CORBA API
Programmer Guide

Document 5010
Customization
Copyright Notice
Document 5010. Copyright © 2001 - present Aprisma Management Technologies, Inc., 273 Corporate Drive, Portsmouth, NH 03801 USA. All rights reserved worldwide. Use, duplication, or disclosure by the United States government is subject to the restrictions set forth in DFARS 252.227-7013(c)(1)(ii) and FAR 52.227-19.

Liability Disclaimer
Aprisma Management Technologies, Inc. ("Aprisma") reserves the right to make changes in specifications and other information contained in this document without prior notice. In all cases, the reader should contact Aprisma to inquire if any changes have been made.

The hardware, firmware, or software described in this manual is subject to change without notice.

IN NO EVENT SHALL APRISMA, ITS EMPLOYEES, OFFICERS, DIRECTORS, AGENTS, OR AFFILIATES BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS MANUAL OR THE INFORMATION CONTAINED IN IT, EVEN IF APRISMA HAS BEEN ADVISED OF, HAS KNOWN, OR SHOULD HAVE KNOWN, THE POSSIBILITY OF SUCH DAMAGES.

Trademark, Service Mark, and Logo Information
SPECTRUM, IMT, and the SPECTRUM IMT/VNM logo are registered trademarks of Aprisma Management Technologies, Inc., or its affiliates. APRISMA, APRISMA MANAGEMENT TECHNOLOGIES, the APRISMA MANAGEMENT TECHNOLOGIES logo, MANAGE WHAT MATTERS, DCM, VNM, SpectroGRAPH, SpectroSERVER, Inductive Modeling Technology, Device Communications Manager, SPECTRUM Security Manager, and Virtual Network Machine are unregistered trademarks of Aprisma Management Technologies, Inc., or its affiliates. For a complete list of Aprisma trademarks, service marks, and trade names, go to http://www.aprisma.com/manuals/trademark-list.htm.

All referenced trademarks, service marks, and trade names identified in this document, whether registered or unregistered, are the intellectual property of their respective owners. No rights are granted by Aprisma Management Technologies, Inc., to use such marks, whether by implication, estoppel, or otherwise. If you have comments or concerns about trademark or copyright references, please send an e-mail to spectrum-docs@aprisma.com; we will do our best to help.
Restricted Rights Notice
(Applicable to licenses to the United States government only.)
This software and/or user documentation is/are provided with RESTRICTED AND LIMITED RIGHTS. Use, duplication, or disclosure by the government is subject to restrictions as set forth in FAR 52.227-14 (June 1987) Alternate III (g)(3) (June 1987), FAR 52.227-19 (June 1987), or DFARS 52.227-7013 (c)(1)(ii) (June 1988), and/or in similar or successor clauses in the FAR or DFARS, or in the DOD or NASA FAR Supplement, as applicable. Contractor/manufacturer is Aprisma Management Technologies, Inc., 273 Corporate Drive, Portsmouth, NH 03801. In the event the government seeks to obtain the software pursuant to standard commercial practice, this software agreement, instead of the noted regulatory clauses, shall control the terms of the government's license.

Virus Disclaimer
Aprisma makes no representations or warranties to the effect that the licensed software is virus-free. Aprisma has tested its software with current virus-checking technologies. However, because no anti-virus system is 100 percent effective, we strongly recommend that you write-protect the licensed software and verify (with an anti-virus system in which you have confidence) that the licensed software, prior to installation, is virus-free.

Contact Information
Aprisma Management Technologies, Inc.
273 Corporate Drive
Portsmouth, NH 03801
Phone: 603.334.2100
U.S. toll-free: 877.468.1448
Web site: http://www.aprisma.com
Preface  8
   User Prerequisites .......................................................... 8
   How To Use This Guide ...................................................... 9

Introduction  10
   What Is CORBA? .................................................................... 10
   Sources of Information About CORBA .................................. 11
   Why SPECTRUM Uses CORBA ........................................... 12
   How CORBA Works ........................................................... 13
   SPECTRUM’s Implementation of CORBA ............................ 15
      The Standard Naming Service and the Visibroker ORB ........ 15
      The SPECTRUM IDL ......................................................... 16
   SPECTRUM Services and CORBA API Classes .................. 18
      CORBA Classes ............................................................ 18
      Non-CORBA Classes ....................................................... 20
      Parameter Classes ......................................................... 20
      Helper Classes .............................................................. 20
      Global Classes .............................................................. 20
   Security ............................................................................... 21
   Error Reporting ............................................................... 21
   Location Service ............................................................. 21
   Callbacks .......................................................................... 22
   Documentation .................................................................. 23
   Examples .......................................................................... 23
   Additional Visibroker Arguments ...................................... 24

SPECTRUM 7.0 and the CORBA API  25
   Overview ............................................................................. 25
   Architecture Improvements ............................................... 25
   Development Requirements ............................................... 26
      Java Development Specifics ............................................. 27
      Windows C++ Development Platform Specifics ................ 28
## Environment Setup

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment Checklist</td>
<td>37</td>
</tr>
<tr>
<td>The SDK (SPECTRUM Developer Toolkits) Directory</td>
<td>39</td>
</tr>
<tr>
<td>Java Documentation</td>
<td>39</td>
</tr>
<tr>
<td>.jar Files</td>
<td>39</td>
</tr>
<tr>
<td>Source Code Files</td>
<td>40</td>
</tr>
<tr>
<td>Environment Verification</td>
<td>40</td>
</tr>
<tr>
<td>Verify Jar File Installation</td>
<td>40</td>
</tr>
<tr>
<td>Verify Client/Server Communication</td>
<td>41</td>
</tr>
<tr>
<td>Verify Development Environment</td>
<td>41</td>
</tr>
<tr>
<td>Verify Java Permissions</td>
<td>42</td>
</tr>
</tbody>
</table>

## Examples

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domains - Hello World</td>
<td>43</td>
</tr>
<tr>
<td>Binding to a SPECTRUM CORBA Service</td>
<td>44</td>
</tr>
<tr>
<td>Security</td>
<td>45</td>
</tr>
<tr>
<td>Model Domain ID</td>
<td>46</td>
</tr>
<tr>
<td>Lists</td>
<td>48</td>
</tr>
<tr>
<td>Property Lists &amp; Properties</td>
<td>49</td>
</tr>
<tr>
<td>Displays</td>
<td>51</td>
</tr>
<tr>
<td>Model Types</td>
<td>52</td>
</tr>
<tr>
<td>MType ID</td>
<td>53</td>
</tr>
<tr>
<td>MType Name</td>
<td>53</td>
</tr>
<tr>
<td>MType Hierarchy</td>
<td>54</td>
</tr>
<tr>
<td>MType Attributes</td>
<td>54</td>
</tr>
<tr>
<td>MType Properties</td>
<td>54</td>
</tr>
<tr>
<td>Associations</td>
<td>55</td>
</tr>
<tr>
<td>Relations</td>
<td>56</td>
</tr>
<tr>
<td>Rules</td>
<td>57</td>
</tr>
<tr>
<td>Attributes</td>
<td>58</td>
</tr>
<tr>
<td>Attribute Properties</td>
<td>58</td>
</tr>
<tr>
<td>MType Attribute Properties</td>
<td>58</td>
</tr>
</tbody>
</table>
Models - Part 1 ..........................................................................................................................61
  Creating the Container ........................................................................................................ ..62
  The CsCValue Object ........................................................................................................... 62
  Model Creation ..................................................................................................................... 64
  Model Association ............................................................................................................. ...64
Models - Part 2 ..........................................................................................................................66
  Modeling a Device ................................................................................................................66
Attribute Values ........................................................................................................................69
  Getting Model IDs and Attribute IDs ..................................................................................69
  Reading an Attribute Value ..................................................................................................70
  Writing an Attribute Value ...................................................................................................70
  Reading Multiple Attribute Values ......................................................................................71
  CsCAtrValList/CsCAtrVal ..............................................................................................72
  Reading Values from an Attribute Table ..............................................................................73
Actions .......................................................................................................................................75
  The doAction Method ........................................................................................................... 75
  The Get Port List Action .......................................................................................................76
  The Create Connections Action ............................................................................................76
Filters .........................................................................................................................................77
  Pseudo Filters ........................................................................................................................77
  Filter Descriptions .................................................................................................................78
  A Simple Attribute Filter ...................................................................................................... 79
  A Double Attribute Filter ..................................................................................................... 80
  A Complex Attribute Filter ................................................................................................. 81
Watches .....................................................................................................................................83
  Callback Base Classes ......................................................................................................... 85
Events and Alarms .................................................................................................................... 86
  Creating an Event .................................................................................................................. 86
  Getting an Event ................................................................................................................... 89
  Watching for Event Creation ............................................................................................... 91
  Watching for Alarm Creation ............................................................................................... 93
  Getting Alarms ...................................................................................................................... 94
  Clearing Alarms .................................................................................................................... 95
Debugging .................................................................................................................................97

Exercises .................................................................................................................................98
  Exercise 1 - GetDomainID ...................................................................................................... 98
Preface

Welcome to the programmer guide for SPECTRUM’s CORBA API—the CORBA-based Java and C++ client interface to SPECTRUM services. The CORBA API replaces the original SpectroSERVER API (SSAPI) with a standard-based and highly refined programming interface designed to make it faster and easier to develop client applications that access the SPECTRUM knowledge base.

Please take a moment to read through this short preface, get an idea of what is included in this guide, and make sure you have the proper background information to start using this powerful new tool.

User Prerequisites

To successfully create SPECTRUM client programs using the CORBA API, there are three main prerequisites:

• A working knowledge of either the Java or C++ programming languages.
• An understanding of the SPECTRUM architecture.
• An understanding of CORBA and the CORBA toolkit you will be using.

You will already be familiar with these aspects of SPECTRUM if you have completed the Level 1 Developers Tools training course, which is highly recommended. However, if you are new to SPECTRUM, or just want to “brush up” on the facts, you should refer to the following SPECTRUM publications:

• SPECTRUM Concepts Guide (0647)
• Event Configuration Files Guide (5070)
• Generic SNMP Device Management User Guide and Toolkit (1316)

You can view and download these and other SPECTRUM product publications from the following Aprisma Management Technologies Web site:
Beyond these minimum requirements, it is recommended that you be familiar with object-oriented concepts and distributed applications.

**How To Use This Guide**

This guide is organized as follows:

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>Provides background information on the CORBA standard and explains how it is implemented through SPECTRUM’s CORBA API. The major classes are defined and described.</td>
</tr>
<tr>
<td>SPECTRUM 7.0 and the CORBA API</td>
<td>Provides information on the changes to the CORBA API that are implemented with SPECTRUM version 7.0.</td>
</tr>
<tr>
<td>Environment Setup</td>
<td>Identifies the components and configurations/settings required to begin programming with the CORBA API, introduces the supporting Java documentation, and provides instructions for running some short environment verification programs.</td>
</tr>
<tr>
<td>Examples</td>
<td>Provides Java source code examples using key methods from each of the major CORBA API classes, including a short “Hello World” example program that sets various properties required for all subsequent programs you will be writing.</td>
</tr>
<tr>
<td>Exercises</td>
<td>Provides exercises to clarify and help you master the concepts and techniques presented by the example programs in the previous section.</td>
</tr>
</tbody>
</table>
Introduction

This section provides background information on the CORBA standard and explains how it is implemented through the SPECTRUM CORBA API (aka SSORB). The major classes of the SPECTRUM CORBA API are defined and described.

The SPECTRUM CORBA API is the CORBA-based interface to SpectroSERVER. Since it depends on an Object Request Broker (the “ORB” in CORBA), it is also referred to as “the SSORB interface” or just “SSORB”—both within this document and in the directories and files that make up the SPECTRUM Developer Toolkits (SDK). The following topics will give you an overview of why Aprisma has developed this API, what it consists of, and what you as a developer can accomplish with it.

- What Is CORBA? (Page 10)
- Why SPECTRUM Uses CORBA (Page 12)
- How CORBA Works (Page 13)
- SPECTRUM’s Implementation of CORBA (Page 15)

What Is CORBA?

The acronym CORBA stands for Common Object Request Broker Architecture, which is a standard developed by the Object Management Group (OMG), a consortium of over 800 members committed to developing “technically excellent, commercially viable and vendor-independent specifications for the software industry.” The CORBA standard can probably be best categorized as “middleware”—i.e., a framework or infrastructure that in this case specifies how distributed applications can communicate across remote locations using an Object Request Broker (ORB), as diagrammed in Figure 1 on Page 11.
Figure 1: Client-Server Communication via an ORB

The ORB enables communication between software components written in different languages (C, C++, Java, Smalltalk, COBOL, ADA, Lisp, Perl, Tcl, Eiffel, Python) for different operating systems with different hardware architectures. In other words it allows disparate objects to find each other at different places on the network and invoke each other’s methods.

The ORB is not an OMG product; rather, it is an implementation of the OMG’s CORBA standard, which specifies the Interface Definition Language (IDL) that is used to define the interfaces to CORBA objects, the programming language mappings, the transport protocols, and several higher level services such as naming and security.

Sources of Information About CORBA


Another source of CORBA information is the vendor for the proprietary VisiBroker ORB used by SPECTRUM, which is from Borland Software Corporation. VisiBroker documentation can be accessed through the [http://www.borland.com](http://www.borland.com) web site.

Why SPECTRUM Uses CORBA

Many software products are composed of two basic parts. The first includes those services that customers value, and the second consists of all the services, libraries, and subsystems necessary for the first part to work—i.e., the infrastructure. Sometimes a company will
buy components of this supporting infrastructure, and sometimes (for reasons of availability or expense) the company will build the component itself. Even though these two basic parts are often somewhat intermingled and interdependent, it is still prudent for a software company to strive toward “componentization,” because it facilitates the replacement of custom infrastructure pieces with industry-standard components as availability and prices change over time.

SPECTRUM is just such a product. It offers highly valued infrastructure modeling and fault management services, but it also includes a significant number of purchased or home-grown infrastructure libraries and services, some of which are both more expensive to maintain than replace, and less functional than the replacement.

The SpectroSERVER API (SSAPI) is a good example of this kind of infrastructure component, especially in light of the pervasive use of Web-based (Java) applications. Written internally, SSAPI has been SPECTRUM's client/server plumbing for many years, and has served the product well. However, SSAPI is expensive infrastructure to maintain, and it does not accommodate Web-based client access very well. CORBA has proven to be a great replacement in both regards. In addition, CORBA opens the door for the further componentization of the SPECTRUM product, which will prove advantageous in the years to come.

SPECTRUM's CORBA API gives both C++ and Java client applications access to the same SPECTRUM core services covered by SSAPI. However, as noted above, the use of Java for Web-based applications is pervasive, and it shows no signs of slowing down. Thus the focus of the CORBA API and of this guide is on application development using Java. Indeed, CORBA is converging with Java, which originally had its own middleware, called RMI (Remote Method Invocation). RMI initially worked only with Java and used its own transport protocol, but RMI has been changed so that it uses the CORBA transport protocol, called the Internet Inter-ORB Protocol (IIOP). Since the JDK 1.2 release, Java has included a CORBA standard ORB. CORBA standards are also being extended to support Java features such as “call by name.” Moreover, the OMG is moving to adopt JavaBeans as its component standard.

In light of these facts, CORBA seems to be the best fit for developing Java applications. As an open standard, it allows customers to write CORBA applications that integrate SPECTRUM with their other CORBA servers.
How CORBA Works

The CORBA component that supports interoperability between remote applications is the Interface Definition Language (IDL). The IDL is used to define an API. Note that the term IDL is also used to define a particular interface. For example, SPECTRUM’s CORBA API consists of several IDL files, one for each type of object. This set of IDL files is often referred to collectively as “the SPECTRUM IDL.”

For each programming language, the CORBA standard defines a language mapping. The language mapping tells how an IDL is converted into a language-specific API. An IDL compiler implements this mapping by compiling an IDL file into a set of “stubs” and “skeletons” as shown in Figure 2. The stubs constitute a library of objects and methods that an application can use. The skeletons are empty objects and methods that are filled in to implement the server. The stubs include the header files used to compile the application (in the case of a C++ mapping) and the object library linked into the application. The application and server must also link in the Object Request Broker (ORB) library.

**Figure 2: CORBA Development Process**

To develop a server, you need to write the IDL and the implementation inside the skeletons. To develop an application, you need to call the stubs. The skeletons and stubs are generated from the IDL and the ORB is provided by a vendor.

