Copyright © 2000 by Novosoft Inc. All rights reserved

FL Modeling Guide

prepared by A. Rozhenko

Contents

1 Introduction
- 1.1 Scope
- 1.2 Related Documents

2 Getting Started
- 2.1 FL Basic Model
- 2.2 FL Basic Classes
- 2.3 What stereotypes are used in FL?
- 2.4 Accessors
- 2.5 Entity is presented via Interface
- 2.6 Entity is mapped to a Database Table

3 FL Model Description
- 3.1 Entity and EntityUser
- 3.2 Access Permissions
- 3.3 User Access Roles
- 3.4 Permissions to Class
- 3.5 Permissions to Objects
- 3.6 Accompanying notes on associations described

4 FL Modeling Process
- 4.1 Creation of packages
- 4.2 Creation of classes
- 4.3 Creation of attributes
- 4.4 Creation of operations
- 4.5 Coupling classes with associations
- 4.6 Association between entity and value
- 4.7 Association between entities
- 4.8 Reverse association
- 4.9 Week association

5 Code Generation
- 5.1 Generation of values
- 5.2 Generation of entities
- 5.3 Simple example
- 5.4 Distinctions between JDK1.1 and JDK1.2 models

6 Revision History

1 Introduction

1.1 Scope

We think that the industrial approach should allow to be independent on non-functional requirements to a system. In Novosoft we use OO CASE technology + framework, codenamed FL (Foundation Layer), which lets us generate a code for an application from UML models (UML is the industry standard notation for Object Oriented Analisys/Design). Currently, the FL supports generation in Java, and we apply this framework dynamically in our current projects with business applications.

The general idea consists of that we build a functional model of the system once, then use FL to generate code on a specific language or technology according to current non-functional requirements. In this case we invent independence on specific non-functional requirements. After that a special business programming is used to support operation with business objects declared in the model.

The UML modeling may be also used to specify the business logic, eg interaction between business objects. This level of modeling is also supported by FL extensions such as Zebra, but the description of the logical layer is out of the scope of this document.

The benefits of our approach are the following:

- Any developer, analyst, or customer may read and discuss problem domain related issues using the same documentation source (UML diagrams in Rose).
- The bulk code for persistence and transaction support which is simple by its nature is automatically generated now. This reduces the number of errors and increases the development speed.
- Writing of business operations and development of GUI are technically separated by the production process.
- Changes of architecture are transparent for developers and do not affect problem specific code (i.e. business operations, GUI).

We believe that such approach would dramatically reduce the development cost and application support.

1.2 Related Documents

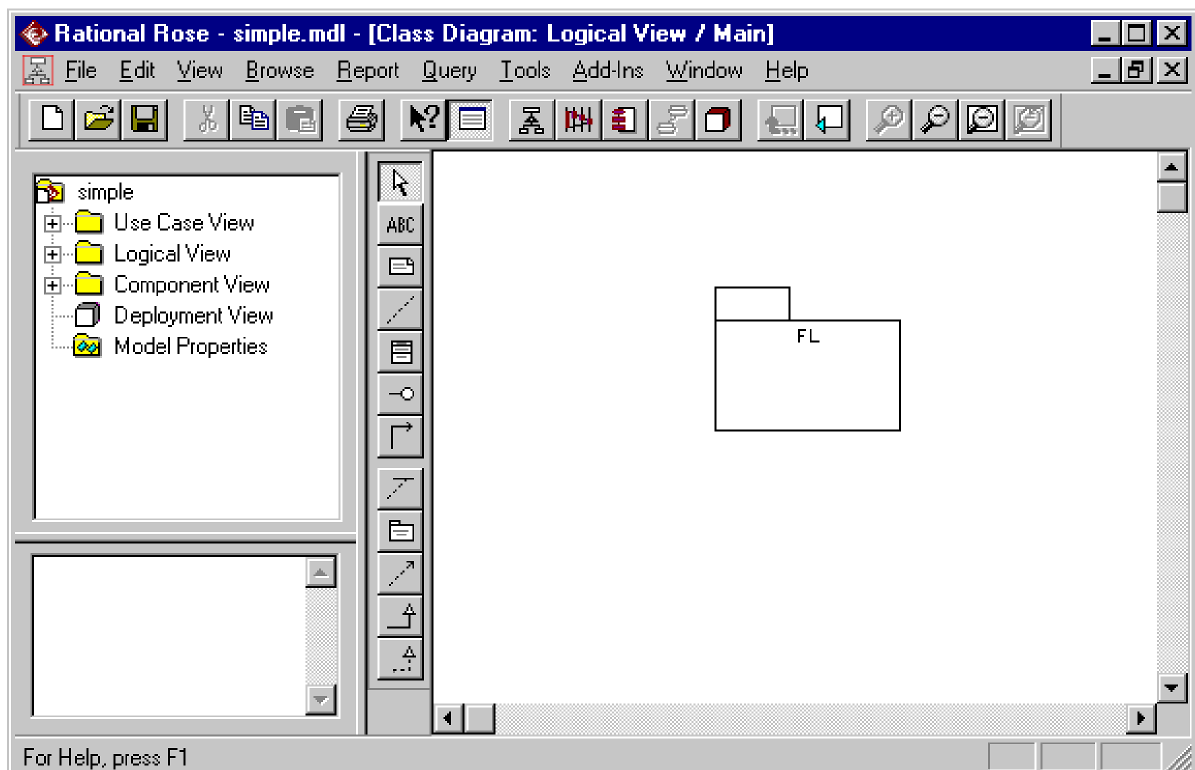
This section is under construction.

