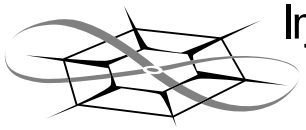
The University of Kansas



Information and Telecommunication Technology Center

Technical Report

ACE General Service Daemon Data Thread, Command Semantics and Client Command Design

Leon Searl and Gary Minden

ITTC-FY2001-TR-23150-03

January 2001

Project Sponsor:
U.S. Air Force and the Defense Advanced Research Projects Agency under contract no. F30602-00-2-0581
and The National Science Foundation
under grant EIA-9972843

1 Introduction

This document provides a 'general' design that can be applied to the two portions of an ACE Service Daemon that are unique to each daemon. The two components consist of the Data Thread and Command Semantics. Each daemon consists of an infrastructure that is common to all ACE Service daemons. The Data Thread component and Command Semantics 'implement' the behavior that is unique to each daemon. The Data Thread may use a low-level library to carry out some of its required behavior (example: Some daemons that communicate to devices through a serial port may use the ACE Serial library).

Also found in this design document is a design for a Java based ACE Client Command interface to send commands to ACE Daemons from an ACE Client.

Many of the requirements in the *'ACE Service Daemon Requirements*' are addressed in this design document.

This document is derived from the *'ACE Service Daemon Design Document'*. Since this document concerns a portion of the ACE Service Daemon, the portion unique to each daemon, it must interface with the remainder of the ACE Service Daemon, the portion common to all ACE daemons. The intra-daemon interface methods are found in the *'ACE Service Daemon Interface Specification'*. The ACE Daemon infrastructure invokes data thread methods to execute client commands. '`ACECmdLine`' classes, provided by the ACE Daemon infrastructure, are examined by data thread command methods and are created by these same methods as return values. The interface specification for manipulating command line classes is in the *'ACE Service Command Object Interface Specification'* document. The same document also specifies the interface to the '`ACECmdSemantics`' command semantics objects used in the daemon semantics constructor class.

The **Design** section describes the general design that all ACE Service Daemons follow. This general design is used to derive the specific ACE Service Daemon design documents. The **Glossary** section contains definitions for terms used in this document. The author, date and reasons for changing this document are found in the **Change Log** section. Any information that does not fit into any of the other sections can be found in the **Notes** section.

1.1 Notation

Document names appear in '*italic*'.

Source code identifiers appear in '`courier`'.

Class diagrams use the UML Class Diagram syntax.

2 Design

This design document primarily addresses the design of the Command Semantics and Data Thread components of the Thread ACE Service Daemon. It also includes design information for a Java version of a Client Command interface for sending commands to an ACE Service Daemon.

The **Overview** subsection describes where the Data Thread, Command Semantics and Client Command interface designs are integrated into the ACE Service Daemon design. The **Command Semantics** subsection describes the design of the Command Semantics Constructor class. Data Thread design is covered in the **Data Thread** subsection. A design for a Java based Client Command interface is in the **Client Command Interface** section.

2.1 Overview

This design concerns those components of ACE Service Daemons and ACE Clients that are unique to each client and daemon. These components include ACE Service Client object, ACE Service Daemon Data Thread and ACE Command Semantics. Each of these design topics is cover in its own subsection.

