## Chapter 5: The Domain Model – Relationships

### Relationships

Its all very well to be able to load business objects and to load collections of business objects but the power of the Domain model is really in correctly modelling the relationships between Business Objects.

To be able to discuss object relationships we need to define and explain a few terms.

**Collaboration:** Objects/classes are defined primarily by their responsibilities (What they are responsible for doing). To fulfill their responsibilities objects collaborate with each other. Collaboration occurs when one object asks another object to do something or asks another object for information this is done by sending messages. This implies that there is some sort of relationship between these objects.

**Relationship**: For an OO program to achieve anything the classes of the program need to collaborate with each other in various ways. These collaborations occur via relationships and method calls.

**Persistent relationships**: a semi-permanent relationship between two persistent objects that is persisted to an DataStore to be used by later (other) instances of the program.

**Transient relationship**: a relationship between two objects that is temporary in nature and is not saved.

**Association**: A relationship between two classes or objects.

**Optionality**: The concept of "do you need to have it?". E.g. Must an employee be associated with a department.

**Cardinality**: A relationship between two classes or objects. How many departments can an employee be associated with.

**Multiplicity**: UML combines the concept of optionality and cardinality.

**Aggregation**: Relationships between two classes or components defined as "is part of".

**Composition**: A strong form of aggregation in which the "whole" is completely responsible for its parts and each "part" object is only associated to one "whole" object. E.g Invoice and Invoice lines. The invoices lines only make sense in terms of the invoice. The invoice lines cannot be created, retrieved, deleted or persisted independently of the invoice. The invoice lines can never be moved to another invoice.

The differentiation between Aggregation and Composition has been dropped from UML 2.0 but is still frequently referred to in texts on Object Oriented system development.

#### Association Relationships In UML:



An association relationship is a persistent relationship between two classes.

**Directionality.** The open arrowheads indicate the directionality of the association. When there is one arrowhead the association is unidirectional: it can be traversed in one direction only (in the direction of the arrow). Where there are two or no arrowheads the relationship can be traversed in both directions.

**Multiplicity.** The multiplicity of the association is labeled on either end of the line, one multiplicity indicator for each direction. I.e. above an employee manages 0 or more departments. A department is managed by exactly 1 employee. Can have a car has 0..5 wheels for multiplicity. So more capability that an ERD which would merely have 0..\*.

**Label.** The label, which is optional, is typically one or two words describing the association. Reading one class, the label, and then the other class should produce a sentence fragment that describes the relationship, e.g., *Professor teaches Seminars*. Avoid generic labels like "has" or "communicates with" as much as possible. E.g. Manages.

**Role.** The role—the context that an object takes within the association— may also be indicated at each end of the association. E.g. Manager.

Two classes can have more than one relationship between them. E.g. A person could lecture one course, Assist in another and be a student in another.

Relationships can be recursive E.g. An Employee manages Employees.

**Association relationships are inherited** i.e. a sub class of a class that has an associative relationship has the same relationship.

**Implementing Associative relationships:** Relationships are implemented in code. Industry norm is that the fields in the database used to implement relationships are not exposed by the business object. I.e. EmployeeID would not be exposed from the Department object above. The Employee would have a addManagedDepartment method, a removeManagedDepartment method and a getManagedDepartments method. In this bidirectional relationship case the Department would also have a getManager and may have a setManager method. *Habanero implements slightly differently.*

#### Aggregation Relationships In UML:



Aggregation is a specialised form of an Association relationship. The common test for aggregation is ‘Is Part of’.

E.g. an airplane is made up of a fuselage, wings, engines etc. A delivery shipment contains one or more packages. A project team consists of two or more employees. These represent "is part of" relationships

Composition is a strong form of aggregation in which the "whole" is completely responsible for its parts and each "part" object is only associated to the one whole object (see scott ambler). Even for composition there are various forms of strength of the relationship. E.g. An invoice is made up of invoice lines. If the invoice is destroyed its lines are destroyed, the lines cannot be associated with any other invoice (I.e. an invoice line cannot be moved from one invoice to another invoice), the lines cannot exist except as part of the invoice etc. Whereas a car is composed of its parts e.g. engine. While the engine is in the car the car is wholly responsible for it (i.e. the developer working with the car interface should not be calling Engine.increase speed). But the engine can be moved 2 another car and can exist by itself.

