Contents

SMC SDK
Programming Guide
for Solaris 8™

Preface Intro Start Tools Toolboxes

Services Registration FAQ Samples Glossary

Index

Contents

Registration

- *Overview*
- *smcconf*

Frequently Asked Questions

Code Samples

Illustrations

Glossary

Copyrights

Copyrights

N'EST ACCORDEE, Y COMPRIS DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE
DE LA PUBLICATION A REPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ELLE NE SOIT
PAS CONTREFAISANTE DE PRODUIT DE TIERS. CE DENI DE GARANTIE NE S'APPLIQUERAIT PAS, DANS
LA MESURE OU IL SERAIT TENU JURIDIQUEMENT NUL ET NON AVENU.

Preface

About This Guide

This guide provides instructions for using the Solaris™ Management Console 2.0 Software Development Kit (SMC SDK) to create and port tools and services based on the SMC distributed application environment. This guide also provides a general overview of the SMC architecture and JavaBeans™ design considerations as they apply to the SMC SDK.

▌▷ *This guide describes the SMC SDK for the **Solaris 8** Operating Environment. There are subtle but important differences between the Solaris 8 SMC SDK and the SMC SDK for the Solaris 9 Operating Environment.*

Top of Page

Who Should Read This?

This guide is intended for programmers who want to create or port applications for the SMC environment. Readers of this guide should be proficient with Java, JavaBeans, and general object-oriented programming techniques.

Please use this guide in conjunction with the SMC javadocs (/usr/sadm/lib/smc/docs/javadoc/index.html) as well the smc(1M) and smcconf(1M) man pages.

Top of Page

How This Guide is Organized

This guide is organized into the general sections listed below, followed by a glossary and a list of illustrations. All sections in the guide can be reached from links in the navigation pane on the left, which can be toggled between Index and TOC views.

Introduction	General introduction to the uses and features of the SMC SDK
Getting Started	Overview of SMC architecture, tools, services, and infrastructure; includes high-level explanations of procedures for creating and porting tools and services with the SMC SDK

Tools	In-depth instructions on how to build and package a Tool, from a simple CLI interface to a more complex GUI, console. Also includes an overview of some of the more important user interface components included in the SDK, with instructions on how and when to use them
Toolboxes	Overview of what a toolbox is, and how to manage them
Services	In-depth instructions on how to build and package a Service
Registration	In-depth instructions on how to use *smcconf* for registering Tools and Services
Frequently Asked Questions	General FAQ for the SMC SDK
Code Samples	Compiled list of sample code used in this guide
Illustrations	Compiled list of illustrations used in this guide
Glossary	Glossary of terms relevant to SMC

While it is not necessary to read the sections in any particular order, you should be familiar with the concepts in the Introduction and Getting Started sections before starting to use the SMC SDK.

Typographic Conventions

- File names, commands, environment variables, class names and methods, and field values are displayed in a `fixed width` font.

- Links to glossary terms are in indicated by small book icons in the main text. For example, Sun Management Center. Use the Back button in your browser to return to the main text.

- Code samples are displayed in separate windows alongside the main text. Code samples can be displayed by clicking the *Sample Code* boxes; for example:

Sample Code	Hello

This guide is also available in a printer-friendly PDF version!

Introduction

What is the the SMC SDK? ~ SMC SDK Components ~ Features and Benefits of the SMC SDK ~ SMC SDK Contents

What is the SMC SDK?

The SMC (Solaris Management Console) SDK is a software development kit designed to give developers a platform on which to develop and deploy distributed applications. For example, Sun Microsystems, Inc. is using the SMC SDK to develop Solaris system management applications which plug into the Solaris*TM* Management Console™ 2.0.

A distinguishing feature of the SMC SDK is its ability to present to the end user a unified console consisting of user interface components that may have been built using several different development platforms and middleware services. For example, a user interface created with the SMC SDK might combine a simple disk management tool with a WBEM-based user management tool, both of which would appear in the same *console* on the user's desktop.

*This version of the SMC SDK Programming Guide is for the SMC SDK for the **Solaris 8** Operating Environment. There are subtle but important differences between the Solaris 8 SMC SDK and the SMC SDK for the Solaris 9 Operating Environment.*

SMC SDK Components

The SMC SDK environment comprises five general components:

Component	Description
Tools	Client-side applications; in SMC, all tools are written as sets of JavaBeans
Consoles	A container for SMC client tools; the SMC "desktop" from which users perform management tasks
Services	Server-side applications that support SMC tools; native SMC services are generally a combination of Java and platform-specific code
Look and Feel	The presentation layer used in a console; in SMC, "Look and Feel" is a pluggable component, and you can use whatever look and feel -- including a command-line interface -- that is most appropriate for your tools and customers

Infrastructure	The "glue" that holds everything together; the SMC infrastructure includes a set of core services and an 📖RMI-based communication model, although SMC tools and services can also be implemented using other infrastructures, such as 📖CIM/📖WBEM, and 📖SunMC.

See the Getting Started section for a more complete description of the SMC architecture.

Features and Benefits of the SMC SDK

The SMC SDK provides several important features and benefits:

- **Common platform for all your tools** -- Perhaps the biggest benefit the SMC SDK provides is a common platform for deploying end user components. These components can all share a common user model, and look and behave in the same manner, regardless of the back end services the components might use.

- **Convenient core services** --The SMC SDK also provides a convenient middleware platform with a set of core services such as authorization, logging, messaging, and others. See Bundled Common Services for more information about the SMC SDK core services.

- **Rapid and secure** -- Another benefit of working with the SMC SDK is the rapid development of secure, distributed applications. For server-side development, the SMC SDK utilizes Java developers' existing knowledge of standard Java RMI (Remote Method Invocation) and avoids presenting developers with any new paradigms or technologies. For client-side development, the SMC SDK simply extends the set of lightweight user interface components found in the Java Foundation Classes (JFC, a.k.a. Swing) with new components and features such as wizards, property sheet editors, filtering, sorting, and more.

- **Pure Java** -- Because the SMC SDK is nearly 100% pure Java, it can be made available on any platform with a Java2 runtime environment. A completely platform-independent version of the SMC SDK is possible, as the only portions of the SMC SDK which are platform dependent are for functions such as user authentication and authorization. These portions can be replaced with different implementations on platforms other than Solaris.

SMC SDK Contents

The SMC SDK is a complete distributed application environment, including a server providing remote services and a local console client providing an integrated user interface. The SMC SDK includes:

- **Core services** such as:
  - Authentication
  - Authorization
  - Logging

- - Messaging
  - User Preferences
  - Registration
  - Application launch management
  - Persistence
- **JFC extensions** such as:
  - Dialog
  - Wizard
  - Property Sheet Editor
- **JavaDOC** -- All public classes documented in HTML
- This **SDK Guide**
- **Example code** demonstrating many key features of the SMC SDK

Getting Started

This section provides an overview of SMC architecture, and explains the basic steps and considerations involved in creating and porting tools and services for the SMC platform.

SMC Architecture

SMC is a Java-based application environment designed to facilitate the rapid development of network management client tools and server-based services.

In the simplest terms, the SMC environment is based on three types of JavaBeans components:

JavaBean	Description
Tools	Client-side applications; for example, a date/time tool that shows a clock and calendar, and which allows the user to change the date and time on a machine. Tools are the simplest GUI presentations in the SMC system.
Consoles	An extension of a Tool, providing a more advanced GUI presentation, and acting as a container for sets of client Tools. For example, a Console could provide a common display hierarchy, toolbar, menu bar, and information pane for the Tools it contains. The benefit of this model is that Tools can be combined into a single Console implementation, sharing the common resources provided by the Console, yet still retain their own complex hierarchies and behavior.
Services	Server-side components that support SMC tools; native SMC services are generally a combination of Java and platform-specific code. SMC includes a set of core services for security, authentication, authorization, messaging, user preferences, registration, logging, and launch management.

These three primary components are integrated by means of three sets of *meta-components*:

Meta-Component	Description
Look and Feel	The presentation layer used in a Console -- that is, the specific window and dialog types, interface widgets, layout, and so forth. In the SMC, "Look and Feel" is a pluggable component, and you can use whatever look and feel -- including a command-line interface -- that is most appropriate for your tools and customers.

Infrastructure	The "glue" that holds everything together; the SMC infrastructure includes the core services listed above and an 📖RMI-based communication model, although SMC tools and services can also be implemented using other infrastructures, such as 📖CIM/ 📖WBEM,and 📖SunMC.
Event Bus	Any tool running in the SMC console can receive events related to any other tool in the console by means of the `VConsoleActionListener` interface. Tools that implement `VConsoleActionListener` are said to be on the *event bus* because, analogous to a hardware bus, all events on the bus are public and available to any tool that is configured to listen for one or more event types. See Events for more information.

The illustration below provides a general overview of SMC architecture.
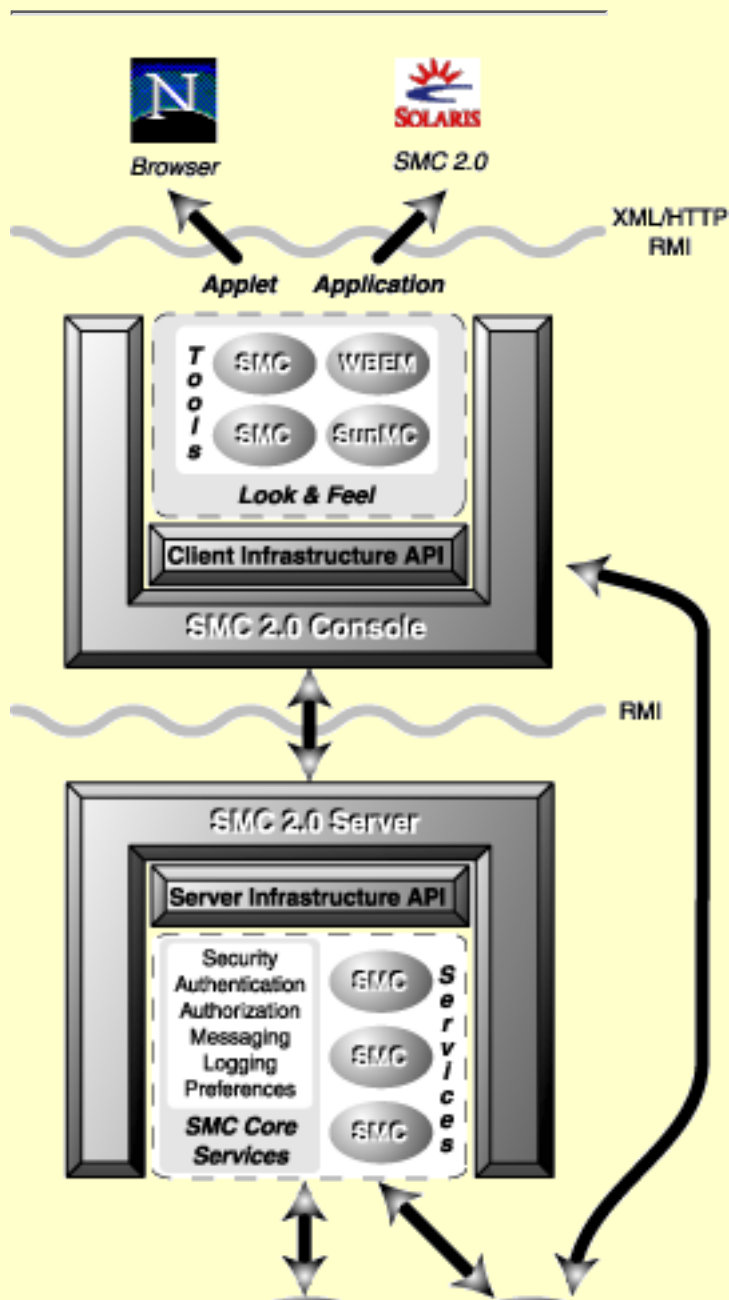
SMC Architecture Overview

Console Description

In the accompanying illustration, the SMC Console contains four tools implemented as JavaBeans. In this case, two of the tools are SMC native, one is WBEM-based, and the fourth is SunMC-based. The specific combination of tools here is for illustration purposes only -- the point is that you can integrate native SMC tools with tools based on other object models.

Tools are organized into ▨ toolkits, which are properties files associated with given users, groups, and/or administrative roles. Toolkits, in turn, are displayed in the console, with the specific user interface determined by the *look and feel* implemented for the console.

▐▷ *Look and feel is a pluggable component that can vary from console to console regardless of the toolkits used. For example, the look and feel of a given toolkit in one console could implement the common MS Windows Explorer-like interface, while the same toolkit could be used in a browser-based interface modeled on Motif property sheets. SMC tools can be ported as standalone Java applications, and browser-based Java applets. The illustration below shows the default SMC look and feel.*
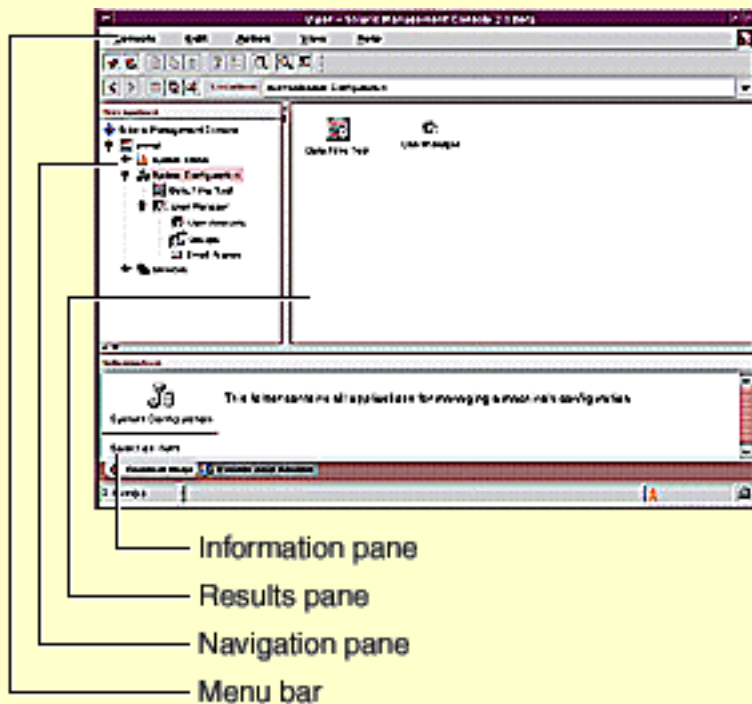
Default SMC Look and Feel

The console provides container and wrapper services for the tools. One of the most important aspects of

this wrapper service is to provide a conduit between the toolkit and the platform-specific APIs. This makes it possible to write tools that have no platform-specific interfaces.

Communication between the client view -- that is, a browser-based applet, a standalone Java application, and so forth -- is accomplished via XML and/or HTTP over RMI.

Note that while creating customized consoles is certainly possible, supporting documentation is not available at this time.

Server Description

Again, in the architecture illustration above, the SMC server contains the SMC core services and three tool-specific SMC services. A SunMC server and a CIM object manager communicate with the SMC server directly. The CIM object manager also communicates with a WBEM tool in the console.

▌▷ *Services implemented in the SMC server can be configured as daemons, singletons, or multiple-instance processes.*

One of the interesting features of the SMC server model is that it supports non-native services like CIM/WBEM and SunMC. The SMC server acts as wrapper for these non-native services, providing a model for extending the server infrastructure APIs to support platform-specific hooks. Moreover, because the SMC server is built from sets of pluggable native and (in this example) non-native components, dynamic and distributed management of services -- for example, updating and replacing components -- is relatively quick and easy.

Similar to the model used for SMC client tools, the SMC server infrastructure API isolates platform-specific code in the server components from the tools by means of the server infrastructure API. In this model, the tools do not need to know about the underlying platform on which a given service runs.

▌▷ *Ideally, to take advantage of SMC's component-oriented design, you should as much as possible adhere to Model->Viewer->Controller paradigm of separating your client and server code as much as possible, and rely as little as possible on platform- or service-specific APIs.*

In the SMC coding model, tools do not have direct access to console components -- for example, the navigation and results pane. Instead, specific interfaces are provided for accessing the underlying data model, and for accessing and customizing the console.

Remote Method Invocation

As mentioned previously, SMC uses 🕮 RMI by default for communications between services and tools. There is nothing in the SMC architectural model that prevents the use of other communication models, but RMI is recommended for its ease of use, which complements the SMC goal of enabling rapid development of management applications.

