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

Foreword vi
1. Introduction 1
    The sample 1
2. 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 13
    Analyzing an unknown corpus 16
    Im- and exporting corpus-structure 16
        Importing the corpus-structure 18
        Exporting the corpus-structure 19
    Customizing the mapping 19
        Property 20
    Monitoring the progress 21
        Logging 22
        Error handling 22
    Prepare and clean-up 23
4. Testing your Pepper module 25
    Running Unit Tests 25
5. little helpers 27
    XML helpers 27
        XML extractor 27

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. sequence diagram of Pepper workflow 12
3.3. corpus-structure represented in file structure 17
3.4. corpus-structure 17
3.5. file structure, where one document is encoded in multiple files 18
3.6. corpus-structure represented in file structure 23

List of Tables

3.1. map of `SElementId` and corresponding URI locations 17

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 mapping 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.

To make Pepper a pluggable framework, we used an underlying plug-in technology called OSGi (see: http://www.osgi.org/). OSGi is a mighty framework and has a lot of impact on 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 sample project which can be used as a template. Maven makes things easier for use especially in dealing with dependencies.

In the following, Chapter 2, *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 Chapter 3, *Just code it* we dig into the source code and explain the mechanism how a Pepper module works and interacts with the Pepper framework. In the section called “Customizing the mapping” we explain how to add properties to your module, so that the user can dynamically customize the behavior of the mapping. Since testing of software often is a pain in the back, the `pepper-testSuite` already comes with some predefined tests. This test bed is based on JUnit (see: junit.org) and can easily be extended for a specific module test. They should save you some time.

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 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 en 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[1], 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/saltnpepper.

[1] 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).

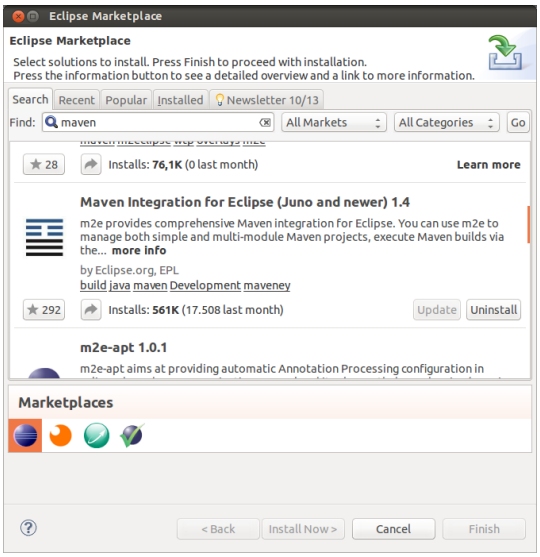
Chapter 2. Run your first Pepper module

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/) 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 [1]
2. Unzip 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/saltNpepper_full/)
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. Now you have an empty Eclipse. [2]
7. 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.
8. Open the 'Eclipse Marketplace' via 'Help' --> 'Eclipse Marketplace...' (see Figure 2.1, “Eclipse marketplace”)

Figure 2.1. Eclipse marketplace



[1] Eclipse is available in several flavors 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/keplersr1).

[2] When starting Eclipse for the first time, the first you will see is a welcome tab, just close this tab.

9. Type in *maven* in text box 'Find:' and press enter (see Figure 2.1, “Eclipse marketplace”).

10. 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”).

11. 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.

