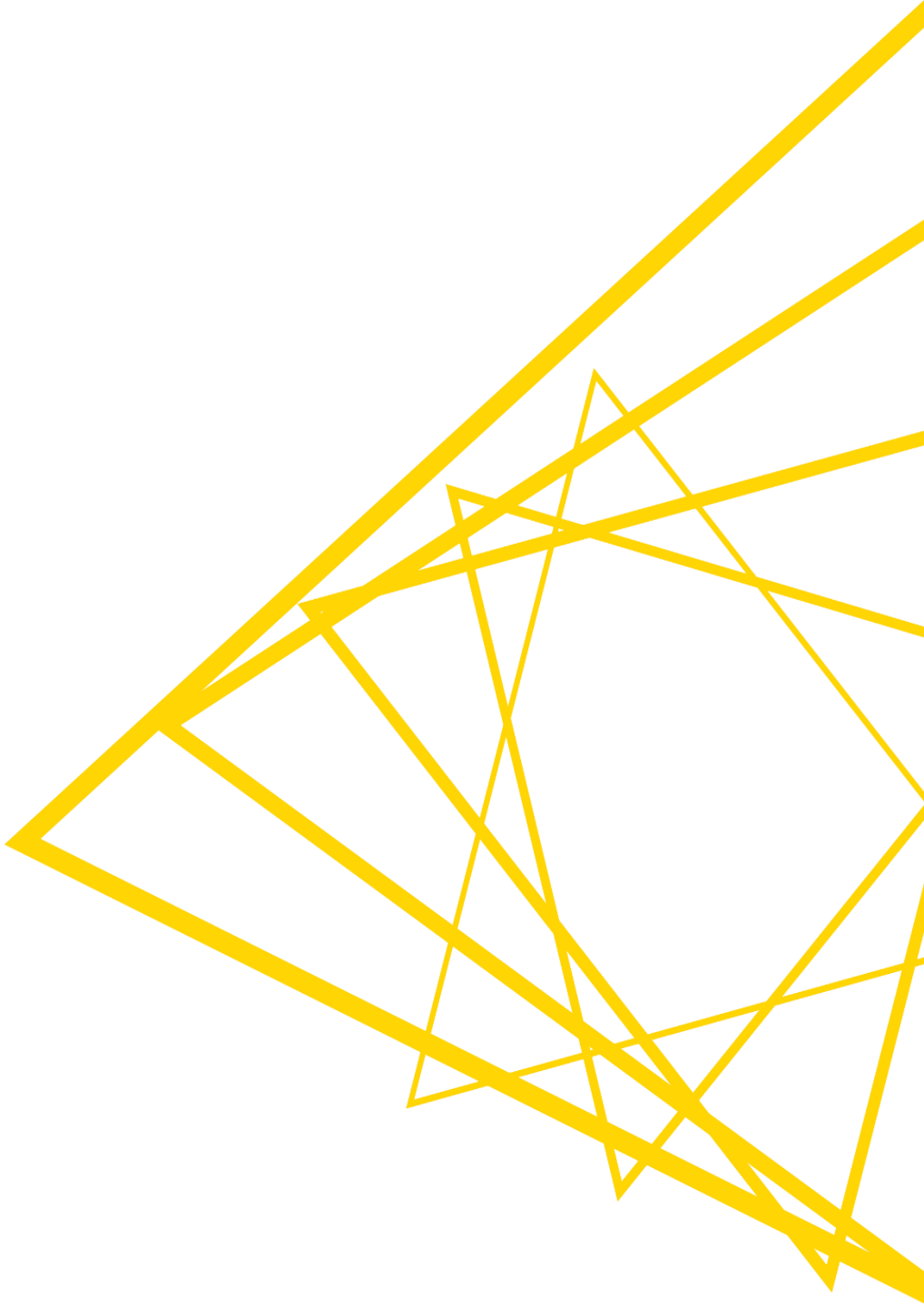
Developer Training



Developer Training

- Goal: enable you to develop your own new node for the KNIME Analytics Platform
- Structure of training sessions:
  - Presentation
  - Demo
  - Exercise

KNIME SDK Installation

- Setup Eclipse
  - Follow these instructions
    - https://bitbucket.org/KNIME/knime-sdk-setup/src/master/

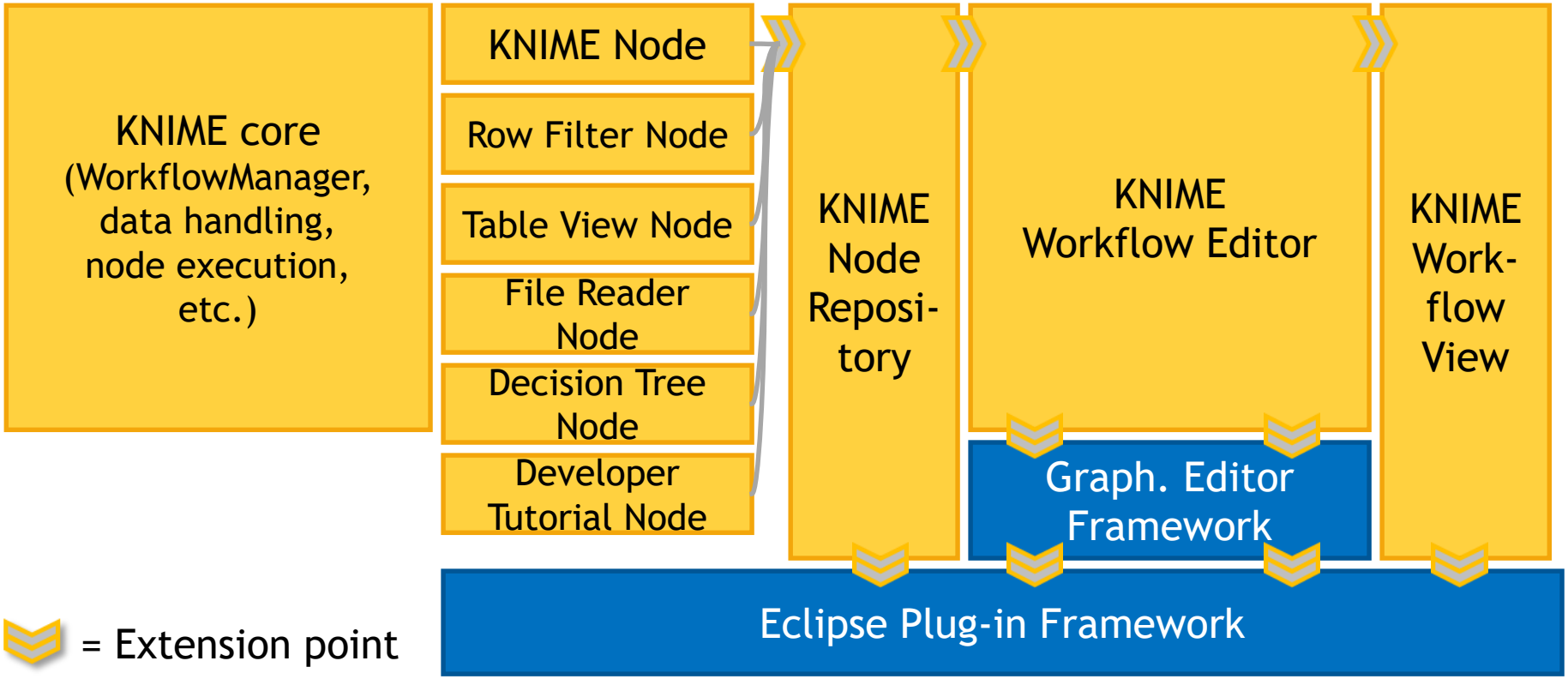
- Import workspace (exercises and solutions)
  - `/workspace/knime-workshop-workspace.zip`

Architecture Overview

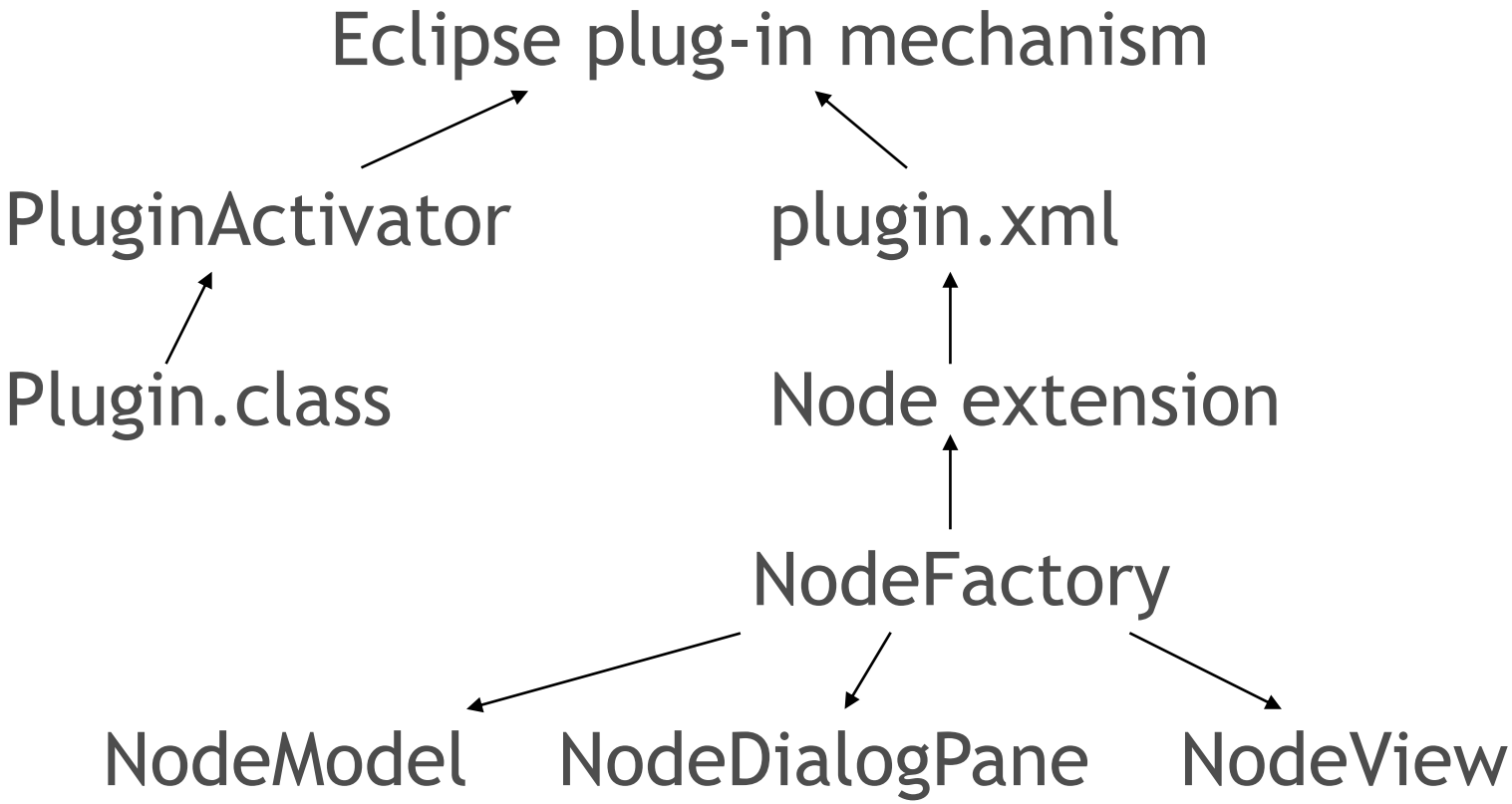
KNIME / Eclipse Application

- The KNIME Analytics Platform is based on Eclipse
  - Eclipse modular framework starts installed plug-ins
  - KNIME is implemented as plug-ins (core, workflow editor, node repository, etc.)
  - Offers „Extension Points“ (category, nodes)
    - Easy to extend and add new nodes
  - Eclipse provides infrastructure: views, editors, update manager, preferences, ...

KNIME in Eclipse



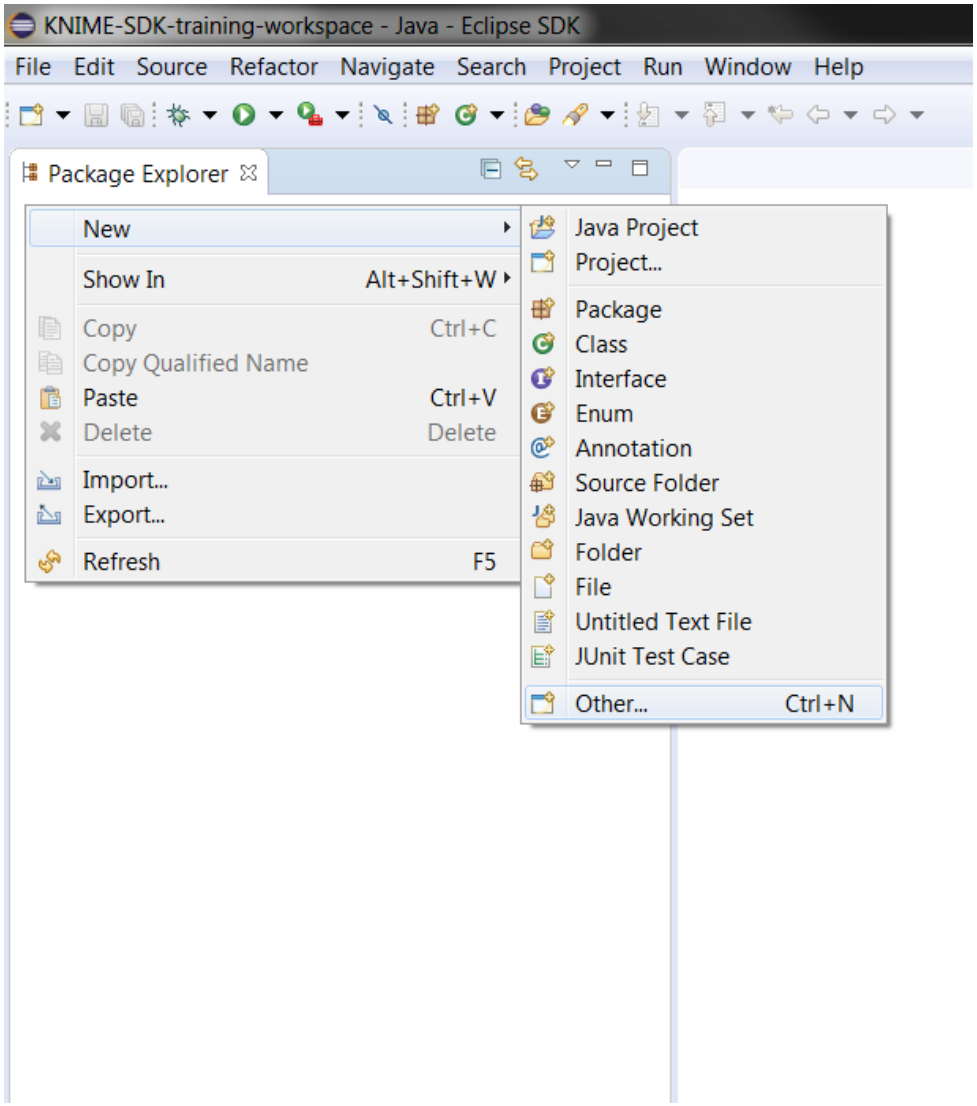
Node as a Plug-in

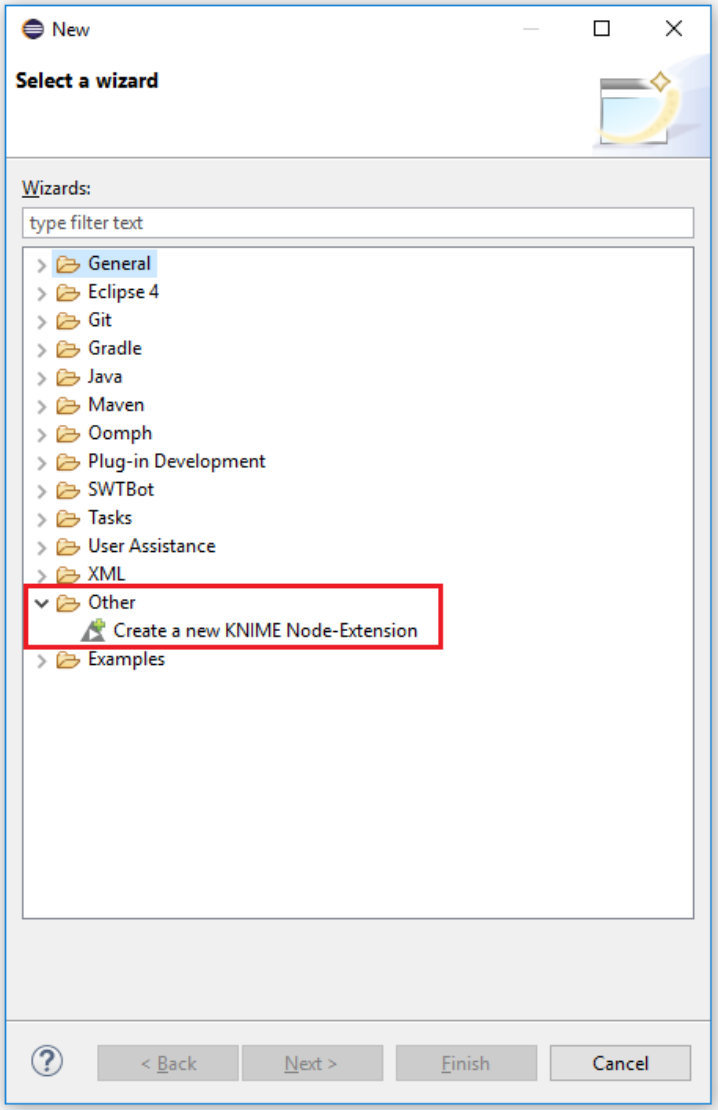


Node Extension Wizard

- Install in eclipse
  - Help -> Install New Software…
  - Work with: http://update.knime.com/analytics-platform/3.6
  - KNIME Node Development Tools -> KNIME Node Wizard
  - Press next and follow instructions
- Allows creation of plugin projects including functioning KNIME nodes (with sample code)
- Helpful for easily creating all node classes
  - Generates all Java classes
  - Node is registered with the plugin project
  - Launch KNIME and enjoy the new node working!

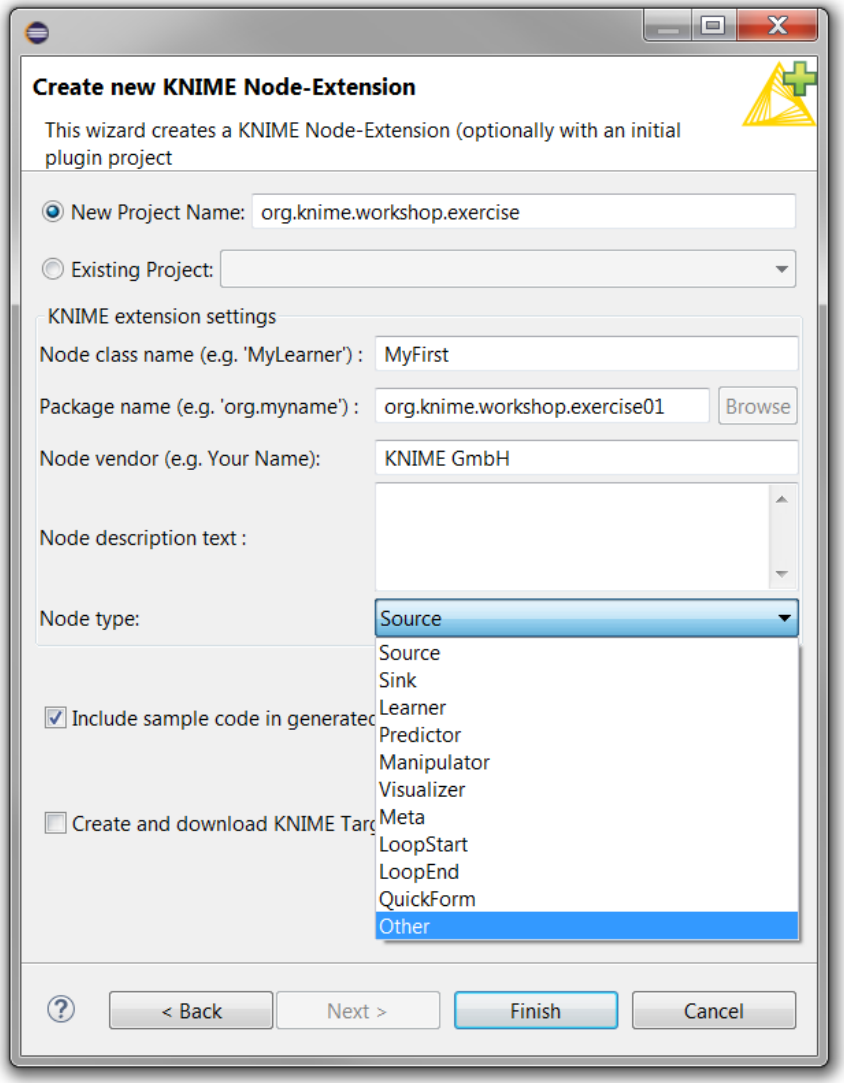
Create a new KNIME Node-Extension I





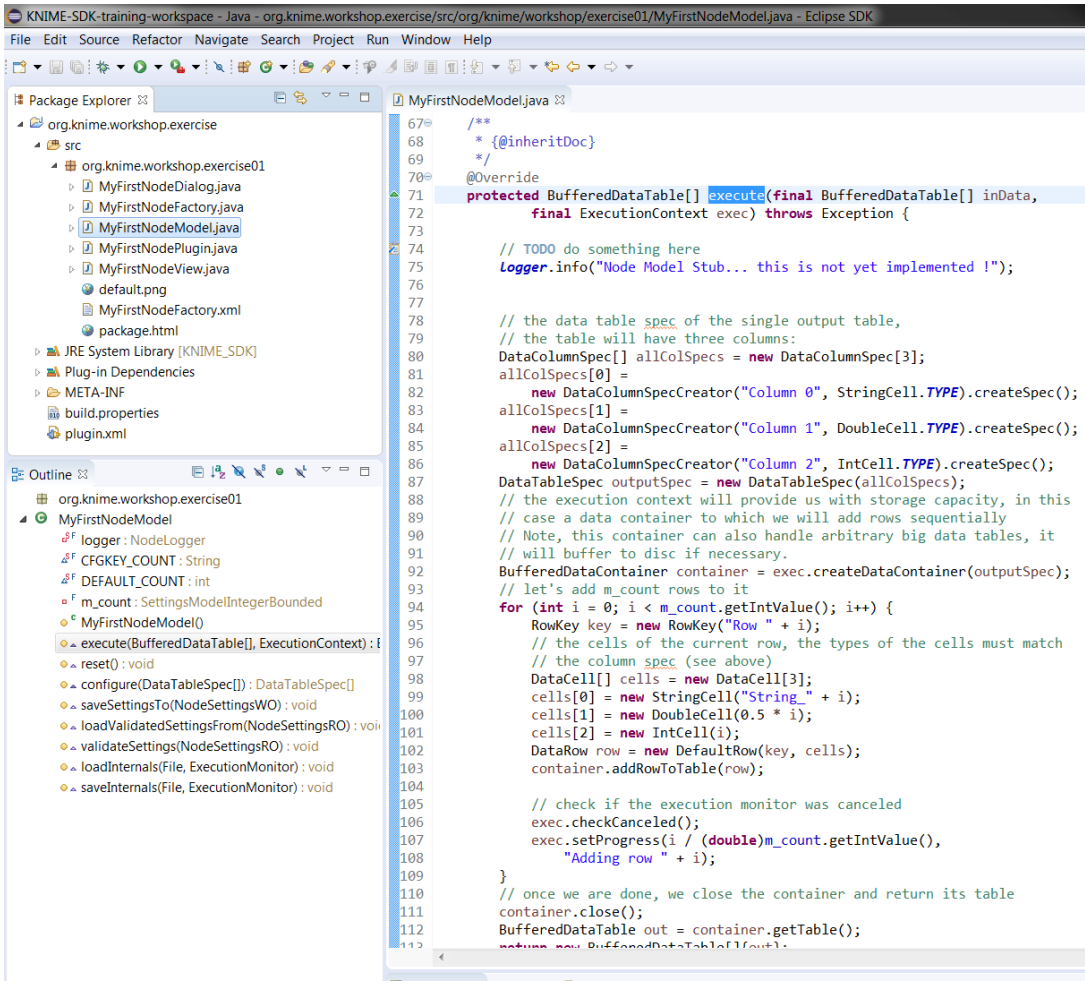
Create a new KNIME Node-Extension II

- Specify all settings to create a new KNIME node
  - In a completely new plugin project, or
  - Into an existing project
- Node type: Source, Sink, Learner, Predictor, Manipulator, Visualizer, Meta, LoopStart, LoopEnd, QuickForm or Other
- Include sample code or do not



