Programming with System Agents

Release 3.5 of Forte™ 4GL

Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 U.S.A. 1-800-786-7638

Part No. 806-6673-01
October 2000, Revision A

Contents

3 Developing Custom Agents

A Example Applications

Preface

Programming with System Agents provides information about customizing and enhancing the system management facilities provided by Forte.

This manual explains how to develop your own custom agents and how to use the commands and instruments for agents in your TOOL code.

This manual also contains information about the SystemMonitor library classes.

You should have a copy of the *Escript and System Agent Reference Manual* readily available so that you can reference information about Forte-defined agents and their commands and instruments.

This manual is intended for application developers. We assume that you:

- are familiar with the system management facilities described in the *Forte 4GL System Management Guide*
- have access to the *Escript and System Agent Reference Manual*
- have TOOL programming experience
- are familiar with your particular window system
- understand the basic concepts of object-oriented programming as described in *A Guide to the Forte 4GL Workshops*
- have used the Forte Workshops to create classes

Organization of This Manual

This manual is organized to explain how to use the SystemMonitor library classes, then to provide reference information about the SystemMonitor library classes.

This manual assumes that you have a copy of *Escript and System Agent Reference Manual* available. You will need to reference this book for information about Forte-defined agents.

This manual contains the following chapters:

Chapter	Description
Chapter 1, "Introduction"	Provides an overview of the SystemMonitor library and its uses.
Chapter 2, "Accessing System Agents Using TOOL Code"	Provides detailed information about how you can use the SystemMonitor library classes in your TOOL code.
Chapter 3, "Developing Custom Agents"	Provides detailed information about how you can develop custom agents for monitoring your system.
Appendix A, "Example Applications"	Describes the examples used in this manual.

Conventions

This manual uses standard Forte documentation conventions in specifying command syntax and in documenting TOOL code.

Command Syntax Conventions

The specifications of command syntax in this manual use a “brackets and braces” format. The following table describes this format:

Format	Description
bold	Bold text is a reserved word; type the word exactly as shown.
italics	Italicized text is a generic term that represents a set of options or values. Substitute an appropriate clause or value where you see italic text.
UPPERCASE	Uppercase text represents a constant. Type uppercase text exactly as shown.
underline	Underlined text represents a default value.
vertical bars |	Vertical bars indicate a mutually exclusive choice between items. See braces and brackets, below.
braces { }	Braces indicate a required clause. When a list of items separated by vertical bars is enclosed in braces, you must enter one of the items from the list. Do not enter the braces or vertical bars.
brackets [ ]	Brackets indicate an optional clause. When a list of items separated by vertical bars is enclosed by brackets, you can either select one item from the list or ignore the entire clause. Do not enter the brackets or vertical bars.
ellipsis …	The item preceding an ellipsis may be repeated one or more times. When a clause in braces is followed by an ellipsis, you can use the clause one or more times. When a clause in brackets is followed by an ellipsis, you can use the clause zero or more times.

TOOL Code Conventions

Where this manual includes documentation or examples of TOOL code, the TOOL code conventions in the following table are used.

Format	Description
parentheses ( )	Parentheses are used in TOOL code to enclose a parameter list. Always include the parentheses with the parameter list.
comma ,	Commas are used in TOOL code to separate items in a parameter list. Always include the commas in the parameter list.
colon :	Colons are used in TOOL code to separate a name from a type, or to indicate a Forte name in a SQL statement. Always include the colon in the type declaration or statement.
semicolon ;	Semicolons are used in TOOL code to end a TOOL statement. Always type a semicolon at the end of a statement.

The Forte Documentation Set

Forte produces a comprehensive documentation set describing the libraries, languages, workshops, and utilities of the Forte Application Environment. The complete Forte Release 3 documentation set consists of the following manuals in addition to comprehensive online Help.

Forte 4GL

- *A Guide to the Forte 4GL Workshops*
- *Accessing Databases*
- *Building International Applications*
- *Escript and System Agent Reference Manual*
- *Forte 4GL Java Interoperability Guide*
- *Forte 4GL Programming Guide*
- *Forte 4GL System Installation Guide*
- *Forte 4GL System Management Guide*
- *Fscript Reference Manual*
- *Getting Started With Forte 4GL*
- *Integrating with External Systems*
- *Programming with System Agents*
- *TOOL Reference Manual*
- *Using Forte 4GL for OS/390*

Forte Express

- *A Guide to Forte Express*
- *Customizing Forte Express Applications*
- *Forte Express Installation Guide*

Forte WebEnterprise and WebEnterprise Designer

- *A Guide to WebEnterprise*
- *Customizing WebEnterprise Designer Applications*
- *Getting Started with WebEnterprise Designer*
- *WebEnterprise Installation Guide*

Forte Example Programs

In this manual, we often include code fragments to illustrate the use of a feature that is being discussed. If a code fragment has been extracted from a Forte example program, the name of the example program is given after the code fragment. If a major topic is illustrated by a Forte example program, reference will be made to the example program in the text.

These Forte example programs come with the Forte product. They are located in subdirectories under $FORTE_ROOT/install/examples. The files containing the examples have a .pex suffix. You can search for TOOL commands or anything of special interest with operating system commands. The .pex files are text files, so it is safe to edit them, though you should only change private copies of the files.

Viewing and Searching PDF Files

You can view and search 4GL PDF files directly from the documentation CD-ROM, store them locally on your computer, or store them on a server for multiuser network access.

Note You need Acrobat Reader 4.0+ to view and print the files. Acrobat Reader with Search is recommended and is available as a free download from http://www.adobe.com. If you do not use Acrobat Reader with Search, you can only view and print files; you cannot search across the collection of files.

To copy the documentation to a client or server:

1. Copy the fortedoc directory and its contents from the CD-ROM to the client or server hard disk.

   You can specify any convenient location for the fortedoc directory; the location is not dependent on the Forte distribution.

2. Set up a directory structure that keeps the fortedoc.pdf and the 4gl directory in the same relative location.

   The directory structure must be preserved to use the Acrobat search feature.

Note To uninstall the documentation, delete the fortedoc directory.

To view and search the documentation:

1. Open the file fortedoc.pdf, located in the fortedoc directory.
2. Click the **Search** button at the bottom of the page or select **Edit > Search > Query**.
3. Enter the word or text string you are looking for in the Find Results Containing Text field of the Adobe Acrobat Search dialog box, and click **Search**.

   A Search Results window displays the documents that contain the desired text. If more than one document from the collection contains the desired text, they are ranked for relevancy.

Note For details on how to expand or limit a search query using wild-card characters and operators, see the Adobe Acrobat Help.

4. Click the document title with the highest relevance (usually the first one in the list or with a solid-filled icon) to display the document.

   All occurrences of the word or phrase on a page are highlighted.

5. Click the buttons on the Acrobat Reader toolbar or use shortcut keys to navigate through the search results, as shown in the following table:

Toolbar Button	Keyboard Command
Next Highlight	Ctrl+]
Previous Highlight	Ctrl+[
Next Document	Ctrl+Shift+]

6. To return to the fortedoc.pdf file, click the Homepage bookmark at the top of the bookmarks list.
7. To revisit the query results, click the **Results** button at the bottom of the fortedoc.pdf home page or select **Edit > Search > Results**.

Chapter 1

Introduction

This chapter provides an overview of Forte agents and how to use the Forte SystemMonitor library classes to write programs that interact with these agents.

This chapter provides an overview of:

- the purpose of agents in the Forte system
- how you can write custom agents
- how you can use agent commands and instruments to monitor your system
- the SystemMonitor library classes

About Agents and the SystemMonitor Library

Agents and managed objects

Forte's system management facilities, the Environment Console and Escript, are built upon a set of management *agents* that monitor and control a corresponding set of *managed objects*. For each object that you can manage using the Environment Console or Escript, there is a system management agent that actually monitors and controls that object. For example, when Forte creates an installed partition object, it also creates a corresponding Installed Partition agent. The managed object performs its normal functions without having any knowledge of its agent, and the agent is automatically created and removed along with its managed object.

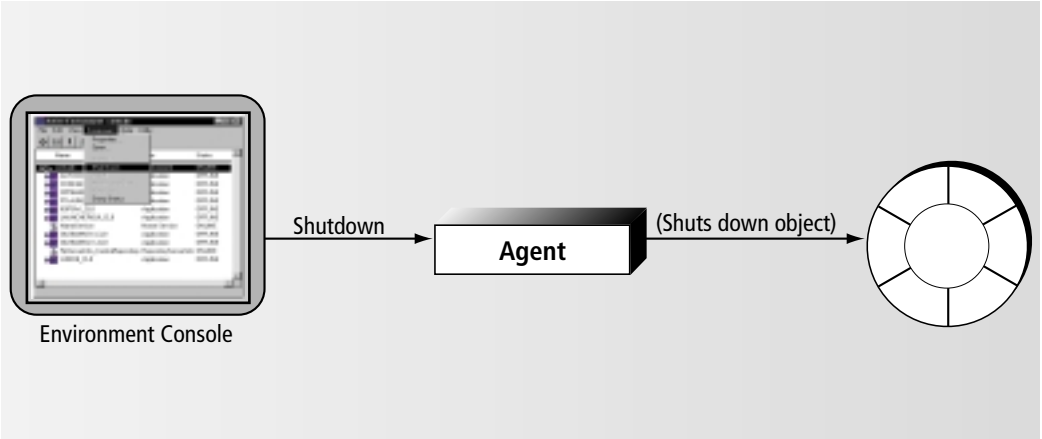

Figure 1 *Using an Agent to Manage an Object*

To enable application developers to access these system agents, as well as define their own agents, Forte provides the SystemMonitor library. The SystemMonitor library defines classes that provide a standard interface that you can use to access all system and user-defined agents. You can also use these classes to define agents of your own that you can add to the agent hierarchy and access using the Environment Console and Escript.

This guide explains how to program your own agents and your own system management applications.

You should be familiar with using the Environment Console and Escript. These system management facilities use the methods, attributes, and events defined for classes in the SystemMonitor library to programmatically access and interact with the agents for the managed object. These agents then monitor and control their managed objects as necessary.

Overview of Agents

This section provides a brief reminder of what agent commands and instruments are and how the agent hierarchy works. For a thorough explanation about Forte agents and Forte's system management facilities, see *Forte 4GL System Management Guide* and *Escript and System Agent Reference Manual*.

Agents define the interface between system management facilities and a managed object. The agents provide a standard interface of methods, attributes, and events that let you access their commands and instruments. Forte provides a set of system management agents, which you can use to manage Forte system objects. You can also define your own agents to manage objects in your applications.

System management facilities and agents

The Environment Console and Escript programmatically access and interact with the agents for the managed object, not the objects themselves. The agent monitors and controls the object as necessary and provides the interface between managing applications and the managed object.

Agent Commands

Each agent has a set of commands or operations it can perform on its managed object. For example, an Installed Partition agent can start an installed partition when you invoke the agent's **Startup** command. Likewise, an Active Partition agent can shut down an active partition when you invoke that agent's **Shutdown** command.

Agent Instruments

Each agent also has a set of instruments, each representing a type of data that can be obtained from or set on the managed object. An instrument can represent a property or attribute of the managed object or it can represent more dynamic information. For example, an Active Partition agent has a ProcessID instrument that represents the partition's process ID. Also, a Distributed Object Manager agent has a MethodsSent instrument that represents the number of messages sent by the distributed object manager to remote partitions during a specified period of time.

The instruments defined for each agent generally represent data that is useful for monitoring or controlling the managed object.

Agent Hierarchy

Forte's system management architecture places each Forte system management agent and user-defined, or *custom,* agent within a hierarchical structure of parent agents and subagents, as shown in Figure 2. This hierarchy is a containment hierarchy, in which the parent agent—an Application agent, for example—contains its subagents—Partition agents, for example.

