Unstructured Information Management Architecture (UIMA) SDK User's Guide and Reference

July 2006

This edition applies to the IBM Unstructured Information Management Architecture (UIMA) SDK Version 1.4.2 and to all subsequent release and modifications until otherwise indicated in new editions.

Acknowledgements

The UIMA SDK Documentation was prepared as part of a collaborative effort between IBM Research and IBM Software Group. We would like to acknowledge the following people for their contributions to the UIMA SDK documentation. They are listed here in alphabetical order.

Eric Brown co-authored the UIMA collection processing developer guide and reference documentation. Eric manages the "UIM Systems" group at T.J. Watson Research Center. His team develops UIMA metadata tooling and advanced knowledge gathering frameworks using UIMA.

Jerry Cwiklik co-authored the UIMA collection processing developer's guide and reference documents. He is the lead developer of the UIMA collection processing manager.

Yurdaer Doganata, together with Marshall Schor, formatted and assembled the SDK documentation set. Yurdaer is the manager of "Information Management solutions" at IBM's T.J. Watson Research Center. His team is focused on UIMA component discovery and reuse and is responsible for UIMA development tooling.

Gabriele Dreckschmidt formatted and assembled parts of the documentation set. She is a member of the information development team within IBM Software Group.

Youssef Drissi authored the PEAR Packager documentation. He is UIMA software engineer focused on UIMA development tooling.

Edward Epstein reviewed various developer and reference documents. Eddie manages the "UIMA Frameworks" group at IBM at the T.J. Watson Research Center. His team is dedicated to the core UIMA framework including the analysis engine and collection processing architecture.

David Ferrucci authored the SDK Overview, the UIMA Conceptual Overview, and the FAQs. He is UIMA's lead architect and the Senior Manager of the "Semantic Analysis and Integration" department at the IBM T.J. Watson Research Center. His department is primarily focused on development of advanced middleware and technologies for processing unstructured information, including UIMA.

Thilo Goetz authored the CAS Visual Debugger Guide and co-authored the CAS Reference. Thilo is a software engineer in IBM Software Group and the lead developer of UIMA's common analysis system. He is focused on the productization of UIMA and its integration with search-related products.

Thomas Hampp reviewed the overall documentation. Thomas leads the UIMA product development work within IBM's Software Group.

Lev Kozakov authored the PEAR Installer documentation. He is a Research Staff Member at IBM's T.J. Watson Research center focused on UIMA tooling and metadata representation.

Adam Lally authored the Annotator and Analysis Engine Developers Guide, the Application Developer's Guide, the CPE Configurator Guide, the Document Analyzer User's Guide, the XCAS Annotation Viewer's Guide, and the Eclipse setup instructions. He co-authored the Component Descriptor Reference and CPE Developer's Guide. Adam is the lead developer of the UIMA framework.

Jonathan Lenchner authored the Component Descriptor Editor Guide. He is software engineer at IBM's T.J. Watson research center and, among several key roles, he is the lead developer of UIMA Component Descriptor Editor.

Yosi Mass adapted documentation written for the Semantic Search Engine for the UIMA SDK. He is a Researcher at IBM's Haifa research center, and is the lead developer of UIMA's advanced semantic search capability.

Marshall Schor authored the JCas and JCasGen documentation and co-authored the CAS Reference. He reviewed and edited all of the UIMA SDK documentation. Marshall is the UIMA SDK development manager at the IBM T.J. Watson Research Center. He chairs the UIMA Architecture Board and is also the Program Manager of IBM's Institute for Search and Text Analysis (ISTA).

Part I: UIMA Overview and Setup

Chapter 1 UIMA SDK Overview

IBM's Unstructured Information Management Architecture (UIMA) is an architecture and software framework for creating, discovering, composing and deploying a broad range of multi-modal analysis capabilities and integrating them with search technologies.

The ***UIMA framework*** provides a run-time environment in which developers can plug in and run their UIMA component implementations and with which they can build and deploy UIM applications. The framework is not specific to any IDE or platform.

The ***UIMA Software Development Kit (SDK)*** includes an all-Java implementation of the UIMA framework for the development, description, composition and deployment of UIMA components and applications. It also provides the developer with an Eclipse-based (www.eclipse.org) development environment that includes a set of tools and utilities for using UIMA.

This chapter is the intended starting point for readers that are new to the UIMA SDK. It includes this introduction and the following sections:

- Section 1.1 "UIMA SDK Documentation Overview" provides a list of the chapters included in the UIMA SDK documentation with a brief summary of each.
- Section 1.2 "Using the Documentation to get started with the UIMA SDK" describes a recommended path through the documentation to help get the reader up and running with UIMA,
- Section 1.3 "UIMA SDK Release Notes" are release notes for this version of the UIMA SDK.
- Finally, Section 1.4 "Summary of UIMA SDK Capabilities" includes an inventory of software capabilities provided in the UIMA SDK.

1.1 UIMA SDK Documentation Overview

Chapter	***Description***
Overviews	
UIMA SDK Overview (This Chapter)	Lists the documents provided in the UIMA SDK documentation set. Provides a recommended path through the documentation for getting started using UIMA.

	Includes release notes. Provides a brief high-level description of the different software modules included in the UIMA SDK.
UIMA Conceptual Overview	Provides a broad conceptual overview of the UIMA component architecture making contextual references to the other documents in the UIMA SDK documentation set that provide more detail.
Setting up	
UIMA Eclipse Tooling Installation and Setup	Provides step-by-step instructions for installing the UIMA SDK in the Eclipse Interactive Development Environment.
Developer's Guides	
Annotator and AE Developer's Guide	Tutorial-style guide for building UIMA annotators and analysis engines. This chapter introduces the developer to creating type systems and using UIMA's common data structure, the CAS or Common Analysis Structure. It demonstrates how to use built in tools to specify and create basic UIMA analysis components.
CPE Developer's Guide	Tutorial-style guide for building UIMA collection processing engines. These manage the analysis of collections of documents from source to sink.
Application Developer's Guide	Tutorial-style guide for using UIMA SDK to create, run and manage UIMA components from your application. Includes integration with semantic search engine and description of a simple GUI provided for submitting and running Semantic Search queries that can exploit UIMA analysis. Also describes APIs for saving and restoring the contents of a CAS using an XML format called XCAS.
Developing Applications using Multiple Subjects of Analysis (Sofas)	A single CAS maybe associated with multiple subjects of analysis (Sofas). These are useful for representing and analyzing different formats or translations of the same document. For multi-modal analysis, Sofas are good for different modal representations of the same stream (e.g., audio and close-captions).This chapter provides the developer details on how to use multiple Sofas in an application.
Tool User Guides	
Component Descriptor Editor	Describes the features of the Component Descriptor Editor Tool. This tool provides a GUI for specifying the details of UIMA component descriptors, including those for Analysis Engines

	(primitive and aggregate), Collection Readers, CAS Consumers and Type Systems.
CPE Configurator	Describes the User Interfaces and features of the CPE Configurator tool. This tool allows the user to select and configure the components of a Collection Processing Engine and then to run the engine.
PEAR Packager	Describes how to use the PEAR Packager utility. This utility enables developers to produce an archive file for an analysis engine that includes all required resources for installing that analysis engine in another UIMA environment.
PEAR Installer	Describes how to use the PEAR Installer utility. This utility installs and verifies an analysis engine from an archive file (PEAR) with all its resources in the right place so it is ready to run.
Document Analyzer	Describes the features of a tool for applying a UIMA analysis engine to a set of documents and viewing the results.
CAS Visual Debugger	Describes the features of a tool for viewing the detailed structure and contents of a CAS. Good for debugging.
JCasGen	Describes how to run the JCasGen utility, which automatically builds Java classes that correspond to a particular CAS Type System.
XCAS Viewer	Describes how to run the supplied viewer for XCASes, used in the examples.
References	
UIMA FAQs	Frequently Asked Questions about general UIMA concepts. (Not a programming resource.)
Glossary	Main UIMA concepts and their basic definitions.
Component Descriptor Reference	Provides detailed XML format for all the UIMA component descriptors, except the CPE (see next)
CPE Descriptor Reference	Provides detailed XML format for the Collection Processing Engine descriptor.
JavaDocs	JavaDocs detailing the UIMA SDK programming interfaces
CAS Reference	Provides detailed description of the principal CAS interface.
JCas Reference	Provides details on the JCas, a native Java interface to the CAS.
Semantic Search Engine Reference	Describes how to write applications that query a semantic search engine index built using the UIMA SDK.
PEAR Reference	Provides detailed description of the deployable archive format for UIMA components.

1.2 Using the Documentation to get started with the UIMA SDK

1. Explore this chapter to get an overview of the different documents that are included with the SDK.
2. Read *Chapter 2* ***UIMA Conceptual Overview*** to get a broad view of the basic UIMA concepts and philosophy with reference to the other documents included in the SDK which provide greater detail.
3. For more general information on the UIMA architecture and how it has been used, refer to the IBM Systems Journal special issue on Unstructured Information Management, on line at http://www.research.ibm.com/journal/sj43-3.html or to the external UIMA website where key publications are listed http://www.research.ibm.com/UIMA/pubs.htm.
4. Set up the UIMA SDK in your Eclipse environment. To do this, follow the instructions in *Chapter 3* ***UIMA SDK Setup for Eclipse***.
5. Develop sample UIMA annotators, run them and explore the results. Read *Chapter 4* ***Annotator and Analysis Engine Developer's Guide*** and follow it like a tutorial to learn how to develop your first UIMA annotator and set up and run your first UIMA analysis engines.
   - As part of this you will use a few tools including
     - The UIMA Component Descriptor Editor, described in more detail in *Chapter 9* ***Component Descriptor Editor User's Guide*** and
     - The Document Analyzer, described in more detail in *Chapter 14* ***Document Analyzer User's Guide***.
   - While following along in *Chapter 4* ***Annotator and Analysis Engine Developer's Guide*** reference documents that may help are:
     - *Chapter 20* for understanding the analysis engine descriptors
     - *Chapter 24* ***JCas Reference*** for understanding the JCas
6. Learn how to create, run and manage a UIMA analysis engine as part of an application. Connect your analysis engine to the provided semantic search engine to learn how a complete analysis and search application may be built with the UIMA SDK. *Chapter 6* ***Application Developer's Guide*** will guide you through this process.
   - As part of this you will use the document analyzer (described in more detail in *Chapter 14* ***Document Analyzer User's Guide***) and semantic search GUI tools (described in section 6.5.2 ***Semantic Search Query Tool***.
7. Pat yourself on the back. Congratulations! If you reached this step successfully, then you have an appreciation for the UIMA analysis engine architecture. You would have built a few sample annotators, deployed UIMA analysis engines to analyze a few documents, searched over the results using the built-in semantic search engine and viewed the results through a built-in viewer – all as part of a simple but complete application.

8. Develop and run a Collection Processing Engine (CPE) to analyze and gather the results of an entire collection of documents. *Chapter 5* ***Collection Processing Engine Developer's Guide*** will guide you through this process.
   - As part of this you will use the CPE Configurator tool. For details see *Chapter 10* ***Collection Processing Engine Configurator User's Guide***.
   - You will also learn about CPE Descriptors. The detailed format for these may be found in *Chapter 21* ***Collection Processing Engine Descriptor Reference***.
9. Learn how to package up an analysis engine for easy installation into another UIMA environment. *Chapter 11* ***PEAR Packager*** and *Chapter 12* ***PEAR Installer*** will teach you how to create UIMA analysis engine archives so that you can easily share your components with a broader community.

1.3 UIMA SDK Release Notes

1.3.1 General

The UIMA SDK supports the development, discovery, composition and deployment of multi-modal analytics for the analysis of unstructured information and its integration with search technologies.

It includes APIs and tools for creating analysis components. Examples of analysis components include tokenizers, summarizers, categorizers, parsers, named-entity detectors etc.

The UIMA SDK also includes a semantic search engine for indexing the results of analysis and for using this semantic index to perform more advanced search.

1.3.2 Programming Language Support

UIMA supports the development and integration of analysis algorithms developed in different programming languages.

This release of the SDK is principally focussed on Java development. It also includes facilities for C++ Enablement for UIMA Components which allow UIMA components to be written in C++ and have access to a C++ version of the CAS. When used in this manner, the Java UIMA framework can incorporate analytic functions written in C++. Optional files included with the UIMA SDK describe this functionality and provide example code. See the Quick Start manual for more information on this.

1.3.3 Multi-Modal Support

The UIMA architecture supports the development, discovery, composition and deployment of multi-modal analytics, including text, audio and video. Chapter 7 ***Developing Applications using Multiple Subjects of Analysis*** on page 7-158 discuss this is more detail.

1.4 Summary of UIMA SDK Capabilities

Module	Description
UIMA Framework Core	A framework integrating core functions for creating, deploying, running and managing UIMA components, including analysis engines and Collection Processing Engines in collocated and/or distributed configurations. The framework includes an implementation of core components for transport layer adaptation, CAS management, workflow management based on declarative specifications, resource management, configuration management, logging, and other functions.
Externalized Framework Plug-ins	**Note that interfaces of these components are available to the developer but different implementations are possible in different implementations of the UIMA framework.**
CAS and TCAS classes	These classes provide the developer with typed access to the Common Analysis Structure (CAS), including type system schema, elements, subjects of analysis and indices. The TCAS class is a specialization of the CAS for supporting multiple subjects of analysis (Sofas). The Sofa mechanism supports the independent or simultaneous analysis of multiple views of the same documents, supporting multi-lingual and multi-modal analysis.
JCas	Provides Java-based UIMA Analysis components with native Java object access to CAS types. It is built on the CAS classes.
Collection Processing Management (CPM)	Core functions for running UIMA collection processing engines in collocated and/or distributed configurations. The CPM provides scalability across parallel processing pipelines, check-pointing, performance monitoring and recoverability.
Resource Manager	Provides UIMA components with run-time access to external resources handling capabilities such as resource naming, sharing, and caching.
Configuration Manager	Provides UIMA components with run-time access to their configuration parameter settings.
Logger	Provides access to a common logging facility.

Tools and Utilities	
JCasGen	Utility for generating a Java object model for CAS types from a UIMA XML type system definition.
Saving and Restoring CAS contents	APIs in the core framework support saving and restoring the contents of a CAS to streams using an XML format for the CAS named XCAS.
PEAR packager for Eclipse	Tool for building a UIMA component archive to facilitate porting, registering, installing and testing components.
PEAR Installer	Tool for installing and verifying a UIMA component archive in a UIMA installation.
Component Descriptor Editor	Eclipse Plug-in for specifying and configuring component descriptors for UIMA analysis engines as well as other UIMA component types including Collection Readers and CAS Consumers.
CPE Configurator	Graphical tool for configuring Collection Processing Engines and applying them to collections of documents.
Java Annotation viewer	Viewer for exploring annotations and related CAS data.
CAS Visual Debugger	Provides developer with detailed visual view of the contents of a CAS.
Document Analyzer	Graphical tool for applying analysis engines to sets of documents and viewing results.
Example Analysis Components	
Semantic Search CAS Indexer	CAS Consumer that uses the semantic search engine indexer to build an index from a stream of CASes. Requires the semantic search engine (included).
Database Writer	CAS Consumer that writes the content of selected CAS types into a relational database, using JDBC. This code is in the doc/examples/src/com/ibm/uima/examples/ cpe/PersonTitleDBWriterCasConsumer
Annotators	Set of simple annotators meant for pedagogical purposes. Includes: Date/time, Room-number, Regular expression and Meeting-finder annotator.
File System Collection Reader	Simple Collection Reader for pulling documents from the file system and initializing CASes.
XML CAS Initializer	Simple CAS Initializer that loads the CAS with text.
Search Components	

Semantic Search Engine	Search Engine that supports searching over results of analysis including annotations and nested annotations using the "XML Fragment" query language.
Components not currently available in this release of the UIMA SDK.	**If interested in these extensions please contact the UIMA team at IBM. T.J. Watson Research Center via www.ibm.com/research/uima**
C++ and other programming language Interoperability	Includes C++ CAS and supports the creation of UIMA compliant C++ components that can be deployed in the UIMA run-time through a built-in JNI adapter. This includes high-speed binary serialization. Includes support for creating service-based UIMA engines outside of SDK. This is ideal for wrapping existing code written in different languages.
Semantic search and Analysis Workbench (SAW)	Graphical User Interface for applying analysis to build search indices and DBs and query interfaces for searching/exploring analysis results. Uses the semantic search engine and the EKDB (see below).
Extracted Knowledge Database (EKDB)	Database schema and APIs for creating and populating a relational database with analysis results including entity and relation annotations. Includes a CAS Consumer that populates the database. Semantic Analysis Workbench provides a front-end to this database and to the Semantic Search Engine's query processor.

Table 1: UIMA SDK Capabilities

Chapter 2 UIMA Conceptual Overview

UIMA is an open, industrial-strength, scaleable and extensible platform for creating, integrating and deploying unstructured information management solutions from powerful text or multi-modal analysis and search components.

IBM is making the UIMA SDK available as free software to provide a common foundation for industry and academia to collaborate and accelerate the world-wide development of technologies critical for discovering vital knowledge present in the fastest growing sources of information today.

This chapter presents an introduction to many essential UIMA concepts. It is meant to provide a broad overview to give the reader a quick sense of UIMA's basic architectural philosophy and the UIMA SDK's capabilities.

This chapter provides a general orientation to UIMA and makes liberal reference to the other chapters in the UIMA SDK documentation set, where the reader may find detailed treatments of key concepts and development practices. It may be useful to refer to *Chapter 19* ***Glossary of Key Terms and*** Concepts, to become familiar with the terminology in this overview.

2.1 UIMA Introduction

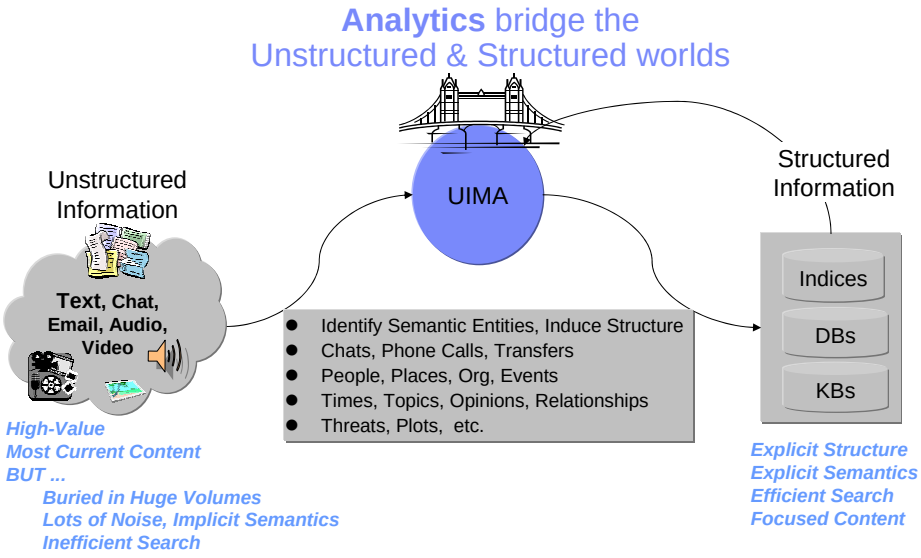


Figure 1: UIMA helps you build the bridge between the unstructured and structured worlds

Unstructured information represents the largest, most current and fastest growing

source of information available to businesses and governments. The web is just the tip of the iceberg. Consider the mounds of information hosted in the enterprise and around the world and across different media including text, voice and video. The high-value content in these vast collections of unstructured information is, unfortunately, buried in lots of noise. Searching for what you need or doing sophisticated data mining over unstructured information sources presents new challenges.

An unstructured information management (UIM) application may be generally characterized as a software system that analyzes large volumes of unstructured information (text, audio, video, images, etc.) to discover, organize and deliver relevant knowledge to the client or application end-user. An example is an application that processes millions of medical abstracts to discover critical drug interactions. Another example is an application that processes tens of millions of documents to discover key evidence indicating probable competitive threats.

First and foremost, the unstructured data must be analyzed to interpret, detect and locate concepts of interest that are not explicitly tagged or annotated in the original artifact, for example, named entities like persons, organizations, locations, facilities, products etc. More challenging analytics may detect things like opinions, complaints, threats or facts. And then there are relations, for example, located in, finances, supports, purchases, repairs etc. The lists of concepts important for applications to detect and find in unstructured resources are large and often domain specific. Specialized component analytics must be combined and integrated to do the job.

The result of analysis must, in turn, be put in structured forms so that powerful data mining and search technologies like search engines, database engines or OLAP (On-Line Analytical Processing, or Data Mining) engines may be leveraged to efficiently find the concepts you need, when you need them.

In analyzing unstructured content, UIM applications make use of a variety of analysis technologies including:

- Statistical and rule-based Natural Language Processing (NLP)
- Information Retrieval (IR)
- Machine learning
- Ontologies
- Automated reasoning and
- Knowledge Sources (e.g., CYC, WordNet, FrameNet, etc.)

These technologies are developed independently by highly specialized scientists and engineers using different techniques, interfaces and platforms.

The bridge from the unstructured world to the structured world is built through the composition and deployment of these analysis capabilities. This integration is often a costly challenge.

IBM's Unstructured Information Management Architecture (UIMA) is an architecture and software framework that helps you build that bridge. It supports creating, discovering, composing and deploying a broad range of analysis capabilities and linking them to structured information services.

IBM's UIMA allows development teams to match the right skills with the right parts of a solution and helps enable rapid integration across technologies and platforms using a variety of different deployment options. These ranging from tightly-coupled deployments for high-performance, single-machine, embedded solutions to parallel and fully distributed deployments for highly flexible and scaleable solutions.

2.2 The Architecture, the Framework and the SDK

UIMA is a software architecture which specifies component interfaces, data representations, design patterns and development roles for creating, describing, discovering, composing and deploying multi-modal analysis capabilities.

The ***UIMA framework*** provides a run-time environment in which developers can plug in their UIMA component implementations and with which they can build and deploy UIM applications. The framework is not specific to any IDE or platform.

The ***UIMA Software Development Kit (SDK)*** includes an all-Java implementation of the UIMA framework for the implementation, description, composition and deployment of UIMA components and applications. It also provides the developer with an Eclipse-based (www.eclipse.org) development environment that includes a set of tools and utilities for using UIMA.

2.3 Analysis Basics

Key UIMA Concepts Introduced in this Section: Analysis Engine, Document, Annotator, Annotator Developer, Type, Type System, Feature, Annotation, CAS, Sofa, JCas, UIMA Context.

2.3.1 Analysis Engines, Annotators and Analysis Results

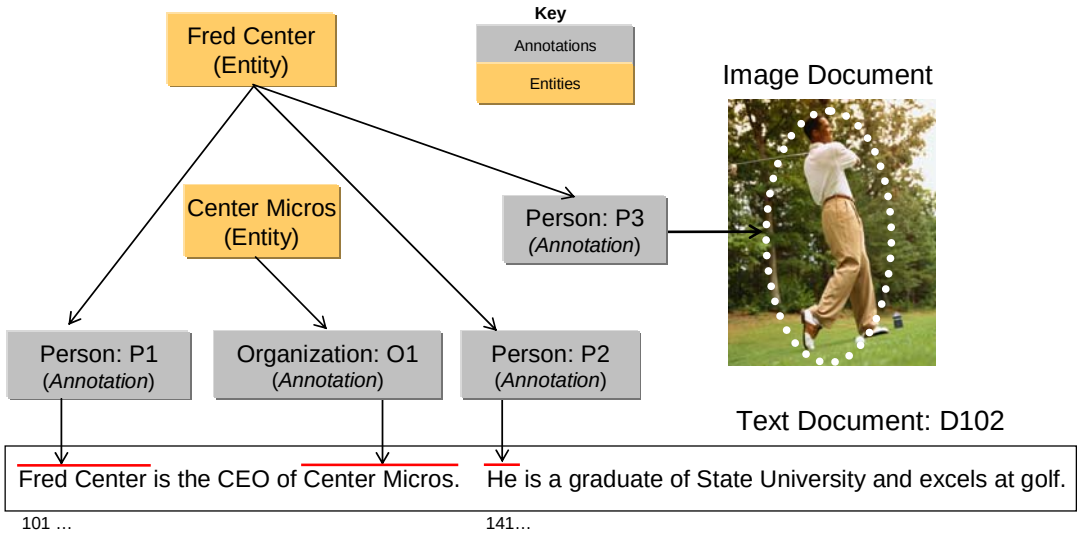


Figure 2: Illustration of objects that might be represented in the UIMA Common Analysis Structure (CAS)

UIMA is an architecture in which basic building blocks called Analysis Engines (AEs) are composed to analyze a document and infer and record descriptive attributes about the document as a whole, and/or about regions therein. This descriptive information, produced by AEs is referred to generally as **analysis results**. Analysis results typically represent meta-data about the document content. One way to think about AEs is as software agents that automatically discover and record meta-data about original content.

UIMA supports the analysis of different modalities including text, audio and video. The majority of examples we provide are for text. We use the term ***document,*** therefore, to generally refer to any unit of content that an AE may process, whether it is a text document or a segment of audio, for example. See the section on page 2-38 for more information on multimodal processing in UIMA.

Analysis results include different statements about the content of a document. For example, the following is an assertion about the topic of a document:

```
(1) The Topic of document D102 is "CEOs and Golf".
```

Analysis results may include statements describing regions more granular than the entire document. We use the term ***span*** to refer to a sequence of characters in a text document. Consider that a document with the identifier D102 contains a span,

"Fred Centers" starting at character position 101. An AE that can detect persons in text may represent the following statement as an analysis result:

```
(2) The span from position 101 to 112 in document D102 denotes a Person
```

In both statements 1 and 2 above there is a special pre-defined term or what we call in UIMA a ***Type***. They are *Topic* and *Person* respectively. UIMA types characterize the kinds of results that an AE may create – more on types later.

Other analysis results may relate two statements. For example, an AE might record in its results that two spans are both referring to the same person:

```
(3) The Person denoted by span 101 to 112 and the Person denoted by span
141 to 143 in document D102 refer to the same Entity.
```

The above statements are some examples of the kinds of results that AEs may record to describe the content of the documents they analyze. These are not meant to indicate the form or syntax with which these results are captured in UIMA – more on that later in this overview.

The UIMA framework treats Analysis engines as pluggable, composible, discoverable, managed objects. At the heart of AEs are the analysis algorithms that do all the work to analyze documents and record analysis results.

UIMA provides a basic component type intended to house the core analysis algorithms running inside AEs. Instances of this component are called ***Annotators***. The analysis algorithm developer's primary concern therefore is the development of annotators. The UIMA framework provides the necessary methods for taking annotators and creating analysis engines.

In UIMA the person who codes analysis algorithms takes on the role of the ***Annotator Developer***. *Chapter 4* ***Annotator and Analysis Engine Developer's*** Guide will take the reader through the details involved in creating UIMA annotators and analysis engines.

At the most primitive level an AE wraps an annotator adding the necessary APIs and infrastructure for the composition and deployment of annotators within the UIMA framework. The simplest AE contains exactly one annotator at its core. Complex AEs may contain a collection of other AEs each potentially containing within them other AEs.

2.3.2 Representing Analysis Results in the CAS

How annotators represent and share their results is an important part of the UIMA architecture. UIMA defines a ***Common Analysis Structure (CAS)*** precisely for these purposes.

The CAS is an object-based data structure that allows the representation of objects, properties and values. Object types may be related to each other in a single-inheritance hierarchy. The CAS logically (if not physically) contains the document being analyzed. Analysis developers share and record their analysis results in terms of an object model within the CAS.[1]

The UIMA framework includes an implementation and interfaces to the CAS. For a more detailed description of the CAS and its interfaces see *Chapter 23* ***CAS Reference***.

A CAS that logically contains statement 2 (repeated here for your convenience)

```
(2) The span from position 101 to 112 in document D102 denotes a Person
```

would include objects of the Person type. For each person found in the body of a document, the AE would create a Person object in the CAS and link it to the span of text where the person was mentioned in the document.

While the CAS is a general purpose representational structure, UIMA defines a few basic types and affords the developer the ability to extend these to define an arbitrarily rich ***Type System***. You can think of a type system as an object schema for the CAS.

A type system defines the various types of objects that may be discovered in documents and recorded by AEs.

As suggested above, Person may be defined as a type. Types have properties or ***features***. So for example, *Age* and *Occupation* may be defined as features of the Person type.

Other types might be *Organization, Company, Bank, Facility, Money, Size, Price, Phone Number, Phone Call, Relation, Network Packet, Product, Noun Phrase, Verb, Color, Parse Node, Feature Weight Array* etc.

[1] We have plans to extend the representational capabilities of the CAS and align its semantics with the semantics of the OMG's Essential Meta-Object Facility (EMOF) and with the semantics of the Eclipse Modeling Framework's (http://www.eclipse.org/emf/) Ecore semantics and XMI-based representation.

There are no limits to the different types that may be defined in a type system. A type system is domain and application specific.

Types in a UIMA type system may be organized into a taxonomy. For example, *Company* may be defined as a subtype of *Organization*. *NounPhrase* may be a subtype of a *ParseNode*.

The Annotation Type

A general and common type used in artifact analysis and from which additional types are often derived is the ***annotation*** type.

The annotation type is used to annotate or label regions of an artifact. Common artifacts are text documents, but they can be other things, such as audio streams. The annotation type for text includes two features, namely *begin* and *end*. Values of these features represent offsets in the artifact and delimit a span. Any particular annotation object identifies the span it annotates with the *begin* and *end* features.

The key idea here is that the annotation type is used to identify and label or "annotate" a specific region of an artifact.

Consider that the Person type is defined as a subtype of annotation. An annotator, for example, can create a Person annotation to record the discovery of a mention of a person between position 141 and 143 in document D102. The annotator can create another person annotation to record the detection of a mention of a person in the span between positions 101 and 112.

Not Just Annotations

While the annotation type is a useful type for annotating regions of a document, annotations are not the only kind of types in a CAS. A CAS is a general representation scheme and may store arbitrary data structures to represent the analysis of documents.

As an example, consider statement 3 above (repeated here for your convenience).

```
(3) The Person denoted by span 101 to 112 and the Person denoted by span
141 to 143 in document D102 refer to the same Entity.
```

This statement mentions two person annotations in the CAS; the first, call it P1 delimiting the span from 101 to 112 and the other, call it P2, delimiting the span from 141 to 143. Statement 3 asserts explicitly that these two spans refer to the same entity. This means that while there are two expressions in the text represented by the annotations P1 and P2, each refers to one and the same person.

The Entity type may be introduced into a type system to capture this kind of information. The Entity type is not an annotation. It is intended to represent an object in the domain which may be referred to by different expressions (or mentions) occurring multiple times within a document (or across documents within a collection of documents). The Entity type has a feature named *occurrences*. This feature is used to point to all the annotations believed to label mentions of the same entity.

Consider that the spans annotated by P1 and P2 were "Fred Center" and "He" respectively. The annotator might create a new Entity object called `FredCenter`. To represent the relationship in statement 3 above, the annotator may link FredCenter to both P1 and P2 by making them values of its *occurrences* feature.

Figure 2 also illustrates that an entity may be linked to annotations referring to regions of image documents as well. To do this the annotation type would have to be extended with the appropriate features to point to regions of an image.

Multiple Views within a CAS

UIMA supports the simultaneous analysis of multiple views of a document. This support comes in handy for processing multiple modalities, for example, the audio and the closed captioned views of a single speech stream.

AEs analyze one or more views of a document. We refer to a single view, generally, as a ***subject of analysis*** (***Sofa***). The CAS therefore may contain one or more Sofas plus the descriptive objects that represent the analysis results for each.

Another common example of using Sofas is for different translations of a document. Each translation may be represented with a different Sofa in a single CAS. Each Sofa may be described by a different set of analysis results. For more details on Sofas see *Chapter 7* on page 7-158.

2.3.3 Interacting with the CAS and External Resources

The two main interfaces that a UIMA component developer interacts with are the CAS and the UIMA Context.

UIMA provides an efficient implementation of the CAS with multiple programming interfaces. Through these interfaces, the annotator developer interacts with the document and reads and writes analysis results. The CAS interfaces provide a suite of access methods that allow the developer to obtain indexed iterators to the different objects in the CAS. See ***Chapter 23 CAS Reference***. While many objects may exist in a CAS, the annotator developer can obtain a specialized iterator to all Person objects, for example.

For Java annotator developers, UIMA provides the JCas. This interface provides the Java developer with a natural interface to CAS objects. Each type declared in the type system appears as a Java Class. So the UIMA framework would render the Person type as a Person class in Java. As the analysis algorithm detects mentions of persons in the documents, it can create Person objects in the CAS. For more details on how to interact with the CAS refer to *Chapter 24 **JCas Reference***.

The component developer in addition to interacting with the CAS can access external resources through the framework's resource manager interface called the ***UIMA Context***. This interface, among other things, can ensure that different annotators working together in an aggregate flow may share the same instance of an external file, for example. For details on using the UIMA Context see *Chapter 4 **Annotator and Analysis Engine Developer's*** Guide.

2.3.4 Component Descriptors

UIMA defines a small set of core components. Annotators and AEs are two of the basic building blocks specified by the architecture. Developers implement them to build and compose analysis capabilities and ultimately applications.

There are others components in addition to these, which we will learn about later, but for every component specified in UIMA there are two parts required for its implementation:

1. the declarative part and
2. the code part.

The declarative part contains metadata describing the component, its identity, structure and behavior and is called the ***Component Descriptor***. Component descriptors are represented in XML. The code part implements the algorithm. The code part may be a program in Java.

As a developer using the UIMA SDK, to implement a UIMA component it is always the case that you will provide two things: the code part and the Component Descriptor. Note that when you are composing an engine, the code may be already provided in reusable subcomponents. In these cases you may not be developing new code but rather composing an aggregate engine by pointing to other components where the code has been included.

Component descriptors are represented in XML and aid in component discovery, reuse, composition and development tooling. The UIMA SDK provides tools for easily creating and maintaining the component descriptors that relieve the developer from editing XML directly. This tool is described briefly in *Chapter 4 **Annotator and Analysis Engine Developer's Guide***, and more thoroughly in *Chapter 9 **Component Descriptor Editor User's Guide***.

Component descriptors contain standard metadata including the component's name, author, version, and a pointer to the class that implements the component.

In addition to these standard fields, a component descriptor identifies the type system the component uses and the types it requires in an input CAS and the types it plans to produce in an output CAS.

For example, an AE that detects person types may require as input a CAS that includes a tokenization and deep parse of the document. The descriptor refers to a type system to make the component's input requirements and output types explicit. In effect, the descriptor includes a declarative description of the component's behavior and can be used to aid in component discovery and composition based on desired results. UIMA analysis engines provide an interface for accessing the component metadata represented in their descriptors. For more details on the structure of UIMA component descriptors refer to *Chapter 20* ***Component Descriptor Reference***.

2.4 Aggregate Analysis Engines

Key UIMA Concepts Introduced in this Section: Aggregate Analysis Engine, Delegate Analysis Engine, Tightly and Loosely Coupled, Flow Specification, Analysis Engine Assembler

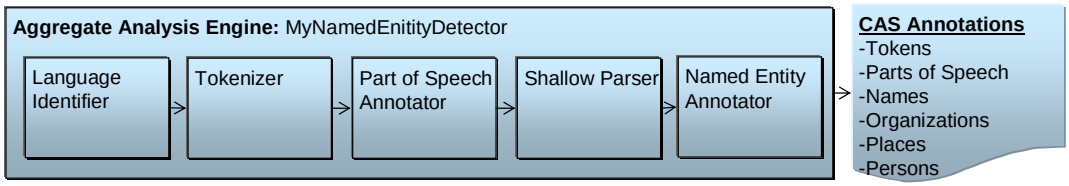


Figure 3: Sample Aggregate Analysis Engine

A simple or primitive UIMA Analysis Engine (AE) contains a single annotator. AEs, however, may be defined to contain other AEs organized in a workflow. These more complex analysis engines are called **Aggregate Analysis Engines.**

Annotators tend to perform fairly granular functions, for example language detection, tokenization or part of speech detection.

Advanced analysis, however may involve an orchestration of many of these primitive functions. An AE that performs named entity detection, for example, may include a pipeline of annotators starting with language detection feeding tokenization, then part-of-speech detection, then deep grammatical parsing and then finally named-entity detection. Each step in the pipeline is required by the

subsequent analysis. For example, the final named-entity annotator can only do its analysis if the previous deep grammatical parse was recorded in the CAS.

Aggregate AEs are built to encapsulate potentially complex internal structure and insulate it from users of the AE. In our example, the aggregate analysis engine developer simply acquires the internal components, defines the necessary flow between them and publishes the resulting AE. Consider the simple example illustrated in Figure 3 where "MyNamed-EntityDetector" is composed of a linear flow of more primitive analysis engines.

Users of this AE need not have to know how it is constructed internally but only its name and its published input requirements and output types. These must be declared in the aggregate AE descriptor. Aggregate AE's descriptors declare the components they contain and a ***flow specification***. The flow specification defines the order in which the internal component AEs should be run. The internal AEs specified in an aggregate are also called the ***delegate analysis engines.***

We refer to the development role associated with building an aggregate from delegate AEs as the ***Analysis Engine Assembler***.

The UIMA framework, given an aggregate analysis engine descriptor, will run all delegate AEs, ensuring that each one gets access to the CAS in the sequence produced by the flow specification. The UIMA framework is equipped to handle different deployments where the delegate engines, for example, are ***tightly-coupled*** (running in the same process) or ***loosely-coupled*** (running in separate processes or even on different machines). The framework supports a number of remote protocols for loose coupling deployments of aggregate analysis engines, including SOAP (which stands for Simple Object Access Protocol, a standard Web Services communications protocol).

The UIMA framework facilitates the deployment of AEs as remote services by using an adapter layer that automatically creates the necessary infrastructure in response to a declaration in the component's descriptor. For more details on creating aggregate analysis engines refer to *Chapter 20* ***Component Descriptor Reference***. The component descriptor editor tool assists in the specification of aggregate AEs from a repository of available engines. For more details on this tool refer to *Chapter 9* ***Component Descriptor Editor User's Guide***.

The UIMA framework implementation currently supports a linear flow between components with conditional branching based on the language of the document. The workflow engine is a pluggable part of the framework, however, and can be easily updated to support more complex flow specifications. Furthermore, the application developer is free to create multiple AEs and provide their own logic to combine the AEs in arbitrarily complex flows. For more details on this the reader may refer to *Chapter 6* ***Application Developer's Guide***.

2.5 Application Building and Collection Processing

Key UIMA Concepts Introduced in this Section: Process Method, Collection Processing Architecture, Collection Reader, CAS Consumer, CAS Initializer, Collection Processing Engine, Collection Processing Manager.

2.5.1 Using the framework from an Application

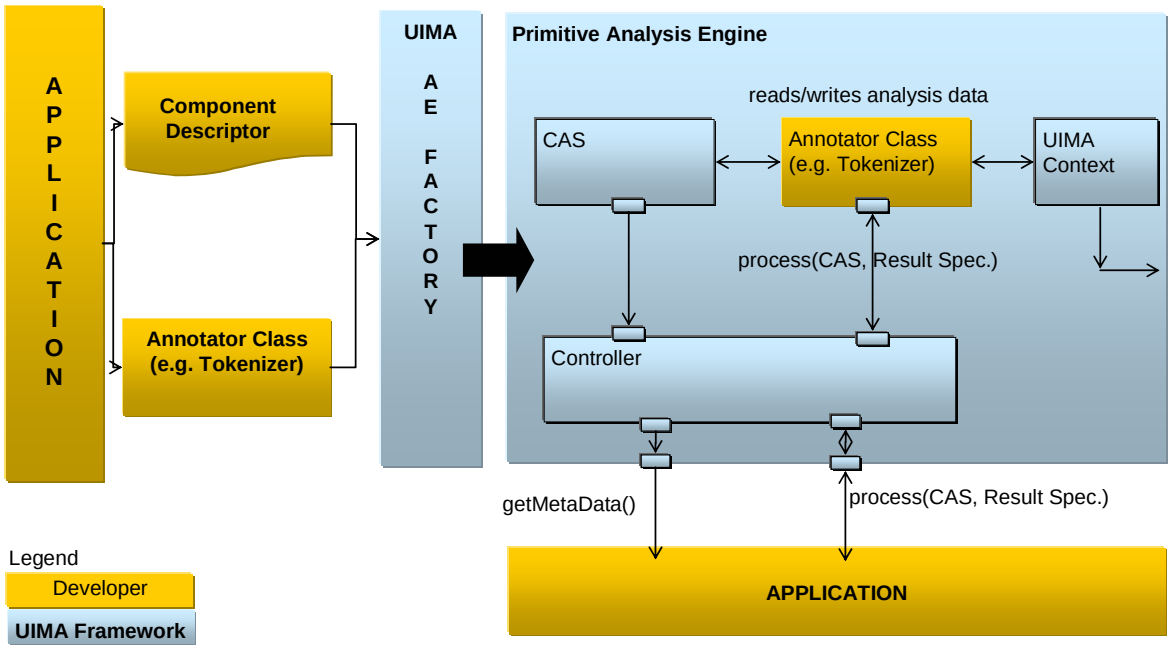


Figure 4: Using UIMA Framework to create and interact with an Analysis Engine

As mentioned above, the basic AE interface may be thought of as simply CAS in/CAS out.

The application is responsible for interacting with the UIMA framework to instantiate an AE, create or acquire an input CAS, initialize the input CAS with a document and then pass it to the AE through the ***process method***. This interaction with the framework is illustrated in Figure 4.

The UIMA AE Factory takes the declarative information from the Component Descriptor and the class files implementing the annotator, and instantiates the AE instance, setting up the CAS and the UIMA Context.

The AE, possibly calling many delegate AEs internally, performs the overall analysis and its process method returns the CAS containing new analysis results.

The application then decides what to do with the returned CAS. There are many possibilities. For instance the application could: display the results, store the CAS to disk for post processing, extract and index analysis results as part of a search or database application etc.

The UIMA framework provides methods to support the application developer in creating and managing CASes and instantiating, running and managing AEs. Details may be found in *Chapter 6* ***Application Developer's Guide***.

2.5.2 Graduating to Collection Processing

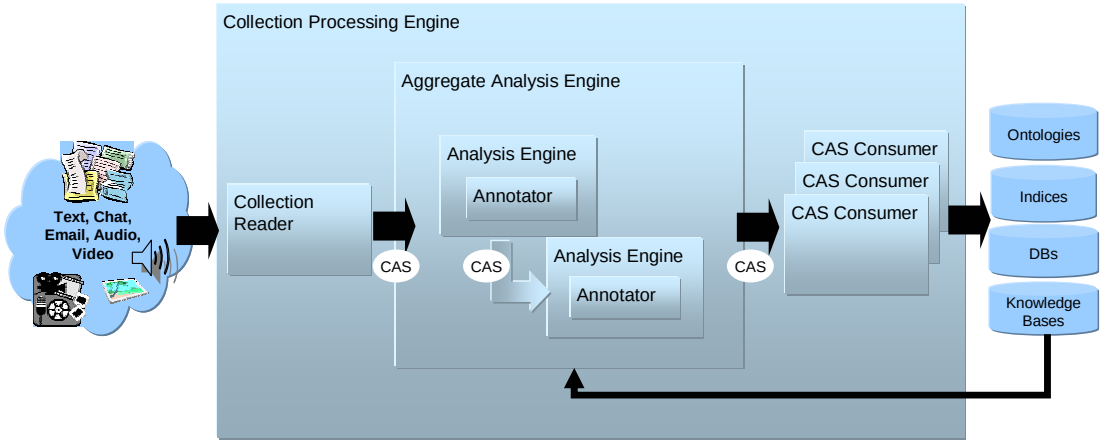


Figure 5: High-Level UIMA Component Architecture from Source to Sink

Many UIM applications analyze entire collections of documents. They connect to different document sources and do different things with the results. But in the typical case, the application must generally follow these logical steps:

1. Connect to a physical source
2. Acquire a document from the source
3. Initialize a CAS with the document to be analyzed
4. Input the CAS to a selected analysis engine
5. Process the resulting CAS
6. Go back to 2 until the collection is processed
7. Do any final processing required after all the documents in the collection have been analyzed

UIMA supports UIM application development for this general type of processing through its **Collection Processing Architecture**.

As part of the collection processing architecture UIMA introduces two primary components in addition to the annotator and analysis engine. These are the **Collection Reader** and the **CAS Consumer**. The complete flow from source, through document analysis, and to CAS Consumers supported by UIMA is illustrated in Figure 5.

The Collection Reader's job is to connect to and iterate through a source collection, acquiring documents and initializing CASes for analysis.

Since the structure, access and iteration methods for physical document sources vary independently from the format of stored documents, UIMA defines another type of component called a **CAS Intializer**. The CAS Initializer's job is specific to a document format and specialized logic for mapping that format to a CAS. In the simplest case a CAS Intializer may take the document provided by the containing Collection Reader and insert it as a subject of analysis (or Sofa) in the CAS. A more advanced scenario is one where the CAS Intializer may be implemented to handle documents that conform to a certain XML schema and map some subset of the XML tags to CAS types and then insert the de-tagged document content as the subject of analysis. Collection Readers may reuse plug-in CAS Initializers for different document formats.

CAS Consumers, as the name suggests, function at the end of the flow. Their job is to do the final CAS processing. A CAS Consumer may be implemented, for example, to index CAS contents in a search engine, extract elements of interest and populate a relational database or serialize and store analysis results to disk for subsequent and further analysis.

A Semantic Search engine is included in the UIMA SDK which will allow the developer to experiment with indexing analysis results and querying for documents based on all the annotations in the CAS. See the section on integrating text analysis and search in *Chapter 6 **Application Developer's Guide**.*

A UIMA **Collection Processing Engine** (CPE) is an aggregate component that specifies a "source to sink" flow from a Collection Reader though a set of analysis engines and then to a set of CAS Consumers.

CPEs are specified by XML files called CPE Descriptors. These are declarative specifications that point to their contained components (Collection Readers, analysis engines and CAS Consumers) and indicate a flow among them. The flow specification allows for filtering capabilities to, for example, skip over AEs based on CAS contents. Details about the format of CPE Descriptors may be found in *Chapter 21 **Collection Processing Engine Descriptor Reference**.*

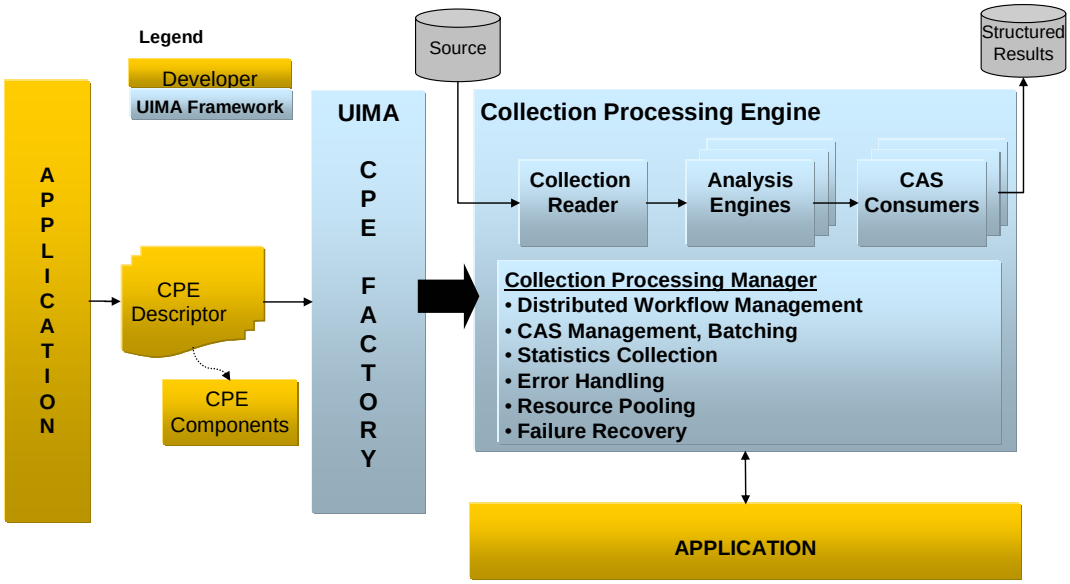


Figure 6: Collection Processing Manager in UIMA Framework

The UIMA framework includes a **Collection Processing Manager** (CPM). The CPM is capable of reading a CPE descriptor, and deploying and running the specified CPE. Figure 6 illustrates the role of the CPM in the UIMA Framework.

Key features of the CPM are failure recovery, CAS management and scale-out.

Collections may be large and take considerable time to analyze. A configurable behavior of the CPM is to log faults on single document failures while continuing to process the collection. This behavior is commonly used because analysis components often tend to be the weakest link -- in practice they may choke on strangely formatted content.

This deployment option requires that the CPM run in a separate process or a machine distinct from the CPE components. A CPE may be configured to run with a variety of deployment options that control the features provided by the CPM. For details see *Chapter 21* ***Collection Processing Engine Descriptor Reference***.

The UIMA SDK also provides a tool called the CPE Configurator. This tool provides the developer with a user interface that simplifies the process of connecting up all the components in a CPE and running the result. For details on using the CPE Configurator see *Chapter 10* ***Collection Processing Engine Configurator User's*** Guide. This tool currently does not provide access to the full set of CPE deployment options supported by the CPM. Anything but the default would have to be

configured by editing the CPE descriptor directly. For details on how to create and run CPEs refer to *Chapter 5* ***Collection Processing Engine Developer's Guide***.

2.6 Exploiting Analysis Results

Key UIMA Concepts Introduced in this Section: Semantic Search, XML Fragment Queries.

2.6.1 Semantic Search

In a simple UIMA Collection Processing Engine (CPE), a Collection Reader reads documents from the file system and initializes CASs with their content. These are then fed to an AE that annotates tokens and sentences, the CASs, now enriched with token and sentence information, are passed to a CAS Consumer that populates a search engine index.

The search engine query processor can then use the token index to provide basic key-word search. For example, given a query “center" the search engine would return all the documents that contained the word “center".

Semantic Search is a search paradigm that can exploit the more powerful analytics pluggable in a UIMA CPE.

Consider that we plugged a named-entity recognizer into the CPE described above. Assume this analysis engine is capable of detecting in documents and annotating in the CAS mentions of persons and organizations.

Complementing the name-entity recognizer we add a CAS Consumer that extracts in addition to token and sentence annotations, the person and organizations added to the CASs by the name-entity detector. It then feeds these into the semantic search engine’s index.

The semantic search engine that comes with the UIMA SDK, for example, can exploit this addition information from the CAS to support more powerful queries. For example, imagine a user is looking for documents that mention an organization with “center" it is name but is not sure of the full or precise name of the organization. A key-word search on “center" would likely produce way too many documents because “center" is a common and ambiguous term. The SDK’s semantic search engine supports a query language called ***XML Fragments***. This query language is designed to exploit the CAS annotations entered in its index. The XML Fragment query, for example,

```
<organization> center </organization>
```

will produce first only documents that contain “center" where it appears as part of a mention annotated as an organization by the name-entity recognizer. This will likely be a much shorter list of documents more precisely matching the user’s interest.

Consider taking this one step further. We add a relationship recognizer that annotates mentions of the CEO-of relationship. We configure the CAS Consumer so that it sends these new relationship annotations to the semantic search index as well. With these additional analysis results in the index we can submit queries like

```
<ceo_of>
    <person> center </person>
    <organization> center </organization>
<ceo_of>
```

This query will precisely target documents that contain a mention of an organization with “center" as part of its name where that organization is mentioned as part of a `CEO-of` relationship annotated by the relationship recognizer.

For more details about using UIMA and Semantic Search see the section on integrating text analysis and search in *Chapter 6* ***Application Developer's Guide***.

2.6.2 Databases

Search engine indices are not the only place to deposit analysis results for use by applications. Another classic example is populating databases. While many approaches are possible with varying degrees of flexibly and performance all are highly dependent on application specifics. We included a simple sample CAS Consumer that provides the basics for getting your analysis result into a relational database. It extracts annotations from a CAS and writes them to a relational database, using the open source Cloudscape / Apache Derby database.

2.7 Multimodal Processing in UIMA

In previous sections we've seen how the CAS is initialized with an initial artifact that will be subsequently analyzed by Analysis engines and CAS Consumers. The first Analysis engine may make some assertions about the artifact, for example, in the form of annotations. Subsequent Analysis engines will make further assertions about both the artifact and previous analysis results, and finally one or more CAS Consumers will extract information from these CASs for structured information

storage.

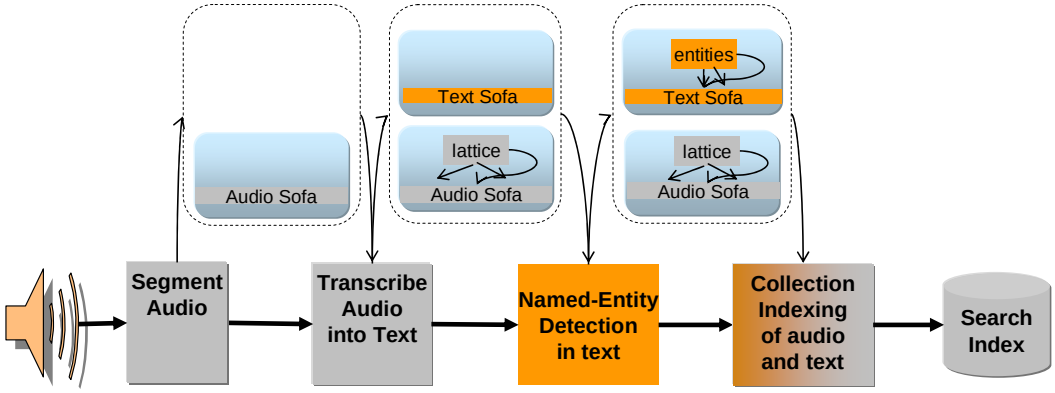


Figure 7: Multiple Sofas in support of multi-modal analysis of an audio Stream. Some engines work on the audio "view", some on the text "view" and some on both.

Consider a processing pipeline, illustrated in Figure 7, that starts with an audio recording of a conversation, transcribes the audio into text, and then extracts information from the text transcript. Analysis Engines at the start of the pipeline are analyzing an audio subject of analysis, and later analysis engines are analyzing a text subject of analysis. The CAS Consumer will likely want to build a search index from concepts found in the text to the original audio segment covered by the concept.

What becomes clear from this relatively simple scenario is that the CAS must be capable of simultaneously holding multiple subjects of analysis. Some analysis engine will analyze only one subject of analysis, some will analyze one and create another, and some will need to access multiple subjects of analysis at the same time.

The support in UIMA for multiple subjects of analysis is called ***Sofa*** support; Sofa is an acronym which is derived from Subject of Analysis. In UIMA a Sofa may be associated with a specialization of the CAS called the ***TCAS***. Multiple TCASes represent different "views" into a common underlying CAS, one for each different subject of analysis.

Analysis results can "belong" to a specific TCAS or they can be independent of any specific TCAS. UIMA components may be Sofa-aware, able to create and access multiple SOFA at the same time, or Sofa-unaware, simply receiving the TCAS view of the CAS corresponding to a single Sofa. In this case the developer need not know about Sofas at all.

Multiple Sofa capability brings benefits to text-only processing as well. An input document can be transformed from one format to another. Examples of this include transforming text from HTML to plain text or from one natural language to another.

Chapter 7 on page 7-158 provides more details on creating Sofa-aware components and creating aggregates and collection processing engines which include specifications on connecting Sofas from one component to Sofas in another.

2.8 Next Step

This chapter presented a high-level overview of UIMA concepts. Along the way, it pointed to other documents in the UIMA SDK documentation set where the reader can find details on how to apply the related concepts in building applications with the UIMA SDK.

At this point the reader may return to the documentation guide in *Chapter 1* on page 1-15 to learn how they might proceed in getting started using UIMA.

For a more detailed overview of the UIMA architecture, framework and development roles we refer the reader to the following paper:

D. Ferrucci and A. Lally, "Building an example application using the Unstructured Information Management Architecture," *IBM Systems Journal* **43**, No. 3, 455-475 (2004).

This paper can be found on line at http://www.research.ibm.com/journal/sj43-3.html

Chapter 3 UIMA SDK Setup for Eclipse

This chapter describes how to set up the UIMA SDK to work with Eclipse. Eclipse (http://www.eclipse.org) is a popular open-source Integrated Development Environment for many things, including Java. The UIMA SDK does not require that you use Eclipse. However, we recommend that you do use Eclipse because some useful UIMA SDK tools run as plug-ins to the Eclipse platform and because the UIMA SDK examples are provided in a form that's easy to import into your Eclipse environment.

If you are not planning on using the UIMA SDK with Eclipse, you may skip this chapter and read *Chapter 4* ***Annotator and Analysis Engine Developer's Guide*** next.

This chapter provides instructions for

- installing Eclipse,
- installing the UIMA SDK's Eclipse plugins into your Eclipse environment, and
- importing the example UIMA code into an Eclipse project.

The UIMA Eclipse plugins are designed to be used with Eclipse version 3.

3.1 Installation

3.1.1 Install Eclipse

- Go to http://download.eclipse.org/downloads
- We recommend using the latest release level (not an "Integration level'). Navigate to the Eclipse Release version you want and download the archive for your platform.
- Unzip the archive to install Eclipse somewhere, e.g., c:\
- Eclipse has a bit of a learning curve. If you plan to make significant use of Eclipse, check out the tutorial under the help menu. It is well worth the effort.

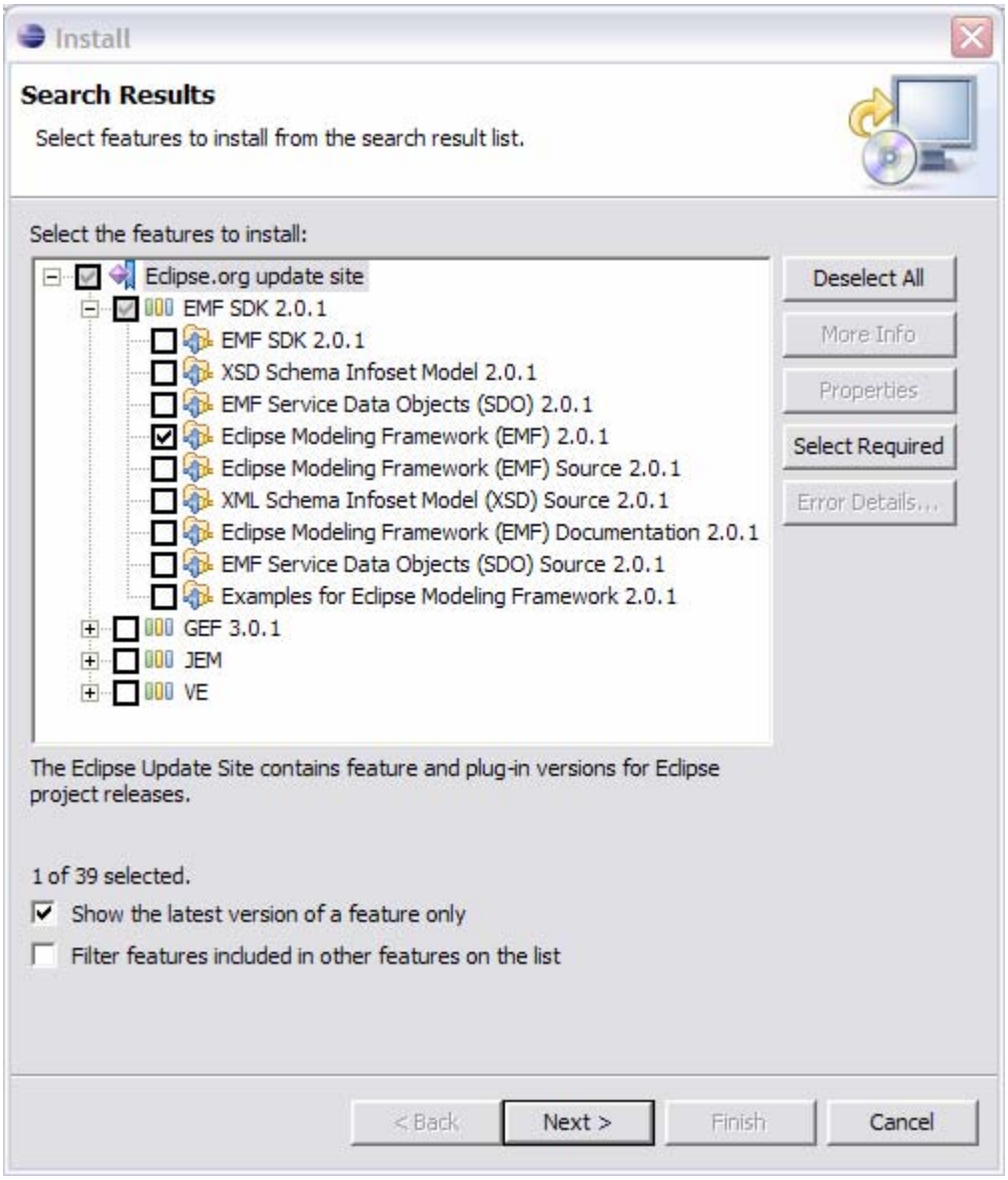
3.1.2 Install EMF

EMF stands for Eclipse Modeling Framework. It is an add-on to Eclipse, and is used by the UIMA Eclipse tooling. Use the built-in facilities in Eclipse to find and install new features.

Activate the software feature finding by using the menu: help >> Software Updates >> Find and Install. Select "Search for new features to install", push "Next". Specify the update site where EMF is found as a site to search, making sure the "Ignore features not applicable to this environment" box is checked, and push "Finish". You

can find out where this site is by going to http://www.eclipse.org and browsing for EMF. In early 2006, the EMF update site was http://download.eclipse.org/tools/emf/updates. If your computer is on an internet connection which uses a proxy server, you can configure Eclipse to know about that. Put your proxy settings into Ecluse using the Eclipse preferences by accessing the menus: Window => Preferences... => Install/Update, and Enable HTTP proxy connection under the Proxy Settings with the information about your proxy.

This will launch a search for updates to Eclipse; it may show a list of update site mirrors – click OK. When it finishes, it shows a list of possible updates in an expandable tree. Expand the tree nodes to find EMF SDK. The specific level may vary from the level shown below as newer versions are released.



Click "Next". Then pick Eclipse Modeling Framework (EMF), and push "Next", accept any licensing agreements, etc., until it finishes the installation. It may say it's an "unsigned feature"; proceed by clicking "Install". If it recommends restarting, you may do that.

This will install EMF, without any extras. (If you want the whole EMF system, including source and documentation, you can pick the "EMF SDK" and the "Examples for Eclipse Modeling Framework".)

3.1.3 Install the UIMA SDK

If you haven't already done so, please download and install the UIMA SDK from http://www.alphaworks.ibm.com/tech/uima.

3.1.4 Install the UIMA Eclipse Plugins

In the directory %UIMA_HOME%/eclipsePlugin, you will find a zip file. (The environment variable %UIMA_HOME% is where you installed the UIMA SDK.) Unzip it into your %ECLIPSE_HOME%/eclipse/plugins directory. %ECLIPSE_HOME% is where you installed Eclipse.

3.1.5 Start Eclipse

If you have Eclipse running, restart it (shut it down, and start it again) using the `clean` option. Start Eclipse by running the command `eclipse -clean` (see explanation in the next section) in the directory where you installed Eclipse. You may want to set up a desktop shortcut at this point for Eclipse. Eclipse has a bit of a learning curve. If you plan to make significant use of Eclipse, check out the tutorial under the help menu. It is well worth the effort. There are also books you can get that describe Eclipse and its use.

The first time Eclipse starts up it will take a bit longer as it completes its installation. A "welcome" page will come up. After you are through reading the welcome information, click on the arrow in the upper right of the main panel to exit the welcome page and get to the main Eclipse screens.

Special startup parameter for Eclipse 3: -clean

If you have modified the plugin structure (by copying or unzipping files directly in the file system) in Eclipse 3.x after you started it for the first time, please include the "-clean" parameter in the startup arguments to Eclipse, ***one time*** (after any plugin modifications were done). This is needed because Eclipse 3.x may not notice the changes you made, otherwise. This parameter forces Eclipse to reexamine all of its plugins at startup and recompute any cached information about them.

3.2 Setting up Eclipse to view Example Code

Later chapters refer to example code. You can create a special project in Eclipse to hold the examples. Here's how:

- In Eclipse, if the Java perspective is not already open, switch to it by going to Window → Open Perspective → Java.
- Set up a class path variable named UIMA_HOME, whose value is the directory where you installed the UIMA SDK., This is done as follows:
  - Go to Window → Preferences → Java → Build Path → Classpath Variables.
  - Click "New"
  - Enter UIMA_HOME (all capitals, exactly as written) in the "Name" field.
  - Enter your installation directory (e.g. `c:/Program Files/IBM/uima`) in the "Path" field
  - Click "OK" in the "New Variable Entry" dialog
  - Click "OK" in the "Preferences" dialog
  - If it asks you if you want to do a full build, click "Yes"
- Select the File -> Import menu option
- Select "Existing Project into Workspace" and click the "Next" button.
- Click "Browse" and browse to the %UIMA_HOME%\docs\examples directory
- Click "Finish." This will create a new project called "uima_examples" in your Eclipse workspace. There should be no compilation errors.

To verify that you have set up the project correctly, check that there are no error messages in the "Tasks" or "Problems" view.

3.3 Running external tools from Eclipse

You can run outside of Eclipse using the shell scripts in the UIMA SDK's bin directory. In addition, many tools can be run from inside Eclipse; examples are the Document Analyzer, CPE Configurator, CAS Visual Debugger, Semantic Search, and JCasGen. The uima_examples project provides Eclipse launch configurations that make this easy to do.

To run these tools from Eclipse:

- If the Java perspective is not already open, switch to it by going to Window → Open Perspective → Java.
- Go to Run → Run...

- In the window that appears, select "UIMA CPE GUI", "UIMA CAS Visual Debugger", "UIMA JCasGen", "UIMA Document Analyzer", or "UIMA Semantic Search" from the list of run configurations on the left. (If you don't see, these, please select the uima_examples project and do a Menu -> File -> Refresh).
- Press the "Run" button. The tools should start. Close the tools by clicking the "X" in the upper right corner on the GUI.

For instructions on using the Document Analyzer and CPE Configurator, see *Chapter 14* ***Document Analyzer User's Guide***, and *Chapter 10* . For instructions on using the CAS Visual Debugger and JCasGen, see *Chapter 15* ***CAS Visual Debugger***, and *Chapter 16* ***JCasGen User Guide***.

Part II: Developer's Guides

Chapter 4 Annotator and Analysis Engine Developer's Guide

This chapter describes how to develop UIMA *type systems*, *Annotators* and *Analysis Engines* using the UIMA SDK. It is helpful to read the UIMA Conceptual Overview chapter for a review on these concepts.

An *Analysis Engine (AE)* is a program that analyzes artifacts (e.g. documents) and infers information from them. A *TAE* is a specialization of an Analysis Engine that analyzes a particular artifact, which is often, for example, a text document (but could be, in general, audio streams, etc.).

In the UIMA SDK, Analysis Engines are constructed from building blocks called *Annotators*. An annotator is a component that contains analysis logic. Annotators analyze an artifact (for example, a text document) and create additional data (metadata) about that artifact. It is a goal of UIMA that annotators need not be concerned with anything other than their analysis logic – for example the details of their deployment or their interaction with other annotators.

An Analysis Engine (AE) may contain a single annotator (this is referred to as a *Primitive AE)*, or it may be a composition of others and therefore contain multiple annotators (this is referred to as an *Aggregate AE*). Primitive and aggregate AEs implement the same interface and can be used interchangeably by applications.

Annotators produce their analysis results in the form of typed *Feature Structures*, which are simply data structures that have a type and a set of (attribute, value) pairs. An *annotation* is a particular type of Feature Structure that is attached to a region of the artifact being analyzed (a span of text in a document, for example).

For example, an annotator may produce an Annotation over the span of text `President Bush`, where the type of the Annotation is `Person` and the attribute `fullName` has the value `George W. Bush`, and its position in the artifact is character position 12 through character position 26.

It is also possible for annotators to record information associated with the entire document rather than a particular span (these are considered Feature Structures but not Annotations).

All feature structures, including annotations, are represented in the UIMA *Common Analysis Structure* (*CAS).* The CAS is the central data structure through which all UIMA components communicate. Included with the UIMA SDK is an easy-to-use, native Java interface to the CAS called the *JCas*. The JCas represents each feature structure as a Java object; the example feature structure from the previous

paragraph would be an instance of a Java class Person with getFullName() and setFullName() methods. Though the examples in this guide all use the JCas, it is also possible to directly access the underlying CAS system; for more information see *Chapter 23 **CAS Reference***.

The remainder of this chapter will refer to the analysis of text documents and the creation of annotations that are attached to spans of text in those documents. Keep in mind that the CAS can represent arbitrary types of feature structures, and feature structures can refer to other feature structures. For example, you can use the CAS to represent a parse tree for a document. Also, the artifact that you are analyzing need not be a text document.

This guide is organized as follows:

Getting Started is a tutorial with step-by-step instructions for how to develop and test a simple UIMA annotator.

Configuration and Logging discusses how to make your UIMA annotator configurable, and how it can write messages to the UIMA log file.

Building Aggregate Analysis Engines describes how annotators can be combined into aggregate analysis engines. It also describes how one annotator can make use of the analysis results produced by an annotator that has run previously.

Other examples

The UIMA SDK include several other examples you may find interesting, including

- SimpleTokenAndSentenceAnnotator – a simple tokenizer and sentence annotator.
- PersonTitleDBWriterCasConsumer – a sample CAS Consumer which populates a relational database with some annotations. It uses JDBC and in this example, hooks up with the Open Source Apache Derby database.

Additional Topics describes additional features of the UIMA SDK that may help you in building your own annotators and analysis engines.

Common Pitfalls contains some useful guidelines to help you ensure that your annotators will work correctly in any UIMA application.

This guide does not discuss how to build UIMA Applications, which are programs that use Analysis Engines, along with other components, e.g. a search engine, document store, and user interface, to deliver a complete package of functionality to an end-user. For information on application development, see *Chapter 6 **Application Developer's Guide***.

4.1 Getting Started

This section is a step-by-step tutorial that will get you started developing UIMA annotators. All of the files referred to by the examples in this chapter are in the docs/examples directory of the UIMA SDK. This directory is designed to be imported into your Eclipse workspace; see section 3.2 ***Setting up Eclipse to view Example Code*** for instructions on how to do this. Also you may wish to refer to the UIMA SDK JavaDocs located in the docs/api directory.

> **Note:** In Eclipse 3.1, if you highlight a UIMA class or method defined in the UIMA SDK JavaDocs, you can conveniently have Eclipse open the corresponding JavaDoc for that class or method in a browser, by pressing Shift + F2.

The example annotator that we are going to walk through will detect room numbers for rooms at the IBM T.J. Watson Research Center (where the UIMA SDK originated). There are two Watson buildings: Yorktown and Hawthorne, and each has its own pattern for room numbers. Here are some examples, together with their corresponding regular expression patterns:

Yorktown: 20-001, 31-206, 04-123 (Pattern: ##-[0-2]##)

Hawthorne: GN-K35, 1S-L07, 4N-B21 (Pattern: [G1-4][NS]-[A-Z]##)

There are several steps to develop and test a simple UIMA annotator.

1. Define the CAS types that the annotator will use.
2. Generate the Java classes for these types.
3. Write the actual annotator Java code.
4. Create the Analysis Engine descriptor.
5. Test the annotator.

These steps are discussed in the next sections.

4.1.1 Defining Types

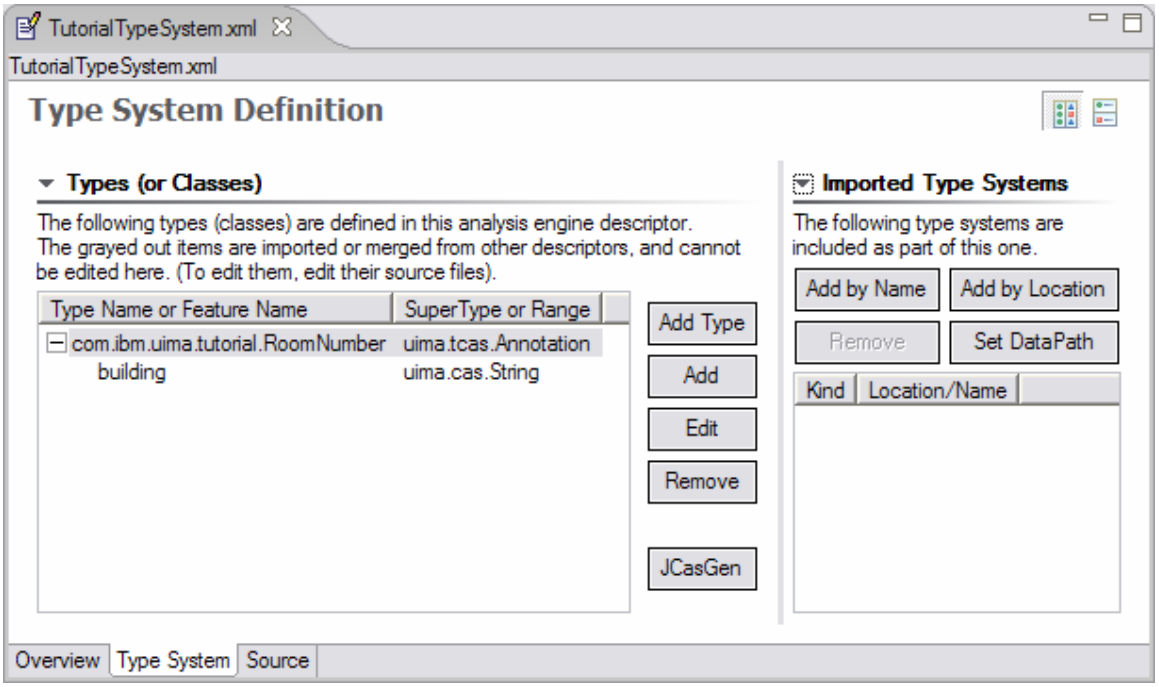
The first step in developing an annotator is to define the CAS Feature Structure types that it creates. This is done in an XML file called a *Type System Descriptor*. UIMA defines some basic built-in CAS types such as TOP, Integer, Float, String, IntegerArray, FloatArray, StringArray, FSArray, and Annotation. TOP is the root of the type system, analogous to Object in Java. FSArray is an array of Feature Structures (i.e. an array of instances of TOP).

UIMA includes an Eclipse plug-in that will help you edit Type System Descriptors, so if you are using Eclipse you will not need to worry about the details of the XML

syntax. See *Chapter 3* ***UIMA SDK Setup for Eclipse*** for instructions on setting up Eclipse and installing the plugin.

The Type System Descriptor for our annotator is located in the file `descriptors/tutorial/ex1/TutorialTypeSystem.xml`. (This and all other examples are located in the docs/examples directory of the UIMA SDK, which can be imported into an Eclipse project for your convenience, as described in 3.2 ***Setting up Eclipse to view Example Code***.)

In Eclipse, expand the uima_examples project in the Package Explorer view, and browse to the file `descriptors/tutorial/ex1/TutorialTypeSystem.xml`. Right-click on the file in the navigator and select Open With -> Component Descriptor Editor. Once the editor opens, click on the "Type System" tab at the bottom of the editor window. You should see a view such as the following:



Our annotator will need only one type – `com.ibm.uima.tutorial.RoomNumber`. (We use the same namespace conventions as are used for Java classes.) Just as in Java, types have supertypes. The supertype is listed in the second column of the left table. In this case our RoomNumber annotation extends from the built-in type `uima.tcas.Annotation`.

Descriptions can be included with types and features. In this example, there is a description associated with the `building` feature. To see it, hover the mouse over the feature.

The bottom tab labeled "Source" will show you the XML source file associated with this descriptor.

The built-in Annotation type declares two fields (called *Features* in CAS terminology) – begin and end. These features store the character offsets of the span of text to which the annotation refers. Our RoomNumber type will inherit these features from `com.tcas.Annotation`, its supertype; they are not visible in this view because inherited features are not shown. One additional feature, `building`, is declared. It takes a String as its value. Instead of String, we could have declared the range-type of our feature to be any other CAS type (defined or built-in).

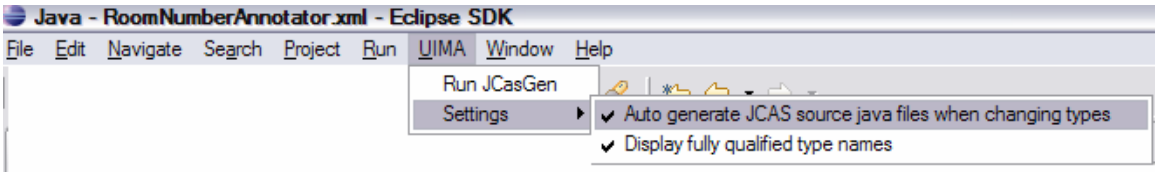
If you are not using Eclipse, if you need to edit the type system, do so using any XML or text editor, directly. The following is the actual XML representation of the Type System displayed above in the editor:

```
<?xml version="1.0" encoding="UTF-8" ?>
<typeSystemDescription
xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <name>TutorialTypeSystem</name>
  <description>Type System Definition for the tutorial examples - as of
Exercise 1</description>
  <vendor>IBM</vendor>
  <version>1.0</version>
  <types>
    <typeDescription>
      <name>com.ibm.uima.tutorial.RoomNumber</name>
      <description></description>
      <supertypeName>uima.tcas.Annotation</supertypeName>
      <features>
        <featureDescription>
        <name>building</name>
        <description>Building containing this room</description>
        <rangeTypeName>uima.cas.String</rangeTypeName>
      </featureDescription>
    </features>
   </typeDescription>
</types>
</typeSystemDescription>
```

4.1.2 Generating Java Source Files for CAS Types

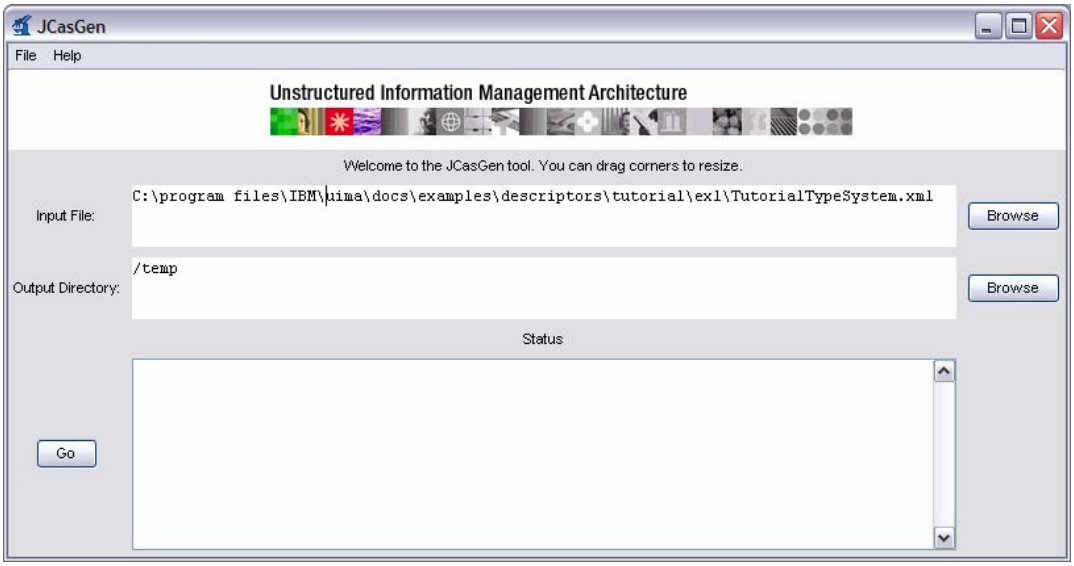
When you save a descriptor that you have modified, the Component Descriptor Editor will automatically generate Java classes corresponding to the types that are defined in that descriptor (unless this has been disabled), using a utility called JCasGen. These Java classes will have the same name (including package) as the CAS types, and will have get and set methods for each of the features that you have defined.

