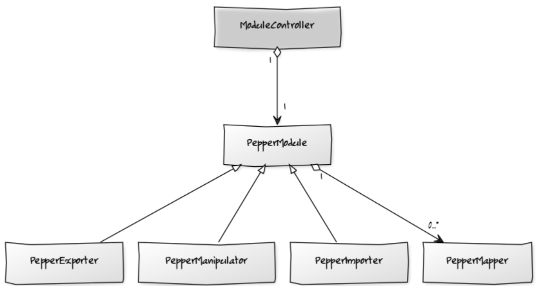
Chapter 1. Just code it

Table of Contents

With Pepper we tried to avoid as much complexity as possible without reducing the funtionality. 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 relaized by class derivation and call-back methods, which always can be overridden.

Pepper differentiates three sorts of modules: the importer, the manipulator and the exporter. An importer maps a corpus given in format X to a Salt model. A manipulator maps a Salt model to another Salt model, in terms of changing it or just retrieving some information. An exporter maps a Salt model to a corpus in format Y. All three modules `PepperImporter`, `PepperManipulator` and `PepperExporter` inherit the super type `PepperModule`. So no matter of what kind of module you are going to implement, it must inherit one of the three named types. Figure Figure 1.1, “class diagram showing the inheritance of Pepper module types”shows this relation.

Figure 1.1. class diagram showing the inheritance of Pepper module types



Now you might ask, what are the classes `ModuleController` and `PepperMapper` good for. The class `ModuleController` acts as a mediator between the Pepper framework and the concrete Pepper module. It initializes, starts and ends the modules processing. To explain the `PepperMapper` class, we want to give a short motivation: Since a mapping process can be relatively time consuming, we could increase the speed of mapping an entire corpus, if we are able to process mapping tasks simultaneously. Therefore we added mechanisms to run the process multi-threaded. Unfortunatly 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 devides and distributes the tasks of mapping corpora and documents to several `PepperModule` objects.

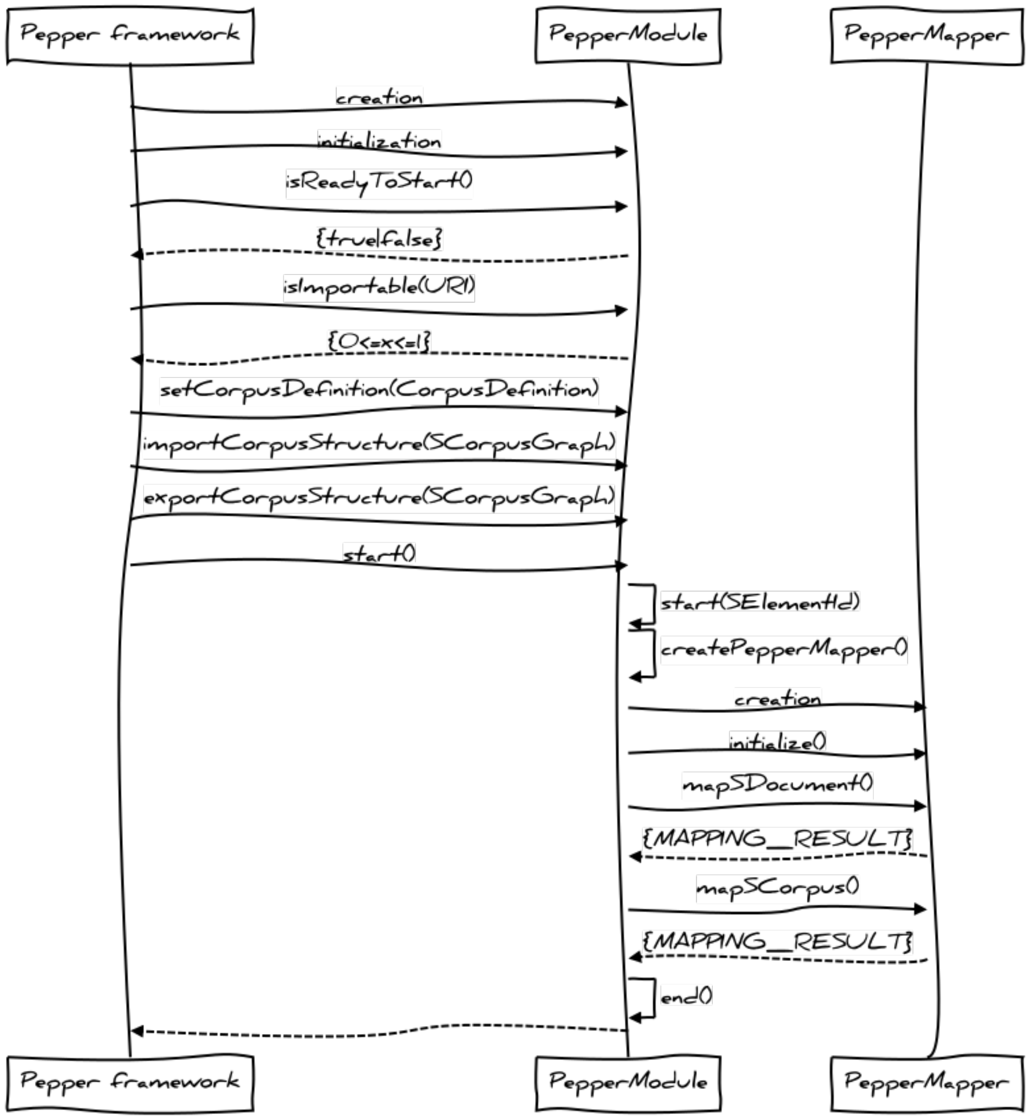
One step for a mapping in the Pepper workflow is not just a monolithic blog. It consists of several smaller conceptual aspects. Not each of the named aspects is essential, and some are belonging on the type of module you are implementing. Some are optional, some are recommanded and some are mandatory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

