The basf2 Framework
an introduction

Python steering

Geant4

Parallel processing

Modules

Gearbox

DataStore

<u>Takanori Hara</u>
Andreas Moll
Martin Heck
Guofu Cao
Ryosuke Itoh
Nobu Katayama

Belle II Computing Workshop
16/06/2010

Belle II

- History of the Belle II framework
- Basic architecture
- Modules
- Data store
- Geometry library
- Simulation
- Steering of the framework
- Summary

History of basf2

or how everything began

Software framework of BELLE: **basf** (**b**elle **a**naly**s**is **f**ramework)

Successfully used since over **10 years** for

Online

HLT (High level trigger)
DAQ (Data acquisition)
DQM (Data quality monitor)

Offline

BELLE detector **optimization**
Physics **analysis**

Conclusion after 10 years:

- \+ **Proven** concepts: modules, paths, usage as online and offline software
- \+ **Verified** and well tested for BELLE
- \+ Combines a lot of knowledge (**treasure box**)
- − Many **hard coded** values: hard to adapt for Belle II
- − Lack of **documentation**
- − Software not **state-of-the-art** any more (steering, external software)

basf not suited for Belle II, for example:

Clustering: Number of ganged strips is hard coded
Tracking: Assumes exactly 4 layers
Simulation: Uses geant3
Analysis: old hbook files (e.g. only float values)

A new software framework for Belle II has to be developed

Two **independent** framework developments started:

roobasf

@ KEK

Adds support for reading/writing ROOT files to **basf** (including parallel processing support !)

solves hbook problem

Until 2010/02/23 roobasf part of basf library

Release of stand alone version

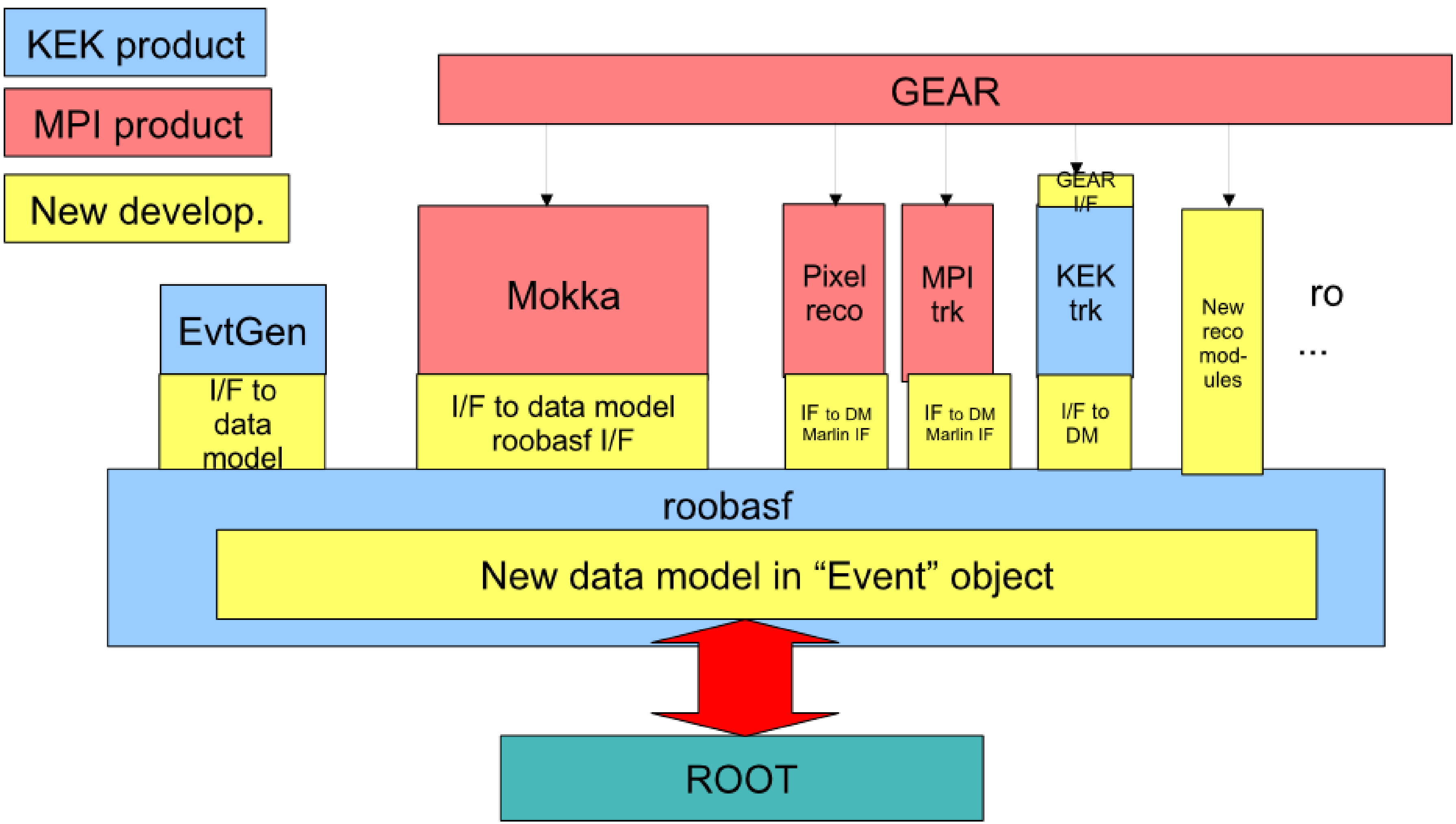
ILC

@ MPI and Charles University

Adapted **ILC** software for PXD optimization (Simulation of Belle II tracking detectors)

geant4 simulation,
detailed description of PXD, SVD

On a **legendary** meeting during July 2009 B2GM (2009/07/07), Itoh-san proposed the **"Aufheben-Solution"** (merging of roobasf with ILC):



Nice idea to **combine** both frameworks. But not the optimal solution for a new software.

Better: write a new software and take the best things from both frameworks.

Parallel development on roobasf and ILC continued... until **2010/04/05**

Development of the new Belle II software framework started:

basf2

Reasons to start the development of a new framework:

ILC

- Event model not flexible enough (e.g. problems implementing CDC wires)
- Steering with XML not optimal
- No common geometry solution for simulation and reconstruction

roobasf

- Code needed restructuring and documentation
- Still driven by the idea of the Aufheben-Solution

manpower

Andreas Moll, **Martin Heck** and **Guofu Cao** can work for **3 months** exclusively on the software framework under the guidance of **Nobu Katayama** and **Ryosuke Itoh**.