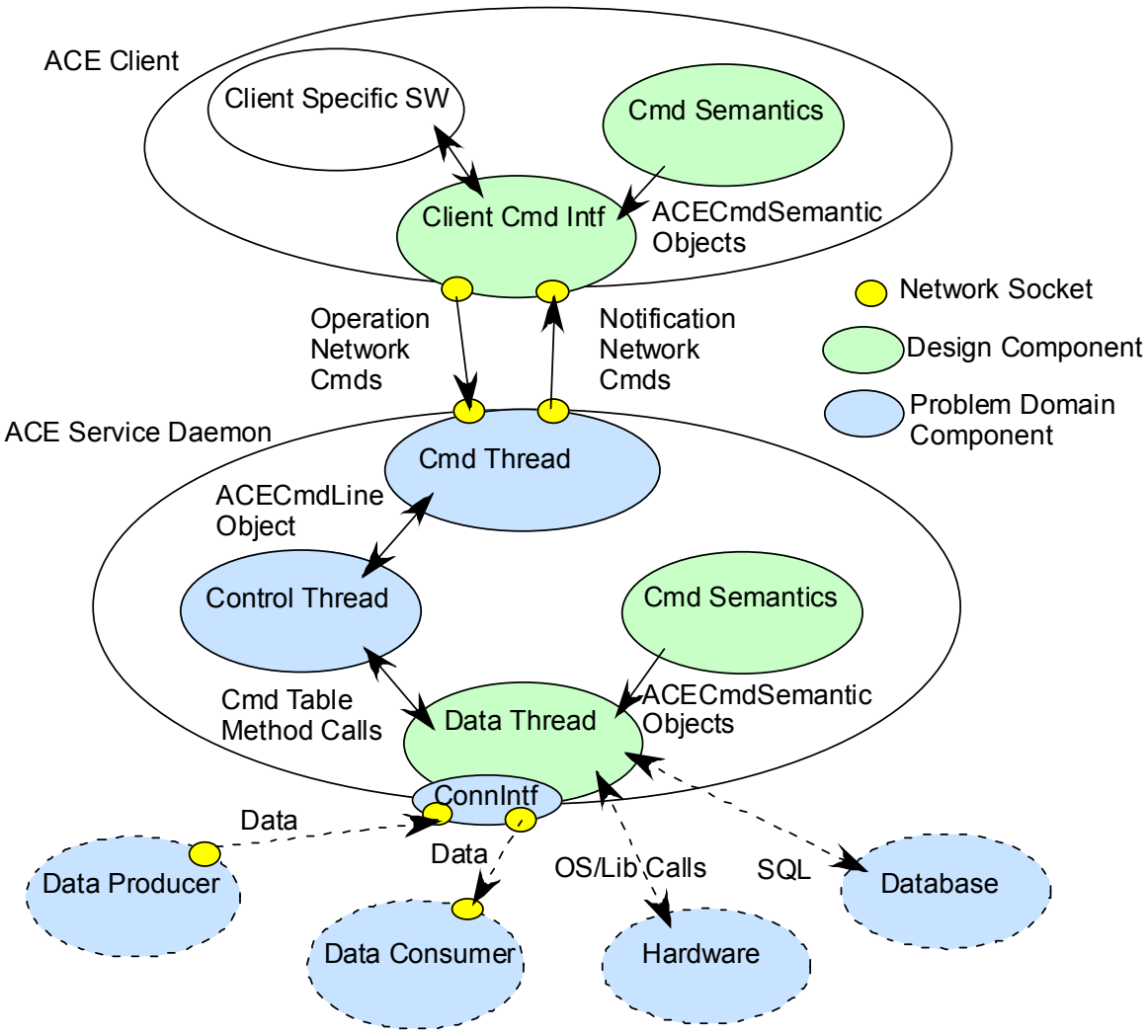


Figure 1 ACE Client and ACE Service Daemon Data Thread Design Components

Figure 1 shows the components described in this design plus the components that these design components interface to. There are two groupings of components in the diagram 1) ACE Client and 2) ACE Service Daemon. The components described in this design document are in green. Network sockets are shown in yellow. Components from the problem domain (outside of the scope of this design document) are shown in blue.

It is assumed by this document that an ACE Service Client has established a network connection to the ACE Service Daemon. See XXX for a description of the connection establishment mechanism.

Client Command Interface: On the client side, establishment of the connection has created a 'client command interface' object instance that is a command interface to the ACE Service Daemon. The Client Command Interface provides to mechanism of communications between the

client and daemon:

- Client to Daemon operation commands and Daemon to Client operation results
- Daemon to Client notification commands and Client to Daemon notification results.