A delivery is composed of one or more packages is even weaker where the packages exist independently of the delivery and the delivery is merely a temporary convenient grouping of packages. The package can be moved from one planned delivery to another planned delivery.

Since Aggregation is a specialised form of an association it has all the properties of an assocation relationship multiplicity, roles etc.

Note: In UML2 you can no longer model Composition and aggregation differently both use the closed diamond now.

The following tips and techniques should help you to model composition/aggregation effectively:

1. **Apply the sentence rule.** First should make sense to say "the part *is part of* the whole."
2. **The whole should manage the part.** Secondly whether the whole manages the part. For example, an airplane should manage its engines.
3. **You should be interested in the part.** An object may actually be a part in the real world, but if you are not interested in keeping track of it, then do not model it. For example, an airplane maintenance system would be interested in keeping track of engines because it needs to record maintenance information about each engine. On the other hand, an air-traffic control system is not interested in tracking engines, just airplanes.

**Composition and aggregation is inherited.** Composition associations, like ordinary associations, are maintained by a combination of attributes and methods that can be inherited.

#### Dependency Relationships In UML:



Dependency relationships are transitory i.e. the relationship is not saved.

All relationships between non persistable objects are by definition transitory. Transitory relationships may however occur between two persistable object. E.g. A course is passed a student object to determine whether the student has the required prerequisites to be enrolled on the course. (scott Ambler)

#### Inheritance Relationships In UML:

Two classes are often similar and may share similar methods or similar data (attributes). Inheritance is a relationship that allows you to reuse methods or attributes that are common. The inheritance relationship between two objects is one of the most powerfull relationships in object oriented programming but is also one of the most misused by less experienced developers. The Inheritance relationship is defined as an ‘Is a’ or ‘Is a type of’. A super Class in this case Contact is a class from which another class inherits a Sub Class in this case the Customer is a class that inherits from another class. In this case Customer Inherits from Contact. Contact may be an abstract or a concrete class.



### Relationships In Habanero:

From the above you can see that the correct identification of the relationship between two Business Objects is critical for a successful domain model.

Habanero provides a framework to implement any required relationship including Inheritance. The dependency relationship is obviously not modelled since it is not a persistent relationship.

Have cascading effects i.e. if a class is added to a relationship update automatically.

If class is removed update automatically

If a class is deleted update automatically

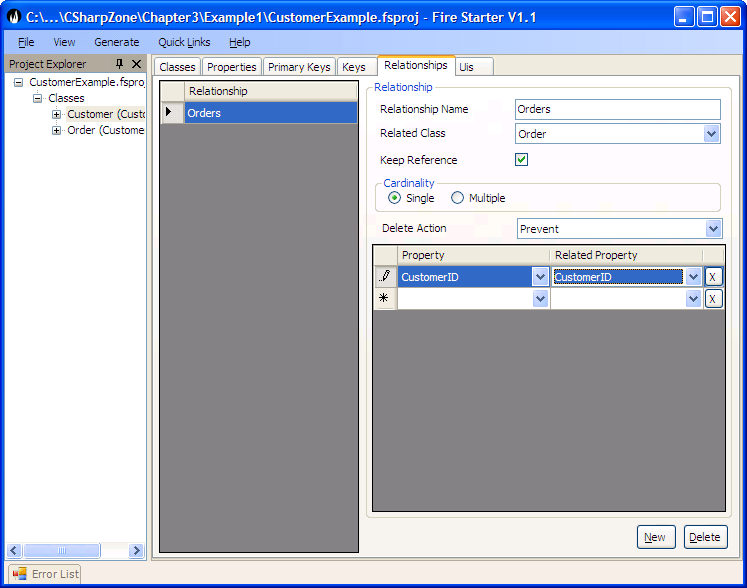
vs cascade update when required

#### 1:M Relationships



OR





These relationships should implement the following.

