition is whether a child object can move from one parent to another. With aggregation the child object can exist without the parent or can be moved from one parent to another, whereas with composition the child is linked to its parent for good, never to be delinked.

|  |  |  |  |
| --- | --- | --- | --- |
|  | Association | Aggregation | Composition |
| Child can exist independently? | Yes | Yes | No |
| New child can be added? | Yes | Yes | Yes |
| Already persisted child can be added? | Yes | Yes | No |
| Child can be removed? | Yes | Yes | No |
| Child can be created? | Yes | Yes | Yes |
| Child can be marked for deletion? | Yes | Yes | Yes |
| Deletion strategies allowed | Prevent Delete; Dereference; Do Nothing | Prevent Delete; Dereference; Delete Related; Do Nothing | Prevent Delete; Delete Related; Do Nothing |
| Parent is dirty if has created children? | Yes | Yes | Yes |
| Parent is dirty if has added children? | Yes | Yes | N/A (new added children are considered created.) |
| Parent is dirty if has removed children? | Yes | Yes | N/A |
| Parent is dirty if has children marked for deletion? | Yes | Yes | Yes |
| Parent is considered dirty if its children are dirty? | No | Yes | Yes |
| Dirty relationships are persisted? | Yes | No (entire child is persisted, not just relationship) | No (entire child is persisted, not just relationship) |
| Dirty children are persisted? | No | Yes | Yes |

### Example

TODO

### Implementation

Habanero’s implementation of relationships uses a set of definition objects to model the definition of a relationship, and then another set of objects that represent the relationship itself at run-time – similar to the way its Business Objects are implemented.

#### Relationship Definitions



When modelling relationships in FireStarter, or in XML, you are defining the structures of the relationship (that is the definitions). The definition of a relationship (the RelationshipDef) has a number of properties (such as the relationship type, the delete action, the related class and so on), and it contains a single RelKeyDef that defines the key that the relationship will use to load the objects from a relational database. This in turn contains one or more RelPropDefs – one for each field in the key.

There are two subclasses of RelationshipDef: SingleRelationhipDef and MultipleRelationshipDef, each having the fields specific to that type of relationship.

A typical XML representation of the Car/Owner relationship would be:

<class name="Person">  
 <property name="PersonID" type="Guid" />  
 <primaryKey>  
 <prop name="PersonID" />  
 </primaryKey>  
 <relationship name="Cars" type="multiple" relatedClass="Car" reverseRelationship="Owner" >  
 <relatedProperty property="PersonID" relatedProperty="PersonID" />  
 </relationship>  
</class>  
<class name="Car">  
 <property name="CarID" type="Guid" />  
 <primaryKey>  
 <prop name="CarID" />  
 </primaryKey>  
 <relationship name="Owner" type="single" relatedClass="Person" reverseRelationship="Cars" >  
 <relatedProperty property="PersonID" relatedProperty="PersonID" />  
 </relationship>  
</class>

Because the Car/Owner relationship is an association relationship, the relationship type is not required (associations are the default). Note that the complete model requires the reverseRelationship property to be set so that the relationships are linked together into one bi-directional relationship.

#### Relationship Instances



This diagram indicates what happens at runtime: when a BusinessObject is created it asks each RelationshipDef in its ClassDef to instantiate a Relationship of the correct type (either SingleRelationship<T> or MultipleRelationship<T> where T is the type of the related class), and this Relationship is added to its relationship collection. The RelKey and RelProp objects of a Relationship mirror the RelKeyDef and RelPropDef directly, and are simply instantiations of those definitions. These can be ignored for almost all purposes.

Each BusinessObject has a collection of Relationships which can be accessed by name via its Relationships property. The SingleRelationship’s SetRelatedObject and GetRelatedObject can be used to set or get the object on the other end of the relationship, while MultipleRelationship’s BusinessObjectCollection retrieves the collection of objects at the other end of that relationship.

Most use of relationships when writing client code is done by using the properties generated by FireStarter, and as such the above structures should remain invisible to you. All the loading, updating of properties including the setting of foreign keys, mapping to the database and so on, is done behind the scenes – you need only concentrate on the domain model itself.

#### Accessing relationships dynamically

It can happen that you need to configure a relationship to work in a specific way at run time, perhaps depending on a setting set by the user or some business rule. It is always possible to access the collection of relationships of a Business Object using the Relationships property.