Create a new KNIME Node-Extension III

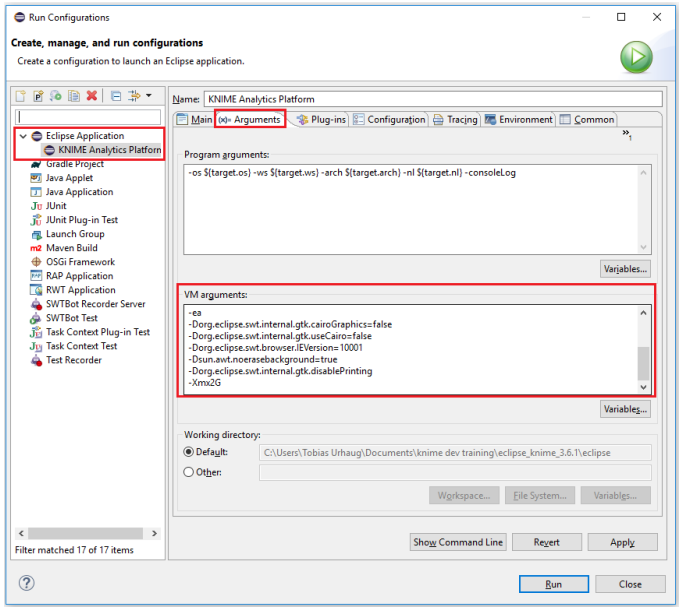
- Contains all Java classes (including sample code)
- Node is registered in the **`plugin.xml`**
- **`NodeDialog`** and **`NodeView`** classes are also created and registered to the **`NodeFactory`**



Launch KNIME

Edit `Run Configuration`

- Menu „Run“ and select „Open Run Dialog...“
- Find „KNIME Analytics Platform“ configuration under „Eclipse Application“
- Find VM arguments under tab “Arguments”
  - Xmx2G -> Adjust



Demo #1

- Create a new KNIME Node-Extension **`org.knime.ws.exercise`**
- Create MyFirstNode in a package **`org.knime.ws.exercise01`** (include sample code)
- Create a new `Run Configuration` and launch KNIME Analytics Platform

Exercise #1 – Creation of KNIME Node-extension

- Create a new KNIME Node-Extension
  `org.knime.ws.exercise`
- Create your first node in a package
  `org.knime.ws.exercise01` (include sample code)
- Create a new `Run Configuration` and launch KNIME Analytics Platform
- Use your node, execute it and visualize the output data

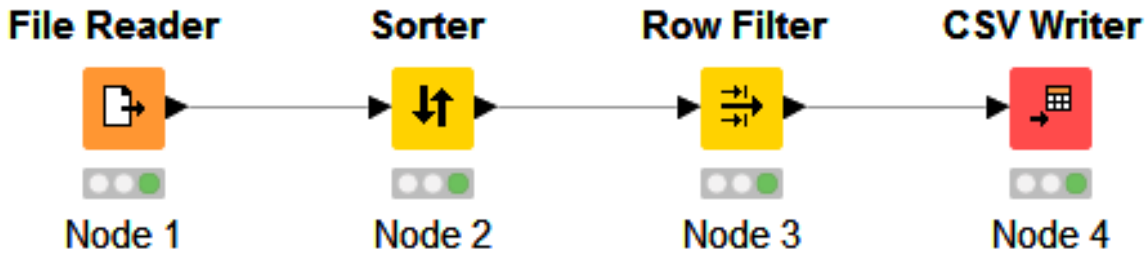
- *Optional*: change the node name and type, unregister **`NodeView`** and **`NodeDialogPane`** in the `NodeFactory.java|.xml`

Node Architecture

Overview

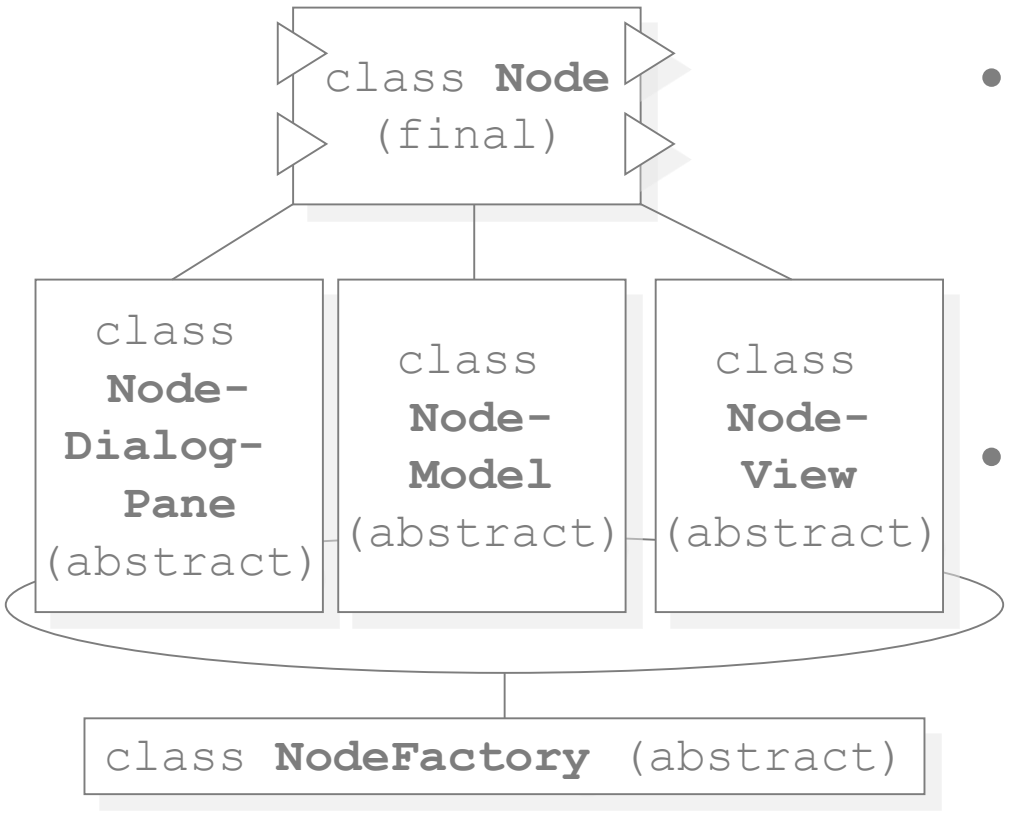
- Node
  - Classes to be created
  - Methods to be implemented
- Data Table
  - Structure of a **`DataTable`**
  - Access to input data
  - Create output **`DataTable`**

KNIME Nodes



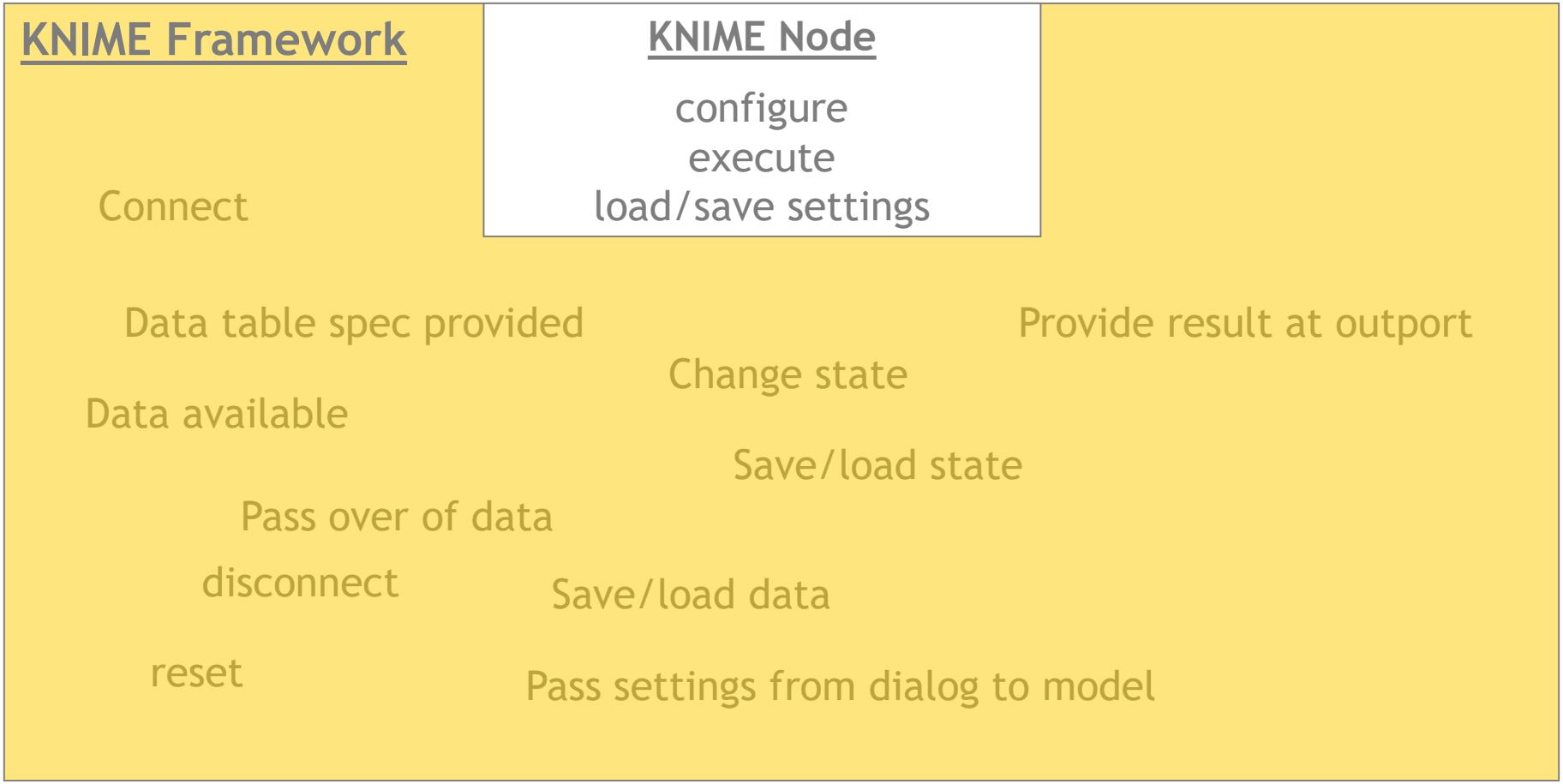
- Nodes: encapsulate functionality of one processing unit
  - Data generation, manipulation, transformation, visualization, ...
  - Embedded in a flow
  - May have a dialog (settings) and/or view
- Ports: connection points to transfer data between nodes
  - Store the result of the node‘s execution
- Connections: pipeline between nodes
  - Transport data in DataTable objects

Node Architecture



- KNIME AP interacts only with a **Node**
- **Node**-class takes care of embedding the node in the infrastructure
- New nodes implement Model/View/Dialog

Framework vs. Node



Node Components

- **`NodeFactory`**:
  - Bundles: **`NodeModel`** / **`NodeView`** / **`NodeDialogPane`**
  - Gets registered with KNIME framework (`plugin.xml`)
- **`NodeDialogPane`** (optional):
  - Panel with GUI components displayed in an SWT dialog
  - Contains components for all user-adjustable settings
- **`NodeView`** (optional):
  - Panel with GUI components displayed as **`JFrame`**
  - Provides view on the **`NodeModel`** result (Decision Tree, …) or just a view on the data (Table, Scatterplot, Boxplot, …)

Node Components: NodeModel

- Implements the task of the node
- Defines number of in- and outputs for data and/or models
- Handles loading and saving of dialog settings
- Implements **`#configure()`**
  - Specifies the structure of output table
  - Determines “executable” state
  - Computes output **`DataTableSpec`** objects
- Implements **`#execute()`**
  - Computes output **`DataTable`** objects where possible

`NodeModel` Implementation I

- **`#constructor`**: init model with number of data ports or with PortType arrays
  - `super(#dataIns, #dataOuts);  // data ports only`
  - `super(PortType[] inPorts, PortType[] outPorts);`

- **`#configure()`:** quickly check settings against incoming data table spec(s) and generate output data table spec(s)
  - if settings are consistent with input data table spec(s)
    `return new DataTableSpec[]{outSpec1,…,outSpecN};`
  - if consistent, but the structure of output data table(s) is still unknown
    ```
    return new DataTableSpec[#dataOuts]; // array with null
                                         // elements. Fill slots
                                         // for those columns that
                                         // are available
    ```
  - if not consistent, throw exception
    `throw new InvalidSettingsException("errorMessage");`

`NodeModel` Implementation II

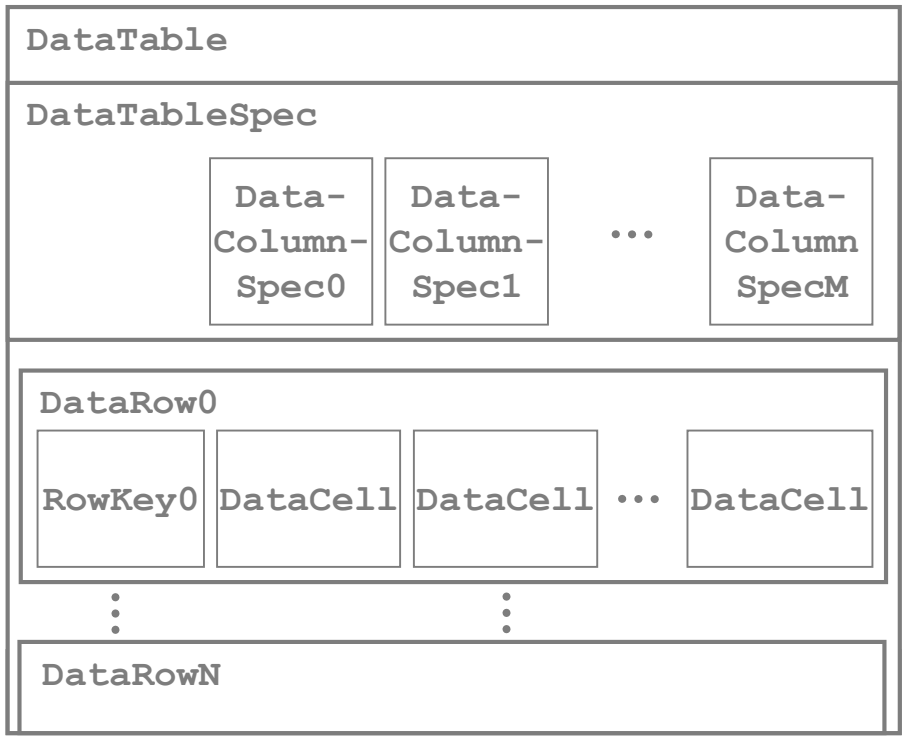
- **`#execute`**: called after configure to process the input data
  - Manipulate input data or generate new data
  - Report progress
  - Execution has finished successfully

    ```
    return new BufferedDataTable[]{outData1,…,outDataN};
    return new PortObject[]{outData1,…,outDataN};
    ```

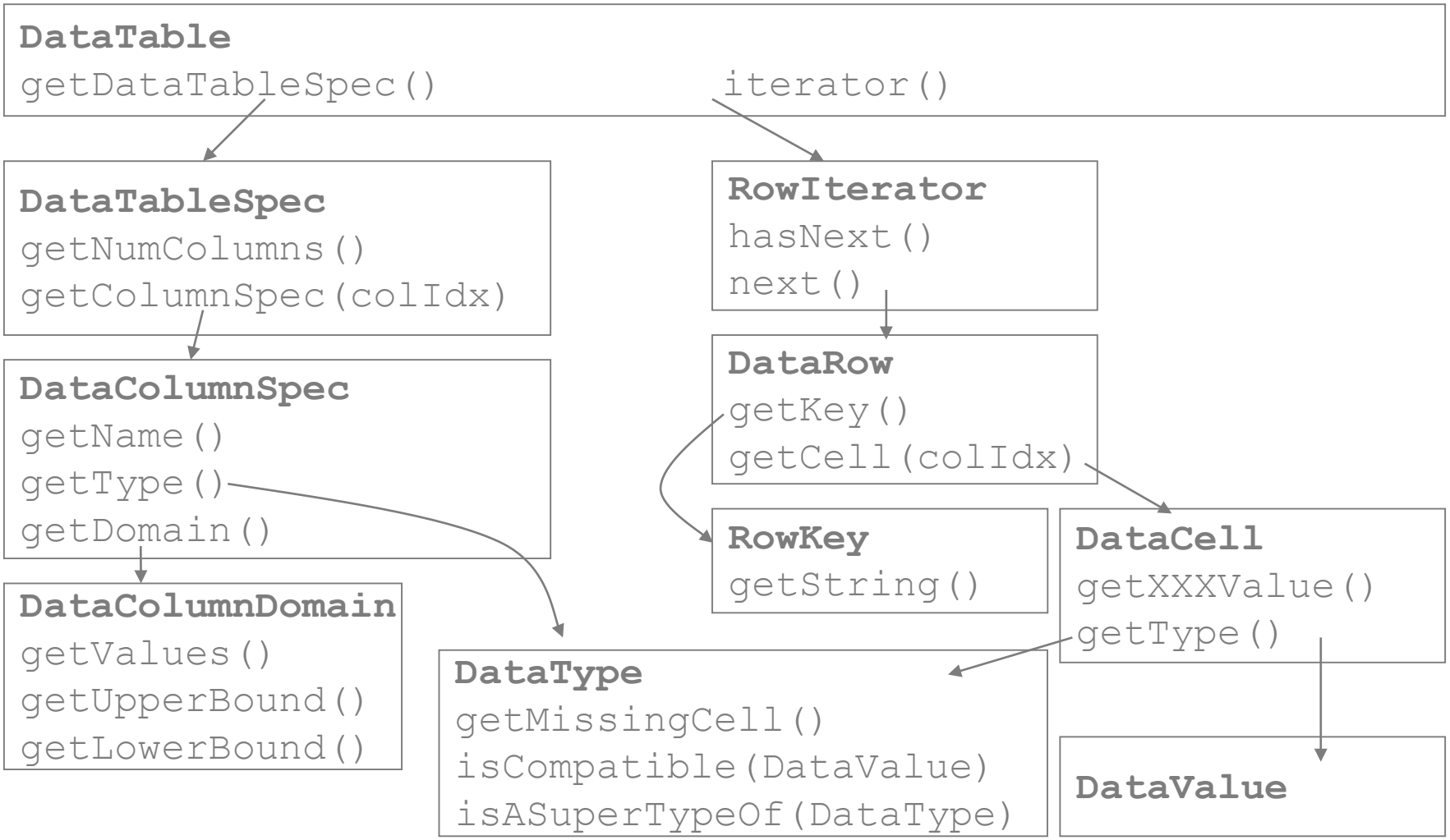
  - Throw exception if something goes wrong
- **`#reset()`**: delete internal data/models generated during execute
  - Do not delete data tables(s) returned as result of **`#execute()`**!
  - Reset HiLiting

DataTable Structure I

- Used to transfer data from one node to all connected successor nodes
- Collection of read-only **`DataRow`** elements (data vectors)
- Fixed number of columns
- Fixed column types (**`String`**, **`Integer`**, **`Double`**, etc.)
- No random access to rows (delivered by an iterator)
- Arbitrary number of rows



DataTable Structure II



Read DataTableSpec and DataTable

- Sample code to access a **DataTableSpec** and a **DataTable**

```
DataTableSpec spec = inData[0].getDataTableSpec();
for (int i = 0; i < spec.getNumColumns(); i++) {
      DataColumnSpec cspec = spec.getColumnSpec(i);
      System.out.println(cspec.getName() + " " + cspec.getType);
}

DataTable data = inData[0];
// iterate over all rows
for (DataRow row : data) {
      // and columns
      for (int i = 0; i < row.getNumCells(); i++) {
            // access data cell
            DataCell cell = row.getCell(i);
            System.out.println(cell.toString());
      }
}
```

Create a `DataTableSpec`

- Sample code to create a **`DataTableSpec`**

```
int nrColumns = ...;
DataColumnSpec[] cspecs = new DataColumnSpec[nrColumns];
// iterate over all column specs
for (int i = 0; i < nrColumns; i++) {
     // each column is created by a column name and type
     specs[i] = new DataColumnSpecCreator("name_" + i,
                    DoubleCell.TYPE).createSpec();
}
DataTableSpec spec = new DataTableSpec(cspecs);
```

Write into a DataContainer

- Sample code to write into a **`DataContainer`** row-by-row:

```
// use previously created spec
DataTableSpec spec = ...;
// create container to add rows
BufferedDataContainer buf = exec.createDataContainer(spec);
DataTable data = inData[0];
for (DataRow row : data) {
      DataCell[] copy = new DataCell[row.getNumCells()];
      for (int i = 0; i < row.getNumCells(); i++) {
            DataCell cell = row.getCell(i);
            copy[i] = cell;
      }
      // add rows to container
      buf.addRowToTable(new DefaultRow(row.getKey(), copy));
}
// close the buffer
buf.close();
return new BufferedDataTable[]{buf.getTable()};
```

Demo #2

- Create a new node within your plugin project in a new package **`org.knime.ws.exercise02`** (do not include sample code)

- Implement **`NodeModel#execute()`** such that the node just passes the **`DataTable`** through to the output port

