IBM Operational Decision Manager

Enabling the Rule Engine to Augment the Decision Data

Michael Johnson

Information Management

Executive summary

This IBM® Redpaper™ publication describes IBM Operational Decision Manager (ODM). ODM is a set of tooling and interfaces that provides an architecture that can encapsulate your decision logic away from program code into rules, which are rendered as natural language. Decisions are now more agile, adaptable to change, and visible to those who need them.

This paper supplements the following IBM Redbooks® publications:

- *Flexible Decision Management with Business Rules on IBM z Systems*, SG24-8014

  `http://www.redbooks.ibm.com/abstracts/sg248014.html`

- *Systems of Insight for Digital Transformation: Using IBM Operational Decision Manager Advanced and Predictive Analytics*, SG24-8293

  `http://www.redbooks.ibm.com/abstracts/SG248293.html`

This paper describes a means to use additional data with your rules.

The challenges

Businesses are faced with many challenges in this increasingly competitive environment. Operational decisions can be challenging to the business, but ODM can help you meet the following concerns:

- Decisions are not consistent across the organization. This can be due to a number of factors. Those of disparate systems, personalities and skills within and outside of the business. The different channels through which the business operates.
- Decisions cannot be made at the correct moment and on time. Many factors can affect this, but particularly decisions cannot be made if there are too many manual tasks and too much data to handle to make decisions.
- It is hard to keep decisions up to date. Business is asking for more agility than the delivery team can support.
- It is hard to understand how decisions are made. Business logic is in people's minds or application code.
- Lack of decisions ownership for business stakeholders. Development skills are required to update automated decisions.

The business challenge

Organizations must find solutions to the following sorts of questions:

- How can we cost-effectively support an expanding customer base, meet regulatory requirements, and deter fraud while maintaining an always available and secure environment?
- How can we reduce cost while improving information accuracy, security and data governance?
- How can we provide fast, accurate analysis to set the correct premiums and improve access to claims data across multiple international locations?
- How can we meet the need for 24 x 7 access to analytics for key services, such as those of courts, hospitals, police, fire services, and financial and insurance markets?

The technical challenge

Customers have had some of the following challenges:

- Unable to quickly extract actionable insights from big data and identify market opportunities to adapt or expand the offering to meet customer demand.
- Need to support more data online, identify fraud before it happens, better understand customer behavior and improve response times.
- Need to increase system availability, optimize workloads, speed queries, and accelerate the generation of claims reports.
- Needing a highly scalable, available platform to provide insight into government for 3 million citizens across the country.

Overview of IBM Operational Decision Manager

The following sections describe an ODM solution that supplements decision runtime data with additional data during the decision process.

This paper investigates how to supplement the runtime data passed to a rule decision with additional data sourced from an external location, such as a database. The platform being used for this is IBM z Systems™, which for the scenario in this paper is where the additional data exists on this system of record.

What is ODM?

Operational Decision Manager is a set of tooling and interfaces, which provide a means to encapsulate decision logic away from program code into rules. These rules are rendered as natural language. When decisions are exposed in this format, they become more agile, adaptable to change, and visible to those who need them. By having a rich set of tooling, the change cycle of a rule and its deployment is much less than the traditional code development and deployment route.

Operational Decision Manager gives you a vocabulary to write business rules. The vocabulary comes from the business object model (BOM) that is created from the execution object model (XOM). The XOM is the runtime model against which rules are run. It references the application objects and data, and is the base implementation of the BOM.

The XOM can be from any of the following sources:

- Compiled Java classes (Java execution object model)
- Extensible Markup Language (XML) Schema (dynamic execution object model)
- Common Business Oriented Language (COBOL) and Programming Language One (PL/I) headers

These are imported to the ODM tooling to create the BOM. Rules are created from the vocabulary of the BOM and then bundled together as a *Decision Service*. The Decision Service is deployed to a *rule engine*. The rule engine makes decisions for the application based on real-time data. This real-time data is passed as input parameters to the rule engine.

Using ODM allows the decisions to be separated from the client code. When the decisions need to be changed due to business requirements, it is only these decisions and not the application that need to be changed. New and changed rules are redeployed to the rule engines without having to change the application code.

To supplement the data that will be passed to a rule engine there are a number of options:

- Obtain this information before passing all the information to ODM.
- An alternative to this is to have the rule engine obtain this information during the rule processing. Obtaining the information before passing it to ODM is the preferred approach, but there might be times when this information can only be obtained during rule processing. If this is the case, and the additional information is held in a database, a connection to the database must be established, and then obtain the information from the database using a query.

Considerations

If the data needed to make a decision is not available to the client application, and this data is in a database on IBM z Systems, it is probable that the decisions should be made as close as possible to the data. This would mean running the decision service on the rules engine running on z Systems. ODM has a stand-alone rule engine that runs on IBM z, known as the zRule Execution Server (*zRES*) for IBM z/OS® environments. The rule engine, which runs inside a zRES, is a Java Platform Standard Edition (Java SE) solution.

Ideally, if data were to be read over and over from the database it would be good to have a pool of database connections that could be reused, as opposed to establishing a connection each time a query was made. Establishing a Java Database Connectivity (JDBC) connection is more costly than using an existing idle connection. zRES is not on an application server, so it doesn’t have a pool to hold connections to the database. Therefore, we need to create a connection pool that decisions could use when accessing the data in the database.

There are many patterns out there, including the following one, illustrating how this could be achieved using a singleton pattern:

http://theopentutorials.com/tutorials/java/jdbc/jdbc-examples-introduction/#Singleton_design_pattern

The solution

The proposal here is to have a pool of connections that have access to the database. For ODM to be able to use the data from the database, a model of the data will be needed.

