Lizard

A Flexible and Modular Data Analysis Tool using Abstract Types

Andreas Pfeiffer
CERN IT/API
andreas.pfeiffer@cern.ch

Outline

- Introduction and Motivation
- Architecture overview and design criteria
- Basic Types and their Functionality
- Present status and planning
- Summary

Introduction and Motivation

- 1995 "Libraries for Hep Computing" - LHC++ project initiated
  - aim: full, modular replacement of CERNLIB (lib and tools)
  - concentrated on using (industrial and de facto) standards (STL, OpenGL, ...) and commercial components (e.g., NAG_C, OpenInventor, IrisExplorer)
- 1998 First iteration on physics data analysis tool in LHC++ context
  - data driven approach (based on IRIS Explorer)
  - GUI based, not command line driven

Introduction and Motivation (II)

- Request to create new physics analysis tool (September 99)
  - new requirements defined together with experiments
  - identified categories/components and Abstract Types
- Presentation at HepVis'99 workshop
  - triggered creation of working group (AIDA)
  - together with developers of other tools (HippoDraw, Iguana, JAS, OpenScientist)
  - aiming at interoperability

USDP-like approach to create tool

- Start with OO analysis
  - collection of user requirements
- OO design phase
  - define categories and classes
  - find patterns
- Create prototype
  - considered “throw away”
- get feedback from users
- Iterate, iterate, iterate ...

User requirements for a physics analysis tool

- Ease of use ("like PAW")
- Foresee customization/integration
  - e.g., use persistency/messaging/... from experiment
- Framework used will not be exposed/imposed
  - needs to be compatible with experiment's framework
- Plan for extensions
  - "code for now, design for the future"
- Maximize flexibility/interoperability

Architecture Overview

- Maximize flexibility and re-use
  - Abstract Interfaces allow each component to develop independently
    - not bound to a specific implementation of any component ("plugin" style)
  - re-use of existing packages to implement components reduces start-up time significantly
- Identify and use patterns - avoid anti-patterns
  - learn from other people's experiences/failures

Patterns identified so far

- AbstractFactory
  - for object creation
- Strategy
  - Factory is strategy for manager
- Facade (controller)
  - promotes weak coupling of classes
- Visitor
  - extend functionality of base classes

Architectural issue: Components (I)

- Identify components by functionality
  - not by "historic use"
- Emphasize separation of different aspects for each component
  - example: Histogram
    - statistical entity (density distribution)
    - view of a "collection of data points" (which can be a density distribution but also a detector efficiency)
    - command to manipulate/store/plot/fit/...
  - "User's view" is different from "implementor's view"

Architectural issue: Components (II)

- Minimize coupling between components
  - allows components to be developed/maintained independent from each other
- UserInterface as a separate component
  - by definition couples to most of the other components (*Facade* pattern)
  - promotes weak coupling between the other components
  - interfaces to scripting and/or GUI
- Distributed computing

Architectural issue: Abstract Types

- Define Abstract Types for each component
  - components use other components only through their Abstract Interface
    - weak coupling, as no implementation specific features are used (avoids the (in-)famous "N x M problem" of strongly coupled systems)
- Use "AbstractFactory" pattern for creation of Abstract Types
  - concrete implementation of Factory and Types are in dynamically loaded shared library ("plugin")
  - "Manager" class to load specific implementation

Architectural issue: Scripting

- Typical use of scripting is quite different from reconstruction
  - history "go back to where I was before"
  - repetition - with "modifiable parameters"
  - ....
- Scripting language is an interface to the UserInterface component
  - SWIG allows flexibility to choose amongst several scripting languages
    - Python, Perl, ...