2 Getting Started

2.1 FL Basic Model

To start modeling, you should install Rose Enterprise Edition 98i modeling tool of version 6.1 or later from Rational Rose, and the Foundation Layer 4 released for JDK 1.1 or 1.2 (see FL Installation Guide).

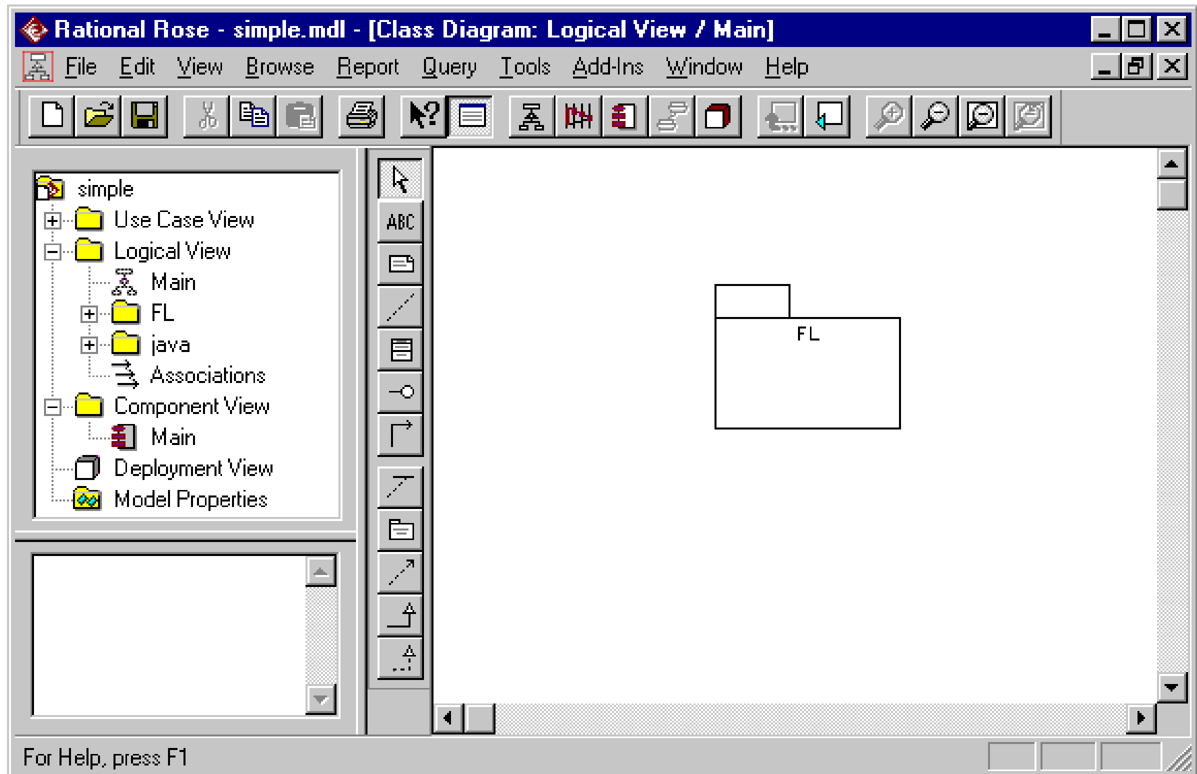
The FL distribution contains the basic FL model, named simple.mdl. Lunch it in Rose. You will see the following:



The Rose browser window contains a number of views. The main point of interest in FL modeling is the **Logical View**. You can use another views also to describe Use Cases, collaborations, component diagrams,

state/activity diagrams and so on, but only the contents of the "Logical View" is the subject of code generation (the FL modeling extensions can generate Java code from state/activity diagrams also, but this is out of the default FL generation scope).

