TANGO Device Server Programmer's Manual

Version 1.0

E. Taurel

April 4, 2001

Contents

Are you ready to dance the TANGO ?

Chapter 1

Introduction

1.1 Introduction to device server

Device servers were first developed at the European Synchrotron radiation Facility (ESRF) for controlling the 6 Gev synchrotron radiation source. This document is a Programmer's Manual on how to write TANGO device servers. It will not go into the details of the ESRF, nor its Control System nor any of the specific device servers in the Control System. The role of this document is to help programmers faced with the task of writing TANGO device servers.

Device servers have been developed at the ESRF in order to solve the main task of Control Systems viz provide read and write access to all devices in a distributed system. The problem of distributed device access is only part of the problem however. The other part of the problem is providing a programming framework for a large number of devices programmed by a large number of programmers each having different levels of experience and style.

Device servers have been written at the ESRF for a large variety of different devices. Devices vary from serial line devices to devices interfaced by field-bus to memory mapped VME cards or PC cards to entire data acquisition systems. The definition of a device depends very much on the user's requirements. In the simple case a device server can be used to hide the serial line protocol required to communicate with a device. For more complicated devices the device server can be used to hide the entire complexity of the device timing, configuration and acquisition cycle behind a set of high level commands.

In this manual the process of how to write TANGO device servers will be treated. The manual has been organised as follows :

- A getting started chapter.
- The TANGO device server model is treated in chapter 3
- The TANGO device server pattern and framework are described in chapter 4
- Chapter 5 describes how to client and a device server exchange data
- How to write a device server is explained in chapter 6
- How to compile, link and execute a Tango device server is detailed in chapter 7
- Advanced programming techniques for device server are described in chapter 8

Throughout this manual examples of source code will be given in order to illustrate what is meant. The examples have been taken from the StepperMotor class - a simulation of a stepper motor which illustrates how a typical device server for a stepper motor at the ESRF functions. The simulation runs under HP-UX, Solaris, Linux and Windows-NT and requires no hardware in order to run.

1.2 Device server history

The concept of using device servers to access devices was first proposed at the ESRF in 1989. It has been successfully used as the heart of the ESRF Control System for the institute accelerator complex. This Control System has been named TACO[1]. Then, it has been decided to also used TACO to control devices in the beam-lines. Today, more than 30 instances of TACO are running at the ESRF. The main technologies used within TACO are the leading technologies of the 80's. The Sun Remote Procedure Call (RPC) is used to communicate over the network between device server and applications, OS-9 is used on the front-end computers, C is the reference language to write device servers and clients and the device server framework follows the MIT Widget model. In 1999, a renewal of the control system was started. The new version of the ESRF control system is named TANGO[2] and is based on the 21 century technologies :

- CORBA[3] to communicate between device server and clients
- C++ and Java as reference programming languages
- Linux, Solaris, HP-UX and Windows-NT as operating systems
- Modern object oriented design pattern

[1] TACO stands for **T**elescope and **A**ccelerator **C**ontrolled with **O**bjects
[2] TANGO stands for **TA**co **N**ext **G**eneration **O**bject
[3] CORBA stands for **C**ommon **O**bject **R**equest **B**roker **A**rchitecture

Chapter 2

Getting Started

The code given in this chapter as example has been generated using POGO. Pogo is a code generator for Tango device server. See [13] for more information about POGO. The following examples briefly describe how to write device class with commands which receives and return different kind of Tango data types and also how to write device attributes The device class implements 5 commands and 3 attributes. The commands are :

- The command **DevSimple** deals with simple Tango data type
- The command **DevString** deals with Tango strings
- **DevArray** receive and return an array of simple Tango data type
- **DevStrArray** which does not receive any data but which returns an array of strings
- **DevStruct** which also does not receive data but which returns one of the two Tango composed types (DevVarDoubleStringArray)

For all these commands, the default behaviour of the state machine (command allways allowed) is acceptable. The attributes are :

- A spectrum type attribute of the Tango string type called **StrAttr**
- A readable attribute of the Tango::DevLong type called **LongRdAttr**. This attribute is linked with the following writable attribute
- A writable attribute also of the Tango::DevLong type called **LongWrAttr**.

2.1 The commands and attributes code in C++

For each command called DevXxxx, pogo generates in the device class a method named dev_xxx which will be executed when the command is requested by a client. In this chapter, the name of the device class is *DocDs*

2.1.1 The DevSimple command

This method receives a Tango::DevFloat type and also returns a data of the Tango::DevFloat type which is simply the double of the input value. The code for the method executed by this command is the following:

```
 1  Tango::DevFloat DocDs::dev_simple(Tango::DevFloat argin)
 2  {
 3          Tango::DevFloat argout ;
 4          cout1 << "DocDs::dev_simple(): entering... !" << endl;
 5
 6          //      Add your own code to control device here
 7
 8          argout = argin * 2;
 9          return argout;
10  }
```

This method is fairly simple. The received data is passed to the method as its argument. It is doubled at line 8 and the method simply returns the result.

2.1.2 The DevArray command

This method receives a data of the Tango::DevVarLongArray type and also returns a data of the Tango::DevVarLongArray type. Each element of the array is doubled. The code for the method executed by the command is the following :

```
 1  Tango::DevVarLongArray *DocDs::dev_array(const Tango::DevVarLongArray *argin)
 2  {
 3          //      POGO has generated a method core with argout allocation.
 4          //      If you would like to use a static reference without copying,
 5          //      See "TANGO Device Server Programmer's Manual"
 6          //              (chapter x.x)
 7          //-------------------------------------------------------------
 8          Tango::DevVarLongArray  *argout  = new Tango::DevVarLongArray();
 9
10          cout1 << "DocDs::dev_array(): entering... !" << endl;
11
12          //      Add your own code to control device here
13
14          long argin_length = argin->length();
15          argout->length(argin_length);
16          for (int i = 0;i < argin_length;i++)
17                  (*argout)[i] = (*argin)[i] * 2;
18
19          return argout;
20  }
```

The argout data array is created at line 8. Its length is set at line 15 from the input argument length. The array is populated at line 16,17 and returned. This method allocates memory for the argout array. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). It is also possible to return data from a statically allocated array without copying. Look at chapter 5 for all the details.

2.1.3 The DevString command

This method receives a data of the Tango::DevString type and also returns a data of the Tango::DevString type. The command simply displays the content of the input string and returns a hard-coded string. The code for the method executed by the command is the following :

```
1  Tango::DevString DocDs::dev_string(Tango::DevString argin)
2  {
3          //      POGO has generated a method core with argout allocation.
4          //      If you would like to use a static reference without copying,
5          //      See "TANGO Device Server Programmer's Manual"
6          //              (chapter x.x)
7          //--------------------------------------------------------
8          Tango::DevString        argout;
9          cout1 << "DocDs::dev_string(): entering... !" << endl;
10
11         //      Add your own code to control device here
12
13         cout << "the received string is " << argin << endl;
14
15         string str("Am I a good Tango dancer ?");
16         argout = new char[str.size() + 1];
17         strcpy(argout,str.c_str());
18
19         return argout;
20 }
```

The argout string is created at line 8. Internally, this method is using a standard C++ string. Memory for the returned data is allocated at line 16 and is initialized at line 17. This method allocates memory for the argout string. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). It is also possible to return data from a statically allocated string without copying. Look at chapter 5 for all the details.

2.1.4 The DevStrArray command

This method does not receive input data but returns an array of strings (Tango::DevVarStringArray type). The code for the method executed by this command is the following:

```
1  Tango::DevVarStringArray *DocDs::dev_str_array()
2  {
3          //      POGO has generated a method core with argout allocation.
4          //      If you would like to use a static reference without copying,
5          //      See "TANGO Device Server Programmer's Manual"
6          //              (chapter x.x)
7          //--------------------------------------------------------
8          Tango::DevVarStringArray        *argout  = new Tango::DevVarStringArray();
9
10         cout1 << "DocDs::dev_str_array(): entering... !" << endl;
11
12         //      Add your own code to control device here
```

```
13
14          argout->length(3);
15          (*argout)[0] = CORBA::string_dup("Rumba");
16          (*argout)[1] = CORBA::string_dup("Waltz");
17          string str("Jerck");
18          (*argout)[2] = CORBA::string_dup(str.c_str());
19          return argout;
20  }
```

The argout data array is created at line 8. Its length is set at line 14. The array is populated at line 15,16 and 18. The last array element is intialized from a standard C++ string created at line 17. Note the usage of the *string_ dup* function of the CORBA namespace. This is necessary for strings array due to the CORBA memory allocation schema.

2.1.5 The DevStruct command

This method does not receive input data but returns a structure of the Tango::DevVarDoubleStringArray type. This type is a composed type with an array of double and an array of strings. The code for the method executed by this command is the following:

```
 1  Tango::DevVarDoubleStringArray *DocDs::dev_struct()
 2  {
 3          //      POGO has generated a method core with argout allocation.
 4          //      If you would like to use a static reference without copying,
 5          //      See "TANGO Device Server Programmer's Manual"
 6          //              (chapter x.x)
 7          //--------------------------------------------------------------
 8          Tango::DevVarDoubleStringArray  *argout  = new Tango::DevVarDoubleStringArra
 9
10          cout1 << "DocDs::dev_struct(): entering... !" << endl;
11
12          //      Add your own code to control device here
13
14          argout->dvalue.length(3);
15          argout->dvalue[0] = 0.0;
16          argout->dvalue[1] = 11.11;
17          argout->dvalue[2] = 22.22;
18
19          argout->svalue.length(2);
20          argout->svalue[0] = CORBA::string_dup("Be Bop");
21          string str("Smurf");
22          argout->svalue[1] = CORBA::string_dup(str.c_str());
23
24          return argout;
25  }
```

The argout data structure is created at line 8. The length of the double array in the output structure is set at line 14. The array is populated between lines 15 and 17. The length of the string array in the output structure is set at line 19. This string array is populated between lines 20 an

22 from a hard-coded string and from a standard C++ string. This method allocates memory for the argout data. This memory is freed by the Tango core classes after the data have been sent to the caller (no delete is needed). Note the usage of the *string_ dup* function of the CORBA namespace. This is necessary for strings array due to the CORBA memory allocation schema.

2.1.6 The three attributes

Some data have been added to the definition of the device class in order to store attributes value. These data are (part of the class definition) :

```
1
2
3  protected :
4          //      Add your own data members here
5          //-----------------------------------------
6          Tango::DevString        attr_str_array[5];
7          Tango::DevLong          attr_rd;
8          Tango::DevLong          attr_wr;
```

One data has been created for each attribute. As the StrAttr attribute is of type spectrum with a maximun X dimension of 5, an array of length 5 has been reserved.

Three methods are necessary to implement these attributes. The code for these methods is the following :

```
 1 void DocDs::write_attr_hardware(vector<long> &attr_list)
 2 {
 3          cout << "In write_attr_hardware for " << attr_list.size();
 4          cout << " attribute(s)" << endl;
 5
 6          for (long i=0 ; i < attr_list.size() ; i++)
 7          {
 8                  WAttribute &att = dev_attr->get_w_attr_by_ind(attr_list[i]);
 9                  string attr_name = att.get_name();
10
11                  cout << "Attribute name = " << attr_name;
12
13                  //      Switch on attribute name
14                  //---------------------------------
15                  if (attr_name == "LongWrAttr")
16                  {
17                          //      Add your own code here
18                          att.get_write_value(attr_wr);
19                          cout << "Value to be written = " << attr_wr << endl;
20                  }
21          }
22 }
23
24 void DocDs::read_attr_hardware(vector<long> &attr_list)
25 {
```

```
26          cout << "In read_attr_hardware for " << attr_list.size();
27          cout << " attribute(s)" << endl;
28
29          //      Add your own code here
30          //--------------------------------
31
32          string att_name;
33          for (long i = 0;i < attr_list.size();i++)
34          {
35                  att_name = dev_attr->get_attr_by_ind(attr_list[i]).get_name();
36
37                  if (att_name == "LongRdAttr")
38                  {
39                          attr_rd = 5;
40                  }
41          }
42  }
43
44  void DocDs::read_attr(Tango::Attribute &attr)
45  {
46          string &attr_name = attr.get_name();
47
48          cout << "In read_attr for attribute " << attr_name << endl;
49
50          //      Switch on attribute name
51          //--------------------------------
52          if (attr_name == "LongRdAttr")
53          {
54                  //      Add your own code here
55                  attr.set_value(&attr_rd);
56          }
57          if (attr_name == "LongWrAttr")
58          {
59                  //      Add your own code here
60                  attr.set_value(&attr_wr);
61          }
62          if (attr_name == "StrAttr")
63          {
64                  //      Add your own code here
65                  attr_str_array[0] = CORBA::string_dup("Rock");
66                  attr_str_array[1] = CORBA::string_dup("Samba");
67
68                  attr.set_value(attr_str_array, 2);
69          }
70  }
```

The *write_attr_hardware()* method is executed when an attribute value is set by a client. In our example only one attribute is writable (the LongWrAttr attribute). The new attribute value coming from the client is stored in the object data at line 18. The *read_attr_hardware()* method is executed once when a client execute the read_attributes CORBA request. The rule of this method is to read the hardware and to store the read values somewhere in the device object. In our example, only the LongRdAttr attribute internal value is set by this method at line 39. The

method *read_ attr()* is executed for each attribute to be read by the read_attributes CORBA call. Its rule is to set the attribute value in the TANGO core classes object representing the attribute. This is done at line 55 for the LongRdAttr attribute, at line 60 for the LongWrAttr attribute and at line 68 for the StrAttr attribute. This last attribute is initialised in this method at line 65 and 66 with the *string_ dup* CORBA function.

2.2 The commands and attributes code in java

For each command called DevXxxx, pogo generates in the device class a method named dev_xxx which will be executed when the command is requested by a client. In this chapter, the name of the device class is *DocDs*

2.2.1 The DevSimple command

This method receives a Tango DevFloat type and also returns a data of the Tango DevFloat type which is simply the double of the input value. Using java, the Tango::DevFloat type is mapped to classical java float type. The code for the method executed by this command is the following:

```
1  public float dev_simple(float argin) throws DevFailed
2  {
3          float   argout =  (float)0;
4
5          Util.out2.println("Entering dev_simple()");
6
7          // ---Add your Own code to control device here ---
8
9          argout = argin * 2;
10         return argout;
11 }
```

This method is fairly simple. The received data is passed to the method as its argument. It is doubled at line 9 and the method simply returns the result.

2.2.2 The DevArray command

This method receives a data of the Tango::DevVarLongArray type and also returns a data of the Tango::DevVarLongArray type. Each element of the array is doubled. Using java, the Tango DevVarLongArray type is mapped to an array of java long. The code for the method executed by the command is the following :

```
1  public int[] dev_array(int[] argin) throws DevFailed
2  {
3          int[]   argout = new int[argin.length];
4
5          Util.out2.println("Entering dev_array()");
6
7          // ---Add your Own code to control device here ---
8
9          for (int i = 0;i < argin.length;i++)
```

```
10                  argout[i] = argin[i] * 2;
11          return argout;
12  }
```

The argout data array is created at line 3. The array is populated at line 9,10 and returned.

2.2.3 The DevString command

This method receives a data of the Tango DevString type and also returns a data of the Tango DevString type. The command simply displays the content of the input string and returns a hard-coded string. Using java, the Tango DevString type simply maps to java String.The code for the method executed by the command is the following :

```
1  public String dev_string(String argin) throws DevFailed
2  {
3          Util.out2.println("Entering dev_string()");
4
5          // ---Add your Own code to control device here ---
6
7          System.out.println("the received string is "+argin);
8
9          String argout = new String("Am I a good Tango dancer ?");
10         return argout;
11 }
```

The argout string is created at line 9.

2.2.4 The DevStrArray command

This method does not receive input data but returns an array of strings (Tango DevVarStringArray type). Using java, the Tango DevVarStringArray type maps to an array of java String. The code for the method executed by this command is the following:

```
1  public String[] dev_str_array() throws DevFailed
2  {
3
4          Util.out2.println("Entering dev_str_array()");
5
6          // ---Add your Own code to control device here ---
7
8          String[] argout = new String[3];
9          argout[0] = new String("Rumba");
10         argout[1] = new String("Waltz");
11         argout[2] = new String("Jerck");
12         return argout;
13 }
```

The argout data array is created at line 8. The array is populated at line 9,10 and 11.

2.2.5 The DevStruct command

This method does not receive input data but returns a structure of the Tango DevVarDoubleStringArray type. This type is a composed type with an array of double and an array of strings. This is mapped to a specific java class called DevVarDoubleStringArray. The code for the method executed by this command is the following:

```
1  public DevVarDoubleStringArray dev_struct() throws DevFailed
2  {
3          DevVarDoubleStringArray argout = new DevVarDoubleStringArray();
4
5          Util.out2.println("Entering dev_struct()");
6
7          // ---Add your Own code to control device here ---
8
9          argout.dvalue = new double[3];
10         argout.dvalue[0] = 0.0;
11         argout.dvalue[1] = 11.11;
12         argout.dvalue[2] = 22.22;
13
14         argout.svalue = new String[2];
15         argout.svalue[0] = new String("Be Bop");
16         argout.svalue[1] = new String("Smurf");
17
18         return argout;
19 }
```

The argout data structure is created at line 3. The double array in the output structure is created at line 9. The array is populated between lines 10 and 12. The string array in the output structure is created at line 14. This string array is populated between lines 15 and 16 from hard-coded strings.

2.2.6 The three attributes

Some data have been added to the definition of the device class in order to store attributes value. These data are (part of the class definition) :

```
1  protected String[]      attr_str_array = new String[5];
2  protected int           attr_rd;
3  protected int           attr_wr;
```

One data has been created for each attribute. As the StrAttr attribute is of type spectrum with a maximun X dimension of 5, an array of length 5 has been reserved.

Three methods are necessary to implement these attributes. The code for these methods is the following :

```
1  public void write_attr_hardware(Vector attr_list)
2  {
3          Util.out2.println("In write_attr_hardware for "+attr_list.size()+" attribute
4
5          for (int i=0 ; i<attr_list.size() ; i++)
6          {
7                  int ind = ((Integer)(attr_list.elementAt(i))).intValue();
8                  WAttribute att = dev_attr.get_w_attr_by_ind(ind);
9                  String attr_name = att.get_name();
10
11                 //      Switch on attribute name
12                 //--------------------------------
13                 if (attr_name.equals("LongWrAttr") == true)
14                 {
15                         //      Add your own code here
16                         attr_wr = att.get_lg_write_value();
17                         System.out.println("Value to be written = "+attr_wr);
18                 }
19         }
20 }
21
22
23 public void read_attr_hardware(Vector attr_list)
24 {
25         Util.out2.println("In read_attr_hardware for "+attr_list.size()+" attribute(
26
27         //      Add you own code here
28         //--------------------------------
29
30         for (int i=0; i<attr_list.size() ; i++)
31         {
32                 int ind = ((Integer)(attr_list.elementAt(i))).intValue();
33                 Attribute att = dev_attr.get_attr_by_ind(ind);
34                 String attr_name = attr_list.elementAt(i);
35
36                 if (attr_name.equals("LongRdAttr") == true)
37                 {
38                         attr_rd = 5;
39                 }
40                 else if (attr_name.equals("StrAttr") == true)
41                 {
42                         attr_str_array[0] = new String("Rock");
43                         attr_str_array[1] = new String("Samba");
44                 }
45         }
46 }
47
48
49 public void read_attr(Attribute attr) throws DevFailed
50 {
51         String attr_name = attr.get_name();
52         Util.out2.println("In read_attr for attribute "+attr_name);
53
54         //      Switch on attribute name
```

```
55          //---------------------------------
56          if (attr_name.equals("LongWrAttr") == true)
57          {
58                  //      Add your own code here
59                  attr.set_value(attr_wr);
60          }
61          if (attr_name.equals("LongRdAttr") == true)
62          {
63                  //      Add your own code here
64                  attr.set_value(attr_rd);
65          }
66          if (attr_name.equals("StrAttr") == true)
67          {
68                  //      Add your own code here
69                  attr.set_value(attr_str_array);
70          }
71  }
```

The *write_attr_hardware()* method is executed when an attribute value is set by a client. In our example only one attribute is writable (the LongWrAttr attribute). The new attribute value coming from the client is stored in the object data at line 16. The *read_attr_hardware()* method is executed once when a client execute the read_attributes CORBA request. The rule of this method is to read the hardware and to store the read values somewhere in the device object. In our example, the LongRdAttr attribute internal value is set by this method at line 38 at the StrAttr attribute internal value is set at lines 42 and 43. The method *read_attr()* is executed for each attribute to be read by the read_attributes CORBA call. Its rule is to set the attribute value in the TANGO core classes object representing the attribute. This is done at line 64 for the LongRdAttr attribute, at line 59 for the LongWrAttr attribute and at line 69 for the StrAttr attribute.

Chapter 3

The TANGO device server model

This chapter will present the TANGO device server object model hereafter referred as TDSOM. First, it will introduce CORBA. Then, it will describe each of the basic features of the TDSOM and their function. The TDSOM can be divided into the following basic elements - the *device*, the *server*, the *database* and the *application programmers interface*. This chapter will treat each of the above elements separately.

3.1 Introduction to CORBA

CORBA is a definition of how to write object request brokers (ORB). The definition is managed by the Object Management Group (OMG [1]). Various commercial and non-commercial implementations exist for CORBA for all the mainstream operating systems. CORBA uses a programming language independent definition language (called IDL) to defined network object interfaces. Language mappings are defined from IDL to the main programming languages e.g. C++, Java, C, COBOL, Smalltalk and ADA. Within an interface, CORBA defines two kinds of actions available to the outside world. These actions are called **attributes** and **operations**.

Operations are all the actions offered by an interface. For instance, within an interface for a Thermostat class, operations could be the action to read the temperature or to set the nominal temperature. An attribute defines a pair of operations a client can call to send or receive a value. For instance, the position of a motor can be defined as an attribute because it is a data that you only set or get. A read only attribute defines a single operation the client can call to receives a value. In case of error, an operation is able to throw an exception to the client, attributes cannot raises exception except system exception (du to network fault for instance).

Intuitively, IDL interface correspond to C++ classes and IDL operations correspond to C++ member functions and attributes as a way to read/write public member variable. Nevertheless, IDL defines only the interface to an object and say nothing about the object implementation. IDL is only a descriptive language. Once the interface is fully described in the IDL language, a compiler (from IDL to C++, from IDL to Java...) generates code to implement this interface. Obviously, you still have to write how operations are implemented.

The act of invoking an operation on an interface causes the ORB to send a message to the corresponding object implementation. If the target object is in another address space, the ORB run time sends a remote procedure call to the implementation. If the target object is in the same address space as the caller, the invocation is accomplished as an ordinary function call to avoid the overhead of using a networking protocol.

For an excellent reference on CORBA with C++ refer to [2]. The complete TANGO IDL file can be found in the TANGO web page[3]

3.2 The model

The basic idea of the TDSOM is to treat each device as an **object**. Each device is a separate entity which has its own data and behavior. Each device has a unique name which identifies it in network name space. Devices are organised according to **classes**, each device belonging to a class. All classes are derived from one root class thus allowing some common behavior for all devices. Four kind of requests can be sent to a device (locally i.e. in the same process, or remotely i.e. across the network) :

- Execute actions via **commands**
- Read some basic device data available for all devices via CORBA attributes.
- Read/Set data specific to each device belonging to a class via TANGO **attributes**
- Execute a predefined set of actions available for every devices via CORBA operations

Each device is stored in a process called a **device server**. Devices are configured at runtime via **properties** which are stored in a **database**.

3.3 The device

The device is the heart of the TDSOM. A device is an abstract concept defined by the TDSOM. In reality, it can be a piece of hardware (an interlock bit) a collection of hardware (a screen attached to a stepper motor) a logical device (a taper) or a combination of all these (an accelerator). Each device has a unique name in the control system and eventually alias. At the ESRF a four field name space has been adopted consisting of

[//FACILITY/]DOMAIN/CLASS/MEMBER

Facility refers to the control system instance, domain refers to the sub-system, class the class and member the instance of the device. Device name alias(es) must also be unique within a control system. There is no predefined syntax for device name alias.

Each device belongs to a class. The device class contains a complete description and implementation of the behavior of all members of that class. New device classes can be constructed out of existing device classes. This way a new hierarchy of classes can be built up in a short time. Device classes can use existing devices as sub-classes or as sub-objects. The practice of reusing existing classes is classical for Object Oriented Programming and is one of its main advantages.

All device classes are derived from the same class (the device root class) and implement **the same CORBA interface**. All devices implementing the same CORBA interface ensures all control object support the same set of CORBA operations and attributes. The device root class contains part of the common device code. By inheriting from this class, all devices shared a common behavior. This also makes maintenance and improvements to the TDSOM easy to carry out.

All devices also support a **black box** where client requests for attributes or operations are recorded. This feature allows easier debugging session for device already installed in a running control system.

3.3.1 The commands

Each device class implements a list of commands. Commands are very important because they are the client's major dials and knobs for controlling a device. Commands have a fixed calling syntax - consisting of one input argument and one output argument. Arguments type must be chosen in a fixed set of data types (All simple types and arrays of simple types plus array of strings and longs and array of strings and doubles). Commands can execute any sequence of actions.

Commands can be executed synchronously (the requester is blocked until the command ended) or asynchronously (the requester send the request and is called back when the command ended).

Commands are executed using two CORBA operations named **command_inout** for synchronous commands and **command_inout_async** for asynchronous commands. These two operations called a special method implemented in the device root class - the *command_handler* method. The *command_handler* calls an *is_allowed* method implemented in the device class before calling the command itself. The *is_allowed* method is specific to each command[1]. It checks to see whether the command to be executed is compatible with the present device state. The command function is executed only if the *is_allowed* method allows it. Otherwise, an exception is sent to the client.

3.3.2 The CORBA attributes

Some key data implemented for each device can be read without the need to call a command. These data are :

- The device state
- The device status
- The device name
- The administration device name called adm_name
- The device description

The device state is a number representing its state. A set of predefined states are defined in the TDSOM. The device status is a string describing in plain text the device state and any additional useful information of the device as a formatted ascii string. The device name is its name as defined in 3.3. For each set of devices grouped within the same server, an administration device is automatically added. This adm_name is the name of the administraion device. The device description is also an ascii string describing the device rule.

These five CORBA attributes are implemented in the device root class and therefore do not need any coding from the device class programmer. As explained in 3.1, the CORBA attributes are not allowed to raise exceptions whereas command (which are implemented using CORBA operations) can.