- mapping document-structure and corpus-structure [mandatory]
- analyzing an unknown corpus [recommanded, if module is an importer]
- im- and export corpus-structure [mandatory, if module is an im- or exporter]
- customizing the mapping [recommanded]
- error handling [recommanded]
- monitoring the progress [recommanded]
- 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 1.2, “class diagram showing the inheritance of Pepper module types” illustrates the communication between the framework, the `PepperModule` and `PepperMapper` class.

Figure 1.2. class diagram showing the inheritance of Pepper module types



Mapping document-structure and corpus-structure

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

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

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

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

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

```
public class SampleMapper implements PepperMapperImpl {

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

    @Override
    public DOCUMENT_STATUS mapSCorpus() {
     //1: returns the resource in case that a module is an importer or exporter
       getResourceURI();
       //2: getSCorpus() returns the SCorpus object, which for instance can be a
       getSCorpus().createSMetaAnnotation(null, "author", "Bart Simpson");
       //3: returns that the process was successful
       return(DOCUMENT_STATUS.COMPLETED);
    }

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

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

Note

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

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

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

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

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

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

```
setIsMultithreaded(false);
```

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

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

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

Note

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

Analyzing an unknown corpus

The experience has shown, that a lot of users, do not care a lot about formats and don't want to. 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 most users are not very interested, in the source format of a corpus, they just want to bring the corpus into any kind of tool to make further annotations or analysis. Therefore Pepper provides a possibility to automatically detect the source format of a corpus. 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, which makes it recommended.

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

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

```
<paula version="1.0">
<!-- ... -->
```

or

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

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

Im- and exporting corpus-structure

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

Pepper's default mapping maps the root folder (the direct `URI` location) to a root-corpus (`SCorpus` object). A sub-folder than corresponds to a sub-corpus (`SCorpus` object). The relation between super- and sub-corpus, is represented as a `SCorpusRelation` object. Following the assumption, that files contain the document-structure, there is one `SDocument` corresponding to each file in a sub-folder. The `SCorpus` and the `SDocument` object are linked with a `SCorpusDocumentRelation`. To get an impression of the described mapping, figure Figure 1.3, “corpus-structure represented in file-structure” shows a file structure and figure Figure 1.4, “corpus-structure” shows the corresponding corpus-structure.

Figure 1.3. corpus-structure represented in file-structure

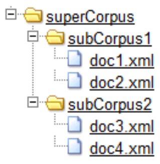
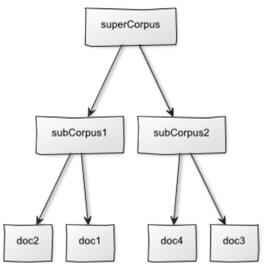
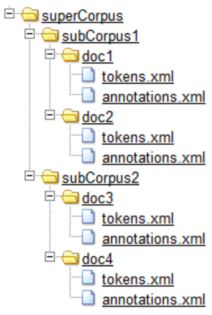


Figure 1.4. corpus-structure



Other formats do not encode the document-structure in just one file, they use a bunch of files instead. In that case the folder containing all the files (let's call it leaf-folder) corresponds to a `SDocument` object. Figure Figure 1.5, “” shows an example for these kind of file-structure, which corresponds also corresponds to the corpus-structure of figure Figure 1.4, “corpus-structure”.

Figure 1.5.



For keeping the correspondance between the corpus-structure and the file-structure, both the im- and the export of the corpus-structure make use of a map storing this correspondance. Corresponding to figure the corpus-structure of figure Figure 1.4, “corpus-structure” and the file-structure of figure Figure 1.3, “corpus-structure represented in file-structure” table Table 1.1, “map of `SElementId` and corresponding URI locations” shows the stored correletion between them.

Table 1.1. 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

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

Importing corpus-structure

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

```
importCorpusStructure(SCorpusGraph corpusGraph)
```

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