Code to represent the data from the database to extend the XOM, and the additional classes needed to implement the connection to the database, are required.

Implementation

The proposed implementation has the following components for the connection pool and database access:

- ConnectionFactoryCreator
  - This is a singleton pattern to hold a connection pool.
  - It is implemented by Apache Commons Database Connection Pools (DBCP).
  - The database credentials are provided in this class.
- The data access object (DAO)
  - Obtains a connection from the pool of connections.
  - Creates a statement from the connection.
  - Runs the Structured Query Language (SQL) query on the statement.
- The transfer object (TO)

  Holds the data that has been returned from the Database.
- DbUtil

  Helper class for the database.

Business object models

For rules to access and use the TO, a new BOM is created for the DAO and the TO. Next, a new BOM to XOM method is created that will be used by the rules to access the data in the database.

The design of this BOM to XOM method completes the following actions:

1. Create a new DAO using the previous classes.
2. Call the method on the DAO to populate a TO.

This new method is verbalized so that the rules can call it. The TO is also verbalized so that the values returned in the TO can then be used.

Rules can now use this additional information.

How to give the rules access to data in the database

From the Rules designer tooling, import the classes for the database access and give the rule project access to them. Create a BOM to XOM mapping function, which asks the DatabaseUtilities implementation for access to the database and retrieve the data from the database. This BOM to XOM method is verbalized so that it can be used in a rule and return a value that the rules can act on based on its value.

The structure of the data that will be returned from the database is modeled as a BOM, and the individual fields verbalized.

How to use the sample in this Redpaper publication

The example in this Redpaper publication is built on top of the z Systems MiniLoan sample.

The MiniLoanDemo can be found in the following location:

```
/usr/lpp/zDM/V8R7M0/zexecutionserver/samples/ruleProjects
```

The product sample enables client applications written in COBOL and PL/I to make rule calls that use the same ODM rules.

You can import the projects as supplied in this paper, or build the projects following the information given.

We advise importing the full example by importing the `.zip` file available with this paper and select the five projects contained in it:

1. Go to the Rule Designer eclipse perspective.
2. Select **File** → **import** → **existing projects into Workspace**.
3. Point at the `.zip` file supplied with this paper and select all of the displayed projects.
4. You need to change the path to the `commons-xxxx-<version>.jar` Java Archive (JAR) files on your system, as shown in substep b on page 6.
5. After this has been done, you should see a workspace that looks like that shown in Figure 1.

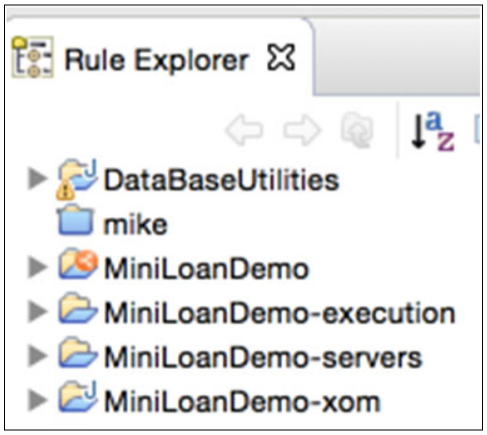


Figure 1 Sample project

> **Note:** DataBaseUtilities will display a warning.

Alternatively, you can build the project up from the product sample and the example in this Redpaper publication by completing the following steps.

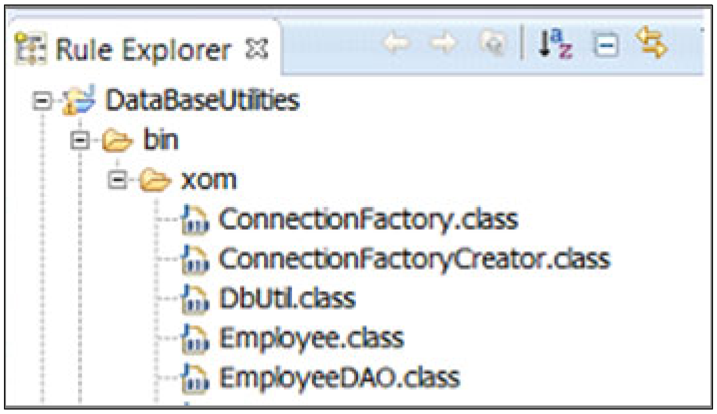
To use the product sample with this example extension, complete the following steps:

1. If you want to build the project step by step, the product sample can be found from the `ODM zexecutionserver/samples` directory on the z Systems server:

   `MiniLoanDemo-Workspace.zip`

2. Use File Transfer Protocol (FTP) to move this file to the location where the tooling is running.
3. From Rule Designer Eclipse perspective, select **File** → **import** → **existing projects into Workspace**. Import the MiniLoan sample.

4. Next, you need to import the example project for this Redpaper publication. Import only the `DatabaseUtilitiesproject` from `mikesredpaperDB2pool.zip`, as shown in Figure 2.



```
DataBaseUtilities
  bin
    xom
      ConnectionFactory.class
      ConnectionFactoryCreator.class
      DbUtil.class
      Employee.class
      EmployeeDAO.class
```

Figure 2 The DataBaseUtilities folder

5. Next, under the Properties for the DatabaseUtilities, under the Java Build Path in Libraries, point to the commons JAR files:

   a. Under Projects, select the `DataBaseUtilities` folder. Right-click and select **Properties** → **Java build path**.

