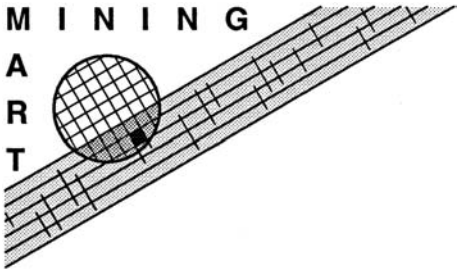

Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
Deliverable No. D12.2

Description of the M4 Interface used by the HCI of WP12
Deliverable D12.2

Bert Laverman and Olaf Rem

Perot Systems
NL-3821 AE, Netherlands
{Bert.Laverman, Olaf.Rem}@ps.net

July 16, 2002

Abstract

In this document the M4 Interface is described. It is the software component that is used by the Human Computer Interface that is being developed in WP12. It is responsible for providing access to the M4-Relational Metadata-Schema (see deliverable D7a). In this way an extra layer is constructed between client programs and the database. This takes away the burden for client programs to implement their own database access code and gives more flexibility when the implementation of the database schema is changed. The document discusses the architecture, design choices, installation and configuration, the current state and shows example code how to use the interface.

Contents

Chapter 1

Introduction

The objective for work package 12 is to provide users with a graphical user interface for the specification of an entire preprocessing task. It should enable the user to transform business data into a form that is ready for mining (the learning stage in the knowledge discovery process). The users that are targeted to use the HCI are the case designer and the database administrator. The case designer does the main part of the work. He specifies a conceptual model and decides which operators to apply and in what order. It is the task of the database administrator to map the conceptual model to the relational model: a necessary step before executing a case.

The most important components in the architecture that is being developed in WP12 are the Chain Editor, Concept Editor, Compiler and the M4 Interface. The M4 Interface is responsible for providing access to the M4-Relational Metadata-Schema. In this way an extra layer is constructed between client programs and the database. This takes away the burden for client programs to implement their own database access code and gives more flexibility when the implementation of the database schema is changed.

The Chain Editor and the Concept Editor will use the M4 Interface to create and manipulate meta data. Examples of meta data objects that can be manipulated with the interface are: Case, Concept, Relationship, Operator, Step, BaseAttribute and ColumnSet. The interface provides methods to create, retrieve, update and delete these objects. Further information about the WP12 HCI requirements (including the M4 Interface) can be found in the document by the Fraunhofer Institut Autonome Intelligente Systeme and Perot System Netherlands titled *MiningMart Human Computer Interface, Requirements and specifications* that was presented in March 2002.

This document further discusses the M4 interface architecture, design choices, installation and configuration and the current state of the interface. In the appendix information is included about programming standards used in developing the M4 interface and example code is included showing how to use the interface.

Chapter 2

Interface architecture and design

2.1 Interface architecture

The position of the M4 Interface can best be shown using the picture of the HCI architecture. This is shown in figure 2.1. The Concept Editor and the Case Editor are part of the presentation layer. They should not contain much business logic. The business logic is part of the business layer and handles the communication of the presentation layer with the database layer and the Compiler. The M4 interface forms a buffer between the business logic and the database. It provides methods for creating, updating, deleting, finding and retrieving information in or from an M4 instance. The Compiler manages the compilation and execution of (parts of) a Case.

In figure 2.2 the architectural view is presented, showing the three tier model superimposed on the major components of figure 2.1. It also shows which elements are part of the M4Interface and how the different components are distributed over the client, the application server and the database server. In figure 2.2 we introduce the Client Object Library (COL). It abstracts the data centric view used in the data layer for the application client and hides the communication with the application server. Within the application server we will have some (as few as possible) Stateful Session Beans, which keep track of individual clients and their interactions with the data. This is the level where locking of a case is implemented, so as to prevent access to a case when it is being updated, and conversely prevent an update session to start when read-only sessions are active. Further "down" into the data layer Stateless Session Beans are used to provide access to the data stored in the RDBMS.

