Paul Bright-Thomas
Alan Watson
11th May 2000

OO Experiences in BaBar

- BaBar overview and status
- BaBar environment
- BaBar code
- Personal experiences

BaBar overview

- BaBar is high-lumi "B factory" running at $\Upsilon(4s)$
- Asymmetric beam energies of $9\,\text{GeV} + 3.1\,\text{GeV} \Rightarrow$ boosted B's, time-dependent CP measurements
- Detector: 5-layer SVT, Drift chamber, DIRC, CsI EMC, IFR
- Design luminosity of 3.10^{33} $\text{cm}^{-2}\,\text{s}^{-1}$, integrating to $30\,\text{fb}^{-1}$ per year or 30 Million B events
- Online event processing (OEP) $\Rightarrow$ Level-3 trigger $\Rightarrow$ Prompt Reco (OPR)
- Max 200 kHz Level-3 trigger input, 100 Hz reconstructed, 5 Mbyte/s to storage
- ~ 600 collaborators, minority active in physics

BaBar Status

- BaBar up and running since May 1999 — 1 year's data-taking!!!
- Good accelerator start-up $\Rightarrow > 2/3$ design lumi, $100+$ pb^{-1} per day
- DAQ and reconstruction now keeping up, **Minor miracle!**
- Environment and tools put in place early
- Code itself rather turbulent until end 99, development now smoother
- BaBar Objectivity database is already > 130 Tbyte in size
- First results at Osaka from $\sim 10\,\text{fb}^{-1}$ of data — CP result on $\sin 2\beta$ from $B \to J/\psi K^0_s$ flagship channel expected

BaBar environment

- 4 platforms: **Solaris**, **AIX**, **OSF** and **Linux** (HP deprecated)
- Availability of commercial tools limits cross-platform compatibility
- All BaBar SW (incl WWW) under **AFS** — significant reliability problems
- Version management through **CVS** and Software Release Tools (**SRT**)
- Build and test managed with **GNUmake**
- User environment standardised through **HEPiX**
- Job configuration through **Tcl** scripting; **Perl** also widely used for integration
- Central storage in **Objectivity** OO database — separate federations by function, throughput problems improved
- Alternative data access via **Kanga** — KangaROO(T) uses **ROOT** as storage, growing in popularity due to portability
- Good use of **WWW**: 30k pages, wrapped for uniformity, with navigation bars
- **HyperNews** for subscriber announcements and threaded discussions — $\sim$ 200 forums
- **LiGHT** has been used to document some releases, but not maintained
- **Remedy** for problem reporting and tracking
- Several tools for SW quality: **Purify**, **GreatCircle**, **CodeWizard**

BaBar code

- **Structure:**
  - Code organised as 500 **packages**, each with responsible co-ordinator
  - Most BaBar code is **C++**, with **Java** GUI applications, e.g. Java Analysis Studio (JAS)
  - Few **FORTRAN** packages C++-wrapped, e.g. Hbook, Minuit
  - Many packages build on **CLHEP** base classes, deriving for specific usages
  - Several vendor libraries: **RogueWave**, **CORBA**, **STL**
  - High inter-dependence between packages inevitable
- **Management:**
  - Nightly builds of approved package versions **tags**
  - Coherent set of for each package issued as **release** (2-weekly)
  - Online and reconstruction releases are made separately
  - Environment turbulent when packages in development
  - Pace of development has slowed and inclusion of new tags more strictly controlled $\Rightarrow$ better code stability
- **Documentation: poor!**
  - A partially complete, but thorough, tutorial **Workbook** exists, which is useful for new user
  - Specific "user" packages are well-documented, such as **Beta** analysis framework
  - Many packages inadequate documentation, even within code
- **Design and quality:**
  - Little evidence of formal design or automated code generation — growth seems to be "organic"
  - Code quality: biggest problem **memory management**

Personal experiences of OO

- C++ syntax intrinsically evil, but will have to do (for now)
- Key advantages of OO are supposedly:
  - **Good design**
  - **Maintainability**
  - **Efficiency through re-use**
- Reality for BaBar:
  - **Good design** — in places, methodology not obvious
  - **Maintainability** — coupling through interface volatility
  - **Re-use** — yes, but initial *use* problematic without documentation
- Good **base classes** with well-defined **interfaces** can make code very modular
- However, most classes **dependent** on *something* and life is hard when interfaces badly documented or designed
- Learn the tricks — Abstraction, inheritance, patterns
- Mundane tasks tackled by **class libraries** — lists, iterators, matrices, vectors, strings — see **CLHEP**
- Classic **"Design patterns"** helpful — managers, environments, proxies, wrappers, factories, helpers, composites — see Gamma, Helm, Johnson and Vlissides
- Discipline required for memory management — e.g. pass by reference for longevity, pass by pointer for deletion
- **Recommendation:**
  - 30% design
  - 10% coding
  - 20% testing
  - 40% documentation
- Jump in ASAP — learn while there's less of it in ATLAS!!!