This feature is enabled/disabled using the UIMA menu pulldown (or the Eclipse Preferences – UIMA). If automatic running of JCasGen is not happening, please make sure the option is checked:



The Java class for the example com.ibm.uima.tutorial.RoomNumber type can be found in `src/com/ibm/uima/tutorial/RoomNumber.java`. You will see how to use these generated classes in the next section.

If you are not using the Component Descriptor Editor, you will need to generate these Java classes by using the *JCasGen* tool. JCasGen reads a Type System Descriptor XML file and generates the corresponding Java classes that you can then use in your annotator code. To launch JCasGen, simply execute the jcasgen shell script located in the bin directory of the UIMA SDK. This should launch a GUI that looks something like this:



Use the "Browse" buttons to select your input file (TutorialTypeSystem.xml) and output directory (the root of the source tree into which you want the generated files placed). Then click the "Go" button. Assuming no errors in the Type System Descriptor, new Java source files should be generated under the specified output directory.

There are some additional options to choose from when running JCasGen; please refer to the *Chapter 16* ***JCasGen User Guide***, for details.

4.1.3 Developing Your Annotator Code

Annotator implementations all implement a standard interface, having several methods, the most important of which are:

- `initialize`,
- `process`, and
- `destroy`.

`initialize` is called by the framework once when it first creates the annotator. `process` is called once per item being processed. `destroy` may be called by the application when it is done. There is a default implementation of this interface for annotators using the JCas, called JTextAnnotator_ImplBase, which has implementations of all required methods except for the process method.

Our annotator class extends the JTextAnnotator_ImplBase; most annotators that use the JCas will extend from this class, so they only have to implement the process method. Even though this class name has the word "Text" in it, it is not restricted to handling just text; see Chapter 7 ***Developing Applications using Multiple Subjects of Analysis*** on page 7-158.

Annotators are not required to extend from the JTextAnnotator_ImplBase class; they may instead directly implement the JTextAnnotator interface, and provide all method implementations themselves. This allows you to have your annotator inherit from some other superclass if necessary. If you would like to do this, see the JavaDocs for JTextAnnotator for descriptions of the methods you must implement.

Annotator classes need to be public and have public, 0-argument constructors, so that they can be instantiated by the framework[2].

The class definition for our RoomNumberAnnotator implements the process method, and is shown here. You can find the source for this in the `uima_examples/src/com/ibm/uima/tutorial/ex1/RoomNumberAnnotator.java`. Note: In Eclipse, in the "Package Explorer" view, this will appear by default in the project `uima_examples`, in the folder `src`, in the package `com.ibm.uima.tutorial.ex1`. In Eclipse, open the RoomNumberAnnotator.java in the uima_examples project, under the src directory.

```
package com.ibm.uima.tutorial.ex1;

import java.util.regex.Matcher;
```

[2] Although Java classes in which you do not define any constructor will, by default, have a 0-argument constructor that doesn't do anything, a class in which you have defined at least one constructor does not get a default 0-argument constructor.

```
import java.util.regex.Pattern;

import com.ibm.uima.analysis_engine.ResultSpecification;
import
com.ibm.uima.analysis_engine.annotator.AnnotatorConfigurationException;
import com.ibm.uima.analysis_engine.annotator.AnnotatorContext;
import
com.ibm.uima.analysis_engine.annotator.AnnotatorInitializationException;
import com.ibm.uima.analysis_engine.annotator.AnnotatorProcessException;
import com.ibm.uima.analysis_engine.annotator.JTextAnnotator_ImplBase;
import com.ibm.uima.jcas.impl.JCas;
import com.ibm.uima.tutorial.RoomNumber;

/**
 * Example annotator that detects room numbers using Java 1.4 regular
 * expressions.
 */
public class RoomNumberAnnotator extends JTextAnnotator_ImplBase
{
  private Pattern mYorktownPattern =
    Pattern.compile("\\b[0-4]\\d-[0-2]\\d\\d\\b");
  private Pattern mHawthornePattern =
    Pattern.compile("\\b[G1-4][NS]-[A-Z]\\d\\d\\b");;

  /**
   * @see JTextAnnotator#process(JCas,ResultSpecification)
   */
  public void process(JCas aJCas, ResultSpecification aResultSpec)
    throws AnnotatorProcessException
  {
    // Discussed Later
  }
}
```

The two Java class fields, mYorktownPattern and mHawthornePattern, hold regular expressions that will be used in the process method. Note that these two fields are part of the Java implementation of the annotator code, and not a part of the CAS type system. We are using the regular expression facility that is built into Java 1.4. It is not critical that you know the details of how this works, but if you are curious the details can be found in the Java API docs for the java.util.regex package.

The only method that we are required to implement is process. This method is typically called once for each document that is being analyzed. This method takes two arguments. The JCas holds the document to be analyzed and all of the analysis results. We'll ignore the ResultSpecification for now; its use is not required.

```
  /**
   * @see JTextAnnotator#process(JCas,ResultSpecification)
   */
  public void process(JCas aJCas, ResultSpecification aResultSpec)
    throws AnnotatorProcessException
  {
    //get document text
```

```
    String docText = aJCas.getDocumentText();
    //search for Yorktown room numbers
    Matcher matcher = mYorktownPattern.matcher(docText);
    int pos = 0;
    while (matcher.find(pos))
    {
      //found one - creation annotation
      RoomNumber annotation = new RoomNumber(aJCas);
      annotation.setBegin(matcher.start());
      annotation.setEnd(matcher.end());
      annotation.setBuilding("Yorktown");
      annotation.addToIndexes();
      pos = matcher.end();
    }
    //search for Hawthorne room numbers
    matcher = mHawthornePattern.matcher(docText);
    pos = 0;
    while (matcher.find(pos))
    {
      //found one - creation annotation
      RoomNumber annotation = new RoomNumber(aJCas);
      annotation.setBegin(matcher.start());
      annotation.setEnd(matcher.end());
      annotation.setBuilding("Hawthorne");
      annotation.addToIndexes();
      pos = matcher.end();
    }
  }
```

The Matcher class is part of the java.util.regex package and is used to find the room numbers in the document text. When we find one, recording the annotation is as simple as creating a new Java object and calling some set methods:

```
RoomNumber annotation = new RoomNumber(aJCas);
annotation.setBegin(matcher.start());
annotation.setEnd(matcher.end());
annotation.setBuilding("Yorktown");
```

This RoomNumber class was generated from the type system description by the Component Descriptor Editor or the JCasGen tool, as discussed in the previous section.

Finally, we call annotation.addToIndexes() to add the new annotation to the indexes maintained in the CAS. By default, the CAS implementation used for analysis of text documents keeps an index of all annotations in their order from beginning to end of the document. Subsequent annotators or applications use the indexes to iterate over the annotations. It is also possible to define your own custom indexes in the CAS (see *Chapter 23 **CAS Reference*** for details).

Note: If you don't add the instance to the indexes, it cannot be retrieved by down-stream annotators, using the indexes.

We're almost ready to test the RoomNumberAnnotator. There is just one more step remaining.

4.1.4 Creating the XML Descriptor

The UIMA architecture requires that descriptive information about an annotator be represented in an XML file and provided along with the annotator class file(s) to the UIMA framework at run time. This XML file is called an *Analysis Engine Descriptor*. The descriptor includes:

- 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

The *Component Descriptor Editor* plugin, which we previously used to edit the Type System descriptor, can also be used to edit Analysis Engine Descriptors.

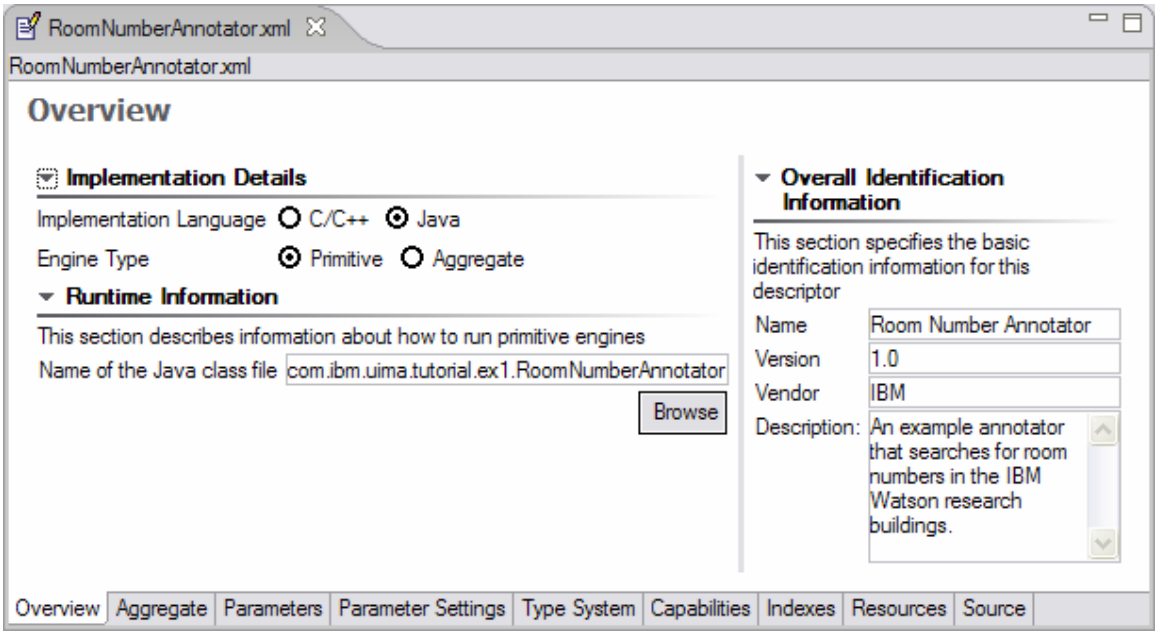
A descriptor for our RoomNumberAnnotator is provided with the UIMA distribution under the name `descriptors/tutorial/ex1/RoomNumberAnnotator.xml`. To edit it in Eclipse, right-click on that file in the navigator and select Open With → Component Descriptor Editor.

> Eclipse tip: You can double click on the tab at the top of the Component Descriptor Editor's window identifying the currently selected editor, and the window will "Maximize". Double click it again to restore the original size.

If you are not using Eclipse, you will need to edit Analysis Engine descriptors manually. See ***Introduction to Analysis Engine Descriptor XML Syntax*** on page 4-95 for an introduction to the Analysis Engine descriptor XML syntax. The remainder of this section assumes you are using the Component Descriptor Editor plug-in to edit the Analysis Engine descriptor.

The Component Descriptor Editor consists of several tabbed pages; we will only need to use a few of them here. For more information on using this editor, see *Chapter 9* ***Component Descriptor Editor User's Guide***.

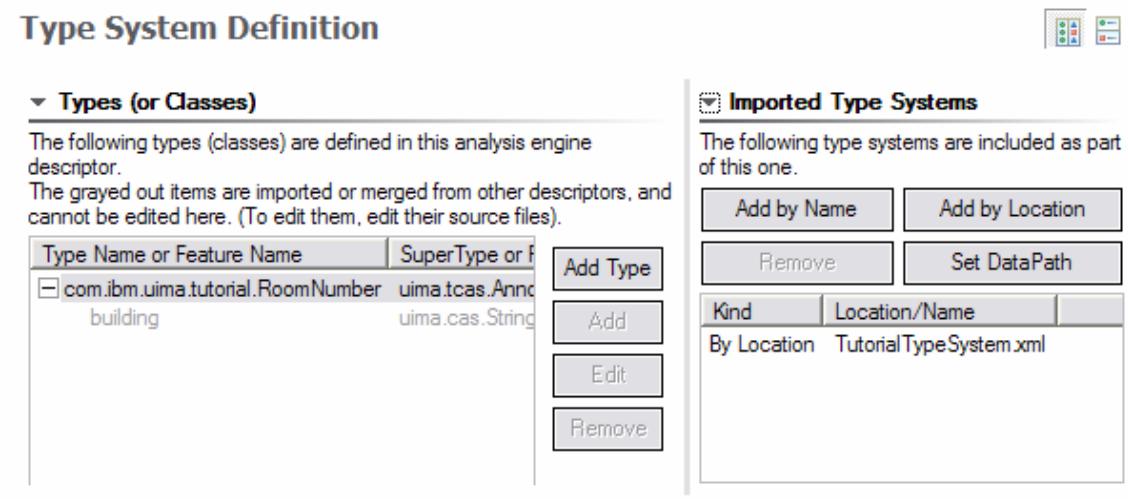
The initial page of the Component Descriptor Editor is the Overview page, which appears as follows



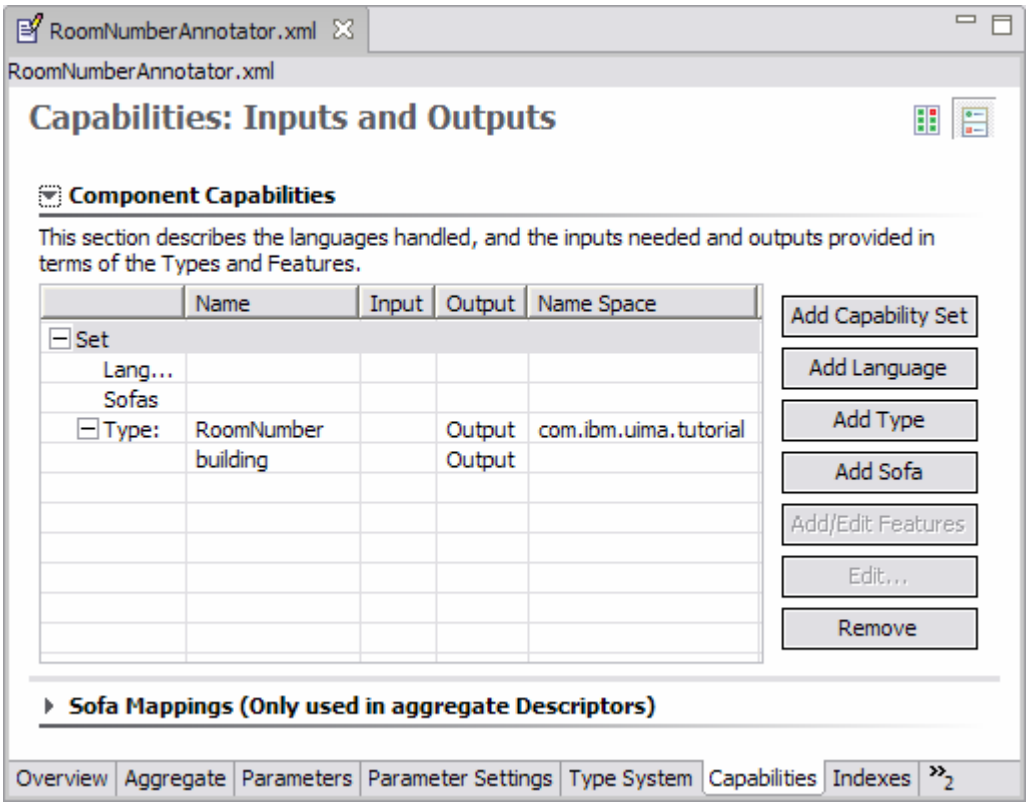
This presents an overview of the RoomNumberAnnotator Analysis Engine (AE). The left side of the page shows that this descriptor is for a *Primitive* AE (meaning it consists of a single annotator), and that the annotator code is developed in Java. Also, it specifies the Java class that implements our logic (the code which was discussed in the previous section). Finally, on the right side of the page are listed some descriptive attributes of our annotator.

The other two pages that need to be filled out are the Type System page and the Capabilities page. You can switch to these pages using the tabs at the bottom of the Component Descriptor Editor. In the tutorial, these are already filled out for you.

The RoomNumberAnnotator will be using the TutorialTypeSystem we looked at in Section *4.1.1* ***Defining Types***. To specify this, we add this type system to the Analysis Engine's list of Imported Type Systems, using the Type System page's right side panel, as shown here:



On the Capabilities page, we define our annotator's inputs and outputs, in terms of the types in the type system. The Capabilities page is shown below:



Capabilities come in sets; here we're just using one set. The RoomNumberAnnotator is very simple. It requires no input types, as it operates directly on the document text -- which is supplied as a part of the CAS initialization (and which is always assumed to be present). It produces only one output type

(RoomNumber), and it sets the value of the building feature on that type. This is all represented on the Capabilities page.

The Capabilities page has two other parts for specifying languages and Sofas. The languages section allows you to specify which languages your Analysis Engine supports. The RoomNumberAnnotator happens to be language-independent, so we can leave this blank. The Sofas section allows you to specify the names of additional subjects of analysis. This capability and the Sofa Mappings at the bottom are advanced topics, described in Chapter 7 ***Developing Applications using Multiple Subjects of Analysis*** on page 7-158

This is all of the information we need to provide for a simple annotator. If you want to peek at the XML that this tool saves you from having to write, click on the "Source" tab at the bottom to view the generated XML.

4.1.5 Testing Your Annotator

Having developed an annotator, we need a way to try it out on some example documents. The UIMA SDK includes a tool called the Document Analyzer that will allow us to do this. To run the Document Analyzer, execute the documentAnalyzer shell script that is in the bin directory of your UIMA SDK installation, or, if you are using the example Eclipse project, execute the "UIMA Document Analyzer" run configuration supplied with that project. (To do this, click on the menu bar Run > Run ... > and under Java Applications in the left box, click on UIMA Document Analyzer.)

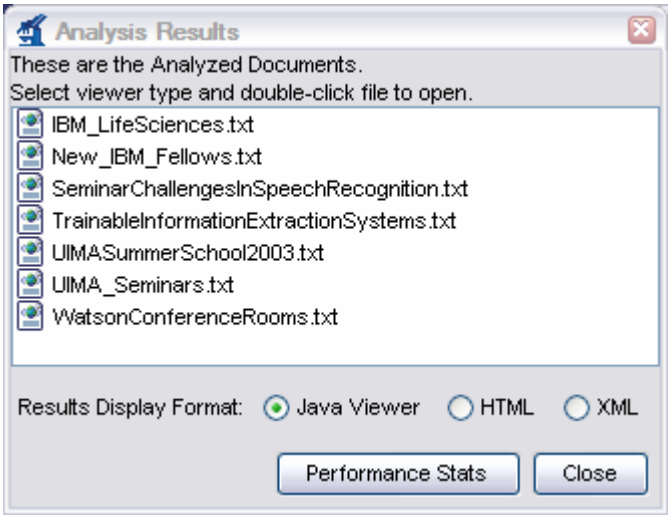
You should see a screen that looks like this:

There are six options on this screen:

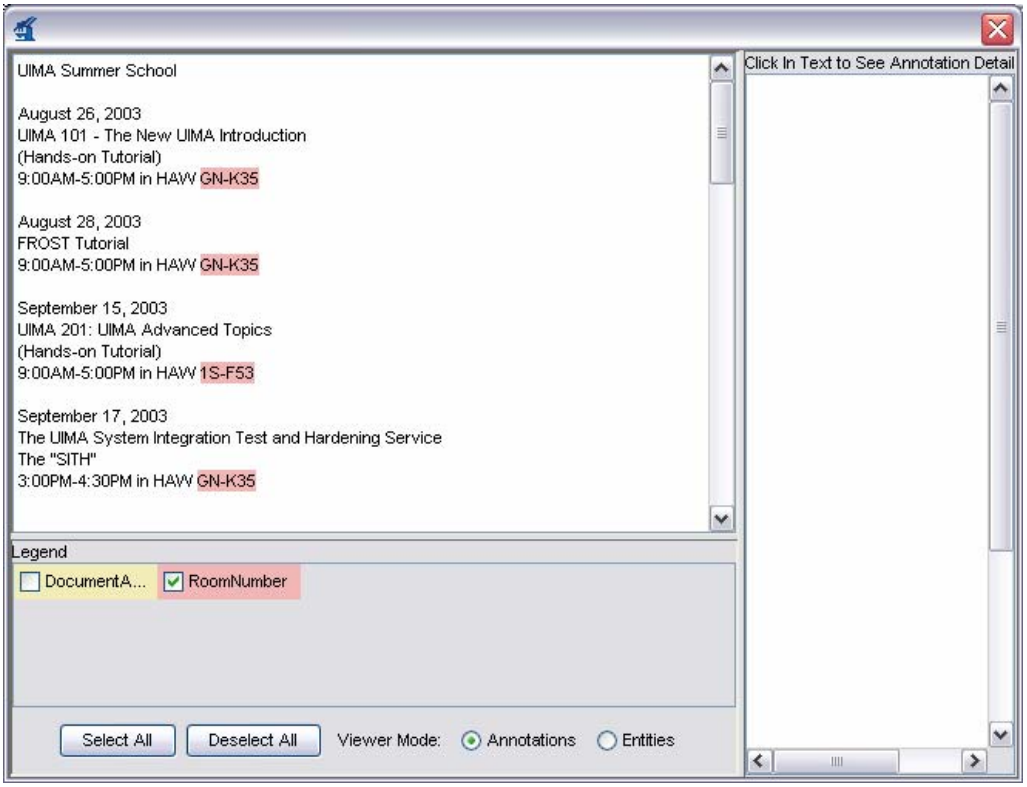
1. Directory containing documents to analyze
2. Directory where analysis results will be written
3. The XML descriptor for the Analysis Engine (TAE) you want to run
4. (Optional) an XML tag, within the input documents, that contains the text to be analyzed. For example, the value TEXT would cause the TAE to only analyze the portion of the document enclosed within <TEXT>...</TEXT> tags.
5. Language of the document
6. Character encoding

Use the Browse button next to the 3rd item to set the "Location of TAE XML Descriptor" field to the descriptor we've just been discussing – `uima/docs/examples/descriptors/tutorial/ex1/RoomNumberAnnotator.xml`. Set the other fields to the values shown in the screen shot above (which should be the default values if this is the first time you've run the Document Analyzer). Then click the "Run" button to start processing.

When processing completes, an "Analysis Results" window should appear.



Make sure "Java Viewer" is selected as the Results Display Format, and **double-click** on the document UIMASummerSchool2003.txt to view the annotations that were discovered. The view should look something like this:



You can click the mouse on one of the highlighted annotations to see a list of all its features in the frame on the right.

> **Note:** The legend will only show those types which have at least one instance in the CAS, and are declared as outputs in the capabilities section of the descriptor (see **Creating the XML Descriptor** on page 4-60).

You can use the DocumentAnalyzer to test any UIMA annotator – just make sure that the annotator's classes are in the class path.

4.2 Configuration and Logging

4.2.1 Configuration Parameters

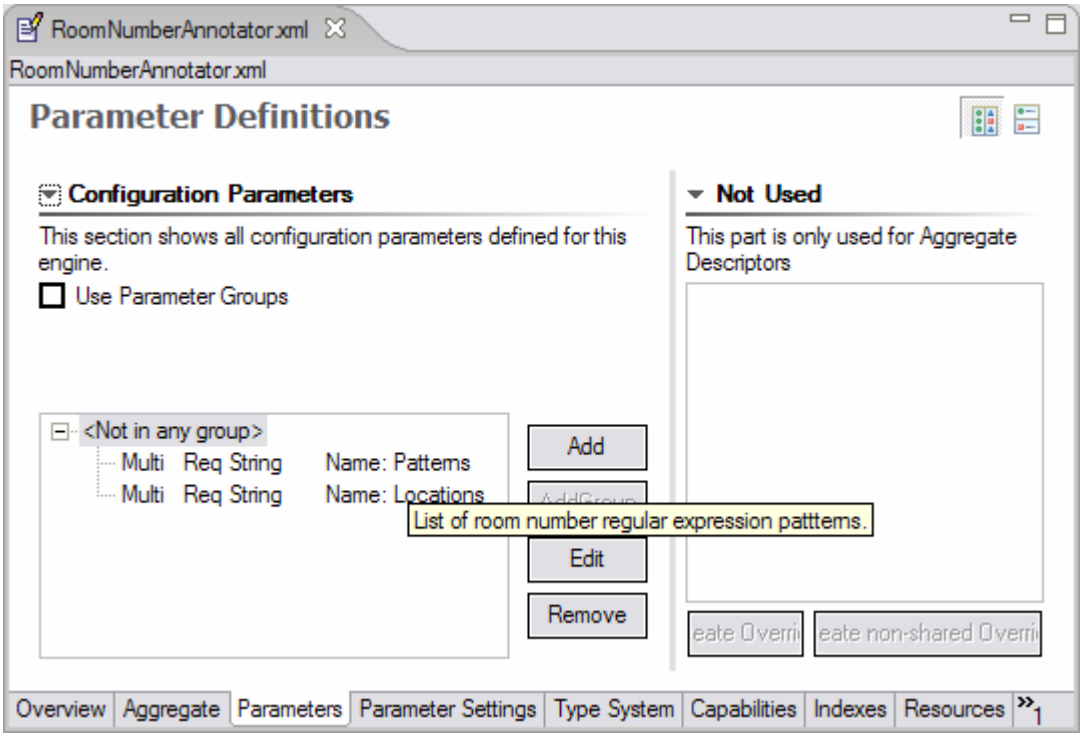
The example RoomNumberAnnotator from the previous section used hardcoded regular expressions and location names, which is obviously not very flexible. For example, there is actually a third Watson building – Hawthorne II – whose room numbers are not detected by our annotator. Rather than add a new hardcoded regular expression, a better solution is to use configuration parameters.

UIMA allows annotators to declare configuration parameters in their descriptors. The descriptor also specifies default values for the parameters, though these can be overridden at runtime.

Declaring Parameters in the Descriptor

The example descriptor `descriptors/tutorial/ex2/RoomNumberAnnotator.xml` is the same as the descriptor from the previous section except that information has been filled in for the Parameters and Parameter Settings pages of the Component Descriptor Editor.

First, in Eclipse, open example 2's RoomNumberAnnotator in the Component Descriptor Editor, and then go to the Parameters page (click on the parameters tab at the bottom of the window), which is shown below:

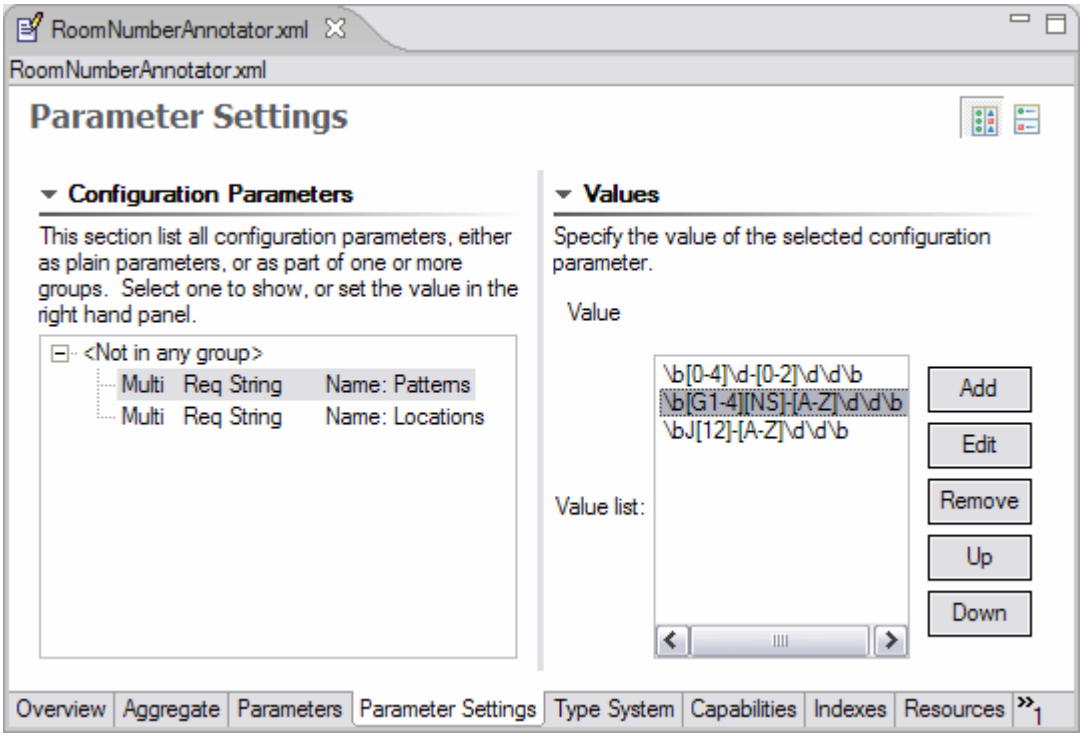


Two parameters – Patterns and Locations -- have been declared. In this screen shot, the mouse (not shown) is hovering over Patterns to show its description in the small popup window. Every parameter has the following information associated with it:

- name – the name by which the annotator code refers to the parameter
- description – a natural language description of the intent of the parameter
- type – the data type of the parameter's value – must be one of String, Integer, Float, or Boolean.
- multiValued – true if the parameter can take multiple-values (an array), false if the parameter takes only a single value. Shown above as `Multi`.
- mandatory – true if a value must be provided for the parameter. Shown above as `Req` (for required).

Both of our parameters are mandatory and accept an array of Strings as their value.

Next, default values are assigned to the parameters on the Parameter Settings page:



Here the "Patterns" parameter is selected, and the right pane shows the list of values for this parameter, in this case the regular expressions that match rooms in each of the IBM T.J. Watson Research Center buildings. Notice the third pattern is new, for matching the style of room numbers in the third building, which has room numbers such as `J2-A11`.

Accessing Parameter Values from the Annotator Code

The class `com.ibm.uima.tutorial.ex2.RoomNumberAnnotator` has overridden the initialize method. The initialize method is called by the UIMA framework when the annotator is instantiated, so it is a good place to read configuration parameter values. The default initialize method does nothing with configuration parameters, so you have to override it. To see the code in Eclipse, switch to the src folder, and open `com.ibm.uima.tutorial.ex2`. Here is the method body:

```
  /**
   * @see BaseAnnotator#initialize(AnnotatorContext)
   */
  public void initialize(AnnotatorContext aContext)
    throws AnnotatorInitializationException,
AnnotatorConfigurationException
  {
    // invoke the standard initialization
```

```
    // This saves the value of aContext in a field and makes
    // it available via the getContext() method of the superclass
    super.initialize(aContext);
    try
    {
      //Get config. parameter values
      String[] patternStrings =
        (String[])aContext.getConfigParameterValue("Patterns");
      mLocations =
        (String[])aContext.getConfigParameterValue("Locations");

      //compile regular expressions

      mPatterns = new Pattern[patternStrings.length];
      for (int i = 0; i < patternStrings.length; i++)
      {
        mPatterns[i] = Pattern.compile(patternStrings[i]);
      }
    }

    catch(AnnotatorContextException e)
    {
      throw new AnnotatorInitializationException(e);
    }
  }
```

The first two lines inside the try block are where the configuration parameter values are retrieved. Configuration parameter values are accessed through the AnnotatorContext. As you will see in subsequent sections of this chapter, the AnnotatorContext is the annotator's access point for all of the facilities provided by the UIMA framework – for example logging and external resource access.

The AnnotatorContext.getConfigParameterValue method takes the name of the parameter as an argument; this must match one of the parameters declared in the descriptor. The return value of this method is Object, so it is up to the annotator to cast it to the appropriate type, String[] in this case.

If there is a problem retrieving the parameter values, the AnnotatorContext could throw an AnnotatorContextException. Generally annotators would just catch this exception and rethrow it as an AnnotatorInitializationException, which is what our example annotator does.

To see the configuration parameters working, run the Document Analyzer application and select the descriptor `docs/examples/descriptors/tutorial/ex2/RoomNumberAnnotator.xml`. In the example document `WatsonConferenceRooms.txt`, you should see some examples of Hawthorne II room numbers that would not have been detected by the ex1 version of RoomNumberAnnotator.

Supporting Reconfiguration

If you take a look at the JavaDocs (located in the docs/api directory) for `com.ibm.uima.analysis_engine.Annotator.BaseAnnotator` (which our annotator implements indirectly through JTextAnnotator_ImplBase), you will see that there is a reconfigure() method, which is called by the containing application through the UIMA framework, if the configuration parameter values are changed.

The JTextAnnotator_ImplBase class provides a default implementation that just calls the annotator's destroy method followed by its initialize method. This works fine for our annotator. The only situation in which you might want to override the default reconfigure() is if your annotator has very expensive initialization logic, and you don't want to reinitialize everything if just one configuration parameter has changed. In that case, you can provide a more intelligent implementation of reconfigure() for your annotator.

Configuration Parameter Groups

For annotators with many sets of configuration parameters, UIMA supports organizing them into groups. It is possible to define a parameter with the same name in multiple groups; one common use for this is for annotators that can process documents in several languages and which want to have different parameter settings for the different languages.

The syntax for defining parameter groups in your descriptor is fairly straightforward – see *Chapter 20* for details. Values of parameters defined within groups are accessed through the two-argument version of `AnnotatorContext.getConfigParameterValue`, which takes both the group name and the parameter name as its arguments.

4.2.2 Logging

The UIMA SDK provides a logging facility, which is very similar to the java.util.logging.Logger class that was introduced in Java 1.4.

In the Java architecture, each logger instance is associated with a name. By convention, this name is often the fully qualified class name of the component issuing the logging call. The name can be referenced in a configuration file when specifying which kinds of log messages to actually log, and where they should go.

The UIMA framework supports this convention using the `AnnotatorContext` object. If you access a logger instance using `getContext().getLogger()` within an Annotator, the logger name will be the fully qualified name of the Annotator implementation class.

Here is an example from the process method of `com.ibm.uima.tutorial.ex2.RoomNumberAnnotator`:

```
getContext().getLogger().log(Level.FINEST,"Found: " + annotation);
```

The first argument to the log method is the level of the log output. Here, a value of FINEST indicates that this is a highly-detailed tracing message. While useful for debugging, it is likely that real applications will not output log messages at this level, in order to improve their performance. Other defined levels, from lowest to highest importance, are FINER, FINE, CONFIG, INFO, WARNING, and SEVERE.

If no logging configuration file is provided (see next section), the Java Virtual Machine defaults would be used, which typically set the level to INFO and higher messages, and direct output to the console.

If you specify the standard UIMA SDK `Logger.properties`, the output will be directed to a file named uima.log, in the current working directory (often the "project" directory when running from Eclipse, for instance).

> Eclipse Note: The uima.log file, if written into the Eclipse workspace in the project uima_examples, for example, may not appear in the Eclipse package explorer view until you right-click the uima_examples project with the mouse, and select "Refresh". This operation refreshes the Eclipse display to conform to what may have changed on the file system.

Specifying the Logging Configuration

The standard UIMA logger uses the underlying Java 1.4 logging mechanism. You can use the APIs that come with that to configure the logging. In addition, the Java 1.4 logging initialization look for a Java System Property named `java.util.logging.config.file` and if found, will use the value of this property as the name of a standard "properties" file, for setting the logging level. Please refer to the Java 1.4. documentation for more information on the format and use of this file.

Two sample logging specification property files can be found in the UIMA_HOME directory: `Logger.properties`, and `FileConsoleLogger.properties`. These specify the same logging, except the first logs just to a file, while the second logs both to a file and to the console. You can edit these files, or create additional ones, as described below, to change the logging behavior.

When running your own Java application, you can specify the location of the logging configuration file on your Java command line by setting the Java system property `java.util.logging.config.file` to be the logging configuration filename. This file specification can be either absolute or relative to the working directory. For example:

```
java "-Djava.util.logging.config.file=c:/Program
Files/IBM/uima/Logger.properties"
```

Note: In a shell script, you can use environment variables such as UIMA_HOME if convenient.

Setting Logging Levels

Within the logging control file, the default global logging level specifies which kinds of events are logged across all loggers. For any given facility this global level can be overridden by a facility specific level. Multiple handlers are supported. This allows messages to be directed to a log file, as well as to a "console". Note that the ConsoleHandler also has a separate level setting to limit messages printed to the console. For example:

```
.level= INFO
```

The properties file can change where the log is written, as well.

Facility specific properties allow different logging for each class, as well. For example, to set the com.xyz.foo logger to only log SEVERE messages:

```
com.xyz.foo.level = SEVERE
```

If you have a sample annotator in the package com.ibm.uima.SampleAnnotator you can set the log level by specifying:

```
com.ibm.uima.SampleAnnotator.level = ALL
```

There are other logging controls; for a full discussion, please read the contents of the `Logger.properties` file and the Java specification for logging in Java 1.4.

Format of logging output

The logging output is formatted by handlers specified in the properties file for configuring logging, described above. The default formatter that comes with the UIMA SDK formats logging output as follows:

```
Timestamp - threadID: sourceInfo: Message level: message
```

Here's an example:

```
7/12/04 2:15:35 PM - 10: com.ibm.uima.util.TestClass.main(62): INFO: You are
not logged in!
```

Meaning of the logging severity levels

These levels are defined by the Java logging framework, which was incorporated into Java as of the 1.4 release level. The levels are defined in the JavaDocs for java.util.logging.Level, and include both logging and tracing levels:

- OFF is a special level that can be used to turn off logging.
- ALL indicates that all messages should be logged.
- CONFIG is a message level for configuration messages. These would typically occur once (during configuration) in methods like `initialize()`.
- INFO is a message level for informational messages, for example, connected to server IP: 192.168.120.12
- WARNING is a message level indicating a potential problem.
- SEVERE is a message level indicating a serious failure.

Tracing levels, typically used for debugging:

- FINE is a message level providing tracing information, typically at a collection level (messages occurring once per collection).
- FINER indicates a fairly detailed tracing message, typically at a document level (once per document).
- FINEST indicates a highly detailed tracing message.

Using the logger outside of an annotator

An application using UIMA may want to log its messages using the same logging framework. This can be done by getting a reference to the UIMA logger, as follows:

```
Logger logger = UIMAFramework.getLogger(TestClass.class);
```

The optional class argument allows filtering by class (if the log handler supports this). If not specified, the name of the returned logger instance is "com.ibm.uima".

4.3 Building Aggregate Analysis Engines

4.3.1 Combining Annotators

The UIMA SDK makes it very easy to combine any sequence of Analysis Engines to form an *Aggregate Analysis Engine*. This is done through an XML descriptor; no Java code is required!

If you go to the `docs/examples/descriptors/tutorial/ex3` folder (in Eclipse, it's in your uima_examples project, under the `descriptors/tutorial/ex3` folder), you will find a descriptor for a TutorialDateTime annotator. This annotator detects dates and times (and also sentences and words). To see what this annotator can do, try it out using the Document Analyzer. If you are curious as to how this annotator works,

the source code is included, but it is not necessary to understand the code at this time.

We are going to combine the TutorialDateTime annotator with the RoomNumberAnnotator to create an aggregate Analysis Engine. This is illustrated in Figure 8.

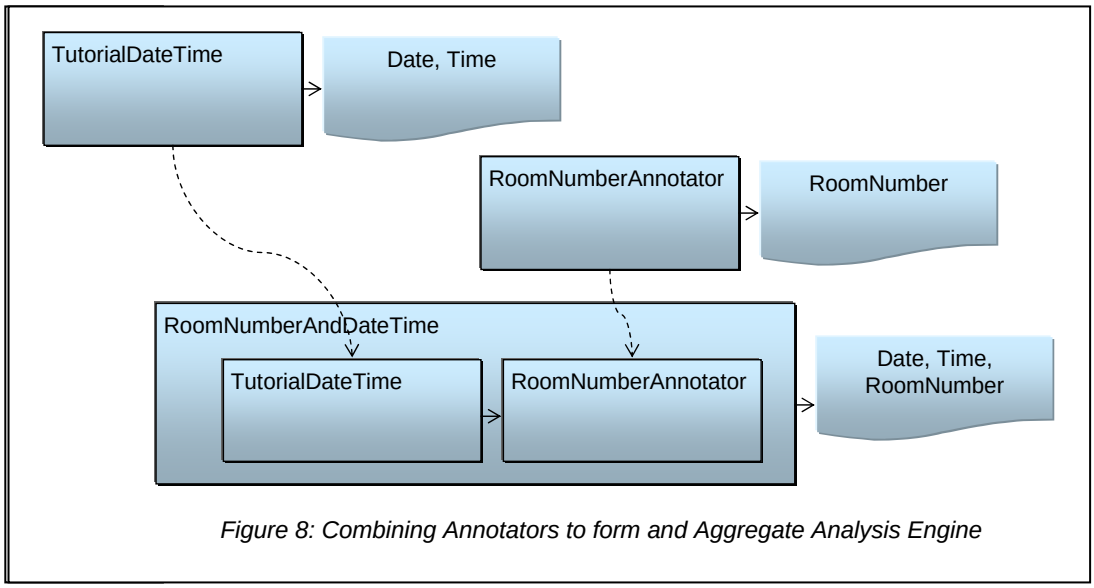
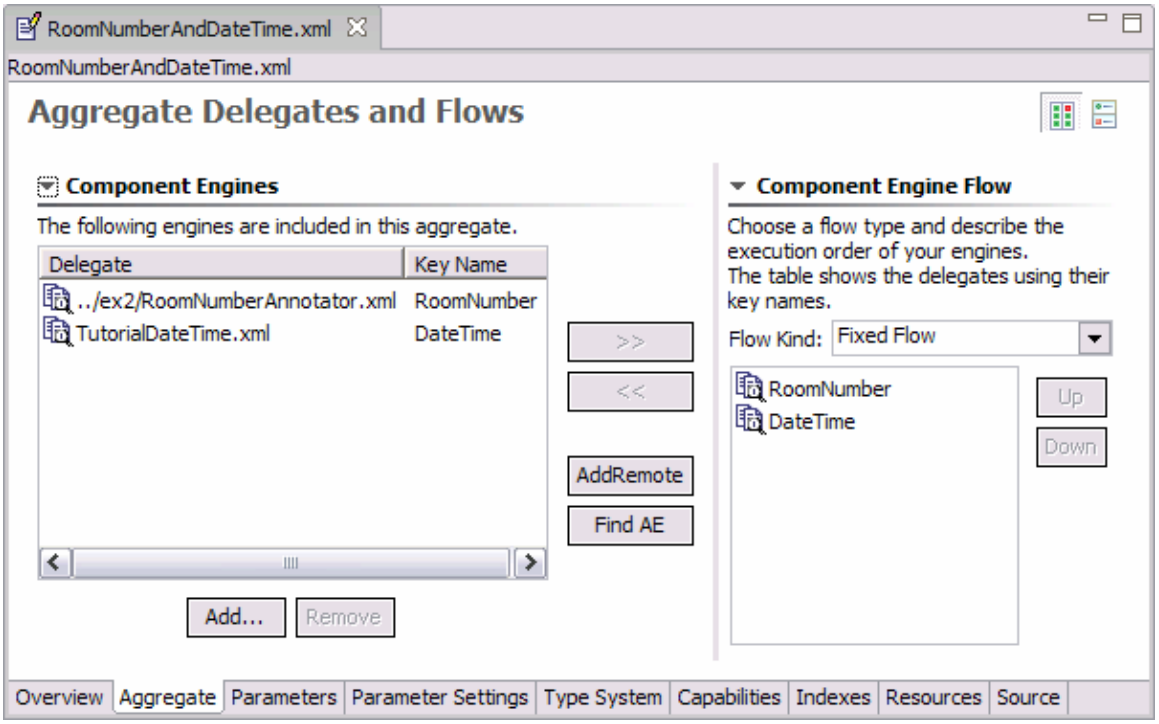


Figure 8: Combining Annotators to form and Aggregate Analysis Engine

The descriptor that does this is named `RoomNumberAndDateTime.xml`, which you can open in the Component Descriptor Editor plug-in. This is in the uima_examples project in the folder `descriptors/tutorial/ex3`.

The "Aggregate" page of the Component Descriptor Editor is used to define which components make up the aggregate. A screen shot is shown below. (If you are not using Eclipse, see *Section 4.8* ***Introduction to Analysis Engine Descriptor XML Syntax*** for the actual XML syntax for Aggregate Analysis Engine Descriptors.)



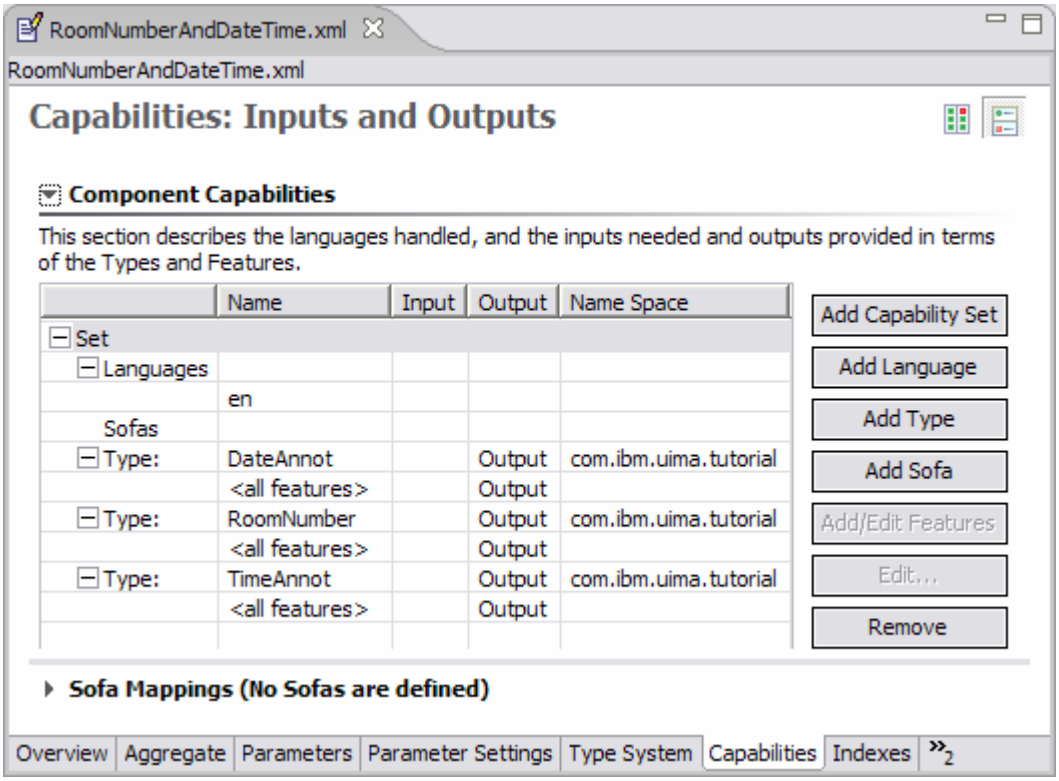
On the left side of the screen is the list of component engines that make up the aggregate – in this case, the TutorialDateTime annotator and the RoomNumberAnnotator. To add a component, you can click the "Add" button and browse to its descriptor. You can also click the "Find AE" button and search for an Analysis Engine in your Eclipse workspace.

> **Note:** The "AddRemote" button is used for adding components which run remotely (for example, on another machine using a remote networking connection). This capability is described in section 6.6.3 ***How to Call a UIMA Service*** on page 6-151.

The order of the components in the left pane does not imply an order of execution. The order of execution, or "flow" is determined in the "Component Engine Flow" section on the right. UIMA supports different types of algorithms (possibly dynamic) for determining the flow. Here we pick the simplest: `FixedFlow`. We have chosen to have the RoomNumberAnnotator execute first, although in this case it doesn't really matter, since the RoomNumber and DateTime annotators do not have any dependencies on one another.

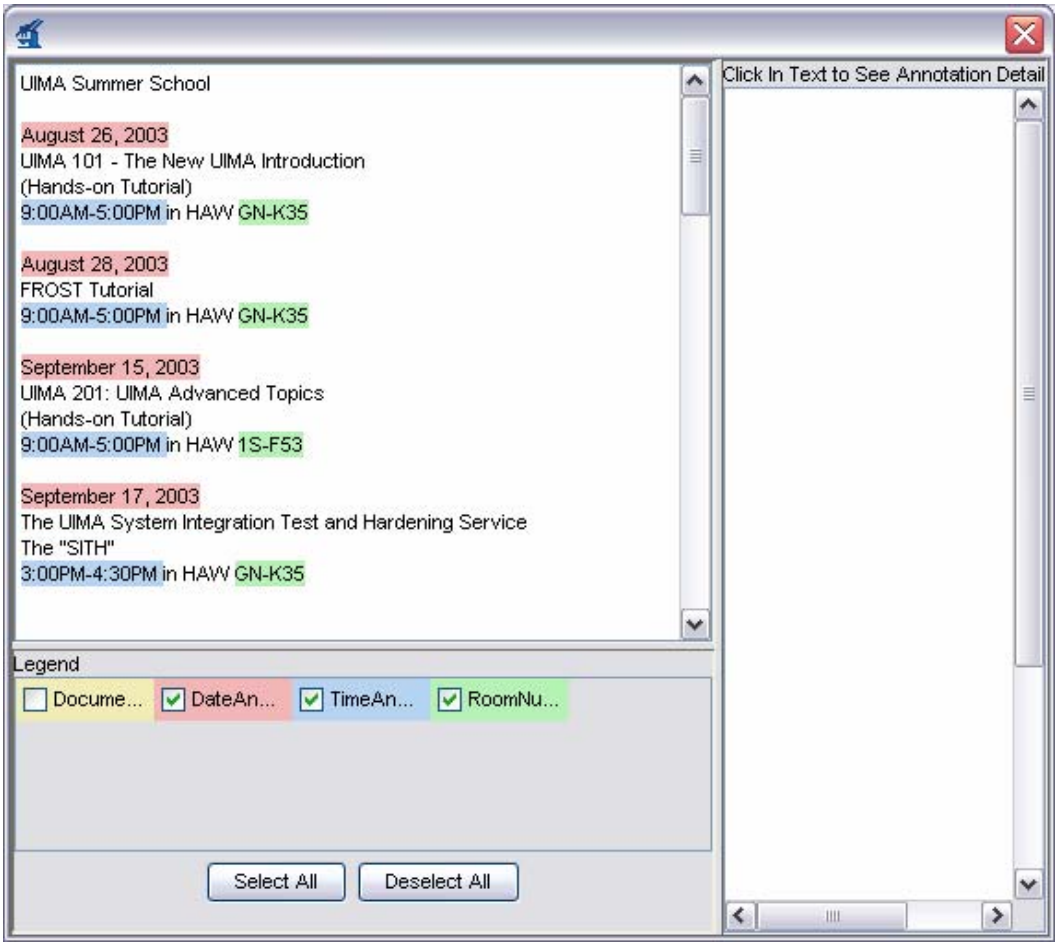
If you look at the "Type System" page of the Component Descriptor Editor, you will see that it displays the type system but is not editable. The Type System of an Aggregate Analysis Engine is automatically computed by merging the Type Systems of each of its components.

The Capabilities page is where you explicitly declare the aggregate Analysis Engine's inputs and outputs. Sofas and Languages are described later.



Note that it is not automatically assumed that all outputs of each component Analysis Engine (AE) are passed through as outputs of the aggregate AE. In this case, for example, we have decided to suppress the Word and Sentence annotations that are produced by the TutorialDateTime annotator.

You can run this AE using the Document Analyzer in the same way that you run any other AE. Just select the `docs/examples/descriptors/tutorial/ex3/RoomNumberAndDateTime.xml` descriptor and click the Run button. You should see that RoomNumbers, Dates, and Times are all shown but that Words and Sentences are not:



4.3.2 Aggregate Engines can also contain CAS Consumers

In addition to aggregating Analysis Engines, Aggregates can also contain CAS Consumers (see ***Developing CAS Consumers*** on page ***5-119***), or even a mixture of these components. The UIMA Examples has an example of an Aggregate which contains both an analysis engine and a CAS consumer, in `docs/examples/descriptors/MixedAggregate.xml`.

4.3.3 Reading the Results of Previous Annotators

So far, we have been looking at annotators that look directly at the document text. However, annotators can also use the results of other annotators. One useful thing we can do at this point is look for the co-occurrence of a Date, a RoomNumber, and two Times – and annotate that as a Meeting.

The JCas maintains *indexes* of annotations, and from an index you can obtain an iterator that allows you to step through all annotations of a particular type. Here's some example code that would iterate over all of the TimeAnnot annotations in the JCas:

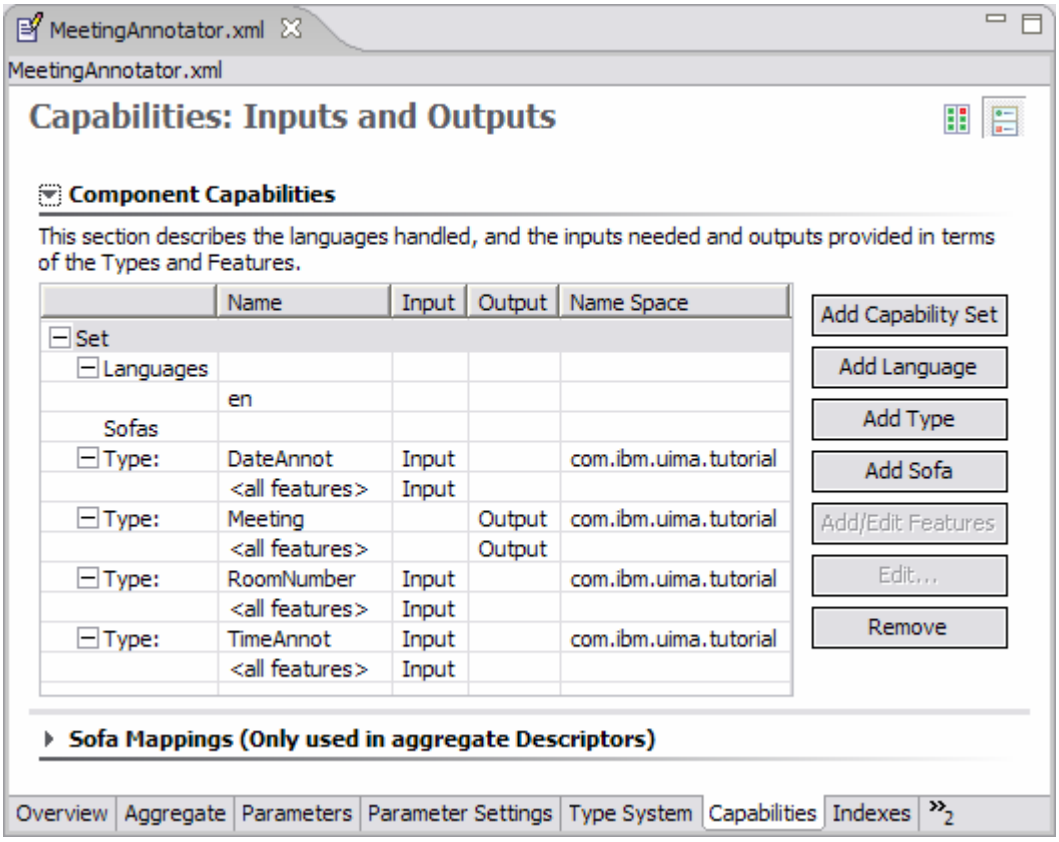
```
JFSIndexRepository indexes = aJCas.getJFSIndexRepository();
FSIndex timeIndex = indexes.getAnnotationIndex(TimeAnnot.type);
Iterator timeIter = timeIndex.iterator();
while (timeIter.hasNext())
{
  TimeAnnot time = (TimeAnnot)timeIter.next();

  //do something
}
```

Now that we've explained the basics, let's take a look at the process method for `com.ibm.uima.tutorial.ex4.MeetingAnnotator`. Since we're looking for a combination of a RoomNumber, a Date, and two Times, there are four nested iterators. (There's surely a better algorithm for doing this, but to keep things simple we're just going to look at every combination of the four items.)

For each combination of the four annotations, we compute the span of text that includes all of them, and then we check to see if that span is smaller than a "window" size, a configuration parameter. There are also some checks to make sure that we don't annotate the same span of text multiple times. If all the checks pass, we create a Meeting annotation over the whole span. There's really nothing to it!

The XML descriptor, located in `docs/examples/descriptors/tutorial/ex4/MeetingAnnotator.xml`, is also very straightforward. An important difference from previous descriptors is that this is the first annotator we've discussed that has input requirements. This can be seen on the "Capabilities" page of the Component Descriptor Editor:



If we were to run the MeetingAnnotator on its own, it wouldn't detect anything because it wouldn't have any input annotations to work with. The required input annotations can be produced by the RoomNumber and DateTime annotators. So, we create an aggregate Analysis Engine containing these two annotators, followed by the Meeting annotator. This aggregate is illustrated in Figure 9. The descriptor for this is in `docs/examples/descriptors/tutorial/ex4/MeetingDetectorTAE.xml`. Give it a try in the Document Analyzer.

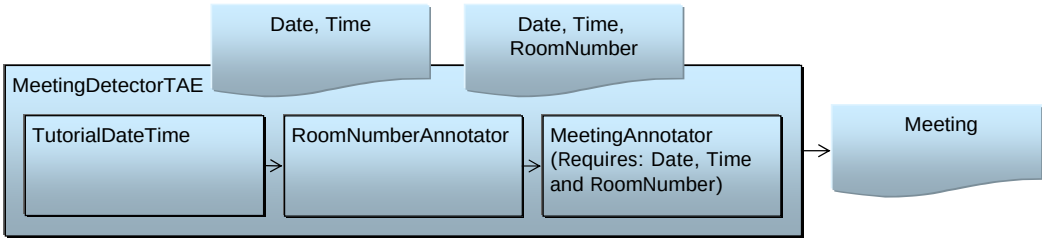


Figure 9: An Aggregate Analysis Engine where an internal component uses output from previous engines.

4.4 Other examples

The UIMA SDK include several other examples you may find interesting, including

- SimpleTokenAndSentenceAnnotator – a simple tokenizer and sentence annotator.
- PersonTitleDBWriterCasConsumer – a sample CAS Consumer which populates a relational database with some annotations. It uses JDBC and in this example, hooks up with the Open Source Apache Derby database.

4.5 Additional Topics

4.5.1 Contract for Annotator methods called by the Framework

Every instance of an Annotator is associated with one and only one thread. An instance never has to worry about running some method on one thread, and then asynchronously being called using another thread. This approach simplifies the design of annotators – they do not have to be designed to support multi-threading. When multiple threading is wanted, for performance, multiple instances of the Annotator are created, each one running on just one thread.

The following table defines the methods called by the framework, when they are called, and the requirements annotator implementations must follow.

Method	When Called by Framework	Requirements
initialize	Called once, when instance is created.	Should read configuration parameter information and set up for processing CASes
typeSystemInit	Called before Process whenever the type system in the CAS being passed in differs from what was previously passed in a Process call (and called for the first CAS passed in, too). The Type System being passed to an annotator only changes for the case of remote annotators that are active as servers, receiving possibly different type systems to operate on.	Typically, users of JCas do not implement any method for this. An annotator can use this call to read the CAS type system and setup any instance variables that make accessing the types and features convenient.
process	Called once for each CAS. Called by the application if not using Collection Processing Manager (CPM); the application calls the process method on the analysis engine, which is then delegated by the framework to all the annotators in the engine. For Collection Processing application, the CPM calls the process method. If the application creates and manages your own Collection Processing Engine via API calls (see JavaDocs), the application calls this on the Collection Processing Engine, and it is delegated by the framework to the components.	Process the CAS, adding and/or modifying elements in it

destroy	This method is called by the Collection Processing Manager framework when the collection processing completes. It can also be called by an application on the Engine object, in which case it is propagated to all contained annotators.	An annotator should release all resources, close files, close database connections, etc., and return to a state where another initialize call could be received to restart. Typically, after a destroy call, no further calls will be made to an annotator instance.
reconfigure	This method is never called by the framework, unless an application calls it on the Engine object – in which case it the framework propagates it to all annotators contained in the Engine. Its purpose is to signal that the configuration parameters have changed.	A default implementation of this calls destroy, followed by initialize. This is the only case where initialize would be called more than once. Users should implement whatever logic is needed to return the annotator to an initialized state, including re-reading the configuration parameter data.

4.5.2 Reporting errors from Annotators

There are two broad classes of errors that can occur: recoverable an unrecoverable. Because Annotators are often expected to process very large numbers of artifacts (for example, text documents), they should be written to recover where possible.

For example, if an upstream annotator created some input for an annotator which is invalid, the annotator may want to log this event, ignore the bad input and continue. It may include a notification of this event in the CAS, for further downstream annotators to consider. Or, it may throw an exception (see next section) – but in this case, it cannot do any further processing on that document.

> **Note:** The choice of what to do can be made configurable, using the configuration parameters.

4.5.3 Throwing Exceptions from Annotators

Let's say an invalid regular expression was passed as a parameter to the RoomNumberAnnotator. Because this is an error related to the overall configuration, and not something we could expect to ignore, we should throw an appropriate exception, and most Java programmers would expect to do so like this:

```
throw new AnnotatorConfigurationException("The regular expression " + x + "
is not valid.");
```

UIMA, however, does not do it this way. All UIMA exceptions are *internationalized*, meaning that they support translation into other languages. This is accomplished by eliminating hardcoded message strings and instead using external message

digests. Message digests are files containing (key, value) pairs. The key is used in the Java code instead of the actual message string. This allows the message string to be easily translated later by modifying the message digest file, not the Java code. Also, message strings in the digest can contain parameters that are filled in when the exception is thrown. The format of the message digest file is described in the JavaDocs for the Java class `java.util.PropertyResourceBundler` and in the load method of `java.util.Properties`.

The first thing an annotator developer must choose is what Exception class to use. There are three to choose from:

1. AnnotatorConfigurationException should be thrown from the annotator's initialize() method if invalid configuration parameter values have been specified.
2. AnnotatorInitializationException should be thrown from the annotator's initialize() method if initialization fails for some other reason.
3. AnnotatorProcessException should be thrown from the annotator's process() method if the processing of a particular document fails for any reason.

Generally you will not need to define your own custom exception classes, but if you do they must extend one of these three classes, which are the only types of Exceptions that the annotator interface permits annotators to throw.

All of the UIMA Exception classes share common Constructor varieties. There are four possible arguments:

1. The name of the message digest to use (optional – if not specified the default UIMA message digest is used).

2. The key string used to select the message in the message digest.

3. An object array containing the parameters to include in the message. Messages can have substitutable parts. When the message is given, the string representation of the objects passed are substituted into the message. The object array is often created using the syntax new Object[]{x, y}.

4. Another exception which is the "cause" of the exception you are throwing. This feature is commonly used when you catch another exception and rethrow it. (optional)

If you look at source file (folder: src in Eclipse) `com.ibm.uima.tutorial.ex5.RoomNumberAnnotator`, you will see the following code:

```
Try
{
```

```
      mPatterns[i] = Pattern.compile(patternStrings[i]);
    }
    catch(PatternSyntaxException e)
    {
      throw new AnnotatorConfigurationException(
         MESSAGE_DIGEST, "regex_syntax_error",
         new Object[]{patternStrings[i]}, e);
    }
```

where the MESSAGE_DIGEST constant has the value "com.ibm.uima.tutorial.ex5.RoomNumberAnnotator_Messages".

Message digests are specified using a dotted name, just like Java classes. This file, with the .properties extension, must be present in the class path. In Eclipse, you find this file under the src folder, in the package com.ibm.uima.tutorial.ex5, with the name RoomNumberAnnotator_Messages.properties. Outside of Eclipse, you can find this in the uima_examples.jar with the name com/ibm/uima/tutorial/ex5/RoomNumberAnnotator_Messages.properties. If you look in this file you will see the line:

```
regex_syntax_error = {0} is not a valid regular expression.
```

which is the error message for the example exception we showed above. The placeholder {0} will be filled by the toString() value of the argument passed to the exception constructor – in this case, the regular expression pattern that didn't compile. If there were additional arguments, their locations in the message would be indicated as {1}, {2}, and so on.

If a message digest is not specified in the call to the exception constructor, the default is UIMAException.STANDARD_MESSAGE_CATALOG (whose value is "com.ibm.uima.UIMAException_Messages" in the current release but may change). This message digest is located in the uima_core.jar file at com/ibm/uima/UIMAException_messages.properties – you can take a look to see if any of these exception messages are useful to use.

To try out the regex_syntax_error exception, just use the Document Analyzer to run docs/examples/descriptors/tutorial/ex5/RoomNumberAnnotator.xml, which happens to have an invalid regular expression in its configuration parameter settings.

To summarize, here are the steps to take if you want to define your own exception message:

1. Create a file with the .properties extension, where you declare message keys and their associated messages, using the same syntax as shown above for the regex_syntax_error exception. The properties file syntax is more completely described in the JavaDocs for the load method of the java.util.Properties class.

2. Put your properties file somewhere in your class path (it can be in your annotator's .jar file).

3. Define a String constant (called MESSAGE_DIGEST for example) in your annotator code whose value is the dotted name of this properties file. For example, if your properties file is inside your jar file at the location `org/myorg/myannotator/Messages.properties`, then this String constant should have the value `org.myorg.myannotator.Messages`. Do not include the .properties extension. In Java Internationalization terminology, this is called the Resource Bundle name. For more information see the JavaDocs for the PropertyResourceBundle class.

4. In your annotator code, throw an exception like this:

```
throw new AnnotatorConfigurationException(MESSAGE_DIGEST,
"your_message_name",new Object[]{param1,param2,...});
```

You may also wish to look at the JavaDocs for the UIMAException class.

For more information on Java's internationalization features, see the Java Internationalization Guide at http://java.sun.com/j2se/1.4/docs/guide/intl/index.html.

4.5.4 Accessing External Resource Files

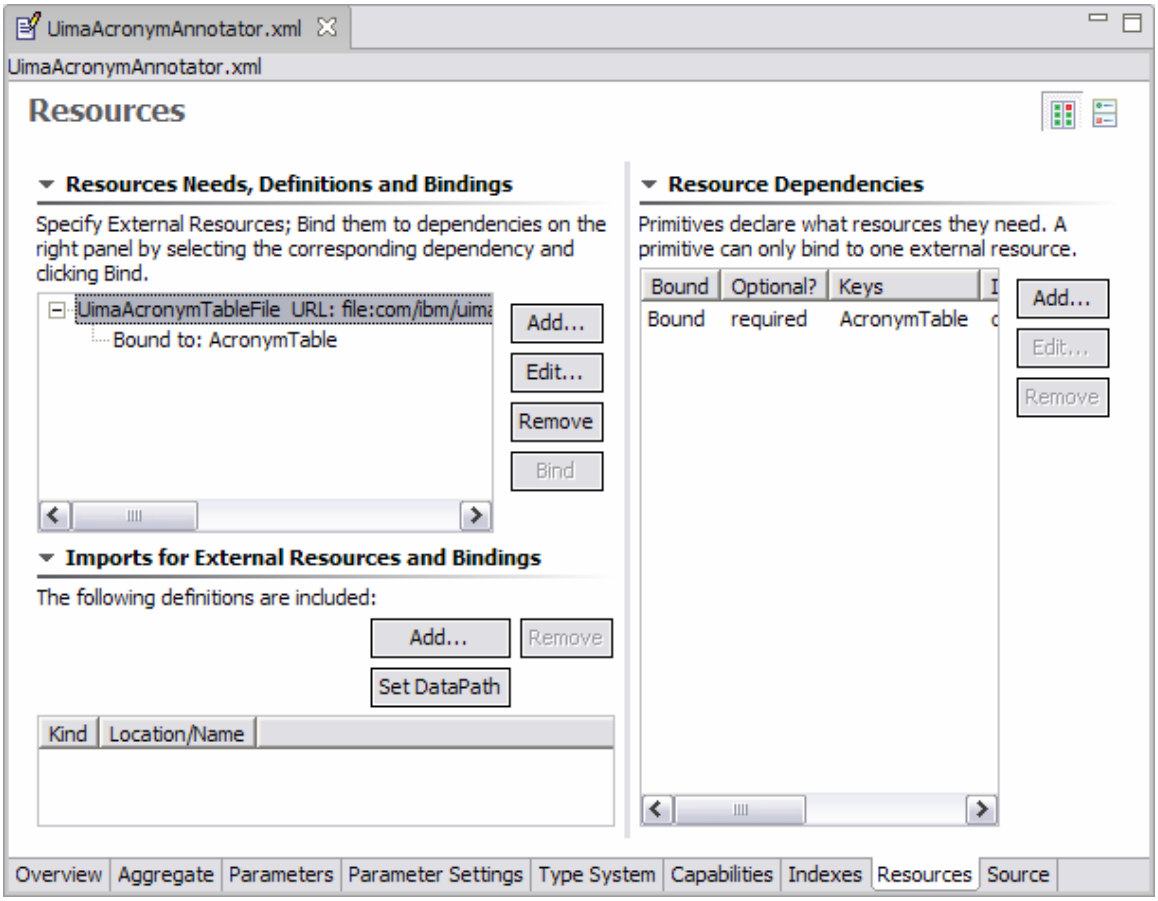
Sometimes you may want an annotator to read from an external file – for example, a long list of keys and values that you are going to build into a HashMap. You could, of course, just introduce a configuration parameter that holds the absolute path to this resource file, and build the HashMap in your annotator's initialize method. However, this is not the best solution for three reasons:

1. Including an absolute path in your descriptor makes your annotator difficult for others to use. Each user will need to edit this descriptor and set the absolute path to a value appropriate for his or her installation.
2. You cannot share the HashMap between multiple annotators. Also, in some deployment scenarios there may be more than one instance of your annotator, and you would like to have the option for them to use the same HashMap instance.
3. Your annotator would become dependent on a particular data representation – the word list would have to come from a file on the local disk and it would have to be in a particular format. It would be better if this were decoupled.

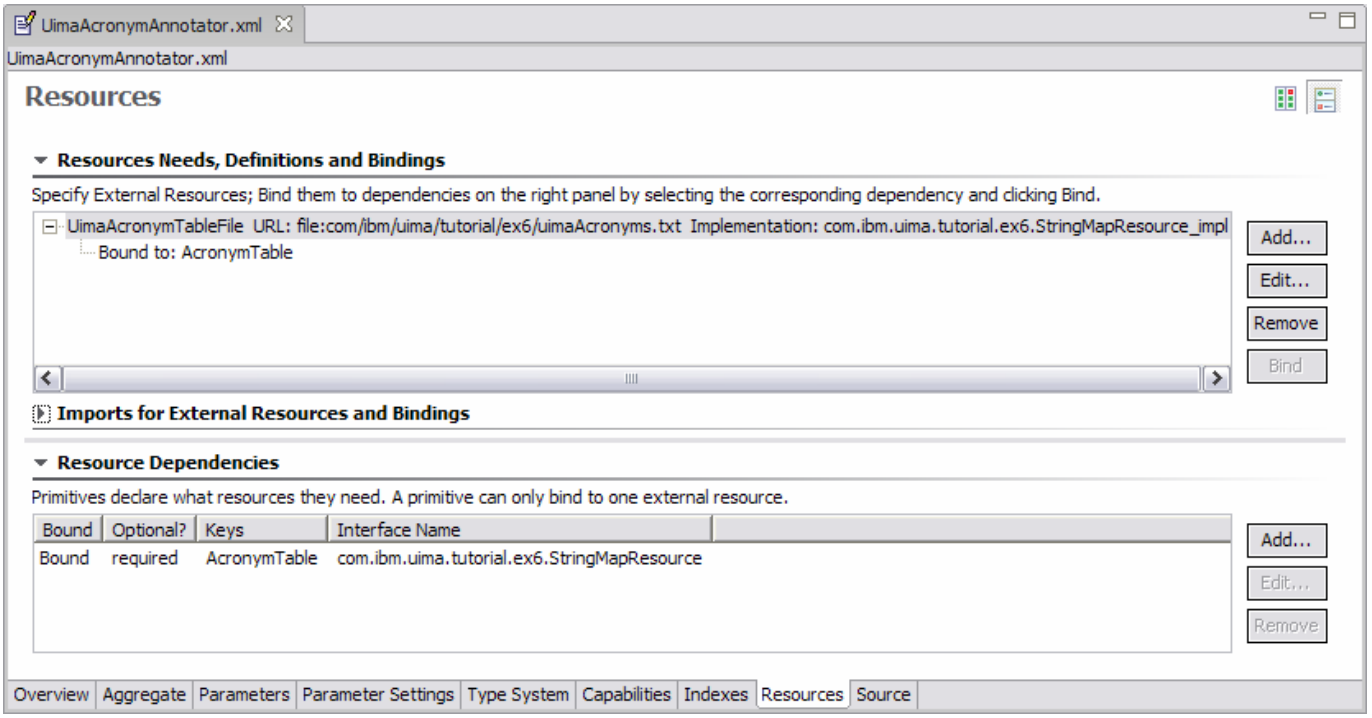
A better way to access external resource is through the ResourceManager component. In this section we are going to show an example of how to use the Resource Manager.

This example annotator will annotate UIMA acronyms (e.g. UIMA, TAE, CAS, JCas) and store the acronym's expanded form as a feature of the annotation. The acronyms and their expanded forms are stored in an external file.

First, look at the `docs/examples/descriptors/tutorial/ex6/UimaAcronymAnnotator.xml` descriptor.



The values of the rows in the two tables are longer than can be easily shown. You can click the small button at the top right to shift the layout from two side-by-side tables, to a vertically stacked layout. You can also click the small twisty on the "Imports for External Resources and Bindings" to collapse this section, because it's not used. Then the same screen will appear like this:



The top window has a scroll bar allowing you to see the rest of the line.

Declaring Resource Dependencies

The bottom window is where an annotator declares an external resource dependency. The XML for this is as follows:

```
<externalResourceDependency>
  <key>AcronymTable</key>
  <description>Table of acronyms and their expanded
               forms.</description>
  <interfaceName> com.ibm.uima.tutorial.ex6.StringMapResource
  </interfaceName>
</externalResourceDependency>
```

The <key> value (AcronymTable) is the name by which the annotator identifies this resource. The key must be unique for all resources that this annotator accesses, but the same key could be used by different annotators to mean different things. The interface name (`com.ibm.uima.tutorial.ex6.StringMapResource`) is the Java interface through which the annotator accesses the data. Specifying an interface name is optional. If you do not specify an interface name, annotators will get direct access to the data file.

Accessing the Resource from the AnnotatorContext

If you look at the `com.ibm.uima.tutorial.ex6.UimaAcronymAnnotator` source, you will see that the annotator accesses this resource from the AnnotatorContext by calling:

```
StringMapResource mMap =
   (StringMapResource)getContext().getResourceObject("AcronymTable");
```

The object returned from the `getResourceObject` method will implement the interface declared in the `<interfaceName>` section of the descriptor, `StringMapResource` in this case. The annotator code does not need to know the location of the data nor the Java class that is being used to read the data and implement the `StringMapResource` interface.

Note that if we did not specify a Java interface in our descriptor, our annotator could directly access the resource data as follows:

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

If necessary, the annotator could also determine the location of the resource file, by calling:

```
URL url = getContext().getResourceURL("AcronymTable");
```

These last two options are only available in the case where the descriptor does not declare a Java interface.

Declaring Resources and Bindings

Refer back to the top window in the Resources page of the Component Descriptor Editor. This is where we specify the location of the resource data, and the Java class used to read the data. For the example, this corresponds to the following section of the descriptor:

```
<resourceManagerConfiguration>
  <externalResources>
    <externalResource>
      <name>UimaAcronymTableFile</name>
      <description>
         A table containing UIMA acronyms and their expanded forms.
      </description>
      <fileResourceSpecifier>
        <fileUrl>file:com/ibm/uima/tutorial/ex6/uimaAcronyms.txt
        </fileUrl>
      </fileResourceSpecifier>
      <implementationName>
         com.ibm.uima.tutorial.ex6.StringMapResource_impl
      </implementationName>
    </externalResource>
  </externalResources>


  <externalResourceBindings>
    <externalResourceBinding>
      <key>AcronymTable</key>
      <resourceName>UimaAcronymTableFile</resourceName>
    </externalResourceBinding>
```

```
  </externalResourceBindings>
</resourceManagerConfiguration>
```

The first section of this XML declares an externalResource, the `UimaAcronymTableFile`. With this, the fileUrl element specifies the path to the data file. This can be an absolute URL (e.g. one that starts with file:/ or file:///, or file://my.host.org/), but that is not recommended because it makes installation of your component more difficult, as noted earlier. Better is a relative URL, which will be looked up within the classpath (and/or datapath), as used in this example. In this case, the file `com/ibm/uima/tutorial/ex6/uimaAcronyms.txt` is located in `uima_examples.jar`, which is in the classpath. If you look in this file you will see the definitions of several UIMA acronyms.

The second section of the XML declares an externalResourceBinding, which connects the key `AcronymTable`, declared in the annotator's external resource dependency, to the actual resource name `UimaAcronymTableFile`. This is rather trivial in this case; for more on bindings see the example `UimaMeetingDetectorTAE.xml` below. There is no global repository for external resources; it is up to the user to define each resource needed by a particular set of annotators.

In the Component Descriptor Editor, bindings are indicated below the external resource. To create a new binding, you select an external resource (which must have previously been defined), and an external resource dependency, and then click the `Bind` button, which only enables if you have selected two things to bind together.