3.3.3 The TANGO attributes

In addition to commands, TANGO devices also support normalised data types called attributes[2]. Commands are device specific and the data they transport are not normalised i.e. they can be any one of the TANGO data types with no restriction on what each byte means. This means that it is difficult to interpret the output of a command in terms of what kind of value(s) it represents. Generic display programs need to know what the data returned represents, in what units it is, plus additional information like minimum, maximum, quality etc. Tango attributes solve this problem.

TANGO attributes are zero, one or two dimensional data which have a fix set of properties e.g. quality, minimum and maximum, alarm low and high. They are transferred in a specialised TANGO type and can be read or read-write. A device can support a list of attributes. Clients can read one or more attributes from one or more devices. To read TANGO attributes, the client uses the **read_attributes** operation. To write TANGO attributes, a client uses the **write_attributes** operation. To query a device for all the attributes it supports, a client uses the **get_attribute_config** operation. These three operations are defined in the device CORBA interface.

[1] In contrary to the state_handler method of the TACO device server model which is not specific to each command.

[2] TANGO attributes were known as signals in the TACO device server model

3.3.4 The remaining CORBA operations

The TDSOM also supports a list of actions defined as CORBA operations in the device interface and implemented in the device root class. Therefore, these actions are implemented automatically for every TANGO device. These operations are :

ping	to ping a device to check if the device is alive. Obviously, it checks only the connection from a client to the device and not all the device functionalities
command_list_query	request a list of all the commands supported by a device with their input and output types and description
command_query	request information about a specific command which are its input and output type and description
info	request general information on the device like its name, the host where the device server hosting the device is running...
black_box	read the device black-box as an array of strings

3.4 The server

Another integral part of the TDSOM is the server concept. The server (also referred as device server) is a process whose main task is to offer one or more services to one or more clients. To do this, the server has to spend most of its time in a wait loop waiting for clients to connect to it. The devices are hosted in the server process. A server is able to host several classes of devices. In the TDSOM, a device of the **DServer** class is automatically hosted by each device server. This class of device supports commands which enable remote device server process administration.

TANGO supports device server process on four operating system : Linux, Solaris, HP-UX and Windows NT.

3.5 The database

To achieve complete device independence, it is necessary however to supplement device classes with a possibility for configuring device dependencies at runtime. The utility which does this in the TDSOM is the **property database**. Properties[3] are identified by an ascii string and the device name. TANGO attributes are also configured using properties. This database is also used to store device network addresses (CORBA IOR's), list of classes hosted by a device server process and list of devices for each class in a device server process. The database ensure the uniqueness of device name and of alias. It also links device name and it list of aliases.

TANGO uses MySQL[4] as its database. MySQL is a relational database which implements a subset of the SQL language. However, this subset is enough to implement all the functionalities needed by the TDSOM. The database is accessed via a classical TANGO device hosted in a device server. Therefore, client access the database via TANGO commands requested on the database device. For a good reference on MySQL refer to [5]

3.6 The application programmers interface

To be filled in later

[3]Properties were known as resources in the TACO device server model

Chapter 4

The device server framework

This chaper will present the TANGO device server framework. It will introduce what is the device server pattern and then it will describe a complete device server framework. A definition of classes used by the device server framework is given in this chapter. This manual is not intended to give the complete and detailed description of classes data member or methods, refer to [6] to get this full description. But first, the naming convention used in this project is detailed.

The aim of the class definition given in this chapter is only to help the reader to understand how a TANGO device server works. For a detailed description of these classes (and their methods), refer to chapter 6 or to [6].

4.1 Naming convention and programming language

TANGO fully supports two different programming languages which are **C++** and **Java**. When the Java code differs from the C++ code, examples in both languages will be given. For C++, its standard library has been used. Details about this library can be found in [7].

Every software project needs a naming convention. The naming convention adopted for the TDSOM is very simple and only defines two guidelines which are:

- Class names start with uppercase and use capitalization for compound words (For instance MyClassName).
- Method names are in lowercase and use underscores for compound words (For instance my_method_name).

These conventions will be use herafter for both C++ and Java.

4.2 The device pattern

Device server are written using the Device pattern. The aim of this pattern is to provide the control programmer with a framework in which s/he can develop new control objects. The device pattern uses other design patterns like the Singleton, Command and Factory patterns. These paterns are fully described in [8]. The device pattern class diagram for stepper motor device is drawn in figure 4.1 . In this figure, only classes surrounded with a dash line square are device specific. All the other classes are part of the TDSOM core and are developped by the Tango system team. Different kind of classes are used by the device pattern.

- Three of them are root classes and it is only necessary to inherit from them. These classes are the **DeviceImpl**, **DeviceClass** and **Command** classes.

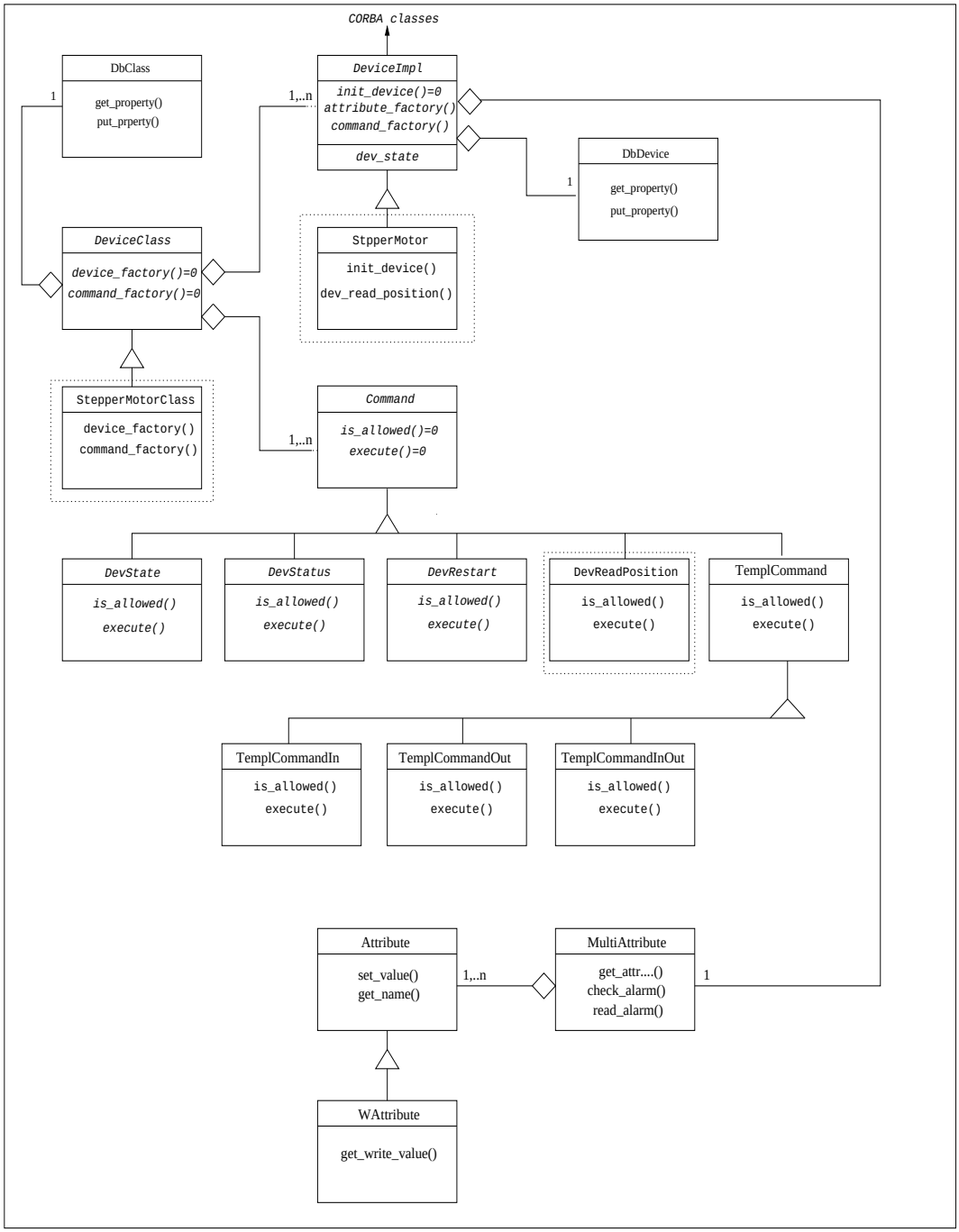


Figure 4.1: Device pattern class diagram

- Classes necessary to implement commands. The TDSOM supports two ways to create command : Using inheritance or using the template command model. It is possible to mix model within the same device pattern

  1. Using **inheritance**. This model of creating command heavily used the polymorphism offered by each modern object oriented programming language. In this schema, each command supported by a device via the command_inout or command_inout_async operation is implemented by a separate class. The Command class is the root class for each of these classes. It is an abstract class. A *execute* method must be defined in each sub-class. A *is_ allowed* method may also be re-defined in each class if the default one does not fulfill all the needs[1]. In our stepper motor device server example, the DevReadPosition command follows this model.
  2. Using the **template command** model. Using this model, it is not necessary to write one class for each command. You create one instance of classes already defined in the TDSOM for each command. The link between command name and method which need to be exceuted is done through pointers to method for C++ and through method names for Java. To support different kind of command, four classes are part of the TDSOM. These classes are :
     (a) The **TemplCommand** class for command without input or output parameter
     (b) The **TemplCommandIn** class for command with input parameter but without output parameter
     (c) The **TemplCommandOu**t class for command with output parameter but without input parameter
     (d) The **TemplCommandInOut** class for all the remaining commands

- Classes necessary to implement TANGO device attributes. All these classes are part of the TANGO core classes. These classes are the **MultiAttribute**, **Attribute** and **WAttribute** classes.

- The other are device specific. For stepper motor device, they are named StepperMotor, StepperMotorClass and DevReadPosition.

4.2.1 The DeviceImpl class

Description

This class is the device root class and is the link betwen the Device pattern and CORBA. It inherits from CORBA classes and implements all the methods needed to execute CORBA operations and attributes. For instance, its method *command_ inout* is executed when a client requests a command_inout operation. The method *name* of the DeviceImpl class is executed when a client requests the name CORBA attribute. This class also encapsulates some key device data like its name, its state, its status, its black box.... This class is an abstract class and cannot be instanciated as is.

Contents

The contents of this class can be summarise as :

- Different constructors and one destructor
- Methods to access instance data members outside the class or its derivated classes. These methods are necessary because data members are declared as protected.
- Methods triggered by CORBA attribute request

[1] The default is_allowed method behaviour is to always allows the command

- Methods triggered by CORBA operation request
- The *init_device()* method. This method makes the class abstract. It should be implemented by a sub-class. It is used by the inherited classes constructors.
- Methods triggered by the automatically added DevState and DevStatus commands. These methods are declared virtual and therefore can be redefined in sub-classes. These two commands are automatically added to the list of commands defined for a class of devices. They are discussed in chapter 4.5
- A method called *always_executed_hook()* always excuted for each command before the device state is tested for command execution. This method gives the programmer a hook where he(she) can program some mandatory action which must be done before any command execution. An example of the such action is an hardware access to the device to read its real hardware state.
- Methods triggerred by the read_attributes CORBA operation. The *read_attr_hardware* method is called once for each read_attributes call. The *read_attr* method is called for each attribute(s) involved in the read_attributes CORBA operation. These two methods are virtual and may be redefined in sub-classes.
- Method triggered by the write_attributes CORBA operation. This method is virtual and therefore may be redefined in a sub-class
- Methods for signal management (C++ specific)
- Data members like the device name, the device status, the device state
- Some private methods and data members

4.2.2 The DbDevice class

Each DeviceImpl instance is an aggregate with one instance of the DbDevice class. This DbDevice class can be used to query or modify device properties. It provides an easy to use interface for device objects in the database. The description of this class can be found in the Tango java or C++ API documentation.

4.2.3 The Command class

Description of the inheritance model

Within the TDSOM, each command supported by a device and implemented using the inheritance model is implemented by a separate class. The Command class is the root class for each of these classes. It is an abstract class. It stores the command name, the command argument types and description and mainly defines two methods which are the *execute* and *is_allowed* methods. The *execute* method should be implemented in each sub-class. A default *is_allowed* method exists for command always allowed.

Description of the template model

Using this method, it is not necessary to create a separate class for each device command. In this method, each command is represented by an instance of one of the template command classes. They are four template command classes. All these classes inherits from the Command class. These four classes are :

1. The **TemplCommand** class. One object of this class must be created for each command without input nor output parameters

2. The **TemplCommandIn** class. One object of this class must be created for each command without output parameter but with input parameter
3. The **TemplCommandOut** class. One object of this class must be created for each command without input parameter but with output parameter
4. The **TemplCommandInOut** class. One object of this class must be created for each command with input and output parameters

These four classes redefine the *execute* and *is_allowed* method of the Command class. These classes provides constructors which allow the user to :

- specify which method must be executed by these classes *execute* method
- optionally specify which method must be executed by these classes *is_allowed* method.

The method specification is done via pointer to method with C++ and simply with method name for java.

Remember that it is possible to mix command implementation method within the same device pattern.

Contents

The content of this class can be summarises as :

- Class constructors and destructor
- Declaration of the *execute* method
- Declaration of the *is_allowed* method
- Methods to read/set class data members
- Methods to extract data from the object used to transfer data on the network
- Methods to insert data into the object used to transfer data on the network
- Class data members like command name, command input data type, command input data description...

4.2.4 The DeviceClass class

Description

This class implements all what is specific for a controlled object class. For instance, every device of the same class supports the same list of commands and therefore, this list of available commands is stored in this DeviceClass. The stucture returned by the info operation contains a documentation URL[2]. This documentation URL is the same for every device of the same class. Therefore, the documentation URL is a data member of this class. There should have only one instance of this class per device pattern implementation. The device list is also stored in this class. It is an abstract class because the two methods *device_factory()* and *command_factory()* are declared as pure virtual. The rule of the *device_factory()* method is to create all the devices belonging to the device class. The rule of the *command_factory()* method is to create one instance of all the classes needed to support device commands. This class also stored the *attribute_factory* method. The rule of this method is to store in a vector of strings, the name of all the device attributes. This method has a default implemention which is an empty body for device without attribute.

[2]URL stands for **U**niform **R**esource **L**ocator

Contents

The contents of this class can be summarise as :

- The *command_handler* method
- Methods to access data members.
- Signal related method (C++ specific)
- Class constructor. It is protected to implements the Singleton pattern
- Class data members like the class command list, the device list...

4.2.5 The DbClass class

Each DeviceClass instance is an aggregate with one instance of the DbClass class. This DbClass class can be used to query or modify class properties. It provides an easy to use interface for device objects in the database. The description of this class can be found in the Tango java or C++ API documentation.

4.2.6 The MultiAttribute class

Description

Thic class is a container for all the TANGO attributes defined for the device. There is one instance of this class for each device. This class is mainly an aggregate of Attribute object(s). It hs been developped to ease TANGO attribute management.

Contents

The class contents could be summarises as :

- Miscellaneous methods to retrieve one attribute object in the aggregate
- Method to retrieve a list of attribute with an alarm level defined
- Get attribute number method
- Miscellaneous methods to check if an attribute value is outside the authorized limits
- Method to add messages for all attribute with an alarm set
- Data members with the attribute list

4.2.7 The Attribute class

Description

There is one object of this class for each device attribute. This class is used to store all the attribute properties, the attribute value and all the alarm related data.

Contents

- Miscellaneous method to get boolean attribute information
- Methods to access some data members
- Methods to get/set attribute properties
- Method to check if the attribute is in alarm condition
- Methods related to attribute data
- Friend function to print attribute properties
- Data members (properties value and attribute data)

4.2.8 The WAttribute class

Description

This class inherits from the Attribute class. There is one instance of this class for each writable device attribute. On top of all the data already managed by the Attribute class, this class stores the attribute set value.

Contents

Within this class, you will mainly find methods related to attribute set value storage and some data members.

4.2.9 The StepperMotor class

Description

This class inherits from the DeviceImpl class and is the class implementing the controlled object behaviour. Each command will trigger a method in this class written by the device server programmer and specific to the object to be controlled. This class also stores all the device specific data.

Definition

```
 1  class StepperMotor: public DeviceImpl
 2  {
 3  public :
 4          StepperMotor(DeviceClass *,string &);
 5          StepperMotor(DeviceClass *,const char *);
 6          StepperMotor(DeviceClass *,const char *,const char *);
 7          ~StepperMotor() {};
 8
 9          long dev_read_position(long);
10          long dev_read_direction(long);
11          bool direct_cmd_allowed(const CORBA::Any &);
12
13          virtual Tango_DevState dev_state();
14          virtual Tango_DevString dev_status();
15          virtual void always_executed_hook();
16
```

```
17          virtual void init_device();
18
19  protected :
20          long axis[AGSM_MAX_MOTORS];
21          long position[AGSM_MAX_MOTORS];
22          long direction[AGSM_MAX_MOTORS];
23          long state[AGSM_MAX_MOTORS];
24  };
```

Line 1 : The StepperMotor class inherits from the DeviceImpl class
Line 4-7 : Class constructors and destructor
Line 9 : Method triggered by the DevReadPosition command
Line 10-11 : Methods triggerred by the DevReadDirection command
Line 13 : Redefinition of the *dev_state* method of the DeviceImpl class. This method will be triggered by the DevState command
Line 14 : Redefinition of the *dev_status* method of the DeviceImpl class. This method will be triggered by the DevStatus command
Line 15 : Redefinition of the *always_executed_hook* method.
Line 17 : Definition of the *init_device* method (declared as pure virtual by the DeviceImpl class)
Line 20-23 : Device data

4.2.10 The StepperMotorClass class

Description

This class inherits from the DeviceClass class. Like the DeviceClass class, there should be only one instance of the StepperMotorClass. This is ensured because this class is written following the Singleton pattern as defined in [8]. All controlled object class data which should be defined only once per class must be stored in this object.

Definition

```
 1  class StepperMotorClass : public DeviceClass
 2  {
 3  public:
 4          static StepperMotorClass *init(const char *);
 5          static StepperMotorClass *instance();
 6          ~StepperMotorClass() {_instance = NULL;}
 7
 8  protected:
 9          StepperMotorClass(string &);
10          static StepperMotorClass *_instance;
11          void command_factory();
12
13  private:
14          void device_factory(Tango_DevVarStringArray *);
15  };
```

Line 1 : This class is a sub-class of the DeviceClass class
Line 4-5 and 9-10: Methods and data member necessary for the Singleton pattern
Line 6 : Class desctructor
Line 11 : Definition of the *command_factory* method declared as pure virtual in the DeviceClass call
Line 13-14 : Definition of the *device_factory* method declared as pure virtual in the DeviceClass class

4.2.11 The DevReadPosition class

Description

This is the class for the DevReadPosition command. This class implements the *execute* and *is_allowed* methods defined by the Command class. This class is necessary because this command is implemented using the inheritance model.

Definition

```
1  class DevReadPositionCmd : public Command
2  {
3  public:
4          DevReadPositionCmd(const char *,Tango_CmdArgType, Tango_CmdArgType, const ch
5          ~DevReadPositionCmd() {};
6
7          virtual bool is_allowed (DeviceImpl *, const CORBA::Any &);
8          virtual CORBA::Any *execute (DeviceImpl *, const CORBA::Any &);
9  };
```

Line 1 : The class is a sub class of the Command class
Line 4-5 : Class constructor and destructor
Line 7-8 : Definition of the *is_allowed* and *execute* method declared as pure virtual in the Command class.

4.3 Startup of a device pattern

To start the device pattern implementation for stepper motor device, four methods of the StepperMotorClass class must be executed. These methods are :

1. The creation of the StepperMethodClass singleton via its *init*() method
2. The *command_factory*() method of the StepperMotorClass class
3. The *attribute_factory*() method of the StepperMotorClass class. This method has a default empty body for device class without attributes.
4. The *device_factory*() method of the StepperMotorClass class

This startup procedure is described in figure 4.2 . The creation of the StepperMotorClass will automatically create an instance of the DeviceClass class. The constructor of the DeviceClass class will create the DevStatus and the DevState command objects and store them in its command list.

The *command_factory*() method will simply create all the user defined commands and add them in the command list.

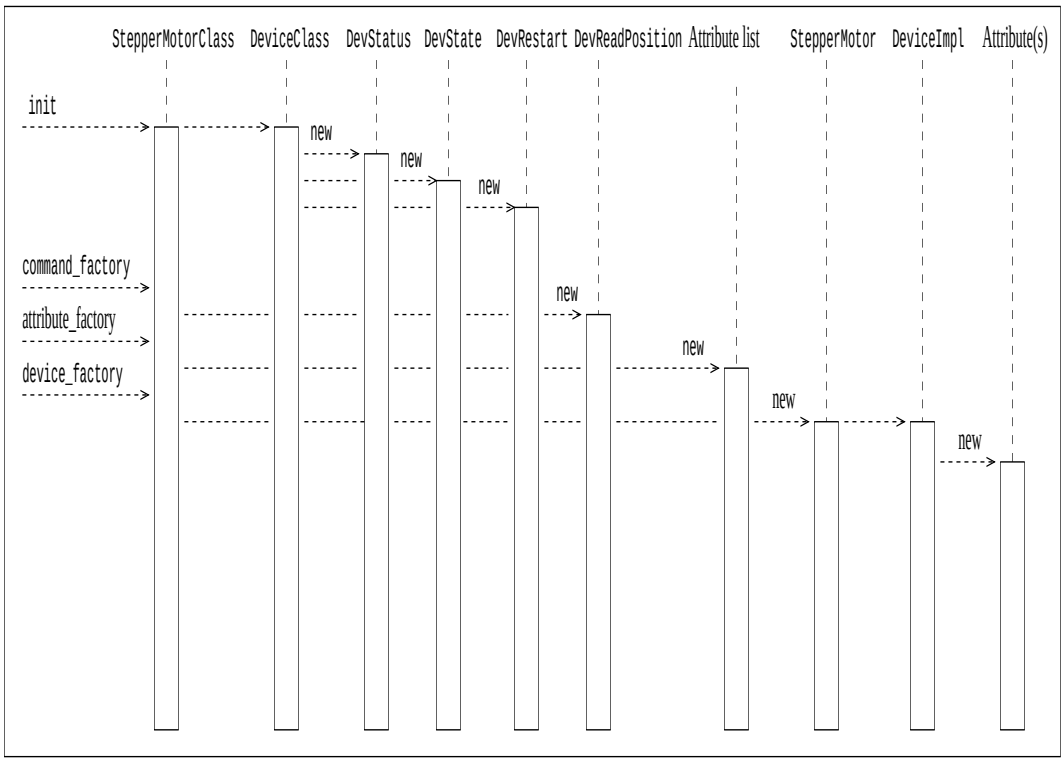


Figure 4.2: Device pattern startup sequence

The *attribute_factory*() method will simply build a list of device attribute names.

The *device_factory*() method will create each StepperMotor object and store them in the StepperMotorClass instance device list. The list of devices to be created and their names is passed to the *device_factory* method in its input argument. StepperMotor is a sub-class of DeviceImpl class. Therefore, when a StepperMotor object is created, a DeviceImpl object is also created. The DeviceImpl constructor builds all the device attribute object(s) from the attribute list built by the *attribute_factory()* method.

4.4 Command execution sequence

The figure 4.3 described how the method implementing a command is executed when a command_inout CORBA operation is requested by a client. The *command_inout* method of the StepperMotor object (inherited from the DeviceImpl class) is triggered by an instance of a class generated by the CORBA IDL compiler. This method calls the *command_handler*() method of the StepperMotorClass object (inherited from the DeviceClass class). The *command_handler* method searchs in its command list for the wanted command (using its name). If the command is found, the *always_executed_hook* method of the StepperMotor object is called. Then, the *is_allowed* method of the wanted command is executed. If the *is_allowed* method returns correctly, the *execute* method is executed. The *execute* method extracts the incomming data from the CORBA object use to transmit data over the network and calls the user written method which implements the command.

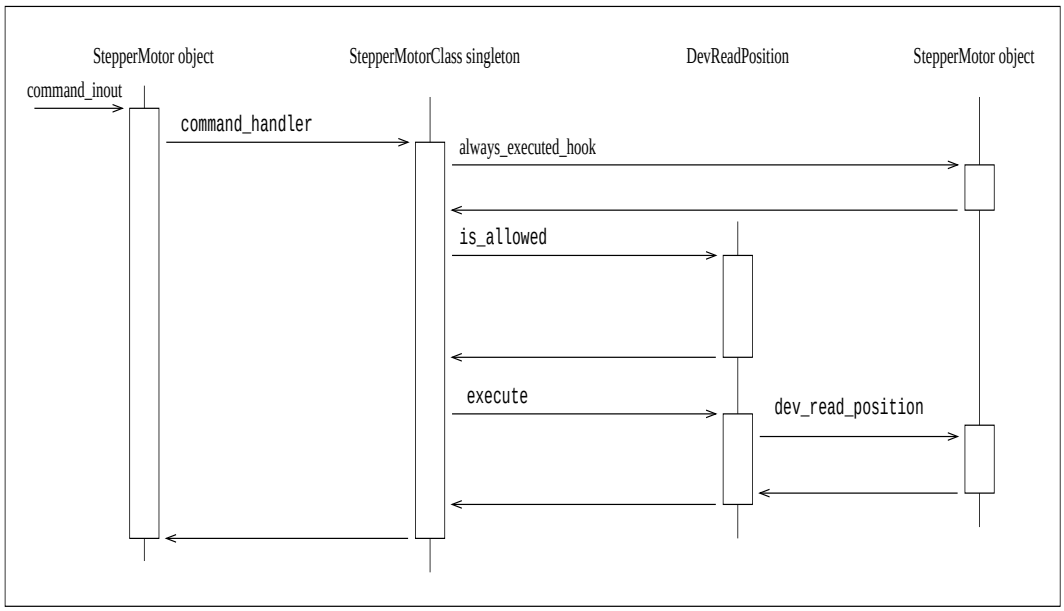


Figure 4.3: Command execution timing

4.5 The automatically added commands

In order to increase the common behaviour of every kind of devices in a TANGO control system, three commands are automatically added to each class of devices. These commands are :

- DevState
- DevStatus

The default behaviour of the method called by the DevState command depends on the device state. If the device state is ON or ALARM, the method will :

- read the attribute(s) with an alarm level defined
- check if the read value is above/below the alarm level and eventually change the device state to ALARM.
- returns the device state.

