Open ModelSphere 3.0

Developer's Guide

You can reach Grandite at:

20-1220 Lebourgneuf Blvd.
Quebec, QC
Canada G2K 2G4

or

open-modelsphere@grandite.com

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

CHAPTER 1 - INTRODUCTION

Welcome to the world of Open ModelSphere. This application is used for building whole or partial models of an organization as seen from the viewpoint of the process analyst, the database designer or the object developer. As a free and open software, anyone including you can improve ModelSphere.

This guide is targeted to Java developers who want to fix bugs, improve the user interface, support new diagrams, or simply understand its architecture. To fully appreciate the guide, it is recommended to have a deep knowledge of the Java programming language and of the Swing graphical library.

1 Acknowledgments

Although derived from an existing product, releasing Open ModelSphere as a free software, was an exciting challenge.

Grandite would like to thank

- Prof. Daniel Pascot, Laval University, Quebec for his visionary impulse
- Prof. Dzenan Ridjanovich, Laval University, Quebec for his technical advice
- the Government of Quebec, in particular Ministère de la Santé et Services sociaux, for being the pioneer in adopting ModelSphere as an open source product
- NeoSapiens for their supportive collaboration

and last, but not least, its development team for their engagement that made this mission possible.

2 Version History

Open ModelSphere is an integrated tool for business process, data and UML modeling developed with the experience of more than 20 years.

The journey began when a team of professors and students at Laval University in Quebec City (Canada) started a development of the first graphically-based CASE tool which became a commercial product at the beginning of the 90s - the SILVERRUN suite for business process and data modeling. While the SILVERRUN Professional & Enterprise Series has been and will be further developed and updated, as early as 1998, pioneers in the SILVERRUN team started a parallel research on using alternative technologies to respond to the challenge of the new paradigm of object-oriented application design.

As a first result, SILVERRUN JD, a class modeling tool from Java developers for Java developers was created. It was one of the first products intensively using the Swing library. Inspired by the promising results, a next major step was scheduled: SILVERRUN ModelSphere - a full-featured modeling tool that is platform-independent and supports relational as well as object-oriented approaches.

Under Grandite's management SILVERRUN ModelSphere has been further developed as a commercial product to its current version. It has incorporated the former SILVERRUN-JD functionalities and includes relational data modeling, business process modeling as well as UML modeling. Sign of maturity, major classes of the existing ModelSphere code were written almost 10 years ago.

Driven by the spirit of the pioneers, in September 2008 Grandite as one of the first modeling tool vendors decided to release the SILVERRUN ModelSphere core application under GPL license as the free software Open ModelSphere.

3 Document Conventions

This document uses the following conventions:

- Words referring to user interface elements are specified in **bold**. The arrow character ($\rightarrow$) describes a sequence that the user must accomplish in the interface. For instance the **File→Open** sequence means the user has to click the File item in the menu bar first, and once the File menu appears, the user then clicks the Open menu item.

- In-text references to code elements (Java identifiers and keywords) included in this document use the font `Courier New`. Code listings are also printed with the font `Courier New`, and appears within gray boxes.

```
//Code Sample
public class MyClass {
```

- The document recommends several developing guidelines. Guidelines appear within two horizontal lines. All the guidelines are numbered, and they are grouped in the chapter 15 on page 127 for recapitulation purposes.

   Guideline #1: It is recommended to..

- Cross-references within this document are preceded by an arrow sign.

   ➢ For more details, consult the chapter x on page y.

- The document gives warnings about pitfalls that should be avoided. Warnings are indicated *in italics* within a box whose background is yellow and whose borders are red.

   Warning: You should avoid to..

4 Related Documentation

This document is a complement to the ModelSphere user's guide and the Javadoc.

The ModelSphere user's guide is an on-line help that can be consulted from the ModelSphere application. It is an HTML-based documentation and it is oriented to end users.

The Javadoc is generated from the source code of ModelSphere. It is oriented to the ModelSphere programmers. Javadoc gives implementation details of classes covered in this document.

5 Trademarks

Apache and related products are trademarks of The Apache Software Foundation.

DB2, IBM and Informix are registered trademarks of International Business Machines Corp.

Eclipse is a trademark of Eclipse Foundation Inc.

Java, NetBeans, Solaris, Swing and related products are trademarks of Sun MicroSystems Inc.

Linux is the registered trademark of Linus Torvalds.

Microsoft, SQL Server and Windows are trademarks of Microsoft Corporation.

Oracle is a registered trademark of Oracle Corporation.

Pentium is a trademark of Intel Corporation.

Python is a trademark of the Python Software Foundation.

Unix is a registered trademark of Unix System Laboratories, Inc.

Other company, product and service names may be used in this document, to the benefit of the trademark owner, with no intention of infringing upon the trademark.

CHAPTER 2 - GETTING STARTED

1 Requirements

For ModelSphere development, it is recommended to have a computer whose processor is 800 MHz (Pentium 4) or higher, 512 MB RAM, and with 300 MB of free disk space. The operating system can be Windows, Linux, Solaris or any Java 5 supported OS.

A Java Development Kit (version 1.5 or later) is required.

> http://java.sun.com/

An Integrated Development Environment (IDE) (for instance, Eclipse 3.2 or later, NetBeans 5 or later) is also strongly recommended.

> http://www.eclipse.org/

> http://www.netbeans.org/

2 Projects Installation and Configurations

2.1 Getting the source files

Download the source code and follow the instructions found on the ModelSphere web site:

> http://www.modelsphere.org

The following external tools are not required to run ModelSphere, but may be required to maintain its development.

- Ant (version 1.7.0)

    > http://ant.apache.org/

- JUnit (version 4.4)

    > http://www.junit.org/

- JavaCC (version 4.0)

    > https://javacc.dev.java.net/

2.2 Projects and libraries

ModelSphere's source code depends on several open-source Java archives. The following table lists all Java archives, grouped by providers.

The Apache Project	
The Jakarta Regular Expression Package (Regexp)	http://jakarta.apache.org/regexp/ jakarta-regexp-1.5.jar
Jython (formerly known as JPython)	
Jython	http://www.jython.org/Project/download.html jython.jar

Table 1: Java Archives Required by ModelSphere

It is possible that the versions of the archives and URLs may have changed since the moment they have been included in the ModelSphere project. Use the archives provided with the ModelSphere distribution rather than the archives mentioned above.

The table below summarizes the project and library dependencies for each project. The specified Java archives (.jar files) are included with the ModelSphere distribution.

Projects	JACK	SMS	Plug-ins
Project dependencies	Does not depend on any other project	JACK	JACK and SMS
Library dependencies	jython.jar jakarta-regexp-1.5.jar		

Table 2: ModelSphere Project and Library Dependencies

2.3 Building ModelSphere within an Integrated Development Environment

This section describes how to set up a ModelSphere project in an IDE, and how to build the application. Eclipse 3.3 is used as an example, but NetBeans or other IDEs can be used as well.

The following rules are recommended for the integration with Eclipse:

- Create one workspace for ModelSphere
- Create three projects: org.modelsphere.jack, org.modelsphere.sms and org.modelsphere.plugins
- All settings are workspace-specific rather than project-specific; use the same settings than those defined by default in Eclipse
    - The Compiler Compliance Level must be set to Java 1.5
- For each project, use **Properties→Java Build Path** to define compile-time dependencies, based on the previous table
- For the plug-in project, include each plug-in folder as the source of the build path

➢ Consult the chapter 13 on page 92 to know how to build ModelSphere outside an IDE.

2.4 Running ModelSphere within an Integrated Development Environment

- Create a run configuration:
  - With org.modelsphere.sms.Application as the main class
  - With -Dsun.swing.enableImprovedDragGesture -ms64m -mx512m -ss16m -server as Virtual Machine arguments
  - With the plug-ins folder as part of the class path

If you work on the Eclipse platform, you should have the following configuration:
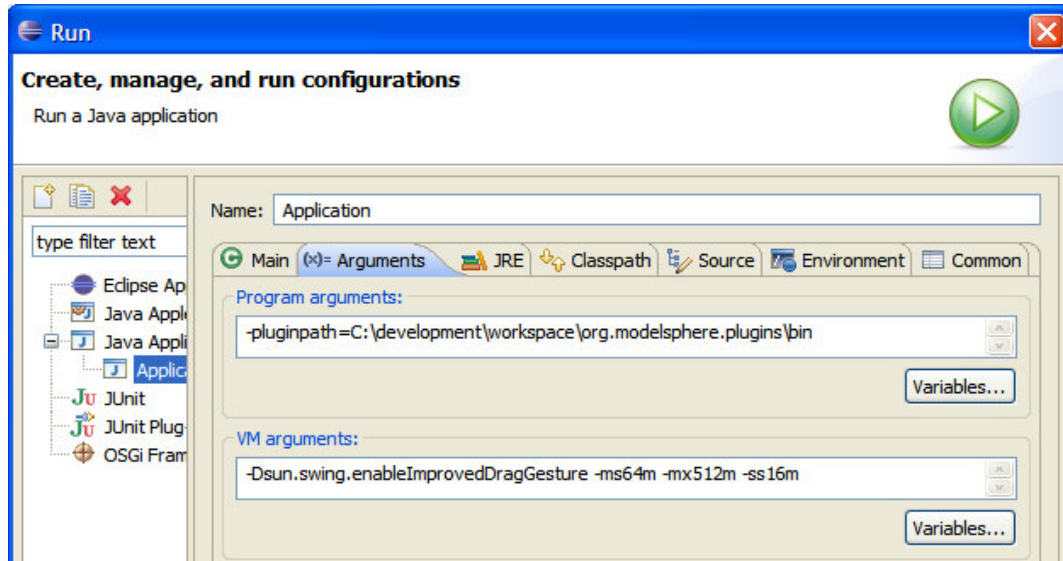
Figure 1: Run Configuration with the Eclipse IDE

It is possible to add the following option as program argument.

- -nosplash: starts ModelSphere without displaying the splash screen

It is possible to add the following option as program argument.

- -Dsun.java2d.noddraw: to disable java2d draw
- -Dsun.java2d.ddlock: to set the ddlock option in java2d

Refer to the Sun's documentation for further details on these options.

➢ Consult the chapter 13 on page 92 to know how to run ModelSphere outside an IDE.

2.5 Generating the Javadoc Pages

The source code includes javadoc comments. Follow the instructions of your IDE to generate the Javadoc pages from ModelSphere's source code.

If you work on the Eclipse platform, it is possible to open a Javadoc view (**Window→Show View→Javadoc**); Javadoc is generated on the fly while the developer is editing source code. It is also possible to generate all the Javadoc in a batch by doing **Project→Generate Javadoc..**

➢ Consult the chapter 13 on page 92 to know how to automatically generate the Javadoc pages outside an IDE.

2.6 Submitting Bug Fixes and Software Improvements

If you have fixed bugs in ModelSphere or added new features, you are welcome to submit changes that can be useful for the rest of the ModelSphere community. Consult the ModelSphere web site to learn how to contribute to the evolution of ModelSphere.

http://www.modelsphere.org

CHAPTER 3 - MODELSPHERE REQUIREMENTS

This section does not offer an exhaustive list of software requirements, but describes the basic requirements followed by developers at the moment ModelSphere was developed.

The requirements reflect the current status of ModelSphere, but they are called to be updated to guide the future evolution of ModelSphere.

1 User Interface Requirements

- Users can undo/redo their modifications to model elements; when it is not possible, users must be notified before performing the modification.

- They are several ways to modify model elements (by editing the Design Panel, by using in-place editing in the diagram, and so on). When a modification is made at one place, the modification must be immediately visible in all the other views.

- When users modify a model element by using a dialog, the changes are committed only when users press the Apply button.

- Long-running operations must be performed using the Progress Dialog; users should be able to cancel an operation at any time.

2 Diagramming Requirements

- A model may be rendered by several diagrams; modifying a model element on a diagram must update the other diagrams.

- A model element may be rendered by several figures (graphical objects) on the same diagram, modifying a model element by editing a figure must update the other figures. When a model element is rendered by several figures, each figure must be numbered (with the notation 1/2, 2/2).

- In addition to default styles and notations, it is possible to define custom (user-defined) styles and notations. On a diagram, each figure may have its own style.

- All diagrams can be zoomed in/out; Magnifiers and overview are available for all kinds of diagrams.

- Stamps, notes, graphical links, and drawing figures can be added on each kind of diagram.

3 Behavioral Diagrams Requirements

- Behavioral diagrams include Business Process Diagrams and non-structural UML diagrams.

- Each behavioral diagram can be refined in sub diagrams.

- The structure of a behavioral diagram and its sub diagrams may be represented by a tree diagram.

4 Object-Oriented Requirements

- ModelSphere supports all the concepts found at least in Java 1.4. In the future, concepts introduced by Java 1.5, such as generics and enumerations, may be added to requirements.

- ModelSphere is a Java-centric tool rather than a UML-centric tool. If there is a conflict between Java and UML, ModelSphere tends to be compliant with Java rather than UML[1].

- Besides Java concepts, ModelSphere supports concepts and diagrams defined in UML 1.4 at a minimum.

- ModelSphere does not support all concepts defined in UML, but supported concepts must be compliant to UML 1.4, at least.

- When users modify a object-oriented model, the model integrity is validated only when it is requested by users.

5 Relational-Oriented Requirements

- ModelSphere supports conceptual, logical and physical data modeling.

- ModelSphere is not tied to a particular notation, but supports several built-in notations (Information Engineering, Datarun, ..) and is extensible to user-defined notations.

- When users modify a relational model, the relational integrity is validated only when it is requested by users.

6 Persistence Requirements

- It is possible to open several projects during the same session, without having to close previous projects. It is possible to copy/paste model elements among different projects.

- It is possible to open a project built with a previous version of ModelSphere; this is called backward-compatibility. When a project is opening, it is updated to the new current version of ModelSphere. If the project is saved, it won't be possible to re-open it with an older version of ModelSphere.

---

1 For instance, ModelSphere allows to add fields to interfaces because it is permitted by Java (as long as they are static and final), even if it is not allowed by UML.

CHAPTER 4 - ARCHITECTURE OVERVIEW

This chapter describes the overall architecture of ModelSphere, and explains the roles and responsibilities of each major module. It presents the development principles behind the product, and the motivations for design choices.

1 The Three Layers of ModelSphere

The ModelSphere code is divided into three layers[2]: JACK, SMS and plug-ins. Each layer is tightly coupled with its underlying layer, but all the dependencies are one-way.
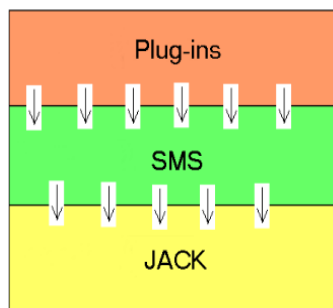
Figure 2: Layers

1.1 JACK

JACK (Java Abstract Class Kit) is the framework layer. It represents the application-independent level. Other applications than ModelSphere could be built using the same JACK framework as their underlying layer. JACK only depends on the Java standard library, and on third-party libraries. JACK is model-agnostic, changing the meta-model (discussed later) does not have any impact on the JACK layer.

Guideline #1: Never import an SMS class from the JACK library. JACK must stay independent from SMS.

1.2 SMS

SMS (Shared Modeling Software) is the application layer. It contains all the classes that implements features related to the ModelSphere application. SMS depends on the JACK layer, on the Java standard library, and on third-party libraries. SMS defines the meta-model, i.e. the application's model.

Guideline #2: Classes in SMS depend on the ModelSphere meta-data. If an SMS class is independent from the meta-model, consider to move it to the JACK framework.

---

2   See the glossary for the definition of Layer

1.3 Plug-ins

Plug-ins are the upper layer. The application must be able to run with or without the plug-ins loaded. When the application is developed, it is always possible to add functionalities by adding a new plug-in. Plug-ins depend on the JACK, the SMS, the Java standard library, and on third-party libraries. In general, plug-ins are also independent among them.

Guideline #3: Try to avoid inter dependencies among plug-ins; this will allow to publish a plug-in without having to change and publish other plug-ins.

2 The Three Partitions of ModelSphere

ModelSphere is divided in three logical partitions[3] : the application model (the persistent classes), the operational partition (controller), and the view (GUI) partition. The partitions depend on each other, but the coupling between partitions should be as limited as possible.
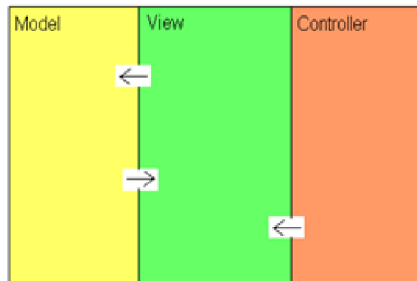
Figure 3: Partitions

2.1 The Meta-Model

The meta-model is the core of the application. The meta-classes represent the persistent data of the application, the data that can be saved, and thus can persist when the application session terminates.

A crucial point here is to distinguish between users' *models* and the ModelSphere *meta-model*. ModelSphere allows the user to create and modify *models* (this includes relational data models, UML class models and business process models). The framework that supports these different kinds of models is called ModelSphere's *meta-model*. End users cannot modify the meta-model, they simply use it to create their own models. ModelSphere's developers may have to update the meta-model to support new concepts in ModelSphere.

The meta-model is divided in several packages, all ending with the .db suffix.

Packages	Responsibilities
org.modelsphere.sms.db	Persistent data common to all kinds of models.
org.modelsphere.sms.be.db	Persistent data related to behavioral models, including business process modeling and UML dynamic models.
org.modelsphere.sms.oo.db org.modelsphere.sms.oo.java.db	Persistent data related to object-oriented models, UML and Java.
org.modelsphere.sms.or.db org.modelsphere.sms.or.oracle.db	Persistent data related to relational models, and DBMS-specific models, such as Oracle.

---

3   See the glossary for the definition of Partition

All the packages above all generated by a special plug-in called the *genmeta* plug-in; consequently developers cannot modify the contents of .db packages, otherwise their contents will be lost the next time the meta-model is generated.

Guideline #4: Do not modify SMS classes in a package ending with .db, otherwise the modifications will be lost the next time the meta-model is generated.

The meta-model classes, generated by *genmeta*, counts for nearly 15% of the total size of the application. Generating the meta-model allows the developers to quickly add new concepts in the application at a very low development effort.

2.2 *The Controller and the View Partitions*

The controller and the view partitions count for the remaining 85% of the application. ModelSphere being an application primarily based on graphical capabilities, its initial architects did not consider the need to rigorously separate processing capabilities from the rest of the application.

If future developers estimate that some ModelSphere features could be invoked in a headless way (where the workbench is not available), then they could isolate these features in separate packages. These packages (the controller) should not import any Swing classes to keep them independent of any GUI-constructs.

The org.modelsphere.sms.be.features package is an example of a headless package. Headless operations can be invoked from the command prompt, or by the JUnit framework.

3 Criteria for Layer Subdivision

Classes are divided into either the JACK or the SMS layer according to their dependency to the meta-classes. Classes that are independent from meta-classes are placed in JACK, they represent application-independent classes. Classes that are dependent (directly or not) to one or several meta-classes are placed in the SMS layer.

Classes that are related to a given plug-in are placed in the plug-ins layer, the ones not associated to a plug-in are placed in the SMS layer.

CHAPTER 5 - PACKAGES OVERVIEW

Since the overall architecture of ModelSphere has been described in the previous chapter, we will now describe each package at a deeper level.

1 The JACK (Java Abstract Classes Kit) Project

This layer gathers standard library extension packages, core packages and tool packages.

1.1 Extensions to the Standard Library (jack)

ModelSphere does not try to re-invent the wheel and uses the features provided by the Java standard libraries as much as possible. However, it may happen that functions found in the standard libraries do not exactly fit to the needs of ModelSphere, or that functions were not available in the standard libraries at the moment ModelSphere was developed. In this case, missing features were added in the JACK layer.

Guideline #5: Before adding a new feature in the JACK library, verify if it does not already exist in the Java standard library.

The Java standard library organizes its classes into logical packages. The java.awt package contains windowing classes, the package java.io contains input/output classes, the java.net package contains networking classes, and so on.

The JACK layer follows the same naming pattern, and these packages are named jack.awt[4], jack.io, jack.util, and so on. If ModelSphere needs an input-output feature, developers first look at the java.io package to find this feature. If it does not exist, this feature is implemented in the jack.io package. Each extension package should only depend on its underlying standard package, when possible.

Guideline #6: Packages in the JACK library should be named awt, io, net, text, util to indicate they are implementing features not found in the standard Java library.

Because JACK classes are very independent of the rest of the application, their reusability is very high.

It is possible that a new version of Java fills a lack encountered in a previous version. For instance, a spinner widget was implemented in jack.awt because absent in the 1.3 standard library, the current version of Java at the moment this widget was implemented in ModelSphere. Because spinners are provided starting with Java 1.4, they are no longer useful in the jack.awt package.

The following tables gives some examples of classes added in the JACK packages.

Guideline #7: When new features are added in the standard Java library, consider to remove unnecessary classes in JACK and to use standard classes instead.

---

4   Actually, the complete package name is org.modelsphere.jack.awt; we use jack.awt for concision purposes.

JACK Packages	Main Classes
jack.awt	FlowLayout2 : improves the behavior of java.awt.FlowLayout FontChooserDialog: a dialog to choose a font, a font size, bold and italics options. NumericTextField : a subclass of JTextField that filters out non-numeric character. TriStateButton : self-explanatory
jack.awt.dirchooser	DirectoryChooser: same as FileChooser, but only directories can be selected, and hierarchy of directories displayed as a tree explorer.
jack.io	DirectoryList: a class that returns an Iterator object that enumerates files recursively in a hierarchy of subfolders. IndentWriter: a subclass of java.io.PrintWriter that incorporates indent and unindent methods. LineRandomAccessFile: a subclass of java.io.RandomAccessFile, with random access possible by specifying a line number.
jack.util	JarUtil: a class of utility methods for Java archives. StringUtil: a class of utility methods for strings.

Table 4: Main Classes Implemented in the JACK Packages

1.2 The Modeling Framework Library (jack.baseDb)

The persistence data library, or Db object library, is the library that manages operations and persistence for Db objects. In ModelSphere, all model objects are an instance of a subclass of DbObject.

➤ A complete chapter (see chapter 6 on page 21) of this document is dedicated to the modeling framework.

1.3 The Graphic Library (jack.graphic)

➤ A specific chapter (see chapter 7 on page 52) in this document covers the graphic package. Refer to this chapter for more details.

1.3.1 jack.graphic

This package contains the diagramming graphic classes. Diagram is the central class of this package. ZoneBox is a graphical object separated into several compartments (called zones), it serves to display graphical tables, views, classes, processes, etc. The other classes are self-explanatory.

1.3.2 jack.graphic.shape

This package contains all the shapes that can be associated to graphical objects: RectangleShape is used for the graphical representations of tables and classes; OvalShape is used for the processes and UML use cases, RoundRectShape is used for UML states. Other classes are self-explanatory.

1.3.3 jack.graphic.zone

This package contains the classes that operate on graphical object compartments. A SingletonZone is a zone with only one element. The name compartment of a class is a SingletonZone. A MatrixZone is a zone separated into several Cells. The field compartment and the method compartment of classes are MatrixZone.

1.4 The GUI Library

Contains GUI management and advanced components, including focus management within ModelSphere.

➢ The chapter 8 on page 58 is dedicated to GUI features and components. Refer to the chapter for more information.

1.5 The Sharable and Reusable Tools Library (jack.srtool)

The jack.srtool (Sharable and Reusable Tools) package contains generic classes used by ModelSphere, but that could be used by a suite of modeling products. Because ModelSphere supports relational, object and process modeling, ModelSphere's designers did not feel the need to create a suite of interdependent modeling tools, but at the moment of implementation of srtool, this was still an option.

1.5.1 jack.srtool.actions

All classes contained in this package are named XXXAction and are ultimately subclasses of javax.swing.AbstractAction. This package groups all the actions common to any standalone application, such as OpenAction, CloseAction, SaveAction, ExitAction, and so on. Actions specific to ModelSphere (and to its meta-model) are placed in the org.modelsphere.sms.actions package. Actions are often available in the main menu (per instance, OpenAction, CloseAction, SaveAction and ExitAction are available in the File menu). Each action has a name and may have an associated icon. Each action implements a doActionPerformed() method that is invoked when the user clicks on the appropriate menu item. The first line of doActionPerformed() is a very good place to put a breakpoint to trace what is done when the user invokes a given operation.

Guideline #8: All the action classes (that inherits from the Swing's AbstractAction) should have the Action suffix and should be place in a package named .actions. This helps developers to find the action classes quickly.

1.5.2 jack.srtool.forward

This package contains all the classes related to forward engineering and template invocations. Template files are specialized plug-ins for generating files using a template language instead of compiled java files. Connector, Group, Property and Template classes implement the behavior of the CONN, GROUP, ATTR and TEMPL rules in template files.

1.5.3 jack.srtool.integrate

This package contains generic classes of the Compare/Integrate feature of ModelSphere.

2 The SMS Project

The SMS project is related to the ModelSphere application. It is organized into a behavioral part, a relational part and an object-oriented part. All other packages comprise the generic part of SMS.

2.1 The SMS Generic layer

This layer groups anything not specific to behavioral, relational or object-oriented modeling.

2.1.1 SMS

This is the root package of all SMS packages. AddElementPool specifies what can be added under a selected object. For instance, if a table is selected (in the explorer or in the diagram), the pop-up menu will show Add Column, Add PK, and so on. "Add" operations are available according to the specification provided by AddElementPool.

Application class

This class is very important because it contains the main() method to launch and initialize the application. It also contains the build number of the application (814, 815, etc.).

MainFrame class

Another central class because it defines the application frame, its getSingleton() static method can be invoked from anywhere and returns the MainFrame object.

2.1.2 sms.db

The meta-classes generated by genmeta. Most of these classes are abstract classes sub-classed by BE, OR or OO concrete classes. As any .db package, do not modify these classes manually because changes will be lost the next time the meta-model classes are generated.

2.2 The Behavioral Module (BE)

The behavioral module groups anything related to behavioral constructs. This includes the process modeling (BPM), but also the dynamic diagrams of UML (all except the class model and the package model).

2.2.1 sms.be

The BEModule is the entry point of that library. BEModelToolkit groups all the tool required to perform behavioral modeling. BESemanticalIntegrity is responsible to keep the integrity of the model when an element of the model is changed: for instance, if the user changes the partial cost of a process, BESemanticalIntegrity will update the total cost of that process.

2.2.2 sms.be.actions

This package groups all actions specific to behavioral modeling. For instance SplitAction and MergeAction are defined here. Each action being a subclass of javax.swing.AbstractAction, all of them implement doActionPerformed().

> The section Swing Action Extensions (section 9 on page 64) covers the action mechanism with more details.

2.2.3 sms.be.db

The meta-classes generated by genmeta for behavioral modeling. Do not modify these classes directly.

2.2.4 sms.be.db.util

Because it is not a good approach to put utility methods directly into meta-classes (longer and harder to maintain), all the utility methods are grouped into a BEUtility class. This singleton class contains utility methods to initialize DB objects.

2.2.5 sms.be.features

Contains the operational features (the functionalities) of the behavioral package. These operations are normally called by the action classes. Usually, it is recommended to avoid GUI classes (AWT and Swing) in feature packages, to keep them purely operational. This package is also a good place to make JUnit tests because the feature classes should be GUI-independent.

2.2.6 sms.be.graphic

Contains BEActorBox, BEStoreBox and BEUseCaseBox that are the graphical representations for actors, stores and processes. The content of actor boxes, store boxes, etc. and their behaviors are defined here.

2.2.7 sms.be.notation

Elements related to behavioral notations.

2.3 The Object-Oriented Module (OO)

The OO module is divided into two layers: an sms.oo layer, generic to all OO languages, and an sms.oo.java layer, specific to the Java Programming language. It would be possible in the future to add support for another OO language, like Microsoft C#, by creating a sms.oo.csharp layer under sms.oo. The OOModule is the entry point of OO features, and OOSemanticalIntegrity is responsible to keep the integrity of the OO model when the user modifies OO elements.

2.3.1 sms.oo.actions

Actions generic to all OO languages.

2.3.2 sms.oo.db

The meta-classes generated by genmeta for OO modeling. Do not modify these classes directly.

2.3.3 sms.oo.db.util

Because it is not a good approach to put utility methods directly into meta-classes (longer and harder to maintain), all the utility methods are grouped into a OoUtilities class. This singleton class contains utility methods to create DB objects.

2.3.4 sms.oo.graphic

The graphical representation of an association (OOAssociation), an inheritance (OOInheritance), a package (OOPackage), etc.

2.3.5 sms.oo.java.actions

Actions specific to the Java programming language.

2.3.6 sms.oo.java.db

The meta-classes generated by genmeta for Java modeling. For instance, DbJVClass, DbJVDataMember and DbJVMethod are defined here. Do not modify these classes directly.

2.3.7 sms.oo.java.db.srtypes

Contains the SR (sharable and reusable) types for metafield typing. A DbJVClass has a metafield called stereotype of type JVClassCategory. A Java method or field has a metafield called visibility of type JVVisibility. JVClassCategory is defined in srtypes package and can have one of the following values: CLASS, INTERFACE or EXCEPTION. A JVVisibility class JVClassCategory is defined in srtypes package and can have one of the following values: PUBLIC, PACKAGE, PROTECTED or PRIVATE.

2.3.8 sms.oo.java.graphic

Defines the graphical representations of classes.

2.3.9 sms.oo.java.validation

Implements the 'Validate for Java' feature.

2.4 The Relational Module (OR)

The relational module is named OR (Object-Relational) because it is intended to support object-relational concepts in addition to standard relational concepts (several systems incorporate OO concepts into their RDBMS products (Oracle 8.0 allows to associate method to type, Informix UDB allows to define inheritance between types). But for now, only standard relational concepts are supported.

The OR module is divided into two layers: a sms.or layer, generic to all relational target systems, and a sms.or.XXX layer, specific to a given target system. The current supported target systems are sms.or.ibm (for DB2-UDB), sms.or.informix, sms.or.oracle, and sms.or.generic (for all other RDBMS, in that case, only concepts defined by ANSI-SQL are supported).

To add support for another OR target system, for instance SQLServer, an sms.or.sqlserver layer under sms.or would need to be created. ORModule is the entry point of OR features, and ORSemanticalIntegrity is responsible to keep the integrity of the OR model when the user modifies OR elements. ORValidation implements the relational validation.

The AnyORObject class is used for type conversion between different DBMSs.

2.4.1 sms.or.actions

Actions generic to all relational target systems, like GenerateForeignKeyAction and ChangeTargetSystemAction.

2.4.2 sms.or.db

The meta-classes generated by genmeta for OO modeling. Do not modify these classes directly.

2.4.3 sms.or.db.srtypes

Defines types for typing of metafields. It includes ORIndexKeySort (ASC or DESC), ORTriggerEvent (AFTER, BEFORE..).

2.4.4 sms.or.features.dbms

Generic classes for forward and reverse engineering. These classes are sub classed by plug-ins. For more information, refer to the Plug-ins chapter.

2.4.5 DBMS Specific Packages

There are four supported target systems. The generic one is for ANSI-databases:

- sms.or.generic
- sms.or.ibm
- sms.or.informix
- sms.or.oracle

2.4.6 sms.or.graphic

Defines the graphical representations of relational elements, e.g. ORTableBox defines the contents of a table in a diagram.

2.4.7 sms.or.graphic.tool

Defines the tools for relational operations. ORTableTool creates a new table in the current diagram, ORKeyTool defines a key on a selected column, ORAssociationTool creates an association between two tables, etc.

2.4.8 sms.or.notation

Elements related to relational notations.

3 The Plug-ins Project

The ModelSphere functionalities can be extended by adding plug-ins to the product. Some open-source plug-ins are distributed with ModelSphere, including:

- a forward engineering plug-in, responsible to generate SQL 92 scripts.

- a reverse engineering plug-in, responsible to reverse engineer generic elements[5] from a legacy database.

- a validation plug-in, responsible to validate the integrity of a data model.

In addition to these public domain plug-ins, commercial plug-ins exist but they are not covered by this document.

---

5 DBMS-specific elements, such as tablespaces and procedure libraries, are not reverse engineered by the generic reverse engineering plug-in.

CHAPTER 6 - MODELING FRAMEWORK

ModelSphere includes a modeling framework called DB that describes structured object model and generates high-quality model classes. The generated model classes include meta-data, generic getters and setters, and transaction support.

Those who know EMF[6] already understand the advantages of using a modeling framework; these people are invited to read the section 3.3 on page 48 in this chapter to quickly understand the similarities and differences between DB and EMF.

Those who are not familiar with any modeling framework should note that it is important to distinguish between the genmeta model, the genmeta plug-in and the generated model classes.
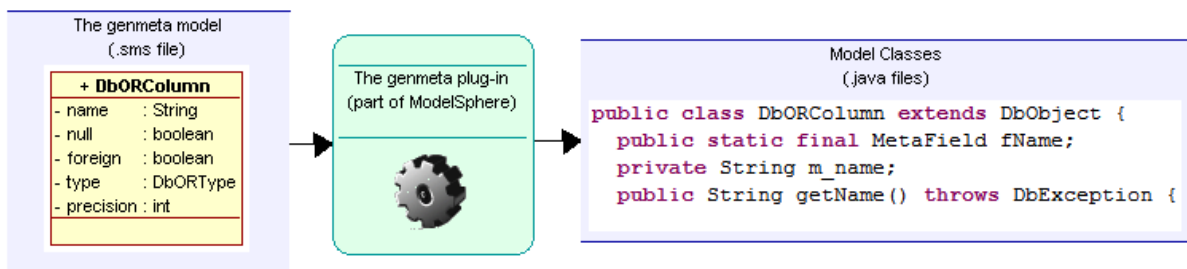
Figure 4: The Genmeta Process

The genmeta model

This is the source model, a standard .sms file. ModelSphere concepts, such as tables and columns, are modeled in the form of a class diagram. Special UDFs are defined, such as alias and icon, to define UI display name of columns and the icon associated to the column concept in ModelSphere. By convention, classes start with Db.

The genmeta plug-in

A special plug-in that takes the genmeta model as input and generates model classes as output. The logic of how to construct model classes is embedded in this plug-in.

Model classes

The code generated by the genmeta plug-in. For each modeled field in the genmeta model (e.g. the name field in the DbORColumn), the following are generated:

● A meta-field (fName), that can be consulted at the runtime to get information about a field;

● A value field (m_name), that contains the value (here a string);

● A getter (getName), that returns the current value of the field;

● A setter (setName), that change the current value of the field.

---

6   http://www.eclipse.org/modeling/emf/

Data and meta-data

The generated model classes contain data (here m_name), as any Java class. In addition, they contain meta-data (here fName). The data are associated to each instance (each individual column has its own name), while the same meta-data are shared by all the instances of a column. The meta-data can be consulted at runtime. For instance, application code can interrogate the column meta-data to obtain its GUI name, here metaField.getGUIName() will return "Column" (if the application runs in English) and not its programmatic name "DbORColumn".