12. 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[3] with the command:

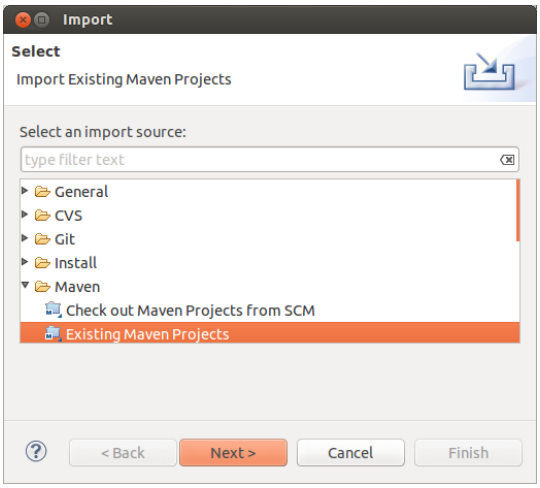
    ```
    svn export https://korpling.german.hu-berlin.de/svn/saltnpepper/pepper/tags/${project.version}/pepper-sampleModules/
    ```

    Download the project to a location of your choice, let's call it SAMPLE_HOME.

13. 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.

14. Click the entry 'Existing Maven Projects' as shown in figure Figure 2.2, “Eclipse Import dialogue”.

Figure 2.2. Eclipse Import dialogue



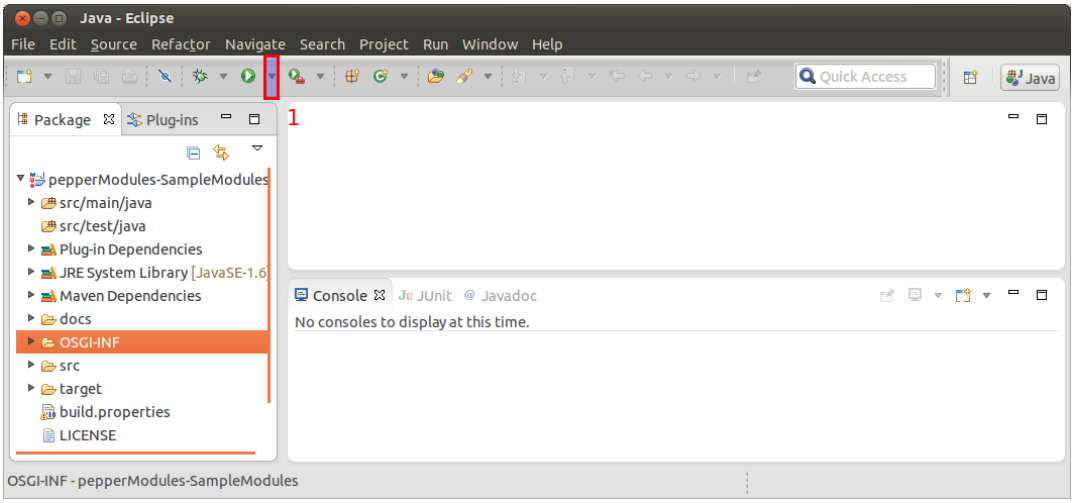
15. In the opening wizard choose SAMPLE_HOME as location of your Maven project and first click 'Next' than 'Finish'. Eclipse may asks you to install 'Tycho'[4]. Installing tycho and the necessary connectors could take some time.

16. Now Eclipse should import your first Pepper module project. After all, your Eclipse should look very similar to figure Figure 2.3, “Eclipse”).

[3] 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.

[4] Tycho is an Eclipse plug-in to connect the Maven integration with OSGi projects (see: http://www.eclipse.org/tycho/).

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.