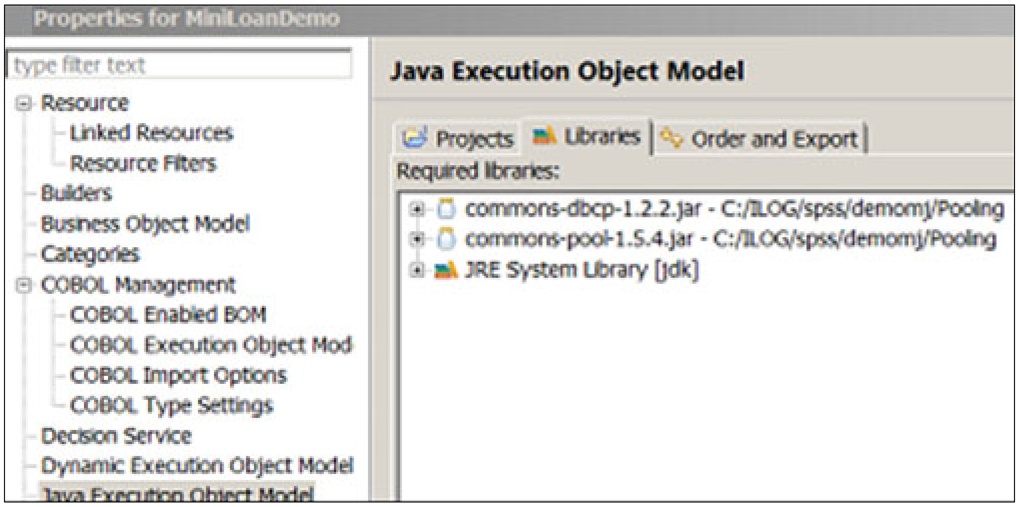
   b. Under Libraries, add external Java archive (JAR) files:

      - `Commons-dbcp-<version>.jar`
      - `Commons-pool-<version>.jar`

Tip: These JAR files are found with the product at the following location:

`usr/lpp/zDM/V8R7M0/zexecutionserver/resconsole/lib`

Figure 3 shows the Java Execution Object Model properties.


Properties for MiniLoanDemo

type filter text

- Resource
  - Linked Resources
  - Resource Filters
- Builders
- Business Object Model
- Categories
- COBOL Management
  - COBOL Enabled BOM
  - COBOL Execution Object Mod
  - COBOL Import Options
  - COBOL Type Settings
- Decision Service
- Dynamic Execution Object Model
- Java Execution Object Model

Java Execution Object Model

Projects | Libraries | Order and Export

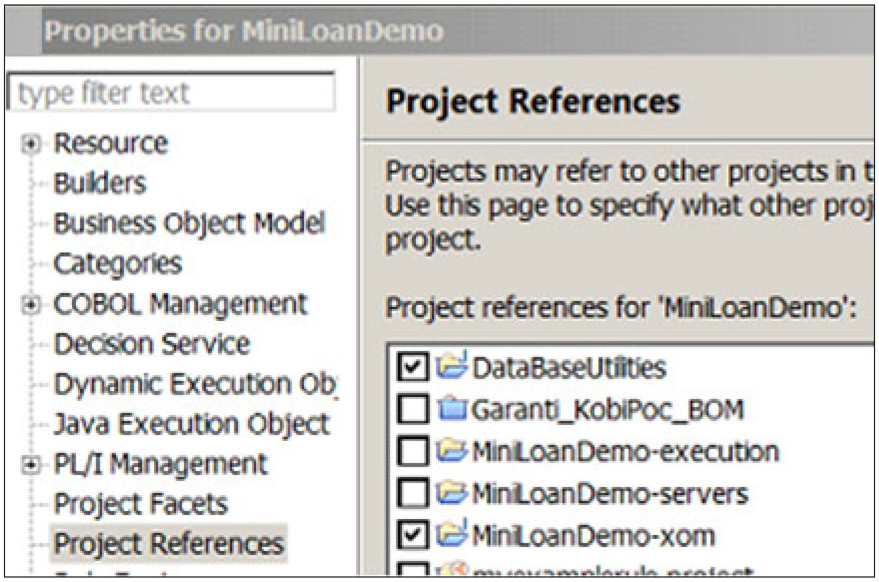
Required libraries:

- commons-dbcp-1.2.2.jar - C:/ILOG/spss/demomj/Pooling
- commons-pool-1.5.4.jar - C:/ILOG/spss/demomj/Pooling
- JRE System Library [jdk]

Figure 3 Java Execution Object Model properties

6. Next, associate this DatabaseUtilities project with the MiniLoanDemo project.

   Select the project references from the properties of the project and select the check boxes next to the project, as shown in Figure 4.



Properties for MiniLoanDemo

Project References

Projects may refer to other projects in t
Use this page to specify what other proj
project.

Project references for 'MiniLoanDemo':

- [✓] DataBaseUtilities
- [ ] Garanti_KobiPoc_BOM
- [ ] MiniLoanDemo-execution
- [ ] MiniLoanDemo-servers
- [✓] MiniLoanDemo-xom

Figure 4   Properties of the MiniLoanDemo project

7. Create an additional BOM called `DatabaseModel` and import the Employee class (TO) and the Employee DAO by selecting **New BOM entry from XOM**.
8. Browse the `xom` under `DatabaseModel` and select the two classes, as shown in Figure 5.

