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`

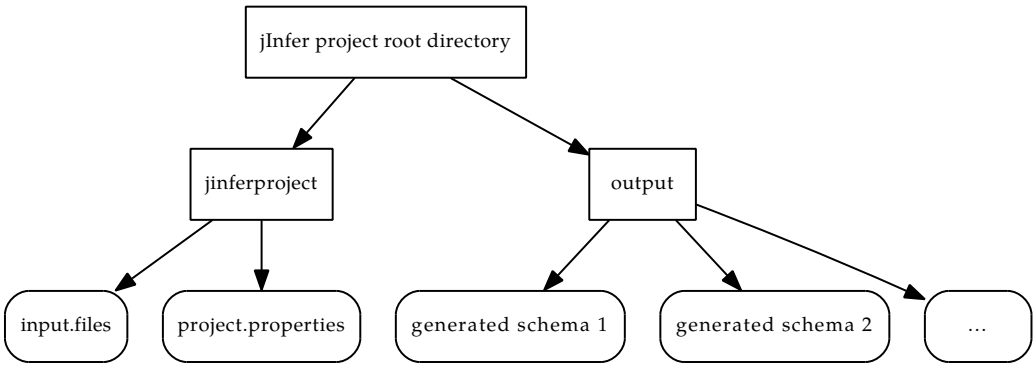
1 Introduction

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

For each jInfer project a specific directory structure in the filesystem describing this project is created. This structure is described in figure 1. A folder in the filesystem is considered to be jInfer project folder if it contains a *jinferproject* subfolder. This folder contains two files: *project.properties*, where all properties set for that particular project are stored. The second file is *input.files* which is an XML file filled with paths to files selected as particular project input (structure of this file is described in section 2.1.3). *output* subfolder contains schema files which were generated by jInfer inference runs.

2 Structure

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



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

Figure 1: Project directory structure.

- Base classes - Classes providing core functionality like project creation and definitions of 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 jInfer 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 provide actions allowing adding input files into the project, removing input files, or running the inference on the project.
- Properties - Classes responsible for creation of project properties window with properties category tree. These are located in `cz.cuni.mff.ksi.jinfer.projecttype.properties` package.
- Project wizard - Classes representing new project creation through a project wizard from `cz.cuni.mff.ksi.jinfer.projecttype.sample` package.

2.1 Base classes

This section describes classes needed for jInfer project creation and its integration into the NBP. The two main classes representing a 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 the jInfer project from/to the filesystem and also determining if a folder in filesystem is of jInfer project type (= contains *jinferproject* subfolder). For this purpose, class has three methods: `isProject()`, `save()` and `load()`. When jInfer project is loaded from the filesystem, `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 the `Project` interface from NBP and represents an in-memory representation of a jInfer project. Inner structure of this class is very simple, interface specifies only two methods: `getProjectDirectory()` and `getLookup()`. All the extensibility of a project type is done by the *Lookup* functionality of NBP. Project lookup contains properties, `Input` class (encapsulating input files), class creating output files, class building project tree in the Projects view, etc.

2.1.3 InputFiles

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

Example of *input.files* follows.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<jinferinput>
    <xml>
        <file loc="/home/user/example1.xml"/>
        <file loc="/home/user/example2.xml"/>
    </xml>
    <schemas>
        <file loc="/home/user/example.dtd"/>
    </schemas>
    </queries/>
</jinferinput>
```

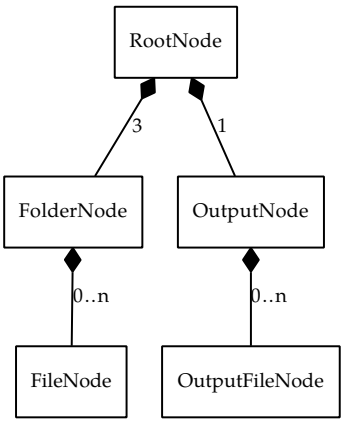


Figure 2: Project nodes structure.

2.2 Visualization classes

Each jInfer project in NBP is open after its creation in the Projects view and has tree structure where each node is represented by a NBP `Node` class. This structure in jInfer project type is provided by the `JInferLogicalView` class implementing `LogicalViewProvider` interface from NBP and is available through the `JInferProject` lookup. This class has one important method: `createLogicalView()` returning an instance of `RootNode` class. Simple class structure is described in figure 2.

`RootNode` extends NBP `AbstractNode` class which is a 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 the inference (Output). Root node also offers actions for adding input files, inference running and standard project actions like delete or move/rename project.

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

`OutputNode` class represents an *Output* node in jInfer project type tree and is slightly different from *input* nodes, because it refers to *output* subfolder in project filesystem hierarchy. In other words, files contained in this folder are shown as subnodes in *Output* node in project tree. These subnodes 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 deleted from filesystem too.

2.3 Actions

Each node in jInfer project tree hierarchy contains a set of actions available from its context menu. Besides standard NBP actions for root node or file node, jInfer project type introduces its own actions. These actions can be divided 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 a 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: an action that allows adding input files into *input folder* nodes (implemented by `FilesAddAction` class). Second one is slightly different, because it implements functionality for an item that already exists in context menu - Run action.

After invoking the first of the two actions, a file chooser dialog is open. Its file filter is set according to available classes implementing `Processor` interface registered in jInfer with `ServiceProvider`. After selecting file(s) in dialog, these files are inserted into the particular *input folder* node according to mapping of their file extensions to folder type retrieved from `Processor` classes. If no mapping for some extension exists, this file is inserted into a default folder defined in *Miscellaneous* properties category (section 2.6).

Run action, available from 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 an information dialog pops up, informing about the already running inference. This check is done invoking the `setActiveProject()` from `RunningProject` class (section 2.7).

2.3.2 Input folder node actions

Two actions belong to this group: action for adding input files into *input folder* node (`FileAddAction` class) and action for deleting all input files in the *input folder* node (`DeleteAllAction` class). `FileAddAction` class implements action which opens a file chooser dialog and adds the selected files into the particular *input folder* node. `DeleteAllAction` class simply removes all input files previously added into that particular *input folder* node.

2.3.3 ValidateAction

`ValidateAction` class implements action that 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 (its 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 the action in context menu according to node selection. Action is enabled only if exactly one schema file and one or more XML files are selected at the same time. If the selected XML files are valid against selected schema, information dialog pops up, otherwise error dialog pops up and an error message describing the cause of non-validity of XML files against this schema is printed in *validate* output window.

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 an example.

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

2.4 Properties

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

A category in the properties window is represented by NBP `ProjectCustomizer.Category` class, properties pane is represented by `JPanel` class. Classes responsible for creation of this properties window are `JInferCustomizer Provider` and `JInferComponentProvider`. From jInfer module point of view the properties category is 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 properties window is described in section 2.4.1.

`JInferComponentProvider` is a 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 (`AbstractProperties Panel` 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 `PropertiesPanel Provider` 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 this category.
- getDisplayName() - Gets a name of this category which will be displayed in properties window.
- getPriority() - Gets priority of this category, by which it is sorted among other sibling categories in the tree.
- getPanel(Properties) - Gets the `AbstractPropertiesPanel` instance representing the category *properties pane*. The only parameter is an instance of `Properties` class holding stored properties for the *properties pane*.
- getParent() - Getter for a programmatic name of category which is the parent of this category in the 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(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() {
    List<VirtualCategoryPanel> result = new ArrayList<VirtualCategoryPanel>();
    result.add(new VirtualCategoryPanel("virtual_categoryID", "example virtual category",
            ModuleSelectionHelper.lookupImpls(IGGenerator.class)));

    return result;
  }
```