Exercise #2 – Copy input to output

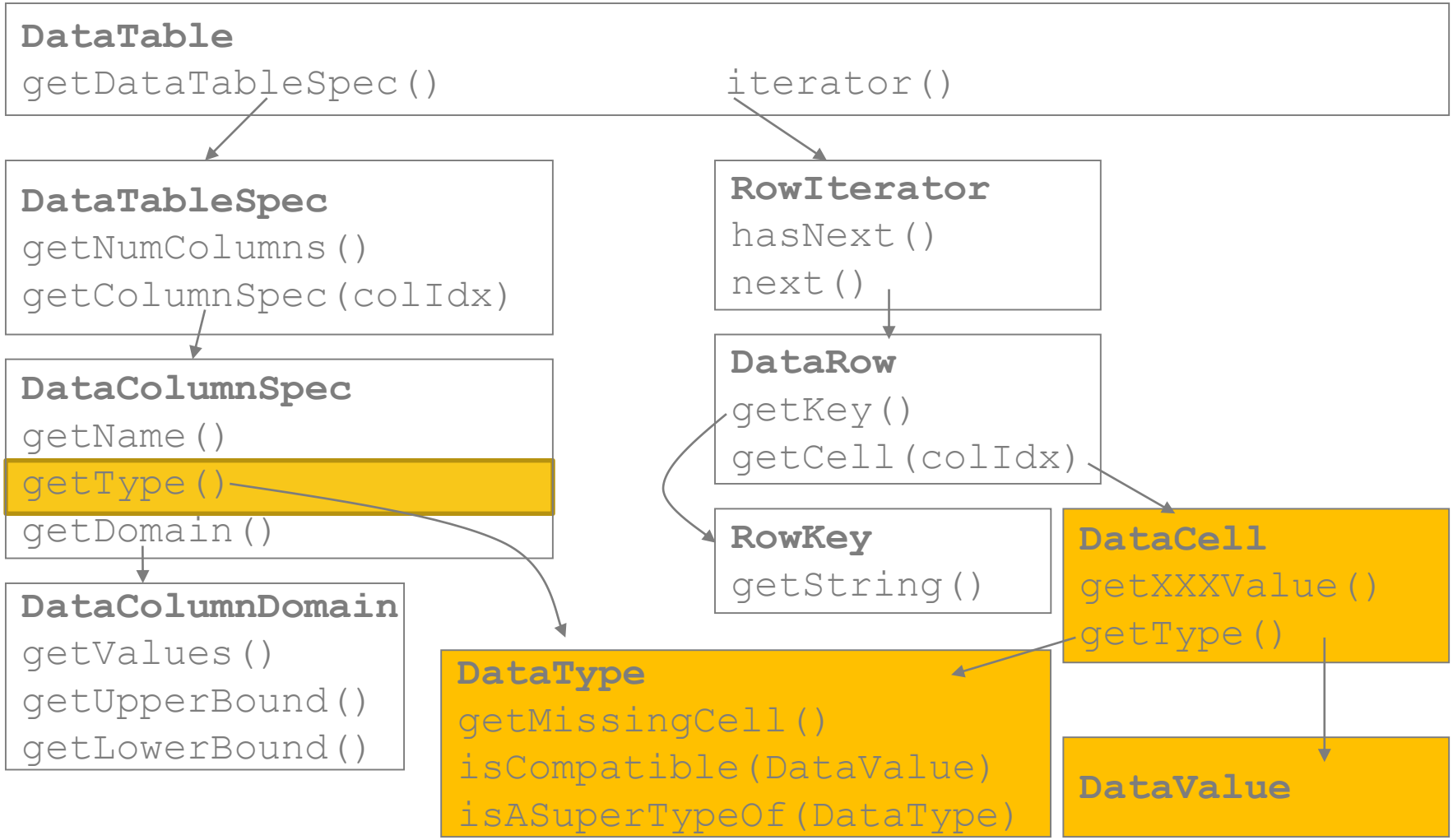
- Create a new node within your existing plugin project in a new package **`org.knime.ws.exercise02`** (do not include sample code)
- Implement **`#configure()`** to copy the input data table spec to the output
- Implement **`#execute()`** to copy the input data table to the output

Data Types

Overview

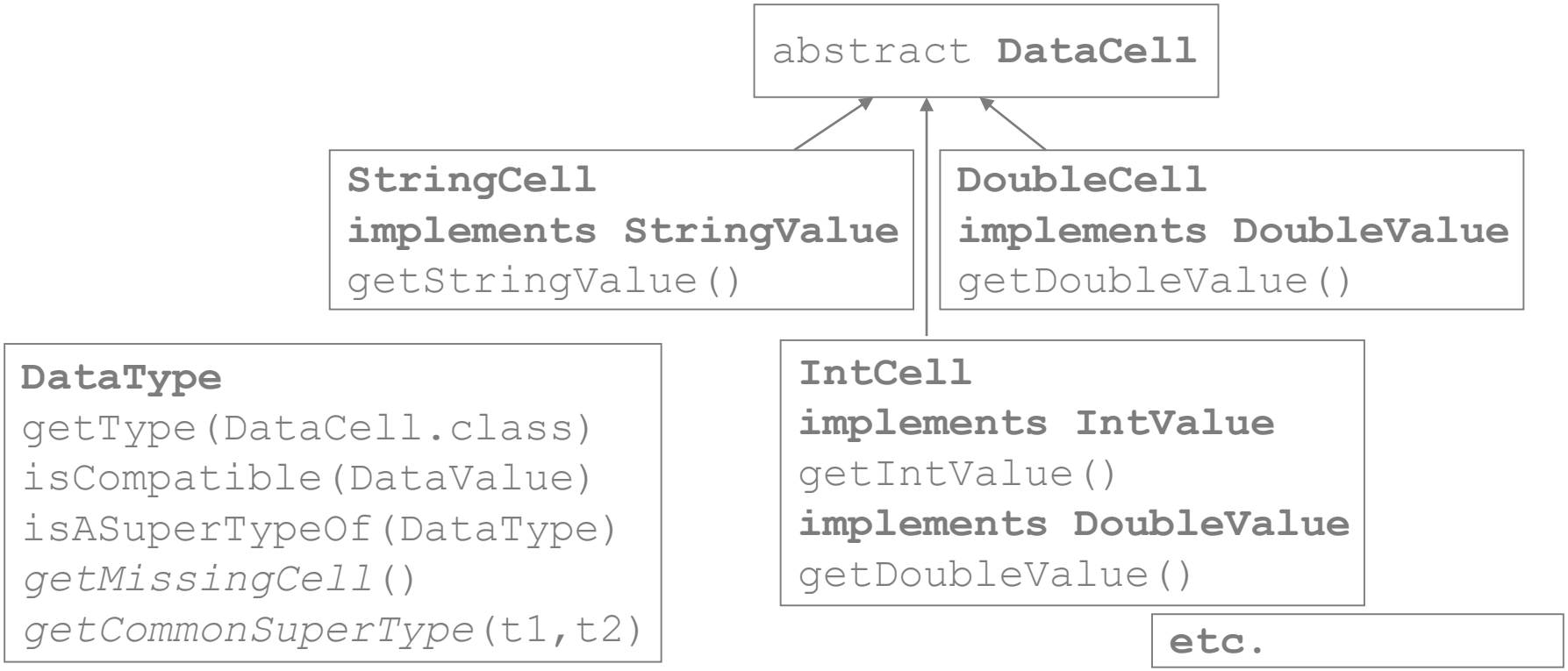
- DataTypes in the KNIME framework
- How to access values
- Defining new **`DataType`**s

DataTable Structure



DataCell, DataType, DataValue

- Each data cell implementation extends abstract **DataCell** and implement one or multiple **DataValue** interfaces
- For each **DataCell** a specific **DataType** is created



Data Types

- Available default cell implementations: **`DoubleCell`**, **`IntCell`**, **`StringCell`**,…
- Chemistry/Bio related types: **`SmilesCell`**, **`SdfCell`**, **`PdbCell`**, … (all just typed Strings), **`CDKCell`**, **`BitVectorCell`**
- Others: **`ImageCell`**, **`DocumentCell`**, **`Object3DCell`**, **`GenericBlobCell`**

DataCell access via DataValue

- Sample code to access double values:

```
BufferedDataTable data = inData[0];
DataTableSpec dataTableSpec = data.getDataTableSpec();
boolean[] doubleCompatible = new boolean[dataTableSpec.getNumColumns()];
for (int i = 0; i < dataTableSpec.getNumColumns(); i++) {
    DataColumnSpec colSpec = dataTableSpec.getColumnSpec(i);
    doubleCompatible[i] = colSpec.getType().isCompatible(DoubleValue.class);
}
// iterate over all rows
for (DataRow row : data) {
    // and columns
    for (int i = 0; i < row.getNumCells(); i++) {
        DataCell cell = row.getCell(i);
        // check compatibility
        if (doubleCompatible[i]) {
            double d = ((DoubleValue) cell).getDoubleValue();
            // do something with the double value
        } else {
            Logger.info(cell + " " + cell.getType());
        }
    }
}
```

Demo #3a

- A node that adds all Integer columns and appends the result in a column ‘Sum’

Exercise #3a – Concatenate string columns

- Implement a node that concatenates all string cells in each row of the input table and puts the value in an appended column named ‘Concatenate’
- Use the skeleton in `org.knime.workshop.exercise.exercise03_a` package and implement the todo’s
- Optional: check for missing values during **`#execute()`**

Data Types – Nice to know

- Columns in **`DataTable`**s are typed, i.e. have a **`DataType`** associated with it
- **`DataType`** class keeps meta information to a **`DataCell`** renderer, comparator, type icon, compatibility list
- **`DataCells`** contained in a column can be type-casted to **`DataValue`** interface (unless missing) according to column spec's compatibility list
- **`DataType`** is generated automatically (at runtime)

- Rule of thumb: Never cast to a specific **`DataCell`** class, always use associated **`DataValue`** interface

Defining new Data Types

- Definition of new type requires appropriate definition of a **`DataValue`** interface and a default **`DataCell`** implementation

- **`DataValue`** interface also defines meta information:
  - Access methods (read-only!!!)
  - Renderer, Icon, Comparator

- **`DataCell`** class implements compatible **`DataValue`** interfaces
  - Possibly defines Serializer for efficient storage
  - May extend BlobDataCell – efficient handling for large cell objects

Defining new `DataTypes`

- Where to get help (when defining some new types):
  - Documentation: https://www.knime.com/docs/api/org/knime/core/data/DataType.html
  - Copy & adopt existing implementations
  - KNIME Forum (https://forum.knime.com/)
  - Developer Guide (https://www.knime.com/developers)

Demo #3b

- JSON DataCell

Exercise #3b – Create new Data Type

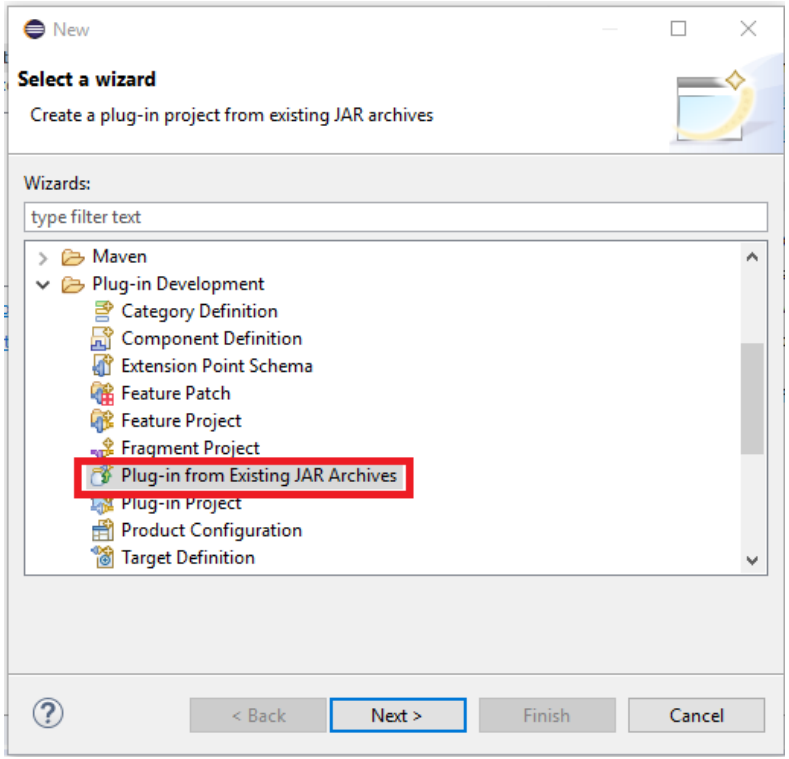
- Create GPS Data Type!
- Finish the implementation in `org.knime.workshop.exercise.exercise03_b`

Using external libraries

- If you need external libraries
  - First check if there is already an Eclipse plug-in, e.g. at Eclipse Marketplace
  - Create a new plug-in containing only this library
    - Preferred way
  - Use library directly in your plug-in but do not export its packages
    - Only for very specific/exotic libraries

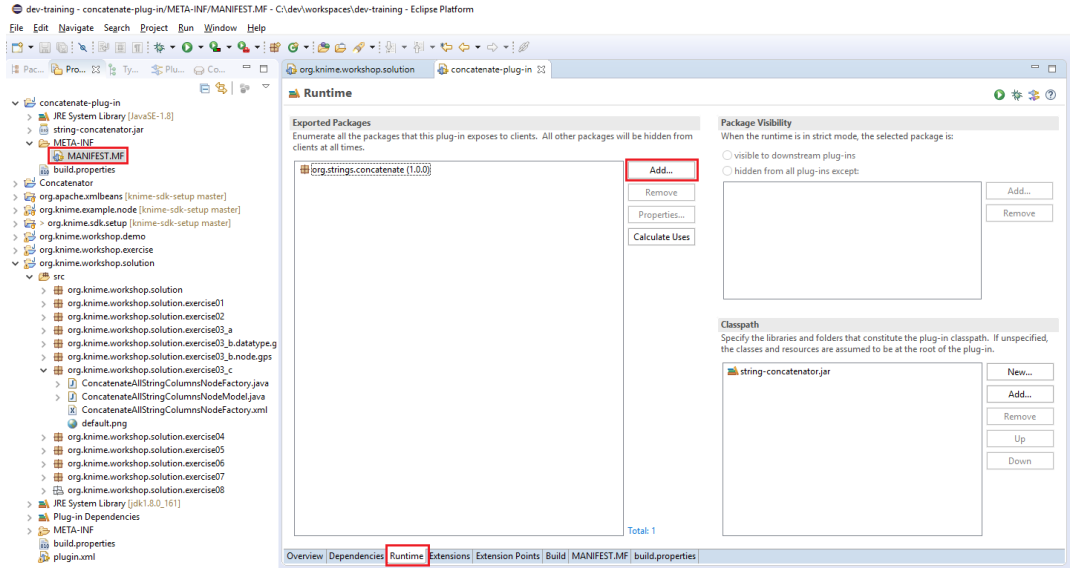
Creating a plug-in project for a jar

- File -> New -> Other…
- Select Plug-in from Existing JAR Archives under Plug-in development
- Add External…
  -> find your jar file
- Fill in project name and plug-in info
- Finish



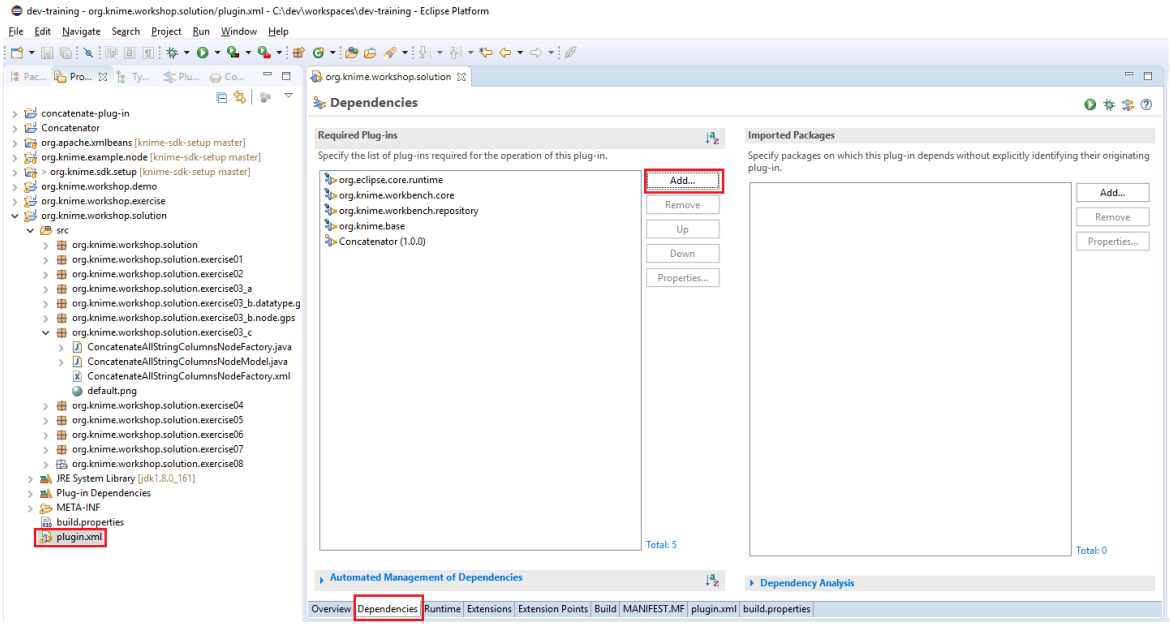
Making a plug-in project visible for other plug-ins

- Open the plug-ins MANIFEST.MF
- On the runtime page, under Exported Packages, click Add… to add the packages of the jar that should be exported to the other plug-ins



Add a dependency to a plug-in

- Open the plugin.xml in the depending plug-in
- Under the tab Dependencies, click Add… and select the plug-in



Exercise #3c – Create a plug-in for an external library

- Create a new plug-in wrapping the string-concatenator.jar (On the USB stick)
- Export the org.strings.concatenate package in the new plug-in's MANIFEST.MF
- Add the new plug-in as a dependency in the org.knime.workshop.exercise plugin.xml
- Use the skeleton in `org.knime.workshop.exercise.exercise03_c` package and implement the todo in the NodeModel

Node Dialog and Dialog Settings

Overview

- User settings: handling and storage
  - **`NodeSettings`**: transports values in the framework
  - **`NodeModel`**: stores and uses user values
  - **`NodeDialog`**: GUI to adjust/enter values

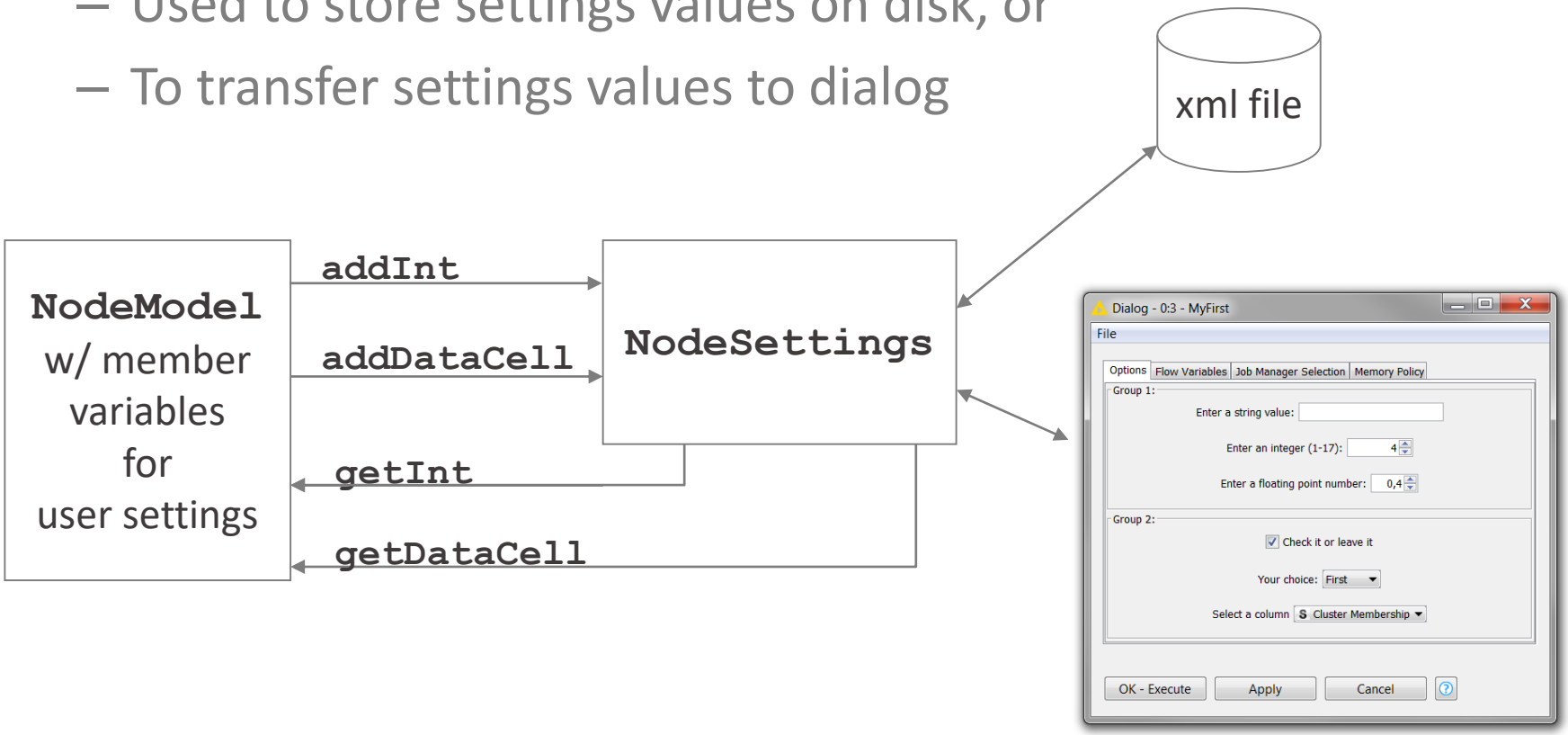
User Settings

- Needed when
  - A node's computation can be adjusted
  - User input/selection is required
- Examples
  - Set the number of clusters
  - Select the class column
  - Set the filter criteria

NodeSettings: read/write

NodeSettings object:

- Transport vehicle of the platform for user values
  - Used to store settings values on disk, or
  - To transfer settings values to dialog



`NodeSettings` object

- Values are stored as <key, value> pairs
  - E.g. **`addInt(„NoOfClusters“, m_clusterCnt)`**
- Supports all standard types (**`boolean`**, **`int`**, **`double`**, etc.) and **`DataCell`**, **`DataType`**, and arrays
- Must use keys that are unique within the settings object
- Hierarchical (**`addNodeSettings`**)
- Retrieve values: e.g. **`getInt(„NoOfClusters“);`**

User Settings: `NodeModel`

`NodeModel` save/validate/load user settings:

- **`saveSettings(NodeSettingsWO)`**
  - Add current values of user settings to NodeSettings object
  - Handle no user settings (after node creation)

User Settings: `NodeModel`

`NodeModel` save/validate/load user settings:

- **`validateSettings(NodeSettingsRO)`**
  - Read values from **`NodeSettings`** object
  - DO NOT modify member variables
  - Check existence of required settings
  - Check consistency of values
  - Throw **`InvalidSettingsException`** if not acceptable
  - Further checking done by configure()

User Settings: `NodeModel`

`NodeModel` save/validate/load user settings:

- **`loadValidatedSettings(NodeSettingsRO)`**
  - Safely assume object passed validate method
  - Read values from **`NodeSettings`** object
  - Store them in member variables

User Settings: `NodeModel`

`NodeModel` save/validate/load user settings:

- **`configure(DataTableSpec[] inspecs)`**
  - Validate user settings against incoming table structure
  - Set default user values that depend on the DataTableSpec (if possible)
- **`reset()`**
  - DO NOT change user settings in reset method

User Settings: `NodeDialogPane`

`NodeDialogPane`:

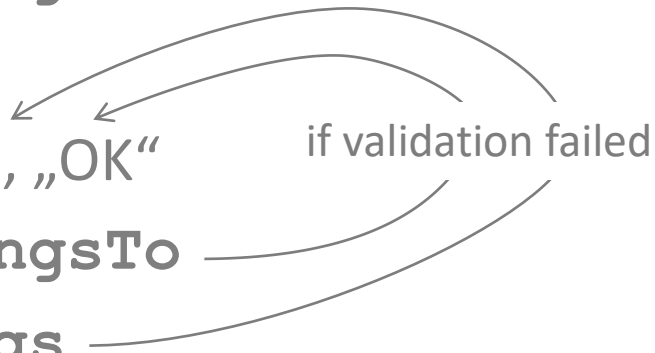
- Contains SWING components
- Implements load/save settings

- **`loadSettingsFrom(NodeSettingsRO,DataTableSpec[])`**
  - Reads values into dialog components
  - Handle empty **`NodeSettings`** by calling **`NodeSettingsRO#getX(key,default)`**
  - Throw **`Exception`** if dialog should not open (only in very rare cases as last option!)

