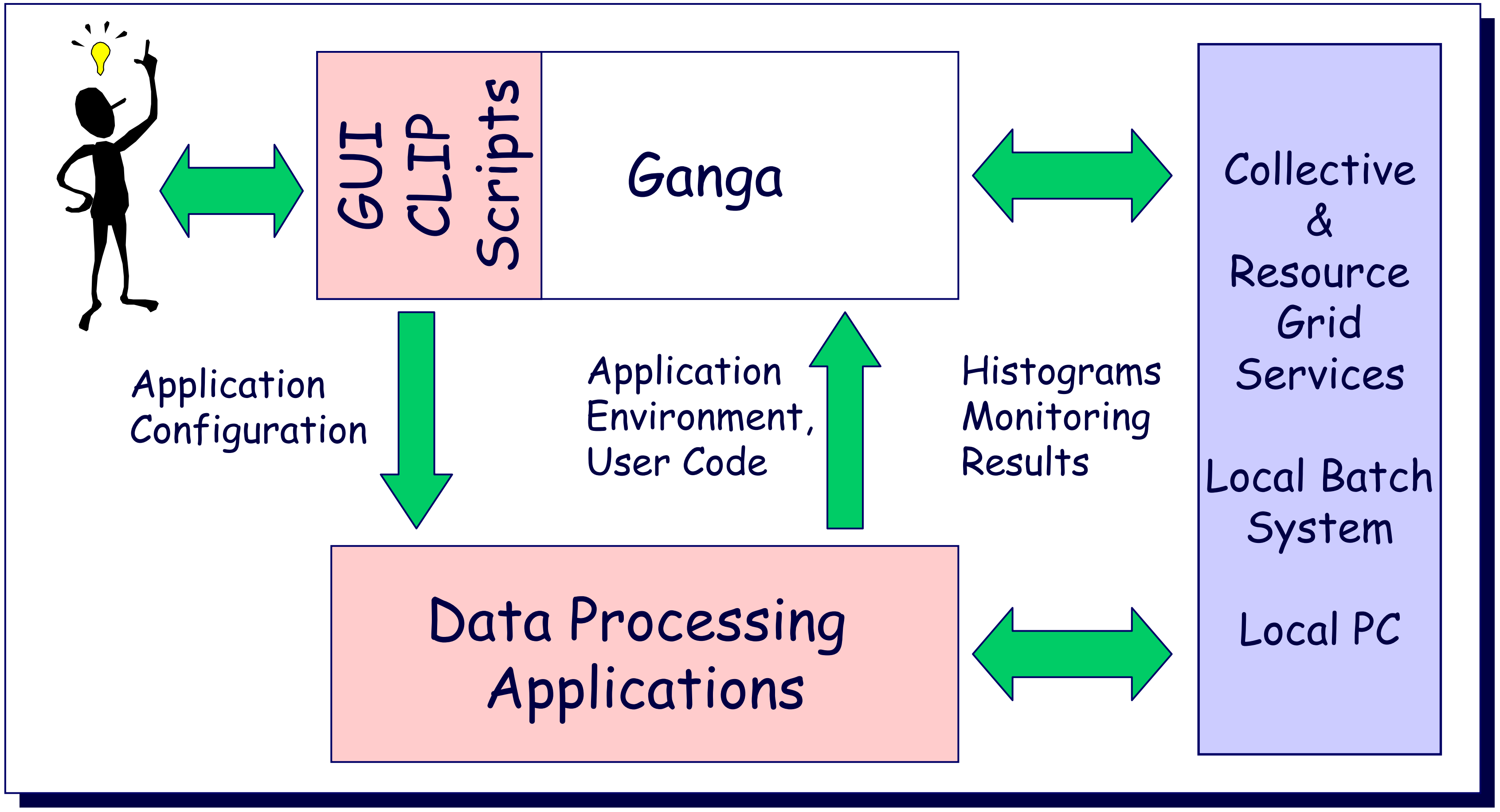
Using Python in the Development of a Grid User Interface for Distributed Data Analysis

Alexander Soroko (Oxford University)
K. Harrison (Cavendish Laboratory, University of Cambridge)
C.L. Tan (School of Physics and Astronomy, University of Birmingham)
D. Liko, A. Maier, J.T. Moscicki, A. Muraru, H.C. Lee (CERN)
U. Egede (Department of Physics, Imperial College London)
R.W.L. Jones (Department of Physics, University of Lancaster)
J. Elmsheuser (Ludwig-Maximilians-Universitat, Munchen)
G.N. Patrick (Rutherford Appleton Laboratory)