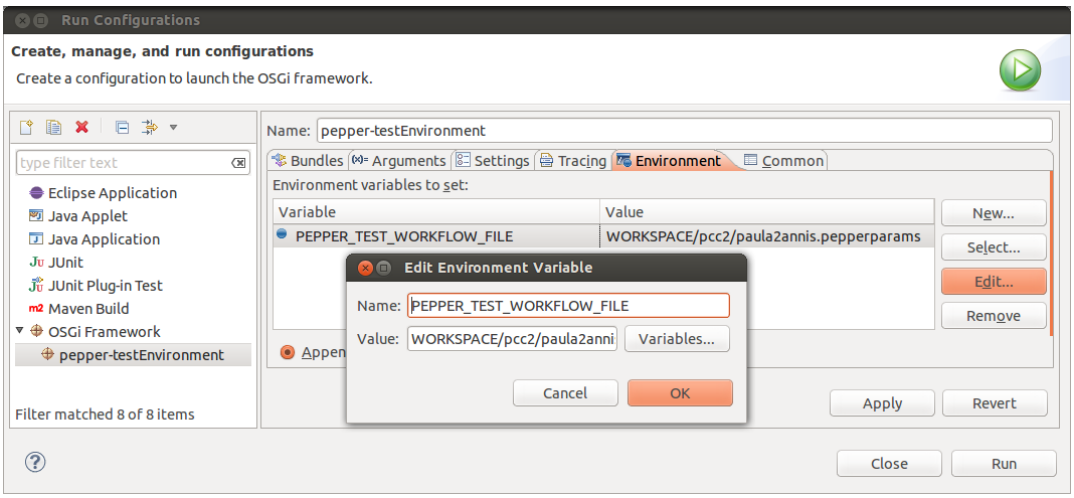
Let's have a first run

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'.[5]

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...'.

Figure 2.4. Run Configuration dialog



[5]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 'run_sampleProject' will be opened (similar to figure Figure 2.4, “Run Configuration dialog”)[6]. 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 (see figure Figure 2.4, “Run Configuration dialog”). The sample module comes with a predefined workflow 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.[7].

   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 in text box 'vm arguments' add the flag '-Dlogback.configurationFile=SAMPLE_HOME/src/test/resources/logback_test.xml' (if it is not already there). 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 welcome you and start the conversion producing some outputs to standard out. So what has happened?

   a. The framework starts and notifies about the registered modules, which are some more, than just our sample modules. This is because you copied all modules to the Eclipse dropin path.

   b. Pepper lists the workflow as defined in the workflow file.

   c. The SampleImporter imports a predefined corpus and logs some status messages to its progress.

   d. The SampleManipulator outputs 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 exports 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. **project structure**

   - 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...")

   - Rename the packages (when working with Eclipse, right click on the package name and choose the menu entry "Refactor" --> "Rename...")

2. **pom.xml**

   Open the pom.xml in the sample 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 pom.xml file. Here you will find a lot of outcommented 'TODO' lines. These comments are

[6] If this entry does not appear, open the file 'SAMPLE_HOME/src/test/resources/run_sampleProject.launch' by double clicking

[7] 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.

explanations to the following lines and guide you what could/ should be changed. In the following, this document also gives a description of what to be changed and addresses the xml element with the use of the XPath syntax.

- 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'.
- Change the "artifactId" (to the module name). You will find the entry under '/project/artifactId'.

3. **Java code**

   We recommend to delete the modules you do not want to implement for this time and just to keep the one(s) you are currently needing[8].

   - Change the name of the module, for instance to *MyImporter*, *MyExporter* etc. We recommend to use the formats name and the ending Importer, Exporter or Manipulator.[9]
   - Change the components name[10], 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
```

That have been all required changes to be made. But as you might have seen, a pom.xml contains a lot more entries to describe your projects. Some of these entries are very useful, depending on the Maven modules you want to use. Here we shortly describe how to adapt the optional entries, we have used in the sample project. You can also do the following adaptions later and skip them for now.

- Modify the description: Write what the module is supposed to do. You will find the entry under '/project/description'.
- Change the project homepage to the url of your project. You will find the entry under '/project/url'.
- 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'.
- 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'.
- Change the inception year to the current year. You will find the entry under '/project/inceptionYear'.
- Change the name of the organization to the one you are working for. You will find the entry under '/project/organization'.
- 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'.
- 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'.
- 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'.

[8] Otherwise your project will mess up with non-functional modules.
[9] Renaming the modules is important, when you want to use your modules together with others in Pepper. A user will not find your module, when every module has the name SampleImporter.
[10] a component here is an OSGi service component (for more details see http://wiki.osgi.org/wiki/Declarative_Services)

- 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 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"

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 (-Dlogback.configurationFile=SAMPLE_HOME/src/test/resources/logback_test.xml).
3. And the same goes for 'Environment', where you have to adapt the location of the WORKFLOW_DESCRIPTION_FILE reference.

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

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 check the SaltNPepper homepage under u.hu-berlin.de/saltnpepper or write an e-mail to <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 clean install assembly:single
```

Or use Eclipse by right clicking the project and then click 'Run as' --> 'Maven build...'. In form field 'Goals' type in 'clean install assembly:single'.

