DIPF
Bildungsforschung
und Bildungsinformation

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Introduction to UIMA

Dr. Judith Eckle-Kohler, Richard Eckart de Castilho, Roland Kluge, Dr. Torsten Zesch

UBIQUITOUS
KNOWLEDGE
PROCESSING

Part 1: UIMA

UIMA – Unstructured Information Management Architecture

Major goal:

- transform unstructured information to structured information

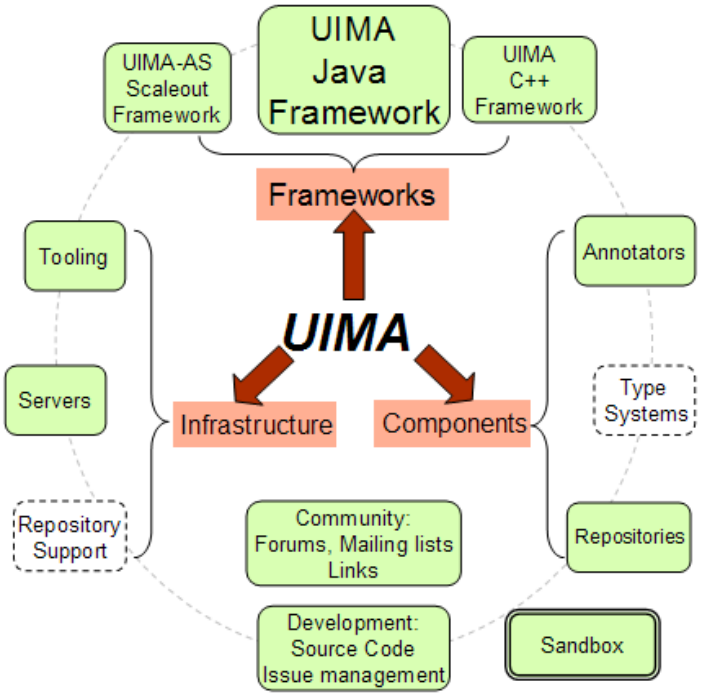
… in order to discover knowledge that is relevant to an end user

- Component-based architecture for analysis of unstructured content like text, video, audio

- How it works: think of UIMA components as machines in an assembly line

A Short History of UIMA

- Unstructured Information Management Architecture
- Originally developed at IBM – today an Apache project
- Used in commercial as well as educational contexts
  - LanguageWare, Watson (IBM)
  - uimaFIT (TU Darmstadt, University of Colorado)
  - DKPro Core (!) (TU Darmstadt)
  - many more...
- Java and C++ implementations



Unstructured Information Management Architecture
An Apache Incubator Project

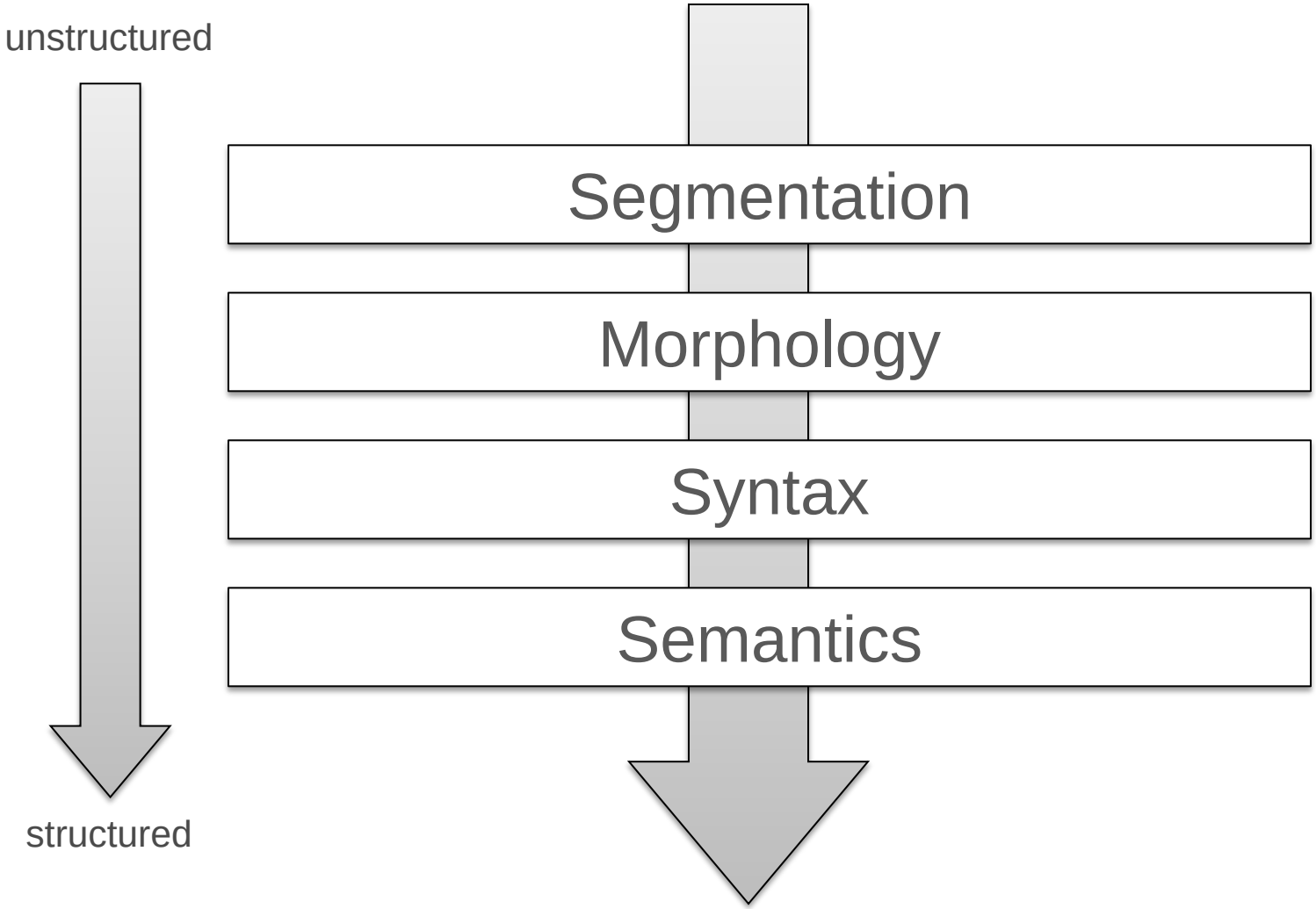
The Apache Software Foundation
http://www.apache.org/