- **`saveSettingsTo(NodeSettingsWO)`**
  - Transfer values from dialog components
  - Do basic validation (e.g. empty fields)

`NodeSettings` flow

- Sequence when dialog opens:
  - **`NodeModel#saveSettingsTo`**
  - **`NodeDialogPane#loadSettingsFrom`**
  - (**`NodeDialog#show`**)
  - User changes values in Components, „OK“
  - **`NodeDialogPane#saveSettingsTo`**
  - **`NodeModel#validateSettings`**
  - **`NodeModel#loadValidatedSettingsFrom`**
  - (…)
  - **`NodeModel#configure`**



Custom Dialog

Full featured **`NodeDialog`**:

- Derive from **`NodeDialogPane`**
- Create, place and layout Swing Components
- Add panels as tabs to dialog
- Implement **`saveSettingsTo`**
- Implement **`loadSettingsFrom`**
- … or use **`DefaultDialogComponents`**

`DefaultComponents` for `NodeDialog`

`DefaultNodeSettingsPane`

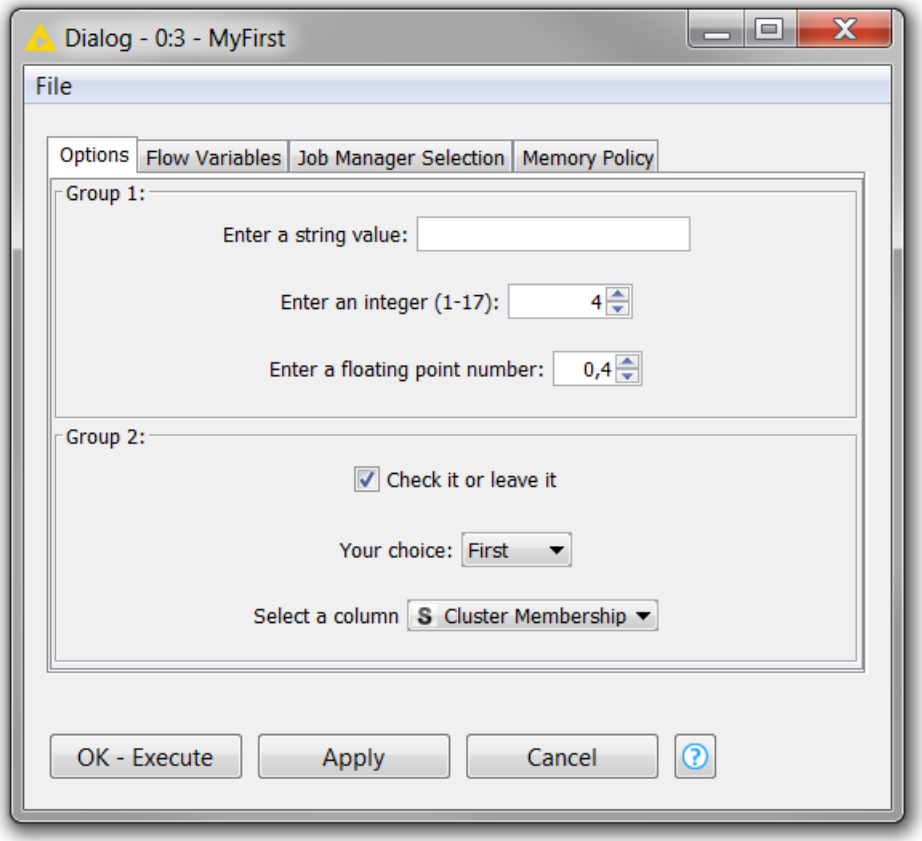
- Easy creation of simple dialog
- Supports parameters of standard types
- Components are placed one below the other
- Handles save/load

`SettingsModel` and `DialogComponent`

- **`SettingsModel`** associated with component
- **`SettingsModel`** holds parameter value
- Same kind of model used to keep value in **`NodeModel`**
- Simply create components and add them to the pane

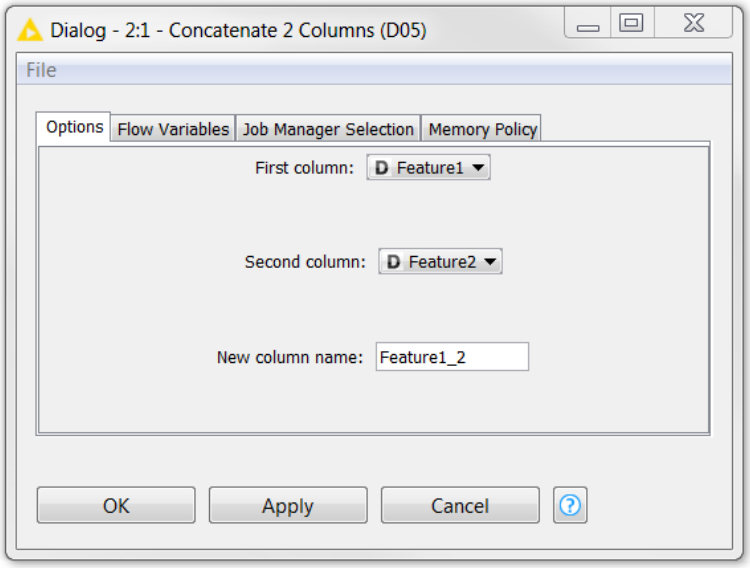
DefaultNodeSettingsPane

- Easy way to create dialogs
- Limited layout
- Limited complexity



Demo #4

Dialog with two components to select the integer columns that should be added and a component to set the name of the appended column.

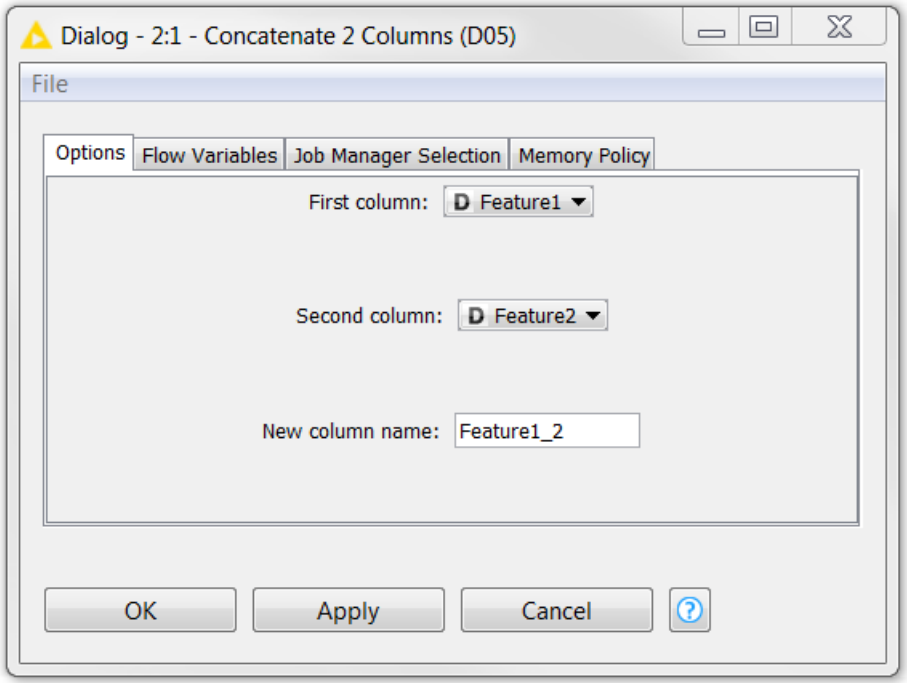


Exercise #4 – Dialog using standard components

- Implement a dialog where two string columns can be selected for concatenation and a name for the new concatenated column can be set
  - **`NodeDialog`**
    - Add a DialogComponentColumnNameSelection for the first and second column
    - Add DialogComponentString for the new column name
    - Add package private static methods for the SettingsModels
  - **`NodeModel`**:
    - Add member variables for the setting models from the static factory methods in the NodeDialog
    - Modify **`createOutSpec`** to use new value
    - Modify save/validate/load methods
- Use the skeleton classes in `org.knime.workshop.exercise.exercise04` and implement the todo's

Demo #5

Dialog with SWING components to select columns for concatenation.



Exercise #5 – Dialog using custom components(swing)

- Use a configuration object instead of the settings models to transport settings between the dialog and the model
- Use swing components instead of default components in dialog
- Implement a dialog where two string columns can be selected for concatenation and a name for the new concatenated column can be set
  - **`NodeDialog`**
    - Add a DialogComponentColumnNameSelection for the first and second column
    - Add DialogComponentString for the new column name
    - Add package private static methods for the SettingsModels
  - **`NodeModel`**:
    - Add member variables for the setting models from the static factory methods in the NodeDialog
    - Modify **`createOutSpec`** to use new value
    - Modify save/validate/load methods
- Use the skeleton in `org.knime.workshop.exercise.exercise05` package and implement the todo's

Data Handling

Overview

- HowTo on data creation in KNIME
- Discussion on **`execute(…)`** method in derived **`NodeModel`** class
  - Defines action that happens during node execution
  - Read input data, produce output data

Signature of model's `execute()` method

```
BufferedDataTable[] execute(BufferedDataTable[] ins,
        ExecutionContext exec) throws Exception {…}
```

- Node input data provided as **`BufferedDataTable`** array (length = #(data-)in-ports)
- Calculated data is returned as **`BufferedDataTable`** array (required length = #(data-)out-ports)

Signature of model's `execute()` method

```
BufferedDataTable[] execute(BufferedDataTable[] ins,
        ExecutionContext exec) throws Exception {…}
```

- **`BufferedDataTable`** – special implementation of **`DataTable`**, added benefit:
  - Controlled lifecycle
  - Efficient storage possibilities
  - Efficient BLOB (**B**inary **L**arge **OB**ject) handling
  - Data referencing from input
- **`ExecutionContext`**, used for
  - Progress information (**`exec.setProgress(…)`**),
  - Cancellation (**`exec.checkCancelled()`**),
  - Creation of **`BufferedDataTable`** (**`exec.create…(…)`**)

Construction of `BufferedDataTable`

1. `#createBufferedData{Table|Container}`
   - Create from scratch
2. `#createColumnRearrangeTable`
   - Reference input data, add/remove/replace column(s)
3. `#createSpecReplacerTable`
   - Reference input data with modified DataTableSpec
4. `#createConcatenateTable`
   - Row-wise concatenation of a set of tables
5. `#createJoinedTable`
   - Column-wise join of (correctly ordered!) tables

1. #createBufferedDataTable

- Copy a generic table (rarely used however):

```
BufferedDataTable result =
        exec.createBufferedDataTable(DataTable);
```

Might reside in main memory ☹

- Create BufferedDataContainer, sequentially add rows

```
BufferedDataContainer con =
        exec.createDataContainer(DataTableSpec);
while (…) {
    con.addRowToTable(DataRow);
}
con.close();
BufferedDataTable result = con.getTable();
```

Buffers to disk if necessary ☺

2. #createColumnRearrangeTable

- Create table with removed/added/replaced column(s), saves only columns that have changed
- Customization by using **`ColumnRearranger`** object
- Cell content of newly added columns supplied by **`CellFactory`** interface (with its default class **`SingleCellFactory`**)

```
BufferedDataTable input = ins[0];
CellFactory fac = new SingleCellFactory(DataColumnSpec) {
    public DataCell getCell(DataRow) {…}
};
ColumnRearranger arrange = new
    ColumnRearranger(input.getDataTableSpec());
arrange.append(fac);  // appends one new column
BufferedDataTable result =
    exec.createColumnRearrangeTable(input, arrange, exec);
```

2. #createColumnRearrangeTable

- **ColumnRearranger** allows for various column-based transformations, such as
  - **append(CellFactory)**
  - **insertAt(String, CellFactory)**
  - **keepOnly(String…)**
  - **remove(String…)**
  - **replace(CellFactory, String…)**
- **ColumnRearranger** can (should!) also be used to create output **DataTableSpec**
  - **ColumnRearranger#createSpec();**

address column names, also possible to use column indices (i.e. int)

3. #createSpecReplacerTable

- Change table specification of input table (e.g. rename/retype column), does not copy input data!
- Structure of new and previous **`DataTableSpec`** must be the same (e.g. same column count)
- No domain checking is done (may lead to inconsistent table state)

```
BufferedDataTable result = exec.createSpecReplacerTable(
        BufferedDataTable, DataTableSpec);
```

4. #createConcatenateTable

- Row wise concatenation of a set of input tables
- Tables must have same structure (same order of columns, same column names and types)
- Column domain information is set appropriately (e.g. possible values as union of input tables' possible values)
- Caller needs to assert that there are no duplicate row keys (otherwise execution fails!)

```
BufferedDataTable result = exec.createConcatenateTable(
        ExecutionMonitor, BufferedDataTable[]);
```

- **Rarely used** (only in cases such as parallelized node execution to merge the final results)

5. #createJoinedTable

- Column wise join of two input tables
- Tables must have disjoint column names
- Tables must have same row keys (in same order)

```
BufferedDataTable result = exec.createJoinedTable(
BufferedDataTable, BufferedDataTable, ExecutionMonitor);
```

- **Rarely used** (only in cases such as parallelized node execution to merge the final results)

Node execution – Nice to Know

- Execution triggered in separate thread
- **`BufferedDataTable`** objects can solely be created via **`ExecutionContext`**
- **`DataTableSpec`** of returned tables must structurally match (same column names and types) **`DataTableSpec`** returned by **`configure()`** (unless it returned null)
- **`BufferedDataTable`** construction can be arbitrarily encapsulated
- Node implementation should not keep the input tables as members (if need arises, for instance for a data view, copy parts of the data)!

Demo #6

- Review "`Concatenate Two Columns`"-node

- Use **`ColumnRearranger`** to append the result column

Exercise #6 – Data handling with Column Rearranger

- Use the **`ColumnRearranger`** in the "`Concatenate Two Columns`" node in both #configure() and #execute()
- Use the skeleton in `org.knime.workshop.exercise.exercise06` package and implement the todo's
- Optional: Add Boolean option to remove the source columns of the concatenation

Simple Streaming I

- Rows are pushed to the next node as soon as they are processed
- Prerequisite: processing of individual rows is independent from other rows
- Allows the next node to start computation before predecessor is finished

Simple Streaming II

- Implementation
  - Extend `SimpleStreamableFunctionNodeModel` instead of `NodeModel`
  - Overwrite `#createColumnRearranger`
  - Remove `#execute`
- Wrap streamable nodes and configure the wrapped metanode to use „Simple Streaming“ as the job manager

Demo #7

- Make “`Concatenate Two Columns`”-node streamable

Exercise #7 – Simple Streaming

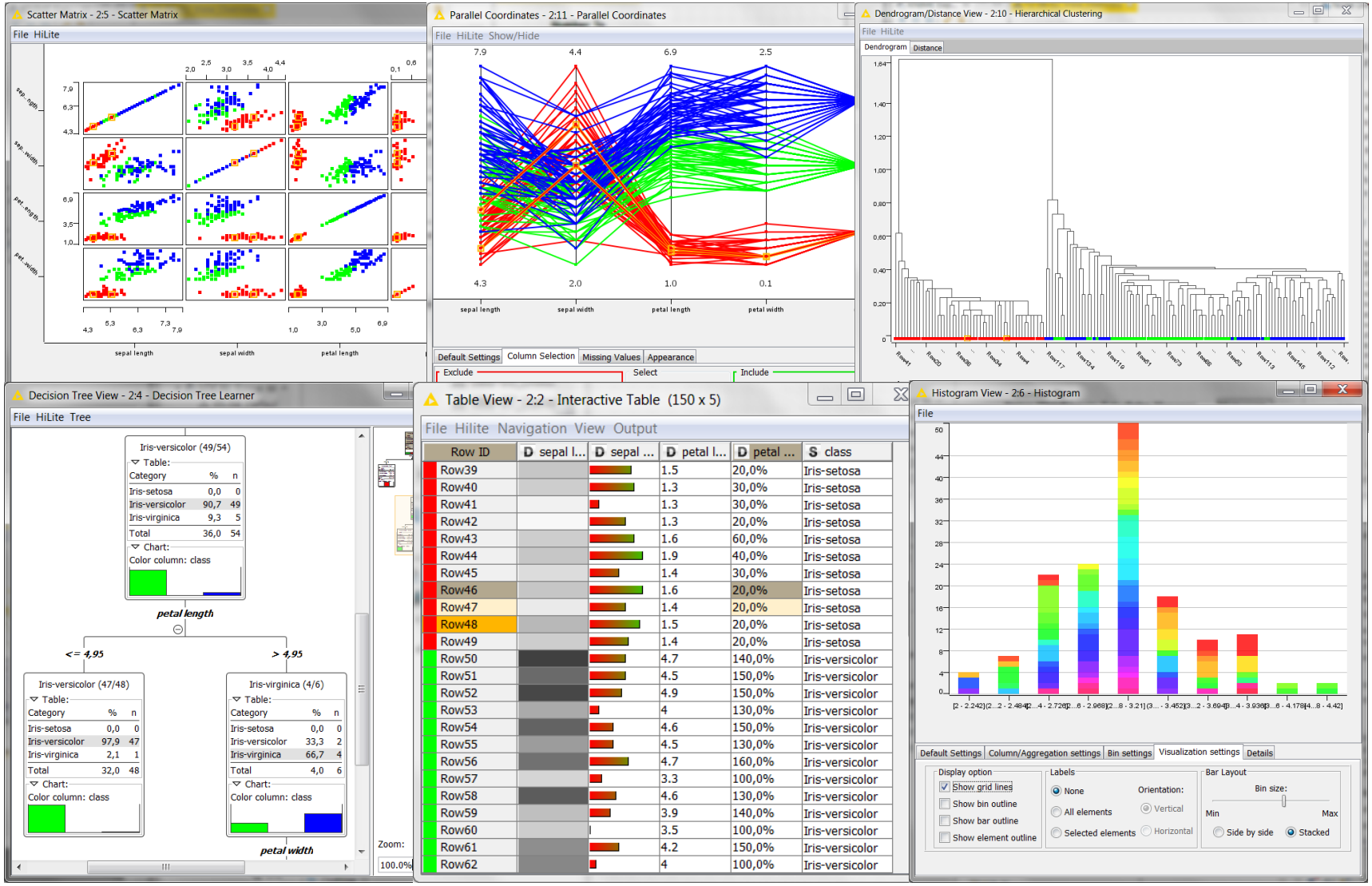
- Make the "`Concatenate Two Columns`"-node streamable
  - Extend **`SimpleStreamableFunctionNodeModel`** instead of **`NodeModel`**
  - Overwrite **`#createColumnRearranger`**
  - Remove **`#execute`**

NodeView

Overview

- Implementation:
  - `#load-/#saveInternals`
  - `HiLiteListener`
  - `HiLiteHandler`
  - `HiLiteTranslator`

KNIME NodeViews



Implementing Views in KNIME

- **`NodeModel#execute`**: create data structure to visualize
- **`NodeView#modelChanged`**: interpret data structure
- Local information in NodeView (several instances per Node are possible)
- Everything data model specific in **`modelChanged`** (not in constructor: model can change while view is open)

Why #load-/#saveInternals?

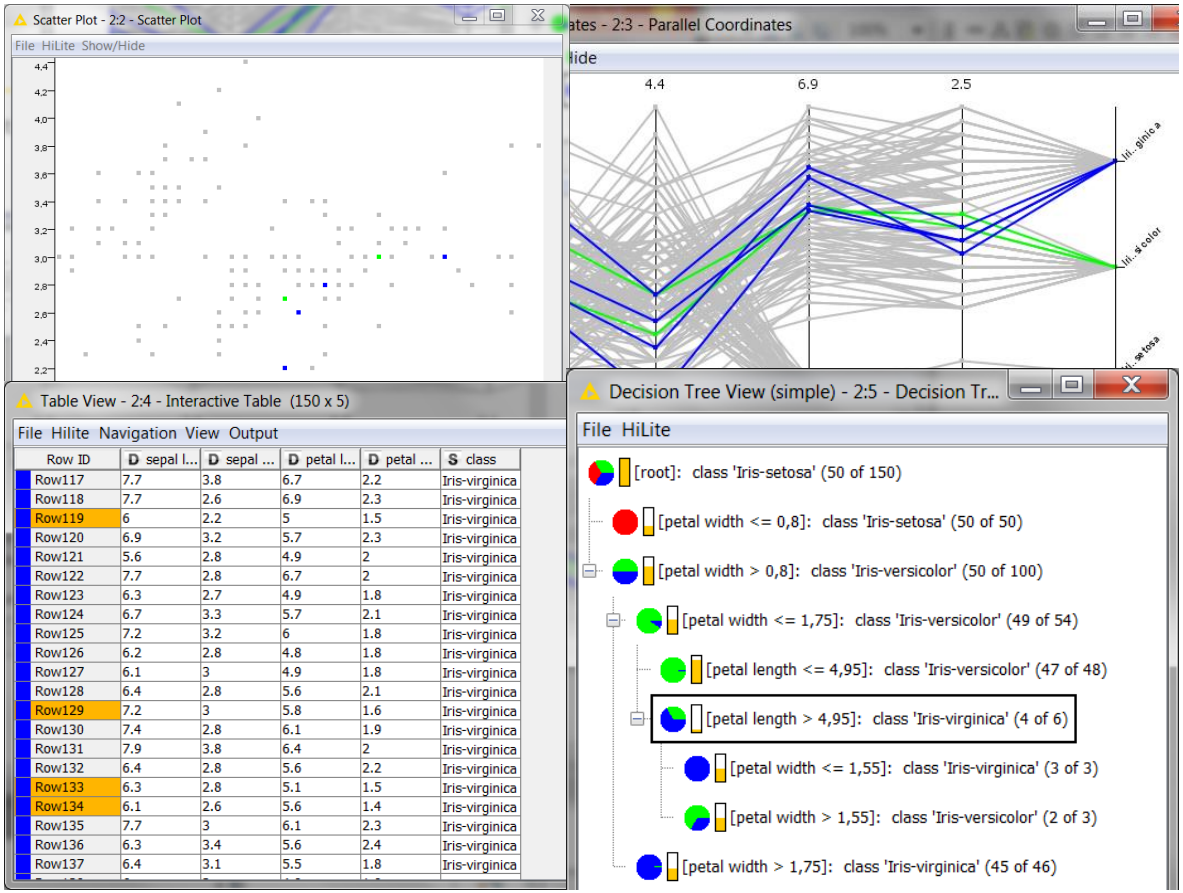
- Internal models for the view(s) not automatically stored
- Special structure for every node, not predictable
- Node has to care about loading and storing of the specific content

#load-/#saveInternals

- **`#saveInternals`** stores all relevant fields of the internal model as **`NodeSettings, DataTable`** or in an own structure (not recommended)
- **`#loadInternals`** retrieves these fields and restores internal model with these values
- **`#reset`** corresponds to load/saveInternals: should delete what is loaded/saved

What is HiLiting?