Meta-fields and meta-classes

A meta-field is simply the meta-data associated to a model field, and a meta-class is the meta-data associated to a model class. It is possible to consult a meta-class at runtime and programatically obtain all its meta-fields.

1 The Data Access Layer (package db)

The modeling framework package is used for managing all the persistent objects.

1.1 The Db class

A Db instance is used to manage an entire graph of persistent objects. A Db instance usually corresponds to one project file. Many Db instances can be managed in ModelSphere at the same time. This allows operations such as copy and paste to be performed between projects. It is not permitted to associate objects managed by different db instances.

The method getDb() called on a DbObject returns the Db instance to which DbObject belongs.

The static method Db.getDbs() returns an array of all the Db objects opened.

The method getRoot() on a Db instance returns the root object of the database, which contains all the projects of the database (a single project in the case of a DbRAM).

1.1.1 DbRAM

There may be a number of DbRAM objects, each one representing a RAM project (a project loaded from a binary file). DbRAM is instantiated when loading a RAM project. The method terminate() on DbRAM closes the project. A DbRAM instance contains and manages a single project.

1.1.2 Client-Server Dbs

In a Client/Server application, there is a single 'Db' object representing the connection to the repository. This repository Db is instantiated at application start-up and closed at application exit. It contains all the projects of the repository. These Db specialized classes are currently disabled in ModelSphere and additional work is required before they can be used.

1.2 Packages Overview

Packages described here are sub packages of org.modelsphere.jack.baseDb package.

1.2.1 the db package

This package contains the main classes of the Db library:

DbObject

DbObject is the superclass of all the meta classes. DbObject implements java.io.Serializable.

DbException

Exception thrown when a Db object is accessed outside a transaction, when two write transactions are opened at the same time, when an enumeration is not closed, etc.

DbRelationN

A one-to-many relationship between two Db objects.

DbEnumeration

Subclass of java.util.Enumeration to enumerate DbObject.

1.2.2 db.event

Contains Db listener and event classes. They are described in a later section in this chapter.

1.2.3 db.srtypes

Contains SR (sharable and reusable) basic types. SR types serve to type meta fields. Some examples of SrTypes are SrInteger, SrColor, SrFont, SrRectangle, etc. See jack.baseDb.db.meta below.

1.2.4 db.xml

Contains the functionality to load and save a project file into an XML format instead of the standard binary format (.sms serialized file).

1.2.5 db.meta

Contains the meta framework. One MetaClass instance is associated to each meta class (DbJVClass is an example of MetaClass). A meta-class contains several MetaField and several MetaRelation. A metafield can be typed by primitive types, String or any sharable and reusable type (SrInteger, SrFloat). A MetaRelation navigates to other Db Objects.

1.2.6 screen

Contains the code to display generic screen. A generic screen is a screen with two columns of entries. The left column displays the property names. The right column displays the property values. The Design Panel and the Design tab in the properties window are two generic screens. The right column renders and edits property values.

1.2.7 screen.plugins

For each meta field XXX data type, a renderer and editor can be specified (XXXRenderer and XXXEditor classes). The renderers and editors are contained in this package. Note that the sms layer contains its own screen plugins package.

As example, for the Boolean type, the editor is BooleanEditor (a checkbox) and the renderer is BooleanRenderer.

Classes prefixed with MultiXXX are for the Design Panel, which can display and edit values for a selection of multiple objects. Other renderer and editors are targeted to single object.

1.3 Transactions

The modeling framework uses and enforces transactions. A database object cannot be accessed outside a read or a write transaction (otherwise a DbException is thrown). A database object cannot be modified from outside a write transaction. For instance, the following example will generate an exception because getName() cant be read outside a transaction.

```
void readName(DbJVClass claz) {
  String name = claz.getName(); //Wrong, generates a DbException
}
```

To correct this, a transaction must be started before accessing the claz object:

```
void readName (DbJVClass claz) {
  Db db = claz.getDb();
  db.beginReadTransaction();
  String name = claz.getName(); //Good
  db.commitTransaction(); //don't forget to commit transactions
}
```

All transactions must be closed either by committing the changes or by aborting the transaction (rollback). When aborting a transaction, all the modifications within the transaction are canceled. Once you have committed or aborted the transaction, you cannot access a DbObject before starting a new transaction.

The method commitTrans() commits the transaction and the method abortTrans () aborts the transaction.

Write transactions must be named. This name is displayed to the user for undo-redo operations.

```
void writeName (DbJVClass claz, String newName) {
  Db db = claz.getDb();
  db.beginWriteTransaction("Rename class"); //each write transaction must have a name
  claz.setName(newName); //Good
  db.commitTransaction(); //don't forget to commit transactions
}
```

Transaction's name is ignored for nested transactions.

Starting and committing a transaction causes some accesses to the server, so avoid starting a read transaction in each method that accesses DbObjects. Instead, start and commit a single transaction at the outermost level of the command execution.

On the other hand, you cannot keep a transaction opened for a long period of time. So you must insure that no transaction is opened when you display a modal dialog (for example an option dialog or a message box). If you are executing a process that needs some information from the user, you normally proceed as follows:

1. Start a read transaction to get from the database the information needed to initialize the option dialog then you close the transaction.

2. Display the option dialog.

3. If the user closes the dialog with OK, start a write transaction, perform the process, then close the transaction.

4. Display a termination message on the screen (AFTER closing the transaction).

It is possible to listen to transactions. See the listeners and events section for more information.

1.3.1 Nesting Transactions and Threading

You may nest transactions; the nested transactions only have a conceptual meaning, i.e. committing the nested transaction does nothing else than decrementing the nesting level; only the commit of the outermost transaction performs the real commit.

A transaction can be started on a db instance if another transaction was already running but they can't be started from two different threads. If a transaction starts when another transaction was running, the db instance will throw an exception if the threads differ. This check is always performed by the db instance.

> *Aborting a nested transaction aborts in reality the outermost transaction. All the modifications done since the outermost transaction was started are canceled, and you are no more in an opened transaction (you cannot access a DbObject before starting a new transaction). So use abort only at the outermost level.*

1.4 Types

1.4.1 Fields Types

An ordinary field of a DbObject (i.e. not a relation field) must have one of the following types:

1. A Java primitive type

2. A String

3. A type inheriting from SrType

Any other type is not allowed. Arrays are not allowed.

The 1-relation fields have as type the neighbor class. The N-relation fields have as type 'DbRelationN'. For example, the recursive 1-N relation 'composite-components' on DbObject defines the two following fields:

● DbObject composite

● DbRelationN components

SrTypes

Rules defining a SrType.

The fields of a SrType must have one of the following types:

- A primitive type.
- A string.
- A type inheriting from SrType
- A one-dimension array of primitive type or String.

The SrType must define the method equals.

If the SrType contains fields of type SrType, or array fields, it must override the method dbFetch(db), defined as follows:

```
public final void dbFetch(Db db) throws DbException {
  db.fetch(this); // first line mandatory
  db.fetch(array); // use this expression for each array field
  srtype.dbFetch(db); // use this expression for each SrType field
}
```

It must also override the method dbCluster(db, parent) in a similar way:

```
public final void dbCluster(Db db, Object parent) throws DbException {
 db.cluster(this, parent); // first line mandatory
 db.cluster(array, this); // use this expression for each array field
 srtype.dbCluster(db, this); // use this expression for each SrType field
}
```

The property serialVersionUID for the SrType must be computed.

There are two categories of SrTypes: 3rd-party types and abstract types.

3rd-Party Types

The ones that correspond to a 3rd party type; in this case, the name of the SrType must be 'Sr' + the name of the 3rd party type (example: SrFont correspond to Font). This category of SrType does never appear in the API. The API always uses the 3rd party type. The API methods convert internally from the 3rd party type to the SrType and vice-versa.

These types must define one constructor with a single parameter of the 3rd party type (for example: SrFont(Font font)). It must also define the method toApplType(), which returns an object of the 3rd party type.

Abstract Types

They must inherit from DbtAbstract and their name must begin by the prefix "Dbt". Example: DbtPrefix.

These types must implement the method duplicate(), which is equivalent to clone() (clone() cannot be defined for a SrType because it inherits from the abstract persistent-capable class supplied by the OODBMS, which does not support clone()). The method duplicate() is used in the DbObject's get/set methods to give a copy of the DbtAbstract value kept in the database (not the value itself) to the application. The method duplicate() must perform a 'deep clone', i.e. must duplicate recursively the fields of the DbtAbstract that are SrType or arrays.

1.5   Working with DbObject

1.5.1   Getting a field's value on a DbObject

To get the value of the field 'XXX' of a DbObject, call the method getXXX(). For example getName() on a DbSemanticalObject returns the value of the field 'name'.

If the field is an N-relation, the method returns a DbRelationN object, which contains the list of neighbors. DbRelationN defines the following methods:

```
public int size();
public DbObject elementAt(int index);
public DbObject getParent(); // returns the DbObject containing this DbRelationN.
public MetaRelationN getMetaRelation();
public DbEnumeration elements();
public DbEnumeration elements(metaClass);
public DbEnumeration elements(metaClass, Db.ENUM_FORWARD or Db.ENUM_REVERSE);
```

The method 'elements' returns a DbEnumeration. DbEnumeration defines the following methods:

```
public boolean hasMoreElements();
public DbObject nextElement();
public void close();
```

It is mandatory to close the enumeration with the method close(). If a DbEnumeration is opened when the commit is called, a RuntimeException will be thrown. On the other hand, aborting a transaction will also automatically close any opened DbEnumeration. Consequently, if a DbException is thrown, the transaction will be aborted in the process of throwing the exception, which will cause all DbEnumerations to be closed. Thus it is not necessary to put a DbEnumeration in a block try-finally in order to close the enumeration in case of DbException.

The DbRelationN can be modified during the enumeration (neighbors added, removed, reinserted). The removed neighbors positioned after the cursor are not enumerated. The added neighbors positioned before the cursor are not enumerated. The added neighbors positioned after the cursor are enumerated.

Example: enumerating all the projects of the repository:

```
DbRelationN components = db.getRoot().getComponents();
DbEnumeration enu = components.elements(DbProject.metaClass);
while (enu.hasMoreElements()) {
   DbProject project = (DbProject)enu.nextElement();
   ........
}
enu.close();
```

Another way to get the value of a field is to use the generic method get(metaField). For example, get(DbSemanticalObject.fName) is equivalent to getName().

To get the value that the field had at the beginning of the transaction, use the method getOld(metaField). If the field has not been changed during the transaction, getOld returns the same value as get. This method can't be used on a N-relation.

The method get(metaField, Db.NEW_VALUE or Db.OLD_VALUE) combines the actions of get(metaField) and getOld(metaField). It returns the actual value or the value at the beginning of the transaction according to the second parameter. This method can't be used on an N-relation.

The method getAccordingToStatus(metaField) returns the value at the beginning of the transaction, if the DbObject has been removed in the transaction, otherwise it returns the actual value. This method can't be used on an N-relation.

The method hasChanged(metaField) returns true if the field has been modified during the transaction, false otherwise. This method can't be used on an N-relation.

The method getNbNeighbors(metaRelationship) returns the number of neighbors connected to this relation.

1.5.2 Setting a Field's Value on a DbObject

To set the value of the field 'XXX' of a DbObject, call the method setXXX(value). For example: setName("table") on a DbSemanticalObject assigns to the field 'name' the value 'table'.

In a 1-N relation, the method setXXX is defined only on the child side.

In the case of a 1-1 relation, the new neighbor to connect to must not itself already be connected to a third object.

In the case of an N-N relation, the method setXXX requires a second parameter; Db.ADD_TO_RELN or Db.REMOVE_FROM_RELN, indicating if we want to add a new neighbor or remove an existing neighbor in the relation. You may alternatively use the methods addToXXX or removeFromXXX.

The method setXXX calls one of the protected methods 'basicSet' or 'setRelationNN', not accessible to the application, to set the actual value. These methods return true if the new value is different from the previous one, false otherwise. The method setXXX may check this return code if required by its logic.

Another way to set the field's value is to use the generic method set(metaField, value). For example: set(DbSemanticalObject.fName, "table") is equivalent to setName("table"). The method 'set' just calls the corresponding setXXX. This is a bit slower, but it insures that any modification to a field passes necessarily by setXXX. Code that would normally be put on triggers (update listeners) can thus be put in setXXX.

The method 'set' can be used on any field, including N-relations. In the latter case, it adds a new neighbor to the relation. If called on the parent side of a 1-N relation, it calls the setXXX method of the opposite side. For any side of an N-N relation, it calls setXXX(value, Db.ADD_TO_RELN).

For an N-N relation, you may use the following overload of 'set': set(metaRelationN, neighbor, Db.ADD_TO_RELN or Db.REMOVE_FROM_RELN). This calls the corresponding setXXX(neighbor, Db.ADD_TO_RELN or Db.REMOVE_FROM_RELN) method.

1.5.3 Adding a DbObject

To add a DbObject, simply call its constructor passing as parameter the composite of the new object (i.e. its parent in the composition hierarchy). For some classes of DbObject, the constructor may require additional parameters. The following code shows the creation of a DbSMSProject using the constructor.

```
DbSMSProject project = new DbSMSProject(db.getRoot());
```

Another way to add a DbObject is to use the composite method composite.createComponent(metaClass). This method returns the new object, or null if the object could not be instantiated because its constructor requires additional parameters or because the composite cannot have components objects of the class <metaClass>. The following example shows how to create a new DbSMSProject using the composite method. In the case of DbSMSProject, the root object is a valid composite.

```
DbSMSProject project = db.getRoot().createComponent(DbSMSProject.metaClass);
```

1.5.4 Removing a DbObject

To remove a DbObject, call the method remove(). This method sets the status of the object to Db.OBJ_REMOVED, unlinks the removed object from all the relations to which it is connected, and applies remove() recursively on all the objects connected to the removed object with a minimal connectivity of 1 (components and other dependencies). The method remove() may be overridden by subclasses of DbObject to propagate the removal to more objects than those connected with this generic rule. Ensure that a call to super.remove() is performed after the additional code.

A removed object is still accessible to get methods until the transaction is closed. Since a removed object is disconnected from all its neighbors, a get on a relation field will always return null. In this case, use getOld() instead of get() on a removed object. For code that must work on existing and removed objects, use getAccordingToStatus().

getOld() and getAccordingToStatus() cannot be used on an N-relation field.

After the transaction is closed, the removed objects become dead objects, and are no more accessible in subsequent transactions. Any attempt to access a dead object throws a DbDeadObjectException. It is possible to verify if an object has been removed by calling isAlive(). This method returns false if the object has been removed.

1.5.5 Reordering an N-relation

```
public void reinsert(MetaRelationN relation, int oldIndex, int newIndex);
```

The method 'reinsert' removes the element at oldIndex from the DbRelationN, then reinserts it at newIndex. After the operation, the element is at newIndex. Repeating the same operation by inverting oldIndex and newIndex undo the first reinsert.

If the method 'reinsertInXXX(int oldIndex, int newIndex)' is defined, the method 'reinsert' on relation XXX will call 'reinsertInXXX' to perform the reordering. Thus if reordering on a given relation must trigger some additional updates, simply define the method 'reinsertInXXX' and put in it all the code for the reordering and the additional updates. For the reordering itself, call the low-level method 'basicReinsert(relation, oldIndex, newIndex)'.

1.5.6 Utility Methods on a DbObject

The method getTransStatus() returns the status of the DbObject in the transaction, in the form of one of the following constants:

Db Constant	Description
Db.OBJ_ADDED	If the object has been added in the current transaction.
Db.OBJ_REMOVED	If the object has been removed in the current transaction.
Db.OBJ_MODIFIED	If the object has at least one field modified in the current transaction.
Db.OBJ_UNTOUCHED	Otherwise.

Table 5: Db Constants and Description

The method getProject() returns the DbProject to which the DbObject belongs. Calling this method on a DbProject will return the project itself. Calling getProject() on DbRoot returns null. The project is kept as a transient variable in each DbObject, so this method is instantaneous, may be called outside a transaction and does not throw DbException. This method can also be used on a removed object.

The method getCompositeOfType(metaClass) returns the first ancestor of the class specified encountered while going up the composition hierarchy, or returns null if no ancestor found. This method can also be used on a removed object.

The method getSemanticalName(SHORT_FORM or LONG_FORM) returns the semantic name of the DbObject, under its short or long (qualified) form according to the parameter.

The method hasField(metaField) returns true if the field exists for a DbObject, i.e. it is a field belonging to the class of the DbObject or one of its super classes.

The method componentTree(metaClass) on a DbObject returns an enumeration of all the descendants (in the composition hierarchy) of the object. If <metaClass> is not null, the enumeration will involve only the descendants that are instance of that class. This method is also available with an additional boundaries parameter. The boundaries indicate that if an object matches the type of any boundary class, its components will be excluded recursively.

1.5.7 Transient Fields in DbObjects

The transient fields are initialized to null (or 0 if primitive type) the first time they are loaded in the cache (i.e. the first time you get a reference to the object as result of a get on a relation field).

The transient fields are not in the meta (do not have a corresponding metaField). It is not possible to attach a listener to them.

The transient fields are not transactional. It is possible to access them outside of a transaction. They are not affected by abortTrans or undo.

1.6 Listeners and Events

ModelSphere's framework offers complete mechanisms to listen for any changes occurring on Db objects. It also provides listeners for tracking transactions. These listeners are a key feature because they are used by all GUI elements that requires to be updated in order to be in sync with the Db objects. During a transaction, the Db instance collects all the informations required to provide notifications on any changes.

Listeners observes Db objects, and reacts when Db objects are modified. There are two types of Listeners: refresh and update. Both are used for different purposes and it is very important to understand the differences between them.

DbRefreshListener

DbRefreshListener is called when the transaction commits. They are typically used to refresh the diagrams and other GUI components. Refresh listeners are notified once the commit has been performed on the transaction. During that post transaction phase, only read operations are allowed.

The refresh listeners are called at the end of the commitTrans process (after the physical commit), and also at the end of an undo or redo transaction, in order to refresh the interface elements from the modifications just committed in the database.

> *Refresh listeners cannot modify the database. Doing so will throw a DbException.*

The refresh listeners can be registered in two different ways:

A Refresh Listener on MetaField

A refresh listener registered to a metaField is called for each object having the status Db.OBJ_MODIFIED for which this field has been modified during the transaction. The field may be an N-relation.

For added or removed objects, the refresh listeners registered to a metaField are not called, except the ones registered to the field <DbObject.fComposite>. For this particular field, the refresh listeners are called for each object added or removed during the transaction, and also for each object whose composite has changed during the transaction.

A DbProject can be specified when registering a refresh listener to a metaField. In this case, the listener is called only for objects belonging to this project.

To register / unregister a refresh listener to a metaField (for example, DbObject.fComposite):

```
DbObject.fComposite.addDbRefreshListener(listener [, project]);
DbObject.fComposite.removeDbRefreshListener(listener);
```

There are corresponding static methods in MetaField to register / unregister a refresh listener to many metaFields in one call.

A Refresh Listener on a DbObject

A refresh listener can be registered to a DbObject using the parameter values:

Constant	Description
DbRefreshListener.CALL_FOR_EVERY_FIELD	The listener is called for all modified fields.
DbRefreshListener.CALL_ONCE	The listener is called only once regardless of how many fields have changed.

Table 6: DbRefreshListener Constants and Description

A refresh listener registered to an object with CALL_FOR_EVERY_FIELD is called for each field of the object modified in the transaction, including the N-relations; however, if the object has the status Db.OBJ_ADDED or Db.OBJ_REMOVED, the listener is called only once for the field DbObject.fComposite.

A refresh listener registered to an object with CALL_ONCE is called only once if the object has one of the status Db.OBJ_ADDED, Db.OBJ_REMOVED or Db.OBJ_MODIFIED.

To register / unregister a refresh listener to the DbObject dbo:

```
dbo.addDbRefreshListener(listener, DbRefreshListener.CALL_FOR_EVERY_FIELD);// or CALL_ONCE
dbo.removeDbRefreshListener(listener);
```

A refresh listener may be registered to any number of metaFields and/or DbObjects. Any number of refresh listeners may be registered to a metaField or a DbObject.

The interface DbRefreshListener defines one method:

```
void refreshAfterDbUpdate (DbUpdateEvent event);
```

See bellow in **Update Listeners** for a description of the event object.

In the case of a listener registered to an object with CALL_ONCE, event.metaField is null.

The refresh listeners are called only for local transactions. They are not called for external transactions (transactions performed by other users).

You can also register 'refresh pass listeners', which are called twice in each transaction. Their method beforeRefreshPass (db) is called before the first refresh listener is called and their method afterRefreshPass (db) is called after the last refresh listener was called. Refresh pass listeners are registered globally (for all the databases). Use the static methods Db.addDbRefreshPassListener(listener) to register a refresh pass listener and Db.removeDbRefreshPassListener(listener) to unregister it.

DbUpdateListener

A DbUpdateListener is called at the moment a Db object is modified. Used for semantic integrity. All semantic integrity code is regrouped based on which module they apply. For example, org.modelsphere.sms.or.ORSemanticalIntegrity contains all the integrity code related to object-relational modeling.

GUI components must not use update listeners to refresh their contents because update listeners are very costly from a performance point of view.

Update listeners can perform additional write operations.

The update listeners are called during the commitTrans process to complete the transaction before the physical commit. They correspond to the concept of trigger in RDBMS. They are called for all the modifications made to the database **before** the commitTrans. They are **not** called for modifications made during the commitTrans process by other update listeners.

An update listener is registered to a metaField, and is called for each object having the status Db.OBJ_MODIFIED for which this field has been modified during the transaction. The field may be an N-relation.

For added or removed objects, the update listeners are not called, except the ones registered on the field <DbObject.fComposite>. For this particular field, the update listeners are called for each object added or removed during the transaction, and also for each object whose composite has changed during the transaction.

An update listener may be registered on any number of metaFields and any number of update listeners may be registered on the same metaField.

An execution priority is specified for each pair <metaField, update listener>. The update listeners are executed in increasing priority order. For a same priority, the order of execution is undefined.

To register / unregister an update listener to a metaField (for example, DbObject.fComposite):

```
DbObject.fComposite.addDbUpdateListener(listener, priority);
DbObject.fComposite.removeDbUpdateListener(listener);
```

There are corresponding static methods in MetaField to register / unregister an update listener to a number of metaFields in one call.

The interface DbUpdateListener defines one method: void dbUpdated(DbUpdateEvent event);

The DbUpdateEvent object contains the following variables:

```
DbObject  dbo;         // the modified, added or removed object.
MetaField metaField;   // the modified field.
DbObject  neighbor;    // if N-relation, the neighbor added to / removed from the relation
                       // (null if reinsert).
int       op;          // if N-relation, Db.ADD_TO_RELN  / REMOVE_FROM_RELN /
                       // REINSERT_IN_RELN.
```

When possible, avoid the use of update listeners. For example, put the code that would go into an insert (add) trigger into the constructor, put the code that would go into a remove trigger into the remove method, and put the code that would go into an update trigger into the setXXX method.

It is safe to do so because adding an object always calls the constructor, removing an object is always performed by the remove method, and setting a field is always performed by the setXXX method. There is one exception to this rule: the method deepCopy, which creates a copy of an object, does not call the constructor to create the copy nor the setXXX methods to set the fields of the copy. However, you can customize the deepCopy process by supplying a DeepCopyCustomizer. For example, the method initFields of the DeepCopyCustomizer is called after creating the copy of the object and setting the non-relation fields of the copy. It is possible to add code in this method to modify the setting of the non-relation fields of the copy.

You can also register 'update pass listeners', which are called twice in each transaction: their method beforeUpdatePass(db) is called before the first update listener is called; and their method afterUpdatePass(db) is called after the last update listener was called. Update pass listeners are registered globally (for all the databases). Use the static methods Db.addDbUpdatePassListener(listener) to register an update pass listener and Db.removeDbUpdatePassListener(listener) to unregister it.

Guideline #10: Use the refresh listener to refresh graphical objects after a modification on the model has been reported; use the update listener to change values on model elements after a modification on the model has been reported.

DbTransListener

You can register transaction listeners, which are called at the begin and the end of each not nested transaction. Use the static methods Db.addDbTransListener(listener) to register a transaction listener, Db.removeDbTransListener(listener) to unregister it. Their method dbTransBegun(db) is called immediately after a transaction has started, and their method dbTransEnded(db) is called immediately after the transaction was terminated (commit or abort).

DbUndoRedoListener

You can register undo and redo operations, and use these listeners to be notified when an undo/redo operation is performed.

1.7 Exceptions

1.7.1 DbException

The Db methods beginReadTrans, beginWriteTrans and commitTrans, and all the methods that access or modify a DbObject, may throw a DbException. The most frequent case of DbException is a conflict between two concurrent transactions (from two users).

> *When a DbException arises, all the currently opened transactions on all the Db's are automatically aborted before throwing the DbException. Thus when the control is passed to the first catch block, there is no open transaction, so you need to start a new transaction if you want to access a DbObject.*

Because all the transactions are aborted when a DbException is thrown, the try/catch block should be installed around the outermost transaction (which should be at the outermost level of the command execution). All the methods that access DbObjects should be declared 'throws DbException' and should not catch the DbException themselves.

A transaction at the outermost level should be coded as follows:

```
import jack.srtool.ApplicationContext;
import jack.util.ExceptionHandler;

try {
    db.beginWriteTrans("action");
    performAction();
    db.commitTrans();
} catch (Exception e) {
    MainFrame frame = ApplicationContext.getDefaultMainFrame();
    ExceptionHandler.processUncatchedException(frame, e);
}
```

Guideline #11: Catch any exception (not only DbException) and call processUncatchedException, which ensures that any open transaction (necessary if the exception is not DbException) is aborted and displays the message of the exception on the screen.

1.7.2 DbDeadObjectException

Subclass of DbException. This exception indicates that you are accessing a DbObject that was removed in a previous transaction (normally by another user).

1.8 Collections

It is possible to iterate through collections by using DbEnumeration. DbEnumeration works in a similar way as the standard enumeration class defined in java.util. The main difference is that DbEnumeration needs to be closed. The following example shows how it is used:

```
void enumerateClassMethod(DbJVClass claz) {
  Db db = claz.getDb();
  db.beginReadTransaction();
  DbRelationN relationN = claz.getSuperInheritances();
  DbEnumeration enum = relationN.elements(DbJVInheritance.metaClass);
  while(enum.hasMoreElements()) {
    //safe cast, because elements are all DbJVInheritance instances
    DbJVInheritance inher = (DbJVInheritance)enum.nextElement();
    ..
  } //end while
  enum.close(); //enumerations MUST be closed
  db.commitTransaction();
}
```

Note that it is possible to filter specific types when creating the enumeration. This is useful when working with components:

```
...
//DbJVMethod objects are components of DbJVClass
DbRelationN relationN = claz.getComponents();
DbEnumeration enum = relationN.elements(DbJVMethod.metaClass);
...
```

1.9 User-Defined Fields

The static method DbUDF.getUDF(project, metaClass, udfName) returns the DbUDF object corresponding to the UDF <udfName> in the class <metaClass>, or null if this UDF is not defined in the project.

The two following methods on DbObject, get(dbUDF) and getUDF(udfName), return the UDF value for this object (null if no value).

The method set(dbUDF, value) on DbObject sets the UDF value for this object (value may be null).

Note: User-defined fields are called User-defined properties in the user interface.

1.10 Matching Facility

The matching facility allows to associate objects in pairs between two projects (or more generally between two composition structures). When two objects are associated in pair, each of them cannot be associated with a third one: the matching association is exclusive. The matching facility allows the user to compare two projects and to find out which objects have been added, deleted or modified in a target project compared to a source project. This facility is used by the Compare/Integrate action in ModelSphere.
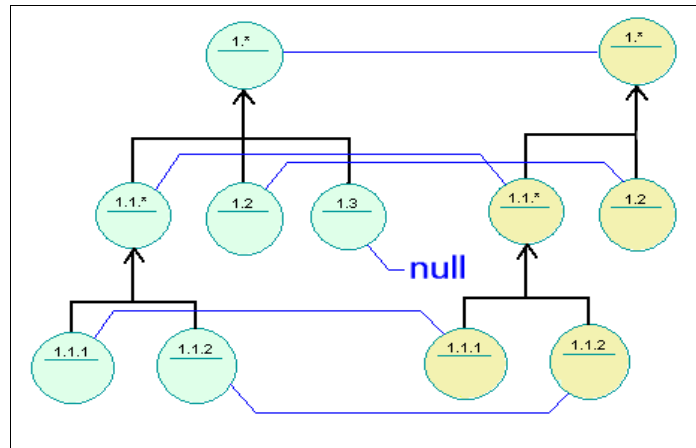
Figure 5: The Matching Facility Illustrated

The matching facility navigates through the composition structure in the source project. For each composite, it iterates all its components, recursively. The matching facility tries to associate each component in the source structure (in green in the figure) to a peer in the target structure (in yellow). The matching association (in blue) is exclusive: objects are matched with no more than one peer. Two objects are matched if they have the same semantic identifiers.

The matching operation must be performed within a matching session. The matching facility is a single resource common to all databases and only one matching session can be active at a time.

To begin a matching session, call the static method Db.beginMatching() and to terminate it, call the static method Db.endMatching(). The matching session is not bound to a transaction, nor to a database. Thus it is possible to match objects from different databases.

Once the matching session is started, you may use the following methods:

The method dbObj1.setMatchingObject(dbObj2) makes each object reference the other. None of the two objects may already be matched to another object. The dbObj1 and dbObj2 instances may refer to the same object.   The parameter dbObj2 may be null, meaning that dbObj1 has no matching object.

The method dbObj.findMatchingObject() returns the matching object (null if no matching object). If findMatchingObject() is called for the first time on dbObj, it searches for an object with the same semantic identifier in the other composition structure; and calls setMatchingObject to match dbObj with the object found (null if no object found).

Before calling findMatchingObject, you must associate the two projects this way:

```
project1.setMatchingObject(project2); // project is matched to itself
```

If matching is to be done between two subtrees of the same project, associate the two subtrees this way:

```
project.setMatchingObject(project); // project is matched to itself
tree1.setMatchingObject(tree2);
```

1.11 Method deepCopy()

The static method DbObject.deepCopy(srcObjs, composite) creates a copy of all the source objects and their descendants (in the composition hierarchy), and attaches the copies to <composite>.

> *Note that deepCopy does not call the constructors and the setXXX methods to create the copies and fill in their fields.*

All the source objects must be in the same project, and a source object cannot be a descendant of another source object. The destination composite may be in a different database; in this case, a read transaction must be active on the source database, and a write transaction must be active on the destination database.

The copied objects have all their non-relation fields set to the value of the source. For relation fields, the following rules apply:

1. If the neighbor of the source object is copied, link the copy of the source to the copy of the neighbor.

2. Else if the relation is not marked 'copy links', set the link to null in the copied object.

3. Else if the copy operation is within a single project, link the copied object to the same neighbor as the source object.

4. Else (copy from one project to another), find in the destination project the destination neighbor (i.e. the object having the same semantic identifier as the source neighbor), then link the copied object to the destination neighbor, or set the link to null if the destination neighbor does not exist.

> *deepCopy uses the matching facility.*

1.12 Migrating Models

A new version of ModelSphere may come with new concepts in the model classes. When a newer version of ModelSphere tries to read a model generated by a older version, the model does not contain the concepts inserted since the last version; in this case it is necessary to migrate the model built with the older version of ModelSphere to the format compliant with the newer meta-model.

The class SMSVersionConverter in the org.modelsphere.sms package is responsible to migrate models to the newer meta-model version. Consult the "How to" section for further details.

Of course, when an older version of ModelSphere tries to open a model written with a newer version of ModelSphere, an incompatibility error occurs. There is no need to "migrate back" the model. ModelSphere does not pretend to be forward-compatible.

➢ Consult the section 7 on page 121 for more details about the SMSVersionConverter class and model version converter.

2 Meta Model

This section covers meta-data concepts, such as MetaField and MetaClass. Consult the introduction of this chapter to distinguish between data and meta-data.

2.1 Working with the Meta Data

Each class in the DbObject hierarchy is represented by a MetaClass object, given by the static constant "metaClass" (ex: DbSemanticalObject.metaClass); to get the MetaClass of a DbObject, use the method getMetaClass().

Each field of a class in the DbObject hierarchy is represented by a MetaField or MetaRelationship object; given by the static constant whose name is "f" followed by the field name (ex: DbSemanticalObject.fName is the MetaField representing the field "name" of DbSemanticalObject).

Note: MetaRelationship is an abstract subclass of MetaField; MetaRelationship has 3 concrete subclasses: MetaRelation1, MetaRelationN and MetaChoice.

The method getMetaFields() on a MetaClass returns an array of the MetaFields belonging to that class (but not the ones belonging to superclasses); the method getAllMetaFields() on a MetaClass returns an array of all the MetaFields belonging to that class and to all its superclasses.

The method compositeIsAllowed(MetaClass compositeMetaClass) on a MetaClass returns true if a DbObject of this class can have as composite a DbObject of the class passed as parameter, false otherwise.

2.2 Model Management

One important aspect of the meta model management is that all the sources for the model's classes (with a few exceptions) are generated and should not be changed manually. Classes contained in JACK are not generated and thus can be edited manually. These classes are the super classes or root classes in the graph. The 'genmeta' plug-in is used to generate all the java files for the model.

Domains (a subtype of SrType) enumerate a list of pre-determined values. For instance a domain called GeoDirections would enumerate North, South, East and West as pre-determined values. Domains are referred in the meta-model (they can be used to type meta-fields), but they are not generated. Domains must be located in the db.srtypes package.

2.2.1 Editing the Model

The model is available in the ModelSphere format (it's a standard .sms binary file). Refer to the user guide for specifics on using ModelSphere.

Many properties refer to UDFs (User-Defined Field). The ModelSphere meta model leverages the ability provided by ModelSphere to create custom fields. This allows more detailed and complete specifications to be defined at the modeling level.

Note: UDF are called user-defined properties in the user interface.

Classes

Naming conventions

> The meta model must be defined in the package 'xxx.db', where 'xxx' is the name or abbreviation identifying the tool (for object modeling, the package will be oo.db).
>
> All classes in the meta model must inherit from jack.baseDb.db.DbObject and have their name beginning with 'Db'.
>
> A class named DbXXXProject must be created in the meta model. This class inherits from jack.baseDb.db.DbProject. The project instance is contained in the graph's root and is the entry point in the graph to access semantic objects.
>
> A class DbXXXSemanticalObject inheriting from jack.baseDb.db.DbSemanticalObject must also be created for the application. For ModelSphere, this class corresponds to DbSMSProject and DbSMSSemanticalObject.
>
> Note that these classes already exist for the ModelSphere application.

