Java program generation

Introduction

Genova's generators may be used to generate a complete Java application with GUI, application and data access code. Four different code generators are involved.

- Generation of java domain classes from the class model in Rose.
- Generation of java GUI client code based on Swing from dialog models. This is done with Dialog Designer with Java/Jfc as target.
- Generation of database service code with support for complex database operations from object selections. This is done with Service Designer with Java as target.
- Data access code supported by Object to Relational mappings for Hibernate. The descriptions are generated by Database Designer, data access generation with Hibernate as target.

The full application system consists of the generated code together with a runtime framework of java classes distributed with Genova, genova-runtime.jar. The jar file contains both class files and java source.

The domain model

See Genova810DokDomenemodell.doc

The Genova models

The two code generators, Client Generator and Server Generator, both uses models developed in Genova. Client Generator uses the dialog model when generating the GUI code and it uses the underlying object selection when generating code for manipulating data in the GUI. Server Generator uses only the object selections as bases for the code it generate.

Object Selections

Object selections represents a subset of the class model and are used both as a basis for creating dialogs, generating client code and for generating service code. Object selections are described in the Dialog Designer manual, chapter 5 "Object Selections".

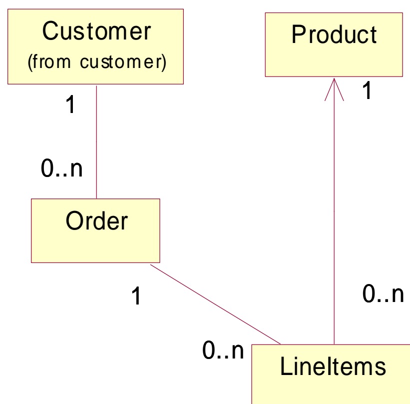

Figure 1: An UML-model with four associated classes

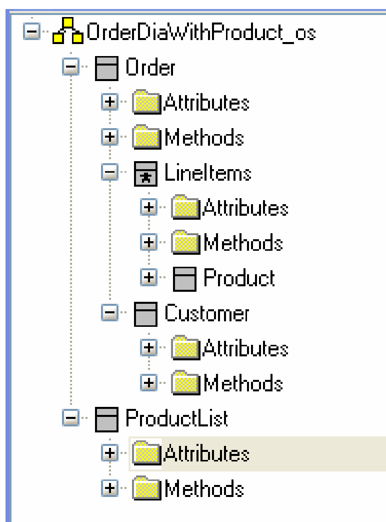

Figure 2: An object selection based on the UML-model in figure 1

When defining an object selection one first choose a root class. From the root class one adds one or more classes associated with the root class. Figure 1 shows a class-model with four associated classes and figure 2 shows an object selection containing these four classes.

The object selection in figure 2 has two roots, Order and ProductList. The two nodes Product and ProductList are both nodes representing the class Product.
The following definitions will be used throughout this manual.

Owner and Member roles for an association. In an association between two classes, one class represents the owner of the association while the other represents the member. In a one to many association the one side always represent the owner while the many side represent the member. In figure 1 Customer is the owner and Order is the member in the association between the two classes.

Downward-related role: In an object selection downward-related roles are the classes found when following associations in the owner-member direction. In figure 2 LineItems is downward-related from Order.

Upward-related role: In an object selection upward-related roles are the classes found when following associations in the member-owner direction. In figure 2 Customer is upward-related from Order and Product is upward-related from LineItems,

Main tree: A main tree is defined as the classes found when recursively following downward-related associations from a root node in the object selection. An object selection may have one or more root nodes and therefore consists of one or more main trees. An object selection may then be described as one or more main trees with upward-related sub-trees. In figure 2 the first main tree consists of the root node Order and the downward-related node LineItems. This main tree has two upward related sub-trees, Customer and Order. The second main tree consists of the single node productList.

Target role: All actions supported by the server code have a target role. Any role in an object selection may be the target role. In figure 2, if Product is the target, the path to target contains the two nodes LineItems and Order.

Path to target: The path to target is the list of nodes in an object selection from the target role to the root role. The target role itself is not a part of path to target, i.e. if the target role is a root in the object selection, the path to target is empty.

Main keys: The object selections must contain a complete unique key (main key) for each class/role. This means that if the main key in class X is a group containing a related attribute from class Y, then class Y must also be included in the object selection. The main key is by default the attribute or group specified as the primary key for the class. If the entire primary key is not present then the first unique attribute or group present is regarded as the main key. When searching for the first unique key Genova will first search true all attributes in a class, then all of the groups. If the class is an inherited class the attributes from the super class is taken before those from the subclass.
Note: Client Generator and Service Generator use the same strategy when picking the main key. However, Client Generator will only choose between attributes/groups that are contained in the dialog. The two may therefore disagree about the main key, but the client code will always send the name of the key the service should use.

