AALib::Framework
__concepts__

Asynchronous Action Library
AALib – PyAALib – JyAALib
Tutorial and Techniques

by
R. A. Pieritz

AALib

Asynchronous

Asynchrony, in the general meaning, is the state of not being synchronized.

Action

Action is something a person can do...

Library

A library is a collection of information, sources, resources, and services: it is organized to be used for something.

Framework

Wikipedia Definition

http://en.wikipedia.org/wiki/Framework

"In software development, a framework is a defined support structure in which another software project can be organized and developed. A framework may include support programs, code libraries, a scripting language, or other software to help develop and glue together the different components of a software project."

Free Dictionary Definition

http://computingdictionary.thefreedictionary.com/application%20framework

"- (1) The building blocks of an application.

- (2) A set of software routines that provide a foundation structure for an application. Frameworks take the tedium out of writing an application from scratch. Object-oriented application frameworks, which are the norm today, are structured as a class library.

- Each class library has its way of doing things, and although the purpose of a framework is to eliminate a certain amount of programming drudgery, programmers must first learn the structure and peculiarities of the framework in order to use it."

AALib

What is it?

The project is a Framework based in Pure Object Oriented Concepts to develop asynchronous communication between threads and external process

An application can be entirely designed and developed based on dynamic multithread PLUGINS.

Why?

"__Code once and run anywhere__"

"We want to code easily asynchronous actions to do a lot of things together and independently"

Where?

Only a single code for all platforms: Windows, MacOSX, Linux and Solaris

How?

The actual version was developed using Python in the basic layer - PyAALib - and it is designed to be natively translated to Java using Jython – JyAALib.

AALib
__main::concept__ = Think OBJECT !!

Everything is an object to be managed by objects.

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

AALib Framework Main Features

- Independent of the System architecture and OS;
- Independent of the Programming Language;
- Independent of the Programming IDE;
- Full OOP design – Object Oriented Programming;
- One code running in all platforms with the same features and actions;
- Asynchronous Multithread Actions + Thread Safety + Grid Computing;
- Advanced Resources and Generic Application Design (i.e.: RAID);
- Implements the standard internet services and protocols;
- Use only Objects with simple and standard definitions;
- Each Object is independent and self-contained (no hierarchy to generate and manage entities);
- The data persistence is assured by each individual object and is preserved when it is shared with other entities or stored (duplication or other operation);
- Complete and independent Unittest framework;
- Multithread exception handling and log;
- Automatic generation of code documentation in html, PDF, and man;
- Incorporate Debug and Exception control Model;
- Well defined code syntax and rules for the code design;
- Simple code maintenance (everything is an independent object defined by a class, all code is identical in all operational system);
- Code reliability (i.e.: runtime flow control robustness);
- No necessary external licenses or agreements;

AALib
Basic Design Patterns

- **MVC**

  Model-View-Controller

- **Object Factory**

  Plugin Model

- **Callback**

  Generic code connection and execution

AALib Framework

Implementation

- Python

  PyAALib 1.0

  2005-2007

  http://pyaalib.sourceforge.net

- JAVA-Jython-Python

  PyAALib-JyAALib 2.0

  - initiated on July 2007
  - full functional – testing and adding new features
  - to be released on 2008

  http://jyaalib.sourceforge.net

TODAY

PyAALib – JyAALib

version 2.0

Why?

It allows mix Java code and the AALib framework with the simplicity of Python and the powerful deployment architecture of Java.

How?

You can use java inside the PyAALib plugins in a pure java system.

Where?

The same code source runs on Python or Java/Jython. It can be deployed in Java environments without Python installed.

Full Compatible?

- Python 2.2 and later
- Jython 2.2
- JAVA 6.0

PyAALib-JyAALib Overview

It is a well tested open source framework to be used on RAID development of asynchronous multi-thread applications.

It implements the basic plugin architecture to be used with the special PyAALib action objects and standard Python and JAVA classes. This mechanism allows the rapid development of dynamic and modular applications to facilitate the continuous improvement.

Modern software engineering techniques based on unittest are used to guarantee the quality of each part of the code during the development cycle.

The library is full compatible with all standard frameworks using C, C++, Fortran, Java, Jython and Python.

Projects Using The Framework?

[DRank]

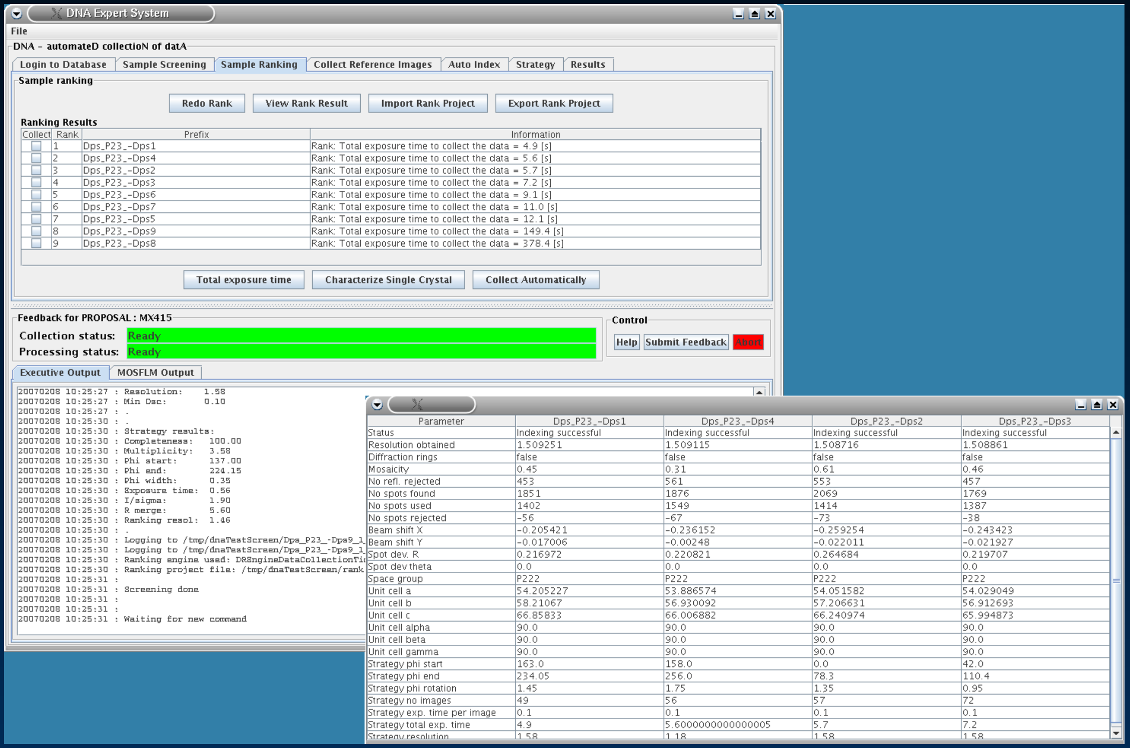
Crystal Data Ranking Module in DNA Package.

The DNA project - http://www.dna.ac.uk - BioXHIT http://www.bioxhit.org

TODAY

PyAALib – JyAALib

version 0.9



Projects Using The Framework?

[DNABenchmark]

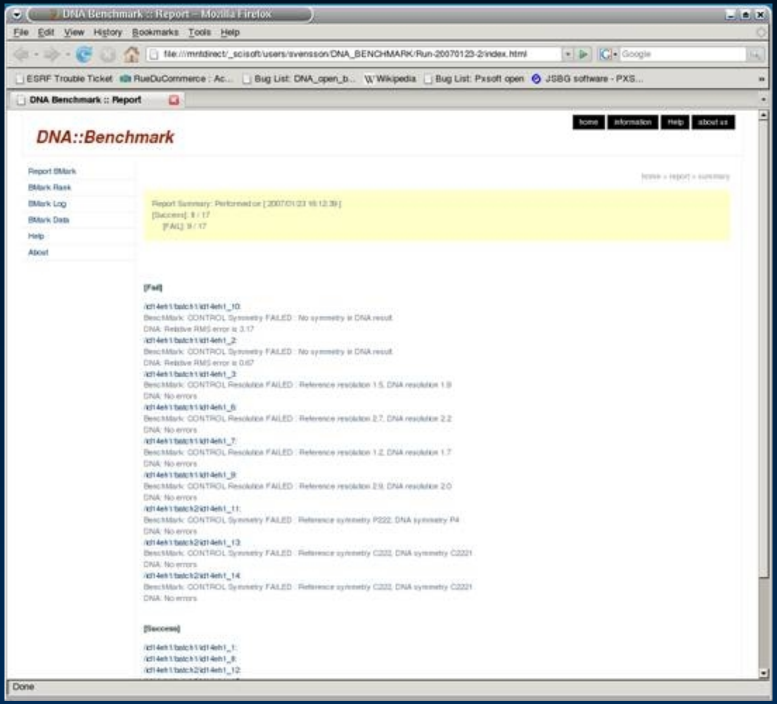
Test benchmark manager to evaluate the scientific results.

The DNA - http://www.dna.ac.uk - project is a collaboration initially between the ESRF, the CCLRC Daresbury Laboratory and MRC-LMB in Cambridge, with the aim of completely automating the collection and processing of X-Ray protein crystallography data - Founded by European Community Project: BioXHIT http://www.bioxhit.org

TODAY

PyAALib – JyAALib

version 1.0



TODAY

PyAALib – JyAALib

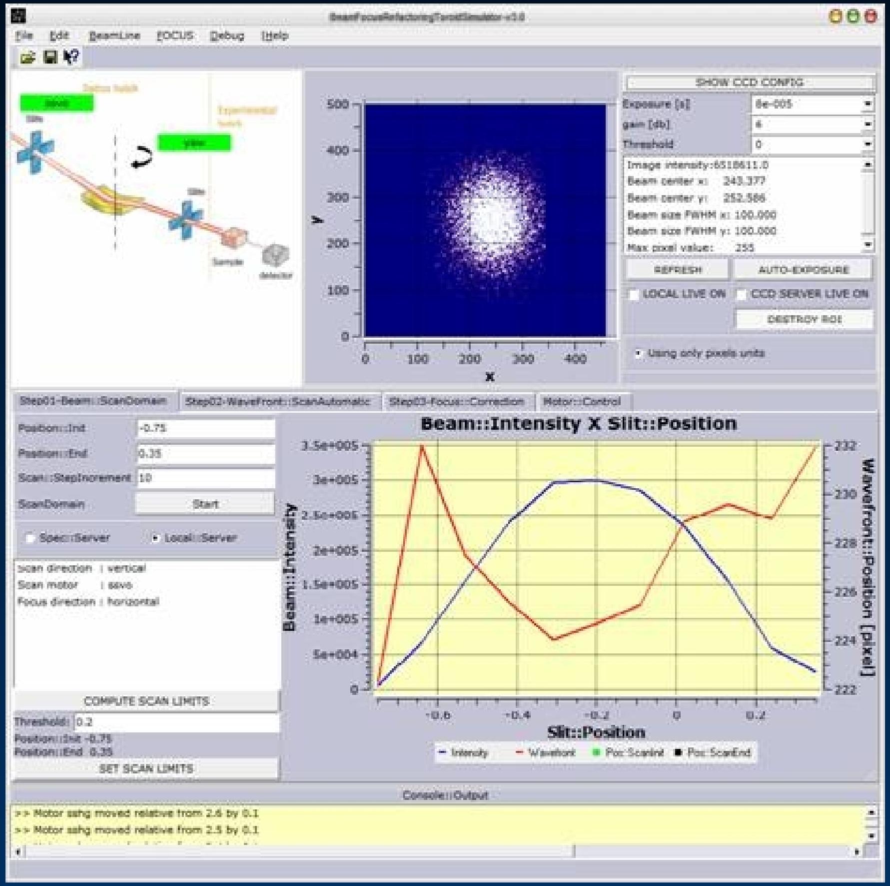
version 1.0

Projects Using The Framework?

[BeamFocus]

BeamLine Assisted Focus Application.

