# Experience with OO Programming at BaBar

Andreas Meyer

Lawrence Berkeley Laboratory

- BaBar Experiment at SLAC
- BaBar Data Acquisition and Storage
- BaBar Reconstruction and Analysis Software
- Aspects of OO
- Why OO/C++?
- Conclusions

H1 Software Meeting, June 16, 1999

abmeyer lbl.gov

# B-Factory: BaBar at PEP-II

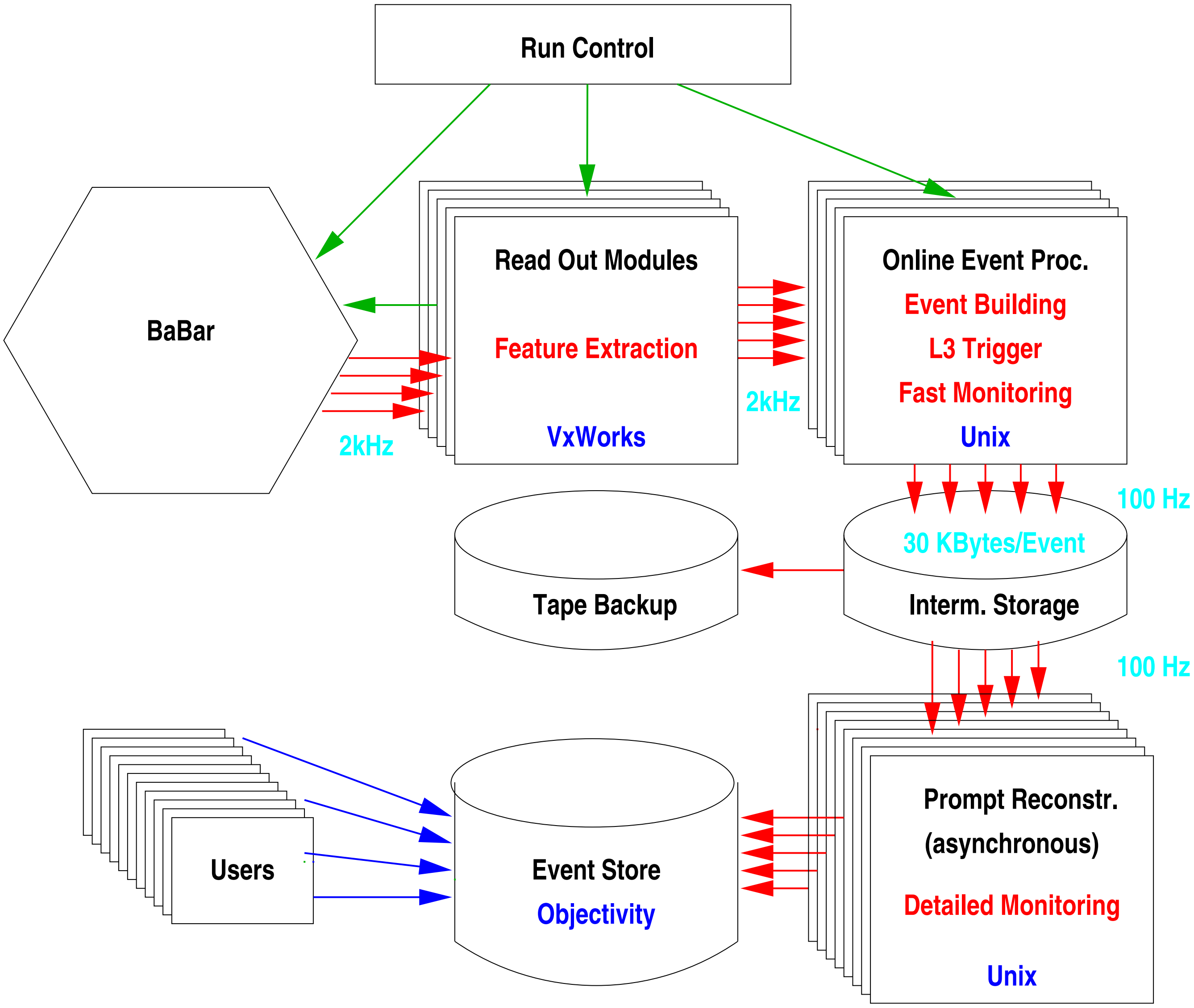
- Measure CP violation in $B\bar{B}$ System:
  - CKM Matrix, Unitarity Triangle
- B physics in general: lifetimes, oscillations, BR (rare dacays)
- Charm and $\tau$ physics (factory)
- 2-photon-physics and QCD