Properties

Behavioral

> Mark the class 'abstract', if not instantiable, and 'final', if it has no subclasses.
>
> Set the UDF 'cluster root' to **true,** if each object of this class will form a cluster with its descendants (for Objectivity). If this UDF is set on a superclass, its effect is propagated to all the subclasses. Note: This property was defined to support persistence using the Objectivity object database system and is no longer supported.
>
> Set the UDF 'naming root' to **true,** if this class is the root for building qualified name; the qualified name begins with the children of this class (in the composition hierarchy). If this UDF is set on a superclass, its effect is propagated to all the subclasses.
>
> Set the UDF 'matchable' to **true,** if the class has a semantic identifier that findMatchingObject can use to match objects between two projects. This is used during copy/paste or integration operations between projects. If you specify 'matchable' = true, you must define the method matches(dbObj), which returns true if <this> and <dbObj> have the same semantic identifier; the method can assume that the semantic parents of <this> and <dbObj> have the same semantic identifier (the semantic parents are those obtained from the relations returned by getDependencyRelations). For a DbSemanticalObject, there is a default implementation of matches(dbObj) which simply compares the names of <this> and <dbObj>; you must override matches(dbObj) if the semantic identifier is not formed of the name only. If this UDF is set on a superclass, its effect is propagated to all the subclasses.
>
> Set the UDF 'no UDF' to **true,** if you do not allow user defined properties on this class. This will prevent users from creating UDF for instances of that class. If this UDF is set on a superclass, its effect is propagated to all the subclasses.
>
> Set the UDF 'access control' to **true,** if you want write-access control on the objects of this class. This control is performed by validating privileges on the login. If this UDF is set on a superclass, its effect is propagated to all the subclasses. Note that this UDF was implemented for supporting a shared model repository like the Objectivity ODBMS.

Set the UDF 'icon name' to the name of the image file (png, gif, jpg) containing the icon representing the instances of this class. This UDF may be set on a superclass and is inherited by all the subclasses that do not have their own icon.

Define the variable serialVersionUID initialized to zero. If you forget to define this variable, at the first modification in the class, you will get an error when opening a model saved (serialized) before the modification. When this happens, assign the first long integer value that appears in the error message thrown by the de-serialization process to serialVersionUID; this will allow to open the model.

Localized Properties

Put in 'alias' the English GUI name. The GUI name must be in title form (first letter of words in upper case); if the plural is irregular, you must specify the singular and the plural forms, separated by ';'. 'fr_alias' contains the equivalent value for the French locale.

Fields

The types allowed for an ordinary field (i.e. non relation field) are:

- A primitive type.

- String.

- A type that inherits from SrType (Note: SrVector does not inherit from SrType and cannot be used).

Any other type is not allowed; arrays are not allowed.

If the field must be initialized in the constructor, specify the initialization expression in "initial value".

Behavioral Properties

Set the UDF 'copy links' to **true** on a relation field if the link is to be copied from the source to the copied object in a deepCopy operation. Note that links are always copied when the two linked objects are copied; so this UDF only concerns links between a copied object and a non-copied object.

> *Warning: This UDF can only be set on relations that allow any objects in the project to be linked together; for example, the relation "DbColumn.fType" is "copy links", because a column can be linked to any type of the project; on the other hand, the relation "DbConstraint.fColumns" is not, because a constraint can be linked only to columns of its table.*
>
> *If you want to set this UDF on relations that have some restrictions on objects that can be linked, you must check these restrictions in the DeepCopyCustomizer object passed to deepCopy.*

Set the UDF 'huge relation N' to **true** on a relation N that may become very big, and whose order is not controlled by the user. If this UDF is **true**, the relation N is represented by a DbHugeRAMRelationN, which is a scalable Set, instead of a DbRAMRelationN, which is an array List. DbHugeRAMRelationN is much faster than DbRAMRelationN for large collections, but it does not preserve the order of the elements.

The UDF 'screen order' allows to change the order of the fields in the properties screen. By default, the properties screen displays the fields in the order they appear in their respective class, from the superclass to the subclass. If you need a subclass field to appear before or after a superclass field, you set the UDF 'screen order' in the subclass field in this way: enter the character '<' (before) or '>' (after), followed by the name of the superclass field. Note that all the fields (in all the subclasses) following the moved field will also be moved with it, up to the next field specifying a value for this UDF. In other words, the directive given by this UDF applies to a range of fields, from the current field to the next field specifying a directive.

If you want to specify a special renderer / editor for this field, set the UDF 'plugin name' to the name of the renderer / editor. Example: the value 'FileName' in the UDF means to use the renderer FileNameRenderer and the editor FileNameEditor for this field. FileNameRenderer may not exist, in which case an appropriate default renderer is used (DefaultRenderer for a normal field, DbSemanticalObjectRenderer for a relation field).

It is possible to specify a name for the renderer and a different name for the editor; the two names are separated by ';'. For example, the field <DbORProcedure.javaMethod> has the following value for this UDF: "DbSemObjFullName;DbJVMethod", which means to use DbSemObjFullNameRenderer as renderer, and DbJVMethodEditor as editor.

If you do not specify a plug-in name for a field, the editor having the type of the field + the suffix 'Editor' as its name is used. If it does not exist, the field is not editable. The renderer has the same name as the editor, but with the suffix 'Renderer'. If it does not exist, the default renderer is used.

For a multi-line text, set the UDF 'plugin name' to 'LookupDescription'. Refer to the screen section of this document for details on available editor and renderer and how to define a new one.

Set the UDF 'hide on screen' to **true,** if this field must not appear in the properties screen.

Set the UDF 'not editable' to **true,** if this field appears in the properties screen, but is not editable.

Set the UDF 'integrable' to **true,** on a relation N, if you want this relation N to appear in the integration property tree.

Set the UDF 'write check' to **true** on a relation N, if the modification of this relation N requires the user to have write access to the object (by default, the modification of a relation N does not require write access).

NOTE: On an N-N association, you should normally set 'write check' to **true** on at least one side of the association; otherwise, modification of the association will always be allowed regardless of access restrictions.

Set the UDF 'no write check' to **true,** if the modification of this field does not require write access (by default, the modification of any field that is not a relation N requires write access).

Localized Properties

Put the GUI name in 'alias'. The GUI name must be in title form (first letter of each word in upper case). 'fr_alias' contains the equivalent value for the French locale.

Methods

If required for the class, override the methods getSemanticalName and getSemanticalIcon. This can be useful, if the name and/or icon vary depending on the instances instead of being the same for all instances of that class.

The setXXX methods may be overridden, but not the getXXX methods. In 1-1 and N-N relations, setXXX is defined on both sides of the relation; so you must override both sides. The best way to do that is to make the setXXX on one side call the setXXX on the other side, and put the real code in the latter setXXX.

You must override the method getMinCard(MetaRelationship), if you have relations in the class whose minimal cardinality is not fix, but depends on the state of the object. For example, you may have a single class representing ordinary fields and relation fields; the relation between <field> and <associationEnd> has a minimal cardinality depending on the type of the field: 0 for an ordinary field, 1 for a relation field. The method getMinCard is used by the method remove() to propagate the remove along the relations having a minimal cardinality constraint.

You must override the method remove(), if you need to propagate the remove to related objects that will not be removed automatically by the minimal cardinality constraint (as defined by the method getMinCard). In the overriding method remove first the related objects that will not be removed automatically, then call super.remove(); check all the 'get' on relations you use in the overriding method for the null value, because the linked object may already have been removed.

The method getDependencyRelations() returns an array of the MetaRelation1's constituting the semantic dependency. The default implementation returns the relation <DbObject.fComposite> as the only dependency. This method must be overridden for classes with multiple dependencies. When overriding this method in the list of dependency relations, specify those relations having the least children first, in order to optimize lookups.

The method hasWriteAccess() checks if the user is allowed to modify this object. The default implementation goes up the composition hierarchy to the first ancestor that has access control, and checks if the user has write access to this ancestor. You may override this method to change the default behavior.

General Rules

Composite

Abstract classes should not have a composite. Composites should only be specified for leaf classes in the graph.

Associations

Associations can be specified on leaves or any super classes. The general rule is to add the association at the top most level without permitting invalid objects to be linked. Ensure that the generic set method will only accept valid objects to be associated.

2.2.2 Generating the Model Classes

The 'genmeta' (meta generator) plug-in is used to generate the java source files for the entire model. This plug-in needs to be installed first.

The generation process generates the class files in "xxx.db" and DbResources in "xxx.international" ("xxx" is the name of the tool).

Sample meta generated source file

The following example shows the generated code for the class DbORColumn (this class defines a column of a relational table).

```
package org.modelsphere.sms.or.db;

import java.awt.*;

import org.modelsphere.jack.baseDb.db.*;
import org.modelsphere.jack.baseDb.db.srtypes.*;
import org.modelsphere.jack.baseDb.meta.*;
import org.modelsphere.jack.baseDb.util.*;
import org.modelsphere.sms.SMSFilter;
import org.modelsphere.sms.db.*;
import org.modelsphere.sms.db.srtypes.*;
import org.modelsphere.sms.or.db.srtypes.*;
import org.modelsphere.sms.or.international.LocaleMgr;


/**
<b>Direct subclass(es)/subinterface(s) : </b><A
HREF="../../../../../com/silverrun/sms/or/oracle/db/DbORAColumn.html">DbORAColumn</A>, <A
HREF="../../../../../com/silverrun/sms/or/ibm/db/DbIBMColumn.html">DbIBMColumn</A>, <A
HREF="../../../../../com/silverrun/sms/or/informix/db/DbINFColumn.html">DbINFColumn</A>,
<A
HREF="../../../../../com/silverrun/sms/or/generic/db/DbGEColumn.html">DbGEColumn</A>.<br>

    <b>Composites : </b>none.<br>
    <b>Components : </b><A
HREF="../../../../../com/silverrun/sms/db/DbSMSObjectImport.html">DbSMSObjectImport</A>.<b
r>
 **/
```

A group of packages is systematically added during the generation process to ensure that all domains and referenced types can be accessed by the compilation unit. Generated comment includes references to subclasses and sub interfaces. A reference to the composite class is also specified. (Note that in this case, the class is abstract and thus does not have a composite).

```
public abstract class DbORColumn extends DbORAttribute {

  //Meta

  public static final MetaField fNull
    = new MetaField(LocaleMgr.db.getString("null"));
  public static final MetaField fForeign
    = new MetaField(LocaleMgr.db.getString("foreign"));
    = new MetaField(LocaleMgr.db.getString("isAssociation"));
  ..

  public static final MetaClass metaClass = new MetaClass(
    LocaleMgr.db.getString("DbORColumn"), DbORColumn.class,
    new MetaField[] {fNull,
      fForeign,
      ..
```

The meta data specification contains a static field declaration for each property in the model classes. Specialized MetaFields exist for associations. A meta data specification is also specified for the model classes. All the meta data related information are declared as public and static, thus they can be accessed by all the features and do not require specific objects. The meta class field is always named metaClass. All the meta fields start with the letter 'f'.

Meta Initialization Method

The initMeta() method is called once during initialization after all classes in the meta model have been loaded.

```
/**
   For internal use only.
**/
 public static void initMeta() {
   try {
     metaClass.setSuperMetaClass(DbORAttribute.metaClass);
     metaClass.setIcon("dborcolumn.gif");

     fNull.setJField(DbORColumn.class.getDeclaredField("m_null"));
     fForeign.setJField(DbORColumn.class.getDeclaredField("m_foreign"));
       fForeign.setEditable(false);
     ..
   }
   catch (Exception e) { throw new RuntimeException("Meta init"); }
 }
```

> *It is possible to specify additional initialization code in the model. But adding Java statements directly in the meta-model can be costly from a maintenance point of view because it requires changes to the meta-model each time a change is required in the code. To avoid this situation, initialization code can be placed in the DbInitialization class (or other package equivalence) if the code is prone to changes.*
>
> *All Db objects define a setDefaultInitialValues() that contains statements to execute when the Db object is created. The content of that method in defined in the meta-model and is generated using default values specified on the fields. Additional initialization can be delegated to an external method contained in a non generated class like DbInitialization.*

Serialization Specification

All classes use a serial ID of 0 to avoid any conflicts with older serialized versions of a project.

```
static final long serialVersionUID = 0;
```

Field Declaration.

There is a corresponding meta field for each field. The fields are always prefixed with 'm_'.

```
//Instance variables
boolean m_null;
boolean m_foreign;
```

Constructors

There are two constructors. The parameterless constructor should not be used. It is required to provide a composite object when creating new objects. The composite object is checked to ensure that it is a valid composite for the object.

```
   //Constructors

/**
    Parameter-less constructor.  Required by Java Beans Conventions.
 **/
  public DbORColumn() {}

/**
    Creates an instance of DbORTable.
    @param composite the object which will contain the newly-created instance
 **/
  public DbORColumn(DbObject composite) throws DbException {
    super(composite);
    setDefaultInitialValues();
  }
```

Default Initial Values

The setDefaultInitialValues() method initializes the fields using the default values specified in the model.

```
private void setDefaultInitialValues() throws DbException {
    setNull(Boolean.TRUE);
    setReference(Boolean.FALSE);
    DbORDataModel dataModel = (DbORDataModel)getCompositeOfType(DbORDataModel.metaClass);
    TerminologyUtil terminologyUtil = TerminologyUtil.getInstance();
    Terminology term = terminologyUtil.findModelTerminology(dataModel);
    setName(term.getTerm(metaClass));
    fNull.setRendererPluginName("Boolean");
}
```

Specific Setter Methods.

There is one specific setter method generated for each field defined in a class of the meta-model. The setter name is the concatenation of 'set' and the name of the field, as defined in the meta-model.

```
  //Setters

public final void setNull(Boolean value) throws DbException {
    basicSet(fNull, value);
}

public final void setForeign(Boolean value) throws DbException {
    basicSet(fForeign, value);
}
```

Specific Getter Methods.

There is at least one specific getter method for each field. The getter name is the concatenation of 'get' and the name of the field. If the field is a Boolean value, an additional getter is generated ('is' plus the name of the Boolean field).

```
  //Getters

public final Boolean getNull() throws DbException {
    return (Boolean)get(fNull);
}

public final Boolean getForeign() throws DbException {
    return (Boolean)get(fForeign);
}

public final boolean isNull() throws DbException {
    return getNull().booleanValue();
```

```
  }

  public final boolean isForeign() throws DbException {
    return getForeign().booleanValue();
  }
}
```

Generic Setter and Getter Methods.

This is one generic setter and one generic getter per class. These methods require a MetaField parameter. The generic setter appears in the generated model class, and the generic getter is defined in the DbObject and inherited by all the generated model classes.

```
public void set(MetaField metaField, Object value) throws DbException {
  if (metaField.getMetaClass() == metaClass) {
    basicSet(metaField, value);
  }
  else super.set(metaField, value);
}

//defined in DbObject
public final Object get(MetaField metaField) throws DbException {
  return toApplType(metaField, basicGet(metaField));
}
```

Generic Setter Method for Relationships.

Verification is performed on the field to ensure that it is valid for the class. On the parent side of association N, the set is dispatched on the value object.

```
/**

 **/
 public void set(MetaRelationN relation, DbObject neighbor, int op) throws DbException {
   super.set(relation, neighbor, op);
 }
```

Defining Custom Methods in the Meta-Model.

When developers edit the meta-model and add new fields in a class, the genmeta plug-in automatically the specific getter and setter. Besides fields, developers may also define methods of the class in the meta-model; the genmeta plug-in will generate methods as they are defined in the meta-model.

Developers must be very careful when they add methods manually in the meta-model. The genmeta plug-in generates the methods "as is": if the body of a method is empty (or if it contains syntax errors), the genmeta plug-in does not perform any Java syntax validation and will generate erroneous model classes. For this reason, it is recommended to avoid methods whose body contains a lot of Java statements. When applicable, the body just call a method in a utility class, without other statements.

Guideline #12: When developers define methods in the meta-model, their body should stay as simple as possible.

Developers can define the generic set() method in the meta-model, even if it's normally an automatically generated method. The genmeta plug-in check if there is a set() method defined in the meta-model; if no set() method if found, it generates the set() method automatically.

When developers define a set() method in the meta-model, it's their responsibility to maintain this method. If new fields are added to the class, developers must change the body of the set() method in the meta-model to ensure the set() method handles the new fields properly. Because defining a set() method directly in the meta-model required more maintenance effort, it is not recommended to do so.

3 ModelSphere Supported Concepts

3.1 Top-Level Containers

Top-level containers can be created directly under a DbSMSProject instance. The following table displays the different top-level containers supported in ModelSphere:

Top-Level Containers	Description
Behavioral Model	*Contains process units, data stores, flows and external entities.* *Can have 0 or many graphical representations.*
Class Model	*Contains classes and packages.* *Can have 0 or many graphical representations.*
Data Model *Generic Data Model* *Oracle Data Model* *Informix Data Model* *IBM Data Model*	*Regroups tables and views composing the relational model.* *Can have 0 or many graphical representations.* *Associated to 1 target system.*
Type Model *Oracle Type Model* *Informix Type Model* *IBM Type Model*	*Contains data types.* *Can have 0 or many graphical representations.* *Associated to 1 target system.*
Built-In Type Node	*Contains built-in data type for target systems.* *No graphical representation.* *Associated to 1 target system.*
User-Defined Package	*User-defined node.* *Used to regroup/structure models and packages.* *No associated target system.*
Database *Oracle Database* *Informix Database* *DB2 Database*	*Physical objects defining a database.* *Contains graphical objects associated to a database.* *No associated target system.*
Operation Library *Generic Operation Library* *Oracle Operation Library* *Informix Operation Library* *DB2 Operation Library*	*Contains procedures/functions (code).* *Associated to 1 target system.*

Top-Level Containers	Description
Event Model	*Regroups event categories: Request, Insert, Update et Delete.* *No associated target system.*
Domain Model	*Regroups domains which are used as data types.* *No associated target system.*
Common Item Model	*Regroups common items.* *No associated target system.*

Table 7: ModelSphere Top-Level Containers

3.2 Major Decomposable Components

It would be useless to list all the components here that a ModelSphere model could contain. Consult the javadoc to get the complete list. Here we describe the major components that can contain other components.

Decomposable Components	Description
Process units	*Process units may be decomposed into subprocesses.*
OO packages	*Packages may contain nested sub packages.*
OO classifiers	*Classifiers (classes and interfaces) may contain inner classifiers.*
User-defined packages	*User-defined packages may contain nested user-defined packages as components.*

Table 8: ModelSphere Decomposable Components

3.3 A Short Comparison Between EMF and the DB Framework

Modeling framework is an advanced topic in software engineering and may be new for most of the readers. People who already have an experience with modeling frameworks probably know EMF (Eclipse Modeling Framework), one of the rare open-source modeling frameworks. This section makes a short comparison between EMF and DB, to allow developers who already know EMF to rapidly comprehend and make correlations with equivalent concepts in the DB framework.

EMF Top-Level Concepts	Equivalent Concepts in DB
EPackage	ApplClasses
EClass	MetaClass
EDataType	SrType
EEnum	Domain

Table 9: EMF and DB Top-Level Concepts

ApplClasses is not modeled but it is always generated; it allows the users to list all the meta-classes of a given package. There is one ApplClasses generated in each .db package. All the model fields are typed with a built-in type or a SrType (or a subclass of it). Domains (a subtype of SrType) enumerate a list of pre-determined values. For instance a domain called GeoDirections would enumerate North, South, East and West as pre-determined values. As opposed to EMF, SrType and Domain subclasses are not modeled, but are manually added to a db.srtypes package.

EMF Structural Concepts	Equivalent Concepts in DB
EAttribute	MetaField
EReference	MetaRelation
EAnnotation	No equivalent

Table 10: EMF and DB Structural Concepts

MetaField is very similar to EAttribute: A MetaClass instance may return the list of MetaField defined for this class, and MetaField objects may be used as parameter of the generic setter and getter method. In EMF and DB, references are one-way, but can be associated to another reference as opposite reference. In EMF, minimum and maximum cardinalities of references can be any integers, while MetaRelation is subclassed to MetaRelation1 or MetaRelationN to indicate a one-to-one or a one-to-many relationship.

EClass Properties	MetaClass Properties
ESuperTypes	superMetaClass and subMetaClasses[]
EAttributes, EReferences	MetaFields[]
No equivalent	GUIName, icon
isAbstract	No equivalent
isInterface	No equivalent
defaultValue	No equivalent

Table 11: EMF and DB Meta Class Properties

EMF supports multiple inheritance while DB does not; DB allows a MetaClass to return its subclasses, while this is not possible in EMF. EMF has a method which returns all the attributes and a second method which returns all the references; by contrast DB provides only one method, getMetaFields(), and it is up to the programmer to distinguish MetaRelation from ordinary MetaField.

DB allows to define a GUI name and an icon for a meta-class; EMF users may use the standard Java Beans methods (getDisplayName() and getIcon()) for the same functionality.

The abstract and interface boolean values do not exist for MetaClasses: anything generated is a class, never an interface.

EAttribute Properties	MetaField Properties
No equivalent	GUIName
isID	No equivalent

isChangeable	editable
isVolatile	No equivalent
isTransient	No equivalent
defaultValue and defaultValueLiteral	No equivalent
isUnsettable	No equivalent
IsDerived	No equivalent

Table 12: EMF and DB Meta Field Properties

If an EAttribute is not changeable, the setter method is not generated for this attribute. In DB, all meta-fields have a getter and a setter, but a non-editable field cannot be modified in the Design Panel (even if it can be programmatically).

EMF Default Types (in org.eclipse.emf.ecore)	DB Default Types (in org.modelsphere.jack.baseDb.db.srtypes)
EBoolean EByte EDouble EFloat EInt ELong EShort EString	SrBoolean SrColor SrDouble SrFont SrInteger SrLong SrPoint SrString

Table 13: EMF and DB Default Types

3.3.1 Generated Code

Both EMF and DB provide a code generation feature from the model. This is the main advantage of using a modeling framework. In the two frameworks, getters and setters are generated for each EAttribute or MetaField, in addition to the generic getter and setter. In both frameworks, it is possible to access meta-data on the class and attribute concepts.

The classes generated by EMF follow the bridge design pattern[7]: for each modeled EClass, EMF generates an interface extending EObject, and a class extending EObjectImpl. By contrast in DB, a modeled class only gives one implementation class, which extends DbObject.

For each modeled package, EMF generates a factory class, that inherits from EFactory. This class gathers all the factory methods, named createXXX(), where XXX is the name of the class in the EMF model. DB does not generate factory methods, but generate standard constructors and makes sure that the composite object is passed as parameter in the constructor. In DB, it is impossible to create a component before its composite, because the composite parameter is required to construct the component.

In EMF, when a parent returns the list of its children, the getter method returns an EList<X>, where X is the type of its children. EList is a generic type extending java.util.List. This allows the developer to obtain the size of the list (by calling list. ..) before getting the list iterator. In DB, a parent returns the list of its children by returning a DbRelationN, from which it is possible to get its size and a DbEnumeration, that inherits from java.util.Enumeration. EMF supports generics and enumerations introduced in Java 1.5; while DB does not.

---

7   See Design Patterns, [Gamma95].

While EMF does not force anything, DB forces to call getters within a read transaction, and to call setters within a write transaction. This simplifies model integrity, and undo/redo support. The transaction support is the main advantage of DB over EMF[8].

The DB framework allows a piece of code to be notified when changes occurs in the DB model, in which case a DbListener listens for model changes. EMF provides the same kind of notification.

---

8   A transaction component in EMF now supports the concept of transaction: http://www.eclipse.org/modeling/emf/

CHAPTER 7 - GRAPHICS / DIAGRAMMING

The graphics module is the framework responsible for the drawing of diagrams and diagram elements, and for the interaction with the user who manipulates diagram elements. The source code of this module is located in the org.modelsphere.jack.graphic package.

The Graphics/Diagramming module heavily depends on the Graphics class in java.awt package. The Graphics class contains methods to draw lines, rectangles, polygons and texts. It is recommended to get familiar with this class first before trying to add and to modify the classes of the graphics modules. The Graphics/Diagramming module also depends on the modeling framework (the classes in jack.baseDb whose name starts with Db, such as DbObject), but does not depend of the meta-model.

1 Diagram Views

Diagram views are the entry points of the graphics module. The class DiagramView is a subclass of the Swing's JPanel. The DiagramView constructor requires a Diagram object as parameter and is responsible to paint it (see its paintComponent() method). A diagram view paints graphical components according to a zoom factor. By default, the zoom factor is equal to 1.

The DiagramView class has two specialized subclasses: MagnifierView and OverviewView. These classes are responsible to display the magnifier and the overview windows on the active diagram view.
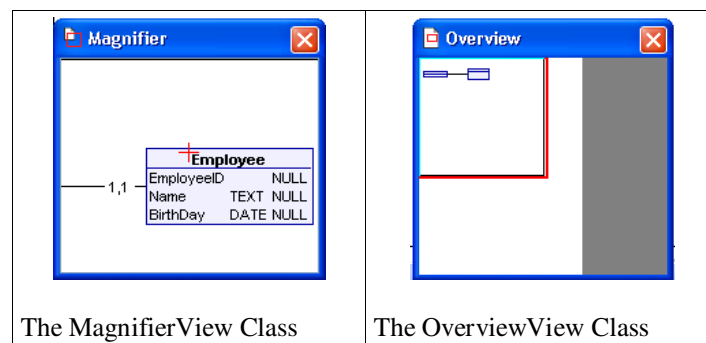
Figure 6: Magnifier and Overview Views

2 Graphic Components

An important point to understand the graphics module is to distinguish between semantic objects, graphical objects and graphic components.

A semantic object corresponds to a model element (for instance a relational table or column, or an object-oriented class or field). A semantic object can be rendered by several graphic representations, or by no graphic representation at all. A semantic object inherits from DbObject, and consequently is saved (serialized) in a .sms file. See DbSemanticalObject for more details.

A graphical object corresponds to the graphical representation of a semantic object, a graphical object has one and only one semantic object. A graphical object belongs to a diagram, while a semantic object is not attached to any diagram. A graphical object also inherits from DbObject, and consequently is saved in a .sms file (it is necessary to save graphical objects in the .sms file, otherwise the graphical attributes (position, color, etc.) of a object would be lost each time the user closes the application). Also, a graphical object being a DbObject, modifications on the graphical object can be undone/redone. See DbGraphicalObject for more details.

A graphical object is independent of the Swing library, but has a graphic peer, called the graphical component. The GraphicComponent is the central class of the org.modelsphere.jack.graphic package. Anything displayed in a ModelSphere diagram is a graphical component.

A graphical component has several attributes: it has a line color (the color used to draw its borders), a fill color (the color used to paint its contents, and a text color (the color used to paint texts of the component). A graphical component can be selected or not; when it is selected it has the responsibility to draw selection handles. A graphical component has a Boolean autofit attribute. When autofit is true, the component determines itself its size in order to display all its components; when autofit is false, the size is determined by the ModelSphere user. When autofit is true, the selection handles are grayed out, indicating that the user cannot resize the shape.
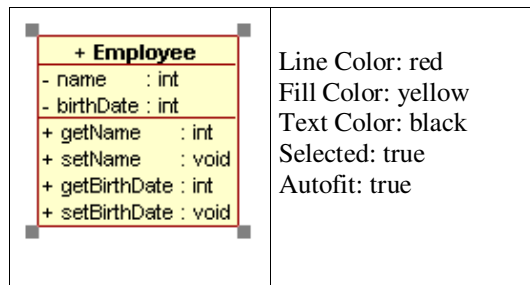
Figure 7: A Graphical Component and some of its Attributes

A graphical component has also a layer, the layer determines the order of painting of this component: the diagram starts painting the background layer components, and ends by top-level components.

The GraphicComponent has some important methods: the paint() method is responsible to paint the component. The getRectangle() method returns a rectangle representing the size and the location of the component; if the component is not actually a rectangle (for instance, it can be an oval or a polyline), the method returns the smallest rectangle that contains the entire graphical component. The contains() method determines if the point XY is contained within the graphical components. Finally, the delete() method is responsible to remove the component from a diagram.

The constructor of a graphical component requires two parameters: the diagram to which it belongs, and the graphical shape used to represent it graphically. Diagrams are graphical shapes are discussed below in two separate sections.

3 Diagrams

Diagrams are containers of graphical components. They correspond to the class Diagram in the org.modelsphere.jack.graphic package. Diagrams implement the Pageable and Printable interfaces of java.awt. The paint() method paints diagram components by beginning by the first layer, conforming to the Painter's algorithm[9].

---

9 See http://en.wikipedia.org/wiki/Painter's_algorithm

Constant	Value
Diagram.LAYER_BACKGROUND	*0*
Diagram.LAYER_FREE_GRAPHICS	*1*
Diagram.LAYER_LINE	*2*
Diagram.LAYER_LINE_LABEL	*3*
Diagram.LAYER_GRAPHIC	*4*

Table 14: Diagram Layers

The order of painting is illustrated by the following figures.

Lines are drawn (layer 2), and then graphical boxes (layer 4), hiding the lines under the boxes.	If box transparency is set to 40%, lines drawn on layer 2 appear under the boxes.
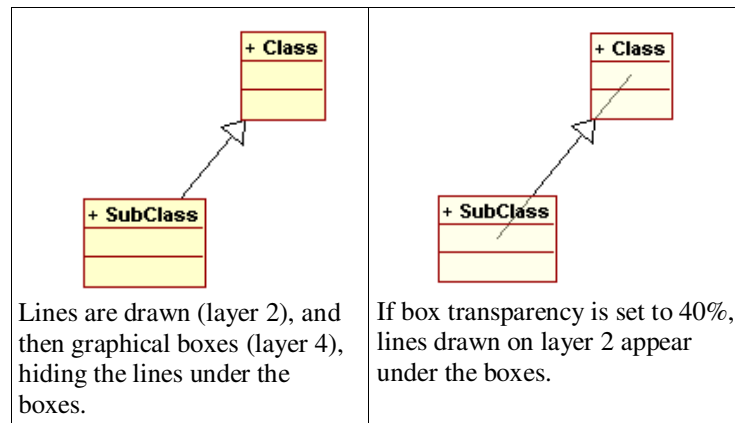
Figure 8: The Order of Painting by Layers

The Diagram.layers variable is a vector of five layers, each layer being a vector of GraphicComponent. When a new graphical component is created, it is added at the end of the vector corresponding to its type (for instance, when a new line is created in the diagram, the Line graphical component is added to the line layer vector). Consequently, when two objects are in the same layer, the order of creation determines the order of painting (objects created first are drawn before objects created after).

Showing nested packages is possible if the javax package was created before the Swing package.	If the Swing package was created before the javax package, the Swing package will always be painted first, and will appear under the second package. There is no "Bring to Top" command.

Figure 9: The Order of Painting for Two Components on the Same Layer.

There is neither "Bring to Top" nor "Push Back" command to change the order of painting. Such a command could be implemented by changing the order of GraphicComponent within the graphical component vector.

4 Nodes and Lines

A ModelSphere diagram is a specialized form of graph, composed of nodes and lines (or edges)[10]. Relational tables, object-oriented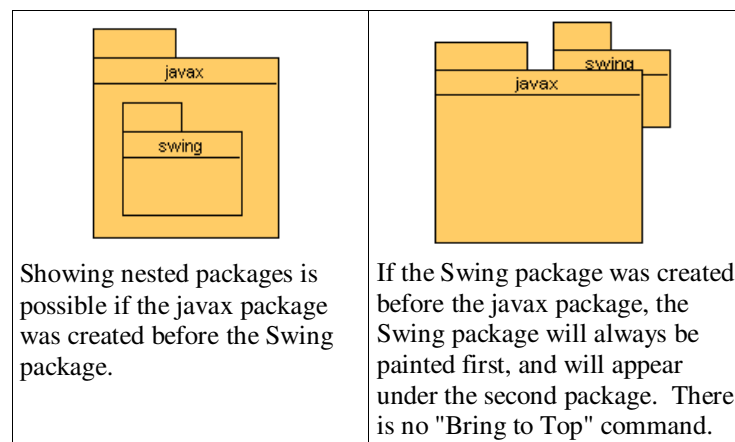 classes represent the nodes of the graph, while associations (and inheritance links in object-oriented diagrams) represent the lines. Nodes are implemented by the GraphicNode class and edges by the Line class, both in the org.modelsphere.jack.graphic package. Nodes and lines are subclasses of GraphicComponent.

The GraphicNode class is a graphical component that may have lines connected to it. When a graphical node is deleted, it has the responsibility to delete lines connected to it.

A Line is connected to two graphical nodes (or twice to the same graphical node, in which case it is called reflective). The Line's constructor requires four arguments: the diagram to which it belongs, the two graphical nodes to which the line is connected, and a polyline for intermediate points. If the polyline is empty, the line between the two nodes will be straight.

ImageComponent is a special subclass of GraphicComponent; it represents an arbitrary image displayed on the diagram.

5 Zones

Usually a graphical node is more than a rectangle with a single text in it; a graphical node is subdivided into several compartments, called zones. A relational table has at least two compartments, the name zone and the column zone. A object-oriented class has generally three compartments, the name zone, the field zone and the method zone.
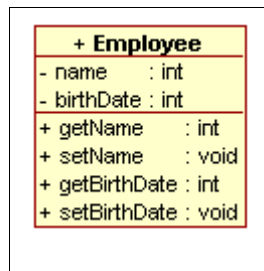
Figure 10: The Name Zone, the Field Zone and the Method Zone

A graphical component that can be divided in zones in a ZoneBox. Because GraphicNode contains zones, it inherits from ZoneBox, which in turn inherits from GraphicComponent.

The ZoneBox class represents the graphical component divided in zones, while the Zone class represents compartments of the graphical components. A zone has its name. The Zone class is located in the org.modelsphere.jack.graphic.zone package.

There are two main kinds of Zone, SingletonZone and MatrixZone. A SingletonZone contains only one text. The name compartment is an example of a SingletonZone. A MatrixZone contains several zone cells (see the class ZoneCell in the org.modelsphere.jack.graphic.zone package). The field compartment of a object-oriented class is an example of a MatrixZone.

---

10 See http://en.wikipedia.org/wiki/Glossary_of_graph_theory

Note that a graphical object is considered selected, if one of its sub zones is selected.

6 Attachments

An attachment is a special case of ZoneBox attached to a graphical node. It moves when the graphical node to which it is attached moves. It is used in business process modeling diagrams only.
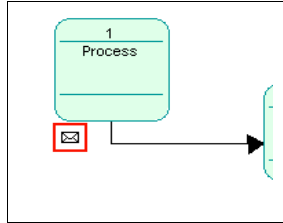
Figure 11: An Attachment (the Letter Icon) Linked to a Node (the Process)

7 Each Graphical Component has its Shape

The constructor of GraphicComponent requires a graphical shape as parameter, describing which shape to use to draw the graphical component. This is implemented by the GraphicShape interface in the org.modelsphere.jack.graphic.shape package.

This package defines several classes implementing the GraphicShape interface, such as RectangleShape or OvalShape. The following figure shows several shapes and the name of the class implementing that shape.

