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}

Table of Contents

List of Figures

List of Tables

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 linguistic 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/) as well. 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 fromat 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/) providing 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 developement 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) and can easily be extended for a specific module test.

In the following, Chapter 2, *Setting up your environment* explains how to set up your environment to start developing your Pepper module. Chapter 3, *Adopting the SampleModules* explains how to download and adopt a template module to your own needs. This module then is the base for your own module. In Chapter 4, *Customizing behaviour of your Pepper module* 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. These should save you some time, not to test even the simplest things. 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>`.

Chapter 2. Setting up your environment

If you do not belong to the hardcore 'vi' developer community, you may want to use a developement environment and an IDE for developing 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/) but you should make sure that the OSGi framework is set up correctly for your IDE. In that case, skip the Eclipse specific parts.

Download and set up Eclipse

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. For this tutorial, we used Eclipse juno, version 4.2 (see: http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/junosr2). Unfortunately, in this version Eclipse does not come with an integrated OSGi console anymore. It now uses the Apache Gogo Shell which is not part of the Eclipse distribution. Therefore we have to download the Equinox SDK from http://download.eclipse.org/equinox/, we used version 3.8.2 which is available on http://www.eclipse.org/downloads/download.php?file=/equinox/drops/R-3.8.2-201302041200/equinox-SDK-3.8.2.zip.

- Download the Eclipse IDE and unzip it to ECLIPSE_HOME (a folder of your choice).
- Download the equinox sdk and unzip it to EQUINOX_SDK (a folder of your choice).
- Copy all (not already contained) bundles (.jar) from EQUINOX_SDK/plugins folder to ECLIPSE_HOME/plugins folder.

Making Eclipse ready for SaltNPepper

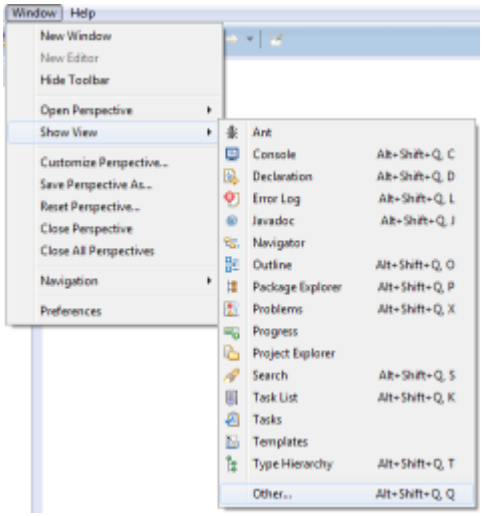
- Download the latest Pepper distribution from http://korpling.german.hu-berlin.de/saltnpepper/repository/saltNpepper_full/
- Unzip the Pepper distribution to PEPPER_HOME.
- Copy all bundles (.jar files) from PEPPER_HOME/plugins to ECLIPSE_HOME/dropins.
- Either copy all folders having the same name as the bundles (.jar files) from PEPPER_HOME/plugins to a folder of your choice, or let them be where they are. But in any way, let's call that location BUNDLE_RESOURCES/plugins for the following steps.
- Copy the folder PEPPER_HOME/conf to BUNDLE_RESOURCES/conf.

Starting Eclipse

Now lets check wether Eclipse was set up correctly and is able to find all necessary bundles.

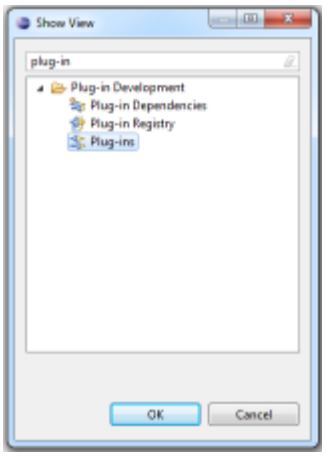
- Start Eclipse. (When starting Eclipse, having no projects registered in your workspace, a welcome tab may appears. Just close this tab and go on.)
- Open 'Plug-ins view' via 'Window --> Show View --> Others (see Figure 2.1, “”).

Figure 2.1.



- Type in 'plug-ins' and choose 'Plug-ins' to open the corresponding view (see Figure 2.2, “”).

  Figure 2.2.

- In that view, you will find all OSGi bundles, which are accessable from your Eclipse installation.

- Check wether all bundles from SaltNPepper are there. The main ones are
  - de.hu_berlin.german.korpling.saltnpepper.pepper-exceptions.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-framework.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-logReader.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-modules.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-moduleTests.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-testEnvironment.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-workflow.jar
  - de.hu_berlin.german.korpling.saltnpepper.salt-graph.jar
  - de.hu_berlin.german.korpling.saltnpepper.salt-saltCommon.jar

- de.hu_berlin.german.korpling.saltnpepper.salt-saltCore.jar

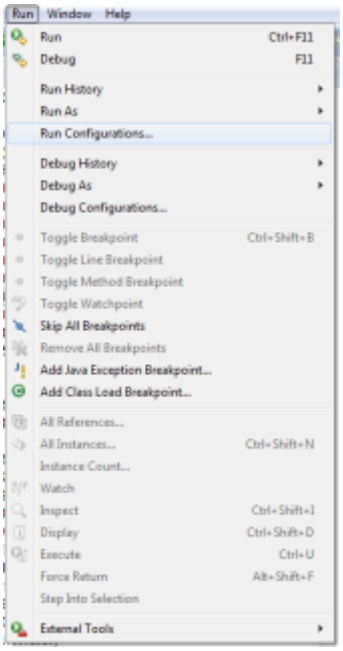
Congratulations, now you are done setting up your IDE.

Let's run

Now we want to let the ghost out of the bottle and run the Pepper framework.

