Apache UIMA (& Maven)

PU Tools, Ressourcen, Infrastruktur

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

January 14, 2021
(Winter term 2020/21)

Section 1

Recap

Exercise 8

`https://github.com/idh-cologne-tools-ressourcen-infra/exercise-08`

Summary: NLP Data Structures

- NLP data structures are complex, because language is
- Annotations can be rooted in tokens or character positions
- Many linguistic structures can be represented as trees
  - Trees are a special kind of graph

Section 2

Dependency Management with Maven

Introduction

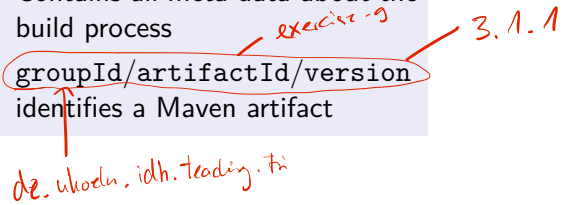
- Maven: A build tool
  - Build process: Convert source code to binary code
  - More than pure compilation: Libraries (in correct versions), ressources, packages, documentation, testing, ...
- Build process often automated (continuous integration, CI)
  - Maven can also run on GitHub – "Actions"
- Documentation: `https://maven.apache.org/guides/`

How to use Maven

Two core ingredients

pom.xml

- »Project Object Model«
- XML-file
- Contains all meta data about the build process
- groupId/artifactId/version identifies a Maven artifact



How to use Maven

Two core ingredients

pom.xml

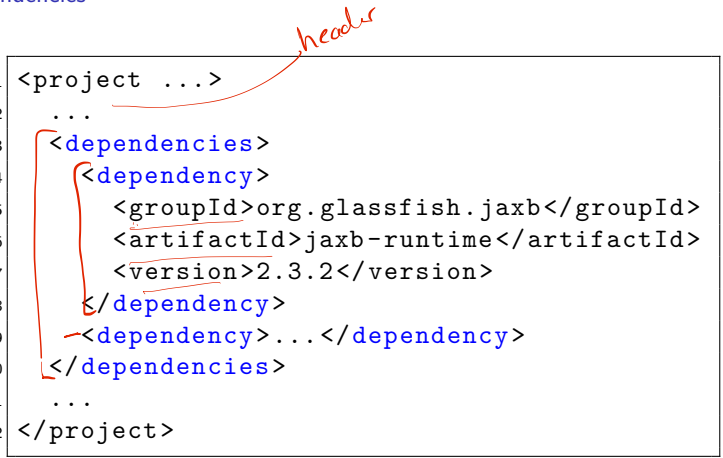
- »Project Object Model«
- XML-file
- Contains all meta data about the build process
- groupId/artifactId/version identifies a Maven artifact

Standard Directory layout

- pom.xml
- src
  - main
    - java
    - resources
  - test
    - java
    - resources
- target

pom.xml Sections

Dependencies



pom.xml Sections

Dependencies

```
<project ...>
  ...
  <dependencies>
    <dependency>
      <groupId>org.glassfish.jaxb</groupId>
      <artifactId>jaxb-runtime</artifactId>
      <version>2.3.2</version>
    </dependency>
    <dependency>...</dependency>
  </dependencies>
  ...
</project>
```

- Specify each dependency as triple `groupId`, `artifactId`, and `version`
- All dependencies in maven universe!

pom.xml

Properties

```
1  <project>
2    <properties>
3      <version.javafx>14.0.2.1</version.javafx>
4    </properties>
5    ...
6    <dependencies><dependency>
7        <groupId>org.openjfx</groupId>
8          <artifactId>javafx-controls</artifactId>
9          <version>${version.javafx}</version>
10       </dependency><dependency>
11         <groupId>org.openjfx</groupId>
12         <artifactId>javafx-swing</artifactId>
13         <version>${version.javafx}</version>
14       </dependency></dependencies>
15 </project>
```

- Properties can be used throughout the POM
- Can be defined in other ways as well: Profiles, inheritance, default, system OS, ...
  - `http://maven.apache.org/ref/3.6.3/maven-model-builder/#Model_Interpolation`

Summary

- Maven to organise your dependencies
- Works across computers
  - because dependencies are downloaded automatically from Maven central
- We can use Maven from now on to handle dependencies
- Many more Maven things not covered here: `http://maven.apache.org`

Section 3

Apache UIMA

UIMA

- UIMA: Unstructured Information Management Architecture
- `https://uima.apache.org`
- Apache Project: Open source

Apache License

UIMA

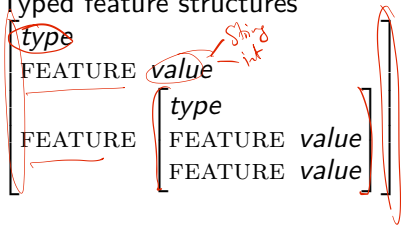
- UIMA: Unstructured Information Management Architecture
- `https://uima.apache.org`
- Apache Project: Open source
- Framework for
  - Data structures, with efficient access
  - Processing, in particular pipeline architectures
  - Modularization

UIMA

- UIMA: Unstructured Information Management Architecture
- `https://uima.apache.org`
- Apache Project: Open source
- Framework for
  - Data structures, with efficient access
  - Processing, in particular pipeline architectures
  - Modularization
- Eco system
  - uimaFIT: Facilitates use of UIMA
  - DKPro: Prepackaged NLP components