Figure 12: Classes in the org.modelsphere.jack.graphic.shape Package

8 Graphical Layout

The GraphicLayout class in the org.modelsphere.jack.graphic package is responsible to lay out graphical nodes in a diagram. It is possible to lay out only selected nodes, or the entire diagram. The layout algorithm finds a pivot (the node that has the most lines connected to it), places the pivot in the center and the other nodes in a circle around the pivot. The layout algorithm is not trivial, and further implementations details are given in the comments embedded in the GraphicLayout's code.
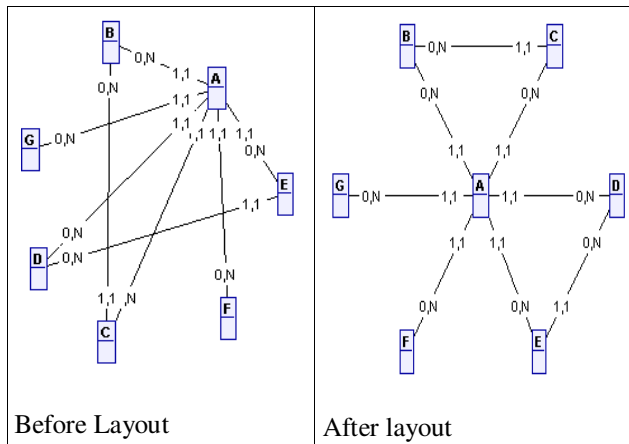

Figure 13: Before and After Performing a Graphical Layout

9 Review

The following figure shows a typical ModelSphere diagram, with the name of the classes discussed earlier in this chapter linked with the graphical elements. The classes are located in the org.modelsphere.jack.graphic package, and in the zone and shape subpackages.
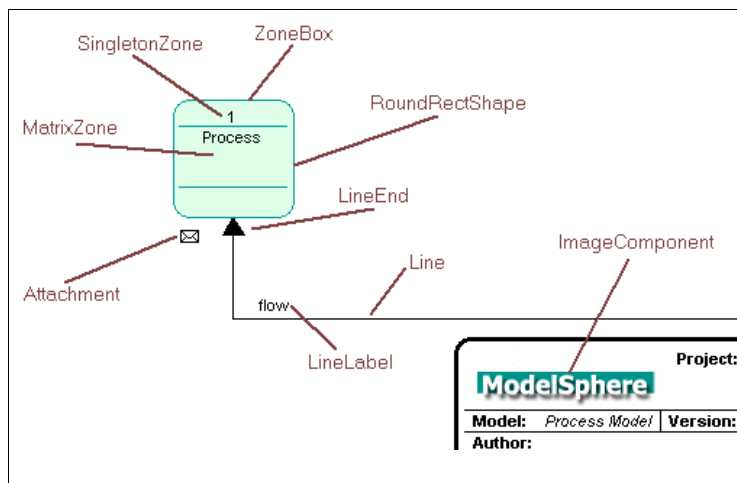
Figure 14: Nodes, Lines and Images

CHAPTER 8 - GUI DESIGN AND FEATURES

1 Overview

The ModelSphere application interface is divided in several GUI components, as illustrated below.
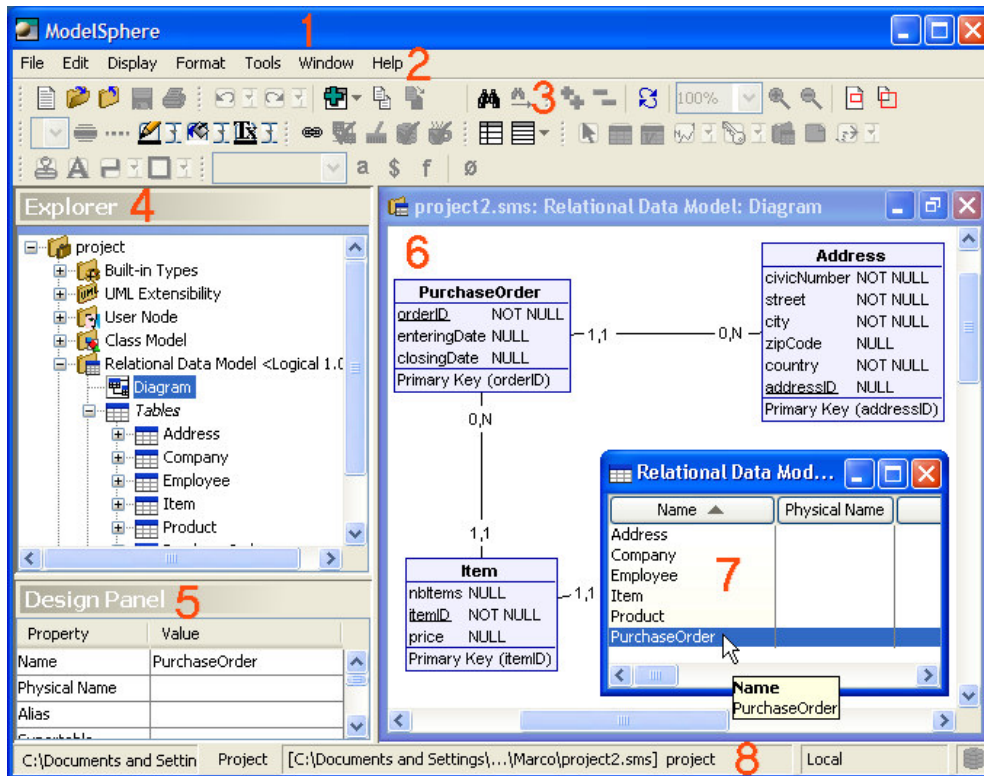
Figure 15: The ModelSphere Application Interface

Several GUI components are numbered in the figure above. The following list makes the association between the GUI widgets and the code where it is implemented.

1: The title bar of the main frame, implemented by the org.modelsphere.sms.MainFrame class.

2: The menu bar of the main frame, implemented by the org.modelsphere.sms.MainFrameMenu class.

3: The application tool bar, implemented by the org.modelsphere.sms.ToolsToolBar class.

4: The explorer view, implemented by the org.modelsphere.jack.srtool.explorer.ExplorerView class.

5: The design panel, implemenetd by the org.modelsphere.jack.srtool.screen.DesignPanel class.

6: The diagram, implemented by the org.modelsphere.jack.srtool.graphic.ApplicationDiagram class.

7: The list view, implemented by the org.modelsphere.jack.baseDb.screen.ListView.

8: The status bar, implemented by the org.modelsphere.sms.MainFrame class.

2 Focus Manager

In the previous figure, you may have noticed that the window identified by the number 7 is the window on which the user is currently working: its borders are darker compared to the borders of the other windows: this object has the user's focus.

ApplicationContext is a façade class that is the entry point of a set of services; it can be accessed from everywhere in the application. This class returns the focus manager, which is a singleton: there is only one focus manager instance during a ModelSphere session. When invoked, the focus manager returns the current object on which the focus is applied.

```
import org.modelsphere.jack.srtool.ApplicationContext;
import org.modelsphere.jack.srtool.FocusManager;

FocusManager manager = ApplicationContext.getFocusManager();
Object object = manager.getFocusObject();

if (object instanceof ApplicationDiagram) {
  ..
```

The object having the focus can be one of the following:

- org.modelsphere.jack.srtool.graphic.ApplicationDiagram: if the user is working on a diagram. See the previous chapter for more details on the diagramming.

- org.modelsphere.jack.srtool.explorer.ExplorerView: if the user is exploring the explorer at the right of the application. See the Explorer section in this chapter for more details on the explorer view.

- org.modelsphere.jack.baseDb.screen.DbDataEntryFrame: if the user is editing model values.

- org.modelsphere.jack.baseDb.screen.ListView: if the user is viewing a list, that was the case in the previous figure.

It's up to the developer to verify, by using the instanceof Java primitive, the actual type of the object having the focus.

The focus manager defines also a useful method, getSelectedObjects(), that returns the list of selected objects. As you can see in the code below, the focus manager supports multiple selection. If programmers want to perform an action on a single object only, it's their responsibility to verify the number of selected objects.

```
Object[] objects = manager.getSelectedObjects();

if (objects.length == 1) {
   ..
```

2.1 Focus Listener

The org.modelsphere.jack.srtool.CurrentFocusListener interface defines the currentFocusChanged() method. When a class extends CurrentFocusListener, it must implement the currentFocusChanged method.

```
import org.modelsphere.jack.srtool.CurrentFocusListener;

class MyAction extends AbstractApplicationAction
    implements CurrentFocusListener {

    //construct the action and register to the focus listener
    public MyAction() {
        super(..);
        ..
        focusManager.addCurrentFocusListener(this);
    }

    //implements CurrentFocusListener
    public void currentFocusChanged(Object oldFocusObject, Object focusObject) {
        //reacts to focus change
    }
}
```

Refer to the CurrentFocusListener interface, and examine classes that implement this interface to learn more about focus listener.

3 Explorer

The explorer view displays semantic elements of the model in a hierarchical structure. The root nodes represent the composite semantic elements, and the leaves represent the components.
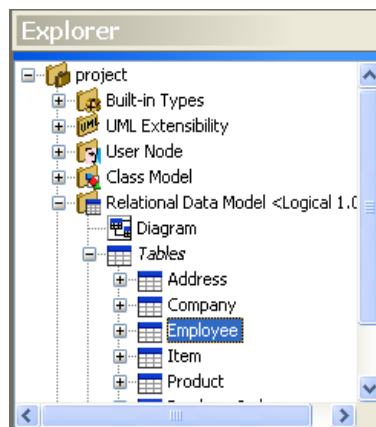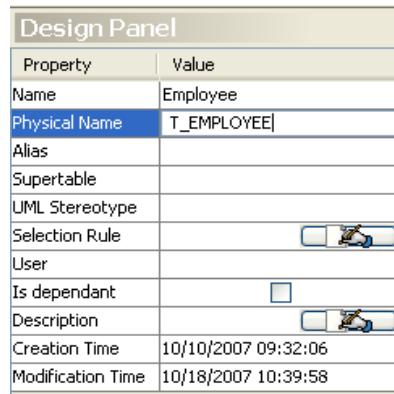
Figure 16: The Explorer View

The code implementing the explorer view is located in the class ExplorerView of the package org.modelsphere.jack.srtool.explorer; examine this class to learn more about the explorer view.

4 Design Panel

The design panel is a property sheet that reacts to focus changes, and shows the properties of the selected object (in the explorer or in a diagram). The properties are editable directly in the design panel.

Figure 17: The Design Panel

The code implementing the design panel is located in the class DesignPanel of the package org.modelsphere.jack.srtool.screen; examine this class to learn more about the design panel.

5 Workspace and Document Windows

The large panel under the tool bar and at the right of the explorer is called the Workspace (no. 3 in the figure below). The workspace contains one or several document windows. Lists (no. 1) and diagrams (no. 2) are different kinds of document windows.
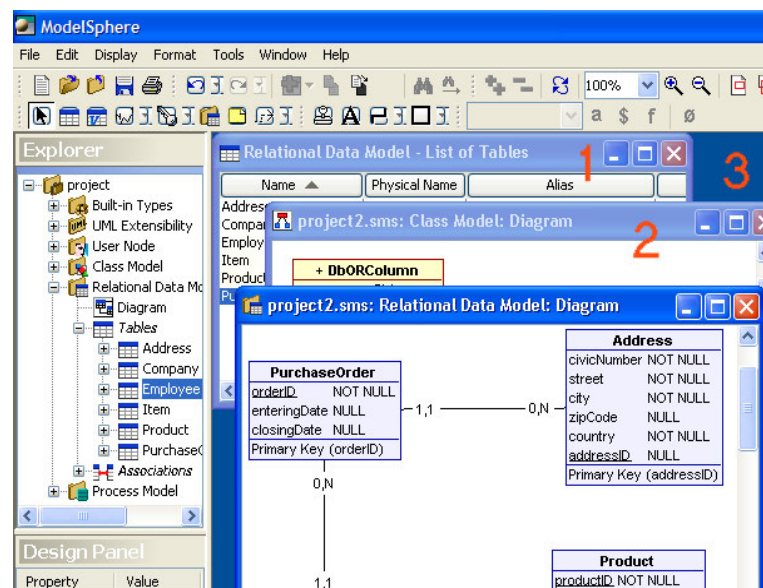
Figure 18: Workspace and Document Windows

6 Options Dialog (Preferences)

The Options Dialog is accessible in the ModelSphere interface by clicking **Tools→Options..**
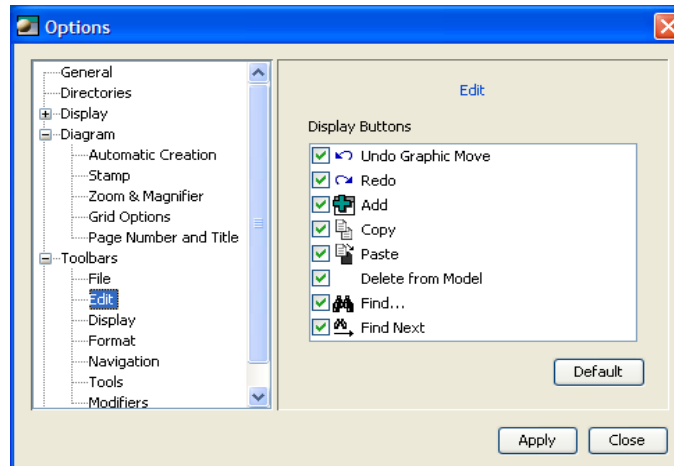
Figure 19: The Options Dialog

The dialog is implemented by the org.modelsphere.jack.preference.OptionDialog class. Because there are too many options in ModelSphere to show them in a single page, they are grouped in option groups. The tree explorer at the right of the dialog contains the option groups. They correspond to the classes contained in the org.modelsphere.sms.preference package. Examine the classes of this package to learn how to create new options and new option groups in ModelSphere.

7 Properties Screens

Properties screens are similar to the design panel. Users can open the properties of an object by selecting and right-clicking the object (in a diagram or in the explorer),
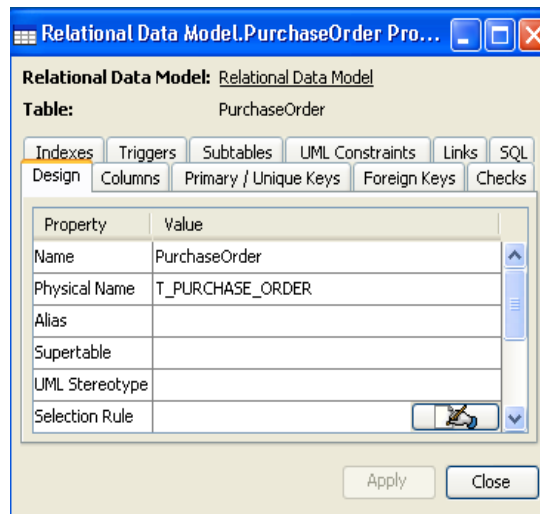
Figure 20: A Properties Screen

8 Threads

Java is a multi-thread programming language, and not surprisingly the ModelSphere application runs on several threads. Even if several threads are involved, it is important to call Swing elements only on the events dispatch thread.

Guideline #13: It is strongly recommended to avoid Swing changes on threads other than the events dispatch thread. The SwingUtilities.invokeLater(Runnable) method makes sure that changes to GUI elements are performed on the events dispatch thread.

It is allowed to perform non-GUI operations on the events dispatch thread, as long as these operations are fast to execute. Performing long-running operations on the events dispatch thread will slow down the GUI operations, and will make the application slow to react to the user's commands.

Guideline #14: It is recommended to avoid performing long tasks on the event dispatch thread. The Worker-Controller classes defined in JACK provide a mechanism to execute long-running tasks while providing feedback to the user.

➢ It is also important to remember that Db is not multi-threaded: you cannot start a transaction on a thread if another transaction is still active on another thread, otherwise Db will throw an exception. Refer to the section 1.3.1 of the chapter Modeling Framework (on page 25) for further details.

8.1 Worker-Controller

org.modelsphere.jack.gui.task.Worker: When a long task needs to be performed, this class is used to handle the task execution and provide a detailed progress monitoring dialog along with the controller.

Figure 21: Progress Dialog

9 Swing Action Extensions

9.1 What is an Action?

An action allows uniform execution of tasks in the GUI. The action specifies different properties like 'enabled'. The main action's goal is to regroup the execution of a task in one place. This approach allows adding features to unlimited GUI elements like menu and tool bars to enforce uniformity on all the widgets and task executions. Instead of adding a menu item in a menu, the action is added directly and the menu creates the menu item for representing the acti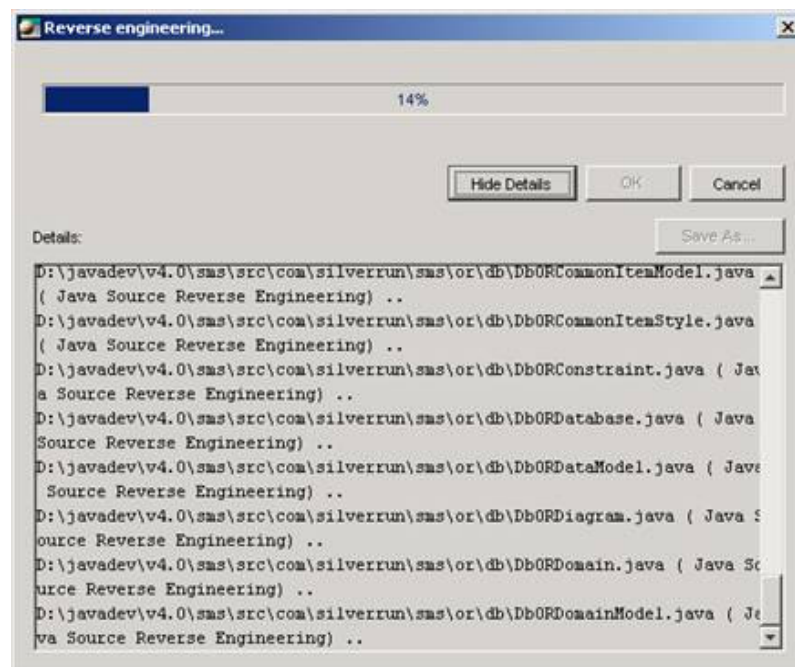on. It also ensures that the item is kept updated (enabled, text, ..), if the action's properties change by adding a PropertyChangeListener on the action. The menu item registers the action as ActionListener in order to execute the targeted task if activated (Swing actions ultimately implement AbstractAction).

Guideline #15: Do not use methods setIcon(), setText(), setEnabled(), setMnemonic(), setAccelerator() and setVisible() on the buttons and menu items. Instead the methods defined on Action should be used.

Swing action mechanisms will update the components automatically. Refer to the Swing documentation for more details on how actions work.

9.2 Basic Action Properties (Swing)

Action properties correspond to common widgets properties. The class 'swing.AbstractAction' supports the following properties:

Property	Description
Name	Used for text and tool tips.
Enabled	Used for enabled/disabled state.
Small_icon	Displayed icon in the GUI.
Short_description	Currently unused by swing. This property has been defined to store a string containing the tool tips.
Long_description	Currently unused by swing but supported by AbstractApplicationAction. This property is used to store a contextual help text. The text is currently displayed on the status bar.
Mnemonic_key	Store an integer representing a character.
Accelerator_key	Store a KeyStroke.

Table 15: Swing AbstractAction Properties

9.3 Extended Action Features in JACK

ModelSphere's actions must extend AbstractApplicationAction (or one of its subclasses). This class provides support for additional properties and exception handling on action executions. Adding support for new properties also requires specific menu (jack.awt.JackMenu), pop-up menu (jack.awt.JackPopupMenu) and tool bar (jack.awt.JackToolbar) classes. These actions containers must be used in order to make the extended features work. Extended properties are:

Property	Description
Long_description	Currently unused by swing but supported by AbstractApplicationAction. This property is used to store a contextual help text. The text is currently displayed on the status bar on mouseOver. Use method setHelpText(String) to change the value.
Mnemonic_key	Store an integer representing a character.
Accelerator_key	Activation KeyStroke. This property is fully supported in JACK.
Visible	Allows hiding of widgets created with the action. Use the method setVisible().

Table 16: JACK AbstractApplicationAction Properties

9.4 Using Listeners and Actions

It is common to add listeners on the actions in order to maintain their states (such as enabling or disabling depending on the active focus in the application).

ModelSphere contains many actions, which require to be updated based on active GUI selection (explorer, graphic selections, ...). A selection listener is time-intensive because it is called every time a selection change occurs in the application. A mechanism has been implemented to allow optimization of selection listeners for Action. Instead of implementing SelectionListener and adding the implementee as listener on the FocusManager, actions should implement SelectionActionListener directly. This provides a way to refine the need for the action by specifying an update mode on selection change. Possible modes are UPDATE_SELECTION_ONLINE and UPDATE_SELECTION_OFFLINE (default mode) and can be set by using setUpdateSelectionMode. If the mode is set to UPDATE_SELECTION_ONLINE, the update method is called to allow a perfect synchronization between the action and the selection change events (this mode is equivalent to registering a SelectionListener on the FocusManager). If the mode is UPDATE_SELECTION_OFFLINE, the update method is only called before displaying a popup menu or a menu. The rule is to use the online mode only if the action is added to a toolbar or if an accelerator key stroke has been specified.

Actions don't need to manage the selection update mode if they implement the interface 'SelectionActionListener' instead of 'SelectionListener'. When adding an action to a JackToolbar, the container manages this mode to ensure that these actions are updated. It is also automatically managed when an accelerator is set on the action using the setAccelerator() method.

Changing the accelerator to a non null value changes the mode to UPDATE_SELECTION_ONLINE and prevents the mode to be changed manually. This is done to ensure that the action will be ready if the accelerator is activated. Remember that OFFLINE actions are updated when the menu is selected and that menu selection events are not triggered when an accelerator is used. Since these events are triggered for mnemonics, this rule does not apply for mnemonics.

9.5 Modifying Action Properties

It is strongly recommended to avoid updating action properties during the actionPerformed() execution. This can be considered illogical to do so. If for example an action updates the name of a DbObject, it should not change its action's name to update itself (assuming the object name is displayed in the action's text). Normally this action should already have registered to the name meta field to be notified on changes (in case the name change is performed from another component). Thus, by modifying the DbObject's name in its actionPerformed() method, the action is assured to be notified of changes performed on the name by way of its meta field listener. This also ensures that if errors occur for any reason while setting the name on the DbObject (causing the name to not be changed or the change to be rolled back), the action will still be in a correct state. If the action updates its text and the change is not committed, the action's text may be invalid. This also applies to any action properties and should also be applied to any components of the application.

Guideline #16: Avoid updating action properties during the actionPerformed() execution.

9.6 Using Accelerators

Acceleration key strokes definition implies that the actionPerformed() can be invoked at any time. This also applies when actions are added to the toolbar. In both cases, the action must register on all events required to update its state. Most actions only have to listen for a selection change event; in that case, the action should implement SelectionActionListener instead of SelectionListener (see section above). If the setVisible(false) is used, ensure that setEnable(false) is also used in order to prevent the accelerator from triggerring the actionPerformed().

9.7 Specialized Actions

Some subclasses of AbstractApplicationAction provide additional behaviors and properties. The main subclasses are AbstractDomainAction, AbstractTriStateAction and AbstractTwoStateAction. These classes are also extended to provide more specialized features. In all cases, refer to the API documentation for more details.

9.7.1 AbstractDomainAction

This action provides an additional property VALUES for displaying a list of values. This class also supports a selection (SELECTED_VALUE) property. This action is supported by JackMenu, JackPopupMenu and JackToolbar. The current tool bar implementation displays a combo box for selecting a value. The menu implementation creates a sub menu with a group of radio menu items for each possible value. Any values can be provided for VALUES. By default, the value provided by toString() is displayed for an object. Color and Icon values are special cases. A small icon filled with the color is displayed if the value represents a color. If a value represents an icon, the latter one will be shown instead of the text value returned by toString(). Null values can be provided to add separators.

9.7.2 AbstractTriStateAction

Provides three states corresponding to selected (pressed), not selected and an additional state to represent that the values differ. (For example, the abstract button is pressed to show that a class is abstract and not pressed if the class is not abstract. If two classes are selected and only one of them is abstract, the third state is applied).

9.7.3 AbstractTwoStateAction

Provides two states corresponding to selected (pressed) and not selected.

9.8 *Tracking actions executions*

AbstractApplicationAction offers the possibility for any application's components to be notified when an action has been performed. Interested components need to register a jack.actions.ApplicationActionListener using the method addApplicationActionListener() on AbstractApplicationAction.

9.9 *Programmatic action execution*

To allow actions to be executed without user intervention, actions should override the protected method ' doActionPerformed()'. This allows the action to be executed programmatically.

9.10 *Class AbstractActionStore*

Actions are in most cases singleton (only one instance of the class exists in the application).   The actions' store (also a singleton) can return the action instance using the corresponding key. It is also possible to execute an action using the action's key by calling the method 'performAction(String)'. Actions defined in JACK should be instanciated in AbstractActionStore. The specific SMS package defines its own ActionStore subclass (SMSActionStore). This specific class is used to instanciate and store specific SMS actions. For each modeling module, an ActionFactory exists, and actions specific to these modules are instanciated in that factory.

9.11 *Actions Utilities (jack.actions.util)*

Since many projects can be opened in ModelSphere at the same time, and for each project a different Db instance is used to manage the project, actions are required to ensure that a transaction is opened for each selected object in order to access information in the model (refer to the Modeling Framework for more details on transactions). Objects from different projects can be selected at the same time (mostly by using selection in the Explorer). The utility class DbMultiTrans provides a method to handle transactions without the need to check for all active Db instances.   The begin transaction and commit transaction automatically check and ensure that a transaction is opened to access the provided objects.

Guideline #17: It is a good practice to use DbMultiTrans for managing transactions within Action's actionPerformed().

9.12 *How to create a new Action in ModelSphere*

- Create the class in the feature's corresponding actions package. The action should inherit AbstractApplicationAction or one of its subclasses depending on the desired behaviors.

- For an OFFLINE action, the class must implement SelectionActionListener. In that case, the method 'updateSelectionAction' is called before the menu becomes visible (either from the main menu bar or a contextual popup menu). Code in this method should check if the current focus of the application is valid for this action to be executed and the 'enable' and 'visible' must be updated accordingly. Note: This method should not initiate a transaction, it is already invoked within a read transaction on all the db instances implied in respect to the current focus.

- For an ONLINE action (added to a toolbar or using accelerator key strokes), identify the events that could affect the action's states and implements the corresponding listeners. If the action's state only depends on selections, implements 'SelectionActionListener' in the same way as for OFFLINE actions; though in this case, the method 'updateSelectionAction' will be called for each selection change instead of when the popup menu is activated.

- Implement actionPerformed(). Note: When this method is called, it is correct to assume that the selection is valid because the 'updateSelectionAction' is guaranteed to be invoked when the selection changes. If a selection contains unsupported objects for the action and if the action changes its state to disabled during 'updateSelectionAction', the method actionPerformed() will never be invoked.

- If the action is located in JACK, add a key in jack.actions.AbstractActionStore to identify the action. The action's singleton instance should also be instanciated in this class. The same steps also apply to SMS actions. In this case, use the class sms.actions.SMSActionStore instead. For modules, add the key in the class '<modulePrefix> + ActionConstant' contained in the actions package for the module and instanciate the action in the class '<modulePrefix> + ActionFactory' contained in the same package. Centralized actions identification and instanciation ensure that all actions are singleton (1 instance) and limit the amount of action's updates, thus reducing the memory required and increasing the global performance. This also facilitates exchanges between actions when an action needs to update or invoke another action (for rare cases only; in most cases, actions should rely on event mechanisms to ensure that they have full knowledge of how they and their dependencies are updated). In some cases, like when an action extends AbstractDomainAction or if the action is used for debug purposes, you are not required to follow the rules defined in this step. Note: If an action applies to more than one module, use the SMS actions' package.

- If the action needs to be added to popup menus, add the action's key in the String array contained in sms.popup.ApplicationPopupMenu. This string (keys) array is used to determine the order in which actions need to be added to popup menus. This ensures uniformity for grouping and ordering of menu elements. Note that this is only used for popup (contextual) menus.

- Add the singleton instance to desired menu, toolbars and popup menus. To add the action to popup menu, use the action's key. To add the action to menus and toolbars, use the action instance. Actions' instances can be obtained from AbstractActionStore. The action store instance can be obtained either by SMSActionStore (singleton) or from the application's context (jack.srtool.ApplicationContext). For popup, each module has a class named xxPopupMenuPool containing the String array (actions' keys) for each specific object types. The mapping between classes and arrays are specified in modules' classes.

Inheritance is taken into consideration when popup menus are created. The key array representing the actions applying to a specific type is merged with all its superclass' key array. It is not needed to specify an action key for a type B if that key has already been specified for its superclass A. When an object is instance of any interface or classe mapped to a keys array, it inherits the keys from these arrays.

In these arrays order is not important. Action ordering is performed using the array defined in ApplicationPopupMenu. When adding a key to an array, always make sure to add it to the highest type in the inheritance hierarchy for which the action applies. For example, if an action applies to all tables (regardless of the DMBS specific classes), this action should be mapped with DbORAbsTable. The same rule applies for graphical objects. In addition, graphical objects also inherit the actions specified for their corresponding semantic objects.

10 The Wizard Dialog

A wizard is a dialog with several panels; the user can navigate among panels using the **Previous** or **Next** button. Swing does not provide standard libraries to construct wizard dialogs (but Swing will eventually support this). Because wizards are useful in many contexts, they are implemented by the JACK framework. The classes of the org.modelsphere.jack.gui.wizard package serve to construct wizards.
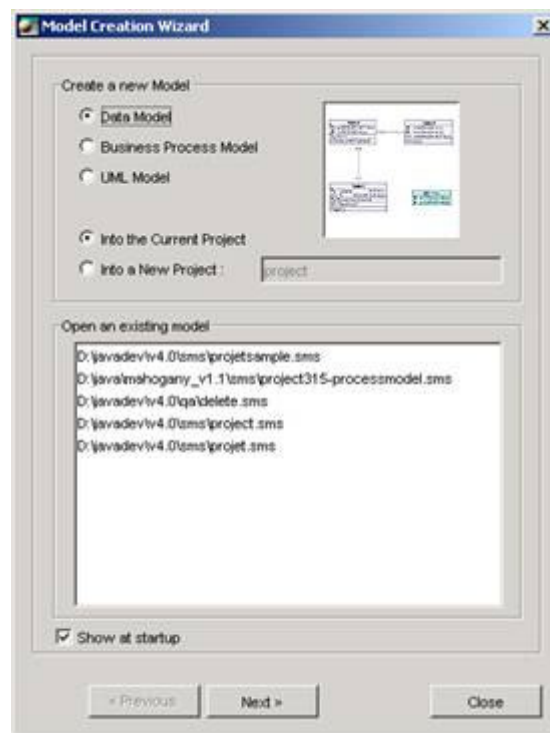

Figure 22: A Wizard Dialog

11 Using DbObject References in GUI Components

A GUI component cannot directly contain a DbObject, but rather a comparable element refering the DbObject instance (see the DefaultComparableElement class in the org.modelsphere.jack.util package).

For instance, the combo box model[11] must always contain an instance of DefaultComparableElement rather than the DbObject itself. The combo's renderer invokes the toString() method by default if the model is not an instance of Icon. If the combo's model was a DbObject, access to this DbObject would be done within a transaction.

The renderer is called by the repaint() method[12], and therefore must be very fast to execute. Involving a transaction when the repaint() method is called would dramatically slow down the performance of the application (of any GUI-based application).

The comparable element stores the DbObject for future reference and the string value (returned by the toString() method) for the display.

---

11 The term *model* here refers to the data of a GUI widget. Do not confuse with the general use of *model*, a container of semantic objects. See the glossary and the end of the document.

12 The Swing framework invokes the repaint() method to redraw a damaged area. Consult the Swing documentation for more details.

CHAPTER 9 - EXCEPTION HANDLING

Exception handling is a very important issue of every application. ModelSphere offers mechanisms to manage exceptions. These mechanisms can not guarantee that the application will be restored in a valid state in every situation. Developers are required to follow rules to ensure that the application's valid state is restored.

1 ExceptionHandler

This utility class is defined in the JACK package. It provides utility methods to handle exceptions. All the features should use the ExceptionHandler to process incorrect or unexpected behavior. The ExceptionHandler ensures the following:

- All opened transactions are rolled back.

- For some specific exceptions, a simple message is displayed to the user. (Error, Warning or Information messages). These messages are displayed if the Throwable receives an instance of MessageException. This mechanism can be used to abort the execution of a given task with a user feedback while ensuring the application returns to a normal state. Some special processings are also performed for IOException and DbException.

- In all other cases, a detailed error message is produced and shown to the user. The dialog clearly indicates that something abnormal occurred and the application's and virtual machine parameter details are collected and displayed along with the stack traces if available.

The following code shows how to use processUncatchedException().

```
try {
   ...
} catch (Throwable th) {
   ExceptionHandler.processUncatchedException(this, th);
} //end try
```

2 Programming Rules

Exceptions should normally be caught at the top most level. When a user clicks on a toolbar button, the try-catch block for unhandled exceptions should be put at the end of the actionPerformed() method. Resulting caught exceptions should be redirected to the ExceptionHandler. By doing this at the top most level, it ensures that no more operations will be performed after calling the ExceptionHandler. Not respecting this rule could result in many error windows being shown to the user instead of one.

The following code fragment shows how to manage exceptions correctly.

```
protected final void doActionPerformed() {
    ..
    try {
        performAction();
    } catch (Throwable th) {
        DefaultMainFrame frame = ApplicationContext.getDefaultMainFrame();
        ExceptionHandler.processUncatchedException(frame, th);
    } //end try
} //end doActionPerformed()

//perform the action
private void performAction() throws DbException {
    ..
}
```

If an error originates from the user, not the programmer, then the user has to be notified with a user-friendly error message, not with the crude ExceptionHandler interface.

```
protected final void doActionPerformed() {
    ..
    try {
        saveFile();
    } catch (FileNotFoundException ex) {
        String msg = MessageFormat.format(pattern, ..);
        JOptionPane.showMessageDialog(msg, ..);
    } catch (Throwable th) {
        DefaultMainFrame frame = ApplicationContext.getDefaultMainFrame();
        ExceptionHandler.processUncatchedException(frame, th);
    } finally {
        //dispose the resources
    } //end try
} //end doActionPerformed()
```

The `finally` clause is always executed, whatever xception is thrown. This is a good place to free resources, such as streams.

Guideline #18: When resources have to be freed, it is recommended to free them in a `finally` clause to ensure that they will actually be disposed.

The following code fragment is an example of bad code, because it propagates the exception beyond the scope of the action. If an exception occurs, we don't know where and how it will be handled by the caller method.

```
//BAD
protected final void doActionPerformed() throws DbException {
    //perform the action
    ..
} //end doActionPerformed()
```

The following code fragment shows also a bad way to handle exceptions. In this case, many error windows could be shown if several errors occur. Also, all kinds of exceptions could be caught, not just the DbException.

```
//BAD
protected final void doActionPerformed() {
    ..
    while (! done) {
        performAction();
        done = ..
    }
```

```
    } //end doActionPerformed()

    //perform the action
    private void performAction() {
        try {
            ..
        } catch (DbException ex) {
            ExceptionHandler.processUncatchedException(frame, ex);
        }
    } //end  performAction()
```

A called method (here the private performAction()) can handle exceptions (instead of propagating it to the caller method) at the condition that the exception does not break the normal execution of the action.

```
    //GOOD
    private void performAction() {
        try {
          //read a file line by line until we reach EndOfFile
        } catch (IOException ex) {
          //expected exception, end of file has been reached
        }
    } //end  performAction()
```

Of course, calling System.exit() must NEVER EVER appear in the code, otherwise the application will suddenly close, without allowing users to save their work.

```
    //CODE HORROR (VERY BAD)
    protected final void doActionPerformed() {
        try {
            ..
        } catch (DbException ex) {
            System.exit();
        }
    } //end  performAction()
```

CHAPTER 10 - INTERNATIONALIZATION

This section describes how internationalization is implemented in ModelSphere. It is important that readers distinguish between internationalization, localization and locale.