- Linking&Brushing: select data points of interest and identify them in any other view



HiLiting vs. Selection

- Selection is local, HiLiting is global

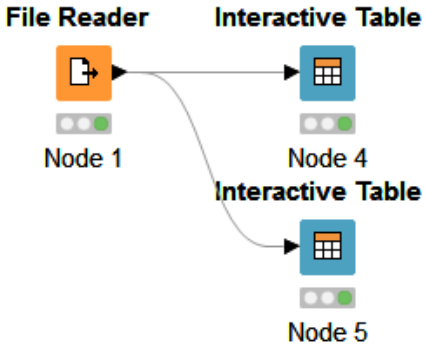


Table View - 3:4 - Interactive Table "iris....

File Hilite Navigation View Output

Row ID	D sepal l...	D sepal ...	D petal l...	D petal ...
Row0	5.1	3.5	1.4	0.2
Row1	4.9	3	1.4	0.2
Row2	4.7	3.2	1.3	0.2
Row3	4.6	3.1	1.5	0.2
Row4	5	3.6	1.4	0.2
Row5	5.4	3.9	1.7	0.4
Row6	4.6	3.4	1.4	0.3
Row7	5	3.4	1.5	0.2
Row8	4.4	2.9	1.4	0.2
Row9	4.9	3.1	1.5	0.1

Table View - 3:5 - Interactive Table "iris...

File Hilite Navigation View Output

Row ID	D sepal l...	D sepal ...	D petal l...	D petal ...
Row0	5.1	3.5	1.4	0.2
Row1	4.9	3	1.4	0.2
Row2	4.7	3.2	1.3	0.2
Row3	4.6	3.1	1.5	0.2
Row4	5	3.6	1.4	0.2
Row5	5.4	3.9	1.7	0.4
Row6	4.6	3.4	1.4	0.3
Row7	5	3.4	1.5	0.2
Row8	4.4	2.9	1.4	0.2
Row9	4.9	3.1	1.5	0.1

HiLiteHandler

- HiLiting is propagated through the whole workflow (upwards and downwards)
- For all nodes with the same **`HiLiteHandler`**

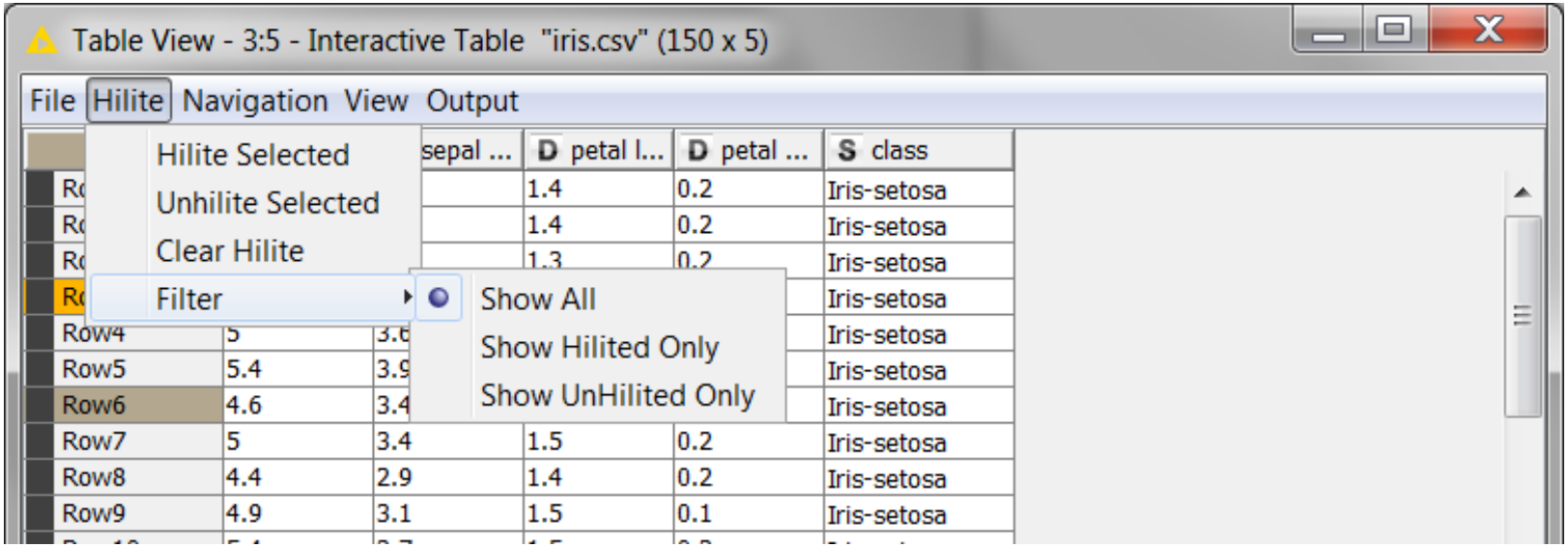
- HiLiting can be translated from one **`HiLiteHandler`** to another

- HiLiting is connected to the rowIDs

HiLiteListener

- Get informed about HiLite events
- Display hilited data differently
- Implement **`HiLiteListener`**:
  - **`#hiLite(KeyEvent)`**
  - **`#unHiLite(KeyEvent)`**
  - **`#unHiLiteAll(KeyEvent)`**
- **`KeyEvent`** contains all **`RowKey`** instances to be changed

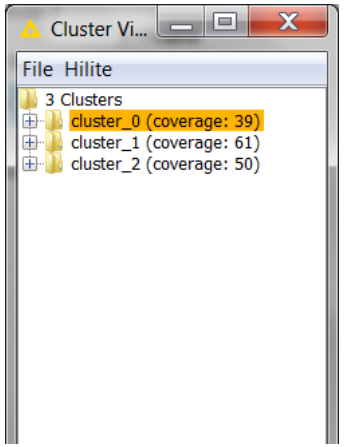
HiLite Functionality

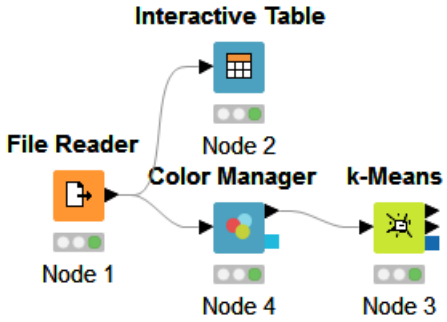


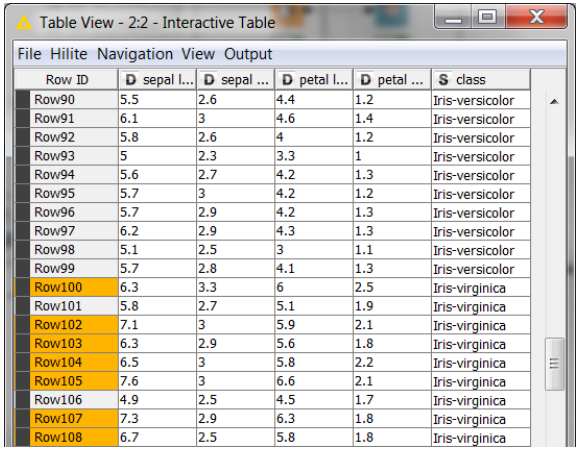
- `#fireHiLiteEvent(...)`
- `#fireUnHiLiteEvent(...)`
- `#fireClearHiLiteEvent(...)`

HiLite Translation

- Sometimes a translation between two or more HiLiteHandlers is needed
- Examples:
  - Cluster -> contained data points
  - Rule -> covered data points
  - Nominal Value -> all rows with this value



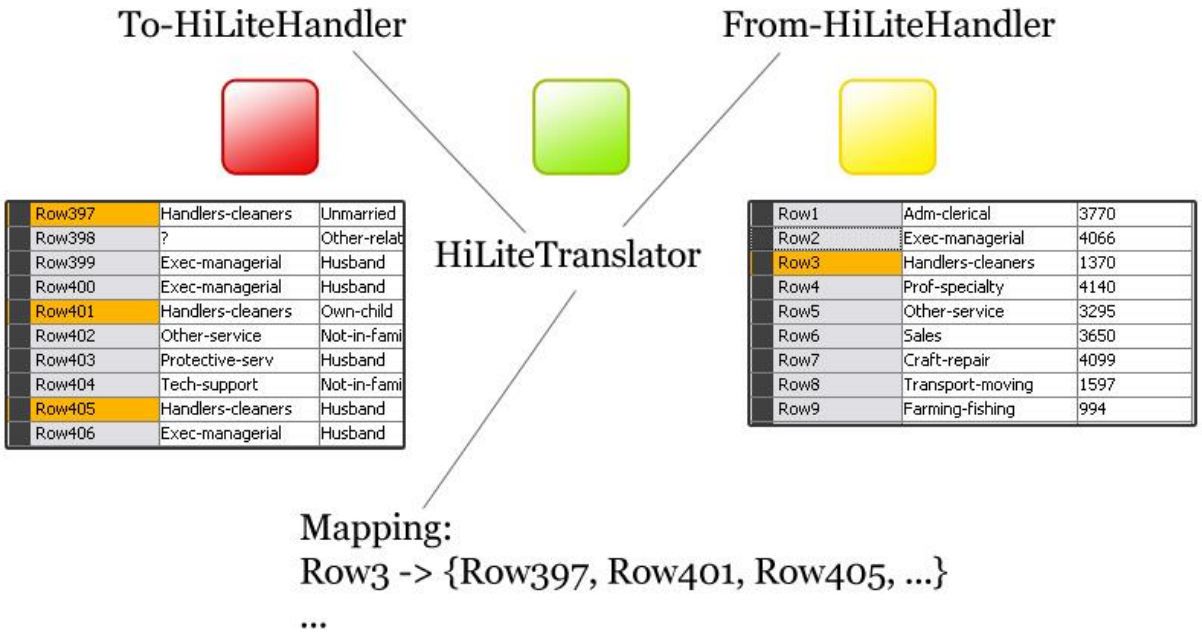




Row ID	D sepal l...	D sepal ...	D petal l...	D petal ...	S class
Row90	5.5	2.6	4.4	1.2	Iris-versicolor
Row91	6.1	3	4.6	1.4	Iris-versicolor
Row92	5.8	2.6	4	1.2	Iris-versicolor
Row93	5	2.3	3.3	1	Iris-versicolor
Row94	5.6	2.7	4.2	1.3	Iris-versicolor
Row95	5.7	3	4.2	1.2	Iris-versicolor
Row96	5.7	2.9	4.2	1.3	Iris-versicolor
Row97	6.2	2.9	4.3	1.3	Iris-versicolor
Row98	5.1	2.5	3	1.1	Iris-versicolor
Row99	5.7	2.8	4.1	1.3	Iris-versicolor
Row100	6.3	3.3	6	2.5	Iris-virginica
Row101	5.8	2.7	5.1	1.9	Iris-virginica
Row102	7.1	3	5.9	2.1	Iris-virginica
Row103	6.3	2.9	5.6	1.8	Iris-virginica
Row104	6.5	3	5.8	2.2	Iris-virginica
Row105	7.6	3	6.6	2.1	Iris-virginica
Row106	4.9	2.5	4.5	1.7	Iris-virginica
Row107	7.3	2.9	6.3	1.8	Iris-virginica
Row108	6.7	2.5	5.8	1.8	Iris-virginica

HiLite Translator

- **`HiLiteTranslator`** translates between **`HiLiteHandlers`** (incoming – outgoing)
- Uses a **`HiLiteMapper`**: maps between one rowID and one or more other rowIDs



Row397	Handlers-cleaners	Unmarried
Row398	?	Other-relat
Row399	Exec-managerial	Husband
Row400	Exec-managerial	Husband
Row401	Handlers-cleaners	Own-child
Row402	Other-service	Not-in-fami
Row403	Protective-serv	Husband
Row404	Tech-support	Not-in-fami
Row405	Handlers-cleaners	Husband
Row406	Exec-managerial	Husband

Row1	Adm-clerical	3770
Row2	Exec-managerial	4066
Row3	Handlers-cleaners	1370
Row4	Prof-specialty	4140
Row5	Other-service	3295
Row6	Sales	3650
Row7	Craft-repair	4099
Row8	Transport-moving	1597
Row9	Farming-fishing	994

Loading/Saving of the Mapping

- **`DefaultHiLiteMapper`** provides two methods for loading/saving of the Mapping:
  - **`DefaultHiliteMapper#load(ConfigRO cfg)`**
  - **`DefaultHiliteMapper#save(ConfigWO cfg)`**

JavaScript Views

- Introduced to KNIME Labs in v3.0
- Three flavors:
  - Individual nodes: Fast and easy to use
  - Generic JavaSript View: Create your own visualization
  - JavaScript in **`NodeView`**
- Based on
  - D3
  - JSFreeChart
  - jQuery
  - jQuery UI

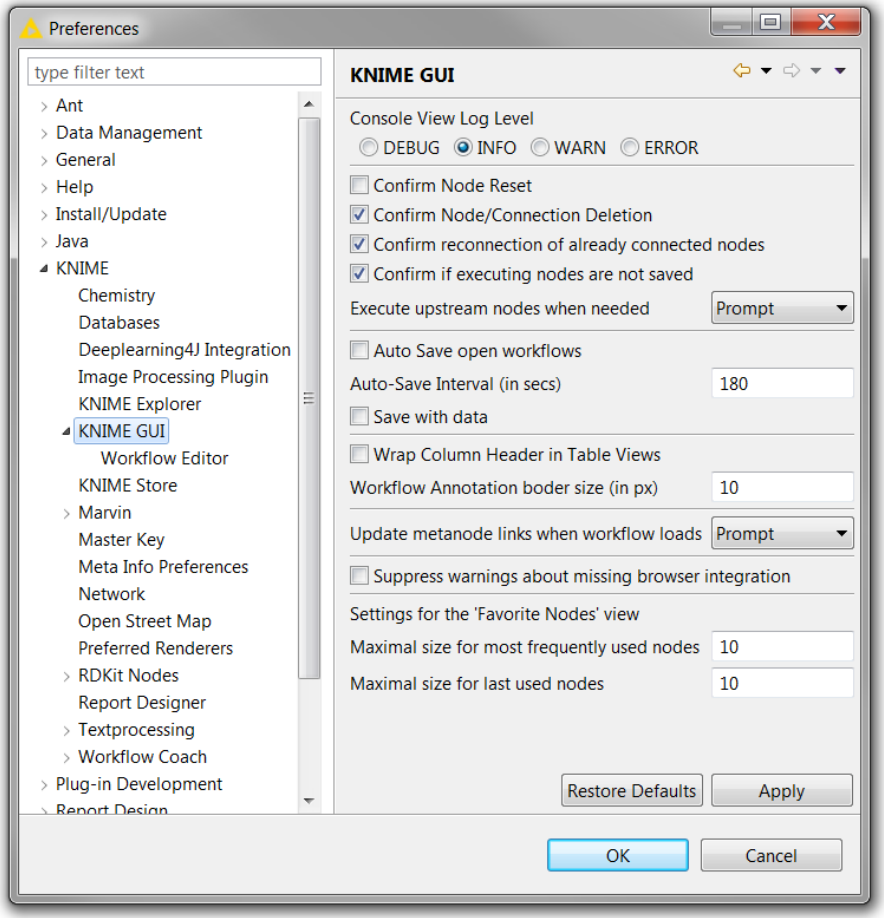
Demo

- Generic JavaScript View with D3.js in 10 minutes
- Based on KNIME blog post
- Example workflow available on USB-Stick
  `/d3_days-hour_heatmap_example.knwf`

Best Practices & Noding Guidelines

Preference Pages

- Global settings that apply to all workflows should be stored in “Preference Pages”
- Examples:
  - Web Service URLs
  - Executable Paths
- If the user is not supposed to change it: use config files



Database Access & Credentials

- Nodes that require credentials should use KNIME credentials provider:
  `NodeModel#getCredentialsProvider()`
- Passwords not stored as part of workflow, queried when opened (also on server)
- Used also in database nodes

GUI and Model Separation

- General advice: Separate UI from model
- Model should not use any UI classes
- Be aware of static initializers and icons

→ Problems when KNIME is run on the server or in batch mode

Logger

- Use **LOGGER** instead of **System.err/out**.
- Bad practice:

```
System.err.println("Some message");
e.printStackTrace();
```

- Use instead:

```
private static final NodeLogger LOGGER =
NodeLogger.getLogger(nodeModel.class)
…
LOGGER.error("Error message", Throwable);
```

- Warning messages on the node:
  - **NodeModel#setWarningMessage(String)**

Backward Compatibility

- If you change a node, it should still be able to read settings of existing workflows properly
- If not possible, deprecate old node and create a new node
  - Deprecate by setting „`deprecated`“ to **`true`** in `plugin.xml` (node extension) but leave it in place

Source Code Organization

Reuse existing code:

- Host source code used in different plugins in one base plugin (declare dependencies)
- Use existing libraries (e.g. Apache Commons – all available as eclipse plugins)
- Use KNIME nodes as reference (nodes often not API but can give good coding hints) → the main reason why it is open source (Use Eclipse´s „Navigate“ menu to find code)

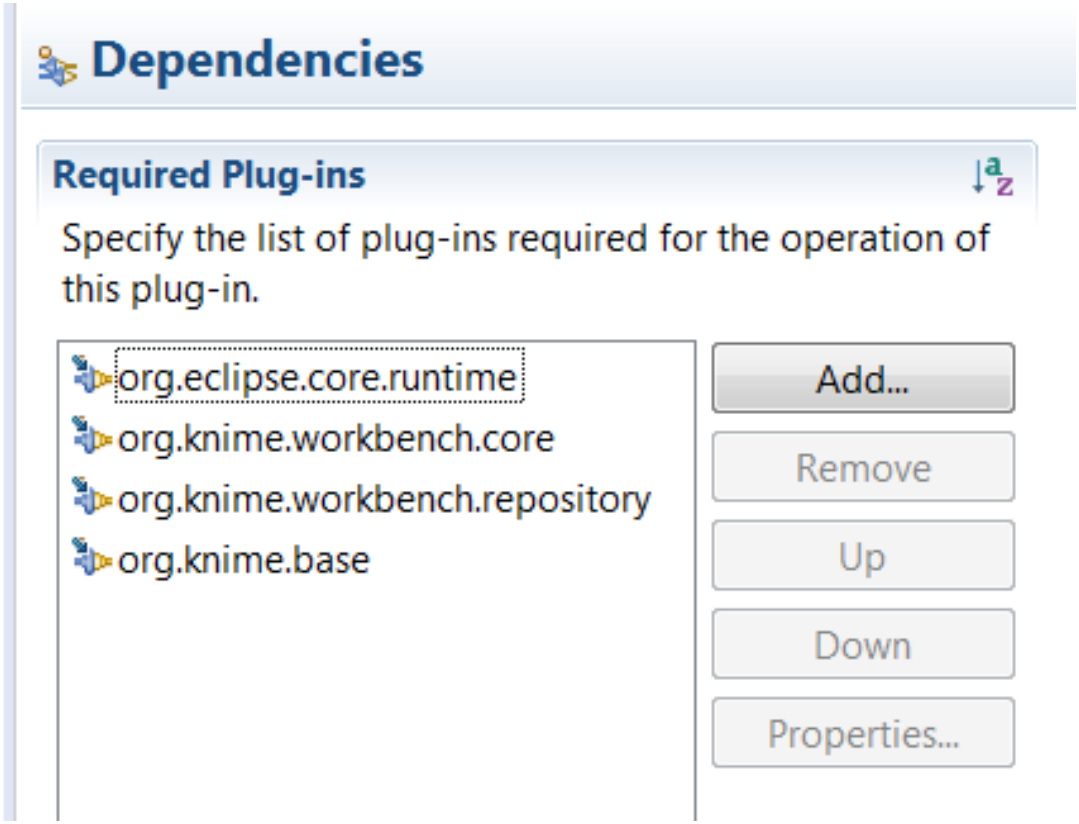
Dependencies I

- If you need external libraries
  - First check if there is already an Eclipse plug-in, e.g. at Eclipse Marketplace
  - Create a new plug-in containing only this library
    - Preferred way
  - Use library directly in your plug-in but do not export its packages
    - Only for very specific/exotic libraries

Dependencies II

- Add version ranges on dependencies
  - Potential API breakages with new major version
  - New API with new minor version
  - Bug fixes with new revision
- Minimum version (incl) = current version
- Maximum version (excl) = next major version

Dependencies III



Noding Guidelines

- https://www.knime.com/sites/default/files/inline-images/noding_guidelines.pdf
  - General guidelines how a KNIME node should behave