Service Generator

Service Generator generates code for the service part of an application. The code generated is based on the Object Selections together with the specification given in the Service Designer setup for the wanted target. The only target available at the moment is Java.

The Java code generated by Service Generator is intended to be used together with code generated by Client Generator with Java/Jfc as target, but may also be used from any self

written code. The service code uses an implementation of the interface no.genova.service.DataService to access the database. The runtime library comes with an implementation based on Hibernate.

Code interface

Service Generator generates a set of methods for each node in the object selection. This set consists of methods for standard database operations. The methods are defined in the interface no.genova.service.JGrapeService. The following methods are defined in the interface:

```
Object find(ObjectSelection objectSelection, ClientContext ctx)
List findAll(ObjectSelection objectSelection, ClientContext ctx)
Object save(ObjectSelection objectSelection, ClientContext ctx)
Object insert(ObjectSelection objectSelection, ClientContext ctx)
Object update(ObjectSelection objectSelection, ClientContext ctx)
Object delete(ObjectSelection objectSelection, ClientContext ctx)
Object connect(ObjectSelection objectSelection, ClientContext ctx)
Object disconnect(ObjectSelection objectSelection, ClientContext ctx)
```

The methods all take two parameters, an ObjectSelection containing the data to work on and a ClientContext describing the user context.

Except for findAll all the other methods return an object that may either be a List of root objects or a single root object. The client code should always test if the result object is a List or not. The findAll method will always return a list of objects searched for.

ClientContext

ClientContext may contain the following data

String userId	String identifying current user.
String applicationName	String identifying current application
String databaseContext	String identifying current database context
boolean debug	Flag telling if running in debug mode.
Map params	Map containing additional parameters.

With the exception of databaseContext, the generated service code does not use the client context. Any content and usage is left to the application programmers. Usage of databaseContex is described in XXXX.

ObjectSelection

Object Selection is used to specify the data details needed by the service code to perform the service action wanted by the client. The class contains the following data

String currentRootName	Name of the root node in the object selection
String targetRoleName	Name of the target role node in the object selection.

	The target role is a node in the tree started from current root.
String targetMainKey	Name of the identifying key for the target role. If it is not specified, the primary key for the class will be used.
Map roots	Map containing pointers to root nodes identified by rootName. Will at least contain a pointer to the data representing current root, but data for additional roots may be added.
String objectSelectionName	Name of the object selection
String applicationName	String identifying current service application.
FindData findData	Details for Find and FindAll actions. The findData is not generated by the client code. This data may be added by the programmer to extent the search possibilities. See further description of FindData in XXXX

Action supported by generated code

For each node in an Object Selection, Service Generator supports the following server side actions:

- Find
- FindAll
- Save
- Insert
- Update
- Delete

All of these actions take an Object Selection role as target and the generated code will do the action for the target role and all roles below the target role in the Object Selection. All actions operate on a single occurrence of the target role, even if the role itself is connected into the Object Selection via a many-association.

Find action

The Find action will identify one occurrence of the target role based on the value of the main key. This should always identify a single occurrence of the target role. The target should be one of the root roles in the object selection. If a target object is found all objects related to the target object will be returned too.

FindAll action

The FindAll action will identify all occurrences of the target role. The target should either be one of the root roles or one of the upward-related roles in the object selection. For each object found upward-related objects are also retrieved. The downward-related objects are not retrieved.

Save/Insert/Update actions

These three actions all do an update of the target role, but with the following restrictions:

- Insert: This action will always try to insert a new occurrence into the database. An exception is thrown if the object already exists in the database.
- Update: This action will always try to update an existing occurrence of the object. An exception is thrown if the object does not exist in the database.
- Save: This action will either do an insert, if the object does not exist, or an update, if the object already exists.

All three actions will not only update the target object, but also all objects in the sub-tree having the target object as root. In addition these actions will update all objects in the path from the target and recursively all upward-related objects to the objects in path to target.

When a save/insert/update-action is updating any other object than the target object, the action performed on the object depends on the object's legal function specification in the object selection. See chapter 5.10, Legal functions, in "Dialog Designer manual" on how to edit these properties. The generated code uses the Insert and Update specification for the role:

Insert == true and Update == true
The generated code will do a save action, that is either insert a new of update an existing occurrence.
Insert == true and Update == false
The generated code will do an insert action. If the object already exists an exception is thrown.
Insert == false and Update == true
The generated code will do an update action. If the object does not exist an exception is thrown.
Insert == false and Update == false
The generated code will only try to identify the object. If the object does not exist an exception is thrown.

The save/insert/update action is always done on a single occurrence of the target role, even if this role is member of a one-to-many association in the object selection

Delete action

This action will delete an occurrence of the target object from the database. The delete action will either delete or disconnect related objects from the one deleted. A related object is deleted if legal function specifies that delete is allowed. If delete is not allowed, but disconnect is allowed the action will only disconnect the related object from the one deleted. If neither delete nor disconnect is allowed the action will throw an exception.