The Client Command Interface uses the Command Semantics object to convert from client function/method calls at the interface into the Operation Network Commands. Results from the Operation Network Commands are converted into function/method return values utilizing the same Command Semantics object.

For Notification Network Commands received from the daemon the Client Command Interface uses the Command Semantics objects to convert the network command arguments into method/function arguments that are given to client methods/functions. The return values from the client methods/functions are converted to network command results and sent back to the daemon.

Operation Network Commands and Notification Network Commands may occur asynchronously.

ACE Service Daemon Data Thread: Is a component of an ACE Service Daemon. The Data Thread implements the unique behavior of each type of daemon. Each daemon also has a Command Semantics object that sets up the command semantics supported by the particular type of daemon. The remainder of the ACE Service Daemon is infrastructure that is common to all ACE Daemons.

Figure 1 only shows those portions of the ACE Service Daemon infrastructure that interact directly with the daemon elements designs in this document.

The Command Semantics object information is used by a Command Thread to parse incoming Operations Commands to check for semantics errors. Commands that are semantically correct are passed to the Control Thread. The Control Thread performs a command table lookup and calls the appropriate Data Thread object method with arguments from the network command to execute the command.

The Data Thread object methods invoked by the Control Thread execute commands and return the command results to the Control Thread. Command result semantics are obtained by the Data Thread from the Command Semantics Object and are used to construct the result returned to the Control Thread.

Data Thread object command methods may control hardware connected to the daemon's host through serial communications or operating system calls.

Some daemons provide information storage and/or query services. These typically utilize an SQL database.

If the Data Thread has data stream inputs and/or outputs the Data Thread '`run`' method continuously processes data to/from its data stream(s) in its own thread. The '`run`' method uses the Connection Interface to send/receive data on data stream sockets. Hardware connected to the daemon's host may be a source or sink for data streams.

2.1.1 Class Diagram

Figure 2 shows the ACE Service Daemon Data Thread class diagram. The diagram illustrates the relationships between the DataThread class and classes it interfaces with in the daemon.

The class diagram is referenced in other parts of this design document.