This talk presents the result of the work done in the last 2.5 months

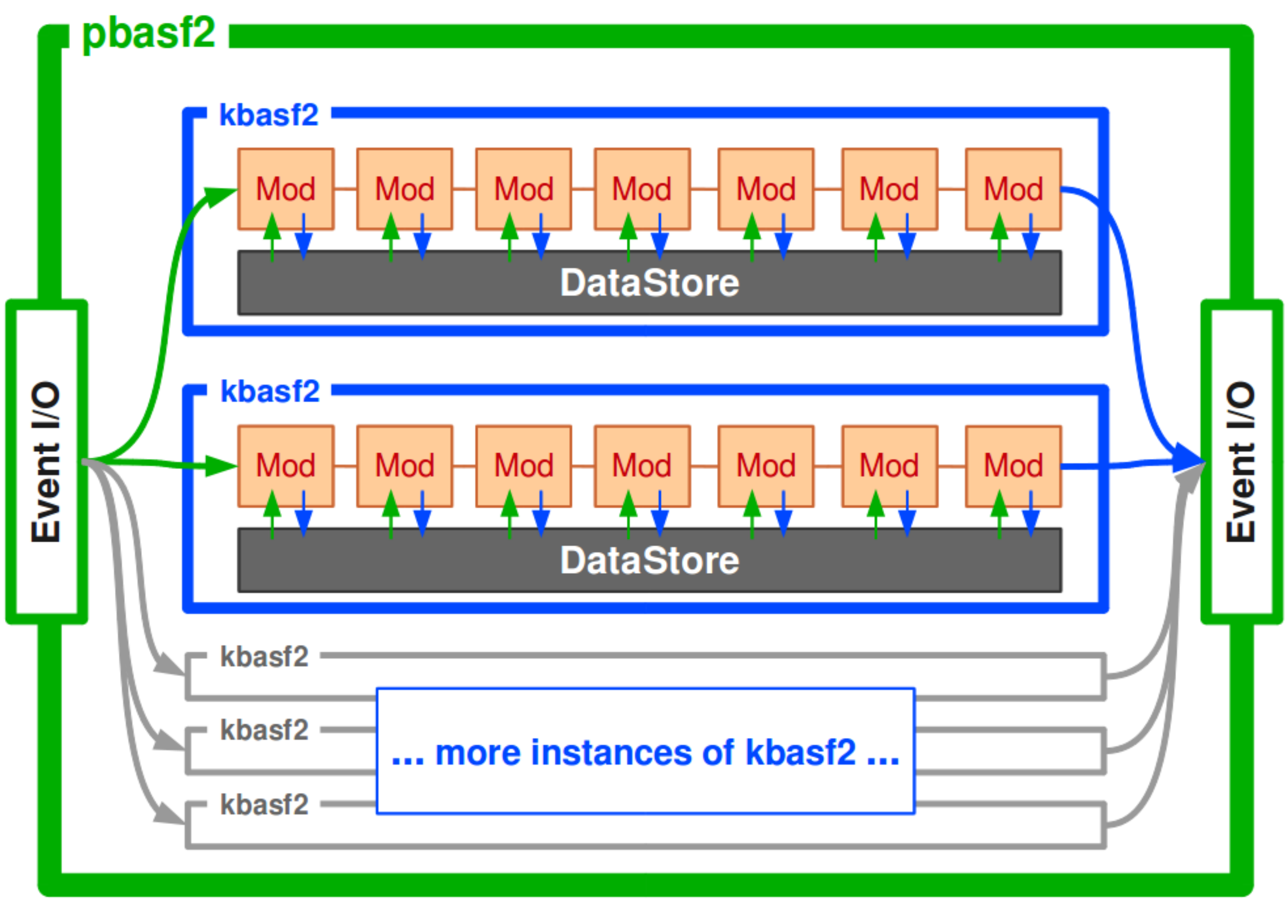
Basic architecture

designing the tools of tomorrow

basf2 is divided into two subsystems: **kbasf2** and **pbasf2**

kbasf2 processes a single stream of data by executing smaller data processing blocks called **modules**.

pbasf2 runs multiple instances of **kbasf2** in parallel and handles the parallel reading and writing of event data.



The data processing chain consists of a linear arrangement of **modules**.

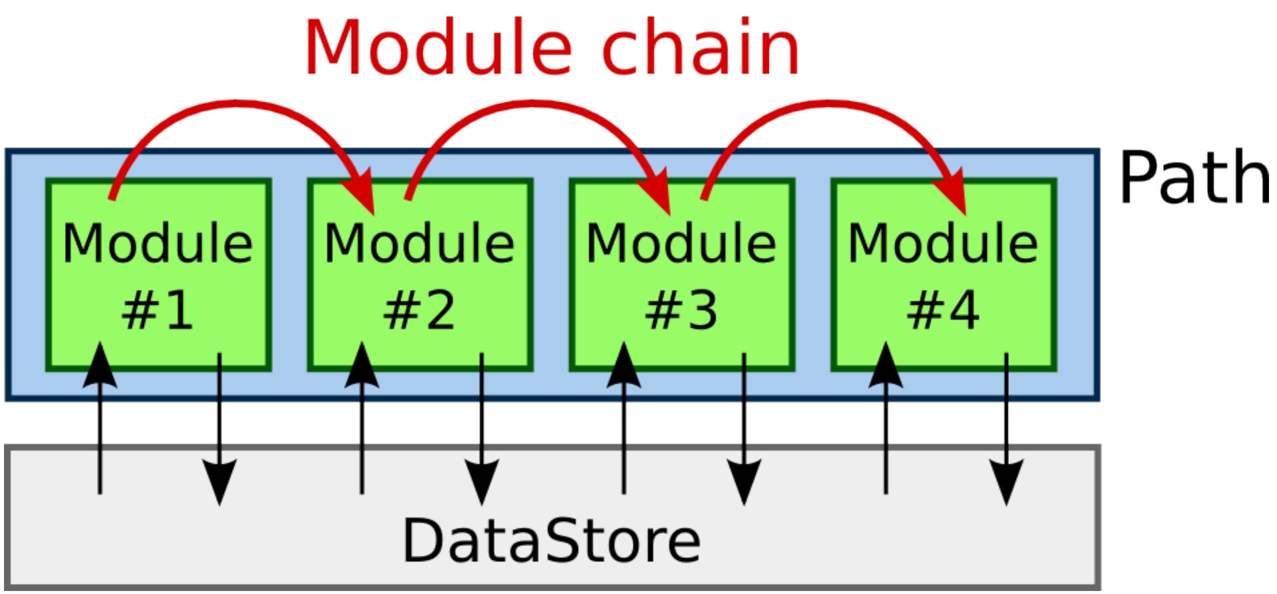
Typical modules: *data input*, *geometry input*, *simulation*, *tracking*, *data output*...