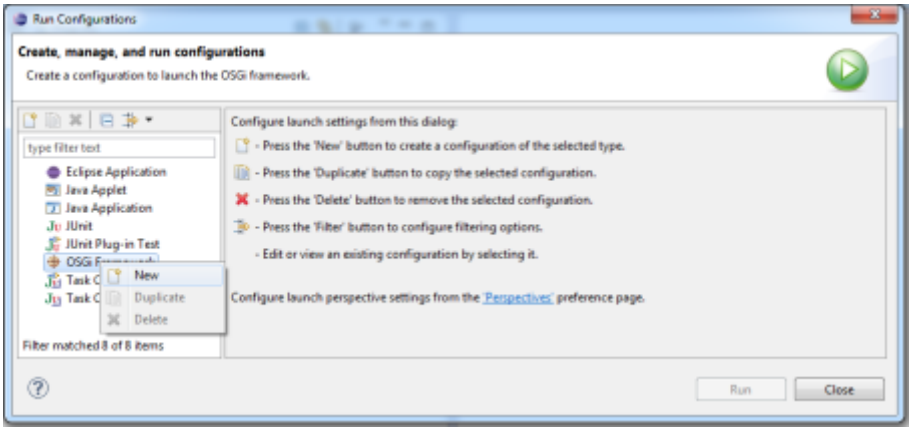
- Select 'Run --> Run configurations...' (see Figure 2.3, “”).

Figure 2.3.



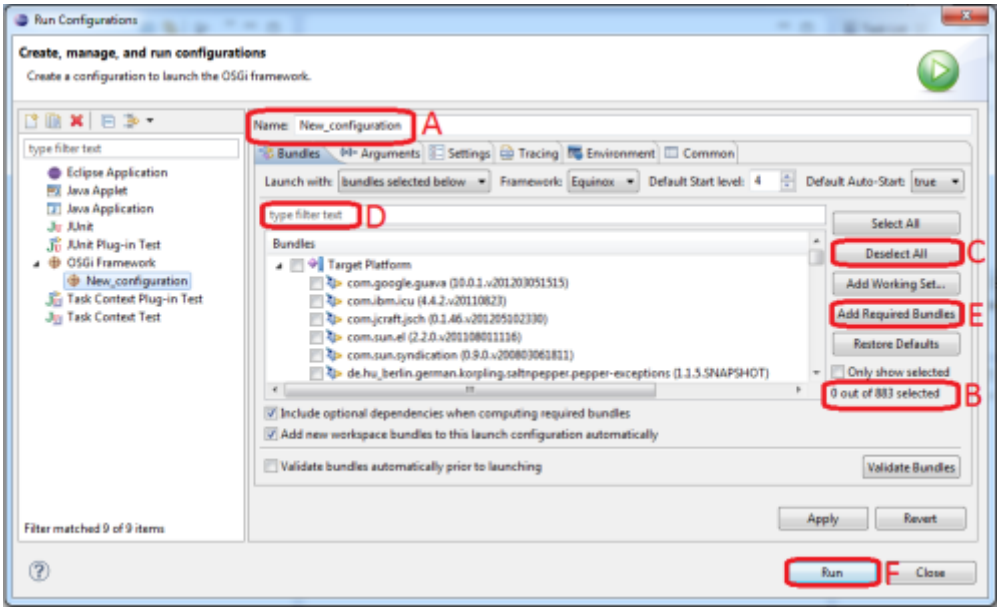
- Select 'OSGi Framework', press right mouse button and select 'New' (see Figure 2.4, “”).

Figure 2.4.



- Give your run configuration a name under 'Name:', (see Figure 2.5, “” position A).

Figure 2.5.



- For not loading all available bundles, which could be more than 800 (see Figure 2.5, “” position B), press the button 'Deselect All' (see Figure 2.5, “” position C).

Note

This may take a while, so don't be feared, in general your computer is not frozen.

- Now, activate all necessary bundles for running Pepper. In 'type filter text' (see Figure 2.5, “” position D) type in 'de.hu_berlin' and you will see all bundles having this string in the package name. To active the necessary bundles, click the box on the left of their name. Make sure that all bundles you have copied from PEPPER_HOME are activated. These are at least the following ones:
  - de.hu_berlin.german.korpling.saltnpepper.pepper-exceptions.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-framework.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-logReader.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-modules.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-moduleTests.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-testEnvironment.jar
  - de.hu_berlin.german.korpling.saltnpepper.pepper-workflow.jar
  - de.hu_berlin.german.korpling.saltnpepper.salt-graph.jar
  - de.hu_berlin.german.korpling.saltnpepper.salt-saltCommon.jar
  - de.hu_berlin.german.korpling.saltnpepper.salt-saltCore.jar
- There are some other bundles necessary for running Pepper to be activated:

Note

These packages also have to be select manually. Filtering them can even take a while.

- org.eclipse.equinox.console
- org.eclipse.equinox.ds
- org.apache.felix.gogo.shell
- org.apache.log4j

- Since these are not all of the necessary bundles, we have to select some more, but Eclipse now should be able to detect them automatically, by pressing the button 'Add Required Bundles' (see Figure 2.5, “” position E).

Note

In case of this button is disabled, remove the string from the filter

Now some more bundles are added, on the right you can see the increased number of currently selected bundles (see Figure 2.5, “” position B).

- Since the start order of some bundles is important, set 'Start Level' of the following bundles as follows.