- Additional Developer information under https://www.knime.com/developers

Deployment

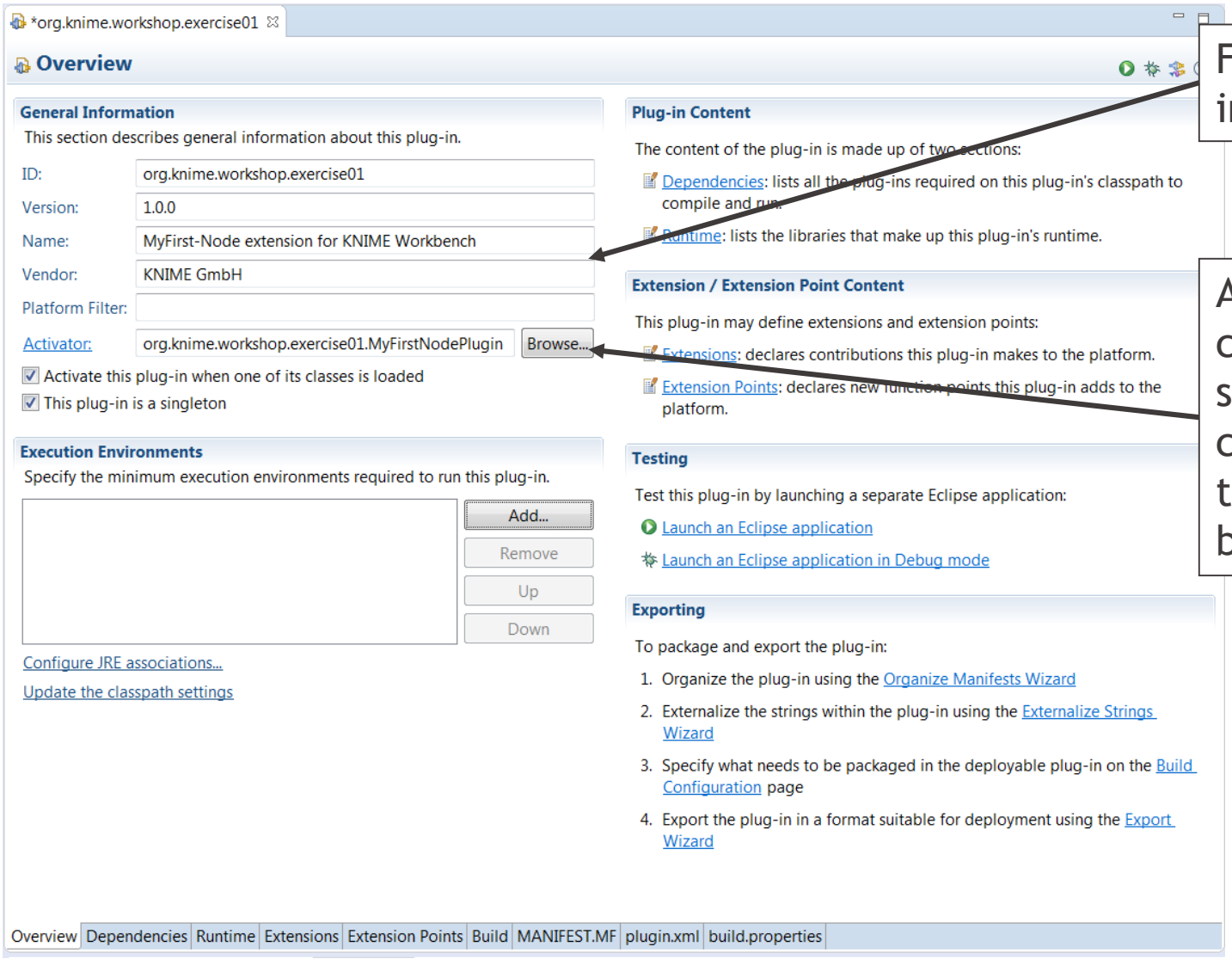
Overview

- Plug-in internals
- Features
- Update Sites

The Plug-in Manifest

- Contains information about runtime and build of the plug-in
- Consists of
  - META-INF/MANIFEST.MF
  - build.properties
  - plugin.xml
- All editable with the Plug-in Manifest Editor
  - Double-click on META-INF/MANIFEST.MF

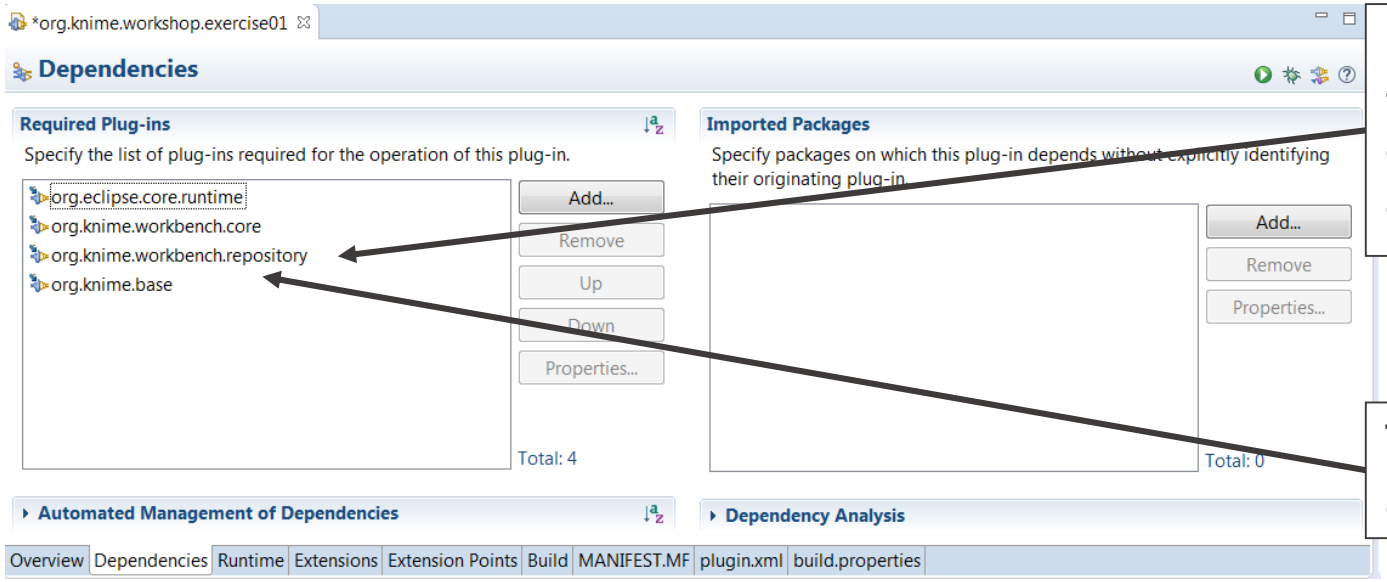
Plug-In Manifest – Overview



Fill in your personal information

A plug-in activator class can be specified, which is called once, at the time the plug-in is being activated.

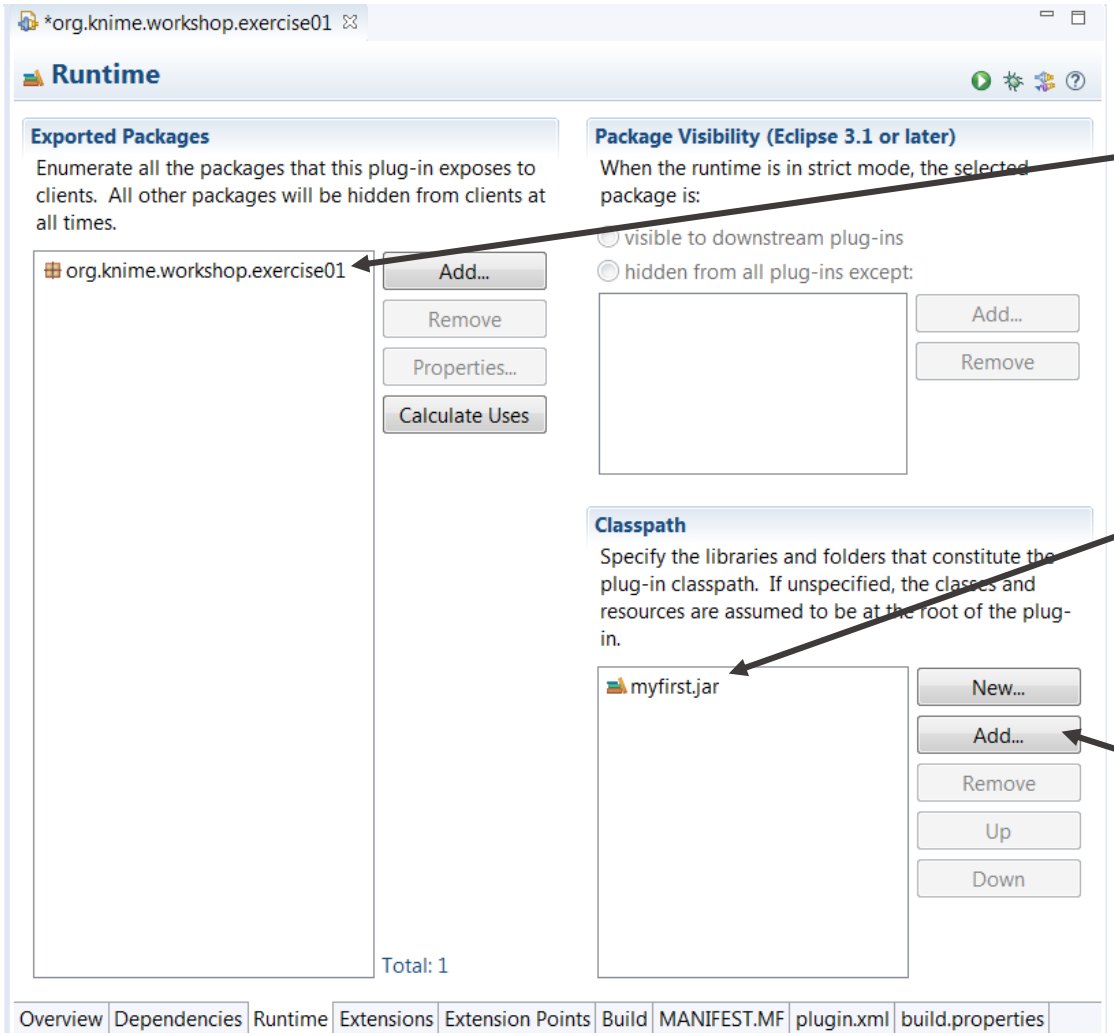
Plug-In Manifest – Dependencies



If you use classes from other plug-ins, you must add these plug-ins here.

These plugins are always needed.

Plug-In Manifest – Runtime



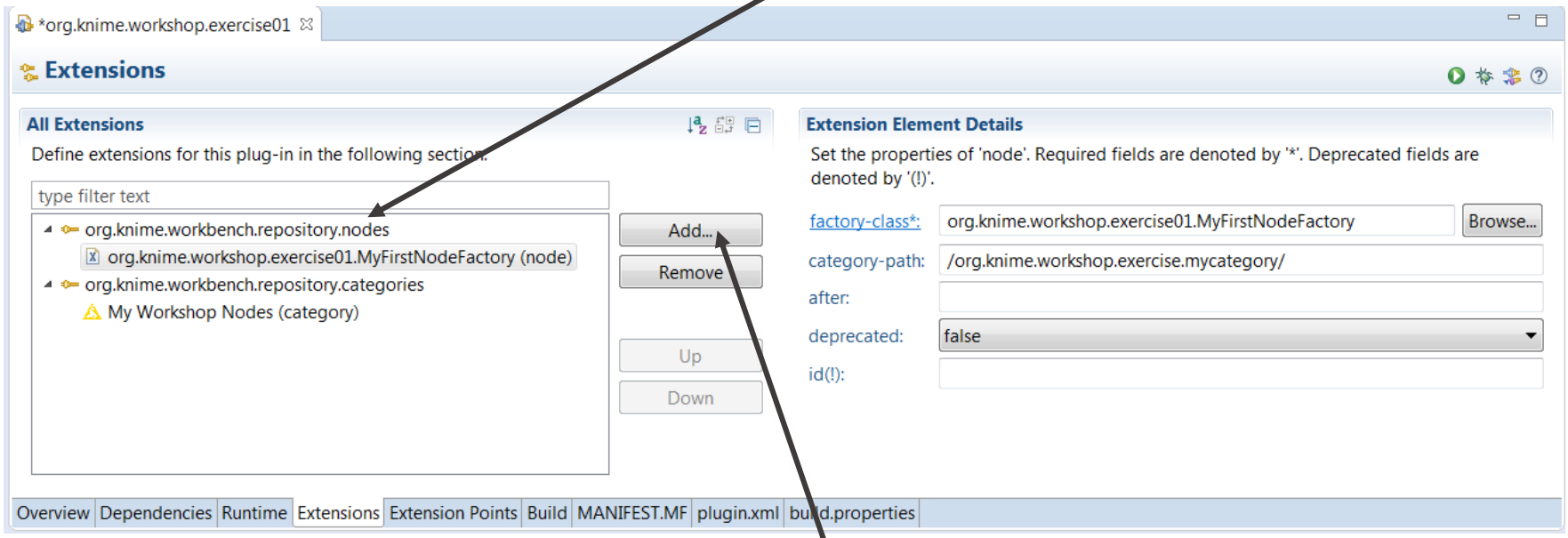
Add all packages that contain classes used by other plug-in projects.

Add „.“ or the JAR that is defined in the Build tab to the classpath (with the „New...“ button)

If your project uses an external library, use „Add...“ and specify it here.

Plug-In Manifest – Extensions I

Nodes are extensions to this.
To register a new node:
Right-click → New… → „node“



To register a new extension (e.g. a new repository category):
Click „Add…“ and select one of „org.knime.workbench…“

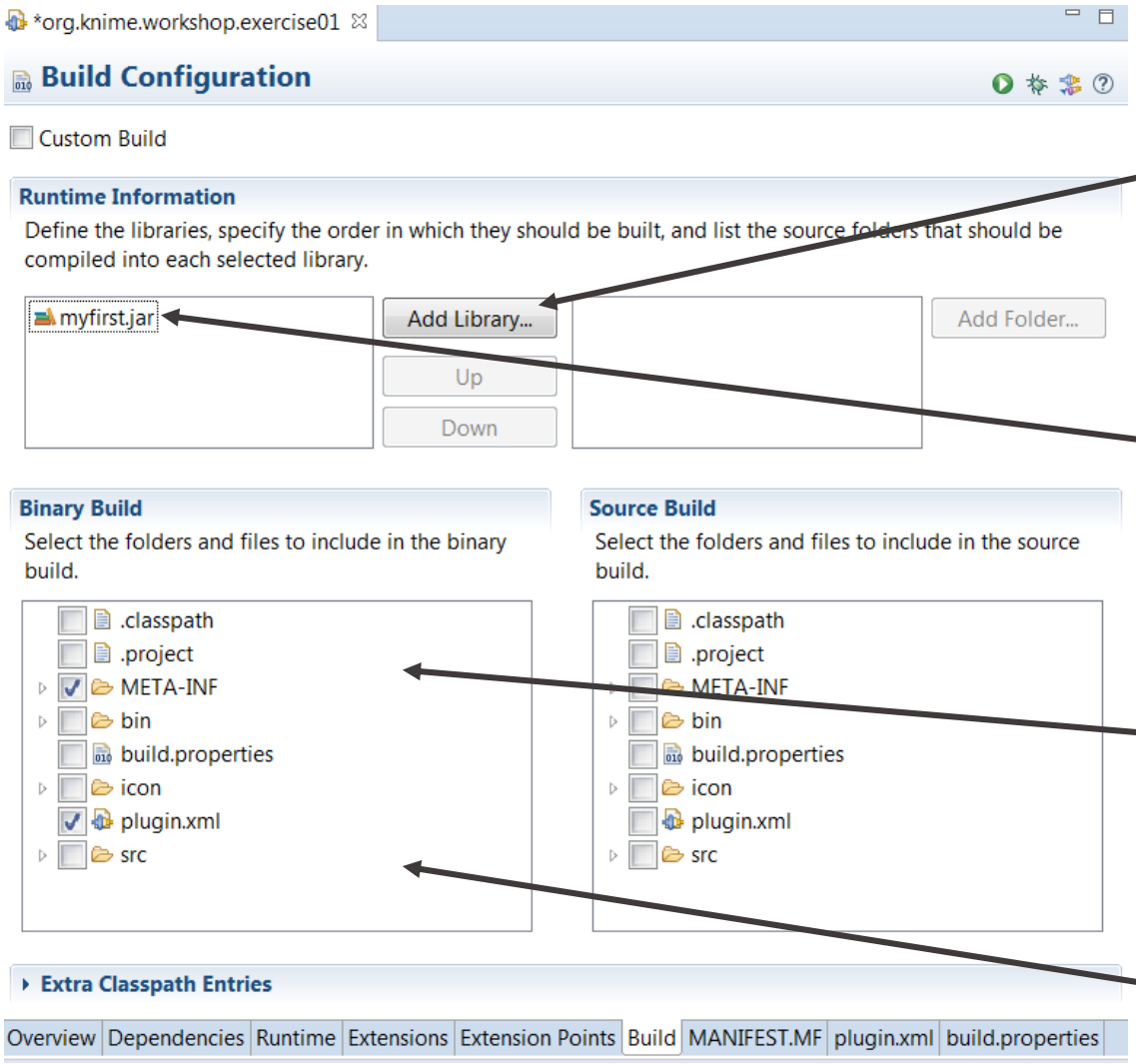
Plug-in Manifest – Extensions II

- Node extension properties
  - *id* – a unique id for the node, e.g. the class name
  - *factory-class* – the node factory´s class name
  - *category-path* – full path of the category in the node repository
    - Look into `plugin.xml` files to find out the category identifiers
  - *after* – id of a node in the same category after which this node should be listed
    - By default alphabetical order

Plug-in Manifest – Extensions III

- Category extension properties
  - *name* – of the category shown in the node repository
  - *path* – full path of the parent category
  - *level-id* – unique id of this category
  - *after* – id of a category in the parent category after which this node should be listed
    - By default alphabetical order
  - *description* – currently not used
  - *icon* – path (relative to plug-in root) to the icon

Plug-In Manifest – Build



Define the project´s library: „Add Library...“, enter a name. Select it, „Add Folder...“, select the source code folder.

To remove a library, right-click on it, then „Delete“.

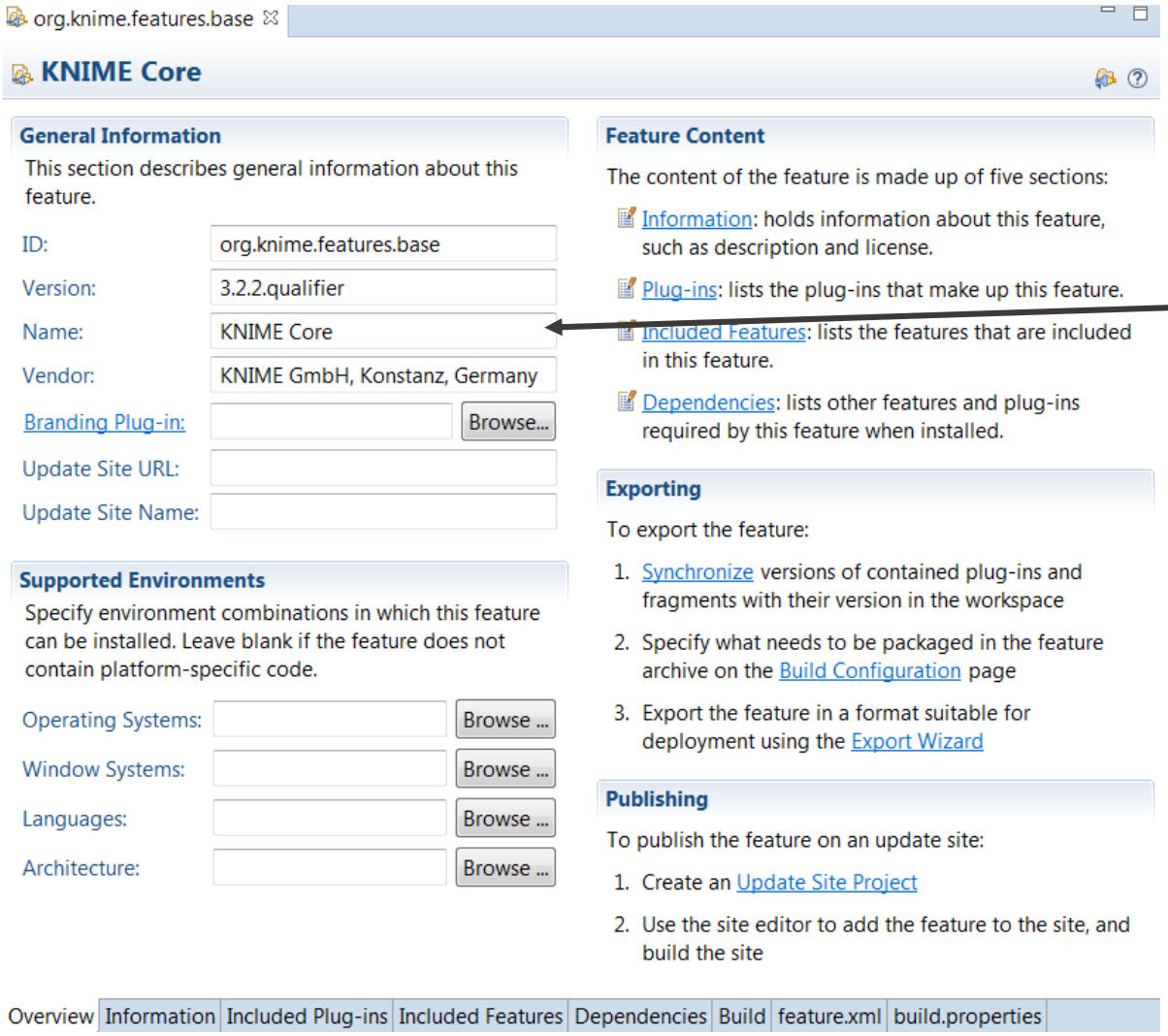
Check all files that should be included in the built plug-in. Include README files, License or Release Notes.

Include all external libraries (if required).

