KNIME Noding Guidelines

Draft v1.0beta – March 5, 2009

This document describes best practices for the development, documentation and deployment of KNIME nodes, plug-ins and features. Some of the items below are designed to ease usage of new KNIME modules whereas others target interoperability and maintenance. In the following, the term "KNIME modules" refers to nodes, plug-ins or plug-ins grouped into features that form a functional entity.

Content

Installation / License Issues

Plug-in/Node Installation:

- A new KNIME module should be installable via the Eclipse update mechanism.
  - An installed module may require an external setup of the system (e.g. setting LD_LIBRARY_PATH or the presence of an executable/application). However, if these requirements are broken (e.g. path not set), the module should report an error message (e.g. the node fails during execution or the renderer is disabled; it should also set an appropriate error message to the NodeLogger). In no case it should crash or freeze KNIME.
  - The installation of a module must not launch other installers or make modifications to the system other than in the eclipse installation directory.

  - A module must not require any changes to the KNIME launch mechanism, i.e. no changes to the `knime.ini` file or special start-up wrappers are necessary.
- A new KNIME module should be installable by simply copying it into the KNIME installation directory (though this is the recommended path for enabling future updates to this module via the Eclipse update mechanism.)
- An installed KNIME module should continue to work properly when the underlying Eclipse or KNIME version is updated (minor updates).
- Minor updates to the KNIME module should be update-able through the Eclipse update mechanism.
- KNIME modules should not attempt to write into the installation directory during runtime. (Users might not have write permissions, depending upon where KNIME itself is installed on the file system.)
- Installation of a module should not alter the behavior of KNIME at start up (for instance by asking for a license!). If a license is missing, the corresponding node(s) can report this before execution or during/after configuration. (This avoids having to go through several of these dialogs before KNIME starts for the first time .)
- Distribution of KNIME with partner modules ("KNIME Extensions") should always be based on a clean KNIME product as distributed on the KNIME website. There is no need to build own products. It is, of course, preferable to point users to your own update site.
- Vendor nodes should all be grouped within their own (usually top-level) category in the NodeRepository. Moving nodes into existing, internal KNIME categories makes it impossible for users to identify (and find) nodes contributed by a specific vendor.
- **Backward Compatibility:** Nodes whose settings have changed from one version to another (more or different configuration possibilities) should be backward compatible. If possible the new version should provide default values for the new settings fields. These default values should cause the previous behavior of the node, and avoid exceptions if the new value cannot be found in the settings. If it is not possible to make the new version backward compatible with the old version in the way described above, the old version of the node should be moved into an extra plugin (deprecated) and removed from the node repository. The new version should be registered to the node

repository. Using this mechanism, old workflows will load, because the old node can be found in the deprecated plugin, but all new workflows will use the new version, since there are no means to use the old version of the node in the GUI.

KNIME User Experience

Using KNIME should be as easy as possible. Nodes which can guess their settings should do so if there is clear, unique preference for default settings. Dialogs that only have to be opened and closed before the node can be executed (because they guess the settings but force the user to open the dialog) should be avoided. The goal is to allow a fast creation and execution of workflows. Users should not be forced to manually set an option which can be automated. However, at the same time, nodes should not automatically set an option which could lead to confusion or significant waste of computing cycles.
When a node is first dragged onto the workbench, no node settings are set. There are 3 possible configuration scenarios (types of actions required inside a node – some of these may co-occur in a given node!):

1) **Auto-Configuration**

   The node performs a defined operation on a column of a certain type. The configure method is the only place where you can determine if the input table has one or several columns of the compliant type :

   1. **Auto-configure:** if there is only one column of the desired type available – choose it.
   2. **Auto-guessing:** if there are several compliant columns, choose the first one and set a warning in order to inform the user about this selection.

   Example: DecisionTreeLearner (auto-configure/auto-guessing of NominalValue column as class column), CDK Translators (auto-configure/auto-guessing of CDK column).

2) **Standard Settings**

   Learner nodes often have many possible parameters and a do-nothing-configuration is impossible. But usually there are **standard settings**, which lead to reasonable results for most data and/or define a standard behavior for the

underlying algorithm. If standard settings are possible, apply them in order to allow execution of the node with these standard settings without forcing the (possibly unexperienced) user to simply confirm them.
Example: DecisionTreeLearner (use of standard settings like *gain ratio* as the quality measure and *no pruning*)

3) **User Action Required**
For some nodes **no reasonable settings** can be guessed and a configuration where the node has no effect on the data is not possible. In this case, the node should stay red (unconfigured) in order to signal to the user that the settings have to be adjusted. When the dialog opens the default settings are such that they would not have any effect on the data. Thus, the user is forced to adjust the settings in the desired way in order to achieve any effect on the data.

