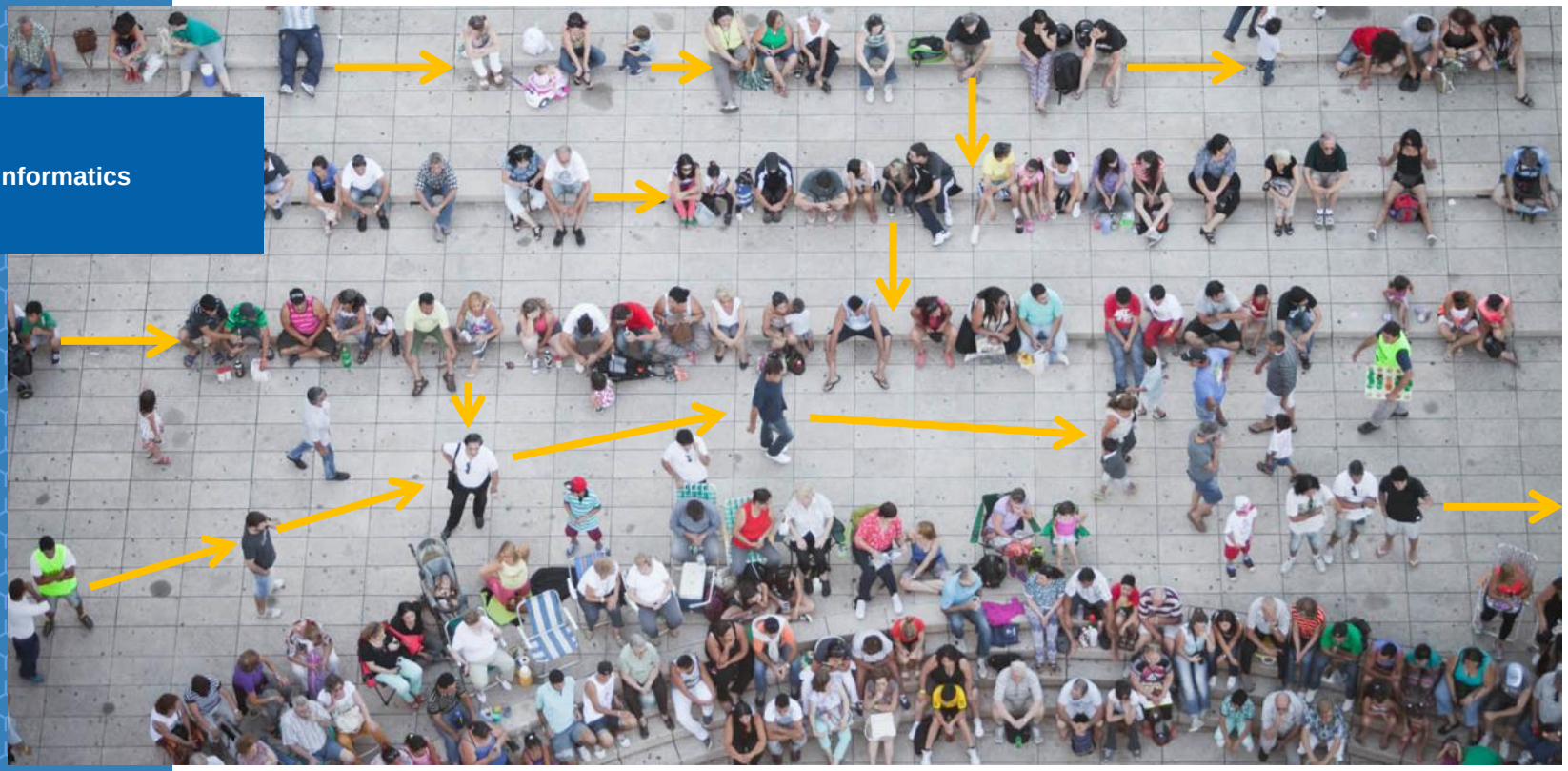
NIBR Informatics

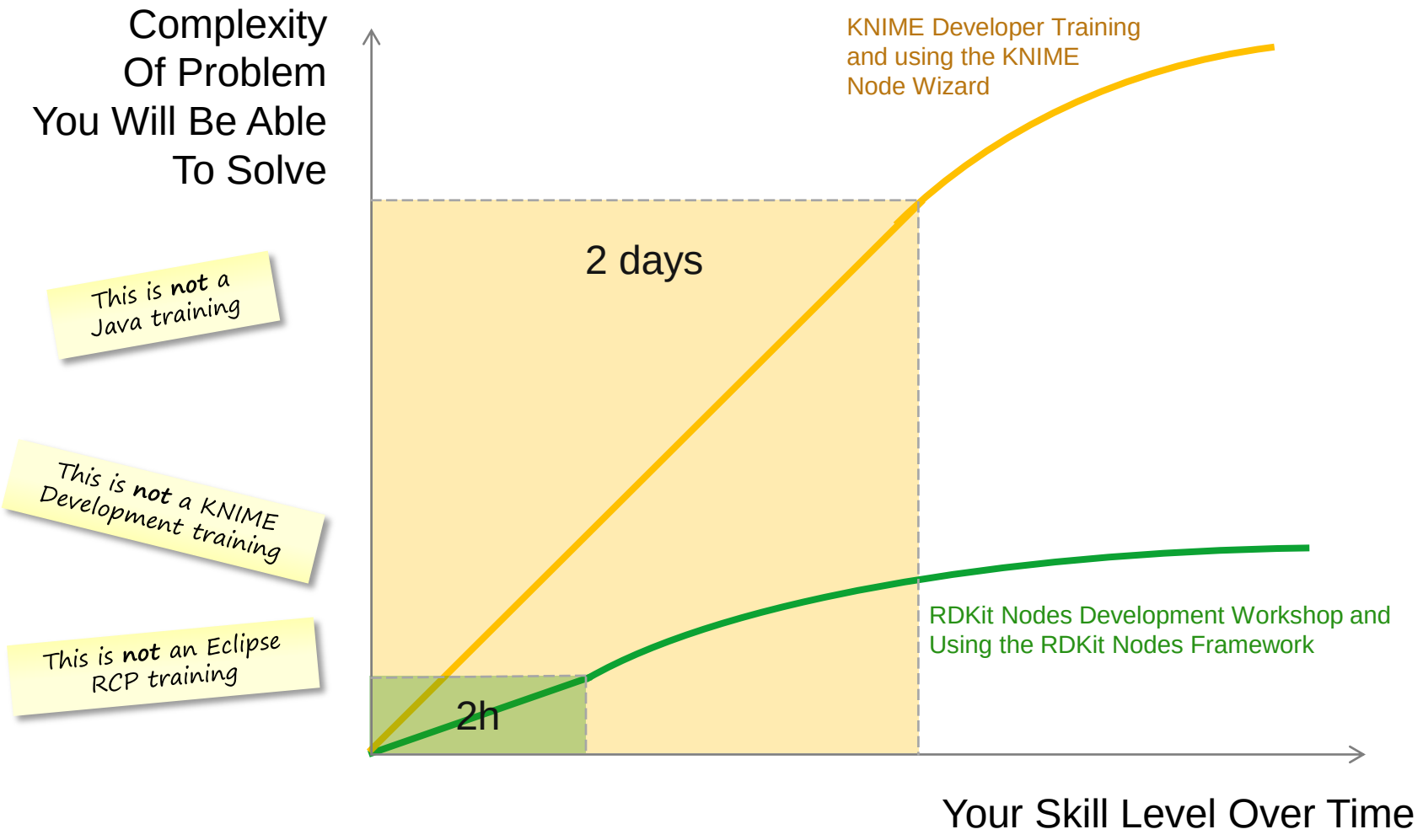


Workshop:
Writing RDKit/KNIME Nodes Hands-On

Manuel Schwarze, Senior Principal Software Engineer
Basel, Switzerland
October 26-28, 2016

NOVARTIS

What to Expect …



Memory Stick Content

- **Downloads** from knime.org and www.eclipse.org/egit/download:
  - **KNIME 3.2 SDK** Versions for Windows, Linux and Mac

    Unzip to your laptop or install on your laptop close to the root folder
  - **KNIME 3.2** Versions for Windows, Linux and Mac (Full versions)