When the Analysis Engine is initialized, it creates a single instance of `StringMapResource_impl` and loads it with the contents of the data file. The UimaAcronymAnnotator then accesses the data through the `StringMapResource` interface. This single instance could be shared among multiple annotators, as will be explained later.

Note that all resource implementation classes (e.g. StringMapResource_impl in the provided example) need to be public and have public, 0-argument constructors, so that they can be instantiated by the framework. (Although Java classes in which you do not define any constructor will, by default, have a 0-argument constructor that doesn't do anything, a class in which you have defined at least one constructor does not get a default 0-argument constructor.)

This annotator is illustrated in Figure 10. To see it in action, just run it using the Document Analyzer. When it finishes, open up the UIMA_Seminars document in the processed results window, (double-click it), and then left-click on one of the

highlighted terms, to see the expandedForm feature's value.

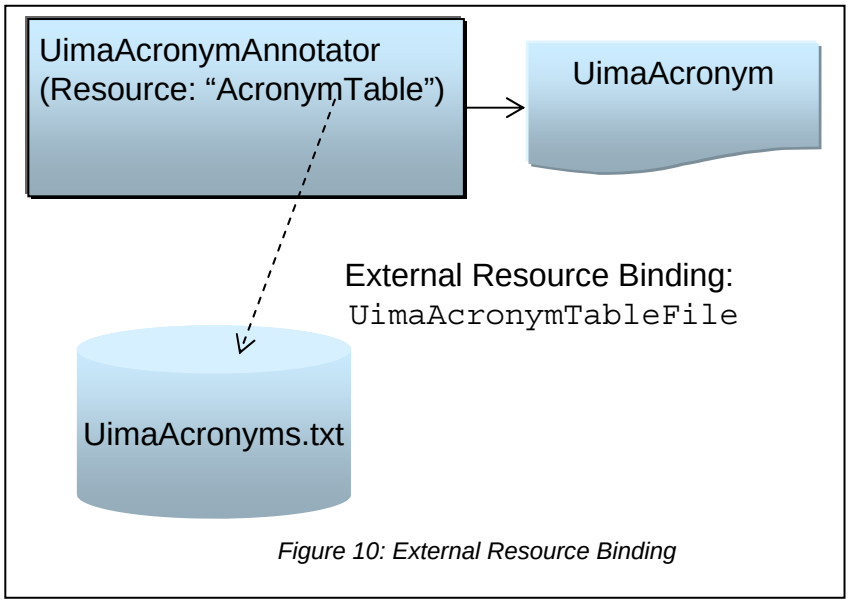


Figure 10: External Resource Binding

By designing our annotator in this way, we have gained some flexibility. We can freely replace the StringMapResource_impl class with any other implementation that implements the simple StringMapResource interface. (For example, for very large resources we might not be able to have the entire map in memory.) We have also made our external resource dependencies explicit in the descriptor, which will help others to deploy our annotator.

Sharing Resources between Annotators

Another advantage of the Resource Manager is that it allows our data to be shared between annotators. To demonstrate this we have developed another annotator that will use the same acronym table. The UimaMeetingAnnotator will iterate over Meeting annotations discovered by the Meeting Detector we previously developed and attempt to determine whether the topic of the meeting is related to UIMA. It will do this by looking for occurrences of UIMA acronyms in close proximity to the meeting annotation. We could implement this by using the UimaAcronymAnnotator, of course, but for the sake of this example we will have the UimaMeetingAnnotator access the acronym map directly.

The Java code for the UimaMeetingAnnotator in example 6 creates a new type, UimaMeeting, if it finds a meeting with 50 characters of the UIMA acronym.

We combine three analysis engines, the UimaAcronymAnnotator to annotate UIMA acronyms, the MeetingDectector from example 4 to find meetings and finally the

UimaMeetingAnnotator to annotate just meetings about UIMA. Together these are assembled to form the new aggregate analysis engine, UimaMeetingDectector. This aggregate and the sharing of a common resource are illustrated in Figure 11.

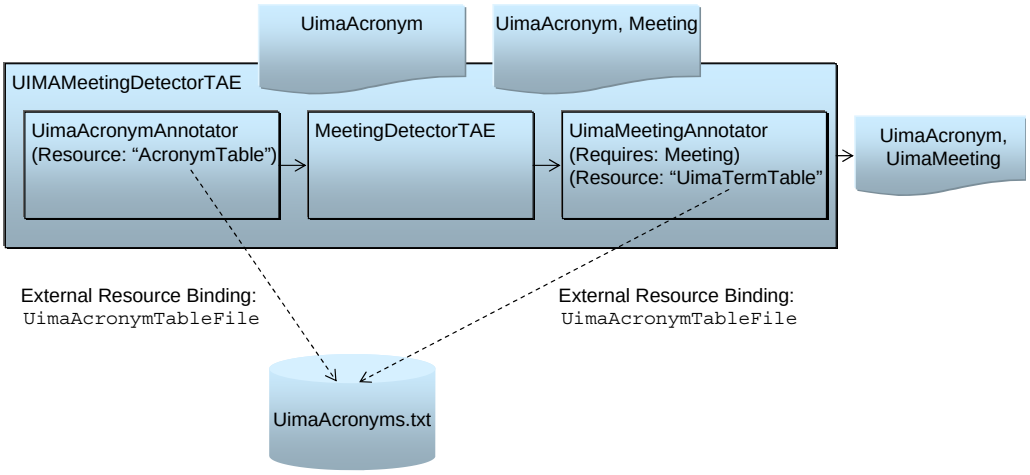
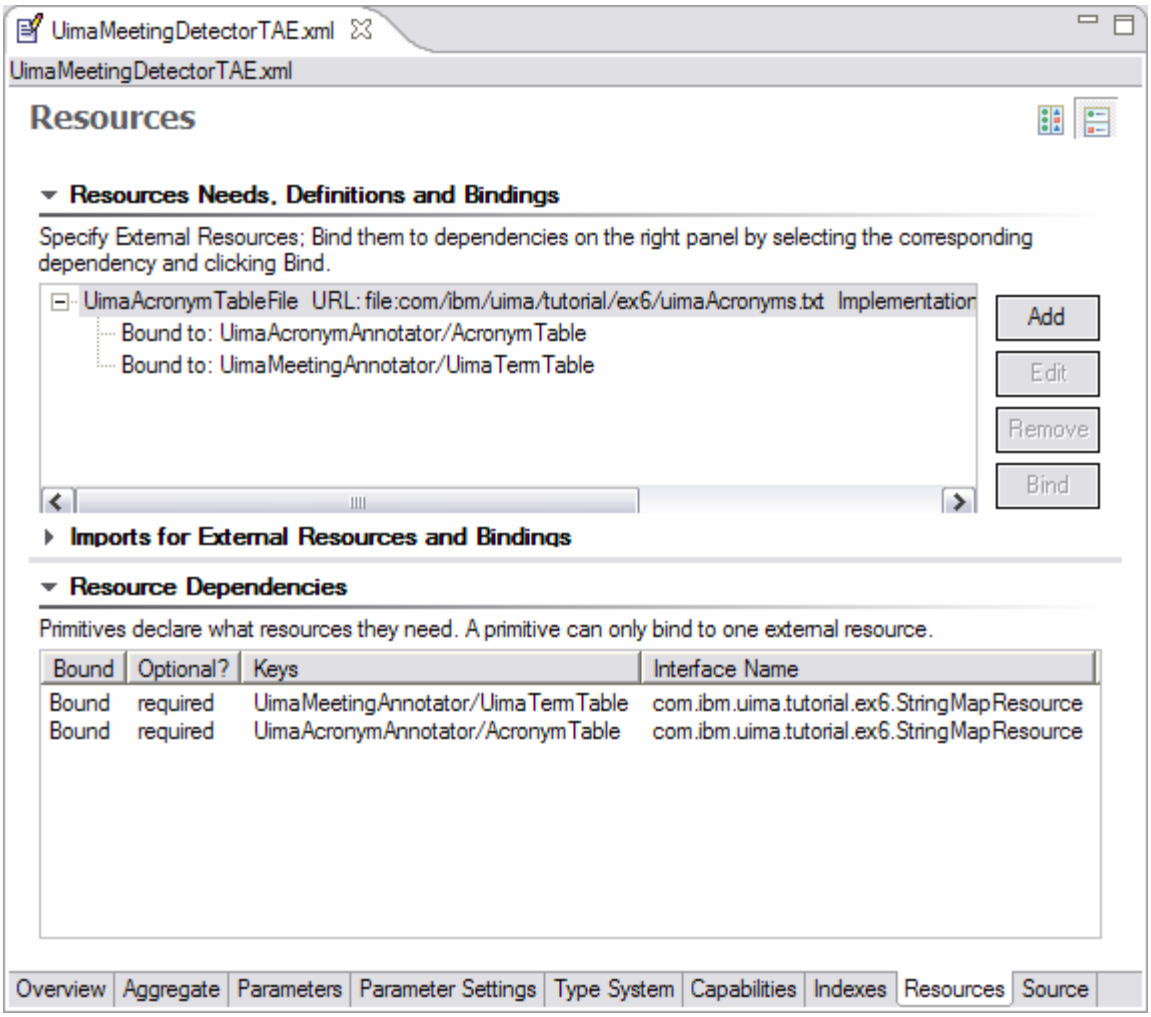


Figure 11: Component engines of an aggregate share a common resource

The important thing to notice is in the `UimaMeetingDetectorTAE.xml` aggregate descriptor. It includes both the UimaMeetingAnnotator and the UimaAcronymAnnotator, and contains a single declaration of the UimaAcronymTableFile resource. (The actual example has the order of the first two annotators reversed versus the above picture, which is OK since they do not depend on one another).

It also binds the resources as follows:




```
<externalResourceBindings>
  <externalResourceBinding>
    <key>UimaAcronymAnnotator/AcronymTable</key>
    <resourceName>UimaAcronymTableFile</resourceName>
  </externalResourceBinding>

  <externalResourceBinding>
    <key>UimaMeetingAnnotator/UimaTermTable</key>
    <resourceName>UimaAcronymTableFile</resourceName>
  </externalResourceBinding>
</externalResourceBindings>
```

This binds the resource dependencies of both the UimaAcronymAnnotator (which uses the name AcronymTable) and UimaMeetingAnnotator (which uses UimaTermTable) to the single declared resource named UimaAcronymFile. Therefore they will share the same instance. Resource bindings in the aggregate descriptor ***override*** any resource declarations in individual annotator descriptors.

If we wanted to have the annotators use different acronym tables, we could easily do that. We would simply have to change the resourceName elements in the

bindings so that they referred to two different resources. The Resource Manager gives us the flexibility to make this decision at deployment time, without changing any Java code.

4.5.5 Result Specification Setting

The Result Specification is a parameter passed to all annotators, as the second argument in the process(...) call. It is a list of output types and / or type:feature specifications, which are expected to be "output" from the annotator. Annotators may use this to optimize their operations, when possible, for those cases where only particular outputs are wanted. The interface to the Result Specification object (see the JavaDocs) allows querying both types and particular features of types.

Sometimes you can specify the Result Specification; othertimes, you cannot (for instance, inside a Collection Processing Engine, you cannot). When you cannot specify it, or choose not to specify it (for example, using the form of the process(...) call on an Analysis Engine that doesn't include the Result Specification), a "Default" Result Specification is used.

Default Result Specification

The default Result Specification is taken from the Engine's output Capability Specification. Remember that a Capability Specification has both inputs and outputs, can specify types and / or features, and there can be more than one Capability Set. If there is more than one set, the logical union of these sets is used. The default Result Specification is exactly what's included in the output Capability Specification.

Passing Result Specifications to Annotators

If you are not using aggregation or collection processing, but instead are instantiating your own primitive analysis engines and calling their process methods, you can pass whatever Result Specification is appropriate in your call to process(CAS, ResultSpecification). For primitive engines, whatever you pass in is passed along as the value of the 2nd argument in the annotator's process() method. If you use the form of the call without the Result Specification, the default Result Specification is created and passed, as above.

Aggregates

For aggregate engines, the value passed to the primitive annotator code depends on the kind of flow.

Fixed Flow

For FixedFlow, any ResultSpecification passed into the aggregate is ignored, and instead, each primitive annotator is passed a result spec that corresponds to the union of its output capability specifications at the primitive descriptor level. If no output capability specification is given, the annotator will still be called, but the result specification will be empty.

CapabilityLanguageFlow

For CapabilityLanguageFlow, each annotator is passed a ResultSpecification that is the intersection of the primitive annotator's output Capability Specification with the ResultSpecification passed to the aggregate. If this intersection is null (the annotator does not produce any type or feature included in the ResultSpecification), the annotator will not be called at all.

Therefore, if using the CapabilityLanguageFlow, if you want to supply a custom ResultSpecification for the aggregate it must include any intermediate types that need to be produced internally in the flow, or else things will not work properly.

Special rule for skipping Analysis Engines

When using the CapabilityLanguageFlow, an annotator will be also be skipped if all of its outputs are in the output capability of some annotator(s) that has (have) executed previously in the flow. The concept here is that if all of an annotator's output types have already been produced, that annotator will not be called.

For an Aggregate, each annotator is passed a Result Specification that is the intersection of the set of types mentioned in its output with the Result Specification passed to the aggregate. If this intersection is null (the annotator does not produce any type included in the ResultSpecification), the annotator will not be called at all.

Therefore, if using the CapabilityLanguageFlow, if you want to supply a custom ResultSpecification for the aggregate it must include any intermediate types that need to be produced, or else things will not work properly.

Collection Processing Engines

The Default Result Specification is always used for all components of a Collection Processing Engine.

4.5.6 Class path setup when using JCas

JCas provides Java classes that correspond to each CAS type in an application. These classes are generated by the JCasGen utility (which can be automatically invoked from the Component Descriptor Editor).

The Java source classes generated by the JCasGen utility are typically compiled and packaged into a JAR file. This JAR file must be present in the classpath of the UIMA application.

More details on issues around setting up this class path, including deployment issues where class loaders are being used to isolate multiple UIMA applications inside a single running Java Virtual Machine, please see ***Class Loaders in UIMA*** on page 24-342.

4.5.7 Using the Shell Scripts

The SDK includes a /bin subdirectory containing shell scripts, for Windows (.bat files) and Linux (.sh files). Many of these scripts invoke sample Java programs which require a class path. The UIMA required files and directories on the class path are set up using the shell script: `setUimaClassPath`.

If you need to include files on the class path, the scripts are set up to add anything you specify in the environment variable UIMA_CLASSPATH to the classpath. So, for example, if you are running the document analyzer, and wanted it to find a Java class file named (on Windows) c:\a\b\c\myProject\myJarFile.jar, you could first issue a set command to set the UIMA_CLASSPATH to this file, followed by the documentAnalyzer script:

```
set UIMA_CLASSPATH=c:\a\b\c\myProject\myJarFile.jar
documentAnalyzer
```

Other environment variables are used by the shell scripts, as follows:

UIMA_HOME	Path where the UIMA SDK was installed. Set automatically if installing via the InstallShield installer.
JAVA_HOME	(Optional) Path to a Java Runtime Environment. If not set, the Java JRE that is shipped with the UIMA SDK (InstallShield versions) is used.
UIMA_DATAPATH	(Optional) if specified, a path specification to use as the default DataPath (see section 20.2)
ECLIPSE_HOME	(Optional) Needs to be set to the root of your Eclipse installation when using shell scripts that invoke Eclipse (e.g. jcasgen_merge)

4.6 Common Pitfalls

Here are some things to avoid doing in your annotator code:

Retaining references to JCas objects between calls to process()

The JCas will be cleared between calls to your annotator's process() method. All of the analysis results related to the previous document will be deleted to make way for analysis of a new document. Therefore, you should never save a reference to a JCas Feature Structure object (i.e. an instance of a class created using JCasGen) and attempt to reuse it in a future invocation of the process() method. If you do so, the results will be undefined.

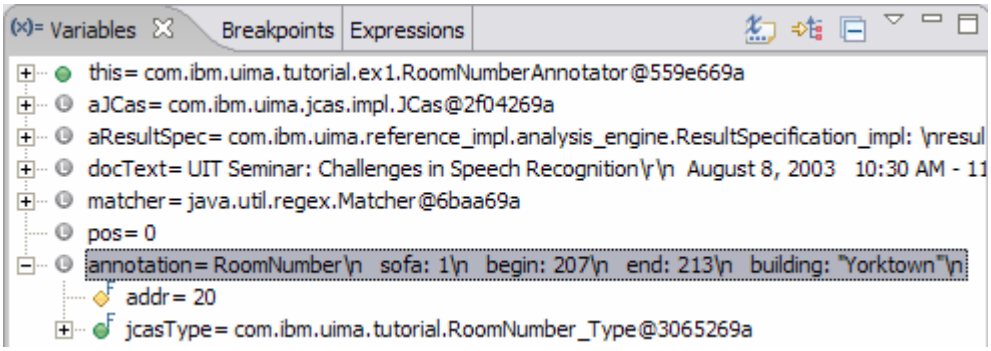
Careless use of static data

Always keep in mind that an application that uses your annotator may create multiple instances of your annotator class. A multithreaded application may attempt to use two instances of your annotator to process two different documents simultaneously. This will generally not cause any problems as long as your annotator instances do not share static data.

In general, you should not use static variables other than static final constants of primitive data types (String, int, float, etc). Other types of static variables may allow one annotator instance to set a value that affects another annotator instance, which can lead to unexpected effects. Also, static references to classes that aren't thread-safe are likely to cause errors in multithreaded applications.

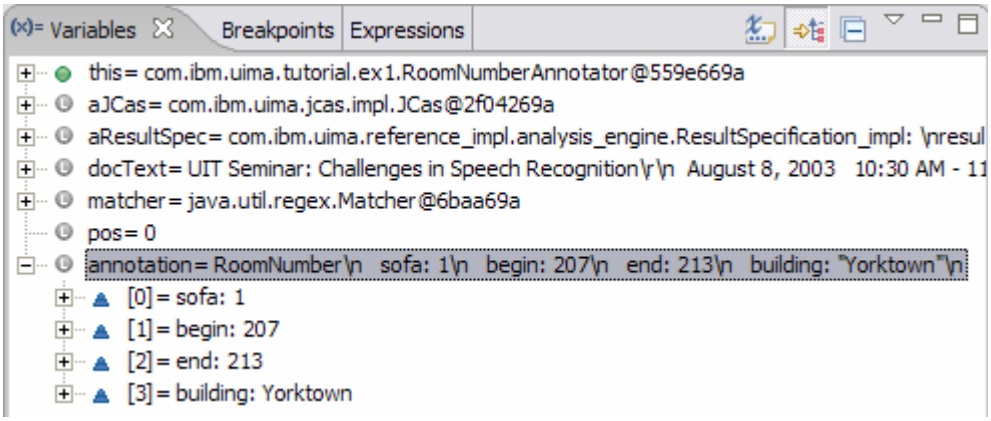
4.7 Viewing FeatureStructures in the Eclipse debugger

Eclipse (as of version 3.1 or later) has a new feature for viewing Java Logical Structures. When enabled, it will permit you to see a view of FeatureStructure objects which show all of the features. For example, here is a view of a FeatureStructure for the RoomNumber annotation, from the tutorial example 1:



The "annotation" object in Java shows as a 2 element object, not very convenient for seeing the features. But if you turn on the Java Logical Structure mode by pushing this button:

the features of the FeatureStructure instance will be shown:



4.8 Introduction to Analysis Engine Descriptor XML Syntax

This section is an introduction to the syntax used for Analysis Engine Descriptors. Most users do not need to understand these details; they can use the Component Descriptor Editor Eclipse plugin to edit Analysis Engine Descriptors rather than editing the XML directly.

This section walks through the actual XML descriptor for the RoomNumberAnnotator example introduced in section 4.1 . The discussion is divided into several logical sections of the descriptor.

The full specification for Analysis Engine Descriptors is defined in *Chapter 20 **Component Descriptor Reference***.

4.8.1 Header and Annotator Class Identification

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--  Descriptor for the example RoomNumberAnnotator. -->
<taeDescription xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <frameworkImplementation>com.ibm.uima.java</frameworkImplementation>
  <primitive>true</primitive>
  <annotatorImplementationName>
    com.ibm.uima.tutorial.ex1.RoomNumberAnnotator
  </annotatorImplementationName>
```

The document begins with a standard XML header and a comment. The root element of the document is named `<taeDescription>`, and must specify the XML namespace `http://uima.watson.ibm.com/resourceSpecifier`.

The first subelement, `<frameworkImplementation>`, must contain the value `com.ibm.uima.java`. The second subelement, `<primitive>`, contains the Boolean value true, indicating that this XML document describes a *Primitive* Analysis Engine. A Primitive Analysis Engine is comprised of a single annotator. It is also possible to construct XML descriptors for non-primitive or *Aggregate* Analysis Engines; this is covered later.

The next element, `<annotatorImplementationName>`, contains the fully-qualified class name of our annotator class. This is how the UIMA framework determines which annotator class to instantiate.

4.8.2 Simple Metadata Attributes

```
<analysisEngineMetaData>
  <name>Room Number Annotator</name>
  <description>An example annotator that searches for room numbers in
     the IBM  Watson research buildings.</description>
  <version>1.0</version>
  <vendor>IBM</vendor>
```

Here are shown four simple metadata fields – name, description, version, and vendor. Providing values for these fields is optional, but recommended.

4.8.3 Type System Definition

```
<typeSystemDescription>
  <imports>
    <import location="TutorialTypeSystem.xml"/>
  </imports>
</typeSystemDescription>
```

This section of the XML descriptor defines which types the annotator works with. The recommended way to do this is to *import* the type system definition from a separate file, as shown here. The location specified here should be a relative path, and it will be resolved relative to the location of the aggregate descriptor. It is also possible to define types directly in the Analysis Engine descriptor, but these types will not be easily shareable by others.

4.8.4 Capabilities

```
<capabilities>
  <capability>
    <inputs />
```

```
    <outputs>
      <type>com.ibm.uima.tutorial.RoomNumber</type>
      <feature>com.ibm.uima.tutorial.RoomNumber:building</feature>
    </outputs>
  </capability>
</capabilities>
```

The last section of the descriptor describes the *Capabilities* of the annotator – the Types/Features it consumes (input) and the Types/Features that it produces (output). These must be the names of types and features that exist in the ANALYSIS ENGINE descriptor's type system definition.

Our annotator outputs only one Type, RoomNumber and one feature, RoomNumber:building. The fully-qualified names (including namespace) are needed.

The building feature is listed separately here, but clearly specifying every feature for a complex type would be cumbersome. Therefore, a shortcut syntax exists. The <outputs> section above could be replaced with the equivalent section:

```
<outputs>
  <type allAnnotatorFeatures ="true"> com.ibm.uima.tutorial.RoomNumber
  </type>
</outputs>
```

4.8.5 Configuration Parameters (Optional)

Configuration Parameter Declarations

```
<configurationParameters>
  <configurationParameter>
    <name>Patterns</name>
    <description>List of room number regular expression patterns.
    </description>
    <type>String</type>
    <multiValued>true</multiValued>
    <mandatory>true</mandatory>
  </configurationParameter>
  <configurationParameter>
    <name>Locations</name>
    <description>List of locations corresponding to the room number
       expressions specified by the Patterns parameter.
    </description>
    <type>String</type>
    <multiValued>true</multiValued>
    <mandatory>true</mandatory>
  </configurationParameter>
</configurationParameters>
```

The <configurationParameters> element contains the definitions of the configuration parameters that our annotator accepts. We have declared two parameters. For each configuration parameter, the following are specified:

- **name** – the name that the annotator code uses to refer to the parameter
- **description** – a natural language description of the intent of the parameter
- **type** – the data type of the parameter's value – must be one of String, Integer, Float, or Boolean.
- **multiValued** – true if the parameter can take multiple-values (an array), false if the parameter takes only a single value.
- **mandatory** – true if a value must be provided for the parameter

Both of our parameters are mandatory and accept an array of Strings as their value.

4.8.6 Configuration Parameter Settings

```
<configurationParameterSettings>
  <nameValuePair>
    <name>Patterns</name>
    <value>
      <array>
        <string>\b[0-4]\d-[0-2]\d\d\b</string>
        <string>\b[G1-4][NS]-[A-Z]\d\d\b</string>
        <string>\bJ[12]-[A-Z]\d\d\b</string>
      </array>
    </value>
  </nameValuePair>
  <nameValuePair>
    <name>Locations</name>
    <value>
      <array>
        <string>Watson - Yorktown</string>
        <string>Watson - Hawthorne I</string>
        <string>Watson - Hawthorne II</string>
      </array>
    </value>
  </nameValuePair>
</configurationParameterSettings>
```

4.8.7 Aggregate Analysis Engine Descriptor

```
<?xml version="1.0" encoding="UTF-8" ?>
<taeDescription xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <frameworkImplementation>com.ibm.uima.java</frameworkImplementation>
  <primitive>false</primitive>

  <delegateAnalysisEngineSpecifiers>
    <delegateAnalysisEngine key="RoomNumber">
      <import location="../ex2/RoomNumberAnnotator.xml" />
    </delegateAnalysisEngine>
```

```
    <delegateAnalysisEngine key="DateTime">
      <import location="TutorialDateTime.xml" />
    </delegateAnalysisEngine>
  </delegateAnalysisEngineSpecifiers>
```

The first difference between this descriptor and an individual annotator's descriptor is that the `<primitive>` element contains the value `false`. This indicates that this Analysis Engine (AE) is an aggregate AE rather than a primitive AE.

Then, instead of a single annotator class name, we have a list of `delegateAnalysisEngineSpecifiers`. Each specifies one of the components that constitute our Aggregate . We refer to each component by the relative path from this XML descriptor to the component AE's XML descriptor.

This list of component AEs does not imply a fixed ordering. Ordering is done by another section of the descriptor:

```
 <analysisEngineMetaData>
  <name>Aggregate TAE - Room Number and DateTime Annotators</name>
  <description>Detects Room Numbers, Dates, and Times</description>
  <flowConstraints>
    <fixedFlow>
      <node>RoomNumber</node>
      <node>DateTime</node>
    </fixedFlow>
  </flowConstraints>
```

Currently, a fixedFlow is required, and we must specify the exact ordering in which the AEs will be executed. In this case, it doesn't really matter, since the RoomNumber and DateTime annotators do not have any dependencies on one another.

Finally, the descriptor has a capabilities section, which has exactly the same syntax as a primitive AE's capabilities section:

```
<capabilities>
  <capability>
    <inputs />
    <outputs>
      <type allAnnotatorFeatures="true">com.ibm.uima.tutorial.RoomNumber
      </type>
      <type allAnnotatorFeatures="true">com.ibm.uima.tutorial.DateAnnot
      </type>
      <type allAnnotatorFeatures="true">com.ibm.uima.tutorial.TimeAnnot
      </type>
    </outputs>
    <languagesSupported>
      <language>en</language>
    </languagesSupported>
  </capability>
</capabilities>
```

Chapter 5 Collection Processing Engine Developer's Guide

The UIMA Analysis Engine interface provides support for developing and integrating algorithms that analyze unstructured data. Analysis Engines are designed to operate on a per-document basis. Their interface handles one CAS at a time. UIMA provides additional support for applying analysis engines to collections of unstructured data with its *Collection Processing Architecture*. The Collection Processing Architecture defines additional components for reading raw data formats from data collections, preparing the data for processing by Analysis Engines, executing the analysis, extracting analysis results, and deploying the overall flow in a variety of local and distributed configurations.

The functionality defined in the Collection Processing Architecture is implemented by a *Collection Processing Engine* (CPE). A CPE includes an Analysis Engine and adds a *Collection Reader*, a *CAS Initializer*, and *CAS Consumers*. The part of the UIMA Framework that supports CPEs is called the Collection Processing Manager, or CPM.

A Collection Reader provides the interface to the raw input data and knows how to iterate over the data collection. Collection Readers are discussed in Section 5.4.1 . The CAS Initializer prepares an individual data item for analysis and loads it into the CAS. CAS Initializers are discussed in Section 5.4.2 . A CAS Consumer extracts analysis results from the CAS and may also perform *collection level processing*, or analysis over a collection of CASes. CAS Consumers are discussed in Section 5.4.3 .

Analysis Engines and CAS Consumers are both instances of *CAS Processors*. A CPM may contain multiple CAS Processors. An Analysis Engine may be a Primitive or an Aggregate (composed of other Analysis Engines). Aggregates may contain Cas Consumers. While Collection Readers and CAS Initializers always run in the same JVM as the CPM, a CAS Processor may be deployed in a variety of local and distributed modes, providing a number of options for scalability and robustness. The different deployment options are covered in detail in Section 5.5 .

Each of the components in a CPE has an interface specified by the UIMA Collection Processing Architecture and is described by a declarative XML descriptor file. Similarly, the CPE itself has a well defined component interface and is described by a declarative XML descriptor file.

A user creates a CPE by assembling the components mentioned above. The UIMA SDK provides a graphical tool, called the CPE Configurator, for assisting in the assembly of CPEs. Use of this tool is summarized in Section 5.2 , and more details can be found in *Chapter 10* ***Collection Processing Engine Configurator User's*** Guide.

Alternatively, a CPE can be assembled by writing an XML CPE descriptor. Details on the CPE descriptor, including its syntax and content, can be found in the *Chapter 21 **Collection Processing Engine Descriptor Reference***. The individual components have associated XML descriptors, each of which can be created and / or edited using the Component Description Editor.

A CPE is executed by a UIMA infrastructure component called the *Collection Processing Manager* (CPM). The CPM provides a number of services and deployment options that cover instantiation and execution of CPEs, error recovery, and local and distributed deployment of the CPE components.

5.1 CPE Concepts

Figure 12 illustrates the data flow that occurs between the different types of components that make up a CPE.

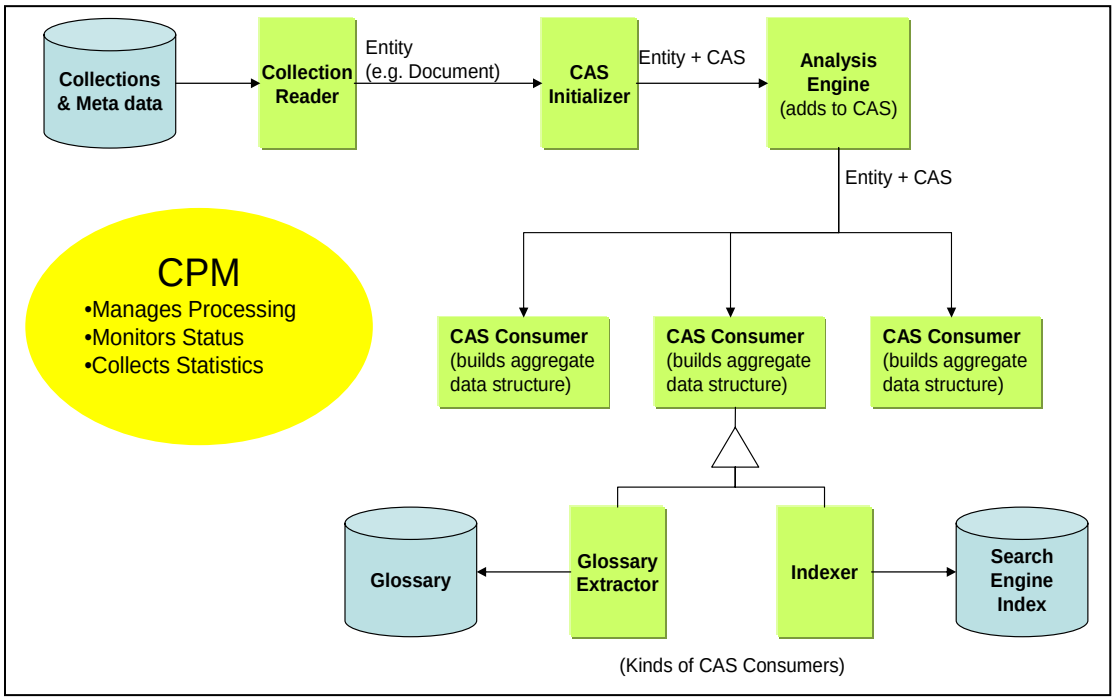


Figure 12 CPE Components

The components of a CPE are:

- *Collection Reader* – interfaces to a collection of data items (e.g., documents) to be analyzed. Collection Readers return CASes that contain the documents to analyze, possibly along with additional metadata.
- *Analysis Engine* – takes a CAS, analyzes its contents, and produces an enriched CAS. Analysis Engines can be recursively composed of other Analysis Engines (called an *Aggregate* Analysis Engine). Aggregates may also contain CAS Consumers.

- *CAS Consumer* – consume the enriched CAS that was produced by the sequence of Analysis Engines before it, and produce an application-specific data structure, such as a search engine index or database.

A fourth type of component, the *CAS Initializer*, may be used by a Collection Reader to populate a CAS from a document. An example of a CAS Initializer is an HTML parser that de-tags an HTML document and also inserts paragraph annotations (determined from `<P>` tags in the original HTML) into the CAS. The Collection Processing Manager orchestrates the data flow within a CPE, monitors status, optionally manages the life-cycle of internal components and collects statistics.

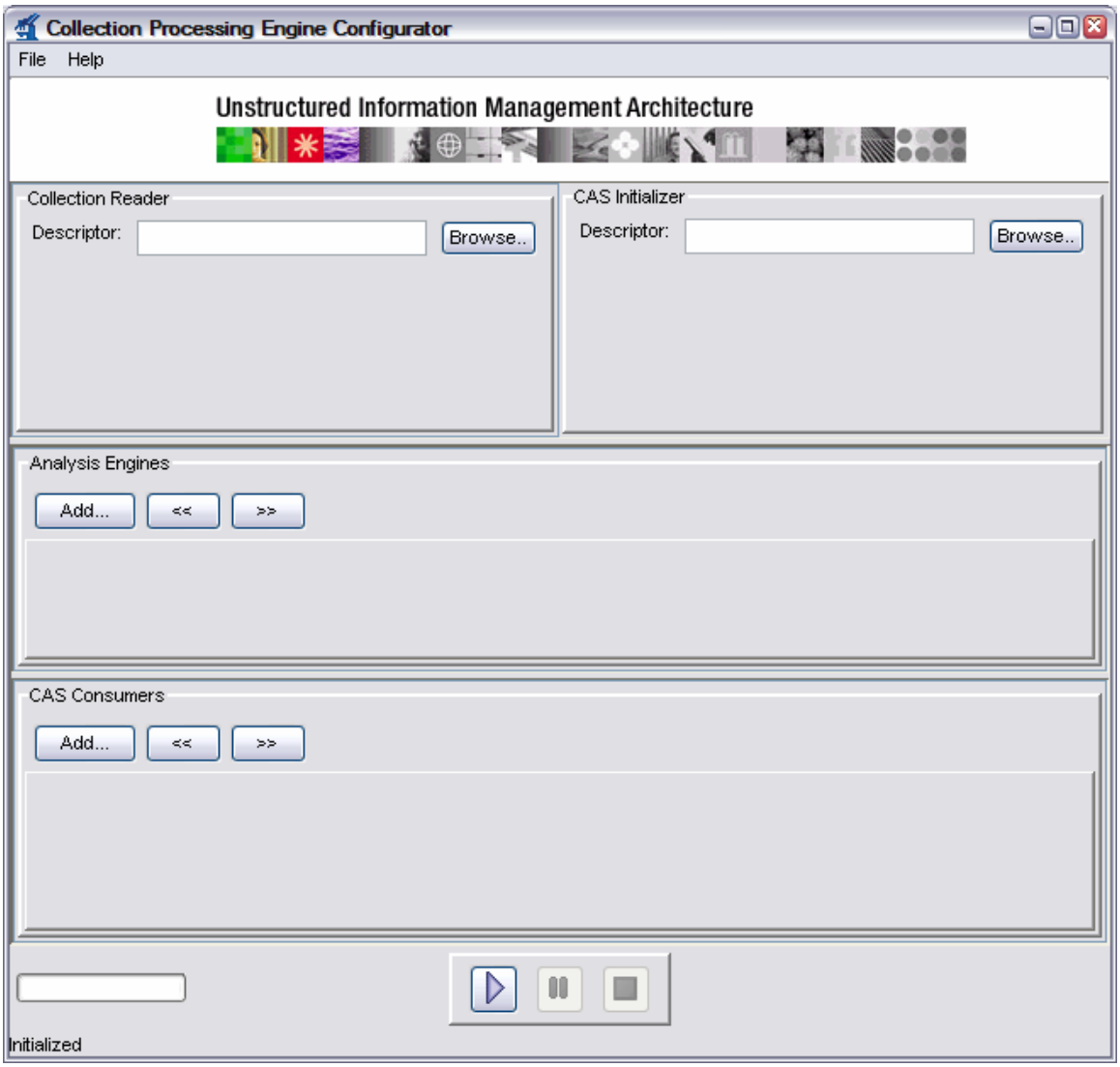
CASes are not saved in a persistent way by the framework. If you want to save CASes, then you have to save each CAS as it comes through (for example) using a CAS Consumer you write to do this, in whatever format you like. The UIMA SDK supplies an example CAS Consumer to save CASes to files, in the externalized XCAS format (an XML version of the CAS). It also supplies an example CAS Consumer to extract information from CASes and store the results into a relational Database, using Java's JDBC APIs.

5.2 The CPE Configurator and the XCAS viewer

5.2.1 Using the CPE Configurator

A CPE can be assembled by writing an XML CPE descriptor. Details on the CPE descriptor, including its syntax and content, can be found in *Chapter 21* ***Collection Processing Engine Descriptor Reference***. Rather than edit raw XML, you may develop a CPE Descriptor using the CPE Configurator tool. The CPE Configurator tool is described briefly in this section, and in more detail in *Chapter 10* ***Collection Processing Engine Configurator User's*** Guide.

The CPE Configurator tool can be run from Eclipse (see ***Running the CPE Configurator from Eclipse*** on page 5-108), or using the `cpeGui` shell script (`cpeGui.bat` on Windows, `cpeGui.sh` on Unix), which is located in the `bin` directory of the UIMA SDK installation. Executing this batch file will display the window shown here:

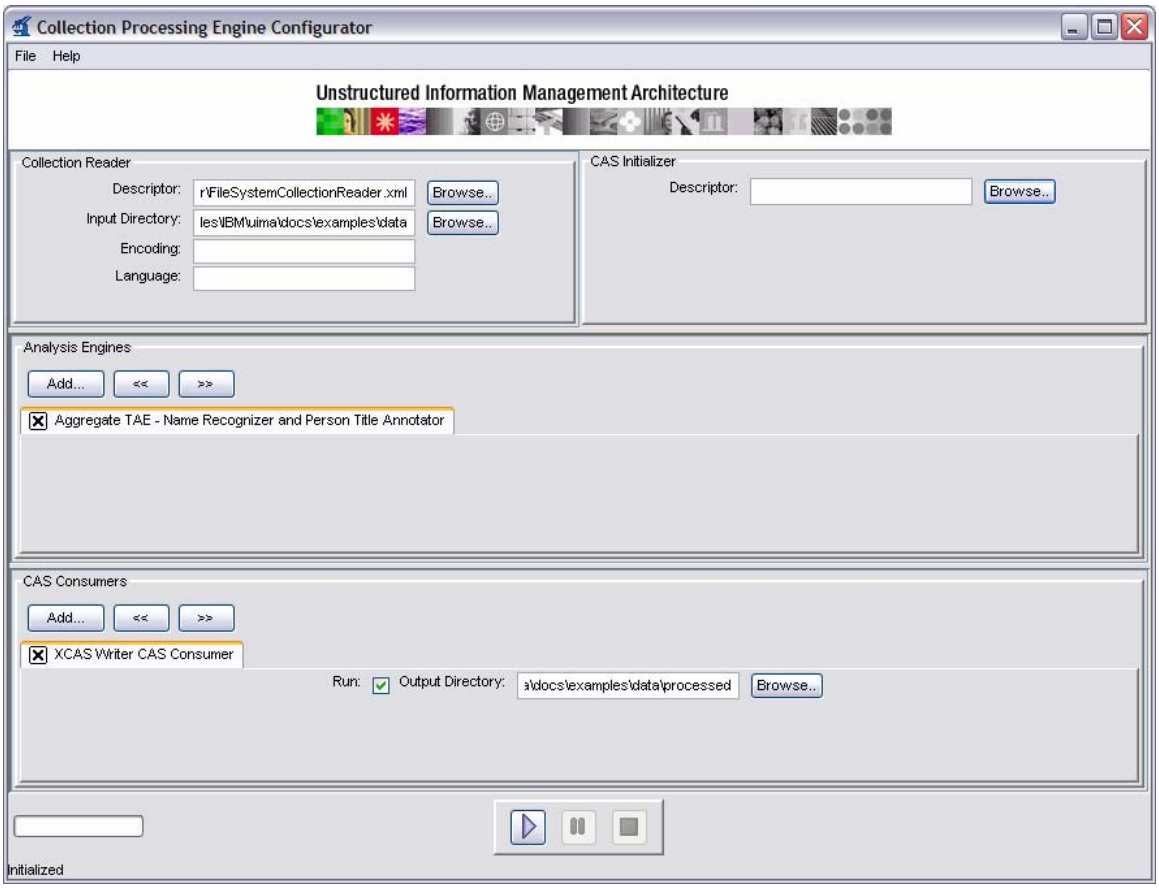


The window is divided into 4 sections, one each for the Collection Reader, CAS Initializer, Analysis Engines, and CAS Consumers. In each section, you select the component(s) you want to include in the CPE by browsing to their XML descriptors. The configuration parameters present in the XML descriptors will then be displayed in the GUI; these can be modified to override the values present in the descriptor. For example, the screen shot below shows the CPE Configurator after the following components have been chosen:

```
Collection Reader: %UIMA_HOME%\docs\examples\descriptors\collection_reader\
FileSystemCollectionReader.xml

Analysis Engine:
%UIMA_HOME%\docs\examples\descriptors\analysis_engine\NamesAndPersonTitles_
TAE.xml

CAS Consumer:
%UIMA_HOME%\docs\examples\descriptors\cas_consumer\XCasWriterCasConsumer.xm
l
```



For the File System Collection Reader, ensure that the Input Directory is set to `%UIMA_HOME%\docs\examples\data`. The other parameters may be left blank. For the XCAS Writer CAS Consumer, ensure that the Output Directory is set to `%UIMA_HOME%\docs\examples\data\processed`.

After selecting each of the components and providing configuration settings, click the play (forward arrow) button at the bottom of the screen to begin processing. A progress bar should be displayed in the lower left corner. (Note that the progress bar will not begin to move until all components have completed their initialization, which may take several seconds.) Once processing has begun, the pause and stop buttons become enabled.

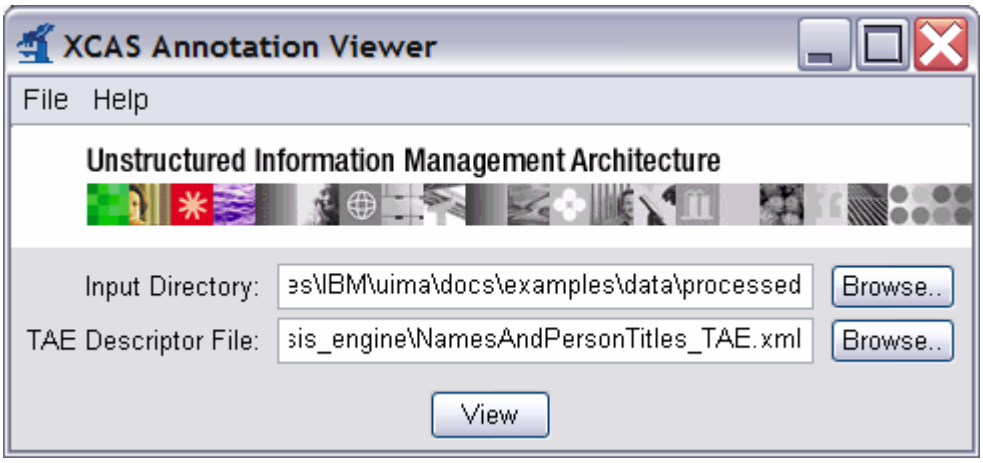
If an error occurs, you will be informed by an error dialog. If processing completes successfully, you will be presented with a performance report.

Using the File menu, you can select `Save CPE Descriptor` to create an .xml descriptor file that defines the CPE you have constructed. Later, you can use `Open CPE Descriptor` to restore the CPE Configurator to the saved state. Also, CPE descriptors can be used to run a CPE from a Java program – see section 5.3 . CPE Descriptors

allow specifying operational parameters, such as error handling options, that are not currently available for configuration through the CPE Configurator. For more information on manually creating a CPE Descriptor, see the *Chapter 21* ***Collection Processing Engine Descriptor Reference***.

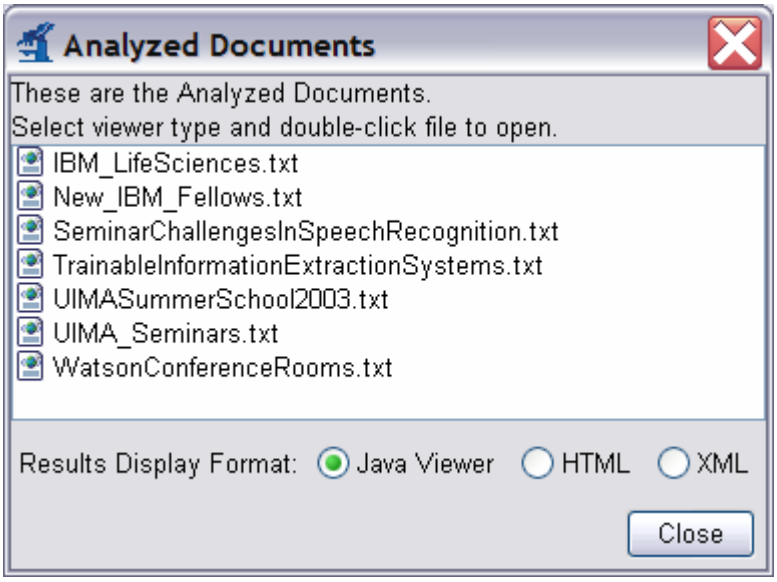
Note that CPE descriptors identify *which* components comprise the CPE, but they do not capture the individual configuration settings for these components. That information is kept in the individual component descriptors. If you have made changes to these settings in the CPE Configurator tool and wish to save the settings back to the original descriptor files, use the `File → Save Component Configuration` action.

The CPE configured above runs a simple name and title annotator on the sample data provided with the UIMA SDK and stores the results using the XCAS Writer CAS Consumer. To view the results, start the XCAS Annotation Viewer by running the `xcasAnnotationViewer` batch file (`xcasAnnotationViewer.bat` on Windows, `xcasAnnotationViewer.sh` on Unix), which is located in the `bin` directory of the UIMA SDK installation. Executing this batch file will display the window shown here:

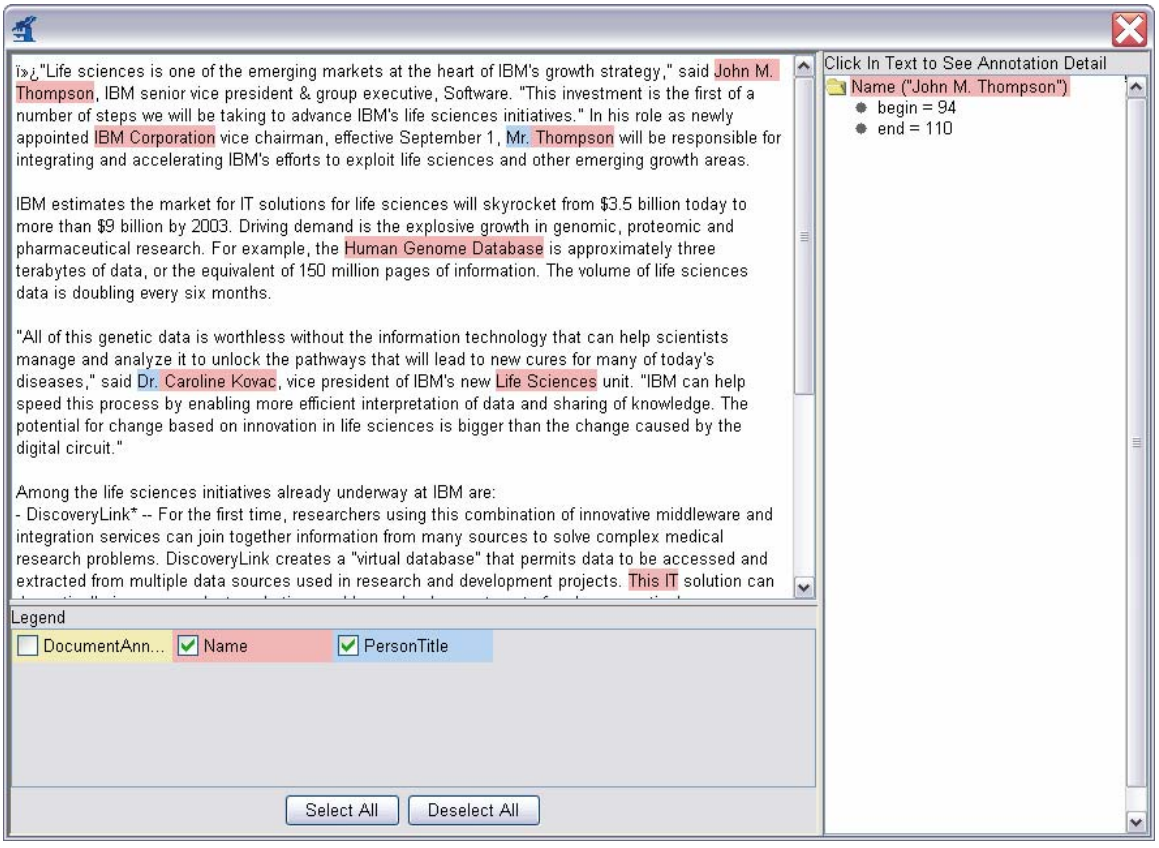


Ensure that the Input Directory is the same as the Output Directory specified for the XCAS Writer CAS Consumer in the CPE configured above (e.g., `%UIMA_HOME%\docs\examples\data\processed`) and that the TAE Descriptor File is set to the Analysis Engine used in the CPE configured above (e.g., `%UIMA_HOME%\docs\examples\descriptors\analysis_engine\NamesAndPersonTitles_TAE.xml`).

Click the View button to display the Analyzed Documents window:



Double click on any document in the list to view the analyzed document. Double clicking the first document, IBM_LifeSciences.txt, will bring up the following window:



This window shows the analysis results for the document. Clicking on any highlighted annotation causes the details for that annotation to be displayed in the

right-hand pane. Here the annotation spanning "John M. Thompson" has been clicked.

Congratulations! You have successfully configured a CPE, saved its descriptor, run the CPE, and viewed the analysis results.

5.2.2 Running the CPE Configurator from Eclipse

If you have followed the instructions in *Chapter 3* ***UIMA SDK Setup for Eclipse*** and imported the example Eclipse project, then you should already have a Run configuration for the CPE Configurator tool (called `UIMA CPE GUI`) configured to run in the example project. Simply run that configuration to start the CPE Configurator.

If you haven't followed the Eclipse setup instructions and wish to run the CPE Configurator tool from Eclipse, you will need to do the following. As installed, this Eclipse launch configuration is associated with the "uima_examples" project. If you've not already done so, you may wish to import that project into your Eclipse workspace. It's located in %UIMA_HOME%/docs/examples. Doing this will supply the Eclipse launcher with all the class files it needs to run the CPE configurator. If you don't do this, please manually add the JAR files for UIMA to the launch configuration.

Also, you need to add any projects or JAR files for any UIMA components you will be running to the launch class path.

> **Note:** A simpler alternative may be to change the CPE launch configuration to be based on your project. If you do that, it will pick up all the files in your project's class path, which you should set up to include all the UIMA framework files. An easy way to do this is to specify in your project's properties' build-path that the uima_examples project is on the build path.

Next, in the Eclipse menu select `Run→ Run...`, which brings up the Run configuration screen.

1. In the Main tab, set the main class to `com.ibm.uima.reference_impl.application.cpm.CpmFrame`

2. In the arguments tab, add the following to the VM arguments `-Xms128M -Xmx256M -Duima.home="C:\Program Files\IBM\uima"` (or wherever you installed the UIMA SDK)

Click the Run button to launch the CPE Configurator, and use it as previously described in this section.

5.3 Running a CPE from Your Own Java Application

The simplest way to run a CPE from a Java application is to first create a CPE descriptor as described in the previous section. Then the CPE can be instantiated and run using the following code:

```
//parse CPE descriptor in file specified on command line
CpeDescription cpeDesc = UIMAFramework.getXMLParser().
parseCpeDescription(new XMLInputSource(args[0]));

//instantiate CPE
mCPE = UIMAFramework.produceCollectionProcessingEngine(cpeDesc);

//Create and register a Status Callback Listener
mCPE.addStatusCallbackListener(new StatusCallbackListenerImpl());

//Start Processing
mCPE.process();
```

This will start the CPE running in a separate thread.

5.3.1 Using Listeners

Updates of the CPM's progress, including any errors that occur, are sent to the callback handler that is registered by the call to `addStatusCallbackListener`, above. The callback handler is a class that implements the CPM's `StatusCallbackListener` interface. It responds to events by printing messages to the console. The source code is fairly straightforward and is not included in this chapter – see the `com.ibm.uima.examples.cpe.SimpleRunCPE.java` in the `%UIMA_HOME%\docs\examples\src` directory for the complete code.

If you need more control over the information in the CPE descriptor, you can manually configure it via its API. See the JavaDocs for package `com.ibm.uima.collection` for more details.

5.4 Developing Collection Processing Components

This section is an introduction to the process of developing Collection Readers, CAS Initializers, and CAS Consumers. The code snippets refer to the classes that can be found in `%UIMA_HOME%\docs\examples\src` example project.

In the following sections, classes you write to represent components need to be public and have public, 0-argument constructors, so that they can be instantiated by the framework. (Although Java classes in which you do not define any constructor will, by default, have a 0-argument constructor that doesn't do anything, a class in which you have defined at least one constructor does not get a default 0-argument constructor.)

5.4.1 Developing Collection Readers

A Collection Reader is responsible for obtaining documents from the collection and returning each document as a CAS. Like all UIMA components, a Collection Reader consists of two parts – the code and an XML descriptor.

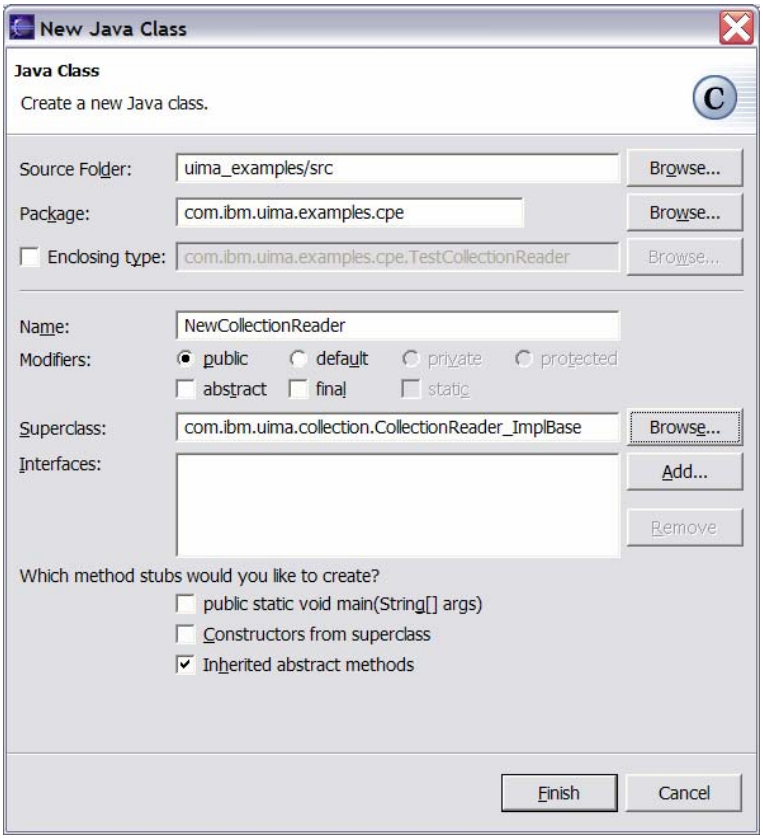
A simple example of a Collection Reader is the "File System Collection Reader," which simply reads documents from files in a specified directory. The Java code is in the class `com.ibm.uima.examples.cpe.FileSystemCollectionReader` and the XML descriptor is `%UIMA_HOME%\docs\examples\descriptors\collection_reader\FileSystemCollectionReader.xml`.

Java Class

The Java class for a Collection Reader must implement the `com.ibm.uima.collection.CollectionReader` interface. You may build your Collection Reader from scratch and implement this interface, or you may extend the convenience base class `com.ibm.uima.collection.CollectionReader_ImplBase`.

The convenience base class provides default implementations for many of the methods defined in the `CollectionReader` interface, and provides abstract definitions for those methods that you are required to implement in your new Collection Reader. Note that if you extend this base class, you do not need to declare that your new Collection Reader implements the `CollectionReader` interface.

Eclipse tip – if you are using Eclipse, you can quickly create the boiler plate code and stubs for all of the required methods by clicking `File` → `New` → `Class` to bring up the "New Java Class" dialogue, specifying `com.ibm.uima.collection.CollectionReader_ImplBase` as the Superclass, and checking "Inherited abstract methods" in the section "Which method stubs would you like to create?", e.g.,



For the rest of this section we will assume that your new Collection Reader extends the `CollectionReader_ImplBase` class, and we will show examples from the `com.ibm.uima.examples.cpe.FileSystemCollectionReader`. If you must inherit from a different super class, you must ensure that your Collection Reader implements the `CollectionReader` interface – see the JavaDocs for `CollectionReader` for more details.

Required Methods

The following abstract methods must be implemented:

initialize()

The `initialize()` method is called by the framework when the Collection Reader is first created. `CollectionReader_ImplBase` actually provides a default implementation of this method (i.e., it is not abstract), so you are not strictly required to implement this method. However, a typical Collection Reader will implement this method to obtain parameter values and perform various initialization steps.

In this method, the Collection Reader class can access the values of its configuration parameters and perform other initialization logic. The example File System Collection Reader reads its configuration parameters and then builds a list of files in the specified input directory, as follows:

```
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]);
    }
  }
}
```

Note: This is the zero-argument version of the initialize method. There is also a method on the Collection Reader interface called `initialize(ResourceSpecifier, Map)` but it is not recommended that you override this method in your code. That method performs internal initialization steps and then calls the zero-argument `initialize()`.

hasNext()

The `hasNext()` method returns whether or not there are any documents remaining to be read from the collection. The File System Collection Reader's `hasNext()` method is very simple. It just checks if there are any more files left to be read:

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

getNext(CAS)

The `getNext()` method reads the next document from the collection and populates a CAS. In the simple case, this amounts to reading the file and calling the CAS's `setDocumentText` method. The example File System Collection Reader is slightly more complex. It first checks for a CAS Initializer. If the CPE includes a CAS Initializer, the CAS Initializer is used to read the document, and `initialize()` the CAS. If the CPE does not include a CAS Initializer, the File System Collection Reader reads the document and sets the document text in the CAS.

The File System Collection Reader also stores additional metadata about the document in the CAS. In particular, it sets the document's language in the special built-in feature structure `uima.tcas.DocumentAnnotation` (see *Chapter 23* ***CAS Reference*** for details about this built-in type) and creates an instance of

com.ibm.uima.examples.SourceDocumentInformation, which stores information about the document's source location. This information may be useful to downstream components such as CAS Consumers. Note that the type system descriptor for this type can be found in com.ibm.uima.examples.SourceDocumentInformation.xml.

The getNext() method for the File System Collection Reader looks like this:

```
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++);
  FileInputStream fis = new FileInputStream(file);
  try
  {
    //if there's a CAS Initializer, call it
    if (getCasInitializer() != null)
    {
      getCasInitializer().initializeCas(fis, aCAS);
    }
    else  //No CAS Initializer, so read file and set document
          //text here
    {
      byte[] contents = new byte[(int)file.length() ];
      fis.read( contents );
      String text;
      if (mEncoding != null)
      {
        text = new String(contents, mEncoding);
      }
      else
      {
        text = new String(contents);
      }
      //put document in CAS (assume TCAS)
      jcas.setDocumentText(text);
    }
  }
  finally
  {
    if (fis != null)
      fis.close();
  }
```

```
    //set language if it was explicitly specified as a
    //configuration parameter
    if (mLanguage != null)
    {
      ((DocumentAnnotation)jcas.getDocumentAnnotationFs())
          .setLanguage(mLanguage);
    }

    //Also store file location information in CAS metadata.
    //This information is critical
    //if CAS Consumers will need to know where the
    //original document contents are located.

    //For example, the Semantic Search CAS Indexer writes this
    //information into the search index that it creates, which allows
    //applications that use the search index to
    //locate the documents that satisfy their semantic queries.

    SourceDocumentInformation srcDocInfo =
      new SourceDocumentInformation(jcas);
    srcDocInfo.setUri(file.getAbsoluteFile().toURL().toString());
    srcDocInfo.setOffsetInSource(0);
    srcDocInfo.setDocumentSize((int)file.length());
    srcDocInfo.addToIndexes();
  }
```

The Collection Reader can create additional annotations in the CAS at this point, in the same way that annotators create annotations. However, if you are doing complex initialization of the CAS, it may be better to use a CAS Initializer as described in Section 5.4.2 .

getProgress()

The Collection Reader is responsible for returning progress information; that is, how much of the collection has been read thus far and how much remains to be read. The framework defines progress very generally; the Collection Reader simply returns an array of `Progress` objects, where each object contains three fields – the amount already completed, the total amount (if known), and a unit (e.g. entities (documents), bytes, or files). The method returns an array so that the Collection Reader can report progress in multiple different units, if that information is available. The File System Collection Reader's `getProgress()` method looks like this:

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

In this particular example, the total number of files in the collection is known, but the total size of the collection is not known. As such, a `ProgressImpl` object for `Progress.ENTITIES` is returned, but a `ProgressImpl` object for `Progress.BYTES` is not.

close()

The close method is called when the Collection Reader is no longer needed. The Collection Reader should then release any resources it may be holding. The FileSystemCollectionReader does not hold resources and so has an empty implementation of this method:

```
public void close() throws IOException { }
```

Optional Methods

The following methods may be implemented:

reconfigure()

This method is called if the Collection Reader's configuration parameters change.

typeSystemInit()

If you are only setting the document text in the CAS, or if you are using the JCas (recommended, as in the current example), you do not have to implement this method. If you are directly using the CAS API, this method is used in the same way as it is used for an annotator – see *Chapter 4* ***Annotator and Analysis Engine Developer's*** Guide for more information.

XML Descriptor

You can use the Component Description Editor to create and / or edit the File System Collection Reader's descriptor. Here is its descriptor (abbreviated somewhat to fit on a page), which is very similar to an Analysis Engine descriptor:

```
<collectionReaderDescription
xmlns="http://uima.watson.ibm.com/resourceSpecifier">
    <frameworkImplementation>com.ibm.uima.java</frameworkImplementation>
    <implementationName>
        com.ibm.uima.util.FileSystemCollectionReader
    </implementationName>
    <processingResourceMetaData>
        <name>File System Collection Reader</name>
        <description>Reads text files from the filesystem</description>
        <version>1.0</version>
        <vendor>IBM</vendor>
        <configurationParameters>
            <configurationParameter>
                <name>InputDirectory</name>
                <description>Directory containing input files</description>
```

```
                <type>String</type>
                <multiValued>false</multiValued>
                <mandatory>true</mandatory>
            </configurationParameter>

            <!-- Other Configuration Parameters Omitted -->
        </configurationParameters>

        <configurationParameterSettings>
            <nameValuePair>
                <name>InputDirectory</name>
                <value>
                    <string>C:\program files\uima\data</string>
                </value>
            </nameValuePair>
        </configurationParameterSettings>

        <!-- Type System of CASes returned by this Collection Reader -->

        <typeSystemDescription>
            <imports>
                <import
name="com.ibm.uima.examples.SourceDocumentInformation"/>
            </imports>
        </typeSystemDescription>

        <capabilities>
           <capability>
              <inputs/>
               <outputs>
                 <type allAnnotatorFeatures="true">
                    com.ibm.uima.examples.SourceDocumentInformation
                 </type>
               </outputs>
           </capability>
        </capabilities>
    </processingResourceMetaData>
</collectionReaderDescription>
```

5.4.2 Developing CAS Initializers

Although Collection Readers can directly write to the CAS, it is best that they do so only for simple cases. If the task of populating the CAS from a raw document is complex and might be reusable with other data collections, then it is worthwhile to encapsulate it in a separate CAS Initializer component.

An example where the use of a CAS Initializer is ideal is a scenario where the documents in the collection contain inline HTML or XML markup. Since Analysis Engines often ingest plain-text documents with stand-off annotations, it is necessary to translate the inline HTML or XML markup into this form. For example, an HTML document with inline <p> and <h1> tags could be translated into a CAS with a plain-text document and stand-off `Paragraph` and `Heading` annotations. Since this HTML parsing logic could be used regardless of the source of the HTML documents (e.g. a

file system, a web connection, or a relational database), it would be ideal to implement this using a CAS Initializer that could be plugged-in to multiple Collection Readers.

A CAS Initializer Java class must implement the interface `com.ibm.uima.collection.CasInitializer`, and will also generally extend from the convenience base class `com.ibm.uima.collection.CasInitializer_ImplBase`. A CAS Initializer also must have an XML descriptor, which has the exact same form as a Collection Reader Descriptor except that the outer tag is `<casInitializerDescription>`.

CAS Initializers have optional `initialize()`, `reconfigure()`, and `typeSystemInit()` methods, which perform the same functions as they do for Collection Readers. The only required method for a CAS Initializer is `initializeCas(Object, CAS)`. This method takes the raw document (for example, an `InputStream` object from which the document can be read) and a CAS, and populates the CAS from the document.

An example CAS Initializer is implemented by the class `com.ibm.uima.examples.cpe.SimpleXmlCasInitializer`. The SimpleXmlCasInitializer shows how a CAS Initializer can invoke an XML Parser on the raw document. In this very simple example the only thing extracted from the XML document is the text to be processed. You can configure the `SimpleXmlCasInitializer` with the name of an XML tag that contains the text; it will then filter out the rest of the document.

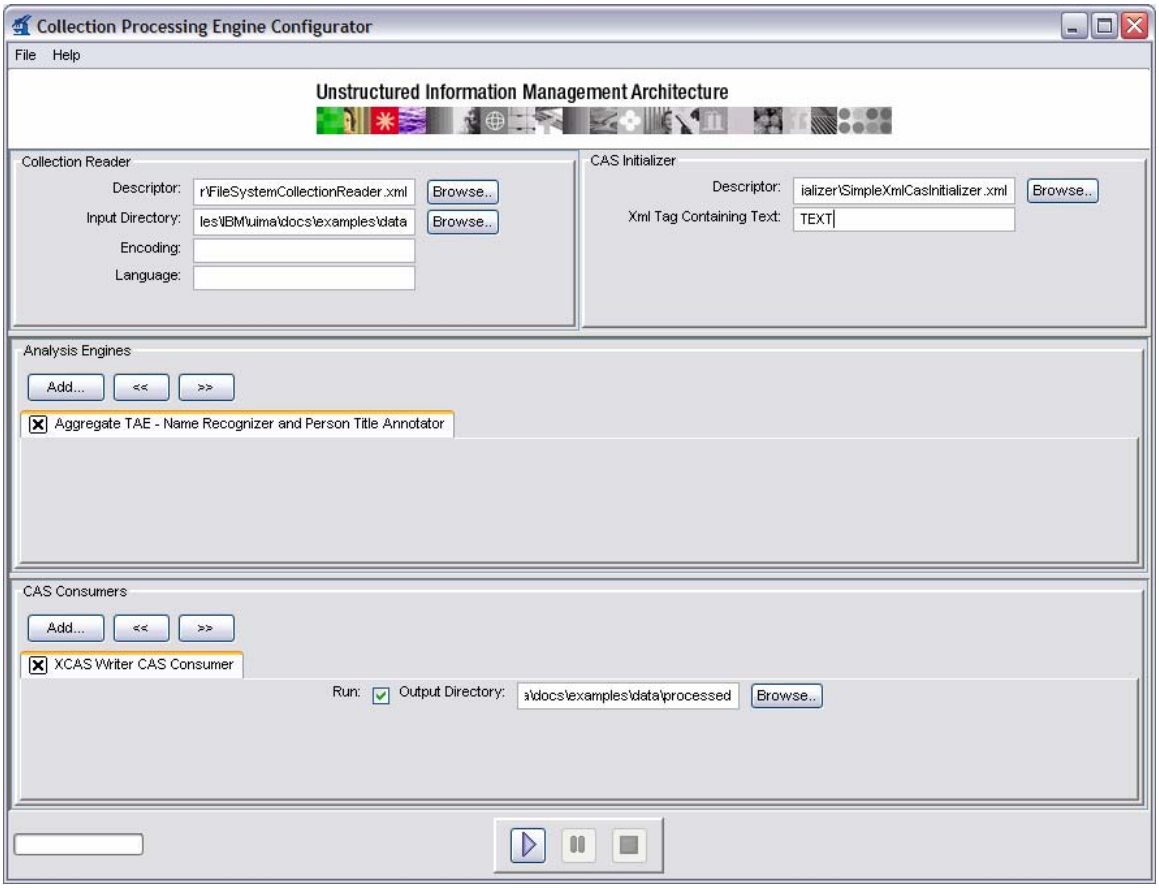
Here is the implementation of the `initializeCas()` method for this example:

```
public void initializeCas(Object aObj, CAS aCAS)
  throws CollectionException, IOException
{
  //build SAX InputSource object from InputStream supplied
  //by the CollectionReader
  InputSource inputSource;
  if (aObj instanceof InputStream)
  {
    inputSource = new InputSource((InputStream)aObj);
  }
  else
  {
      throw new CollectionException(
                  CollectionException.INCORRECT_INPUT_TO_CAS_INITIALIZER,
                  new Object[]{InputStream.class.getName(),
                  Obj.getClass().getName()});
  }
  //create SAX ContentHandler that populates CAS
  SaxHandler handler = new SaxHandler(aCAS);
  //parse
  try
  {
    SAXParser parser = mParserFactory.newSAXParser();
```

```
      XMLReader reader = parser.getXMLReader();
      reader.setContentHandler(handler);
      reader.parse(inputSource);
    }
    catch (Exception e)
    {
      throw new CollectionException(e);
    }
  }
```

The `SaxHandler` class referenced here is an inner class that does the actual work of extracting the text from the specified XML element. For the full implementation, see the example code under `docs/examples`.

To try out the CAS Initializer, use the CPE Configurator GUI as described in section 10.3 . However, in addition to selecting a Collection Reader, Analysis Engine, and CAS Consumer as described in that section, also select a CAS Initializer by using the "Browse" button on the CAS Initializer panel. Browse to the `%UIMA_HOME%/docs/examples/descriptors/cas_initializer` directory and select the `SimpleXmlCasInitializer.xml` descriptor file. Then, set the "Xml Tag Containing Text" parameter to the value TEXT. The CPE Configurator should then look like this:



The SimpleXmlCasInitializer only works with XML documents, so you will need to change the "Input Directory" parameter of the Collection Reader by clicking the "Browse" button and selecting the %UIMA_HOME%/docs/examples/data/xml directory. Then click the play button. Once processing has completed, you can use the XCAS Annotation Viewer, as described in *Chapter 17* , to view the results. Notice that only the contents of the <TEXT> elements in the original source documents appear in the analysis results.

It is important to note that CAS Initializers will only work with Collection Readers that are designed to use them. The Collection Reader needs to call its getCasInitializer() method to see if a CAS Initializer has been supplied, and call the CAS Initializer's initializeCas() method, rather than setting up the CAS itself. Our File System Collection Reader example from section 5.4.1 optionally uses a CAS Initializer as follows:

```
//if there is a CAS Initializer, call it
if (getCasInitializer() != null)
{
  getCasInitializer().initializeCas(fis, aCAS);
}
else  //No CAS Initializer, so read file and set document text ourselves
{
  ...
}
```

When you write your own Collection Reader, in the description element of your Collection Reader's descriptor you should document whether your Collection Reader supports (or requires) a CAS Initializer, so that users will know how to configure their CPE properly.

5.4.3 Developing CAS Consumers

A CAS Consumer receives each CAS after it has been analyzed by the Analysis Engine. CAS Consumers typically do not update the CAS; they typically extract data from the CAS and persist selected information to aggregate data structures such as search engine indexes or databases.

A CAS Consumer Java class must implement the interface com.ibm.uima.collection.CasConsumer, and will also generally extend from the convenience base class com.ibm.uima.collection.CasConsumer_ImplBase. A CAS Consumer also must have an XML descriptor, which has the exact same form as a Collection Reader Descriptor except that the outer tag is <casConsumerDescription>.

CAS Consumers have optional initialize(), reconfigure(), and typeSystemInit() methods, which perform the same functions as they do for Collection Readers and CAS Initializers. The only required method for a CAS Consumer is processCas(CAS),

which is where the CAS Consumer does the bulk of its work (i.e., consume the CAS).

The `CasConsumer` interface additionally defines batch and collection level processing methods. The CAS Consumer can implement the `batchProcessComplete()` method to perform processing that should occur at the end of each batch of CASes. Similarly, the CAS Consumer can implement the `collectionProcessComplete()` method to perform any collection level processing at the end of the collection.

A very simple example of a CAS Consumer, which writes an XML representation of the CAS to a file, is the XCAS Writer CAS Consumer. The Java code is in the class `com.ibm.uima.examples.cpe.XCasWriterCasConsumer` and the descriptor is in `%UIMA_HOME%\docs\examples\descriptors\cas_consumer\XCasWriterCasConsumer.xml`.

Required Methods

When extending the convenience class `com.ibm.uima.collection.CasConsumer_ImplBase`, the following abstract methods must be implemented:

initialize()

The `initialize()` method is called by the framework when the CAS Consumer is first created. `CasConsumer_ImplBase` actually provides a default implementation of this method (i.e., it is not abstract), so you are not strictly required to implement this method. However, a typical CAS Consumer will implement this method to obtain parameter values and perform various initialization steps.

In this method, the CAS Consumer can access the values of its configuration parameters and perform other initialization logic. The example XCAS Writer CAS Consumer reads its configuration parameters and sets up the output directory:

```
public void initialize() throws ResourceInitializationException
{
  mDocNum = 0;
  mOutputDir = new File((String)getConfigParameterValue(PARAM_OUTPUTDIR));
  if (!mOutputDir.exists()) {
    mOutputDir.mkdirs();
  }
}
```

processCas()

The `processCas()` method is where the CAS Consumer does most of its work. In our example, the XCAS Writer CAS Consumer obtains an iterator over the document metadata in the CAS (in the SourceDocumentInformation feature structure, which is created by the File System Collection Reader) and extracts the URI for the current

document. From this the output filename is constructed in the output directory and a subroutine (`writeXCas`) is called to generate the output file. The `writeXCas` subroutine uses the `XCASSerializer` class provided with the UIMA SDK to serialize the CAS to the output file (see the example source code for details).

