jInfer ProjectType Module Description

Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, Matej Vitásek
Advisors: RNDr. Irena Mlýnková, Ph.D., Martin Nečaský, Ph.D.

Praha, 2011

Target audience: developers willing to extend jInfer, specifically extend jInfer project structure.

Responsible developer:	Michal Švirec
Required tokens:	cz.cuni.mff.ksi.jinfer.base.interfaces.inference.IGGenerator cz.cuni.mff.ksi.jinfer.base.interfaces.inference.SchemaGenerator cz.cuni.mff.ksi.jinfer.base.interfaces.inference.Simplifier org.openide.windows.IOProvider
Provided tokens:	none
Module dependencies:	Base Runner
Public packages:	cz.cuni.mff.ksi.jinfer.projecttype.actions

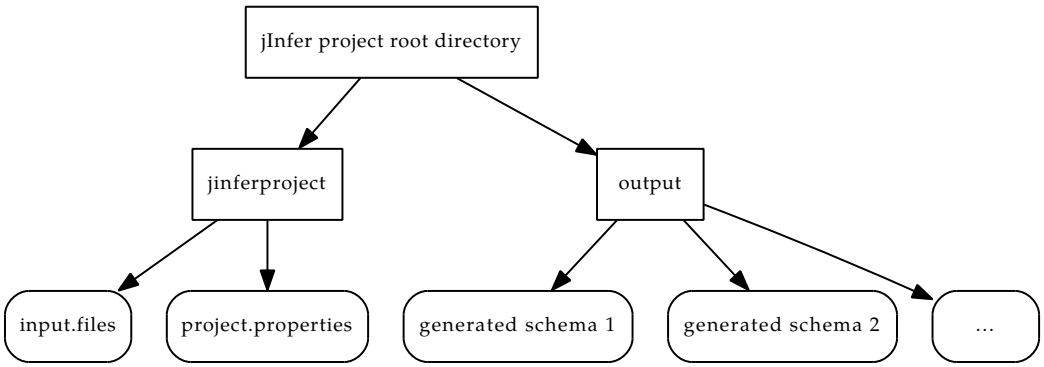
1 Introduction

ProjectType is the module representing a jInfer project in the NB platform, which groups input/output files that belongs to one specific inference. Each jInfer project also allows user to set properties specific for a inference.

With each project created in jInfer is also created specific directory structure on file-system that describes jInfer project. This structure is described in figure 1. Folder in file-system is considered as jInfer project folder, if contains *jinferproject* sub-folder. This folder contains two files, first is *project.properties*, where all properties set for particular project are saved. Second file is *input.files* which is XML file filled with paths to files inserted as input into particular project (structure of this file is described in section 2.1.3). *Output* sub-folder contains schema files which were generated by jInfer inference.

2 Structure

Structure of *ProjectType* can be divided into following five main parts.



(a) Legend: Rectangle is folder, Rounded rectangle is file

Figure 1: Project directory structure

- Base classes - Classes providing main functionality like creation of project, defining operations like move, delete, copy etc. All the base classes are contained in the `cz.cuni.mff.ksi.jinfer.projecttype` package.
- Visualization classes - These classes create tree structure of project in NBP Projects view and are contained in the `cz.cuni.mff.ksi.jinfer.projecttype.nodes`.
- Actions - Classes from `cz.cuni.mff.ksi.jinfer.projecttype.actions` package that provides actions allowing adding input files into project, removing input files, or running the project.
- Properties - Classes responsible for creation of project properties window with properties category tree. These are situated in `cz.cuni.mff.ksi.jinfer.projecttype.properties` package.
- Project wizard - Classes representing creation of project through new project wizard from `cz.cuni.mff.ksi.jinfer.projecttype.sample` package.

2.1 Base classes

This section describes classes needed for creation of the jInfer project and correct integration into NBP. Main two classes representing jInfer project type are `JInferProjectFactory` and `JInferProject`.

2.1.1 JInferProjectFactory

`JInferProjectFactory` is a factory class implementing `ProjectFactory` interface provided by NBP. This class is responsible for loading/saving of jInfer project from/to file-system and also determining if folder in file-system is type of jInfer project (contains *jinferproject* sub-folder). For this purpose, class has three methods: `isProject()`, `save()` and `load()`. When jInfer project is loaded from file-system, `JInferProjectFactory` creates new instance of `JInferProject` passing path to project folder as a constructor parameter. When `save()` method is invoked, all the project properties are saved in *project.properties* file and paths to input files are saved into *input.files*.

