PNIF – the PNI file format

Design and development status

Eugen Wintersberger

PNIF-Design state

HDRI-Meeting, 16.05.2011

HELMHOLTZ ASSOCIATION

DESY

Motivation

Babylonian abundance of file formats for:

- Detector data
- Beamline data
- Simulation programs
- Other sources ...

Need importers for all these formats → large amount of code which is hard to maintian.

Solution: “standard” file format which provides

- Standard procedure how to access data (Data model + API)
- Facilities for defining standardized names (semantic standardization)

Why not Nexus?

Nexus has tow major problems:

1. The existing Nexus-API is not useful for beamline applications
2. Only a specification is available → 80 classes with all together ~800 attributes!
3. The specification is ambiguous in many points

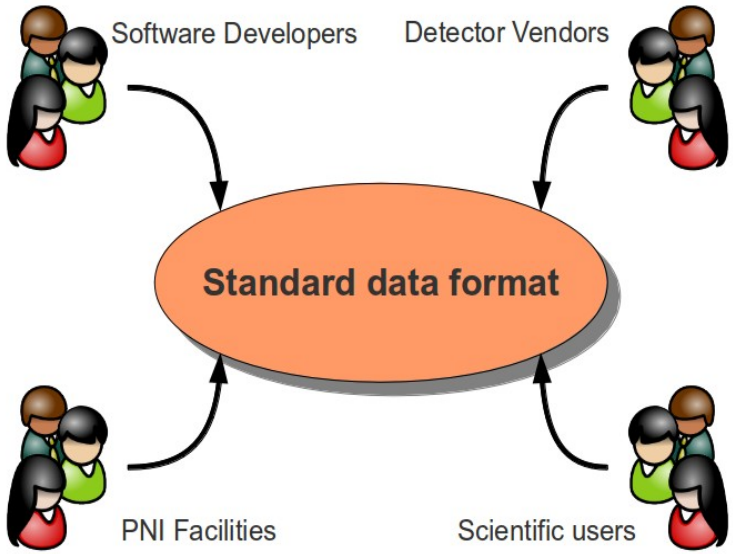
Consequences:

1. Would need a new implementation of the over 80 classes
2. Due to ambiguities the API can hardly implement creation rules of data structures
3. Users must be familiar with all classes and their attributes

Nexus is to complex to implement and to use!

Users and their demands

The success of a standard depends on the acceptance by its users!



- High performance
- Support for many programming environments
- Support for many different operating systems
- **Simple** to use
- **Simple** to maintain on long time scales
- Must be able to handle large data volumes
- Must provide a facility to standardize experimental method

Basic Design of PNIF

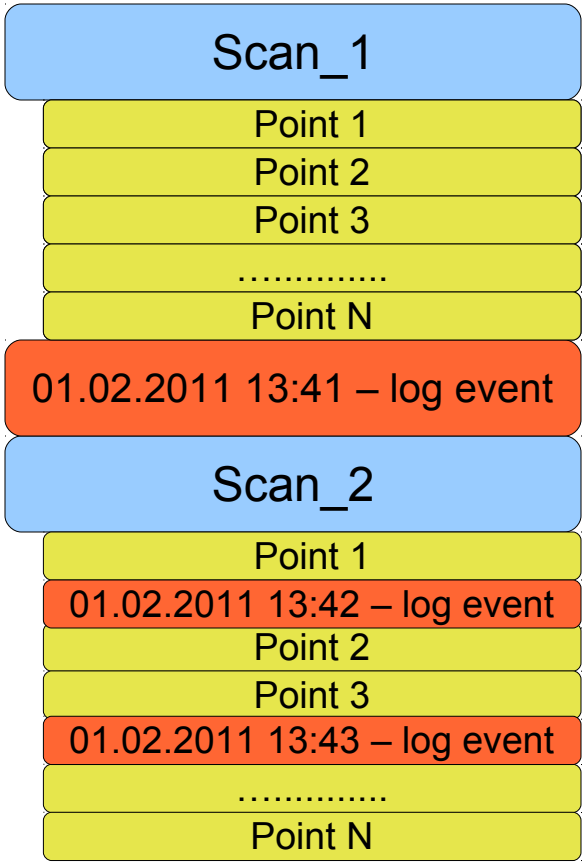
Separate Log and Data files!

Can make two observations

- Data is organized by “scans” or samples
- Logs events can happen at arbitrary points in time – organized by time stamps

Advantages of the separation

- Acquisition system not responsible for logging
- Must not touch data file for log information
- Data organization remains consistent