Table 2.1. Sample Table

Bundles	**Start Level**
de.hu_berlin.german.korpling.saltnpepper.pepper-logReader	2
de.hu_berlin.german.korpling.saltnpepper.pepper-framework	5
de.hu_berlin.german.korpling.saltnpepper.pepper-testEnvironment	6

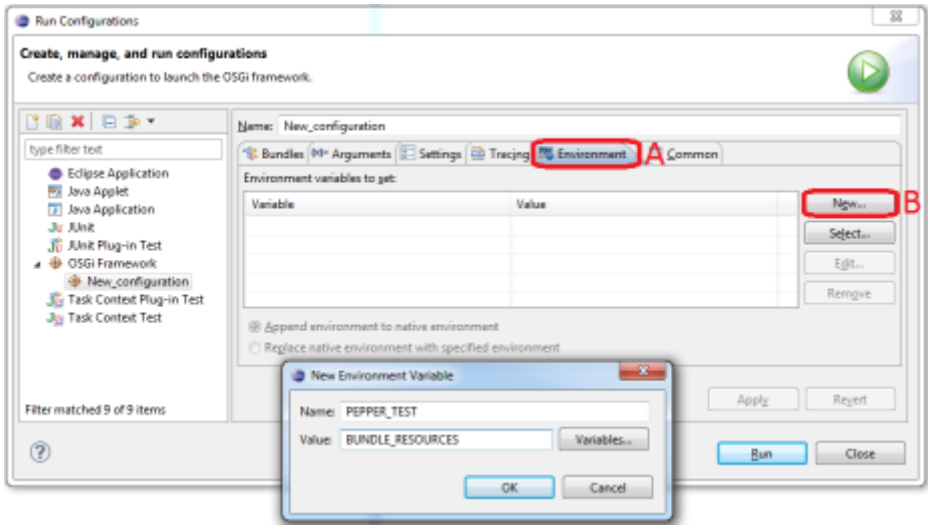
Note

We assume that the 'Default Start Level' is set to 4.

- Now we need to set an environment variable named 'PEPPER_TEST' to register resources like configuration files of the bundles, if needed.

  1. Go to tab 'Environment' (see Figure 2.6, “” position A).

Figure 2.6.



2. Press button 'New...' (see Figure 2.6, “” position B).

3. Type in 'PEPPER_TEST' for name and the folder known as BUNDLE_RESOURCES (without the 'plugins' subfolder) as value.

- Press button 'Run' (see Figure 2.5, “” position F).

Now Pepper is running and shows all registered importers, exporters and manipulators. A screen like shown in Figure 2.5, “” comes up.

Note

The output can vary, depending on the modules you have registered and the version of the OSGi console. Eventually a lot of log messages are outputted as well.

Figure 2.7.

```
Welcome to Apache Felix Gogo

g!
************************************************************************
***                    Test Pepper Converter                         ***
************************************************************************
* Pepper converter is a salt model based converter for a lot of        *
* linguistical formats.                                                *
* for contact write an eMail to: saltnpepper@lists.hu-berlin.de        *
************************************************************************
```

To make things easier, we provide a preconfigured file, which can be used for this, unfortunately this configuration is for Eclipse only. This file contains information for the run configurations of the `pepper-testEnvironment` project and can be found under `https://korpling.german.hu-berlin.de/svn/saltnpepper/pepper/tags/pepper-XXX/pepper-testSuite/pepper-testEnvironment/pepper-testEnvironment.launch` (where XXX is the version number). Just copy that file into your poject, update the view in your project or package explorer by pressing 'F5', right click that file and select 'run as...' --> 'pepper-testEnvironment'.

To install further Pepper modules, follow the given instructions:

1. Download a module of your choice from http://korpling.german.hu-berlin.de/saltnpepper/repository/pepperModules/de/hu_berlin/german/korpling/saltnpepper/pepperModules/ or any other source.
2. Unzip the file to a folder of your choice, let's call it MODULE_HOME
3. Copy the bundel (.jar file) to ECLIPSE_HOME/dropins
4. Copy the folder having the same name as the bundle (.jar file) to BUNDLE_RESOURCES/plugins.
5. Select the downloaded module in your run configurations.
6. Press button 'Add required bundles' (see Figure 2.5, “” position E).
7. To check if all bundle dependencies could be resolved, press 'Validate Bundles'. (see Figure 2.5, “”)

Making a real run

Now we want to start a real run using a workflow description file. To do so, we need a sample corpus and a sample workflow description file. The sample workflow description file specifies a workflow converting a corpus in paula to relANNIS.

Note

Make sure, that the PAULAImporter and the RelANNISExporter (or more precisely the PAULAModules and the RelANNISModules bundles) are registered in your Eclipse installation (see 'plugins view' the section called “Starting Eclipse”) and in your run configurations.

- Download the sample corpus from http://korpling.german.hu-berlin.de/saltnpepper/samples/corpora/pcc2.zip and unzip it to SAMPLE_CORPUS.
- Create an environment variable named 'PEPPER_TEST_WORKFLOW_FILE' in your 'Run configurations' and point it to SAMPLE_CORPUS/paula2relANNIS.pepperParams (see Figure 2.6, “” for creating environment variables; name='PEPPER_TEST_WORKFLOW_FILE', value='SAMPLE_CORPUS/paula2relANNIS.pepperParams')
- Press 'Run'.

Now the Pepper test environment is set up, and Pepper does the conversion task.

Setting up maven

As already mentioned, we used maven as lifecycle and dependency management tool. For the developement of your own modules, it would be very helpful, to also use maven. But you are free to use any other lifecycle management tool like ant (see http://ant.apache.org/), gradle (see http://www.gradle.org/) or none.

If you are working with Netbeans, you can skip the rest of this section, because Netbeans already comes with an integrated maven. In the case that you are working with Eclipse, or even want to compile your Pepper module via the command line, you have to install maven. Therefore just follow the given instructions:

- Download the latest maven distribution from http://maven.apache.org/ (we used version 3.0.3).
- Unzip the file to a folder of your choice and you will get a folder like apache-maven-version in it. Lets call it MAVEN_HOME.