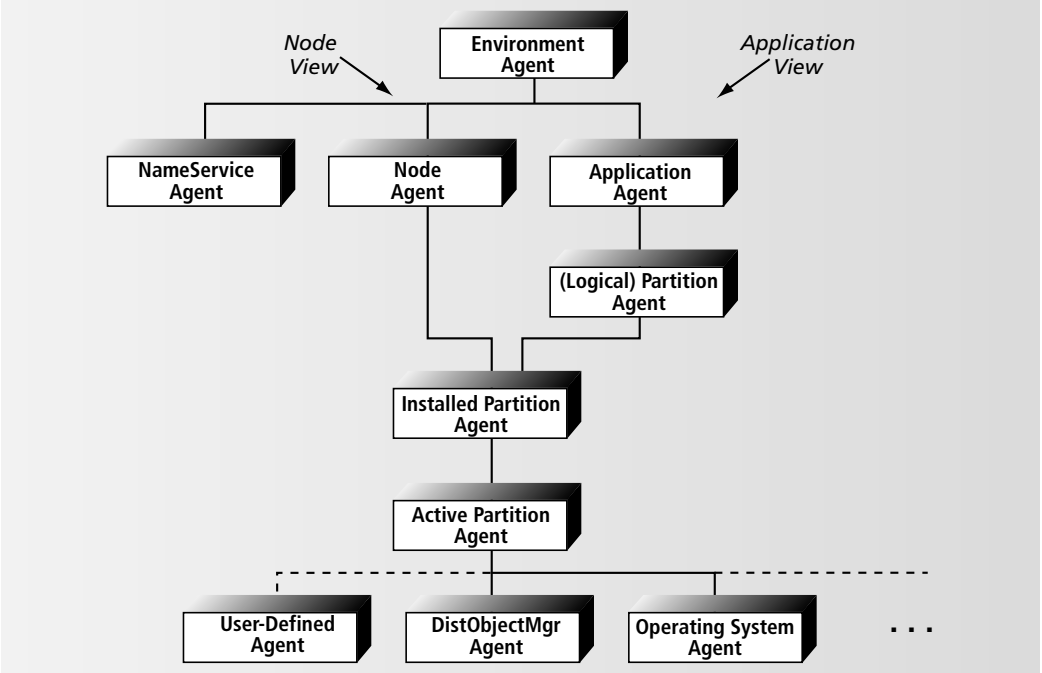

Figure 2 *System Management Agent Hierarchy*

To navigate around this hierarchy, you can start by getting references to the top of this hierarchy, to the Environment agent, or near the bottom, to an Active Partition agent. You can then navigate to any other agents in the hierarchy and get references to those agents. Once you have these references, you can invoke commands and access instruments on the agents, or even attach new custom agents into the hierarchy.

For more information about how this hierarchy works and about the Forte system management agents, see *Forte 4GL System Management Guide*.

Accessing Agent Commands and Instruments

Both the agents defined by Forte and custom agents defined by application developers provide commands and instruments that can be accessed by TOOL code. These commands and instruments enable you to write special applications that manage your system.

Forte defines a standard method interface for all system agents. This interface allows you to write TOOL code that works with all agents, even custom agents. Using this interface, developers can access the commands, events, and instruments dynamically, even ones that have been added or removed since the last time the agent was accessed.

Working with agent commands

Agent commands do not correspond to any classes. Commands belonging to a particular agent exist only in the runtime system, and are defined in the methods for that agent. Forte provides methods on the SystemAgent class that return a list of commands that are available for the agent and that instruct the agent to execute a specific command.

Accessing and setting instruments

Forte also provides methods on the SystemAgent class that return a list of available instruments, let you add additional instruments, delete instruments, and find specific instruments. Forte also provides classes that map to the types of instruments supported by Forte. These classes provide methods and attributes that let you get and set values for agent instruments.

For detailed instructions for accessing agents from TOOL code, see Chapter 2, "Accessing System Agents Using TOOL Code."

Developing Custom Agents

You can define and add any number of custom agents to the agent hierarchy. For example, you could develop a custom agent that has instruments that track specific data about a shared service object in your application. This agent could also define commands that let you, for example, cancel tasks, display current waiting events, or list the current clients that are using the service object.

The SystemMonitor library provides a SystemAgent class that represents a generic system agent. When you define a custom agent, you subclass this SystemAgent class and override some of the SystemAgent class methods to define and implement commands and instruments for this new agent. To implement instruments on the new agent, you use objects of the Instrument class and its subclasses, which are also in the SystemMonitor library.

When you have implemented an agent using the SystemAgent class, you can access the agents using Forte's standard system management facilities, the Environment Console and Escript, as well as any other applications that interact with agents using their standard interfaces.

For detailed instructions for developing custom agents, see Chapter 3, "Developing Custom Agents."

About the SystemMonitor Library

The SystemMonitor library provides classes that let you develop custom agents for managing the processing and performance of objects in your system. You can also use certain SystemMonitor library classes to programmatically access the commands, events, and instruments provided by Forte-defined or custom system agents.

SystemAgent Class

The SystemAgent class is the superclass of all Forte-defined or custom agent classes. This class, described in detail in the Forte online Help, provides methods and attributes for defining custom agents and accessing the commands, events, and instruments provided by an existing agent.

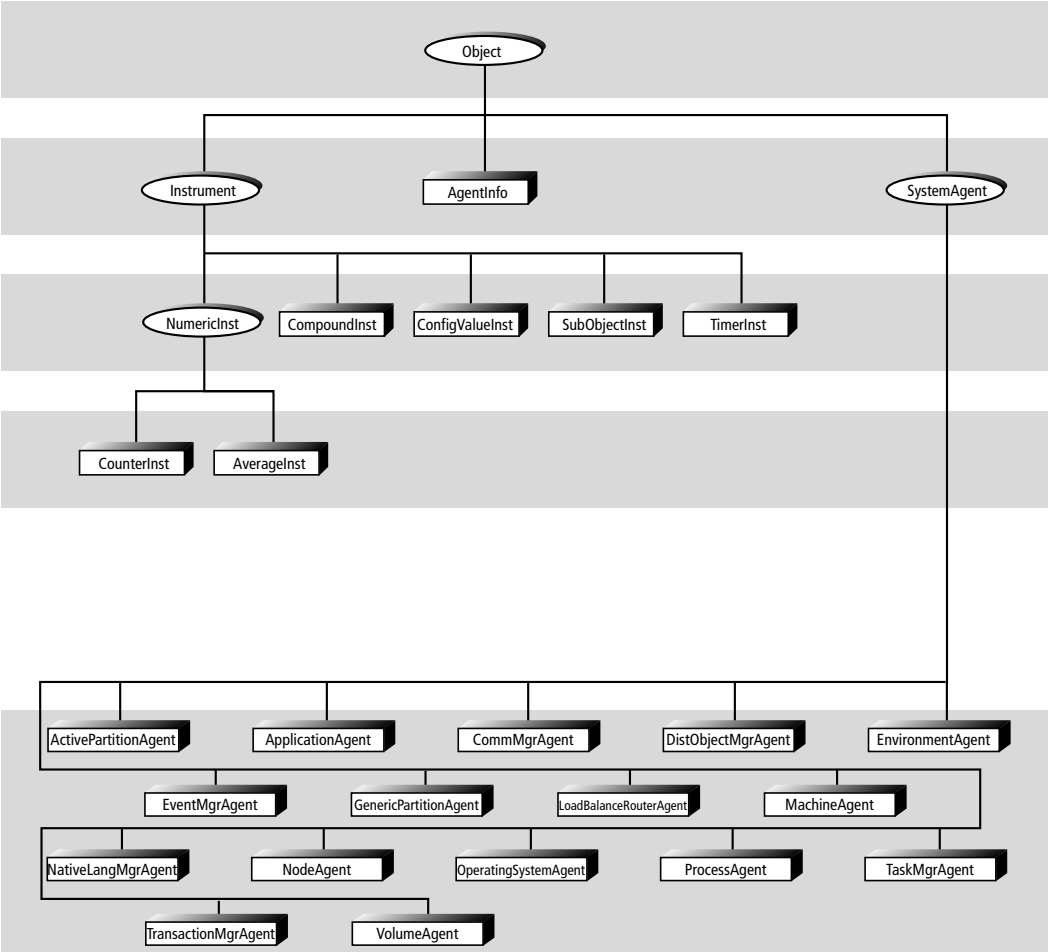

Figure 3 *SystemMonitor Library Classes*

Subclasses of the SystemAgent class

The SystemMonitor library also contains classes for many of the Forte-defined agents, such as ActivePartitionAgent and NodeAgent. However, all of the Forte-defined agents, like the custom agents, are subclasses of the SystemAgent class. By definition, all agents have the same methods and attributes as the SystemAgent class. For information about the commands and instruments implemented for the Forte-defined agents, see *Escript and System Agent Reference Manual*.

Instrument Subclasses

Forte provides several subclasses of the abstract Instrument class that map to the instrument types supported by Forte. You can use these instrument subclasses to define new instruments for a custom agent or to access an instrument defined for an existing agent. For information on any of these instrument subclasses, see the Forte online Help.

Instrument Type	Instrument Subclass	Description
Average	AverageInst	Read only. Contains an average value.
Compound	CompoundInst	Contains a static set of instruments. These instruments can be of different types.
Configuration	ConfigValueInst	Read/write or read only. Contains a simple value, such as an integer value or a TextData object.
Counter	CounterInst	Read only. Contains a value based on counting something.
SubObject	SubObjectInst	Contains a dynamic set of instruments. All instruments are of the same type.
Timer	TimerInst	Read/write. Timer that prompts the agent to do something after a certain interval or set of intervals.

You cannot subclass any of these Forte subclasses of the Instrument class or the Instrument class itself.

Chapter 2

Accessing System Agents Using TOOL Code

This chapter explains how you can use SystemMonitor library classes in your TOOL code to monitor your system. These classes let you use the commands and instruments provided by the agents defined by Forte, as well as commands and instruments provided by agents defined by application developers.

This chapter covers the following topics:

- getting a reference to an agent
- navigating to a particular agent in the agent hierarchy
- getting information about an agent
- invoking agent commands
- accessing and updating agent instrument data

About Accessing System Agents

This section provides an overview of how you can access Forte-defined and user-defined agents from within your TOOL code.

This chapter uses examples from the AgentAccess and AgentBanking examples provided with Forte. For specific information about locating and running these applications, see Appendix A, "Example Applications."

Agent objects provide standard agent interface

All agents that are running in a Forte system have the same attributes and methods that are defined by the SystemAgent class. All agents are instances of a subclass of the SystemAgent class. The SystemAgent class defines a standard method and attribute interface that is used by all agents and by other programs that are accessing the agents.

The methods and attributes of the SystemAgent class are described in the Forte online Help.

This standard interface to all agents lets you write TOOL code that works with all agents, even custom agents. If you want to write an application that generically accesses many different agents, your application does not have to know anything special about any particular agent. For example, Escript interacts with all agents, even user-defined agents, in exactly the same way, without needing to know anything about a particular agent.

Within your TOOL code, you can get a reference to the agent hierarchy, navigate to a particular agent in the agent hierarchy, get information about an agent, and work with an agent's commands and instruments.

Getting a Reference to the Agent Hierarchy

The Forte system lets you get references to the following two points in the agent hierarchy:

- Environment agent

  The Environment agent, which lets you interact with the Environment Manager, is the top of the agent hierarchy.

- Active Partition agent

  The Active Partition agent, which lets you interact with a running partition.

Once you have a reference to either of these agents, you can navigate to any other agent that you need to, as described in "Navigating Around the Agent Hierarchy" on page 22.