As an example, let’s say I want to write a general purpose utility to dump an entire object tree (of unknown type) to a text file, for debugging or troubleshooting purposes. I also only want to dump related objects if they are related by composition, and ignore aggregations and associations.

public void Dump(IBusinessObject bo, Stream stream)

{

StreamWriter writer = new StreamWriter(stream);

writer.WriteLine("------ {0} : {1}", bo.ClassDef.ClassName, bo);

foreach (IBOProp prop in bo.Props)

{

writer.WriteLine(prop.PropertyName + ": " + prop.Value);

}

foreach (IRelationship relationship in bo.Relationships)

{

if (relationship.RelationshipDef.RelationshipType != RelationshipType.Composition) continue;

writer.WriteLine("--- Relationship : {0}", relationship.RelationshipName);

IMultipleRelationship multipleRelationship = relationship as IMultipleRelationship;

if (multipleRelationship != null)

{

foreach (IBusinessObject relatedBO in multipleRelationship.BusinessObjectCollection)

{

Dump(relatedBO, stream);

}

}

ISingleRelationship singleRelationship = relationship as ISingleRelationship;

if (singleRelationship != null)

{

IBusinessObject relatedObject = singleRelationship.GetRelatedObject();

if (relatedObject != null) Dump(relatedObject, stream);

}

}

}

In this sample code, first the type of the Business object is dumped, then the properties, and then the related objects. There is a little work done to figure out whether each IRelationship is an ISingleRelationship or an IMultipleRelationship, and to do the appropriate operation for each, and there is one check to see if the relationship is a composition one, but overall the code is really simple.

Of course, you can develop as if all these constructs are not there (which is a good thing – you need to be concentrating on your domain model rather than the infrastructure), but when you need to do something generic like this it’s entirely possible to write code that applies to each and every object without any complex reflection logic, a very powerful possibility.

#### Single Relationship Behaviour

We’ve already seen how the single relationship is used in client code:

Person owner = car.Owner;

The Owner property itself has been generated as follows:

public virtual Person Owner {

get {

return Relationships.GetRelatedObject<Person>("Owner");

}

set {

Relationships.SetRelatedObject("Owner", value);

}

}

The Relationships collection is passing the GetRelatedObject<Person>(string relationshipName) call through to the SingleRelationship<Person> called Owner, and the same for the SetRelatedObject(string relationshipName, object value) call.

##### GetRelatedObject

The first time the GetRelatedObject<T>() (or GetRelatedObject()) method is hit, the relationship loads the related object from the data source. From then on, the loaded object is cached and remains cached in memory as long as its owning Business Object is cached. In the example above, once the Owner property’s get is called the owner will stay in memory as the car stays in memory. Further calls to GetRelatedObject result in the cached object being returned, unless something is changed.

If by some chance the properties that are relating the two objects to each other are changed manually, for example:

car.OwnerID = newOwner.OwnerID;

or, if the OwnerID is not settable (as is correct):

car.SetPropertyValue(“OwnerID”, newOwner.OwnerID);

This code has bypassed the relationship which is not recommended, but could be useful if performance is vital such as for an import procedure (although this should only be decided after actual performance analysis). When GetRelatedObject is later called a quick check is made to see if the “foreign key” properties have changed, and if they have then the object will be reloaded using the new properties.

##### SetRelatedObject

This method is used to change the current related object. Calling it triggers off a GetRelatedObject() call so that the relationship knows if the object being set is really a new object or is the same as the current one. If the object being set is the same, nothing happens, but if its new then the previous object will be removed from any reverse relationship and the new one added to any reverse relationship. We saw this earlier in the example of bob and jim. If bob is the current owner of toyotaCorolla, and we do the following:

toyotaCorolla.Owner = jim;

SetRelatedObject will remove toyotaCorolla from bob’s cars (this is the reverse relationship of the Owner relationship) and add it to jim’s cars. This is why modelling the reverse relationship is of vital performance to a sensibly behaving domain model.

#### Multiple Relationship Behaviour

We’ve already seen how the multiple relationship is used in client code:

BusinessObjectCollection<Car> cars = person.Cars;

The Cars property on the Person class has been generated as follows:

public virtual BusinessObjectCollection<Car> Cars

{

get {

return Relationships.GetRelatedCollection<Car>("Cars");

}

}

The GetRelatedCollection<Car> call on the Relationships collection finds the appropriate collection called Cars (which is a MultipleRelationship<Car>), and returns the BusinessObjectCollection property, thus returning a BusinessObjectCollection<Car>.

##### BusinessObjectCollection

Calling this property, as the above example code does, causes the collection of objects that make up this relationship to refresh (or load if it is the first time). This happens every time the property is accessed – the database (or other datastore) is hit each time to keep the relationship collection thoroughly up to date. This obviously has an impact on performance, so where necessary a reference to the returned collection should be retained.

It is possible to get at the collection without refreshing it by calling CurrentBusinessObjectCollection (accessible via the IMultipleRelationship interface). This property returns the current collection as is, so if you have not ever loaded it (via a call to BusinessObjectCollection) it will be empty.

It is worth noting that although the return type is BusinessObjectCollection, the actual object returned is a subtype of that, the RelatedBusinessObjectCollection. This is important because the RelatedBusinessObjectCollection overrides some of the behaviour of the BusinessObjectCollection when it comes to adding, creating, removing and marking objects for deletion. The rules governing this have already been discussed (see Behaviour of different relationship types), but suffice to say that these collections do not always act exactly the same as a normal BusinessObjectCollection as discussed in Chapter 4, and in fact act differently depending on what relationship type the relationship is.

### Extending Relationships