2.1.2 JInferProject

`JinferProject` class implements `Project` interface from NBP and represents in-memory representation of jInfer project. Inner structure of this class is very simple, interface provides only two methods: `getProjectDirectory()` and `getLookup()`. All the extensibility of project type is done by *Lookup* functionality of NBP. Project lookup contains besides properties and `Input` class, which encapsulates input files, also class that creates output files, class that builds project tree in project window, etc.

2.1.3 InputFiles class

`InputFiles` is utility class that stores/loads input file paths into/from *input.files* XML file. For this purpose, class uses *JAXB* [JAX] architecture to unmarshall/marshall XML file into/from java content objects. XML structure of the input.files is very simple, *jinferinput* is root element under which are *documents*, *schemas* and *queries* elements. Each of this elements could have none or more *file* elements with string attribute *loc* that contains absolute path to input file.

Example of the structure follows.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jinferinput>
    <xml>
        <file loc="/home/user/example.xml"/>
        <file loc="/home/user/example2.xml"/>
    </xml>
    <schemas>
        <file loc="/home/user/example.dtd"/>
    </schemas>
    </queries/>
</jinferinput>
```

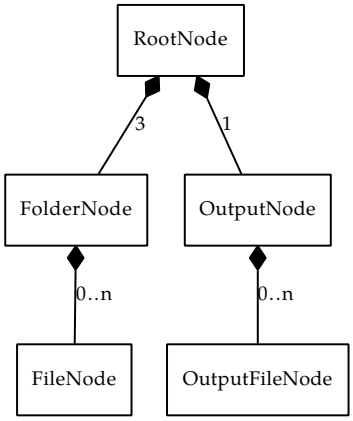


Figure 2: Project directory structure

2.2 Visualization classes

Each jInfer project type in NBP is after creation opened in project window and has some visual tree structure where each node in this tree is represented by NBP `Node` class. This structure is in jInfer project type provided by `JInferLogicalView` class that implements `LogicalViewProvider` interface from NBP and is available through `JInferProject` lookup. This class has one important method, `createLogicalView()` that returns instance of `RootNode` class of root node. Simple class structure is described in figure 2.

`RootNode` extends NBP `AbstractNode` class which is basic implementation of NBP `Node`. This node contains four children nodes: three for input files (Document, Schema and Query) and one for output files generated by inference (Output). Root node also offers actions for adding input files, run inference and standard project actions like delete or move/rename project.

Each of the *input* nodes is represented by `FolderNode` class and contains *input file* nodes (represented by `FileNode` class) representing files added to the jInfer project. Notice that these *input file* nodes represents files from file-system and are not copied to the project folder structure. Because of this, if you delete this nodes from project tree, files from file-system are not deleted.

`OutputNode` class represents *Output* node in jInfer project type tree and is slightly different from *input* nodes, because it refers to *output* sub-folder in project file-system hierarchy. In other words, files contained in this folder are shown as sub-nodes in *Output* node in project tree. This sub-nodes are represented by `OutputFileNode` class. Unlike *input file* nodes, if you delete *output file* node under the *Output* node, file represented by this node is also deleted from file-system.

2.3 Actions

Each node in project tree hierarchy contains set of actions available from its context menu. Besides standard actions for root node or file node, jInfer project type introduce its own actions. These actions could be separated into three groups: actions for *root* node, actions for *input folder* nodes and one action for *input/output file* nodes implemented by `ValidateAction` class. All classes implementing some jInfer project action extend standard java `javax.swing.AbstractAction` class besides `ValidateAction` class which is described in 2.3.3.

2.3.1 Root node actions

This group contains two actions, first is action that allows adding input files into *input folder* nodes (implemented by `FilesAddAction` class). Second one is slightly different, because unlike from the previous action, this one doesn't create new item in *root* node context menu but implements functionality for item that already exists in context menu - Run action.

First of the two actions, after its invocation, file choose dialog is opened which has set file filter according to available classes that implement `Processor` interface registered in jInfer with `ServiceProvider`. After selecting files in dialog, these files are inserted into particular *input folder* node according to mapping of file extension to folder type gained from `Processor` classes. If no mapping for some extension exists, this file is inserted into folder which is defined in *Miscellaneous* properties category (section 2.6).