`VirtualCategoryPanel` extends `JPanel`, 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 with a custom category, class extending `AbstractPropertiesPanel` class must be implemented. As you can see from the class diagram in the Figure 3, this abstract class has a 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 fields, checkboxes etc.) to the provided *properties* field. On the other hand, `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 JCheckBox checkBox;

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

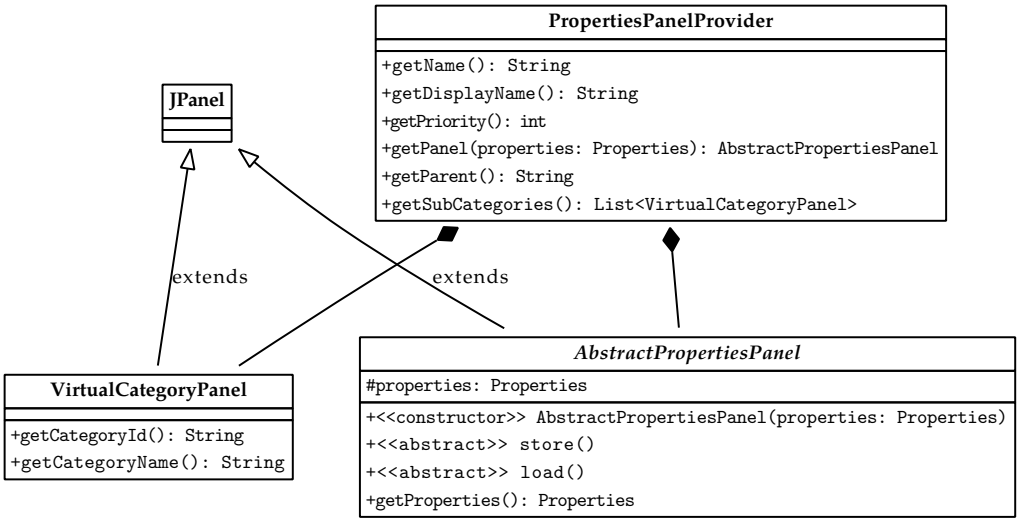


Figure 3: Class diagram for the custom jInfer project properties category.

selection of Initial Grammar Generator, Simplifier and Schema Generator modules which will be used in inference for this newly created project. New project creation process can be finished in the first step, in which case the default modules will be used for the inference. After finishing the wizard, new jInfer project is created with chosen name in the chosen location. It is important to notice that package `cz.cuni.mff.ksi.jinfer.projecttype` contains `JInferTemplateProject.zip` file, in which the default folder structure of a jInfer project is stored. The content of this zip is copied into location defined in the wizard. For more information on how to create a new project wizard, please refer to [pro].

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

2.6 Preferences

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

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 *`module`*s to access the `JInferProject` instance of the actual running inference. Notice that in jInfer application only one running inference is allowed at a time.

`RunningProject` class implements these methods:

- `setActiveProject()` - Setter of the `JInferProject` instance of a running project, which sets the `JInferProject` instance if no other inference is running (no other `JInferProject` instance is already set). Returns `boolean` value, based on whether the setting was successful.
- `removeActiveProject()` - Removes the `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 the `Properties` instance from the actual running inference (`Properties` instance is part of the `JInferProject` instace). If no project is running at the moment, an empty `Properties` instance is returned - this allows the JUnit tests to run with default settings.
- `getNextModuleCaps()` - Getter of capabilities of the next module in the inference chain.
- `setNextModuleCaps()` - Getter of capabilities 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 Vyhnanovská. 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`.