

T  
A  
N  
G



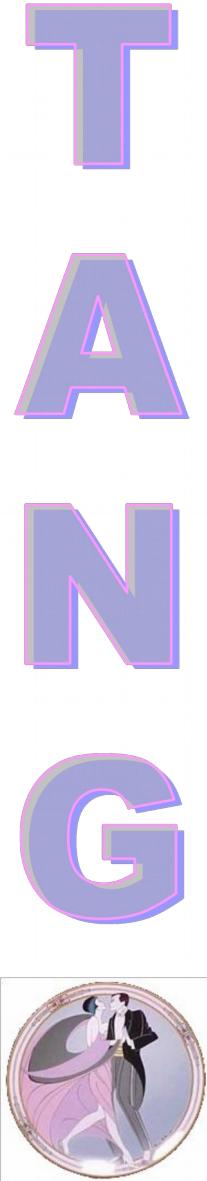
# Tango Training



# Tango Training

- Introduction (1)
- Device and device server (2)
- Writing device server and client (the basic) (3 – 5)
- Events (6)
- Device server level 2 (7)
- Advanced features (8)
- Python binding (9)
- ATK (10)
- Miscellaneous (11)





# Tango Training:

## Part 1 : Introduction

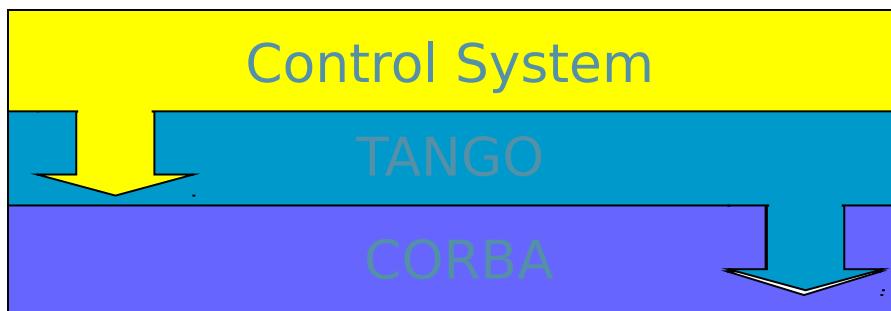
- What is Tango?
- Collaboration
- Languages/OS/compilers
- CORBA



T  
A  
N  
G  


# What is Tango?

- A CORBA framework for doing controls
  - A toolbox to implement a control system
  - A specialization of CORBA adapted to Control
  - Hide the complexity of Corba to the programmer
  - Adds specific contol system features



T  
A  
N  
G  
E



# What is Tango?

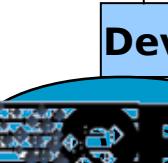
- A software bus for distributed objects

Java, C++, Python

Linux, Windows, Solaris



*TANGO Software Bus*



Linux, Windows, Solaris



Labview RT



# What is Tango?

- Provides a unified interface to all equipments, hiding how they are connected to a computer (serial line, USB, sockets....)
- Hide the network
- Location transparency
- Tango is one of the Control System available today but other exist (EPICS, ACS...)



# The Tango Collaboration

- Tango collaboration history
  - Started in 2000 at ESRF
  - In 2002, Soleil joins ESRF to develop Tango
  - End 2003, Elettra joins the club
  - End 2004, Alba also joins
  - 2006: Hasilab, GKSS will use Tango for Petra 3 beamlines
- An official paper signed by institute directors



# The Tango Collaboration

## ■ How it works:

- Two collaboration meetings per year
- A mailing list ([tango@esrf.fr](mailto:tango@esrf.fr))
- One Tango coordinator per site
- WEB site to download code, get documentation, search the mailing list history, read collaboration meeting minutes...  
<http://www.tango-controls.org>
- Collaborative development using SourceForge



# Language/OS/compilers

- Tango is now (June 2008) at release 6.1.
  - The training is based on the features of this release.
- Languages/Commercial tools

	C++	Java	Python	Matlab	LabView	IgorPro
Client	OK	OK	OK	OK	OK	OK
Server	OK	OK ***	OK			



# Language/OS/Compilers

## ■ Linux (x86 – PPC – ARM)

- Suse 8.2 / 9.3 / 10.2, Debian 3.0, Debian 3.0,  
Redhat E4.0 / E5.0, Ubuntu 7.04, 7.10 and 8.04
- gcc 3.3 / 3.4 / 4.1 / 4.2

## ■ Solaris

- Solaris 7 / 9 + CC 5.4 and CC 5.5
- Solaris 9 + gcc 3.3 / 3.4

## ■ Windows

- Windows 98 / 2000 / XP with VC8



# CORBA

- Common Object Request Broker Architecture
  - Promoted by OMG
  - It's just paper, not software
- CORBA defines the ORB: a way to call an object “method” wherever the object is
  - In the same process
  - In another process
  - In a process running somewhere on the network
- CORBA also defines services available for all objects (event, naming, notification)



# CORBA

- CORBA allows mixing languages: a client is not necessarily written in the same language as server
- CORBA uses an Interface Definition Language (IDL)
- CORBA defines bindings between IDL and computing languages (C++, Java, Python, Ada....)
- It uses IOR (Interoperable Object Reference) to locate an object

# CORBA

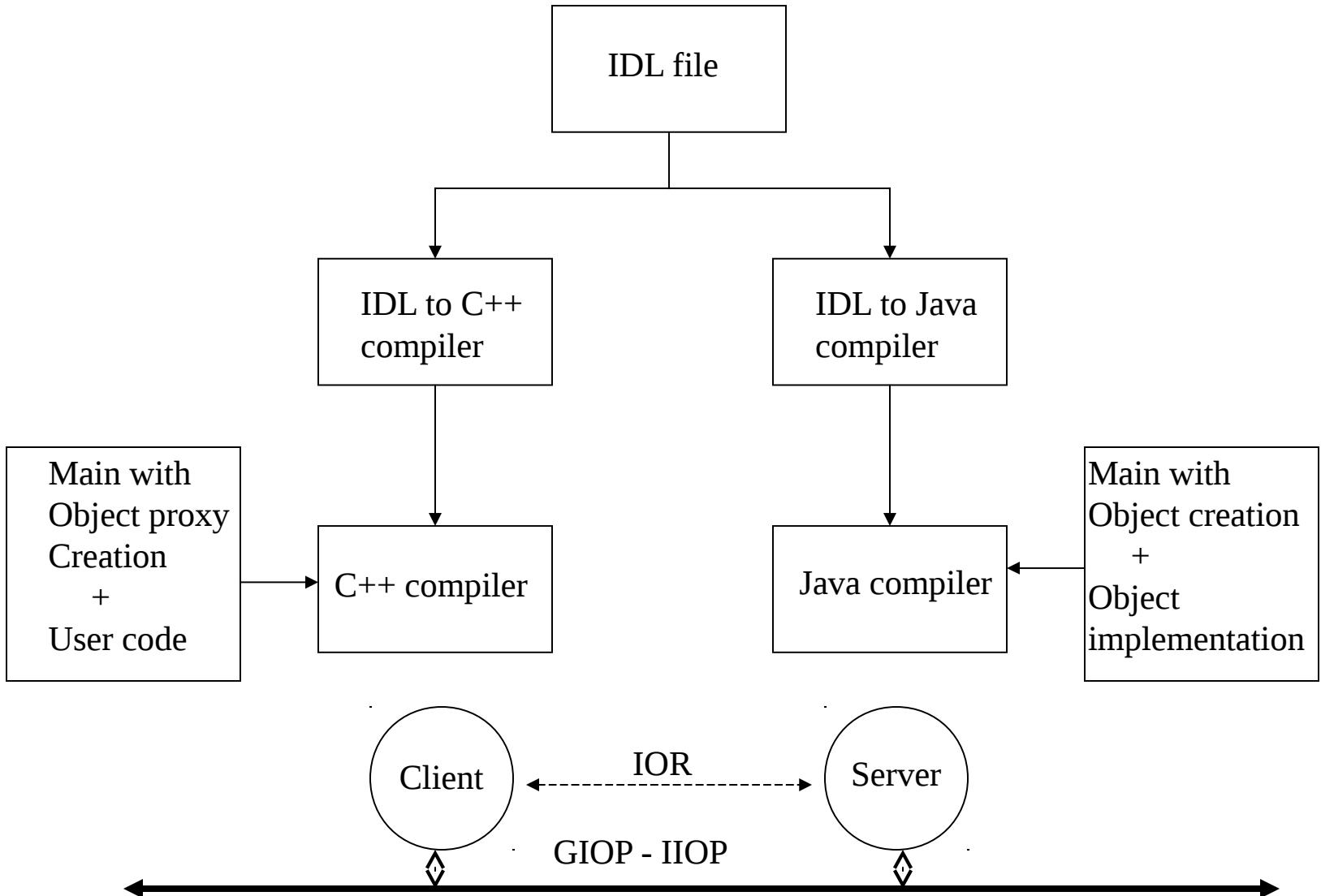
## ■ IDL for a remote controlled car

```
interface remote_car
{
    void go_forward(void);
    void go_backward(void);
    void stop(void);
    void turn(float angle);
};
```



# CORBA

T  
A  
N  
G  



# CORBA

- Many CORBA ORB and services available
- Tango uses
  - omniORB for C++ ORB  
(<http://omniorb.sourceforge.net>)
  - JacORB for Java ORB (<http://www.jacorb.org>)
  - omniNotify for CORBA notification service  
(<http://omninotify.sourceforge.net>)

T  
A  
N  
G

# Tango Training: Part 2 : Device and Device Server

- The Tango device
- The Tango device server
- A minimum Tango System



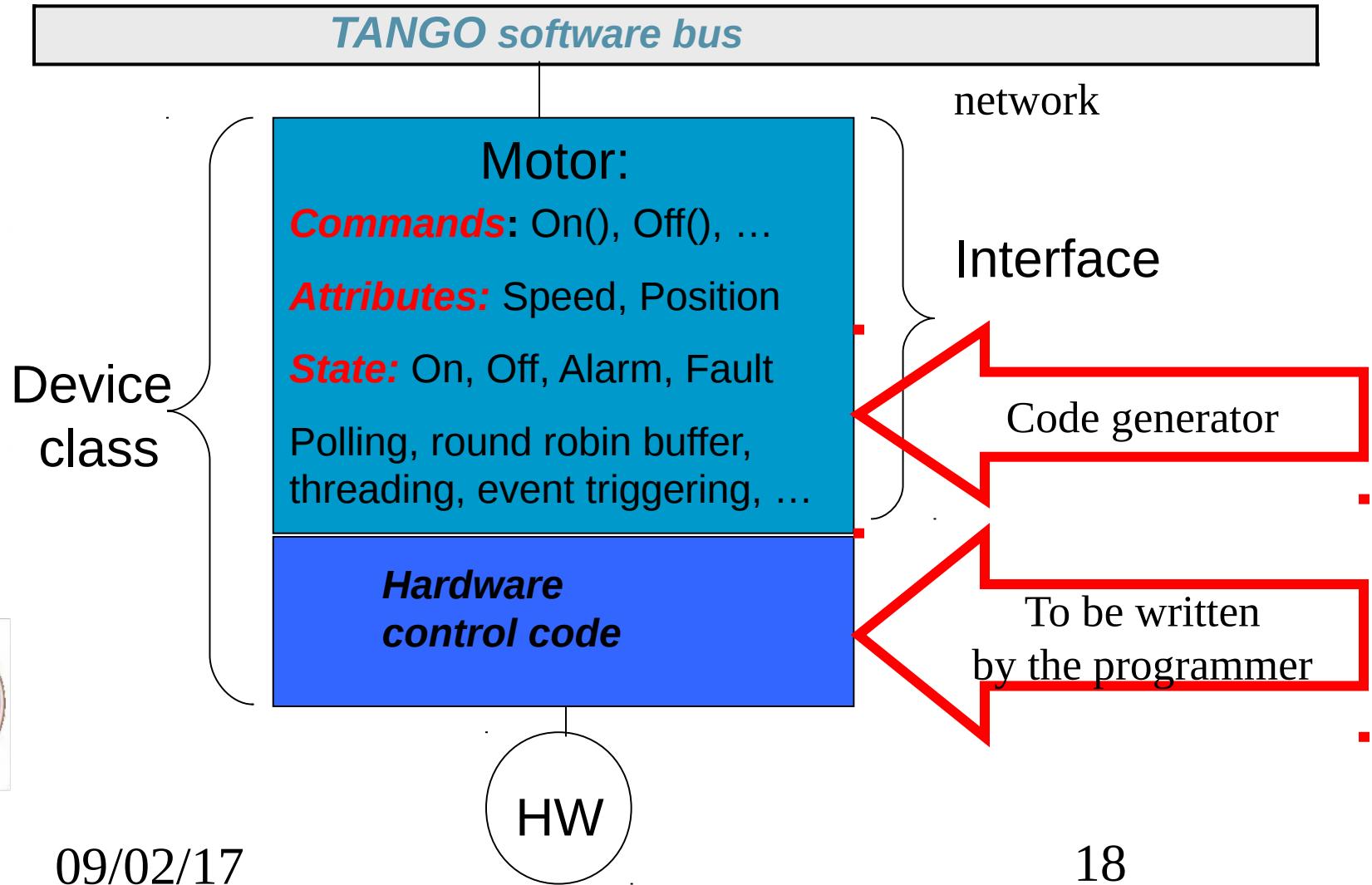


# The Tango Device

- The fundamental brick of Tango is the device!
- Everything which needs to be controlled is a “device” from a very simple equipment to a very sophisticated one
- Every device has a three field name “domain/family/member”
  - sr/v-ip/c18-1, sr/v-ip/c18-2
  - sr/d-ct/1
  - id10/motor/10

