The Abstract Device Pattern

or

How to implement multiple classes of the same device type in TANGO

Andy Gotz (andy.gotz@esrf.fr)

14/3/2005

Abstraction gives you the power to take care *of the big picture and worry about the details later*.

1 Introduction

Object oriented device control offers a wonderful opportunity to implement abstraction via standard device classes, which implement standard device interfaces for families of devices e.g. motors, power supplies, function generators etc. By standardising the abstractions we can decouple client applications from device implementations. This means more generic code, which in turn means more code sharing. TANGO is an object oriented control system and therefore ideally placed to realise this dream. In order to avoid multiple interface definitions for the same family of devices it is necessary to agree on a standard interface for each family of devices. This is usually the hardest part; this paper offers some advice on going about abstraction. The standard interface is referred to as the Abstract Device. Once we have defined the interface we need a method for implementing them in TANGO device servers so that all sub-classes of that device type are guaranteed to implement the interface. This paper makes a proposal on how to implement such an interface in TANGO device servers using the Abstract Device design pattern.

2 Abstract Device Pattern

The Abstract Device pattern is derived from the Abstract Factory pattern. Let us remind ourselves what the Abstract pattern is:

> *"Provide an interface for creating families of related or dependent objects without specifying their concrete classes." (Gamma, E., R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented*

Software. Reading, MA: Addison-Wesley, 1995)

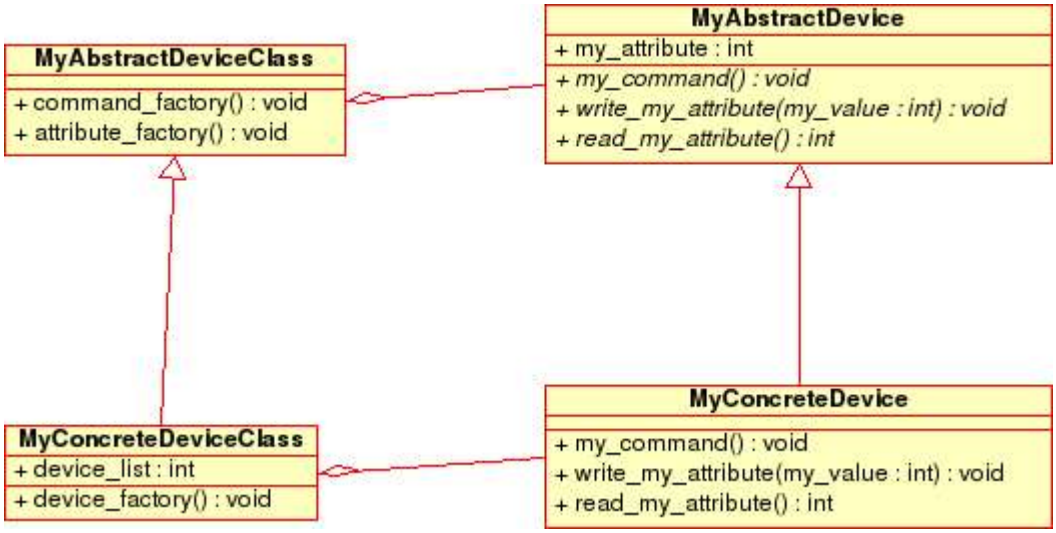
The main idea of the abstract pattern is to provide a common interface to a wide range of concrete implementations. The abstract factory will return an abstract interface to a concrete implementation of the desired object.

The Abstract Device pattern can be summarised as follows:

> *"Provide an interface for creating families of devices which represent related devices without specifying their concrete implementation e.g. Motor, PowerSupply, FunctionGenerator etc."*

The main motivation of the Abstract Device pattern is:

1.That same client applications can communicate with multiple concrete implementations of related devices
2.To define standard interfaces to a families of devices
3.To avoid duplication of similar but incompatible interfaces for related devices e.g. two power supplies from different suppliers,
4.Guide new programmer's on what the essential methods need to be implemented for a family of devices



3 Implementing the Code

How to go about implementing an abstract device class in TANGO. The technique described here has been derived from a paper written by Emmanuel Taurel. Implementations for TANGO version 5 are presented.