For all the other device state, the method simply returns the device state stored in the DeviceImpl classs. Nevertheless, the method used to return this state (called *dev_state*) is defined as virtual and can be redefined in DeviceImpl sub-class. The difference between the default DevState command and the state CORBA attribute is the ability of the DevState command to signal an error to the caller by throwing an exception.

The default behaviour of the method called by the DevStatus command depends on the device state. If the device state is ON or ALARM, the method returns the device status stored in the DeviceImpl class plus additional message(s) for all the attributes which are in alarm condition. For all the other device state, the method simply returns the device status as it is stored in the DeviceImpl class. Nevertheless, the method used to return this status (called *dev_status*) is defined as virtual and can be redefined in DeviceImpl sub-class. The difference between the default DevStatus command and the status CORBA attribute is the abaility of the DevStatus command to signal an error to the caller by throwing an exception.

4.6 Reading/Writing attributes

4.6.1 Reading attributes

A Tango client is able to read Tango attribute(s) with the CORBA read_attributes call. Inside the device server, this call will trigger two methods of the device class (StepperMotor in our example) :

1. A method call *read_attr_hardware()*. This method is called first and one time per read_attributes CORBA call. The aim of this method is to read the device hardware and to store the result in a device class data member.

2. A method call *read_attr()*. This method is called as many times as attributes to be read. It has one parameter which is a reference to the Attribute object to be read. The aim of this method is to extract the real attribute value from the harware read-out and to store the attribute value into the attribute object.

These two methods have a default empty body for classes without attributes.

4.6.2 Writing attributes

A Tango client is able to write Tango attribute(s) with the CORBA write_attributes call. Inside a device server, this call will trigger one method of the device class (StepperMotor in our example) called *write_attr_hardware()*. This method has one parameter which is a vector of long. Each vector element is the index into the attribute vector[3]of the attribute to be written. The aim of this method is to get the data to be written from the WAttribute object and to write it into the corresponding hardware.

4.7 The device server framework

4.7.1 Vocabulary

A device server pattern implementation is embedded in a process called a **device server**. Several instances of the same device server process can be used in a TANGO control system. To identify instances, a device server process is started with an **instance name** which is different for each instance. The device server name is the couple device server executable name/device server instance name. For instance, a device server started with the following command

Perkin id11

starts a device server process with an instance name id11, an executable name Perkin and a device server name Perkin/id11.

4.7.2 The DServer class

In order to simplify device server process administration, a device of the DServer class is automatically added to each device server process. Thus, every device server process supports the same set of administration commands. The implementation of this DServer class follows the device pattern and therefore, its device behaves like any other devices. The device name is

dserver/device server executable name/device server instance name

For instance, for the device server process described in chapter 4.7.1, the dserver device name is dserver/perkin/id11. This name is the name returned by the adm_name CORBA attribute available for every device. On top of the two automatically added commands, this device supports the following commands :

[3]The vector attribute is a MultiAttribute data member

- DevRestart
- DevRestartServer
- DevQueryClass
- DevQueryDevice
- DevKill
- DevSetTraceLevel
- DevGetTraceLevel
- DevSetTraceOutput
- DevGetTraceOutput

These commands will be fully described later in this document.

Several controlled object classes can be embedded within the same device server process and it is the rule of this device to create all these device server patterns and to call their command and device factories as described in 4.3. The name and number of all the classes to be created is known to this device after the execution of a method called *class_factory*. With C++, it is the user responsability to write this method. Using Java, this method is already written and automatically retrieves which classes must be created and creates them.

4.7.3 The Tango::Util class

Description

This class merges a complete set of utilities in the same class. It is implemented as a singleton and there is only one instance of this class per device server process. It is mandatory to create this instance in order to run a device server. The description of all the methods implemented in this class can be found in [6].

Contents

Within this class, you can find :

- Static method to create/retrieve the singleton object
- Miscellaneous utility methods like getting the server output trace level, getting the CORBA ORB pointer, retrieving device server instance name, getting the server PID and more. Please, refer to [6] to get a complete list of all these utility methods.
- Method to create the device pattern implementing the DServer class (*server_init()*)
- Method to start the server (*server_run()*)
- TANGO database related methods

4.7.4 A complete device server

Within a complete device server, at least two implementations of the device server pattern are created (one for the dserver object and the other for the class of devices to contol). On top of that, one instance of the Tango::Util class must also be created. A drawing of a complete device server is in figure 4.4

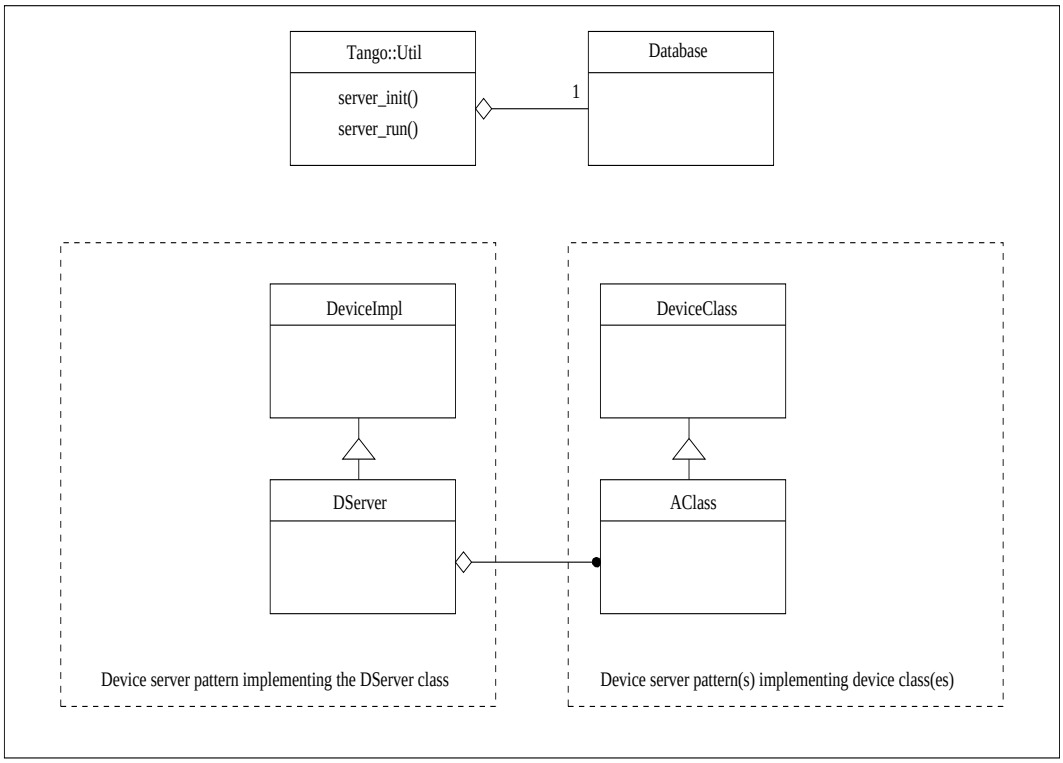


Figure 4.4: A complete device server

4.7.5 Device server startup sequence

The device server startup sequence is the following :

1. Create an instance of the Tango::Util class. This will initialize the CORBA Object Request Broker
2. Called the *server_init* method of the Tango::Util instance The call to this method will :
   (a) Create the DServerClass object of the device pattern implementing the DServer class. This will create the dserver object which during its construction will :
      i. Called the *class_factory* method of the DServer object. This method must create all the xxxClass instance for all the device pattern implementation embedded in the device server process.
      ii. Call the *command_factory* and *device_factory* of all the classes previously created. The list of devices passed to each call to the *device_factory* method is retrieved from the TANGO database.
3. Wait for incoming request with the *server_run()* method of the Tango::Util class.

Chapter 5

Exchanging data between client and server using commands

Exchanging data between clients and server means most of the time passing data between processes running on different computer using the network. Tango limits the type of data exchanged between client and server and defines a way to exchange these data. This chapter details these features. Memory allocation and error reporting are also discussed.

All the rules described in this chapter are valid only for data exchanged between client and server. For device server internal data, classical C++ or Java types can be use.

5.1 Command data types

Commands have a fixed calling syntax - consisting of one input argument and one output argument. Arguments type must be chosen out of a fixed set of 19 data types. The following table details type name, code and the corresponding CORBA IDL types.

The type name used in the type name column of this table is the C++ name. In the IDL file, all the Tango definition are grouped in a IDL module named Tango. The IDL module maps to C++ namespace. Therefore, all the data type are parts of a namespace called Tango. For Java, the IDL module maps to Java package and name are not changed related to the IDL file.

Type name	IDL type
Tango::DevBoolean	boolean
Tango::DevShort	short
Tango::DevLong	long
Tango::DevFloat	float
Tango::DevDouble	double
Tango::DevUShort	unsigned short
Tango::DevULong	unsigned long
Tango::DevString	string
Tango::DevVarCharArray	sequence of unsigned char
Tango::DevVarShortArray	sequence of short
Tango::DevVarLongArray	sequence of long
Tango::DevVarFloatArray	sequence of float
Tango::DevVarDoubleArray	sequence of double
Tango::DevVarUShortArray	sequence of unsigned short
Tango::DevVarULongArray	sequence of unsigned long
Tango::DevVarStringArray	sequence of string
Tango::DevVarLongStringArray	structure with a sequence of long and a sequence of string
Tango::DevVarDoubleStringArray	structure with a sequence of double and a sequence of string
Tango::DevState	enumeration

The CORBA Interface Definition Language uses a type called **sequence** for variable length array. This sequence type is mapped differently according to the language used (C++ or Java). The Tango::DevUxxx types are used for unsigned types. The Tango::DevVarxxxxArray must be used when the data to be transferred are variable length array. The Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray are structures with two fields which are variable length array of long and variable length array of strings for the Tango::DevVarLongStringArray and variable length array of double and variable length array of string for the Tango::DevVarDoubleStringArray. The Tango::DevState type is used by the DevState command to return the device state.

5.1.1 Using command data types with C++

Unfortunately, the mapping between IDL and C++ was defined before the C++ class library had been standardised. This explains why the standard C++ string class or vector classes are not used in the IDL to C++ mapping.

TANGO commands argument types can be grouped on five groups depending on the IDL data type used. These groups are :

1. Data type using basic types (Tango::DevBoolean, Tango::DevShort, Tango::DevLong, Tango::DevFloat, Tango::DevDouble, Tango::DevUshort and Tango::DevULong)

2. Data type using strings (Tango::DevString type)

3. Data types using sequences (Tango::DevVarxxxArray types except Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray)

4. Data types using structures (Tango::DevVarLongStringArray and Tango::DevVarDoubleStringArray types)

5. Data type using enumeration (Tango::DevState type)

In the following sub chapters, only summaries of the IDL to C++ mapping are given. For a full description of the C++ mapping, please refer to [2]

Basic types

For these types, the mapping between IDL and C++ is obvious and defined in the following table.

Tango type name	IDL type	C++	typedef
Tango::DevBoolean	boolean	CORBA::Boolean	unsigned char
Tango::DevShort	short	CORBA::Short	short
Tango::DevLong	long	CORBA::Long	long
Tango::DevFloat	float	CORBA::Float	float
Tango::DevDouble	double	CORBA::Double	double
Tango::DevUShort	unsigned short	CORBA::UShort	unsigned short
Tango::DevULong	unsigned long	CORBA::ULong	unsigned long

The types defined in the column named C++ should be used for a better code portability. All these typs are defined in the CORBA namespace and therefore their qualified names is CORBA::xxx.

Strings

Strings are mapped to **char ***. The use of *new* and *delete* for dynamic allocation of strings is not portable. Instead, you must use helper functions defined by CORBA (in the CORBA namspace). These functions are :

```
char *CORBA::string_alloc(unsigned long len);
char *CORBA::string_dup(const char *);
void CORBA::string_free(char *);
```

These functions handle dynamic memory for strings. The *string_alloc* function allocates one more byte than requested by the len parameter (for the trailing 0). The function *string_dup* combines the allocation and copy. Both *string_alloc* and *string_dup* return a null pointer if allocation fails. The *string_free* function must be used to free memory allocated with *string_alloc* and *string_dup*. Calling *string_free* for a null pointer is safe and does nothing. The following code fragment is an example of the Tango::DevString type usage

```
1     Tango::DevString str = CORBA::string_alloc(5);
2     strcpy(str,"TANGO");
3
4     Tango::DevString str1 = CORBA::string_dup("Do you want to danse TANGO?");
5
6     CORBA::string_free(str);
7     CORBA::string_free(str1);
```

Line 1-2 : TANGO is a five letters string. The CORBA::string_alloc function parameter is 5 but the function allocates 6 bytes

Line 4 : Example of the CORBA::string_dup function

Line 6-7 : Memory deallocation

Sequences

IDL sequences are mapped to C++ classes that behave like vectors with a variable number of elements. Each IDL sequence type results in a separate C++ class. Within each class representing a IDL sequence types, you find the following method (only the main methods are related here) :

1. Four constructors.
   (a) A default constructor which creates an empty sequence.
   (b) The maximun constructor which creates a sequence with memory allocated for at least the number of elements passed as argument. This does not limit the number of element in the sequence but only the way how memory is allocated to store element
   (c) A sophisticated constructor where it is possible to assign the memory used by the sequence with a preallocated buffer.
   (d) A copy constructor which does a deep copy
2. An assignement operator which does a deep copy
3. A *length* accessor which simply returns the current number of elements in the sequence
4. A *length* modifier which changes the length of the sequence (which is different than the number of elements in the sequence)
5. Overloading of the [] operator. The subscript operator [] provides access to the sequence element. For a sequence containing elements of type T, the [] operator is overloaded twice to return value of type T & and const T &. Insertion into a sequence using the [] operator for the const T & make a deep copy. Sequence are numbered between 0 and *length*() -1.

Note that using the maximum constructor will not prevent you from setting the length of the sequence with a call to the length modifier. The following code fragment is an example of how to use a Tango::DevVarLongArray type

```
1     Tango::DevVarLongArray *mylongseq_ptr;
2     mylongseq_ptr = new Tango::DevVarLongArray();
3     mylongseq_ptr->length(4);
4
5     (*mylongseq_ptr)[0] = 1;
6     (*mylongseq_ptr)[1] = 2;
7     (*mylongseq_ptr)[2] = 3;
8     (*mylongseq_ptr)[3] = 4;
9
10    // (*mylongseq_ptr)[4] = 5;
11
12    CORBA::Long nb_elt = mylongseq_ptr->length();
13
14    mylongseq_ptr->length(5);
15    (*mylongseq_ptr)[4] = 5;
16
17    for (int i = 0;i < mylongseq_ptr->length();i++)
18         cout << "Sequence elt " << i + 1 << " = " << (*mylongseq_ptr)[i] << endl;
```

Line 1 : Declare a pointer to Tango::DevVarLongArray type which is a sequence of long
Line 2 : Create an empty sequence
Line 3 : Change the length of the sequence to 4
Line 5 - 8 : Initialise sequence elements
Line 10 ; Oups !!! The length of the sequence is 4. The behaviour of this line is undefined and may be a core can be dumped at run time
Line 12 : Get the number of element actually stored in the sequence
Line 14-15 : Grow the sequence to five elements and initialise element number 5
Line 17-18 : Print sequence element
Another example for the Tango::DevVarStringArray type is given

```
1     Tango::DevVarStringArray mystrseq(4);
2     mystrseq.length(4);
3
4     mystrseq[0] = CORBA::string_dup("Rock and Roll");
5     mystrseq[1] = CORBA::string_dup("Bossa Nova");
6     mystrseq[2] = CORBA::string_dup(“Waltz”);
7     mystrseq[3] = CORBA::string_dup("Tango");
8
9     CORBA::Long nb_elt = mystrseq.length();
10
11    for (int i = 0;i < mystrseq.length();i++)
12         cout << "Sequence elt " << i + 1 << " = " << mystrseq[i] << endl;
```

Line 1 : Create a sequence using the maximum constructor
Line 2 : Set the sequence length to 4. This is mandatory even if you used the maximum constructor.
Line 4-7 : Populate the sequence
Line 9 : Get how many strings are stored into the sequence
Line 11-12 : Print sequence elements.

Structures

Only two TANGO types are defined as structures. These types are the Tango::DevVarLongStringArray and the Tango::DevVarDoubleStringArray. IDL structures map to C++ structures with corresponding members. For the Tango::DevVarLongStringArray, the two members are named *svalue* for the sequence of strings and *lvalue* for the sequence of longs. For the Tango::DevVarDoubleStringArray, the two structure members are called *svalue* for the sequence of strings and *dvalue* for the sequence of double. An example of the usage of the Tango::DevVarLongStringArray type is detailed below.

```
1     Tango::DevVarLongStringArray my_vl;
2
3     myvl.svalue.length(2);
4     myvl.svalue[0] = CORBA_string_dup("Samba");
5     myvl.svalue[1] = CORBA_string_dup("Rumba");
6
7     myvl.lvalue.length(1);
8     myvl.lvalue[0] = 10;
```

Line 1 : Declaration of the structure
Line 3-5 : Initialisation of two strings in the sequence of string member
Line 7-8 : Initialistaion of one long in the sequence of long member

Enumeration

Only one TANGO type is an enumeration. This is the Tango::DevState type used to transfer device state between client and server. IDL enumerated types map to C++ enumerations (amazing no!) with a trailing dummy enumerator to force enumeration to be a 32 bit type. The first enumerator will have the value 0, the next one will have the value 1 and so on.

```
1     Tango::DevSate state;
2
3     state = Tango::ON;
4     state = Tango::FAULT;
```

5.1.2 Using command data types with Java

All the rules described in this chapter are valid only for data exchanged between client and server. For device server internal data, classical Java types can be use.

TANGO commands argument types can be grouped on four groups depending on the IDL data type used. These groups are :

1. Data type using basic types (DevBoolean, DevShort, DevLong, DevFloat, DevDouble, DevUShort, DevULong and DevString)
2. Data types using sequences (DevVarxxxArray types except DevVarLongStringArray and DevVarDoubleStringArray)
3. Data types using structures (DevVarLongStringArray and DevVarDoubleStringArray types)
4. Data type using enumeration (DevState type)

In the following sub chapters, only summaries of the IDL to Java mapping are given. For a full description of the Java mapping, please refer to [10].

Basic types

For these types, the mapping between IDL and Java is obvious and defined in the following table.

Tango type name	IDL type	Java type
DevBoolean	boolean	boolean
DevShort	short	short
DevLong	long	int
DevFloat	float	float
DevDouble	double	double
DevString	string	String
DevUShort	unsigned short	short
DevULong	unsigned long	int

The Java int is a 32 bits type[1] and therefore, the DevLong type maps to Java int. Java does not support unsigned types, this is why the DevUShort type maps to short and the DevULong

[1] The Java long type is a 64 bits data type

type maps to int. In the contrary of C++, Java does not support a preprocessor and therefore, declaring a data from the DevLong type (or any other type in the previous table) will result in compiler errors. Instead, the Java types must be used.

IDL string maps directly to java.lang.String class.

Sequences

IDL sequences map to Java array. The following tables details the mapping used for Tango sequence types.

Tango type name	IDL type	Java type
DevVarCharArray	sequence of byte	byte[]
DevVarShortArray	sequence of short	short[]
DevVarLongArray	sequence of long	int[]
DevVarFloatArray	sequence of float	float[]
DevVarDoubleArray	sequence of double	double[]
DevVarUShort array	sequence of unsigned short	short[]
DevVarULongArray	sequence of unsigned long	int[]
DevVarStringArray	sequence of string	String[]

Structures

IDL structures map to a final Java class with the same name. This class provides instance variables for all IDL structure fields. It also provides a default constructor and a constructor from all structures fields values. The class name, the field name and types are summaries in the following table

Tango type name	Java class name	field name	field Java type
DevVarLongStringArray	DevVarLongStringArray	lvalue	int[]
		svalue	String[]
DevVarDoubleStringArray	DevVarDoubleStringArray	dvalue	double[]
		svalue	String[]

Enumeration

Enumeration does not exist in Java. An IDL enumeration is mapped to a final class with the same name as the enum type. This class has the following members :

1. A *value* method which returns the value as an integer.
2. A pair of static data members per label.
   (a) The first one is an integer with a name equals to the label name prepanded with an underscore ("_") like _ON for instance.
   (b) The second one is a reference to an object of the class representing the enumeration with its value set to the label value.
3. An integer conversion method called *from_int* which returns a reference to an object of the class representing the enumeration
4. A private constructor

The following code fragment is an example of Tango command data types usage

```
1           short l = 2;
2
3           String[] str_array = new String[2];
4           str_array[0] = new String("Be Bop");
5           str_array[1] = new String("Break dance");
6
7           System.out.println("Elt nb in DevVarStringArray data " + str_array.length);
8           for (int i = 0;i < str_array.length;i++)
9                   System.out.println("Element value = " + str_array[i]);
10
11          DevVarLongStringArray ls = new DevVarLongStringArray();
12          ls.lvalue = new int[1];
13          ls.lvalue[0] = 1;
14          ls.svalue = new String[2];
15          ls.svalue[0] = new String("Smurf");
16          ls.svalue[1] = new String("Pogo");
17
18          DevState st = DevState.FAULT;
19          switch (st.value())
20          {
21          case DevState._ON :
22                  System.out.println("The state is ON");
23                  st = DevState.FAULT;
24                  break;
25
26          case DevState._FAULT :
27                  System.out.println("The state is FAULT");
28                  st = DevState.ON;
29                  break;
30          }
```

Line 1 : Use of a DevShort type (pretty simple no)
Line 3-5 : Use of a DevVarStringArray data type with 2 elements
Line 7-9 : Print DevVarStringArray data element number and value
Line 11-16 : Use of a DevVarLongStringArray data type
Line 18 : Initialisation of a DevState data with the FAULT state
Line 19 : Test on the DevState data value
Line 21 : Use the integer value associated to each enumeration label to test DevState data
Line 23 : Update DevState data value

5.2 Passing data between client and server

In order to have one definition of the CORBA operation used to send a command to a device whatever the command data type is, TANGO uses CORBA IDL **any** object. The IDL type *any* provides a universal type that can hold a value of arbitrary IDL types. Type *any* therefore allows you to send and receive values whose types are not fixed at compile time.

Type *any* is often compared to a void * in C. Like a pointer to void, an *any* value can denote a datum of any type. However, there is an important difference; whereas a void * denotes a completely untyped value that can be interpreted only with advance knowledge of its type, values of type *any* mainain type safety. For example, if a sender places a string value into an *any*, the receiver cannot extract the string as a value of the wrong type. Attemps to read the contents of an *any* as the wrong type cause a run-time error.

Internally, a value of type *any* consists of a pair of values. One member of the pair is the actual value contained inside the *any* and the other member of the pair is the type code. The type code is a description of the value's type. The type description is used to enforce type safety when the receiver extracts the value. Extraction of the value succeeds only if the receiver extracts the value as a type that matches the information in the type code.

Within TANGO, the command input and output parameters are objects of the IDL *any* type. Only insertion/extraction of all types defined as command data types is possible into/from these *any* objects.

5.2.1 C++ mapping for IDL any type

The IDL any maps to the C++ class **CORBA::Any**. This class contains a large number of methods with mainly methods to insert/extract data into/from the any. It provides a default constructor which builds an any which contains no value and a type code that indicates "no value". Such an any must be used for command which does not need input or output parameter. The operator <<= is overloaded many times to insert data into an any object. The operator >>= is overloaded many times to extract data from an any object.

Inserting/Extracting TANGO basic types

The insertion or extraction of TANGO basic types is straight forward using the <<= or >>= operators. Nevertheless, the Tango::DevBoolean type is mapped to a unsigned char and other IDL types are also mapped to char C++ type (The unsigned is not taken into account in the C++ overloading algorithm). Therefore, it is not possible to use operator overloading for these IDL types which map to C++ char. For the Tango::DevBoolean type, you must use the *CORBA::Any::from_boolean* or *CORBA::Any::to_boolean* intermediate objects defined in the CORBA::Any class.

Inserting/Extracting TANGO strings

The <<= operator is overloaded for const char * and always makes a deep copy. This deep copy is done using the CORBA::*string_dup* function. The extraction of strings uses the >>= overloaded operator. The main point is that the Any object retains ownership of the string, so the returned pointer points at memory inside the Any. This means that you must not deallocate the extracted string and you must treat the extracted string as read-only.

Inserting/Extracting TANGO sequences

Insertion and extraction of sequences also uses the overloaded <<= and >>= operators. The insertion operator is overloaded twice: once for insertion by reference and once for insertion by pointer. If you insert a value by reference, the insertion makes a deep copy. If you insert a value by pointer, the Any assumes the ownership of the pointed-to memory.

Extraction is always by pointer. As with strings, you must treat the extracted pointer as read-only and must not deallocate it because the pointer points at memory internal to the Any.

Inserting/Extracting TANGO structures

This is identical to inserting/extracting sequences.

Inserting/Extracting TANGO enumeration

This is identical to inserting/extracting basic types

```
 1          CORBA::Any a;
 2          Tango::DevLong l1,l2;
 3          l1 = 2;
 4          a <<= l1;
 5          a >>= l2;
 6
 7          CORBA::Any b;
 8          Tango::DevBoolean b1,b2;
 9          b1 = true;
10          b <<= CORBA::Any::from_boolean(b1);
11          b >>= CORBA::Any::to_boolean(b2);
12
13          CORBA::Any s;
14          Tango::DevString str1,str2;
15          str1 = "I like dancing TANGO";
16          s <<= str1;
17          s >>= str2;
18
19  //      CORBA_string_free(str2);
20  //      a <<= CORBA_string_dup("Oups");
21
22          CORBA::Any seq;
23          Tango::DevVarFloatArray fl_arr1;
24          fl_arr1.length(2);
25          fl_arr1[0] = 1.0;
26          fl_arr1[1] = 2.0;
27          seq <<= fl_arr1;
28          const Tango::DevVarFloatArray *fl_arr_ptr;
29          seq >>= fl_arr_ptr;
30
31  //      delete fl_arr_ptr;
```

Line 1-5 : Insertion and extraction of Tango::DevLong type

Line 7-11 Insertion and extraction of Tango::DevBoolean type using the CORBA::Any::from_boolean and CORBA::Any::to_boolean intermediate structure

Line 13-17 : Insertion and extraction of Tango::DevString type

Line 19 : Wrong ! You should not deallocate a string extracted from an any

Line 20 : Wrong ! Memory leak because the <<= operator will do the copy.

