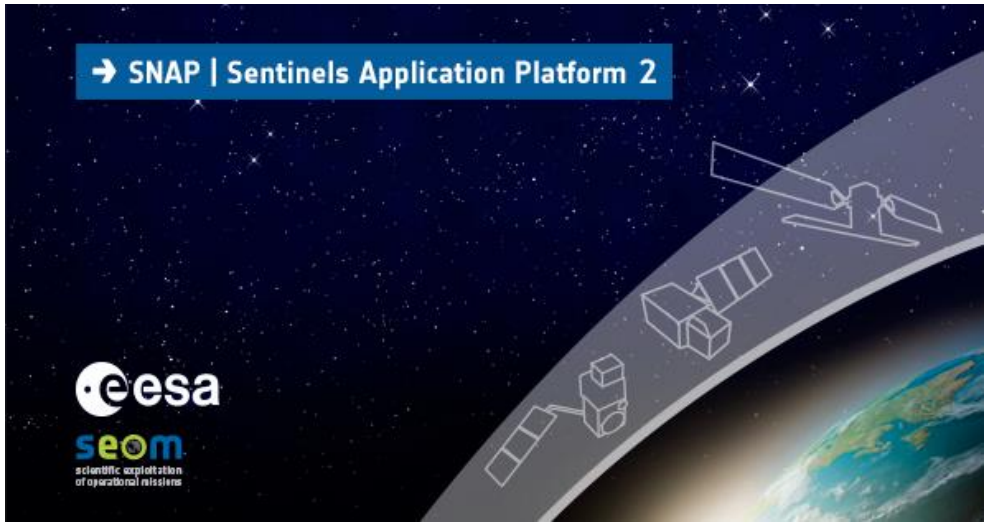
1ST ESA SNAP HACKATHON

SNAP Architecture

SNAP Development Team

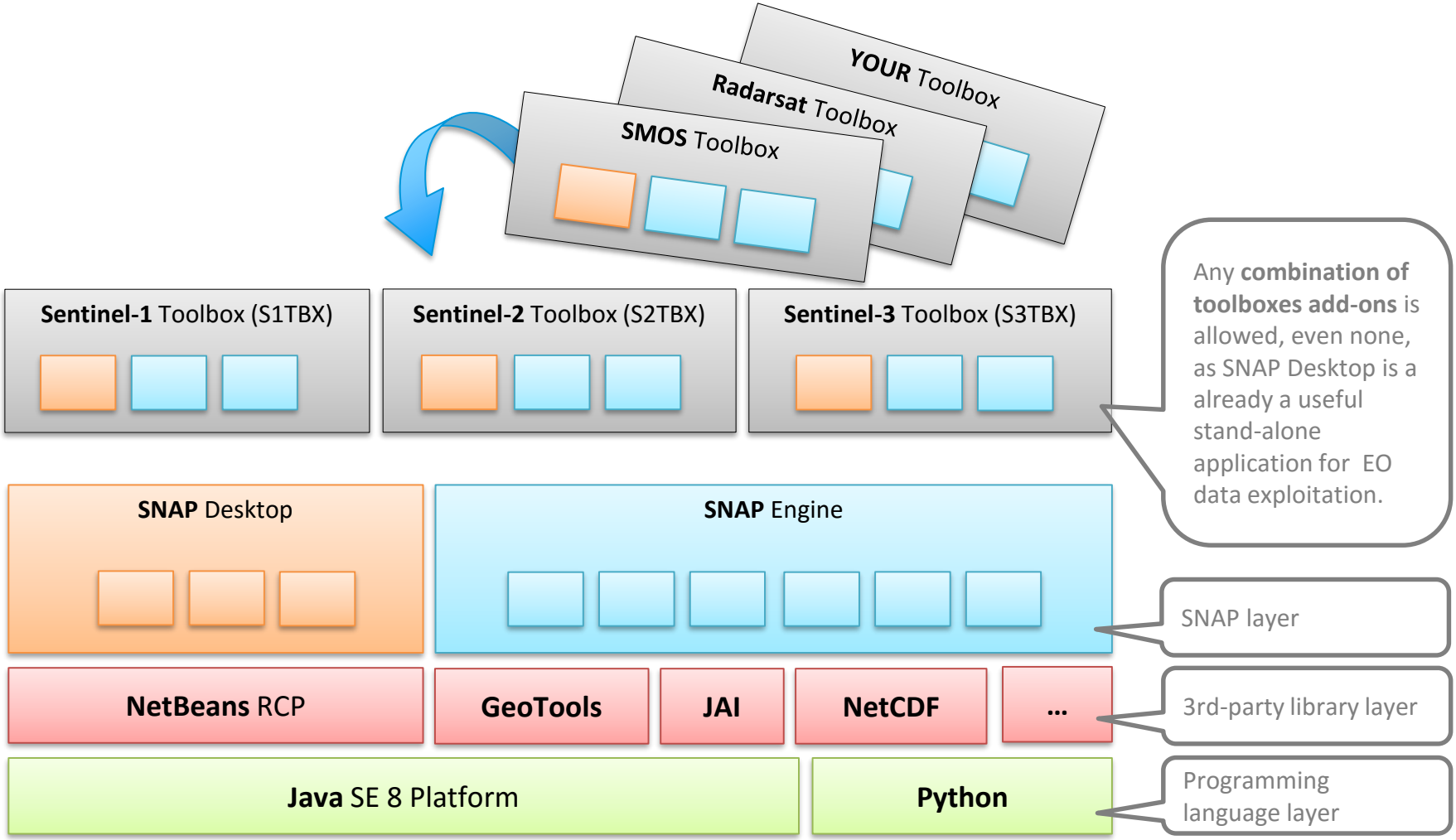


ESA ESRIN, 15 + 16 October 2015

SNAP Architecture

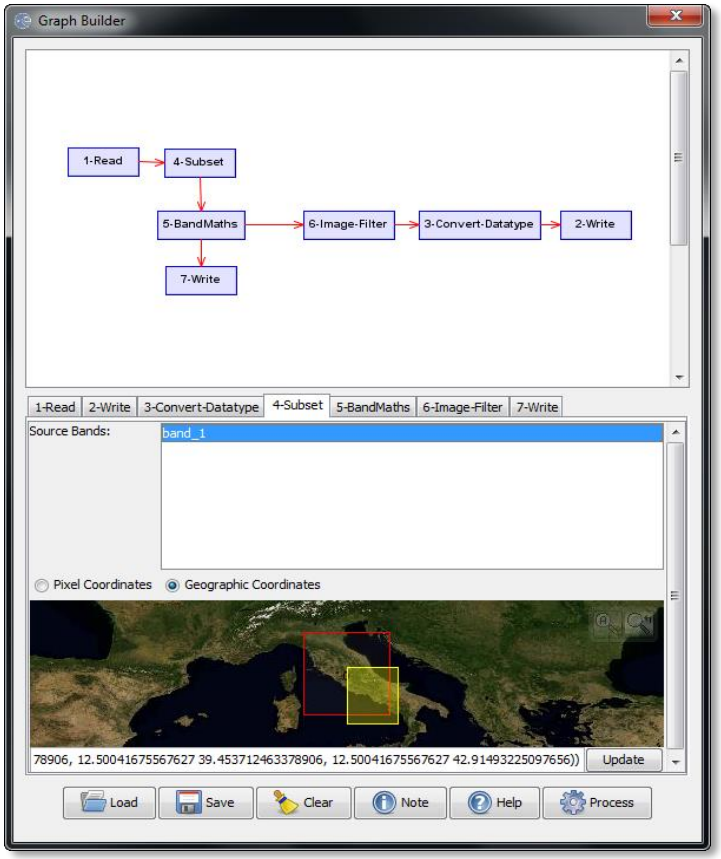
- Dynamic, module-based architecture, with various extension points and extensions
- Install and update of extension modules
- Use from Java and Python programs, extend by Java and Python plugins
- Write a SNAP plugin, use it in all toolboxes
- High-level architecture comprises two subsystems
  - **SNAP Engine**, the core and command-line interface
  - **SNAP Desktop**, the graphical user interface