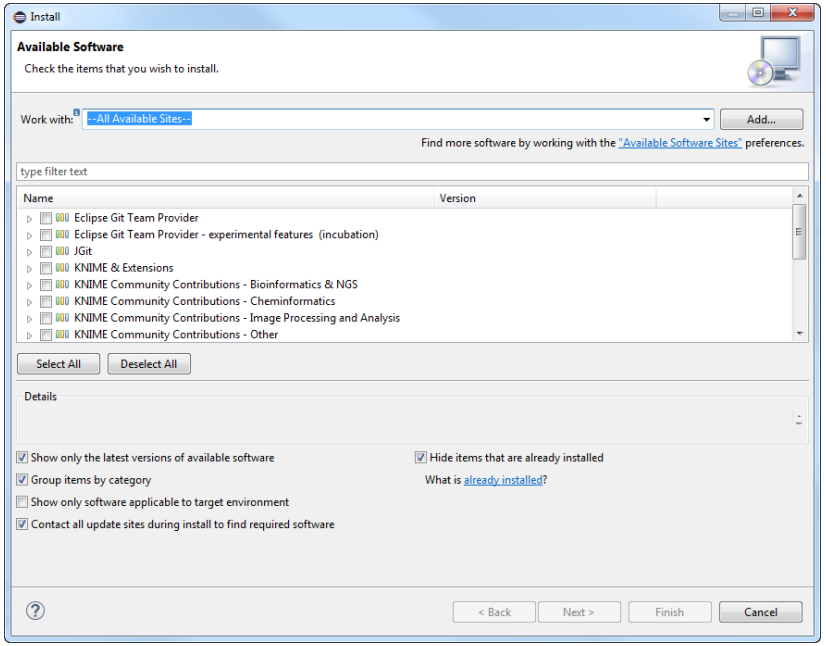
    We won't need those in the workshop
  - **KNIME Update Sites** as ZIP files: (State Oct 25, 2016)

    Register these files as Local Update Sites to install extensions
    - KNIME 3.2 Update Site (KNIME Core things and KNIME Lab)
    - KNIME Community Update Sites (Nightly Build, Trusted Extensions, Untrusted Extensions)
    - eGit Update Site (if you really want to contribute things to git)
- **RDKit Nodes** Source Code from https://github.com/rdkit/knime-rdkit:
  - Current RDKit extensions source code

    Copy and import this code into your KNIME SDK Workspace
- RDKit UGM – KNIME Talk and Workshop **Presentations** (PDFs)
- Workshop **Exercises**:
  - MS Word template for RDKit Node Specification
  - RDKit Node Specifications for workshop exercises and **Solutions** source code

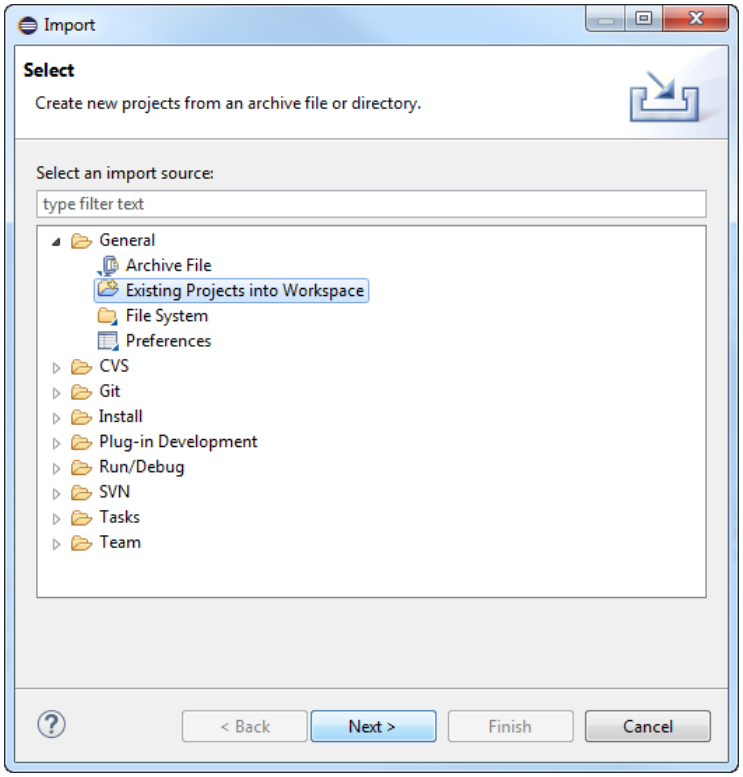
Which Extensions To Install ...

- Menu Help – Install New Software ...
- To add new update site click link «Available Software Sites»
- Back in Installation Dialog click select Work with «-- All Available Sites --» and select extensions to be installed
- Required for RDKit Nodes development:
  - KNIME Base Chemistry Types and Nodes
  - KNIME Chemistry Add-Ons
  - KNIME Python Integration
  - KNIME SVG Support
  - KNIME Streaming Execution (Beta)
  - ChemAxon/InfoCom Marvin Extensions Feature
  - RDKit KNIME Wizards
- Recommended:
  - KNIME Virtual Nodes
  - KNIME JavaScript Views
  - KNIME REST Client Extension
  - KNIME Distance Matrix
  - KNIME File Handling Nodes
  - Associated sources (same name, but in front «Source of»)



Importing All Existing Projects

- Menu File – Import ...
- Select General –
  Existing Projects into Workspace
- Select as Root Directory the knime-rdkit
  folder, which contains all existing projects as
  subfolders
- Select all projects which are found except the *.bin.* binaries projects that do not match your operating system
- Click the «Finish» button to import everything
- Wait until Eclipse has built the complete workspace



Structure of RDKit Extensions

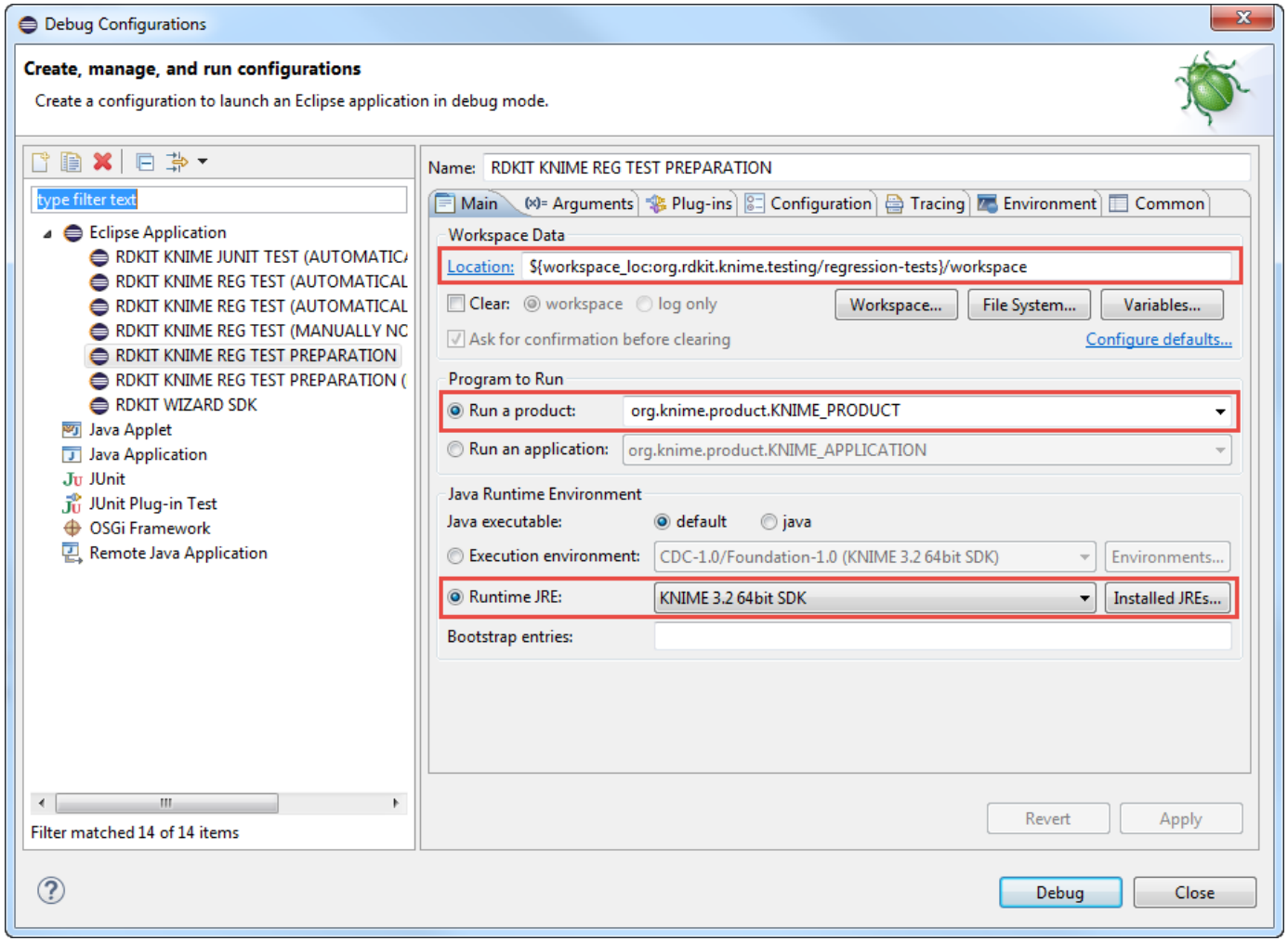
- org.rdkit.knime.bin.<your OS flavor>
  Plugins that contain the RDKit binaries
- org.rdkit.knime.feature
  Defines which plugins shall be contained in it and the version info
- **org.rdkit.knime.nodes**
  Contains all node implementations incl. RDKit nodes framework and util classes
- org.rdkit.knime.testing
  Contains regression test workflows and all logic to test
- org.rdkit.knime.testing.feature
  Defines the testing feature
- **org.rdkit.knime.types**
  Contains all data type and renderer logic as well as RDKit binary invocation and util classes
- org.rdkit.knime.update
  Defines configurations for automated build system
- org.rdkit.knime.wizard
  Contains the RDKit KNIME Wizards logic and node templates
- org.rdkit.knime.wizard.feature
  Defines the wizard feature