Learning to read is difficult for computers …

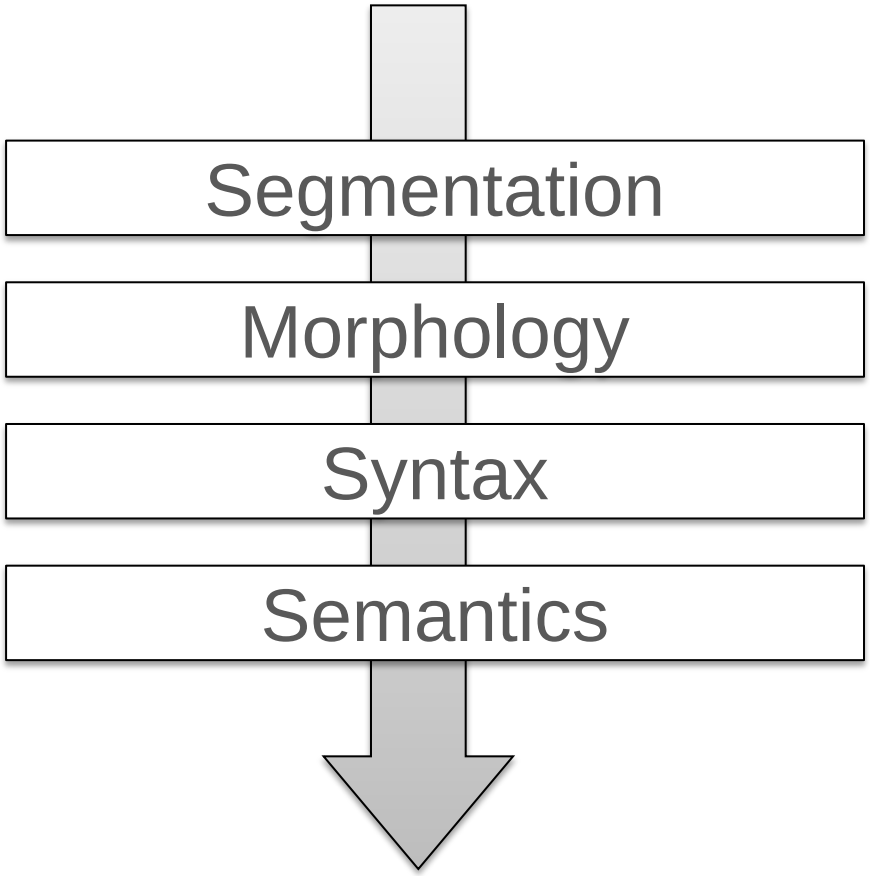
$◆Ⱶ◆◩⏏%Ⱶ $%$◆▷⊷◯▷↕ ♦▷%¥ ⊥↕%б↕ -- $▷.⁝∇б▷ ↕♦▷] #%‡ ↕%Ⱶ** ▷ ⁑ ▷‡ $▷.⁝∇б▷ ↕♦▷] #%‡ $%$$Ⱶ▷ % ⊥◆◩Ⱶ▷ □∇б¥, $%$◆▷⊥ ◆‡ $◆Ⱶ◆◩⏏%Ⱶ ♦∇⏏⊥▷♦∇Ⱶ¥⊥ =%◯▷↕ % ♦▷%¥ ⊥↕%б↕ ◆‡ Ⱶ◆.⁝▷, %##∇б¥◆◩ ↕∇ % ↕▷%= ∇.⁝ ⊥#◆▷‡↕◆⊥↕⊥ ◆‡ ◆↕%Ⱶ]. б%↕♦▷б ↕♦%‡ #∇‡.⁝⏏⊥◆◩ $%$◆▷⊥, ♦▷%б◆◩ =∇б▷ ↕♦%‡ ∇‡▷ Ⱶ%◩⏏%◯◆ ⁑ ▷⊥ ‡▷□$∇б‡⊥ % =▷‡↕%Ⱶ $∇∇⊥↕, %##∇б¥◆◩ ↕∇ ↕♦▷ ‡▷□ ⊥↕⏏¥], □♦◆#♦ ↕▷⊥↕▷¥ ⊥▷ ⁑ ▷‡-=∇‡↕♦-∇Ⱶ¥ ◆‡.⁝%‡↕⊥ "◆‡ =%‡] ▷⏏б∇◦▷%‡ #∇⏏‡↕б◆▷⊥, ◦%б▷‡↕⊥ %б▷ □%б] ∇.◯◆ ⁑ ◆◩ % $◆Ⱶ◆◩⏏%Ⱶ ▷¥⏏#%↕◆∇‡ ↕∇ ↕♦▷◆б **◆¥⊥ %‡¥ ↕б] ↕∇ ⊥◦▷%** ∇‡Ⱶ] ∇‡▷ Ⱶ%◩⏏%◯▷," ⊥%◆¥ ⊥↕⏏¥] %⏏↕♦∇б ↗%#□⏏▷⊥ =▷♦Ⱶ▷б ∇.⁝ ↕♦▷ Ⱶ%◩⏏%◯▷, #Ⓥ‡◆↕◆∇‡, %‡¥ ¥▷ ⁑ ▷Ⱶ∇◦=▷‡↕ Ⱶ%$ %↕ ↕♦▷ ◆‡↕▷б‡%↕◆∇‡%Ⱶ ⊥#♦∇∇Ⱶ .⁝∇б %¥ ⁑ %‡#▷¥ ⊥↕⏏¥◆▷⊥ ◆‡ ↕б◆▷⊥↕▷, ◆↕%Ⱶ]. "↕♦▷] %б▷ %.⁝б%◆¥ [↕♦▷◆б #♦◆Ⱶ¥б▷‡] =◯♦↕ ⊥⏏.⁝.⁝▷б □♦▷‡ ↕♦▷◯▷↕ ↕∇ ⊥#♦∇∇Ⱶ %‡¥ ⊥∇ ∇‡," =▷♦Ⱶ▷б ⊥%◆¥. "$▷#%⏏⊥▷ ∇.⁝ ∇⏏б б▷⊥⏏Ⱶ↕⊥, ◆ ¥∇⏏$↕ ↕♦%↕ ⁑ ▷б] =⏏#♦.“