Referencing the Environment Agent

task.Part.GetEnvironmentMgr method

The Framework Partition class provides a method called GetEnvironmentMgr, which returns a distributed reference to the currently active Environment agent. The syntax for this method is:

GetEnvironmentMgr()

Returns Object

You can always access the partition on which an application is currently running by using the Part attribute of the TaskHandle class. You can reference the current task's task handle using the **task** keyword. The following example shows how you could use the GetEnvironmentMgr method to get a reference to the Environment agent:

```
-- Connect to currently active environment
self.currActEnv = EnvironmentAgent(task.part.GetEnvironmentMgr());
```

See AgentAccess example: **Project:** AgentAccessSvc • **Class:** AgentLogMgr • **Method:** Init

Because the GetEnvironmentMgr method returns a generic object, you need to cast the returned object as EnvironmentAgent to be able to use the methods and attributes defined for the Environment agent, including being able to navigate to other agents.

For more information about the task keyword, see *TOOL Reference Manual* and the Forte online Help.

Referencing an Active Partition Agent

task.Part.ActPartAgent attribute

The Framework Partition class has an attribute called ActPartAgent, which contains a distributed reference to the current partition's agent.

You can always access the partition on which an application is currently running by using the Part attribute of the TaskHandle class. You can reference the current task's task handle using the **task** keyword. The following example shows how you could use the ActPartAgent attribute to get a reference to an Active Partition agent:

```
partAgent : ActivePartitionAgent;
partAgent = ActivePartitionAgent(task.Part.ActPartAgent);
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

Because the ActPartAgent attribute contains a generic object, you need to cast the returned object as ActivePartitionAgent (or SystemAgent) to be able to use the methods and attributes defined for the agent, including being able to navigate to other agents.

For more information about the task keyword, see *TOOL Reference Manual* and the Forte online Help.

Navigating Around the Agent Hierarchy

After you get a reference to either the Environment agent or an Active Partition agent, you need to navigate to whatever agent you want to access. The following figure shows how Forte-defined agents are related to one another in the agent hierarchy.

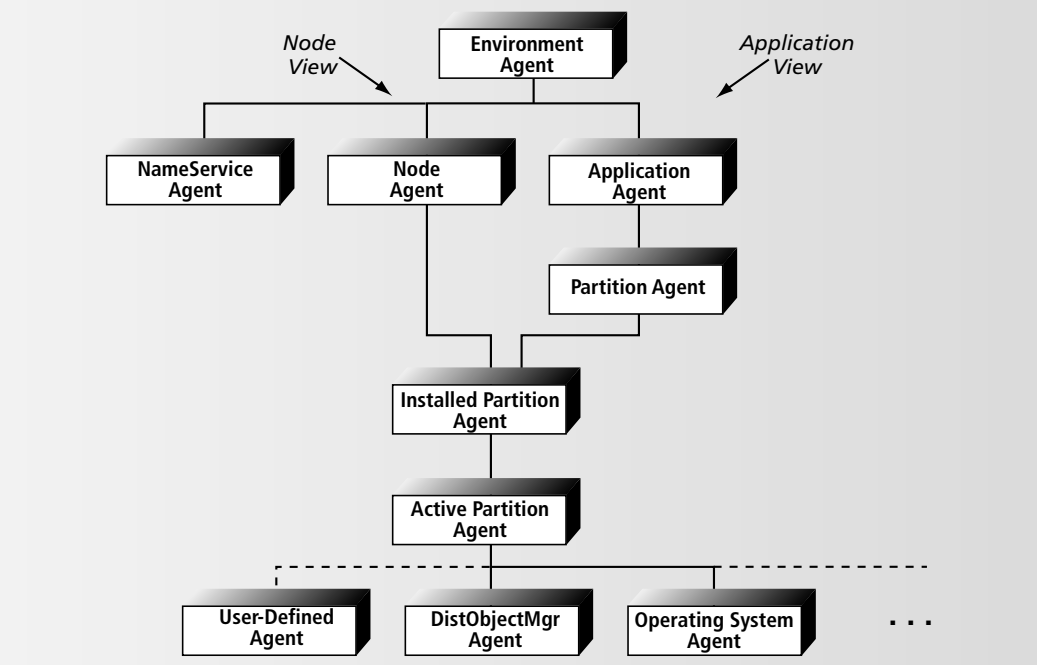

Figure 4 *System Management Agent Hierarchy*

To move "up" the hierarchy relative to the agent you are referencing, you need to locate its parent agent. To move "down" the hierarchy relative to the agent you are referencing, you need to locate one if its subagents.

Cast custom agents as SystemAgent

When you navigate to a custom agent, be aware that you should only cast user-defined agents as SystemAgent (or leave them as SystemAgent). This guarantees that your calling application can call any custom agent through the generic methods and attributes defined for the distributed SystemAgent class.

These Forte-defined agents and their commands and instruments are listed in *Escript and System Agent Reference Manual*.

Navigating to Parent Agents

SystemAgent.ParentAgent attribute

Each agent inherits an attribute from SystemAgent called ParentAgent (SystemAgent) that references the parent agent for the agent. Therefore, you can navigate to this parent agent using code like the following:

```
-- parentAgent references the agent that is the parent
-- of the partAgent agent.

parentAgent : SystemAgent;
parentAgent = partAgent.ParentAgent;
```

The Environment agent is at the top of the agent hierarchy, so if you try to navigate to the parent agent of the Environment agent, you will get a NIL value.

The ParentAgent attribute is also described in the Forte online Help.

Navigating to Subagents

SystemAgent.FindSubAgent method

The SystemAgent class also has a FindSubAgent method, which returns a SystemAgent object based on the name of a subagent. For syntax information about this method, see the Forte online Help.

The following example shows how you can use the FindSubAgent method to assign a reference to a subagent based on the subagent's name:

```
-- Set a reference to the BankServiceAgent.
self.BankAgent = BankServer.CurrentActivePartition.
  FindSubAgent(name= 'BankServiceAgent');
```

See AgentBanking example: **Project:** AgentBanking • **Class:** AdminWindow • **Method:** Init

SystemAgent.SubAgents attribute

Each agent inherits an attribute from SystemAgent called SubAgents (GenericArray of SystemAgent), which contains the subagents of the current agent. You can use this array to get a reference to an agent that is a subagent of the current agent, as shown in the following example:

```
-- Set mySubagent to the third subagent in the SubAgents array.
mySubagent : SystemAgent;
mySubagent : currentAgent.SubAgents[3];
```

One reason you might want to navigate to a subagent is if you are writing TOOL code that starts the server partitions of an application. In the following example, the TOOL code navigates to the Application agent for an application at compatibility level 0 whose name is AgentBanking to start the server partitions for the application.

```
-- Connect to currently active environment
currActEnv : SystemAgent =
  EnvironmentAgent(task.part.GetEnvironmentMgr());

-- Navigate to the Application agent
myApp : SystemAgent =
  currActEnv.FindSubAgent(name = 'AgentBanking_cl0');

-- Start all server partitions
myApp.ExecuteCommand(command='Startup');
```

For more information about the SubAgents attribute, see the Forte online Help.

Getting Information about an Agent

When you use agents in your TOOL application, you might need some information about an agent or its subagent to present to the user or to make decisions in the logic of your application.

SystemAgent.GetInfo method

Each agent inherits a GetInfo method from SystemAgent (described in the Forte online Help), which returns an AgentInfo object. AgentInfo, a SystemMonitor library class, contains information about this agent, such as the agent's name, the type of the managed object, and the agent's state. For a full description of the AgentInfo class, see the Forte online Help. The following example shows how you could use information from the AgentInfo class:

```
-- Print the name of the current agent to the log.
agentStuff : AgentInfo;
agentStuff = currentAgent.GetInfo();
agentName : string = agentStuff.AgentName;
task.Part.LogMgr.Put('Agent's name is ');
task.Part.LogMgr.PutLine(agentName);
```

SystemAgent.SubAgentInfo attribute

The SystemAgent class has a SubAgentInfo attribute (GenericArray of AgentInfo), which you can use to get information about subagents of the current agent. The following example shows how you can print information about the subagents of the current agent to a log file:

```
textString.SetValue(source='SubAgents of Environment agent:');
self.logFile.WriteLine(source=textString);
for subag in currActEnv.SubAgentInfo do
  textString.Concat(source='   ');
  textString.Concat(source=subag.AgentName);
  textString.Concat(source=' agent manages ');
  textString.Concat(source=subag.MOTypeName);
  textString.Concat(source=' object. It is ');
  textString.Concat(source=subag.StateName);
  textString.Concat(source='.');
  self.logFile.WriteLine(source=textString);
end for;
```

See AgentAccess example: **Project:** AgentAccessSvc • **Class:** AgentLogMgr • **Method:** LogAgentInfo

This example from the AgentAccess example program produces output like the following:

```
SubAgents of Environment agent:
   PC-Windows agent manages Model Node object. It is OFFLINE.
   Mac-client agent manages Model Node object. It is OFFLINE.
   NameService agent manages Name Service object. It is ONLINE.
   hillary agent manages Node object. It is ONLINE.
   AgentBanking_cl0 agent manages Application object. It is ONLINE.
   AgentAccess_cl0 agent manages Application object. It is IN-PROGRESS.
```

SystemAgent. FindSubAgentInfo method

Each agent inherits a FindSubAgentInfo method from SystemAgent, which returns an AgentInfo object that contains information about a specified subagent. The syntax for this method is described in the Forte online Help.

You can then access the attributes of the AgentInfo object (described in the Forte online Help) to get information about the agent, such as its name, the type of its managed object, and its state.

Invoking Agent Commands

Each agent has a set of commands, which are defined for the agent's class. You can invoke these commands within your TOOL code by using methods defined for the SystemAgent class.

Do not rely on command output

The output for commands are not considered a programmatic interface. The printed output for a command is not documented, so you should not rely on any particular format or content of the command output in your application. Forte does not guarantee that the command output will remain the same between releases or for different international locales.

For a list of agent commands provided by Forte-defined agents that can be used in TOOL code, see *Escript and System Agent Reference Manual*.

Getting a List of Agent Commands

SystemAgent. GetCommands method

Unlike methods, commands exist only at runtime, so the only way to determine what commands are available for a specific agent at runtime (besides looking at the agent's source code or documentation) is to use a method inherited from the SystemAgent class called GetCommands. The syntax for this method is described in the Forte online Help.

The GetCommands method returns a GenericArray of CommandDesc. The CommandDesc class is a part of the Framework library, and is described in the Forte online Help. This class has attributes that contain information about the command, such as command name, attributes, help, and so forth.

You can derive information from the array of CommandDesc objects to generate command requests to the agent that owns these commands.

Invoking Commands on the Current Agent

SystemAgent. ExecuteCommand method

Each agent inherits an ExecuteCommand method from SystemAgent, which lets you invoke any command supported for a given agent. The syntax for this method is described in the Forte online Help.

The following TOOL code shows how you can submit a command string and a string argument as the command parameter. i.ClientID is the name of an active client.

