Frameworks and Common Packages

LASER Summer School on Software Engineering 2013, Elba Island, Italy
Pere Mato/CERN
Frameworks

- Applications
- Event
- Det Desc.
- Calib.
- Simulation
- Data Mngmt.
- Analysis
- Core Libraries
- non-HEP specific software packages

Experiment Framework
A software framework is an abstraction in which common code providing generic functionality can be selectively overridden or specialized by user code providing specific functionality.

A software framework is similar to software libraries in that they are reusable abstractions of code wrapped in a well-defined API.

Typically the framework “calls” the user provided adaptations for specific functionality.

Is the realization of a software architecture and facilitates software re-use.

In practice: a skeleton of an application into which developers plug in their code and provides most of the common functionality.
Not a Single Framework

- A single **Framework** does for fit everywhere
- Each software domain provides its specialized framework
  - E.g. a GUI framework based on signal-slot can be used to build GUI application
  - E.g. a Simulation framework can be used to simulate the passage of particles through matter
- Real complex applications are made typically with a collaboration of frameworks
  - The main challenge is often that each framework ‘thinks’ is the main one
Framework Benefits

- **Common vocabulary**, better specifications of what needs to be done, better understanding of the system
  - E.g. GUI framework: windows, widgets, slots, events, etc.

- Low coupling between concurrent developments
  - Smooth integration
  - Organization of the development

- Robustness, resilient to change (change-tolerant)
  - Provides some isolation between developer’s code and services (OS, networking, I/O, etc.)

- Fostering code re-use
  - E.g. generic widgets can be re-used many times to build GUI applications
Gaudi Architecture and Framework

An Example of Framework for HEP Applications
Principal Design Choices

- Separation between “data” and “algorithms”
- Three basic categories of “data”
  - event data, detector data, statistical data
- Separation between “transient” and “persistent” representations of data
- Data store-centered (“whiteboard”) architectural style
- “User code” encapsulated in only few specific places
- Well defined component “interfaces” with plug-in capabilities

The Gaudi Framework is used by ATLAS and LHCb among others
Algorithms and Data Flows

The meat of the applications is coded by physicists in terms of Algorithms

- They transform raw input event data into processed data
  - e.g. from digits -> hits -> tracks -> jets -> etc

- Algorithms solely interact with the Event Data Store (“whiteboard”) to get input data and put the results
  - Agnostic to the actual “producer” and “consumer” of the data
  - Complete data-flows are programmed by the integrator of the application (e.g. Reconstruction, Trigger, etc.)
Definition of Terms

* Algorithm
  * Atomic data processing unit (visible & controlled by framework)

* Algorithm Tool
  * Class called by the Algorithm or another Tool to perform a specific function (private and public)

* Data Object
  * Atomic data unit (visible and managed by transient data store)

* Transient Data Store
  * Central service and repository for data objects (data location, life cycle, load on demand, …)

* Services
  * Globally available software components providing framework functionality

* Data Converter
  * Provides explicit/implicit conversion from/to persistent data format to/from transient data

* Properties
  * Control and data parameters for Algorithms and Services
Algorithm

IAlgorithm
- initialize()
- execute()
- finalize()

IProperty
- setProperty()
- getProperty()

• Users write Concrete Algorithms
• It is called once per physics event
• Implements three methods in addition to the constructor and destructor
  * initialize(), execute(), finalize()
Interfaces

ApplicationMgr

EventDataSvc

DetectorDataSvc

HistogramSvc

MessageSvc

ParticlePropertySvc

Concrete Algorithm

ISvcLocator

IDataProviderSvc

IDataProviderSvc

IHistogramSvc

IMessageSvc

IParticlePropertySvc

IAlgorithm

IProperty

Obj_A

Obj_B
VCR Interface Model

- Each interface is specialized in a domain.
- Interfaces are independent of concrete implementations.
- You can mix devices from several constructors.
- Application built by composing.
- Standardizing on the interfaces gives us big leverage.
Plug-ins