Features

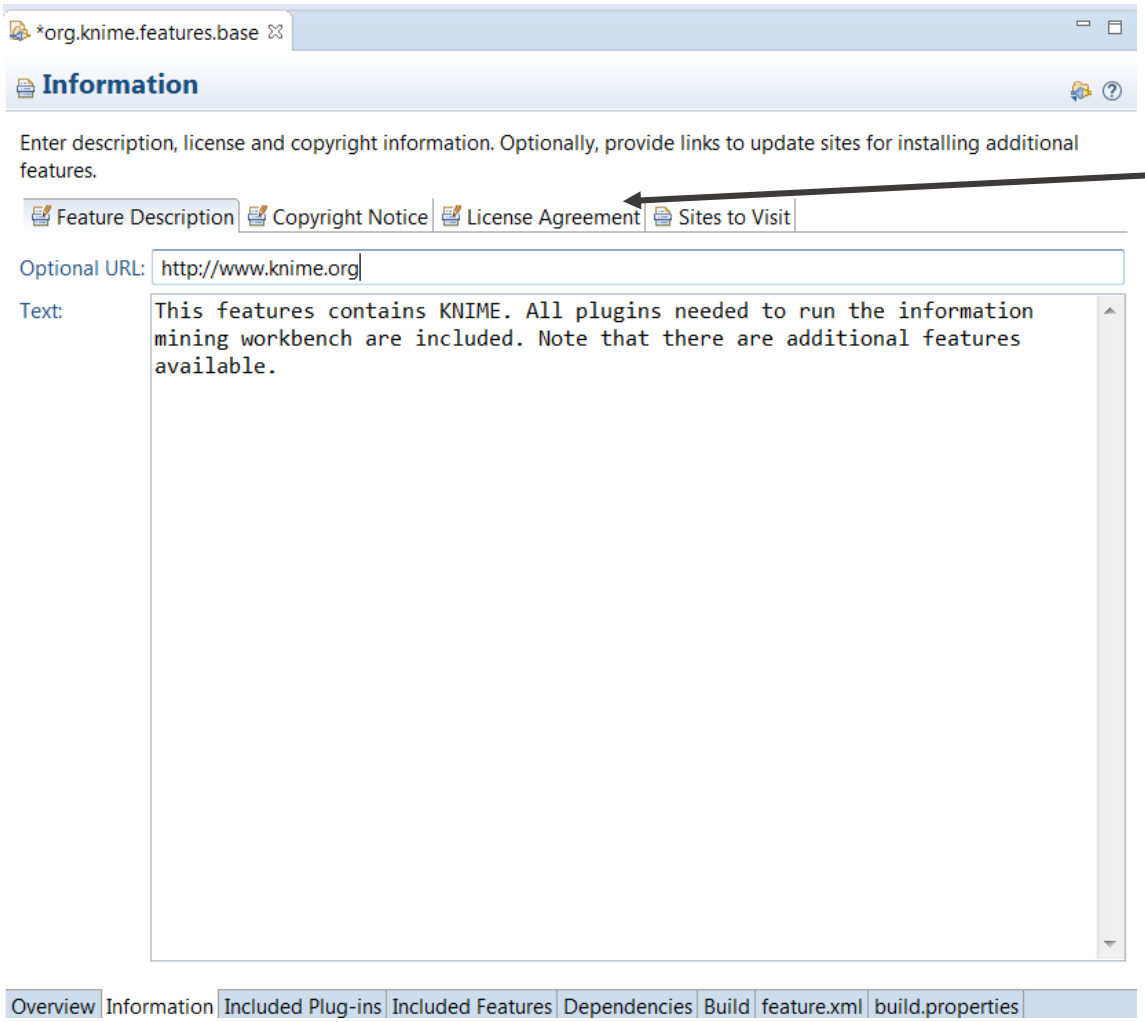
- A feature is a collection of one or more plug-ins
- Only features can be installed into KNIME
  - Not needed during development
- Separate Eclipse project type for features
- `feature.xml` editable with Feature Manifest editor

Feature Manifest – Overview



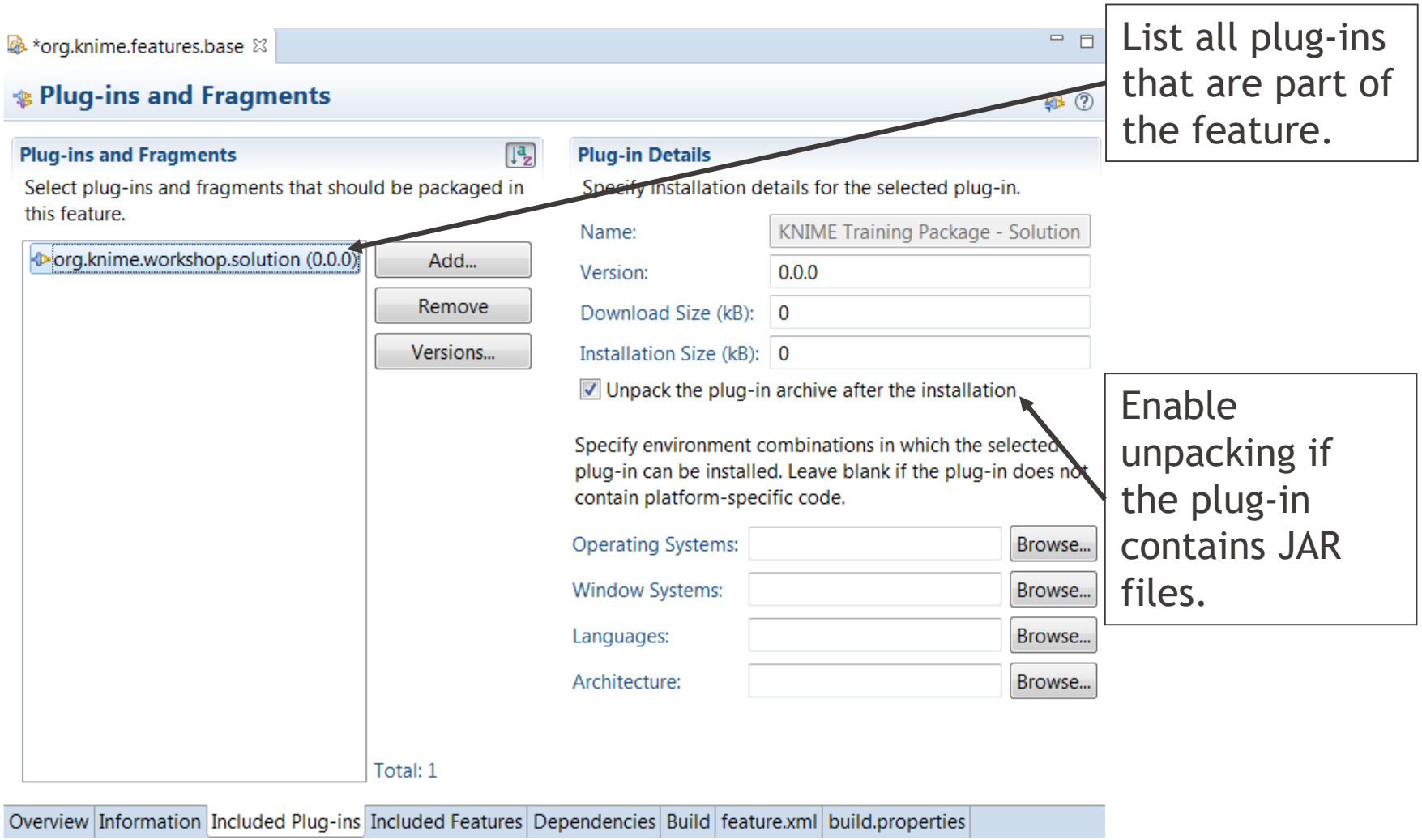
Information similar to plug-in´s

Feature Manifest – Information



The user will see this information during installation, therefore fill out all three fields with sensible data.

Feature Manifest – Plug-ins



List all plug-ins that are part of the feature.

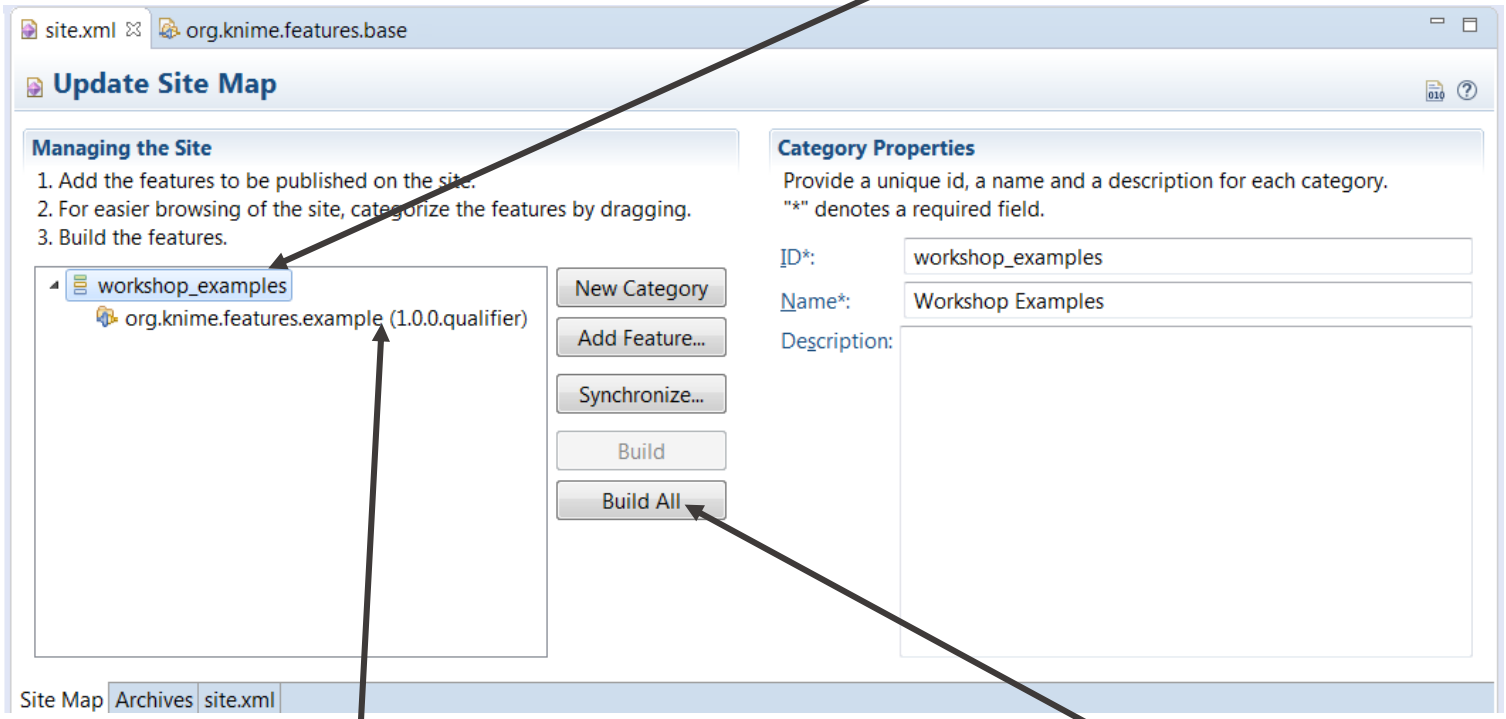
Enable unpacking if the plug-in contains JAR files.

Update Sites

- Bundle one or more features
- Available from a webserver, directory or ZIP file
- Separate Eclipse project type for update sites
- `site.xml` editable with the Site Manifest Editor

Site Manifest – Site Map

Create one or more categories for your features.

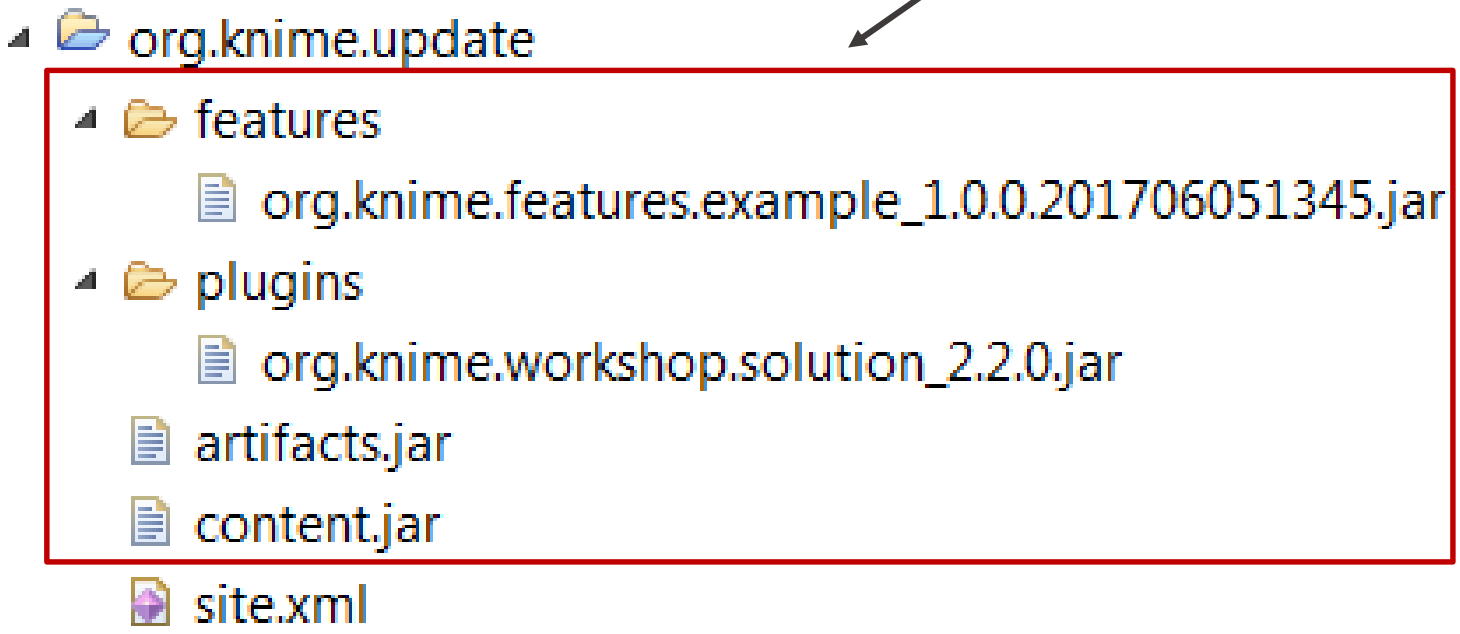


Add one or more features to the categories.

Builds the Update Site.

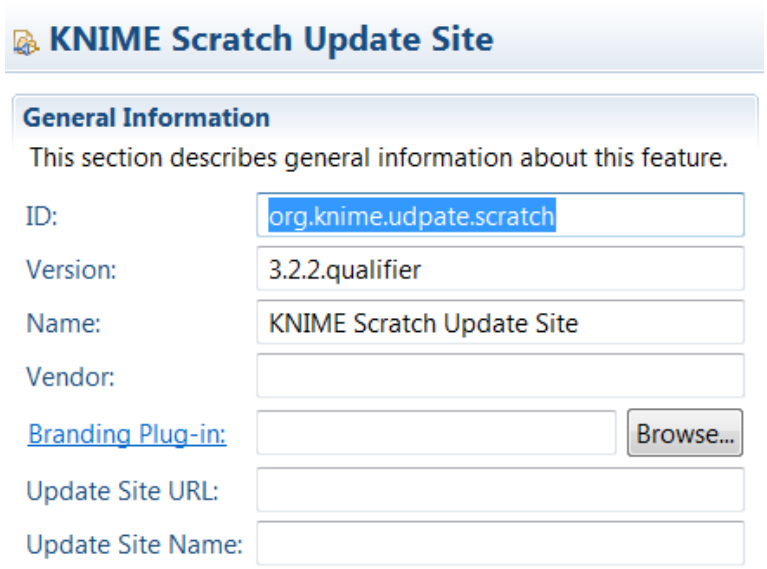
Final Update Site

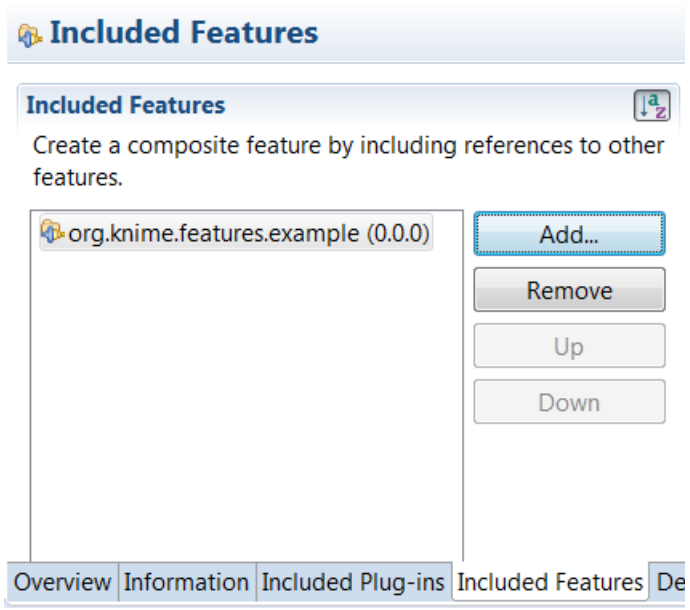
Copy these files onto your webserver or archive them into a ZIP file.



Update Sites with Buckminster I

- Update site is a normal feature project
- Add all features for the update site in „Included Features“





Update Sites with Buckminster II

- Categorize features via build.properties
  - XXXX can be replaced with any identifier

```
category.id.XXXX=Name of the category as shown to the user
category.members.XXXX=Comma-separated list of feature IDs in the category
category.description.XXXX=Description of the category
```

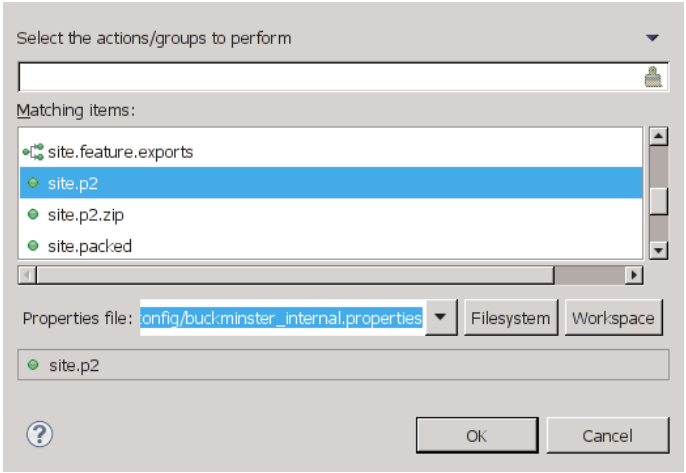
Update Sites with Buckminster III

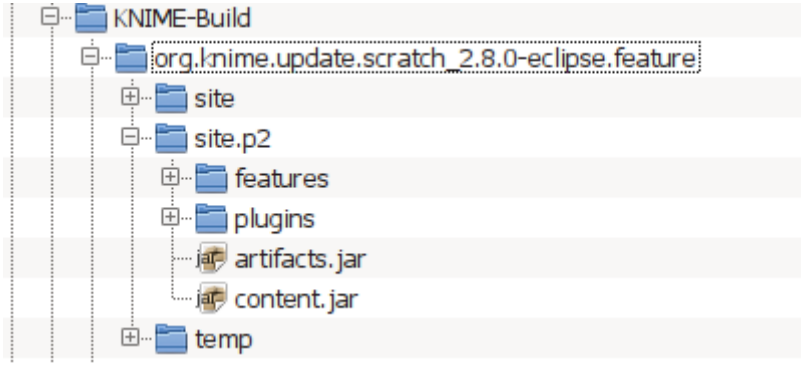
- Create properties file for Buckminster

```
buckminster.output.root=Output folder for Buckminster, e.g. /tmp/buckminster
buckminster.temp.root=${buckminster.output.root}/tmp

# how .qualifier in versions should be replaced
qualifier.replacement.*=generator:lastRevision
generator.lastRevision.format={0,number,0000000}
target.os=*
target.ws=*
target.arch=*
target.nl=en
```

Update Sites with Buckminster IV

- Invoke Buckminster action „site.p2“
- Output will be in `${buckminster.output.root}/us-feature-id_version-eclipse.feature/site.p2`





Testing

Overview

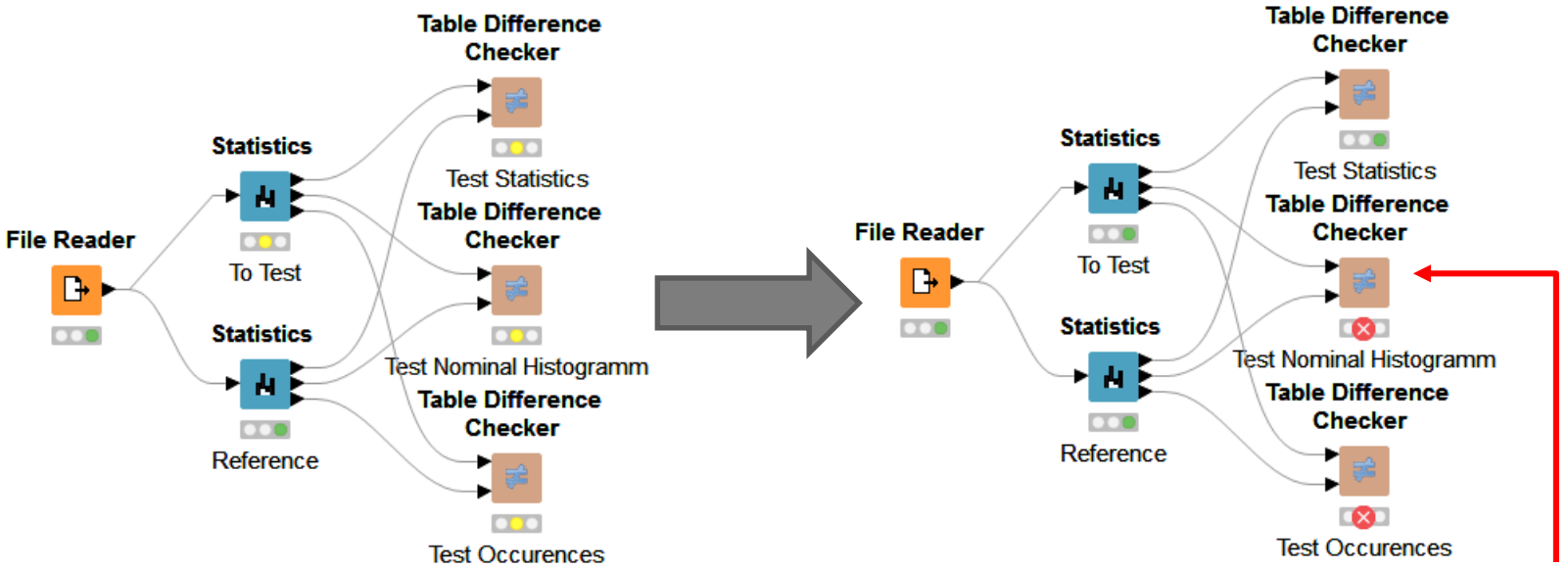
- Test workflows
- Test automation

Testing

- KNIME-independent modules are best tested with standard JUnit tests
- Nodes can be tested with test workflows („Testflows“)