T  
A  
N  
G  


# The Tango Device



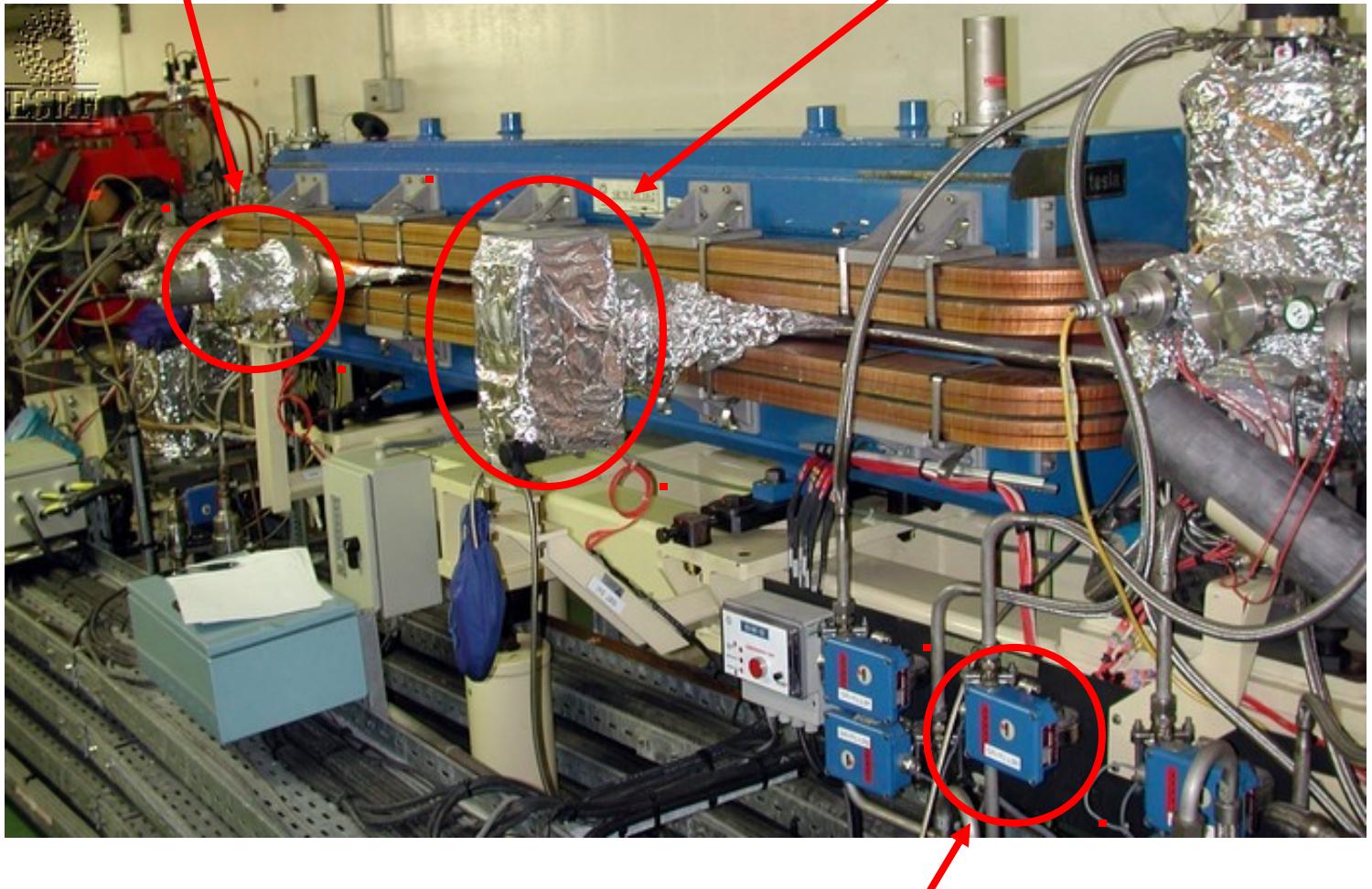
T  
A  
N  
G



# Some device(s)

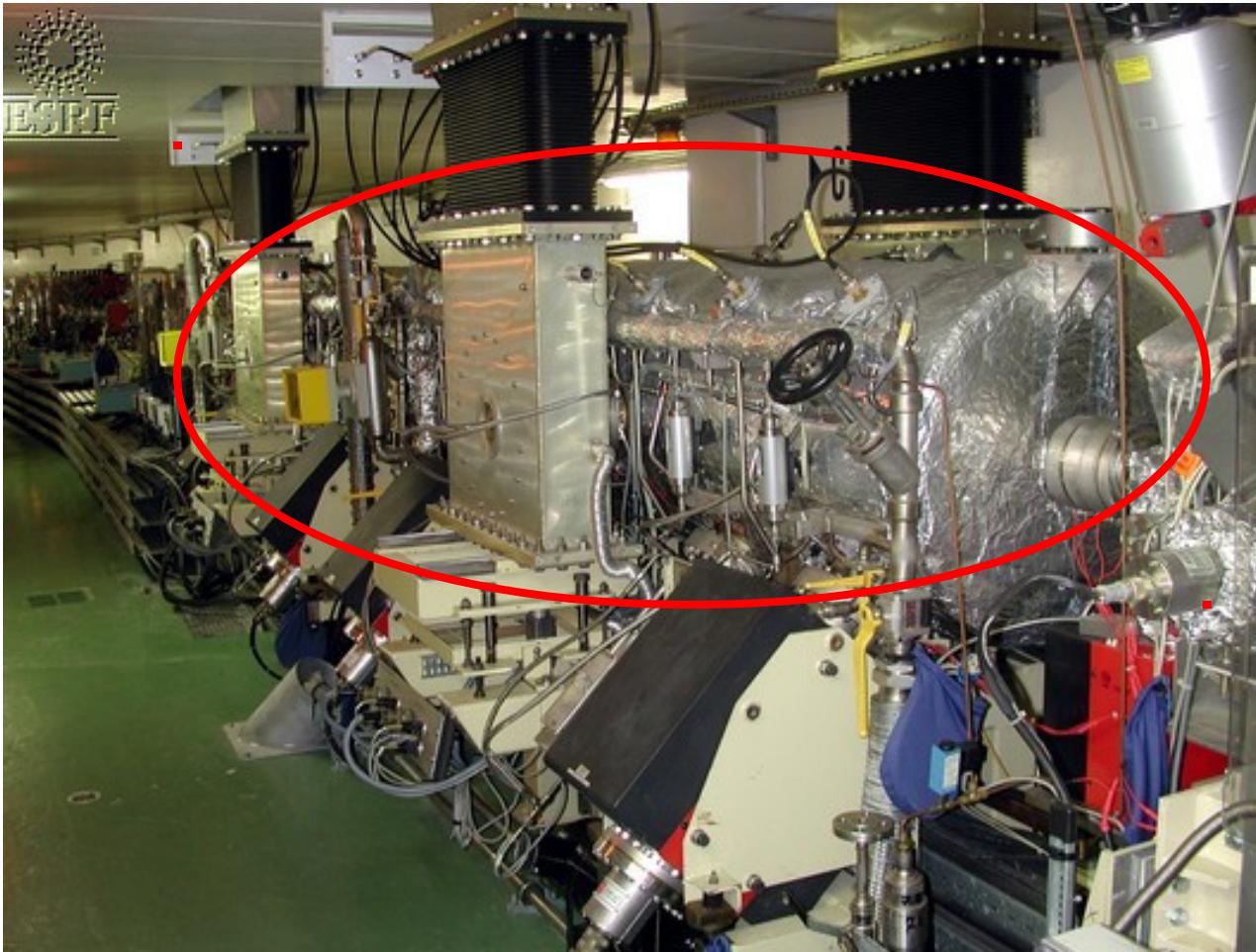
One device

One device



T  
A  
N  
G  


# A sophisticated device (RF cavity)



another  
device



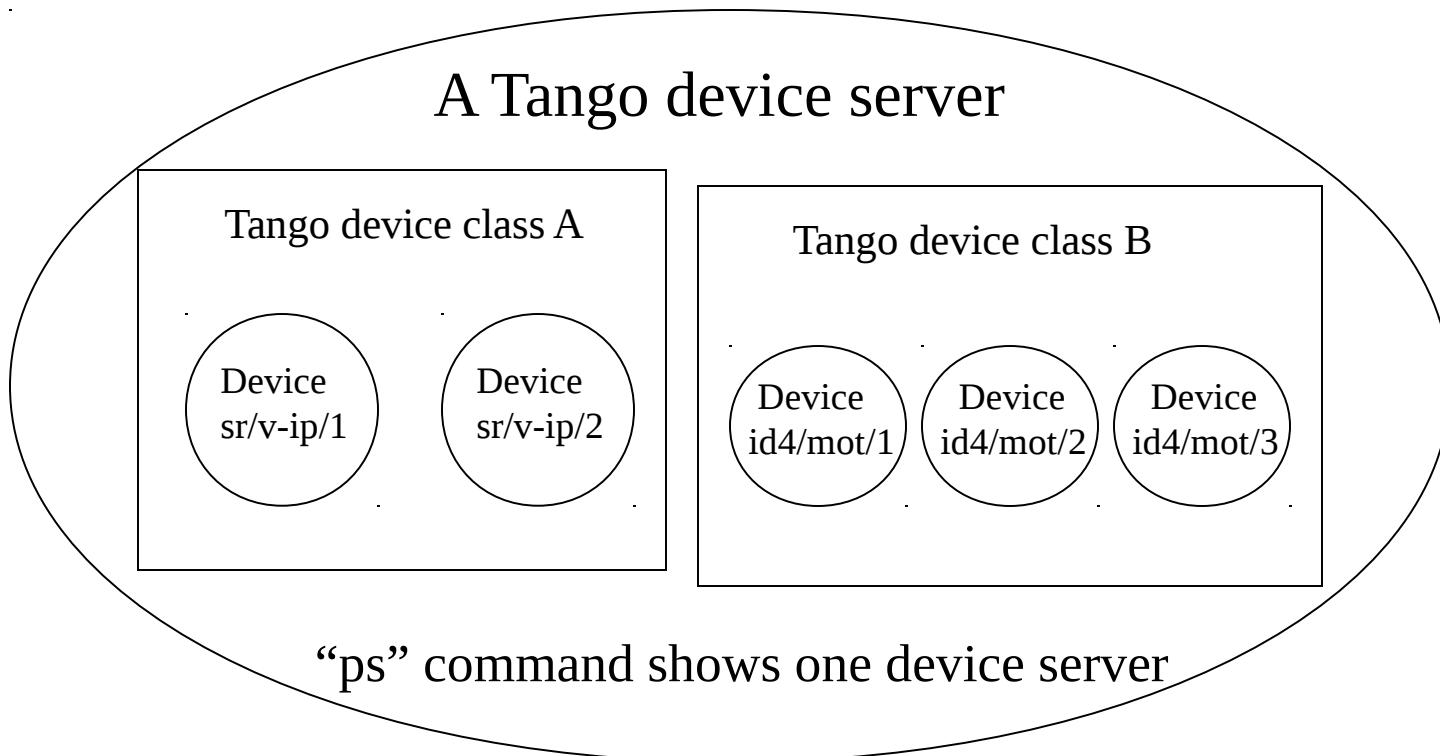
# The Tango Class

- Every device belongs to a Tango class (not a computing language class)
- Every device inherits from the same root class (DeviceImpl class)
- A Tango class implements the necessary features to control one kind of equipment
  - Example : The Agilent 4395a spectrum analyzer controlled by its GPIB interface

T  
A  
N  
G  


# The Tango Device Server

- A Tango device server is the process where the Tango class(es) are running.





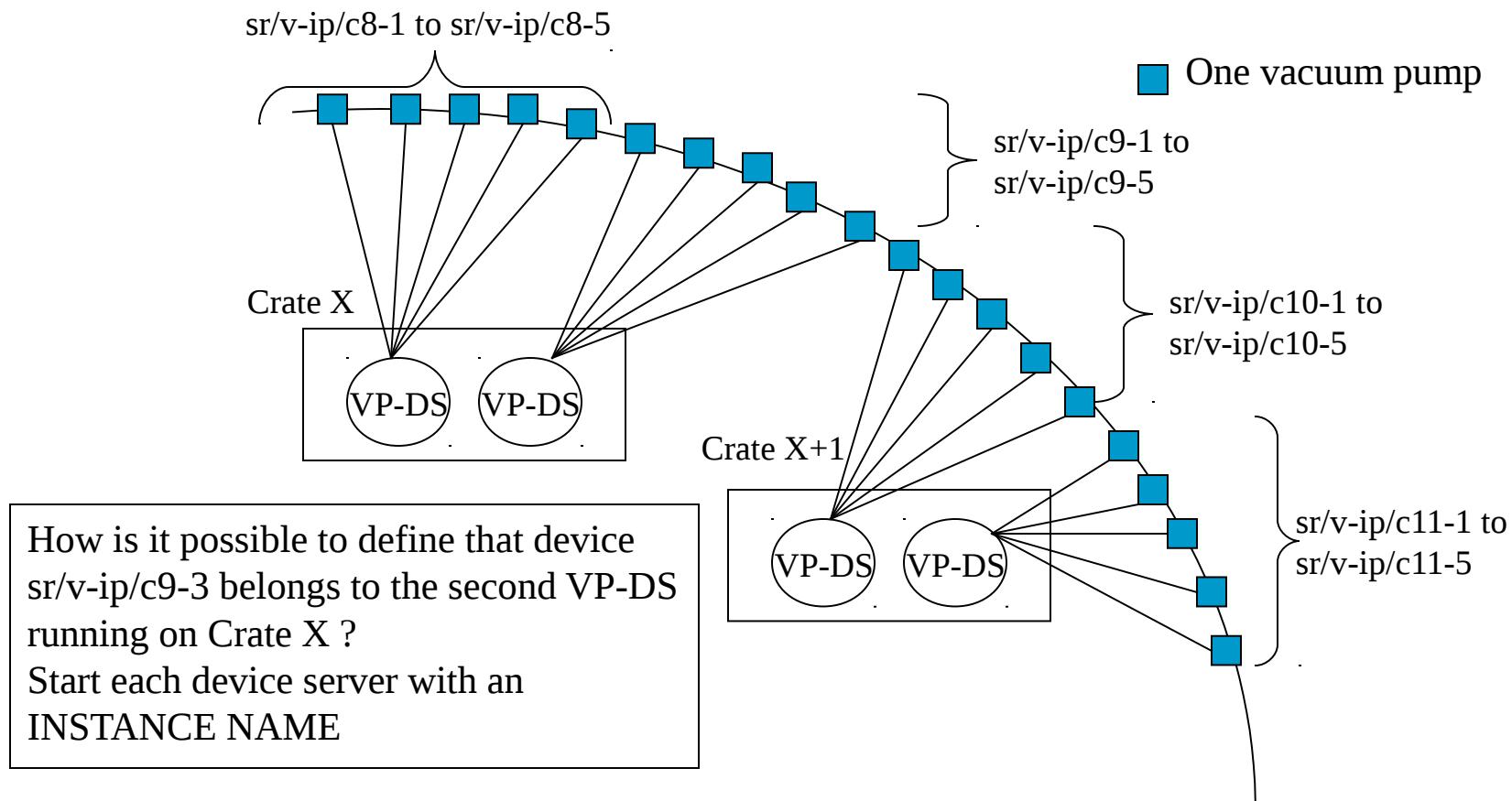
# The Tango Device Server

- Tango uses a database to configure a device server process
- Device number and names for a Tango class are defined within the database **not in the code.**
- Which Tango class(es) are part of a device server process is defined in the database but also in the code (training part 6)

T  
A  
N  
G  


# The Tango Device Server

- Each device server is defined by the couple “executable name / instance name”





# The Tango Device Server

- During its startup sequence, a Tango device server asks the database which devices it has to create and to manage (number and names)
- Device servers are started like
  - VP-DS c8
  - VP-DS c10

DS exec name	Inst name	Class name	Device name
VP-DS	c8	RubberPump	sr/v-ip/c8-1
VP-DS	c8	RubberPump	sr/v-ip/c8-2
VP-DS	c8	RubberPump	sr/v-ip/c8-3



# A minimum Tango System

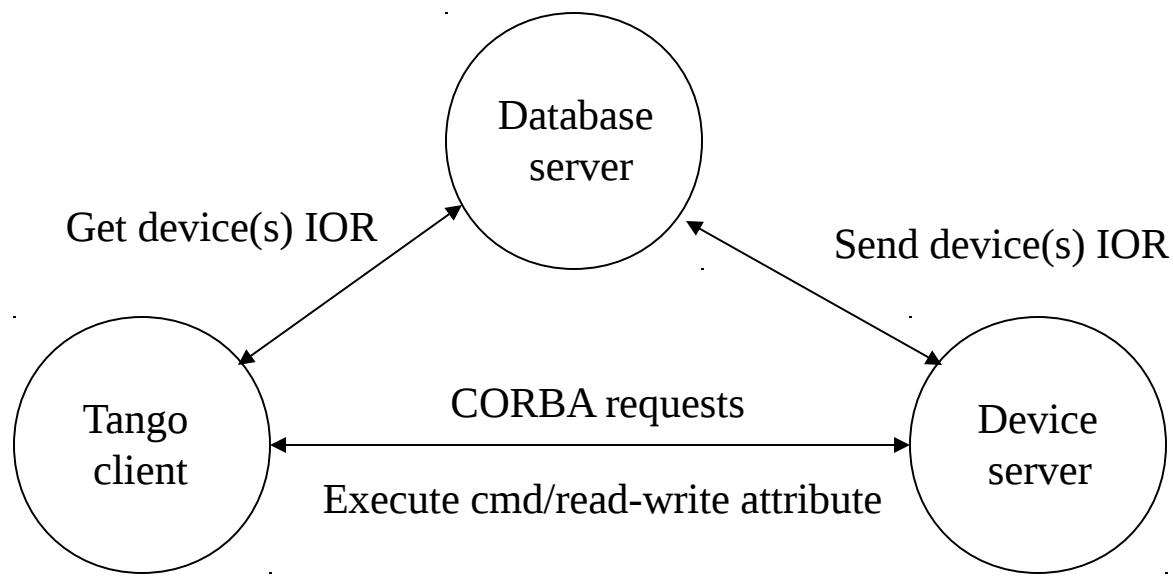
- To run a Tango control system, you need
  - A running MySQL database
  - The Tango database server
    - It is a C++ Tango device server with one device
- To start the database server on a fixed port
- The environment variable **TANGO\_HOST** is used by client/server to know
  - On which **host** the database server is running
  - On which **port** it is listening

T  
A  
N  
G  


# A minimum Tango System

DataBases 2 -ORBendPoint giop:tcp:host:10000

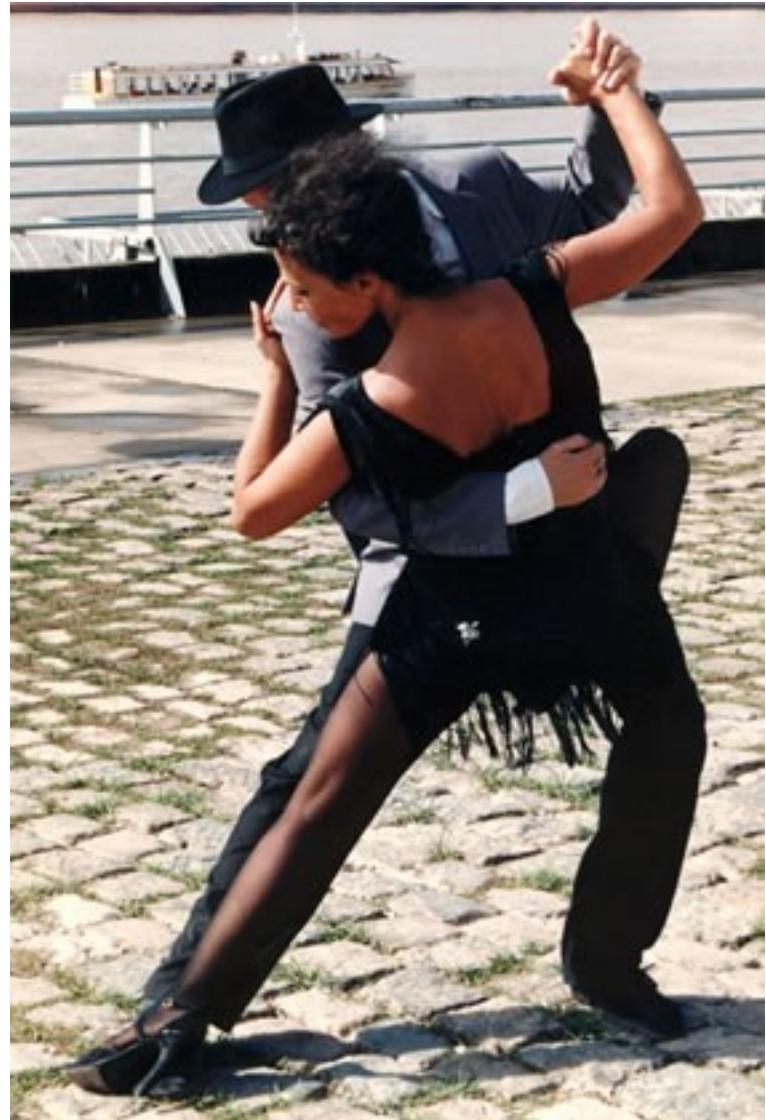
TANGO\_HOST=host:port (Ex : TANGO\_HOST=orion:10000)





# Tango Training: Part 3 : Writing a device server

- Tango device command/attributes
- Coding a Tango class
- Errors
- Properties



# Tango Device

- Each Tango device is a CORBA object
- Each Tango device supports the same network interface
- What do we have in this interface ?





# Command/Attribute

- On the network a Tango device mainly has
  - **Command(s)**: Used to implement “action” on a device (switching ON a power supply)
  - **Attribute(s)**: Used for physical values (a motor position)
- Clients ask Tango devices to execute a command or read/write one of its attributes
- A Tango device also has a **state** and a **status** which are available using command(s) or as attribute(s)



# Tango Device Command

- A command may have one input and one output argument.
- A limited set of argument data types are supported
  - Boolean, short, long, long64, float, double, string, unsigned short, unsigned long, unsigned long64, array of these, 2 exotic types and State data type

# Tango Device Attribute

- Self describing data via a configuration
- Twelve data types supported:
  - Boolean, unsigned char, short, unsigned short, long, long64, unsigned long, unsigned long64, float, double, string and State data type
- Three accessibility types
  - Read, write, read-write
- Three data formats
  - Scalar (one value), spectrum (an array of one dimension), image (an array of 2 dimensions)
- Tango adds 2 attributes which are state and status





# Tango Device Attribute

- When you read an attribute you receive:
  - The attribute data (luckily...)
  - An attribute quality factor
    - ATTR\_VALID, ATTR\_INVALID, ATTR\_CHANGING, ATTR\_ALARM, ATTR\_WARNING
  - The date when the attribute was read (number of seconds and usec since EPOCH)
  - Its name
  - Dimension
- When you write an attribute, you send
  - The new attribute data
  - The attribute name

# Device Attribute Configuration

- Attribute configuration defined by its properties
  - Five type of properties
    - Hard-coded
    - Modifiable properties
      - GUI parameters
      - Max parameters
      - Alarm parameters
      - Event parameters
- A separate network call allows clients to get attribute configuration (get\_attribute\_config)



# Device Attribute Configuration

- The hard coded attribute properties (8)
  - name
  - data\_type
  - data\_format
  - writable
  - max\_dim\_x
  - max\_dim\_y
  - writable\_attr\_name
  - display level

# Device Attribute Configuration

- The GUI attribute properties (6)
  - Description
  - Label
  - Unit
  - Standard\_unit
  - Display\_unit
  - Format (C++ or printf)
- The Maximum attribute properties (used only for writable (2))
  - min\_value
  - max\_value



# Tango Device State

- A limited set of 14 device states is available.
  - ON, OFF, CLOSE, OPEN, INSERT, EXTRACT, MOVING, STANDBY, FAULT, INIT, RUNNING, ALARM, DISABLE and UNKNOWN
- All defined within an enumeration.



# Writing a Tango Device Class

- Writing Tango device class need some glue code. We are using a code generator with a GUI called **POGO** : Program Obviously used to Generate Objects
- Following some simple rules, it's possible to use it during all the device class development cycle (not only for the first generation)
- POGO generates
  - C++, Python and Java Tango device class glue code
  - Makefile
  - Basic Tango device class documentation (HTML)

# A Tango Device Class (example)

- A ski lift class
  - 3 states
    - ON, OFF, FAULT
  - 3 commands
  - 3 attributes

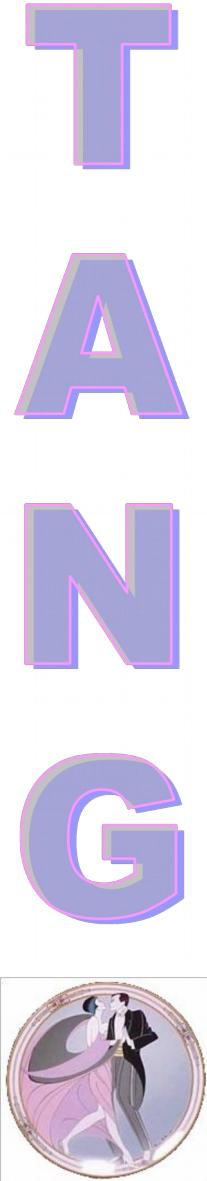
Name	In	Out	Allowed
Reset	Void	Void	If FAULT
On	Void	Void	If OFF
Off	Void	Void	Always

Name	type	format	Writable
Speed	double	scalar	Read/Write
Wind_speed	double	scalar	Read
Seats_pos	long	spectrum	Read



# Compiling/Linking a Tango DS

- Two include directories
  - \$(TANGO\_ROOT)/include
  - \$(OMNI\_ROOT)/include
- Two library directories
  - \$(TANGO\_ROOT)/lib
  - \$(OMNI\_ROOT)/lib
- Libraries needed (UNIX like – See doc for Windows)
  - 2 Tango libs: libtango.so, liblog4tango.so
  - 4 CORBA libs: libomniORB4.so, libCOS4.so, libomniDynamic4.so, libomnithread.so
  - OS libs:
    - libpthread.so for Linux
    - libposix4.so, libsocket.so, libnsl.so and libpthread.so for Solaris



# Exercise 1

- Generate the ski lift class with Pogo
  - 3 states :
    - ON, OFF, FAULT
    - OFF at startup
  - 3 commands (without arguments)
    - On to switch device ON
      - allowed only when switched off
    - Off to switch device OFF
      - allowed only when switched on
    - Reset to reset the device in case of a FAULT
      - allowed only when FAULT
  - 3 attributes :
    - Speed: read/write – scalar – double. Min=0, Max=5, Alarm>4
    - WindSpeed: read – scalar - double
    - SeatsPos: read – spectrum – long
- Generate the documentation



# Coding a Tango Device Class

- Four things to code
  - Device creation
  - Implementing commands
  - Reading attribute(s)
  - Writing attributes





# Coding a Tango Class

- For the SkiLift class, Pogo has created 5 files plus 2 files for the device server process and a Makefile
  - SkiLift.h and .cpp files
  - SkiLiftClass.h and .cpp files
  - SkiLiftStateMachine.cpp
  - class\_factory.cpp and main.cpp files for device server process
- Most of the times only SkiLift.h and SkiLift.cpp files have to be modified



# Coding a Tango Class

- Which methods can I use within a Tango class?
  - SkiLift class inherits from a Tango class called `Device_<x>Impl`
    - All the methods from `Device_<x>Impl` class
  - Some methods received a `Attribute` or `Wattribute` object
    - All the methods of these two classes
- Doc available at <http://www.tango-controls.org> then “Tango Kernel” and “Tango device server classes”

# T Creating the Device (constructor)

- ## A N G
- A `init_device()` method to construct the device
    - `void SkiLift::init_device()`
  - A `delete_device()` to destroy the device
    - `void SkiLift::delete_device()`
  - All memory allocated in `init_device()` must be deleted in `delete_device()`





# Creating the device (constructor)

- Hardware returned speeds in one array and seat position in another array
- Add one static array in the SkiLift.h file for seats. The other one is allocated on the heap

```
//           Here is the end of the automatic code generation part  
-----  
long double   seat_pos[120];  
                  *hardware_speed_array;  
  
protected :  
           //           Add your own data members here
```