Running KNIME Out Of The SDK

- RDKit extensions come with automated tests – there are .launch files in the org.rdkit.knime.testing subproject to
  - Set up a testing workspace with all test workflow
    (RDKIT CLEAN REG WORKSPACE AND UNZIP REGRESSION TESTS, running Ant as External Tools)
  - Start KNIME to create and change tests in the testing workspace
    (RDKIT KNIME REG TEST PREPARATION, running Eclipse Application)
  - Logic to run the tests unattended as the build server would do
    (RDKIT KNIME REG TEST (AUTOMATICALLY NON-GUI), running Eclipse Application)
- How to run stuff out of Eclipse - Possibilities:
  - Right-click on .launch file, then Run As / Debug As ... – XXX
  - *For Eclipse Applications:* Menu Run – Run / Debug Configurations ... to see dialog with all settings before starting the application
  - *For External Tools (e.g. Ant):* Menu Run – External Tools – External Tools Configuration ... to see dialog with all settings before starting the tool

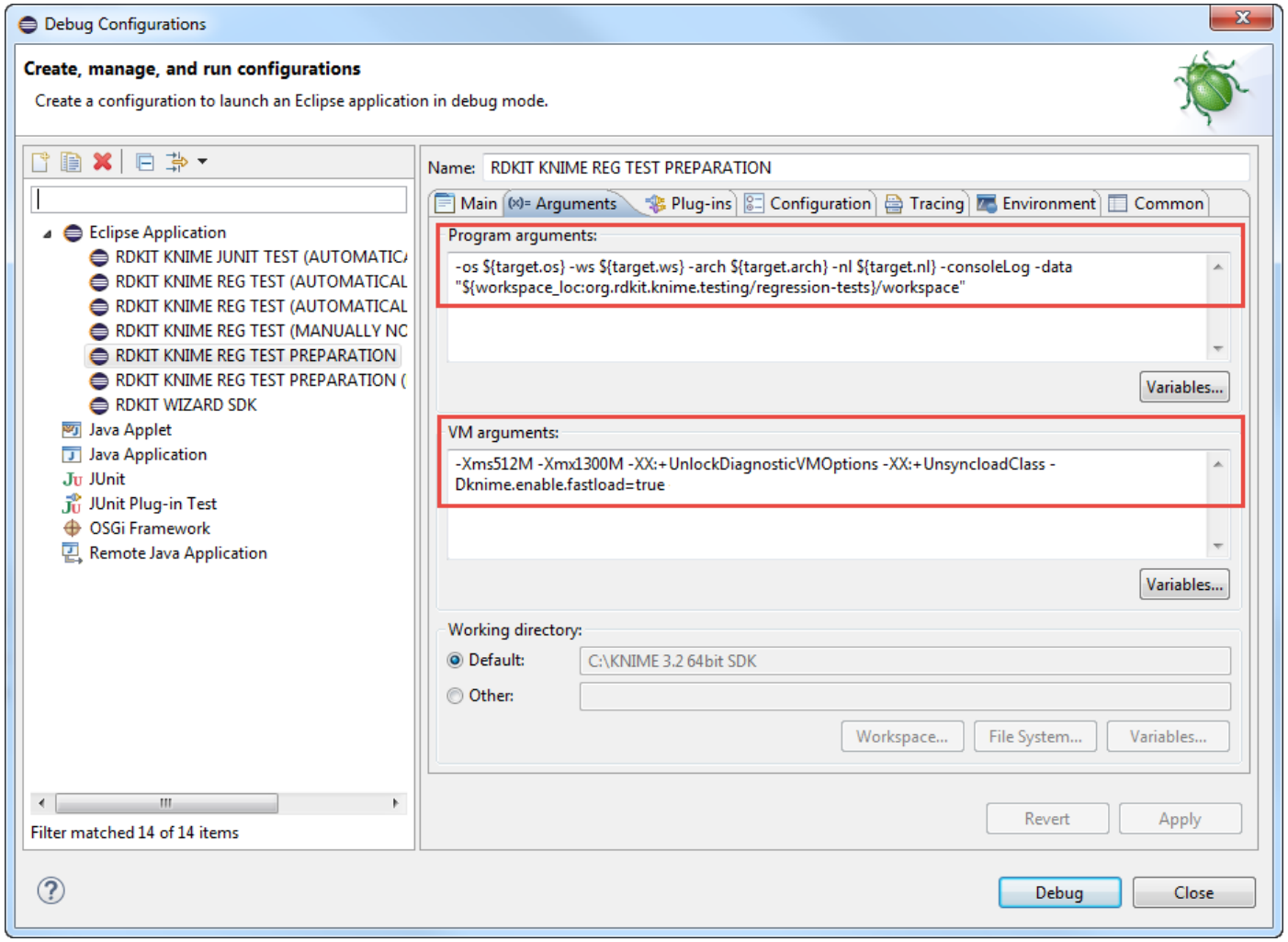
Running KNIME Out Of The SDK

- Necessary settings to run KNIME:



Running KNIME Out Of The SDK

- Necessary settings to run KNIME:



Ideas For New RDKit Nodes

- Jag: BRICS/RECAP integration
- Fab: Similarity map, MIF
- Eric Vangrevelinghe, Novartis: Conformers clustering and alignment with option to perform it on the whole structure or a substructure
- JK: More intuitive enumeration node (compared to reaction nodes) for creating virtual libraries, fragmentation node (MCS based), rebuilding node
- Greg Landrum: Filter catalog node, Adjust query properties node, Fingerprint highlighting
- ...