Line 22-29 : Insertion and extraction of Tango::DevVarxxxArray types. This is an insertion by reference and the use of the <<= operator makes a deep copy of the sequence. Therefore, after line 27, it is possible to deallocate the sequence

Line 31: Wrong.! You should not deallocate a sequence extracted from an any

5.2.2 The insert and extract methods of the Command class

In order to simplify the insertion/extraction into/from Any objects, small helper methods have been written in the Command class. The signatures of these methods are :

```
1          void extract(const CORBA::Any &,<Tango type> &);
2          CORBA::Any *insert(<Tango type>);
```

An *extract* method has been written for all Tango types. These method extract the data from the Any object passed as parameter and throw an exception if the Any data type is incompatible with the awaiting type. An *insert* method have been written for all Tango types. These method create an Any object, insert the data into the Any and return a pointer to the created Any. For Tango types mapped to sequences or structures, two *insert* methods have been written: one for the insertion from pointer and the other for the insertion from reference. For Tango strings, two *insert* methods have been written: one for insertion from a classical Tango::DevString type and the other from a const Tango::DevString type. The first one deallocate the memory after the insert into the Any object. The second one only inserts the string into the Any object.

The previous example can be rewritten using the insert/extract helper methods (We suppose that we can use the Command class insert/extract methods)

```
 1          Tango::DevLong l1,l2;
 2          l1 = 2;
 3          CORBA::Any *a_ptr = insert(l1);
 4          extract(*a_ptr,l2);
 5
 6          Tango::DevBoolean b1,b2;
 7          b1 = true;
 8          CORBA::Any *b_ptr = insert(b1);
 9          extract(*b_ptr,b2);
10
11          Tango::DevString str1,str2;
12          str1 = "I like dancing TANGO";
13          CORBA::Any *s_ptr = insert(str1);
14          extract(*s_ptr,str2);
15
16          Tango::DevVarFloatArray fl_arr1;
17          fl_arr1.length(2);
18          fl_arr1[0] = 1.0;
19          fl_arr1[1] = 2.0;
20          insert(fl_arr1);
21          CORBA::Any *seq_ptr = insert(fl_arr1);
22          Tango::DevVarFloatArray *fl_arr_ptr;
23          extract(*seq_ptr,fl_arr_ptr);
```

Line 1-4 : Insertion and extraction of Tango::DevLong type

Line 6-9 : Insertion and extraction of Tango::DevBoolean type

Line 11-14 : Insertion and extraction of Tango::DevString type

Line 16-23 : Insertion and extraction of Tango::DevVarxxxArray types. This is an insertion by reference which makes a deep copy of the sequence. Therefore, after line 20, it is possible to deallocate the sequence

5.2.3 Java mapping for IDL any type

The IDL any maps to the Java class **org.omg.CORBA.Any** . This class has all the necessary methods to insert and extract instances of IDL native types (short, int, float, string..). The method name to insert native IDL types is *insert_* <*type name*> (*insert_short*(), *insert_float*(), *insert_string*()). They all take a reference to the element to be inserted as argument. The

method name to extract basic types is *extract_ <type name>* (*extract_ short*(), *extract_ float*() or *extract_ string*()). These extract methods do not need argument and return a reference to the extracted data. If the extraction operations have a mismatched type, the CORBA BAD_OPERATION exception is raised. An "any" object is constructed with the *create_ any*() method of the CORBA "orb" object. This orb object represents the Object Request Broker. Within a Tango device server, you can retrieve it with a method of the TangoUtil class described in [6].

Inserting/Extracting TANGO basic types and strings

The insertion or extraction of TANGO basic types and strings is straight forward using the insert or extract methods provided by the org.omg.CORBA.Any class.

Inserting/Extracting TANGO sequences, structures or enumeration.

The IDL to Java compiler generates Helper classes for all types defined in the IDL file. The generated classes name is the name of the type followed by the suffix **Helper** (DevVarCharArrayHelper, DevLongHelper). Classes are generated even for types which directly map to native Java types. Several static methods needed to manipulate the type are supplied in these classes. These include "Any" insert and extract operations for the type. For a data type <typename>, the insert and extract method are :

- public static void insert(org.omg.CORBA.Any a, <typename> t) {..}
- public static <typename> extract(Any a) {...}

Such classes exists for all the TANGO data types. The following code fragment is an example of the insertion/extraction in/from Any object with Java

```
 1  Any a = TangoUtil.instance().get_orb().create_any();
 2  int l1 = 1;
 3  a.insert_long(l1);
 4  int l2 = a.extract_long();
 5
 6  DevLongHelper.insert(a,l1);
 7  int l3 = DevLongHelper.extract(a);
 8
 9  Any s = TangoUtil.instance().get_orb().create_any();
10  String str = new String("I like dancing TANGO");
11  s.insert_string(str);
12  String str_ex = s.extract_string();
13
14  DevStringHelper.insert(s,str);
15  String str_help = DevStringHelper.extract(s);
16
17  Any arr = TangoUtil.instance().get_orb().create_any();
18  int[] array = new int[2];
19  array[0] = 1;
20  array[1] = 2;
21  DevVarLongArrayHelper.insert(arr,array);
22  int[] array_ext = DevVarLongArrayhelper.extract(arr);
```

Line 1 : Create an instance of the Any class.

Line 3 : Insert a DevLong data into the Any object. The method name is insert_long because this is a method to insert an IDL long type into the object even if the IDL long type maps to an int in Java.

Line 4 : Extract a DevLong type from the Any

Line 6-7 : Insert or Extract DevLong data type to/from the Any object using the Helper class.

Line 9-12 : Create an Any object and a DevString data. Insert and Extract this string into/from the Any using the method provided by the any object

Line 14-15 : Insert or Extract string into/from the Any using methods provided by the Helper class

Line 17-22 : The same thing for data of the DevVarLongArray type. Note that DevVarLongArray is not a basic IDL type and the Any class does not provide method to insert/extract data of this type into/from the Any. The use of the methods provided by the Helper class is mandatory in this case.

5.2.4 The insert and extract methods of the Command class for Java

In order to simplify the insertion/extraction into/from Any objects, small helper methods have been written in the Command class. The signatures of these methods are :

```
1          <java type> extract_<Tango type_name>(Any);
2          Any insert(<Tango type>);
```

An *extract* method has been written for all Tango types. These method extract the data from the Any object passed as parameter and throw an exception if the Any data type is incompatible with the awaiting type. All these *extract* methods take the same input parameter and only differ in their return type which is not taken into account for method overloading. Therefore, the name of the method depends on the type of the data to be extracted. The following is some example of these method names and signatures :

- *int extract_DevLong(Any) throws DevFailed* for the DevLong type
- *int[] extract_DevVarULongArray(Any) throws DevFailed* for DevVarULongArray type
- *String[] extract_DevVarStringArray(Any) throws DevFailed* for DevVarStringArray

An *insert* method have been written for all Tango types. These method create an Any object, insert the data into the Any and return a pointer to the created Any. The previous example can be rewritten using the insert/extract helper methods (We suppose that we can use the Command class insert/extract methods)

```
1  int l1 = 1;
2  Any a = insert(l1);
3  int l2 = extract_DevLong(a);
4
5  String str = new String("I like dancing TANGO");
6  Any s = insert(str);
7  String str_ex = extract_DevString(s);
8
9  int[] array = new int[2];
```

```
10  array[0] = 1;
11  array[1] = 2;
12  Any arr = insert(array);
13  int[] array_ext = extract_DevVarLongArray(arr);
```

Line 1-3 : Insertion/Extraction of DevLong type
Line 5-7 : Insertion/Extraction of DevString type
Line 9-13 : Insertion/Extraction of DevVarLongArray type

5.3 C++ memory management

The rule described here are valid for variable length command data types like Tango::DevString or all the Tango:: DevVarxxxxArray types.

The method executing the command must allocate the memory used to pass data back to the client or use static memory (like buffer declares as object data member. If necessary, the ORB will deallocate this memory after the data have been sent to the caller. Fortunately, for incoming data, the method have no memory management responsabilities. The details about memory management given in this chapter assume that the insert/extract methods of the Tango::Command class are used and only the method in the device object is discussed.

5.3.1 For string

Example of a method receiving a Tango::DevString and returning a Tango::DevString is detailed just below

---

```
1  Tango::DevString MyDev::dev_string(Tango::DevString argin)
2  {
3          Tango::DevString        argout;
4
5          cout << "the received string is " << argin << endl;
6
7          string str("Am I a good Tango dancer ?");
8          argout = new char[str.size() + 1];
9          strcpy(argout,str.c_str());
10
11         return argout;
12 }
```

Note that there is no need to deallocate the memory used by the incoming string. Memory for the outgoing string is allocated at line 8, then it is initialised at the following line. The memory allocated at line 8 will be automatically freed by the usage of the *Command::insert()* method. Using this schema, memory is allocated/freed each time the command is executed. For constant string length, a statically allocated buffer can be used.

```
1  Tango::ConstDevString MyDev::dev_string(Tango::DevString argin)
2  {
3          Tango::ConstDevString   argout;
```

```
4
5           cout << "the received string is " << argin << endl;
6
7           argout = "Hello world";
8           return argout;
9   }
```

A Tango::ConstDevString data type is used. It is not a new data Tango data type. It has been introduced only to allows *Command::insert()* mehod overloading. The argout pointer is intialised at line 7 with memory statically allocated. In this case, no memory will be freed by the *Command::insert()* method. There is also no memory copy in the contrary of the previous example. A buffer defined as object data member can also be used to set the argout pointer.

5.3.2 For array/sequence

Example of a method returning a Tango::DevVarLongArray is detailed just below

```
1   Tango::DevVarLongArray *MyDev::dev_array()
2   {
3           Tango::DevVarLongArray  *argout  = new Tango::DevVarLongArray();
4
5           long output_array_length = ...;
6           argout->length(output_array_length);
7           for (int i = 0;i < output_array_length;i++)
8                   (*argout)[i] = i;
9
10          return argout;
11  }
```

In this case, memory is allocated at line 3 and 6. Then, the sequence is populated. The sequence is created and returned using pointer. The *Command::insert()* method will insert the sequence into the CORBA::Any object using this pointer. Therefore, the CORBA::Any object will take ownership of the allocated memory. It will free it when it will be destroyed by the CORBA ORB after the data have been sent away. It is also possible to use a statically allocated memory and to avoid copying in the sequence used to returned the data. This is explained in the following example assuming a buffer of long data is declared as device data member and named buffer.

```
1   Tango::DevVarLongArray *MyDev::dev_array()
2   {
3           Tango::DevVarLongArray  *argout;
4
5           long output_array_length = ...;
6           argout = create_DevVarLongArray(buffer,output_array_length);
7           return argout;
8   }
```

At line 3 only a pointer to a DevVarLongArray is defined. This pointer is set at line 6 using the *create_DevVarLongArray()* method. This method will create a sequence using this buffer without memory allocation nor copying. The *Command::insert()* method used here is the same than the one used in the previous example. The sequence is created in a way that the destruction of the CORBA::Any object in which the sequence will be inserted will not destroy the buffer. The following create_xxx methods are defined in the DeviceImpl class :

Method name	data type
create_DevVarCharArray()	unsigned char
create_DevVarShortArray()	short
create_DevVarLongArray()	long
create_DevVarFloatArray()	float
create_DevVarDoubleArray()	double
create_DevVarUShortArray()	unsigned short
create_DevVarULongArray()	unsigned long

5.3.3 For string array/sequence

Example of a method returning a Tango::DevVarStringArray is detailed just below

```
1  Tango::DevVarStringArray *MyDev::dev_str_array()
2  {
3          Tango::DevVarStringArray *argout  = new Tango::DevVarStringArray();
4
5          argout->length(3);
6          (*argout)[0] = CORBA::string_dup("Rumba");
7          (*argout)[1] = CORBA::string_dup("Waltz");
8          string str("Jerck");
9          (*argout)[2] = CORBA::string_dup(str.c_str());
10         return argout;
11 }
```

Memory is allocated at line 3 and 5. Then, the sequence is populated at lines 6,7 and 9. The usage of the *CORBA::string_dup* function also allocates memory. The sequence is created and returned using pointer. The *Command::insert()* method will insert the sequence into the CORBA::Any object using this pointer. Therefore, the CORBA::Any object will take ownership of the allocated memory. It will free it when it will be destroyed by the CORBA ORB after the data have been sent away. For portability reason, the ORB uses the *CORBA::string_free* function to free the memory allocated for each string. This is why the corresponding *CORBA::string_dup* or *CORBA::string_alloc* function must be used to reserve this memory.It is also possible to use a statically allocated memory and to avoid copying in the sequence used to returned the data. This is explained in the following example assuming a buffer of pointer to char is declared as device data member and named int_buffer.

```
1  Tango::DevVarStringArray *DocDs::dev_str_array()
2  {
3          int_buffer[0] = "first";
4          int_buffer[1] = "second";
```

```
5
6           Tango::DevVarStringArray *argout;
7           argout = create_DevVarStringArray(int_buffer,2);
8           return argout;
9   }
```

The intermediate buffer is initialised with statically allocated memory at lines 3 and 4. The returned sequence is created at line 7 with the *create_DevVarStringArray()* method. Like for classical array, the sequence is created in a way that the destruction of the CORBA::Any object in which the sequence will be inserted will not destroy the buffer.

5.3.4 For Tango composed types

Tango supports only two composed types which are Tango::DevVarLongStringArray and Tango::DevVarDoubleStringA These types are translated to C++ structure with two sequences. It is not possible to use memory statically allocated for these types. Each structure element must be intialised as described in the previous sub-chapters using the dynamically allocated memory case.

5.4 Reporting errors

Tango uses the C++ and Java try/catch plus exception mechanism to report errors. Two kind of errors can be transmitted between client and server :

1. CORBA system error. These exceptions are raised by the ORB and indicates major failures (A communication failure, An invalid object reference...)

2. CORBA user exception. These kind of exceptions are defined in the IDL file. This allows an exception to contain an arbitrary amount of error information of arbitrary type.

TANGO defines one user exception called **DevFailed**. This exception is a variable length array of **DevError** type (a sequence of DevError). The DevError type is a four fields structure. These fields are :

1. A string describing the type of the error. This string replaces an error code and allows a more easy management of include files.

2. The error severity. It is an enumeration with the three values which are WARN, ERR or PANIC.

3. A string describing in plain text the reason of the error

4. A string describing the origin of the error

The Tango::DevFailed type is a sequence of DevError structures in order to transmit to the client what is the primary error reason when several classes are used within a command. The sequence element 0 must be the DevError structure describing the primary error. A method called *print_exception*() defined in the Tango::Except class prints the content of exception (CORBA system exception or Tango::DevFailed exception). Some static methods of the Tango::Except class called *throw_exception*() can be used to throw Tango::DevFailed exception. With Java, these functions are static methods of the Except class. Details on these methods can be found in [6].

5.4.1 Example of throwing exception using C++

This example is a piece of code from the *command_handler*() method of the DeviceImpl class. An exception is thrown to the client to indicate that the requested command is not defined in the command list.

```
1          TangoSys_OMemStream o;
2
3          o << "Command " << command << " not found" << ends;
4          Except::throw_exception((const char *)"API_CommandNotFound",
5                                  o.str(),
6                                  (const char *)"DeviceClass::command_handler");
```

Line 1 : Build a memory stream. Use the TangoSys_MemStream because memory streams are not managed the same way between Windows and Unix

Line 3 : Build the reason string in the memory stream

Line 4-5 : Throw the exception to client using one of the *throw_exception* static method of the Except class. This throw_exception method used here allows the definition of the error type string, the reason string and the origin string of the DevError structure. The remaining DevError field (the error severity) will be set to its default value. Note that the first and third parameters are casted to a *const char **. Standard C++ defines that such a string is already a *const char ** but the GNU C++ compiler (release 2.95) does not use this type inside its function overloding but rather uses a *char ** which leads to calling the wrong function.

5.4.2 Example of throwing exception using Java

This example is a fragment of code from the *command_handler*() method of the DeviceImpl class. An exception is thrown to the client to indicate that the requested command is not defined in the command list.

```
1  StringBuffer o = new StringBuffer("Command ");
2  o.append(command);
3  o.append(" not found");
4
5  Except.throw_exception("API_CommandNotFound",
6                         o.toString(),
7                         "DeviceClass.command_handler");
```

Line 1-3 : Build a string with a message describing the error. The StringBuffer class is used instead of the String class because the StringBuffer class allows dynamic resizing of the string.

Line 5-7 : Throw the exception to client using the static *throw_exception* method of the Except class. The *throw_exception* method used here allows the definition of the reason string, the description string and the origin string of the DevError structure. The remaining DevError field (the error severity) will be set to its default value.

Note that the CORBA system exception inherits from the java.lang.RuntimeException. Exception derivated from this class do not need to be catched or re-thrown. This is the case for the BAD_OPERATION exception thrown when a mismatched type is used to extract data from an Any object. CORBA user exception (like the DevFailed exception) inherits from the java.Exception class and needs to be catched or re-thrown.

Chapter 6

Writing a device server

Writing a device server can be made easier by adopting the correct approach. This chapter will describe how to write a device server. It is divided into the following parts : understanding the device, defining device commands, chosing device state and writing the necessary classes. All along this chapter, examples will be given using the stepper motor device server. Writing a device server for our stepper motor example device means writing :

- The *main* function
- The *class_factory* method (only for C++ device server)
- The *StepperMotorClass* class
- The *DevReadPositionCmd* class
- The *StepperMotor* class.

All these functions and classes will be detailed. The stepper motor device server described in this chapter supports 2 commands and 3 attributes which are :

- Command DevReadPosition implemented using the inheritance model
- Command DevReadDirection implemented using the template command model
- Attribute Position (position of the first motor). This attribute is readable and is linked with a writable attribuute (called SetPosition). When the value of this attribute is requested by the client, the value of the associated writable attribute is also returned.
- Attribute SetPosition (writable attribute linked with the Position attribute)
- Attribute Direction (direction of the first motor)

In order to also gives an example of how the database objects part of the Tango device pattern could be used, our device have two properties. These properties are of the Tango long data types and are names “Max” and “Min”.

6.1 Understanding the device

The first step before writing a device server is to develop an understanding of the hardware to be programmed. The Equipment Responsible should have description of the hardware and its operating modes (manuals, spec sheets etc.). The Equipment Responsible must also provide specifications of what the device server should do. The Device Server Programmer should demand an exact description of the registers, alarms, interlocks and any timing constraints which have

to be kept. It is very important to have a good understanding of the device interfacing before starting designing a new class.

Once the Device Server Programmer has understood the hardware the next important step is to define what is a logical device i.e. what part of the hardware will be abstracted out and treated as a logical device. In doing so the following points of the TDSOM should be kept in mind

- Each device is known and accessed by its ascii name.
- The device is exported onto the network to be imported by applications.
- Each device belongs to a class.
- A list of commands exists per device.
- Applications use the device server api to execute commands on a device.

The above points have to be taken into account when designing the level of device abstraction. The definition of what is a device for a certain hardware is primarily the job of the Device Server Programmer and the Applications Programmer but can also involve the Equipment Responsible. The Device Server Programmer should make sure that the Applications Programmer agrees with her definition of what is a device.

Here are some guidelines to follow while defining the level of device abstraction -

- **efficiency**, make sure that not a too fine level of device abstraction has been chosen. If possible group as many attributes together to form a device. Discuss this with the Applications Programmer to find out what is efficient for her application.
- **hardware independency**, one of the main reasons for writing device servers is to provide the Applications Programmer with a *software* interface as opposed to a *hardware* interface. Hide the hardware structure of the device. For example if the user is only interested in a single channel of a multichannel device then define each channel to be a logical device. The user should not be aware of hardware addresses or cabling details. The user is very often a scientist who has a physics-oriented worldview and not a hardware-oriented worldview. Hardware independency also has the advantage that applications are immune to hardware changes to the device
- **object oriented worldview**, another *raison d'etre* behind the device server model is to build up an object oriented view of the world. The device should resemble the user's view of the object as closely as possible. In the case of the ESRF's beam lines for example, the devices should resemble beam line scientist's view of the machine.
- **atomism**, each device can be considered like an atom - is a independent object. It should appear independent to the client even if behind the scenes it shares some hardware or software with other objects. This is often the case with multichannel devices where the user would like to see each channel as a device but it is obvious that the channels cannot be programmed completely independently. The logical device is there to hide or make transparent this fact. If it is impossible to send commands to one device without modifying another device then a single device should be made out the two devices.
- **tailored** *vs* **general**, one of the philosophies of the TDSOM is to provide tailored solutions. For example instead of writing one *serial line* class which treats the general case of a serial line device and leaving the device protocol to be implemented in the client the TDSOM advocates implementing a device class which handles the protocol of the device. This way the client only has to know the commands of the class and not the details of the protocol. Nothing prevents the device class from using a general purpose serial line class if it exists of course.

6.2 Defining device commands

Each device has a list of commands which can be executed by the application across the network or locally. These commands are the Application Programmer's network knobs and dials for interacting with the device.

The list of commands to be implemented depends on the capabilities of the hardware, the list of sensible functions which can be executed at a distance and of course the functionality required by the application. This implies a close collaboration between the Equipment Responsible, Device Server Programmer and the Application Programmer.

When drawing up the list of commands particular attention should be paid to the following points

- **performance**, no single command should monopolise the device server for a long time (a nominal value for long is one second). Commands should be implemented in such a way that it executes immediately returning with a response. At best try to keep command execution time down to less than the typical overhead of an rpc call i.e. som milliseconds. This of course is not always possible e.g. a serial line device could require 100 milliseconds of protocol exchange. The Device Server Programmer should find the best trade-off between the users requirements and the devices capabilities. If a command implies a sequence of events which could last for a long time then implement the sequence of events in another thread - don't block the device server.
- **robustness**, should be provided which allow the client to recover from error conditions and or do a warm startup.

6.2.1 Standard commands

A minimum set of two commands exist for all devices. These commands are

- DevState which returns the state of a device
- DevStatus which returns the status of the device as a formatted ascii string

These commands have already been discussed in 4.5

6.3 Choosing device state

The device state is a number which reflects the avaibility of the device. To simplify the coding for generic application, a predefined set of states are supported by TANGO. This list has 14 members which are

State name
ON
OFF
CLOSE
OPEN
INSERT
EXTRACT
MOVING
STANDBY
FAULT
INIT
RUNNING
ALARM
DISABLE
UNKNOWN

The names used here have obvious meaning.

6.4 Device server utilities to ease coding/debugging

The device server framework supports two set of utilities to ease the process of coding and debugging device server code. These utilities are :

1. The device server verbose option
2. The device server output redirection system

Using these two facilities avoids the usage of the classical "#ifdef DEBUG" style which makes code less readable.

6.4.1 The device server verbose option

Each device server supports a verbose option called **-v**. Four verbose levels are defined from 1 to 4. Level 4 is the most talkative one. If you use the -v option without specifying level, level 4 will be assumed. A device server started with output level n will print all the message of level between 1 and n. For instance, if you start a device server using -v3 option, only the output for level 1,2 and 3 will be displayed. Output for level 4 will not be printed. If you don't used the -v option, the output level is set to 0. By convention, level 3 and 4 are reserved for print message embedded into the Tango library. Level 1 and 2 are free for the user.

Choosing the output level using C++

With C++, four macros have been defined. These macros are called *cout1*, *cout2*, *cout3* and *cout4*. The first one (*cout1*) defines a message which should be printed only when output level 1 or more is requested. The second one (*cout2*) defines a message which should be printed only when output level 2 or more is requested. The same philosophy is used for *cout3* and *cout4*. The usage of these *coutx* macros is the same than the classical cout.

```
1          cout3 << "What a nice dance" << endl;
2          cout3 << "What's its name ?" << endl;
3
4          cout << "Its name is TANGO" << endl;
```

Line 1-2 : The two questions are level 3 messages.
Line 4 : This print will be printed whatever the print level is.
If this piece of code is part of a device server started with a -v2 option, only the message defined line 4 will be displayed. If the device server is started with a -v3, -v4 or -v option, the two messages defined at lines 1 and 2 will also be displayed.

Choosing the output level using Java

With Java, four static objects inside the Util class have been defined. These objects are called *out1*, *out2*, *out3* and *out4*. These four objects support the *println* method exactly as the *out* object inside the System class does. The first object (*out1*) defines a message which should be printed only when output level 1 or more is requested. The second one (*out2*) defines a message which should be printed only when output level 2 or more is requested. The same philosophy is used for *out3* and *out4*. The usage of these *outx* objects is the same than the classical *out*.

```
1          Util.out3.println("What a nice dance");
2          Util.out3.println("What's its name ?");
3
4          System.out.println("Its name is TANGO");
```

Line 1-2 : The two questions are level 3 messages.
Line 4 : This print will be printed whatever the print level is.
If this piece of code is part of a device server started with a -v2 option, only the message defined line 4 will be displayed. If the device server is started with a -v3, -v4 or -v option, the two messages defined at lines 1 and 2 will also be displayed.

Changing the output level at run time

It is possible to change the output level at run time. You do so using commands of the dserver device. These two commands are :

- DevSetTraceLevel. This command needs the new trace level as input parameter. Using this command superseeds the level requested at device server process command line
- DevGetTraceLevel. This command returns the actual trace level.

6.4.2 Device server output redirection

Two commands of the dserver device allow device server output redirection. Theses two commands are :

- DevSetTraceOutput. This command sets all the device server output used to print message to be redirected to a file. This command needs the complete file path as input parameter. The file is local to the computer where the device server process is running.
- DevGetTraceOutput. This command returns the name of the file used to redirect device server process output. If no DevSetTraceOutput command has been used prior to the execution of this command, it returns a special string ("Initial Output") to indicates that the output is still the output defines at process startup.

6.4.3 Usage example

These two previously described features can ease device server debugging. Suppose a device server process is started with the following command line (UNIX command line)

Perkin id22 >/dev/null

This command line does not define any output level. Therefore the default output level is chosen (0) and no message are printed. Sending a DevSetTraceLevel command requesting level 4 and a DevSetTraceOutput command with a file name /tmp/server.out will make the device server sending all the output to the /tmp/server.out file **without stopping the process**. The inspection of the /tmp/server.out file will hopefully help to find the reason of the device server problem. When the output are not needed anymore, sending a DevSetTraceOutput command with the input parameter set to "Initial Output" followed by a DevSetTracelevel command with a requested level of 0 will return the server to its original state.

6.5 Avoiding name conflicts

6.5.1 Using C++

Namespace are used to avoid name conflicts. Each device pattern implementation is defined within its own namespace. The name of the namespace is the device pattern class name. In our example, the namespace name is *StepperMotor*.