- **General Overview**
- **Architecture**
- **Serialization and Persistency**
- **Configuration, Logging and Help**
- **Building the User Interfaces**
- **Conclusions**

General Overview



General Overview

- Ganga is a Grid user interface for High Energy Physics experiments
- It is a key piece of the distributed-analysis systems for ATLAS and LHCb
- Gigabytes or even terabytes of input data are distributed around the globe
- Discovery of dataset locations => recourse to various metadata and file catalogues
- Ganga manages large scale scientific applications on the Grid:
  - configuring the applications
  - switching between testing on a local batch system and large-scale processing on the Grid
  - keeping track of results

Why Python?

- Clear object-oriented language, easy to learn
  - New developers can start faster
- Need to glue together many different packages
  - Python extension mechanism provides an easy possibility to integrate external packages like Qt and ROOT written in C/C++, into the framework
  - Most of such external packages have already pre build Python wrappers (pyQT, pyROOT, GaudiPython)
  - Implementation of internal Ganga modules in Python is a natural choice
- Easy execution of the OS commands
  - Not all external resources have an API => access via os.popen*(), os.spawn*(), commands module
  - Setting up environment => os.environ
  - File and directory management => os and shutil modules

Why Python?

- Most of the functionality we need is readily accessible via standard Python modules:
  - Multithreading support => thread, threading
  - Networking => socket, xmlrpclib
  - String manipulation => string, re
  - Messaging and logging => logging
  - Package Configuration => ConfigParser
  - Command line argument interpretation => optparse
  - Testing framework => unittest

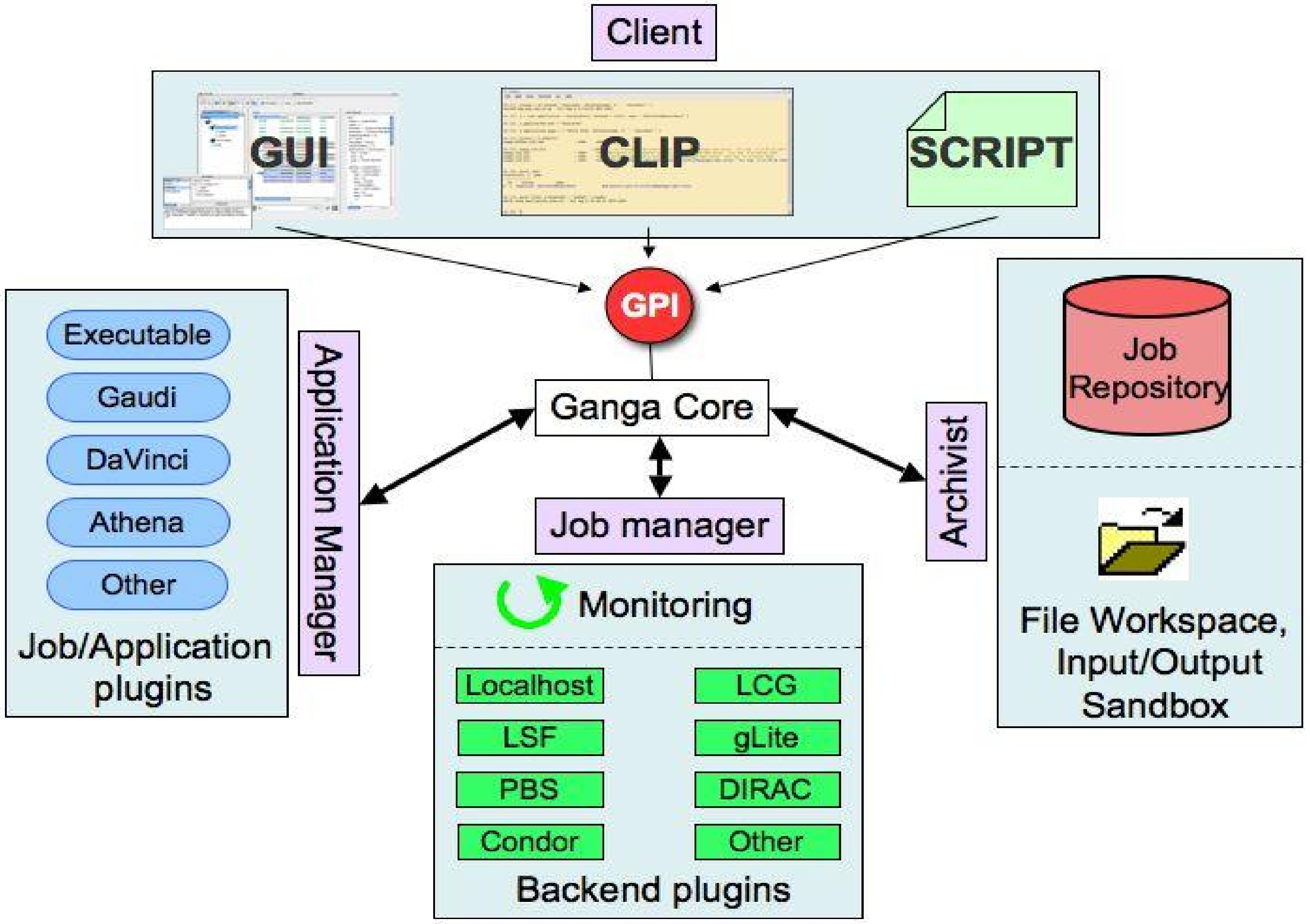
Why Python?