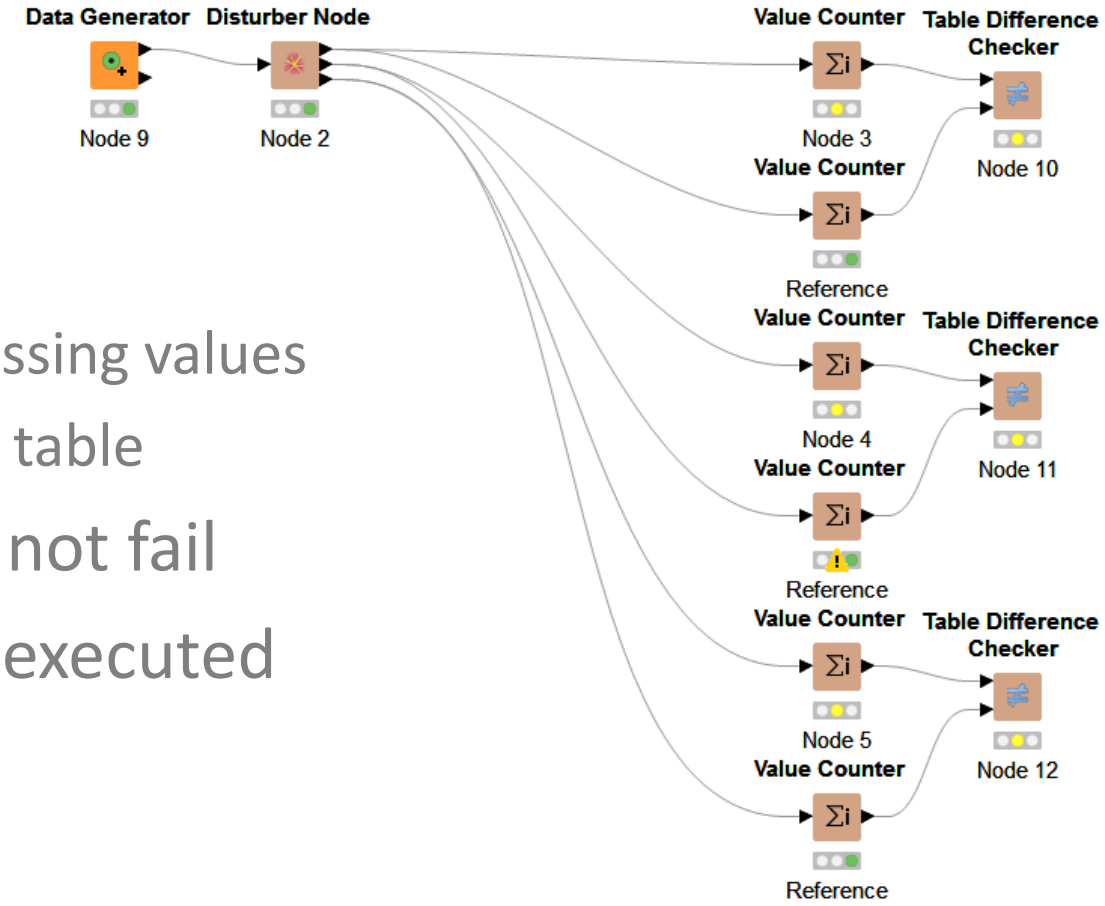
Testflows I

- Simple tests for all node functionality
- Reference table is compared with results of current execution
  - Reference node is saved in executed state



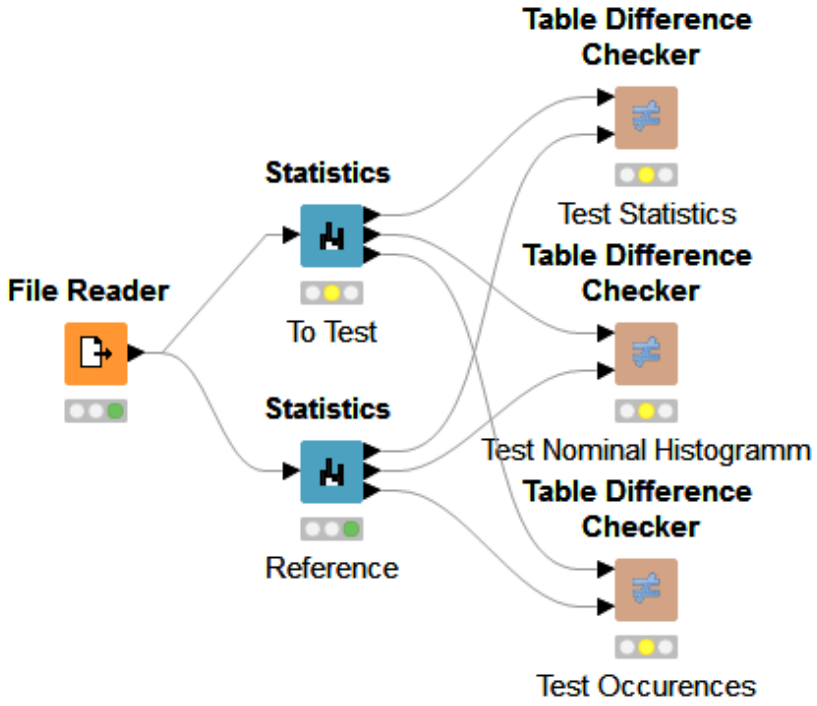
Testflows II

- Disturber node
  - Inserts random missing values
  - Outputs an empty table
- Node to test must not fail
- All nodes must be executed in the end



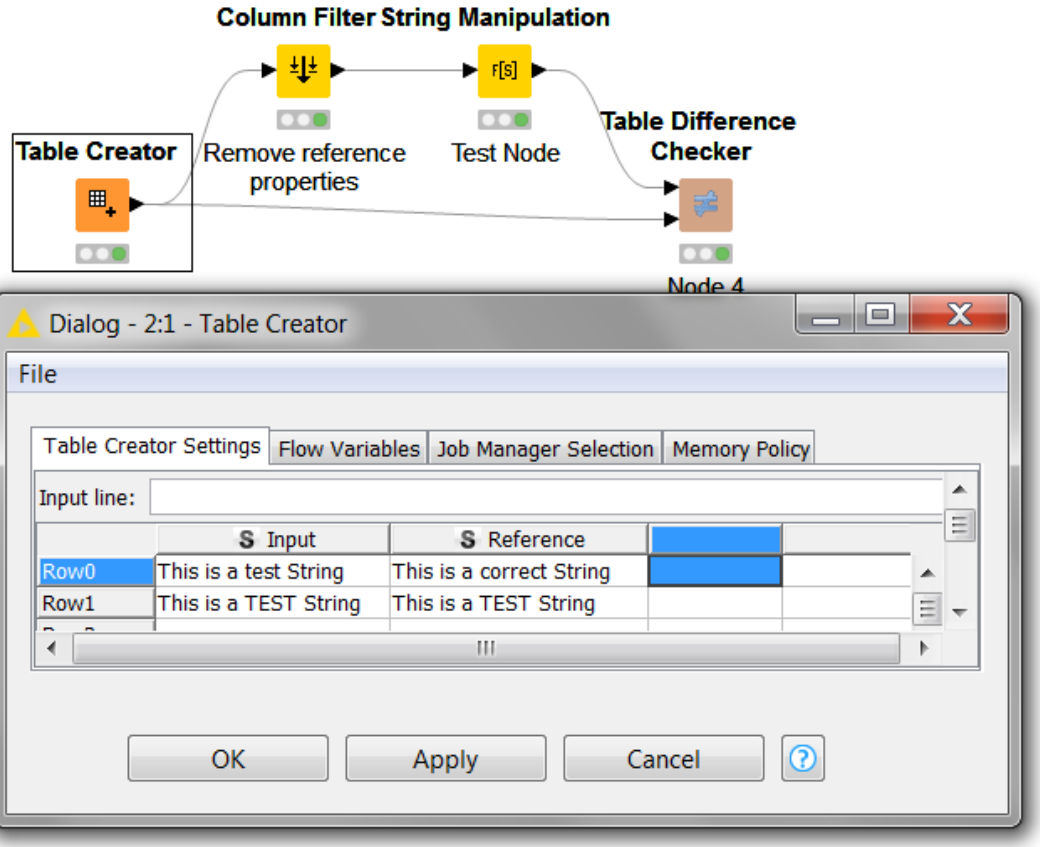
Compatibility Test

- Check if new node version computes same results as previous versions
- Easy setup with two copies of the same node, the reference node is saved executed (i.e. with data)



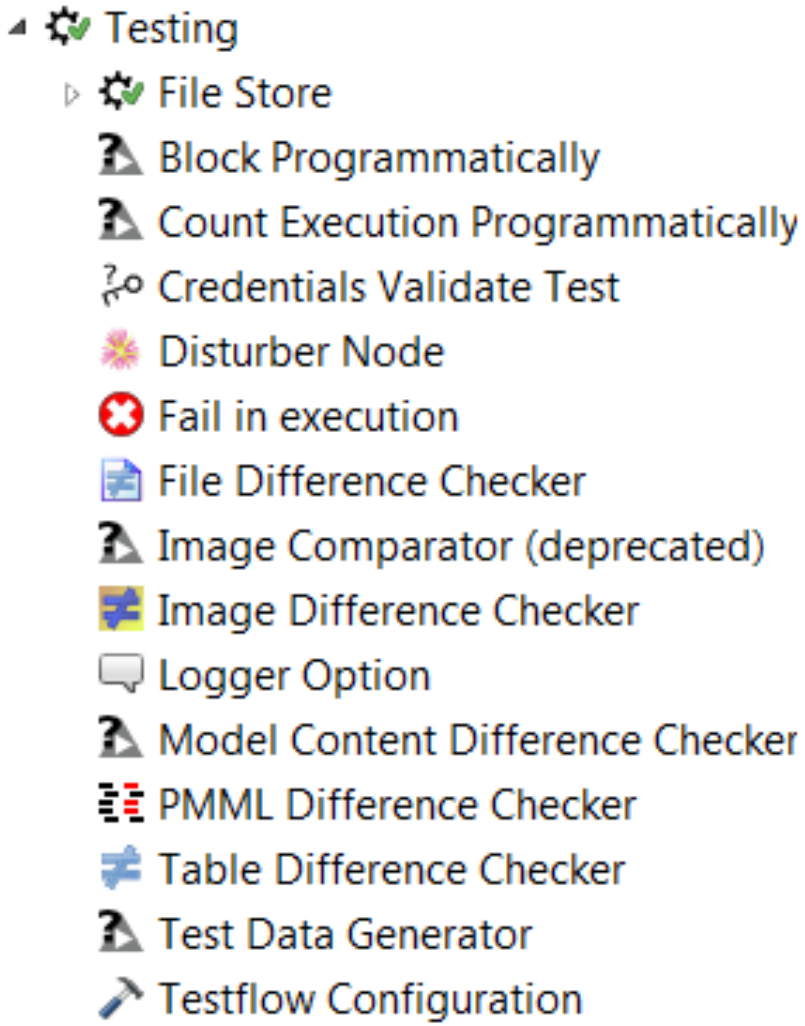
Correctness Tests

- Check if node computes correct results
- Use manually created reference table



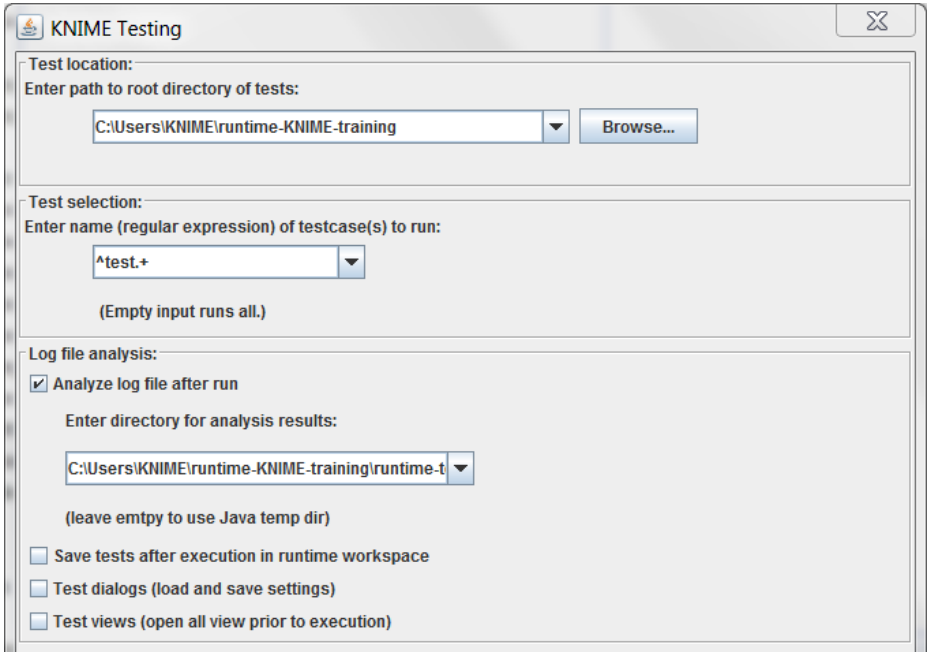
Testing nodes

- Several other useful nodes in the Testing category
  - Install „KNIME Testing Application“



Running Testflows I

- Special Testing application
  - `./knime -application org.knime.testing.KNIME_TESTING_APPLICATION`
  - Or select the application in the run configuration of the IDE



Running Testflows II

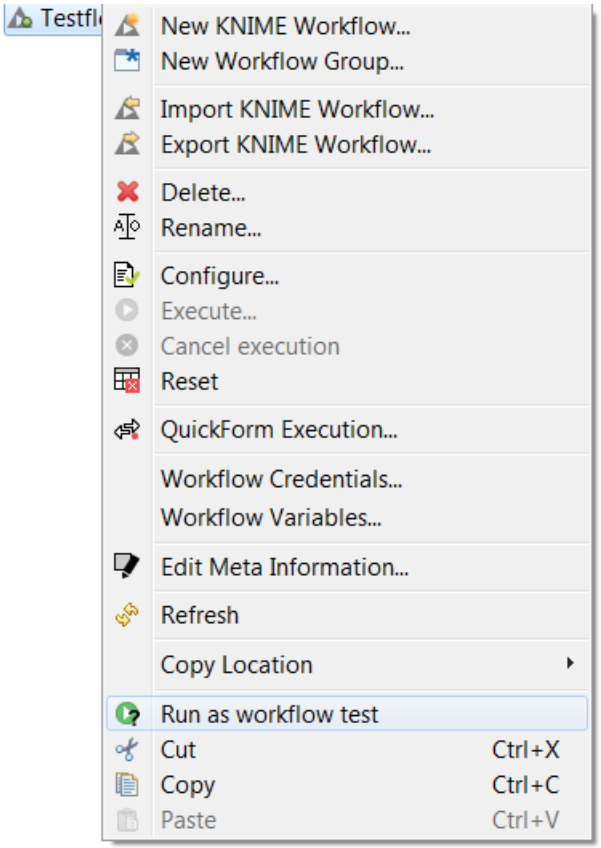
- All matching workflows are executed
- Failures are reported
  - Unexpected non-executed nodes
  - Unexpected error/warning messages
  - Missing expected error/warning messages

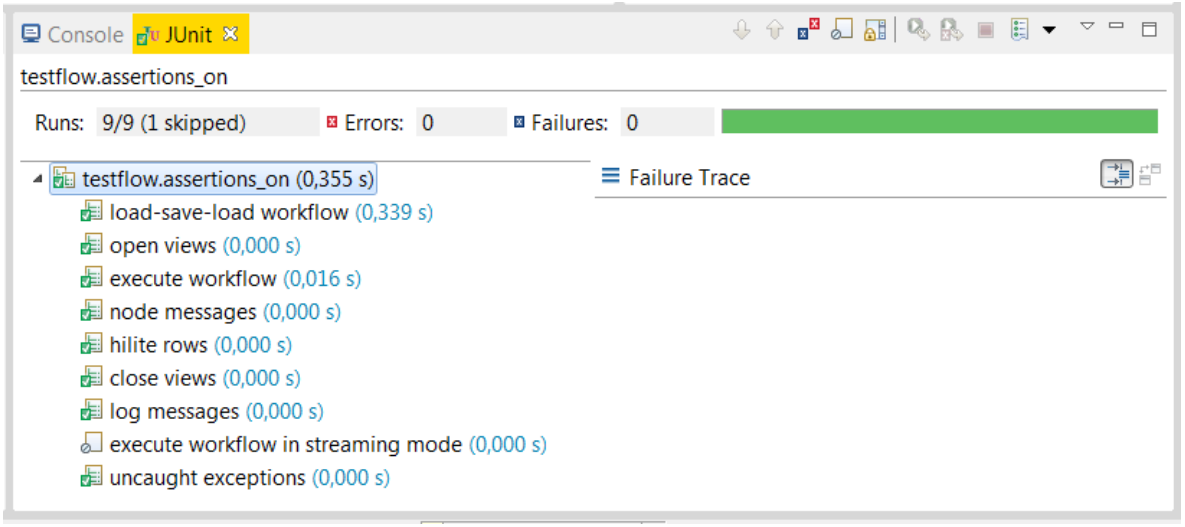
Running Testflows III

```
•Regression run on Oct03_16_10_24
•Tests run: 2, failing: 1, succeeding: 1
•Success rate: 50%

•--------------------------------------------------------------------------------
•--------------------------------------------------------------------------------
•Failing tests: (see individual logs for details)
•--------------------------------------------------------------------------------
•--------------------------------------------------------------------------------
•Test 'testMoSS' failed.(Owner: workflow.owner@company.com).

•--------------------------------------------------------------------------------
•--------------------------------------------------------------------------------
•Succeeding tests: (no individual log exists)
•--------------------------------------------------------------------------------
•--------------------------------------------------------------------------------
•Test 'testNewJoiner' succeeded (Owner: workflow.owner@company.com).



•INFO  main KnimeTestCase : Result ------------------------------------
•INFO  main TestingConfig : Got error: Node Difference Checker 0:0:44 is not executed. (node's
status message: RESET: )
•INFO  main TestingConfig : Got error: Node MoSS 0:0:46 is not executed. (node's status
message: RESET: )
•INFO  main TestingConfig : Got error: Node MoSS 0:0:47 is not executed. (node's status
message: RESET: )
•ERROR main TestingConfig : Unexpected error messages during test run.
•FATAL main TestingConfig : Unexpected error messages -> failing test!
```

Running Testflows IV

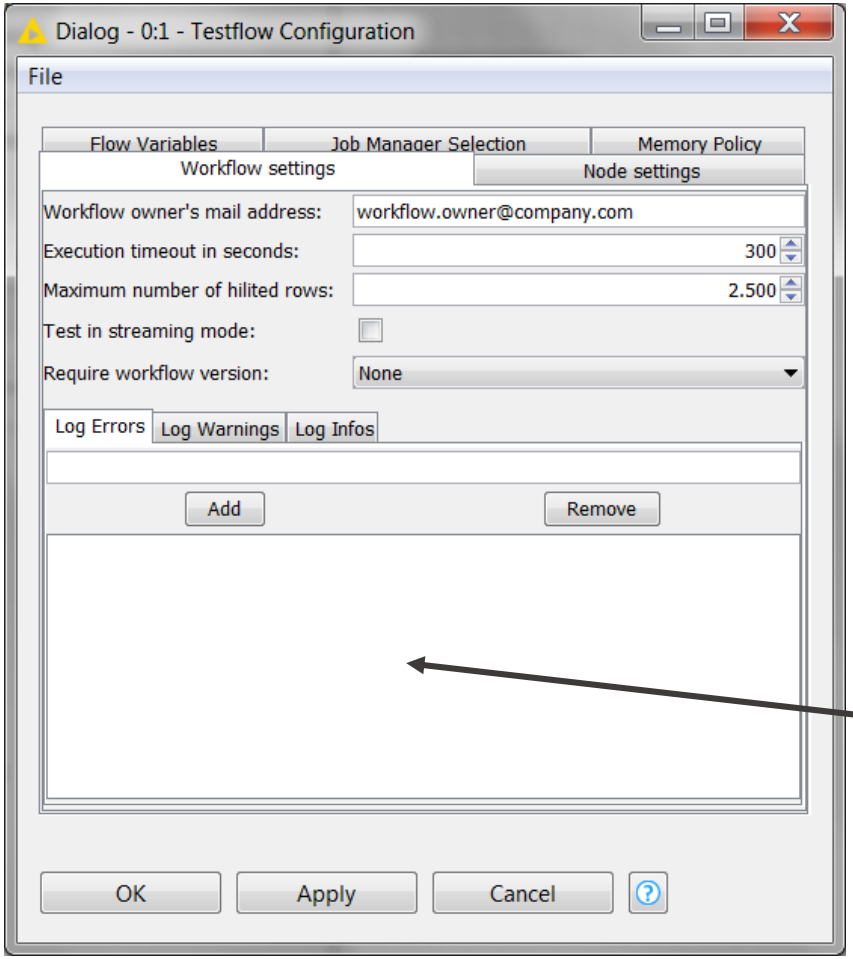
- Context-Menu „Run as workflow test“





Testflow Configuration I

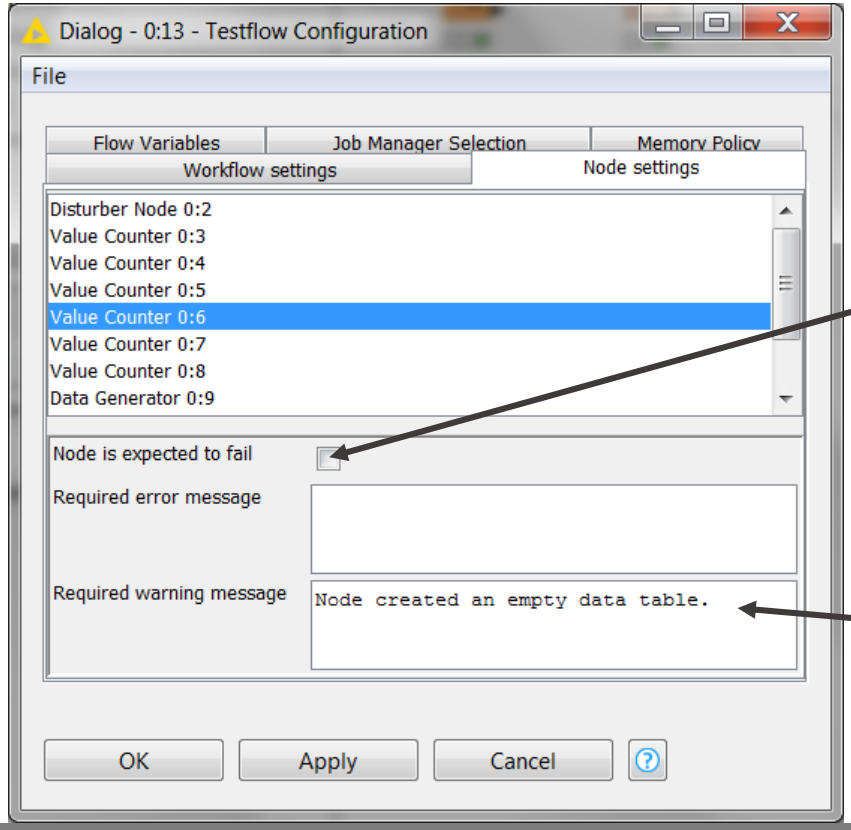
- Via the Testflow Configuration Node



Error/Warning/Info messages that must appear in the log file.

Testflow Configuration II

- Warning/error messages are pre-filled with current values but shown in gray. Click into the field and once the text becomes black the message is used.



Check for nodes that are expected to fail (and enter error message below

Required error/warning messages on a particular node

Automation

- Headless application for running testflows
  - **`./knime -application org.knime.testing.TestflowRunner`**
- Parameters
  - `pattern` – regex for included test workflows
  - `root` – root directory for workflows
  - `server` – URI of a workflow group
  - `analyze` – analyzes the log file afterwards
  - `xmlResult` – creates a JUnit result file suitable for further processing
  - `dialogs` – opens and closes all node dialogs during the test
  - `views` – opens and closes all node views during the test

Exercise #8: Testflow

- Create a testflow for the Concatenate Two Columns node
  - Configure testflow
  - Run workflow with test application
  - Test both with and without remove source columns
  - Test for missing values