  For Linux: Or install maven from your distribution repositories.

Now, when switching to MAVEN_HOME/bin folder, you can run maven from the command line via calling:

```
mvn
```

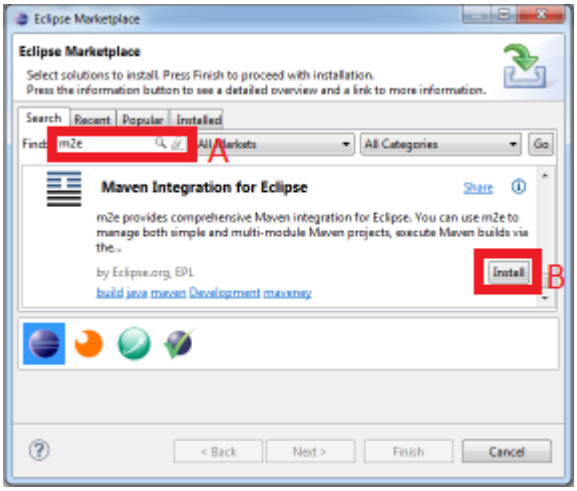
For Windows: If you want to run mvn everywhere, write the path MAVEN_HOME/bin into your path environment variable (under Windows).

For Linux: Maven should automatically be installed in the default directory if you use the Distribution repository (e.g. /bin or /usr/bin) and thus be runnable.

To enable maven in Eclipse you have to use a specific plugin called m2e.

- Open the 'Eclipse Marketplace' via 'Help --> Eclipse Marketplace...' (see Figure 2.8, “”)

Figure 2.8.



- Type in 'm2e' in text box 'Find:' and press enter (see Figure 2.8, “” position A).
- The plugin 'Maven integration for Eclipse' should be displayed (you may have to scroll down). Click "Install" to install that plugin (see Figure 2.8, “” position B).
- During the installation Eclipse asks you to agree to the license of this plugin it may be recommended to restart Eclipse. Just follow the instructions and go ahead.
- After restarting Eclipse, the plugin should be installed. You can check the installation, when clicking on 'File --> Import'. The dialog which opens should contain an option/a folder named 'Maven'.

If you are working with Eclipse and the command line simultaneously (which might be very helpful, since maven does not always run error-free in Eclipse), it might be useful, to synchronize the maven local repository of your m2e installation and the one of your maven installation.

- Go to 'Window --> Preferences'.
- Expand the entry 'Maven'.
- Select the 'Installations' view.
- Press the 'Add' button on the upper right and choose MAVEN_HOME (location of the maven installation, not of the maven repository).

A Pepper module is created using the OSGi declarative service mechanism. If you don't know what this is and you don't want to know, never mind. We put the OSGi layer in the background so that you do not have to care about (as long as everything works fine ;-)). But therefore we used maven to configure

the OSGi part. To enable the declarative service mechanism, we used some maven dependencies called 'carrot-osgi-anno-scr'. Since there is no m2e adapter installed in Eclipse by default, you have to download a further plugin.

- Go to the 'Eclipse Marketplace' via 'Help --> Eclipse Marketplace...'.
- Enter in 'Find:' the string 'CarrotGarden' and press enter.
- The dialog will show the 'CarrotGarden SCR' plugin. Press the 'install' button and follow the instructions. Eclipse may recommends to restart itself.

Chapter 3. Adopting the SampleModules

Download the skeleton

To start creating your own module for Pepper, please download the SampleModule, to be used as a skeleton for your own module. SampleModules is stored in our svn repository, therefore please export the project via an SVN client. The SVN location depends on the version you want to use and can be found at SampleModules (see: https://korpling.german.hu-berlin.de/svn/saltnpepper/PepperModules/SampleModules/tags).

Adopting the project

Here, we give a list of things to do to adapt the project to your needs. In some cases, the order of the entries is arbitrary, but in some cases it is easier to respect the given order.

- Import the project as a Maven project (when working with Eclipse, use the import option existing maven project, it might be that you have to install further plugins or m2e connectors. So don't be scared if a dialog pops up.)
- Rename the 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...")
- Rename the packages (when working with Eclipse, right click on the package name and choose the menu entry "Refactor" --> "Rename...")

Adopting the pom.xml

Maven follows the paradigm convention over configuration, therefore a lot of things are already predefined and do not need to be changed.

But still, there are some things like the name of the project and so on, which are project specific and therefore have to be adopted. Just follow the given instructions to adopt the 'project object model' (pom). The items in the numbered list correspond to the TODO entries in the pom.xml file. You will find the pom.xml file in the parent directory of the SampleProject.

1. Change the "groupId" to the name of your module (conventionally, the groupId is the same as the package name). You will find the entry under '/project/groupId'.
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. Import needed maven dependencies (this is necessary to resolve dependencies to libraries handled by maven). You will find the entry under '/project/dependencies'.

11. 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'.

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. Unfortunatly, you have 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=ARTI
    -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"

Adopting the Java code

In Pepper we tried to avoid as much as complexity as possible without reducing the funtionality. We want to enable you to concentrate on the main issues, which are the mapping of objects. This trade-off has been realized by first using some default assumptions, which reduces complexity in a lot of cases and second by class derivation and call-back methods, to still provide the full functionality if necessary.

In Salt and in Pepper we differeciate between the corpus-structure and the document-structure. The document-structure contains the primary data (datasoruces) 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 a bunch of documents to a corpus or sub-corpus (`SCorpus` in Salt).