▌▷ *In cases where difficulties using RMI arise -- for example, when operating through some firewalls -- RMI has the capability to fall back to HTTP as its transport mechanism.*

The illustration below shows a typical RMI client/server interaction, in which a skeleton object is used for passing client and server parameters. In this model, a typical server application creates some remote

objects, makes references to them accessible, and waits for clients to invoke methods on these remote objects. A typical client application gets a remote reference to one or more remote objects on the server, and then invokes methods on them. RMI provides the mechanism by which the server and client communicate and pass information.
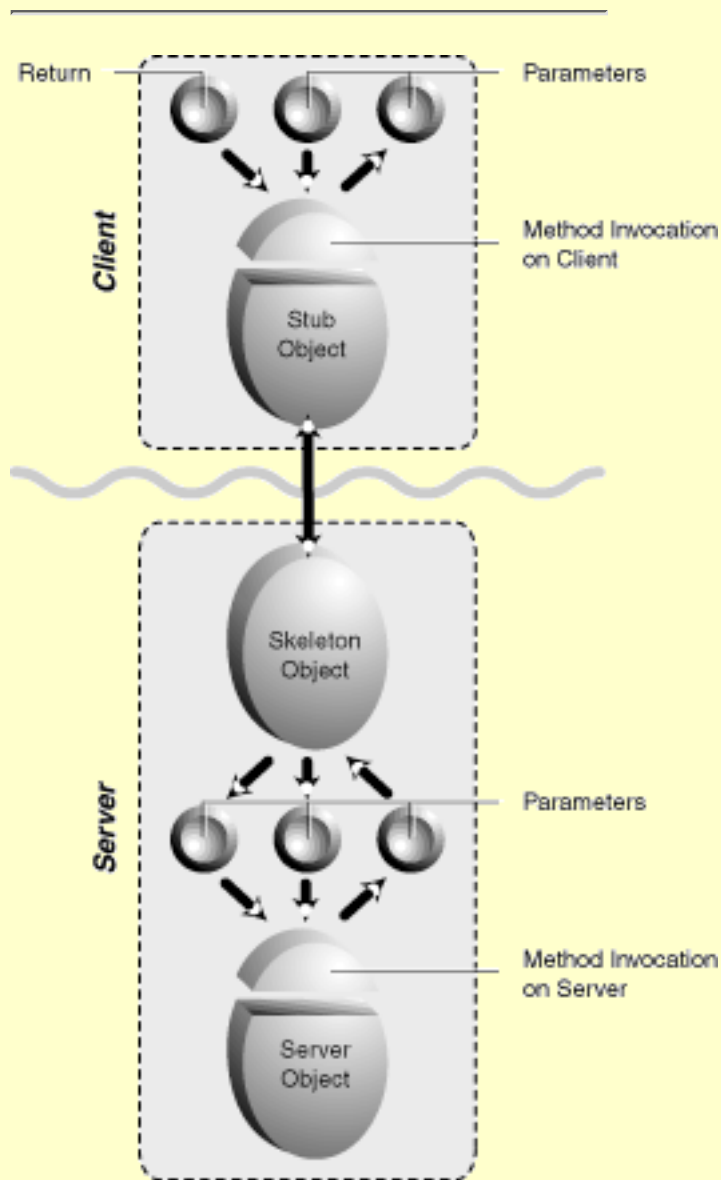
RMI Client/Server Model

▌▷ *Writing an SMC application is similar to writing a common RMI application, except that SMC developers can ignore the "stub" and "skeleton" components shown in the illustration.*

Sample User Session

Before stepping through the flow of events in a typical SMC user session, it is useful to describe SMC tools and toolkits in slightly more detail.
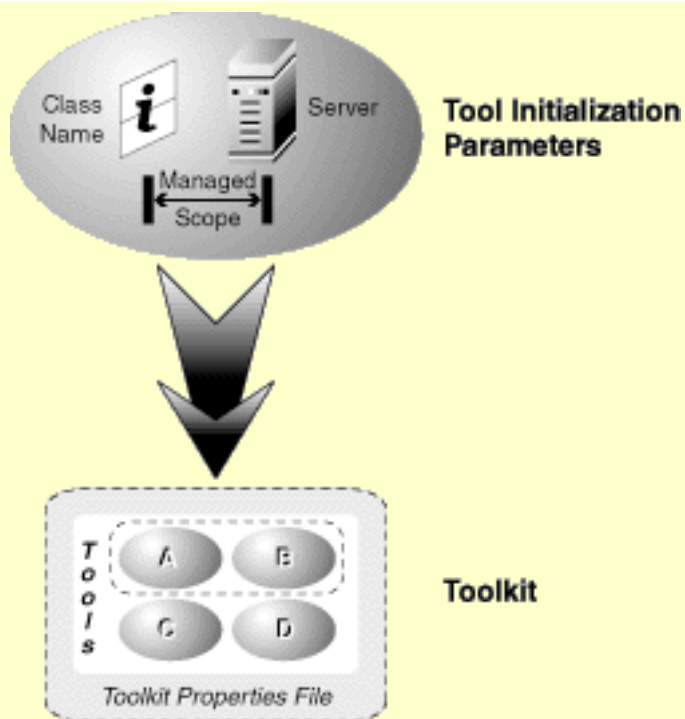
Tool Initialization Parameters

Toolkits are defined by properties files, in which sets of tools are specified. In particular, as shown in the illustration above, an SMC toolkit needs to know three things about a tool before that tool can be initialized:

- **Class name** -- The tool's class name, as provided by the tool information file (XML format)

- **Server name** -- The name of the local or remote server on which the tool is located

- **Managed scope** -- The scope, in URL form, that the tool can manage; if that scope requires a session to some other server, the tool obtains a session to that machine using `getExternalClient()`

With this in mind, the illustration below shows an example of the flow of events in a typical SMC session.

Tool Parameters

Authentication

Authentication

Authentication

**Toolkit View
Downloaded to Console**
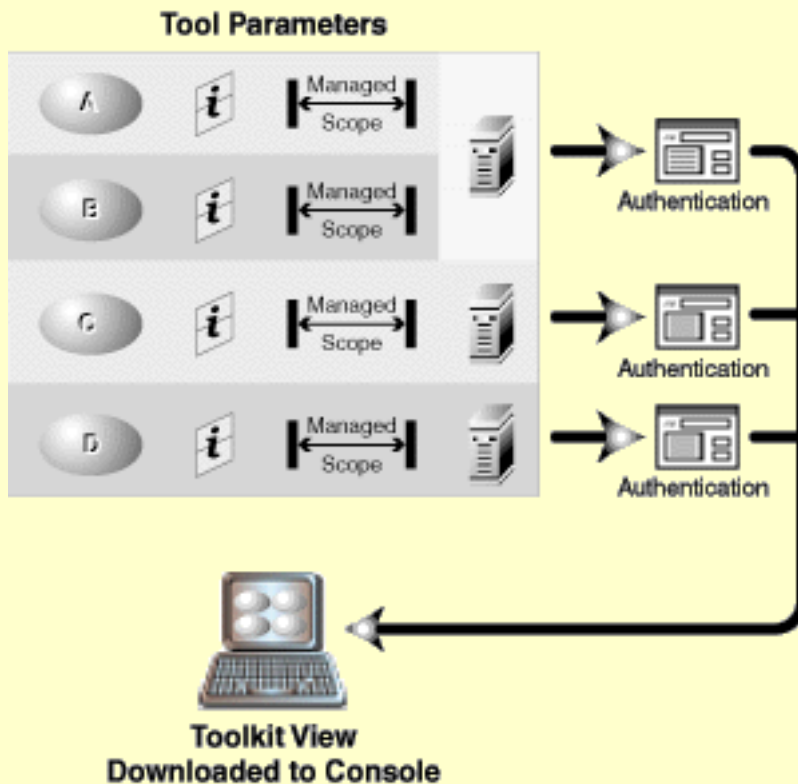
Typical SMC Session

Before doing anything in a console, the user must open a toolbox. The console attempts to construct a view of the toolbox by reading the server location and the class name for each tool encountered.

For each server on which a tool is located, the SMC console must authenticate the user before any tool or tool information can be retrieved. This authentication happens through one or more dialog boxes, depending on where the tools are located.

For example, in the illustration, tools *A* and *B* both reside on the same server, so the console only requires the user to log in once. However, tools *C* and *D* reside on two different servers, and so the user is presented with two separate authentication dialogs. In this example, the user must log in three times, to three different servers, before the toolkit will load.

How to Proceed

It's probably best to understand the main classes that you need to implement and/or will work with most often in the SMC environment. The reader is urged to consult the SMC SDK javadocs for details on these classes.

The primary classes and files are:

- Tool
- Tool Descriptor
- VConsoleProperties

- [VConsoleActionListener](#)
- [VScopeNode](#)

Creating Tools

The general steps for creating an SMC tool are as follows:

1. Develop a JavaBean implementing the SMC Tool interface
2. Include the Java interface for the Bean's corresponding Service Bean (optional)
3. Package the Bean and register the Bean with the SMC server
4. Add the Bean's authorizations to users, profiles, roles, etc.
5. Add the Bean to a configuration

Creating Services

The general steps for creating an SMC service are as follows:

1. Develop a Java interface for the server-side Bean
2. Create a Bean which implements the interface and the SMC Service interface
3. Package the Bean and register the Bean with the SMC server
4. Add the Bean's authorizations to users, profiles, roles, etc.

Migrating Applications to SMC

The first question one might ask when considering porting their applications to SMC is "How much of my existing code is salvageable?". It depends, you might salvage more than half, or you might salvage nothing. For applications that display their data as simple nodes or in tables, the SMC engine will handle all the rendering for you; you need only provide your icons and data. So if your application maintains a clear seperation between it's data model and the presentation of that data, you can salvage the data model and it's management code, but throw away the presentation code. Note that applications can still do their own rendering, but that SMC provides "Windows Explorer-like" rendering services for free for those applications that need a similar presentation.

Many applications are integrated into a parent console which is proprietary to their product. If access to that console is provided thru a proxy class which provides the interface to the console, then only the proxy needs to be changed to "point" to the SMC console, with little or no changes to the application. Otherwise, you'll need to specifically port every instance where the application has specific knowledge of the parent console.

In the SMC framework, applications do not have direct access to console components (eg., the navigation pane, the results pane). Instead, specific interfaces are provided for accessing the underlying data model and for accessing and customizing the console.

The SMC Service model is similar to other RPC models. You need to seperate out the presentation layer from those functions that need to be performed on the server. The presentation layer becomes the client side tool and the part that runs on the server becomes one or more services. An interface needs to be defined between the tool and service(s). Some of the native code on server side can be preserved by adding a JNI interface to it. Click here for details on porting an RMI service implementation to an SMC service.

Starting the Console

In most cases, you start the console with the command `/usr/sadm/bin/smc`. This will launch the console and allow you to run tools from the toolboxes you load. By default, SMC will load the toolbox called `this_computer.tbx` for the local machine on which the console was started.

IMPORTANT: Note that only upon the first time the console is started after installation, there will be a delay of several minutes before the toolbox is loaded. This is due to SMC server configuration taking place. During this time, SMC is bootstrapping it's registry with the tools and services required for the SMC framework to run properly. It is important to be patient and let this configuration complete, otherwise SMC may not function properly. This delay will be eliminated in a future release.

Starting Services

SMC services are bundled with WBEM services into a single JVM, and thus can be started and stopped thru the WBEM script `/etc/init.d/init.wbem`.

To start the services, enter the following command:

```
/etc/init.d/init.wbem start
```

To stop the services, enter the command:

```
/etc/init.d/init.wbem stop
```

Tools

Overview

This section provides detailed information about the SMC client tool model, and describes the core SMC client classes you must use in your tool implementations.

Tool Model

The SMC tool model defines the basic components of a tool implementation, and how these components are used to manage the presentation of data. A tool consists of five major components, as described below.

Tool

The `Tool` class is the main interface client tools must implement, and is the top-level client class instantiated by the console.

You are not limited to one instance of the `Tool` class. For example, you can create one instance each for all your navigation nodes, or create one master instance with which all nodes are associated, or create one for each of the non-leaf nodes, among other possibilities.

The design you choose depends on how you want to structure your application, and how much may be in common among your navigation nodes. Navigational nodes are associated with specific `Tool` instances by means of `VScopeNode.setTool(Tool)`.

While you must implement all methods in the `Tool` interface, six are of particular interest, and called in the order as presented here:

Method	Description
`Tool.setToolContext()`	Provides a handle to the ToolContext object, used for retrieving information (such as it's management scope) about the environment the tool is run in.
`Tool.setProperties()`	Provides a handle to the shared properties object that all components in the SMC system can use for property storage

`Tool.addConsoleActionListener()`	Allows other system components to register to receive your events; you simply need to maintain a list of these listeners
`Tool.init()`	Called after the Tool is instantiated, it is the time when a Tool should connect to server-side services.
`Tool.start()`	Called whenever any node associated with the `Tool` instance is clicked; for a one `Tool` instance application, it is a signal that your application has focus
`Tool.stop()`	Called whenever any node *not* associated with the `Tool` instance is clicked; for a one `Tool` instance application, it is a signal that your application has lost focus ▐▷ *It is important to know when your application does or does not have focus because it will receive all events on the* ▦ *event bus, even those related to other applications.*

Tool Descriptor

Every tool must have a deployment descriptor before it can be registered and maintained by SMC. A descriptor includes static information about the tool component that is used by the SMC console to form a representation of the tool without actually instantiating it, and to manage the lifecycle of its instances.

There are several attributes common to all tools, such as:

- **Tool package path (*provider-class* tag)** -- SMC needs the *provider-class* in order to instantiate your Tool.

- **Resource bundle location** -- Defines the base location and name of the Tool's ResourceBundle (*resource-bundle* tag). The *resource-bundle* is used to lookup special localized properties that are needed in order to present the Tool in the console without instantiating it. These special properties are: `LARGEICON`, `SMALLICON`, `BEANNAME`, `DESCRIPTION`, and `VENDOR`, and are discussed TBD.

- **Help file location (Optional)** -- Defines the base location and name of the JavaHelp help set for this component. See TBD for details on the policy established by SMC for packaging of JavaHelp bundles.

- **SMC SDK API version (Required)** -- Defines the String version number of the SMC SDK API this component builds against. This must be in the form of: `major[.minor]`

- **Supported tool display contexts** -- Application GUI, applet, CLI.

- **Supported Management Scopes (Optional)** -- Defines the management scopes this service supports. Supported management scopes specifies which one or more of the following name services contents will be accessed and/or changed by this service:

  ```
  file | nis | nisplus | ldap | dns
  ```

  If not specified, `file` scope is assumed.

- **Runtime parameters (optional)** -- Defines runtime attributes that effect service behavior. Unlike global registry properties, these runtime parameters can be different for each tool.

An SMC tool descriptor is defined as an XML-based file. Please refer to the "Deployment Descriptor DTD" at `/usr/sadm/lib/smc/lib/dtds/viperbean_1_0.dtd` for detailed syntax information.

Sample Code	• Tool Descriptor

VConsoleProperties

`VConsoleProperties` represents a shared `Properties` object that all components in an SMC system can use for property storage.

The `setProperties()` method in your `Tool` instance is called once after instantiation to provide a handle to the Properties object. You may want to cache the `setProperties()` reference because you will eventually need it to access console properties.

Sample Code	• Tool.setProperties()

▌▶ *Additionally, you can register a `PropertyChangeListener` with the `Properties` object so you can be notified when properties change. The easiest way to do this is to have your main Tool class also implement `PropertyChangeListener`.*

Sample Code	• PropertyChangeListener

In addition to the various console properties as defined in VConsoleProperties, you can create your own properties for storage of tool preferences, so that they can be restored in subsequent sessions. To avoid namespace collisions with similarly named preferences in other tools, or even within the same tool, it is recommended that preference names be based on the full class name of the class in which the preference is used.

Sample Code	• Preferences

VConsoleActionListener

Any class that implements `Tool` should also implement the `VConsoleActionListener` interface, which enables the `Tool` to be notified of various events on the ▦event bus. All events are `VConsoleEvent` with String-type event IDs.

There are numerous console-specific events which your application can listen for, all of which are listed under `VConsoleActions`. Additionally, you can define your own event IDs and send them to other components in your own application or to other applications that have knowledge of your event IDs. You just need to ensure your event ID's are globally unique, similar to system properties -- for example, `com.mycompany.myproduct.mytool.formatDiskNow`.

If your application is registered on the event bus, it will receive all console events, even those related to other applications. You therefore need to make sure your application does not execute, say, a refresh operation while another application has focus. You can use the `Tool.start()` and `Tool.stop()` methods to track whether or not your application has focus, and then react appropriately when listening for events.

Sample Code	🔹 <u>VConsoleActionListener</u>

One of the most significant benefits of the event bus architecture is that it makes it possible to develop dynamically configured display models with no API-specific dependencies. For example, when the user selects "large icon view" from the toolbar, the toolbar will modify the `ICONSTYLE` property to be `LARGE`. As soon as this property is modified, all components in the system which have registered for property changes are notified of the property changed and of the new value. The result pane for instance, will update its view to correspond to the new `ICONSTYLE` property setting.

By comparison, if SMC used an API driven model, the toolbar would have to have a reference to the results pane and know the method to call to update its display, as well as any other component in the system which needed to know the value of the `ICONSTYLE` change.

An example of a console event would be the user selecting a node in the navigation tree. The tree view component will create a `SCOPESELECTED` event and send it on the event bus. Components which are interested in `SCOPESELECTED` events will process the event and react accordingly. For instance, the result pane will display the children of this newly selected node. The `InfoBar` component will count the number of children the newly selected node has and set its `textfield` display to reflect that, such as `6 Item(s)`.

Eliminating API dependencies thus creates a very flexible display model. Components no longer need any references to other components in the system, nor do they need to know the methods to invoke, they simply modify a known property or generate a known event. The other components in the system that are interested in the event or property will update themselves accordingly. Display components can easily be added and removed without fear of breaking API dependencies or passing around references to necessary components. Only one reference needs to be set to the properties object, and components added as event listeners to create the event bus.

As mentioned previously, `Tool.addConsoleActionListener()` lets other system components register to receive your application's events, and you simply need to maintain a list of these listeners. You use the standard "fire" method to notify all registered listeners of a particular event.

Sample Code	🔹 <u>Console Listeners</u>

VScopeNode

Perhaps the most widely used class is `VScopeNode`. This is the class in which you provide information (like icons, column headers, and so forth) about your data model to the console, whether that information is rendered in the navigation pane or results pane.

▐▷ *Use the **payload** field to associate your application-specific object with the node, so that you can easily get a handle to your data object when events are received for a specified node.*

You create an instance of `VScopeNode` for every object you want to appear in the navigation pane. All navigation pane nodes that are children of a given parent node will automatically be rendered in the results pane when the parent node is selected.

You denote *non-leaf* nodes by setting the *internal root* of the node to null (`node.setInternalRoot(null)`). Perhaps a better term for these non-leaf nodes in the SMC context is *exposed* nodes, because you are exposing the model completely to the SMC engine for it to render and manage.

Additionally, SMC manages the opening of the corresponding results pane representation of the exposed node when it is double-clicked by automatically navigating to its node representation in the navigation pane. You do not need to do anything special to manage the results pane in a model that does not have an internal root.

