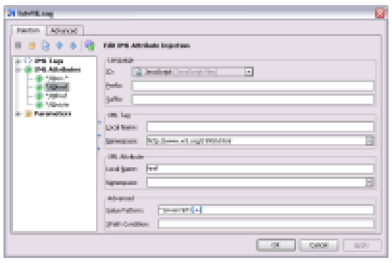
IntelliLang Configuration

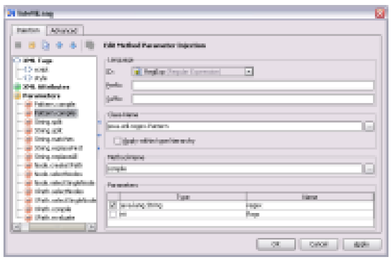
The configuration dialog provides the following options:

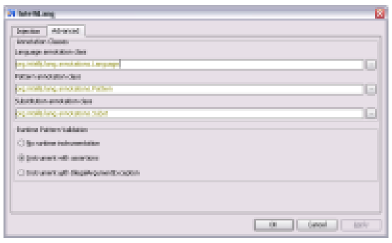
- Language injection for XML text
- Language injection for XML attributes
- Language injection for method parameters
- Advanced settings that let you change the names of the recognized base-annotations and the way how the pattern-validation is performed during runtime.

Screen shots

The screen shots below show some of the configuration options that can be tweaked and will be described in the following sections.







Language injection

This tab allows you to configure the language injection feature for XML text, attributes and method parameters. Use the buttons in the toolbar or the context menu actions to add, remove, copy or import new entries. To add a new entry, the appropriate group has to be selected first.

Reordering the entries is possible with the **Move Up** and **Move Down** buttons. This can be important to define the precedence of different XML-injection entries. If an entry matches, no more injections are applied unless the injection specifies a value pattern.

XML text

After adding a new XML text injection, select the ID of the Language to inject and optionally specify the prefix/suffix that make up the injection's context.

In the *XML Tag* pane, specify the local name (i.e. the name *without* any namespace prefix) and the namespace URI of the XML tag surrounding the text that should be treated as the selected language. The *name* field should not be empty, however the *namespace* field is optional.

The *Local Name* field takes a regular expression which makes it is possible to specify e.g. multiple tag names (**name1|name2**), case-insensitive names (e.g. **(?i)tagname** matches *tagname* as well as *TagName*), etc. Be sure not to enter any whitespace characters as they would be significant for the match.

XML attributes

This is similar to the XML text configuration. However, it is possible to leave the *name* of the *XML Tag* empty which means that the configuration will apply to any attribute that matches the configured name, regardless of its containing XML tag.

The attribute's name takes the local name of the attribute and is also specified as a regular expression. This can be e.g. used to match e.g. HTML event handler attributes by specifying the name "on.*". The attribute name may also be empty (unless the tag name is empty as well) which means that the configuration applies to all attributes of the containing tag.

Advanced XML options

The *Advanced* pane for XML Text and Attributes allows an even more fine-grained control over the injection process.

Value pattern

This field takes a regular expression that determines what part of the XML text's or attribute's value the language should be injected into. This can be used to inject the language only into values that match a certain pattern or to inject it into multiple parts that match the pattern. This is done by using the first capturing group of the pattern as the target for the injection.

Examples:

```
[$#]\{(.*?)\}
```

This matches the pattern used by the JSP/JSF Expression Language.

```
^javascript:(.*)
```

This matches the *javascript*-protocol that can be used in hyperlink-hrefs to execute JS code.

XPath condition

This field takes an XPath expression that can be used to address the injection-target more precisely than just by supplying its name and namespace URI. The context the expression is evaluated in is the surrounding XML tag for XML Text injection and the attribute itself for XML Attribute injection.

Example:

```
lower-case(@type)='text/javascript'
```

This limits the injection to tags whose **type** attribute contains the value "text/javascript".

It's possible to use the XPath extension functions that are provided by the Jaxen XPath engine, e.g. **lower-case()**. Also, there are three additional functions that can be used to determine the current file's name, extension and file type: **file-name()**, **file-ext()** and **file-type()**. You can also use normal code-completion to get a list of available functions.

For performance reasons, it's recommended to keep these expressions as simple as possible. Especially expressions that cause the whole document to be scanned, such as **//foo/bar** might cause performance problems with large files.

Method parameters

This is a possibility to make use of IntelliLang's features, if, for any reason, the injection annotations cannot be used. This mainly applies to configuring 3rd party/library methods as well as projects that still have to use Java 1.4.

The language selection is identical to the one described above. To select one or more parameters of a certain method, first choose its containing class either by typing in the name (the textfield supports completion) or by using the class chooser available through the [...] button. Next, select a method using the [...] button inside the "Method-Name" pane (it's not possible to edit the method name manually). Once a method has been chosen, the table in the *Parameters* pane is populated with the parameters of the selected method. Select the check box in the first column to have the selected language injected into arguments passed for this parameter. Note that only parameters of type String can be selected.

> There's already an IntelliJ IDEA-core (Inspection Gadgets rather) inspection that checks the arguments of some well-known methods to be a valid regular expression. However, IntelliLang can provide more detailed error messages and all the features of the Regular Expression Support plugin.

> The XPath language that is configured by default for some of the standard XPath-APIs is provided by the XPathView + XSLT-Support plugin, which is available here.

Importing configuration entries

To import the injection configuration from another IntelliJ IDEA installation use the *Import* button from the toolbar and select the file "IntelliLang.xml" from **IDEA-CONFIG-HOME**/options or from a JAR file that contains some exported (via File | Export Settings) IntelliJ IDEA settings. After selecting the file, a new dialog displays the entries contained in that file. Use the **Delete**-button to remove all entries you don't want to import. Note that this will *not* change the selected configuration file.

This selective import-feature makes it easy to share certain configurations in a team without losing any local entries like when importing settings via the core File | Import Settings action.

> To prevent inconsistent data, the import is only possible if the existing configuration is unchanged or has been saved with the *Apply* button.

Advanced settings

The advanced configuration tab allows you to specify different names for the base-annotations that should be used. This helps to avoid any dependencies on foreign code where this is not desired or possible. The custom annotations should just provide the same properties as the original ones, i.e. **value** for all of them and an optional (default = "") **prefix** and **suffix** for the **@Language** replacement.

Here it's also possible to configure the kind of runtime checks the plugin should generate for the **@Pattern** validation:

- No instrumentation: No checks will be inserted and doesn't touch any compiled class files
- Instrumentation using **assert**: Pattern-validation is controlled with the **-ea** JVM switch and throws **AssertionError**. This is the recommended way due to the potentially negative impact on the performance for methods that are invoked very often.
- Instrumentation using IllegalArgument- (for method parameters) and IllegalStateExceptions (for return values). This is the same the **@NotNull** instrumentation of IntelliJ IDEA does.

Contributed configurations

Additional configurations can be downloaded and imported from here. This community effect can help to minimize the configuration efforts and make IntelliLang even easier to use.

See Also

External Links:

- IntelliLang

Web Resources:

- Developer Community