UIMA for Natural Language Analysis

Xiaobin Chen Björn Rudzewitz

Tübingen University

May 17, 2017

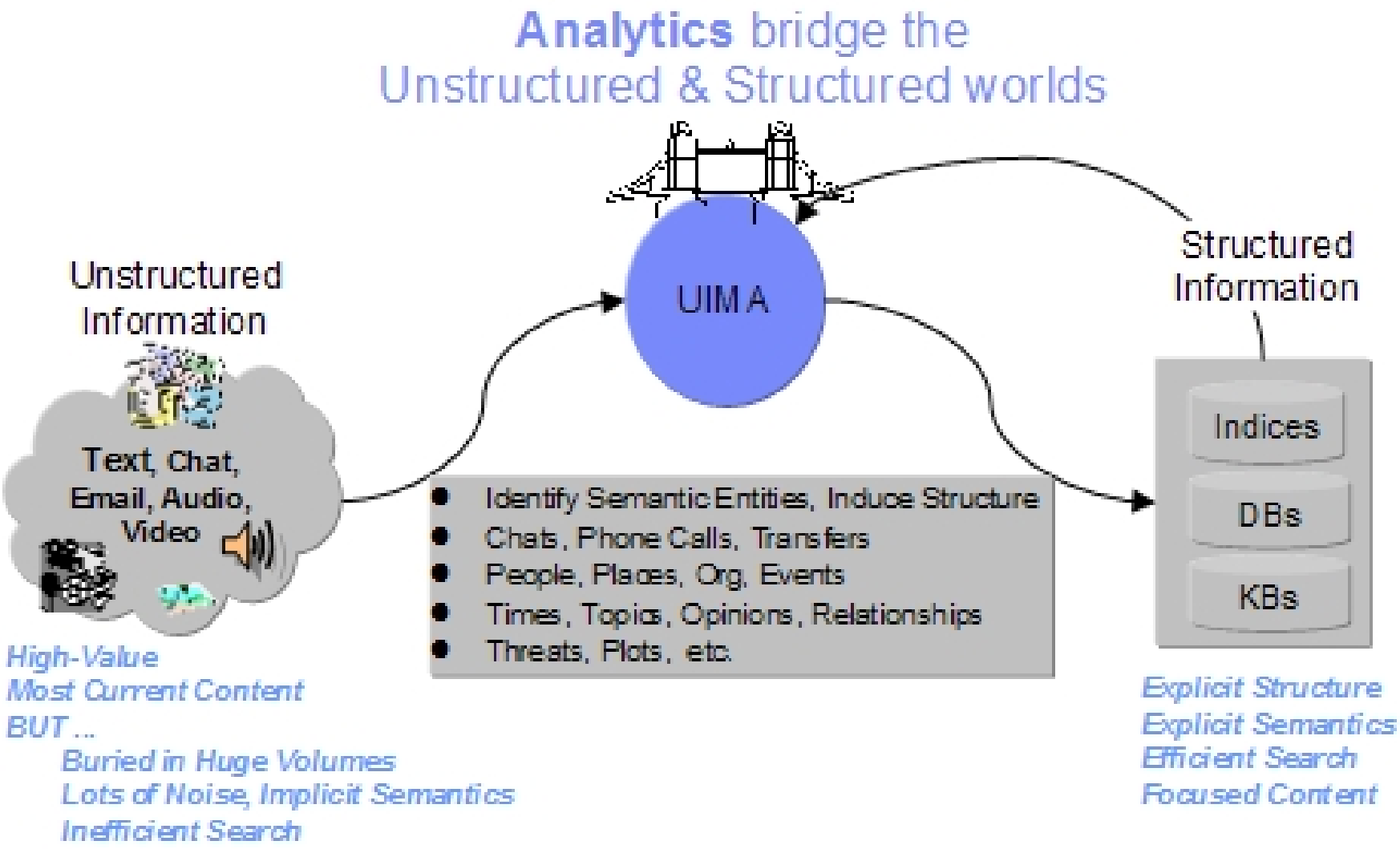
What is UIMA

Unstructured Information Management

- software systems that analyze large volumes of unstructured information in order to discover knowledge that is relevant to an end user.
- unstructured information: natural language documents, images, video, audio. . .
- discovered knowledge: entities, such as persons, places, organizations; or relations, such as works-for or located-at

