NeXusBeans: object-oriented software components for the NeXus scattering science data format using Java, UML, and XML Schema technologies

Authors: Dr Darren Kelly, Dr Nick Hauser

Version date: 23/09/07

!!! DRAFT ONLY (distribution to NIAC members welcome) !!!

The Bragg Institute in partnership with the

Australian Nuclear Science and Technology Organisation (ANSTO)

Abstract: For the first time truly object-oriented software components for the NeXus neutron science data format have been made possible with the aid of metamodel-driven software engineering, highly automated conversion of NeXus XML templates into Java **NeXusBean** classes, and graphical Unified Modelling Language (UML) software engineering as a bridge to an XML Schema for NeXus. NeXusBeans generated from NeXus XML templates will be presented in different forms: as Java code; as reverse-engineered UML; as XML Schema elements. The supporting classes of the NeXusBeans system - such as the universal NXobject inheritance base - will be presented. Construction of a Java NeXusBean representation of the Platypus Reflectometer neutron beam instrument will be shown - both as Java components and as graphical UML structures - along with a demonstration of completely validated serialization and deserialization using XML streaming technologies. Recommendations for improvements to the underlying NeXus XML templates to support further automated object-orientation will be made. The authors make a case for migration of the NeXus format to XML Schema technology via Java NeXusBeans and UML. The possible pros and cons of leveraging the Eclipse Modelling Framework (EMF) will be discussed.

Table of Contents

1) Glossary of terms, acronyms, and technologies

Neutron Beam Instrument (NBI): an instrument that takes advantage of the physics of Bragg Scattering of neutrons incident on a sample to reveal structure and properties of the studied sample. Variations include: diffractometers, reflectometers, and Small Angle Neutron Scattering (SANS) machines.

NeXus: The neutron, x-ray, and muon scattering science data format hosted at: http://www.nexusformat.org/

The NeXus International Advisory Committee (NIAC): meets roughly annually to discuss and ratify changes to the NeXus classes, NeXus instrument definitions, and the metaformat.

Meta-DTD: An XML "template" used to define the NeXus neutron science format according to the rules of the metaformat. Each NeXus class is specified by a single Meta-DTD XML file.

NeXML: A web site for presenting Open Source NeXusBeans technologies and free downloads, maintained by the primary author or this report, and hosted at: http://www.webel.com.au/nexml.

JavaBeans: the property-based component architecture for the Java 2 Platform, Standard Edition (J2SE). based on the JavaBeans specification. Classes that meet the JavaBeans design rules can be used immediately in a huge range of powerful JavaBeans software engineering tools. Visit also: http://java.sun.com/products/javabeans/ and http://java.sun.com/products/javabeans/docs/spec.html

Unified Modelling Language (UML): a metamodel-based graphical software engineering and general modelling and analysis language, developed by the Object Management Group (OMG), and hosted at http://www.uml.org/. Considered by many the *lingua franca* of information technology.

XML Schema: a powerful schema language for the Extensible Markup Language (XML) developed by the W3C and hosted at the web site: http://www.w3.org/XML/Schema. XML instance documents can be validated against XML Schema files using standalone validators and within many modern IDEs. XML Schema files can be graphically represented in many XML tools, or reverse-engineered into graphical UML models.

Netbeans IDE: a Java-based integrated development environment: http://www.netbeans.org

Eclipse IDE: a Java-based integrated development environment: http://www.eclipse.org

Eclipse Modelling Framework (EMF): "a modeling framework and code generation facility for building tools and other applications based on a structured data model. From a model specification described in XMI, EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor". http://www.eclipse.org/modeling/emf/

Magicdraw UML: a powerful commercial UML tool used by the primary author of this report: http://www.magicdraw.com

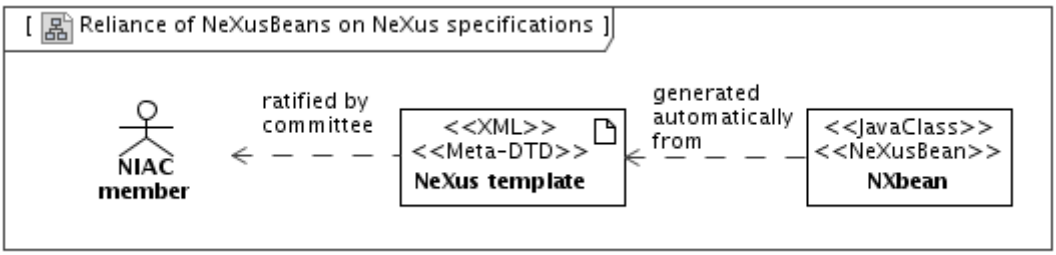
XStream: A customisable XML serialization/deserialization system for Java offered through: http://xstream.codehaus.org/index.html

ANSTO: The Australian Nuclear Science and Technology Organisation: http://www.ansto.gov.au

OPAL: A multi-purpose research reactor at ANSTO that acts as a neutron source for various NBIs: http://www.ansto.gov.au/discover/opal_reactor.html

2) Introduction to NeXusBeans

NeXusBeans are based on technologies developed by Dr Darren Kelly from May 2005 to Sep 2007 within the Bragg Institute's NBI Computing and Electronics Group at the OPAL Research Reactor at ANSTO, Sydney, Australia, under supervision of project manager and NIAC member Dr Nick Hauser. A record of a functioning prototype of the NeXusBeans system was presented by Dr Kelly to the NIAC as UML diagrams at the NIAC2006 in Grenoble, France, at the Institut Laue-Langevin (ILL). NeXusBeans are generated from the XML template files of the NeXus specifications, and therefore value-add to the NeXus neutron science data format effort.



Until NeXusBeans there was no true validation w.r.t. the NeXus data format, and no use of automated serialization technologies to simplify reading/writing of NeXus files. NeXus is defined by NIAC as XML files known officially as *Meta-DTDs*, although they are better described simply as *XML templates*, and that is the term that shall be used here. Rather than an industry-standard schema specification, NeXus employs some seemingly ad-hoc attribute strings and conventions[1], along with a set of design rules given on the NeXus web site, which rules are in part difficult to enforce electronically, as shall be demonstrated. Software readers/writers for the NeXus format relied until now on "string programming" against the specified XML attributes, and had to be tediously maintained each time NeXus templates were changed to correspond "by eye" to the NeXus XML templates, an extremely inefficient and error prone practice[2].

The NeXusBeans project - employing Java, UML, and the XML Schema technologies - attempts to elevate the NeXus data format from a largely ad-hoc specification in XML as "templates" to a true digital data format that supports XML Schema validation and industry standard notations (such as inheritance in UML and XML Schema), file formats, and schema specifiers, while also providing object-oriented Java bindings.

The main steps employed in the generation of Java NeXusBeans are as follows:

- the NeXus XML templates are parsed (using Java) and interpreted. This requires detailed knowledge of the design rules and attribute strings of the NeXus XML templates.
- a minimal, UML-like Java metamodel - just sufficient for modelling and writing packaged Java classes as code - is used to store class representations of the NeXux XML templates.
- Each XML template element is interpreted as inheriting from either an `NXgroup` class, or an `NXDataItem` class, which in turn inherit directly or indirectly from a universal `NXObject` inheritance base[3].
- Each instance of the Class <<metaclass>> of the Java metamodel is able to write itself as packaged Java code, thus facilitating a simple yet effective form of forward engineering[4].

1 They are also rather inconsistently used in the NeXus XML templates, since there was until now no systematic electronic screening of the templates to detect inconsistencies; NeXusBeans technologies help solve that, too.
2 It is unlikely that any substantial NeXus files anywhere, ever, corresponded to the latest NeXus XML templates.
3 This is the first known use of inheritance and the universal NXobject within the NeXus community.
4 Advantages of decoupled class writers that navigate a language-neutral metamodel instance will be discussed.

Indeed the entire process has been automated and can be performed live via the internet from the remote NeXus SVN repository direct to any user's client computer, ensuring that a user has completely valid and up-to-date NeXusBeans that correspond to the NeXus XML templates (as far as those templates can be accurately interpreted). The system has been bundled for download as NeXusTools on the NeXML web site, and will be further demonstrated and explained below.

As a result of the encapsulation of NeXus as Java classes one can now, for the first time ever:

- Build NeXus models of NBIs and neutron beam experiments with validation w.r.t. NeXus
- Take advantage of validated prompting facilities in Java IDEs to assist coding, which is an extremely powerful and time-saving feature.
- Serialize NeXus models using automated XML serialization technologies (see below for the NeXusBeans XML I/O system), without having to maintain tedious string I/O manually.
- Analyse NeXus graphically as UML using reverse-engineered Java classes.
- Build NeXus model instances graphically using UML.
- Transform reverse-engineered UML NeXusBeans to an XML Schema for NeXus (to the extent that the sometimes cryptic NeXus specifications can be interpreted).

The effectiveness of the technique is limited by the degree to which the NeXus specification as XML templates can be parsed and interpreted; the urgent need for reworking of the sometimes cryptic NeXus metaformat to support digital interpretation is thus a recurring theme of this report.

Please note that while the paper provides solutions based on Java and Javabeans, it is in principle possible to implement the UML representations in C++. (Indeed, the UCAR `MultiArray` API employed in the `NXmultiDataItem` base class is available in both Java and C++.) If the NeXus class system is successfully translated/adapted to XML Schema form, then both Java and C++ can leverage that XML Schema, and bean-like C++ bindings to the schema classes could be generated.

Likewise, while the examples provided in the paper are for neutron scattering, the NeXusBeans strategy applies as well to other scattering sciences for which the NeXus data format may be used.

2.1) Assumed knowledge about information technologies used here

Readers should be familiar with the design principles of NeXus, and should understand the difference between a *Data Group*, a *Data Item*, and a *Data Attribute*. Please review the guidelines at http://www.nexusformat.org/Design.

Although this report provides common-sense examples throughout, the underlying technologies used will not be explained in detail. Readers should be familiar with the following technologies required for modern data modelling and software engineering:

- As the NeXus format is specified using XML some knowledge of the Extensible Markup Language (XML) is required. A good starting point is the W3C XML site: http://www.w3.org/XML/.
- Some familiarity with Java is assumed, and especially with JavaBeans concepts such as setters/getters, and how bean properties may or may not relate to underlying fields. Most readers familiar with an object-oriented language will be able to understand the Java examples given. Readers should know that Javadoc is generated from Java code as HTML.
- It is assumed that the reader has some knowledge of the Unified Modelling Language

(UML)[5], an indispensible aid in modern data modelling and software engineering, which is often called the *lingua franca* of IT. Although the work presented here is largely "UML-driven", it is hoped that non-UMLers will be able to glean much from the simple UML diagrams used throughout this report. End users do not need to understand UML to use the Java NeXML tools to generate Java NeXusBeans !

- Some familiarity with modern Integrated Development Environments (IDEs) like Netbeans, Eclipse, or JDeveloper is desirable.
- Some familiarity with the Eclipse Modelling Framework (EMF) is desirable.

Lastly, of course readers should have a basic knowledge of the functioning of a neutron beam instrument, and how Bragg scattering experiments are performed.

5 The authors will NOT be explaining the meaning of any UML symbols used; please see the UML2.1.1 superstructure and/or one of the many books or online tutorials on the UML.

3) NeXusBeans by example

Before expanding on the details of the software engineering technologies used to generate and transform NeXusBeans, examples will be given to assist the reader in later sections.

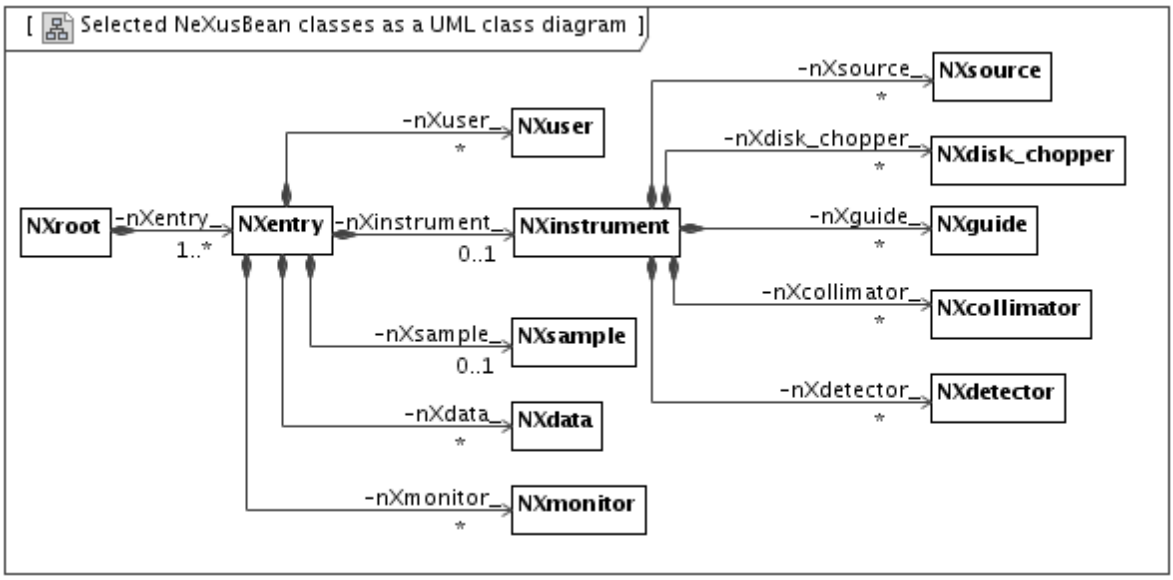


Figure 1: UML overview of some NeXusBean classes generated from NeXus class templates

Figure 1 gives an overview of some NeXusBean Java classes that have been reverse engineered as graphical UML. It provides the first ever graphically engineered view of NeXus. Only the relationships between the classes are shown, i.e. the attributes and operations are not shown. Already this simple diagram demonstrates the power of UML-driven software engineering and data modelling. We will now examine the NeXus XML templates, Java NeXusBeans code, detailed UML diagrams, and XML Schema representation for some selected classes from this overview.

3.1) NeXus XML templates (a.k.a. "Meta-DTD")

We inspect some NeXus XML templates that will be transformed to NeXusBeans. We examine a few aspects (only) of the anatomy of a NeXus XML template, those most pertinent to the NeXusBean conversion process. A full description of the rules for NeXus XML templates can be found at:

`http://www.nexusformat.org/Metaformat`

The following definition of a disc chopper was taken from:

`http://www.nexusformat.org/NXdisk_chopper`

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXdisc_chopper.xml
Editor:  Mark Koennecke
$Id: NXdisk_chopper.xml 4 2005-07-19 04:10:26Z rio $

Template of NXdisc_chopper.