# T A N G Creating the Device (constructor)

## ■ The init\_device() method

- Allocate memory
- Init state and status

```
//+-----  
//  
// method :           SkiLift::init_device()  
//  
// description :     will be called at device initialization.  
//  
//-----  
void SkiLift::init_device()  
{  
    INFO_STREAM << "SkiLift::SkiLift() create device " << device_name << endl;  
  
    // Initialise variables to default values  
    //-----  
  
    hardware_speed_array = new double[2];  
  
    system_off();  
  
    set_state(Tango::OFF);  
    set_status("The ski lift is OFF");  
}
```





# Creating the Device

- The delete\_device() method
  - Delete memory allocated in init\_device

```
//  
+-----  
//  
// method :          SkiLift::delete_device()  
//  
// description :    will be called at device destruction or at init  
command.  
//  
//-----  
void SkiLift::delete_device()  
{  
    // Delete device's allocated object  
    delete [] hardware_speed_array;  
}
```



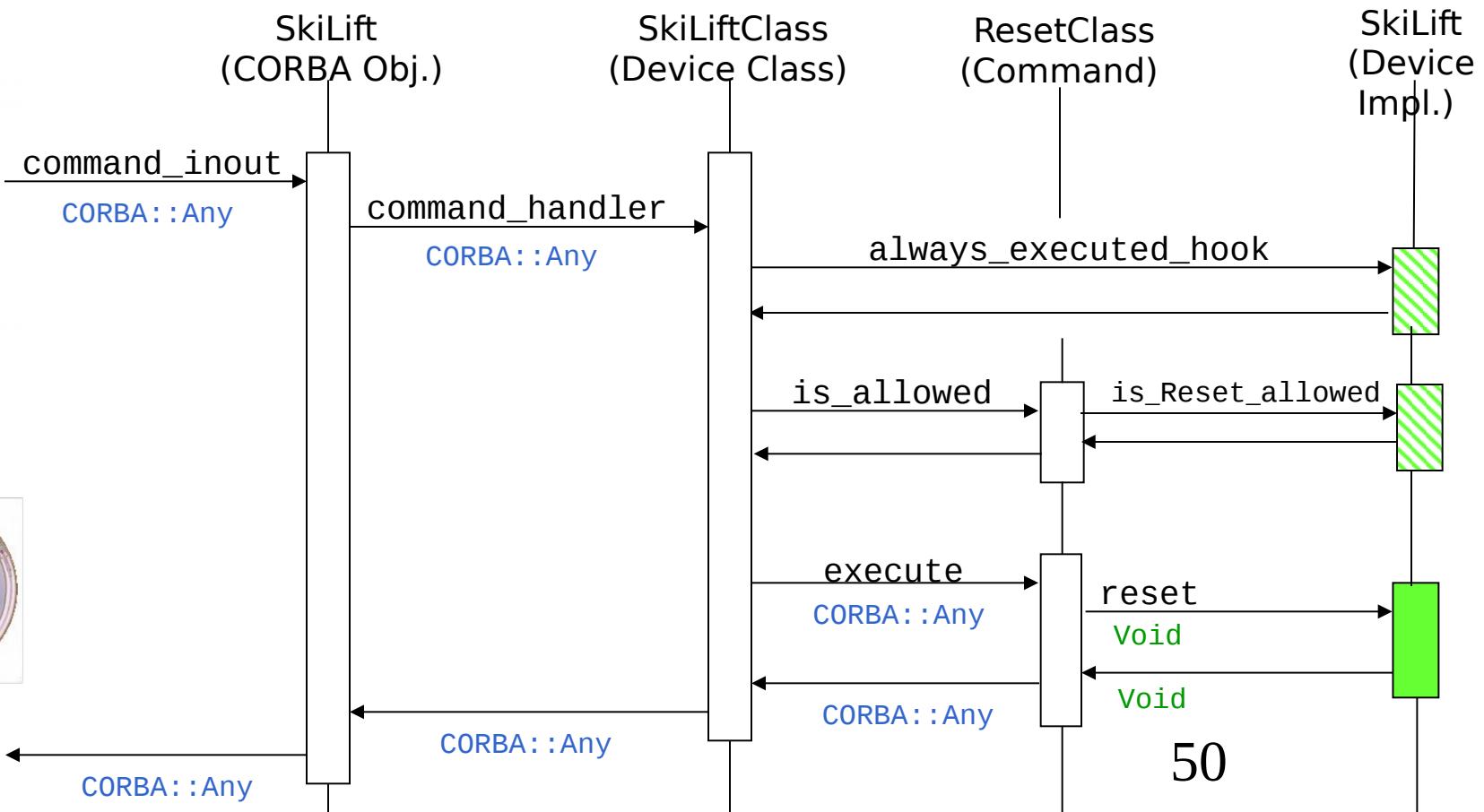
# Implementing a Command

- One method `always_executed_hook()` for all commands
  - `void SkiLift::always_executed_hook()`
- If state management is needed, one `is_xxx_allowed()` method in `SkiLiftStateMachine.cpp` file
  - `bool SkiLift::is_reset_allowed(const CORBA::Any &)`
- One method per command
  - `void SkiLift::reset()`

# Implementing a Command

## ■ Reset command sequencing

T  
A  
N  
G  



# Implementing a Command

## ■ SkiLift::is\_Reset\_allowed method coding

```
//+-----  
//  
// method :          SkiLift::is_Reset_allowed  
//  
// description :    Execution allowed for Reset command.  
//  
//-----  
bool SkiLift::is_Reset_allowed(const CORBA::Any &any)  
{  
    if (get_state() == Tango::ON || get_state() == Tango::OFF)  
    {  
        //      End of Generated Code  
  
        //      Re-Start of Generated Code  
        return false;  
    }  
    return true;  
}
```



# Implementing a Command

## ■ SkiLift::reset command coding

```
//+-----  
/**  
 *      method:    SkiLift::reset  
 *  
 *      description:      method to execute "Reset"  
 *  
 */  
//+-----  
void SkiLift::reset()  
{  
    DEBUG_STREAM << "SkiLift::reset(): entering... !" << endl;  
  
    // Add your own code to control device here  
    system_reset();  
    set_state(TANGO::OFF);  
    set_status("The ski lift is OFF");  
}
```

# Command Memory Management

- For string dynamically allocated (Pogo style)
  - Memory allocated in the command code and freed by the Tango layer

```
Tango::DevString MyDev::dev_string(Tango::DevString argin)
{
    Tango::DevString argout;

    cout << "The received string is " << argin << endl;

    string str("Am I a good Tango dancer?");
    argout = new char[str.size() + 1];
    strcpy(argout,str.c_str());

    return argout;
}
```



# Command Memory Management

- For string statically allocated
  - ConstDevString is not a new type, just to allow type overloading
  - Pogo gives you the choice (for free !)

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

    cout << "The received string is " << argin << endl;
    argout = "Hola todos";

    return argout;
}
```

# Command Memory Management

- For array dynamically allocated (Pogo)
  - Memory freed by Tango (how lucky are the user !)

```
Tango::DevVarLongArray *MyDev::dev_array()
{
    Tango::DevVarLongArray *argout = new Tango::DevVarLongArray();

    output_array_length = .....,;
    argout->length(output_array_length);
    for (unsigned int i = 0;i < output_array_length;i++)
        (*argout)[i] = i;

    return argout;
}
```

# T A N G **Command Memory Management**

- For array statically allocated
  - Tango provides a simple function to build Tango array types from a pointer (`create_xxxx`)

```
Tango::DevVarLongArray *MyDev::dev_array()
{
    Tango::DevVarLongArray *argout;

    long argout_array_length = ....;
    argout = create_DevVarLongArray(buffer, argout_array_length);
    return argout;
}
```

# T A N G **Command Memory Management**

- For string array dynamically allocated
  - Again memory will be freed by Tango layer

```
Tango::DevVarStringArray *MyDev::dev_str_array()
{
    Tango::DevVarStringArray *argout = new Tango::DevVarStringArray();

    argout->length(3);
    (*argout)[0] = CORBA::string_dup("Rumba");
    (*argout)[1] = CORBA::string_dup("Waltz");
    string str("Jerck");
    (*argout)[2] = Tango::string_dup(str.c_str());

    return argout;
}
```





# Exercise 2 (Arg !!...)

- Use Pogo to add commands to the ski lift class
- Code the commands in the class as:
  - Cmd EchoShort : return (in \* 2)
  - Cmd EchoLongArray :
    - Out size = in size \* 2
    - Out = [in, in \* 2] (Ex : In = [1,3] Out = [1,3,2,6])
    - Dynamically allocated
  - Cmd StringReverse : dynamically allocated



# Back to the init\_device method

```
//+-----  
//  
// method :           SkiLift::init_device()  
//  
// description :      will be called at device initialization.  
//  
//-----  
void SkiLift::init_device()  
{  
    INFO_STREAM << "SkiLift::SkiLift() create device " << device_name << endl;  
  
    // Initialise variables to default values  
    //-----  
  
    hardware_speed_array = new double[2];  
  
    attr_Speed_read = &(hardware_speed_array[0]);  
    attr_WindSpeed_read = &(hardware_speed_array[1]);  
    attr_SeatsPos_read = seat_pos;  
  
    system_off();  
  
    set_state(Tango::OFF);  
    set_status("The ski lift is OFF");  
}
```



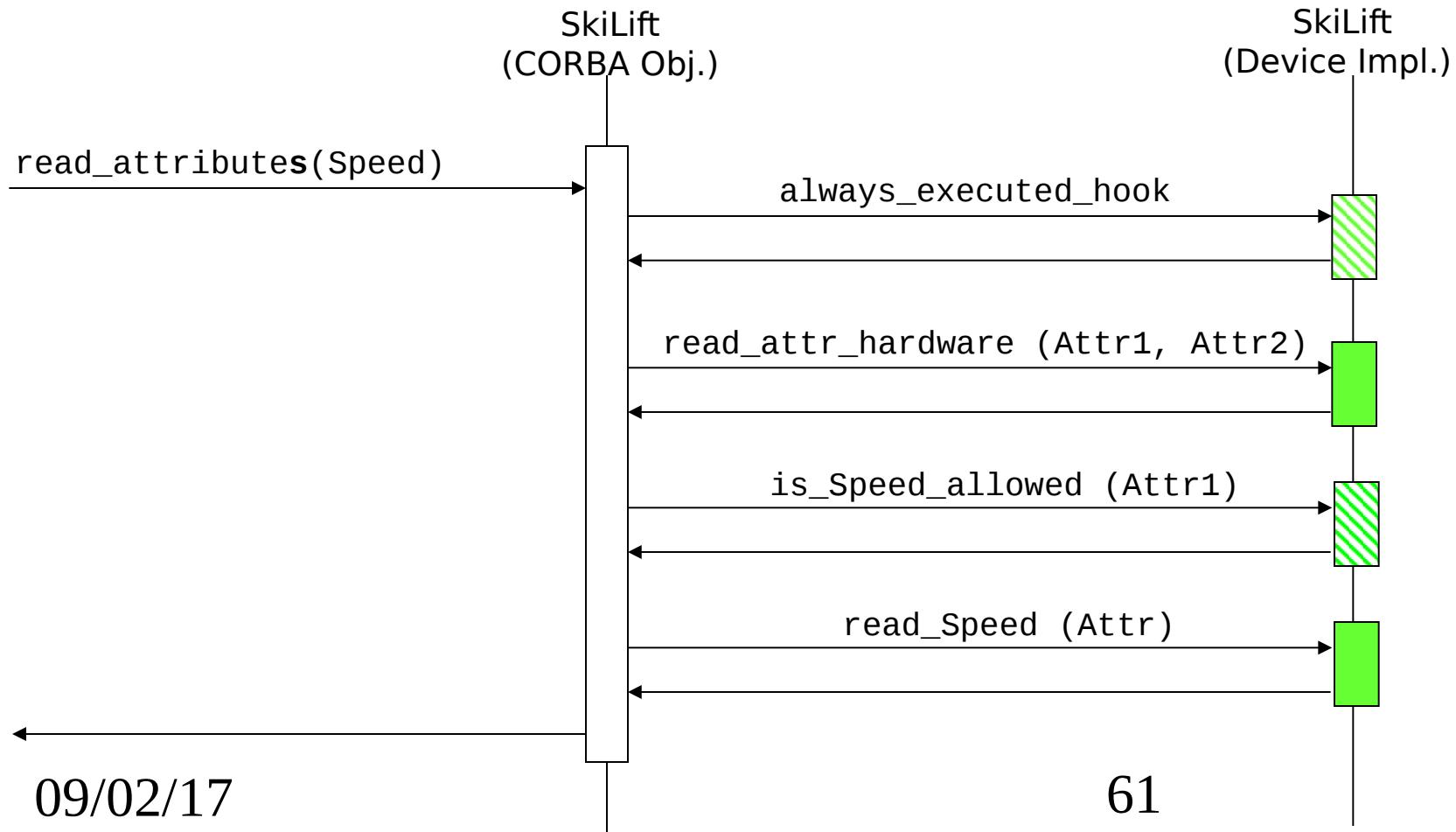
# Reading Attribute(s)

- One method to read hardware
  - `void SkiLift::read_attr_hwre(vector<long> &)`
- If state management is needed, one `is_xxx_allowed()` method (in `SkiLiftStateMachine.cpp` file)
  - `bool SkiLift::is_Speed_allowed(Tango::AttReqType &)`
- One method per attribute
  - `void SkiLift::read_Speed(Tango::Attribute &)`

T  
A  
N  
G  


# Reading Attribute(s)

## ■ Reading attribute(s) sequence





# Reading Attribute(s)

- Most of the attribute Tango feature are implemented in a Tango kernel class called “Attribute”. The user only manage attribute data
- Reading sequence
  - `read_attr_hw`
    - 1 call even if several attributes must be read
    - Rule: Reading the hardware only once
    - Update internal variable
  - `is_<attribute>_allowed`
    - 1 call per attribute
    - Rule: Enable/disable attribute reading



# Reading Attribute(s)

## ■ Reading sequence

### – `read_<attribute>`

- 1 call per attribute to read
- Rule: Affect a value to the attribute
- Associate the attribute and a variable which represents it with :
  - `attr.set_value(pointer_to_data,...)`
- Pogo defines the pointer as a SkiLift class data member but it is not initialised!



# Reading Attribute(s)

## ■ `read_attr_hw` method

```
//+-----  
//  
// method :          SkiLift::read_attr_hw  
//  
// description :    Hardware acquisition for attributes.  
//  
//-----  
void SkiLift::read_attr_hw(vector<long> &attr_list)  
{  
    DEBUG_STREAM << "SkiLift::read_attr_hw() entering... " << endl;  
//      Add your own code here  
  
    system_read_hw(speed_array, seat_pos);  
}
```



# Reading Attribute(s)

## ■ read\_Speed() method

```
//+-----  
//  
// method :      SkiLift::read_Speed  
//  
// description : Extract real attribute values for Speed acquisition result.  
//  
//-----  
void SkiLift::read_Speed(Tango::Attribute &attr)  
{  
    DEBUG_STREAM << "SkiLift::read_Speed(Tango::Attribute &attr) entering... " << endl;  
  
    attr.set_value(attr_Speed_read);  
}
```



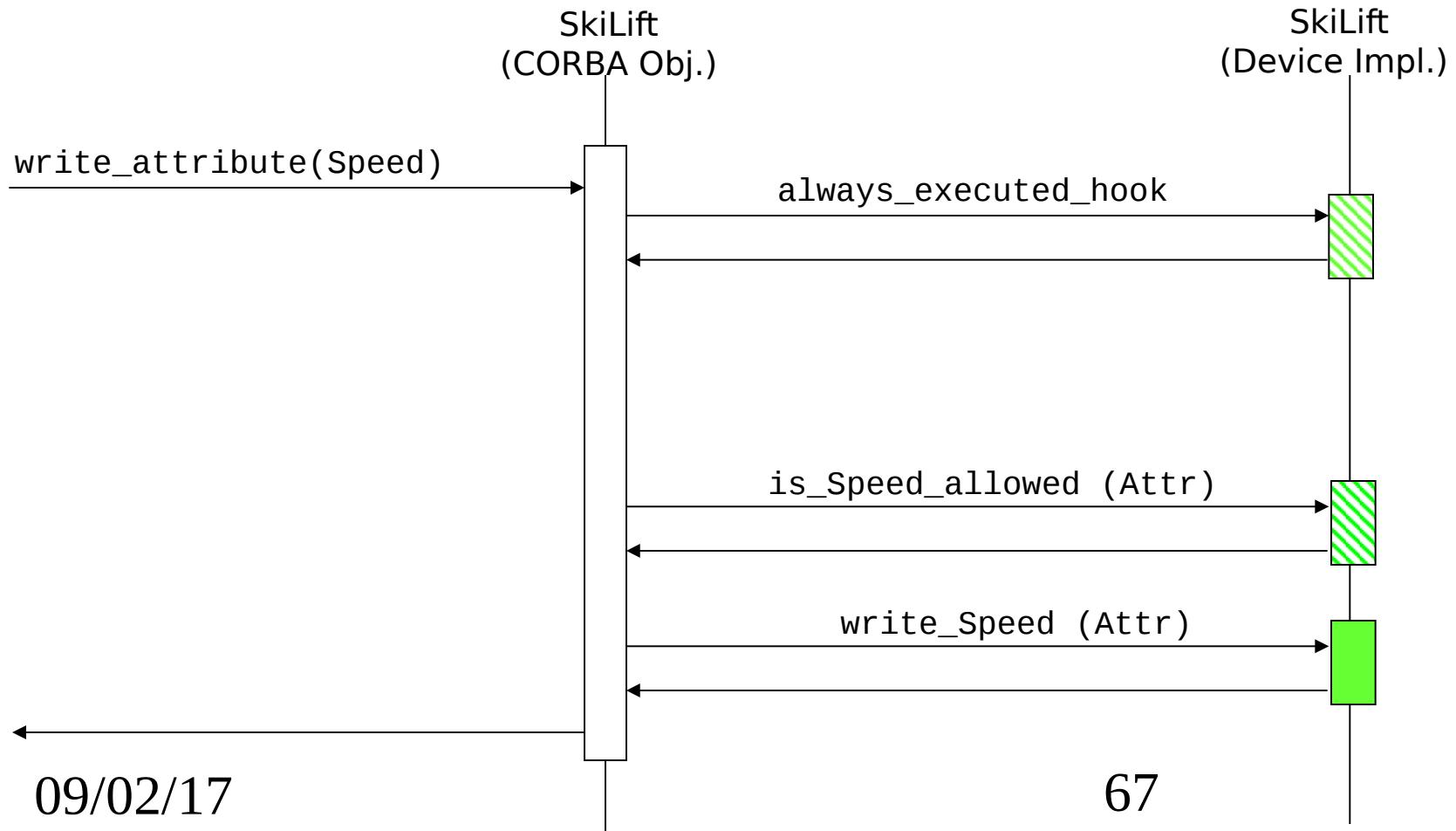
# Writing Attribute(s)

- If state management is needed, one `is_xxx_allowed()` method (in `SkiLiftStateMachine.cpp` file)
  - `bool SkiLift::is_Speed_allowed(Tango::AttReqType &`
- One method per attribute
  - `void SkiLift::write_Speed(Tango::Wattribute &)`

T  
A  
N  
G  


# Writing Attribute(s)

## ■ Writing attribute(s) sequence



# Writing Attribute(s)

## ■ Writing sequence

- `is_<attribute>_allowed`
  - 1 call per attribute
  - Rule: Enable/disable attribute writing
- `write_<attribute>`
  - 1 call per attribute to write
  - Rule: Get the value to be written and set the hardware
  - Get the value to be written with :
    - `attr.get_write_value(reference_to_data)`





# Writing Attribute(s)

## ■ write\_Speed() method

```
//+-----  
//  
// method :           SkiLift::write_Speed  
//  
// description :    Write Speed attribute values to hardware.  
//  
//-----  
void SkiLift::write_Speed(Tango::WAttribute &attr)  
{  
    DEBUG_STREAM << "SkiLift::write_Speed() entering... " << endl;  
  
    attr.get_write_value(attr_speed_write);  
    system_set_speed(attr_speed_write);  
}
```

# Attribute Memory Management

- Designed to reduce data copy
  - Uses a pointer to a memory area which by default is not freed

```
void MyDev::read_LongSpecAttr(Tango::Attribute &attr)
{
    ....
    attr.set_value(buffer);
}
```

But it is possible to ask Tango to free the allocated memory

```
void MyDev::read_LongSpecAttr(Tango::Attribute &attr)
{
    long length = ....
    long *buffer = new long[length];

    attr.set_value(buffer,length,0,true);
}
```

# Attribute Memory Management

## ■ What about a string spectrum attribute ?