Software used to assist the beamline operator to optimally focus X-Rays beams at ESRF European Synchrotron Radiation Facility – SciSoft Group http://beamfocus.sourceforge.net



TODAY

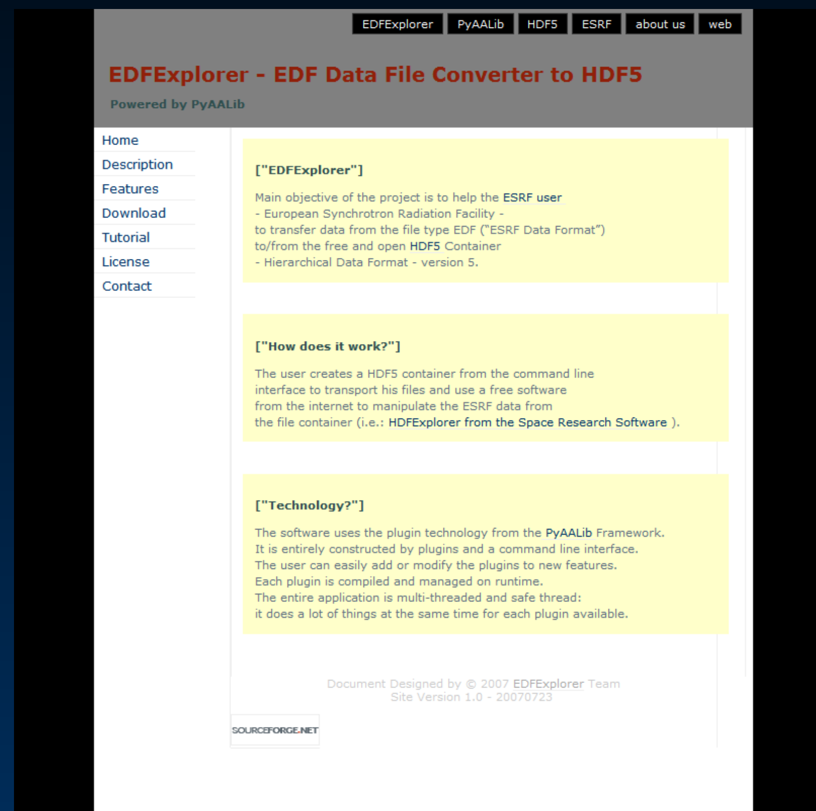
PyAALib – JyAALib

version 2.0

Projects Using The Framework?

[EDFExplorer]

Software used to assist the beamline user to transfer his data in EDF format to an HDF5 container - ESRF European Synchrotron Radiation Facility – SciSoft Group - http://edfexplorer.sourceforge.net



Overview Resume

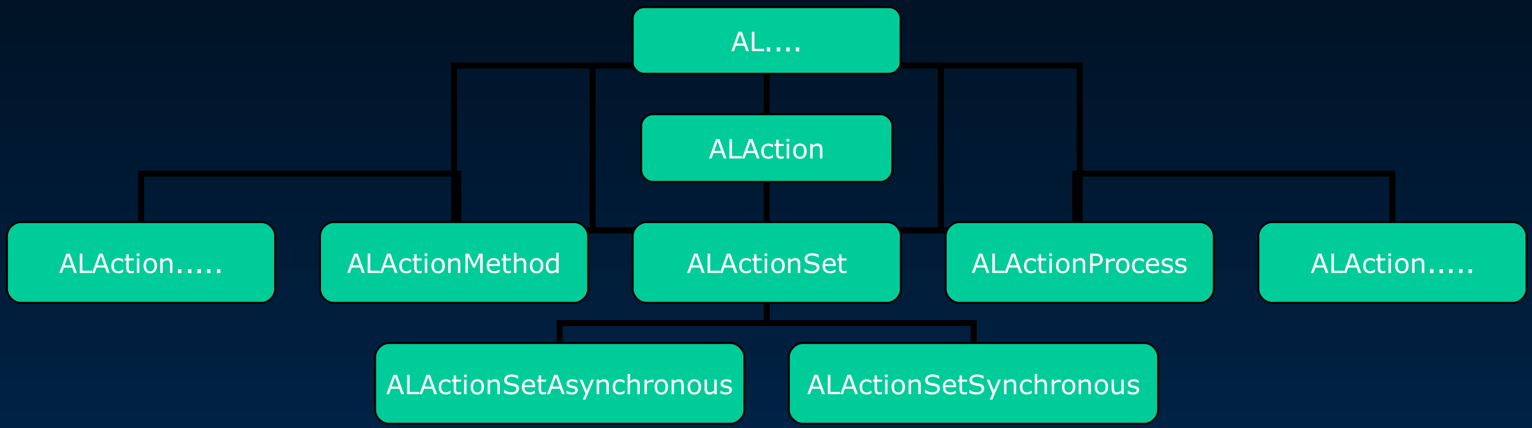
- AALib is an OOP Framework
- It is composed by classes
- It is free and Open Source
- It has a built in multithread plugin architecture
- PyAALib runs on Python
- PyAALib- JyAALib runs on Jython-JAVA virtual machine platform

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

AALib Basis

- Everything is an Object defined by a Class
- Each class is defined in a single file
- The Class use the "ClassNameSpace": "AL"+Something
- The name order of the class reflects the inheritance – "it is not the English language", i.e.:



- Almost the main classes and their Childs are "Thread Safety" – they can be used in simultaneous events – i.e.: ALVerbose, ALAction, ALLogFile, etc....
- The "Thread Safety" mechanism is based on POSIX semaphores to be used on Massive parallel tasks.

AALib Basis

You can use an AALib class in any kind of code or library

```
from ALVerbose  import  ALVerbose

ALVerbose.screen( "Hello world" )
```

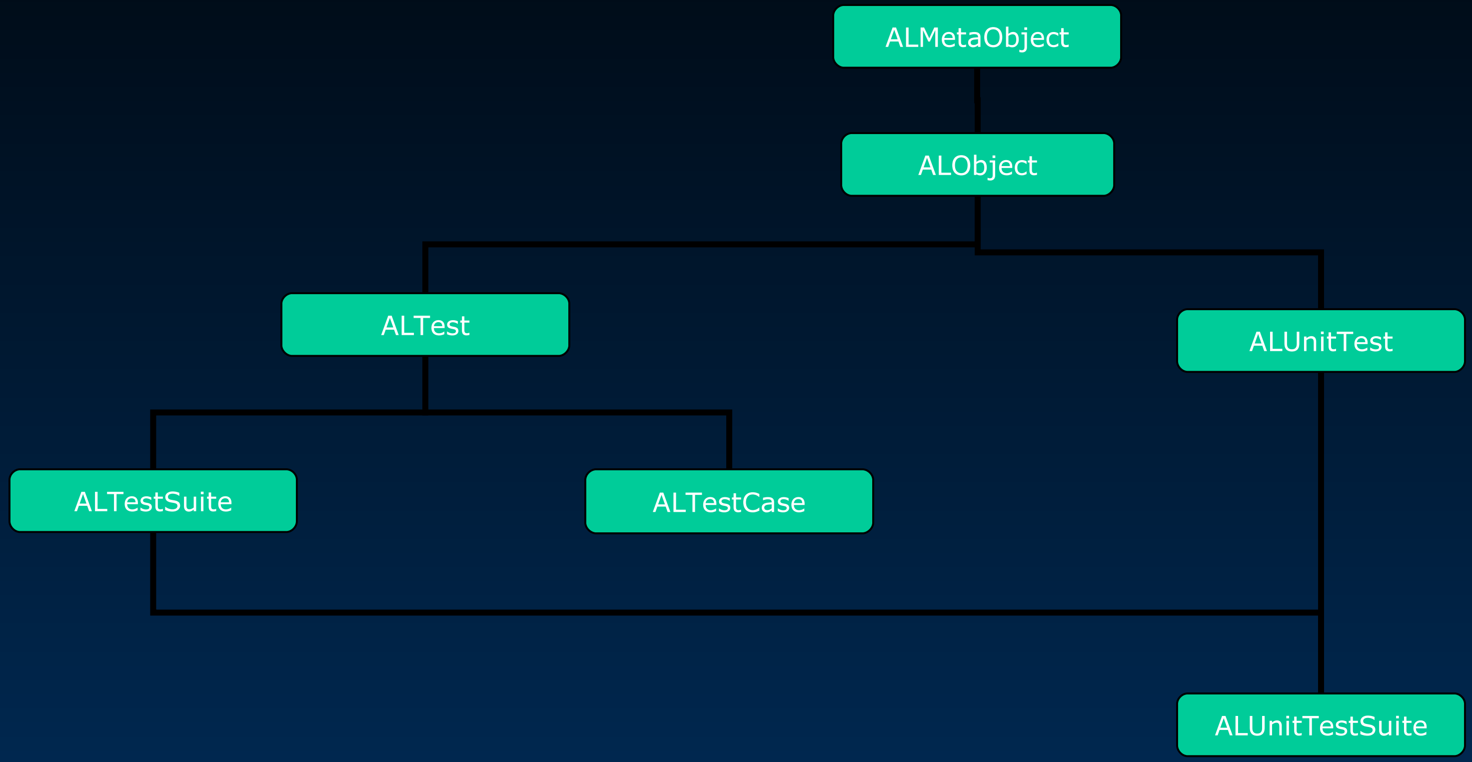
```
from ALVerbose    import  ALVerbose
from ALString     import  ALString

oalString = ALString( "Hello world" )
ALVerbose.screen( oalString )
```

We have about 140 different classes today (october/2008)

We have at least 1000 methods to test these classes (and increasing)

AALib Basis

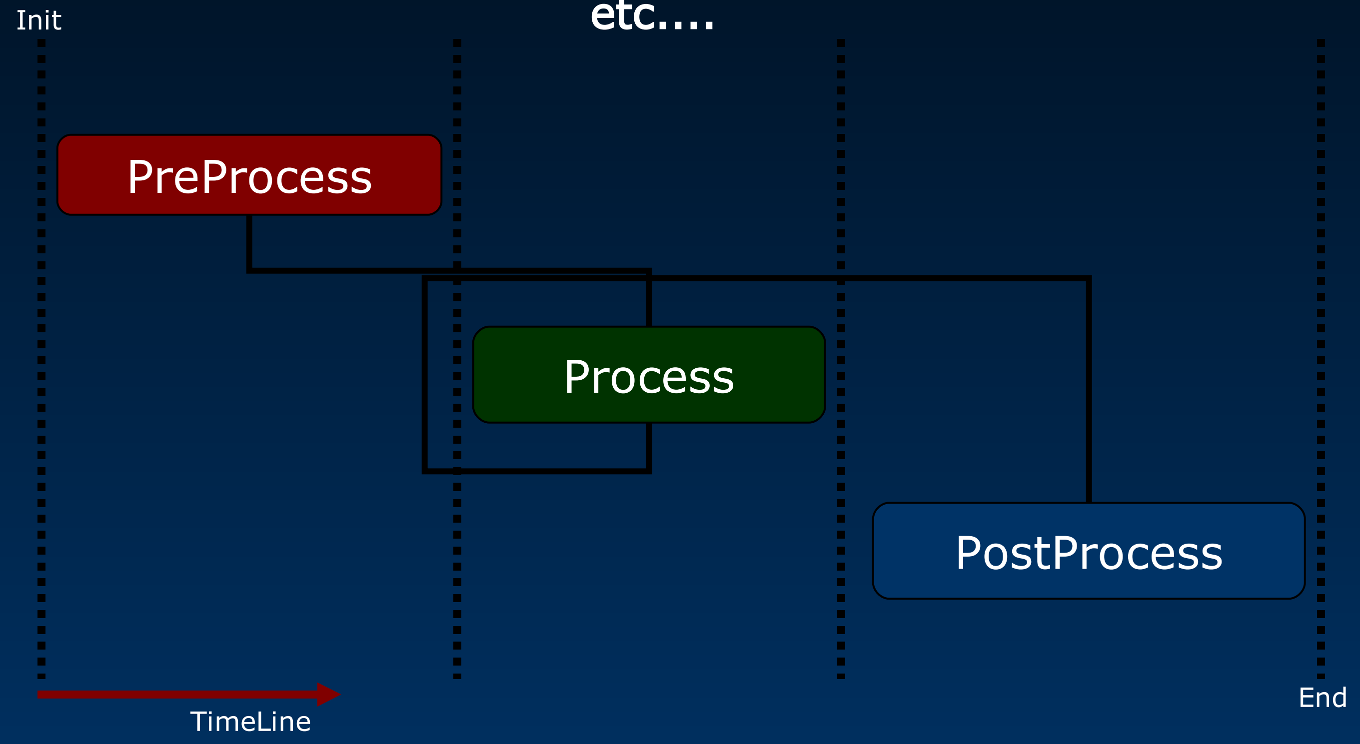
We use the UnitTest concept to test all methods and classes of the Library



AALib Basis

- All complex events (i.e.: Actions, ALApplication, etc) are modelled by a simple standard sequential STEP cycle "engine"

- These "engines" are encapsulated on threads and can be used in a massive multithread application – i.e.: ALAction, ALApplication, ALComandLine, ALTest, etc....