6.5.2 Using Java

Package are used to avoid name conflicts. Each device pattern implementation is defined within its own package. The name of the package is the device pattern class name. In our example, the package name is *StepperMotor*.

6.6 The device server main function

A device server main function (or method) always follows the same framework. It exactly implements all the action described in chapter 4.7.5. Even if it could be always the same, it has not been included in the library because some linkers are perturbed by the presence of two main functions.

6.6.1 Using C++

```
 1  #include <tango.h>
 2
 3  int main(int argc,char *argv[])
 4  {
 5
 6          try
 7          {
 8
 9                  Tango::Util *tg = Tango::Util::init(argc,argv);
10
11                  tg->server_init();
12
13                  cout << "Ready to accept request" << endl;
14                  tg->server_run();
15          }
16          catch (bad_alloc)
17          {
18                  cout << "Can't allocate memory!!!" << endl;
19                  cout << "Exiting" << endl;
20          }
21          catch (CORBA::Exception &e)
22          {
23                  Tango::Except::print_exception(e);
24
25                  cout << "Received a CORBA::Exception" << endl;
26                  cout << "Exiting" << endl;
27          }
28
29          return(0);
```

```
30  }
```

Line 1 : Include the **tango.h** file. This file is a master include file. It includes several other files. The list of files included by tango.h can be found in [6]

Line 9 : Create the instance of the Tango::Util class (a singleton). Passing argc,argv to this method is mandatory because the device server command line is checked when the Tango::Util object is constructed.

Line 11 : Start all the device pattern creation and initialisation with the *server_init()* method

Line 14 : Put the server in a endless waiting loop with the *server_run()* method. In normal case, the process should never returns from this line.

Line 16-20 : Catch all exceptions due to memory allocation error, display a message to the user and exit

Line 21 : Catch all standard TANGO exception which could occur during device pattern creation and intialisation

Line 23 : Print exception parameters

Line 25-26 : Print an additional message

6.6.2 Using Java

The *main* method can be defined in any class. There is no mandatory class where it should be defined. In our StepperMotor example, the *main* method has been implemented in the StepperMotor class because it is the most logical place.

```
 1  package StepperMotor
 2
 3  import java.util.*;
 4  import org.omg.CORBA.*;
 5  import fr.esrf.Tango.*;
 6  import fr.esrf.TangoDs.*;
 7
 8  public class StepperMotor extends DeviceImpl implements TangoConst
 9  {
10          public static void main(String[] argv)
11          {
12                  try
13                  {
14
15                          Util tg = Util.init(argv,"StepperMotor");
16
17                          tg.server_init();
18
19                          System.out.println("Ready to accept request");
20
21                          tg.server_run();
22                  }
23                  catch (OutOfMemoryError ex)
24                  {
25                          System.err.println("Can't allocate memory !!!!");
26                          System.err.println("Exiting");
27                  }
```

```
28                  catch (UserException ex)
29                  {
30                          Except.print_exception(ex);
31
32                          System.err.println("Received a CORBA user exception");
33                          System.err.println("Exiting");
34                  }
35                  catch (SystemException ex)
36                  {
37                          Except.print_exception(ex);
38
39                          System.err.println("Received a CORBA system exception");
40                          System.err.println("Exiting");
41                  }
42
43                  System.exit(-1);
44
45          }
46  }
```

---

line 1 : The StepperMotor class is part of the StepperMotor package

Line 3-6 : Import several packages. The reason of importing these package will be explained when the StepperMotor class will be detailed later in this chapter

Line 8 : Definition of the StepperMotor class (will be explained later)

Line 10 : Definition of the *main* method

Line 15 : Create the instance of the Util class (a singleton). Passing argv to this method is mandatory because the device server command line is checked when the Util object is constructed. The second argument of this *init* method is the device server executable name as defined in 4.7.1

Line 17 : Start all the device pattern creation and initialisation

Line 21 : Put the server in a endless waiting loop. In normal case, the process should never returns from this line.

Line 23-27 : Catch all exceptions due to memory error and display a message to the user. It seems strange to deal with memory allocation error with Java.The Java garbage collection system reclaims memory only for object which have a reference count equal to zero. If, inside a program, objects are created and stay with an object refence count different than zero, they will never be destructed. If many of these objects are created, memory allocation errors can occurs. You may think that the author of this manual is paranoid but have a look at [11]

Line 28-34 : Catch CORBA user exception included the TANGO DevFailed exception which could occur during device pattern creation and intialisation

Line 30 : Use the static *print_exception* method of the Except class to print all the data members of the exception object.

Line 35-41 : catch CORBA system exception.

Line 37 : Use the static *print_exception* method of the Except class to print all the data members of the exception object.

Line 43 : Exit the device server

6.7 The DServer::class_factory method (C++ specific)

As described in chapter 4.7.2, C++ device server needs a *class_factory*() method. This method creates all the device pattern implemented in the device server by calling their *init*() method. The following is an example of a *class_factory* method for a device server with one implementation of the device server pattern for stepper motor device.

```
1  #include <tango.h>
2  #include <steppermotorclass.h>
3
4  void Tango::DServer::class_factory()
5  {
6
7          add_class(StepperMotor::StepperMotorClass::init("StepperMotor"));
8
9  }
```

Line 1 : Include the Tango master include file
Line 2 : Include the steppermotorclass class definition file
Line 7 : Create the StepperMotorClass singleton by calling its *init* method and stores the returned pointer into the DServer object. Remember that all classes for the device pattern implementation for the stepper motor class is defined within a namespace called *StepperMotor*.

6.8 Writing the StepperMotorClass class

6.8.1 Using C++

The class definition file

```
1  #include <tango.h>
2
3  namespace StepperMotor
4  {
5
6  class StepperMotorClass : public Tango::DeviceClass
7  {
8  public:
9          static StepperMotorClass *init(const char *);
10         static StepperMotorClass *instance();
11         ~StepperMotorClass() {_instance = NULL;}
12
13 protected:
14         StepperMotorClass(string &);
15         static StepperMotorClass *_instance;
16         void command_factory();
17         void attribute_factory(vector<Tango::Attr *> &);
18
19 public:
20         void device_factory(const Tango::DevVarStringArray *);
21 };
22
23 } /* End of StepperMotor namespace */
```

Line 1 : Include the Tango master include file

Line 3 : This class is defined within the *StepperMotor* namespace

Line 6 : Class StepperMotorClass inherits from Tango::DeviceClass

Line 9-10 : Definition of the *init* and *instance* methods. These methods are static and can be called even if the object is not already constructed.

Line 11: The desctructor

Line 14 : The class constructor. It is protected and can't be called from outside the class. Only the *init* method allows a user to create an instance of this class. See [8] to get details about the singleton design pattern.

Line 15 : The instance pointer. It is static in order to set it to NULL during process initialisation phase

Line 16 : Definition of the *command_factory* method

Line 17 : Definition of the *attribute_factory* method

Line 20 : Definition of the *device_factory* method

The singleton related methods

```
 1  #include <tango.h>
 2
 3  #include <steppermotor.h>
 4  #include <steppermotorclass.h>
 5
 6  namespace StepperMotor
 7  {
 8
 9  StepperMotorClass *StepperMotorClass::_instance = NULL;
10
11  StepperMotorClass::StepperMotorClass(string &s):
12  Tango::DeviceClass(s)
13  {
14          cout2 << "Entering StepperMotorClass constructor" << endl;
15
16          cout2 << "Leaving StepperMotorClass constructor" << endl;
17  }
18
19
20  StepperMotorClass *StepperMotorClass::init(const char *name)
21  {
22          if (_instance == NULL)
23          {
24                  try
25                  {
26                          string s(name);
27                          _instance = new StepperMotorClass(s);
28                  }
29                  catch (bad_alloc)
30                  {
31                          throw;
32                  }
33          }
34          return _instance;
```

```
35  }
36
37  StepperMotorClass *StepperMotorClass::instance()
38  {
39          if (_instance == NULL)
40          {
41                  cerr << "Class is not initialised !!" << endl;
42                  exit(-1);
43          }
44          return _instance;
45  }
```

Line 1-4 : include files: the Tango master include file (tango.h), the StepperMotorClass class definition file (steppermotorclass.h) and the StepperMotor class definition file (steppermotor.h)

Line 6 : Open the *StepperMotor* namespace.

Line 9 : Initialise the static _instance field of the StepperMotorClass class to NULL

Line 11-18 : The class constructor. It takes an input parameter which is the controlled device class name. This parameter is passed to the constructor of the DeviceClass class. Otherwise, the construtor does nothing except printing a message

Line 20-35 : The *init* method. This method needs an input parameter which is the controlled device class name (StepperMotor in this case). This method checks is the instance is already constructed by testing the _instance data member. If the instance is not constructed, it creates one. If the instance is already constructed, the method simply returns a pointer to it.

Line 37-45 : The *instance* method. This method is very similar to the *init* method except that if the instance is not already constructed. the method print a message and abort the process.

As you can understand, it is not possible to construct more than one instance of the StepperMotorClass (it is a singleton) and the *init* method must be called prior to any other method.

The command_factory method

Within our example, the stepper motor device supports two commands which are called DevReadPosition and DevReadDirection. These two command takes a Tango::DevLong argument as input and output parameter. The first command is created using the inheritance model and the second command is created using the template command model.

```
 1
 2  void StepperMotorClass::command_factory()
 3  {
 4          command_list.push_back(new DevReadPositionCmd("DevReadPosition",
 5                                                        Tango::DEV_LONG,
 6                                                        Tango::DEV_LONG,
 7                                                        "Motor number (0-7)",
 8                                                        "Motor position"));
 9
10          command_list.push_back(
11              new TemplCommandInOut<Tango::DevLong,Tango::DevLong>
12                  ((const char *)"DevReadDirection",
13                   static_cast<Tango::Lg_CmdMethPtr_Lg>
14                          (&StepperMotor::dev_read_direction),
15                   static_cast<Tango::StateMethPtr>
```

```
16                              (&StepperMotor::direct_cmd_allowed))
17                                    );
18  }
19
```

Line 4 : Creation of one instance of the DevReadPositionCmd class. The class is created with five arguments which are the command name, the command type code for its input and output parameters and two strings which are the command input and output parameters description. The pointer returned by the new C++ keyword is added to the vector of available command.

Line 10-14 : Creation of the object used for the DevReadDirection command. This command has one input and output parameter. Therefore the created object is an instance of the TemplCommandInOut class. This class is a C++ template class. The first template parameter is the command input parameter type, the second template parameter is the command output parameter type. The second TemplCommandInOut class constructor parameter (set at line 13) is a pointer to the method to be executed when the command is requested. A casting is necessary to store this pointer as a pointer to a method of the DeviceImpl class[1]. The third TemplCommandInOut class constructor parameter (set at line 15) is a pointer to the method to be executed to check if the command is allowed. This is necessary only if the default behaviour (command always allowed) does not fulfill the needs. A casting is necessary to store this pointer as a pointer to a method of the DeviceImpl class. When a command is created using the template command method, the input and output parameters type are determined from the template C++ class parameters.

The device_factory method

The *device_factory* method has one input parameter. It is a pointer to Tango::DevVarStringArray data which is the device name list for this class and the instance of the device server process. This list is fetch from the Tango database.

```
1  void StepperMotorClass::device_factory(const Tango::_DevVarStringArray *devlist_ptr)
2  {
3
4          for (long i = 0;i < devlist_ptr->length();i++)
5          {
6                  cout4 << "Device name : " << (*devlist_ptr)[i] << endl;
7
8                  device_list.push_back(new StepperMotor(this,
9                                                         (*devlist_ptr)[i]));
10
11                 export_device(device_list.back());
12         }
13 }
```

Line 4 : A loop for each device

Line 8 : Create the device object using a StepperMotor class constructor which needs two arguments. These two arguments are a pointer to the StepperMotorClass instance and the device name. The pointer to the constructed object is then added to the device list vector

Line 11 : Export device to the outside world using the *export_ device* method of the DeviceClass class.

[1]The StepperMotor clas inherits from the DeviceImpl class and therefore is a DeviceImpl

The attribute_factory method

The rule of this method is to fullfill a vector of pointer to attributes. A reference to this vector is passed as argument to this method.

```
1  void StepperMotorClass::attribute_factory(vector<Tango::Attr *> &att_list)
 2  {
 3          att_list.push_back(
 4                  new Tango::Attr("Position",
 5                                  Tango::DEV_LONG,
 6                                  Tango::READ_WITH_WRITE,
 7                                  "SetPosition"));
 8          att_list.push_back(
 9                  new Tango::Attr("SetPosition",
10                                  Tango::DEV_LONG,
11                                  Tango::WRITE));
12          att_list.push_back(
13                  new Tango::Attr("Direction",
14                                  Tango::DEV_LONG));
15  }
```

Line 3-7 : Build a one dimension attribute of Tango::DevLong type with an associate writable attribute. Store the pointer to this attribute object into the attribbute pointer vector.

Line 8-11 : Build a one dimension writable attribute. Store the pointer to this attribute object into the attribute pointer vector.

Line 12-14 : Build a one dimension attribute. Store the pointer to this attribute object into the attribbute pointer vector.

6.8.2 Using Java

The singleton related method

```
 1  package StepperMotor;
 2
 3  import java.util.*;
 4  import fr.esrf.Tango.*;
 5  import fr.esrf.TangoDs.*;
 6
 7  public class StepperMotorClass extends DeviceClass implements TangoConst
 8  {
 9          private static StepperMotorClass        _instance = null;
10
11
12          public static StepperMotorClass instance()
13          {
14                  if (_instance == null)
15                  {
16                          System.err.println("StepperMotorClass is not initialised !!!
17                          System.err.println("Exiting");
```

```
18                          System.exit(-1);
19                  }
20                  return _instance;
21          }
22
23
24          public static StepperMotorClass init(String class_name) throws DevFailed
25          {
26                  if (_instance == null)
27                  {
28                          _instance = new StepperMotorClass(class_name);
29                  }
30                  return _instance;
31          }
32
33          protected StepperMotorClass(String name) throws DevFailed
34          {
35                  super(name);
36
37                  Util.out2.println("Entering StepperMotorClass constructor");
38
39                  Util.out2.println("Leaving StepperMotorClass constructor");
40          }
41  }
```

Line 1 : This class is part of the StepperMotor package.

Line 3-5 : Import different packages. The first one (**java.lang.util**) is a classical Java package from the JDK. The second one (**fr.esrf.Tango**) is the package generated by the IDL compiler from the Tango IDL file. The last one (**fr.esrf.TangoDs**) is the name of the package with all the root classes of the device server framework.

Line 7 : The StepperMotorClass inherits from the DeviceClass and implements the **TangoConst** interface. The TangoConst interface does not defines any method but simply defines constant variables. The TangoConst interface is a member of the TangoDs package.

Line 9 : The instance pointer. It is static and private. It is intialised to NULL

Line 12-21 : The *instance* method. This method is very similar to the *init* method except that if the instance is not already constructed. the method print a message and abort the process.

Line 24-31: The *init* method. This method needs an input parameter which is the controlled device class name (StepperMotor in this case). This method checks is the instance is already constructed by testing the _instance data member. If the instance is not constructed, it creates one. If the instance is already constructed, the method simply returns a pointer to it.

Line 33-40 : The class constructor which is protected. It takes an input parameter which is the controlled device class name. This parameter is passed to the constructor of the DeviceClass class (line 35). Otherwise, the construtor does nothing except printing a message

As you can understand, it is not possible to construct more than one instance of the StepperMotorClass (it is a singleton) and the *init* method must be called prior to any other method.

The command_factory method

Within our example, the stepper motor device supports two commands which are called DevReadPosition and DevReadDirection. These two command takes a Tango_DevLong argument as input and output parameter. The first command is created using the inheritance model and the second command is created using the template command model.

```
1  public void command_factory()
2  {
3          String str = new String("DevReadPosition");
4          command_list.addElement(new DevReadPositionCmd(str,
5                                  Tango_DEV_LONG,Tango_DEV_LONG,
6                                  "Motor number (0-7)",
7                                  "Motor position"));
8
9          str =  new String("DevReadDirection");
10         command_list.addElement(new TemplCommandInOut(str,
11                                    "dev_read_direction",
12                                    "direct_cmd_allowed"));
13 }
```

Line 4: Creation of one instance of the DevReadPositionCmd class. The class is created with five arguments which are the command name, the command type code for its input and output parameters and the parameters description (input and output). The Tango_DEV_LONG constant is defined in the TangoConst interface. The reference returned by the *new* Java keyword is added to the vector of available command via the *addElement* method of the Java Vector class.

Line 10-12 : Creation of the object used for the DevReadDirection command. This command has one input and output parameter. Therefore the created object is an instance of the TemplCommandInOut class. The second TemplCommandInOut class constructor parameter (set at line 11) is the method name to be executed when the command is requested. The third TemplCommandInOut class constructor parameter (set at line 12) is the method name to be executed to check if the command is allowed. This is necessary only if the default behaviour (command always allowed) does not fulfill the needs. When a command is created using the template command method, the input and output parameter types are determined from the given method declaration.

The device_factory method

The *device_factory* method has one input parameter. It is a pointer to a DevVarStringArray[2]data which is the device name list for this class and the instance of the device server process. This list is fetch from the Tango database.

```
1  public void device_factory(String[] devlist) throws DevFailed
2  {
3          for (int i = 0;i < devlist.length;i++)
4          {
5                  Util.out4.println("Device name : " + devlist[i]);
6
7                  device_list.addElement(new StepperMotor(this,
8                                                          devlist[i],
9                                                          "A Tango motor",
10                                                         DevState.ON,
11                                                         "The motor is ON"));
12
13                 export_device(((DeviceImpl)(device_list.lastElement())));
14         }
15 }
```

[2]DevVarStringArray maps to Java String[]

Line 3 : A loop for each device

Line 7 : Create the device object using a StepperMotor class constructor which needs five arguments. These five arguments are a reference to the StepperMotorClass instance, the device name, the device description, the device original state and the device original status. The reference to the constructed object is then added to the device list vector with the *addElement* method of the java.util.Vector class.

Line 13 : Export device to the outside world using the *export_device* method of the DeviceClass class. The *lastElement* method of the java.util.Vector class returns a reference to an object of the java Object class. It must be casted before being passed to the *export_device* method

The attribute_factory method

The rule of this method is to fullfill a vector of references to attribute. A reference to this vector is passed to this method. The Tango core classes will use this vector to build all the attributes related objects (An instance of the MultiAttribute class and one Attribute or WAttribute object for each attribute defined in this vector).

```
1  public void attribute_factory(Vector att) throws DevFailed
2  {
3          att.addElement(new Attr("Position",
4                                  Tango_DEV_LONG,
5                                  AttrWriteType.READ_WITH_WRITE,
6                                  "SetPosition"));
7          att.addElement(new Attr("SetPosition",
8                                  Tango_DEV_LONG,
9                                  AttrWriteType.WRITE));
10         att.addElement(new Attr("Direction",
11                                 Tango_DEV_LONG));
12 }
```

Line 3-6 : Build a one dimension attribute of TANGO_DEV_LONG type with an associate writable attribute. Store a reference to this attribute in the vector.

Line 7-9 : Build a one dimension writable attribute and store a reference to it in the vector

Line 10-11 : Build a one dimension attribute and store a reference to it in the vector

6.9 The DevReadPositionCmd class

6.9.1 Using C++

The class definition file

```
1  #include <tango.h>
2
3  namespace StepperMotor
4  {
5
6  class DevReadPositionCmd : public Tango::Command
```

```
 7  {
 8  public:
 9          DevReadPositionCmd(const char *,Tango::CmdArgType,
10                             Tango::CmdArgType,
11                             const char *,const char *);
12          ~DevReadPositionCmd() {};
13
14          virtual bool is_allowed (Tango::DeviceImpl *, const CORBA::Any &);
15          virtual CORBA::Any *execute (Tango::DeviceImpl *, const CORBA::Any &);
16  };
17
18  } /* End of StepperMotor namespace */
```

Line 1 : Include the tango master include file
Line 3 : Open the *StepperMotor* namespace.
Line 6 : The DevReadPositionCmd class inherits from the Tango::Command class
Line 9 : The constructor
Line 12 : The destructor
Line 14 : The definition of the *is_allowed* method. This method is not necessary if the default behaviour implemented by the default *is_allowed* method fullfil the requirements. The default behaviour is to always allows the command execution (always return true).
Line 15: The definition of the *execute* method

The class constructor

The class constructor does nothing. It simply invoke the Command constructor by passing it its five arguments which are:

1. The command name
2. The command input type code
3. The command output type code
4. The command input parameter description
5. The command output parameter description

If the command does not have input or output parameter, it is not possible to use the Command class constructor defined with five parameters. In this case, the command constructor execute the Command class constructor with three elements (class name, input type, output type) and set the input or output parameter description fields with the *set_in_type_desc* or *set_out_type_desc* Command class methods.

The is_allowed method

In our example, the DevReadPosition command is allowed only if the device is in the ON state. This method receives two argument which are a pointer to the device object on which the command must be excuted and a reference to the command input Any object. This method returns a boolean which must be set to true if the command is allowed. If this boolean is set to false, the DeviceClass *command_handler* method will automatically send an exception to the caller.

```
1  bool DevReadPositionCmd::is_allowed(Tango::DeviceImpl *device,
2                                      const CORBA::Any &in_any)
3  {
4          if (device->get_state() == Tango::ON)
5                  return true;
6          else
7                  return false;
8  }
```

Line 4 : Call the *get_state* method of the DeviceImpl class which simply returns the device state

Line 5 : Authorise command if the device state is ON

Line 7 : Refuse command execution in all other cases.

The execute method

This method receives two arguments which are a pointer to the device object on which the command must be executed and a reference to the command input Any object. This method returns a pointer to an any object which must be initialised with the data to be returned to the caller.

```
1  CORBA::Any *DevReadPositionCmd::execute(
2                            Tango::DeviceImpl *device,
3                            const CORBA::Any &in_any)
4  {
5          cout2 << "DevReadPositionCmd::execute(): arrived" << endl;
6          Tango::DevLong motor;
7
8          extract(in_any,motor);
9          return insert(
10             (static_cast<StepperMotor *>(device))->dev_read_position(motor));
11 }
```

Line 8 : Extract incoming data from the input any object using a Command class *extract* hepler method. If the type of the data in the Any object is not a Tango::DevLong, the *extract* method will throw an exception to the client.

Line 9 : Call the stepper motor object method which execute the DevReadPosition command and insert the returned value into an allocated Any object. The Any object allocation is done by the *insert* method which return a pointer to this Any.

6.9.2 Using Java

The class constructor

The class constructor does nothing. It simply invoke the Command constructor by passing it its five arguments which are:

1. The command name
2. The command input type code
3. The command output type code

4. The command input parameter description

5. The command output parameter description

If the command does not have input or output parameter, it is not possible to use the Command class constructor defined with five parameters. In this case, the command constructor execute the Command class constructor with three elements (class name, input type, output type) and set the input or output parameter description fields with the *set_in_type_desc* or *set_out_type_desc* Command class methods.

The is_allowed method

In our example, the DevReadPosition command is allowed only if the device is in the ON state. This method receives two argument which are a reference to the device object on which the command must be excuted and a reference to the command input Any object. This method returns a boolean which must be set to true if the command is allowed. If this boolean is set to false, the DeviceClass *command_handle*r method will automatically send an exception to the caller.

```
 1  package StepperMotor;
 2
 3  import org.omg.CORBA.*;
 4  import fr.esrf.Tango.*;
 5  import fr.ersf.TangoDs.*;
 6
 7  public class DevReadPositionCmd extends Command implements TangoConst
 8  {
 9          public  boolean is_allowed(DeviceImpl dev, Any data_in)
10          {
11                  if (dev.get_state() == DevState.ON)
12                          return(true);
13                  else
14                          return(false);
15          }
16
17  }
```

Line 1 : This class is part of the StepperMotor package

Line 3-5 : Import different packages. The first one (**org.omg.CORBA**) is a package which contains all the CORBA related classes. The second one (**fr.esrf.Tango**) is the package generated by the IDL compiler from the Tango IDL file. The last one (**fr.ersf.TangoDs**) is the name of the package with all the root classes of the device server pattern.

Line 7 : The DevReadPositionCmd class inherits from the Command class and implements the **TangoConst** interface. The TangoConst interface does not defines any method but simply defines constant variables. The TangoConst interface is a member of the TangoDs package.

Line 11 : Call the *get_state* method of the DeviceImpl class which simply returns a reference to the device state

Line 12 : Authorise command if the device state is ON

Line 14 : Refuse command execution in all other cases.

The execute method

This method receives two arguments which are a reference to the device object on which the command must be executed and a reference to the command input Any object. This method returns a reference to an any object which must be initialised with the data to be returned to the caller.

```
1  public Any execute(DeviceImpl device,Any in_any) throws DevFailed
2  {
3          Util.out2.println("DevReadPositionCmd.execute(): arrived");
4
5          int motor = extract_DevLong(in_any);
6
7          return insert(((StepperMotor)(device)).dev_read_position(motor));
8  }
```

Line 5 : Extract incoming data from the input any object
Line 7 : Call the stepper motor object method which execute the DevReadPosition command, insert its return value into an any and return.