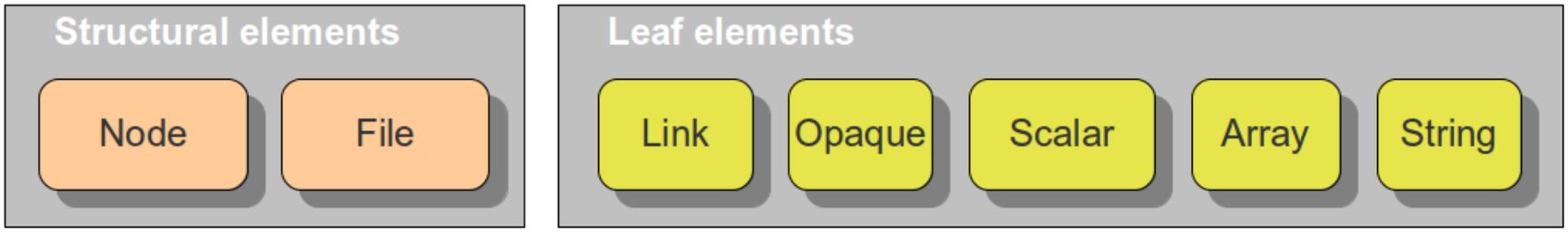


Data files: *.pnif

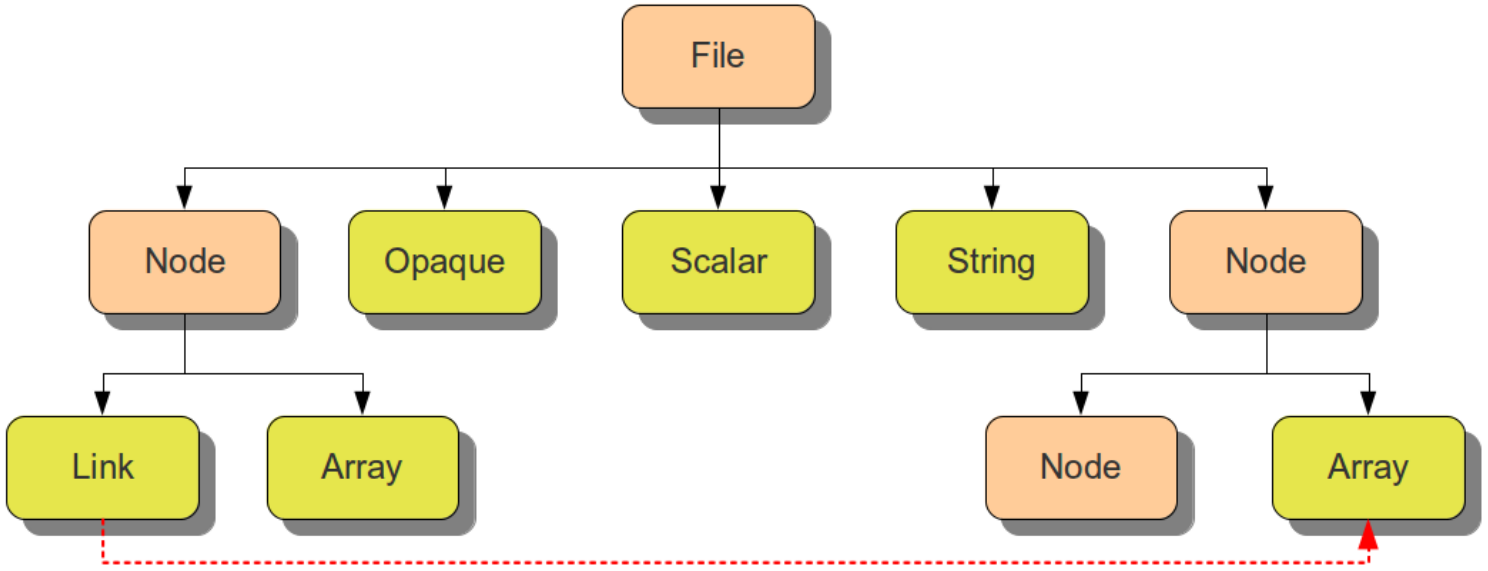
Log files: *.pnil

Basic Objects – organizing data

Tow types of objects in each file:



Each file represents a tree of objects with the File as its root:



Basic objects – objects and attributes

- Every object in the tree has a name (obviously necessary)
- Each object in the tree can hold an arbitrary number of attributes
- Attributes can be objects of type: Scalar, Array, and String
- **Some objects have mandatory attribute (except String-objects)!**

Mandatory attributes:

- **All objects except String**: `description` (class String)
- **Scalar**: `unit` (class String)
- **Array**: `unit` (class String), `axes` (class String)
- **Opaque**: `mime-type` (class String)

Mandatory attributes provide a minimal documentation of each quantity stored in a file!

Basic Concepts – data access

Each object including attributes can be addressed by a **unique path**:

Accessing an object:
`/path/to/object/objectname`

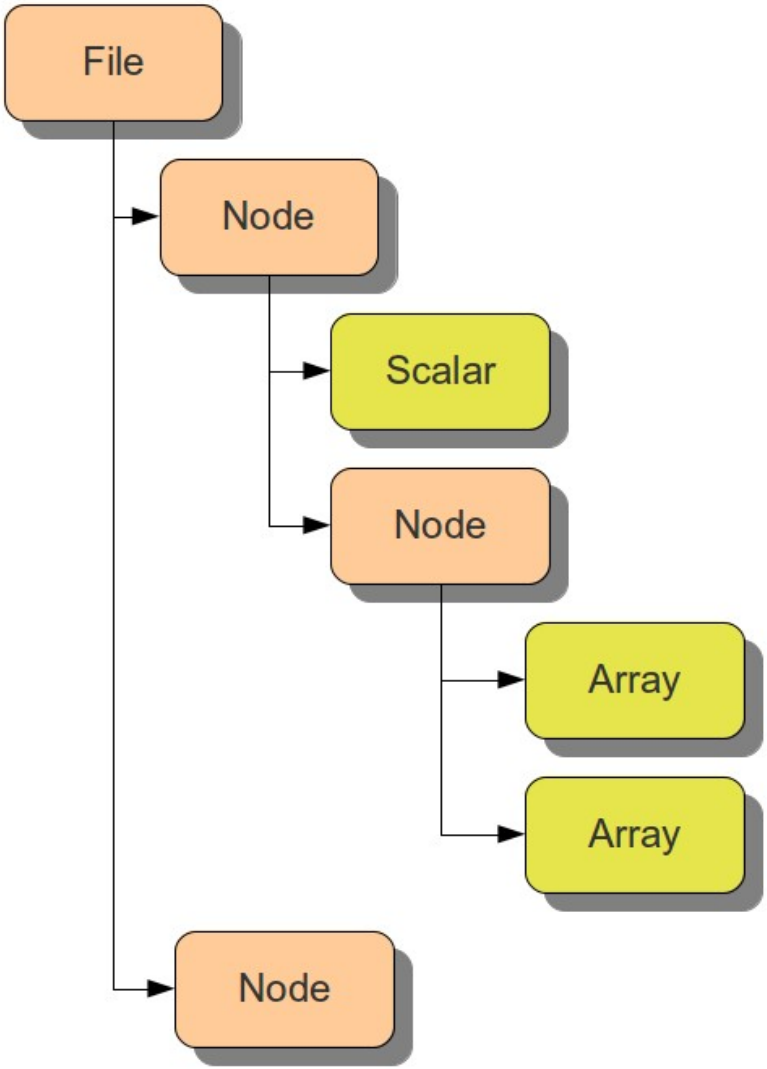
Accessing an attribute
`/path/to/object/objectname@attributename`

Accessing an object and an attribute including file reference:
`Filename.pnif:/path/to/object/objectname`
`Filename.pnif:/path/to/object/objectname@attributename`

- The same path syntax is used by API calls, Link objects, and maybe in the user-interface of applications
- Log and data files make use of the same API objects → objects can be referenced between log and data files!

Data files

Organization of objects in the Data-file



Only one restriction:
leaf-nodes must not reside directly below the File-node!

To ensure that this restriction is not broken: Nodes & Files act as factories for leafs and nodes!