-->
<NXdisk_chopper name="{chopper_name}">
        <type type="NX_CHAR">
                {Chopper type single|contra_rotating_pair|synchro_pair}?
        </type>
        <rotation_speed type="NX_FLOAT" units="rpm|hertz">
                {chopper rotation speed}?
        </rotation_speed>
        <slits type="NX_INT">
                {Number of slits}
        </slits>
        <slit_angle type="NX_FLOAT" units="degree">
                {angular opening}
        </slit_angle>
        <pair_separation type="NX_FLOAT" units="cm">
                {disc spacing in direction of beam}?
        </pair_separation>
        <radius type="NX_FLOAT" units="cm">
                {radius to centre of slit}
        </radius>
        <slit_height type="NX_FLOAT" units="cm">
                {total slit height}
        </slit_height>
        <phase type="NX_FLOAT" units="degree">
                {chopper phase angle}?
        </phase>
        <ratio type="NX_INT">
                {pulse reduction factor of this chopper in relation to other
choppers/fastest pulse in the instrument}?
        </ratio>
        <distance type="NX_FLOAT" units="cm">
                {Effective distance to the origin}?
        </distance>
        <wavelength_range type="NX_FLOAT[2]" units="nm">
                {low and high values of wavelength range transmitted}?
        </wavelength_range>
        <NXgeometry name="">
                {geometry of the disk chopper}?
        </NXgeometry>
</NXdisk_chopper>
```

We can identify already some potential difficulties that the NeXus templates present to the NeXusBean generator's parser and interpreter.

The metadata buried in the header comment presents an issue for the NeXusBeans parser, so a simple solution is to merely propagate the header comment as is through to the Java NeXusBean as Javadoc documentation without further interpretation.

It will be shown that the tedious (and redundant) restatement of a `name` attribute such as:

```
<NXdisk_chopper name="{chopper_name}">
```

is much better handled by a single `name` attribute inherited from a universal `NXobject` base. The `name` value `'{chopper_name}'` adds no useful information, whereas the template's class name provides a useful context for the name anyway, so the NeXusBeans generator ignores the redundant name documentation. The `name` attribute of a NeXusBean is simply its inherited `NXobject` name.

The `<type>` of the `<NX_diskchopper>` is documented inline as:

```
{Chopper type single|contra_rotating_pair|synchro_pair}
```

which indicates a kind of enumeration of chopper types, however it is not clearly defined as such. By comparison, XML Schema language supports explicit enumeration definitions.

The units for the chopper speed also imply a kind of enumeration inline in the XML attribute value:

```
<rotation_speed type="NX_FLOAT" units="rpm|hertz">
```

Validation against such ad-hoc specifications in existing NeXus reader/writer pairs is extremely error prone. And if the enumeration is extended, the reader/writer pair needs to be manually updated ! Further, the possible units are not derived by reference to a units library (with namespace). In an XML Schema this could be well handled by either inline enumeration specification in the XML Schema, or by asserting that the units should be handled as a separate XML element with units definitions imported from an existing schema. There are surely no units specific to neutron science !

The type of the chopper's rotation speed data item value is given as:

```
NX_FLOAT
```

To enforce validation against correct types one would do well to use an XML Schema with clear specification of valid values. In the case above the type information is easily mapped to Java. However, in many NeXus XML templates one has choices such as:

```
NX_FLOAT|NX_INT
```

Further, one has the possibility of dimensioned data, such as:

```
<wavelength_range type="NX_FLOAT[2]" units="nm">
```

One important recommendation of this report concerning the existing XML templates will be that the specification of dimensionality data should be separated from the type specification, to afford clearer specification, easier XML parsing, and easier mapping, thus:

```
<wavelength_range type="NX_FLOAT" dimension=″2″ units="nm">
```

This matter is of such importance that it is the subject of a dedicated section below.

Multiplicity (a.k.a. occurrence) information in NeXus XML templates is currently stored within the XML element's text (often after the template's comment text), using indicators similar to UNIX regular expressions, thus:

```
<NXgeometry name="">
        {geometry of the disk chopper}?
</NXgeometry>
```

The trailing question mark '?' here indicates that the geometry group:

'may occur 0 or one times (i.e. no more than once)'[6].

These multiplicity indicators are tricky to parse, and it is recommended that the occurrence information be instead given as the XML Schema-like attributes `minOccurs` and `maxOccurs`.

The NeXus multiplicities are easily mapped to the UML system: `MultiplicityElement, MultiplicityKind`. However, they are not easily enforced within Java code without generating quite complex code within setters and constructors and an associated exception handling system. The Java annotations approach employed in the Eclipse Modelling Framework (EMF)[7] lends itself as an alternative solution, indeed the NeXusBean's `NXobject` base could be made an `EObject`, thus attaching NeXusBeans to the power of the EMF system, a strategy to be discussed further.

Currently, NeXusBeans propagate the NeXus multiplicity information as Javadoc comments only. Elements that may not occur more than once are represented as simple bean properties with a getter/setter pair, while those elements that may occur more than once are generated as indexed bean properties with indexed setters, getters, and adders (appenders).

An important NeXus class is the top-level `NXroot` group listed below. It can contain one or more `NXentry` groups, and it MUST contain at least one, as indicated by the '+' which means, according the NeXus metaformat:

'may occur one or more times (i.e. at least once)'.

It has a number of special *global attributes* not found in other nexus groups. The NeXusBeans generator currently maps all of these as string attributes (although the times should be constrained as ISO 8601 times according to the NeXus Design).

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXroot.xml
Editor:  NIAC
$Id: NXroot.xml 13 2006-09-27 13:44:48Z faa59 $

Definition of the root NeXus group.

-->
<NXroot file_name="{File name of original NeXus file}"
        file_time="{Date and time of file creation}"
        file_update_time="{Date and time of last file change at close}"
        NeXus_version="{Version of NeXus API used in writing the file}"
        HDF_version="?"
        HDF5_version="?"
        creator="{facility or program where file originated}?">
        <NXentry name="{entry name}">
            +
        </NXentry>
</NXroot>
```

6 http://www.nexusformat.org/Metaformat

7 http://www.eclipse.org/emf

Since NeXus is used for recording data from neutron scattering experiments, another important NeXus class is NXdata:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXdata.xml
Editor:  NIAC
$Id: NXdata.xml,v 1.2 2005/07/19 04:10:26 rio Exp $

Template of plottable data and their dimension scales. It is mandatory that
there is at least one group of this type in each NXentry group.  Note that
"variable" and "data" can be defined with different names.  The "signal" and
"axes" attribute of the "data" item define which items are plottable data and
which are dimension scales.

-->
<NXdata name="{Name of data block}">
      <variable type="NX_FLOAT[:]|NX_INT[:]" long_name="{Axis label}" distribution="0|1"
first_good="{Index of first good value}" last_good="{Index of last good value}">
            {Dimension scale defining an axis of the data}?
      </variable>
      <variable_errors type="NX_FLOAT[:]|NX_INT[:]">
            {Errors associated with axis "variable"}?
      </variable_errors>
      <data type="NX_FLOAT[:...]|NX_INT[:...]" signal="1" axes="[...]" long_name="{Title
of data}">
            {Data values}?
      </data>
      <errors type="NX_FLOAT[:...]">
            {Standard deviations of data values - the data array is identified by the
attribute signal="1". This array must have the same dimensions as the data}?
      </errors>
</NXdata>
```

The NeXusBean generator will now have to cope with type specifications such as:

```
type="NX_FLOAT[:]|NX_INT[:]"
type="NX_FLOAT[:...]|NX_INT[:...]
```

and attribute definitions like:

```
axes="[...]"
```

The colon is described thus in the NeXus Metaformat:

> *"Use a colon if the dimension length is not defined by the DTD"*

However the triple-dot ellipsis '...' is only mentioned in another context:

> *"Replace the colon with i, j, ... if you wish to match the dimension length to other data items within the same group"*

A degree of human interpretation and guesswork is required to assist the NeXusBeans generator.

We now examine a far more complicated NeXus template for the crucial detector class, a challenge for the NeXusBean generator:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXdetector.xml
Editor:  NIAC
$Id: NXdetector.xml 28 2006-10-06 21:00:24Z pfp $

Template of a detector, detector bank, or multidetector. The indices
require explanation:
  np - the number of points in a scan. This dimension is only present
     in scanning measurements.
  tof - the number of points in a time-of-flight axis. This is only present
     in time-of-flight measurements.
  i - the number of pixels in the slowest varying direction. This is
     only present in missing in the point detector.
  j - the number of pixels in the fastest varying direction. This
     exists only in "area" detectors.

HOUSEKEEPING:
 - remove all question marks
 - fix units
 - polar, azimuthal, distance, and wavelength need description
 - remove generic attributes
-->
<NXdetector name="{Name of detector bank}">
        <time_of_flight type="NX_FLOAT[tof+1]" axis="3" primary="1?" long_name="{Axis
label}" units="10^-6 second|10^-7 second" link="{absolute path to location in
NXdetector}">
                {Total time of flight}
        </time_of_flight>
        <raw_time_of_flight type="NX_INT[tof+1]" units="clock_pulses"
frequency="NX_FLOAT:{Clock frequency in Hz}">{In DAQ clock pulses}</raw_time_of_flight>
        <detector_number type="NX_INT[i?,j?]">
                {Identifier for detector}?
        </detector_number>
        <data type="NX_FLOAT[np?,i?,j?,tof?]|NX_INT[np?,i?,j?,tof?]" signal="1"
axes="[number of scan points,x_offset?,y_offset?,time_of_flight?]?" long_name="{Title of
measurement}?" check_sum="{Integral of data as check of data integrity} (NX_INT)?"
units="number" link="{absolute path to location in NXdetector}">
                {Data values}?
        </data>
        <data_error type="NX_FLOAT[np?,i?,j?,tof?]|NX_INT[np?,i?,j?,tof?]"
units="number" link="{absolute path to location in NXdetector}">
                {Data values}
        </data_error>
        <x_pixel_offset axis="1" primary="1?" type="NX_FLOAT[i?]" units="10^-3
meter|10^-2 meter" long_name="{Axis label}" link="{absolute path to location in
NXdetector}">
                {offset from the detector center in x-direction}?
        </x_pixel_offset>
        <y_pixel_offset axis="2" primary="1?" type="NX_FLOAT[j?]" units="10^-3
meter|10^-2 meter" long_name="{Axis label}">
```

```
                {offset from the detector center in the y-direction}?
        </y_pixel_offset>
        <distance type="NX_FLOAT[np?,i?,j?]">
        </distance>
        <polar_angle type="NX_FLOAT[np?,i?,j?">
        </polar_angle>
        <azimuthal_angle type="NX_FLOAT[np?,i?,j?]">
        </azimuthal_angle>
        <description type="NX_CHAR">
                {name/manufacturer/model/etc. information}?
        </description>
        <NXgeometry name="">
                {Position and orientation of detector element}?
        </NXgeometry>
        <solid_angle type="NX_FLOAT[i?,j?]" units="steradians">
                {Solid angle subtended by the detector at the sample}?
        </solid_angle>
        <x_pixel_size type="NX_FLOAT[i?,j?]" units="mili*metre">
                {Size of each detector pixel. If it is scalar all pixels are the same
size}?
        </x_pixel_size>
        <y_pixel_size type="NX_FLOAT[i?,j?]" units="mili*metre">
                {Size of each detector pixel. If it is scalar all pixels are the same
size}?
        </y_pixel_size>
        <dead_time type="NX_FLOAT[np?,i,j?]">
                {Detector dead time}?
        </dead_time>
        <gas_pressure type="NX_FLOAT[i?,j?]" units="bars">
                {Detector gas pressure}?
        </gas_pressure>
        <detection_gas_path type="NX_FLOAT" units="cm">
                {maximum drift space dimension}?
        </detection_gas_path>
        <crate type="NX_INT[i?,j?]" local_name="{Equivalent local term}">
                {Crate number of detector}?
        </crate>
        <slot type="NX_INT[i?,j?]" local_name="{Equivalent local term}">
              " local_name="{Equivalent local term}">{Input number of detector}?</input>
        <type type="NX_CHAR">
                {Description of type such as He3 gas cylinder, He3
PSD, scintillator, fission chamber, proportion counter, sciliation
counter, ion chamber, ...}
        </type>
        <NXdata name="efficiency">
                {Efficiency of detector with respect to e.g. wavelength}
           <efficiency type="NX_FLOAT[i?,j?,k?]" units="">{efficiency
of the detector}</efficiency>
           <wavelength type="NX_FLOAT[i?,j?,k?]" units=""/>
        </NXdata>
```

```
        <calibration_date type="ISO8601">
                {date of last calibration (geometry and/or efficiency) measurements}?
        </calibration_date>
        <calibration_method type="NXnote">
                {summary of conversion of array data to pixels (e.g. polynomial
approximations) and location of details of the calibrations}?
        </calibration_method>
        <layout type="NX_CHAR">point|linear|area{How the detector  is
represented}</layout>
        <count_time type="NX_INT[np?]|NX_FLOAT[np?]" units="">{Elapsed
actual counting time}</count_time>
</NXdetector>
```

By admission of the many header comments this detector class template is in a rather poor state ! The header comments in fact contain extremely important constraints as natural language and pseudo-maths that can barely be applied by a software reader/writer system, and if so then only in rather error prone fashion. In practice it is almost impossible to maintain synchronicity between a reader/writer (and/or builder) and such a specification.

Even with detailed knowledge of NBIs and with the assistance of the documentation the class is hard for humans to understand and difficult for computers to process using prevailing XML technologies. For example, not even the name of the class is clearly defined:

```
<NXdetector name="{Name of detector bank}">
```

So is it a detector, or is it merely a bank within a detector ? A consistent object-oriented model might use a detector composed of one or more banks, with a dedicated bank class. In fact in the `NXinstrument` class that uses the `NXdetector` we find:

```
<NXdetector name="{Name of detector, bank of detectors, or multidetector}">
```

One should not of course have to view the documentation of another client class to find out what the name of a used class means !

The units declaration may now contain flags/variables defined elsewhere:

```
<time_of_flight type="NX_FLOAT[tof+1]" ..
```

Clearly the detector class is trying to be an “animal with everything”, with parameter tricks used to switch between animal types. Classes that attempt to cover all cases after this fashion quickly become bloated and cryptic. This situation is far better handled with classic polymorphism: an `AbstractDetector` class and a hierarchy of abstract/concrete detector classes, such as a `Detector_TOF`[8], which important strategy will be discussed in detail later.

Some parts of the spec seem to have syntax errors, which represent a particular challenge to the NeXusBeans generator. The following element seems to have a misplaced comment (as underlined):

```
<raw_time_of_flight type="NX_INT[tof+1]" units="clock_pulses"
frequency="NX_FLOAT:{Clock frequency in Hz}">{In DAQ clock
pulses}</raw_time_of_flight>
```

while another element has a missing closing bracket in the dimension specifier.

```
<polar_angle type="NX_FLOAT[np?,i?,j?">
```

There are now complicated variable data dimension specifications such as:

```
<solid_angle type="NX_FLOAT[i?,j?]" units="steradians">
```

8 The case for polymorphism was presented inn detail by Darren Kelly at NIAC2006 using the Detector example.

and some that even appear to combine variable data dimensions with a parameter like:

```
<dead_time type="NX_FLOAT[np?,i,j?]">
```

and then even with parameterised, variable dimensions with options:

```
<data_error type="NX_FLOAT[np?,i?,j?,tof?]|NX_INT[np?,i?,j?,tof?]"
```

In the following element an enumeration of possible values is implied:

```
<layout type="NX_CHAR">point|linear|area{How the detector is represented}</layout>
```

One theme of this report is that such cases are far better handled by object-oriented polymorphism.

In the following case the units seem at best strangely defined:

```
<x_pixel_offset axis="1" primary="1?" type="NX_FLOAT[i?]"
units="10^-3 meter|10^-2 meter" long_name="{Axis label}"
link="{absolute path to location in NXdetector}">
```

How might a parser deal with the ungrouped negative powers of 10 ? Surely `10^(-3)` would be a safer construct, and much easier to parse.

The NeXus XML templates currently rely far too much on the unique abilities of humans to interpret “what the author meant”, and that does not scale well to digital parsers and interpreters. Attempts to systematically digitally parse and interpret all of the current NeXus XML templates prove that they not a robust format specification. There are at least three solutions to this problem:

1. Improve the NeXus metaformat so that the templates can be successfully parsed and interpreted digitally by systems like the NeXusBean generator.
2. Extract as much information as possible from the existing templates using the NeXusBean generator, from which an XML Schema can be generated, which interim schema might serve as a fresh starting point for a new NeXus effort based on the XML Schema language.
3. Rewrite the entire NeXus class system from scratch using the XML Schema language, subject to its ability to encapsulate the NeXus design and handle multi-dimensional data.

We now investigate how much information could be extracted to date as Java NeXusBeans, which classes serve as a bridge to UML class models, to an XML Schema[9], or alternatively to an EMF model (which can also then bridge to an XML Schema).

9 Darren Kelly is working in parallel on a separate strategy using XSL Stylesheets to transform the NeXus templates directly to an XML Schema, however so far it has proven far easier to capture the many nuances of the NeXus metaform and templates in Java, and in any case one still needs Java-to-XML bindings, so Java is preferred.