SNAP Architecture



Graph Processing Framework

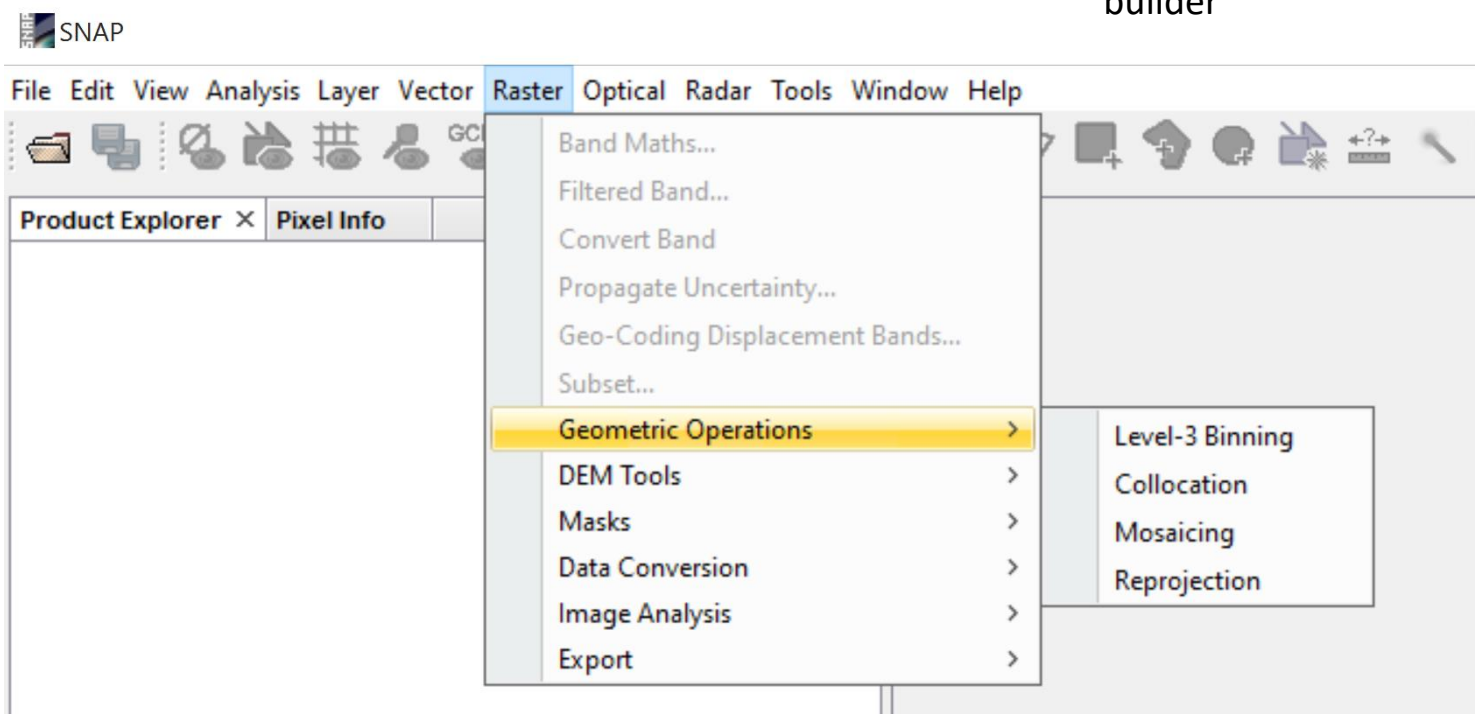
- Majority of SNAP „functions“ are implemented as operators
- Each operator can be invoked from SNAP Desktop and from the command line
- Processing chains („graphs“) are configured in XML files
- Graphical Graph Builder
- Graph Processing Tool (gpt) for executing of graphs (chains)
  >gpt -help