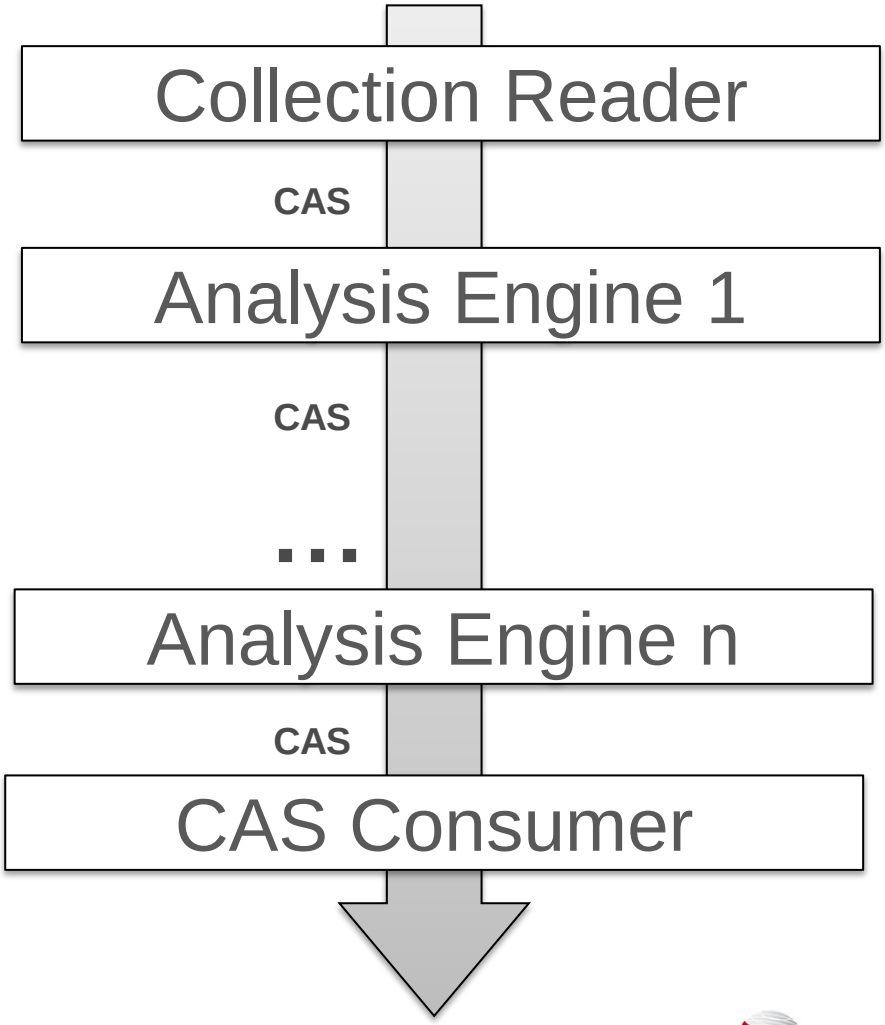
Unstructured text

Analysis Levels in Text Processing

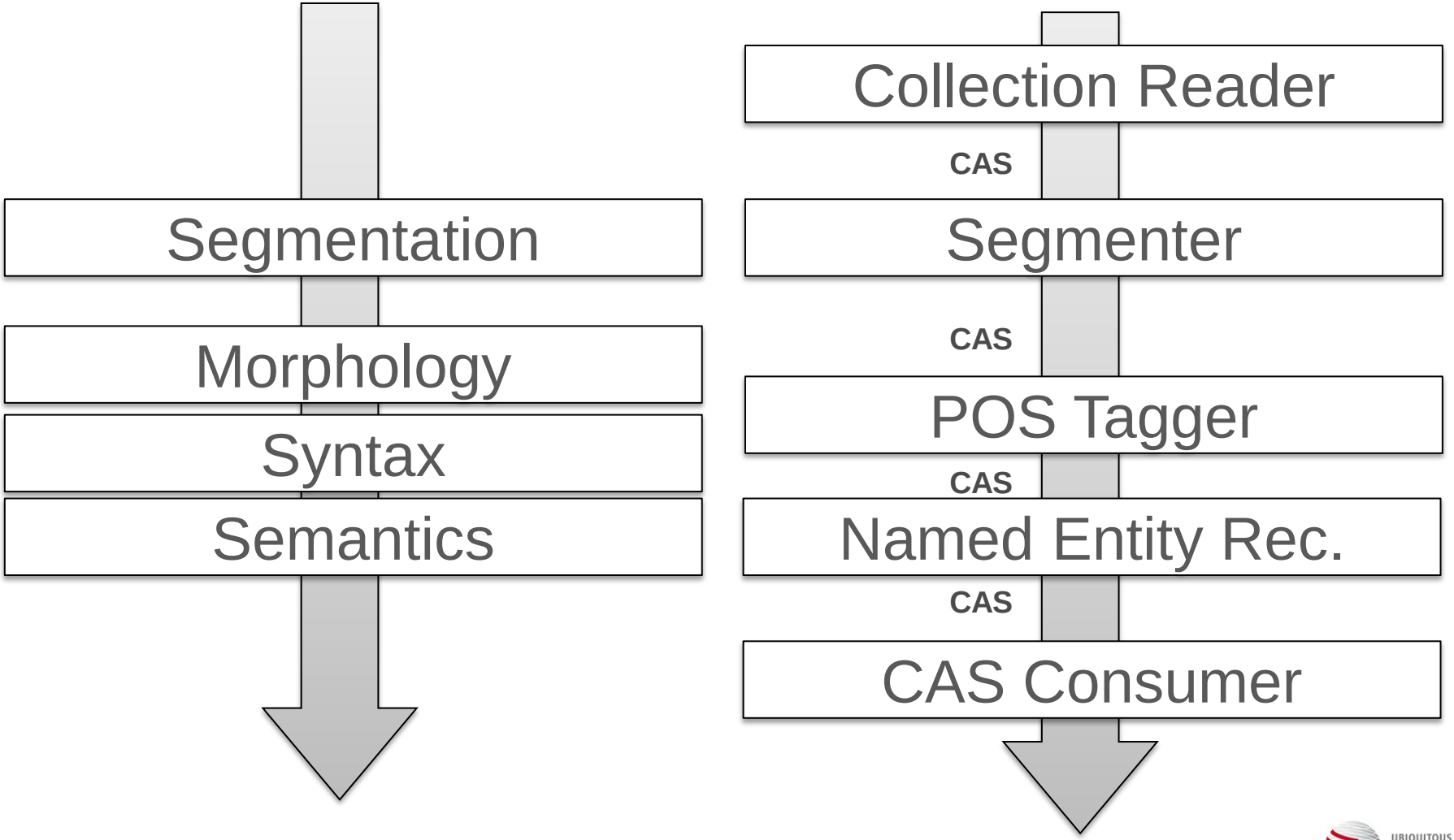


UIMA Pipeline Example





UIMA Example Pipeline for Text Processing



UIMA Concepts I

Pipeline Stages/Components:

- Collection Reader: start of pipeline, abstraction of input files
- Analysis Engine: performs analysis (tokenization, segmentation, etc.)
- CAS Consumer: e.g. for writing out results (XML, text, console)

UIMA Concepts II

Data Structures

- Common Analysis System (CAS): „data transfer object“

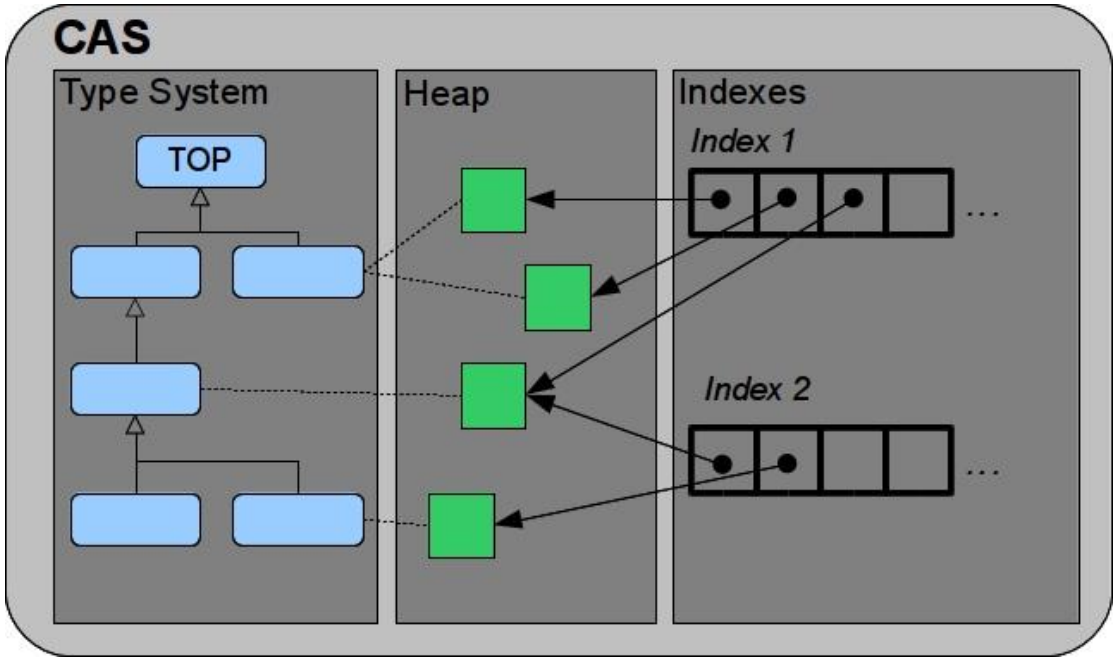
- Type System: representation of annotations, contracted interface between components
- Indexes: accessing annotations

- Views: e.g. raw HTML view, cleaned text view

- Subject-of-Analysis (SofA): e.g., document text of the current view

Common Analysis System (CAS)

- High-density data structure, functions like an in-memory database
- Provides access to
  - primary data (document/artifact under consideration)
  - secondary data (meta-data/annotations)



Type System

A UIMA type system specifies the type of data that can be manipulated by annotator components.