- Development in Python is much faster than in other languages
  - Interpreted language => if all modules are written in Python no need for a special debugger
  - Excellent exception handling mechanism
  - Is the dynamic typing really a disadvantage?
- Python allows running Ganga interactively or as a script in the batch mode
  - Extra modules can be loaded at run time => no complicated procedure is required (import …)
  - As the scripting language Python is much more clear and convenient than bash, tcsh, etc => more potential users

Architecture

- Ganga can be viewed as an abstraction layer for job submission and manipulation
- Ganga Core performs most common tasks. It is represented by 4 main components:
  - Client
  - Application Manager
  - Job Manager
  - Job Repository and File Workspace
- Plugin components provide application- and backend-specific functionality
- All components are linked together and communicate via the Ganga Public Interface (GPI)

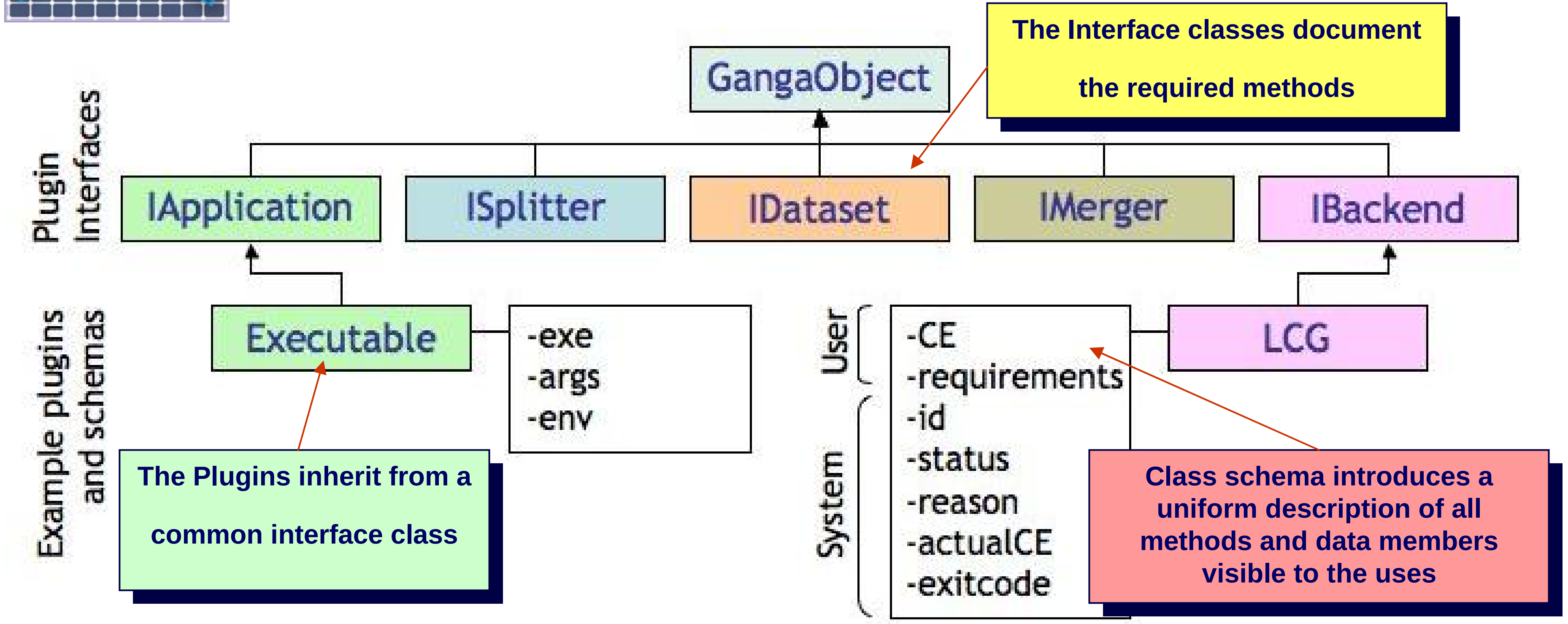
Architecture



The Core Functionality

- The Client controls setting up environment and Ganga initialization. It has GUI and command line (CLIP) interfaces and may run user’s scripts
- The Application Manager describes the application to be run, the user code to be executed, the values to be assigned to any configurable parameters, and the data to be processed => provides configuration object which contains backend-independent information.
- The Job Manager takes care of any job splitting, packages up required user code, submit jobs to the backend, monitors job progress, and retrieves output files when jobs complete => takes as input the configuration object prepared by the Application Manager and calls the application runtime handler responsible for the backend-specific part of the application configuration.
- Job Repository and File Workspace provide persistency for jobs and their input/output sandboxes => Ganga can be restarted having access to the same information. If necessary this information can be searched for particular jobs according to job metadata which is also stored in the Job Repository.