```
cmdString : TextData = new(value='CancelClient ');
cmdString.concat(source=i.ClientID);
self.BankAgent.ExecuteCommand(command=cmdString);
```

See AgentBanking example: **Project:** AgentBanking • **Class:** AdminWindow • **Method:** Display

The following TOOL code shows how you can submit a command string as the command parameter and have any command output placed in the location specified for the outStream parameter:

```
cmdString : TextData = new(value='ListAllAccounts');
outPutStream : MemoryStream = new;
outPutStream.IsAnchored=TRUE;
outPutStream.Open(accessMode=SP_AM_WRITE);
self.BankAgent.ExecuteCommand(command=cmdString, objList=NIL,
  outStream=outPutStream);
outPutStream.Close();
```

See AgentBanking example: **Project:** AgentBanking • **Class:** AdminWindow • **Method:** Display

Invoking Commands on Subagents

SystemAgent.ExecCmdOnSubAgents method

Each agent inherits an ExecCmdOnSubAgents method from SystemAgent, which invokes the specified command on all of a given agent's subagents. This command is useful when all the subagents are the same kind of agent, and therefore support the same commands. For example, you could invoke a **ModLogger** command on the subagents of an Installed Partition agent to modify the logger flags on all of its Active Partition agents.

For a complete description of this method, see the Forte online Help.

Accessing and Updating Instrument Data

Agents can have six different types of instruments, each of which maps to a SystemMonitor library class, as shown in the following table.

Instrument Type	Instrument Subclass	Description
Average	AverageInst	Read only. Contains an average value.
Compound	CompoundInst	Contains a static set of instruments. These instruments can be of different types.
Configuration	ConfigValueInst	Read/write or read only. Contains a simple value, such as an integer value or a TextData object.
Counter	CounterInst	Read only. Contains a value based on counting something.
SubObject	SubObjectInst	Contains a dynamic set of instruments. All instruments are of the same type.
Timer	TimerInst	Read/write. Timer that prompts the agent to do something after a certain interval or set of intervals.

For more information about each type of instrument, see the class for that instrument in the Forte online Help

Each agent can have any number of instruments, and instruments that are Compound or SubObject instruments can have any number of subinstruments.

For a list of the agent instruments provided by Forte-defined agents, see *Escript and System Agent Reference Manual*.

Locating the Instrument to Access or Update

SystemAgent. Instrumentation attribute

As with commands, agent instruments exist only at runtime. Therefore, the only way to determine what instruments are available for a specific agent at runtime (besides looking at the agent's source code or documentation) is to use an attribute inherited from SystemAgent called Instrumentation (GenericArray of Instrument), which contains a list of the instruments attached to the agent. This attribute is also described in the Forte online Help.

Each object of the Instrument class contains information about the instrument, such as its name, the agent that owns it, any instruments that contain this instrument, and so forth.

Referencing an Instrument

SystemAgent. FindInstrument method

Each agent inherits an overloaded method from SystemAgent called FindInstrument. This method returns an instrument based on either the instrument's ID or its name. Each time you invoke this method, the Forte system updates the instrument's values.

Syntax **FindInstrument(instId** = *integer*) **:** *Instrument*

Syntax **FindInstrument(name** = *string* | *TextData*) **:** *Instrument*

For a complete description of this method, see the Forte online Help.

The following example shows how you can get a reference to an instrument using its name:

```
tmpInst1 = ConfigValueInst(self.BankAgent.
  FindInstrument(name='MaxClientSessionLength'));
```

See AgentBanking example: **Project:** AgentBanking • **Class:** AdminWindow • **Method:** Display

Accessing and Updating Configuration Instruments

Configuration instruments are the only instruments that can contain data values and yet be updated by users. When you get a reference to a Configuration instrument that is defined as an Instrument object, you need to cast the object to the ConfigValueInst class.

ConfigValueInst.GetData method

To access the value of a Configuration instrument, use the GetData method provided by the ConfigValueInst class.

Syntax **GetData() :** *DataValue*

For a complete description of this method, see the Forte online Help.

The following example shows how you could use the GetData method to get instrument data to include in a text string:

```
textString.Concat(source=instrument.Name);
textString.Concat(source=' = ');
textString.Concat(source=ConfigValueInst(instrument).GetData());
```

ConfigValueInst. UpdateData method

To update the value of a Configuration instrument, use the UpdateData method provided by the ConfigValueInst class.

Syntax **UpdateData(data** = *DataValue***)**

For a complete description of this method, see the Forte online Help.

The following example shows how you could use the UpdateData method to change the data value of a Configuration instrument that contains an integer value:

```
intData : IntegerData = new(value=self.MaxClientNumber);
tmpInst2.UpdateData(Data = intData);
```

See AgentBanking example: **Project:** AgentBanking • **Class:** MaxUpdateWin • **Method:** Display

Accessing Average and Counter Instruments

Because Average and Counter instruments contain only read-only numeric values, you can access their values using attributes. When you get a reference to an Average or Counter instrument that is defined as an Instrument object, you need to cast the object to the appropriate AverageInst or CounterInst class.

Retrieving Values

NumericInst.RealValue and NumericInst.Value attributes

Both the AverageInst and CounterInst classes inherit two attributes from the NumericInst class for accessing the current value of the instrument:

Attribute	Data Type	Description
RealValue	double	Represents the value as a double.
Value	i4	Represents the value as an integer.

For complete information about these attributes, see the Forte online Help.

The following TOOL code from the AgentAccess example program shows how you can use the RealValue attribute to retrieve data from an Average or Counter instrument and log it:

```
-- Log the Average (AverageInst) or Counter (CounterInst)
-- instrument value.
textString : TextData = new;
textString.Concat(source=instrument.Name);
textString.Concat(source=' = ');
```

```
textString.Concat(source=NumericInst(instrument).RealValue);
self.logFile.WriteLine(source=textString);
```

Setting Ranges for Values

NumericInst.LowerLimit and NumericInst.UpperLimit attributes

You can set maximum and minimum values for an Average instrument or a Counter instrument using the LowerLimit (i4) and UpperLimit (i4) attributes that the AverageInst and CounterInst classes inherit from the NumericInst class. These attributes are described in the Forte online Help.

If the instrument's value goes above or below the range defined by these attributes, then the instrument's agent posts an InstrumentAlert event, described in the Forte online Help.

Accessing a Compound Instrument and its SubInstruments

A Compound instrument is different than other instruments in that it contains a variety of other instruments, much the way a C struct contains a variety of other data types. When you get a reference to a Compound instrument that is defined as an Instrument object, you need to cast the object to the CompoundInst class.

CompoundInst. SubInstruments attribute

To access the instruments contained by the Compound instrument, you need to use the SubInstruments attribute (GenericArray of Instrument) provided by the CompoundInst class. This attribute, which is described in the Forte online Help, contains a list of the instruments contained by the Compound instrument. You can interact with each subinstrument as you would with any instrument of that type that is not a subinstrument.

The following TOOL code shows how you can determine the types of the subinstruments of a Compound instrument and pass the instruments to the appropriate user-defined routines for further processing:

```
-- Determine what kind of instruments are subinstruments
-- of this compound instrument.
for i in CompoundInst(instrument).SubInstruments do
  if i.IsA(AverageInst) then
    self.LogNumInst(i, currIndentLevel);
  elseif i.IsA(CompoundInst) then
    self.LogCompoundInst(i, currIndentLevel);
  elseif i.IsA(ConfigValueInst) then
    self.LogConfigVInst(i, currIndentLevel);
  elseif i.IsA(CounterInst) then
    self.LogNumInst(i, currIndentLevel);
  elseif i.IsA(SubObjectInst) then
    self.LogSubObjInst(i, currIndentLevel);
  elseif i.IsA(TimerInst) then
    self.LogTimerInst(i, currIndentLevel);
  else
    task.part.logmgr.putline('Unidentifiable classtype here.');
  end if;
end for;
AgentLogMgr.LogCompoundInst
```

See AgentAccess example: **Project:** AgentAccessSvc • **Class:** AgentLogMgr • **Method:** LogCompoundInst

After you determine the kind of instrument that the Compound instrument contains, you can interact with the instrument just as you would if it was not part of a Compound instrument.

Accessing a SubObject Instrument and its Subobjects

A SubObject instrument contains a group of instruments, typically of the same type, that represent information about objects in the system that are transient. For example, a typical way to use a SubObject instrument is to store information about currently running tasks. When you get a reference to a SubObject instrument that is defined as an Instrument object, you need to cast the object to the SubObjectInst class.

SubObjectInst. ActiveObjects attribute

To access the instruments contained in the SubObject instrument, you need to use the ActiveObjects attribute (GenericArray of Instrument) of the SubObjectInst class. This attribute contains a list of the instruments that represent currently available objects of the type being monitored by this SubObject instrument, and is described in the Forte online Help.

In the following TOOL code from the AgentBanking example program, the SubObject instrument contains an array of Compound instruments containing information about currently logged-in clients:

```
clientList : SubObjectInst;
clientList =
  SubObjectInst(self.BankAgent.FindInstrument(
    name='ActiveClients'));
for i in clientList.ActiveObjects do -- For each Compound instrument
  tempClassSession : ClientSession = new;
  for j in CompoundInst(i).SubInstruments do  -- For each instrument
    if ConfigValueInst(j).Name.Value = 'ActiveClientName' then
      tempClassSession.ClientID =
        TextData(ConfigValueInst(j).GetData());
    else if ConfigValueInst(j).Name.Value =
      'ActiveClientSessionLength' then
      tempClassSession.SessionLength =
        (IntegerData(ConfigValueInst(j).GetData())).Value;
    else do
      task.part.logmgr.putline(
        'There is a problem in AdminWindow.Init.');
    end if;
    end if;
  end for;
  self.ClientSessions.AppendRow(object=tempClassSession);
end for;
```

See AgentBanking example: **Project:** AgentBanking • **Class:** AdminWindow • **Method:** GetClientList

Accessing historical data

A SubObject instrument can also contain historical data about subobjects that no longer exist in the system. You can use the HistoricalData attribute of the SubObjectInst class to access historical data in a SubObject instrument:

For more information about the HistoricalData attribute, see the Forte online Help.

Chapter 3

Developing Custom Agents

This chapter explains how to use SystemMonitor library classes to develop custom agents for monitoring a Forte system.

This chapter describes the steps involved in developing an agent, including how to:

- define commands on the agent
- define instruments for the agent

About Developing Custom System Agents

Forte provides an architecture for creating custom agents that you can use to manage your system. Typically, custom agents are subagents of an Active Partition agent. By using the steps and methods described in this chapter, you can develop custom agents that you can access and interact with using standard Forte system management facilities, such as the Environment Console and Escript.

You can design a custom agent to manage any object in the system. For example, you could develop a custom agent that tracks specific data about a shared service object in your application, which can help you tune the performance of your service object. You can define the data to be tracked by defining instruments. You can also define commands for your agent that let you perform tasks such as cancelling tasks, displaying current waiting events, or listing the current clients using the service object.

When you define a custom agent for a managed object, the agent must know about its managed object. The agent must be able to give data to and receive data from the managed object, as well as implement commands that affect the agent. However, the managed object, for the most part, knows nothing about its agent.

Testing an agent

After you develop an agent, you can test that agent by deploying and installing your application in your development environment. You cannot test an agent using any of the Forte workshops.

This section provides an overview of the steps required to develop custom system agents. The rest of this chapter leads you step-by-step through the process of developing a custom system agent. The code for the system agent used in this chapter is provided as an example called AgentBanking. For details about locating and using this example, see Appendix A, “Example Applications.”

Forte defines a standard interface of methods and attributes to all agents, including custom agents. These methods and attributes are defined on the SystemAgent class, and inherited by all the agent subclasses provided by Forte or defined by application developers.

Forte also defines a set of methods of the SystemAgent class which you can override in the custom agent subclass to define the commands and instruments for your custom agent.

To define a custom system agent, you define a subclass of the SystemAgent class and override these SystemAgent class methods to define and implement commands and instruments for this new agent.

The SystemAgent methods that you override for all new agents are:

Method	Description
Init	Contains processing that is executed when this agent object is initialized.
GetMOTypeName	Returns a string that contains the name of the type of object managed by this agent.

The SystemAgent methods that you override to define the commands for a new agent are:

Method	Description
InitCmdProcessor	Initializes the command processor and defines the call syntax and indexes of agent commands.
ProcessCmdRequest	Implements the commands defined in the InitCmdProcessor method.

The SystemAgent methods that you override to define instruments that can be set and accessed are:

Method	Description
AttachMO	Attaches the managed object to the agent and defines the name and instrument IDs of agent instruments.
UpdateInstrument	Obtains values from the agent's managed object and updates the specified instrument.
InstrumentUpdated	Provides a new value in the instrument to the managed object.

To define a new instrument, use the following classes:

- AverageInst
- CompoundInst
- ConfigValueInst
- CounterInst
- SubObjectInst
- TimerInst

These classes are all subclasses of the abstract class Instrument, and are described in the Forte online Help.

You should not subclass these instrument classes, because the Forte system management facilities, the Environment Console and Escript, only recognize these instrument classes.

Designing the Custom Agent

Before you start defining a custom agent, you need to determine what commands and instruments you want your agent to provide for managing this object. You also need to decide how your system manager will access this agent.

Selecting the Object to be Managed

You can design a custom agent to manage any object in the system. What object you want to develop an agent for depends on what kind of objects are available in the application and how the objects are partitioned and deployed. You should also consider how you plan to interact with the agent and what kinds of monitoring or managing you want to perform on the object.

Selecting a User Interface

Typically, your system manager will use the Environment Console or Escript to access the custom agent, but if she will use another system management tool or a customized user interface, you need to consider how these alternative interfaces will interact with the agent. For more information about accessing agents, see Chapter 2, "Accessing System Agents Using TOOL Code."

The AgentBanking example provides a customized user interface to the BankServiceAgent agent that it also defines.

Designing Commands

You need to determine which commands the system manager needs to use with the managed object. The SystemAgent class automatically provides the **DumpStatus** and **Shutdown** commands. However, you can decide what other commands are appropriate for the managed object.

For example, if the managed object is a shared service that manages data storage and retrieval, you might want to provide agent commands that list the tasks that are currently running or cancel a database task that is taking too long

Designing Instruments

You also need to determine which instruments the system manager needs to use to set values in the managed object and retrieve values from the managed object. Forte provides several Instrument subclasses that support different kinds of instruments, which are described in "Defining Instruments for the Custom Agent" on page 43.

For example, if the managed object is a shared service that manages data storage and retrieval, you might want to provide agent instruments that retrieve and set the maximum number of clients running at one time, retrieve the current database name, and retrieve the number of running queries.

Enhancing the Managed Object's Class

Although you implement commands within methods on the agent's class, you need to make sure that the managed object's class provides sufficient access to itself. You might need to add methods and attributes to the managed object's class so that you can implement the commands and instruments that you want the agent to provide.

For example, you might need to make some private methods of the managed object's class public so that the agent's methods can use those methods to access data in the managed object.

Testing the Custom Agent

You cannot test an agent using any of the Forte workshops. To test your agent, you need to deploy and install your application in your development environment.

Defining a Class for the Custom Agent

The first step for defining a class for the system agent is to create a new class that is a subclass of the SystemAgent class in the SystemMonitor library. You will usually define this agent in the project that contains the class for this agent's managed object.

You must include the SystemMonitor library as a supplier plan for the project in which you are defining an agent.

Define the class for the new agent as a nonwindow class, with SystemAgent class as its Superclass.

Within this new agent class, you need to define the following methods that override methods defined in the SystemAgent class:

- AttachMO
- GetMOTypeName
- Init
- InitCmdProcessor
- InstrumentUpdated
- ProcessCmdRequest
- UpdateInstrument

For information about defining commands, see "Defining Commands for the Custom Agent" on page 38. For information about defining instruments, see "Defining Instruments for the Custom Agent" on page 43. For information about writing code to initialize an agent and connect it to its managed object, see "Connecting the Custom Agent and its Managed Object" on page 56.

Writing the Init method

The Init method contains processing that is executed when the agent object is initialized.

Syntax

Init()

Set the managed object type: SystemAgent.SetMOType method

You can add TOOL code within this Init method to initialize an agent object as you would any other class. You can also use the SetMOType method to define the type of managed object that this agent should be connected to. Setting the type of managed object allows the system to check that you have connected an appropriate object to the agent. The SetMOType method is described in the Forte online Help.

For example, if you are creating an agent for the BankService service object in the AgentBanking example, you can use the following code in the Init method to define the kind of managed object expected for this agent object:

```
super.Init();
self.SetMOType(MOType=BankService);
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** Init

Forte invokes this Init method when this agent is initialized by the managed object, as explained in "Connecting the Custom Agent and its Managed Object" on page 56.

Writing the GetMOTypeName method

SystemAgent. GetMOTypeName method

The GetMOTypeName method returns a string that contains the name of the type of object managed by this agent.

Syntax **GetMOTypeName**(): *string*

The only processing you need to do in this method is to return the appropriate object type name to the calling method. For example, certain Forte system applications, such as the Environment Console and Escript, use this method to get the name of the type of the managed object. The GetMOTypeName method is also described in the Forte online Help.

You can define this method by simply including a **return** statement, as shown in the following example:

```
return 'BankServer';
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** GetMOTypeName

Of course, you can add additional code if you want this method to include more processing.

Defining Commands for the Custom Agent

To define commands for your new custom agent, you need to override two methods in your new agent class:

Method	Description
InitCmdProcessor	Initializes the command processor and defines the call syntax and indexes of agent commands.
ProcessCmdRequest	Implements the commands defined in the InitCmdProcessor method.

In the InitCmdProcessor method, you need to define the command name, syntax, and index, and register this information with the command processor.

In the ProcessCmdRequest method, you provide a **case** statement that uses the command index to identify the command to be processed. The **when** statement for each index contains the TOOL code that implements each command.

Forte automatically calls the InitCmdProcessor method as part of the SystemAgent.Init method.

When a client of this agent uses one of these agent commands, the client uses the ExecuteCommand method on this agent to pass the command syntax, as defined in the InitCmdProcessor method. The ExecuteCommand method then parses the command and passes the parameters to this agent's version of the ProcessCmdRequest method.

The following sections explain how you can define commands for your agents.

Writing the InitCmdProcessor Method

The InitCmdProcessor method initializes the command processor and defines the call syntax and indexes of agent commands.

Syntax InitCmdProcessor()

In this method, you define a set of commands that the processor can recognize and parse for this custom agent. For more information about the CommandProcessor class, see the Forte online Help.

Forte automatically initializes a CommandProcessor object and associates it with this agent during the SystemAgent.Init method.

Because this agent inherits commands from the SystemAgent class, and might inherit commands from another custom agent, you must always call the super.InitCmdProcessor method at the beginning of your InitCmdProcessor method.

DumpStatus and **Shutdown** commands

All agents automatically inherit two commands defined in the SystemAgent superclass:

Command Name	Command Index
DumpStatus	SystemAgent.CommandIndexValues.SM_DUMPSTATUS
Shutdown	SystemAgent.CommandIndexValues.SM_SHUTDOWN

These commands have no predefined function. If you want your agent to perform some processing when these commands are executed on the agent, you need to identify and implement these commands in the ProcessCmdRequest method, as described in "Writing the ProcessCmdRequest Method" on page 41.

CommandProcessor.AddCommand method

To add command definitions, you need to use the AddCommand method of the CommandProcessor class. The syntax for this method is described in the Forte online Help.

Uniqueness of command indexes

For each command that you define using the AddCommand method, you need to define a name and a command index that are both unique within the current branch of subclasses of the SystemAgent class. For example, if this custom agent is a subclass of another custom agent class, all the command indexes and command names must be unique between the two classes.

Range of command index values

Forte defines some commands on the SystemAgent class itself, and the number of the highest index value used for these system-defined commands is stored as an integer in SystemAgent.CommandIndexValues.AGENT_CMD_LASTVALUE. You can use this value to determine what values are available for your own agent commands. The command index numbers and instrument ID numbers are independent of each other, and can overlap.

In the following TOOL code, the command uses the CANCELCLIENT_CMD constant defined at the project level plus the value of the highest index used by Forte to define its command index value:

```
super.InitCmdProcessor();
systemindex : integer =
  SystemAgent.CommandIndexValues.AGENT_CMD_LASTVALUE;

self.CommandProcessor.AddCommand(cmdName = 'CancelClient',
  argDesc = 's',
  cmdIndex = systemindex + CANCELCLIENT_CMD,
  helpText =
    'Syntax: CancelClient <client_name>.\nCancels the specified
client.',
  argNames = 'clientName', groupName = 'BankServer',
    menuString = 'Cancel Client');
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** InitCmdProcessor

The command defined in this example is named CancelClient and has one required string argument with the name clientName. When you use the **Help** command in Escript, you can see the help text defined by the helpText parameter. This command appears in the Environment Console in the BankServer menu as a menu item named Cancel Client.

For information about defining command syntax using the AddCommand method of the CommandProcessor class, see the Forte online Help.

Commands in Escript and the Environment Console

Parameters of the CommandProcessor.AddCommand method

Escript and the Environment Console automatically let you navigate to your agent and its commands and instruments where they are attached into the agent hierarchy. How your agent commands appear and behave depends on how you define certain parameters of the CommandProcessor.AddCommand method.

Parameter	Description
cmdName	Defines the name of the command that is used in Escript.
helpText	Defines help text that can be accessed using the Escript **Help** command.
argDesc and argNames	Define the arguments for this command. The Environment Console displays these arguments' names in the Execute Command Dialog dialog, which prompts for the arguments' values. If a command specifies a return value, the command cannot be accessed by Escript or the Environment Console.
groupName	Defines the Environment Console menu under which the command is displayed. By default, the Environment Console displays user-defined commands under the Utility menu. In Figure 5, the Cancel Client command has been added to the BankServer menu.
menuString	Defines the string that the Environment Console displays on a menu. This string represents the command in the Environment Console. If a command has any arguments, the Environment Console automatically adds "…" to the end of the command, as shown for the Cancel Client command in Figure 6.

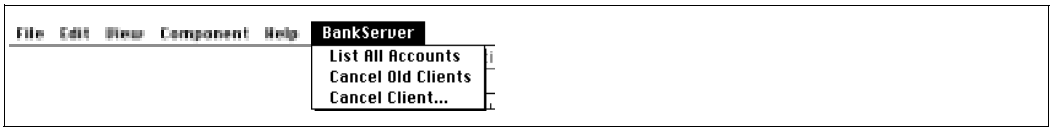

Figure 5 *User-defined Agent Commands in the Environment Console*

Note that the Cancel Client command is automatically followed by "…" in the menu because this list item opens a dialog, where you enter the argument required by this command, as shown in the following figure:

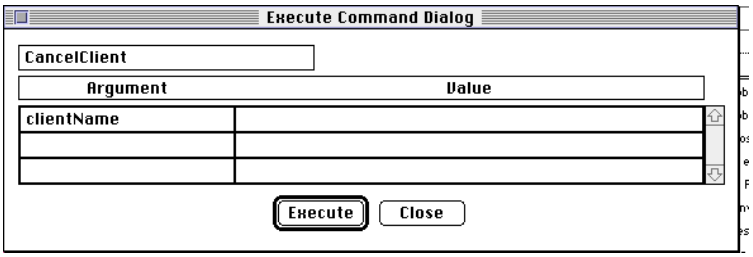

Figure 6 *User-defined Agent Command in its Execute Command Dialog*

For a complete description of the AddCommand parameters, see the Forte online Help.

Writing the ProcessCmdRequest Method

The ProcessCmdRequest method implements the commands defined in the InitCmdProcessor method.

Syntax **ProcessCmdRequest**(**cmdIndex**=*ui2*,
parameters=*GenericArray of Object*, **outStream**=*Stream*): *Object*

This ProcessCmdRequest method actually provides the implementation for commands defined in the InitCmdProcessor method, as described in "Writing the InitCmdProcessor Method" on page 38.

An application, such as Escript or the Environment Console, invokes an agent's command within TOOL code using the ExecuteCommand method. The Forte system then uses methods on the command processor to parse the command before passing the parameters to the agent's version of the ProcessCmdRequest method.

In this method, use a TOOL **case** statement, with a set of **when** clauses that check for the command index. In the **else** clause, you should include the overridden ProcessCmdRequest method.

