User Manual

Version 0.1

Table of Contents

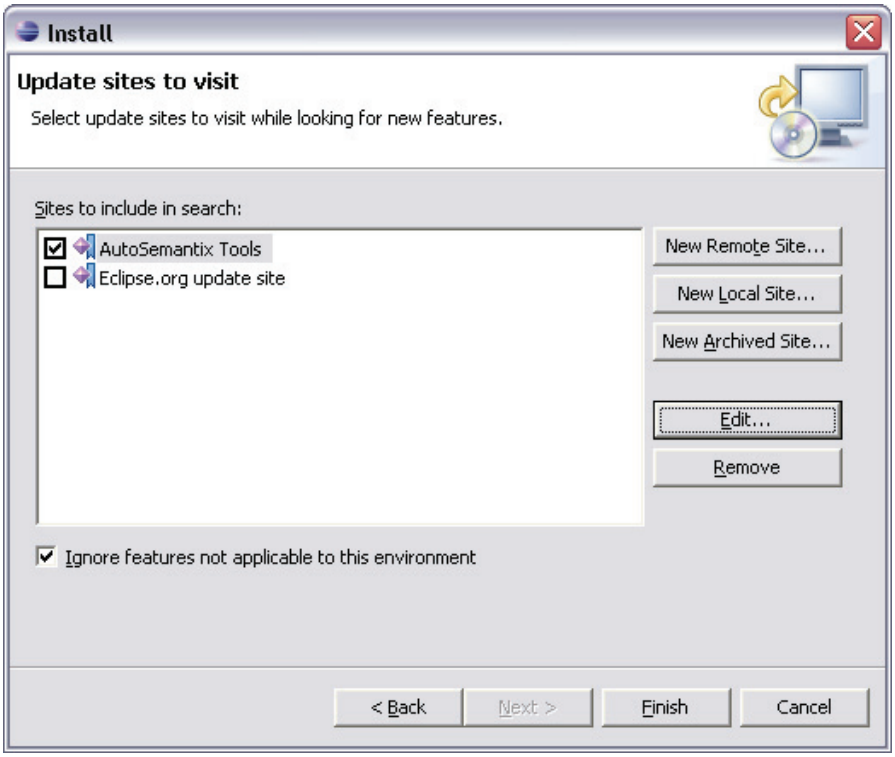
1. Installation

Requirements:

- Eclipse IDE version 3.1
  http://www.eclipse.org
- Eclipse Modeling Framework version 2.0.2
  http://www.eclipse.org/emf/
- Jena Framework version 2.2
  http://jena.sourceforge.net

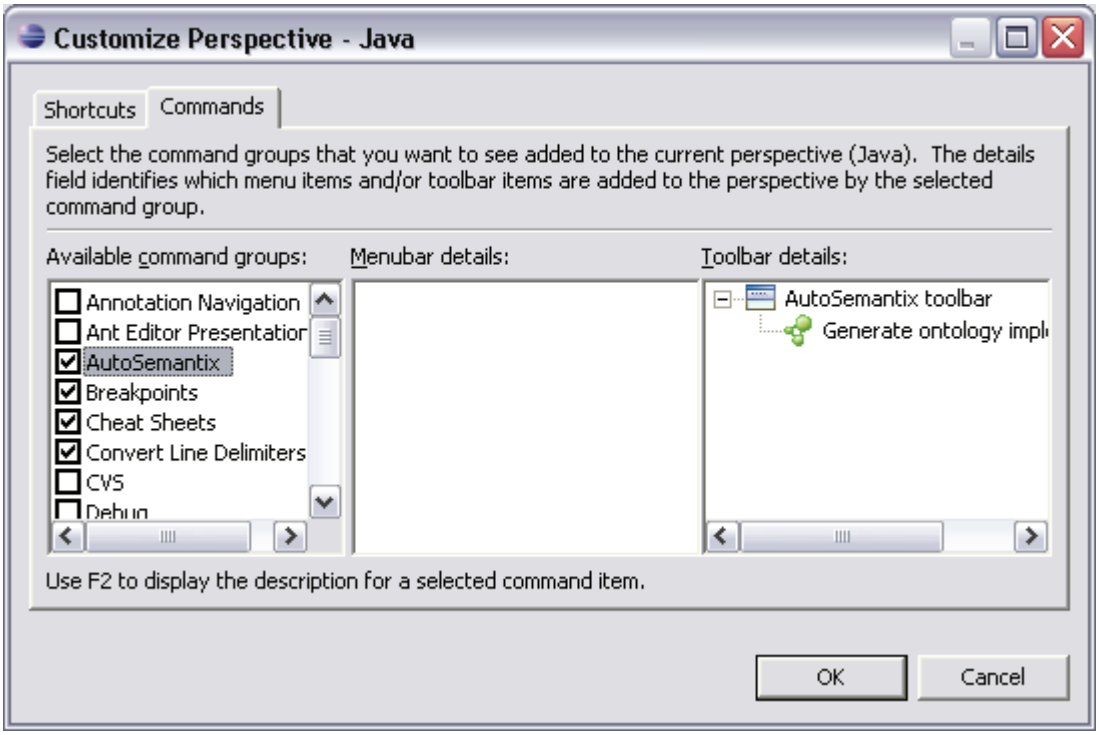
To install the AutoSemantix feature in your Eclipse 3.1 installation,