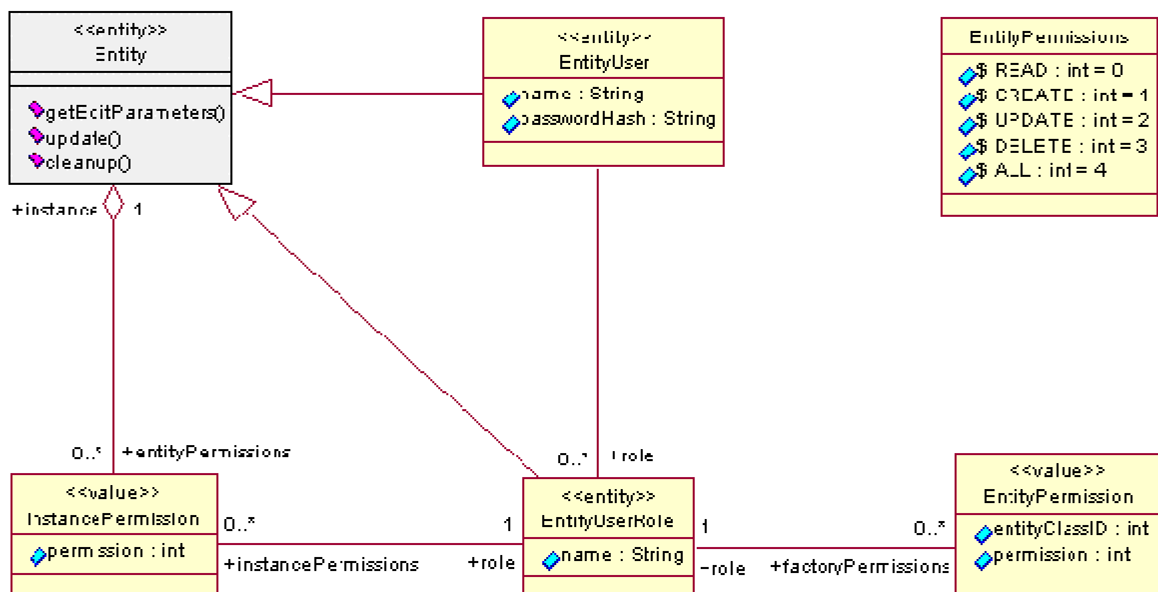
Let us open now the Logical View:



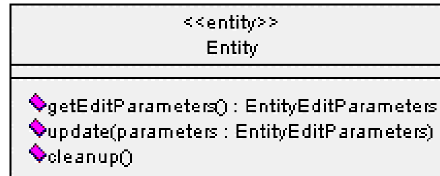
It contains two packages, FL and java, represented as folders in the browser window. In the drawing window (to the right), the **main** diagram from the Logical View is shown. It consists of the FL package only.

2.2 FL Basic Classes

To open the diagram describing the Foundation Layer basic classes, do double click on the FL package in the right window, or open FL folder in the browser window and do double click on the **Foundation Layer** diagram. You will see the following in the drawing window:

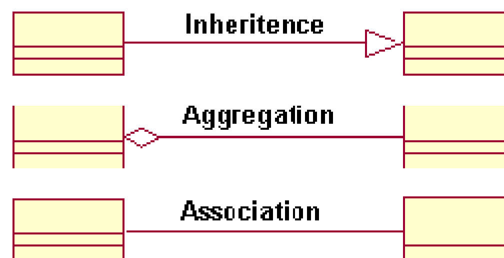


Let us describe this diagram. Classes are shown as rectangles consisting of three parts delimited by horizontal lines. The top part contains a stereotype, <<entity>> or <<value>>, and a class name. The medium part contains class attributes, eg encapsulated data in the form name : type. The bottom part contains available class operations in brief notation: name(). To look in operations signature, do right button mouse click on a class you are interested in, then select the "Options" item in popup menu and click on the "Show Operations Signature" item. For example, proceeding this action with the Entity class you will see the following:



As you can see, the getEditParameters operation returns a value of the EntityEditParameters type. Another operations don't return a value.

Classes on the diagram are linked with relationships. Three types of relationships are shown here:



In the "inheritance" relationship, the arrow points to superclass. In the "aggregation" relationship, the diamond points to the class which aggregates another class. An "association" relationship may be both navigable (with thin arrow on one side) and non-navigable (without arrows).

The FL generator doesn't distinguish aggregations and associations. The navigation arrows are also ignored on generation. So, the choice between aggregation or association, between navigable or non-navigable relationship is up to you. **We recommend to use associations only, because aggregation and composition are reserved for future FL versions.**

An association usually has a **role name** and a **multiplicity** on at least one its side. It is very important to understand what this means, because a role name and multiplicity directly influences to generated code for classes having such a relationship. See Section 4.5 fore more details.

2.3 What stereotypes are used in FL?

The use of stereotypes comes to FL from the version 4.0. The stereotype is the most essential merit of a class which joins it with another similar classes, eg having the same stereotype. We use a stereotype name when we want to point out some common characteristics for all classes having this stereotype. The following stereotypes are used in FL modeling now: entity, value, exception, primitive, or unspecified stereotype.

1. The **entity** is the main concept of the FL. An entity instance is always the **persistent object** with a life cycle longer than a session. The protocol for transferring the state of the entity between its instance and an underlying database, container, or something else is referred to as **object persistence**. The FL usually supports the persistence of entities over a database. It supports all the life cycle operations for persistent objects (create, delete, find, update). To provide integrity of data, the FL allows **transactions**.

An entity can have both attributes and operations.

2. The **value** stereotype is used for another classes which should be generated. A value instance is the **transient object**. However, the value may also be persistent object if it has an association with an entity. In this case, it is aggregated into the entity if the association has a simple multiplicity or it may stand apart of the entity if the multiplicity is complex ("zero or more").

A value can have attributes only. Operations are not supported yet by the current version of the FL generator.

3. The **exception** stereotype is used to describe an exception. The exception is not generated to Java code by the FL generator. However, if an operation throws an exception, the exception class must be described with the exception stereotype.