- Program extensions to provide a certain, usually very specific function "on demand"
- Applications/frameworks support plug-ins for many reasons (in HEP)
  - to enable third-party developers to create capabilities to extend an application
  - to support features yet unforeseen
  - to reduce the size of the basic application
Configuring the Application

- Each Framework component can be configured by a set of ‘properties’ (name/ value pairs)
- In total thousands of parameters need to be specified to fully configure a complex HEP application
- Using Python to facilitate the task
  - Build-in type checking
Interactivity and Scripting

- Interactivity and scripting are essential use cases for any HEP framework
  - Scripts for rapid prototyping and trying new ideas
  - Testing frameworks
  - GUI applications

- A convenient way to achieve it is to provide bindings to a scripting language such as Python (or a C++ interpreter)
  - Once this is done the rest comes automatically
From class definitions (.h files) a “dictionary” library is produced
- Description of the class
- "stub" functions to class methods

Absolutely non-intrusive

The PyROOT module does the adaptation between Python objects and C++ objects in a generic way
- It works for any dictionary
Summary: Frameworks

- All HEP experiments have developed Software Frameworks
  - General architecture of any event processing applications (simulation, trigger, reconstruction, analysis, etc.)
  - To achieve coherency and to facilitate software re-use
  - Hide technical details to the end-user Physicists
  - Help the Physicists to focus on their physics algorithms

- Applications are developed by customizing the Framework
  - By the “composition” of elemental Algorithms to form complete applications
  - Using third-party components wherever possible and configuring them

- ALICE: AliROOT; ATLAS+LHCb: Athena/Gaudi; CMS: CMSSW
Common Packages (examples)
ROOT

- ROOT is an extensive data handling and analysis framework
  - Efficient object data store scaling from KB’s to PB’s
  - C++ and Python interpreter
  - Extensive 2D+3D scientific data visualization capabilities
  - Extensive set of data fitting, modeling, and analysis methods
  - Complete set of GUI widgets
  - Classes for threading, shared memory, networking, etc.
  - Parallel version of analysis engine runs on clusters and multi-core
  - Fully cross-platform, Unix/Linux, Mac OS X and Windows
  - 2.7 million lines of C++, building into more than 100 shared libs

- Development started in 1995

- Licensed under the LGPL

http://root.cern.ch
ROOT Interpreter

- ROOT is shipped with an C/C++ interpreter, CINT
  - C++ not trivial to interpret and not foreseen in the language standard!
- Provides interactive shell
- Can interpret “macros” (not compiled programs)
  - Rapid prototyping possible
- ROOT provides also Python bindings (PyROOT), which are very popular among physicists
- Starting from ROOT 6, the interpreter will be Cling (based on LLVM/Clang)
ROOT in Numbers

- Ever increasing number of users
  - 6800 forum members, 68750 posts, 1300 on mailing list
  - Used by basically all HEP experiments and beyond

As of today 177 PB of LHC data stored in ROOT format

ALICE: 30PB, ATLAS: 55PB, CMS: 85PB, LHCb: 7PB
ROOT in Plots
ROOT Object Persistency

- Scalable, efficient, machine independent format
- Orthogonal to object model
  - Persistency does not dictate object model
- Based on object serialization to a buffer
- Automatic schema evolution (backward and forward compatibility)
- Object versioning
- Compression
- Easily tunable granularity and clustering
- Remote access
  - HTTP, HDFS, Amazon S3, CloudFront and Google Storage
- Self describing file format (stores reflection information)

ROOT I/O is used to store all LHC data (actually all HEP data)
Object Containers - TTree

- Special container for very large number of objects of the same type (events)
  - Minimum amount of overhead per entry

- Objects can be clustered per sub object or even per single attribute (clusters are called branches)

- Each branch can be read individually
  - A branch is a column

Physicists perform final data analysis processing large TTrees (more later)
ROOT in Javascript

- Provide ROOT file access entirely locally in a browser without any prior ROOT installation on the server or client
- ROOT files are self describing
EVE Event Display
PROOF - The Parallel Query Engine