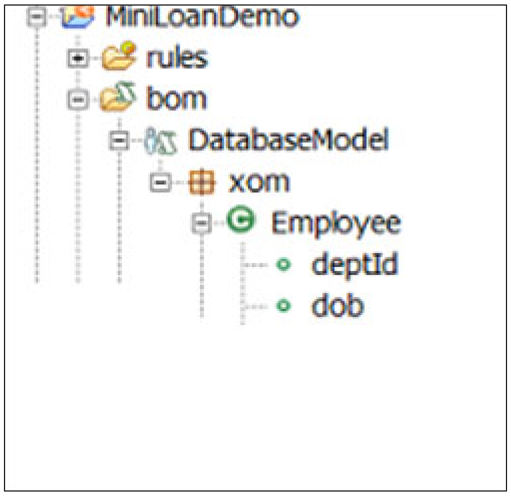
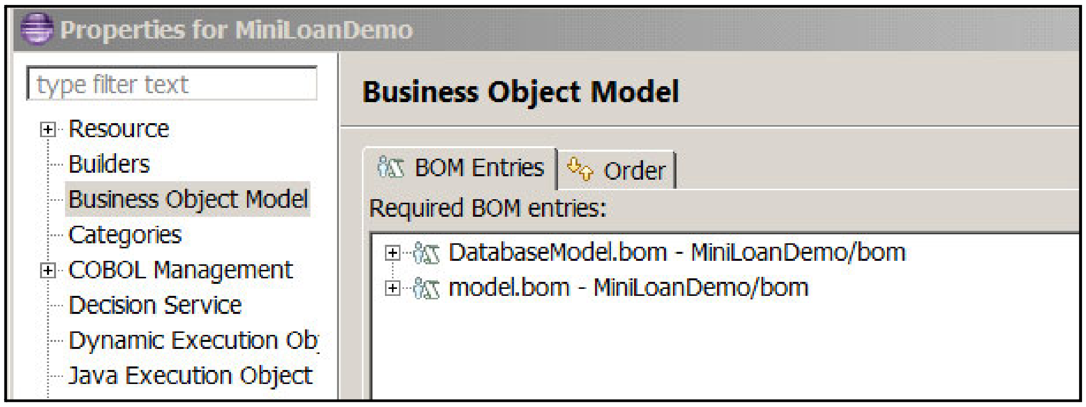


Figure 5   The BOM structure

9. Under **Properties** → **Business Object Model**, if the BOM is not already displayed, add this new Database Model BOM, as shown in Figure 6.



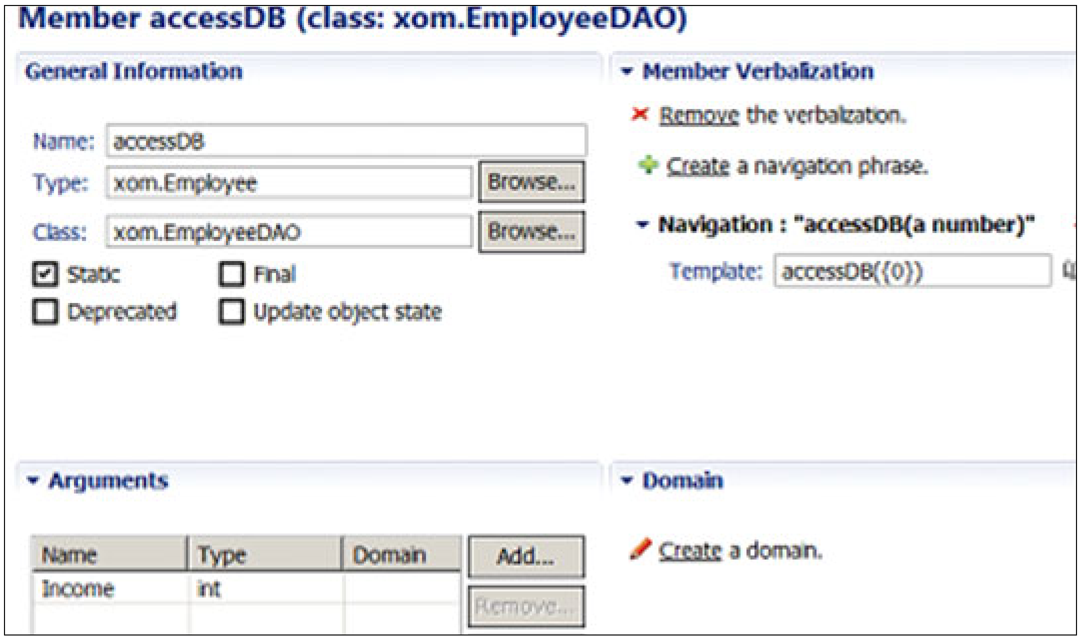
Required BOM entries:

DatabaseModel.bom - MiniLoanDemo/bom
model.bom - MiniLoanDemo/bom

Figure 6 BOMs associated with the project

10. Create a new BOM to XOM method under EmployeeDAO called `accessDB` (int), as shown in Figure 7. This method takes an integer, which represents the database record to retrieve, and returns an Employee record:
    a. Select the class in the BOM to add a new member (for example, **Class** `xom.EmployeeDAO`).
    b. Scroll down to the Members section.
    c. Select **New**.
    d. Give the new method a **Name** and a **Type**, such as `xom.Employee`.

    Figure 7 shows the member information.



Name: accessDB
Type: xom.Employee
Class: xom.EmployeeDAO
[✓] Static
[ ] Final
[ ] Deprecated
[ ] Update object state

Navigation : "accessDB(a number)"
Template: accessDB({0})

Name	Type	Domain
Income	int	

Figure 7 Creating the accessDB BOM to XOM method

The previous figure shows the input parameters and the verbalization of the method.