This will create a zip file in the target folder of your project 'SAMPLE_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 Maven is also installed for system-wide use and not only in Eclipse. For more information see the Maven site http://maven.apache.org/.

Congratulations

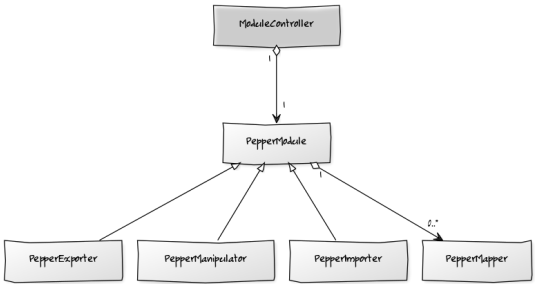
Now you are done, a new Pepper module is running under your name and you can provide it for download. So the work is done, is it? Unfortunately not, now we come to the hardest part ;-), the coding of your tasks. But don't be afraid, the next chapter is supposed to guide you through the jungle and to 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 retrievs 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`. No matter of what kind of module you are going to implement, it must inherit one of the three named types. 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 concrete Pepper module and the Pepper framework. 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[1]. 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.

A mapping task in the Pepper workflow is not a monolithic blog. 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 in 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]

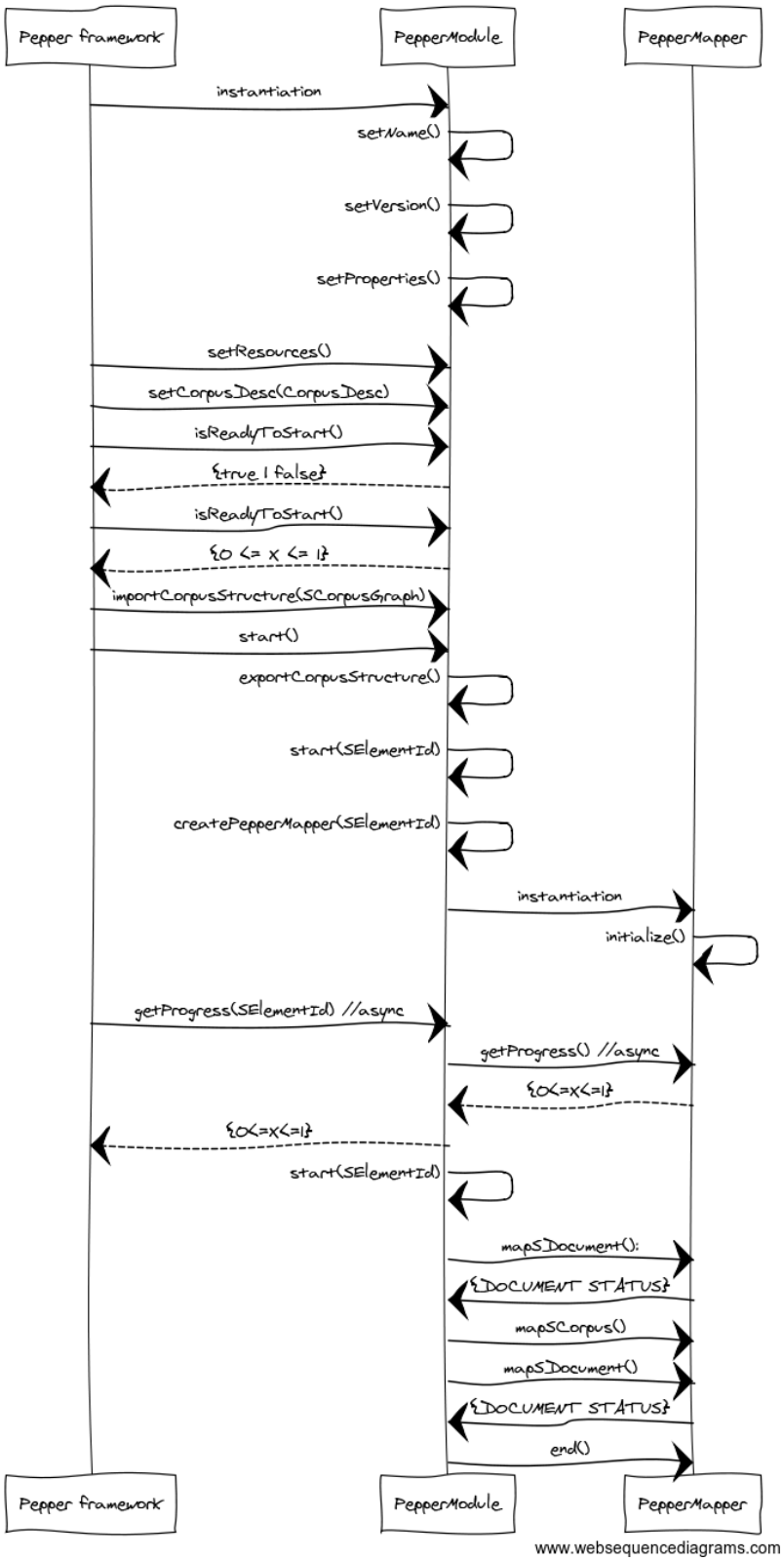
[1]This mechanism enables to map several document-structures and corpus-structures at once.