6.10 The StepperMotor class

6.10.1 Using C++

The class definition file

```
1  #include <tango.h>
2
3  #define AGSM_MAX_MOTORS 8 // maximum number of motors per device
4
5  namespace StepperMotor
6  {
7
8  class StepperMotor: public Tango::DeviceImpl
9  {
10 public :
11         StepperMotor(Tango::DeviceClass *,string &);
12         StepperMotor(Tango::DeviceClass *,const char *);
13         StepperMotor(Tango::DeviceClass *,const char *,const char *);
14         ~StepperMotor() {};
15
16         long dev_read_position(long);
17         long dev_read_direction(long);
18         bool direct_cmd_allowed(const CORBA::Any &);
19
20         virtual Tango::DevState dev_state();
21         virtual Tango::ConstDevString dev_status();
22
23         virtual void always_executed_hook();
```

```
24
25          virtual void read_attr_hardware(vector<long> &attr_list);
26          virtual void write_attr_hardware(vector<long> &attr_list);
27          virtual void read_attr(Tango::Attribute &attr);
28
29          virtual void init_device();
30
31          void get_device_properties();
32
33  protected :
34          long axis[AGSM_MAX_MOTORS];
35          long position[AGSM_MAX_MOTORS];
36          long direction[AGSM_MAX_MOTORS];
37          long state[AGSM_MAX_MOTORS];
38
39          Tango::DevLong  *attr_Position_read;
40          Tango::DevLong  *attr_Direction_read;
41          Tango::DevLong  attr_SetPosition_write;
42
43          Tango::DevLong  min;
44          Tango::DevLong  max;
45  };
46
47  } /* End of StepperMotor namespace */
```

Line 1 : Include the Tango master include file
Line 5 : Open the *StepperMotor* namespace.
Line 8 : The StepperMotor class inherits from the DeviceImpl class
Line 11-13 : Three different object constructors
Line 14 : The destructor
Line 16 : The method to be called for the execution of the DevReadPosition command. This method must be declared as virtual if it is needed to redefine it in a class inheriting from StepperMotor. See chapter 9.2 for more details about inheriting.
Line 17 : The method to be called for the execution of the DevReadDirection command
Line 18 : The method called to check if the execution of the DevReadDirection command is allowed. This method is necessary because the DevReadDirection command is created using the template command method and the default behaviour is not acceptable
Line 20 : Redefinition of the *dev_state*. This method is used by the DevState command
Line 21 : Redefinition of the *dev_status*. This method is used by the DevStatus command
Line 23 : Redefinition of the *always_executed_hook* method. This method is the place to code mandatory action which must be executed prior to any command.
Line 25-27 : Attribute related methods
Line 29 : Definition of the *init_device* method.
Line 31 : Definition of the *get_device_properties* method
Line 34-44 : Data members.
Line 39-40 : Pointers to data for readable attributes Position and Direction
Line 41 : Data for the SetPosition attribute
Line 43-44 : Data members for the two device properties

The constructors

Three constructors are defined here. It is not mandatory to defined three constructors. But at least one is mandatory. The three constructors take a pointer to the StepperMotorClass instance

as first parameter[3]. The second parameter is the device name as a C++ string or as a classical pointer to char array. The third parameter necessary only for the third form of constructor is the device description string passed as a classical pointer to a char array.

```
 1  #include <tango.h>
 2  #include <steppermotor.h>
 3
 4  namespace StepperMotor
 5  {
 6
 7  StepperMotor::StepperMotor(Tango::DeviceClass *cl,string &s)
 8  :Tango::DeviceImpl(cl,s.c_str())
 9  {
10          init_device();
11  }
12
13  StepperMotor::StepperMotor(Tango::DeviceClass *cl,const char *s)
14  :Tango::DeviceImpl(cl,s)
15  {
16          init_device();
17  }
18
19  StepperMotor::StepperMotor(Tango::DeviceClass *cl,const char *s,const char *d)
20  :Tango::DeviceImpl(cl,s,d)
21  {
22          init_device();
23  }
24
25  void StepperMotor::init_device()
26  {
27          cout << "StepperMotor::StepperMotor() create " << device_name << endl;
28
29          long i;
30
31          for (i=0; i< AGSM_MAX_MOTORS; i++)
32          {
33                  axis[i] = 0;
34                  position[i] = 0;
35                  direction[i] = 0;
36          }
37
38          get_device_properties();
39  }
```

Line 1-2 : Include the Tango master include file (tango.h) and the StepperMotor class definition file (steppermotor.h)

Line 4 : Open the *StepperMotor* namespace

Line 7-11 : The first form of the class constructor. It execute the Tango::DeviceImpl class constructor with the two parameters. Note that the device name passed to this constructor as

[3]The StepperMotorClass inherits from the DeviceClass and therefore is a DeviceClass

a C++ string is passed to the Tango::DeviceImpl constructor as a classical C string. Then the *init_device* method is executed.

Line 13-17 : The second form of the class constructor. It execute the Tango::DeviceImpl class constructor with its two parameters. Then the *init_device* method is executed.

Line 19-23: The third form of constructor. Again, it execute the Tango::DeviceImpl class constructor with its three parameters. Then the *init_device* method is executed.

Line 25-39 : The *init_device* method. All the device data initialisation is done in this method. The device properties are also retrieved from database with a call to the *get_device_properties* method at line 38.

The methods used for the DevReadDirection command

The DevReadDirection command is created using the template command method. Therefore, there is no specific class needed for this command but only one object of the TemplCommandInOut class. This command needs two methods which are the *dev_read_direction* method and the *direct_cmd_allowed* method. The *direct_cmd_allowed* method defines here implements exactly the same behaviour than the default one. This method has been used only for pedagogic issue. The *dev_read_direction* method will be executed by the *execute* method of the TemplCommandInOut class. The *direct_cmd_allowed* method will be executed by the *is_allowed* method of the TemplCommandInOut class.

```
 1  long StepperMotor::dev_read_direction(long axis)
 2  {
 3          if (axis < 0 || axis > AGSM_MAX_MOTORS)
 4          {
 5                  cout1 << "Steppermotor::dev_read_direction(): axis out of range !";
 6                  cout1 << endl;
 7                  TangoSys_OMemStream o;
 8
 9                  o << "Axis number " << axis << " out of range" << ends;
10                  throw_exception((const char *)"StepperMotor_OutOfRange",
11                                  o.str(),
12                                  (const char *)"StepperMotor::dev_read_direction");
13          }
14
15          return direction[axis];
16  }
17
18
19  bool StepperMotor::direct_cmd_allowed(const CORBA::Any &in_data)
20  {
21          cout2 << "In direct_cmd_allowed() method" << endl;
22
23          return true;
24  }
25
```

Line 1-16 : The *dev_read_direction* method

Line 5-12 : Throw exception to client if the received axis number is out of range

Line 7 : A TangoSys_OMemStream is used as stream. The TangoSys_OMemStream has been defined in improve portability across platform. For Unix like operating system, it is a ostrtream

type. For operating system with a full implementation of the standard library, it is a ostringstream type.

Line 19-24 : The *direct_cmd_allowed* method. The command input data is passed to this method in case of it is needed to take the decision. This data is still packed into the CORBA Any object.

The write attribute related method

To enable writing of writable attributes, the StepperMotor class must re-define a method called *write_attr_hardware()*. The aim of this method is to write the hardware. This method receives a vector of long as parameters. These long data are the indexes of the attributes to be written into the main attribute vector stored in the MultiAttribute object. Methods of the MultiAttribute class allows the retrieval of the the correct attribute object from these indexes. The value to be written is stored in the WAttribute object and can be retrieved with WAttribute class methods called *get_write_value()*. A data member called *attr_<Attribute_name>_write* is foreseen to temporary store this extracted value.

```
1  void StepperMotor::write_attr_hardware(vector<long> &attr_list)
2  {
3          cout2 << "In write_attr_hardware for " << attr_list.size();
4          cout2 << " attribute(s)" << endl;
5
6          for (long i = 0;i < attr_list.size();i++)
7          {
8                  Tango::WAttribute &att = dev_attr->get_w_attr_by_ind(attr_list[i]);
9                  string att_name = att.get_name();
10
11                 if (att_name == "SetPosition")
12                 {
13                         att.get_write_value(attr_SetPosition_write);
14                         cout2 << "Attribute SetPosition value = ";
15                         cout2 << attr_SetPosition_write << endl;
16                         position[0] = attr_SetPosition_write;
17                 }
18         }
19 }
```

Line 6 : A loop on each attribute to be written
Line 8-9: Retrieve attribute name
Line 11: A test on attribute name
Line 13 : Retrieve new attribute value
Line 16 : Set the hardware (very simple in our example)

The read attribute related methods

To enable reading of attributes, the StepperMotor class must re-define two methods called *read_attr_hardware()* and *read_attr()*. The aim of the first one is to read the harware. It will be called only once at the beginning of each read_attributes CORBA call. The second method aim is to build the exact data for the wanted attribute and to store this value into the Attribute object. This method will be called for each attribute to be read. Special care has been taken in order to minimize the number of data copy and allocation. The data passed to the Attribute object as attribute value is passed

using pointers. It must be allocated by the method[4] and the Attribute object will not free this memory. Data members called attr_<Attribute_name>_read are foreseen for this usage. As for the *write_attr_hardware()* method, the *read_attr_hardware()* method receives a vector of long which are indexes into the main attributes vector of the attributes to be read. The *read_attr()* method receives a reference to the Attribute object.

```
1       void StepperMotor::read_attr_hardware(vector<long> &attr_list)
 2  {
 3          cout2 << "In read_attr_hardware for " << attr_list.size();
 4          cout2 << " attribute(s)" << endl;
 5
 6          for (long i = 0;i< attr_list.size();i++)
 7          {
 8                  string attr_name =
 9                          dev_attr->get_attr_by_ind(attr_list[i]).get_name();
10
11                  if (attr_name == "Position")
12                  {
13                          attr_Position_read = &(position[0]);
14                  }
15                  else if (attr_name == "Direction")
16                  {
17                          attr_Direction_read = &(direction[0]);
18                  }
19          }
20  }
21
22  void StepperMotor::read_attr(Attribute &attr)
23  {
24
25          string &attr_name = attr.get_name();
26          cout2 << "In read_attr for attribute " << attr_name << endl;
27
28          if (attr_name == "Position")
29          {
30                  attr.set_value(attr_Position_read);
31          }
32          else if (attr_name == "Direction")
33          {
34                  attr.set_value(attr_Direction_read);
35          }
36  }
```

Line 6 : A loop on each attribute to be read
Line 8 : Get attribute name
Line 11 : Test on attribute name
Line 13 : Read hardware (pretty simple in our case)

[4]It can also be data declared as object data members or memory declared as static

Line 25 : Get attribute name
Line 28 : Test on attribute name
Line 30 : Set attribute value in Attribute object

Retrieving device properties

Retrieving properties is farily simple with the use of the database object. Each Tango device is an agregate with a DbDevice object (see figure 4.1). This has been grouped in a method called *get_ device_ properties*(). The classes and methods of the Dbxxx objets are described in the Tango API documentation.

```
1  void DocDs::get_device_property()
2  {
3          Tango::DbData   data;
4          data.push_back(DbDatum("Max"));
5          data.push_back(DbDatum("Min"));
6
7          get_db_device()->get_property(data);
8
9          if (data[0].is_empty()==false)
10                 data[0]  >>  max;
11         if (data[1].is_empty()==false)
12                 data[1]  >>  min;
13 }
```

Line 4-5 : Two DbDatum (one per property) are stored into a DbData object
Line 7 : Call the database to retrieve properties value
Line 9-10 : If the Max property is defined in the database, extract its value from the DbDatum object and store it in a device data member
Line 11-12 : If the Min property is defined in the database, extract its value from the DbDatum object and store it in a device data member

The remaining methods

The remaining methods are the *dev_ state, dev_ status, always_ executed_ hook* and *dev_ read_ position* methods. The *dev_ state* method parameters are fixed. It does not receive any input parameter and must return a Tango_DevState data type. The *dev_ status* parameters are also fixed. It does not receive any input parameter and must return a Tango string. The *always_ executed_ hook* receives nothing and return nothing. The *dev_ read_ position* method input parameter is the motor number as a long and the returned parameter is the motor position also as a long data type.

```
1  long StepperMotor::dev_read_position(long axis)
2  {
3
4          if (axis < 0 || axis > AGSM_MAX_MOTORS)
5          {
6                  cout1 << "Steppermotor::dev_read_position(): axis out of range !";
7                  cout1 << endl;
8
9                  TangoSys_OMemStream o;
```

```
10
11                  o << "Axis number " << axis << " out of range" << ends;
12                  throw_exception((const char *)"StepperMotor_OutOfRange",
13                                  o.str(),
14                                  (const char *)"StepperMotor::dev_read_position");
15          }
16
17          return position[axis];
18  }
19
20  void always_executed_hook()
21  {
22          cout2 << "In the always_executed_hook method << endl;
23  }
24
25  Tango_DevState StepperMotor::dev_state()
26  {
27          cout2 << "In StepperMotor state command" << endl;
28          return Tango::DeviceImpl::dev_state();
29  }
30
31  Tango_DevString StepperMotor::dev_status()
32  {
33          cout2 << "In StepperMotor status command" << endl;
34          return Tango::DeviceImpl::dev_status();
35  }
```

Line 1-18 : The *dev_read_position* method

Line 6-14 : Throw exception to client if the received axis number is out of range

Line 9 : A TangoSys_OMemStream is used as stream. The TangoSys_OMemStream has been defined in improve portability across platform. For Unix like operating system, it is a ostrtream type. For operating system with a full implementation of the standard library, it is a ostringstream type.

Line 20-23 : The *always_executed_hook* method. It does nothing. It has been included here only as pedagogic usage.

Line 25-29 : The *dev_state* method. It does exectly what the default *dev_state* does. It has been included here only as pedagogic usage

Line 31-35 : The *dev_status* method. It does exectly what the default *dev_status* does. It has been included here only as pedagogic usage

6.10.2 Using Java

The constructor

The constructor take a reference to the StepperMotorClass instance as first parameter[5]. The second parameter is the device name as a Java string.

```
1  package StepperMotor;
2
3  import java.util.*;
```

[5]The StepperMotorClass inherits from the DeviceClass and therefore is a DeviceClass

```
 4  import org.omg.CORBA.*;
 5  import fr.esrf.Tango.*;
 6  import fr.esrf.TangoDs.*;
 7
 8  public class StepperMotor extends DeviceImpl implements TangoConst
 9  {
10          protected final int SM_MAX_MOTORS = 8;
11
12          protected int[]                 axis = new int[SM_MAX_MOTORS];
13          protected int[]                 position = new int[SM_MAX_MOTORS];
14          protected int[]                 direction = new int[SM_MAX_MOTORS];
15          protected int[]                 state = new int[SM_MAX_MOTORS];
16
17          protected int[]                 attr_Direction_read = new int[1];
18          protected int[]                 attr_Position_read = new int[1];
19          protected int                   attr_SetPosition_write;
20
21
22          StepperMotor(DeviceClass cl,String s,String desc,
23                       DevState state,String status) throws DevFailed
24          {
25                  super(cl,s,desc,state,status);
26                  init_device();
27          }
28
29          public void init_device()
30          {
31                  System.out.println("StepperMotor() create motor " + dev_name);
32
33                  int i;
34
35                  for (i=0; i< SM_MAX_MOTORS; i++)
36                  {
37                          axis[i] = 0;
38                          position[i] = 0;
39                          direction[i] = 0;
40                          state[i] = 0;
41                  }
42
43          }
44  }
```

Line 3-6: Import different packages. The first one (**java.lang.util**) is a classical Java package from the JDK. The second one (**org.omg.CORBA**) is a package which contains all the CORBA related classes. The third one (**fr.esrf.Tango**) is the package generated by the IDL compiler from the Tango IDL file. The last one (**fr.esrf.TangoDs**) is the name of the package with all the root classes of the device server pattern.

Line 8 : The StepperMotor class inherits from the DeviceImpl class and implements the **TangoConst** interface. The TangoConst interface does not defines any method but simply defines constant variables. The TangoConst interface is a member of the TangoDs package.

Line 10 : Define an internal constant

Line 12-15 : Device internal variable

Line 17-19 : Device internal variable linked to attributes

Line 22-27 : The class constructor. It execute the DeviceImpl class constructor with five parameters. Then the *init_device* method is executed.

Line 29-43 : The *init_device* method. All the device data initialisation is done in this method.

The methods used for the DevReadDirection command

The DevReadDirection command is created using the template command method. Therefore, there is no specific class needed for this command but only one object of the TemplCommandInOut class. This command needs two methods which are the *dev_read_direction* method and the *direct_cmd_allowed* method. The *direct_cmd_allowed* method defines here implements exactly the same behaviour than the default one. This method has been used only for pedagogic issue. The *dev_read_direction* method will be executed by the *execute* method of the TemplCommandInOut class. The *direct_cmd_allowed* method will be executed by the *is_allowed* method of the TemplCommandInOut class.

```
1  public int dev_read_direction(int axis) throws DevFailed
2  {
3          if (axis < 0 || axis > SM_MAX_MOTORS)
4          {
5                  Util.out1.println("Steppermotor.dev_read_direction(): axis out of ra
6
7                  StringBuffer o = new StringBuffer("Axis number ");
8                  o.append(axis);
9                  o.append(" out of range");
10
11                 Except.throw_exception("StepperMotor_AxisOutOfRange",
12                                        o.toString(),
13                                        "StepperMotor.dev_read_direction()");
14         }
15
16         return direction[axis];
17 }
18
19 public boolean direct_cmd_allowed(Any data_in)
20 {
21         Util.out2.println("In StepperMotor.direct_cmd_allowed method");
22
23         return true;
24 }
```

Line 1-17 : The *dev_read_direction* method

Line 3-14 : Throw exception to client if the received axis number is out of range

Line 19-24 : The *direct_cmd_allowed* method. The command input data is passed to this method iin case of it is needed to take the decision. This data is still packed into the CORBA Any object.

The write attribute related method

To enable writing of writable attributes, the StepperMotor class must re-define a method called *write_attr_hardware()*. The aim of this method is to write the hardware. This method receives a

vector of Integer objects as parameters. These data are the indexes of the attributes to be written into the main attribute vector stored in the MultiAttribute object. Methods of the MultiAttribute class allow the retrieval of the the correct attribute object from these indexes. The value to be written is stored in the WAttribute object and can be retrieved with WAttribute class methods called *get_xx_write_value()*. A data member called *attr_<Attribute_name>_write* is foreseen to temporary store this extracted value.

```
1  public void write_attr_hardware(Vector attr_list)
2  {
3          Util.out2.println("In write_attr_hardware for "+attr_list.size()+" attribute
4
5          for (int i = 0;i < attr_list.size();i++)
6          {
7                  int ind = ((Integer)(attr_list.elementAt(i))).intValue();
8                  WAttribute att = dev_attr.get_w_attr_by_ind(ind);
9                  String att_name = att.get_name();
10
11                 if (att_name.equals("SetPosition") == true)
12                 {
13                         attr_SetPosition_write = att.get_lg_write_value();
14                         Util.out2.println("Attribute SetPosition value = "+attr_SetP
15                         position[0] = attr_SetPosition_write;
16                 }
17         }
18 }
```

Line 5 : A loop on each attribute to be written
Line 7-9 : Retrieve attribute name
Line 11 : A test on attribute name
Line 13 : Retrieve new attribute value
Line 15 : Set the hardware (very simple in our example)

The read attribute related methods

To enable reading of attributes, the StepperMotor class must re-define two methods called *read_attr_hardware()* and *read_attr()*. The aim of the first one is to read the harware. It will be called only once at the beginning of each read_attributes CORBA call. The second method aim is to build the exact data for the wanted attribute and to store this value into the Attribute object. This method will be called for each attribute to be read. Special care has been taken in order to minimize the number of data copy and allocation. The data passed to the Attribute object as attribute value is passed using pointers. It must be allocated by the method[6] and the Attribute object will not free this memory. Data members called attr_<Attribute_name>_read are foreseen for this usage. As for the *write_attr_hardware()* method, the *read_attr_hardware()* method receives a vector of long which are indexes into the main attributes vector of the attributes to be read. The *read_attr()* method receives a reference to the Attribute object.

```
1  public void read_attr_hardware(Vector attr_list)
2  {
```

[6]It can also be data declared as object data members or memory declared as static

```
 3          Util.out2.println("In read_attr_hardware for "+attr_list.size()+" attribute(
 4          for (int i = 0;i< attr_list.size();i++)
 5          {
 6                  int ind = ((Integer)(attr_list.elementAt(i))).intValue();
 7                  String attr_name = dev_attr.get_attr_by_ind(ind).get_name();
 8
 9                  if (attr_name == "Position")
10                  {
11                          attr_Position_read[0] = position[0];
12                  }
13                  else if (attr_name == "Direction")
14                  {
15                          attr_Direction_read[0] = direction[0];
16                  }
17          }
18  }
19
20
21  public void read_attr(Attribute attr) throws DevFailed
22  {
23          String attr_name = attr.get_name();
24          Util.out2.println("In read_attr for attribute "+attr_name);
25          if (attr_name.equals("Position") == true)
26          {
27                  attr.set_value(attr_Position_read);
28          }
29          else if (attr_name.equals("Direction") == true)
30          {
31                  attr.set_value(attr_Direction_read);
32          }
33  }
```

Line 4 : A loop on each attribute to be read
Line 6 -7: Get attribute name
Line9 : Test on attribute name
Line 11 : Read hardware (pretty simple in our case)
Line 23 : Get attribute name
Line 25 : Test on attribute name
Line 27 : Set attribute value in Attribute object

Retrieving device properties

Retrieving properties is farily simple with the use of the database object. Each Tango device is an agregate with a DbDevice object (see figure 4.1). This has been grouped in a method called *get_ device_ properties*(). The classes and methods of the Dbxxx objets are described in the Tango API documentation.

```
1  void public get_device_property() throws DevFailed
2  {
3          String[] prop_names = {"Max","Min"};
4
```

```
5           DbDatum[] res_value = db_dev.get_property(prop_names);
6
7           if (res_value[0].is_empty() == false)
8                   min = res_value[0].extractInt();
9           if (res_value[1].is_empty() == false)
10                  max = res_value[1].extractInt();
11  }
```

Line 3 : Define the names of the properties to be retrieved

Line 5 : Call the database to retrieve properties value

Line 7-8 : If the Max property is defined in the database, extract its value from the DbDatum object and store it in a device data member

Line 9-10 : If the Min property is defined in the database, extract its value from the DbDatum object and store it in a device data member

The remaining methods

The remaining methods are the *dev_state, dev_status, always_executed_hook()* and *dev_read_position* methods. The *dev_state* method parameters are fixed. It does not receive any input parameter and must return a DevState data type. The *dev_status* parameters are also fixed. It does not receive any input parameter and must return reference to a Java string. The *always_executed_hook* receives nothing and return nothing The *dev_read_position* method input parameter is the motor number as an int and the returned parameter is the motor position also as an int data type.

```
1   int dev_read_position(int axis) throws DevFailed
2   {
3
4           if (axis < 0 || axis > SM_MAX_MOTORS)
5           {
6                   Util.out1.println("Steppermotor.dev_read_position(): axis out of ran
7
8                   StringBuffer o = new StringBuffer("Axis number ");
9                   o.append(axis);
10                  o.append(" out of range");
11
12                  Except.throw_exception("StepperMotor_AxisOutOfRange",
13                                         o.toString(),
14                                         "StepperMotor.dev_read_position()");
15          }
16
17          return position[axis];
18  }
19
20  public void always_executed_hook()
21  {
22          Util.out2.println("In always_executed_hook method");
23  }
24
25  public DevState dev_state() throws DevFailed
26  {
27          Util.out2.println("In StepperMotor state command");
```

```
28          return super.dev_state();
29  }
30
31  public String dev_status() throws DevFailed
32  {
33          Util.out2.println("In StepperMotor status command");
34          return super.dev_status();
35  }
```

Line 1-18 : The *dev_read_position* method
Line 4-15 : Throw exception to client if the received axis number is out of range
Line 20-23 : The *always_executed_hook* method.It does nothing. It has been included here only as pedagogic usage.
Line 25-29 : The *dev_state* method. It does exectly what the default *dev_state* does. It has been included here only as pedagogic usage
Line 31-35 : The *dev_status* method. It does exectly what the default *dev_status* does. It has been included here only as pedagogic usage

6.11 Source files management

Tango uses CVS as source file management system. To ease device server developper work, some tools have been written to ease CVS use. These tools are :

prjcreate This command initialise a new project in the CVS repository

prjin It records into CVS database a new project release

prjout Extract the last release of a project

prjadd To add file(s) to an existing project

prjremove To remove file(s) from an existing project

prjdiff Summarises difference between project files in the working directory and in the CVS repository

These tools are self documented. To get a minimun help, simply type the tool name and hit the return key. In the ESRF file system structure, they are stored in the */segfs/tango/bin* directory. You can find more information about CVS in [12].

Chapter 7

Device server under Windows

Two kind of programs are available under Windows. These kinds of programs are called console application or Windows application. A console application is started from a MS-DOS window and is very similar to classical UNIX program. A Windows application is most of the time not started from a MS-DOS window and is generally a graphical application without standard input/output. Writing a device server in a console application is straight forward following the rules described in the previous sub-chapters. Writing a device server in a Windows application needs some changes detailed in the following sub-chapters.

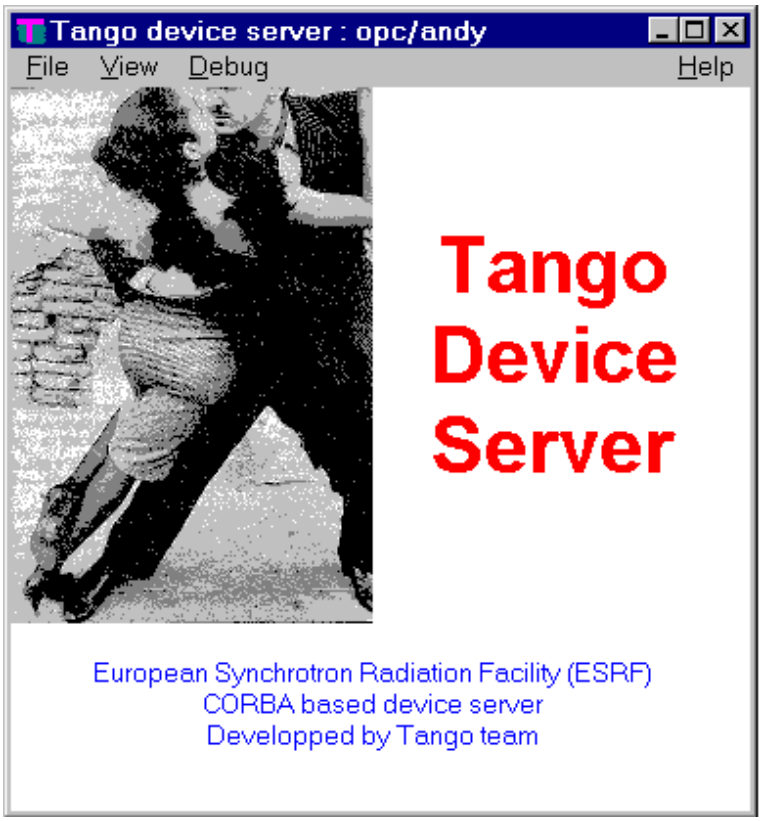
7.1 The Tango device server graphical interface

Within the Windows operating system, most of the running application has a window user interface. This is also true for the Windows Tango device server. Using or not this interface is up to the device server programmer. The choice is done with an argument to the *server_init()* method of the Tango::Util class. This interface is pretty simple and is based on three windows which are :

- The device server main window
- The device server console window
- The device server help window

7.1.1 The device server main window

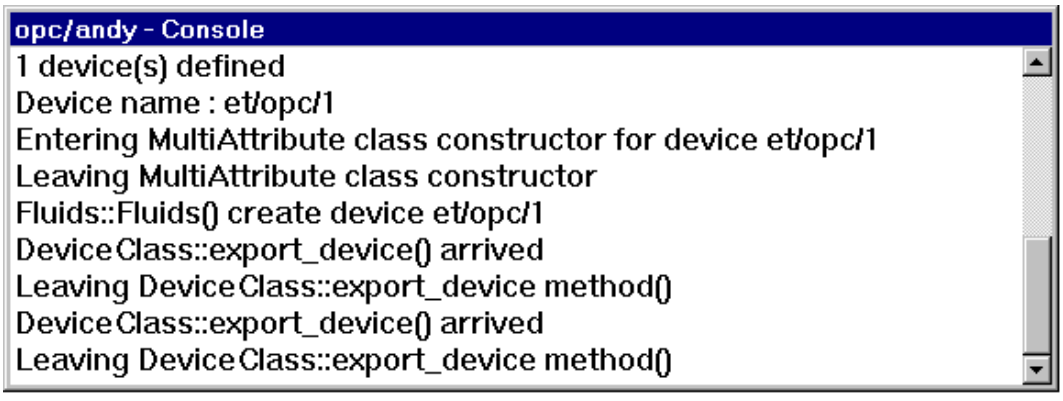
This window looks like :



Four menus are available in this window. The File menu allows the user to exit the device server. The View menu allows you to display/hide the device server console window. The Debug menu allows the user to change the server output verbose level. All the outputs goes to the console window even if it is hidden. The Help menu displays the help window. The device server name is displayed in the window title. The text displayed at the bottom of the window has a default value (the one displayed in this window dump) but may be changed by the device server programmer using the *set_main_window_text()* method of the Tango::Util class. If used, this method must be called prior to the call of the *server_init()* method. Refer to [6] for a complete description of this method.

