EORelationship

Inherits From:	NSObject
Conforms To:	NSObject (NSObject)
Declared In:	EOAccess/EORelationship.h

Class Description

An EORelationship describes an association between two entities, based on attributes of those two entities. By defining EORelationships in your application's EOModel, you can cause the relationships defined in the database to be automatically resolved as enterprise objects are fetched. For example, an Employee entity may contain its department ID as an attribute, but without an EORelationship this will only appear in an employee enterprise object as a number. With an EORelationship explicitly connecting the Employee entity to the Department entity, an employee enterprise object will automatically be given its department enterprise object when an EODatabaseChannel fetches it from the database. The two entities that make up a relationship can be in the same model or two different models, as long as they are in the same model group.

You usually define relationships in your EOModel with the EOModeler application, which is documented in the *Enterprise Objects Framework Developer's Guide*. EORelationships are primarily for use by the Enterprise Objects Framework; unless you have special needs you shouldn't need to access them in your application's code.

A relationship is directional: One entity is considered the source, and the other is considered the destination. The relationship belongs to the source entity, and may only be traversed from source to destination. To simulate a two-way relationship you have to create an EORelationship for each direction. Although the relationship is directional, no inverse is implied (although an inverse relationship may exist).

A relationship maintains an array of joins identifying attributes from the related entities (see the EOJoin class specification for more information). Most relationships simply relate the objects of one entity to those of another by comparing attribute values between them. Such a relationship must be defined as to-one or to-many based on how many objects of the destination match each object of the source. This is called the *cardinality* of the relationship. In a to-one relationship, there must be exactly one destination object for each source object; in a to-many relationship there can be any number of destination objects for each source object. See "Creating a Simple Relationship," below, for more information.

A chain of relationships across several entities can be flattened, creating a single relationship that spans them all. For example, a relationship between employees and departments and a relationship between departments and facilities can be flattened into a relationship going directly from employees to facilities. A flattened relationship is determined to be to-many or to-one based on the relationships it spans; if all are to-one, then the flattened relationship is to-one, but if any of them is to-many the flattened relationship is to-many. See "Creating a Flattened Relationship," below, for more information.

Like the other major modeling classes, EORelationship provides a user dictionary that the application can use to store application-specific information related to the relationship.

Specifying the Join Semantic

The relationship holds the join semantic; you specify this semantic with **setJoinSemantic:**. There are four types of join semantic, as specified by the EOJoinSemantic type: EOInnerJoin, EOFullOuterJoin, EOLeftOuterJoin, and EORightOuterJoin. An inner join produces results only for destinations of the join relationship that have non-NULL values. A full outer join produces results for all source records, regardless of the values of the relationships. A left outer join preserves rows in the left (source) table, keeping them even if there's no corresponding row in the right table, while a right outer join preserves rows in the right (destination) table.

Note: Not all join semantics are supported by all database servers.

Creating a Simple Relationship

A simple relationship is defined by the attributes it compares in connecting its source and destination entities. Each source-destination pair of attributes is encapsulated in an EOJoin object. For example, to create a relationship from the Employee entity to the Department entity, a join has to be created from the **deptID** attribute of the Employee entity to the same attribute of the Department entity. The values of these two attributes must be equal for a match to result. Note that **deptID** is the primary key attribute for the Department entity, so there can only be one department per employee; this relationship is therefore to-one.

This code excerpt creates an EORelationship for the relationship described above and adds it to the EOEntity for the Employee entity:

```
EOEntity *employeeEntity;      /* Assume this exists. */
EOEntity *deptEntity;          /* Assume this exists. */
EOAttribute *empDeptIDAttribute;
EOAttribute *deptIDAttribute;
EOJoin *toDeptJoin;
EORelationship *toDeptRelationship;

empDeptIDAttribute = [employeeEntity attributeNamed:@"deptID"];
deptIDAttribute = [deptEntity attributeNamed:@"deptID"];

toDeptJoin = [[[EOJoin alloc]
    initWithSourceAttribute:empDeptIDAttribute
    destinationAttribute:deptIDAttribute] autorelease];

toDeptRelationship = [[[EORelationship alloc]
    initWithName:@"department"]] autorelease];
[toDeptRelationship setName:@"department"];
[employeeEntity addRelationship:toDeptRelationship];
[toDeptRelationship addJoin:toDeptJoin];
```

```
[toDeptRelationship setToMany:NO];
[toDeptRelationship setJoinSemantic:EOInnerJoin];
```

This code first gets the attributes from the source and destination entities, and then creates an EOJoin with them. Next, a new EORelationship is created, the EOJoin is added to it, and it's set as to-one. Then, in the **setJoinSemantic:** line, EOInnerJoin indicates that only objects that actually have a matching destination object will be included in the result when the relationship is traversed. Finally, **employeeEntity** is assigned the new EORelationship.