AALib Basis

[Basic Application Framework skeleton]

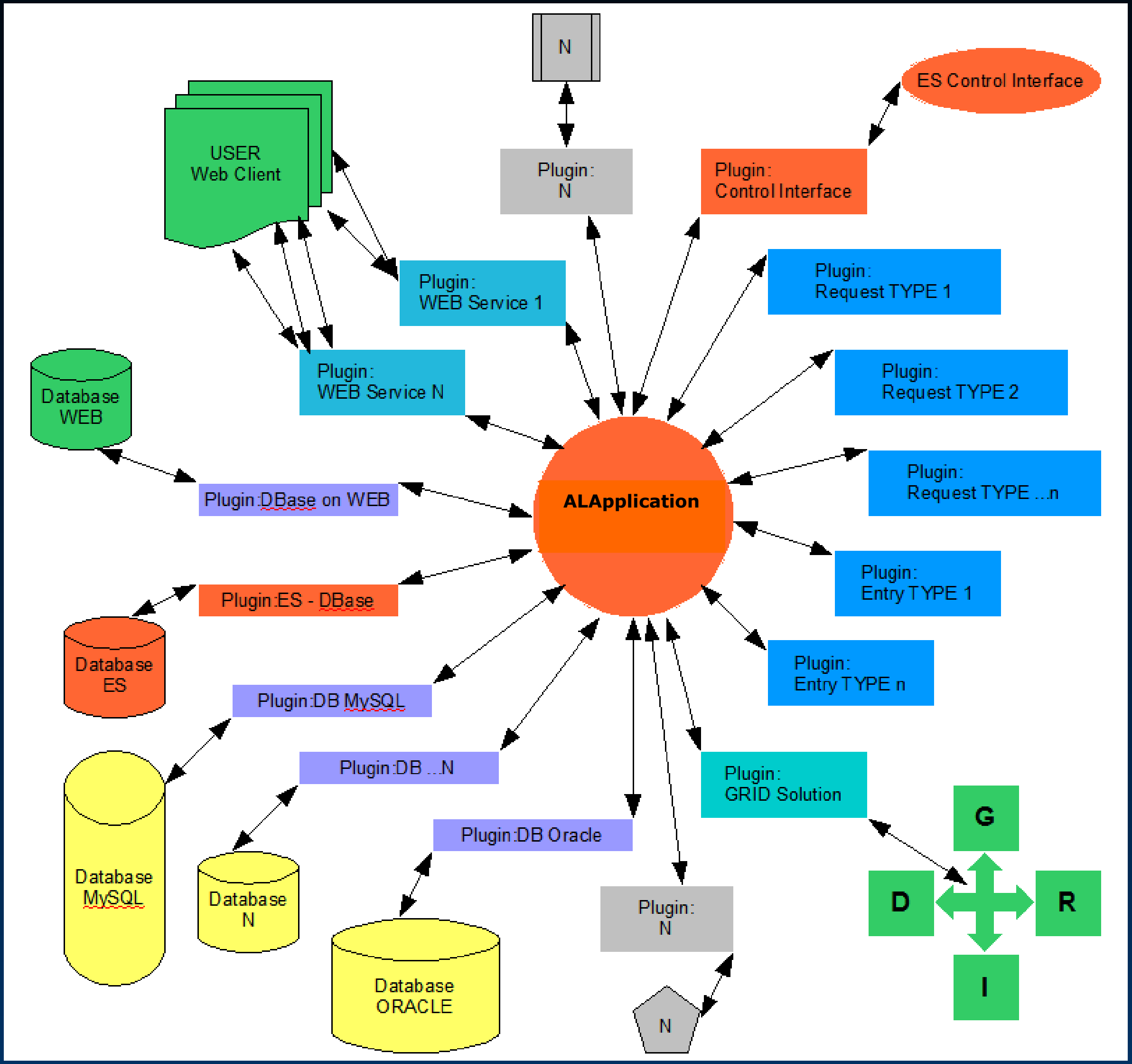
“A single code line to define and run a complete asynchronous multithread application based on a dynamic plugin architecture”

```
from ALImportKernel        import  ALApplication

if __name__ == '__main__':

        ALApplication( "DemoApplication", “2.0” ).execute()
```

- Dynamic Plugin Architecture

The plugin architecture add new functionalities to the ALApplication.

AALib
Basis - Plugin



AALib Basis Resume

- AALib has “out of the box” classes for modern software
- The basic engine model is simple and flexible:

  PreProcess – Process- PostProcess
- The main use case is “one line of code”
- The basic use case is a complete modern application model
- It has resources, multithread log manager, built-in basic controls, etc
- The plugin model is built-in in the main use case

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Main Classes

```
>PyAALib-JyAALib-HEAD-v2.0  [lapquark.esrf.fr]
  >.externalToolBuilders
  .settings
  >libraries
  modules
    ALAction.py 1.5 (ASCII -kkv)
    ALActionMethod.py 1.1 (ASCII -kkv)
    ALActionMethodTimer.py 1.1 (ASCII -kkv)
    ALActionProcess.py 1.4 (ASCII -kkv)
    ALActionSet.py 1.1 (ASCII -kkv)
    ALActionSetAsynchronous.py 1.1 (ASCII -kkv)
    ALActionSetSynchronous.py 1.1 (ASCII -kkv)
    ALAppDefinition.py 1.3 (ASCII -kkv)
    ALAppDefinitionCommandLine.py 1.3 (ASCII -kkv)
    ALAppDefinitionControl.py 1.2 (ASCII -kkv)
    ALAppDefinitionCore.py 1.4 (ASCII -kkv)
    ALAppDefinitionEnd.py 1.5 (ASCII -kkv)
    ALAppDefinitionInit.py 1.2 (ASCII -kkv)
    ALAppDefinitionOutput.py 1.1 (ASCII -kkv)
    ALApplication.py 1.1 (ASCII -kkv)
    ALApplicationActionControl.py 1.2 (ASCII -kkv)
    ALApplicationCommunicator.py 1.2 (ASCII -kkv)
    ALApplicationServerXmlRpc.py 1.1 (ASCII -kkv)
    ALArgument.py 1.1 (ASCII -kkv)
    ALAssert.py 1.2 (ASCII -kkv)
    ALClient.py 1.1 (ASCII -kkv)
    ALClientHttp.py 1.1 (ASCII -kkv)
    ALCom.py 1.1 (ASCII -kkv)
    ALComClient.py 1.1 (ASCII -kkv)
    ALComClientHttp.py 1.1 (ASCII -kkv)
    ALCommand.py 1.1 (ASCII -kkv)
    ALCommandLine.py 1.4 (ASCII -kkv)
    ALCommandLineArgument.py 1.1 (ASCII -kkv)
    ALCommandLineInterface.py 1.2 (ASCII -kkv)
    ALCommandLineInterfaceDefinition.py 1.1 (ASCII -kkv)
    ALCommandLineInterfaceDefinition.xsd 1.1 (ASCII -kkv)
    ALCommandLineInterfaceElement.py 1.1 (ASCII -kkv)
    ALCommandLineInterfaceElement.xsd 1.1 (ASCII -kkv)
    ALCommunicator.py 1.1 (ASCII -kkv)
    ALCompiler.py 1.1 (ASCII -kkv)
    ALComServer.py 1.1 (ASCII -kkv)
    ALComServerHttp.py 1.1 (ASCII -kkv)
    ALDefinition.py 1.3 (ASCII -kkv)
    ALDict.py 1.1 (ASCII -kkv)
    ALDictionary.py 1.4 (ASCII -kkv)
    ALDiskExplorer.py 1.3 (ASCII -kkv)
    ALEdf.py 1.4 (ASCII -kkv)
    ALException.py 1.1 (ASCII -kkv)
    ALExceptionControl.py 1.1 (ASCII -kkv)
    ALExceptionSignal.py 1.1 (ASCII -kkv)
    ALExternalPackEdfFile.py 1.1 (ASCII -kkv)
    ALExternalPackGenerateDS.py 1.1 (ASCII -kkv)
    ALFactory.py 1.1 (ASCII -kkv)
    ALFactoryPlugin.py 1.1 (ASCII -kkv)
    ALFactoryPluginStatic.py 1.1 (ASCII -kkv)
    ALFile.py 1.3 (ASCII -kkv)
    ALHandler.py 1.1 (ASCII -kkv)
    ALHandlerRequest.py 1.1 (ASCII -kkv)
```

```
    ALHandler.py 1.1 (ASCII -kkv)
    ALHandlerRequest.py 1.1 (ASCII -kkv)
    ALHandlerRequestHttp.py 1.1 (ASCII -kkv)
    ALHdf5.py 1.1 (ASCII -kkv)
    ALHdf5Edf.py 1.2 (ASCII -kkv)
    ALHistogram.py 1.1 (ASCII -kkv)
    ALHistogramImage.py 1.1 (ASCII -kkv)
    ALHttp.py 1.1 (ASCII -kkv)
    ALImage.py 1.1 (ASCII -kkv)
    ALImageMatrix.py 1.4 (ASCII -kkv)
    ALImagePil.py 1.2 (ASCII -kkv)
    ALImportAction.py 1.1 (ASCII -kkv)
    ALImportCommunication.py 1.1 (ASCII -kkv)
    ALImportDataTools.py 1.2 (ASCII -kkv)
    ALImportJython.py 1.3 (ASCII -kkv)
    ALImportKernel.py 1.1 (ASCII -kkv)
    ALImportMath.py 1.1 (ASCII -kkv)
    ALImportPackageJython.py 1.1 (ASCII -kkv)
    ALImportPackagePython.py 1.1 (ASCII -kkv)
    ALImportPython.py 1.2 (ASCII -kkv)
    ALImportScript.py 1.2 (ASCII -kkv)
    ALImportSystem.py 1.8 (ASCII -kkv)
    ALInformation.py 1.1 (ASCII -kkv)
    ALInformationAction.py 1.1 (ASCII -kkv)
    ALJythonSubProcess.py 1.4 (ASCII -kkv)
    ALList.py 1.1 (ASCII -kkv)
    ALLog.py 1.1 (ASCII -kkv)
    ALLogFile.py 1.1 (ASCII -kkv)
    ALManager.py 1.1 (ASCII -kkv)
    ALManagerSignal.py 1.1 (ASCII -kkv)
    ALManagerSignalSingleton.py 1.1 (ASCII -kkv)
    ALMd5.py 1.1 (ASCII -kkv)
    ALMessage.py 1.1 (ASCII -kkv)
    ALMessageCommand.py 1.1 (ASCII -kkv)
    ALMetaObject.py 1.1 (ASCII -kkv)
    ALMultimethod.py 1.1 (ASCII -kkv)
    ALObject.py 1.5 (ASCII -kkv)
    ALPipe.py 1.1 (ASCII -kkv)
    ALPlugin.py 1.1 (ASCII -kkv)
    ALPythonSubProcess.py 1.3 (ASCII -kkv)
    ALRandom.py 1.1 (ASCII -kkv)
    ALResource.py 1.3 (ASCII -kkv)
    ALScript.py 1.1 (ASCII -kkv)
    ALScriptCompile.py 1.1 (ASCII -kkv)
    ALScriptEpydoc.py 1.1 (ASCII -kkv)
    ALScriptEric3.py 1.1 (ASCII -kkv)
    ALScriptKomodo.py 1.1 (ASCII -kkv)
    ALScriptMd5sum.py 1.1 (ASCII -kkv)
    ALScriptVersionControl.py 1.1 (ASCII -kkv)
    ALSemaphore.py 1.1 (ASCII -kkv)
    ALServer.py 1.1 (ASCII -kkv)
    ALServerHttp.py 1.1 (ASCII -kkv)
    ALServerXmlRpc.py 1.1 (ASCII -kkv)
    ALSignal.py 1.1 (ASCII -kkv)
    ALSlot.py 1.2 (ASCII -kkv)
    ALStream.py 1.1 (ASCII -kkv)
    ALStreamFile.py 1.2 (ASCII -kkv)
    ALStreamPipe.py 1.1 (ASCII -kkv)
```

```
    ALServer.py 1.1 (ASCII -kkv)
    ALServerHttp.py 1.1 (ASCII -kkv)
    ALServerXmlRpc.py 1.1 (ASCII -kkv)
    ALSignal.py 1.1 (ASCII -kkv)
    ALSlot.py 1.2 (ASCII -kkv)
    ALStream.py 1.1 (ASCII -kkv)
    ALStreamFile.py 1.2 (ASCII -kkv)
    ALStreamPipe.py 1.1 (ASCII -kkv)
    ALStreamSerial.py 1.1 (ASCII -kkv)
    ALStreamXml.py 1.1 (ASCII -kkv)
    ALString.py 1.2 (ASCII -kkv)
    ALSubProcess.py 1.2 (ASCII -kkv)
    ALSynchronized.py 1.1 (ASCII -kkv)
    ALSysInfo.py 1.1 (ASCII -kkv)
    ALSystem.py 1.4 (ASCII -kkv)
    ALTemplateClass.py 1.1 (ASCII -kkv)
    ALTest.py 1.4 (ASCII -kkv)
    ALTestCase.py 1.6 (ASCII -kkv)
    ALTestSuite.py 1.4 (ASCII -kkv)
    ALThread.py 1.2 (ASCII -kkv)
    ALThreadControl.py 1.3 (ASCII -kkv)
    ALThreadingCondition.py 1.1 (ASCII -kkv)
    ALTime.py 1.3 (ASCII -kkv)
    ALToken.py 1.1 (ASCII -kkv)
    ALTrace.py 1.1 (ASCII -kkv)
    ALUnitTest.py 1.6 (ASCII -kkv)
    ALUnitTestSuite.py 1.1 (ASCII -kkv)
    ALVerbose.py 1.3 (ASCII -kkv)
    ALXml.py 1.3 (ASCII -kkv)
    ALXmlElement.py 1.2 (ASCII -kkv)
    ALXmlNode.py 1.1 (ASCII -kkv)
    ALXsd.py 1.3 (ASCII -kkv)
    ALXsdDataBinding.py 1.1 (ASCII -kkv)
  >tests
    >DataForTests
    >PythonJythonTestDataFor
  >testsuite
    ALImportTestCase.py 1.3 (ASCII -kkv)
    ALTestCaseALObject.py 1.4 (ASCII -kkv)
    ALTestCaseALResource.py 1.3 (ASCII -kkv)
    ALTestCaseALVerbose.py 1.4 (ASCII -kkv)
    ALTestCaseALXml.py 1.1 (ASCII -kkv)
    >ALTestCaseALXsd.py 1.1 (ASCII -kkv)
    ALTestCaseALXsdDataBinding.py 1.1 (ASCII -kkv)
    ALTestSuiteKernel.py 1.4 (ASCII -kkv)
    ALTestSuiteSystem.py 1.4 (ASCII -kkv)
  >unittestsuite
    >ALUnitTestAction.py 1.3 (ASCII -kkv)
    >ALUnitTestAll.py 1.2 (ASCII -kkv)
    ALUnitTestCommunication.py 1.2 (ASCII -kkv)
    ALUnitTestJython.py 1.3 (ASCII -kkv)
    ALUnitTestKernel.py 1.3 (ASCII -kkv)
    ALUnitTestMath.py 1.2 (ASCII -kkv)
    ALUnitTestPackageJython.py 1.2 (ASCII -kkv)
    >ALUnitTestPackagePython.py 1.2 (ASCII -kkv)
    ALUnitTestScript.py 1.2 (ASCII -kkv)
    ALUnitTestServerXmlRpc.py 1.2 (ASCII -kkv)
    ALUnitTestSystem.py 1.5 (ASCII -kkv)
```