Internationalization

The operation of making a computer application flexible enough to run under any locale. In practice, this consists of following some programming guidelines, and to remove from the hard-coded character strings that are visible to the application's end-user. Internationalization is under the responsibility of the developers. The i18n acronym is often used for internationalization.

Localization

The operation of making a computer application able to run under a specific locale. Localization is under the responsibility of professional translators or technical writers.

Locale

Geographical, linguistic or cultural environment to which a computer application must be adapted.

1 Localization

ModelSphere is an internationalized application. All locale-dependent strings are removed from the Java sources and put in properties files instead. The current supported locales are English and French.

All strings visible to the users should be localized; this includes:

- GUI (Graphical User Interface) strings;

- Files generated as the result of a user request.

- Transaction names.

Example of localized resources:

```
String text = LocaleMgr.screen.getString("TextKey");
```

Resources are separated in four categories:

- Action

- Message

- Screen

- Misc

For each category, one properties file exists. This structure exists at many levels in the application.

2 Recommended Practices for Internationalization

2.1 Formatting Numbers and Dates

Number and dates must be formatted. Use NumberFormat to format numbers (this class takes the current locale in account and formats numbers accordingly). For instance decimal points will be replaced by the comma if the application runs in French. Do not use Float.toString() to format numbers.

Guideline #19: Use the NumberFormat and DateFormat in the java.util package to format locale-dependent strings.

2.2 Practices for Properties Files

All strings that can be shown to the user must be extracted from the code and localized in a properties file.

Properties file entries must conform to the following rules:

- The first 17 letters in the key can't be duplicated inside the same properties file.

- The key can't contain spaces, equal signs or special characters.

- Comments can't be added at the end of the string (they would be shown in the displayed text)

- Avoid building sentences using more than one entry. Translation in some languages may not work.

Guideline #20: Properties in the properties files should be ordered alphabetically.

Each line in the property file is composed of the key and the localized value. The key ends at the first '=' character (excluded). The localized text can contain the '=' character.

Keys are the same for all locales. Only values change.

2.3 Formatting Messages

If the value contains replacement codes (i.e. {0}, {1}, ...), it is intended to be processed by the MessageFormat class. Programmers and translators must be aware of the following limitations of the MessageFormat's format() method:

- Opening and closing braces must be preceded by a single quote to avoid confusion with replacement code: "The set '{0, 1, 2'} " will be rendered as "The set {0, 1, 2} " in the UI.

- Double quotes must be escaped: "The disk \"{0}\"." in the properties file will be rendered as "The disk "XX"." in the UI if {0} equals "XX".

- Single quotes must be repeated: "Don''t ask again" in the properties file will be rendered as "Don't ask again" in the UI.

Guideline #21: Programmers and translators must be aware of the limitations of the MessageFormat's format() method

3 Extended Features

The internationalization mechanism in ModelSphere relies on the features provided in Java, such as ResourceBundle. Some extensions have been added.

3.1 Mnemonics

Mnemonics are defined using a character contained in the text of menus, buttons or menu items. Since this text needs to be localized, a mnemonic is not guaranteed to work with other languages if the text doesn't contain the character. To avoid this problem, mnemonic support was added to localization in ModelSphere. Mnemonics can be specified in the properties files along the resource to which it applies. The mnemonic is separated from the text value by [>Mnc<].

The following shows an entry in a property file for the Open action. The displayed text for items derived from the action is 'Open' and the mnemonic 'O' (for GUI items where it applies).

```
openAction=Open[>Mnc<]O
```

Mnemonics can be accessed by using the method getMnemonic('key') instead of getString('key').

3.2 Accelerators

Accelerators use the same mechanism as mnemonics (see section above) except that it is specified using the tag '[>Acl<]

Accelerators can be accessed by using the method getAccelerator('key') instead of getString('key').

3.3 Others

[>Ima<] Image path, getImageIcon(); getImage()

[>Tip<] Tooltip, getToolTip()

[>Url<] Url, getUrl()

4 Resource Maintenance

This subsection is targeted to professional translators and technical writers.

4.1 Utility Scripts

Internationalization can be a tedious task, three scripts have been written in order to detect automatically translation errors or omissions.

str : detect hard-coded strings in the Java sources.

missing : compare two translations (properties files) to find if keys are missing.

res_en : extract translations in plain text for correction purposes.

These three UNIX scripts can be called by the UWIN application in Windows.

4.2 How to Use Scripts

The **str** script: go to the directory containing the source (by using the cd, change directory command), and then type 'str'. It scans all the Java sources of the current directory and sub-directories, recursively. Each time a string is detected, it is printed out, preceded by the name file and the number of the line containing the string. It is possible to redirect the output of this command in a file for future examination (type 'str > output.txt' in the shell).

Example:

```
sms/templates/GenericMapping.java:6093
     throw new RuleException("Repository function does not exist: " + repositoryString);

sms/templates/SQLForward.java:107
    if (varScope.isDefined("html")) {

sms/templates/SQLForward.java:108
     VariableDecl.VariableStructure varStruct = varScope.getVariable("html");

sms/templates/TemplateParametersInfo.java:56
    public String getDisplayName() {return "Generation Options"; }

sms/templates/TemplateParametersInfo.java:57
    public String getShortDescription() {return "Sets some generation options."; }

sms/templates/TemplateParametersInfo.java:62
    ParameterFileDescriptor() throws IntrospectionException {super("parameterFile",
TemplateParameters.class);}

sms/templates/TemplateParametersInfo.java:63
    public String getDisplayName() {return "Output folder"; }

sms/templates/TemplateParametersInfo.java:64
    public String getShortDescription() {return "It will generate comments on each
table."; }
```

Strings must contain at least one letter to be taken in account. Thus, strings like "0", "---" and "\" are ignored. Moreover, strings passed to the methods getString(), getUrl(), getImageIcon(), System.out.println(), etc. are ignored. Finally, any line with the comment //NOT LOCALIZABLE will be ignored. Consequently, if you want to exclude a hard-coded string in the code because you know that this string will never be displayed in the UI, you must add the **//NOT LOCALIZABLE** comment followed by a short explanation.

For instance:

```
public class Toto {
  private static final String GIF = "gif"; //NOT LOCALIZABLE, file extension
  private static final String HTML_BEGIN = "<html>; //NOT LOCALIZABLE, HTML tag
  ..
```

The **missing** script: go to a directory containing properties files (or a superdirectory) using the UNIX cd command, and type 'missing'. This will compare all the .properties files with their French counterparts, recursively (e.g. it will compare ScreenResource.properties with ScreenResource_fr.properties. Each time a key exists in the English properties and does not in the French ones, or vice-versa, or when a key is located at a different place in the file (keys should be kept alphabetically sorted), then the missing or moved key is printed out, preceded by the name of the file where the difference has been detected. It is possible to redirect the output of this command in a file for future examination (type 'missing > output.txt' in the shell).

An example of output:

```
diff english french
1855d1854
< src/org/modelsphere/sms/or/international/MiscResources.properties NoDocCommentsGenerated
1856a1856
> src/org/modelsphere/sms/or/international/MiscResources.properties NoDocCommentsGenerated
2069c2069
< src/org/modelsphere/sms/or/oracle/international/MiscResources.properties insteadof
---
> src/org/modelsphere/sms/or/oracle/international/MiscResources.properties instead of
```

How to read the output: the key `NoDocCommentsGenerated` has been deleted at the relative position 1855 but added at the relative position 1856 (it has been moved); the key `insteadof` in the English version has been erroneously renamed `instead of` in the French version. Relative positions are computed from a temporary file and are useless for further examination.

The **res_en** and **res_fr** scripts: go to a directory containing properties files (or a superdirectory) using the UNIX cd command, and type 'res_en' (or 'res_fr' to examine the French properties). This extracts all the values that follow the '=' sign in all the .properties files of the current directory and subdirectories, recursively. It is possible to redirect the output of this command in a file for future examination (type 'res_en > output.txt' in the shell).

An example of output:

```
Abbreviation Dictionary
Aborted
Aborting
Aborting...
Abstract
abstract class name
Abstract Data Type
Abstract Data Type Graphical object
Abstract Method
```

The text extracted here could be analyzed by a spell checking program in order to detect spelling mistakes.

➢ The source code of these three scripts is provided in the appendix C on page 143.

4.3 DbResources Properties Files

DbResources.properties in the SMS modules are generated by the genmeta plug-in. They cannot be modified manually, because all the changes will be lost the next time the genmeta will be performed. Refer to the meta management section of this document for more details on how to change these resources.

The file 'DbResources.properties' in the directory 'jack\baseDb\international' is the only DbResources.properties that should be updated manually.

CHAPTER 11 - PLUG-INS API

This section describes how to use the plug-in interface to add new functionalities to ModelSphere without modifying the core application. This section does not cover the internal code managing the plug-ins.

There are a number of ways to extend ModelSphere's functionality. The simplest way to achieve that is to use the Java plug-in interface. The Java interface lets you develop your own code and refer to the ModelSphere API in order to enhance the application model, behavior and user interface.

Typically, to build a plug-in, you write and compile Java code, and place the resulting class file on the plug-in search path of ModelSphere, under the \plugins directory. Since ModelSphere keeps a list of plug-ins in its records for fast loading, the modelsphere.plugins file has to be deleted from the application's root directory, to accomplish that your new plug-in searched and loaded in the application memory space the next time ModelSphere is launched. Alternatively, it is possible to manually load the plug-in from the Plug-ins Manager dialog, available from the Help menu of the application.

A typical challenge when developing a plug-in is debugging, which requires the knowledge of an advanced technique called remote debugging. The last section walks you through the process of using this technique to achieve successful plug-in development for the ModelSphere application.

For more details on plug-in classes, please refer to the ModelSphere API javadoc.

1   The Plugin and Plugin2 interfaces

1.1   The Plug-in Path

The plug-in path is the enumeration of directories that ModelSphere scans to find its plug-ins. It is specified by the *pluginpath* program argument. They are two ways to specify this argument. First, you may declare it in your IDE as a program argument, as illustrated below for the case of Eclipse.
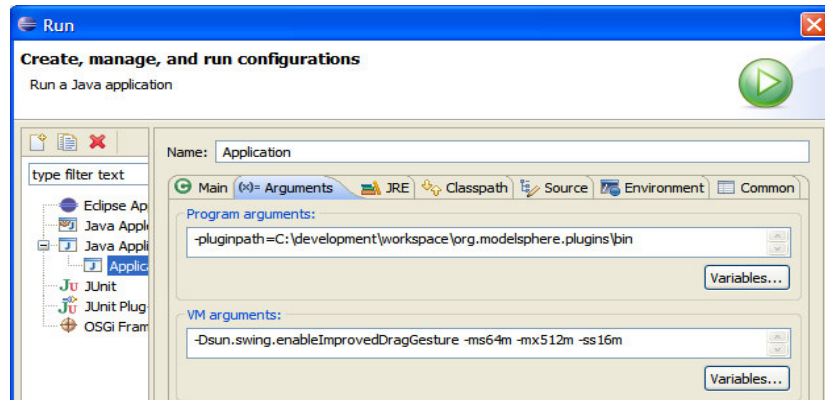

Figure 23: Run Configuration with the Eclipse IDE

The second way is to create a text file named modelsphere.args that contains the following text, and to place this file in ModelSphere's root directory.

```
-pluginpath=C:\development\workspace\org.modelsphere.plugins\bin
```

Table 17: Fragment of modelsphere.args

The plug-in path can contain several directories, separated by a semi-colon (;) character, in the case your plug-ins are placed in different directories. For instance you may have the ModelSphere standard plug-ins at one place, commercial plug-ins at a second place, and your own plug-ins at a third place.

The class PluginMgr of the package org.modelsphere.jack.plugins is responsible for loading the plug-ins. The loadPlugins() method scans all the .class and the .jar files found in the plug-in path, and tests if the current class inherits from org.modelsphere.jack.plugins.Plugin, in which case it loads this class.

The scan operation may be long to execute. Once it is terminated the PluginMgr class creates a modelsphere.plugins text file that contains all the detected plug-ins. The subsequent times ModelSphere launches, it reads this file instead of scanning again the plug-in path. This allows ModelSphere to start faster. If you change the contents of modelsphere.args, delete the modelsphere.plugins file; PluginMgr will perform a scan of the plug-in path and will create a new modelsphere.plugins file.

```
URL=file:C:/..;org.modelsphere...AnsiForwardToolkit;isRule=false;name=Ansi Forward
Engineering;loadAtStartup=true
URL=file:C:/..;org.modelsphere...ProcessTree;isRule=false;name=Tree Diagram;
loadAtStartup=true
```

Table 18: Fragment of modelsphere.plugins

1.2 The Plugin Inheritance Structure

This diagram above shows an overall view of the inheritance structure of the Plugin interface. Each plug-in ultimately inherits from Plugin. To ease the comprehension, JACK packages are shown in orange, SMS packages in green and the concrete plug-ins in blue.

The next sections describe the Plugin interface and its implementation classes. The Plugin interface in JACK is the most generic and is covered first. The more specific subclasses are covered in the later sections.
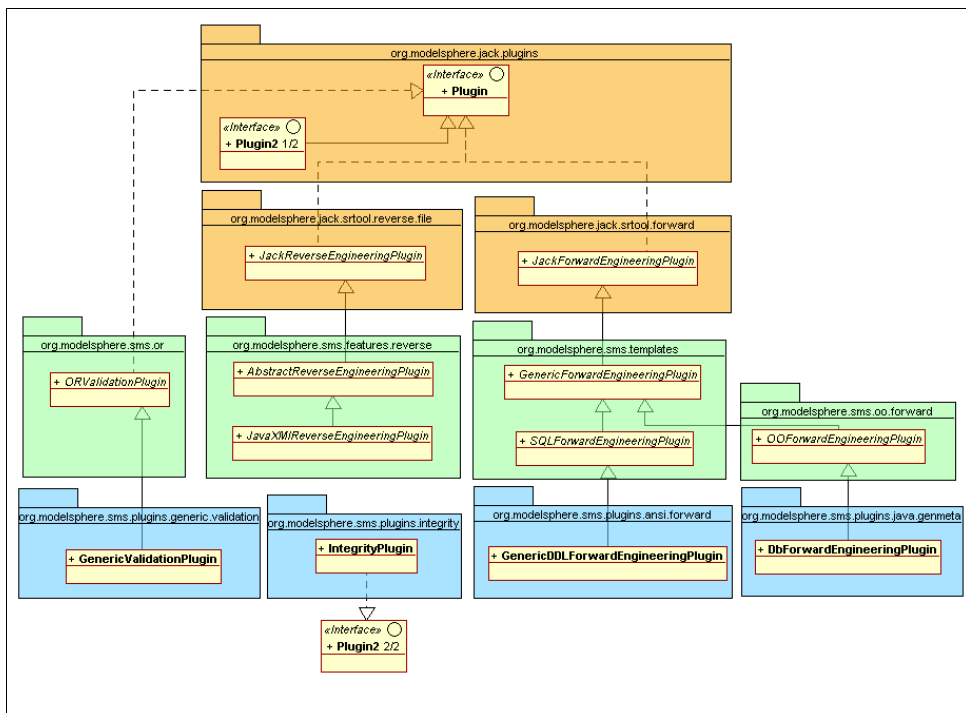
Figure 24: The Plugin Inheritance Structure

1.3 The Plugin Interface

All the plug-ins loaded by ModelSphere inherit from a common interface: Plugin (in the org.modelsphere.jack.plugins package). Plugin is used for reverse and forward engineering, and JDBC driver plug-ins. It is the common interface for adding functionalities to the application.

Each plug-in has a signature, represented by the org.modelsphere.jack.plugins.PluginSignature class. The plug-in signature contains the following properties.

Signature Property	Description
Name	The name of the plug-in.
Version	A string identifying the version of the plug-in.
Author	The author of the plug-in.
Date	The publishing date of this version of the plug-in.
Build-Required (integer)	The build ID of ModelSphere required to instantiate the plug-in. If this build ID is higher than the current build ID of ModelSphere, the plug-in is not loaded.
Hide (Boolean)	If true, this plug-in is not loaded. This allows ModelSphere to start faster.

Table 19: Plug-in Signature Properties

The name, version, author and date properties appear in the application by selecting the **Help→Plug-ins..** menu item.
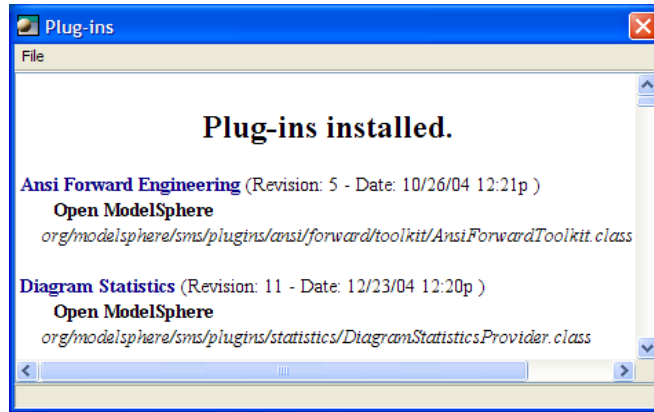
Figure 25: The Plug-ins Window

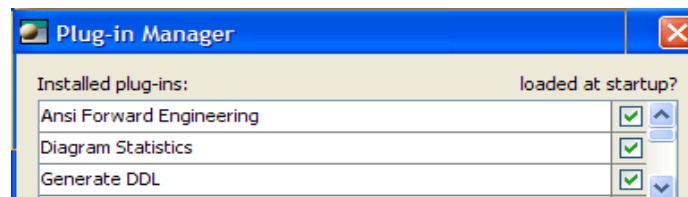The Hide property is set when the user selects **Help→Plug-in Manager..** menu item.

Figure 26: The Plug-in Manager Window

1.4 The Plugin2 Interface

The following diagram focus on the JACK part of the whole diagram. In accordance to the UML standard, interfaces are identified by the «interface» stereotype, abstract classes are in italics, and realization inheritances are depicted with a dash line.
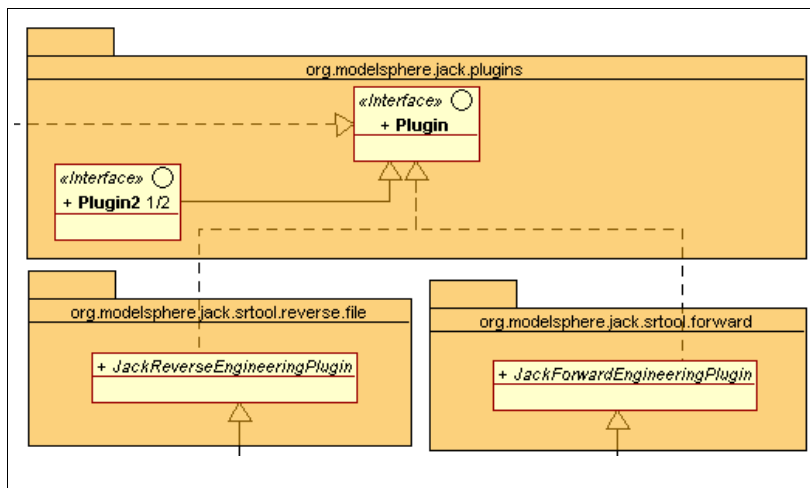

Figure 27: The Plugin interface and its Direct Subclasses

The Plugin2 interface in the org.modelsphere.jack.plugins package extends the Plugin interface and declares two additional methods: getOptionGroup(), that adds an option panel associated with this plug-in in the Option dialog, and getPluginAction(), that adds an action to use in the tool bar or in the menu bar. The PublicAction is a subclass of AbstractApplicationAction that can be placed in menus and toolbars. OptionGroup defines options (user's preferences) in the Tool->Options.. menu.

JACK also contains two other subclasses of Plugin:

- `JackForwardEngineeringPlugin`: This class provides common services for all forward engineering plug-ins.

- `JackReverseEngineeringPlugin`: This class provides common services for all file-based reverse engineering plug-ins. The canReverse(File f) method returns true if a given plug-in is able to perform reverse engineering on the file.

The Plugin2 has been added after Plugin, and it is more complete. If you want to create a new plug-in that requires defining actions or options, implement the Plugin2 interface.

The JackForwardEngineeringPlugin class defines a method named isSupportedClass(Class cl). When the user right-clicks an element of the model to invoke the pop-up menu, a call to isSupportedClass() is performed, with the type of model element passed as a parameter. If this method returns true, then the Generate menu item adds a sub item for this instance of plug-in. If the user selects this item, then the execute() method of JackForwardEngineeringPlugin is invoked.

2 Specialized Plug-ins

2.1 Forward Engineering Plug-ins

Specialized plug-ins, such as the GenericDDLForwardEngineeringPlugin in the org.modelsphere. sms.plugins.ansi.forward package, do not inherit directly from Plugin, but from SQLForward EngineeringPlugin defined in the org.modelsphere.sms.templates. The SQLForwardEngineering Plugin provides functionalities common to any forward engineering plug-in.
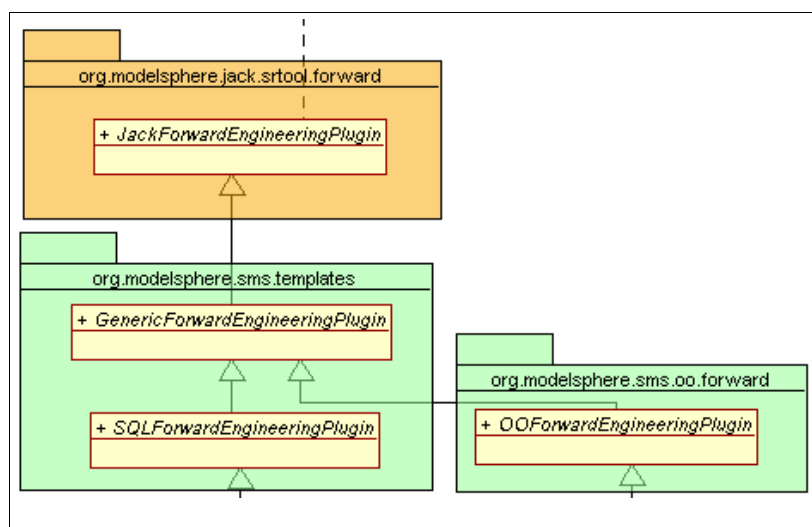

Figure 28: The Forward Engineering Plug-ins

- `GenericForwardEngineeringPlugin`: This class provides common services for code generation plug-ins.

- `SQLForwardEngineeringPlugin`: This class provides common services for SQL generation plug-ins.

- `OOForwardEngineeringPlugin`: This class provides common services for OO code generation plug-ins.

If you want to implement your own forward engineering plug-ins, you are invited to examine the contents of these classes listed above.

2.2 *Reverse Engineering Plug-ins*

Although no public domain plug-ins are currently available to perform reverse engineering of Java classes or XMI files, ModelSphere already defines extension points where it is possible to plug these kinds of plug-ins.
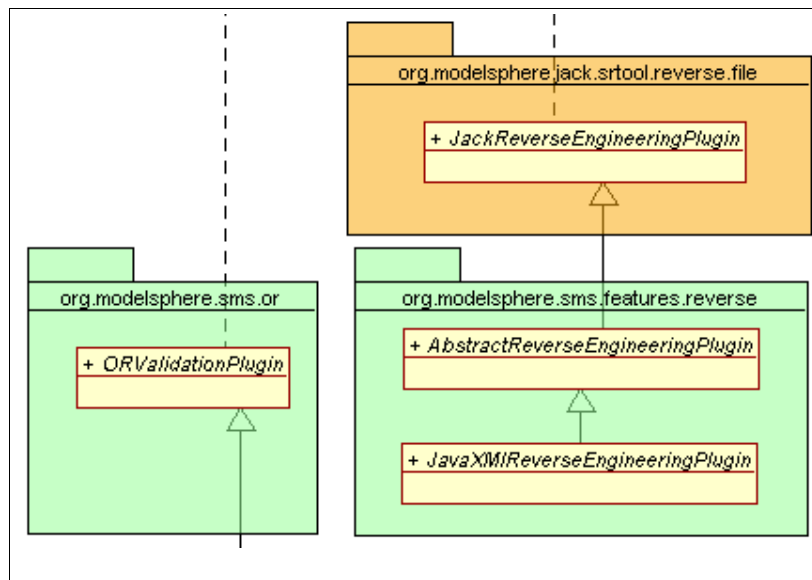
Figure 29: The Reverse Engineering Plug-ins

- `AbstractReverseEngineeringPlugin`: The common abstract class for reverse engineering plug-ins.

- `JavaXMIReverseEngineeringPlugin`: The common abstract class for Java and XMI reverse engineering plug-ins. All the plug-ins inheriting from this file can be invoked from the **Tools➜Java➜Java / XMI Reverse Engineering..**. If no plug-in that inherits from this class has been loaded, then the menu is not visible.

- `ORValidation`: Common services for plug-ins that perform validation of relational rules and clean-up of models. Implements directly the Plugin interface.

The MainFrameMenu class in the org.modelsphere.sms package creates the menu items in the application. It adds a menu item Java under Tools, and listens for a selection event on this menu.

The ReverseAction in the org.modelsphere.sms.oo.actions package returns the list of loaded packages that inherit from JavaXMIReverseEngineeringPlugin. If at least one of these plug-ins is found when the user clicks the **Tools→Java** menu item for the first time, a menu item named **Java / XMI Reverse Engineering...** is added under the **Java** item.

2.3 DBMS Reverse Engineering

The JDBC Reverse Engineering plug-in allows to reverse engineer all databases that provide a JDBC connection (most do). The JdbcReverseToolkitPlugin class is located in the org.modelsphere.sms.plugins. jdbc.bridge package. It inherits from ReverseToolkitPlugin, which in turn inherits from Plugin. Similarly to the SQLForwardEngineeringPlugin class, ReverseToolkitPlugin provides functionalities common to all reverse engineering plug-ins.

2.4 Validation and Integrity Plug-ins

The generic validation plug-in inherits from the ORValidationPlugin class and the IntegrityPlugin implements directly the Plugin2 interface.
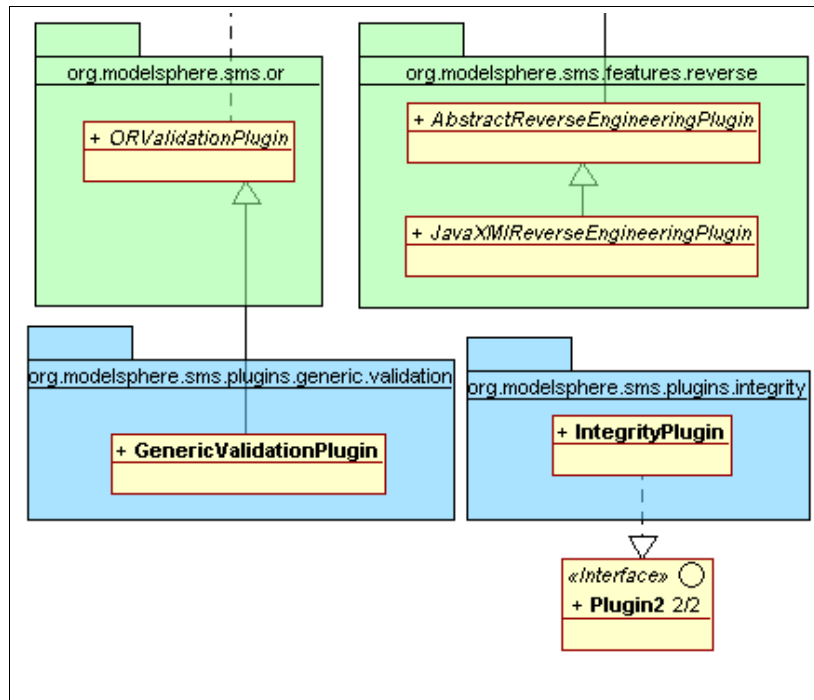

Figure 30: The Validation and Integrity Plug-ins

- GenericValidationPlugin: Validates common rules for relational models.

- IntegrityPlugin: Checks the relational integrity and clean-up models. Because this class inherits from Plugin2 (instead of Plugin), it defines its own Swing action (PluginAction) and knows how to plug itself in the application.

The Integrity plug-in places an action in the Tools menu. Examine this class if you want to create your own plug-in and want to know where place the Swing action in the UI to allow the user to invoke your plug-in.

2.5 *SQL and Java Code Generation Plug-ins*

ModelSphere also comes with public-domain plug-in to perform SQL and Java code generation. It is important to notice that the DbForwardEngineeringPlugin only serves to generate the model classes of ModelSphere, it is not a general-purpose code generator.
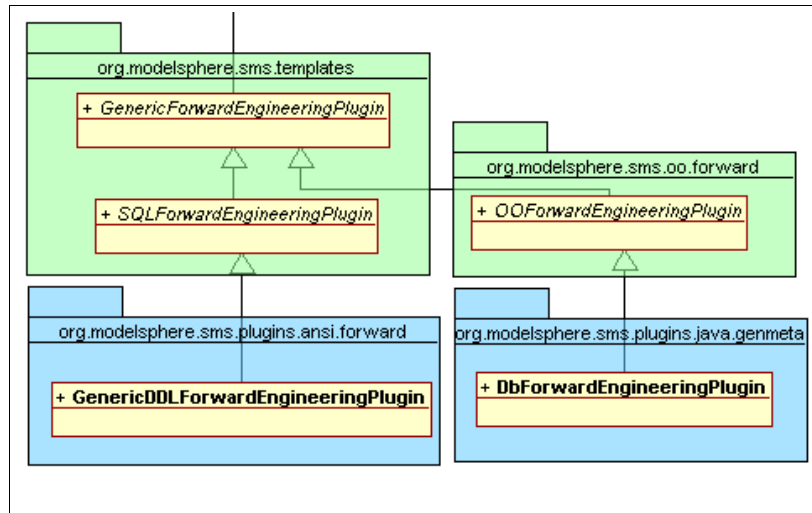

Figure 31: The SQL and Java Code Generation Plug-ins

- `GenericDDLForwardEngineeringPlugin`: Generates ANSI (SQL-92) data definition language.
- `DbForwardEngineeringPlugin`: Generates the model classes from the meta-model.

These classes can serve as examples if you want to create your own forward engineering plug-ins.

3 Debugging your plug-in

The ability to debug your plug-in is provided by the Java technology and the development environment that supports it. To debug a plug-in developed for ModelSphere, you should use a technique called *remote debugging*. Remote debugging is useful for debugging server side applications, or applications that run in a different process. The debugger client uses a special API and "connects" to the server application (here the JVM running in another process that hosts the ModelSphere application) and performs similar operations as a regular debugging session, while letting you step through the execution from the client computer.

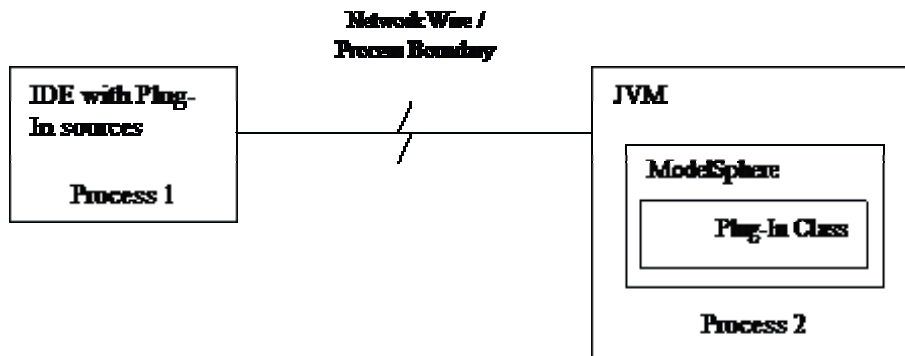The following schema outlines the principle and process boundaries:

Figure 32: Plug-in Remote Debugging

What is particularly useful with this technique is that you can step through the code that is executed remotely (or locally in another process). Moreover, the server process needs only to know debugging information about the classes you want to debug even though the application itself is made of release binaries.

Debugging your plug-in requires:

- That you have access to an installed version of the product.

- A development environment that supports Java Remote Debugging, such as the Eclipse Platform version 3.1.0 or higher.

- Plug-in source code and one or more class files.

3.1 *The debugging process*

This section walks you through the process of successfully establishing a remote debugging session to debug a plug-in running in the process space of the ModelSphere application.

3.1.1 **Prepare the remote debugging session**

In the IDE of your choice, create a new debug configuration for a Remote Java Application for your Java Project containing your plug-in. In Eclipse, access this menu in **Run→Debug…** after you have successfully compiled your plug-in. Verify the Connection Properties, and make sure your port is set to 4505 (or another valid free port). This port is chosen here, since in the next step, we instruct the server to use this same one. Please also make sure that no firewall blocks this port on the host computer.

Launch the server process

To establish the session, launch ModelSphere from the command line in a console window or batch file, instructing the JVM to start in debug mode. A sample command line is provided below. If your command line is incorrect, the only indication will be that the debugging session will not start when you attach the debugger in the next step. Refer to the Java 2 Standard Edition (J2SE) documentation for parameter signification if you require customization.

Sample Command Line:

```
C:\Program Files\Open_ModelSphere_3_0>"C:\Program Files\Java\jdk1.5.0_05\jre/bin/java.exe"
-Xdebug -Xnoagent -Djava.compiler=none
-Xrunjdwp:transport=dt_socket,server=y,suspend=y,address=4505 -ms64m -mx512m -ss16m -
classpath
".\modelsphere.jar;.\resources.zip;.\targets;.\lib\mailapi.jar;.\lib\mail.jar;.\lib\regexp
.jar;.\lib\activation.jar;.\lib\xerces.jar;.\lib\xalan.jar;.\lib\imap.jar;.\lib\parser.jar
;.\lib\smtp.jar;.\lib\jpython11.jar;.\lib\jaxp.jar;.\lib\pop3.jar;.\lib\JdDb.jar;.\lib\jdb
c901-thin-classes12.zip" org.modelsphere.sms.Application

Listening for transport dt_socket at address: 4505
```

Attach the debugger

In this step, you need to attach the debugger (typically Eclipse) to the server process (the JVM hosting ModelSphere). To do this, simply connect to the corresponding port on the target machine (or localhost if the process is local). The connection is established by launching the debug session.

The execution flow will be returned to the server process once the client debugger has connected successfully. You can then invoke your plug-in from ModelSphere, at which point the execution flow will be passed back to the debugger at the location of your first breakpoint.