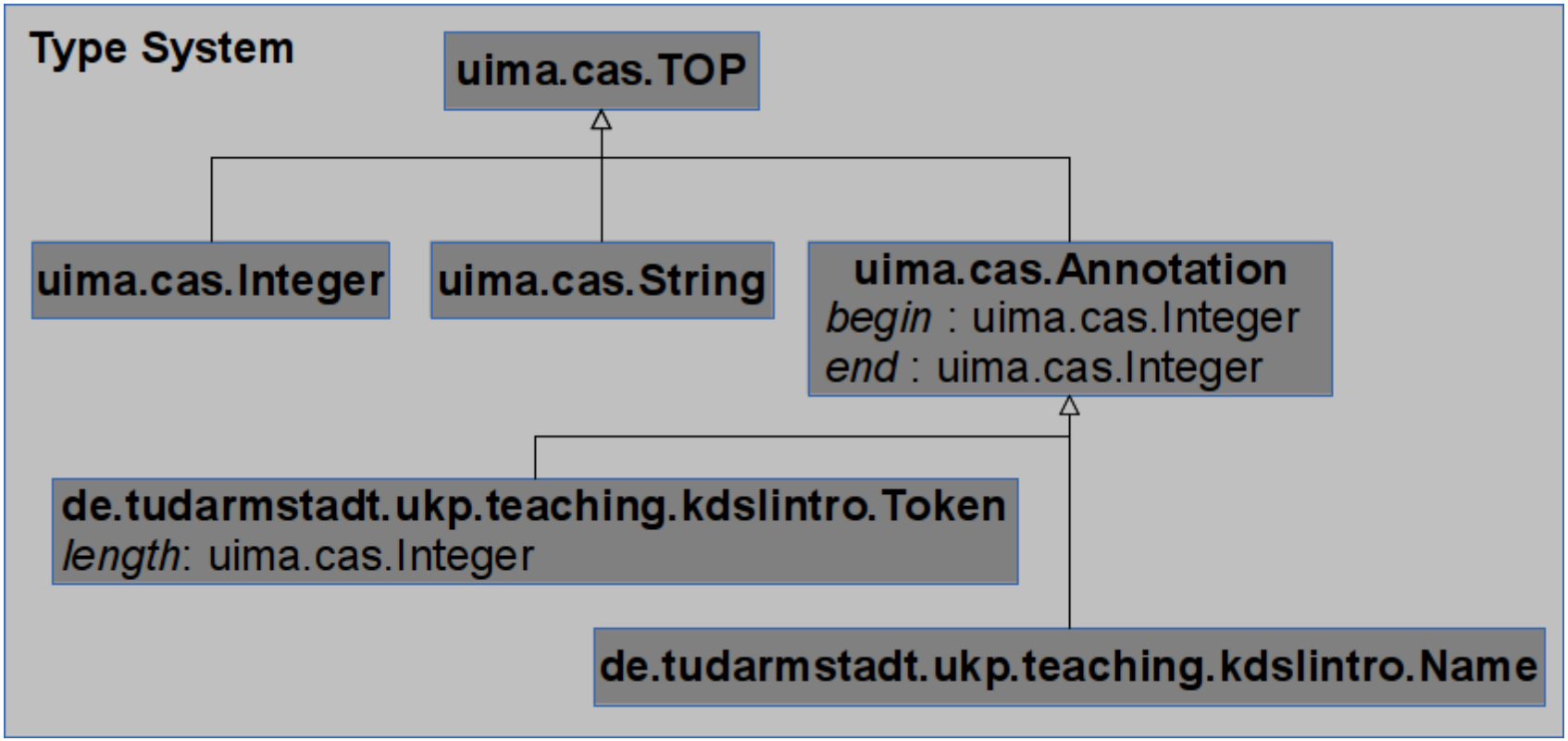
- UIMA provides an “object-oriented” type system

- A type system defines two kinds of objects:
  - Types (Type -> class)
  - Features (Feature -> class member, Feature Structure -> instance)