4. Another stereotypes are supported with plagins to the FL generator. For example, to support Java primitive types, the FL generator uses a plagin which understands the **primitive** stereotype. The collection of primitive types is frozen. You can not create classes with this stereotype.

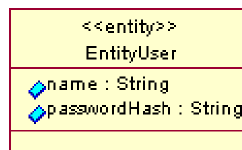
5. Classes imported from common use Java packages have no stereotype. They are only used to provide the close description of the model. They are never generated to Java by the FL generator.

2.4 Accessors

When Java code is generated, every attribute of a model class is mapped to the Java attribute with **private access**, and two special operations, getter and setter, are added to provide access to this attribute. The getter allows to get the attribute value and the setter allows to set a new value to the attribute. We call this special operations as **accessors** to distinct them from operations explicitly described in a model class.

The FL generator composes accessor's names by the following rule: the prefix "get" or "set" + the attribute name with the first letter capitalized. If an attribute is of boolean type, the generator uses "is" as a prefix instead of "get".

Let us demonstrate this on the EntityUser class from the FL model:



Its interface contains the following accessors:

```
public String getName();
public void   setName(String name);
public String getPasswordHash();
public void   setPasswordHash(String passwordHash);
```

It is the important distinction in possible access levels to class attributes for entities and values:

- entity's attributes in the model should always be public, but
- value's attributes may have any access level (public, protected, or private).

An attribute's access level in the model really describes the access level for attribute's accessors in the code generated. So, using protected access to a value's attribute, you really describe protected accessors to this attribute, but the attribute itself will always have private access.

2.5 Entity is presented via Interface

The FL generator produces 5 classes and 2 interfaces from a model class marked with entity stereotype. The upper level interface for an entity has just the same name as the corresponding model class. This interface contains prototypes for all operations described in the model class and prototypes for accessors to class attributes (two accessors per attribute). Note that a Java interface may contain public operations only. This is the only reason for entity attributes to be public too.

A value stereotyped model class is mapped to a unique Java class having just the same name as its model prototype.

2.6 Entity is mapped to a Database Table

The model is mapped by FL generator to Java code and, finally, to table names for storing entities in a database. Every entity stereotyped class is mapped to a unique table in the underlying database. Value stereotyped classes may be also mapped into separate tables in the database. Many well-known databases has serious restrictions in naming their tables, such as limited name length or case insensitivity. So, a conversion of class names in the model to short names used in a database should be done. The conversion rules must be stable and should be never changed for existing classes when the model is changed.

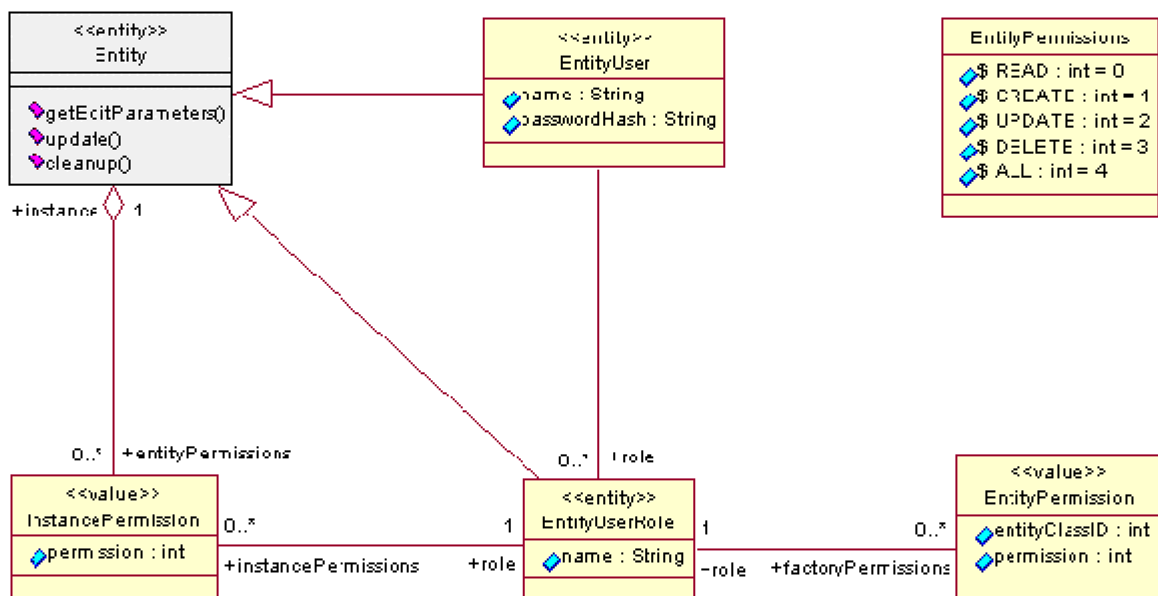
To provide the conversion stability, a special file, FL.atr, is distributed with the FL. The file contains conversion rules for basic FL classes. When you generate Java code from a model, you should inform the FL generator what attribute file you use. It reads conversion rules from the file and adds to it new conversion rules for new classes added on modeling phase. It never removes unused conversion rules. So, the attribute file may only grow up from one generation to another.

It is a good style to copy the FL.atr file to the *modelName*.atr file and generate code using the last file.