- Future ideas? Post them to Github - Issues https://github.com/rdkit/knime-rdkit/issues

Selected «Workshop RDKit Nodes»

- Adjust Query Properties Node
  - Create «calculator» node to adjust properties of query molecules to be used later in substructure filter nodes (More information: http://rdkit.blogspot.ch/2016/07/tuning-substructure-queries-ii.html)
- Filter Catalog Node
  - A «filter» or «splitter» node which performs similar things as the demo workflow (More information: Demo workflow, Java testing code)
- For enhusiasts and the hackathon afternoon:
  - Other feasible ideas have been placed in Github issues section (https://github.com/rdkit/knime-rdkit/issues)

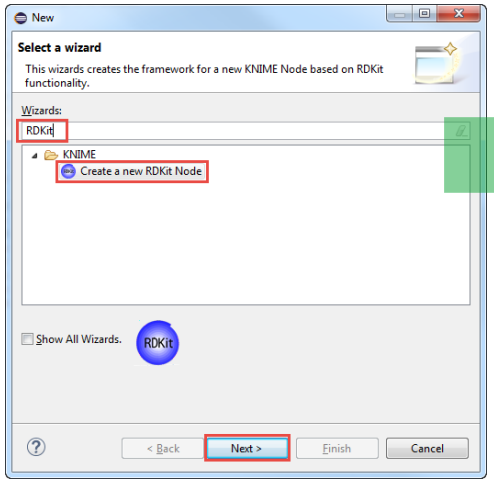
Developing A New Node

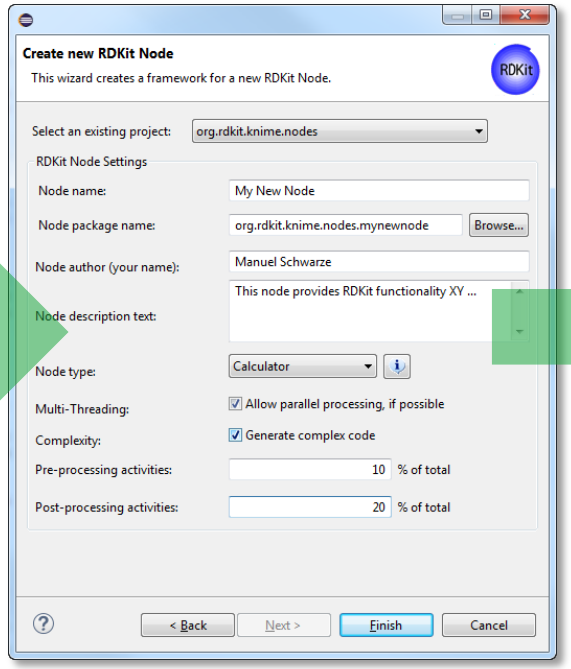
1. Define what the KNIME node shall do
   - Input data (tables, data types)
   - Output data (tables, data types)
   - Processing model (calculator, modifier, filter, splitter)
   - Pre-processing model (what to do)
   - Post-processing model (filter, splitter)
   - Main functionality
   - Node parameters, their data types, default values and constraints (validity conditions)
   - Sophisticated: View design (not part of this workshop)

Developing A New Node

2. Use RDKit KNIME Wizard to auto-generate RDKit Node source code based on the pre-, post and main processing model you need

- Select menu File – New – Other ... and enter «RDKit» to filter wizards
- Click on «Create a new RDKit Node» and the «Next>» button