11. Select the new method and move down to the BOM to XOM Mapping.

12. Paste the following code snippet in the `Body` section, as shown in Example 1.

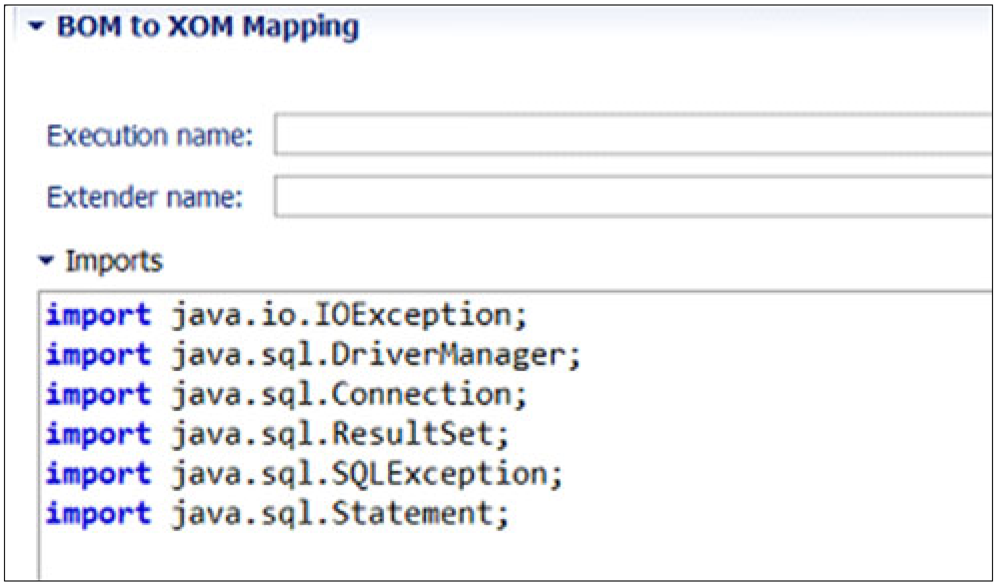
Example 1   The BOM to XOM method body

```
xom.Employee employee = null;
try {
IntemployeeId = Income;
       // create a DataAccessObject
xom.EmployeeDAOempDao = newxom.EmployeeDAO();

System.out.println("accessDB");
System.out.println("created 1 DAOs");
// call the DAO method to make the connection to the DB
// the DAO asks the ConnectionFactory for this
// runs the SQL call
// returns a built employee object from the returned SQL data
employee = empDao.getEmployee(employeeId);

if(employee != null) {
System.out.println("b2x Employee Name:" + employee.getEmpName());
System.out.println("");

       }
else {
System.out.println("No Employee with Id: " + employeeId);

       }
     } catch (NumberFormatException e) {
e.printStackTrace();
     } catch (IOException e) {
e.printStackTrace();
     } catch (SQLException e) {
e.printStackTrace();
     }
return employee;
```

13. Add the imports that this method requires, as shown in Figure 8:

   a. Move down to the BOM to XOM Mapping section of the new method.
   b. Open the imports section and paste the required imports:

      - `importjava.io.IOException;`
      - `importjava.sql.DriverManager;`
      - `importjava.sql.Connection;`
      - `importjava.sql.ResultSet;`
      - `importjava.sql.SQLException;`
      - `importjava.sql.Statement;`



```
import java.io.IOException;
import java.sql.DriverManager;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
```

Figure 8 The imports for the code

Writing the rules

With this implementation, the verbalization enables the following information to be written. One of the fields of the Employee object has been verbalized as `the dept id`, as shown in Example 2.

Example 2 Code to return a field

```
add "accessing db" to the messages of 'the loan' ;
   add " the dept id = " + the dept id of accessDB ( 1  ) to the messages of 'the
loan' ;
```

In the rule, the value of `the dept id` field of the Employee object is returned. Behind this rule, a query against the database is made for the record where the dept id is `1`. A side effect of this rule is that for each field of the Employee a query is made against the database.

To make only one query to the database, the following code could be used at the start of the rule, as shown in Example 3.

Example 3 Rule to return the second entry in the database

```
definitions
   set myEmployee to accessDB ( 2 );
```

This gets the second entry in the database. This shows a specific value, in this case 2, but we could easily replace it with a variable, such as `empId` for example. Therefore, in the body of the following rule (the `MinimumAmount` rule), it is possible to reference each field of the Employee record that has been returned from the database, as shown in Example 4.

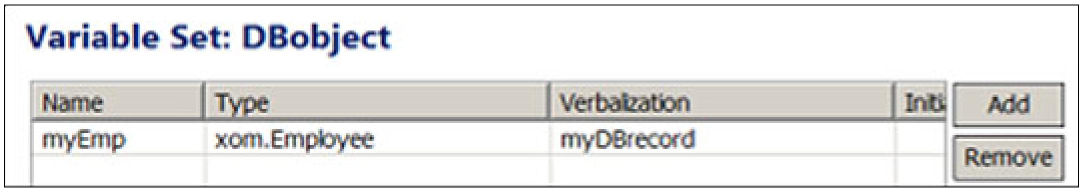
Example 4   Adding myEmployee fields to messages

```
definitions
setmyEmployeetoaccessDB ( 2 );
if
    the yearly repayment of 'the loan' is more than the yearly income of
'the borrower' * 0.3
then
    add "Too big Debt-To-Income ratio" to the messages of 'the loan' ;
    reject 'the loan';

    add " name = "    + the emp name of myEmployee to the messages of 'the loan' ;
    add " dep id  = " + the dept id of myEmployee  to the messages of 'the loan' ;
    add " Salary = "  + the salary of  myEmployee  to the messages of 'the loan' ;
```

It might be possible depending on the scenario to populate the object with data from the database before entering a rule package. In the Ruleflow editor of Rule Designer make an access to the database to populate the object. This object is then available for use by all of the rules.

