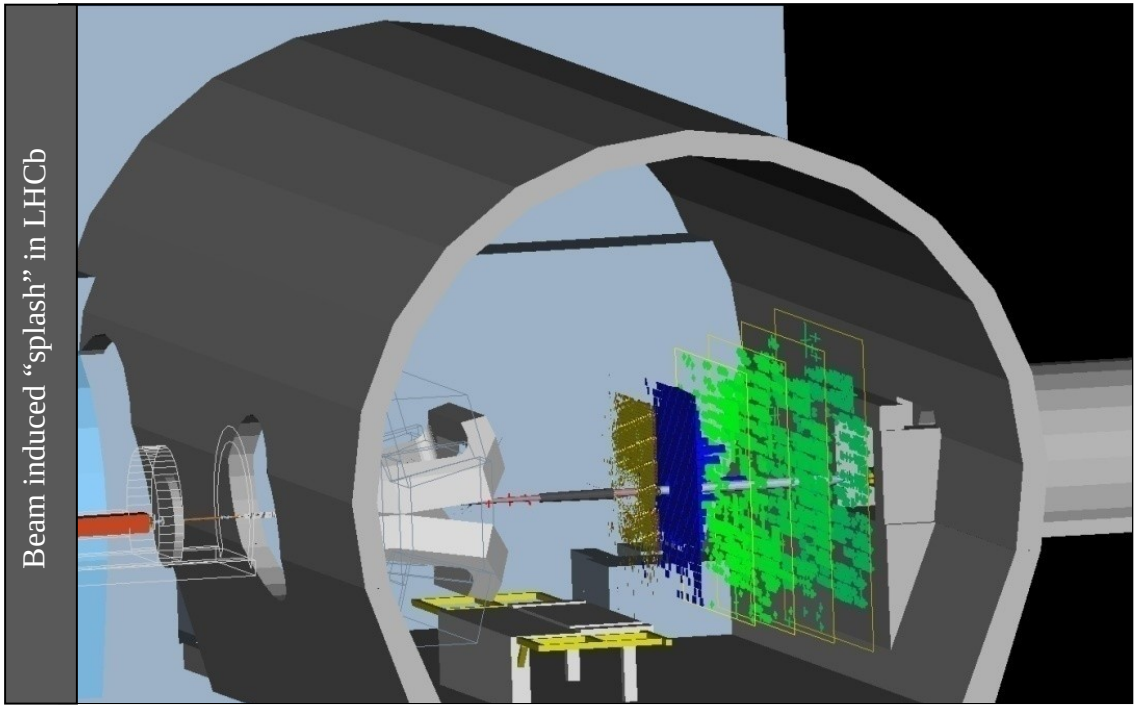
Imperial College
London




# Past experience on analysis environments

**Ulrik Egede**

**11 March 2010**

# Introduction

I will try here to look at

- User and developer relationship
- Tools to ease problem solving and consistency checking
- Documentation and training

Will illustrate both good practise from the past and pitfalls to avoid

Recommendations are written in **bold red** for easy identification.

# Use-cases and interface reviews

Really just comments on any kind of software development but of particular importance for analysis

Develop and document a large number of use-cases

```
Pre job merging
    A user defines before a job is running how the output data
    from the sub jobs should be merged. When all parts of the job
    have finished the ones that terminated successfully have
    their output merged without any user interaction.
```

Review the user interface by dry-testing use cases against it, develop partial implementations in prototypes etc.

**Iterate this process throughout lifetime of experiment.**

# Pseudo languages

The configuration of jobs often develops into pseudo-languages

- Different workings in different part of code
- Notation not documented
- Mixture of static and dynamic features
- Home written interpreters which are full of bugs and features

An example from LHCb

```
(BPVDIRA> %(IntDIRA)s ) & (INTREE((ABSID=='mu-') & (TRCHI2DOF< %
(TrackChi2)s ))) & (INTREE((ABSID=='pi+') & (TRCHI2DOF< %(TrackChi2)s
))) & ( BPVVDCHI2 > %(IntFlightCHI2)s )
```

Lesson is to **use object oriented parts of scripting languages such as Python or Ruby for the configuration**.

