Apache uimaFIT™

Apache UIMA™ Development Community

Version 3.3.0

User Guide

Introduction . . . 2
- Simplify Component Implementation . . . 2
- Simplify Component Instantiation . . . 2
  - From a class . . . 2
  - From an XML descriptor . . . 3
- Is this cheating? . . . 3
- Conclusion . . . 4

Getting Started . . . 5
- Adding uimaFIT to your project . . . 5
  - Maven users . . . 5
  - Non-Maven users . . . 5
- A simple analysis engine implementation . . . 5
- Running the analysis engine . . . 6
- Generate a descriptor file . . . 7

Pipelines . . . 8
- Examples . . . 9
- Tips & Tricks . . . 10

Validating CASes . . . 11
- Example use case . . . 11
- Defining a validation check . . . 11
- Registering the check for auto-detection . . . 12
- Validating a CAS . . . 12

Running Experiments . . . 13

CAS Utilities . . . 14
- Access methods . . . 14

Configuration Parameters . . . 16

External Resources . . . 19
- Resource injection . . . 19
  - Regular UIMA components . . . 19
  - uimaFIT-aware components . . . 20
  - Resources extending Resource_ImplBase . . . 22
  - Resources implementing SharedResourceObject . . . 23
  - Note on injecting resources into resources . . . 24
- Resource locators . . . 24

Type System Detection . . . 26
- Making types auto-detectable . . . 26
- Making index definitions and type priorities auto-detectable . . . 27
- Using type auto-detection . . . 27

Multiple META-INF/org.apache.uima.fit/types.txt files . . . 27
Performance note and caching . . . 28
Potential problems . . . 28
m2eclipse fails to copy descriptors to target/classes . . . 28
Class version conflicts . . . 28
Classes and resources in the default package . . . 29
Building an executable JAR . . . 30
uimaFIT Maven Plugin . . . 32
enhance goal . . . 32
generate goal . . . 34
Migration Guide . . . 36
Version 3.0.x to 3.1.x . . . 36
Version 2.x to 3.x . . . 36
Version 2.3.0 to 2.4.0 . . . 37
Version 2.2.0 to 2.3.0 . . . 38
Version 2.1.0 to 2.2.0 . . . 38
Version 2.0.0 to 2.1.0 . . . 38
Version 1.4.0 to 2.0.0 . . . 38

The document is a manual for users of uimaFIT, a friendly API to the Apache UIMA framework.

License and Disclaimer

Trademarks

Introduction

While uimaFIT provides many features for a UIMA developer, there are two overarching themes that most features fall under. These two sides of uimaFIT are,while complementary, largely independent of each other. One of the beauties of uimaFIT is that a developer that uses one side of uimaFIT extensively is not required to use the other side at all.

Simplify Component Implementation

The first broad theme of uimaFIT provides features that *simplify component implementation*. Our favorite example of this is the `@ConfigurationParameter` annotation which allows you to annotate a member variable as a configuration parameter. This annotation in combination with the method `ConfigurationParameterInitializer.initialize()` completely automates the process of initializing member variables with values from the `UimaContext` passed into your analysis engine's initialize method. Similarly, the annotation `@ExternalResource` annotation in combination with the method `ExternalResourceInitializer.initialize()` completely automates the binding of an external resource as defined in the `UimaContext` to a member variable. Dispensing with manually writing the code that performs these two tasks reduces effort, eliminates verbose and potentially buggy boiler-plate code, and makes implementing a UIMA component more enjoyable. Consider, for example, a member variable that is of type `Locale`. With uimaFIT you can simply annotate the member variable with `@ConfigurationParameter` and have your initialize method automatically initialize the variable correctly with a string value in the `UimaContext` such as `en_US`.

Simplify Component Instantiation

The second broad theme of uimaFIT provides features that *simplify component instantiation*. Working with UIMA, have you ever said to yourself "but I just want to tag some text!?" What does it take to "just tag some text?" Here's a list of things you must do with the traditional approach:

- wrap your tagger as a UIMA analysis engine
- write a descriptor file for your analysis engine
- write a CAS consumer that produces the desired output
- write another descriptor file for the CAS consumer
- write a descriptor file for a collection reader
- write a descriptor file that describes a pipeline
- invoke the Collection Processing Manager with your pipeline descriptor file

From a class

Each of these steps has its own pitfalls and can be rather time consuming. This is a rather unsatisfying answer to our simple desire to just tag some text. With uimaFIT you can literally eliminate all of these steps.

Here's a simple snippet of Java code that illustrates "tagging some text" with uimaFIT:

```
import static org.apache.uima.fit.factory.JCasFactory.createJCas;
import static org.apache.uima.fit.pipeline.SimplePipeline.runPipeline;
import static
 org.apache.uima.fit.factory.AnalysisEngineFactory.createEngineDescription;

JCas jCas = createJCas();

jCas.setDocumentText("some text");

runPipeline(jCas,
    createEngineDescription(MyTokenizer.class),
    createEngineDescription(MyTagger.class));

for (Token token : iterate(jCas, Token.class)){
    System.out.println(token.getTag());
}
```

This code uses several static method imports for brevity. And while the terseness of this code won't make a Python programmer blush - it is certainly much easier than the seven steps outlined above!

From an XML descriptor

uimaFIT provides mechanisms to instantiate and run UIMA components programmatically with or without descriptor files. For example, if you have a descriptor file for your analysis engine defined by `MyTagger` (as shown above), then you can instead instantiate the analysis engine with:

```
AnalysisEngineDescription tagger = createEngineDescription(
    "mypackage.MyTagger");
```

This will find the descriptor file *mypackage/MyTagger.xml* by name. Similarly, you can find a descriptor file by location with `createEngineDescriptionFromPath()`. However, if you want to dispense with XML descriptor files altogether (and you probably do), you can use the method `createEngineDescription()` as shown above. One of the driving motivations for creating the second side of uimaFIT is our frustration with descriptor files and our desire to eliminate them. Descriptor files are difficult to maintain because they are generally tightly coupled with java code, they decay without warning, they are wearisome to test, and they proliferate, among other reasons.

Is this cheating?

One question that is often raised by new uimaFIT users is whether or not it breaks the *UIMA way*. That is, does adopting uimaFIT lead me down a path of creating UIMA components and systems that are incompatible with the traditional UIMA approach? The answer to this question is *no*. For starters, uimaFIT does not skirt the UIMA mechanism of describing components - it only skips the XML part of it. For example, when the method `createEngineDescription()` is called (as shown above) an `AnalysisEngineDescription` is created for the analysis engine. This is the same object type that is instantiated when a descriptor file is used. So, instead of parsing XML to instantiate an analysis engine description from XML, uimaFIT uses a factory method to instantiate it from method

parameters. One of the happy benefits of this approach is that for a given `AnalysisEnginedDescription` you can generate an XML descriptor file using `AnalysisEngineDescription.toXML()`. So, uimaFIT actually provides a very simple and direct path for *generating* XML descriptor files rather than manually creating and maintaining them!

It is also useful to clarify that if you only want to use one side or the other of uimaFIT, then you are free to do so. This is possible precisely because uimaFIT does not workaround UIMA's mechanisms for describing components but rather uses them directly. For example, if the only thing you want to use in uimaFIT is the `@ConfigurationParameter`, then you can do so without worrying about what effect this will have on your descriptor files. This is because your analysis engine will be initialized with exactly the same `UimaContext` regardless of whether you instantiate your analysis engine in the *UIMA way* or use one of uimaFIT's factory methods. Similarly, a UIMA component does not need to be annotated with `@ConfiguratioParameter` for you to make use of the `createEngineDescription()` method. This is because when you pass configuration parameter values in to the `createEngineDescription()` method, they are added to an `AnalysisEngineDescription` which is used by UIMA to populate a `UimaContext` - just as it would if you used a descriptor file.

