C++ Design Concepts for Scientific Data Processing And Analysis

Jakub T. Moscicki
CERN IT/API
jakub.moscicki@cern.ch
Outline

- Architecture: managing software complexity
  - applications
  - libraries and frameworks
    - components
- C++ design: specific techniques
  - templates
  - reference vs value semantics
- Software process
  - metrics and testing
Architecture
Application Architecture

- Prediction and Risk Analysis (PRERIA)
  - geophysical data processing and statistical analysis
  - commercial product born of scientific research (1996)
  - tremor and rock-burst prediction for coal mines
Subsystems

• Reducing and managing complexity
  • GUI subsystem
    • MVC architecture (Qt, MFC, OWL)
      • document / view (observer pattern)
  • Domain model subsystem
    • data model
    • processing model (numerical algorithms, ...)

• Goals:
  • Portability
    • decouple domain model from particular GUI system
  • Maintainability
    • minimize interaction points between subsystems
PRERIA object model

Document

Data model

View

Processing model (missing)
Library and framework architecture

- Analysis for Physics Experiments (Anaphe, 1995-)
- CERN-IT effort for CERNLIB/PAW replacement
- Lizard: interactive analysis tool (2000-)

```python
#: exec("ntup.py")
Explorables present:
Charml
#: h1=h1.retrieveHist1D(10)
#: xFit = h1("G")
### Some of the bins have zero errors. They will be ignored in the chi-square fit.
### Use setDefaultError(double) to redefine zero errors.
### Performing a chi-square fit.
```
Interactive Data Analysis

- **Aim:** "OO replacement for PAW"
  - analysis of "ntuple-like data" ("Tags", "Ntuples", □)
  - visualisation of data (Histograms, scatter-plot, "Vectors")
  - fitting of histograms (and other data)
  - access to experiment specific data/code
- Maximize **flexibility** and re-use
- Foresee customization/integration
  - allow use from within experiment's s/w
- Plan for extensions
  - "code for now, design for the future"
## Anaphe Components

<table>
<thead>
<tr>
<th>Category</th>
<th>Tools/Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Analysis</td>
<td>Lizard - AIDA</td>
</tr>
<tr>
<td>Custom graphics (2-D)</td>
<td>Qt - Qplotter</td>
</tr>
<tr>
<td>Basic graphics (3-D)</td>
<td>OpenInventor - OpenGL</td>
</tr>
<tr>
<td>Basic math</td>
<td>NAG C</td>
</tr>
<tr>
<td>HEP foundation</td>
<td>CLHEP</td>
</tr>
<tr>
<td>HEP math</td>
<td>FML - Gemini - CLHEP</td>
</tr>
<tr>
<td>Histograms</td>
<td>HTL</td>
</tr>
<tr>
<td>Database</td>
<td>HepODBMS</td>
</tr>
<tr>
<td>Persistency</td>
<td>ODMG/Objectivity DB</td>
</tr>
<tr>
<td>C++</td>
<td>Standard Libraries</td>
</tr>
</tbody>
</table>
Insulation and layering

- Components are **individual C++ class libraries**.
  - Easy replacing one part without throwing away everything
  - Alternative implementations interchangeable
    - HepDBMS versus HBOOK Ntuples
    - Nag C minimizers versus MINUIT
  - Easy customization to match experiment specific needs
  - Runtime flexibility
  - Small number of **well-defined package dependencies**
    - No cyclic dependencies
    - Components may be used individually
- Insulate components through **Abstract Interfaces**
  - "wrapper" layer to implement Interfaces in terms of existing libs
- Design patterns
  - **Factory, Facade**
Lizard Internals: Interfaces
Distributed Ntupple Analysis

- R&D project (2001-2002)
  - transparent remote analysis for HEP
  - parallel computing on GRID farms
- Possible users:
  - Ntupple analysis
  - GEANT 4 (simulation)
    - Event generation
    - Track level (harder)
  - Custom applications
    - Code distribution
    - Environment replication
    - Binary dependencies (shlibs)
Topology

- **I/O intensive** rather than CPU intensive
- Fast DB access from cluster
- Slow network from user to cluster
- Very small amount of data exchanged between the tasks in comparison to "input" data
Deployment of Distributed Components

- Layering: abstract middlewares
- Dynamic application loading
- Plugin components
User Analysis Applications

IUserNTupleAnalysis <<interface>>

DefaultUserNTupleAnalysis

End User NTuple Analysis

this abstraction level presents a simplified interface (US_EVT) facade pattern

NTuple Analysis Generic App

Any Analysis App

this abstraction level deals with worker, master, integrator

Abstract Task Model <<interfaces>>

Data Exchange Protocol

Runtime distributed framework

Analysis Applications
C++ Design Techniques
Data Exchange Protocol (1)