Application in NLP: annotates natural language with meta-linguistic information

The Bridge


Analytics bridge the
Unstructured & Structured worlds
Unstructured Information
Text, Chat, Email, Audio, Video
UIMA
Structured Information
Indices
DBs
KBs
Identify Semantic Entities, Induce Structure
Chats, Phone Calls, Transfers
People, Places, Org, Events
Times, Topics, Opinions, Relationships
Threats, Plots, etc.
High-Value
Most Current Content
BUT ...
Buried in Huge Volumes
Lots of Noise, Implicit Semantics
Inefficient Search
Explicit Structure
Explicit Semantics
Efficient Search
Focused Content

Capabilities

- decomposing analysis into components, e.g. language identification, sentence detection, tokenization, POS tagging, parsing, entity detection, etc.
- managing analytical components and the data flow between them.
- supporting language independent development, e.g. C++, Java
- scaling out to cluster of networked nodes to handle very large volumes of analysis needs

What UIMA is NOT?

UIMA is not

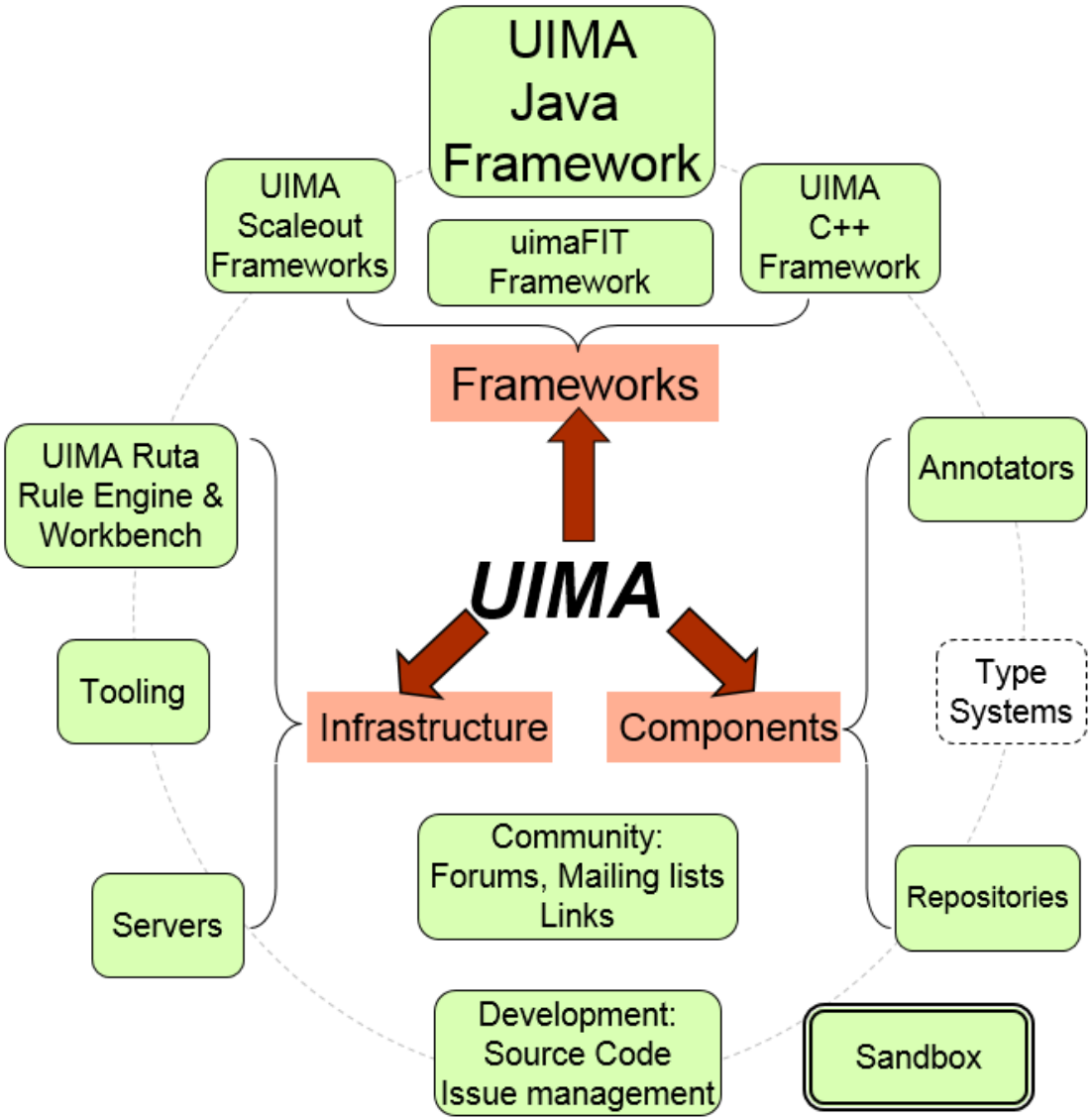
- a programming language
- an NLP toolkit
- database