In Pepper we talk about three different kinds of modules, the `PepperImporter`, the `PepperExporter` and the `PepperManipulator`. An importer is used, to map data to a salt model, an exporter is used to map a Salt model to other kind of data and a manipoulator is used to map a Salt model to another Salt model for instance, for renaming data, merging data etc.. A mapping process is separated into three phases: import-phase, manipulation-phase and export-phase. In each phase an unbound number of Pepper modules can be used. Other than in the manipulation phase, in the import and export-phase at least one module must be involved. Each phase again is separated into mapping steps. A mapping step more or less is a pair of a Pepper module and a Salt object (which can be a `SCorpus` or a `SDocument`). In Salt, an `SDocument` object can be seen as a partition, allowing no links between onjects contained in such a document to the outside. The same goes for `SCorpus` objects. This allows, several steps to work independently to each other. Each module can be sure to be the only one processing a Salt object at a time. Since a mapping process can be relative time consuming, we could decrease the needed time to map an entire corpus, if we are able to process mapping tasks simultaneously. In Pepper each module can be ran in multi-threading mode by default. But keep in mind, that multi-threading can not be added to a module as a new feature, it more needs an implementor to take care of during all the implementation work. If you are not very familar with multi-threading in java, we want to give you the hint to avoid class variables and methods. The same goes for static variables and methods[1]. If it is still necessary, to use such constructs, we recommand, to go deeper in the sometimes annoying but powerfull world of multi-threading. If you do not want to blow up things and don't want to benefit of the better performance, you can also switch of multi-threading by a flag (just call the method `setIsMultithreaded(false)` in your modules constructor).

Enough of theory, lets step into the code now. In the SampleModules project, you will find three classes according to the three sorts of modules: `SampleImporter`, `SampleManipulator` and `SampleExporter`. All of them are located in the package de.hu_berlin.german.korpling.saltnpepper.pepperModules.sampleModules. 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. In case of you want to have several importers, manipulators or exporters, just duplicate the classes. Otherwise, you will end up with non-functional modules in your project.

A Pepper module (derived from class `PepperModule`) represents an interface to be accessed by the Pepper framework. Most of the provided methods therefore have a default implementation, which should be overridden, to have a real functional module, but do not have to be. Other mthods have to be overridden, since they do not contain any functionality. If you want to go along the Pepper bestpractices, we recommand to give each implementation of `PepperImporter`, `PepperExporter` and `PepperManipulator` an own implementation of `PepperMapper` by hand. Doing this has to benefits, first the multi-threading part is realized with this distinction and second, it will take a lot of comlexity from your shoulders.

We now want to give a brief overview of what to do to adopt the classes of SampleModules project. Inside the single items, you will find links to further explanaitions.

1. Change the name of the module, for instance to MyImporter, MyExporteretc. We recommend to use the format name and the ending Importer, Exporter or Manipulator (like `FORMATImporter`).

2. Change the name of the component, 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
   ```

3. 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:

[1] If such a variable never is changed or such a method only returns constants, you can even use them. Marking methods with the keyword synchronized and variables with the keyword volatile can also help to avoid mult-threading problems.