3 FL Model Description

3.1 Entity and EntityUser

Let us return once more to the Foundation Layer's diagram:

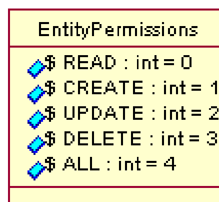


The Entity class is the fundamental FL model class. All classes with the entity stereotype should extend the Entity class directly or via another entities.

The EntityUser class describes a database user. A user can have different access permissions to database objects. The access level is defined through roles play by a user. A user may have many roles at a time. In this case, the access permission to concrete database object is defined as the most wider access given by all roles played.

3.2 Access Permissions

The possible access permissions to entities are described by EntityPermissions class:

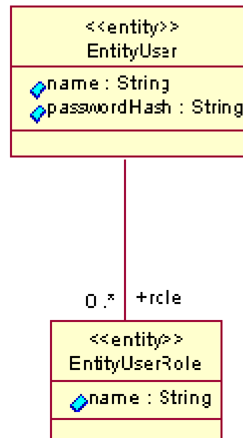


This class has a number of static final values shown. Note that it is never generated because it has no stereotype. It is implemented in Java as public interface only having the constants described here.

The access permissions to entities are taken into account for authorized access only. If the authorization is turned off, a user has all access permissions to all database entities.

3.3 User Access Roles

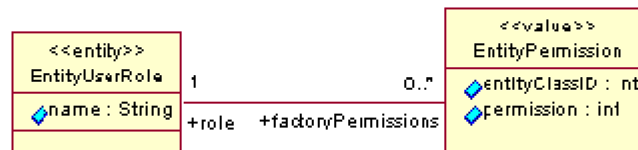
You can see 0..* multiplicity near the association between EntityUser and EntityUserRole classes:



This means that the EntityUser has access to a list of EntityUserRole instances describing the roles played. The +role near the association side says that this association is mapped to EntityEser class as the attribute with the role name. Two accessors, getRole and setRole, appear in the EntityUser interface if this class is generated. They allow access to the list of user's roles, eg an instance returned by getRole() is a list of objects having the EntityUserRole interface. The described association is of "one to many" type (one EntityUser is related with many EntityUserRole instances). Note that EntityUserRole knows nothing on EntityUser which it is related with, because an association between them has no name on the EntityUser's side. As a result, **many users can share the same role**.

3.4 Permissions to Class

Every role can have a list of EntityPermission instances. This is shown by the association between EntityUserRole and EntityPermission:

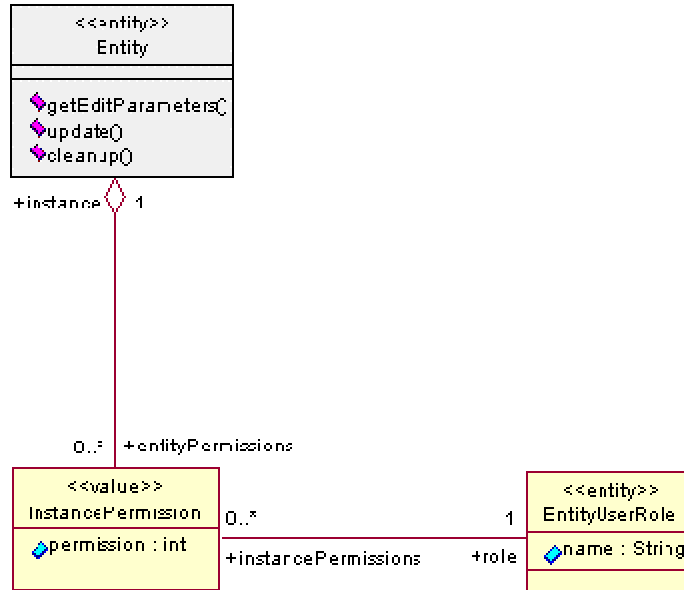


The multiplicities in this association shows that it is also of "one to many" type (one EntityUserRole can have many EntityUserPermission values). Moreover, both its sides have names. This leads to adding attributes to both linked classes: the role attribute is added to EntityPermission and the factoryPermissions attribute is added to EntityUserRole. Corresponding accessor are also created if these classes are generated by the FL generator. As a result, an EntityPermission instance is linked with one EntityUserRole instance, eg all entity permissions are individual values for roles. **One entity permission instance is newer shared by many roles.**

An EntityPermission instance describes a permission to concrete **Class Factory**. Note that every entity stereotyped class has the corresponding Java factory class (it is generated together with the class). If a role has an EntityPermission of some type to a class factory, the user having this role has such a permission to all instances of this class. So, the EntityPermission class is responsible for the **access to all instances of a concrete entity type**.

3.5 Permissions to Objects

A role can also have individual permissions for access to some entity instances. This is shown by associations between the InstancePermission class from one side and Entity and EntityUserRole class from another side:



Many InstancePermission objects may be linked with a concrete Entity instance. Every InstancePermission contains a link to a connected role and the permission delegated to this role. A role also contains a list of its individual permissions. These connections are shown by the association between the InstancePermission and EntityUserRole. In fact, the EntityUserRole and Entity are related with "many to many" association through intermediate InstancePermission objects.

