

# **Pepper**

## **Developer's Guide for Pepper modules**

**Florian Zipser <saltnpepper@lists.hu-berlin.de>**

**INRIA**

**SFB 632 Information Structure / D1 Linguistic Database**

**Humboldt-Universität zu Berlin**

**Universität Potsdam**

---

# Pepper: Developer's Guide for Pepper modules

by Florian Zipser, , , and

Version \${project.version}

Copyright © 2012 ??, ??, ??, ??,???, All rights reserved.

---

# Table of Contents

Foreword .....	vi
1. Introduction .....	1
The sample .....	1
2. X minutes to run your first Pepper module .....	3
Setting Up .....	3
Let's have a first run .....	5
Let's give the baby a name .....	6
Let's have a second run .....	8
Congratulations .....	9
3. Just code it .....	10
Mapping document-structure and corpus-structure .....	11
Analyzing an unknown corpus .....	14
Im- and exporting corpus-structure .....	15
Importing corpus-structure .....	16
Exporting corpus-structure .....	17
Customizing the mapping .....	18
Property .....	18
Property registration .....	18
Monitoring the progress .....	20
Logging .....	21
Error handling .....	21
Prepare and clean-up .....	21
4. Testing your Pepper module .....	23
Running Unit Test .....	23
Adopting consistency tests .....	23
Writing your own tests .....	23
Running live tests .....	24
5. Documenting your Pepper module .....	25
Transformation .....	25
6. some little helpers .....	26
xml extractor .....	26

---

## List of Figures

2.1. Eclipse marketplace .....	3
2.2. Eclipse Import dialogue .....	4
2.3. Eclipse .....	5
2.4. Run Configuration dialog .....	5
3.1. class diagram showing the inheritance of Pepper module types .....	10
3.2. class diagram showing the inheritance of Pepper module types .....	11
3.3. corpus-structure represented in file-structure .....	15
3.4. corpus-structure .....	16
3.5. ....	16
3.6. corpus-structure represented in file-structure .....	21

---

## List of Tables

3.1. map of SElementId and corresponding URI locations .....	16
--	----

---

# Foreword

The aim of this document is to provide a helpful guide to create your own Pepper module, that can be plugged into the Pepper framework.

We are trying to make things as easy to use as possible, but we are a non-profit project and we need your help. So please tell us if things are too difficult and help us improving the framework.

You are even very welcome to help us improving this documentation by reporting bugs, requests for more information or by writing sections. Please write an email to `<saltnpepper@lists.hu-berlin.de>`.

Have fun developing with SaltNPepper!

---

# Chapter 1. Introduction

With SaltNPepper we provide two powerful frameworks for dealing with linguistically annotated data. SaltNPepper is an Open Source project developed at the Humboldt University of Berlin (see: <http://www.hu-berlin.de/>) and INRIA (Institut national de recherche en informatique et automatique, see: <http://www.inria.fr/>). In linguistic research a variety of formats exists, but no unified way of processing them. To fill that gap, we developed a meta model called Salt which abstracts over linguistic data. Based on this model, we also developed the pluggable universal converter framework Pepper to convert linguistic data between various formats.

Pepper is a container controlling the workflow of a conversion process, the conversion itself is done by a set of modules called Pepper modules mapping the linguistic data between a given format and Salt and vice versa. Pepper is a highly pluggable framework which offers the possibility to plug in new modules in order to incorporate further formats. The architecture of Pepper is flexible and makes it possible to benefit from all already existing modules. This means that when adding a new or previously unknown format Z to Pepper, it is automatically possible to map data between Z and all already supported formats A, B, C, ... . A Pepper workflow consists of three phases:

1. the import phase (mapping data from a given format to Salt),
2. the optional manipulation phase (manipulating or enhancing data in Salt) and the
3. export phase (mapping data from Salt to a given format).

The three phase process makes it feasible to influence and manipulate data during conversion, for example by adding additional information or linguistic annotations, or by merging data from different sources.

Since Pepper is a pluggable framework, we used an underlying framework called OSGi (see: <http://www.osgi.org/>) that provides such a functionality. OSGi is a mighty framework and has a lot of impact in the way of programming things in java. Because we do not want to force you to learn OSGi, when you just want to create a new module for pepper, we tried to hide the OSGi layer as good as possible. Therefore, and for the lifecycle management of such projects, we used another framework named maven (see: <http://maven.apache.org/>). Maven is configured via an xml file called pom.xml, you will find it in all SaltNPepper projects and also in the root of the SampleModule project. Maven makes things easier for use especially in dealing with dependencies.

The SaltNPepper framework also comes with an environment for a direct start up in Eclipse. for an easier development of a Pepper module. It further contains a test bed for checking simple consistency issues of Pepper modules. This test bed is based on JUnit (see: [junit.org](http://junit.org)) and can easily be extended for a specific module test.

In the following, Chapter 2, *X minutes to run your first Pepper module* explains how to set up your environment to start developing your Pepper module and explains how to download and adopt a template module to your own needs. This module then is the base for your own module. In the section called “Customizing the mapping” we explain how to add properties to your module, so that the user can dynamically customize the behaviour of the mapping. Chapter 5, *Documenting your Pepper module* shows the usage of a documentation template written in doocbook (see <http://www.docbook.org/>) to make a standard documentation for the user of your module. Since testing of software often is a pain in the back, the pepper-testSuite already comes with some predefined tests. They should save you some time. Last but not least, this documentation contains a FAQ. Since a FAQ lives of questions, if you have some, just help us increasing that part and send a mail to [<saltnpepper@lists.hu-berlin.de>](mailto:saltnpepper@lists.hu-berlin.de).

## The sample

Before we start, we want to introduce our sample implementation of a Pepper module. This guide will show the power of Pepper and in parts even of Salt along that sample module project. Therefore it

contains sample implementations, which could be used as a template and a kind of an inspiration for your own project. So don't hesitate to override the specific parts. The project contains three modules, one of each kind: an importer, a manipulator and an exporter. The importer just imports a static corpus containing one super-corpus, two sub-corpora and four documents. Here you can see how to create a corpus-structure and a document-structure<sup>1</sup>, as a shortcut to the Salt framework. The manipulator, traverses over the document-structure and prints out some information about it, like the frequencies of annotations, the number of nodes and edges and so on. What you can see here is how to access data and traverse the graph structure in the Salt framework. The exporter exports the corpus into a format named DOT, which can be used for visualization of the corpus. The main logic of that mapping is not contained in the exporter itself, since such a component is already part of the Salt framework. The exporter should just give an impression of how to deal with the creation of the specific output files. Please note, that these modules are just samples and do not show the entire power of Salt or Pepper. For learning more about Salt, please read the Salt model guide or the Salt quick User's guide. Both are available on [u.hu-berlin.de/saltpepper](http://u.hu-berlin.de/saltpepper).

---

<sup>1</sup>Salt differentiates between the corpus-structure and the document-structure. The document-structure contains the primary data (data sources) and the linguistic annotations. A bunch of such information is grouped to a document (`SDocument` in Salt). The corpus-structure now is a grouping mechanism to group several documents to a corpus or sub-corpus (`SCorpus` in Salt).



---

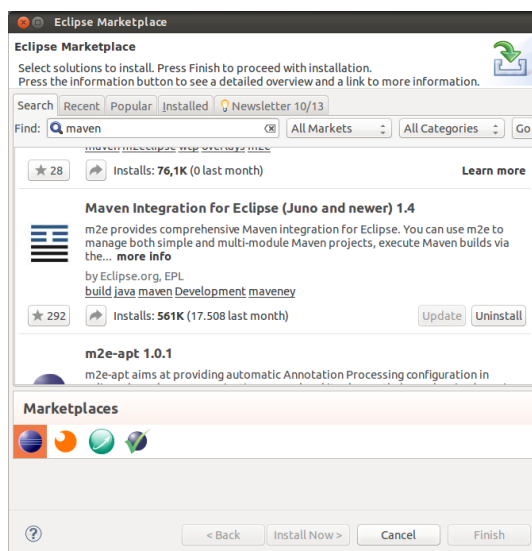
# Chapter 2. X minutes to run your first Pepper module

If you do not belong to the hardcore 'vi' developer community, you may want to use a development environment and an IDE for implementing your Pepper module. For that case we here describe how to set up your environment for the Eclipse IDE (see: <http://download.eclipse.org/>). You also can stick to another IDE like NetBeans (see: [www.netbeans.org/](http://www.netbeans.org/)) but you should make sure that the OSGi framework is set up correctly for your IDE. In that case, skip the Eclipse specific parts.