1. Select **Help** > **Software Updates** > **Find and Install**... to open the Install/Update wizard.
2. Select the **Search for new features to install** radio button and click **Next**.
3. Click **New Remote Site** to open the New Remote Site dialog.
4. Enter the name "AutoSemantix Tools", the URL:
   http://autosemantix.sourceforge.net/update, and click **Ok**.
5. Make sure that the checkbox to the left of "AutoSemantix Tools" is checked in the Install/Update wizard, and click **Finish**.
6. Click **Select All** and Click **Next**.

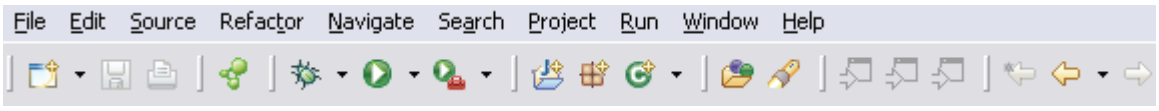


2. Getting Started

Once you have installed the AutoSemantix plug-in, the EMF and Jena framework you must customize your Eclipse perspective to show the AutoSemantix toolbar. To do this select the **Window** menu and click on **Customize Perspective…** Then go to the **Commands** tab and check the box next to the entry labeled “AutoSemantix”.



By enabling the AutoSemantix toolbar, you will now be able to see the AutoSemantix Update button. This button is used to run AutoSemantix on the currently selected Java project.