Of course, the above-mentioned scenarios are not exclusive, i.e. a node can fall into several categories, e.g. apply standard settings and perform some auto-configuration/auto-guessing at the same time, as demonstrated in the example of the DecisionTreeLearner.

Node Look&Feel

GUI:

- Each node should have an icon (16px * 16px), which illustrates the functionality of the node. Reusing icons (from KNIME or own icons) should be avoided whenever possible.
- An icon should also be assigned to categories.
- A node should report its progress in the progress bar; provide an explanatory message whenever possible. This is accomplished by ExecutionContext#setProgress(double progress, String message). If the task of the node consists of several subtasks, the ExecutionContext can be used to create subprogress monitors with ExecutionContext#createSubprogress(double d), where the argument in [0,1] defines the fraction of the whole task.
- It must be possible to cancel the node during execution without long delays. The call to ExecutionContext#checkCanceled will throw a CanceledExecutionException which is

handled by KNIME.

- Every node should have comprehensible and extensive documentation for the `NodeDescriptionView`, which has to be located in the `*NodeFactory.xml` file, and should describe what the node does, which options are available in the dialog and – if the node has view(s) – what is displayed in the view(s).
- Port sorting: If a node has ports of different types, then they should be sorted in the following way: For out ports the data ports should be at the top and other port types towards the bottom. For in ports this is vice versa: data ports towards the bottom and other port types towards the top. (This avoids crossing connections between learner and predictor nodes.)

Data Handling:

- A node should be able to handle empty input tables (with no rows and/or no columns) (even if that means the node merely raises an exception and communicates to the log that it could not execute due to there being no input data on which to operate.)
- A node should be able to handle tables containing missing values.

Columns:

- By default the columns of the input table should be retained in the output table. If a column is generated from other column(s), a node can offer an option to either replace or retain the used columns.
- Columns whose types are not handled by the node should be ignored, but passed through the node and not removed.

Logging:

- Do not use `System.out` or `System.err`, but rather KNIME built-in logging mechanism by using the `org.knime.core.node.NodeLogger`.

Node Development – Best Practices

Data Handling:

- A node should generally avoid the use of temporary files – but if unavoidable it must take care to close and delete them.
- Temporary files shall be created in the directory represented by

`System.getProperty("java.io.tmpdir")`. (This variable can be changed in the KNIME preferences.) Use the utility methods `java.io.File#createTempFile()` and `org.knime.core.util.FileUtil#createTempDir()`, which creates files or directories at the correct location. Note, this directory should also be used by external tools / libraries, which are used inside the node implementation.

- Proper handling of KNIME DataTables is supported by the use of `BufferedDataTable`, `BufferedDataContainer`, and `ColumnRearranger`. These tools take advantage of intelligent memory caching functionality inherent in KNIME and are optimally designed for the efficient manipulation and creation of DataTables.
- When columns are added, changed or removed, do not copy the table, but use the `ColumnRearranger` instead.
- File Reader Handling: Nodes reading input files (e.g. File Reader, SD Reader, Table Reader) must not verify the existence of the File during the load of the settings. They should check if the URL has a valid syntax but not test its existence. This is to be done during the `configure()` and, possibly, in the `saveSettings()` method of the dialog. (Reason being that settings may be loaded into a workflow, whereby the file isn't present on the hard disk – in this case the node should successfully load all settings but fail during configure()).
- Do not store any tables as member variables. If a view displays the incoming data, copy reasonable portions of it into a `BufferedDataContainer`, let the user configure how many rows should be displayed but use a reasonable small default value (2500 is recommended) and warn the user when rows are skipped. (Displaying millions of data points results in visual clutter and does not support understanding of the data.)
- A node implementation must not cast the elements of a data row to specific cell implementations. Instead it should always refer to the corresponding DataValue interface (as there may be more than only one data cell implementation). The following code is error-prone:

```
int x = .. // index of column containing DoubleValues
```

```
for (DataRow r : inData[0]) {
  DataCell cell = r.getCell(x);
  if (!cell.isMissing()) {
    // WILL FAIL FOR OTHER DOUBLE COMPATIBLE CELL IMPLEMENTATIONS
    double d = ((DoubleCell)r.getCell(x).getDoubleValue());
    // do something with d
  }
}
```

A correct implementation would be:

```
double d = ((DoubleValue)r.getCell(x).getDoubleValue());
```

Row ID:

- If the incoming table is updated the row IDs should be preserved in the output table whenever possible.
- New row IDs should only be generated if new tables are generated. It is recommended to use the pattern "RowXX" for the row IDs (counts start at "0"!). Use the static method `RowKey.createRowKey(int)` to ensure compliance.

Configure:

- The configure method should check for columns the node can work with. If there is no such column, throw an NotConfigurableException with a detailed message.
- In the configure method it should be checked – if necessary – if the incoming table spec fits the user settings or if it contains columns the node can work with.
- Provide output table spec as completely as possible, i.e. if possible also provide domain information such as possible values for nominal columns and lower and upper bound for numerical columns.
- See also the section on KNIME user experience, above.

