Java Management Programmer's Guide
Draft Version

spec release

Sun Microsystems Computer Company
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
415 960-1300 FAX 415 969-9131

Part No.: 802-6616-01
Revision C, August 1996

Please
Recycle

Contents

Figures

Tables

Preface

The *JavaManagement Programmer's Guide* documents the application programmers interface for developing Java-based system's management applications in Sun's Solstice JavaManagement API environment.

Note – The specification release version of the JavaManagement API documentation is intended to reflect work in progress by the SunSoft development team. Feedback about the usefulness of the product will be sought from the groups selected to provide evaluation. This feedback should be sent to:

```
jim-interest@rmtc.central.sun.com
```

How This Book Is Organized

This manual has four chapters, as follows:

Chapter 1, "Introduction," provides a high level overview of the JavaManagement API objects.

Chapter 2, "Managed Objects," provides an example that shows how to build a managed object.

Chapter 3, "Event Management," discusses the event management service (EMS), which provide the basic foundation that allow you to build complex event management.

Chapter 4, "Build and Execute Environment," gives instructions for building and executing JMAPI applications.

Related Books

Documentation related to the JavaManagement API includes:

- *JavaManagement API Architecture*
- *JavaManagement API User Interface Style Guide*
- *JavaManagement API User Interface Visual Design Style Guide*
- *JavaManagement API Help Developer's Guide*

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with on-screen computer output	`machine_name% su` `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename.*
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide.* These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-2 Shell Prompts

Shell	**Prompt**
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	#

JavaManagement API Overview 1

The JavaManagement API (JMAPI) is a collection of Java language classes and interfaces that allow developers to more easily build system, network, and service management applications that solve management problems.

The JavaManagement APIs are a standard extension to the Java programming language.

JavaManagement Model

JMAPI builds on existing technologies that minimizes the amount of infrastructure that must be implemented. This strategy also allows us to focus on solving customer's distributed system management problem.

As mentioned above, it is crucial that the architecture scales to a number of different environments. It does this in two ways. First, the component that allows a machine to be managed is small. Agent objects for management operations are securely downloaded and executed.This minimizes the distribution and versioning problem for management operations and easily allows for modification and extension of the management operations. Second, he Java Virtual Machine resides on the key platforms we need to manage. At the highest-level, the architecture consists of the following components.

- **Browser User Interface**
  The Bowser User Interface is the mechanism from which an administrator issues management operations. These operations may be invoked either

interactively through a web browser or standalone application. A standalone application can have either a graphical or command-line user interface style.

- **Admin Runtime Module**
  The Admin Runtime Module is the mechanism that provides active instantiated management objects to applications. It includes the Agent Object Interfaces, Notification Interfaces, and Managed Data Interfaces. Within the JDBC compliant Managed Data Interfaces, security and data access provisions exist to ensure data security. For example, the database can perform data authorization and authentication.

- **Appliances**
  There are simply the networked devices to be managed. The strategy is to push management close to managed devices with dynamic downloading of agents. A Java Terminal can be thought of as an Appliance as well as a DNS server. Though these machines perform radically different functions, they are managed through the Admin Runtime Module and are required to have our “agent” software installed.

JMAPI from a Programmers Perspective

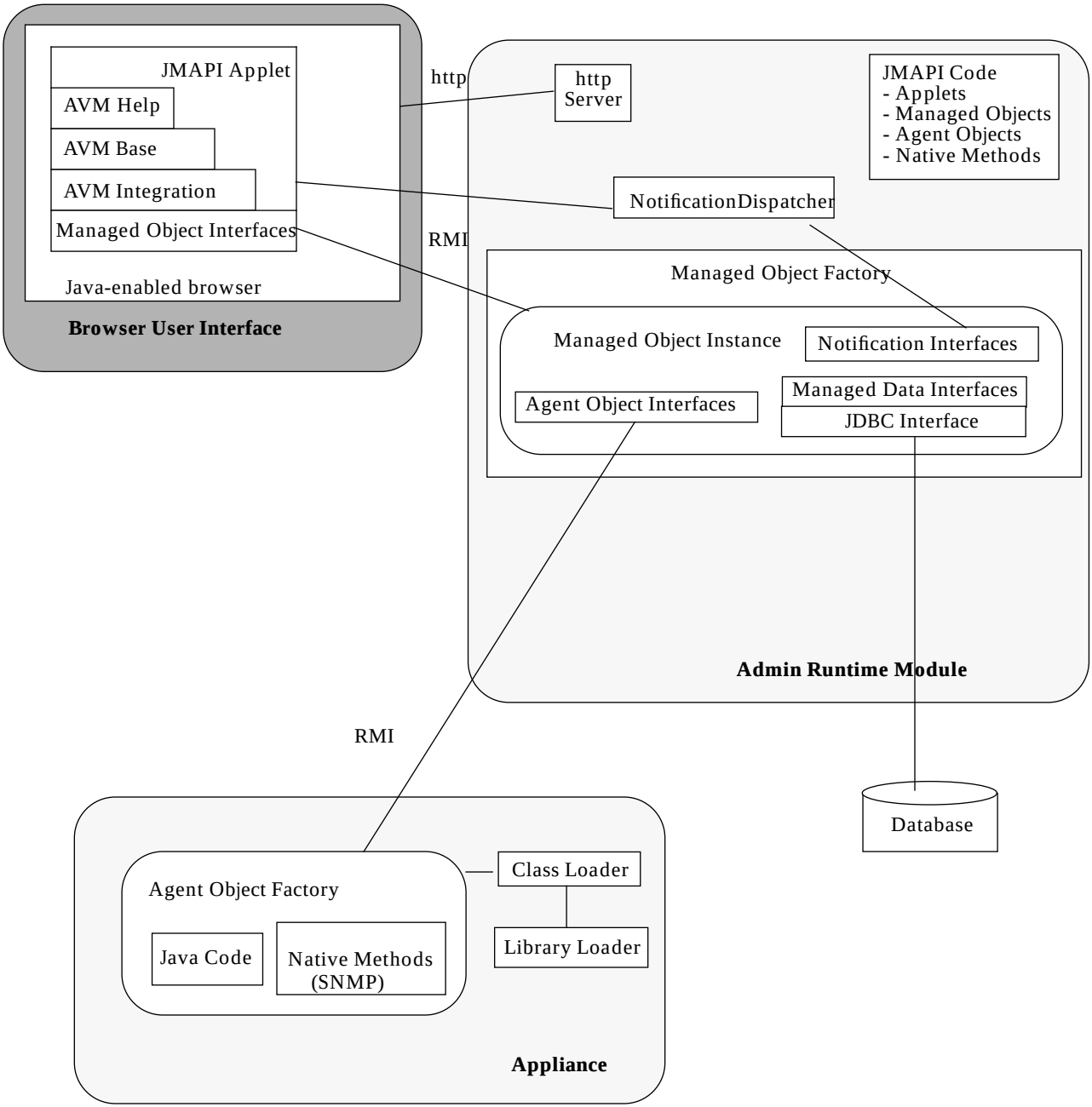
JMAPI is inherently distributed and a programmer must write components for each of the three elements (BUI, ARM, and Appliance) for the JMAPI model.