The following example shows the structure for the ProcessCmdRequest method. The command indexes were defined by adding constants to the SystemAgent.CommandIndexValues.AGENT_CMD_LASTVALUE when each command was defined in the InitCmdProcessor method, as explained in "Writing the InitCmdProcessor Method" on page 38.

```
systemindex : integer =
  SystemAgent.CommandIndexValues.AGENT_CMD_LASTVALUE;

return_value : object = NIL;

case (cmdIndex - systemindex)
  when 100 do
  -- Processing for the command with an index of 100 + systemindex.
  ...
  when 200 do
  -- Processing for the command with an index of 200 + systemindex.
  ...
  else do
    return_value =
      super.ProcessCmdRequest(cmdIndex, parameters, outStream);

end case;
return return_value;
```

outStream parameter and command output

The outStream parameter of the ProcessCmdRequest method specifies where the output of the command should go. For Escript and the Environment Console, command output is printed to standard output (stdout). Because the outStream parameter is a stream, you can use the Stream.WriteText method within the ProcessCmdRequest method to print output to the output stream.

The following TOOL code shows the command implementation for a command named ListAllAccounts:

```
when LISTALLACCOUNTS_CMD do
    outStream.WriteText(source='Current accounts:\n');
    task.part.logmgr.putline(source='Current accounts:');
    for i in BankServer.AcctList do
      outtext : TextData = new;
      outtext.Concat(i.AcctNumber);
      outtext.Concat(' ');
      outtext.Concat(i.AcctName);
      outtext.Concat(' ');
      tmpDoubleData : DoubleData = new;
      tmpDoubleData.SetValue(source=i.AcctBalance);
      tmpNumFormat : NumericFormat = new;
      tmpTemplate : TextData = new(value='$#,##0.00');
      tmpNumFormat.Template = tmpTemplate;
      outtext.Concat(tmpNumFormat.
        FormatNumeric(source=tmpDoubleData));
      outtext.Concat('\n');
      outStream.WriteText(source=outtext);
      task.part.logmgr.putline(outtext);
    end for;
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** ProcessCmdRequest

You might need to reference attributes of the managed object using the ManagedObject attribute of the agent. Because the value of the ManagedObject attribute for this agent is of class Object, you need to cast the ManagedObject to the class of the managed object to access the attributes and methods of that class. The following line from the above example demonstrates casting the object as ManagedObject:

```
    task.lgr.put(BankService(self.ManagedObject).MaxClients);
```

DumpStatus and **Shutdown** commands

All agents inherit the following two commands defined in the SystemAgent superclass:

Command Name	Command Index
DumpStatus	SystemAgent.CommandIndexValues.SM_DUMPSTATUS
Shutdown	SystemAgent.CommandIndexValues.SM_SHUTDOWN

These commands have no predefined function. If you want your agent to perform some processing when these commands are executed on the agent, you need to identify these commands in **when** clauses of the **case** statement in the ProcessCmdRequest method.

If your agent is the subagent of any Forte-defined agent other than an Active Partition agent, the **DumpStatus** or **Shutdown** command on your agent is invoked automatically when its parent agent executes a **DumpStatus** or **Shutdown** command. The Active Partition agent does not automatically propagate the **DumpStatus** or **Shutdown** command to its subagents.

Defining Instruments for the Custom Agent

To define instruments for your new custom agent, you need to override two or three methods in your new agent class:

- AttachMO

  This method attaches the managed object to the agent and defines the name and instrument IDs of agent instruments.

  "Writing the AttachMO Method" on page 44 describes how to write this method. This method instantiates the instruments for this agent, defines its name and instrument ID, and adds the instrument to the agent.

  The AttachMO method is called by the Init method of the class of the managed object, to attach the managed object to the agent. The code you need to add to this Init method is explained in "Connecting the Custom Agent and its Managed Object" on page 56.

- UpdateInstrument

  This method obtains values from the agent's managed object and updates the specified instrument.

  "Writing the UpdateInstrument Method" on page 46 describes how to write this method. This method provides a **case** statement that uses the instrument ID to identify the instrument to be updated. The **when** clause for each instrument ID contains the TOOL code that retrieves data from the managed object and updates the identified instrument.

  When a client of this agent, such as Escript or the Environment Console, requests an instrument value, the system automatically calls the UpdateInstrument method, which obtains current data from the managed object, and updates the instrument value before returning this value to the client.

- InstrumentUpdated

  This method provides appropriate processing when an instrument is assigned a new value or when a Timer instrument ticks.

  "Writing the InstrumentUpdated Method" on page 53 describes how to write this method. This method provides a **case** statement that uses the instrument ID to identify the instrument whose value has been updated. The **when** clause for each index contains the TOOL code that transfers this new instrument value to the managed object.

  When a client of this agent, such as Escript or the Environment Console, updates one of its instruments, Forte automatically calls the InstrumentUpdated method, which updates the changed values in the managed object, as appropriate.

The following sections demonstrate how you can define instruments for your agents.

Forte Instrument Subclasses

Forte provides an abstract Instrument class, and also defines several subclasses of Instruments, which you use to define the instruments for your custom agent. You instantiate one of the following classes to define a new instrument:

Instrument Type	Instrument Subclass	Description
Average	AverageInst	Read only. Contains an average value.
Compound	CompoundInst	Contains a static set of instruments. These instruments can be of different types.
Configuration	ConfigValueInst	Read/write or read only. Contains a simple value, such as an integer value or a TextData object.
Counter	CounterInst	Read only. Contains a value based on counting something.
SubObject	SubObjectInst	Contains a dynamic set of instruments. All instruments are of the same type.
Timer	TimerInst	Read/write. Timer that prompts the agent to do something after a certain interval or set of intervals.

You cannot subclass any of these Forte subclasses of the Instrument class or the Instrument class itself. For more information, see the Forte online Help.

Writing the AttachMO Method

The AttachMO method attaches the managed object to the agent and defines the name and instrument IDs of agent instruments.

Syntax **AttachMO(managedObject**=*Object*)

SystemAgent.AddInstrument method

In the AttachMO method, you instantiate the instruments for this agent, define each instrument's name and instrument ID, and add the instrument to the agent. You use the AddInstrument method of the SystemAgent class to add the instrument to the agent.

Because this agent could inherit instruments from the SystemAgent class, and might inherit instruments from another custom agent, you must always call the overridden AttachMO method at the beginning of your AttachMO method.

Uniqueness of instrument IDs

For each instrument that you define in the AttachMO method, you need to define a name and an instrument ID that are both unique within the current branch of subclasses of the SystemAgent class. For example, if this custom agent is a subclass of another custom agent class, all the instrument IDs and instrument names must be unique between the two classes. The instrument ID numbers and command index numbers are independent of each other, and can overlap.

The value of the highest instrument ID used by Forte for system-defined commands is stored as an integer in SystemAgent.InstrumentID.Values.AGENT_IID_LASTVALUE. You can use this value to determine what values are available for your own agent instruments.

In the following TOOL code, the instrument IDs are defined as constants at the project level.

```
super.AttachMO(managedObject);
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;

-- Add instrument that gets and sets the maximum clients.
MaxActiveClients : ConfigValueInst = new;
MaxActiveClients.Name.SetValue('MaxActiveClients');
MaxActiveClients.InstrumentID=(systemID + MAXACTIVECLIENTS_INST);
self.AddInstrument(MaxActiveClients);
...
-- Add SubObjectInst instrument that displays a dynamic array
-- of client info.
ActiveClients : SubObjectInst = new;
ActiveClients.Name.SetValue('ActiveClients');
ActiveClients.InstrumentID=(systemID + ACTIVECLIENTS_INST);
self.AddInstrument(ActiveClients);
...
-- Add CounterInst instrument to count transactions that have
-- occurred.
TransactionCount  : CounterInst = new;
TransactionCount.Name.SetValue('TransactionCount');
TransactionCount.InstrumentID=(systemID + TRANSACTIONCOUNT_INST);
self.AddInstrument(TransactionCount);

-- Add AverageInst instrument to get average length of client
-- sessions.
self.AddInstrument(TransactionCount);
AverageSessionLength : AverageInst = new;
AverageSessionLength.Name.SetValue('AverageSessionLength');
AverageSessionLength.InstrumentID=
  (systemID + AVERAGESESSIONLENGTH_INST);
self.AddInstrument(AverageSessionLength);

-- Add TimerInst instrument to check for clients that have been
-- logged on too long.
CheckSessionLengths : TimerInst = new;
CheckSessionLengths.Name.SetValue('CheckSessionLengths');
CheckSessionLengths.InstrumentID=
  (systemID + CHECKSESSIONLENGTHS_INST);
self.AddInstrument(CheckSessionLengths);
CheckSessionLengths.TickInterval=60000;
CheckSessionLengths.IsActive=TRUE;
```

See AgentBanking example **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** AttachMO

Note In this example, although the SubObject instrument will contain Compound instruments and their subinstruments, these instruments are not defined in the AttachMO method. These instruments are defined, instantiated, and added to the SubObject instrument in the UpdateInstrument method.

To define a Timer instrument and its attributes, follow these steps:

1. Attach the instrument to the agent using the AddInstrument method.
2. Define the TimerInst.TickInterval attribute.
3. Make the Timer instrument active by setting the TimerInst.IsActive attribute to TRUE.

Writing the UpdateInstrument Method

The UpdateInstrument method obtains values from the agent's managed object and updates the specified instrument.

Syntax **UpdateInstrument**(**inst**=*Instrument*)

In the UpdateInstrument method, you implement how the instrument obtains data from the managed object and is updated.

When an application, such as Escript or the Environment Console, requests an instrument value, the system automatically calls the UpdateInstrument method. This method obtains current data from the managed object and updates the instrument value before returning this value to the client application.

In the UpdateInstrument method, you provide a **case** statement that uses the instrument ID to identify the instrument to be updated. The **when** clause for each instrument ID contains the TOOL code that retrieves data from the managed object and updates its instrument. In the **else** clause, you should include the overridden UpdateInstrument method.

The following example shows the structure for this method. This example assumes that the instrument IDs were ensured to be unique by adding constants to the SystemAgent.InstrumentID.Values.AGENT_IID_LASTVALUE when each instrument was defined in the AttachMO method, as explained in "Writing the AttachMO Method" on page 44.

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;

case (inst.InstrumentID - systemID)
  when 100 do
  -- Processing to update the instrument with an ID of 100 +
systemID.
  ...
  when 200 do
  -- Processing to update the instrument with an ID of 200 +
systemID.
  ...
  else do
    super.UpdateInstrument(inst);
end case;
```

Each of the subclasses of Instrument that you can use to define instruments has specific methods for changing the instrument values.

Updating Configuration Instruments

ConfigValueInst.IsReadOnly attribute

The Configuration instrument is the only instrument that can be read only or read-write. The ConfigValueInst class provides an IsReadOnly attribute (boolean) that lets you change whether the Configuration instrument is read only (TRUE) or not (FALSE). Note that if you want to update your read-only Configuration instrument in the UpdateInstrument method, you need to set the IsReadOnly attribute to FALSE, update the value, then set the IsReadOnly attribute to TRUE. For more information about this attribute and the ConfigValueInst class, see the Forte online Help.

ConfigValueInst.UpdateData method

To update the value of the Configuration instrument, use the UpdateData method of the ConfigValueInst class. To use this method, the IsReadOnly attribute must be set to FALSE. For a complete description of this method, see the Forte online Help. The UpdateData method requires an input parameter of the DataValue class or of a DataValue subclass defined by Forte.

Note

You should update the Configuration instrument using a Forte DataValue subclass to contain the new value. If you update a Configuration instrument value using a user-defined DataValue subclass, the Environment Console and Escript cannot access this value.