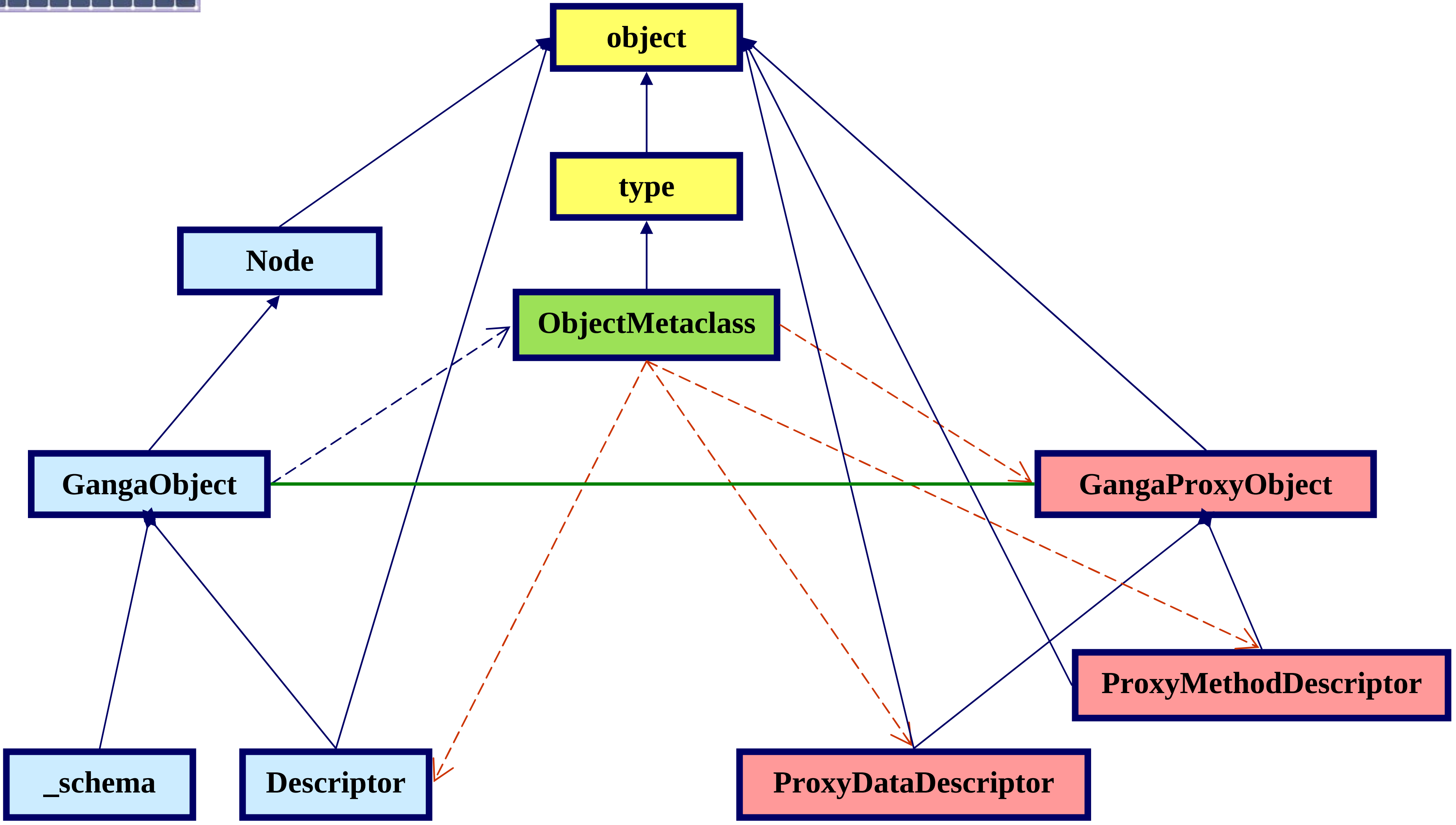
Plugins



The Interface classes document the required methods
GangaObject
Plugin Interfaces
IApplication
ISplitter
IDataset
IMerger
IBackend
Example plugins and schemas
Executable
-exe
-args
-env
User
System
-CE
-requirements
-id
-status
-reason
-actualCE
-exitcode
LCG
The Plugins inherit from a common interface class
Class schema introduces a uniform description of all methods and data members visible to the uses

- The plugin modules represent different types of applications and backends.
- These modules are controlled correspondingly by the Application and Job Manager.
- Such a modular design allows the functionality to be extended in an easy way to suit particular needs of the experiments

The Class Diagram



The Ganga Objects

- All Ganga classes accessible in GPI are derived from the Python “object” class, i.e. they are the new Python classes
- Plugin classes have to be derived from the common base class -- GangaObject
- GangaObject class introduces a uniform description (schema) of all class data members and methods to be visible in GPI.
  - GUI and CLIP representations of the plugin classes can be build automatically
  - Ganga developer can add a new plugin without special knowledge of the GUI and CLIP frameworks

```
_schema = Schema(Version(2,0), {
        'exe' : SimpleItem(defvalue=None, doc='A path (string) or a File object specifying an executable.‘),
        'args' : SimpleItem(defvalue=[],sequence=1, doc= "List of arguments for the executable.“ ),
        'env' : SimpleItem(defvalue={}, doc= 'Environment‘ )
        } )
```

- GangaObject class allows user default values for plugin properties to be set via a configuration file

The Hierarchy of Ganga Objects