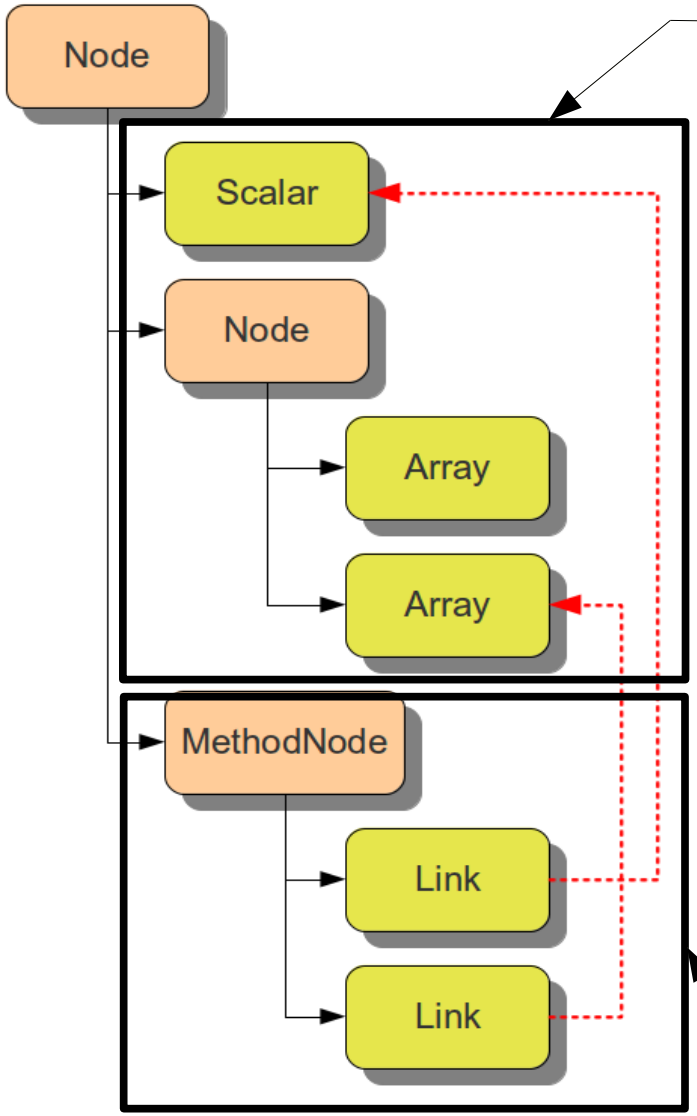
File-Node provides only a method to create Node-objects!

```
File *f = new File("test.pnif");
Node *n;
Array *a;

n = f->createNode(...); //works
a = f->createArray(...); //does not exist

a = n->createArray(...); //works
```


DESY

Standardization facility: the Method-Node

Need a method how to standardize methods (semantic standardization):



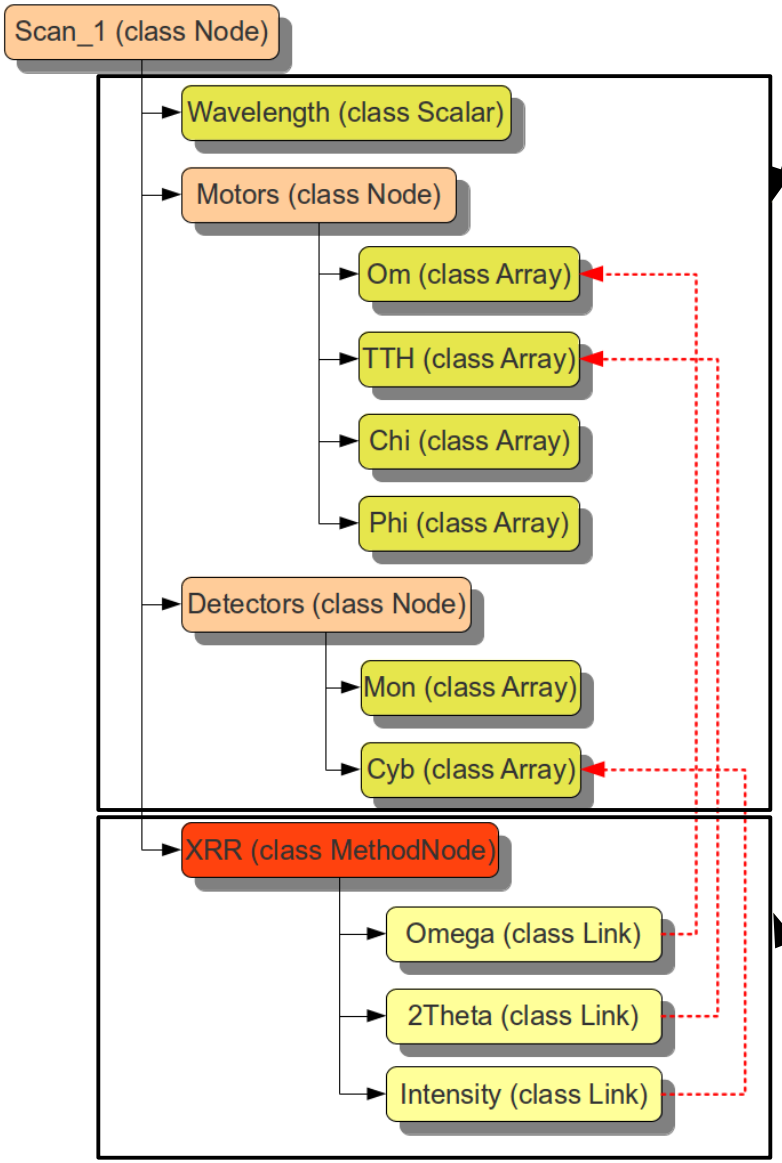
Objects have arbitrary names

- Objects in the data tree can have arbitrary names
- MethodNode – a special node: Objects below this node have fixed and standardized names
- An application requesting data for a standardized method will find this data item by its standard name below the corresponding MethodNode
- Multiple methods in a single tree are allowed

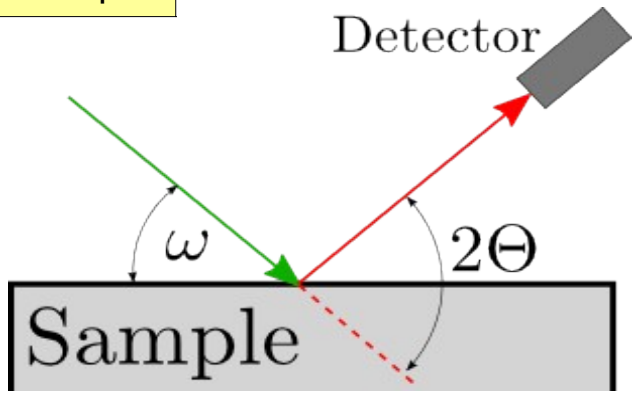
Objects have standardized names

DESY

X-ray reflectometry – a simple example



Beamline dependent part



Advantages of this concept:

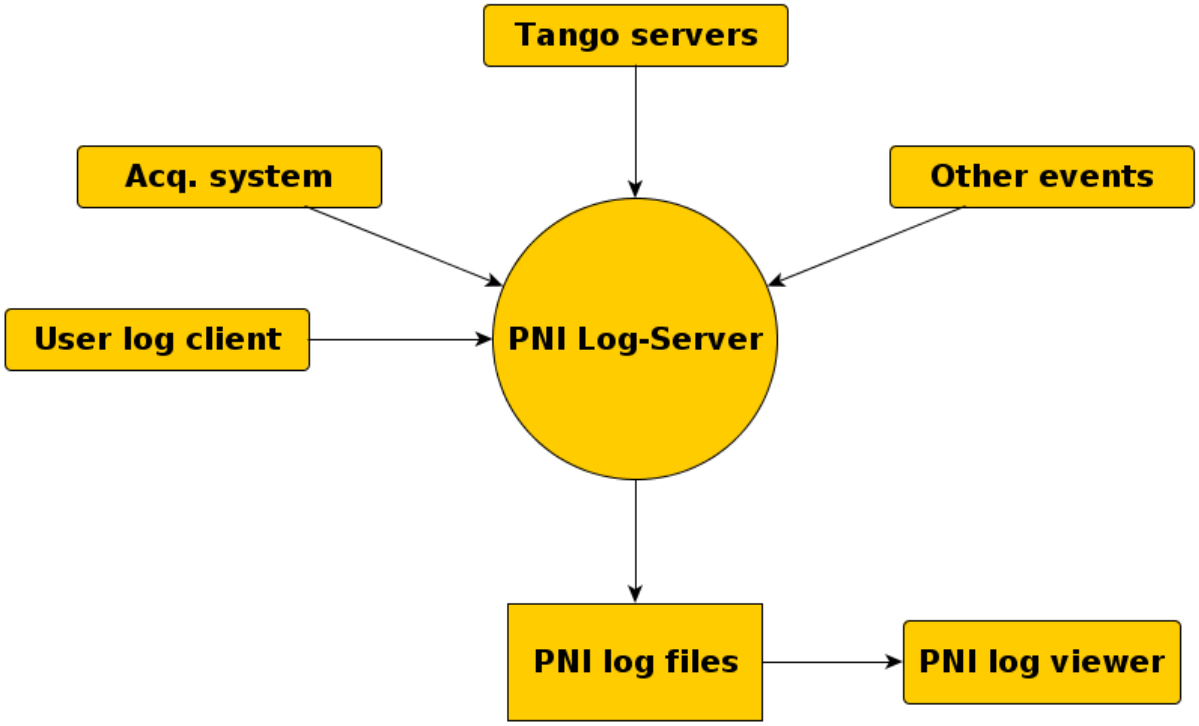
- Beamline scientists can layout data in a way appropriate for them
- Standard applications can still obtain data by standardized name from defined locations

Standardized part

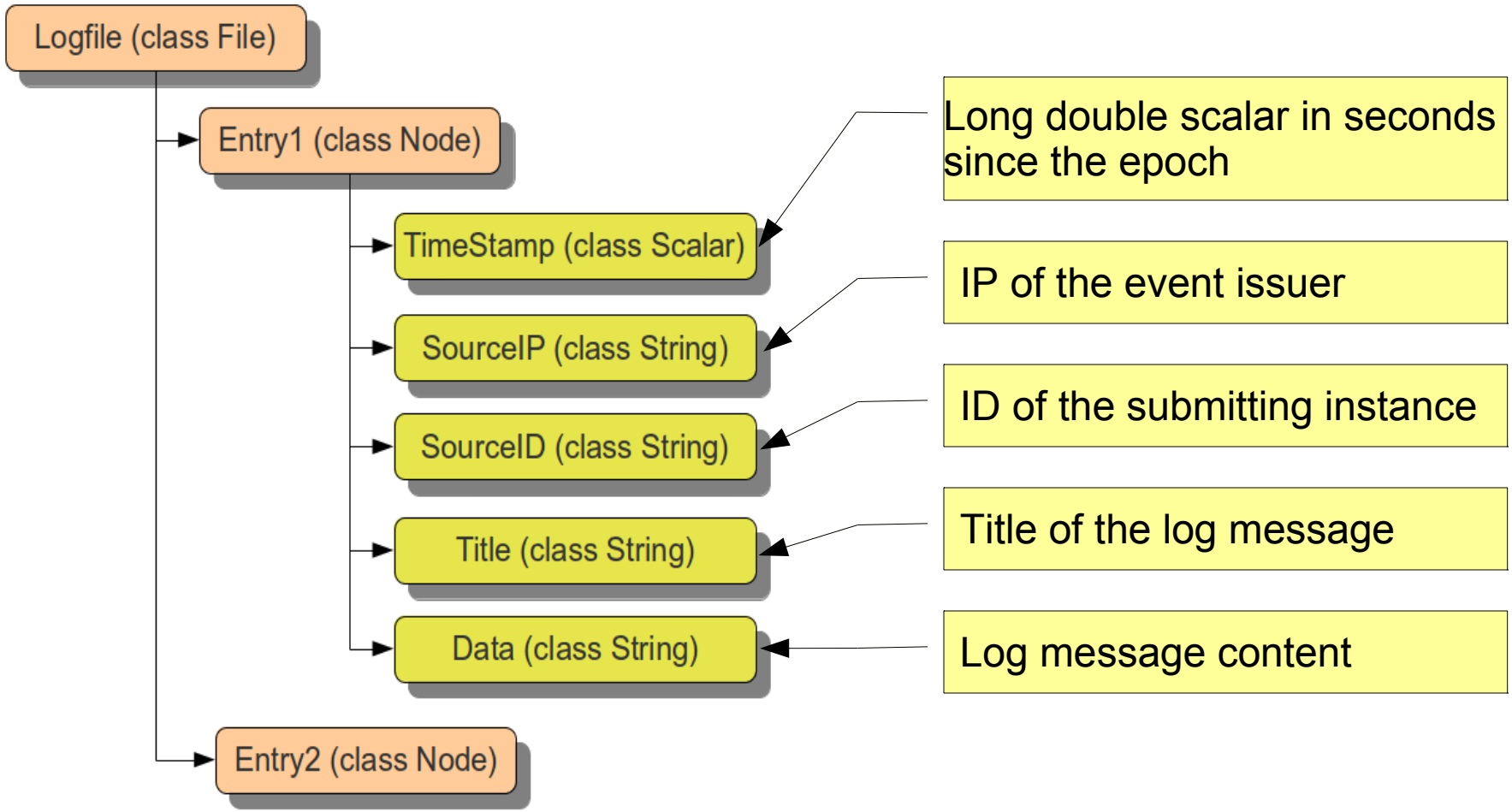
The log system

The logging system

- Log data from different sources is gathered by a logging server and dumped to a file!
- File rotation prohibits log-files from growing too large
- Logging system and data acquisition system are distinct instances!