Conclusion

Because uimaFIT can be used to simplify component implementation and instantiation it is easy to assume that you can't do one without the other. This page has demonstrated that while these two sides of uimaFIT complement each other, they are not coupled together and each can be effectively used without the other. Similarly, by understanding how uimaFIT uses the UIMA component description mechanisms directly, one can be assured that uimaFIT enables UIMA development that is compatible and consistent with the UIMA standard and APIs.

Getting Started

This quick start tutorial demonstrates how to use uimaFIT to define and set a configuration parameter in an analysis engine, run it, and generate a descriptor file for it. The complete code for this example can be found in the *uimaFIT-examples* module.

Adding uimaFIT to your project

The following instructions describe how to add uimaFIT to your project's classpath.

Maven users

If you use Maven, then uimaFIT can be added to your project by simply adding uimaFIT as a project dependency by adding the following snippet of XML to your pom.xml file:

```
<dependency>
  <groupId>org.apache.uima</groupId>
  <artifactId>uimafit-core</artifactId>
  <version>3.3.0</version>
</dependency>
```

uimaFIT distributions are hosted by Maven Central and so no repository needs to be added to your pom.xml file.

Non-Maven users

If you do not build with Maven, then download uimaFIT from the Apache UIMA downloads page. The file name should be uimafit—bin.zip. Download and unpack this file. The contents of the resulting upacked directory will contain a directory called *lib*. Add all of the files in this directory to your classpath.

A simple analysis engine implementation

Here is the complete analysis engine implementation for this example.

```
public class GetStartedQuickAE
    extends org.apache.uima.fit.component.JCasAnnotator_ImplBase {

  public static final String PARAM_STRING = "stringParam";
  @ConfigurationParameter(name = PARAM_STRING)
  private String stringParam;

  @Override
  public void process(JCas jCas) throws AnalysisEngineProcessException {
    System.out.println("Hello world!  Say 'hi' to " + stringParam);
  }
}
```

The first thing to note is that the member variable `stringParam` is annotated with `@ConfigurationParameter` which tells uimaFIT that this is an analysis engine configuration parameter. It is best practice to create a public constant for the parameter name, here `PARAM_STRING` The second thing to note is that we extend uimaFIT's version of the `JCasAnnotator_ImplBase`. The initialize method of this super class calls:

```
ConfigurationParameterInitializer.initializeConfigurationParameters(
  Object, UimaContext)
```

which populates the configuration parameters with the appropriate contents of the `UimaContext`. If you do not want to extend uimaFIT's `JCasAnnotator_ImplBase`, then you can call this method directly in the `initialize` method of your analysis engine or any class that implements `Initializable`. You can call this method for an instance of any class that has configuration parameters.

Running the analysis engine

The following lines of code demonstrate how to instantiate and run the analysis engine from a main method:

```
JCas jCas = JCasFactory.createJCas();

AnalysisEngine analysisEngine = AnalysisEngineFactory.createEngine(
  GetStartedQuickAE.class,
  GetStartedQuickAE.PARAM_STRING, "uimaFIT");

analysisEngine.process(jCas);
```

In a more involved example, we would probably instantiate a collection reader and run this analysis engine over a collection of documents. Here, it suffices to simply create a `JCas`. Line 3 instantiates the analysis engine using `AnalysisEngineFactory` and sets the string parameter named `stringParam` to the value `uimaFIT`. Running this simple program sends the following output to the console:

```
Hello world!  Say 'hi' to uimaFIT
```

Normally you would be using a type system with your analysis components. When using uimaFIT, it is easiest to keep your type system descriptors in your source folders and make them known to uimaFIT. To do so, create a file *META-INF/org.apache.uima.fit/types.txt* in a source folder and add references to all your type descriptors to the file, one per line. You can also use wildcards. For example:

```
classpath*:org/apache/uima/fit/examples/type/Token.xml
classpath*:org/apache/uima/fit/examples/type/Sentence.xml
classpath*:org/apache/uima/fit/examples/tutorial/type/*.xml
```

Generate a descriptor file

The following lines of code demonstrate how a descriptor file can be generated using the class definition:

```
AnalysisEngine analysisEngine = AnalysisEngineFactory.createEngine(
  GetStartedQuickAE.class,
  GetStartedQuickAE.PARAM_STRING, "uimaFIT");

analysisEngineDescription.toXML(
  new FileOutputStream("GetStartedQuickAE.xml"));
```

If you open the resulting descriptor file you will see that the configuration parameter `stringParam` is defined with the value set to `uimaFIT`. We could now instantiate an analysis engine using this descriptor file with a line of code like this:

```
AnalysisEngineFactory.createEngine("GetStartedQuickAE");
```

But, of course, we really wouldn't want to do that now that we can instantiate analysis engines using the class definition as was done above!

This chapter, of course, did not demonstrate every feature of uimaFIT which provides support for annotating external resources, creating aggregate engines, running pipelines, testing components, among others.

Pipelines

UIMA is a component-based architecture that allows composing various processing components into a complex processing pipeline. A pipeline typically involves a *collection reader* which ingests documents and *analysis engines* that do the actual processing.

Normally, you would run a pipeline using a UIMA Collection Processing Engine or using UIMA AS. uimaFIT offers a third alternative that is much simpler to use and well suited for embedding UIMA pipelines into applications or for writing tests.

As uimaFIT does not supply any readers or processing components, we just assume that we have written three components:

- `TextReader` - reads text files from a directory
- `Tokenizer` - annotates tokens
- `TokenFrequencyWriter` - writes a list of tokens and their frequency to a file

We create descriptors for all components and run them as a pipeline:

```
CollectionReaderDescription reader =
  CollectionReaderFactory.createReaderDescription(
    TextReader.class,
    TextReader.PARAM_INPUT, "/home/uimafit/documents");

AnalysisEngineDescription tokenizer =
  AnalysisEngineFactory.createEngineDescription(
    Tokenizer.class);

AnalysisEngineDescription tokenFrequencyWriter =
  AnalysisEngineFactory.createEngineDescription(
    TokenFrequencyWriter.class,
    TokenFrequencyWriter.PARAM_OUTPUT, "counts.txt");

SimplePipeline.runPipeline(reader, tokenizer, writer);
```

Instead of running the full pipeline end-to-end, we can also process one document at a time and inspect the analysis results:

```
CollectionReaderDescription reader =
  CollectionReaderFactory.createReaderDescription(
    TextReader.class,
    TextReader.PARAM_INPUT, "/home/uimafit/documents");

AnalysisEngineDescription tokenizer =
  AnalysisEngineFactory.createEngineDescription(
    Tokenizer.class);

for (JCas jcas : SimplePipeline.iteratePipeline(reader, tokenizer)) {
  System.out.printf("Found %d tokens%n",
    JCasUtil.select(jcas, Token.class).size());
}
```

[[_ugr.tools.uimafit.testing"]] = Testing UIMA components

Writing tests without uimaFIT can be a laborious process that results in fragile tests that are very verbose and break easily when code is refactored. This page demonstrates how you can write tests that are both concise and robust. Here is an outline of how you might create a test for a UIMA component *without* uimaFIT:

- write a descriptor file that configures your component appropriately for the test. This requires a minimum of 30-50 lines of XML.
- begin a test with 5-10 lines of code that instantiate the e.g. analysis engine.
- run the analysis engine against some text and test the contents of the CAS.
- repeat steps 1-3 for your next test usually by copying the descriptor file, renaming it, and changing e.g. configuration parameters.</para>

If you have gone through the pain of creating tests like these and then decided you should refactor your code, then you know how tedious it is to maintain them.