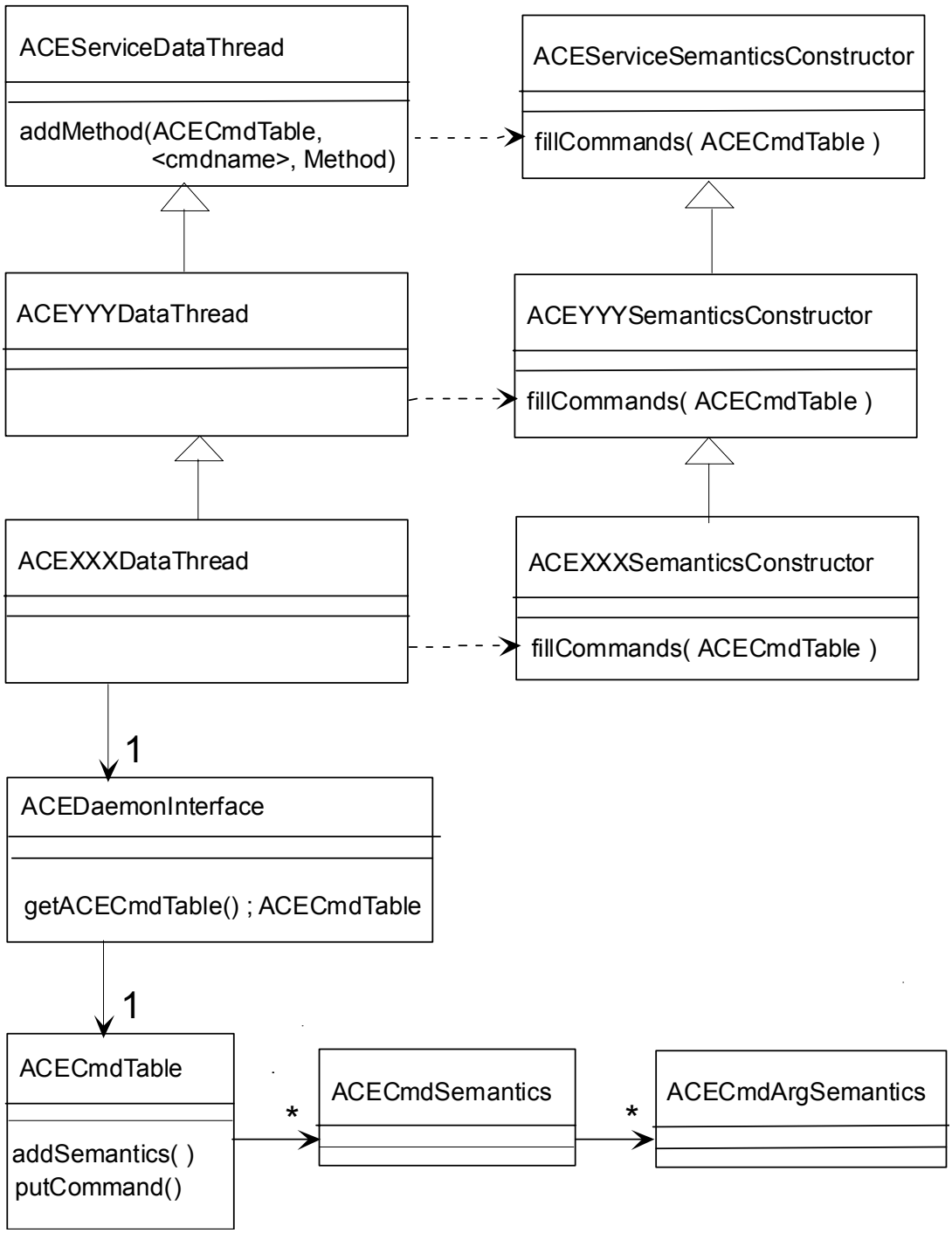


Figure 2 Data Thread Class Diagram

2.2 Command Semantics

A Command Semantics object sets up the network command semantics in a command table. The daemon uses the command semantics to parse incoming operations commands and notification

results and to construct out going operations results and notification commands. A client may also use the Command Semantics object to set up a command table for the same commands since a client and a daemon that communicate both use the same command semantics.

Language: The design for the Command Semantics is for the Java programming language.

As shown in figure 2, for each hierarchy level of the Data Thread object in a daemon there is a corresponding Command Semantic object level. All Command Semantics objects are derived from the '`ACEServiceSemanticsConstructor`' Java class. Each Command Semantic level adds the command semantics to the command table that are common to all services that inherit the level. There is an Interface Specification Document for each level describing the command semantics for the level. See the *ACE Service Interface Specification* document for the command semantics that are common to all ACE Service Daemons.

Example:

- ACEServiceSemanticsConstructor –adds commands common to all ACE services (shutdown, reset, etc).
- ACEDeviceSemanticsConstructor – adds commands common to all devices (setPower, etc).
- ACEPTZCameraSemanticsConstructor – adds commands common to all Pan/Tilt/Zoom cameras (setPosition, zoom, etc).
- ACEVCCEPTZCameraSemanticsConstructor – adds commands that only the Cannon VCC3 has.

2.2.1 ACEXXXSemanticsConstructor Class

The ACEXXXSemanticsConstructor class where XXX is replace by the name of the 'level' of the service (Examples, 'Service', 'Database', 'Device', 'PTZCamera', 'Projector', 'Security', 'UserID', 'iButton'). The semantics constructor is responsible for filling a command table will all of the command semantics that are use at the 'XXX' service level of the damon/client.

The following is an example Java Command Semantics class declaration:

```
public class ACEXXXSemanticsConstructor extends ACEYYYSemanticsConstructor {
    public static void
      fillCommands( ACECmdTable cmdTable );
    private static ACECmdSemantics
      mkAAACmdSemantics(); // Example of a method to create a ACECmdSemantics
}
```

Each `ACEXXXSemanticsConstructor` extends a higher level `ACEYYYSemanticsConstructor`. The highest level is the `ACEServiceSemanticsConstructor`.