- XDR concept in C++
  - Specify data format
    - Type and order of data fields
- Data messages
  - Sender and receiver agree on the format
  - Message is send as opaque object (any)
  - C++ type may be different at each side
- Interfaces with flexible data types
  - E.g. store list of identifiers (unknown type)
Data Exchange Protocol (2)

```cpp
class A : public DXP::DataObject {

public:
    DXP::String name; // predefined fundamental types
    DXP::Long index;
    DXP::SequenceDataObject<DXP::plain_Double> ratio;
    B b;            // nested complex object

    A(DXP::DataObject *parent) : DXP::DataObject(parent),
        name(this), index(this), ratio(this), B(this) {}  
}
```

CERN IT/API, Jakub.Moscicki@cern.ch
Data Exchange Protocol (3)

- External streaming supported, e.g
  - Serialize as CORBA::byte_sequence
  - Serialize to XML (ascii string)
  - Visitor pattern: new formats easy

- Handles
  - Opaque objects (any)
  - Typed objects: safe "casts"

```cpp
DXP::TDataObject<A> a1, a2; // explicit format
DXP::AnyDataObject x = a1; // opaque object
a2 = x;
if(a1.isValid()) // "cast successful"
```
Plugin components

- Plugin Online HTML Doc
Templates

- Histogram Library
  - HistOOgrams Prototype (1997)
  - HTL (1998-1999)
    - statistics separated from display and persistency
    - compile-time flexibility, run-time efficiency
- Histogram is a template:
  - template <class T_Life, class T_Bin, class T_Partition> T_Histo1D
  - bins and partitions may vary independently
Templates

- predefined common histo types:
  - typedef `T_Histo_1D< T_Lifetime<WeightedBin>, WeightedBin, EvenPartition> T_Histo_1D_WED`

- persistent histograms in Objy model:
  - Initial data model (HistOOgrams)

- Template model: physical clustering (by...
Templates

- Traits for automatic template composition

```
CompositePartition2D<P1,P2>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>e1</td>
<td>P1</td>
</tr>
<tr>
<td>e2</td>
<td>P1</td>
</tr>
</tbody>
</table>

binID map_bin(P1::e_type x, P2::e_type double y)
```

```
FixedLengthPartition1D
typedef double e_type;
void map_bin(double x)

VariableLengthPartition1D
typedef double e_type;
void map_bin(double x)
```
Template metaprogramming

- Recursive template instantiation
- Compilation may simulate Turing machine :-)
- Special (numerical) optimizations
  - blitz++

```cpp
template <unsigned int N> class Histogram
{
    public:
        Histogram() {}  
        Histogram<N-1> projection() { return Histogram<N-1>(); }  
    
};
Histogram<10> h;
h.projection().out();
```
Reference semantics

• smart pointers convey precise semantics
  • `std::auto_ptr<T>`
    • ownership transfer
    • exception safe
    • no-need for naming convention
  • `boost::shared_ptr<T>`
    • sharing objects
    • proper resource cleanup
  • `boost::scoped_ptr<T>`
    • "resource acquisition is initialization"
    • automatic resource cleanup
    • exception safe
Value semantics

• valuable when we can afford it (small objects)
  • easier than reference semantics
  • still may use const & for optimization

• DataXML (2001)
  • simple DOM-like interface
  • based on STL vector and map
  • using entirely value semantics
    • extremaly easy implementation of the object tree
    • matching value semantics of STL containers
Bits and pieces for Software Process
Software analysis tools

- Big tools (e.g. ROSE, Together)
  - Heavy to use
  - Expensive
- Most of the CASE utilization is
  - Reverse engineering
  - Generating diagrams/graphs etc
- Small analysis tools are nicer!
  - perform single task
  - may be combined to perform more complex task
Package level analysis

- Include dependencies (doxygen)
Package level analysis

- Inheritance and composition (doxygen)
Inter package analysis

Anaphe 3.6.5 Direct Dependencies: FML by Purpose
CERN IT/API, Jakub.Moscicki@cern.ch
Software quality control

- Using tools for testing/checking has started
  - Insure++, CodeWizard
- Package dependencies: **Ignominy**
  - Set of perl and shell scripts by Lassi Tuura (CMS)
  - Ignominy **scans**
    - Make dependency data produced by the compilers (*.d files)
    - Source code for #includes (resolved against the ones actually seen)
    - Shared library dependencies (ldd output)
    - Defined and required symbols (nm output)
  - And **maps**
    - Source code and binaries into packages
    - #include dependencies into package dependencies
    - Unresolved/defined symbols into package dependencies

**Ignominy**: dishonour, disgrace, shame; infamy; the condition of being in disgrace, etc.  
(Oxford English Dictionary)

Thanks to Andreas Pfeiffer (API)
Ignominy Analysis of Anaphe

- Distribution of tools and utilities for LHC era physics
  - Combination of commercial, free and HEP software
  - Claims to be a toolkit
- Seems to live up to its toolkit claims
  - Good work on modularity
  - Clean design is evident in many places
  - Dependency diagrams often split naturally into functional units

Thanks to Lassi Tuura (CMS)
Thanks to Andreas Pfeiffer (API)