Instead of pasting variants of the setup code (see step 2) into other tests we began to create a library of utility methods that we could call which helped shorten our code. We extended these methods so that we could instantiate our components directly without a descriptor file. These utility methods became the initial core of uimaFIT.

Examples

There are several examples that can be found in the *uimafit-examples* module.

- There are a number of examples of unit tests in both the test suite for the *uimafit-core* module and the *uimafit-examples* module. In particular, there are some well-documented unit tests in the latter which can be found in `RoomNumberAnnotator1Test`.
- You can improve your testing strategy by introducing a `TestBase` class such as the one found in `ExamplesTestBase`. This class is intended as a super class for your other test classes and sets up a `JCas` that is always ready to use along with a `TypeSystemDescription` and a `TypePriorities`. An example test that subclasses from `ExamplesTestBase` is `RoomNumberAnnotator2Test`.

- Most analysis engines that you want to test will generally be downstream of many other components that add annotations to the CAS. These annotations will likely need to be in the CAS so that a downstream analysis engine will do something sensible. This poses a problem for tests because it may be undesirable to set up and run an entire pipeline every time you want to test a downstream analysis engine. Furthermore, such tests can become fragile in the face of behavior changes to upstream components. For this reason, it can be advantageous to serialize a CAS as an XMI file and use this as a starting point rather than running an entire pipeline. An example of this approach can be found in `XmiTest`.

Tips & Tricks

The package <package>org.apache.uima.fit.testing</package> provides some utility classes that can be handy when writing tests for UIMA components. You may find the following suggestions useful:

- add a `TokenBuilder` to your `TestBase` class. An example of this can be found in `ComponentTestBase`. This makes it easy to add tokens and sentences to the CAS you are testing which is a common task for many tests.
- use a `JCasBuilder` to add text and annotations incrementally to a JCas instead of first setting the text and then adding all annotations.
- use a `CasDumpWriter` to write the CAS contents is a human readable format to a file or to the console. Compare this with a previously written and manually verified file to see if changes in the component result in changes of the components output.

Validating CASes

The uimaFIT CAS validation feature allows you to define consistency rules for your type system and to automatically check that CASes comply with these rules.

Example use case

Imagine a system which uses machine learning to automatically identify persons in a text. Such a system might define an annotation type called `Person` having a feature called `confidence` of type `float`. However, a requirement of the system should be that the confidence score must be within range from 0 to 1. Any value outside that range would probably be a bug in the systems implementation. Now imagine that you want to implement not only one, but a bunch of different UIMA analysis engines, each based on a different machine learning approach and plug these into the system. Instead of repeating the test code that checks the range of the confidence feature with each implementation, it would be much nicer if the range check could be included with the type system that all these implementations share. The unit tests should be able to pick this check (any any other consistency checks) up automatically and use them.

Defining a validation check

To define a validation check, all you need to do is to create a class implementing the `org.apache.uima.fit.validation.CasValidationCheck` interface. This interfaces defines a single method `List<CasValidationResult> check(CAS cas)`. Or if you prefer working against the JCas API, you can implement the `org.apache.uima.fit.validation.JCasValidationCheck` interface. Implementations of both interfaces (`CasValidationCheck` and `JCasValidationCheck`) can be applied to CAS as well as JCas instances - so it does not matter against which interface you build your check.

```
public class ConfidenceRangeCheck implements JCasValidationCheck {
  @Override
  public List<ValidationResult> validate(JCas aJCas) throws ValidationException {
    List<ValidationResult> results = new ArrayList<>();
    for (Person person : JCasUtil.select(aJCas, Person.class)) {
      if (person.getConfidence() < 0.0d || person.getConfidence() > 1.0d) {
        results.add(ValidationResult.error(this, "Invalid confidence score (%f) on %s
at [%d,%d]",
                person.getConfidence(), person.getType().getName(),
                person.getBegin(), person.getEnd()));
      }
    }
    return results;
  }
}
```

Checks are instantiated by the system as singletons. This means that their implementations must be stateless and must have a zero-argument constructor (or no constructor at all).

Registering the check for auto-detection

uimaFIT uses the Java Service Locator mechanism to locate validation check implementations. So to make a check available for auto-detection, its fully-qualified class name must be added to a file `META-INF/services/org.apache.uima.fit.validation.ValidationCheck`. Multiple checks can be added by putting each class name on separate lines.

Validating a CAS

The `org.apache.uima.fit.validation.Validator` class can be used to validate your (J)CASes. This class is typically constructed using a builder:

```
CAS cas = ...

// By default, the builder auto-detects all registered checks
Validator validator = new Validator.Builder().build();

// You could also pass in a JCas here instead of a CAS
ValidationSummary summary = validator.check(cas);
```

The output of a check is a `ValidationSummary` which contains a bunch of `ValidationResult` items. A `ValidationResult` essentially is a message with a severity level. When a summary contains any result with an error-level severity, the validation should be considered as failed.

The `Validator.Builder` can be configured, e.g. to exclude certain checks or to entirely disable the auto-detection of checks and instead work with only a set of explicitly specified checks.

Running Experiments

The *uimafit-examples* module contains a package org.apache.uima.fit.examples.experiment.pos which demonstrates a very simple experimental setup for testing a part-of-speech tagger. You may find this example more accessible if you check out the code from subversion and build it in your own environment.

The documentation for this example can be found in the code itself. Please refer to `RunExperiment` as a starting point. The following is copied from the javadoc comments of that file:

> `RunExperiment` demonstrates a very common (though simplified) experimental setup in which gold standard data is available for some task and you want to evaluate how well your analysis engine works against that data. Here we are evaluating `BaselineTagger` which is a (ridiculously) simple part-of-speech tagger against the part-of-speech tags found in `src/main/resources/org/apache/uima/fit/examples/pos/sample-gold.txt`

The basic strategy is as follows:

- post the data *as is* into the default view,
- parse the gold-standard tokens and part-of-speech tags and put the results into another view we will call *GOLD_VIEW*,
- create another view called *SYSTEM_VIEW* and copy the text and `Token` annotations from the *GOLD_VIEW* into this view,
- run the `BaselineTagger` on the *SYSTEM_VIEW* over the copied `Token` annoations,
- evaluate the part-of-speech tags found in the *SYSTEM_VIEW* with those in the *GOLD_VIEW*.

CAS Utilities

uimaFIT facilitates working with the CAS and JCas by offering various convenient methods for accessing and navigating annotations and feature structures. Additionally, the the convenience methods for JCas access are fully type-safe and return the JCas type or a collection of the JCas type which you wanted to access.

Access methods

uimaFIT supports the following convenience methods for accessing CAS and JCas structures. All methods respect the UIMA index definitions and return annotations or feature structures in the order defined by the indexes. Unless the default UIMA index for annotations has been overwritten, annotations are returned sorted by begin (increasing) and end (decreasing).

- `select(cas, type)` - fetch all annotations of the given type from the CAS/JCas. Variants of this method also exist to fetch annotations from a `FSList` or `FSArray`.
- `selectAll(cas)` - fetch all annotations from the CAS or fetch all feature structures from the JCas.
- `selectBetween(type, annotation1, annotation2)`* - fetch all annotations between the given two annotations.
- `selectCovered(type, annotation)`* - fetch all annotations covered by the given annotation. If this operation is used intensively, `indexCovered(…)` should be used to pre-calculate annotation covering information.
- `selectCovering(type, annotation)*` - fetch all annotations covering the given annotation. If this operation is used intensively, `indexCovering(…)` should be used to pre-calculate annotation covering information.
- `selectByIndex(cas, type, n)` - fetch the n-th feature structure of the given type.
- `selectSingle(cas, type)` - fetch the single feature structure of the given type. An exception is thrown if there is not exactly one feature structure of the type.
- `selectSingleRelative(type, annotation, n)`* - fetch a single annotation relative to the given annotation. A positive `n` fetches the n-th annotation right of the specified annotation, while the a negative `n` fetches to the left.
- `selectPreceding(type, annotation, n)`* - fetch the n annotations preceding the given annotation. If there are less then n preceding annotations, all preceding annotations are returned.
- `selectFollowing(type, annotation, n)`* - fetch the n annotations following the given annotation. If there are less then n following annotations, all following annotations are returned.