- customizing the mapping [recommended]
- monitoring the progress, logging and error handling [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, “sequence diagram of Pepper workflow” illustrates the communication between the framework, the `PepperModule` and `PepperMapper` class.

Figure 3.2. sequence diagram of Pepper workflow



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 technically be realized, we strongly recommend, to read the Salt model guide and the quick user guide on u.hu-berlin.de/saltnpepper/. We here primarily focus on the technical part of the Pepper workflow and especially on the Pepper modules. 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.

There are two aspects having a big impact on the inner architecture of a Pepper module. 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 defaults, 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 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 physical 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`. Although the main initializations, necessary for the workflow (e.g. passing the customization properties, see 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 annotated
       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());
       //6: create a primary text "Is this example..."
       STextualDS primaryText= getSDocument().getSDocumentGraph()
                .createSTextualDS("Is this example more complicated "
                    + "than it appears to be?");
       //7: create a meta-annotation
       getSDocument().createSMetaAnnotation(null, "author",
                   "Bart Simpson");
       //8: 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 you set the physical location at position 1 in method 'createPepperMapper()', you can now get that location via calling 'getResourceURI()' as shown at position 1 and 4 (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()', at 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 at position 6. In Salt each object can be annotated or meta-annotated, so do the `SDocument` objects, as shown at

position 7. Last but not least, both methods have to return a value describing whether the mapping was successful or not (see position 3 and 8). 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).

Usually you only need to return the `DOCUMENT_STATUS.COMPLETED` when everything was ok. In case of an error, Pepper will set the status `DOCUMENT_STATUS.FAILED` automatically, as long, as the exception is thrown[2], which marks it to be not processed any further.

During the mapping it is very helpful for the user, if you give some progress status from time to time. Especially when a mapping takes a longer term, it will keep the user from a frustrating experience to have a not responding tool. More information on that can be found in 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 section just handles ways to not using the default mechanisms and making more adaptions.