You denote *leaf* nodes by setting the *internal root* of the node to a `VScopeNode` instance. Leaf nodes refer to models that your application will manage -- the model is not completely exposed to the SMC engine. In the SMC context, this is referred to as an *extended* or *internal root* data model.

In an *extended* model, your application is responsible for creating `VScopeNode` instances for each object you want to render in the right-side results pane. Each of these nodes must be added as siblings to the same parent node -- that is, to the *internal root* node. Whenever you change this model -- for example, by adding deleting or modifying -- you must post an `UPDATESCOPE` event.

Sample Code	● UPDATESCOPE

UI Components

The illustration below shows the default SMC look and feel, and the location of each of its components. Note that tools do not have direct access to any of the components in the console. Instead, specific interfaces are provided for accessing the underlying data model and for accessing and customizing the console.
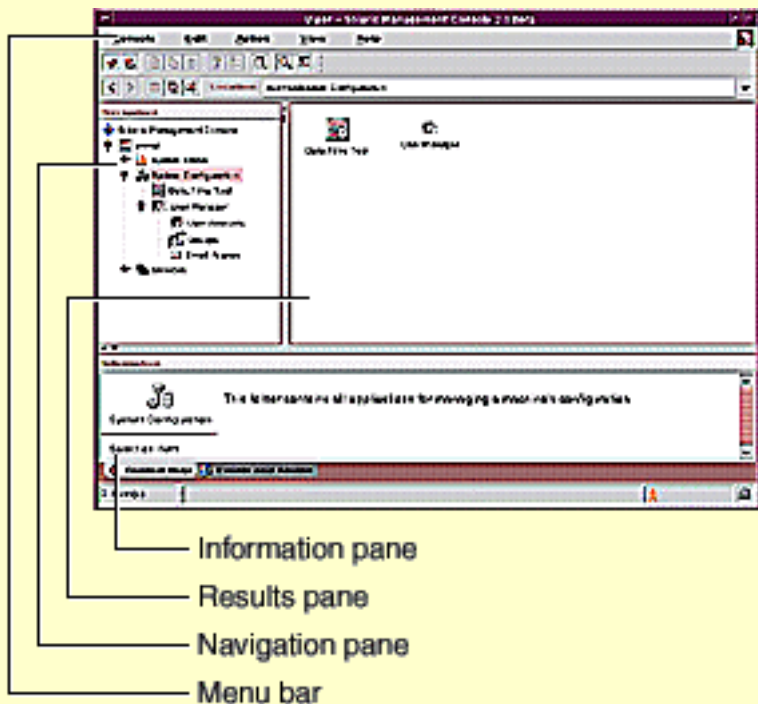
Default SMC Console Window

Navigation Pane

The Navigation pane works like a frame in a web page: clicking an item determines what appears in the View pane. Before proceeding, it is probably best to have an understanding of the VScopeNode class, and how it can be used to manage nodes in the Navigation pane.

While SMC knows how to create a the node representation of your tool in the Navigation pane without actually instantiating it (via the Tool Descriptor), a tool can re-create or replace this representation during it's instantiation. It is important that this top-level node for your tool be returned by your `Tool.getScopeNode()` method:

Sample Code	◆ Create Tool Node

To add a node as a child for a specified parent node:

Sample Code	◆ Add Child Node

To remove a node that is a child of a specified parent:

Sample Code	◆ Remove Child Node

View Pane

The View pane (usually the "right-side" pane, and sometimes called the results pane) displays the contents of the node selected in the Navigation pane, where the contents could consist of folders or tools. If the node selected in the Navigation pane is a folder, the contents of that folder are displayed in the View pane. If the node selected is a Tool, the top-level contents for the tool is displayed, whether that be

folders or simply the implementation of that Tool.

SMC supports 4 display styles in the View pane, accessible via the `VConsoleProperties.ICONSTYLE` property:

Style	VConsoleProperties Name
Rows and columns of small icons	SMALL
Rows and columns of large icons	LARGE
A single column of small icons, one per row	LIST
Tabular view of detailed data arranged in columns	DETAILS

A tool does not need to do anything to support all these styles. The SMC automatically ensures that when the user changes display style for the current tool which has focus, changing focus to another tool will preserve that display style.

A tool can restrict the available styles available for the user to choose from. However, since the `ICONSTYLE` property is a shared property amongst all the tools, then a Tool must restore the previous style when it loses focus.

Sample Code	◈ Details Style Only

Tools are not limitted to restricting the presentation to a single style. A tool can restrict more than one style and still allow the remaining styles to be selected by the user.

Sample Code	◈ Enable Styles

Information Pane

The Information at the bottom of the console displays either context help for the object selected in the Navigation pane, or a list of alarm types, depending on whether the Context Help or Console Events tab is selected.

Context help must be in HTML format. Typically, the help is simply included in the Tool's jar file and retrieved via `ResourceManager.getLocalizedTextFile()`.

Sample Code	◈ Set Context Help

The Console Events log provides a view of events that occur between the console and its tools, for example authentication events and tool loading problems. There are 3 types of events defined in the `VLogEvent` class: `INFORMATION`, `WARNING`, and `ERROR`. Note that console events are not persistent, and are lost when the console is exitted.

Sample Code	◈ Log Console Event

Menu bar

The Menu bar includes a series of menus which are common for all tools. Tools can implement the own

menu bars by extending JMenuBar and creating their own menus in the usual manner. These menus can be added to the console's menus via the `JMenu.setActionCommand()` method and the constants defined in the `VMenuID` class.

Sample Code	⬥ Menubar Integration

Status bar

The Status bar at the very bottom of the Console has 3 distinct panes for displaying certain kinds of information.

The left pane indicates the number of items (nodes) in the View Pane for the currently selected node in the Navigation pane. Send an `VConsoleActions.UPDATESELINFO` event to the console to display your text in the left-side info pane. By default, SMC provides a message of "# Items":

Sample Code	⬥ Set Left Status Info Pane

The center pane indicates console activity -- for example, a progress meter, or back and forth "shade" movement. If your tool is not able to determine progress status for a long operation, then enabling the back and forth "shade" movement can be done by simply sending a `VConsoleActions.UPDATEPROGRESS` event to the console. Re-send the same event to disable the movement.

If your tool is able to track progress, then you can specify a JProgressBar instance to be displayed in this center pane, and simply update the JProgressBar as needed to show progress:

Sample Code	⬥ Set Center Status Info Pane

The right pane provides progress information in the form of text messages. Send an `VConsoleActions.UPDATESTATUS` event to the console to display your text, similar to the left-side pane as shown above.

Accessing Resources

All resources should be loaded using the `ResourceManager` and `ConsoleUtility` classes, and should be located on the same root path as your main `Tool` class. For example, if the *provider-class* name for your main `Tool` specified in the Tool Descriptor file is `com.mycompany.myproduct.mytool.client.VMytoolMgr`, then all resources should be rooted at `com.mycompany.myproduct.mytool.client`, ideally in subdirectories of this path.

▐▶ *Although the `ResourceManager.getLocalized*()` methods are convenient for downloading files, these methods do **not** cache files on the client. Therefore, successive attempts to access the same resource come at the expense of another download. With this in mind, you should implement some sort of caching scheme to minimize the number of downloads, while at the same time taking into account the memory costs of caching. It is anticipated that caching will be a feature of SMC in a future release.*

Resource Bundles

Use `ResourceManager.getBundle()` to load resource bundles.

> *Sample Code*   🔹 [ResourceManager.getBundle()](#)

Online Help

Online help can be provided in 2 forms: "Spot" or context-sensitive help provides short-and-simple information for dialog components in the form of HTML files. "Extended" help provides more extensive information and search capabilities in the form of JavaHelp helpsets. For more information, see the sections that discuss these two formats further under [Localization](#).

Use `ResourceManager.getLocalizedTextFile()` to load basic (non-JavaHelp) HTML help files. Note the special heuristic used with respect to the current locale and the "C" default locale. Default English HTML files might best be located in the `C/html` of your main `Tool`; for example, `com.mycompany.myproduct.foomgr.client.C.html`.

> *Sample Code* 🔹 [ResourceManager.getLocalizedTextFile()](#)

Note that SMC does not at this time provide support for hyperlinking between HTML files, nor to helpsets. You must implement your own hyperlink listening code on the JEditorPane component of a `VOptionPane` and render the target of the link using `VOptionPane.setHelpHTML()`. This will be provided in a future release. In the meantime, you must implement your own link listener.

If you do implement your own link listener, it is possible to hyperlink to a specific helpset target under the following conditions:

- Due to a Java limitation, linking to helpsets is only useful from non-modal dialogs or main frames, as modal dialogs will block input and prevent you from using the helpset viewer.

- Hyperlinks from within a context-sensitive html file to a helpset target must be of the form `helpset://<helpset filename>/<target>`, where `<helpset filename>` may or may not include the `.hs` extension. For example: `helpset://my_helpset.hs/my_target` or `helpset://my_helpset/my_target`.

Then your non-modal dialog or frame must implement a `VConsoleActionListener` to rout the link event onto the event bus so the console can act on it and launch the help viewer for the specified target. Note that you need to also check the link event to make sure it is indeed a link to an external target, and NOT rout the event if it isn't. The best way to do this is check that the URL's protocol specification is `helpset://` and that the URL does NOT end in `.html`.

> *Sample Code*   🔹 [Hyperlink to Helpset](#)

Exceptions

`VException` is the class used for managing exceptions. It is not necessary to invoke the ResourceManager directly, as this is done automatically when you attempt to retrieve the exception's localized message via `VException.getLocalizedMessage()`.

You should override two methods in your `VException` subclass.

Sample Code	✦ [Exceptions](#)

Images

Use `ConsoleUtility.loadImageIcon()` to load image icons.

Sample Code	✦ [ConsoleUtility.loadImageIcon()](#)

Top of Page

Packaging

Manifest

All tools must include a manifest file in its jar file. The manifest must include the full package path of the main Tool class, and the full package file of the Tool Descriptor file:

Sample Code	✦ [Manifest for Tools](#)

Resource bundles

It's required that all tools provide a resource bundle that contains information like name, description, icons, vendor, version under predefined message keys:

- **BEANNAME** -- Localized tool name
- **DESCRIPTION** -- Localized description of the tool
- **VENDOR** -- Localized vendor name
- **VERSION** -- Localized version number
- **LARGEICON** -- Path of large icon relative to this bundle
- **SMALLICON** -- Path of small icon relative to this bundle
- ***<property_name>*.DESCRIPTION** -- Optional description of a property this component defines
- ***<parameter_name>*.DESCRIPTION** -- Optional description of a parameter this component defines

The icon paths specified as LARGEICON and SMALLICON should be relative to this resource bundle's location.

ResourceBundles can be implemented as compilable subclasses of `ListResourceBundle`, or as `.properties` files.

Sample Code	✦ [Tool Resource Bundle](#)

Scope

The `AdminMgmtScope` class represents a management scope or domain; that is, a name service domain or a single system. Use the `ToolContext` instance (obtained via your Tool.setContext() method) to retrieve context-specific information about the environment the Tool is running in.

Sample Code	⚫ Scope

Registration

If a tool uses classes or resources from some other library jars, you need to register them first, for example:

```
# smcconf repository add library ALL <path>/mylibrary.jar
```

Here we are using the pseudo bean name 'ALL' to represent all tools and services.

Next, you register a tool jar to the server repository:

```
# smcconf repository add bean <path>/mytool.jar
```

You can check the repository to see the tool is successfully imported by doing repository listing:

```
# smcconf repository list
```

See the Registration section for more details regarding the *smcconf* command.

Localization

ResourceBundles

Localized ResourceBundles are implemented using the standard Java heuristics (*baseclass_language_country_variant*). Consult the JDK docs for detailed information.

As mentioned ealier, ResourceBundles can be implemented as compilable subclasses of `ListResourceBundle`, or as `.properties` files. If you use `.properties` files, they will not load properly at runtime if the translations are in a non Latin-1 based character set (multi-byte environments) because there is no way to specify a character set encoding. Therefore, they must be converted to Latin-1 or Unicode-encoded characters using the `native2ascii` command.

Sample Code	⚫ `native2ascii`

HTML files

Localized HTML files are implemented using a similar heuristic as ResourceBundles, although each localization must reside in a unique directory based on locale, with identical filenames across locales. For example:

`com/mycompany/myproduct/C/html/foobar.html`	English locale; system default, referred to as the "C" locale
`com/mycompany/myproduct/fr/html/foobar.html`	French locale
`com/mycompany/myproduct/de/html/foobar.html`	German locale

When translating HTML files, you must specify the proper character set via the CONTENT field. Note that this information must be in a line that is NOT embedded withint a `<HEAD></HEAD>` block.

Sample Code	▪ [Setting Charcter Set Encoding in HTML](#)

JavaHelp Helpsets

Localized JavaHelp helpsets must be structured by locale, as discussed in the [JavaHelp User's Guide](#). Specifically:

- The `.hs` file must be named following the *baseclass_language_country_variant* scheme as for Java ResourceBundles (`foo_locale.hs`).

- All other files belonging to the helpset must live in a subdirectory named for the locale. The illustration below shows a sample layout of the filesystem hierarchy of localized helpsets.

```
...\help
    my_helpset.hs
    my_helpset_de.hs
    my_helpset_fr.hs
```

JavaHelp Localization Hierarchy

- All references to those files in the `.hs` file must reflect this subdirectory name.

Sample Code	◆ Localized Helpset

- URLs must reflect the location of the html files relative to the helpset, regardless of locale. For example, if some html files were located in the `topics/topicA` subdirectory, as shown in the illustration above, then URLs for those files referenced in `Map.jhm` would begin with `topics/topicA/` in front of each file referenced.

Sample Code	◆ Helpset Map File

To translate a JavaHelp helpset, edit the specified files as follows:

`.hs`	Specify proper character set encoding; translate all `<label>` tag values
`*.html`	Translate and also specify the proper character set via the CONTENT field, as shown here
`index.xml, map.jhm`	Specify proper character set encoding
`toc.xml`	Specify proper character set encoding; translate all tocitem "text" fields only (do NOT translate "target" fields)

The final step in constructing a helpset is to build the search index. This requires that you have JavaHelp 1.1 installed on your machine.

Sample Code   ● Build Search Index

Sample Code   ● Build Search Index

Toolboxes

Overview

SMC uses the concept of a toolbox to provide a view of various system administration tools or applications, possibly on different servers, within a common user interface. A toolbox is a file in XML format, that is registered with the SMC server. It is a hierarchical collection of folders, tools, legacy applications, and links to other toolboxes. This collection defines what you see in the Console navigation pane.

The root toolbox or container is called "Management Tools". Its default behavior is to look for a default toolbox on the host machine and link to it when the SMC is started. The default toolbox is called "This Computer", with the file name `this_computer.tbx`, and comes pre-configured with several tools. You can modify this toolbox or create additional toolboxes to suit your needs using the SMC Toolbox Editor. The Editor is essentially the SMC console run in a special mode.

For more detailed information regarding the SMC Toolbox Editor and management of toolboxes, start the Editor and consult the extensive online help that is available.

Top of Page

Starting the Toolbox Editor

In most cases, you start the editor with the command `/usr/sadm/bin/smc edit`. This launches the editor and allows you to edit toolboxes you load. By default, the editor will load the toolbox called `this_computer.tbx` for the local machine on which the editor was started. Consult the SMC man page `smc(1M)` for additional options for starting the editor.

Services

Overview

This section provides detailed information about the SMC service model, and describes the core SMC service functions you can use in your service implementations.

Common Services Model

The SMC service model defines how services are used by clients and the basic components of service implementation. A service consists of three major components: interface, implementation, and descriptor, as described below.

Service Interface

From the client's point of view, service is defined only by its public interfaces. A client should be able to retrieve a service handle by providing information on the public interface it is expecting, without knowing the specific service implementation details. For example, an SMC client may want to access a logging service in an environment where there may be many different implementations of logging services available. In this case, the client simply requests a service that implements some well-known logging interface.

Sample Code	● Access a Log Service

SMC predefines a set of common service interfaces, including log, authorization, message, persistence and launch. Vendors can define their own public service interfaces and share between its clients and its implementations, as long as the service interface satisfies the following rules:

- Extends base interface `Service`; this interface serves the similiar purpose as `java.rmi.Remote`. The methods defined in its subclasses are considered public business functions.

- Each method must be public.

- Each method must declare `java.rmi.RemoteException` in its throws statements; this is required even if this interface is only accessed locally.

Sample Code	● Service Interface Definition

Service Implementation

A service implementation in SMC must satisfy the following rules:

- Implements at least one service interface

- Implements interface `ServiceProvider`; the SMC SDK provides a convenient base class for service implementation, `VService`. Services can optionally override its methods to plug in their own logic.

- Has a public default constructor that takes no parameters. SMC will use this default constructor to instantiate this service.

Sample Code	⬥ Service Implementation

SMC services can be designed to run on client-side as well as on server-side. A server-side implementation basically acts as a Java RMI server object. Clients do not have a direct handle to the service object, and can only remotely access it through an automatically generated stub. A single instance of such a server-side service can handle all requests from different client processes. This is the default running mode for service implementation objects.

In some circumstances, it may be preferable to run a service directly in the client process/VM. For example, when a client running inside a firewall tries to access a news service. If the news service can be downloaded to the client VM and poll an outside server for updates, then the client won't have to cope with firewall issues. Usually client-side services act as local agents that preprocess client requests and passes them on to the real backend on remote servers. This is very useful for third party integration. A log service proxy that represents a certain backend framework service, like WBEM, Jiro, or J2EE, can shield all the details of its backend framework dependency from its clients. Please see Third-Party Integration for more information on how a service proxy accesses different framework backends.

Besides the basic requirements of a service implementation, a client-side implementation class needs to implement the `ServiceProxy` interface. The SMC SDK provides a convenient base class, `VServiceProxy`, for vendors to extend.

