|  |  |  |
| --- | --- | --- |
| Uni_Logo.png |  | Zuschneiden.jpg |
| **Stiftung University of Hildesheim**  Marienburger Platz 22  31141 Hildesheim  Germany |  | **Software Systems Engineering (SSE)**  Institute for Computer Science  Faculty for Mathematics, Natural Science, Economics, and Computer Science |
|  |  |  |



**EASy-Producer**

Engineering Adaptive Systems

**Developers Guide**

Version 0.3

04.12.2012

**Version**

|  |  |  |
| --- | --- | --- |
| 0.1 | 28.08.2012 | Initial version |
| 0.2 | 10.09.2012 | Reasoning section moved to the end of the document, prerequisite and installation added, debug flags added to section 3 |
| 0.3 | 04.12.2012 | Preface added, Section 3.1.1 added |
|  |  |  |

**Preface**

EASy-Producer is a Software Product Line Engineering tool developed by the Software Systems Engineering (SSE) group at the University of Hildesheim.

The tool is available as an Eclipse plug-in under the terms of the Eclipse Public License (EPL) Version 1.0

The SSE group hosts the following EASy-Producer update site for easy installation and updates:

<http://projects.sse.uni-hildesheim.de/easy/>

©2012 Software Systems Engineering (SSE) Group, University of Hildesheim, Germany.

Table of Contents

[1. Introduction 5](#_Toc342477834)

[2. Installation 6](#_Toc342477835)

[2.1. Prerequisites 6](#_Toc342477836)

[2.2. Installation: Step by Step 6](#_Toc342477837)

[2.3. Recommendations 8](#_Toc342477838)

[3. EASy-Producer Extensions 9](#_Toc342477839)

[3.1. Implementing a New Instantiator 9](#_Toc342477840)

[3.1.1. Instantiation Concept in EASy-Producer 10](#_Toc342477841)

[3.1.2. Eclipse Plug-in Project Creation and Configuration 12](#_Toc342477842)

[3.1.3. Instantiator Implementation 17](#_Toc342477843)

[3.1.4. Instantiator Integration 20](#_Toc342477844)

[3.2. Implementing a New Reasoner 21](#_Toc342477845)

[4. Known Bugs 22](#_Toc342477846)

# Introduction

EASy-Producer[[1]](#footnote-1) is a Software Product Line Engineering (SPLE) tool which facilities the most recent trends and concepts in SPLE, such as large-scale Multi-Software Product Lines (MSPL), product line hierarchies, and staged configuration and instantiation. The focus of this tool is to support these rather complex concepts in an easy-to-use way. Thus, this tool allows developing a first prototypical Software Product Line (SPL) within minutes. However, EASy-Producer is a research prototype for demonstrating new approaches to SPLE in general and, in particular approaches for simplifying the development of SPLs developed by the Software Systems Engineering group (SSE) at the University of Hildesheim.

This document provides a prototypical[[2]](#footnote-2) developers guide that introduces the reader to the basic capabilities of EASy-Producer and how to develop further extensions to this tool. In Section 2, we will give a brief overview on the SPLE concepts supported by EASy-Producer. This will include introductions to the concepts of SPLE in general, staged configuration and instantiation, MSPL, and product line hierarchies. Further, we will discuss the corresponding approaches in EASy-Producer that were developed for easing the application of these concepts.

Section 3 will give guidance for the first steps with EASy-Producer. This section includes the mandatory prerequisites, the installation guide, and additional recommendations for running the tool successfully.

In Section 4, we will introduce EASy-Producer in terms of describing the development of a first prototypical SPL and the derivation of a product line product. This will cover all aspects of SPL development ranging from creating a new product line project in EASy-Producer, defining a variability model and implementing the corresponding product line artifacts, to the derivation, configuration, and instantiation of a specific product. While the purpose of this section is to describe and illustrate the basic application of EASy-Producer, we will not discuss technical details of the tool at this point. This will be part of the next section.

Section 5 will describe EASy-Producer in detail. This includes detailed descriptions of the individual editors and views as well as discussions on the underlying conceptual decisions and technical capabilities.

In Section 6, we will discuss advanced SPL concepts, like MSPL, staged configuration, etc., and how these concepts can be applied using EASy-Producer.

Finally, Section 7 will summarize known bugs and current workarounds.