```
public void processCas(CAS aCAS) throws ResourceProcessException
{
  JCas jcas;
  try {
    jcas = aCAS.getJCas();
  }
  catch (CASException e) {
    throw new ResourceProcessException(e);
  }

  // retrieve the filename of the input file from the CAS

  FSIterator it = jcas.getJFSIndexRepository().
                      getAnnotationIndex(
                        SourceDocumentInformation.type).iterator();
  File outFile = null;
  if (it.hasNext()) {
    SourceDocumentInformation fileLoc =
                (SourceDocumentInformation)it.next();
    File inFile;
    try {
      inFile = new File(new URL(fileLoc.getUIR()).getPath());
      outFile = new File(mOutputDir, inFile.getName());
    } catch (MalformedURLException e1) {
        // invalid URL, use default processing below
    }
  }
  if (null == outFile) {
    outFile = new File(mOutputDir, "doc"+ mDocNum++);
  }
  // serialize XCAS and write to output file
  try {
    writeXCas(jcas.getCas(), outFile);
  }
  catch (IOException e) {
    throw new ResourceProcessException(e);
  }
  catch (SAXException e) {
    throw new ResourceProcessException(e);
  }
}
```

Optional Methods

The following methods are optional in a CAS Consumer, though they are often used.

batchProcessComplete()

The framework calls the batchProcessComplete() method at the end of each batch of CASes. This gives the CAS Consumer an opportunity to perform any batch level processing. Our simple XCAS Writer CAS Consumer does not perform any batch level processing, so this method is empty. Batch size is set in the Collection Processing Engine descriptor.

collectionProcessComplete()

The framework calls the collectionProcessComplete() method at the end of the collection (i.e., when all objects in the collection have been processed). At this point in time, no CAS is passed in as a parameter. This gives the CAS Consumer an opportunity to perform collection processing over the entire set of objects in the collection. Our simple XCAS Writer CAS Consumer does not perform any collection level processing, so this method is empty.

5.5 Deploying a CPE

The CPM provides a number of service and deployment options that cover instantiation and execution of CPEs, error recovery, and local and distributed deployment of the CPE components. The behavior of the CPM (and correspondingly, the CPE) is controlled by various options and parameters set in the CPE descriptor. The current version of the CPE Configurator tool, however, supports only default error handling and deployment options. To change these options, you must manually edit the CPE descriptor – a potentially error prone task.

Eventually the CPE Configurator tool will support configuring these options and a detailed tutorial for these settings will be provided. In the meantime, we provide only a high-level, conceptual overview of these advanced features in the rest of this chapter, and refer the advanced user to *Chapter 21* ***Collection Processing Engine Descriptor Reference*** for details on setting these options in the CPE Descriptor.

Figure 13 shows a logical view of how an application uses the UIMA framework to instantiate a CPE from a CPE descriptor. The CPE descriptor identifies the CPE components (referencing their corresponding descriptors) and specifies the various options for configuring the CPM and deploying the CPE components.

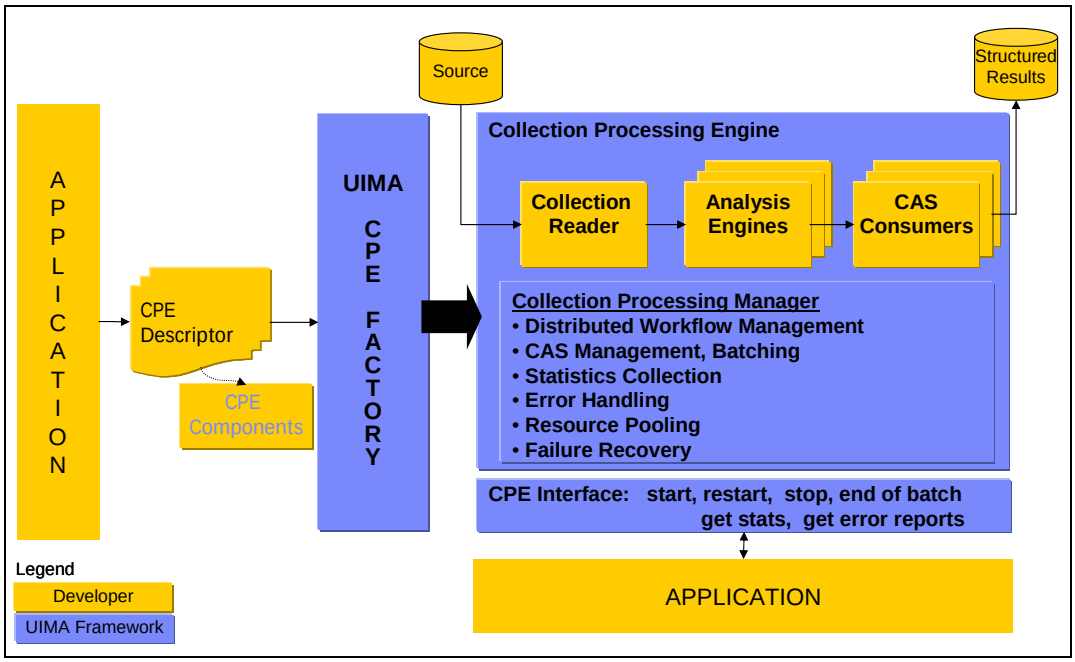


Figure 13 CPE instantiation

There are three deployment modes for CAS Processors (Analysis Engines and CAS Consumers) in a CPE:

1. **Integrated** (runs in the same Java instance as the CPM)
2. **Managed** (runs in a separate process on the same machine), and
3. **Non-managed** (runs in a separate process, perhaps on a different machine).

An integrated CAS Processor runs in the same JVM as the CPE. A managed CAS Processor runs in a separate process from the CPE, but still on the same computer. The CPE controls startup, shutdown, and recovery of a managed CAS Processor. A non-managed CAS Processor runs as a service and may be on the same computer as the CPE or on a remote computer. A non-managed CAS Processor ***service*** is started and managed independently from the CPE.

For both managed and non-managed CAS Processors, the CAS must be transmitted between separate processes and possibly between separate computers. This is accomplished using *Vinci*, a communication protocol used by the CPM that ships with the UIMA SDK. Vinci handles service naming and location and data transport (see *6.6.2* ***How to Deploy a UIMA Component as a Vinci Service*** for more information). Service naming and location are provided by a *Vinci Naming Service*, or *VNS*. For managed CAS Processors, the CPE uses its own internal VNS. For non-managed CAS Processors, a separate VNS must be running.

Note: The UIMA SDK also supports using unmanaged remote services via the web-standard SOAP communications protocol (see ***How to Deploy a UIMA Component as a SOAP Web Service*** on page 6-148). This approach is based on a proxy implementation, where the proxy is essentially running in an integrated mode. To use this approach with the CPM, use the Integrated mode, with the component being an Aggregate which, in turn, connects to a remote service.

The CPE Configurator tool currently only supports constructing CPEs that deploy CAS Processors in integrated mode. To deploy CAS Processors in any other mode, the CPE descriptor must be edited by hand (better tooling support is being worked on). Details on the CPE descriptor and the required settings for various CAS Processor deployment modes can be found in *Chapter 21* ***Collection Processing Engine Descriptor Reference***. In the following sections we merely summarize the various CAS Processor deployment options.

5.5.1 Deploying Managed CAS Processors

Managed CAS Processor deployment is shown in Figure 14. A managed CAS Processor is deployed by the CPE as a Vinci service. The CPE manages the lifecycle of the CAS Processor including service launch, restart on failures, and service shutdown. A managed CAS Processor runs on the same machine as the CPE, but in a separate process. This provides the necessary fault isolation for the CPE to protect it from non-robust CAS Processors. A fatal failure of a managed CAS Processor does not threaten the stability of the CPE.

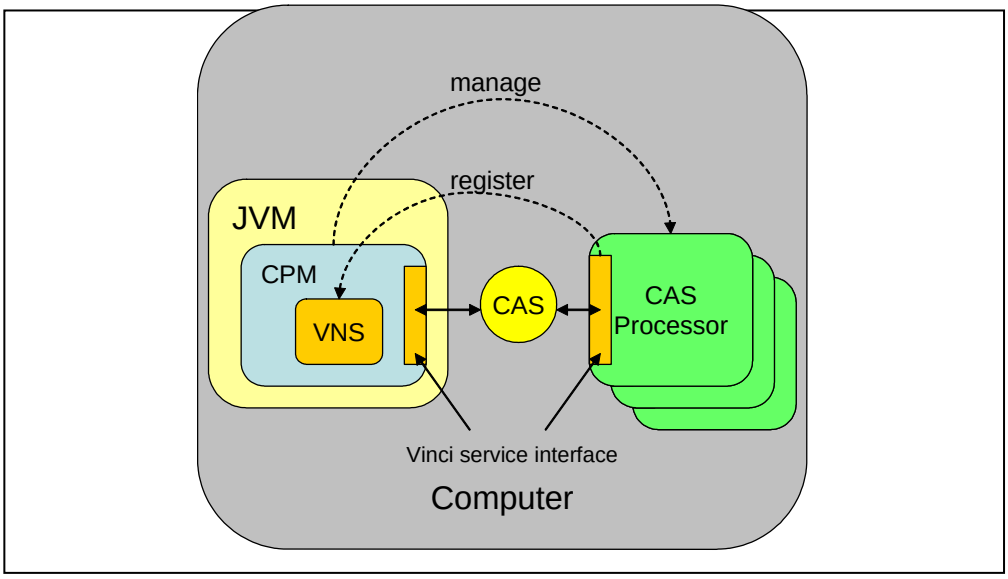


Figure 14 CPE with managed CAS Processors

The CPE communicates with managed CAS Processors using the Vinci communication protocol. A CAS Processor is launched as a Vinci service and its `process()` method is invoked remotely via a Vinci command. The CPE uses its own

internal VNS to support managed CAS processors. The VNS, by default, listens on port 9005. If this port is not available, the VNS will increment its listen port until it finds one that is available. All managed CAS Processors are internally configured to "talk" to the CPE managed VNS. This internal VNS is transparent to the end user launching the CPE.

To deploy a managed CAS Processor, the CPE deployer must change the CPE descriptor. The following is a section from the CPE descriptor that shows an example configuration specifying a managed CAS Processor.

```
<casProcessor deployment="local" name="Meeting Detector TAE">
  <descriptor>
    <include href="deploy/vinci/Deploy_MeetingDetectorTAE.xml"/>
  </descriptor>
  <runInSeparateProcess>
    <exec dir="." executable="java">
      <env key="CLASSPATH"
value="src;C:/Program Files/IBM/uima/lib/uima_core.jar;C:/Program
Files/IBM/uima/lib/uima_cpe.jar;C:/Program
Files/IBM/uima/lib/uima_examples.jar;C:/Program
Files/IBM/uima/lib/uima_adapter_vinci.jar;C:/Program
Files/IBM/uima/lib/uima_jcas_builtin_types.jar;C:/Program
Files/IBM/uima/lib/vinci/jVinci.jar;C:/Program
Files/IBM/uima/lib/xml.jar"/>
      <arg>-DLOG=C:/Temp/service.log</arg>
      <arg>com.ibm.uima.reference_impl.collection.
         service.vinci.VinciCasObjectProcessorService_impl</arg>
      <arg>${descriptor}</arg>
    </exec>
  </runInSeparateProcess>
  <deploymentParameters/>
  <filter/>
  <errorHandling>
    <errorRateThreshold action="terminate" value="1/100"/>
    <maxConsecutiveRestarts action="terminate" value="3"/>
    <timeout max="100000"/>
  </errorHandling>
  <checkpoint batch="10000"/>
</casProcessor>
```

See Chapter 21 *Collection Processing Engine Descriptor Reference* on page 21-293 for details and required settings.

5.5.2 Deploying Non-managed CAS Processors

Non-managed CAS Processor deployment is shown in Figure 15. In non-managed mode, the CPE supports connectivity to CAS Processors running on local or remote computers using Vinci. Non-managed processors are different from managed processors in two aspects:

1. Non-managed processors are neither started nor stopped by the CPE.

2. Non-managed processors use an independent VNS, also neither started nor stopped by the CPE.

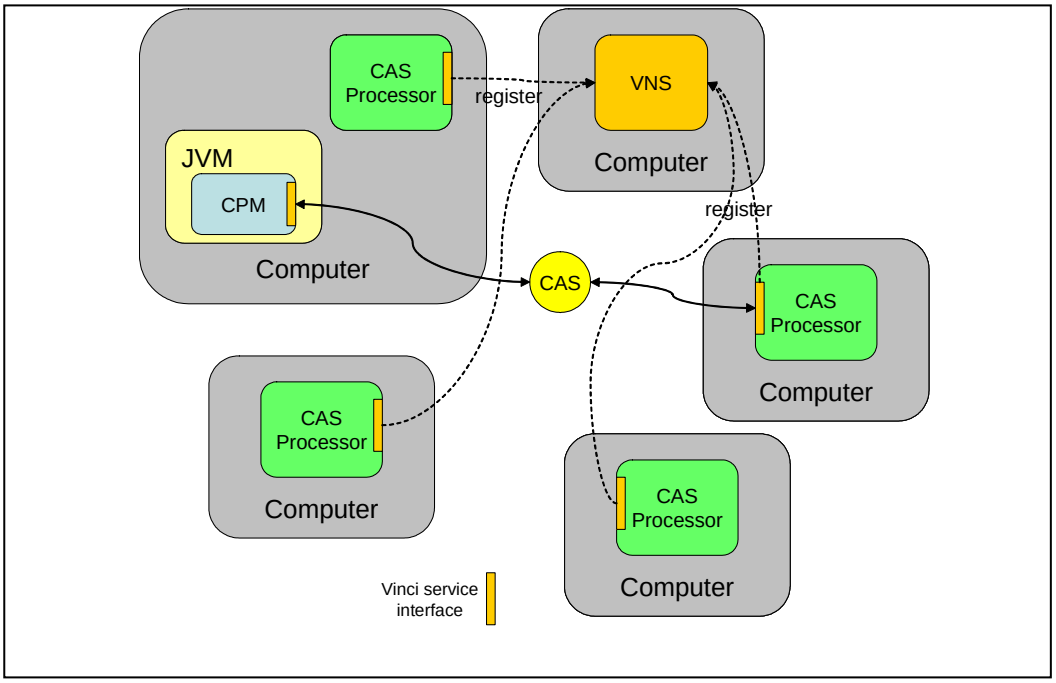


Figure 15 CPE with non-managed CAS Processors

While non-managed CAS Processors provide the same level of fault isolation and robustness as managed CAS Processors, error recovery support for non-managed CAS Processors is much more limited. In particular, the CPE cannot restart a non-managed CAS Processor after an error.

Non-managed CAS Processors also require a separate Vinci Naming Service running on the network. This VNS must be manually started and monitored by the end user or application. Instructions for running a VNS can be found in section 6.6.5 ***Starting VNS***, on page 6-154.

To deploy a non-managed CAS Processor, the CPE deployer must change the CPE descriptor. The following is a section from the CPE descriptor that shows an example configuration for the non-managed CAS Processor.

```
<casProcessor deployment="remote" name="Meeting Detector TAE">
  <descriptor>
    <include href=
"descriptors/vinciService/MeetingDetectorVinciService.xml"/>
  </descriptor>
  <deploymentParameters/>
  <filter/>
  <errorHandling>
    <errorRateThreshold action="terminate" value="1/100"/>
```

```
    <maxConsecutiveRestarts action="terminate" value="3"/>
    <timeout max="100000"/>
  </errorHandling>
  <checkpoint batch="10000"/>
</casProcessor>
```

See Chapter 21 ***Collection Processing Engine Descriptor Reference*** on page 21-293 for details and required settings.

5.5.3 Deploying Integrated CAS Processors

Integrated CAS Processors are shown in Figure 16. Here the CAS Processors run in the same JVM as the CPE, just like the Collection Reader and CAS Initializer. This deployment method results in minimal CAS communication and transport overhead as the CAS is shared in the same process space of the JVM. However, a CPE running with all integrated CAS Processors is limited in scalability by the capability of the single computer on which the CPE is running. There is also a stability risk associated with integrated processors because a poorly written CAS Processor can cause the JVM, and hence the entire CPE, to abort.

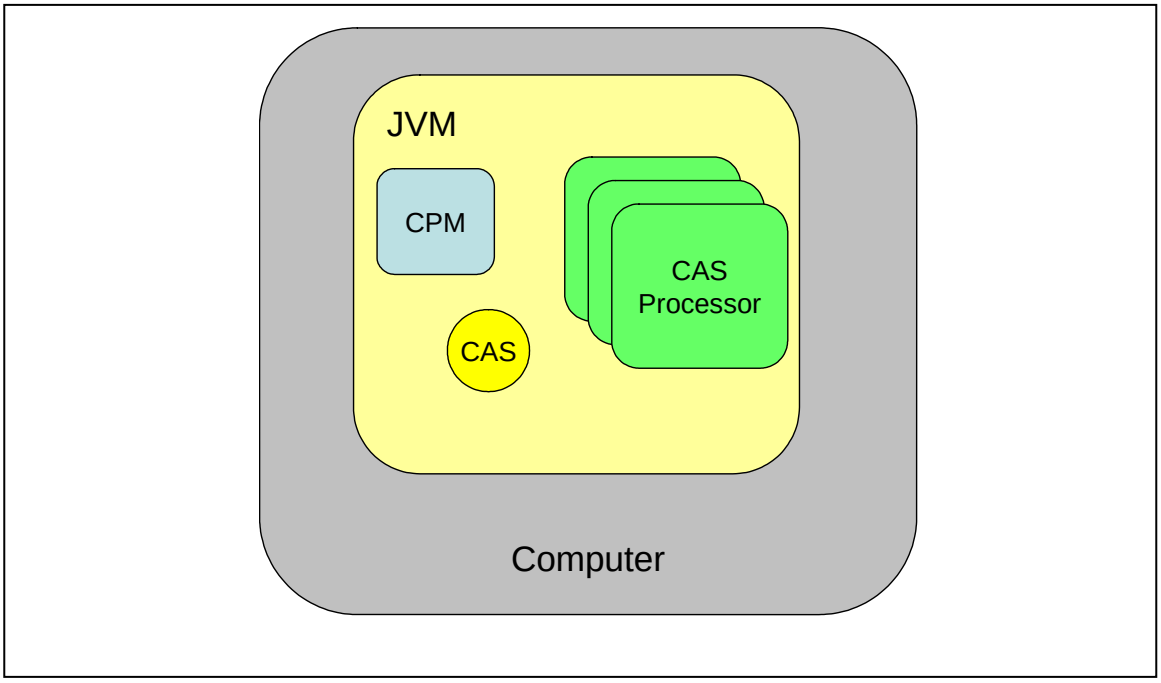


Figure 16 CPE with integrated CAS Processor

The following is a section from a CPE descriptor that shows an example configuration for the integrated CAS Processor.

```
<casProcessor deployment="integrated" name="Meeting Detector TAE">
  <descriptor>
    <include href="descriptors/tutorial/ex4/MeetingDetectorTAE.xml"/>
  </descriptor>
  <deploymentParameters/>
```

```
    <filter/>
    <errorHandling>
      <errorRateThreshold action="terminate" value="100/1000"/>
      <maxConsecutiveRestarts action="terminate" value="30"/>
      <timeout max="100000"/>
    </errorHandling>
    <checkpoint batch="10000"/>
  </casProcessor>
```

See Chapter 21 ***Collection Processing Engine Descriptor Reference*** on page 21-293 for details and required settings.

5.6 Collection Processing Examples

The UIMA SDK includes a set of examples illustrating the three modes of deployment, integrated, managed, and non-managed. These are in the `/docs/examples/descriptors/collection_processing_engine` directory. There are three CPE descriptors that run an example annotator (the Meeting Finder) in these modes.

To run either the integrated or managed examples, use the `runCPE` script in the /bin directory of the UIMA installation, passing the appropriate CPE descriptor as an argument.

> **Note:** The `runCPE` script ***must*** be run from the `%UIMA_HOME%\docs\examples` directory, because it uses relative path names that are resolved relative to this working directory. For instance,

```
runCPE descriptors\collection_processing_engine\MeetingFinderCPE_Integrated.xml
```

If you installed the examples into Eclipse, you can run directly from Eclipse by creating a run configuration. To do this, highlight the SimpleRunCPE.java source file in the examples src/com/ibm/uima/examples/cpe directory, and then

1. pick the menu Run -> Run... Select
2. click "Java Application" and press "New"
3. click on the Arguments panel, and insert a path to the appropriate CPE descriptor in the "Program Arguments" box by typing, for instance: `descriptors/collection_processing_engine/MeetingFinderCPE_Integrated.xml`
4. Then press "Run"

To run the non-managed example, there are some additional steps.

1. Start a VNS service by running the `startVNS` script in the `/bin` directory.

2. Deploy the Meeting Detector Analysis Engine as a Vinci service, by running the `startVinceService` script in the `/bin` directory, and passing it the location of the descriptor to deploy, in this case `%UIMA_HOME%/docs/examples/deploy/vinci/Deploy_MeetingDetectorTAE.xml`
3. Now, run the runCPE script, passing it the CPE for the non-managed version (`%UIMA_HOME%/docs/examples/descriptors/collection_processing_engine/MeetingFinderCPE_NonManaged.xml`).

This assumes that the Vinci Naming Service, the runCPE application, and the `MeetingDetectorTAE` service are all running on the same machine. Most of the scripts that need information about VNS will look for values to use in environment variables VNS_HOST and VNS_PORT; these default to "localhost" and "9000". You may set these to appropriate values before running the scripts, as needed.

Alternatively, you can edit the scripts and/or the XML files top specify alternatives for the VNS_HOST and VNS_PORT. For instance, if the `runCPE` application is running on a different machine from the Vinci Naming Service, you can edit the `MeetingFinderCPE_NonManaged.xml` and change the vnsHost parameter:

```
<parameter name="vnsHost"  value="localhost" type="string"/>
```

to specify the VNS host instead of "localhost". Another example: if the `MeetingDetectorTAE` service is deployed on a different machine from the Vinci Naming Service, you can edit the the `startVinciService` script and specify where to find the Vinci Naming Service: `-DVNS_HOST=<your VNS hostname>.`

Chapter 6 Application Developer's Guide

This chapter describes how to develop an application using the Unstructured Information Management Architecture (UIMA). The term *application* describes a program that provides end-user functionality. A UIMA application incorporates one or more UIMA components such as Analysis Engines, Collection Processing Engines, a Search Engine, and/or a Document Store and adds application-specific logic and user interfaces.

6.1 The UIMAFramework Class

An application developer's starting point for accessing UIMA framework functionality is the `com.ibm.uima.UIMAFramework` class. The following is a short introduction to some important methods on this class. Several of these methods are used in examples in the rest of this chapter. For more details, see the JavaDocs (in the docs/api directory of the UIMA SDK).

- UIMAFramework.getXMLParser(): Returns an instance of the UIMA XML Parser class, which then can be used to parse the various types of UIMA component descriptors. Examples of this can be found in the remainder of this chapter.
- UIMAFramework.produceXXX(ResourceSpecifier): There are various produce methods that are used to create different types of UIMA components from their descriptors. The argument type, ResourceSpecifier, is the base interface that subsumes all types of component descriptors in UIMA. You can get a ResourceSpecifier from the XMLParser. Examples of produce methods are:
  - produceAnalysisEngine
  - produceCasConsumer
  - produceCasInitializer
  - produceCollectionProcessingEngine
  - produceCollectionReader
  - produceTAE

  There are other variations of each of these methods that take additional, optional arguments. See the JavaDocs for details.
- UIMAFramework.getLogger(<optional-logger-name>): Gets a reference to the UIMA Logger, to which you can write log messages. If no logger name is passed, the name of the returned logger instance is "com.ibm.uima".
- UIMAFramework.getVersionString(): Gets the number of the UIMA version you are using.

- UIMAFramework.newDefaultResourceManager(): Gets an instance of the UIMA ResourceManager. The key method on ResourceManager is setDataPath, which allows you to specify the location where UIMA components will go to look for their external resource files. Once you've obtained and initialized a ResourceManager, you can pass it to any of the produceXXX methods.

6.2 Using Analysis Engines

This section describes how to add analysis capability to your application by using Analysis Engines developed using the UIMA SDK. An *Analysis Engine (AE)* is a component that analyzes artifacts (e.g. documents) and infers information about them. A *TAE* is a specialization of an Analysis Engine that analyzes artifacts, such as a text document. The examples in this chapter primarily discuss TAEs, as this has been the most common use of the SDK.

TAE formerly was an acronym for Text Analysis Engine. Today, the TAE can be used for text, but also for other kinds of artifacts to be analyzed. Some of the XML elements for TAEs are still tagged with the textAnalysisEngine name, but you should think of this now as more generally applying to the analysis of arbitrary artifacts. This is more fully described in Chapter 7 ***Developing Applications using Multiple Subjects of Analysis*** on page 7-158.

An Analysis Engine consists of two parts - Java classes (typically packaged as one or more JAR files) and *AE descriptors* (one or more XML files). You must put the Java classes in your application's class path, but thereafter you will not need to directly interact with them. The UIMA framework insulates you from this by providing standard AnalysisEngine and TextAnalysisEngine interfaces.

The AE descriptor XML files contain the configuration settings for the Analysis Engine as well as a description of the AE's input and output requirements. You may need to edit these files in order to configure the AE appropriately for your application - the supplier of the AE may have provided documentation (or comments in the XML descriptor itself) about how to do this.

6.2.1 Instantiating an Analysis Engine

The following code shows how to instantiate a TAE from its XML descriptor:

```
{
  //get Resource Specifier from XML file or PEAR
  XMLInputSource in = new XMLInputSource("MyDescriptor.xml");
  ResourceSpecifier specifier =
    UIMAFramework.getXMLParser().parseResourceSpecifier(in);
```

```
  //create TAE here
  TextAnalysisEngine tae =
    UIMAFramework.produceTAE(specifier);
}
```

Creating a generic Analysis Engine (which may process something other than text documents) is similar: just replace `TextAnalysisEngine` with `AnalysisEngine` and `produceTAE` with `produceAnalysisEngine`.

The first two lines parse the XML descriptor (for AEs with multiple descriptor files, one of them is the "main" descriptor - the AE documentation should indicate which it is). The result of the parse is a `ResourceSpecifier` object. The third line of code invokes a static factory method `UIMAFramework.produceTAE`, which takes the specifier and instantiates a `TextAnalysisEngine` object.

There is one caveat to using this approach - the Analysis Engine instance that you create will not support multiple threads running through it concurrently. If you need to support this, see section 6.2.6 .

6.2.2 Analyzing Text Documents

There are two ways to use the TAE interface to analyze documents. You can either use the *JCas* interface, which is described in detail by *Chapter 24* ***JCas Reference*** or you can directly use the *TCAS* interface, which is described in detail in *Chapter 23* ***CAS Reference***. Besides text documents, other kinds of artifacts can also be analyzed; see Chapter 7 ***Developing Applications using Multiple Subjects of Analysis*** on page 7-158 for more information.

The basic structure of your application will look similar in both cases:

Using the JCas

```
{
  //create a JCas
  JCas jcas = tae.newJCas();

  //analyze a document
  jcas.setDocumentText(doc1text);
  tae.process(jcas);
  doSomethingWithResults(jcas);
  jcas.reset();

  //analyze another document
  jcas.setDocumentText(doc2text);
  tae.process(jcas);
  doSomethingWithResults(jcas);
  jcas.reset();
  ...
  //done
```

```
  tae.destroy();
}
```

Using the TCAS

```
{
  //create a TCAS
  TCAS tcas = tae.newTCAS();

  //analyze a document
  tcas.setDocumentText(doc1text);
  tae.process(tcas);
  doSomethingWithResults(tcas);
  tcas.reset();

  //analyze another document
  tcas.setDocumentText(doc2text);
  tae.process(tcas);
  doSomethingWithResults(tcas);
  tcas.reset();
  ...
  //done
  tae.destroy();
}
```

First, you create the TCAS or JCas that you will use. Then, you repeat the following four steps for each document:

- Put the document text into the TCAS or JCas.
- Call the TAE's process method, passing the TCAS or JCas as an argument
- Do something with the results that the TAE has added to the TCAS or JCas
- Call the TCAS's or JCas's reset() method to prepare for another analysis

6.2.3 Analyzing Non-Text Artifacts

Analyzing non-text artifacts is similar to analyzing text documents. The main difference is that instead of using the `setDocumentText` method, you need to use the Sofa APIs to create an artifact plus (perhaps multiple) views of it – each view corresponding to a particular TCAS. See the section ***Sofas and TCAS Views*** on page 7-162 for details.

6.2.4 Accessing Analysis Results using the JCas

See:

- *Chapter 4.1.3* ***Developing Your Annotator Code***
- *Chapter 24* ***JCas Reference***
- The source code for `com.ibm.uima.examples.AnnotationFilter`, which is in `docs\examples\src`.

- The JavaDocs for `com.ibm.uima.jcas.impl.JCas`.

6.2.5 Accessing Analysis Results using the CAS

See:

- *Chapter 23 **CAS Reference***
- The source code for `com.ibm.uima.examples.PrintAnnotations`, which is in `docs\examples\src`.
- The JavaDocs for the `com.ibm.uima.cas and com.ibm.uima.cas.text` packages.

6.2.6 Multi-threaded Applications

The simplest way to use an AE in a multi-threaded environment is to use the Java synchronized keyword to ensure that only one thread is using an AE at any given time. For example:

```
public class MyApplication
{
  private AnalysisEngine mAnalysisEngine;
  private CAS mTCAS;

  public MyApplication()
  {
    //get Resource Specifier from XML file or PEAR
    XMLInputSource in = new XMLInputSource("MyDescriptor.xml");
    ResourceSpecifier specifier =
        UIMAFramework.getXMLParser().parseResourceSpecifier(in);

    //create Analysis Engine here
    mAnalysisEngine = UIMAFramework.produceAnalysisEngine(specifier);
    mTCAS = mAnalysisEngine.newTCAS();
  }

  // Assume some other part of your multi-threaded application could
  // call "analyzeDocument" on different threads, asynchronusly

  public synchronized void analyzeDocument(String aDoc)
  {
    //analyze a document
    mTCAS.setDocumentText(aDoc);
    mTAE.process();
    doSomethingWithResults(mTCAS);
    mTCAS.reset();
  }
  ...
}
```

Without the synchronized keyword, this application would not be thread-safe. If multiple threads called the analyzeDocument method simultaneously, they would both use the same CAS and clobber each others' results. The synchronized keyword

ensures that no more than one thread is executing this method at any given time. For more information on thread synchronization in Java, see http://java.sun.com/docs/books/tutorial/essential/threads/multithreaded.html.

The synchronized keyword ensures thread-safety, but does not allow you to process more than one document at a time. If you need to process multiple documents simultaneously (for example, to make use of a multiprocessor machine), you'll need to use more than one CAS instance.

Because CAS instances use memory and can take some time to construct, you don't want to create a new CAS instance for each request. Instead, you should use a feature of the UIMA SDK called the *TCAS Pool*. The TCAS pool class is com.ibm.uima.util.TCasPool. (Note: there are also CasPool and JCasPool classes. Use the CasPool when you have multiple Sofas; the JCasPool is a convenience for the common case of a single Sofas with JCas.)

A TCAS Pool contains some number of TCAS instances (you specify how many when you create the pool). When a thread wants to use a TCAS, it *checks out* an instance from the pool. When the thread is done using the TCAS, it must *release* the TCAS instance back into the pool. If all instances are checked out, additional threads will block and wait for an instance to become available. Here is some example code:

```
public class MyApplication
{
  private TCasPool mTCasPool;
  public MyApplication()
  {
    //get Resource Specifier from XML file or PEAR
    XMLInputSource in = new XMLInputSource("MyDescriptor.xml");
    ResourceSpecifier specifier =
      UIMAFramework.getXMLParser().parseResourceSpecifier(in);

    //create multithreadable TAE that will
    //accept 3 simultaneous requests
    mTAE = UIMAFramework.produceTAE(specifier,3);

    //create CAS pool with 3 CAS instances
    mTCasPool = new TCasPool(mTAE,3);
  }

  public void analyzeDocument(String aDoc)
  {
    //check out a CAS instance (argument 0 means no timeout)
    TCAS tcas = mTCasPool.getTCas(0);
    try
    {
      //analyze a document
      tcas.setDocumentText(aDoc);
      mTAE.process(tcas);
```

```
        doSomethingWithResults(tcas);
      }
      finally
      {
        //MAKE SURE we release the CAS instance
        mTCasPool.releaseTCas(tcas);
      }
    }
    ...
  }
```

There is not much more code required here than in the previous example. First, there is one additional parameter to the TextAnalysisEngine producer, specifying the number of annotator instances to create[3]. Then, instead of creating a single TCAS in the constructor, we now create a TCasPool containing 3 instances. In the analyze method, we check out a CAS, use it, and then release it.

> **Note:** Frequently, the two numbers (number of CASes, and the number of TAEs) will be the same. It would not make sense to have the number of CASes less than the number of TAEs – the extra TAE instances would always block waiting for a CAS from the pool. It could make sense to have additional CASes, though – if you had other multi-threaded processes that were using the CASes, other than the TAEs.

Note the use of the try...finally block. This is very important, as it ensures that the TCAS we have checked out will be released back into the pool, even if the analysis code throws an exception. You should always use try...finally when using the TCAS pool; if you do not, you risk exhausting the pool and causing deadlock.

The parameter 0 passed to the TCasPool.getTCas() method is a timeout value. If this is set to a positive integer, it is the maximum number of milliseconds that the thread will wait for an instance to become available in the pool. If this time elapses, the getTCas method will return null, and the application can do something intelligent, like ask the user to try again later. A value of 0 will cause the thread to wait forever.

6.2.7 Using Multiple Analysis Engines (and creating shared CASes)

In most cases, the easiest way to use multiple Analysis Engines from within an application is to combine them into an aggregate AE. For instructions, see section 4.3 ***Building Aggregate Analysis Engines***. Be sure that you understand this method before deciding to use the more advanced feature described in this section.

[3] Both the UIMA Collection Processing Manager framework and the remote deployment services framekwork have implementations which use CAS pools in this manner, and thereby relieve the annotator developer of the necessity to make their annotators thread-safe.

If you decide that your application does need to instantiate multiple AEs and have those AEs share a single CAS, then you will no longer be able to use the `AnalysisEngine.newCAS()`method (or the newTCAS() or newJCas() variants) to create your CAS. This is because these methods create a CAS with a data model specific to a single AE and which therefore cannot be shared by other AEs. Instead, you create a CAS as follows:

Suppose you have two analysis engines, and one CAS Consumer, and you want to create one type system from the merge of all of their type specifications. Then you can do the following:

```
AnalysisEngineDescription aeDesc1 =
  UIMAFramework.getXMLParser().parseAnalysisEngineDescription(...);
AnalysisEngineDescription aeDesc2 =
  UIMAFramework.getXMLParser().parseAnalysisEngineDescription(...);
CasConsumerDescription ccDesc =
  UIMAFramework.getXMLParser().parseCasConsumerDescription(...);
List list = new ArrayList();
list.add(aeDesc1);
list.add(aeDesc2);
list.add(ccDesc);
CAS cas = CasCreationUtils.createCas(list);    // (OR)
TCAS tcas = CasCreationUtils.createTCas(list); // (OR)
JCas jcas = CasCreationUtils.createTCas(list).getJCas();
```

The CasCreationUtils class takes care of the work of merging the AEs' type systems and producing a CAS for the combined type system. If the type systems are not compatible, an exception will be thrown.

6.2.8 Saving CASes to file systems

The UIMA framework provides APIs to save and restore the contents of a CAS to streams. The CASes are stored in an XML format (called XCAS). To save an XCAS representation of a CAS, use the method `XCASSerializer.serialize`; see the JavaDocs for details. XCASes can be read back in, using the `XCASDeserializer.deserialize` method. These methods deserialize into a pre-existing CAS, which you must create ahead of time, pre set up with the proper type system. See the JavaDocs for details.

6.3 Using Collection Processing Engines

A *Collection Processing Engine (CPE)* processes collections of artifacts (documents) through the combination of the following components: a Collection Reader, an optional CAS Initializer, Analysis Engines, and CAS Consumers. Collection Processing Engines and their components are described in *Chapter 5* ***Collection Processing Engine Developer's Guide***.

Like Analysis Engines, CPEs consist of a set of Java classes and a set of descriptors. You need to make sure the Java classes are in your classpath, but otherwise you only deal with descriptors.

6.3.1 Running a CPE from a Descriptor

Section 5.3 ***Running a CPE from Your Own Java Application*** on page 5-109 describes how to use the APIs to read a CPE descriptor and run it from an application.

6.3.2 Configuring a CPE Descriptor Programmatically

For the finest level of control over the CPE descriptor settings, the CPE offers programmatic access to the descriptor via an API. With this API, a developer can create a complete descriptor and then save the result to a file. This also can be used to read in a descriptor (using XMLParser.parseCpeDescription as shown in the previous section), modify it, and write it back out again. The CPE Descriptor API allows a developer to redefine default behavior related to error handling for each component, turn-on check-pointing, change performance characteristics of the CPE, and plug-in a custom timer.

Below is some example code that illustrates how this works. See the JavaDocs for package com.ibm.uima.collection.metadata for more details.

```
//Creates descriptor with default settings
CpeDescription cpe = CpeDescriptorFactory.produceDescriptor();

//Add CollectionReader
cpe.addCollectionReader([descriptor]);

//Add CasInitializer
cpe.addCasInitializer(<cas initializer descriptor>);


// Provide the number of CASes the CPE will use

cpe.setCasPoolsSize(2);

//  Define and add Analysis Engine
CpeIntegratedCasProcessor personTitleProcessor =
   CpeDescriptorFactory.produceCasProcessor ("Person");

// Provide descriptor for the Analysis Engine
personTitleProcessor.setDescriptor([descriptor]);

//Continue, despite errors and skip bad Cas
personTitleProcessor.setActionOnMaxError("terminate");

//Increase amount of time in ms the CPE waits for response
//from this Analysis Engine
personTitleProcessor.setTimeout(100000);

//Add Analysis Engine to the descriptor
cpe.addCasProcessor(personTitleProcessor);
```

```
//  Define and add CAS Consumer
CpeIntegratedCasProcessor consumerProcessor =
CpeDescriptorFactory.produceCasProcessor("Printer");
consumerProcessor.setDescriptor([descriptor]);

//Define batch size
consumerProcessor.setBatchSize(100);

//Terminate CPE on max errors
personTitleProcessor.setActionOnMaxError("terminate");

//Add CAS Consumer to the descriptor
cpe.addCasProcessor(consumerProcessor);

//  Add Checkpoint file and define checkpoint frequency (ms)
cpe.setCheckpoint("[path]/checkpoint.dat", 3000);

//  Plug in custom timer class used for timing events
cpe.setTimer("com.ibm.uima.reference_impl.util.JavaTimer");

//  Define number of documents to process
cpe.setNumToProcess(1000);

//  Dump the descriptor to the System.out
((CpeDescriptionImpl)cpe).toXML(System.out);
```

The CPE descriptor for the above configuration looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<cpeDescription>
  <collectionReader>
    <collectionIterator>
      <descriptor>
        <include href="[descriptor]"/>
      </descriptor>
      <configurationParameterSettings>...</configurationParameterSettings>
    </collectionIterator>

    <casInitializer>
      <descriptor>
        <include href="[descriptor]"/>
      </descriptor>
      <configurationParameterSettings>...</configurationParameterSettings>
    </casInitializer>
  </collectionReader>

  <casProcessors casPoolSize="3"
      processingUnitThreadCount="1">

    <casProcessor deployment="integrated" name="Person">
      <descriptor>
        <include href="[descriptor]"/>
      </descriptor>
      <deploymentParameters/>
       <filter/>
      <errorHandling>
        <errorRateThreshold action="terminate"
```

```
          value="100/1000"/>
        <maxConsecutiveRestarts action="terminate"
          value="30"/>
        <timeout max="100000"/>
      </errorHandling>
      <checkpoint batch="1" time="10000"/>
    </casProcessor>

    <casProcessor deployment="integrated" name="Printer">
      <descriptor>
        <include href="[descriptor]"/>
      </descriptor>
      <deploymentParameters/>
      <filter/>
      <errorHandling>
        <errorRateThreshold action="terminate"
          value="100/1000"/>
        <maxConsecutiveRestarts action="terminate"
          value="30"/>
        <timeout max="100000"/>
      </errorHandling>
      <checkpoint batch="100"/>
    </casProcessor>
  </casProcessors>

  <cpeConfig>
    <numToProcess>1000</numToProcess>
    <deployAs>immediate</deployAs>
    <checkpoint file="[path]/checkpoint.dat" time="3000"/>
    <timerImpl>
      com.ibm.uima.reference_impl.util.JavaTimer</timerImpl>
  </cpeConfig>
</cpeDescription>
```

6.4 Setting Configuration Parameters

Configuration parameters can be set using APIs as well as configured using the XML descriptor metadata specification (see ***Configuration Parameters*** on page 4-65).

There are two different places you can set the parameters via the APIs.

- After reading the XML descriptor for a component, but before you produce the component itself, and
- After the component has been produced.

Setting the parameters before you produce the component is done using the ConfigurationParameterSettings object. You get an instance of this for a particular component by accessing that component description's metadata. For instance, if you produced a component description by using UIMAFramework.getXMLParser().parse... method, you can use that component description's getMetaData() method to get the metadata, and then the metadata's

getConfigurationParameterSettings method to get the ConfigurationParameterSettings object. Using that object, you can set individual parameters using the setParameterValue method. Here's an example, for a CAS Consumer component:

```
// Create a description object by reading the XML for the descriptor

CasConsumerDescription casConsumerDesc =
UIMAFramework.getXMLParser().parseCasConsumerDescription( new
XMLInputSource("descriptors/cas_consumer/InlineXmlCasConsumer.xml"));

// get the settings from the metadata
ConfigurationParameterSettings consumerParamSettings
=casConsumerDesc.getMetaData().getConfigurationParameterSettings();

// Set a parameter value
consumerParamSettings.setParameterValue(  InlineXmlCasConsumer.PARAM_OUTPUTDIR,
outputDir.getAbsolutePath());
```

Then you might produce this component using:

```
CasConsumer component = UIMAFramework.produceCasConsumer(casConsumerDesc);
```

A side effect of producing a component is calling the component's "initialize" method, allowing it to read its configuration parameters. If you want to change parameters after this, use

```
component.setConfigParameterValue("<parameter-name>", "<parameter-value>");
```

and then signal the component to re-read its configuration by calling the component's reconfigure method:

```
component.reconfigure();
```

Although these examples are for a CAS Consumer component, the parameter APIs also work for other kinds of components.

6.5 Integrating Text Analysis and Search

The UIMA SDK includes a search engine that you can use to build a search index that includes the results of the analysis done by your TAE. This combination of TAEs with a search engine capable of indexing both words and annotations over spans of text enables what UIMA refers to as *semantic search*.

Semantic search is a search where the semantic intent of the query is specified using one or more entity or relation specifiers. For example, one could specify that they

are looking for a person (named) "Bush." Such a query would then not return results about the kind of bushes that grow in your garden.

6.5.1 Indexing

To build a semantic search index using the UIMA SDK, you run a Collection Processing Engine that includes your TAE along with a CAS Consumer called the *Semantic Search CAS Indexer*, which is provided with the UIMA SDK. Your TAE must include an annotator that produces Tokens and Sentence annotations, along with any "semantic" annotations, because the Indexer requires this. The Semantic Search CAS Indexer's descriptor is located at: `docs/examples/descriptors/cas_consumer/SemanticSearchCasIndexer.xml`.

Configuring the Semantic Search CAS Indexer

Since there are several ways you might want to build a search index from the information in the CAS produced by your TAE, you need to supply the Semantic Search CAS Indexer with configuration information in the form of an *Index Build Specification* file. An example of an Indexing specification tailored to the TAE from the tutorial in the *Chapter 4* ***Annotator and Analysis Engine Developer's Guide*** is located in `docs/examples/descriptors/tutorial/search/MeetingIndexBuildSpec.xml`. It looks like this:

```
<indexBuildSpecification>
  <indexBuildItem>
    <name>com.ibm.uima.tutorial.WordAnnot</name>
    <indexRule>
      <style name="Term"/>
    </indexRule>
  </indexBuildItem>

  <indexBuildItem>
    <name>com.ibm.uima.tutorial.SentenceAnnot</name>
    <indexRule>
      <style name="Breaking"/>
    </indexRule>
  </indexBuildItem>

  <indexBuildItem>
    <name>com.ibm.uima.tutorial.Meeting</name>
    <indexRule>
      <style name="Annotation"/>
    </indexRule>
  </indexBuildItem>

  <indexBuildItem>
    <name>com.ibm.uima.tutorial.RoomNumber</name>
    <indexRule>
      <style name="Annotation"/>
    </indexRule>
  </indexBuildItem>
```

```
  <indexBuildItem>
    <name>com.ibm.uima.tutorial.DateAnnot</name>
    <indexRule>
      <style name="Annotation"/>
    </indexRule>
  </indexBuildItem>

  <indexBuildItem>
    <name>com.ibm.uima.tutorial.TimeAnnot</name>
    <indexRule>
      <style name="Annotation"/>
    </indexRule>
  </indexBuildItem>
</indexBuildSpecification>
```

The index build specification is a series of index build items, each of which identifies a CAS annotation type (a subtype of `uima.tcas.Annotation` – see *Chapter 23 **CAS Reference***) and a style.

The first item in this example specifies that the annotation type `com.ibm.uima.tutorial.WordAnnot` should be indexed with the "Term" style. This means that each span of text annotated by a WordAnnot will be considered a single token for standard text search purposes.

The second item in this example specifies that the annotation type `com.ibm.uima.tutorial.SentenceAnnot` should be indexed with the "Breaking" style. This means that each span of text annotated by a SentenceAnnot will be considered a single sentence, which can affect that search engine's algorithm for matching queries. The semantic search engine always requires tokens and sentences in order to index a document.

> **Note:** Requirements for Term and Breaking rules: The Semantic Search indexer supplied with the UIMA SDK requires that the items to be indexed as words be designated using the Term rule. Furthermore, due to a limitation of this indexer, the number of terms allowed in-between break boundaries must be less than 256.

The remaining items all use the "Annotation" style. This indicates that each annotation of the specified types will be stored in the index as a searchable span, with a name equal to the annotation name (without the namespace).

At the end of the batch or collection, the Semantic Search CAS Indexer builds the index. This index can be queried with simple tokens or with xml tags

Example : A query on the word "UIMA" will retrieve all documents that have the occurrence of the word. But a query of the type `<Meeting>UIMA</Meeting>` will retrieve only those documents that contain a Meeting annotation (produced by our MeetingDetector TAE, for example), where that Meeting annotation contains the

word "UIMA". More information on the syntax of these kinds of queries, called XML Fragments, can be found in Chapter 25 ***Semantic Search Engine Reference*** on page 25-345.

For more information on the Index Build Specification format, see the UIMA JavaDocs for class `com.ibm.uima.search.IndexBuildSpecification`. Accessing the JavaDocs is described on page 22-315.

Building and Running a CPE including the Semantic Search CAS Indexer

The following steps illustrate how to build and run a CPE that uses the UIMA Meeting Detector TAE and the Simple Token and Sentence Annotator, discussed in the *Chapter 4* ***Annotator and Analysis Engine Developer's Guide*** along with the Semantic Search CAS Indexer, to build an index that allows you to query for documents based not only on textual content but also on whether they contain mentions of Meetings detected by the TAE.

Run the CPE Configurator tool by executing the `cpeGuit` shell script in the `bin` directory of the UIMA SDK. (For instructions on using this tool, see the *Chapter 10* ***Collection Processing Engine Configurator User's Guide***.)

In the CPE Configurator tool, select the following components by browsing to their descriptors:

- Collection Reader: `%UIMA_HOME%/docs/examples/descriptors/collectionReader/FileSystemCollectionReader.xml`
- Analysis Engine: include both of these; one produces tokens/sentences, required by the indexer in all cases and the other produces the meeting annotations of interest.
  `%UIMA_HOME%/docs/examples/descriptors/tutorial/ex6/UIMAMeetingDetector.xml` and
  `%UIMA_HOME%/docs/examples/descriptors/analysis_engine/SimpleTokenAndSentenceAnnotator.xml`
- Two CAS Consumers:

  `%UIMA_HOME%/docs/examples/descriptors/casConsumer/SemanticSearchCasIndexer.xml`

  `%UIMA_HOME%/docs/examples/descriptors/casConsumer/XCasWriterCasConsumer.xml`

Set up parameters :

- Set the File System Collection Reader's "Input Directory" parameter to point to the `%UIMA_HOME%/docs/examples/data` directory.
- Set the Semantic Search CAS Indexer's "Indexing Specification Descriptor" parameter to point to `%UIMA_HOME%/docs/examples/descriptors/tutorial/search/MeetingIndexBuildSpec.xml`

- Set the Semantic Search CAS Indexer's "Index Dir" parameter to whatever directory into which you want the indexer to write its index files. ***WARNING***: The Indexer ***erases old versions of the files it creates in this directory.***
- Set the XCAS Writer CAS Consumer's "Output Directory" parameter to whatever directory into which you want to store the XCAS files containing the results of your analysis for each document.

Click on the Run Button. Once the run completes, a statistics dialog should appear, in which you can see how much time was spent in each of the components involved in the run.

6.5.2 Semantic Search Query Tool

The UIMA SDK contains a simple tool for running queries against a semantic search index. After building an index as described in the previous section, you can launch this tool by running the shell script: semanticSearch, found in the `/bin` subdirectory of the UIMA install, at the command prompt. If you are using Eclipse, and have installed the UIMA examples, there will be a Run configuration you can use to conveniently launch this, called `UIMA Semantic Search`. This will display the following screen:

Configure the first three fields on this screen as follows:

- Set the "Index Directory" to the directory where you built your index. This is the same value that you supplied for the "Index Dir" parameter of the Semantic Search CAS Indexer in the CPE Configurator.

- Set the "XCAS Directory" to the directory where you stored the XCAS files containing the results of your analysis. This is the same value that you supplied for the "Output Directory" parameter of XCAS Writer CAS Consumer in the CPE Configurator.
- Set the "Type System Descriptor" to the location of the descriptor that describes your type system. For this example, this will be `%UIMA_HOME%/docs/examples/descriptors/tutorial/ex4/TutorialTypeSystem.xml`

Now, in the "XML Fragments" field, you can type in single words or xml queries where the xml tags correspond to the labels in the index build specification file (e.g. `<Meeting>UIMA</Meeting>`). XML Fragments are described in Chapter 25 25-345.

After you enter a query and click the "Search" button, a list of hits will appear. Select one of the documents and click "View Analysis" to view the document in the UIMA Annotation Viewer.

The source code for the Semantic Search query program is in `docs/examples/src/com/ibm/uima/examples/search/SemanticSearchGUI.java`. A simple command-line query program is also provided in `docs/examples/src/com/ibm/uima/examples/search/SemanticSearch.java`. Using these as a model, you can build a query interface from your own application. For details on the Semantic Search Engine query language and interface, see *Chapter 25 **Semantic Search Engine Reference***.

6.6 Working with Analysis Engine and CAS Consumer Services

The UIMA SDK allows you to easily take any Analysis Engine or CAS Consumer and deploy it as a service. That Analysis Engine or CAS Consumer can then be called from a remote machine.

The UIMA SDK provides support for two communications protocols

- SOAP, the standard Web Services protocol
- Vinci, an IBM-developed, lightweight version of SOAP

The UIMA framework can make use of these services in two different ways:

1. An Analysis Engine can create a proxy to a remote service; this proxy acts like a local component, but connects to the remote. The proxy has limited error handling and retry capabilities. Both Vinci and SOAP are supported.
- A Collection Processing Engine can specify non-Integrated mode (see ***Deploying a CPE*** on page 5-122). The CPE provides more extensive error recovery capabilities. This mode only supports the Vinci communications protocol

6.6.1 How to Deploy a UIMA Component as a SOAP Web Service

To deploy a UIMA component as a SOAP Web Service, you need to first install the following software components:

- Apache Tomcat 5.0 or 5.5 (http://jakarta.apache.org/tomcat/)
- Apache Axis 1.1 or 1.3 (http://ws.apache.org/axis/)

Later versions of these components will likely also work, but have not been tested.

Next, you need to do the following three setup steps:

- Set the CATALINA_HOME environment variable set to the location where Tomcat is installed.
- Copy all of the JAR files from `%UIMA_HOME%/lib` to the `%CATALINA_HOME%/webapps/axis/WEB-INF/lib` in your installation.
- Copy your JAR files for the UIMA components that you wish to `%CATALINA_HOME%/webapps/axis/WEB-INF/lib` in your installation.

  Note: ***IMPORTANT***: any time you add JAR files to TomCat (for instance, in the above 2 steps), you must shutdown and restart TomCat before it "notices" this. So now, please shutdown and restart TomCat.

  Note: All the Java classes for the UIMA Examples are packaged in the `uima_examples.jar` file which is included in the `%UIMA_HOME%/lib` folder.

- In addition, if an annotator needs to locate resource files in the classpath, those resources must be available in the Axis classpath, so copy these also to `%CATALINA_HOME%/webapps/axis/WEB-INF/classes`.

  As an example, if you are deploying the GovernmentTitleRecognizer (found in `docs/examples/descriptors/analysis_engine/GovernmentOfficialRecognizer_RegEx_TAE`) as a SOAP service, you need to copy the file `docs/examples/resources/GovernmentTitlePatterns.dat` into `.../WEB-INF/classes`.

Test your installation of Tomcat and Axis by starting Tomcat and going to `http://localhost:8080/axis/happyaxis.jsp` in your browser. Check to be sure that this reports that all of the required Axis libraries are present. One common missing file may be activation.jar, which you can get from java.sun.com.

After completing these setup instructions, you can deploy Analysis Engines or CAS Consumers as SOAP web services by using the `deploytool` utility, with is located in the `/bin` directory of the UIMA SDK. `deploytool` is a command line program utility that takes as an argument a web services deployment descriptors (WSDD file); example WSDD files are provided in the `docs\examples\deploy\soap` directory of the

UIMA SDK. Deployment Descriptors have been provided for deploying and undeploying some of the example Analysis Engines that come with the SDK.

As an example, the WSDD file for deploying the example Person Title annotator looks like this (important parts are in bold italics):

```
<deployment name="PersonTitleAnnotator"
            xmlns="http://xml.apache.org/axis/wsdd/"
            xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <service name="urn:PersonTitleAnnotator" provider="java:RPC">

    <parameter name="scope" value="Request"/>

    <parameter name="className"
value="com.ibm.uima.reference_impl.analysis_engine.service.soap.AxisAnalysi
sEngineService_impl"/>

    <parameter name="allowedMethods" value="getMetaData process"/>
    <parameter name="allowedRoles" value="*"/>
    <parameter name="resourceSpecifierPath" value="c:/Program Files/IBM/
uima/docs/examples/descriptors/analysis_engine/PersonTitleAnnotator.xml"/>

    <parameter name="numInstances" value="3"/>
    <parameter name="timeoutPeriod" value="30000"/>

    <!-- Type Mappings omitted from this document; you will not need to
edit them. -->

    <typeMapping .../>
    <typeMapping .../>
    <typeMapping .../>

  </service>

</deployment>
```

To modify this WSDD file to deploy your own Analysis Engine or CAS Consumer, just replace the areas indicated in bold italics (deployment name, service name, and resource specifier path) with values appropriate for your component.

The `timeoutPeriod` parameter only is used when there are multiple clients accessing the service. When a new request comes in, if the service is busy with other requests (all instances are busy, in the case where it has multiple instances), it waits for one to become available - and this parameter specifies the maximum time for that wait. If it takes longer than this, the service wrapper will throw an exception back to the client and abort the processing for this document on the service.

To deploy the Person Title annotator service, issue the following command:

```
C:\Program Files\IBM\uima>bin\deploytool
docs\examples\deploy\soap\Deploy_PersonTitleAnnotator.wsdd
```

Test if the deployment was successful by starting up a browser, pointing it to your TomCat installation's "axis" webpage (e.g., http://localhost:8080/axis) and clicking on

the List link. This should bring up a page which shows the deployed services, where you should see the service you just deployed.

The other components can be deployed by replacing `Deploy_PersonTitleAnnotator.wsdd` with one of the other Deploy descriptors in the deploy directory. The deploytool utility can also undeploy services when passed one of the Undeploy descriptors.

Note: The `deploytool` shell script assumes that the web services are to be installed at `http://localhost:8080/axis`. If this is not the case, you will need to update the shell script appropriately.

Once you have deployed your component as a web service, you may call it from a remote machine. See "How to Call a UIMA Service," below, for instructions.

6.6.2 How to Deploy a UIMA Component as a Vinci Service

There are no software prerequisites for deploying a Vinci service. The necessary libraries are part of the UIMA SDK. However, before you can use Vinci services you need to deploy the Vinci Naming Service (VNS), as described in section 6.6.5 .

To deploy a service, you have to insure any components you want to include can be found on the class path. One way to do this is to set the environment variable UIMA_CLASSPATH to the set of class paths you need for any included components. Then run the `startVinciService` shell script, which is located in the UIMA SDK bin directory, and pass it the path to a Vinci service deployment descriptor, for example:

```
C:\UIMA>bin\startVinciService
docs\examples\deploy\vinci\Deploy_PersonTitleAnnotator.xml
```

This example deployment descriptor looks like:

```
<deployment name="Vinci Person Title Annotator Service">
  <service name="uima.annotator.PersonTitleAnnotator" provider="vinci">
    <parameter name="serializerClassName"
value="com.ibm.uima.reference_impl.analysis_engine.service.vinci.VinciXCASS
erializer_NoDocText"/>
    <parameter name="resourceSpecifierPath" value="c:/Program
Files/IBM/uima/docs/examples/descriptors/analysis_engine/PersonTitleAnnotat
or.xml"/>
    <parameter name="numInstances" value="1"/>
    <parameter name="timeoutPeriod" value="30000"/>
    <parameter name="serverSocketTimeout" value="120000"/>
  </service>
```

```
</deployment>
```

To modify this deployment descriptor to deploy your own Analysis Engine or CAS Consumer, just replace the areas indicated in bold italics (deployment name, service name, and resource specifier path) with values appropriate for your component.

The `timeoutPeriod` parameter only is used when there are multiple clients accessing the service. When a new request comes in, if the service is busy with other requests (all instances are busy, in the case where it has multiple instances), it waits for one to become available - and this parameter specifies the maximum time for that wait. If it takes longer than this, the service wrapper will throw an exception back to the client and abort the processing for this document on the service.

The `serverSocketTimout` parameter specifies the number of milliseconds (default = 5 minutes) that the service will wait between requests to process something. After this amount of time, the server will presume the client may have gone away - and it "cleans up", releasing any resources it is holding. The next call to process on the service will result in a cycle which will cause the client to re-establish its connection with the service (some additional overhead).

The `startVinciService` script takes two additional optional parameters. The first one overrides the value of the VNS_HOST environment variable, allowing you to specify the name server to use. The second parameter if specified needs to be a unique (on this server) non-negative number, specifying the instance of this service. When used, this number allows multiple instances of the same named service to be started on one server; they will all register with the Vinci name service and be made available to client requests.

Once you have deployed your component as a web service, you may call it from a remote machine. See "How to Call a UIMA Service," below, for instructions.

6.6.3 How to Call a UIMA Service

Once an Analysis Engine or CAS Consumer has been deployed as a service, it can be used from any UIMA application, in the exact same way that a local Analysis Engine or CAS Consumer is used. For example, you can call an Analysis Engine service from the Document Analyzer or use the CPE Configurator to build a CPE that includes Analysis Engine and CAS Consumer services.

To do this, you use a *service client descriptor* in place of the usual Analysis Engine or CAS Consumer Descriptor. A service client descriptor is a simple XML file that indicates the location of the remote service and a few parameters. Example service client descriptors are provided in the UIMA SDK under the directories `examples/docs/descriptors/soapService` and

examples/docs/descriptors/vinciService. The contents of these descriptors are explained below.

Also, before you can call a SOAP service, you need to have the necessary Axis JAR files in your classpath. If you use any of the scripts in the /bin directory of the UIMA installation to launch your application, such as documentAnalyzer, these JARs are added to the classpath, automatically, using the CATALINA_HOME environment variable. The required files are the following (all part of the Apache Axis download):

- activation.jar
- axis.jar
- commons-discovery.jar
- commons-logging.jar
- jaxrpc.jar
- saaj.jar.

SOAP Service Client Descriptor

The descriptor used to call the PersonTitleAnnotator SOAP service from the example above is:

```
<uriSpecifier xmlns="http://uima.watson.ibm.com/resourceSpecifier">
   <resourceType>AnalysisEngine</resourceType>
   <uri>http://localhost:8080/axis/services/urn:PersonTitleAnnotator</uri>
    <protocol>SOAP</protocol>
</uriSpecifier>
```

The <resourceType> element must contain either AnalysisEngine or CasConsumer. This specifies what type of component you expect to be at the specified service address.

The <uri> element describes which service to call. It specifies the host (localhost, in this example) and the service name (urn:PersonTitleAnnotator), which must match the name specified in the deployment descriptor used to deploy the service.

Vinci Service Client Descriptor

To call a Vinci service, a similar descriptor is used:

```
<uriSpecifier xmlns="http://uima.watson.ibm.com/resourceSpecifier">
   <resourceType>AnalysisEngine</resourceType>
   <uri>uima.annot.PersonTitleAnnotator</uri>
   <protocol>Vinci</protocol>
   <parameters>
     <parameter name="VNS_HOST" value="some.internet.ip.name-or-address"/>
     <parameter name="VNS_PORT" value="9000"/>
```

```
    </parameters>
  </uriSpecifier>
```

Note that Vinci uses a centralized naming server, so the host where the service is deployed does not need to be specified. Only a name (uima.annot.PersonTitleAnnotator) is given, which must match the name specified in the deployment descriptor used to deploy the service.

The host and/or port where your Vinci Naming Service (VNS) server is running can be specified by the optional <parameter> elements. If not specified, the value is taken from the specification given your Java command line (if present) using `-DVNS_HOST=<host>` and `-DVNS_PORT=<port>` system arguments. If not specified on the Java command line, defaults are used: localhost for the `VNS_HOST`, and `9000` for the `VNS_PORT`. See the next section for details on setting up a VNS server.

6.6.4 Restrictions on remotely deployed services

Remotely deployed services are started on remote machines, using UIMA component descriptors on those remote machines. These descriptors supply any configuration and resource parameters for the service (configuration parameters are not transmitted from the calling instance to the remote one). Likewise, the remote descriptors supply the type system specification for the remote annotators that will be run (the type system of the calling instance is not transmitted to the remote one).

The remote service wrapper, when it receives a CAS from the caller, instantiates it for the remote service, making instances of all types which the remote service specifies. Other instances in the incoming CAS for types which the remote service has no type specification for are kept aside, and when the remote service returns the CAS back to the caller, these type instances are re-merged back into the CAS being transmitted back to the caller. Because of this design, a remote service which doesn't declare a type system won't receive any type instances.

> **Note:** This behavior may change in future releases, to one where configuration parameters and / or type systems are transmitted to remote services.

6.6.5 The Vinci Naming Service (VNS)

Vinci consists of components for building network-accessible services, clients for accessing those services, and an infrastructure for locating and managing services. The primary infrastructure component is the Vinci directory, known as VNS (for Vinci Naming Service).

On startup, Vinci services locate the VNS and provide it with information that is used by VNS during service discovery. Vinci service provides the name of the host machine on which it runs, and the name of the service. The VNS internally creates a binding for the service name and returns the port number on which the Vinci

service will wait for client requests. This VNS stores its bindings in a filesystem in a file called vns.services.

In Vinci, services are identified by their service name. If there is more than one physical service with the same service name, then Vinci assumes they are equivalent and will route queries to them randomly, provided that they are all running on different hosts. You should therefore use a unique service name if you don't want to conflict with other services listed in whatever VNS you have configured jVinci to use.

Starting VNS

To run the VNS use the `startVNS` script found in the /bin directory of the UIMA installation.

> **Note:** VNS runs on port 9000 by default so please make sure this port is available. If you see the following exception:
>
> ```
> java.net.BindException: Address already in use: JVM_Bind
> ```
>
> it indicates that another process is running on port 9000. In this case, add the parameter `-p <port>` to the `startVNS` command, using `<port>` to specify an alternative port to use.

When started, the VNS produces output similar to the following:

```
[10/6/04 3:44 PM | main] WARNING: Config file doesn't exist, creating a new
empty config file!
[10/6/04 3:44 PM | main] Loading config file : .\vns.services
[10/6/04 3:44 PM | main] Loading workspaces file : .\vns.workspaces
[10/6/04 3:44 PM | main] ==================================
(WARNING) Unexpected exception:
java.io.FileNotFoundException: .\vns.workspaces (The system cannot find
the file specified)
        at java.io.FileInputStream.open(Native Method)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.io.FileInputStream.<init>(Unknown Source)
        at java.io.FileReader.<init>(Unknown Source)
        at
com.ibm.vinci.transport.vns.service.VNS.loadWorkspaces(VNS.java:339)
at com.ibm.vinci.transport.vns.service.VNS.startServing(VNS.java:237)
        at com.ibm.vinci.transport.vns.service.VNS.main(VNS.java:179)
[10/6/04 3:44 PM | main] WARNING: failed to load workspace.
[10/6/04 3:44 PM | main] VNS Workspace : null
[10/6/04 3:44 PM | main] Loading counter file : .\vns.counter
[10/6/04 3:44 PM | main] Could not load the counter file : .\vns.counter
[10/6/04 3:44 PM | main] Starting backup thread, using files
.\vns.services.bak
and .\vns.services
[10/6/04 3:44 PM | main] Serving on port : 9000
[10/6/04 3:44 PM | Thread-0] Backup thread started
```

```
[10/6/04 3:44 PM | Thread-0] Saving to config file : .\vns.services.bak
>>>>>>>>>>>>> VNS is up and running! <<<<<<<<<<<<<<<<<
>>>>>>>>>>>>> Type 'quit' and hit ENTER to terminate VNS <<<<<<<<<<<<<
[10/6/04 3:44 PM | Thread-0] Config save required 10 millis.
[10/6/04 3:44 PM | Thread-0] Saving to config file : .\vns.services
[10/6/04 3:44 PM | Thread-0] Config save required 10 millis.
[10/6/04 3:44 PM | Thread-0] Saving counter file : .\vns.counter
```

Note: Disregard the *java.io.FileNotFoundException: .\vns.workspaces (The system cannot find the file specified).* It is just a complaint not a serious problem. VNS Workspace is a feature of the VNS that is not critical. The important information to note is `[10/6/04 3:44 PM | main] Serving on port : 9000`

which states the actual port where VNS will listen for incoming requests. All Vinci services and all clients connecting to services must provide the VNS port on the command line IF the port is not a default. Again the default port is 9000. Please see section ***Launching Vinci Services*** below for details about the command line and parameters.

VNS Files

The VNS maintains two external files

```
vns.services
vns.counter
```

These files are generated by the VNS in the same directory where the VNS is launched from. Since these files may contain old information it is best to remove them before starting the VNS. This step ensures that the VNS has always the newest information and will not attempt to connect to a service that has been shutdown.

Launching Vinci Services

When launching Vinci service, you must indicate which VNS the service will connect to. To define Vinci's default VNS you must provide the following JVM parameters:

```
java -DVNS_HOST=localhost -DVNS_PORT=9000 ...
```

The above setting is for the VNS running on the same machine as the service. Of course one can deploy the VNS on a different machine and the JVM parameter will need to be changed to this:

```
java -DVNS_HOST=<host> -DVNS_PORT=9000 ...
```

where '<host>' is a machine name or its IP where the VNS is running.

Note: VNS runs on port 9000 by default. When you see the following exception:

```
(WARNING) Unexpected exception:
com.ibm.vinci.transport.ServiceDownException: VNS inaccessible:
java.net.Connect
Exception: Connection refused: connect
```

then, perhaps the VNS is not running OR the VNS is running but its port has been changed. To correct the latter, change the command line by providing the correct port

```
-DVINCI_PORT=<port>
```

To get the right port check the VNS output for something similar to the following

```
[10/6/04 3:44 PM | main] Serving on port : 9000
```

It is printed by the VNS on startup.

6.7 Increasing performance using parallelism

There are several ways to exploit parallelism to increase performance in the UIMA Framework. These range from running with additional threads within one Java virtual machine on one host (which might be a multi-processor or hyper-threaded host) to deploying analysis engines on a set of remote machines.

The Collection Processing facility in UIMA provides the ability to scale the pipe-line of analysis engines. This scale-out runs multiple threads within the Java virtual machine running the CPM, one for each pipe in the pipe-line. To activate it, in the `<casProcessors>` descriptor element, set the attribute `processingUnitThreadCount`, which specifies the number of replicated processing pipelines, to a value greater than 1, and insure that the size of the CAS pool is equal to or greater than this number (the attribute of `<casProcessors>` to set is `casPoolSize`). For more details on these settings, see ***CAS Processors*** on page 21-297.

For deployments that incorporate remote analysis engines in the Collection Manager pipe-line, running on multiple remote hosts, scale-out is supported which uses the Vinci naming service. If multiple instances of a service with the same name, but running on different hosts, are registered with the Vinci Name Server, it will assign these instances to incoming requests.

There are two modes supported: a "random" assignment, and a "exclusive" one. The "random" mode distributes load using an algorithm that selects a service instance at random. The UIMA framework supports this only for the case where all of the instances are running on unique hosts; the framework does not support starting 2 or more instances on the same host.

The exclusive mode dedicates a particular remote instance to each Collection Manager pip-line instance. This mode is enabled by adding a configuration parameter in the <casProcessor> section of the CPE descriptor:

```
<deploymentParameters>
  <parameter name="service-access" value="exclusive" />
</deploymentParameters>
```

If this is not specified, the "random" mode is used.

In addition, remote UIMA engine services can be started with a parameter that specifies the number of instances the service should support (see the `<parameter name="numInstances">` xml element in remote deployment descriptor on page 6-147. Specifying more than one causes the service wrapper for the analysis engine to use multi-threading (within the single Java Virtual Machine – which can take advantage of multi-processor and hyper-threaded architectures).

> **Note:** When using Vinci in "exclusive" mode (see service access under ***<deploymentParameters> Element*** on page 21-303), only one thread is used. To achieve multi-processing on a server in this case, use multiple instances of the service, instead of multiple threads (see ***How to Deploy a UIMA Component as a Vinci Service*** on page 6-150).

Chapter 7 Developing Applications using Multiple Subjects of Analysis

This chapter describes how to develop a UIMA application that makes use of multiple subjects of analysis in the same processing pipeline. Multiple Subject of Analysis (Sofa) capability can simplify text applications that need different versions of the text at different stages. Multiple Sofa capability is also a key to enabling multimodal applications where the initial artifact is transformed from one modality to another, or where the artifact itself is multimodal such as the audio, video and closed-captioned text associated with an MPEG object. Each Sofa can be analyzed independently with the standard UIMA programming model, or analyzed together with other Sofas utilizing the Sofa programming model extensions.

A basic multi-Sofa application design issue for UIMA CAS processing components is whether they should be Sofa-aware or Sofa-unaware. Sofa-aware components create new Sofas and/or will access multiple Sofas during processing. Components which analyze data from a single, previously created Sofa can be standard (Sofa-unaware) processing components. The application designer must arrange that Sofas created and output by one component are appropriately connected as input Sofas for dependent components.

A key goal of the Sofa support is to recognize that most analytic components are (by design) Sofa-unaware, and to not place any burden on these components related to Sofas. The Sofa support is designed to enable these Sofa unaware components to be used in complex, multi-Sofa application scenarios.

This chapter is organized into the following subsections:

- 7.1 Basic Sofa Concepts and Methods describes how to create Sofas, declare them in component descriptors and how to create and access Sofa data.
- 7.2 Sofas and TCAS Views explains the difference between CAS and TCAS, the special relationship between a Sofa and its TCAS, and the delivery of one or more Sofas to UIMA processing components via CAS and TCAS objects.
- 7.3 Sofa Name Mapping shows how to connect output Sofas from one component to input Sofas in other, dependent components.
- 7.4 Sofa Impact on XCAS Format describes how Sofas impact the XCAS format.
- 7.5 Sofa Sample Application describes an application using multiple text Sofas.

7.1 Basic Sofa Concepts and Methods

Unstructured information must often be transformed from its original representation in order to achieve the desired analysis. For example, web page content is often converted into a different character representation and the HTML formatted content converted into plain text before deeper analysis. In typical UIMA applications the plain text result of these conversions would be stored in a Java string called the “TCAS document" and sent through the processing pipeline. In order to support more complex processing scenarios supporting multimodal content UIMA formalizes the distinction between the original unstructured artifact and the various forms the artifact may take during analysis, enabling all to coexist in the CAS.

A Sofa is a generalization of the TCAS document and shares some of the same characteristics: a Sofa is immutable over the course of processing; analysis results for a Sofa are stored in the Sofa’s TCAS using TCAS or JCas methods; such analysis results can “belong" to a specific Sofa, i.e. an annotation points to a subset of the Sofa.

The main differences between a Sofa and the original TCAS document are that there can be multiple Sofas contained in a CAS -- and therefore many TCAS documents in the same CAS, and a Sofa can be arbitrary binary data.

Each Sofa is represented in the CAS using the built-in CAS type: uima.cas.Sofa. Features of the Sofa type include

- SofaID: Every Sofa in a CAS must have a unique SofaID. SofaIDs are the primary handle for access.
- Mime type: This string feature can be used to describe the type of the data represented by a Sofa.
- Sofa Data: The Sofa data itself. This data can be resident in the CAS or it can be a reference to data outside the CAS.

Although Sofa type instances are implemented as standard feature structures, *generic CAS APIs can not be used to create Sofas or set their features*. Specially designed methods for creating Sofa instances and accessing Sofa instance features must be used to assure proper Sofa bookkeeping operations.

The rest of this section documents Sofa access methods available to Sofa-aware components.

7.1.1 Multiple names for the same Sofa

Each independently-developed component can specify it works with one or more Sofas. Each component identifies these Sofas using a name string. An assembler assembling multiple components together can specify mappings for these names, so that differently named Sofas in different components can actually refer to the same Sofa. See ***Sofa Name Mapping*** on page 7-165 for more information about this.

Most user APIs that work with a particular Sofa need the unique SofaId as an argument. A component gets that SofaId using:

```
SofaID sofaID = getUimaContext().mapToSofaID("detagContent");
```

This maps the component's name for the Sofa to the unique SofaID for that Sofa in the application. The `getUimaContext()` method is a method on all Resource objects (including Analysis Engines). Within an annotator, however, use the `AnnotatorContext` object, which is is equlivalent to the `UimaContext` object returned by the `getUimaContext()`. The `AnnotatorContext` object is passed in as an argument to the annotator's initialize() method, and can be retrieved using the `getContext()` method on the annotator's instance object.

7.1.2 Instantiating Sofa Feature Structures

When a UIMA application starts up, it typically reads an XML descriptor and instantiates Analysis Engines or Collection Processing Engines, and creates one or more CAS instances to use in processing. For backwards compatibility, UIMA can create a default "text" Sofa. New applications will create a "base CAS" (see ***CAS versus TCAS View*** on page 7-162 for a definition of “base CAS"). Sofas are then created as they are needed by the application or by individual UIMA components in the processing pipeline. For example, to create one Sofa feature structure corresponding to the component Sofa named “detagContent", setting its mime type to “text", you use:

```
SofaID sofaID = getUimaContext().mapToSofaID("detagContent");
SofaFS aSofaFS = aCAS.createSofa(sofaID, "text");
```

Once a Sofa feature structure is created, it can be accessed from the CAS using the sofaID:

```
SofaFS aSofaFS = aCAS.getSofa(sofaID);
```

7.1.3 Setting Sofa Data

Sofa data (the thing being analyzed) can be contained locally within the CAS itself or it can be remote from the CAS. Data that is held inside the CAS is further differentiated as a Java string or as binary data. To set local Sofa data in the SofaFS created above, use this method:

```
aSofaFS.setLocalSofaData(theData);
```

where “theData" can be one of:

- a Java string, or
- a previously created CAS array type feature structure

For backwards compatibility local Sofa string data can still be set using the TCAS setDocumentText() method. See below for more on this in the section ***Sofas and TCAS Views***.

If the data is remote from the CAS, use:

```
aSofaFS.setRemoteSofaURI(theData);
```

where "theData"is a string conforming to the standard URI format.

7.1.4 Accessing Sofa Features and Sofa Data

Given a reference to a SofaFS object, aSofaFS, the mime type is retrieved using:

```
String mimeType = aSofaFS.getSofaMime();
```

Local Sofa data is accessed in one of three ways. If the data is a Java string, use:

```
String document = aSofaFS.getLocalStringData();
```

If it is an array of integers or floats, use:

```
FeatureStructure arrayFS = aSofaFS.getLocalFSData();
```

If it is a URI, you can retrieve the URI string using:

```
String uri = aSofaFS.getSofaURI();
```

Java offers built-in support for several URI schemes including “FILE:", "HTTP:", "FTP:"and has an extensible mechanism, `URLStreamHandlerFactory`, for customizing access to an arbitrary URI. See more details at http://java.sun.com/j2se/1.4.2/docs/api/java/net/URLStreamHandlerFactory.html.

Any of these three kinds of Sofa data may be accessed by getting an InputStream using:

```
InputStream is = aSofaFS.getSofaDataStream();
```

Annotators written to access Sofa data as a stream will work for all three kinds of Sofa data, including remote data accessed via the URI.

7.1.5 Declaring Sofas in Component Descriptors

Each Sofa-aware component that creates a Sofa or expects to find a previously created Sofa must declare the Sofa name in the capabilities section. For the example above:

```
<capabilities>
  <capability>
    <inputs/>
    <outputs/>
    <inputSofas/>
    <outputSofas>
      <sofaName>detagContent</sofaName>
    </outputSofas>
  </capability>
</capabilities>
```

Details on this specification are found reference chapter section on ***Capabilities*** on page 20-276. The Component Descriptor Editor supports Sofa declarations on the ***Capabilities Page*** (page 9-198).

7.2 Sofas and TCAS Views

A CAS (see Chapter 23 ***CAS Reference*** on page 23-317) is the part of UIMA concerned with creating and handling the data that annotators manipulate. The TCAS class extends a CAS with additional access methods for annotation types and a Java string known as the TCAS document. Earlier UIMA platforms made little distinction between CAS and TCAS objects and these names were used interchangeably. In fact, all applications simply created TCAS objects because the only subject of analysis ever used was the TCAS document.

In order to both support multiple subjects of analysis and maintain compatibility with all pre-existing UIMA components, the relationship between CAS and TCAS was changed.

7.2.1 CAS versus TCAS View

Instead of the TCAS being a type of CAS which is independent of other TCAS objects, *a TCAS object is now a "view" of a CAS tied to a particular Sofa in the CAS*. A CAS can have as many TCAS views as it has Sofas. The CAS is sometimes referred to as the "base CAS" for all views.

TCAS views can be created dynamically as needed.

When local Sofa data (see ***Setting Sofa Data*** on page 7-160) is set to a Java string it automatically becomes the TCAS document for the TCAS view tied to that Sofa. The TCAS DocumentAnnotation features such as language still need to be set as usual.

Instantiate a TCAS view of a Sofa in a Sofa-aware component off a base CAS named aCAS with:

```
TCAS aTCAS = aCAS.getTCAS(aSofaFS);
```

A TCAS view has the same properties as the TCAS in previous UIMA versions. Although derived from CAS it is very lightweight as most of its CAS structures are simply references to those in the base CAS. Unique for each TCAS view is the Sofa it is tied to and its Index Repository object.

> **Note:** Feature Structures subsumed by `uima.tcas.Annotation` cannot be created off of the base CAS; these can only be created within a TCAS view.

A TCAS is a subtype of a CAS. The CAS object contains all the views; different views can be materialized from any other view.

If you have a TCAS object, say `aTCAS`, you can access its associated SofaFS using:

```
SofaFS aSofaFS = aTCAS.getSofa();
```

You can also iterate over all the Sofas, by obtaining an FSIterator over all the Sofa Feature Structures:

```
FSIterator sofaIterator = aCAS.getSofaIterator();
```

7.2.2 Each Sofa has its own Index Repository

Index repositories hold indexes, which in turn, hold references to Feature Structure instances in the CAS. Each TCAS view has a separate copy of the indexes; when you use an particular index, you always get it from a particular repository belonging to one particular view.

There is an additional Index Repository object used for the base CAS. The base Index Repository is used to index Sofa feature structures, and can be used by applications to reference feature structures that describe the original artifact independently of any particular view, feature structures that relate multiple Sofas to each other, etc.

7.2.3 Non Text TCAS

Historically the “T" in TCAS stood for text. Now a TCAS view is tied to a Sofa feature structure whose Sofa data may be a feature structure array of binary data or it may be remote from the CAS. When Sofa data is not a Java String the TCAS document will be null. Although annotation type feature structures can be created off such TCAS, the annotation method getCoveredText() will return null for these annotations.

7.2.4 Getting a JCas

As before, the following method gives a JCas object interface for both CAS and TCAS objects:

```
JCas aJCas = aCAS_or_TCAS.getJCas();
```

A JCas object view for a particular Sofa can be obtained with

```
JCas aJCas = aCAS.getJCas(aSofaFS);
```

If a JCas exists for this CAS or TCAS, it is returned; otherwise a JCas is created for this CAS or TCAS and returned.

7.2.5 Do UIMA Components Receive a CAS or a TCAS?

UIMA components here include annotators, Collection Readers and CAS Consumers.