Service Descriptor

Every service must have a deployment descriptor before it can be registered and maintained by SMC. A descriptor includes static information about the service component and is used by the SMC console to manage the lifecycle of its instances.

There are several attributes common to all services, such as:

- Daemon versus Non Daemon

- Service loading sequence

- Public interfaces (Required); an interface element represents a public interface class through which other components access current service. It should be the fully qualified interface class name that this service wants to publish.

- Implementation class name (Required); defines the implementation class to instantiate when a

service is being loaded.

- Help file location (Optional); defines the base location and name of the JavaHelp help set for this component. See TBD for details on the policy established by SMC for packaging of JavaHelp bundles.

- SMC SDK API version (Required); defines the String version number of the SMC SDK API this component builds against. This must be in the form of : `major[.minor]`

- Singleton versus Multi-Instances (Required); defines if only one instance of the service should be instantiated to serve all clients, or whether a new instance of the service is needed for every client.

- Supported Management Scopes (Optional); defines the management scopes this service supports. Supported management scopes specifies which one or more of the following name services contents will be accessed and/or changed by this service:

  ```
  file | nis | nisplus | ldap | dns
  ```

  If not specified, `file` scope is assumed.

- Load dependency (optional); defines what other services should be running before this service can be loaded.

- Runtime parameters (optional); defines runtime attributes that effect service behavior. Unlike global registry properties, these runtime parameters can be different for each service.

An SMC service descriptor is defined as an XML-based file. Please refer to the "Deployment Descriptor DTD" at `/usr/sadm/lib/smc/lib/dtds/viperbean_1_0.dtd` for detailed syntax information.

Sample Code	◈ Service Descriptor

Accessing other services

A service can access other services using `getServiceByName()` calls. Also important in this context are call delegation and accessing remote services, both of which are described below.

Call Delegation

Call delegation is when a service needs to call other services to fulfill a client request; for example, a service may need to pass on the client's identity to the services it calls. The service can specify its access to the other services in this mode by specifying this in the `getServiceByName` call, as:

Sample Code	◈ Call Delegation

▐▶ *Be aware that the other service handle running in delegation mode only works in the current client's calling thread and its child threads. Services should not try to use delegation mode in any threads that are not triggered by current client call.*

Accessing remote services

SMC does not currently support access to remote services running on different servers -- that is, different VMs on the same or different machines.

Bundled Common Services

The SMC environment bundles several common services in its release. Developers can find interface definitions for these services in the `ServiceList` class. The bundled services are briefly described below.

Logging

The SMC Log service logs messages into the default system log. When an SMC server runs on its own, the default log is `syslog`. When running with a WBEM server in the same VM (the SMC default), the default log is the `WBEM log`.

To enable localization, it is required that all messages logged are generated from resource bundles. The Log service accepts message keys and resource bundle base names as parameters, rather than directly accepting fixed strings. The real messages are later retrieved from the resource bundle by the log viewer.

Sample Code	◈ [Logging](#)

Authorization

The Authorization service is used to decide whether an action towards some critical system resource is to be allowed or denied, based on the security policy currently in effect. Its public interface is defined as `com.sun.management.viper.services.Authorization`.

Different implementations use different security policies and different policy datastores. The implementation bundled with SMC is using the Role Based Access Control (rbac(5)) which was introduced since Solaris 8. See RBAC for details on how to install rights into its data store. Administrators can use one of the SMC tools, Users Tool, to manage the data store and grant rights to Solaris users.

In SMC, each action authorization is represented with a VPermission object. VPermission is defined by a properly scoped action name string, for example: `solaris.admin.usermgr.read` defines the read action in Solaris's User Manager tool.

Sample Code	◈ [Checking Authorizations](#)

Messaging

SMC provides a facility to exchange messages between clients (Tools or Services). There are two types of messaging interfaces, one is MessagePullAgent and the second is MessagePushAgent. Once a user gets a handle to the Core Message service, he needs to get a handle to one of the message services (push

or pull), initialize that service with either ToolInfraStructure or ServiceInfrastructure, create the named channel, subscribe to that channel and post messages to that channel. Clients need to implement the MessageListener interface to get notification from other subscribed users to receive messages.

The user has to make a decision to use either the push or pull message mechanism. Pull message is preferred if the client is running inside a firewall and needs to use a message service outside of firewall. The MessagePullAgent interface will contact the message service for updated list of messages.

Push message service will be notified by the message service through a callback mechanism so client will get the message immediately after message has been posted.

Sample Code	⬥ [Messaging](#)

Persistence

SMC provides a facility to store persistent data to SMC server. Tools and services can store/restore/delete persistent data by the `PersistanceAgent` utility class. To use the Persistance service:

1. Create the `PersistenceAgent` object with either `ToolInfrastructure` or `ServiceInfrastructure`

2. Store the serialized object with key (String) and version number (String); key is used for indexing to retrieve the object later, and version another piece of information associated with key for tools or services

3. Restore the data by key

Sample Code	⬥ [Persistence](#)

Launching

There are many non-SMC aware applications that provide management functionality administrators want to use. They can be run on the server and displayed on the client machine SMC is running on through the Launching Service. This service only supports X protocol, so appropriate permissions for the X display server are required to launch any application. You can grant display permission on the X display to the remote machine on which the application is running via the following command:

```
% xhost +<server>
```

Applications that are launched from this service fall into 3 different types:

Application Type	Description
CLI (TTY-based application)	The Launch service runs these type of applications inside a `dtterm`.

XAPP (X application)	Applications of this type are executed with the DISPLAY environment variable set to the same X server as the tool in which the given application is running.
HTML (URLs)	The first browser that can be found in the PATH environment set by the client tool will be launched to load the URL specified. See sdtwebclient(1) for the detail rules of which browser will be used.

To use the Launch service to run a command on the server and displayed back on the client:

1. Get a handle to launching service from infrastructure 'inf'

2. Create a LaunchInfo object with information about this request.

   If the application needs some environment variables been set, we can specify them in the environments parameter in the format of *<key>=<value>*, for example:

   ```
   String[] envs = {"PATH=/usr/bin:/sbin:/usr/ucb/bin",
   "EDITOR=/usr/local/bin/vim" };
   ```

   Environment variables PATH, DISPLAY, HOME, USER are always set by the service. But caller can override their values by explicitly specifying the environment parameter.

3. Launch the command; Launch service uses Authorization service to determine whether current user has enough rights to execute the command.

Sample Code	● [Launching](#)

![Top of Page]

Packaging

Shared interfaces and classes

A service publishes its functionality through one or more public interfaces. The caller of this service needs those interface classes as well. So we encourage developers bundle all the public interfaces and classes they directly reference into an individual jar and registered as a shared library jar across tools and services. The real implemetation of those interfaces can be bundled and registered as a service jar.

Platform Dependent Libraries

Many service implementations need some native library support. You can bundle those native libraries (.so) in the service jar and identify them in the manifest file so they can be included in the library path.

Sample Code	● [Manifest for Native Library](#)

Resource bundles

It is required that all services provide a resource bundle that contains the certain information under predefined message keys:

- **BEANNAME** -- Localized tool name
- **DESCRIPTION** -- Localized description of the tool
- **VENDOR** -- Localized vendor name
- **VERSION** -- Localized version number

Registration

If a service uses classes or resources from some other library jars, you need to register them first, for example:

```
# smcconf repository add library ALL <path>/mylibrary.jar
```

Here we are using the pseudo bean name `ALL` to represent all tools and services.

Next, you register a service jar to the server repository:

```
# smcconf repository add bean <path>/myservice.jar
```

You can check the repository to see the service is successfully imported by doing repository listing:

```
# smcconf repository list
```

See the Registration section for more details regarding the `smcconf` command.

Multiple Implementations of A Common Service

SMC repository supports multiple implementations of one service interface. You can register a syslog service and a WBEM log service that both implement interface `com.sun.management.viper.services.Log`.

Get Service By Name

When a service handle is requested through `getServiceByName()` call to the infrastructure, the repository scans the registered service list, starting from the most recent registered. The first one that implements the given interface and successfully loads itself (`init()` and `start()` methods are successfully called) is returned to the caller.

Configure services with Properties

A service can have several properties set to different values to customize the behavior of itself. The properties are kept in the repository with the service and can be set at registrar time or later on through `smcconf`.

Sample Code	◆ Configure Service with Properties

Debugging

The SMC SDK has a debug utility class, `com.sun.management.viper.util.Debug`. This class acts like a delegate to the real implementation of the output manager interface, VDebug. All service containers will provide an implementation that plug into the class Debug. Services can always call the `trace()` method with proper severity without worrying about what level of detail should be really displayed or how they should be displayed.

Sample Code	◆ [Debugging](#)

The default implementation of Debug in the SMC server can be configured with the environment variable SMC_DEBUG set to 0-2 to print out INFORMATION+, to ERROR+ level messages.

Third-Party Integration

Support for use of other services from third-party frameworks is not currently provided, and may be provided in a future release

Registration

Overview

The SMC object registry is a repository of object information used by the SMC console to configure toolboxes, tools, and services. All tools and services must be registered in the SMC object registry. Tools must be associated with a toolbox before they can appear in an SMC console.

The registry of SMC contains two types of entries: SMC beans and toolboxes. Executable components, like client side GUI/CLI tools, external client providers, server side services, are considered beans and are deployed in the format or jar files. All the beans can have additional library jars attached to them. Toolboxes are XML-based files that describe collections of tool beans and the presentation layout of them. We will only cover the registration part of those created toolboxes and several command line editing commands that shell scripts can use. SMC comes with a GUI toolbox editor that can help administrators to create the toolboxes.

▐▶ *`smcconf` is a command-line tool for administering the SMC registry. It provides the capability to manipulate a toolbox and perform various registry-related tasks. The `smcconf` command is explained more later in this section.*

Registry Basics

The SMC registry contains information about:
- Registered tools and services
- Resource jars, if any, attached to tools and services
- Properties (key/value pairs), if any defined for tools, services, and resource jars

If you want the tool be displayed inside an SMC console, you need to:
- Register the tool
- Add the tool to a specific toolbox

These steps are described more later in this section.

▐▶ *SMC-based tools may or may not refer to backend SMC services. If the tool is dependent on any backend SMC service(s), the service(s) also need(s) to be registered using `smcconf`. Unless the dependent service(s) are registered, SMC will not be able to invoke or display the corresponding tool.*

The name of the tool or service bean can be found in the manifest of the jar file specified on the command line. The bean name can be used later to unregister the tool or service, or as a handle to which libraries/properties may be attached or detached.

---

smcconf

Some of the common object registry tasks you can perform with the `smcconf` tool include:

- [Registering tool and service beans](#)
- [Unregistering tools and services](#)
- [Attaching and detaching resource jars](#)
- [Adding and removing properties](#)
- [Managing toolboxes](#)
- [Listing registered tools/services](#)
- [Registering legacy tools](#)

Each of these tasks is discussed below.

Registering Tool and Service Beans

Registering the tool or service jar file does not remove the jar file from its original location. It simply makes the tool or service usable from within the SMC by adding the information related to the new tool or service to the SMC registry. The command used to register a tool or service bean is:

```
smcconf repository add bean <path>/<jarfile>.jar
```

If the tool or service bean has already been registered, `smcconf` will not allow you to overwrite the existing tool/service bean unless you use the `-f` option as show below.

```
smcconf repository add -f bean <path>/<jarfile>.jar
```

where `<jarfile>.jar` is an existing tool or service bean.

For example, to register `/usr/lib/MyTool.jar`:

```
smcconf repository add -f bean /usr/lib/myTool.jar
```

Any service jar that requires a native library can simply include the native library in the jar file, but also add an entry for the native library to the manifest file, as discussed in [Packaging](#).

Unregistering Tools and Services

Unregistering a tool or service will make it unavailable from within the SMC. It removes the registered tool or service information from the SMC registry. The command used to unregister a tool or service is:

```
smcconf repository remove bean <beanname>
```

where `<beanname>` is a registered tool or service.

For example, to unregister the bean `com.mycompany.myproduct.MyTool`:

```
smcconf repository remove bean com.mycompany.myproduct.MyTool
```

Attaching and Detaching Resource Jars

You can attach or detach resource jars to or from any of the following:

- Tool/Service Beans
- All Services
- All Tools
- All Tools and Services

Any SMC bean can have pluggable libraries attached to it. These library jars can be resource bundles in different locales, as well as function code that needs to be seperately upgradable. Library jars that are attached to specific beans will be visible to that bean only at runtime. However, there are 3 special bean keywords recognized by smcconf that allow you to control the scope of library usage on a wider scale: `ALL` allows the library to be used by all tools and services, `ALLTOOL` allows the library to be used only by other tools, and `ALLSERVICE` allows the library to be used only by other services.

▐▶ *Use the* `-f` *(force) option to override any resource jar attachment. The* `-f` *option can be used while attaching a library jar only.*

- **Attaching Tool/Service beans**

  The command used to attach a resource jar to a tool or service is:

  ```
  smcconf repository add library <beanname> <path>/<jarfile>.jar
  ```

  where `<beanname>` is a registered tool or service and `<jarfile>.jar` is the library resource jar

  For example, to attach a localization library jar `/usr/lib/MyTool_fr.jar` to the already registered bean `com.mycompany.myproduct.MyTool`:

  ```
  smcconf repository add library com.mycompany.myproduct.MyTool
  /usr/lib/MtTool_fr.jar
  ```

- **Detaching Tool/Service beans**

  The command used to detach a resource jar from a tool or service is:

  ```
  smcconf repository remove library <beanname> <jarfile>.jar
  ```

  where `<beanname>` is a registered tool and `<jarfile>.jar` is the resource jar

  For example, to detach the localization library `MyTool_fr.jar` from the tool `com.mycompany.myproduct.MyTool`:

  ```
  smcconf repository remove library com.mycompany.myproduct.MyTool
  MyTool.jar
  ```

- **All tools**

To add a resource jar attachment to be shared by all registered tools only, use the following command:

```
smcconf repository add library ALLTOOL <jarfile>.jar
```

To remove any jar attachment which is shared by all registered tools only, use the following command:

```
smcconf repository remove library ALLTOOL <jarfile>.jar
```

- **All services**

  To add a resource jar attachment to be shared by all registered services only, use the following command:

  ```
  smcconf repository add library ALLSERVICE <jarfile>.jar
  ```

  To remove any jar attachment which is shared by all registered services only, use the following command:

  ```
  smcconf repository remove library ALLSERVICE <jarfile>.jar
  ```

- **All tools and services**

  To add a resource jar attachment to be shared by all registered tools and services, use the following command:

  ```
  smcconf repository add library ALL <jarfile>.jar
  ```

  To remove any jar attachment which is shared by all registered tools and services, use the following command:

  ```
  smcconf repository remove library ALL <jarfile>.jar
  ```

Adding and Removing Properties

You can define and undefine properties (key/value pairs) for any of the following:
- Tool/Service Beans
- All Services
- All Tools
- All Tools and Services
- Specific resource jars in specific tool/service beans

As with library jars, the keywords `ALL`, `ALLTOOL`, and `ALLSERVICE` allow you to control the scope of properties beyond specific beans.

- **Tool/Service Beans**

  Tools and services can have properties associated to their registry entries. To add properties to a registered tool/service, use the commmand below. More than one property could be added at a

time to the specified tool/service as shown below by specifying multiple `-P` *<key=value>* arguments.

```
smcconf repository add property -P HOMEDIR=/tmp -P
MYHOME=/home/kd <beanname>
```

The above command will add two properties (`HOMEDIR=/tmp` and `MYHOME=/home/kd` to the *<beanname>* bean).

To remove properties already defined on the specified registered tool/service, use the following command:

```
smcconf repository remove property -P HOMEDIR -P MYHOME
<beanname>
```

or

```
smcconf repository remove property -P HOMEDIR=/tmp -P
MYHOME=/home/kd <beanname>lt:beanname<beanname>gt;
```

Any one of the above commands can be used for removing the specified properties.

- **All tools**

  To add properties to be shared by all registered tools, use the following command:

  ```
  smcconf repository add property -P HOMEDIR=/tmp ALLTOOL
  ```

  To remove properties shared by all registered tools, use the following command:

  ```
  smcconf repository remove property -P HOMEDIR=/tmp ALLTOOL
  ```

- **All services**

  To add properties to be shared by all registered services, use the following commmand

  ```
  smcconf repository add property -P HOMEDIR=/tmp ALLSERVICE
  ```

  To remove properties shared by all registered services, use the following commmand:

  ```
  smcconf repository remove property -P HOMEDIR=/tmp ALLSERVICE
  ```

  or

  ```
  smcconf repository remove property -P HOMEDIR ALLSERVICE
  ```

- **All tools and services**

  To add properties to be shared by all registered tools and services, use the following command:

  ```
  smcconf repository add property -P HOMEDIR=/tmp ALL
  ```

  To remove properties shared by all registered tools and services, use the following command:

  ```
  smcconf repository remove property -P HOMEDIR=/tmp ALL
  ```

or

```
smcconf repository remove property -P HOMEDIR ALL
```

- **Specific resource jar in a specific tool/service bean**

  It is also possible to add/remove properties to a specified resource jar attachment to a specified tool/service bean; for example, `CronTool.client.VCronTool` tool has two resource jar attachments: `CronTool_C.jar` and `CronHelpSet.jar`.

  To add properties to the `CronHelpSet.jar` resource jar attachment only, use the following command:

```
smcconf repository add property -P HOMEDIR=/tmp \
CronTool.client.VCronTool \
CronHelpSet.jar
```

Managing Toolboxes

- **Folders**

  To create a folder in a toolbox:

```
smcconf toolbox add [-f] folder "Devices" [-F "Hardware] "Device
Mgt. Tools" \
smallDevice.gif largeDevice.gif -B /home/user/myToolbox.tbx
```

  The above command will create the "Devices" folder, as a subfolder inside the "Hardware" folder, in the toolbox file `/home/user/myToolbox.tbx`. The specified icons and description of "Device Mgt. Tools" will be be used to represent the folder in the SMC console.