# Installation

In this section, we will describe the installation of EASy-Producer. In order to guarantee a successful installation, we will introduce a set of mandatory prerequisites. This will be part of Section 2.1 and sets up the environment in which EASy-Producer is intended to be executed. In Section 2.2, we will describe the installation of the tool in a step-wise manner using the Eclipse update site mechanism and the EASy-Producer update site. Finally, Section 2.3 will give some useful recommendations.

## Prerequisites

EASy-Producer is developed as an **Eclipse[[3]](#footnote-3)** plug-in and requires **Xtext[[4]](#footnote-4) version 2.3.1**. Thus, in general, any Eclipse installation with Xtext version 2.3.1 is fine for installing and running EASy-Producer. However, we cannot guarantee that any combination of Eclipse and Xtext version 2.3.1 will work with EASy-Producer. Thus, we propose the following Eclipse versions to be tested with EASy-Producer (and Xtext version 2.3.1):

* Eclipse 3.6 (Helios)
* **Eclipse 3.7 (Indigo)**
* Eclipse 4.0 (Juno)

We recommend using Eclipse 3.7 (Indigo) as this is the most exhaustively tested version of Eclipse with EASy-Producer. Download an Eclipse package from <http://www.eclipse.org/downloads/>.

Further, Xtext version 2.3.1 has to be installed in the newly downloaded Eclipse instance. Typically, this is installed automatically when installing EASy-Producer due to plug-in dependencies. However, we encountered situations in which these dependencies were not automatically resolved. In such a case, install Xtext manually as follows:

* **Define a new update site** in Eclipse using the following location (if you are not familiar with defining new Eclipse update sites, please consider Section 2.2): <http://download.eclipse.org/modeling/tmf/xtext/updates/composite/releases/>
* **Select the additional plug-ins** to be installed. This will be “*Xtext-2.3.1*” only. All other dependencies will be resolved automatically. **IMPORTANT**: If the dependencies will not be resolved automatically or you encounter problems regarding missing “*EMF*”-plug-ins, please download and use **Eclipse Modeling Tools** instead of any other Eclipse packages.

## Installation: Step by Step

The SSE group hosts an EASy-Producer update site for easy installation and updates. Thus, the first step for installing EASy-Producer is to define a new update site in Eclipse. For this purpose, start Eclipse and open the *Install New Software* dialog by clicking *Help 🡪 Install New Software…* as shown in Figure 1:



Figure 1: Open the “Install New Software” dialog

The *Install* Dialog will appear (cf. Figure 2). In this dialog, a new location for available software has to be added. Thus, click on the *Add…* button in the upper right location of the dialog.



Figure 2: Add a new location for software updates

The *Add Repository* dialog requires the definition of a name for the new update site and a location as illustrated in Figure 2. The name is up to the user. For example, enter “*EASy-Producer update site*”. The location is the URL of the update site:

EASy-Producer update site: <http://projects.sse.uni-hildesheim.de/easy/>

Finish the definition of the new update site by clicking the *OK* button of the *Add Repository* dialog.

The *Install* Dialog will now contain multiple categories. If you are installing EASy-Producer for the first time and do not know which features to select, expand the *Quick Installation of EASy-Producer* category and select the *EASy-Producer Quick Installation* feature. This will install all required components automatically. For more experienced users, select the categories and features as needed and click the *Next* button. Follow the steps for installing EASy-Producer (accept the license agreement and ignore the security warning for installing software of unsigned content, etc.), and restart Eclipse as prompted.

Finally, you have successfully installed the EASy-Producer.

## Recommendations

In order to avoid memory problems while using EASy-Producer, we recommend increasing the memory of the Eclipse application in which EASy-Producer is executed. The memory problems are due to Xtext which requires more memory than defined in a typical Eclipse configuration.

Open the “*eclipse.ini*” file in your Eclipse directory and enter the following parameters at the end of the file:

-vmargs

-Xms40m

-Xmx512m

-XXMaxPermSize=128m

# EASy-Producer Extensions

EASy-Producer provides an extension point mechanism to add additional functionality to the basic implementation. An extension is always implemented as an Eclipse plug-in and may provide customer-specific functionalities in terms of individual reasoners or instantiators. A new reasoner may provide new or adapted capabilities to check, for example, whether a variability model or a specific product configuration is valid. Custom instantiators may be capable of instantiating artifacts of different types or in a specific way. In order to easy the development and integration of such extensions, EASy-Producer is capable of automatically searching and integrating new plug-ins. Thus, developers only have to provide the necessary information to EASy-Producer to include their desired functionalities.