Run action, which is available in root node context menu, is also present in NBP toolbar as a *Run* button and in jInfer Welcome window. This action is implemented in `RunAction` class and simply invokes *`Runner`*s `run()` method, if no other inference is already running, otherwise information dialog pops up, informing about already running inference. This check is done with `setActiveProject()` from `RunningProject` class (section 2.7).

2.3.2 Input folder node actions

To this group of actions belong two actions: action for adding input files into *input folder* node (`FileAddAction` class) and action for deleting all input files in *input folder* node (`DeleteAllAction` class). `FileAddAction` class implements action which opens file choose dialog and selected files are added into particular *input folder* node. `DeleteAllAction` class simply removes all input files previously added into particular *input folder* node.

2.3.3 ValidateAction class

`ValidateAction` class implements action, which allows documents and schema files to be validated against each other. Because of this, action is presented in context menu of document and schema files (in actual implementation is available for XML, dtd and XML schema files). Unlike the other action classes, this extends NBP `NodeAction` class, which allows to enable/disable action in context menu according to node selection. Action is enabled only if exactly one schema file is selected and one or more XML files. If the selected XML files are valid against selected schema, information dialog is popped up, otherwise error dialog is popped up and in *validate* output window is printed error message describing cause of non-validity XML files against schema.

For the validation against XML Schema, we use standard Java validation API [xml]. Besides XML Schema, this API allows to validate XML also against *Relax NG* and *Schematron* schema. If you want to extend the functionality of one of these schemas, you need to set `schemaLanguage` local variable to appropriate `XMLConstant`. If you need to extend validation to some other type of schema, you can use dtd validation as a sample.

For a validation error handling is used `JInferErrorHander` class which extends `DefaultHandler` class and overrides `error()` and `fatalError()` methods from this class. In these methods is created error message, which is consequently printed in validation *output* window.

2.4 Properties

Each jInfer project type has associated properties window through which can user change project-wide properties. This window is accessible through *properties* action in context menu of project *root* node. Properties window consists of category tree on a left side and properties pane on a right side. With each category in a tree is associated one properties pane. Structure of the category tree consists of two type of categories: actual category provided by some module and *virtual* category. Actual category has associated properties pane, which is displayed in a right side of the properties window. *Virtual* category displays as a properties pane special information panel that provides information about count and name of modules which are children of this virtual category. Each actual category could have virtual categories as a children or another actual categories, but not both at a same time. On the other side *virtual* category can have only actual categories as a children. If a *virtual* category has no child, this category is not displayed in a category tree.

Category in a properties window is represented by NB platform `ProjectCustomizer.Category` class and properties pane is represented by `JPanel` class. Classes responsible for creation of this properties window are `JInferCustomizerProvider` and `JInferComponentProvider`. From Jinfer module point of view is properties category represented by `PropertiesPanelProvider` interface and associated properties pane by `AbstractPropertiesPanel` abstract class. How to use this interface and abstract class to create custom project properties category presented in jInfer project type properties window is described in section 2.4.1.

`JInferComponentProvider` is very simple class that extends `ProjectCustomizer.CategoryComponentProvider` class from NBP and its only method is `create()` which takes category as a parameter and returns properties pane. In the `JInferCustomizerProvider` class, which extends NBP `CustomizerProvider` class, is the core functionality of collecting categories (`PropertiesPanelProvider` interface) with associated properties panes (`AbstractPropertiesPanel` class) from all jInfer modules.

2.4.1 Custom project properties category

As was written before, each jInfer project properties category is represented in jInfer modules by `PropertiesPanelProvider` interface. To create new category, this interface must be implemented and implementation must be annotated by the following code.

