The BEAM 3 Architecture

Summary

The BEAM Toolbox provides a rich Java™ framework for creating EO applications and plug-ins, which extend the toolbox itself. The purpose of this article is to provide an introduction to the BEAM 3.x architecture. It shall serve as a starting point from where on Java developers can continue to study the Java API documentation and BEAM source code examples.

Norman Fomferra, Brockmann Consult
May, 2006

System Overview

The BEAM Toolbox comprises the VISAT desktop imaging application, and several scientific data processors and command-line tools for format conversion and batch-mode re-projection. All BEAM applications have been implemented in Java[1] and building blocks of the BEAM system have been designed with respect to reusability and extendibility. As the following diagram shows, the high-level BEAM applications have been build on lower-level frameworks which offers programming interfaces for extension on their own. Therefore BEAM can be reused or extended at different levels of complexity. However, this article focuses solely on the BEAM core architecture which is not likely to change significantly in the near future.

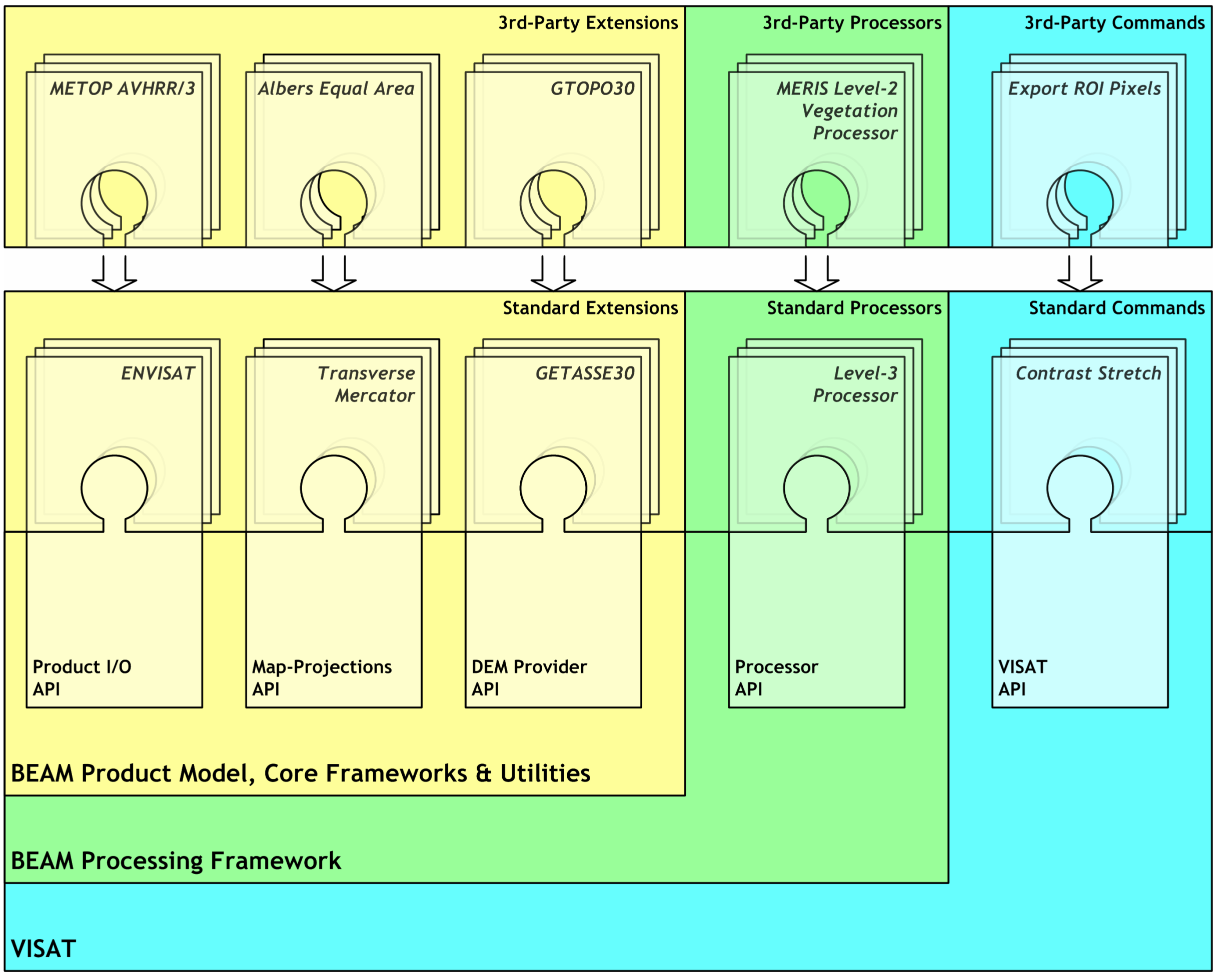

Figure 1: BEAM system overview

[1] BEAM 3.5 has been developed using J2SE 5.0

Main Concept