For historical reasons, the method marked with * also exist in a version that accepts a CAS/JCas as the first argument. These may not work as expected when the annoation arguments provided to the method are from a different CAS/JCas/view. Also, for any method accepting two annotations, these should come from the same CAS/JCas/view. In future, the potentially problematic signatures may be deprecated, removed, or throw exeptions if these conditions are not met.

You should expect the structures returned by these methods to be backed by the CAS/JCas contents. In particular, if you remove any feature structures from the CAS while iterating over these structures may cause failures. For this reason, you should also not hold on to these structures longer than necessary, as is the case for UIMA `FSIterator`s as well.

Depending on whether one works with a CAS or JCas, the respective methods are available from the JCasUtil or CasUtil classes.

JCasUtil expect a JCas wrapper class for the `type` argument, e.g. `select(jcas, Token.class)` and return this type or a collection using this generic type. Any subtypes of the specified type are returned as well. CasUtil expects a UIMA `Type` instance. For conveniently getting these, CasUtil offers the methods `getType(CAS, Class<?>)` or `getType(CAS, String)` which fetch a type either by its JCas wrapper class or by its name.

Unless annotations are specifically required, e.g. because begin/end offsets are required, the JCasUtil methods can be used to access any feature structure inheriting from `TOP`, not only annotations. The CasUtil methods generally work only on annotations. Alternative methods ending in "FS" are provided for accessing arbitrary feature structures, e.g. `selectFS`.

Examples:

```
// CAS version
Type tokenType = CasUtil.getType(cas, "my.Token");
for (AnnotationFS token : CasUtil.select(cas, tokenType)) {
  ...
}

// JCas version
for (Token token : JCasUtil.select(jcas, Token.class)) {
  ...
}
```

Configuration Parameters

uimaFIT defines the `@ConfigurationParameter` annotation which can be used to annotate the fields of an analysis engine or collection reader. The purpose of this annotation is twofold:

- injection of parameters from the UIMA context into fields
- declaration of parameter metadata (mandatory, default value, description) which can be used to generate XML descriptors

In a regular UIMA component, parameters need to be manually extracted from the UIMA context, typically requiring a type cast.

```
class MyAnalysisEngine extends CasAnnotator_ImplBase {
  public static final String PARAM_SOURCE_DIRECTORY = "sourceDirectory";
  private File sourceDirectory;

  public void initialize(UimaContext context)
      throws ResourceInitializationException {

    sourceDirectory = new File((String) context.getConfigParameterValue(
      PARAM_SOURCE_DIRECTORY));
  }
}
```

The component has no way to declare a default value or to declare if a parameter is optional or mandatory. In addition, any documentation needs to be maintained in !JavaDoc and in the XML descriptor for the component.

With uimaFIT, all this information can be declared in the component using the `@ConfigurationParameter` annotation.

Table 1. `@ConfigurationParameter` annotation

Parameter	Description	Default
name	parameter name	name of annotated field
description	description of the parameter	
mandatory	whether a non-null value must be specified	true
defaultValue	the default value if no value is specified	

```
class MyAnalysisEngine
    extends org.apache.uima.fit.component.CasAnnotator_ImplBase {

  /**
   * Directory to read the data from.
   */
  public static final String PARAM_SOURCE_DIRECTORY = "sourceDirectory";
  @ConfigurationParameter(name=PARAM_SOURCE_DIRECTORY, defaultValue=".")
  private File sourceDirectory;
}
```

Note, that it is no longer necessary to implement the `initialize()` method. uimaFIT takes care of locating the parameter `sourceDirectory` in the UIMA context. It recognizes that the `File` class has a `String` constructor and uses that to instantiate a new `File` object from the parameter. A parameter is mandatory unless specified otherwise. If a mandatory parameter is not specified in the context, an exception is thrown.

The `defaultValue` is used when generating an UIMA component description from the class. It should be pointed out in particular, that uimaFIT does not make use of the default value when injecting parameters into fields. For this reason, it is possible to have a parameter that is mandatory but does have a default value. The default value is used as a parameter value when a component description is generated via the uimaFIT factories unless a parameter is specified in the factory call. If a component description in created manually without specifying a value for a mandatory parameter, uimaFIT will generate an exception.

> You can use the *enhance* goal of the uimaFIT Maven plugin to pick up the parameter description from the JavaDoc and post it to the `description` field of the `@ConfigurationParameter` annotation. This should be preferred to specifying the description explicitly as part of the annotation.

The parameter injection mechanism is implemented in the `ConfigurationParameterInitializer` class. uimaFIT provides several base classes that already come with an `initialize()` method using the initializer:

- `CasAnnotator_ImplBase`
- `CasCollectionReader_ImplBase`
- `CasConsumer_ImplBase`
- `CasFlowController_ImplBase`
- `CasMultiplier_ImplBase`
- `JCasAnnotator_ImplBase`
- `JCasCollectionReader_ImplBase`
- `JCasConsumer_ImplBase`
- `JCasFlowController_ImplBase`
- `JCasMultiplier_ImplBase`

- `Resource_ImplBase`

The `ConfigurationParameterInitializer` can also be used with shared resources:

```
class MySharedResourceObject implements SharedResourceObject {
  public static final String PARAM_VALUE = "Value";
  @ConfigurationParameter(name = PARAM_VALUE, mandatory = true)
  private String value;

  public void load(DataResource aData)
      throws ResourceInitializationException {

    ConfigurationParameterInitializer.initialize(this, aData);
  }
}
```

Fields that can be annotated with the `@ConfigurationParameter` annotation are any array or collection types (including if they are only typed via interfaces such as `List` or `Set`) of primitive types (`int`, `boolean`, `float`, `double`). Enum types, as well as, fields of the types `Charset`, `File`, `Locale`, `Pattern`, `URI`, and `URL` can also be used. These can be initialized either using an object value (e.g. `StandardChartsets.UTF_8`) or a string value (e.g. `"UTF-8"`). Additionally it is possible to inject any fields of types that define a constructor accepting a single `String`. These must be initialized from a string value.

Multi-valued parameters can be initialized from single values without having to wrap these into a container.

External Resources

An analysis engine often uses some data model. This may be as simple as word frequency counts or as complex as the model of a parser. Often these models can become quite large. If an analysis engine is deployed multiple times in the same pipeline or runs on multiple CPU cores, memory can be saved by using a shared instance of the data model. UIMA supports such a scenario by so-called external resources. The following sections illustrates how external resources can be used with uimaFIT.