## Setting Up

1. Download Eclipse <sup>1</sup>
2. Unzip the the Eclipse archive to a location of your choice, lets call it ECLIPSE\_HOME
3. Download the latest version of Pepper (<https://korpling.german.hu-berlin.de/saltnpepper/repository/XXX>)
4. Unzip SaltNPepper to a location of your choice, let's call it PEPPER\_HOME.
5. Copy all files from PEPPER\_HOME/plugins to ECLIPSE\_HOME/dropins
6. When you now open Eclipse, a menu will pop up and asks for a workspace location. Point the location to a folder of your choice. Let's call it WORKSPACE.
7. Now you have an empty Eclipse looking like figure XXX. <sup>2</sup>
8. Since SaltNPepper uses maven (see: <http://maven.apache.org/>) for dependency and lifecycle management, it would make things much easier, to install maven in your Eclipse environment.
9. Open the 'Eclipse Marketplace' via 'Help --> Eclipse Marketplace...' (see Figure 2.1, “Eclipse marketplace”)

**Figure 2.1. Eclipse marketplace**



---

<sup>1</sup>Eclipse is available in several flavours e.g. for web developers, mobile developers, C++ developers etc.. We recommend the Eclipse Modeling Tools, since you might want to create or use a model for the format you want to support. In this documentation, we used Eclipse kepler, version 4.3.1 (see: <http://www.eclipse.org/downloads/packages/eclipse-standard-431/keplers1>).

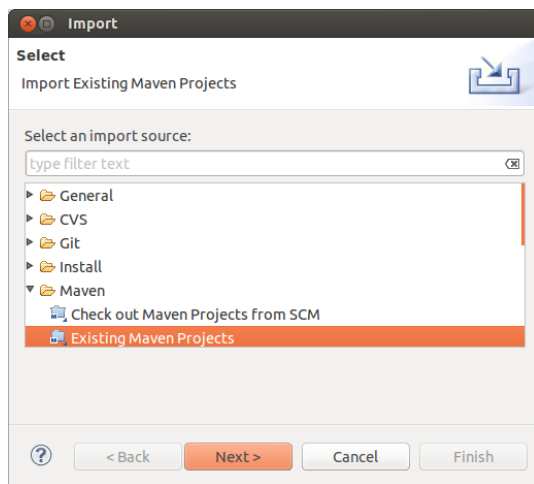
<sup>2</sup>When starting Eclipse for the first time, the first you will see is a welcome tab, just close this tab.

10. Type in 'maven' in text box 'Find:' and press enter (see Figure 2.1, “Eclipse marketplace” position A).
11. The plugin 'Maven integration for Eclipse' should be displayed (you may have to scroll down). Click "Install" to install that plugin (see Figure 2.1, “Eclipse marketplace” position B).
12. During the installation Eclipse asks you to agree to the license of this plugin and it may be recommended to restart Eclipse. Just follow the instructions and go ahead.
13. For an easier start, we provide a sample project which could be used as a template for your own Pepper modules. This project contains in importer, a manipulator and an exporter. The sample project is available via the versioning control system of Pepper and can be downloaded using SVN<sup>3</sup> with the command:  
  

```
svn export https://korpling.german.hu-berlin.de/svn/saltnpepper/pepper/tags/V
```

  
where VERSION stands for a released version of the Pepper framework. Download the project to a location of your choice, let's call it SAMPLE\_HOME.
14. Back in Eclipse, click on the menu item 'File' --> 'Import...'. When the maven integration for Eclipse was set up correctly, you will find an entry named maven in the popped up dialog.
15. Click the entry 'Existing Maven Projects' as shown in figure Figure 2.2, “Eclipse Import dialogue” XXX.

**Figure 2.2. Eclipse Import dialogue**

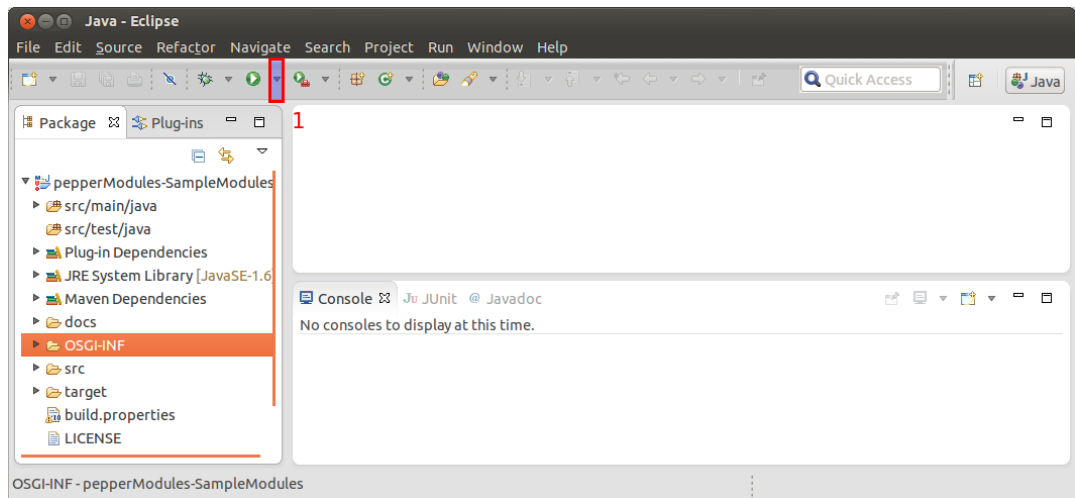


16. In the opening wizard choose SAMPLE\_HOME as location of your maven project and click finish.
17. Now Eclipse should import your first Pepper module project. After all your Eclipse should look very similar to figure Figure 2.3, “Eclipse”).

---

<sup>3</sup>For downloading the sample via SVN, you need an SVN client. Clients are available for all main OS. Most linux distributions already come with a command-line SVN client, also for Windows command-line clients are available, but even the very nice graphical client TortoiseSVN (<http://tortoisesvn.net/>) is available.

**Figure 2.3. Eclipse**



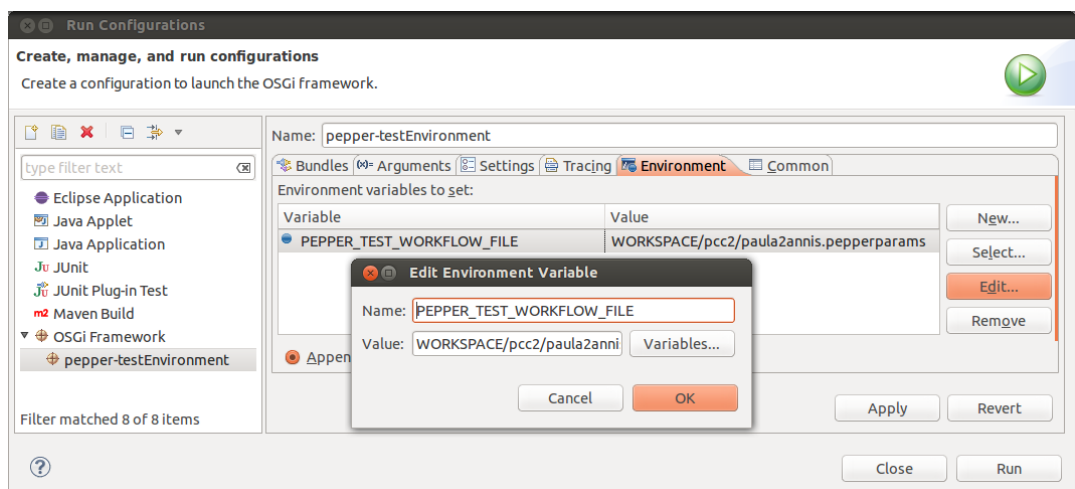
On the left hand side, you can see all active projects in your workspace (which currently is just the pepperModules-SampleModules project). When you open the project (by double-clicking), you can see all files and folders of this project.

## Let's have a first run

To run a project in eclipse, you need a configuration of what should be run and how it should be run. This is called a run configuration.

1. Before you can run the sample module, you need to compile its sources. Since Pepper is using maven, right click on the project and select 'Run as...' --> 'Maven Install'.<sup>4</sup>
2. To open the run configuration dialog, click on the small arrow on the right of the 'play' symbol (see position 1 in figure Figure 2.3, “Eclipse”) and choose menu entry 'Run Configurations...'. Alternative you can click on 'Run'--> 'Run Configurations...'.<sup>4</sup>

**Figure 2.4. Run Configuration dialog**



---

<sup>4</sup>The project now should be compiled. If you cannot see the compilation progress, you need to open the 'Console' view by clicking on 'Window'-->'Show View'-->'Console'

3. Double-click on the entry 'OSGi Framework', a subentry named 'pepper-testEnvironment' will be opened<sup>5</sup>. Double-click on that entry.
4. Now switch to the tab 'Environment'. You will find the definition of an environment variable named 'PEPPER\_TEST\_WORKFLOW\_FILE'. The value of this variable determines, where to find the description file of a Pepper workflow. The sample module comes with a predefined workflow description file at 'SAMPLE\_HOME/src/test/resources/sample.pepperparams'. In this workflow all three modules of the sample module project are in use: an importer for importing a static corpus, a manipulator, to print out some information about the corpus like annotation frequencies and an exporter to export the corpus into dot format.<sup>6</sup>.