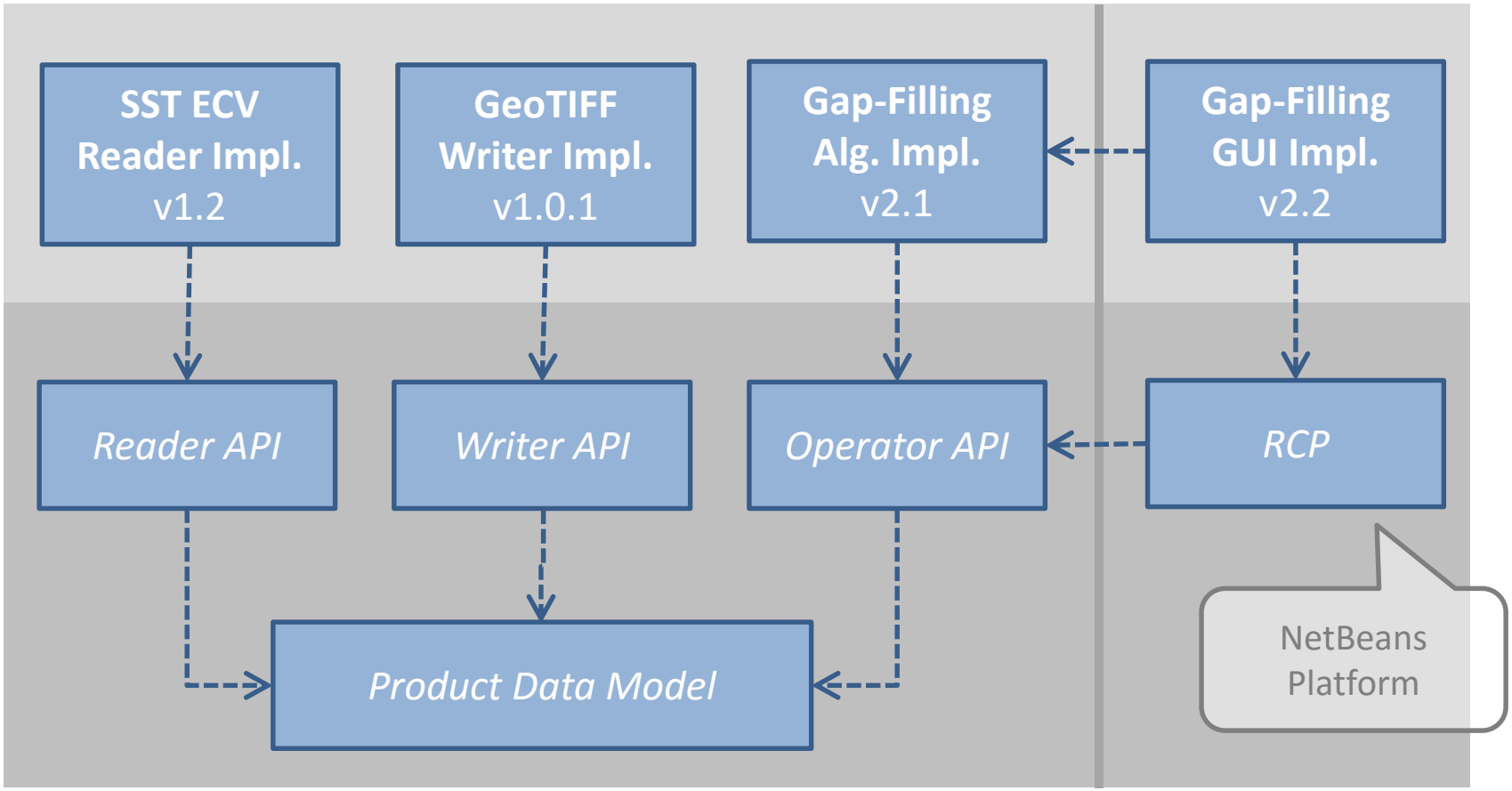
Generic Functions and Tools

- Applicable for all toolboxes and wide range of sensors
- Raster data and vector data tools
- Visualisation
  - Multi-layer displays, layer editors
  - Image, mask, shapes overlays
  - Colour management, fast navigation

- Data Analysis
  - Various statistics and plot types
  - Spectrum display (optical)
- Data processing
  - Reprojection, Collocation, Mosaicing
  - Level-3 processor
  - Graph processing, ürocessing graph builder



SNAP Basic Concept



SNAP Engine **SNAP Desktop**

Basic Concepts

- SNAP has been designed as a **<u>EO raster data</u> visualisation, analysis and processing system**
- However **vector data is supported** through an OGC compatible model (GeoTools)
- In SNAP, almost all EO data processing is done through a *pull-processing* approach:
  - If some raster data is required, such as for display or processing, it is obtained from some raster data source
  - The source can be a product reader that reads from a file, or a raster data operator that computes its output from other raster data
  - This way, raster data read/computation requests are made from one raster data node to another forming directed acyclic imaging graphs

Basic Concepts – Product Data Model

- **Product** :
  - has a file-name, type-name
  - has a scene raster size
  - has a ***GeoCoding***
  - has a model coordinate reference system (CRS)
  - has any number of ***RasterDataNodes***
  - has any number of ***VectorDataNodes***
  - has ***MetadataElements***

- **RasterDataNode**:
  - has a name, unit, description
  - has a raster data type and size
  - provides raster data
  - can be a ***TiePointGrid***, ***Band***, ***VirtualBand***, ***FilteredBand***, ***Mask***

Basic Concepts – Product Data Model

- **TiePointGrid**:
  - a grid that is up-sampled by linear interpol. to a larger raster size
- **Band**:
  - receives its raster data on-demand from a ***ProductReader*** or an ***Operator*** or a ***MultiLevelImage***
- **VirtualBand**:
  - a band which computes its raster data on demand from an band maths expression
- **FilteredBand**:
  - a band which computes its raster data on demand from filter kernels
- **Mask**:
  - a band with 1-byte Boolean pixels which can be used as ROI

Basic Concepts - Behaviour

- **ProductWriter**:
  1. writes header once (structure + metadata)
  2. writes raster data to output on demand
  3. requests raster data from sources
- **Operator**:
  1. One target product, N source products
  2. computes raster data for target on demand
  3. requests raster data from sources
- **ProductReader**:
  1. reads product header once (structure + metadata)
  2. reads raster data regions from input on demand

Programming Language

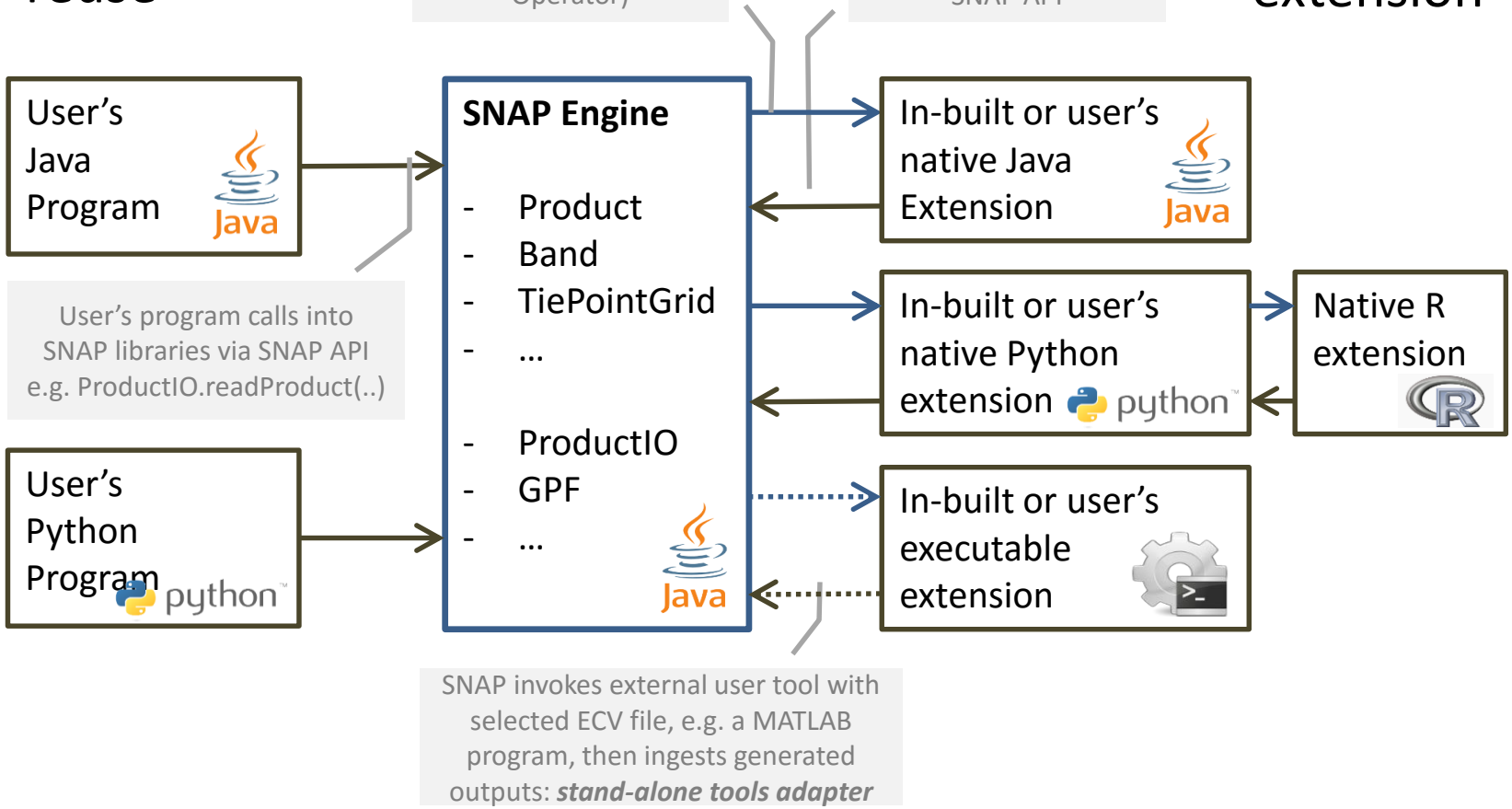
- SNAP is programmed entirely in **Java**.
  Why? Java is cool and beautiful.
  Java is also very fast (as C), secure,
  and truly platform independent.
- So we have a **SNAP Java API**