Where to start? First identify what family your device belongs to. Then define what constitutes the essence of the commands and attributes of that family of devices. For more help on this part refer to the next section. Once you have a basic definition of the

abstract interface you can start generating the abstract class.

3.1 In C++

3.1.1 The Abstract Device class

The abstract class will be the super class for all concrete implementations of that device type. The abstract class will not implement any device specific code – the abstract class is ONLY an interface. The reason behind this is to concentrate on the interfaces and not get bogged down in details of what should go in the code. Code should be delegated to the subclasses. The abstract class has to be a standard TANGO device server. So what better way to generate it than using POGO? Use pogo to generate your abstract device server with all the commands and attributes predefined. You will get the usual set of files e.g. in C++ you should have:

MyAbstractDevice.h, MyAbstractDevice.cpp,
MyAbstractDeviceClass.h, MyAbstractDeviceClass.cpp.

The next step is to implement the commands and attributes in the abstract class so that all concrete implementations which are derived from the Abstract Device class are forced to implement the standard set of commands and attributes for that abstract class. There are many ways of doing this, my solution is described here.

3.1.2 Commands

For commands modify all command method signatures in the MyAbstractDevice.h to be void virtual (in the future pogo will do this for you). This way the concrete implementation must override the command methods. *This modification has to be done every time pogo is used to regenerate the source code for the Abstract Device super class.* Obviously it would be better to define the methods as pure virtual which will then force the concrete subclasses to implement these commands. But this has not been done because pogo overwrites the virtual and regenerates the command methods every time it generates the code. In the concrete implementation MyConcreteDevice.h define the commands as virtual and implement them to do the real work e.g. accessing the hardware.

3.1.3 Attributes

Attributes are implemented differently in TANGO V4 and V5. In V5 every attribute has a read and write method. In V4 there is one read and one write method for all attributes and a big switch. The best solution I have found, which is V4 and V5 compatible is to define read and write methods in V4 which resemble the ones in V5

i.e. read_my_attribute() and write_my_attribute().Then when you move your server to V5 you will already have the attribute methods done. These methods should be defined as pure virtual in the MyAbstractDevice.h file to enforce their implementation in the concrete class e.g.

```
virtual void write_my_attribute()=0;
virtual Tango::DevDouble read_my_attribute()=0;
```

Call these read and write methods from the write_attr_harware() and read_attr() functions respectively in the Abstract Device class:

```
if (attr_name == "my_attribute")
{
    //    Add your own code here
    write_my_attribute(attr);

}
```

and

```
if (attr_name == "my_attribute")
{
    //    Add your own code here
    read_my_attribute(attr);
}
```

3.1.4 Abstract Device Class class

The AbstractDeviceClass.cpp needs one line to be modified in order to compile. Because the getters and setters are defined as pure virtual methods it is not possible to do a new AbstractDevice() in the device_factory(). Comment this line out (in the future pogo will do this for you):

```
//device_list.push_back(new MyAbstractDevice(this,*devlist_ptr)[i]));
```

3.1.5 The Concrete Device class

The concrete device class implements a real case of the device i.e. actually talks to real hardware if this is a hardware device server. The concrete device class inherits from of the abstract device class i.e. respectively the MyAbstractDevice and the MyAbstractDeviceClass classes. It is generated using pogo the first time but after that it must be maintained manually.

The best way to start is to generate a tango device server using pogo and the same definition for the commands and attributes as for the first version of the MyAbstractDevice class. Derive the concrete class and class class from the abstract classes respectively:

```
namespace MyConcreteDevice {

class MyConcreteDevice: public MyAbstractDevice::MyAbstractDevice
```

and

```
namespace MyConcreteDevice {

class MyConcreteDeviceClass : public
MyAbstractDevice::MyAbstractDeviceClass
```

Modify the MyConcreteDevice.h and MyConcreteDevice.cpp files to implement the command and attribute writers and readers defined as virtual in the abstract class e.g.

```
    virtual void my_command(void);
    virtual void read_my_attribute (Tango::Attribute& attr);
    virtual void write_my_attribute(Tango::Attribute& attr);
```

The command and attribute methods implementations in MyConcreteDevice.cpp will access the real hardware.