Golden channel: $B^0 \rightarrow J/\Psi K_S^0$ (BR: 0.5 $\cdot 10^{-3}$)

- $10^8 \Upsilon(4S)$ decays for 1000 observed events.
- PEP-II design luminosity $10^{34}$ cm$^2$s$^{-1}$
- $\Upsilon(4S)$ production rate: $\sim$ 10 Hz
- Recorded event rate: $\sim$ 100 Hz

30 kBytes/event $\rightarrow$ 30 TBytes/year Raw Data

# BaBar Data Acquisition System




30 TBytes/year Raw Data

+ 100 TBytes/year Prompt Reconstruction

+ Simulation and Calibration Data

Factory mode: no global reprocessing

Access to data: within hours after recording

# BaBar Data Store

Storage Hierarchy:

| | | |
|---|---|---|
| SIM | simulated truth | |
| RAW | raw data | $\sim 30$ kB |
| REC | reconstructed | $\sim 100$ kB |
| ESD | event summary data | $\sim 10$ kB |
| AOD | analysis object data | $\sim 1$ kB |
| TAG | event tags | $\sim 0.1$ kB |

Objectivity: Data Management

- independent access to objects $\rightarrow$ fast skimming
- reconstruction: add information $\rightarrow$
  no replication of data necessary
- placement strategies $\rightarrow$
  re-grouping for fast retrieval

Objectivity: User access to data

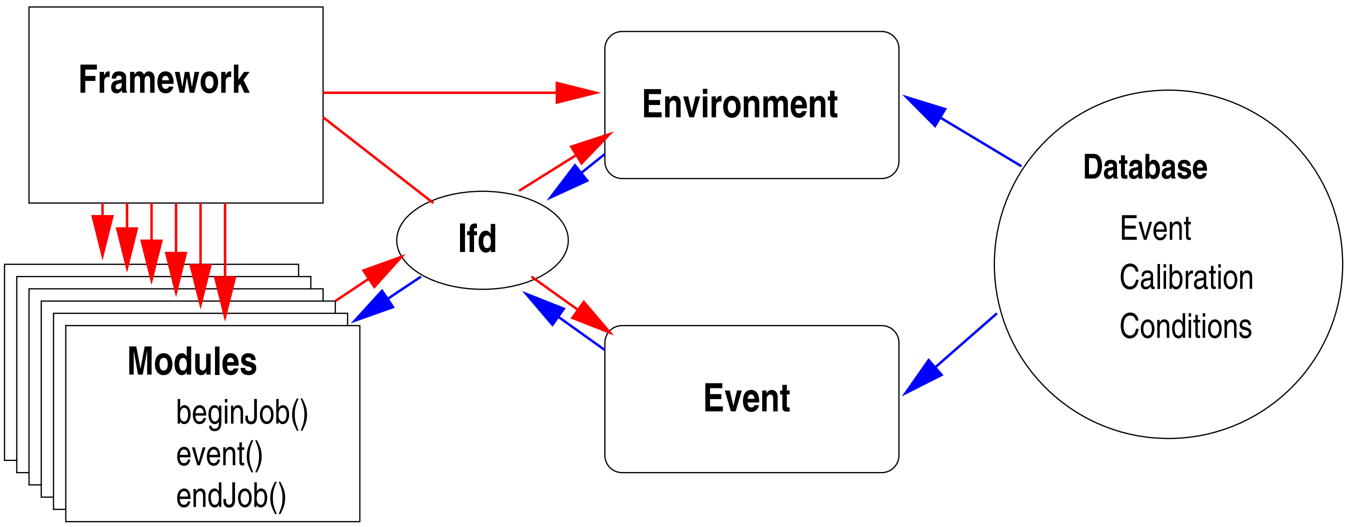
- specify sets of events in 'collections'
- free to add objects, calibrations
- use as ntuple (not yet)

# BaBar Reconstruction Software

Event-by-event processing by sequential execution of Modules = Algorithmic Units:

- coupled: require reasonable order
- replaceable: plug-in capability

All modules have access to all data in event.