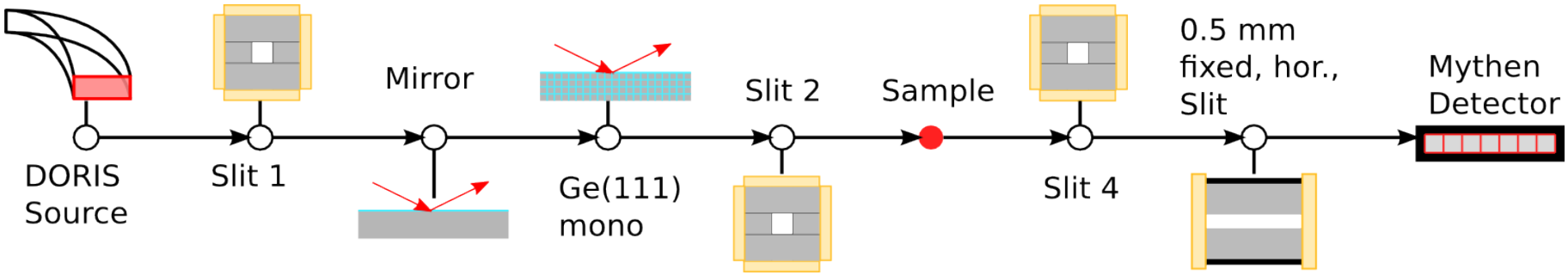


The Log-file structure



Log example: beamline setup

Idea: store beamline elements along the beam path



- Such a sketch can be produced by a rudimentary SVG editor using some standard symbols
- SVG stream can be stored in the data section of a log entry
- SVG can be rendered by a lot of web-browsers and image viewers

Standardization process

How to standardize a method?

Question: what quantities are needed to define a standard?

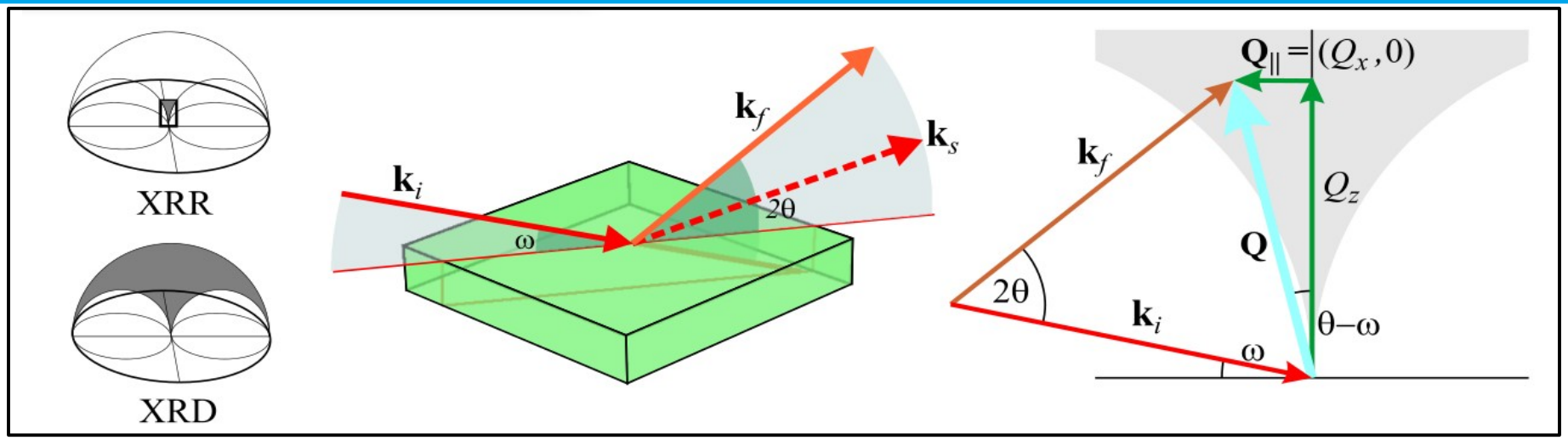
- ***Q*** a vector of quantities needed by the scientist to evaluate the data
- ***E*** a vector of quantities accessible by the experiment
- Units of the components of ***Q*** and ***E***
- *T[]* a transformation with ***Q***=*T[**E**]*

A method *M* can thus be defined as *M:={**Q**,**E**,T[]}*

Advantage of this approach:

- Completeness can be verified by evaluating *T[]*
- Not only the quantities (names) are defined but also their relation to each other
- Can be cited by users if published

An application to XRR and XRD



T[]:

$$k = \frac{2\pi}{\lambda}$$

$$I(q_x, q_z) = \frac{I(\omega, 2\Theta)}{\tau \, monitor}$$

$$q_x = 2k \sin\left(\frac{2\Theta}{2}\right) \sin\left(\omega - \frac{2\Theta}{2}\right)$$

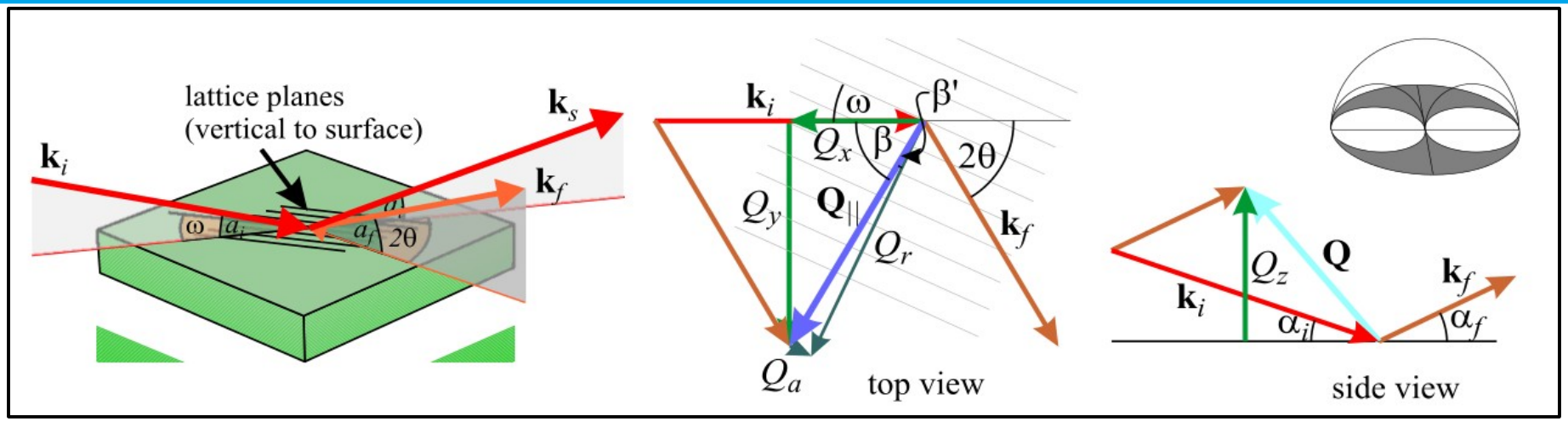
$$q_z = 2k \sin\left(\frac{2\Theta}{2}\right) \cos\left(\omega - \frac{2\Theta}{2}\right)$$

E

Q

quantity	unit
ω	(°)
2Θ	(°)
I(ω,2Θ)	(counts)
τ	(seconds)
monitor	(a.u.)
λ	(Å)
q_x	(1/Å)
q_z	(1/Å)
$I(q_x,q_z)$	(a.u.)