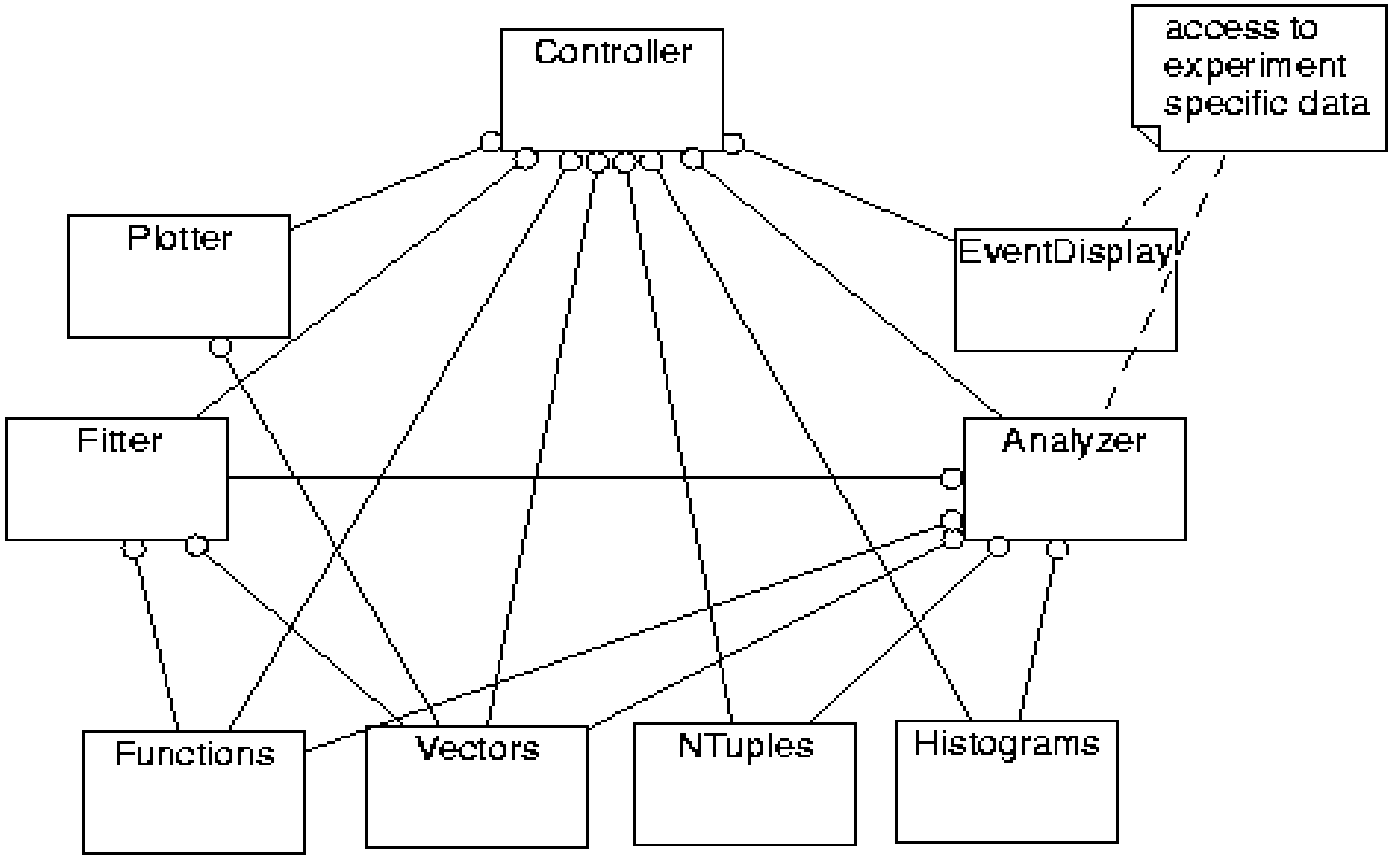
Architectural issue: Distributed Computing

- Very complex field where several basic questions are not yet answered
  - data to processes (classical farm approach) ?
  - processes to data (CORBA, JAVA RMI) ?
- Easy if "hidden" in implementation of component(s)
  - e.g., distributed filling of Histograms in *Analyzer*
  - Abstract Interface hides the complexity, no change in tool(s) needed

Categories identified

- Basic functionality
  - Histograms, Vectors, Ntuples, Functions
- Visualization
  - Plotter, EventDisplay (access to experiment's data)
- Analysis
  - Fitter, Analyzer (access to experiment's data)
- User Interface
  - Controller (defines (part of) the commands/shortcuts)
- Misc
  - Messages (between components)

Categories and dependencies



Basic Types and their Functionality in Lizard (I)

- VectorOfPoints - collection of DataPoints
  - "measured value" at n-dim space-point (with errors)
    - (x,eX-,eX+(,*y,eY-,eY+*,...), value, eVal-, eVal+)
  - behaves like the *content* of a PAW-like histogram
  - all arithmetic operations, shifting and scaling
  - used by *Fitter* and *Plotter*
  - can be created from *Histogram*
  - can be read/written from/to ASCII file
    - XML format foreseen