- A system for running ROOT queries in parallel on a large number of distributed computers or many-core machines
- PROOF is designed to be a transparent, scalable and adaptable extension of the local interactive ROOT analysis session
- For optimal CPU load it needs fast data access (SSD, disk, network) as queries are often I/O bound
- The packetizer is the heart of the system
  - It runs on the client/master and hands out work to the workers
  - It takes data locality and storage type into account
  - Tries to avoid storage device overload
  - It makes sure all workers end at the same time
**PROOF User Interface: TSelector**

* Users write specializations of TSelector class to perform data analysis
  * Results are typically N-tuples, histograms, etc.
* Only 3 methods have to be defined
  * **Begin**: initializations (histograms, global variables, etc...)
  * **Process**: event selection and treatment. It is called for each entry in the ROOT tree
  * **Terminate**: end of the analysis (global calculations, write results in a file, etc...)
Various Flavors of PROOF

- PROOF-Lite (optimized for single many-core machines)
  - Zero configuration setup (no config files and no daemons)
  - Workers are processes and not threads for added robustness
  - Once your analysis runs on PROOF Lite it will also run on PROOF
  - Works with exactly the same user code as PROOF

- Dedicated PROOF Analysis Facilities (multi-user)
  - Cluster of dedicated physical nodes
    - E.g. department cluster, experiment dedicated farm
  - Some local storage, sandboxing, basic scheduling, basic monitoring

- PROOF on Demand (single-user)
  - Create a temporary dedicated PROOF cluster on batch resources (Grid or Cloud)
  - Uses an resource management system to start daemons
  - Each user gets a private cluster
  - Sandboxing, daemon in single-user mode (robustness)
Geant4

- A software (C++) toolkit for the Monte Carlo simulation of the passage of particles through matter
  - ‘propagates’ particles through geometrical structures of materials, including magnetic field
  - simulates processes the particles undergo
    - creates secondary particles
    - decays particles
  - calculates the deposited energy along the trajectories and allows to store the information for further processing (‘hits’)

Simulated Higgs event in CMS
Monte Carlo simulation

• What is Monte Carlo
  • Throwing random numbers
    • to calculate integrals
    • to pick among possible choices

• Why Monte Carlo
  • Because Einstein was wrong: God does throw dice!
  • Quantum mechanics: amplitudes => probabilities
    • Anything that possibly can happen, will! (but more or less often)
  • Want to generate events in as much detail as Mother Nature
    • get average and fluctuations right
    • make random choices, as in nature
Why do we need it?

- To design and to construct the devices
  - to get what we are looking for
  - to be able to see the relevant ‘events’
  - not to get what we do not want
  - radiation damage, etc

- To operate the devices
  - to adjust and to tune the apparatus
    - medical physics: dose calculation, etc

- To understand the experimental results
  - to compare to the existing theories and models
  - to make discoveries
Needed to make Discoveries

- We need to understand the detector to do physics
  - imperfections
  - calibrations
- We need to know what to expect to
  - verify existing models
  - find new physics
Geant4 Application

- Geometry implementation
- User ‘actions’
- Primary particles
- Magnetic field
- Geant4 kernel
- Physics models
- ‘Hits’ (energy deposition)
- ‘MC truth’ (generated secondaries)
- Visualisation
Geometry and Materials

- How to implement (efficiently) this in your computer program?
- You need ‘bricks’
  - ‘solids’, ‘shapes’
  - you need to position them
  - you want to ‘reuse’ as much as possible the same ‘templates’
- Database of Materials
  - National Institute of Standards (NIST)
- Magnetic Fields
  - numerical integration of the equation of motion (Runge-Kutta method)
Physics...