Each software program participating in the CORBA framework relies on its local ORB to overcome the communication barriers of language and process locality, feeding requests and responses to and from a data pipeline composed of all the ORBs in the system, a network of ORBs often called “the ORB.”
Each CORBA service is implemented as an object, written in a certain programming language and resident on a particular machine. To make itself available to clients, the object advertises its service via the ORB. A naming service (one of a handful of standard CORBA services) hears the broadcast and records the name of the service together with the service's object reference. When a client wants to use the service, it contacts the naming service via the ORB, specifying the service name and receiving the corresponding object reference. It then uses the object reference to access the object's services.

The object reference and its corresponding server object are both objects, instances of different classes with the same interface. An object reference (obtained by the client from the naming service) executes on the client machine. The corresponding server object resides on the server machine. When a client invokes the method of an object reference, the invoked method uses the ORB to call the corresponding method in the server object. That method then fulfills the service request and returns any relevant data. As far as the client is concerned, the local object reference acts just like the “real” service object when, in reality, it is only performing remote method invocations to the real server object.

To summarize: A programmer who creates a CORBA service object defines the interface in a CORBA-standard language called the Interface Definition Language (IDL) which, when run through the appropriate compilers, produces both server and client code in standard programming languages. The server code is called the “skeleton” because it represents a bare bones server object in the form of an abstract base class from which the programmer derives a class that implements the service. The client code consists of a set of object reference “stubs,” which require no further coding.

**SPECTRUM’s Implementation of CORBA**

This section describes how SPECTRUM works with the standard naming service, how the SPECTRUM IDL was developed, and identifies the major classes, supporting services, and documentation that are included in the SPECTRUM CORBA API.

**The Standard Naming Service and the Visibroker ORB**

CORBA Client Applications can now be migrated to use the Standard Naming Service. 6.X CORBA Client applications were restricted to using the proprietary OSAgent mechanism provide by Borland’s VisiBroker ORB. Migrating to the Standard Naming Service allows for ORB independence. With ORB independence, a client application does not require an OSAgent. All that is required is an ORB implementing the Standard
Naming Service. See *Standard Naming Service* on Page 32 for more information on ORB independence and *Location Service* on Page 21 for more information on Visibroker and the OSAgent mechanism.

**The SPECTRUM IDL**

The SPECTRUM IDL files define the interfaces to discrete services (model domain, event domain, etc.) within SPECTRUM. In addition to the IDL files, the SPECTRUM CORBA API consists of helper classes, examples, and documentation. However, the IDL is the essential part of the API. The IDL files are installed in the `<$SPECROOT>/SDK/examples/SSORB/idl` directory.

The SPECTRUM IDL is proprietary. This is important, because it means two CORBA-compliant management platforms will still not interoperate unless they use the same IDL. The SPECTRUM IDL is based on SSAPI and IHAPI in order to keep it close to the concepts that SPECTRUM’s legacy application developers know, so that they can learn the IDL quickly. At the same time, however, the API was improved, based on years of feedback from both internal and external developers.

As shown in Figure 3 on Page 17, the SPECTRUM IDL is based on a data model similar to what was used for the earlier toolkits. There are still models with attributes, model types, associations between models, relations, alarms, events, and statistics. The containers were renamed to make their role clearer: the term “landscape” has been replaced by “model domain”; the term “modeling catalog” had been replaced by “type catalog”; and the term “globalscape” has been replaced by “global model domain.” Some methods were moved to better encapsulate the objects — for example, model and association create methods now belong to model domain instead of model type and relation. Model and model domain handles are more hidden. New, more convenient helper methods were added, and method names were made consistent between objects.
Note that the CORBA API data model shows both “persistent” and “transient” services. This distinction refers to the fact that object references for persistent services can be stored by a client and reused to obtain a new object reference after a restart, whereas an object reference to a transient service is valid only as long as the current server session is running. For more information, see Location Service on Page 21.

The domains/services shown in Figure 3 are described individually in the following section.

**SPECTRUM Services and CORBA API Classes**

The CORBA API includes both CORBA and non-CORBA classes. The CORBA classes implement the SPECTRUM services shown in Figure 3. The non-CORBA classes include parameter, helper, and global classes.
CORBA Classes

These are the classes whose objects exist in SpectroSERVER or Archive Manger (ArchMgr) memory. These are the true CORBA classes that you will access by obtaining and using object references, as described under How CORBA Works (Page 13). You can find the Javadoc for these classes by opening the <$SPECROOT>/SDK/docs/SSORB/index.html file in your browser and clicking on com.aprisma.spectrum.core.idl under the Packages list. The supporting idl file for each of the CORBA classes listed below is located in the <$SPECROOT>/SDK/examples/SSORB/idl/ directory.

• CsCModelDomain - Model domain. Each instance of this class is a container of models and associations for a single model domain. One model domain is implemented in each SpectroSERVER. This is a persistent CORBA service. The supporting IDL file is cscmodeldomain.idl.

• CsCModel - Model. This class gives access to SPECTRUM models and their attributes. Every model has its own instance of this service. This is a transient CORBA service. The supporting IDL file is cscmodel.idl.

• CsCTypeCatalog - Type catalog. This class gives access to a single type catalog, which is a container of model types and relations. Even though each SpectroSERVER of the same release at a site is supposed to have the identical type catalog, differences are often found to occur in practice. Aprisma uses this type catalog to tie the server with the model domain. Each type catalog has its own instance of this service, with one type catalog being implemented in each SpectroSERVER. This is a transient CORBA service. The supporting IDL file is csctypecatalog.idl.

• CsCModelType - Model type. This class gives access to a model type and its attributes. A model type is a template for creating and classifying models. Each model type has its own instance of this service. This is a transient CORBA service. The supporting IDL file is cscmodeltype.idl.

• CsCRelation - Relation. This class gives access to a relation and its rules. A relation is a template for creating and classifying associations. Each relation has its own instance of this service. This is a transient CORBA service. The supporting IDL file is cscrelation.idl.
• CsCAlarmDomain - Alarm domain. This class gives access to a single alarm domain. An alarm domain is a container of alarms for models in a single model domain. Each alarm domain has its own instance of this service, with one alarm domain being implemented in each SpectroSERVER. This is a persistent CORBA service. The supporting IDL file is cscalarmdomain.idl.

• CsCEventDomain - Event domain. This class gives access to a single event domain. An event domain is a container of events for models in a single model domain. Each event domain has its own instance of this service, and one event domain is implemented in each Archive Manager. This is a persistent CORBA service. The supporting IDL file is csceventdomain.idl.

• CsCStatisticsDomain - Statistics domain. This class gives access to a single statistics domain. A statistics domain is a container of statistics for models in a single model domain. Each statistics domain has its own instance of this service. One statistics domain is implemented in each Archive Manager. This is a persistent CORBA service. The supporting IDL file is cscstatdomain.idl.

• CsCGlobalModelDomain - Global model domain. This class gives access to all models in all model domains. It is implemented identically in every SpectroSERVER. This is a persistent CORBA service. The supporting IDL file is cscgmodeldomain.idl.

The following IDL files are also included in SDK/examples/SSORB/idl, but the services they support are not considered to be real CORBA services:

cscattribute.idl — Attribute types and attribute values, which are reused throughout the other IDL.

cscdeveloper.idl — Definitions of the structures used for management module developers in the type catalog.

csecerror.idl — Error codes that are reused throughout the other IDL.

csecexception.idl — SPECTRUM exception that is reused throughout the other IDL since most methods raise this exception.

### Non-CORBA Classes
There are two categories of non-CORBA classes: parameter and helper.
Parameter Classes
This category of SSORB classes includes all non-CORBA classes used by SSORB methods to package and pass data. Your client programs will instantiate these classes and pass them to SSORB methods. The methods will also return instances of these classes to your programs. Examples include CsCModelPropList, CsCMTypePropList, and CsCAttrValList.

Helper Classes
These non-CORBA classes help you build and unwrap more complex SSORB objects. You can find the Javadoc files for these classes by clicking the com.aprisma.spectrum.core.util link located in the <$SPECROOT>/SDK/docs/SSORB/index.html file. Examples include CsCorbaValueHelper and CsCAttrFilterHelper.

Global Classes
These non-CORBA classes contain standard security/login objects used by SPECTRUM. You can find the Javadoc files for these classes in the <$SPECROOT>/SDK/docs/GLOBL/index.html file.

Security
Because the CORBA API is designed to accept calls from Java applets over the Web, security is a major concern. To continue to provide a level of security comparable to previous versions of SPECTRUM, the CORBA API includes passwords for SPECTRUM users. The API also includes helper classes to set the login and to maintain the secure connection. This security does not include encryption of the requests, but a future version will likely use the Secure Socket Layer (SSL) technique for encryption.

Error Reporting
The CORBA standard uses exceptions in Java and C++ to report errors. While consistency with the standard’s method of error reporting is desirable, an exception is an “all or nothing” mechanism; no other results are returned. This causes problems for partial
failures, in which a list of items is requested and only a few items fail. In practice, it would be inefficient to raise an exception whenever any item in a list failed. Aprisma therefore chose to report complete failures with exceptions and partial failures with error codes in result lists, similar to what SPECTRUM does with SSAPI. Since it is inefficient to search a list to see if any item failed, result lists also have an overall error code that indicates whether there is a partial failure.

**Location Service**

VisiBroker provides its own location service, called the **osagent** (for Orbeline Smart Agent; “Orbeline” was an earlier name for VisiBroker). VisiBroker also provides a program called **locserv**, which implements a CORBA service to access the osagent. This means you can use the VisiBroker IDL to get location service data, including the list of available model domains. Since one of the goals of developing a CORBA API was to reduce the amount of infrastructure that needs to be supported, Aprisma did not add its own location service IDL.

The distinction between persistent and transient services is important in regard to the VisiBroker location service, since it provides a persistent service but not a transient one. Also, persistent object references can be converted into a string and stored for later use, whereas transient objects cannot be used once a server has been restarted.
Callbacks

Most SPECTRUM applications watch for data to change, rather than polling the data repeatedly. This is more efficient and more responsive than polling for updates. It is more efficient because most polls merely confirm that the data is the same, and they read all the data instead of just the changed parts. It is more responsive because the application can handle the changed data as soon as it changes, rather than waiting for the next poll cycle.

With SSAPI, data is watched by attaching to a service. This is done for alarms, events, and statistics. The same thing happens when the application registers to watch for model, attribute value, or association changes. When the data changes, the change is sent as an unsolicited notification, meaning you get a response without having had to provide a request for the data.

---

**Note:**

The clients and servers communicate TCP with each other using TCP/IP. When the server is started, if the BOA is initialized with the `OAport` property, `OAport <port number>` is the port number to be used by the object adapter when listening for a new connection. This port is embedded in the server's Interoperable Object reference (IOR) and is used by the client to establish a connection to the server. If no `OAport` property is specified, a TCP port is randomly picked for the server listener port.

If the TCP port assignment changes when the server restarts, it is not possible to access the previously stored persistent information.

To ensure that the same TCP port is chosen consistently, set the `OAport` by adding the following line to the `<$SPECROOT>/SS/.vnmrc` file:

```
orb_args=-OAport <port number>
```

Where `<port number>` can be any unreserved port.
CORBA does not handle unsolicited notifications in the same way that SSAPI does. To achieve the same results, the application must implement a CORBA callback object and then send an object reference for the callback object to the server. Thus the application essentially becomes a CORBA server for the callback object. When the data changes, the SPECTRUM server will call a method of the callback object.

For more information, see *Watches* on Page 82 and *Callback Base Classes* on Page 84

**Documentation**

The SPECTRUM CORBA API development team has attempted to provide clear and comprehensively commented IDL, presuming that this would be the primary source that most developers would use. The IDL is also used to create Javadoc Web pages through the Java generated stubs. For more information, see *The SDK (SPECTRUM Developer Toolkits) Directory* on Page 38.

**Examples**

The toolkit also includes many working examples and some test programs, Java versions of which can be found in the `<$SPECROOT>/SDK/examples/SSORB/com/aprisma/spectrum/core/examples/cmdline` directory. The examples range from a simple program to get a model domain ID to a simple alarm view that uses callbacks to report alarms and a model view that uses callbacks to report associations between collected models.

**Additional Visibroker Arguments**

Additional Visibroker arguments that can be used with the classes provided by SPECTRUM’s CORBA API can be found in the Visibroker Programmer’s Reference. See [http://info.borland.com/techpubs/books/bes/pdfs52/pdf_index52.html](http://info.borland.com/techpubs/books/bes/pdfs52/pdf_index52.html) for more information.
SPECTRUM 7.0 and the CORBA API

This section details the changes made to the CORBA API as of SPECTRUM version 7.0.

Overview
Note that this section applies to upgrade considerations when adapting a pre-existing CORBA integration to the changes introduced in 7.0.

The CORBA API has been upgraded and enhanced for SPECTRUM 7.0. It is important to note that CORBA API applications developed against a SPECTRUM 6.X release will need to be modified and recompiled to properly integrate with SPECTRUM 7.0. Please refer below to the ORB Binding (Page 29) section for further modification details. It is also important to note that applications updated for 7.0 will not be compatible with 6.X releases.

Architecture Improvements
The CORBA API has undergone some significant improvements. First, native thread spawning was added to handle callback processing. For cases where client applications implement callbacks, this workload will be redistributed onto native threads thus freeing SpectroSERVER processing cycles for other work on multi-CPU systems.

SPECTRUM 7.0 has been updated to use Borland's Visibroker 5.2. SPECTRUM now optionally advertises through the CORBA Naming Service in addition to advertising services through the Visibroker OSAgent. This allows interoperability with other ORBs that support the standard Naming Service, such as Orbacus and Java.
Development Requirements

Table 1 documents relevant version requirements to be aware of when moving your CORBA-based SPECTRUM integration from SPECTRUM 6.x to 7.0.

Table 1: Platform Development Requirements

<table>
<thead>
<tr>
<th>Development Tools</th>
<th>SPECTRUM Version 7.0</th>
<th>SPECTRUM Version 6.X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Borland Visibroker</td>
<td>5.2</td>
<td>3.3</td>
</tr>
<tr>
<td><strong>Compilers for Solaris OS 7, 8 and 9</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sun C++ Compiler</td>
<td>5.3</td>
<td>5.0</td>
</tr>
<tr>
<td>Java SDK</td>
<td>1.4.2</td>
<td>1.3.1</td>
</tr>
<tr>
<td><strong>Compilers for Windows 2000 and XP</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Microsoft Visual C++</td>
<td>6.0</td>
<td>6.0</td>
</tr>
<tr>
<td>Java SDK</td>
<td>1.4.2</td>
<td>1.3.1</td>
</tr>
</tbody>
</table>
Java Development Specifics

Listed in Table 2 are the JAR files necessary for Java development. The SPECTRUM JAR file names have all been updated to reflect the 7.0 version. In addition for 7.0, three new JAR files are required for Java development:

- `vbhelper70.jar` contains the new bind helper.
- `lm.jar` and `vbsec.jar` are the required Borland Visibroker JAR files.

The location of all the jar files should be set by the `CLASSPATH` environment variable.

<table>
<thead>
<tr>
<th>JAR File</th>
<th>SDK</th>
<th>Directory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>global70.jar</td>
<td>SPECTRUM</td>
<td></td>
</tr>
<tr>
<td>ssorb70.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssorbbeaus70.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ssorbutil70.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>utilsrv70.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vbhelper70.jar</td>
<td></td>
<td>&quot;$SPECROOT*/lib&quot;</td>
</tr>
<tr>
<td>vbjapp.jar</td>
<td>Borland Visibroker</td>
<td></td>
</tr>
<tr>
<td>vbjorb.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>vbsec.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lm.jar</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Windows C++ Development Platform Specifics

Table 3 and Table 4 document the library, dynamic link library and header files necessary for C++ development on the Windows platform.