The main concept used for the BEAM architecture is simple and based on

1. a common, unified data model designed specifically for remote sensing products,
2. imaging, processing and analysis modules exclusively operating on this data model,
3. data readers and writers for distinct EO file formats which convert into/from this data model.

Such a product model is the key element of the BEAM architecture. It has been designed to ingest the information contained in the data products of a wide range of imaging sensors and make this data accessible through the BEAM application programming interfaces.

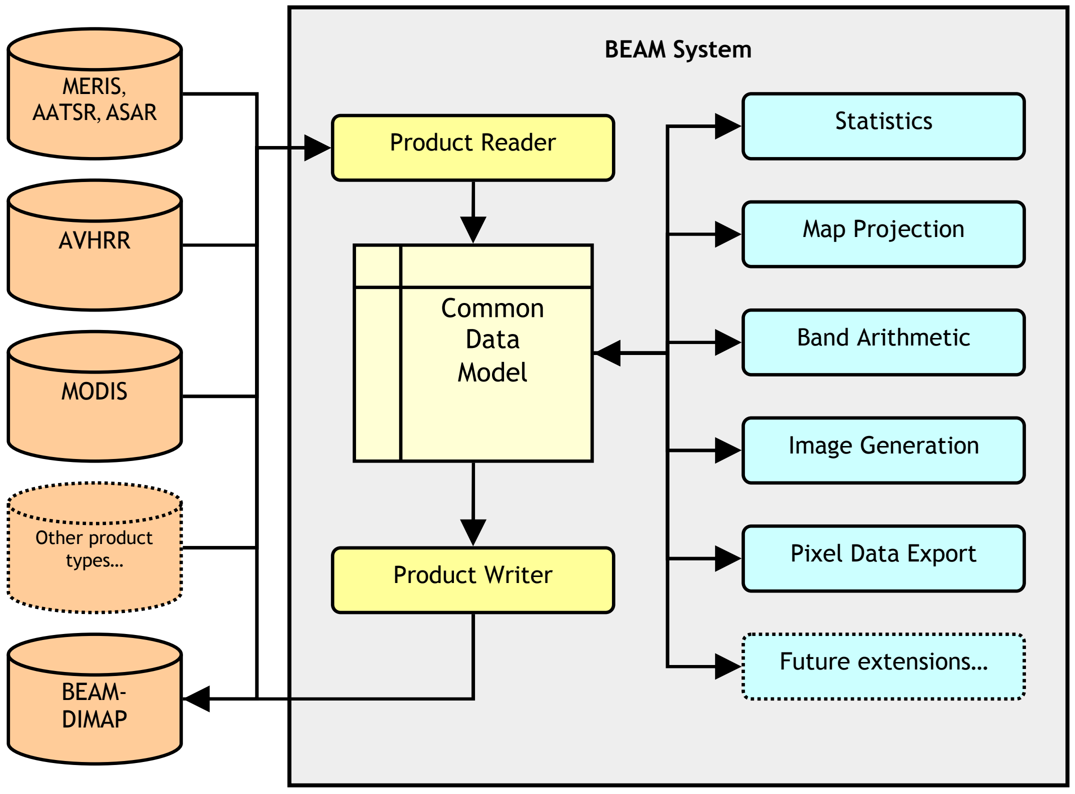

Figure 2: Generic product model concept

The BEAM software system realises this concept by providing an API which represents the static product data model and a plug-in API for both the product readers and writers. The imaging, analysis, and data processing functions are separated into modules with as little as possible mutual dependencies. BEAM 3.5 can read Level-1b and Level-2 data from the sensors MERIS, (A)ATSR and MODIS, furthermore the data from ASAR, AVHRR and AVNIR. It can also read NetCDF EO data formats conforming to the Climate and Forecast (CF) Conventions. The BEAM-DIMAP format is the standard I/O data format used by VISAT and the default output format for the BEAM data processors.