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Callback Design Pattern

- What does that mean?

It allows connect different codes (or programs) with NO RELATION between each other!! and control them...

HostCode X *ExternalCode(s)*

- Why?

To extend a code with no changes, to add features to a complex code, to connect asynchronous code, to synchronize events, etc....

- How does it works? – Steps:
  - A *HostCode* has a *"Slot"* object to connect the *ExternalCode* (in fact: the Slot object store a list of *ExternalCodes*);
  - An *ExternalCode* is connected to the *HostCode* Slot;
  - In the *HostCode* loop, an "Event" calls the Slot to execute the *ExternalCode*;
  - Optional: the *HostCode* can pass an Object to the *ExternalCode* list by its Slot.

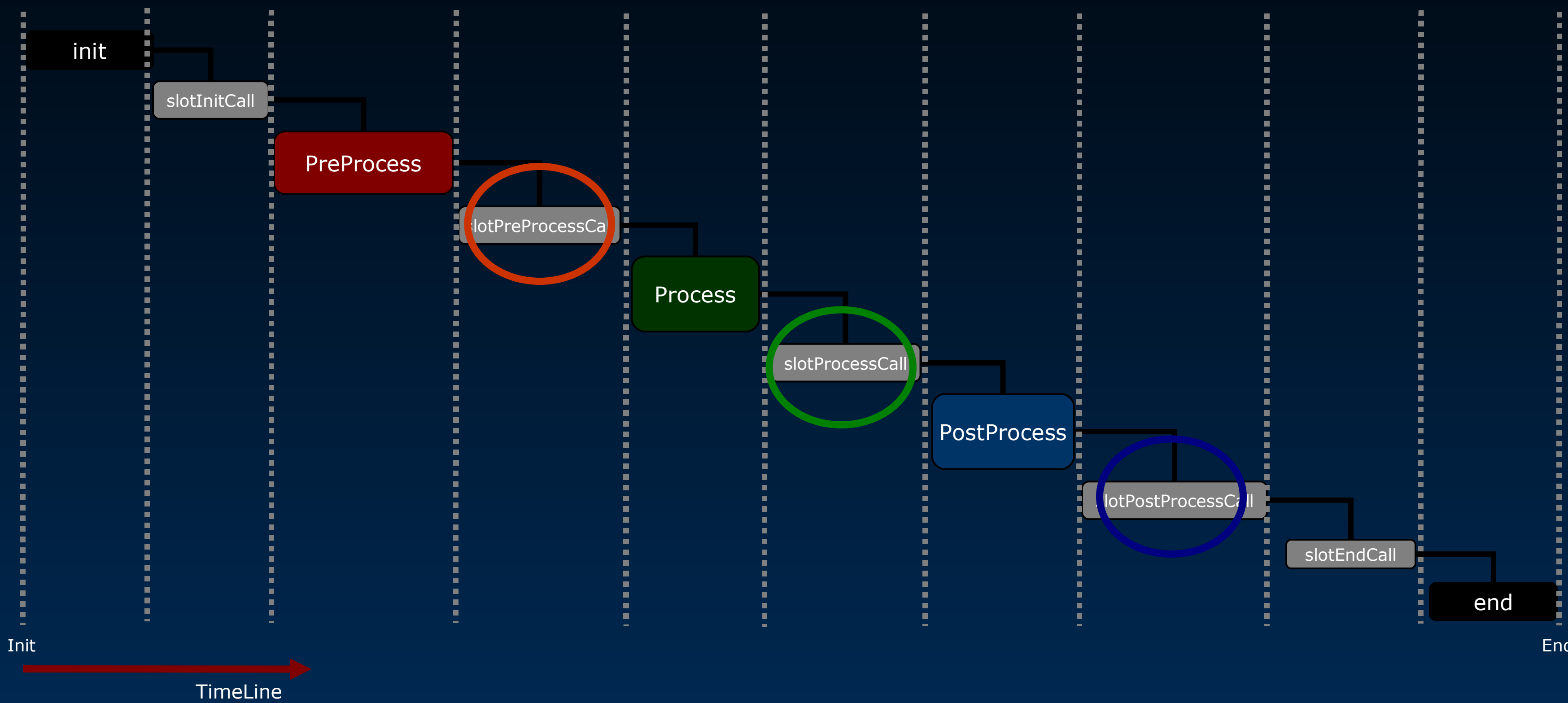
CallBack Example:
ALAction::executeAction()

```
class ClassBT :

   def methodPrintInit( self, _obj = None  ):
      ALVerbose.unitTest( "     Class B::printing in INIT: " + ALString( _obj ) )

   def methodPrintEnd( self, _obj = None  ):
      ALVerbose.unitTest( "     Class B::printing in END : " + ALString( _obj ) )

   def methodPrintRun( self, _obj = None ):
      ALVerbose.unitTest( "     Class B::printing in RUN : " + ALString( _obj ) )


........
```

```
oClassBT = ClassBT()

oalAction = ALAction()

oalAction.connectPreProcess(     oClassBT.methodPrintInit    )
oalAction.connectPostProcess(   oClassBT.methodPrintEnd   )
oalAction.connectProcess(         oClassBT.methodPrintRun    )

oalAction.executeAction()
```

CallBack Example:

ALAction::executeAction()




```
oalAction.connectPreProcess(    oClassBT.methodPrintInit )
oalAction.connectProcess(       oClassBT.methodPrintRun  )
oalAction.connectPostProcess(  oClassBT.methodPrintEnd  )
```

CallBack Example:

ALAction::executeAction()

connecting a sequence of events

```
class ClassBT :

   def methodPrintInit( self, _obj = None  ):
      ALVerbose.unitTest( "     Class B::printing in INIT: " + ALString( _obj ) )

   def methodPrintEnd( self, _obj = None  ):
      ALVerbose.unitTest( "     Class B::printing in END : " + ALString( _obj ) )

   def methodPrintRun( self, _obj = None ):
      ALVerbose.unitTest( "     Class B::printing in RUN : " + ALString( _obj ) )


........
```

```
oClassBT = ClassBT()

oalAction = ALAction()

oalAction.connectPostProcess(  oClassBT.methodPrintEnd  )
oalAction.connectPostProcess(  oClassBT.methodPrintEnd  )
oalAction.connectPostProcess(  oClassBT.methodPrintEnd  )

oalAction.connectPreProcess(    oClassBT.methodPrintInit  )
oalAction.connectPreProcess(    oClassBT.methodPrintInit  )
oalAction.connectPreProcess(    oClassBT.methodPrintInit  )

oalAction.connectProcess(       oClassBT.methodPrintRun  )


oalAction.executeAction()
```

Callback Resume

It allows connect different codes (or programs) with NO RELATION between each other!! and control them...

To extend a code with no changes, to add features to a complex code, to connect asynchronous code, to synchronize events, etc....

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Case”
- Design of the main “use case”
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Main Use Case

[Case 1: Basic Application Framework skeleton]

"A single code line to define and run a complete asynchronous multithread application based on a dynamic plugin architecture"

```
from ALImportKernel        import  ALApplication

if __name__ == '__main__':

        ALApplication( "DemoApplication", "2.0" ).execute()
```

Main Use Case

[Case 1: Basic Application Framework skeleton]

A basic application class defines a "skeleton" to RAID ("Rapid Application Development and Deployment").

It manages all logs, resources and actions creating and connecting all code with the heterogeneous operational system.

All signals and events are controlled and monitored by the main code inspector.

The main architecture to generate and control the multi-thread actions is encapsulated on the framework and its basic classes to simplify the code development.

[Case 1: Basic Application Framework skeleton]

The code fragment shows the construction of the basic skeleton. It implements the RAID platform. These code implements the ALApplication class to manage all actions (controlled threads) and callbacks. The External method "methodPrintHello" is connected to the application by the callback system.

```
from ALImportKernel        import  ALApplication
from ALImportSystem       import  ALVerbose


def methodPrintHello( _oalObject = None ):
    ALVerbose.screen( "Hello World" )


if __name__ == '__main__':

    # JIT compiler accelerator
     ALCompiler.accelerator()

    # Application Framework definition
    obLoadPlugins   = False
    oalApplication   = ALApplication( "DemoApplication", "2.0", obLoadPlugins )

    # Application Callback Connexion
    oalApplication.connectExecute( methodPrintHello )

    # Application Execution
    oalApplication.execute()
```

[Case 1: Basic Application Framework skeleton]

How does it work?

```
# JIT compiler accelerator
ALCompiler.accelerator()
```

- The Python runtime code accelerator is optional. It optimises the code execution only in X86 processors.

```
# Application Framework definition
obLoadPlugins   = False
oalApplication  = ALApplication( "DemoApplication", "2.0", obLoadPlugins )
```

- The ALApplication class is created with the option "no plugins". It avoid the search and store of the information of plugins. The plugins are loaded when the plugin object is instantiated, not when it is found. This mechanism is introduced later in other tutorial.

[Case 1: Basic Application Framework skeleton]

```
# Application Callback Connexion
oalApplication.connectExecute( methodPrintHello )
```

- The callback system is a "slot" to connect any external code to the central loop of the application. In the example, the "methodPrinterHello" method is glued to the main execution point. The main loop will call this external method during the execution.

```
# Application Execution
oalApplication.execute()
```

- The main application loop is started when the method "execute" is invocated.