Type System

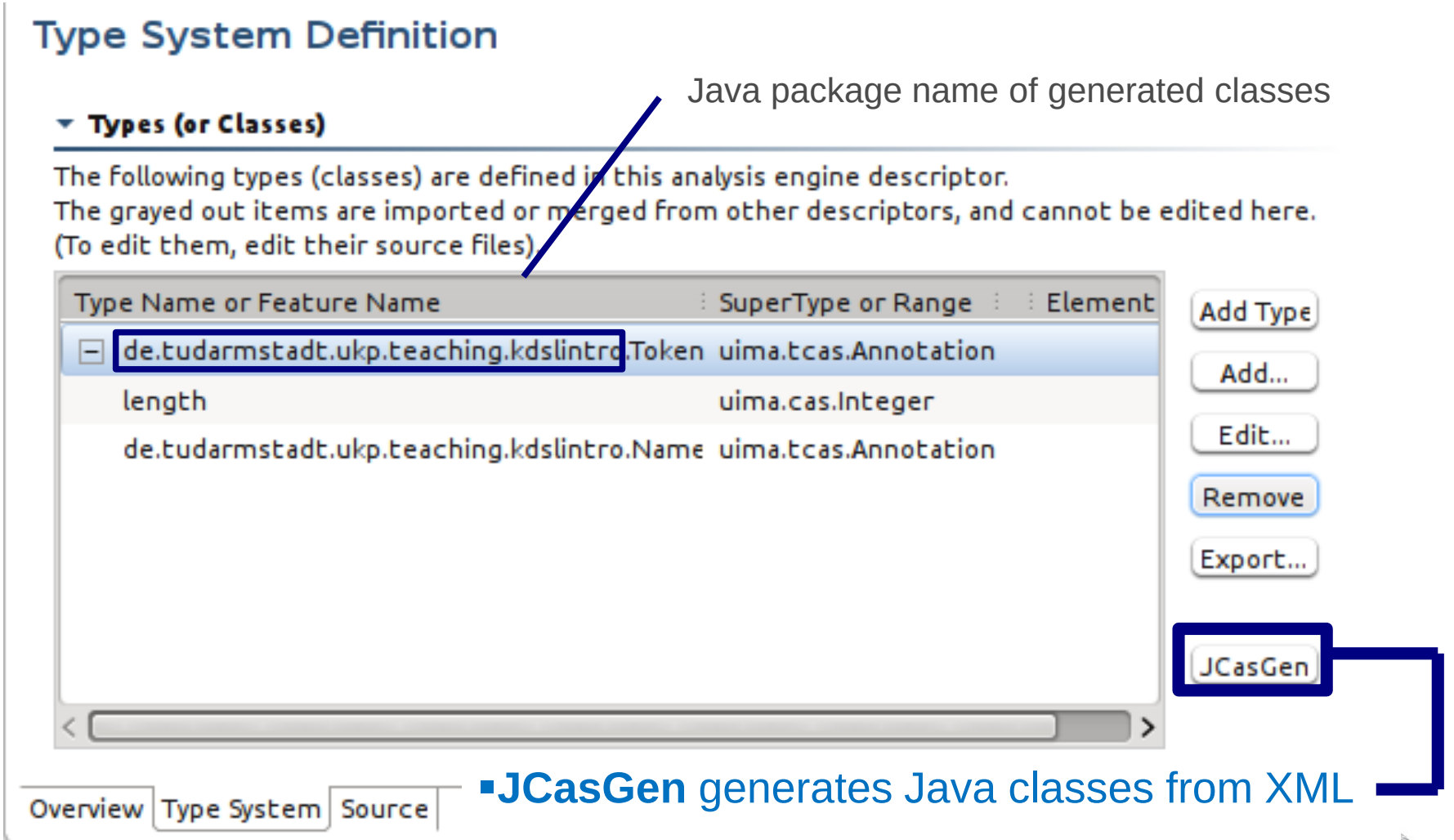
- Single inheritance
- Sub-type polymorphism
- **Primitive types**: integer, float, boolean, String
- **Built-in complex types**: arrays, lists, Annotation
- Type system is part of **communication contract** between components

Example Type System



Type System Editor (Eclipse)

File: src/main/resources/desc/types/TypeSystem.xml



- **JCasGen** generates Java classes from XML

Java + CAS = JCas

- **JCas** maps CAS types into the Java type system
- **JCasGen** generates Java classes from the XML type system descriptor
  - *Token.java* – feature structure wrapper with getters and setters
  - *Token_type.java* – type wrapper (cf. Java 'Class' class)
- Do not edit these automatically generated Java classes manually!
- JCas wrappers **cannot** be used **stand-alone**
- XML type system descriptors still needed to initialize the underlying CAS

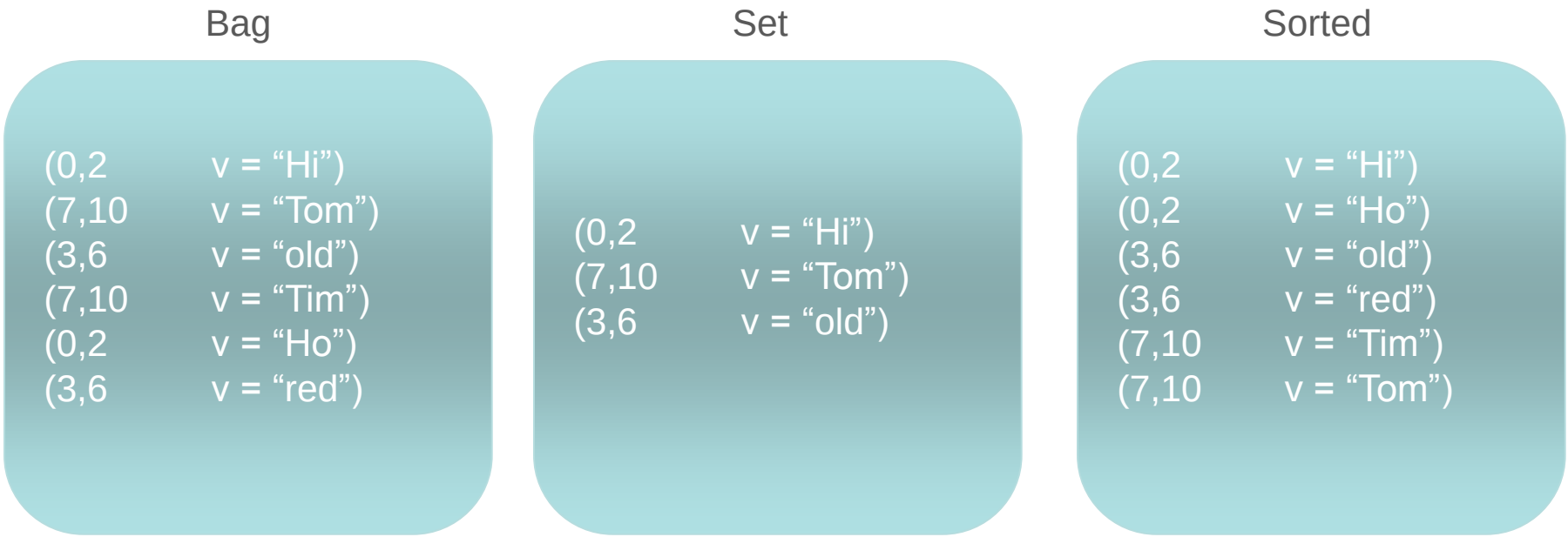
Java Code Example:

```
JCas jCas = …;
Token token = new Token(jCas); // new allocates memory in the CAS!
token.addToIndexes(); // never forget this!
```