|  |  |
| --- | --- |
| ParentClassName | Name of the class that owns the relationship. E.g. Invoice/Customer |
| RelatedClassName | Name of the class that this relationship is to. E.g. InvoiceLines/Order |
| ParentClassRelationshipPropertyList | The list of Properties in the parent class that implement the relationship. i.e. what properties on Customer provide the values that relate to the Order. Ideally this will be the PrimaryKey of the Customer (CustomerID). |
| RelatedClassRelationshipPropertyList | The list of properties in the related class that implement the relationship. (Usually only one property the ‘Foreign key’ in the related class. E.g. The CustomerID in the Order Class.). |
| loadCriteria | This is used when the relationship has constraints. E.g. An Customer may have two relationships to Orders e.g. has a relationship current Orders (i.e. all orders that are not yet fulfilled) as well as a relationship of all the Orders that have ever been placed (including those that have been fulfilled).  This fulfils the role of a constraint modelled on the relationship in UML. |
| orderBy | By default what order do you expect to see the related objects loaded in e.g. Customer Orders must be loaded in delivery date the sort order for presenting on forms and in reports. |
| MinNumRelatedObjects | Can be 0 or more. In the case of a car having wheels the car can have a minimum of 3 wheels. |
| MaxNumRelatedObjects | Can be 2 or more. Once again in the case of a car maybe a max of 4 wheels (excluding issues like spare wheels) |
| Delete Action | What must the framework do when trying to delete the parent class if there are related objects. E.g. in the case of a customer the Customers orders make no sense without a related customer. In this case you would either set up to delete related (The framework will then delete all Customer orders if the customer is deleted) or prevent delete (The framework will prevent the deletion of the Customer if it has any related Orders but will allow the deletion if there are no related orders.) |
| CreateRelatedObjects | True if this relationship can create related objects. This would typically be a very strong relationship where the Parent object totally owns the child object e.g. An Invoice should create an invoice line. In most cases a customer should probably not create its Orders (This is a decision that is dependent on the domain). |
| DeleteRelatedObject | True if this relationship can delete related objects. In this case if a Customer Order is removed from the Collection of customer orders for a customer must it be deleted from the database. |
| RemoveRelatedObject | True if this relationship can remove related objects from its collection. I.e. if a Wheel is removed from its relationship to a car then the wheel should be dereferenced from the Car (since the wheel can exist as an object in the database without a reference to a Car). |
| AddObject | True if this relationship can add related objects to its collection (other than via CreateObjects). In the case of a wheel a wheel can be added to a Car and the wheel will then be related to the Car. This would be false in the case of an Invoice Line where the only way to associate an Invoice Line to an Invoice should be by the Invoice Creating the Invoice Line. |
| parentPersistsChildren | Does the parent of the relationship ‘own’ its children e.g. if an invoice is persisted then all edits to the invoice lines must also be persisted via the invoice. The save on the invoice line should thus throw an error. |
|  |  |

If you Call CreateObject or Add Object then the objects relatedProperties will be updated with the parentsID and the object will be added to an AddedChildred Collection.?? Or should adding an existing object just add it to the main collection.

If you call DeleteObject is called then the object.delete is called and it is added to the DeletedChildren collection.

If you call RemoveObject then the object’s relatedProperties are updated to null. The object will be added to the relationships RemovedObjects collection.

When parent.Save is called then all the objects in the added, deleted and removed collections will be persisted. Note in the case of objects being removed or added where the parentPersistsChildren is not set true we should consider only updating the related properties and leaving any other dirty properties.

Questions: Should the relationship register for the updated/deleted etc event of all of its children I think so.

Should the parent register for the updated/deleted events from its relationships?

Cancel Edits on Parent must this cancel all the relationships.

Think of having warning for save and cancel as well as errors.

E.g. BO has .CanSave and CanDelete that returns UserWarning and msg or UserError and Msg

The Save will however only throw a UserError

How deal with a child that deletes independently of the parent? How get msg back to parent.