In a few cases, a format does not allow or difficulty allow to process it in parallel. 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()
                .getsDocumentId();
    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 initializes a PepperMapper instance **/
```

[2] If not and your module is catching the exception, you have to think about what to do in error case. If you want the document to be processed further by other modules than return `DOCUMENT_STATUS.COMPLETED`. If the document now might be corrupt, than return the `DOCUMENT_STATUS.FAILED`

```
@Override
public PepperMapper createPepperMapper(SElementId sElementId){
    ...
}
```

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

Note

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

Analyzing an unknown corpus

According to our experience 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. 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 analyses. Therefore Pepper provides a possibility to automatically detect the source format of a corpus. Unfortunately this task highly depends on 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 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 with the Pepper workflow (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 recursivly traverse through directories, in case 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/) start 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 provide 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 and Pepper ignores the module in automatic detection phase.

Im- and exporting corpus-structure

The classes `PepperImporter` and `PepperExporter` provide an automatic mechanism to im- or export the corpus-structure. This mechanism is adaptable step by step, according to your specific

purpose. Since many formats do not care about the corpus-structure and they only encode the document-structure, the corpus-structure is simultaneous to the file structure of a corpus.

Pepper's default mapping maps the root-folder to a root-corpus (`SCorpus` object). A sub-folder then 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` objects are linked with a `SCorpusDocumentRelation`. To get an impression of the described mapping, Figure 3.3, “corpus-structure represented in file structure” shows a file structure whereas Figure 3.4, “corpus-structure” shows the corresponding corpus-structure.

Figure 3.3. corpus-structure represented in file structure

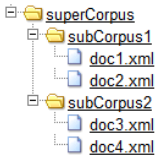
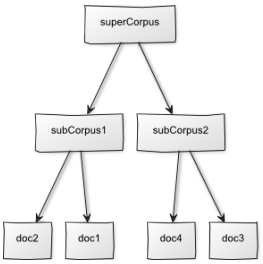


Figure 3.4. corpus-structure



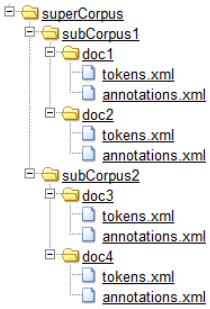
For keeping the correspondance between the corpus-structure and the file structure, both the im- and the exporter make use of a map, which can be accessed via '*getSElementId2ResourceTable()*'. Corresponding Figure 3.4, “corpus-structure” and Figure 3.3, “corpus-structure represented in file structure”, table Table 3.1, “map of `SElementId` and corresponding URI locations” shows the stored correletion between them.

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

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

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 3.5, “file structure, where one document is encoded in multiple files” shows an example for this kind of file structure, which also corresponds to the corpus-structure of Figure 3.4, “corpus-structure”.

Figure 3.5. file structure, where one document is encoded in multiple files



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

Importing the corpus-structure

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

```
importCorpusStructure(SCorpusGraph corpusGraph)
```

For totally changing 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 and puts them into 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 set 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 precisly two lists). An ignore list containing file endings, which are supposed to be ignored during the import and a list of file endings which are supposed to be used for the import.[3] Now let's show some code for adapting. The following snippet is placed into the method 'isReadyToStart()', but even could be located 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);
    ...
```

[3]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 wjat's the ending.

```
}
```

In general the paramter of the method '*getSDocumentEndings()*' is just a String, but there are some predefined endings 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 what's its ending. But the following line excludes all files having the ending 'txt'. Last but not least we look at 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, “file structure, where one document is encoded in multiple files”.

Exporting the corpus-structure

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

```
exportCorpusStructure()
```

It is invoked on top of the method '*start()*' of the `PepperExporter`. For totally changing the default behavior just override this method. The aim of this method is to fill the map of corresponding 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 `URI`s pointing to 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()'.