The following example shows how you could update the value of the MaxActiveClients instrument defined in the AttachMO method, as shown in "Writing the AttachMO Method" on page 44:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
case (inst.InstrumentID - systemID)

  when MAXACTIVECLIENTS_INST do
    tmp : integerdata = new;
    tmp.value = BankService(self.ManagedObject).MaxClients;
    ConfigValueInst(Inst).UpdateData(tmp);
    ...
end case;
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** UpdateInstrument

Updating Average Instruments

The Average instrument is a read-only instrument that contains the running average of one or more values that are included in the average using the RecordValue method.

AverageInst.ClearValue and AverageInst.RecordValue attributes

The AverageInst class has two methods you can use to change the value contained by this instrument:

Method	Description
ClearValue	Sets the value of the instrument to zero.
RecordValue	Recalculates the running average to include the specified value.

The Average instrument automatically recalculates the average of the values that have been included when you invoke the RecordValue method or access the RealValue and Value attributes of the AverageInst class.

For more information about these methods and the AverageInst class, see the Forte online Help.

The following TOOL code shows how you can update the value of an Average instrument using the ClearValue and RecordValue methods. In this example, the value of the instrument AverageSessionLength is the average of the session length of all the current clients and all terminated client sessions:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
case (inst.InstrumentID - systemID)
...
  when AVERAGESESSIONLENGTH_INST do
    (AverageInst(inst)).ClearValue();
    task.part.logmgr.putline(
      'Adding values to AverageSessionLength instrument');
    task.part.logmgr.putline('Active client sessions:');
    if (BankServer.ActiveClients.Items = 0) and
      (BankServer.TerminatedSessionLengths.Items = 0) then
      (AverageInst(inst)).RecordValue(value=0);
    else do
      for i in BankServer.ActiveClients do

(AverageInst(inst)).RecordValue(value=i.GetLengthOfSession());
       task.part.logmgr.putline(i.GetLengthOfSession);
      end for;
      task.part.logmgr.putline('Terminated client sessions:');
      for i in BankServer.TerminatedSessionLengths do
        (AverageInst(inst)).RecordValue(value=i.IntegerValue);
        task.part.logmgr.putline(i.IntegerValue);
      end for;
    end if;
  else do
    super.UpdateInstrument(inst);
...
end case;
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** UpdateInstrument

Updating Counter Instruments

The Counter instrument is a read-only instrument that counts integer values.

CounterInst.ClearValue, CounterInst.Increment, and CounterInst.Decrement

The CounterInst class has three methods that you can use to update the Counter instrument value:

Method	Description
ClearValue	Sets the value of the instrument to zero by clearing the list of values being averaged.
Increment	Increases the value of the instrument by a specified amount.
Decrement	Decreases the value of the instrument by a specified amount.

For more information about these methods and the CounterInst class, see the Forte online Help.

The following TOOL code shows how you can update the value of an Counter instrument using the Increment method. In this example, the UpdateInstrument method uses the data stored in the BankServer.TransactionCount attribute to determine the amount by which the value of TransactionCount instrument should be incremented:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
case (inst.InstrumentID - systemID)
...
when TRANSACTIONCOUNT_INST do

(CounterInst(inst)).Increment(amount=BankServer.TransactionCount);
    BankServer.TransactionCount=0;
  else do
    super.UpdateInstrument(inst);
...
end case;
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** UpdateInstrument

Updating Compound Instruments

CompoundInst class methods

The Compound instrument is a container for other related instruments. You could think of a Compound instrument as a struct of instruments. The CompoundInst class provides two methods for adding and removing instruments from the Compound instrument:

Method	Description
AddSubInstrument	Adds an instrument to the Compound instrument's set of instruments.
DeleteSubInstrument	Removes an instrument from the Compound instrument's set of instruments.

For more information about these methods and the CompoundInst class, see the Forte online Help.

A Compound instrument is not usually initialized until the first time the Forte system calls the UpdateInstrument method. In the UpdateInstrument method, you include the TOOL code that defines the Compound instrument's subinstruments, initializes them, and adds them to the Compound instrument's set of subinstruments.

Instruments that are subinstruments of a Compound instrument are individually updated the same way they would be if they were not subinstruments.

The following TOOL code shows how you can update a Compound instrument. In this example, the Compound instrument is a subobject of a SubObject instrument, and the UpdateInstrument method calls the RefreshActiveClientInst method to update the SubObject instrument:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
case (inst.InstrumentID - systemID)
  ...
  when ACTIVECLIENTS_INST do
    tmp : SubObjectInst = SubObjectInst(inst);
    self.RefreshActiveClientInst(activeClientInst=tmp);
  ...
end case;
```

See AgentBanking example **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** UpdateInstrument

The following TOOL code shows how the Compound instrument is updated within the RefreshActiveClientInst method, which has the following header:

```
RefreshActiveClientInst(
  input output activeClientInst:SubObjectInst)
```

The RefreshActiveClientInst method derives the values of the ActiveClientName and ActiveClientSessionLength instruments from the attributes of the ClientInfo objects in the BankServer.ActiveClients array:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
...
for i in BankServer.ActiveClients do
  ActiveClientInfo : CompoundInst = new;
  ActiveClientInfo.Name.SetValue('ActiveClientInfo');
  ActiveClientInfo.InstrumentID=(systemID + ACTIVECLIENTINFO_INST + i);

  -- For each ActiveClientInfo instrument, instantiate and evaluate
  -- an ActiveClientName and ActiveClientSessionLength, and add to the
  -- ActiveClientInfo instrument as subinstruments.
  ActiveClientName : ConfigValueInst = new;
  ActiveClientName.Name.SetValue('ActiveClientName');
  ActiveClientName.InstrumentID=(systemID + ACTIVECLIENTNAME_INST + i);
  ActiveClientName.UpdateData(data=i.ClientName);
  ActiveClientName.IsReadOnly = TRUE;
  ActiveClientInfo.AddSubInstrument(subInst=ActiveClientName);

  ActiveClientSessionLength : ConfigValueInst = new;
  ActiveClientSessionLength.Name.SetValue('ActiveClientSessionLength');
  ActiveClientSessionLength.InstrumentID=(systemID +
    ACTIVECLIENTSESSIONLENGTH_INST+ i);
  tempdata : IntegerData = new(Value = i.GetLengthOfSession());
  ActiveClientSessionLength.UpdateData(data=tempdata);
  ActiveClientSessionLength.IsReadOnly = TRUE;
  ActiveClientInfo.AddSubInstrument(subInst=ActiveClientSessionLength);
...
end for;
```

See AgentBanking example

Project: AgentBankServices • **Class:** BankServerAgent • **Method:** UpdateInstrument

Updating SubObject Instruments

The SubObject instrument is a container for a set of instruments of the same type, much like an array of instruments. You can define a SubObject instrument that contains, for example, a set of Compound instruments, each containing information about a running task.

SubObjectInst class methods

The SubObjectInst class provides methods for adding and removing instruments from the SubObject instrument:

Method	Description
AddActiveObject	Adds an instrument to the SubObject instrument's set of instruments.
DeleteActiveObject	Removes an instrument from the SubObject instrument's set of instruments.

For more information about these methods and the SubObjectInst class, see the Forte online Help.

A SubObject instrument is not usually initialized until the first time the Forte system calls the UpdateInstrument method. In the UpdateInstrument method, you include the TOOL code that defines the SubObject instrument's subobjects, initializes them, and adds them to the SubObject instrument's set of active objects.

Instruments that are subobjects of a SubObject instrument are individually updated the same way as they would be if they were not subobjects.

The following TOOL code from the AgentBanking example program shows how you can update a SubObject instrument. In this example, the SubObject instrument contains a set of Compound instruments, and the UpdateInstrument method calls the RefreshActiveClientInst method to update the SubObject instrument and its subobjects:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
case (inst.InstrumentID - systemID)
...
when ACTIVECLIENTS_INST do
    tmp : SubObjectInst = SubObjectInst(inst);
    RefreshActiveClientInst(activeClientInst=tmp);
...
end case;
```

See AgentBanking example

Project: AgentBankServices • **Class:** BankServerAgent • **Method:** UpdateInstrument

The following TOOL code shows how the ActiveClients SubObject instrument is updated within the RefreshActiveClientInst method, which has the following header:

```
RefreshActiveClientInst(
  input output activeClientInst:SubObjectInst)
```

The RefreshActiveClientInst method creates the Compound instruments, as shown in "Updating Compound Instruments" on page 49, then adds each Compound instrument to the SubObject instrument, as shown:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;

-- Clear ActiveClients SubObjectInst instrument array
activeClientInst.ActiveObjects.Clear();
for i in BankServer.ActiveClients do
...
-- For each registered client, create an ActiveClientInfo
-- instrument and add them to the ActiveClients instrument array.
...
  -- Add the new ActiveClientInfo compound instrument to the
     ActiveClients
  -- instrument array.
  activeClientInst.AddActiveObject(objInst=ActiveClientInfo);

end for;
```

See AgentBanking example

Project: AgentBankServices • **Class:** BankServerAgent • **Method:** UpdateInstrument

Maintaining historical data

A SubObject instrument can also contain historical data about subobjects that no longer exist in the system. You can use the following attributes and methods of the SubObjectInst class to maintain historical data in a SubObject instrument:

Attribute/Method	Description
CollectionType attribute	Indicates the type of historical data being collected.
AddHistoricalObject method	Adds an instrument containing some collected historical data.
ClearHistory method	Clears any collected historical data.

For more information about this attribute and these methods, see the Forte online Help.

Writing the InstrumentUpdated Method

The InstrumentUpdated method provides the appropriate processing when an instrument is assigned a new value or when a Timer instrument ticks.

Syntax **InstrumentUpdated(inst=***Instrument***)**

In the InstrumentUpdated method, you implement how Forte transfers the changed instrument value to the managed object.

When a client of this agent updates one of its instruments, Forte automatically calls the InstrumentUpdated method, which updates the changed values in the managed object, as appropriate. Of the Forte-defined instrument types, only Timer and Configuration instruments have attributes can be updated from outside the agent.

In the InstrumentUpdated method, you provide a **case** statement that uses the instrument ID to identify the instrument whose value has been updated. The **when** clause for each index contains the TOOL code that transfers this new instrument value to the managed object. In the **else** clause, you should include the overridden UpdateInstrument method.

The following example shows the structure for the InstrumentUpdated method. The instrument IDs were ensured to be unique by adding constants to the SystemAgent.InstrumentID.Values.AGENT_IID_LASTVALUE when each instrument was defined in the AttachMO method, as explained in "Writing the AttachMO Method" on page 44.

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;

case (inst.InstrumentID - systemID)
  when 100 do
  -- Processing to retrieve the new value if the instrument
  -- has an ID of 100 + systemID and to perform processing
  -- triggered by this new value.
  ...
  when 200 do
  -- Processing to retrieve the new value if the instrument
  -- has an ID of 200 + systemID and to perform processing
  -- triggered by this new value.
  ...
  else do
    super.InstrumentUpdated(inst);
end case;
```

See AgentBanking example **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** InstrumentUpdated

All of the subclasses of Instrument that you can use to define instruments have specific methods for obtaining the instrument values.

Handling an Updated Configuration Instrument

To update a value in the managed object that maps to an updated Configuration instrument, you need to write a routine for a **when** clause of the **case** statement in the InstrumentUpdated method that identifies an instrument that has been updated.

ConfigValueInst.GetData method

Within this routine, you need use the GetData method provided by the ConfigValueInst class to access the updated value of the Configuration instrument.

Syntax **GetData**() **:** *DataValue*

For a complete description of the GetData method, see the Forte online Help.

The following TOOL code shows how you could use the GetData method to get the updated instrument value. This method then sets the BankServer.MaxClients attribute to the new value:

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
case (inst.InstrumentID - systemID)
  when MAXACTIVECLIENTS_INST do
    tmp : DataValue = ConfigValueInst(inst).GetData();
    BankServer.MaxClients = tmp.TextValue.IntegerValue;
...
  else do
    super.UpdateInstrument(inst);
end case;
```

See AgentBanking example **Project:** AgentBankServices • **Class:** BankServerAgent • **Method:** InstrumentUpdated

Handling an Updated Timer Instrument

If you are writing a routine for a **when** clause that identifies a Timer instrument, be aware that the Forte system calls the InstrumentUpdated method only when the Timer instrument ticks. The Forte system automatically handles changes to the TickInterval and IsActive attributes of a Timer instrument. Therefore, in your routine, you only need to write the TOOL code that performs the tasks that you want to occur when the timer ticks.