3.2) NeXusBean classes as Java code

We examine some NeXusBeans as Java code generated from the NeXus XML templates presented above by the `NeXusBeanGenerator`. The first listing is the disc chopper class as a Java NeXusBean. Already this simple class corresponds to quite a lot of Java code - counted by lines - and we look forward to the far more concise graphical UML view that follows the few long listing of this section (or view already the class diagram Figure 12) ! Note:

- The NeXusBean class depends on (an older version 2.0 of) the `neutron.nexus` Java API, only to access standard NX type definitions; the reader/writer system is not used.
- The NeXusBean class depends on the Bragg NeXus base, which offers `NXgroup, NXgeometrical,` and `NXdataItem`, which classes inherit directly or indirectly from the universal `NXobject` root.
- This `NXdisk_chopper` NeXusBean extends the `NXgeometrical` class(which in turn extends `NXgroup,` so it inherits automatically an `NXgeometry` child.
- Every data item gets its own inner subclass - within the outer NeXusBean class - that implements `NXdataItem.` `OBSOLETE: now via scalar/vector/multi`
- To `set` an NXdataItem in a NeXusBean means to overwrite the entire data item object !
- The original NeXus template's documentation is propagated as Javadoc
- Additional diagnostics and warnings are given as `// programmatic comments`
- Allowed multiplicities and related errors are NOT enforced in this version, they are merely reported in the Javadoc. (EMF annotations provide a possible solution to this problem.)
- There are no indexed properties in this simple class, so no arrays or lists are required.
- For brevity here many formatting newlines have been removed from the listing.
- Despite the apparent length of the Java code listing for even this simple class, the class is efficient, and Java handles the use of many inner classes easily.

```
package au.gov.ansto.bragg.nexus.auto;
import neutron.nexus.*;
import au.gov.ansto.bragg.nexus.base.*;

//  Generated from XML template(http://svn.nexusformat.org/definitions/trunk/base_classes/NXdisk_chopper.xml)
/**
URL:     http://www.nexus.anl.gov/classes/xml/NXdisc_chopper.xml
Editor:  Mark Koennecke
$Id$

Template of NXdisc_chopper.

 */
 public class NXdisk_chopper
 extends NXgeometrical_ {
/** Property string for bean property 'type' */
final static public String $type = "type";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
```

```
/** {Chopper type single|contra_rotating_pair|synchro_pair}? */
private Type type_;
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {Chopper type single|contra_rotating_pair|synchro_pair}? */
public Type getType(){
 return type_;
}
public void setType(final Type type){
 type_ = type;
}
/** Property string for bean property 'rotation_speed' */
final static public String $rotation_speed = "rotation_speed";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {chopper rotation speed}? */
private Rotation_speed rotation_speed_;
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {chopper rotation speed}? */
public Rotation_speed getRotation_speed(){
 return rotation_speed_;
}
public void setRotation_speed(final Rotation_speed rotation_speed){
 rotation_speed_ = rotation_speed;
}
/** Property string for bean property 'slits' */
final static public String $slits = "slits";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {Number of slits} */
private Slits slits_;
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {Number of slits} */
public Slits getSlits(){
 return slits_;
}
public void setSlits(final Slits slits){
 slits_ = slits;
}
/** Property string for bean property 'slit_angle' */
final static public String $slit_angle = "slit_angle";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {angular opening} */
private Slit_angle slit_angle_;
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {angular opening} */
public Slit_angle getSlit_angle(){
 return slit_angle_;
}
public void setSlit_angle(final Slit_angle slit_angle){
 slit_angle_ = slit_angle;
}
/** Property string for bean property 'pair_separation' */
final static public String $pair_separation = "pair_separation";
// indexed = false
// required = false
```

```
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {disc spacing in direction of beam}? */
private Pair_separation pair_separation_;
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {disc spacing in direction of beam}? */
public Pair_separation getPair_separation(){
 return pair_separation_;
}
public void setPair_separation(final Pair_separation pair_separation){
 pair_separation_ = pair_separation;
}
/** Property string for bean property 'radius' */
final static public String $radius = "radius";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {radius to centre of slit} */
private Radius radius_;
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {radius to centre of slit} */
public Radius getRadius(){
 return radius_;
}
public void setRadius(final Radius radius){
 radius_ = radius;
}
/** Property string for bean property 'slit_height' */
final static public String $slit_height = "slit_height";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {total slit height} */
private Slit_height slit_height_;
// Extracted NeXus multiplicity indicator ( ) means (WARNING: no multiplicity indicator found)
/** {total slit height} */
public Slit_height getSlit_height(){
 return slit_height_;
}
public void setSlit_height(final Slit_height slit_height){
 slit_height_ = slit_height;
}
/** Property string for bean property 'phase' */
final static public String $phase = "phase";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {chopper phase angle}? */
private Phase phase_;
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {chopper phase angle}? */
public Phase getPhase(){
 return phase_;
}
public void setPhase(final Phase phase){
 phase_ = phase;
}
/** Property string for bean property 'ratio' */
final static public String $ratio = "ratio";
// indexed = false
```

```
// required = false
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {pulse reduction factor of this chopper in relation to other choppers/fastest pulse in the instrument}? */
private Ratio ratio_;
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {pulse reduction factor of this chopper in relation to other choppers/fastest pulse in the instrument}? */
public Ratio getRatio(){
 return ratio_;
}
public void setRatio(final Ratio ratio){
 ratio_ = ratio;
}
/** Property string for bean property 'distance' */
final static public String $distance = "distance";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {Effective distance to the origin}? */
private Distance distance_;
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {Effective distance to the origin}? */
public Distance getDistance(){
 return distance_;
}
public void setDistance(final Distance distance){
 distance_ = distance;
}
/** Property string for bean property 'wavelength_range' */
final static public String $wavelength_range = "wavelength_range";
// indexed = false
// required = false
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {low and high values of wavelength range transmitted}? */
private Wavelength_range wavelength_range_;
// Extracted NeXus multiplicity indicator (?) means (May occur 0 or one times (i.e. no more than once))
/** {low and high values of wavelength range transmitted}? */
public Wavelength_range getWavelength_range(){
 return wavelength_range_;
}
public void setWavelength_range(final Wavelength_range wavelength_range){
 wavelength_range_ = wavelength_range;
}
/** constructor */
public NXdisk_chopper(){
 super();
}
static public class Type
 extends NXscalarDataItem_ {
/** constructor */
public Type(String value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_CHAR,value);
}
}
static public class Rotation_speed
 extends NXscalarDataItem_<Float> {
/** constructor */
public Rotation_speed(Float value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_FLOAT32,value);
}
```

```
}
static public class Slits
 extends NXscalarDataItem_<Short> {
/** constructor */
public Slits(Short value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_INT16,value);
}
}
static public class Slit_angle
 extends NXscalarDataItem_<Float> {
/** constructor */
public Slit_angle(Float value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_FLOAT32,value);
}
}
static public class Pair_separation
 extends NXscalarDataItem_<Float> {
/** constructor */
public Pair_separation(Float value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_FLOAT32,value);
}
}
static public class Radius
 extends NXscalarDataItem_<Float> {
/** constructor */
public Radius(Float value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_FLOAT32,value);
}
}
static public class Slit_height
 extends NXscalarDataItem_<Float> {
/** constructor */
public Slit_height(Float value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_FLOAT32,value);
}
}
static public class Phase
 extends NXscalarDataItem_<Float> {
/** constructor */
public Phase(Float value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_FLOAT32,value);
}
}
static public class Ratio
 extends NXscalarDataItem_<Short> {
/** constructor */
public Ratio(Short value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_INT16,value);
}
}
static public class Wavelength_range
 extends NXvectorDataItem_<Float> {
/** constructor */
public Wavelength_range(Float[] value) throws au.gov.ansto.bragg.nexus.base.InvalidNXtypeException {
super(neutron.nexus.NexusFile.NX_FLOAT32,value);
}
}
}
```

We now review the code listing of the NeXusBean for `NXroot`. Please note:

- many XML attributes are handled as Java Strings
  (some time attributes SHOULD be constrained).
- the indexed property `NXentry` is handled by a lazily instantiated array list hidden from users, who see the public indexed setter/getter methods, as well as an `addNXentry` method.
- there is no proper enforcement of the requirement that there be at least one `NXentry` yet, which could be achieved by lazy instantiation of at least one entry on access to the entry list.

```
package au.gov.ansto.bragg.nexus.auto;
import neutron.nexus.*;
import au.gov.ansto.bragg.nexus.base.*;
//  Generated from XML template(http://svn.nexusformat.org/definitions/trunk/base_classes/NXroot.xml)
/**
URL:     http://www.nexus.anl.gov/classes/xml/NXroot.xml
Editor:  NIAC
$Id$

Definition of the root NeXus group.

 */
 public class NXroot
 extends NXgroup_ {
/** Property string for bean property 'file_name' */
final static public String $file_name = "file_name";
// indexed = false
// required = false
private String file_name_;
public String getFile_name(){
 return file_name_;
}
public void setFile_name(final String file_name){
 file_name_ = file_name;
}
/** Property string for bean property 'file_time' */
final static public String $file_time = "file_time";
// indexed = false
// required = false
private String file_time_;
public String getFile_time(){
 return file_time_;
}
public void setFile_time(final String file_time){
 file_time_ = file_time;
}
/** Property string for bean property 'file_update_time' */
final static public String $file_update_time = "file_update_time";
// indexed = false
// required = false
private String file_update_time_;
public String getFile_update_time(){
 return file_update_time_;
}
public void setFile_update_time(final String file_update_time){
 file_update_time_ = file_update_time;
}
/** Property string for bean property 'NeXus_version' */
final static public String $NeXus_version = "NeXus_version";
```

```
// indexed = false
// required = false
private String NeXus_version_;
public String getNeXus_version(){
 return NeXus_version_;
}
public void setNeXus_version(final String NeXus_version){
 NeXus_version_ = NeXus_version;
}
/** Property string for bean property 'HDF_version' */
final static public String $HDF_version = "HDF_version";
// indexed = false
// required = false
private String HDF_version_;
public String getHDF_version(){
 return HDF_version_;
}
public void setHDF_version(final String HDF_version){
 HDF_version_ = HDF_version;
}
/** Property string for bean property 'HDF5_version' */
final static public String $HDF5_version = "HDF5_version";
// indexed = false
// required = false
private String HDF5_version_;
public String getHDF5_version(){
 return HDF5_version_;
}
public void setHDF5_version(final String HDF5_version){
 HDF5_version_ = HDF5_version;
}
/** Property string for bean property 'creator' */
final static public String $creator = "creator";
// indexed = false
// required = false
private String creator_;
public String getCreator(){
 return creator_;
}
public void setCreator(final String creator){
 creator_ = creator;
}
/** Property string for bean property 'nXentry' */
final static public String $nXentry = "nXentry";
// indexed = true
// required = true
// Extracted NeXus multiplicity indicator (+) means (May occur one or more times (i.e. at least once))
/** + */
private java.util.List<NXentry> nXentry_;
private java.util.List<NXentry> nXentry(){
 if (nXentry_ == null) nXentry_ = new java.util.ArrayList<NXentry>();
 return nXentry_;
}
// Extracted NeXus multiplicity indicator (+) means (May occur one or more times (i.e. at least once))
/** + */
public NXentry[] getNXentry(){
 return nXentry().toArray(new NXentry[0]);
}
public NXentry getNXentry(int index){
```

```
 return nXentry().get(index);
}
public void addNXentry(final NXentry nXentry){
 nXentry().add(nXentry);
}
/** constructor */
public NXroot(){
 super();
}
}
```

It is not intended that users interact with NeXusBeans via such lengthy, automatically generated code. Already in a modern IDE a NeXusBean can be conveniently navigated as a Javabean, and the Javadoc can be easily viewed and navigated in a web browser. Figure 2 shows the NXdata NeXusBean in the Netbeans IDE's browser:

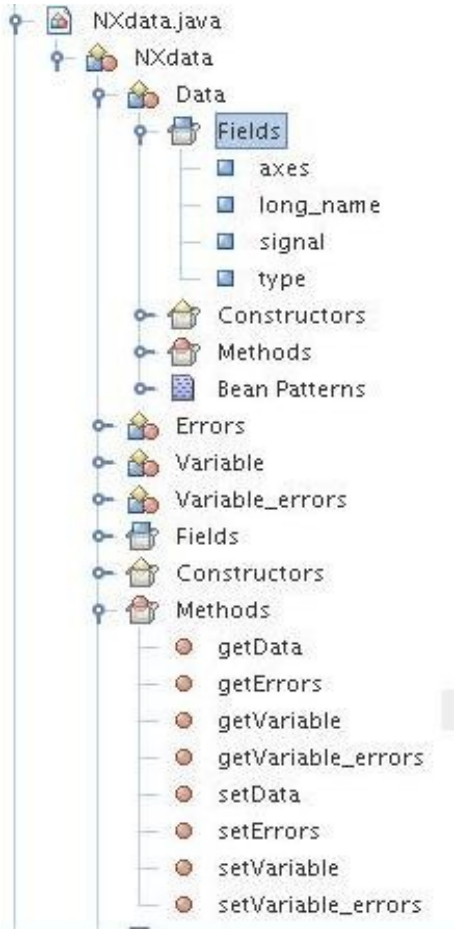


Figure 2: Snapshot of the NXdata NeXusBean in the Netbeans IDE

Code listings for the other examples in the overview will not be given. We wish instead now to focus on how one uses NeXusBeans in Java, and how one can analyse them graphically in UML.

3.3) NeXusBeans as Java objects in a Java NeXus model

Once the Java NeXusBean classes have been compiled, instances of NeXusBeans can be be created and added to each other to build entire NeXus models, thus:

```
                    !!! WARNING: PLACEHOLDER ONLY: COMPARISON WITH RITA-2 DIFFICULT !!!
                !!! THIS IS BEING REDONE WITH NEW PLATYPUS EXAMPLE AND NEW MULTIARRAY !!!
NXentry entry = new NXentry();
NXentry.Title title = new NXentry.Title("title of an experiment");
entry.setTitle(title);
entry.setStart_time(new NXentry.Start_time("2005-10-11 13:41:44"));
NXentry.Definition definition = new NXentry.Definition("NXmonotas");
definition.setURL("http://www.nexus.anl.gov/instruments/xml/NXmonotas.xml");
definition.setVersion("1.0");
entry.setDefinition(definition);
NXuser user  = new NXuser();
user.setName("Neutron Man");
user.setExtraName(new Nxuser.ExtraName("Neutron Man"));
user.setAffiliation(new NXuser.Affiliation("the neutron beam institute"));
user.setAddress(new Nxuser.Address("123 bragg lane, scattering land"));
user.setEmail(new NXuser.Email("unknown@nowhere"));
entry.addNXuser(user);
NXinstrument instrument = new NXinstrument();
instrument.setName("RITA-2");
NXcrystal crystal_m = new NXcrystal();
crystal_m.setName("monochromator");
instrument.addNXcrystal(crystal_m);
NXcrystal crystal_a = new NXcrystal();
crystal_a.setName("analyzer");
NXcrystal.Polar_angle polar_angle = new NXcrystal.Polar_angle(new float[]{16.9000f,17.0000f,17.1000f});
polar_angle.setUnits("degree");
crystal_a.setPolar_angle(polar_angle);
instrument.addNXcrystal(crystal_a);
NXdetector detector = new NXdetector();
detector.setName("detector");
NXdetector.Polar_angle polar_angle_d = new NXdetector.Polar_angle(new float[]{13.1470f, 13.1470f, 13.1470f});
polar_angle_d.setUnits("degree");
detector.setPolar_angle(polar_angle_d);
NXdetector.Azimuthal_angle azimuthal_angle = new NXdetector.Azimuthal_angle(180.0000);
azimuthal_angle.setUnits("degree");
detector.setAzimuthal_angle(azimuthal_angle);
instrument.addNXdetector(detector);
entry.setNXinstrument(instrument);
NXmonitor monitor = new NXmonitor();
monitor.setName("control");
entry.addNXmonitor(monitor);
NXmonitor.Preset preset = new NXmonitor.Preset(1.0000f);
preset.setNXtype(NXtypes.NX_FLOAT);
monitor.setPreset(preset);
NXmonitor.Mode mode = new NXmonitor.Mode("timer");
monitor.setMode(mode);
NXsample sample = new NXsample();
sample.setName("sample");
sample.setOrientation_matrix(new NXsample.Orientation_matrix(
new float[][] {{-0.0051f, -0.1426f, 0.0679f},{0.0002f, -0.0323f, -0.3005f},{0.1678f, -0.0043f,  0.0024f}}));
entry.setNXsample(sample);
NXroot root = new NXroot();
root.addNXentry(entry);
```

3.4) A NeXusBean class as reverse-engineered UML

Java can be reverse-engineered into graphical UML. In the following examples the powerful **Magicdraw UML** tool has been used, although most good UML tools will handle Java OK. However, the specified multiplicities of properties cannot be represented in Java without annotations, so multiplicities have been introduced manually in the UML models in this report.