- SNAP now supports also **Python**.
  Why? Python is cool and beatiful, and most of you use it.
  However, for raster data processing plain Python is very, very slow.
  Unless you use *numpy* and the like.
- Therefore SNAP supports two Python implementations
  - **CPython 2.7, 3.3, 3.4** to use with numpy etc (→ snap**py**)
  - **Jython 2.7** to develop also SNAP GUI extensions
- There is **no dedicated SNAP Python API**
- Instead we **reuse the SNAP Java API from Python**

SNAP API Use Cases

Use case #1: Toolbox reuse

Use case #2: SNAP extension



Development Environment

- **SNAP 2.0 beta 8**, the installed version
- Python coding
  - **Python 2.7, 3.3 or 3.4**
  - **git** to clone code from github.com/senbox-org, especially *snap-examples*
  - **PyCharm**, probably the best Python IDE
- Java coding
  - **JDK 1.8 update 60**
  - **IntelliJ IDEA** – for sure the best Java IDE
  - **Maven** – to compile and package Java sources

Java API Docs

- Java API Docs
  - SNAP Engine http://step.esa.int/docs/v2.0/apidoc/engine/
  - SNAP Desktop http://step.esa.int/docs/v2.0/apidoc/desktop/
- Start Python coding
  - by looking at the examples
  - by looking up classes used in the Java API Docs
- How to interpret as Python developer?

Java API Docs - ProductIO

ProductIO (SNAP Engine 2 ×

step.esa.int/docs/v2.0/apidoc/engine/

Apps G W a D Py SNAP BC PFA GODIVA2 Data Visu... Gemeinschaftspraxis...

All Classes

Packages

com.bc.ceres.binding
com.bc.ceres.binding.access

ProductData.UInt
ProductData.UShort
ProductData.UTC
ProductFilter
ProductFlipper
ProductFlipperExt
ProductIO
ProductIOException
class in org.esa.snap.core.dataio
ProductIOPlugin
ProductIOPlugInManager
ProductLayerContext
ProductManager
ProductManager.Event
ProductManager.Listener
ProductNamespaceExtende

step.esa.int/docs/v2.0/apidoc/engine/org/esa/snap/core/dataio/ProductIO.html

static ProductReader	getProductReaderForInput(Object input) Tries to find a product reader instance suitable for the given input.
static ProductWriter	getProductWriter(String formatName) Gets a product writer for the given format name.
static String[]	getProductWriterExtensions(String formatName) Gets an array of writer product file extensions for the given format name.
static Product	readProduct(File file) Reads the data product specified by the given file.
static Product	readProduct(File file, String... formatNames) Reads the data product specified by the given file.
static Product	readProduct(String filePath) Reads the data product specified by the given file path.
static void	writeProduct(Product product, File file, String formatName, boolean incremental)

Java API Docs → Python code

- From ProductIO:

```
static Product        readProduct(String filePath)
                      Reads the data product specified by the given file path.
```

```
from snappy import ProductIO
p = ProductIO.readProduct('/eodata/test.nc')
```

- From Product:

getBand

```
public Band getBand(String name)
```

Returns the band with the given name.

```
b13 = p.getBand('radiance_13')
```

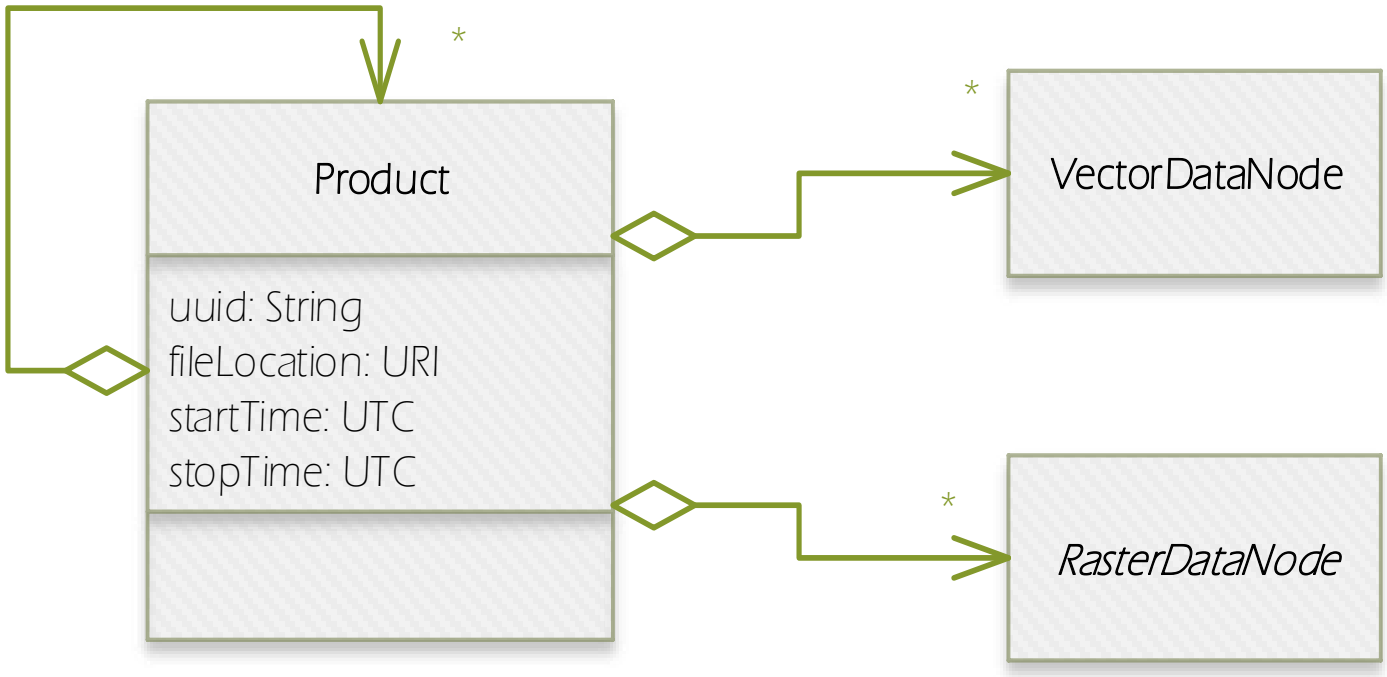
Java API Docs → Python (cont.)