Generated code

For an Object Selection, Server Generator generates two java classes for each node. If the node has role name roleName, the two classes are named roleNameManager and roleNameDefaultManager. DefaultManager contains all generated code while the Manager is an initially empty class inheriting from DefaultManager. DefaultManager is regenerated each time Server Generator is run, while Manger contains your own code and is never regenerated. All customizing of the generated code should be done in the Manager class.

Service Generator does not generate code for nodes with the Persistent property set to False. Nodes in an object selection based on transient classes will always have the Persistent property set to false. For nodes based on persistent classes from the class model the default value of the property is True, but it may be changed to False. If a node has property Persistent False no code is generated for that node and for any nodes related to the node.

To prevent Service Generator from generating code for such nodes the property should be set to False.

The DefaultManager class for a object selection role will contain one entry method for each of the six server actions find, findAll, save, insert, update and delete.

When Service Generator generates code the methods generated for each role depends on the roles place in the object selection. The following list shows which methods that may be generated for each node. A method is only generated if needed, i.e. findDownRelated is only generated if the node has at least one downward-related node in the object selection, findAsRelated is not generated for root nodes etc.

delete
deleteSingle
deleteRelated
deleteAsRelated
findAll
find
findSingle
findUpRelated
findDownRelated
findAsRelated
identifySingle
identifyAsRelated
insert
insertSingle
insertAsRelated
save
saveSingle
saveUpRelated

saveDownRelated
saveAsRelated
saveAsParent
saveUpRelatedAsParent
update
updateSingle
updateAsRelated

Customizing the generated code

Writing your own code and merging it with the generated code can be done in two different ways, either by overriding one or more of the generated methods in DefaultManager or by writing content to hook methods called from the generated code.

Using hook methods

The hook methods are called from the actual code in Default Manger. All hook methods are implemented in DefaultManager as methods doing nothing. The hooks are intended as action to be performed before and after the default implementation of actions. Three different types of hooks exist:

Boolean hooks: The default method returns "true". Returning "false" from such a hook tells the generated code to skip the rest of the implementation.

Object hooks: These types of hooks return a domain class. The default method returns null. Returning an Object from such a hook tells the generated code that the hook did the job and that the returned object is the result.

Void hooks. No return value, the generated code will always continue after the hook call.

In addition to the normal return, the hooks may throw an exception and by that stopping the ongoing action.

Example:

```
Artist saveSingle(Artist targetObject,
                  String targetKey,
                  FindData findData,
                  ClientContext ctx,
                  Session session) {
    Artist resultObject = null;
    resultObject = hookSaveSingleStart(
                       targetObject,targetKey,
                       findData,ctx,session);
    if (resultObject == null) {
        resultObject = findSingle(
                targetObject,targetKey,
                findData,ctx,session);
        if (resultObject != null) {
            resultObject = updateSingle(
                    targetObject,targetKey,
                    findData,ctx,session);
```

```
            }
            else {
                resultObject = insertSingle(
                        targetObject,targetKey,
                        findData,ctx,session);
             }
        }
        hookSaveSingleEnd(resultObject,ctx,session);
        return resultObject;
}
```

This example contains calls to two different hooks, hookSaveSingleStart and hookSaveSingelEnd. If one want to do the actually save oneself, this may be done in the start hook. Returning an object from that hook will prevent the saveSingle method from doing its actions. The start hook may also be used to change attributes in the target object before the save action is performed. In this case the hook should return a null pointer. A third way to use such a hook is to do checks inside the hooks, maybe to check if the object to save has all its needed data. If the hook code will prevent the action from taking place, it should throw an exception.

Code structure

Setup parameters for Java service target

The setup for service target specifies contains the following settings:

Template Directory: %GENOVA8_HOME%\templates\service\java
The directory containing the template files.

Target directory: %GENOVA8_HOME%\target\service\java
The directory that will contain the generated files.

Skip generation of .new files: False
The generated files intended to be filled with manually written code will not be overwritten, but if the file does not exist it will be created. Existing files will either be skipped (True) or created at with a .new extension.

Create full Generated tag: True
All files generated will be tagged with a timestamp string identifying the generation time. This tagging will give a difference between different generations of files even if the content is unchanged. Setting this property to False will prevent the generation of the timestamp tag.

In the templates the following setup parameters are used:

ServicePackage: example.service
This is the top package containing the generated java service code. For each object selection the code will be placed in the package %ServicePackage%.%ObjectSelectionName%.

Client Generator

Client Generator generates code for the GUI part of an application. The code generated is based on the dialog models and their Object Selections together with the specifications given in the Client Designer setup for the wanted target.

The code generated for the Java/Jfc target is intended for use together with the code generated with Server Generator for the Java target. The client code may either access the service code thru a local interface or it may access the service thru e EJB interface using Spring. Both interfaces are supported by the distributed framework.