On the BUI, the programmer must construct Java applets using classes from the Admin View Module and the Managed Object Interfaces. On the BUI, Managed Objects must be implemented. On the Appliance, Agent Objects must be implemented. Figure 1-1 shows these elements and their relationship.

The components use the *Remote Method Invocation (RMI)* system for communication across machine boundaries. The components require that computations running in different address spaces, potentially on different hosts, be able to communicate. For a basic communication mechanism, the Java language supports sockets, which are flexible and sufficient for general communication. However, sockets require the client and server to engage in applications-level protocols to encode and decode messages for exchange, and the design of such protocols is cumbersome and can be error-prone.

The Java remote method invocation system has been specifically designed to operate in the Java environment. The Java language's RMI system assumes the homogeneous environment of the Java Virtual Machine, and the system can therefore follow the Java object model whenever possible. Though not shown on Figure 1-1 the Java runtime system must be present on every component.

Figure 1-1 Component Decomposition


JMAPI Applet
AVM Help
AVM Base
AVM Integration
Managed Object Interfaces
Java-enabled browser
Browser User Interface
http
RMI
http Server
JMAPI Code
- Applets
- Managed Objects
- Agent Objects
- Native Methods
NotificationDispatcher
Managed Object Factory
Managed Object Instance
Notification Interfaces
Agent Object Interfaces
Managed Data Interfaces
JDBC Interface
Admin Runtime Module
RMI
Database
Agent Object Factory
Java Code
Native Methods (SNMP)
Class Loader
Library Loader
Appliance

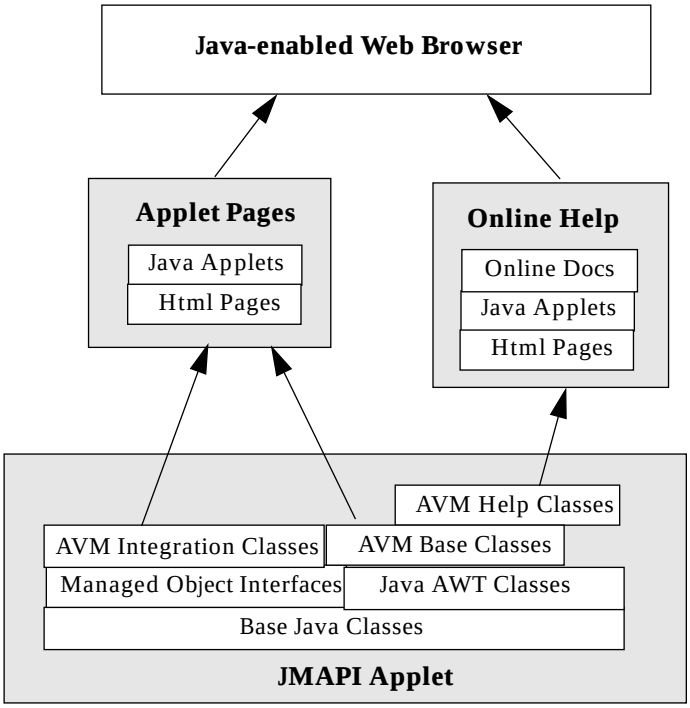
JMAPI Applet

The JMAPI applet is the component that gets down loaded onto an administrators web browser in order to perform some management operation.

Admin View Module (AVM)

The AVM is comprised of key client side classes for developers of JMAPI-based applets. The primary role of these classes is to provide user interface and application-level functionality. The AVM is built completely on top of Java's *Abstract Window Toolkit (AWT)*. By taking this approach the AVM is completely portable and window system independent. Figure 1-2 shows the overall architecture of the Admin View Module and how it is used to construct a JMAPI-based applet.

Figure 1-2 JMAPI Applet Model



The AVM classes are broken into three packages: AVM Help, AVM Base, and AVM Integration. This allows a developer to best decide what level of functionality and integration he wishes to introduce into an applet.

AVM Help

The primary objective of AVM Help is to provide a general purpose help environment. General purpose means it is intended that JMAPI application developers and ISVs can use the help system in a range of ways to provide application online help. As an example, applications that comply with JMAPI standards for conformity could have help files that integrate seamlessly into one common help system shared by multiple applications. This means that multiple applications would share one common index, glossary, table of contents / navigator, and so on. Applications that do not comply with JMAPI standards for conformity could have help files activated and displayed within AVM Help, but those help files are *not* integrated into one common help information space. Help for a non-conforming application simply has use of the help system, but does not reap benefits of integration such as consistency and related ease of use.