In Figure 3 we see `NXroot` as a UML class:

- CAVEAT: the time attributes should be constrained to ISO 8601.

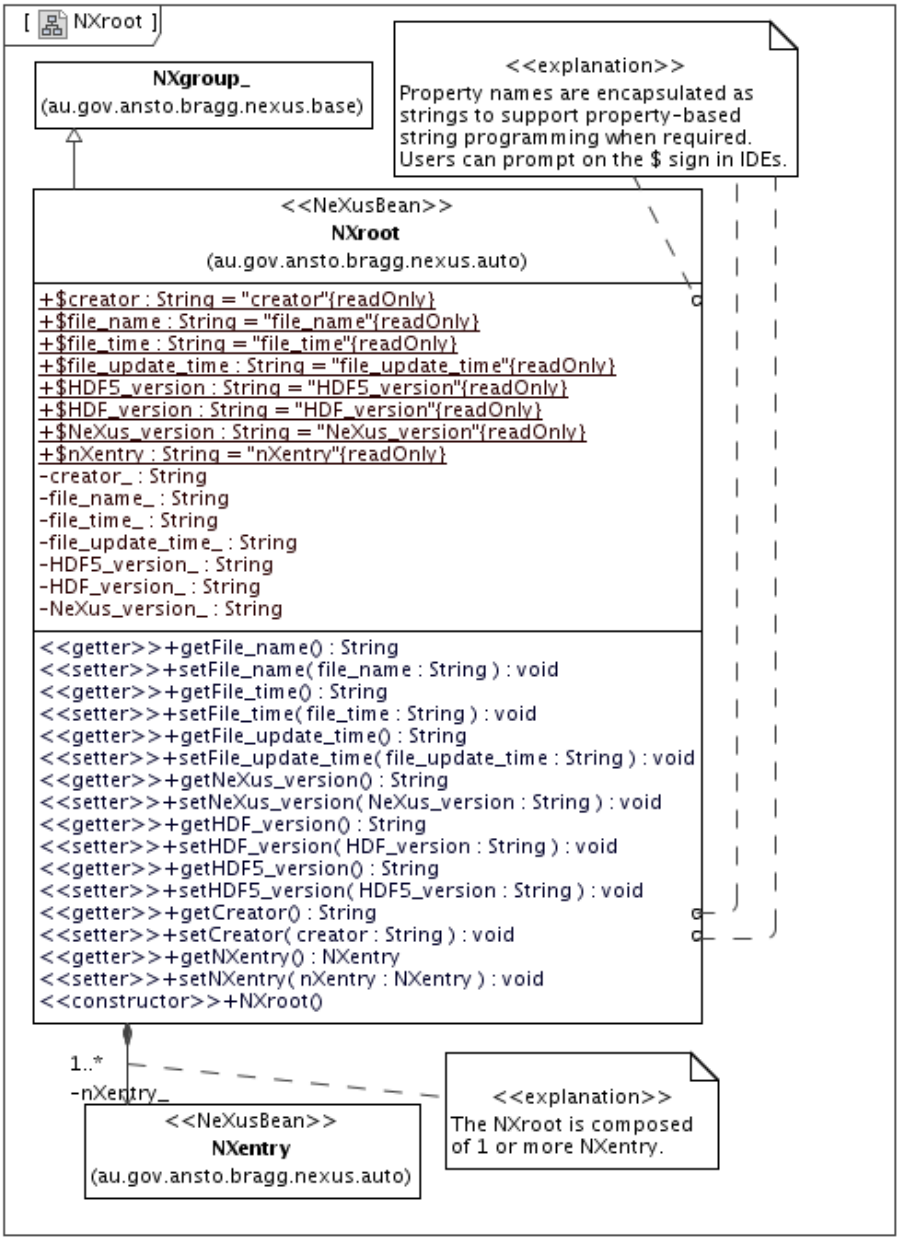


Figure 3: The `NXroot` NeXusBean reverse engineered to graphical UML from Java in the Magicdraw UML tool.

In Figure 4 we see `NXentry` as a UML class, with its many data items and composed groups.

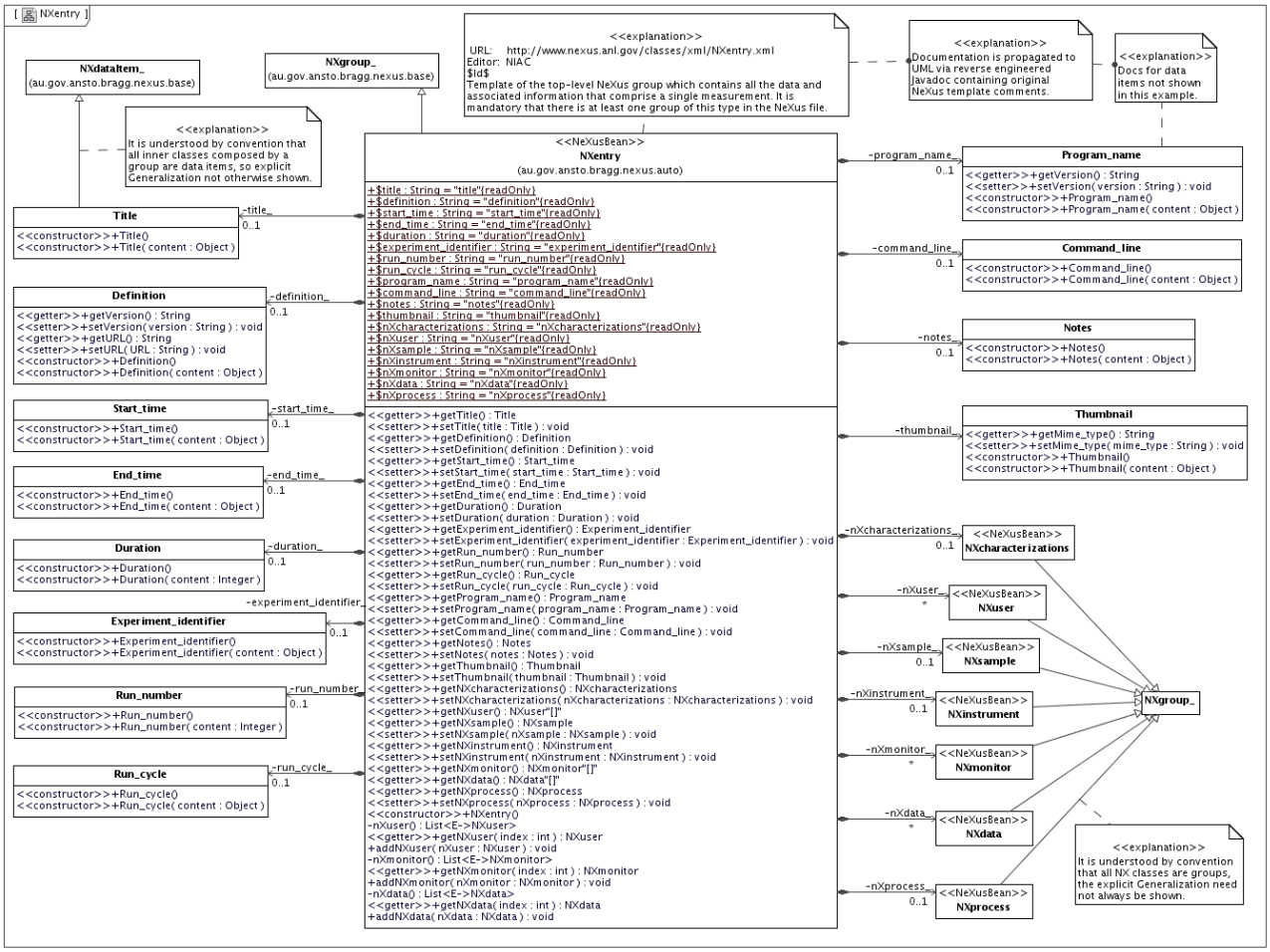


Figure 4: The `NXentry` NeXusBean in UML with composed data items and groups.
WARNING: image obsolete: does not show new scalar, vector, multi data items

Note the following:

- The original NeXus template documentation is propagated to UML via the Javadoc in the NeXusBean code, and can be retrieved into UML Comments in the diagram.
- Some data items have additional bean properties (beyond those inherited via NXdataItem) corresponding to additional XML attributes, with associated getters/setters.
- The relationship of NX classes to `NXgroup` and `NXdataItem` may be made explicit by showing the UML Generalization (or understood by convention for simpler diagrams). These relationships are of course always present in the UML model, whether shown or not.
- In the Magicdraw UML tool one can easily navigate between NeXusBeans hyperlinked to their respective focus diagrams, so that the entire NeXusBean system can be explored.
- CAVEAT: the `NX_CHAR[]` type for `Experiment_identifier` and `run_cycle` is not yet correctly mapped to a String. UPDATE to show as Vector<String> !
- The property name strings like `$title = 'title'` are merely a programmatic device to support developers during “string programming” against bean property names.

Unfortunately, the current NXdetector class template is too inconsistent for conversion by the NeXusBeans generator, so an older version (as presented at NIAC2006) is given below. Note the use of passive UML "wrapper components" to graphically and logically group the classes; these play no role in the software engineering, they are merely a graphical device, and do not affect packaging. (Please ignore the trailing '$' on data item field names, a convention no longer used.)

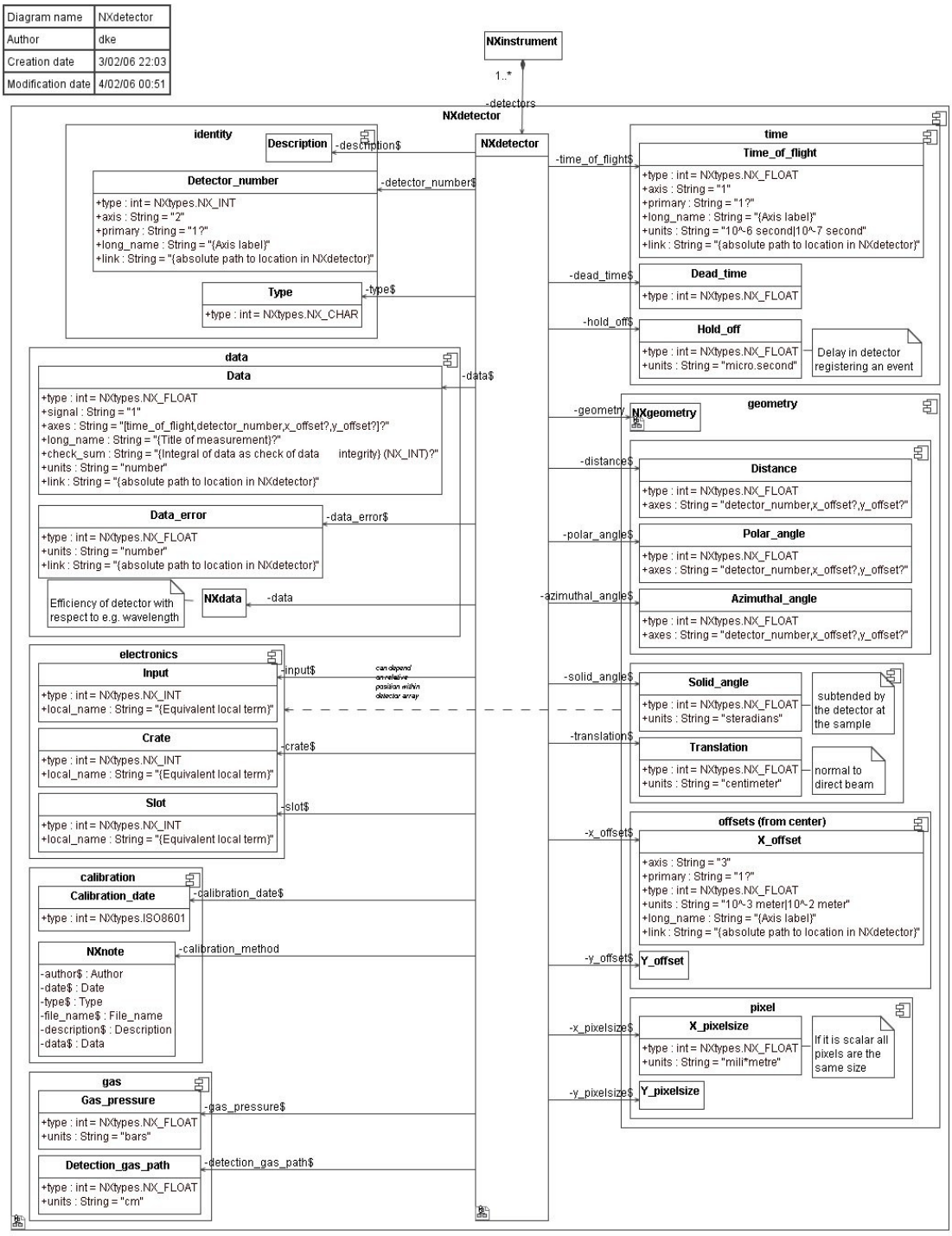


Figure 5: The NXdetector NeXusBean as UML class diagram with logical groupings

3.5) The entire NeXus system as Java NeXusBeans reverse-engineered to UML

In Figure 6 we see a UML class diagram of all reverse-engineered Java classes NeXusBean, providing a concise overview of the NeXus class system. Only the associations between classes are shown, and the relationships to the NeXus base classes are omitted. The classes are graphically and logically grouped according to the composition hierarchy using passive wrapper components.

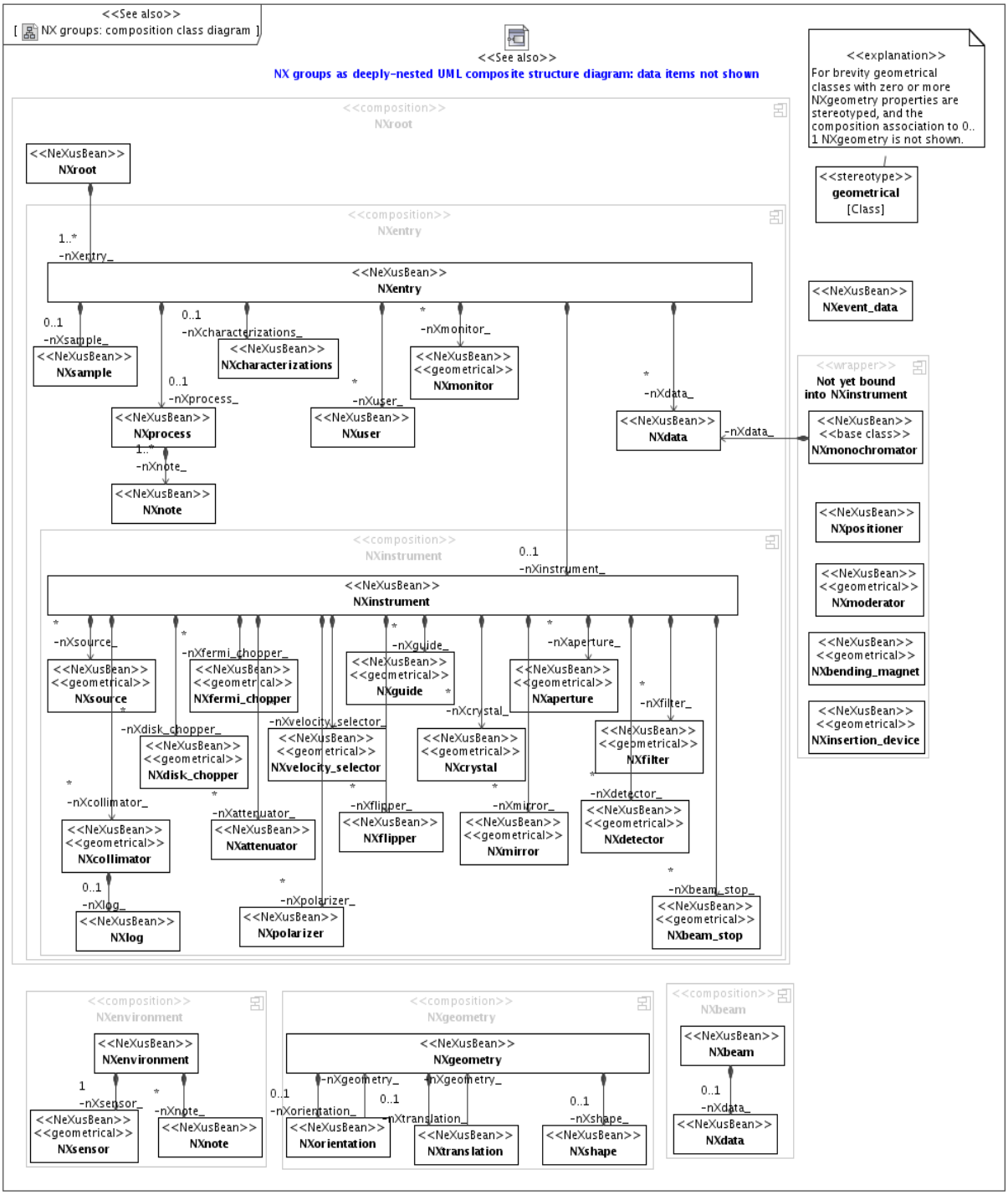


Figure 6: UML class diagram of all reverse-engineered Java NeXusBeans.