### Table 3: DLL and Library Files

<table>
<thead>
<tr>
<th>DLL / Library File</th>
<th>SDK</th>
<th>Directory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>libGlobl.a</td>
<td></td>
<td>SPECTRUM</td>
</tr>
<tr>
<td>libssorb.a</td>
<td></td>
<td>&lt;$SPECROOT&gt;\lib</td>
</tr>
<tr>
<td>libssorbutil.a</td>
<td></td>
<td>&lt;$SPECROOT&gt;\lib</td>
</tr>
<tr>
<td>orb_pr.lib</td>
<td></td>
<td>Borland Visibroker</td>
</tr>
<tr>
<td>orb_pr.dll</td>
<td></td>
<td>&lt;$SPECROOT&gt;\lib</td>
</tr>
<tr>
<td>cosnm_pr.lib</td>
<td></td>
<td>&lt;$SPECROOT&gt;\lib</td>
</tr>
<tr>
<td>cosnm_pr.dll</td>
<td></td>
<td>&lt;$SPECROOT&gt;\lib</td>
</tr>
<tr>
<td>vport_pr.lib</td>
<td></td>
<td>&lt;$SPECROOT&gt;\lib</td>
</tr>
<tr>
<td>vport_pr.dll</td>
<td></td>
<td>&lt;$SPECROOT&gt;\lib</td>
</tr>
</tbody>
</table>

### Table 4: Windows Header File Locations

<table>
<thead>
<tr>
<th>DLL / Library File</th>
<th>Header File Directory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>libGlobl.a</td>
<td>&lt;$SPECROOT&gt;\SDK\include\GLOBL</td>
</tr>
<tr>
<td>libssorb.a</td>
<td>&lt;$SPECROOT&gt;\SDK\include\SSORB\idl</td>
</tr>
<tr>
<td>libssorbutil.a</td>
<td>&lt;$SPECROOT&gt;\SDK\include\SSORB\util</td>
</tr>
<tr>
<td>File Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>----------------------------------------------------------</td>
</tr>
<tr>
<td>orb_pr.lib</td>
<td>Provided by Borland Visibroker SDK</td>
</tr>
<tr>
<td>orb_pr.dll</td>
<td>(not included in SPECTRUM CORBA toolkit)</td>
</tr>
<tr>
<td>cosnm_pr.lib</td>
<td></td>
</tr>
<tr>
<td>cosnm_pr.dll</td>
<td></td>
</tr>
<tr>
<td>vport_pr.lib</td>
<td></td>
</tr>
<tr>
<td>vport_pr.dll</td>
<td></td>
</tr>
</tbody>
</table>
Solaris C++ Development Platform Specifics

Table 5 and Table 6 document the shared object, library and header files necessary for C++ development on the Solaris Platform.

### Table 5: Shared Object and Library Files

<table>
<thead>
<tr>
<th>Shared Object / Library File</th>
<th>SDK</th>
<th>Directory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>libGlobl.a</td>
<td>SPECTRUM</td>
<td></td>
</tr>
<tr>
<td>libGlobl.so</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libssorb.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libssorb.so</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libssorbutil.a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libssorbutil.so</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libcosnm_r.so</td>
<td>Borland Visibroker</td>
<td></td>
</tr>
<tr>
<td>liborb_r.so</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libvport_r.so</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Table 6: Solaris Header File Locations

<table>
<thead>
<tr>
<th>Shared Object / Library File</th>
<th>Header File Directory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>libGlobl.a</td>
<td>&lt;SPECROOT&gt;/SDK/include/GLOBL</td>
</tr>
<tr>
<td>libGlobl.so</td>
<td></td>
</tr>
<tr>
<td>libssorb.a</td>
<td>&lt;SPECROOT&gt;/SDK/include/SSORB/idl</td>
</tr>
<tr>
<td>libssorb.so</td>
<td></td>
</tr>
<tr>
<td>libssorbutil.a</td>
<td>&lt;SPECROOT&gt;/SDK/include/SSORB/util</td>
</tr>
<tr>
<td>libssorbutil.so</td>
<td></td>
</tr>
<tr>
<td>libcsm_r.so</td>
<td>Provided by Borland Visibroker SDK</td>
</tr>
<tr>
<td></td>
<td>(not included in SPECTRUM CORBA toolkit)</td>
</tr>
<tr>
<td>liborb_r.so</td>
<td></td>
</tr>
<tr>
<td>libvport_r.so</td>
<td></td>
</tr>
</tbody>
</table>

**ORB Binding**

When developing an application with a SPECTRUM 6.X release, it was necessary to define an ORB class either through use of a Java Properties Class or by passing in the ORB class name as a java argument at runtime. Then an ORB object needed to instantiated and security needed to be set on the ORB before finally binding to a domain. Example 1 shows the previous binding method.

**Example 1: 6.X Binding Implementation**

```java
Properties props = System.getProperties();
props.put("org.omg.CORBA.ORBClass",
    "com.visigenic.vbroker.orb.ORB");
props.put("org.omg.CORBA.ORBSingletonClass",
    "com.visigenic.vbroker.orb.ORB");

com.inprise.vbroker.CORBA.ORB orb =
    (com.inprise.vbroker.CORBA.ORB)
    org.omg.CORBA.ORB.init(new String[0],null);

CsCorbaSec.setLoginInfo(orb,username,"","GetDomainID","1.0");
CsCModelDomain md = CsCModelDomainHelper.bind(orb,servername);
```
Example 2. 7.0 Binding Implementation

For SPECTRUM 7.0, a CORBA helper object was developed to simplify the binding process. Example 2 illustrates the method of binding. It is important to note that the CORBA Helper Class is part of the com.aprisma.util.corba package, so an import statement is required.

```java
import com.aprisma.spectrum.core.idl.*;
import com.aprisma.spectrum.core.util.*;
import com.aprisma.util.corba.*;
import java.io.*

public class Example70Bind
{
    public static void main(String[] args)
    {
        try
        {
            //*************************************************************
            // Initialize.
            //*************************************************************
            String domainName = new String("mySpectroServer");
            // New way
            CORBAHelper helper = CORBAHelper.getHelperImpl();
            helper.init(null, null);
            CsCModelDomain md = (CsCModelDomain)
            helper.getObjectImplementation(CsCModelDomain.class, domainName);
            //****************************************************************
            //Get the Model Domain Name
            //****************************************************************
            String name = md.getModelDomainName();
            System.out.println(name);
        }
        catch(Throwable e)
        {
            System.out.println(e);
        }
    }
}
```
CORBA Interface Updates

The com.aprisma.spectrum.core.idl package has been enhanced in SPECTRUM 7.0. Under the Interface section of the com.aprisma.spectrum.core.idl package, there are interfaces that are appended with the word Operations. These interfaces are super classes for their respective derive class and should not be extended from. The interfaces and classes of this package have introduced new methods. In particular, the domain interfaces have introduced new methods to simplify information retrieval. Please refer to the updated Java Documentation (Page 38) for information on the new methods available.

Additionally, a few specific enhancements have been made, which should be considered when porting a CORBA-based application to SPECTRUM 7.0. Only the first enhancement in the bulleted list below changes the behavior of an object/method that existed in 6.6. The remaining enhancements offer new functionality that may be leveraged to produce more efficient programs.

- If a client reads a list-typed attribute via one of the readAttrValue methods on CsCModel or CsCModelDomain, it will now correctly return a CsCValue of type CSC_OID_VALUE_LIST. Previously, when a list attribute was read, the first scalar value in the list was returned. The CSC_OID_VALUE_LIST type is an array of OID/value pairs. The CsCorbaOIDValueListHelper should be used to parse the CSC_OID_VALUE_LIST. The only exception is a list-typed attribute with an OID reference (like ifAdminStatus). In this case, the scalar result is returned.

- The following new methods have been added to the CsCModelDomain:
  - stopWatchAttrValsOfModelsByIDs
  - stopWatchAssocModelsOfModelsByIDs
  - doActionOnModelsByIDs
  - startWatchModelsByTypeIDs
  - stopWatchModelsByTypeIDs

- Two new elements (modelDomainID and modelID) have been added to the CsCModelAttrValList structure, for efficiency, so the caller doesn't have to invoke the CsCModel to get it.

- New CsCModelProperties::modelTypeID and CsCListOfModelPropList elements have been added to CsCModel to provide easy/convenient access to certain data without performing another SSORB call.
• A new `clearListAttrs` method has been added to `CsCModel` to allow applications to clear any existing values for list-typed attributes.

**Standard Naming Service**

CORBA Client Applications can now be migrated to use the Standard Naming Service. 6.X CORBA Client applications were restricted to using the proprietary OSAgent mechanism provide by VisiBroker. Migrating to the Standard Naming Service allows for ORB independence. With ORB independence, a client application does not require an OSAgent. All that is required is an ORB implementing the Standard Naming Service.

![Figure 4: Interoperability of ORBs](image)

- The CORBA naming service is disabled by default when SPECTRUM is installed. To enable it, modify the `<$SPECROOT>/lib/SDPM/partslist/NAMINGSERVICE.idb` file so processd automatically starts the Visibroker naming service. This is done by changing the line:
AUTOBOOTSTART;n;
to
AUTOBOOTSTART;y;

• At this point, processd must be restarted and the nameserv process will start.

Note:
If you are working on a machine that does not include a SpectroSERVER, you could instead start another CORBA standard naming service that another vendor may provide. See Figure 4 for an illustration of this.

• Then add the line below to both the <$SPECROOT>/SS/.vnmrc and <$SPECROOT>/SS/DDM/.configrc and restart the SpectroSERVER and ArchMgr. For CsCLocServMapInt add the use_naming_service line to <$SPECROOT>/LS/.locrc and restart the location service.

        use_naming_service=true

• The naming service advertisements can be verified by navigating to <$SPECROOT>/bin/VBNS and running.

        nsutil list <SS hostname>

        This will show:

        Bindings in <SS hostname>
        Object : CsCAlarmDomain
        Object : CsCEventDomain
        Object : CsCStatisticDomain
        Object : CsCTypeCatalogTranslation
        Object : RTMDomain
        Object : CsCModelDomain
        Object : CsCModelDomainTranslation
Examples of how to build CORBA client applications with other ORBs from other vendors using the standard naming service are included as part of the CORBA Toolkit. These examples have been compiled and tested with SPECTRUM 7.0. ORBs and Directory Locations are listed in Table 7.

**Table 7: Naming Service Examples**

<table>
<thead>
<tr>
<th>ORB</th>
<th>Development</th>
<th>Platform</th>
<th>Directory Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun</td>
<td>C++</td>
<td>Solaris</td>
<td>&lt;$SPECROOT&gt;/SDK/examples/SSORB/nameserv/cc/Sun_CC_lib</td>
</tr>
<tr>
<td>Orbacus</td>
<td>C++</td>
<td>Solaris</td>
<td>&lt;$SPECROOT&gt;/SDK/examples/SSORB/nameserv/cc/Orbacus</td>
</tr>
<tr>
<td>Java</td>
<td>Java</td>
<td>Windows Solaris</td>
<td>&lt;$SPECROOT&gt;/SDK/examples/SSORB/nameserv/java/sun</td>
</tr>
</tbody>
</table>
Environment Setup

This section identifies the components and configurations/settings required to begin programming with the CORBA API and provides instructions for running some short environment verification programs.

Environment Checklist

__Development Requirements. __See Development Requirements (Page 25) for information on the development requirements necessary for using the CORBA API with SPECTRUM 7.0 and above.

__SPECTRUM Version. __You must have SPECTRUM 7.0.X or above installed on your SPECTRUM machine with SpectroSERVER and ArchMgr running. You should also have SpectroGRAPH running on either your SPECTRUM server or client machine.

__Broadcast Domain. __Your SPECTRUM server machine and client machine can be different computers but, for purposes of the examples/exercise in this guide, they must reside in the same broadcast domain (i.e.—the set of all devices that will receive broadcast frames originating from any device within the set. A broadcast domain is typically bounded by routers, which do not forward broadcast frames).

__SPECTRUM Access. __If your SPECTRUM server and client machines are different computers, you must ensure that the .hostrc file in the SPECTRUM directory on your SPECTRUM server machine allows programs running on your client machine to interact with the SpectroSERVER machine. To do so, open the .hostrc file with a text editor, remove the “-” character if it exists, and add either a “+” character or the name of your client machine. Once you have made these changes to the .hostrc file, you can close
the file. You do not have to shut down and restart any SPECTRUM processes; they will automatically re-read the configuration file within a minute.

__User Model__. You must verify that SPECTRUM has a user model with a name that matches the username you will be using on your client machine when running the example programs. You can create a user model using the “User Editor,” which can be launched from the Tools menu in any SpectroGRAPH Topology view.

__Devices__. Many of the exercises require you to create models or read model attribute values. For this reason it is helpful to have a few actual SNMP devices (switches, routers, etc) running on your network so that you can model them. Remember, you can always model your own client machine if it is running a supported SNMP agent.

__Work Directory__. You should create a “work” directory in which to perform the exercises associated with the examples discussed in the following section (Examples on Page 42). Also, depending on the type of machine where you will be compiling and/or running your programs, you will need to set the environment variable CLASSPATH to point to the CORBA jar files as shown in one of the following platform-specific examples (where $SPECROOT is the complete path to your SPECTRUM installation directory):

For Windows NT/2000

CLASSPATH = <$SPECROOT>\lib\global70.jar;<$SPECROOT>\lib\ssorb65.jar;<$SPECROOT>\lib\ssorbutil70.jar;<$SPECROOT>\lib\vbjapp.jar;<$SPECROOT>\lib\vbjorb.jar;.

For Solaris

CLASSPATH = <$SPECROOT>\lib\global70.jar:<$SPECROOT>\lib\ssorb65.jar:<$SPECROOT>\lib\ssorbutil70.jar:<$SPECROOT>\lib\vbjapp.jar:<$SPECROOT>\lib\vbjorb.jar:.

You can then compile programs by entering:

javac <ProgramName>.java

And you can execute programs by entering:

java <ProgramName>
The SDK (SPECTRUM Developer Toolkits) Directory

This section describes how to locate, arrange, and use the example programs, jar (Java ARchive) files, and javadoc files you will need to perform the exercises in this guide and start using the CORBA API. If you have installed SPECTRUM, you may already have all the required files and directories, which are described below.

Java Documentation

To access the Java documentation (javadoc) for the SPECTRUM CORBA API, point your browser to <$SPECROOT>/SDK/docs/SSORB/index.html and under the list of Packages click on the hyperlink labeled com.aprisma.spectrum.core.idl. This will display the Interface Summary and, further down the page, the Class Summary. Clicking on any of the hyperlinks in these lists will take you to a description of that interface/class. If you scroll down past the description, you will find field, constructor, and method summaries. Under the Method Summary, you will see descriptions of the methods available on the selected object. Clicking on the hyperlink for any of the listed methods will take you to the Method Detail section, where you will see a description of the functionality of that method.

.jar Files

The <$SPECROOT>/lib directory contains several jar files including the following:

- global70.jar
- ssorb70.jar
- ssorbbeans70.jar
- ssorbutil70.jar
- utilsrv70.jar
- vbhelper70.jar
- vbjapp.jar
- vbjorb.jar
• vbsec.jar
• lm.jar

Note that the names of jar files supplied by Aprisma include a version indicator. For example, the “70” in global70.jar indicates version 7.0.0 of SPECTRUM.

**Source Code Files**

Another, lower-level directory (SDK/examples/SSORB/com/aprisma/spectrum/core/examples/cmdline) contains the .java source code files for all the command-line example programs.

**Environment Verification**

With a SpectroSERVER and ArchMgr running on your SPECTRUM machine, and the JDK installed on your client machine along with the SPECTRUM SDK containing the necessary example programs, jar files, and javadoc files, you are ready to verify your development environment by running the following tests:

- Verify Jar File Installation
- Verify Client/Server Communication
- Verify Development Environment

**Verify Jar File Installation**

This test verifies that Java can find and execute an example program located in the SDK/examples/SSORB/jars/ssorbcmdline.jar file. Perform the following steps:

1. cd SDK/examples/SSORB/com/aprisma/spectrum/core/examples/cmdline
2. rm *.class
3. java GetModels help

The GetModels example program should display the following output:

GetModels [ domainName [ filename ]]

---

CORBA API Programmer Guide   Page 38
Verify Client/Server Communication

To verify that the example programs can actually talk to your SpectroSERVER, perform the following steps:

1  cd SDK/examples/SSORB/com/aprisma/spectrum/core/examples/cmdline
2  java GetModels <domainName>

Be sure to substitute the actual DNS/network name of your SpectroSERVER host system for <domainName>. The GetModels example program should display a list of model IDs and model names from your SpectroSERVER.

Verify Development Environment

This test verifies that you can compile and run the example programs from the cmdline directory rather than from the ssorbcmdline.jar file. Here are the steps:

1  cd SDK/examples/SSORB/com/aprisma/spectrum/core/examples/cmdline
2  javac GetModels.java
3  java GetModels <domainName>

If you list the directory contents, you should see a GetModels.class file. This file is created by Step 2 and executed by Step 3. To verify that you are executing this class file (and not the one in the jar file) you may want add a print statement to the java source file, compile and run the program, and verify that the program executes your print statement.

Verify Java Permissions

At a minimum you must set the following permissions:

- The java.net.SocketPermission: You must copy the listen permission for server, accept the client connection, and, if you are using osagent, you must connect to the client and the server.
- The java.util.PropertyPermission: You must set both the read and the write permission.
- The java.lang.RuntimePermission: You must use the setContextClassLoader.
Other permissions will depend on the application that you are developing. For example, if you would like to read a properties file or write to a log file, you need to set the file permission.
Examples

This section provides examples with Java source code using key methods from each of the major CORBA API classes.