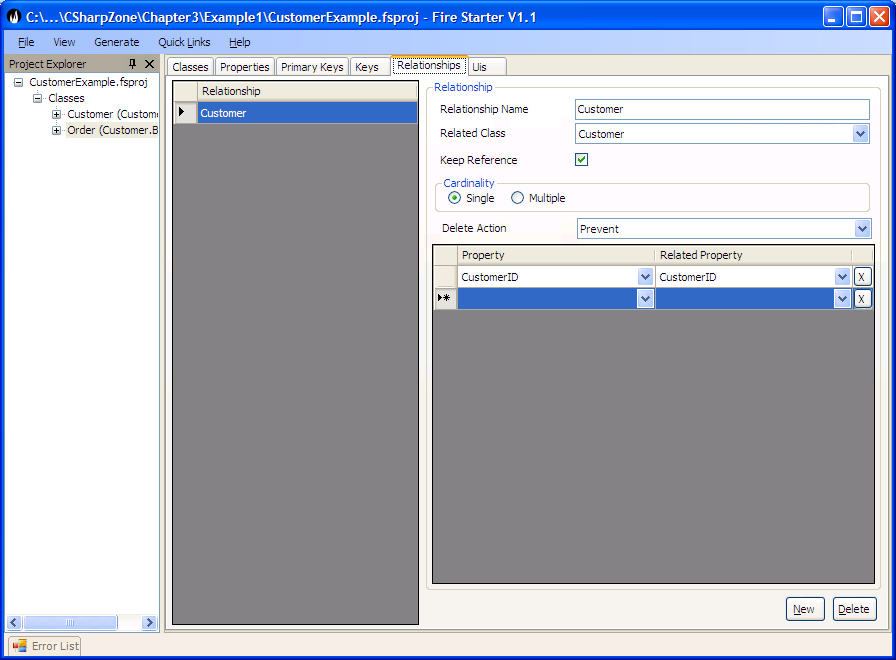
#### M:1 Relationships



OR



The M:1 relationship is the reverse of a 1:M relationship. This relationship is required when you need to access the relationship from the many side e.g. A Customer Order needs to be able to access the customer that it belongs to or an Invoice line needs to access the invoice. This relationship is very simple e.g. The Order will have a method GetCustomer which will return the customer related to the order. Whether the relationship is compulsory or not is determined by the rules set on the property i.e. if the CustomerID property on the Order object is set as Compulsory then the relationship is compulsory else the relationship is set as optional.



The M:1 relationship should implement the following.

|  |  |
| --- | --- |
| ParentClassName | Name of the class that owns the relationship. E.g. InvoiceLine/Order |
| RelatedClassName | Name of the class that this relationship is to. E.g. Invoice/Customer |
| RelationshipName | In some cases the relationship name will differ from the related class name e.g. if an Employee of a company is the manager of a department then this relationship between the department and the person has a relationship name of manager. |
| ParentClassRelationshipPropertyList | The list of Properties in the parent class that implement the relationship. i.e. what properties on Customer provide the values that relate to the Order. Ideally this will be the PrimaryKey of the Customer (CustomerID). |
| RelatedClassRelationshipPropertyList | The list of properties in the related class that implement the relationship. (Usually only one property the ‘Foreign key’ in the related class. E.g. The CustomerID in the Order Class.). |
| Delete Action | This should always be Do Nothing |

#### 1:1 Relationships



The 1:1 relationships is modelled in firestarter in the same way that the M:1 is modelled. The only real difference is that if the relationship is navigated in both directions then the reverse relationship is also a 1:1. In a 1:1 relationship it is not as obvious which object should have the foreign key reference. In these cases we recommend that if there is some sort of an owning relationship then the foreign key is placed on the child e.g. in this case the Engine. Firestarter and Habanero effectively work out both directions of the relationship regardless.

#### M:M Relationships



The Many to many (M:M) relationship is a lot less common than the relationships discussed so far but is still fairly common. Many people confuse a true M:M with two 1:M relationships this is often the result of how these are modeled in the database but in Domain Modeling there is no need to implement an intermediate class. At this point it is important to understand a true M:M. This occurs when two objects such as a driver can be permitted to drive many cars and a car can be driven by many drivers. There is no extra information stored on the M:M relationship. This is different from what is sometimes mistaken as a M:M but is actually two 1:M’s i.e. when the relationship stores extra information e.g. Employee : Department, this appears to be a M:M but in fact the relationship has additional information e.g. the date that the employee joined the department.