First create a class for the shared data model. Usually this class would load its data from some URI and then expose it via its methods. An example would be to load word frequency counts and to provide a `getFrequency()` method. In our simple example we do not load anything from the provided URI - we just offer a method to get the URI from which data be loaded.

```
// Simple model that only stores the URI it was loaded from. Normally data
// would be loaded from the URI instead and made accessible through methods
// in this class. This simple example only allows accessing the URI.
public static final class SharedModel implements SharedResourceObject {
  private String uri;

  public void load(DataResource aData)
      throws ResourceInitializationException {

    uri = aData.getUri().toString();
  }

  public String getUri() { return uri; }
}
```

Resource injection

Regular UIMA components

When an external resource is used in a regular UIMA component, it is usually fetched from the context, cast and copied to a class member variable.

```
class MyAnalysisEngine extends CasAnnotator_ImplBase {
  final static String MODEL_KEY = "Model";
  private SharedModel model;

  public void initialize(UimaContext context)
      throws ResourceInitializationException {

    configuredResource = (SharedModel)
      getContext().getResourceObject(MODEL_KEY);
  }
}
```

uimaFIT can be used to inject external resources into such traditional components using the `createDependencyAndBind()` method. To show that this works with any off-the-shelf UIMA component, the following example uses uimaFIT to configure the OpenNLP Tokenizer:

```
// Create descriptor
AnalysisEngineDescription tokenizer = createEngineDescription(
  Tokenizer.class,
  UimaUtil.TOKEN_TYPE_PARAMETER, Token.class.getName(),
  UimaUtil.SENTENCE_TYPE_PARAMETER, Sentence.class.getName());

// Create the external resource dependency for the model and bind it
createDependencyAndBind(tokenizer, UimaUtil.MODEL_PARAMETER,
  TokenizerModelResourceImpl.class,
  "http://opennlp.sourceforge.net/models-1.5/en-token.bin");
```

We recommend declaring parameter constants in the classes that use them, e.g. here in `Tokenizer`. This way, the parameters for a class can be found easily. However, OpenNLP declares parameters centrally in `UimaUtil`. Thus, the example above is correct, although unconventional.

Note that uimaFIT is unable to perform type-coercion on parameters if a descriptor is created from a class that does not contain `@ConfigurationParameter` annotations, such as the OpenNLP `Tokenizer`. Such a descriptor does not contain any parameter declarations! However, it is still possible to configure such a component using uimaFIT by passing exactly the expected types as parameter values. Thus, we need use the `getName()` method to get the class name as a string, instead of simply passing the class itself. Also, setting multi-valued parameter from a list or single value does not work here. Multi-values parameters must be passed as an array of the required type. Only the default UIMA types are possible: `String`, `boolean`, `int`, and `float`.

uimaFIT-aware components

uimaFIT provides the `@ExternalResource` annotation to inject external resources directly into class

member variables.

Table 2. `@ExternalResource` annotation

Parameter	Description	Default
key	Resource key	field name
api	Used when the external resource type is different from the field type, e.g. when using an ExternalResourceLocator	field type
mandatory	Whether a value must be specified	true

```
// Example annotator that uses the SharedModel. In the process() we only
// test if the model was properly initialized by uimaFIT
public static class Annotator
    extends org.apache.uima.fit.component.JCasAnnotator_ImplBase {

  final static String MODEL_KEY = "Model";
  @ExternalResource(key = MODEL_KEY)
  private SharedModel model;

  public void process(JCas aJCas) throws AnalysisEngineProcessException {
    assertTrue(model.getUri().endsWith("gene_model_v02.bin"));
    // Prints the instance ID to the console - this proves the same
    // instance of the SharedModel is used in both Annotator instances.
    System.out.println(model);
  }
}
```

Note, that it is no longer necessary to implement the `initialize()` method. uimaFIT takes care of locating the external resource `Model` in the UIMA context and assigns it to the field `model`. If a mandatory resource is not present in the context, an exception is thrown.

The resource injection mechanism is implemented in the `ExternalResourceInitializer` class. uimaFIT provides several base classes that already come with an `initialize()` method using the initializer:

- `CasAnnotator_ImplBase`
- `CasCollectionReader_ImplBase`
- `CasConsumer_ImplBase`
- `CasFlowController_ImplBase`
- `CasMultiplier_ImplBase`
- `JCasAnnotator_ImplBase`
- `JCasCollectionReader_ImplBase`

- `JCasConsumer_ImplBase`
- `JCasFlowController_ImplBase`
- `JCasMultiplier_ImplBase`
- `Resource_ImplBase`

When building a pipeline, external resources can be set of a component just like configuration parameters. External resources and configuration parameters can be mixed and appear in any order when creating a component description.

Note that in the following example, we create only one external resource description and use it to configure two different analysis engines. Because we only use a single description, also only a single instance of the external resource is created and shared between the two engines.

```
ExternalResourceDescription extDesc = createSharedResourceDescription(
  SharedModel.class, new File("somemodel.bin"));

// Binding external resource to each Annotator individually
AnalysisEngineDescription aed1 = createEngineDescription(
  Annotator.class,
  Annotator.MODEL_KEY, extDesc);

AnalysisEngineDescription aed2 = createEngineDescription(
  Annotator.class,
  Annotator.MODEL_KEY, extDesc);

// Check the external resource was injected
AnalysisEngineDescription aaed = createEngineDescription(aed1, aed2);
AnalysisEngine ae = createEngine(aaed);
ae.process(ae.newJCas());
```

This example is given as a full JUnit-based example in the the *uimaFIT-examples* project.

Resources extending Resource_ImplBase

One kind of resources extend `Resource_ImplBase`. These are the easiest to handle, because uimaFIT's version of `Resource_ImplBase` already implements the necessary logic. Just be sure to call `super.initialize()` when overriding `initialize()`. Also mind that external resources are not available yet when `initialize()` is called. For any initialization logic that requires resources, override and implement `afterResourcesInitialized()`. Other than that, injection of external resources works as usual.

```
public static class ChainableResource extends Resource_ImplBase {
  public final static String PARAM_CHAINED_RESOURCE = "chainedResource";
  @ExternalResource(key = PARAM_CHAINED_RESOURCE)
  private ChainableResource chainedResource;

  public void afterResourcesInitialized() {
    // init logic that requires external resources
  }
}
```

Resources implementing SharedResourceObject

The other kind of resources implement `SharedResourceObject`. Since this is an interface, uimaFIT cannot provide the initialization logic, so you have to implement a couple of things in the resource:

- implement `ExternalResourceAware`
- declare a configuration parameter `ExternalResourceFactory.PARAM_RESOURCE_NAME` and return its value in `getResourceName()`
- invoke `ConfigurationParameterInitializer.initialize()` in the `load()` method.

Again, mind that external resource not properly initialized until uimaFIT invokes `afterResourcesInitialized()`.

```
public class TestSharedResourceObject implements
    SharedResourceObject, ExternalResourceAware {

  @ConfigurationParameter(name=ExternalResourceFactory.PARAM_RESOURCE_NAME)
  private String resourceName;

  public final static String PARAM_CHAINED_RESOURCE = "chainedResource";
  @ExternalResource(key = PARAM_CHAINED_RESOURCE)
  private ChainableResource chainedResource;

  public String getResourceName() {
    return resourceName;
  }

  public void load(DataResource aData)
      throws ResourceInitializationException {

    ConfigurationParameterInitializer.initialize(this, aData);
    // rest of the init logic that does not require external resources
  }

  public void afterResourcesInitialized() {
   // init logic that requires external resources
  }
}
```

Note on injecting resources into resources

Nested resources are only initialized if they are used in a pipeline which contains at least one component that calls `ConfigurationParameterInitializer.initialize()`. Any component extending uimaFIT's component base classes qualifies. If you use nested resources in a pipeline without any uimaFIT-aware components, you can just add uimaFIT's `NoopAnnotator` to the pipeline.

Resource locators

Normally, in UIMA an external resource needs to implement either `SharedResourceObject` or `Resource`. In order to inject arbitrary objects, uimaFIT has the concept of `ExternalResourceLocator`. When a resource implements this interface, not the resource itself is injected, but the method `getResource()` is called on the resource and the result is injected. The following example illustrates how to inject an object from JNDI into a UIMA component:

```
class MyAnalysisEngine2 extends JCasAnnotator_ImplBase {
  static final String RES_DICTIONARY = "dictionary";
  @ExternalResource(key = RES_DICTIONARY)
  Dictionary dictionary;
}

AnalysisEngineDescription desc = createEngineDescription(
  MyAnalysisEngine2.class);

bindResource(desc, MyAnalysisEngine2.RES_DICTIONARY,
  JndiResourceLocator.class,
  JndiResourceLocator.PARAM_NAME, "dictionaries/german");
```

Type System Detection

UIMA requires that types that are used in the CAS are defined in XML files - so-called *type system descriptions* (TSD). Whenever a UIMA component is created, it must be associated with such a type system. While it is possible to manually load the type system descriptors and pass them to each UIMA component and to each created CAS, it is quite inconvenient to do so. For this reason, uimaFIT supports the automatic detection of such files in the classpath. Thus is becomes possible for a UIMA component provider to have component's type automatically detected and thus the components becomes immediately usable by adding it to the classpath.