Creating a to-many relationship in the opposite direction merely swaps the source and destination attributes, and assigns the relationship to the EOEntity for the Department entity:

```
EOJoin *toEmployeesJoin;
EORelationship *toEmployeesRelationship;

toEmployeesJoin = [[[EOJoin alloc]
    initWithSourceAttribute:deptIDAttribute
    destinationAttribute:empDeptIDAttribute] autorelease];

toEmployeesRelationship = [[[EORelationship alloc] init] autorelease];
[toEmployeesRelationship setName:@"department"];
[toEmployeesRelationship addJoin:toEmployeesJoin];
[toEmployeesRelationship setToMany:YES];
[toEmployeesRelationship setJoinSemantic:EOInnerJoin];

[deptEntity addRelationship:toEmployeesRelationship];
```

Note that this relationship is to-many precisely because the destination attribute isn't the primary key for its entity (Employee), and therefore isn't unique with regard to that entity.

A relationship isn't restricted to only one EOJoin. It's entirely possible for a relationship to be defined based on two or more attributes in the source and destination entities. For example, if the employees database contains a picture of each employee identified by first and last name, you define the relationship by joining each of the first and last names in the Employee entity to the same attribute in the **EmpPhoto** attribute.

A simple relationship is considered to reference all of the attributes in its joins. You can use the **referencesProperty:** method to find out if an EORelationship references a particular attribute.

Creating a Flattened Relationship

A flattened relationship depends on several simple relationships already existing. Assuming that several do exist, creating a flattened relationship is straightforward. Suppose that, in addition to the **department** relationship used above, the Department entity has a relationship to its facility (building)—the **facility** relationship. This code excerpt creates a flattened relationship directly from Employee to each employee's facility:

```
EOEntity *employeeEntity;      /* Assume this exists. */
```

```
EORelationship *toFacilityRelationship;

toFacilityRelationship = [[EORelationship alloc] init];
[toFacilityRelationship setName:@"facility"];
[toFacilityRelationship setEntity:employeeEntity];
[employeeEntity addRelationship:toFacilityRelationship];
[toFacilityRelationship setDefinition:@"department.facility"];
```

All that's needed to establish the relationship is a data path (also called the definition) naming each component relationship connected, with the names separated by periods. Note that because the cardinality of a flattened relationship is determinable from its components, no **setToMany:** message is required here.

A simple relationship is considered to reference all of the relationships in its definition, plus every attribute referenced by the component relationships. You can use the **referencesProperty:** method to find out if an EORelationship references another relationship or attribute.

Method Types

Setting the relationship name	– beautifyName – name – setName: – validateName:
Using joins	– addJoin: – joins – joinSemantic – removeJoin: – setJoinSemantic:
Getting attributes joined on	– destinationAttributes – ownsDestination – sourceAttributes
Getting the definition	– componentRelationships – definition – setDefinition:
Getting the entities joined	– anyInverseRelationship – destinationEntity – entity – inverseRelationship – setEntity:

Checking the relationship type	– isCompound – isFlattened – isMandatory – setIsMandatory:
Setting to-many	– isToMany – setToMany:
Relationship qualifiers	– qualifierWithSourceRow:
Checking references	– referencesProperty:
Controlling batch fetches	– numberOfToManyFaultsToBatchFetch – setNumberOfToManyFaultsToBatchFetch:
Taking action upon a change	– deleteRule – propagatesPrimaryKey – setDeleteRule: – setPropagatesPrimaryKey:
Setting the user dictionary	– setUserInfo: – userInfo

Instance Methods

addJoin:

– (void)**addJoin:**(EOJoin *)*aJoin*

Adds a source-destination attribute pair to the relationship. Raises an NSInvalidArgumentException if the relationship is flattened, if either the source or destination attributes are flattened, or if either of *aJoin*'s attributes already belongs to another join of the relationship.

See also: – **joins**, – **isFlattened**, – **setDefinition:**

anyInverseRelationship

– (EORelationship *)**anyInverseRelationship**

Searches the relationship's destination entity for a user-created, back-pointing relationship joining on the same keys. If none is found, it looks for a "hidden" inverse relationship that was manufactured by the Framework. If none is found, the Enterprise Objects Framework creates a "hidden" inverse relationship and returns that. Hidden relationships are used internally by the Framework.