You may then begin your debugging activities and use an iterative approach to patch your code by replacing the plug-in class files every time your source is modified and re-launch the remote debugging session as needed.

CHAPTER 12 - OTHER FEATURES

1 DBMS Connection

ModelSphere users can establish a connection to a database by using a JDBC connection. Users invoke **Tools→Database→Connect..** to create a connection configuration. A connection configuration defines the URL, the user's name and the password ModelSphere needs to establish a connection to an external database.

Once connected, users can open a SQL Shell by activating **Tools→SQL Shell**. They can also perform reverse engineering on JDBC-compliant databases.

These classes are involved in the DBMS connection. Refer to them for further implementation details.

Package (all in org.modelsphere)	Class	Description
jack.srtool.actions	ConnectAction	Action associated to the **Tools→ Database→Connect..** command.
jack.srtool.reverse.jdbc	ActiveConnectionManager	Open the Connect dialog and let the user define a connection configuration.
jack.srtool.actions	SQLShellAction	Action associated to the **Tools→SQL Shell** command.
jack.srtool.reverse.jdbc	SQLShell	Open the SQL Shell dialog and execute the SQL commands entered by the user.

Table 20: Classes Involved in the DBMS Connection

2 Debugging Utilities

ModelSphere runs either in a debug or a release mode. The class Debug in the package org.modelsphere.jack.debug package defines a Boolean variable named DEBUG. The application runs in debug mode when the DEBUG variable is true, and in release model when the variable is false.

Developers who want to make a release version have to set the DEBUG variable to false, otherwise ModelSphere end users will get error messages targeted to developers. The Ant build script that is used to make a distribution version of ModelSphere automatically set the DEBUG variable to false.

➢ Consult the section 6 of the next chapter (on page 96) for more details on the build script and the build process.

A useful method defined in the Debug class is the assert2() method. Assertions are conditions that must be true at the execution. For instance, at the beginning of a method, we can assert that a given parameter may not be null. Assertions at the beginning of a method are called *pre-conditions*, and those at the end of a method are called *post-conditions*.

```
public void method(Object o) {
   assert(o !=null);
}
```

Assertions help to avoid that current development does not degrade the quality of the methods already written. On the other hand, it slows down the execution of the application. Because assertions are only verified when ModelSphere runs at the debug mode, there is no cost to use them for the end user's point of view.

Guideline #22: It is recommended to add assertions within methods to maintain the quality of the code.

Another useful facility provided by the debug class is the trace() method. It calls a System.out.println() statement, but only when ModelSphere runs in the debug mode. It is preferable to use it instead of the Java standard println() method, because just by setting DEGUG = false has the effect of making the trace() statement silent.

Guideline #23: It is recommended to call the trace() method instead of directly invoking the System.out.println() statement.

3 User's Preferences

User's preferences are very similar to application options, but they are generally not modified in the user interface, as opposed to application options that are modified by choosing **Tools→Options..** User's preferences are automatically saved when the application exits, and restored at the next session without the end user's intervention. They are persisted without the user's knowing.

The dimension of the application screen, the height and width of the explorer and the design panel are examples of preferences. If the user decides that the explorer will take 25% of the screen, this ratio is saved as a preference. When ModelSphere starts up, it reads the user's preferences and automatically allocates 25% of the screen for the explorer.

Another example is the **Show at startup** check box in the bottom of the Licenses dialog, as shown below.
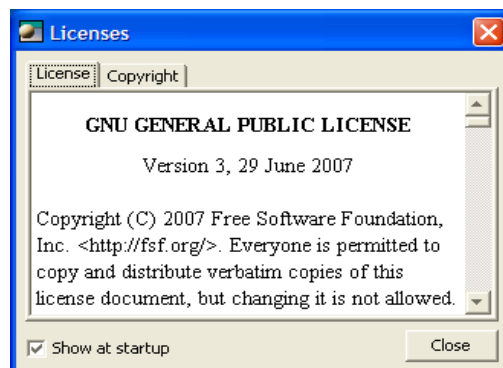

Figure 33: The Show-at-startup Preference

A preference belongs to a class. The following code shows how to add a ShowLicenseAtStartup preference in the class ShowLicenseAction. The key must be unique within the class, but two preferences in different classes may have the same name. Preferences must have a default value.

```
public final class ShowLicenseAction extends {

  public static final String SHOW_LICENSE_AT_STARTUP_KEY =
    "ShowLicenseAtStartup";  // NOT LOCALIZABLE, property key
  public static final Boolean SHOW_LICENSE_AT_STARTUP_KEY_DEFAULT = Boolean.TRUE;
  private static boolean g_showLicenseAtStartup = true;
```

The following code fragment shows how to get the preference. PropertiesManager handles several properties set, including the preferences set. Properties set needs the class where the preference has been defined (here ShowLicenseAction), its key and a default value for the case the preference has not been stored yet.

```
//get ShowLicenceAtStartup preference
public boolean getShowLicenceAtStartupPreference() {
  PropertiesSet preferences = PropertiesManager.getPreferencePropertiesSet();
  boolean showLicenceAtStartup = (preferences == null) ?
    true :
    preferences.getPropertyBoolean(
      ShowLicenseAction.class,
      SHOW_LICENSE_AT_STARTUP_KEY,
      SHOW_LICENSE_AT_STARTUP_KEY_DEFAULT);
  return showLicenceAtStartup;
} //end getShowLicenceAtStartupPreference()
```

The following code fragment shows how to set the preference. It is worth to mention that the saving/retrieving mechanism is completely left to the PropertiesSet and PropertiesManager classes.

```
public void setShowLicenseAtStartupPreference(boolean value) {
  PropertiesSet preferences = PropertiesManager.getPreferencePropertiesSet();
  if (preferences != null) {
    preferences.setProperty(ShowLicenseAction.class, SHOW_LICENSE_AT_STARTUP_KEY, value);
  } //end if
} //end setShowLicenseAtStartupPreference()
```

Refer to the PropertiesSet and PropertiesManager classes in the org.modelsphere.jack.preference package for further implementation details.

CHAPTER 13 - PACKAGING MODELSPHERE

The previous chapters describe how to develop ModelSphere within a development environment. Of course, end users do not want to execute ModelSphere from Eclipse or another development environment. Making a version of ModelSphere for distribution, that runs outside a development environment, is called packaging the application.

This section describes how to package ModelSphere from its source to make a distribution version.

1 Update the product, build and meta versions

This sections explains the differences between the product versions (2.4, 2.5, 3.0..), the build versions (300, 301, 302), and the meta versions.

Product version, build version and meta versions all have different goals. A product version is marketing-oriented. A product version only changes when ModelSphere publishers decide to release a new version of ModelSphere. The product version has no impact on the product execution. It is only a string defined in a property file. It can be changed in the misc resource file in the sms.international package.

The build version is used internally and is development-oriented. The build ID is an integer value hard coded in the main class Application. This build ID must be incremented each time a new compilation is delivered to the quality assurance (QA) team. The build ID appears in the splash screen. This is the only way for the QA to distinguish two compilations.

The meta version identifies a version of the meta-model. This ID is specified in the class SMSVersionConverter of the sms package. This number must be incremented only if a change to the meta requires data conversions.

Product Versions (Marketing)	Build versions (Development)	Meta versions (Development)	Release Dates
ModelSphere 1.0	Internal: 100-129; Release candidates: 150-155	1 (build > 100)	February 2002
ModelSphere 2.0	200-223	2 (build = 215) 3 (build > 215)	July 2002
ModelSphere 2.1	300-399	4 (build >= 300)	September 2003
ModelSphere 2.2	400-499	5 (build >= 400)	November 2004
ModelSphere 2.3	500-599	6 (build >= 500)	September 2005
ModelSphere 2.4	600-699	7 (build >= 600)	March 2006
ModelSphere 2.5	700-799	8 (build >= 700)	September 2006
ModelSphere 3.0	Internal: 800-899 Release candidates: 900-999	9 (build >= 800) 10 (build >= 900)	December 2007

Table 21: Product, Build and Meta Versions

The BUILD_ID constant in Application.java identifies the build version. The mapping of these three IDs is kept in the class SMSVersionConverter. It is important to keep this mapping up to date. Note that the meta version must not be incremented more than once for a 'marketing release'. Any changes in the meta during development phase must be managed internally using convertForDebugUpdate() method in MainFrame. Also note that both build ID and the meta version are saved at the beginning of .sms files.

This limitation is due to the fact that some modifications may be rolled back or changed many times and we do not want to pollute the meta and the version converter. Some of those changes would require to keep, for example, a metafield added for an alpha version in the final release even if it was a mistake to add it, and should be removed. The limitation also restricts repository migrations for future enterprise releases since some OODBMSs don't offer good support for meta migration. The repository migration depends on the choice for the OODBMS, but the limitation of meta versions is a safer choice.

Some modifications to the meta don't require to change the meta version. The rules are the same as the rule and limitations for Java serialization. Removing a metafield or adding a new metafield don't require an update.

2 Manage the sources with a version controller software

Even if ModelSphere is an open-source project, versions must be managed with the same rigor as a commercial tool. It up to the publishing group to decide which version controller software to use and how to use it. ModelSphere was initially managed using Microsoft's SourceSafe version control software, but developers may use CVS or Subversion as alternatives.

At determined times during the development, a version must be labeled using the version controller. This version (identified by a build version) is not published and is targeted to the QA team. When a version is judged stable, the development is frozen, and a new version is published.

It is beyond the scope of this document to explain how all the version controller tools on the market work. The following section describes the main tasks for developers who used SourceSafe to manage ModelSphere's code.

2.1 Managing the sources with MicroSoft SourceSafe

Updating the project

- Perform a "Show Difference" from the project's root (slower, but safer)
- Perform a "Get Latest.." to get the latest changes.

Making Corrections

- Perform a "Show History" on the last label to see all the files that have been modified
- Put a temporary label on the root to mark the beginning of the corrections. Labels are not mandatory, but they are useful to manage multiple corrections.
- Check out files, make corrections in the IDE, and check in modified files.
- Perform a "Show History" on the temporary label to see modifications made by other developers during this period of correction.

- Finally, put a permanent label.

Consult the SourceSafe documentation for more details on the operations described above.

3 Generate the metamodel and the parser

The genmeta plug-in reads a genmeta.sms file and creates the DbX.java files. The JavaCC application reads a grammar file (.jj file) and generates source code in the org.modelsphere.jack.templates.parsing package. The genmeta and the parser generator are covered in the How To chapter.

Before deploying a version, the publishing team must make sure that the latest version of the metamodel fits with the latest version of the model classes, and that the latest version of the grammar fits with the latest version of the generated parsers.

4 Internationalization

➢ Internationalization is covered in details in the chapter 10 on page 74.

Before deploying a version, the publishing team must make sure that no hard-coded strings are left in the source code. They can use the scripts described in chapter 10 to find hard-coded strings, missing keys, and so on.

5 Generate the User's Guide

The User's Guide is an HTML document that can be consulted from a Web browser, or directly within the ModelSphere application by clicking **Help→User Guide…** The user's guide is organized in an hierarchical tree structure, and the user may expand or collapse tree nodes and select a particular page.
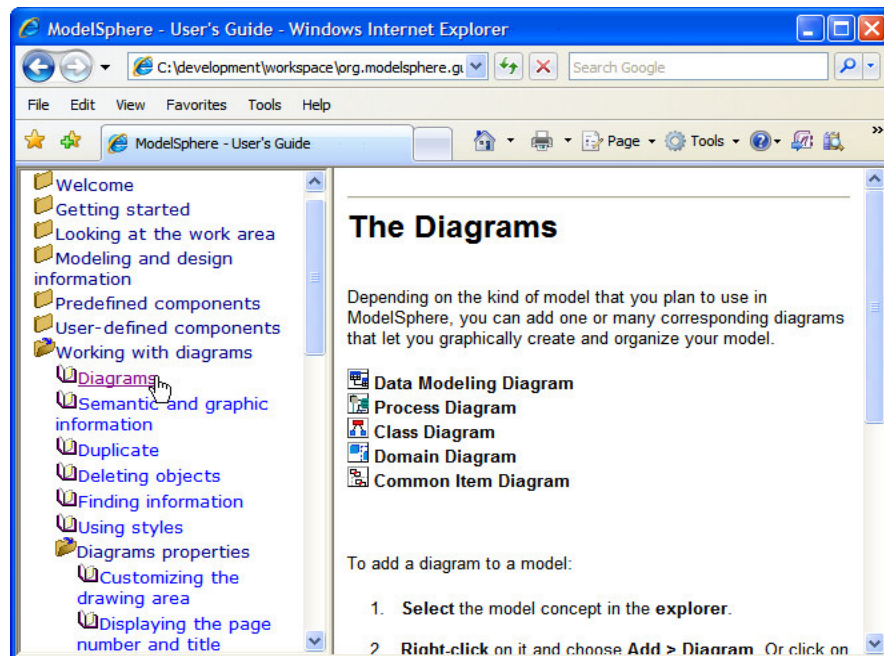
5.1 Structure of the Help Files

The table below shows the structure of the help files, in the org.modelsphere.guide project.
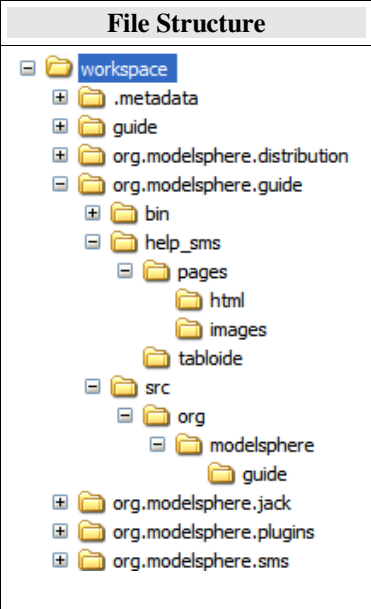
File Structure	Folder	Files

☐ 📁 workspace
 ⊞ 📁 .metadata
 ⊞ 📁 guide
 ⊞ 📁 org.modelsphere.distribution
 ☐ 📁 org.modelsphere.guide
    ⊞ 📁 bin
    ☐ 📁 help_sms
       ☐ 📁 pages
          📁 html
          📁 images
       📁 tabloide
    ☐ 📁 src
       ☐ 📁 org
          ☐ 📁 modelsphere
             📁 guide
 ⊞ 📁 org.modelsphere.jack
 ⊞ 📁 org.modelsphere.plugins
 ⊞ 📁 org.modelsphere.sms
	help_sms	User Guide.html
		Guide utilisateur.html
		toc.xml

	help_sms/pages	tree_en.xml tree_fr.xml xmlTree.js closed.gif, doc.gif, open.gif
	help_sms/pages/html	en_0010_intro.html en_0020_whatis.html ..
	help_sms/pages/images	abstract.gif ADD_ASSOCIATION_en.jpg ..
	src/org/modelsphere/guide	BuildHelpMain.java IndentWriter.java TableOfContents.java

Table 22: Structure of the Help Files

User Guide.html is the main page of the HTML guide. Open this file with a web browser to display the documentation, as illustrated at the figure 34: The User's Guide. This page splits the screen in two frames, the explorer frame takes 25% of the total screen and shows the tree structure, and the content frame takes the remaining 75% at the right of the explorer. The tree structure is defined in the tree_en.xml file. The content frame refreshes when the user clicks a tree node.

Guide utilisateur.html is the French counterpart of the User Guide.html. It reads the tree_fr.xml file to construct the structure of its explorer frame.

toc.xml defines the structure of the help files (both English and French). Building the User's Guide consists of reading this file and generating the tree_en.xml and the tree_fr.xml files.

tree_en.xml this file is generated from the toc.xml and defines the structure of the English version of the user's guide. **tree_fr.xml** is its French counterpart.

xmlTree.js is a Java script that allows the user to expand and to collapse tree nodes.

closed.gif (📁), **doc.gif** (📖) and **open.gif** (📂) are the images used in the explorer to show expanded and collapsed nodes.

en_0010_intro.html is the English version of the introduction, and may contain images located in the **images** folder.

The **BuildHelpMain.java**, **IndentWriter.java** and **TableOfContents.java** are the source code of the BuildHelp tool that generates the tree_en.xml and tree_fr.xml file from the toc.xml file.

5.2 *The XML Files that Define the Structure*

The toc.xml (source)	The tree_en.xml (generated)

`<help>`
 `<category en="Welcome" fr="Bienvenue"`
  `<book en = "Introduction" fr = ..`
   `file="0010_intro.html">`
   `...`
	`<tree>`
	 `<branch id="Welcome">`
	  `<branchText>Welcome</branchText>`
	  `<leaf>`
	   `<leafText>Welcome</leafText>`
	   `<link>html/en_0010_intro.html</link>`
	  `</leaf>`

Table 23: The contents of the XML Files

The English and French versions (and eventually other languages) of the User's Guide must follow the same structure. Instead of duplicating the structure for the two languages, the structure is defined in a single file, the toc.xml file. In this file, books are grouped into categories, and categories may be nested. This file avoids redundancy (there is only one occurrence of the word Welcome), and consequently facilitates its maintenance. Moreover, this file forces the English and the French versions of the user's guide to follow the same structure.

Unfortunately, the xmlTree.js script cannot read the toc.xml file, because it expects special tags such as <branch>, <branchText>, and <leaf>. It's why a small tool, the BuildHelp, is required to process the toc.xml file and generates the tree_en.xml and tree_fr.xml, that are then used by the Java script.

5.3 *Building the Help Files*

Within your IDE, invoke the main() method of the BuildHelpMain class. This reconstructs the tree_en.xml and the tree_fr.xml files from the toc.xml.

6 Build ModelSphere with Ant

Building the application is the process of generating the Java archives (.jar files) from the compiled classes (.class files). The Build operation is usually done with a build script (the build.xml file).

The build script also contains a javadoc task, to generate the Javadoc documentation at the build time.

Depending on your IDE, it is possible to invoke the build script directly from your development environment. Otherwise, the script can be invoked from a DOS shell or from a UWIN shell.

Before building the application and making a public release, developers must:

- Clean the output directory so that ModelSphere will create the locale.properties, the modelsphere.args and the modelsphere.properties files when being launched for the first time after installation.

- Not obfuscate the code (if this option is provided by an IDE). It is not necessary to obfuscate open source code, and error stack will be easier to analyze if the code is not obfuscated.

Developers should set the DEBUG variable in the org.modelsphere.jack.debug.Debug class to **false**, to prevent the end user to get messages targeted to a developer. If developers forget to set Debug to false, this task is automated with the build script.

7 Build the Plug-ins

Each plug-in must be deployed in a separate archive (.jar file). Each plug-in must contain one and only one class that inherits from the Plugin interface (directly or indirectly).   This class defines a PluginSignature variable.

```
public class ReportGenerator implements Plugin {

  private static final PluginSignature signature = new PluginSignature(
    LocaleMgr.misc.getString("ReportGenerator"),
    revisionString,
    ApplicationContext.APPLICATION_AUTHOR,
    dateString,
    201);
```

The qualified name of this class must be specified in the manifest file of the plug-in.

Properties Defined in the manifest.mf File	Description
Manifest-Version: 1.0	
Class-Path:	Ignored by PluginMgr, but could eventually allow to make available external libraries required by the plug-in.
Specification-Title: Open ModelSphere	
Specification-Version: 201	The minimal build version of ModelSphere requires to activate the plug-in. Must match with the number specified in the PluginSignature variable.
Specification-Vendor: Open ModelSphere	Can be the name of a company for a commercial plug-in.
Implementation-Title: Generate DDL	
Implementation-Vendor: Open ModelSphere	Can be the name of a company for a commercial plug-in.
Implementation-URL:	Ignored
Implementation-Class: org.modelsphere.sms.plugins.ReportGenerator	The name of the class that inherits from Plugin.
ModelSphere-Licence: 2	Optional

Table 24: Properties Defined in the Manifest File

Each plug-in is currently loaded with a specific class loader (an instance of PluginClassLoader). This class loader could be modified to read the Class-Path property (ignored for the moment). This type of improvement should be investigated as it could be interesting in the future.

The Implementation-URL is also ignored, but could eventually be used by PluginManager to specify an address for upgrading the plug-in.

The ModelSphere-Licence property is optional. If empty, the plug-in is always loaded. If it contains a value, it allows ModelSphere to decide whether the plug-in must be loaded. The value specified depends on predefined values in the SMSFilter.java and the Security.java files.

8 Running the Build Script

This section explains how to edit and run the build script under the Eclipse environment. The steps described here should be very similar if you use another IDE.

The file structure of your projects should be similar to the picture below. The workspace folder is the development root folder. It contains anything visible for the Eclipse environment. The four sub folders named "org.modelsphere.." correspond to the four Eclipse projects. The distribution project is a special project that contains no source code, but the scripts required to build the application and make a distribution package of ModelSphere. The guide project includes the developer's guide (this document) and the user's guide in HTML. The three other projects correspond to the source code projects.
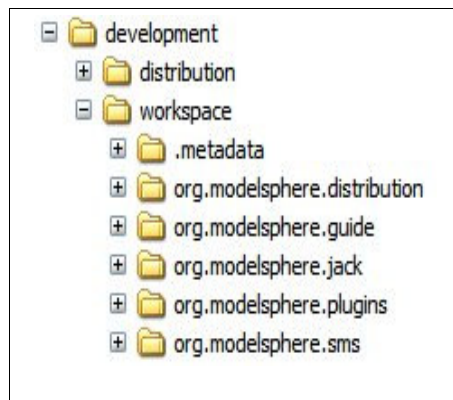
Figure 35: Development File Structure

The distribution folder is a sibling folder of the workspace, consequently not visible for the IDE. All the files built by the Ant script are generated in this folder; thus building the application for distribution has no impact on your development environment, and on the development team. If the distribution folder does not exist when the build script is executed, then it will be created at this moment.

The next illustration shows the contents of the Package Explorer in the Eclipse environment. As said before, only the contents of the workspace folder is visible for Eclipse users.

During the development phase, there is no need to access the distribution project. It is preferable that developers close this project. In fact, developers who do not have to make a release version of ModelSphere are never required to get this project. The distribution project should be opened in the deployment phase (when you produce the distribution package).
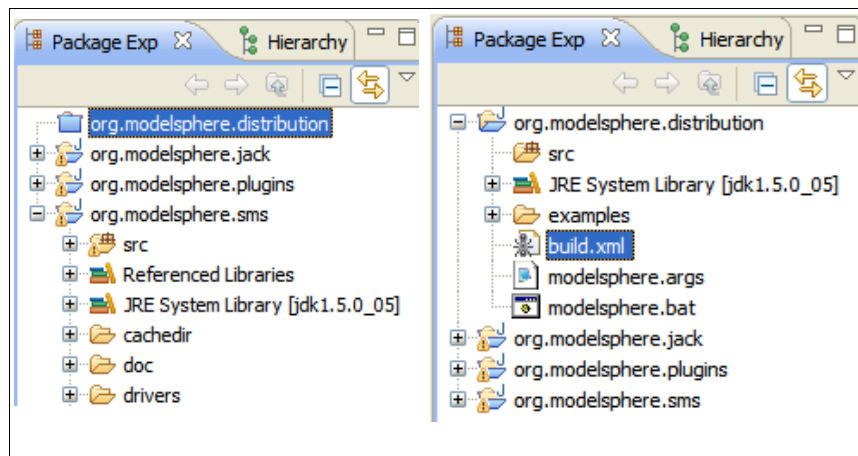
Figure 36: The Distribution Project in the Development (left) and the Deployment (right) Phase

Select the build script (named build.xml) and open it in a text editor. Change the following settings to reflect the configuration of your projects on your machine.

```
<!-- Folder properties -->
  <property name="root"         value="c:/development" />
  <property name="workspace"    value="${root}/workspace" />
  <property name="distribution" value="${root}/distribution" />
```
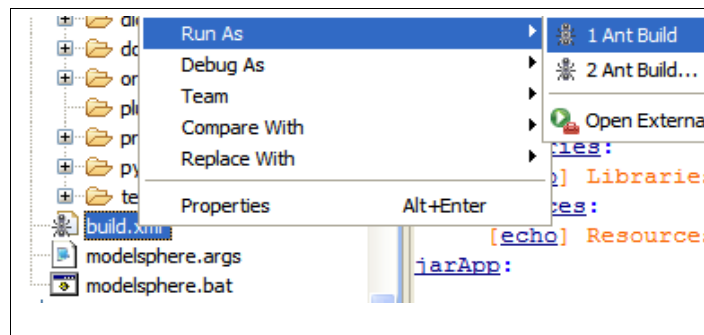
The build script can be run within the Eclipse environment, as illustrated below.

Figure 37: Running the build script

If the build is successful, you should get a corresponding message in the Eclipse Console view.

```
Buildfile: C:\development\workspace\org.modelsphere.distribution\build.xml
env:
     [echo] Java Home Directory : C:\Program Files\Java\jre1.6.0_02
init:
     [echo] mkdir distribution directory
compileJACK:
    [echo] Compiling Jack..
     ...
makeAll:
BUILD SUCCESSFUL
Total time: 3 minutes 14 seconds
```

If you open the distribution folder after having executed the build script, you should see the following contents.

Figure 38: The Content of the Distrubution Folder after Building ModelSphere

It is recommended to verify the distribution version of ModelSphere immediately after it has been built. Click the modelsphere.bat file to launch ModelSphere. When ModelSphere opens for the first time, it creates the locale.properties and the modelsphere.plugins files. In ModelSphere, open the **Help→Plugins..** menu to verify all the plug-ins have been successfully loaded.

9 Troubleshooting

This section lists the most common problems in the case you were not able to complete the previous section successfully.

➢ Problem: Unable to find a javac compiler

Solution: Verify the output of the ${java.home} variable echoed by the 'env' target of the build script. It should point to the location of your Java Development Kit (JDK). Make sure it refers to a JDK. A Java Runtime Environment (JRE) does not contain the Java compiler (javac).

➢ Problem: The Ant executable is not found (DOS console)

Solution: Check if the Ant path has been added to PATH environment variable:

- In Windows, select **Start→Setting→Control Panel→System** and click the **Advanced→Environment Variables** tab.

- In the **User variables** list, select the PATH variable (case insensitive) and click **Edit…**.

- At the end of the text box, add "c:\java\3rdparty\jakarta-ant-1.5-bin\jakarta-ant-1.5\bin (the semi-colon (;) is the separator character in DOS). Click OK and close all the windows of the control panel.

- In Windows, select **Start→Program→Accessories→Command Prompt**

- After the prompt, type "PATH" to know the value of the environment variable.

- Type "ANT" to verify that DOS accesses Ant.

- After the prompt, type "JAVA_HOME" tto know the value of the environment variable.

- Type "java –version" to know which version of JDK is used.

10 Packaging

Once a distribution version of ModelSphere has been created, you may use an installer program to package ModelSphere and produce an executable file (.exe). After downloading ModelSphere, end users click the executable to launch the installation on their machine. This significantly simplifies the task of installing ModelSphere.

InstallAnywhere is an example of an installer program, specialized for Java applications,

Describing how to use an installer program is beyond the scope of this document.

CHAPTER 14 - HOW TO

This chapter covers some typical tasks a ModelSphere developer may have to implement, describes each task step by step, and refers to the appropriate classes where changes are required. The tasks are listed from the most simple to the most complicated.

- How to Add a New Plug-in
- How to Add a New Action
- How to Execute a Long-Running Operation
- How to Add a New Notation/Style
- How to Add Validations
- How to Support a New Locale (User's Language)
- How to Update the Meta-Model (advanced)
- How to Update the Template Engine (advanced)

This chapter reviews the same concepts covered in the previous chapters, but the approach differs. Previous chapters were specialized in a particular topic (modeling framework, diagramming, user-interface) covered at a deep level. Each section of this chapter describes a typical development task that may implies several topics (modeling, graphics), covered superficially. Refer to the previous chapters to have more details on a particular topic.

How to Add a New Plug-in

This section explains the first steps of making a new generation plug-in.

How to Add a New Action

This section describes how to add a new action in ModelSphere, for instance how to spread the current diagram by a scale factor, and where to add this new functionality in the ModelSphere menus.

How to Execute a Long-Running Operation

When an operation takes time to execute, it is essential to provide a visual feedback to users, and offer to them to cancel the operation. This section explains how to use Controller dialog to monitor a long-running operation. This section also discusses how to undo, if possible, a long-running operation.

How to Add a New Notation/Style

ModelSphere already supports several notations for each kind of diagrams; for instance business process diagrams may be rendered by the Gane-Sarson, Yourdon-DeMarco or Datarun notation. This section explains how to create a new notation or a new style.

How to Add Integrity Rules

ModelSphere supports several integrity rules to verify if a model is valid. For instance, a table with no columns, or a process with no incoming or outgoing flow makes the model invalid. This section explains how to add new validation rules for a given type of diagram.

How to Support a New Locale (User's Language)

ModelSphere is and must stay an internationalized application, i.e. an application where no string displayed in the UI is hard-coded in the software. ModelSphere already runs in two languages, English and French. This section explains how to change the language in the user interface.

How to Update the Meta-Model (advanced)

ModelSphere allows the user to create and modify *models*. The framework that supports the different kinds of models is called the ModelSphere *meta-model*. ModelSphere's developers may have to update the meta-model to support new concepts in ModelSphere.

1 How to Add a New Plug-in

This section explains the first steps of making a new generation plug-in. Suppose you want to extend ModelSphere by adding a new feature, e.g. to select several tables, and to create a new diagram containing only the selected tables. This would be very useful if you have a large diagram with hundreds of tables, and you want to create a small diagram that shows a subset of all the tables.

Create a new project in your IDE. Do not mix your own code with the core part of ModelSphere. Give the name of your organization to the project.
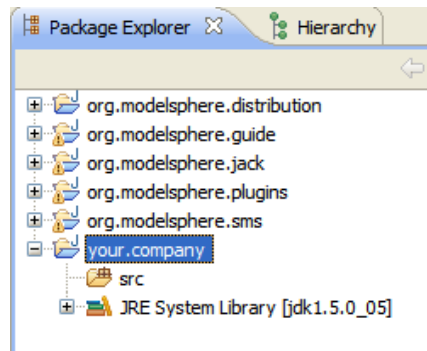
Figure 39: Create a New Project

Create a new package, and then a new class. Put the Plugin suffix at the end of the class name, to remember this class inherits from Plugin.
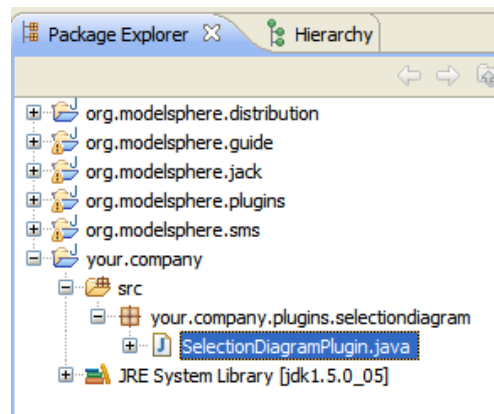
Figure 40: Create a New Package and a New Class

Make your project dependent on JACK and SMS.
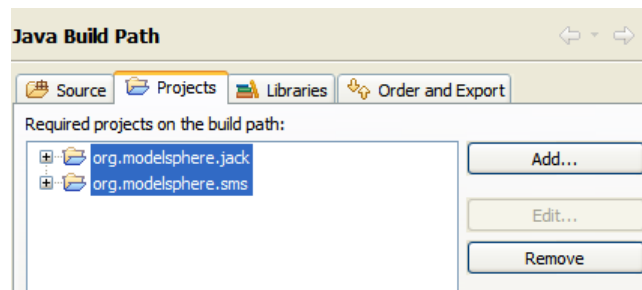
Figure 41: Create Dependencies among Packages

Make your plug-in class inheriting from JackForwardEngineeringPlugin, as shown below.

```java
package your.company.plugins.selectiondiagram;

import org.modelsphere.jack.srtool.forward.JackForwardEngineeringPlugin;

public class SelectionDiagramPlugin extends JackForwardEngineeringPlugin {
}
```

Implement the abstract method, to get rid of the syntax errors.

```java
public class SelectionDiagramPlugin extends JackForwardEngineeringPlugin {

  @Override
  public PluginSignature getSignature() {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  public Class[] getSupportedClasses() {
    // TODO Auto-generated method stub
    return null;
  }

  @Override
  protected void forwardTo(DbObject semObj, ArrayList generatedFiles)
```

```
    throws DbException, IOException, RuleException {
    // TODO Auto-generated method stub
  }

  @Override
  public void setOutputToASCIIFormat() {
    // TODO Auto-generated method stub
  }
}
```

Complete the plug-in signature. Notice that the plug-in name ("Diagram from Selected Elements") is hard-coded. It is not acceptable to do this in a released version, but the objective here is just to illustrate the plug-in mechanism. Also, Date is deprecated and should be replaced by Calendar.

```
private PluginSignature signature;
  private Date date = new Date(2007, 11, 21);

  public PluginSignature getSignature() {
    if (signature == null) {
      signature = new PluginSignature(
        "Diagram from Selected Elements",
        "1.0",
        "Your Company",
        date,
        900);
    }

    return signature;
}
```

Add the folder that contains the binary code (the .class files) to the plug-in path. This allows ModelSphere to search in this path when it starts up, and load the plug-in you have just written. Delete the modelsphere.plugins file, otherwise ModelSphere will read this file instead of scanning the folder structure defined by the -pluginpath program argument.
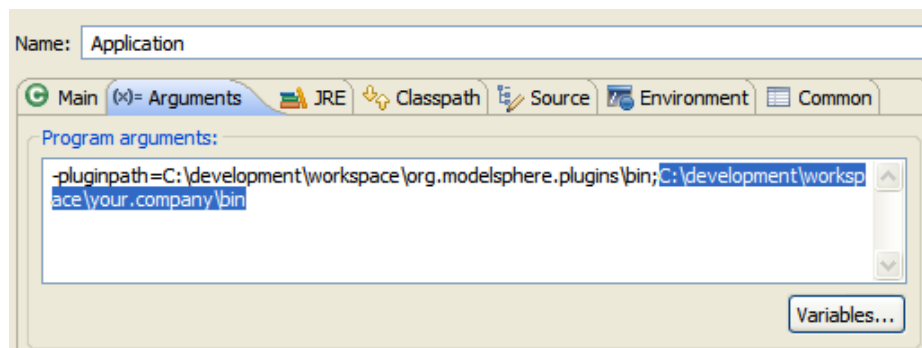
Figure 42: Adding a Folder to the Plug-in Path

When ModelSphere is running, open the **Help➞Plug-ins..** menu item to make sure your plug-in has successfully been loaded.
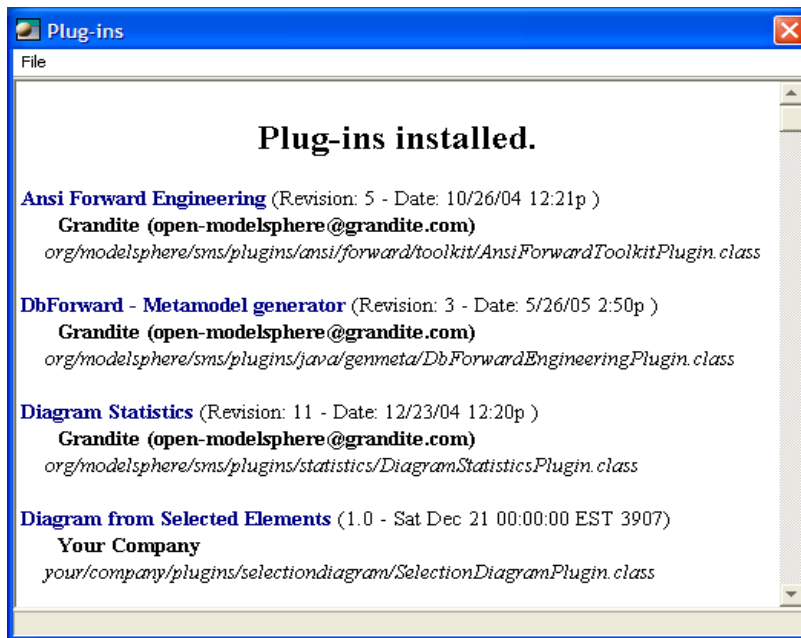
Figure 43: The Loaded Plug-ins

The plug-in is successfully loaded, but it cannot be called yet by the user. The next step is to allow the user to invoke it.