In keeping with the goal of a general-purpose help system, AVM Help does not require any proprietary tools, scripts, or programs to produce or generate help files. While tools may come along to facilitate authoring of AVM Help, they are not required. The only requirement is that help files be in HTML format and include a set of JMAPI help authoring tags that can be parsed during installation time to produce an index, glossary, and table of contents / navigator.

The AVM Help includes the following:

- **TOC/Navigator** — Displays a hierarchy of topics available in application help.
- **Documentation Generator** — The `jmapidoc` program is available for generating index, glossary, or TOC / Navigator files for an application. This doc generator can also be used by help authors to prototype and test their work.
- **Standard Help Files** — These are help files that application developers and ISVs can pass to their end users. These primarily include information such as Help on AVM Help, and Help on basic user interface components.

- **Help Templates** — These are HTML template files, including AVM Help keywords and examples of their use.
- **Search Engine** — TBD

The AVM Help classes depend only on the AVM Base classes, the Java AWT and the Java Base classes. This allows AVM Help to be used without any dependencies on the underlying managed objects.

AVM Base

The AVM Base classes are an extension of Java's Abstract Window Toolkit (AWT) that are specifically designed for developers creating integrated management solutions. These classes are used to implement a user model that builds on the Web browser hypertext style of navigation. The links used in navigation are between the services and resources that comprise the enterprise computing environment.

The AVM Base is further divided into three packages that reflect common functionality.

The Base User Interface classes naturally extend the AWT to provide a set of components necessary to build a rich user interface. These interfaces define the following classes:

- Image button
- Scrolling windows and panels
- Toolbar
- Image canvas
- Convenience dialogs
- Busy tool

The Power User Interface classes enable developers to use more graphical and eloquent components in constructing their solutions. These interfaces define the following classes:

- Table
- Hierarchy browser
- Charts and meters

The Management User Interface classes support the user interface style and interaction model described in the *JavaManagement API User Interface Style Guide*. These interfaces define the following classes:

- Property book
- Page header
- Link label
- View configuration

The AVM Base classes depend only on the Java AWT and the Java Base classes.

AVM Integration

The AVM Integration classes primary role is to provide integration between the AVM Base classes and the Managed Object Interfaces. By using these classes, an application developer constructs a management solution that is tightly integrated with all other JMAPI-based solutions. Probably, the most usage of these classes is to generate Property Books directly from a managed object.

The AVM Integration classes also allow for management applications to register their components, so when new solutions are provided, the integrity of the Web of management information can be maintained. These interfaces provide the following functionality:

- Register / Unregister management applets
- Register / Unregister management pages
- Register / Unregister management links
- Register / Unregister new extensions to the Managed Object Interfaces. This includes the Java class definitions, the implementation of the discover interfaces, and if necessary, default property books and class browsers. If the object simply extends existing object types, links must be applied to the existing object's property book.

The AVM Integration classes depend on the Managed Object Interfaces, AVM Base classes, Java AWT, and the Java Base Classes.

Managed Object Interfaces

Managed Objects are used to provide abstractions of resources and services in the enterprise. They use RMI to perform remote management methods and are derived from the class `ManagedObject`.

An application developer calls `MOFactory.initialize()` to establish a connection to a Managed Object Factory. To get a list of instances of a particular class of managed objects the method `MOFactory.listMOClass()` is invoked. This method uses the Managed Data Interfaces to retrieve a set of interfaces of a class from the database.To create a new instance of particular class of managed object the method `MOFactory.newObj()` is invoked. This method causes the Managed Object Factory to create a “shell” of the object which can be manipulated by the applet.

Only one instance of each object is created in the Managed Object Factory. The Managed Object Factory returns unique handles to each client. This means that multiple clients could potentially be updating the same managed object. JMAPI applets can register for notifications on changes to an instance of a managed object via the `ManagedObject.addObserver()` method.

Managed Objects

Managed Objects are RMI remote objects that are created and maintained by the Managed Object Factory. Managed Objects are where a large portion of the management functionality is implemented.

JMAPI applets access Managed Object Instances through three primary interfaces, as well as the getters and setters for the managed object attributes. These interfaces are `ManagedObject.addObject()`, `ManagedObject.deleteObject()`, and `ManagedObject.modifyObject()`. These methods all perform the same basic operations.

1. Establishes a transaction context to perform the operation
2. Prepares the database for any updates

3. Invokes `perform(Add/Delete/Modify)Actions()`. This method invokes Agent Object Instances to perform operations on the appliances. The `performActions` method must be implemented by a developer of a JMAPI managed object. If there are no actions, the developer can inherit an "empty" `performActions` method from the `ManagedObject` class.
4. Commits any database updates.
5. Closes the transaction context

The `performActions` method take a `CommitAndRollback` object as an argument. This object tracks what has been performed so that if any exceptions occur during the execution of a `performActions` method, the appropriate rollback operations can be performed. Likewise, when a `performActions` method completes successfully, any commit operations will be performed. This model enables developers of management operations to have a single way to perform clean-up when there is a failure.

Agent Objects

Agent Object Interfaces

The Agent Object are RMI objects that reside on the Appliance Since Agent Objects are really RMI objects, invocation of their methods look just like invocations on local Java instances. As discussed in "Managed Objects," Agent Objects are typically invoked in one of the `performActions` methods, though this is not required. Requests to create an Agent Object are made from managed objects by invoking `AgentFactory.newAgentObj()`.

When a method is invoked on an Agent Object, it calls Java code or Native Methods to implement the management operation. If necessary, the Class Loader will download the Java Code and the Library Loader will download the Native Methods.

