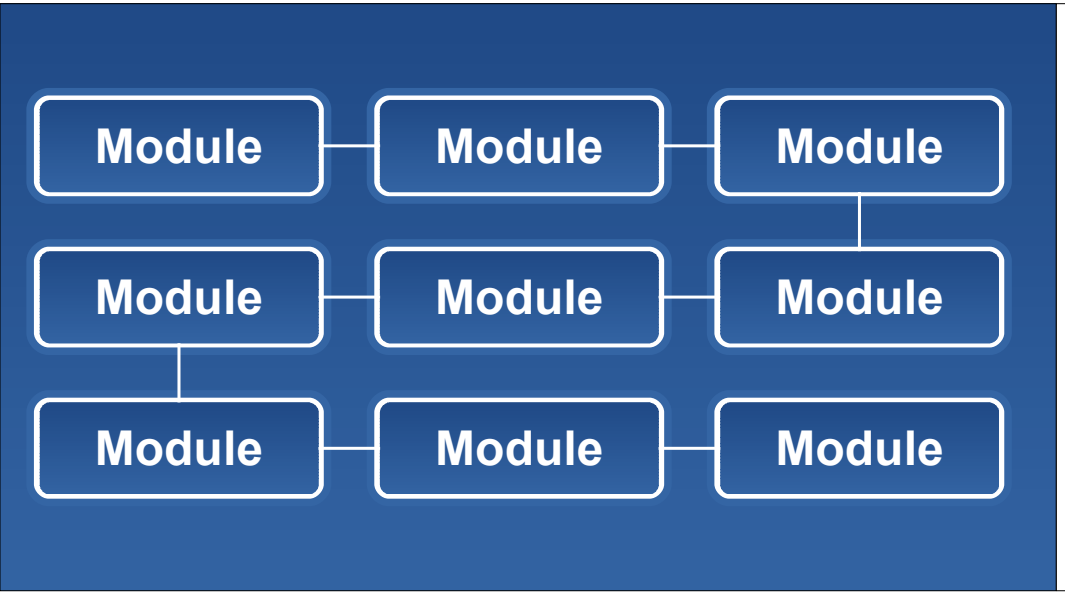
The Design of basf2
ideas and considerations



Takanori Hara
Andreas Moll
Martin Heck
Guofu Cao
Ryosuke Itoh
Nobu Katayama

Belle II Computing Workshop
16/06/2010

Belle II

- Design is driven by ...
- External libraries
- Modularization
- Generalization
- Design patterns
- Object oriented design
- Summary

The **design** and **architecture** of basf2 is driven by:

- ✓ Technology choices
- ✓ Modern approaches
- ✓ Lack of man power

Especially the last point was the driving force

Target: Find a design which

allows keeping the **maintenance** work of the framework core part **low** and at the same time **simplifies distributing** the work.

Solution:

- Rely on well established external libraries
- Modularization
- Generalization
- Design patters

Use modern, well documented **external libraries/tools**

Python Well established, documented scripting language.
Works very well with C++ together

Boost The de-facto standard for advanced C++ functionality.
Well documented, very high quality (libraries have to pass a referee committee)

ROOT The de-facto standard in HEP

XML Widely used throughout the industry and the HEP community.
A lot of documentation and tools are available.

By using well-documented, established external tools, maintenance work is moved from the **computing group** to **external groups**.

Python **Steering** of the framework by the user

Used as internal script language
(e.g. used for the command line parameters “-i” “-v” “-m”)

basf2 -m

```
from basf2 import *

avModList = fw.available_modules()
print "The following modules were found (%s):" %(len(avModList))

for item in avModList:
  print ""
  print "==================="
  print item.type()
  print "-------------------"

  paramList = item.available_params()
  for paramItem in paramList:
    defaultStr = ", ".join(['%s' % defaultItem for defaultItem in paramItem.default])
    print '%-20s %-14s %-30s %s' % (paramItem.name, paramItem.type, defaultStr, paramItem.description)
```

Boost
- operating system independent filesystem access
- shared pointer
- Python binding

XML
- XML document object model
- XInclude
- XPath

Keep the framework core **small** and **clean**

- Core has no knowledge about data
- Core is not responsible for data input/output
- Core has no knowledge about the event model (DataStore structure)
- Core does not deal with parallel processing

Use **modules** for nearly all tasks:

Data input/output

Data processing

Testing

Event/Run number generation for simulation

Instead of implementing a lot of features into the **framework**, move features to **modules** and **libraries**.

Keep subdetector groups busy and not the computing group

Data input / output modules

Add input/output modules according to the user's requirements:

Simple input/output modules	for single file access
Parallel input/output modules	for DAQ, HLT

Remove input/output modules for MonteCarlo event generation

Add multiple input modules for mixing signal and background events

Event/Run number generation

For **MonteCarlo** event generation the current *exp/run/evt number* has to be set

- store the values as event meta data in the **DataStore**
- use a module (*EvtMetaGen*) to set the values

Testing

Add unit test like features to a special test module for automated tests

Don't find a quick solution for a specific problem !

Always ask yourself if the solution can be **generalized**.

Advantages:

- Similar problems will already have a solution in the future
- Related parts of the source code are able to benefit from the solution

Basic idea behind generalization in basf2:

The framework should behave like a construction kit

Provide the user "**building blocks**" instead of pre-defined solutions

For example:

many (small) specialized modules which can be **freely combined** to adapt to any kind of task

provide python methods to access **framework information**:

built-in python functionality can then be used to manipulate the information (e.g. print to pdf, sent to ROOT etc.)

Looking forward to see creative solutions from the users !

Problem:

For **parallel processing** we needed to know, if there is an **input module** available which can read events from an **event server**.

Quick solution:

Search in the module chain for a module having a **specific name** (e.g. “pIOModule”)

Generalization:

Introduced module properties

```
EvtMetaGen::EvtMetaGen(bool selfRegisterType)
  : Module("EvtMetaGen", selfRegisterType)
{
 setPropertyFlags(c_TriggersNewRun | c_TriggersEndOfData |
                  c_RequiresSingleProcess);
}
```

Solves the initial problem. **But** also allows to check for modules requiring single process mode, GUI etc.

Design pattern: A general reusable solution to a commonly occurring problem in software design.

Object oriented design problem ➡ Apply design pattern ➡ **Problem solved !**

Save time by avoiding to reinvent the wheel !

Reference:

- **Design Patterns: Elements of Reusable Object-Oriented Software**, E. Gamma et al., Addison-Wesley professional computing series. ISBN 0-201-63361-2

Design patters used in **basf2**:

- ✓ Singleton
- ✓ Iterator
- ✓ Decorator pattern

Abstraction:

Use abstract classes to clearly define a class interface.

basf2:

Gearbox I/O classes

Logging I/O classes

Encapsulation:

Hide internal information. Clearly define access methods.

basf2:

Avoid returning pointers.

References or **shared pointers** are used.

Inheritance:

Avoid complex inheritance hierarchies

basf2:

One **Creator** base class for **geometry**, **materials**, **global params**

One **module** base class

- basf2 design is driven by **simplifying work sharing**
- **Modularization** and **Generalization** are the cornerstones of the basf2 design
- **Design patterns** help to find well known solutions to well known problems
- Make use of the **beauty** of object oriented design

Think carefully before you start writing source code