  To remove a folder from a toolbox:

```
smcconf toolbox remove folder "Devices" -B
/home/user/myToolbox.tbx
```

- **Tools**

  To add a tool to a toolbox:

```
smcconf toolbox add [-f] tool <bean_class_name> [-D <domain>] [-H
<host>:<port>] \
[-F <parent_folder>lt:parent_folder<parent_folder>gt;] [-B
<toolbox_file>] [-P <key>:<value>]
```

  To remove a tool from a toolbox:

```
smcconf toolbox remove tool ...
```

- **Links to other toolboxes**

To add a link to another toolbox:

```
smcconf toolbox add [-f] tbxURL
<toolbox_url>lt:toolbox_url<toolbox_url>gt; [-F <parent_folder>]
[-B <toolbox_file>]
```

Copy external toolboxes to SMC server:

```
smcconf toolbox add toolbox <toolbox_url> -B <toolbox_file>
```

Listing Registered Tools/Services

To list the contents of registered tools/services/attachments/properties, use the following command:

```
smcconf repository list
```

This command lists the following information:

- Properties defined for all tools and services
- Properties defined for all tools only.
- Properties defined for all services only.
- Resource jars/shared libraries attached to all tools and services
- Resource jars/shared libraries attached to all services
- Registered services; for each registered service, the following is displayed:
    - Native libraries
    - Properties
    - Resource Jars/Shared Libraries attachments and properties, if any are defined on them.
    - Resource jars/shared libraries attached to all tools
- Registered tools (no legacy tools are listed here); for each registered tool, the following is displayed:
    - Native libraries
    - Properties
    - Resource Jars/Shared Libraries attachments and properties, if any aredefined on them.

Registering Legacy Tools

Any CLI (Command Line Interface) or XAPP (X Applications) tool can also be registered with the SMC registry. This will allow the SMC to invoke the corresponding CLI/XAPP tool from within the console; for example:

```
smcconf toolbox add legacy -N "Ls Tool" -T CLI -E /usr/bin/ls
-P " -alR "
smcconf toolbox add legacy -N "Clock" -T XAPP -E
/usr/openwin/bin/xclock
```

For more information, please refer to the `smcconf` man page.

Frequently Asked Questions

- How Much of My Existing Code Can I Salvage?
- How Do I Load Images, ResourceBundles, and Online Help?
- How Do I Access the Console Frame Parent for Dialogs?
- How Do I Access the Selection Set in the Results Pane?
- How Do I Set Selections in the Results Pane?
- How Do I Get the Currently Selected Node in the Navigation Pane?
- How Do I Specify System Properties on the Console Command Line?
- How Do I Update the Console Display to Show Changes in the Data Model?
- How Do I Integrate Menubars and Toolbars?
- How Do I Create Dialogs?
- How Do I add an About Box for my Tool?
- How Do I Save and Restore User Preferences?
- How Do I Manage Sorting Preferences?
- How Do I Customize the First Column Header in Details View?
- How Do I Align Column Values in Details View?
- Can I Use Global Static Variables
- Why Can't SMC find my new jar file I just registered?
- I Re-Registered My `jar` File, But I Get a ClassNotFoundException at Runtime.

How Much of My Existing Code Can I Salvage?

See the section Migrating Applications to SMC.

How Do I Load Images, ResourceBundles, and Online Help?

See the section Accessing Resources.

How Do I Access the Console Frame Parent for Dialogs?

If you descend from one of the AWT or Swing top-level windows (for example, `Window`, `Frame`, `Dialog`, `JFrame`, `JDialog`), you limit the contexts in which your code will run, which is often not

required. Using the `VConsoleProperties.DIALOGTYPE` property will do the following:

- Allows your tool to know the environment it is in, either `FRAME` or `INTERNALFRAME`.

- Lets you know the type of top level container in which to place your component; for example, `JFrame/JDialog` or a `JinternalFrame`.

If the `DIALOGTYPE` setting is `FRAME`, (SMC style), you need the parent frame to create dialogs. If the `DIALOGTYPE` setting is `INTERNALFRAME` (as in the "desktop" console), you need the Swing desktop pane to add your `JInternalFrame`.

Sample Code	⬢ [Console Frame Parent for Dialogs](#)

How Do I Access the Selection Set in the Results Pane?

Obtain a handle to the `VDisplayModel` and call `getSelectedNodes()`.

Sample Code	⬢ [Getting Selections in Results Pane](#)

How Do I Set Selections in the Results Pane?

There several methods for effecting the selection set, depending on the what you want to do. Each require obtaining a handle to the `VDisplayModel`.

Sample Code	⬢ [Setting Selections in Results Pane](#)

How Do I Get the Currently Selected Node in the Navigation Pane?

Obtain a handle to the `VDisplayModel` and call `getSelectedNavigationNode()`.

Sample Code	⬢ [Get Selected Navigation Pane Node](#)

How Do I Specify System Properties on the Console Command Line?

```
smc J-Dname1=value1 -J-Dname2=value2 ...
```

This is convenient if you want to dynamically effect application behavior without having to change code. However, your application will by default only have access to the standard properties (or whatever access is permitted by the security policy in effect).

To grant `read` permission for application-specific properties, you can create a policy file in your home directory (`.java.policy`).

Sample Code	● [Specify Properties on Command Line](#)

How Do I Update the Console Display to Show Changes in the Data Model?

Send an `UPDATESCOPE` event to the console.

Sample Code	● [UPDATESCOPE](#)

How Do I Integrate Menubars and Toolbars?

You can specify your own menubar and toolbar on a per-navigation-node basis, via the `VScopeNode` constructor. You should also do this for the root node of each internal-root model. Your menu/toolbar will appear to the right of the console's menu/toolbar.

See the [Menu bar](#) section for details on how to integrate your menu items with the console menus.

How Do I Create Dialogs?

Extend `VOptionPane` to create your own dialogs. When you need to display it, create a `VDialog` or `VFrame`, place the `VOptionPane` inside it and then display your `VDialog` or `VFrame`.

Sample Code	● [Creating a Dialog](#)

How Do I add an About Box for my tool?

First integrate an **About** menu item into the Help menu, as discussed in the [Menu bar](#) section. When you receive the event (in the `ActionListener` for the menu item) associated with this menu item, instantiate a `VAboutBox` dialog, set the title and description information relative to your tool, and display the dialog.

Sample Code	● [About Box](#)

How Do I Save and Restore User Preferences?

See the section VConsoleProperties.

How Do I Manage Sorting Preferences?

Your tool is responsible for tracking what the sort attribute (column identifier) and sort order (ascend or descend) are, and saving them as a property. Sort preferences are in `[+/-]#` format, where:

- `+` implies ascending sort order
- `-` implies descending sort order
- `#` is the column number to sort by

For example, `+2` means to sort column 2 in ascending order, `-1` means to sort column 1 (the first column) in descending order.

Sample Code	● Manage Sorting Preferences

How Do I Customize the First Column Header in Details View?

By default, SMC assigns the title for the first column in details view to be `Name`. You can use the `VConsoleProperties.DEFAULTCOLUMNHEADER` property to customize the title.

If your tool is not limitted to the Details view only, then at the same time you customize the first column, you might also want to customize the width of the column grid for the other views using `VConsoleProperties.DEFAULTCOLUMNWIDTH`. Specifiy the value as a pixel width in String format.

Sample Code	● Customize first Column Header

How Do I Align Column Values in Details View?

See `VScopeNode.columnHeaders` in the SMC SDK Javadocs for detailed information about how to specify column headers and their widths.

Specifying alignment involves adding fixed values to the desired column widths:

- Width values `>20000` will be right-aligned.
- Width values `>10000` and `<20000` will be center-aligned.
- Width value `<10000` will be left-aligned.

Sample Code	◆ [Align Column Values](#)

Can I Use Global Static Variables?

The short answer is no. Depending on user-specific console configuration, your application may be instantiated more than once, and no variable should have scope beyond the instance for which it was originally set, otherwise you may get very strange results. Furthermore, you cannot use VConsoleProperties for per-application global data because the `Properties` object is a console-wide shared object.

Why Can't SMC find my new jar file I just registered?

Newly registered jar files require that the SMC server be restarted. See the Starting Services section for how to do this.

I Re-Registered My `jar` File, But I Get a ClassNotFoundException at Runtime.

This is a common error during development, typically after you've added a new class or new properties file. It is especially common during GUI development, where named and anonymous inner classes are used quite frequently, and often without the developer even realizing that this results in additional `.class` files. New classes and properties files require that the SMC server be restarted. See the Starting Services section for how to do this.

Code Samples

This page provides links to the code samples used in numerous places in this guide. Code samples are listed below in alphabetical order. Numbers in this list are provided for ease of reference only, and do not refer to the order in which the code is presented in the guide.

1. About Box
2. Accessing a Log Service
3. Add Child Node
4. Align Column Values
5. Build Search Index
6. Call Delegation
7. Checking Authorization
8. Configure Services with Properties
9. Connect to External Client Provider
10. Console Listeners
11. Create Tool Node
12. Creating a Dialog
13. Customize 1st Column Header
14. Debugging
15. Details Style Only
16. Enable Styles
17. Exceptions
18. External Client Provider
19. Get Frame Parent
20. Getting Selected Navigation Pane Node
21. Getting Selections
22. Getting Sort Preferences
23. Hello
24. Helpset Map File
25. Hyperlink to Helpset
26. Launching
27. Loading Help Files
28. Loading Images

29. [Loading Resource Bundles](#)

30. [Localized Helpset](#)

31. [Log Console Event](#)

32. [Logging](#)

33. [Manifest for External Client Provider](#)

34. [Manifest for Native Library](#)

35. [Manifest for Tools](#)

36. [Menubar Integration](#)

37. [Messaging](#)

38. [native2ascii](#)

39. [Persistence](#)

40. [Preferences](#)

41. [PropertyChangeListener](#)

42. [Remove Child Node](#)

43. [Scope](#)

44. [Service Descriptor](#)

45. [Service Implementation](#)

46. [Service Interface Definition](#)

47. [Set Center Status Info Pane](#)

48. [Set Context Help](#)

49. [Set Left Status Info Pane](#)

50. [Setting Character Set Encoding in HTML](#)

51. [Setting Selections](#)

52. [System Properties on Command Line](#)

53. [Tool Descriptor](#)

54. [Tool Resource Bundle](#)

55. [Tool.setProperties](#)

56. [Update Display with UPDATESCOPE](#)

57. [UPDATESCOPE](#)

58. [VConsoleActionListener](#)

🖼 Illustrations Used in this Guide

This page provides links to the illustrations used in this guide. Illustrations are listed below in alphabetical order. Numbers in this list are provided for ease of reference only, and do not refer to the order in which the illustrations are presented in the guide.

📖 Glossary

Click on the term for which you want a definition. Click on the up arrows (🔼) to return to the top of this list.

- [authentication](#)
- [authorization](#)
- [CIM](#)
- [console](#)
- [event bus](#)
- [infrastructure](#)
- [JavaBean](#)
- [launch](#)
- [logging](#)
- [Look and Feel](#)
- [messaging](#)
- [properties](#)
- [RBAC](#)
- [registration](#)
- [RMI](#)
- [service](#)
- [Sun Management Center](#)
- [tool](#)
- [Tool class](#)
- [Tool Descriptor class](#)
- [toolkit](#)
- [VConsoleActionListener class](#)
- [VConsoleProperties class](#)
- [VScopeNode class](#)
- [WBEM](#)

⇧ authentication

The service that is used to verify a user's login credentials, such as a username and password.

⇧ authorization

The service that is used to decide whether an action towards some critical system resource is to be allowed or denied, based on the security policy currently in effect. See Authorization for more information.

⇧ CIM

Common Information Model; a set of templates (schemas) specifying a data format for enterprise management information that is independent of platform and management application.

⇧ console

A container for SMC client tools; the SMC "desktop" from which users perform management tasks.

⇧ event bus

In SMC, a chain that allows components to create, send, and listen for events to or from other components in the console. Every component in the console is given a reference to these properties and is added to the event bus; components then can get and set properties to effect behavior. They may also send and receive events which may/may not correspond directly with user interaction.

Event Bus

⇧ infrastructure

Collective term for the object model, communications protocols, platform-specific APIs, and core services used as the "glue" layer between client and server components; for example, native SMC tools and services used JavaBeans communicating over RMI, with server components having some Solaris-specific dependencies, and using the core SMC authentication, authorization, security, messaging, logging, preferences, and launch services.

⬆ JavaBean

A portable, platform-independent reusable component model; native SMC tools are written as sets of JavaBeans, while SMC services are often written as a combination of JavaBeans and platform-specific code.

⬆ launch

The act of starting a computer application. With respect to SMC, this refers to the service for launching legacy (non-SMC aware) applications. See Launching for more information.

⬆ logging

The service for posting and tracking messages that pertain to important system events. See Logging for more information.

⬆ Look and Feel

A pluggable user interface component in the SMC system; SMC includes a default Explorer-like look and feel for the Solaris Management tools, with a tree view on the left, a results pane on the right, and an information pane on the botton. See UI Components for more information.

⬆ messaging

A mechanism for exchanging messages between 2 or more clients. See Messaging for more information.

⬆ properties

Named values that effect application behavior and/or presentation, and which can persist from one session to the next. Properties are managed by the VConsoleProperties class.

⬆ RBAC

Determining the authorization for an access request by mapping to an attribute of the requestor, such as membership in a group, job function, or organizational level, rather than on the individual's unique identity; assumes that a person will take on different roles over time, and different responsibilities in relation to IT systems; access control based on specific rules relating to the nature of the subject and object, beyond just their identities.

⬆ registration

The process by which a tool or service is made known to the SMC console. All tools and services must be registered in the SMC object registry and associated with a toolbox before they can appear in an SMC console.

▌▷ *smcconf is a command-line tool to administer the application registry. It provides the capability to manipulate the toolbox and perform registry-related tasks.*

See Registration for more information.

⬆ RMI

Remote Method Invocation; a distributed object model for communication between Java programs, in which the methods of remote objects written in Java can be invoked from other Java virtual machines, possibly on different hosts. RMI is the native communications model used by SMC tools and services.

⬆ service

Server-side applications that support SMC tools; native SMC services are generally a combinatation of Java and platform-specific code.

⬆ SunMC

Sun Management Center; an open, extended, standards-based server monitoring and management solution that uses JavaTM and SNMP protocols to provide an integrated and comprehensive enterprise-wide management of Sun server products and their subsystems, components, and peripheral devices.

⬆ tool

Client-side applications; in SMC, all tools are written as sets of JavaBeans.

⬆ `Tool` class

The top-level client class instantiated by the Console; the main interface that SMC clients must implement. See Tool for more information.

⬆ `Tool Descriptor`

Provides information to represent a tool without actually instantiating the tool. See Tool Descriptor for more information.

⬆ toolkit

Collections of 📖 tools associated with a given user, group, or administrative role. Toolkits are defined with *toolkit properties files*, which specify tool names, locations, and managed scope.

⬆ `VConsoleActionListener` class

Provides the interface through which tools can be notified about various events in the system. See VConsoleActionListener for more information.

⬆ `VConsoleProperties` class

Shared properties object used by all components in an SMC system for property storage. See VConsoleProperties for more information.

⬆ `VScopeNode` class

The most common (and arguably the most important) SMC tool class; provides information (icons, column headers, payload, etc.) about a data model to the console, whether that information is rendered in the left-side navigation pane, or the right-side results pane. See VScopeNode for more information.

⬆ WBEM

Web-Based Enterprise Management; standard for defining platform-independent management information across platforms; initiated by the Distributed Management Task Force (DMTF) to define a Common Information Model (CIM), and further refined by Sun Microsystems; management information is made available to management applications via eXtensible Markup Language (XML) over the common Web protocol HyperText Transport Protocol (HTTP).

Index

Contents

Index

Sample Code: About Box

```
// Create About Box, setting title and description
//
VAboutBox aboutBox = new VAboutBox();
aboutBox.setTitle("My Tool 1.0");
aboutBox.setDescription(
    "Long-winded Copyright notice\nthat only a lawyer can comprehend");

// Add some extra space below the copyright text, otherwise
// the default icons at the bottom will crop some of the text.
// The space we add must be relative to the current font.
//
Dimension d = aboutBox.getMinimumSize();
FontMetrics fm = aboutBox.getFontMetrics(aboutBox.getFont());
d.height += (2 * fm.getHeight());
aboutBox.setMinimumSize(d);

// Create container for About box
//
JFrame consoleFrame =
(Jframe)(properties.getPropertyObject(VConsoleProperties.FRAME));
VDialog container = new VDialog(consoleFrame, true);
aboutBox.setContainer(container);

// Set title for container
//
container.setTitle("About My Tool");

// Put it all together and render
//
container.getContentPane().setLayout(new BorderLayout());
container.getContentPane().add(aboutBox, BorderLayout.CENTER);
container.pack();
container.showCenter(consoleFrame);
```

Sample Code: Accessing a Log Service

```
method1 () {
     ...
     Log logsvc = (Log)
infrastructure.getServiceByName("com.mycompany.myproduct.MyLogService");
     logsvc.writeLog(...);
}
```

The `infrastructure` handle is given to the client upon loading by the SMC console.