```
Class MyDev:.....  
{  
....  
DevString attr_str_array[2];  
};
```

```
void MyDev::read_StringSpecNoRelease(Tango::Attribute &attr)  
{  
    attr_str_array[0] = “Donde esta”;  
    attr_str_array[1] = “la cerveza?”;  
  
    attr.set_value(attr_str_array,2);  
}  
  
void MyDev::read_StringSpecRelease(Tango::Attribute &attr)  
{  
    Tango::DevString *str_array = new Tango::DevString [2];  
  
    str_array[0] = Tango::string_dup(“La cerveza”);  
    str_array[1] = Tango::string_dup(“esta en la nevera”);  
  
    attr.set_value(str_array,2,0,true);  
}
```



# Pogo and Attributes

- For attributes Pogo generates (in your class):
  - For readable attributes :
    - A pointer called attr\_<att\_name>\_read
    - Don't forget to initialise it (in init\_device) or die
  - For writable attribute
    - A data called attr\_<att\_name>\_write
    - Not well adapted to Spectrum and Image attribute. A pointer is more adapted in these cases !!



# Memorised Attributes

- Only for writable scalar attributes!
- For every modification the attribute set point is saved to the database as attribute property `_value`
- Memorized attributes initialization options (supported by Pogo)

`Attr::set_memorized()` : marks attribute as memorised

`Attr::set_memorized_init (bool write_on_init)`

`write_on_init = True`: calls the attribute write method during the server startup

`write_on_init = False`: only initializes the attribute set point to the memorized value



# Exercise 3 (Arg !!...)

- Use Pogo to add attributes to the ski lift class
- Code the attributes in the class as:
  - Attribute ScaAttr : return write value \* 2, the data is only valid when the state is ON, the set value is memorised
  - Attribute SpecAttr : Return what has been written + 1, 0 at init
  - Attribute ImaAttr : Return an array 5\*3 (x\*y) with  $\text{array}[i] = x + y$



# Reporting Errors

## ■ Using exception (C++ or Java)

- Tango::DevFailed class which is an array of the Tango::DevError data type
- Tango::DevError data type has 4 elements :
  - reason (string)
    - The exception summary
  - desc (string)
    - The full error description
  - origin (string)
    - The method throwing the exception
  - severity (enum)
    - Type of error (Not really used)



# Reporting Errors

- Static method to help throwing an exception
- Another method to re-throw an exception and to add one element in the error stack (Often used in a “catch” block)

```
Tango::Except::throw_exception((const char *)"SkiLift::NoCable",
                               (const char *)"Oups, the cable has fall down !!",
                               (const char *)"SkiLift::init_device());
```

```
Tango::Except::re_throw_exception(Tango::DevFailed &ex,
                                   string &reason,
                                   string &desc,
                                   string &origin);
```

# Properties

- Properties are stored within a MySQL database
- No file – Use Jive to create/update/delete properties
- You can define properties at
  - Object level
  - Class level
  - Device level
  - Attribute level



# Properties

- Property data type
  - Simple type
    - boolean, short, long, float, double, unsigned short, unsigned long, string
  - Array type
    - short, long, float, double, string
- Pogo generates code to retrieve properties from the database and store them in your device
  - Method MyDev::get\_device\_property()

# Properties

## ■ Algorithm generated by Pogo to simulate default property values

- /IF/ class property has a default value
  - property = class property default value
- /ENDIF/
- /IF/ class property is defined in db
  - property = class property as found in db
- /ENDIF/
- /IF/ device property has a default value
  - property = device property default value
- /ENDIF/
- /IF/ device property is defined in db
  - property = device property as found in db
- /ENDIF/



# Attribute Properties

- Several ways to define them with a priority schema (from lowest to highest priority) :
  - There is a default value hard-coded within the library
  - You can define them at class level
  - You can define them by code (POGO) at class level
  - If you update them, the new value is taken into account by the device server and written into the database. Device level.



## Exercise 4 (Ai no ...)

- Create a new command (called “exception” in  
= void, out = long) to your class which  
according to a Boolean property
  - Returns 123
  - Throws an exception
- Create three devices for this class, two of  
them return values, one throws an exception  
(with property management)
  - By default (prop. not defined), do not throw  
exception

# T A N G **Some code executed only once ?**

- Yes, it is foreseen
- Each Tango class has a MyDevClass class (SkiLiftClass) with only one instance.
- Put code to be executed only once in its constructor
- Put data common to all devices in its data members
- The instance of MyDevClass is constructed before any devices



# A Tango Device Server Process

- The ClassFactory and main.cpp files allow to build device server executable from Tango class(es)
- The ClassFactory.cpp file

```
#include <tango.h>
#include <SkiLiftClass.h>

/**
 *          Create SkiLiftClass singleton and store it in DServer object.
 */

void Tango::DServer::class_factory()
{
    add_class(SkiLift_ns::SkiLiftClass::init("SkiLift"));
}
```

# A Tango Device Server Process

## The main.cpp file

```
int main(int argc,char *argv[])
{
    Tango::Util *tg;
    try
    {
        tg = Tango::Util::init(argc,argv);

        tg->server_init(false);

        cout << "Ready to accept request" << endl;
        tg->server_run();
    }
    catch (bad_alloc)
    {
        cout << "Can't allocate memory to store device object !!" << endl;
        cout << "Exiting" << endl;
    }
    catch (CORBA::Exception &e)
    {
        Tango::Except::print_exception(e);

        cout << "Received a CORBA_Exception" << endl;
        cout << "Exiting" << endl;
    }
    tg->server_cleanup();
    return(0);
}
```

# Automatically added Commands/Attributes

- Three commands are automatically added
  - **State** : In = void Out = DevState
    - Return the device state and check for alarms
    - Overwritable
  - **Status** : In = void Out = DevString
    - Return the device status
    - Overwritable
  - **Init** : In = void Out = void
    - Re-initialise the device (delete\_device + init\_device)
- Two attributes are automatically added
  - State and Status



# The remaining Network Calls

- **ping**
  - Just ping a device. Is it available on the network?
- **command\_list\_query**
  - Returns the list of device supported commands with their descriptions
- **command\_query**
  - Return the command description for one specific command
- **info**
  - Return general info on a device (class, server host....)



# The remaining Network Calls

- `get_attribute_config`
  - Return the attribute configuration for x (or all) attributes
- `set_attribute_config`
  - Set attribute configuration for x attributes
- `blackbox`
  - Return x entries of the device black box
  - Each device has a black box (round robin buffer) where each network call is registered with its date and the calling host

T  
A  
N  
G  


# The remaining Network Calls

- For completeness
  - Five CORBA attributes
    - state
    - status
    - name
    - description
    - adm\_name





# Tango Training:

## Part 4 :

### The Client Side

- The C++ client API
- Error management
- Asynchronous call
- Group call





# Tango on the Client Side

- A C++, Python and Java API is provided to simplify developer's life
  - Easy connection building between clients and devices
  - Manage re-connection
  - Hide some IDL call details
  - Hide some memory management issues
- These API's are a set of classes



# Tango C++ Client

- On the client side, each Tango device is an instance of a **DeviceProxy** class
- DeviceProxy class
  - Hide connection details
  - Hide which IDL release is supported by the device
  - Manage re-connection
- The DeviceProxy instance is created from the device name

```
Tango::DeviceProxy dev("id13/v-pen/12");
```

# Tango C++ Client

- The DeviceProxy *command\_inout()* method sends a command to a device
- The class DeviceData is used for the data sent/received to/from the command.

```
DeviceData DeviceProxy::command_inout(const char *, DeviceData &);
```

```
Tango::DeviceProxy dev("sr/v-pen/c1");
Tango::DeviceData d_in,d_out;

vector<long> v_in,v_out;

d_in << v_in;
d_out = dev.command_inout("MyCommand",d_in);
d_out >> v_out;
```





# Tango C++ Client

- The DeviceProxy *read\_attribute()* method reads a device attribute (or *read\_attributes()*)
- The class DeviceAttribute is used for the data received from the attribute.

```
DeviceAttribute DeviceProxy::read_attribute(string &);
```

```
Tango::DeviceProxy *p_dev = new DeviceProxy("id12/pen/2");
Tango::DeviceAttribute da;
float press;
string att_name("Pressure");

da = p_dev->read_attribute(att_name);
da >> press;
```

# Tango C++ Client

- The DeviceProxy *write\_attribute()* method writes a device attribute (or *write\_attributes()*)
- The class DeviceAttribute is used for the data sent to the attribute.

```
void DeviceProxy::write_attribute(DeviceAttribute &);
```

```
Tango::DeviceProxy dev("id234/motor/17897);
long spe = 102;
Tango::DeviceAttribute da("Speed",spe);

dev.write_attribute(da);
```



T  
A  
N  
G

# Tango C++ Client

- The API manages re-connection
  - By default, no exception is thrown to the caller when the automatic re-connection takes place
  - Use the *DeviceProxy::set\_transparency\_reconnection()* method if you want to receive an exception
- Don't forget to catch the Tango::DevFailed exception!



# Tango C++ Client

- Many methods available in the **DeviceProxy** class
  - ping, info, state, status, set\_timeout\_millis, get\_timeout\_millis, attribute\_query, get\_attribute\_config, set\_attribute\_config.....
- If you are interested only in attributes, use the **AttributeProxy** class
- Look at chapter 6 of Tango book or die...  
(chapter 5 for Java fans)





# Errors on the Client Side

- All the exception thrown by the API are Tango::DevFailed exception
- One catch block is enough
- Ten exception classes (inheriting from DevFailed) have been created
  - Allow easier error treatment
- These classes do not add any new information compared to the DevFailed exception



# Errors on the Client Side

- Exception classes :
  - ConnectionFailed, CommunicationFailed, WrongNameSyntax, NonDbDevice, WrongData, NonSupportedFeature, AsynCall, AsynReplyNotArrived, EventSystemFailed, NamedDevFailedList
- Documentation tells you (or should) which kind of exception could be thrown.



# Errors on the Client Side

## ■ A small example

```
try {  
    Tango::AttributeProxy ap("id18/pen/2Press");  
    Tango::DeviceAttribute da;  
  
    da = ap.read();  
    float pre;  
    da >> pre;  
}  
catch (Tango::WrongNameSyntax &e) {  
    cout << "Et couillon, faut 3 / !" << endl;  
}  
catch (Tango::DevFailed &e) {  
    Tango::Except::print_exception(e);  
}
```



## Exercise 5 (A very easy one)

- Write a simple C++ client for the SkiLift device.
- The client should set the skilift speed, switch the skilift on and read the actual wind speed.
- A second version of the client should read the skilift state, the wind speed and ScaAttr via the read\_attributes() method.



# Asynchronous Call

- Asynchronous call :
  - The client sends a request to a device and does not block waiting for the answer.
  - The device informs the client process that the request has ended
- Does not request any changes on the server side
- Supported for
  - command\_inout
  - read\_attribute(s)
  - write\_attribute(s)



# Asynchronous call

- Tango supports two models for clients to get requested answers
  - The **polling** model
    - The client decides when it checks for requested answers
      - With a non blocking call
      - With a blocking call
  - The **callback** model
    - The request reply triggers a callback method
      - When the client requested it with a synchronization method (Pull model)
      - As soon as the reply arrives in a dedicated thread (Push model)
- DeviceProxy is not thread safe (callback push model)



# Asynchronous Call

- For polling mode, use
  - *DeviceProxy::command\_inout\_asynch()* method to send commands
  - *DeviceProxy::command\_inout\_reply()* method to get command replies (blocking or not blocking)

```
Tango::DeviceProxy dev(...);  
long asyn_id;  
  
asyn_id = dev.command_inout_asynch("MyCmd");  
  
Bla bla bla  
  
Tango::DeviceData dd;  
dd = command_inout_reply(asyn_id);
```



# Asynchronous Call

- For callback, write a class inheriting from `Tango::CallBack` and overwrite
  - `cmd_ended()` method for command execution
  - `attr_read()` method for attribute reading
  - `attr_written()` method for attribute writing
- For callback, by default, a client is in pull model. Use  
`ApiUtil::set_asynch_cb_sub_model()` to change the sub-model



# Asynchronous call

```
using namespace Tango;
class MyCb:CallBack
{
public:
    MyCb(double d): data(d) {};
    void cmd_ended(CmdDoneEvent *);
```

private:

```
    double data;
```

```
DeviceProxy dev(...);
double my_data = 3.2;

MyCb cb(my_data);
dev.command_inout_ynch("MyCmd",cb);
....
```

```
dev.get_asynch_replies(150);
```

```
Void MyCb::cmd_ended(CmdDoneEvent *cmd)
{
    if (cmd->err == true)
        Tango::Except::print_error_stack(cmd->errors);
    else
    {
        short cmd_result;
        cmd->argout >> cmd_result;
        cout << "Cmd = " << cmd_result;
        cout << ", perso = " << data << endl;
    }
}
```



## Exercise 6 (ohhh yes)

- Rewrite your client from Exercise 5 with an asynchronous reading of the attributes state, wind speed and ScaAttr.
- Send the asynchronous reading request, do some work (sleep for some seconds) and get the reading results.



# Group Call

- Provides a single point of control for a Group of devices
- Group calls are executed asynchronously!
- You create a group of device(s) with the Tango::Group class
  - It's a hierarchical object (You can have a group in a group) with a forward or not forward feature
- You execute a command (or R/W attribute) on the group

```
Tango::GroupCmdReplyList crl = g1->command_inout("Status");
```



# Group Call

- Using groups, you can
  - Execute one command
    - Without argument
    - With the same input argument to all group members
    - With different input arguments for group members
  - Read one attribute
  - Write one attribute
    - With same input value for all group members
    - With different input value for group members



# Tango Training: Part 5 : More info on Device Servers

- The Administration Device
- The Logging System
- The Polling





# The Administration Device

- Every device server has an administration device
- Device name
  - dserver/<exec name>/<instance name>
- This device supports 20 (23) commands and 0 (2) attributes
  - 8 miscellaneous commands
  - 7 commands for the logging system
  - 1 command for the event system
  - 7 commands for the polling system



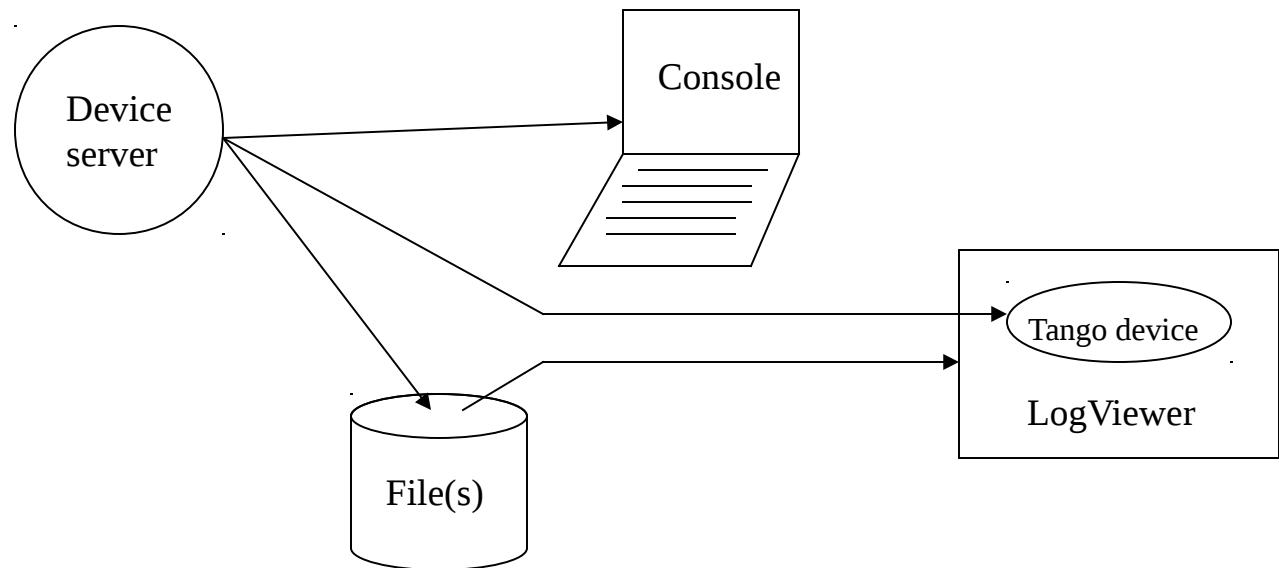
# The Administration Device

- Miscellaneous commands
  - DevRestart destroy and re-create a Device. The client needs to re-connect to the device
  - RestartServer to restart a complete device server
  - QueryClass to get the list of available classes
  - QueryDevice to get the list of available devices
  - Kill to kill the process
  - State, Status, Init

T  
A  
N  
G  


# The Tango Logging System

- Send device server messages to a target
  - A file
  - The console
  - A centralized application called **LogViewer**





# The Tango Logging System

- Each Tango device has a logging level
- Each logging request also has a logging level
- Six ordered logging levels are defined
  - DEBUG < INFO < WARN < ERROR < FATAL < OFF
- Each logging request with a level lower than the device logging level is ignored
- Device default logging level is WARN



# The Tango Logging System

- Five macros to send logging messages
  - C++ streams like
    - {FATAL, ERROR, WARN, INFO, DEBUG}\_STREAM
  - C printf like
    - LOG\_{FATAL, ERROR, WARN, INFO, DEBUG}
- Usage :