- Ganga objects are linked together in a tree like structure, e.g. job object contains application and backend objects, the application object contains options and output objects, etc. => GangaObject class is a subclass of the Node class
  - The special methods __getstate__ , __setstate__ , __copy__ , and __deepcopy__ are aimed to resolve the problem of cyclic references and make the copies faster
  - The key functionality is provided by the the __new__ method

```
def __deepcopy__ (self, memo = None):
        cls = type(self)
        obj = super(cls, cls).__new__(cls)
        dict = self.__getstate__()
        for n in dict:
            dict[n] = copy.deepcopy(dict[n],memo)
        obj.__setstate__(dict)
        return obj
```

The Metaclass and Descriptors

- In the GPI updating an attribute of a Ganga object with a deep copy of value rather than the value itself appears to be more user friendly
  - j1.application = j2.application; j1.application is not j2.application
- We use the descriptor classes to implement this feature
  - Descriptor is a new style class
  - Must have __get__, __set__, __delete__ methods
  - m.d <==> m.d.__get__(m, type(m)) if d is a descriptor, and m is the owner
  - m.d = v <==> m.d.__set__(m, v)
  - It is better than __setattr__ because only the attributes associated with the descriptors get the special treatment
- Descriptors are inserted into the class definition automatically according to the class schema at the time of class construction
  - The metaclass of a class controls this process
  - If a class is derived from another class than the base class metaclass becomes a metaclass of the derived class <==> Only the GangaObject class needs a special metaclass
  - We derive the metaclass class from the type class

M&D Implementation

The Descriptor

```
class Descriptor(object):
    def __get__ (self, obj, cls):
    ……………………………………………………………………
```

