ISMLA Session 3 - UIMA

Björn Rudzewitz

Tübingen University

November 6, 2017

Plan

1. Analysis Engine
   - Configuration Parameters
   - initialize/process
2. Aggregate Analysis Engine
3. Collection Reader

UIMA Workflow

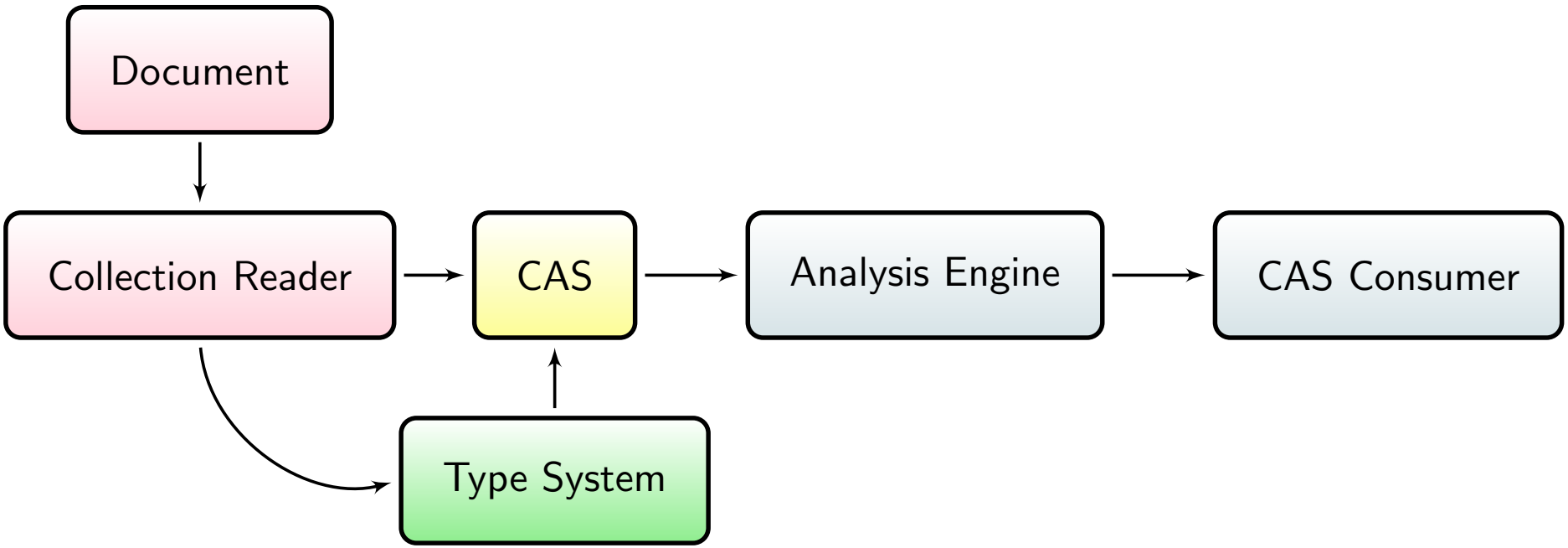

Figure: UIMA Workflow

Analysis Engine

- Analysis Engine (AE)
- adds analyses (meta data) of a specific kind to a CAS
- can reuse all previous analyses made by other analysis engines (defined in descriptor which types it relies on)
- modular component, performing exactly one specific analysis step

Analysis Engine

AEs as data-driven components:

- type of action performed depends on the input the component receives
- based on document text and potential previous annotations
- [Gotz and Suhre, 2004, p. 479]: AEs as *"producers of data for downstream components and as consumers of data from upstream components"*

Analysis Enginge Setup

Add two files:

1. Java
   File $\rightarrow$ *New* $\rightarrow$ *Class*
   extends
   `org.apache.uima.analysis_component.JCasAnnotator_ImplBase`
2. XML
   New $\rightarrow$ *Other* ... $\rightarrow$ *UIMA* $\rightarrow$ *Analysis Engine Descriptor File*

Analysis Engine Setup

Configuration

- link Java file in descriptor
- import type system
- set input and output capabilities
- add configuration parameters