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use case
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Design of the main use case

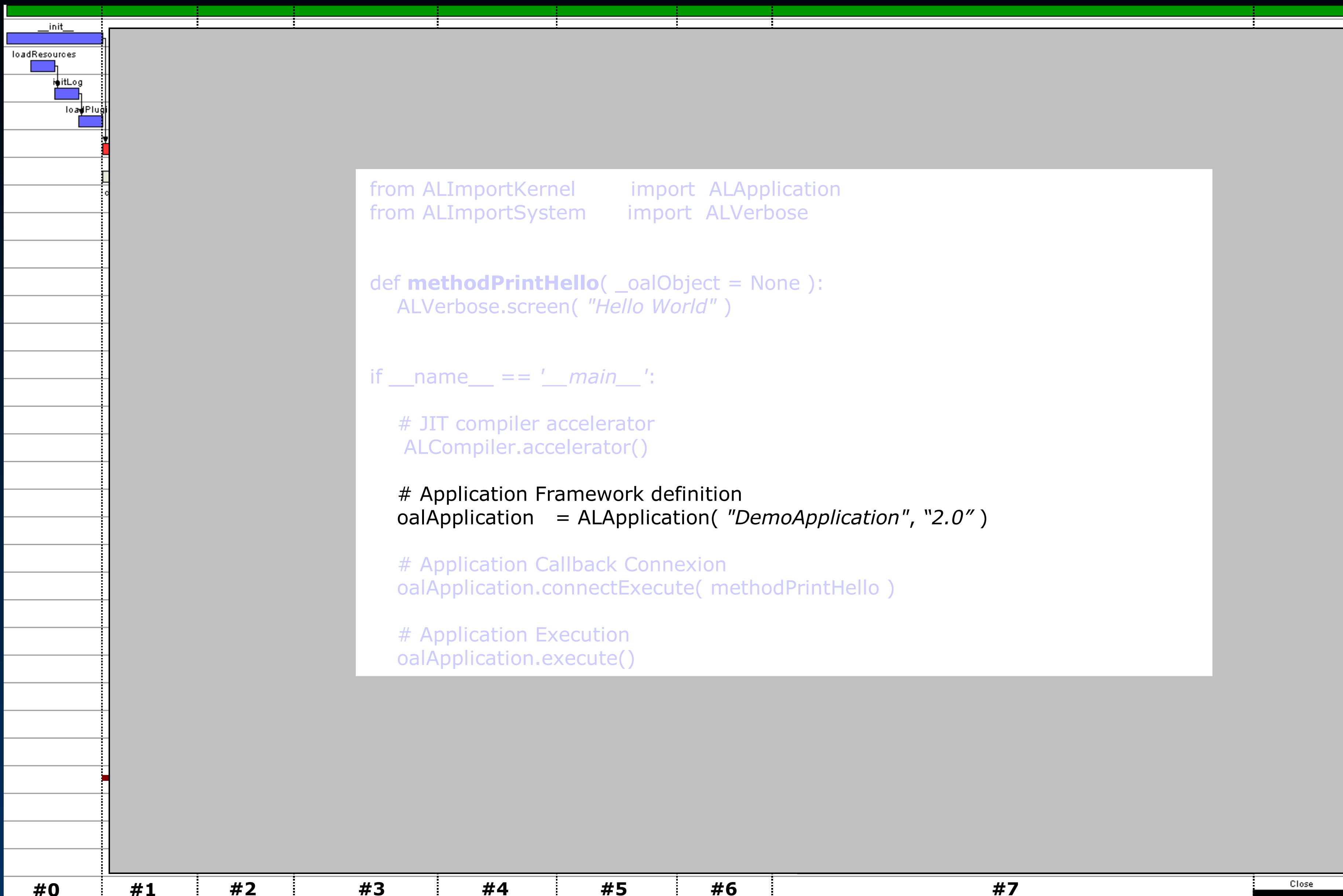
```
from ALImportKernel        import  ALApplication

if __name__ == '__main__':

    ALApplication( "DemoApplication", "2.0" ).execute()
```

AALib Internals:

ALApplication::execute ()

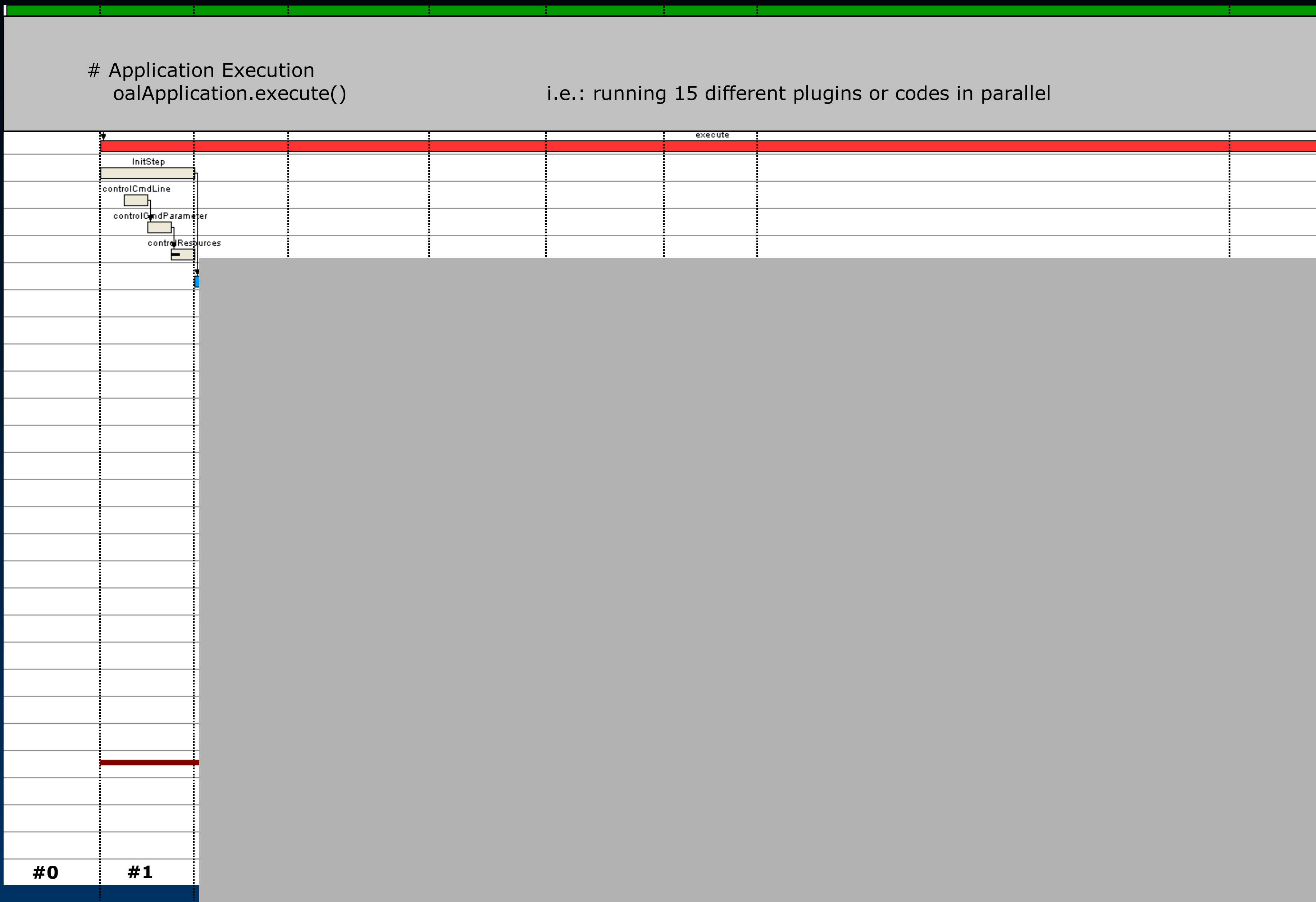
The Massive Parallel Kernel

ALApplication::execute()
"Gantt Chart – 8 STEPS"

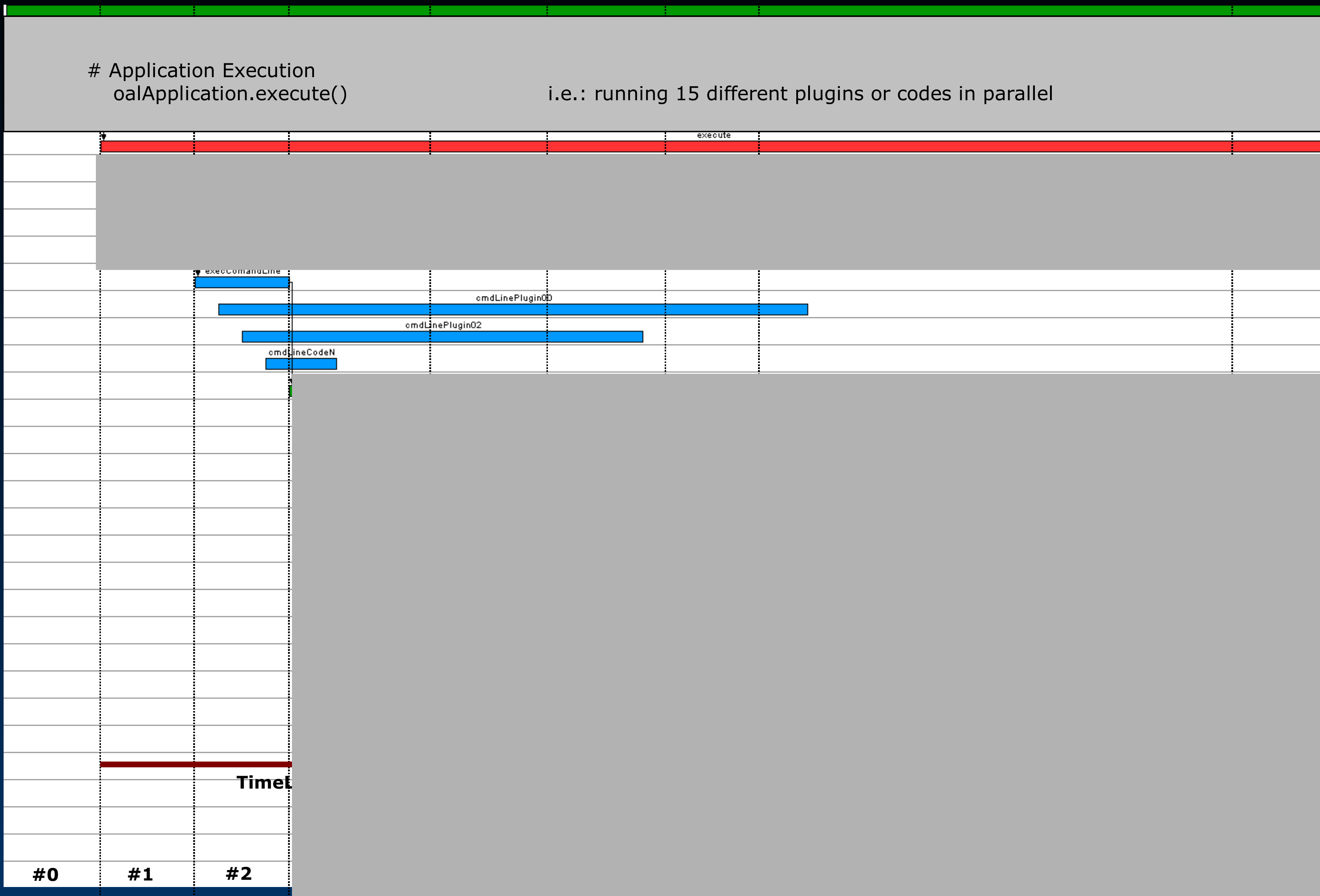



```
from ALImportKernel      import  ALApplication
from ALImportSystem      import  ALVerbose


def methodPrintHello( _oalObject = None ):
    ALVerbose.screen( "Hello World" )


if __name__ == '__main__':

    # JIT compiler accelerator
     ALCompiler.accelerator()

    # Application Framework definition
    oalApplication   = ALApplication( "DemoApplication", "2.0" )

    # Application Callback Connexion
    oalApplication.connectExecute( methodPrintHello )

    # Application Execution
    oalApplication.execute()
```

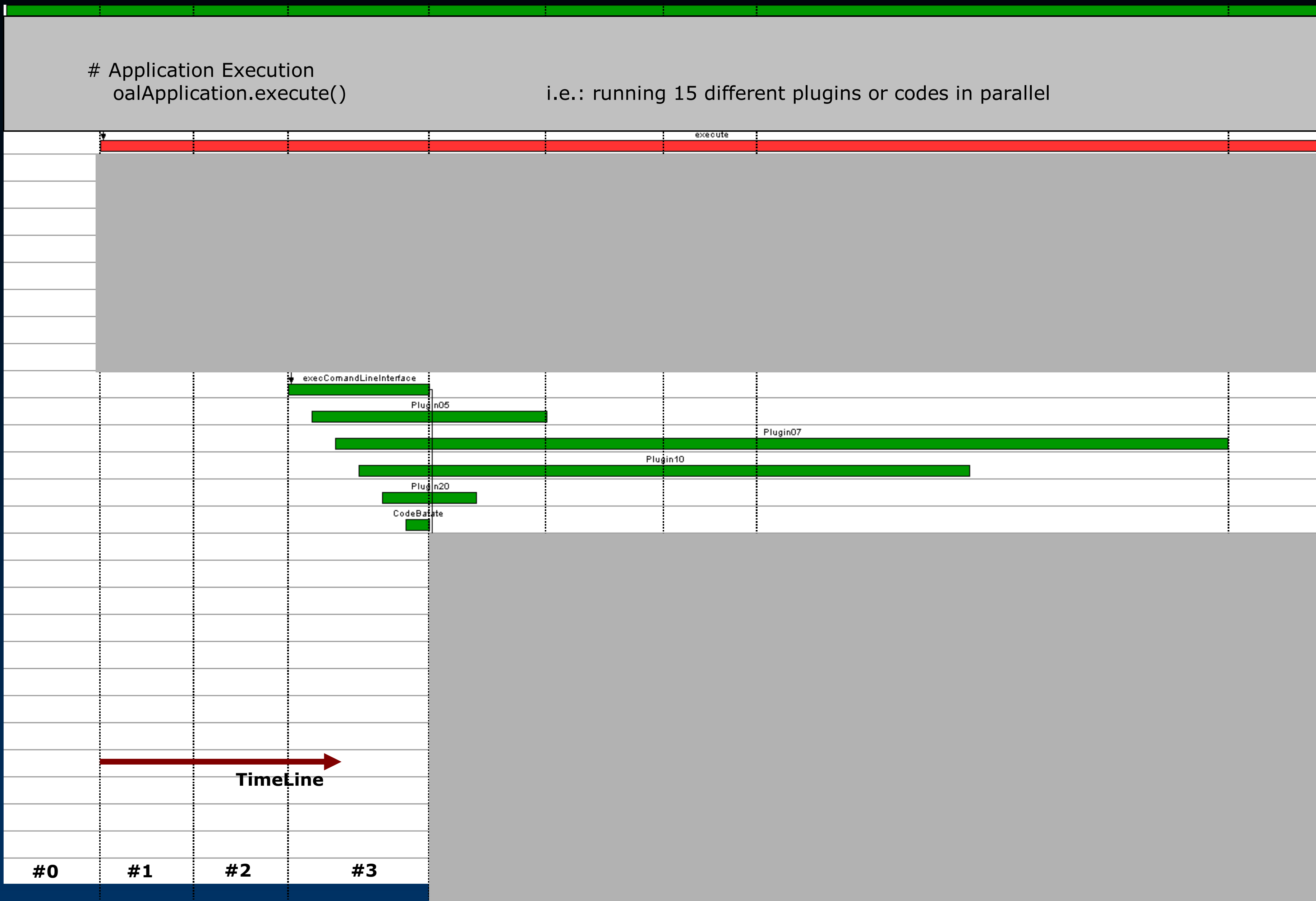
ALApplication::execute()
"Gantt Chart – 8 STEPS"