CERN IT/API, Jakub.Moscicki@cern.ch
## Package Metrics

<table>
<thead>
<tr>
<th>Project</th>
<th>Release</th>
<th>Packages</th>
<th>Average # of direct dependencies</th>
<th>Cycles (Packages Involved)</th>
<th># of levels</th>
<th>ACD*</th>
<th>CCD*</th>
<th>NCCD*</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anaphe</td>
<td>3.6.1</td>
<td>31</td>
<td>2.6</td>
<td>--</td>
<td>8</td>
<td>5.4</td>
<td>167</td>
<td>1.3</td>
<td>630/170k</td>
</tr>
<tr>
<td>ATLAS</td>
<td>1.3.2</td>
<td>230</td>
<td>6.3</td>
<td>2 (92)</td>
<td>96</td>
<td>70</td>
<td>16211</td>
<td>10</td>
<td>1350k</td>
</tr>
<tr>
<td></td>
<td>1.3.7</td>
<td>236</td>
<td>7.0</td>
<td>2 (92)</td>
<td>97</td>
<td>77</td>
<td>18263</td>
<td>11</td>
<td>1350k</td>
</tr>
<tr>
<td>CMS/ORCA</td>
<td>4.6.0</td>
<td>199</td>
<td>7.4</td>
<td>7 (22)</td>
<td>35</td>
<td>24</td>
<td>4815</td>
<td>3.6</td>
<td>420k</td>
</tr>
<tr>
<td>CMS/COBRA</td>
<td>5.2.0</td>
<td>87</td>
<td>6.7</td>
<td>4 (10)</td>
<td>19</td>
<td>15</td>
<td>1312</td>
<td>2.7</td>
<td>180k</td>
</tr>
<tr>
<td>CMS/IGUANA</td>
<td>2.4.2</td>
<td>35</td>
<td>3.9</td>
<td>--</td>
<td>6</td>
<td>5.0</td>
<td>174</td>
<td>1.2</td>
<td>150/38k</td>
</tr>
<tr>
<td>Geant4</td>
<td>4.3.2</td>
<td>108</td>
<td>7.0</td>
<td>3 (12)</td>
<td>21</td>
<td>16</td>
<td>1765</td>
<td>2.8</td>
<td>680k</td>
</tr>
<tr>
<td>ROOT</td>
<td>2.25/05</td>
<td>30</td>
<td>6.4</td>
<td>1 (19)</td>
<td>22</td>
<td>19</td>
<td>580</td>
<td>4.7</td>
<td>660k</td>
</tr>
</tbody>
</table>

*) John Lakos, Large-Scale C++ Programming

- **Size** = total amount of source code (not normalised across projects!)
- **ACD** = average component dependency (~ libraries linked in)
- **CCD** = sum of single-package component dependencies over whole release
  - Indicates testing/integration cost
- **NCCD** = Measure of CCD compared to a balanced binary tree
  - A good toolkit’s NCCD will be close to 1.0
  - < 1.0: structure is flatter than a binary tree (= independent packages)
  - > 1.0: structure is more strongly coupled (vertical or cyclic)
- **Aim**: NCCD ~ 1 for given software/functionality

Thanks to Lassi Tuura (CMS)  
Thanks to Andreas Pfeiffer (API)
Metrics: NCCD vs ACD

Av. Component Deps (Fraction of Packages)

Toolkits & Frameworks

Thanks to Lassi Tuura (CMS)
Metrics: NCCD vs Cycles

Fraction of Packages in Cycle

Toolkits & Frameworks

ATLAS
ORCA
G4
COBRA
Anaphe
IGUANA

NCCD

0% 10% 20% 30% 40% 50% 60% 70%

0
10
20
30
40
50
60
70
80
90
100
110
120

Thanks to Lassi Tuura (CMS)
Testing and QA

- Testing is an art :-)  
  - unit testing  
  - integration testing  
  - regression testing  
  - black-box, white-box, gray-box testing

- Quality Assurance  
  - programming by contract  
    - assertions  
    - pre- and post-conditions  
    - invariants  
  - tools  
    - insure++,...
Testing (the art of)

- Gemini Test Suite (2001)
  - regression tests reusing existing example programs
  - common programming infrastructure for testing
  - reference data for fitting
    - NIST/ITL StRD Statistical Reference Data (on the web)
- Negative compilation test
  - check that expected compile-time error was detected
    - particularly useful for C++ implicit conversions
    - part of the test suite for DXP
  - boost.org also does it
    - compile, link, run tests
    - compile-fail, link-fail, run-fail tests
Summary

- Architecture and design
  - modularity is crucial for long term projects
  - components and subsystems help to achieve it
- C++ design
  - good practices help to maintain good design
    - use the patterns
    - reuse proven code
      - boost.org
  - templates are a powerful tool
- Software Process
  - metrics may help to assess quality of the product
  - even testing may be fun :)

CERN IT/API, Jakub.Moscicki@cern.ch