See also: – **inverseRelationship**

beautifyName

– (void)**beautifyName**

Makes the relationship's name conform to a standard convention. Names that conform to this style are all lower-case except for the initial letter of each embedded word other than the first, which is upper case. Thus, "NAME" becomes "name", and "FIRST_NAME" becomes "firstName".

See also: – **setName:**, – **validateName:**

componentRelationships

– (NSArray *)**componentRelationships**

Returns an array of base relationships making up a flattened relationship, or **nil** if the relationship isn't flattened.

See also: – **definition**

definition

– (NSString *)**definition**

Returns the data path of a flattened relationship; for example "department.facility". If the relationship isn't flattened, **definition** returns **nil**.

See also: – **componentRelationships**

deleteRule

– (EODeleteRule)**deleteRule**

Returns a rule that describes the action to take when an object is being deleted. The returned rule is one of the following:

EODeleteRuleNullify
EODeleteRuleCascade
EODeleteRuleDeny

For example, suppose you have a department with multiple employees. When a user tries to delete the department, your application could:

- Delete the department and remove any back pointer the employee has to the department (nullify)
- Delete the department and all of the employees it contains (cascade)
- Refuse the deletion if the department contains employees (deny)

destinationAttributes

– (NSArray *)**destinationAttributes**

Returns the destination attributes of the relationship. These correspond one-to-one with the attributes returned by **sourceAttributes**. Returns **nil** if the relationship is flattened.

See also: – **joins**, – **destinationAttribute** (EOJoin)

destinationEntity

– (EOEntity *)**destinationEntity**

Returns the relationship's destination entity, which is determined by the destination entity of its joins for a simple relationship, and by whatever ends the data path for a flattened relationship. For example, if a flattened relationship's definition is "department.facility", the destination entity is the Facility entity.

See also: – **entity**

entity

– (EOEntity *)**entity**

Returns the relationship's source entity.

See also: – **destinationEntity**, – **addRelationship:** (EOEntity)

inverseRelationship

– (EORelationship *)**inverseRelationship**

Searches the relationship's destination entity for a user-created, back-pointing relationship joining on the same keys. Returns the inverse relationship if one is found, **nil** otherwise.

See also: – **anyInverseRelationship**

isCompound

– (BOOL)**isCompound**

Returns YES if the relationship contains more than one join (that is, if it joins more than one pair of attributes), NO if it has only one join. See "Creating a Simple Relationship" in the class description for information on compound relationships.

See also: – **joins**, – **joinSemantic**

isFlattened

– (BOOL)**isFlattened**

Returns YES if the relationship traverses more than two entities, NO otherwise. See “Creating a Flattened Relationship” in the class description for an example of a flattened relationship.

isMandatory

– (BOOL)**isMandatory**

Returns YES if the target of the relationship is required, NO if it can be **nil**.

See also: – **setIsMandatory:**

isToMany

– (BOOL)**isToMany**

Returns YES if the relationship is to-many, NO if it’s to-one.

See also: – **setToMany:**

joinSemantic

– (EOJoinSemantic)**joinSemantic**

Returns the semantic used to create SQL expressions for this relationship. The returned join semantic is one of the following:

EOInnerJoin
EOFullOuterJoin
EOLeftOuterJoin
EORightOuterJoin:

See also: – **joins**

joins

– (NSArray *)**joins**

Returns all joins used by relationship.

See also: – **destinationAttributes,** – **joinSemantic**, – **sourceAttributes**

name

– (NSString *)**name**

Returns the relationship's name.

numberOfToManyFaultsToBatchFetch

– (unsigned int)**numberOfToManyFaultsToBatchFetch**

Returns the number of to-many faults that are triggered at one time.

ownsDestination

– (BOOL)**ownsDestination**

Returns YES if the relationship should insert or delete the target object when added or removed, NO otherwise.

See also: – **destinationAttributes**

propagatesPrimaryKey

– (BOOL)**propagatesPrimaryKey**

Returns YES if objects should propagate their primary key to related objects through this relationship. Objects only propagate their primary key values if the corresponding values in the destination object aren't already set.

qualifierWithSourceRow:

– (EOQualifier *)**qualifierWithSourceRow:**(NSDictionary *)*sourceRow*

Returns a qualifier that can be used to fetch the destination of the receiving relationship, given *sourceRow.*

referencesProperty:

– (BOOL)**referencesProperty:**(id)*aProperty*

Returns YES if *aProperty* is in the relationship's data path or is an attribute belonging to one of the relationship's joins; otherwise, it returns NO. See the class description for information of how relationships reference properties.