Basic Types and their Functionality in Lizard (II)

- Histogram - purely *statistical* entity
  - representation of density distribution + summary
  - "operations" taken over by *VectorOfPoints*
  - with "*Annotation*" to keep non-statistical information
    - units of axes, ID, cuts used in creation, ...
- NTuple - access to disk resident data
  - similar in functionality to PAW RWN
  - based on *GenericTags* (HETag package)
- Functions - mainly for visualization and fitting
  - not yet defined

Basic Types and their Functionality in Lizard (III)

- Fitter - uses *VectorOfPoints*
  - Abstract Interface in preparation
- Analyzer - access to experiment specific data and libraries
  - on-the-fly compilation and dynamic loading
  - customizable makefile to access experiment s/w
  - simple interface at present
- EventDisplay - visualization of experiment specific data
  - will make use of *HepRep* interfaces (Joe Perl, SLAC)

Basic Types and their Functionality in Lizard (IV)

- Plotter - 2-D visualisation of *VectorOfPoints*
  - based on *Qt* libraries (Troll Tech, Norway, www.troll.no)
    - "A better motif than Motif. Better foundation classes than MFC" (according to Troll Tech)
    - KDE foundation
    - Available on most UNIX systems and Windows
    - No runtime fee. Developer license on Windows only
  - added *Qplotter* package
    - implements HIGZ/HPLOT functionality
  - 3-D graphics by using OpenInventor/OpenGL through Qt extensions

Basic Types and their Functionality in Lizard (V)

- Controller - interface to the User Interface
  - its *methods* define the complete set of *commands*
  - *uses* the other categories to *implement* them
  - methods/commands can be called by scripting language and/or GUI

Scripting in Lizard

- Using public domain tools for scripting
  - SWIG to (semi-) automatically create connection to chosen scripting language
  - Python - OO scripting, no “strange $!%-variables”
    - other scripting languages possible
  - Can be enhanced and/or replaced by a GUI
    - scripting window within GUI application

Example script (fitting)


```
# book and fill a histogram
h1=hm.create1D(20, ”gauss-fit” ,50,0.,50.)
for i in range(1.,50.):
   h1.fill(i,100.*exp(-(i-25.)**2/100.)+random.gauss(0,10))

# prepare to fit the histo h1
fit=Fitter()                                  # create a new fitter:
fit.setModel(“G”)                             # set the model
fit.addParameter(“amp”,100)                   # define (and init) parameters
fit.addParameter(“mean”,h1.mean())            # ... for these the order
fit.addParameter(“rms”,h1.rms())              # ... is relevant
fit.chi2Fit1D(h1)                             # perform fit on histogram
fit.plot(pl)      # plot vector with fitted curve using Plotter pl
```