Java to (C)Python type conversions:

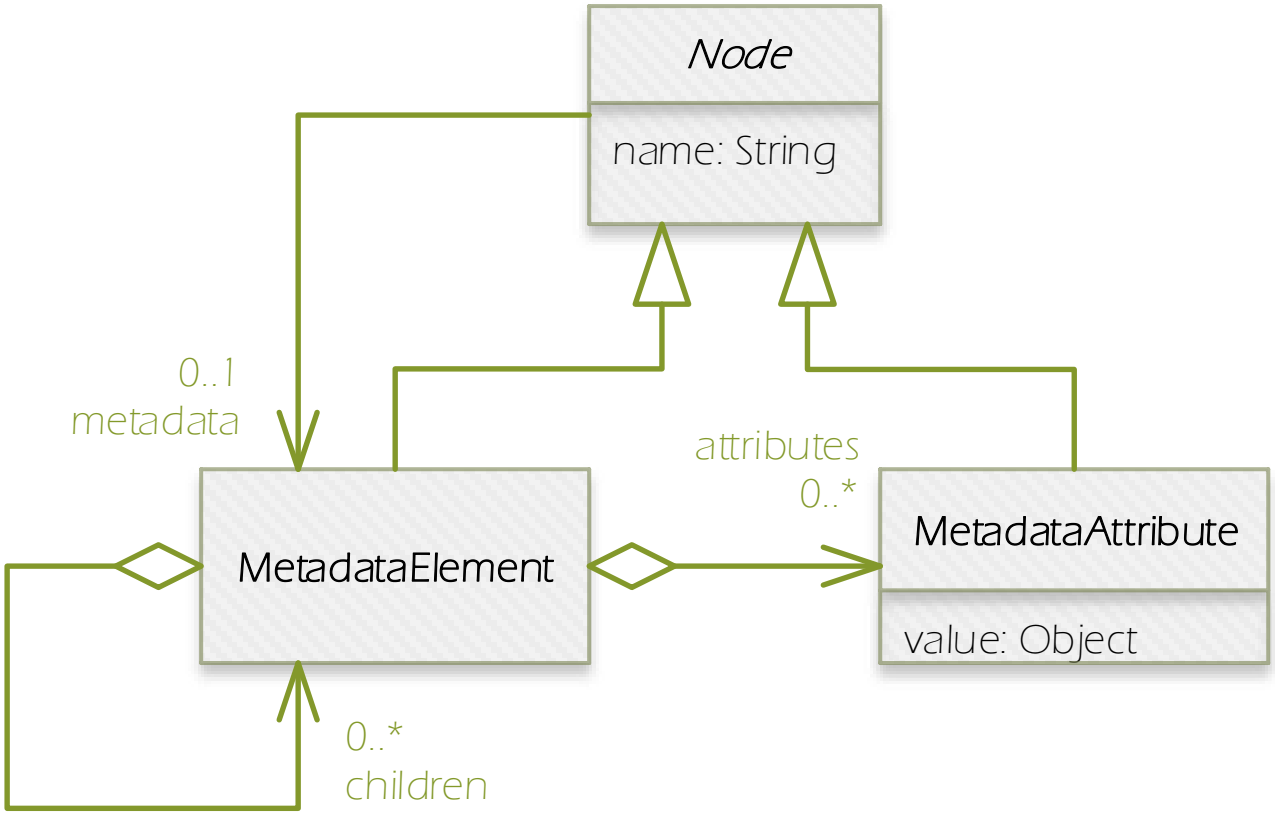
Java	Python
`java.lang.String`	`str`
`boolean`	`bool`
`byte, short, int, long`	`int`
`float, double`	`float`
`byte[], short[], int[], long[], float[], double[]`	*Python buffer protocol,* *Python sequence protocol* → e.g. pass to **numpy** array constructors
`null` *value*	`None` *value*
any `java.lang.Object`	*Wrapped Java object*