To achieve this, complete the following steps:

1. Create a *Variable Set*, give it a name, and then add a variable to hold the information from the database. An example named `DBobject` is shown in Figure 9.

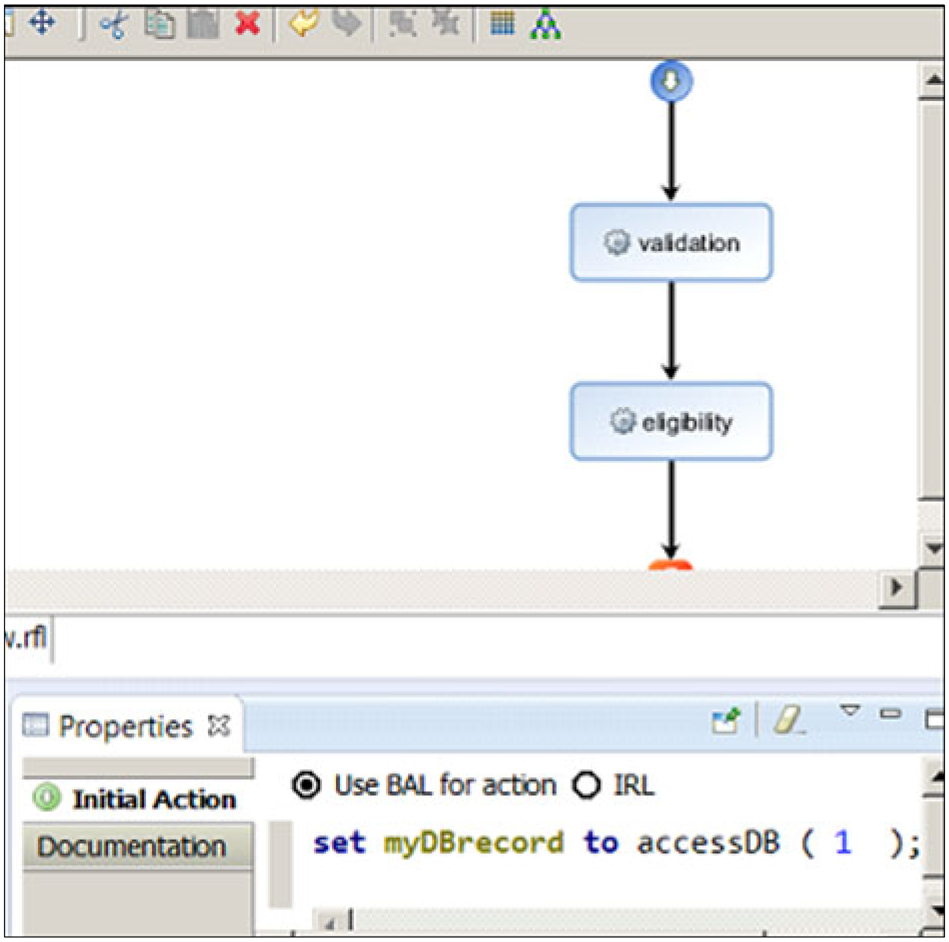


Variable Set: DBobject

Name	Type	Verbalization	Initi
myEmp	xom.Employee	myDBrecord	

Figure 9   Creating a Variable Set

2. In the Ruleflow, create an Initial Action. Write some Business Action Language (BAL) to access the database and populate the object with this data, as shown in Figure 10.



Use BAL for action [✓]

```
set myDBrecord to accessDB ( 1  );
```

Figure 10 Rule flow for the project

In this example, the record that has an ID of `1` is returned from the database, and this data is held in the `myDBrecord` object. The data can now be accessed by the rule in the following way, shown in Example 5.

Example 5 Accessing the salary field of myDBrecord

```
add " the salary = "    + the salary of myDBrecord to the messages of 'the loan'
```

Deployment

When the artifacts are deployed from the Rule Designer tooling, a package of artifacts is sent to the runtime location. When viewed from the Rule Execution Server (RES) console, the Rule run time has all the pieces to evaluate the rule questions from the data provided.

The deployed artifacts include the following components:

- The MiniLoan execution object model

  Rules use this at run time to access the Borrow and Loan objects.

- The DatabaseUtilities

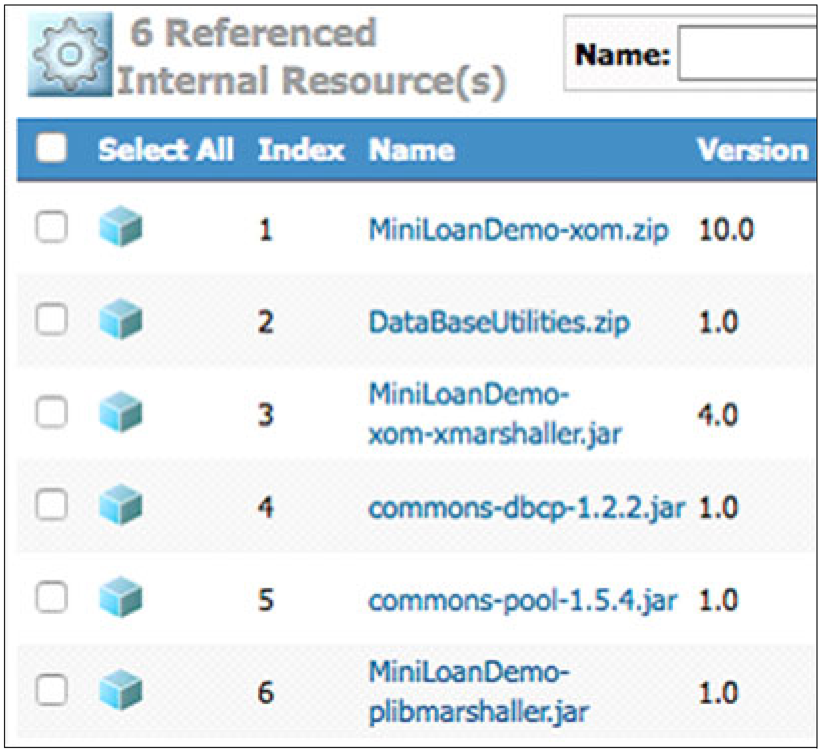
  Implements the pool of database connections and the code that runs queries against the database.