We have chosen to use Session Beans rather than Entity Beans to access the database. Important for this choice is to consider the amount of data involved, the usage of the data and the differences between Session

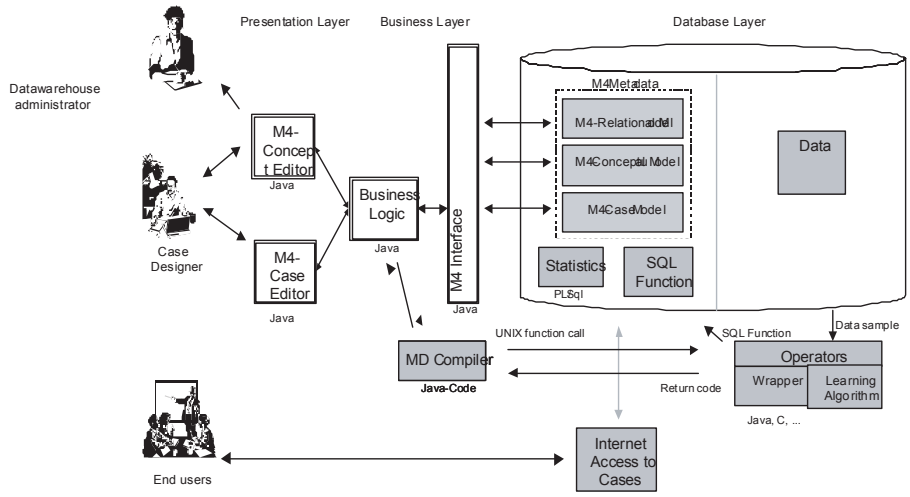

Figure 2.1: Conceptual view of the architecture for the MiningMart system showing users, layers and different components in the system (based on previous drawings of UniDo and SwissLife). The objects with double borders are to be implemented in WP12.

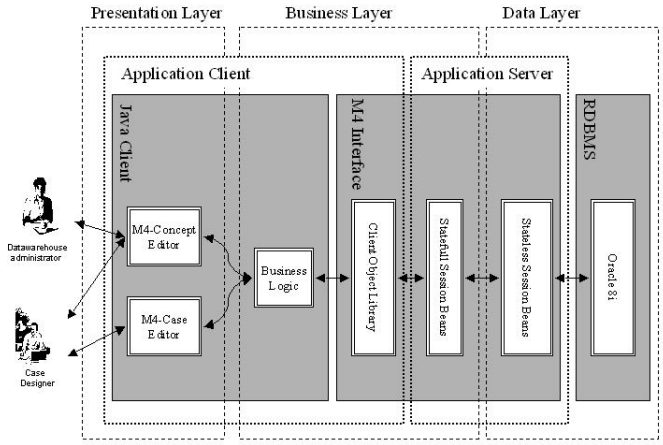

Figure 2.2: View of the WP12 architecture focusing on elements of the M4 Interface.

Beans and Entity Beans. Considering that the example case of the Mailing Action has about 400 steps, that each step may have several inputs and that every input may be related to many other objects (e.g. a Concept has FeatureAttributes) it is clear that the M4 Relational Metadata Schema will store a large number of objects. We expect that when working on a Case

there may be frequent updates to meta data objects like Steps, Parameters, Concepts and FeatureAttributes. Using Session Beans for database access is quite different than using Entity Beans. With Session Beans oneself has to implement the code to access the database, Entity Beans on the other hand can use Container Managed Persistence. Then the application servers container is responsible for storage and a lot less code needs to be produced. Entity Beans can be seen as a representation of a database row and have a higher memory overhead that Session Beans. Considering our expectations about the amount of data and data usage and the larger overhead of Entity Beans we decided for using Session Beans.

2.2 Design remarks

The M4 Interface uses three major design patterns: "Value Objects", "Factory", and "Facade". In this section we will introduce these patterns, and describe how they are applied in the M4Interface design.

Design Patterns have become more formalized over the last few years, leading not only to more generally accepted patterns, but also standardized names and terminology. The three patterns described here will be familiar to most developers, if not in name, then certainly in their structure, as they represent common approaches for layering an application over several tiers.

2.2.1 Value Objects

In the context of a J2EE application server, a common problem is the overhead of communications between the client software and the application server tier. Even presuming a high-bandwidth, high-speed network between the two, querying several thousands of server objects for their several attributes value will most certainly cause noticeable delays in response times to the end-user. To solve this, related data is bundled in so called Value Objects, and transported between the two tiers in bulk. The most common form in a J2EE environment is the usage of JavaBean compliant object representing one row of data from a database table. References to other objects can be represented by identifying values for the referred objects, or by actually adding that object as well and storing a reference to it. Javas RMI transport layer will then provide efficient delivery to the client. A second form usable in situations where large collections of objects are needed, is to return them packaged in a suitable container object.