Making types auto-detectable

The provider of a type system should create a file *META-INF/org.apache.uima.fit/types.txt* in the classpath. This file should define the locations of the type system descriptions. Assume that a type `org.apache.uima.fit.type.Token` is specified in the TSD *org/apache/uima/fit/type/Token.xml*, then the file should have the following contents:

```
classpath*:org/apache/uima/fit/type/Token.xml
```

> Mind that the file *types.txt* is must be located in *META-INF/org.apache.uima.fit* where *org.apache.uima.fit* is the name of a sub-directory inside *META-INF*. *We are not using the Java package notation here!*

To specify multiple TSDs, add additional lines to the file. If you have a large number of TSDs, you may prefer to add a pattern. Assume that we have a large number of TSDs under *org/apache/uima/fit/type*, we can use the following pattern which recursively scans the package org.apache.uima.fit.type and all sub-packages for XML files and tries to load them as TSDs.

```
classpath*:org/apache/uima/fit/type/**/*.xml
```

Try to design your packages structure in a way that TSDs and JCas wrapper classes generated from them are separate from the rest of your code.

If it is not possible or inconvenient to add the `types.txt` file, patterns can also be specified using the system property `org.apache.uima.fit.type.import_pattern`. Multiple patterns may be specified separated by semicolon:

```
-Dorg.apache.uima.fit.type.import_pattern=\
  classpath*:org/apache/uima/fit/type/**/*.xml
```

> The `\` in the example is used as a line-continuation indicator. It and all spaces following it should be ommitted.

Making index definitions and type priorities auto-detectable

Auto-detection also works for index definitions and type priority definitions. For index definitions, the respective file where to register the index definition XML files is *META-INF/org.apache.uima.fit/fsindexes.txt* and for type priorities, it is *META-INF/org.apache.uima.fit/typepriorities.txt*.

Using type auto-detection

The auto-detected type system can be obtained from the `TypeSystemDescriptionFactory`:

```
TypeSystemDescription tsd =
  TypeSystemDescriptionFactory.createTypeSystemDescription()
```

Popular factory methods also support auto-detection:

```
AnalysisEngine ae = createEngine(MyEngine.class);
```

Multiple META-INF/org.apache.uima.fit/types.txt files

uimaFIT supports multiple *types.txt* files in the classpath (e.g. in differnt JARs). The *types.txt* files are located via Spring using the classpath search pattern:

```
TYPE_MANIFEST_PATTERN = "classpath*:META-INF/org.apache.uima.fit/types.txt"
```

This resolves to a list URLs pointing to ALL *types.txt* files. The resolved URLs are unique and will point either to a specific point in the file system or into a specific JAR. These URLs can be handled by the standard Java URL loading mechanism. Example:

```
jar:/path/to/syntax-types.jar!/META-INF/org.apache.uima.fit/types.txt
jar:/path/to/token-types.jar!/META-INF/org.apache.uima.fit/types.txt
```

uimaFIT then reads all patters from all of these URLs and uses these to search the classpath again. The patterns now resolve to a list of URLs pointing to the individual type system XML descriptors. All of these URLs are collected in a set to avoid duplicate loading (for performance optimization - not strictly necessary because the UIMA type system merger can handle compatible duplicates). Then the descriptors are loaded into memory and merged using the standard UIMA type system merger (`CasCreationUtils.mergeTypeSystems()`). Example:

```
jar:/path/to/syntax-types.jar!/desc/types/Syntax.xml
jar:/path/to/token-types.jar!/org/foobar/typesystems/Tokens.xml
```

Voilá, the result is a type system covering all types could be found in the classpath.

It is recommended

1. to put type system descriptors into packages resembling a namespace you "own" and to use a package-scoped wildcard search

   ```
   classpath*:org/apache/uima/fit/type/**/*.xml`
   ```

2. or when putting descriptors into a "well-known" package like desc.type, that *types.txt* file should explicitly list all type system descriptors instead of using a wildcard search

   ```
   classpath*:desc/type/Token.xml
   classpath*:desc/type/Syntax.xml
   ```

Method 1 should be preferred. Both methods can be mixed.

Performance note and caching

Currently uimaFIT evaluates the patterns for TSDs once and caches the locations, but not the actual merged type system description. A rescan can be forced using `TypeSystemDescriptionFactory.forceTypeDescriptorsScan()`. This may change in future.

Potential problems

The mechanism works fine. However, there are specific issues with Java in general that one should be aware of.

m2eclipse fails to copy descriptors to target/classes

There seems to be a bug in some older versions of m2eclipse that causes resources not always to be copied to *target/classes*. If UIMA complains about type definitions missing at runtime, try to *clean/rebuild* your project and carefully check the m2eclipse console in the console view for error messages that might cause m2eclipse to abort.

Class version conflicts

A problem can occur if you end up having multiple incompatible versions of the same type system in the classpath. This is a general problem and not related to the auto-detection feature. It is the same as when you have incompatible version of a particular class (e.g. `JCas` wrapper or some third-party-library) in the classpath. The behavior of the Java Classloader is undefined in that case. The detection will do its best to try and load everything it can find, but the UIMA type system merger may barf or you may end up with undefined behavior at runtime because one of the class versions is used at random.

Classes and resources in the default package

It is bad practice to place classes into the default (unnamed) package. In fact it is not possible to import classes from the default package in another class. Similarly it is a bad idea to put resources at the root of the classpath. The Spring documentation on resources explains this in detail.

For this reason the *types.txt* resides in */META-INF/org.apache.uima.fit* and it is suggest that type system descriptors reside either in a proper package like */org/foobar/typesystems/XXX.xml* or in */desc/types/XXX.xml*.

Building an executable JAR

Building an executable JAR including uimaFIT components typically requires extra care. Per convention, uimaFIT expects certain information in specific locations on the classpath, e.g. the *types.txt* file that controls the automatic type system detection mechanism must reside at *META-INF/org.apache.uima.fit/types.txt*. It often occurs that a project has several dependencies, each supplying its own configuration files at these standard locations. However, this causes a problem with naive approaches to creating an executable *fat-jar* merging all dependencies into a single JAR file. Without extra care, the files supplied by the different dependencies overwrite each other during the packaging process and only one file *wins* in the end. As a consequence, the types configured in the other files cannot be detected at runtime. Such a native approach is taken, for example, by the Maven Assembly Plugin.