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 nothing will be mapped. Option 2 means that only `SCorpus` objects are mapped to a folder and `SDocument` objects will be ignored. And option 3 means 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 with 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 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 a 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 dependent on user customization. The past showed, that it is very frustrating for a user, when a Pepper module breaks up, because of not specifiied properties. You should always define a default behavior in case that the user has not specified a property.

Property

A property is just an attribute-value pair, consisting of a name called *property name* and a value called *property value*. Properties can be used for customizing the behavior of a mapping. Such a property must be specified by the user and determined in the Pepper workflow. 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 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 file.

One customization property in Pepper is represented with an object of type `PepperModuleProperty`. Such an object consists of the property's name, its datatype, a short description and a flag specifying whether this property is optional or mandatory as shown in the following snippet:

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

Even a default value could be passed:

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

To register a customization property, you need to add the created object to registry object, which is managed and accessed by the Pepper framework. The registry is realized via a specified `PepperModuleProperties` object. To create such an object, first implement a registry class as shown in the following snippet:

```
//...
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(){
    //...
    //1: adding a customization property to registry
    this.addProperty(new PepperModuleProperty<String>
        ("MyProp", String.class, "description of MyProp", true));
    //...
 }
 //2: return value of customization property
 public String getMyProp(){
  return((String)this.getProperty("MyProp").getValue());
 }
```

```
 //3: check constraints on customization property
 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);
}
```

The snippet shows on position 1 how to create, specify and register a `PepperModuleProperty`. The method 'getMyProp()' on position 2 shows the creation of a specialized method to access the property. Such a getter-method would be very helpful in your code to have a fast access without casting values in your mapping code. Position 3 shows the method 'checkProperty()', which can be used to check the passed property's value. Since customization properties or more specificly their values are entered manually by the user, it might be necessary to check the passed values. The Pepper framework calls this method before the mapping process is started. If a constraint fails the user will be informed immediately.

Last but not least, you need to initialize your property object. The best place for doing this 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());
```

You can access the `PepperModuleProperties` object at all places in your `PepperModule` object or your `PepperMapper` object. The following snippet shows how:

```
this.getProperty(String propName);
```

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 them a precise response about the progress of the conversion job.

Although Pepper is providing a mechanism to make the monitoring of the progress as simple as possible, a rest work for you remains to do. But don't get afraid, monitoring the progress just means the call of a single method.

When 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 process, than to divide the total process costs 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 costs of process.

```
//...
//map all STextualDS objects (one fifth if total process is done)
addProgress(0.2);
//map all SToken objects (two fifth if total process is done)
addProgress(0.2);
//map all SSpan objects (three fifth if total process is done)
addProgress(0.2);
//map all SStruct objects (four fifth if total process is done)
addProgress(0.2);
//map all SPointingRelation objects
//(process done, should be one of the last lines)
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 you do not want to use the default mechanism, you need to override the methods `getProgress(SElementId sDocumentId)` and `getProgress()` of `PepperModule`. The method `getProgress(SElementId sDocumentId)` shall return the progress of the `SDocument` or `SCorpus` object 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 framework (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. The following snippet shows the instantiation of the static logger[4] and its usage.

```
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");
```

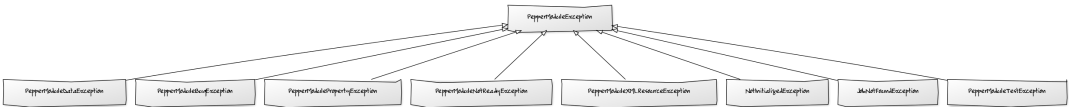
Error handling

Another important aspect of monitoring, is the monitoring when 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 precise description. Either to notify you, the module developer, or to find the bug in the data on her or his own. And unfortunately just a `NullPointerException` is not very useful to the user and is 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 Figure 3.6, “corpus-structure represented in file structure” gives an overview over the Exception classes for modules.

[4]Normally one logger is instantiated for exactly one class.

Figure 3.6. corpus-structure represented in file structure



The main and general class `PepperModuleException` can be used in case 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 the modules name etc. . A more detailed description of the exception classes could be found in the JavaDoc.