Where To Go From Here

For a complete reference of all the classes, it is best to use the html files generated from JavaDocs. These files are generated directly from the JMAPI source and by default are the most accurate.

The remainder of this book describes in detail how to construct each of the above elements to make a completely JMAPI based solution.

Managed Objects 2

The JavaManagement API allows any GUI application running in a Java-enabled web browser to interact with managed object instances on a central server and for the objects to manipulate any number of appliances (management target systems) through remote agent methods.

Managed Objects implement distributed management functionality. These objects map closely to the resources that are to be managed. Basically, the managed objects are a proxy to the real resources in the Distributed Management Server and the appliance. These objects supply the interfaces that perform the real management operations. These objects all inherit directly or indirectly from the `managedObject` class. The `managedObject` class provides the methods that allow object developers to build on the database, security, and distribution elements of the architecture.

Example Managed Object

An example of a working managed object follows. Since it is intended only to demonstrate the mechanisms of the framework, it really doesn't do anything useful, however it will compile and run.

Note that the Managed Object is defined by an interface class and an implementation class, and that the classes are named `ExMO` and `ExMOImpl`.

Note – It is a requirement that the files be named this way; with the implementation class having the same name as the interface with `Impl` appended to it.

Look for the comment `//*** PACKAGE ***` throughout the code; in these places there are package name specifications that will probably need to be modified for your installation. Make these changes before compiling the code.

Before running the test, the Management Server must be configured. This includes setting up the database, importing your `ManagedObject` class into the database. This is discussed in more detail in Chapter 4, “Build and Execute Environment.”

Now, a subclass of `DBObject` is defined to hold the data attributes for the Managed Object. The attributes must be public because of some mechanism in the framework that allows the attributes to be set and retrieved by name. Since the instance of the `DBObject` that is stored in the managed object is hidden (private), there is no problem with the attributes of the instance being public. Since this class is now O2 persistent, there is some pre-process generated code that must be appended to the file before compiling. See Chapter 4, “Build and Execute Environment” for details.

Code examples for building a managed object are given in the following tables:

Code Example 2-1 Persistent Database Code Example

```
//*** PACKAGE ***
package sun.jmapi.example
// file DBExMO.java
import jar.manager.*;
// o2 imports
import database.api.*;
import database.runtime.*;

public class DBExMO extends DBObject {
```

```
    // Some simple attributes that are stored in the database. Data
    // that gets pushed out to the name services (NIS or NIS+) are
     // treated the same way as the Prism database data.

     public String exStringDBAttr;
     public int    exIntDBAttr;

     // More attributes, these are data that affect remote systems
     // (appliances) and are used as parameters in Agent Methods.

     public String exAMName;
     public int    exAMValue;

    // Actual appliance names for the agent methods. If it is not
     // evident from your other attributes what the appliance name
     // is for the agent methods, you will need explicit attribute
     // slots for them.

     public String applianceHostName;

     public String get MOClassName(){
     // ***PACKAGE***
     return “sun.jmapi.example.ExMOImpl”;
     }

     // NOTE: There is 02 generated code that must be appended to
    // this file before it is compiled. See the section on the Build
     // and Execution Environment for instructions.

}
```

Next, a server-side subclass of `ManagedObjectImpl` is defined to implement the operations on the MO:

Code Example 2-2 Implementation of a Managed Object

```
//*** PACKAGE ***
package sun.jmapi.example
// file ExMOImpl.java
// jdk imports
import java.io.*;
// jmapi imports
import jar.common.*;
import jar.manager.*;
import jar.agents.*;
```

```
import jar.rmi.*;
//rmi imports
import java.rmi.*;
import java.rmi.registry.*;

import agentclasses.ex.*;

import java.util.Properties;
import jar.common.Debug_level;

public class ExMOImpl extends ManagedObjectImpl implements ExMO {

   // Constructor allocates a DB object.

   public ExMOImpl() throws RemoteException {

   // Wrappers around set and get-by-name for the DBObject
   // attributes. Must be implemented for all attributes in the
   // DBObject.

   public String getExStringDBAttr() {
      return (String)getAttribute(“exStringDBAttr”);
   }

   public void setExStringDBAttr(String exStringDBAttr) {
      setKnownAttributes(“exStringDBAttr”, exStringDBAttr);
   }

   public int getExIntDBAttr() {
      return ((Long)getAttribute(“exIntDBAttr”)).intValue();
   }

   public void setExIntDBAttr(int exIntDBAttr) {
      setKnownAttribute(“exIntDBAttr”, new Long(exIntDBAttr));
   }

   public String getExAMName() {
      return (String)getAttribute(“exAMName”);
   }

   public void setExAMName(String exAMName) {
      setKnownAttribute(“exAMName”, exAMName);
   }
```

```
public int getExAMValue() {
   return ((Long)getAttribute("exAMValue")).intValue();
}

public void setExAMValue(int exAMValue) {
   setKnownAttribute("exAMValue", new Long(exAMValue));
}

public String getApplianceHostName() {
   return (String)getAttribute("applianceHostName");
}

public void setApplianceHostName(String applianceHostName) {
   setKnownAttribute("applianceHostName", applianceHostName);
}

// Define the Agent Methods for Add, Modify, and Delete
// operations on the Managed Object. Whenever the
// ManagedObject.Add() method is called, the
// performAddActions() method will be called. Same goes for
// Modify and Delete. Add, Modify, and Delete are standard
// operations supported by all Managed object classes.

protected synchronized void performAddActions(
      StatusObservable o, CommitAndRollback v)
   throws Exception
}

   // The "createFileMethod" agent method creates a file in
   // tmp with the "Name" argument as its name and writes the
   // value "Value" into the file.

   try {
   if (Debug_level.debugging(Debug_level.FLOW_DEBUGGING)) {
   System.out.println("Entering ExMOImpl:performAddActions");
   }

   ExAddAgentMethods agentMethod =  (ExAddAgentMethods)
      AgentObject.newNameAgentObj(getApplianceHostName(),
      "agentclasses.ex.ExAddAgentMethods",
      "ExampleAgentMethod1");

   agentMethod.createFileMethod(v, getApplianceHostName(),
      getExAMName(), getExAMValue());
```

```
      // foobarMethod doesn't do anything, this is just an example
       // to show that you can run as many remote agent methods,
      // on different appliances, as is mandated by your Managed
       // Object model.

       int result = agentMethod.foobarMethod();

   } catch (Exception e) {
     e.printStackTrace();
      throw e;
   }
   }

   protected synchronized void performModifyActions(
      StatusObservable o, CommitAndRollback v)
    throws Exception
   {
   }

   protected synchronized void performDeleteActions(
      StatusObservable o, CommitAndRollback v)
    throws Exception
{
      try {
      if (Debug_level.debugging(Debug_level.FLOW_DEBUGGING)) {
      System.out.println("Entering ExMOImpl:
         performDeleteActions");
      }

      ExDelAgentMethods agentMethod = (ExDelAgentMethods)
          AgentObject.newAgentObj(getApplianceHostName(),
          "agentclasses.ex.ExDelAgentMethods");
      lagentMethod.deleteFileMethod(getExAMName());

   } catch (Exception e) {
      e.printStackTrace();
      throw e;

   }
   }

   public String getDBClassName() {
   return "managed_objects.ex.DBExMO";
   }
}
```