See also: – **referencesProperty:** (EOEntity)

removeJoin:

– (void)**removeJoin:**(EOJoin *)*aJoin*

Deletes *aJoin* from the relationship. Does nothing if the relationship is flattened.

See also: – **addJoin:**

setDefinition:

– (void)**setDefinition:**(NSString *)*definition*

Changes the relationship to a flattened relationship by releasing any joins and attributes (both source and destination) associated with the relationship and setting *definition* as its data path. “department.facility” is an example of a definition that could be supplied to this method.

If the relationship’s entity hasn’t been set, this method won’t work correctly. See “Creating a Flattened Relationship” in the class description for more information on flattened relationships.

See also: – **addJoin:**, – **setEntity:**

setDeleteRule:

– (void)**setDeleteRule:**(EODeleteRule)*deleteRule*

Set a rule describing the action to take when object is being deleted. *deleteRule* can be one of the following:

EODeleteRuleNullify
EODeleteRuleCascade
EODeleteRuleDeny

For more discussion of what these rules mean, see the method description for **deleteRule**.

setEntity:

– (void)**setEntity:**(EOEntity *)*anEntity*

Sets the entity of the relationship to *anEntity.* If the relationship is currently owned by a different entity, this method will remove the relationship from that entity. This method doesn’t add the relationship to the new entity. EOEntity’s **addRelationship:** method invokes this method.

You only need to use this method when creating a flattened relationship; use EOEntity’s **addRelationship:** to associate an existing relationship with an entity.

See also: – **setDefinition:**

setIsMandatory:

– (void)**setIsMandatory:**(BOOL)*flag*

Specifies according to *flag* whether the target of the relationship must be supplied or can be **nil**.

setJoinSemantic:

– (void)**setJoinSemantic:**(EOJoinSemantic)*joinSemantic*

Sets the semantic used to create SQL expressions for this relationship. *joinSemantic* should be one of the following:

EOInnerJoin
EOFullOuterJoin
EOLeftOuterJoin
EORightOuterJoin:

See also: – **addJoin:**

setName:

– (void)**setName:**(NSString *)*name*

Sets the relationship's name to *name*. Raises a verification exception if *name* is not a valid relationship name, and NSInvalidArgumentException if *name* is already in use by an attribute or another relationship in the same entity.

This method forces all objects in the model to be loaded into memory.

See also: – **beautifyName:**, – **validateName:**

setNumberOfToManyFaultsToBatchFetch:

– (void)**setNumberOfToManyFaultsToBatchFetch:**(unsigned int)*size*

Sets the number of "toMany" faults that are fired at one time to *size*.

See also: – **isToMany**

setPropagatesPrimaryKey:

– (void)**setPropagatesPrimaryKey:**(BOOL)*flag*

Specifies according to *flag* whether objects should propagate their primary key to related objects through this relationship. For example, an Employee object might propogate its primary key to an EmployeePhoto

object. Objects only propagate their primary key values if the corresponding values in the destination object aren't already set.

setToMany:

– (void)**setToMany:**(BOOL)*flag*

Sets a simple relationship as to-many according to *flag*. Raises an NSInvalidArgumentException if the receiver is flattened. See the class description for considerations in setting this flag.

See also: – **isFlattened**

setUserInfo:

– (void)**setUserInfo:**(NSDictionary *)*dictionary*

Sets the *dictionary* of auxiliary data, which your application can use for whatever it needs. *dictionary* can only contain property list data types (that is, NSDictionaries, NSStrings, NSArrays, and NSDatas).

sourceAttributes

– (NSArray *)**sourceAttributes**

Returns the source attributes of a simple (non-flattened) relationship. These correspond one-to-one with the attributes returned by **destinationAttributes**. Returns **nil** if the relationship is flattened.

See also: – **joins**, – **sourceAttribute** (EOJoin)

userInfo

– (NSDictionary *)**userInfo**

Returns a dictionary of user data. Your application can use this data for whatever it needs.

validateName:

– (NSException *)**validateName:**(NSString *)*name*

Validates *name* and returns **nil** if its a valid name, or an exception if it isn't. A name is invalid if it has zero length; starts with a character other than a letter, a number, or "@", "#", or "_"; or contains a character other than a letter, a number, "@", "#", "_", or "$".

setName: uses this method to validate its argument.