Unlike **roobasf** and **ILC**, reading and writing of even data is done in modules

Modules live in a **path** (container where the modules are arranged in a strict linear order.)

Framework executes **modules** one at a time, exactly in the order in which they were placed into the **path**

Processed data is stored in a common storage, the **DataStore**.



basf2 allows the user to create an arbitrary number of **paths**.

Paths can then be connected with each other using **conditions**.

Each **module** can return an **integer** number or a **boolean** value.

➡ Depending on these return values and a user defined **condition**, the process flow can switch from one **path** to another.

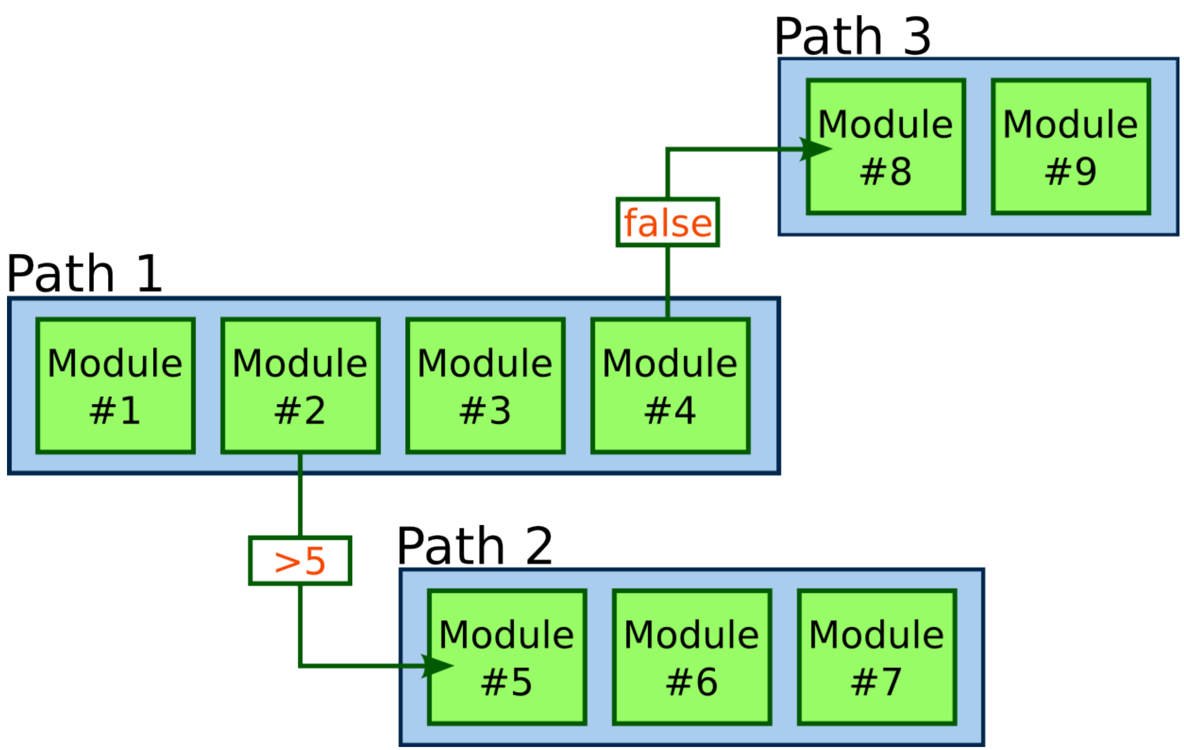
C++ code: in the ***event()*** method (see later):