In Figure 7 we see another representation of the reverse-engineered Java NeXusBeans, a deeply-nested composite structure diagram. This representation corresponds exactly to the class diagram Figure 6, however instead of representing composition through associations, part properties within the composite structure are used. In the Magicdraw UML tool there is precise validation of permitted parts against the underlying model. Note the representation of UML multiplicities on the parts properties in [] brackets. Each part property has an attribute name and a NeXusBean type.

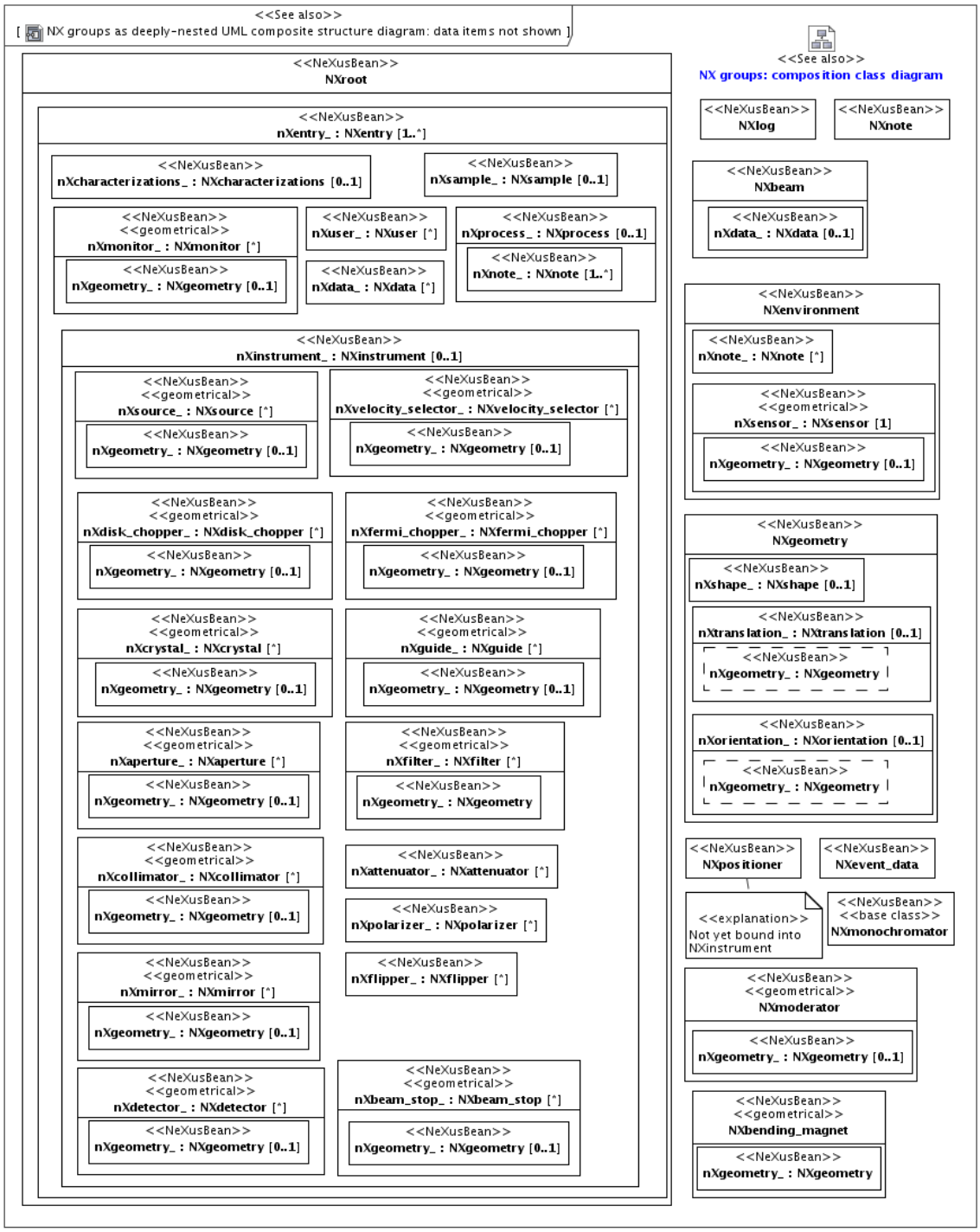


Figure 7: UML composite structure diagram of all reverse-engineered Java NeXusBean classes, employing deeply-nested parts in the Magicdraw UML tool.

3.6) NeXusBeans UML class and composite structure models: Platypus example

Java NeXusBeans can be used to build class models of neutron beam instruments and to create UML diagrams equivalent to those models using reverse-engineered NeXusBeans and reverse-engineering instrument models. Such models may be different from the official NeXus instrument definitions; one can create any number of different NeXusBean models for a variety of purposes by logically organising the NeXusBeans in desired subsystems completely "parasitically". The same Java model class can of course be used to also create valid NXroot, NXentry, and NXinstrument structures from the same shared NeXusBeans, and that has been done in the examples below.

Figure 8 shows a simplified class model of the Platypus reflectometer, which emphasises the hierarchical containment of components of the instrument using UML associations. Note that:

- The `<<Subsystem>>` classes merely group NeXusBeans, they do not have any influence over the `NXinstrument`, `NXentry`, `NXroot` objects or their serialization to a NeXus file. Formally the subsystems share NeXusBeans that are composed (directly or indirectly) by the `Nxroot,` as indicated by the "open diamond" shared aggregation kind notation in UML.
- The Platypus class is NOT and `NXinstrument`. It merely organised the NeXusBeans.
- The `NXinstrument` created by the `newInstrument()` method is a valid NeXus structure (the NeXusBeans of which are merely shared by the class model) and may be serialized.

Such class diagrams (software engineering view) tend to be difficult for non-experts to read. An equivalent composite structure diagram (systems engineering view) is shown in Figure 9. The support for deeply nested structures lends itself well to representation of the subsystems of a scientific instrument. Instead of showing associations, the properties of each class are shown as named "part Properties" with multiplicity indicators within the composite structure at each level. These diagrams are completely synchronised with each other; they are merely different views of the same UML model created by reverse-engineering the Java NeXusBeans classes and model class.

The graphical power of the UML-based NeXusBeans strategy is already of great value for analysis and illustration of structural aspects of the NeXus systems.

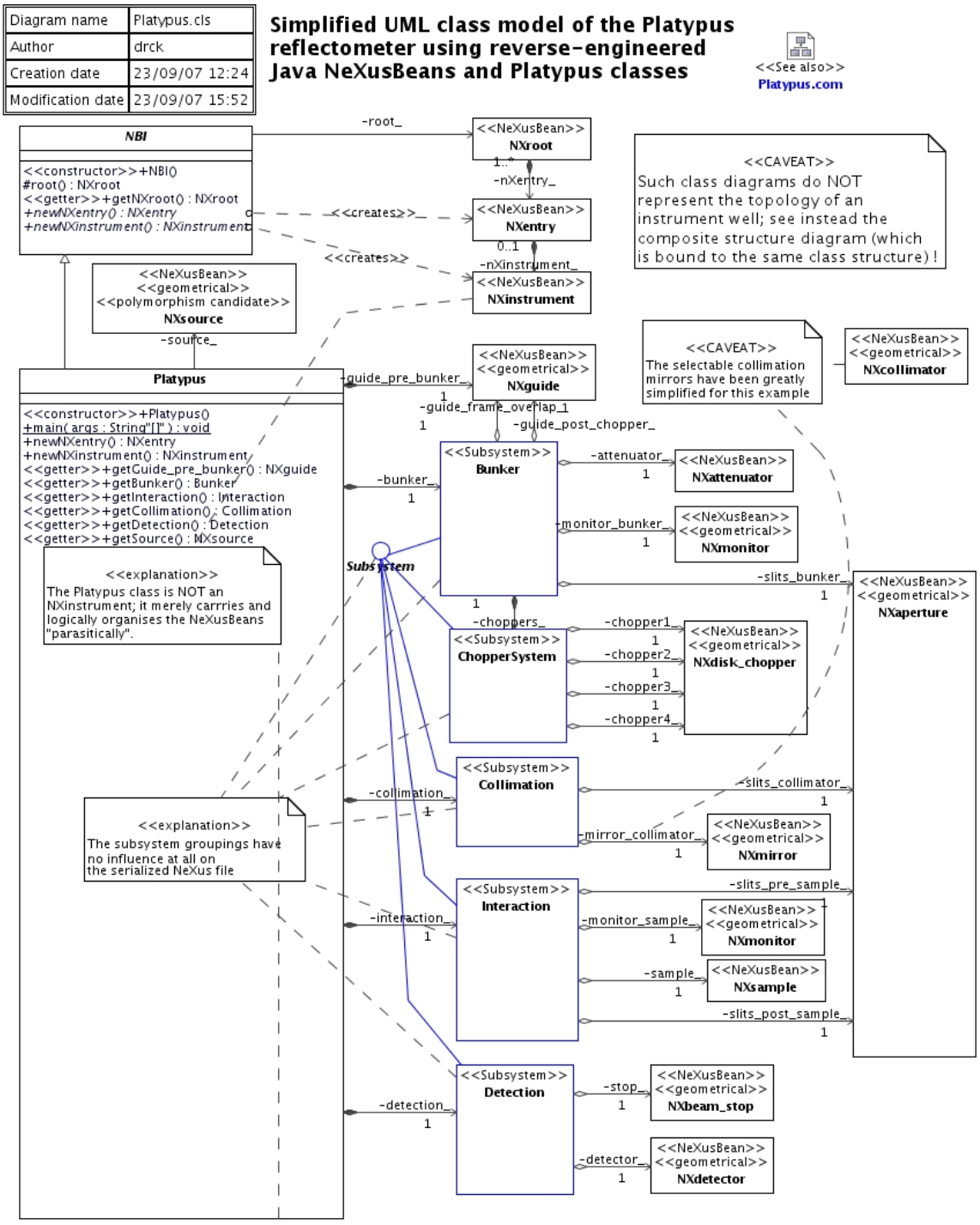


Figure 8: NeXusBeans class model of the Platypus reflectometer

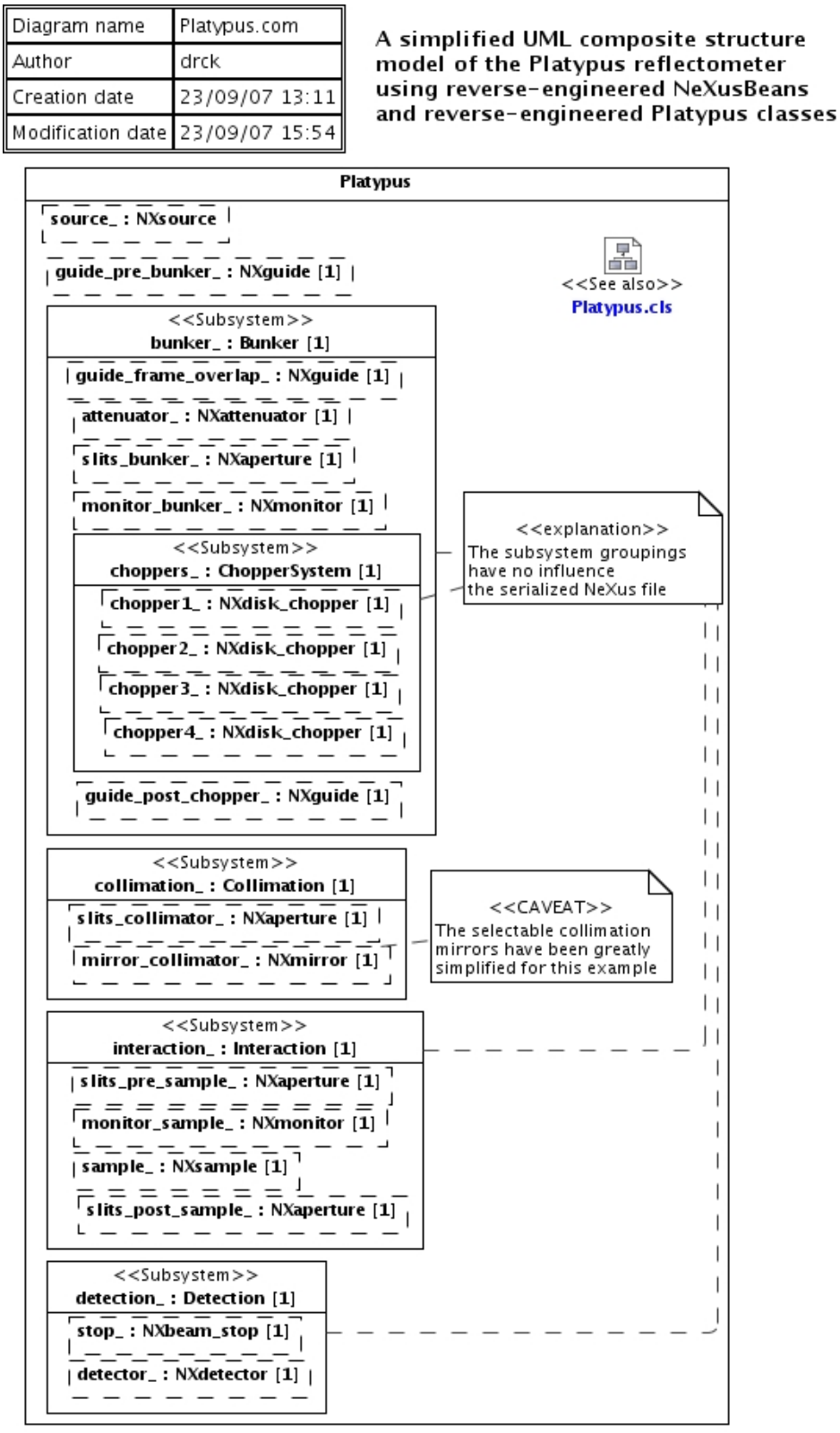


Figure 9: UML Composite Structure Diagram of the Platypus NeXusBeans instrument model

4) The Bragg NeXus base classes

Many of the NeXus design principles and shared (inherited) attributes can be encapsulated in reusable base classes. The Bragg base - already presented in prototype form at NIAC 2006 – serves as a starting point for object-orientation of the NeXus system. A brief overview only is provided here, many more details are available in the UML model and in the Javadoc, which will all be made available through the NeXML web site.