The following TOOL code shows the processing that the BankServerAgent performs when the CheckSessionLengths Timer instrument ticks. In this example, the agent posts the SessionTooLong event if an active client session has be active longer than the defined maximum session length.

```
systemID : integer =
  SystemAgent.InstrumentIDValues.AGENT_IID_LASTVALUE;
case (inst.InstrumentID - systemID)
...
  -- When the CheckSessionLengths timer ticks, check whether
  -- any agents have exceeded the maximum.
  when CHECKSESSIONLENGTHS_INST do
    for i in BankServer.ActiveClients do
      if i.GetLengthOfSession() > BankServer.MaxClientTimeMin then
        post self.SessionTooLong(i.ClientName);
        task.part.logmgr.put('Client ');
        task.part.logmgr.put(i.ClientName);
        task.part.logmgr.put(' has been logged on for more than ');
        task.part.logmgr.put(BankServer.MaxClientTimeMin);
        task.part.logmgr.putline(' minutes (maximum).');
      end if;
    end for;
  else do
    super.UpdateInstrument(inst);
end case;
```

See AgentBanking example

Project: AgentBankServices • **Class:** BankServerAgent • **Method:** InstrumentUpdated

Connecting the Custom Agent and its Managed Object

After you have defined your custom agent class, you can instantiate an agent of this type and attach any object type that is compatible with the value you set with the SetMOType method, as described in "Writing the Init method" on page 36.

To connect a custom agent and its managed object, you must add TOOL code to the Init method belonging to the class of the managed object. This TOOL code must perform the following steps in the following order.

Testing a custom agent

If you want to test your agent, you need to deploy and install your application in your development environment. You cannot test agents within the Forte Workshops.

To connect a custom agent and its managed object, add code to the Init method belonging to the class of the managed object:

1 Instantiate the custom agent class, as shown in the following example:

```
BankingServAgent : BankServerAgent = new;
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

2 Set the name of the agent by setting its Name attribute, as shown:

```
BankingServAgent.Name.SetValue('BankServiceAgent');
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

You could also set the name of the agent in the Init method of the custom agent class, or in the AttachMO method of the custom agent class.

3 Attach the object to which this Init method belongs to the newly-instantiated agent as the agent's managed object, using the overridden AttachMO method of this agent's class, as shown:

```
BankingServAgent.AttachMO(self);
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

For information about the AttachMO method of the SystemAgent class, see the Forte online Help.

4 Set the state of this agent to online, using a Forte constant, as shown:

```
BankingServAgent.State = SM_ONLINE;
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

For information about the SystemAgent.State attribute, see the Forte online Help.

5 Get a reference to an agent that you want to define as the parent agent of your custom agent.

In the following example, the parent agent is the Active Partition agent, as shown:

```
partAgent : ActivePartitionAgent;
partAgent = ActivePartitionAgent(task.Part.ActPartAgent);
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

For information about navigating around the agent hierarchy, see "Navigating Around the Agent Hierarchy" on page 22.

6 Set the agent defined in Step 5 as the parent agent of this custom agent, using the SetParentAgent method of the SystemAgent class on the custom agent, as shown:

```
BankingServAgent.SetParentAgent(parentAgent = partAgent,
  name = partAgent.Name);
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

For information about the SystemAgent.SetParentAgent method, see the Forte online Help.

7 Define this custom agent as the subagent of the parent agent, using the AddSubAgent method of the SystemAgent class on the parent agent, as shown:

```
partAgent.AddSubAgent(subAgent = BankingServAgent,
  subAgentInfo = BankingServAgent.GetInfo());
```

See AgentBanking example: **Project:** AgentBankServices • **Class:** BankService • **Method:** Init

For information about the SystemAgent.AddSubAgent method, see the Forte online Help.

Developing Agents for Load-Balanced Service Objects

If you have a service object that is load balanced, you might want to develop a custom agent that provides more statistical information about the status and load for a load-balanced service object than the Load Balance Router agent provides.

In this case, you should define another environment-visible service object that will be the managed object of the custom agent. The custom agent attaches itself to this service object as though it is managing this service object. The managed object contains statistical information about the load-balanced service object. For example, the service object could contain a list of the currently active replicates and information about their loads over a given period of time. The load-balanced service object would contain code that calls methods on the managed object to register information about itself. Each replicate of the load-balanced service object could then report its data to this managed object.

When you implement the agent, you can implement agent instruments that retrieve information from the managed object, which reflects information from the load-balanced service object. In this case, you should only implement commands that manage the statistical information stored in the managed object. You cannot implement instruments for this agent that change data known in the load balanced service object.

When you partition the application containing the load-balanced service object and the agent and its managed object, you should place the environment-visible service object that is the managed object for the custom agent and the router for the load-balanced service object in the same partition. For more information about partitioning this kind of application, see *A Guide to the Forte 4GL Workshops*.

For example, if you knew that the BankServer service object was going to be load-balanced, you could define a service object that stores information about the load-balancing, such as the number of messages processed by each replicate. You could then add code to the BankServer service object to call a method on the new service object to update the number of messages processed by the BankServer service object at a specific timed interval. Each replicate would then update the data in the new service object regularly.

You could then define an agent that uses this new service object as its managed object. This new agent defines an instrument that contains a count of the messages for each replicate of the load balanced partition, and retrieves this information from its managed object in its implementation of the UpdateInstrument method.

Appendix A

Example Applications

This appendix provides instructions on how to install the examples that are used in this book to explain how to write and access system agents. Typically, you run an example application, then examine it in the various Forte Workshops to see how it is implemented. You can modify the examples if you wish.

Application Descriptions

This section lists the example applications in alphabetical order. Each example has five sections describing it.

The **Description** section defines the purpose of the example, what problem it solves, and what TOOL features and Forte classes it illustrates.

The **Pex Files** section gives you the subdirectory and file names of the exported projects. The examples are in subdirectories under the FORTE_ROOT/install/examples directory. You can import example applications individually if you wish. When multiple .pex files are listed, there are supplier projects in addition to the main project. You will need to import all the files listed to run the application. Import them in the order given so that dependencies will be satisfied.

The **Mode** section indicates whether the application can be run in either standalone or distributed mode, or whether it must be run in distributed mode.

The **Special Requirements** section identifies whether you need a database connection, an external file, or any other special setup.

Finally, the **To Use** section tells you how to step through the application's functions.

See the *Forte 4GL System Management Guide* if you need directions for setting up a Forte server.

AgentAccess

Description AgentAccess shows how to retrieve information from one or more system-management agents and log this data into a file.

Pex Files sysmon/agentasv.pex, sysmon/agentacc.pex.

Mode Standalone or Distributed.

Special Requirements You need to use Escript or the Environment Console to set up agent instruments to be logged by this application at regular intervals.

▶ **To use AgentAccess:**

1 Select instruments that will be logged by this application, and set them to be logged using the Environment Console or Escript. Set instruments for logging by first locating the agent that owns the instruments, then by setting each instrument's isLogged property to TRUE. For example, to log instrument data from the DistObjectMgr agent, you can use a series of Escript commands, like the following:

```
escript> ShowAgent
escript> FindSubAgent <node_name>
escript> showag
escript> findsub Forte_Executor
escript> showag
escript> findsub <partition_identifier>
escript> showag
escript> findsub DistObjectMgr
escript> showag
escript> SetInstrumentLogging MethodsReceived TRUE
escript> SetInstrumentLogging MethodsSent TRUE
```

This series of commands assumes that you have just started Escript, or are at the Environment Agent before you issue these commands. The **ShowAgent** command, abbreviated as **showag**, displays information about the current agent, including a list of instruments and subagents. The **FindSubAgent** command, abbreviated as **findsub**, moves to a subagent of the current agent, based on the name of the subagent.

To use this series of commands, you need to substitute the name of a node that has a running standard partition for <node_name>. You need to substitute the hexadecimal active partition identifier for <partition_identifier>. You can see the list of available active partitions in the listing provided by the previous **showag** command. For example, if the active partition name is Forte_Executor_0x14d, you can substitute 0x14d for <partition_identifer>.

For information about setting instruments for logging using Escript or the Environment Console, see *Forte 4GL System Management Guide*.

2 Set the LogTimer for the Active Partition agent where the ActivePartition or one of its subagents contains the instrument. Starting from the ending point in the above example, you can use a series of Escript commands like the following to set a LogTimer to tick every 30 seconds:

```
escript> FindPar
escript> UpdateInstrument LogTimer “TRUE 30000”
```

In this series of commands, the **FindParent** command (abbreviated as **findpar**) moves to the parent agent of the DistObjectMgr agent, which is the Active Partition agent. The LogTimer instrument has a property that determines whether it is ticking or not; the TRUE value enables the LogTimer. The value 30000 means that the LogTimer will tick every 30 seconds (30000 milliseconds).

For more information about setting the LogTimer for an Active Partition agent in Escript, see *Escript and System Agent Reference Manual*. For instructions for setting the LogTimer in the Environment Console, see *Forte 4GL System Management Guide*.

3 When you start AgentAccess, you should define the Log File name. If a file by the specified name already exists, the logged data is appended to the file. The default log file name is agent.log, and the file is always stored in FORTE_ROOT/log on the partition where the AgentLogSvc service object resides.

4 Select the Start button to start logging instrument data to your log file.

AgentBanking

Description AgentBanking shows how to implement an agent for the service object of a simple banking application. This application also provides a simple administrator’s window that lets the user manage the AgentBanking application using instruments and commands provided by the service object’s agent.

Pex Files sysmon/agentbsv.pex, sysmon/agentb.pex.

Mode Distributed only.

Special Requirements You need to install this application to be able to see that the agent is working. This application is designed to run with several clients in a distributed environment. While it will work with a single client, some features do not make sense with only one client running.

To use AgentBanking:

1. Partition the AgentBanking application with the AgentBanking project as the main project, using either the **Partition** command in the Repository Workshop or the **Partition** command in Fscript.
2. Make a distribution of the application and auto-install the application in your environment, using either the **Make Distribution** command in the Partition Workshop or the **MakeAppDistrib** command (with the arguments 1 "" "" 1) in Fscript.
3. Start the application using either the application icon (Windows), or a command like the following:

   ```
   ftexec -fi ct:$FORTE_ROOT/userapp/agentban/cl0/agentb0
   ```

   For more information about starting client partitions, see *Forte 4GL System Management Guide*.
4. In the Banking: Welcome! window, enter any numeric value for the user ID. (Note that each time you log in, you should enter a different user ID.) Select either the User or Administrator buttons, then select the Start button.

   If you login as a user, you can select an account to work with, then add or subtract amounts of money from the selected account.

   If you login as an administrator, you can perform actions that affect the running banking application by invoking commands and reading and updating instruments on the BankServiceAgent agent for the service object of the banking application.
5. You can also monitor the BankServiceAgent agent by using the Environment Console or Escript at the same time as this application and looking at the instrument values of the BankServiceAgent. This agent is a subagent of the Active Partition agent for the server partition of the banking application. For information about using the Environment Console or Escript, see *Forte 4GL System Management Guide* and *Escript and System Agent Reference Manual*.

Index

H

I

L

M

P

S

T

U