Boolean:

```
void EvtMetaGen::event()
{
  //...
  setReturnValue(false);
}
```

Integer:

```
void EvtMetaGen::event()
{
  //...
  setReturnValue(10);
}
```

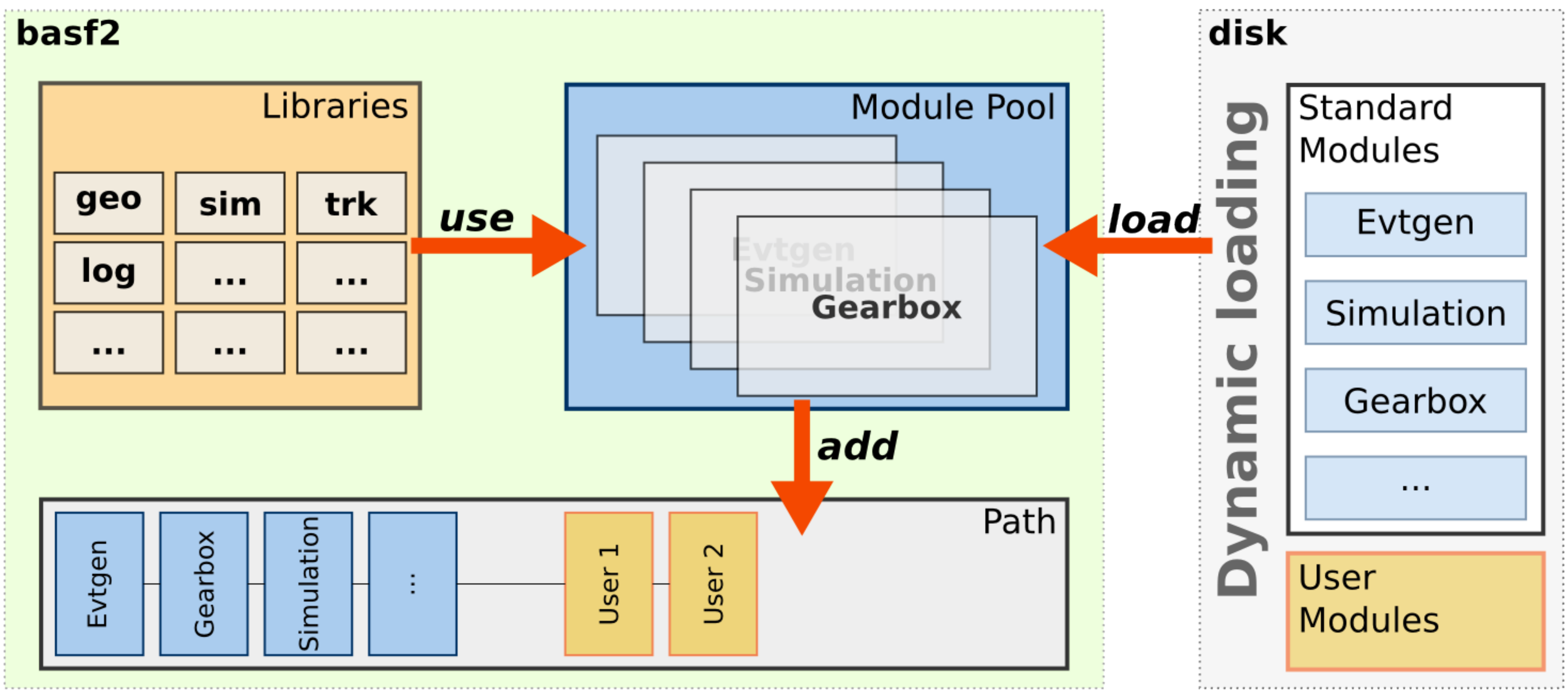


In **basf2** **libraries** and **modules** are separated:

> **Modules** live in a "*Module Pool*"
> **Libraries** in a "*Library Pool*"

Usually, a **library** encapsulates a specific set of functionality of the **basf2** framework (e.g. *geometry handling*, *simulation code*, *tracking algorithms*, etc.)

Advantage: The algorithms/functionality of the framework can be used by multiple modules (e.g. tracking algorithms: in the tracking module, analysis module etc.)



Modules

the LEGO® bricks of basf2

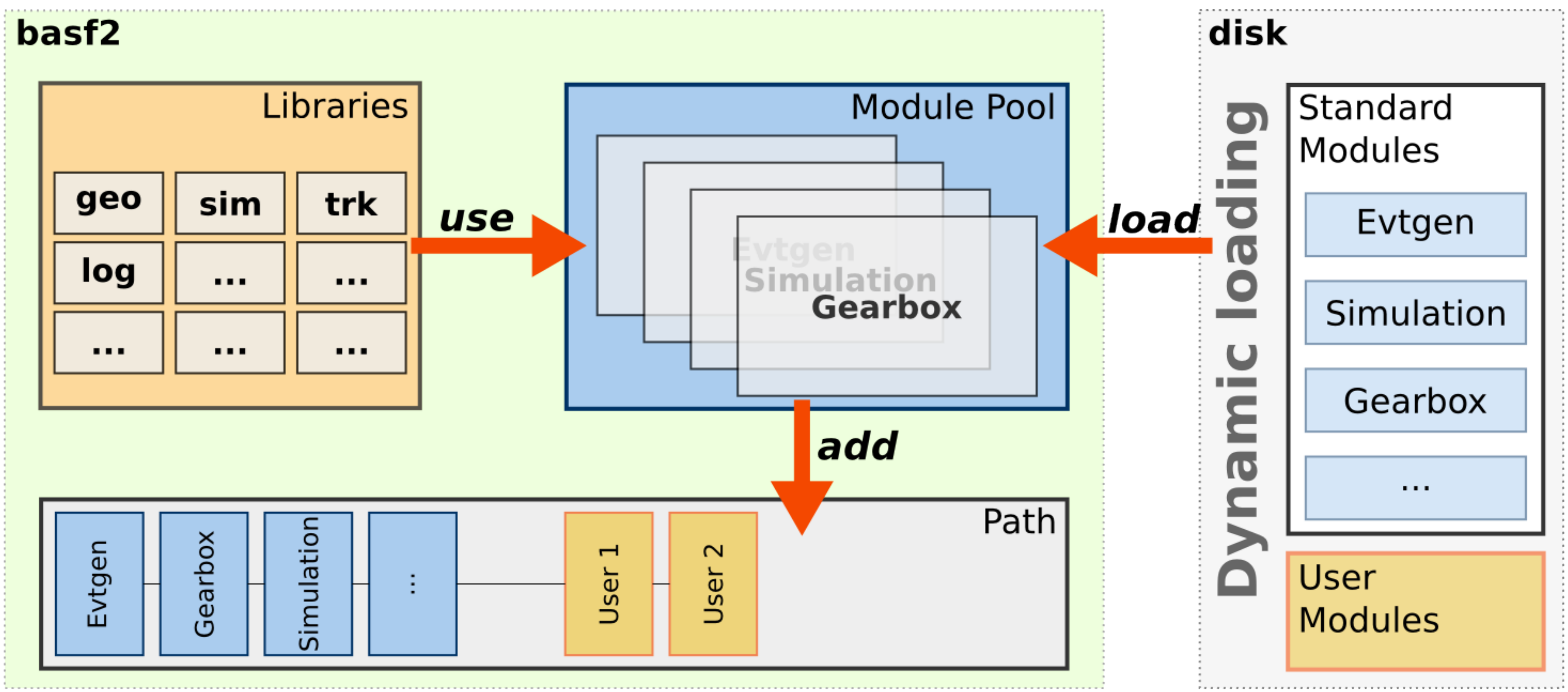
Modules are the building blocks of any event processing chain.

They are **automatically** loaded by the framework at **runtime** and put to a "*Module Pool*".

Standard modules are shipped with the framework
Additional modules (e.g. analysis modules) can be added by the user

The user can then select an arbitrary number of modules from the "Module Pool" and add them to a path.

A module is identified by its unique name.



Module structure

A **module** is a C++ class having a clearly defined structure:

```
class EvtMetaGen : public Module {

public:
   NEW_MODULE(EvtMetaGen)

   EvtMetaGen(bool selfReg = true);
   virtual ~EvtMetaGen();

   virtual void initialize();

   virtual void beginRun();

   virtual void event();

   virtual void endRun();

   virtual void terminate();

protected:

private:

};
```

Inherits from abstract base class

Macro which returns a new instance of the module

Constructor (add param, set properties)

Initialize the module (set variables etc.)

Called at the beginning of each run

“Worker” method

Called at the end of each run

Terminate the module
(unset variables, delete memory etc.)

Parameters allow the user to steer a **module** (e.g. change its behaviour)

```
private:

double m_dEdxCut;
std::string m_rootFilename;
```

Add a *member variable* to the **module** class which stores the value of the parameter.