BEAM-DIMAP Format

The BEAM-DIMAP format has been developed in order to consistently support the generic product model. It is a specialisation of the *DIMAP* format, a format which has been developed by Spot Image and CNES [3]. The most important feature of the BEAM-DIMAP format is its simplicity: The actual raster data is stored in flat binary files, one file for each band, line-wise. The metadata is stored in a separate XML header. BEAM can efficiently read from and write to BEAM-DIMAP data products.

Core Architecture

Product

The main component of the product model is the `Product` class. This represents a single EO data product which can exist as either a physical file or as an in-memory object in the computer's RAM. A product represents one or more earth observation scenes and is therefore a composite of one or more raster data bands. Optional components of a product are the geo-coding, ancilliary data stored as tie-point grid, flag-codings and the metadata.

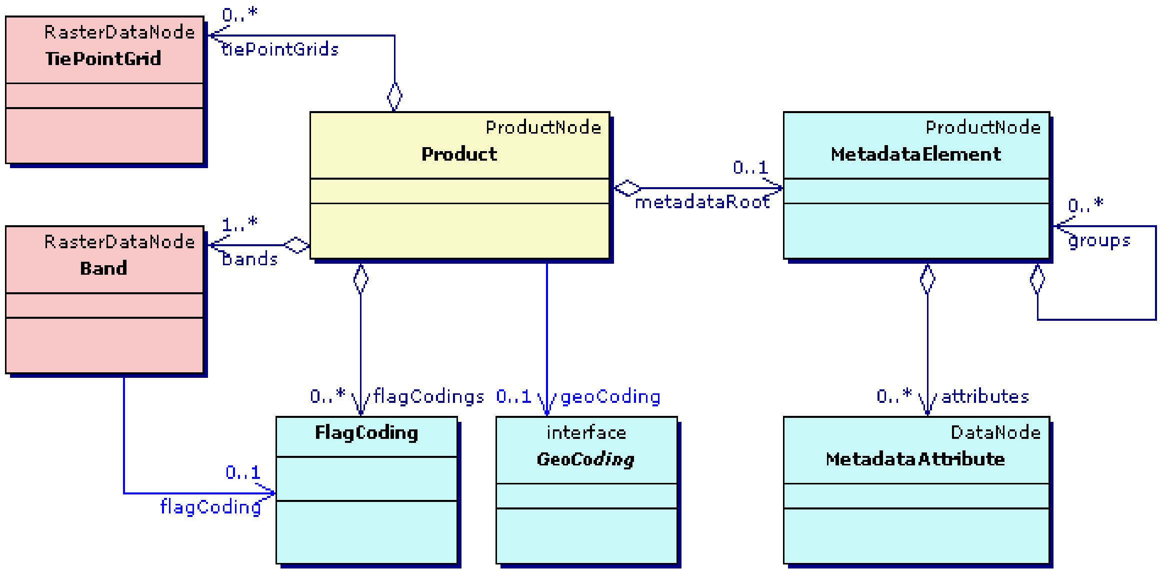

Figure 3: UML class diagram showing the containment hierarchy of the product model