Each of the following topics includes links to one or more exercises designed to help you practice the techniques covered in that example. It is suggested that you perform them in the sequence listed, since subsequent examples and exercises build on techniques and concepts introduced in earlier ones. Note that the first example on domains includes a “Hello World” type exercise that will verify proper setup of your development environment.

- **Domains - Hello World**
- **Lists** (Page 47)
- **Displays** (Page 50)
- **Model Types** (Page 51)
- **Associations** (Page 54)
- **Models - Part 1** (Page 60)
- **Models - Part 2** (Page 65)
- **Attribute Values** (Page 68)
- **Actions** (Page 74)
- **Filters** (Page 76)
- **Watches** (Page 82)
- **Events and Alarms** (Page 85)
- **Debugging** (Page 95)
Domains - Hello World

If you have set up your development environment and familiarized yourself with the contents of the SPECTRUM CORBA API as described in the previous section, you are ready to use the API to access SPECTRUM core services.

The simple client program described in this section, `GetDomainID.java`, does not actually output a “Hello World” string, but, like a traditional Hello World program, it does perform a simple task that tests your development environment, and it serves as a template for all the other programs you will be writing if you perform the exercises associated with each of the topics in this section.

In a SPECTRUM environment, each SpectroSERVER has one model domain with a unique name and a unique identification number. The unique name is the DNS/network name of the SpectroSERVER host system itself. The unique ID is the same entity that is called the “landscape handle” in SSAPI terminology. The simple task that the `GetDomainID` program performs is to use a model domain name to get and display a model domain ID. In doing so, `GetDomainID` also performs the following functions, all of which are required of any program you will be writing:

- Bind to a SPECTRUM CORBA service
- Set security (optional)
- Display the Model Domain ID

These functions are described individually below.

Binding to a SPECTRUM CORBA Service

There is a CORBA helper object that has been developed to make binding to a SPECTRUM CORBA service simple. The following example shows you how to implement this object to bind to the model domain.

It is important to note that the CORBA Helper Class is part of the `com.aprisma.util.corba` package, so an import statement is required.

```java
import com.aprisma.spectrum.core.idl.*;
import com.aprisma.spectrum.core.util.*;
import com.aprisma.util.corba.*;
import java.io.*;
```
public class GetDomainID
{
    public static void main(String[] args)
    {
        try
        {
            // Initialize.
            String domainName = new String("mySpectroServer");
            // Use CORBA helper object
            CORBAHelper helper = CORBAHelper.getHelperImpl();
            helper.init(null, null);
            CsCModelDomain md = (CsCModelDomain)
            helper.getObjectImplementation(CsCModelDomain.class, domainName);
            // Get the Model Domain Name
            String name = md.getModelDomainName();
            System.out.println(name);
        }
        catch(Throwable e)
        {
            System.out.println(e);
        }
    }
}

Security

Each CORBA service requires its own level of security. For example, a financial database object might require an encrypted username and password. A SPECTRUM model domain, on the other hand, requires an unencrypted username. By default, security is preset to the user and host in which the Java application is implemented. Making changes to this security information is optional. If this ORB needs to be set to a user and host other than the default values, you can use the setLocalSecurityInfo method provided in the CorbaHelper Class. First you must instantiate and initial the helper object, then you can call the setLocalSecurityInfo method. The example below illustrates this.

```java
import com.aprisma.spectrum.core.idl.*;
import com.aprisma.spectrum.core.util.*;
import com.aprisma.util.corba.*;
import java.io.*;

public class GetDomainID
{
    public static void main(String[] args)
```
try {

// Initialize.
String domainName = new String("mySpectroServer");
// Use CORBA helper object
CORBAHelper helper = CORBAHelper.getHelperImpl();
helper.init(null, null);

// Modify User and Host
helper.setLocalSecurityInfo("Jon Doe", 
"mySPECTRUMComputer");

CsCModelDomain md = (CsCModelDomain)
    helper.getObjectImplementation(
       CsCModelDomain.class, domainName);

// Get the Model Domain Name
String name = md.getModelDomainName();
System.out.println(name);
}

Model Domain ID

Using the model domain object reference obtained earlier, GetDomainID can get and display the model domain ID as follows:

```java
import com.aprisma.spectrum.core.idl.*;
import com.aprisma.spectrum.core.util.*;
import com.aprisma.util.corba.*;
import java.io.*

public class GetDomainID {
    public static void main(String[] args) {
        try {
```
// Initialize.
String domainName = new String("mySpectroServer");
// Use CORBA helper object
CORBAHelper helper = CORBAHelper.getHelperImpl();
helper.init(null, null);

// Modify User and Host
helper.setLocalSecurityInfo("Jon Doe", "mySPECTRUMComputer");

CsCModelDomain md = (CsCModelDomain)
        helper.getObjectImplementation(
            CsCModelDomain.class, domainName); 

// Get the Model Domain Name
String name = md.getModelDomainName();
System.out.println(name);

// Get the Model Domain ID
int id = md.getModelDomainID();
System.out.println("0x" + Integer.toHexString(id));

} catch(CsCSpectrumException e) {
    System.out.println(e);
}

Note: Note:
The getModelDomainID method appears within a “try” statement because,
like most SSORB methods, it throws a CsCSpectrumException when it
fails. Subsequent code snippets in this manual frequently omit the “try” and
“catch” statements for the sake of brevity and clarity. However, you must
include them when you use the code snippets in actual programs.
Summary — This section described a simple SPECTRUM CORBA API client program called GetDomainID which binds to a model domain, sets the appropriate security information (optional), and finally gets and displays a model domain ID. These steps are common for all the client programs you will write using SSORB. To help you further understand the basic outline of an SSORB program, perform Exercise 1 - GetDomainID on Page 96.

Lists

The SSORB interface enables you to obtain lists of models, model types, relations, developers, attributes, attribute values, and several other types of information. Although these lists differ in content, they share certain qualities of form. This section helps you write a client program that obtains an SSORB list, and, in so doing, exemplifies several features common to SSORB lists and list handling. This program, called GetMTypes.java, gets and displays a list of all the 3000-plus model types (also referred to as mtypes) in your SpectroSERVER's type catalog.

Most client applications developed using the SPECTRUM CORBA API will use services provided by one or more of the following classes:

- CsCStatisticDomain
- CsCTypeCatalog
- CsCEventDomain
- CsCAlarmDomain
- CsCModelDomain

Aprisma recommends that you obtain object references to these services by first binding to the model domain and then invoking the appropriate CsCModelDomain method as follows:

```java
1 CsCModelDomain md = (CsCModelDomain)helper.getObjectImplementation(CsCModelDomain.class, domainName);
2 CsCAlarmDomain ad = md.getAlarmDomain();
3 CsCEventDomain ed = md.getEventDomain();
4 CsCTypeCatalog tc = md.getTypeCatalog();
5 CsCStatisticDomain sd = md.getStatisticDomain();
```
While it is possible for a client program to bind directly to any of the domains listed above, it is
discouraged in almost all cases. For one reason, if you bind directly to a type catalog, you
cannot specify to which SpectroSERVER's catalog you will bind, and while all type catalogs in
a multi-server installation should theoretically be identical, this is not always the case.
Further, even if you bind directly to an alarm, event, or statistic domain, you will almost
invariably need to access the corresponding model domain anyway.

**Property Lists & Properties**

To obtain the list of model types, the GetMTypes example program invokes the type catalog's
getAllMTypePropList method, which returns a CsCMTypePropList object, as follows:

```java
1  CsCModelDomain md =
( CsCModelDomain) helper.getObjectImplementation
2  ( CsCModelDomain.class, domainName) ;
3  CsCTypeCatalog tc = md.getTypeCatalog();
4  CsCMTypePropList mtpl = tc.getAllMTypePropList();
```

Here is what a CsCMTypePropList object looks like:

```java
1  class CsCMTypePropList
2  {  
3    CsCMTypeProperties[] list;
4    CsCError_e error;
5  };
```

This class contains an array and an error code. Each member of the array defines a single
mtype encased in a CsCMTypeProperties class:

```java
1  class CsCMTypeProperties
2  {  
3    CsCModelType modelType;
4    int modelTypeID;
5    String name;
6    CsCMTypeFlags flags;
7    CsCError_e error;
8    ...
9  };
```

This class contains a CsCModelType instance (which is a CORBA object reference), three
other variables, and an error code. The for() loop to unpack the list might look like this:

```java
for (int i = 0; i < mtpl.list.length; i++) {
    CsCMTypeProperties prop = mtpl.list[i];
    /* Process prop */
}
```
for(int i=0;i<mtpl.list.length;i++)
{
    CsCMTypeProperties mtp = mtpl.list[i];
    System.out.println("0x" + Integer.toHexString(mtp.modelTypeID) + " " + mtp.name);
}

There is nothing particularly difficult or noteworthy about CsCMTypePropList and CsCMTypeProperties except that many SSORB lists follow his same pattern including the following:

<table>
<thead>
<tr>
<th>Property List Class</th>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>CsCAtrrPropList</td>
<td>CsCAtrrProperties</td>
</tr>
<tr>
<td>CsCAttrValList</td>
<td>CsCAttrValue</td>
</tr>
<tr>
<td>CsCDevPropList</td>
<td>CsCDevProperties</td>
</tr>
<tr>
<td>CsCModelPropList</td>
<td>CsCModelProperties</td>
</tr>
<tr>
<td>CsCMTypePropList</td>
<td>CsCMTypeProperties</td>
</tr>
<tr>
<td>CsCRelPropList</td>
<td>CsCRelProperties</td>
</tr>
</tbody>
</table>

**Summary**  — Many SSORB lists conform to a pattern that includes a “Property List” class and a “Properties” class. This chapter outlined an SSORB example program called GetMTypes.java, which gets and displays a list of all model types in your SPECTRUM database. For practice in SSORB list handling perform Exercise 2 - GetMTypes on Page 96 and Exercise 3 - GetRelations on Page 97.

**Displays**

Many of the examples and exercises in this guide involve getting and displaying SSORB objects. For example, the exercise associated with the previous section directs you to write a program called GetMTypes.java, which will get and display an instance of the class CsCMTypePropList, as follows:
CsCMTypePropList mtpl = tc.getAllMTypePropList();
for(int i=0; i<mtpl.list.length; i++)
{
    CsCMTypeProperties mtp = mtpl.list[i];
    System.out.println("0x" + Integer.toHexString(mtp.modelTypeID) + " "+ mtp.name);
}

The first line of code obtains the mtype list, and the next several lines display the information. To help minimize the time you will need to spend writing code to display SSORB objects in subsequent exercises, and in actual development, you can use a special class developed for CORBA API training called SSORBWriter.java, which contains methods for displaying SSORB objects in a standard way. This writer class can be used in place of the for () loop in the code snippet above as follows:

CsCMTypePropList mtpl = tc.getAllMTypePropList();
SSORBWriter writer = new SSORBWRITER();
writer.printAll(mtpl);

SSORBWriter includes three display methods for each class that it supports. For example, it includes the following three methods for the CsCMTypePropList class:

public void print(CsCMTypePropList value) {}  
public void println(CsCMTypePropList value) {}  
public void printAll(CsCMTypePropList value) {}

The first method displays the object on one line with no carriage return/line feed. The second method does the same but adds the carriage return. The third uses as many lines as necessary to display the object, adding a carriage return to the end of each line.

Sometimes two or more kinds of SSORB data are represented by the same data type. For example, alarm IDs and event IDs are both represented by the data type byte[]. For these cases, SSORBWriter includes the name of the SSORB data in the method name, as in the following example:

public void printAlarmID(byte[] value) {}  
public void printAlarmIDln(byte[] value) {}
3 public void printAlarmIDAll(byte[] value){}

The SSORBWriter class provides the following two constructors:
1 public SSORBWriter(){
2 public SSORBWriter(String filename){}

The first constructor writes object information to System.out (which is usually your computer screen). The second constructor writes output to the specified file, creating the file if necessary, and truncating the file if it already exists.

Summary — This section described a special SSORBWriter class that you can use within your SSORB client programs to conveniently output SSORB objects to your computer screen or to a file. To practice using SSORBWriter, perform Exercise 4 - GetList on Page 97.

Model Types

The relationship between SPECTRUM model types (mtypes) and models is analogous to the relationship between classes and objects in the object-oriented world. Both mtypes and classes act as templates, while models and objects are instances. In preparation for working with models (see Models - Part 1 on Page 60 and Models - Part 2 on Page 65), this section presents several SSORB methods that enable you to inspect an mtype by examining its name and ID, positioning it within the mtype hierarchy, listing its attributes, and exploring its properties.

MType ID

Given an mtype ID, you can obtain the mtype object reference using the following code:

1 Integer mtypeID = Integer.decode(args[0]);
2 Cs CTypeCatalog tc = md.getTypeCatalog();
3 CsCElementType mt = tc.getModelType(mtypeID.intValue());
4 SSORBWriter writer = new SSORBWriter();
5 writer.printAll(mt);

The first line gets an mtype ID from the command line. The second gets an object reference to the type catalog. The third gets an object reference to the particular model
type. The fourth creates an SSORBWriter object for output to the computer screen. The fifth outputs the mtype's ID and name.

**MType Name**

Given an mtype name, you can obtain the mtype object reference in this way:

```java
1 String mtypeName = args[0];
2 CsCTypeCatalog tc = md.getTypeCatalog();
3 CsCMTypePropList mtpl = tc.getMTypePropListByName(mtypeName);
4 SSORBWriter writer = new SSORBWriter();
5 writer.printAll(mtpl);
```

The first line gets an mtype name from the command line. The second gets an object reference to the type catalog. The third gets a list of all mtypes that have the specified name. In almost all cases, this should have only one member. The fourth line creates an SSORBWriter object for output to the computer screen. The fifth line outputs the mtype list.

**MType Hierarchy**

Once you have a CsCModelType object reference, you can use two methods to determine the mtype's base and derived relatives. In the following example, `mt` is the CsCModelType object reference that you have previously obtained:

```java
1 CsCMTypePropList bmtpl = mt.getBaseMTypePropList();
2 CsCMTypePropList dmtpl = mt.getDerivedMTypePropList();
```

**MType Attributes**

To list an mtype's attributes, you can invoke a CsCModelType method with a notoriously long name: `getAllMTypeAttrGroupForest`. This method returns an array of CsCMTypeAttrGroupNode objects. Each node represents either an attribute or an attribute group. If the node represents the former, you can simply extract the attribute information from the node and output it. If it represents the latter, however, you must ask the attribute group how many attributes/attribute groups it contains. Assuming this number is `n`. The next `n` number of nodes will be members of the group. Fortunately, as discussed earlier, SSORBWriter includes a “print all” method that knows how to display an array of these
nodes. Once again, the \texttt{mt} in the following code snippet is an \texttt{CsCModelType} object reference:

1. \texttt{CsCMTypeAttrGroupNode[]} \texttt{nodes} = 
2. \texttt{mt.getAllMTypeAttrGroupForest();} 
3. \texttt{SSORBWriter writer = new SSORBWriter();} 
4. \texttt{writer.printAll(nodes);} 

\section*{MType Properties}

Each \texttt{mtype} in the type catalog has associated with it the following five boolean characteristics:

- \textbf{instantiable} - Models can be created from this model type.
- \textbf{underivable} - No model types can be derived from this model type.
- \textbf{noDestroy} - Models of this model type cannot be destroyed.
- \textbf{unique} - At most one model of this model type can exist.
- \textbf{required} - At least one model of this model type is required.

To read these characteristics, you must first obtain a \texttt{CsCMTypeProperties} object representing a single \texttt{mtype}. (Both the \texttt{CsCTypeCatalog} and \texttt{CsCModelType} classes have methods that return such an object.) Then you must access the \texttt{CsCMTypeProperties} “flags” member variable (of type \texttt{CsCMTypeFlags}) which contains one boolean variable for each of these five characteristics. Here is an example:

1. \texttt{Integer mtypeID = Integer.decode(args[0]);} 
2. \texttt{CsCTypeCatalog tc = md.getTypeCatalog();} 
3. \texttt{CsCMTypeProperties mtp = tc.getMTypeProperties} 
4. \texttt{(mtypeID.intValue());} 
5. \texttt{CsCMTypeFlags flags = mtp.flags;} 
6. \texttt{SSORBWriter writer = new SSORBWriter();} 
7. \texttt{writer.printAll(flags);} 

The first line gets an \texttt{mtype} ID from the command line. The second gets an object reference to the type catalog. The third and fourth get an \texttt{mtype} properties object. The fifth extracts an \texttt{mtype} flags object. The sixth gets a writer object. The seventh outputs the five \texttt{mtype} flags. Note that these five boolean characteristics are sometimes referred to as “\texttt{mtype} properties.”
Summary — This section outlined how to get and display a variety of mtype information including name and ID, hierarchical position, attributes, and properties. To help you write an example program that uses this new knowledge, perform Exercise 5 - InspectMType on Page 97.