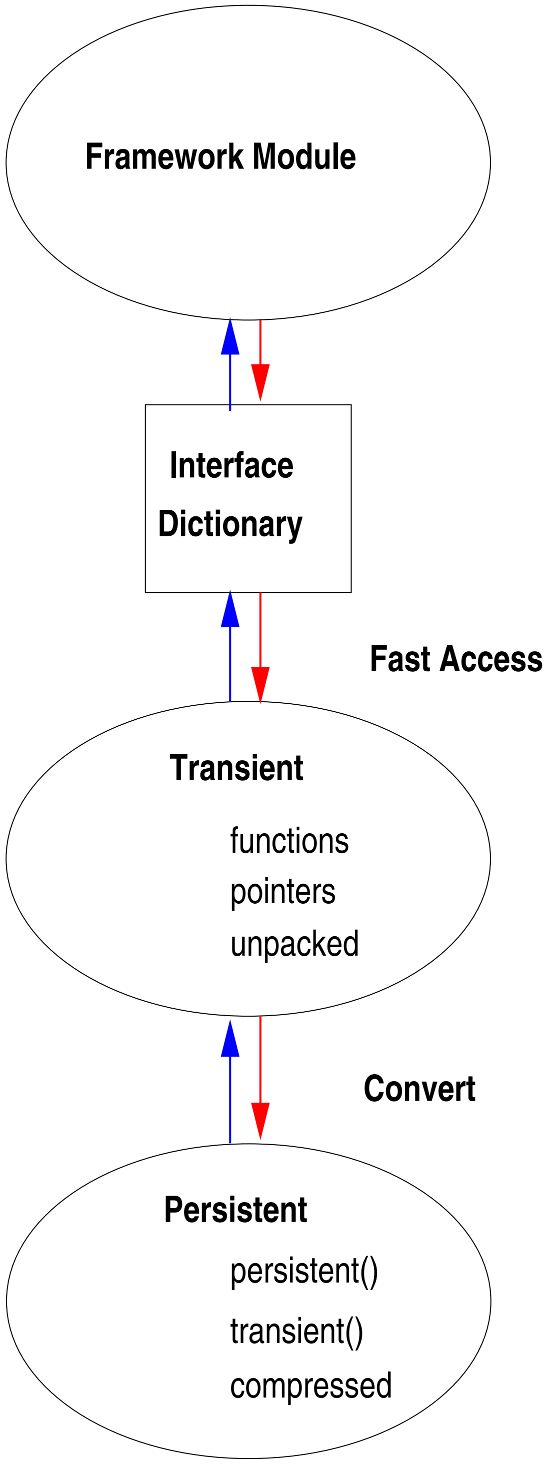



User reconstruction / analysis package 'Beta':

- Default reconstruction (sequences)
- Associations, 4-Vectors, particle ID
- Example analysis

OO methodology to avoid broad interdependencies.

# Interface to Database




Minimize interdependency between:

reconstruction code $\leftrightarrow$ data model $\leftrightarrow$ database

# OO

- Data Encapsulation:

  Functions and data in classes (same file).

  $\rightarrow$ helps localizing places of repair

  data hiding $\rightarrow$ enforces proper access of data

- Inheritance:

  Abstract classes contain set of generic functions.

  1. Modules inherit from basemodule

     $\rightarrow$ have uniquely defined interface with:

     – Execution main() ('Framework')
     – Interface dictionary ('Transient Event')
     – Calibration constants ('Environment')

     $\rightarrow$ just add specific code

  2. Most reco-objects are also abstract classes:

     $\rightarrow$ enforce common interface formats

  Case distinctions do not require if-statements

  – Strong object-type checking

  – Dynamic binding

  $\rightarrow$ add new code, don't touch old code

# Graphics Analysis Tools

1999: OO graphics/analysis tools just arriving.

BaBar: Leave decision to user (at time of analysis).

Abstract interface 'HepTuple'

- HbkTuple $\rightarrow$ Hbook (default in 99)
- DhpTuple $\rightarrow$ DHP
- RooTuple $\rightarrow$ Root

Example: Fast-Monitoring

- Online: Dhp $\rightarrow$ JAS
- Offline: Hbook $\rightarrow$ paw

Downside: HepTuple is 'flat' interface

- Can not pass objects, only create and fill histograms and ntuples of basic types.

Future:

- JAS $\leftrightarrow$ Objectivity
- (Root $\leftrightarrow$ root files)

# BaBar Software Status

- 400 Packages in CVS
- $10^6$ Lines of code
- On-line/L3: test at 1.5 kHz successful
- Prompt Reconstruction: 2s/event (/160 CPUs)
- Simulation (with beam-bg): 15 s/event
- Database: 0.7 sec to write complete event, still problems with 'locks'
- Off-line reconstruction with 'Beta' reading 'Micro-DST': 30 Hz

BaBar code is complete and working
Physics events !!!