Configuration Parameters

- possibility to pass parameters to a UIMA component, e.g. language-specific model file or output directory
- access external resources via `InputStream`
- define via ComponentDescriptorEditor
- access values:

```
(String)getConfigParameterValue("outputDir")
```

```
aContext.getResourceAsStream((String)
aContext.getConfigParameterValue(RESOURCE_KEY))
```

initialize/process

- UIMA Java components typically implement/override `initialize` and `process` method
- `initialize`: only executed once, useful for loading models etc.
- `process`: executed for every CAS in the process

Exercise

Discussion of homework exercise:

1. set up a Java Maven project with UIMA dependencies
2. define types and generate Java code form it
3. wrap the OpenNLP sentence detector in an analysis engine and test it with the DocumentAnalyzer

Aggregate Analysis Engine

- last time: primitive whitespace* tokenizer
- (one) problem: role of punctuation[1]
  - part of words (e.g. abbreviations)
  - sentence delimiters

→ tokenizer would need to know sentence boundaries

[1] cf. also discussion forum

Aggregate Analysis Engine

- conceptual solution:
  1. annotate sentence boundaries
  2. use sentence boundaries in tokenization (tokenize on sentence basis)

Aggregate Analysis Engine

- conceptual solution:
  1. annotate sentence boundaries
  2. use sentence boundaries in tokenization (tokenize on sentence basis)
- technical solution:
  - write a tokenization AE and a sentence detection AE
  - chain the sentence and token AE together

$\Rightarrow$ Aggregate Analysis Engine (AAE)

Aggregate Analysis Engine

- AAE as a wrapper around primitive analysis engines
- flow controller detemines the order of execution of AEs
- Example: part-of-speech tagging a document usually requires sentence detection and tokenization first

$\rightarrow$ AEs can be recursive programming structures

Analysis Engine Setup

Aggregate AE

- create new AE descriptor
- in first tab, check option *Aggregate*
- in the tab *Aggregate* add AEs and determine the flow of the AEs
- set other parameters (I/O Capabilities, . . .) like for primitive AEs

Accessing existing information

- access JCas from CAS:

```
JCas jcas = cas.getJCas();
```

- access original text:

```
String docText = jcas.getDocumentText()
```

- access previously annotated information:

```
Iterator sentIter =
arg0.getAnnotationIndex(Sentence.type).iterator();
```

iterator default behavior: extract information based on start indices

Demonstration

- creating an AE that builds on previous information: naive NP detector
- creating a AAE descriptor
- running an AE in DocumentAnalyzer

Exercise

- writing analysis engine wrapping OpenNLP tokenizer
- tokenizer should receive sentences as input
- writing AAE chaining sentence detector and tokenizer
- test via DocumentAnalyzer

→ see handout

Collection Reader

- initializes for each document a Common Analysis Structure
- observations can be extracted from various sources, e.g.
  - a CSV file,
  - a directory with files,
  - a data base, ...
- initializes an iterator over observations, then while processing selects the next element
- pipeline automatically stops when iterator has no next element
- essentially a collection reader assumes a specific file format and creates an empty container for subsequent analyses

Collection Reader

- extends `org.apache.uima.collection.CollectionReader_ImplBase`
- functions:
  - getNext: sets language and document text
  - hasNext
  - getProgress
  - close
  - (initialize): prepares component for iterating

Demonstration

- writing a collection reader to recursively read all files

Exercise

- write a collection reader that reads in every line of a file as a document (see handout)

Xiaobin Chen and Detmar Meurers. Ctap: A web-based tool supporting automatic complexity analysis. *CL4LC 2016*, page 113, 2016.

David Ferrucci and Adam Lally. Uima: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004.

David Ferrucci, Eric Brown, Jennifer Chu-Carroll, James Fan, David Gondek, Aditya A Kalyanpur, Adam Lally, J William Murdock, Eric Nyberg, John Prager, et al. Building watson: An overview of the deepqa project. *AI magazine*, 31(3):59–79, 2010.

Stephen C Gates, Wilfried Teiken, and Keh-Shin F Cheng. Taxonomies by the numbers: building high-performance taxonomies. In *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 568–577. ACM, 2005.

Thilo Gotz and Oliver Suhre. Design and implementation of the uima common analysis system. *IBM Systems Journal*, 43(3): 476–489, 2004.

Guergana K Savova, James J Masanz, Philip V Ogren, Jiaping Zheng, Sunghwan Sohn, Karin C Kipper-Schuler, and Christopher G Chute. Mayo clinical text analysis and knowledge extraction system (ctakes): architecture, component evaluation and applications. *Journal of the American Medical Informatics Association*, 17(5):507–513, 2010.