The metaclass

```
class ObjectMetaclass(type):
   _descriptor = Descriptor
   def __init__(cls, name, bases, dict):
      super(ObjectMetaclass, cls).__init__(name, bases, dict)
      ……………………………………………………………………

      for attr, item in cls._schema.allItems():
          setattr(cls, attr, cls._descriptor(attr, item))
   ……………………………………………………………………
```

The GangaObject declaring a metaclass

```
class GangaObject(Node):
   __metaclass__ = ObjectMetaclass
    ……………………………………………………………………
```

The Ganga ProxyObjects

- We need more control over the object attributes and methods exposed to the user in the CLIP or Ganga GUI
  - Some attributes are declared “protected” in the schema of GangaObject class <==> user can only read them
  - Some attributes are declared “hidden” <==> user can’t see them
  - Some methods should be available only to the developers but not to the users
  - Job objects presented to the user have to have a build-in automatic persistency mechanism missing at the developers level
  - Objects may have customizable methods, e.g. __repr__
- GangaProxyObject class provides the solution
  - Every subclass of GangaObject has its proxy counterpart class derived from GangaProxyObject
  - The GangaProxyObject classes are created automatically by the GangaObject metaclass according to the class schema
  - It has its own Data and Method Descriptors

The GangaProxyObject class

The proxy class factory

```
# create a new GPI class for a given ganga (plugin) class
def GPIProxyClassFactory(name, pluginclass):
    ……………………………………………………………………….
    return type(name, (GPIProxyObject,), d)
```

The metaclass

```
class ObjectMetaclass(type):
   _descriptor = Descriptor
   def __init__(cls, name, bases, dict):
       super(ObjectMetaclass, cls).__init__(name, bases, dict)
       ……………………………………………………………………….
       # produce a GPI class (proxy)
       proxyClass = GPIProxyClassFactory(name,cls)
       ……………………………………………………………………….
       # store generated proxy class
       cls._proxyClass = proxyClass
```

Serialization

```
#Ganga# File created by Ganga - Mon Jun 26 20:42:31 20
#Ganga#
#Ganga# Object properties may be freely edited before r
#Ganga#
#Ganga# Lines beginning #Ganga# are used to divide ob
#Ganga# and must not be deleted

#Ganga# Job object (category: jobs)
Job (
 name = '' ,
 outputsandbox = [] ,
 splitter = None ,
 inputsandbox = [ ] ,
 application = Executable (
   exe = 'echo' ,
   env = {} ,
   args = ['Hello World']
   ) ,
 inputdata = None ,
 backend = Local (
   ) )
```

An example of the serialized Job object

- The Serialization is aimed to help sharing Ganga jobs among different users
- The streamer (export function) converts Ganga objects into strings using the schemas of corresponding GangaObject classes:
  - The string is a valid Python expression
  - It is human readable, and can be edited if required
  - If the string is evaluated (with eval()) the GPI object is created
  - The created object is equivalent to the original one except that all non-transient attributes are set to the default values

Persistency

The persistency is provided by the the JobRepository package :

Job object → Repository → Job object

- The API provides methods to commit, checkout, and update jobs in the analogy with the cvs repository
- It also provides special methods for making selections and getting summary data without the need to check out the entire jobs
  - The Job object is saved together with some metadata
- It supports bulk operations in order to optimise communication with the underlying database
  - A special treatment is done for the split job which are kept in a tree
- The same API has "local" and "remote" implementations
  - Both implementations rely on the metadata database interface provided by the ARDA project at CERN

The Remote Repository

- The remote implementation of the ARDA interface is an external code
  - Only Python modules are required on the client side
  - The Server side supports various types of database (mysql, postgres, etc)
  - ARDA native communication protocol proves to be simple and reliable
  - Supports user authentication and authorisation based on the Grid proxy certificates

The Local Repository

- The local implementation of the ARDA interface is a joint effort of the Ganga and ARDA developers
  - 100% Python code, no need for the external modules => 100% portable
  - The underlying lightweight database does not require a server => easy to use and maintain
  - The information are kept in files, which can be stored on the distributed file systems like afs
  - Locking mechanism provides transaction safety in the multi-process and multithreaded environment
  - Shows high scalability and performance => may have some potential interest outside of Ganga