Replace the entry with an absolute path to that file.

5. For logging Pepper uses the slf4j framework (see: <http://www.slf4j.org/>). Switch to tab 'Arguments' and type in in text box vm arguments the flag '-Dlogback.configurationFile=SAMPLE\_HOME/src/test/resources/logback\_test.xml'. Make sure to replace SAMPLE\_HOME with an absolute path.
6. Now press the button 'Run'.
7. In optimal case everything is fine and Pepper should give an output similar to this: XXX make a screenshot of expected output XXX

So what has happend?

- a. The frameworks starts and notifies about the registered modules, which are some more, than just our sample modules. This is because Pepper already comes with a small set of integrated modules.
- b. Pepper lists the workflow as defined in the workflow description file.
- c. The SampleImporter imported a predefined corpus and logs some status messages to its progress.
- d. The SampleManipulator outputed some statistical information about that imported corpus like, how many nodes and relations are contained. Which annotations are used and in which frequency they occur.
- e. The SampleExporter exported that corpus in dot format to the path 'SAMPLE\_HOME/target/sampleModuleOutput'.

## Let's give the baby a name

... and configure the project. Configuring means, to adapt the default configurations to your specific project needs like the name of your project, its purpose, the name of your organization and so on.

1. Rename the sample project to a name of your choice like 'pepperModules-MyModules' (when working with Eclipse, right click on the project name and choose the menu entry "Refactor" --> "Rename...")
2. Rename the packages (when working with Eclipse, right click on the package name and choose the menu entry "Refactor" --> "Rename...")

Open the pom.xml in the sampel project by double clicking on it. A specific editor the 'POM editor' will open. On the bottom, a set of tabs is visible, choose the tab 'pom.xml'. An xml editor will open. The following steps correspond to one TODO entry in the file.

1. Change the "groupId" to the name of your working group (conventionally, the groupId is the same as the package name). You will find the entry under '/project/groupId'.

---

<sup>5</sup>If this entry does not appear, open the file 'SAMPLE\_HOME/src/test/resources/run\_sampleProject.launch' by double clicking

<sup>6</sup>The dot format (see: <http://www.graphviz.org/content/dot-language>) is a format for rendering graphs. This format can be converted with for instance the GraphViz tool (see: <http://www.graphviz.org/>) into a svg, png or another graphical format.

2. Change the "artifactId" (to the module name). You will find the entry under '/project/artifactId'.
3. Modify the description: Write what the module is supposed to do. You will find the entry under '/project/description'.
4. Change the project homepage to the url of your project. You will find the entry under '/project/url'.
5. Change the issue tracker to the one you are using, in case that you do not use any one, remove this entry. You will find the entry under '/project/issueManagement'.
6. Change the continuous integration management system to the one you are using, in case that you do not use any one, remove this entry. You will find the entry under '/project/ciManagement'.
7. Change the inception year to the current year. You will find the entry under '/project/inceptionYear'.
8. Change the name of the organization to the one you are working for. You will find the entry under '/project/organization'.
9. Modify the scm information or remove them (the scm specifies the location of your versioning repository like SVN, GIT, CVS etc.). You will find the entry under '/project/scm'.
10. Change the connection to the tags folder of your scm, what you can see here is the subversion connection for the pepperModules-SampleModules project. You will find the entry under '/project/build/plugins/plugin/configuration/tagBase'.
11. If your module needs some 3rd party maven dependencies add them to the dependencies section. (adding 3rd party dependencies is necessary to resolve them via maven). You will find the entry under '/project/dependencies'.
12. Sometimes it is necessary to include libraries, which are not accessible via a maven repository and therefore can not be resolved by maven. In that case we recommend, to create a 'lib' folder in the project directory and to copy all the libraries you need into it. Unfortunately, you have to register them twice, first for maven and second for OSGi.

To register such a library to maven, you need to install them to your local maven repository. You can do that with:

```
mvn install:install-file -Dfile=JAR_FILE -DgroupId=GROUP_ID \
-DartifactId=ARTIFACT_ID -Dversion=VERSION -Dpackaging=PACKAGING
```

Now you need to add the library as a dependency to your pom. The following snippet shows an example:

```
<dependency>
  <groupId>GROUP_ID</groupId>
  <artifactId>ARTIFACT_ID</artifactId>
  <version>VERSION</version>
</dependency>
```

To make them accessible for OSGi, add them to the bundle-classpath of the plugin named 'maven-bundle-plugin'. You will find the entry under '/project/build/plugins/plugin[artifactId/text()='maven-bundle-plugin']/configuration/instructions/Bundle-ClassPath'. You further need to add them to a second element named include-resource, which you will find under '/project/build/plugins/plugin[artifactId/text()='maven-bundle-plugin']/configuration/instructions/Include-Resource'. The following snippet gives an example:

```
<Bundle-ClassPath>., {maven-dependencies}, lib/myLib.jar</Bundle-ClassPath>
```

```
<Include-Resource>{maven-resources}, LICENSE, NOTICE,  
lib/myLib.jar=lib/myLib.jar</Include-Resource>
```

You include libraries not handled by maven, i.e. jar files, by setting the bundle-path and extending the include-resources tag. `"/project/build/plugins/plugin[artifactId/text()='maven-bundle-plugin']/configuration/instructions/Include-Resource"`

Now we come to some configurational changes in the Java source code.

1. In most cases users do not want to implement all three sorts of modules. Users often want to implement only an importer an exporter or a manipulator. Then just delete the classes you won't implement. Otherwise, you will end up with non-functional modules in your project. In case that you want to have several importers, manipulators or exporters, just duplicate and rename the classes.
2. Change the name of the module, for instance to `MyImporter`, `MyExporter` etc. We recommend to use the format name and the ending `Importer`, `Exporter` or `Manipulator` (like `FORMATImporter`).
3. Change the name of the component<sup>7</sup>, for instance use the classes name and add 'Component' to it (e.g. `MyImporterComponent`) like in the following example.

```
@Component (name="MyImporterComponent",  
            factory="PepperImporterComponentFactory")  
public class MyImporter extends PepperImporterImpl implements PepperImporter
```

4. Set the coordinates, with which your module shall be registered. The coordinates (modules name, version and supported formats) are a kind of a fingerprint, which should make your module unique. See the following example:

```
public MyImporterImporter()  
{  
    super();  
    this.setName("MyImporterImporter");  
    //we recommend to synchronize this value with the maven version  
    //in your pom.xml  
    this.setVersion("1.1.0");  
    this.addSupportedFormat("myFormat", "1.0", null);  
    ...  
}
```

## Let's have a second run

Since your project's configuration is now adapted to your needs, you need to re-compile the sources. Right click on the project and select 'Run as...' --> 'Maven Install'. After renaming the project in Eclipse and the pom.xml, you need to adapt the 'Run configurations'.

1. In tab 'Bundles' you need to select the bundle again (having the new name).
2. In the tab 'Arguments' you need to adapt the reference to the slf4j configuration file, by replacing the folder `SAMPLE_HOME` with the new projects name.
3. And the same goes for 'Environment', where you have to adapt the location of the `WORKFLOW_DESCRIPTION_FILE` reference.

Also needs to know about the name change of the module, therefore you need to adapt the workflow file under `'SAMPLE_HOME/src/test/resources/sample.pepperparams'`.

---

<sup>7</sup>a component here is an OSGi service component (for more details see [http://wiki.osgi.org/wiki/Declarative\\_Services](http://wiki.osgi.org/wiki/Declarative_Services))

1. Change the name of the modules and/or the format descriptions.
2. In case you have removed some modules, delete them from the workflow file.

Now your own Pepper module hopefully is able to run inside the Eclipse framework. If not please take a look into section XXXTroubleshootingXXX, if you do not find a solution there, please check the SaltNPepper homepage under [u.hu-berlin.de/saltnpepper](http://u.hu-berlin.de/saltnpepper) or write an e-mail to [saltnpepper@lists.hu-berlin.de](mailto:saltnpepper@lists.hu-berlin.de) [mailto:saltnpepper@lists.hu-berlin.de].