7.1.2 The console window

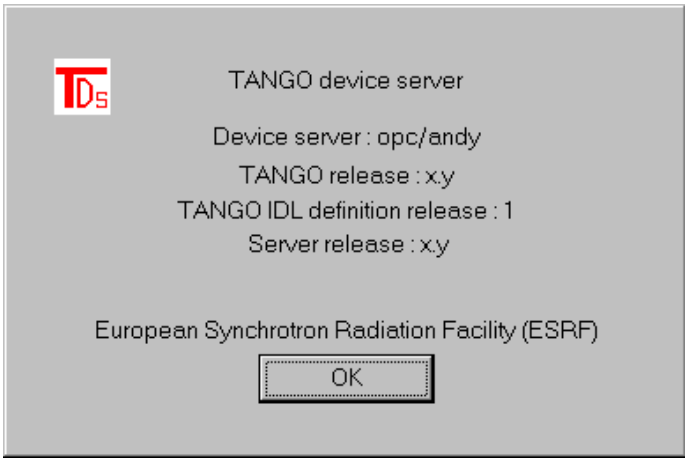
This window looks like :



It simply displays all the output of the *coutx* used in the device server.

7.1.3 The help window

This window looks like :



This window displays

- The device server name
- The Tango library release
- The Tango IDL definition release
- The device server release. The device server programmer may set this release number using the *set_server_version()* method of the Tango::Util class. If used, this must be done prior to the call of the *server_init()* method. If the *set_server_version()* method is not used, x.y is displays as version number. Refer to [6] for a complete description of this method.

7.2 MFC device server

There is no *main* function within a classical MFC program. Most of the time, your application is represented by one instance of a C++ class which inherits from the MFC CWinApp class. This CWinApp class has several methods that you may overload in your application class. For a device server to run correctly, you must overload two methods of the CWinApp class. These methods are the *InitInstance()* and *ExitInstance()* methods. The rule of these methods is obvious following their names.

Remember that if the Tango device server graphical user interface is used, you must link your device server with the Tango windows resource file. This is done by adding the Tango resource file to the Project Settings/Link/Input/Object, library modules window in VC++.

7.2.1 The InitInstance method

The code to be added here is the equivalent of the code written in a classical *main()* function. Don't forget to add the *tango.h* file in the list of included files.

```
1 BOOL FluidsApp::InitInstance()
2  {
3          AfxEnableControlContainer();
4
```

```
 5          // Standard initialization
 6          // If you are not using these features and wish to reduce the size
 7          //  of your final executable, you should remove from the following
 8          //  the specific initialization routines you do not need.
 9
10  #ifdef _AFXDLL
11          Enable3dControls();                     // Call this when using MFC in a sha
12  #else
13          Enable3dControlsStatic();       // Call this when linking to MFC statically
14  #endif
15          Tango::Util *tg;
16          try
17          {
18
19                  tg = Tango::Util::init(m_hInstance,m_nCmdShow);
20
21                  tg->server_init(true);
22
23                  tg->server_run();
24
25          }
26          catch (bad_alloc)
27          {
28                  MessageBox((HWND)NULL,"Memory error","Command line",MB_ICONSTOP);
29                  return(FALSE);
30          }
31          catch (Tango::DevFailed &e)
32          {
33                  MessageBox((HWND)NULL,,e.errors[0].desc.in(),"Command line",MB_ICONST
34                  return(FALSE);
35          }
36          catch (CORBA::Exception &)
37          {
38                  MessageBox((HWND)NULL,"Exception CORBA","Command line",MB_ICONSTOP);
39                  return(FALSE);
40          }
41
42          m_pMainWnd = new CWnd;
43          m_pMainWnd->Attach(tg->get_ds_main_window());
44
45          return TRUE;
46  }
```

Line 19 : Initialise Tango system. This method also analises the argument used in command line.

Line 21 : Create Tango classes requesting the Tango Windows graphical interface to be used

Line 23 : Start Network listenner. Note that under NT, this call returns in the contrary of UNIX like operating system.

Line 26-30 : Display a message box in case of memory allocation error and leave method with a return value set to false in order to stop the process

Line 31-35 : Display a messag ebox in case of error during server ionitialisation phase.

Line 36-40 : Display a message box in case of error other than memory allocation. Leave method with a return value set to false in order to stop the process.

Line 37-38 : Create a MFC main window and attach the Tango graphical interface main window to this MFC window.

7.2.2 The ExitInstance method

This method is called when the application is stopped. For Tango device server, its rule is to destroy the Tango::Util singleton if this one has been correctly constructed.

```
1  int FluidsApp::ExitInstance()
2  {
3          bool del = true;
4
5          try
6          {
7                  Tango::Util *tg = Tango::Util::instance();
8          }
9          catch(Tango::DevFailed)
10         {
11                 del = false;
12         }
13
14         if (del == true)
15                 delete (Tango::Util::instance());
16
17         return CWinApp::ExitInstance();
18 }
```

Line 7 : Try to retrieve the Tango::Util singleton. If this one has not been constructed correctly, this call will throw an exception.

Line 9-12 : Catch the exception in case of incomplete Tango::Util singleton construction

Line 14-15 : Delete the Tango::Util singleton. This will unregister the Tango device server from the Tango database.

Line 17 : Execute the *ExitInstance* method of the CWinApp class.

If you dont't want to use the Tango device server graphical interface, do not pass any parameter to the *server_init()* method and instead of the code display in lines 37 and 38 in the previous example of the *InitInstance()* method, use your own code to initialise your own application.

7.2.3 Example of how to build a Windows device server MFC based

This sub-chapter gives an example of what it is needed to do to build a MFC Windows device server. Rather than being a list of actions to stricly follow, this is some general rules of how using VC++ to build a Tango device server using MFC.

1. Create your device server using Pogo. For a class named MyMotor, the following files will be needed : *class_factory.cpp*, *MyMotorClass.h*, *MyMotorClass.cpp*, *MyMotor.h* and *MyMotor.cpp*.

2. On a Windows computer running VC++, create a new project of type "MFC app Wizard (exe)" using static MFC libs. Ask for a dialog based project without ActiveX controls.

3. Copy the five files generated by Pogo to the Windows computer and add them to your project

4. Remove the dialog window files (xxxDlg.cpp and xxxDlg.h), the Resource include file and the resource script file from your project

5. Add #include <stdafx.h> as first line of the include files list in *class_factory.cpp*, *MyMotorClass.cpp* and *MyMotor.cpp* file. Also add your own directory and the Tango include directory to the project precompiler include directories list.

6. Enable RTTI in your project settings (see chapter 8.1.2)

7. Change your application class:

   (a) Add the definition of an *ExitInstance* method in the declaration file. (xxx.h file)

   (b) Remove the include of the dialog window file in the xxx.cpp file and add an include of the Tango master include files (tango.h)

   (c) Replace the *InitInstance*() method as described in previous sub-chapter. (xx.cpp file)

   (d) Add an *ExitInstance()* method as described in previous sub-chapter (xxx.cpp file)

8. Add all the libraries needed to compile a Tango device server (see chapter 8.1.2) and the Tango resource file to the linker Object/Libraries modules.

7.3 Win32 application

Even if it is more natural to use the C++ structure of the MFC class to write a Tango device server, it is possible to write a device server as a Win32 application. Instead of having a *main()* function as the application entry point, the operating system, provides a *WinMain()* function as the application entry point. Some code must be added to this *WinMain* function in order to support Tango device server. Don't forget to add the *tango.h* file in the list of included files.

```
 1  int APIENTRY WinMain(HINSTANCE hInstance,
 2                       HINSTANCE hPrevInstance,
 3                       LPSTR     lpCmdLine,
 4                       int       nCmdShow)
 5  {
 6          MSG msg;
 7          Tango::Util *tg;
 8
 9          try
10          {
11                  tg = Tango::Util::init(hInstance,nCmdShow);
12
13                  string txt;
14                  txt = "Blabla first line\n";
15                  txt = txt + "Blabla second line\n";
16                  txt = txt + "Blabla third line\n";
17                  tg->set_main_window_text(txt);
18                  tg->set_server_version("2.2");
19
20                  tg->server_init(true);
21
22                  tg->server_run();
23
24          }
```

```
25          catch (bad_alloc)
26          {
27                  MessageBox((HWND)NULL,"Memory error","Command line",MB_ICONSTOP);
28                  return (FALSE);
29          }
30          catch (Tango::DevFailed &e)
31          {
32                  MessageBox((HWND)NULL,e.errors[0].desc.in(),"Command line",MB_ICONST
33                  return (FALSE);
34          }
35          catch (CORBA::Exception &)
36          {
37                  MessageBox((HWND)NULL,"Exception CORBA","Command line",MB_ICONSTOP);
38                  return(FALSE);
39          }
40
41          while (GetMessage(&msg, NULL, 0, 0))
42          {
43                  TranslateMessage(&msg);
44                  DispatchMessage(&msg);
45          }
46
47          delete tg;
48
49          return msg.wParam;
50  }
```

Line 11 : Create theTango::Util singleton
Line 13-18 : Set parameters for the graphical interface
Line 20 : Initialise Tango device server requesting the display of the graphical interface
Line 22 : Run the device server
Line 25-39 : Display a message box for all ther kinds of error during Tango device server initialisation phase and exit WinMain function.
Line 41-45 : The Windows message loop
Line 47 : Delete the Tango::Util singleton. This class destructor unregissters the device server from the Tango database.

Remember that if the Tango device server graphical user interface is used, you must link your device server with the Tango windows resource file. This is done by adding the Tango resource file to the Project Settings/Link/Input/Object, library modules window in VC++.

If you don't want to use the tango device server graphical user interface, do not use any parameter in the call of the *server_init()* method and do not link your device server with the Tango Windows resource file.

7.4 Device server as NT service

With Windows NT, if you want to have processes wich survive to logoff sequence and/or are automatically started during computer startup sequence, you have to write them as service. It is possible to write Tango device server as service. You need to

1. Write a class which inherits from a pre-written Tango class called NTService. This class must have a *start* method.

2. Write a main function following a predefined skeleton.

7.4.1 The service class

It must inherits from the *NTService* class and defines a *start* method. The NTService class must be constructed with one argument which is the device server executable name. The *start* method has three arguments which are the number of arguments passed to the method, the argument list and a reference to an object used to log info in the NT event system. The first two args must be passed to the Tango::Util::init method and the last one is used to log error or info messages. The class definition file looks like

```
 1  #include <tango.h>
 2  #include <ntservice.h>
 3
 4  class MYService: public Tango::NTService
 5  {
 6  public:
 7          MYService(char *);
 8
 9          void start(int,char **,OB::Logger_ptr);
10  };
```

Line 1-2 : Some include files
Line 4 : The MYService class inherits from *Tango::NTService* class
Line 7 : Constructor with one parameter
Line 9 : The *start()* method

The class source code looks like

```
 1  #include <myservice.h>
 2  #include <tango.h>
 3  #include <ob/logger.h>
 4
 5  using namespace std;
 6
 7  MYService::MYService(char *exec_name):NTService(exec_name)
 8  {
 9  }
10
11  void MYService::start(int argc,char **argv,OB::Logger_ptr log)
12  {
13          Tango::Util *tg;
14          try
15          {
16                  Tango::Util::_service = true;
17
18                  tg = Tango::Util::init(argc,argv);
19
20                  tg->server_init();
```

```
21
22                  tg->server_run();
23          }
24          catch (bad_alloc)
25          {
26                  logger_->error("Can't allocate memory to store device object");
27          }
28          catch (Tango::DevFailed &e)
29          {
30                  logger_->error(e.errors[0].desc.in());
31          }
32          catch (CORBA::Exception &)
33          {
34                  logger_->error("CORBA Exception");
35          }
36  }
```

Line 7-9 : The MYService class constructor code.
Line 16 : Set to true the *_service* static variable of the *Tango::Util* class.
Line 18-22 : Classical Tango device server startup code
Line 24-35 : Exception management. Please, note that within a service. it is not possible to print data on a console. This method receives a reference to a logger object. This object sends all its output to the Windows NT event system. It is used to send messages when an exception has occured.

7.4.2 The main function

The main function is used to create one instance of the class describing the service, to check the service option and to run the service. The code looks like :

```
 1  #include <tango.h>
 2  #include <MYService.h>
 3
 4  using namespace std;
 5
 6
 7  int main(int argc,char *argv[])
 8  {
 9          MYService service(argv[0]);
10
11          int ret;
12          if ((ret = service.options(argc,argv)) <= 0)
13                  return ret;
14
15          service.run(argc,argv);
16
17          return 0;
18  }
```

Line 9 : Create one instance of the MYService class with the executable name as parameter
Line 12 : Check service option with the *options()* method inherited from the NTService class.
Line 15 : Run the service. The *run()* method is inherited from the NTService class. This method will after some NT initialisation sequence execute the user *start()* method.

7.4.3 Service options and messages

When a Tango device server is written as a Windows NT service, it supports several new options. These option are linked to Windows NT service usage.

Before it can be used, a service must be installed. A name and a title is associated to each service. For Tango device server used as service, the service name is build from the executable name followed by the underscore character and the instance name. For example, a device server service executable file named "opc" and started with "fluids" as instance name, will be named "opc_fluids". The title string is built from the service executable name followed by the sentence "Tango device server" and the instance name between paranthesis. In the previous example, the service title will be "opc Tango device server (fluids)". Once a service is installed, you can configure it with the "Services" application of the control panel. Services title are displayed by this application and allow the user to select one specific service. Once a service is selected, it is possible to start/stop it and to configure its startup type as manual (with the Services application) or as automatic. When the automatic mode is chosen, the service starts when the computer is started. In this case, the service executable code must resides on the computer local disk.

Tango device server loggs message in the Windows event system when the service is started or stopped. You can see these messages with the "Event Viewer" application (Start->Programs->Administrative tools->Event Viewer) and choose the Application events.

The new options are -i, -s, -u, -h and -d.

- -i : Install the service
- -s : Install the service and choose the automatic startup mode
- -u : Uninstall the service
- -h : Display help message
- -d : Run in console mode to debug service. The service must have been installed prior to used it. The classical -v device server option can be used with the -d option.

On the command line, all these options must be used after the device server instance name ("opc fluids -i" to install the service, "opc fluids -u" to uninstall the service, "opc fluids -v -d" to debug the service)

7.4.4 Tango device server using MFC as Windows NT service

If your Tango device server uses MFC and must be written as a Windows NT service, follow these rules :

- Don't forget to add the *stdafx.h* file as the first file included in all the source files making the project.
- Comment out the definition of VC_EXTRALEAN in the *stdafx.h* file.
- Change the proprocessor definitions, replace _WINDOWS by _CONSOLE
- Add the /SUBSYSTEM:CONSOLE option in the linker options window of the project settings.
- Add a call to initialise the MFC (*AfxWinInit()*) in the service main function

```
 1  int main(int argc,char *argv[])
 2  {
 3          if (!AfxWinInit(::GetModuleHandle(NULL),NULL,::GetCommandLine(),0))
 4          {
 5                  cerr << "Can't initialise MFC !" << endl;
 6                  return -1;
 7          }
 8
 9          service serv(argv[0]);
10
11          int ret;
12          if ((ret = serv.options(argc,argv)) <= 0)
13                  return ret;
14
15          serv.run(argc,argv);
16
17          return 0;
18  }
```

Line 3 : The MFC classes are initialised with the *AfxWinInit()* function call.

Chapter 8

Compiling, linking and executing a TANGO device server process

8.1 Compiling and linking a C++ device server

8.1.1 On UNIX like operating system

Supported development tools

The supported compiler for Linux and Solaris is **gcc** release 2.95.2 and above. For HP-UX, aCC release 1.21 and above is supported. Please, note that to debug a Tango device server running under Linux, **gdb** release 5 and above is needed in order to correctly handle threads.

Compiling

TANGO uses two products from the OOC[1] company [9](free for non-commercial usage). These two products are called JTC[2] and ORBacus. JTC emulates Java like thread in a C++ software and ORBacus is the CORBA Object Request Broker. To compile a TANGO device server, your include search path must be set to :

- The JTC include directory
- The ORBacus include directory
- The Tango include directory
- Your development directory

ORBacus uses thread. With HP computers, the source files must be compiled with the -D_REENTRANT and -D_CMA_NOWRAPPERS_ options in order to correctly handle thread.

Linking

To build a running device server process, you need to link your code with several libraries. Three of them are always the same whatever the operating system used is. These three libraries are:

- The Tango library (called **libtango**)
- The ORBacus package library (called **libOB**)
- The JTC package library (called **libJTC**)

[1] OOC stands for **O**bject **O**riented **C**oncept
[2] JTC stands for **J**ava **T**hread in **C**++

On top of that, you need additional libraries depending on the operating system :

- For HP-UX 10.20, add the cma library (**libcma**)
- For Solaris 7, add the posix4 library (**libposix4**), the socket library (**libsocket**), the nsl library (**libnsl**) and the posix thread library (**libpthread**)
- For Linux (Suse 6), add the posix thread library (**libpthread**)

The following is an example of a Makefile for Linux. Obviously, all the paths are set to the ESRF file system structure.

```
 1  #
 2  #               Makefile to generate a Tango server
 3  #
 4
 5  CC = c++
 6  BIN_DIR = suse64
 7  TANGO_HOME = /segfs/tango
 8
 9  INCLUDE_DIRS = -I $(TANGO_HOME)/include/$(BIN_DIR) \
10                 -I .
11
12  LIB_DIRS = -L $(TANGO_HOME)/lib/$(BIN_DIR)
13
14
15  CXXFLAGS =  $(INCLUDE_DIRS)
16  LFLAGS =  $(LIB_DIRS) -ltango -lOB -lJTC -lpthread
17
18
19  SVC_OBJS =      main.o \
20                  classfactory.o \
21                  steppermotorclass.o \
22                  steppermotor.o
23
24
25  .SUFFIXES:      .o .cpp
26  .cpp.o:
27          $(CC) $(CXXFLAGS) -c $<
28
29
30  all: StepperMotor
31
32  StepperMotor:   $(SVC_OBJS)
33          $(CC) $(SVC_OBJS) -o $(BIN_DIR)/StepperMotor $(LFLAGS)
34
35  clean:
36          rm -f *.o core
```

Line 5-7 : Define Makefile macros
Line 9-10 : Set the include file search path
Line 12 : Set the linker library search path

Line 15 : The compiler option setting
Line 16 : The linker option setting
Line 19-22 : All the object files needed to build the executable
Line 25-27 : Define rules to generate object files
Line 30 : Define a "all" dependency
Line 32-33 : How to generate the StepperMotor device server executable
Line 35-36 : Define a "clean" dependency

8.1.2 On Windows NT using Developer Studio

Supported Windows compiler for Tango is Visual C++ release 6 **with its service pack number 3** installed. Most problems in building a Windows device server revolve around the /M compiler switch family. This switch family controls which run-time library names are embedded in the object files, and consequently which libraries are used during linking. Attemps to mix and match compiler settings and libraries can cause link error and even if successful, may produce undefined run-time behaviour.

Selecting the correct /M switch in Developer Studio is done through a dialog box. To open this dialog box, click on the "Project" menu and select the "Settings" option. To change the compiler switch click on the "C/C++" tab and select "Code Generation" from the "Category" drop-down list. The "Use run-time library" drop-down list is used to change the compiler switch. By looking at the string in the "Project options" edit box, you can see what the switch value is for the drop-down list selection.

- Single-threaded = /ML
- Multithreaded = /MT (Supported)
- Multihtreaded DLL = /MD
- Debug Single-threaded = /MLd
- Debug Multithreaded = /MTd (Supported)
- Debug Multithreaded DLL = /MDd

Compiling a file with a value of the /M switch family will impose at link phase the use of libraries also compiled with the same value of the /M switch family. If you compiled your source code with the /MT option (Multihtreaded), you must link it with libraries also compiled with the /MT option.

The ORBacus package used by TANGO, makes extensive use of exceptions and RTTI[3]. This requires the /GX and /GR options be enabbled when compiling. The setting can be found in Developper Studio in the "Project Settings" dialog box. Click on the "C/C++" tab and select "C++ language" in the "Category" drop-down list.

ORBacus and TANGO relies on the preprocessor identifier WIN32 being defined in order to configure itself. Normally this will already be defined in a Developper Studio created project.

To build a running device server process, you need to link your code with several libraries on top of the Windows libraries. These libraries are:

- The Tango library (called **tango.lib**)
- The ORBacus packge library (called **ob.lib** or **obd.lib**)
- The JTC package library (called **jtc.lib** or **jtcd.lib**)
- A windows network library (**wsock32.lib**)

[3]RTTI stands for **R**un **T**ime **T**ype **I**dentification

To add these libraries in Developper Studio, open the "Project Settings" dialog box and click on the "Link" tab. Select "Input" from the "Category" drop-down list and add these library names to the list of library in the "Object/library modules" box.

If your device server is written using the MFC, use the static release of the class library. On top of that, for the MFC device server or a Win32 graphic application add the following library to your link order

- The Windows common controls library called **comctl32.lib**

The "Win32 Debug" or "Win32 Release" configuration that you change with the "Build/Set active configuration" menu changes the /M switch compiler. If you select a "Win32 Debug" configuration, use the obd.lib and jtcd.lib libraries and the tango.lib library in the *debug* directory (at the ESRF). If you select the "Win32 Release" configuration, use the ob.lib and jtc.lib libraries and the tango.lib library in the *release* directory (at the ESRF).

8.2 Running a C++ device server

To run a C++ Tango device server, you must set an environment variable. This environment variable is called **TANGO_HOST** and has a fixed syntax which is

TANGO_HOST = <host>:<port>

The host field is the host name where the TANGO database device server is running. The port field is the port number on which this server is listenning. For instance, a valid syntax is TANGO_HOST=dumela:10000. For UNIX like operating system, setting environment variable is possible with the *export* or *setenv* command depending on the shell used. For Windows NT, setting environment variable is possible with the "Environment" tab of the "System" application in the control panel.

8.3 Compiling a Java device server

8.3.1 Supported java release

Tango device server written using Java language needs release **1.2.2** (or above) of the Java environment.

8.3.2 Setting the CLASSPATH

To correctly compile a Java Tango device server, the CLASSPATH environment variable must be set to :

- The jar file with all the CORBA classes. This file is named OB.jar
- The jar file with all the Tango and TangoDs package classes. This file is named Tango.jar
- The jar file with all the JDK classes (not always necessary, could be implicit)
- Your own directory

For UNIX like operating system, setting environment variable is done with the *export* or *setenv* command depending on the shell used. For Windows NT, setting environment variable is possible with the "Environment" tab of the "System" application in the control panel.

8.3.3 Makefile

The following is an example of a Makefile for a Java Tango device server. Obviously, all the paths are set to the ESRF file system structure.

```
 1  #
 2  #               Makefile to generate a TANGO java device server
 3  #
 4
 5  JAVAC = javac -classpath $(CLASSPATH):..
 6
 7  # ------------------------------------------------
 8  #
 9  #               The compiler flags
10  #
11  #-------------------------------------------------
12
13  JAVAFLAGS = -g
14
15  #-------------------------------------------------
16
17
18  CL_LIST =       DevReadPositionCmd.class \
19                  StepperMotor.class \
20                  StepperMotorClass.class
21
22  PACKAGE = server
23
24  #
25  # Rule for compiling
26  #
27
28  .SUFFIXES:      .class .java
29  .java.class:
30          $(JAVAC) $(JAVAFLAGS) $<
31
32  #--------------------------------------------------
33
34
35  all:    $(PACKAGE)
36
37  $(PACKAGE):     $(CL_LIST)
38
39  clean:
40          rm -f *.class
```

Line 5 : Makefile macro. The Makefile file is the same directory than the source files. As all files are part of the StepperMotor package, it is necessary to add the top directory to the classpath.

Line 13 : The java compiler flag

Line 18 : List of class to be compiled

Line 22 : Define a dependency name

Line 28-30 : Define how source files must be compiled
Line 35 : The "all" dependency
Line 37 : The device server dependency
Line 39-40 : The "clean" dependency

8.3.4 Tango core software release number

All the Tango core classes are packaged in the Tango.jar file. A litlle utility tool called **TangoVers** allows a user to know which release of the Tango core classes he(she) is using. This utility is available only with Java 1.2 virtual machine (on *apus* computer at the ESRF). To run this utility, simply type

TangoVers <path to Tango.jar file>

if the directory /segfs/tango/bin is in your PATH environment variable.