Configuration

- Almost of aspects of Ganga behaviour are configurable, e.g.:
  - GUI/IPython/Console
  - Local/Remote repository
  - list of plugins (applications, backends,...)
  - properties of plugins, e.g. Virtual Organization on the Grid
  - logging level, monitoring polling rate, ...
- The configuration system is based on ConfigParser and INI files:
  - consists of configuration units (defined by developers)

```
config = getConfig(name)
           config['opt'] = value
```

  - configuration unit <=> section in the configuration file
  - configuration may be manipulated at runtime by users:
    - customizable setter hooks provide additional flexibility
- Configuration files can be cascaded:
  - customization at site, group, and user levels

The Help and Logging Systems

- Help:
  - interactive help based on pydoc module
  - the help text is generated from docstrings and schema descriptions
- Logging:
  - enhanced logging module provides integrated logging system
  - the logger's name is automatically created from module/package name when the logger is defined
  - configuration of loggers

```
[Logging]
            Ganga.Core.JobManager = "D
            Ganga.Utility = "WARNING"
```

Section in the configuration file

The User Interfaces

- The Ganga CLIP provides interactive access to the GPI objects either via the native Python shell or the enhanced shell (IPython)

An example of CLIP commands for a job submission

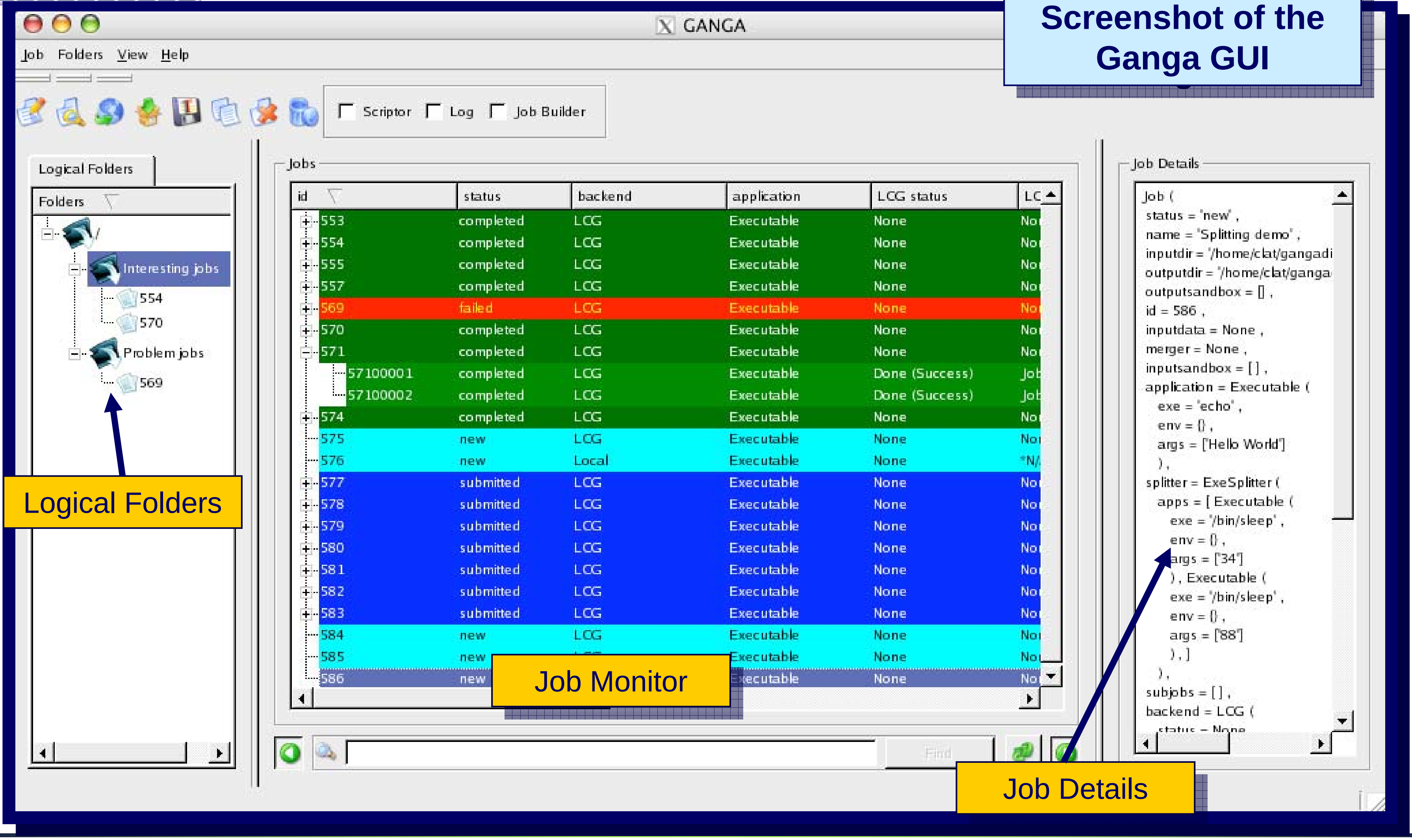
```
# Define an application object
dv = DaVinci(version = 'v12r15', cmt_user_path = '~/public/cmt', optsfile = 'myopts.opts')
# Define a dataset
dataLFN = LHCbDataset(files=[
'LFN:/lhcb/production/DC04/v2/00980000/DST/Presel_00980000_00001212.dst' ,
'LFN:/lhcb/production/DC04/v2/00980000/DST/Presel_00980000_00001248.dst'])
# Put application, backend, dataset and splitting strategy together in a job
j = Job(application=dv, backend=Dirac(), inputdata=dataLFN, splitter=SplitByFiles())
# Submit your complete analysis to the Grid.
j.submit()
```

