Chapter 1. Just code it

Table of Contents

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

In Salt and Pepper we differentiate between the corpus-structure and the document-structure. The document-structure contains the primary data (datasoruces) and the linguistic annotations. A bunch of such information is grouped to a document (`SDocument` in Salt). The corpus-structure now is a grouping mechanism to group a bunch of documents to a corpus or sub-corpus (`SCorpus` in Salt).

In Pepper we talk about three different kinds of modules, the `PepperImporter`, the `PepperExporter` and the `PepperManipulator`. An importer is used, to map data to a Salt model, an exporter is used to map a Salt model to other kind of data. A manipoulator is used to map one Salt model into another Salt model, for instance, for renaming data, merging data etc.. A mapping process is separated into three phases: import-phase, manipulation-phase and export-phase. In each phase an unbound number of Pepper modules can be used. Other than in the manipulation phase, in the import and export-phase at least one module must be involved. Each phase again is separated into mapping steps. A mapping step more or less is a pair of a Pepper module and a Salt object (which can be a `SCorpus` or a `SDocument`). In Salt, an `SDocument` object can be seen as a partition, allowing no links between objects contained in such a document to objects outside the document. The same goes for `SCorpus` objects. This allows, several steps to work independently from each other. Each module can be sure to be the only one processing a Salt object at a time. Since a mapping process can be relative time consuming, we could decrease the needed time to map an entire corpus, if we are able to process mapping tasks simultaneously.

In Pepper each module can be ran in multi-threading mode by default. But keep in mind, that multi-threading can not be added to a module as a new feature, it more needs an implementor to take care of during all the implementation work. If you are not familar with multi-threading in Java, we want to give the hint to avoid class variables and methods. The same goes for static variables and methods[1]. If it is still necessary, to use such constructs, we recommand, to go deeper in the sometimes annoying but powerfull world of multi-threading. If you do not want to blow up things and don't want to benefit of the better performance, you can also switch of multi-threading by a flag (just call the method `setIsMultithreaded(false)` in your modules constructor).

Enough of theory, lets step into the code now. In the SampleModules project, you will find three classes according to the three sorts of modules: `SampleImporter` (derived from `PepperImporter`), `SampleManipulator` (derived from `PepperManipulator`)and

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

`SampleExporter` (derived from `PepperExporter`). All of them are located in the package de.hu_berlin.german.korpling.saltnpepper.pepperModules.sampleModules. In most cases users do not want to implement all three sorts of modules. Users often want to implement only an importer an exporter or a manipulator. Then just delete the classes you won't implement. Otherwise, you will end up with non-functional modules in your project. In case of you want to have several importers, manipulators or exporters, just duplicate and rename the classes.

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

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

1. [MANDATORY]

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

2. [MANDATORY]

   Change the name of the component, for instance use the classes name and add 'Component' to it (e.g. MyImporterComponent) like in the following example.

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

3. [MANDATORY]

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

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

4. [OPTIONAL]

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

   ```
   public boolean isReadyToStart()
   ```

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

5. [RECOMMENDED]

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

6. [OPTIONAL]

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

7. [MANDATORY]

   The main interesting part of a mapping probably is the mapping of the document-structure. It means the mapping of the real linguistic data, so the reason why we are doing all of this. Since such a mapping is not even trivial, we recommand to delegate it to another class which is derived of class `PepperMapper`.

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

8. [RECOMMENDED]