2.2.1.1 Constructor **Method**

There is no constructor for this class. This class is never meant to be instantiated. All methods of

this class are thus static methods.

The Command Semantics classes were not merged into the Data Thread classes so that an ACE Client could use the same Command Semantics class to fill in its command table. Since a command sender and a command receiver both require knowledge of the command semantics both have to have the same semantics information. If there are any changes to the Command Semantics in the future then only one class has to be changed (the Command Semantics class) instead of two or more classes (a class in the client and a class in the daemon).

2.2.1.2 fillCommands **Method**

This method is called by the corresponding `ACEXXXDataThread` class hierarchy level in the `ACEXXXDataThread` constructor method.

```
class ACEXXXSemanticsConstructor
public static void
      fillCommands( ACECmdTable cmdTable );
```

Notice that his is a static method so that no instance of the `ACEXXXSemanticsConstructore` is required.

This method is responsible for filling the `ACECmdTable` argument with the `ACECmdSemantics` class instances that represent the command semantics for the `ACEXXXDataThread` level of the ACE Service hierarchy.

Design:

- For each network command supported by the 'current' service XXX hierarchy level (including operations commands, operations results commands, notification commands and notification results commands):
    - Call a private method `mkAAACmdSemantics` that returns an `ACECmdSemantics` class instance. See the **mkAAACmdSemantics** subsection for more information.
    - Use the inherited method ACEServiceSemanticsConstrutor.`addSemantics(…)` to add the `ACECmdSemantics` to the `ACECmdTable`.

Example: The ACE Projector service hierarchy level supports 4 commands:

1. GetVideoInputSource
2. GetVideoInputSourceResult
3. SetVideoInputSource
4. SetVideoInputSourceResult

The `ACEProjectorSemanticsConstructor` class `fillCommand()` adds `ACECmdSemantics` instances to the `ACECmdTable` argument for each command of these 4 commands.

2.2.1.3 mkAAACmdSemantics **Methods**

There is a method for each network command supported a service hierarchy level where 'AAA' is replace by some reasonable name representing the network command name. mkGetVidInSrcCmdSemantics is a good method name for the 'GetVideoInputSource' network command.

A Network Command is an ASCII representation of a command. It is used to transmit commands over network sockets between ACE Clients and ACE Service Daemons. The handling of Network Commands are not a concern of this document since they are handled by the ACE Service Daemon infrastructure. The Client Command Interface has a closer interface with the Network Commands and more information can be found in the Client Command Interface section of this document.

Network Commands specifications, found in ACE Service Daemon Interface Specification documents, are used in this design to create specific `ACECmdSemantic` class instances by each ACE Service Daemon's Command Semantics Constructor class. The Network Command specification defines command name, arguments (name, type, extent and dependencies on other arguments). These are needed by the mkAAACmdSemantics methods to create the `ACECmdSemantic` instances.

```
private static ACECmdSemantics
   mkAAACmdSemantics(); // Example of a method to create a ACECmdSemantics
```

Note that these are private methods. There must be NO access to these methods outside the class.

Design:

The following is a general sequence of events to follow in coding these methods:

- For each command argument create a new instance of an `ACECmdArgSemantics` class using the 'new' operator. See XXX for the API to the constructor. Use the appropriate interface specifications document to find the network command information to setup the type, extent, dependencies, etc. arguments to the construtor.
- Don't add the 'status', 'cmdErrorNo', or 'msg' arguments. They are handled later.
- Assemble the argument semantics class instances into a vector for the next step.
- Use the 'new' operator to create a new instance of the `ACECmdSemantics` class.
- Call the `'addCommonResultCmdArgSemantics()'` method to add the 'status', 'cmdErrorNo', and 'msg' arguments to the semantics instance.