Your plug-in inherits from JackForwardEngineeringPlugin; this class defines the method getSupportedClass() that contains the information to which model classes your plug-in applies. You want to create a diagram from selected tables and views. Tables and views are implemented by the DbORAbsTable class.

Adapt the getSupportedClass() in the following way:

```
public Class[] getSupportedClasses() {
  Class[] supportedClasses = new Class[] {DbORAbsTable.class};
  return supportedClasses;
}
```

Re-start ModelSphere again. If the user selects one or several tables (or views) and right-clicks to get the pop-up menu, your plug-in should be available in the Generate menu item. It is where the forward engineering plug-ins are invoked.
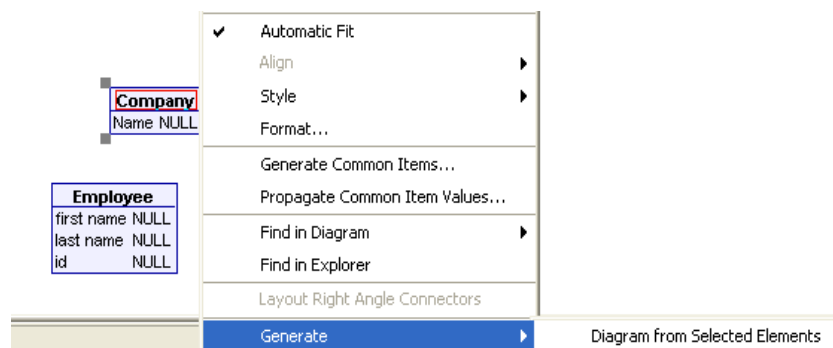
Figure 44: Invoking the Plug-in from ModelSphere

If the user selects the "Diagram from Selected Elements" sub item, ModelSphere calls the execute() method. Because the body of this method is empty, nothing happens when the user clicks the item. The objects passed as parameters correspond to the objects selected by the user. In the current case, the semObjs[] array contains one instance of DbORATable, named Company.

```java
/**
 * The entry point of the plug-in.
 */
public void execute(DbObject[] semObjs) throws DbException {

}
```

From there, you should be able to implement your plug-in code. Look at the ProcessTreePlugin class in the org.modelsphere.sms.plugins.bpm to know how to start a DB transaction, how to create a new diagram and new graphical objects, and how to internationalize your plug-in.

2 How to Add a New Action

This section describes how to add a new action in ModelSphere, for instance how to implement an actions that spreads the current diagram by a scale factor, and where to add this new functionality in the ModelSphere menus.

ModelSphere, as any application, can always be improved by adding new functionalities. For instance, if a diagram is overloaded, a developer may want to add a "Spread Diagram" function to expand all the graphical elements on a diagram.
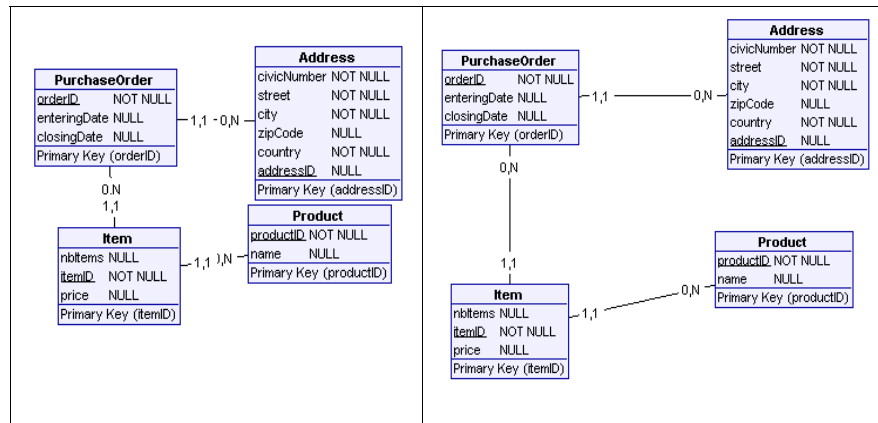
Figure 45: Before and After Spreading the Diagram

Another good functionality to implement would be a "Create Neighborhood Diagram" action; this would create a new diagram with the selected figure and all its immediate neighbors, i.e. figures directly connected to it. In the figure above, right-clicking the PurchaseOrder figure and calling this action would create a new diagram containing PurchaseOrder and its neighbors, Address and Item. This should be useful to comprehend huge diagrams.

All functions that a user can initiate must be implemented as an action. An action is a Swing concept that encapsulates a functionality and may be accessed by several places in the application (in the main menu, in the tool bar, in a pop-up menu, etc.). An action has a name, has a state (enabled or disabled) and may be associated to an icon.

To know how to implement an action, look for an existing one, such as the ShowAboutAction class in the org.modelsphere.sms.actions package. This class can be copied and renamed SpreadDiagramAction.

Many of the actions are singletons and created when the application starts. They are stored in the SMSActionsStore class.

Now the SpreadDiagramAction has been defined, the developer has to decide in which part of the application the user will invoke this action. The ShowAboutAction can be accessed in the application's main menu, and added in the MainFrameMenu class of the org.modelsphere.sms package. Obviously it is preferable to call the SpreadDiagramAction when the user right-clicks the diagram surface.
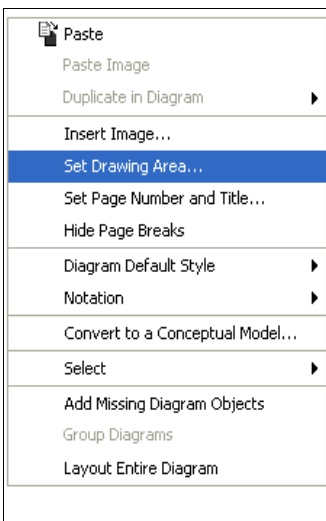
Figure 46: The Pop-up Menu Shown on a Diagram Right-Click

The "Set Drawing Area" functionality is implemented by the SetDrawingAreaAction class in the org.modelsphere.jack.srtool.actions package. It is referred by the SMSPopupMenuPool class in the org.modelsphere.sms.popup package. Look in these classes to know how to insert the SpreadDiagramAction in the diagram pop-up menu. If successful, the user should get a "Spread Diagram" item in the diagram pop-up menu.

➢ Refer to the section 9 on page 64 for more details about actions

The doActionPerformed() method of the action is invoked when the user triggers this action. Finally you should implement doActionPerformed() in order the spread the current diagram. The current diagram is obtained by calling the getFocusObject(), as illustrated below:

```
protected final void doActionPerformed(){
    FocusManager manager = ApplicationContext.getFocusManager();
    ApplicationDiagram diag = (ApplicationDiagram)manager.getFocusObject();
```

➢ Refer to the section 1.3 on page 59 for more details about the focus manager.

Get the diagram graphical object and implement the spreadDiagram() method with the diagram object as parameter. Because the operation is going to write the diagram object, it is required to encapsulate the operation in a write transaction. The spread transaction will appear in the command history and can be undone (under the Edit menu, the user will see "Undo Spread Diagram", where "Spread Action" is the English-localized string returned by the action.getString() method. All exceptions must be caught using the processUncatchedException() method.

```
DbObject diagGO = diag.getDiagramGO();
try {
    String transactionName = LocaleMgr.action.getString("spreadDiagram");
    diagGO.getDb().beginTrans(Db.WRITE_TRANS, transactionName);
    spreadDiagram(diagGO);
    diagGO.getDb().commitTrans();
} catch (Exception ex){
    DefaultMainFrame frame = ApplicationContext.getDefaultMainFrame();
    ExceptionHandler.processUncatchedException(frame, ex);
}
```

➢ Refer to the section 1.3 on page 24 for more details about transactions.

The spreadDiagram() method will iterate through all the diagram figures, and change their locations by multiplying their X,Y position with a scale factor (let's say 1.5). A scale factor less than 1.0 will actually *shrink* the diagram. The spreadDiagram() method throws DbException, you know that the caller will handle the exception, if it occurs.

```
private void spreadDiagram(DbObject diagGO) throws DbException {
      DbRelationN components = diagGO.getComponents();
      DbEnumeration enu = components.elements();
      while (enu.hasMoreElements()) {
          DbObject dbo = enu.nextElement();
          if (dbo instanceof DbSMSGraphicalObject) {
              DbSMSGraphicalObject go = (DbSMSGraphicalObject)dbo;
              Rectangle rect = go.getRectangle();
              rect = scale(rect);
              go.setRectangle(rect);
          } //end if
      } //end while
  } //end spreadDiagram()
```

➢ Refer to the chapter 7 on page 52 for more details about diagramming.

Based on that, a developer should know how to implement a "Spread Diagram" functionality in the application.

As a future extension, a developer may offer to let the user choose the scale factor instead of hard-code it in the SpreadDiagramAction. This can be done by adding sub menus under the "Spread Diagram" menu item.

3 How to Execute a Long-Running Operation

When an operation takes time to execute, it is essential to provide a visual feedback to users, and offer to them to cancel the operation. This section explains how to use the Controller dialog to monitor a long-running operation. This section also discusses how to undo, if possible, a long-running operation.

In the previous example (how to implement the Spread Diagram functionality), we assumed that spreading a whole diagram is a fast operation. It can be done on the UI thread (event-dispatch thread), without disturbing the interaction of the user with the interface. If the operation is long to complete, the operation must run on a dedicated thread. This table gives the recommended strategy to use depending on the duration of an operation.

Time to Complete the Operation	Appropriate Strategy
less than 0.2 second	Run on the UI thread
between 0.2 and 2 seconds	May run on a dedicated thread; the mouse cursor changes to an hourglass while the operation is running
more than 2 seconds	Should run on a dedicated thread; use the Controller to provide a visual feedback to the user

Table 25: The Appropriate Strategy According the Duration of an Operation

If an operation takes between 0.2 and two seconds, it may run on a dedicated thread at the condition the operation does not involve Db transactions. Remember that Db is not multi-threaded: you cannot start a transaction on a thread if another transaction is still active on another thread, otherwise Db will throw an exception.

➢ Refer to the section 1.3.1 of the chapter Modeling Framework (on page 25) for further details on Db and threads.

If an operation takes more than two seconds, it is recommended to use the Controller dialog. The Controller dialog (as illustrated below), offers to cancel the operation while it is running. A details panel, that can be shown or hidden, gives the user more details about the progression of the operation. It is possible to save the details to a text file. The Controller dialog optionally displays a progress bar to indicate to the user which percentage of the job has been completed.
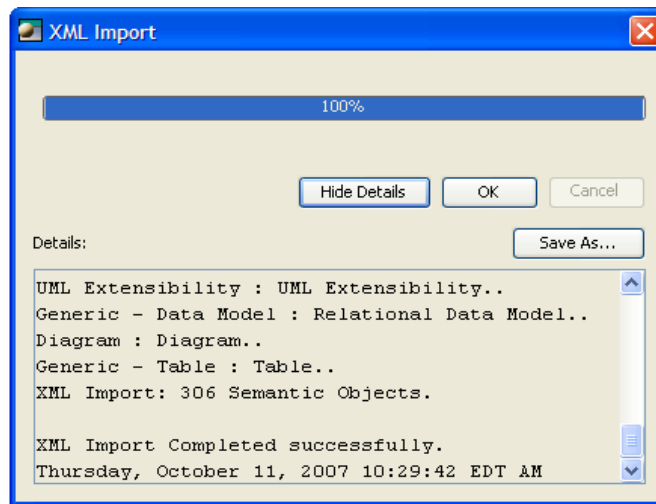
Figure 47: The Controller Dialog

➢ Refer to the section 8.1 on page 63 for more details about the controller dialog.

The developer must first create a SpreadDiagramOperation class that inherits from Worker. The controller is then instantiated, and the start() method is called to launch the operation, as illustrated below.

```
DbObject diagGO = ..;
Worker worker = new SpreadDiagramOperation(diagGO);
DefaultController controller = new DefaultController();
controller.start(worker);
```

The SpreadDiagramOperation class has to implement the runJob() method, which is a long-running operation. Most of the time is spent within the while iteration.

```
class SpreadDiagramOperation extends Worker() {
    private DbObject diagGO;

    protected void runJob() throws Exception {
        DbRelationN components = diagGO.getComponents();
        DbEnumeration enu = components.elements();

        //the time-consuming iteration
        while (enu.hasMoreElements()) {
            ..
        }
    }
}
```

The long-running operation now runs on a dedicated thread (because Worker inherits from java.lang.Thread) and its execution is monitored by the Controller dialog. The next thing to implement now is to give the user the option to cancel the operation while it is running.

The operation cannot be aborted at any moment, but at some synchronization points, called *check points*. A check point notifies the controller of which percentage of the task has been completed; the controller uses this percentage to increment the progress bar. The checkPoint() method also returns a Boolean value to communicate if the operation is in progress, or has been canceled by the user.

```
//the time-consuming iteration
int jobDone;
while (enu.hasMoreElements()) {
    jobDone = evaluateJobDone(); //returns a percentage between 0 and 100%
```

```
    boolean inProgress = controller.checkPoint(jobDone);

    if (! inProgress)
        break; //stop to iterate
    ..
}
```

The check point should be placed within a time-consuming iteration, to give visual feedback while the operation is running. If there are not enough calls to checkPoint(), the progress bar will not increase in a smooth manner. If checkPoint() is called too often, the overhead of monitoring the operation will cause the operation to take more time to execute. It's up to the developer to determine the optimal rate to call the checkPoint() method.

➢ For more details, see the DefaultController and the Worker classes in the org.modelsphere.jack.gui.task package.

Large transactions

In the section above, we have discussed time-consuming operations. A time-consuming operation does not necessarily mean that it involves a large number of DbObject, but often it does. All the changes to DbObject's must be nested into a transaction, that is normally reversible. But if the transaction is large, i.e. it accesses a huge number of DbObjects, it will require more memory and slow down the application.

After committing a large transaction, it is recommended to clear the transaction history in order to free memory, as illustrated below:

```
Db db = ..

try {
    db.beginTrans(Db.WRITE_TRANS, transactionName);
    //large transaction
    db.commitTrans();
    db.resetHistory(); //free the memory
} catch (DbException) {
    //handle exception
```

This will have the effect to clear the command history, and consequently the Undo action will be disabled under the Edit menu. If the history has to be reset after a transaction, it is a good practice to notify users and ask them if they want to proceed, knowing that they will be unable to undo anything if they perform the operation. This way, users have the option to save their current work before proceeding.

```
String msg = "Warning: You are about to perform an operation that\n" +
        "will clear the command history. It won't be possible to undo\n" +
        "previous actions.  Do you still want to proceed?";

if (userChoise == JOptionPane.YES_OPTION) {
    try {
        db.beginTrans(Db.WRITE_TRANS, transactionName);
        //large transaction
        ...
```

➢ Refer to the section 1.3 on page 24 for more details about transactions.

Of course, in a real development, the warning message cannot be hard-coded and must be placed in a properties file.

4 How to Add a New Notation/Style

One of ModelSphere's strengths is that the tool is not limited to a specific notation or methodology. Relational diagrams may be rendered with the Information Engineering or the Datarun notation, while business process diagrams may be rendered by a dozen of different notations.

A ModelSphere developer may have to implement an additional notation for relational diagrams (e.g. IDEF1X), or an in-house notation for business process diagrams. Let's say we need to create a new business process notation.

ModelSphere behavioral models describe the behavior of a system rather that its structure. Behavioral models include business process models, data flow models, and various UML models. All the behavioral notations are created in the initBeNotations() method of the DbInitialization class in the org.modelsphere.sms.db.util package.

Behavioral models support the same fundamental concepts, that are rendered differently according the notation used. These concepts are:

● The DbBEUseCase concept *processes and alters* data; depending on the notation, it is called a process unit (or simply a process), a task, an operation, or a use case. It can be decomposed in smaller process units.

● The DbBEFlow concept *transfers* data without modifying it; depending on the notation, it is called a flow, a message or a transition.

● The DbBEStore concept *stores* data for a certain period of time.

● The DbBEActor concept represents an external entity that interacts with the system you describe.

Each notation defines how to name a fundamental concept, and which shape to use to display it.

The following code shows the simplest notation you can implement. The code fragment may be inserted into the initBeNotations() method. Process figures will be rendered as small rectangles of 60 pixels by 25. Trying to create flows, data stores and external entities will fail because they are not defined yet in the notation.

```
// My Own Notation
DbBENotation myOwnNotation = new DbBENotation(thisProject);
myOwnNotation.setName("My Own Notation");
myOwnNotation.setTerminologyName(DbBENotation.GANE_SARSON);
myOwnNotation.setNotationID(new Integer(GANE_SARSON));
myOwnNotation.setMasterNotationID(new Integer(GANE_SARSON));
myOwnNotation.setNotationMode(new Integer(DbSMSNotation.BE_MODE));
myOwnNotation.setBuiltIn(Boolean.TRUE);

//Process
myOwnNotation.setUseCaseShape(SMSNotationShape.getInstance(
    SMSNotationShape.RECTANGLE));
myOwnNotation.setUseCaseZoneOrientation(
    SMSZoneOrientation.getInstance(SMSZoneOrientation.HORIZONTAL));
new DbBESingleZoneDisplay(myOwnNotation, DbBEUseCase.fName, true,
    ZoneJustification.getInstance(ZoneJustification.CENTER), false,
    BEZoneStereotype.getInstance(BEZoneStereotype.USECASE));
myOwnNotation.setUseCaseDefaultWidth(new Integer(60));
myOwnNotation.setUseCaseDefaultHeight(new Integer(25));
```

This code is intentionally simple for concision purposes. In a real development situation, programmers do not hard-code strings (like "My Own Notation") and do associate a specific ID instead of re-using the GANE_SARSON's notation ID.

Styles are very similar to notations and change the visual aspect of diagram figures. Notations are associated to diagrams, while styles are associated to individual figures. Thus, it is possible to have different styles on the same diagram, while a diagram has only one notation.

Look at various notations and styles defined in the DbInitialization class of the org.modelsphere.sms.db.util package for more details about these concepts.

5 How to Add Integrity Rules

ModelSphere supports several integrity rules to verify if a model is valid. For instance, a table with no columns, or a process with no incoming or outgoing flow makes the model invalid. This section explains how to add new validation rules for a given type of diagram.

Users editing a model are not disturbed by error messages each time they make their model invalid (for instance when they create a table without column), but they have the possibility at any moment to verify if all the integrity rules are respected. Of course, before saving or generating data definition language (DDL) scripts, it is strongly recommended to verify if the model is compliant with integrity rules.

Integrity rules may be applied on behavioral (business process) and relational modeling. They are located in the org.modelsphere.sms.plugins.integrity plug-in. These two methods are the entry points of the verification process:

- BEIntegrity.verifyIntegrityRules()
- ORIntegrity.verifyIntegrityRules()

Additional rules must be added in one of these two methods. Integrity rules include errors and warnings. It is recommended to verify the errors first, and the warnings after. This way, errors (if any) will appear at the top of the verification report, before the warnings.

As any strings displayed to the user, all the error messages must be adapted to the locale. Error messages are defined in the MiscResources.properties file.

6 How to Support a New Locale (User's Language)

ModelSphere is and must stay an internationalized application, i.e. an application where no string displayed in the UI is hard-coded in the software. ModelSphere already runs in two natural languages, English and French. This section explains how to change the language in the user interface.

> *In this section, it's important to distinguish natural languages from programming languages. A natural language refers to the user's language (for instance English or French); while a programming language refers to the development language (for instance Java or Python). A locale is the use of a natural language and formats preferred by a cultural community, such as the number format and the date format.*

Internationalizing an application has several advantages. An internationalized application may be adapted to another language with a translation effort, but almost without any development cost. Because all the strings displayed in the UI are removed from the code and placed in separate properties files (text files), a non-programmer may translate them without any coding skills. Even if the application runs only in one language, a developer can make language mistakes. If the text appears in a text file, a technical writer (with no programming skills) can easily find and correct the error. Any software pretending to be a world-class application and to meet highest software engineering standards must be internationalized.

The table below shows a part of a properties file for the English and the French locales.

English	French
Aborting=Aborting	Aborting=Interruption en cours
Add=Add[>Mnc<]A	Add=Ajouter[>Mnc<]A
AddToList=Add to List	AddToList=Ajouter à la liste
AngularLine=Free Angle	AngularLine=Angle libre
Apply=Apply[>Mnc<]Y	Apply=Appliquer[>Mnc<]Q
Back=Back	Back=Retour
Bold=Bold	Bold=Gras
BoldItalic=Bold Italic	BoldItalic=Gras Italique
Browse=Browse	Browse=Parcourir
Cancel=Cancel	Cancel=Annuler

Table 26: ScreenResources.properties and ScreenResources_fr.properties

Each line contains a key/value pair; the equal character (=) is the separator. The key is the name of the property as referred in the Java code; a key only contains letters and digits. The key is the same for all languages. The value is what users see when they use the application. A value sometimes contains special codes, such as [>Mnc<], explained later.

The properties must be sorted alphabetically: it helps to avoid duplicated entries, and it allows the application the get an entry quickly using a binary search.

The properties file with no language suffix (ScreenResources.properties) is the default language. This language is used if the language specified by the user is not available. A property file may have a language suffix, such as _fr. The suffix is the ISO 636-1 code for the language (e.g. 'de' for German, 'es' for Spanish, and so on). If translators want to make a German version of ModelSphere, they have to create German properties files (here ScreenResources_de.properties) from a language they know and translate all the values (keys must stay untouched, otherwise the application won't find them).

Translators are invited to use the localization word list recommended by Sun. These words are commonly used by software programs; refer to Sun's site for the complete list[13].

English	French	German	Spanish
Apply	Appliquer	Übernehmen	Aplicar
Bold	Gras	Fett	Negrita
Browse	Parcourir	Durchsuchen	Explorar
Cancel	Annuler	Abbrechen	Cancelar

Table 27: Translation List Recommended by Sun

There are five kinds of properties files: ActionResources, MessageResources, ScreenResources, DbResources, MiscResources.

ActionResources contains the text used in the Swing actions (see the 'How to Add Action' section). In English, the action text must be capitalized as a headline. All words must be capitalized except articles and conjunctions. Refer to the Swing guidelines for a complete description of the headline capitalization convention[14].

ActionResources contains special codes, such as [>Mnc<] for mnemonics and [>Acl<] for accelerators (also known as Keyword Shortcuts). A mnemonic is a letter underscored in an action (for instance in the Edit menu, C and P are the mnemonics for the Copy and Paste actions). Shortcuts are used to trigger an action without navigating in the menu. CTRL-C and CTRL-V are shortcuts for the Copy and Paste actions. Reuse the standard shortcuts as much as possible[15].

MessageResources contains text displayed in a message dialog. The text must be capitalized using the sentence capitalization convention.

ScreenResources contains text for buttons and labels used in a screen. The text must be capitalized as headlines, but no shortcuts must be used.

DbResources contains text for meta-model concepts. Contrary to other kinds of properties files, the DbResources is generated by the genmeta plug-in; consequently it is useless to edit these files, since changes will be lost the next time meta-model classes are generated. If mistakes occur in the DbResources.properties file, open the genmeta file and make the correction in the alias field. If mistakes occur in the DbResources_fr.properties file, make the correction in the fr_alias field. To support other languages, the translator can manually edit the DbResources_xx.properties for the generated DbResources.properties.

➢ Refer to the section 4.3 on page 78 for more information about DbResources.

---

13 http://java.sun.com/products/jlf/ed2/book/Appendix.C.html
14 http://java.sun.com/products/jlf/ed2/book/HIG.Visual3.html
15 http://java.sun.com/products/jlf/ed2/book/Appendix.A.html#29092

MiscResources contains miscellaneous text that does not fit in the ActionResources, MessageResources or ScreenResources.

Replacement codes

Sometimes message text contains replacement codes, such as {0} and {1}. In this case, the value text is in fact a pattern passed to the format() method of the java.text.MessageFormat class. The pattern is passed with an array of strings, where {0} refers to the first string.

It may happen (but it's rare) that the order of replacement codes differs depending on the language used. In this case, the replacement codes are sorted according the English usage.

| TodayMonthDay=Today: {0} {1} | TodayMonthDay=Aujourd'hui le {1} {0} |

Table 28: Properties in the English and French locales

Java code does not have to care about the order of words in a particular natural language when it constructs messages.

```
String pattern = LocaleMgr.message.getString("TodayMonthDay");
String message = MessageFormat.format(pattern,
    new String[] {month, day});
JOptionPane.showMessageDialog(parent, message);
```

If ModelSphere runs in French, the pattern returned by the getString() method places the {0} after {1}, that gives: "Aujourd'hui le 12 octobre". Translators must pay attention to the order of replacement codes when they translate patterns.

➢ Translators must also be aware of the limitations of the MessageFormat's format() method. Consult the section 2.3 on page 75 to know how to use MessageFormat.format().

When all the properties files have been translated into a target language, how to cause ModelSphere to run in this locale? The locale to use when ModelSphere starts up is defined in the locale.properties file. This file is located in the application's root folder. The user can edit its contents and replace 'en' with the new target language code.

```
#Specifies in which locale the application starts up
Locale=en
```

Normally, users do not have to edit manually this file. They use **Tools→Options...** in the ModelSphere environment to change the locale. Unfortunately, only English and French are supported using this interface:
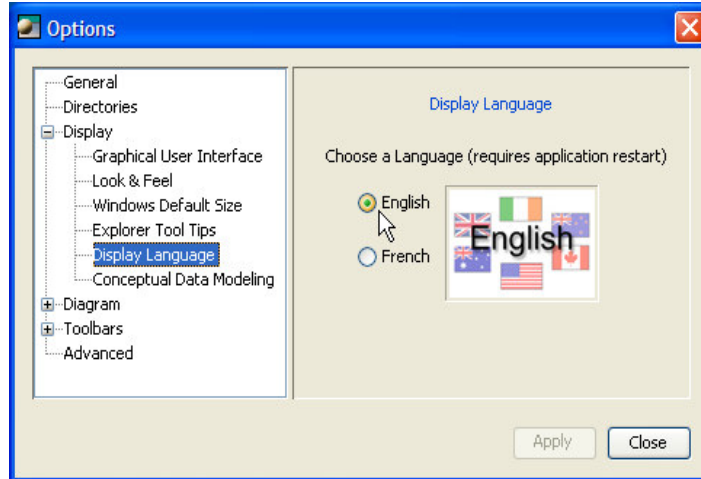
Figure 48: Display Language Option

Update the DisplayLanguageOptionGroup class in the org.modelsphere.sms.preference package to add new languages.

➢ Refer to the section 6 on page 62 for more details about the preferences.

Tip: If developers want to improve a given screen but they are not familiar with the code, how can they quickly find which class in the code is related to this screen?

Suppose we want to improve the Display Language screen, as shown above. The screen has the label "Choose a Language". First, using their IDE, developers could make a search on the "Choose a Language" in all the .properties files of the project. Developers should find the ScreenResources.properties file, that contains:

```
ChooseLanguage=Choose a Language
```

Once you have found the ChooseLanguage key, search this string in all the *.java files of the project. You should find the DisplayLanguageOptionGroup.java file. By examining this class, you will find a main() method that performs a unit test. You can run it to verify the impact of your code changes without restarting the whole application.

---

7 How to Update the Meta-Model (advanced)

ModelSphere allows the user to create and modify *models* (this includes relational data models, UML class models and business process models). The framework that supports these different kinds of models is called ModelSphere's *meta-model*. End users cannot modify the meta-model, they simply use it to create their own models. ModelSphere's developers may have to update the meta-model to support new concepts in ModelSphere. This section explains how to do it.

WARNING: Updating the meta-model implies re-engineering the heart of ModelSphere, it is not a trivial task to take lightly. It should be reserved to the most experienced Modelsphere developers. Before updating the meta-model, verify if existing features in the current meta-model cannot achieve the desired goal. If they can't, consider adding user-defined fields (UDFs). Updating the meta-model is always the last option.

If you adapt the meta-model for your own purposes, remember that the models you will generate won't be compatible with those created by the current version of ModelSphere. When the meta-model is updated, ModelSphere's version number must change to reflect the modifications. This prevents ModelSphere to open models created with a version superior to the current one.

Here are the steps to follow to update the meta-model:

- Start ModelSphere, and open the genmeta.sms file. Edit the meta-model diagrams;
- Fill the genmeta-specific fields;
- Use the genmeta plug-in to generate model classes;
- Copy the generated files in the db folders;
- Change the version converter;

7.1 Edit the meta model file

The meta-model is a physical file, genmeta.sms, which has the same format of any other .sms file. This file contains only a class diagram. To make sure models written with a previous version of ModelSphere will still be opened by a new version without losing anything, avoid to delete or to modify fields that already exist in the meta-model.

Guideline #24: It is not recommended to modify or delete existing classes or fields in the meta-model, it will break the backward compatibility; just add new concepts without changing existing ones.

If developers want to eliminate unnecessary fields in the model, they should mark them deprecated and adapt the model converter (described below) to migrate values from the old fields to the new fields.

7.2 Fill the genmeta-specific fields

The genmeta model is a standard class model, but it contains additional user-defined fields (UDFs). These UDFs are used for many purposes, including naming concepts in the English and the French locale. The fields are required to produce valid model classes.

➢ Refer to the section 1.9 on page 35 for more details about user-defined fields.

7.3 Generate the meta-model classes

This step generates the meta-model classes (the Java files) from the ModelSphere meta-model (the genmeta.sms file). This is similar to a Java forward engineering, but the generated classes cannot work by themselves: they are designed to work in the ModelSphere environment only.

Generating the meta-model requires of course the metagen plug-in to be loaded. Verify it is loaded by choosing **Help→Plugins..** in the application menu.

> **DbForward - Metamodel generator** (Revision: 3 - Date 5/26/05 3:50p)
> **Internal Development Team**
> *org/modelsphere/sms/plugins/java/genmeta/DbForward.class*

Table 29: The Meta-Model Generator Plug-in is Loaded

If the genmeta plug-in is loaded, a new option should appear under the Generate menu item. Select the node of the sms package, and right-click to get the pop-up menu.
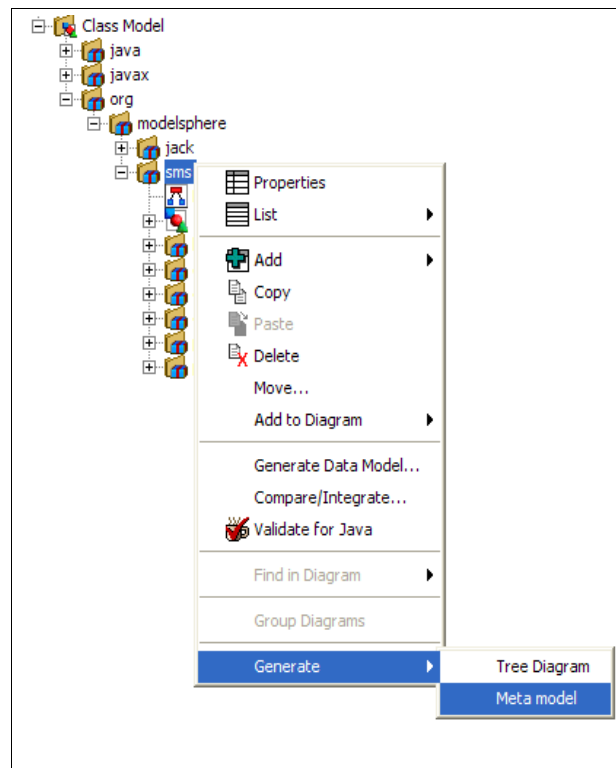
Figure 49: Generating the Meta Model

Meta-model classes have names starting with Db and are located in db packages. Having generated meta-model classes, you should get the following (generated) packages:

org.modelsphere.sms.db	org.modelsphere.sms.or.db
org.modelsphere.sms.be.db	org.modelsphere.sms.or.generic.db
org.modelsphere.sms.oo.db	org.modelsphere.sms.or.ibm.db
org.modelsphere.sms.oo.java.db	org.modelsphere.sms.or.informix.db
	org.modelsphere.sms.or.oracle.db

It is recommended not to generate the meta-model classes directly in the development folders. Set the generation folder to an empty folder to avoid to overwrite existing meta-model classes.

7.4 *Copy the generated files in the db folders*

Once the Java files have been generated, open the files and examine their contents to make sure they have been properly generated. If they have been generated with errors, you have to find out the source of the errors, and generate the meta-model classes again.

Here are some possible causes that produce erroneous meta-model classes:

- The 'alias' and 'fr_alias' UDFs have not been filled, or they are filled with invalid characters;
- The ' icon name' UDF refers to a file that does not exists;
- The contents of the 'default value' field is invalid for this type of variable;
- The 'method body' field contains Java code that does not compile;

Remember, the genmeta plug-in is not a Java compiler, it will generate the contents of fields "as is" and it's up to the developers who update the meta-model to be aware of what they do.

If you are satisfied with the generated code, copy it to the development folders (and overwrite existing meta-model classes).

Guideline #25: Do not edit meta-model classes (DbXXX.java) manually. These files are generated by the genmeta plug-in. If you edit these classes in an IDE environment to fix a bug, the next time you will generate the meta-model classes, your changes will be lost.

7.5 *Change the version converter*

When the meta-model changes, the format of .sms files also changes. Consequently, you may experience problems if you try to open a .sms file saved with a previous version of ModelSphere.

To ensure backward compatibility with previous versions of ModelSphere, it is required, when you open a .sms file, to verify with which version of ModelSphere it has been saved, and to convert it to the current version of ModelSphere. The class SMSVersionConverter in the org.modelsphere.sms package is responsible to convert models to the current version.

```
private void convertProject(DbSMSProject project) throws DbException {
    ..
    MainFrame mf = ApplicationContext.getDefaultMainFrame();
    DbSMSProject newProject = (DbSMSProject)mf.createDefaultProject(null);

    try {
        switch (oldVersion) {
            case 1:
                convertToVersion2(project); //Convert to 2.0 (build = 215)
                // no break, cascade to next converter.
            case 2:
                convertToVersion3(project);//Convert to 2.0 (build > 215)
                // no break, cascade to next converter.
            case 3:
                convertToVersion4(project);//Convert to 2.1 (build >= 300)
```

```
            // no break, cascade to next converter.
        ...
```

Note that the switch structure in the convertProject() method does not contain a break statement after each case (one of the rare situations where it's not an error to not use a break after each case statement). Be careful and do not accidentally add a break here, otherwise you will break the cascading chain.