It

- is an "empty" framework by itself.
- enables world-wide, diverse community to develop inter-operable, often complex analytic components, and allow them to be combined and run together, with framework supplied scaled-out and remoting as needed.

The specification
`http://www.oasis-open.org/committees/download.php/28492/uima-spec-wd-05.pdf`

The UIMA Framework



Required setup

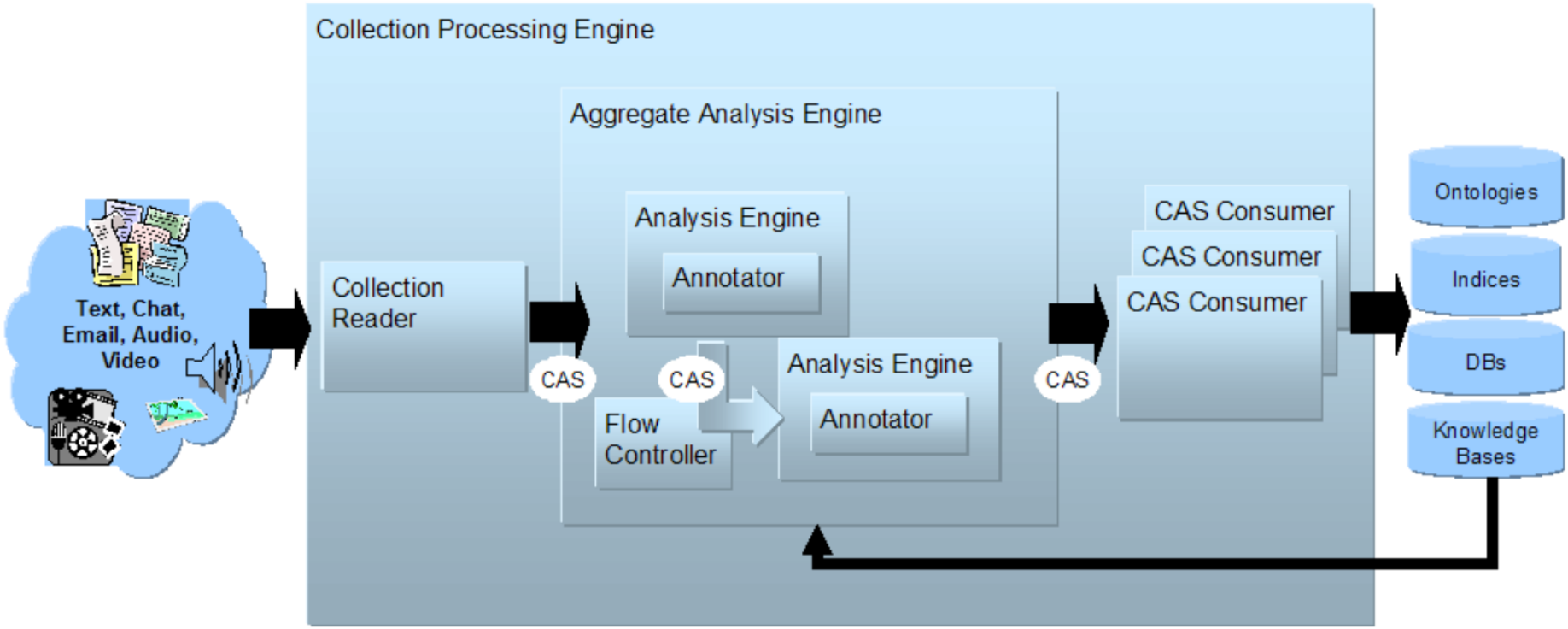
- Eclipse (`http://www.eclipse.org`)
- UIMA plugin for Eclipse (update site: `http://www.apache.org/dist/uima/eclipse-update-site`)
- Maven plugin for Eclipse (update site: `http://download.eclipse.org/technology/m2e/releases`)
- UIMA SDK (`https://uima.apache.org/downloads.cgi`)
  - (optional) in the shell, set the environment variable `$UIMA_HOME` to point to the extracted SDK folder
  - in Eclipse, set the classpath variable `$UIMA_HOME` to point to the same folder