http://jpy.readthedocs.org/en/latest/reference.html

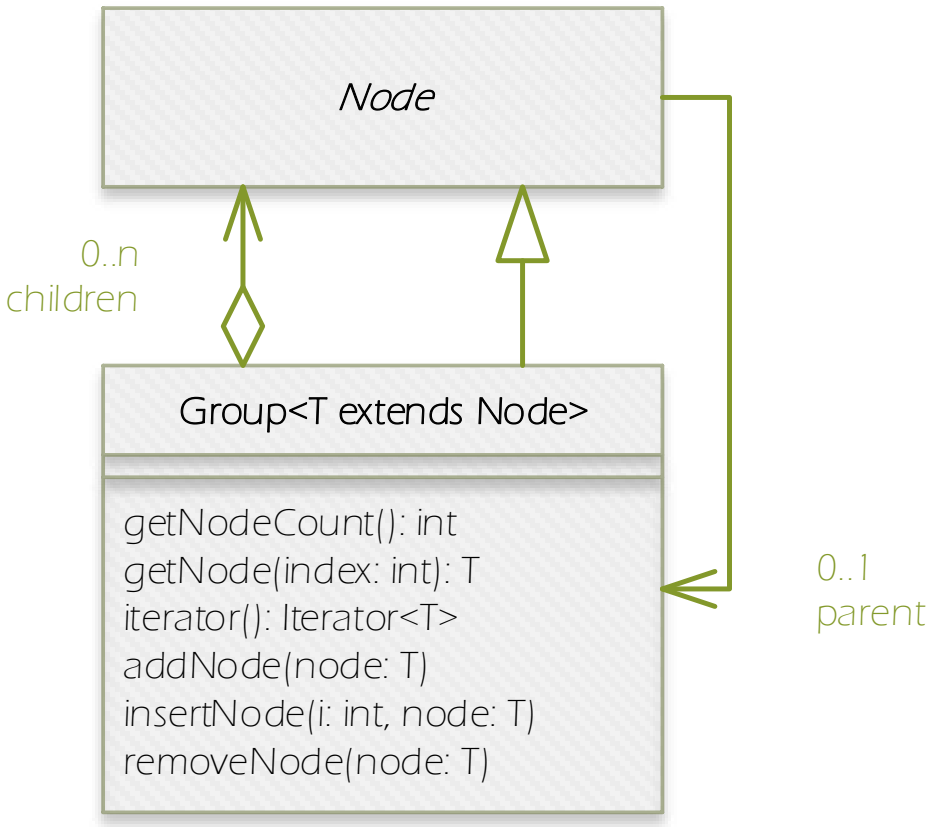
Product Data Model - Product



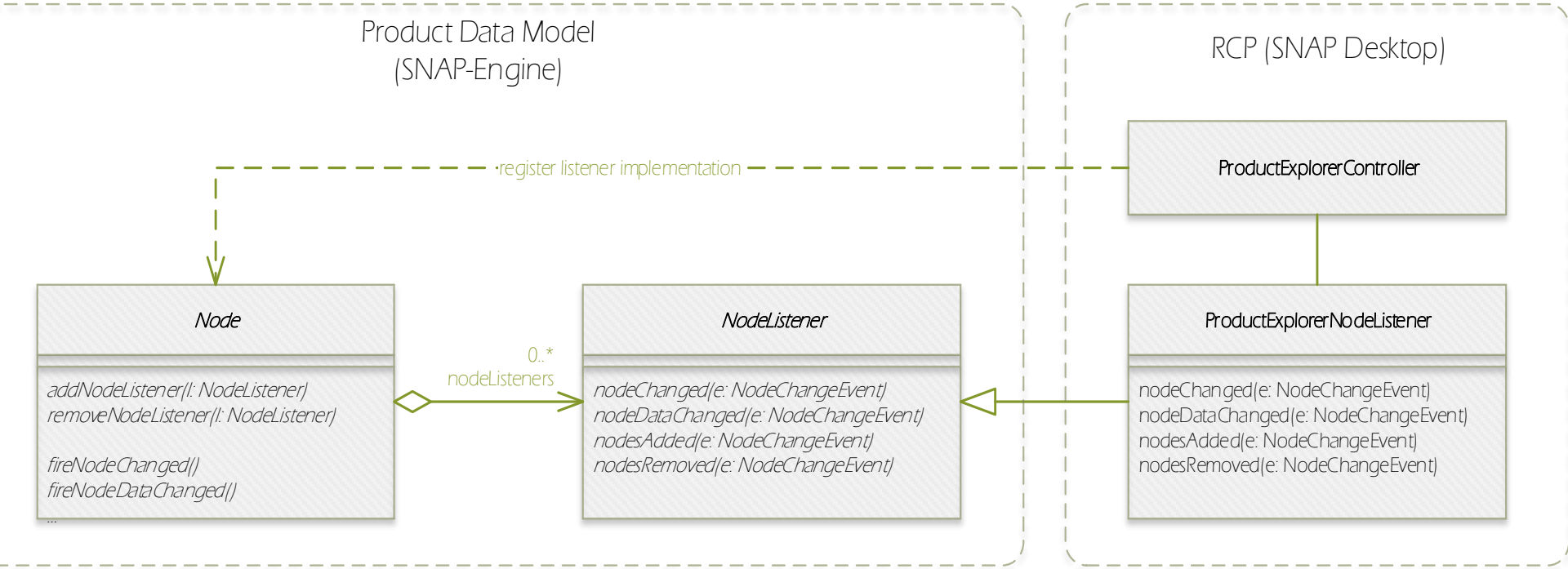
Product Data Model – Metadata