Next, a client-side extension of the `ManagedObject` interface is defined for the public methods on the remote instance:

Code Example 2-3 Interface to a Managed Object

```
//*** PACKAGE ***
package sun.jmapi.example
// file ExMO.java
// jmapi imports
import jar.manager.ManagedObject;
// rmi imports
import java.rmi.*;

public interface ExMO extends ManagedObject {

   public String getExStringDBAttr() throws RemoteException;
   public void setExStringDBAttr(String exStringDBAttr) throws
      RemoteException;
   public int getExIntDBAttr() throws RemoteException;
   public void setExIntDBAttr(int exIntDBAttr) throws
      RemoteException;
   public String getExAMName() throws RemoteException;
   public void setExAMName(String exAMName) throws
      RemoteException;
   public int getExAMValue() throws RemoteException;
   public void setExAMValue(int exAMValue) throws
      RemoteException;
   public String getApplianceHostName() throws RemoteException;
   public void setApplianceHostName(String applianceHostName)
      throws RemoteException;
}
```

The Agent Object class for adding an interface follows:

Code Example 2-4 Agent Object Interface Add Example

```
//*** PACKAGE ***
package sun.jmapi.example

// file ExAddAgentMethods.java

import java.io.*;
import jar.agents.*;
import jar.common.*;
import jar.manager.CommitAndRollback;
import java.rmi.*;

public interface  ExAddAgentMethods extends NamedAgentObject {

   public void createFileMethod(CommitAndRollback transState,
      String agentHost, String filename, int val)
     throws Exception, RemoteException;

   public int foobarMethod() throws Exception, RemoteException;
}
```

The Agent Object class for implementation of an add follows:

Code Example 2-5 Agent Object Implementation Add Example

```
//*** PACKAGE ***
package sun.jmapi.example

//file ExAddAgentMethods.java
import java.io.*;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;
import jar.agents.*;
import jar.common.*;
import jar.manager.CommitAndRollback;

public class ExAddAgentMethodsImpl  extends NamedAgentObjectImpl
      implements ExAddAgentMethods {
   public ExAddAgentMethodsImpl() throws RemoteException {
   super();
   }
```

```
    public void createFileMethod(CommitAndRollback transState,
       String agentHost, String file name, int val)
       throws Exception, RemoteException
    {
       // There are three parts to an Agent Object: the setup, the
       // commit, and the rollback. In the case of this object,
       // where we are creating a file, we can look at it two ways:
       // 1. Setup is create the file, commit is no-op, rollback
       //    is to remove the file.
       // 2. Setup is no-op, commit is create the file, rollback is
       //    a no-op.
       //
       // There may be performance concerns when choosing a setup,
       // commit, rollback strategy. It is up to the Managed Object
       // developer to make the choice.
       //
       // Here, we are coding the first choice, just to demonstrate
       // doing something in both the setup and rollback methods.

       String fname = "/tmp/" + filename;

       DataOutputStream f = new DataOutputStream(new
          FileOutputStream(fname));
       f.writeInt(val);
       f.flush();
       f.close();

       Vector arguments = new Vector();
       arguments.addElement(fname);

       AgentCallImpl ac = new AgentCallImpl(this,
          "rollbackCreateFileMethod",arguments);
          transState.addRollbackCall(ac);
       }
    public void rollbackCreateFileMethod(String inargs) throws
        Exception {
       File f = new File(inargs);
       f.delete();
    }

    public int foobarMethod() throws RemoteException, Exception {
    return 9;
    }
}
```

The Agent Object class for deleting an interface follows:

Code Example 2-6 Agent Object Interface Delete Example

```
//*** PACKAGE ***
package sun.jmapi.example

// file ExDelAgentMethods.java

import java.rmi.*;
import jar.agents.*;
import jar.common.*;
public interface ExDelAgentMethods extends Remote {
   public void deleteFileMethod(String filename) throws
      RemoteException, Exception;
}
```

The Agent Object class for deleting an implementation follows:

Code Example 2-7 Agent Object Interface Delete Example

```
//*** PACKAGE ***
package sun.jmapi.example

// file ExDelAgentMethodsImpl.java

import java.io.*;
import jar.rmi.*;
import jar.rmi.server.*;
import jar.agents.*;
import jar.common.*;

public class ExDelAgentMethodsImpl extends UnicastRemoteServer
      implements ExDelAgentMethods {
   public ExDelAgentMethodsImpl() throws Remote Exception {
   super();
   }

   public void deleteFileMethod(String filename throws
      RemoteException, Exception {
   try {
      Stirng fname = “/tmp/” + filename;
      File f = new File)fname);
      f.delete();
   } catch (Exception e) {
```

```
        e.printStackTrace();
        throw e;
    }
    }
}
```

A test driver to see the `ManagedObject` in action. Note that since the test is doing an add and then a delete, it appears that nothing happens. You can comment out the delete to see that a stringified `VecList` representing the object's data is written to the database (to the file `/export/db/managed_objects.ex.DBExMO`) and that the add agent method is run, which creates the file `/tmp/exMOfilename`).