Applicatoin to Gracing Incidence Diffraction (GID)



T[]:

$$k = \frac{2\pi}{\lambda}$$

$$I(q_x, q_z) = \frac{I}{\tau \, monitor}$$

$$q_x = -q_m \cos(\beta)$$

$$q_y = -q_m \sin(\beta)$$

$$q_m = k\sqrt{\cos^2(\alpha_i) - 2\cos(\alpha_i)\cos(\alpha_f)\cos(2\Theta) + \cos^2(\alpha_f)}$$

$$\sin(\beta) = \frac{k\cos(\alpha_f)\sin(2\Theta)}{q_m}$$

E

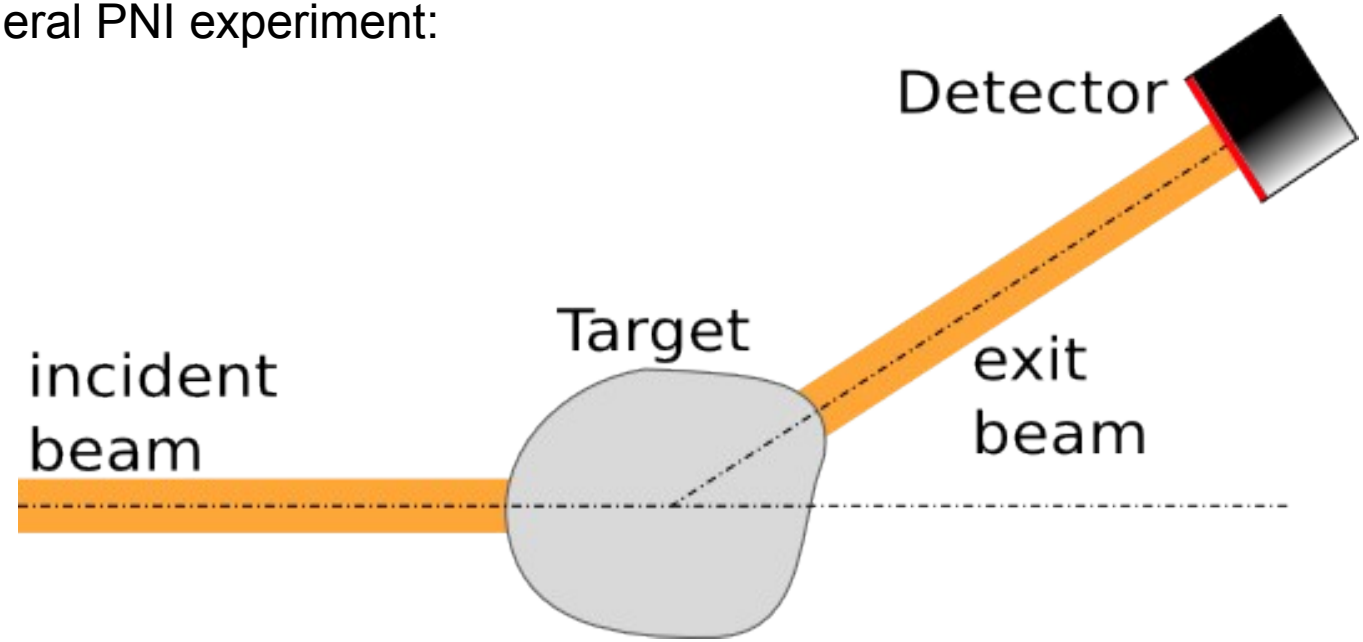
quantity	unit
ω	(°)
2Θ	(°)
I(ω,2Θ)	(counts)
τ	(seconds)
monitor	(a.u.)
λ	(Å)
α_i	(°)
α_f	(°)

Q

quantity	unit
q_x	(1/Å)
q_y	(1/Å)
q_z	(1/Å)
$I(q_x,q_y,q_z)$	(a.u.)

The most general PNI scattering experiment

In order to define standards we need to define positions in space → consider the most general PNI experiment:

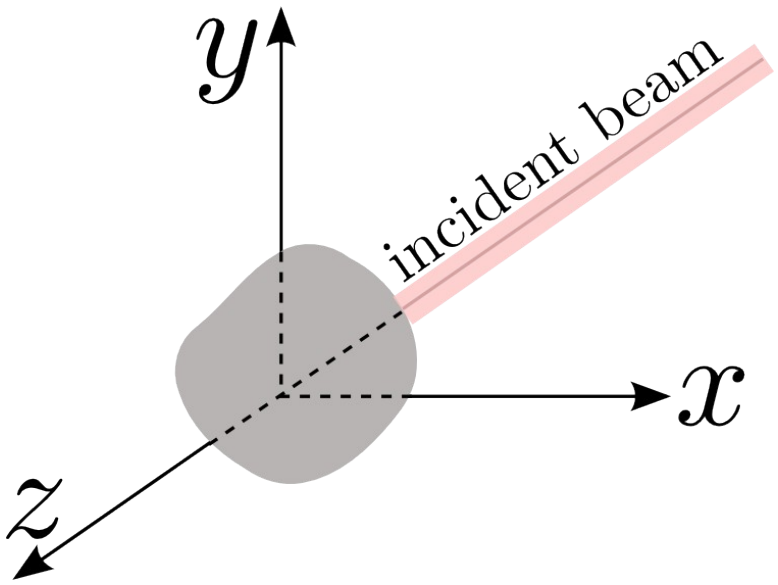


- Incident beam of probe particles (photons, ions, neutrons) impinges on the target
- Target = usually the sample under investigation
- Exiting particles (as a reaction on the incident beam) are recorded by a detector