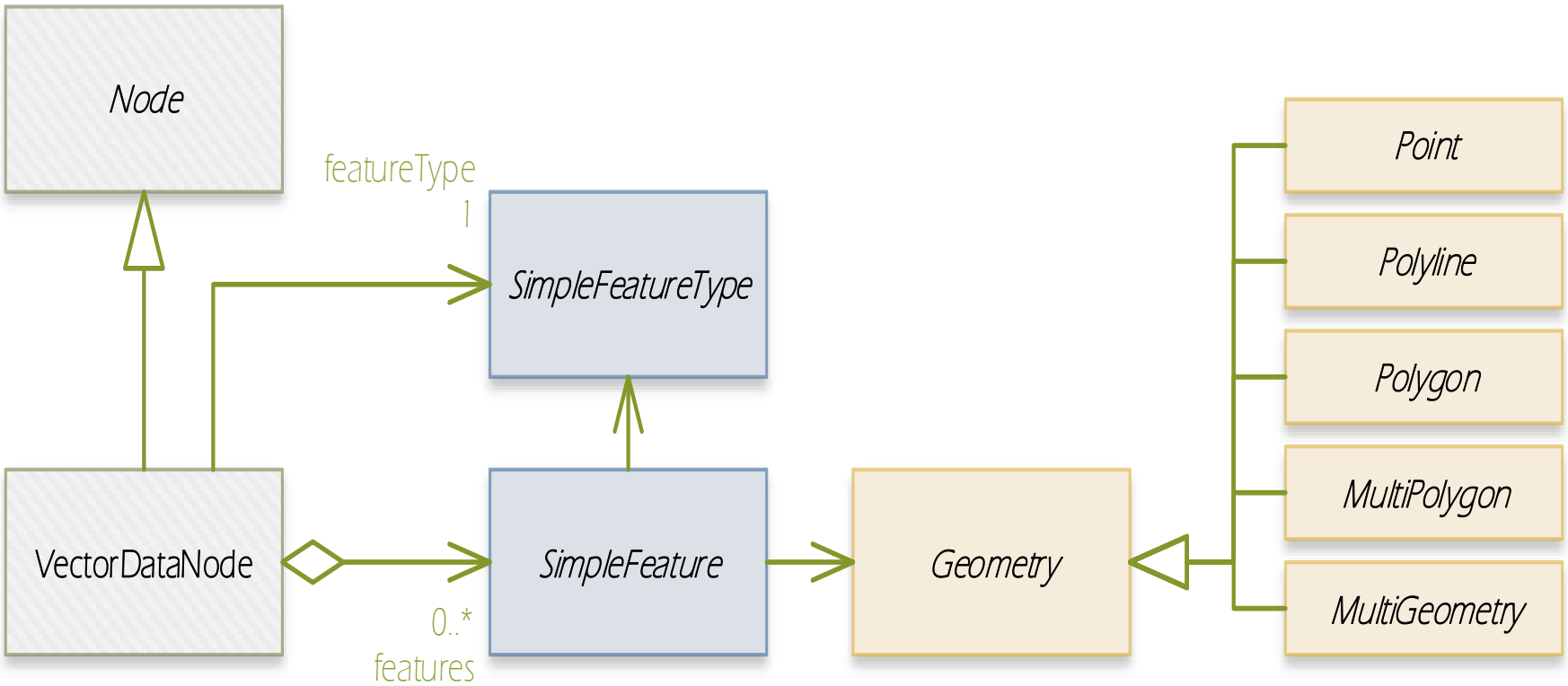
Product Data Model – Node Groups



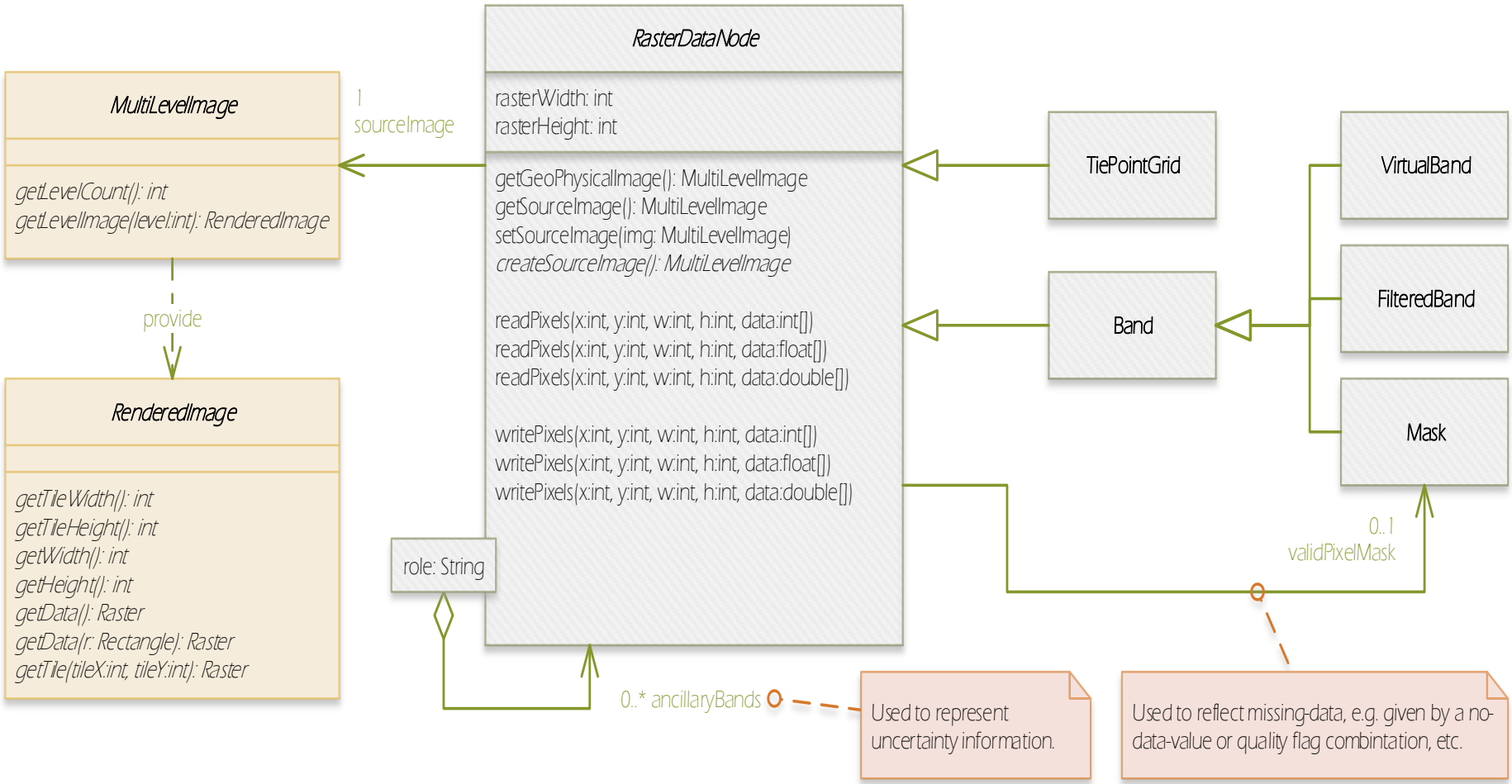
Product Data Model – Node Observation



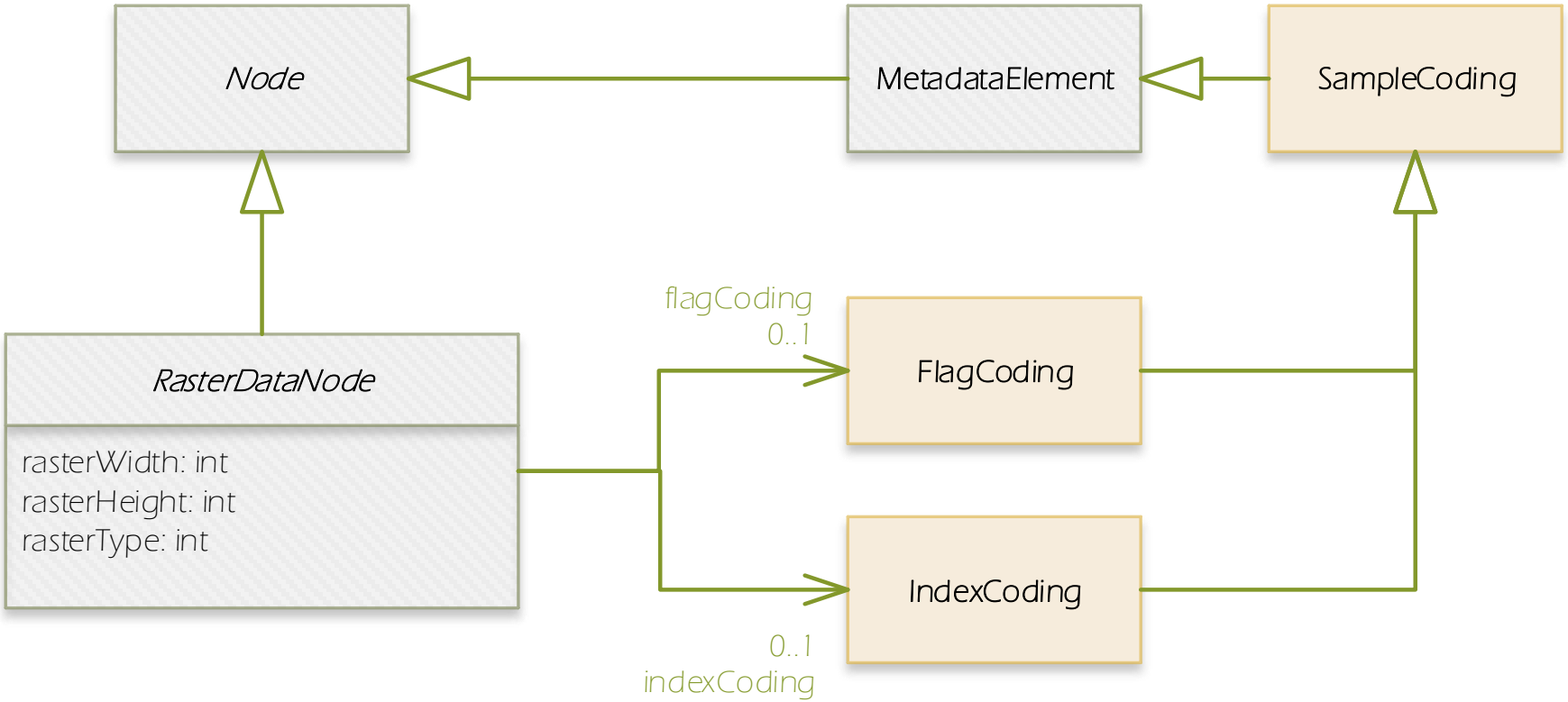
Product Data Model – Vector Data