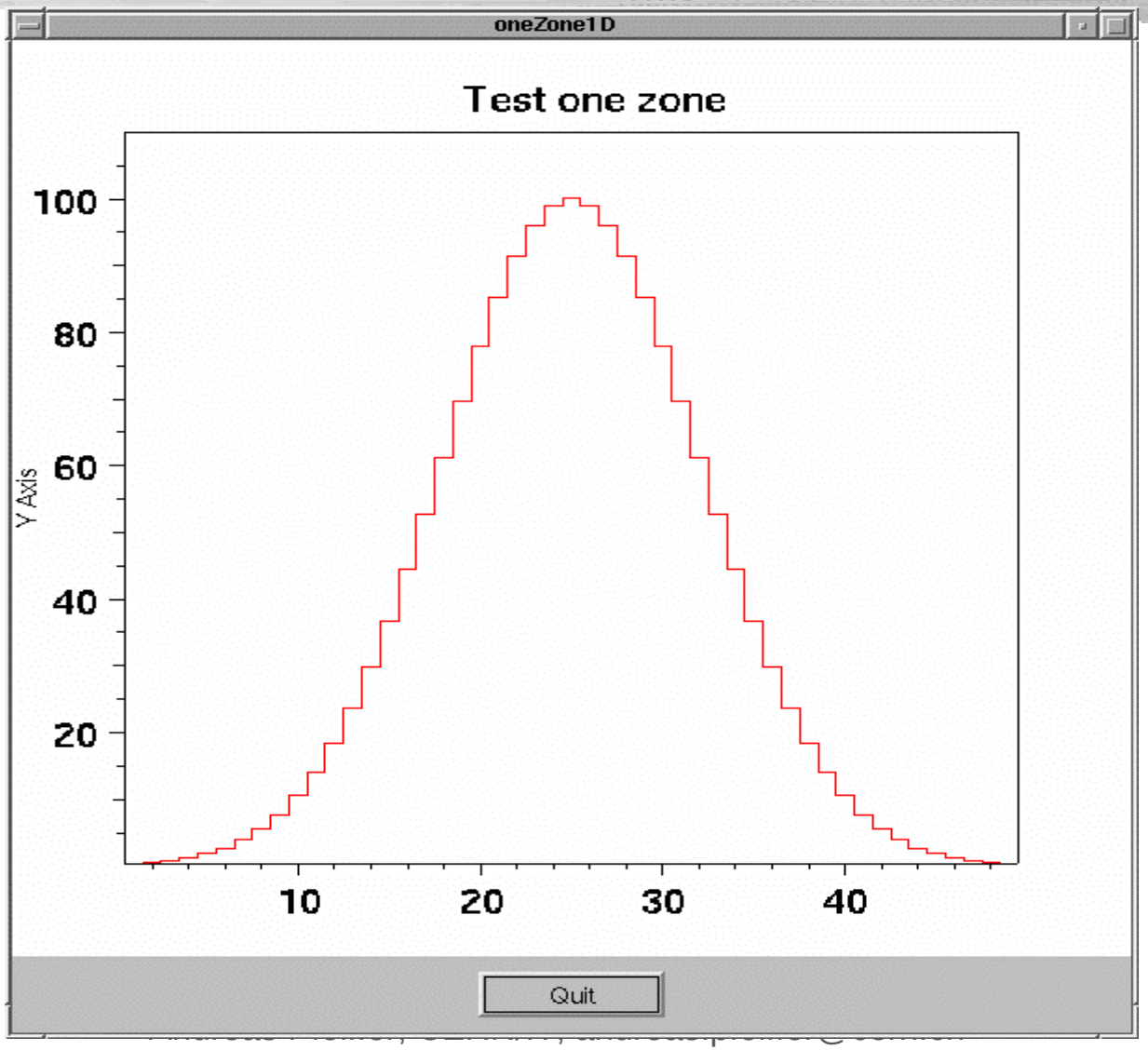
Example script (ntuple)


```
# get list of names of all tuples from tuplemanager
ntm.listTuples()
nt1=ntm.findNtuple("track-fit")          # retrieve tuple by name
nt1.list()                               # print names and types of attributes
# create 1D histos to project into ("direct plotting" will come)
 h1=hm.create1D(10, "thePreN" ,100,0.,500.)
h2=hm.create1D(20, "thePostN for preN>300" ,100,0.,500.)

# project the attribute "thePreN" into histo h1 without  cut ("")
ntm.project1D("track-fit", h1, "" , "thePreN")

# project the attribute "thePreN" with cut ("thePreN>200")
ntm.project1D("track-fit", h2, "thePreN>200", "thePostN")
```