Refer to the System Setup Guide for details.

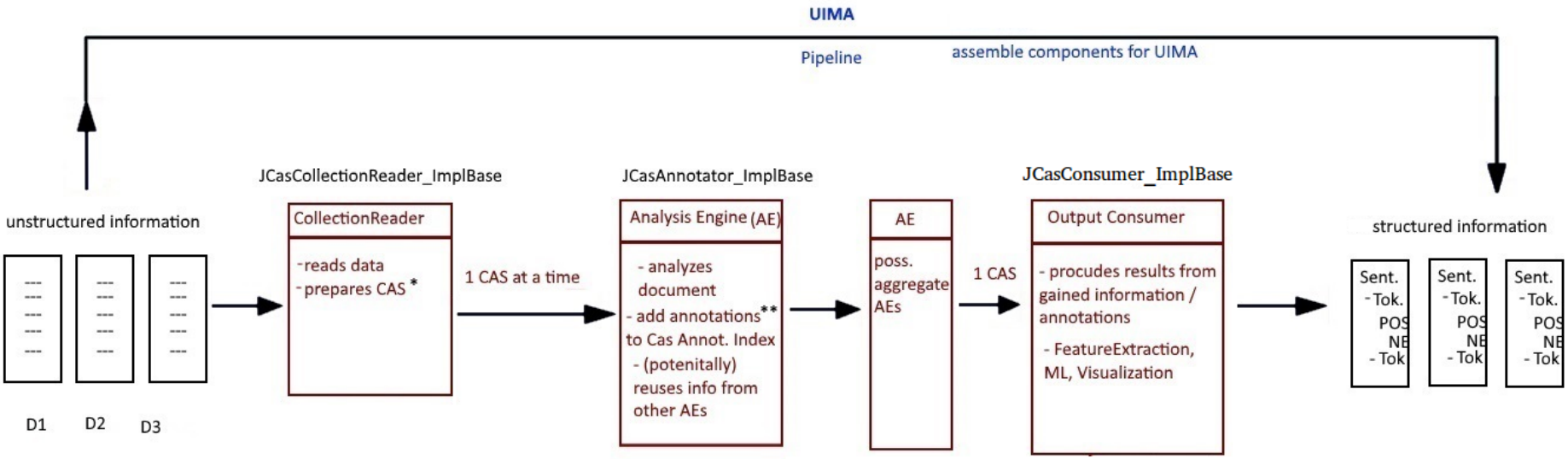
Overview of UIMA Concepts

- Analysis Engine (AE): a program that analyzes artifacts (e.g. documents) and infers information from them.
- Annotator: a component that contains analysis logic. Annotators analyze an artifact and create additional data (metadata) about that artifact.
- What's the difference?
  An AE may contain a single annotator (*Primitive AE*) or a composition of multiple annotators (*Aggregate AE or AAE*).
- Feature Structure: the analysis results, a data structure that have a type and a set of (attribute, value) pairs.
- Common Analysis Structure (CAS): the central data structure through which all UIMA components communicate. It contains: SofA, analysis results, indices to the analysis results, the type system.
- Type System: an object schema for the CAS.

Collection processing



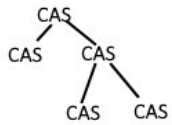
Concept illustration



* CAS = Common Analysis Structure = View

hierarchical annotation index for a document or a part of a document
"what is analyzed together"

CAS
- AnnotationIndex
- DocumentText
- DocumentLanguage
- Meta Data



** Type System

explicit hierarchical declaration of all annotation types shared by all modules

Type
- begin
- end
- name