Connect the *member variable* to a parameter name (in the constructor of the class)

```
EvtMetaGen::EvtMetaGen(bool selfRegisterType)
  : Module("EvtMetaGen", selfRegisterType)
{
 addParam(“dEdxCut”, m_dEdxCut, 0.4, “Only use tracks above this cut”)
 addParam(“rootOutput”, m_rootFilename, “debug.root”, “Root output”);
}
```

Set the value of the parameter in the python steering file

```
evtmetagen.param(“dEdxCut”, 0.75)
evtmetagen.param(“rootOutput”, “test075.root”)
```

Inside the **module**, the *member variable* holding the parameter value can be used like any other variable.

Supported parameter types:

Basic types

Python	C++	Description
int	int	Integer number
float	double	Floating point number
str	string	Text
bool	bool	Boolean value

List types

Python	C++	Description
list (*int*)	std::vector<int>	List of integer numbers
list (*float*)	std::vector<double>	List of floating point num
list (*str*)	std::vector<string>	List of single line strings
list (*bool*)	std::vector<bool>	List of boolean values

A **module** can flag its functionality:

- **Check** if running environment matches module chain
- **Prepare** the framework for the use in DAQ, HLT, DQM

Example usage scenarios:

- Run framework in *single processing mode* if there is at least one **module** in the chain which requires single processing.
- In a parallel processing environment: Check if an input **module** with *multi processing capabilities* is available.

Set the **properties** in the *constructor* or the *initialize()* method of the class

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

Available **properties**:

c_TriggersNewRun	This module is able to trigger **new runs**.
c_TriggersEndOfData	This module is able to send the message that there is **no more data** available.
c_ReadsDataSingleProcess	This module is able to **read data** from a single data stream (disk/server).
c_ReadsDataMultiProcess	This module is able to **read data** from an event streaming server.
c_WritesDataSingleProcess	This module is able to **write data** into a single data stream (disk/server).
c_WritesDataMultiProcess	This module is able to **write data** to an event streaming server.
c_RequiresSingleProcess	This module requires the framework to run in **single processing mode**.
c_RequiresGUISupport	This module requires the framework to have **GUI support** enabled.

DataStore

storing data has never been easier

The **DataStore** manages all data loaded or created during the processing of events

Can store **any class** that inherits from *TObject* and has a ROOT dictionary.

Three different "***durability types***" are available:

Event

Contains the data stored for one single event.

Content is automatically deleted by the framework after each event.

Run

Contains the data stored for one run.

Content is automatically deleted by the framework after each run.

Persistent

Contains the data stored for the duration of the event processing.

Content is automatically deleted by the framework before event processing starts.

No pre-defined event model (like ILC).

The data which should be stored is defined by each **subdetector group**.

The content of the **DataStore** is accessed (read/write) by two ***accessor classes:***

StoreObjPtr	single objects
StoreArray	object arrays

Templates are used to have typesafe objects

Example: get a single CDC hit carrying the name “testHit1”

```
StoreObjPtr<HitCDC> cdcPointer1("TestHit1");
```

If the CDC hit does not yet exist, it is created. Default durability type is ***c_Event***.

Calling the ***methods*** of the (*HitCDC*) class is then easy. **For example**:

```
cdcPointer1->setWireId(243);
```

Same result: create an empty *access pointer* and assign an object to it

```
StoreObjPtr<HitCDC> cdcPointer2;
cdcPointer2.assignObject("TestHit1", c_Event, true);
```

Durability type

Generate new object if it does not exist.

In order to save your own class to the **DataStore**, your class should full fill the following requirements:

1. The class has to **inherit** from *TObject*

```
class YourClass : public TObject {
...
```

2. Add *ClassDef* to the header file:

```
class YourClass : public TObject {
...
private:
...
  ClassDef(YourClass, 1);
};
```

3. Add *ClassImp* to the source file:

```
#include <yourclass/YourClass.h>
ClassImp(YourClass);
```

4 Create a *linkdef.h* file

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link C++ nestedclasses;

#pragma link C++ class YourClass;