4.1) The universal NXobject, the NXdataItem classes, and the NXgroup classes

Please note these features with reference to Figure 10:

- The `NXobject` is a universal root interface for the NeXusBeans system. It has shared `name` and `id` attributes. The implementing class `NXobject_` offers internal services via `All_`.
- Every NeXusBean (directly or indirectly) implements the `NXgroup` interface.
- All NeXusBeans import `au.gov.ansto.bragg.nexus.base` and are compiled against it.
- Some NeXusBeans implement the `NXgeometrical` interface and have a 0..1 `NXgeometry` property; they are stereotyped as <<geometrical>> throughout the UML models here.
- The `NXdataItem<T>` interface is parametrised by an unbounded Java type using Java5 generics, which is represented in UML using intermediate template bindings (which unfortunately leads to some clutter in the UML diagrams, although it is easy to read in Java).
- Although the `NXdataItem` is parametrised by type, it must also by constructed with a given integer NeXus type indicator, since there is no obvious mapping between all NeXus types and available Java types. Mappings are made via the `NXtypes` utility class to and from the `neutron.nexus.NexusFile` integer type definitions (from JNexus version 2.0).
- Data items are further divided into scalar, vector, and multi-dimensional data items, with dedicated implementations of content handling.
- The `NXmultiDataItem_` implementation leverages the UCAR Java MultiArray API.
- There is currently no facility for handling type specifications like: `NX_FLOAT|NX_INT`, although Java generics do offer a possible strategy. Current policy is to take the first type.

In addition the Bragg NeXus base offers some exception handlers and type mappings (not shown).

4.2) Recurring data items

Some data items that recur amongst the NeXus templates without any real variation have been encapsulated as dedicated data items in the base,, according to the classic object-oriented reuse strategy. Such encapsulation promotes reuse, reduced maintenance, and reduces errors due to inconsistencies.

The NeXusBeans generator does not generate on-the-fly inner classes for these selected recurring data items, which are:

1. `Distance`: always interpreted as the distance from the sample and always in the same units: (`NXbeam, NXmonitor, NXattenuator, NXsource`)

2. `Description`: merely describes the instance of whatever class it is in: (`NXaperture, NXpolarizer, NXsample, NXenvironment, NXlog`)
3. `Name:` is redefined as `ExtraName` to avoid a clash with the `name` attribute in `NXobject:` (`NXenvironment, NXinstrument, NXuser, NXsample, NXsensor, NXsource`)

There may be other candidates for this strategy, too.

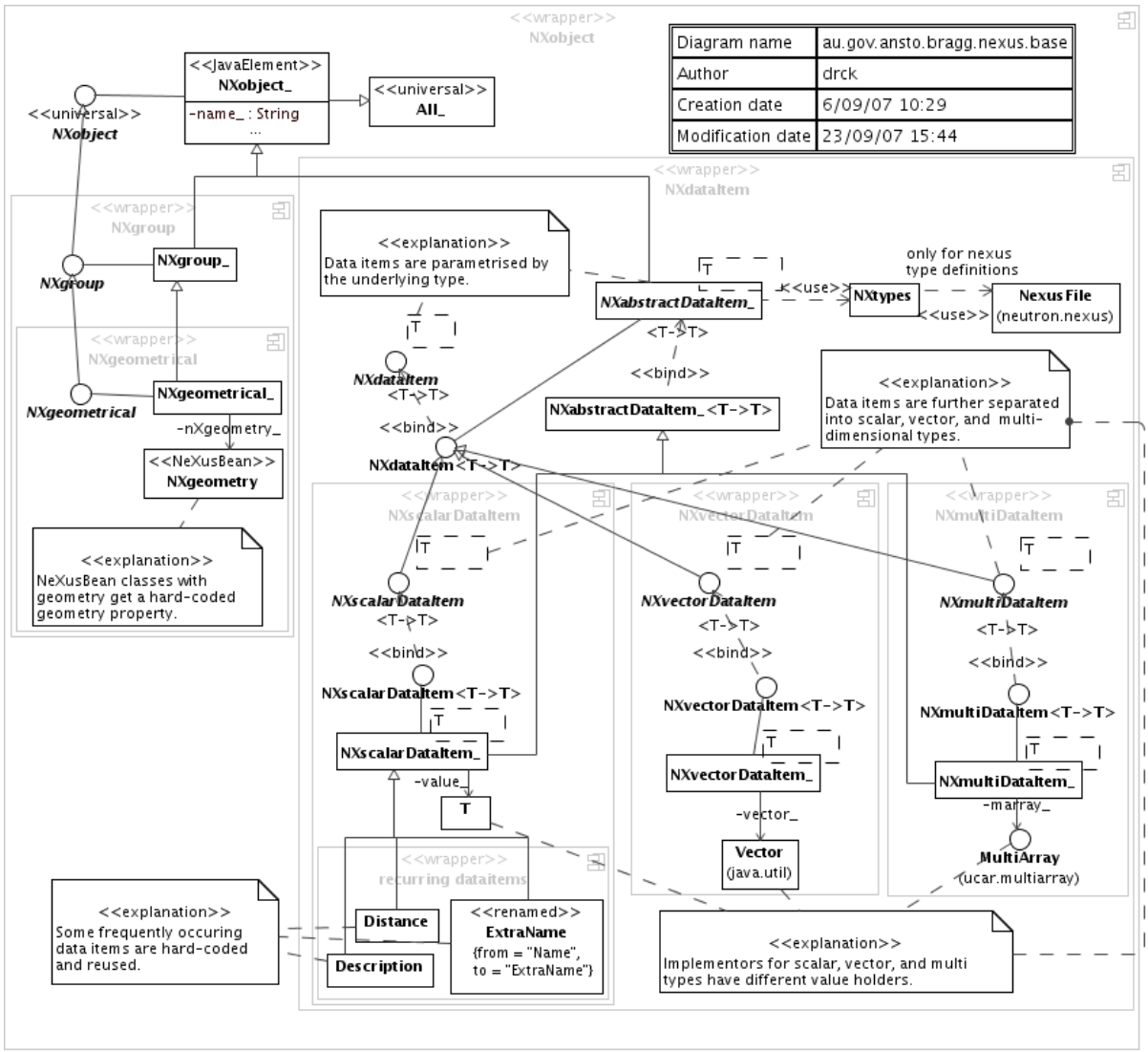


Figure 10: UML class diagram overview of the object-oriented Bragg NeXus Base employing Java5 generics for a family of typed data items.

4.3) NeXus type remapping

Standard NeXus types are defined as Java integers in the `neutron.nexus.NexusFile`. The Bragg `NXtypes` class (see Figure 11) offers a number of mappings to and from these integer types, and services for converting between String, Integer, and compilable Java code forms. It also identifies those group classes that are (probably incorrectly) used as types of data items in some NeXus class templates. There are many inconsistencies in the NeXus class templates and some also in `neutron.nexus.NexusFile`, so it is currently not at all clear how the NeXusBeans generator to map all NeXus types encountered in data items. This remapping strategy is thus quite fragile !

For example:

- type specification errors like `'NXFLOAT'` and `'FLOAT'` are remapped to `'NX_FLOAT'`
- type specification errors like `'NX_FLOAT[np?,i?,j?'` in `NXdetector` are remapped to `'NX_FLOAT[np?,i?,j?]'`
- neither `NX_Binary` nor `NX_Boolean` are supported in the JNexus 2.0 API `NexusFile`, so they are simply mapped to `NexusFile.INT8`.

The `NXtypes` class provided diagnostic warnings for many such errors, and therefore serves also as a kind of validator for the NeXus templates. When the entire set of NeXus template classes is pulled from SVN and processed with the aid of `NXtypes` many errors must be programatically handled. Clearly, the NeXus templates are NOT being digitally validated before submission to the SVN.

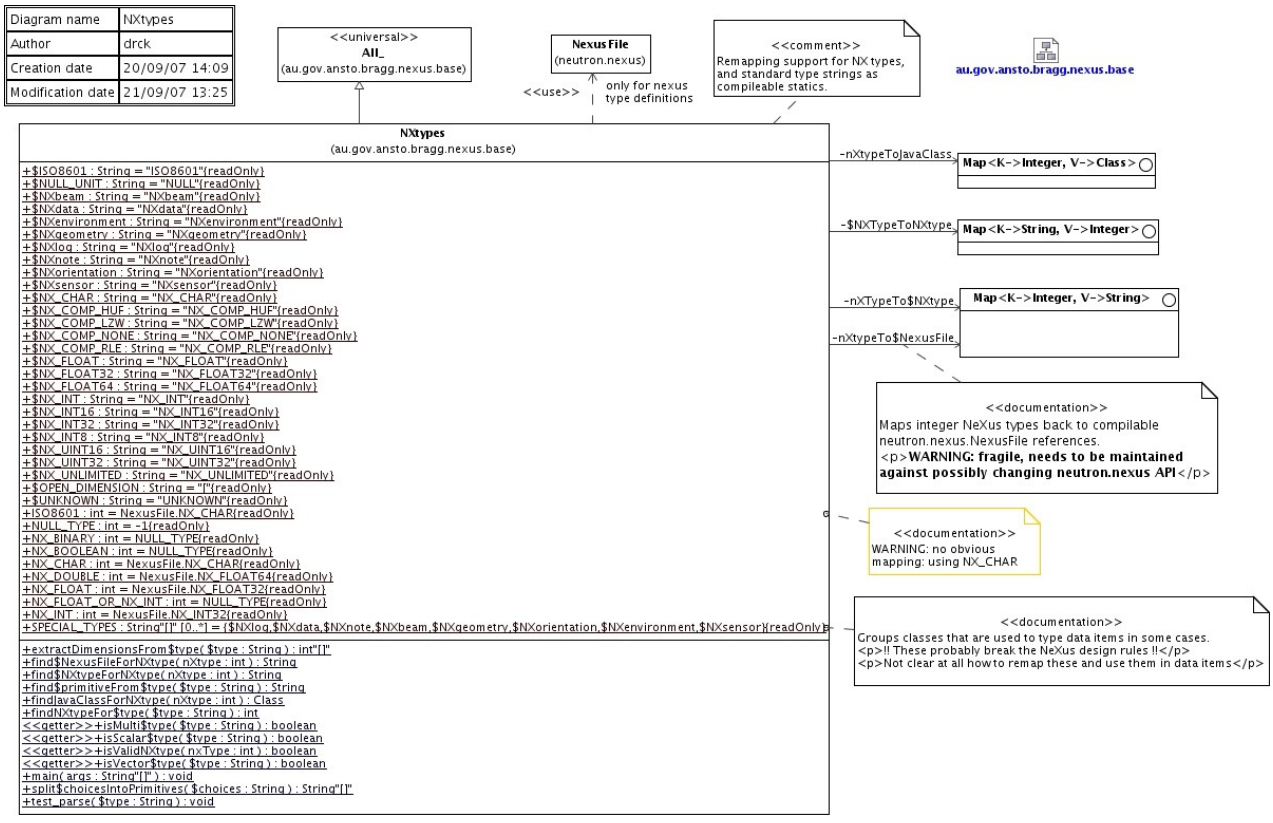


Figure 11: NXtypes: NeXusBean type mapping support and conversion services

4.4) Parametrising data items with Java5 generics (template bindings)

Once the NeXus 'type' attribute for a data item has been identified and mapped to a Java class, a "genericised" Java data item class may be generated. However, the NeXus integer type is still tracked in the `NXabstractDataItem_` constructor so that the NeXusBean can be serialized. The result is demonstrated as a UML class diagram in Figure 12. Although it makes the resulting Java code far more concise and consistent, the use of Java5 generics is not without its cons, especially concerning issues when mapping to XML Schema, and slightly more verbose UML diagrams.

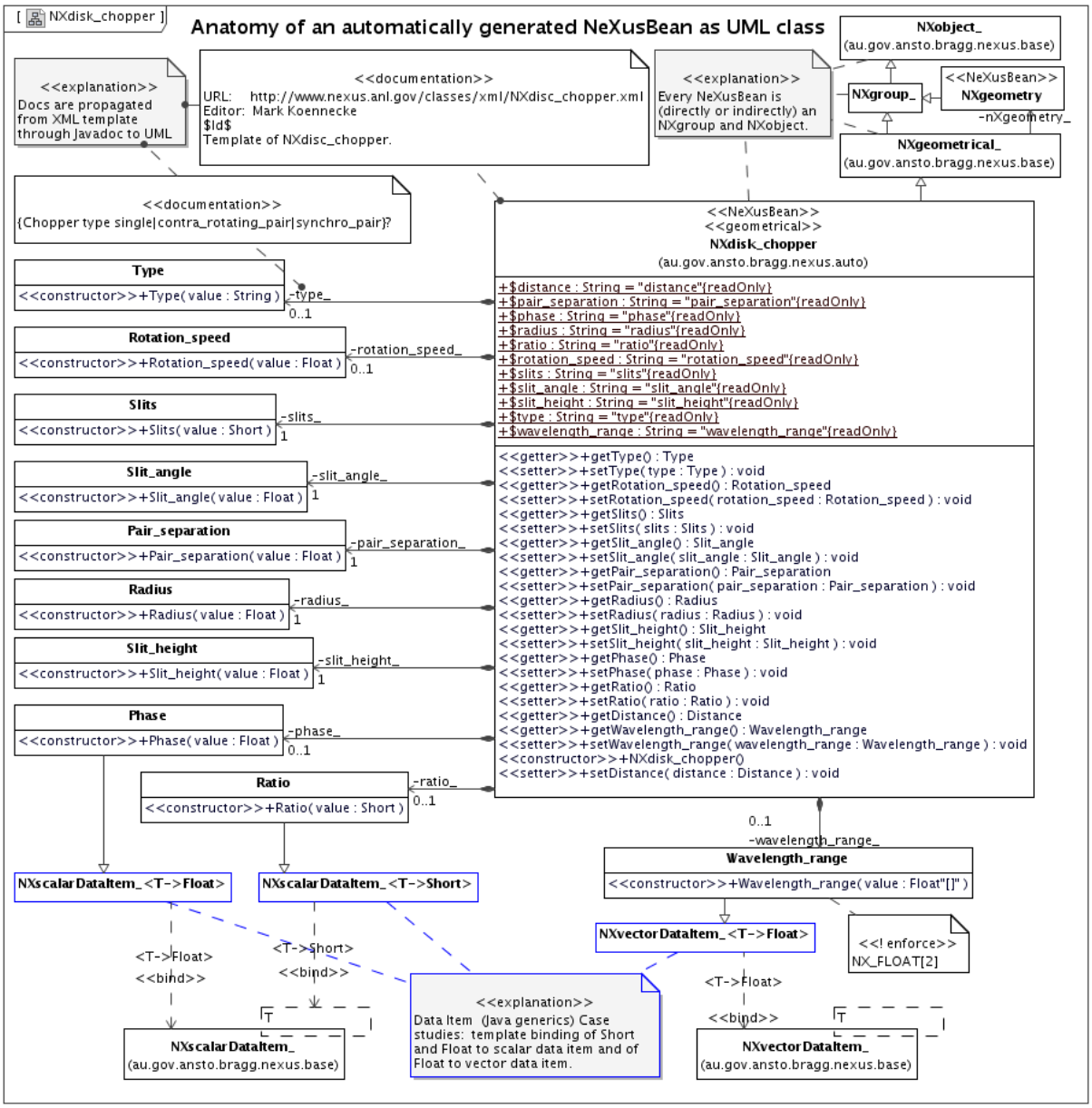


Figure 12: The NXdisk_chopper NeXusBean reverse-engineered to UML with template bindings shown for selected scalar and vector data items.

5) Serialization of Java NeXusBeans models to NeXus XML files

TODO: URGENT WORK IN PROGRESS ! Leverage new scalar/vector/multi data items

5.1) The XStream-based NeXusBeans marshalling/unmarshalling system

TODO: URGENT WORK IN PROGRESS !

5.2) Serialization example: the Bragg Institute's Platypus reflectometer at OPAL

TODO: URGENT WORK IN PROGRESS !

6) Metamodel-driven forward engineering for Java NeXusBeans

TODO: text overview of the Bragg Forward metamodel for generating NeXusBeans. Image only.