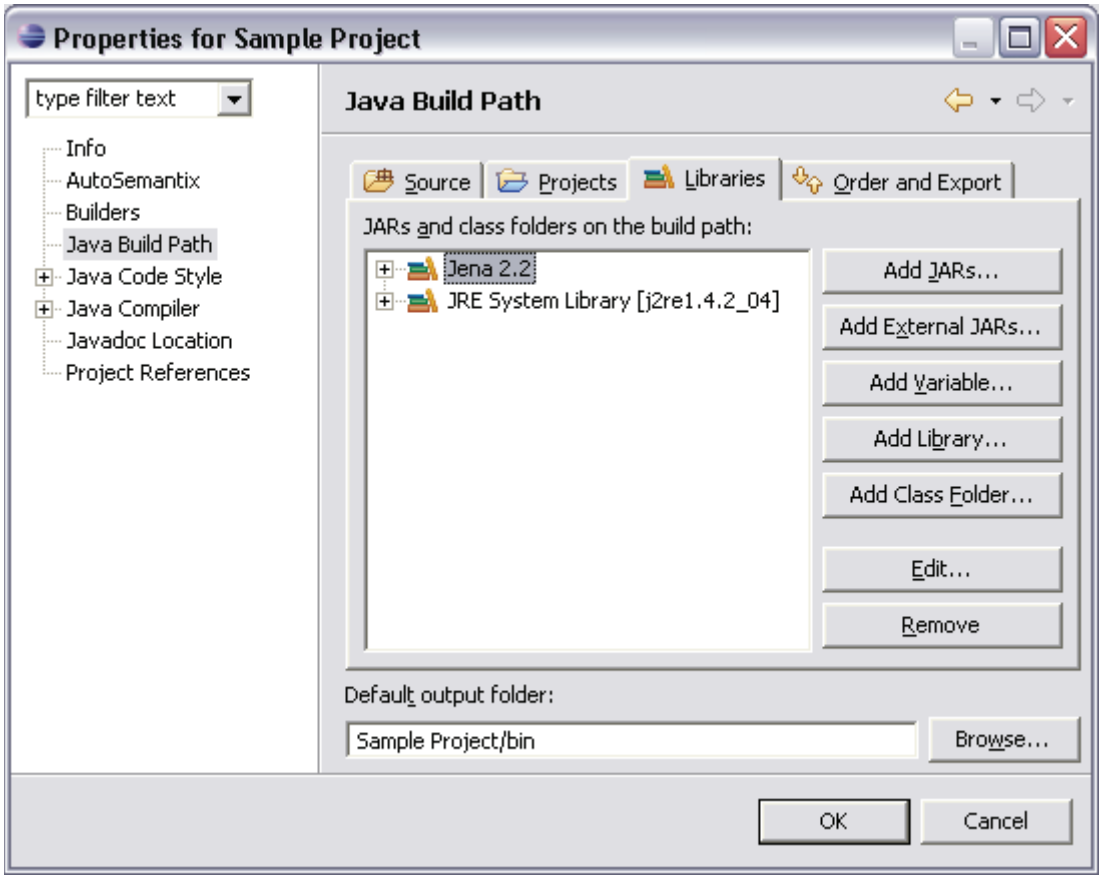


- Means that the current project has an ontology associated with it.
- Means that AutoSemantix is not setup for the current project.

3. Setting up a New Project

Start off by creating a new Java Project called “Sample Project” in your current workspace. Make sure that you select the option to create separate directories for the source and binaries.

Next you will need to add the Jena libraries to your project. The libraries that AutoSemantix generates rely heavily on the Jena framework to process the XML instance data. You can add Jena to the project by going to **Project** > **Properties** and selecting the **Libraries** tab in the **Java Build Path** section.



Click on **Add Library** then select **User Libarary** and click **Next**. Now click **User Libraries** and create a new library called “Jena”. Once you have created the new library you must add all the Jena JAR files (in the /lib directory of you Jena install directory) to the library.

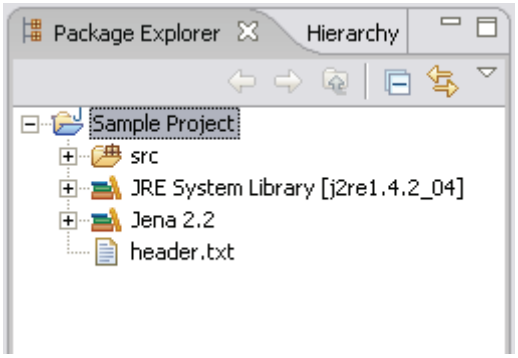
Once you have the Jena library on your class-path, create a text file in the root of your project and enter the following text into it.

header.txt

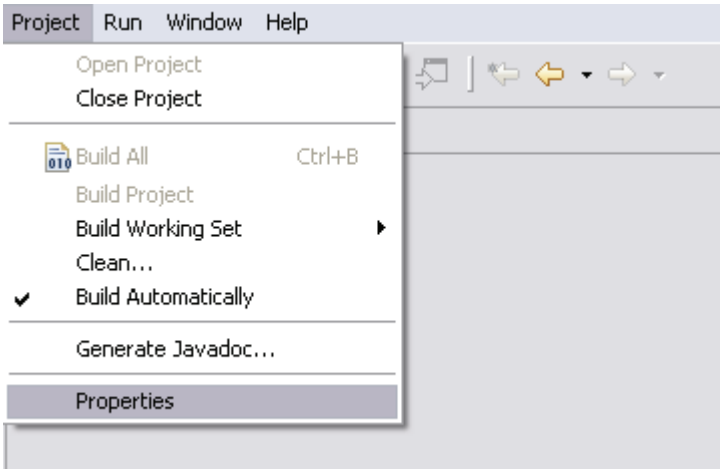
```
/**
 * This source code was automatically generated using AutoSemantix.
 * For more information please see http://autosemantix.sourceforge.net
 */
```

This text will serve as a header for each of the generated source files. You can change this later on so that it lists important information such as copyrights and licensing information.

Your project should now look like this:

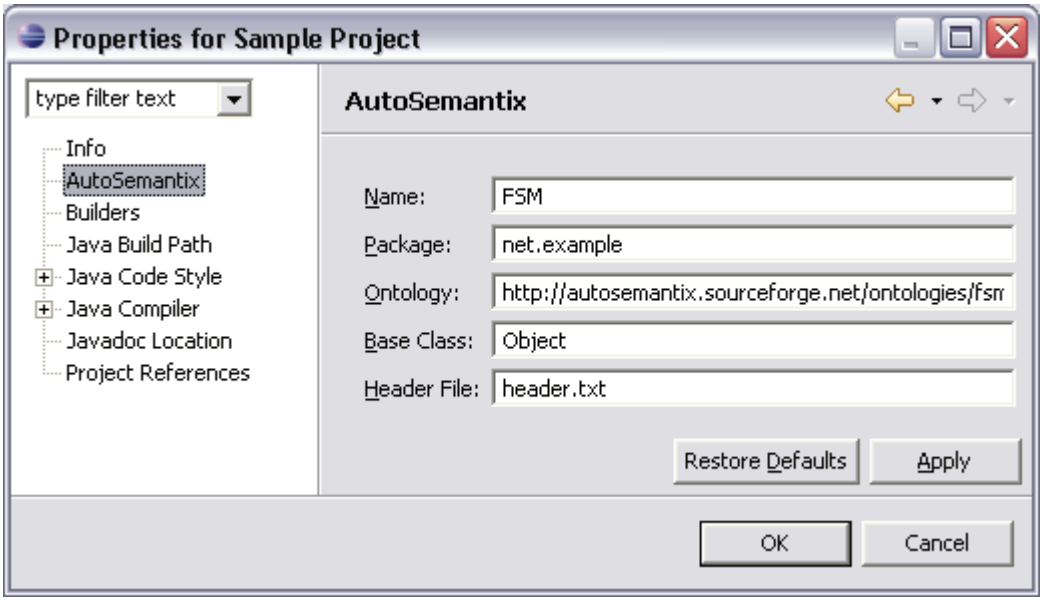


Now you are ready to set up the AutoSemantix properties for your project. Right click on the project in the Package Explorer and select **Properties**. This is the same as using the **Properties** option from the **Project** menu.



Now go to the AutoSemantix section and start to enter the following settings into the appropriate text fields.

Name: FSM
Package: net.example
Ontology: http://autosemantix.sourceforge.net/ontologies/fsm_v2.owl
Base Class: Object
Header File: header.txt



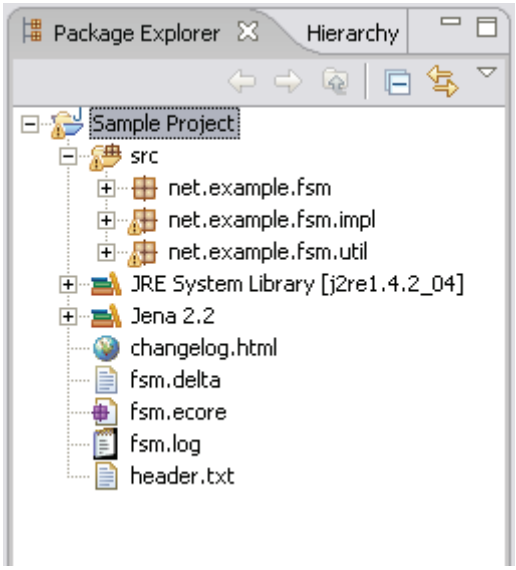
Each of the listed properties controls a different aspect of the code generation process so it is important to understand what each one does.

- **Name** – Specifies the name of the library to be generated.
- **Package** – This lets you specify a package into which all the generated source code is placed.
- **Ontology** – The ontology is the URI of the OWL file which represents your ontology.
- **Base Class** – Specifies the class from which Thing is derived. This lets you customize the Thing class which all other classes inherit from.
- **Header File** – The header file is simply a text file that gets inserted at the beginning of each generated source file. It is often used for copyright and licensing information.

Now you're ready to generate your implementation.

4. Generating a Library from an Ontology

Once you have defined the ontology properties for your project you can click on the Update button to generate your Java implementation.



Every time the implementation is updated, the following project files are produced.

- **Change Log** – An HTML representation of the Delta file.
- **Delta File** – An XML file which keeps track of the changes in each release on the ontology.
- **Ecore File** – An XMI file which stores a model of the Java implementation of the ontology.
- **Log File** – A plain text log file for recording runt-time warning sand errors.

The update process can be repeated as often as needed and changes to the ontology will be merged into the existing source code.

NOTE: As you can see, it's ok if there are warnings in some of the generated code. This is usually caused by the deprecated classes.