- features

Excercise

- Design a UIMA pipeline for calculating the **lexical density** of a text. Lexical density is defined as the percentage of lexical words among all words used in a text.
- Lexical words are simply nouns, adjectives, verbs and adverbs which give us information about what is being communicated. Lexical density is considered as a measure of how informative a text is.
- Compare:
  - LD 70%: The **quick brown fox jumped swiftly** over the **lazy dog**.
  - LD 20%: She could have **told** him that she had **loved** him.

Questions

- What NLP processes are required if we want to calculate the LD of a text automatically?
- What UIMA components should we use to realize the function?

Steps to Create an Annotator

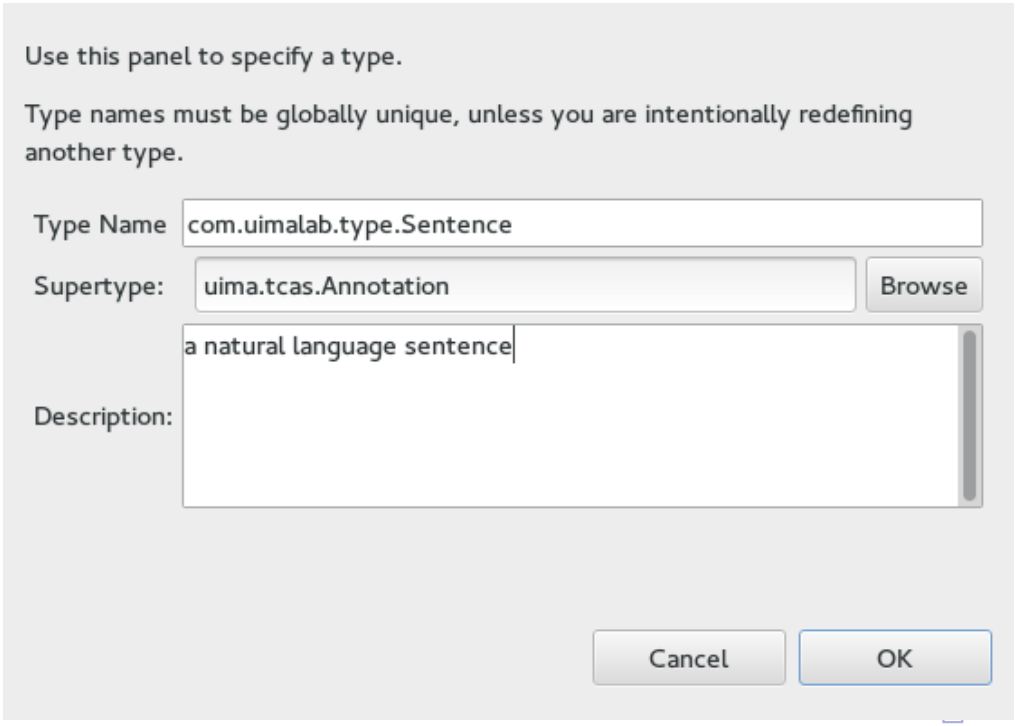
1. Create a type system: type descriptor and corresponding type classes.
2. Create an annotator class that extends JCasAnnotator_implBase.
3. In the annotator class, implement the `process()` method, optionally override `initialize()` or `destroy()`.
4. In the `process()` methods, implement the required analysis logic. Populate the results into the CAS by calling the FS object's `addToIndexes()` mehtod.
5. Create an annotator descriptor and point it to the annotator class.
6. Test the annotator in Document Analyzer.