The process() method exposed by an annotator will receive a TCAS if its class implements “TextAnnotator"; it will receive a CAS if it implements “GenericAnnotator". This corresponds to the process method definitions of TextAnnotator and GenericAnnotator classes.

Annotators that implement “JTextAnnotator" will receive a JCas, associated with an underlying TCAS.

Unlike annotators, however, Collection Readers and CAS Consumers have no such differentiation and only expose CAS interfaces.

For backward compatibility with Sofa-unaware Collection Readers, a default Sofa is created under the covers and its TCAS view delivered to the getNext() method. A Collection Reader becomes Sofa-aware and receives a base CAS by adding an output Sofa to the capabilities section of its descriptor. See ***Declaring Sofas in Component Descriptors*** on page 7-162 for explicit details.

The situation is similar with CAS Consumers, which receive a base CAS in the process() method only if one or more input Sofas are declared in its capabilities, otherwise a TCAS is received.

7.2.6 The Default Text Sofa

For backwards compatibility with TCAS components, there is a default text Sofa which can be created using:

```
TCAS aTCAS = aCAS.getTCAS(); // no Sofa argument - means default text Sofa
```

This creates or gets (if it is already created) a TCAS which has the "default text Sofa". The default text Sofa is determined by Sofa mapping -- see ***Specifying the Sofa for a Sofa-unaware TCAS processor*** on page 7-167.

7.3 Sofa Name Mapping

Sofa Name mapping is the mechanism which enables CAS component developers to choose locally meaningful Sofa names in their source code and let aggregate and collection processing engine developers connect output Sofas created in one component to input Sofas required in another.

7.3.1 mapToSofaID() method

Sofa mapping establishes a full mapping tree from the primitive component declaration of a Sofa up through zero or more aggregate descriptors and finally to the CPE descriptor. The mapping tree is then used in the method:

```
SofaID sofaID = getUimaContext().mapToSofaID("localName");
```

This method maps the local component name to the highest level defined mapping for that Sofa. The highest level mapping would usually be in the CPE descriptor, but could be in an aggregate descriptor or even specified in the “additional parameter" option of the UIMAFramework method used to produce the component.

If no mapping is defined for a Sofa, mapToSofaID returns a SofaID set to the input string.

The name returned by mapToSofaID is the actual name used for the Sofa in the CAS. All Sofas in a CAS must have unique names. This can be accomplished by mapping all declared Sofas as described in the following sections, or by inspection. The framework will throw an exception if createSofa() is called with a Sofa name that is already in use.

7.3.2 Name Mapping in an Aggregate Descriptor

For all components of an Aggregate, the mapping specifies a map between component Sofa names, and names at the aggregate level.

Here's an example. Assume two Sofa-aware annotators to be assembled into an aggregate which takes audio segments consisting of spoken English and produces a German text translation.

The first aggregate node takes an audio segment as input Sofa and produces a text transcript as output Sofa. Such an annotator may support many languages, but be configured to process only one at a time, in this case English. To standardize the

inputs and outputs independent of the chosen language, the annotator designer might choose Sofa names to be “AudioInput" and “TranscribedText".

The second annotator is a mono-language translator designed to translate text from English to German. The developer might choose the input and output Sofa names to be “EnglishDocument" and “GermanDocument", respectively.

In order to hook these two annotators together, the following section would be added to the top level of the aggregate descriptor:

```
<sofaMappings>
  <sofaMapping>
    <componentKey>SpeechToText</componentKey>
    <componentSofaName>AudioInput</componentSofaName>
    <aggregateSofaName>SegementedAudio</aggregateSofaName>
  </sofaMapping>
  <sofaMapping>
    <componentKey>SpeechToText</componentKey>
    <componentSofaName>TranscribedText</componentSofaName>
    <aggregateSofaName>EnglishAudioTranscript</aggregateSofaName>
  </sofaMapping>
  <sofaMapping>
    <componentKey>EnglishToGermanTranslator</componentKey>
    <componentSofaName>EnglishDocument</componentSofaName>
    <aggregateSofaName>EnglishAudioTranscript</aggregateSofaName>
  </sofaMapping>
  <sofaMapping>
    <componentKey>EnglishToGermanTranslator</componentKey>
    <componentSofaName>GermanDocument</componentSofaName>
    <aggregateSofaName>GermanTranslation</aggregateSofaName>
  </sofaMapping>
</sofaMappings>
```

The Component Descriptor Editor supports Sofa name mapping in aggregates and simplifies the task. See ***Sofa name mappings*** on page 9-200 for details.

7.3.3 Name Mapping in a CPE Descriptor

The CPE descriptor aggregates together a Collection Reader (with an optional Cas Initializer), and CAS Processors (some of which can be CAS Consumers). Sofa mappings can be added to the `<collectionIterator>`, the `<casInitializer>` and the `<casProcessor>` CPE descriptor elements. Unlike the Sofa maps for Aggregates, the maps for the CPE descriptor are distributed among the XML markup for each of the parts (collectionIterator, casInitializer, casProcessor. Because of this, they do not use the `<componentKey>` element. Finally, rather than sub-elements for the parts, the XML markup for these uses attributes. See ***<sofaNameMappings> Element*** on page 21-301.

Here's an example. Let's use the aggregate from the previous section in a collection processing engine. Here we will add a Collection Reader that outputs audio segments in an output Sofa named "nextSegment". We'll add a CAS Consumer in the next section, ***Specifying the Sofa for a Sofa-unaware TCAS processor***.

For Collection Reader components, the Sofa mapping section is added to the <casInitializer> section:

```
<collectionReader>
  <collectionIterator>
    <descriptor>
    . . .
    </descriptor>
    <configurationParameterSettings>...</configurationParameterSettings>
  </collectionIterator>
  <casInitializer>
    <descriptor>
    . . .
    </descriptor>
    <configurationParameterSettings>...</configurationParameterSettings>
    <sofaNameMappings>
      <sofaNameMapping componentSofaName="nextSegment"
                       cpeSofaName="SegementedAudio"/>
      </sofaNameMappings>
  </casInitializer>
<collectionReader>
```

At this point the CAS Processor section for the aggregate does not need any Sofa mapping because the aggregate input Sofa has the same name, "SegementedAudio", as is being produced by the Collection Reader.

7.3.4 Specifying the Sofa for a Sofa-unaware TCAS processor

Let's now assume that the CAS Consumer to be used in our CPE is a Sofa-unaware component that expects the analysis results associated with the input TCAS, and that we want it to use the results from the translated German text Sofa. The following mapping added to the CAS Processor section for the CPE will instruct the CPE to instantiate a TCAS from the German text Sofa and pass it to the CAS Consumer:

```
<casProcessor>
  . . .
  <sofaNameMappings>
    <sofaNameMapping cpeSofaName="GermanTranslation"/>
  <sofaNameMappings>
</casProcessor>
```

In this situation the CPE has determined that this CAS Consumer should get a TCAS view because the component descriptor did not declare any input Sofa. Under the covers the CPE obtains the TCAS using the following code:

```
TCAS aTCAS = aCAS.getTCAS(); // no Sofa arg means get default text Sofa
```

The single element mapping syntax above simply overrides the name of the default text Sofa.

7.3.5 Name Mapping in a UIMA Application

Applications which instantiate UIMA components directly using the UIMAFramework methods can also create a top level Sofa mapping using the "additional parameters" capability.

```
input = new XMLInputSource("test.xml");
desc = UIMAFramework.getXMLParser()
            .parseAnalysisEngineDescription(input);
HashMap additionalParams = new HashMap();
//setup sofa name mappings using the api
HashMap sofamappings = new HashMap();
sofamappings.put("localName1","globalName1");
sofamappings.put("localName2","globalName2");
additionalParams.put(
                Resource.PARAM_SOFA_MAPPINGS, sofamappings);

//instantiate AE
anAnnotator = UIMAFramework
              .produceAnalysisEngine(desc,additionalParams);
```

7.3.6 Name Mapping in a Remote Service

Currently no configuration information is passed from a UIMA client to a remote service. Therefore the Sofa names expected by the service will those defined in the service capabilities. For services connected using SOAP transports (see ***Service Client Descriptors*** on page 20-290) it is a constraint on the CPE client application to make sure that the highest level Sofa names used by the CPE correspond to the names expected by the service.

Services using Vinci transports can still use Sofa mapping in the CPE descriptor. For this case the Sofa name is converted from the CPE name to the service capability name and back again during client XCAS serialization and deserialization.

7.4 Sofa Impact on XCAS Format

Applications using entirely Sofa-unaware components will use a CAS containing a single text Sofa with the default name. A CAS with this property is called a backwards-compatible-CAS and is serialized with no changes from UIMA v1.0.

A CAS containing other Sofa names output each Sofa in standard feature structure format.

7.5 Sofa Sample Application

The UIMA SDK contains a simple Sofa example application which demonstrates many Sofa specific concepts and methods. The source code for the application driver is in docs/examples/src/com/ibm/uima/examples/SofaExampleApplication.java and the Sofa-aware annotator is given in SofaExampleAnnotator.java in the same directory.

This sample application demonstrates a language translator annotator which expects an input text Sofa with the English document and creates an output text Sofa containing a German translation. Some of the key Sofa concepts illustrated here include:

- Access of multiple Sofas via TCAS views.
- Unique feature structure index space for each TCAS.
- Feature structures containing cross references between annotations in different Sofas.
- The strong affinity of annotations with a specific Sofa.

Annotator Descriptor

The annotator descriptor in docs/examples/descriptors/analysis_engine/SofaExampleAnnotator.xml declares an input Sofa with SofaID="EnglishDocument" and an output Sofa named “GermanDocument". A custom type “CrossAnnotation" is also defined:

```
<typeDescription>
  <name>sofa.test.CrossAnnotation</name>
  <description/>
  <supertypeName>uima.tcas.Annotation</supertypeName>
  <features>
    <featureDescription>
      <name>otherAnnotation</name>
      <description/>
      <rangeTypeName>uima.tcas.Annotation</rangeTypeName>
    </featureDescription>
  </features>
</typeDescription>
```

The CrossAnnotation type is derived from `uima.tcas.Annotation` and includes one new feature: a reference to another annotation.

Application Setup

The application driver instantiates an analysis engine, seAnnotator, from the annotator descriptor, obtains a new base CAS using that engine's CAS definition, and creates the expected input Sofa using:

```
CAS cas = seAnnotator.newCAS();
SofaFS ls = cas.createSofa(seAnnotator.getUimaContext()
               .mapToSofaID("EnglishDocument"), "text");
```

Since seAnnotator is a primitive component, and no Sofa mapping has been defined, the SofaID will be "EnglishDocument". Local Sofa data is set using:

```
ls.setLocalSofaData("this beer is good");
```

At this point the CAS contains all necessary inputs for the translation annotator and its process method is called.

Annotator Processing

Annotator processing consists of parsing the English document into individual words, doing word-by-word translation and concatenating the translations into a German translation. Analysis metadata on the English Sofa will be an annotation for each English word. Analysis metadata on the German Sofa will be a CrossAnnotation for each German word, where the otherAnnotation feature will be a reference to the associated English annotation.

Code of interest includes two TCAS views:

```
// get English text Sofa and open TCAS view
engTcas = aCas.getTCAS(aCas.getSofa(getContext()
              .mapToSofaID("EnglishDocument")));

// Create the output German text Sofa and open TCAS view
germTcas = aCas.getTCAS(aCas.createSofa(getContext()
               .mapToSofaID("GermanDocument"), "text"));
```

the indexing of annotations with the appropriate view:

```
engTcas.getIndexRepository().addFS(engAnnot);
. . .
germTcas.getIndexRepository().addFS(germAnnot);
```

and the combining of metadata belonging to different Sofas in the same feature structure:

```
// add link to English text
germAnnot.setFeatureValue(other, engAnnot);
```

Back in the Application, accessing the results of analysis

Analysis results for each Sofa are dumped independently by iterating over all annotations for each associated TCAS. For the English Sofa:

```
//get annotation iterator for this TCAS
FSIndex anIndex = eTcas.getAnnotationIndex();
FSIterator anIter = anIndex.iterator();
while (anIter.isValid()) {
  AnnotationFS annot = (AnnotationFS) anIter.get();
  System.out.println(" " + annot.getType().getName()
                         + ": " + annot.getCoveredText());
  anIter.moveToNext();
}
```

Iterating over all German annotations looks the same, except for the following:

```
if (annot.getType() == cross) {
  AnnotationFS crossAnnot =
          (AnnotationFS) annot.getFeatureValue(other);
  System.out.println("   other annotation feature: "
          + crossAnnot.getCoveredText());
}
```

Of particular interest here is the built-in Annotation type method getCoveredText(). This method uses the “begin" and “end" features of the annotation to create a substring from the TCAS document. This means that annotations must have an additional feature which points to a particular text Sofa.

The example program output is:

```
---Printing all annotations for English Sofa---
uima.tcas.DocumentAnnotation: this beer is good
uima.tcas.Annotation: this
uima.tcas.Annotation: beer
uima.tcas.Annotation: is
uima.tcas.Annotation: good

---Printing all annotations for German Sofa---
uima.tcas.DocumentAnnotation: das bier ist gut
sofa.test.CrossAnnotation: das
  other annotation feature: this
sofa.test.CrossAnnotation: bier
  other annotation feature: beer
sofa.test.CrossAnnotation: ist
  other annotation feature: is
sofa.test.CrossAnnotation: gut
  other annotation feature: good
```

7.6 Sofa API summary

The recommended way to get a TCAS view for a particular Sofa in a ***Sofa-unaware*** component is to specify the Sofa to use by Sofa-mapping in the XML descriptors,

and simply receive it as a parameter in the process call. The component in this case is completely unaware of Sofas.

Otherwise, for ***Sofa-aware*** components or applications, the following methods are used:

Getting the SofaID from the name of a Sofa, or all SofaIDs, for a component:

```
SofaID sofaID = aContextObject.mapToSofaID("sofa-name-in-this-component");
SofaID[] sofaIDs = aContextObject.getSofaMappings();
```

Creating a Sofa Feature Structure (holds references to the subject of analysis):

```
SofaFS sofaFS = aCAS.createSofa(sofaID, mimeType_string);
```

Getting existing Sofa Feature Structure(s) from a CAS:

```
SofaFS sofaFS = aCAS.getSofa(sofaID);
FSIterator sofaIterator = aCAS.getSofaIterator();
```

Getting existing Sofa Feature Structure(s) associated with a particular TCAS or JCas:

```
SofaFS sofaFS = aTCAS.getSofa();
SofaFS sofaFS = aJCas.getSofa();
```

Creating an existing TCAS or JCas view associated with a particular sofaFS from an existing CAS (or if the TCAS/JCas is already created, getting it):

```
TCAS aTCAS = aCAS.getTCAS(sofaFS);
JCas aJCas = aCAS.getJCas(sofaFS);
```

The recommended way to get TCAS views for a particular Sofa in an application is to use:

```
XMLInputSource in = new XMLInputSource("MyDescriptor.xml");
ResourceSpecifier specifier =
  UIMAFramework.getXMLParser().parseResourceSpecifier(in);
AnalysisEngine ae = UIMAFramework.produceAE(aSpecifier);

CAS aCAS = ae.newCAS();
SofaID sofaID = ae.getUimaContext().mapToSofaID("sofa-name-in-component");
SofaFS sofaFS = aCAS.createSofa(sofaID, mimeType_string);

TCAS aTCAS = aCAS.getTCAS(sofaFS);  // or
JCas aJCas = aCAS.getJCas(sofaFS);
```

Chapter 8 XMI and EMF Interoperability

8.1 Overview

In traditional object-oriented terms, a UIMA Type System is a class model and a UIMA CAS is an object graph. There are established standards in this area – specifically, UML is an OMG standard for class models and XMI (XML Metadata Interchange) is an OMG standard for the XML representation of object graphs.

Furthermore, the Eclipse Modeling Framework (EMF) is an open-source framework for model-based application development, and it is based on UML and XMI. In EMF, you define class models using a metamodel called Ecore, which is similar to UML. EMF provides tools for converting a UML model to Ecore. EMF can then generate Java classes from your model, and supports persistence of those classes in the XMI format.

The UIMA SDK now provides tools for interoperability with XMI and EMF. These tools allow conversions of UIMA Type Systems to and from Ecore models, as well as conversions of UIMA CASes to and from XMI format. This provides a number of advantages, including:

You can define a model using a UML Editor, such as Rational Rose or EclipseUML, and then automatically convert it to a UIMA Type System.

You can take an existing UIMA application, convert its type system to Ecore, and save the CASes it produces to XMI. This data is now in a form where it can easily be ingested by an EMF-based application.

More generally, we are adopting the well-documented, open standard XMI as the standard way to represent UIMA-compliant analysis results (replacing the UIMA-specific XCAS format). This use of an open standard enables other applications to more easily produce or consume these UIMA analysis results.

For more information on XMI, see Grose et al. Mastering XMI. Java Programming with XMI, XML, and UML. John Wiley & Sons, Inc. 2002.

For more information on EMF, see Budinsky et al. Eclipse Modeling Framework 2.0. Addison-Wesley. 2006.

For details of how the UIMA CAS is represented in XMI format, see the ***XMI CAS Serialization Reference*** on 27-357.

8.2 Converting an Ecore Model to or from a UIMA Type System

The UIMA SDK provides the following two classes:

`Ecore2UimaTypeSystem:` converts from an .ecore model developed using EMF to a UIMA-compliant TypeSystem descriptor. This is a Java class that can be run as a standalone program or invoked from another Java application. To run as a standalone program, execute:

```
java com.ibm.uima.ecore.Ecore2UimaTypeSystem <ecore file> <output file>
```

The input .ecore file will be converted to a UIMA TypeSystem descriptor and written to the specified output file. You can then use the resulting TypeSystem descriptor in your UIMA application.

`UimaTypeSystem2Ecore:` converts from a UIMA TypeSystem descriptor to an .ecore model. This is a Java class that can be run as a standalone program or invoked from another Java application. To run as a standalone program, execute:

```
java com.ibm.uima.ecore.UimaTypeSystem2Ecore
      <TypeSystem descriptor> <output file>
```

The input UIMA TypeSystem descriptor will be converted to an Ecore model file and written to the specified output file. You can then use the resulting Ecore model in EMF applications. The converted type system will include any `<import...>`ed TypeSystems; the fact that they were imported is currently not preserved.

To run either of these converters, your classpath will need to include the UIMA jar files as well as the following jar files from the EMF distribution: common.jar, ecore.jar, and ecore.xmi.jar.

Also, note that the uima_core.jar file contains the Ecore model file uima.ecore, which defines the built-in UIMA types. You may need to use this file from your EMF applications.

8.3 Using XMI CAS Serialization

The UIMA SDK provides XMI support through the following two classes:

`XmiCasSerializer:` can be run from within a UIMA application to write out a CAS to the standard XMI format. The XMI that is generated will be compliant with the Ecore model generated by `UimaTypeSystem2Ecore`. An EMF application could use this Ecore model to ingest and process the XMI produced by the XmiCasSerializer.

`XmiCasDeserializer:` can be run from within a UIMA application to read in an XMI document and populate a CAS. The XMI must conform to the Ecore model generated by `UimaTypeSystem2Ecore`.

Also, the uima_examples Eclipse project contains some example code that shows how to use the serializer and deserializer:

`com.ibm.uima.examples.xmi.XmiWriterCasConsumer:` This is a CAS Consumer that writes each CAS to an output file in XMI format. It is analogous to the XCasWriter CAS Consumer that has existed in prior UIMA versions, except that it uses the XMI serialization format.

`com.ibm.uima.examples.xmi.XmiCollectionReader:` This is a Collection Reader that reads a directory of XMI files and deserializes each of them into a CAS. For example, this would allow you to build a Collection Processing Engine that reads XMI files, which could contain some previous analysis results, and then do further analysis.

Finally, in under the folder `uima_examples/ecore_src` is the class `com.ibm.uima.examples.xmi.XmiEcoreCasConsumer`, which writes each CAS to XMI format and also saves the Type System as an Ecore file. Since this uses the `UimaTypeSystem2Ecore` converter, to compile it you must add to your classpath the EMF jars common.jar, ecore.jar, and ecore.xmi.jar – see ecore_src/readme.txt for instructions.

Part III: Tool User's Manuals

Chapter 9 Component Descriptor Editor User's Guide

The Component Descriptor Editor is an Eclipse plug-in that provides a forms-based interface for creating and editing several kinds of UIMA descriptors.

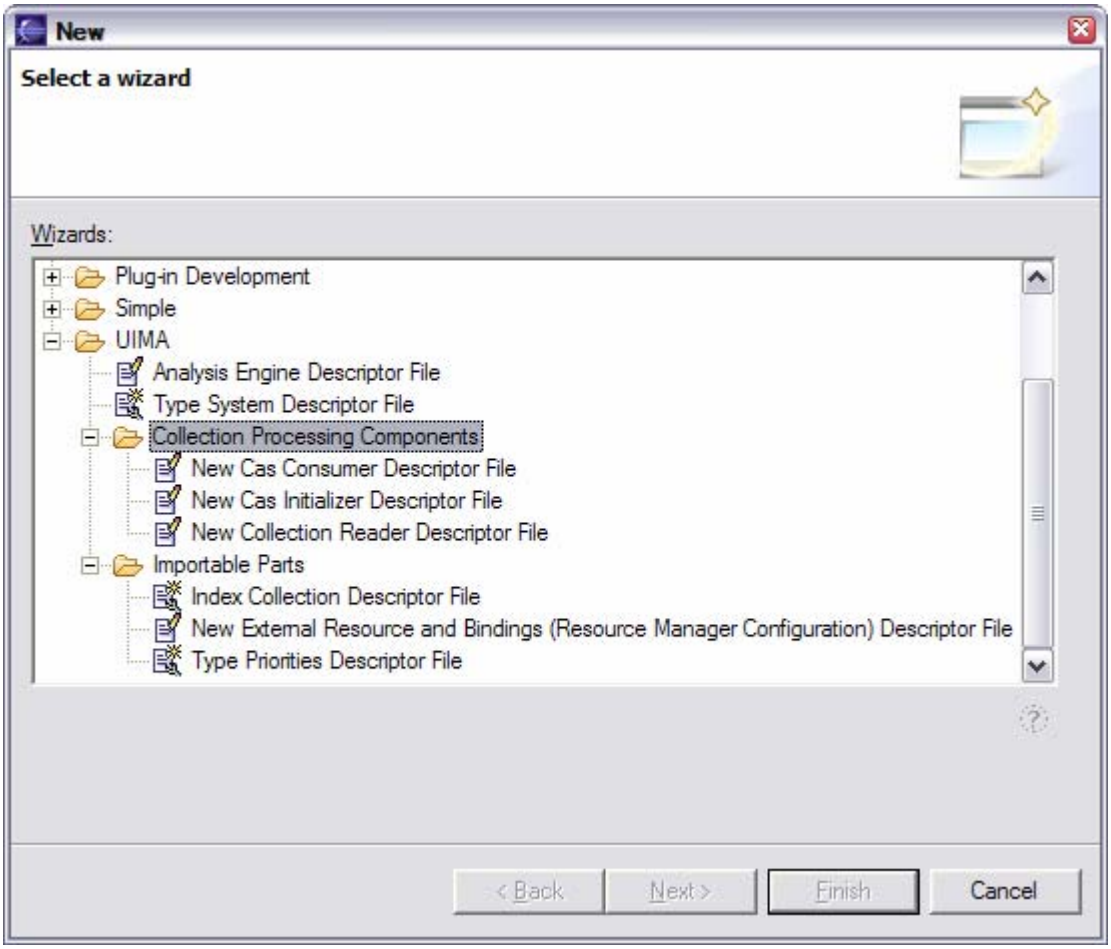
9.1 Launching the Component Descriptor Editor

Here's how to launch this tool on a descriptor contained in the examples. This presumes you have installed the examples as described in the SDK Installation and Setup chapter.

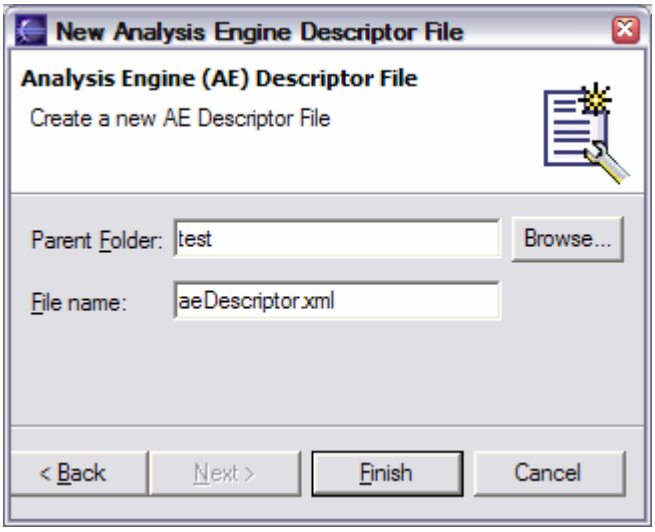
- Expand the uima_examples project in the Eclipse Navigator or Package Explorer view
- Within this project, browse to the file descriptors/tutorial/ex1/RoomNumberAnnotator.xml.
- Right-click on this file and select Open With → Component Descriptor Editor. (If this option is not present, check to make sure you installed the plug-ins as described on page 3-45, ***Install the UIMA Eclipse Plugins***, above).
- This should open a graphical editor and display the contents of the RoomNumberAnnotator descriptor.

9.2 Creating a New AE Descriptor

A new AE descriptor file may be created by selecting the File->New->Other... menu. This brings up the following dialog:

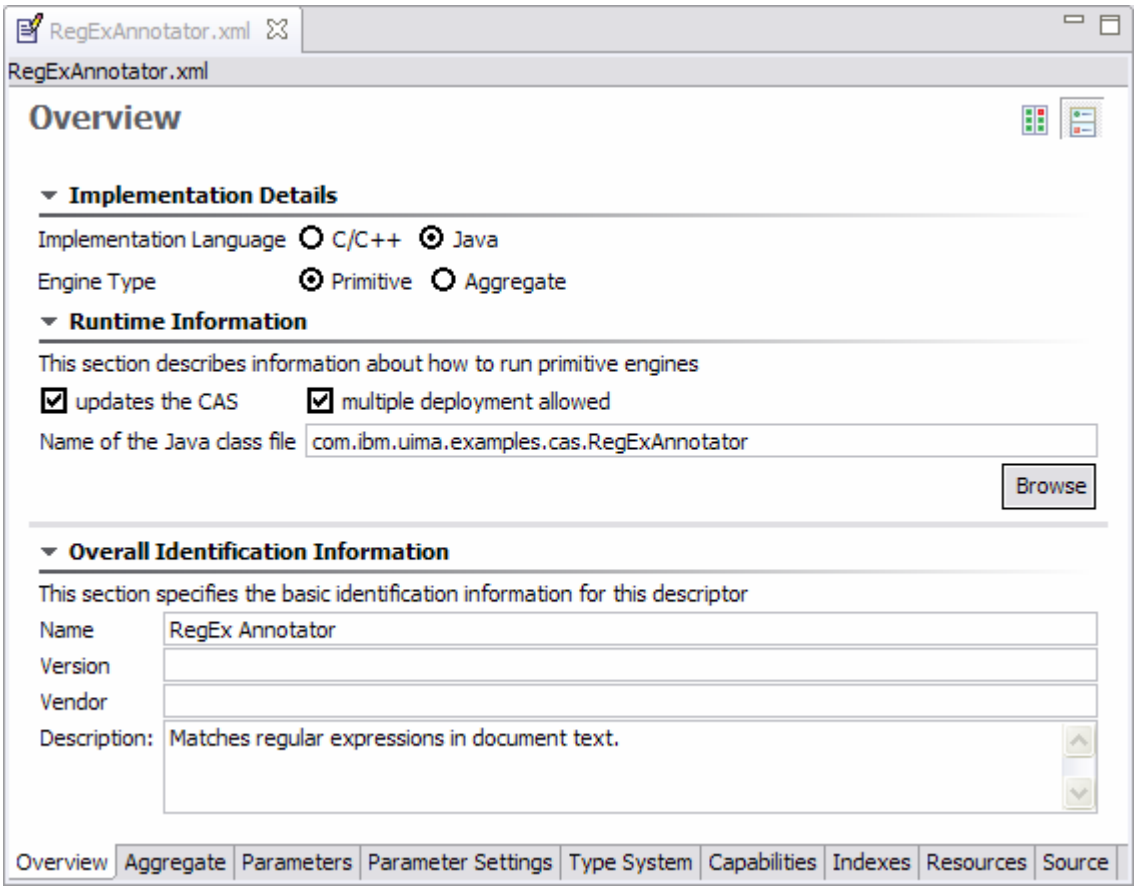


If the user then selects UIMA and Analysis Engine Descriptor File, and clicks the Next > button, the following dialog is displayed. We will cover creating other kinds of components later in the documentation.



After entering the appropriate parent folder and file name, and clicking Finish, an initial AE descriptor file is created with the given name, and the descriptor is opened up within the Component Descriptor Editor.

At this point, the display inside the Component Descriptor Editor is the same whether one started by creating a new AE descriptor, as in the preceding paragraph, or one merely opened a previously created AE descriptor from, say, the Package Explorer view. We show a previously created AE in the figure below:



To see all the information shown in the main editor pane with less scrolling, double click the title tab to toggle between the "full screen" and normal views.

It is possible to set the Component Descriptor Editor as the default editor for all .xml files by going to Window->Preferences, and then selecting File Associations on the left, and *.xml on the right, and finally by clicking on Component Descriptor Editor, the Default button and then OK. If AE and Type System descriptors are not the primary .xml files you work with within the Eclipse environment, we recommend not setting the Component Descriptor Editor as your default editor for all .xml files. To open an .xml file using the Component Descriptor Editor, if the Component Descriptor Editor is not set as your default editor, right click on the file in the Package Explorer, or other navigational view, and select Open With-> Component

Descriptor Editor. This choice is remembered by Eclipse for subsequent open operations.

9.3 Pages within the Editor

The Component Descriptor Editor follows a standard Eclipse paradigm for these kinds of editors. There are several pages in the editor; each one can be selected, one at a time, by clicking on the bottom tabs. The last page contains the actual XML source file being edited, and is displayed as plain text.

The same set of tabs appear at the bottom of each page in the Component Descriptor Editor. The Component Descriptor Editor uses this "multi-page editor" paradigm to give the user a view of conceptually distinct portions of the Descriptor metadata in separate pages. At any point in time the user may click on the Source tab to view the actual XML source. The Component Descriptor Editor is, in a way, just a fancy GUI for editing the XML. The tabs provide quick access to the following pages: Overview, Aggregate, Parameters, Parameter Settings, Type System, Capabilities, Indexes, Resources, and Source. We discuss each of these pages in turn.

9.3.1 Adjusting the display of pages

Most pages in the editor have a "sash" bar. This is a light gray bar which separates sub-sections of the page. This bar can be dragged with the mouse to adjust how the display area is split between the two sash panes. On some pages, you can also change the orientation of the Sash so it splits vertically, instead of horizontally.

All of the sections on a page have subtitles, with an indicator to the left which you can click to collapse or expand that particular section. Collapsing sections can sometimes be useful to free up screen area for other sections.

9.4 Overview Page

Normally, the first page displayed in the Component Descriptor Editor is the Overview page (the name of the page is shown in the GUI panel at the top left). If there is an error reading and parsing the source, the Source page is shown instead, giving you the opportunity to correct the problem. For many components, the Overview page contains three sections: Implementation Details, Runtime Information and overall Identification Information.

Implementation Details

In the Implementation Details section you specify the Implementation Language and Engine Type. There are two kinds of Engines: Aggregate, and non-Aggregate (also called Primitive). An Aggregate engine is one which is composed of additional

component engines and contains no code, itself. Several of the pages in the Component Descriptor Editor have different formats, depending on the engine type.

Runtime Information

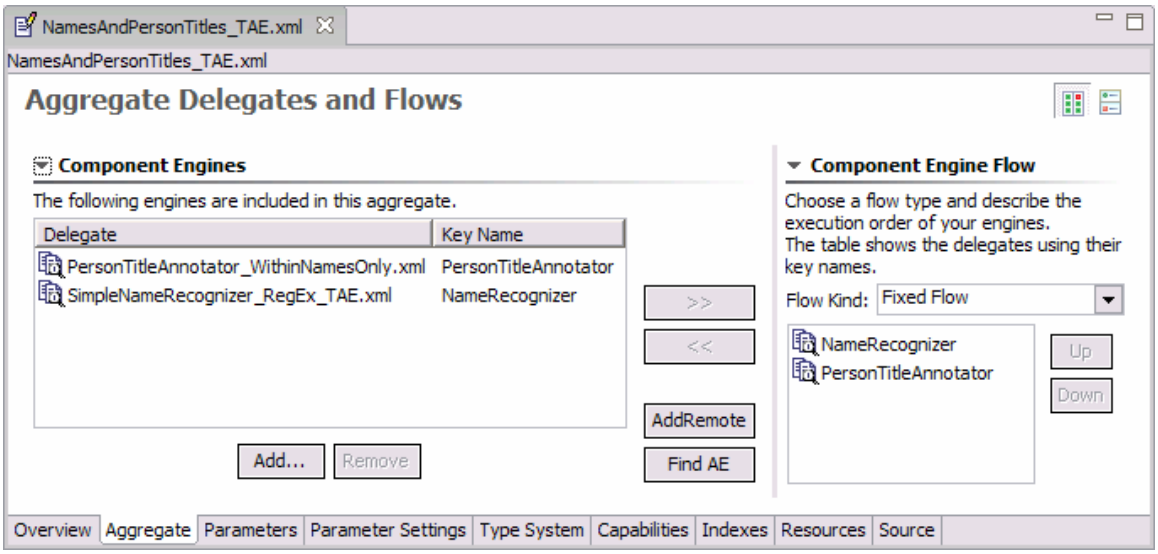
Runtime information is only applicable for primitive engines and is disabled for aggregates. This is where you specify the .class name of the annotator implementation, if you are doing a Java implementation. This documentation always assumes you are doing a Java implementation. Most Analysis Engines will specify that they update the CAS, and that they can be replicated when deployed for performance. If a particular Analysis Engine must see every CAS (for instance, if it is counting the number of CASes), then uncheck the "multiple deployment allowed" box. If the Analysis Engine doesn't update the CAS, uncheck the "updates the CAS" box. (Most CAS Consumers do not update the CAS).

Overall Identification Information

The Name should be a human-readable name that describes this component. The Version, Vendor, and Description fields are optional, and are arbitrary strings.

9.5 Aggregate Page

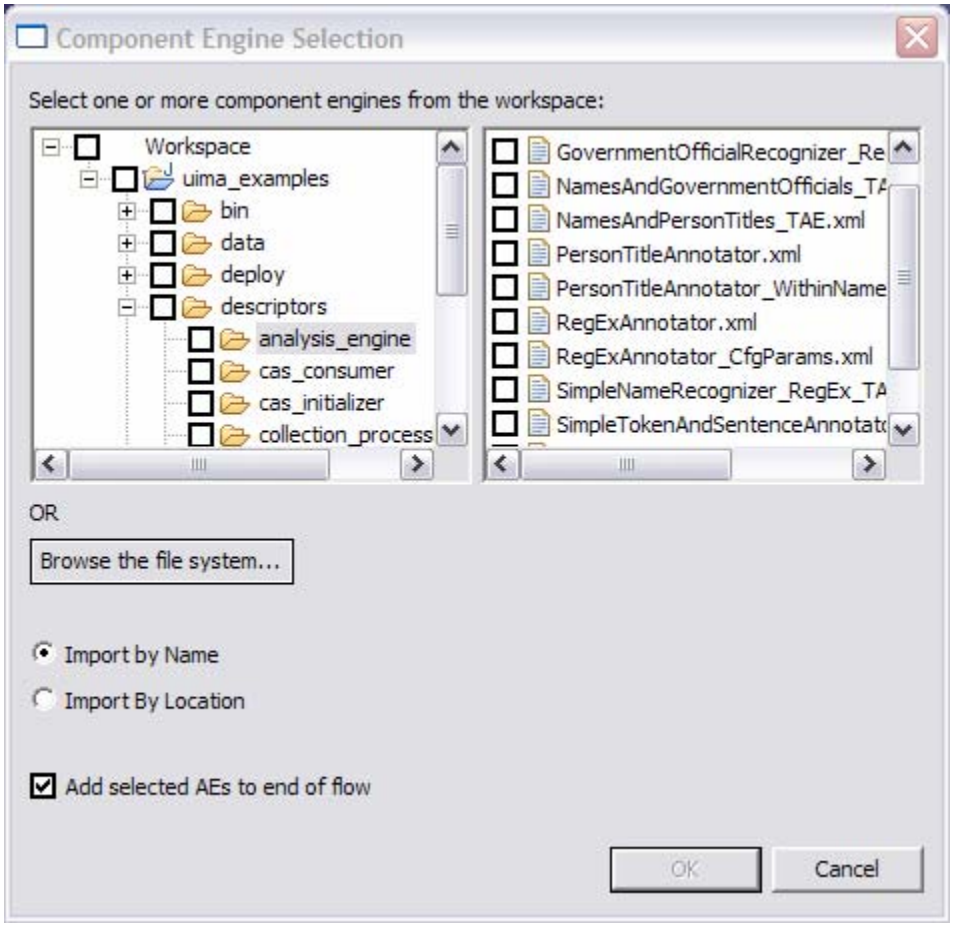
For primitive Analysis Engines or Collection Processing components, the Aggregate page is not used. For aggregate engines, the page looks like this:



On the left we see a list of component engines, and on the right the flow order. If you hover the mouse over an item in the list of component engines, that engine's description meta data will be shown. If you right-click on one of these items, you

get an option to open that delegate descriptor in another editor instance. Any changes you make, however, won't be seen until you close and reopen the editor on the importing file.

Engines can be added to the list on the left by clicking the Add button at the bottom of the Component Engine section. This brings up the following dialog:



This is a rather complex dialog, but the basic idea is that it enables you to select multiple descriptors from various levels of your workspace. As you select descriptors, highlight the appropriate directory on the left, and select, or multi-select descriptors on the right. To select all descriptors in a particular directory, you may place a checkmark next to the directory on the left. It is also possible to add descriptors that are not part of your Eclipse workspace by clicking the Browse file system... button. It is only possible to add one descriptor at a time when picking from outside of the Eclipse workspace.

You can specify that the import should be by Name (the name is looked up using both the Project's class path, and DataPath), or by location. If it is by name, it may

contain part of the path within the name. For instance, if the file name picked is `c:/project/subproject/src/com/company/prod/xyz.xml`, and the class path includes `c:/project/subproject/src`, the name in the descriptor will be `"com.company.prod.xyz"`. If it is by location, the file reference is converted to a relative reference if possible, in the descriptor.

The final selection at the bottom tells whether or not the selected engine(s) should automatically be added to the end of the flow section (the right section on the Aggregate page). The OK button does not become activated until at least one descriptor file is selected.

To remove an analysis engine from the component engine list simply select an engine and click the Remove button, or press the delete key. If the engine is already in the flow list you will be warned that deletion will also delete the specified engine from this list.

Adding components more than once

Components may be added to the left panel more than once. Each of these components will be given a key which is unique. A typical reason this might be done is to use a component in a flow several times, but have each use be associated with different configuration parameters (different configuration parameters can be associated with each instance).

Adding or Removing components in a flow

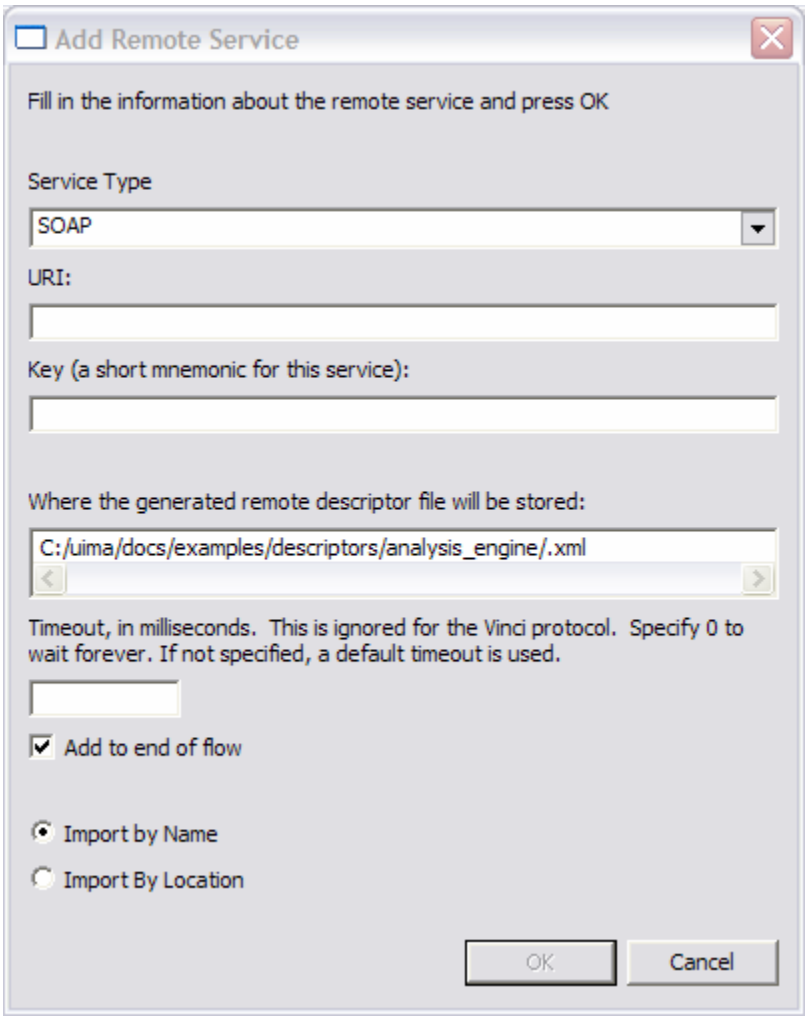
The button in-between the Component Engines and the Flow List, labeled **>>,** adds a chosen engine to the flow list and the button labeled << removes an engine from the flow list. To add an engine to the flow list you must first select an engine from the left hand list, and then press the >> button. Engines may appear any number of times in the flow list. To remove an engine from the flow list, select an engine from the right hand list and press the << button.

Adding remote Analysis Engines

There are two ways to add remote engines: add an existing descriptor, which specifies a remote engine (just as if you were adding a non-remote engine) or use the Add Remote button which will create a remote descriptor, save it, and then import it, all in one operation. The Add Remote button enables you to easily specify the information needed to create a Service Client descriptor for a remote AE - one that runs on a different computer connected over the network. The Service Client descriptor is described on page 20-290. The Add Remote button creates this descriptor, saves it as a file in the workspace, and imports it into the aggregate.

Of course, if you already have a Service Client descriptor, you can add it to the set of delegates, just like adding other kinds of analysis engines.

After clicking on Add Remote, the following dialog is displayed:



To define a remote service you specify the Service Type, URI and Key. You can also specify a Timeout in milliseconds. Just like when one adds an engine from the file system, you have the option of adding the engine to the end of the flow. The Component Descriptor Editor currently only supports Vinci and SOAP services using this dialog.

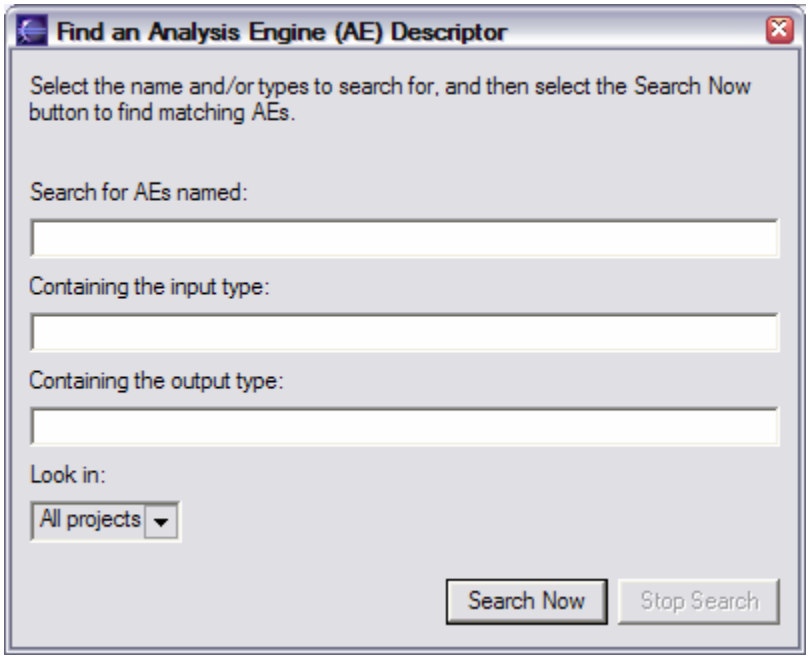
Remote engines are added to the descriptor using the <import ... > syntax. The information you specify here is saved in the Eclipse project as a file, using a generated name, <key-name>.xml, where <key-name> is the name you listed as the Key. Because of this, the key-name must be a valid file name. If you want a different name, you can change the path information in the dialog box.

Connecting to Remote Services

If you are using the Vinci protocol, it requires that you specify the location of the Vinci Name Server (an IP address and a Port number). You specify these, globally, for your Eclipse workspace, using the Eclipse menu item: Window -> Preferences... -> UIMA Preferences. If the remote service is available, additional operations become possible. For instance, hovering the mouse over the remote descriptor will show the description metadata from the remote service.

Finding Analysis Engines by searching

The next button that appears between the component engine list and the flow list is the Find AE button. When this button is pressed the following dialog is displayed, which allows one to search for AEs by name, by input or output types, or by a combination of these criteria. This function searches the existing Eclipse workspace for matching *.xml descriptor source files; it does not look inside Jar files.



The search automatically adds a "match any characters" - style (*) wildcard at the beginning and end of anything entered. Thus, if person is specified for an output type, a "*person*" search is performed. Such a search would match such things as "my.namespace.person" and "person.governmentOfficial." One can search in all projects or one particular project. The search does an implicit *and* on all fields which are left non-blank.

Component Engine Flow

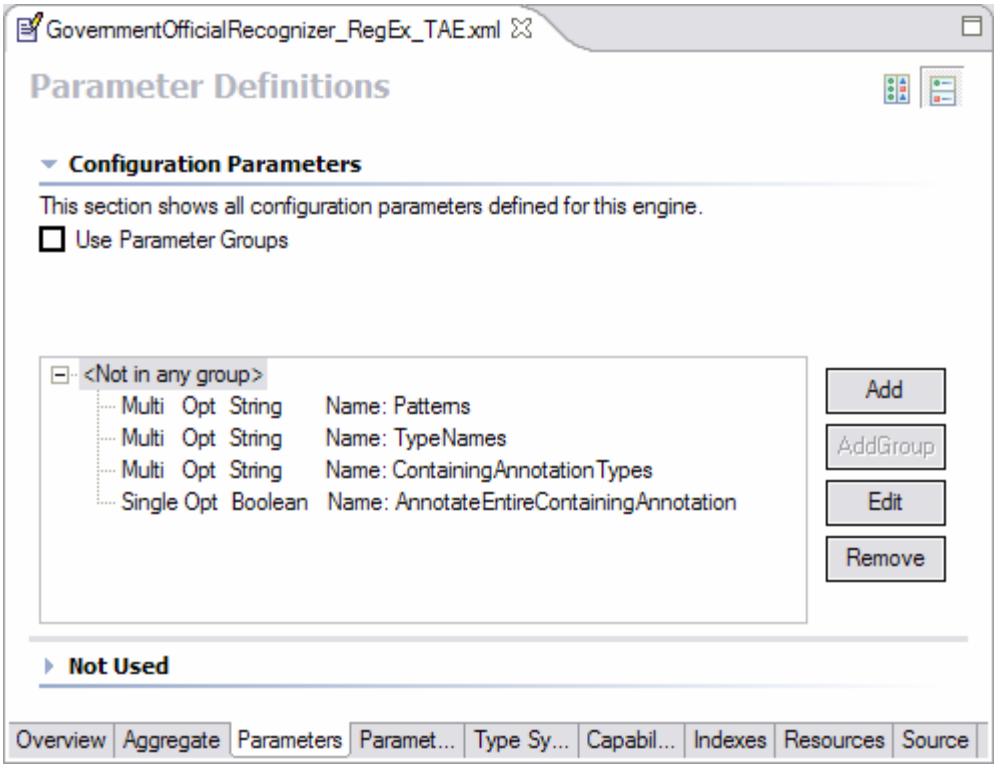
The UIMA SDK currently supports two kinds of sequencing flows: Fixed, and CapabilityLanguageFlow (see ***Capability Language Flow*** on page 20-283). Both of

these flows require specification of a linear flow sequence. The Component Engine Flow section allows specification of these items.

The pull-down labeled Flow Kind picks between the two flow models. The Up and Down buttons to the right in the Flow section are activated when an engine in the flow is selected. The Up button moves the selected engine up one place in the execution order, and down moves the selected engine down one place in the execution order. It is worth repeating that engines can appear multiple times in the flow (or not at all).

9.6 Parameters Definition Page

There are two pages for parameters: the first one is where parameters are defined, and the second one is where the parameter settings are configured. The first page is the Parameter Definition page and has two alternatives, depending on whether or not the descriptor is an Aggregate or not. We start with a description of parameter definitions for Primitive engines. Here is an example:

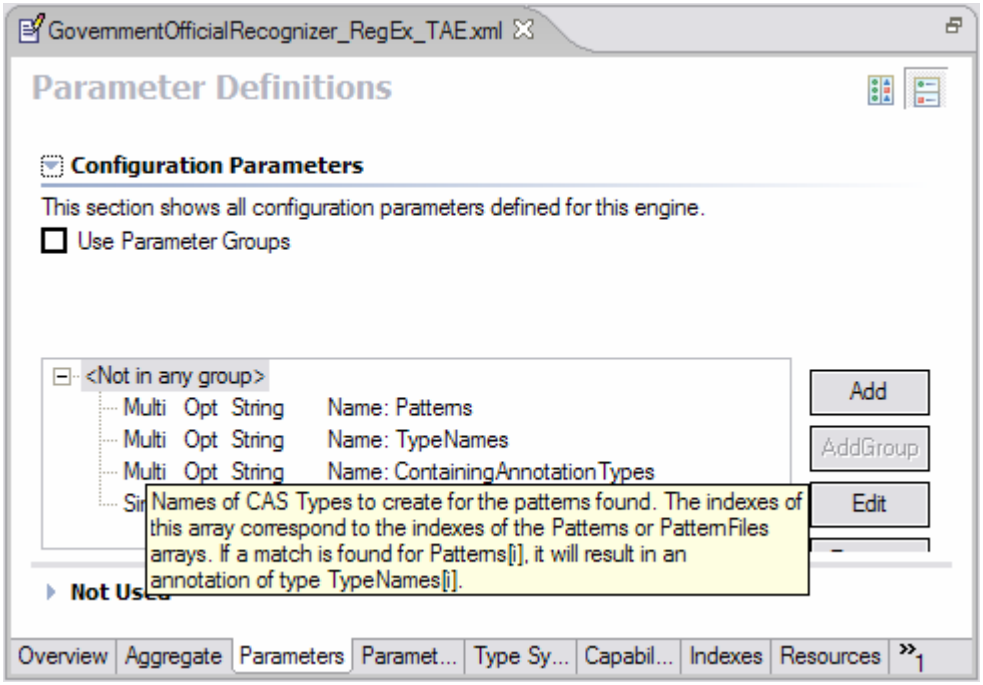


The first checkbox at the top simplifies things if you are not using Parameter Groups (see the following section for a discussion of groups). In this case, leave the check box unchecked. The main area shows a list of parameter definitions. Each parameter has a name, which must be unique for this Analysis Engine. The other three attributes specify whether the parameter can have a single or multiple values

(an array of values), whether it is Optional or Mandatory, and what the value type it can hold (String, Integer, Float, and Boolean).

In addition to using the buttons on the right to edit this information, you can double-click a parameter to edit it, or remove (delete) a selected parameter by pressing the delete key. Use the Add button to add a new parameter to the list.

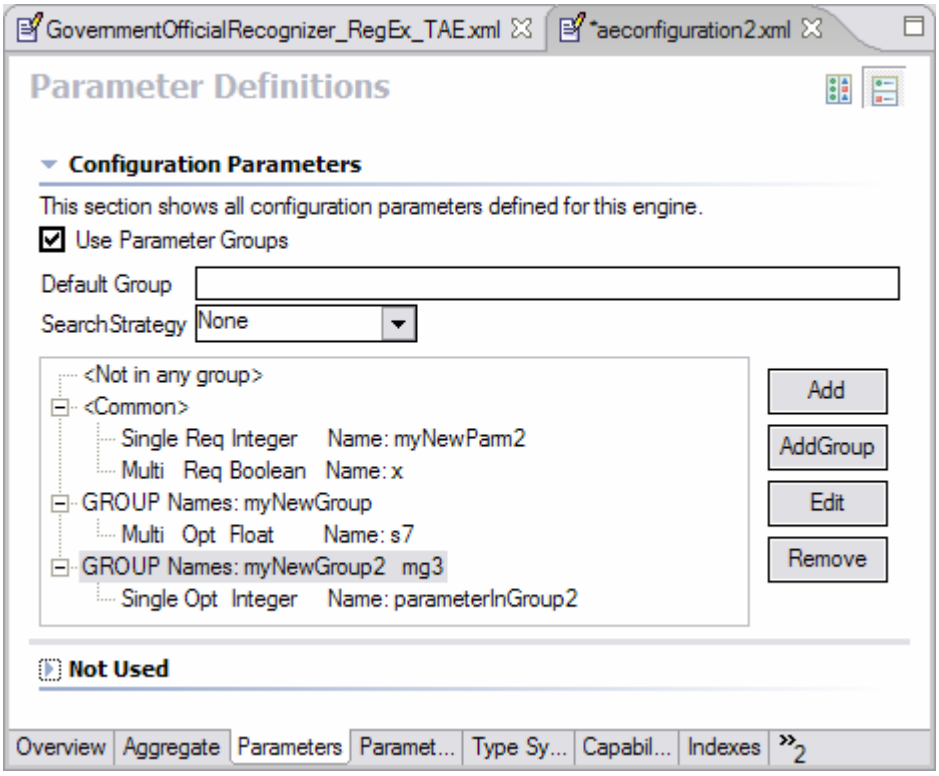
Parameters have an additional description field, which you can specify when you add or edit a parameter. To see the value of the description, hover the mouse over the item, as shown in the picture below:



Using groups

The group concept for parameters arose from the observation that sets of parameters were sometimes associated with different configuration needs. As an example, you might have an Analysis Engine which needed different configuration based on the language of a document.

To use groups, you check the "Use Parameter Groups" box. When you do this, you get the ability to add groups, and to define parameters within these groups. You also get a capability to define "Common" parameters, which are parameters which are defined for all groups. Here is a screen shot showing some parameter groups in use:



You can see the "<Common>" parameters as well as two different sets of groups.

The Default Group is an optional specification of what Group to use if the parameter is not available for the group requested.

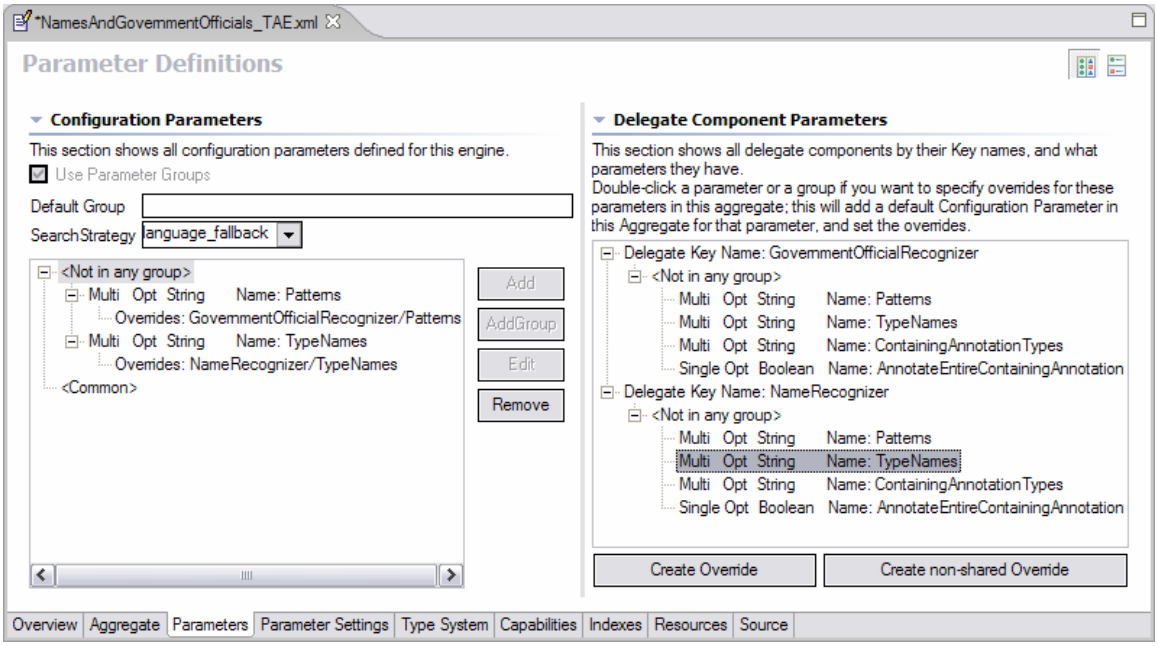
The Search strategy specifies what to do when a parameter is not available for the group requested. It can have the values of None, language_fallback, or default_fallback. These are more fully described in the section ***Configuration Parameter Declaration*** on page 20-267.

Groups are added using the Add Group button. Once added, they can be edited or removed, using the buttons to the right, or the standard gestures for editing (double-clicking the item) and removing (pressing the delete key after an item is selected). Removing a group removes all the parameter definitions in the group. If you try and remove the "<Common>" group, it just removes the parameters in the group.

Each entry for a group in the table specifies one or more group names. For example, the highlighted entry above, specifies two groups: "myNewGroup2" and "mg3". The parameter definition underneath is considered to be in both groups.

9.6.2 Parameter declarations for Aggregates

Aggregates declare parameters which always must override a parameter setting for a component Analysis Engine, making up the aggregate. They do this using the version of this page which is shown when the descriptor is an Aggregate; here's an example:



There is an additional panel shown (on the right) which lists all of the component Analysis Engines by their key names, and shows for each of them their defined parameters. To add a new override for one or more of these parameters to the aggregate, select the component parameter you wish to override and push the Create Override button (or, you can just double-click the component parameter). This will automatically add a parameter of the same name (by default – you can change the name if you like) to the aggregate, putting it into the same group(s) (if groups are being used in the component – this is required), and setting the properties of the parameter to match those of the component (this is required).

> **Note:** If the name of the parameter being added already is in use in the aggregate, and the parameters are not compatible, a new parameter name is generated by suffixing the name with a number. If the parameters are compatible, the selected component parameter is added to the existing aggregate parameter, as an additional override. If you don't want this behavior, but want to have a new name generated in this case, push the Create non-shared Override button instead, or hold down the "shift" key when double clicking the component parameter.

Note: The required / optional setting in the aggregate parameter is set to match that of the parameter being overridden. You may want to make an optional delegate parameter required. You can do this by changing that value manually in the source editor view.

In the above example, the user has just double-clicked the "TypeNames" parameter in the "NameRecognizer" component. This added that parameter to this aggregate under the "<Not in any group>" section – since it wasn't part of a group.

Once you have added a parameter definition to the aggregate, you can use the buttons on the right side of the left panel to add additional overrides or remove parameters or their overrides. You can also remove groups; removing a group is like removing all the parameter definitions in the group.

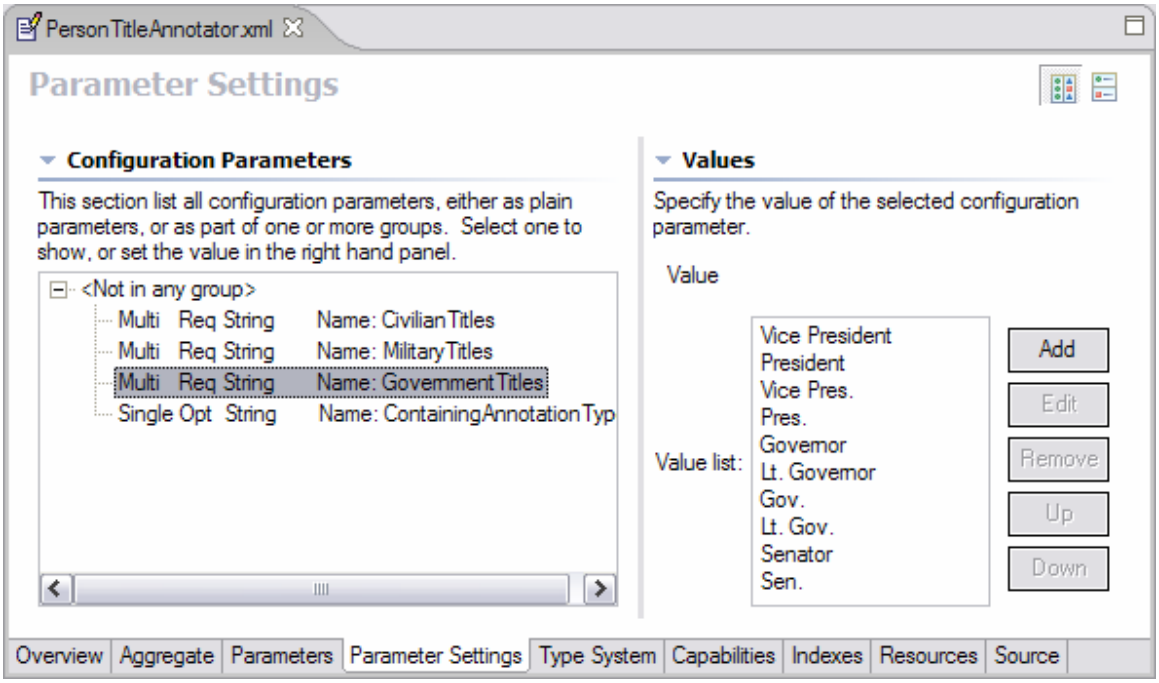
In addition to adding one parameter at a time from a component, you can also add all the parameters for a group within a component, or all the parameters in the component, by selecting those items.

If you double-click (or push Create Override) the "<Common>" group or a parameter in the <Common> group in a component, a special group is created in the Aggregate consisting of all of the groups in that component, and the overriding parameter (or parameters) are added to that. This is done because each component can have different groups belonging to the Common group notion; the Common group for a component is just shorthand for all the groups in that component.

The Aggregate's specification of the default group and search strategy override any specifications contained in the components.

9.7 Parameter Settings Page

The Parameter Settings page is rather straightforward; it is where the user defines parameter settings for their engines. An example of such a page is given below:



For single valued attributes, the user simply types the default value into the Value box on the right hand side. For multi-valued parameters the user should use the Add, Edit and Remove buttons to manage the list of multiple parameter values.

Values within groups are shown with each group separately displayed, to allow configuring different values for each group.

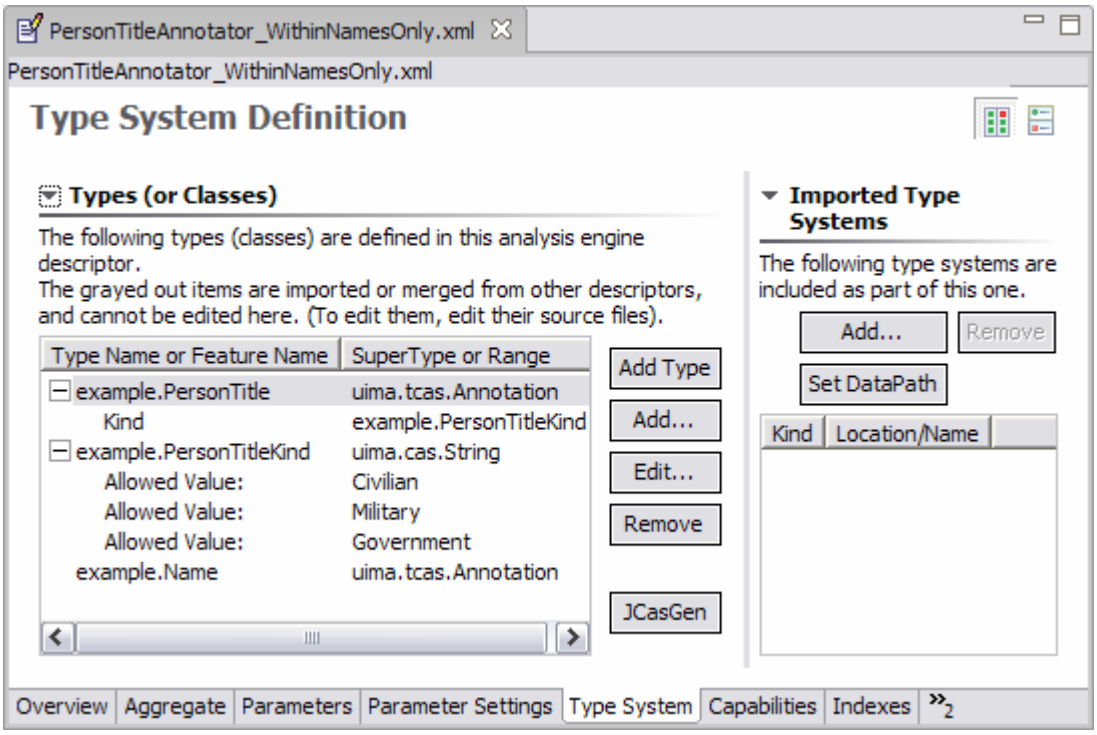
Values are checked for validity. For Boolean values in a list, use the words `true` or `false`.

> **Note:** If you specify a value in a single-valued parameter, and then delete all the characters in the value, the CDE will treat this as if you wanted to not specify any setting for this parameter. In order to specify a 0 length string setting for a String-valued parameter, you will have to manually edit the XML using the "Source" tab.
>
> For array valued parameters, if you remove all of the entries for a particular array parameter setting, the XML will reflect a 0-length array. To change this to an unspecified parameter setting, you will have to manually edit the XML using the "Source" tab.

9.8 Type System Page

This page declares the type system used by the annotator. For aggregates it is derived by merging the type systems of all constituent AEs. The types used by the AE constitute the language in which the inputs and outputs are described in the

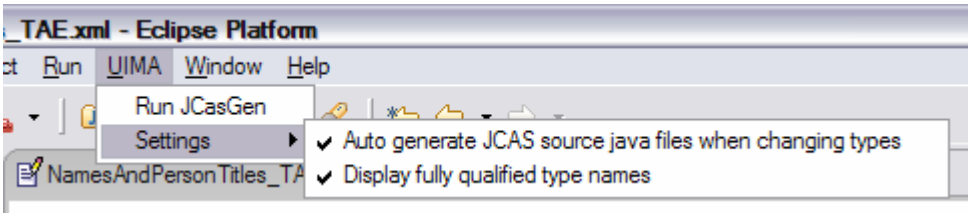
Capabilities page and also affect the choice of indexes on the Indexes page. The Type System page looks like the following:



Before discussing this page in detail, it is important to note that there are two settings that affect the operation of this page. These are accessed by selecting the UIMA->Settings (or by going to the Eclipse Window -> Preferences -> UIMA Preferences) and checking or unchecking one of the following: "Auto generate .java files when defining types" and "Display fully qualified type names."

When the Auto generate option is checked and the development language for the AE is Java, any time a change is made to a type and the change is saved, the corresponding .java files are generated using the JCasGen tool. The results are stored in the primary source directory defined for the project. The primary source directory is that listed first when you right click on your project and select Properties->Java Build Path, click on the Source tab and look in the list box under the text that reads: "Source folder on build path." If no source folders are defined, you will get a warning that you have no source folders defined and JCasGen will not be run. (For information on JCasGen see *Chapter 16* ***JCasGen User Guide***.) When JCasGen is run, you can monitor the progress of the generation by observing the status on the Eclipse status line (normally at the bottom of the Eclipse window). JCasGen runs on the fully-merged type system, consisting of the type specification plus any imported type system, plus (for aggregates) the merged type systems of all the components in an aggregate.

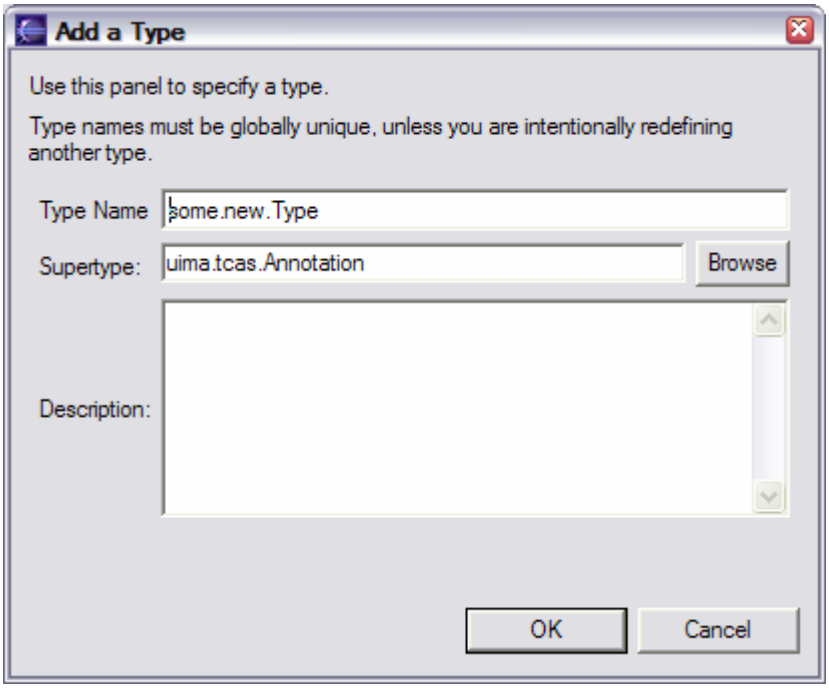
Note: In addition to running automatically, you can manually run JCasGen on the fully merged type system by clicking the JCasGen button, or by selecting Run JCasGen from the UIMA pulldown menu:



When "Display fully qualified type names" is left unchecked, the namespace of types is not displayed, i.e. if a fully qualified type name is my.namespace.person, only the abbreviated type name person will be displayed. In the Type page diagram shown above, "Display fully qualified type names" is in fact unchecked.

To add, edit, or remove types the buttons on the top left section are used. When adding or editing types, fully qualified type names should of course be used, regardless of whether the "Display fully qualified type names" is unchecked. Removing or editing a type will have a cascading effect in that the type removal/edit will effect inputs, outputs, indexes and type priorities in the natural way.

When a type is added, this dialog is shown:



Type names should be specified using a namespace. The namespace is like a Java package name, and serves to insure type names are unique. It also serves as the

package name for the generated JCas classes. The namespace name is the set of names up to the last period in the string.

The supertype must be picked from an existing type. The entry field for the supertype supports Eclipse-style content assist. To use it, put the cursor in the supertype field, and type a letter or two of the supertype name (lower case is fine), either starting with the name space, or just with the type name (without the name space), and hold down the Control key and then press the spacebar. When you do this, you can see a list of suitable matching types. You can then type more letters to narrow down your choices, or pick the right entry with the mouse.

To see the available types and pick one, press the Browse button. This will show the available types, and as you type letters for the type name (in lower case – capitalization is ignored), the available types that match are narrowed. When you've typed enough to specify the type you want, press Enter. Or you can use the list of matching type names and pick the one you want with the mouse.

Once you've added the type, you can add features to it by highlighting the type, and pressing the Add button.

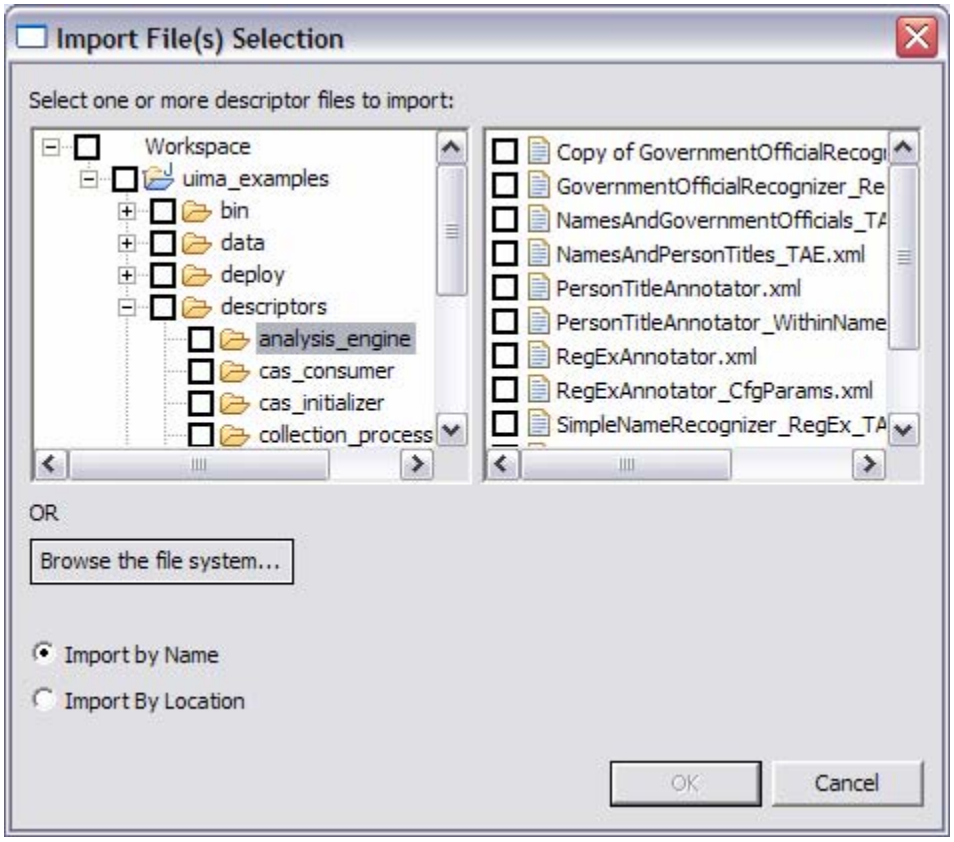
If the type being defined is a subtype of uima.cas.String, the Add button allows you to add allowed values for the string, instead of adding features.

To edit a type or feature, you can double click the entry, or highlight the entry and press the Edit button. To delete a type or feature, you highlight the entry to be deleted, and click the delete button or push the delete key.

It is also possible to import type systems for inclusion in your descriptor. To do this, use the Type Import panel's `Add...` button. This allows you to import a type system descriptor.

When importing by name, the name is resolved using the class path for the Eclipse project containing the descriptor file being edited, or by looking up this name in the UIMA DataPath. The DataPath can be set by pushing the Set DataPath button. It will be remembered for this Eclipse project, as a project Property, so you only have to set it once (per project). The value of the DataPath setting is written just like a class path, and can include directories or JAR files, just as is true for class paths.

The following dialog allows you to pick one or more files from the Eclipse workspace, or one file (at a time) from the file system:



This is essentially the same dialog as was used to add component engines to an aggregate. This dialog supports multi-selection – all selected items will be imported. To import from a type system descriptor that is not part of your Eclipse workspace, click the Browse the file system.... button.

Imported types are validated, and if OK, they are added to the list in the Imported Type Systems section of the Type System page. Any types they define are merged with the existing type system.

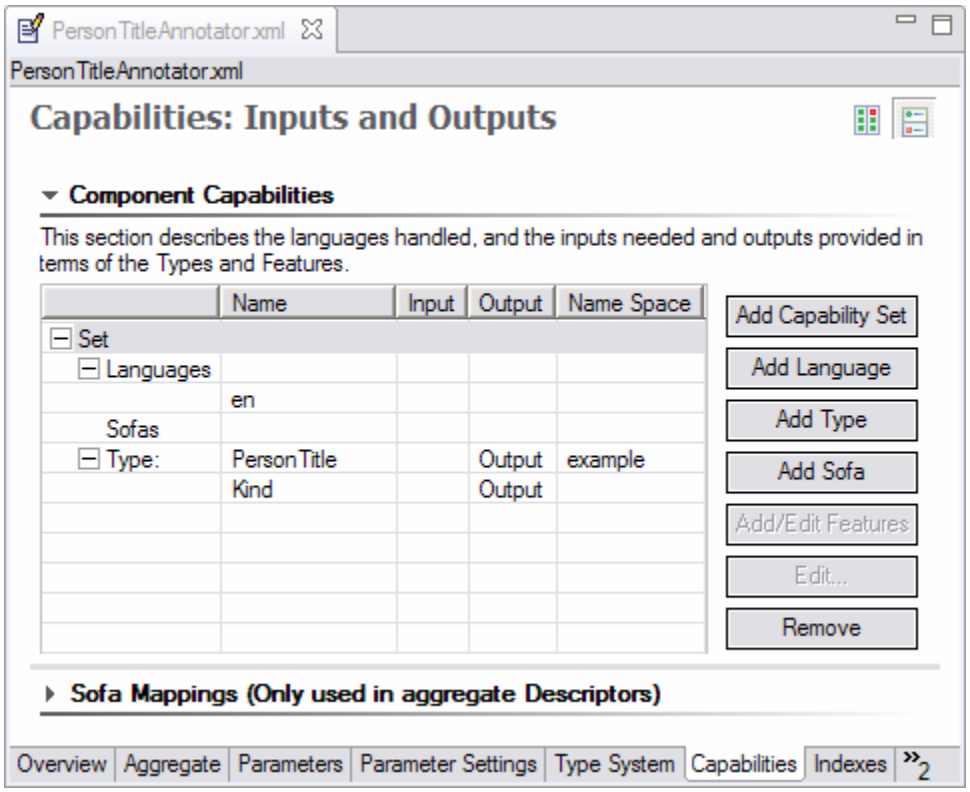
Imported types and features which are only defined in imports are shown in the Type System section, but in a grayed-out font; these type cannot be edited here. To change them, open up the imported type system descriptor, and change them there.

If you hover the mouse over an import specification, it will show more information about the import. If you right-click, it will bring up a context menu that allows opening the imported file in the Editor, if the imported file is part of the Eclipse workspace. Changes you make, however, won't be seen until you close and reopen the editor on the importing file.

It is not possible to define types for an aggregate analysis engine. In this case the type system is computed from the component AEs. The Type System information is shown in a grayed-out font.