2.3 Service Daemon Data Thread

This subsection describes the design for the ACE Service Daemon Data Thread object. The Data Thread object is one of several threads that constitute an ACE Service Daemon but it and the

Command Semantics are the only objects that are different for each daemon. The Data Thread is responsible for implementing the specific behavior of each ACE Service Daemon. It executes the commands received from a client and may generate notifications sent to a client.

The Data Thread object is composed of an inherited hierarchy of Data Thread objects that follows a path through the ACE Command Object Hierarchy. See Figure 2 'Class Diagram' to view the class hierarchy mechanism and where it fits into the daemon infrastructure.

There is only one data thread instance per daemon.

Language: This design subsection assumes an object oriented language is used to implement the design (example: Java, Lisp, C++). In particular the Java language is addressed.

There are two threads of execution that use the Data Thread class. One is the Control thread and the other is the Data thread. Note that there is a distinction between the Data Thread class and the Data thread (big 'T' class and little 't' thread). This distinction exists because Data Thread class methods are executed by the daemon's Control thread.

2.3.1 ACEXXXDataThread class

This class is has two responsibilities:

- Implement the command methods that the Control thread executes for each command
- Implement the 'run' method for the Data thread.

```
public [abstract] class ACEXXXDataThread extends ACEYYYDataThread {
  public ACEXXXDataThread (…);
  public void run(…); // only for leaf class
  public XXXCmd(…); // only for leaf class
}
```

By convention the name of the data thread class always ends in 'DataThread'.

The class is declared '`abstract`' if it is not a leaf class. A leaf class is a class that is not extended by any other class. By declaring the class '`abstract`' there will be no instances of the class. Only leaf classes of the Data Thread are instantiated.

The most 'super' class for ACE Data Thread classes is the '`ACESerivceDataThread`'. This abstract class must be inherited by all Data Thread classes either through direct or indirect inheritance.

The command methods are methods of the Data Thread class that are associated with specific Network Commands. When a Network Command is received by the daemon the Control Thread in the daemon performs a table lookup on the command to determine which Data Thread method to invoke. It calls the method with an `ACECmdLine` class instance that represents the contents of the Network Command. Note: command methods are associated with command semantics in the constructor of the Data Thread class.

The ‘run’ method of the Data Thread class performs the tasks of the Data thread. It performs any service initialization and data stream processing service if required.

2.3.1.1 Constructor method

The constructor method of a Data Thread class performs class specific initialization. It does not do initialization of hardware, databases, etc. Initialization of the hardware, databases, etc. is left to the ‘`run()`’ method of the class.

```
public ACEXXXDataThread (java.lang.ThreadGroup threadGroup,
                         java.lang.String threadName,
                         ACEDaemonInterface mainThread)
```

Design:

- Call the super() method on the 3 arguments passed into the constructor.
- Get the '`ACECmdTable`' created by using the daemon infrastructure method '`ACEDaemonInterface.getACECmdTable()`'.
- Fill the `ACECmdTable` for the class's level of the service using the '`ACEXXXSemantics.fillCommands`' method where XXX is the semantics level that corresponds to the Data Thread class's level. Not that each super class constructor calls it’s own fillCommands method.
- For each command implemented by this class, use the inherited '`addMethod`' method to associate a command with the method to invoke to execute the command. Note: Use the inherited '`getMethod`' method to obtain a method reference using the method's name. There does not have to be a one-to-one correspondence between the Network Commands and the Data Thread method implementing the command. A single Data Thread method is allowed to implement more than one command.
- Abort the program if there is a problem finding an expected command or method since it is a coding error and not recoverable.

2.3.1.2 Run method

This method, implemented only by leaf Data Thread classes, carries out initialization not performed by the constructor method. For those daemons that perform data stream processing this method also handles that function.

Any initialization required by the Data Thread commands must be performed in the `run()` method. This might include opening a serial port to a device controlled by the daemon over a serial interface.

There are a number of Service Daemons in ACE that generate, consume, or transform streams of data. The data streams in the ACE environment include, but are not limited to, video and audio. Data streams are communicated between ACE daemons and clients through network sockets.