Associations

Associations between SPECTRUM models are defined and governed according to the concepts of “relations” and “rules.”

A relation describes the way in which two associated models can interact and is usually expressed as a verb or verb phrase, such as Contains, Manages, Is_Adjacent_to, etc.

A rule is a statement such as “Model Type A Contains Model Type B” that applies the relation to two specific model types, thus permitting models of these two types to be associated through the Contains relation. Note that the grammatical subject of the rule statement (Model Type A) is referred to as the left side of the rule, while the grammatical object (Model Type B) is referred to as the right side. In other words, the side the model type appears on indicates its role in the relationship.

For many of the relations in the SPECTRUM knowledge base, there are one or more “meta-rules” that specify which model types may participate in that relation. Any model types derived from a model type specified in a meta-rule will inherit the ability to participate in that relation. For example, one of the meta-rules for the Contains relation is: Room Contains Device. If a model type called Closet is derived from the Room model type, then SpectroSERVER will automatically observe a Closet Contains Device rule. For a more detailed discussion of relations, rules, and meta-rules, refer to the SPECTRUM Concepts Guide.

The following two sections discuss example programs that deal with Relations (Page 55) and Rules (Page 56) respectively.
Relations

As noted above, relations are used to describe associations between individual models. Like attributes and model types, each relation has a unique hexadecimal handle.

Here are some examples:

<table>
<thead>
<tr>
<th>Relation</th>
<th>Handle</th>
</tr>
</thead>
<tbody>
<tr>
<td>Collects</td>
<td>0x10002</td>
</tr>
<tr>
<td>HASPART</td>
<td>0x10004</td>
</tr>
<tr>
<td>Manages</td>
<td>0x1001f</td>
</tr>
</tbody>
</table>

To obtain the complete relation list, the `GetRelation` example program invokes the type catalog's `getAllRelPropList` method, which returns a `CsCRelPropList` object, as follows:

```java
1 CsCTypeCatalog tc = md.getTypeCatalog();
2 CsCRelPropList rpl = tc.getAllRelPropList();
3 writer.printAll(rpl);
```

Here is what a `CsCRelPropList` object looks like:

```java
1 class CsCRelPropList
2 {
3     CsCRelProperties[] list;
4     CsCError_e error;
5 }
```

This class contains an array and an error code. Each member of the array defines a single relation encased in a `CsCRelProperties` class:

```java
1 class CsCRelProperties
2 {
3     CsCCardinality_e cardinality;
4     CsCError_e error;
5     String name;
6     CsCRelation relation;
7     int relationID;
8 }
```
Rules

The purpose of rules (and meta-rules) is to prevent the creation of illogical associations that may cause the SpectroSERVER intelligence to malfunction. Each rule specifies two model types that can participate in a particular relation. The rule also specifies the relative role of each model type by its position (left or right) within the statement.

Here are some examples:

<table>
<thead>
<tr>
<th>Left Side</th>
<th>Relation</th>
<th>Right Side</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universe</td>
<td>Collects</td>
<td>LAN</td>
</tr>
<tr>
<td>Port</td>
<td>Connects_to</td>
<td>Device</td>
</tr>
<tr>
<td>Building</td>
<td>Contains</td>
<td>Room</td>
</tr>
</tbody>
</table>

For programs that will create associations between models, it is good practice to check the existing rules to see which model types can be associated through the relation you have in mind. The following example program can be used to obtain a list of model types that participate on the left-hand side of a relation for a particular model type.

```java
1  CsCModelType mtype = tc.getModelType( modelTypeHandle );
2  CsCRelation relation = tc.getRelation ( relHandle );
3  CsCMTypePropList mtype_prop_list =
   4   relation.getRelMTypePropList( mtype, CsCSide_e.CSC_RIGHT_SIDE);
5  writer.printAll( mtype_prop_list );
```

In this example, a model type and the side the model type resides on (RIGHT_SIDE) are passed into the method, and a CsCMTypePropList is returned. The CsCMTypePropList will contain a list of model types that can exist on the other (left-hand) side of the rule.

Attributes

As described in the previous section, SPECTRUM mtypes have associated with them five boolean flags that are often termed “mtype properties.” Attributes have their own metadata which is divided into two parts: attribute properties and mtype attribute properties.
Attribute Properties

Each attribute has a set of seven boolean flags called “attribute properties” as described below:

- **external** - Attribute is external to SPECTRUM. It is an object that resides remote from SPECTRUM and must be obtained through a protocol such as SNMP.
- **readable** - Attribute is readable. It can be read by an application.
- **writable** - Attribute is writable. It can be written by an application.
- **shared** - Attribute has only one value for all models of a model type.
- **list** - Attribute is a list. List attributes generally represent a column of an attribute table.
- **guaranteed** - Attribute is guaranteed by the management module developer never to be removed.
- **global** - (Reserved for Aprisma use only)
- **preserveLegacyValue** - The attribute value in the database will not be overwritten by imported database catalog (.e) files. If you customize certain model types for specific requirements of your system and this this flag is set, those changed model type attribute values will remain in place when database model types are updated by subsequent release versions.

To read attribute properties, you must first obtain a CsAttrProperties object representing a single attribute. Then you must access the CsAttrProperties class’s “flags” member variable (of type CsAttrFlags), which contains one boolean variable for each of these eight characteristics. Here is an example:

```java
1   Integer attrID = Integer.decode(args[0]);
2   CsTypeCatalog tc = md.getTypeCatalog();
3   CsAttrProperties ap =
4       tc.getAttrProperties(attrID.intValue());
5   CsAttrFlags flags = ap.flags;
6   SSORBWriter writer = new SSORBWriter();
7   writer.printAll(flags);
```

The first line gets an attrID from the command line. The second gets an object reference to the type catalog. The third and fourth get an attribute properties object. The fifth extracts
an attribute flags object. The sixth gets a writer object. The seventh outputs the eight attribute properties flags.

**MType Attribute Properties**

In addition to attribute properties, each attribute also has the following set of mtype attribute properties:

- **polled** - Attribute value is read periodically. The frequency is set by the polling interval.
- **logged** - Attribute value is read periodically and saved in the SPECTRUM statistics log. The frequency is set by the logging interval.
- **memory** - Attribute value is stored in memory. This is used for attributes that are read frequently.
- **database** - Attribute value is stored in the database. This is used for attribute values that must be preserved between SpectroSERVER restarts.
- **oidPrefix** - First part of the object identifier (OID) within the MIB of an agent supporting a network entity. It is the branch of the MIB tree where a table attribute begins.
- **oidReference** - An attribute ID whose value is an OID suffix. This object identifier supplements the OID prefix in order to fully locate the object within the tree structure of the MIB. The rows of a table attribute are represented using this value as suffix. A zero value indicates that the OID reference is not specified for this model type attribute.
- **pollingGroup** - Polling group identifier. Attributes within a polling group are polled together and logged together. There are up to 256 polling groups for each model type. Each polling group is identified by a number from 0 to 255.
- **pollingInterval** - (This property is reserved for future use.)
- **loggingInterval** - (This property is reserved for future use.)

MType attribute properties are different than attribute properties in two ways. First, mtype attribute properties are not limited to boolean values. Second, they apply to an attribute for a particular mtype. To read mtype attribute properties, you must first obtain a CsCModelType object reference representing a single mtype. Then you must obtain a
CsCMTypeAttrProperties object and access several data members of the object, which will comprise the mtype attribute properties, as in the following example:

1. Integer mtypeID = Integer.decode(args[0]);
2. Integer attrID = Integer.decode(args[1]);
3. CsCModelType mt = tc.getModelType(mtypeID.intValue());
4. CsCMTTypeAttrProperties mtap =
   mt.getMTypeAttrProperties(attrID.intValue());
5. CsCMTTypeAttrFlags flags = mtap.flags;
6. int[] oidPrefix = mtap.oidPrefix;
7. int oidReference = mtap.oidReference;
8. int pollingInterval = mtap.pollingInterval;
9. int loggingInterval = mtap.loggingInterval;
10. int pollingGroup = mtap.pollingGroup;

The first and second lines of code get the mtype ID and attribute ID from the command line respectively. The third line gets the type catalog object reference. The fifth gets the mtype attribute properties object. The remaining lines access all the fields that compose the mtype attribute properties discussed in this section.

**Summary** — This section drew a distinction between attribute properties and mtype attribute properties. Both are attribute metadata. The first applies to the attribute in general. The second applies to the attribute for a specific mtype. To help you explore attribute metadata more fully, perform Exercise 6 - GetAttrProp on Page 98 and Exercise 7 - GetMTypeAttrProp on Page 98.

**Models - Part 1**

The CsCModelDomain class provides several methods for creating SPECTRUM models. Some methods perform very discrete tasks. For example, createModel does nothing more than create a model of a specific mtype. It does not name the model or set any other model attribute values, nor does it associate the new model with any other model. On the other hand, createAssocModelByIP discovers a network device at the specified IP address, creates a set of appropriate SPECTRUM models to represent the device, relates the models together, and relates the primary “device” model to a specified container model. In this section—Models (Part 1)—you will create and name a Network container model (mtype = 0x1002e) in the Universe view. In the next section—Models - Part 2 on Page 65—you will model devices and then collect the models into your new container.
Creating the Container

To create a container model and collect the model into the Universe view, you will need to use the following two CsCModelDomain methods: createModelWithAttrValue and createAssociation.

The first method creates a model of the specified mtype, and initializes one of the new model's attribute values to a specified value. Since you want to create a container model, you will need to specify a Network mtype (0x1002e). And to name the container, you will set the new model's Model_Name attribute (0x1006e) to a string representing the name. The second method creates an association between two models. For our purposes, we will specify Collects as the relation (0x10002), the Universe model as the left-hand model, and our container as the right-hand model. Note that the new container will not appear in the SpectroGRAPH's topology view until after the client program invokes the second method.

The CsCValue Object

If you look up createModelWithAttrValue in the SSORB Java documentation, you will notice that the third parameter is of type CsCValue. Used as a member variable in many SSORB classes and as an argument in several member methods, a CsCValue object is like an enum, in that it is able to hold one of a variety of different kinds of objects including the following:

<table>
<thead>
<tr>
<th>actionID</th>
<th>eventIDList</th>
<th>objectID</th>
</tr>
</thead>
<tbody>
<tr>
<td>agentID</td>
<td>gaugeValue</td>
<td>octetString</td>
</tr>
<tr>
<td>alarmID</td>
<td>hiddenValue</td>
<td>operatorValue</td>
</tr>
<tr>
<td>attributeID</td>
<td>intValue</td>
<td>realValue</td>
</tr>
<tr>
<td>attrGroupID</td>
<td>ipAddress</td>
<td>relation</td>
</tr>
<tr>
<td>attrValID</td>
<td>model</td>
<td>relationID</td>
</tr>
<tr>
<td>boolValue</td>
<td>modelID</td>
<td>taggedOctetString</td>
</tr>
<tr>
<td>counterValue</td>
<td>modelDomain</td>
<td>textString</td>
</tr>
<tr>
<td>dateTime</td>
<td>modelDomainID</td>
<td>timeTicks</td>
</tr>
</tbody>
</table>
To use the CsCValue class, you first need to instantiate it using the `void` constructor, which creates an object with no type or value:

```
1 CsCValue v = new CsCValue();
```

Next, set the object to a particular value type (e.g., textString) and value using the appropriate “set” method:

```
2 v.textString("Container III");
```

To extract the value from the object, use the appropriate “get” method:

```
3 String name = v.textString();
```

Note that the “set” and “get” methods are overloaded methods. If you want to determine the type of a particular CsCValue object, do the following:

```
4 CsCValueType_e type = v.discriminator();
```

To display the type and value of a CsCValue object, use the CsCorbaValueHelper class like this:

```
5 CsCorbaValueHelper vh = new CsCorbaValueHelper();
6 System.out.println(
7      v.discriminator().toString()
8      + " 
9      + vh.toStringValue(v));
```

The CsCorbaValueHelper class has a number of methods to help you deal with CsCValue objects. You can learn more about this helper class by clicking on the

<table>
<thead>
<tr>
<th>actionID</th>
<th>eventIDList</th>
<th>objectID</th>
</tr>
</thead>
<tbody>
<tr>
<td>developerID</td>
<td>modelType</td>
<td>tagString</td>
</tr>
<tr>
<td>enumValue</td>
<td>modelTypeID</td>
<td>uLong64Value</td>
</tr>
<tr>
<td>eventId</td>
<td>nullValue</td>
<td></td>
</tr>
</tbody>
</table>
Model Creation

The following example shows how you can use the CsCValue class to invoke the CsCModelDomain's create model method:

```java
1  Integer mtypeID = Integer.decode(args[0])
2  String modelName = args[1];
3  CsCTypeCatalog tc = md.getTypeCatalog();
4  CsCModelType mt = tc.getModelType(mtypeID.intValue());
5  CsCValue v = new CsCValue();
6    v.textString(modelName);
7  CsCModelProperties mp = md.createModelWithAttrValue(
8    mt, 0x1006e, v);
```

The first four lines are probably familiar to you by now. The first and second lines get the mtype and (future) model name from the command line respectively. The third and fourth get the type catalog and mtype object references. The fifth line creates a CsCValue object with no specific type or value. The sixth line sets the CsCValue object to a specific type (CSC_TEXT_STRING) and value (modelName). The seventh line creates the model, specifying the mtype, the attribute to initialize, and the value to use in the attribute initialization.

Model Association

After creating the container model, you can collect the model into the Universe as follows:

```java
1  CsCModel universe = ...
2  CsCModel myContainer = ...
3  CsCRelation relation = tc.getRelation(0x10002);
4  md.createAssociation(relation, universe, myContainer);
```

The first and second lines of code represent the two models (the Universe and your new container) for which you will create an association. The third line gets a CsCRelation object reference to the Collects relation. The fourth line creates the actual association, placing Universe in the “Collector” position and myContainer in the “Collected”
position. Once this method returns, an icon representing your “myContainer” model will appear in the Universe topology view of the SpectroGRAPH.

**Summary** — This section outlined how to create and associate models. To help you build a client application with which you can create your container model and collect it inside the Universe view, perform *Exercise 8 - CreateAndCollectNetwork* on Page 98.
Models - Part 2

“Creating a model” and “modeling a device” are different. The first involves explicitly telling SPECTRUM to create one model that usually represents a simple, unaddressable entity like a container or user. When you model a device, however, you simply stipulate the device (by IP address) to SPECTRUM, and let the server examine the device and create a set of interrelated models that appropriately represent the components and complexities thereof. In the previous section, you created a network model. In this section, you are going to model a device and place it into a Network container model.

Modeling a Device

Modeling a device is fairly straightforward, requiring a single method invocation:

```java
1  CsCModelProperties mp = md.createAssocModelByIP(
2       ip,                       // IP address of device.
3       community,                // community string.
4       5000,                     // time to wait for ping reply.
5       3,                        // retry count.
6       collects,                 // relation
7       collector,                // relate new model to universe.
8       CsCSide_e.CSC_LEFT_SIDE); // collector side of relationship.
```

The parameters from Lines 2 through 8 are discussed separately below. The `args[i]` variables represent command-line arguments passed to an imaginary program that contains all the code snippets.

**Line 2:** `ip` - This IP address identifies the target device. You can use a helper class to convert an IP address from string form (e.g., “134.141.64.81”) to the required `int[]` form as follows:

```java
CsCorbaValueHelper helper = new CsCorbaValueHelper();
int[] ip = helper.parseIPAddress(args[0]);
```

**Line 3:** `community` - This is the SNMP community string that SPECTRUM will use to satisfy security requirements when it contacts the device. It is a simple string:

```java
String community = args[2];
```
**Line 4:** 5000 - This is the time (in milliseconds) that SPECTRUM will wait between pings as it attempts to contact the device.