Note that the appliance hostname is wired in as `localhost`. To see real remote appliance manipulation, change the name of the appliance host in the set call in the driver and make sure a `MasterAgent` is running on the target appliance.

Code Example 2-8 Test Driver Example

```
//*** PACKAGE ***
package sun.jmapi.example
import jar.manager.*;
import jar.rmi.*;
import java.rmi.*;
import java.rmi.registry.*;

public class ExMODriver {
   public static void main(String[] args) throws Exception [

      ExMO     mo;
     StatusObservable s= new StatusObservable();

     try {
         //*** PACKAGE ***
```

```
          mo = ExMO)MOFactoryObj.newObj("sun.jmapi.example.ExMO",
                "example");
      } catch (Exception x) {
       System.err.println("Failed to instantiate remote object:");
      x.printStackTrace();
      System.exit(1);
      }

      try {
      mo.setExStringDBAttr(the string db attribute");
      mo.setExIntDBAttr(9);
      mo.setExAMName("exMOfilename");
      mo.setExAMValue(42);
      mo.setApplianceHostName("localhost");
      } catch (RemoteException rex) {
          System.err.println(
            "Failed to call method on remote object:");
      rex.printStackTrace();
      System.exit(2);
      }

      try [
          mo.addObject(s);
      } catch (Exception x) {
          System.err.println(
            "Failed to call add method on remote object:");
          x.printStackTrace();
      }

   try {
      mo.setExIntDBAttr(mo.getExIntDBAttr() +1);
      } catch (RemoteException rex) {
          System.err.println(
            "Failed to call method on remote object:");
      rex.printStackTrace();
      System.exit(3);
   }

      try {
```

```
      mo.modifyObject(s);
      } catch (Exception x) {
          System.err.println(
             “Failed to call mod method on remote object:”);
          x.printStackTrace();
      }

      try {
      mo.deleteObject(s);
      } catch (Exception x) {
          System.err.println(
             “Failed to call del method on remote object:”);
          x.printStackTrace();
      }

      System.exit(0);
    }
}
```

Managed Object Notification Callback Mechanism

In the JMAPI system, managed object instances reside in a single Java VM on the management server system. Multiple clients of a single managed object interact with RMI handles that refer to the same object instance. Because of this, it is desirable to have a mechanism that allows clients to be notified when another client makes a change to a shared managed object. This paper describes a mechanism implemented in the JMAPI for managed object notification, and presents the Java API for using the callback mechanism in client code.

Notification Callback Model

The implementation is modeled closely after the Java Developer’s Kit (JDK) `util` package `Observer` and `Observable` classes. Clients register interest as observers of observable managed objects. However, there are some differences in the mechanism due to the fact that the actual managed object instance lives in a different Java VM than the client (and may be living of separate systems).

Clients call methods on remote managed objects through the Java remote method invocation (RMI) system. The notification callback mechanism was also implemented with the Java RMI. Because the managed object can’t make direct method calls on the client because they are in different Java VMs, the

client creates a remote object to act as an “observer proxy,” and hands the observer proxy object to the managed object in a registration call. When a change of interest occurs, the managed object makes a RMI call on the proxy, which in turn makes a local Java method call on the client.

Notification Callback API

There are several new classes and additions to existing classes that are of interest to users of the notification mechanism. The Java documentation (javadocs) for these classes should be consulted for documentation of the API. The following section will provide an example of how a client might register for notification with a managed object.

- `jar.manager.MOObserver.java` – Interface that must be implemented by a client who wishes to observe ManagedObject changes.
- `jar.rmi.ObserverProxy(Impl).java` – The RMI class that provides the remote callback mechanism that allows `ManagedObjects` on the Management Server to call back to a client with information about an observable change. A client will instantiate a `ObserverProxyImpl`, and hand it to a MO via a registration call. The managed object will make RMI calls on the object through the `ObserverProxy` interface. The proxy object will make local calls to the client’s update() method, which is required to implement the `MOObserver` interface.
- `jar.manager.ManagedObject(Impl).java` – New public methods have been added for registering and unregistering as an observer of a managed object instance.
- `jar.manager.Observation.java` – Abstract class for Observation objects, which contain information about an observable change to be returned to the observer.
- `jar.manager.AddObservation`,`DeleteObservation`, `ModifyObservation`, `SetterObservation.java` `jar.rmi.NewObservation.java` – Concrete observation classes for observable operations `AddObj`, `DeleteObj`, `ModifyObj`, `SetKnownAttribute`, and `NewSemanticObj`.
- `jar.rmi.ManagedObjFactory(Impl).java` – New public methods have been added for registering and unregistering as an observer of managed object class instantiations.

- `jar.rmi.MOFactory.java` – New client-side public methods have been added to communicate with the server-side factory to de/register as an observer of managed object class instantiations.