Processing of these streams is generally CPU intensive. Since ACE Service Daemons are implemented in Java, and C is much more efficient than Java for processing high data rate data streams, the run() method should use the JNI to call a native C function to processes data streams. The C function continually loops gathering data, from a socket or local device, processing the data and sending the data, to a socket or local device. On a regular basis the function must check a memory location to determine if a command executed by the Control Thread has changed the value in the memory location to signify that the function must change its state/behavior.

```
public void run();
```

Design:

Information needed to perform the initialization is obtained from the Control Thread.

- Get the control thread using the inherited 'getControlThread()' method.
- Get configuration data from the control thread using its 'ACEControlThreadInterface.getConfig()' method. Example: The string ‘port’ as an argument returns the name of the serial device to initialize.
- Perform any initialization required before any commands are processed. Example: initialize a hardware device to a known state, initialize a serial port for control of a hardware device.
- If implementing a data stream daemon enter an endless loop.
    - Wait for the command to process the stream. This usually involves using the Java `wait()` method. A command executed by the Control thread calls `notify()` to wake up the Data thread.
    - Process the stream until a ‘stop’ command is issued. The ‘stop’ method should wait until the run method has finished stream processing. This prevents additional commands from coming in while the ‘run’ method is between states.
    - Return to the top of the loop.

2.3.1.3 General Command Method Design

This subsection describes in general what command execution method designs should be like. This includes both 'set' and 'get' command methods.

These methods are normally only implemented in leaf classes. Any methods that are declared in non leaf classes must be 'abstract' unless the implementation is known to be common for all subclasses, then the method may be implemented in an intermediate class. There are a number of command methods that are implemented at the ACEServiceDataThread class.

Arguments:

ACECmdLine - the command line received from the client.

Design:

- Get the semantics object for the result command using the '`getSemantics`' method.
- Create a new result command line using the '`ACECmdLine`' constructor.
- Obtain any required argument information from the `ACECmdLine` argument.
- Perform the processing for the specific command. If there is an underlying C API then call the Java wrapper method(s).
- Fill in the `ACECmdLine` with return values if any using the `ACECmdLine.addArg` method.
    - For a 'Set' command always add the 'status' argument. To give the client good information about why a command failed the 'errorNo' and 'msg' arguments should be added also. Add as much useful information as possible to the 'msg' argument.
    - Any error that occurs while adding an argument to the command line must abort the program. An error here is a coding error and thus fatal.
- Return the ACECmdLine.

2.3.1.4 Service Required Methods

These abstract methods must be implemented by every ACE Service Daemon data thread leaf class. See the ACE Service Interface Specification for more information on these methods.

- Reset - Reset the state of the daemon to an initialized state.

2.3.1.5 Device Required Methods

These abstract methods must be implemented by every ACE device service daemon thread leaf class. See the ACE Device Interface Specification for more information on these methods.

- DeviceReset
- GetPowerState
- SetPoweState

2.3.1.6 Other ACE Service Subclass Required Methods

The class must also implement other methods that are inherited from ACE Service decendent classes. Example: The 'Epson3750DataThread' class must implement 'ProjectorDataThread' class abstract methods.

2.3.2 Java C Library Wrapper

If the data thread requires the use of a C library a Java wrapper is required to use the C library. By convention a separate class is created specifically for the wrapper. The name of the wrapper describes the C library and ends in 'JNI'. There is no requirement to have a one-to-one match between the JNI methods and the C library entry functions. The JNI methods may perform extra processing (example: argument unit conversions).

2.4 Client Command Interface

To aid ACE Clients in communicating with ACE Daemons a hierarchy of command interface classes are designed. The interface class has methods that the clients may execute to send commands to an ACE Daemon. The client creates an instance of a command interfaces class for each daemon instance it wishes to command. Note that the command interface to the ASD aids in creating instances of command interface classes.

Language: All current command interface classes are implemented in the Java programming language.