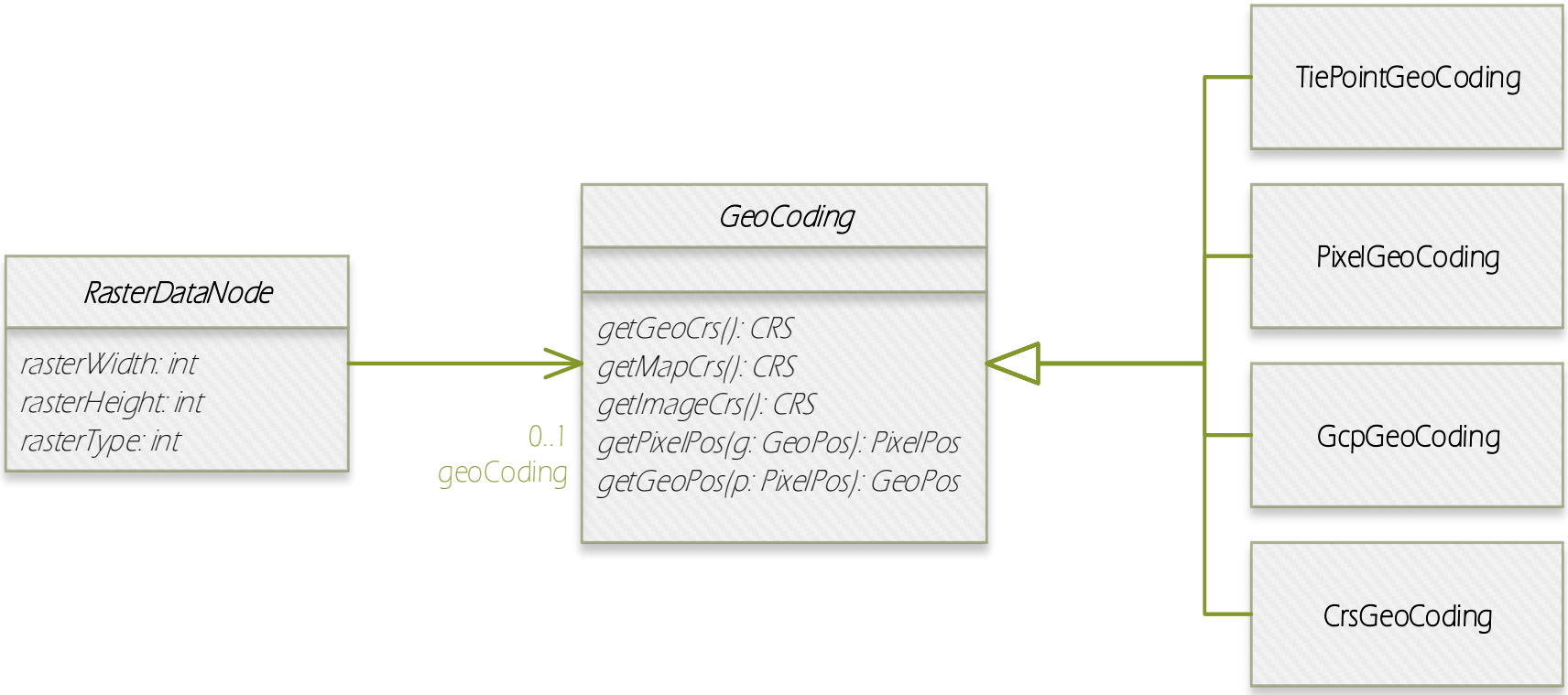
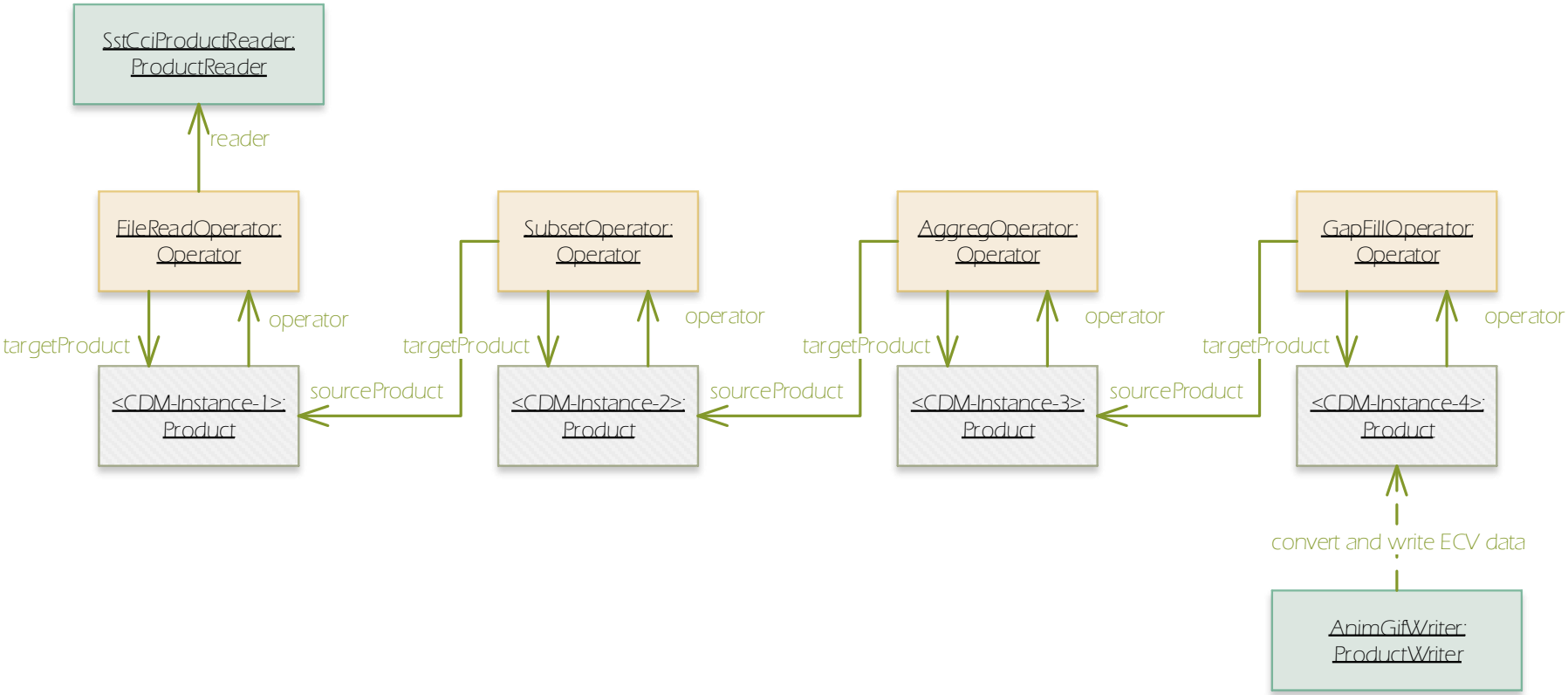
Product Data Model - Raster Data



Product Data Model – Flags and Index Coding



Product Data Model – Geo Coding



Data „Pull“ Processing