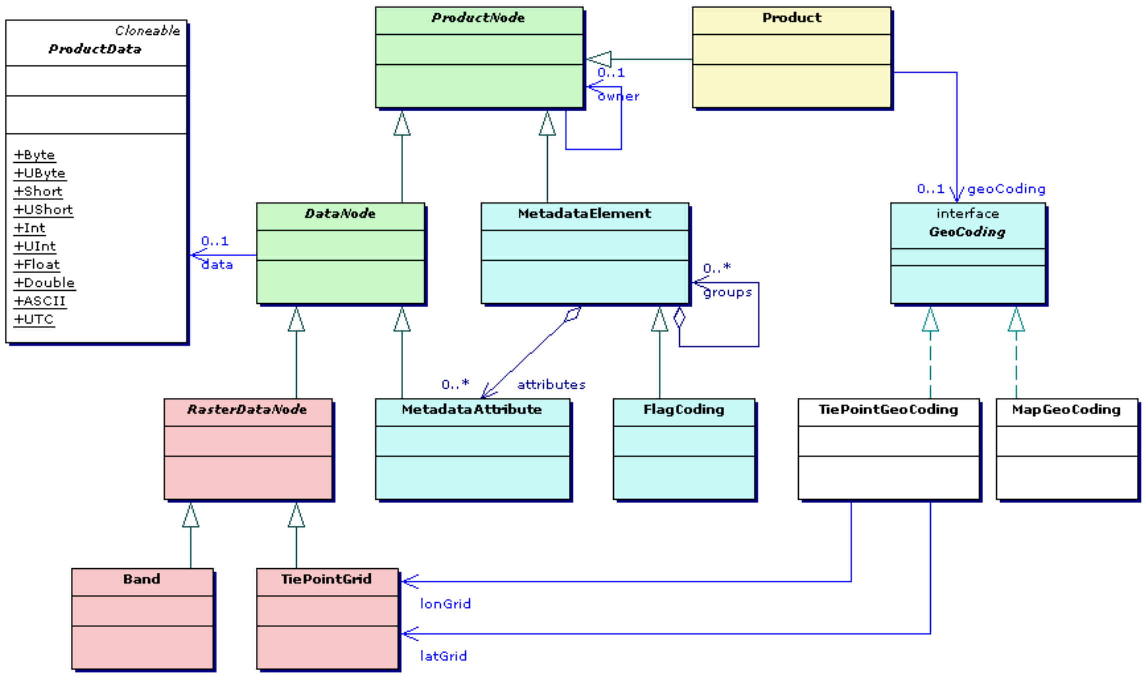

Figure 4: UML class diagram showing the inheritance hierarchy of the product model

Band and Tie-point Grid

Objects of the `Product` class can have one or more bands, modelled by the `Band` class. A band provides sample values, geophysical parameter values or quality flags as a function of the pixel position. A `Product` can also have any number of `TiePointGrid`s, which internally store anciliary data on a regular sub-grid. All of the imaging, processing and analysis functions are applicable to tie-point grids and bands as well since they are derived from the same base type `RasterDataNode`. Common properties to all raster data nodes are a scaling factor and a scaling offset which convert raw counts into geophysical units, and a no-data value which indicates missing pixel values. These properties play a vital role in the BEAM product model.

Geo-coding

A `Product` is geo-coded if is associated with a `GeoCoding` object. The geo-coding is responsible for the transformation of pixel coordinates into geographic coordinates and vice-versa. As the inheritance class diagram shows, there are two most important implementations of `GeoCoding` provided in BEAM, the `TiePointGeoCoding` which is based on latitude/longitude tie-point grids and `MapGeoCoding` which uses a map-projection in order to perform the coordinate transformations. Further geo-codings are the `PixelGeoCoding` which operates on the latitude and longitude values given at the band resolution and the very special `ModisTiePointGeoCoding`, which performs a correction of the MODIS bow-tie effect. Alternatively , and since the BEAM 3.5 version, geo-codings can also be assigned band-wise (association not shown in Figure 3).

Metadata

A product's metadata, such as the main and specific product headers and annotation datasets are stored as nested groups of `MetadataElement`s. A product can have any number of metadata elements. Most metadata is composed of single name-value pairs. These are stored as generic `MetadataAttribute`s as part of a `MetadataElement`.

Flag-coding

Flag-codings are a special type of metadata. The `FlagCoding` class describes the classification/quality flags stored as per-pixel bit fields in "flag bands". A product can have multiple flag bands and therefore also multiple flag-codings. Flag-codings play a very important role in BEAM, e.g. for the exclusion of geophysical values in statistics or for the creation of bit-mask overlays derived from the flag values in a flag band.