```
public MyImporterImporter()
{
 super();
 this.name= "MyImporterImporter";
 //we recommend to synchronize this value with the maven version in your pom.x
 this.setVersion("1.1.0");
 this.addSupportedFormat("myFormat", "1.0", null);
 //see also predefined endings beginning with 'ENDING_'
 this.getSDocumentEndings().add("myFormat");
}
```

4. After the module is created, the Pepper framework calls the method `isReadyToRun()`, at this stage all intializations of module have been completed, so that now the module can make some own initializations. For instance pathes like a path for storing temprorary data (see `getTemproraries()`) is set and a path, where to find additional resources (see `getResources()`) if given.

   ```
   public boolean isReadyToStart() throws PepperModuleNotReadyException
   {
       //make some initializations if necessary
    return(true);
   }
   ```

5. In case that you are creating an importer, check if the default behavior of the corpus-structure import mechanism fits for your need. For further details, see: the section called “Importing the corpus-structure”.

6. The main interesting part of a mapping probably is the mapping of the document structure. It means the mapping of the real linguistic data, so the reason why where are doing all of this. Since such a mapping is not even trivial, we recommand to delegate it to another class which is derived of class `PepperMapper`. To register that class create a method named `createPepperMapper(SElementId sElementId)` returning your specific mapper object, for instance called MyMapper, see:

   ```
   public PepperMapper createPepperMapper(SElementId sElementId)
   {
    MyMapper mapper= new MyMapper();
    return(mapper);
   }
   ```

   For further details, or even if you do not want to use the default mechanism, see: the section called “Mapping the document-structure”.

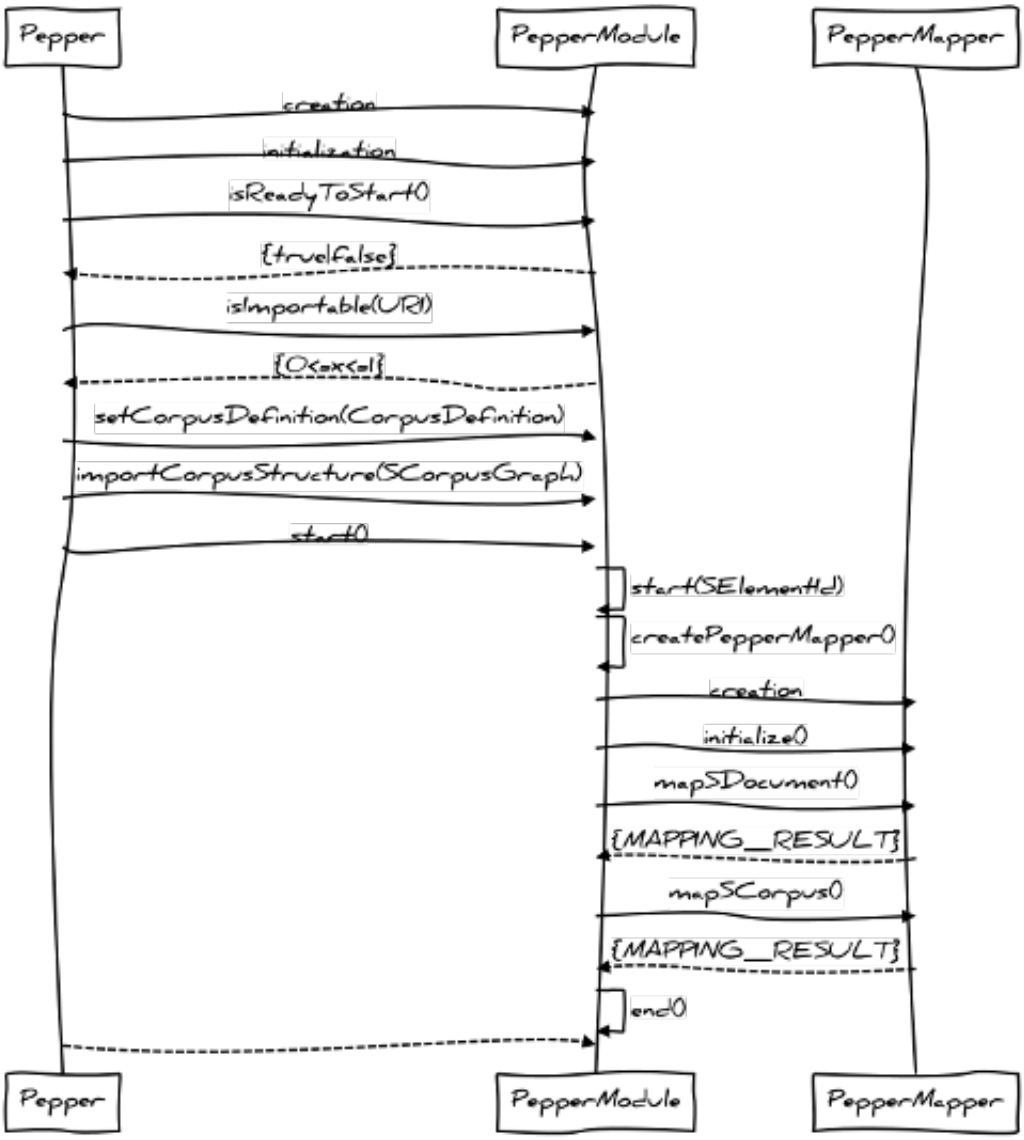
7. Monitoring the progress would give a good feedback to the user. Since the conversion may take a while, we want to prevent the user to kill the conversion job, by providing the progress status. Therefore Pepper offers possibilities for a module to notify the framework about its progress. In case of you are using the default behaviour via `PepperMapper`, call the method `addProgress(Double progress)` and pass the additional progress for the current `SDocument` or `SCorpus`. In case of you do not use the default behaviour please override the method `getProgress(SElementId sDocumentId)` and `getProgress()` in your module. For further details, see: the section called “Monitoring the progress”.

8. Pepper provides a mechanism to automatically detect, if a format can be read by an importer, therefore you have to override the method `isImportable(URI corpusPath)`, which returns a value for determining if the resource at passed path is importable by this module. The return values are 1 if corpus is importable, 0 if corpus is not importable, $0 < X < 1$, if no definitiv answer is possible, null if method is not overridden. For further details, see: the section called “Analyzing the unknown”.

9. If you have to clean up things (e.g. delete temporary files etc.), override the method `end()`, see: the section called “Cleaning up after all”.

The following figure shows a sequence diagram displaying the sequence of method calls between the Pepper framework and the classes to implemented `Peppermodule` and `PepperMapper`. Please note, that this is just a simplified representation, some methods are not shown, and even a further class called `PepperMapperController`, which interacts between the `PepperModule` and the `PepperMapper` class.

Figure 3.1. Pepper workflow overview



In the following sections, you will find more detailed explanaitions to the prior given single steps. If the information we are giving here still not enough, please take a look into the corresponding JavaDoc.

Importing the corpus-structure

A corpus-structure consists of corpora (represented via the Salt element `SCorpus`), documents (represented represented via the Salt element `SDocument`), a linking between corpora and a linking between a corpus and a document (represented via the Salt element `SCorpusRelation` and `SCorpDocRelation`). Each corpus can contain 0..* subcorpus and 0..* documents, but a corpus

cannot contain both document and corpus. For more information, please take a look into the Salt User Guide.

The general class `PepperImporter` provides an adoptable and automatical mechanism to create a corpus-structure. This mechanism is adoptable step by step, according to your specific purpose. In many cases, the corpus structure is simulatneaus to the file structure of a corpus. Since many formats does not care about the corpus-structure, they only encode the document-strcuture.

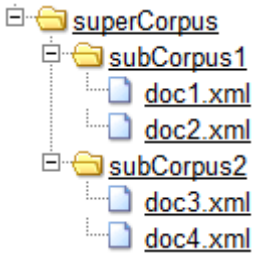
Peppers default mapping maps the root folder (the direct `URI` location) into a root-corpus (`SCorpus` object). For each sub-folder a sub-corpus (`SCorpus` object) is created and added to the `SCorpusGraph`. For each super- and sub-corpus relation, a `SCorpusRelation` object is created. For each interesting[2] file a `SDocument` object is created and added to the current `SCorpusGraph` object. The `SCorpus` and `SDocument` objects are connected via a `SCorpusDocumentRelation` object. While traversing the file-structure, an `SElementId` object is created representing the corpus-hierarchy and added to the created `SCorpus` or `SDocument` object. A map of these `SElementId` objects corresponding to the `URI` objects is returned, so that in method `start(SElementId)` this map can be used to identify the `URI` location of the `SDocument` objects.

To customize the default behaviour, Pepper provides three steps ascending in their implementation efforts:

1. Adopting the one or more of the collections: `sDocumentEndings`, `sCorpusEndings` or `ignoreEndings`, accessible via the methods `getSDocumentEndings`, `getSCorpusEndings` and `getIgnoreEndings`. To `sDocumentEndings`you can add all endings of files, containing the document-structure (you can even add a value for leaf folders `ENDING_LEAF_FOLDER`). To `sCorpusEndings` you can add all endings of files, containing data for the corpus-structure (the value `ENDING_FOLDER` for folders is set by default). And to `ignoreEndings` you can add all endings of files, to be ignored for the import process.

   Lets give an example:

   Figure 3.2.

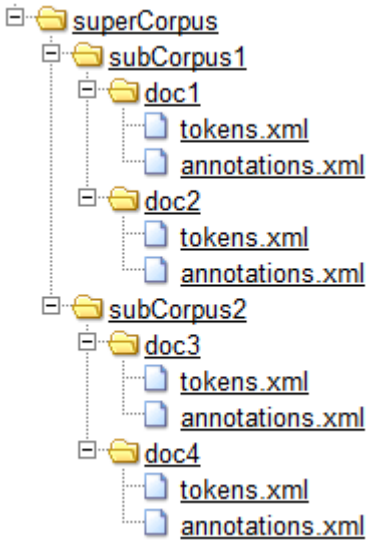
   

   To map this structure, you just have to add the ending 'xml' to the collection `sDocumentEndings`, you can do this using the constant `ENDING_XML` (even others are predefined).

   To give another example, the following file-structure will result in the same corpus-structure, when setting the collection `sDocumentEndings` to the value `ENDING_LEAF_FOLDER`.

[2] interesting here means, a file whichs ending is registered in a collection (see: `getSDocumentEndings()`)

Figure 3.3.



In case of the ending of a file does not matter, you can use `ENDING_ALL_FILES` to correspond all types of files to a `SDocument` object.

You can do the collection adoptions anywhere you want to, but make sure, that they have to be done, before the method `importCorpusStructure` was called by the framework. A good location for instance is the constructor:

```
public MyImporter()
{
  ...
  this.getSDocumentEndings().add(PepperImporter.ENDING_LEAF_FOLDER);
  ...
}
```

2. If the previous step does not fully fit your needs, Pepper uses a callback mechanism which can be overridden. Using the method `setTypeOfResource` Pepper checks for each resource (folder and pure file) whether it represents a `SDocument` or a `SCorpus` object. The origin implementation therefore uses the above mentioned collections. If adopting these collections does not fullfill your specific purpose, you can override it with a more complex behavior.

3. The last opportunity to adopt the behavior is to override the methode `importCorpusStructure`

```
public void importCorpusStructure(SCorpusGraph corpusGraph) throws PepperModu
{
    //implement your specific behaviour
}
```

This method must generate a map containing the correspondence between a `SElementId` (representing a `SDocument` or a `SCorpus` object) and a resource. For instance, remember the sample of Figure 3.2, “”, this could result in the following map:

Table 3.1.

salt://corp1	/superCorpus

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

Later on, this map is used when importing the document-structure, to locate the content of a `SDocument` object.

Mapping the document-structure

The document-structure in Salt is a notion, to determine the real linguistic objects, like primary data (e.g. primary texts), tokenizations, constructs like words, sentences and so on and even their annotations in form of attribute-value-pairs. In Salt all such data are contained in a graph object called `SDocumentGraph`. This graph itself is contained in the `SDocument` object and can be accessed via `SDocument.getSDocumentGraph()`. The `SDocumentGraph` objects contains a bunch of different kinds of nodes and edges representing the linguistic data. For instance all primary texts in a document can be accessed via teh method `SDocumentGraph.getStextualDSs()`. For a more detailed description of how to access a Salt model, please read the Salt User guide.

Mapping the document-structure, can mean to map one Salt model to another one, like a manipulator does, it can mean to import a document-structure like an importer does or to export a document-structure like an exporter does. In this section, we describe the mechanism in Pepper of how to map a document-structure and meta data for corpora and documents. And which methods are needed to be overridden.

Similar to the mapping of the corpus-structure, Pepper provides several levels where to intervene depending of how much the default behaviour matches your needs.

The easiest and may be most beneficial way of creating a mapping is the use of the default mechanism, which means to derive the class `PepperMapper`. This also enables to run your module multithreading in multithreeading mode without taking care of creating threads in Java.

Let's start with the most important methods `mapSDocument()` and `mapSCorpus()`. `mapSDocument()` is the method to map the document-structure and is called by the framework for each `SDocument` in `SCorpusGraph`. If this method is called, you can get the `SDocument` object to be imported, exported or manipulated by the method `getSDocument()`. In case of you are implementing an im or an exporter, you need to know the resource location. This can be accessed via `getResourceURI()` and will return a uri pointing to the location of the resource. The same goes for the method `mapSCorpus()`, but here the method `getSDocument()` will return an empty result. So call `getSCorpus()` to get the current object to be manipulated. Often manipulating the `SCorpus` is necessary to add further meta-data, which were not be added during import phase of corpus-structure or to manipulate or export the meta-data. Both methods shall return a value determining the success of the mapping. Therefore the following three possible values are predefined `MAPPING_RESULT.FINISHED`, `MAPPING_RESULT.FAILED` and `MAPPING_RESULT.DELETED`. Finished means, that a document or corpus has been processed successfully, failed means, that the corpus or document could not be processed because of any kind of error and deleted means, that the document or corpus was deleted and shall not be processed any further (by following modules).

If you need to do some initializations before the methods `mapSDocument()` and or `mapSCorpus()` are called, but after the constructor has been called, just override the method `initialize()`. This methods enables the possibility to make some initilizations depending on the values set by the framework.The general initialization, like setting the resource path and the `SDocument` or `SCorpus` object to be manipulated is done by the framework itself.