8.4 Running a Java device server

A correct setting of the CLASSPATH environment variable is not enough to run a Java Tango device server. You must also set a Java system property. The name of the system property is **TANGO_HOST** and its syntax is the same than the syntax described in chapter 8.2. Setting a Java system property is done by using -D option of the java interpreter command. To run a Java Tango device server, the command line must start with

java -DTANGO_HOST=<host>:<port> xxxx

As all the device server files are part of a package, you have to run this command in the directory above the package directory. For instance, for our StepperMotor device server started with et s instance name, all files must be stored in a directory called StepperMotor and the command line must be

java -DTANGO_HOST=<host>:<port> StepperMotor/StepperMotor et

run from the directory above the StepperMotor one.

Chapter 9

Advanced programming techniques

The basic techniques for implementing device server pattern are required by each device server programmer. In certain situations, it is however necessary to do things out of the ordinary. This chapter will look into programming techniques which permit the device server serve more than simply the network.

9.1 Receiving signal (C++ specific)

It is **UNSAFE** to use any CORBA call in a signal handler. It is also UNSAFE to use some system calls in a signal handler. Tango device server solved this problem by using threads. A specific thread is started to handle signals. Therefore, every Tango device server is automatically a threaded process. This allows the programmer to write the code which must be executed when a signal is received as ordinary code.

Nevertheless, signal management is not trivial and some care have to be taken. The signal management differs from operating system to operating system. It is not recommanded that you install your own signal routine using any of the signal routines provided by the operationg system calls or library.

Using Linux

The classsical thread library is used by the Tango device server. The thread management offered by the Linux kernel and this library is a pure kernel-thread based implementation. This means that each thread is seen as a process (each thread has a separate PID, the *ps* command displays one line for each thread) even if they are not real process. For a Tango device server, a *ps* command will show you three threads which are :

1. The device server main thread
2. The thread manager (created by the Linux thread library)
3. The device server signal management thread

The PID stored in the Tango database is the PID of the signal thread. All signals should be sent to the signal thread and to kill a server from a console window, the PID of the signal thread should be used. The Linux thread library is using the SIGUSR1 and SIGUSR2 signal for its own purpose. It is forbidden to use these two signals in a Linux Tango device server. The Tango core classes will refuse to install something for these two signals.

Nevertheless, the Linux thread library is not fully POSIX compliant about thread and signal management. The POSIX specification says that an asynchronous signal must be delivered to one of the thread of the program which does not block the signal (it is not specified which). Using this Linux thread library, the signal is delivered to the thread it is been sent to, based on the PID

of the thread. If that thread is currently blocking the signal, the signal remains pending...This is a problem for Tango device server under Linux using the *alarm()* system call. In this case, the system will send the signal to the device server main thread and not to the device server signal management thread. A special case of the *register_signal* method (detailed in the next sub-chapter) have been developped for such case. This is available only for Linux.

Using Solaris

There is no restriction on the signal to be used.

Using HP-UX

Signal and thread correctly works only for asynchronous signal. Asynchronous signals are generated asynchronously with respect to the instruction stream, and thus may arrive at any time regadless of what the process is currently doing. SIGTERM, which is typically sent by another process in order to terminate the target process, is an example of an asynchronous signal. Within a Tango device server, you can use only asynchronous signal. The Tango core classes will refuse to install something for synchronous signals. The following list is all the synchronous signal : SIGABRT, SIGKILL, SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, SIGSYS, SIDPIPE, SIGSTOP. These signals can't be used in HP-UX device server. The SIGVTALRM signal is used internally by the HP-UX package and must not be used in a Tango device server. The ITIMER_VIRTUAL timer cannot be used also through the setitimer() call. However, ITIMER_REAL or ITIMER_PROF may be used.

9.1.1 Using signal

It is possible for C++ device server to receive signals from drivers or other processes. The TDSOM supports receiving signal at two levels: the device level and the class level. Supporting signal at the device level means that it is possible to specify interest into receiving signal on a device basis. This feature is supported via three methods defined in the DeviceImpl class. These methods are called *register_signal*, *unregister_signal* and *signal_handler*.

The ***register_signal*** method has one parameter which is the signal number. This method informs the device server signal system that the device want to be informed when the signal passed as parameter is received by the process. There is a special case for Linux as explained in the previous sub-chapter. It is possible to register a signal to be executed in the a signal handler context (with all its restrictions). This is done with a second parameter to this *register_signal* method. This second parameter is simply a boolean data. If it is true, the signal_handler will be executed in a signal handler context in the device server main thread. A default value (false) has been defined for this parameter.

The ***unregister_signal*** method also have an input parameter which is the signal number. This method removes the device from the list of object which should be warned when the signal is received by the process.

The ***signal_handler*** method is the method which is triggered when a signal is received if the corresponding *register_signal* has been executed. This method is defined as virtual and can be redefined by the user. It has one input argument which is the signal number.

The same three methods also exist in the DeviceClass class. Their action and their usage are similar to the DeviceImpl class methods. Installing a signal at the class level does not mean that all the device belonging to this class will receive the signal. This only means that the *signal_handler* method of the DeviceClass instance will be executed. This is useful if an action has to be executed once for a class of devices when a signal is received.

The following code is an example with our stepper motor device server configured via the database to serve three motors. These motors have the following names : id04/motor/01, id04/motor/02 and id04/motor/03. The signal SIGALRM (alarm signal) must be propagated only to the motor number 2 (id04/motor/02)

```
 1  void StepperMotor::init_device()
 2  {
 3          cout << "StepperMotor::StepperMotor() create motor " << dev_name << endl;
 4
 5          long i;
 6
 7          for (i=0; i< AGSM_MAX_MOTORS; i++)
 8          {
 9                  axis[i] = 0;
10                  position[i] = 0;
11                  direction[i] = 0;
12          }
13
14          if (dev_name == "id04/motor/02")
15                  register_signal(SIGALRM);
16  }
17
18  StepperMotor::~StepperMotor()
19  {
20          unregister_signal(SIGALRM);
21  }
22
23  void StepperMotor::signal_handler(long signo)
24  {
25          cout2 << "Inside signal handler for signal " << signo << endl;
26
27  //      Do what you want here
28
29  }
```

The *init_device* method is modified.

Line 14-15 : The device name is checked and if it is the correct name, the device is registered in the list of device wanted to receive the SIGALARM signal.

The destructor is also modified

Line 20 : Unregister the device from the list of devices which should receives the SIGALRM signal. Note that unregister a signal for a device which has not previously registered its interest for this signal does nothing.

The *signal_handler* method is redefined

Line 25 : Print signal number

Line 27 : Do what you have to do when the signal SIGALRM is received.

If all devices must be warned when the device server process receives the signal SIGALRM, removes line 14 in the *init_device* method.

9.1.2 Exiting a device server gracefully

A device server has to exit gracefully by unregistering itself from the database. The necessary action to gracefully exit are automatically executed on receiption of the following signal :

- SIGINT, SIGTERM, SIGHUP and SIGQUIT for device server running on HP-UX, Solaris or Linux

- SIGINT, SIGTERM, SIGABRT and SIGBREAK for device server running on Windows-NT

This does not prevents device server to also register interest at device or class levels for those signals. The user installed *signal_handler* method will first be called before the gracefull exit.

9.2 Inheriting

This sub-chapter details how it is possible to inherit from an existing device pattern implementation. As the device pattern includes more than a single class, inheriting from an existing device pattern needs some explanations.

Let us suppose that the existing device pattern implementation is for devices of class A. This means that classes A and AClass already exists plus classes for all commands offered by device of class A. One new device pattern implementation for device of class B must be written with all the features offered by class A plus some new one. This is easily done with the inheritance. Writing a device pattern implementation for device of class B which inherits from device of class A means :

- Write the BClass class
- Write the B class
- Write B class specific commands
- Eventually redefine A class commands

9.2.1 Using C++

The miscellaneous code fragments given belows detail only what has to be updated to support device pattern inheritance

Writing the BClass

As you can guess, BClass has to inherit from AClass. The *command_factory* method must also be adapted.

```
 1  namespace B
 2  {
 3
 4  class BClass : public A::AClass
 5  {
 6  .....
 7  }
 8
 9  BClass::command_factory()
10  {
11          A::AClass::command_factory();
12
13          command_list.push_back(....);
14  }
15
16  } /* End of B namespace */
```

Line 1 : Open the B namespace
Line 4 : BClass inherits from AClass which is defined in the A namespace.
Line 11 : Only the *command_factory* method of the BClass will be called at start-up. To create the AClass commands, the *command_factory* method of the AClass must also be executed. This is the reason of the line
Line 13 : Create BClass commands

Writing the B class

As you can guess, B has to inherits from A.

```
 1  namespace B
 2  {
 3
 4  class B : public A:A
 5  {
 6          .....
 7  };
 8
 9  B::B(Tango::DeviceClass *cl,const char *s):A::A(cl,s)
10  {
11          ....
12          init_device();
13  }
14
15  void B::init_device()
16  {
17          ....
18  }
19
20  } /* End of B namespace */
```

Line 1 : Open the B namespace.
Line 4 : B inherits from A which is defined in the A namespace
Line 9 : The B constructor calls the right A constructor

Writing B class specific command

Noting special here. Write these classes as usual

Redefining A class command

It is possible to redefine a command which already exist in class A **only if the command is created using the inheritance model** (but keeping its input and output argument types). The method which really execute the class A command is a method implemented in the A class. This method must be defined as **virtual.** In class B, you can redefine the method executing the command and implement it following the needs of the B class.

9.2.2 Using Java

The miscellaneous code fragments given belows detail only what has to be updated to support device pattern inheritance

Writing the BClass

As you can guess, BClass has to inherit from AClass. Some change must be done in the definition of the *init* and *instance* methods. The *command_factory* method must also be adapted.

```
1  public class BClass extends AClass implements TangoConst
2  {
3          public static AClass init(String name) throws DevFailed
4          {
5
6          }
7
8          public static AClass instance()
9          {
10
11         }
12
13         public void command_factory()
14         {
15                 super.command_factory();
16
17                 command_list.addElement(....);
18         }
19 };
```

Line 1 : BClass inherits from AClass and implements TangoConst interface

Line 3 : The return data type of the *init* method must be the same as the type defines in the AClass (therefore a reference to AClass) oherwise, the compiler complains. BClass inherits from AClass and a reference to a BClass is also a reference to the AClass

Line 8 : The return data type of the *instance* method must also be adapted as explained for the *init* method

Line 15 : Only the *command_factory* method of the BClass will be called at start-up. To create the AClass commands, the *command_factory* method of the AClass must also be executed. This is the reason of the line

Line 17 : Create BClass commands

Writing the B class

As you can guess, B has to inherits from A. The *init_device* method must be adapted, the constructor has to be modified and an instance variable must be added

```
1  public class B extends A implements TangoConst
2  {
3          boolean constructed = false;
4
5          A(DeviceClass cl,String s)
6          {
7                  super(cl,s);
8                  constructed = true;
9                  ...
```

```
10                  init_device();
11          }
12
13          public void init_device()
14          {
15                  if (constructed == false)
16                  {
17                          return;
18                  }
19                  super.init_device();
20
21                  ...
22          }
23  };
```

Line 1 : B inherits from A and implements TangoConst interface

Line 3 : A boolean initialised to false is added as instance variable

Line 8 : The constructor is modified to set the constructed boolean to true after all the super classes have been created and before the call to the *init_ device* method.

Line 15-18 : The *init_ device* method immediately returns if the constructed boolean is false (if the super classes are not correctly created)

Line 19 : The *init_ device* method of class A is called

Writing B class specific command

Noting special here. Write these classes as usual

Redefining A class command

It is possible to to redefine a command which already exist in class A **only if the command is created using the inheritance model** (but keeping its input and output argument types). The method which really execute the class A command is a method implemented in the A class. With Java, it is possible to redefine all methods except those which are declared as "final". Therefore, in class B, you can redefine the method executing the command and implement it following the needs of the B class. The following is an example for a command xxx which is programmed to call a *my_ cmd* method[1].

```
1  public class A extends DeviceImpl implements TangoConst
2  {
3          public void my_cmd(long input)
4          {
5          }
6  }
7
8  public class B extends A implements TangoConst
9  {
10         public void my_cmd(long input)
11         {
12         }
13 }
```

[1] In the command *execute* method

Line 3 : The *my_cmd* method is defined in class A
Line 10 : The *my_cmd* method is redefined in class B

Inside the device pattern, the device object is created as an instance of class B[2]. Java will call the *my_cmd* method of the B class when the command is received. It is still possible to call the *my_cmd* method of the A class with the help of the Java "super" keyword inside the code of the *my_cmd* method of the B class.

9.3 Using another device pattern implementation within the same server

It is often necessary that inside the same device server, a method executing a command needs a command of another class to be executed. For instance, a device pattern implementation for a device driven by a serial line class can use the command offered by a serial line class embedded within the same device server process. To execute one of the command (or any other CORBA operations/attributes) of the serial line class, just call it as a normal client will do with the *command_inout* CORBA operation. The ORB will recognize that all the devices are inside the same process and will execute the *command_inout* as a local call. To retrieve the reference to the serial line device, you can use the *get_device_by_name()* method of the Tango::Util class.

```
1       string dev_name("sys/serial/2");
2
3       Tango::DeviceImpl *dev;
4       Tango::Util *tg = Tango::Util::instance();
5       dev = tg->get_device_by_name(dev_name.c_str());
6
7       Tango::DevState d_state = dev->state();
8       cout2 << "The serial line device state is " << d_state << endl;
```

Line 1 : The Tango name of the serial line device embedded within the same server
Line 5 : A reference to the serial line device is retrieved
Line 7 : Get serial line device state

[2]By the *device_factory* method of the BClass class

Appendix A

Reference part

This chapter is only part of the TANGO device server reference guide. To get reference documentation about the C++ library classes, see [6]. To get reference documentation about the Java classes, also see [6].

A.1 Device parameter

A black box, a device description field, a device state and status are associated with each TANGO device.

A.1.1 The device black box

The device black box is managed as a circular buffer. It is possible to tune the buffer depth via a device property. This property name is

device name/blackbox_depth

A default value is hard-coded to 25 if the property is not defined. This black box depth property is retrieved from the Tango property database during the device creation phase.

A.1.2 The device description field

There are two ways to intialise the device description field.

- At device creation time. Some constructors of the DeviceImpl class supports this field as parameter. If these constructor are not used, the device description field is set to a default value which is *A Tango device.*
- With a property. A description field defines with this method overrides a device description defined at construction time. The property name is

  device name/description

A.1.3 The device state and status

Some constructors of the DeviceImpl class allows the initialisation of device state and/or status or device creation time. If these fields are not defined, a default value is applied. The default state is Tango::UNKOWN, the default status is *Not Initialised.*

A.2 Device class parameter

A device documentation field is also defined at Tango device class level. It is defined as Tango device class level because each device belonging to a Tango device class should have the same behaviour and therefore the same documentation. This field is store in the DeviceClass class. It is possible to set this field via a class property. This property name is

class name/doc_url

and is retrieved when instance of the DeviceClass object is created. A default value is defined for this field.

A.3 The device black box

This black box is a help tool to ease debugging session for a running device server. The TANGO core software records every device request in this black box. A tango client is able to retrieve the black box contents with a specific CORBA operation availabble for every device. Each black box entry is returned as a string with the following information :

- The date where the request has been executed by the device. The date format is dd/mm/yyyy hh24:mi:ss:SS (The last field is the second hundredth number).
- The type of CORBA requests. In case of attributes, the name of the requested attribute is returned. In case of operation, the operation type is returned. For "command_inout" operation, the command name is returned.
- The client host name

A.4 Automatically added commands

As already mentionned in this documentation, each Tango device supports at least two commands which are DevState and DevStatus. The following array details command input and output data type

Command name	Input data type	Output data type
DevState	void	Tango::DevState
DevStatus	void	Tango::DevString

A.4.1 The DevState command

This command gets the device state (stored in its *device_state* data member) and returns it to the caller. The device state is a variable of the Tango_DevState type (packed into a CORBA Any object when it is returned by a command)

A.4.2 The DevStatus command

This command gets the device status (stored in its *device_status* data member) and returns it to the caller. The device status is a variable of the string type.

A.5 DServer class device commands

As already explained in 4.7.2, each device server process has its own Tango device. This device supports the two commands previously described plus 9 commands which are DevRestart, DevRestartServer, DevQueryClass, DevQueryDevice, DevKill, DevSetTraceLevel, DevGetTraceLevel, DevSetTraceOutput and DevGetTraceOutput. The following table give all commands input and output data types

Command name	Input data type	Output data type
DevState	void	Tango::DevState
DevStatus	void	Tango::DevString
DevRestart	Tango::DevString	void
DevRestartServer	void	void
DevQueryClass	void	Tango::DevVarStringArray
DevQueryDevice	void	Tango::DevVarStringArray
DevKill	void	void
DevSetTraceLevel	Tango::DevLong	void
DevGetTraceLevel	void	Tango::DevLong
DevSetTraceOutput	Tango::DevString	void
DevGetTraceOutput	void	Tango::DevString

The device description field is set to "A device server device".

A.5.1 The DevState command

This device state is always set to ON

A.5.2 The DevStatus command

This device status is always set "The device is ON"

A.5.3 The DevRestart command

The DevRestart command restart a device. The name of the device to be re-started is the command input parameter. The command destroys the device by calling its desctructor and re-create it from its constructor.

A.5.4 The DevRestartServer command

The DevRestartServer command restarts all the device pattern(s) embedded in the device server process. Therefore, all the devices implemented in the server process are destroyed and re-built[1]. The network connection between client(s) and device(s) implemented in the device server process is destroyed and re-built.

Executing this command allows a complete restart of the device server without stopping the process.

A.5.5 The DevQueryClass command

This command returns to the client the list of Tango device class(es) embedded in the device server. It returns only class(es) implemented by the device server programmer. The DServer device class name (implemented by the TANGO core software) is not returned by this command.

[1] Their black-box is also destroyed and re-built

A.5.6 The DevQueryDevice command

This command returns to the client the list of device name for all the device(s) implemented in the device server process. The name of the DServer class device is not returned by this command.

A.5.7 The DevKill command

This command stops the device server process. In order that the client receives a last answer from the server, this command starts a thread which will after a short delay, kills the device server process.

A.5.8 The DevSetTraceLevel command

This command allows a remote control of the device server trace level. Its input data is the new trace level. Setting the trace level to a value lower than 0 has the same effect than a 0 level. Setting the trace level to a value higher than 4 has the same effect than a 4 level.

A.5.9 The DevGetTraceLevel command

This command simply returns the device server process trace level.

A.5.10 The DevSetTraceOutput command

This command allows a remote control of where the device server process send all its trace output. The command input data is a file name. This file is created if it does not already exist. Otherwise, it is truncated to a zero length and new output are appended to it. It is always possible to reset the process output to the value is has at startup time by using the predefined string **Tango_InitialOutput** as data file name.

A.5.11 The DevGetTraceOutput command

This command returns the process output file name. If the process is using its startup time output, the returned string is set to the predefined **Tango_InitialOutput** string.

A.6 C++ specific

A.6.1 The Tango master include file (tango.h)

Tango has a master include file called

tango.h

This master include file includes the following files :

- C++ language include file : **typeinfo**
- CORBA include file : **CORBA.h**
- C++ streams include file :
  - **iostream**, **sstream** and **fstream** for Windows NT
  - **iostream.h**, **strstream.h** and **fstream.h** for the other operating systems
- Some standard C++ library include files : **string** and **vector**
- The main include file generated by the CORBA IDL compiler : **idl/tango.h**

- The Tango database API include file : **dbapi.h**

- A list of other Tango include files : **tango_const.h**, **tango_config.h**, **utils.h**, **device.h**, **command.h**, **except.h**, **attrmanip.h** and **dserver.h**

A.6.2 Tango specific types

Operating system free type

Some data type used in the TANGO core software are not the same under UNIX like operating system and Windows NT. In order to have less "#ifdef" in the source code, some Tango types have been defined. They are described in the following table.

Type name	Unix like	Windows NT
TangoSys_MemStream	ostrstream	ostringstream
TangoSys_Pid	pid_t	int
TangoSys_Cout	_IO_ostream_withassign	ostream

These types are defined in the tango_config.h file

Template command model related type

As explained in 6.8, command created with the template command model uses static casting. Many type definition have been written for these casting.

Class name	Command allowed method (if any)	Command execute method
TemplCommand	Tango::StateMethodPtr	Tango::CmdMethPtr
TemplCommandIn	Tango::StateMethodPtr	Tango::CmdMethPtr_xxx
TemplCommandOut	Tango::StateMethodPtr	Tango::xxx_CmdMethPtr
TemplCommandInOut	Tango::StateMethodPtr	Tango::xxx_CmdMethPtr_yyy

The **Tango::StateMethPtr** is a pointer to a method of the DeviceImpl class which returns a boolean and has one parameter which is a reference to a const CORBA::Any obect.

The **Tango::CmdMethPtr** is a pointer to a method of the DeviceImpl class which returns nothing and needs nothing as parameter.

The **Tango::CmdMethPtr_xxx** is a pointer to a method of the DeviceImpl class which returns nothing and has one parameter. xxx must be set according to the method parameter type as described in the next table

Tango type	short cut (xxx)
Tango::DevBoolean	Bo
Tango::DevShort	Sh
Tango::DevLong	Lg
Tango::DevFloat	Fl
Tango::DevDouble	Db
Tango::DevUshort	US
Tango::DevULong	UL
Tango::DevString	Str
Tango::DevVarCharArray	ChA
Tango::DevVarShortArray	ShA
Tango::DevVarLongArray	LgA
Tango::DevVarFloatArray	FlA
Tango::DevVarDoubleArray	DbA
Tango::DevVarUShortArray	USA
Tango::DevVarULongArray	ULA
Tango::DevVarStringArray	StrA
Tango::DevVarLongStringArray	LSA
Tango::DevVarDoubleStringArray	DSA
Tango::DevState	Sta

For instance, a pointer to a method which takes a Tango::DevVarStringArray as input parameter must be statically casted to a Tango::CmdMethPtr_StrA, a pointer to a method which takes a Tango::DevLong data as input parameter must be statically casted to a Tango::CmdMethPtr_Lg.

The **Tango::xxx_CmdMethPtr** is a pointer to a method of the DeviceImpl class which returns data of one of the Tango type and has no input parameter. xxx must be set according to the method return data type following the same rules than those described in the previous table. For instance, a pointer to a method which returns a Tango::DevDouble data must be statically casted to a Tango::Db_CmdMethPtr.

The **Tango::xxx_CmdMethPtr_yyy** is a pointer to a method of the DeviceImpl class which returns data of one of the Tango type and has one input parameter of one of the Tango data type. xxx and yyy must be set according to the method return data type and parameter type following the same rules than those described in the previous table. For instance, a pointer to a method which returns a Tango::DevDouble data and which takes a Tango::DevVarLongStringArray must be statically casted to a Tango::Db_CmdMethPtr_LSA.

All those type are defined in the tango_const.h file.

A.6.3 Tango device state code

The Tango::DevState type is a C++ enumeration starting at 0. The code associated with each state is defined in the following table.

State name	Value
Tango::ON	0
Tango::OFF	1
Tango::CLOSE	2
Tango::OPEN	3
Tango::INSERT	4
Tango::EXTRACT	5
Tango::MOVING	6
Tango::STANDBY	7
Tango::FAULT	8
Tango::INIT	9
Tango::RUNNING	10
Tango::ALARM	11
Tango::DISABLE	12
Tango::UNKNOWN	13

A strings array called **Tango::DevStateName** can be used to get the device state as a string. Use the Tango device state code as index into the array to get the correct string.

A.6.4 Tango data type

A "define" has been created for each Tango data type. This is summarised in the following table

Type name	Type code	Value
Tango::DevBoolean	Tango::DEV_BOOLEAN	1
Tango::DevShort	Tango::DEV_SHORT	2
Tango::DevLong	Tango::DEV_LONG	3
Tango::DevFloat	Tango::DEV_FLOAT	4
Tango::DevDouble	Tango::DEV_DOUBLE	5
Tango::DevUShort	Tango::DEV_USHORT	6
Tango::DevULong	Tango::DEV_ULONG	7
Tango::DevString	Tango::DEV_STRING	8
Tango::DevVarCharArray	Tango::DEVVAR_CHARARRAY	9
Tango::DevVarShortArray	Tango::DEVVAR_SHORTARRAY	10
Tango::DevVarLongArray	Tango::DEVVAR_LONGARRAY	11
Tango::DevVarFloatArray	Tango::DEVVAR_FLOATARRAY	12
Tango::DevVarDoubleArray	Tango::DEVVAR_DOUBLEARRAY	13
Tango::DevVarUShortArray	Tango::DEVVAR_USHORTARRAY	14
Tango::DevVarULongArray	Tango::DEVVAR_ULONGARRAY	15
Tango::DevVarStringArray	Tango::DEVVAR_STRINGARRAY	16
Tango::DevVarLongStringArray	Tango::DEVVAR_LONGSTRINGARRAY	17
Tango::DevVarDoubleStringArray	Tango::DEVVAR_DOUBLESTRINGARRAY	18
Tango::DevState	Tango::DEV_STATE	19

For command which do not take input parameter, the type codeTango::DEV_VOID (value = 0) has been defined.

A strings array called **Tango::CmdArgTypeName** can be used to get the data type as a string. Use the Tango data type code as index into the array to get the correct string.

A.7 Java specific

A.7.1 Packages

All the Tango core classes are bundled in the a Java package called **fr.esrf.TangoDs**. All the classes generated by the IDL compiler are bundled in a Java package called **fr.esrf.Tango**. All the CORBA related classes are stored in a package called **org.omg.CORBA**. The two package Tango and TangoDs are stored in the same jar file called Tango.jar. The org.omg.CORBA package is stored in a file called OB.jar.

Bibliography

[1] OMG home page - http://www.omg.org

[2] "Advanced CORBA programming with C++" by M.Henning and S.Vinosky (Addison-Wesley 1999)

[3] TANGO home page - http://www.esrf.fr/tango/index.html

[4] MySQL home page - http://www.mysql.com

[5] "MySQL and mSQL" by Randy Jay Yarger, George Reese and Tim King (O'Reilly 1999)

[6] TANGO classes on-line documentation - http://www.esrf.fr/tango/tango_classes/index.html

[7] "C++ programming language" third edition by Stroustrup (Addison-Wesley)

[8] "Design Patterns" by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley 1995)

[9] OOC home page - http://www.ooc.com

[10] The Common Object Request Broker: Architecture and Specification Revision 2.3 available from OMG home page - http://www.omg.org

[11] Java Pro - June 1999 : Plugging memory leak by Tony Leung

[12] CVS WEB page - http://www.cyclic.com

[13] POGO home page - http://www.esrf.fr/tango/tango_doc/pogo_doc/index.html

Index