Virtual Bands and Band Arithmetic

A special type of band is represented by the `VirtualBand` class. The raster data of this band is computed from a mathematic expression instead of being read directly from a product data file. The mathematical expression is given as a character string and can be composed of the names of any band, tie-point grid or single flag value contained in the product. The strength of the virtual band concept is its great flexibility without requiring any additional disk storage.

Product Data

As the inheritance diagram in Figure 4 shows, `Band`s, `TiePointGrid`s and `MetadataAttribute`s are `DataNode`s. A data node stores the actual product data in objects of the `ProductData` class. This class is an abstraction of arrays of primitive data types, namely 8-,16,-32-bit integer and 32-,64-bit floating point numbers.

Product Reader and Writer

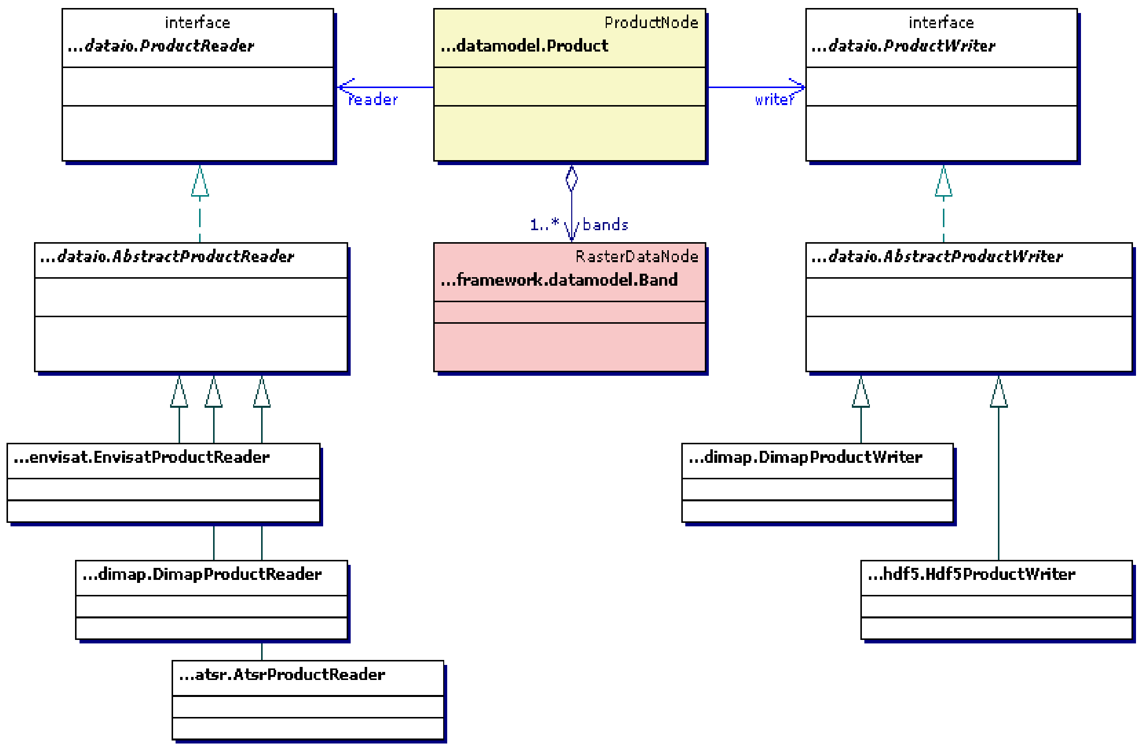
interface
...dataio.ProductReader
ProductNode
...datamodel.Product
interface
...dataio.ProductWriter
reader
writer
1..*
bands
...dataio.AbstractProductReader
RasterDataNode
...framework.datamodel.Band
...dataio.AbstractProductWriter
...envisat.EnvisatProductReader
...dimap.DimapProductWriter
...dimap.DimapProductReader
...hdf5.Hdf5ProductWriter
...atsr.AtsrProductReader

Figure 5: UML class diagram showing the product class with its associated data reader and writer

Furthermore a product can have a `ProductReader` attached to it. Objects of the `ProductReader` interface allow a product to read data from any kind of data source. Since a `ProductReader` is only an interface, concrete implementations of product readers must be provided for each product type or group of similar product types. Accordingly, a product can have a `ProductWriter` attached to it. The concrete product readers and writers shown in the class diagram above, such as the `EnvisatProductReader`, are all derived from abstract reader and writer classes which already provide a set of default implementations for frequently used functions.