3.6 Accompanying notes on associations described

Make some notes before the end of this section.

1. The FL uses the following rule when converts associations to class attributes: a role and a multiplicity on one association's side influences to the generation of a class on the opposite association's side.
2. The role is converted to class attribute with the name specified, and two accessors are added in the class interface. The multiplicity influences on the body of getters and setters used for access to related instance(s).
3. Usually, a role name and multiplicity are specified together. If both a role and multiplicity are not specified on the association side, the association is considered as unidirectional and means "one to one" or "one to more" according to a multiplicity specified on the opposite association side.
4. If an association has no roles at all, it is ignored by the FL generator. At least one role name and multiplicity must be specified for an association. See Section 4.5 for more details.

4 FL Modeling Process

The development process consists of many dependent jobs. We especially mark the following jobs:

Analysis

is the development of a conceptual model reflecting the customer's problem. It is done by an Analyst and may be described by Analysis Model in the Use Case View. The description of analysis process is out of this document scope.

Design

is the development of a logical model reflecting design solutions selected. It is done by a Designer, and its result is the description of packages, classes and relationships between classes in the Logical View.

Programming

is the reflection of the Design Model to code. We distinguish the following programming levels:

- Business Programming is the implementation of all operations described in the Design Model;
- Logical Programming is the implementation of logical interaction between entities. It may be described in Rose by State/Activity and Collaboration diagrams.
- User Interface Programming is the implementation of the interface with a user.

The FL Modeling Process is the preparation of a Design Model based on the Foundation Layer's conceptual model. It has many intersections with the FL Business Programming, because the Design Model is mapped to the bulk of programming code by the FL generator. Note that the special extension of the FL generator, FL-Zebra generator, also supports the mapping of logics described in State/Activity Diagrams to programming code, but the description of FL-Zebra Modeling is out of this document scope.

The FL Modeling Process include the following actions in Rose: creation of new packages, classes, class attributes and operations, and relationships between classes.

4.1 Creation of packages

To create a new package (or subpackage), do right button click on the "Logical View" (or upper package's name) in the browser window and select "New ® Package" in popup window. Another way is to open the Logical View's (or package's) main diagram, select the "Package" instrument, and click on the drawing window.

When a (sub)package is created, open its specification and select the "ngen" bookmark. You can see there the following properties:

ngen.generate
: is the generation flag. If it is set to "false", the package's classes are never generated. The default is "true". This means the permission to generate package's classes as Java code. The decision to generate a concrete class depends on "ngen.generate" flag of the class specification and on the class stereotype. The entity and value are stereotypes for generation. Another stereotypes (and unspecified also) forbid the class generation.

ngen.defaultPackage
: is set to "true" for java.lang package only. It controls the appearance of class names in Java code. The "true" value means the insertion of brief class name, and "false" value means full class name insertion.

ngen.package
: describes the package name. If the property is empty, the class name is generated from packages's structure in the model. **We highly recommend to set the package name explicitly.**

If you do double click on a package icon in the drawing window, the main class diagram for the package will be created.

4.2 Creation of classes

To create a new class in the package, do right button click on the package's name in the browser window and select "New ® Class" in popup window. Another way is to open a package's class diagram, select the "Class" instrument, and click on the drawing window.

Then you should set the class stereotype. Open the class specification and select a stereotype in the "General" bookmark. The following stereotypes are essential for the FL generator: entity, value, and exception. You need not specify a stereotype if this is non-exception type class and it is not generated.

After that, you must define the inheritance rules:

- If the stereotype is "entity", the class should inherit from another entity-class;
- If the stereotype is "value", the class may inherit from another value-class;
- In other cases the inheritance has no sense.

To import a class from another package (for example, Entity from the FL package) select it in the browser window and drag it by the mouse to the diagram for import to.

Finally, you may open the "ngen" bookmark in class specification and modify the ngen.generate property for the class. Note that the ngen.final property is not now used by the FL generator.

4.3 Creation of attributes

To add a new attribute to a class, do right button click on its icon in the drawing window and select "New attribute". Then edit the attribute name, write the colon ":" after it, and specify the attribute type. The current version of the FL generator permits the following attribute types:

- Java primitive types: boolean, char, byte, short, int, long, float, and double;
- String type; and
- Date type.

If you select the String type, you must set its maximum length. To do this, open the "Attributes" bookmark in the class specification, then do double click on the attribute name, then open the "ngen" bookmark, and correct the flgen.length property value. You can also correct the flgen.columnType property by setting the CHAR or VARCHAR value.

After that you must change the attribute's export control. As it was mentioned in Section 2.4, all entity's attributes must be public. For value's attributes, the protected control may be also used. **We recommend to set the public export control for all attributes.**

4.4 Creation of operations

To add a new operation to a class, do right button click on its icon in the drawing window and select "New operation". Then edit the operation name, the operation signature, and the return value. The template for operation description is the following:

name ([*par*:*type*, ...]) [:*type*]

Here, the square brackets mean optional fields and the suspension points mean the repetition. If the return value (typed after the close brace) is omitted, the operation will return no value (eg will have the void type). You can use arbitrary types in the operation signature and return value, but you should remember that **all used types MUST present in the model**.