# What is the default value?

To understand the configuration of an analysis job is often very hard due to an overlay of:

- “Default” value specified in C++ code
- Different “default” value in associated configuration file
- Yet another “default” value set in some “default” configuration file that is always loaded.
- Users setting “magic” values they found in some old email thread and then forgot about.

**Turn configuration into intelligent objects**

- Full history of overlaying configuration changes.
- Persist configuration with job results so it can be queried in retrospect.

# Users and developers

There is a real danger in analysis frameworks to develop into a "*them versus us*" way of working.

Typical steps are like ...

- In the beginning not many tools are available and everybody is an expert
- Tools start to become available
- Pseudo-language develops to configure tools
- Developers and users no longer speak the same language
- Users need developers to add even trivial extensions
- Physics suffers as users "*make do*" with what is already there.

I recommend to **create interfaces in consultation with non-experts**. Maybe create short-lived focus groups.

# GUI's

Several attempts of high-level *drag-and-drop* style GUI's have been attempted in the past.

- Great for getting an initial feel of a framework
- A real pain for performing repetitive work

If developing a GUI make sure that it has one-to-one correspondence to a well documented and easy to use API.

- **Don't allow the developers to hide a poor API behind a few sleek looking windows ...**

# Separation of data, code and configuration

In many current frameworks there is often in the same script file

- A list of the data files that are to be analysed
- The creation of new code
- The configuration of existing code

From any automatic processing of analysis code this is a disaster; different operations are required on each part.

**Keep distinct what are logically distinct entities, even if implementation is the same**.

# User datasets

A tricky bit of many user analyses is simple bookkeeping of what data has been analysed and under which conditions.

- Dataset definitions often exchanged on Wiki pages, email or pieces of paper.

**Ensure that all user datasets are registered in some bookkeeping with proper history of how they were created**.

**Make it easy to replicate user data for distributed analysis**.

# Duplication of data

Any experiment I have been associated with faced the problem of users duplicating complicated but space efficient data structures into simple Ntuples in ZEBRA or ROOT format.

The gain from the user perspective is a familiar interface, independence from experiment code and access speed.

The loss from experiment (and ultimately user) perspective is

- Duplication of data (might not be harmful)
- Loss of bookkeeping (always harmful)
- Duplication of selection/fitting code (often harmful)
- Lack of documentation (always harmful)

# Duplication of data

Most collaborations keep fighting the “do not copy data” wars – and always lose them!

So what to do?

**Design your most compressed data type (μDST or whatever) first and not last.**

**Make access to this data format as simple as possible; minimal installation and working on multiple architectures.**

# Enforce documentation

If the implementation of interfaces and configuration is well defined, it is easy to enforce documentation at time of development

- Type checking
- Short document string
- Example of usage

# Software training

Training in the use of software is one of the things that LHCb has got right.

- Beginners training a part of every collaboration week
- Always teach the use of the latest software
- No registration required
- Often users will attend more than once to get updated on latest changes.

**Recommendation is to copy this**

**Request user feedback on any software training**

# Analysis in a distributed environment

In BaBar there were several attempts to catch up on the Grid world for analysis of data.

- They all failed (afaik)

To make efficient use of a distributed system, it needs to be built in from the beginning. This requires:

- Equivalent execution in Local and Grid environments
- Ability to perform a “static” analysis of analysis job
- Performance and debug information available for Grid jobs

Use a proper user interface for interaction with the Grid

- **Ganga or similar tool with properly developed application plugins**

# Non-centralised users

The support of all users at a centralised facility (CERN/SLAC) through interactive logins is expensive

**Maybe consider simply not doing it !**

This obviously puts up some requirements

- Full Grid access to **all** data
- Software installation should be trivial
  - Consider yum or similar from modern Linux distributions.
  - Going down the route of pre-configured virtual machines (like cernVM) is a possibility. Need user and site acceptance.
- Self testing should be an integral part of the installation, ie a given software module should be able to tell if it works.

# Conclusion

To develop a great system for end-users, they should be included in the design process.

- Focus groups
- Extensive development of use cases

Configuration, data access, Grid access and user training are the trickiest areas to get right.

Don't re-invent the wheel where there are good solutions already available.