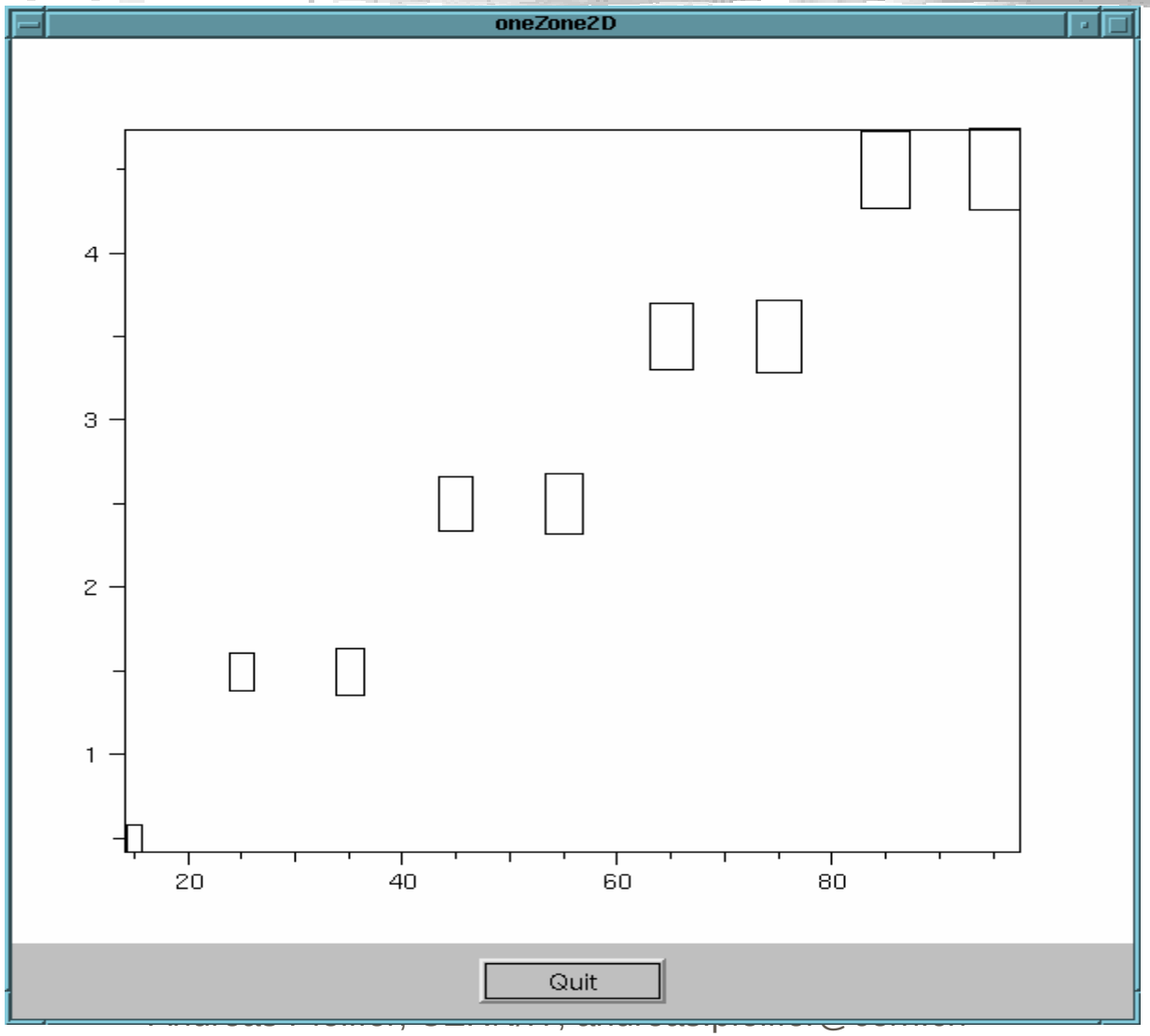
Example Analyzer


```
void myAnalyzer::doIt(HistoManager *hm, NtupleManager *ntm)
{
  // Create a new histogram using the HistoManager instance
  Histogram1D *first =
          hm->create1D(200,"from analyzer",100,0.,100.);
  // ... and fill it with some double gaussian
  double w;
  for (int i=0; i < first->nBins(); i++) {
    w = exp(-(i-50.)*(i-50.)/10.) + exp(-(i-20.)*(i-20.)/100.);
    first->fill(i,w);
  }
}
```