When, for example, VISAT asks a `Band` of the `Product` for data to be displayed in an image, the band delegates to its `ProductReader` in order to obtain the requested data. The concept of having a fully abstract data source and data sink for a product makes the product completely independent from a special data format. Within BEAM, the data source can also be another `Product` object. When a map projection is applied to a product, an instance of a new, map-projected product is created in this case. The new product will have a `ProductReader` which uses the original product as data source.

VISAT Architecture

From the very beginning, VISAT has been designed to be easily extendible. The class diagram shown in Figure 6 gives an overview of the key components of the flexible VISAT architecture.

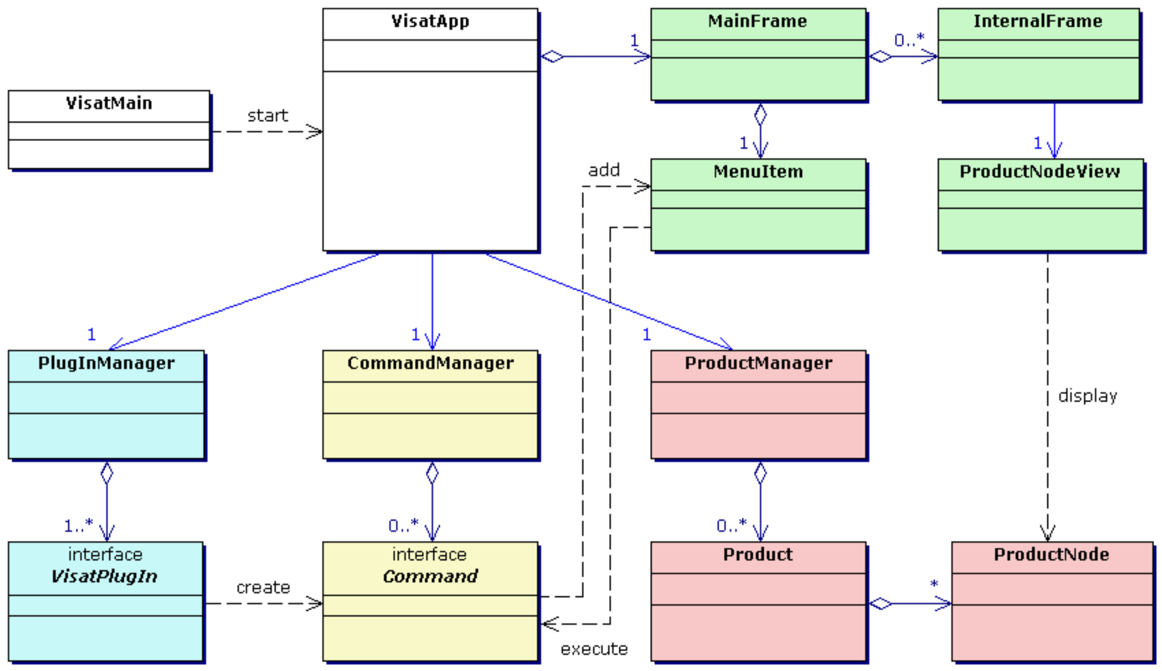

Figure 6: VISAT architecture overview

When VISAT is started, it dynamically loads all plug-ins located in a dedicated location within the BEAM installation directory. After a plug-in has been loaded, VISAT asks it to initialise itself. At initialisation time, a plug-in can already access all BEAM framework APIs. Usually a plug-in creates one or more specific commands which it registers with VISAT's command manager. The commands implement and provide the actual plug-in functionality. VISAT will use the properties of the command to automatically create new menu items or toolbar buttons. When a user clicks the menu item, the command is activated and the plug-in specific code is executed. The plug-in specific code can then ask VISAT for the currently selected objects such as product, band or image and then apply some kind of processing or modifications to it.