In this section, we will describe how to implement extensions to the EASy-Producer tool. In Section 3.2, we will describe the implementation of a new instantiator and its integration with EASy-Producer. In Section 3.1, we will describe the implementation and integration of a new reasoner. Both sections will provide detailed guidance from project creation and configuration to the actual application of the custom plug-ins.

In order to debug errors and failures during the development of EASy-Producer extensions, add the following flags to the “Run Configuration” of your Eclipse as needed (introduce the new flags with a single prefixed “-D” in the Run Configuration):

* **-debug:** This flag will print information on the variability model of EASy-Producer
* **-log:** This flag will print EASy-internal debug messages, such as errors, etc.
* **-equinox.ds.debug:** This flag will print debug messages regarding the service registration mechanism. For more details, see Section .
* **-equinox.ds.print:** This flag will print additional information regarding the service registration mechanism. For more details, see Section .

## Implementing a New Instantiator

An instantiator is an external and maybe third-party tool that processes product line artifacts in its specific way. For example, the Velocity instantiator, which is shipped as a default instantiator with EASy-Producer, resolves Velocity-specific tags within Java code in accordance to a specific configuration[[5]](#footnote-5). This resolution capability allows deriving individual product variants based on the configuration values and the corresponding manipulation of the Java code. However, the default instantiators of EASy-Producer may be insufficient in some situations. Thus, we provide a simple extension mechanism for integrating custom instantiators with EASy-Producer.

In the first part of this section, we will introduce the basic instantiation concept of EASy-Producer to form a common understand of how an instantiator works. In the second part, we will describe how to set-up a new plug-in project in Eclipse for implementing a custom instantiator. This also includes the specific configurations that have to be done to utilize the automated search and integration mechanism provided by EASy-Producer. The third part will discuss the methods that are required when implementing a new instantiator. The focus of this part will be on how, when and why EASy-Producer invokes specific methods of an instantiator. In the fourth part, we will finally show how to integrate a new instantiator.

### Instantiation Concept in EASy-Producer

In this section, we will introduce the basic instantiation concept in EASy-Producer in order to describe how the instantiators work. In the first part, we will have a black-box view on a generic instantiator for identifying the required input (prerequisites) for an instantiator. Please note that a generic instantiator is not related to any specific variability implementation technique (VIT). Thus, we can only give a very simple view on the instantiators in general. In the second part, we will relate the identified prerequisites in terms of giving a white-bob view on the generic instantiator. However, the actual logic that defines how to process artifacts depends on the used VIT. Thus, this view is again simplified.

An instantiator in EASy-Producer in general takes a set of possibly different input and produces another set of output. Typically, the input consists of a configuration based on an IVML variability model, generic artifacts (i.e. the artifacts of a software product line including variation points, etc.), and different variants that can be applied to the variation points of the generic artifacts. Please note that the presences as well as the location of generic artifacts, variation points and variants depends on the used VIT (we will detail this below). Based on this input, the instantiator produces an instantiator- or VIT-specific output, typically, a set of product-specific artifacts with resolved variability as illustrated by Figure 3.



Figure 3: Black-box view of a generic instantiator (simplified)

Below, we will describe the required input (prerequisites) of an instantiator in detail:

* **IVML Configuration:** The IVML configuration is based on the previously defined variability model using the IVML modeling language (explicit prerequisite not mentioned in Figure 3). A configuration includes all variable-value pairs, which are valid with respect to the constraints defined in the model. The validity of such a configuration is automatically checked before the instantiation process. This prevents from calling the instantiator with an invalid configuration, which will yield corrupted product-specific artifacts.
* **Generic Artifacts:** The generic artifacts, i.e. of a software product line, include variation points (indicated by gray shapes in Figure 3) to which one or multiple variants (indicated as colored shapes in Figure 3) can be bound. However, the way of specifying such variation points (and the related variants) depends on the used VIT. For example, using preprocessing as a VIT, the variation points might be indicated by #if-statements in the generic artifacts.
* **Variants:** The different variants that can be applied to a variation point of a generic artifact may be implemented independent from the generic artifacts. However, this also depends on the used VIT. In the example of preprocessing, the different variants will be part of the generic artifacts. The variants not selected as part of the product will then be deleted by the preprocessor. In case of using aspect-orientation as VIT, the variants are implemented as independent aspects, which can be woven into the generic artifacts if they are selected as part of the product.