```
@ServiceProvider(service = PropertiesPanelProvider.class)
```

`PropertiesPanelProvider` interface provides the following methods:

- getName() - Gets a programmatic name of category.
- getDisplayName() - Gets a name of category displayed in properties window.
- getPriority() - Gets a priority of category, by which is sorted among other sibling categories in a tree.
- getPanel(Properties) - Gets the `AbstractPropertiesPanel` instance representing the category *properties pane*. The only parameter is instance of `Properties` class to which are saved properties which were set in *properties pane*.
- getParent() - Getter for a programmatic name of category which is parent of this category in a tree hierarchy.
- getSubCategories() - Gets a `List` of `VirtualCategoryPanel` instances representing virtual category children.

Example of the implemented `PropertiesPanelProvider`.

```
@ServiceProvider(service = PropertiesPanelProvider.class)
public class PropertiesPanelProviderImpl implements PropertiesPanelProvider{

  @Override
  public AbstractPropertiesPanel getPanel(final Properties properties) {
    return new ExamplePropertiesPanel(properties);
  }

  @Override
  public String getName() {
    return "exampleCategoryID";
  }

  @Override
  public String getDisplayName() {
    return "Example category";
  }

  @Override
  public int getPriority() {
    return 1000;
  }

  @Override
  public String getParent() {
    return null;
  }
```

```
  @Override
  public List<VirtualCategoryPanel> getSubCategories() {
    final List<VirtualCategoryPanel> result = new ArrayList<VirtualCategoryPanel>();
    result.add(new VirtualCategoryPanel("virtual_categoryID", "example virtual category",
            ModuleSelectionHelper.lookupImpls(IGGenerator.class)));

    return result;
  }
```

`VirtualCategoryPanel` extends `JPanel` and its constructor has three parameters:

- `categoryId` - programmatic name of the category or ID.
- `categoryName` - human readable name of the category displayed in jInfer properties window.
- `modules` - `List<? extends NamedModule>` of modules that will be children of this virtual category.

To create properties pane associated to custom category, class that extends `AbstractPropertiesPanel` class must be implemented. As you can see from a class diagram in the Figure 3, this abstract class has constructor with `Properties` parameter and two abstract methods, `store()` and `load()`, which must be implemented. Constructor gets a `Properties` instance as a parameter and saves it to `protected` field `properties`. This `Properties` instance is associated with jInfer project type *lookup* and is used to store properties that are set in this panel.

Purpose of the method `store()` can be easily guessed from its name, it's used to store values that are set in *properties pane* elements (text field, checkbox etc.) to the provided *properties* field. On the other side, `load()` method is used to populate *properties pane* elements with already saved values in the *properties*.
Example of the implemented `AbstractPropertiesPanel`.

```
public class ExamplePropertiesPanel extends AbstractPropertiesPanel {

  private javax.swing.JCheckBox checkBox;

  public ExamplePropertiesPanel(final Properties properties) {
    super(properties);
    init();
  }

  public void init() {
    ...
    initialize checkBox
    ...
  }

  @Override
  public void store() {
    properties.setProperty("some_property", Boolean.toString(checkBox.isSelected()));
  }