The Maven Shade Plugin provides a convenient alternative for the creation of executable fat-jars, as it provides a mechanism to concatenate the configuration files from different dependencies while creating the fat-jar. To use the Maven Shade Plugin with uimaFIT, use the following configuration section in your POM file and make sure to change the `mainClass` as required for your project:

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-shade-plugin</artifactId>
      <version>2.2</version>
      <executions>
        <execution>
          <phase>package</phase>
          <goals><goal>shade</goal></goals>
          <configuration>
            <transformers>
              <!-- Set the main class of the executable JAR -->
              <transformer
                implementation="org.apache.maven.plugins.shade.\
                                resource.ManifestResourceTransformer">
                <mainClass>org.apache.uima.fit.example.Main</mainClass>
              </transformer>
              <!-- Merge the uimaFIT configuration files -->
              <transformer
                implementation="org.apache.maven.plugins.shade.\
                                resource.AppendingTransformer">
                <resource>\
                  META-INF/org.apache.uima.fit/fsindexes.txt\
                </resource>
              </transformer>
              <transformer
                implementation="org.apache.maven.plugins.shade.\
                                resource.AppendingTransformer">
                <resource>\
                  META-INF/org.apache.uima.fit/types.txt\
```

```
                </resource>
              </transformer>
              <transformer
                implementation="org.apache.maven.plugins.shade.\
                                resource.AppendingTransformer">
                <resource>\
                  META-INF/org.apache.uima.fit/typepriorities.txt\
                </resource>
              </transformer>
              <!-- Merge CAS validation check registrations -->
              <transformer
                implementation="org.apache.maven.plugins.shade.\
                                resource.ServicesResourceTransformer"/>
            </transformers>
            <!--
              Prevent huge shaded artifacts from being deployed
              to a Maven repository (remove if not desired)
            -->
            <outputFile>\
              ${project.build.directory}/\
              ${artifactId}-${version}-standalone.jar\
            </outputFile>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Due to formatting constraints in the PDF version of this manual, the example above uses \ to indicate a line continuation. Remove these and join the lines when you copy/paste this example.

You might want to consider also merging additional files, such as LICENSE, NOTICE, or DEPENDENCY files, configuration files for the Java Service Locator API, or files used by other frameworks that uses similar conventions for configuration file locations. Check the documentation of the Maven Shade Plugin, as different kinds of configuration files require different specialized transformers.

uimaFIT Maven Plugin

uimaFIT dynamically generates UIMA component descriptions from annotations in the Java source code. The uimaFIT Maven plugin provides the ability to automatically create such annotations in already compiled classes and to automatically generate XML descriptors from the annotated classes.

enhance goal

The goal enhance allows automatically augmenting compiled classes with uimaFIT annotations. Information like vendor, copyright, or version can be obtained from the Maven POM. Additionally, descriptions for parameters and components can be generated from Javadoc comments. Existing annotations are not overwritten unless forced.

```
<plugin>
  <groupId>org.apache.uima</groupId>
  <artifactId>uimafit-maven-plugin</artifactId>
  <version></version> <!-- change to latest version -->
  <configuration>
    <!-- OPTIONAL -->
    <!-- Override component description in generated descriptors. -->
    <overrideComponentDescription>false</overrideComponentDescription>

    <!-- OPTIONAL -->
    <!-- Override version in generated descriptors. -->
    <overrideComponentVersion>false</overrideComponentVersion>

    <!-- OPTIONAL -->
    <!-- Override vendor in generated descriptors. -->
    <overrideComponentVendor>false</overrideComponentVendor>

    <!-- OPTIONAL -->
    <!-- Override copyright in generated descriptors. -->
    <overrideComponentCopyright>false</overrideComponentCopyright>

    <!-- OPTIONAL -->
    <!-- Version to use in generated descriptors. -->
    <componentVersion>${project.version}</componentVersion>

    <!-- OPTIONAL -->
    <!-- Vendor to use in generated descriptors. -->
    <componentVendor>Apache Foundation</componentVendor>

    <!-- OPTIONAL -->
    <!-- Copyright to use in generated descriptors. -->
    <componentCopyright>Apache Foundation 2013</componentCopyright>

    <!-- OPTIONAL -->
    <!-- Source file encoding. -->
```

```
    <encoding>${project.build.sourceEncoding}</encoding>

    <!-- OPTIONAL -->
    <!-- Generate a report of missing meta data in
         $project.build.directory/uimafit-missing-meta-data-report.txt -->
    <generateMissingMetaDataReport>true</generateMissingMetaDataReport>

    <!-- OPTIONAL -->
    <!-- Fail on missing meta data. This setting has no effect unless
         generateMissingMetaDataReport is enabled. -->
    <failOnMissingMetaData>false</failOnMissingMetaData>

    <!-- OPTIONAL -->
    <!-- Constant name prefixes used for parameters and external resources,
         e.g. "PARAM_". -->
    <parameterNameConstantPrefixes>
      <prefix>PARAM_<prefix/>
    </parameterNameConstantPrefixes>

    <!-- OPTIONAL -->
    <!-- Fail on missing meta data. This setting has no effect unless
         generateMissingMetaDataReport is enabled. -->
    <externalResourceNameConstantPrefixes>
      <prefix>KEY_<prefix/>
      <prefix>RES_<prefix/>
    </externalResourceNameConstantPrefixes>

    <!-- OPTIONAL -->
    <!-- Mode of adding type systems found on the classpath via the
         uimaFIT detection mechanism at compile time to the generated
         descriptor. By default, no type systems are added. -->
    <addTypeSystemDescriptions>NONE</addTypeSystemDescriptions>

  </configuration>
  <executions>
    <execution>
      <id>default</id>
      <phase>process-classes</phase>
      <goals>
        <goal>enhance</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

When generating descriptions for configuration parameters or external resources, the plugin supports a common practice of placing the Javadoc on a constant field instead of the parameter or external resource field. Per default, parameter name constants must be prefixed with `PARAM_` and external resource key constants must be prefixed with `RES_` ` or `KEY_`.

```
/**
 * Enable or disable my feature.
 */
public static final String PARAM_ENABLE_FEATURE = "enableFeature";
@ConfigurationParameter(name=PARAM_ENABLE_FEATURE)
private boolean enableFeature;

/**
 * My external resource.
 */
public static final String RES_MY_RESOURCE = "resource";
@ExternalResource(key=RES_MY_RESOURCE)
private MyResource resource;
```

By enabling `generateMissingMetaDataReport`, the build can be made to fail if meta data such as parameter descriptions are missing. A report about the missing data is generated in *uimafit-missing-meta-data-report.txt* in the project build directory.

generate goal

The generate goal generates XML component descriptors for UIMA components.

```
<plugin>
  <groupId>org.apache.uima</groupId>
  <artifactId>uimafit-maven-plugin</artifactId>
  <version></version> <!-- change to latest version -->
  <configuration>
    <!-- OPTIONAL -->
    <!-- Path where the generated resources are written. -->
    <outputDirectory>
      ${project.build.directory}/generated-sources/uimafit
    </outputDirectory>

    <!-- OPTIONAL -->
    <!-- Skip generation of META-INF/org.apache.uima.fit/components.txt -->
    <skipComponentsManifest>false</skipComponentsManifest>

    <!-- OPTIONAL -->
    <!-- Source file encoding. -->
    <encoding>${project.build.sourceEncoding}</encoding>
  </configuration>
  <executions>
    <execution>
      <id>default</id>
      <phase>process-classes</phase>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

In addition to the XML descriptors, a manifest file is written to `META-INF/org.apache.uima.fit/components.txt`. This file can be used to conveniently locate the XML descriptors, which are written in the packages next to the classes they describe.

```
classpath*:org/apache/uima/fit/examples/ExampleComponent.xml
```

It is recommended to use both, the enhance and the generate goal. Both goals should be specified in the same execution, first enhance, then generate:

```
<execution>
  <id>default</id>
  <phase>process-classes</phase>
  <goals>
    <goal>enhance</goal>
    <goal>generate</goal>
  </goals>
</execution>
```

Migration Guide

This section provides helpful information on incompatible changes between versions.