Notification Callback Example

The following is a notification callback example:

Code Example 2-9

```
// client.java
// class must implement MOObserver to be notified of MO changes

class Client implements MOObserver {
   // generic client code -- instantiates a managed object,
   // registers interest on modify operations.
   // ... code joined in progress

   ManagedObject mo = (ManagedObject)MOFactory.newObj(
        className, moName);

  // Create an observer proxy for the client so the managed object
  // can make RMI calls back to the proxy, which can then in turn
  // call the client. Observer takes a MOObserver as an argument,
   // usually a reference to the client (“this”).

   ObserverProxyImpl op = new ObserverProxyImpl(this);
   mo.addObserver(op, ManagedObject.OBSERVE_MODIFY);
```

```
    public void update(ManagedObject mo, Observation ob) {
     // If the client is only registered for interest on Modify, it
      // can assume that any Observation that's passed in is a
     // ModifyObservation. Otherwise, it should call "instanceof"
     // to determine what sort of notification it got. For example,
     // if the client had registered interest on Modify and Delete,
      // the code might look like this:

      if (ob instanceof DeleteObservation) {
          DeleteObservation dob = (DeleteObservation)ob;
         // Handle the delete notification appropriately, use the
          // information provided in the DeleteObservation object
      } else if (ob instanceof ModifyObservation) {

         ModifyObservation mob = (ModifyObservation)ob;
         // Handle the modify notification appropriately, use the
         // information provided in the ModifyObservation object.

      } else {

        // Surprise! Some other type of notification. Won't happen
         // unless the client was registered for other than Delete
        // or Modify, but can always get the message field that all
         // observations contain.

         System.out.println(ob.getMessage());
      }
    }
}
```

Event Management 2

The JavaManagement API event management service (EMS) provides the basic foundation that allows you to build complex event management. The model features asynchronous event notification between managed objects and management applications. This eliminates the need for management applications to poll at regular intervals to check for state changes.

The event management model facilitates the building of hierarchical event management services. This allows the event management services to be configured in a number of different ways. Different EMS can register interest in each other's events. This allows each level of the hierarchy to isolate the event to the domain that it has control over.

The EMS itself is an efficient light weight mechanism that can and should be used liberally. Any host in the enterprise can be designated as providing an event management service.

All JMAPI framework events are sent to this branch of the tree. Applications can register with this branch to receive all JMAPI events.

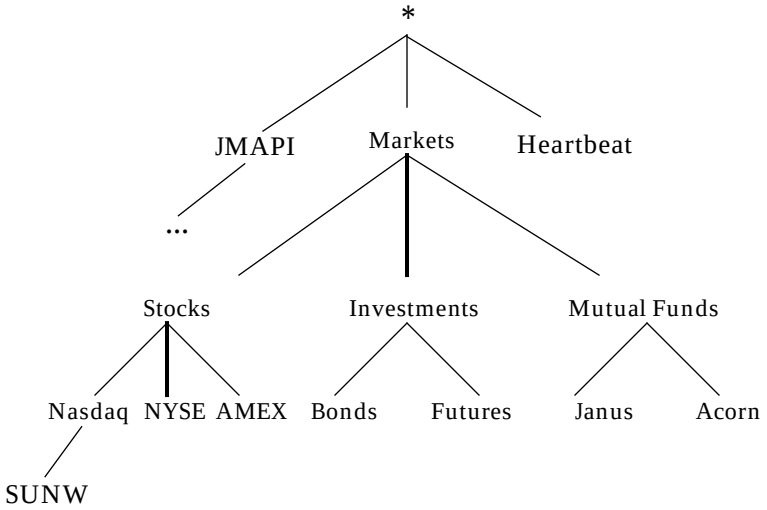
Event Tree

The basic EMS component is the event tree. The event tree is a hierarchical, ordered (n-ary) tree of period separated strings or words. Each of these words represents a branch or level of the tree. Applications can register interest in a specific branch of the tree. When an event is received, by the Event Dispatcher, the event tree is traversed until the branch is found.

Once the branch is found all registered consumer's Filter objects are called. If the Filter evaluates true then the Action class for that consumer is executed. Once all of the registered consumers of this branch have been found, the Event Dispatcher then works it way back up the tree searching for additional registrations. All of the found registrations are then processed just like the initial branch. The further down the tree hierarchy the more specific the event registration becomes. The opposite is true the higher up the tree the registration. Consumers registered at the tree root are considered for all events to the tree.

For instance the following tree could be constructed to monitor financial markets.

Figure 2-1 Event Management Service Example



In the example shown in Figure 2-1, you could register with the branch `*.markets.stocks.Nasdaq.SUNW` to be notified of events to the Nasdaq stock SUNW. To be notified of all events posted for Nasdaq stocks you could register with the `*.branch markets.stocks.Nasdaq` branch. To receive events for all market events you would register with the `*.markets` branch.

Event Generation

Events can originate from within the JMAPI framework or come from an outside source. The event has associated with it a set of attribute name and value pairs (note: see `VecList`). This data itself is opaque to the event dispatching mechanism and is passed directly to the registered consumer's filter and action objects.

Note – It has not been determined how to generate an event from C.

Registration

To receive notification that an event has occurred the application must register interest in the event. To register for event notification the application must provide two classes, the filter and the action. These classes are the basis for filtering and performing an action based on the event.

Unregistration

Once an application no longer is interested in receiving events it can unregister with the event handler.

Filter and Actions

When an application registers with the Event Dispatcher it must supply a filter and an action class. These classes control the behavior the registry when an event is triggered.

The filter and action classes that are registered must implement the Filter and Action interfaces.

Heartbeat

The Event Dispatcher generates a heartbeat event every 60 seconds. Applications can register with this branch and be delivered a heartbeat events.

Build and Execute Environment 4

The Java RMI system requires that you are running Java 1.0.2. If you are running an earlier Java Development Kit, such as version 1.0, you must upgrade.

The current O2 layer supports Sybase as a backend. You need a Sybase server installed and running on your management server.