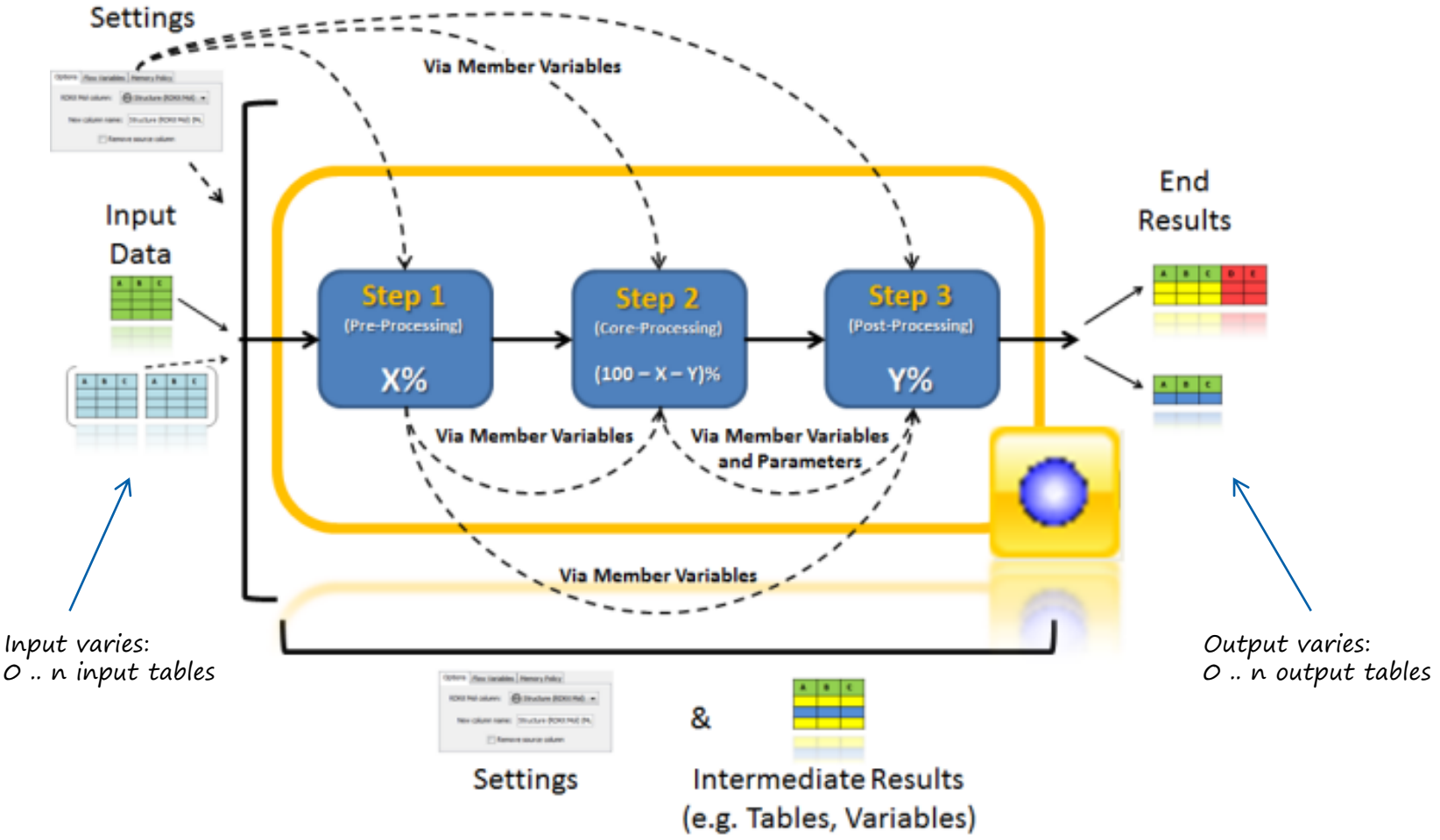




NodeDialog.java
NodeModel.java
NodeFactory.java
NodeFactory.xml
default.png

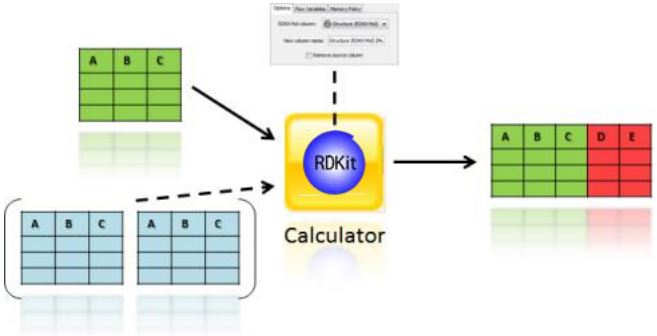
The RDKit KNIME Wizard

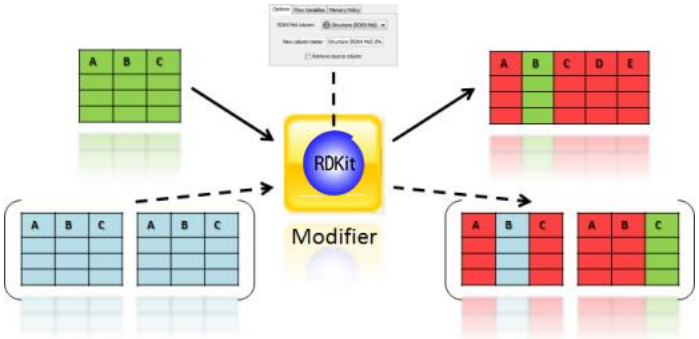
Typical RDKit node execution scheme

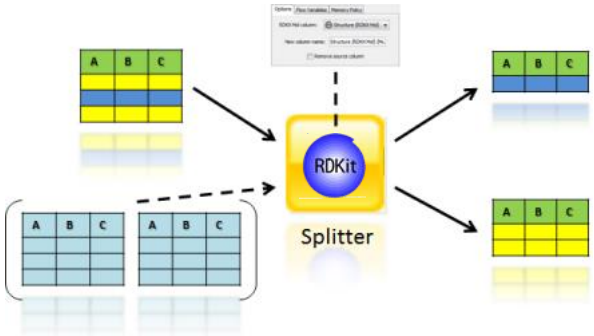


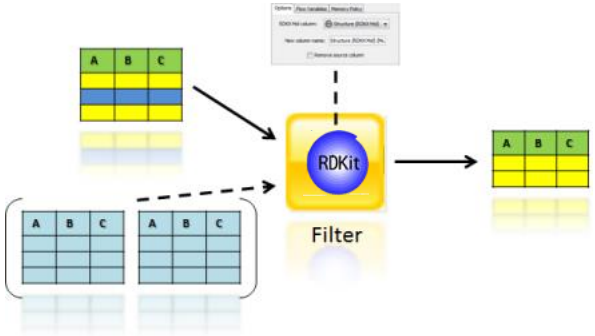
The RDKit KNIME Wizard

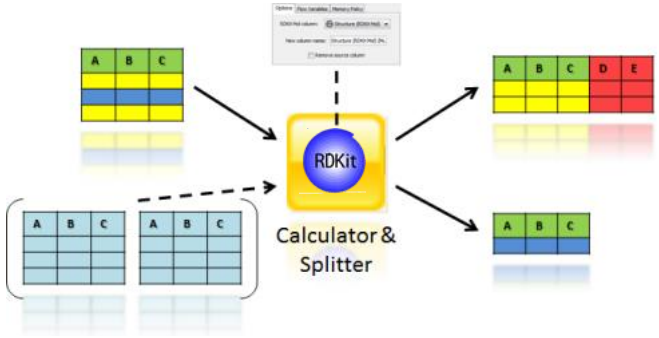
RDKit node types supported by the wizard

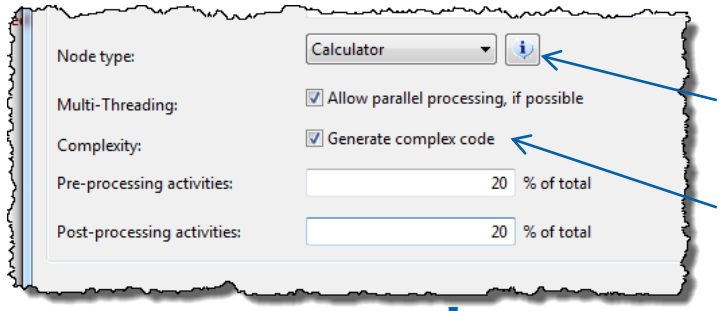












Detailed descriptions of different node types

Enables pre- and post-processing

The RDKit KNIME Wizard

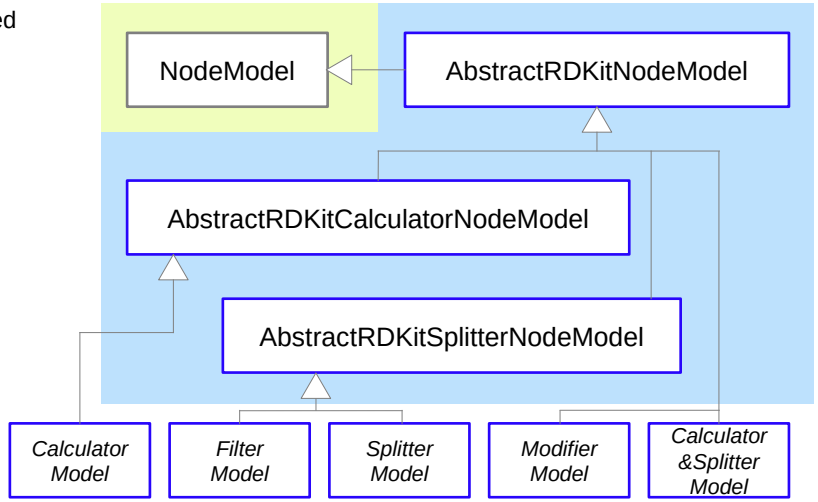
Generate source code and plugin changes

- Creating new Java package
  `org.rdkit.knime.nodes.mynewnode`
  - **`RDKitMyNewNodeDialog.java`**
    - Is derived from super class of RDKit Nodes framework
    - Defines parameters for the node (setting models)
    - Defines graphical user interface to enter parameters (dialog components)
  - **`RDKitMyNewNodeFactory.java`**
    - Defines if there is a dialog to provide node parameters (normally there is one)
    - Defines number of views (normally there are no views)
      Note: The normal output table is not a view
  - **`RDKitMyNewNodeModel.java`**
    - Is derived from a super class of the RDKit Nodes framework that is based on the selected node type
    - Defines number of input and output tables (normally 1:1 or 1:2)
    - Defines input columns that shall be processed for each input table and how they are handled (e.g. empty cells, errors handling)
    - Defines output table specification (columns with their names and data types)
    - Uses parameters defined in the dialog class
    - Defines configuration method used to validate node parameters and provide some dynamic default settings
    - Defines execution logic in output factory - used to calculate new data, usually based on RDKit functionality
  - Look for «TODO»s in the code and follow the instructions

```
/**
 * Creates the settings model to be used for the input column.
 *
 * @return Settings model for input column selection.
 */
static final SettingsModelString createInputColumnNameModel() {
    return new SettingsModelString("input_column", null);
}
```

```
...
super.addDialogComponent(new DialogComponentColumnNameSelection(
    createInputColumnNameModel(), "RDKit Mol column: ", 0,
    RDKitMolValue.class));
...
```

Tipp: Auto-completion is your friend, e.g. try to write «DialogComponent», then hit Ctrl+Space to learn what dialog components are available in KNIME and RDKit Nodes



The RDKit KNIME Wizard

Generate source code and plugin changes

- Further things in Java package `org.rdkit.knime.nodes.mynewnode`
  - **`default.png`**
    - This icon will show up on the node – change it to reflect what the node does
  - **`RDKitMyNewNodeFactory.xml`**
    - Defines node name and help content (description, parameters)
    - Defines input and output ports descriptions of the node (seen as tooltips)
- Adding new node package to **`/META-INF/MANIFEST.MF`**
- Adding new node extension point registration to **`/plugin.xml`**
  - Defines that the new node appears in the RDKit category
  - Defines the sub-category and order of the node relative to others
- Let's explore some generated source code for a calculator node!

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE knimeNode PUBLIC "-//UNIKN//DTD KNIME Node 2.0//EN"
"http://www.knime.org/Node.dtd">
<knimeNode icon="./default.png" type="Manipulator">
    <name>RDKit Add Hs</name>

    <shortDescription>
        Adds hydrogens to an RDKit molecule.
    </shortDescription>

    <fullDescription>
        <intro>Adds hydrogens to an RDKit molecule.
        </intro>
        <option name="RDKit Mol column">The input column with RDKit
         Molecules.</option>
        <option name="New column name">The name of the new column,
         which will contain the calculation results.</option>
        <option name="Remove source column">Set to true to remove
         the specified source column from the result table.</option>
    </fullDescription>