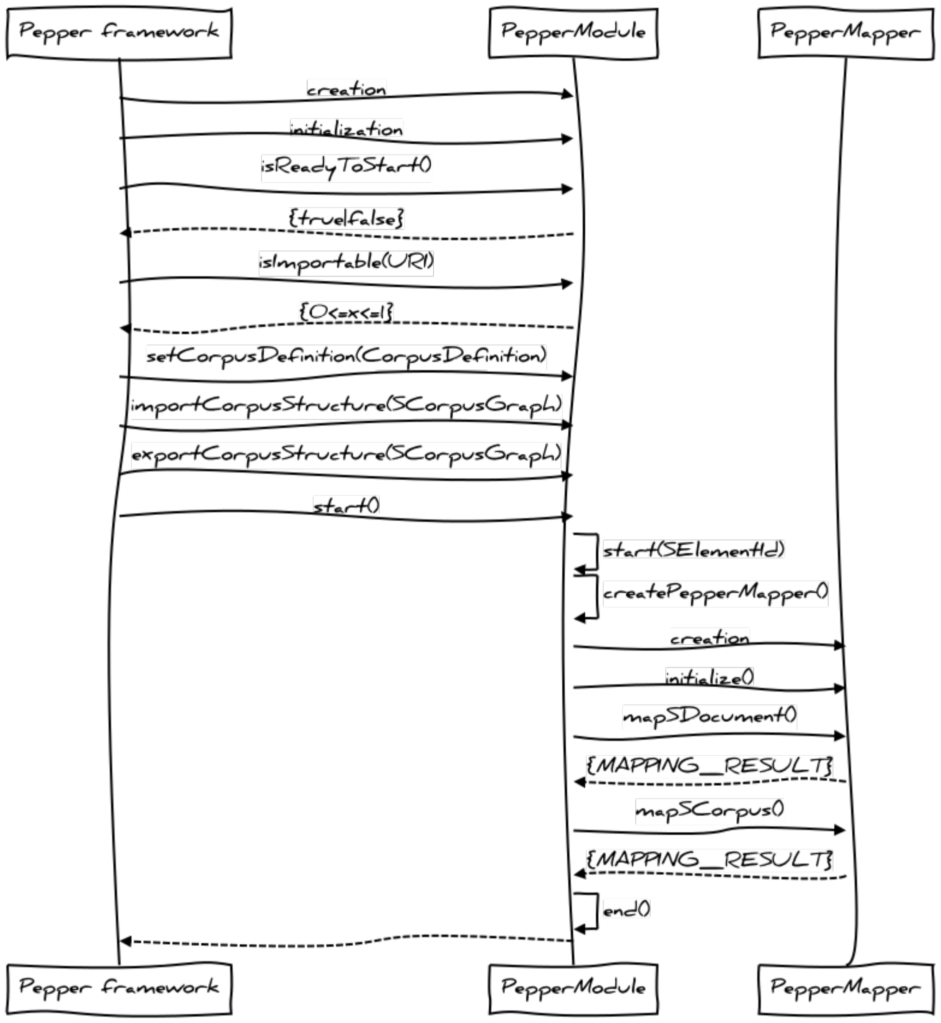
   Monitoring the progress would give a good feedback to the user. Since the conversion may take a while, we want to prevent the user to kill the conversion job, by providing the progress status. Therefore Pepper offers possibilities for a module to notify the framework about its progress. In case of you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

9. [OPTIONAL]

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

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

Figure 1.1. Pepper workflow overview



In the following sections, you will find more detailed explanaitions to the prior given single steps. If the given information aren't enough, please take a look into the corresponding JavaDoc.

Importing the corpus-structure

A corpus-structure consists of corpora (represented via the Salt element `SCorpus`), documents (represented via the Salt element `SDocument`), a linking between corpora and a linking between a corpus and a document (represented via the Salt element `SCorpusRelation` and `SCorpDocRelation`). Each corpus can contain 0..* subcorpus and 0..* documents, but a corpus cannot contain both a document and a corpus. For more information, please take a look into the Salt User Guide.

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

Peppers default mapping maps the root folder (the direct `URI` location) to a root-corpus (`SCorpus` object). For each sub-folder a sub-corpus (`SCorpus` object) is created and added to the `SCorpusGraph`. For each super- and sub-corpus relation, a `SCorpusRelation` object is created. For each interesting[2] file a `SDocument` object is created and added to the

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

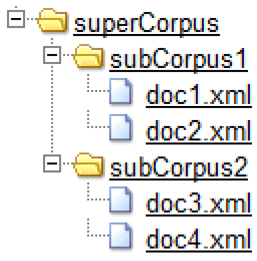
current `SCorpusGraph` object. The `SCorpus` and `SDocument` objects are connected via a `SCorpusDocumentRelation` object. While traversing the file-structure, an `SElementId` object is created representing the corpus-hierarchy and added to the created `SCorpus` or `SDocument` object. A map of these `SElementId` objects corresponding to the `URI` objects is returned, so that in the following methods like `start(SElementId)` this map can be used to identify the `URI` location of the `SDocument` objects.

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

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

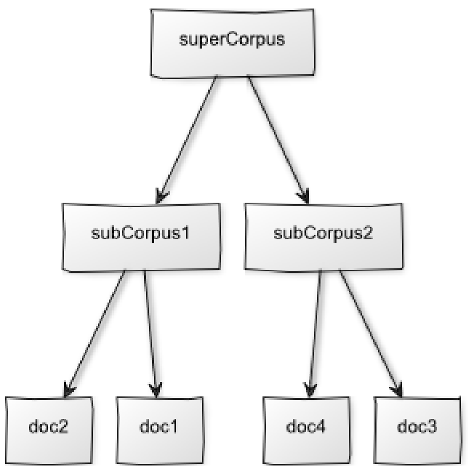
   Lets give an example. Imagine the following file-structure:

   Figure 1.2. corpus-structure represented in file-structure

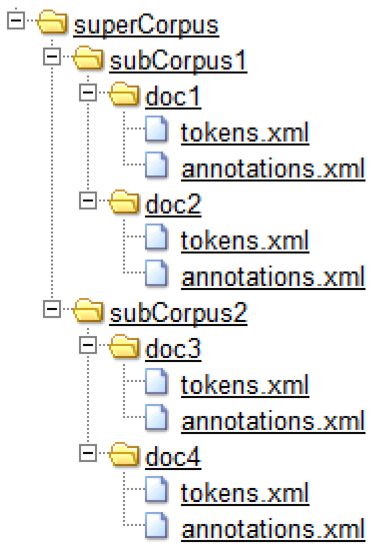
   To map this structure, you just have to add the ending 'xml' to the collection `sDocumentEndings`, you can do this using the constant `ENDING_XML` (even others are predefined). The resulting corpus-structure will look like shown in figure Figure 1.3, “corpus-structure”.

