

SOLEIL proposals for a CLI (*Command Line Interface*)

- A command-line interface (CLI) is a mechanism for **interacting** with a computer operating system or software **by typing commands** to perform specific tasks.
- This **text-only interface contrasts** with the **use of a mouse** pointer with a graphical user interface (GUI) to click on options, or menus.
- **This method** of instructing a computer to perform a given task **is referred to as "entering" a command**: the **system waits for the user** to conclude the submitting of the text command by **pressing the "Enter" key**
- A **command-line interpreter** then receives, analyses, and executes the requested command.
 - ✓ The command-line interpreter may be run in a text terminal or in a terminal emulator window.
- Upon completion, the command usually returns **output** to the user in the **form of text lines on the CLI**.
 - ✓ This output may be an answer if the command was a question, or otherwise a summary of the operation.

- The concept of the CLI originated when teletypewriter machines (TTY) were connected to computers in the 1950s, and offered results on demand, compared to batch oriented mechanical punched card input technology.
- Dedicated text-based CRT terminals followed, with faster interaction and more information visible at one time
- Then graphical terminals enriched the visual display of information.



1950s



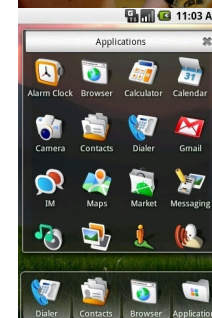
1980s



1990s



Today



- The PyTango binding with the so-called « simplified syntax layer » allows typing commands like :

```
➤ theta = DeviceProxy  
(« D13/EX/THETA »)  
➤ theta.position = 10  
➤ theta.position += 10  
➤ theta.stop()
```

This gives access to the whole Tango Control system in a generic way
(i.e which do not depend on SOLEIL specific devices)

- The CLI access to the scan service is done through a specific device
 - ✓ the SimpleScan one

```
➤ scan = DeviceProxy («D13/CA/SIMPLESCAN»)  
➤ scan.configuration = theta_2theta_scan  
➤ scan.UploadConfiguration()  
➤ Scan.start()
```

- Nothing specific is added to the PyTango binding !!
 - It is currently deployed and works on SOLEIL beamlines

- When writing attributes, the command line interpreter is **not blocking**
 - ✓ This may be considered either as :
 - ❑ an advantage : *I can move simultaneously many motors*
 - ❑ Or a disadvantage : *I would prefer to do one thing at a time*
- Syntax could be even **simpler**
- Some useful stuff could be added to make thing simpler :
 - ✓ **command completion** for instance
 - ✓ **Loading** some Tango database definition to **avoid initial CLI** painful configuration like

```
➤ From PyTango import *  
➤ scan = DeviceProxy  
  («D13/CA/SIMPLESCAN»)  
➤ theta = DeviceProxy (« D13/EX/THETA »)
```

- Flow control operators (if , then , else, while) ARE NOT PART of a CLI

- For example :
 - ✓ DOS do not include (real) flow control operator
 - ✓ BASIC was invented for this purpose

- At SOLEIL people who need to add flow control have 2 ways of doing so :
 - ✓ The Passerelle way (which does not require any programming skills)
 - ✓ The python way, for python programmers

These 2 solutions are already working and used on SOLEIL beamlines

- Putting non generic stuff within the Python binding is not maintainable on a large scale basis
 - ✓ When a new device class is added in the Control System, not added customisation must be done at the CLI level
 - ✓ Otherwise, we will end up mirroring all the control system in the CLI

```
V:\toto.py - Notepad++
Fichier Edition Recherche Affichage Encodage Langage Paramétrage Macro Exécution TextFX Compléments Documents ?
ScreenShotFactory.java toto.py
3 Device.__init__(self, name, True, PyTango.DevState.OFF)
4
5 def __call__(self, dt):
6     self.integrationTime = dt
7     self.start()
8
9 class Lakeshore_340(Device):
10     def __init__(self, name):
11         Device.__init__(self, name, True, PyTango.DevState.STANDBY)
12
13 class Attenuator(Device):
14     def __init__(self, name):
15         Device.__init__(self, name, False, PyTango.DevState.ON)
16
17 class NI6602(Device):
18     def __init__(self, name):
19         Device.__init__(self, name, True)
20
21     def __call__(self, value, counter = None):
22         continuous = self.continuous
23         it = self.integrationTime
24         self.continuous = 0
25         self.integrationTime = value
26         self.start()
27         while self.state() != PyTango.DevState.STANDBY:
28             pass
29         if counter:
30             print getattr(self, "counter%d" % counter)
31         self.continuous = continuous
32         self.integrationTime = it
33         if continuous:
34             self.Start()
35
36 class Motor(Device):
37     def __init__(self, name):
38         Device.__init__(self, name, True, PyTango.DevState.STANDBY)
39
40     def __setattr__(self, name, value):
41         if name.lower() == "position":
42             try:
43                 Device.__setattr__(self, name, value)
44             except KeyboardInterrupt:
45                 self.Stop()
46                 print "Current position %f" % self.position
47         else:
48             Device.__setattr__(self, name, value)
49
Python file 1205 chars 1253 bytes 49 lines Ln : 20 Col : 1 Sel : 0 (0 bytes) in 0 ranges UNIX ANSI INS
```

•Encapsulation violation !!
•What appends if Tango device interface changes ?

- When you have 24 beamlines, are these local violations everywhere the same ?
- How do you manage large scale software updates
 - ✓ It implies that at the same time , you must change DeviceServers AND all client applications
 - ✓ Is it the way to go ?

- We would be happy to have Pytango enhancements to have a better CLI than today :
 - ✓ *Blocking mechanism for attributes*
 - ✓ *Configuration simplification*
 - ✓ *Command completion*
 - ✓
- The Python language is the right place :
 - ✓ to implement complex workflows for experimented programmers
- Passerelle graphical environment is the right place to :
 - ✓ implement complex workflows for non- programmers

Annex

- Batch processing is execution of a series of programs ("jobs") on a computer without manual intervention.
- Batch jobs are set up so they can be run to completion without manual intervention, so all input data is preselected through scripts or command-line parameters. This is in contrast to "online" or interactive programs which prompt the user for such input. A program takes a set of data files as input, process the data, and produces a set of output data files. This operating environment is termed as "batch processing" because the input data are collected into batches on files and are processed in batches by the program.