To describe exceptions thrown by an operation, open the "Operations" bookmark in the class specification, then do double click on the operation name, then open the "Detail" bookmark, and fill out the "Exceptions" field with a list of exception classes delimited by commas. Note that all used exception classes should be described in the model with the exception stereotype.

4.5 Coupling classes with associations

If you couple two classes with an association, then at least one class should have the entity stereotype. Another class may have both entity and value stereotypes.

An association has two sides. You can specify a role name and multiplicity at any side of the association. But an association between entity and value should have a role and multiplicity at one its side only. Associations between entities may be more complex.

A role name is mapped to attribute of class linked to opposite association side. A multiplicity influences to generated attribute type. We distinguish the following multiplicities:

Simple multiplicity
is defined as 1 or 0..1 (it is no difference between these notations).
List multiplicity
is defined as 0..* or 1..* (it is no difference also).

The simple multiplicity is converted to the attribute of the class type closed to the multiplicity. In the following example, we omit stereotypes to simplify the notations:

This relationship is implemented in Class1 as an attribute

```
private Class2 class2;
```
and the following pair of accessors is added to its interface:
```
public Class2 getClass2();
public void setClass2(Class2 value);
```

The list multiplicity is converted to List type in assumption that every list item has instances of the class type closed to the multiplicity. An example:

This relationship is implemented in Class1 as an attribute

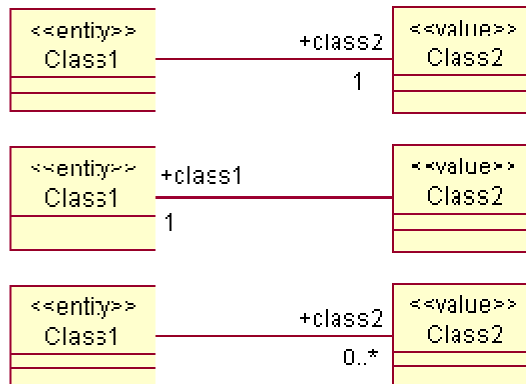
```
private List class2;
```
and the following pair of accessors is added to its interface:

```
public List getClass2();
public void setClass2(List values);
```

More detail description of possible associations is done below.

4.6 Association between entity and value

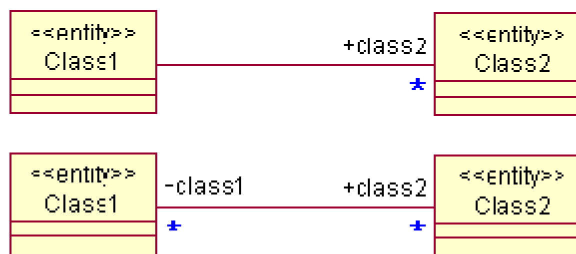
The following associations between entity and value classes are possible:



The first two associations are simple and the last association is converted to a list.

4.7 Association between entities

We describe ordinary associations here. A special "reverse" association is described in the next section. The following ordinary associations between entity and value classes are possible:



The blue asterisk means here a multiplicity of a simple or list type. So, the FL generator supports associations of "one to one", "one to many", and "many to many" types.

4.8 Reverse association

The association of "one to many" type has low performance when a list of related objects is too large. There exists another way to describe such an association and work with it. We call here an object having multiplicity "one" as **parent** object, and objects in its list as **child** objects.

The performance improvement for such an association is achieved by reversing relationships between the parent and its children. In ordinary case, the parent object "knows all its children" (has a list of children). In reverse case, every child "knows its parent", but the parent "knows nothing on his children" . The reverse association is described by the following pattern:

The blue asterisk means here a multiplicity of a simple or list type. This pattern is more general than "one to many" relationship. It also supports the "many to many" relationship when a child can have many parents. **The reverse association should be between entities only.**

The reverse association is processed by the same manner as an ordinary association (attribute and accessors are added to the Child class). In addition, and a special SQL request is added to SQL resources of the Child class to support the reverse technique.

When we want to generate a list of children for a concrete parent, we call a special findReverse operation of the object manager:

```
List children = objManager.findReverse(parent, Child.class, "parent");
```

Here parent is a parent instance which children we search, Child.class is a class where we search, and "parent" is a role name which we test to contain a reference to the parent.

Another overloaded findReverse method are also possible:

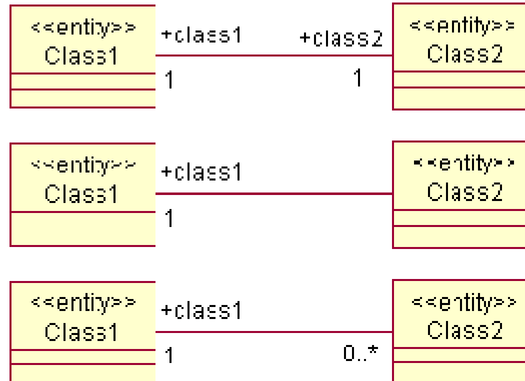
```
List children = objManager.findReverse(parent, Child.class, "parent", n);
List children = objManager.findReverse(parent, Child.class, "parent", ID);
List children = objManager.findReverse(parent, Child.class, "parent", ID, n);
```

Here n is a quantity of children to load, ID is a child ID which we search from. Note that the child with specified ID is not included into the search, eg the operation extracts children going after this ID.