- The GUI interface is built on the top of the GPI using the PyQt graphics libraries
  - Most of the panels are build dynamically using the widget description from the class schema

```
# preferences for the GUI...
_GUIPrefs = [ { 'attribute' : 'id' }, { 'attribute' : 'status' },
              { 'attribute' : 'inputsandbox‘, 'widget' : 'FileOrString_List‘, 'displayLevel' : 1 },
              { 'attribute' : 'inputdata' },{ 'attribute' : 'outputsandbox' }]
```

The User Interfaces

Screenshot of the Ganga GUI



Jobs

id	status	backend	application	LCG status	LC
553	completed	LCG	Executable	None	No
554	completed	LCG	Executable	None	No
555	completed	LCG	Executable	None	No
557	completed	LCG	Executable	None	No
569	failed	LCG	Executable	None	No
570	completed	LCG	Executable	None	No
571	completed	LCG	Executable	None	No
57100001	completed	LCG	Executable	Done (Success)	Job
57100002	completed	LCG	Executable	Done (Success)	Job
574	completed	LCG	Executable	None	No
575	new	LCG	Executable	None	No
576	new	Local	Executable	None	*N/
577	submitted	LCG	Executable	None	No
578	submitted	LCG	Executable	None	No
579	submitted	LCG	Executable	None	No
580	submitted	LCG	Executable	None	No
581	submitted	LCG	Executable	None	No
582	submitted	LCG	Executable	None	No
583	submitted	LCG	Executable	None	No
584	new	LCG	Executable	None	No
585	new	[illegible]	Executable	None	No
586	new	[illegible]	xecutable	None	No

Job Details

```
Job (
 status = 'new' ,
 name = 'Splitting demo' ,
 inputdir = '/home/clat/gangadi
 outputdir = '/home/clat/ganga
 outputsandbox = [] ,
 id = 586 ,
 inputdata = None ,
 merger = None ,
 inputsandbox = [ ] ,
 application = Executable (
   exe = 'echo' ,
   env = {} ,
   args = ['Hello World']
   ) ,
 splitter = ExeSplitter (
   apps = [ Executable (
     exe = '/bin/sleep' ,
     env = {} ,
     args = ['34']
     ) , Executable (
     exe = '/bin/sleep' ,
     env = {} ,
     args = ['88']
     ) , ]
   ) ,
 subjobs = [ ] ,
 backend = LCG (
   status = None
```

The Programmatic Interface

- Ganga may be embedded into an existing Python framework (this option has already been investigated/used by Atlas).
- There are two aspects of Ganga programmatic use:
  - passive API (i.e. like any other library)
  - active runtime services (such as monitoring threads)

Conclusions

- Python allows us to build a user-friendly Grid user interface, which bas already been tried out by close to 100 people from ATLAS and LHCb, and growing number of physicists use it routinely for running analyses on the Grid, with considerable success.
- Although specifically addressing the needs of ATLAS and LHCb for running applications performing large-scale data processing on today's Grid systems, Ganga has a component architecture that can be easily adjusted for future Grid evolutions and for other user communities.
- Some software solutions we use in Ganga may be of general interest for the Python developers working in different areas.

Further information at *ganga.web.cern.ch*

Easy download and installation !!!