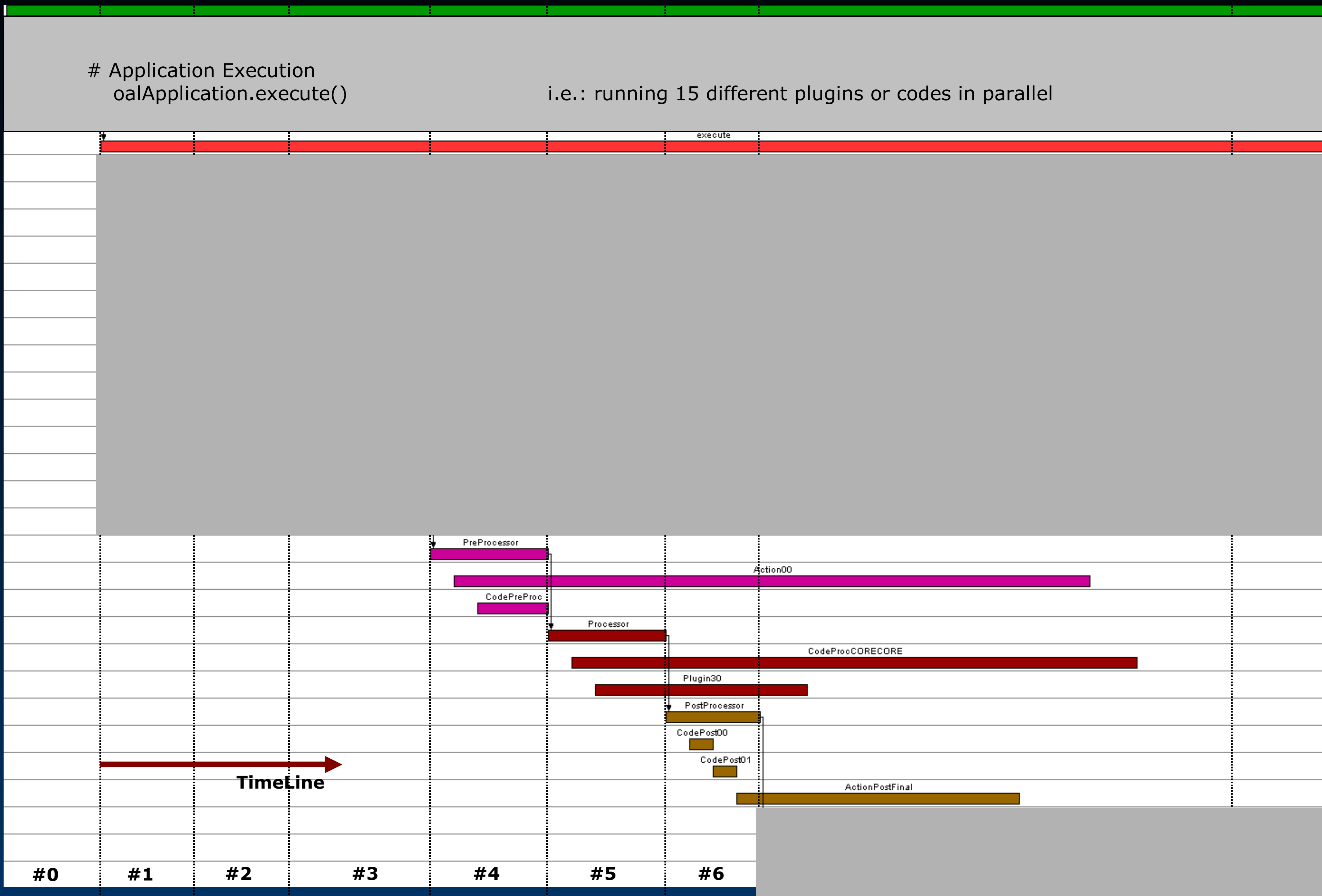
ALApplication::execute()
“Gantt Chart – 8 STEPS”



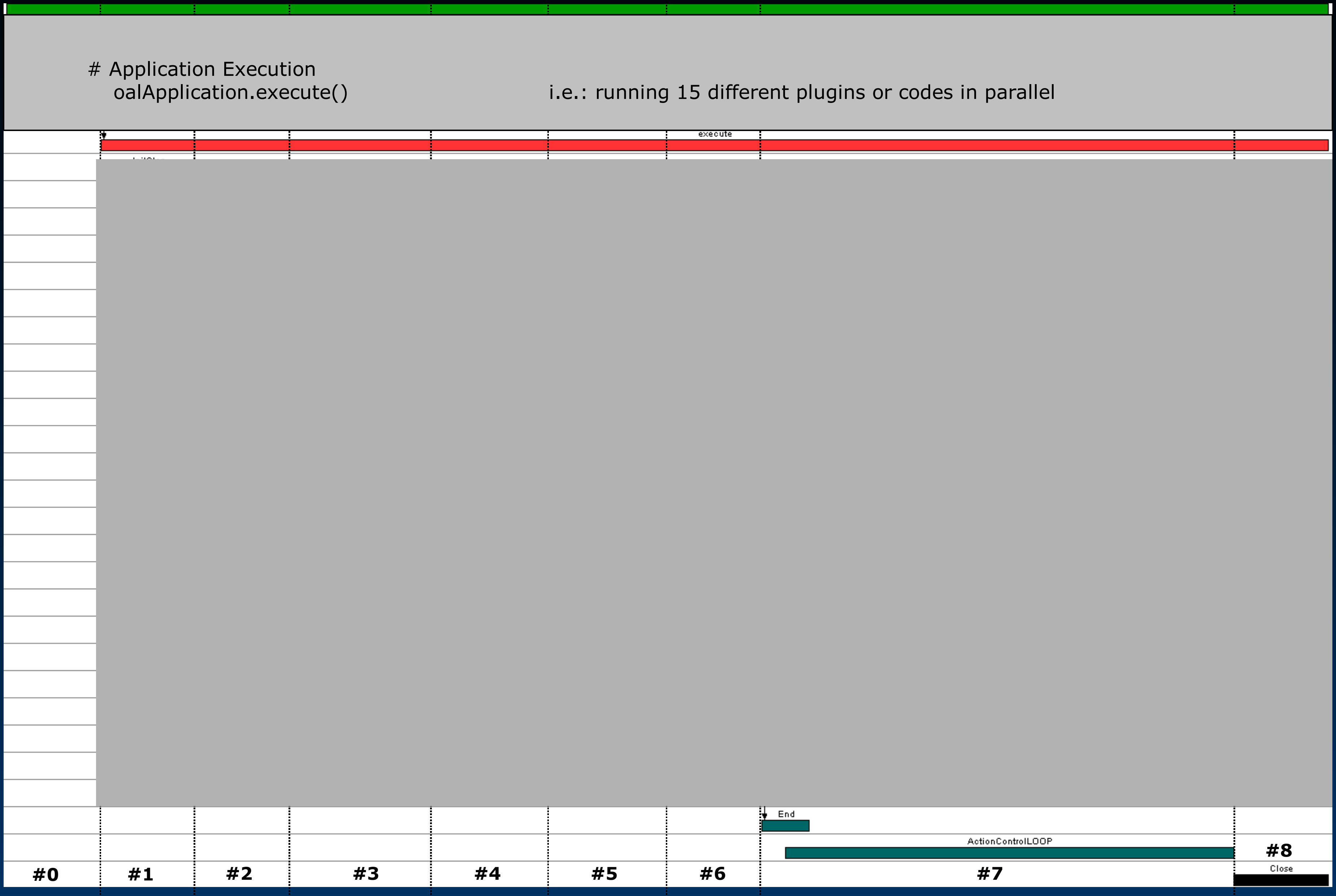
ALApplication::execute()
“Gantt Chart – 8 STEPS”



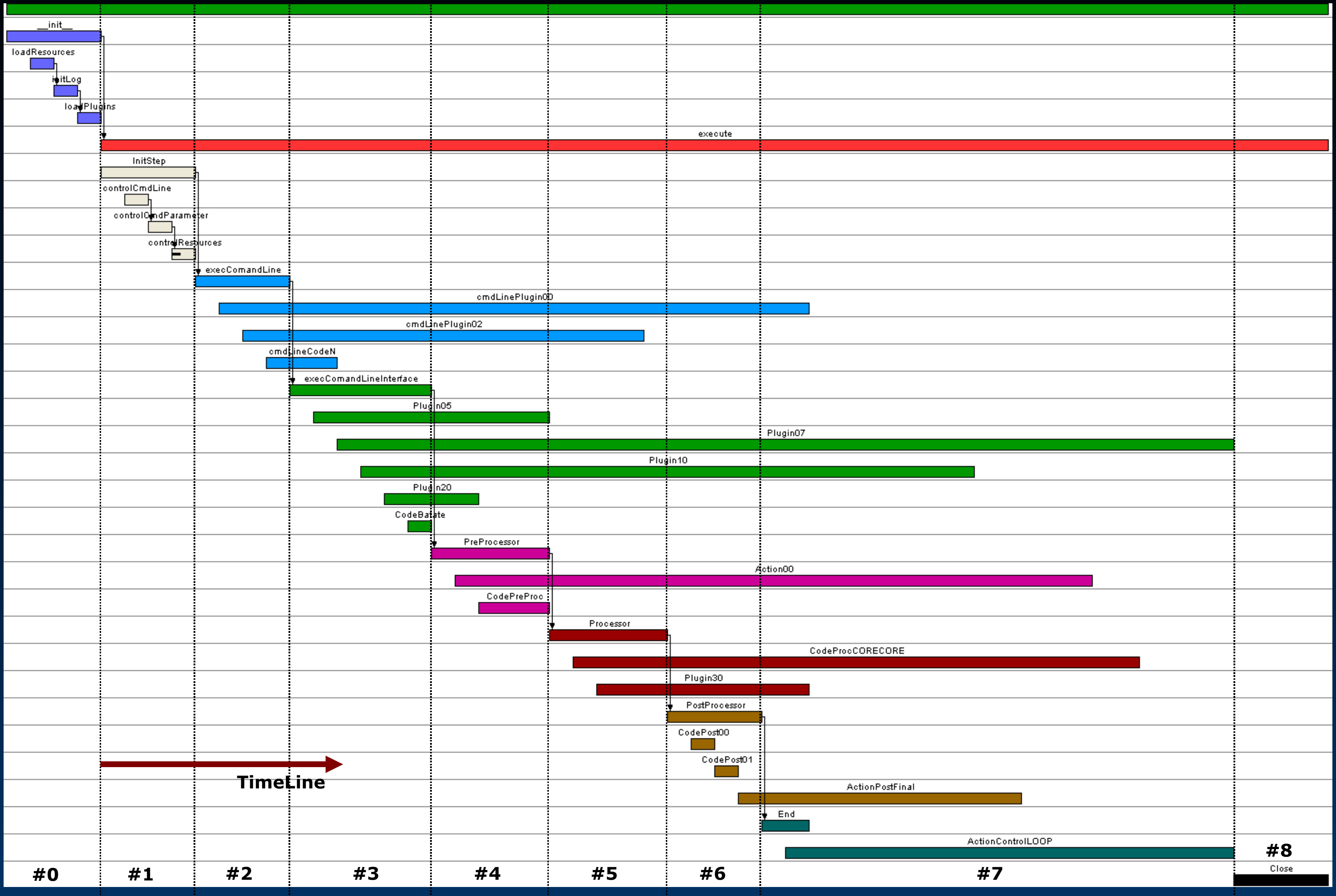
ALApplication::execute()
"Gantt Chart – 8 STEPS"