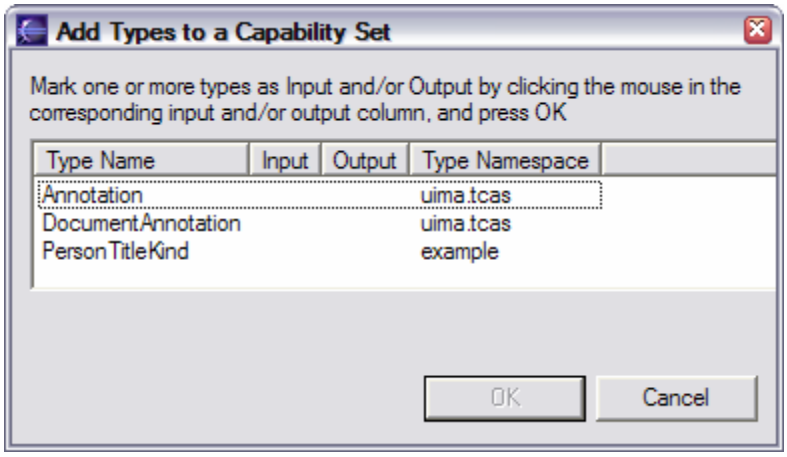
9.9 Capabilities Page

Capabilities come in "sets". You can have multiple sets of capabilities; each one specifies languages supported, plus inputs and outputs of the Analysis Engine. The idea behind having multiple sets is the concept that different inputs can result in different outputs. Many Analysis Engines, though, will probably define just one set of capabilities. A sample Capabilities page is given below:



When defining the capabilities of a primitive analysis engine, input and output types can be any type defined in the type system. When defining the capabilities of an aggregate the inputs must be a subset of the union of the inputs in the constituent analysis engines and the outputs must be a subset of the union of the outputs of the constituent analysis engines.

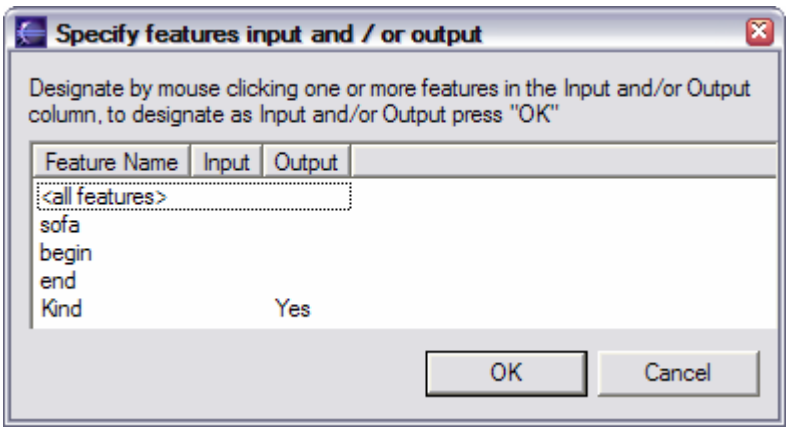
To add a type, first select something in the set you wish to add the type to, and press Add Type. The following dialog appears presenting the user with a list of types which are candidates for additional inputs:



Follow the instructions to mark the types as input and / or output (a type can be both). By default, the <all features> flag is set to true. If you want to specify a subset of features of a type, read on.

When types have features, you can specify what features are input and / or output. A type doesn't have to be an output to have an output feature. For example, an Analysis Engine might be passed as input a type Token, and it adds (outputs) a feature to the existing Token types. If no new Token instances were created, it would not be an output Type, but it would have features which are output.

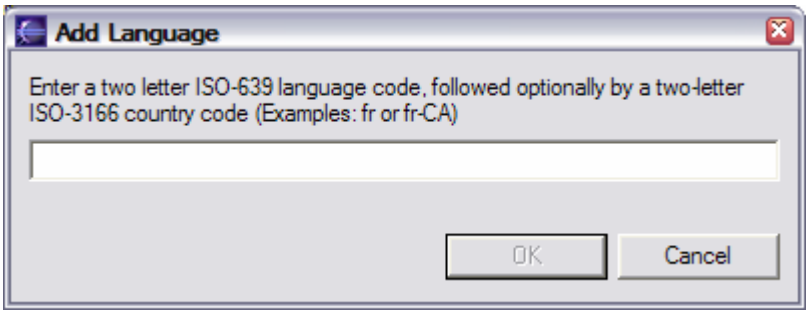
To specify features as input and / or output (they can be both), select a type, and press Add. The following dialog box appears:



To mark a feature as being input and / or output, click the mouse in the input and / or output column for the feature. If you select <all features>, it unmarks any individual feature you selected, since <all features> subsumes all the features.

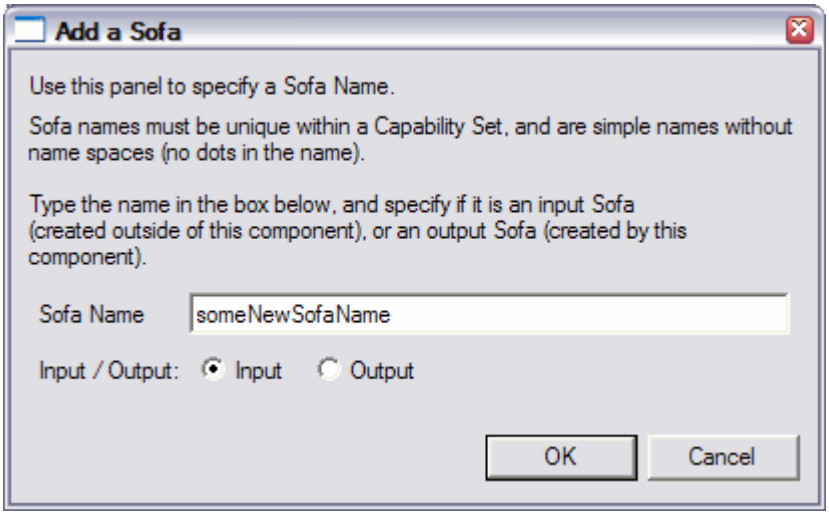
The Languages part of the capability is where you specify what languages are supported by the Analysis Engine. Supported languages should be listed using

either a two letter ISO-639 language code, or an ISO-639 language code followed by a two-letter ISO-3166 country code. Add a language by selecting Languages and pressing the Add button. The dialog for adding languages is given below.



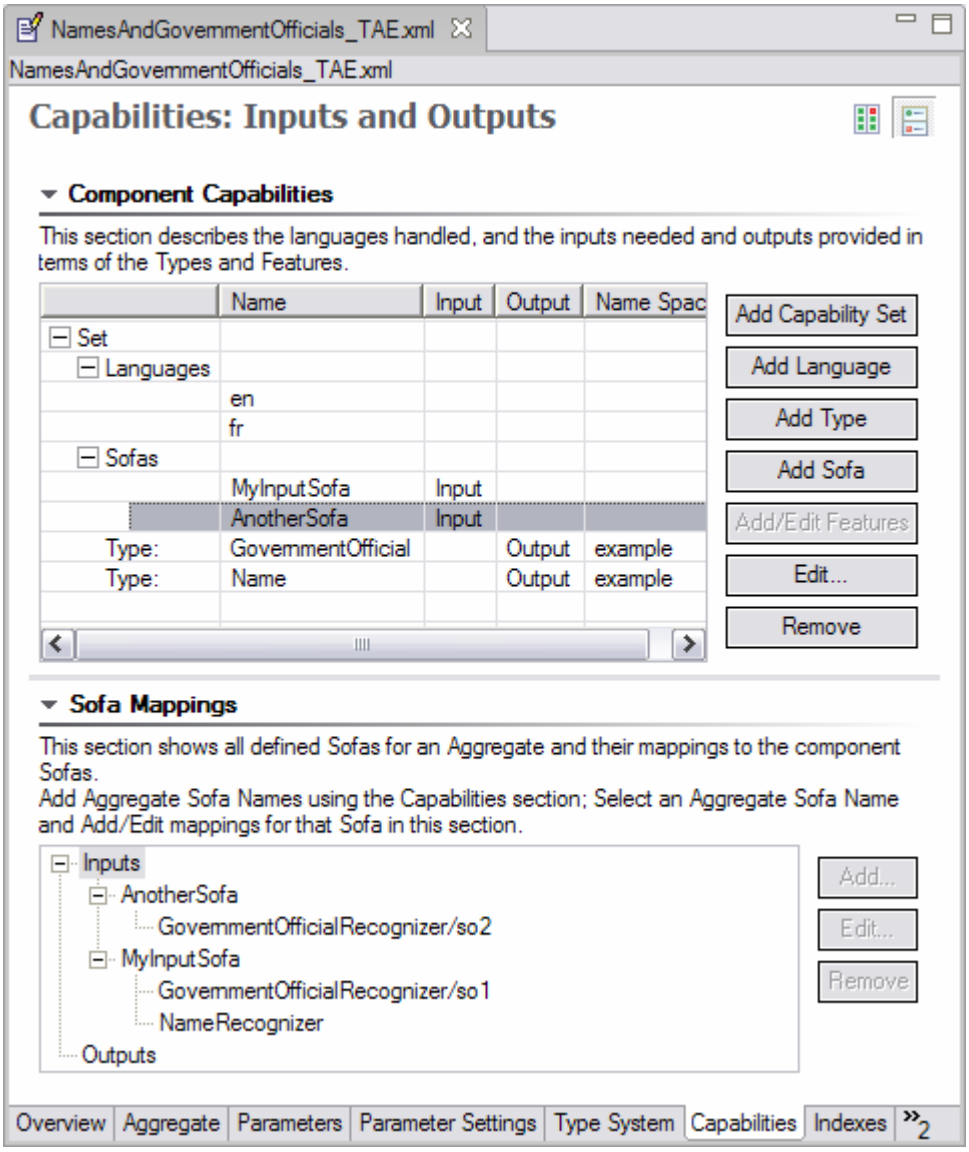
The Sofa part of the capability is optional; it allows defining Sofa names that this component uses, and whether they are input (meaning they are created outside of this component, and passed into it), or output (meaning that they are created by this component). Note that a Sofa can be either input or output, but can't be both.

To add a Sofa name, press the Add Sofa button, and this dialog appears:



9.9.1 Sofa name mappings

Sofa names, once created, are used in Sofa Mappings. These are optional mappings, done in an aggregate, that specify which Sofas are the same ones but with different names. The Sofa Mappings section is minimized unless you are editing an Aggregate descriptor, and have one or more Sofa names defined for the aggregate. In that case, the Sofa Mappings section will look like this:

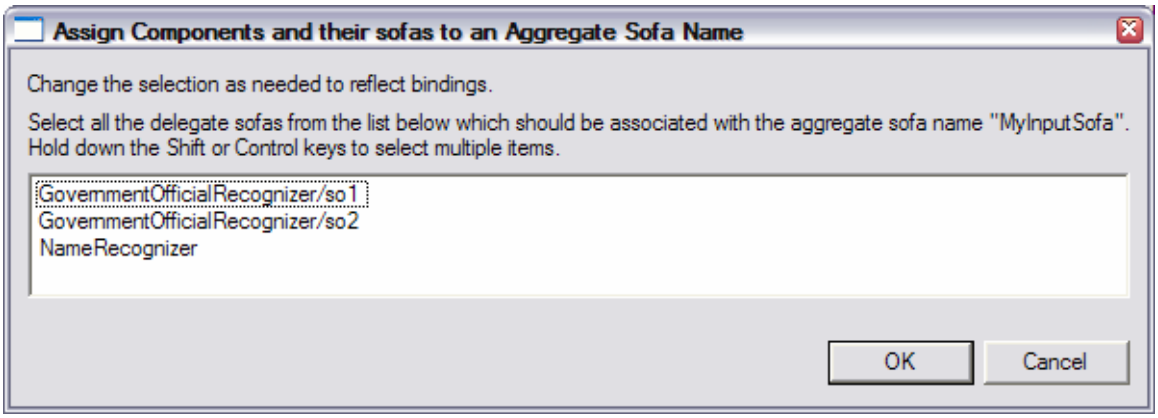


Here the aggregate has defined two input Sofas, named "MyInputSofa", and "AnotherSofa". Any named sofas in the aggregate's capabilities will appear in the Sofa Mapping section, listed either under Inputs or Outputs. Each name in the Mappings has 0 or more delegate's sofa names mapped to it. A delegate may have multiple Sofas, as in this example, where the GovernmentOfficialRecognizer delegate has Sofas named "so1" and "so2".

Delegate components may not be "Sofa aware". In this case, they have one implicit, default Sofa, and to map to it you use the form shown for the "NameRecognizer" – you map to the delegate's key name in the aggregate, without specifying a Sofa name.

To add a new mapping, select the Aggregate Sofa name you wish to add the mapping for, and press the Add button. This brings up a window like this, showing

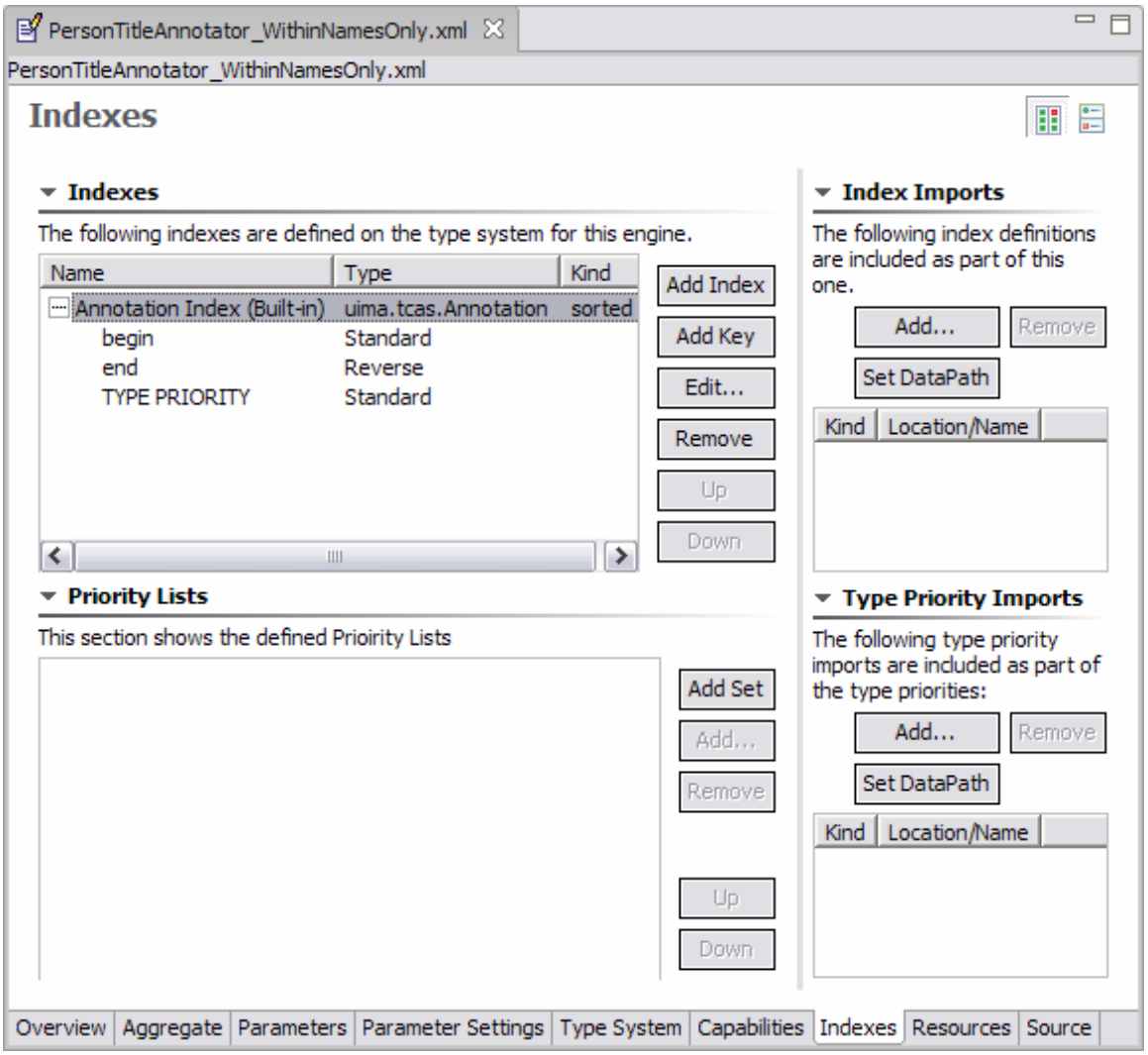
all available delegates and their Sofas; select one or more (use the normal multi-select methods) of these and press OK to add them.



To edit an existing mapping, select the mapping and press Edit. This will show the existing mapping with all mapped items "selected", and other available items unselected. Change the items selected to match what you want, deselecting some, and perhaps selecting others, and press OK.

9.10 Indexes Page

The Indexes page is where the user declares what indexes and type priority lists are used by the analysis engine. Indexes are used to determine the order in which Feature Structures of a particular type are fetched, using an iterator in the UIMA API. An unpopulated Indexes page is displayed below:



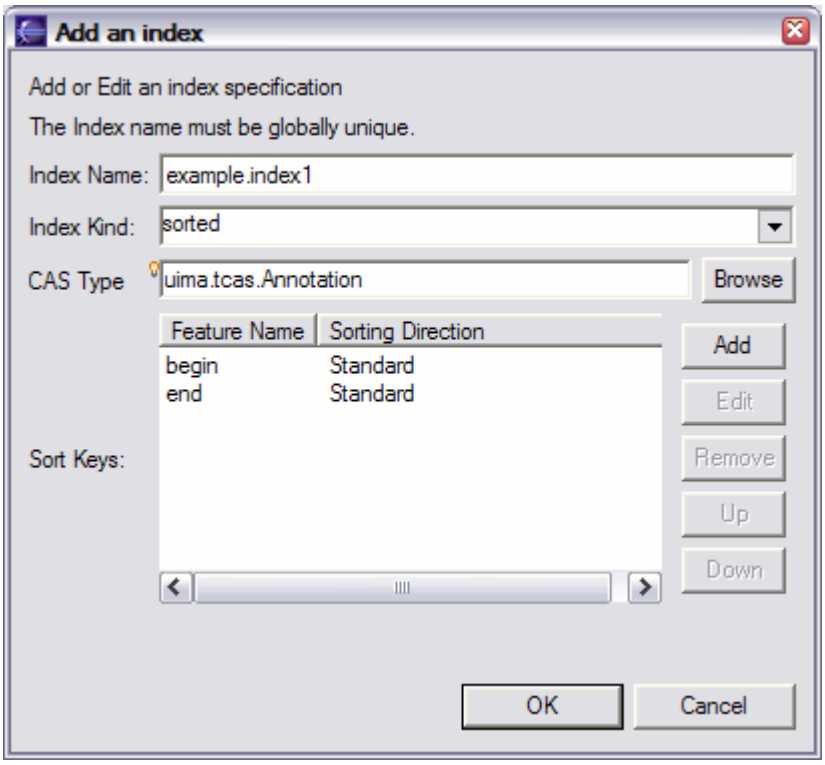
Both indexes and type priority lists can have imports. These imports work just like the type system imports, described above.

The built-in Annotation Index is always present for *TAEs*. It is based on the type `uima.tcas.Annotation` and has keys begin (Ascending), end (Descending) and TYPE_PRIORITY. There are no built-in type priorities, so this last sort item does not play a role in the index unless type priorities are specified.

Type priority may be combined with other keys. Type priorities are defined in the Priority Lists section, using one or more priority list. A given priority list gives an ordering among a group of types. Types that appear higher in the priority list are given higher priority, in other words, they sort first when TYPE_PRIORITY is specified as the index key. Subtypes of these types are also sorted, unless such a sort is overridden by other specific type priority specifications. To get the ordering used among all the types, all of the type priority lists are merged. This gives a partial ordering among the types. Ties are resolved in an unspecified fashion. The

Component Descriptor Editor checks for incompatible orderings, and informs the user if they exist, so they can be corrected.

To create a new index, use the Add Index button in the top left section. This brings up this dialog:



Each index needs a globally unique index name. Every index indexes one CAS type (and its subtypes). The entry field for this has content assist (start typing the type name and press Control – Spacebar to get help, or press the Browse button to pick a type).

Indexes can be sorted, in which case you need to specify one or more keys to sort on. Sort keys are selected from features whose range type is Integer, Float, or String. Some elements will be disabled if they are not relevant. For instance, if the index kind is "bag", you cannot provide sort keys. The order of sort keys can be adjusted using the up and down buttons, if necessary.

A set index will contain no duplicates of the same type, where a duplicate is defined by the indexing comparator. That is, if you commit two feature structures of the same type that are equal with respect to the indexing comparator, only the first one will be entered into the index. Note that you can still have duplicates with respect to the indexing order, if they are of a different type. A set index is not guaranteed to be sorted. If no keys are specified for a set index, then all instances are considered by default to be equal, so only the first instance (for a particular type or subtype of

the type being indexed) is indexed. On the other hand, "bag" indicates that all annotation instances are indexed, including duplicates.

The Priority Lists section of the Indexes page is used to specify Priority Lists of types. Priority Lists are unnamed ordered sets of type names. Add a new priority list by clicking the Add Set button. Add a type to an existing priority list by first selecting the set, and then clicking Add. You can use the up and down buttons to adjust the order as necessary; these buttons move the selected item up or down.

Although it is possible to import self-contained index and type priority files, the creation of such files is not yet supported by the Component Descriptor Editor. If you create these files using another editor, they can be imported using the corresponding Import panels, shown on the right. Imports are specified in the same manner as they are for Type System imports.

9.11 Resources Page

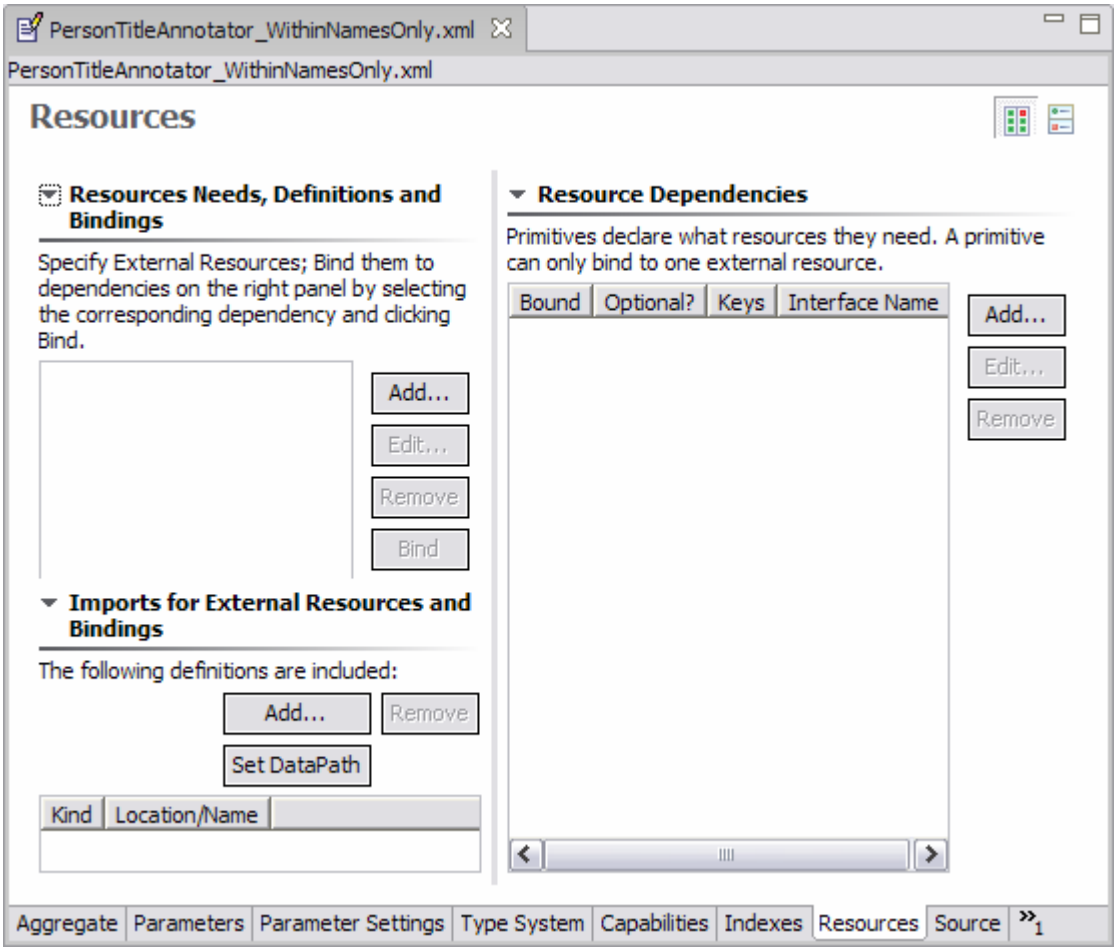
The resources page describes resource dependencies (for primitive Analysis Engines) and external Resource specification and their bindings to the resource dependencies.

Only primitive Analysis Engines define resource dependencies. Primitive and Aggregate Analysis Engines can define external resources and connect them (bind them) to resource dependencies.

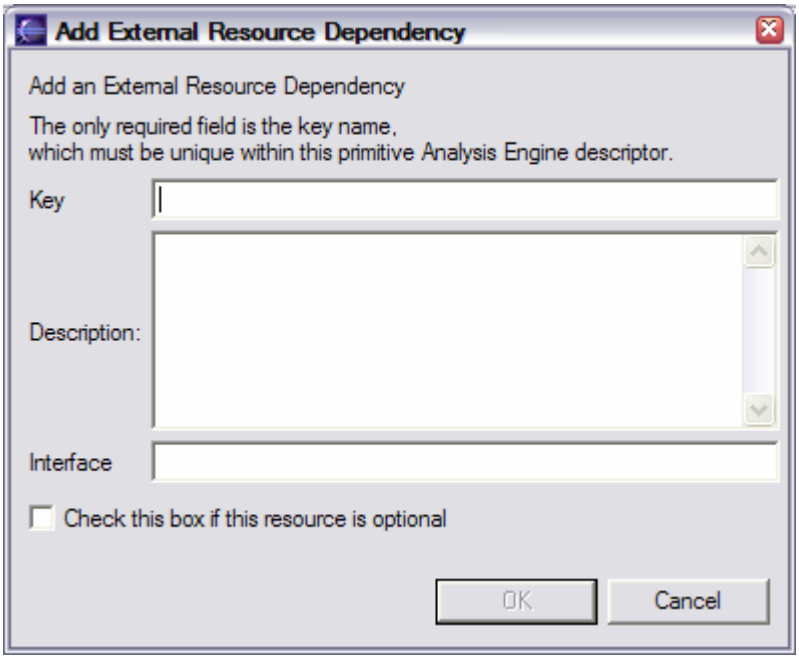
When an Aggregate is providing an external resource to be bound to a dependency, the binding is specified using a possibly multi-level path, starting at the Aggregate, and specify which component (by its key name), and then if that component is, in turn, an Aggregate, which component (again by its key name), and so on until you reach a primitive. The sequence of key names is made into the binding specification by joining the parts with a "/" character. All of this is done for you by the Component Descriptor Editor.

Any external resource provided by an Aggregate will override any binding provided by any lower level component for the same resource dependency.

There are two views of the Resources page, depending on whether the Analysis Engine is an Aggregate or Primitive. Here's the view for a Primitive:

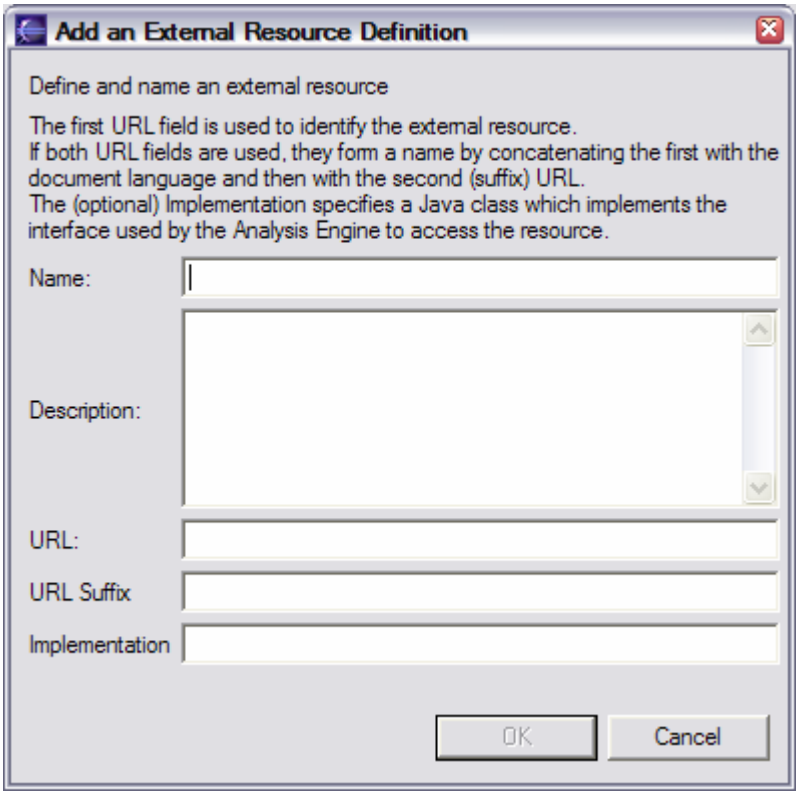


To declare a resource dependency, click the Add button in the right hand panel. This puts up the dialog:



The Key must be unique within the descriptor declaring it. The Interface, if present, is the name of a Java interface the Analysis Engine uses to access the resource.

Declare actual External resource on the left side of the page. Clicking "Add" brings up this dialog:



The Name must be unique within this Analysis Engine. The URL identifies a file resource. If both the URL and URL suffix are used, the file resource is formed by combining the first URL part with the language-identifier, followed by the URL suffix; see ***Resource Manager Configuration*** on page 20-279. URLs may be written as "relative" URLs; in this case they are resolved by looking them up relative to the classpath and/or datapath. A relative URL has the path part starting without an intial "/"; for example: file:my/directory/file. An absolute URL starts with file:/ or file:/// or file://some.network.address/. For more information about URLs, please read the javaDoc information for the Java class "URL".

The Implementation is optional, and if given, must be a Java class that implements the interface specified in any Resource Dependencies this resource is bound to.

9.11.1 Binding

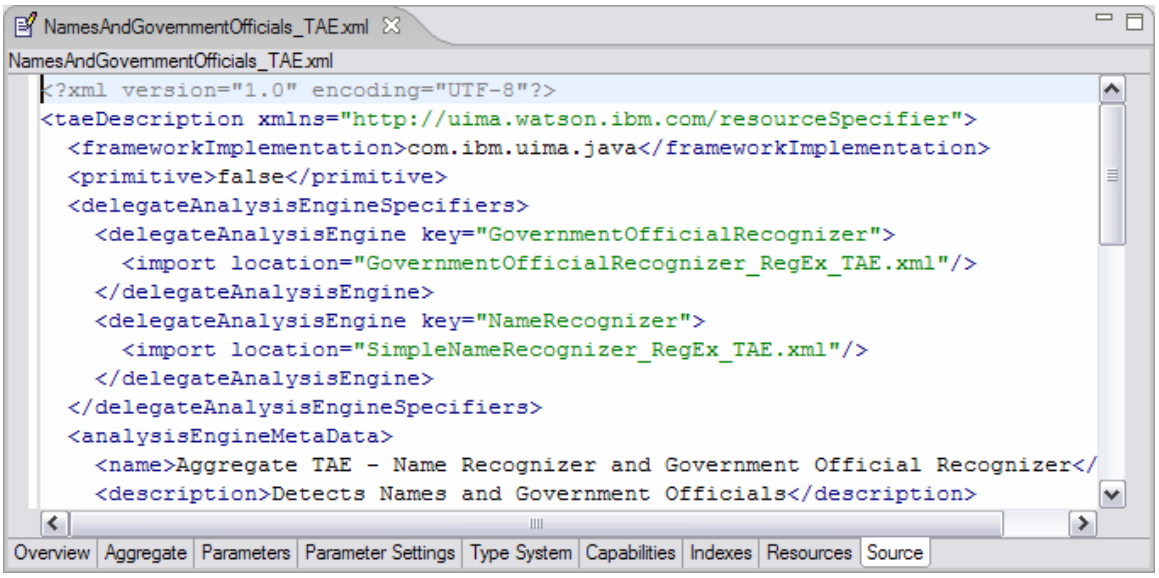
Once you have an external resource definition, and a Resource Dependency, you can bind them together. To do this, you select the two things (an external resource definition, and a Resource Dependency) that you want to bind together, and click Bind.

9.11.2 Resources with Aggregates

When editing an Aggregate Descriptor, the Resource definitions panel will show all the resources at the primitive level, with paths down through the components (multiple levels, if needed) to get to the primitives. The Aggregate can define external resources, and bind them to one or more uses by the primitives.

9.12 Source Page

The Source page is a text view of the xml content of the Analysis Engine or Type System being configured. An example of this page is displayed below:



Changes made in the GUI are immediately reflected in the xml source, and changes made in the xml source are immediately reflected back in the GUI. The thought here is that the GUI view and the Source view are just two ways of looking at the same data. When the data is in an unsaved state the file name is prefaced with an asterisk in the currently selected file tab in the editor pane inside Eclipse (as in the example above).

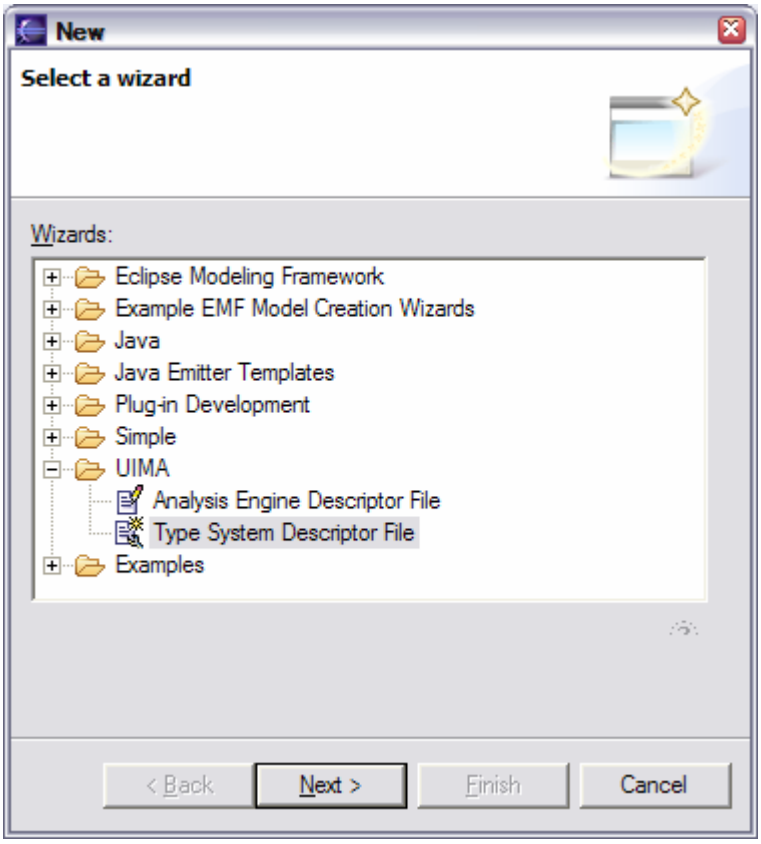
You may accidentally create invalid descriptors or XML by editing directly in the Source view. If you do this, when you try and save or when you switch to a different view, the error will be detected and reported. In the case of saving, the file will be saved, even if it is in an error state.

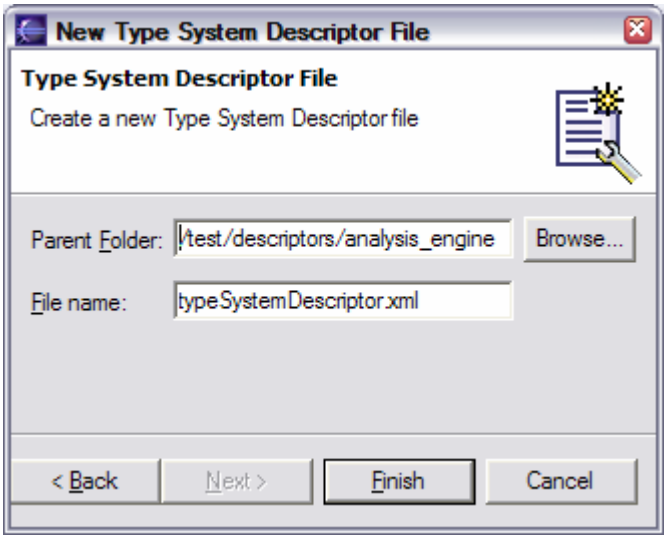
9.12.1 Source formating – indentation

The XML is indented using an indentation amount saved as a global UIMA preference. To change this preference, use the Eclipse menu item: Windows -> Preferences -> UIMA Preferences.

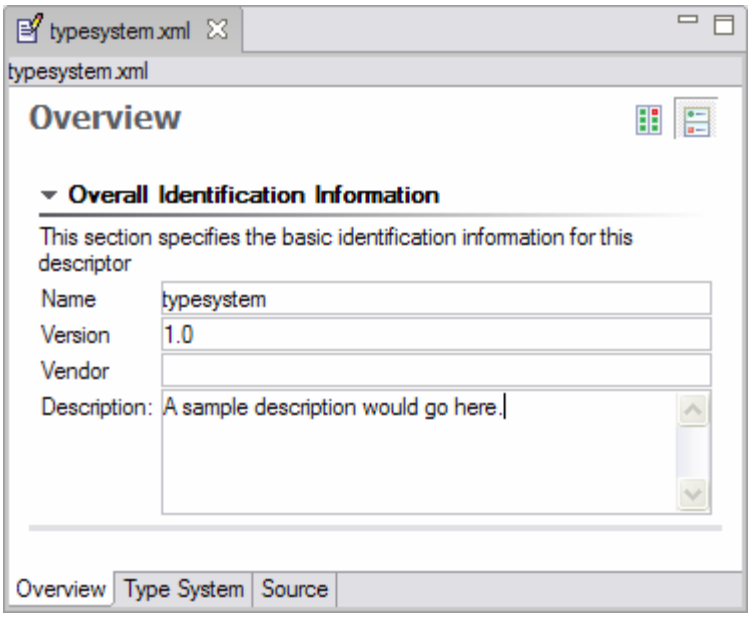
9.13 Creating a Self-Contained Type System

It is also possible to use the Component Descriptor Editor to create or edit self-contained type systems. To create a self-contained type system, select the menu item File->New->Other and then select Type System Descriptor File. From the next page of the selection wizard specify a Parent Folder and File name and click Finish.





This will take you to a stripped down interface for editing a type system file which contains just three pages: an overview page, a type system page, and a source page. The overview page is a bit more Spartan than in the case of an AE. It looks like the following:



Just like an AE has an associated name, version, vendor and description, the same is true of a self-contained type system. The Type System page is identical to that in an AE descriptor file, as is the Source page. It is worthy of note that a self-contained type system can import type systems just like the type system associated with an AE.

9.14 Creating Other Descriptor Components

The new wizard can create two other kinds of components: Collection Processing Management (CPM) components, and importable parts (besides Type Systems, described above, there is some limited editing support for Indexes, Type Priorities, and Resource Manager Configuration imports).

The CPM components supported by this editor include the Collection Reader, Cas Initializer, and CasConsumer descriptors. Each of these is basically treated just like a primitive AE descriptor, with small changes to accommodate the different semantics. For instance, a CasConsumer can't declare in its capabilities section that it outputs types or features.

The importable part support is limited because much of the power of this editor comes from extensive checking that requires additional information, other than what is available in just the importable part. For instance, although you can create an Indexes import, the facility for adding new indexes doesn't work, because it needs the type information, which is not present in this part when it is edited alone. However, you can use the source editor, and some kinds of editing (which can be done in isolation) are supported, such as removing an index, or reordering them.

Likewise, the Resource Manager Configuration editing capability can't add any bindings because these require information outside of this descriptor (the Resource Dependencies).

Chapter 10 Collection Processing Engine Configurator User's Guide

A *Collection Processing Engine (CPE)* processes collections of artifacts (documents) through the combination of the following components: a Collection Reader, an optional CAS Initializer, Analysis Engines, and CAS Consumers.

The *Collection Processing Engine Configurator(CPE Configurator)* is a graphical tool that allows you to assemble and run CPEs.

For an introduction to Collection Processing Engine concepts, including developing the components that make up a CPE, read *Chapter 5* ***Collection Processing Engine Developer's Guide***. This chapter is a user's guide for using the CPE Configurator tool, and does not describe UIMA's Collection Processing Architecture itself.

10.1 Limitations of the CPE Configurator

The CPE Configurator only supports basic CPE configurations.

It only supports "Integrated" deployments (although it will connect to remotes if particular CAS Processors are specified with remote service descriptors). It doesn't support configuration of the error handling. It doesn't support Sofa Mappings; it assumes all Sofa-unaware components are operating with a default text Sofa. Sofa-aware components will not have their names mapped. It sets up a fixed-sized CAS Pool.

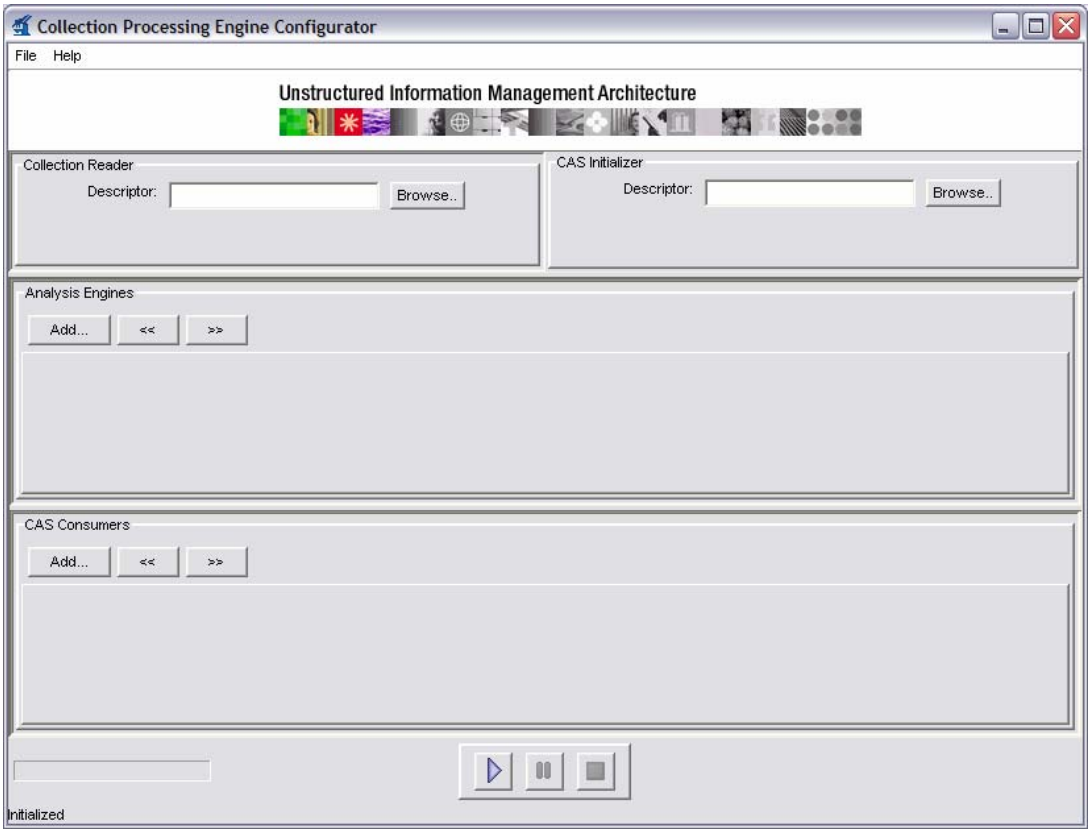
For running arbitrary CPE descriptors, or running with other than the default configuration supplied by the CPE Configurator, you can write your own application, or use the runCPE script, which invokes an example application, SimpleRunCPE.

10.2 Starting the CPE Configurator

The CPE Configurator tool can be run using the `cpeGui` shell script, which is located in the `bin` directory of the UIMA SDK. If you've installed the example Eclipse project (see *Chapter 3* ***UIMA SDK Setup for Eclipse***), you can also run it using the "UIMA CPE GUI" run configuration provided in that project.

Note that if you are planning to build a CPE using components other than the examples included in the UIMA SDK, you will first need to update your CLASSPATH environment variable to include the classes needed by these components.

When you first start the CPE Configurator, you will see the main window shown here:



10.3 Selecting Component Descriptors

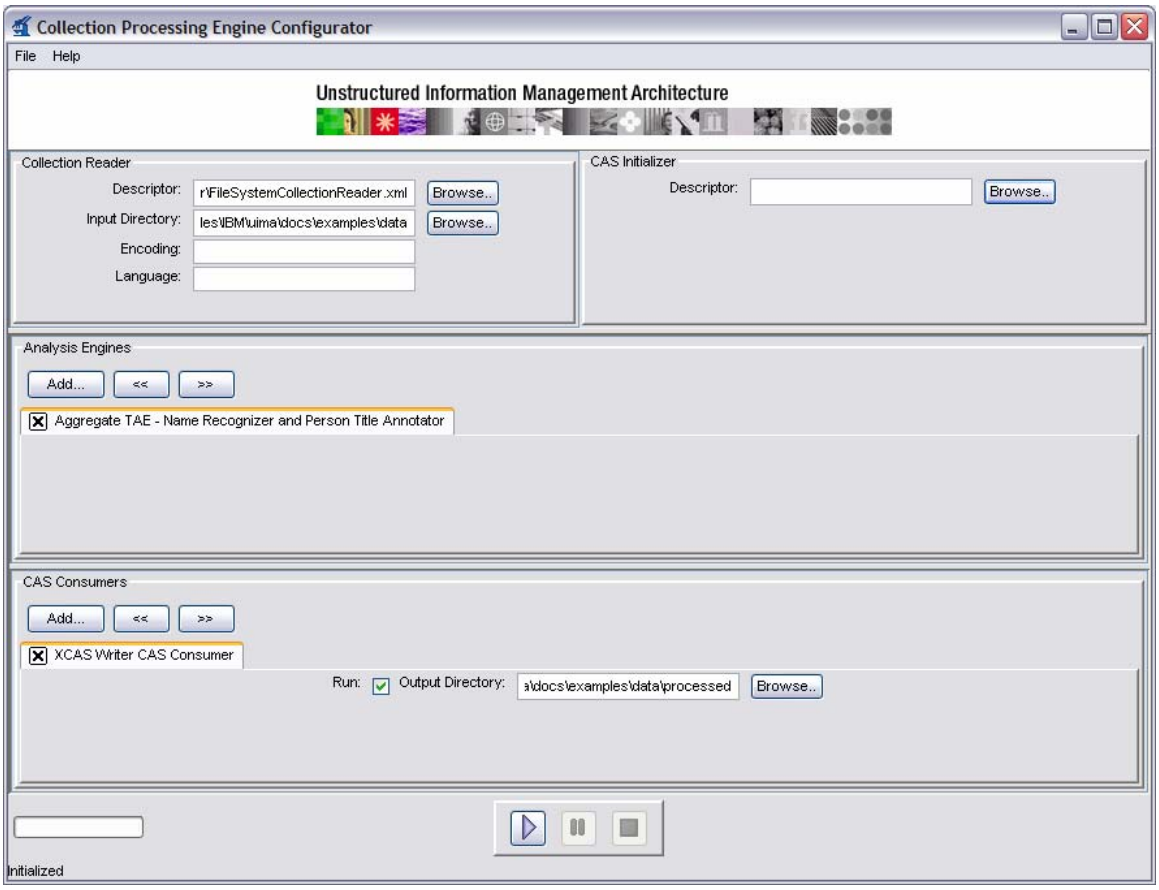
The CPE Configurator's main window is divided into 4 sections: one for each of the types of components that constitute a CPE: CollectionReader, CAS Initializer, Analysis Engines, and CasConsumers. Each CPE has exactly one CollectionReader, an optional CAS Initializer, and at least one each of Analysis Engines and CAS Consumers.

In each section of the CPE Configurator, you can select the component(s) you want to use by browsing to (or typing the location of) their XML descriptors. You must select a Collection Reader, at least one Analysis Engine, and at least one CAS Consumer. You may or may not need to select a CAS Initializer; this depends on the particular Collection Reader that you are using.

When you select a descriptor, the configuration parameters that are defined in that descriptor will then be displayed in the GUI; these can be modified to override the values present in the descriptor.

For example, the screen shot below shows the CPE Configurator after the following components have been chosen:

```
docs/examples/descriptors/collectionReader/FileSystemCollectionReader.xml
docs/examples/descriptors/analysis_engine/NamesAndPersonTitles_TAE.xml
docs/examples/descriptors/cas_consumer/XCasWriterCasConsumer.xml
```



10.4 Running a Collection Processing Engine

After selecting each of the components and providing configuration settings, click the play (forward arrow) button at the bottom of the screen to begin processing. A progress bar should be displayed in the lower left corner. (Note that the progress bar will not begin to move until all components have completed their initialization, which may take several seconds.) Once processing has begun, the pause and stop buttons become enabled.

If an error occurs, you will be informed by an error dialog. If processing completes successfully, you will be presented with a performance report.

10.5 The File Menu

The CPE Configurator's File Menu has six options:

- Open CPE Descriptor
- Save CPE Descriptor
- Refresh Descriptors from File System
- Clear All
- Exit

Open CPE Descriptor will allow you to select a CPE Descriptor file from disk, and will read in that CPE Descriptor and configure the GUI appropriately.

Save CPE Descriptor will create a CPE Descriptor file that defines the CPE you have constructed. This CPE Descriptor will identify the components that constitute the CPE, as well as the configuration settings you have specified for each of these components. Later, you can use "Open CPE Descriptor" to restore the CPE Configurator to the state. Also, CPE Descriptors can be used to easily run a CPE from a Java program – see *Chapter 6* ***Application Developer's Guide***.

CPE Descriptors also allow specifying operational parameters, such as error handling options that are not currently available for configuration through the CPE Configurator. For more information on manually creating a CPE Descriptor, see *Chapter 21* ***Collection Processing Engine Descriptor Reference***.

Refresh Descriptors from File System will reload all descriptors from disk. This is useful if you have made a change to the descriptor outside of the CPE Configurator, and want to refresh the display.

Clear All will reset the CPE Configurator to its initial state, with no components selected.

Exit will close the CPE Configurator. If you have unsaved changes, you will be prompted as to whether you would like to save them to a CPE Descriptor file. If you do not save them, they will be lost.

When you restart the CPE Configurator, it will automatically reload the last CPE descriptor file that you were working with.

10.6 The Help Menu

The CPE Configurator's Help menu provides "About" information and some very simple instructions on how to use the tool.

Chapter 11 PEAR Packager User's Guide

A PEAR (Processing Engine ARchive) file is a standard package for UIMA (Unstructured Information Management Architecture) components. The PEAR package can be used for distribution and reuse by other components or applications. It also allows applications and tools to manage UIMA components automatically for verification, deployment, invocation, testing, etc. Please refer to the PEAR Reference chapter for more information about the internal structure of a PEAR file.

This chapter describes how to use the PEAR Eclipse Plugin to create PEAR files for standard UIMA components. This plugin is installed if you followed the directions in Chapter 3 ***UIMA SDK Setup for Eclipse*** on page 3-43.

11.1 Using the PEAR Eclipse Plugin

Using the PEAR Eclipse Plugin involves the following two steps:

- Add the UIMA nature to your project
- Create a PEAR file using the PEAR generation wizard

11.1.1 Add UIMA Nature to your project

First, create a project for your UIMA component:

- Create a Java project, which would contain all the files and folders needed for your UIMA component.
- Create a source folder called "src" in your project, and make it the only source folder, by clicking on "Properties" in your project's context menu (right-click), then select "Java Build Path", then add the "src" folder to the source folders list, and remove any other folder from the list.
- Specify an output folder for your project called bin, by clicking on "Properties" in your project's context menu (right-click), then select "Java Build Path", and specify "*your_project_name*/bin" as the default output folder.

Then, add the UIMA nature to your project by clicking on "Add UIMA Nature" in the context menu (right-click) of your project. Click "Yes" on the "Adding UIMA custom Nature" dialog box. Click "OK" on the confirmation dialog box.

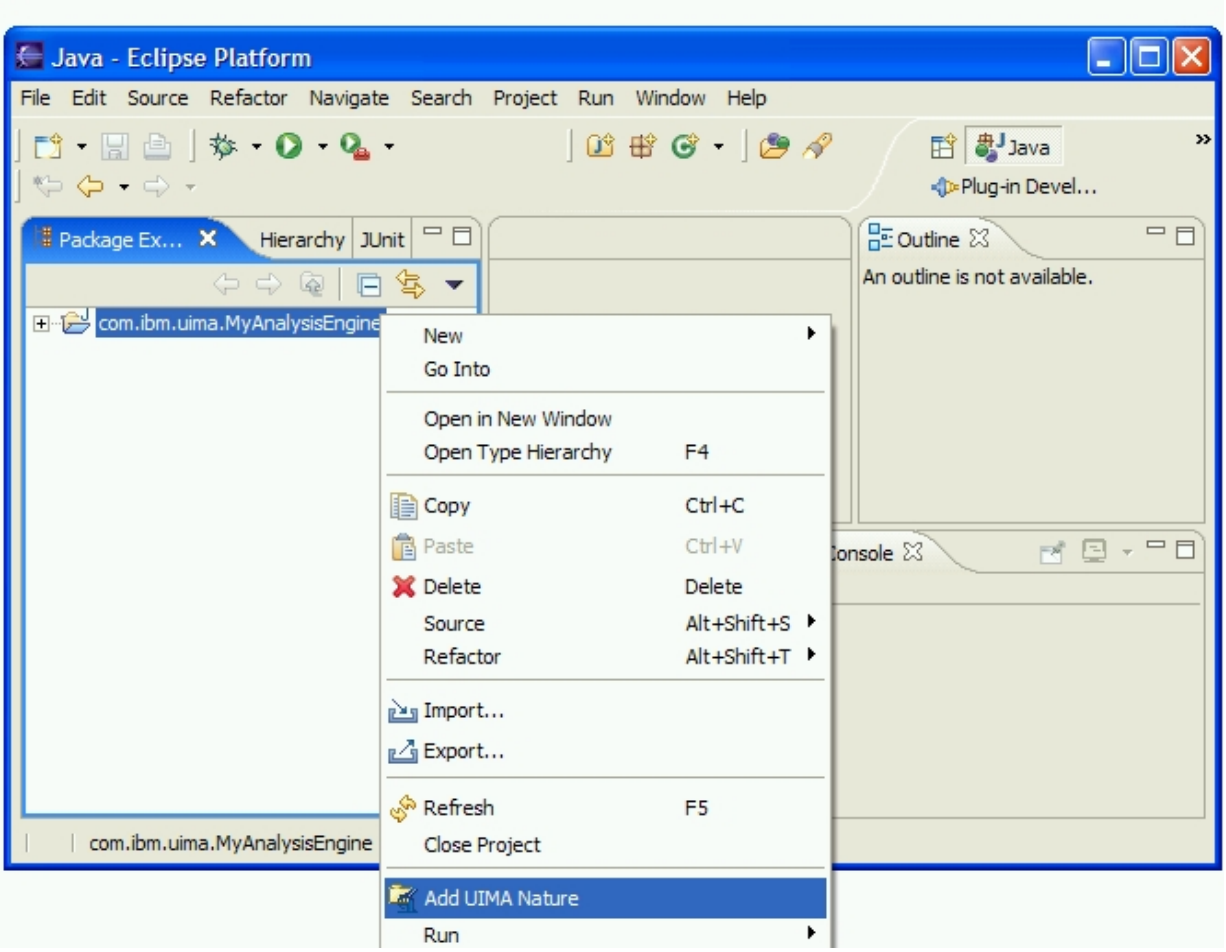


Figure 17. Adding the UIMA Nature

Adding the UIMA nature to your project creates the PEAR structure in your project. The PEAR structure is a structured tree of folders and files, including the following elements:

- **Required Elements:**
  - The **metadata** folder which contains the PEAR installation descriptor and properties files.
  - The installation descriptor (**metadata/install.xml**)
- **Optional Elements:**
  - The **desc** folder to contain descriptor files of analysis engines, component analysis engines (all levels), and other component (Collection Readers, CAS Consumers, etc).
  - The **src** folder to contain the source code
  - The **bin** folder to contain executables, scripts, class files, dlls, shared libraries, etc.
  - The **lib** folder to contain jar files.

- The **doc** folder containing documentation materials, preferably accessible through an index.html.
- The **data** folder to contain data files (e.g. for testing).
- The **conf** folder to contain configuration files.
- The **resources** folder to contain other resources and dependencies.
- Other user-defined folders or files are allowed, but should be avoided.

For more information about the PEAR structure, please refer to the "Processing Engine Archive" section.

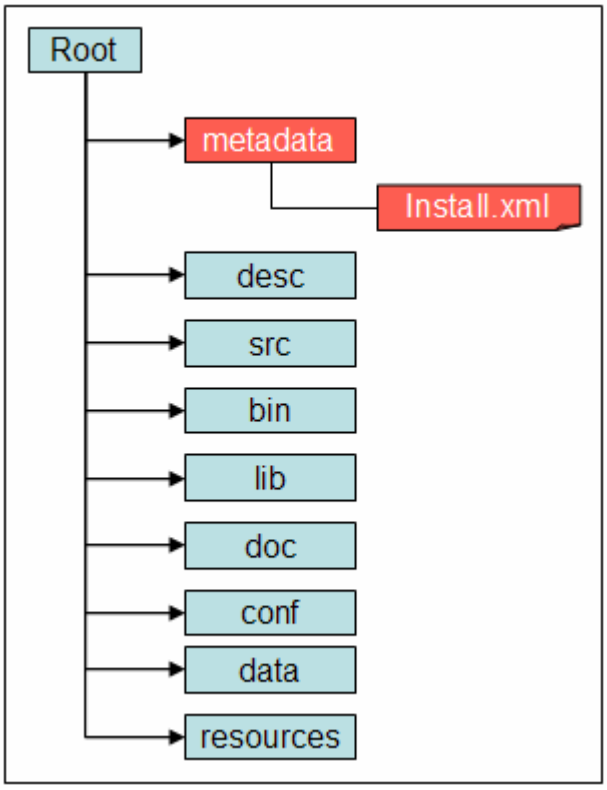


Figure 18. The PEAR Structure

11.1.2 Use the PEAR Generation Wizard

Before using the PEAR Generation Wizard, make sure you add all the files needed to run your component including descriptors, jars, external libraries, resources, and component analysis engines (in the case of an aggregate analysis engine), etc. It's recommended to generate a jar file from your code as an alternative to building the project and making sure the output folder (bin) contains the required class files.

Then, click on "Generate PEAR file" from the context menu (right-click) of your project, to open the PEAR Generation wizard, and follow the instructions on the wizard to generate the PEAR file.

The Component Information page

The first page of the PEAR generation wizard is the component information page. Specify in this page a component ID for your PEAR and select the main Analysis Engine descriptor. The descriptor must be specified using a pathname relative to the project's root (e.g. "desc/MyTAE.xml). The component id is a string that uniquely identifies the component. It should use the JAVA naming convention (e.g. com.ibm.uima.mycomponent).

Optionally, you can include specific Collection Iterator, CAS Initializer, or CAS Consumers. In this case, specify the corresponding descriptors in this page.

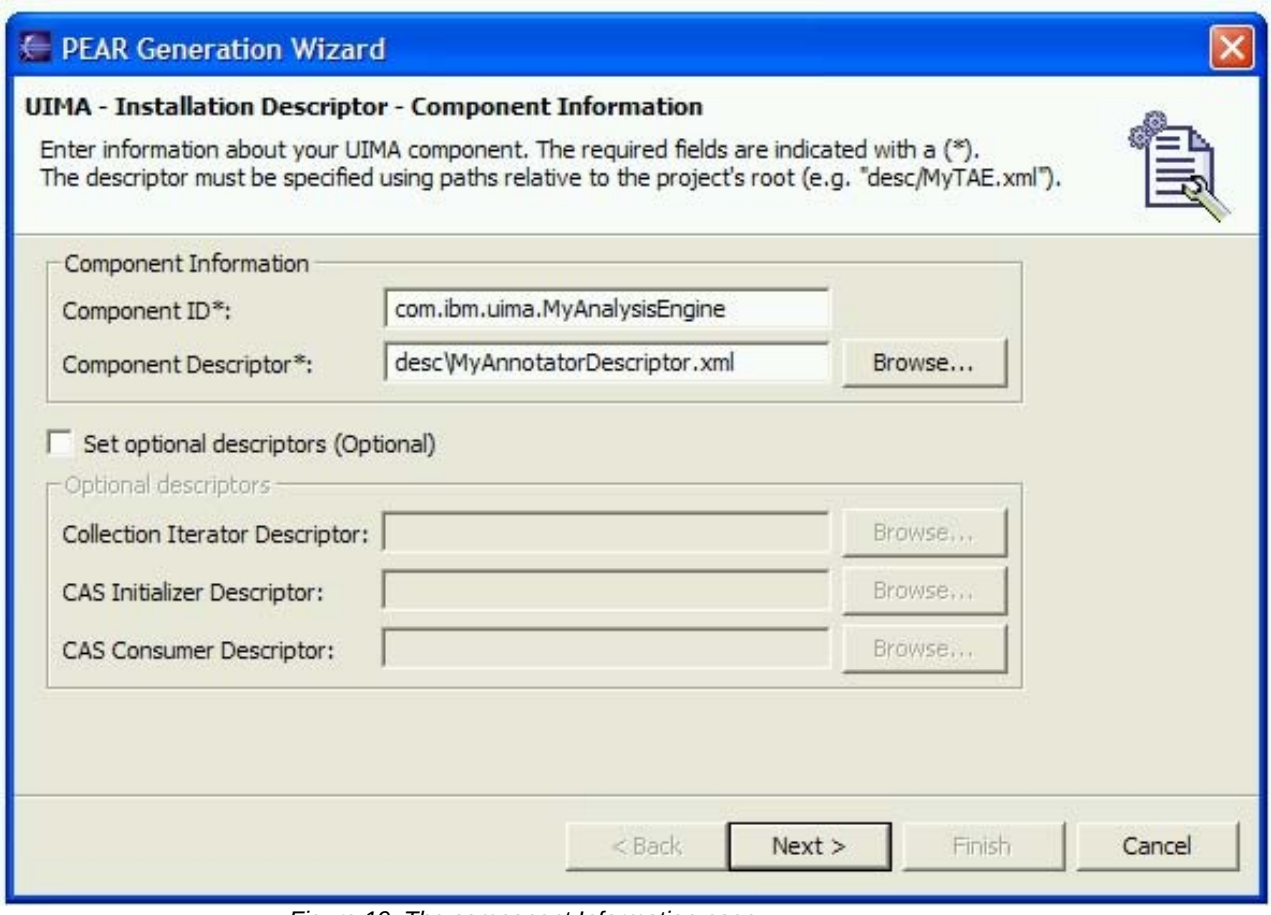


Figure 19. The component Information page

The Installation Environment page

The installation environment page is used to specify the following:

- Preferred operating system
- Required JDK version, if applicable.
- Required Environment variable, such as CLASSPATH

Path names should be specified using macros (see below), instead of hard-coded absolute paths that might work locally, but probably won't if the PEAR is deployed in a different machine and environment.

Macros are variables such as $main_root, used to represent a string such as the full path of a certain directory.

These macros should be defined in the PEAR.properties file using the local values. The tools and applications that use and deploy PEAR files should replace these macros (in the files included in the conf and desc folders) with the corresponding values in the local environment as part of the deployment process.

Currently, there are two types of macros:

- $main_root , which represents the local absolute path of the main component root directory after deployment.
- $*component_id*$root, which represents the local absolute path to the root directory of the component which has *component_id* as component ID. This component could be, for instance, a delegate component.

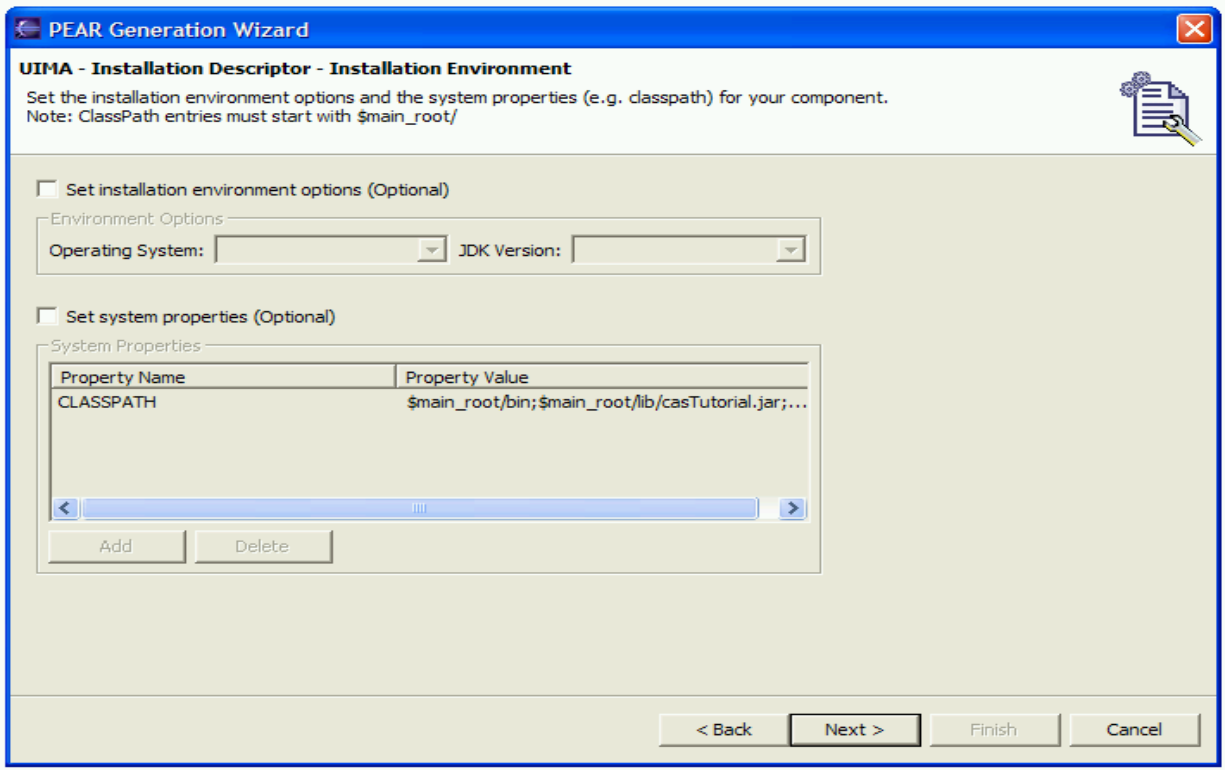


Figure 20. The Installation Environment Page

The PEAR file content page

The last page of the wizard is the "PEAR file Export" page, which allows the user to select the files to include in the PEAR file. The metadata folder and all its content is mandatory. Make sure you include all the files needed to run your component including descriptors, jars, external libraries, resources, and component analysis engines (in the case of an aggregate analysis engine), etc. It's recommended to generate a jar file from your code as an alternative to building the project and making sure the output folder (bin) contains the required class files.

Note: If you are relying on the class files generated in the output folder (usually called bin) to run your code, then make sure the project is built properly, and all the required class files are generated without errors. In this case make sure your output folder (e.g. $main_root/bin) is in the classpath (see the "Installation Environment" page.

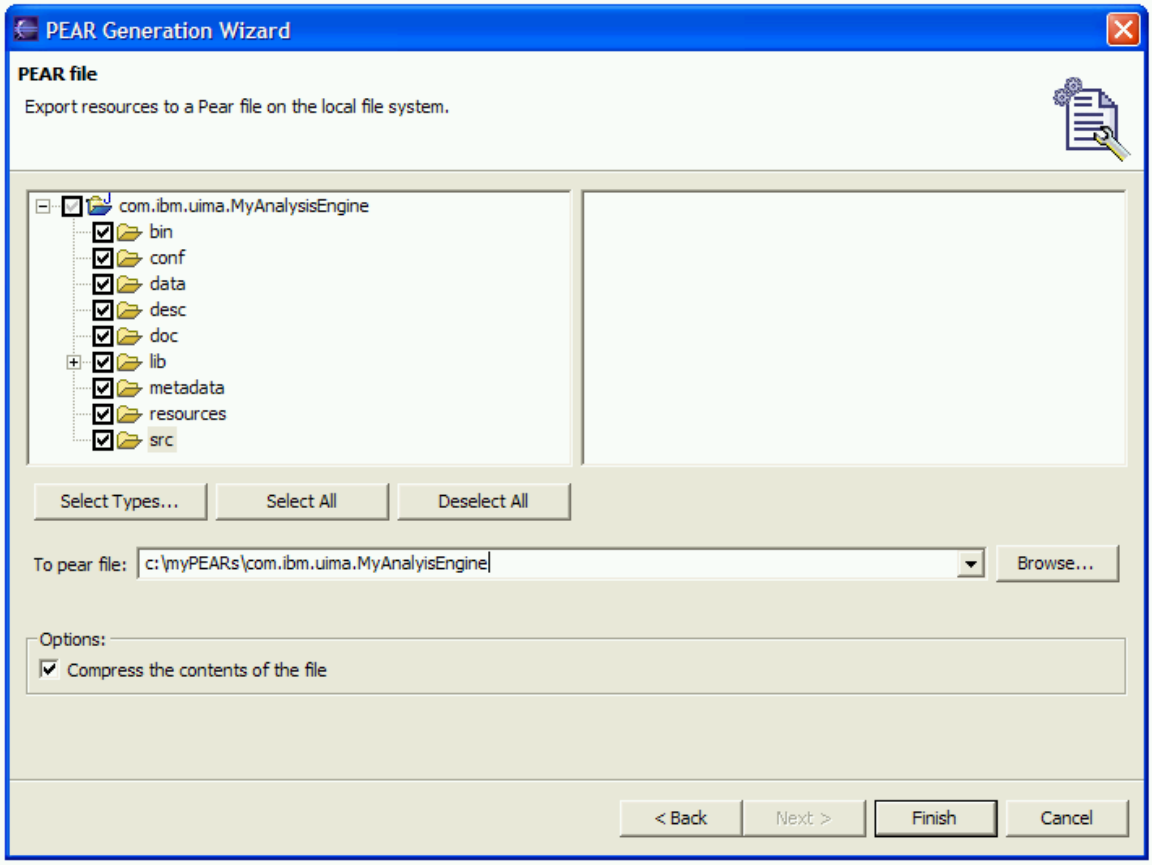
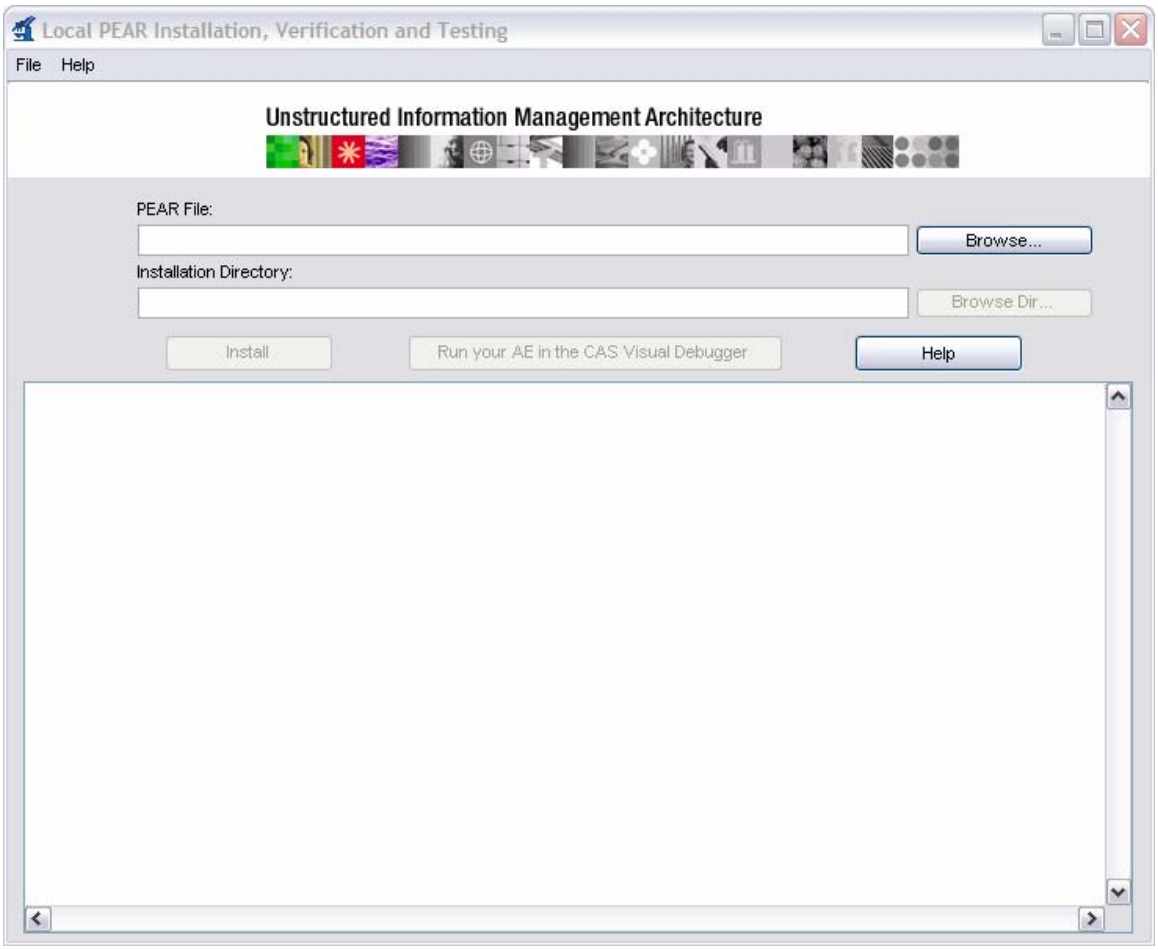


Figure 21. The PEAR File Export Page

Chapter 12 PEAR Installer User's Guide

PEAR (Processing Engine ARchive) is a new standard for packaging UIMA compliant components. This standard defines several service elements that should be included in the archive package to enable automated installation of the encapsulated UIMA component. The major PEAR service element is an XML Installation Descriptor that specifies installation platform, component attributes, custom installation procedures and environment variables.

The installation of a UIMA compliant component includes 2 steps: (1) installation of the component code and resources in a local file system, and (2) verification of the serviceability of the installed component. Installation of the component code and resources involves extracting component files from the archive (PEAR) package in a designated directory and localizing file references in component descriptors and other configuration files. Verification of the component serviceability is accomplished with the help of standard UIMA mechanisms for instantiating analysis engines.



PEAR Installer is a simple GUI based Java application that helps installing UIMA compliant components (analysis engines) from PEAR packages in a local file system. To install a desired UIMA component the user needs to select the appropriate PEAR file in a local file system and specify the installation directory (optional). During the component installation the user can read messages printed by the installation program in the message area of the application window. If the installation fails, appropriate error message is printed to help identifying and fixing the problem.

After the desired UIMA component is successfully installed, the PEAR Installer allows testing this component in the CAS Visual Debugger (CVD) application, which is provided with the UIMA package. The CVD application will load your UIMA component using its XML descriptor file. If the component is loaded successfully, you'll be able to run it either with sample documents provided in the `<UIMA_HOME>/docs/examples/data` directory, or with any other sample documents. See `CASVisualDebugger.pdf` in the `docs` directory for more information about the CVD application. Running your component in the CVD application helps to make sure the component will run in other UIMA applications. If the CVD application fails to load or run your component, or throws an exception, you can find more information about the problem in the uima.log file in the current working directory. The log file can be viewed with the CVD.

PEAR Installer creates the `setenv.txt` file in the `<component_root>/metadata` directory. This file contains environment variables required to run your component in any UIMA application. For instance, if you want to run your component in the Collection Processing Engine Configurator GUI application, you need to add the environment variables settings from the component's `setenv.txt` file to the `cpeGui.bat (cpeGui.sh)` script file in the `<UIMA_HOME>/bin` directory.

Chapter 13 PEAR Merger User's Guide

The PEAR Merger utility takes two or more PEAR files and merges their contents, creating a new PEAR which has, in turn, a new Aggregate analysis engine whose delegates are the components from the original files being merged. It does this by (1) copying the contents of the input components into the output component, placing each component into a separate subdirectory, (2) generating a UIMA descriptor for the output Aggregate text analysis engine and (3) creating an output PEAR file that encapsulates the output Aggregate.

The merge logic is quite simple, and is intended to work for simple cases. More complex merging needs to be done by hand. Please see the Restrictions and Limitations section, below.

This is a command-line utility; there are shell scripts (.bat for Windows, and .sh for Unix) to run it.

```
runPearMerger 1st_input_pear_file ... nth_input_pear_file
              -n output_analysis_engine_name [ -f output_pear_file ]
```

The first group of parameters are the input PEAR files. No duplicates are allowed here. The `-n` parameter is the name of the generated Aggregate Analysis Engine. The optional `-f` parameter specifies the name of the output file. If it is omitted, the output is written to `output_tae_name.pear` in the current working directory.

During the running of this tool, work files are written to a temporary directory created in the user's home directory.

13.1 Details of the merging process

The PEARs are merged using the following steps:

1. A temporary working directory, is created for the output aggregate component.
2. Each input PEAR file is extracted into a separate 'input_component_name' folder under the working directory.
3. The extracted files are processed to adjust the '$main_root' macros. This operation differs from the PEAR installation operation, because it does not replace the macros with absolute paths.
4. The output PEAR directory structure, 'metadata' and 'desc' folders under the working directory, are created.

5. The UIMA TAE descriptor for the output aggregate component is built in the 'desc' folder. This aggregate descriptor refers to the input delegate components, specifying 'fixed flow' based on the original order of the input components in the command line. The aggregate descriptor's 'capabilities' and 'operational properties' sections are built based on the input components' specifications.
6. A new PEAR installation descriptor is created in the 'metadata' folder, referencing the new output aggregate descriptor built in the previous step.
7. The content of the temporary output working directory is zipped to created the output PEAR, and then the temporary working directory is deleted.