Indexes

- Recap: feature structures (FS) are stored on the heap
- Components cannot directly access FS, but only via indexes
- Feature structures **only accessible when added to an index**
- Feature structures can only be removed from index, never from CAS
- Properties of an index (excerpt):
  - *Type* to be indexed (index implicitly contains all sub-types)
  - *Kind*: bag, set, sorted (see next slide)
- Example: Built-in *Annotation* index
  - *Type*: Annotation
  - *Kind*: sorted, begin (standard), end (reverse)

Figure: Indexes



Indexes – all you need to know

- normally nobody needs to define indexes
- indexes are the only way for UIMA annotators to access annotations in the CAS
- it is necessary to **generate** these indexes, they are not provided automatically within UIMA

Views and SofAs – Conceptual

- **CAS** represents the analysis of a **single artifact (a document)**
- Each view contains a copy of the artifact,
  - referred to as the **Subject of Analysis (SofA)** – the primary data associated with a view (as returned by *getDocumentText()*),
  - and a set of indexes, the **FSIndexRepository,** that UIMA annotators use to access data in the CAS
- **Usual setting: View** is **one representation** of the artifact, e.g.
  - *Translation scenario*: original text, translated text
  - *Transformation scenario*: original text, transformed text
  - *Multi-modal scenario*: video frames, close-captions

Figure: Views and SofAs

SofA unaware component receives **default view** in *process* (*CAS*) when calling *getDocumentText()*

SofA aware component receives base CAS in *process*(*CAS*) needs to call *getView(viewName)*

Views and SofAs – Use Cases

CAS represents the analysis of a **single artifact (a document)**

- This is true in most applications

- **But**: Views can also be used to compare different artifacts (mostly pairs)
  - this requires a customized reader that reads in several artifacts into a single CAS,
  - and then stores each artifact in a separate view.

Part 2: uimaFIT

uimaFIT

- „add-on“ for UIMA **simplifying** typical **development tasks**

- for instance:
  - consistency with XML descriptor files
  - component configuration
  - (shared) resource management

- CAS/JCas access
- Component base classes
- @ConfigurationParameter annotation
- Factories

http://code.google.com/p/uimafit/

Steps of Implementing a Collection Reader

- subclass the uimaFIT component **JCasCollectionReader_ImplBase**
- Methods to be implemented:
  - **void getNext(JCas)**: store next document in the given output parameter
  - **boolean hasNext()**
  - **Progress[] getProgress()**: returns progress information
    - common implementation:
      ```
      new Progress[]{new ProgressImpl(remaining, total, Progress.ENTITIES)}
      ```
  - **void close()**: free resources
- Optional:
  - **void initialize(UimaContext):** may be used for opening files etc.

Steps of Implementing an Annotator

- subclass the uimaFIT component **JCasAnnotator_ImplBase**
- **void process(JCas)** performs the actual analysis
- Optional:
  - **void initialize(UimaContext):** may be used for opening files etc.
    - always call `super.initialize(context);`

```
public class NameAnnotator
    extends JCasAnnotator_ImplBase
{
    @Override
    public void process(JCas aJCas)
        throws AnalysisEngineProcessException {}
}
```

Steps of Implementing a CAS Consumer I

uimaFIT does not distinguish between CAS Consumer and Annotation Engine (as UIMA does):

- Both are initialized almost identically in uimaFIT
  - See implementation of *JCasConsumer_ImplBase* and *JCasAnnotator_ImplBase*

- The only difference between the initialization of a CAS Consumer and an Analysis Engine in uimaFIT is the ability of multi-threading
  - multi-threading is allowed for Analysis Engines by default, but it is not allowed for CAS Consumers

Steps of Implementing a CAS Consumer II

- subclass the uimaFIT component **JCasConsumer_ImplBase**
  - **void process(JCas)** extracts data from the CAS
- Optional:
  - **void initialize(UimaContext):** may be used for opening files etc.
    - always call `super.initialize(context);`
  - **void collectionProcessComplete():** is called when all CASes have been processed

```
public class AnnotationFrequencyConsumer
    extends JCasConsumer_ImplBase
{
    @Override
    public void process(JCas aJCas)
        throws AnalysisEngineProcessException{}
}
```

Create and Configure Your Component - @ConfigurationParameter

- uimaFIT provides us with a powerful **annotation-based configuration mechanism**
  - declare property as field (any primitive + classes with String-only constructor, Locale, Pattern, …)
  - add annotation @ConfigurationParameter
- **Attributes** (excerpt)**:**
  - **name**: referred to when configuring the component
  - **mandatory**: fail if missing/null
  - **defaultValue**: string

```
public static final String PARAM_DICTIONARY_FILE =
        "dictionaryFile";
@ConfigurationParameter(name = PARAM_DICTIONARY_FILE,
        mandatory = true)
private File dictionaryFile;
```

Create and Configure Your Component: @ConfigurationParameter – Best Practices

- Best Practice: use the field name as value of the string constant
- Example:
  - `public static final String PARAM_DICTIONARY_FILE = "dictionaryFile";`
  - `private File dictionaryFile;`

```
public static final String PARAM_DICTIONARY_FILE =
        "dictionaryFile";
@ConfigurationParameter(name = PARAM_DICTIONARY_FILE,
        mandatory = true)
private File dictionaryFile;
```

Create and Configure Your Component: @ConfigurationParameter – Best Practices

- Best Practice: attributes `mandatory, default`
  - If possible, set the `default` value, even if `mandatory=true`
  - Why: components are not able to handle a `null` value

- In the example, it is not possible to set a default value, as a meaningful value can only be set by a user

```
public static final String PARAM_DICTIONARY_FILE =
        "dictionaryFile";
@ConfigurationParameter(name = PARAM_DICTIONARY_FILE,
        mandatory = true)
private File dictionaryFile;
```

Create and Configure Your Component II

uimaFIT instantiates the components for you!