For creating a deliverable assembly of your project, to be plugged into the Pepper framework, open a command line and run:

```
mvn install assembly:single
```

This will create a zip file in the target folder of your project 'YOUR\_PROJECT/target/distribution', which can simply be plugged into Pepper via extracting its content to the plugin folder of Pepper 'PEPPER\_HOME/plugins'.



### Note

If the command mvn was not found, make sure you have installed maven even in your system not only in Eclipse. For more information see the maven site <http://maven.apache.org/>.

## Congratulations

Now you are done, a new Pepper module is running under your name and you can provide it for download. So the work is done, or isn't it? Unfortunately not, now we come to the hardest part ;-), the coding of your tasks. But don't be afraid, the next chapter is supposed to guide you through the jungle and to not losing your path.

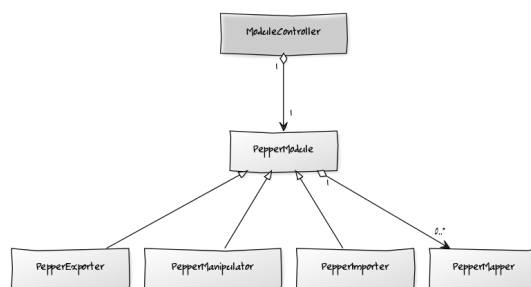
---

## Chapter 3. Just code it

With Pepper we tried to avoid as much complexity as possible without reducing the functionality. We want to enable you to concentrate on the main issues, which are the mapping of objects. But still there are some things you need to know about the framework. Therefore we here introduce some aspects of the Pepper framework and its interaction with a Pepper module. Reducing the complexity is not always possible, but we tried. To manage this trade-off, we followed the approach convention over configuration. That means, we made some assumptions, which apply to many mapping tasks. This makes implementing very simple if the default case matches. But if not, you always have the possibility to adapt the module to your needs. The adaptable default behavior mostly is realized by class derivation and call-back methods, which always can be overridden.

Pepper differentiates three sorts of modules: the importer, the manipulator and the exporter. An importer maps a corpus given in format X to a Salt model. A manipulator maps a Salt model to another Salt model, in terms of changing it or just retrieving some information. An exporter maps a Salt model to a corpus in format Y. All three modules `PepperImporter`, `PepperManipulator` and `PepperExporter` inherit the super type `PepperModule`. So no matter of what kind of module you are going to implement, it must inherit one of the three named types. Figure Figure 3.1, “class diagram showing the inheritance of Pepper module types” shows this relation.

**Figure 3.1. class diagram showing the inheritance of Pepper module types**



Now you might ask, what are the classes `ModuleController` and `PepperMapper` good for. The class `ModuleController` acts as a mediator between the Pepper framework and the concrete Pepper module. It initializes, starts and ends the modules processing. To explain the `PepperMapper` class, we want to give a short motivation: Since a mapping process can be relatively time consuming, we could increase the speed of mapping an entire corpus, if we are able to process mapping tasks simultaneously. Therefore we added mechanisms to run the process multi-threaded. Unfortunately in Java multi-threading is not that trivial and the easiest way to do it is to separate each thread in an own class. This is where `PepperMapper` comes into game. A `PepperModule` object is a singleton instance for one step in the Pepper workflow and divides and distributes the tasks of mapping corpora and documents to several `PepperModule` objects.

One step for a mapping in the Pepper workflow is not just a monolithic block. It consists of several smaller conceptual aspects. Not each of the named aspects is essential, and some are belonging on the type of module you are implementing. Some are optional, some are recommended and some are mandatory to implement. For a better understanding, the rest of this section is structured according these aspects instead of the order of the code.

- mapping document-structure and corpus-structure [mandatory]
- analyzing an unknown corpus [recommended, if module is an importer]
- im- and export corpus-structure [mandatory, if module is an im- or exporter]
- customizing the mapping [recommended]
- error handling [recommended]

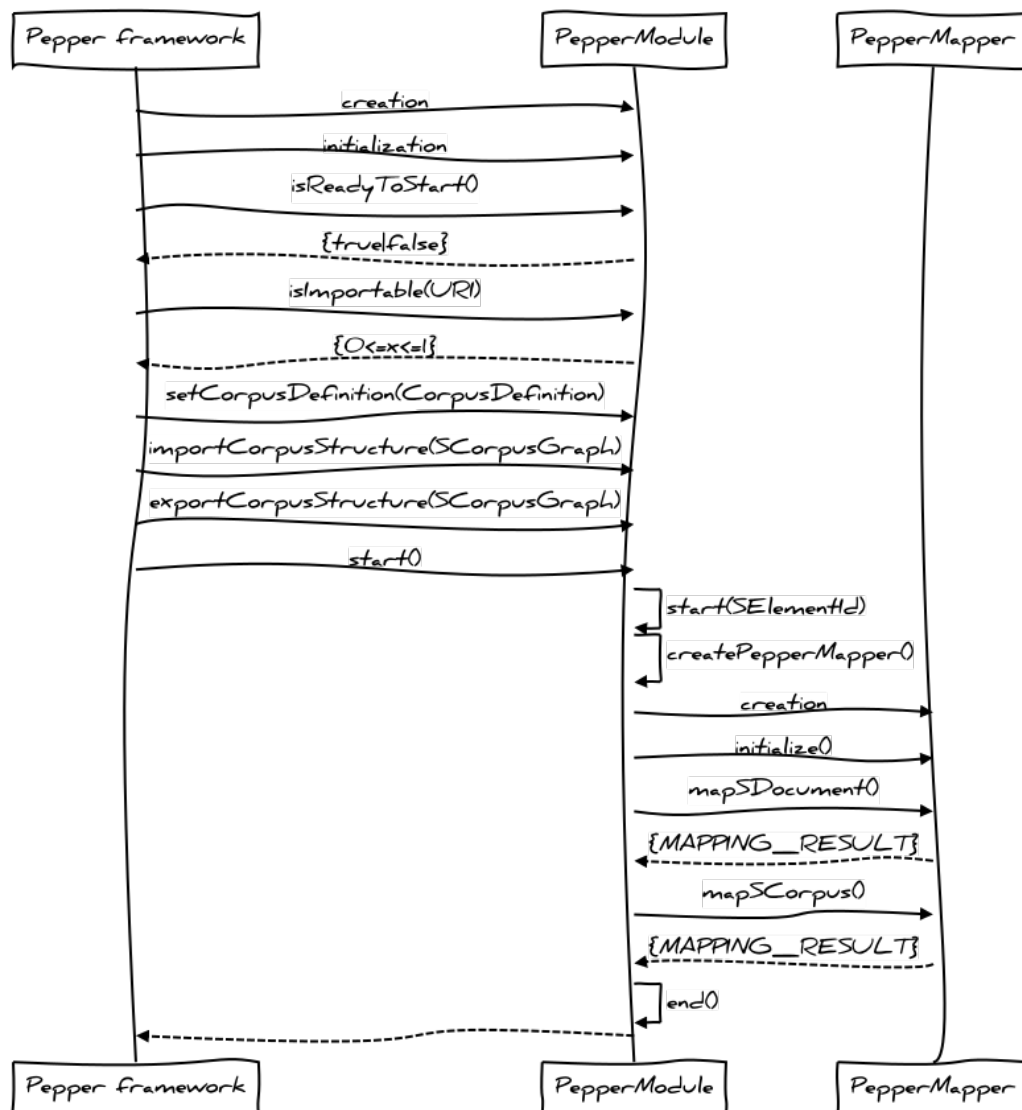


- monitoring the progress [recommended]
- prepare and clean-up [optional]

The main aspect surely is the mapping of the document-structure and corpus-structure. This aspect deals with the creation, manipulation and export of Salt models. In this sense, the others are more sideaspects and not essential for the mapping itself, but important for the workflow.

Some of the aspects are spread over several classes (PepperModule and PepperMapper) and methods. The single paragraphs mention which methods are involved. To get an overview of the entire method stack, figure Figure 3.2, “class diagram showing the inheritance of Pepper module types” illustrates the communication between the framework, the PepperModule and PepperMapper class.