This pattern has as major drawback that the client receives a local copy, which may over time become outdated, when other clients modify the original data in the database. For the MiningMart system this will not be an issue, as at the session level access will be restricted, preventing all access for new clients as soon as a read/write session is opened.

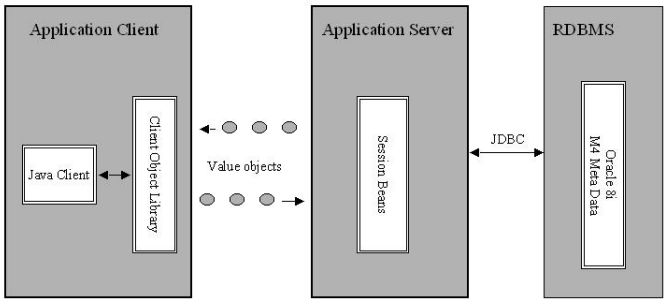

Figure 2.3: Value objects are used by the M4 Interface between the COL and the session beans.

Figure 2.3 shows that the COL and the session beans use value objects to exchange data with each other.

2.2.2 Facade

An example of a facade is a film-set, where houses in a street turn out to be no more than just the front wall. This image has been applied since days of old in programming, allowing different implementations to be used through a common interface.

For Java two major approaches are:

- using interfaces to describe how the provided functionality can be used, and
- using abstract objects which provide not only the interface, but also some common functionality.

For the M4 Interface both approaches are used. The package miningmart.m4 contains java interface objects like Case and Concept that describe the methods that objects implementing these interfaces should offer. The package miningmart.m4 also contains the abstract M4Interface class that provides some common functionality.

2.2.3 Factory

This pattern most often appears in conjunction with interface patterns such as Facade. The idea is that, rather than providing classes to be instantiated, a factory object is provided which will perform the instantiation. The former approach has as its main disadvantage that the user needs to know what the specific classname is of the class to be instantiated, whereas a factory object can do this without revealing that information. Factory objects often use

the Facade pattern to hide the actual factory object itself. As an example of a simple factory, the javax.swing.BorderFactory class defines static factory methods to create different styles of borders. A more flexible approach is employed by java.awt.Toolbox, which only defines a static method for returning a factory object of that same class. The methods for creating graphical element objects are methods of this instance. This allows the static method to select a suitable factory object, which can then perform the construction of the actual objects needed. The M4 Interface uses this last approach to allow clients to create an instance of some subclass of miningmart.m4.M4Interface through a static method called getInstance(). This instance is then an object from another package.

Chapter 3

Interface implementation

The COL can be accessed through interfaces and classes defined in the miningmart.m4 package. Most of the elements in this package are interfaces, implemented by classes in another package. Data is transferred between the application server and the client using Value Objects (see figure **??**).

3.1 The M4Interface Class

This class provides for a static method to retrieve an instance of the actual factory object: *M4Interface.getInstance()*. The M4Interface class is the central contact point for retrieving and storing top-level data, such as Cases and Operators. Using for example the method *M4Interface.findCaseForUpdate (String name)* a Case can be retrieved. Dependent objects are accessed through the other objects using the relations defined in M4. Note that the choice for a read-only or read-write session should be made at the Case level. A second functionality available through the M4Interface is direct SQL access.

3.2 The Value Objects

The value objects are defined conforming to the JavaBeans specification, in the sense that all attributes are private, and get/set accessor methods are available. These value objects map closely on the tables as defined in the M4 physical data model. Value object names consist of the object name with a suffix V. For example for the CASE_T table that is part of the M4-Relational Metadata-Schema an object CaseV is defined that has getters and setters for the fields: id, name, mode, output, outputName, population, populationName and validity. Note that all value objects implement the Serializable interface, so as to make them transportable to and from the application server using RMI.

