Apache UIMA: Components and Pipelines

PU Machine learning mit Java, Weka und UIMA

Nils Reiter,
nils.reiter@uni-koeln.de

January 19, 2021
Winter term 2020/21

Section 1

Looking Back

Exercise 9

`https://github.com/idh-cologne-machine-learning-mit-java/exercise-09`

- Create types in XML
- Add jcasgen maven plugin to pom.xml
- Create a few annotations

Exercise 9

`https://github.com/idh-cologne-machine-learning-mit-java/exercise-09`

- Create types in XML
- Add jcasgen maven plugin to pom.xml
- Create a few annotations

Typesystem auto-discovery

- Creata a file `META-INF/org.apache.uima.fit/types.txt`
- Add pattern(s) for file system descriptor(s)
- Doc: `https://uima.apache.org/d/uimafit-current/tools.uimafit.book.html#ugr.tools.uimafit.typesystem`

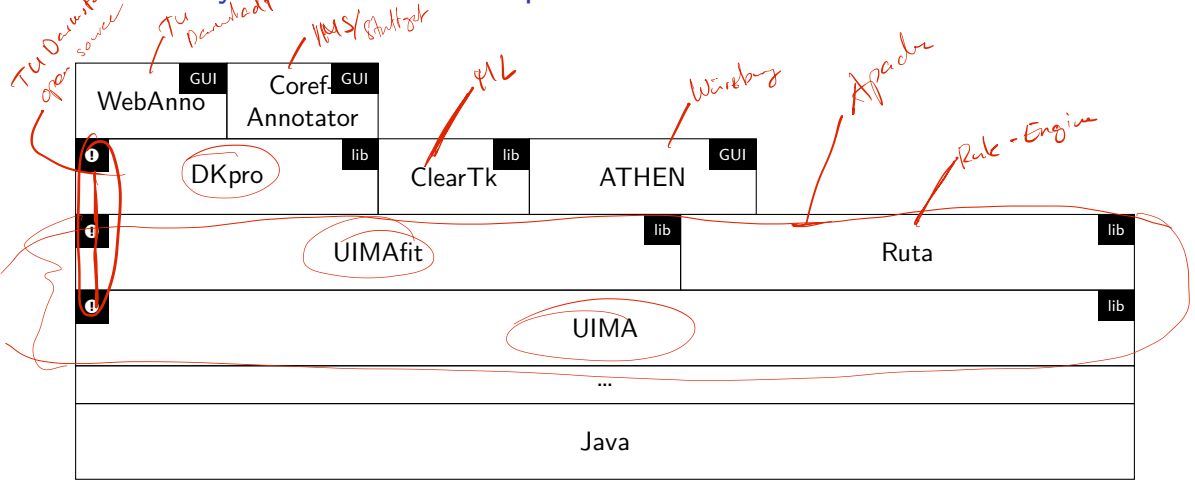
UIMA Data Structures

- Type: An annotation layer such as 'token'
- All annotations are stored as character positions in typed feature structures
- Types are defined in XML files
- JCasGen to convert from XML file to Java code
  - Can be integrated into maven
- Types have atomic or complex features and thus can encode entire graphs

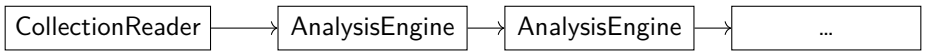
Section 2

UIMA Processing Pipelines

UIMA Library and Tool Landscape



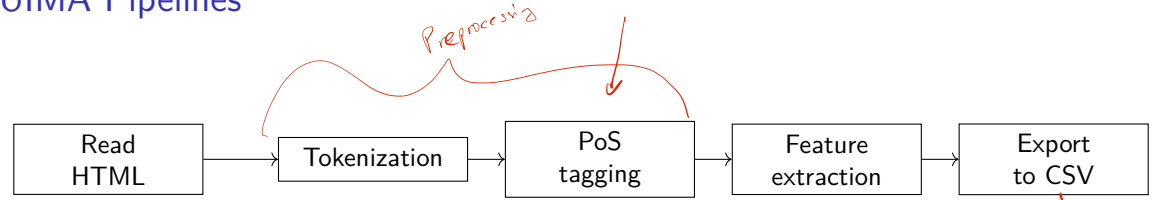
UIMA Pipelines



UIMA Pipelines

- Each component (= AnalysisEngine) represented by a class
  - Object created when pipeline is run
- Analysis engines
  - Inherit from `org.apache.uima.fit.component.JCasAnnotator_ImplBase` or `org.apache.uima.analysis_component.JCasAnnotator_ImplBase`
  - Implement a single method `void process(JCas jcas)`