Execute:

- All time-consuming tasks and calculations should happen in the execute method.
- Poll the cancel status regularly, by calling the

`ExecutionContext#checkCanceled()` method.

- Incoming data tables can be arbitrarily large (do not keep them in memory, avoid unnecessary iterations)
- Before accessing the value of a DataCell, check if it is a missing value.
- Report progress by using `ExecutionContext#setProgress`.

Dialog Look&Feel

Comprehensive Layout:

- The dialog should be comprehensive and should not change layout when different input tables are available. Grey out options which are not applicable instead. (Do not surprise the user with unexpected changes.)
- Each input field should provide a label which (very) briefly explains the input.
- Resizing of the component should not destroy the layout of the dialog.

Documentation:

- Each option in the dialog should be explained in the NodeDescription using the `<option>` tag in the `*NodeFactory.xml` file.

Modal Sub-dialogs:

- Sub-dialogs should be used as rarely as possible. Input fields can either be grouped inside one panel or by using additional tabs (especially useful for expert settings). If sub-dialogs are unavoidable they must be modal, i.e. they open and remain in front of the parent dialog and disable the parent dialog. For example:

```
// figure out the parent to be able to make the dialog modal
Frame f = null;
Container c = getPanel().getParent();
while (c != null) {
  if (c instanceof Frame) {
       f = (Frame)c;
       break;
```

```
    }
    c = c.getParent();
 }
 // pop open the advanced settings dialog with our current
//settings
 JDialog dialog = new DerivedJDialog(f);
 dialog.setModal(true);
 dialog.setVisible(true);
```

Loading and Saving:

- Loading and saving must work correctly, i.e. the settings are stored and restored on re-opening the dialog and if the workflow was saved and re-opened.
- Do not rely on the settings creation of your NodeModel (e.g. it is possible to store node settings in the dialog, change them in a text editor and load them again)
- Throw a NotConfigurableException after loading all settings if the user cannot make a valid choice, e.g. if no column of correct type is in the passed DataTableSpec. Provide a comprehensible message for the user. See also the section on KNIME User Experience.

View Look&Feel

- The view must resize properly.
- The KNIME built-in view properties *color*, *size* and *shape* should be supported where possible.

Open View & Execute:

- It should be possible to open a view before execution of the underlying node without any exception. (KNIME makes sure nothing is displayed.)
- It should be possible to execute the node while the view is opened. (After execution is done, the view will be notified.)
- All time-consuming aspects of a view should be created during the execution of the node to ensure that the view opens instantaneously.

Open View & Reset:

- It should be possible to reset the node while the view is opened.

- The view must then clean up all its displayed content.

Several View Instances Per Node:

- Several view instances for one node should be possible and independent of each other.
- Selection only happens locally in one view instance.

Loading and Saving:

- View content must be restored when an executed workflow is re-opened.
- *Hilite* status must be reset for a newly opened workflow.

HiLiting

Listening:

- Views should support *hiliting* as a listener when appropriate.
- On opening a view, the current *hilite* state should be displayed correctly.
- If a *hilite* event is received for a row, it should immediately be visible in the view. The standard KNIME visual variables for *hilite* should be used (`ColorAttr.HILITE`).
- If an *unhilite* event is received for that row, it should be displayed like the other *unhilited* data points (e.g. normal, faded or hidden).
- A clear *hilite* event should result in all displayed rows being displayed as *unhilited*.
- In order to detect the hilite status of a data point in the paint method, do not ask the HiliteHandler each time, but store the hilite status in the model.
- Aggregating nodes should use hilite translators to forward hilite events for their output data to the input data table(s).

Handling:

- A view should support – where possible – the triggering of *hilite* for selected rows. Standard KNIME visual variables should be used for displaying selected, *hilited* and selected&*hilited*.
- A *hilite* menu should be provided and obey the naming conventions:
  - *hilite* selected,
  - *unhilited* selected, and
  - clear *hilite*
- The use of predefined names defined by `HiLiteHandler.HILITE_SELECTED` etc. is

highly recommended.
- Also recommended is to provide a menu in the menu bar and a context menu.
- Hilite events triggered in a view should only be sent to the handler and not be executed in the view until the event comes back through the listener interface (that means the view (if it is a listener) will also receive hilite events it triggered itself).

View Development – Best Practices

- If the view listens to `HiLiteHandler`(s) from input ports, the listener has to be deregistered from the old `HiLiteHandler` and registered with the new `HiLiteHandler` in the `modelChanged` method. Be aware, that the HiliteHandler can be `null`.

Type Look&Feel

- If new types are introduced, they should have an icon.