```
DEBUG_STREAM << "Hola amigo, que tal ?" << endl;
```

```
LOG_DEBUG(("Still %d hours of this f... training!\n",nb_hours));
```

# The Tango Logging System

- Logging on a console
  - Send messages to the console on which the device server has been started
- Logging in a file
  - Logging message stored in a XML file
  - Manage 2 files
  - Swap files when file size is greater than a pre-defined value (a property). Rename the old one as “xxx\_1”. Default file size threshold is 2 MBytes
  - Default file names: “/tmp/tango/process/instance/device.log” or “C:\tango\....” (create directory by hand...)
  - Read files with the “LogViewer” application



# The Tango Logging System

- Logging with the LogViewer
  - Send messages to a Tango device embedded in the LogViewer application
- LogViewer (Java appl.)
  - Graphical application to display, filter and sort logging messages
  - Two modes
    - Static: Memorize a list of Tango devices for which it will get/display messages
    - Dynamic: The user (with a GUI) chooses devices for which messages must be displayed



# The Tango Logging System

- Seven administration device commands dedicated to logging
  - AddLoggingTarget
  - RemoveLoggingTarget
  - GetLoggingTarget
  - GetLoggingLevel
  - SetLoggingLevel
  - StopLogging
  - StartLogging





# The Tango Logging System

- Logging configuration with Jive
  - current\_logging\_level
    - Not memorized
  - logging\_level
    - Memorized in db
  - current\_Logging\_target
    - Not memorized
    - console::cout, file::/tmp/toto or device::tmp/log/xxx
  - logging\_target
    - Memorized in db



# The Tango Logging System

- Each device server has a “-v” option
  - v1 and v2
    - Level = INFO and target = console::cout for all DS devices
  - v3 and v4
    - Level = DEBUG and target = console::cout for all DS devices
  - v5
    - Like v4 plus library messages (there are many) on target = console::cout
  - Without level is a synonym for –v4



# Exercise 7 (It starts again!!!)

- Experiment the administration device
- Ask the device server to generate logging to
  - A file
  - The LogViewer



# The Polling

- Each Tango device server has a polling thread
- It's possible to poll attributes and/or commands (without input parameters)
- The polling result is stored in a polling buffer (round robin buffer)
- Each device has its own polling buffer
- Polling buffer depth is tunable
  - By device (default is 10)
  - By command/attribute



# The Polling

- A client is able to read data from
  - The real device
  - The last record in the polling buffer
  - The polling buffer and in case of error from the real device
  - The choice is done with the *DeviceProxy::set\_source()* method
- A network call to read the complete polling buffer is also provided (*command\_inout\_history* or *read\_attribute\_history* defined in the Tango IDL)
  - *DeviceProxy::command\_history()* or  
*DeviceProxy::attribute\_history()*

# The Polling

- Seven administration device commands allow the polling configuration
  - AddObjPolling
  - RemObjPolling
  - UpdObjPolling
  - StartPolling
  - StopPolling
  - PolledDevice
  - DevPollStatus





# The Polling

- How it starts ?
  - At device startup if, configured via the administration device (Jive)
- For completeness
  - Externally triggering mode (C++ DS only)
  - External polling buffer filling (C++ DS only)
    - Get data with the command\_inout\_history or read\_attribute\_history calls



# Exercise 8 (Oh, no ...)

- Add another attribute (SlowAttr – Scalar – RO – Float) which needs 300 mS before returning (usleep)
- Start the polling from Jive and look at the response time according to the “source” parameter
- Stop polling with Jive
- Mark the attribute as polled in Pogo
- Make
- Restart and check what happens

T  
A  
N  
G  


# Tango Training: Part 6 : Events



# Events

- Another way to write applications
  - Applications do not poll any more
  - The device server informs the applications that “something” has happened
- Polling done by the device server polling thread
- Uses a CORBA service called “Notification Service”
- Tango uses omniNotify as Notification Service



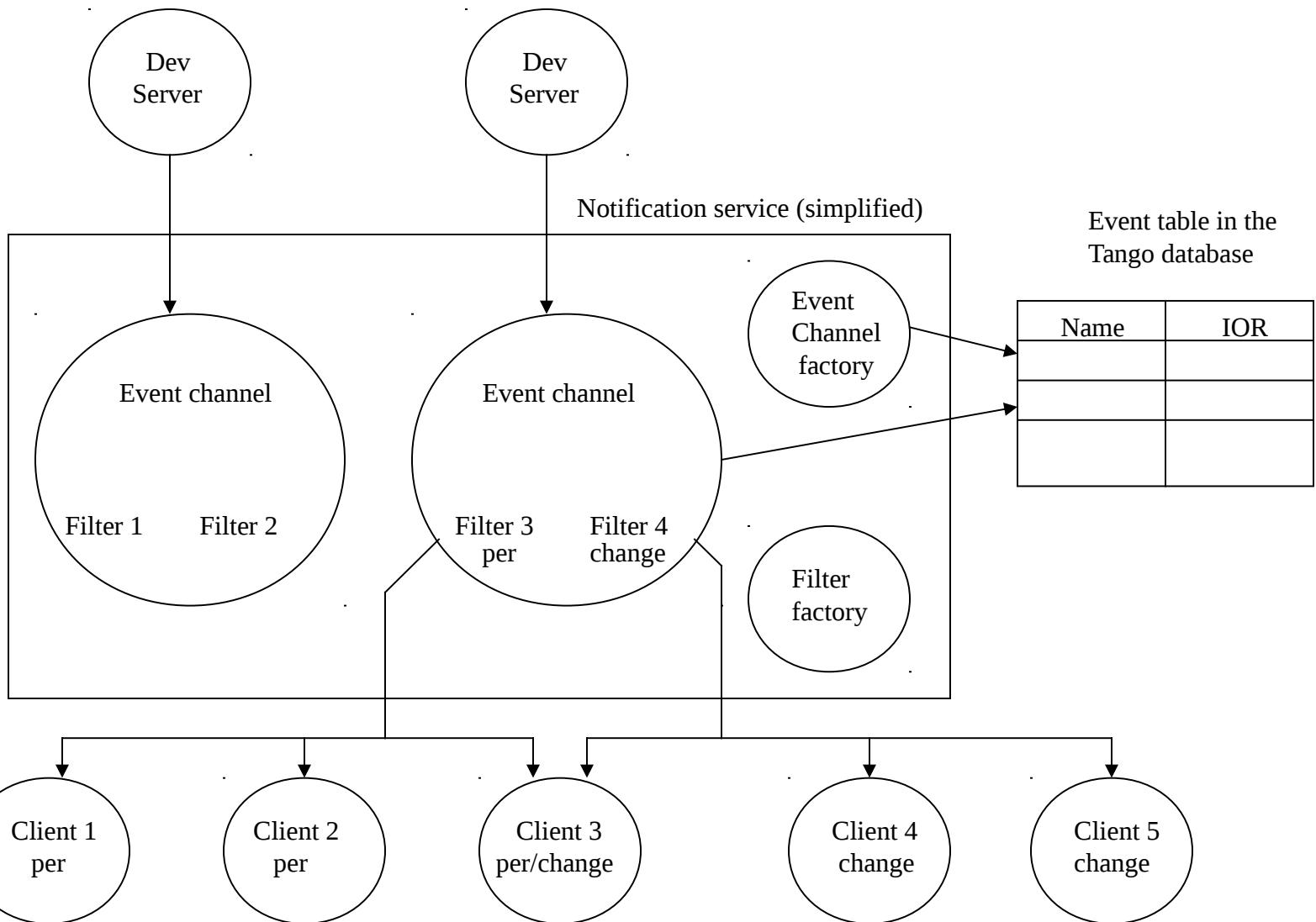


# Events

- One Notification service daemon (notifd) running on each host
- Event propagation
  - The event is detected by the polling thread and sent to the notification service
  - The notification service sends the event to all the registered client(s)
- It is possible to ask the notification service to filter events

# Events

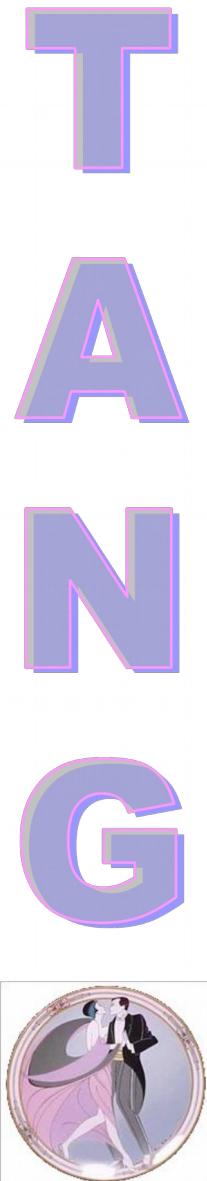
T  
A  
N  
G





# Events

- Only available on attributes!
- Does not require any changes in the device server code
- Based on callbacks. The client callback is executed when an event was received
  - Event data or an error stack in case of an exception
- 5 types of events
  - Periodic, Change, Archive
  - Attribute configuration change, User defined



# Events

- Periodic event
  - Event pushed:
    - At event subscription
    - On a periodic basis
- Change event
  - Event pushed when
    - a change is detected in attribute data
    - a change is detected in attribute size (spectrum/image)
    - At event subscription
    - An exception was received by the polling thread
    - the attribute quality factor changes
    - When the exception disappears



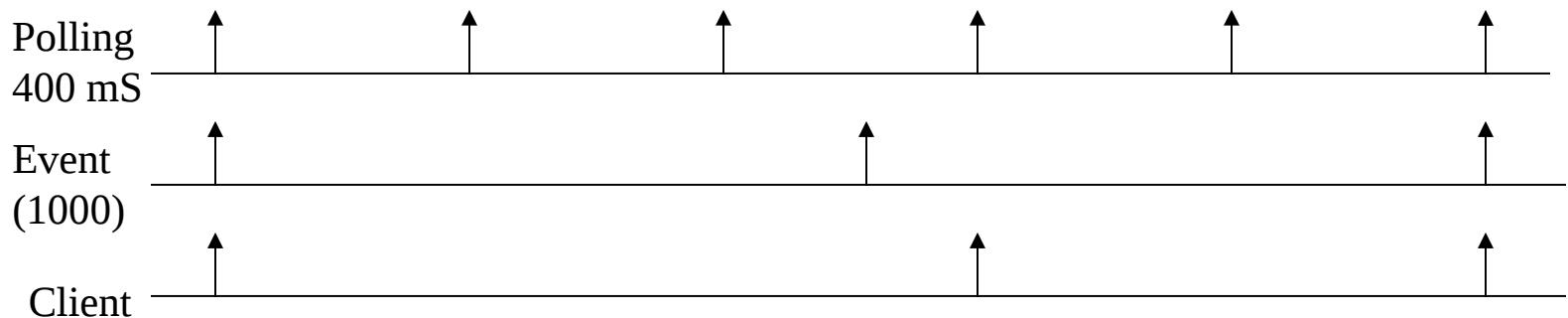
# Events

- Archive event
  - A mix of periodic and change events
- Attribute configuration change
  - Event pushed when:
    - At event subscription
    - The attribute configuration is modified with `set_attribute_config()`
- User defined event
  - Event pushed when the user decides it



# Events (configuration)

- Periodic event configuration
  - event\_period (in mS).
    - Default is 1000 mS
    - Cannot be faster than the polling period
  - Polling period != event period
  - The event system does not change the attribute polling period if already defined





# Events (configuration)

- Change event configuration
  - Checked at the polling period
  - rel\_change and abs\_change
    - Up to 2 values (positive, negative delta)
    - If both are set, relative change is checked first
    - If none is set -> **no change event!**



# Events (configuration)

- Archive event configuration
  - Checked at the polling period
  - `event_period` (in mS).
    - Default is 0 mS -> **no periodic archive event!**
  - `rel_change` and `abs_change`
    - Up to 2 values (positive, negative delta)
    - If both are set, relative change is checked first
    - If none is set -> **no archive event on change!**



# Events (configuration)

- Event configuration parameters  
(event\_period, abs\_change, rel\_change...) are part of the attribute configuration properties
- Can be configured with Jive



# Events (pushed from the code)

- Only possible for change and archive events
- To push events manually from the code a set of data type dependent methods can be used:  
`DeviceImpl::push_change_event (string attr_name, ....);`  
`DeviceImpl::push_archive_event(string attr_name, ....);`
- It is possible to push events from the code and from the polling thread at the same time
- Attribute configuration with Pogo



# Events (pushed from the code)

- To allow a client to subscribe to events of non polled attributes the server has to declare that events are pushed from the code

```
DeviceImpl::set_change_event( string attr_name,  
                                bool implemented, bool detect = true);
```

```
DeviceImpl::set_archive_event( string attr_name,  
                                bool implemented, bool detect = true);
```

- *implemented=true* indicates that events are pushed manually from the code
- *detect=true* triggers the verification of the same event properties as for events send by the polling thread.
- *detect=false*, no value checking is done on the pushed value!



# Events (filtering)

- When you subscribe to an event, you may ask for a filters
- **All filters are compared to the last event value send and not to the actual attribute value!**
- Periodic event filter
  - Filterable data name : “counter”
  - Incremented each time the event is sent
  - Ex : “\$counter % 2 == 0”



# Events (filtering)

## ■ Change event filters are

- “quality” is true when the event was pushed on a quality change
  - “Ex: \$quality == 1
- “forced\_event” is true when the event was pushed due to an exception, an exception change or when the exception disappears
- “delta\_change\_rel” and “delta\_change\_abs” contain the change detected by server compared to the last event pushed
  - Ex : “\$delta\_change\_abs >= 2”



# Events (filtering)

- Archive event filters are
  - “counter” as for the periodic event
  - “quality” and “forced\_event” as for the change event
  - “delta\_change\_rel” and “delta\_change\_abs” as for the change event
  - “delta\_event” contains the delta time in ms since the last archive event was pushed
    - Ex: “\$delta\_event >= 2000”



# Events (heartbeat)

- To check that the device server is alive
  - A specific “heartbeat event” is sent every 10 seconds to all clients connected on the event channel
- To inform the server that no more clients are interested in events
  - A re-subscription command is sent by the client every 200 seconds. The device server stops sending events as soon as the last subscription command is older than 600 seconds



# Events (heartbeat)

- A dedicated client thread (KeepAliveThread) wakes up every 10 seconds to check the server's 10 seconds heartbeat and to send the subscription command periodically.

# Events (threading)

## On the client side

- As soon as you create a DeviceProxy -> 2 threads (main thread + omniORB scavenger thread)
- First event subscription adds 3 threads
  - (orb thread, omniORB thread and KeepAliveThread)
- Clients are servers : One more thread per Notification service sending events to the client
- thread number:  $5 + n$  ( $n$  = Notif service connected (+1 for linux))
- Warning : Callbacks are not executed by the main thread !

## On the server side

- No changes



# Events (client side)

- Event subscription with the *DeviceProxy::subscribe\_event()* method
- Event un-subscription with the *DeviceProxy::unsubscribe\_event()* method
- Call-back idem to asynchronous call
  - Method *push\_event()* to overwrite in your class
  - This method receives a pointer to an instance of a *Tango::EventData* class

# Events (client side)

```
int DeviceProxy::subscribe_event( const string &attribute, EventType event  
                                CallBack *cb, const vector<string> &filters,  
                                bool stateless);
```

```
void DeviceProxy::unsubscribe_event(int event_id);
```

```
class CallBack  
{  
....  
    virtual void push_event(EventData *);  
}
```

```
class EventData  
{  
    DeviceProxy *dev;  
    string &attr_name;  
    string &event;  
    DeviceAttribute *attr_value;  
    bool err;  
    DevErrorList &errors;  
}
```



# Exercise 9 (Uff ...)

- Test set up
  - Add another attribute to your class (TestEvent – Scalar – Double – RO – Not polled)
  - Add a command which increments by 2 the TestEvent attribute (IncrAtt – void –void)
- Start the notification service and register the service to the Tango database
  - notifd –n
  - notifd2db
- Write a client which subscribes to a change event and sleeps waiting for events



# Tango Training: Part 7 : Device Server Level 2...

- Several classes in the same device server
- Threading model
- Abstract classes
- Device servers on Windows





# Multi Classes Device Server

- Define which Tango classes are embedded in your server in the `class_factory` file.
- To communicate between classes, uses the `DeviceProxy` instance
- All devices of all classes are “exported”
- Classes are created in the order defined in the `class_factory` file and destroyed in the reverse order
- Obviously, modify the linker command line



# Multi Classes Device Server

## ■ Example of multi classes class\_factory file

```
#include <tango.h>
#include <SerialClass.h>
#include <ParagonClass.h>
#include <PLCmodbusClass.h>
#include <IRMirrorClass.h>

void Tango::DServer::class_factory()
{
    add_class(Serial_ns::SerialClass::init("Serial"));
    add_class(Paragon_ns::ParagonClass::init("Paragon"));
    add_class(PLCmodbus::PLCmodbusClass::init("PLCmodbus"));
    add_class(IRMirror_ns::IRMirrorClass::init("IRMirror"));
}
```



# The Threading Model

- omniORB is a multi-threaded ORB
  - A Tango device server also...
- One thread is created in a device server for each client
- A scavenger thread destroy thread(s) associated to unused connections (omniORB feature)
- Not always adapted to hardware access
- Tango also has its own polling thread



# The Threading Model

- Each Tango device has a monitor to serialize the device access.
- Four modes of serialization
  - By device (the default)
  - By class (one monitor by Tango class)
    - Use this model if your Tango device need to access a non threadsafe library
  - By process (one monitor for the whole Tango device server)
  - No serialization (**extreme care**)



# The Threading Model

- The *Util::set\_serial\_mode()* method is used to set the serialization model  
(in the main function)

```
int main(int argc, char *argv[])
{
    try
    {
        Tango::Util *tg = Tango::Util::init(argc,argv);

        tg->set_serial_model(Tango::BY_CLASS);

        tg->server_init();
        ....
```



# Abstract Classes

- Based on the C++ abstract classes (or Java interfaces)
- A way to standardize interfaces
  - What is the minimum number of commands/attributes that my kind of device should provide
  - Write an abstract class which defines only this minimum (no code) with Pogo
  - Write the concrete class which inherits from the abstract class

T  
A  
N  
G

# Abstract Classes

- This allows to have a minimum common interface/behavior for the same type of device
- If possible, an application uses only the minimum interface defined in the abstract class and is independent of the real hardware
- Pogo also supports writing of the abstract class itself.





# Device Server on Windows

- Two kinds of Tango device servers on Windows
  - Running as a Windows console application
    - No changes
  - Running as a Windows application
    - Written using MFC
    - Written using Win32 API

T  
A  
N  
G  


# DS on Windows



tst\_mfc/et - Console

```
1117783317 [2224] DEBUG dserver/tst_mfc/et In add_obj_polling method
1117783317 [2224] DEBUG dserver/tst_mfc/et Input string = et/mfc/1
1117783317 [2224] DEBUG dserver/tst_mfc/et Input string = command
1117783317 [2224] DEBUG dserver/tst_mfc/et Input string = readvalue
1117783317 [2224] DEBUG dserver/tst_mfc/et Input long = 2000
1117783317 [2440] DEBUG dserver/tst_mfc/et Received an exit command
1117783317 [2224] DEBUG dserver/tst_mfc/et Entering Util::unregister_server method
1117783317 [2224] DEBUG dserver/tst_mfc/et Leaving Util::unregister_server method
1117783317 [2224] DEBUG dserver/tst_mfc/et Entering DeviceClass destructor for class Tst_m
1117783317 [2224] DEBUG dserver/tst_mfc/et Entering DeviceImpl destructor for device et/mfc
1117783317 [2224] DEBUG dserver/tst_mfc/et Leaving DeviceImpl destructor for device et/mfc
1117783317 [2224] DEBUG dserver/tst_mfc/et Leaving DeviceClass destructor for class Tst_m
1117783317 [2224] DEBUG dserver/tst_mfc/et Going to shutdown ORB
1117783317 [2224] DEBUG dserver/tst_mfc/et ORB shutdown
```

09/02/17

# DS on Windows

- With the Win32 API
  - Very similar to a traditional “main” but
    - Replace main by WinMain
    - Display message box for errors occurring during the device server start-up phase
    - Code the Windows message loop
  - See example in doc chapter 8.5.3
- With MFC, see chapter 8.5.2
- Don’t forget to link your device server with the Tango windows resource file



# Device Server on Windows

- Take extreme care with the kind of libraries used for linking (No mix)
- Tango supports
  - Multithreaded (/MT)
  - Debug Multithreaded (/MTd)
  - Multithreaded DLL (/MD)
  - Debug Multithreaded DLL (/MDd)



# Tango Training:

## Part 8 : Advanced Features

- OS signals
- Attribute alarms
- I don't like database
- Multi CS / Multi DB
- DS and CVS
- Tango/Jive wizard
- Windows service





# OS signals in Device Server

- It is UNSAFE to do what you want in a signal handler
- Device servers provide a dedicated thread for a signal handler.
  - You can code what you want in a Tango device signal handler
- Use the *register\_signal()* and *unregister\_signal()* methods to register/unregister signal handlers

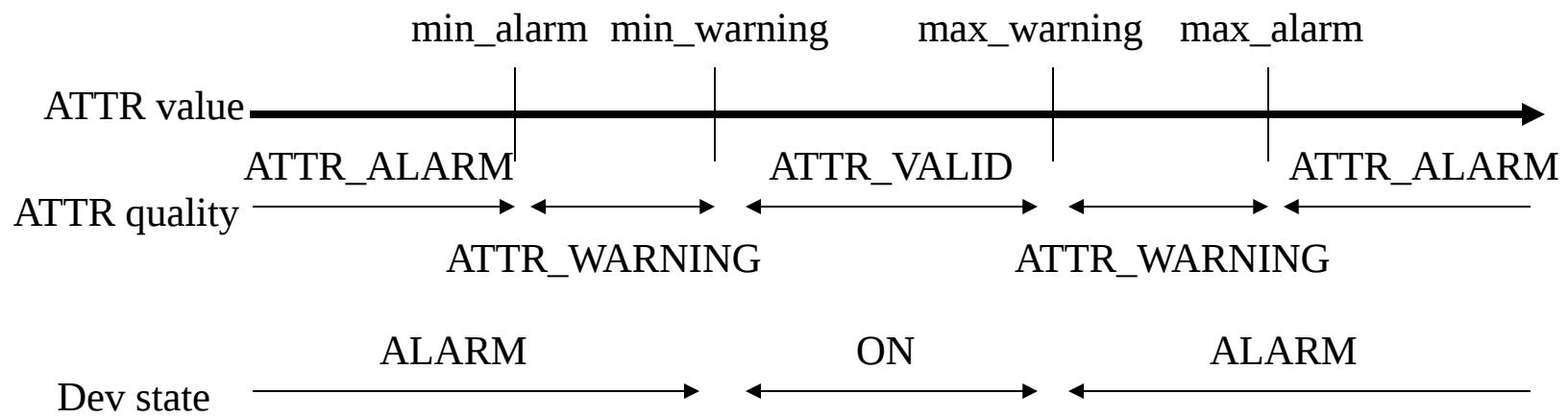
# OS signal in Device Server

- Code your handler in the *signal\_handler()* method
- You can install a *signal\_handler* on a device basis if you filter the registering/un-registering methods
- It is also possible to install a signal handler at class level



# Attribute Alarms

- Two types of alarms
  - On value
  - On read different than set
- Alarm on value
  - Two thresholds called ALARM and WARNING





# Attribute Alarms

- Read different than set alarm
  - Two parameters to tune this alarm
    - The authorized delta on value
    - The delta time between the last attribute setting and the attribute value check
  - Obviously, only on Read-Write attributes and not available for string and boolean



# Attribute Alarms

- Six parameters to tune the alarm part of the attribute configuration
  - min\_alarm, min\_warning, max\_warning, max\_alarm
  - delta\_t, delta\_val
- Attribute alarms are checked during the State command (attribute) execution



# DS using a File as Database

- Tango device server supports using a file instead of the database
- Generate the file with Jive
- It is possible
  - Get, update, delete class properties
  - Get, update, delete device properties
  - Get, update, delete class attribute properties
  - Get, update, delete device attribute properties



# DS using a File as Database

- Start the device server on a specified port

```
MyDs inst -file=<file_path> -ORBendPoint giop:tcp::<port>
```

- Device name used in a client must be changed
  - With database:
    - sr/d-fuse/c04
  - With file as database:
    - tango://<host>:<port>/sr/d-fuse/c04#dbase=no



# DS using a File as Database

## ■ Limitations

- The delete is not reported back to DB
- No check that the same device server is running twice
- Manual management of host/port
- No alias



# DS not using a Database at all!

- Also possible to start a device server without using a database at all
  - Do not code database access within the device server...
- The option is **-nodb**
- Another option –dlist allow the definition of device names at the command line for the highest tango class



# DS not using a Database at all

- A method DeviceClass::device\_name\_factory is used to define device names for a class, when it is not possible to define them at command line