ALApplication::execute()
"Gantt Chart – 8 STEPS"



ALApplication::execute()
"Gantt Chart – 8 STEPS"



AALib Internals:
ALApplication::execute ()
The CODE

```
################################################################################


    def execute( self, _bInit = True ):
        """
        This is the Main ALAppDefinitionCore Loop Control.
        """
        ALVerbose.DEBUG( "Execution: ALAppDefinitionCore.execute()" )
        try:
            try:
                if (self.__executeCoreInit( _bInit )==True):
                    self.__executeCore()


            except ALExceptionSignal, _oalExceptionSignal:
                oalSignal = _oalExceptionSignal.getSignal()
                ALVerbose.DEBUG( "Execution: ALAppDefinitionCore.execute() - AL_FRAMEWORK_FORCE_CLOSE_APPLICATION" )
                if ( oalSignal!=None ):
                    ALVerbose.DEBUG("ALAppDefinitionCore.execute() - stopped by Signal: "  + ALString( oalSignal.getSignal() ) )
                else:
                    ALVerbose.DEBUG("ALAppDefinitionCore.execute() - stopped under Unknown signal: "  + ALString( oalSignal.getSignal() ) )
                self.__executeCoreRaise()


            except:
                ALVerbose.error( "Execution: ALAppDefinitionCore.execute() - FATAL ERROR - UNEXPECTED ERROR RAISED" )
                self.__executeCoreRaise()

        finally:
            self.__executeCoreClose()


################################################################################
```

Show Time

Other Use Cases

[Plugin Framework]

[XML-RPC server plugin architecture]

[Basic Callback System]

[Data Binding – XML Scheme object construction]

[SubProcess Management]

Main Use Case Resume

A basic application class defines a "skeleton" to RAID ("Rapid Application Development and Deployment").

It manages all logs, resources and actions creating and connecting all code with the heterogeneous operational system.

All signals and events are controlled and monitored by the main code inspector.

The main architecture to generate and control the multi-thread actions is encapsulated on the framework and its basic classes to simplify the code development.

Main Use Case Resume

A basic application "skeleton" elements for dynamic plugins:

Automatic:

- **One line code to define the ALApplication object;**
- **a folder to store the directories of plugins**
  default place = "PROJECT_ROOT/plugins";
- **the folders with plugins;**

Static:

- **One line code to define the ALApplication object;**
- **a folder to store the directories of plugins**
  default place = "PROJECT_ROOT/plugins";
- **the folders with plugins;**
- **a plugin mapping file definition in the plugin folder - default file = "PROJECT_ROOT/plugins/comandLineInterface.xml";**

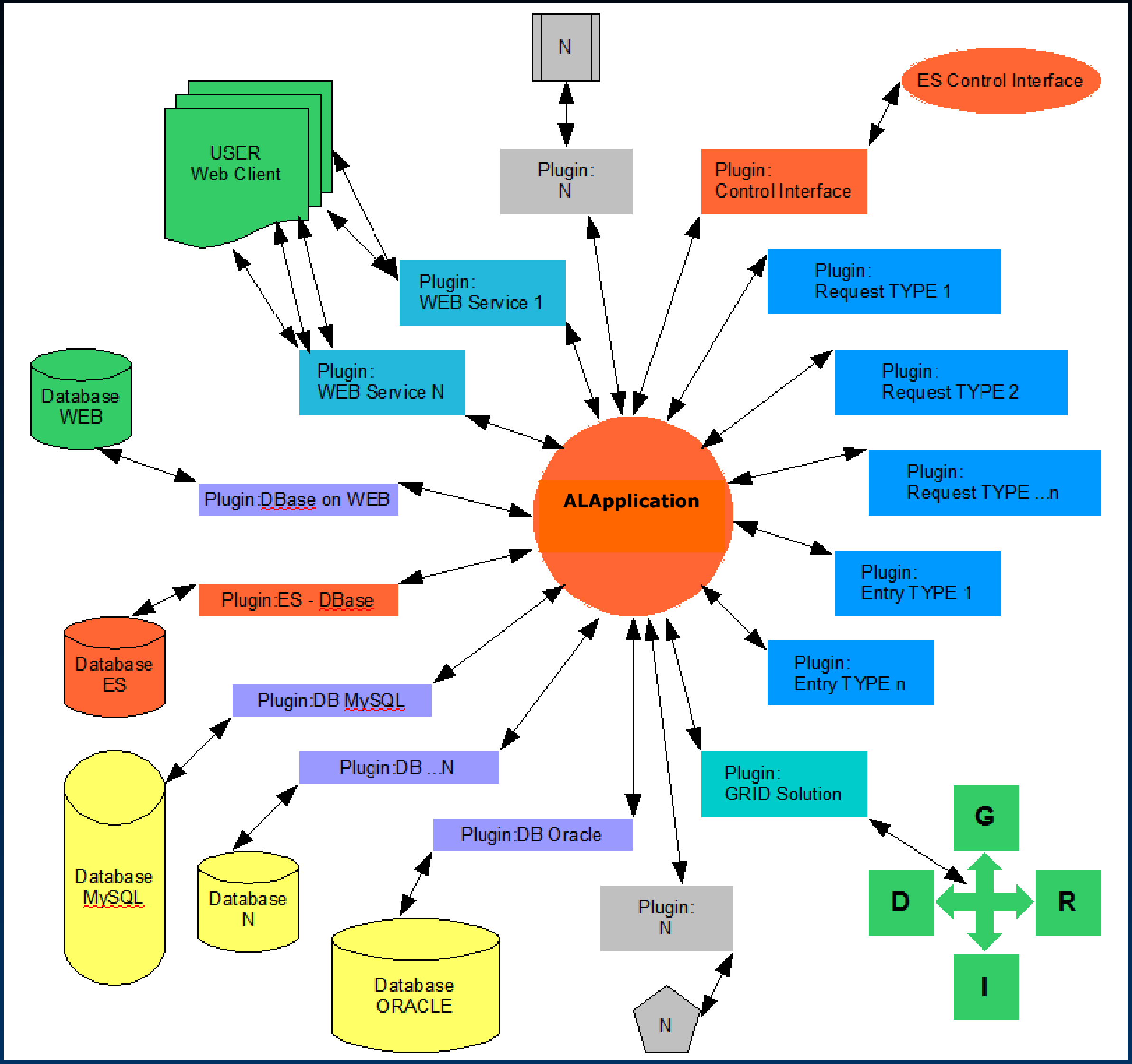
Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main "Use Cases"
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Plugin Architecture

- Dynamic Plugin Architecture

The plugin architecture add new functionalities to the ALApplication.



Plugin Architecture

- Dynamic Plugin Architecture

The “plugin” is a python-jython-java code dynamically generated and add to the main ALApplication framework. The mechanism can be used with any code.

The based “plugin” defined by the AALib comes from the ALAction and it is full “multi-thread” and “Thread Safety”

The “plugin” is defined by the name of the “file”. The main class inside must have the same “name”

The “plugin” can be generated from a pair of “XSD+Xml” files = Data Binding Mechanism.

A special xml file(s) can “map” the plugin set. It manages all information needed to characterise the commands and information of the plugin (help, man, etc...)

A basic Plugin

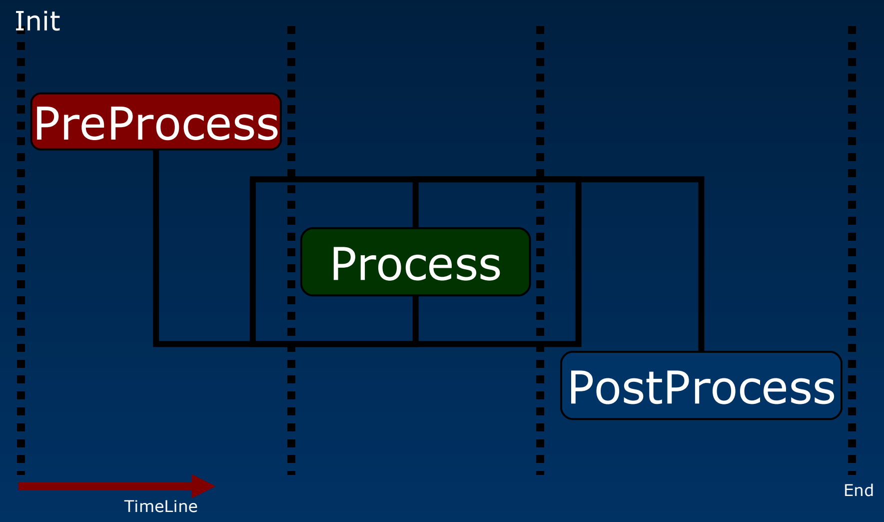
- A standard plugin code

the file: "PluginExample.py"

```
from ALVerbose   import ALVerbose
from ALPlugin    import ALPlugin


class PluginExample( ALPlugin ):

    def process( self, _oalObject = None ):
        ALVerbose.screen( "Hello World from the PluginExample" )
......
# To be used in a code somewhere:


oalPlugin = ALApplication.getPluginObject( "PluginExample" )
if (oalPlugin!=None):
    oalPlugin.executePlugin()
```

- Thread Architecture (inherits from ALAction)

ALPlugin::executePlugin()



Plugin Architecture Resume

The “plugin” is a python-jython-java code dynamically generated and add to the main ALApplication framework.

The based “plugin” is full “multi-thread” and “Thread Safety”

The “plugin” can be generated from a pair of “XSD+Xml” files = Data Binding Mechanism.

A special xml file(s) can “map” the plugin set. It manages all information needed to characterise the commands and information of the plugin (help, man, etc...)

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Plugin Mapping with a XML file

Defined by:
“The Command Line Interface Class”
presented on the use case Gantt chart

```xml
<!-- It shows the execution of a standart Python class -->
<commandLineElement>
	<command>--test</command>
	<argument></argument>
	<commandDescription>Default test argument to see this information</commandDescription>
	<class>ClassPluginFileDiferentName</class>
	<classMethod>methodVerbose</classMethod>
	<classFileName>./DataForTests/ClassPluginFile.py</classFileName>
	<plugin></plugin>
	<pluginMethod></pluginMethod>
	<pluginFileName></pluginFileName>
	<help>The HELP lines on the screen for
			the command --test.</help>
	<manual>The MANUAL lines on the screen for
			the command --test.</manual>
	<webAdress>http://aalib.sourceforge.net</webAdress>
	<webUpDate>http://aalib.sourceforge.net</webUpDate>
	<copyright>AALib 2005-2007</copyright>
	<author>Romeu Andre Pieritz</author>
	<date>[ 2007/05/15 13:47:13 ]</date>
</commandLineElement>

<!-- Shows a AALIB plugin execution -->
<commandLineElement>
	<command>--test2</command>
	<argument>9999</argument>
	<commandDescription>Test number two</commandDescription>
	<class></class>
	<classMethod></classMethod>
	<classFileName></classFileName>
	<plugin>PluginTest01</plugin>
	<pluginMethod></pluginMethod>
	<pluginFileName>./DataForTests/FolderPlugins/PluginTest01.py</pluginFileName>
	<help>The HELP lines on the screen for
			the command --test2.</help>
	<manual>The MANUAL lines on the screen for the command --test2.</manual>
	<webAdress>http://aalib.sourceforge.net</webAdress>
	<webUpDate>http://aalib.sourceforge.net</webUpDate>
	<copyright>AALib 2005-2007</copyright>
	<author>Romeu Andre Pieritz</author>
	<date>[ 2007/05/15 13:50:13 ]</date>
</commandLineElement>

<!-- This example shows the minimum to execute an standart AALib plugin : executePlugin() -->
<commandLineElement>
	<command>--test3</command>
	<plugin>PluginTest02</plugin>
	<pluginFileName>./DataForTests/FolderPlugins/PluginTest02.py</pluginFileName>
	<!-- optional line: need it only when the application framework is no used-->
</commandLineElement>
```

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Data Binding by XSD definition

- **Why?**

XML data binding refers to the process of representing the information in an XML document as an object in computer memory.

This allows applications to access the data in the XML from the object rather than using the DOM to retrieve the data from a direct representation of the XML itself.

- **How?**

An XML data binder accomplishes this by automatically creating a mapping between elements of the XML schema of the document we wish to bind and members of a class to be represented in memory.

- **Where?**

When this process is applied to convert an XML document to an object, it is called unmarshalling. The reverse process, to serialize an object as XML, is called marshalling.