#endif
```

The build system will create automatically a dictionary file.

Your class can then be written to and read from the **DataStore**.

The code which writes/reads the data of your class to disk is automatically generated.

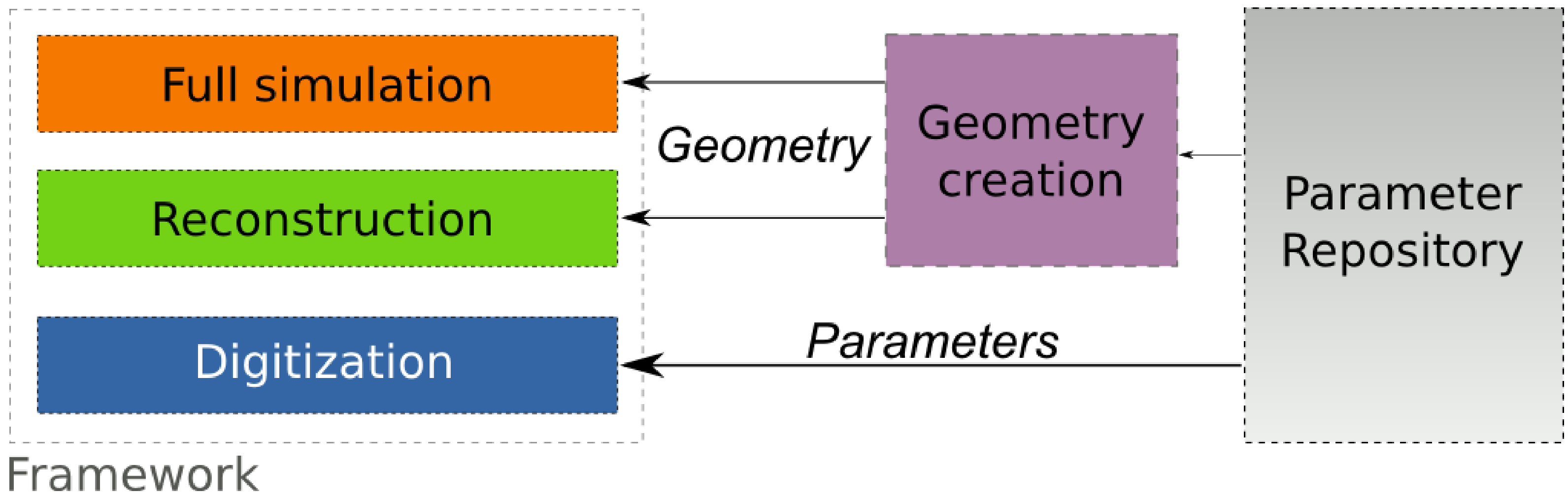
Geometry

shape your detector

Detector geometry is **ingredient** for

- geant4 detector simulation
- digitization
- reconstruction (e.g. tracking)

basf2 concept: Store parameters describing the geometry centrally
Create geometry objects (volumes) on-demand

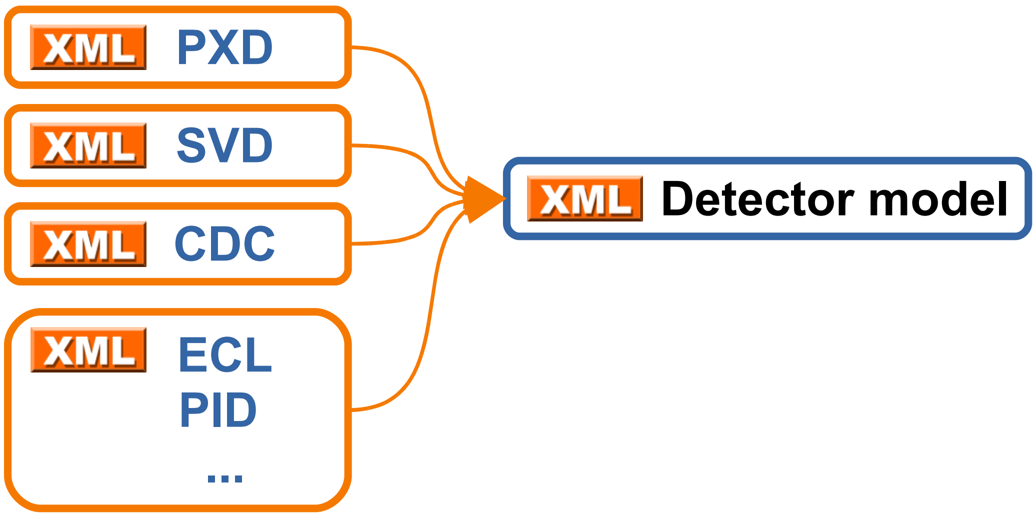


The geometry parameters are stored in **XML** files

Advantages of XML documents

- Human readable (e.g. content can be tracked by a version control system)
- Developed and maintained by the **World Wide Web Consortium** (W3C) (the main international standards organization for the World Wide Web)
- Industry-standard:
  - a lot of tools are available
  - sophisticated libraries for handling **XML** files exist
  - OpenSource/Free/Commercial GUI applications

- ✓ One **XML** file per subdetector
- ✓ One **XML** file describing a specific detector model



XML file example – PXD

Header

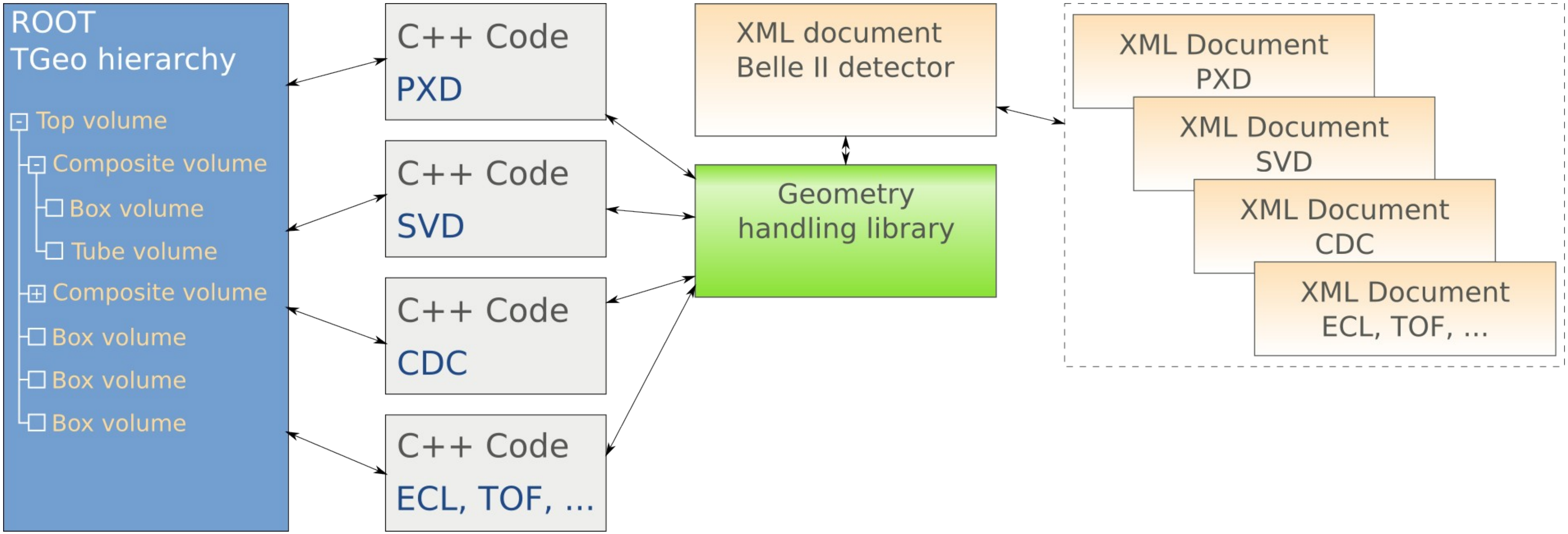
Content

```xml
<Subdetector type="PXD">
    <Name>PXD BelleII 1600pix</Name>
    <Description>The famous RedBull can</Description>
    <Version>0</Version>
    <Creator>PXDBelleII</Creator>
    <Content>
        <Layers>
            <Layer id="1">
                <Phi0 desc="..." unit="deg">90.0</Phi0>
                <Radius desc="..." unit="mm">13</Radius>
                <OffsetY desc="..." unit="mm">-2.25</OffsetY>
                <OffsetZ desc="..." unit="mm">11.7</OffsetZ>
                <NumberOfLadders desc="...">8</NumberOfLadders>

                <Ladder desc="...">
                    <Length desc="...">76.4</Length>
                    <Width desc="...">12.5</Width>
                    <Thickness desc="..." unit="um">50</Thickness>
                    <NumberOfSensors>2</NumberOfSensors>