Modify the concrete class class to call the abstract class' command and attribute factories:

```
void MyConcreteDeviceClass::command_factory()
{
     MyAbstractDevice::MyAbstractDeviceClass::command_factory();

     //     add any private commands here

     /*
       command_list.push_back(new PrivateCmd("PrivateCmd",
           Tango::DEV_VOID, Tango::DEV_VOID,
           "No argin",
           "No argout",
           Tango::OPERATOR));
      */
}
void MyConcreteDeviceClass::attribute_factory(vector<Tango::Attr *>
&att_list)
{
```

```
	MyAbstractDevice::MyAbstractDeviceClass::attribute_factory(att_list);

	// add any private attributes here

	/*
	Tango::Attr	*private_attr = new Tango::Attr("private_attr",
					Tango::DEV_DOUBLE,
					Tango::READ_WRITE);
	att_list.push_back(private_attr);
	*/
}
```

Add any specific commands needed by the concrete device class in the command and attribute factories.

3.1.6 Attribute properties

The attributes defined by the Abstract class need to be customised for each concrete class in terms of properties e.g. the mininum and maximum values of all concrete devices are not the same. One way of doing this is defining a method :

```
virtual void get_attribute_default_properties( const	char*,
				Tango::UserDefaultAttrProp &)=0;
```

in the Abstract Class class which has to be implemented in every concrete class class. This method will set the concrete class device specific default properties. Here is an example of such a method :

```
void Agilent33x20AClass::get_attribute_default_properties(const char
*attr_name, Tango::UserDefaultAttrProp &attr_prop)
{
	string attr_name_str(attr_name);

	//	Attribute : frequency
	if (cmp_nocase(attr_name_str,"frequency") == 0)
	{
		attr_prop.set_format("%9.3f");
		attr_prop.set_max_value("15000000");
		attr_prop.set_min_value("0.0001");
		return;
	}
}
```

This method has to be called from the AbstractClass class attribute_factory() method just before you call the set_default properties (this could be done by pogo in the future) e.g. :

```
 void SignalGeneratorClass::attribute_factory(vector<Tango::Attr *> &att_list)
{
	//	Attribute : frequency
	Tango::Attr	*frequency =
			new Tango::Attr("frequency", Tango::DEV_DOUBLE,
			Tango::READ_WRITE);
	Tango::UserDefaultAttrProp	frequency_prop;
	frequency_prop.set_label("frequency");
	frequency_prop.set_unit("Hz");
	frequency_prop.set_standard_unit("Hz");
	frequency_prop.set_display_unit("Hz");
	frequency_prop.set_format("%3f");
	frequency_prop.set_description("stop sweep frequency");
	get_attribute_default_properties("frequency", frequency_prop);
	frequency->set_default_properties(frequency_prop);
	att_list.push_back(frequency);
	  .
	  .
	  .
}
```

3.1.7 Device Properties

Device properties can be handled in the standard TANGO way. Properties which are required by all members of the abstract class should be implemented in the Abstract class. Properties which are specific to concrete implementations should be implemented in the concrete class.

3.1.8 Testing

Write the unit tests using PyUnit, Junit or CppUnit frameworks whichever one you prefer.

Compile, debug, test, document and publish !

3.2 In Java

How to do implement the Abstract Device pattern in Java? More work need to be done here. Java interfaces seems the best way to go but an example is required.

4 Changes to POGO

If this method of writing device servers is generally accepted then it would be desirable that POGO supports it. For this Pogo has to be modified to support the

notion of Abstract and Concrete class. It needs to generate code which does not need to be modified by hand e.g. generating the correct class hierarchy and not making all classes derive from DeviceImpl. It should generate pure virtual functions where necessary to enforce their implementation in the concrete classes. It should provide a list of known abstract classes for the program writer to choose from.