```
MyDs inst -nodb -dlist id13/pen/1,id13/motor/2  
-ORBendPoint giop:tcp::<port>
```



# DS not using a Database at all

- Change of device name
  - tango://<host>:<port>/sr/d-fuse/c04#dbase=no
- Limitation
  - The same as for a server with file database
  - No properties at all
  - No events



# Multi TANGO\_HOST

- A client running in control system A is able to access device running in control system B by specifying the correct name
- Full Tango device name syntax

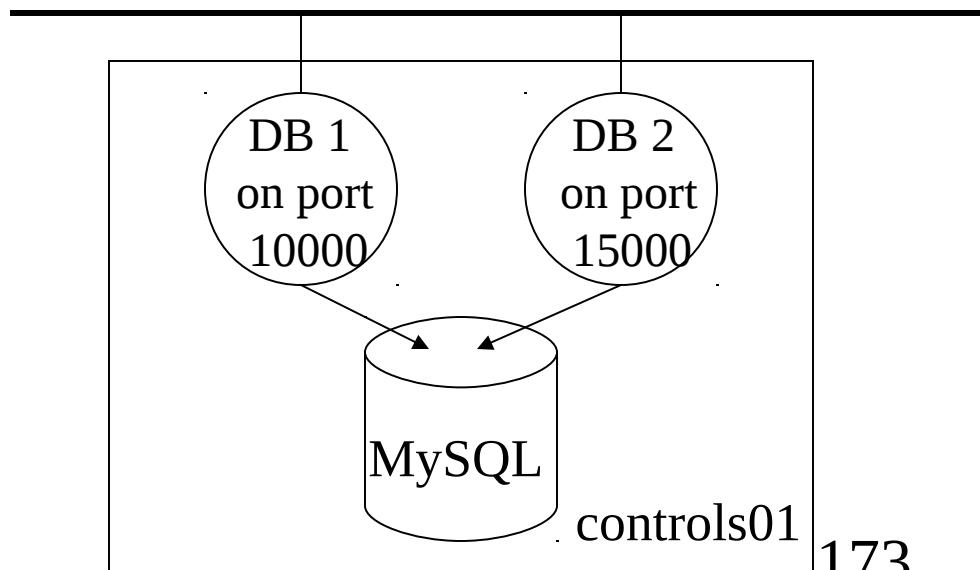
```
[protocol://][host:port]device_name[/attribute][->property][#dbase=xx]
```

- Examples
  - tango://freak:1234/id00/pen/c11#dbase=no
  - tango:///orion:10000/sr/d-vlm/1
  - sr/d-irm/1/position->label

# Tango Control System with Several Database Servers

- Defined using the TANGO\_HOST environment variable
- Client and servers will automatically switch from one server to the other if one dies

```
TANGO_HOST=controls01:10000,controls01:15000
```





# Device Servers and CVS

- In the xxxClass, POGO creates static strings for :
  - The CVS tag
  - The CVS Header
- Class properties called “cvs\_tag” and “cvs\_location” are created/updated in the device server startup sequence
- The device “info” remote call (available from DeviceTest) returns these values in its doc\_url field.



# Device Servers and CVS

- POGO generates a Makefile with a make entry “tag” which will tag the device server in the CVS with a tag “Release\_X\_Y”
- Define X and Y with the variables MAJOR\_VERS and MINOR\_VERS in the Makefile



# The Wizard

- Available from Jive or Astor
- Allows a user to create and configure a new device server in the database without knowing device properties
- The wizard will
  - Automatically retrieve class properties and will ask for new value
  - Automatically retrieve device properties and will ask for new value



# DS as Windows Service

- A Tango device server is able to run as a Windows service but
  - Needs changes in the code (See doc chapter 8.5.4)
  - Needs to be registered in the Windows service manager
    - A new set of options is available when a device server is used as a Windows service
      - -i, -u or -s



# Tango training: Part 9: The Python binding: PyTango

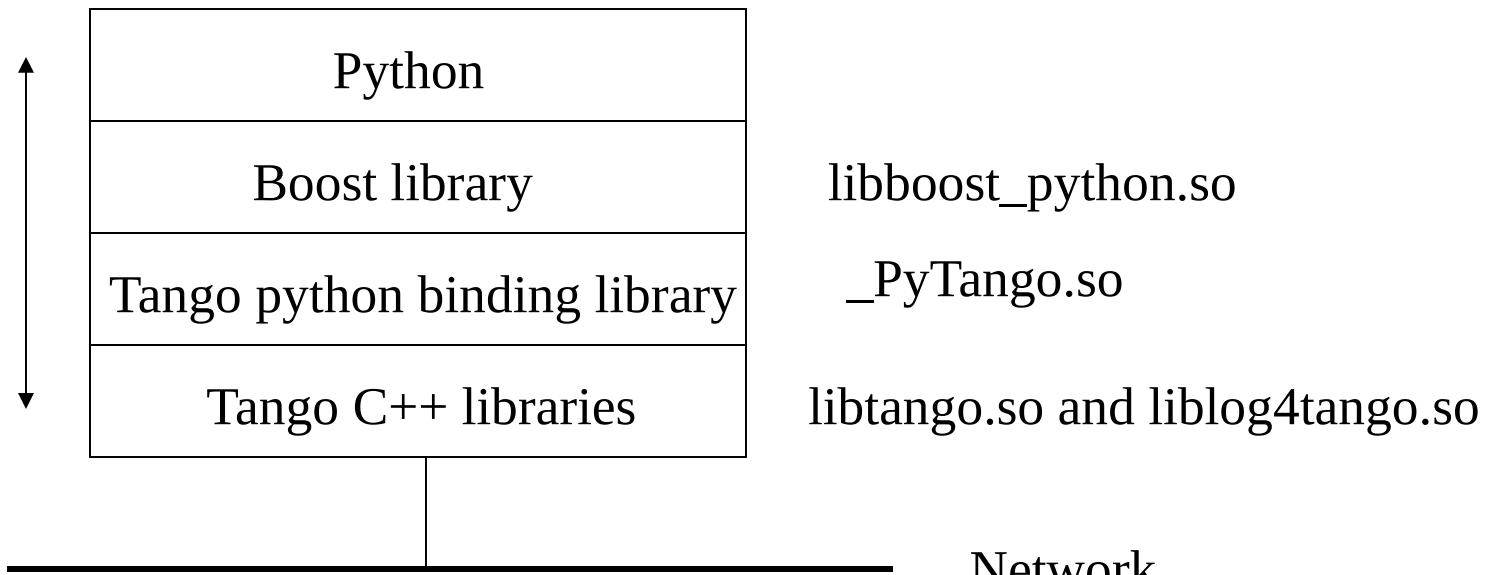
- Basic architecture
- Clients
- Servers





# Python Binding

- Based on the C++ API and boost for the C++ to Python link (<http://www.boost.org/>)





# Python Binding

- Module name = **PyTango** and it's actual release is 3.0.4.
- To use it, you need to have:
  - In your LD\_LIBRARY\_PATH
    - The boost release 1.33 (or more) library
    - The Tango and ORB libraries
  - In your PYTHONPATH
    - The PyTango python package (Explained in the package ReadMe file)

# Python Binding



## ■ Release 1 supports

- Sending command to device
- Read/Write attribute device
- Getting polled objects history
- Interface to the Tango database  
(getting/updating/deleting properties....)

## ■ Release 2 adds

- Added support for new Attribute data type introduced in Tango V5
- Support for event
- Support for AttributeProxy

## ■ Release 3 adds

- The possibility to write Tango device servers in Python.
- Groups calls



# Python Binding

```
>>>from PyTango import *
>>>d=DeviceProxy("et/echo/1")
>>>d.state()
PyTango.DevState.ON
>>>res=d.command_inout("StringReverse","abcd")
>>>print res
dcba
>>>res=d.command_inout("EchoShort",20)
>>>print res
40
>>>res=d.StringReverse("efgh")
>>>print res
hgfe
>>>res=d.EchoShort(15)
>>>print res
30
```

- See the doc in “binding” page of the Tango WEB site

T  
A  
N  
G

# Python binding

- See the doc in “binding” page of the Tango WEB site

```
>>>from PyTango import *
>>>d=DeviceProxy("et/echo/1")
>>>dat=d.read_attribute("ScaAttr")
>>>print dat.value
2
>>>dat.value=4
>>>d.write_attribute(dat)
>>>d.write_attribute("ScaAttr",10)
>>>
```





# Python Binding

```
from PyTango import *

class PyCallback:
    def push_event(self,event):
        if not event.err:
            print event.attr_name, event.attr_value.value
        else:
            print event.errors

cb=PyCallback()

dev=DeviceProxy("dev/test/10")
print dev.Status()
print "DeviceProxy created"
ev=dev.subscribe_event("event_change_tst", EventType.CHANGE, cb, [])
print "After subscribe"
```



# Python Device Server

- All the commands are managed by one C++ command class (PyCmd)
  - One instance of this class per command
  - Command name is a data member of this class (Tango feature)
- All the attributes are managed by three C++ attribute classes (PyScaAttr, PySpecAttr and PyIma Attr)
  - One instance of one of these classes per attribute
  - Attribute name is a data member



# Python Device Server

## ■ Command

- Python methods executed by a command have well defined names
  - `is_<Cmd_name>_allowed()` (`is_On_allowed()`)
  - `<Cmd_name>` (`On()`)
- They are defined in a Python dictionary called **cmd\_list** data member of the xxxClass (PowerSupplyClass class)



# Python Device Server

## ■ Command

- cmd\_list dictionary :

- ‘cmd\_name’:[[in\_type,<“In desc”>],[out\_type,<“Out desc”>],<{Opt parameters}>]

```
cmd_list = {'IOLong',[[PyTango.ArgType.DevLong,"Number"],  
                     [PyTango.ArgType.DevLong,"Number * 2"],  
                     {"display level": PyTango.DispLevel.EXPERT}]}  
                    
```



# Python Device Server

- Command simple data types

Tango data type	Python type
Void	Nothing
Boolean	Python boolean
Short, Long, Unsigned short, Unsigned long	Python integer
Float, Double	Python float
String	Python string

# Python Device Server

## ■ Command array data types

Tango data type	Python type
Char, Short, Long, unsigned short, unsigned long array	Python list of Python integer
Float and Double array	Python list of Python float
String array	Python list of Python string
Long and String	Python tuple with a Python list of integer and a Python list of string
Double and String	Idem (integer -> float)



# Python Device Server

## ■ Command

- During device server startup phase, the code
  - Analyzes the cmd\_list dictionary
  - Checks that methods necessary to execute commands are defined
  - Checks the optional command parameters
  - Creates one instance of the PyCmd class for each valid command



# Python Device Server

## ■ Attribute

- Python methods executed when reading or writing attributes have well defined names
  - `is_<Attr_name>_allowed()` (`is_Current_allowed()`)
  - `read_<Attr_name>` (`read_Current()`)
  - `Write_<Attr_name>` (`write_Current()`)
- They are defined in a Python dictionary called **attr\_list** data member of the xxxClass (PowerSupplyClass class)



# Python Device Server

## ■ Attribute

- attr\_list dictionary :
  - ‘attr\_name’:[[attr data type, attr data format, attr data R/W type],<{opt parameters}>]

```
attr_list = {'Long_attr':[[PyTango.ArgType.DevLong,  
                         PyTango.AttrDataFormat.SCALAR,  
                         PyTango.AttrWriteType.READ],  
                        {"label": "A dummy attribute",  
                         "min alarm": 1000, "max alarm": 1500}]}  
                         