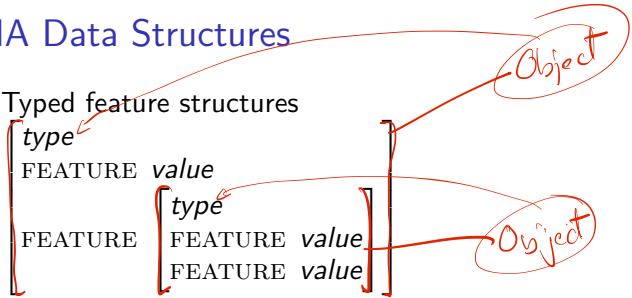
UIMA Data Structures

- Typed feature structures

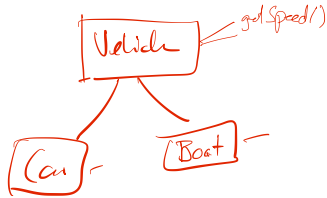


UIMA Data Structures

- Typed feature structures



- Each FS represented as an object of class *type*
  - E.g., the class `Token` represents a FS of type ›Token‹
- Types define which features instances of the type have
- Regular class hierarchy for inheritance



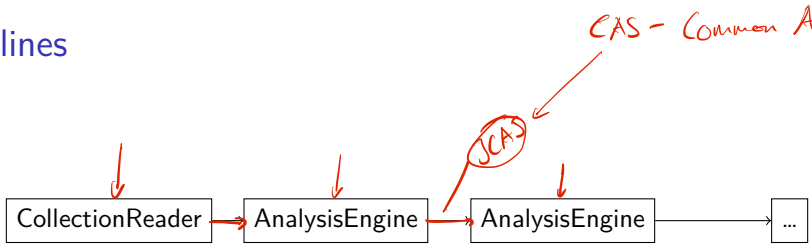
UIMA Data Structures

- Typed feature structures

$$\begin{bmatrix} \textit{type} \\ \text{FEATURE} & \textit{value} \\ \text{FEATURE} & \begin{bmatrix} \textit{type} \\ \text{FEATURE} & \textit{value} \\ \text{FEATURE} & \textit{value} \end{bmatrix} \end{bmatrix}$$

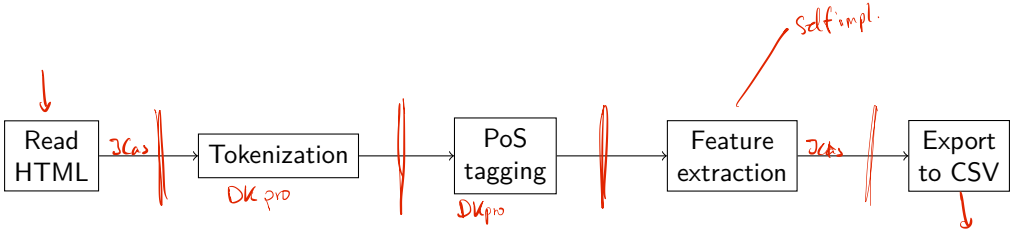
- Each FS represented as an object of class *type*
  - E.g., the class `Token` represents a FS of type ›Token‹
- Types define which features instances of the type have
- Regular class hierarchy for inheritance
- UIMA defines basic building blocks for NLP types
- DKPro defines actual NLP types for UIMA
  - We will use these

UIMA Pipelines



- Each component (= AnalysisEngine) represented by a class
- Inherits from `org.apache.uima.fit.component.JCasAnnotator_ImplBase` or `org.apache.uima.analysis_component.JCasAnnotator_ImplBase`
- Implements a single method `void process(JCas jcas)`

UIMA Pipelines



- Each component (= AnalysisEngine) represented by a class
- Inherits from `org.apache.uima.fit.component.JCasAnnotator_ImplBase` or `org.apache.uima.analysis_component.JCasAnnotator_ImplBase`
- Implements a single method `void process(JCas jcas)`

Inside a Component

Listing: Creating a new annotation

```
1  // create a new annotation object
2  Annotation a = new Annotation(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: Iterating over existing annotations

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

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
1 SelectFSs<Token> selector = jcas.select(Token.class);
2 selector.asList(); // a list of all feature structures
```

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
1 SelectFSs<Token> selector = jcas.select(Token.class);
2 selector.asList(); // a list of all feature structures
3 selector.count(); // number of feature structures
```

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
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.coveredBy(anotherAnnotation); // feature structures
5                                       // covered by another
```

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
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.coveredBy(anotherAnnotation); // feature structures
5                                       // covered by another
6 selector.following(position); // first token after position
```

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
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.coveredBy(anotherAnnotation); // feature structures
5                                       // covered by another
6 selector.following(position); // first token after position
7 selector.get(index); // get nth token
```

Select Interface

- UIMA indexes all feature structures, such that we can efficiently access them

```
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.coveredBy(anotherAnnotation); // feature structures
5                                       // covered by another
6 selector.following(position); // first token after position
7 selector.get(index); // get nth token
8 selector.allMatch(predicate); // get tokens for which predicate
9                               // is fulfilled
```

demo

Section 4

Appendix

References I