4.9 Week association

Sometimes the relationship between a parent and its children is "week". This means that a child never modifies the parent object. An example is a relationship between a folder and messages included into. When we add, update, or delete a message, the folder will be locked until completing this action. As a result, parallel transactions with another messages of this folder will be rolled back. To avoid rolling back, we should declare that the association between the folder and its messages is weak, eg the same folder may be shared by many transactions at a time. The pattern for this relationship is the following:

To declare this association as weak, open the "ngen" bookmark in association's specification and set the flgen.weekReference property to "true". The weakness property is acceptable for associations having the simple multiplicities at named sides. (This is the restriction of current FL version. It may be removed in future versions.) So you can declare the following associations as week:



If you want to compose a two-sided association having only one week side, you should use two one-sided associations instead and declare one of them as week.

5 Code Generation

The flgen.cmd file in the FL distribution processes Java Code Generation from a Rose model. To start it, you should set the environment variables FL_HOME and JAVA_HOME to the FL and JDK home directories respectively. This command has the following syntax:

flgen.cmd *petalModel genDir attrFile*

Here the *petalModel* is a model prepared in Rose, *genDir* is the directory to store generated code (usually "."), and *attrFile* is a file containing model attributes needed for generation. The default attribute's file, FL.atr, is distributed together with the basic FL model.

The generator creates Java files and auxiliary files and stores them in the subdirectories of the *genDir* directory according to packages structure in the model. It also updates the attribute's file. So, we recommend to create the local copy of the default attribute's file and do the subsequent generations with it.

5.1 Generation of values

Every value class from the model is mapped to a unique Java class having just the same name as its model prototype. It is stored in the package the model prototype belongs to. You don't need any programming for such a class.

5.2 Generation of entities

Every entity class from the model is mapped to a number of classes and interfaces. Only two of them are essential for developers: the interface for the model class and operations' interface:

- The interface for the model class has just the same name as the model class, and it is stored in the package the model class belongs to. It contains operations described in the model class and accessors to class data and associations.
- The interface for the operations is stored in the Operations subpackage. Its name is composed from the model class name and the Operations suffix. This interface contains operations described in the model class. Their signature differs from model prototypes: the parameter containing the class instance is added to the beginning of a parameters' list.

The only thing you should program is the implementation of the operations' interface in the class having the name composed from the model class name and the suffix OperationsImpl. This class should be stored together with the operations' interface, eg in the Operations subdirectory.

5.3 Simple example

For example, let us consider a simple Folder/Message model, named testfolder.mdl, having a reverse association between folders and their entries:

Let the package name in the model is set to test.folder, and flgen.cmd is started with testfolder.atr file which is a simple copy of the FL.atr file:

flgen.cmd testfolder.mdl . testfolder.atr

The following interfaces are generated (we use Unix like notation for directories):

./test/folder	**./test/folder/operations**
Entry.java	EntryOperations.java
Folder.java	FolderOperations.java
Message.java	MessageOperations.java

The generated code is the following (we remove comments and import directives):

The Entry.java file:

```
package test.folder;
import ...
public interface Entry
       extends com.novosoft.fl.Entity
{
  public String getTitle();
  public void setTitle(String arg);
  public Folder getFolder();
  public void setFolder(Folder arg);
}
```

The EntryOperations.java file:

```
package test.folder.operations;
import ...
public interface EntryOperations
       extends com.novosoft.fl.operations.EntityOperations
{
}
```

The Folder.java file:

```
package test.folder;
import ...
public interface Folder
       extends test.folder.Entry
{
  public void composeMessage(String title, String body);
  public void addSubfolder(String title);
}
```

The FolderOperations.java file:

```
package test.folder.operations;
import ...
public interface FolderOperations
       extends test.folder.operations.EntryOperations
{
  public void composeMessage(Folder object, String title, String body);
  public void addSubfolder(Folder object, String title);
}
```

The Message.java file:

```
package test.folder;
import ...
public interface Message
       extends test.folder.Entry
{
  public String getBody();
  public void setBody(String arg);
}
```

The MessageOperations.java file:

```
package test.folder.operations;
import ...
public interface MessageOperations
       extends test.folder.operations.EntryOperations
{
}
```

To implement described operations, you should create the following files in "./test/folder/operations" subdirectory:

EntryOperationsImpl.java

FolderOperationsImpl.java

MessageOperationsImpl.java

See FL Developer's Guide fore more details.

5.4 Distinctions between JDK1.1 and JDK1.2 models

The only distinction in models prepared for use with JDK1.1 and JDK1.2 is the package name set by the ngen.package property for the java.util.collections model package:

- For JDK1.1, ngen.package = com.sun.java.util.collections
- For JDK1.2, ngen.package = java.util

So, you can easy modify the model for use with another JDK simply correcting this property.

6 Revision History

March 30, 2000
Initial version.

File translated from TEX by TTH, version 2.25.
On 31 Mar 2000, 17:06.