UIMA Pipelines



- Each component (= AnalysisEngine) represented by a class
  - Object created when pipeline is run
- Analysis engines
  - Inherit from `org.apache.uima.fit.component.JCasAnnotator_ImplBase` or `org.apache.uima.analysis_component.JCasAnnotator_ImplBase`
  - Implement a single method `void process(JCas jcas)`

Common Analysis Structure (CAS / JCas)

- Transferred from one component to the next
- Contains
  - The source document
  - All feature structures
- Allows access and creation of new feature structures / annotations

Common Analysis Structure (CAS / JCas)

- Transferred from one component to the next
- Contains
  - The source document
  - All feature structures
- Allows access and creation of new feature structures / annotations
- CAS: low-level / C implementation
- JCas: More Java-friendly interface (which we will use)

Inside a Component

Listing 1: Creating a new annotation

```
1  // create a new object of type T
2  T a = new T(jcas);
3  // set begin position
4  a.setBegin(5);
5  // set end position
6  a.setEnd(15);
7  // set other feature values
8  a.setFeature("FeatureValue");
9  // add the annotation to the index
10 // (don't forget this!)
11 a.addToIndexes();
```

Listing 2: Iterating over ex. annotations

```
1  // select all tokens
2  SelectFSs<Token> tokens =
3    jcas.select(Token.class);
4
5  // SelectFSs<T> inherits from
6  // Iterable<T>, allowing:
7  for (Token t : tokens) {
8    // do something with each token
9    String tokenSurface =
10     t.getCoveredText();
11   // ...
12 }
```

Inside a Component

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them
- This improved a lot with UIMA 3

```
1 SelectFSs<Token> selector = jcas.select(Token.class);
2 selector.asList(); // a list of all feature structures
3 selector.count(); // number of feature structures
4 selector.get(index); // get nth token
5 selector.coveredBy(anotherAnnotation); // feature structures covered by another FS
6 selector.following(position); // first token after position
7 selector.allMatch(predicate); // get tokens for which predicate is fulfilled
```

Inside a Component

Select Interface

```
1 // iterate over sentences
2 for (Sentence sentence : jcas.select(Sentence.class)) {
3   // iterate over tokens in a sentence
4   for (Token token : jcas.select(Token.class).coveredBy(sentence)) {
5     // do stuff with each token
6   }
7 }
```

Inside a Component

Select Interface

```java
// iterate over sentences
for (Sentence sentence : jcas.select(Sentence.class)) {
  // iterate over tokens in a sentence
  for (Token token : jcas.select(Token.class).coveredBy(sentence)) {
    // do stuff with each token
  }
}
```

```java
// iterate over sentences that start with "T"
for (Sentence sentence : jcas.select(Sentence.class)
    .allMatch(s -> s.getCoveredText().startsWith("T"))) {
  // iterate over tokens in sentence with id larger than 15
  for (Token token : jcas.select(Token.class).coveredBy(sentence)
      .allMatch(t -> t.getId > 15)) {
    // do stuff with token
  }
}
```

Lambda Expressions in Java

```java
// long form
Predicate<Token> predicate = new Predicate<Token>() {
  public boolean test(Token t) {
    return t.getId() > 15;
  }
};

// short form ("syntactic sugar")
Predicate<Token> predicate = t -> t.getId() > 15;
```

- `java.util.function` package (since 1.8)
- Functional interface: Classes/interfaces with a single method
  - Predicate: `T` → `boolean` (`public boolean test(T)`)
  - BiPredicate: `T`, `S` → `boolean` (`public boolean test(T, S)`)
  - Function: `T` → `S`
  - …

Resources and Arguments

- Many components need resources and/or parameters
- Define fields as usual
- Components may implement a method `public void initialize(UimaContext)`
  - $\simeq$ constructor, but frameworks handles construction as needed
- Arguments need to be passed as atomic values (or read from file)

Resources and Arguments

- Many components need resources and/or parameters
- Define fields as usual
- Components may implement a method `public void initialize(UimaContext)`
  - $\simeq$ constructor, but frameworks handles construction as needed
- Arguments need to be passed as atomic values (or read from file)

```
1  public class MyComponent extends JCasAnnotator_ImplBase {
2
3    @Override
4    public void initialize(final UimaContext context)
5      throws ResourceInitializationException {
6
7      // call super class method
8      super.initialize(context);
9
10   }
11
12   @Override
13   public void process(JCas jcas) throws AnalysisEngineProcessException { }
14 }
```