                    <Sensor id="1">
                        <Gap desc="...">0</Gap>
                        <PadSizeRPhi desc="..." unit="um">50</PadSizeRPhi>
```

Specifies the C++ code which creates the geometry (see next slide)



The geometry objects (volumes) are ROOT **TGeo** objects, organized in a hierarchy.

The C++ code which creates the geometry objects (volumes) is called **Creator**

Creators behave similar to **modules** in the framework:

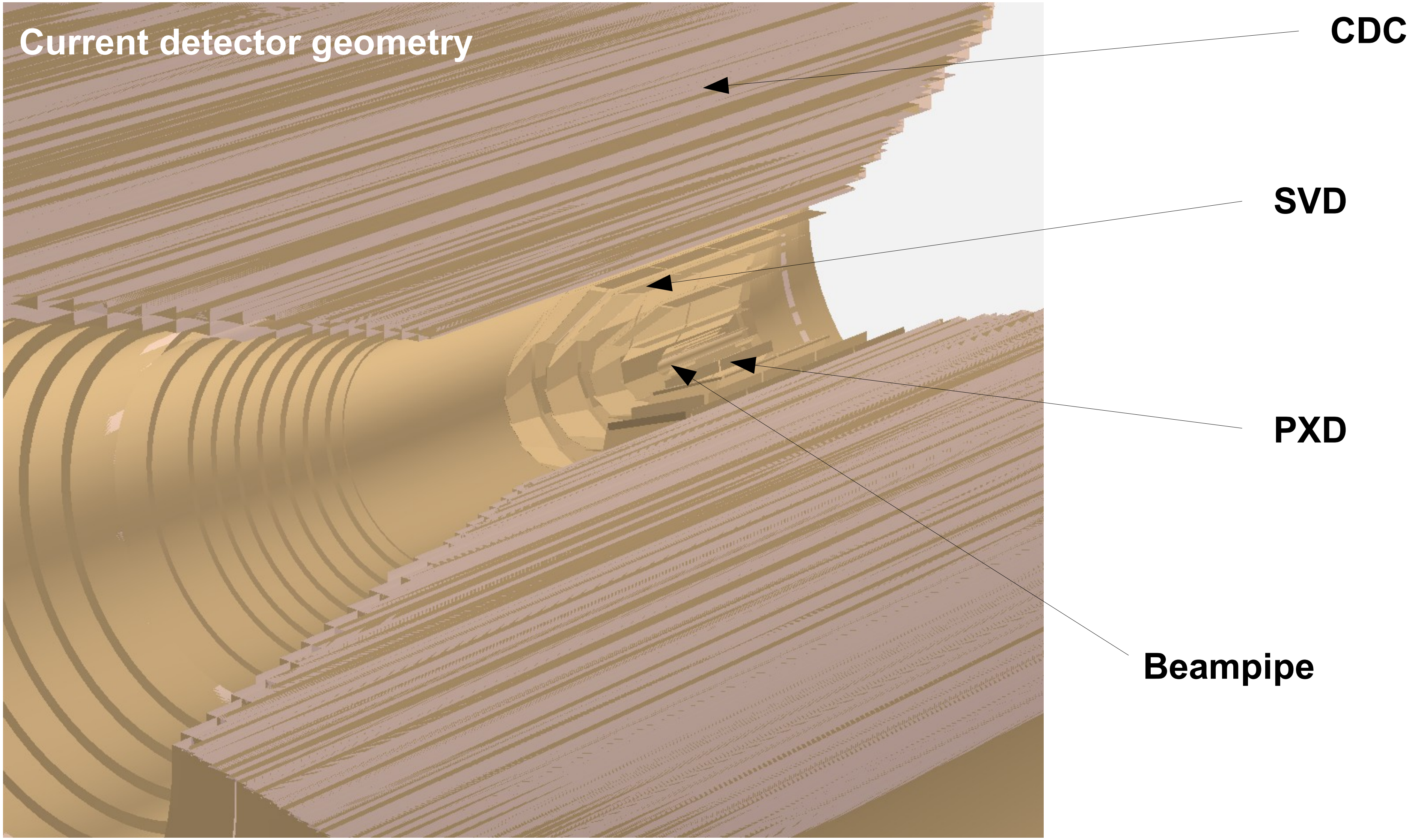
- Identified by their **unique name**
- **Inherit** from a single base class
- Implement a **defined interface**

Creators have only access to the **content** part of the XML document (see previous slide)

In **basf2**:

The geometry is created by the **Gearbox module** and stored in memory.

The created geometry is then available to all modules in the **module chain**.



Simulation

gives your particle wings

basf2 contains a **geant4** based simulation library (+**module**)

! great work done by **Guofu Cao**

The ROOT ***TGeo*** geometry is automatically converted to **geant4** using **g4root**.

- **Passive (dead)** volumes are automatically converted.
  Active (sensitive) volumes have to be defined by each subdetector.

Therefore, each **subdetector** has to provide

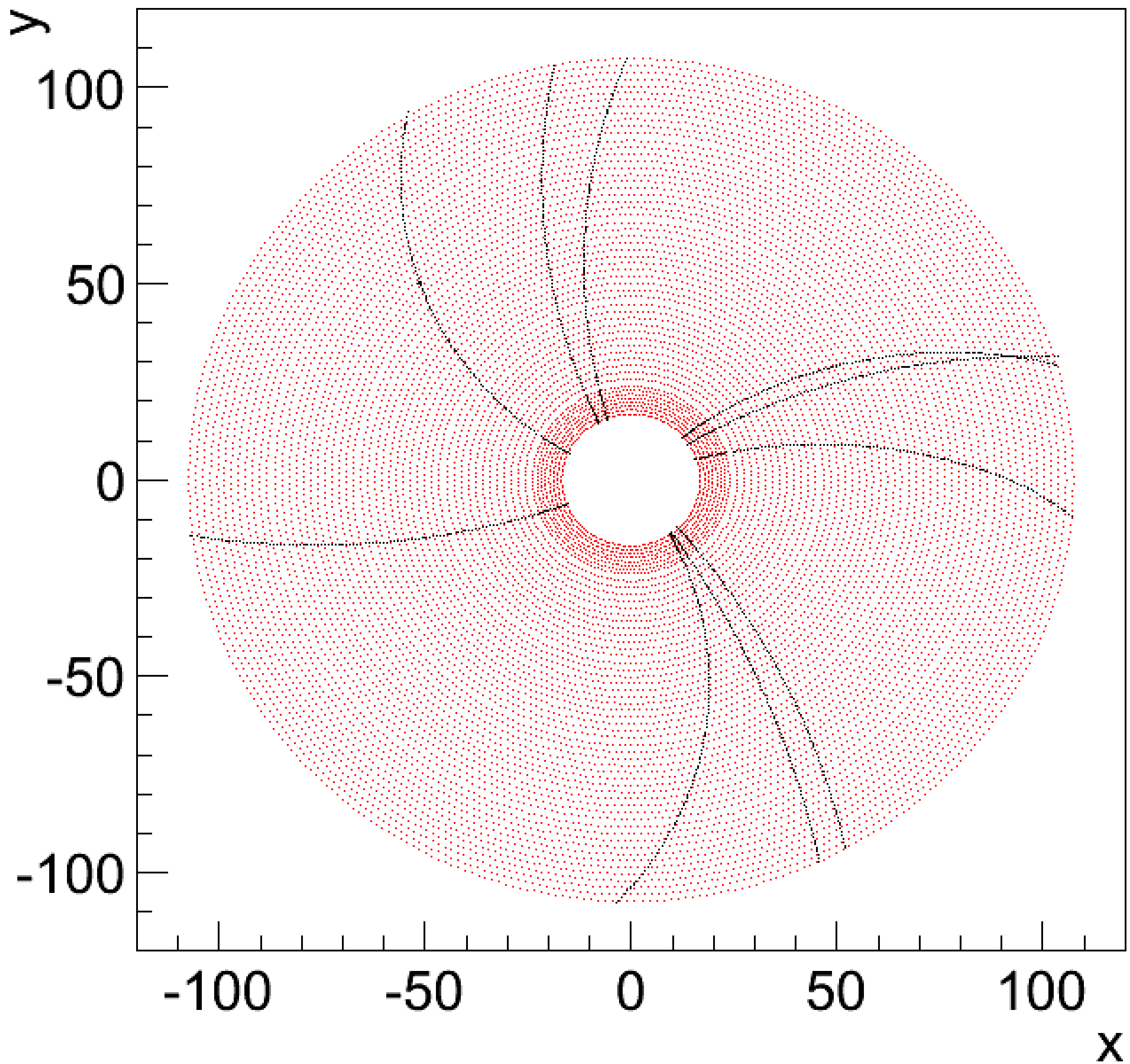
- a class which handles the sensitive detector
- a class which represents the simulation result (hit)
- a class which stores the result into the DataStore

Already implemented **subdetectors**:

PXD	Currently only the **sensitive silicon** parts are implemented
SVD	Currently only the **sensitive silicon** parts are implemented
CDC	**Highly detailed** implementation

Current implementation uses an **uniform magnetic field**.

CDC Sense Wires (Backward endplate):



Ten tracks in one event

π^+, 0.3 GeV < p < 1.3 GeV

In the figure the **geant4** pre-step positions are drawn

A simulation **module** (*simModule*) is available.

Since the **sensitive detectors** register themselves automatically to the simulation library, the **simulation module** has access to them without having to change the module.

➡ Advantage of having separated the libraries from the modules

Features of the simulation module:

- set the different **verbosity** levels of **geant4**
- activate **geant4** **interactive** mode
- supports **geant4** **visualization**
- Reads **geant4** macros
  - **Particle gun**
  - **HEPEvt files**

Steering

Command & Conquer basf2

After having installed **basf2**, you can start it by typing:

```
basf2
```

Show the available ***command line*** options:

```
basf2 --help
```

```
Generic options:
  -h [ --help ]            print all available options
  -v [ --version ]         print version string
  -i [ --info ]            print information about basf2
  -m [ --modules ]         print a list of all available modules

Configuration:
  --steering arg           the python steering file
```

Start **basf2** with a steering file (e.g. *steering.py*):

```
basf2 steering.py
```

basf2 uses Python to steer the framework

Advantages:

- Python is a standard scripting language
- Well documented (extensive language reference, books, tutorials)
- Add calculations, print statements and even analysis code (PyROOT) to your steering file.

Python steering file example:

```
from basf2 import *

#Create module
test = fw.register_module("Hello")

#Create path
main = fw.create_path()

#Add module to path
main.add_module(test)

#Start event processing
fw.process(main,100,1)
```

Import **basf2** environment

Register a **module**

Create a new **path**

Add the registered **modules** to the **path**.

Start event processing with path "*main*"
(100 events, run number 1)

Conditions (switching paths):

```
test.condition(path1)
```

If the **boolean return value** of test is **false**, the event processing continues with the first module in *path1*

```
test.condition(">5",path1)
```

If the **integer return value** of the module test is greater than **5**, the event processing continues with the first module in *path1*

Setting module **parameters**:

Directly:

```
test.param("CutdEdx",1.4)
test.param("Filename","/home/belle2/testFile.root")
```

Python dictionary:

```
testDict = {'CutdEdx'           : 1.4,
            'Filename'          : "/home/belle2/testFile.root",
            'Resolutions'       : [20.2, 23.4, 50.4, 55.7]
            'TrackDetectors'    : ["PXD", "SVD", "CDC"]}

test.param(testDict)
```

Basic architecture	Modules	DataStore
Geometry	Simulation	Steering

- ✓ All basic functionality of a framework are already available in basf2
- ✓ Documentation online (Doxygen + TWiki)

Under development or still missing:

- ? Tracking (GENFIT integration started)
- ? Integration of other subdetectors than PXD, SVD, CDC
- ? Vertex fitting (RAVE)
- ? Condition database

This talk just **scratched the surface**. Framework features not presented in this talk:

Architecture

- Module / Creator / Sensitive detector self registration mechanism
- Internal event processing mechanism
- Internal usage of shared pointer / templates / STL / boost
- Error handling
- Logging
- Build system

Modules

- Process-record return values

DataStore

- EventMetaData handling
- DataStore arrays
- DataStore iterators
- DataStore special objects

Geometry

- Geometry parameter access
- Standard units
- Creator development
- Materials

Simulation

- Simulation architecture
- Sensitive detector development
- Particle gun / HepEvt support
- Simulation module parameters

Steering

- Access to framework information
- Access to the DataStore
- Error statistics
- Process statistics

basf2 manual (introduction + installation + reference)

http://b2comp.kek.jp/~twiki/bin/view/Computing/Basf2manual

Doxygen documentation

http://www-ekp.physik.uni-karlsruhe.de/~heck/doxygen/