Also: first-pass results on
$dE/dx$, $\theta_C$ vs. $p$, calibration, alignment, efficiencies

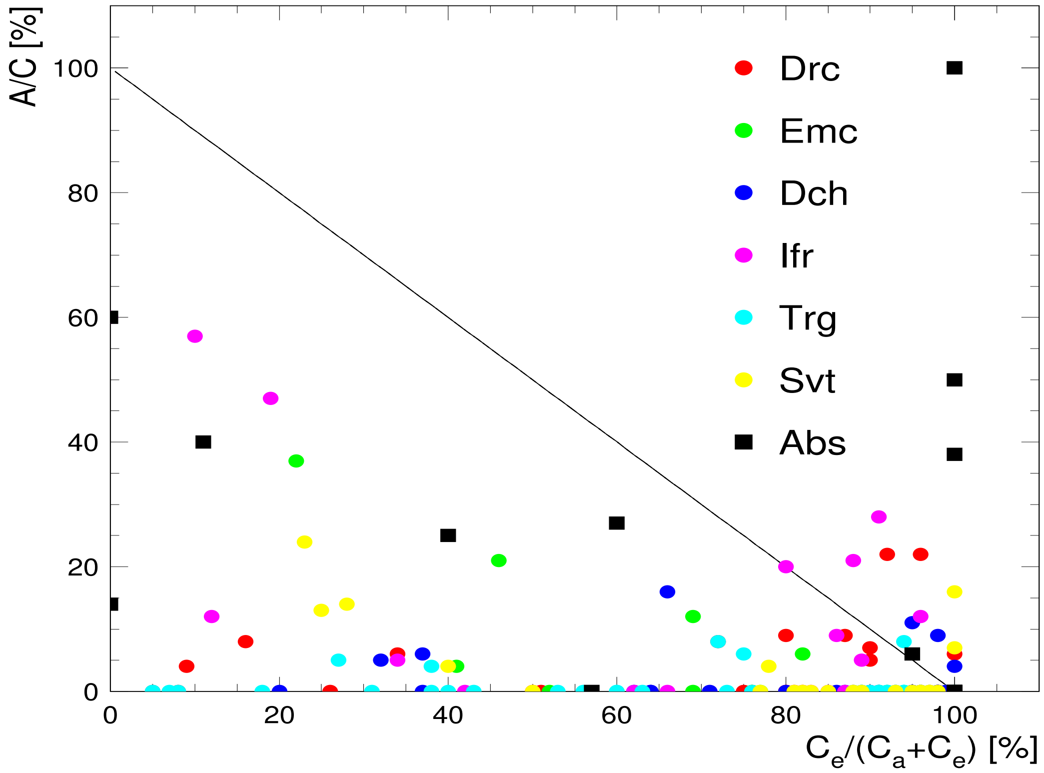
- Speedup in progress: factor 5 gained since january
- Re-analyze, optimize/rewrite slow code

# C++ Code Evaluation

Many tools available (commercial, public domain)

- document, e.g. Light
- analyze, e.g. Rose (reverse engineering)
- generate, e.g. from UML Diagrams (forward engineering)

Example: Quality Metrics of BaBar Packages




$C_e$: # inside classes in package dependent on outside classes

$C_a$: # outside classes dependent on classes inside package

$A/C$: Fraction of abstract classes in package

# Why (moderate) OO with C++ ?

Goals of OO/C++ not surprising ...

- safe access to large & complex data sets
- good stable interfaces

... C++ makes these issues part of the language.

- Allows object-modeling → code potentially easy to read
- Enforces common interfaces ("bring interfaces to the surface")
- True data hiding
- Strong type checking: Add new code without modifying/recompiling old code
- Built-in dynamic memory allocation.
- Language development & tools support by outside world.

Compiler helps with software administration

# Why OO?

Gains:

- More code transparency
- More data safety
- Better extensibility
- Wider sharing of responsibilities possible

Downsides:

- Allows object-modeling $\longrightarrow$ code potentially hard to read.
- Few experienced people.
- Imposes need for design
- Potentially person power intensive
- Language just being standardized (still differences in compilers, bugs in libraries etc.)
- Potentially poor performance

Handle C++ with care!

and do not make full use of the syntax

# Conclusions

"Writing software is becoming a literary effort"

For software projects $> 10^4$ lines OO is necessary

C++ helps enforcing OO policies

BaBar strategy: "Learning while doing":

Moderate OO /C++ approach has been successful,
Software is complete and working, analysis starting

- Impressive: achieved modularity, flexibility, extensibility
- Disappointing: performance (is improving)

C++ is not optimal (but maybe the best we have)

– gain experience with OO

– refine OO paradigm,

– restrict the syntax to the OO useful,

– standardize the OO programming styles

Next-generation OO-language:

Combine design and management advantages of standardized/optimized C++ with performance and robustness advantages of F77/C?

JAVA with execution time optimization ???