```



# Python Device Server

## ■ Attribute

- During device server startup phase, the code
  - Analyzes the attr\_list dictionary
  - Checks that methods necessary to read/write attributes are defined
  - Checks the optional attribute parameters
  - Creates one instance of the PyScaAttr, PySpecAttr or PylmaAttr class for each valid attribute



# Python Device Server

- General methods
  - *init\_device()*
  - *delete\_device()*
  - *always\_executed\_hook()*
  - *read\_attr\_hardware()*
  - *signal\_handler()*
- State and Status can be re-defined if needed



# Python Device Server

- Coding difference related to C++
  - Command's `is_allowed` methods does not have an argument
  - No `class_factory` file/method. Replaced by the `add_TgClass()` and `add_Cpp_TgClass()` methods of the Python PyUtil class
  - No `device_factory()` method



# Python Device Server

- Several Tango classes within the same interpreter
  - All Python Tango classes
    - No problem, use the client part of PyTango
  - Mixing Python and C++ Tango classes
    - Need a small C function (extern “C”) in C++ Tango classes to create it. The Tango class has to be created in a separate shared library.
    - Integrated in Pogo templates and Makefile



# Python Device Server

- Some timing measurement (2 CPU Xeon 3.2 Ghz Suse 9.3, 2 GB mem)
  - In = One Long, Out = One long
  - In = Void, Out = 100000 double (Python list)

	Python server	C++ server
Python client	120 us 36 mS	108 us 16 mS
C++ client	100 us 25 mS	85 us 5 mS



# Exercise (The last one ...)

- Send a command to your device from Python
- Write a small DS with Python:
  - One command with xxx



# Tango training: Part 10: ATK

- Goals
- Architecture
- Inside ATK
- Synoptic
- Examples



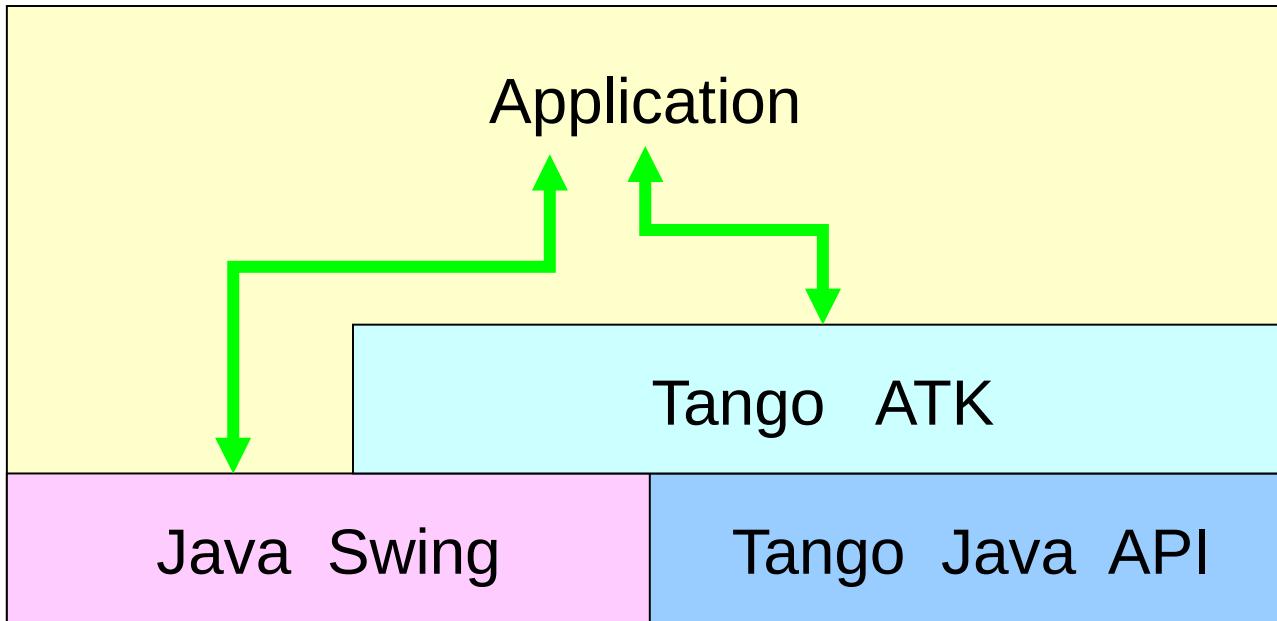


# Tango ATK goals

- Provide a framework to speed up the development of Tango Applications
- Help standardize the look and feel of the applications
- Implements the core of “any” Tango Java client
- Is extensible

T  
A  
N  
G

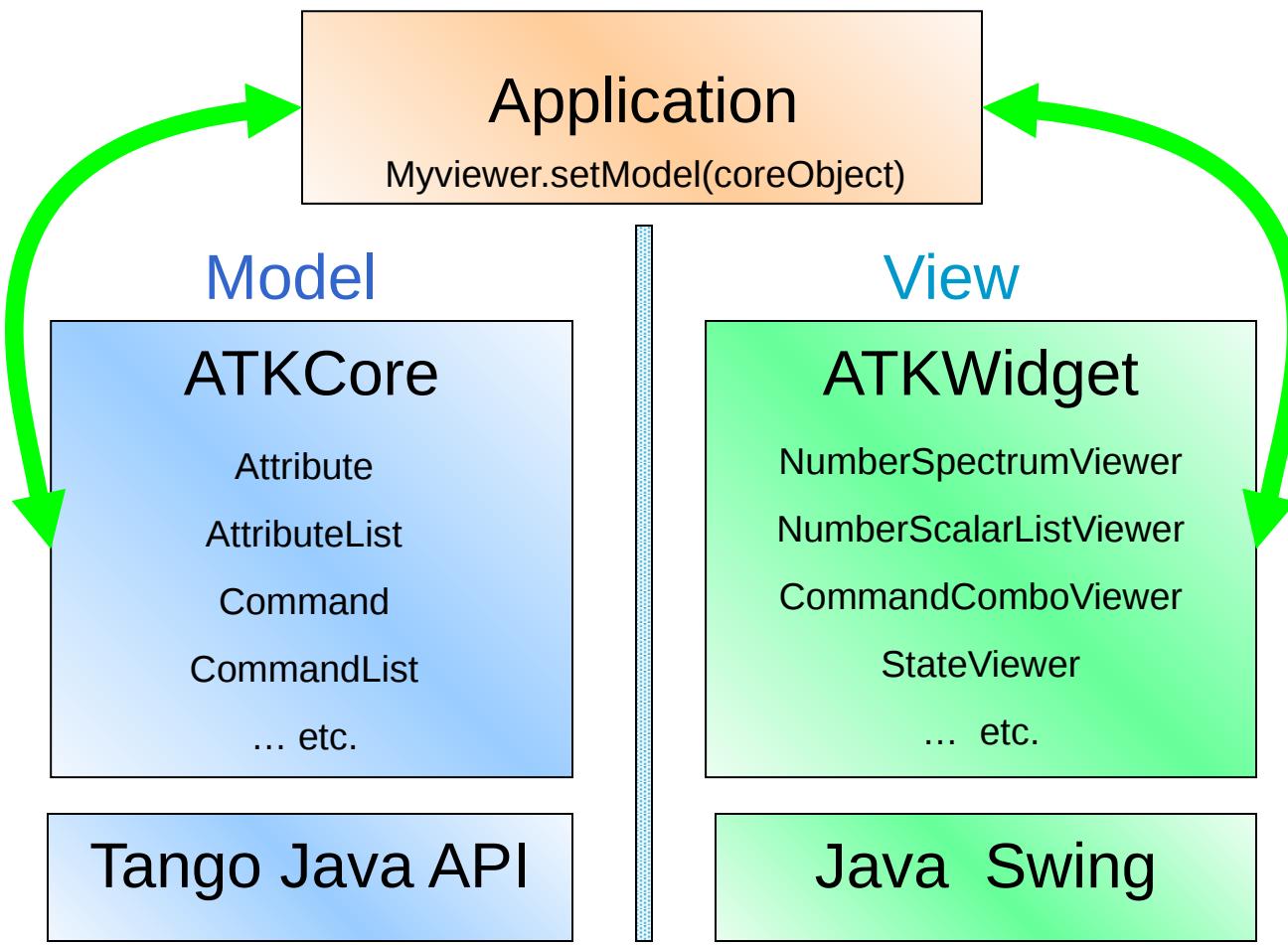
# Software Architecture



T  
A  
N  
G  

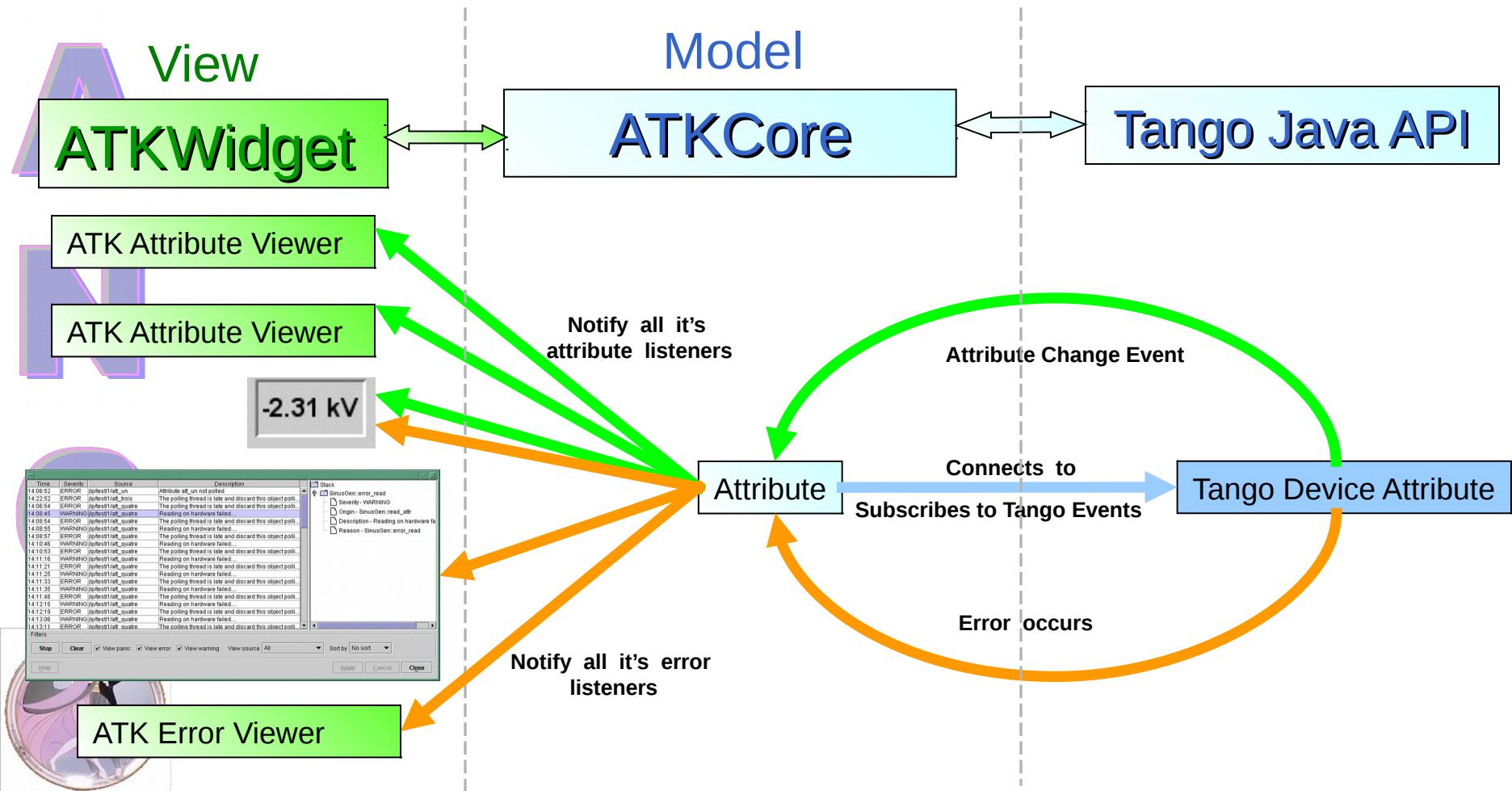

# Software Architecture

Control



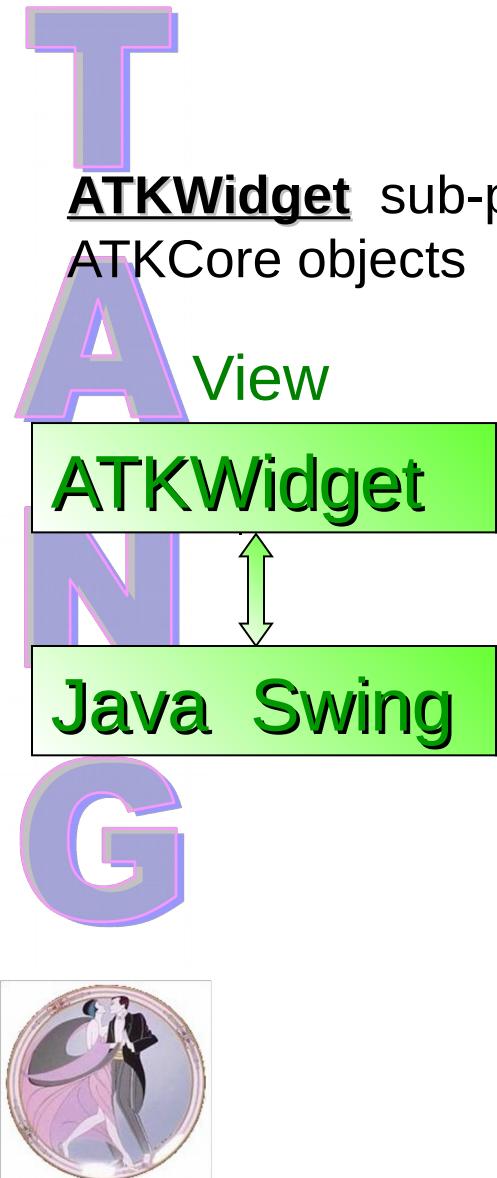
# Inside Tango ATK

**ATKCore** sub-package provides the classes which implement the model

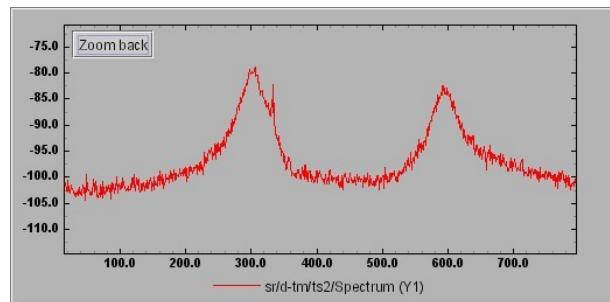


# Inside Tango ATK

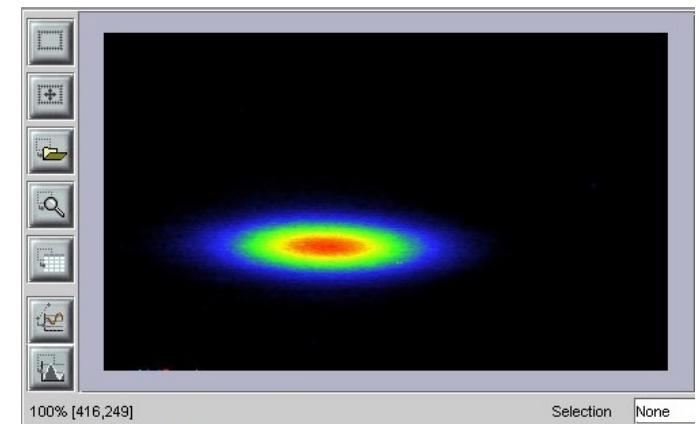
**ATKWidget** sub-package provides the classes to view and to interact with ATKCore objects



NumberSpectrumViewer



NumberImageViewer



CommandComboViewer

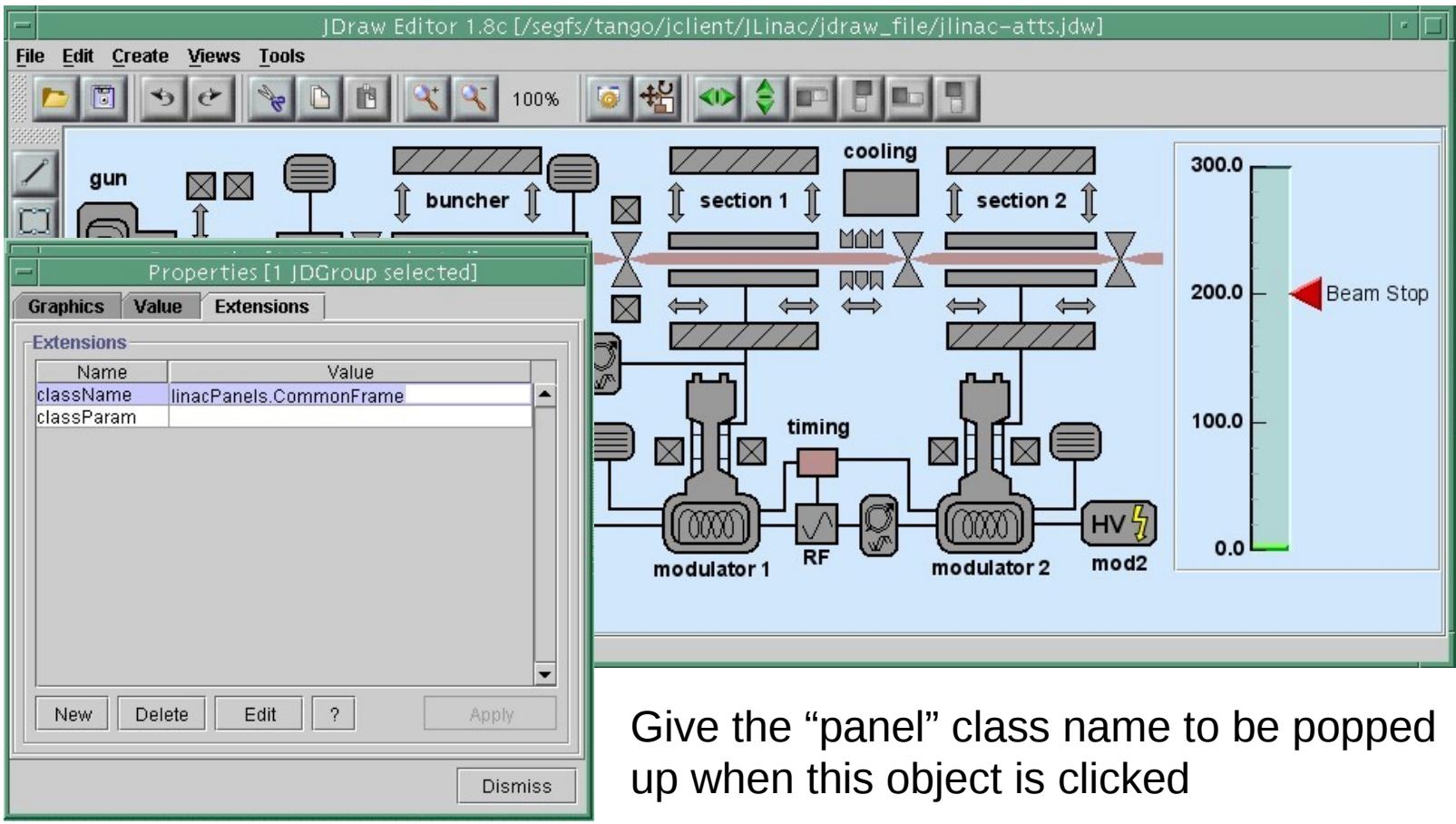


ScalarListViewer

Freq Min	100.000 khz	000100 .000	Bandwidth	0.030 khz
Freq Max	165.000 khz	000165 .000	Shaker Plane	W Noise
Reference level	-60.0 dB	-060 .0	Measure Plane	Diagonal
Scale/Div	5.0 dB	005 .0		Horizontal
N average	5	05		Vertical
Shaker Level	-30.0 dBm	-030 .0	Ave	Diagonal
Sweep Time	0.7 s	003 .2		Auxiliar
			C29 H Shaker	<input type="checkbox"/>
			C29 V Shaker	<input checked="" type="checkbox"/>

# Synoptic

Use Jdraw editor to draw the synoptic

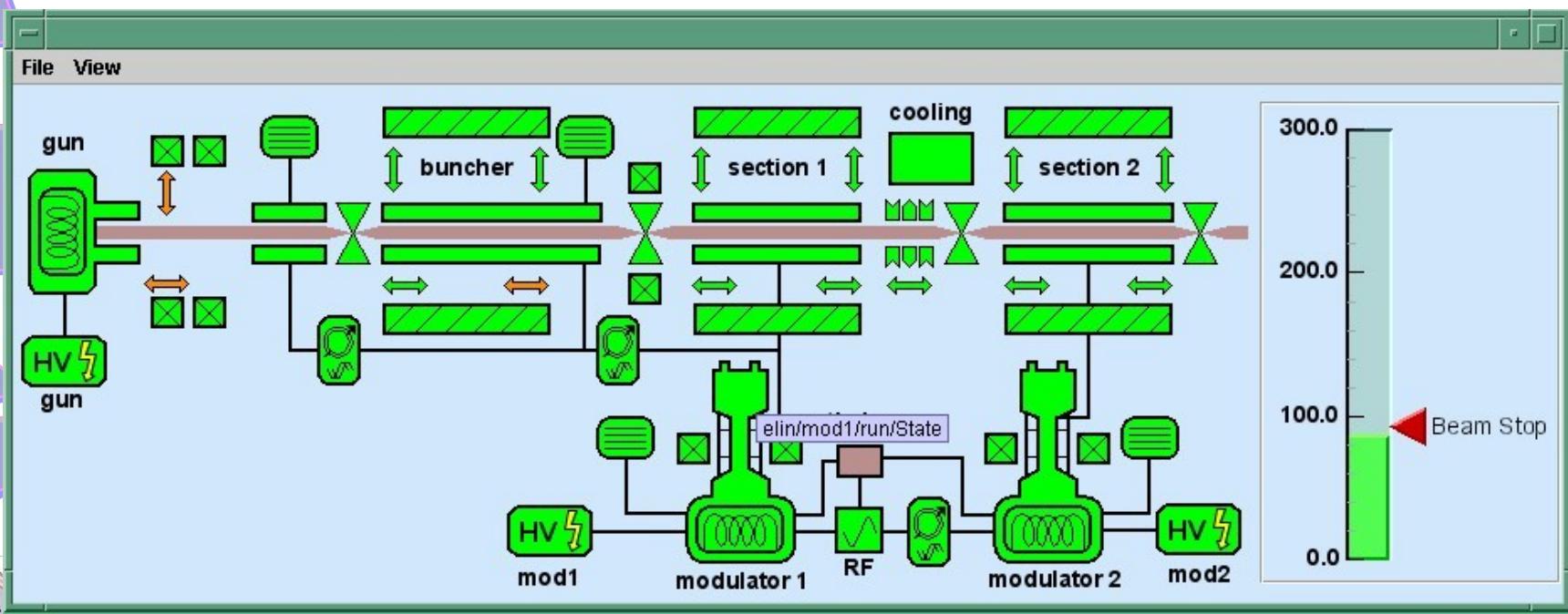


Give the “panel” class name to be popped up when this object is clicked



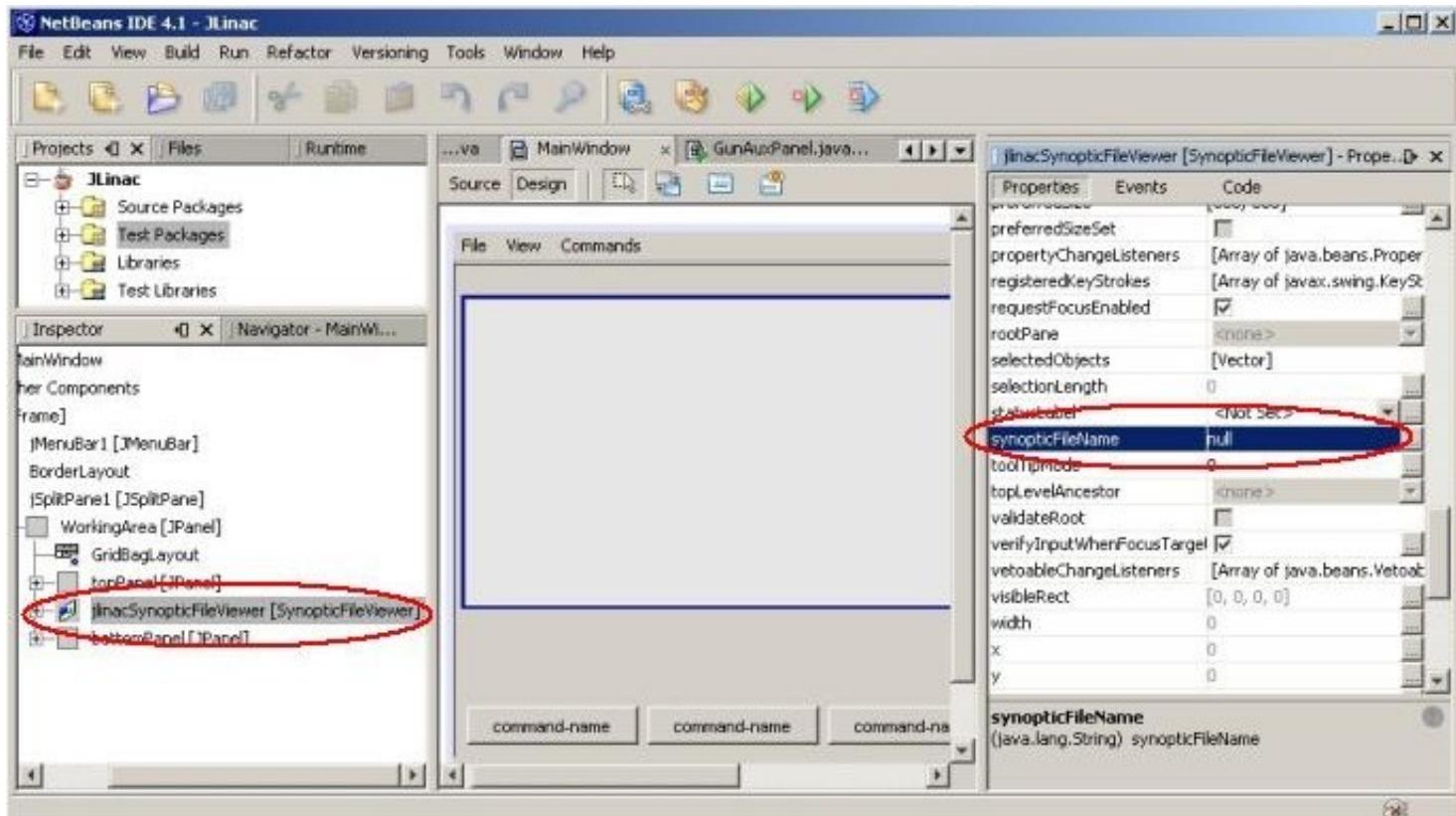
# Synoptic

Launch the ready to use ATK application “SimpleSynopticAppli” to test the synoptic at run time



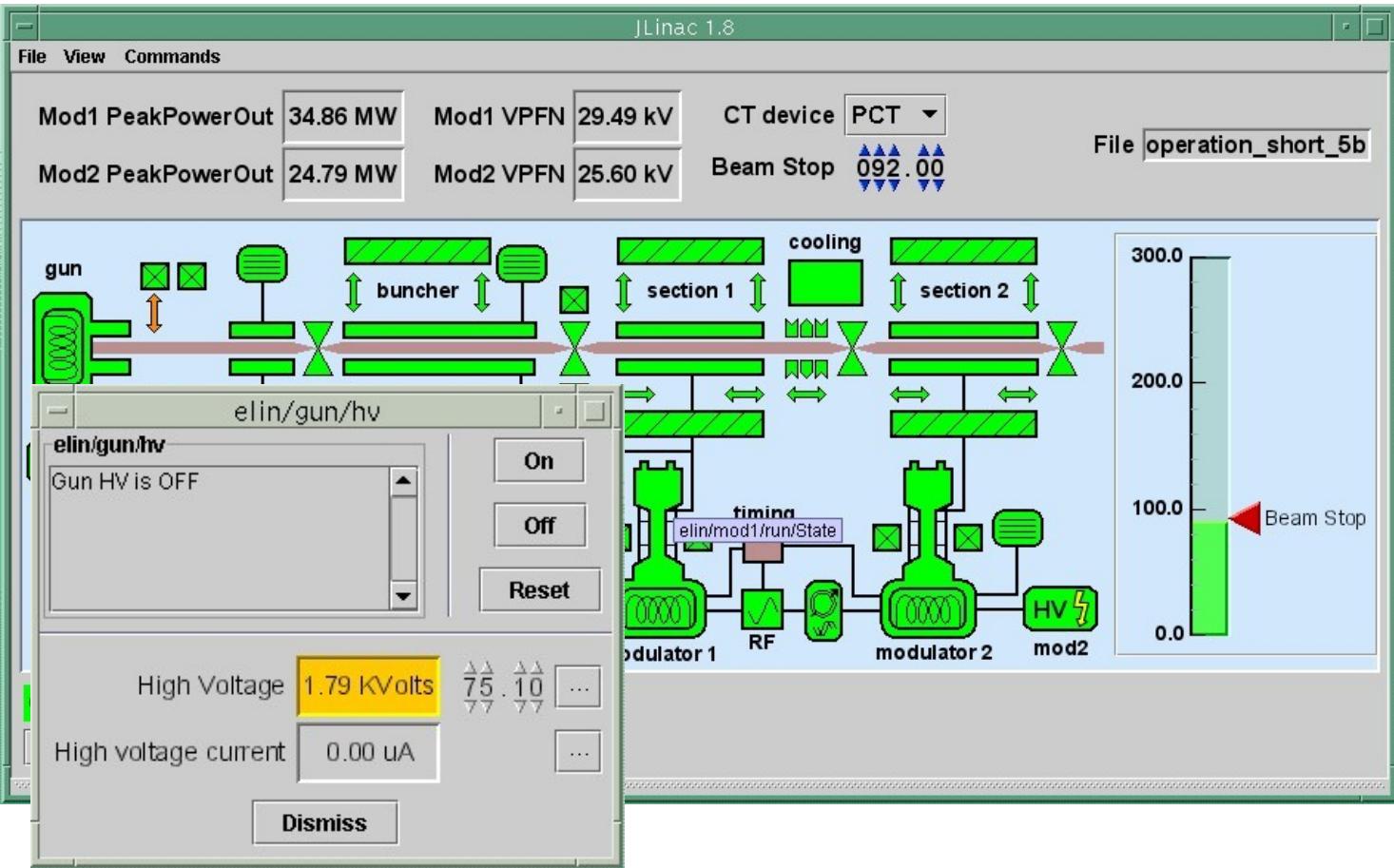
# Synoptic

Design your own specific ATK application using your favorite Java IDE



# Synoptic

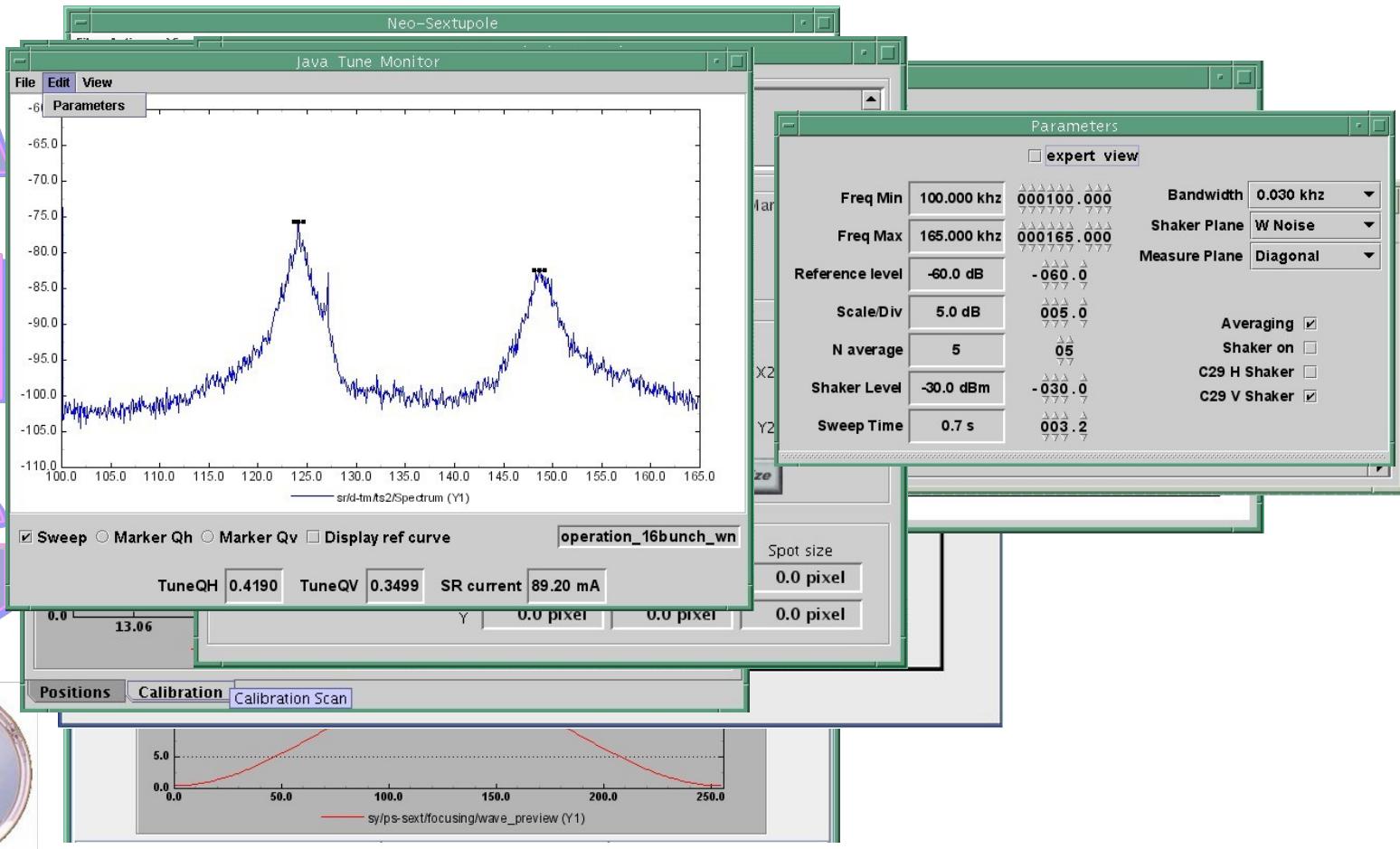
Final synoptic application



T  
A  
N  
G  
C  
S

# Examples

More information : ➤<http://www.tango-controls.org>  
➤<http://www.tango-controls.org/tutorials>



# T A N G



## Tango training: Part 11 : Miscellaneous

- HDB
- Astor and Starter
- Getting software
- Who is doing what





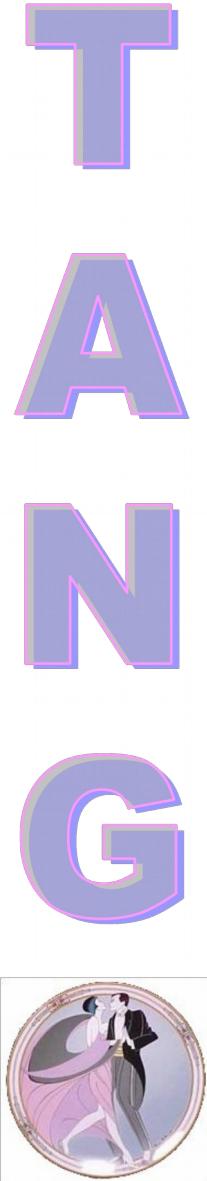
# History DataBase (HDB)

- A set of three databases to keep history of what's going on in the control system
  - HDB (History DataBase)
  - TDB (Temporary DataBase)
  - Snap (Snapshot database)
- Two supported underlying RDBMS
  - Oracle (Soleil)
  - MySQL (Alba, Elettra, ESRF)
- Running 7 days a week, 24 hours a day



## HDB

- Storage of data coming from attributes only
  - Command are not supported
- HDB is dedicated to long term storage
  - Data are never deleted
  - Smallest storage period = 10 sec (0.1 Hz)
- TDB is dedicated to temporary storage
  - 3 days max
  - Smallest storage period = 0.1 sec (10 Hz)



# HDB

- Snap is a database to take control system snapshot which can be re-applied later on
- 3 graphical applications
  - Mambo
    - Configure HDB and TDB
    - Display data stored in HDB / TDB
  - Bensikin
    - Configure Snapshot
    - Take and restore a snapshot
  - E-Giga
    - Display data coming in HDB in your WEB browser



# HDB

- Implemented using
  - Set of Java device servers to
    - Get data from the control system
    - Send extracted data to caller
  - JDBC to access the database itself

# HDB

- Device servers common for HDB / TDB
  - ArchivingManager
    - Provide global command(s)
    - Load balancing
- Device servers for HDB
  - HdbArchiver(s)
    - Get data from control system and send them to RDBMS
      - Today, use polling. Soon updated to also support archiving event
  - HdbExtractor(s)
    - Extract data from the RDBMS and send them to caller
  - HdbArchivingWatcher
    - Global Hdb archiving status and reports



# HDB

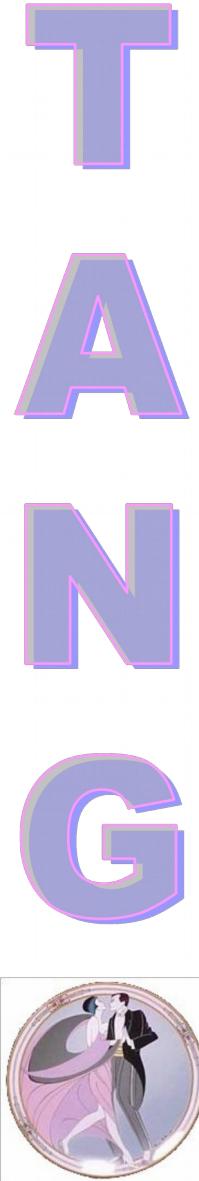
## ■ Device servers for TDB

- TdbArchiver(s)
  - Get data from control system and send them to RDBMS
    - Today, use polling. Soon updated to also support archiving event
- TdbExtractor(s)
  - Extract data from the RDBMS and send them to caller
- TdbArchivingWatcher
  - Global Tdb archiving status and reports



# HDB

- Device servers for Snap
  - SnapManager
    - Manage snapshot configuration
    - Send command(s) to SnapArchiver
  - SnapArchiver
    - To take the snapshot and send the data to the RDBMS
  - SnapExtractor
    - To extract snapshot data from database



# HDB

- HDB / TDB: Several storage modes:
  - Periodic: Data stored at a fixed period (mandatory)
  - Different:
    - Data stored when reading is different than the last stored value
    - Data stored when the difference between read value and last stored value greater/lower than value in absolute
    - Data stored when the difference between read value and last stored value greater/lower than value in %
  - Threshold: Data stored greater/lower than a pre-defined threshold



# Tango CS Administration

- A couple to administrate a Tango control system
  - Astor - Java graphical application
  - The Starter device server
- The aim is to
  - Rapidly check if your control system is OK
  - Start / Stop a device servers without logging on the host where the device server is running



# Tango CS Administration

- The Starter is able to
  - Start device server(s)
    - Manage 5 (default) startup levels
  - Get which device server runs on which host from the database
  - Kill a device server (command “kill” of the admin device)
  - Check that a device server is running.
  - ping the device server process admin device to check if it is alive
  - Start even before the database is running and wait for it
  - Check if the notifd is running