The relation between IVML configuration, generic artifacts, and variants is illustrated in Figure 4. The decision variables and their values as well as the files to instantiate will be saved (temporarily) in an instantiator context (using an instantiator context is a best-practice; however, this is not required). An instantiator context defines the scope of the instantiator. To be more precise, the instantiator context subsumes all files that are selected to be instantiated by this instantiator. Further, the instantiator context includes all variable-value pairs of the configuration. The instantiator context will then be processed by the instantiator. The way of processing this information depends on the implemented instantiator logic. Figure 4 sketches two possible variants of such logic in pseudo-code:

* **Variant A:** In variant A the instantiator will process all files given by the instantiator context, i.e. in terms of searching for variation points in each file (this is described as VIT-statement in Figure 4 some VITs may also introduce further concepts besides variation points that can be searched and processed by an instantiator). However, what to do if a certain variation point (or VIT-statement) is found heavily depends on the used VIT and the intention of the domain engineer to define these variation points. Thus, we cannot give further information regarding the actual logic.
* **Variant B:** In variant B the instantiator will process all decision variables given by the instantiator context, i.e. in terms of searching for a specific variable-value pair. However, what to do if a certain variable-value pair is found heavily depends on the used VIT and the relation between the specific decision variable and the artifacts. Thus, we cannot give further information regarding the actual logic.

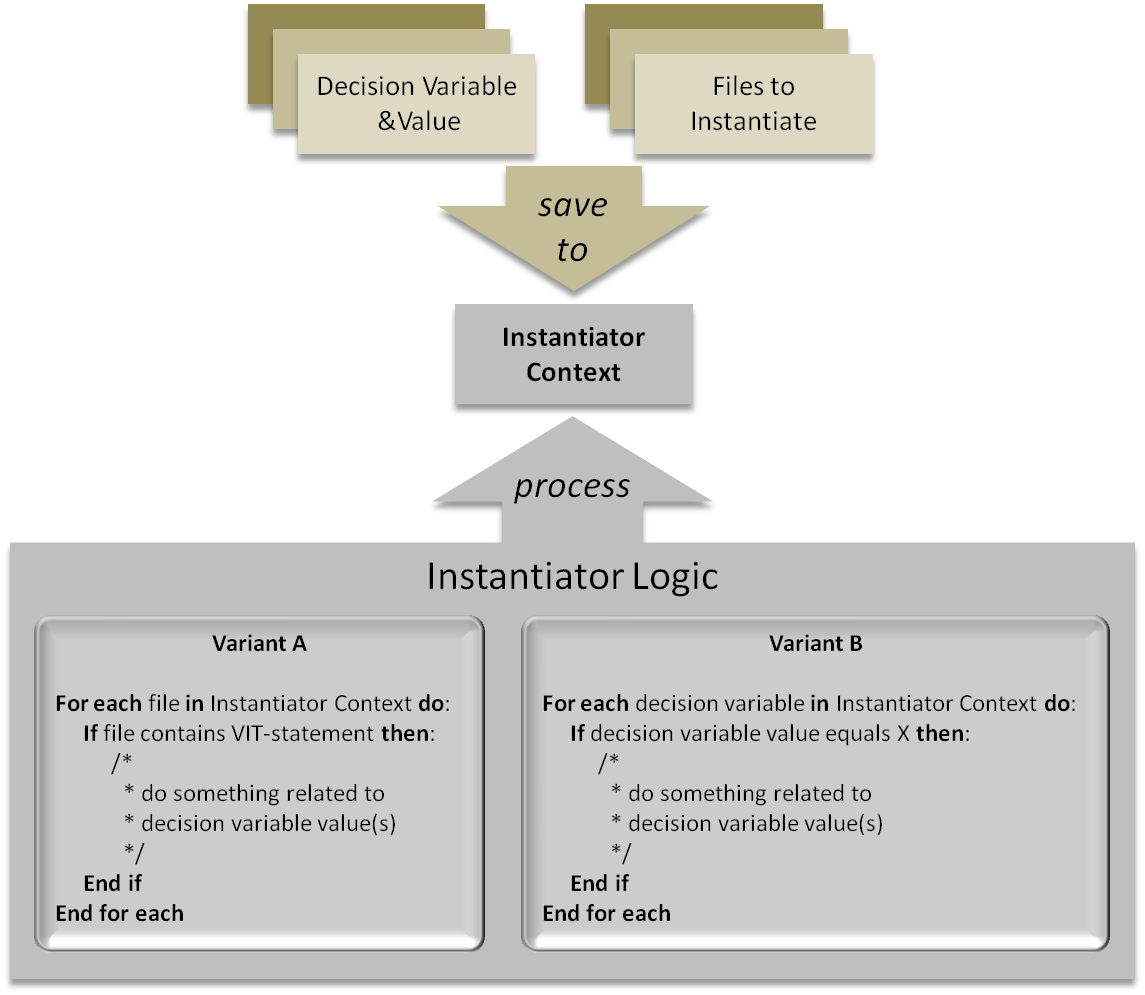


Figure 4: White-box view of a generic instantiator (simplified)