**Line 5:** 3 - This is the number of times SPECTRUM will attempt to ping the device before giving up.

**Line 6:** collects - This parameter is an object reference to a CsCRelation object containing the value 0x10002 which represents the “Collects” relation in SPECTRUM. It defines the relationship you want SPECTRUM to establish between the new model (i.e., the primary device model among the set of models that SPECTRUM creates to represent the specified device) and some container model (specified in the Line 7 description below). The object reference is initialized as follows:

```java
CsCRelation collects = tc.getRelation(0x10002);
```

**Line 7:** collector - This parameter is an object reference to a CsCModel object representing the model that will collect the new device. In this case, it is the container model created in the previous example:

```java
1   Integer collectorID = Integer.decode(args[1]);
2   CsCModel collector = md.getModel(collectorID.intValue());
```

Suppose you wanted to use the Universe model instead of your container model. In that case, you would have find the Universe model’s object reference. One way to do it is as follows:

```java
1   CsCValue value = new CsCValue();
2   value.modelTypeID(0x10091);
3
4   CsCModelPropList mpl = md.getModelPropListByAttrVal(5    0x10001
6       CsCOperator_e.CSC_EQUALS,
7       value);
8
9   if(mpl.list.length == 0)
10      { System.out.println("Universe model not found.");
11         return;
12    }
13} else if(mpl.list.length 1)
14      { System.out.println("Multiple Universe models found.");
15        
16    }
```

---

*CORBA API Programmer Guide*  
Page 64
In this code snippet, the value 0x10091 in the second line is the Universe mtype ID, and the value 0x10001 in the fourth line is the Modeltype_Handle attribute ID. The meaning of the model domain method invocation in the third line is this: Give me a list of all models that are instances of the Universe mtype. Although the answer to this question should always consist of one model, the code snippet still checks for possible conditions.

**Line 8:** CsCSide_e.CSC_LEFT_SIDE - This parameter specifies the left/right position of the collector model in the association that SPECTRUM will create. In a Collects relationship, the left side represents the collector and the right side specifies the collected.

**Summary** — The CsCModelDomain class provides two methods for modeling devices: createModelByIP and createAssocModelByIP. Both deal with modeling a device. The createAssocModelByIP method also creates an association between some existing model and the primary model (among the new set of models) representing the specified device. To practice modeling a device, perform *Exercise 9 - CreateModelByIP* on Page 98.
Attribute Values

As noted earlier, SPECTRUM models are instances of model types, just as objects are instances of classes. Further, models and objects contain values for the attributes defined in mtypes and classes. In SPECTRUM, these values are called, as you might expect, “attribute values.” This section shows you how to read and write a model’s attribute values.

Getting Model IDs and Attribute IDs

In order to read or write a model’s attribute value, you will need to specify a model ID and an attribute ID. The following example programs (located in the SDK/examples/SSORB/com/aprisma/spectrum/core/examples/cmdline directory) show you how to acquire a model ID:

- GetModel
- GetModels
- GetModelsByIP
- GetModelsByMType
- GetModelsByName

Once you have obtained a model ID, you must determine (if you have not done so already) the model’s mtype, which you can do using GetModelType. Then, with the mtype known, you can use GetAttrs to list all the attributes associated with that particular mtype. Your output, in part, might look something like this:

- Attribute: 0x1197d WatchNames
- Attribute: 0x1197e WatchAttrs
- Attribute: 0x127b5 PolicyDestList
- AttrGroup: 0x11ae3 Root 8
- Attribute: 0x10000 Modeltype_Name
- Attribute: 0x10001 Modeltype_Handle
- Attribute: 0x1007c Model_State
- Attribute: 0x110b4 model_verify
- Attribute: 0x111f1 Version_Number
- Attribute: 0x11b3d PollingMultiplier
- Attribute: 0x11b41 createtime
- Attribute: 0x11ee8 Model_Class
As you can see, each entry contains an attribute ID and an attribute name. By following the sequence described in this section, you can obtain the model ID and the attribute ID you will need to read or write an attribute value.

Reading an Attribute Value

The following code snippet demonstrates how to read and display a model's attribute value:

```java
1. Integer modelID = Integer.decode(args[0]);
2. Integer attrID = Integer.decode(args[1]);
3. CsCModel m = md.getModel(modelID.intValue());
4. CsCValue v = m.readAttrValue(
   attrID.intValue(),
   CsCAttrReadMode_e.CSC_MOST_CURRENT);
5. writer.printAll(v);
```

The first and second lines of code retrieve a model ID and an attribute ID from the command line. The third line obtains a model object reference from the model domain. The fourth line invokes the model's readAttrValue method, which returns a CsCValue object containing the value of the specified attribute. The seventh line displays the attribute value.

Writing an Attribute Value

The following example shows how to write a model's attribute value:

```java
1. Integer modelID = Integer.decode(args[0]);
2. Integer attrID = Integer.decode(args[1]);
3. String value = args[2];
4. CsCTypeCatalog tc = md.getTypeCatalog();
5. CsCModel m = md.getModel(modelID.intValue());
6. CsCAttrProperties ap = tc.getAttrProperties(attrID.intValue());
7. CsCorbaValueHelper helper = new CsCorbaValueHelper();
8. CsCValue v = helper.parseValue(value, ap.type, null);
9. m.writeAttrValue(attrID.intValue(),v);
```

Again, the first and second lines of code retrieve a model ID and an attribute ID from the command line. The third line retrieves the attribute value (to be written) and stores it (for
now) in a string. The fourth line gets the type catalog object reference from the model domain. The fifth gets the model object reference. The sixth gets all the properties of the target attribute. The seventh gets a CsCorbaValueHelper object. The eighth uses the attribute type (one of the attribute's properties) to convert the value from string form to native form. The ninth writes the attribute value to the target model.

**Reading Multiple Attribute Values**

You can also invoke a single method to read several attribute values of a particular model at a time as demonstrated by the following code:

```java
1  Integer modelID = Integer.decode(args[0]);
2  int[] ava = new int[]
3  {
4      SSORBHelper.attrIDVersion,
5      SSORBHelper.attrIDCreateDate,
6      SSORBHelper.attrIDCreateTime
7  };
8  CsCModel m = md.getModel(modelID.intValue());
9  CsCArrValList avl = m.readAttrValList(
10     ava,
11     CsCAttrReadMode_e.CSC_MOST_CURRENT);
12  writer.printAll(avl);
```

The first line of code gets the target model ID from the command line. Lines 2 - 7 construct an array of integers, each of which is an attribute ID. Note that SSORBHelper is a class created specifically for SSORB training and the examples/exercises within this document; it stores a variety of IDs. The eighth line gets a model object reference from the model domain. Lines 9 - 11 pass the read request to the model (in SPECTRUM), which reads the attribute values of all specified attribute IDs and returns the results in a CsCArrValList object (see below). The twelfth line displays the list of attribute values.

**CsCArrValList/CsCArrValue**

The CsCArrValList and CsCArrValue classes together provide a technique to package attribute ID/value pairs. Here is the definition for CsCArrValList:

```java
1  class CsCArrValList
2  {
3      CsCArrValue[] list;
4      CsCError_e error;
```
And here is the definition for the CsCAtrValue class:

```java
class CsCAtrValue {
    int attributeID;
    CsCError_e error;
    CsCValue value;
}
```

Several SSORB methods return a CsCAtrValList object. The following code snippet demonstrates how to unwrap a CsCAtrValList object called “avl”:

```java
for(int i=0;i<avl.list.length;i++)
{
    CsCAtrValue av = avl.list[i];
    int attrID = av.attributeID;
    CsCValue v = av.value;
}
```

Other methods, like the CsCModel class’s doAction method (discussed under Actions on Page 74) require a CsCAtrValList object as an argument.

### Reading Values from an Attribute Table

The reading of values from an attribute table is similar to the reading of a value from a single attribute. However, tables can sometimes be very large, and this can have some serious side effects beyond increased processing time—namely device performance degradation and increased network traffic. Fortunately, the CORBA table-read ability is robust in that it allows you to specify ranges (by defining high and low instances) and cap limits on the number of entries that can be retrieved. This can help improve performance when there are just a select few pieces of information that are needed from a table. Here is an example:

```java
int [] attr = {attrhandle.intValue()};
CsC OIDSpec low = new CsC OIDSpec();
CsC OIDSpec high = new CsC OIDSpec();
int [] lowValue = { 0 };
```
6 int [] highValue = { 100 };  
7 low.objectID( lowValue );  
8 high.objectID( highValue );  
9  
10 int length = 23;  
11  
12 CsCModel model = md.getModel(modelhandle.intValue());  
13 CsCArrtValTable tablevalues = model.readAttrValTable( attr, low,  
14 high, length,  
15 CsCAtrValReadMode_e.CSC_MOST_CURRENT );  
16  
17 for( int j = 0; j < tablevalues.table.length; j++ )  
18 {  
19     System.out.println(  
20         tablevalues.table[j][0].value.toString());  
21 }  

The first parameter is an array of attributes that will be read. It is recommended that the  
array specify attributes that belong in the table, since the instance array bounds may not be  
the same between different tables.  

The second and third parameters specify the low and high ends of the range of instances to  
read. The range is specified using CsCOIDSpec data parameters because instances can  
comprise more than one value, and because more often than not, tables are instanced by a  
series of values.  

The fourth parameter specifies the maximum number of table entries to collect. If the  
range of data exceeds the maximum limit, then just the maximum is specified. If the  
maximum is not met it will return just the entries it found in the table.  

The fifth parameter follows the single read convention.  

Note that the low, high, and limit parameters all have wild card ability.  

Here is a an example:  

1 int [] attr = {attrhandle.intValue()};  
2  
3 CsCOIDSpec low = new CsCOIDSpec();  
4 CsCOIDSpec high = new CsCOIDSpec();  
5 low._default();  
6 high._default();

The first parameter is an array of attributes that will be read. It is recommended that the  
array specify attributes that belong in the table, since the instance array bounds may not be  
the same between different tables.  

The second and third parameters specify the low and high ends of the range of instances to  
read. The range is specified using CsCOIDSpec data parameters because instances can  
comprise more than one value, and because more often than not, tables are instanced by a  
series of values.  

The fourth parameter specifies the maximum number of table entries to collect. If the  
range of data exceeds the maximum limit, then just the maximum is specified. If the  
maximum is not met it will return just the entries it found in the table.  

The fifth parameter follows the single read convention.  

Note that the low, high, and limit parameters all have wild card ability.  

Here is a an example:
7 int length = 0;
8
9 CsCModel model = md.getModel(modelhandle.intValue());
10 CsCAttrValTable tablevalues = model.readAttrValTable( attr, low,
11    high, length,
12    CsCAttrReadMode_e.CSC_MOST_CURRENT );
13
14 for( int j = 0; j < tablevalues.table.length; j++ )
15 {
16    System.out.println( tablevalues.table[j][0].value.toString());
17}

In the example above, the default method is called on the CsCOIDSpec objects. That sets the object up for the wild card capability. For the limit parameter, the wild card capability is set up by setting the limit to zero.

Summary — In this section you learned how to read and write attribute values. To help you write a client program using this knowledge, perform Exercise 10 - ReadAttrValue on Page 99 and Exercise 11 - WriteAttrValue on Page 99.

Actions

The CsCModel class has about 20 methods. Some methods enable client applications to get model information like name and ID. Others allow programs to read and write attribute values. And one method, doAction, exposes the ideosyncratic capabilities of specific mtypes. For example, a device model (one whose mtype derives from the Device mtype) can provide a list of its own ports. A device model can also discover and create associations to its neighboring device models. This chapter shows you how to use the doAction method to tap into mtype-specific functionality.

The doAction Method

You send an action to a SPECTRUM model by obtaining the model's object reference and then invoking the doAction method, which is defined as follows:

1 public abstract CsCAttrValList doAction(
2 int actionID, CsCAttrValList attrValList)
The first argument is the action ID. All actions are defined by a unique action ID. For example, the respective action IDs for the two actions discussed below—Get Port List and Create Connections—are 0x1d000a and 0x1d0001.

The second argument is a CsCAttrValList object used to pass action-specific data from the client to the model performing the action. Some actions require data in this parameter. For those actions that do not require any custom data, you still must pass a CsCAttrValList object for this argument with a zero-length array, which you can create as follows:

```java
CsCAttrValue[] ava = new CsCAttrValue[0];
CsCAttrValList avl = new CsCAttrValList(ava, CsCError_e.SUCCESS);
```

### The Get Port List Action

All SPECTRUM models whose mtypes derive from the “Device” mtype respond to the “Get Port List” action, which returns a list of all device ports. One way to invoke this action is as follows:

```java
Integer modelID = Integer.decode(args[0]);
CsCModel m = md.getModel(modelID.intValue());
CsCAttrValue[] ava = new CsCAttrValue[0];
CsCAttrValList avl = new CsCAttrValList(ava, CsCError_e.SUCCESS);
avl = m.doAction(0x1d000a, avl);
writer.printActionGetPortListAll(avl);
```

Note that the “Get Port List” action requires no custom data, so you need to pass a CsCAttrValList object with a zero-length array as the second parameter to the doAction method. The action does, however, return a CsCAttrValList object with the requested
information. And, like all actions that do return data, the format of the information in the returned object is very specific to the action. To see how to unpack this returned information for this particular action, refer to the printActionGetPortListAll method in the SSORBWriter class.

The Create Connections Action

The “Create Connections” action is a second example of SPECTRUM actions. Using SNMP requests, this action discovers all port-level connections between the target model and all other devices modeled in SPECTRUM. You can send this action as follows:

```java
1  Integer modelID = Integer.decode(args[0]);
2  CsCModel m = md.getModel(modelID.intValue());
3  CsCAttrValue[] ava = new CsCAttrValue[0];
4  CsCAttrValList avl = new CsCAttrValList(
5      ava,CsCError_e.SUCCESS);
6  avl = m.doAction(0x1d0001,avl);
7  writer.printAll(avl);
```

Summary — From time to time a third-party developer will add an mtype to the SPECTRUM mtype hierarchy. Actions provide a means for such a developer to access functionality specific to the new mtype via a standard interface like SSORB. For more experience dealing with actions, perform Exercise 12 - GetPortList on Page 99 and Exercise 13 - CreateConnections on Page 99.

Filters

The SSORB interface provides several techniques to help you specify exactly the information you want returned in a list. Server-side filtering provides two main benefits. First, it cuts down on network traffic. Second, it provides the client application with the information it needs without extraneous data. This section shows you how to ask SPECTRUM to filter information before sending it to your client application.
Pseudo Filters

The chart below lists several different techniques for limiting the information that SPECTRUM returns in a list. The second and third columns show classes and methods that exemplify each technique, and the last column indicates an example program that uses the technique. Techniques are discussed individually following the chart. (Note that only the last technique listed actually contains the word “filter.”)

<table>
<thead>
<tr>
<th>Filter By</th>
<th>Example Class</th>
<th>Example Method</th>
<th>Example Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>(None)</td>
<td>CsCModelDomain</td>
<td>getAllMTypePropList</td>
<td>GetList</td>
</tr>
<tr>
<td>Method</td>
<td>CsCModelType</td>
<td>getBaseMTypePropList</td>
<td>GetHierarchy</td>
</tr>
<tr>
<td>Name</td>
<td>CsCTypeCatalog</td>
<td>getMTypePropListByName</td>
<td>GetMTypesByName</td>
</tr>
<tr>
<td>Flags</td>
<td>CsCTypeCatalog</td>
<td>getMTypePropListByFlags</td>
<td>GetMTypesByFlags</td>
</tr>
<tr>
<td>IDs</td>
<td>CsCModelType</td>
<td>read DefaultAttrValList</td>
<td>ReadAttrValues</td>
</tr>
<tr>
<td>OID</td>
<td>CsCModelType</td>
<td>getMTypeAttrPropListByOID</td>
<td>GetAttrsByOID</td>
</tr>
<tr>
<td>Assoc</td>
<td>CsCModelDomain</td>
<td>getAssocModelPropList</td>
<td>GetAllAssocs</td>
</tr>
<tr>
<td>Attr Value</td>
<td>CsCModelDomain</td>
<td>getModelPropListByAttrValue</td>
<td>GetModelsByName</td>
</tr>
<tr>
<td>Attr Filter</td>
<td>CsCModelDomain</td>
<td>getModelPropListByAttrFilter</td>
<td>GetDevices</td>
</tr>
</tbody>
</table>

Filter Descriptions

**None** - This row represents the “Get All” methods in the SSORB interface, methods that offer no filtering on the server side. The GetList example program shows four of these methods.

**By Method** - The names of some methods imply that SPECTRUM will do some filtering. The CsCModelType class’s getBaseMTypePropList method is a good example. It limits the returned mtype list to only those mtypes that serve as base classes for the current mtype object.

**By Name** - Some methods use a name argument to limit items in a returned list. The CsCTypeCatalog class’s getMTypePropListByName method returns all mtypes of the specified name. It does not do partial string matching.
**By Flags** - The CsCTypeCatalog class’s getMTypePropListByFlags method returns a list of mtypes that match the specified mtype property flags.

**By IDs** - Several methods take an array of IDs as an argument. For example, the CsCModelType class’s readDefaultAttrValList method takes an array of attribute IDs as a parameter and returns the default value of each attribute.

**By OID** - Sometimes you know the SPECTRUM attribute ID and you want to find out the associated MIB object ID. You can do this using the GetMTypeAttrProp example program, which calls the CsCModelType class’s getMTypeAttrProperties method to display the “OID” mtype attribute property.