    <ports>
        <inPort index="0" name="Input table with RDKit Molecules">
          Input table with RDKit Molecules</inPort>
        <outPort index="0" name="Output table with RDKit Molecules
          with added Hs">Output table with RDKit Molecules with
          added Hs</outPort>
    </ports>
</knimeNode>
```

AbstractRDKitCellFactory | AbstractRDKitNodeSettingsPane

RDKitMolCell2 | *RDKitMolValue* | RDKitAdapterCell

RDKitReactionCell | *RDKitReactionValue*

InputDataInfo | SettingsUtils

DialogComponentEnum* | SettingsModelEnumeration*

Other useful classes of RDKit Nodes Framework

Developing A New Node

3. Set the category in which the node shall appear
   - Edit your <node> tag in the file org.rdkit.knime.nodes/plugin.xml
4. Run the node (just for fun)
   - It does not do anything meaningful yet
5. Walk through TODOs in the code and make the code meaningful

   Recommendations:

   - *Work first with standard settings (hardcoded) and replace them later with configurable settings*
   - *Learn from existing node implementations (they would share the same super class)*

Main Methods In Node Model Class

RDKitXYZNodeModel.java

- **RDKitXYZNodeModel() - Constructor**
  - Called when the node gets created or loaded in a workflow
- **protected DataTableSpec[] configure(DataTableSpec[] inSpecs)**
  - Called after the node got created and whenever the input connection changes, e.g. another node gets connected or has suddenly data
- **protected InputDataInfo[] createInputDataInfos(int inPort, DataTableSpec inSpec)**
  - Called whenever easy access to input column data or specification is required based on a table specification
- **protected DataTableSpec getOutputTableSpec(int outPort, DataTableSpec[] inSpecs)**
  - Called whenever the output table specification is required, e.g. at the end of the configure(...) method or during node execution
- **protected BufferedDataTable[] processing(BufferedDataTable[] inData, InputDataInfo[][] arrInputDataInfo, ExecutionContext exec)**
  - Called when the node is executed, after pre-processing and before post-processing (hidden in Calculator, Filter and Splitter Node)
- **protected AbstractRDKitCellFactory[] createOutputFactories(int outPort, DataTableSpec inSpec)**
  - Called at the end of configure(...) method and when the node is executed (only used in Calculator Node and Calculator & Splitter Node)

Developing A New Node – TODOs

`RDKitXYZNodeModel.java`

- **`RDKitXYZNodeModel()`** - Constructor
  - Change number of input and output ports in super constructor call, e.g. to `super(2, 1)`
- **`protected static final int INPUT_COLUMN_MOL = 0;`**
  - Constants section (the very first TODO point)
  - Define input data indexes for 0..n data from input table 1, 0..m data from table 2, ...
- **`protected InputDataInfo[] createInputDataInfos(int inPort, DataTableSpec inSpec)`**
  - For each input data index `INPUT_COLUMN_XXX` define here an `InputDataInfo` object with the following information:
    - Table specification object (inSpec that is passed in as parameter to `createInputDataInfos` method)
    - Setting model, that contains the input column model (e.g. m_settingModelInputColumn)
    - Empty cell policy that defines, what should happen, if the input cell is empty: `TreatAsNull`, `UseDefault`, `DeliverEmptyRow`, `StopExecution`, `Custom`
    - Default cell to be used, if the empty cell policy is set to `UseDefault`
    - Array of acceptable KNIME data types (all derived of `DataValue`) (e.g. RDKitMolValue.class, IntegerValue.class, BooleanValue.class, DoubleValue.class)

Developing A New Node – TODOs

`RDKitXYZNodeModel.java`

- **`protected void preProcessing( BufferedDataTable[] inData, InputDataInfo[][] arrInputDataInfo, ExecutionContext exec)`**
  - If pre-processing is required, e.g. reading in a second table with things that are required for processing the main table, do this here
  - Add a member variable (on class level) and store the intermediate data there
  - Return in the method `getPreProcessingPercentage()` the expected percentage to be taken for pre-processing, e.g. 0.1d for 10%
- **`protected void cleanupIntermediateResults()`**
  - Delete all intermediate member variables of last step and set the to null

Developing A New Node – TODOs
RDKitXYZNodeModel.java

- **`protected AbstractRDKitCellFactory[] createOutputFactories(int outPort, DataTableSpec inSpec)`**
  - Code the RDKit functionality of the node in a factory class
  - Some important RDKit Java classes:
    - `ROMol` – Read-only molecule object with native backing and some functionality
    - `RWMol` – Modifyable molecule object with native backing and some functionality
    - `RDKFuncs` – Utility class which offers most functionality
    - `ExplicitBitVect` – Fingerprint object with native backing
    - `XYZ_Vect` – Really just a collection object for native datatype XYZ with list functions
  - Important: Calling `delete()` on created objects disposes native memory.
    If you forget this you will sooner or later run out of memory and crash KNIME!
    - The RDKit Nodes framework does it for you, if you
      - Call `markForCleanup(<RDKit Object>)` if the object shall be disposed at the end of the node execution, e.g. for intermediate results
      - Call `markForCleanup(<RDKit Object>, <UniqueWaveId>)` if the object shall be disposed at the end of a processing unit with a certain "UniqueWaveId", e.g. the `process(...)` call of your factory receives such "UniqueWaveId" for each row that shall be processed, and objects are disposed immediately after the row was fully processed

Developing A New Node – TODOs
RDKitXYZNodeModel.java

- `protected AbstractRDKitCellFactory[] createOutputFactories(int outPort, DataTableSpec inSpec)` **(cont.)**
  - Define the output data types in the array arrOutputSpec, e.g.

```
DataColumnSpec[] arrOutputSpec = new DataColumnSpec[2]; // For 2 columns

arrOutputSpec[0] = new DataColumnSpecCreator(m_modelNewColumnName.getStringValue(),
        RDKitAdapterCell.RAW_TYPE).createSpec();
arrOutputSpec[1] = new DataColumnSpecCreator(m_modelAnotherNewColumnName.getStringValue(),
        IntCell.TYPE).createSpec();
```

    - Note: For molecule cells always define XYZAdapterCell.RAW_TYPE, for non-molecule cells usually XYZCell.TYPE which points to the correct data type definition
  - At the end generate as many output cells as your node defines above and return them as array, e.g.

```
DataCell outputCell1 = RDKitMolCellFactory.createRDKitAdapterCell(molecule);
DataCell outputCell2 = new IntCell(anIntegerNumber);
return new DataCell[] { outputCell1, outputCell2 };
```

  - To return an empty cell use as cell value `DataType.getMissingCell()`
  - Review, if parallel processing is possible (just a boolean to set)

- **`protected BufferedDataTable[] postProcessing(BufferedDataTable[] inData, InputDataInfo[][] arrInputDataInfo, BufferedDataTable[] processingResult, ExecutionContext exec)`**
  - If post-processing is required, e.g. splitting the table up, adding another column with information that required access to all other results first, do this here
  - Note: Skills of the KNIME Developer Training are recommended
  - Return in the method `getPostProcessingPercentage()` the expected percentage to be taken for post-processing, e.g. 0.2d for 20%

Developing A New Node – TODOs
RDKitXYZNodeModel.java

- `protected AbstractRDKitCellFactory[] createOutputFactories(int outPort, DataTableSpec inSpec)` **(cont.)**
  - Define the output data types in the array arrOutputSpec, e.g.

```
DataColumnSpec[] arrOutputSpec = new DataColumnSpec[2]; // For 2 columns

arrOutputSpec[0] = new DataColumnSpecCreator(m_modelNewColumnName.getStringValue(),
        RDKitAdapterCell.RAW_TYPE).createSpec();
arrOutputSpec[1] = new DataColumnSpecCreator(m_modelAnotherNewColumnName.getStringValue(),
        IntCell.TYPE).createSpec();
```

    - Note: For molecule cells always define XYZAdapterCell.RAW_TYPE, for non-molecule cells usually XYZCell.TYPE which points to the correct data type definition
  - At the end generate as many output cells as your node defines above and return them as array, e.g.

```
DataCell outputCell1 = RDKitMolCellFactory.createRDKitAdapterCell(molecule);
DataCell outputCell2 = new IntCell(anIntegerNumber);
return new DataCell[] { outputCell1, outputCell2 };
```

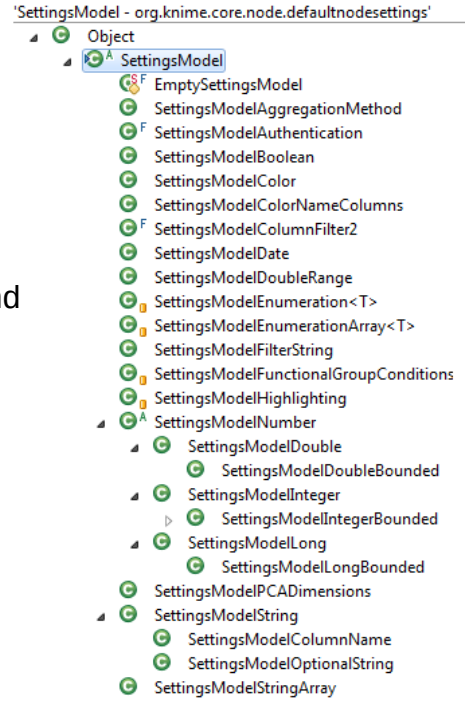
  - To return an empty cell use as cell value `DataType.getMissingCell()`
  - Review, if parallel processing is possible (just a boolean to set)

- **`protected BufferedDataTable[] postProcessing(BufferedDataTable[] inData, InputDataInfo[][] arrInputDataInfo, BufferedDataTable[] processingResult, ExecutionContext exec)`**
  - If post-processing is required, e.g. splitting the table up, adding another column with information that required access to all other results first, do this here
  - Note: Skills of the KNIME Developer Training are recommended
  - Return in the method `getPostProcessingPercentage()` the expected percentage to be taken for post-processing, e.g. 0.2d for 20%

Developing A New Node – TODOs

RDKitXYZNodeDialog.java