```
@Override
public MAPPING_RESULT mapSCorpus() {
 //retunrs the resource in case of module is an importer or exporter
   getResourceURI();
   //returns the SDocument object to be manipulated
   getSDocument();
   //returns that process was successful
   return(MAPPING_RESULT.FINISHED);
}

@Override
public MAPPING_RESULT mapSDocument() {
   //retunrs the resource in case of module is an importer or exporter
   getResourceURI();
   //returns the SCorpus object to be manipulated
   getSCorpus();
   //returns that process was successful
   return(MAPPING_RESULT.FINISHED);
}

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

If you are using the default behaviour, you are done. Congratulations ;-).

If you need more flexibility to adopt the mapping behaviour, you have to step into the mechanism a level above. The class `PepperModule` specifies the `start(SElementId sElementId)`. In your module implementation, you can override that method. The following lines of code will separate if the object to be processed is of type `SDocument` or `SCorpus`.

```
if (sElementId.getSIdentifiableElement() instanceof SCorpus)
{
 //map for instance some meta-data
}
else if (sElementId.getSIdentifiableElement() instanceof SDocument)
{
  //map the document-structure
}
```

Note

Overriding the method `start(SElementId sElementId)` does not enable multithreading automatically. If you still want a multithreading processing, you have to implement it on your own.

Note

If you have decided to not use the default behaviour, please check the source code of the `PepperModule` and `PepperImporter`, `PepperManipulator` or `PepperExporter` depending on which kind of module you are implementing. The sources will give you a more detailed view for what you have to take care of.

The class `PepperModule` specifies the `start()`, this is the modt general method in terms of mapping and is deirectly called by the Pepper framework or more

precisly spoken the `PepperModuleController`. To access the `SDocument` or `SCorpus`, you have to use the `PepperModuleController`, which can be accessed via `getPepperModuleController()`. Since a `PepperMoculeController` object is connected to the controller object of the preceding and the following module, the objects to be processed are exchanged via a queue on which both controllers have access. The module you are implementing can get the current `SElementId` object to be processed via `getPepperModuleController().get()`. But take, care that this method will first return a result when the preceding module has processed its object. Until then the method will let your module wait.

Note

If you have decided to not use the default behaviour, please check the source code of the `PepperModule` and `PepperImporter`, `PepperManipulator` or `PepperExporter` depending on which kind of module you are implementing. The sources will give you a more detailed view for what you have to take care of.

Cleaning up after all

Sometimes it might be necessary to make a clean up, after module did the job. For instance when writing an im- or an exporter it might be necessary to close file streams or a db connection. Therefore, after all the processing was done, the Pepper framework calls the method `end`. To run your clean up, just override it and your done.

Monitoring the progress

What could be more annoying as a not responding program. When you do not know if it is still working or not. Since a conversion job could take some time, what 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.

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

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 devide the entire progress in several steps and to return a progress for each step. for instance

```
//...
//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 will be done automatically.

In case of 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.

Analyzing the unknown

Note

This section only is usefull, in case of you are implementing a `PepperImporter`.

Experience has shown, that a lot of users, do not care a lot about formats and don't want to. Unfortunatly, 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 the most users are not very interested, in the source format of a coprus, they only want to bring it 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. Unfortunatly 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, what makes it recommanded.

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 locationpoints to a directory), and to read their header section. For instance some formats like the xml formats PAULA () or TEI () starts with a header section like

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

or

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-model href="http://www.tei-c.org/release/xml/tei/custom/schema/relaxng/te
<?xml-model href="http://www.tei-c.org/release/xml/tei/custom/schema/relaxng/te
 schematypens="http://purl.oclc.org/dsdl/schematron"?>
<TEI xmlns="http://www.tei-c.org/ns/1.0">
```

. Such formats where only reading the first lines will bring information about the formats name and its version make automatic detection very easy. The method `isImportable(URI corpusPath)` shall return 1 if corpus is importable, 0 if corpus is not importable or a value between $0 < X < 1$, if no definitiv 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.

building the project

We adopt the settings for using OSGi and to create Pepper modules, which are simply pluggable into the Pepper framework. So in the best case, you can just follow this guide, and run

```
mvn install assembly:single
```

and you will find a zip file in the target folder of your project (YOUR_PROJECT/target/distribution), which can simply plugged into Pepper via extracting its content to the plugin folder of Pepper (PEPPER_HOME/plugins). In the bad case some mean errors occur and you have to dig in a lot of online tutorials and forums and lose a lot of time. Please do not despair and write us an e-mail instead. <saltnpepper@lists.hu-berlin.de>

Chapter 4. Customizing behaviour of your Pepper module

via properties

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 possiblity. 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 specifified properties. You should always define a default behaviour in case of 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 posibility 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 fullfill 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.PepperModu
import de.hu_berlin.german.korpling.saltnpepper.pepper.pepperModules.PepperModu
//...
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 existance. 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 exception, in case of set 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());
```

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/). 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 shall 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. 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) 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 developement 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 correct 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 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 of 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 7. FAQ

Cannot run maven install under Eclipse?

Restart Eclipse and try again, sometimes Eclipse does not recognize a maven project at first time.