The PEAR merger utility logs all the operations both to standard console output and to a log file, pm.log, which is created in the current working directory.

13.2 Testing and Modifying the resulting PEAR

The output PEAR file can be installed and tested using the PEAR Installer. The output aggregate component can also be tested by using the CVD or DocAnalyzer tools.

The PEAR Installer creates Eclipse project files (.classpath and .project) in the root directory of the installer PEAR, so the installed component can be imported into the Eclipse IDE as an external project. Once the component is in the Eclipse IDE, developers may use the Component Descriptor Editor and the PEAR Packager to modify the output aggregate descriptor and re-package the component.

13.3 Restrictions and Limitations

The PEAR Merger utility only does basic merging operations, and is limited as follows. You can overcome these by editing the resulting PEAR file or the resulting Aggregate Descriptor.

1. The Merge operation specifies Fixed Flow sequencing for the Aggregate.
2. The merged aggregate does not define any parameters, so the delegate parameters cannot be overridden.
3. No External Resource definitions are generated for the aggregate.
4. No Sofa Mappings are generated for the aggregate.
5. Name collisions are not checked for. Possible name collisions could occur in the fully-qualified class names of the implementing Java classes, the names of JAR files, the names of descriptor files, and the names of resource bindings or resource file paths.

6. The input and output capabilities are generated based on merging the capabilities from the components (removing duplicates). Capability sets are ignored - only the first of the set is used in this process, and only one set is created for the generated Aggregate. There is no support for merging Sofa specifications.
7. No Indexes or Type Priorities are created for the generated Aggregate. No checking is done to see if the Indexes or Type Priorities of the components conflict or are inconsistent.
8. You can only merge Analysis Engines and CAS Consumers.
9. Although PEAR file installation descriptors that are being merged can have specific XML elements describing Collection Reader and CAS Consumer descriptors, these elements are ignored during the merge, in the sense that the installation descriptor that is created by the merge does not set these elements. The merge process does not use these elements; the output PEAR's new aggregate only references the merged components' main PEAR descriptor element, as identified by the PEAR element:
```
<SUBMITTED_COMPONENT>
  <DESC>the_component.xml</DESC>...
</SUBMITTED_COMPONENT>.
```

Chapter 14 Document Analyzer User's Guide

The *Document Analyz*er is a tool provided by the UIMA SDK for testing annotators and TAEs. It reads text files from your disk, processes them using a TAE, and allows you to view the results. The Document Analyzer is designed to work with text files and cannot be used with Analysis Engines that process other types of data.

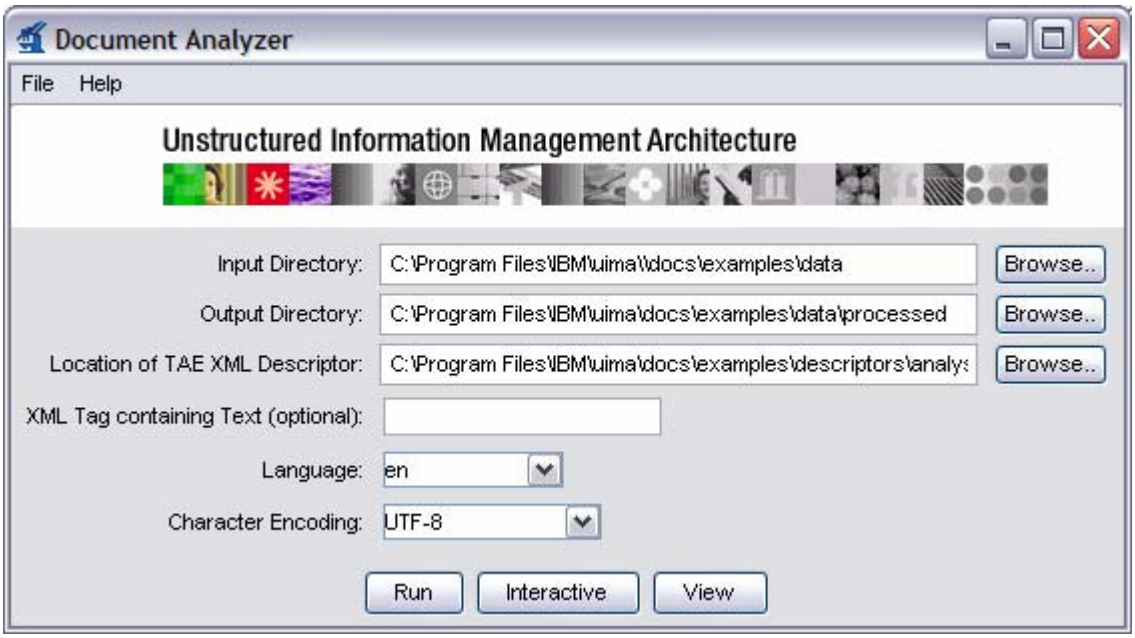
For an introduction to developing annotators and Analysis Engines, read *Chapter 4 **Annotator and Analysis Engine Developer's Guide***. This chapter is a user's guide for using the Document Analyzer tool, and does not describe the process of developing annotators and Analysis Engines.

14.1 Starting the Document Analyzer

To run the Document Analyzer, execute the `documentAnalyzer` script that is in the `bin` directory of your UIMA SDK installation, or, if you are using the example Eclipse project, execute the "UIMA Document Analyzer" run configuration supplied with that project.

Note that if you're planning to run an Analysis Engine other than one of the examples included in the UIMA SDK, you'll first need to update your CLASSPATH environment variable to include the classes needed by that Analysis Engine.

When you first run the Document Analyzer, you should see a screen that looks like this:



14.2 Running a TAE

To run a TAE, you must first configure the six fields on the main screen of the Document Analyzer.

Input Directory: Browse to or type the path of a directory containing text files that you want to analyze. Some sample documents are provided in the UIMA SDK under the `docs/examples/data` directory.

Output Directory: Browse to or type the path of a directory where you want output to be written. (As we'll see later, you won't normally need to look directly at these files, but the Document Analyzer needs to know where to write them.) The files written to this directory will be an XML representation of the analyzed documents. If this directory doesn't exist, it will be created. If you leave this field blank, your TAE will be run but no output will be generated.

Location of TAE XML Descriptor: Browse to or type the path of the descriptor for the TAE that you want to run. There are some example descriptors provided in the UIMA SDK under the `docs/examples/descriptors/analysis_engine` and `docs/examples/descriptors/tutorial` directories.

XML Tag containing Text: This is an optional feature. If you enter a value here, it specifies the name of an XML tag, expected to be found within the input documents, that contains the text to be analyzed. For example, the value `TEXT` would cause the TAE to only analyze the portion of the document enclosed within <TEXT>...</TEXT> tags.

Language: Specify the language in which the documents are written. Some Analysis Engines, but not all, require that this be set correctly in order to do their analysis. You can select a value from the drop-down list or type your own. The value entered here must be an ISO language identifier, the list of which can be found here: http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt

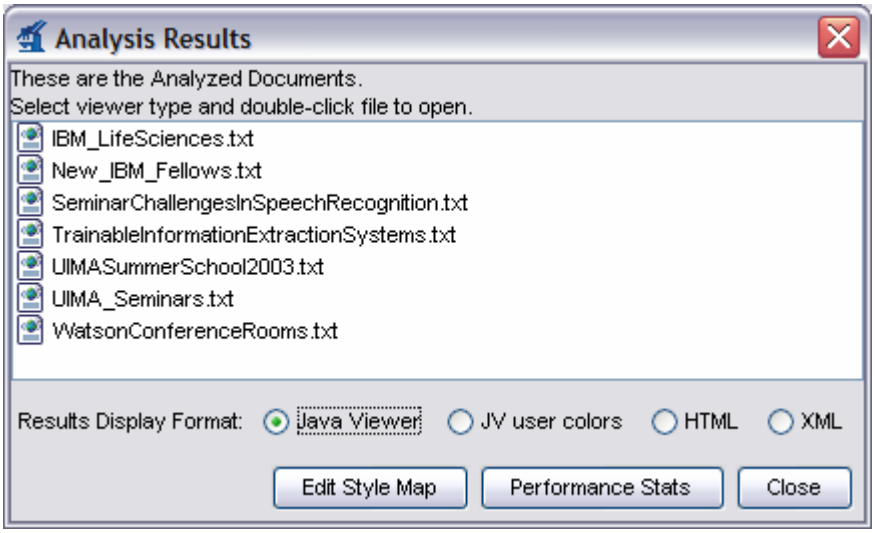
Character Encoding: The character encoding of the input files. The default, UTF-8, also works fine for ASCII text files. If you have a different encoding, enter it here. For more information on character sets and their names, see the JavaDocs for `java.nio.charset.Charset`.

Once you've filled in the appropriate values, press the "Run" button.

If an error occurs, a dialog will appear with the error message. (A stack trace will also be printed to the console, which may help you if the error was generated by your own annotator code.) Otherwise, an "Analysis Results" window will appear.

14.3 Viewing the Analysis Results

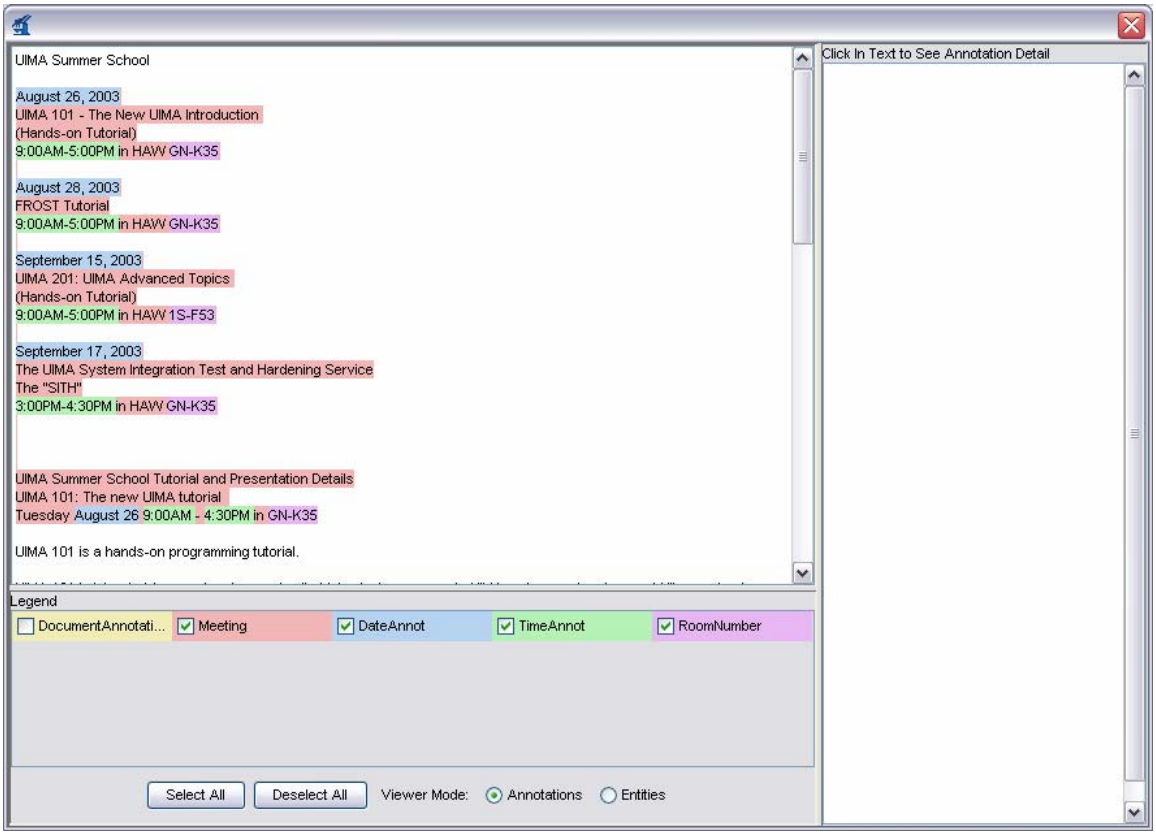
After a successful analysis, the "Analysis Results" window will appear.



The "Results Display Format" options at the bottom of this window show the different ways you can view your analysis – the Java Viewer, Java Viewer (JV) with User Colors, HTML, and XML. The default, Java Viewer, is recommended.

Once you have selected your desired Results Display Format, you can double-click on one of the files in the list to view the analysis done on that file.

For the Java viewer, the results display looks like this (for the TAE descriptor `docs/examples/descriptors/tutorial/ex4/MeetingDetectorTAE.xml`):



You can click the mouse on one of the highlighted annotations to see a list of all its features in the frame on the right.

If there are multiple annotation types in the view, you can control which ones are selected by using the checkboxes in the legend, the Select All button, or the Deselect All button.

14.4 Configuring the Annotation Viewer

The "JV User Colors" and the HTML viewer allow you to specify exactly which colors are used to display each of your annotation types. For the Java Viewer, you can also specify which types should be initially selected, and you can hide types entirely.

To configure the viewer, click the "Edit Style Map" button on the "Analysis Results" dialog. You should see a dialog that looks like this:

To change the color assigned to a type, simply click on the colored cell in the "Background" column for the type you wish to edit. This will display a dialog that allows you to choose the color. For the HTML viewer only, you can also change the foreground color.

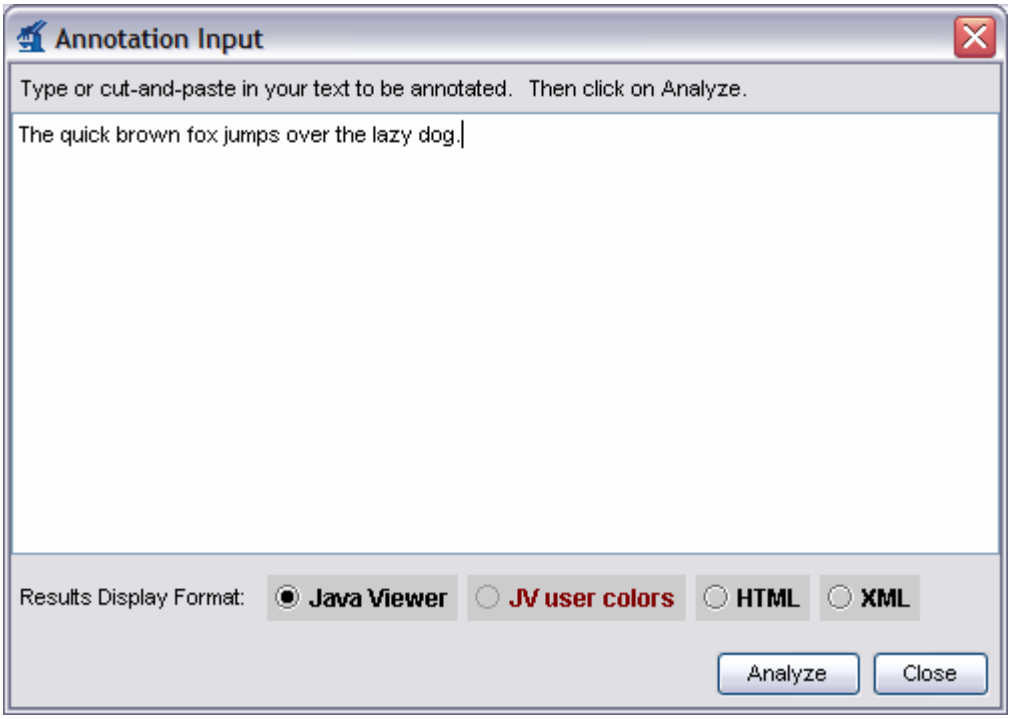
If you would like the type to be initially checked (selected) in the legend when the viewer is first launched, check the box in the "Checked" column. If you would like the type to never be shown in the viewer, click the box in the "Hidden" column. These settings only affect the Java Viewer, not the HTML view.

When you are done editing, click the "Save" button. This will save your choices to a file in the same directory as your TAE descriptor. From now on, when you view analysis results produced by this TAE using the "JV User Colors" or "HTML" options, the viewer will be configured as you have specified.

14.5 Interactive Mode

Interactive Mode allows you to analyze text that you type or cut-and-paste into the tool, rather than requiring that the documents be stored as files.

In the main Document Analyzer window, you can invoke Interactive Mode by clicking the "Interactive" button instead of the "Run" button. This will display a dialog that looks like this:



You can type or cut-and-paste your text into this window, then choose your Results Display Format and click the "Analyze" button. Your TAE will be run on the text that you supplied and the results will be displayed as usual.

14.6 View Mode

If you have previously run a TAE and saved its analysis results, you can use the Document Analyzer's View mode to view those results, without re-running your analysis. To do this, on the main Document Analyzer window simply select the location of your analyzed documents in the "Output Directory" dialog and click the "View" button. You can then view your analysis results as described in Section 14.3 ***Viewing the Analysis Results***.

Chapter 15 CAS Visual Debugger

The documentation for this component is found in a separate file in the docs/ directory, called `CASVisualDebugger.pdf`.

Chapter 16 JCasGen User Guide

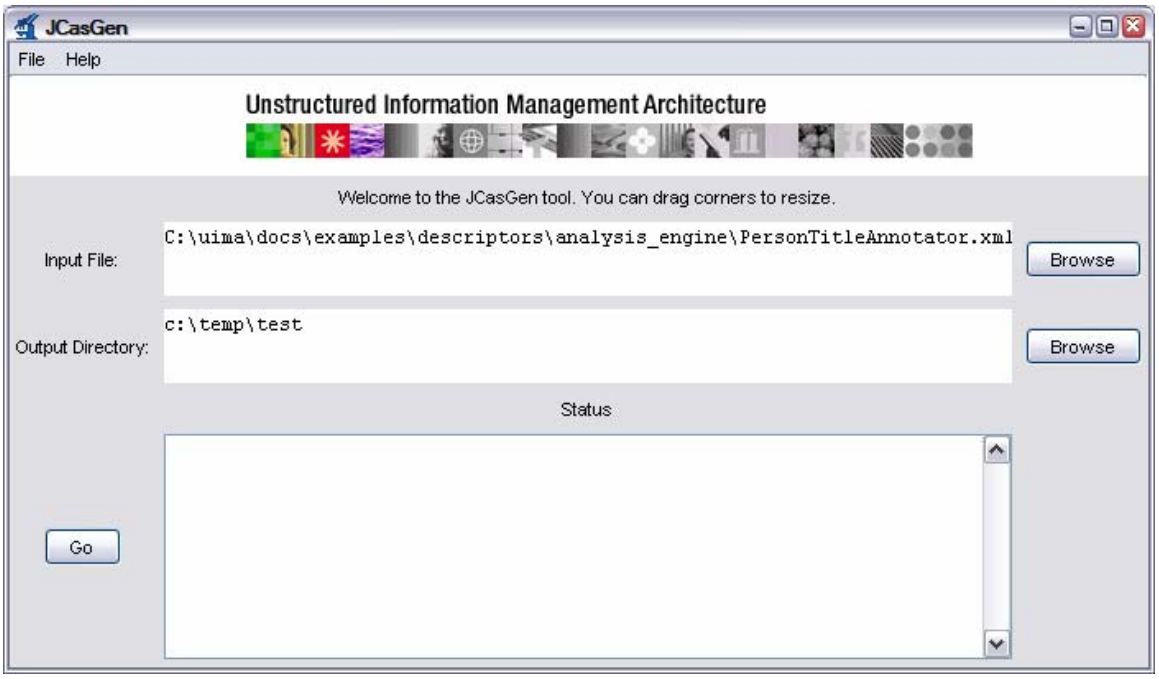
JCasGen reads a descriptor for an application, creates the merged type system specification by merging all the type system information from all the components referred to in the descriptor, and then uses this merged type system to create Java source files for classes that enable JCas access to the CAS. Java classes are not produced for the built-in types, except for the `uima.tcas.DocumentAnnotation` built-in type, which is the only built-in type that can be extended by users by adding features to it.

There are several versions of JCasGen. The basic version reads an XML descriptor which contains a type system descriptor, and generates the corresponding Java Class Models for those types. Variants exist for the Eclipse environment that allow merging the newly generated Java source code with previously augmented versions; see page 24-337 for a discussion of how the Java Class Models can be augmented by adding additional methods and fields.

Input to JCasGen needs to be mostly self-contained. In particular, any types that are defined to depend on user-defined supertypes must have that supertype defined, if the supertype is `uima.tcas.Annotation` or a subtype of it. Any features referencing ranges which are subtypes of uima.cas.String must have those subtypes included. If this is not followed, a warning message is given stating that the resulting generation may be inaccurate.

JCasGen is typically invoked using a shell script. These scripts can take 0, 1, or 2 arguments. The first argument is the location of the file containing the input XML descriptor. The second argument specifies where the generated Java source code should go. If it isn't given, JCasGen generates its output into a subfolder called JCas (or sometimes JCasNew – see below), of the first argument's path.

If no arguments are given to JCasGen, then it launches a GUI to interact with the user and ask for the same input. The GUI will remember the arguments you previously used. Here's what it looks like:



When running with automatic merging of the generated Java source with previously augmented versions, the output location is where the merge function obtains the source for the merge operation.

As is customary for Java, the generated class source files are placed in the appropriate subdirectory structure according to Java conventions that correspond to the package (name space) name.

The Java classes must be compiled and the resulting class files included in the class path of your application; you make these classes available for other annotator writers using your types, perhaps packaged as an xxx.jar file. If the xxx.jar file is made to contain only the Java Class Models for the CAS types, it can be reused by any users of these types.

Running stand-alone without Eclipse

There is no capability to automatically merge the generated Java source with previous versions, unless running with Eclipse. If run without Eclipse, no automatic merging of the generated Java source is done with any previous versions. In this case, the output is put in a folder called "JCasNew" unless overridden by specifying a second argument.

The distribution includes a shell script/bat file to run the stand-alone version, called jcasgen.

Running stand-alone with Eclipse

If you have Eclipse and EMF (EMF = Eclipse Modeling Framework; both of these are available from http://www.eclipse.org) installed (version 2.1 or later) JCasGen can merge the Java code it generates with previous versions, picking up changes you might have inserted by hand. The output (and source of the merge input) is in a folder "JCas" under the same path as the input XML file, unless overridden by specifying a second argument.

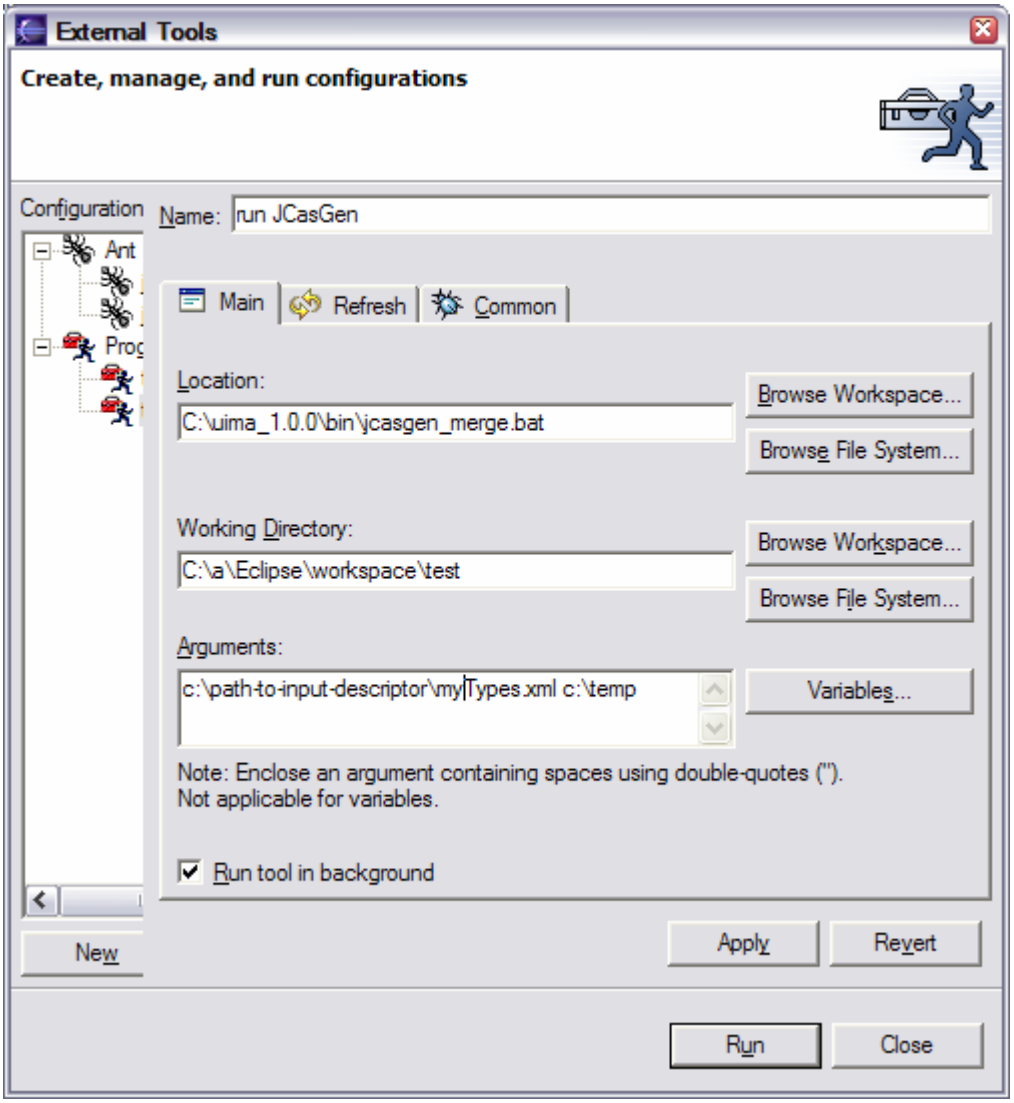
You must install the UIMA plug-ins into Eclipse to enable this function.

The distribution includes a shell script/bat file to run the stand-alone with Eclipse version, called jcasgen_merge. This works by starting Eclipse in "headless" mode (no GUI) and invoking JCasGen within Eclipse. You will need to set the ECLIPSE_HOME environment variable or modify the jcasgen_merge shell script to specify where to find Eclipse. The version of Eclipse needed is 2.1 or higher, with the EMF plug-in and the UIMA runtime plug-in installed. A temporary workspace is used; the name/location of this is customizable in the shell script.

Log and error messages are written to the UIMA log. This file is called uima.log, and is located in the default working directory, which if not overridden, is the startup directory of Eclipse.

Running within Eclipse

There are two ways to run JCasGen within Eclipse, with this release. The first way is to configure an Eclipse external tools launcher, and use it to run the stand-alone shell scripts, with the arguments filled in. Here's a picture of a typical launcher configuration screen (you get here by navigating from the top menu: Run → External Tools → External tools...).



The second way to run within Eclipse is to use the Analysis Engine Configurator tool *Chapter 7. The UIMA Component Descriptor Editor User's Guide*. This tool can be configured to automatically launch JCasGen whenever the descriptor is modified. In this release, this operation completely regenerates the files, even if just a small thing changed. So you probably don't want to enable this all the time. The configurator tool has an option to enable/disable this function.

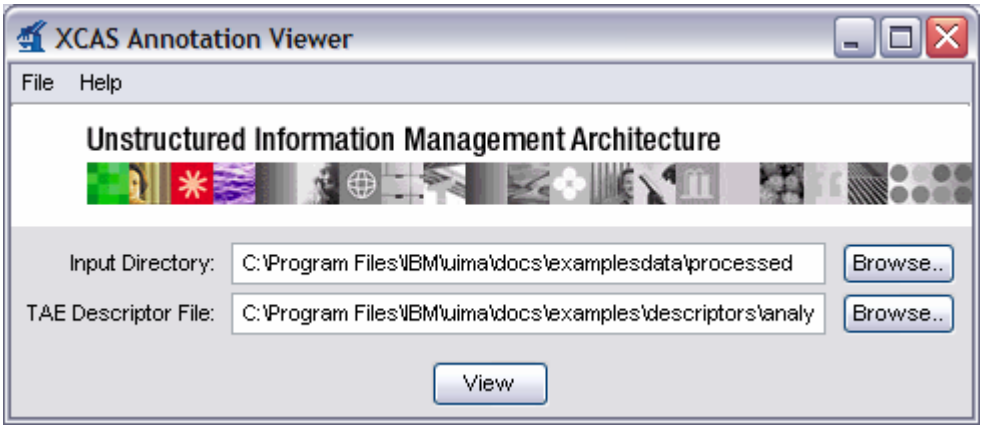
Chapter 17 XCAS Annotation Viewer

The *XCAS Annotation Viewer* is a tool for viewing analysis results that have been saved to your disk as *XCAS* files. XCAS is the XML representation of the CAS. In the UIMA SDK, XCAS files can be generated by:

- Running the Document Analyzer (see *Chapter 14* ***Document Analyzer User's Guide***), which saves XCAS files to the specified output directory.
- Running a Collection Processing Engine that includes the *XCAS Writer* CAS Consumer (`docs/examples/descriptors/cas_consumer/XCasWriterCasConsumer.xml`).
- Explicitly creating XCAS files from your own application using the com.ibm.uima.cas.impl.XCasSerializer class. The best way to learn how to do this is to look at the example code for the XCAS Writer CAS Consumer, located in `docs/examples/src/com/ibm/uima/examples/cpe/XCasWriterCasConsumer.java`.

  Note: The XCAS Annotation Viewer is not aware of Sofas, and only shows annotations for a default text Sofa.

You can run the XCAS Annotation Viewer by executing the `xcasAnnotationViewer` shell script located in the bin directory of the UIMA SDK. This will open the following window:



Select an input directory (which must contain XCAS files), and the descriptor for the TAE that produced the Analysis (which is needed to get the type system for the analysis). Then press the "View" button.

This will bring up a dialog where you can select a viewing format and double-click on a document to view it. This dialog is the same as the one that is described in Chapter *14.3* ***Viewing the Analysis Results***.

Part IV: Reference

Chapter 18 UIMA FAQs

What is UIMA? UIMA stands for Unstructured Information Management Architecture. It is component software architecture for the development, discovery, composition and deployment of multi-modal analytics for the analysis of unstructured information and its integration with search and knowledge management technologies.

UIMA processing occurs through a series of modules called analysis engines. The result of analysis is an assignment of semantics to the elements of unstructured data, for example, the indication that the phrase "Washington" refers to a person's name or that it refers to a place.

UIMA supports the rendering of these results in conventional structures, for example, relational databases or search engine indices, where the content of the original unstructured information may be efficiently accessed according to its inferred semantics.

UIMA is specifically designed to support the developer in creating, integrating, and deploying components across platforms and among disperse teams working to develop unstructured information management applications.

What's the difference between UIMA and the UIMA SDK? UIMA is an architecture which specifies component interfaces, design patterns, data representations and development roles.

The UIMA Software Development Kit (SDK) is a software system which includes a run-time framework, APIs and tools for implementing, composing, packaging and deploying UIMA components. It comes with a semantic search engine for indexing and querying over the results of analysis.

The UIMA run-time framework allows developers to plug-in their components and applications and run them on different platforms and according to different deployment options that range from tightly-coupled (running in the same process space) to loosely-coupled (distributed across different processes or machines for greater scale, flexibility and recoverability).

What is an Annotation? An annotation is a label, typically represented as string of characters, associated with a region of a document. The region may be the whole document.

An example is the label "Person" associated with the span of text "George Washington". We say that "Person" annotates "George Washington" in the sentence

"George Washington was the first president of the United States". The association of the label "Person" with a particular span of text is an annotation. Another example may have an annotation represent a topic, like "American Presidents" and be used to label an entire document.

Annotations are not limited to text. A label may annotate a region of an image or a segment of audio. The same concepts apply.

What is the CAS? The CAS stands for Common Analysis Structure. It provides cooperating UIMA components with a common representation and mechanism for shared access to the artifact being analyzed (e.g., a document, audio file, video stream etc.) and the current analysis results.

What does the CAS contain? The CAS is a data structure for which UIMA provides multiple interfaces. It contains and provides the analysis algorithm or application developer with access to

- the subject of analysis (the artifact being analyzed, like the document),
- the analysis results or metadata(e.g., annotations, parse trees, relations, entities etc.)
- indices to the analysis results and
- the type system (a schema for the analysis results)

Does the CAS only contain Annotations? No. The CAS contains the artifact being analyzed plus the analysis results. Analysis results are those statements recorded by analysis engines in the CAS. The most common form of analysis result is the addition of an annotation. But an analysis engine may write any structure that conforms to the CAS's type system into the CAS. These may not be annotations but may be other things, for example links between annotations and properties of objects associated with annotations.

Is the CAS just XML? No, in fact there are many possible representations of the CAS. If all of the analysis engines are running in the same process, an efficient, in-memory data object is used. If a CAS must be sent to an analysis engine on a remote machine, it can be done via an XML or a binary serialization of the CAS.

The UIMA framework provides serialization and de-serialization methods for a particular XML representation of the CAS named the XCAS. There are plans in the works to support an XMI representation of the CAS as well.

What is a Type System? Think of a type system as a schema or class model for the CAS. It defines the types of objects and their properties (or features) that may be instantiated in a CAS. A specific CAS conforms to a particular type system. UIMA components declare their input and output with respect to a type system.

Type Systems include the definitions of types, their properties, range types (these can restrict the value of properties to other types) and single-inheritance hierarchy of types.

What is a Sofa? Sofa stands for "Subject of Analysis". A CAS is associated with a single artifact being analysed by a collection of UIMA analysis engines. But a single artifact may have multiple independent views, each of which may be analyzed separately by a different set of analysis engines. For example, given a document it may have different translations, each of which are associated with the original document but each potentially analyzed by different engines. A CAS may have multiple Sofas each containing a different view of the original artifact. This feature is ideal for multi-modal analysis, where for example, one view of a video stream may be the video frames and the other the close-captions. In UIMA each view would get its own Sofa.

What's the difference between an Annotator and an Analysis Engine? In the terminology of UIMA, an annotator is simply some code that analyzes documents and outputs annotations on the content of the documents. The UIMA framework takes the annotator, together with metadata describing such things as the input requirements and outputs types of the annotator, and produces an analysis engine.

Analysis Engines contain the framework-provided infrastructure that allows them to be easily combined with other analysis engines in different flows and according to different deployment options (collocated or as web services, for example).

Are UIMA analysis engines web services? They can be deployed as such. Deploying an analysis engine as a web service is one of the deployment options supported by the UIMA framework.

How do you scale a UIMA application? The UIMA framework allows components such as analysis engines and CAS Consumers to be easily deployed as services or in other containers and managed by systems middleware designed to scale. UIMA applications tend to naturally scale-out across documents allowing many documents to be analyzed in parallel.

A component in the UIMA framework called the CPM (Collection Processing Manager) has a host of features and configuration settings for scaling an application to increase its throughput and recoverability.

What does it mean to embed UIMA in systems middleware? An example of an embedding would be the deployment of a UIMA analysis engine as an Enterprise Java Bean inside an application server such as IBM WebSphere. Such an embedding allows the deployer to take advantage of the features and tools provided by WebSphere for achieving scalability, service management, recoverability etc.

UIMA is independent of any particular systems middleware, so analysis engines could be deployed on other application servers as well.

Do Analysis Engines have to be "stateless"? Technically, No. But Analysis Engines developers are encouraged not to maintain state between documents that would prevent their engine from working as advertised if switched into a different flow or onto a different document collection.

UIMA defines another type of component, the CAS Consumer, which is intended to maintain state across documents and is typically associated with some resource like a database or search engine that aggregates analysis results across an entire collection.

Is engine meta-data compatible with web services and UDDI? All UIMA component implementations are associated with Component Descriptors which represents metadata describing various properties about the component to support discovery, reuse, validation, automatic composition and development tooling. In principle, UIMA component descriptors are compatible with web services and UDDI. However, the UIMA framework currently uses its own XML representation for component metadata. It would not be difficult to convert between UIMA's XML representation and the WSDL and UDDI standards.

How is the CPM different from a CPE? The UIMA framework includes a Collection Processing Manager or CPM for managing the execution of a workflow of UIMA components orchestrated to analyze a large collection of documents. The UIMA developer does not implement or describe a CPM. It is a built-in part of the framework. It is a piece of infrastructure code that handles CAS transport, instance management, batching, check-pointing, statistics collection and failure recovery in the execution of a collection processing workflow.

A Collection Processing Engine (CPE) is component that the UIMA developer creates by specifying a CPE descriptor. A CPE descriptor points to a series of UIMA components including a Collection Reader, CAS Initializer, Analysis Engine(s) and CAS Consumers. These components organized in a work flow define a collection analysis job or CPE. A CPE acquires documents from a source collection, initializes CASs with document content, performs document analysis and then produces collection level results (e.g., search engine index, database etc). The CPM is the execution engine for a CPE.

What is Semantic Search and what is its relationship to UIMA? Semantic Search refers to a document search paradigm that allows users to search based not just on the keywords contained in the documents, but also on the semantics associated with the text by analysis engines. UIMA applications perform analysis on text documents and generate semantics in the form of annotations on regions of text. For example, a UIMA analysis engine may discover the text "First Financial Bank" to refer to an

organization and annotated it as such. With traditional keyword search, the query *"first"* will return all documents that contain that word. "First" is a frequent and ambiguous term – it occurs a lot and can mean different things in different places. If the user is looking for organizations that contain that word "first" in their names, s/he will likely have to sift through lots of documents containing the word "first" used in different ways. Semantic Search exploits the results of analysis to allow more precise queries. For example, the semantic search query *<organization> first </organization>* will rank first documents that contain the word "first" as part of the name of an organization. The UIMA SDK documentation demonstrates how UIMA applications can be built using semantic search. It provides details about the XML Fragment Query language. This is the particular query language used by the semantic search engine that comes with the SDK.

Is an XML Fragment Query valid XML? Not necessarily. The XML Fragment Query syntax is used to formulate queries interpreted by the semantic search engine that ships with the UIMA SDK. This query language relies on basic XML syntax as an intuitive way to describe hierarchical patterns of annotations that may occur in a CAS. The language deviates from valid XML in order to support queries over "overlapping" or "cross-over" annotations and other features that affect the interpretation of the query by the query processor. For example, it admits notations in the query to indicate whether a keyword or an annotation is optional or required to match a document.

Does UIMA support modalities other than text? The UIMA architecture supports the development, discovery, composition and deployment of multi-modal analytics including text, audio and video. Applications that process text, speech and video have been developed using UIMA. This release of the SDK, however, does not include examples of these multi-modal applications.

It does however include documentation and programming examples for using the key feature required for building multi-modal applications. UIMA supports multiple subjects of analysis or Sofas. These allow multiple views of a single artifact to be associated with a CAS. For example, if an artifact is a video stream, one Sofa could be associated with the video frames and another with the closed-captions text. UIMA's multiple Sofa feature is included and described in this release of the SDK.

How does UIMA compare to other similar work? A number of different frameworks for NLP have preceded UIMA. Two of them were developed at IBM Research and represent UIMA's early roots. For details please refer to the UIMA article that appears in the IBM Systems Journal Vol. 43, No. 3 (http://www.research.ibm.com/journal/sj/433/ferrucci.html).

UIMA has advanced that state of the art along a number of dimensions including: support for distributed deployments in different middleware environments, easy framework embedding in different software product platforms (key for commercial

applications), broader architectural converge with its collection processing architecture, support for multiple-modalities, support for efficient integration across programming languages, support for a modern software engineering discipline calling out different roles in the use of UIMA to develop applications, the extensive use of descriptive component metadata to support development tooling, component discovery and composition. (Please note that not all of these features are available in this release of the SDK.)

How does UIMA relate to IBM Products? UIMA analysis engines and annotators are already used within several IBM products, including, IBM's new enterprise search offering, WebSphere Information Integrator OmniFind Edition (http://www.ibm.com/software/data/integration/search.html), and IBM's WebSphere Portal Server offering. All new analysis technology deployed into IBM products is based on the UIMA architecture.

Is UIMA Open Source? Yes. The UIMA SDK is freely available on the IBM alphaWorks site (http://www.alphaworks.ibm.com/tech/uima) and the source code for the UIMA framework is available on SourceForge (http://uima-framework.sourceforge.net).

What Java level and OS are required for the UIMA SDK? The UIMA SDK requires a Java 1.4 level; it will not run on a 1.3 (or earlier levels). It has been tested with IBM Java SDK v1.4.2, which is included as part of the UIMA SDK. It has been tested on Windows 2000, Windows XP and Linux Intel 32bit platforms. Other platforms and JDK implementations, including Java 1.5, may work, but have not been significantly tested.

Can I build my UIM application on top of UIMA? Yes. The UIMA SDK license does not restrict its usage to specific scenarios, and we are of course very interested in your feedback to help us making UIMA the right platform for building UIMA applications. UIMA is officially supported inside IBM's WebSphere Information Integration Omnifind Edition product (http://www.ibm.com/developerworks/db2/zones/db2ii or http://www.ibm.com/software/data/integration/db2ii/editions_womnifind.html). The UIMA SDK on IBM's alphaWorks is supported on a "best can do" basis. If you are interested in a more formal support agreement, or would like to include UIMA in a commercial solution, beyond using the officially supported product, please contact IBM for additional options.

Chapter 19 Glossary of Key Terms and Concepts

Analysis Engine: A program that analyzes artifacts (e.g. documents) and infers information about them, and which implements the UIMA Analysis Engine interface Specification. It does not matter how the program is built, with what framework or whether or not it contains component ("sub") Analysis Engines.

Annotation: The association of a label with a region of text (or other type of artifact). For example, the label "Person" associated with a region of text "John Doe" constitutes an annotation. We say "Person" annotates the span of text from X to Y containing exactly "John Doe". An annotation is represented as a special type in a UIMA type system. It is the type used to record the labeling of regions of a subject of analysis.

Annotator: A software component that implements the UIMA annotator interface. Annotators are implemented to produce and record annotations over regions of an artifact (e.g., text document, audio, and video).

Aggregate Analysis Engine: An Analysis Engine that is implemented by configuring a collection of component Analysis Engines.

CAS: The UIMA Common Analysis Structure is the primary data structure which UIMA analysis components use to represent and share analysis results. It contains:

- The artifact. This is the object being analyzed such as a text document or audio or video stream. The CAS projects one or more views of the artifact. Each view is referred to as a Subject of Analysis.
- A type system description – indicating the types, subtypes, and their features.
- Analysis metadata – "standoff" annotations describing the artifact or a region of the artifact
- An index repository to support efficient access to and iteration over the results of analysis.

UIMA's primary interface to this structure is provided by a class called the Common Analysis System. We use "CAS" to refer to both the structure and system. Where the common analysis structure is used through a different interface, the particular implementation of the structure is indicated, For example, the JCas is a native Java object representation of the contents of the common analysis structure.

CAS Consumer: A component that receives each CAS in the collection after it has been processed by an Analysis Engine. The CAS Consumer may then perform collection-level analysis and construct an application-specific, aggregate data structure.

CAS Initializer: A component that populates a CAS from a raw document. For example, if the document is HTML, a CAS Initializer might store a detagged version of the document in the CAS and also create inline annotations derived from the tags. For example <p> tags might be translated into inline Paragraph annotations in the CAS.

CAS Processor: A component that takes a CAS as input and returns a CAS as output. There are two types of CAS Processors: Analysis Engines and CAS Consumers.

CDE: The Component Descriptor Editor . This is the Eclipse tool that lets you conveniently edit the UIMA descriptors, described in Chapter 9 ***Component Descriptor Editor User's Guide*** on page 9-179.

Collection Processing Engine (CPE): Performs Collection Processing through the combination of a Collection Reader, an optional CAS Initializer, an Analysis Engine, and one or more CAS Consumers. The Collection Processing Manager (CPM) manages the execution of the engine.

Collection Processing Manager (CPM): A module in the framework that manages the execution of collection processing, routing CASs from the Collection Reader to an Analysis Engine and then to the CAS Consumers. The CPM provides feedback such as performance statistics and error reporting and may implement other features such as parallelization.

Collection Reader: A component that reads documents from some source, for example a file system or database. Each document is returned as a CAS that may then be processed by Analysis Engines. If the task of populating a CAS from the document is complex, a Collection Reader may choose to use a CAS Initializer for this purpose.

Fact Search: A search that given fact pattern, returns facts extracted from a collection of documents by a set of analysis engines that match the fact pattern.

Feature: A data member or attribute of a type. Each feature itself has an associated type. In the database analogy where types are tables, features are columns.

Hybrid Analysis Engine: An Aggregate Analysis Engine where more than one of its component Analysis Engines are deployed the same address space and one or more are deployed remotely (part tightly and part loosely-coupled).

Index: Data in the CAS can only be retrieved using Indexes. Indexes are analogous to the indexes that are specified on tables of a database. Indexes belong

to Index Repositories; there is one Repository for the base CAS as well as additional ones for each TCAS view of the CAS. Indexes are specified to retrieve instances of some CAS Type (including its subtypes), and can be sorted in a user-definable way. For example, all types derived from the UIMA built-in type `uima.tcas.annotation` contain begin and end features, which mark the begin and end offsets in the text where this annotation occurs. One may then specify that types should be retrieved sequentially by begin (ascending) and end (descending). In this case, iterating over the annotations, one first obtains annotations that come sequentially first in the text, while favoring shorter annotations, in the case where two annotations start at the same offset.

JCas: A Java object interface to the contents of the CAS, where each type in the CAS is represented as a Java class, each feature a property and each instance of a type a Java object.

Keyword Search: The standard search method where one supplies words (or "keywords") and candidate documents are returned.

Knowledge Base: A collection of data that may be interpreted as a set of facts and rules considered true in a possible world.

Loosely-Coupled Analysis Engine: An Aggregate Analysis Engine where no two of its component Analysis Engines run in the same address space but where each is remote with respect to the others that make up the aggregate. Ideal for using remote Analysis Engine services that are not locally available, or for quickly assembling and testing functionality in cross-language, cross-platform distributed environments. Also better enables distributed scaleable implementations where quick recoverability may have a greater impact on overall throughput than analysis speed.

Ontology: The part of a knowledge base that defines the semantics of the data axiomatically.

PEAR: An archive file that packages up a UIMA component its code, descriptor files and other resources required to install and run it in another environment. You can generate PEAR files using utilities that come with the UIMA SDK.

Primitive Analysis Engine: An Analysis Engine that is composed of a single Annotator but NO component (or "sub") Analysis Engines inside of it.

Semantic Search: A search where the semantic intent of the query is specified using one or more entity or relation specifiers. For example, one could specify that they are looking for a person (named) "Bush." Such a query would then not return results about the kind of bushes that grow in your garden but rather just persons named bush.

Structured Information: Items stored in structured resources such as search engine indices, databases or knowledge bases. The canonical example of structured information is the database table. Each element of information in the database is associated with a precisely defined schema where each table column heading indicates its precise semantics, defining exactly how the information should be interpreted by a computer program or end-user.

Subject of Analysis (Sofa): An piece of data (e.g., text document, image, audio segment, or video segment), intended for analysis by UIMA analysis components. It is made available to UIMA analysis components through a TCAS, which is a particular view of a CAS associated with a particular Subject of Analysis. There can be multiple Sofas contained within one CAS, each representing a different view of the original artificat – for example, an audio file could be the original artifact, and correspond to one Sofa, and another could be the output of a voice-recognition component, where the Sofa would be the corresponding text. document. Sofas maybe analyzed independently or simultaneously.

TAE: A specialization of Analysis Engine for processing artifacts where annotations over those artifacts are being produced. See Sofa.

TCAS: a particular view of the CAS corresponding to one particular Sofa. The TCAS has all the methods of a CAS and additional methods related to its Sofa (subject of analysis). It also provides access to an index repository holding instances of Feature Structures associated with this Sofa.

Tightly-Coupled Analysis Engine: An Aggregate Analysis Engine where all of its component Analysis Engines run in the same address space.

Type: An object used to store the results of analysis. Types are defined using inheritance, so some types may be defined purely for the sake of defining other types, and are in this sense "abstract types." Types usually contain features, which are attributes or properties of the type. A type is roughly equivalent to a class in an object oriented programming language, or a table in a database. Types may be indexed for fast retrieval.

Type System: A collection of related types. Each Analysis Engine or CAS Consumer has its own type system. Type systems are often shared across Analysis Engines. A type system is roughly analogous to a set of related classes in object oriented programming, or a set of related tables in a database. The type system / type / feature terminology comes from computational linguistics.

Unstructured Information: The canonical example of unstructured information is the natural language text document. The intended meaning of a document's content is only implicit and its precise interpretation by a computer program requires some degree of analysis to explicate the document's semantics. Other

examples include audio, video and images. Unstructured information is contrasted with *structured information*. The canonical example of structured information is the database table. Each element of information in the database is associated with a precisely defined schema where each table column heading indicates its precise semantics, defining exactly how the information elements should be interpreted by a computer program or end-user.

UIMA: **U**nstructured **I**nformation **M**anagement **A**rchitecture: a software architecture which specifies component interfaces, design patterns and development roles for creating, describing, discovering, composing and deploying multi-modal analysis capabilities.

UIMA Framework: A Java-based implementation of the UIMA architecture. It provides a run-time environment in which developers can plug in and run their UIMA component implementations and with which they can build and deploy UIM applications. The framework is not specific to any IDE or platform. The original design for the framework was largely inspired by the original TAF and Talent systems developed in IBM Watson Research labs and IBM Software Group.

UIMA Software Development Kit (SDK): includes an all-Java implementation of the UIMA framework for the implementation, description, composition and deployment of UIMA components and applications. It also provides the developer with an Eclipse-based (www.eclipse.org) development environment that includes a set of tools and utilities for using UIMA.

XCAS: An XML representation of the CAS. The XCAS can be used for saving and restoring CASs to and from streams. The UIMA SDK provides serialization and de-serialization methods for the XCAS format.

Chapter 20 Component Descriptor Reference

This chapter is the reference guide for the UIMA SDK's Component Descriptor XML schema. A *Component Descriptor* (also sometimes called a *Resource Specifier* in the code) is an XML file that either (a) completely describes a component, including all information needed to construct the component and interact with it, or (b) specifies how to connect to and interact with an existing component that has been published as a remote service. *Component* (also called *Resource*) is a general term for modules produced by UIMA developers and used by UIMA applications. The types of Components are: Analysis Engines, Collection Readers, CAS Initializers, CAS Consumers, and Collection Processing Engines. However, Collection Processing Engine Descriptors are significantly different in format and are covered in a separate chapter, *UIMA Collection Processing Engine Descriptor Reference.*

Section 20.1 describes the notation used in this chapter.

Section 20.2 describes the UIMA SDK's *import* syntax, used to allow XML descriptors to import information from other XML files, to allow sharing of information between several XML descriptors.

Section 20.4 describes the XML format for *Analysis Engine Descriptors*. These are descriptors that completely describe Analysis Engines, including all information needed to construct and interact with them.

Section 20.5 describes the XML format for *Collection Processing Component Descriptors*. This includes Collection Iterator, CAS Initializer, and CAS Consumer Descriptors.

Section 20.6 describes the XML format for *Service Client Descriptors*, which specify how to connect to and interact with resources deployed as remote services.

20.1 Notation

This chapter uses an informal notation to specify the syntax of Component Descriptors. The formal syntax is defined by an XML schema definition, which is contained in two files – `resourceSpecifierSchema.xsd` and `TaeSpecifierSchema.xsd`, both of which are in the `uima_core.jar` file.

The notation used in this chapter is:

- An ellipsis (...) inside an element body indicates that the substructure of that element has been omitted (to be described in another section of this chapter). An example of this would be:

```
<analysisEngineMetaData>
...
</analysisEngineMetaData>
```

- An ellipsis immediately after an element indicates that the element type may be may be repeated arbitrarily many times. For example:

```
<parameter>[String]</parameter>
<parameter>[String]</parameter>
...
```

  indicates that there may be arbitrarily many parameter elements in this context.

- Bracketed expressions (e.g. `[String]`) indicate the type of value that may be used at that location.
- A vertical bar, as in `true|false`, indicates alternatives. This can be applied to literal values, bracketed type names, and elements.
- Which elements are optional and which are required is specified in prose, not in the syntax definition.

20.2 Imports

The UIMA SDK defines a particular syntax for XML descriptors to import information from other XML files. When one of the following appears in an XML descriptor:

```
<import location="[URL]" /> or
<import name="[Name]" />
```

it indicates that information from a separate XML file is being imported. Note that imports are allowed only in certain places in the descriptor. In the remainder of this chapter, it will be indicated at which points imports are allowed.

If an import specifies a `location` attribute, the value of that attribute specifies the URL at which the XML file to import will be found. This can be a relative URL, which will be resolved relative to the descriptor containing the `import` element, or an absolute URL. Relative URLs can be written without a protocol/scheme (e.g., "file:"), and without a host machine name. In this case the relative URL might look something like `com/ibm/myproj/MyTypeSystem.xml`.

An absolute URL is written with one of the following prefixes, followed by a path such as `com/ibm/myproj/MyTypeSystem.xml`:

- file:/ << has no network address
- file:/// << has an empty network address
- file://some.network.address/

For more information about URLs, please read the javadoc information for the Java class "URL".

If an import specifies a `name` attribute, the value of that attribute should take the form of a Java-style dotted name (e.g. `com.ibm.myproj.MyTypeSystem`). An .xml file with this name will be searched for in the classpath or datapath (described below). As in Java, the dots in the name will be converted to file path separators. So an import specifying the example name in this paragraph will result in a search for `com/ibm/myproj/MyTypeSystem.xml` in the classpath or datapath.

The datapath works similarly to the classpath but can be set programmatically through the resource manager API. Application developers can specify a datapath during initialization, using the following code:

```
ResourceManager resMgr = UIMAFramework.newDefaultResouceManager();
resMgr.setDataPath(yourPathString);
TextAnalysisEngine tae = UIMAFramework.produceTAE(desc, resMgr, null);
```

The default datapath for the entire JVM can be set via the `uima.datapath` Java system property, but this feature should only be used for standalone applications that don't need to run in the same JVM as other code that may need a different datapath.

The UIMA SDK also supports XInclude, a W3C candidate recommendation, to include XML files within other XML files. However, it is recommended that the import syntax be used instead, as it is more flexible and better supports tool developers.

> **Note:** UIMA tools for editing XML descriptors do not support the use of xi:include because they cannot correctly determine what parts of a descriptor are updatable, and what parts are included from other files. They do support the use of <import>.

To use XInclude, you first must include the XInclude namespace in your document's root element, e.g.:

```
<taeDescription xmlns="http://uima.watson.ibm.com/resourceSpecifier"
xmlns:xi="http://www.w3.org/2001/XInclude">
```

Then, you can include a file using the syntax `<xi:include href="[URL]"/>`

where [URL] can be any relative or absolute URL referring to another XML document. The referred-to document must be a valid XML document, meaning that it must consist of exactly one root element and must define all of the namespace prefixes that it uses. The default namespace (generally `http://uima.watson.ibm.com/resourceSpecifier`) will be inherited from the parent document. When UIMA parses the XML document, it will automatically replace

the <xi:include> element with the entire XML document referred to by the href. For more information on XInclude see http://www.w3.org/TR/xinclude/.

20.3 Type System Descriptors

A Type System Descriptor is used to define the types and features that can be represented in the CAS. A Type System Descriptor can be imported into an Analysis Engine or Collection Processing Component Descriptor.

The basic structure of a Type System Descriptor is as follows:

```
<typeSystemDescription
xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <name> [String] </name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>
  <imports>
    <import ...>
    ...
  </imports>
  <types>
    <typeDescription>
      ...
    </typeDescription>
    ...
  </types>
</typeSystemDescription>
```

All of the subelements are optional.

Imports

The imports section allows this descriptor to import types from other type system descriptors. The import syntax is described in section 20.1 of this chapter. A type system may import any number of other type systems and then define additional types which refer to imported types. Circular imports are allowed.

Types

The types element contains zero or more typeDescription elements. Each typeDescription has the form:

```
<typeDescription>
  <name>[TypeName]</name>
  <description>[String]</description>
  <supertypeName>[TypeName]</supertypeName>
  <features>
```

```
    ...
  </features>
</typeDescription>
```

The name element contains the name of the type. A [TypeName] is a dot-separated list of names, where each name consists of a letter followed by any number of letters, digits, or underscores. TypeNames are case sensitive. Letter and digit are as defined by Java; therefore, any Unicode letter or digit may be used (subject to the character encoding defined by the descriptor file's XML header). The name following the final dot is considered to be the "short name" of the type; the preceding portion is the namespace (analogous to the package.class syntax used in Java). Namespaces beginning with uima are reserved and should not be used. Examples of valid type names are:

- test.TokenAnnotation
- org.myorg.tae.TokenAnnotation
- com.my_company.proj123.TokenAnnotation

These would all be considered distinct types since they have different namespaces. Best practice here is to follow the normal Java naming conventions of having namespaces be all lowercase, with the short type names having an initial capital, but this is not mandated, so ABC.mYtyPE is an allowed type name. While type names without namespaces (e.g. TokenAnnotation alone) are allowed, the JCas does not support them and so their use is strongly discouraged.

The description element contains a textual description of the type. The superTypeName element contains the name of the type from which it inherits (this can be set to the name of another user-defined type, or it may be set to any built-in type which may be subclassed, such as "uima.tcas.Annotation" for a new annotation type or "uima.cas.TOP" for a new type that is not an annotation). All three of these elements are required.

Features

The features element of a typeDescription is required only if the type we are specifying introduces new features. If the features element is present, it contains zero or more featureDescription elements, each of which has the form:

```
<featureDescription>
  <name>[Name]</name>
  <description>[String]</description>
  <rangeTypeName>[Name]</rangeTypeName>
  <elementType>[Name]</elementType>
  <multipleReferencesAllowed>true|false</multipleReferencesAllowed>
</featureDescription>
```

A feature's name follows the same rules as a type short name – a letter followed by any number of letters, digits, or underscores. Feature names are case sensitive.

The feature's rangeTypeName specifies the type of value that the feature can take. This may be the name of any type defined in your type system, or one of the predefined types. All of the predefined types have names that are prefixed with uima.cas or uima.tcas, for example:

```
uima.cas.TOP
uima.cas.Sofa
uima.cas.String
uima.cas.Integer
uima.cas.Float
uima.cas.FSArray
uima.cas.StringArray
uima.cas.IntegerArray
uima.cas.FloatArray
uima.cas.FSList
uima.cas.StringList
uima.cas.IntegerList
uima.cas.FloatList
uima.tcas.Annotation.
```

For a complete list of predefined types, see the CAS API documentation.

The elementType of a feature is optional, and applies only when the rangeTypeName is uima.cas.FSArray or uima.cas.FSList The elementType specifies what type of value can be assigned as an element of the array or list. This must be the name of a non-primitive type. If omitted, it defaults to uima.cas.TOP, meaning that any FeatureStructure can be assigned as an element the array or list. Note: depending on the CAS Interface that you use in your code, this constraint may or may not be enforced.

The multipleReferencesAllowed feature is optional, and applies only when the rangeTypeName is an array or list type (it applies to arrays and lists of primitive as well as non-primitive types). Setting this to false (the default) indicates that this feature has exclusive ownership of the array or list, so changes to the array or list are localized. Setting this to true indicates that the array or list may be shared, so changes to it may affect other objects in the CAS. Note: there is currently no guarantee that the framework will enforce this restriction. However, this setting may affect how the CAS is serialized.

String Subtypes

There is one other special type that you can declare – a subset of the String type that specifies a restricted set of allowed values. This is useful for features that can have only certain String values, such as parts of speech. Here is an example of how to declare such a type:

```
<typeDescription>
  <name>PartOfSpeech</name>
  <description>A part of speech.</description>
```

```
    <supertypeName>uima.cas.String</supertypeName>
    <allowedValues>
      <value>
        <string>NN</string>
        <description>Noun, singular or mass.</description>
      </value>
      <value>
        <string>NNS</string>
        <description>Noun, plural.</description>
      </value>
      <value>
        <string>VB</string>
        <description>Verb, base form.</description>
      </value>

      ...
    </allowedValues>
  </typeDescription>
```

20.4 Analysis Engine Descriptors

Analysis Engine (AE) descriptors completely describe Analysis Engines. There are two basic types of Analysis Engines – *Primitive* and *Aggregate*. A *Primitive* Analysis Engine is a container for a single *annotator*, where as an *Aggregate* Analysis Engine is composed of a collection of other Analysis Engines. (For more information on this and other terminology, see *Chapter 2* ***UIMA Conceptual Overview***)

Both Primitive and Aggregate Analysis Engines have descriptors, and the two types of descriptors have some similarities and some differences. Primitive Analysis Engine descriptors are discussed first, in Section 20.4.1 . Section 20.4.2 then describes how Aggregate Analysis Engine descriptors are different.

Analysis Engines can analyze any type of data. A common case is the analysis of particular Subjects, including things like text documents, in which case a specialization of Analysis Engine called a *TAE* is used. TAE descriptors are almost exactly identical to general Analysis Engine descriptors, but include information relating to the subject of analysis.

20.4.1 Primitive Analysis Engine Descriptors

Basic Structure

```
<?xml version="1.0" encoding="UTF-8" ?>
<taeDescription xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <frameworkImplementation>com.ibm.uima.java</frameworkImplementation>

  <primitive>true</primitive>
  <annotatorImplementationName> [String] </annotatorImplementationName>
```

```
    <analysisEngineMetaData>
      ...
    </analysisEngineMetaData>
    <externalResourceDependencies>
      ...
    </externalResourceDependencies>
    <resourceManagerConfiguration>
      ...
    </resourceManagerConfiguration>
  </taeDescription>
```

The document begins with a standard XML header. This example is for a TAE descriptor, for which the root element is named `<taeDescription>`. For a general Analysis Engine descriptor, use `analysisEngineDescription`, as the name of the root element instead.

Within the root element we declare that we are using the XML namespace `http://uima.watson.ibm.com/resourceSpecifier`. It is required that this namespace be used; otherwise, the descriptor will not be able to be validated for errors.

The first subelement, `<frameworkImplementation>`, currently must have the value `com.ibm.uima.java`, or `com.ibm.uima.cpp`. In future versions, there may be other framework implementations, or perhaps implementations produced by other vendors.

The second subelement, `<primitive>`, contains the Boolean value `true`, indicating that this XML document describes a *Primitive* Analysis Engine.

The next subelement, `<annotatorImplementationName>` is how the UIMA framework determines which annotator class to use. This should contain a fully-qualified Java class name for Java implementations, or the name of a .dll or .so file for C++ implementations.

The `<analysisEngineMetaData>` object contains descriptive information about the analysis engine and what it does. It is described in the section ***Analysis Engine Metadata***.

The `<externalResourceDependencies>` and `<resourceManagerConfiguration>` elements declare the `external` resource files that the analysis engine relies upon. They are optional and are described in the section ***External Resource Dependencies*** and ***Resource Manager Configuration***.

Analysis Engine Metadata

```
  <analysisEngineMetaData>
    <name> [String] </name>
    <description>[String]</description>
```

```
    <version>[String]</version>
    <vendor>[String]</vendor>
    <configurationParameters> ...  </configurationParameters>
    <configurationParameterSettings>
      ...
    </configurationParameterSettings>
    <typeSystemDescription> ... </typeSystemDescription>
    <typePriorities> ... </typePriorities>
    <fsIndexes> ... </fsIndexes>
    <capabilities> ... </capabilities>
    <operationalProperties> ... </operationalProperties>
  </analysisEngineMetaData>
```

The `analysisEngineMetaData` element contains four simple string fields – `name`, `description`, `version`, and `vendor`. Only the `name` field is required, but providing values for the other fields is recommended. The `name` field is just a descriptive name meant to be read by users; it does not need to be unique across all Analysis Engines.

The other sub-elements – `configurationParameters`, `configurationParameterSettings`, `typeSystemDescription`, `typePriorities`, `fsIndexes`, `capabilities` and `operationalProperties` are described in the following sections. The only one of these that is required is `capabilities`; the others are all technically optional but generally necessary for an analysis engine of any complexity.

Configuration Parameter Declaration

Configuration Parameters are made available to annotator implementations and applications by the following interfaces: `AnnotatorContext` (passed as an argument to the initialize() method of an annotator), `ConfigurableResource` (every Analysis Engine implements this interface), and the `UimaContext` (you can get this from any resource, including Analysis Engines, using the method `getUimaContext()`).

Use AnnotatorContext within annotators and UimaContext outside of annotators (for instance, in CasConsumers, or the containing application) to access configuration parameters.

Configuration parameters are set from the corresponding elements in the XML descriptor for the application. If you need to programmatically change parameter settings within an application, you can use methods in ConfigurableResource; if you do this, you need to call reconfigure() afterwards to have the UIMA framework notify all the contained analysis components that the parameter configuration has changed (the analysis engine's reinitialize() methods will be called). Note that in the current implementation, only integrated deployment components have configuration parameters passed to them; remote components obtain their

parameters from their remote startup environment. This will likely change in the future.

There are two ways to specify the <configurationParameters> section – as a list of configuration parameters or a list of groups. A list of parameters, which are not part of any group, looks like this:

```
<configurationParameters>
  <configurationParameter>
    <name>[String]</name>
    <description>[String]</description>
    <type>String|Integer|Float|Boolean</type>
    <multiValued>true|false</multiValued>
    <mandatory>true|false</mandatory>
    <overrides>
      <parameter>[String]</parameter>
      <parameter>[String]</parameter>
        ...
    </overrides>
  </configurationParameter>
  <configurationParameter>
    ...
  </configurationParameter>
    ...
</configurationParameters>
```

For each configuration parameter, the following are specified:

- **name** – the name by which the annotator code refers to the parameter. All parameters declared in an analysis engine descriptor must have distinct names. (required). The name is composed of normal Java identifier characters.
- **description** – a natural language description of the intent of the parameter (optional)
- **type** – the data type of the parameter's value – must be one of `String`, `Integer`, `Float`, or `Boolean` (required).
- **multiValued** – `true` if the parameter can take multiple-values (an array), `false` if the parameter takes only a single value (optional, defaults to false).
- **mandatory** – `true` if a value must be provided for the parameter (optional, defaults to false).
- **overrides** – this is used only in aggregate Analysis Engines, but is included here for completeness. See ***Configuration Parameter Overrides*** for a discussion of configuration parameter overriding in aggregate Analysis Engines. (optional)

A list of groups looks like this:

```
<configurationParameters defaultGroup="[String]"
    searchStrategy="none|default_fallback|language_fallback" >
```

```
  <commonParameters>
    [zero or more parameters]
  </commonParameters>

  <configurationGroup names="name1 name2 name3 ...">
    [zero or more parameters]
  </configurationGroup>

  <configurationGroup names="name4 name5 ...">
    [zero or more parameters]
  </configurationGroup>

  ...
</configurationParameters>
```

Both the `<commonParameters>` and `<configurationGroup>` elements contain zero or more `<configurationParameter>` elements, with the same syntax described above.

The `<commonParameters>` element declares parameters that exist in all groups. Each `<configurationGroup>` element has a names attribute, which contains a list of group names separated by whitespace (space or tab characters). Names consist of any number of non-whitespace characters; however the Component Description Editor tool restricts this to be normal Java identifiers, including the period (.) and the dash (-). One configuration group will be created for each name, and all of the groups will contain the same set of parameters.

The `defaultGroup` attribute specifies the name of the group to be used in the case where an annotator does a lookup for a configuration parameter without specifying a group name. It may also be used as a fallback if the annotator specifies a group that does not exist – see below.

The `searchStrategy` attribute determines the action to be taken when the context is queried for the value of a parameter belonging to a particular configuration group, if that group does not exist or does not contain a value for the requested parameter. There are currently three possible values:

- **none** – there is no fallback; return null if there is no value in the exact group specified by the user.
- **default_fallback** – if there is no value found in the specified group, look in the default group (as defined by the `default` attribute)
- **language_fallback** – this setting allows for a specific use of configuration parameter groups where the groups names correspond to ISO language and country codes (for an example, see below). The fallback sequence is: `<lang>_<country>_<region> -> <lang>_<country> -> <lang> -> <default>`.

Example

```
<configurationParameters defaultGroup="en"
        searchStrategy="language_fallback">
```

```
    <commonParameters>
      <configurationParameter>
        <name>DictionaryFile</name>
        <description>Location of dictionary for this
             language</description>
        <type>String</type>
        <multiValued>false</multiValued>
        <mandatory>false</mandatory>
      </configurationParameter>
    </commonParameters>

    <configurationGroup names="en de en-US"/>

    <configurationGroup names="zh">
      <configurationParameter>
        <name>DBC_Strategy</name>
        <description>Strategy for dealing with double-byte
            characters.</description>
        <type>String</type>
        <multiValued>false</multiValued>
        <mandatory>false</mandatory>
      </configurationParameter>
    </configurationGroup>

  </configurationParameters>
```

In this example, we are declaring a `DictionaryFile` parameter that can have a different value for each of the languages that our TAE supports – English (general), German, U.S. English, and Chinese. For Chinese only, we also declare a `DBC_Strategy` parameter.

We are using the `language_fallback` search strategy, so if an annotator requests the dictionary file for the `en-GB` (British English) group, we will fall back to the more general `en` group.

Since we have defined `en` as the default group, this value will be returned if the context is queried for the `DictionaryFile` parameter without specifying any group name, or if a nonexistent group name is specified.

Configuration Parameter Settings

If no configuration groups were declared, the `<configurationParameterSettings>` element looks like this:

```
  <configurationParameterSettings>
    <nameValuePair>
      <name>[String]</name>
      <value>
        <string>[String]</string>  |
        <integer>[Integer]</integer> |
        <float>[Float]</float> |
```

```
      <boolean>true|false</boolean>  |
      <array> ... </array>
    </value>
  </nameValuePair>

  <nameValuePair>
    ...
  </nameValuePair>

  ...
</configurationParameterSettings>
```

There are zero or more `nameValuePair` elements. Each `nameValuePair` contains a name (which refers to one of the configuration parameters) and a value for that parameter.

The `value` element contains an element that matches the type of the parameter. For single-valued parameters, this is either `<string>`, `<integer>`, `<float>`, or `<boolean>`. For multi-valued parameters, this is an `<array>` element, which then contains zero or more instances of the appropriate type of primitive value, e.g.:

```
<array><string>One</string><string>Two</string></array>
```

If configuration groups were declared, then the `<configurationParameterSettings>` element looks like this:

```
<configurationParameterSettings>
  <settingsForGroup name="[String]">
    [one or more <nameValuePair> elements]
  </settingsForGroup>
  <settingsForGroup name="[String]">
    [one or more <nameValuePair> elements]
  </settingsForGroup>
...
</configurationParameterSettings>
```

where each `<settingsForGroup>` element has a name that matches one of the configuration groups declared under the `<configurationParameters>` element and contains the parameter settings for that group.

Example

Here are the settings that correspond to the parameter declarations in the previous example:

```
<configurationParameterSettings>
  <settingsForGroup name="en">
    <nameValuePair>
      <name>DictionaryFile</name>
      <value><string>resources\English\dictionary.dat></string></value>
```

```
    </nameValuePair>
  </settingsForGroup>

  <settingsForGroup name="en-US">
    <nameValuePair>
      <name>DictionaryFile</name>
      <value><string>resources\English_US\dictionary.dat</string></value>
    </nameValuePair>
  </settingsForGroup>

  <settingsForGroup name="de">
    <nameValuePair>
      <name>DictionaryFile</name>
      <value><string>resources\Deutsch\dictionary.dat</string></value>
    </nameValuePair>
  </settingsForGroup>

  <settingsForGroup name="zh">
    <nameValuePair>
      <name>DictionaryFile</name>
      <value><string>resources\Chinese\dictionary.dat</string></value>
    </nameValuePair>

    <nameValuePair>
      <name>DBC_Strategy</name>
      <value><string>default</string></value>
    </nameValuePair>

  </settingsForGroup>

</configurationParameterSettings>
```

Type System Definition

```
<typeSystemDescription>

  <name> [String] </name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <imports>
    <import ...>
    ...
  </imports>

  <types>
    <typeDescription>
      ...
    </typeDescription>

    ...

  </types>

</typeSystemDescription>
```

A `typeSystemDescription` element defines a type system for an Analysis Engine. The syntax for the element is described in section 20.3 of this chapter.

The recommended usage is to import an external type system, using the import syntax described in section 20.1 of this chapter. For example:

```
<typeSystemDescription>
  <imports>
    <import location="MySharedTypeSystem.xml">
  </imports>
</typeSystemDescription>
```

This allows several AEs to share a single type system definition. The file MySharedTypeSystem.xml would then contain the full type system information, including the name, description, vendor, version, and types.

Type Priority Definition

```
<typePriorities>
  <name> [String] </name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>

  <imports>
    <import ...>
    ...
  </imports>

  <priorityLists>
    <priorityList>
      <type>[TypeName]</type>
      <type>[TypeName]</type>
        ...
    </priorityList>

    ...

  </priorityLists>
</typePriorities>
```

The <typePriorities> element contains zero or more <priorityList> elements; each <priorityList> contains zero or more types. Like a type system, a type priorities definition may also declare a name, description, version, and vendor, and may import other type priorities. The import syntax is described in section 20.1 of this chapter.

Type priority is used when iterating over feature structures in the CAS. For example, if the CAS contains a Sentence annotation and a Paragraph annotation with the same span of text (i.e. a one-sentence paragraph), which annotation should be returned first by an iterator? Probably the Paragraph, since it is conceptually "bigger," but the framework does not know that and must be explicitly told that the Paragraph annotation has priority over the Sentence annotation, like this:

```
<typePriorities>
  <priorityList>
    <type>org.myorg.Paragraph</type>
```

```
    <type>org.myorg.Sentence</type>
  </priorityList>
</typePriorities>
```

All of the <priorityList> elements defined in the descriptor (and in all component descriptors of an aggregate analysis engine descriptor) are merged to produce a single priority list.

Subtypes of types specified here are also ordered, unless overridden by another user-specified type ordering. For example, if you specify type A comes before type B, then subtypes of A will come before subtypes of B, unless there is an overriding specification which declares some subtype of B comes before some subtype of A.

If there are inconsistencies between the priority list (type A declared before type B in one priority list, and type B declared before type A in another), the framework will throw an exception.

User defined indexes may declare if they wish to use the type priority or not; see the next section.

Index Definition

```
<fsIndexCollection>
  <name>[String]</name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>
  <imports>
    <import ...>
    ...
  </imports>
  <fsIndexDescription>
    ...
  </fsIndexDescription>
  <fsIndexDescription>
    ...
  </fsIndexDescription>
  ...
</fsIndexCollection>
```

The `fsIndexCollection` element declares *Feature Structure Indexes*, which define an index that holds feature structures of a given type. Information in the CAS is always accessed through an index. There is a built-in default annotation index declared which can be used to access instances of type Annotation (or its subtypes), but if there is a need for a specialized index it must be declared in this element. See *Chapter 23 **CAS Reference*** for details on FS indexes.

Like type systems and type priorities, an fsIndexCollection can declare a name, description, vendor, and version, and may import other fsIndexCollections. The import syntax is described in section 20.1 of this chapter.

An fsIndexCollection may also define zero or more fsIndexDescription elements, each of which defines a single index. Each fsIndexDescription has the form:

```
<fsIndexDescription>
  <label>[String]</label>
  <typeName>[TypeName]</typeName>
  <kind>sorted|bag|set</kind>
  <keys>
    <fsIndexKey>
      <featureName>[Name]</featureName>
      <comparator>standard|reverse</comparator>
    </fsIndexKey>
    <fsIndexKey>
      <typePriority/>
    </fsIndexKey>
    ...
  </keys>
</fsIndexDescription>
```

The label element defines the name by which applications and annotators refer to this index. The typeName element contains the name of the type that will be contained in this index. This must match one of the type names defined in the <typeSystemDescription>.

There are three possible values for the <kind> of index. Sorted indexes enforce an ordering of feature structures, and may contain duplicates. Bag indexes do not enforce ordering, and also may contain duplicates. Set indexes do not enforce ordering and may not contain duplicates. If the <kind>element is omitted, it will default to sorted, which is the most common type of index.

An index may define one or more *keys*. These keys determine the sort order of the feature structures within a sorted index, and determine equality for set indexes. Bag indexes do not use keys. Keys are ordered by precedence – the first key is evaluated first, and subsequent keys are evaluated only if necessary.

Each key is represented by an fsIndexKey element. Most fsIndexKeys contains a featureName and a comparator. The featureName must match the name of one of the features for the type specified in the <typeName> element for this index. The comparator defines how the features will be compared – a value of standard means that features will be compared using the standard comparison for their data type (e.g. for numerical types, smaller values precede larger values, and for string types, Unicode string comparison is performed). A value of reverse means that features

will be compared using the reverse of the standard comparison (e.g. for numerical types, larger values precede smaller values, etc.). For Set indexes, the comparator direction is ignored – the keys are only used for the equality testing.

Each key used in comparisons must refer to a feature whose range type is String, Float, or Integer.

There is a second type of a key, one which contains only the <typePriority/>. When this key is used, it indicates that Feature Structures will be compared using the type priorities declared in the <typePriorities> section of the descriptor.

Capabilities

```
<capabilities>
  <capability>
    <inputs>
      <type allAnnotatorFeatures="true|false">[TypeName]</type>
      ...
      <feature>[TypeName]:[Name]</feature>
      ...
    </inputs>
    <outputs>
      <type allAnnotatorFeatures="true|false">[TypeName]</type>
      ...
      <feature>[TypeName]:[Name]</feature>
      ...
    </output>
    <languagesSupported>
      <language>[ISO Language ID]</language>
        ...
    </languagesSupported>
    <inputSofas>
      <sofaName>[name]</sofaName>
      ...
    </inputSofas>
    <outputSofas>
      <sofaName>[name]</sofaName>
      ...
    </outputSofas>
  </capability>
  <capability>
    ...
  </capability>
  ...
</capabilities>
```

The capabilities definition is used by the UIMA Framework in several ways, including setting up the Results Specification for process calls, routing control for aggregates based on language, and as part of the Sofa mapping function.

The `capabilities` element contains one or more `capability` elements. Because you can therefore declare multiple capability sets, you can use this to model component behavior that for a given set of inputs, produces a particular set of outputs.

Each `capability` contains `inputs`, `outputs`, `languagesSupported`, `inputSofas`, and `outputSofas`. Inputs and outputs element are required (though they may be empty); `<languagesSupported>`, `<inputSofas>`, and `<outputSofas>` are optional and is used only used for TAEs.

Both inputs and outputs may contain a mixture of type and feature elements.