3.3 Class diagram

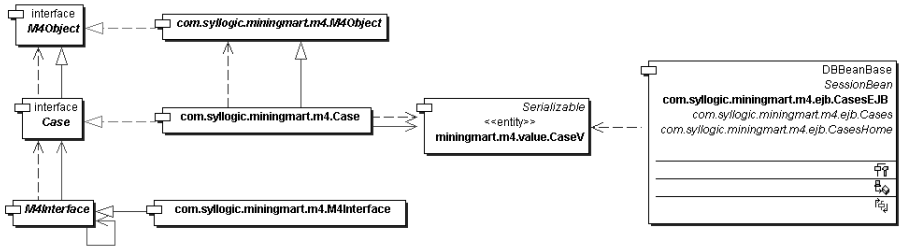

Figure 3.1: Class diagram showing M4Object, M4Interface and Case components in the M4Interface.

Figure 3.1 shows how a few components in the M4Interface are related to each other. It shows that the interface miningmart.m4.Case extends miningmart.m4.M4Object and that these interfaces are implemented by corresponding classes in the com.syllogic.miningmart.m4 package. The miningmart.m4.M4Interface class is an abstract class and is extended by a corresponding class in the com.syllogic.miningmart.m4 package. It also shows that the com.syllogic.miningmart.m4.Case class and the CasesEJB session bean both use the CaseV value object. The implementation of other components like Concept and ColumnSet is equivalent to the implementation of the Case component.

3.4 Level of support for the M4-Relational Metadata Schema

The M4Interface provides an application programming interface that in general supports reading, inserting, updating and deleting records of all tables. Database constraints and triggers guard the data integrity. The current implementation does have a few restrictions however that are important to note.

For creating or retrieving a certain object always a parent object must exist. This parent object provides the context for the child object that is created or sought for. At the highest level we have for example the M4Interface class that must be available in order to create or retrieve a Case. That Case is then in turn needed in order to be able to create other objects like for example Concepts, Chains and Steps. This has as a consequence that when a parent is deleted its children are also lost for the interface, because then they cannot be retrieved anymore from the database.

Further the M4Interface hides the primary keys from the programmers interface, because the ids are not a property of the stored objects; they are there for the database to handle the objects. This means the programmer must identify an object in another way. We have chosen to use the name of the object for this. This means that names must be unique within a certain context. For example a Case name must be globally unique (i.e. in the CASE_T table) and a Step name must be unique within the Case.

The M4-Relational Metadata Schema supports Concepts to be part of more than one Case. The current M4Interface implementation, however, only support Concepts to be part of one Case. In our view this makes the handling of Concepts much easier. Many other objects may be related to a Concept and if the Concept would be part of more Cases these related objects could also be seen as being part of more Cases. Updating one object would mean it would also be updated in other Cases it is part of. The user must be very careful about this. Therefore to simplify things currently it only is allowed for Concepts to be part of one Case. If the Concept is also needed in another Case we suggest to make a copy of it. If it becomes clear that it should be possible for Concepts to be part of more Cases then the M4Interface can be adapted for this.

Finally fields in the schema that are specially used by the Compiler can not be modified using the M4Interface.

Chapter 4

Using the interface

Documentation describing the interface has been produced using the documentation features of Java. The api is available in HTML format and is part of this deliverable. It describes the Objects and methods available in the M4 Interface. When using the M4 interface the api will be the place to start. In the appendix the source is listed of an example client program that uses methods from the m4 Interface. It shows how to invoke the interface, create objects and set object properties. Note that the COL sends and receives value objects to the Session Beans. These value objects are representations of database rows. When setting a property using the COL the corresponding attribute of a value object is set.

It is important to note that the value object is only stored to the database when the store() method is called on the corresponding object. There is an exception to this rule however. When an object is created in relation to another object the object needs to be stored first before this relation can be set. For example when creating a Concept using the createConcept method of the Case class the Concept will automatically be stored by the interface before connecting it to the Case. The example listing in the appendix comments further on this and shows how store() is used. One can use the isChanged() method that is part of the miningmart.m4.M4Object interface to check if an object needs to be stored.

Chapter 5

Installation and configuration with JBoss

The M4Interface has been implemented using the JBoss 3.0.0RC3 application server. Several configuration files are needed to set up the software. Our configuration files have been included with the deliverable. Other parties may need to edit some of the files and set information about their own server name, SID, user name and password. Information which files should be edited and where to place the files is included with the configuration files.