When an exception was thrown for a single document, Pepper will not abort the entire conversion process. Pepper will set the status of this document to `DOCUMENT_STATUS.FAILED` and will remove it from the rest of the workflow, so that modules coming afterwards will not process the corrupted document. Pepper will provide 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 other sections above, 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 things went wrong. If you now wonder where the best location should be, 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
    //supported formats
    addSupportedFormat("sample", "1.0", null);
    //using an own properties object
    setProperties(new SampleProperties());
}

@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 method 'isReadyToRun()', since it depends on some files in the resource folder[5]. Other reasons could be dependencies on the passed customizations, which are also set after the constructor was called.

[5]All files in the folder 'SAMPLE_HOME/src/main/resources' will be copied to the resource location in a modules distribution.

Sometimes it might be necessary to 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 you're done.

Chapter 4. Testing your Pepper module

Running Unit Tests

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 tests 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 (the three classes 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 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.

To set up the tests for the JUnit framework override the method `setUp()` as shown in the following snippet:

```
@Before
protected void setUp() throws Exception {
    //1: setting and initializing of class to test
    this.setFixture(new SampleImporterTest);
    //2: adding format information
    this.supportedFormatsCheck.add(new FormatDesc()
  .setFormatName("sample")
  .setFormatVersion("1.0"));
}
```

Since the JUnit framework can not know about the classes to test, you need to set and initialize it as shown on position 1. In case 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 needs to know the correct pairs of format name and version. Replace FORMAT_NAME and FORMAT_VERSION with your specific ones. You can even add more than one `FormatDefinition` object.

The provided test classes emulate 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 model 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(){
```

```
        getFixture().setCorpusDesc(new CorpusDesc().
            setCorpusPath(URI
                .createFileURI(PATH_TO_SAMPLE_CORPUS)));
        getFixture().getCorpusDesc().getFormatDesc()
            .setFormatName(FORMAT_NAME)
            .setFormatVersion(FORMAT_VERSION);

        //...

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

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

        //checks that the corpus graph object
        //is not empty any more
        assertNotNull(getFixture().getSaltProject()
            .getSCorpusGraphs());
        //checks that the corpus graph contains
        //X SCorpus objects
        assertEquals(X, getFixture().getSaltProject()
            .getSCorpusGraphs().getSCorpora().size());
    }
```

More samples for tests can be found in the sample project in the classes `SampleImporterTest`, `SampleManipulatorTest` and `SampleExporterTest`.

For testing an exporter, you may want to use Salt's sample generator (de.hu_berlin.german.korpling.saltnpepper.salt.samples.SampleGenerator). 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. For more information, take a look into the Salt quick user's guide (see u.hu-berlin.de/saltnpepper).

Chapter 5. little helpers

Pepper provides further little helpers, which have not been discussed yet. Some of them are just method calls, others are tiny programs, which can be used from command line. All of them are developed for special purposes and might be useful for one or another use case.

XML helpers

To read xml files Pepper provides a simple method, which shortcuts the instantiation of a SAX parser (see http://www.saxproject.org/quickstart.html). This method can be used in a derivation of `PepperImporter` as shown in the following snippet:

```
readXMLResource(contentHandler, documentLocation);
```

XML extractor

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 an 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>
```

results in the following interface, where INTERFACE_NAME is the name of the xml file:

```
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";
}
```

and in the following class:

```
public class INTERFACE_NAME_Reader extends DefaultHandler2
   implements INTERFACE_NAME{
   public void startElement(  String uri,
                              String localName,
                              String qName,
                              Attributes attributes)throws SAXException
```

```
   {
      if (TAG_TOKEN.equals(qName)){
      }else if (TAG_SENTENCE.equals(qName)){
      }
   }
}
```

For generating the class and interface you can either run the extractor as a library in your code, as shown in the following snippet:

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

or from the command line with the following call:

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