If you find yourself using lots of M:M relationships then you should review your domain model as this might indicate a problem.

At the moment you cannot model a M:M directly into Firestarter.

Car Class

Get Driver

Could have code that checks if the getCarsCollection IsLoaded so as to avoid loading Cars unneccessarily.

AddDriver(Driver addedDriver)

{

CheckAddedDriverNotNull

CheckAddedDriver not already associated with the Car.

addedDriver.getCars.Add(this);

this.Drivers.Add(addedDriver);

}

#### Composition Relationships in Habanero:

The composition relationship is independent of being a 1:M or a 1:1 by definition a M:1 and a M:M cannot be a composition relationship.

* A typical example of a composition relationship is an Invoice and its Invoice lines. An invoice Line cannot exist independently of its invoice and an invoice line can only belong to a single invoice.
* An invoice that has invoice lines cannot be deleted without it deleting its invoice lines. The invoice’s InvoiceLines relationship would be marked as either prevent delete, delete invoice lines or do nothing.
* An already persisted invoice line cannot be added to an Invoice (In habanero a new invoice line can be added to an invoice).
* An Invoice line cannot be removed from its invoice.
* An invoice can create a new invoice line via its InvoiceLines Relationship.
* An invoice is considered to be dirty if it has any dirty invoice line. A dirty invoice line would include a newly created invoice line and an invoice line that has been marked for deletion.
* If an invoice is persisted then it must persist all its invoice lines.

#### Aggregation Relationships in Habanero:

The aggregation relationship can be a 1:M or a 1:1. By definition a M:1 and a M:M cannot be a aggregation relationship.

* A typical example of an aggregation relationship is a Car and its Tyres. A Tyre can exist independently of its Car and a Tyre can only belong to a single Car at any point in time. The Tyre may however be transferred from one car to another.
* The Car that has tyres cannot be deleted without it deleting or removing its tyres. The car’s Tyres relationship would be marked as either prevent delete, dereference tyres, delete tyres or do nothing.
* An already persisted tyre can be added to a car (In habanero a new tyre can be added to a car). A tyre can be removed from its car.
* A car can create a new tyre via its Tyres Relationship.
* A car is considered to be dirty if it has any dirty tyres. A dirty tyre would include a newly created tyre, an added tyre, a removed tyre or a tyre that has been marked for deletion.
* If a car is persisted then it must persist all its tyres.

#### Associative Relationships in Habanero:

The associative relationship can be a 1:M or a 1:1. All M:1 and a M:M are associative relationships.

* A typical example of an associative relationship is a Car and its Drivers (assuming a car can have many drivers but a driver may only drive one car). A Driver can exist independently of any Car and a Car can exist independently of a driver. The Driver may however be associated with one car and later associated with a different car.
* The rules for whether a car that is associated with one or more drivers can be deleted or not is dependent upon the rules configured for the Drivers relationship (i.e. a car’s drivers relationship could be marked prevent delete, dereference or do nothing).
* An already persisted driver can be added to a car (In habanero a new driver can be added to a car).
* A driver can be removed from its related car.
* A car can create a new driver via its Drivers Relationship (this is not a strict implementation of domain design but is allowed due to the convenience of this).
* A car is considered to be dirty only if it has added, created or removed dirty drivers.
* If a car is persisted then it will only persist its driver’s relationship and will not persist a related driver that is dirty.

#### Inheritance Relationships in Habanero:

TODO: fdsafsafdasf

private Vector instructors;

public void addInstructor(Professor professor)

{

// If the professor does not exist in the collection

add it

// The if statement avoids an infinite loop managing

the association

if (! getInstructors().contains(professor)) {

getInstructors().add(professor);

// Update the other end of the association

association

professor.addSeminar(this);

}

}

public void removeInstructor(Professor professor)

{

if (getInstructors().contains(professor)) {

getInstructors().remove(professor);

// Update the other end of the association

professor.removeSeminar(this);

}

} This is an excellent technique we need to consider this would require relationships to cross reference related relationships.

Composition, Aggregation, Association