Chapter 6

Current state of the interface

All components of the M4 Interface have been implemented and several tests have been done testing the functionality of different components within the M4 Interface. There are still some important open issues at this time however that should be especially be noted by partners that want to use the current software:

- Most testing effort has been spend on the Session Beans and less on the Client Object Library. More tests on the Client Object Library are still needed however in order to show that all functionality indeed works.
- There is an issue that the Client Object Library caches objects that have been read from the database. Database triggers providing object consistency may however delete records or set columns to NULL. Objects in the COL that correspond to records that are deleted should be destroyed which is not the case yet.
- Case locking still has to be implemented.
- As the object id's have erroneously been implemented as integer objects instead of longs the current sequence of the M4 Relational Schema is not supported. Currently the maximum id nummer supported is 2,147,483,647. The id type will have to be adjusted.
- Direct sql access through the interface is not supported yet.

These issues will be solved with updates of the M4 Interface software within the near future.

Appendix

Programming standards

Copyrights

The source code will be copyrighted by the company where it originated, unless otherwise decided during the course of the MiningMart project. Due to the sensitive nature of copyright protection however, it should be noted that any material produced should be protected by copyright (default will be a copyright owned by the producer of that material) at all times. If it is decided to place material in the public domain, then experience shows that an Open Source style license such as the GPL or LGPL provides adequate protection coupled with full disclosure and availability. Note however that such a license does not preclude a copyright ownership by the MiningMart partners. Perot Systems Nederland will include the following code snippet at the top of all of its code: // Copyright 2002 by Perot Systems Nederland // All rights reserved.

Java Package Name Choices

Java package naming has solved many of the problems faced by integrators of code from different sources. We would however propose to standardize the names of the packages themselves as follows:

- All code which is common for the entire MiningMart system should reside in packages under the global package miningmart. For example, the client object library for the M4 interface will reside in miningmart.m4, with value objects defined in miningmart.m4.value.
- All code which is internal to parts developed by an individual partner should reside in either a package under the MiningMart package mentioned above, or else in a package named according to that partners own standards. For example, the implementation of the object library mentioned above will reside under the package com.syllogic.miningmart.

This approach, when combined with the Factory and Facade design patterns, will allow partners to interface to each others code, without ever hav-

ing to known partner specific package names. In fact, one would never know of the com.syllogic packages unless one browsed the jar files or performed getClass().getName() calls on objects received from the factory methods. This is the same approach also followed by Sun Microsystems for large parts of the standard Java library.

Naming Standards

Names are generally defined as is common for Java code:

- Package names are all lowercase.
- Classes and interfaces start with an uppercase letter, while methods, attributes, and variables start with a lowercase letter. Names are generally chosen to describe value (Classes, interfaces, and attributes) or function (methods) and not abbreviated. If the name is a concatenation of several words, then the start of each word is highlighted by putting the first letter in uppercase.
- Accessor methods are named getXxx() or setXxx(), where Xxx is the capitalized name of the attribute involved.
- Single letter variables are only used in loops:
  - Counters are called n, m,
  - Indexes in arrays are called i, j,
  - Characters (typically retrieved from strings are read from streams) are called c.

  For other uses descriptive names should be used.
- In a few cases suffixes will be used:
  - Enterprise Java Beans will be referred to by the name of the Remote interface. The Home interface will have this name suffixed with Home, and the implementation will have suffix EJB.
  - Value objects will have suffix V.
  - When several methods are needed with the same name and parameter list, but with different return types, a single letter suffix may be used to distinguish between them. For example, if an integer value is sometimes needed in a wrapper object, and sometimes as an int, the Integer returning method will get a suffix O to signify as object.

Version Control Related Standards

Perot Systems uses CVS for its version control. Since CVS relies on RCS code for the actual processing of the files, RCS style markers are used and substituted. We normally use *Id* at the top of the file to provide identification, and *Log* at the bottom to keep track of the change history.