- **AnalysisEngineFactory.createPrimitiveDescription**
  - for analysis engines and CAS consumers
- **CollectionReaderFactory.createDescription**
  - for collection readers

```
CollectionReaderDescription reader = createDescription(
    TextReader.class,
    TextReader.PARAM_PATH, "src/test/resources/txt",
    TextReader.PARAM_PATTERNS, new String[] {"[+]*.txt"},
    TextReader.PARAM_LANGUAGE, "de");

AnalysisEngineDescription segmenter =
       createPrimitiveDescription(BreakIteratorSegmenter.class);
AnalysisEngineDescription consumer =
       createPrimitiveDescription(FrequencyConsumer.class);

SimplePipeline.runPipeline(reader, segmenter, consumer);
```

Using and Exploring Annotations

The utility class **org.uimafit.util.JCasUtil** provides convenient access to the annotations.

- `selectCovered` is the preferred way to retrieve annotations from the CAS

```
int i = 0;
for(Sentence sentence : JCasUtil.select(jCas, Sentence.class))
{
        System.out.println("Tokens of sentence " + (i++) + ":");
        for(Token token :
              JCasUtil.selectCovered(jCas, Token.class, sentence))
    {
        System.out.println(token.getCoveredText());
    }
}
```

Manually Creating JCas Instances

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

jCas.setDocumentText("some text");
jCas.setDocumentLanguage("en"); // IMPORTANT!

AnalysisEngineDescription tokenizer =
       createPrimitiveDescription(MyTokenizer.class);

runPipeline(jCas, tokenizer);

for(Token token : JCasUtil.select(jCas, Token.class)){
    System.out.println(token.getCoveredText());
}
```

uimaFIT Best Practices – *descriptions*

If available, use the Factory methods of uimaFIT to create **component descriptions**, e.g., *CollectionReaderFactory.createDescription*

- Why: a reader created this way can be used multiple times in different pipelines (see example pipeline in `de.tudarmstadt.kdsl.teaching.dkprocore.intro`)

uimaFIT Best Practices – *CollectionReader*

- Always set the parameter PARAM_LANGUAGE, this is required by many Analysis Engines
- Always set the parameters PARAM_PATH and PARAM_PATTERNS in combination
  - PARAM_PATTERNS is specified by ANT-style patterns, i.e., you have to set a pattern that specifies the files to be **included**, see http://ant.apache.org/manual/dirtasks.html#patterns
- Good to know: CollectionReader instances can also process compressed files (zip format)

Type System Auto-Discovery

uimaFIT needs to know the XML type system descriptor's location at runtime, see http://code.google.com/p/uimafit/wiki/TypeDescriptorDetection

Either

- create file *src/resources/META-INF/types.txt*
- add path to your XML file in the following manner:

```
classpath*:desc/types/*.xml
```

- (uimaFIT will take into account any XML file in *desc/types*)

Or

- add VM option to Launch Configuration:

```
-Dorg.apache.uima.fit.type.import_pattern=classpath*:desc/types/*.xml
```

For more information see Chapter 7 of the uimaFit Guide at http://code.google.com/p/uimafit

Can you answer these questions?

- What is UIMA?
- What is uimaFIT?
- What is the benefit of using uimaFIT component descriptions?
- What is the basic structure of UIMA-based projects?
- What is an Annotation?
- How do you create a new annotation type?
- How do you add annotations to a JCas?
- Why do you need to call `addToIndexes()`?
- When do you need different views of an artifact?
- How to implement a Collection Reader? (Annotator, CAS Consumer)

Exercises (I)

- Take a look at uimaFIT's *JCasUtil*

UIMA Basics

- Explore the project
- Pipeline, CR, AE, Consumer
- Typesystem Descriptor:
  - *src/test/resources/desc/types/TypeSystem.xml*
  - *src/test/resources/META-INF/org.uimafit/types.txt*
- Micro-corpus
  - *src/test/resources/txt*
- (optional) explore the structure of the multimodule project
  - *pom.xml*
  - aggregator's *pom.xml*

Exercises (II)

UIMA Exploring

- **Objective:** Write your own pipeline and analyze the results
- Get the *uimaexploring.exercise* project
- Write your own *NameAnnotator* which looks up each token in a name list (*src/main/resources/dictionaries/names.txt*)
- read in dictionary in *initialize(UimaContext)* method
- Write a *NamePrintConsumer* which nicely prints out your name annotations; output how many name annotations you have assigned (for each document/all documents in total)
- Hint: *collectionProcessComplete()* may be helpful
- Serialize your CASes to XML and use the GUI tools to examine their contents

References

- T. Götz, O. Suhre, 2004: **Design and implementation of the UIMA Common Analysis System**, IBM Systems Journal Vol 43 #3, p. 476-489

- http://uima.apache.org/doc-uima-why.html

- http://uimafit.googlecode.com/svn/tags/uimafit-parent-1.4.0/apidocs/index.html