- What happens to particles in matter?
- We need to implement the physics we know
  - each possible physics process provides the "interaction length" compared with distance to next boundary
    - the smallest wins
  - generating a "final state" and secondaries tracks

- Electromagnetic
  - gammas and charged particles

- Hadronic
  - neutrons, mesons (K, π), muons, ...
Software Re-use:
Integrating Technologies
When Frameworks are not Possible

- At occasions you need to build a software system/application made of independently developed components
  - Using existing class libraries
  - They cannot be re-done using a single ‘framework’
  - Building adaptation layers are not always possible and effective

- Examples
  - Integrating MC generators in ROOT
  - Performing ROOT I/O on Geant4 Applications
Software Integration Elements

✤ Dictionaries
  ✤ Dictionaries provide meta data information (reflection) to allow introspection and interaction of objects in a generic manner

✤ Scripting languages
  ✤ Interpreted languages are ideal for rapid prototyping
  ✤ They allow integration of independently developed software modules (software bus)
  ✤ Standardizing on CINT and Python scripting languages

✤ Component model and plugin management
  ✤ Modeling the application as components with well defined interfaces
  ✤ Loading the required functionality at runtime
Strategic role of C++ reflexion

- Object I/O
- Scripting (CINT, Python)
- Plug-in management etc.

ROOT

Root meta C++

Python

CINT

Reflex

Reflex DS

ROOT

XDictcint.so

rootcint -cint

rootcint -reflex

rootcint -gccxml

X.h

Monday, September 9, 13
Python <-> C++ Interoperation

- The bulk of code for the new HEP experiments is written in C++
  - Still some portions of FORTRAN with plans to migrate
  - Java and other languages almost non-existent
- Need Python bindings to C++ code
  - Hand-written (C-API) or generated
  - Requires taking care of:
    - Object, parameter conversions
    - Memory management
    - C++ function overloading
    - C++ templates
    - Inheritance and function callbacks
    - etc.
Python as Software “Bus”

- Very rich set of Python standard modules
- Several GUI toolkits
- Gateways to other frameworks
- Very rich set of LHC modules
- Very rich set of generic modules

- EDG API
- PVSS
- XML
- Database
- GUI
- shell
- JPE
- PyROOT
- GaudiPython
- math
- Java Classes
- ROOT Classes
- Gaudi Framework
- ROOT Classes
Language Symbiosis

- Getting the best from C++ (number crunching) and Python (rapid prototyping)
- Scientists are able to express sophisticated data analysis algorithms using high level Python constructs
  - E.g. D* analysis in LHCb
  - Fits in a single page
- Very little performance impact
- Very easy to make changes

```python
from bendermodule import *
class Dstar(Algo):
  def analyse( self ) :
    self.select ( tag='K-', cuts=('K-' ==ID)&(PT>1*GeV) )
    self.select ( tag='pi+',cuts=('pi+'==ID)&(P >3*GeV) )
    dmass = ABSDM("D0") < 30 * MeV

    for D0 in self.loop ( formula='K- pi+', pid='D0' ) :
      if ( VCHI2(D0) < 4 ) &  dmass( D0 ) : D0.save('D0')

    tup = self.nTuple ( title = "D*+ N-Tuple " )

    for Dst in self.loop ( formula='D0 pi+', pid='D*(2010)+' ) :
      dm = M(Dst) – M1(Dst)
      h1 = self.plot( title = "Delta mass for D*++" ,
      value = dm , low=130 , high=170 )
      tup.column( name = ‘M’ , value = M(Dst) / GeV )
      tup.column( name = ‘DM’ , value = dm / GeV )
      tup.column( name = ‘p’ , value = P (Dst) / GeV )
      tup.column( name = ‘pt’ , value = PT(Dst) / GeV )
      tup.write ()

    return SUCCESS
```
Take Away Messages

* Introduced software frameworks used in HEP and their hierarchy
* Used GAUDI framework as an example of HEP event data processing framework
  * The main design criteria
  * Introduction to few of the main concepts and functionalities
* Two examples of common software packages: ROOT and Geant4
  * Basically used by any HEP experiment
  * Examples of software re-use
* Software integration elements
  * Dictionaries, Plugins and the role of Python