Conversely, you may know an OID and want to find out (for a given mtype) all attributes that point to this OID. In this case, you would use the CsCModelType class’s getMTypeAttrPropListByOID method as shown in the table. You can try this using the OID 1.3.6.1.2.1.2.2.1.6 and the mtype 0x100c9.

**By Assoc** - Given a model, SPECTRUM can generate a list of associated models.

**By Attr Value** - You can use this technique to get a list of all SPECTRUM models that have a certain attribute set to a particular value. The GetModelsByName example program mentioned in the chart above uses this technique to get a list of models whose “Model Name” attribute (0x1006e) is set to the specified string.

**By Attr Filter** - This is the only technique considered a true SPECTRUM filter. The rest of this section explains this in more detail.

### A Simple Attribute Filter

A SPECTRUM attribute filter (CsCValue[]) enables you to define SPECTRUM database search instructions that range from very simple to fairly complex. Suppose, for example, you want to obtain a list of all models whose mtypes derive from the Device mtype. You would build (and display) the filter as follows:

```
1  CsCorbaFilterAttrNode node
2     CsCorbaAttrFilterHelper.createModelTypeIDNode(
3         SSORBHelper.attrIDMType,
4         CsCOperator_e.CSC_IS_DERIVED_FROM,
5         SSORBHelper.mtypeIDDevice);
6  CsCValue[] filter = node.getFilter();
7  writer.printFilterAll(filter);
```
Central to this example is the SPECTRUM CsCorbaAttrFilterHelper class (line 2), which contains several methods that enable you to build different kinds of filters, including the createModelTypeIDNode method used in the example which takes three parameters: an attribute ID, an operator, and an mtype ID.

Once you have created the node (lines 1 - 5), you call the node's getFilter method to create the actual filter (line 6), which is an array of the following three CsCValue objects:

1. CSC_OPERATOR = IS_DERIVED_FROM
2. CSC_ATTR_VALUE = 0x10001
3. CSC_MODEL_TYPE_ID = 0x1004b

A simple attribute filter always contains an operator followed by an attribute ID and a value. Line 7 displays the filter.

A Double Attribute Filter

Suppose you want to see all models whose mtypes derive from the “Device” mtype except those models of type “Indirect RMON.” You would build the filter as follows:

```java
1  CSC_OPERATOR = IS_DERIVED_FROM
2  CSC_ATTR_VALUE = 0x10001
3  CSC_MODEL_TYPE_ID = 0x1004b
```

A Double Attribute Filter

Suppose you want to see all models whose mtypes derive from the “Device” mtype except those models of type “Indirect RMON.” You would build the filter as follows:

```java
1  CsCorbaFilterAttrNode node1 =
2      CsCorbaAttrFilterHelper.createModelTypeIDNode(
3          SSORBHelper.attrIDMType,
4          CsCOperator_e.CSC_IS_DERIVED_FROM,
5          SSORBHelper.mtypeIDDevice);
6
7  CsCorbaFilterAttrNode node2 =
8      CsCorbaAttrFilterHelper.createModelTypeIDNode(
9          SSORBHelper.attrIDMType,
10         CsCOperator_e.CSC_DOES_NOT_EQUAL,
11         SSORBHelper.mtypeIDIndirectRMON);
12
13 CsCorbaFilterBinaryNode bnode =
14      CsCorbaAttrFilterHelper.createBinaryNode(
15          node1, CsCOperator_e.CSC_AND, node2);
16
17 CsCValue[] filter = bnode.getFilter();
18 writer.printFilterAll(filter);
```
This example creates three nodes. The first, \texttt{node1}, represents all models whose mtypes derive from Device. The second, \texttt{node2}, specifies that the mtype must not equal Indirect RMON. The third node, \texttt{bnode} (binary node) is the logical and of the first two.

Line 17 extracts the filter, consisting of an array of seven \texttt{CsCValue} objects, from the \texttt{bnode} object, as follows:

```java
1  CSC_OPERATOR      = AND
2  CSC_OPERATOR      = IS_DERIVED_FROM
3  CSC_ATTR_VALUE    = 0x10001
4  CSC_MODEL_TYPE_ID = 0x1004b
5  CSC_OPERATOR      = DOES_NOT_EQUAL
6  CSC_ATTR_VALUE    = 0x10001
7  CSC_MODEL_TYPE_ID = 0x59001b
```

Note that the logical and appears in the first \texttt{CsCValue} object followed by two triplets representing \texttt{node1} and \texttt{node2} respectively.

\section*{A Complex Attribute Filter}

Finally, suppose you want to augment the filter created in the last section with one more restriction, namely that all models in the list must have a “Model Class” attribute value of “2.” This attribute specifies whether the model is a bridge, switch, router, brouter, etc. The value “2” specifies “switch.” You would build this filter as follows:

```java
1  CsCorbaFilterAttrNode node1 =
2    CsCorbaAttrFilterHelper.createModelTypeIDNode(  
3      SSORBHelper.attrIDMType,  
4      CsCOperator_e.CSC_IS_DERIVED_FROM,  
5      SSORBHelper.mtypeIDDevice);
6
7  CsCorbaFilterAttrNode node2 =
8    CsCorbaAttrFilterHelper.createModelTypeIDNode(  
9      SSORBHelper.attrIDMType,  
10     CsCOperator_e.CSC_DOES_NOT_EQUAL,  
11     SSORBHelper.mtypeIDIndirectRMON);
12
13 CsCorbaFilterBinaryNode bnode =
14    CsCorbaAttrFilterHelper.createBinaryNode(  
15      node1,CsCOperator_e.CSC_AND,node2);
16
17 CsCValue modelClass = new CsCValue();
```
Lines 1-15 are identical to those in the previous section. At this point in the code, the binary node \texttt{bnode} contains the information from \texttt{node1} and \texttt{node2} plus the logical and operator. Lines 17 - 23 create a new node specifying “Model Class” equals “2,” reusing the available \texttt{node2}. Lines 25 - 26 combine \texttt{bnode} and \texttt{node2} and store the result in \texttt{bnode} again, overwriting the previous information stored there. Line 28 extracts the \texttt{CsCValue[]} array from \texttt{bnode}. Here is the content of the array:

1. \texttt{CSC_OPERATOR AND}
2. \texttt{CSC_OPERATOR AND}
3. \texttt{CSC_OPERATOR IS_DERIVED_FROM}
4. \texttt{CSC_ATTR_VALUE 0x10001}
5. \texttt{CSC_MODEL_TYPE_ID 0x1004b}
6. \texttt{CSC_OPERATOR DOES_NOT_EQUAL}
7. \texttt{CSC_ATTR_VALUE 0x10001}
8. \texttt{CSC_MODEL_TYPE_ID 0x59001b}
9. \texttt{CSC_OPERATOR EQUALS}
10. \texttt{CSC_ATTR_VALUE 0x11ee8}
11. \texttt{CSC_INTEGER 2}

Lines 3 - 5 represent one node. Lines 6 - 8 represent the second. Line 2 represents the logical and between these two nodes. In effect, line 2 makes lines 3 - 8 into a single node. Lines 9 - 12 represent the third node. Line 1 represents the logical and between this node and the single node of lines 3 - 8.

**Summary** — Several SSORB methods allow you to specify the kinds of items you want returned in a list. Of all the techniques mentioned in this chapter, passing an attribute filter (\texttt{CsCValue[]}) is probably the most flexible and powerful. To hone your filtering skills, perform Exercise 14 - GetDevices on Page 100.
Watches

This section demonstrates how to register with SPECTRUM for asynchronous notification of certain SPECTRUM occurrences, including model creation and deletion, attribute value changes, event and statistic creation, and alarm updates.

As discussed earlier, an SSORB client application obtains references to objects that actually reside on the server, and then uses these stubs to make remote method invocations to the server-side objects. When registering for asynchronous notification, client applications do just the opposite—the client itself instantiates a watch object and then passes a reference to the server with the instruction that the server should call a method in the object when certain conditions arise. Then, when the specified condition occurs, the server calls the method in the client's watch object, thus notifying the client of the occurrence.

For the purpose of the following example, assume you are a software engineer who wants to write a client application that will be notified when the value of a particular attribute in the SPECTRUM knowledge base changes. To do this, you would first design an “attribute value change” class, which you might call, for example, AttrValCallback. The client program would instantiate the class and then pass an object reference (and the ID of the attribute to watch) to SPECTRUM. Then, when the attribute value changed, SPECTRUM would call the attrValsChanged method in your AttrValCallback class, which might be defined as follows:

```java
1  public class AttrValCallback
2      extends CsCAttrValWatchCBPOA
3  {
4      SSORBWriter writer = null;
5
6      public AttrValCallback()
7      {
8          writer = new SSORBWriter();
9      }
10
11     public AttrValCallback(String filename)
12         throws FileNotFoundException, IOException
13     {
14         writer = new SSORBWriter(filename);
15     }
16
17     public void attrValsChanged(
```
Line 2 shows that the AttrValCallback class derives from CsCAttrValWatchCBPOA, a class defined in an SSORB IDL file and generated automatically by the CORBA compiler. SPECTRUM is familiar with this base class and knows how to call the attrValsChanged method, which is contained in the base class or in any classes derived from the base class. What the method does is up to you as the designer of the derived “watch” class. In this particular case, the attrValsChanged method outputs data about changed attribute values to the SSORBWriter (line 8), which may be pointing to the display or to a file, depending on the constructor used.

The following example shows how you might use your AttrValCallback class in a client program:

```java
1 Integer modelID = Integer.decode(args[0]);
2 Integer attrID = Integer.decode(args[1]);
3 CsCModel m = md.getModel(modelID.intValue());
4 int[] ia = new int[]{attrID.intValue()};
5 AttrValCallback callback = new AttrValCallback();
6 CsCAttrValWatchCB cb = CsCAttrValWatchCBHelper.narrow
   (helper.servant_to_reference(callback));
7 CsCAttrValList avl = m.startWatchAttrVals(ia, cb);
8 writer.println(avl);
9 System.out.println("Press Enter to exit");
10 System.in.read();
11 m.stopWatchAttrVals(cb);
```

Lines 1 and 2 get a model ID and an attribute ID from the command line of the program. Line 3 gets the model's object reference from the model domain. Line 4 puts the attribute ID into an `int[]` array. Line 5 constructs the `callback` object. Line 6 "narrows" or casts `callback` object to a `CsCAttrValList`. Line 7 invokes the `CsCModel` class’s `startWatchAttrVals` method to pass the `int[]` and the `cb` object reference to the server. Line 8 outputs the return value. Line 9 outputs a line to the user. Line 10 puts the program to sleep pending user input.
Note also that if the program is killed before the stopWatchAttrVals method (Line 11) is called, the SpectroSERVER will automatically remove any reference to the watch so that the program will be able to operate properly when restarted.

At this point, the user thread is suspended, and the SpectroSERVER has the callback object and the list of attributes (containing a single attribute ID) to watch. When the attribute value changes, the SpectroSERVER will invoke the callback object's attrValsChanged method, which will run in a separate thread on the client. It is important to note that the SpectroSERVER calls the client's method via a CORBA one-way invocation, so the server thread does not block waiting for the attrValsChanged method to return. The SpectroSERVER can invoke this client method repeatedly and indefinitely while the program thread sleeps (as in this case) or processes other code. Note also that if the program is killed before the stopWatchAttrVals method (Line 10) is called, the SpectroSERVER will automatically remove any reference to the watch so that the program will be able to operate properly when restarted.

This example put only a single attribute ID into the int[] array that is passed to the server, but you could also load the array with several attribute IDs, in which case the server would invoke the callback method whenever any one of the attribute values changed for the specified model.

### Callback Base Classes

As you can see from the example in the previous section, to register for asynchronous notification of SPECTRUM occurrences, you must write your own “watch” class, which derives from a base class.

The following table lists each of these base classes along with the name of an example program demonstrating its use.

<table>
<thead>
<tr>
<th>Base Class</th>
<th>Example Program</th>
</tr>
</thead>
<tbody>
<tr>
<td>CsCAlarmWatchCBPOA</td>
<td>AlarmCallback.java</td>
</tr>
<tr>
<td>CsCAtrrValWatchCBPOA</td>
<td>AttrValCallback.java</td>
</tr>
<tr>
<td>CsCEventWatchCBPOA</td>
<td>EventCallback.java</td>
</tr>
<tr>
<td>CsCModelWatchCBPOA</td>
<td>ModelCallback.java</td>
</tr>
</tbody>
</table>
Summary — In this section you learned how to create an SSORB client program that receives notification from SPECTRUM when a specified attribute value changes. Client programs that watch for model creations and deletions, event and statistics creation, alarm updates, etc., work in a similar fashion. For practice working with SPECTRUM watches, perform Exercise 15 - WatchAttrValue on Page 101 and Exercise 16 - WatchModel on Page 101.

Events and Alarms

In addition to the SPECTRUM platform, network managers often rely on one or more third-party products to manage special aspects of their networks. For example, one third-party product might monitor firewalls for breaches of security, while another, might watch network traffic. These types of products gather information, filter the data, and send events to SPECTRUM, which can correlate, store, and display the events, and promote them to alarms. This section demonstrates how to use the SSORB interface to create and manage SPECTRUM events and alarms.

Creating an Event

Building an event and sending it to SPECTRUM is a relatively easy process, as shown in the following example:

```java
1  Integer modelID = Integer.decode(args[0]);
2  CsCModel m = md.getModel(modelID.intValue());
3  CsCValue mvalue = new CsCValue();
4  mvalue.model(m);
5  CsCAttrValue mav = new CsCAttrValue(
6      SSORBHelper.attrIDModel,
7      mvalue,
8      CsCError_e.SUCCESS);
9
10 CsCAttrValue[] ava = new CsCAttrValue[] {mav};
11 CsCAttrValList avl = new CsCAttrValList(
12    ava, CsCError_e.SUCCESS);
13
14 byte[] eventID = new byte[0];
15 Integer eventCode = Integer.decode(args[1]);
16
17 CsCEvent event = new CsCEvent(
18    eventID,
```
eventCode.intValue(),
avl,
CsCError_e.SUCCESS);

CsCEventDomain ed = md.getEventDomain();
eventID = ed.createEvent(event);
writer.printEventIDAll(eventID);

Lines 23 and 24 are the key parts of this code. Line 23 obtains an event domain object reference from SPECTRUM, and line 24 invokes the createEvent method that sends a new event to the server.

Lines 17 to 21 build the event passed to the create event method. Line 17 constructs a CsCEvent object which takes an event ID, an event code, an attribute value list, and an error code. This constructor was not man-made. Rather, it was generated by a CORBA compiler, which created an argument in the constructor for each member variable in the class. That is why, even though it may seem a bit odd, you must pass a zero-length byte[] to the constructor for the first parameter (eventID).

The second parameter in the constructor is the event code (aka “event type”). Every SPECTRUM mtype has a set of event codes to which models of that mtype will react.

These event codes are defined in an event disposition (EventDisp) file corresponding to the mtype and found in the appropriate subdirectory of the <$SPECROOT>/SS/CsVendor directory. The EventDisp file for the mtype 0x1c8000a, for example, resides in the <$SPECROOT>/SS/CsVendor/Ctron_SSAH directory and looks like this:

```
0x00010306 E 30
0x00010307 E 30
0x00010308 E 50 A 1,0x010308
0x00010309 E C 0x010308
0x0001030a E 50 A 1,0x01030a
0x0001030b E 50 A 1,0x01030b
0x021b070a E 50
0x021b070b E 50 A 1,0x021b070b
0x021b070c E 50
0x021b070d E 50
0x000d01a0 E 50 A 1,0x0d01a0
0x000d01a1 E 50 A 1,0x0d01a1
0x000d01a2 E 50 A 1,0x0d01a2
```
Each of the numbers in the left-hand column are event codes. The letter “E” tells SPECTRUM to log the event. The “C” tells SPECTRUM to clear the specified event. The “A” tells SPECTRUM to generate an alarm. The “1” specifies the severity (or condition) or the alarm. The numbers in the right-hand column specify the probable cause string to display with the alarm.

The third parameter in the constructor is a CsCAttrValList containing an array of CsCAttrValue objects that define the event. The only required member of this array is the object specifying the target model (lines 1 - 12). You can also specify the following:

<table>
<thead>
<tr>
<th>Data</th>
<th>CsCValue Method</th>
<th>Attribute ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Creator</td>
<td>textString</td>
<td>0x11fb9</td>
</tr>
<tr>
<td>Creation Date/time</td>
<td>dateTime</td>
<td>0x11f4e</td>
</tr>
<tr>
<td>Model</td>
<td>model</td>
<td>0x11f53</td>
</tr>
<tr>
<td>Severity</td>
<td>gauge</td>
<td>0x11fb5</td>
</tr>
<tr>
<td>Custom</td>
<td>***</td>
<td>***</td>
</tr>
</tbody>
</table>

The fourth parameter in the constructor is the error code (CsCError_e.SUCCESS), which initializes the object's error code member variable.