Add Child Node

```
VScopeNode parent = ...

...

VScopeNode child = new VScopeNode(null, null, null,
        myMenuBar, myToolBar, myPopupMenu,
        smallIcon, largeIcon, "Child Node",
        "A child node for some parent",
        null, -1, childDataObject);

// Associate the node with our Tool's instance.
// This allows node selection notification to be handled by the
// console's engine thru the Tool's start/atop methods.
child.setTool(myTool);

// Add the node as a child of the parent
parent.add(child)

...

// Notify console that Navigation pane should be updated.
// (Assumes a general method for firing events).
VConsoleEvent ev = new VConsoleEvent(
     myTool, VConsoleActions.UPDATESCOPE, parent);
fireConsoleAction(ev);
```

Sample Code: Align Column Values

```java
private final Object[][] columnHeaderConfig = {
    // Column key and column width in characters units
    {"My Column 1", new Integer(20)},    // First column, left-aligned
    {"My Column 2", new Integer(10013)}, // Second column, center-aligned
    {"My Column 3", new Integer(20015)}, // Third column, right-aligned
    ...
};

int nCols = columnHeaderConfig.length;
String[][] columnHeaders = new String[nCols][3];

// Get FontMetrics for header. Since we don't have access to the
// header component, create a dummy component that uses the same
// font as the header. Then get the FontMetrics for the dummy
// component.
//
JLabel dummy = new JLabel();
dummy.setFont(ResourceManager.labelFont);
FontMetrics fmHeader = dummy.getFontMetrics(ResourceManager.labelFont);

// Do the same to get FontMetrics for the data
//
dummy.setFont(ResourceManager.bodyFont);
FontMetrics fmData = dummy.getFontMetrics(ResourceManager.labelFont);
for (int i = 0; i < nCols; i++) {
    // Get actual header string
    columnHeaders[i][0] = (String)columnHeaderConfig[i][0];

    // First compute the width of the localized column header.
    // Note that this includes a 2-character margin on each
    // side, based on the character 'A'.
    //
    int headerWidth = fmHeader.stringWidth(columnHeaders[i][0]);
    headerWidth += fmHeader.stringWidth("AAAA");

    // Extract the alignment value from the column width:
    // width values > 20000 -> right aligned
    // width values > 10000 -> center aligned
    // width values > 0     -> left aligned
    //
    int columnWidth = ((Integer)columnHeaderConfig[i][1]).intValue();
    int alignmentValue = 0;
```

```
    if (columnWidth > 20000) {
        alignmentValue = 20000;
    } else if (columnWidth > 10000) {
        alignmentValue = 10000;
    }
    columnWidth -= alignmentValue;

    // Then compute the preferred width of the column's data.  This too,
    // is based on the character 'A'.
    //
    int dataWidth = fmData.stringWidth("A");
    dataWidth *= columnWidth;
    dataWidth += alignmentValue;

    // Actual width is max of header/data width, but in String format.
    columnHeaders[i][1] = new String(
        String.valueOf(Math.max(headerWidth, dataWidth)));
}

...

// The columnHeaders object array can then be set on a VScopeNode,
// whether it is the internal root for your data model, or a node
// in the navigation pane.
node.setColumnHeaders(columnHeaders);
```

Sample Code: Build Search Index

To build the search index for a helpset for a specified `locale` and place it in the `SearchIndex` subdirectory, assuming the helpset is rooted at `${HOME}/helpset`, and the html files are in the `html` subdirectory:

```
setenv JHHOME <path where JavaHelp 1.1 is installed>
setenv JAVA_HOME <path where the JDK is installed>
setenv PATH ${JAVA_HOME}/bin:${PATH}
cd ${HOME}/helpset
rm -rf locale/SearchIndex
${JHHOME}/javahelp/bin/jhindexer -locale locale -db locale/SearchIndexlocale
locale/html/*
```

Sample Code: Call Delegation

```java
        // pass on caller identity to other service
        // by specifying delegation to true
        OtherService os = (OtherService)
infra.getServiceByName(OtherService.class.getName(), true);
        ...
        os.doSomething();
        ...
```

Sample Code: Checking Authorization

```
    // This example illustrates how to check authorizations
    // for a service that supports read and write authorizations


    public static final String AUTH_MYSERVICE_WRITE =
"solaris.admin.myservice.write";
    public static final String AUTH_MYSERVICE_READ  = "solaris.admin.myservice.read";

    PermissionCollection permissionCollection = null;
    ToolInfrastructure infrastructure = <gotten from SMC>;

    // Get authorizations.
    try {
        Authorization auth = (Authorization)infrastructure.getServiceByName(
                    ServiceList.AUTHORIZATION);
        permissionCollection = auth.readUserPermissions(
                    infrastructure.getIdentity());
    } catch (Exception ex) {
        // Report exception
    }

    ...



    /**
     * Determine if user is authorized for "write" access.
     *
     * @return  true if user has write authorization, otherwise false
     */
    public boolean hasWriteAuthorization() {

        // Allow only if explicitly authorized.
        //
        VPermission perm = new VPermission(AUTH_MYSERVICE_WRITE);
        if ((permissionCollection != null)
                && permissionCollection.implies(perm))
            return true;

        // Otherwise, deny
        return false;

    } // hasWriteAuthorization


    /**
     * Determine if user is authorized for "read" access.
     *
     * @return  true if user has read authorization, otherwise false
     */
```

```
public boolean hasReadAuthorization() {

    // Allow only if explicitly authorized.
    //
    VPermission perm = new VPermission(AUTH_MYSERVICE_READ);
    if ((permissionCollection != null)
            && permissionCollection.implies(perm))
        return true;

    // Otherwise, deny
    return false;

} // hasReadAuthorization
```

Sample Code: **Configure Services with Properties**

```
# Set the architecture property when adding a service
smcconf repository add bean MyService.jar -P ARCH=`uname -p`

# Or change a property after the service is already installed
smcconf repository add property com.mycompany.myproduct.MyServiceImpl -P PORT=8080
```

At runtime, the service retrieves the properties through the ServiceContext that is given to the service via the method setContext():

```
import com.sun.management.viper.VService;
public class MyServiceImpl extends VService implements MyService {
    ...
    public void init() {
        super.init();
        ServiceContext context = super.getContext();
        String arch = context.getRegistryProperty("ARCH");
        String port = context.getRegistryProperty("PORT")
        ...
    }
```

Sample Code: **Connect to External Client Provider**

```
ToolInfrastructure tinf;  // Set by SMC console

// Get external client proxy reference for WBEM.
// We pass the target host in the name space parameter.
CIMNameSpace cns = new CIMNameSpace(hostname, "root/cimv2");
Integer protocol = new Integer(CIMClient.RMI);
Object [] params = {cns, protocol};
CIMClient cc = (CIMClient)tinf.getExternalClient("CIMWBEM", params);
// Access providers through this CIMClient
...
```

Sample Code: Console Listeners

```java
public class VProcMgr implements Tool, VConsoleActionListener {

    private Vector consoleListeners = new Vector();

    ...

    /**
     * Adds the specified console actions listener to receive events for actions
     * by our subcomponents.
     *
     * @param listener  the console action listener to forward events to
     */
    public void addConsoleActionListener(VConsoleActionListener listener) {

        if (listener != null)
            consoleListeners.addElement(listener);

    } // addConsoleActionlistener


    /**
     * Notify all registered listeners of the specified console event.
     *
     * @param e     the console action event
     */
    public void fireConsoleAction(VConsoleEvent e) {

        for (int i = 0; i < consoleListeners.size(); i++) {
            VConsoleActionListener l = (VConsoleActionListener)
                consoleListeners.elementAt(i);
            l.consoleAction(e);
        }

    } // fireConsoleAction

    ...

    VConsoleEvent ev = new VConsoleEvent(...);
    fireConsoleAction(ev);

    ...
}
```

Create Tool Node

```
public class MyTool implements Tool ... {

    VScopeNode myToolNode;

    ...

    // Notice how we leave the first 3 parameters as null.
    // This means the console's rendering engine will take
    // over for us and handle the rendering of our child
    // nodes.
    myToolNode = new VScopeNode(null, null, null,
        myMenuBar, myToolBar, myPopupMenu,
        smallIcon, largeIcon, "My Cool Tool",
        "A tool that does cool stuff",
        null, -1, myDataObject);

    // Associate the node with our Tool's instance.
    // This allows node selection notification to be handled by the
    // console's engine thru the Tool's start/atop methods.
    MyToolNode.setTool(this);

    ...

    public void getScopeNode() {
        // Return the root node of our data model
        return myToolNode;
    }

    ...
```

Sample Code: Creating a Dialog

```
public class MyDialog extends VOptionPane {

    getContentPane().setLayout(...);
    ... add components ...

    VFrame container = new VFrame();
    setContainer(container);
    container.setTitle(...);
    ...

    JFrame f = (JFrame)(properties.getPropertyObject(VConsoleProperties.FRAME));
    container.showCenter(f);

}
```

Sample Code: Creating a Dialog

Sample Code: Customize 1st Column Header

```
Properties properties = ...

// Header for the default column in details view.
properties.setPropertyObject(
    VConsoleProperties.DEFAULTCOLUMNHEADER,
    "My Column");

// Pixel width of the column grid during large/small icon view.
// Typically, you wouldn't hardcode the value as shown here, but
// should compute the value based on the font used in the View
// pane.
properties.setProperty(
    VConsoleProperties.DEFAULTCOLUMNWIDTH,
    "28");
```

Sample Code: Debugging

```
import com.sun.management.viper.util.Debug;

public class MyServiceImpl extends VService implements MyService {
    ...
    public void doit() throws RemoteException {
        try {
            ...
        } catch (Exception ex) {
            Debug.trace("MyServiceImpl",
                Debug.ERROR, "Exception during doit() ",
                ex);
        }
    }
}
```

Details Style Only

```
public class MyTool implements Tool ... {

    String style = "";
     ...

    public void start() {
        ...

        // Save the current style set in the console
        style = properties.getProperty(VConsoleProperties.ICONSTYLE);

        // Since we're only allowing one view, we need to disable
        // style menu items, so user can't change style
        properties.setProperty(VConsoleProperties.ICONVIEWSENABLED,
                   VConsoleProperties.FALSE);

        // Set style property for Details only
        properties.setProperty(VConsoleProperties.ICONSTYLE,
                   VConsoleProperties.DETAILS);
        ...
    }

    public void stop() {
        ...

        // Reset style back to original setting
        properties.setProperty(VConsoleProperties.ICONSTYLE, style);

        ...
    }

    ...
```

Enable Styles

```
// Enable on LARGE ICON and DETAILS styles
properties.setProperty(VConsoleProperties.ICONVIEWSENABLED,
        VConsoleProperties.LARGE + VConsoleProperties.DETAILS);
```

Sample Code: Exceptions

```java
public class MyException extends VException {

    // Resource class for Exceptions.properties.
    private static final String RESOURCECLASS =
        "com.mycompany.myproduct.mytool.resources.Exceptions";


    ...

    /**
     * Protected methods to return the base name of the resource
     * bundle property file.
     */
    protected String getBundleName() {
        return RESOURCECLASS;
    }


    /**
     * Protected method to return the ClassLoader for this class.
     */
    protected ClassLoader getResourceClassLoader() {
        try {
            return this.getClass().getClassLoader();
        } catch (Exception e) {
            return ClassLoader.getSystemClassLoader();
        }
    }
}
...

try {
    if (some error)
        throw new MyException("errorKey");
} catch (Exception e) {
    System.out.println("Error doing something.  Exception msg is "
        + e.getLocalizedMessage());
}
```

Sample Code: **External Client Provider**

```java
public class CIMClientProvider implements ExternalClientProvider {
    private static String myType =
"com.sun.management.viper.client.ExternalClientList.CIMWBEM";

    public Object getExternalClient(
        String xcType,
        String host,
        String user,
        String credential,
        String role,
        String roleCredential,
        Object[] params) throws Exception {

        if (!xcType.equals(myType))
            throw new VException("Unknown xc type");

        // validate parameter array skipped

        CIMNameSpace ns = (CIMNameSpace)params[0];
        SolarisUserPrincipal up = new SolarisUserPrincipal(user, role);
        SolarisPasswordCredential pc = new SolarisPasswordCredential(
                credential, roleCredential);

        return new CIMClient(ns, up, pc);
    }
}
```

Sample Code: Accessing the Frame Parent

Assuming you already have a reference to the 📖 VConsoleProperties object:

```
String dialogType = (String)(properties.getProperty(VConsoleProperties.DIALOGTYPE));
if ((dialog.Type == VConsoleProperties.FRAME) {
    JFrame frame =(JFrame)(properties.getPropertyObject(VConsoleProperties.FRAME);
    ...
} else if ((dialog.Type == VConsoleProperties.INTERNALFRAME) {
    JDesktopPane p =
(JDesktopPane)(properties.getPropertyObject(VConsoleProperties.DESKTOPPANE));
    ...
}
```

Sample Code: Getting Selected Navigation Pane Node

Assuming you have a reference to the VConsoleProperties object:

```
VDisplayModel model =
(VDisplayModel)(properties.getPropertyObject(VConsoleProperties.DISPLAY);

VScopeNode node = model.getSelectedNavigationNode();
```

Sample Code: Getting Selected Navigation Pane Node

Sample Code: Getting Selections in Results Pane

Assuming you already have a reference to the ▣ VConsoleProperties object:

```
VDisplayModel model =
(VDisplayModel)(properties.getPropertyObject(VConsoleProperties.DISPLAY);

Vector vSelected = model.getSelectedNodes();
```

where `vSelected` is a Vector of `VScopeNode` objects.

Sample Code: Getting Selections in Results Pane

Sample Code: Getting Sort Preferences

```java
String sortPreferencesKey = getClass().getName() + ".sortPreferences";
Properties properties = ...

// Save the current sort properties as a preference.
properties.setProperty(sortPreferencesKey,
    properties.getProperty(VConsoleProperties.SORTEDCOLUMN));


...

// Get previously saved preferences.  If none exist, then
// presumably the user disabled sorting in the previous
// session, and we should honor that.
//
String sortPreferences = properties.getProperty(sortPreferencesKey);
if ((sortPreferences != null) && !sortPreferences.equals("null")) {

    // Sort preferences are in "[+/-]#" format, where:
    //     + implies ascending sort order
    //     - implies descending sort order
    //     # is the columns number to sort by

    // Extract the sort order
    String sortOrder = VConsoleActions.SORTUP;
    if (sortPreferences.indexOf('-') >= 0)
        sortOrder = VConsoleActions.SORTDOWN;

    // Extract the sort column
    Integer[] sortColumn = new Integer[1];
    try {
        int n = Integer.parseInt(sortPreferences.substring(1));
        sortColumn[0] = new Integer(n);
    } catch (Exception e) {
        // Should never get here, but Murphy's Law...
        sortColumn[0] = new Integer(0);
    }

    // Apply sort criteria to display model
    VConsoleEvent e = new VConsoleEvent(myTool, sortOrder, sortColumn);
    myTool.fireConsoleAction(e);

    // Update console UI controls on applied sort criteria
    properties.setProperty(VConsoleProperties.SORTEDCOLUMN, sortPreferences);
}
```

Sample Code: Hello

Sample code is displayed in this window.

```
/**
 * Comments? Questions?
 */
```

Sample Code: Hello

Sample Code: Helpset Map File

This shows an example helpset map file, where some of the HTML files reside in the `topics/topicA` subdirectory of the helpset.

```xml
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE map PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp Map Version 1.0//EN"
"http://java.sun.com/products/javahelp/map_1_0.dtd">

<map version="1.0">
  <mapID target="mytool_topicA_coolstuff_html" url="topics/topicA/coolstuff.html" />
  <mapID target="mytool_topicA_hotstuff_html" url="topics/topicA/hotstuff.html" />
    ...
</map>
```

Sample Code: Hyperlink to Helpset

```
public class MyDialog extends VOptionPane {

    public MyDialog() {
        ...

        final ... app = <handle to some class that has the event routing method>
        addConsoleActionListener(
            new VConsoleActionListener() {
                public void consoleAction(VConsoleEvent e ) {
                    if (e.getID().equals(VConsoleActions.HYPERLINKEVENT) &&
                        myLinkListener.isExternalLink((String)(e.getPayload()))))

                        app.fireConsoleAction(e);
                }
            }
        });

        ...
    }
    ...
}
```

Sample Code: Launching

```
// Example code for launching the Motif application 'wsinfo'

import com.sun.management.viper.services.Launch;
import com.sun.management.viper.services.LaunchInfo;
import com.sun.management.viper.services.ServiceList;
...
Launch launcher = (Launch)inf.getServiceByName(ServiceList.LAUNCH);
          LaunchInfo wsinfo = new LaunchInfo(
          "/usr/openwin/bin/wsinfo", // application path
          LaunchInfo.APP_TYPE_XAPP,  // type null
          null                       //,
          environments );
try {
    launcher.launch(wsinfo);
} catch (LaunchException le) {
    // problems like command not found, no display
} catch (AuthorizationException ae) {
    // current user has no authorization to launch this command
} catch (RemoteException re) {
    // Other connection problem
}
```

Sample Code: Loading Help Files

```
bundle = ResourceManager.getLocalizedTextFile(
        "html/addUserHelp.html",
        toolClass);
```

where `toolClass` is a class object that has the same codebase as the HTML file to be loaded. It is typically -- but not always -- the class object of your main `Tool` instance (`myTool.getClass()`). For example, a project-wide common dialog, subclassed from `JDialog`, that exists in a library jar file could pass `this.getClass()`.

Sample Code: **Loading Images**

```
imageIcon = ConsoleUtility.loadImageIcon(
        "images/foobar.gif",
        toolClass );
```

where `toolClass` is a class object that has the same codebase as the icon image to be loaded. It is typically -- but not always -- the class object of your main `Tool` isntance. For example, a project-wide common dialog, subclassed from `JDialog`, that exists in a library jar file could pass `this.getClass()`.

Sample Code: Loading Resource Bundles

```
ResourceBundle bundle = ResourceManager.getBundle(
        "com.sun.product.foomgr.client.resources.Resources",
        this.getClass() );
```

where `this` is a handle to your main `Tool` instance.

Sample Code: Localized Helpset

This shows an example helpset file localized for the French (fr) locale.