5. Working with the Generated Library

Once the code has been generated, creating instance data and writing it to an RDF file is pretty straight-forward. As the following code shows, you can easily create instances of any class in the ontology and manipulate their values using the setter methods.

```
FSMFactory factory = new FSMFactory();
OntModel model = factory.getModel();

FiniteStateMachine fsm1 = factory.createFiniteStateMachine("fsm1");

Alphabet alph = factory.createAlphabet("alph");
fsm1.setAlphabet(alph);

State s1 = factory.createState("s1");
s1.setOwningFSM(fsm1);

fsm1.setStartState(s1);

State s2 = factory.createState("s2");
s2.setOwningFSM(fsm1);
s2.setFinal(Boolean.TRUE);

Transition t1 = factory.createTransition("t1");
t1.setSource(s1);
t1.setTarget(s2);
t1.setInput("a");
t1.setOutput("b");

try {
      PrintWriter out = new PrintWriter(new FileWriter(new
                        File("state_machines.rdf")));
      RDFWriter writer = factory.createWriter();
      writer.write(model.getBaseModel(), out, factory.getOntology());
} catch (IOException e) {
      System.err.println(e);
}
```

It is also quite easy to load and manipulate instance data. For example, the following code will load instances from the local file *state_machines.owl* and query the model for a FiniteStateMachine instance called *fsm1*. It then calls a get method on the returned object and outputs the result.

```
FSMFactory factory = new FSMFactory();
OntModel model = factory.getModel();

InputStream in = FileManager.get().open("state_machines.rdf");
if (in == null) {
      throw new IllegalArgumentException(
      "File: state_machines.owl not found");
}

model.read(in, "");

FiniteStateMachine fsm1 = factory.getFiniteStateMachine("fsm1");

Sysem.out.println(fsm1.getAlphabet());
```

7. Customizing the Generated Source Code

If you look closely at the source code generated by AutoSemantix, you'll notice that each method has a *@generated* tag in its Javadoc comments. This tells AutoSemantix that this method was automatically generated from the ontology and can be replaced when the ontology is updated.

Adding you own methods to the implementation code is as simple as just typing it into the file in any editor and saving it. As long as you don't have the *@generated* tag in the comments, AutoSemantix will leave the method alone the next time it updates the implementation.

```
/**
 * @param _final The new value to set the _final attribute to.
 * @see StateImpl#isFinal()
 * @generated
 */
public void setFinal(Boolean _final) {
      setAttribute(StateImpl._final, _final);
}

/**
 * This is a custom method.
 */
public void finalize() {
      setFinal(Boolean.True);
      System.out.println("Node "+getName()+" has been finalized.");
}
```

8. Working with Multiple Inheritance

AutoSemantix uses a hierarchy of interfaces to support multiple inheritance in the generated Java code. Once an object is created it may be type cast to any of its parent classes using the standard Java notation.

It is important to note that when you add your own method to a class you must also add the appropriate method stubs to all of it's subclasses in order for that method to be inherited. Obviously this is not the most efficient way to work so in future releases of AutoSemantix this will be done automatically.

9. Creating an Ontology

Ontologies can be created in any number of ways from using a simple text editor to graphical editors like Protégé. Protégé is an open source tool from Stanford University and can be downloaded from http://protege.stanford.edu

In order for an ontology to be processed correctly by AutoSemantix it must follow certain guidelines.

- The ontology must be written using the OWL Lite schema.
- If you are using more than one version of an ontology, you must use the *owl:version* and *owl:priorVersion* attributes in your ontologies to let AutoSemantix know which order to place them in.
- The ontologyand any prior versions must be available from the specified URIs.

It is also very useful to thoroughly comment each aspect of you ontology as these comments will be transferred to the Javadoc comments of your implementation. Using Javadoc comments will help you qyuickly document your generated source code.

You should also be aware of the AutoSemantix naming system and how it translates class and attribute names into proper Java identifiers. AutoSemantix provides a one-to-one mapping from any string into a valid Java identifier by using the following rules.

- All white space is removed and capitalization is used to differentiate words.

  pinot noir → *pinotNoir*

- All reserved Java words have an underscore (_) added to the front of them.

  public → *_public*

- Terms beginning with a digit have that digit spelled out.

  123 → one23

- Illegal symbols are converted to a textual description of the symbol.

  M&M → MAndM

10. Making changes to the Ontology

There are many ways in which changes can be made to an ontology. AutoSemantix relies on certain attributes being available in order to be able to correctly identify which changes have occurred.

Adding

To add a new element to ontology there are no special requirements, simply add whatever you want and it will automatically be detected by AutoSemantix the next time you do an update.

Removing

When elements of your ontology are no longer needed, you must set them as being deprecated. This insures that your ontology will always be backwards compatible with previous versions and help AutoSemantix keep track of complex changes.

Renaming

To let AutoSemantix know that you have renamed an element, you must first deprecate the old element then create a copy of it under the new name and set the two elements as being equivalent to each other.