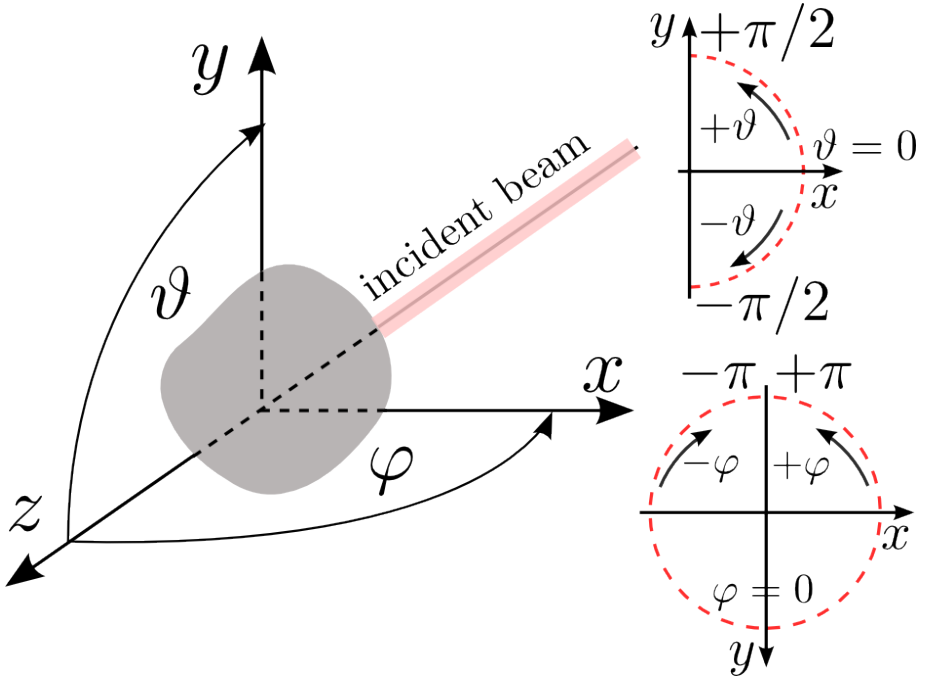
Positions with reference to the target are important!

Coordinate Frames

Cartesian coordinate frame



Spherical coordinate frame

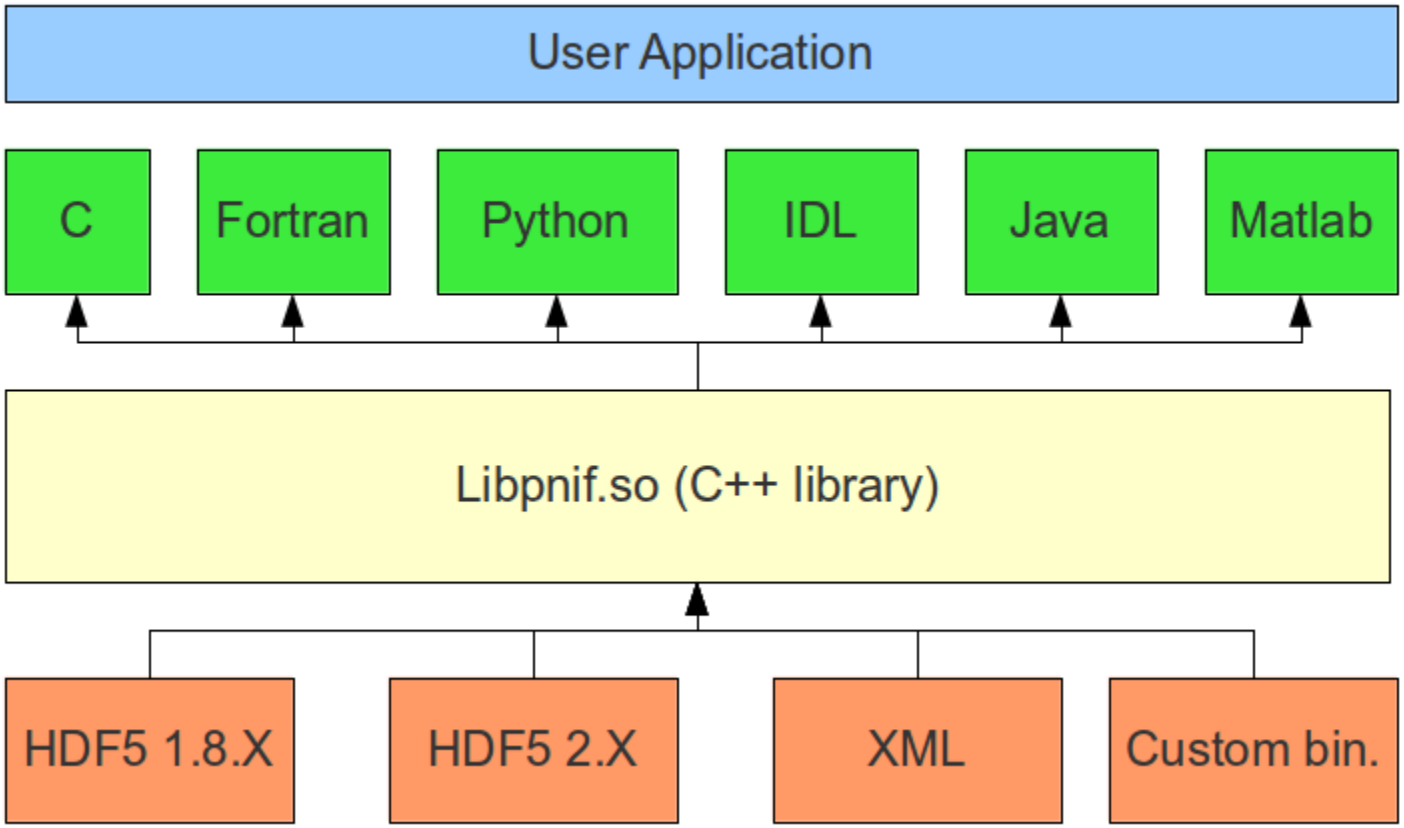


- Origin is defined by the scattering center
- Z-axis is defined by the direction of the beam directly before the sample
- A unique transformation exists between the two systems

Implementation details

Implementation – Part I

File format will be implemented as a C++ library with bindings to many different languages



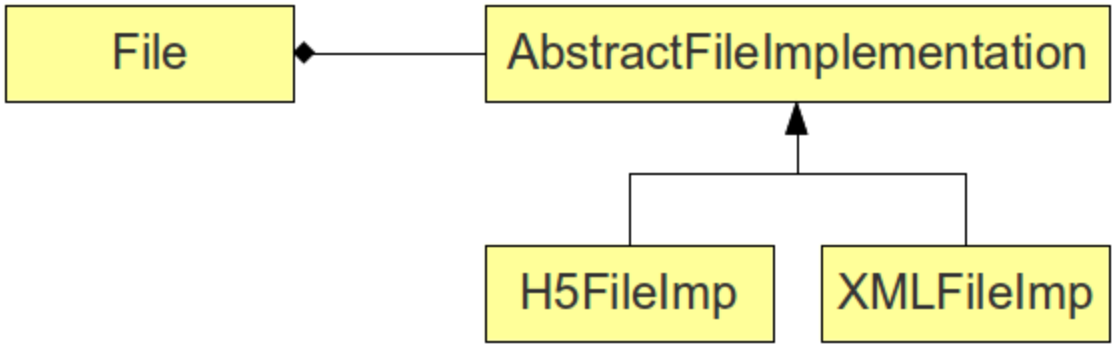
Physically data is stored in a particular format provided by the backend!