One of the exercises at the end of this chapter directs you to write an example program that creates events for a particular network device on your network. You can find the event
codes for the device in the associated EventDisp file, which is located in SS/CsVendor in a directory named for the vendor who created the file. [If the event codes are specific to a particular model type, they will be located in a model type subdirectory within the vendor-specific directory. For more information, refer to the Event Configuration Files Guide (9035070)]. Be sure to use a few event codes (from the EventDisp file) that have an “A” next to them in order to cause some alarms, which you can then view using the Alarm Manager option on your SpectroGRAPH Tools menu. This will help you see the connection between events and alarms.

**Getting an Event**

In the example code snippet in the previous section, the createEvent method on line 24 takes a CsCEvent object and returns an event ID. The printEventIDAll method on the next line displays the event ID, which uniquely identifies the event among all the event domains in the installation.

Here is an example event ID:

3a63543c-0026-1000-00d9-0080108d4051

Conversely, you can use an event ID to obtain the corresponding CsCEvent object from SPECTRUM, as in the following example:

```java
1  CsCorbaValueHelper helper =
2      new CsCorbaValueHelper();
3  byte[] eventID = helper.parseEventID(args[0]);
4
5  byte[][] events = new byte[1][1];
6  events[0] = eventID;
7
8  int[] attrIDs = new int[7];
9  attrIDs[0] = SSORBHelper.attrIDCreator;
10 attrIDs[1] = SSORBHelper.attrIDDateTime;
11 attrIDs[2] = SSORBHelper.attrIDModel;
12 attrIDs[3] = SSORBHelper.attrIDModelName;
13 attrIDs[4] = SSORBHelper.attrIDMType;
14 attrIDs[5] = SSORBHelper.attrIDMTypeName;
15 attrIDs[6] = SSORBHelper.attrIDSeverity;
16
17 CsCEventDomain ed = md.getEventDomain();
18 CsCEventList eventList = ed.getEventList(
19    events,attrIDs);
```
20 writer.printAll(eventList);

The program containing this code snippet takes an event ID on the command line in string form. Lines 1 - 3 convert the event ID string to byte[] form using the CORBA value helper class. Lines 5 - 6 put the event ID into an array. Lines 8 - 15 create an int[] array and insert several attribute IDs specifying the information that will be returned about the event. In this case, the information will include: who created the event, when the event was created, the model ID associated with the event, the model name, the mtype ID and name, and the severity of the event. Finally, lines 17 - 20 get the event domain, obtain an event list (containing an array of CsCEvent objects), and display the list.

Watching for Event Creation

The earlier section on Watches (Page 82) explained how to register for asynchronous notification of SPECTRUM occurrences like model creation and deletion, attribute value modification, etc. You can also watch for event creation using the same techniques. Here is one example of a callback class whose object reference a client application might pass to SPECTRUM in order to watch for (and receive notification about) the creation of events:

1  public class AttrValCallback
2      extends CsCAttrValWatchCBPOA
3  {
4      SSORBWriter writer = null;
5  
6      public EventCallback()
7      {
8          writer = new SSORBWriter();
9      }
10 
11      public EventCallback(String filename)
12          throws FileNotFoundException, IOException
13      {
14          writer = new SSORBWriter(filename);
15      }
16 
17      public void eventsCreated(CsCEventList list)
18      {
19          writer.printAll(list);
20      }
public void eventAttrValsUpdated(CsCEventList list) {
    writer.printAll(list);
}

Lines 1 and 2 indicate that this callback class must derive from _CsCEventWatchCBImplBase. The two constructors in lines 6 and 11 initialize the SSORBWriter object to output information to the screen or to a file respectively. Whenever events occur that meet criteria defined in a filter passed from the client to SPECTRUM when the client registers this watch, SPECTRUM will call the eventsCreated method in line 17.

The following example shows how you might use the EventCallback class in a client program:

```
Integer modelID = Integer.decode(args[0]);
CsCorbaFilterAttrNode node =
    CsCorbaAttrFilterHelper.createModelIDNode(
        SSORBHelper.attrIDModel,
        CsCOperator_e.CSC_EQUALS,
        modelID.intValue());

int[] attrIDs = new int[7];
attrIDs[0] = SSORBHelper.attrIDCreator;
attrIDs[1] = SSORBHelper.attrIDDateTime;
attrIDs[2] = SSORBHelper.attrIDModel;
attrIDs[3] = SSORBHelper.attrIDModelName;
attrIDs[4] = SSORBHelper.attrIDMType;
attrIDs[5] = SSORBHelper.attrIDTypeName;
attrIDs[6] = SSORBHelper.attrIDSeverity;

EventCallback callback = new EventCallback();

CsCEventDomain ed = md.getEventDomain();
ed.startWatchEvents(filter, attrIDs, callback);
System.out.println("Press Enter to exit");
```
The key part of this code snippet is line 22, where the client program asks SPECTRUM to send notification when new events are created. Note that the CsCEventDomain class’s startWatchEvents method takes three parameters: filter, attrIDs, and callback. The filter parameter lets you specify the set of models upon which SPECTRUM will watch for events; this could be a particular model, a set of models associated with the or logical operator, an mtype, etc. The attrIDs parameter governs what information (about each event) SPECTRUM will send to the callback object. The callback parameter specifies the callback object (i.e., it tells SPECTRUM whom to call).

As of SPECTRUM 6.6 Service Pack 4, the startWatchEvents method is used only to monitor the creation of new events and no longer returns historical events when called.

**Watching for Alarm Creation**

In addition to watching for event creation, a client application can also watch for alarm creation. Here is an example of a definition for an alarm callback class:

```java
1  public class AlarmCallback
2      extends CsCAlarmWatchCBPOA
3  {
4      SSORBWriter writer = null;
5
6      public AlarmCallback()
7      {
8          writer = new SSORBWriter();
9      }
10
11     public AlarmCallback(String filename)
12         throws FileNotFoundException, IOException
13     {
14         writer = new SSORBWriter(filename);
15     }
16
17     public void alarmsUpdated(
```
As you can see, the AlarmCallback class is very similar to the EventCallback class described in the previous section. The only significant differences are that it derives from a different base class, and that it defines a different method (alarmsUpdated) by which SPECTRUM can perform asynchronous notification. The following example shows how you might use the AlarmCallback class in a client program:

```java
CsCorbaFilterAttrNode node =
    CsCorbaAttrFilterHelper.createModelIDNode(
        SSORBHelper.attrIDModel,
        CsCOperator_e.CSC_EQUALS,
        modelID.intValue());

CsCValue[] filter = node.getFilter();
int[] attrIDs = new int[0];
AlarmCallback callback = new AlarmCallback();

CsCAlarmDomain ad = md.getAlarmDomain();
ad.startWatchAlarms(filter, attrIDs, callback);
System.out.println("Press Enter to exit");
System.in.read();
ad.stopWatchAlarms(callback);
```

The key part of this program is line 12, where the program registers for notification of alarm creation by calling the CsCAlarmDomain class’s startWatchAlarms method and passing the filter, attrIDs, and callback parameters. (See the previous section, Watching for Event Creation, for a description of these parameters.)

## Getting Alarms

To obtain a list of alarms, you would use a straightforward (synchronous) version of the example in the previous section, as follows:

```java
CsCorbaFilterAttrNode node =
    CsCorbaAttrFilterHelper.createModelIDNode(
        SSORBHelper.attrIDModel,
        CsCOperator_e.CSC_EQUALS,
        modelID.intValue());
```
SSORBHelper.attrIDModel,
CsCOperator_e.CSC_EQUALS,
modelID.intValue());

CsCValue[] filter = node.getFilter();

int[] attrIDs = new int[7];
attrIDs[0] = SSORBHelper.attrIDCreator;
attrIDs[1] = SSORBHelper.attrIDDateTime;
attrIDs[2] = SSORBHelper.attrIDModel;
attrIDs[3] = SSORBHelper.attrIDModelName;
attrIDs[4] = SSORBHelper.attrIDMType;
attrIDs[5] = SSORBHelper.attrIDMTypeName;
attrIDs[6] = SSORBHelper.attrIDSeverity;

CsCAlarmDomain ad = md.getAlarmDomain();
CsCAlarmList al = ad.getAlarmListByAttrFilter(filter,attrIDs);
writer.printAll(al);

Line 19 represents the only difference between this synchronous technique of obtaining a list of alarms and the asynchronous one demonstrated in the last section.

### Clearing Alarms

The SSORB interface also enables you to clear alarms. Here is an example of how you might do this:

```java
CsCorbaValueHelper helper = new CsCorbaValueHelper();
byte[] alarmID = helper.parseAlarmID(args[0]);
CsCAlarmDomain ad = md.getAlarmDomain();
ad.clearAlarm(alarmID);
```

The program takes an alarm ID on the command line, parses it into a `byte[]` array, and passes the result to an alarm domain method for processing.

**Summary** — This section explained how to create events, get events, get alarms, register for notification regarding event and alarm creations, and clear alarms. The following exercises can be used to review this information:
• Exercise 17 - CreateEvent on Page 101
• Exercise 19 - WatchEvent on Page 102
• Exercise 20 - WatchAlarm on Page 102
• Exercise 21 - GetAlarms on Page 102
• Exercise 22 - ClearAlarm on Page 103
Debugging

The following example illustrates how to turn on detailed debugging for the client application.

Properties props = new Properties();
props.put("ORBwarn", "2");
CORBAHelper helper = CORBAHelper.getHelperImpl();
helper.init(null, props);
CsCModelDomain md = (CsCModelDomain)
helper.getObjectImplementation
    (CsCModelDomain.class, "myWorkstation");
Exercises

This section provides exercises to clarify and help you master the concepts and techniques presented by the example programs in the previous section.

Sample files for the exercises in this section are located in the <$SPECROOT>/SDK/examples/SSORB/com/aprisma/spectrum/core/examples/cmdline directory. If do not have a “work” directory yet, you should create one and make sure your CLASSPATH environment variable is set up as described under Environment Checklist (Page 36).

Exercise 1 - GetDomainID

In your work directory, create, compile, and run a command-line SSORB client program called GetDomainID.java that displays the model domain ID of the specified model domain name. Here is the command line syntax:

```
GetDomainID domainName
```

The domainName parameter is the name of the target SpectroSERVER.

The GetDomainID program will serve as a template for most of the other exercise programs you write while performing the exercises in this section, so take your time and add comments that are meaningful to you.

Exercise 2 - GetMTypes

In your work directory, create, compile, and run a command-line SSORB client program called GetMTypes.java that displays all mtypes on your screen. Here is the command line syntax:

```
GetMTypes domainName
```
Exercise 3 - GetRelations

In your work directory, create, compile, and run a command-line SSORB client program called GetRelations.java that displays all relations on your screen. Here is the command line syntax:

    GetRelations domainName

Exercise 4 - GetList

In your work directory, create, compile, and run a command-line SSORB client program called GetList.java that displays one of four types of lists on your screen. Here is the command line syntax:

    GetList listType domainName

The listType parameter should be one of the following strings:

- models
- mtypes
- relations
- developers

Depending on the input, the program should output the appropriate list.

Exercise 5 - InspectMType

In your work directory, create, compile, and run a command-line SSORB client program called InspectMType.java that displays an mtype's properties, attributes, or hierarchy on your screen. Here is the command line syntax:

    InspectMType mtypeID infoType domainName

The infoType parameter should be one of the following strings:

- properties
- attributes
- hierarchy
Depending on the input, the program should output the appropriate data.

**Exercise 6 - GetAttrProp**

In your work directory, create, compile, and run a command-line SSORB client program called `GetAttrProp.java` that displays the properties of the specified attribute ID on your screen. Here is the command line syntax:

```
GetAttrProp attrID domainName
```

**Exercise 7 - GetMTypeAttrProp**

In your work directory, create, compile, and run a command-line SSORB client program called `GetMTypeAttrProp.java` that displays the properties of the specified attribute ID for the specified mtype ID on your screen. Here is the command line syntax:

```
GetMTypeAttrProp mtypeID attrID domainName
```

**Exercise 8 - CreateAndCollectNetwork**

In your work directory, create, compile, and run a program called `CreateAndCollectNetwork.java` that creates a network model (mtypeID = 0x1002e) with the specified model name, collects the new model in the Universe view, and displays the new model's model ID. Here is the syntax:

```
CreateAndCollectNetwork modelName domainName
```

**Exercise 9 - CreateModelByIP**

In your work directory, create, compile, and run a program called `CreateModelByIP.java` that directs SPECTRUM to create a set of models that represent some device at the specified IP address. The program should also collect the principle model into the specified container.

Here is the command line syntax:

```
CreateModelByIP ipAddress collectorModelID community domainName
```
Exercise 10 - ReadAttrValue

In your work directory, create, compile, and run a program called ReadAttrValue.java that reads and displays an attribute value of the specified model. Here is the command line syntax:

ReadAttrValue modelID attrID domainName

Exercise 11 - WriteAttrValue

In your work directory, create, compile, and run a program called WriteAttrValue.java that writes the specified value to the specified model's attribute. Here is the command line syntax:

WriteAttrValue modelID attrID value domainName

Exercise 12 - GetPortList

In your work directory, create, compile, and run a program called GetPortList.java that gets and displays all the ports of the specified device model ID. Here is the command line syntax:

GetPortList modelID domainName

Exercise 13 - CreateConnections

In your work directory, create, compile, and run a program called CreateConnections.java that discovers and creates connections between the ports of the specified model and all other device models (modeled in SPECTRUM) to which the specified model is connected. Here is the command line syntax:

CreateConnections modelID domainName

Be sure that you have two or more models (that are connected in real life) created in your SPECTRUM database before running your program.
Exercise 14 - GetDevices

In your work directory, create, compile, and run a program called `GetDevices.java` that gets and displays all SPECTRUM models of model types derived from 0x1004b except the following model types:

- 0x59001b
- 0x2640001
- 0x22f0000
- 0xd80023
- 0xd80033
- 0x102e3
- 0x100ae
- 0x103d8
- 0x10322

The command line syntax looks like this:

    GetDevices domainName

Once you confirm that this is working, modify the command line syntax to match the following:

    GetDevices categoryID domainName

The `categoryID` is a number (0..6). The meaning of each number is described in the following list:

1. Unknown
2. Other
3. Bridge
4. Router
5. Brouter
6. Hub
7. Switch

Now modify the filter you are using to include the additional filtering information as follows:

    CsCValue categoryID = new CsCValue();
categoryID.intValue(cid);
node2 = CsCorbaAttrFilterHelper.createAttrValNode(
    SSORBHelper.attrIDModelClass,
    CsCOperator_e.CSC_EQUALS,
    categoryID);
bnode = CsCorbaAttrFilterHelper.createBinaryNode(
    bnode,CsCOperator_e.CSC_AND,node2);

**Exercise 15 - WatchAttrValue**

In your work directory, create, compile, and run a program called WatchAttrValue.java that registers with SPECTRUM to be notified when the specified attribute value (of the specified model) changes. Here is the command line syntax:

```
WatchAttrValue modelID attrID domainName
```

**Exercise 16 - WatchModel**

In your work directory, create, compile, and run a program called WatchModel.java that registers with SPECTRUM to be notified when a model (of the specified mtype) is created or deleted in SPECTRUM. Here is the command line syntax:

```
WatchModel mtypeID domainName
```

**Exercise 17 - CreateEvent**

In your work directory, create, compile, and run a program called CreateEvent.java that creates an event with the specified eventCode for the specified model.

Here is the command line syntax:

```
CreateEvent modelID eventCode domainName
```
Exercise 18 - GetEvent

In your work directory, create, compile, and run a program called GetEvent.java that gets and displays event information for the specified event ID. Here is the command line syntax:

GetEvent eventID domainName

Exercise 19 - WatchEvent

In your work directory, create, compile, and run a program called WatchEvent.java that registers with SPECTRUM to be notified when an event is created for the specified model. Here is the command line syntax:

WatchEvent modelID domainName

Exercise 20 - WatchAlarm

In your work directory, create, compile, and run a program called WatchAlarm.java that registers with SPECTRUM to be notified when an alarm is created for the specified model. Here is the command line syntax:

WatchAlarm modelID domainName

Exercise 21 - GetAlarms

In your work directory, create, compile, and run a program called GetAlarms.java that gets and displays all alarms for the specified model. Here is the command line syntax:

GetAlarms modelID domainName

Exercise 22 - ClearAlarm

In your work directory, create, compile, and run a program called ClearAlarm.java that clears the specified alarm. Here is the command line syntax:

ClearAlarm alarmID domainName