Note how references to other files in this helpset are based on the locale-based subdirectory name.

```xml
<?xml version='1.0' encoding='ISO-8859-1' ?>
<!DOCTYPE helpset PUBLIC "-//Sun Microsystems Inc.//DTD JavaHelp HelpSet Version
1.0//EN" "http://java.sun.com/products/javahelp/helpset_1_0.dtd">

<helpset version="1.0">
  <title>My Tool Help</title>
   <maps>
     <homeID>about_my_tool_html</homeID>
     <mapref location="fr/map.jhm" />
  </maps>

  <view>
    <name>TOC</name>
    <label>Table of Contents</label>
    <type>javax.help.TOCView</type>
    <data>fr/toc.xml</data>
  </view>

  <view>
    <name>Index</name>
    <label>Index</label>
    <type>javax.help.IndexView</type>
    <data>fr/index.xml</data>
  </view>

  <view>
    <name>Search</name>
    <label>Search</label>
    <type>javax.help.SearchView</type>
    <data engine="com.sun.java.help.search.DefaultSearchEngine">fr/SearchData</data>
  </view>

</helpset>
```

Log Console Event

```
// Presume we have an Exception which contains the error message
Exception ex = <the exception to be logged>;

VLogEvent logEvent = new VlogEvent(
    myTool, VlogEvent.ERROR, new Date(),
    "Connection Failure",
    "Connection to server XXX failed",
    ex.getMessage(),
    ex,
    null);

// Log the console event
VConsoleEvent ev = new VConsoleEvent(
    myTool, VConsoleActions.LOGEVENT, logEvent);
fireConsoleAction(ev);
```

Sample Code: Logging

```java
import com.sun.management.viper.services.Log;
import com.sun.management.viper.services.LogException;
private static final String MYLOGRESOURCES = "my.service.ServiceResources";

try {
   Log logsvc = (Log) infra.getServiceByName(Log.class.getName());
   logsvc.writeLog("BEANNAME",
      Log.CATOGERY_APPLICATION,
      Log.SEVERITY_ERROR,
      "FailureSummaryKey1",
      "FailureDetailKey1",
      MYLOGRESOURCES,
      null);
} catch (LogException le) {
   System.err.println("can't log message");
} catch (VException ve) {
   System.err.println("can't get log svc");
}
```

Sample Code: Manifest for External Client Provider

```
Name: com.mycompany.myproduct.XXXXClientProvider.class
Java-Bean: True

Name: com.mycompany.myproduct.XXXXClientProviderInfo.xml
Viper-Info: True
```

Sample Code: Manifest for Native Library

```
Name: com/mycompany/myproduct/MyService.class
Java-Bean: True

Name: com/mycompany/myproduct/MyService.xml
Viper-Info: True

Name: com/mycompany/myproduct/libprint.so
Viper-Lib: True
```

Sample Code: Manifest for Native Library

Sample Code: Manifest for Tools

```
Name: com/mycompany/myproduct/MyTool.class
Java-Bean: True

Name: com/mycompany/myproduct/MyTool.xml
Viper-Info: True
```

Menubar Integration

```java
public class MyMenuBar extends JMenuBar {

    JMenu actionMenu;
    JMenu viewMenu;
    JMenu helpMenu;

    public MyMenuBar() {

        JMenuItem mi;
        MyActionsListener actionListener = new MyActionsListener(...);

        actionMenu = new JMenu("Action");

        actionMenu.add(mi = new JMenuItem("Action Item 1"));
        mi.setActionCommand("action1");
        mi.addActionListener(actionListener);

        actionMenu.add(mi = new JMenuItem("Action Item 2"));
        mi.setActionCommand("action2");
        mi.addActionListener(actionListener);
        ...
        actionMenu.setActionCommand(VMenuID.ACTION);
        add(actionMenu);
        ...

        viewMenu = new JMenu("View");

        viewMenu.add(mi = new JMenuItem("View Item 1"));
        mi.setActionCommand("view1");
        mi.addActionListener(actionListener);

        viewMenu.add(mi = new JMenuItem("View Item 2"));
        mi.setActionCommand("view2");
        mi.addActionListener(actionListener);
        ...
        viewMenu.setActionCommand(VMenuID.VIEW);
        add(viewMenu);
        ...

        helpMenu = new JMenu("Help");
```

```
        helpMenu.add(mi = new JMenuItem("About My Tool"));
        mi.setActionCommand("about");
        mi.addActionListener(actionListener);
        ...
        helpMenu.setActionCommand(VMenuID.HELP);
        add(helpMenu);

        ...
    }
}
```

Sample Code: **Messaging**

```
public Chat extends VTool implements VConsoleActionListener,
    MessageListener {
    Message ms;
    MessagePushAgent ca;

    public void init(ToolInfrastructure inf) {
        try {
            ms = (Message)
            inf.getServiceByName(ServiceList.MESSAGE);
            ca = (MessagePushAgent)
            ms.getMessagePushAgent(); ca.init(inf);
            ca.createChannel("ChatChannel_EVERYBODY);
            ca.subscribe("ChatChannel_EVERYBODY", this);
        } catch (Exception e ) {
        }
    }

    public void handleMessage(VMessage message) {
        String str = message.getMessage();
    }
}
```

Sample Code: **native2ascii**

```
native2ascii Resources_<locale>.properties /tmp/mbe.properties
cp /tmp/mbe.properties Resources_<locale>.properties
```

Sample Code: Persistence

```
prefs = new PersistenceAgent(inf);
prefs.store(obj, version, key);
Object obj = (Object)prefs.restore(key);
```

Sample Code: Preferences

```
// Our "size" preference can be "small", "medium", or "large"
// Assume user selected "large".
String sizeKey = getClass().getName() + ".size";
properties.setProperty(sizeKey, "large");

Later on, possibly in the next SMC session, get the preference

String sizePreference = properties.getProperty(sizeKey);
if (sizePreference == "large")
     // do something for "large"
else if (sizePreference == "medium")
     ...
```

Sample Code: **PropertyChangeListener**

```java
public class MyTool implements Tool, PropertyChangeListener {

    ...

    /**
     * Property change listener, used to be notified when property
     * values change.
     *
     * @param e     the property change event
     */
    public void propertyChange(PropertyChangeEvent e) {

        String key = e.getPropertyName();
        if (key.equals(VConsoleProperties.DISPLAYMODEL))
            displayModel = (VDisplayModel)properties.getPropertyObject(
                VConsoleProperties.DISPLAYMODEL);

        else if (key.equals(VConsoleProperties.FRAME))
            consoleFrame = (JFrame)properties.getPropertyObject(
                VconsoleProperties.FRAME);

        else if (...

    }

    ...
}
```

Remove Child Node

```
VScopeNode parent = ...
VScopeNode child = ...

// Add the node as a child of the parent
parent.remove(child)

...

// Notify console that Navigation pane should be updated.
// (Assumes a general method for firing events).
VConsoleEvent ev = new VConsoleEvent(
     myTool, VConsoleActions.UPDATESCOPE, parent);
fireConsoleAction(ev);
```

Remove Child Node

Sample Code: Scope

```
AdminMgmtScope scope =
     (AdminMgmtScope)toolContext.getParameter(ToolContext.MGMTSCOPE);

...

// Connect to remote service, ...  maybe pass scope to service
MyService myService = (MyService)infrastructure.getServiceByName
                      ("com.mycompany.myproduct.MyServiceImpl");
myService.initialize(scope, ...);

...

// Get scope type
AdminMgmtScope mgmtScope =
     (AdminMgmtScope)toolContext.getParameter(ToolContext.MGMTSCOPE);
scopeType = mgmtScope.getMgmtScopeType();
if (scopeType.equals(AdminMgmtScope.ADM_SCOPE_DNS))
     System.out.println("managing DNS")

...


// Get the management server name.
String serverName = scope.getMgmtServerName();

...
```

Sample Code: Service Descriptor

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE component PUBLIC '-//Sun Microsystems, Inc.//Viper Component//EN'
'http://www.sun.com/solaris/management/dtds/viperbean_1_0.dtd'>
<component version="1.0">
<service>
        <interface>com.mycompany.myproduct.MyService</interface>
        <provider-class>com.mycompany.myproduct.MyServiceImpl</provider-class>
        <api-version>1.0</api-version>
        <is-singleton>true</is-singleton>
        <scope>file</scope>
</service>
<resource-bundle>com.mycompany.myproduct.MyServiceImplResources</resource-bundle>
```

Sample Code: Service Implementation

```java
import com.sun.management.viper.Service;
import java.rmi.RemoteException;

public class MyServiceImpl extends VService implements MyService {
    public MyServiceImpl() throws RemoteException, MyException {
      ...
    }

    public void method1(int i) throws MyException, RemoteException {
      ...
    }

    ...
}
```

Sample Code: Service Interface Definition

```
import com.sun.management.viper.Service;
import java.rmi.RemoteException;

public interface MyService extends Service {
    public void method1(int i) throws MyException, RemoteException;
      ...
}
```

Sample Code: Set Center Status Info Pane

```java
boolean showProgress = true;
...
final JProgressBar progressBar = new JProgressBar(0, 100);
progressBar.setValue(0);
progressBar.setStringPainted(true);
Object[] args = new Object[1];
args[0] = new Integer(progressBar.getValue());
progressBar.setString(MessageFormat.format("{0}%", args));
progressBar.setVisible(showProgress);

myTool.fireConsoleAction(new VConsoleEvent(
    myTool, VconsoleActions.UPDATEPROGRESS,
    showProgress ? progressBar : null));

...

// As the operation proceeds (presumably in a seperate thread),
// update the progress bar.
// We assume "count" and "total" are integer variables
// that represent the cumulative status thus far and the total
// expected.
if (count >= total)
    progressBar.setValue(100);
else
    progressBar.setValue((count * 100)/total);
Object[] args1 = new Object[1];
args1[0] = new Integer(progressBar.getValue());
progressBar.setString(MessageFormat.format("{0}%", args1));

...

// Later when the operation is complete, we want to disable
// the progress meter and remove it from the center pane.
myTool.fireConsoleAction(new VConsoleEvent(
    myTool, VConsoleActions.UPDATEPROGRESS, null));
```

Set Context Help

```
VScopeNode node = ...

// Here we retrieve the HTML file from the jar file.
// Note how the path to the file is relative to the
// package path of the specified Tool class.
String html = ResourceManager.getLocalizedTextFile(
     "html/myhelp.html", myTool);

node.setHTMLText(html);
```

Sample Code: Set Left Status Info Pane

```
String format = "{0} Networks";
Object[] args = new Object[1];
args[0] = new Integer(<the number of network objects in View pane>);
String status = MessageFormat.format(format, args);

VConsoleEvent e = new VConsoleEvent(
    myTool,
    VConsoleActions.UPDATESELINFO,
    status)
myTool.fireConsoleAction(e);
```

Sample Code: Setting Character Set Encoding in HTML

```
<html>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
<head>
<title>This title would be translated into the appropriate locale</title>
</head>
<body>
<p>This body would be translated into the appropriate locale.
</p>
</body>
</html>
```

Sample Code: Setting Character Set Encoding in HTML

Sample Code: Setting Selections in Results Pane

Assuming you already have a reference to the VConsoleProperties object, you first need to to get a reference to the display model:

```
VDisplayModel model =
    (VDisplayModel)(properties.getPropertyObject(VConsoleProperties.DISPLAY);
```

To select specific nodes, where `vSelected` is a Vector of `VScopeNode` objects:

```
model.setSelectedNodes(vSelected);
```

To select a range of nodes by index:

```
model.setSelectionInterval(index0, index1);
```

To select all nodes:

```
model.selectAll();
```

To unselect all nodes:

```
model.clearSelection();
```

Sample Code: **Specifying System Properties on Console Command Line**

Create a .java.policy file in your home directory. You can also grant a finer granularity of permissions on specific properties (for example, `mytool.demoMode`), rather than all properties.

```
grant {
    permission java.util.PropertyPermission "mytool.*", "read";
};
```

Add code in your tool to access the property:

```
String serviceType = "Wbem";
try {
    String serviceTypeProp = System.getProperty("mytool.serviceType");
    if (serviceTypeProp != null)
        serviceType = serviceTypeProp;
} catch (Exception ex) {
}

if (serviceType == "Wbem")
    // do something for Wbem
else if (serviceType == "Demo")
    // do something for Demo
...
```

Start SMC, and specify property on command line:

```
/usr/sadm/bin/smc -J-Dmytool.serviceType=Demo
```

Sample Code: Tool Descriptor

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE component PUBLIC '-//Sun Microsystems, Inc.//Viper Component//EN'
'http://www.sun.com/solaris/management/dtds/viperbean_1_0.dtd'>
<component version="1.0">
<tool>
        <interface>com.mycompany.myproduct.MyService</interface>
        <provider-class>com.mycompany.myproduct.MyTool</provider-class>
        <help-base>com.mycompany.myproduct.helpset.myhelpset</help-base>
        <api-version>1.0</api-version>
        <scope>file</scope>
        <tool-context>TC_APPLICATION_GUI</tool-context>
</tool>
<resource-bundle>com.mycompany.myproduct.MyToolResources</resource-bundle>
</component>
```

Sample Code: Tool Resource Bundle

```
BEANNAME=My Cool Management Tool
DESCRIPTION=A cool tool for creating and managing lotsa of information.
VERSION=2.0
VENDOR=My Company

################################################################
# Icon images, used only by SMC in order to render a "stub" of the
# application in the console before actually instantiating it.
# DO NOT LOCALIZE!
#
LARGEICON=../images/largeProgram.gif
SMALLICON=../images/smallProgram.gif
#
################################################################

################################################################
# DO NOT LOCALIZE!
#
locale_file_version=1.0
#
################################################################
```

Sample Code: **Tool.setProperties()**

```java
VDisplayModel displayModel = null;
VConsoleProperties properties = null;
JFrame consoleFrame = null;

/**
 * This method will be called by the console engine when this Tool is
 * created.  It sets the properties object which contains the properties
 * of the environment which the tool is running in.
 *
 * @param properties    the properties object
 */
public void setProperties(VConsoleProperties properties) {

    this.properties = properties;
    if (properties = null)
        return;

    // Get the display model
    displayModel = (VDisplayModel)properties.getPropertyObject(
        VConsoleProperties.DISPLAYMODEL);

    // Get the main console frame
    consoleFrame = (JFrame)properties.getPropertyObject(
        VconsoleProperties.FRAME);

    ...

} // setProperties
```

Sample Code: **Updating Display to Reflect Data Model Changes**

```
VConsoleEvent ev = new VConsoleEvent(
          src, VConsoleActions.UPDATESCOPE, node);
myTool.fireConsoleAction(ev);
```

`fireConsoleAction(ev)` might be a utility method in your main `Tool` class for firing console events.

For models that have an internal root, `node` is the parent node (the internal root) for all VScopeNode objects you are managing in the right-side results pane. For models without an internal root, `node` is the node in the navigation tree that is associated with the content in the right-side results pane.

Click here for a discussion on the differences between models with and without an internal root.

UPDATESCOPE

```
vObjects = <vector of application data objects>
exposedNode = <navigation tree node associated with this results pane>
rootNode = new VScopeNode();
exposeNode.setInternalRoot(rootNode);
for (int i=0; i<vObjects.size(); i++) {
        MyObject myObject = (MyObject)vObjects.elementAt(i);
        VScopeNode node = new VScopeNode(
                ... fill in fields as appropriate ...
                myObject);
        rootNode.add(node);
}
VConsoleEvent ev = new VConsoleEvent(
        src, VConsoleActions.UPDATESCOPE, exposedNode);
myTool.fireConsoleAction(ev);
```