  @Override
  public void load() {
    checkBox.setSelected(Boolean.parseBoolean(properties.getProperty("some_property", "true")));
  }
}
```

2.5 Project wizard

This section describes ability of jInfer to create new jInfer project through *new project* wizard. This wizard contains two steps, where first is standard one with project name and location selection, and the second is the selection of Initial

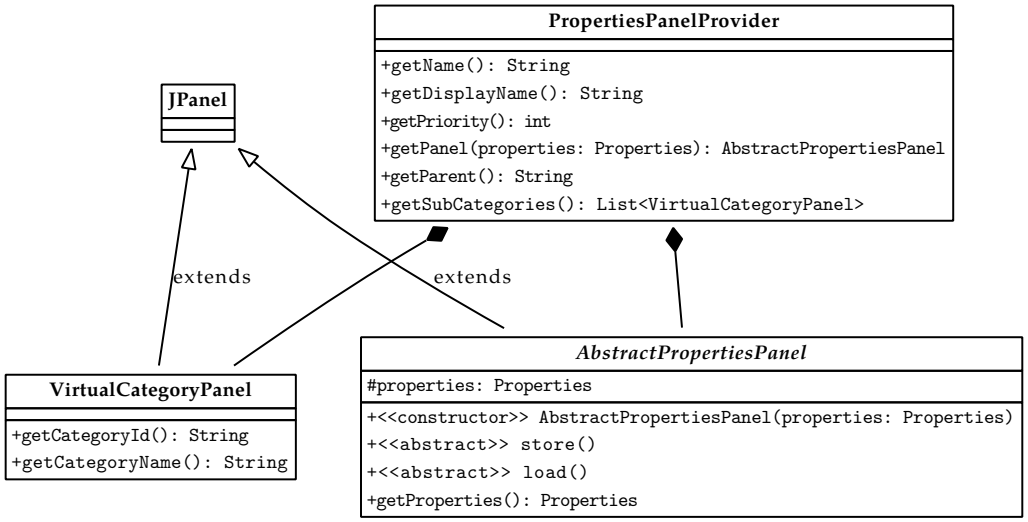


Figure 3: Class diagram for the custom jInfer project properties category.

grammar module, Simplifier and Schema generator module which will be used in inference for newly created module. New project creation process can be already finished in a first step, in that case, default modules will be used for inference. After finishing of the wizard, new jInfer project type is created with typed name in selected location. It is important to notice that package `cz.cuni.mff.ksi.jinfer.projecttype` contains `JInferTemplateProject.zip` file, in which is default folder structure of the jInfer project type, and the content of a zip is copied into location defined in the wizard. For more information, how to create new project wizard, please look at [pro].

All classes responsible for creation of this wizard are in package `cz.cuni.mff.ksi.jinfer.projecttype.sample`. It contains `JinferTemplateWizardIterator` class which defines number of steps in the wizard and creates panel for each step. Each step is represented by a class implementing `WizardDescriptor.Panel` interface, which creates steps panel and validate its data inserted by a user. Finally there is class extending `JPanel` in which are components of the first step.

2.6 Preferences

Also jInfer project type has its own properties category in a project properties window called *Miscellaneous*, and it is implemented by `ProjectPropertiesPanelProvider` and `ProjectPropertiesPanel` classes. In this category, two properties can be set. First property is default *input file* folder to which is inserted input file selected from *FilesAddAction*, that has extension which is not defined by any `Processor` registered in jInfer. Second property is global *LOG* level, which defines the minimum level of messages that will be displayed in jInfer output window. If some module has its own *LOG* level setting, global settings are overridden by these.

2.7 RunningProject utility class

This section describes `RunningProject` utility class, which is not part of the *`ProjectType`* module, but is closely related to it. This class can be found in a `cz.cuni.mff.ksi.jinfer.base.utils` package in the *`Base`* module. The main purpose of this class is to allow all jInfer *`modules`* to access `JInferProject` instance of the actual running inference. Notice that in jInfer application can be only one running inference at a time.

`RunningProject` class implements these methods:

- `setActiveProject()` - Setter of the `JInferProject` instance of a running project, which set the `JInferProject` instance if no other inference is running (no other `JInferProject` instance is already set). Returns `boolean` value, according as the setting was successful or not.
- `removeActiveProject()` - Removes `JInferProject` instance if some inference is running.

- `getActiveProject()` - Getter of the `JInferProject` instance of a running project.
- `isActiveProject()` - Checks if some inference is running (the `JInferProject` instance is set).
- `getActiveProjectProps()` - Gets `Properties` instance of the actual running inference (`Properties` instance is part of the `JInferProject` instace).
- `getNextModuleCaps()` - Getter of a capability of the next module in the inference chain.
- `setNextModuleCaps()` - Setter of a capability of the next module in the inference chain.

References

[Aho96] H. Ahonen. *Generating grammars for structured documents using grammatical inference methods*. PhD thesis, Department of Computer Science, University of Helsinki, Series of Publications A, Report A-1996-4, 1996.

[Bou] Ronald Bourret. Dtd parser, version 2.0. `http://www.rpbourret.com/dtdparser/index.htm`.

[gra] Graph visualization software. `http://www.graphviz.org/`.

[HMU01] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley, 2001.

[HW07] Yo-Sub Han and Derick Wood. Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.*, 370(1-3):110–120, 2007.

[JAX] Java architecture for xml binding. `http://jaxb.java.net/`.

[jun] Java universal network/graph framework. `http://jung.sourceforge.net/`.

[KMS+a] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Architecture*.

[KMS+b] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer AutoEditor automaton visualization and editor module*.

[KMS+c] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer Base Module Description*.

[KMS+d] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer BasicDTDExporter Module Description*.

[KMS+e] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer BasicIGG Module Description*.

[KMS+f] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer BasicRuleDisplayer Module Description*.

[KMS+g] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. jinfer javadoc. `http://jinfer.sourceforge.net/javadoc`.

[KMS+h] Michal Klempa, Mário Mikula, Robert Smetana, Michal Švirec, and Matej Vitásek. *jInfer TwoStep simplifier design and implementation*.

[log] Apache log4j™. `http://logging.apache.org/log4j/`.

[loo] org.openide.util.class lookup. `http://bits.netbeans.org/dev/javadoc/org-openide-modules/org/openide/modules/doc-files/api.html`.

[mod] Module system api. `http://bits.netbeans.org/dev/javadoc/org-openide-modules/org/openide/modules/doc-files/api.html`.

[Nor] Theodore Norvell. A short introduction to regular expressions and context free grammars. `http://www.engr.mun.ca/~theo/Courses/fm/pub/context-free.pdf`.

[pro] Project sample tutorial. `http://platform.netbeans.org/tutorials/nbm-projectsamples.html`.

[VMP08] Ondřej Vošta, Irena Mlýnková, and Jaroslav Pokorný. Even an ant can create an xsd. In *DASFAA'08: Proceedings of the 13th international conference on Database systems for advanced applications*, pages 35–50, Berlin, Heidelberg, 2008. Springer-Verlag.

[Vyh] Julie Vyhňanovská. Automatic construction of an xml schema for a given set of xml documents.

[wik] Regular expression. `http://en.wikipedia.org/wiki/Regular_expression`.

[xml] Xml validation api. `http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/1.6/api/javax/xml/validation/package-summary.html`.