The command interface class handles converting Java method arguments into ACE network command arguments and sending the commands to the daemon. It also converts the daemon command result into a return value or throws a Java exception.

2.4.1 ACEXXX Client Command Interface class

The command interfaces classes follow the same class hierarchy as the daemon Data Thread and Semantics Constructor classes. The most 'super' command interface class is the `ACEService` class.

By convention the name of the command interface class is the same as the data thread class but with the 'DataThread' omitted.

```
Public [abstract] ACEXXX extends ACEYYY {
   ACEXXX( String serviceAddress);
}
```

2.4.1.1 Constructor Member Function

This is the constructor method for the XXX Service.

Argument:

ACEAddress - This is an object (String) that contains address information about the specific service daemon to connect to.

The constructor function does the following:

- Call the super class constructor with the ACEAddress argument..

- Get the 'ACECmdTable' using the inherited member function 'getCommandTable'.
- Call the 'fillCommands' member function of the 'ACEXXXSemantics' class.

2.4.1.2 GetXXX Command Member Functions

Implement a 'Get' member function for each command at this object's ACE Service Hierarchy level that queries information from an ACE Service Daemon. Note that this is different than in the Data Thread where most command methods are declared at their appropriate class hierarchy level and only implemented at the leaf class level.

Example: The ACE Service Daemon ‘getPowerState’ command is implemented at the ACEDevice command interface level.

Note that it is not required to have a one-to-one correspondence between network commands and query methods. It may be reasonable to have several query methods that use the same network command. It is also reasonable to have a single query use more than one network command.

By convention all query methods begin with the letters ‘get’.

Design:

The member function performs the following:

- Get the command's semantics object from the object using the inherited 'getACECmdSemantics' method.
- If the semantics is not found print out a meaningful debug message and abort the program. We have found a programming error.
- Use the 'ACECmdLine' object constructor with the above semantics as an argument to create a command line object.
- For each query argument do the following:
    - Use the 'addArg' method of the 'ACECmdLine' object to add the argument and its value to the command.
    - If there is an error trying to add the argument to the command line then print out a meaningful debug message to standard error and abort the program. We have found a programming error.
- Use the ACEService.sendMessage' method with the 'ACECmdLine' object as and argument to send the command to the daemon.
- From the returned ACECmdLine command line get each response argument value using the 'ACECmdLine.getArgValue' method.

- Return the values from the query method.

2.4.1.3 SetXXX Command Member Functions

Implement a 'set' member function for each command at this object's ACE Service Hierarchy level that changes state in an ACE Service Daemon. Note that this is different than in the Data Thread where most command methods are declared at their appropriate class hierarchy level and only implemented at the leaf class level.

Example: The ACE Service Daemon ‘reset’ command is implemented in the ACEService command interface class. The ACE Service Daemon ‘setPowerState’ command is implemented at the ACEDevice command interface level.

Note that it is not required to have a one-to-one correspondence between network commands and query methods. It may be reasonable to have several query methods that use the same network command. It is also reasonable to have a single query use more than one network command.

By convention all state change methods begin with the letters ‘set’.

Design:

The member function does the following:

- Get the command's semantics object from the object using the inherited 'getACECmdSemantics' method.
- If the semantics is not found print out a meaningful debug message and abort the program. We have found a programming error.
- Use the 'ACECmdLine' object constructor with the above semantics as an argument to create a command line object.
- For each query argument do the following:
    - Use the 'addArg' method of the 'ACECmdLine' object to add the argument and its value to the command.
    - If there is an error trying to add the argument to the command line then print out a meaningful debug message to standard error and abort the program. We have found a programming error.
- Use the service objects 'sendMessage' method with the 'ACECmdLine' object as and argument to send the command to the daemon.

2.4.2 Client Notification

3 Glossary

<Refer to a project glossary for terms used throughout the project>

(TBS)

4 Change Log

Version	Date	Changes
0.1	Jan, 12 2001	Initial Working Version

5 Notes

-