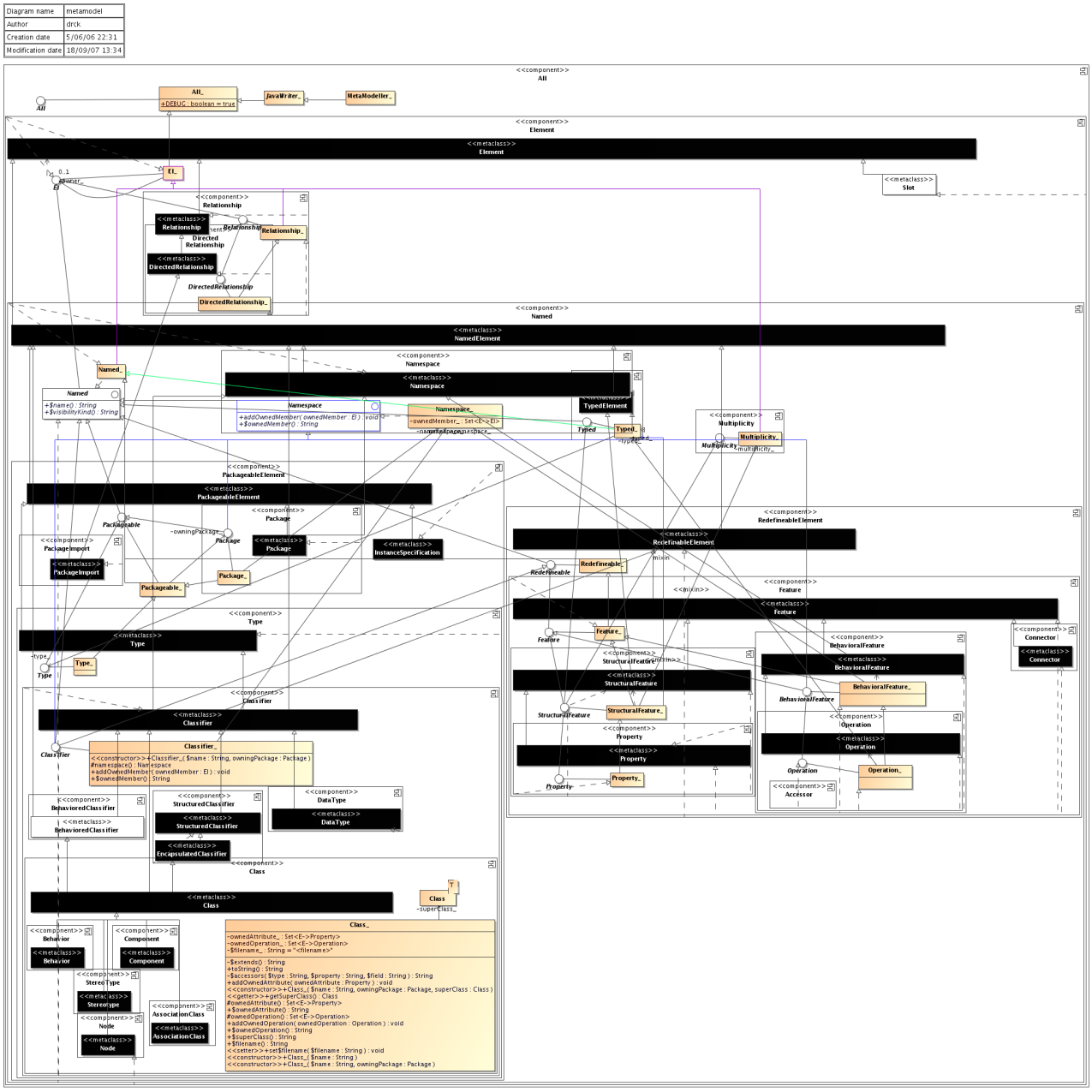


Figure 13: UML class diagram of the UML-like metamodel for the Bragg Forward Engineering system for Java NeXusBeans. The metaclasses can write themselves as Java code string elements.

7) Interpreting the NeXus templates

7.1) Interpreting the NeXus data item type specifications

The type and dimensionality of a NeXus data item is specified using the `type` attribute. Unfortunately, the notation currently employed is extremely overloaded and cryptic, presenting a significant challenge to the NeXusBeans generator. At least the following cases have been identified:

1. A character sequence:
   `<NXenvironment> .. <program type="NX_CHAR">`
2. A vector of character sequences, of parametrised length:
   `<NXsample> .. <component type="NX_CHAR[n_comp]">`
   Note that `n_comp` is never even defined in the class !
3. A numeric scalar of a given type:
   `<NXbeam>..<distance type="NX_FLOAT" units="m">`
   `<NXgeometry name="">.. <component_index type="NX_INT">`
4. A numeric scalar with a choice of type:
   `<NXlog>.. <raw_value units="{..}" type="NX_FLOAT|NX_INT">`
5. A numeric vector with a given type and given length:
   `<NXsample> .. <sample_orientation type="NX_FLOAT[3]" units="degree">`
   `<Nxcrystal> .. <reflection type="NX_INT[3]">`
6. A numeric vector with a given type and parametrised length:
   `<NXsample> .. <unit_cell_volume type="NX_FLOAT[n_comp]"..>`
7. A numeric vector with a given type and parametrised length with inline arithmetic:
   `<NXdetector>.. <time_of_flight type="NX_FLOAT[j+1]"..>`
8. A numeric vector with given type and unknown length:
   `<NXbeam> .. <incident_energy type="NX_FLOAT[:]" units="meV">`
9. A numeric vector with given NXtype and unknown length and index placeholder:
   `<NXbeam> .. <flux type="NX_FLOAT[i]" units="s-1cm-2">`
10. A numeric vector with a choice of type and unknown length:
    `<NXdata> .. <variable type="NX_FLOAT[:]|NX_INT[:]" ..>`
11. A numeric matrix with given type and both dimensions known:
    `<NXcrystal> .. <orientation_matrix type="NX_FLOAT[3,3]">`
12. A numeric matrix with given type, first dimension known, and second unknown:
    `<NXbeam> .. <final_polarization type="NX_FLOAT[3,:]">`
13. A numeric multi-array with a given type, unknown dimensionality, and unknown lengths:
    `<NXdata> .. <errors type="NX_FLOAT[:...]">`
    Note, the notation ':...' is NOT defined in the METAFORMAT !
14. A numeric multi-array with a choice of type, unknown dimensionality, unknown lengths:
    `<NXdata> .. <data type="NX_FLOAT[:...]|NX_INT[:...]"`

15. A numeric multi-array with given type, at least one dimension, and an optional second dimension, and index placeholders:
    `<NXevent_data> .. <pulse_height type="FLOAT`[10]`[i,k?]" units="">`
16. A numeric multi-array with given type, one dimension parametrised, and the other dimensions given:
    `<NXsample> .. <unit_cell type="NX_FLOAT[n_comp,6])">`
    `<NXsample> .. <orientation_matrix type="NX_FLOAT[n_comp,3,3]">`
    Note that `n_comp` is never even defined in the class !
17. A numeric multi-array with choice of type, variable number of dimensions, and index placeholders:
    `<NXdata> .. <data type="NX_FLOAT[i,j,k?,l?]|NX_INT[i,j,k?,l?]"`
    (the above example is from an older `NXdata` version from 2006).
18. A data item with `NXlog`, `NXnote`, `NXdata`, `NXsensor`, or `NXgeometry` given as type:
    `<NXmonitor>..<integral_log type="NXlog" units="">`
    `<NXentry>..<thumbnail type="NXnote" mime_type="{image/*}">`
    `<NXbending_magnet>..<spectrum type="NXdata">`
    `<NXenvironment>..<position type="NXgeometry">`
    `<NXfilter>..<sensor_type type="NXsensor">`

There are currently many errors in the specifications of types. For example:

`<NXcrystal>..<cut_angle type="NXFLOAT"..>`
`<NXevent_data>..<pulse_height type="FLOAT[i,k?]" units="">`

10 Note also error in type specification, `FLOAT` should be `NX_FLOAT`

8) The need for inheritance and polymorphism in NeXus

At NIAC2006 D. Kelly made a strong case for the need for support for polymorphism in NeXus, and likewise a case for the need for industry-standard inheritance notation, and presented examples such as a family of detector classes, using graphical UML class diagrams. That investigation is deepened here.

Inheritance is the ability of *subclasses* (inheritance children) to reuse features defined in *base classes.* For example:

- Every NeXus group has a name attribute; there is no sense in redefining the name and its documentation each time, rather the name can be inherited from an `NXgroup` or even an `NXobject` base class.
- There are a number of NeXus groups with zero or one `NXgeometry` properties. These <<geometrical>> classes can inherit from a common `NXgeometrical` base class.

Polymorphism means “many formed”. There are a number of ways of handling polymorphism in object-oriented languages, including generalization/specialization through inheritance and/or extension, dynamic aggregation, or hybrids of these. In general, one wishes to identify independent ontological axes in the domain and isolate these in the class model, to reduce coupling. For example, a class for a neutron beam instrument that does not exploit time of flight (TOF) should not have to refer to TOF. Similarly, a class for a neutron source that is not pulsed should not have to refer to pulse characteristics, and a class for a neutron source should not have to refer to X-Ray properties.

We now investigate some candidates for polymorphism amongst the current NeXus classes. In what follows, <<analysis>> classes in UML models are distinguished from reverse-engineered NeXusBean Java classes by a preceding '*'.

8.1) Polymorphism candidate: NXchopper

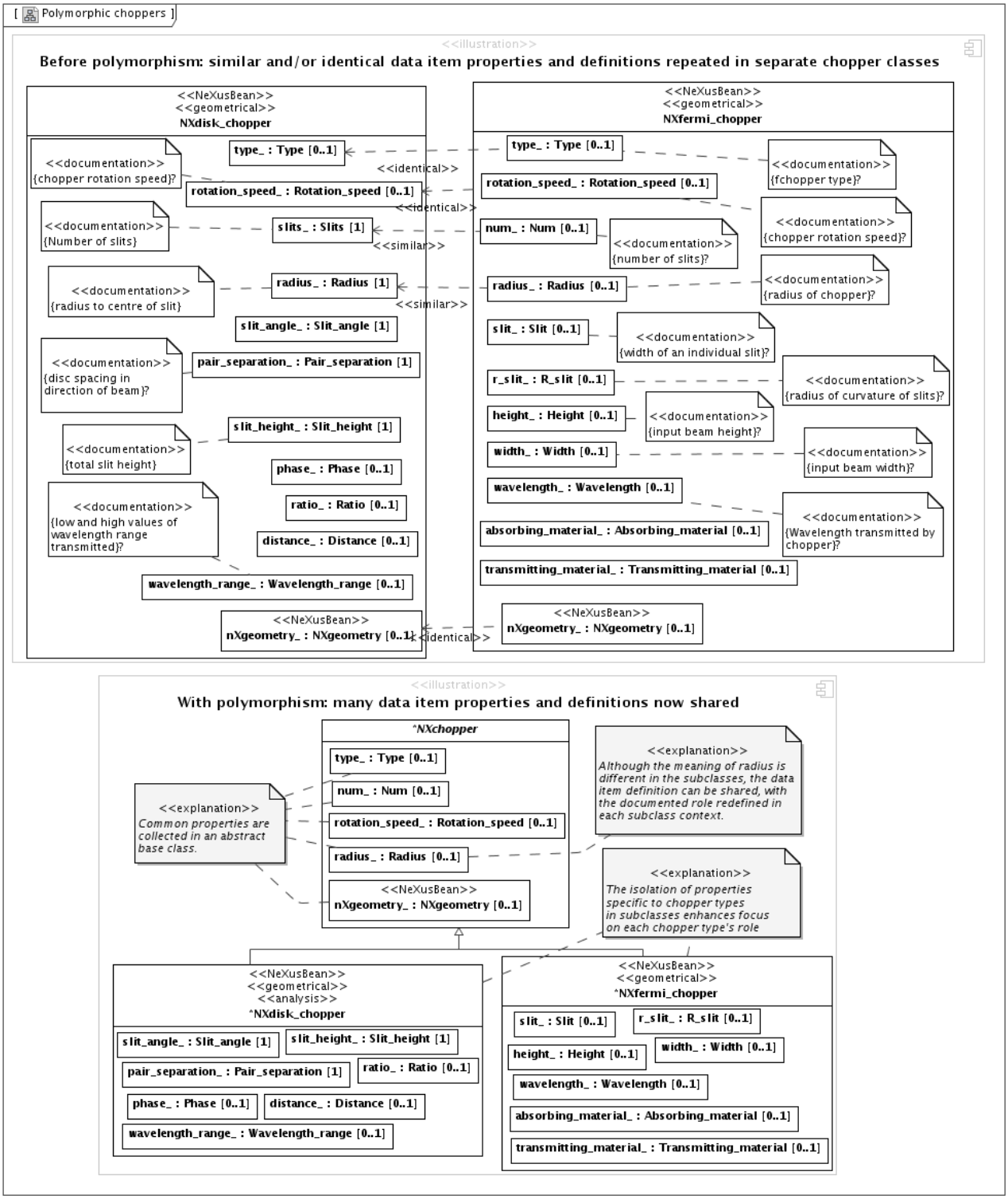
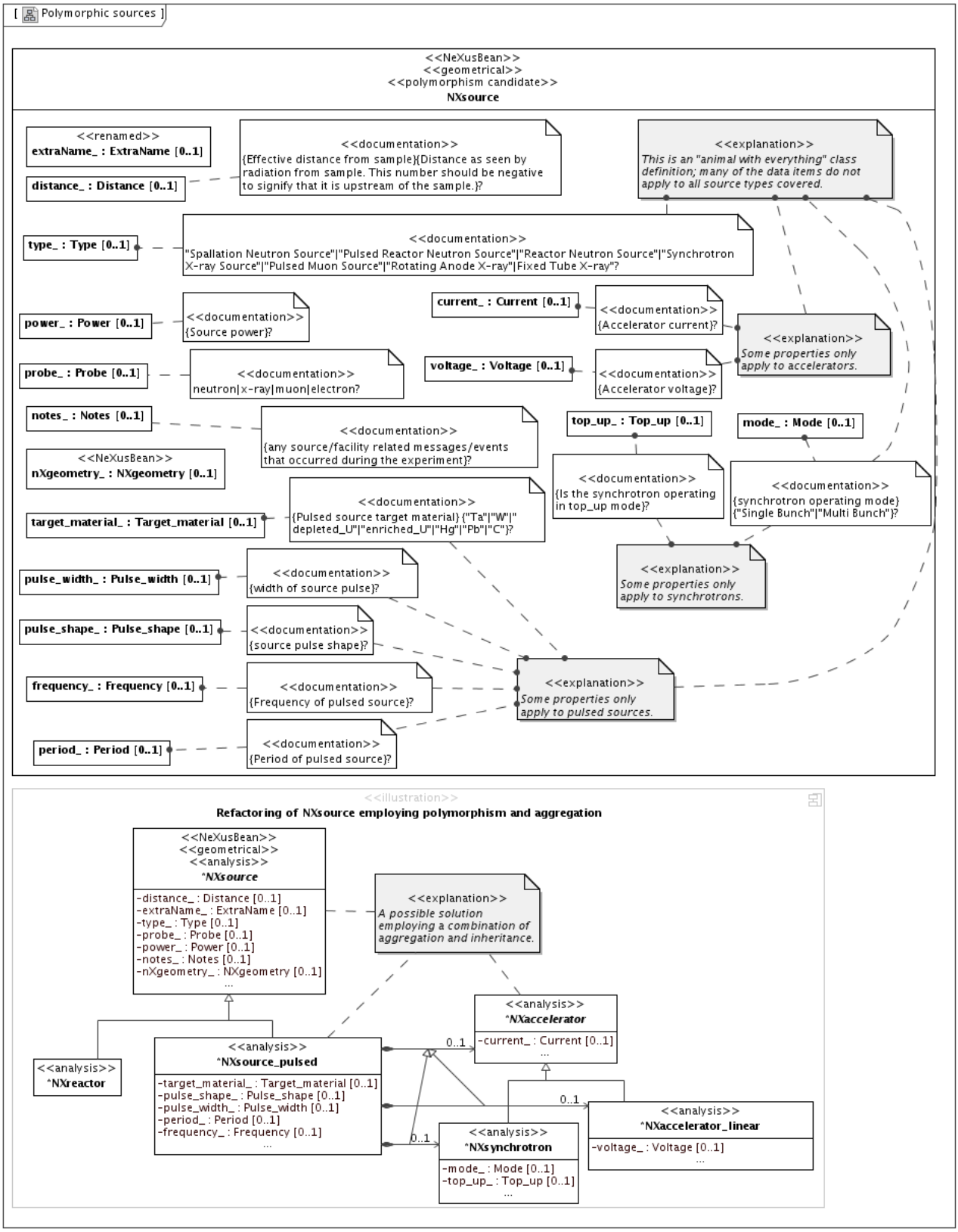


Figure 14: A UML Composite structure diagram with part properties shows how data item definitions and properties can be shared by different choppers by inheritance from an abstract base

8.2) Polymorphism candidate: NXsource



8.3) Polymorphism candidate: NXdetector

TODO: deliberately postponed awaiting developments at NIAC2007.
Also only now have workaround for inconsistent NXdetector.xml on SVN.
See notes from detector discussions at NIAC2006.