You will also need the JMAPI remote call stub generator utility class, lava.Install.class. This is part of the JMAPI distribution.

Before starting the build, you should set the following environment variables to the appropriate values:

Table 4-1 Environment Variables

RMIHOME	/usr/rmtc/mosaic/java-rmi	#your RMI installation
RMI_BIN	$RMIHOME/bin	
RMI_CLASSES	$RMIHOME/classes	
RMI_LIBS	$RMIHOME/lib/sparc	
SYBASEHOME	/net/arete/work/sybase	#your Sybase installation
SYBASE_LIBS	$SYBASEHOME/lib	
O2HOME	/usr/rmtc/mosaic/o2	# your O2 installation
02_LIBS	$O2HOME/lib	
SYBASE_SERVER	arete	# your Sybase server host

After the variables are set in your environment, update your build and execute environment CLASSPATH and LD_LIBRARY_PATH variables. Your CLASSPATH must now additionally include $RMI_CLASSES, and your LD_LIBRARY_PATH must now additionally include $RMI_LIBS, $SYBASE_LIBS, and $O2_LIBS.

The JMAPI system supports calling methods by name (even outside of the RMI objects). Because this is not supported directly by Java, Remote Call Stub classes must be generated for all of the non-RMI classes that you develop (such as Agent Method classes). To generate the remote call stubs, run:

```
% java lava.Install path-to-package-root your.package.your.ClassName
```

For example, if the ExMO agent method code above were being developed in a directory called /dev/agents,

```
% java lava.Install /dev/agents sun.jmapi.example.agents.ExAddAgentMethod
```

would generate a stub source file called sun_jmapi_example_agents_ExAddAgentMethods.java. Then compile and install the stub file along with your managedObject class files, so that your CLASSPATH will locate both the managedObject classes and the Remote Call stub class:

```
% javac sun_jmapi_example_agents_ExAddAgentMethods.java
```

For your MOImpl classes (classes which extend ManagedObjectImpl) you will need to generate RMI Stub and Skeleton classes. After you javac compile these classes, you must run the rmic utility program to generate the stubs and skels. For example, for the ExMOImpl class in the example,

```
% $RMI_BIN/rmic sun.jmapi.example.ExMOImpl
```

will generate classes called ExMOImpl_Stub.class and ExMOImpl_Skel.class. These classes should be installed with your MOImpl class so that your CLASSPATH will locate both the managedObject classes and the stubs and skeletons.

Now you're ready to start up the Management Server. The server system must be running the Sybase database server and an RMI Object Factory.

To start the server:

```
% TBD
```

To start the factory:

```
% java jar.rmi.ManagedObjFactoryImpl <port>
```

The port is a command line argument to jar.rmi.ManagedObjFactoryImpl.

If you get a message indicating that there is already a binding for the factory, possibly another user is running a factory on this host.

The Factory can be passed the `rebind` command line argument to force the old binding to be thrown away and a new binding to be registered:

```
% java ... jar.rmi.SemanticObjFactoryImpl rebind
```

To connect your client to the Management Server, point your test driver at the server:

```
% java -DREGISTRY_HOST=servername [-DREGISTRY_PORT=<port>] \
        sun.jmapi.example.ExMODriver
```

Glossary

Action

In the event management system, this is a class provided by the registering customer. If the filter evaluates true then the Action object is used.

Consumer

Processes the event manager data. Consumers register with the Event Management System for event notification and process the event data. In the event management system, the consumer processes the event data. Consumers register with the even management system for event notification and process the event data.

Dispatcher

Dispatches events that were sent from producers to all consumers that are registered to receive the event.

Event

Is the asynchronous notification that something of significance has occurred. The notification contains specific information about the event. This is often referred to as the event data.

Filter

Is used as a means to only process specific events. An example would be to filter only on type.

Managed Object

A collection of data that a system administrator can manage and the methods that operate on that data. The data should closely resemble a representation of a real world object, such as a user on the system or network and a host system on the network.

Managed Objects are the classes derived from the JMAPI base-level object, `ManagedObject`.

Managed Object Factory

A daemon that resides on the Management Server. This daemon creates the managed object implementations (`ManagedObjectImpl` class) on the server. There is a single managed object implementation per instance of Managed Object class.

Manager Classes

The Java classes that implement the data and methods for managed objects. Manager classes are the managed object types.

`ManagedObject`

The root of the manager class hierarchy. All manager classes will extend `ManagedObject`.

MO

An abbreviation of Managed Object.

Object Factory

A generic term for a daemon that creates objects. See Managed Object Factory.

Prism

The database that will store a golden copy of managed object data, as well as a serve as a repository for our Java classes and system metadata. Prism provides Java interfaces that enable JMAPI developers to access the data.

Producer

Sometimes called a supplier, it is the entity that produces the event and its data.

Registry

Is the repository for all registered consumers. There is a sink for every type of filter that a customer has registered. This is sometimes referred to as a filter repository.

Registration

The process of making consumers known to the event management service. A customer registers to receive notification of certain events.

Sink

Refer to Registry.

Adobe PostScript

CETTE PUBLICATION PEUT CONTENIR DES MENTIONS TECHNIQUES ERRONEES OU DES ERREURS TYPOGRAPHIQUES. DES CHANGEMENTS SONT PERIODIQUEMENT APPORTES AUX INFORMATIONS CONTENUES AUX PRESENTES, CES CHANGEMENTS SERONT INCORPORES AUX NOUVELLES EDITIONS DE LA PUBLICATION. SUN MICROSYSTEMS INC. PEUT REALISER DES AMELIORATIONS ET / OU DES CHANGEMENTS DANS LE(S) PRODUIT(S) ET / OU LE(S) PROGRAMME(S) DECRITS DANS DETTE PUBLICATION A TOUS MOMENTS.