**Figure 3.2. class diagram showing the inheritance of Pepper module types**



## Mapping document-structure and corpus-structure

Remember Salt differentiates between the corpus-structure and the document-structure. The document-structure contains the primary data (data sources) and the linguistic annotations. A bunch

of such information is grouped to a document (SDocument in Salt). The corpus-structure now is a grouping mechanism to group several documents to a corpus or sub-corpus (SCorpus in Salt). Therefore, mapping the document-structure and corpus-structure is the main task of a Pepper module. Normally the conceptual mapping of elements between a model or format X and Salt is the most tricky part. Not necessarily in a technical sense, but in a semantical. For getting a clue how the mapping can be technically realized, we strongly recommend, to read the Salt model guide and the quick user guide on [u.hu-berlin.de/saltnpepper/](http://u.hu-berlin.de/saltnpepper/). We here more focus on the technical part of the Pepper workflow and especially of the Pepper module. But in our Sample module, a lot of templates exist of how to deal with a Salt model. Especially the SampleImporter is full of instructions to create a Salt model. Mapping in this paragraph can mean, mapping a structure from a different model or format to or from Salt (in sense of an im- or exporter) and it can mean a manipulation or just an information retrieval in a Salt model (both in sense of a manipulator).

There are two aspects having a big impact on the inner architecture of a Pepper module for this mapping task. First we have the convention over configuration aspect and second we have the aspect of parallelizing a mapping job. This results in a relatively long stack of function calls to give you an intervention option on several points. We come to this later. But if you are happy, with the default mechanism, it is rather simple to implement your module. Again, the PepperModule is a singleton instance for each Pepper step, whereas there is one instance of PepperMapper per SDocument and SCorpus object in the workflow.

Enough of words, let's dig into the code. Have a look at the method of the following snippet, which is part of each PepperModule:

```
public PepperMapper createPepperMapper(SElementId sElementId){
    SampleMapper mapper= new SampleMapper();
    //1: module is an im-or exporter? passing the ephysical location to mapper
    mapper.setResourceURI(getSElementId2ResourceTable().get(sElementId));
    //2: differentiate between documents and corpora
    if (sElementId.getSIdentifiableElement() instanceof SDocument){
        //do some specific stuff for documents
    }else if (sElementId.getSIdentifiableElement() instanceof SCorpus){
        //do some specific stuff for corpora
    }
    return(mapper);
}
```

This method is supposed to provide a new instance of a specialized PepperMapper. Here you have the chance to make some initializations of your mapper object. Although the main initializations, necessary for the workflow (e.g. passing the customization properties, see section the section called “Customizing the mapping”) are done by Pepper in the back, this is the place to make some specific configurations depending on your implementation. If your module is an im- or exporter, it might be necessary to pass the physical location of that file or folder where the Salt model is supposed to be imported from or exported to (see position 1 in the code). Sometimes it might be necessary to differentiate the type of object which is supposed to be mapped (either an SCorpus or SDocument object). This is shown in the snippet under position 2. That's all we have to do in class PepperModule for the mapping task, now we come to the class PepperMapper. Here you find three methods, supposed to be overridden, as shown in the following snippet.

```
public class SampleMapper implements PepperMapperImpl {

    @Override
    protected void initialize(){
        //do some initializations
    }

    @Override
    public DOCUMENT_STATUS mapSCorpus() {
        //1: returns the resource in case that a module is an importer or exporter
    }
}
```

```
getResourceURI();
//2: getSCorpus() returns the SCorpus object, which for instance can be a
getSCorpus().createSMetaAnnotation(null, "author", "Bart Simpson");
//3: returns that the process was successful
return(DOCUMENT_STATUS.COMPLETED);
}

@Override
public DOCUMENT_STATUS mapSDocument() {
    //4: returns the resource in case that the module is an importer or exporter
    getResourceURI();
    //5: getSDocument() returns the SDocument
    getSDocument().setSDocumentGraph(SaltFactory.eINSTANCE.createSDocumentGraph());
    STextualDS primaryText= getSDocument().getSDocumentGraph()
        .createSTextualDS("Is this example more complicated "
            + "than it appears to be?");
    //6: returns that the process was successful
    return(DOCUMENT_STATUS.COMPLETED);
}
}
```

Not very surprising, the method 'initialize()' is invoked by the constructor and should do some initialization stuff if necessary. The methods 'mapSCorpus()' and 'mapSDocument()' are the more interesting ones. Here is the place to implement the mapping of the corpus-structure or the document-structure. Note, that one instance of the mapper always processes just one object, so either a SCorpus or a SDocument object. If your module is of type im- or exporter, it might be important to know, where the corpus or document has to be read from or stored to. If you set the physical location at position 1 in method 'createPepperMapper()', you can now get that location via calling 'getResourceURI()' as shown on position 1 and XXX (of the current snippet). This method returns a URI pointing to the physical location.



### Note

If your module is an exporter, that location does not physically exist and has to be created on your own.

Position 2 shows, how to access the current SCorpus object and how to annotate it for instance with a meta-annotation (in this sample, the meta-annotation is about an author having the name 'Bart Simpson', the null-value means, that no namespace is used). In method 'mapSDocument()', on position 5, you can access the current object (here it is of type SDocument) with 'getSDocument()'. If your module is an importer, you need to create a container for the document-structure, a SDocumentGraph object. The snippet further shows the creation of a primary text and the creation of a meta-annotation. In Salt each object can be annotated or meta-annotated, so do the SDocument objects. Last but not least, both methods have to return a value describing whether the mapping was successful or not. The returned value can be one of the following three:

- DOCUMENT\_STATUS.COMPLETED - means, that a document or corpus has been processed successfully.
- DOCUMENT\_STATUS.FAILED - means, that the corpus or document could not be processed because of any kind of error.
- DOCUMENT\_STATUS.DELETED - means, that the document or corpus was deleted and shall not be processed any further (by following modules).

During the mapping it is very helpful for the user, if you give some progress status from time to time. Especially when a mappings takes a longer term, it will keep the user from a frustrating experience have a not responding tool. More information on that, you find in paragraph the section called "Monitoring the progress".

That's it. That's it with the mapping of the document-structure and corpus-structure. The rest of this paragraph just handles, ways to not use the default mechanisms and to make more adaptations.

In a few cases, a format does not allow or difficulty allow to have a parallelized mapping. In that case you can switch-off the parallelization in your constructor with

```
setIsMultithreaded(false);
```

If you wondered what we meant, when we said there is a 'long stack of function calls', here is the answer. The Pepper framework does not directly call the method 'createMapper(SElementId)'. The following excerpt illustrates the stack.

```
/** Directly called by Pepper framework,
    waits until a further document or corpus
    can be processed and delegates it */
@Override
public void start(){
    ...
    SElementId sElementId= getModuleController().next().getDocumentId();
    start(sElementId);
    ...
}

/** Only takes control of passed document
    or corpus and creates a mapper object per each*/
@Override
public void start(SElementId sElementId){
    ...
    PepperMapper mapper= createPepperMapper(sElementId);
    ...
}

/** Creates and initilizes a PepperMapper instance */
@Override
public PepperMapper createPepperMapper(SElementId sElementId){
    ...
}
```

Even these two methods could be overridden by your module, to adapt their functionality on different levels.



### Note

Take care when overriding one them, since they handle some more functionality than explained here in this guide. To get clue of happens there, please take a look into the source code. It might be well documented and hopefully is understandable. But if questions occur, please send a mail to [saltnpepper@lists.hu-berlin.de](mailto:saltnpepper@lists.hu-berlin.de).

## Analyzing an unknown corpus

The experience has shown, that a lot of users, do not care a lot about formats and don't want to. Unfortunately, in most cases it is not possible to not annoy the users with the details of a mapping. But we want to reduce the complexity for the user as much as possible. Since most users are not very interested, in the source format of a corpus, they just want to bring the corpus into any kind of tool to make further annotations or analysis. Therefore Pepper provides a possibility to automatically detect the source format of a corpus. Unfortunately this task is very dependent of the format and the module processing the format. That makes the detection a task of the modules implementor. We are sorry.

The mechanism of automatic detection is not a mandatory task, but it is very useful, which makes it recommended.