Implementation – Part II

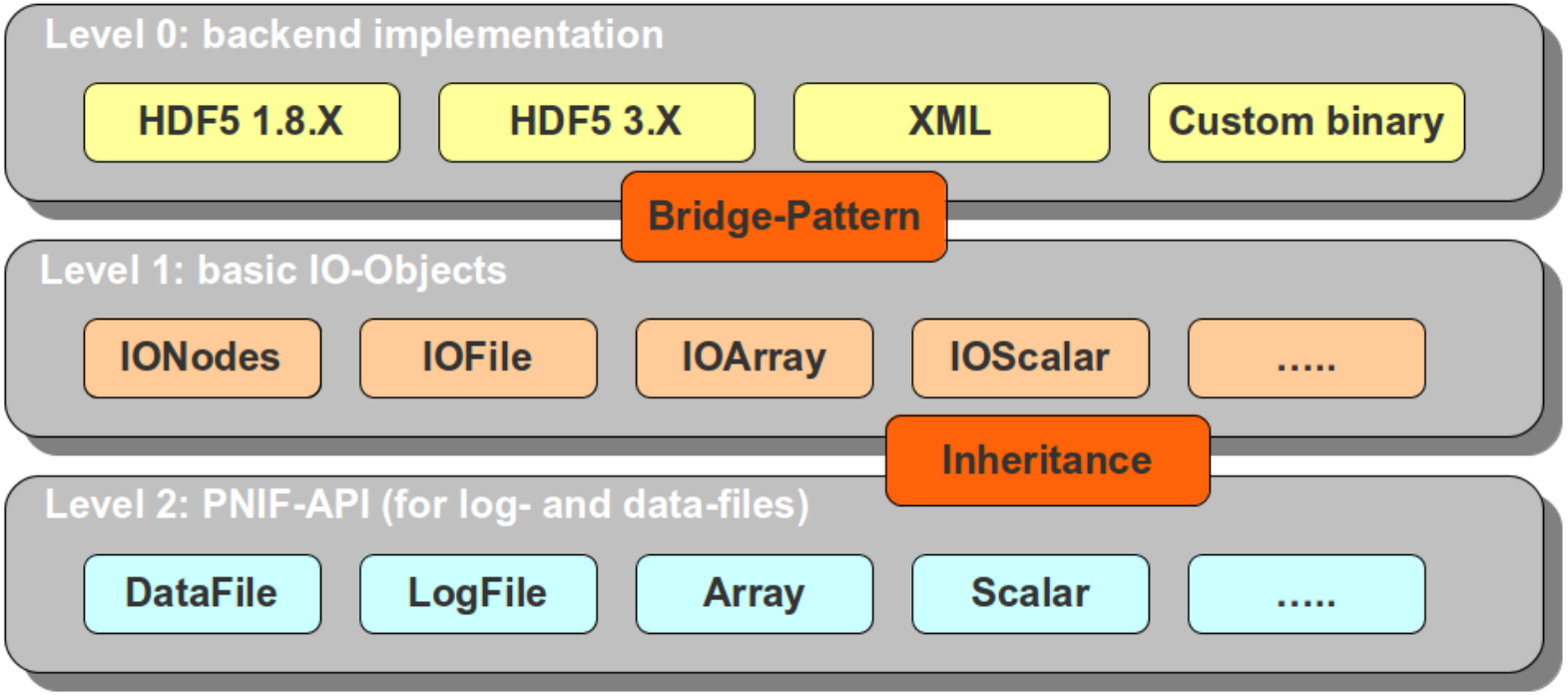
Why different backends? Why not simply HDF5?

- Do not want to depend on a particular physical file format
- HDF5 API may changes → do not want to change application code
- Custom format maybe necessary for performance reasons
- ...

Use a Bridge pattern (see GOF) to connect to the backend → can develop backend code and user PNIF-API independently!



Implementation – Part III



Level 0 → reads and writes data to disk (backend Implementation)

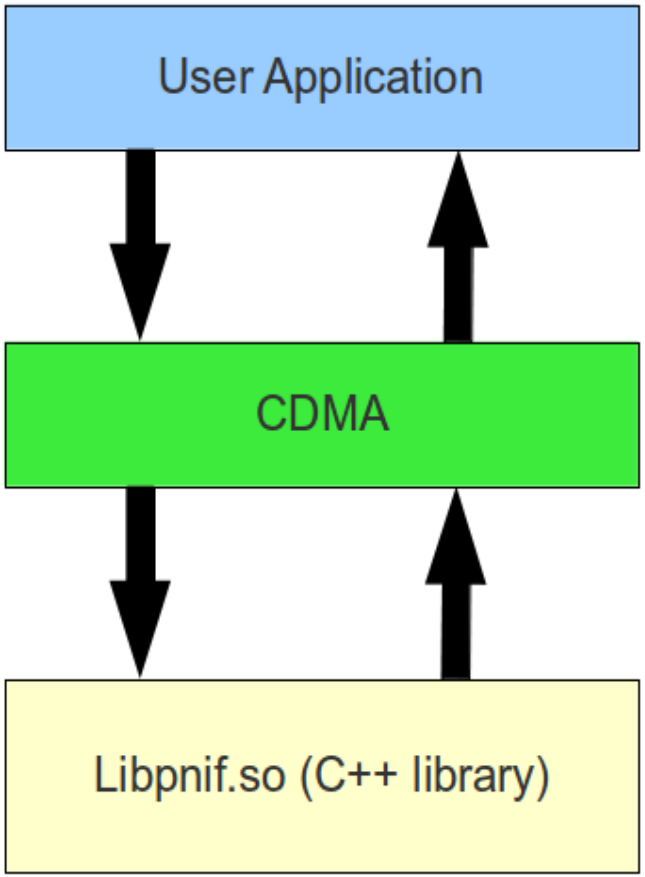
Level 1 → raw IO objects – no limitations in nesting, no required attributes

Level 2 → PNIF user API – including mandatory attributes for objects and object nesting restrictions.

Where do I see PNIF and CDMA (SOLEIL)?

CDMA gives PNIF unaware applications access to PNIF data files!

- CMDA allows full read/write access
- Will be available in C++ in future
- C++ will allow bindings from CDMA to other languages



Conclusion

- PNIF is independent of the physical file format
- Thin – wrapper => should obtain nearly native performance of the backend format
- Only a few classes of objects → easy to use and to learn
- Backend and user API can be developed independently → easy to maintain
- C++ allows the implementation of bindings to a lot of other languages
- Provides facilities to implement standard methods
- Provides a procedure to define standard methods

Thank you for your attention …