Defining Types

- The *Type System Descriptor*
- Primitive types: Boolean, Byte, Short, Integer, Long, Float, Double, as well as Arrays of these primitive types.
- UIMA predefined types: TOP (the root of the type system, analogous to Object in Java), FSArray, Annotation (begin, end).
- Generate Java classes corresponding to the types defined. Set automatic or use JCasGen.

Tutorial 1: Defining a Sentence Type

- Right-click the file **/src/main/resources/descriptor/TypeSystem.xml** and choose **Open with/Component Descriptor Editor**.
- In the editor window, click the **Add Type** button and enter the Type Name, Supertype, and Description infomation as shown in the figure.
- Save the TypeSystem.xml descriptor.



Implementing the Annotator Class

- Create a new class that extneds `JCasAnnotator_ImplBase`. The class needs to be public, not abstract, and must have a public, 0-argument constructor so that the UIMA framework can instantiate it.
- Implements `process()`, which is called for each artifact the framework analyzes. Populate analysis results to the CAS.
- (optional) Overrides `initialize()`, which is called once when the framework first creates an instance of the annotator class.
- (optional) Overrides `destory()`, which is called when the framework is done using the annotator.

Observe the file **SentenceAnnotator.java** in the **com.uimalab.lexical_density** package. Look at the outline of the class and read the comments to understand the function of the sentence annotator.

The AE XML Descriptor

- Using the Component Descriptor Editor
- Name, description, version, and vendor
- The annotator's inputs and outputs, defined in terms of the types in a Type System Descriptor
- Declaration of the configuration parameters that the annotator accepts.
- Setting and accessing parameter values from the annotator code:
  `(String) aContext.GetConfigParameterValue("key");`
  Type casting is necessary because the return type is Object.
- Declaring resource dependencies and accessing the resource from the UimaContext:

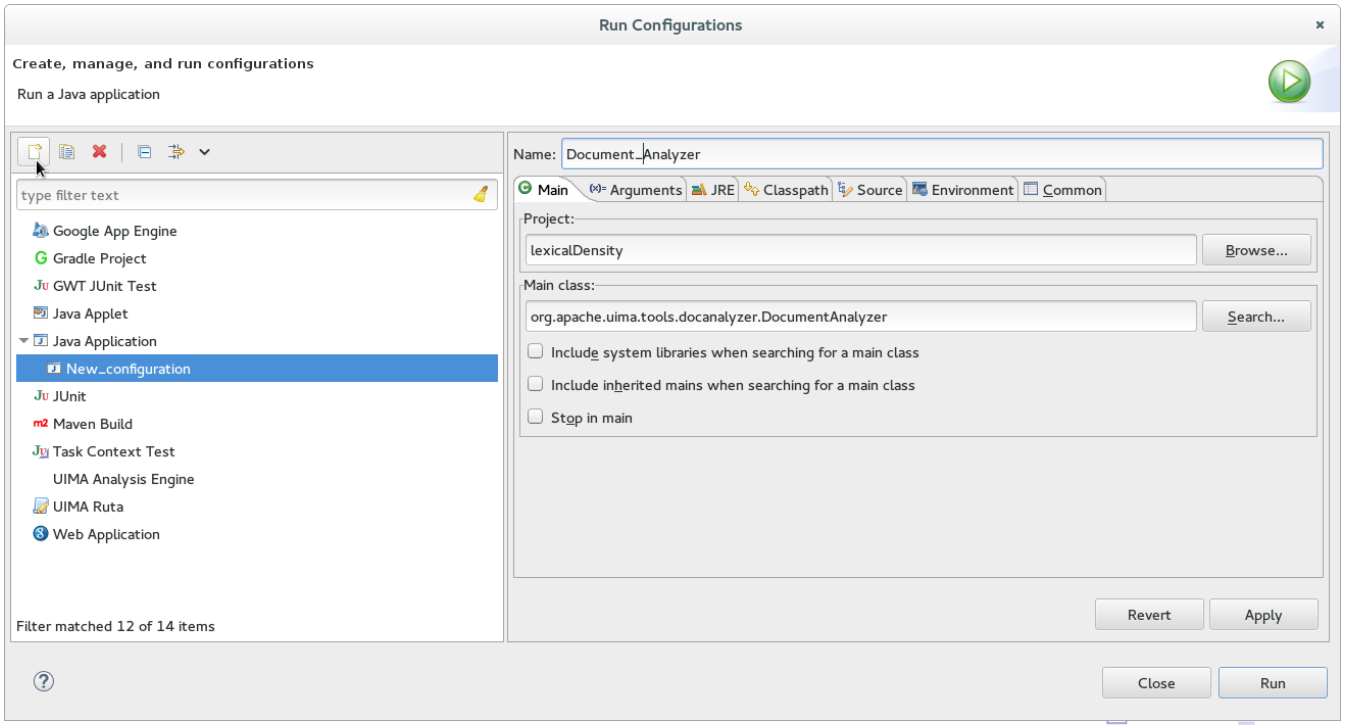
```
InputStream stream =
getContext().getResourceAsStream("resourceKey");
```

Tutorial 1: Linking an AE Descriptor and Its Implementation

- Open the file **resources/descriptor/Tutorial01_Sent_Annotator/SentenceAnnotator.xml** in the component editor.
- On the **Overview** tab, click **Browse** and type the name of the AE implementation class: **SentenceAnnotator**.
- Import the Type System from the **Type System** panel by clicking the **Add...** button on the right panel and selecting the **TypeSystem.xml** file from the list of selections.
- Adding capability declarations on the **Capabilities** tab by clicking the Add Type button and select **Sentence** as output.

Tutorial 1: Testing the Sentence Annotator

- Run the Document Analyzer that comes with the UIMA SDK:
  - Go to the menu **Run** → Run Configurations...
  - Enter the information as shown in the figure and click **Run**.
  - In the Document Analyzer window, set the AE descriptor path, the input and output folders to run the **Sentence Annotator** we've just created.
  - Observe the results from the resulting windows.



Aggregate Analysis Engines

- The Sentence Annotator is a *primitive AE*.
- An AAE is an AE that combines two or more primitive AEs.
- Documents "flow" through these AEs in a predefined order.
- An AAE is a pipeline of analysis that adds multiple annotations/meta-information to a document.

Tutorial 2: Creating an AAE

- A tokenizer AE that depends on the Sentence Annotator output has been created.
- Right-click the **Tutorial02_Token_AAE** folder and choose **New** $\rightarrow$ **Other...**. Select **UIMA** $\rightarrow$ **Analysis Engine Descriptor File** from the Wizard and name the file **TokenAnnotatorAAE.xml**.
- On the **Overview** panel, select **Aggregate** for Engine Type.
- Add the Sentence Annotator and Token Annotator descriptors from the **Aggregate** tab.
- Set the AAE's capabilities as outputing **Token**.
- Run the AAE from the Document Analyzer and observe the resulting annotations.

Tutorial 3: Annotating Part-of-Speech

- Run the POSAnnotator and POSAnnotatorAAE in the Document Analyzerseparately and answer the following questions:
  - What do the two annotators do? What are their differences?
  - What different results do you get from these two analysis engines?
- Try to make sense of the POS tags produced by the POSAnnotatorAAE by refering to the Penn Treebank Tag Set
- For our purpose of calculating the lexical density of a document, what other downstream processes do we need to implement?