8.4) Polymorphism candidate: NXshape

Already the specification's text invites polymorphism. Clearly what is needed is a public interface with common shape aspects and extensible subclasses like `NXcylinder`, `NXbox`, `NXsphere`.

```
<!--
URL:     http://www.nexus.anl.gov/classes/xml/NXshape.xml
Editor:  NIAC
$Id: NXshape.xml 4 2005-07-19 04:10:26Z rio $
This is the description of the general shape and size of a
component, which may be made up of "numobj" separate elements -
it is used by the NXgeometry.xml class
-->
<NXshape name="{name of shape}">
        <shape type="NX_CHAR">
                {"nxcylinder", "nxbox", "nxsphere", ...}?
        </shape>
        <size type="NX_FLOAT[numobj,nshapepar]" units="meter">
                {physical extent of the object along its local axes (after
NXorientation) with the center of mass at the local origin (after Nxtranslate).}
{The meaning and location of these axes will vary according to the value of the "shape"
variable. nshapepar defines how many parameters.
For the "nxcylinder" type the paramters are (diameter,height).
For the "nxbox" type the parameters are (length,width,height).
For the "nxsphere" type the parameters are (diameter).}?
        </size>
</NXshape>
```

8.5) Polymorphism candidate: NXmonochromator

The NeXus class `NXmonochromator.xml` is documented as a "base class"[11]:

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
URL:     http://www.neutron.anl.gov/nexus/xml/NXmonochromator.xml
Editor:  NIAC
$Id$
This is a base class for everything which selects a wavelength or energy, be it a
monochromator crystal, a velocity selector, a undulator or whatever
-->
<NXmonochromator name="name of monochromator">
  <!--
    expected units
    wavelength: angstrom
    energy:     eV
  -->
  <wavelength type="NX_FLOAT[]" units="{unit}">
     {wavelength selected}
  </wavelength>
  <wavelength_error type="NX_FLOAT[]" units="{unit}">
     {wavelength standard deviation}
  </wavelength_error>
  <energy type="NX_FLOAT[]" units="{unit}">
     {energy selected}
  </energy >
  <energy_error type="NX_FLOAT[]" units="{unit}">
     {energy standard deviation}
  </energy_error>
  <NXdata name="distribution"/>
</NXmonochromator>
```

However, candidate subclasses like `NXcrystal` do not yet take advantage of this base class, and there is no agreed notation to indicate the inheritance. Also, similarly named data items do not yet agree:

```
<NXcrystal>..<wavelength type="NX_FLOAT[i]" units="Angstroms"> {Optimum diffracted
wavelength}? </wavelength>
```

11 The introduction of an NXmonochromator base corresponds with a suggestion made by Darren Kelly at NIAC2006.

9) NeXusBean classes as XML Schema elements

Once the NeXus system is encapsulated as Java it can be transformed to an XML Schema in a tool like Magicdraw UML. However, although advanced tools like Magicdraw UML can capture all of the necessary features of the XML Schema language, Java (unless assisted by an advanced annotation system like EMF) cannot easily capture all of the information required for a robust XML Schema, so the resulting schema is at best a structurally valid data modelling starting point. In particular, the multiplicities of properties are not encapsulated in Java, so some manual vetting of the intermediate UML model is required, as has been performed in the UML models in this report.

9.1) Transformation of Java NeXusBeans via UML to XML Schema

TODO: XML Schema element examples: have now changed due to scalar/vector/multi base ALSO, use of Java5 generics poses issues for Magicdraw UML transformation engine.

9.2) The Eclipse Modelling Framework (EMF) as bridge to an XML Schema

TODO: Lower priority, may become separate report. EMF can't handle multi-dim data easily.

10) NeXus instance modelling

So far we have examined mostly class-level models of the NeXus system. We now examine some techniques for building XML instance files and graphical instance models consistent with (validated against) the NeXus class model.

10.1) NeXus instance modelling Java

We have already seen above how one can build a NeXus instance model directly in Java code (and how that can be serialized as XML). An example of the Platypus reflectometer in Java was given. There are the following aspects of validation:

- At compile-time, the NeXusBeans bean setters/getters specify exactly which attribute properties and which group properties (children) are permitted, and this feature may be exploiting in modern Java IDEs (like Netbeans or Eclipse) that prompt on those setters/getters.
- At run-time, the NeXusBeans exception system can be used to enforce allowed multiplicities.

10.2) NeXus instance modelling in XML tools

We now wish to examine ways of achieving similar validation for NeXus XML instance files.

10.2.1) NeXus XML instance editing in Netbeans IDE with schema validation

The Netbeans IDE offers extremely powerful XML support[12], including:

- very convenient XML file editing with XML cold-folds and coloured markup
- complete in-built support for validation against Schema
- dynamic prompting against permitted attributes, enumerations, and child types.
- a superb “cascading” graphical schema editing facility

10.2.2) (NeXus instance file modelling in Altova XML SPY)

This section omitted; readers are encouraged to investigate this powerful commercial XML tool, which offers navigation of XML structures as graphical trees with validation against XML schema.

12 In the opinion of the primary author Netbeans IDE XML support is far more powerful than the Eclipse XML tools.

10.3) Graphical NeXusBean instance modelling in UML and SysML

There are times when one might wish to graphically represent a NeXus file instance, or parts of the structure of such a NeXus file, as a graphical NeXus model instance. It affords an alternative perspective on the NeXus system, and has some illustrative value. Ideally, it would be nice to be able to generate a valid NeXus file graphically using a constrained graphical environment.

> "Wouldn't it be nice if one could just drag n' drop NeXus components
> from a palette onto a diagram to build a model."[13]
>
> Thomas Proffen[14], NIAC 2006

With NeXusBeans one can, and Kelly demonstrated that at NIAC2006. however there are some substantial limits on its practicality due to the UML metamodel (some of which the SysML systems engineering extension of UML addresses).

There are at least three ways to handle "instances" in UML[15], none of them satisfactory for the purpose of practical NeXus modelling. They are presented here for the record, with a clear caveat:

> "UML and current tools are not nearly as well suited to practical graphical
> NeXus instance modelling as my original enthusiasm at NIAC2006 suggested.
> The SysML systems engineering effort for UML may help solve that."
>
> Darren Kelly, Bondi, 2007

Nevertheless, UML instance models of NeXus have some illustrative value. Please note also that the caveats here do not apply to NeXusBean **class** modelling with UML, which is powerful, practical, and convenient.

10.3.1) NeXusBeans models as Lifeline instances in communications diagrams

- con: no facility to specify values
- con: no nesting allowed

10.3.2) NeXusBeans models as InstanceSpecifications in object diagrams:

- pro: it is possible (after the metamodel) to specify values using the Slot mechanism
- con: the assignment of values to Slots is extremely tedious in all known UML tools
- con: there is only contrived graphical nesting (which is specific to UML tools)
  and there is no validation of allowed children (compare with validated parts of composites)

This strategy can only be considered a UML novelty. It requires intimate knowledge of the UML metamodel for Slots and InstanceSpecifications, incredible patience with the UML tool. and it is not recommended. See Figure 15 for an example. WARNING: do not try this at home !

13 Apologies if misquoted by Darren Kelly.
14 Lujan Neutron Scattering Center, Los Alamos National Laboratory, USA
15 UML experts will no doubt know that instance, roles, and parts are subtly different.

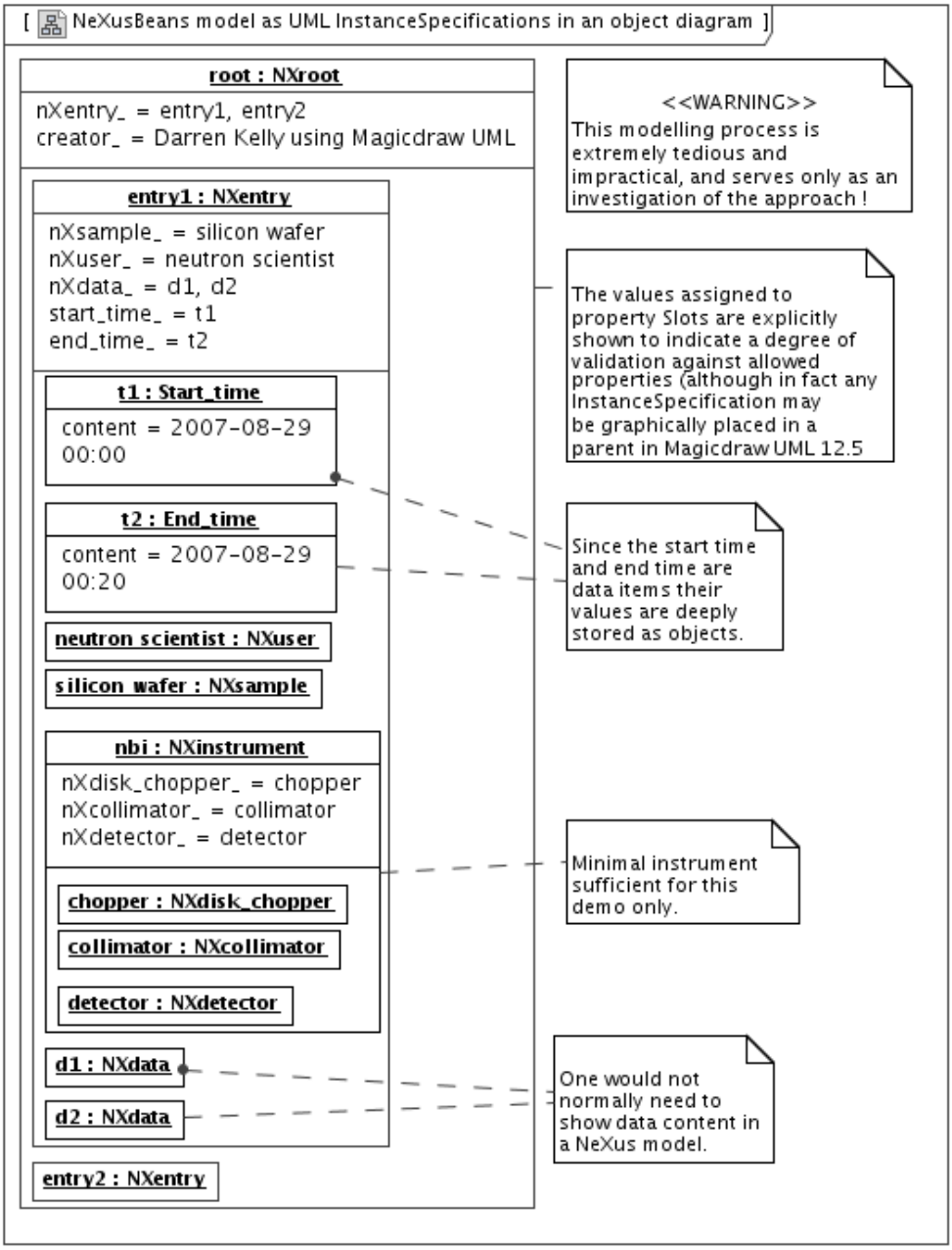


Figure 15: A simple NeXus instance model using UML InstanceSpecifications and Slots in a UML object diagram.
Not recommended practice, shown as proof-of-concept only.

10.3.3) NeXusBeans using UML part Properties within composite structures

- pro: strong graphical nesting support for Parts
- pro: strong validation of permitted child Parts
- con: values of Parts cannot be specified independently of the owning composite Class

Composite structures lend themselves well to NeXus structural modelling, due to their deeply nested hierarchical nature, however the difficulty in defining complete value configurations from top-to-bottom has been identified by many users as a major problem with UML2 metamodel, especially when used for systems modelling.

That problem is being addressed by the OMG's SysML effort, which extends UML2 to add features required for practical systems engineering. SysML introduces the concept of a property-specific type, which effectively creates an on-the-fly “subclass per part” that can carry values of a part specific to a context. This enables one to specify the values for an entire system, where the top-level composite provides the context for the entire configuration.

10.3.4) NeXusBean instance models using SysML parts within SysML blocks

TODO low priority. Requires property-specific types, not yet implemented in Magicdraw UM L

11) Recommendations for changes to the NeXus Metaformat

11.1) Recommend: separate dimensions and types in data items

Example:

```
<wavelength_range type="NX_FLOAT" dimensionality=”1” dimensions=”2” units="nm">..
<orientation_matrix type="NX_FLOAT" dimensionality=”2” dimensions=”3,3”
units="nm">..
```

Stronger option: type and dimensions as separate XML elements:

```
<orientation_matrix>
      <type="NX_FLOAT">
            <dimensions>
                  <dimension>3</<dimension>
                  <dimension>3</<dimension>
            </dimensions>
      </type>
```

11.2) Issue: clarify data items typed by NXgroups:

How should this be treated ?

```
NXentry.xml: <thumbnail type="NXnote" mime_type="{image/*}">
```

11.3) Recommend: enumeration definitions

Recommend: occurrence information be instead given as the XML Schema-like attributes `minOccurs` and `maxOccurs`:

```
<NXentry name="{entry name}" minOccurs="1">
```

11.4) Recommend: units should be imported, rather than defined as strings

Concerning the `units` data item attribute the NeXus design specifies that:

> Units of data which must conform to the standard defined by the Unidata UDunits utility (in particular, see udunits.dat)

However, the NeXus templates often depart from this recommendation, and there is no digitical enforcement of the policy. Consider definitions such as `'mili*metre'`, and `'micro.second'` found in `NXdetector.xml`. The current UDUNITs definition is available at:

> http://www.unidata.ucar.edu/software/udunits/udunits-1/udunits.txt

A candidate for improved units definition is the Units Markup Language (UnitsML) effort at NIST:

> http://unitsml.nist.gov/
> UnitsML is a project underway at the National Institute of Standards and Technology (NIST) to develop a schema for encoding scientific units of measure in XML.

11.5) Recommend: use term 'kind' of all implied enumerations.

Example, `NXshape` has a data item `Shape` with possible values

```
{"nxcylinder", "nxbox", "nxsphere", ...}?
```

The data item name `Shape` clashes with the group name `NXshape`, and would be more understandable as the `Kind` of `NXshape`, in line with popular UML and Java practice.

12) On NeXusBeans and the NeXus instrument definitions

The NeXus format also proscribes instrument definitions based on the NeXus classes. In principle, many of the stategies employed here could be expanded to include instrument schema, however that ambitious goal requires further development of the NeXusBeans class system subject to substantial NeXus metaformat review and/or adoption of XML Schema.

13) Conclusion

14) Acknowledgements

Thanks to the Bragg Insitute and ANSTO for supporting the NeXusBeans effort since Oct 2005. Thanks to the NeXus International Advisory Committee (NIAC) for hosting a presentation of the prototype of the NeXusBeans system in Feb 2006 during NIAC2006 at the Insitute Laue-Langevin (ILL) in Grenoble, France (with thanks also to then-and-now NIAC chairmen Ray Osborne and Peter Petersen, and ILL host Ron Ghosh). Thanks to ISIS and the Rutherford Appleton Laboratory for also hosting a presentation on NeXusBeans in Feb 2006. Thanks to Mark Koennecke for the Jnexus API and for discussions on the NeXus classes.

15) Appendix

TODO: Move large Java file code listings to here ?