Package Overview

The BEAM Java library is organised in a number of packages. Not all of them are public API. The most important API packages are:

- **`org.esa.beam.framework`**: This package defines sub-packages for the internal EO data models, EO data I/O, EO data processing and GUI application frameworks. The latter framework is based on the common model-view-controller (MVC) architecture.

- **`org.esa.beam.utils`**: A package which defines many ready-to-use functions and components which are frequently used in application development with special attention to EO applications.

The rest of the BEAM packages are either application-specific or provide some concrete, reusable implementations of the interfaces defined in the framework:

- **`org.esa.beam.visat`**: Implements the VISAT application and user interface and the standard set of VISAT plug-ins (which implement the actual VISAT functionality)
- **`org.esa.beam.processor`**: Implements the standard BEAM data processors
- **`org.esa.beam.dataio`**: Implements the standard set of reader and writer plug-ins for the product types supported by BEAM
- **`org.esa.beam.dataop`**: Implements the standard set of map projection plug-ins and provides the band arithmetic

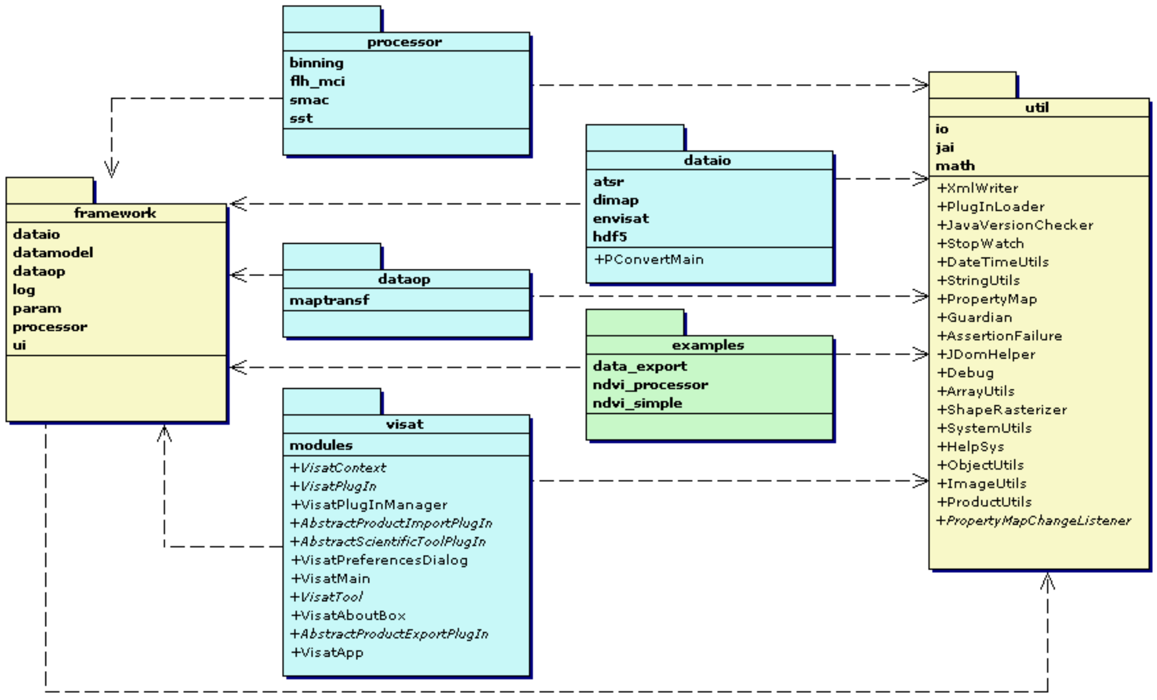

Figure 7: The main sub-systems of BEAM

The framework package defines all interfaces, classes and recurring design patterns used to implement concrete software modules such as VISAT, the processing tools, data product readers and writers or map projections. The following framework packages exist

- **`org.esa.beam.framework.datamodel`**:
  In-memory presentation of a data product - the most important BEAM package
- **`org.esa.beam.framework.dataio`**:
  Product Data I/O plug-in framework
- **`org.esa.beam.framework.dataop`**:
  Product data manipulation interfaces and design patterns
- **`org.esa.beam.framework.processor`**:
  Product processing framework, used by all scientific data processors
- **`org.esa.beam.framework.param`**:
  Generalized parameter handling
- **`org.esa.beam.framework.ui`**:
  User interface related classes and sub-packages

More Information

The BEAM web site [1] contains a lot of valuable information for software development with BEAM. The detailed BEAM Java API documentation [2] can be found here as well, and these are also contained in the local BEAM installation folder (`api-docs.zip`).