Version 3.0.x to 3.1.x

Changes to ExternalResourceFactory

The renaming of methods in the `ExternalResourceFactory` had unfortunately introduced another name clash between unrelated methods. To fix this clash, the following methods have been renamed from `bindResource` to `bindResourceOnce`:

- `void bindResource(ResourceCreationSpecifier aDesc, String aBindTo, ExternalResourceDescription aRes)` was **removed** and replaced by `void bindResourceOnce(ResourceCreationSpecifier aDesc, String aBindTo, ExternalResourceDescription aRes)`
- `void bindResource(ExternalResourceDescription aRes, String aBindTo, ExternalResourceDescription aNestedRes)` was deprecated and replaced by `void bindResourceOnce(ExternalResourceDescription aRes, String aBindTo, ExternalResourceDescription aNestedRes)`
- `void bindResource(ResourceManagerConfiguration aResMgrCfg, String aBindTo, ExternalResourceDescription aRes)` was deprecated and replaced by `void bindResourceOnce(ResourceManagerConfiguration aResMgrCfg, String aBindTo, ExternalResourceDescription aRes)`
- `void bindResource(ResourceCreationSpecifier aDesc, String aBindTo, String aRes)` was **removed** and replaced by `void bindResourceOnceWithoutNested(ResourceCreationSpecifier aDesc, String aBindTo, String aRes)`
- `void bindResource(ResourceManagerConfiguration aResMgrCfg, String aBindTo, String aRes)` was deprecated and replaced by `void bindResourceOnceWithoutNested(ResourceManagerConfiguration aResMgrCfg, String aBindTo, String aRes)`
- `void bindResource(ResourceSpecifier aDesc, String aKey, String aUrl)` was deprecated and replaced by `void bindResourceUsingUrl(ResourceSpecifier aDesc, String aKey, String aUrl)`

Version 2.x to 3.x

Legacy support module removed

The legacy support in uimaFIT 2.x was present allow being compatible with the pre-Apache uimaFIT versions which were based on UIMA 2.x. Since uimaFIT 3.x is not compatible with UIMA 2.x anyway, the legacy module was removed now.

Using List instead of Collection

The `CasUtil`, `JCasUtil` and `FSCollectionFactory` classes were adjusted to return results using `List` instead of the more general `Collection`. Often, lists are already used internally and then again wrapped into new lists in client code. This API change avoids this in the future.

Throwing specific exceptions instead of UIMAException

Several uimaFIT methods were throwing the generic `UIMAException`. These have been adjusted to

declare throwing several of the sub-types of `UIMAException` to be better able to handle specific causes of errors in client code.

CasUtil.selectSingle signature changed

Signature of `CasUtil.selectSingle` has been changed to return `AnnotationFS`. The original signature is available as `selectSingleFS`

Removal of deprecated methods

Various methods that were deprecated in uimaFIT 2.4.0 or earlier have been removed in this release. For details, please refer to the `api-change-report.html` file included in the release.

Changes to ExternalResourceFactory

Most methods in the `ExternalResourceFactory` have seen changes to their names and signature to avoid problematic ambiguities as well as to be shorter. In general, the `External` component of the method names was either removed or replaced. So most methods called `createExternalResourceDescription` are now called `createResourceDescription`. However, some have also been given a more specific name and/or a slightly different order of parameters. For example, this method

```
public static ExternalResourceDescription createExternalResourceDescription(
  Class<? extends SharedResourceObject> aInterface, String aUrl, Object... aParams)
```

was changed to

```
public static ExternalResourceDescription createSharedResourceDescription(
  String aUrl, Class< extends SharedResourceObject> aInterface, Object... aParams)
```

Changes to logging

UIMA v3 has is using SLF4J. As a consequence, the `ExtendedLogger` which uimaFIT had returned on calls to `getLogger()` has been removed and instead the regular UIMA v3 logger class is returned which offers methods quite compatible with what `ExtendedLogger` offered before. However, it is recommended that you go through all your logging calls and replace calls which use string concatenation to construct the logging message with corresponding calls using placeholders. For example, replace `getLogger().error("Cannot access " + filename, exception);` with `getLogger().error("Cannot access {}", filename, exception);`.

Version requirements

Depends on UIMA 3.0.2, Spring Framework 4.3.22 and Java 8.

Version 2.3.0 to 2.4.0

Version requirements

Depends on UIMA 2.10.2, Spring Framework 3.2.16 and Java 7.

Mind the updated version requirements. There should be no other potentially problematic changes in this upgrade.

Version 2.2.0 to 2.3.0

CasIOUtil deprecated

The functionality of the uimaFIT CasIOUtil class has been superseded by the core UIMA class CasIOUtils added in UIMA 2.9.0. The method signatures in the new class are not the same, but provide more functionality. CasIOUtil has been deprecated and documentation has been added which of the CasIOUtils methods should be used instead.

Version requirements

Depends on UIMA 2.9.1, Spring Framework 3.2.16 and Java 7.

Mind the updated version requirements. There should be no other potentially problematic changes in this upgrade.

Version 2.1.0 to 2.2.0

Version requirements

Depends on UIMA 2.8.1, Spring Framework 3.2.16 and Java 7.

Mind the updated version requirements. There should be no other potentially problematic changes in this upgrade.

Version 2.0.0 to 2.1.0

Version requirements

Depends on UIMA 2.6.0 and Java 6.

AnnotationFactory.createAnnotation()

No longer throws `UIMAExcption`. If this exception was cought, some IDEs may complain here after upgrading to uimaFIT 2.1.0.

Version 1.4.0 to 2.0.0

Version requirements

Depends on UIMA 2.4.2.

Backwards compatibility

Compatibility with legacy annotation is provided by the Legacy support module.

Change of Maven groupId and artifactId

The Maven group ID has changed from `org.uimafit` to `org.apache.uima`.

The artifact ID of the main uimaFIT artifact has been changed from `uimafit` to `uimafit-core`.

Change of package names

The base package has been renamed from `org.uimafit` to `org.apache.uima.fit`. A global search/replace on Java files with for lines starting with `import org.uimafit` and replacing that with

`import org.apache.uima.fit` should work.

@ConfigurationParameter

The default value for the mandatory attribute now is `true`. The default name of configuration parameters is now the name of the annotated field only. The classname is no longer prefixed. The method `ConfigurationParameterFactory.createConfigurationParameterName()` that was used to generate the prefixed name has been removed.

Type detection: META-INF/org.uimafit folder

The `META-INF/org.uimafit` was renamed to `META-INF/org.apache.uima.fit`.

JCasUtil

The deprecated `JCasUtil.iterate()` methods have been removed. `JCasUtil.select()` should be used instead.

AnalysisEngineFactory

All `createAggregateXXX` and `createPrimitiveXXX` methods have been renamed to `createEngineXXX`. The old names are deprecated and will be removed in future versions.

All `createAnalysisEngineXXX` methods have been renamed to `createEngineXXX`. The old names are deprecated and will be removed in future versions.

CollectionReaderFactory

All `createDescriptionXXX` methods have been renamed to `createReaderDescriptionXXX`. The old names are deprecated and will be removed in future versions.

All `createCollectionReaderXXX` methods have been renamed to `createReaderXXX`. The old names are deprecated and will be removed in future versions.

JCasIterable

`JCasIterable` now only accepts reader and engine descriptions (no instances) and no longer implements the `Iterator` interface. Instead, new `JCasIterator` has been added, which replaces `JCasIterable` in that respect.

CasDumpWriter

`org.uimafit.component.xwriter.CASDumpWriter` has been renamed to `org.apache.uima.fit.component.CasDumpWriter`.

CpePipeline

`CpePipeline` has been moved to a separate module with the artifact ID `uimafit-cpe` to reduce the dependencies incurred by the main uimaFIT artifact.

XWriter removed

The `XWriter` and associated file namers have been removed as they were much more complex then acutally needed. As an alternative, `CasIOUtil` has been introduced providing several convenience methods to read/write JCas/CAS data.

JCasFactory

Methods only loading JCas data have been removed from `JCasFactory`. The new methods in `CasIOUtil` can be used instead.