Source Code Layout

Unfortunately layout is a subject which can easily waste a lot of time. When working with code produced by others, the simple but hard rule is to follow the style already in use. For newly crafted code, Professor Andrew Tanenbaum of the Amsterdam Free University once stated (in relation to contributions to the Minix Operating System) that he would not look at any piece of code before it had been reformatted by cb using his standard settings. We do not propose to go that far, but will use the following settings:

- ASCII TAB characters will be presumed to expand to 8 spaces.
- Indentation is 2 spaces.
- If/while/for/etc will always have use braces around their sub-statements.
- The opening brace will preferably be placed at the end of the line preceding the block, while the closing brace comes on a line by itself.
- Every statement is on its own line.

Other Standard Elements

Perot Systems Nederland will use Together 5.5 as its primary design/development tool, Together ControlCenter will allow us to develop EJBs using a simplified interface to the three source files involved, and integrates seamlessly with CVS. Together stores most of its integration information in the source files themselves, using the javadoc style comments. Some of these comments are used to control display of UML diagrams, while others control the links to the J2EE deployment tools. As a result of this, several @¡keyword¿ style lines will be present in the source files, which do not come from the javadoc standard. Some of the more common ones are: @notProperty This signifies that a method looks like an accessor function, but should not imply the presence of a JavaBean property. @ejbHome Specifies the EJB Home interface. @ejbRemote Specifies the EJB Remote interface. @ENV-REF Starts an environment reference. @ENV-TYPE Java type of the environment value. @ENV-VALUE The actual value involved. @RESOURCE-REF Starts a resource reference. @RES-JNDI-NAME JNDI name of the resource.

Example code

This section lists sample java code of a client that uses the M4Interface software to access an instance of the M4 Relational Metadata Schema. It demonstrates how to invoke the M4Interface, create objects, set properties and store the objects. It creates a Case "Dummy Case", creates two Concepts (partner, contract) with each two BaseAttributes. The two Concepts are connected to each other by a partnerRole Relationship. Further one Step is created and partner is set as an input Parameter for the Step.