Each time a new version of ModelSphere is released and the meta-model has changed since the last version, it is required to adapt the convertProject() method and to implement a convertToVersionX() method.

8 How to Update the Template Engine (advanced)

ModelSphere uses two different techniques to generate code: template files (.tpl) and plug-ins written in Java. A template file is a text file that defines the structure of generated code; it is processed by the template engine and generates code. Template files are very similar to the Velocity scripts[16]; and users who already know Velocity should be familiar with the template files. The template engine applies a given template file to the chosen ModelSphere model, and generates a text file as output.
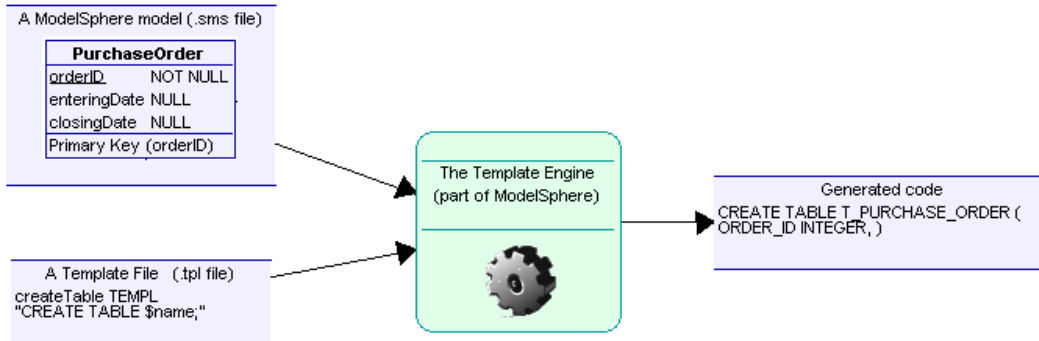
Figure 50: The Template Parser Generation Process

Template files follow their own syntax, defined by the template language. A grammar file (Template.jj) defines the template language. The JavaCC[17] parser generator is used to generate the parser code from the grammar, that creates Java classes in the org.modelsphere.jack.templates.parsing package.

Guideline #26: Do not manually edit the classes in the org.modelsphere.jack.templates. parsing package, otherwise your changes will be lost the next time you use JavaCC to generate the parser.

The grammar file contains the syntactic rules of the language, and semantic actions. Semantic actions are Java code embedded in syntactic rules. You should not have to update the grammar file. If you do, avoid modifying the syntactic rules as much as possible, and limit your changes to the semantic actions, because changing syntactic rules has a greater impact on the generated parser than changing semantic actions.

Guideline #27: If you have to change the grammar file, avoid modifying the syntactic rules and limit your changes to semantic actions.

If you examine the contents of the grammar file, you will find that semantic actions are limited to one or two lines of code. Most of the processing is done by the code called by semantic actions, not by semantic actions themselves. Because parser generators do not verify the syntax of semantic actions (they copy the semantic actions "as is" into the generated parser), it is harder to fix a bug that occurs in semantic actions rather than in the code called by semantic actions.

Guideline #28: Semantic actions must stay as simple as possible; most of the processing should be done by the methods called by the semantic actions, not by the semantic actions themselves.

---

16 http://velocity.apache.org/
17 https://javacc.dev.java.net/

For further details about the grammar syntax, go to the JavaCC site; you can also read comments embedded in the grammar file.

Eclipse developers who want to maintain the grammar file can use the JavaCC Eclipse plug-in[18]; albeit not mandatory, this GPL tool is useful to edit .jj grammars and to generate parsers. This plug-in includes a syntax coloring editor and adds a "Compile with JavaCC" item on the grammar file selected in the Eclipse's explorer. Generated Java sources are marked in the explorer to distinguish from ordinary (not generated) Java sources, as shown below.
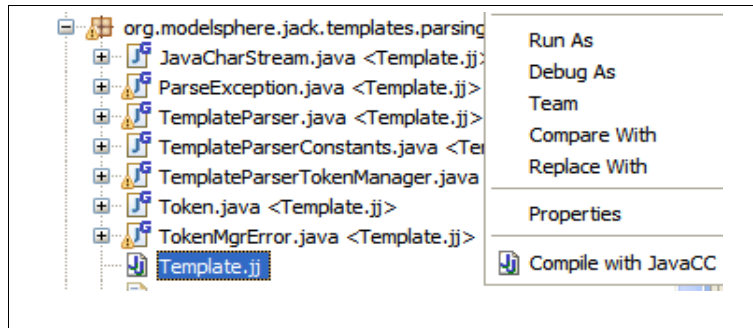
Figure 51: The JavaCC Eclipse Plug-in

---

18 http://sourceforge.net/projects/eclipse-javacc

CHAPTER 15 - GUIDELINES AND PROGRAMMING CONVENTIONS

This chapter includes general programming conventions for Java, standards to build GUI elements, and guidelines to maintain the documentation. All the guidelines introduced in the previous chapters are listed in this chapter for a quick reference.

1 Introduction

In all projects, conventions are required to ensure uniformity and readability. This is more important as the number of contributors increases. Most of the ModelSphere conventions are inherited from the code conventions defined by Sun Microsystems inc. These conventions are not discussed in this chapter. Use the following reference for additional information:

http://java.sun.com/docs/codeconv/index.html

Steve McConnell's Code Complete, Second Edition, also contains numerous programming guidelines that have inspired ModelSphere's developers. Another good on-line reference is the Java Programming Style Guidelines, available at:

http://geosoft.no/development/javastyle.html

2 Naming Conventions

Using naming conventions provides the following:

- Self-documenting for easier maintenance.

- Avoiding bugs during the development process.

- Information in variables' names on what types of operations are allowed.

Standard naming should be used as much as possible. Design patterns provide names to identify their different components. Using existing names greatly increases the reading and understanding of a feature.

3 Language

This is important to distinguish the application's language from the language used in source files. Source files are written using one language, but the application's user interface can be translated in any language.

English must be used to name classes, methods and other identifiers in the code. This is mandatory because several ModelSphere's classes inherit from 3rd-party libraries written in English, and the object-oriented overriding mechanism only works when ModelSphere's methods have the same name as 3rd-party methods.

Comments added to the source files must also be in English, because

- they can be grabbed by the javadoc tool,
- some of them are automatically generated in English (by genmeta, by JavaCC, by an IDE plug-in like VisualEditor[19]),
- to allow a maximum of users to read and understand the software.

All the strings used in the graphical user interfaces (GUI), generated files, or any strings visible to a ModelSphere user should be put in localized files. Localized resources are contained in properties files and consist of pair of key-value. All the keys should be named to be easily identified and understood by a non developer. These keys should be specified in English.

➢ Refer to the chapter 10 on page 74 for more details about internationalization.

4 Code Formatting

Most guidelines provided here are derived from the default formatting defined in the Eclipse IDE. Developers are strongly encouraged to choose automatic formatting if using Eclipse.

Using uniform code formatting provides the following advantages:

- Facilitates maintenance within a multi-developers context;
- Facilitates bug detection during development;
- Makes the code aspect consistent within the organization;
- Facilitates new developers to get familiar with the application code.

5 GUI Conventions

ModelSphere is based on Swing, and is as much as possible[20] compliant with the user-interface guidelines defined by Sun.

Developers are invited to build their windows and dialogs with the GridBagLayout graphical layout. This layout is the most flexible among those found in the Swing library, it allows users to resize their windows without causing problems, and prevents localization issues.

Guideline #29: GridBagLayout should be used to construct complex windows and dialogs.

Developers can code user interfaces from scratch, or they can use a GUI editor that generates the first version of the user interface. NetBeans is packaged with its own GUI editor. Visual Editor[21] is a popular editor available for the Eclipse environment.

---

19 http://www.eclipse.org/vep/WebContent/main.php
20 http://java.sun.com/products/jlf/ed2/book/index.html
21 http://www.eclipse.org/vep/

6 Guidelines for Documentation

Even if the documentation (including the developer's guide (this document) and the user's guide) is written by several people at different moments, it must be consistent to help readers as much as possible. Several image snapshots are inserted in the documentation to illustrate the text. In the future, it is recommended to apply the following guidelines for the snapshots included in the documentation:

- All the snapshots must be rendered by the same look and feel (L&F). It is suggested to use the Windows XP L&F. In the future, Windows Vista may be the recommended L&F. Motif should be avoided (too different from the other Windows applications).

Motif L&F	Windows XP L&F	Windows Vista L&F	Mac OS L&F
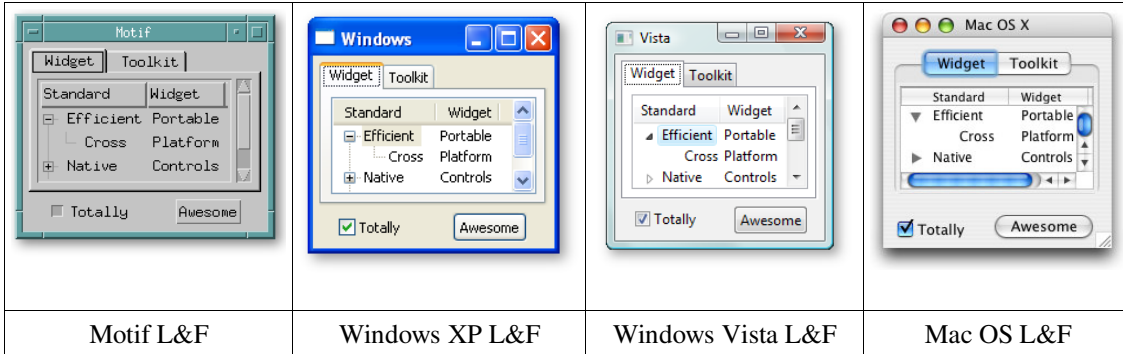
Figure 52: Various Look and Feels[22]

- The new ModelSphere icon ( ) must be used in replacement of the former ModelSphere icon ( ) in all the snapshots.

- When diagrams are illustrated, standard styles and colors must be used (green for processes, yellow boxes with red borders for classes, and so on). For the business process diagrams, it is recommended to use the Gane and Sarson notation.

Table	Process	Class
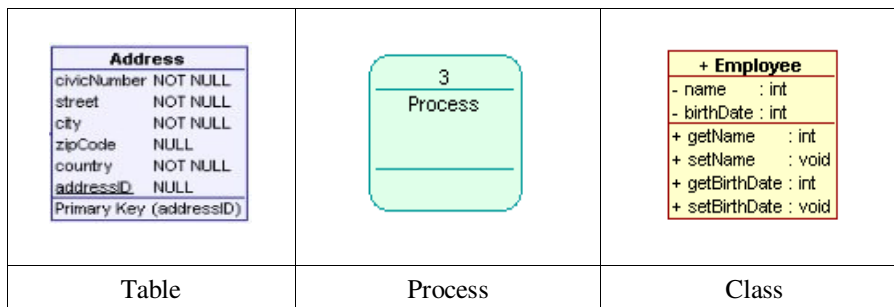
Figure 53: Standard Styles and Notations

- The tool used to capture screen images should be able to capture dialog windows in addition to screen areas.

---

22 Credits: http://www.eclipse.org/swt/

- In the future, we recommend to use an official example model for ModelSphere to illustrate various diagrams. The example model should be small (less than ten tables or classes), but should cover all the major constructs. The example model should not be too specific (tables named Soap and Detergent), and not too generic (tables named Table-1, Table-2). In his book Eclipse Modeling Framework, Budinsky[23] uses real-word concepts understood by everyone such as PurchaseOrder, OrderStatus, Item, Address classes. His model includes twelve classes; the major concepts as aggregation, inheritance, navigability, recursion were all covered. We recommend an example model similar to Budinsky's one for ModelSphere.

7 Additional Recommended Guidelines

Besides the code conventions defined by Sun Microsystems inc., the ModelSphere development team has defined the following guidelines:

Guideline #1: Never import an SMS class from the JACK library. JACK must stay independent from SMS..p. 10

Guideline #2: Classes in SMS depend on the ModelSphere meta-data. If an SMS class is independent from the meta-model, consider to move it to the JACK framework..p. 10

Guideline #3: Try to avoid inter dependencies among plug-ins; this will allow to publish a plug-in without having to change and publish other plug-ins...p. 11

Guideline #4: Do not modify SMS classes in a package ending with .db, otherwise the modifications will be lost the next time the meta-model is generated. ....p. 12

Guideline #5: Before adding a new feature in the JACK library, verify if it does not already exist in the Java standard library..p. 13

Guideline #6: Packages in the JACK library should be named awt, io, net, text, util to indicate they are implementing features not found in the standard Java library...p. 13

Guideline #7: When new features are added in the standard Java library, consider to remove unnecessary classes in JACK and to use standard classes instead...p. 13

Guideline #8: All the action classes (that inherits from the Swing's AbstractAction) should have the Action suffix and should be place in a package named .actions. This helps developers to find the action classes quickly. ....p. 15

Guideline #9: Write the enu.close() statement immediately after declaring the DeEnumeration enu variable to ensure that the enumeration will be closed. Then insert the while iteration between the declaration and the close statement..p. 28

Guideline #10: Use the refresh listener to refresh graphical objects after a modification on the model has been reported; use the update listener to change values on model elements after a modification on the model has been reported. ....p. 34

Guideline #11: Catch any exception (not only DbException) and call processUncatchedException, which ensures that any open transaction (necessary if the exception is not DbException) is aborted and displays the message of the exception on the screen. ....p. 35

---

23 Example models can be downloaded at: www.aw.com/budinsky

APPENDIX A - GLOSSARY

Actor
An entity that is outside of the system, but interacts with it. Also known as external entity. Used in business process modeling and UML use case diagrams.

Alias
(Relational Modeling) An alternative name of relational elements (tables, columns, ..) in addition to logical name and physical name. The alias field in the meta-model is used to generate GUI names.

Association
A binary connection between two tables or two classes. ModelSphere does not support N-ary associations.

Association Role
The two directions of an association. An association role may have a label, a multiplicity and a navigability.

Background
The color used to paint the interior of a figure (graphical object). Compare: Border.

Behavioral Feature
(Object-Oriented) Features of a class describing its behavior, the methods of a class. Opposite: Structural Feature.

Border
The color used to paint the frame of a figure (graphical object). Compare: Background.

Cardinality
The number of elements of a set. An association role may define a minimal cardinality and a maximal cardinality. See Multiplicity.

Classifier
(Object-Oriented) A class or an interface.

Column
A variable defined in a relational table. Equivalent to a field of a class.

Common Item
A commonly known designation of a real world element that can be used within models of different types. For instance the 'Product Unit Price' common item could be linked to a column in a relational database and to a field in a class model.

Compilation Unit
(Object-Oriented) The Java source file, a file with the .java extension.

Concrete Class
(Object-Oriented) A class that can be instantiated. Opposite: abstract class.

Constraint
In the relational paradigm, a restriction applicable to a column or a set of columns. In UML, an extensibility mechanism that can be associated to any model element. Contrary to UML stereotypes, a model element may have more than one constraint.

Design Panel
The property sheet displayed below the explorer in the ModelSphere interface, and that allows a user to modify the properties of a selected object.

Diagram
The graphical representation of a set of graphical objects. Usually, a diagram is a graph of nodes and edges. A model may be associated to zero, one or several diagrams. See Model.

Domain (modeling)
A knowledge body or activity defining a set of concepts and a terminology proper to the practicians of this domain. Domain logic and business logic are synonymous.

Domain (types)
A set of pre-determined values. Equivalent to the enumeration construct introduced by Java 1.5.

Drawing Area
The span of pages available for elements. When a diagram is created, its drawing area is set to one page, but can be extended by the user accordingly to their needs.

Duplicate (in a diagram)
Another graphical object of the same semantic object in the same diagram, thus changing the value in one graphical object automatically updates the other one. Duplicated graphical objects are identified by the 1/2, 2/2 sequence.

Explorer
The hierarchical tree structure displayed above the Design Panel in the ModelSphere interface.

Forward Engineering
The process of generating code from a ModelSphere model. Forward engineering a relational data model produces a data definition language (DDL) script, while forward engineering a class model produces Java code. Opposite: Reverse Engineering.

Free Line / Box / Round Box / Oval Tool
Graphical objects not associated to any semantic object, and used to visually enhance the diagram.

Graphical Object
The graphical representation of a semantic object, displayed as a diagram element. A graphical object is associated to one semantic object. Synonymous: Diagram figure. See: Semantic Object.

Identifier
A name that uniquely identifies an element within its name space. The class name is an identifier because two classes cannot have the same name within the same package, while the name of a method is not an identifier because two methods of the same class can have the same name (as long as their signature is different).

Import / Export

The operation of reading from or writing to an external format. Contrary to the load and save operations, importing and exporting does not imply to read or write the entire information (elements of the imported files may be ignored).

Initialization Block

(Object-Oriented) In Java, the block of statements being executed when an instance is created, or when a class is loaded (in this case, static initialization block). This construct is supported by ModelSphere.

Inner Class / Inner Interface

(Object-Oriented) A classifier defined within a top-level classifier. Classifiers may be nested several times. This construct is supported by ModelSphere.

Integrate; Integration

Brings changes of one model into another one.

Integrity

See Referential Integrity.

Interface

(Object-Oriented) A classifier that only contains constants and abstract methods.
(Target Systems) A set of plug-ins that perform forward and reverse engineering for a particular DBMS.
(User Interface) The graphical interface between the end user and the application code.

Layer (module)

A layered system is an ordered set of virtual worlds, each built in terms of the ones below it and providing the basis of implementation for the ones above it. Knowledge is one-way: a subsystem knows about the layers below it, but has no knowledge of the layers above it. A supplier-client relationship exists between lower layers (providers of services) and upper layers (users of services)[24]

Layout

(Graphical Layout) optimal arrangement of diagram figures, which avoids overlapping of figures and minimizes line crossings.
(Swing) arrangement of Swing components (buttons, text fields, check boxes) when the user resizes the dimension of a Swing container.

Link (model)

A ModelSphere model that contains links. Links associate a source model element to a target model element; the source and the target elements may belong to models of different types.

Localization

The process of adapting a software to a particular locale, by displaying text in the end user's language and formatting dates and numbers accordingly.

---

24 Quoted from [Raumbaug91], page 200

Logical Name

(Relational Modeling) the name of a construct used in a relational model. Complementary: Physical Name.

Look & Feel

The way GUI components (buttons, text fields, check boxes) are rendered graphically (the look) and react to the user's command (the feel). ModelSphere supports the Windows and the Metal Look & Feel.

Magnifier

A small diagram view showing the current diagram (the one having the user's focus) with of zoom factor of 1.0, notwithstanding the zoom factor set for the current diagram. The magnifier view follows the movement of the mouse pointer. See Overview.

Merge

(Process Modeling) The operation of merging a process model from an external project to a process (marked external) of the current process model. The merge operation can only be performed after the split operation. This allows to paste a process specification that was elaborated off line. Opposite: Split.

Method Overloading.

(Object-Oriented) The declaration in the same classifier of several methods sharing the same name but having different parameter signatures. Do not confuse with method overriding.

Method Overriding

(Object-Oriented) The redefinition of the method in a subclass that replaces the method definition found in a superclass. Do not confuse with method overloading.

Model

(Object-Oriented) The container of several semantic objects. A model does not necessarily have a graphical representation, but it can have several ones. See Diagram.
(User Interface) The data associated to GUI widgets. For instance combo boxes, spinners render data stored in their model.

Multiple Inheritance

(Object-Oriented) The ability for a classifier to have more than one inheritance link. Java supports single inheritance of classes and multiple inheritance of interfaces. Multiple inheritance of classes is allowed in ModelSphere, but the "Validate for Java" will report errors.

Multiplicity

A specification of the valid cardinality values for a set. Multiplicities are assigned to association roles.

Name Space

A space in which names must be unique. The name space of a field is its classifier, and the name space of a classifier is its immediate package.

N-ary Association

An association that can have more than two ends. Not supported by ModelSphere.

Navigability

The ability of a class to navigate to an associated class. By default, associations are navigable on the both directions. An association end that is not navigable is graphically represented with an arrow.

Notation

The way to graphically represent diagram figures. A diagram has only one notation. Compare: Style.

Note

A comment attached to a diagram figure. Notes are depicted as a rectangle with the top-right corner folded over. A note has no semantic meaning, it is only used to document the diagram.

Operation Library

(Relational Modeling) The container of procedures and functions, linked to a specific target system.

Overview

A small diagram view showing the entire current diagram (the one having the user's focus). The zoom factor is automatically adjusted to show the entire diagram in the small view. Compare: Magnifier..

Package

(Object-Oriented) A construct that contains classifiers and sub packages. Packages can be nested at any level

Partition (module)

Partitions vertically divide a system into several independent or weakly-coupled sub-systems, each providing one kind of service.[25]

Physical Name

(Relational Modeling) the name of a construct used in a relational database. Physical names must conform to naming rules defined by each target system. ModelSphere may generate physical names in a batch from logical names. Complementary: Logical Name.

Plug-in

A Java archive that extends the functionality of ModelSphere.

Primary Key

(Relational Modeling) A key is a set of columns whose values must be unique. A table has only one primary key, but can have several other unique keys, also called alternative keys.

Project

In ModelSphere, the container of several models. Each project is saved is a separate file (a .sms file). ModelSphere allows the user to open several projects at the same time, and to copy/move model elements from one project to another.

Python Shell

A dialog window in ModelSphere in which the users enter and execute their Python commands. The shell is available from the **Tools→Python Shell** menu.

Qualifier

(Process Modeling) A decorator icon used to qualify a process model element.

Redo / Undo

The Undo action cancels the user's last action and restores the state of the model before the user's action. The Redo action cancels the Undo action; it is only available after performing an Undo action. ModelSphere allows multiple undo/redo actions.

---

25 Quoted from [Raumbaug91], page 201.

Referential Integrity
Model correctness, based on relational rules.

Refresh
An action that redraws diagrams. Changing model elements should be reflected immediately in the diagram, without having to ask users to refresh their diagrams.

Relationship
A semantic link between model elements. It includes associations and inheritance links.

Reposition (Labels)
Association labels can be moved anywhere in the diagram, and after several diagramming manipulations, labels may be placed far away from their original location, making hard to understand the diagram. The reposition command allows the user to place a label over its original association end.

Reverse Engineering
The process of creating a ModelSphere model from a database or from Java source files. Do not confuse with "Decompilation", the process of re-constructing the source code from the binaries, and often forbidden in software licenses. Opposite: Forward Engineering.

Role
The end of an association.

Semantic Object
A model element, which can have none, one or several graphical representations. See Graphical Object.

Signature
(Object-Oriented) A method's parameters and their types.

Split
(Process Modeling) The operation of moving of given process to a separate model, allowing two modelers to work independently on two separate models. In the original model, the split process is marked as an external process, preventing the user to decompose it. Opposite: Merge.

Stamp
A graphical element that identifies the author and the version of a diagram, and gives a description of the diagram.

Status bar
The bar at the bottom of the ModelSphere application frame that provides information on the activity in progress.

Stereotype
A predefined or user-defined semantic type associated to any model element. A model element can have only one stereotype. A specific stereotype can be associated to a given type of model element. Stereotypes are a UML construct, but they can be used in any kind of models.

Straighten
A graphical link may comprise several intermediate points. The straighten action removes the intermediate points of the link.

Structural Feature
Features of a class describing its structure, the fields of a class. Complementary: Behavioral Feature.

Style
A set of graphical properties (border color, line width, font, etc.) associated to a graphical object. A diagram may contain graphical objects of different styles.

Swimlane
(Process Modeling) The partition of diagrams to organize the responsibilities of the actions. These partitions often correspond to organizational units.
(UML) The partition of activity diagrams to organize the responsibilities of the action states.

Synchronization
The generation of a script that updates the structure of a database rather than creating it from scratch. Forward engineering scripts contain CREATE TABLE commands while synchronization scripts contain ALTER TABLE commands.

Tagged Value
(UML) The definition of a property whose value is a character string. Users may define their own tagged values. Tagged values are one extension mechanism of the UML language; the two other extension mechanisms are stereotypes and constraints. ModelSphere supports the user-defined properties (UDP) mechanism, more powerful than tagged values because UDP are not necessarily character strings. See user-defined properties.

Target System
(Relational Modeling) The specification of a system for physical data models. Target systems define DBMS-specific data types for columns. Target systems include DB2, Informix, Oracle, SQL Server.

Template
A script written in the Template Language, used by ModelSphere to extract information from a model and convert it to an external format (e.g. HTML, SQL).

Tile (horizontally; vertically)
The action of arranging document windows (for instance diagram windows) into the workspace (the panel at the right of the Explorer and the Design Panel).

User-Defined Properties (UDP)
A property defined by the user and assigned to one of the following types: string, multi-line text, Boolean value, integer and floating-point value. ModelSphere allows users to add their UDPs by using the **Display→Project User-Defined Properties** menu. UDPs are displayed in italics in the Design Panel. Note: UDPs are called user-defined fields (UDF) in the code.

Validate
The action of verifying the correctness of a model according referential integrity rules (for a data model) or Java language rules (for a class model).

View
(Relational Modeling) A relational construct whose column values actually originate from table column values.

APPENDIX B - ACRONYMS

ANSI
American National Standards Institute

ANT
Another Neat Tool (The Apache tool to build software applications)

API
Application Programming Interface

AWT
Abstract Window Toolkit

CVS
Concurrent Versions System

DB
Data Base (ModelSphere's Modeling Framework)

DBMS
Data Base Management System

DDL
Data Definition Language

DOS
Disk Operating System

EMF
Eclipse Modeling Framework

GIF
Graphics Interchange Format

GNU
GNU's Not Unix (an open source project)

GPL
General Public License

GUI
Graphical User Interface

HTML
Hypertext Markup Language

ID
Identifier

IDE
Integrated Development Environment

J2SE
Java 2, Standard Edition

JACK
Java Abstract Classes Kit (ModelSphere-specific)

JDBC
Java Database Connectivity

JDK
Java Development Kit

JPEG
Joint Photographic Experts Group

JRE
Java Runtime Environment

JVM
Java Virtual Machine

OO
Object-Oriented

OODBMS
Object-Oriented Data Base Management System

OS
Operating System

PK
Primary Key

QA
Quality Assurance

RAM
Random Access Memory

RDBMS

Relational Data Base Management System

SMS

Shared Modeling Software (ModelSphere-specific)

SQL

Structured Query Language

UDF

User-Defined Field (ModelSphere-specific)

UDP

User-Defined Field (ModelSphere-specific)

UK

Primary Key

UI

User Interface

UML

Unified Modeling Language

URL

Uniform Resource Locator

XML

Extensible Markup Language

APPENDIX C - LOCALIZATION SCRIPTS

The 'str' script:

```
#
# Display all lines containing a double-quoted string containing at
# least one letter except lines containing one of the following:
#
# - all LocalMgr methods : getString(), getUrl(), getMnemonic(),
# - getImageIcon(), getToolTip(), getAccelerator(),
# - getResource(),
# - Debug.assert(),
# - System.out.println()
# - System.getProperty()
#
# Usage: go at the root folder of your sources, and type 'str'; this will
# search all .java files, recursively, and write in the standart output
# the list of localizable strings.
#
# Use "str > z.txt" to store the localizable strings in the file z.txt.
#
# Furthur improvements : ignore strings between /* and */, or after //
#
# known bug: a letter between two strings is considered part of a string
#  e.g. : " ( " + nb + " ) "      => string " + nb + " matches
#
# HOW TO UNDERSTAND THE SCRIPT
# awk 'pattern {action}' files    => Perform <action> for each line of <files>
#                                    matching <pattern>
# .+                              => Any number of any character, but at least
#                                    one character
# /\".+\"/                        => A double-quoted string containing at least
#                                    one character
# /\".*[a-zA-Z].*\"/              => Double-quoted string contained at least
#                                    one letter
# '/\".+\"/ ? pattern1 : pattern2 => If current line contains a string, match
#                                    pattern1; elsewhere match pattern2
# (pattern1 && pattern2)          => Match if line matches both pattern1 and pattern2
# (!pattern1 && !pattern2)        => Match if line matches neither pattern1 nor pattern2
# /getString\(/                   => Regular expression "getString("
# /ZZZ/                           => Match regular expression "ZZZ" (should be false
#                                    most of the time)
# print FILENAME ":" FNR          => Print the file name and the line number,
#                                    separated by a ":"
# print ;                         => Print the whole line that matches the previous
pattern
# print " "                       => Print a blank line
# find . -name \*.java -print     => List of all Java files, including those
#                                    in subdirectories, recursively
# grep -v "db/"                   => Remove from the list all the files contained
#                                    in directories ending by "db"
#
awk '/\"[^\"]*[a-zA-Z]+[^\"]*\"/ ? (!/getString\(/ && !/getUrl\(/ && !/getMnemonic\(/
&& !/getImageIcon\(/ && !/getToolTip\(/ && !/getAccelerator\(/ && !/getResource\(/
&& !/System\.getProperty\(/ && !/System\.out\.println\(/ && !/Debug\.assert\(/
&& !/Debug\.trace\(/ && !/NOT LOCALIZABLE/ ) : /ZZZ/ {print FILENAME ":" FNR "\r"; print;
print " \r"}' `find . -name \*.java -print | grep -v "db/"`
```

The 'missing' script:

```
#
#  missing : script to find missing keys in .properties files
#
#  How to use :
#    1) open UWIN shell
#    2) go to the directory where you want to launch the script; all
#       subdirectories under the current directory will be scanned,
#       recursively.
#    3) type 'missing'.  You can also redirect the output in a file
#       by typing 'missing > missing.txt'
#
#  Keys missing in either English or French .properties files will
#  result in an application which is not completely localized, ie.
#  English keys may appear instead of locale text in English or
#  French version.
#
#  The output follows the DIFF syntax (d for deleted lines,
#  c for changed lines and a for added lines).
#
#  For instance, this output:
#    2308,2309d2273
#    < jack/src/org/modelsphere/jack/baseDb/international/DbResources.properties
loginFailed
#    < jack/src/org/modelsphere/jack/baseDb/international/DbResources.properties
loginRepository
#
#  means that loginFailed and loginRepository have not been found in
#  DbResources_fr.properties file.
#
#  Algo :
#  touch keys_en.txt                            : if file doesn't already exist, avoid a
#                                                  memory fault in furthur redirection
#  awk -F=                                      : consider '=' has the field separator
#  /.+=.+/                                      : condider only lines with a '=' character
#                                                  (skip other ones)
#  {print FILENAME,$1}                          : print filename and the 1st field
#                                                  (the key before the '=' sign)
#  find . -name \*Resources.properties -print   : find recursively files ending with
#                                                  Resources.properties
#  sub(/_fr/, "", res)                          : remove _fr in filename
#  diff                                         : looks for differences between English
#                                                  keys and French keys (should be the same)
#
#
touch keys_en.txt;
awk -F= '/.+=.+/ {print FILENAME,$1}' `find . -name \*Resources.properties -print | grep -
v "bak/" | grep -v "classes/"` > keys_en.txt;
touch keys_fr.txt;
awk -F= '/.+=.+/ {res = FILENAME; sub(/_fr/, "", res); print res,$1}' `find .
-name \*Resources_fr.properties -print | grep -v "bak/" | grep -v "classes/"` >
keys_fr.txt;
echo 'diff english french'
diff keys_en.txt keys_fr.txt
```

The 'res_en' script:

```
#
# Gathers all the localized strings contained in resource files. Output
# can be redirected as the following:
#   $ res_en > res_en.txt
#
# This generated file can then be used to:
#   -be check by a word-analyzer to find out English mistakes
#   -easily detect which words have a British variant (center, color, etc.)
#
# This script has a French counterpart, res_fr
#
# Explanation of the AWK command:
#   -F=      : use '=' as the field separator (default separator is whitespace)
#   .+       : any character, repeated
#   .+=.+    : select strings containing at a '=' character, with at least one character
before and
#            at least one character after (other strings are ignored).
#   $2       : print the 2nd field, ie character after the '=' sign
#   find     : select files ending with Resources.properties, that excludes French property
files
#   sort -f : ignore case during sorting
#   sort -u : identical lines are only displayed once
#
awk -F= '/.+=.+/ {print $2}' `find . -name \*Resources.properties -print` | sort -fu
```

APPENDIX D - FILE TYPES

The following table lists the file types contained in the ModelSphere distribution. The 📄 icon identifies text files, the ⊙ icon identifies binary files, and the 🖼 icon identifies images.

File Extensions	Description	Type	Defined By
args	ModelSphere's arguments	📄	ModelSphere
bat	Batch Files	📄	Microsoft
css	Cascade Styles Sheet	📄	W3C
class	Java Byte Codes	⊙	Sun
dtd	Document Type Definition (tree.dtd)	📄	W3C
gif	Graphics Interchange Format	🖼	CompuServe
html	Hypertext Markup Language	📄	W3C
jar	Java Archive	⊙	Sun
java	Java Source Code	📄	Sun
jj	Grammar Files	📄	JavaCC
jpeg	Joint Photographic Experts Group	⊙	JPEG
js	JavaScript (xmlTree.js)	📄	Sun
mf	Manifest Files	📄	Eclipse
odt	Open Office Writer	⊙	OpenOffice
pdf	Portable Document Format (generated by Open Office Writer)	⊙	Adobe
plugins	List of Loaded Plug-ins	📄	ModelSphere
png	Portable Network Graphics	🖼	W3C
properties	Locale-specific properties	📄	Sun

File Extensions	Description	Type	Defined By
py	Python Scripts	📄	Python
sms	Shared Modeling Software (generated by ModelSphere)	⊛	Microsoft
sql	Structured Query Language (generated by ModelSphere)	📄	ISO
tpl	Template Files	📄	ModelSphere
typ	Target System Type Files	📄	ModelSphere
txt	Text Files (readme.txt)	📄	Microsoft
xls	Excel Sheets	⊛	Microsoft
xml	XML	📄	W3C
zip	Zipped Files	⊛	PKZIP

Table 31: The Files Included in a ModelSphere Distribution

REFERENCES

[Budinski] : Eclipse Modeling Framework, Budinsky and al., The Eclipse Series, Addison-Wesley, 2004, 680 pages.

[Fowler] : Patterns of Enterprise Application Architecture, Martin Fowler, Addison-Wesley, 2002, 512 pages.

[Gamma95] : Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, E., et al, Addison-Wesley, Reading, MA, 1995, 374 pages.

[McConnell] : Code Complete, Second Edition, Steve McConnell, Microsoft Press, 2004, 960 pages.

[Rumbaugh91]: Object-oriented modeling and design, J. Rumbaugh, Blaha, Pemerlani, Prentice-Hall, 1991, 500 pages.

INDEX