Tutorial 4: Calculating Lexical Density

- The **Tutorial04_Lexical_Density** folder contains a primitive and an aggregate engine for calculating lexical density of a document.
- The POS to be calculated need to be specified in the parameter **POSTypes** with the **Parameter Settings** panel.
- Figure out the tags of the lexical POS with the Penn Treebank tag set and add them to the parameter setting. These POS types will be calculated in the lexical density calculation.
- Test the AAE in the Document Analyzer.

Exercise: Can you create an AAE that calculate the **Noun Density** of a document?

Tutorial 5: Configuring CPEs with the UIMA CPE GUI

- Create a run configuration that runs the **CpmFrame** class from the UIMA SDK.
- Specify descriptors for:
  - Collection Reader: FileSystemCollectionReader.xml
  - Analysis Engines, or AAEs: LexicalDensityAAE.xml
  - CAS Consumers: GeneralLDCasConsumer.xml
- Set the parameters for each component
- Run the configured CPE
- Save/Load CPE descriptors

Exercise: Plug in another CAS Consumer (**DetailedLDCasConsumer.xml**) to the configured CPE and observe the running results. What data output do you get?

Developing a Collection Reader (Advanced)

- Create a Collection Reader descriptor file
- Point to the Java class that implements `CollectionReader` or extends `CollectionReader_implBase`.
- Implements the abstract methods: `initialize()`, `hasNext()`, `getNext(CAS)`, `getProgress()`, `close()`, etc.

Implementing initialize()

Called once when the collection reader is first created.

- Access configuration parameters
- Load resources
- Implement other initialization logic

```
public void initialize() throws ResourceInitializationException {
  File directory = new File(
            (String)getConfigParameterValue(PARAM_INPUTDIR));
  mEncoding = (String)getConfigParameterValue(PARAM_ENCODING);
  mDocumentTextXmlTagName = (String)getConfigParameterValue(PARAM_XMLTAG);
  mLanguage = (String)getConfigParameterValue(PARAM_LANGUAGE);
  mCurrentIndex = 0;

  //get list of files (not subdirectories) in the specified directory
  mFiles = new ArrayList();
  File[] files = directory.listFiles();
  for (int i = 0; i < files.length; i++) {
    if (!files[i].isDirectory()) {
      mFiles.add(files[i]);
    }
  }
}
```

Implementing hasNext() and getNext(CAS)

`hasNext()`: checks if there are any documents remaining to be read from the collection. e.g.

```
public boolean hasNext() {
  return mCurrentIndex < mFiles.size();
}
```

`getNext()`: reads the next document from the collection and populates a CAS. e.g.

```
public void getNext(CAS aCAS) throws IOException, CollectionException {
  JCas jcas;
  try {
    jcas = aCAS.getJCas();
  } catch (CASException e) {
    throw new CollectionException(e);
  }

  // open input stream to file
  File file = (File) mFiles.get(mCurrentIndex++);
  String text = FileUtils.file2String(file, mEncoding);
    // put document in CAS
  jcas.setDocumentText(text);
```

Implementing getProgress() and close()

`getProgress()`: returns how much of the collection has been read thus far and how much remains to be read. e.g.

```
public Progress[] getProgress() {
  return new Progress[]{
     new ProgressImpl(mCurrentIndex,mFiles.size(),Progress.ENTITIES)};
}
```

`close()`: called when the Collection Reader is no longer needed. Used to release resources it may be holding.

Developing CAS Consumers (Advanced)

- CAS Consumer is just another Analysis Engine! It typically only extracts data fromt he CAS but doe not update it.
- It persists selected information to files or databases.
- two components:
  - XML Descriptor
  - Java class that extends `CasConsumer_ImplBase` or implements `CasConsumer`
- e.g. XmiWriterCasConsumer from the UIMA SDK examples project