Type Development – Best Practices

- New types also should provide a comparator. If no comparator is provided, the standard comparator based on the string representation will be used. Therefore sorting will always be possible – but if no other/better choice is available, it will sort string representations.
- In addition a specialized serializer is recommended to ensure good performance. If possible do not rely on Java built-in serialization, since it is very slow and may break if the serialized classes change. Even for simple cells (holding an integer or an enumeration), this is recommended.
- Creation of a renderer is recommended for new data types. Note that if the renderer paints an image, then it is highly recommended to extend the `AbstractPainterDataValueRenderer`, if it displays some kind of textual representation then the `DefaultDataValueRenderer` should be used.
- If the new type is claimed to be compatible to other types a lossless conversion between those types must be possible (for example every integer can be converted to a double

without any loss – but not vice versa). Although almost every value can be *displayed* as a String, it is not a String, thus it should not implement the StringValue interface.

Appendix: Chemical Conventions

This sections summarizes the conventions regarding chemical types to ensure interoperability of different KNIME software partners.

Supported Types

- The standard molecular formats are:
  - SMILES
  - SDF
    - Structure block (Mol) can be extracted using standard KNIME node
    - SDF cells terminated by $$$$ characters (part of cell)
    - KNIME's can also return an SD – it will silently append the $-signs
- The standard reaction format:
  - RXN
- Biological formats are available in org.knime.bio.types, currently only PDB. (No standards were discussed as of now – please contact KNIME if that is requried.)

- Types in KNIME are simple textual wrappers (no interpretation!) A simple test should be able to read any of the above formats, write it out, and the result should be an exact textual match!

- Every node working on molecular representations should
  - Accept and create if possible the standard formats (e.g. not SMILES if 3D structures are created/expected)
  - If needed: accept and create (preferably) no more than one partner-specific format (Maestro, ...) – these will not be part of the knime.chem base classes (except for cases where they are of general interest; see below)
- Types of possibly general interest should be part of knime types plugin
- If an extension is able to render any of the standard types (e.g. SD), it should add the appropriate renderer class to, e.g. SdfValue.UTILITY.addRenderer(...)
- Each renderer implementation should make any attempt to not block or crash

KNIME if it is unable to render an entry (because of a missing license, e.g.). Instead it should indicate the problem in the drawing area and issue an error to the logging facilities (once!)

- If you are supporting other types, make sure to use the ones available in org.knime.chem.types or inform KNIME to include it if there is a remote chance that others may also use this type.
  - the currently (as of 05 March 2009) available types in org.knime.chem.types are:
    - SdfCell
    - SmilesCell
    - CMLCell
    - CtabCell
    - Mol2Cell
    - MolCell
    - SlnCell
    - RxnCell (to be included in v2.1)

    Please check org.knime.chem.types for an up-to-date list of currently supported types and contact the KNIME team before adding own, new types.
- Type Conversion nodes should be provided if new types are introduced
  - A "Molecule to X" converter, taking ALL standard types as an input,
  - A "X to Molecule" converter, producing ALL standard types on the output.

  (The user will always know the specific type on either the in- or output by looking at the node label.)
- All nodes should provide clear guidance about the expected and generated types
  - Tooltips on output, e.g. "table with additional X type column"
  - Clear error if compatible type is missing, e.g. "expecting SDF, Maestro, or SLN column"

  (users need to know what they need to do if they can't connect two nodes.)
  - Convention:
    - "Molecule" stands for any one of the standard molecule types.
    - Individual types are: SDF, Smiles, Mol2, ChemML, or "X" (where X is descriptive and sufficiently unique).

Handling of multiple representations – sets/lists of cells

- In case of multiple representations of a molecule (e.g. node computes conformations):
  - If at all possible create one row for each conformation and add information about original row ID (similar to a pivoting node)
  - Alternatively use "List of DataCells" (available in KNIME 2.0) to add several representations of the same type.
    - KNIME allows composing/decomposing such sets/lists

Reading/Writing

- Readers
  - Validate format
  - Offer the ability to extract additional information
  - Leave textual information intact!
- Writers
  - Offer to combine additional columns into the type#(e.g. SD cell alone or Mol-Cell with add'l cols)
  - Otherwise leave textual information intact!
- Nodes computing additional information for molecular types (3D conformation…)
  - Create a new column holding cells of the same type with this additional information "inserted".

Fingerprints

- Use Fingerprint cell implementations available in KNIME
  - DenseBitVector, SparseBitVector for {0, 1}
  - DenseByteVector, SparseByteVector for counts (0-255)
- Different operations available on fingerprints (see package description), e.g.
  - concatenate, and, or, xor, mask

Testing / Certification

- Partners should whenever possible submit test workflows for their own nodes:
  - KNIME may not have suitable test structures (format, specific properties)
  - Some testing will be done by non-chemists.
- Partners should submit test cases
  - Allows to test interoperability of partner tools