- The marshaller code

  Maps the input COBOL and or PL/I variables of Borrower and Loan so that they can be used by the Rule run time.

- The Apache package, which implements the pool of connections

Figure 11 shows the list of resources after a deployment, as seen on the ODM RES management console.



Select All	Index	Name	Version
	1	MiniLoanDemo-xom.zip	10.0
	2	DataBaseUtilities.zip	1.0
	3	MiniLoanDemo-xom-xmarshaller.jar	4.0
	4	commons-dbcp-1.2.2.jar	1.0
	5	commons-pool-1.5.4.jar	1.0
	6	MiniLoanDemo-plibmarshaller.jar	1.0

Figure 11 Resources after deployment

The database

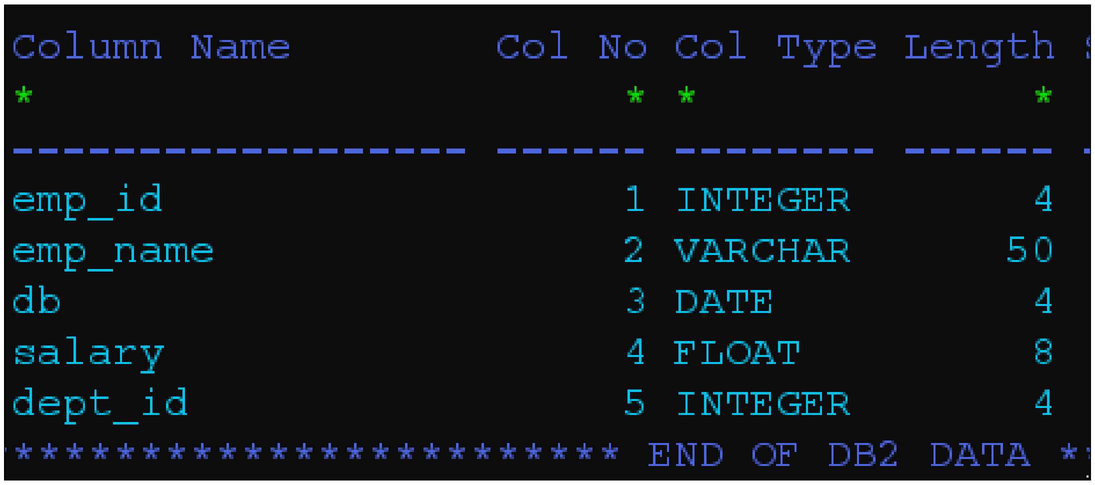
This sample relies on having data stored in a database. Example 6 shows the query that is run against the database.

The database table is defined to have a table called `employee` that has a column called `emp_id`.

Example 6 Query to return emp_id

```
SQL query:   SELECT * FROM employee WHERE "emp_id"  = <a number>
```

Figure 12 shows the columns of the database table. To use this example, the following table needs to be defined with these rows and populated with data.



```
Column Name         Col No Col Type Length 
*                        * *             *
------------------ ------ -------- ------ 
emp_id                   1 INTEGER       4
emp_name                 2 VARCHAR      50
db                       3 DATE          4
salary                   4 FLOAT         8
dept_id                  5 INTEGER       4
************************ END OF DB2 DATA *
```

Figure 12   Columns of the table

In summary

This paper describes a technique to obtain decision data after a call to the rule engine has been made. The solution describes obtaining this additional data from a database, but the data could easily be from another source. The paper demonstrates how this might be done with a runtime on z/OS. ODM runtimes run on different machines and architectures, so you can choose the solution that keeps the access to the data as close as physically possible to the data.

Note that the sample given is an example of the technique. The paper does not consider certain error situations. Stronger error checking should be considered, such as what should the behavior be when there are database connection errors, and when a Null employee record is returned to the rules.

The paper also shows ways in which the data can be used by Rules using Variable Sets and Definitions, and accessed in the `then` section of a rule. This is a very useful technique when circumstances require additional data to be obtained after the decision data has been passed to the rule engine. The technique can be extended to call IBM DB2® stored procedures, and to do analytical scoring of the incoming data.

Systems cited in the example

The following sections describe the systems cited in this example.

z Systems

The z Systems servers are systems of record, which hold the most important data of the business. It is critical that this data is secure, accurate, and maintained with governance. Ideally, making decisions and scoring this data needs to be done as close to the data as one can.

There are a number of advantages to using z Systems:

- No data movement
- Best of class security
- Built-in auditing and governance
- Leverage existing high availability and disaster recovery (HA/DR) capabilities
- Currency of data
- Reduce complexity
- Eliminate data duplication
- Improve synchronization
- Bring analytic functions to the data
- Rules are used in the batch processing
- Decisions made from the preferred application environment

Why IBM z13

The choice to use IBM z13™ provides the following advantages:

- Larger cache = in-memory analytics for faster insight
- IBM Mathematical Acceleration Subsystem (MASS) libraries = 2 - 10 times improvement porting x86 analytic workloads
- Single instruction, multiple data (SIMD) improvements = 80% increase in IBM ILOG® CPLEX® Optimization Studio (CPLEX) on IBM z/OS modeling
- Processor (CPU) enhancements = no performance effect on real-time scoring in transactions

Appendix

This appendix provides information about the following topics:

- ► Extending the XOM
- ► Adding classes to the XOM
- ► The data access object
- ► Class implementations: ConnectionFactory
- ► Using a pool manager for handling connections with DBCP 1.2.2 and pool 1.5.4

Extending the XOM

In the rule designer, we needed to extend the sample MiniLoan XOM to use additional classes, which make the connection to the database.