An instantiator may also provide further functional, i.e. the generation of files based on the variable-value pairs of its context (this may also exclude the selection of files to instantiate as the instantiation process will generate completely new files), the combination of other (non-source) artifacts like documentation, etc. However, this depends on the used VIT and the specific purpose an instantiator is designed for.

### Eclipse Plug-in Project Creation and Configuration

The first step towards a new instantiator is to create new Eclipse plug-in project: *File 🡪 New 🡪 Other…* . In the emerging wizard, open the category *Plug-in Development*, select *Plug-in Project*, and click the *Next* button. In the *New Plug-in Project* wizard, define a name for your project. We will use the following name throughout this section: *EASyExampleInstantiator*. Further, define the target platform with which the plug-in should run. In this case, the instantiator plug-in will run with a *standard OSGi framework*. Figure 6 shows how the first configuration page for the new plug-in project must look like.



Figure 5: Configuration of a new Eclipse plug-in project for a new Instantiator

Click on the *Next* button and define the properties of your plug-in. We will use the following values for the required properties:

* *ID*: de.uni\_hildesheim.sse.easy.instantiator.exampleInstantiator
* *Version*: 0.0.1
* *Name*: EASyExampleInstantiator
* *Provider*: University of Hildesheim – SSE

Leave all other properties and options as-is and finish the configuration by clicking the *Finish* button of the *New Plug-in Project* wizard. Please note that some of the following steps described in this section can also be done by using the wizard. However, we decided to do these steps manually to provide a more detailed explanation.

The plug-in manifest file will open by default. In the *Overview* tab check the *Activate this plug-in when one of its classes is loaded* checkbox and the *This plug-in is a singleton* checkbox. The first check will guarantee that the plug-in is activated when EASy-Producer loads one of its classes, while the second check is related to one of the concepts of EASy-Producer: each instantiator exists only once (only one instance) and can be accessed by any product line project. Thus, this check guarantees that the new instantiator will follow the concepts of EASy-Producer.

The next step is to define the dependencies of the new plug-in. Thus, open the plug-in manifest and select the *Dependencies* tab. On the left side click the *Add…* button in order to specify the following plug-ins:

* *org.eclipse.equinox.ds*: This plug-in simplifies the task of authoring OSGi services by performing the work of registering the service and handling service dependencies[[6]](#footnote-6).
* *org.eclipse.core.runtime*: This plug-in provides support for the Eclipse runtime platform, core utility methods, and the extension registry[[7]](#footnote-7). The latter is important for the EASy-Producer extension mechanism.
* *de.uni-hildesheim.sse.easy.instantiatorCore*: This plug-in provides the core capabilities of the EASy-Producer instantiator concept. We will use parts of this plug-in in Section 3.1.3.

By default, Eclipse adds the package *org.osgi.framework* as *Imported Packages* because of the selected target platform in the *New Plug-in Project* wizard. However, this package is not required for the integration with EASy-Producer and, thus, can be removed. Select the package on the right side of the *Dependencies* tab and click the *Remove* button. Then, click the *Add…* button and select *org.osgi.service.component* as *Imported Packages*. This package provides support for service components and their interaction with the context in which they are executed[[8]](#footnote-8). The *Dependencies* tab should now look like the one illustrated by Figure 6.



Figure 6: Definition of the required plug-ins for the new instantiator

In order to register the new plug-in to EASy-Producer, the service component has to be declared. Thus, switch to the *MANIFEST.MF* tab in the plug-in manifest and add the following *Service-Component* declaration:

Service-Component: OSGI-INF/instantiator.xml

This Service-Component declaration specifies the location where to find the information about the service component, which shall be integrated into EASy-Producer. The declared XML file will be defined in the next step. Figure 7 shows how the manifest file must look like.