```
isReadyToStart()
```

. Back to figure Figure 1.3, “corpus-structure represented in file-structure”, the import mechanism traverses the file-structure beginning at the super-folder via the sub-folders to the files and creates a `SElementId` object corresponding to each folder or file to fill the map of table Table 1.1, “map of `SElementId` and corresponding URI locations”. This map is necessary for instance to retrieve the physical location of a document-structure during the mapping and can be accessed as shown in the following snippet:

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

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

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

In general the paramter of the method 'getSDocumentEndings()' is just a String, but there are some predefined ones you can use. The two lines marked as option 1, will add the endings 'xml' and 'tab' to the list of file endings to be imported. That means, that all files having one of these endings will be read and mapped to a document-structure. The first line of option 2, means to read each file, no matter on its ending. But the following line excludes all files having the ending 'txt'. Last but not least option 3 which is supposed to treat leaf-folders as document-structures and to create one `SDocument` object for each leaf-folder and not for each file, as mentioned in figure Figure 1.5, “”.

Exporting corpus-structure

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

```
exportCorpusStructure()
```

and invoked on top at the method 'start()' of the `PepperExporter`. For a totally change of the default behavior just override this method. The aim of this method is to fill the map between corpus-structure and file-structure (see table Table 1.1, “map of `SElementId` and corresponding URI locations”).

Note

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

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);
    ..
```

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

```
}
```

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

Customizing the mapping

When creating a mapping, it is often a matter of choice to map some data this way or another. In such cases it might be clever not to be that strict and allow only one 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 that the user has not specified one.

Property

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

Note

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

Property registration

The Pepper framework provides a registry mechanism to customize properties. This registry is called `PepperModuleProperties` and can be accessed via `getProperties()` and `setProperties()`. This class only represents a container object for a set of `PepperModuleProperty` objects and provides accessing methods. An instance of `PepperModuleProperty` represents an abstract description of a property and the concrete value at once. In the registration phase it belongs to the tasks of a `PepperModule` to specify the abstract description which consists of the name of the property, its datatype, a short description and a flag specifying whether this property is optional or mandatory. To create such an abstract description of a property use the constructor:

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

and pass the created property object to the property registry by calling the method `addProperty`. The Pepper framework uses the registry to first inform the user about usable properties for customization and second to 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.PepperModuleProperties;
import de.hu_berlin.german.korpling.saltnpepper.pepper
    .pepperModules.PepperModuleProperty;
//...
public class MyModuleProperties extends PepperModuleProperties {
    //...
    public MyModuleProperties(){
    //...
    this.addProperty(new PepperModuleProperty<String>
        ("MyProp", String.class, "description of MyProp", true));
    //...
 }
 //...
 public String getMyProp(){
  return((String)this.getProperty("MyProp").getValue());
 }
}
```

Checking property constraints

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

```
public boolean checkProperty(PepperModuleProperty<?> prop){
   //calls the check of constraints in parent,
   //for instance if a required value is set
   super.checkProperty(prop);
   if ("myProp".equals(prop.getName())){
        File file= (File)prop.getValue();
        //throws an exception, in case that the file does not exist
        if (!file.exists()){
            throw new PepperModuleException("The file "+
            "set to property 'myProp' does not exist.");
        }
   }
   return(true);
}
```

Initializing `MyModuleProperties`

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

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

Monitoring the progress

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

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

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

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

Note

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

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

Logging

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

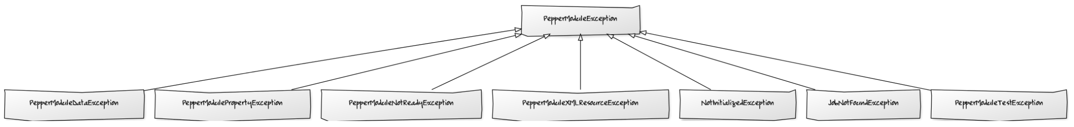
```
private static final Logger logger= LoggerFactory.getLogger(SampleImporter.clas
logger.trace("messages for the implementor");
logger.debug("message for the implementor and user");
logger.info("messages for the user");
logger.error("messages in case of an exception");
```

Error handling

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

Pepper provides a hierarchie 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 1.6, “corpus-structure represented in file-structure” gives an overview over the Exception classes for modules.

Figure 1.6. corpus-structure represented in file-structure


PepperModuleException
PepperModuleDataException
PepperModulePropertyException
PepperModuleNotReadyException
PepperModuleXMLResourceException
NotInitializedException
JobNotFoundException
PepperModuleTestException

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

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

Prepare and clean-up

The aspect of initialization is spread over two methods in a Pepper module. First the constructor of a module and second the method 'isReadyToStart()'. Both methods have been touched already in th eother paragraphs, but here we want to give a bundled overview about things concerning the initialization. Let us start with an explanaition 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()'. Unfortunatly, 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()' fullfills two tasks, first the initialization task and second, it returns a boolean value to determine, if the module can be started or if some things went wrong. If you now wonder, where should be the best location, to do your initialization, we recommend an eaarly-as-possible approach. The following snippet shows a smaple initialization:

```
public SampleImporter(){
    super();
    setName("SampleImporter");    //setting name of module
    setVersion("1.1.0");          //setting version of module
```

```
 addSupportedFormat("sample", "1.0", null);    //supported formats
    setProperties(new SampleProperties()); //using an own properties object
}

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

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

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

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

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

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