Data Binding by XSD definition

Example:
"The ComandLineInterface Data"

The XSD Definition

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:complexType name="ALCommandLineInterfaceElement">
        <xsd:sequence>
            <xsd:element name="command"                type="xsd:string"/>
            <xsd:element name="argument"               type="xsd:string"/>
            <xsd:element name="commandDescription"     type="xsd:string"/>
            <xsd:element name="class"                  type="xsd:string"/>
            <xsd:element name="classMethod"            type="xsd:string"/>
            <xsd:element name="classFileName"          type="xsd:string"/>
            <xsd:element name="plugin"                 type="xsd:string"/>
            <xsd:element name="pluginMethod"           type="xsd:string"/>
            <xsd:element name="pluginFileName"         type="xsd:string"/>
            <xsd:element name="help"                   type="xsd:string"/>
            <xsd:element name="manual"                 type="xsd:string"/>
            <xsd:element name="webAdress"              type="xsd:string"/>
            <xsd:element name="webUpDate"              type="xsd:string"/>
            <xsd:element name="copyright"              type="xsd:string"/>
            <xsd:element name="author"                 type="xsd:string"/>
            <xsd:element name="date"                   type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<xsd:include schemaLocation="ALCommandLineInterfaceElement.xsd"/>

    <xsd:complexType name="ALCommandLineInterfaceDefinition">
        <xsd:sequence>
            <xsd:element name="date"                   type="xsd:string"/>
            <xsd:element name="version"                type="xsd:string"/>
            <xsd:element name="name"                   type="xsd:string"/>
            <xsd:element name="information"            type="xsd:string"/>
            <xsd:element name="author"                 type="xsd:string"/>
            <xsd:element name="commandLineElement"     minOccurs="0" maxOccurs="unbounded"
                                                       type="ALCommandLineInterfaceElement"/>
        </xsd:sequence>
    </xsd:complexType>

</xsd:schema>
```

The code:

```
oalXsdDataBinding = ALXsdDataBinding( "cmdLineInt.xsd" )
oalObject = oalXsdDataBinding.getObject(  "cmdLineIntModel.xml" )
If (oalObject!=None):
     oalObject.outputXml()
```

Data Binding by XSD definition

Example:
"The ComandLineInterface Data"

The XML File

```xml
<!-- This example shows the minimum to execute an standart PyAALib plugin : executePlugin() -->
<commandLineElement>
      <command>--jython01</command>
      <plugin>PluginJythonTest01</plugin>
      <help>The HELP lines on the screen for
                  the command --jython.</help>
      <manual>The MANUAL lines on the screen for
                 the command --jython.</manual>
      <webAdress>http://pypaalib.sourceforge.net</webAdress>
      <webUpDate>http://pyaalib.sourceforge.net</webUpDate>
      <copyright>PyAALib 2005-2007</copyright>
      <author>Romeu Andre Pieritz</author>
      <date>[ 2007/09/10 13:47:13 ]</date>
 </commandLineElement>

<!-- This example shows the minimum to execute an standart PyAALib plugin : executePlugin() -->
<commandLineElement>
      <command>--jython02</command>
      <plugin>PluginJythonTest02</plugin>
      <help>The HELP lines on the screen for
                  the command --jython.</help>
      <manual>The MANUAL lines on the screen for
                 the command --jython.</manual>
      <webAdress>http://pypaalib.sourceforge.net</webAdress>
      <webUpDate>http://pyaalib.sourceforge.net</webUpDate>
      <copyright>PyAALib 2005-2007</copyright>
      <author>Romeu Andre Pieritz</author>
      <date>[ 2007/09/10 13:47:13 ]</date>
 </commandLineElement>

<!-- This example shows the minimum to execute an standart PyAALib plugin : executePlugin() -->
<commandLineElement>
      <command>--jython03</command>
      <plugin>PluginJythonTest03</plugin>
      <help>The HELP lines on the screen for
                  the command --jython.</help>
      <manual>The MANUAL lines on the screen for
                 the command --jython.</manual>
      <webAdress>http://pypaalib.sourceforge.net</webAdress>
      <webUpDate>http://pyaalib.sourceforge.net</webUpDate>
      <copyright>PyAALib 2005-2007</copyright>
      <author>Romeu Andre Pieritz</author>
      <date>[ 2007/09/10 13:47:13 ]</date>
 </commandLineElement>
```

Data Binding by XSD definition

Example:
"The code example to generate the Python - Jython Binding"

```
ALVerbose.unitTest( "Test ALTestCaseALXsdDataBinding" )

oalStrXsdInputFile = ALString( "./DataForTests/people.xsd")
oalStrXmlInputFile = ALString( "./DataForTests/people.xml" )


ALVerbose.unitTest( "Testing the people.xsd import module people.py" )
oalXsdDataBinding3 = ALXsdDataBinding( oalStrXsdInputFile )
oalObject3 = oalXsdDataBinding3.getObject( oalStrXmlInputFile )
oalObject3.outputFile( "testOuputPeopleXsd03.xml" )


ALVerbose.unitTest( "Testing the people.xsd import module module_people4.py" )
oalStrPyOutputFile = ALString( "module_people4.py" )
oalXsdDataBinding4 = ALXsdDataBinding( oalStrXsdInputFile, oalStrPyOutputFile )
oalObject4 = oalXsdDataBinding4.getObject( oalStrXmlInputFile )
oalObject4.outputFile( "testOuputPeopleXsd04.xml" )


ALVerbose.unitTest( "Testing the people.xsd import module module_people5.py" )
oalStrPyOutputFile = ALString( "module_people5.py" )
oalXsdDataBinding5 = ALXsdDataBinding( oalStrXsdInputFile )
oalObject5 = oalXsdDataBinding5.getObject( oalStrXmlInputFile, oalStrPyOutputFile )
oalObject5.outputFile( "testOuputPeopleXsd05.xml" )


ALVerbose.unitTest( "Testing the empty instance of the object" )
ALVerbose.unitTest( "Testing the people.xsd import module module_people6.py" )
oalXsdDataBinding6 = ALXsdDataBinding( oalStrXsdInputFile )
oalObject6 = oalXsdDataBinding6.getObject()
# oalObject6.setToken00()
# oalObject6.set.....
oalObject6.outputFile( "testOuputPeopleXsd06.xml" )


ALVerbose.unitTest()
```

Data Binding by XSD definition

Resume

- It is used to represent only DATA
- Extremely powerful to describe complex object hierarchy of Data
- It can NOT to be used to store LOGIC ("code")
- It can be used to simplify the data manipulation : automatic generation of code
- It is used also for data plugin generation "on the fly" by AALib framework

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

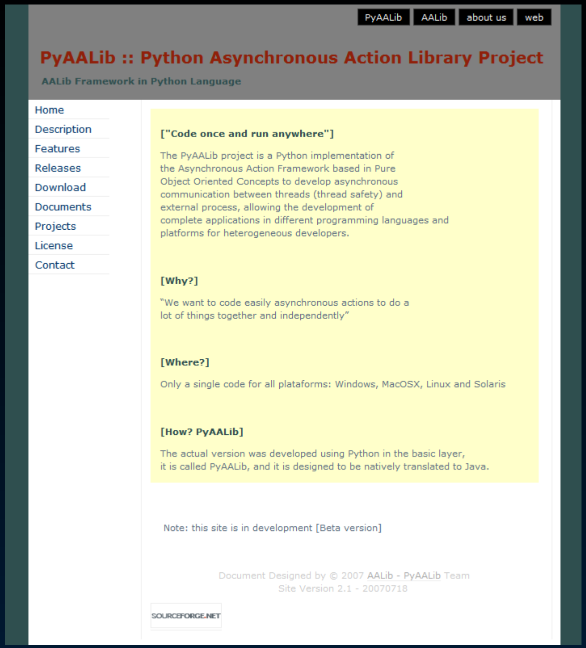
Plugins based on XSD definitions

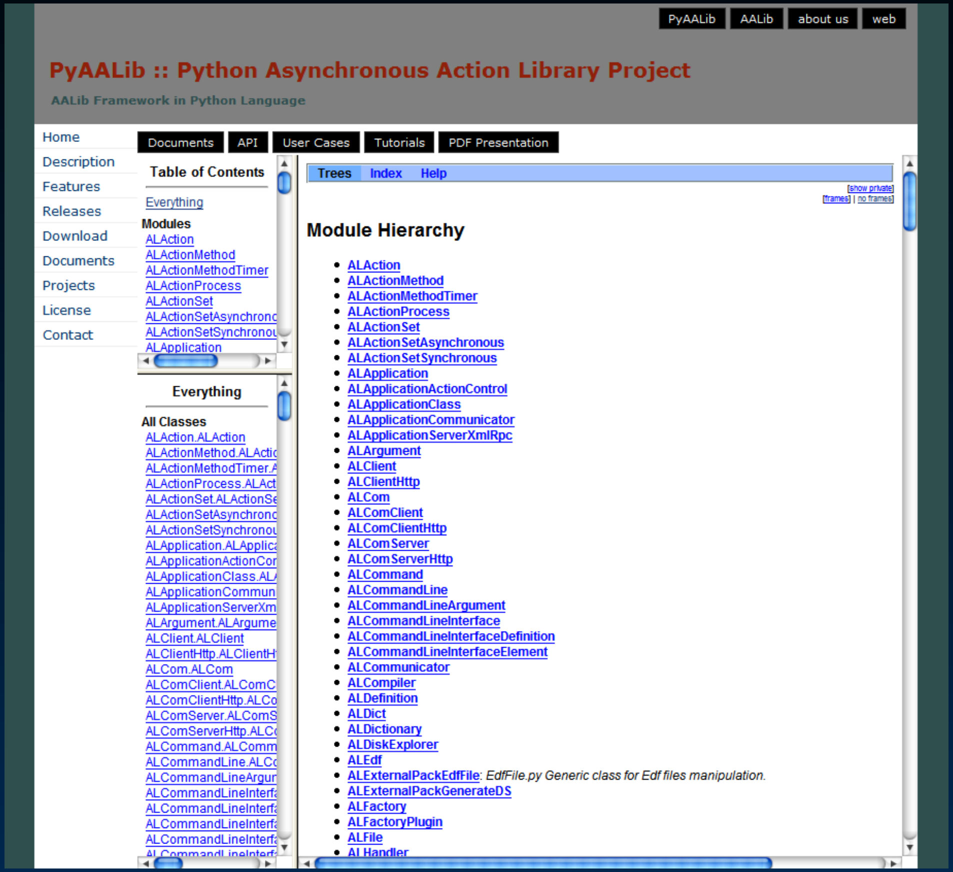
- The plugin is defined by an XSD scheme file
- The object instance will be created from the XML data file.

```
odtFileName = odtTestSet.getFileNameCrystalTestResult( )
if( DTDiskExplorer.existFile( odtFileName )):
    odtPlugin = DTApplication.getPluginObject( "DTCrystalTestResult", odtFileName )
    if (odtPlugin!=None):
        odtGroupTestResult.addCrystalTestResult( odtPlugin )
```

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices





Documentation

http://aalib.sourceforge.net

http://pyaalib.sourceforge.net

http://jyaalib.sourceforge.net

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Code Syntax and rules

```
from ALVerbose   import ALVerbose
from ALPlugin    import ALPlugin


class ALPluginExample( ALPlugin ):

    def process( self, _oalObject = None ):
        ALVerbose.screen( "Hello World from the PluginExample" )
......
# To be used in a code somewhere:


oalPlugin = ALApplication.getPluginObject( "ALPluginExample" )
if (oalPlugin!=None):
    oalPlugin.executePlugin()
```

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main "Use Cases"
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Show Time

Objective TODAY

- Overview
- Understand the basis
- View of the main classes and tree
- Basic Callback Concepts
- Understand the main “Use Cases”
- Design of the main use cases
- Plugin architecture
- Plugin mapping using XML files
- Data Binding with XSD
- Plugins based on XSD definitions
- Documentation
- Code syntax and rules
- Examples using the CVS repository
- Discussion about Best Practices

Discussion

- What we are still developing until the release?

Process manager class

Plugin manager class - Plugin version system

Improve some sub systems to be more flexible – XSD, XML parsing by "path", etc.

Documentation, documentation, documentation

Bugzilla

External packages for Jython-JAVA: Image manipulation, HDF5, EDF files, etc.

Test and Add some interfaces for "high value" standard JAVA libraries as "External Packages"

Testing and encapsulating the SWT (eclipse GUI native elements) to be "easy" to use

..... more .....

- The Limitations of the JAVA architecture!

OOP limitations – the virtual mechanism is not trivial
External Process manager is Poor – or NULL
2x slower compared to native python

Thanks !

Documents:

http://aalib.sourceforge.net
http://pyaalib.sourceforge.net
http://jyaalib.sourceforge.net - 2008
http://edfexplorer.sourceforge.net
http://beamfocus.sourceforge.net
http://www.esrf.eu/UsersAndScience/Experiments/TBS/SciSoft/

Contact:

aalib_info@yahoo.com
romeu.pieritz@gmail.com