Adding classes to the XOM

The XOM classes that make the connection to the database use an implementation that creates a pool of connections to the database. These connections are used in the implementation to give access to the database. Data from ODM is passed through in an SQL statement. The database connection then gets returned to the pool of connections ready for the next ODM rule invocation.

To import these into the Rule-designed project, select **File** → **import** → **General** → **File System (from directory…)** and choose the `XOM` folder.

A compressed file containing the implementation will be supplied with this document.

For the new source files to resolve their externals, add the `pool` and DBCP JAR files to the Project. This is done by completing the following steps:

1. From the Java perspective of the tooling, click **Properties** on the project.
2. In `Java Build Path`, and to the Libraries tab using the **Add external Jars** button, add the two `commons` JAR files:

   ```
   commons-dbcp-1.2.2.jar
   commons-pool-1.5.4.jar
   ```

   These can be found under the `zexecutionserver/resconsole/lib` directory.
3. Next, we have to get the BOM to recognize the new classes in the XOM. We had to create a new BOM class with the name for the class. Select it by right-clicking and then choosing **BOM Update**.

   The previous example adds the additional XOM classes to an existing XOM.

The data access object

In our example, the DAO is the `EmployeeDAO` class. This class is where the SQL statement to run is defined. The SQL statement derived from the IBM SPSS® model is placed here. The first thing the method in this class does is to ask for a connection to the database. Either a new connection is created or an idle connection from the pool is returned. Next, the IBM DB2 statement to run is created. You then perform the following steps:

1. Run the statement.

2. If you get a result set back, you build and return an employee object built from the result of the SQL query, as shown in Example 7.

Example 7 Building and returning an employee object

```
public class EmployeeDAO {
    private Connection connection;

    private Statement statement;

    public EmployeeDAO() { }

    public Employee getEmployee(int employeeId) throws SQLException {
        String query = "SELECT * FROM employee WHERE \"emp_id\"=" + employeeId;

        System.out.println (query);

        ResultSet rs = null;
        Employee employee = null;
        try {
            connection = 
ConnectionFactoryCreator.getInstance().getPoolDataSource().getConnection();

            statement = connection.createStatement();

   rs = statement.executeQuery(query);
    if (rs.next()) {

employee = new Employee();
   employee.setEmpId(rs.getInt("emp_id"));
   employee.setEmpName(rs.getString("emp_name"));
employee.setDob(rs.getDate("db"));
      employee.setSalary(rs.getDouble("salary"));
employee.setDeptId((rs.getInt("dept_id")));
}
        } finally {
            DbUtil.close(rs);
            DbUtil.close(statement);
            DbUtil.close(connection);
        }

        // Display some pool statistics
printDriverStats();
        return employee;
   }
}
```

Class implementations: ConnectionFactory

The class implementations are shown in Example 8.

Example 8 Class implementations

```
package com.mikes.jdbc.db;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConnectionFactory {
    //static reference to itself
    private static ConnectionFactory instance = new ConnectionFactory();
    public static final String URL = "jdbc:db2://localhost:41100/DSNV11GP";
    public static final String USER = "ZILOGDB";
    public static final String PASSWORD = "xxxxxxxx";
    public static final String DRIVER_CLASS = "com.ibm.db2.jcc.DB2Driver";

    //private constructor
    private ConnectionFactory() {
        try {
            Class.forName(DRIVER_CLASS);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }

    private Connection createConnection() {
        Connection connection = null;
        try {
            connection = DriverManager.getConnection(URL, USER, PASSWORD);
        } catch (SQLException e) {
            System.out.println("ERROR: Unable to Connect to Database.");
        }
        return connection;
    }

    public static Connection getConnection() {
        return instance.createConnection();
    }
}
```

Using a pool manager for handling connections with DBCP 1.2.2 and pool 1.5.4

The following list describes design considerations:

- Extended `ConnectionFactory` to `ConnectionFactoryCreator`:
  - This uses DBCP and a pool implementation to maintain a pool of connections.
  - These connections get returned to the pool when released, rather than losing the connection completely.
  - This class is a singleton that loads the database driver and establishes the pool.
- <*Something*>DAO, the data access object, is the class that asks for connections and then runs the query to the database.

The overall design is to have one `ConnectionFactoryCreator` object that creates multiple connections when there are no free connections in the pool. If there is a free connection, this connection is allocated to the next activity.

The DAO does all of the requesting, and a **`Transfer Object TO`** is used to request actions from the DAO.

About the author

Michael Johnson is a developer and gives presentations about Operation Decision Manager on z/OS. He works out of the Hursley IBM lab in the UK. He has spent his career in the messaging and brokering environments across platforms. He has a Combined honors degree in Computing and Statistics.

Now you can become a published author, too

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time. Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run 2 - 6 weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us.

We want our papers to be as helpful as possible. Send us your comments about this paper or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form:

  ibm.com/redbooks

- Send your comments in an email:

  redbooks@us.ibm.com

- Mail your comments:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

Notices

Trademarks

REDP-5333-00

ISBN 0738455059

Printed in U.S.A.