This code is delivered as part of the M4interface by the miningmart.m4.M4ClientExample class.

```
// Copyright 2002 Perot Systems Nederland
// All rights reserved
//
//$Id$

package miningmart.m4;

import miningmart.m4.M4Interface;
import miningmart.m4.Case;
import miningmart.m4.Concept;
import miningmart.m4.BaseAttribute;
import miningmart.m4.Relationship;
import miningmart.m4.Step;
import miningmart.m4.Parameter;
import miningmart.m4.Operator;
import miningmart.m4.CreateException;
import miningmart.m4.DomainDatatype;

import java.util.Iterator;
import java.util.Collection;
import miningmart.m4.StorageException;
import org.apache.log4j.Logger;
import org.apache.log4j.PropertyConfigurator;

/**
 * Example Class that demonstrates how to invoke the M4Interface
 * and shows the use of some of its methods.
 *
 * It does the following:
 * - list all Cases
 * - list all Operators
 * - create Case  "Dummy Case"
```

```
* - create Concept "Partner"
* - add BaseAttributes "Name" and  "Age"
* - create a second Concept "Contract"
* - add BaseAttributes  "Name" and  "Date"
* - create a Relationship "PartnerRole" between concepts "Partner" and "Contract"
* - create a Step "First Step"
* - create an input Parameter for "First Step"
* - connect the Concept "Partner" to the input Parameter
*/
public class M4ClientExample {

 /** The logger class to log messages */
 private static Logger logger;

 //Added to test functionality of this class
 public static void main(String[] args) {

   //initialize Logger
   initLogger();

   //Get an instance
   M4Interface myInterface = null;
   try {
     //The next line will only be necessary for the dummy
     //implementation.
     //The final implementation will allow to directly call
     //the getInstance() method.
     Class.forName("com.syllogic.miningmart.m4.M4Interface");
     myInterface = M4Interface.getInstance();
   }
   catch (ClassNotFoundException e) {
     logger.error("getM4Instance ", e);
   }

   Collection caseList = myInterface.getAllCaseNames();

   //Print List of Case names
   logger.info("Available cases: ");
   printList(caseList);

   //Retrieve Operators
   Collection operatorList = myInterface.getAllOperatorNames();

   //Print List of operator names
```

```
    logger.info("Available operators: ");
    printList(operatorList);


    //Create a Case
    Case myCase = null;
    try {
      myCase = myInterface.createCase("Dummy Case");
      logger.info("Case created: " + myCase.getName());
    }
    catch (CreateException e){
      logger.error("Exception while creating a Dummy Case ", e);
    }

    //Store a Case
    try {
      if(myCase.isChanged()){
        myCase.store();
        logger.info("Case stored: " + myCase.getName());
      }
    }
    catch (StorageException e ) {
      logger.error("Exception while storing Dummy Case ", e);
    }


    //Create a Concept
    Concept partner = null;
    try {
      partner = myCase.createConcept("Partner",
        miningmart.m4.Concept.TYPE_BASE);
      //In order to be able to connect the Concept to the Case
      //the Concept is stored automatically by the M4 Interface,
      //so we need not do a partner.store() here. As a good
      //habit doing a store() here would be fine.
      partner.store();

      logger.info("Concept created and stored: "
        + partner.getName());

      //Add attributes
      //BaseAttributes are also stored directly when created.
BaseAttribute partnerName = partner.createBaseAttribute(
  "Name", DomainDatatype.NOMINAL);
```

```
      partnerName.store();
      logger.info("Attribute added: " + partnerName.getName());

BaseAttribute partnerAge = partner.createBaseAttribute(
  "Age", DomainDatatype.ORDINAL_SCALAR);
      partnerAge.store();
      logger.info("Attribute added: " + partnerAge.getName());
    }
    catch (CreateException e){
      logger.error( "Failed to create the partner concept or
        its attributes", e);
    }
    catch (StorageException e){
      logger.error( "Failed to store the partner concept or
        its attributes", e);
    }

    //Create a second Concept + attributes
    Concept contract = null;
    try {
      contract = myCase.createConcept("Contract"
        ,miningmart.m4.Concept.TYPE_BASE);
      contract.store();
      logger.info("Concept created: " + contract.getName());

      //Add attributes
BaseAttribute contractName = contract.createBaseAttribute(
  "Name", DomainDatatype.NOMINAL);
      contractName.store();
      logger.info("Attribute added: " + contractName.getName());

BaseAttribute contractDate = contract.createBaseAttribute(
  "Date", DomainDatatype.ORDINAL_SCALAR_TIME);
      contractDate.store();
      logger.info("Attribute added: " + contractDate.getName());
    }
    catch (CreateException e){
      logger.error( "Failed to create the contract concept or
        its attributes", e);
    }
    catch (StorageException e){
      logger.error( "Failed to store the contract concept or its
        attributes", e);
    }
```

```
  //Create a Relationship
  Relationship partnerRole = null;
  try {
    partnerRole = partner.createFromConceptRelationship(
      "partnerRole", contract);
    partnerRole.store();
    logger.info("Relationship created: " + partnerRole.getName());
  }
  catch (CreateException e){
    logger.error( "Failed to create retalionship partnerRole ", e);
  }
  catch (StorageException e){
    logger.error( "Failed to store retalionship partnerRole ", e);
  }


  //Create a new Step in the Case
  Step aStep = null;
  try {
    aStep = myCase.createStep("First step");
    logger.info("Step created: " + aStep.getName());

    aStep.setOperator(myInterface.findOperator("RowSelection"));
    aStep.store();
    logger.info("Operator added to Step: "
      + aStep.getOperator().getName());

    Parameter inputParameter = aStep.createParameter(
      "inputParameter");
    inputParameter.setParameterNr(1);
    inputParameter.setParameterType("IN");
    inputParameter.setParameterObject(partner);
    inputParameter.setOperator(aStep.getOperator());
    inputParameter.store();
logger.info("Concept partner set as input parameter for
  the step");
  }
  catch (CreateException e){
    logger.error( "Failed to create a Step or set up
      an input parameter ", e);
  }
```

```
    catch (StorageException e){
      logger.error( "Failed to store a Step or its
        input parameter ", e);
    }

  }


  private static void printList(Collection list) {
    Iterator i = list.iterator();
    while (i.hasNext()) {
      String aName = (String)i.next();
      System.out.println(aName);
    }
    System.out.println();

  }

  private static void initLogger(){
    PropertyConfigurator.configure("log4j.properties");
logger = Logger.getLogger("M4Test");
  }

}

/*
*$Log$
*/
```