The class `PepperImporter` defines the method `isImportable(Uri corpusPath)` which can be overridden. The passed uri locates the entry point of the entire corpus as given in the Pepper workflow definition (so it points to the same location as `getCorpusDefinition().getCorpusPath()` does). Depending on the formats you want to support with your importer the detection can be very different. In the simplest case, it only is necessary, to search through the files at the given location (or to recursive traverse through directories, in case of the the location points to a directory), and to read their header section. For instance some formats like the xml formats PAULA (see: <http://www.sfb632.uni-potsdam.de/en/paula.html>) or TEI (see: <http://www.tei-c.org/Guidelines/P5/>) starts with a header section like

```
<?xml version="1.0" standalone="no"?>
<paula version="1.0">
<!-- ... -->
```

or

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- ... -->
<TEI xmlns="http://www.tei-c.org/ns/1.0">
<!-- ... -->
```

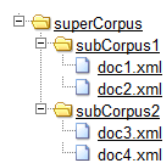
. Formats where reading only the first lines will bring information about the format name and its version make automatic detection very easy. The method `isImportable(Uri corpusPath)` shall return 1 if the corpus is importable by your importer, 0 if the corpus is not importable or a value between  $0 < X < 1$ , if no definitive answer is possible. The default implementation returns null, what means that the method is not overridden. This results in that the Pepper framework will ignore the module in automatic detection phase.

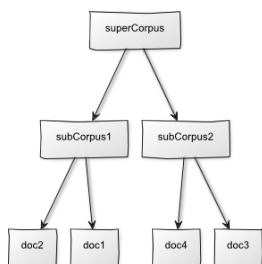
## Im- and exporting corpus-structure

The classes `PepperImporter` and `PepperExporter` provide automatic mechanism to im- or export the corpus-structure. This mechanism is adoptable step by step, according to your specific purpose. In many cases, the corpus-structure is simultaneous to the file structure of a corpus. Since many formats do not care about the corpus-structure, they only encode the document-structure.

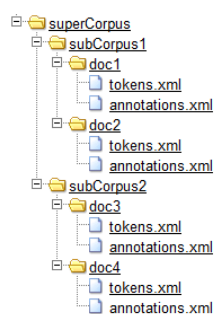
Pepper's default mapping maps the root folder (the direct URI location) to a root-corpus (`SCorpus` object). A sub-folder than corresponds to a sub-corpus (`SCorpus` object). The relation between super- and sub-corpus, is represented as a `SCorpusRelation` object. Following the assumption, that files contain the document-structure, there is one `SDocument` corresponding to each file in a sub-folder. The `SCorpus` and the `SDocument` object are linked with a `SCorpusDocumentRelation`. To get an impression of the described mapping, figure Figure 3.3, “corpus-structure represented in file-structure” shows a file structure and figure Figure 3.4, “corpus-structure” shows the corresponding corpus-structure.

**Figure 3.3. corpus-structure represented in file-structure**



**Figure 3.4. corpus-structure**

Other formats do not encode the document-structure in just one file, they use a bunch of files instead. In that case the folder containing all the files (let's call it leaf-folder) corresponds to a `SDocument` object. Figure Figure 3.5, “” shows an example for these kind of file-structure, which corresponds also corresponds to the corpus-structure of figure Figure 3.4, “corpus-structure”.

**Figure 3.5.**

For keeping the correspondance between the corpus-structure and the file-structure, both the im- and the export of the corpus-structure make use of a map storing this correspondance. Corresponding to figure the corpus-structure of figure Figure 3.4, “corpus-structure” and the file-structure of figure Figure 3.3, “corpus-structure represented in file-structure” table Table 3.1, “map of `SElementId` and corresponding URI locations” shows the stored correleation between them.

**Table 3.1. map of `SElementId` and corresponding URI locations**

<code>salt://corp1</code>	<code>/superCorpus</code>
<code>salt://corp1/subCorpus1</code>	<code>/superCorpus/subCorpus1</code>
<code>salt://corp1/subCorpus1#doc1</code>	<code>/superCorpus/subCorpus1/doc1.xml</code>
<code>salt://corp1/subCorpus1#doc2</code>	<code>/superCorpus/subCorpus1/doc2.xml</code>
<code>salt://corp1/subCorpus2</code>	<code>/superCorpus/subCorpus2</code>
<code>salt://corp1/subCorpus2#doc3</code>	<code>/superCorpus/subCorpus2/doc3.xml</code>
<code>salt://corp1/subCorpus2#doc4</code>	<code>/superCorpus/subCorpus2/doc4.xml</code>

In the following two sections, we are going to describe the import and the export mechanism separately.

## Importing corpus-structure

The default mechanism of importing a corpus-structure is implemented in the method

```
importCorpusStructure(SCorpusGraph corpusGraph)
```

. For a totally change of the default behavior just override this method. To adapt the behavior as described in the following, this has to be done before the method 'importCorpusStructure' was called, so for instance in the constructor or in method

```
isReadyToStart()
```

. Back to figure Figure 3.3, “corpus-structure represented in file-structure”, the import mechanism traverses the file-structure beginning at the super-folder via the sub-folders to the files and creates a `SElementId` object corresponding to each folder or file to fill the map of table Table 3.1, “map of `SElementId` and corresponding URI locations”. This map is necessary for instance to retrieve the physical location of a document-structure during the mapping and can be accessed as shown in the following snippet:

```
public PepperMapper createPepperMapper(SElementId sElementId){
    ...
    mapper.setResourceURI(getSElementId2ResourceTable().get(sElementId));
    ...
}
```

The import mechanism can be adapted by two parameters (or more precisely two lists). An ignore list containing file endings, which are supposed to be ignored during the import and a list of file ending which are supposed to be used for the import.<sup>1</sup> Now let's show some code for adapting. The following snippet is placed into the method 'isReadyToStart()', but even could be located in inside the constructor:

```
public boolean isReadyToStart(){
    ...
    //option 1
    getSDocumentEndings().add(ENDING_XML);
    getSDocumentEndings().add(ENDING_TAB);
    //option 2
    getSDocumentEndings().add(ENDING_ALL_FILES);
    getIgnoreEndings().add(ENDING_TXT);
    //option 3
    getSDocumentEndings().add(ENDING_LEAF_FOLDER);
    ...
}
```

In general the paramter of the method 'getSDocumentEndings()' is just a String, but there are some predefined ones you can use. The two lines marked as option 1, will add the endings 'xml' and 'tab' to the list of file endings to be imported. That means, that all files having one of these endings will be read and mapped to a document-structure. The first line of option 2, means to read each file, no matter on its ending. But the following line excludes all files having the ending 'txt'. Last but not least option 3 which is supposed to treat leaf-folders as document-structures and to create one `SDocument` object for each leaf-folder and not for each file, as mentioned in figure Figure 3.5, “”.

## Exporting corpus-structure

Similar to the import of the corpus-structure for the export, we provide a default behavior and possibilities for adaption. The export of the corpus-structure is handled in the method

```
exportCorpusStructure()
```

and invoked on top at the method 'start()' of the `PepperExporter`. For a totally change of the default behavior just override this method. The aim of this method is to fill the map between corpus-structure and file-structure (see table Table 3.1, “map of `SElementId` and corresponding URI locations”).



### Note

The file-structure is automatically created, there are just URIs pointing the virtual file or folder. The creation of the file or folder has to be done by the Pepper module itself in method 'mapSCorpus()' or 'mapSDocument()'.

<sup>1</sup>In case you are wondering, yes this sounds a bit strange, since each file ending which is not contained in the second list won't be imported by default. But there is an option, to set this to import each file, no matter on the ending.

To adapt the creation of this 'virtual' file-structure, you first have to choose the mode of export. You can do this for instance in method 'readyToStart()', as shown in the following snippet. But even in the constructor as well.

```
public boolean isReadyToStart(){
    ...
    //option 1
    setExportMode(EXPORT_MODE.NO_EXPORT);
    //option 2
    setExportMode(EXPORT_MODE.CORPORA_ONLY);
    //option 3
    setExportMode(EXPORT_MODE.DOCUMENTS_IN_FILES);
    setSDocumentEnding(ENDING_TAB);
    ..
}
```

In this snippet, option 1 means that no mapping will be created. Option 2 means that only `SCorpus` objects are mapped to a folder and `SDocument` objects will be ignored. And option 3 means but that `SCorpus` objects are mapped to a folder and `SDocument` objects are mapped to a file. The ending of that file can be determined by passing the ending to method 'setSDocumentEnding(String)'. In the given snippet a URI having the ending 'tab' is created for each `SDocument`.

## Customizing the mapping

When creating a mapping, it is often a matter of choice to map some data this way or another. In such cases it might be clever not to be that strict and allow only one possibility. It could be beneficially to leave this decision to the user. Customizing a mapping will increase the power of a Pepper module enormously, since it can be used for wider range of purposes without rewriting parts of it. The Pepper framework provides a property system to access such user customizations. Nevertheless, a Pepper module shall not be dependant on user customization. The past showed, that it is very frustrating for a user, when a Pepper module breaks up, because of not specified properties. You should always define a default behaviour in case that the user has not specified one.

## Property

A property is just an attribute-value pair, consisting of a name so called property name and a value so called property value. Properties can be used for customizing the behaviour of a mapping of a Pepper module. Such a property must be specified by the user and determined in the Pepper workflow description. The Pepper framework will pass all customization properties directly to the instance of the Pepper module.



### Note

In the current version of Pepper, one has to specify a property file by its location in the Pepper workflow description file (.pepperParams) in the attribute @specialParams inside the <importerParams>, <exporterParams> or <moduleParams> element. In the next versions this will change to a possibility for adding properties directly to the Pepper workflow description file.

## Property registration

The Pepper framework provides a registry mechanism to customize properties. This registry is called `PepperModuleProperties` and can be accessed via `getProperties()` and `setProperties()`. This class only represents a container object for a set of `PepperModuleProperty` objects and provides accessing methods. An instance of `PepperModuleProperty` represents an abstract description of a property and the concrete value at once. In the registration phase it belongs to the tasks of a `PepperModule` to specify the abstract



description which consists of the name of the property, its datatype, a short description and a flag specifying whether this property is optional or mandatory. To create such an abstract description of a property use the constructor:

```
PepperModuleProperty(String name,  
                      Class<T> clazz,  
                      String description,  
                      Boolean required);
```

and pass the created property object to the property registry by calling the method `addProperty`. The Pepper framework uses the registry to first inform the user about usable properties for customization and second to fulfill the property objects with the property values set by the user.

The value of a specific property can be accessed by passing its name to the registry. The method to be used is the following one:

```
getProperty(String propName);
```

The easiest way of creating an own class for handling customization properties is to derive it from the provided class `PepperModuleProperties`. Imagine you want to register a property named 'MyProp' being of type `String`, which is mandatory to a property class called 'MyModuleProperties'. For having an easier access in your Pepper module, you can enhance the `MyModuleProperties` class with a getter method for property `MyProp` (see: `getMyProp()`).

```
//...  
import de.hu_berlin.german.korpling.saltnpepper.pepper  
    .pepperModules.PepperModuleProperties;  
import de.hu_berlin.german.korpling.saltnpepper.pepper  
    .pepperModules.PepperModuleProperty;  
//...  
public class MyModuleProperties extends PepperModuleProperties {  
    //...  
    public MyModuleProperties(){  
        //...  
        this.addProperty(new PepperModuleProperty<String>  
            ("MyProp", String.class, "description of MyProp", true));  
        //...  
    }  
    //...  
    public String getMyProp(){  
        return((String)this.getProperty("MyProp").getValue());  
    }  
}
```

## Checking property constraints

Since the value of a property can be required, you can check whether its value is set by calling the method `checkProperties()`. To customize the constraints of a property, you can override the method `checkProperty(PepperModuleProperty<?>)`. Imagine a property named 'myProp' having a file as value, you might want to check its existence. The following snippet shows the code how this could be done:

```
public boolean checkProperty(PepperModuleProperty<?> prop){  
    //calls the check of constraints in parent,  
    //for instance if a required value is set  
    super.checkProperty(prop);  
    if ("myProp".equals(prop.getName())){  
        File file= (File)prop.getValue();
```

```
        //throws an exception, in case that the file does not exist
        if (!file.exists()){
            throw new PepperModuleException("The file "+
                "set to property 'myProp' does not exist.");
        }
    }
    return(true);
}
```

## Initializing MyModuleProperties

Last but not least, you need to initialize your property object. The place for best doing that is the constructor of your module. Such an early initialization ensures, that the Pepper framework will use the correct object and will not create a general `PepperModuleProperties` object. Initialize your property object via calling:

```
this.setProperties(new MyModuleProperties());
```

## Monitoring the progress

What could be more annoying than a not responding program and you do not know if it is still working or not? A conversion job could take some time, which is already frustrating enough for the user. Therefore we want to keep the frustration of users as small as possible and give him a precise response about the progress of the conversion job.

Unfortunately, although Pepper is providing a mechanism to make the monitoring of the progress as simple as possible, it remains the task of each module implementor. So you ;-). But don't get afraid, monitoring the progress just means the call of a single method in best case.

If you are using the default mapping mechanism by implementing the class `PepperMapper`, this class provides the methods `addProgress(Double progress)` and `setProgress(Double progress)` for this purpose. Both methods have a different semantic. `addProgress(Double progress)` will add the passed value to the current progress, whereas `setProgress(Double progress)` will override the entire progress. The passed value for progress must be a value between 0 for 0% and 1 for 100%. It is up to you to call one of the methods in your code and to estimate the progress. Often it is easier not to estimate the time needed for the progress, than to divide the entire progress in several steps and to return a progress for each step. For instance the following sample separates the entire mapping process into five steps, which get the same rank of progress.

```
//...
//map all STextualDS objects
addProgress(0.2);
//map all SToken objects
addProgress(0.2);
//map all SSpan objects
addProgress(0.2);
//map all SStruct objects
addProgress(0.2);
//map all SPointingRelation objects
addProgress(0.2);
//...
```



### Note

When using `PepperMapper`, you only have to take care about the progress of the current `SDocument` or `SCorpus` object you are processing. The aggregation of all currently processed objects (`SDocument` and `SCorpus`) will be done automatically.

In case that you do not want to use the default mechanism, you need to override the methods `getProgress(SElementId sDocumentId)` and `getProgress()` of your implementation of `PepperModule`. The method `getProgress(SElementId sDocumentId)` shall return the progress of the `SDocument` or `SCorpus` objects corresponding to the passed `SElementId`. Whereas the method `getProgress()` shall return the aggregated progress of all `SDocument` and `SCorpus` objects currently processed by your module.

## Logging

Another form of Monitoring is the logging, which could be used for passing messages to the user or passing messages to a file for debugging. The logging task in Pepper is handled by the SLF4J (see: <http://www.slf4j.org/>). SLF4J is a logging framework, which provides an abstraction for several other logging frameworks like log4j (see: <http://logging.apache.org/log4j/2.x/>) or java.util logging. Via creating a static logger object you can log several debug levels: trace, debug, info and error.

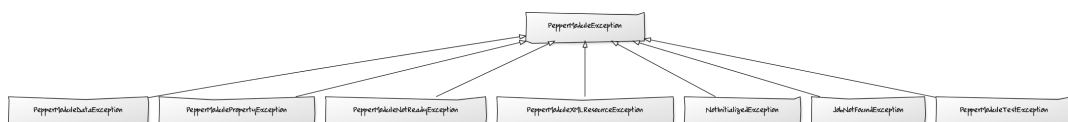
```
private static final Logger logger= LoggerFactory.getLogger(SampleImporter.class);
logger.trace("messages for the implementor");
logger.debug("message for the implementor and user");
logger.info("messages for the user");
logger.error("messages in case of an exception");
```

## Error handling

Another important aspect of monitoring, is the monitoring in case of an error occurred. Even when the module crashes and the conversion could not be ended successfully, the user needs a feedback of what has happened. The main question would be is it a bug in code or a bug in the data. In both cases, the user needs a precisely description. Either to notify you, the module developer or to find the bug in the data. And believe me, a `NullPointerException` is not very useful to the user and very frustrating.

Pepper provides a hierarchy of Exception classes to be used for different purposes, for instance to describe problems in customization properties, the data or general problems in module. Figure 3.6, “corpus-structure represented in file-structure” gives an overview over the Exception classes for modules.

**Figure 3.6. corpus-structure represented in file-structure**



The main and general class `PepperModuleException` can be used in case of the Exception does not match to one of the more specific types. When initializing a `PepperModuleException` or one of its subclasses, you can pass a `PepperModule` or a `PepperMapper` object. The exception itself will expand the error message with this parameter. A more detailed description of the Exceptions could be found in the JavaDoc.

When an exception was thrown for a single document, Pepper will not break up the entire conversion process. Pepper will set the status of this document to failed and will remove it from the rest of the workflow, so that modules coming afterwards will not process the corrupted document. But, Pepper will give a feedback to the user containing the error message provided by the module.

## Prepare and clean-up

The aspect of initialization is spread over two methods in a Pepper module. First the constructor of a module and second the method `isReadyToStart()`. Both methods have been touched already in the other paragraphs, but here we want to give a bundled overview about things concerning the

initialization. Let us start with an explanation why there are two methods. Sometimes, it might be necessary, to read some configuration files for the initialization, but their location is passed by the Pepper framework. Such locations can be accessed with the method 'getResources()'. Unfortunately, this information can only be set after a Pepper module was created, so after the constructor was called. Therefore we need a possibility for initialization at a later point. The method 'isReadyToStart()' fulfills two tasks, first the initialization task and second, it returns a boolean value to determine, if the module can be started or if some things went wrong. If you now wonder, where should be the best location, to do your initialization, we recommend an early-as-possible approach. The following snippet shows a sample initialization:

```
public SampleImporter(){
    super();
    setName("SampleImporter");    //setting name of module
    setVersion("1.1.0");          //setting version of module
    addSupportedFormat("sample", "1.0", null);    //supported formats
    setProperties(new SampleProperties()); //using an own properties object
}