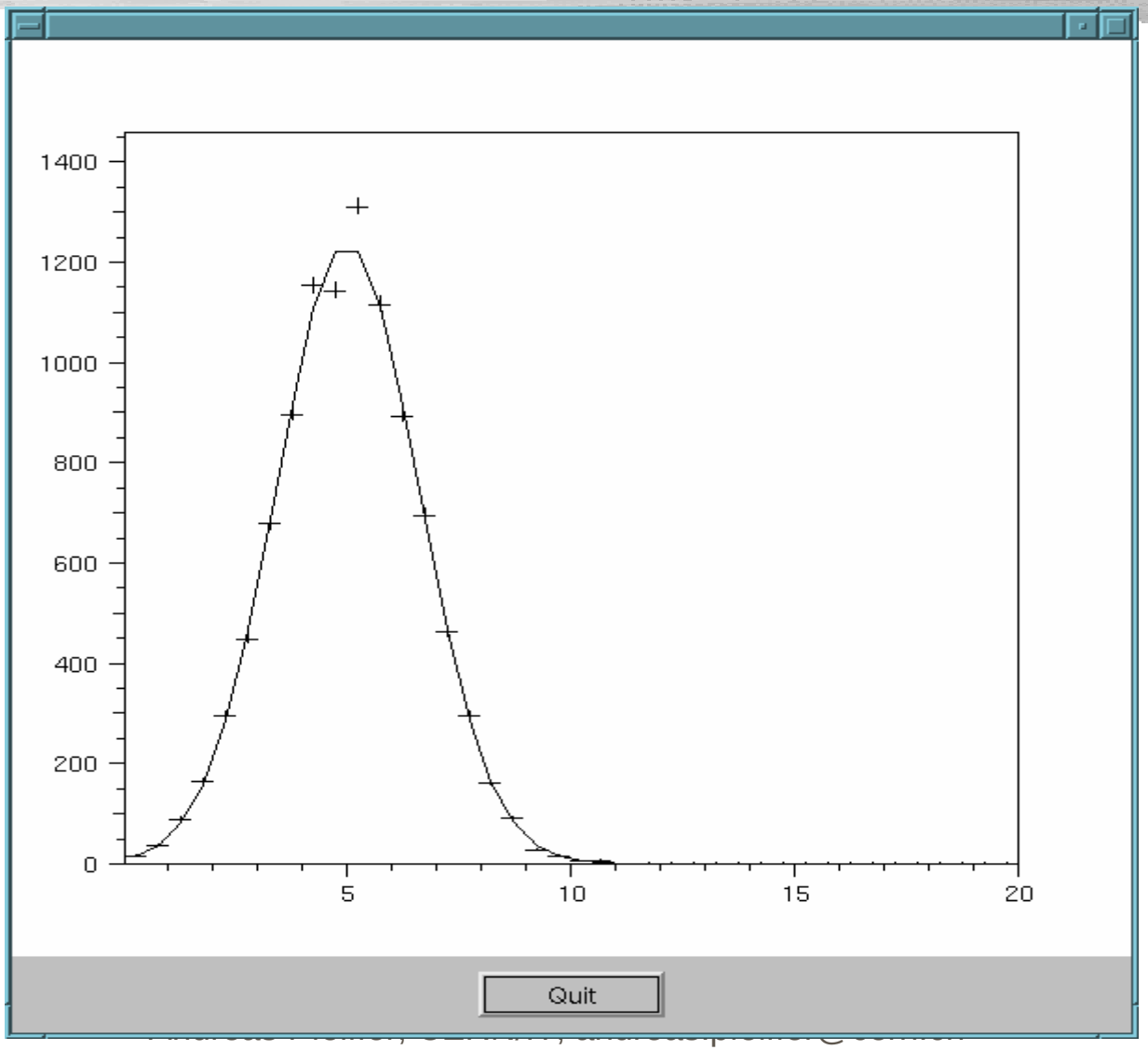
Example: 1D histogram



Example: 2D histogram



Example: histo+fit



Status and Plans

- First prototype (limited functionality) available since CHEP-2000
  - feedback from users on Python scripting IF
  - not based on Abstract Types
  - wrapping existing LHC++ libraries
- Re-design started in April 2000
- Next prototype planned for October 2000
  - still with restricted functionality
  - "alpha" release foreseen end July 2000
- Fully functional version in April 2001
  - "PAW-like" functionality

Possible Future Enhancements

- Access to other implementations of components
  - HBOOK histograms and ntuples (RWN)
  - OpenScientist ?
  - ROOT histograms ?
- Adding other "scripting" languages
  - Perl ? , cint ?
- Communication with Java tools/packages
  - JAS
  - WIRED ?

Summary

- The architecture of Lizard shows some important items for a flexible and modular data analysis tool:
  - use of Abstract Interfaces for the components
  - weak coupling between components
- The present work is based on LHC++ components and Python scripting language (through SWIG)
  - maximizes re-use
- Major criteria are flexibility, extensibility and interoperability with other tools/packages
- **http://wwwinfo.cern.ch/asd/lhc++/Lizard/index.html**