- ```
  static final SettingsModelXYZ
  createMyCustomSettingsModel() {
        return new SettingsModelXYZ(
              <Parameters>,
              <Default Value>);
  }
  ```
  - The abstract KNIME class SettingsModel acts as a wrapper around the setting and has many specific implementations
  - A setting gets at least a name and a default value
  - The SettingsModel implements all logic to validate, save and load setting information to/from node config
  - Define for each setting a separate static method – this method gets called from both, Dialog AND Model class

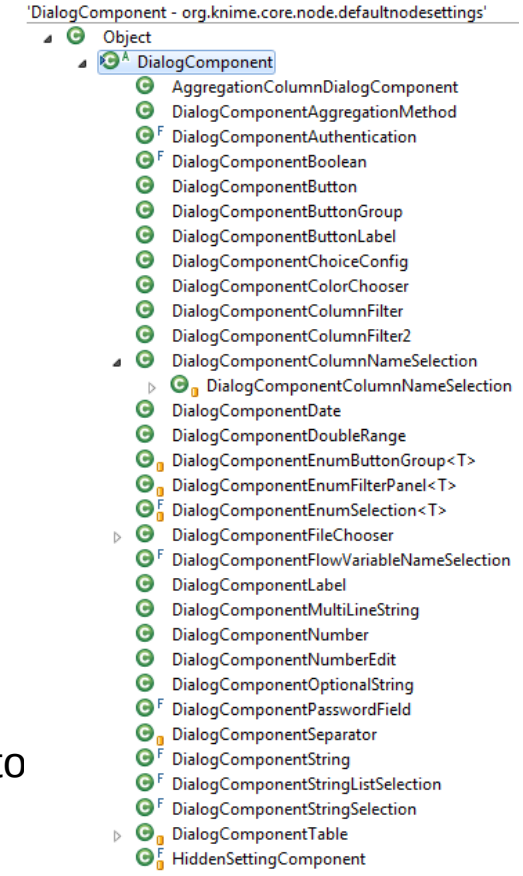

  ```

Developing A New Node – TODOs
RDKitXYZNodeDialog.java

- **RDKitXYZNodeDialog()** - Constructor
  - Add for each new setting a GUI component, a DialogComponent compatible with your SettingModel class

```
super.addDialogComponent(new
DialogComponentXYZ(
        createMyCustomSettingsModel(),
        <Parameters>));
```

  - Parameters vary, some examples:
    - Label name
    - Width of text field
    - Acceptable column data types for a column selection
    - Step size for a numeric spinner component
  - Hand-over an instance of your SettingsModel to load and sto the setting (`createMyCustomSettingsModel()`)



Developing A New Node – TODOs
`RDKitXYZNodeModel.java`

- **`RDKitXYZNodeModel`** - Member variables section
  - Add for each new setting a member variable and register it

```
private final SettingsModelXYZ
      m_modelMyCustomSetting = registerSettings(
      RDKitXYZNodeDialog.createMyCustomSettingsModel());
```

  - Registration ensures that the new setting is saved and loaded properly
  - Optional second parameter: Set to true to ignore non-existence of this setting (important for settings that were added later)
- **`protected DataTableSpec[] configure(DataTableSpec[] inSpecs)`**
  - This method gets called to validate current node configuration
  - Add checks here for your new settings and throw InvalidSettingsException, if the setting's value is invalid
  - Pre-fill an «empty» setting with meaningful value, e.g. based on input table specification, e.g. input column name auto-guessing
  - SettingsUtil class of RDKit framework offers helper methods, e.g.
    - autoGuessColumn(...)
    - checkColumnExistence(...)
    - checkColumnNameUniqueness(...)
    - makeColumnNameUnique(...)
    - createMergedColumnNameList(...)
    - getEnumValueFromString(...)
  - Collect warnings: `getWarningConsolidator().saveWarning("A warning for the user");`

Developing A New Node – TODOs

RDKitXYZNodeModel.java

Advanced

- `protected ColumnRearranger createColumnRearranger(`
  `int outPort, DataTableSpec inSpec)`
  - The ColumnRearranger is specialized in changing table structures without replicating a table in memory
  - Add functionality to modify the output table further

Developing A New Node

6. Document your node
   – Edit the file **`RDKitXYZNodeFactory.xml`**

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE knimeNode PUBLIC "-//UNIKN//DTD KNIME Node 2.0//EN" "http://www.knime.org/Node.dtd">
<knimeNode icon="./default.png" type="Manipulator">
    <name>RDKit My Calculator</name>

    <shortDescription>
        Just a test
    </shortDescription>

    <fullDescription>
        <intro>Just a test

        Insert long description here...

        </intro>
        <option name="RDKit Mol column">The input column with RDKit Molecules.</option>
        <option name="New column name">The name of the new column, which will contain the calculation results.</option>
        <option name="Remove source column">Set to true to remove the specified source column from the result table.</option>
    </fullDescription>

    <ports>
        <inPort index="0" name="Input table with RDKit Molecules">Description of first input port...</inPort>
        <!-- possibly more input ports here-->
        <outPort index="0" name="Result table">Description of first output port...</outPort>
        <!-- possibly more output ports here-->
    </ports>
</knimeNode>
```

7. Change the icon for the node
   – Edit the file **`default.png`**, e.g. with Gimp

Developing A New Node

8. Write a regression test
   - If not done yet before, run the launch file org.rdkit.knime.testing/RDKIT CLEAN REG WORKSPACE AND UNZIP REGRESSION TESTS.launch – Warning: This will remove all regression workflows created since the last run of that file!!!
   - Run the launch file org.rdkit.knime.testing/RDKIT KNIME REG TEST PREPARATION.launch to startup KNIME with the new node
   - Create a new Workflow Group «Tests for RDKit My New Node»
   - Create a new KNIME Workflow «My New Node – Default Tests» (the name must contain the word «Tests»)
   - Write your testing workflow and place a «Testflow Configuration» node inside and configure it
   - All data that the test workflow uses should be self-contained:
     - No local file references!
     - Use either knime://
   - To test your workflow: Right-click on the workflow and select «Run as workflow test»
   - If successful, export the Workflow Group to org.rdkit.knime.testing/regression-tests/zips (since KNIME 3.2 you need to rename it afterwards from .knar extension to .zip)

Difference For Filter Node - TODOs

RDKitXYZNodeModel.java

5. Another TODO item:

- **`public int determineTargetTable(int iInPort, long iRowIndex, DataRow row, InputDataInfo[] arrInputDataInfo, long lUniqueWaveId)`**
  - Code the condition to determine by the return value if the row shall be filtered out (return -1) or not (return 0)
  - Use `arrInputDataInfo[INPUT_COLUMN_XYZ].getXXX(row)` method to get the cell object for the passed in row object
  - Use lUniqueWaveId when working with RDKit objects to mark them for memory cleanup
  - Example:

```
/**
 * This implementation filters out all rows with missing cells and molecules with 0 atoms.
 *
 * @return 0 to keep the row, or -1 if row shall be filtered out completely.
 */
  @Override
  public int determineTargetTable(int iInPort, long iRowIndex, DataRow row, InputDataInfo[] arrInputDataInfo,
      long lUniqueWaveId) throws InputDataInfo.EmptyCellException {
          boolean bRemove = true;

          // Check the input cell for the condition
          ROMol mol = markForCleanup(arrInputDataInfo[INPUT_COLUMN_MOL].getROMol(row), lUniqueWaveId);
          if (mol != null) {
              RWMol temp = markForCleanup(new RWMol(mol), lUniqueWaveId);
              if (temp.getNumAtoms() > 0) {
                  bRemove = false;
              }
          }