@Override
public boolean isReadyToStart(){
    //access the passed resource folder
    File resourceFolder= new File(getResources().toFileString());
    ...
    getSDocumentEndings().add(some value retrieved from resource folder);
}
```

In this sample, the file ending is set in 'isReadyToRun()' method, since it depends on any configuration in the resource folder<sup>2</sup>. Other reasons could be dependencies of the initialization from the passed customizations, which are also set after the constructor was called.

Sometimes it might be necessary to make a clean up, after the module did the job. For instance when writing an im- or an exporter it might be necessary to close file streams, a db connection etc. Therefore, after the processing is done, the Pepper framework calls the method described in the following snippet:

```
@Override
public void end(){
    super.end();
    //do some clean up like closing of streams etc.
}
```

To run your clean up, just override it and your done.

---

<sup>2</sup>All files in the folder 'SAMPLE\_HOME/src/main/resources' will be copied to the resource location in a modules distribution.

---

# Chapter 4. Testing your Pepper module

## Running Unit Test

In every good book about computer programming it is written that testing the software you are developing is a very important issue. Since testing increases the quality of software enormously, we agree to that. Spending time on developing test may seem wasted, but it will decrease development time in the long run. Therefore we help you to write test code faster by providing a test suite skeleton. In the project `pepper-moduleTests` (part of project `pepper-testSuite`) you will find three test classes named `PepperManipulatorTest`, `PepperImporterTest` and `PepperExporterTest`. These classes use the JUnit test framework (see: [junit.org](http://junit.org)) and implement some very basic tests for checking the consistency of a Pepper module. Just benefit from these classes by creating an own test class derived from one of the provided ones and your tests will be ran during the maven build cycle. For getting an immediate feedback, you can also run them directly in your development environment by running a JUnit task. On the one hand the test classes provide tests which can be adopted to your need and check if your module can be plugged into the Pepper framework (by checking if necessary values are set like the supported format for an im- or exporter). And on the other hand, they provide some helper classes and functions, which can be used when adding further test methods for checking the functionality of your module.



### Note

We strongly recommend, to add some module specific test methods, to increase the quality of your code and even to make it easier changing things and still having a correctly running module.

## Adopting consistency tests

In case of you are implementing an im- or exporter, you need to set the supported formats and versions in your test case. Pepper will check them automatically, but the test environment need to know the correct pairs of format name and version. To do so, just add the following lines to method `setUp()`:

```
protected void setUp() throws Exception
{
    //...
    FormatDefinition formatDef= PepperModulesFactory
        .eINSTANCE.createFormatDefinition();
    formatDef.setFormatName("FORMAT_NAME");
    formatDef.setFormatVersion("FORMAT_VERSION");
    this.supportedFormatsCheck.add(formatDef);
    //...
}
```

Replace `FORMAT_NAME` and `FORMAT_VERSION`, with your specific ones. You can even add more than one `FormatDefinition` object.

## Writing your own tests

The provided test classes are faking the Pepper environment, so that you can run your entire module and just check the in- or output.

In case that you write an importer, you can create an input file containing a corpus in the format you want to support, run your importer and check its output against a defined template. The test will return a processed Salt graph which can be checked for its nodes, edges and so on. The following snippet shows how to read a document and how to check the returned Salt model:

```
public void testSomeTest()
{
    //start: creating and setting corpus definition
    CorpusDefinition corpDef= PepperModulesFactory
        .eINSTANCE.createCorpusDefinition();
    FormatDefinition formatDef= PepperModulesFactory
        .eINSTANCE.createFormatDefinition();
    formatDef.setFormatName(FORMAT_NAME);
    formatDef.setFormatVersion(FORMAT_VERSION);
    corpDef.setFormatDefinition(formatDef);
    corpDef.setCorpusPath(URI.createFileURI(PATH_TO_SAMPLE_CORPUS));
    this.getFixture().setCorpusDefinition(corpDef);
    //end: creating and setting corpus definition

    //...

    //create an empty corpus graph, which is filled by your module
    SCorpusGraph importedSCorpusGraph= SaltCommonFactory
        .eINSTANCE.createSCorpusGraph();
    // add the corpus graph to your module
    this.getFixture().getSaltProject().getSCorpusGraphs()
        .add(importedSCorpusGraph);

    //run your Pepper module
    this.start();

    //check the processed corpus graph object
    //checks that the corpus graph object
    //is not empty any more
    assertNotNull(importedSCorpusGraph.getSCorpora());
    //checks for instance, that the corpus
    //graph contains X SCorpus objects
    assertEquals(X, importedSCorpusGraph.getSCorpora().size());
    //...
}
```

For testing an exporter, you may want to use the sample creator for Salt models, which is part of the Salt project. You will find it as a subproject named salt-saltSample. Here you will find a lot of methods creating a sample Salt model with the possibility to just create the layers, you want to use.

## Running live tests

For running an integration test of your module in the Pepper framework and for testing interferences between your module and other modules, Pepper comes with a test environment. Therefore, you need to set up the OSGi environment correctly and set some environment variables. If you have not done that already, please check #sec\_letsRun.

---

# Chapter 5. Documenting your Pepper module

One of the most important but often forgotten tasks when creating a Pepper module is to document the behaviour of it and its functionalities. Therefore the project SampleModules contains a template for creating a documentation. The documentation in SaltNPepper in general is done in DocBook (see: [docbook.org/](http://docbook.org/)). DocBook is a documentation language written in XML and enables to transform the documentation into several target formats like html, pdf, odt, doc etc..



## Note

We recommend to use the template, for a uniform view to all Pepper modules. That makes it possible not to forget important issues to be mentioned and makes it easier for the user to have a good understanding of what the Pepper module is doing.

You will find the template in `SAMPLE_MODULES_HOME/src/main/docbkx/manual.xml` among other directories containing files necessary for the transformation. To refer to images from the manual, put them into the image folder `SAMPLE_MODULES_HOME/src/main/docbkx/images/` and make a relative reference.

## Transformation

The standard transformation which is configured for SampleModules is the transformation to html and pdf. The configuration is done in the `pom.xml` of the `pepper-parentModule` project. To add further output formats, just copy the respective plugins to your pom and change them.

When executing the maven goal site

```
mvn clean site
```

, maven will create a manual folder under `SAMPLE_MODULES_HOME/target/manual`, where you can find the pdf documentation and the html documentation.



## Note

The current configuration does not need to be changed, just write your documentation by overriding the template. The rest will work automatically. In some cases it might be necessary to adopt the transformation, then please take a look to the xsl transformation files in `SAMPLE_MODULES_HOME/docbook-xsl`.

---

# Chapter 6. some little helpers

As part of the Pepper project, we provide some tiny classes which could help to improve the developing speed of Pepper modules. In this chapter, we list all these helpers with a description of their aim.

## xml extractor

This class is a helper class for developing PepperModules. The `XMLTagExtractor` generates a dictionary of the xml vocabulary. The dictionary consists of xml tag names, xml namespaces and attribute names from a source file and generates a java interface and a java class as well. The interface contains the xml namespace declarations, the xml element and attribute names as fields (public static final Strings). The generated java class implements that interface and further extends the `DefaultHandler2` class, to read a xml file following the generated xml dictionary.

This class can be very helpful, when creating `PepperImporter` or `PepperExporter` classes consuming or producing xml formats. In that case, a sample xml file (containing most or better all of the elements) can be used to extract all element names as keys for the implementation.

For instance, the following xml file:

```
<sentence xml:lang="en">
  <token pos="VBZ">Is</token>
  <token pos="DT" lemma="this">this</token>
  <token>example</token>
</sentence>
```

will be result in the following interface:

```
public interface INTERFACE_NAME{
    public static final String TAG_TOKEN= "token";
    public static final String TAG_SENTENCE= "sentence";
    public static final String ATT_LEMMA= "lemma";
    public static final String ATT_XML_LANG= "xml:lang";
    public static final String ATT_POS= "pos";
}
```

where `INTERFACE_NAME` is the name of the xml file. and in the following class:

```
public class INTERFACE_NAMEReader extends DefaultHandler2 implements Berглеute{
    public void startElement( String uri,
                             String localName,
                             String qName,
                             Attributes attributes)throws SAXException
    {
        if (TAG_TOKEN.equals(qName)){
        }else if (TAG_SENTENCE.equals(qName)){
        }
    }
}
```

Using as a library:



```
XMLTagExtractor extractor= new XMLTagExtractor();  
extractor.setXmlResource(input);  
extractor.setJavaResource(output);  
extractor.extract();
```

Running this tiny program from command line:

```
java XMLTagExtractor.class -i XML_FILE -o OUTPUT_PATH
```