Figure 7: Declaration of the service-component for the new instantiator

The definition of the service component requires the creation of a new folder within the plug-in project. Right click on the plug-in project and select *New 🡪 Folder*. The name of the folder has to be *OSGI-INF*. Then, create a new XML file within this folder. Right click on the folder and select *New 🡪 Other…* . In the emerging wizard, open the category *XML*[[9]](#footnote-9), select *XML File*, and click the *Next* button. Define the name of the file in accordance to the file declared in the manifest illustrated in Figure 7: *instantiator.xml*. Clicking the *Finish* button will open the XML editor. Switch to the source tab and edit the file as follows:

<?xml version="1.0" encoding="UTF-8"?>

<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"

immediate="true"

name="EASy Example Instantiator">

<implementation class="easyexampleinstantiator.ExampleEngine"/>

<service>

<provide interface="de.uni\_hildesheim.sse.easy\_producer.

instantiator.InstantiatorEngine"/>

</service>

</scr:component>

Figure 8 shows the final XML file. Please note that we used the names and package-structure of our example in Figure 8. Thus, with respect to different implementations the name of the service component in line 4 as well as the package and the class name of the implementation class element in line 6 (the class, which will implement the instantiator) have to be adapted. Please ignore the warning in line 6 as the class currently does not exist. This will be part of Section 3.1.3.



Figure 8: Definition of the service-component for the new instantiator

The previously defined XML file must be included in the binary build. Thus, open the manifest file again and switch to the *Build* tab. In the left lower part of this tab select the *OSGI-INF* folder to be included in the binary build.

The last step is the inclusion of external, third-party libraries – the actual instantiator. Please note that this step is only required if the main implementation of the instantiator or required other required functionalities are implemented in another plug-in. In such a case, build the plug-in first[[10]](#footnote-10). Then, right click on the current instantiator plug-in project, select *New* and *Folder*. The name of the new folder must be *lib*. Include all libraries in this folder that are required by the new instantiator. The folder and the required libraries have to be included in the *Classpath* of the new plug-in. Thus, open the plug-in manifest and switch to the *Runtime* tab. Add the libraries to the *Classpath* by clicking on the *Add…* button on the right side of the *Runtime* tab. Select all required libraries of the *lib* folder and click the *Ok* button. Switch to the *Build* tab of the plug-in manifest and select the *lib* folder to be part of the *Binary Build* in the left lower part of this tab. Figure 9 and Figure 10 show the result in the context of our example. Figure 9 shows the included library de.uni\_hildesheim.sse\_0.0.1.jar, which provides the main functionality of our prototypical instantiator and, thus, has to be available at runtime. Figure 10 illustrates the build configuration in which the library (highlighted) is selected as part of the *Binary Build*.



Figure 9: Classpath specification of external, required libraries



Figure 10: Binary Build selection of external, required libraries

Finally, the plug-in project is set up, configured and ready to use. In the next section, we will further develop this plug-in by implementing instantiator-specific functionality based on the results of this section.

### Instantiator Implementation

In the previous section, we set up the Eclipse plug-in project for implementing a new instantiator for EASy-Producer. In this section, we will describe how to implement the (basic) functionalities of an instantiator. However, as each instantiator provides its individual capabilities and is used for different purposes, this description will only include the basic functionalities that are common to each instantiator.

The first step is to create a new Java class file. Right click on the package that was defined as the implementation class package in Section 3.1.2 and select *New 🡪 Class*. In the emerging *Java Class* wizard, define the name of the new class in accordance to the name of the implementation class (cf. Section 3.1.2). In our example, we use the name *ExampleEngine*. Leave all other options as-is.

Each new instantiator has to extend the *InstantiatorEngine* class of the EASy-Producer instantiator core package. Thus, the next step is to edit the new class file as follows (please note that we use the packages and class names of our example):

package easyexampleinstantiator;

import org.osgi.service.component.ComponentContext;

import de.uni\_hildesheim.sse.easy\_producer.instantiator.\*;

import de.uni\_hildesheim.sse.model.confModel.DecisionVariable;

public class ExampleEngine extends InstantiatorEngine{

protected void activate(ComponentContext context) {

Transformator.addEngine(this);

}

protected void deactivate(ComponentContext context) {

Transformator.removeEngine(this);

}

@Override

protected void addValue2Context(DecisionVariable arg0) {

}

@Override

protected void clearContext() {

}

@Override

protected void initEngine() {

}

@Override

public void instantiate() throws InstantiatorException {

}

}

We will now discuss each of these methods in detail:

* ***activate*:** This method is used to activate the instantiator plug-in. We recommend not changing this method in order to guarantee that EASy-Producer activates the instantiator properly.
* ***deactivate*:** This method is used to deactivate the instantiator plug-in. We recommend not changing this method in order to guarantee that EASy-Producer deactivates the instantiator properly.
* ***addValue2Context*:** This method will be invoked by EASy-Producer in order to pass the all decision variable objects to the instantiator engine. The method is called for each decision variable in the current configuration, while the way of saving or processing this data is up to the engine. For example, define a local list of type *DecisionVariable*, which stores each passed variable in order to process it when the instantiation process is invoked.
* ***clearContext*:** This method will be invoked by EASy-Producer in order to clear the current context of the instantiator engine. The context of an instantiator is all information that will be saved or used in the context of a specific product line project, i.e. the list of decision variables of the current variability model. EASy-Producer treats every instantiator as a singleton and, thus, tries to clear the context of an instantiator each time it will be used. What has to be cleared and how is up to the developer. However, it is important to note, that clearing all project-specific information has to be cleared in order for the instantiation to work properly.
* ***initEngine*:** This method will be invoked by EASy-Producer in order to initialize the instantiator engine. This method can be used to initialize any instantiator-specific attributes, objects, etc. Please note that this method also calls the *clearContext* to guarantee that initialization will be successful. **Important**: In order to invoke the method *addValue2Context*, the method *initEngine* must include the following call to the super-class: **prepareContext()**.
* ***instantiate*:** This method will be invoked by EASy-Producer in case that the instantiation process has been started. Thus, this method includes the core functionalities of the instantiator, i.e. copying specific files or manipulating their content with respect to the values of the decision variables passed by the method *addValue2Context*. For this purpose, the method *getFilesToInstantiate* of the *Instantiator Engine* class can be used to get the list of files this instantiator was assigned to[[11]](#footnote-11).

Figure 11 shows a very simple example of how to use these methods to implement instantiator-specific functionalities. The prototypical *SEEInstantiator* will only display the files that have been assigned to it using EASy-Producer[[12]](#footnote-12). When the instantiator is initialized (*initEngine*), a new instance of the *SSEInstantiator* is created (line 37). This method will also invoke the method *clearContext*, which will clear the private file list of this instantiator (line 32). The file list is used to store the files that have been assigned to the instantiator (line 42) and to pass these files to the actual *SSEInstantiator* to be displayed if instantiation is invoked (line 43). This will open a simple message dialog in EASy-Producer that lists all assigned files of the *SSEInstantiator*.

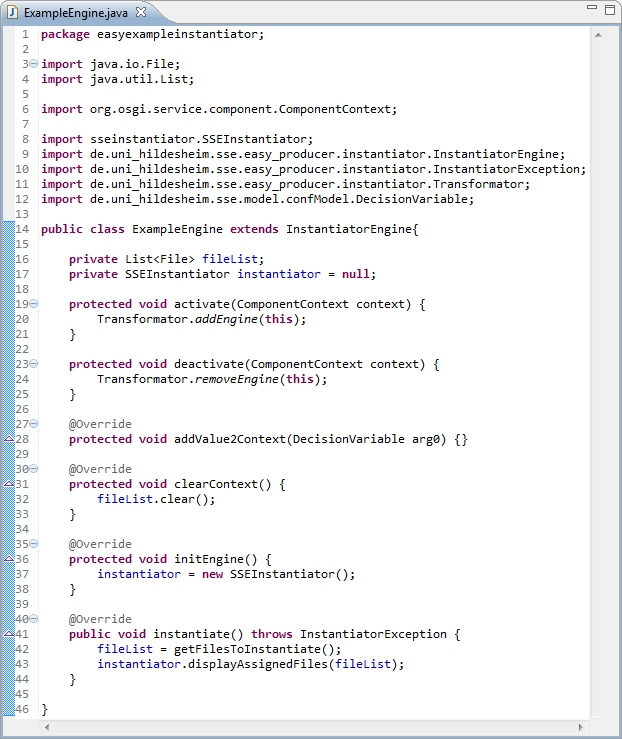


Figure 11: Implementation of instantiator engine functionalities

The last step is to build the plug-in. Open the plug-in manifest file and click on the *Export deployable plug-ins and fragments* button in the upper right corner. In the emerging wizard select the current plug-in project, specify the desired destination[[13]](#footnote-13), and click on the *Finish* button.