The BEAM installation also contains some example source code which can serve as a starting point for developing new applications from scratch or for extending the BEAM toolbox (`examples.zip`). A review of the actual BEAM source code (`src.zip`) should be considered as a valuable source for some implementation details.

Many plug-ins on the BEAM web site are also supplied with source code and demonstrate different API use cases. Furthermore, the BEAM unit-level test code (`src_unit.zip`) can also serve as a source for API usages.

BEAM 4

BEAM is still under development.,BEAM 4 is currently being developed in order to take new features and new use cases onboard. Some APIs of BEAM will change during this development. However, the core API including the product model has been relatively stable since BEAM 1.0. During the past years many new features have been developed and support for many sensors has been added. This work has been performed without affecting the underlying product model and the related APIs too much. For example, the ability to have a different `GeoCoding` in each band has been introduced in BEAM 3.5 in order to support data from the JAXA ALOS satellite. While it is not expected that the core API will drastically change during BEAM 4 development, it is clear that some modules will have to.

VISAT and Data Processor Redesign

Especially the user interfaces and architectures of the VISAT application and the data processors will undergo significant changes. It is planned to migrate them on top of a industry prooven, plug-in oriented computing environment which manages plug-ins and plug-in lifecyles (install, update, uninstall) on the fly. Considered environments are the rich client platforms of the *Eclipse* [4], *NetBeans* [7] projects and more general *OSGi Technology* [6] implementations such as *Eclipse Equinox* [5]. The main goal is to make BEAM 4 *consist* of versioned, updateable plug-ins instead of simply allowing BEAM to be extended by plug-ins. This will introduce a new and mature application architecture where also the application core itself is definitely an exchangeable plug-in. The new architecture will then also allow for on-line downloads and updates of plug-ins from within the applications plug-in update manager.

Upcoming Core API Changes

The `ProductReader` and -`Writer` APIs will be incompatibly changed in order to add the capability of loading single raster data tiles and reloading other data only on demand only. This will be done in order to improve the memory management and display performance for high image resolutions (e.g., ALOS, Landsat, SAR instruments in general). The API design will possibly introduce a compatibility to the standard Java Image I/O API.

The `GeoCoding` class will also be incompatibly changed so that it is composed of a `GeoPositioning` and a `CRS` (Coordinate Reference System) object. The CRS will be designed in accordance to the *OGC* [8], *EPSG* [9] and the *ISO 19111* standard.

The API of the `RasterDataNode` (and thus `Band` and `TiePointGrid`) will possibly be extended in order to meet some interesting aspects stated in the *OGC Coverage Specification* [8] and the *ISO 19123* standard. For example, this comprises the support of irregular grid layouts or the capability to directly obtain pixel values for given CRS coordinates (currently only the image pixel coordinates must be given).

Links

[1]	BEAM Web Site	http://www.brockmann-consult.de/beam/
[2]	BEAM Java API Docs	http://www.brockmann-consult.de/beam/doc/api/index.html
[3]	Spot Image DIMAP	http://www.spotimage.fr/dimap/spec/dimap.htm
[4]	Eclipse RCP	http://wiki.eclipse.org/index.php/Rich_Client_Platform
[5]	Eclipse Equinox	http://www.eclipse.org/equinox/
[6]	OSGi Technology	http://www.osgi.org/osgi_technology/
[7]	NetBeans Platform	http://www.netbeans.org/products/platform/
[8]	OGC Home Page	http://www.opengeospatial.org/
[9]	EPSG Home Page	http://www.epsg.org/