NOTE: Pogo is currently being modified to support abstract devices. Please contact verdier@esrf.fr to get the latest version and details on how to use it.

5 How to abstract a device?

How to go about defining the abstract device class? This is the first step in defining what constitutes the abstract device for the family of devices which your concrete device belongs to. Often this is also the step where many attempts falter and fail to deliver. The main idea of the Abstract Device pattern is to define the most common interface for all devices which are members of this family of devices. The Abstract Device pattern is in many ways similar to the Union design pattern. The superclass represents an abstract representation of the union of all the subclasses. Due to this polymorphism, the subclasses can thus be used wherever the superclass is required.

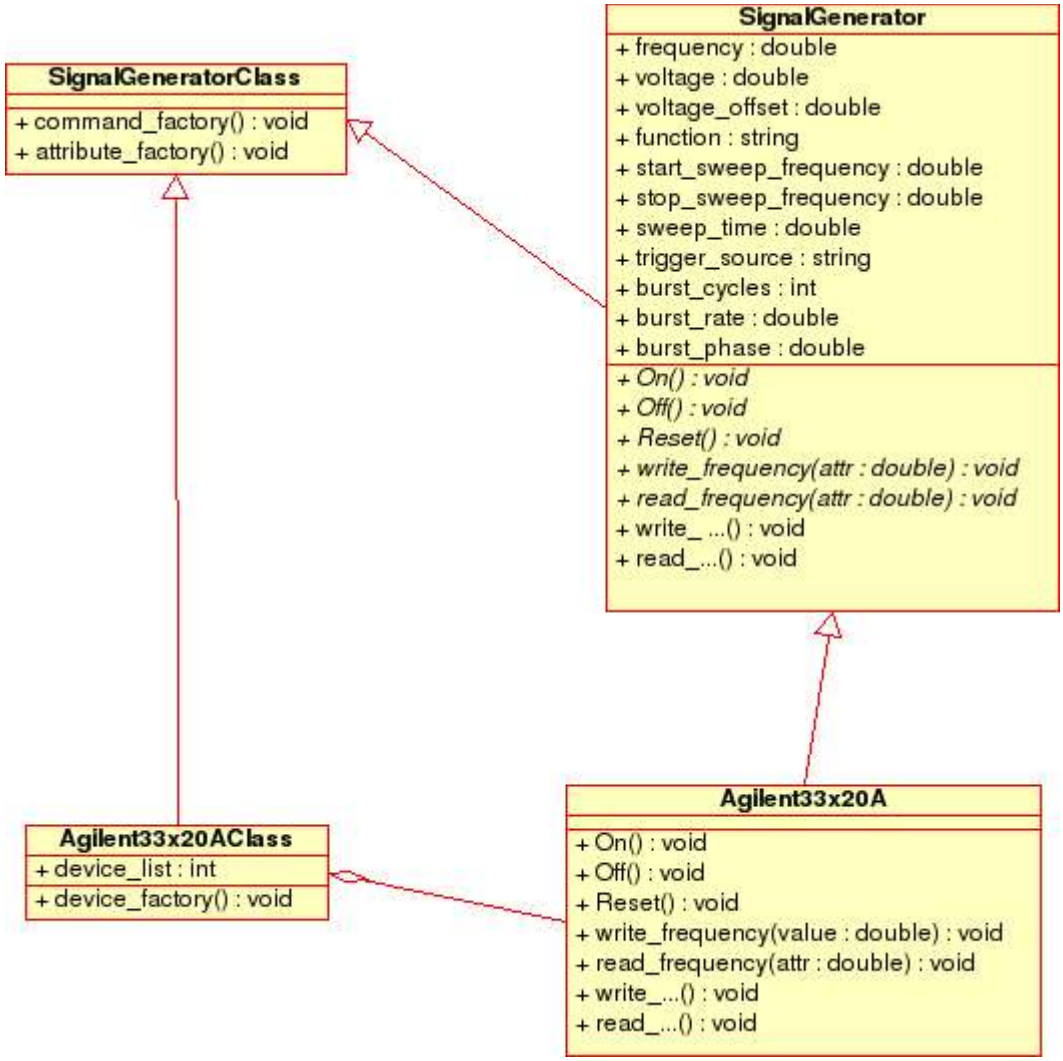
When defining the Abstract Device class the following principles can help:

1. include only the main features which best represent the members of the superclass
2. do not get bogged down in detail, if you find the discussion is taking too long stop it
3. if in doubt, throw it out
4. the important thing is to arrive at a result
5. a bad abstract device class will eventually be replaced by a better one

6 Examples

6.1 Signal Generator

An example of an Abstract Device which uses the above pattern is the SignalGenerator. The original project was to write a TANGO device server for the Agilent 33120A signal generator. I found out that a TANGO device server had already been written for the Agilent 33220A signal generator. Looking at the code I realized the two devices (the 33120A and the 33220A) had a lot in common. On the suggestion of Jean-Michel Chaize I decided to generalize the code even further and define a superclass called SignalGenerator which would represent the archetype of a family of devices called signal generators. I came up with the following class diagram:

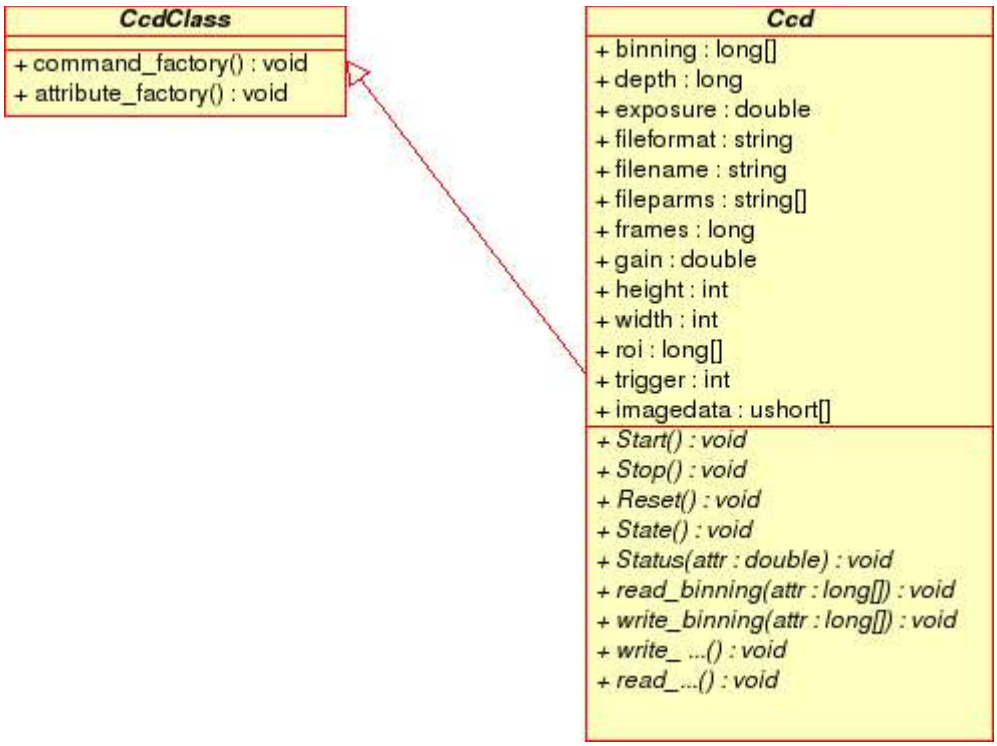


The source code for the SignalGenerator class can be found on SourceForge in the CVS repository of the tango-ds project:

http://cvs.sourceforge.net/viewcvs.py/tango-ds/Instrumentation/

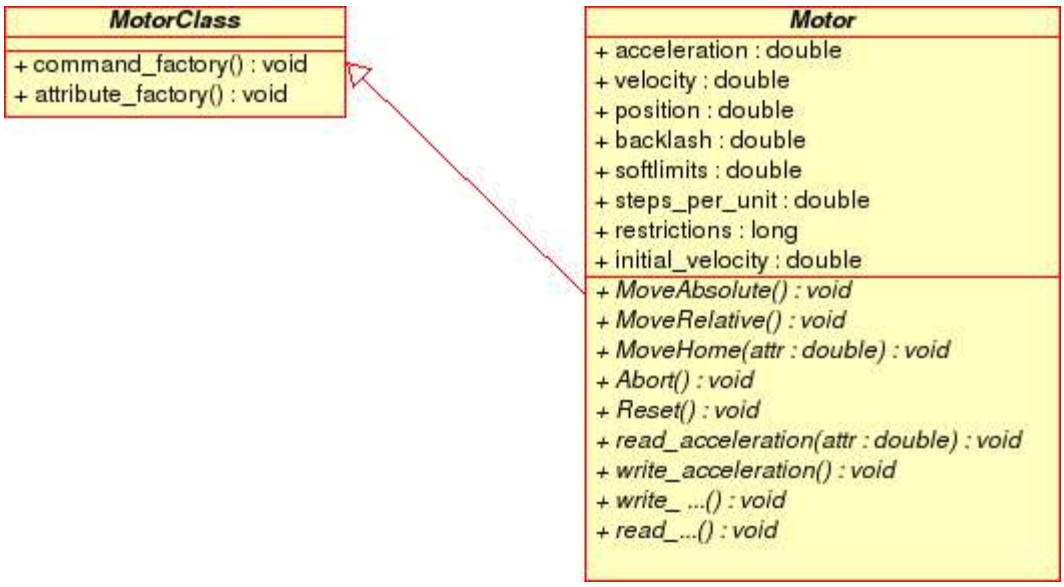
6.2 Ccd

Example of an abstract TANGO interface for ccd's:



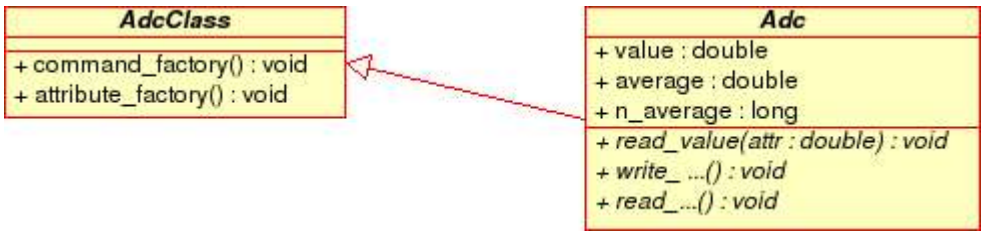
6.3 Motors

Example of an abstract TANGO interface for motor's:



6.4 Adc

Example of an abstract TANGO interface for adc's:



7 Abstract Control System

Abstraction is part of any good object oriented design. I think TANGO is well adapted to go to the next step of defining abstract device patterns for most common control system devices. These patterns could form the basis of an abstract control system for a larger community. Software which implements the TANGO control system interfaces will be easily shared in the wider community. Here are some examples of the more common devices which need to be defined as abstract devices :

PowerSupply
Motor
Ccd
SignalGenerator
Counter
Scan
Sequencer
Adc
Dac
SerialLine
Gpib
LinearAccelerator
CurrentTransformer
FluorescentScreen

These abstract classes and others, which are not mentioned here, could add up to form a super abstract device pattern called ControlSystem, which represents a whole control system. We might even succeed where many before us have failed. I have not given up on the dream of a common set of abstract interfaces for common control system devices.

8 Conclusion

This paper has shown that TANGO offers the technology to implement concrete device classes which respect abstract interfaces. Hopefully this will incite more people

to follow this route and discuss how we can achieve a common set of TANGO interfaces for concrete devices belonging to the same abstract family.

Your comments are welcomed.