          // Determine target table port (-1 is the trash bin)
          return (bRemove ? -1 : 0);
  }
```

Difference For Splitter Node - TODOs

RDKitXYZNodeModel.java

5. Another TODO item:

- **`public int determineTargetTable(int iInPort, long iRowIndex, DataRow row, InputDataInfo[] arrInputDataInfo, long lUniqueWaveId)`**
  - Code the condition to determine by the return value in which table the row shall be placed (0..n)
  - Use `arrInputDataInfo[INPUT_COLUMN_XYZ].getXXX(row)` method to get the cell object for the passed in row object
  - Use lUniqueWaveId when working with RDKit objects to mark them for memory cleanup

Difference For Calculator & Splitter Node - TODOs

RDKitXYZNodeModel.java

5. Another TODO item:

- **`protected BufferedDataTable[] processing(BufferedDataTable[] inData, InputDataInfo[][] arrInputDataInfo, ExecutionContext exec) –`**
  `public void processResults(long rowIndex, DataRow row, DataCell[] arrResults)` [Overwritten method of ResultProcessor]
  - Code the condition here and add the calculated results to the correct table (see sample code in generated class)

Difference For Modifier Node - TODOs

RDKitXYZNodeModel.java

5. Another major TODO item:

- **`protected BufferedDataTable[] processing(BufferedDataTable[] inData, InputDataInfo[][] arrInputDataInfo, ExecutionContext exec)`**
  - The main processing has to be done in that method – Everything that is «hidden» in the other RDKit node types is now visible here and needs to be used like
    - Output table creation
    - Input table iteration
    - Generating «UniqueWaveIds» for RDKit object cleanup registration
    - Cleaning up of RDKit objects after each row has been processed
    - Reporting progress back to KNIME
    - Finish output table
  - Most of the generated sample code can be used as is, just the «heart» needs to be replaced
  - Output is one or more new tables with data

Thank you –

Any Questions?

NOVARTIS

Happy Coding!

NOVARTIS

Backup Slides

NOVARTIS

How To: Setup

- Downloaded OS-specific SDK (64-bit) from https://www.knime.org/downloads/overview
- Run the installer and install in C:\RDKitKnimeWorkshop\KNIME_SDK_3.2
- Use as workspace a new folder C:\RDKitKnimeWorkshop\Workspace
- Create a subfolder org.rdkit in the workspace folder
- Close Welcome Screen
- Install other plugins: Menu Help – Install New Software ...
  - Click on «Available Update Sites» link and add the following sites:
    - For KNIME Core / Labs plugins: http://update.knime.org/analytics-platform/3.2/
    - For KNIME Community Nightly Builds: http://update.knime.org/community-contributions/trunk
    - For Subclipse SVN integration: http://subclipse.tigris.org/update_1.8.x

How To: Setup

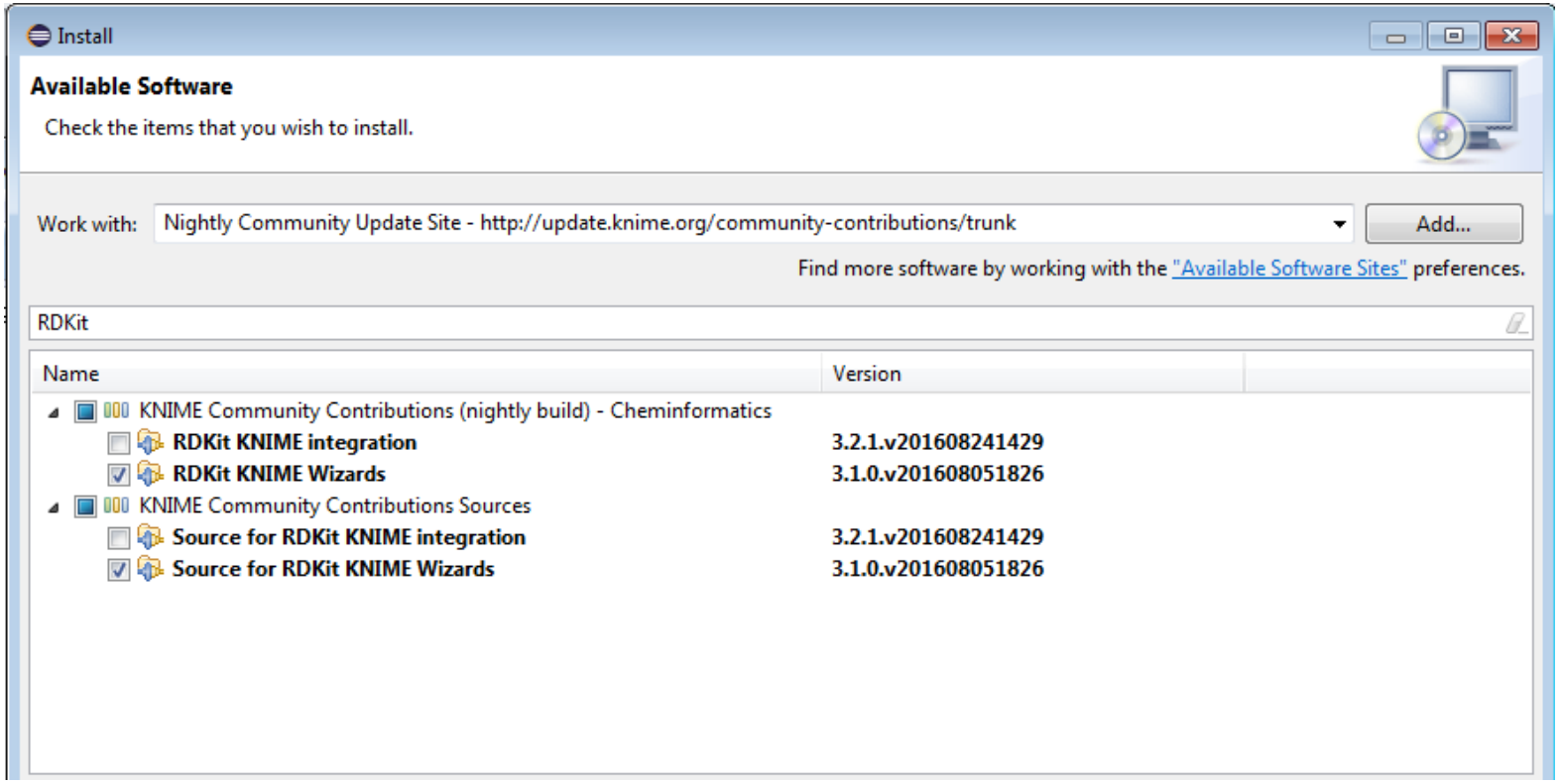
- Install plugins that the RDKit Types and Nodes plugin **depends on** and for test workflows
- Install RDKit plugins from Nightly Community Update Site to get the «RDKit KNIME Wizards» plugin (http://update.knime.org/community-contributions/trunk/)
- Install SVN or Github client with internet access
- Checkout the RDKit community extension source code from SVN or Github into the workspace folder org.rdkit (https://community.knime.org/svn/nodes4knime/trunk/org.rdkit/, https://github.com/rdkit/knime-rdkit)
- The org.rdkit folder contains now multiple projects to be imported to the KNIME SDK as projects:
  - org.rdkit.knime.bin.<your OS flavor>
  - org.rdkit.knime.feature
  - **org.rdkit.knime.nodes** ← This is the project we will work on
  - org.rdkit.knime.testing
  - org.rdkit.knime.testing.feature
  - **org.rdkit.knime.types**
  - org.rdkit.knime.update
  - org.rdkit.knime.wizard
  - org.rdkit.knime.wizard.feature

Dependencies to install:

- *KNIME Base Chemistry Types and Nodes*
- *KNIME Chemistry Add-Ons*
- *KNIME Python Integration*
- *KNIME SVG Support*
- *KNIME Streaming Execution (Beta)*
- *ChemAxon/InfoCom Marvin Extensions Feature*
- *Recommended:*
  - *KNIME Virtual Nodes*
  - *KNIME JavaScript Views*
  - *KNIME REST Client Extension*
  - *KNIME Distance Matrix*
  - *KNIME File Handling Nodes*
- *Associated sources*

How To: Setup

- RDKit Wizard – Getting it from the KNIME update site:



How To: Setup

- KNIME plugins that RDKit nodes depend on and which are useful for testing workflows
- Include source plugins, which are good for debugging
- You can always uninstall or update installed plugins and of course any time add new plugins