Resources and Arguments

- Declare parameters as class fields
- Use `@ConfigurationParameter()` annotation to mark as configuration parameter
  - Javadoc:

    `https://javadoc.io/static/org.apache.uima/uimafit-core/3.1.0/org/apache/uima/fit/descriptor/ConfigurationParameter.html`
  - Arguments
    - `name = ...` Define a name for the parameter
    - `defaultValue = ...` Default value (as a string array)
    - `description = ...` Human-readable description
    - `mandatory = ...` Defines whether it can be omitted, `boolean` value

Resources and Arguments

- Declare parameters as class fields
- Use `@ConfigurationParameter()` annotation to mark as configuration parameter
  - Javadoc:

    https://javadoc.io/static/org.apache.uima/uimafit-core/3.1.0/org/apache/uima/fit/descriptor/ConfigurationParameter.html
  - Arguments
    - `name = ...` Define a name for the parameter
    - `defaultValue = ...` Default value (as a string array)
    - `description = ...` Human-readable description
    - `mandatory = ...` Defines whether it can be omitted, `boolean` value
- Do I need to define `initialize(UimaContext)`?
  - Not for purely filling given argument values into fields variables
  - Only if there is something to be done with the arguments
    - E.g., loading a file

Resources and Arguments: Best Practice

```java
public class MyComponent extends JCasAnnotator_ImplBase {
  // Define a static public variable that is used instead of the name
  public static final String PARAM_WINDOW_SIZE = "window_size";

  @ConfigurationParameter(name = PARAM_WINDOW_SIZE, defaultValue="10", mandatory=false)
  int windowSize = 10;

  @Override
  public void process(JCas jcas) throws AnalysisEngineProcessException {
    for (Token token : jcas.select(Token.class)) {
      // get all tokens
      List<Token> nextTokens = jcas.select(Token.class)
        // ... that follow this token
        .following(token)
        // ... and only the first windowSize, as list
        .limit(windowSize).asList();
      // do something
    }
  }
}
```

UIMA Pipeline

- UIMA pipelines consist of a collection reader and multiple analysis engines
- To create a pipeline, it's easiest to work with descriptions
- Collection reader

```
CollectionReaderDescription crd = CollectionReaderFactory.createReaderDescription(...);
```

- Analysis engine

```
AnalysisEngineDescription engine1 = AnalysisEngineFactory.createEngineDescription(...);
```

UIMA Pipeline

- UIMA pipelines consist of a collection reader and multiple analysis engines
- To create a pipeline, it's easiest to work with descriptions
- Collection reader

```
CollectionReaderDescription crd = CollectionReaderFactory.createReaderDescription(...);
```

- Analysis engine

```
AnalysisEngineDescription engine1 = AnalysisEngineFactory.createEngineDescription(...);
```

- Both take class object as argument

```
AnalysisEngineFactory.createEngineDescription(MyComponent.class);
```

UIMA Pipeline

- UIMA pipelines consist of a collection reader and multiple analysis engines
- To create a pipeline, it's easiest to work with descriptions
- Collection reader

```
CollectionReaderDescription crd = CollectionReaderFactory.createReaderDescription(...);
```

- Analysis engine

```
AnalysisEngineDescription engine1 = AnalysisEngineFactory.createEngineDescription(...);
```

- Both take class object as argument

```
AnalysisEngineFactory.createEngineDescription(MyComponent.class);
```

- Arguments can be supplied as name/value pairs

```
AnalysisEngineFactory.createEngineDescription(MyComponent.class, "window size", 10);
```

MyComponent.PARAM_WINDOW_SIZE

UIMA Pipeline

- UIMA pipelines consist of a collection reader and multiple analysis engines
- To create a pipeline, it's easiest to work with descriptions
- Collection reader

```
CollectionReaderDescription crd = CollectionReaderFactory.createReaderDescription(...);
```

- Analysis engine

```
AnalysisEngineDescription engine1 = AnalysisEngineFactory.createEngineDescription(...);
```

- Both take class object as argument

```
AnalysisEngineFactory.createEngineDescription(MyComponent.class);
```

- Arguments can be supplied as name/value pairs

```
AnalysisEngineFactory.createEngineDescription(MyComponent.class, "window size", 10);
```

- Putting together a pipeline:

```
SimplePipeline.runPipeline(crd, engine1, engine2, ...);
```

demo

Section 3

Next Exercise

Exercise 10

https://github.com/idh-cologne-machine-learning-mit-java/exercise-10