Sample Code: **VConsoleActionListener**

```java
public class MyTool implements Tool, VConsoleActionListener {

    boolean hasFocus = false;

    public void start() {
        hasFocus = true;
        ...
    }

    public void stop() {
        hasFocus = false;
        ...
    }

    public void consoleAction(VConsoleEvent ev) {
        if (!hasFocus)
                return;

        if (ev.eventID.equals(VConsoleActions.XXX)) {
                ...
        } else if (ev.eventID.equals(VConsoleActions.YYY)) {
        ...
    }

    ...
}
```
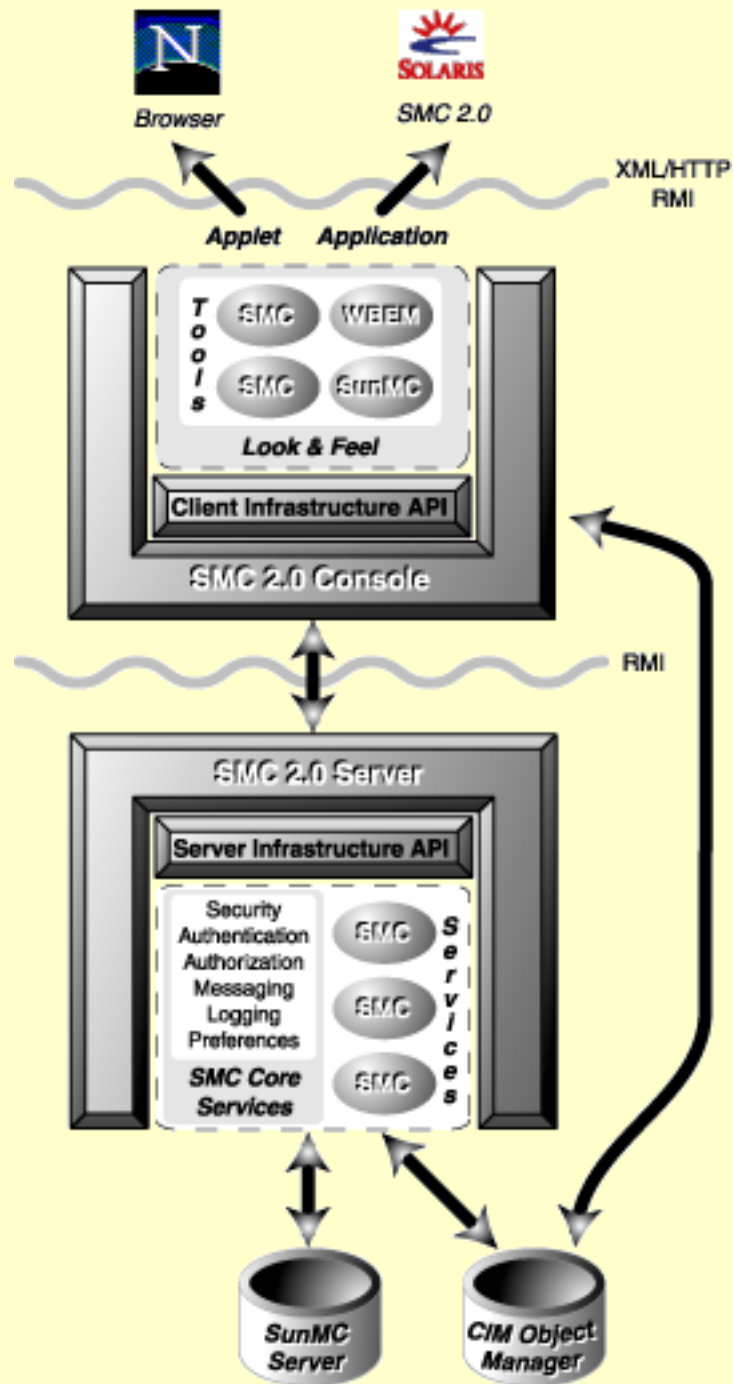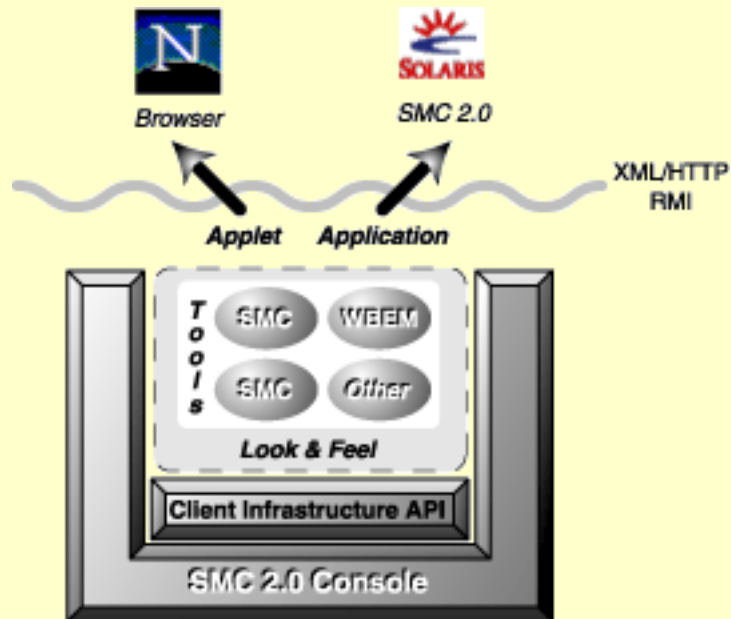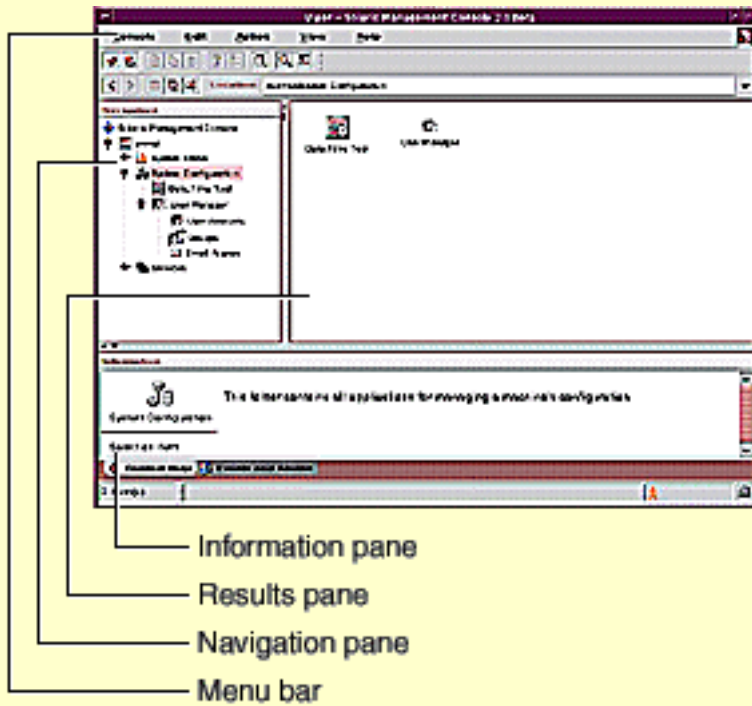
🏭 SMC Architecture

Browser

SMC 2.0

XML/HTTP
RMI

Applet   Application

Tools

SMC   WBEM

SMC   SunMC

Look & Feel

Client Infrastructure API

SMC 2.0 Console

RMI

SMC 2.0 Server

Server Infrastructure API

Security
Authentication
Authorization
Messaging
Logging
Preferences

**SMC Core
Services**

SMC

SMC

SMC

Services

SunMC
Server

CIM Object
Manager

📷 SMC Console

🔧 Default Console Window Layout

Information pane

Results pane

Navigation pane

Menu bar

🧰 Event Bus

Event Bus

🔧 JavaHelp Localization Hierarchy

```
...\help
    my_helpset.hs
    my_helpset_de.hs
    my_helpset_fr.hs
```

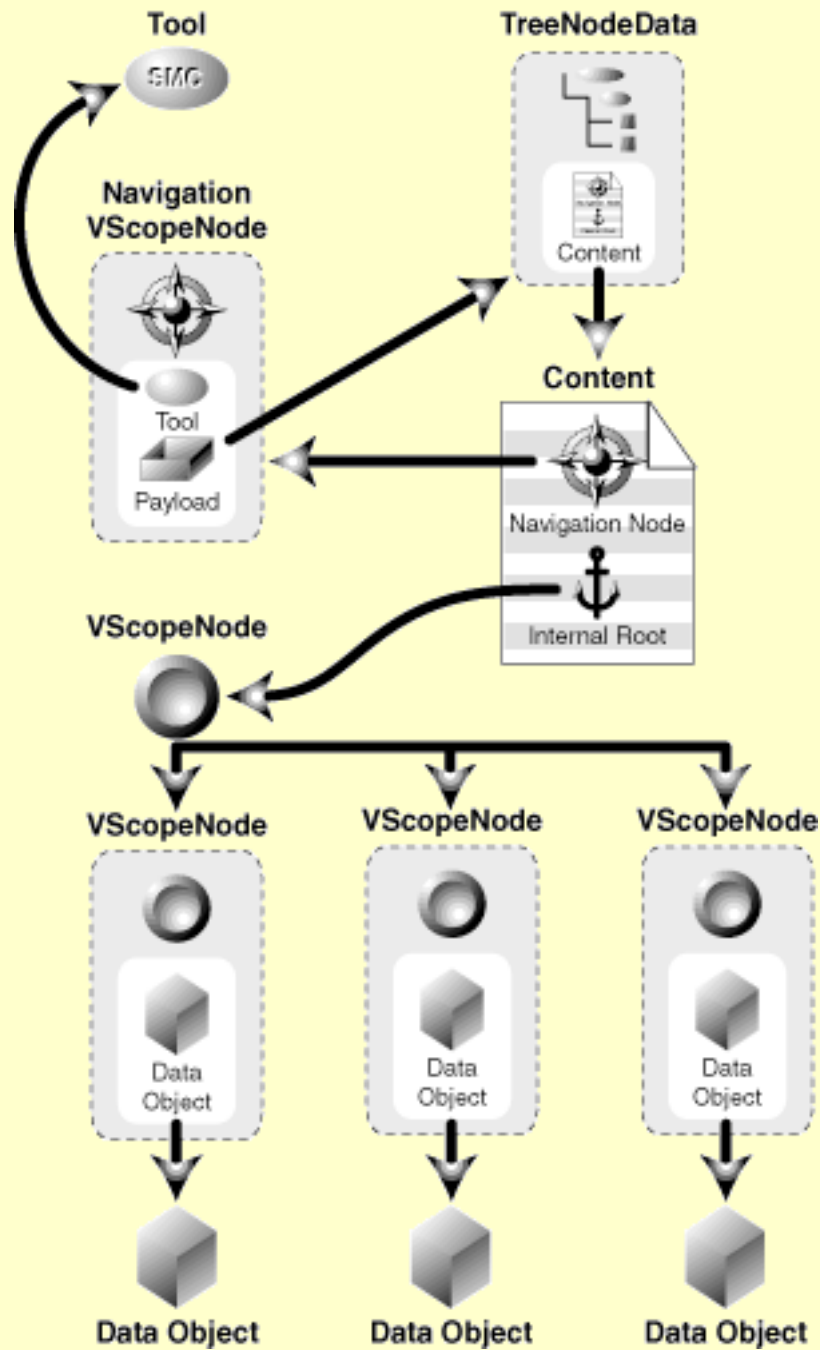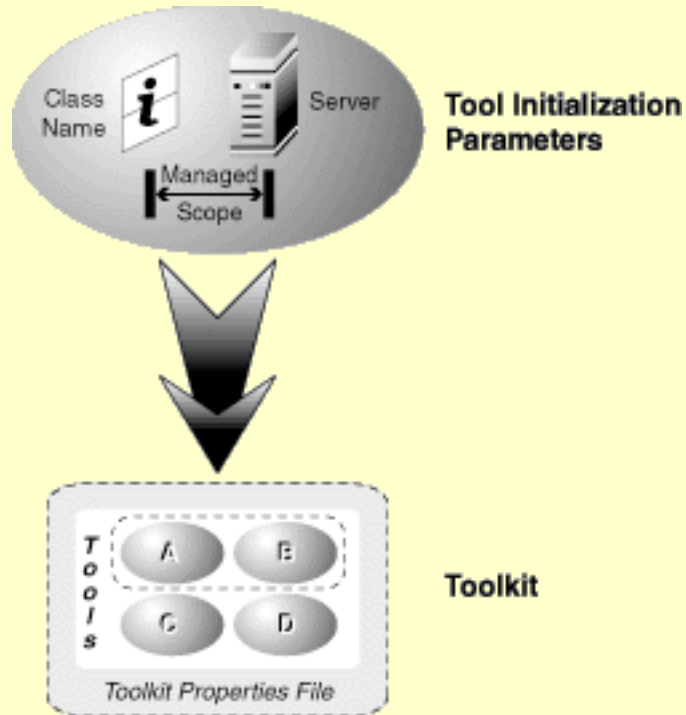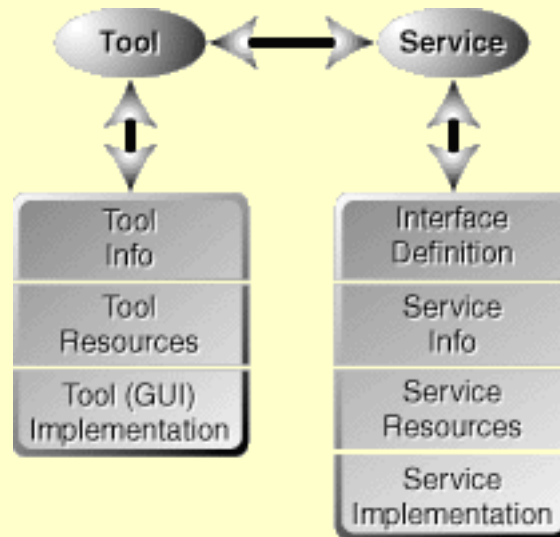| \default	\de	\fr
my_map.jhm	my_map.jhm	my_map.jhm
my_toc.xml	my_toc.xml	my_toc.xml
my_index.xml	my_index.xml	my_index.xml
\JavaHelpSearch	\JavaHelpSearch	\JavaHelpSearch
search db files ...	search db files ...	search db files ...
\topics	\topics	\topics
\topicA	\topicA	\topicA
topic.html	topic.html	topic.html
⋮	⋮	⋮
\topicB	\topicB	\topicB
topic.html	topic.html	topic.html
⋮	⋮	⋮
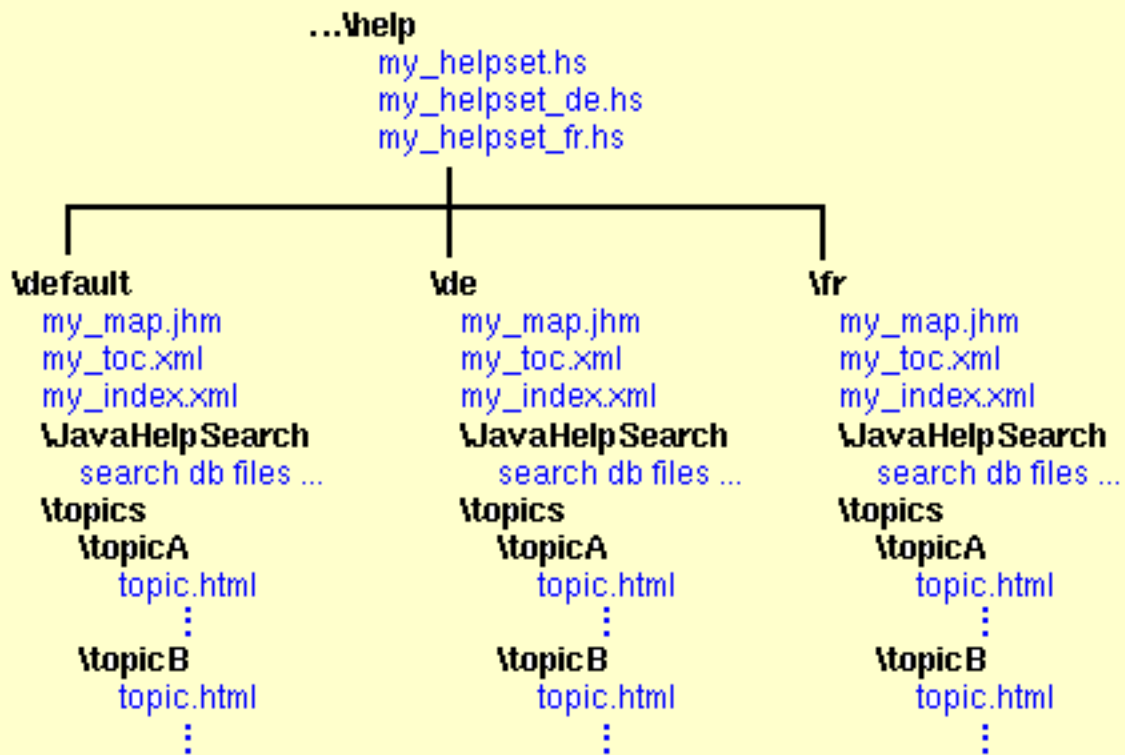
📷 RMI Client/Server Model
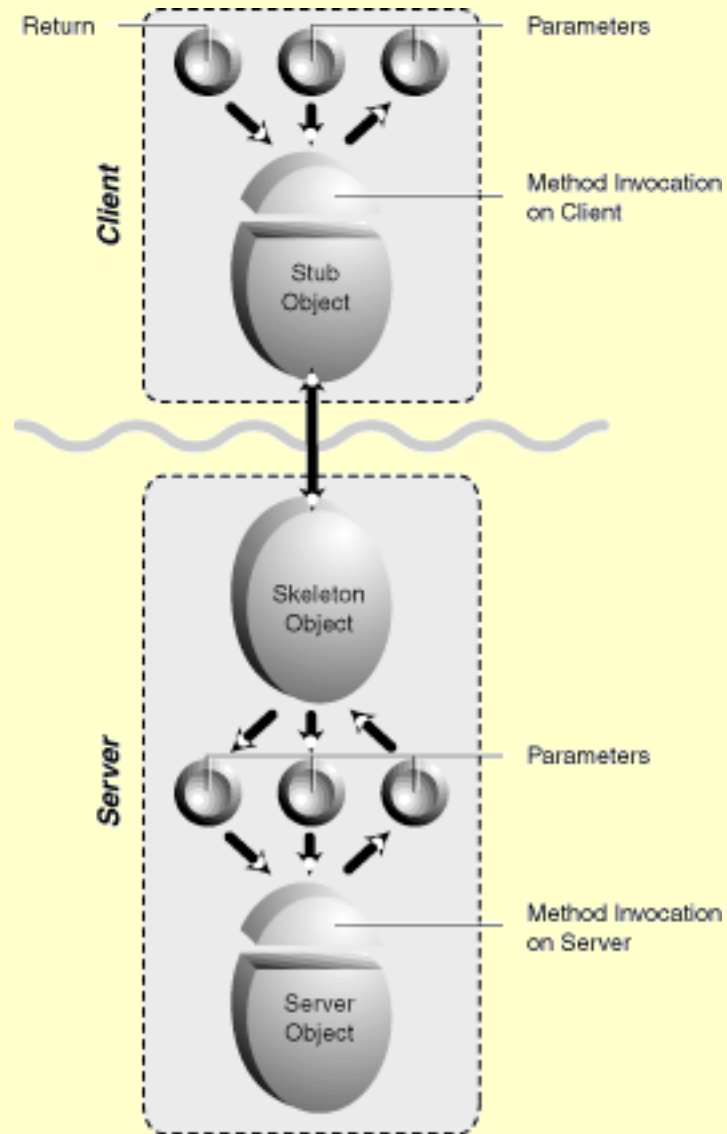
Sample Application Data Model

Tool Initialization Parameters

Tool/Service Interaction

Typical SMC Session

Tool Parameters

Toolkit View
Downloaded to Console