# Tango CS Administration

- Run one Starter device server per host in the control system
- Start the Starter device server using the host name as instance name

Starter <host>

- The starter device name is (only one device)

tango/admin/<host>

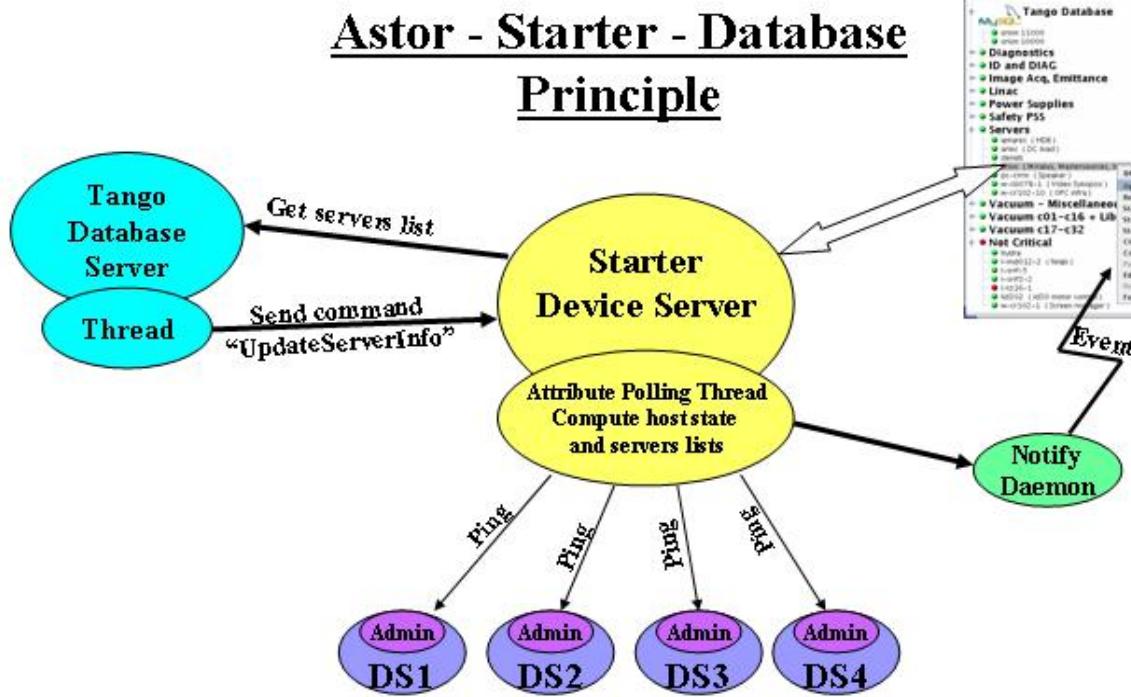


# Tango CS Administration

- Astor is a graphical interface to the starter device(s) and is able to
  - Manage host(s) in a tree structure
  - Display the state of nodes and servers
  - Start / Stop several servers on several host(s) with some clicks
  - See the device server output
  - Open a window on a host
  - Help you creating a new Starter entry
  - Start jive
  - Configure device polling and events
  - Test events

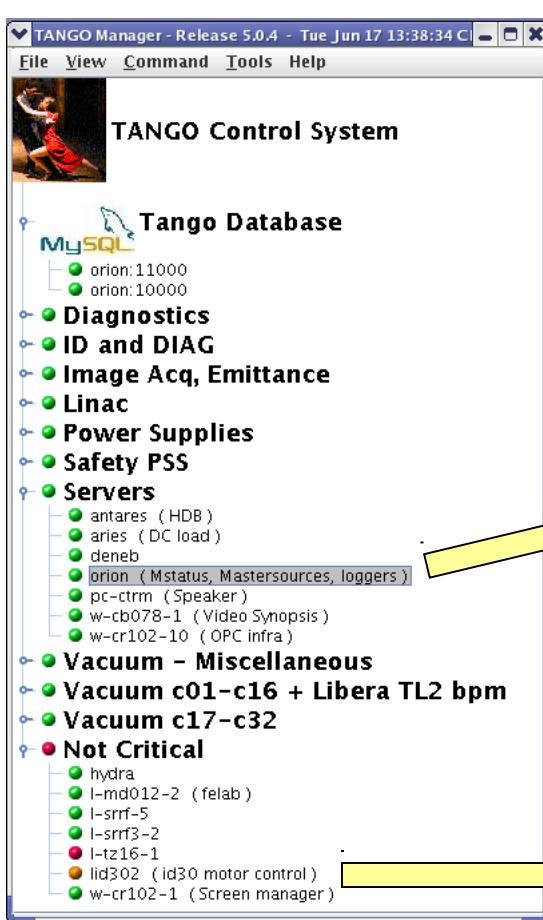
T  
A  
N  
G  


# Tango CS Administration





# Tango CS Administration



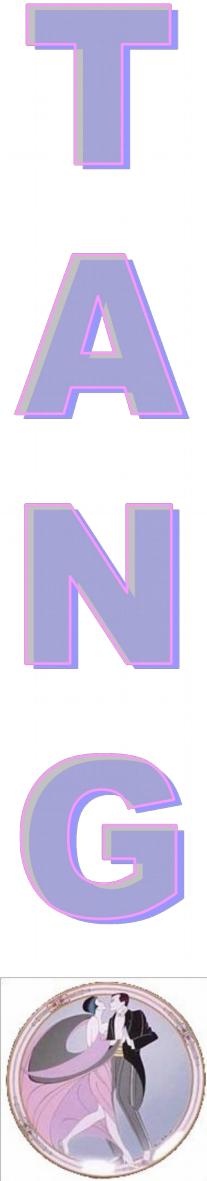
Hosts	Servers
<ul style="list-style-type: none"><li>• All controlled servers are running.</li><li>• Starter is starting server(s).</li><li>• At least, one controlled server is stopped.</li><li>• Starter is not running on host.</li></ul>	<ul style="list-style-type: none"><li>• Server is running</li><li>• Server is running but not alive (Starting ?)</li><li>• Server is not running.</li></ul>





# Tango CS Administration

- **Host (Starter) actions:**
  - Open a control panel (see servers)
  - Remote login (not for win32)
  - Starter test
  - Clone (create a new Starter in database)
  - Cut / Paste (to manage tree)
  - Edit properties (Starter \$PATH, comments...)
  - Remove



# Tango CS Administration

## ■ Servers actions:

- Start / kill server
- Restart (kill wait a bit and start)
- Set startup level
- Polling management (set period, little profiler)
- Configuration (using Jive wizard)
- Server and class info
- Test a device
- Check states
- See standard error.



# Getting the Tango Core

- You can download Tango from the ESRF Tango WEB page (<http://www.tango-controls.org/download>)
  - As a source package for UNIX like OS
  - As a Windows binary distribution
- For Unix (and co), do not forget to first download, compile and install
  - omniORB
  - omniNotify
- For Windows all libraries and binaries for omniORB and omniNotify are included in the distribution.



# Getting the Tango Core

- In both distributions, you have
  - Tango core (libs and jar file)
  - Database device server and a script to create the Tango database for MySQL
  - Pogo, Jive, LogViewer
  - Astor and Starter device server
  - A test device server (TangoTest)
  - ATK



# Getting the Tango Core

- For the UNIX like OS source distribution, you have to compile everything with the famous three commands
  - configure
  - make
  - make install



# Tango Core Sources

- All Tango core sources are stored in a CVS server hosted by SourceForge called Tango-cs (<http://sourceforge.net/projects/tango-cs/>)
- On this project, you find sources for
  - C++ library and Java API
  - Database, Starter and TangoTest device server
  - Pogo, Astor, Jive, LogViewer and ATK
  - Binding for Python, Matlab and Igor
  - The Tango HDB system



# Getting Tango Classes

- Nearly all Tango classes (> 200) are available for download on the WEB from Tango related WEB sites
- Two kind of classes
  - Common interest and interfaces to commercial hardware
  - Specific classes to interface institute specific hardware



# Getting Tango Classes

- On the WEB for each class, you find the HTML pages generated by Pogo
- Common interest classes sources are stored in a CVS server hosted by SourceForge
  - Project name = tango-ds
  - <http://sourceforge.net/projects/tango-ds/>
- Local classes sources are stored in a local CVS repository at each institute



# Getting Tango Classes

Welcome to the TANGO website – the TANGO website - Mozilla Firefox

TANGO home page Synopsis Congrès ESRF/Linux Calepin Montagne

about news events documents tutorials download log in

you are here: home

Tango Device Server User's Guide - Mozilla Firefox

304 Tango classes

Search key Word Alphabetical Order

SourceForge(Tango-AbstractClasses) Archiving Miscellaneous

Acquire\_Waveform\_Lecroy\_Pj User's Guide

AcquireWaveformLecroy Class

Revision: release\_1\_0\_0 - Author: xavela  
Implemented in C++ - CVS repository: tango-ds (Instrumentation)

[Download PDF format](#)

Introduction:

This class allows the acquisition of a waveform (the description and the data), from a specific channel and from any Lecroy scope series. The description is a structure which contains in particular : the length of the array 1 (the raw data) the length of the array 2 (2nd part of the calculated waveform data, if any) the timebase the total length of the acquired data (lgh array1 + lgh array2) ... it represents the context of the acquisition. NOTE : for all Lecroy scope, the port number is the same : 1861

Description:

- Device description
- Properties description
- States description
- Commands description
- Attributes description

ESRF:

- Baselines
- Input/Output
- Linc
- Measures
- Motion
- Power Supply
- Hans
- Protocols
- Radio Frequency
- TA00 Interfaces
- Vacuum
- Miscellaneous

Done

Tango Device Server User's Guide - Mozilla Firefox

304 Tango classes

Search key Word Alphabetical Order

SourceForge(Tango-AbstractClasses) Archiving Miscellaneous

Acquire\_Waveform\_Lecroy\_Pj Device Attributes Description

AcquireWaveformLecroy Class

Revision: release\_1\_0\_0 - Author: xavela

Scalar Attributes				
Attribute name	Data Type	R/W Type	Expert	
waveArray1	Length of the first array. This second array contains data of simple waveforms.	DEV_LONG	READ	Yes
waveArray2	Length of the second array. This second array contains data of complex waveforms (as FFT, Extrema ....).	DEV_LONG	READ	Yes
waveArrayCount	Length of the two arrays. If the length of wavearray1 is different of this length that's means data are present in the array wavearray2. Which is used for complex data as math operations on a waveform.	DEV_LONG	READ	No
nominalBits	ADC resolution. For simple data this ADC is an 8 bits resolution else it's can be 10 up to 12 bits	DEV_SHORT	READ	Yes
horizontalInterval	Sampling interval for time domain waveforms. Needed to scale the waveform data stored in the rawWaveformData attribute.	DEV_DOUBLE	READ	Yes
horizontalOffset	Trigger offset for the first sweep of the trigger. seconds between the trigger and the first data point. Needed to scale the waveform data stored in the rawWaveformData attribute.	DEV_DOUBLE	READ	Yes
verticalGain	The vertical gain. Used to scale the waveform data stored in the rawWaveformData attribute.	DEV_DOUBLE	READ	Yes

Done

# Tango @ Elettra

- Write Tango classes
- Development of
  - An alarm system
  - Canone: A WEB interface using PHP
  - E-Giga: A WEB interface above the Tango HDB
  - QTango: ATK like using C++, Python and Qt



# Tango @ Soleil

- Writte many Tango classes
- Development of
  - Archiving database (Tango HDB) using
    - ORACLE or MySQL
  - Snapshot system also using
    - ORACLE or MySQL
  - WEB protocol for ATK



# Tango @ Alba

- Write Tango classes as well
- Development of
  - Python binding (PyTango release 3.x)
  - Sardana: Control software for experiments
  - Tao: Python graphical layer above Tango (ATK like)



# Tango @ ESRF

- Write Tango classes
- Development of
  - Tango core (C++ and Java)
  - Pogo
  - Jive
  - Astor / Starter
  - ATK