   Figure 1.3. corpus-structure

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

Figure 1.4.



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

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

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

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

3. The last opportunity to adopt the behavior is to override the method `importCorpusStructure()`

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

This method must generate a map containing the correspondence between a `SElementId` (representing a `SDocument` or a `SCorpus` object) and a resource. For instance, remember the sample of Figure 1.2, “corpus-structure represented in file-structure”, this could result in the following map:

Table 1.1.

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

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

Mapping the document-structure

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

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

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

default mechanism

The easiest and may be most beneficial way of creating a mapping is the use of the default mechanism, which means to derive the class `PepperMapper`. This also enables to run your module in multithreading mode without taking care of creating threads in Java. To register that class create a method named `createPepperMapper(SElementId sElementId)` returning your specific mapper object, for instance called MyMapper, see:

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

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

If you need to do some initializations before the methods `mapSDocument()` and or `mapSCorpus()` are called, but after the constructor has been called, just override the method `initialize()` (see figure Figure 1.1, “Pepper workflow overview”). This methods enables the possibility to make some initilizations depending on the values set by the framework.The general initialization, like setting the resource path and the `SDocument` or `SCorpus` object to be manipulated is done by the framework itself. Here we show an excerpt of the `PepperMapper` class.

```
public class PepperMapperImpl implements PepperMapper {

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

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

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

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

deeper adoption level 2

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

```
@Override
public void start(SElementId sElementId) throws PepperModuleException
{
    if (sElementId.getSIdentifiableElement() instanceof SCorpus)
    {
     //map for instance some meta-data
    }
    else if (sElementId.getSIdentifiableElement() instanceof SDocument)
    {
        //map the document-structure
    }
}
```

Note

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

Note

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

deeper adoption level 3

The class `PepperModule` further specifies the method `start()`, this is the most generic method in terms of mapping and is directly called by the Pepper framework or more precisly spoken by the `PepperModuleController`. To access the `SDocument` or `SCorpus`, you have to use the `PepperModuleController`, which can be accessed via `getPepperModuleController()`. Each `PepperModule` object has its own `PepperModuleController` object working as a communicator between the `PepperModule` object and the Pepper framework. Since a `PepperMoculeController` object is connected to the `PepperModuleController` object of the preceding and the following module, the objects to be processed (`SDocument` or `SCorpus`) are exchanged via a queue to which both controllers have access. The module you are implementing can get the current `SElementId` object via `getPepperModuleController().get()`. But take, care that this method will first return a result when the preceding module has processed its object. Until then the method will let your module wait.

Note

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

Cleaning up after all

Sometimes it might be necessary to make a clean up, after 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 all the processing is done, the Pepper framework calls the method `end()`. To run your clean up, just override it and your done.

Monitoring the progress

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

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

If you are using the default mapping mechanism by implementing the class `PepperMapper`, this class provides the methods `addProgress(Double progress)` and `setProgress(Double progress)` for this purpose. Both methods have a different semantic. `addProgress(Double progress)` will add the passed value to the current progress, whereas `setProgress(Double progress)` will override the entire progress. The passed value for progress must be a value between 0 for 0% and 1 for 100%. It is up to you to call one of the methods in your code and to estimate the progress. Often it is easier not to estimate the time needed for the progress, than to devide the entire progress in several steps and to return a progress for each step. For instance 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 of you do not want to use the default mechanism, you need to override the methods `getProgress(SElementId sDocumentId)` and `getProgress()` of your implementation of `PepperModule`.The method `getProgress(SElementId sDocumentId)` shall return the progress of the `SDocument` or `SCorpus` objects corresponding to the passed `SElementId`. Whereas the method `getProgress()` shall return the aggregated progress of all `SDocument` and `SCorpus` objects currently processed by your module.

Analyzing the unknown

Note

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

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 the most users are not very interested, in the source format of a coprus, they only want to bring it into any kind of tool to make further annotations or analysis. Therefore Pepper provides a possibility to automatically detect the source format of a corpus. Unfortunatly this task is very dependent of the format and the module processing the format. That makes the detection a task of the modules implementor. We are sorry. The mechanism of automatic detection is not a mandatory task, but it is very useful, what makes it recommanded.

The class `PepperImporter` defines the method `isImportable(URI corpusPath)` which can be overridden. The passed uri locates the entry point of the entire corpus as given in the Pepper workflow definition (so it points to the same location as `getCorpusDefinition().getCorpusPath()` does). Depending on the formats you want to support with your importer the detection can be very different. In the simplest case, it only is necessary, to search through the files at the given location (or to recursive traverse through directories, in case of the the 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">
<!-- ... -->
```

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

Customizing behaviour of your Pepper module

via properties

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

Property

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

Note

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

Property registration

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

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

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

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

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

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

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

checking property constraints

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

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

Initializing `MyModuleProperties`

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

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