Finally, the plug-in and, thus, the instantiator are implemented, build, and ready for use. In the next section, we will describe how to integrate a new instantiator within EASy-Producer. We will also have a quick look on how to use it. However, for detailed information on how to use an instantiator, please consider the [EASy-Producer User Guide](http://projects.sse.uni-hildesheim.de/easy/docs/guide.pdf).

### Instantiator Integration

In the previous section, we implemented the (basic) functionalities of a new instantiator. Further, we build a deployable plug-in, which we will use in this section for integrating the new instantiator within an EASy-Producer installation.

The first only step is to copy the previously build instantiator plug-in into the *dropins* folder of the Eclipse application in which EASy-Producer installed. Start the Eclipse application and create a new product line project: *File 🡪 New 🡪 Other… 🡪 EASy-Producer 🡪 New EASy-Producer Project*. The name of the new project is up to the developer. The product line editor will emerge. Switch to the *Instantiator View* tab, and expand the drop-down menu to choose an instantiator. In this list the new instantiator will appear as illustrated by Figure 12. Add this instantiator by clicking on the *Add Instantiator* button and select the files of the product line project that should be instantiated by this instantiator.

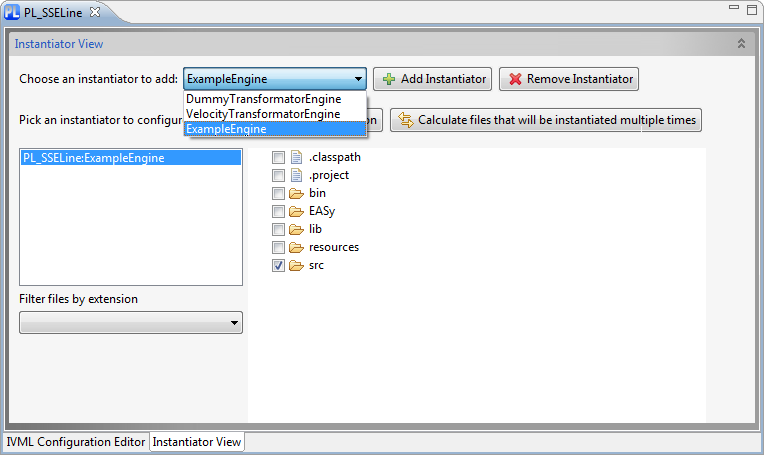


Figure 12: Using the new instantiator

## Implementing a New Reasoner

TODO

The basic mechanism is the same, thus, just write what is different and reference the existing sections as needed!!

# Known Bugs

No known bugs regarding the implementation of extensions.

1. EASy is an abbreviation for Engineering Adaptive Systems. [↑](#footnote-ref-1)
2. Please note that this document is currently under development and will evolve with the evolution of the tool. Changes are documented in the version section. [↑](#footnote-ref-2)
3. Eclipse website: [www.eclipse.org/](http://www.eclipse.org/) [↑](#footnote-ref-3)
4. Xtext website: <http://www.eclipse.org/Xtext/> [↑](#footnote-ref-4)
5. Detailed information regarding the Velocity instantiator can be found in the [EASy-Producer User Guide](http://projects.sse.uni-hildesheim.de/easy/docs/guide.pdf). [↑](#footnote-ref-5)
6. For more information visit: <http://eclipse.org/equinox/> [↑](#footnote-ref-6)
7. For more information visit: [Eclipse API – org.eclipse.core.runtime](http://help.eclipse.org/helios/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fruntime%2Fpackage-summary.html) [↑](#footnote-ref-7)
8. For more information visit: [OSGi API – org.osgi.service.component](http://www.osgi.org/javadoc/r4v42/org/osgi/service/component/package-summary.html) [↑](#footnote-ref-8)
9. If the category *XML* does not exist, install XML support using *Help 🡪 Install New Software* or open the category *General*, select *File*, and define the name as well as the file-type manually. [↑](#footnote-ref-9)
10. If you do not know how to build a plug-in, please consider Section 3.1.3. [↑](#footnote-ref-10)
11. The instantiator core plug-in of EASy-Producer provides a Java documentation including the full list of available methods of the *Instantiator Engine* class. [↑](#footnote-ref-11)
12. We include an external library that provides the *SSEInstantiator* indicated by the import statement in line 6 of Figure 11. How to include external libraries is described in Section 3.1.2. [↑](#footnote-ref-12)
13. The destination is up to the developer. However, we recommend using a location, which is easy to find as we will need the location for integrating the new instantiator in Section 3.1.4. [↑](#footnote-ref-13)