`<type...>` elements contain the name of one of the types defined in the type system or one of the built in types. Declaring a type as an input means that this component expects instances of this type to be in the CAS when it receives it to process. Declaring a type as an output means that this component creates new instances of this type in the CAS.

There is an optional attribute `allAnnotatorFeatures`, which defaults to false if omitted. The Component Descriptor Editor tool defaults this to true when a new type is added to the list of inputs and/or outputs. When this attribute is true, it specifies that all of the type's features are also declared as input or output. Otherwise, the features that are required as inputs or populated as outputs must be explicitly specified in feature elements.

`<feature...>` elements contain the "fully-qualified" feature name, which is the type name followed by a colon, followed by the feature name, e.g. `org.myorg.tae.TokenAnnotation:lemma`. `<feature...>` elements in the `<inputs>` section must also have a corresponding type declared as an input. In output sections, this is not required. If the type is not specified as an output, but a feature for that type is, this means that existing instances of the type have the values of the specified features updated. Any type mentioned in a `<feature>` element must be either specified as an input or an output or both.

`language` elements contain one of the ISO language identifiers, such as `en` for English, or `en-US` for the United States dialect of English.

The list of language codes can be found here:

http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt

and the country codes here:

http://www.chemie.fu-berlin.de/diverse/doc/ISO_3166.html

<inputSofas> and <outputSofas> declare sofa names used by this component. All Sofa names must be unique within a particular capability set. A Sofa name must be an input or an output, and cannot be both. It is an error to have a Sofa name declared as an input in one capability set, and also have it declared as an output in another capability set.

A <sofaName> is written as a simple Java-style identifier, without any periods in the name.

OperationalProperties

Components can specify specific operational properties that can be useful in deployment. The following are available:

```
<operationalProperties>
  <modifiesCas> true|false </modifiesCas>
  <multipleDeploymentAllowed> true|false </multipleDeploymentAllowed>
</operationalProperties>
```

ModifiesCas, if false, indicates that this component does not modify the CAS. If it is not specified, the default value is true except for CAS Consumer components.

> **Note:** If you wrap one or more CAS Consumers inside an aggregate as the only components, you must explicitly specify in the aggregate the ModifiesCas property as false (assuming the CAS Consumer components take the default here); otherwise the framework will complain about inconsistent settings for these.

multipleDeploymentAllowed, if true, allows the component to be deployed multiple times to increase performance throught scale-out techniques. If it is not specified, the default value is true, except for CAS Consumer and Collection Reader components.

External Resource Dependencies

```
<externalResourceDependencies>
  <externalResourceDependency>
    <key>[String]</key>
    <description>[String] </description>
    <interfaceName>[String]</interfaceName>
    <optional>true|false</optional>
  </externalResourceDependency>
  <externalResourceDependency>
    ...
  </externalResourceDependency>
  ...
```

```
</externalResourceDependencies>
```

A primitive annotator may declare zero or more `<externalResourceDependency>` elements. Each dependency has the following elements:

- `key` – the string by which the annotator code will attempt to access the resource. Must be unique within this annotator.
- `description` – a textual description of the dependency
- `interfaceName` – the fully-qualified name of the Java interface through which the annotator will access the data. This is optional. If not specified, the annotator can only get an InputStream to the data.
- `optional` – whether the resource is optional. If false, an exception will be thrown if no resource is assigned to satisfy this dependency. Defaults to false.

Resource Manager Configuration

```
<resourceManagerConfiguration>
  <name>[String]</name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>
  <imports>
    <import ...>
    ...
  </imports>
  <externalResources>
    <externalResource>
      <name>[String]</name>
      <description>[String]</description>
      <fileResourceSpecifier>
        <fileUrl>[URL]</fileUrl>
      </fileResourceSpecifier>
      <implementationName>[String]</implementationName>
    </externalResource>
    ...
  </externalResources>
  <externalResourceBindings>
    <externalResourceBinding>
      <key>[String]</key>
      <resourceName>[String]</resourceName>
    </externalResourceBinding>
    ...
  </externalResourceBindings>
</resourceManagerConfiguration>
```

This element declares external resources and binds them to annotators' external resource dependencies.

The resourceManagerConfiguration element may optionally contain an import, which allows resource definitions to be stored in a separate (shareable) file. See section 20.2 for details.

The externalResources element contains zero or more externalResource elements, each of which consists of:

- name – the name of the resource. This name is referred to in the bindings (see below). Resource names need to be unique within any Aggregate Analysis Engine or Collection Processing Engine, so the Java-like org.myorg.mycomponent.MyResource syntax is recommended.
- description – English description of the resource
- resource specifier – Declares the location of the resource. There are different possibilities for how this is done (see below).
- implementationName – The fully-qualified name of the Java class that will be instantiated from the resource data. This is optional; if not specified, the resource will be accessible as an input stream to the raw data. If specified, the Java class must implement the interfaceName that is specified in the External Resource Dependency to which it is bound.

One possibility for the resource specifier is a <fileResourceSpecifier>, as shown above. This simply declares a URL to the resource data. This support is built on the Java class URL and its method URL.openStream(); it supports the protocols "file", "http" and "jar" (for referring to files in jars) by default, and you can plug in handlers for other protocols. The URL has to start with file: (or some other protocol). It is relative to either the classpath or the "data path". The data path works like the classpath but can be set programmatically via ResourceManager.setDataPath(). Setting the Java System property uima.datapath also works.

file:com/ibm.d.txt is a relative path; relative paths for resources are resolved using the classpath and / or the datapath. For the file protocol, URLs starting with file:/ or file:/// are absolute. Note that file://com/ibm/d.txt is NOT an absolute path starting with com. The '//' indicates that what follows is a host name. Therefore if you try to use this URL it will complain that it can't connect to the host "com"

Another option is a <fileLanguageResourceSpecifier>, which is intended to support resources, such as dictionaries, that depend on the language of the document being processed. Instead of a single URL, a prefix and suffix are specified, like this:

```
<fileLanguageResourceSpecifier>
  <fileUrlPrefix>file:FileLanguageResource_implTest_data_</fileUrlPrefix>
  <fileUrlSuffix>.dat</fileUrlSuffix>
</fileLanguageResourceSpecifier>
```

The URL of the actual resource is then formed by concatenating the prefix, the language of the document (as an ISO language code, e.g. `en` or `en-US` – see ***Capabilities*** for more information), and the suffix.

The `externalResourceBindings` element declares which resources are bound to which dependencies. Each `externalResourceBinding` consists of:

- `key` – identifies the dependency. For a binding declared in a primitive analysis engine descriptor, this must match the value of the `key` element of one of the `externalResourceDependency` elements. Bindings may also be specified in aggregate analysis engine descriptors, in which case a compound key is used – see section ***External Resource Bindings***.
- `resourceName` – the name of the resource satisfying the dependency. This must match the value of the `name` element of one of the `externalResource` declarations.

A given resource dependency may only be bound to one external resource; one external resource may be bound to many dependencies – to allow resource sharing.

Environment Variable References

In several places throughout the descriptor, it is possible to reference environment variables. In Java, these are actually references to Java system properties. To reference system environment variables from a Java analysis engine you must pass the environment variables into the Java virtual machine by using the `-D` option on the `java` command line.

The syntax for environment variable references is `<envVarRef>[VariableName]</envVarRef>`, where [VariableName] is any valid Java system property name. Environment variable references are valid in the following places:

- The value of a configuration parameter (String-valued parameters only)
- The `<annotatorImplementationName>` element of a primitive TAE descriptor
- The `<name>` element within `<analysisEngineMetaData>`
- Within a `<fileResourceSpecifier>` or `<fileLanguageResourceSpecifier>`

For example, if the value of a configuration parameter were specified as: `<string><envVarRef>TEMP_DIR</envVarRef>/temp.dat</string>`, and the value of the `TEMP_DIR` Java System property were `c:/temp`, then the configuration parameter's value would evaluate to `c:/temp/temp.dat`.

20.4.2 Aggregate Analysis Engine Descriptors

Aggregate Analysis Engines do not contain an annotator, but instead contain one or more component (also called *delegate*) analysis engines.

Aggregate Analysis Engine Descriptors maintain most of the same structure as Primitive Analysis Engine Descriptors. The differences are:

- An Aggregate Analysis Engine Descriptor contains the element `<primitive>false</primitive>` rather than `<primitive>true</primitive>`.
- An Aggregate Analysis Engine Descriptor must not include a `<annotatorImplementationName>` element.
- In place of the `<annotatorImplementationName>`, an Aggregate Analysis Engine Descriptor must have a `<delegateAnalysisEngineSpecifiers>` element. See ***Delegate Analysis Engine Specifiers***.
- Under the analysisEngineMetaData element, an Aggregate Analysis Engine Descriptor requires an additional element -- `<flowConstraints>`. See ***FlowConstraints***.
- An aggregate Analysis Engine Descriptors must not contain a `<typeSystemDescription>` element. The Type System of the Aggregate Analysis Engine is derived by merging the Type System of the Analysis Engines that the aggregate contains.
- Within aggregate Analysis Engine Descriptors, `<configurationParameter>` elements may define `<overrides>`. See ***Configuration Parameter Overrides***.
- External Resource Bindings can bind resources to dependencies declared by any delegate AE within the aggregate. See ***External Resource Bindings***.
- An additional optional element, `<sofaMappings>`, may be included.

Delegate Analysis Engine Specifiers

```
<delegateAnalysisEngineSpecifiers>
  <delegateAnalysisEngine key="[String]">
    <taeDescription>...</taeDescription> |
    <import .../>
  </delegateAnalysisEngine>
  <delegateAnalysisEngine key="[String]">
    ...
  </delegateAnalysisEngine>
  ...
</delegateAnalysisEngineSpecifiers>
```

The `delegateAnalysisEngineSpecifiers` element contains one or more `delegateAnalysisEngine` elements. Each of these must have a unique key, and must contain either:

- A complete `analysisEngineDescription` or `taeDescription` element describing the delegate analysis engine **OR**

- An `import` element giving the name or location of the XML descriptor for the delegate analysis engine (see section 20.1).

The latter is the much more common usage.

FlowConstraints

The order in which delegate Analysis Engines are called within the aggregate Analysis Engine is specified using the `<flowConstraints>` element, which must occur immediately following the `configurationParameterSettings` element.

There are two options for flow constraints -- `<fixedFlow>` or `<capabilityLanguageFlow>`. Each is discussed in a separate section below.

Fixed Flow

```
<flowConstraints>
  <fixedFlow>
    <node>[String]</node>
    <node>[String]</node>
    ...
  </fixedFlow>
</flowConstraints>
```

The `flowConstraints` element must be included immediately following the `configurationParameterSettings` element.

Currently the `flowConstraints` element must contain a `fixedFlow` element. Eventually, other types of flow constraints may be possible.

The `fixedFlow` element contains one or more `node` elements, each of which contains an identifier which must match the key of a delegate analysis engine specified in the `delegateAnalysisEngineSpecifiers` element.

Capability Language Flow

```
<flowConstraints>
  <capabilityLanguageFlow>
    <node>[String]</node>
    <node>[String]</node>
    ...
  </capabilityLanguageFlow>
</flowConstraints>
```

If you use `<capabilityLanguageFlow>`, the delegate Analysis Engines named by the `<node>` elements are called in the given order, except that a delegate Analysis Engine is skipped if any of the following are true (according to that Analysis Engine's declared output capabilities):

- It cannot produce any of the aggregate Analysis Engine's output capabilities for the language of the current document.
- All of the output capabilities have already been produced by an earlier Analysis Engine in the flow.

For example, if two annotators produce `org.myorg.TokenAnnotation` feature structures for the same language, these feature structures will only be produced by the first annotator in the list.

Configuration Parameter Overrides

In an aggregate Analysis Engine Descriptor, each `<configurationParameter>` element should contain an `<overrides>` element, with the following syntax:

```
<overrides>
  <parameter>
    [delegateAnalysisEngineKey]/[parameterName]
  </parameter>
  <parameter>
    [delegateAnalysisEngineKey]/[parameterName]
  </parameter>
  ...
</overrides>
```

Since aggregate Analysis Engines have no code associated with them, the only way in which their configuration parameters can affect their processing is by overriding the parameter values of one or more delegate analysis engines. The `<overrides>` element determines which parameters, in which delegate Analysis Engines, are overridden by this configuration parameter.

For example, consider an aggregate Analysis Engine Descriptor that contains delegate Analysis Engines with keys `annotator1` and `annotator2` (as declared in the <delegateAnalysisEngine> element – see ***Delegate Analysis Engine Specifiers***) and also declares a configuration parameter as follows:

```
<configurationParameter>
  <name>AggregateParam</name>
  <type>String</type>
  <overrides>
    <parameter>annotator1/param1</parameter>
    <parameter>annotator2/param2</parameter>
  </overrides>
</configurationParameter>
```

The value of the `AggregateParam` parameter (whether assigned in the aggregate descriptor or at runtime by an application) will override the value of parameter `param1` in `annotator1` and also override the value of parameter `param2` in `annotator2`. No other parameters will be affected.

For historical reasons only, if an aggregate Analysis Engine descriptor declares a configuration parameter with no explicit overrides, that parameter will override any parameters having the same name within any delegate analysis engine. This usage is strongly discouraged. The UIMA SDK currently supports this usage but logs a warning message to the log file. This support may be dropped in future versions.

External Resource Bindings

Aggregate analysis engine descriptors can declare resource bindings that bind resources to dependencies declared in any of the delegate analysis engines (or their subcomponents, recursively) within that aggregate. This allows resource sharing. Any binding at this level overrides (supersedes) any binding specified by a contained component or their subcomponents, recursively.

For example, consider an aggregate Analysis Engine Descriptor that contains delegate Analysis Engines with keys `annotator1` and `annotator2` (as declared in the `<delegateAnalysisEngine>` element – see ***Delegate Analysis Engine Specifiers***), where `annotator1` declares a resource dependency with key `myResource` and `annotator2` declares a resource dependency with key `someResource`.

Within that aggregate Analysis Engine Descriptor, the following `resourceManagerConfiguration` would bind both of those dependencies to a single external resource file.

```
<resourceManagerConfiguration>
  <externalResources>
    <externalResource>
      <name>ExampleResource</name>
      <fileResourceSpecifier>
        <fileUrl>file:MyResourceFile.dat</fileUrl>
      </fileResourceSpecifier>
    </externalResource>
  </externalResources>
  <externalResourceBindings>
    <externalResourceBinding>
      <key>annotator1/myResource</key>
      <resourceName>ExampleResource</resourceName>
    </externalResourceBinding>
    <externalResourceBinding>
      <key>annotator2/someResource</key>
      <resourceName>ExampleResource</resourceName>
    </externalResourceBinding>
  </externalResourceBindings>
</resourceManagerConfiguration>
```

The syntax for the `externalResources` declaration is exactly the same as described previously. In the resource bindings note the use of the compound keys, e.g. `annotator1/myResource`. This identifies the resource dependency key `myResource`

within the annotator with key annotator1. Compound resource dependencies can be multiple levels deep to handle nested aggregate analysis engines.

Sofa Mappings

Sofa mappings are specified between Sofa names declared in this aggregate descriptor as part of the <capability> section, and the Sofa names declared in the delegate components. For purposes of the mapping, all the declarations of Sofas in any of the capability sets contained within the <capabilities> element are considered together.

```
<sofaMappings>
  <sofaMapping>
    <componentKey>[keyName]</componentKey>
    <componentSofaName>[sofaName]</componentSofaName>
    <aggregateSofaName>[sofaName]</aggregateSofaName>
  </sofaMapping>
  ...
</sofaMappings>
```

The <componentSofaName> may be omitted in the case where the component is not aware of Sofas. In this case, the UIMA framework will arrange for the specified <aggregateSofaName> to be the one visible to the delegate component.

The <componentKey> is the key name for the component as specified in the list of delegate components for this aggregate.

The sofaNames used must be declared as input or output sofas in some capability set.

20.5 Collection Processing Component Descriptors

There are three types of Collection Processing Components – Collection Readers, CAS Initializers, and CAS Consumers. Each type of component has a corresponding descriptor. The structure of these descriptors is very similar to that of primitive Analysis Engine Descriptors.

20.5.1 Collection Reader Descriptors

The basic structure of a Collection Reader descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<collectionReaderDescription
    xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <frameworkImplementation>com.ibm.uima.java</frameworkImplementation>
  <implementationName>[ClassName]</implementationName>
```

```
  <processingResourceMetaData>
    ...
  </processingResourceMetaData>
  <externalResourceDependencies>
   ...
  </externalResourceDependencies>
  <resourceManagerConfiguration>
   ...
  </resourceManagerConfiguration>
</collectionReaderDescription>
```

The `frameworkImplementation` element must always be set to the value `com.ibm.uima.java`.

The `implementationName` element contains the fully-qualified class name of the Collection Reader implementation. This must name a class that implements the `CollectionReader` interface.

The `processingResourceMetaData` element contains essentially the same information as a Primitive Analysis Engine Descriptor's' `analysisEngineMetaData` element:

```
<processingResourceMetaData>
  <name> [String] </name>
  <description>[String]</description>
  <version>[String]</version>
  <vendor>[String]</vendor>
  <configurationParameters>
     ...
  </configurationParameters>
  <configurationParameterSettings>
    ...
  </configurationParameterSettings>
  <typeSystemDescription>
   ...
  </typeSystemDescription>
  <typePriorities>
   ...
  </typePriorities>
  <fsIndexes>
   ...
  </fsIndexes>
  <capabilities>
   ...
  </capabilities>
</processingResourceMetaData>
```

The contents of these elements are the same as that described in *20-266* ***Analysis Engine Metadata,*** with the exception that the capabilities section should not declare any inputs (because the Collection Reader is always the first component to receive the CAS).

The `externalResourceDependencies` and `resourceManagerConfiguration` elements are exactly the same as in the Primitive Analysis Engine Descriptors (see *20-278* ***External Resource Dependencies*** and *20-279* ***Resource Manager Configuration***).

20.5.2 CAS Initializer Descriptors

The basic structure of a CAS Initializer Descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<casInitializerDescription
    xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <frameworkImplementation>com.ibm.uima.java</frameworkImplementation>
  <implementationName>[ClassName] </implementationName>
  <processingResourceMetaData>
    ...
  </processingResourceMetaData>
  <externalResourceDependencies>
    ...
  </externalResourceDependencies>
  <resourceManagerConfiguration>
    ...
  </resourceManagerConfiguration>
</casInitializerDescription>
```

The `frameworkImplementation` element must always be set to the value `com.ibm.uima.java`.

The `implementationName` element contains the fully-qualified class name of the CAS Initializer implementation. This must name a class that implements the `CasInitializer` interface.

The `processingResourceMetaData` element contains essentially the same information as a Primitive Analysis Engine Descriptor's' `analysisEngineMetaData` element, as described in *Section 20-266* ***Analysis Engine Metadata,*** with the exception of some changes to the capabilities section. A CAS Initializer's capabilities element looks like this:

```
<capabilities>
  <capability>
    <outputs>
      <type allAnnotatorFeatures="true|false">[String]</type>
```

```
      <type>[TypeName]</type>
      ...
      <feature>[TypeName]:[Name]</feature>
      ...
    </outputs>
    <outputSofas>
      <sofaName>[name]</sofaName>
      ...
    </outputSofas>
    <mimeTypesSupported>
      <mimeType>[MIME Type]</mimeType>
      ...
    </mimeTypesSupported>
  </capability>
  <capability>
    ...
  </capability>
  ...
</capabilities>
```

The differences between a CAS Initializer's capabilities declaration and a TAE's capabilities declaration are that the CAS Initializer does not declare any input CAS types and features or input Sofas (because it is always the first to operate on a CAS), it doesn't have a language specifier, and that the CAS Initializer may declare a set of MIME types that it supports for its input documents. Examples include: text/plain, text/html, and application/pdf. For a list of MIME types see http://www.iana.org/assignments/media-types/. This information is currently only for users' information, the framework does not use it for anything. This may change in future versions.

The `externalResourceDependencies` and `resourceManagerConfiguration` elements are exactly the same as in the Primitive Analysis Engine Descriptors (see *20-278* ***External Resource Dependencies*** and *20-279* ***Resource Manager Configuration***).

20.5.3 CAS Consumer Descriptors

The basic structure of a CAS Consumer Descriptor is as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<casConsumerDescription
    xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <frameworkImplementation>com.ibm.uima.java</frameworkImplementation>
  <implementationName>[ClassName] </implementationName>
  <processingResourceMetaData>
    ...
  </processingResourceMetaData>
```

```
    <externalResourceDependencies>
      ...
    </externalResourceDependencies>
    <resourceManagerConfiguration>
      ...
    </resourceManagerConfiguration>
  </casConsumerDescription>
```

The `frameworkImplementation` element must always be set to the value `com.ibm.uima.java`.

The `implementationName` element must contain the fully-qualified class name of the CAS Consumer implementation. This must name a class that implements the `CasConsumer` interface.

The `processingResourceMetaData` element contains essentially the same information as a Primitive Analysis Engine Descriptor's `analysisEngineMetaData` element, described in *Section 20-266* ***Analysis Engine Metadata***, except that the CAS Consumer Descriptor's `capabilities` element should not declare outputs or outputSofas (since CAS Consumers do not modify the CAS).

The `externalResourceDependencies` and `resourceManagerConfiguration` elements are exactly the same as in Primitive Analysis Engine Descriptors (see *20-278* ***External Resource Dependencies*** and *20-279* ***Resource Manager Configuration***).

20.6 Service Client Descriptors

Service Client Descriptors specify only a location of a remote service. They are therefore much simpler in structure. In the UIMA SDK, a Service Client Descriptor that refers to a valid Analysis Engine or CAS Consumer service can be used in place of the actual Analysis Engine or CAS Consumer Descriptor. The UIMA SDK will handle the details of calling the remote service. (For details on *deploying* an Analysis Engine or CAS Consumer as a service, see *Chapter 21* ***Collection Processing Engine Descriptor Reference***).

The UIMA SDK is extensible to support different types of remote services. In future versions, there may be different variations of service client descriptors that cater to different types of services. For now, the only type of service client descriptor is the `uriSpecifier`, which supports the SOAP and Vinci protocols.

```
<?xml version="1.0" encoding="UTF-8" ?>
<uriSpecifier xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <resourceType>AnalysisEngine | CasConsumer </resourceType>
  <uri>[URI]</uri>
  <protocol>SOAP | SOAPwithAttachments | Vinci</protocol>
  <timeout>[Integer]</timeout>
```

```
      <parameters>
        <parameter name="VNS_HOST" value="some.internet.ip.name-or-address">
        <parameter name="VNS_PORT" value="9000">
      </parameters>
    </uriSpecifier>
```

The `resourceType` element is required for new descriptors, but is currently allowed to be omitted for backward compatibility. It specifies the type of component (Analysis Engine or CAS Consumer) that is implemented by the service endpoint described by this descriptor.

The `uri` element contains the URI for the web service. (Note that in the case of Vinci, this will be the service name, which is looked up in the Vinci Naming Service.)

The `protocol` element may be set to SOAP, SOAPwithAttachments, or Vinci; other protocols may be added later. These specify the particular data transport format that will be used.

The `timeout` element is optional. If present, it specifies the number of milliseconds to wait for a request to be processed before an exception is thrown. A value of zero or less will wait forever. If no timeout is specified, a default value (currently 60 seconds) will be used.

The parameter element is optional. If present, it specifies the Vinci naming service host and/or port number. If not present, the value used for these comes from parameters passed on the Java command line using the `-DVNS_HOST=<host>` and/or `-DVNS_PORT=<port>` system arguments. If not present, and a system argument is also not present, the values for these default to `localhost` for the `VNS_HOST` and `9000` for the `VNS_PORT`.

For details on how to deploy and call Analysis Engine and CAS Consumer services, see *Section* ***6.6 Working with Analysis Engine and CAS Consumer Services***.

Chapter 21 Collection Processing Engine Descriptor Reference

A UIMA *Collection Processing Engine* (CPE) is a combination of UIMA components assembled to analyze a collection of artifacts. A CPE is an instantiation of the UIMA *Collection Processing Architecture*, which defines the collection processing components, interfaces, and APIs. A CPE is executed by a UIMA framework component called the *Collection Processing Manager* (CPM), which provides a number of services for deploying CPEs, running CPEs, and handling errors.

A CPE can be assembled programmatically within a Java application, or it can be assembled declaratively via a CPE configuration specification, called a CPE Descriptor. This chapter describes the format of the CPE Descriptor.

Details about the CPE, including its function, sub-components, APIs, and related tools, can be found in *Chapter 5* ***Collection Processing Engine Developer's Guide***. Here we briefly summarize the CPE to define terms and provide context for the later sections that describe the CPE Descriptor.

21.1 CPE Overview

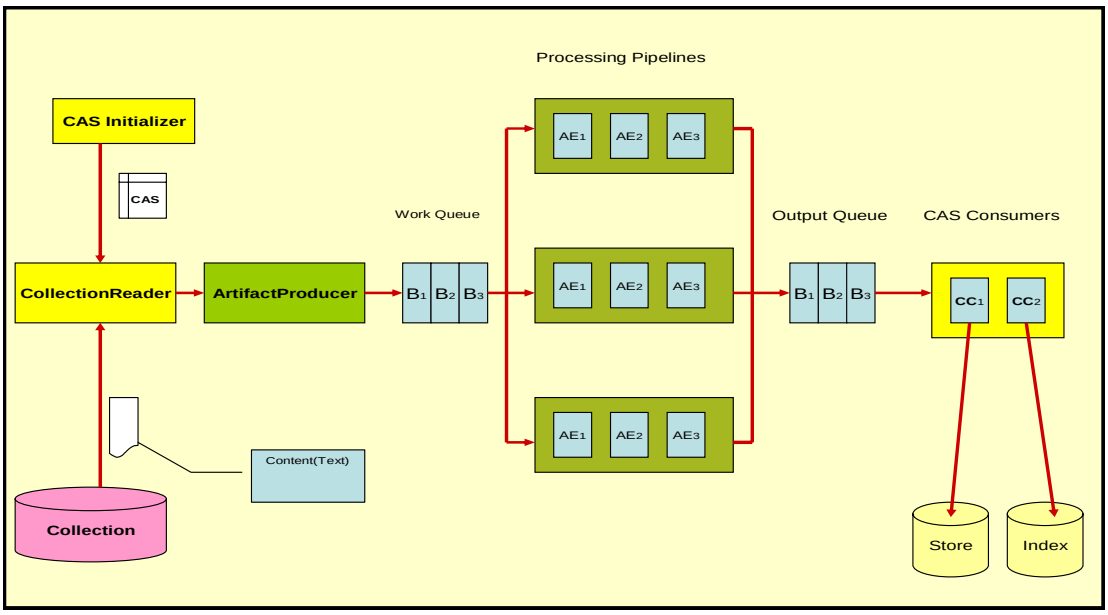


Figure 22 - CPE Runtime Overview

An illustration of the CPE runtime is shown in *Figure 22*. Some of the CPE components, such as the *queues* and *processing pipelines*, are internal to the CPE, but their behavior and deployment may be configured using the CPE Descriptor. Other

CPE components, such as the *Collection Reader* and *CAS Processors*, are defined and configured externally from the CPE and then plugged in to the CPE to create the overall engine. The parts of a CPE are:

Collection Reader –understands the native data collection format and iterates over the collection producing subjects of analysis

CAS Initializer –initializes a CAS with a subject of analysis

Artifact Producer – asynchronously pulls CASes from the Collection Reader, creates batches of CASes and puts them into the work queue

Work Queue – shared queue containing batches of CASes queued by the Artifact Producer for analysis by Analysis Engines

B1-Bn – individual batches containing 1 or more CASes

AE1-AEn – Analysis Engines arranged by a CPE descriptor

Processing Pipelines – each pipeline runs in a separate thread and contains a replicated set of the Analysis Engines running in the defined sequence

Output Queue – holds batches of CASes with analysis results intended for CAS Consumers

CAS Consumers –perform collection level analysis over the CASes and extract analysis results, e.g., creating indexes or databases

21.2 Notation

CPE Descriptors are XML files. This chapter uses an informal notation to specify the syntax of CPE Descriptors.

The notation used in this chapter is:

- An ellipsis (...) inside an element body indicates that the substructure of that element has been omitted (to be described in another section of this chapter). An example of this would be:

```
<collectionReader>
...
</collectionReader>
```

- An ellipsis immediately after an element indicates that the element type may be repeated arbitrarily many times. For example:

  ```
  <parameter>[String]</parameter>
  <parameter>[String]</parameter>
  ...
  ```

  indicates that there may be arbitrarily many parameter elements in this context.

- An ellipsis inside an element means details of the attributes associated with that element are defined later, e.g.:

  ```
  <casProcessor ...>
  ```

- Bracketed expressions (e.g. `[String]`) indicate the type of value that may be used at that location.

- A vertical bar, as in `true|false`, indicates alternatives. This can be applied to literal values, bracketed type names, and elements.

Which elements are optional and which are required is specified in prose, not in the syntax definition.

21.3 Imports

A CPE Descriptor uses the following notation to reference descriptors for other components that are incorporated into the defined CPE:

```
<descriptor>
    <include href="[File]"/>
</descriptor>
```

The `[File]` attribute is a filename for the descriptor of the incorporated component. A fully qualified filename may be provided, or the filename may relative to a directory specified using the `CPM_HOME` variable, e.g.,

```
<descriptor>
    <include href="${CPM_HOME}/desc_dir/descriptor.xml"/>
</descriptor>
```

In this case, the value for the `CPM_HOME` variable must be provided to the CPE by specifying it on the Java command line, e.g.,

```
java -DCPM_HOME="C:/Program Files/IBM/uima/cpm" ...
```

Note that this mechanism for referencing other component descriptor files is different from and in no way related to either of the two import mechanisms described in *Chapter 20*

21.4 CPE Descriptor

A CPE Descriptor consists of information describing the following four main elements.

1. The *Collection Reader*, which is responsible for gathering artifacts and initializing the Common Analysis Structure (CAS) used to support processing in the UIMA collection processing engine.
2. The *CAS Processors* responsible for analyzing individual artifacts, analyzing across artifacts, and extracting analysis results. CAS Processors include *Analysis Engines* and *CAS Consumers*.
3. Operational parameters of the *Collection Processing Manager* (CPM), such as checkpoint frequency and deployment mode.
4. Resource Manager Configuration (optional).

The CPE Descriptor has the following high level skeleton:

```
<?xml version="1.0" encoding="UTF-8"?>
<cpeDescription>
   <collectionReader>
...
   </collectionReader>
   <casProcessors>
...
   </casProcessors>
   <cpeConfig>
...
   </cpeConfig>
   <resourceManagerConfiguration>
...
   </resourceManagerConfiguration>
</cpeDescription>
```

Details of each of the four main elements are described in the sections that follow.

21.4.1 Collection Reader

The `<collectionReader>` section identifies the Collection Reader and optional CAS Initializer that are to be used in the CPE. The Collection Reader is responsible for retrieval of artifacts from a collection outside of the CPE, and the optional CAS Initializer is responsible for initializing the CAS with the artifact.

A Collection Reader may initialize the CAS itself, in which case it does not require a CAS Initializer. This should be clearly specified in the documentation for the Collection Reader. Specifying a CAS Initializer for a Collection Reader that does not make use of a CAS Initializer will not cause an error, but the specified CAS Initializer will not be used.

The complete structure of the <collectionReader> section is:

```
<collectionReader>
  <collectionIterator>
    <descriptor>
      <include href="[File]"/>
    </descriptor>
    <configurationParameterSettings>...</configurationParameterSettings>
    <sofaNameMappings>...</sofaNameMappings>
  </collectionIterator>
  <casInitializer>
    <descriptor>
      <include href="[File]"/>
    </descriptor>
    <configurationParameterSettings>...</configurationParameterSettings>
    <sofaNameMappings>...</sofaNameMappings>
  </casInitializer>
</collectionReader>
```

The <collectionIterator> identifies the descriptor for the Collection Reader, and the <casInitializer> identifies the descriptor for the CAS Initializer. The format and details of the Collection Reader and CAS Initializer descriptors are described in *Chapter 20* . The <configurationParameterSettings> and the <sofaNameMappings> elements are described in the next section.

Error handling for Collection Readers

The CPM will abort if the Collection Reader throws a large number of consecutive exceptions (default = 100). This default can by changed by using the Java initialization parameter `-DMaxCRErrorThreshold xxx`.

21.4.2 CAS Processors

The <casProcessors> section identifies the components that perform the analysis on the input data, including CAS analysis (Analysis Engines) and analysis results extraction (CAS Consumers). The CAS Consumers may also perform collection level analysis, where the analysis is performed (or aggregated) over multiple CASes. The basic structure of the CAS Processors section is:

```
<casProcessors dropCasOnException="true|false" casPoolSize="[Number]"
processingUnitThreadCount="[Number]">
    <casProcessor ...>
        ...
    </casProcessor>
    <casProcessor ...>
        ...
    </casProcessor>
    ...
</casProcessors>
```

The <casProcessors> section has two mandatory attributes and one optional attribute that configure the characteristics of the CAS Processor flow in the CPE. The first mandatory attribute is a casPoolSize, which defines the fixed number of CAS instances that the CPM will create and use during processing. All CAS instances are maintained in a CAS Pool with a check-in and check-out access. Each CAS is checked-out from the CAS Pool by the Collection Reader and initialized with an initial subject of analysis. The CAS is checked-in into the CAS Pool when it is completely processed, at the end of the processing chain. A larger CAS Pool size will result in more memory being used by the CPM. CAS objects can be large and care should be taken to determine the optimum size of the CAS Pool, weighing memory tradeoffs with performance.

The second mandatory <casProcessors> attribute is processingUnitThreadCount, which specifies the number of replicated *Processing Pipelines*. Each Processing Pipeline runs in its own thread. The CPM takes CASes from the work queue and submits each CAS to one of the Processing Pipelines for analysis. A Processing Pipeline contains one or more Analysis Engines invoked in a given sequence. If more than one Processing Pipeline is specified, the CPM replicates instances of each Analysis Engine defined in the CPE descriptor. Each Processing Pipeline thread runs independently, consuming CASes from work queue and depositing CASes with analysis results onto the output queue. On multiprocessor machines, multiple Processing Pipelines can run in parallel, improving overall throughput of the CPM.

> **Note:** The number of Processing Pipelines should be equal to or greater than CAS Pool size.

> **Note:** Elements in the pipeline (each represented by a <casProcessor> element) may indicate that they do not permit multiple deployment in their Analysis Engine descriptor. If so, even though multiple pipelines are being used, all CASes passing through the pipelines will be routed through one instance of these marked Engines.

The final, optional, <casProcessors> attribute is dropCasOnException. It defines a policy that determines what happens with the CAS when an exception happens during processing. If the value of this attribute is set to true and an exception happens, the CPM will notify all registered listeners of the exception (see ***Using Listeners*** on page 5-109), clear the CAS and check the CAS back into the CAS Pool so that it can be re-used. The presumption is that an exception may leave the CAS in an inconsistent state and therefore that CAS should not be allowed to move through the processing chain. When this attribute is omitted the CPM's default is the same as specifying dropCasOnException="false".

Specifying an Individual CAS Processor

The CAS Processors that make up the Processing Pipeline and the CAS Consumer pipeline are specified with the <casProcessor> entity, which appears within the

<casProcessors> entity. It may appear multiple times, once for each CAS Processor specified for this CPE.

The order of the <casProcessor> entities with the <casProcessors> section specifies the order in which the CAS Processors will run. Although CAS Consumers are usually put at the end of the pipeline, they need not be. Also, Aggregate Analysis Engines may include CAS Consumers.

The overall format of the <casProcessor> entity is:

```
<casProcessor deployment="local|remote|integrated" name="[String]" >
    <descriptor>
        <include href=[File]/>
    </descriptor>
    <configurationParameterSettings>...</configurationParameterSettings>
    <sofaNameMappings>...</sofaNameMappings>
    <runInSeparateProcess>...</runInSeparateProcess>
    <deploymentParameters>...</deploymentParameters>
    <filter/>
    <errorHandling>...</errorHandling>
    <checkpoint batch="Number"/>
</casProcessor>
```

The <casProcessor> element has two mandatory attributes, deployment and name. The mandatory name attribute specifies a unique string identifying the CAS Processor.

The mandatory deployment attribute specifies the CAS Processor deployment mode. Currently, three deployment options are supported:

- integrated – indicates *integrated* deployment of the CAS Processor. The CPM deploys and collocates the CAS Processor in the same process space as the CPM. This type of deployment is recommended to increase the performance of the CPE. However, it is NOT recommended to deploy annotators containing JNI this way. Such CAS Processors may cause a fatal exception and force the JVM to exit without cleanup (bringing down the CPM). Any UIMA SDK compliant pure Java CAS Processors may be safely deployed this way.
- remote – indicates *non-managed* deployment of the CAS Processor. The CAS Processor descriptor referenced in the <descriptor> element must be a Vinci *Service Client Descriptor*, which identifies a remotely deployed CAS Processor service (see *Section 6.6* ***Working with Analysis Engine and CAS Consumer Services***). The CPM assumes that the CAS Processor is already running as a remote service and will connect to it using the URI provided in the client service descriptor. The lifecycle of a remotely deployed CAS Processor is not managed by the CPM, so appropriate infrastructure should be in place to start/restart such CAS Processors when necessary. This deployment provides fault isolation and is implementation (i.e., programming language) neutral.

- `local` – indicates *managed* deployment of the CAS Processor. The CAS Processor descriptor referenced in the `<descriptor>` element must be a Vinci *Service Deployment Descriptor*, which configures a CAS Processor for deployment as a Vinci service (see *Section 6.6* ***Working with Analysis Engine and CAS Consumer Services***). The CPM deploys the CAS Processor in a separate process and manages the life cycle (start/stop) of the CAS Processor. Communication between the CPM and the CAS Processor is done with Vinci. When the CPM completes processing, the process containing the CAS Processor is terminated. This deployment mode insulates the CPM from the CAS Processor, creating a more robust deployment at the cost of a small communication overhead. On multiprocessor machines, the separate processes may run concurrently and improve overall throughput.

A number of elements may appear within the `<casProcessor>` element.

<descriptor> Element

The `<descriptor>` element is mandatory. It identifies the descriptor for the referenced CAS Processor using the syntax described in Section 20.2 above.

- For *`remote`* CAS Processors, the referenced descriptor must be a Vinci *Service Client Descriptor*, which identifies a remotely deployed CAS Processor service.
- For *local* CAS Processors, the referenced descriptor must be a Vinci *Service Deployment Descriptor*.
- For *integrated* CAS Processors, the referenced descriptor must be an Analysis Engine Descriptor (primitive or aggregate).

See Section 6.6 **Working with Analysis Engine and CAS Consumer Services** for more information on creating these descriptors and deploying services.

<configurationParameterSettings> Element

This element provides a way to override the contained Analysis Engine's parameters settings. Any entry specified here must already be defined; values specified replace the corresponding values for each parameter. **For Cas Processors, this mechanism is only available when they are deployed in "integrated" mode.** For Collection Readers and Initializers, it always is available.

The content of this element is identical to the component descriptor for specifying parameters (in the case where no parameter groups are specified), except that the names for the primitive types have a "_p" suffixed to them: string_p, integer_p, float_p. Here is an example:

```
<configurationParameterSettings>
  <nameValuePair>
    <name>CivilianTitles</name>
```

```
    <value>
      <array>
        <string_p>Mr.</string_p>
        <string_p>Ms.</string_p>
        <string_p>Mrs.</string_p>
        <string_p>Dr.</string_p>
      </array>
    </value>
  </nameValuePair>
  ...
</configurationParameterSettings>
```

<sofaNameMappings> Element

This optional element provides a mapping from defined Sofa names in the component, or the default Sofa name (if the component does not declare any Sofa names). The form of this element is:

```
<sofaNameMappings>
  <sofaNameMapping cpeSofaName="a_CPE_name"
                   componentSofaName="a_component_Name"/>
  ...
</sofaNameMappings>
```

There can be any number of `<sofaNameMapping>` elements contained in the `<sofaNameMappings>` element. The `componentSofaName` attribute is optional; leave it out to specify a mapping for the default text sofa - that is, for components which are not aware of Sofas.

<runInSeparateProcess> Element

The `<runInSeparateProcess>` element is mandatory for `local` CAS Processors, but should not appear for `remote` or `integrated` CAS Processors. It enables the CPM to create external processes using the provided runtime environment. Applications launched this way communicate with the CPM using the Vinci protocol and connectivity is enabled by a local instance of the VNS that the CPM manages. Since communication is based on Vinci, the application need not be implemented in Java. Any language for which Vinci provides support may be used to create an application, and the CPM will seamlessly communicate with it. The overall structure of this element is:

```
<runInSeparateProcess>
    <exec dir="[String]" executable="[String]">
        <env key="[String]" value ="[String]"/>
        ...
        <arg>[String]</arg>
        ...
    </exec>
</runInSeparateProcess>
```

The <exec> element provides information about how to execute the referenced CAS Processor. Two attributes are defined for the <exec> element. The dir attribute is currently not used – it is reserved for future functionality. The executable attribute specifies the actual Vinci service executable that will be run by the CPM, e.g., java, a batch script, an application (.exe), etc. The executable must be specified with a fully qualified path, or be found in the PATH of the CPM.

The <exec> element has two elements within it that define parameters used to construct the command line for executing the CAS Processor. These elements must be listed in the order in which they should be defined for the CAS Processor.

The optional <env> element is used to set an environment variable. The variable key will be set to value. For example,

```
<env key="CLASSPATH" value ="C:\Java\lib"/>
```

will set the environment variable CLASSPATH to the value C:\Java\lib. The <env> element may be repeated to set multiple environment variables. All of the key/value pairs will be added to the environment by the CPM prior to launching the executable.

Note: *The CPM actually adds ALL system environment variables when it launches the program. It queries the Operating System for its current system variables and one by one adds them to the program's process configuration.*

The <arg> element is used to specify arbitrary string arguments that will appear on the command line when the CPM runs the command specified in the executable attribute.

For example, the following would be used to invoke the UIMA Java implementation of the Vinci service wrapper on a Java CAS Processor:

```
<runInSeparateProcess>
    <exec executable="java">
        <arg>-DVNS_HOST=localhost</arg>
        <arg>-DVNS_PORT=9099</arg>
        <arg>com.ibm.uima.reference_impl.analysis_engine.service.
vinci.VinciAnalysisEngineService_impl</arg>
        <arg>C:\uima\desc\deployCasProcessor.xml</arg>
    </exec>
<runInSeparateProcess>
```

This will cause the CPM to run the following command line when starting the CAS Processor:

```
java -DVNS_HOST=localhost -DVNS_PORT=9099
com.ibm.uima.reference_impl.analysis_engine.service.vinci.VinciAnalysisEngi
neService_impl C:\uima\desc\deployCasProcessor.xml
```

The first argument specifies that the Vinci Naming Service is running on the `localhost`. The second argument specifies that the Vinci Naming Service port number is `9099`. The third argument identifies the UIMA implementation of the Vinci service wrapper. This class contains the `main` method that will execute. That main method in turn takes a single argument – the filename for the CAS Processor service deployment descriptor. Thus the last argument identifies the Vinci service deployment descriptor file for the CAS Processor. Since this is the same descriptor file specified earlier in the `<descriptor>` element, the string `${descriptor}` can be used to refer to the descriptor, e.g.:

```
<arg>${descriptor}</arg>
```

The CPM will expand this out to the service deployment descriptor file referenced in the `<descriptor>` element.

<deploymentParameters> Element

The `<deploymentParameters>` element defines a number of deployment parameters that control how the CPM will interact with the CAS Processor. This element has the following overall form:

```
<deploymentParameters>
    <parameter name="[String]" value="..." type="string|integer" />
    ...
</deploymentParameters>
```

The `name` attribute identifies the parameter, the `value` attribute specifies the value that will be assigned to the parameter, and the `type` attribute indicates the type of the parameter, either `string` or `integer`. The available parameters include:

- `vnsHost` – (Deprecated) string parameter specifying the VNS host, e.g., `localhost` for local CAS Processors, host name or IP address of VNS host for remote CAS Processors. This parameter is deprecated; use the parameter specification instead inside the Vinci *Service Client Descriptor*, if needed. It is ignored for integrated and local deployments. If present, for remote deployments, it specifies the VNS Host to use, unless that is specified in the Vinci *Service Client Descriptor*.
- `vnsPort` – (Deprecated) integer parameter specifying the VNS port number. This parameter is deprecated; use the parameter specification instead inside the Vinci *Service Client Descriptor*, if needed. It is ignored for integrated and local deployments. If present, for remote deployments, it specifies the VNS Port number to use, unless that is specified in the Vinci *Service Client Descriptor*.

- service-access – string parameter whose value must be "exclusive", if present. This parameter is only effective for remote deployments. It modifies the Vinci service connections to be preallocated and dedicated, one service instance per pipe-line. It is only relevant for non-Integrated deployement modes. If there are fewer services instances that are available (and alive – responding to a "ping" request) than there are pipelines, the number of pipelines (the number of concurrent threads) is reduced to match the number of available instances. If not specified, the VNS is queried each time a service is needed, and a "random" instance is assigned from the pool of available instances. If a services dies during processing, the CPM will use its normal error handling procedures to attempt to reconnect. The number of attempts is specified in the CPE descriptor for each Cas Processor using the `<maxConsecutiveRestarts value="10" action="kill-pipeline" waitTimeBetweenRetries="50" />` xml element. The "value" attribute is the number of reconnection tries; the "action" says what to do if the retries exceed the limit. The "kill-pipeline" action stops the pipeline that was associated with the failing service (other pipelines will continue to work). The CAS in process within a killed pipeline will be dropped. These events are communicated to the application using the normal event listener mechanism. The `waitTimeBetweenRetries` says how many milliseconds to wait inbetween attempts to reconnect.

For example, the following parameters might be used with a CAS Processor deployed in local mode:

```
<deploymentParameters>
  <parameter name="service-access" value="exclusive" type="string"/>
</deploymentParameters>
```

<filter> Element

The <filter> element is a required element but currently should be left empty. This element is reserved for future use.

<errorHandling> Element

The mandatory `<errorHandling>` element defines error and restart policies for the CAS Processor. Each CAS Processor may define different actions in the event of errors and restarts. The CPM monitors and logs errant behaviors and attempts to recover the component based on the policies specified in this element.

There are two kinds of faults.

1. One kind only occurs with non-integrated CAS Processors – this fault is either a timeout attempting to launch or connect to the non-integrated component, or some other kind of connection related exception (for instance, the network connection might timeout or get reset).

2. The other kind happens when the CAS Processor component (an Annotator, for example) throws any kind of exception. This kind may occur with any kind of deployment, integrated or not.

The <errorHandling> has specifications for each of these kinds of faults. The format of this element is:

```
<errorHandling>
  <maxConsecutiveRestarts action="continue|disable|terminate"
                          value="[Number]"/>
  <errorRateThreshold action="continue|disable|terminate" value="[Rate]"/>
  <timeout max="[Number]"/>
</errorHandling>
```

The mandatory `<maxConsecutiveRestarts>` element applies only to faults of the first kind, and therefore, only applies to non-integrated deployments. If such a fault occurs, a retry is attempted, up to `value="[Number]"` of times. This retry resets the connection (if one was made) and attempts to reconnect and perhaps re-launch (see below for details). The original CAS (not a partially updated one) is sent to the CAS Processor as part of the retry, once the deployed component has been successfully restarted or reconnected to.

The `action` attribute specifies the action to take when the threshold specified by the `value="[Number]"` is exceeded. The possible actions are:

- `continue` – skip any further processing for this CAS by this CAS Processor, and pass the CAS to the next CAS Processor in the Pipeline.

  Note: The "restart" action is done, because it is needed for the next CAS.

  Note: If the `dropCasOnException="true"`, the CPM will NOT pass the CAS to the next CAS Processor in the chain. Instead, the CPM will abort processing of this CAS, release the CAS back to the CAS Pool and will process the next CAS in the queue.

  Note: The counter counting the restarts toward the threshold is only reset after a CAS is successfully processed.
- `disable` – the current CAS is handled just as in the `continue` case, but in addition, the CAS Processor is marked so that its *process()* method will not be called again (i.e., it will be "skipped" for future CASes)
- `terminate` – the CPM will terminate all processing and exit

The definition of an error for the `<maxConsecutiveRestarts>` element differs slightly for each of the three CAS Processor deployment modes:

local	Local CAS Processors experience two general error types: • launch errors – errors associated with launching a process • processing errors – errors associated with sending Vinci commands to the process A launch error is defined by a failure of the process to successfully

	register with the local VNS within a default time window. The current timeout is 15 minutes. Multiple local CAS Processors are launched sequentially, with a subsequent processor launched immediately after its previous processor successfully registers with the VNS. A processing error is detected if a connection to the CAS Processor is lost or if the processing time exceeds a specified timeout value. For local CAS Processors, the <maxConsecutiveRestarts> element specifies the number of consecutive attempts made to launch the CAS Processor at CPM startup or after the CPM has lost a connection to the CAS Processor.
remote	For remote CAS Processors, the <maxConsecutiveRestarts> element applies to errors from sending Vinci commands. An error is detected if a connection to the CAS Processor is lost, or if the processing time exceeds the timeout value specified in the <timeout> element (see below).
integrated	Although mandatory, the <maxConsecutiveRestarts> element is NOT used for integrated CAS Processors, because Integrated CAS Processors are not re-instantiated/restarted on exceptions. This setting is ignored by the CPM for Integrated CAS Processors but it is required. Future version of the CPM will make this element mandatory for remote and local CAS Processors only.

The mandatory `<errorRateThreshold>` element is used for all faults – both those above, and exceptions thrown by the CAS Processor itself. It specifies the number of retries for exceptions thrown by the CAS Processor itself, a maximum error rate, and the corresponding action to take when this rate is exceeded. The `value` attribute specifies the error rate in terms of errors per sample size in the form "`N/M`", where `N` is the number of errors and `M` is the sample size, defined in terms of the number of documents.

The first number is used also to indicate the maximum number of retries. If this number is less than the `<maxConsecutiveRestarts value="[Number]">`, it will override, reducing the number of "restarts" attempted. A retry is done only if the `dropCasOnException` is false. If it is set to true, no retry occurs, but the error is counted.

When the number of counted errors exceeds the sample size, an action specified by the `action` attribute is taken. The possible actions and their meaning are the same as described above for the `<maxConsecutiveRestarts>` element:

```
continue
disable
terminate
```

The dropCasOnException="true" attribute of the <casProcessors> element modifies the action taken for continue and disable, in the same manner as above. For example:

```
<errorRateThreshold value="3/1000" action="disable" />
```

specifies that each error thrown by the CAS Processor itself will be retried up to 3 times (if dropCasOnException is false) and the CAS Processor will be disabled if the error rate exceeds 3 errors in 1000 documents.

If a document causes an error and the error rate threshold for the CAS Processor is not exceeded, the CPM increments the CAS Processor's error count and retries processing that document (if dropCasOnException is false). The retry means that the CPM calls the CAS Processor's process() method again, passing in as an argument the same CAS that previously caused an exception.

> **Note:** The CPM does not attempt to rollback any partial changes that may have been applied to the CAS in the previous process() call.

Errors are accumulated across documents. For example, assume the error rate threshold is 3/1000. The same document may fail three times before finally succeeding on the fourth try, but the error count is now 3. If one more error occurs within the current sample of 1000 documents, the error rate threshold will be exceeded and the specified action will be taken. If no more errors occur within the current sample, the error counter is reset to 0 for the next sample of 1000 documents.

The <timeout> element is a mandatory element. Although mandatory for all CAS Processors, this element is only relevant for local and remote CAS Processors. For integrated CAS Processors, this element is ignored. In the current CPM implementation the integrated CAS Processor process() method is not subject to timeouts.

The max attribute specifies the maximum amount of time in milliseconds the CPM will wait for a process() method to complete When exceeded, the CPM will generate an exception and will treat this as an error subject to the threshold defined in the <errorRateThreshold> element above, including doing retries.

Retry action taken on a timeout

The action taken depends on whether the CAS Processor is local (managed) or remote (unmanaged). Local CAS Processors (which are services) are killed and restarted, and a new connection to them is established. For remote CAS Processors, the connection to them is dropped, and a new connection is reestablished (which may actually connect to a different instance of the remote services, if it has multiple instances).

<checkpoint> Element

The <checkpoint> element is an optional element used to improve the performance of CAS Consumers. It has a single attribute, batch, which specifies the number of CASes in a batch, e.g.:

```
<checkpoint batch="1000">
```

sets the batch size to 1000 CASes. The batch size is the interval used to mark a point in processing requiring special handling. The CAS Processor's batchProcessComplete() method will be called by the CPM when this mark is reached so that the processor can take appropriate action. This mark could be used as a mechanism to buffer up results in CAS Consumers and perform time-consuming operations, such as check-pointing, that should not be done on a per-document basis.

21.4.3 CPE Operational Parameters

The parameters for configuring the overall CPE and CPM are specified in the <cpeConfig> section. The overall format of this section is:

```
<cpeConfig>
    <startAt>[NumberOrID]</startAt>
    <numToProcess>[Number]</numToProcess>
    <outputQueue dequeueTimeout="[Number]" queueClass="[ClassName]" />
    <checkpoint file="[File]" time="[Number]" batch="[Number]"/>
    <timerImpl>[ClassName]</timerImpl>
    <deployAs>vinciService|interactive|immediate|single-threaded
    </deployAs>
</cpeConfig>
```

This section of the CPE descriptor allows for defining the starting entity, the number of entities to process, a checkpoint file and frequency, a pluggable timer, an optional output queue implementation, and finally a mode of operation. The mode of operation determines how the CPM interacts with users and other systems.

The <startAt> element is an optional argument. It defines the starting entity in the collection at which the CPM should start processing.

The implementation in the CPM passes the this argument to the Collection Reader as the value of the parameter "startNumber". The CPM does not do anything else with this parameter; in particular, the CPM has no ability to skip to a specific document - that function, if available, is only provided by a particular Collection Reader implementation.

If the <startAt> element is used, the Collection Reader descriptor must define a single-valued configuration parameter with the name startNumber. It can declare

this value to be of any type; the value passed in this XML element must be convertible to that type.

A typical use is to declare this to be an integer type, and to pass the sequential document number where processing should start. An alternative implementation might take a specific document ID; the collection reader could search through its collection until it reaches this ID and then start there.

This parameter will only make sense if the particular collection reader is implemented to use the `startNumber` configuration parameter.

The `<numToProcess>` element is an optional element. It specifies the total number of entities to process. Use -1 to indicate ALL. If not defined, the number of entities to process will be taken from the Collection Reader configuration. If present, this value overrides the Collection Reader configuration.

The `<outputQueue>` element is an optional element. It enables plugging in a custom implementation for the Output Queue. When omitted, the CPM will use a default output queue that is based on First-in First-out (FIFO) model.

The UIMA SDK provides a second implementation for the Output Queue that can be plugged in to the CPM, named `"com.ibm.uima.reference_impl.collection.cpm.engine.SequencedQueue"`.

This implementation supports handling very large documents that are split into "chunks"; it provides a delivery mechanism that insures the sequential order of the chunks using information carried in the CAS metadata. This metadata, which is required for this implementation to work correctly, must be added as an instance of a Feature Structure of type `com.ibm.es.tt.DocumentMetaData` and referred to by an additional feature named `esDocumentMetaData` in the special instance of `uima.tcas.DocumentAnnotation` that is associated with the TCAS. This is usually done by the Collection Reader; the instance contains the following features:

1. sequenceNumber – [Number] the sequential number of a chunk, starting at 1. If not a chunk (i.e. complete document), the value should be 0.
2. documentId – [Number] current document id. Chunks belonging to the same document have identical document id.
3. isCompleted – [Number] 1 if the chunk is the last in a sequence, 0 otherwise.
4. url – [String] document url
5. throttleID – [String] special attribute currently used by OmniFind

This implementation of a sequenced queue supports proper sequencing of CASes in CPM deployments that use document chunking. Chunking is a technique of splitting large documents into pieces to reduce overall memory consumption.

Chunking does not depend on the number of CASes in the CAS Pool. It works equally well with one or more CASes in the CAS Pool. Each chunk is packaged in a separate CAS and placed in the Work Queue. If the CAS Pool is depleted, the CollectionReader thread is suspended until a CAS is released back to the pool by the processing threads. A document may be split into 1, 2, 3 or more chunks that are analyzed independently. In order to reconstruct the document correctly, the CAS Consumer can depend on receiving the chunks in the same sequential order that the chunks were "produced", when this sequenced queue implementation is used. To plug in this sequenced queue to the CPM use the following specification:

```
<outputQueue dequeueTimeout="100000"
queueClass="com.ibm.uima.reference_impl.collection.cpm.engine.SequencedQueue"/>
```

where the mandatory `queueClass` attribute defines the name of the class and the second mandatory attribute, `dequeueTimeout` specifies the maximum number of milliseconds to wait for the expected chunk.

> **Note:** The value for this timeout must be carefully determined to avoid excessive occurrences of timeouts. Typically, the size of a chunk and the type of analysis being done are the most important factors when deciding on the value for the timeout. The larger the chunk and the more complicated analysis, the more time it takes for the chunk to go from source to sink.

If the chunk doesn't arrive in the configured time window, the entire document is presumed to be invalid and the CAS is dropped from further processing. This action occurs regardless of any other error action specification. The SequencedQueue invalidate the document, adding the offending document's metadata to a local cache of invalid documents.

If the time out occurs, the CPM notifies all registered listeners (see ***Using Listeners*** on page 5-109) by calling entityProcessComplete(). As part of this call, the SequencedQueue will pass null instead of a CAS as the first argument, and a special exception – CPMChunkTimeoutException. The reason for passing null as the first argument is because the time out occurs due to the fact that the chunk has not been received in the configured timeout window, so there is no CAS available when the timeout event occurs.

The CPMChunkTimeoutException object exposes an API that allows the listener to retrieve the offending document id as well as the other metadata attributes as defined above. These attributes are part of each chunk's metadata and are added by the Collection Reader.

Each chunk that SequencedQueue works on is subjected to a test to determine if the chunk belongs to an invalid document. This test checks the chunk's metadata against the data in the local cache. If there is a match, the chunk is dropped. This

check is only performed for chunks and complete documents are not subject to this check.

If there is an exception during the processing of a chunk, the CPM sends a notification to all registered listeners. The notification includes the CAS and an exception. When the listener notification is completed, the CPM also sends separate notifications, containing the CAS, to the Artifact Producer and the SequencedQueue. The intent is to stop adding new chunks to the Work Queue that belong to an "invalid" document and also to deal with chunks that are en-route, being processed by the processing threads.

In response to the notification, the Artifact Producer will drop and release back to the CAS Pool all CASes that belong to an "invalid" document. Currently, there is no support in the CollectionReader's API to tell it to stop generating chunks. The CollectionReader keeps producing the chunks but the Artifact Producer immediately drops/releases them to the CAS Pool. Before the CAS is released back to the CAS Pool, the Artifact Producer sends notification to all registered listeners. This notification includes the CAS and an exception – SkipCasException.

In response to the notification of an exception involving a chunk, the SequencedQueue retrieves from the CAS the metadata and adds it to its local cache of "invalid" documents. All chunks de-queued from the OutputQueue and belonging to "invalid" documents will be dropped and released back to the CAS Pool. Before dropping the CAS, the CPM sends notification to all registered listeners. The notification includes the CAS and SkipCasException.

The `<checkpoint>` element is an optional element. It specifies a CPE checkpoint file, checkpoint frequency, and strategy for checkpoints (time or count based). At checkpoint time, the CPM saves status information and statistics to the checkpoint file. The checkpoint file is specified in the `file` attribute, which has the same form as the `href` attribute of the `<include>` element described in Section 20.2 . The `time` attribute indicates that a checkpoint should be taken every `[Number]` seconds, and the `batch` attribute indicates that a checkpoint should be taken every `[Number]` batches.

The `<timerImpl>` element is optional. It is used to identify a custom timer plug-in class to generate time stamps during the CPM execution. The value of the element is a Java class name.

The `<deployAs>` element indicates the type of CPM deployment. Valid contents for this element include:

1. `vinciService` – Vinci service exposing APIs for stop, pause, resume, and getStats
2. `interactive` – provide command line menus (start, stop, pause, resume)

3. `immediate` – run the CPM without menus or a service API
4. `single-threaded` – run the CPM in a single threaded mode. In this mode, the Collection Reader, the Processing Pipeline, and the CAS Consumer Pipeline are all running in one thread without the work queue and the output queue.

21.4.4 Resource Manager Configuration

External resource bindings for the CPE may optionally be specified in an element:

```
<resourceManagerConfiguration href="..."/>
```

For an introduction to external resources, refer to sections 4.5.4 , on page 4-83.

In the `resourceManagerConfiguration` element, the value of the href attribute refers to another file that contains definitions and bindings for the external resources used by the CPE. The format of this file is the same as the XML snippet on page 20-285. For example, in a CPE containing an aggregate analysis engine with two annotators, and a CAS Consumer, the following resource manager configuration file would bind external resource dependencies in all three components to the same physical resource:

```
<resourceManagerConfiguration>
  <!-- Declare Resource -->
  <externalResources>
    <externalResource>
      <name>ExampleResource</name>
      <fileResourceSpecifier>
        <fileUrl>file:MyResourceFile.dat</fileUrl>
      </fileResourceSpecifier>
    </externalResource>
  </externalResources>
  <!-- Bind component resource dependencies to ExampleResource -->
  <externalResourceBindings>
    <externalResourceBinding>
      <key>MyAE/annotator1/myResourceKey</key>
      <resourceName>ExampleResource</resourceName>
    </externalResourceBinding>
    <externalResourceBinding>
      <key>MyAE/annotator2/someResourceKey</key>
      <resourceName>ExampleResource</resourceName>
    </externalResourceBinding>
    <externalResourceBinding>
      <key>MyCasConsumer/otherResourceKey</key>
      <resourceName>ExampleResource</resourceName>
    </externalResourceBinding>
  </externalResourceBindings>
</resourceManagerConfiguration>
```

In this example, MyAE and MyCasConsumer are the names of the Analysis Engine and CAS Consumer, as specified by the name attributes of the CPE's <casProcessor> elements. annotator1 and annotator2 are the annotator keys specified within the Aggregate AE Descriptor, and myResourceKey, someResourceKey, and otherResourceKey are the keys of the resource dependencies declared in the individual annotator and CAS Consumer descriptors.

21.4.5 Example CPE Descriptor

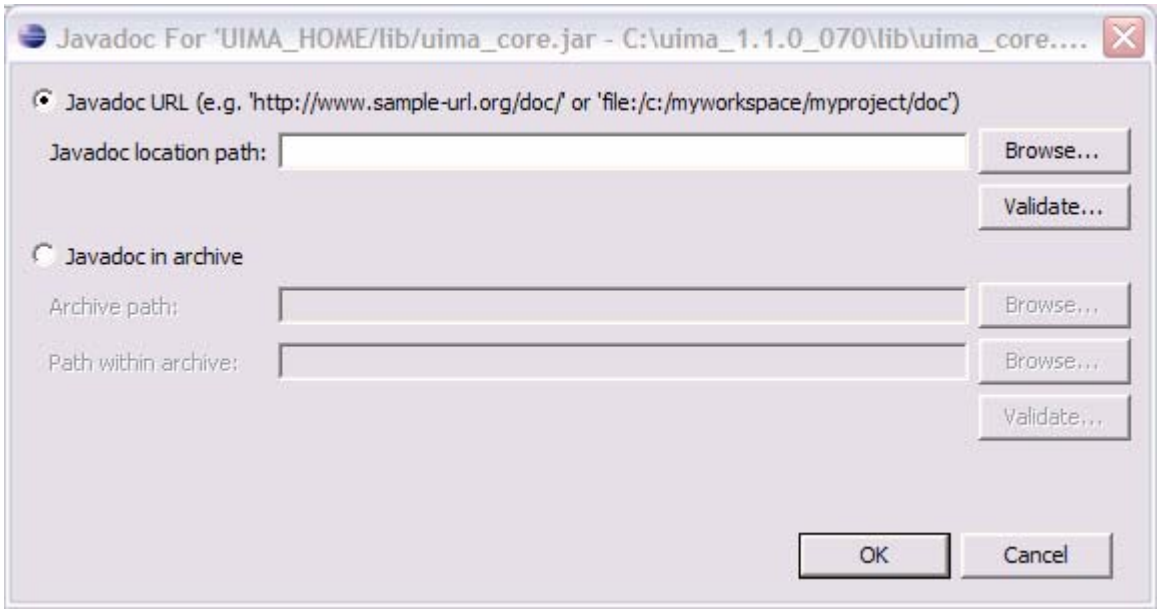
```
<?xml version="1.0" encoding="UTF-8"?>
<cpeDescription>
  <collectionReader>
    <collectionIterator>
      <descriptor>
        <include href="C:\Program
Files\IBM\uima\docs\examples\descriptors\collection_reader\XMLFileCollectio
nReader.xml"/>
      </descriptor>
    </collectionIterator>
    <casInitializer>
      <descriptor>
        <include href="C:\Program
Files\IBM\uima\docs\examples\descriptors\cas_initializer\XMLCasInitializer.
xml"/>
      </descriptor>
    </casInitializer>
  </collectionReader>
  <casProcessors dropCasOnException="true" casPoolSize="1"
processingUnitThreadCount="1">
    <casProcessor deployment="integrated" name="Aggregate TAE - Name
Recognizer and Person Title Annotator">
      <descriptor>
        <include href="C:\Program
Files\IBM\uima\docs\examples\descriptors\analysis_engine\NamesAndPersonTitl
es_TAE.xml"/>
      </descriptor>
      <deploymentParameters/>
      <filter/>
      <errorHandling>
        <errorRateThreshold action="terminate" value="100/1000"/>
                <maxConsecutiveRestarts action="terminate" value="30"/>
                <timeout max="100000"/>
      </errorHandling>
      <checkpoint batch="1"/>
    </casProcessor>
    <casProcessor deployment="integrated" name="Annotation Printer">
      <descriptor>
        <include href="C:\Program
Files\IBM\uima\docs\examples\descriptors\cas_consumer\AnnotationPrinter.xml
"/>
      </descriptor>
```

```
      <deploymentParameters/>
      <filter/>
      <errorHandling>
        <errorRateThreshold action="terminate" value="100/1000"/>
        <maxConsecutiveRestarts action="terminate" value="30"/>
        <timeout max="100000"/>
      </errorHandling>
      <checkpoint batch="1"/>
    </casProcessor>
  </casProcessors>
  <cpeConfig>
    <numToProcess>1</numToProcess>
    <deployAs>immediate</deployAs>
    <checkpoint file="" time="3000"/>
    <timerImpl/>
  </cpeConfig>
</cpeDescription>
```

Chapter 22 JavaDocs

The details of all the public APIs for UIMA are contained in the API JavaDocs. These are located in the docs/api directory; the top level to open in your browser is called index.html.

In recent releases, Eclipse supports the ability to attach the JavaDocs to your project. To use this facility, open a project which is referring to the UIMA APIs in its class path, and open the project properties.



Once you do this, Eclipse can show you JavaDocs for UIMA APIs as you work. To see the JavaDoc for a UIMA API, highlight the API class or method and press shift-F2, or use the menu Navigate -> OpenExternalJavaDoc. This will open the JavaDoc for the selected item in a browser.

Chapter 23 CAS Reference

The CAS (Common Analysis System) is the part of the Unstructured Information Management Architecture (UIMA) that is concerned with creating and handling the data that annotators manipulate.

Java users typically use the JCas (Java interface to the CAS) when manipulating objects in the CAS. This chapter describes an alternative interface to the CAS which allows discovery and specification of types and features at run time. It is recommended for use when the using code cannot know ahead of time the type system it will be dealing with.

The CAS is a general-purpose data container, and can be used for many kinds of data. There is a special instantiation of the CAS, the TCAS, that is considered to be a "view" of the CAS. There can be multiple TCASes for one CAS, corresponding to multiple subjects of analysis (Sofas) – see ***Sofas and TCAS Views*** on page 7-162. The TCAS provides a small number of extensions to the CAS to simplify common analysis tasks associated with having a subject of analysis and annotations. In this chapter, we will usually not make the distinction between the two, unless the distinction is critical for correct API usage.

23.1.1 JavaDocs

The subdirectory docs/api contains the documentation details of all the classes, methods, and constants for the APIs discussed here. Please refer to this for details on the methods, classes and constants, specifically in the packages com.ibm.uima.cas.*.

23.1.2 CAS Overview

There are three main parts to the CAS: the type system, data creation and manipulation, and indexing. We will start with a brief description of these components.

The type system

The type system specifies what kind of data you will be able to manipulate in your annotators. The type system defines two kinds of entities, types and features. Types are arranged in an inheritance tree and define the kinds of entities (objects) you can manipulate in the CAS. Features optionally specify slots within a type. The correspondence to Java is to equate a CAS Type to a Java Class, and the CAS Features to fields within the type. A critical difference is that CAS types have no methods; they are just data structures with named slots (features). These slots can

have as values primitive things like integers, floating point numbers, and strings, and they also can hold references to other instances of objects in the CAS. We call instances of the data structures declared by the type system "feature structures" (not to be confused with "features"). Feature structures are similar to the many variants of record structures found in computer science.[4].

Each CAS Type defines a supertype; it is a subtype of that supertype. This means that any features that the supertype defines are features of the subtype; in other words, it inherits its supertype's features. Only single inheritance is supported; a type's feature set is the union of all of the features in its supertype hierarchy. There is a built-in type called uima.cas.TOP; this is the top, root node of the inheritance tree. It defines no features.

The values that can be stored in features are either built-in primitive values or references to other feature structures. The primitive values are integers, floats, and strings; the official names of these are uima.cas.Integer, uima.cas.Float, and uima.cas.String. The integers correspond to Java "int" (32 bit) values; the floats to Java float (single precision) values, and the strings to Java string values (Unicode strings). The CAS also defines other basic built-in types for arrays of these, plus arrays of references to other objects, called uima.cas.IntegerArray, uima.cas.FloatArray, uima.cas.StringArray, and uima.cas.FSArray.

The TCAS extension of the CAS defines a built-in type called `uima.tcas.Annotation` which inherits from `uima.cas.TOP`. There are two features defined by this type, called `begin` and `end`, both of which are integer valued.

Types and features are defined in XML descriptors. At runtime, annotators are passed an instance of a CAS, TCAS or JCas, depending on the kind of annotator it is, and other factors. See ***Do UIMA Components Receive a CAS or a TCAS?*** On page 7-164 for more details. You can use this object to access all of the metadata about the defined type system in use, as well as information about the CAS indexes.

Creating, accessing and manipulating data

Using the non JCas runtime APIs to access the CAS is a two step process. In step one you query the CAS's type system to obtain type and feature objects corresponding to the types and features. This has to be done once for each CAS type system. Then you use these retrieved type and feature objects in calls to the CAS APIs to create feature structures, set and get feature values from particular feature structures, and add and removed feature structures from indexes.

[4] The name "feature structure" comes from terminology used in linguistics.

Creating and using indexes

Instances of feature structures can be added to CAS indexes. These indexes provide the only way for other annotators to locate existing data in the CAS. The only way for an annotator to use data that another annotator has created is to get feature structures the first annotator created, out of the CAS using an index. If you want the data you create to be visible to other annotators, you must index it.

Indexes are named; they are used to index one specific CAS type (including its subtypes). To access an index, you minimally need to know its name. The CAS provides an index repository which you can query for indexes. Once you have a handle to an index, you can get information about the feature structures in the index, the size of the index, as well as an iterator over the feature structures.

Indexes are defined in the XML descriptor metadata for the application. The indexes are grouped into repositories. Each view of the CAS (corresponding to each TCAS) has a separate repository, containing all the indexes. When you obtain an index, it is always from a particular TCAS view (or the base CAS). When you index an item, it is always added to all indexes where it belongs, within just one repository. You can specify different repositories to use; a given instance may be indexed in more than one repository.

Iterators allow you to enumerate the feature structures in an index. The iterators are a subclass of the normal Java Iterator class; they add methods to allow both forward and backward traversal, and you can set the iterator to arbitrary points in the index.

Indexes are created by specifying them in the annotator's or aggregate's resource descriptor. An index specification includes its name, the CAS type being indexed, the kind of index it is, and an (optional) ordering relation on the feature structures to be indexed. Feature structures need to be explicitly added to the index repository by a method call. Feature structures that are not indexed will not be visible to other annotators, (unless they are located via being referenced by some other feature of another feature structure, which is indexed).

The TCAS extension defines a standard, built-in annotation index, called AnnotationIndex, which indexes the `uima.tcas.Annotation` type. All feature structures of type `uima.tcas.Annotation` or its subtypes are automatically indexed with this built-in index.

The ordering relation used by this index is to first order by the value of the "begin" features (in ascending order) and then by the value of the "end" feature (in descending order). This ordering insures that longer annotations starting at the same spot come before shorter ones. For Subjects of Analysis other than Text, this may not be an appropriate index.

23.2 Built-in CAS Types

The CAS has two kinds of built-in types – primitive and non-primitive. The primitive types are:

```
uima.cas.Float
uima.cas.Integer
uima.cas.String
```

The `uima.cas.String` type can be sub-typed to create sets of allowed values; see *Chapter 20* These types can be used to specify the range of a feature. They act like Strings, but have additional checking to insure the setting of values into them conforms to one of the allowed values. Note that these sub-types cannot be used as a supertype for another type definition; only `uima.cas.String` can be sub-typed.

The non-primitive types exist in a type hierarchy; the top of the hierarchy is the type

```
uima.cas.TOP
```

All other non-primitive types inherit from some supertype.

There are 4 built-in array types. These arrays have a size specified when they are created; the size is fixed at creation time; they are named:

```
uima.cas.FloatArray
uima.cas.IntegerArray
uima.cas.StringArray
uima.cas.FSArray
```

The `uima.cas.FSArray` type is an array whose elements are arbitrary other feature structures (instances of non-primitive types).

There are 2 built-in types associated with the artifact being analyzed:

```
uima.tcas.Annotation
uima.tcas.DocumentAnnotation
```

The `Annotation` type defines 2 features, taking `uima.cas.Integer` values, called `begin` and `end`. The `begin` feature typically identifies the start of a span of text the annotation covers; the `end` feature identifies the end. The values refer to character offsets; the starting index is 0. An annotation of the word "CAS" in a text "CAS Reference" would have a start index of 0, and an end index of 3; the difference between end and start is the length of the span the annotation refers to.

Annotations are always with respect to some Sofa (Subject of Analysis – see **7.2** 7-162).

> **Note:** Artifacts which are not text strings may have a different interpretation of the meaning of begin and end.

The DocumentAnnotation type has one special instance, created when a TCAS is generated. It is a subtype of the annotation type, and the built-in definition defines one feature, language, which is a string indicating the language of the document in the TCAS. The value of this language feature is used by the system to control flow among annotators, allowing the flow to skip over annotators that don't process particular languages. Users may extend this type by adding additional features to it, using the XML Descriptor element for defining a type.

Each TCAS view of a CAS has a different associated instance of the DocumentAnnotation type.

The instance of this type can be accessed in two ways: using the getDocumentationAnnotation method on a TCAS object, or using the getDocumentationAnnotationFs method on a JCas object. There is a deprecated JCas method with the same method name as the method used with the TCAS object (i.e., without the trailing "Fs"), but it is not safe to use in an environment where class loaders are being used. The getDocumentationAnnotationFs method returns an item of type TOP, which you need to cast to DocumentAnnotation. The JCas model for this is the Java type DocumentAnnotation in the package com.ibm.uima.jcas.tcas.

There are also built-in types supporting lists, in the style of Lisp. Their use is not recommended, however, as this is not a particularly efficient representation. The implementation is type specific; there are different list building objects for each of the primitive types, plus one for general feature structures. Here are the type names:

```
uima.cas.FloatList
uima.cas.IntegerList
uima.cas.StringList
uima.cas.FSList

uima.cas.EmptyFloatList
uima.cas.EmptyIntegerList
uima.cas.EmptyStringList
uima.cas.EmptyFSList

uima.cas.NonEmptyFloatList
uima.cas.NonEmptyIntegerList
uima.cas.NonEmptyStringList
uima.cas.NonEmptyFSList
```

For each primitive type, there is a base type, for instance, uima.cas.FloatList. For each of these, there are two subtypes, corresponding to a non-empty element, and a marker that serves to indicate the end of the list, or an empty list. The non-empty types define two features – head and tail. The head feature holds the particular value for that part of the list. The tail refers to the next list object (either a non-empty one or the empty version to indicate the end of the list).

There are no other built-in types. Users are free to define their own type systems, building upon these types.

23.3 Accessing the type system

When using the JCas, the type system declaration is converted to Java class definitions; these allow strongly typed references to the CAS data objects. When you are designing an application which can't use this approach, perhaps because it is a general tool that is built to handle unknown (at compile-time) type systems, you use the CAS (not JCas) APIs, described here.

These APIs presume as a starting point a reference to an existing CAS, or a CAS's type system. This CAS reference can be something returned by utilities that create new CASes, or is a parameter passed to an annotator's process method. The CAS's type system can be obtained by calling the getTypeSystem method on the CAS object.

Non-JCas annotators implement an additional method, `typeSystemInit`, which is called by the UIMA framework before the annotator's process method. This method, implemented by the annotator writer, is passed a reference to the CAS's type system metadata. The method typically uses the type system APIs to obtain type and feature objects corresponding to all the types and features the annotator will be using in its process method. This initialization step should not be done during an annotator's initialize method since the type system can change after the initialize method is called; it should not be done during the process method, since this is presumably work that is identical for each incoming document, and so should be performed only when the type system changes (which will be a rare event). The UIMA framework guarantees it will call the `typeSystemInit` method of an annotator whenever the type system changes, before calling the annotator's `process` method.

The initialization done by `typeSystemInit` is done by the UIMA framework when you use the JCas APIs; you only need to provide a `typeSystemInit` method, as described here, when you are not using the JCas approach.

23.3.1 TypeSystemPrinter example

Here is a code fragment that, given a CAS Type System, will print a list of all types.

```
// Get all type names from the type system
// and print them to stdout.
private void listTypes1(TypeSystem ts) {
  // Get an iterator over types
  Iterator typeIterator = ts.getTypeIterator();
  Type t;
  System.out.println("Types in the type system:");
  while (typeIterator.hasNext()) {
    // Retrieve a type...
```

```
      t = (Type) typeIterator.next();
      // ...and print its name.
      System.out.println(t.getName());
    }
    System.out.println();
  }
```

This method is passed the type system as a parameter. (The type system is passed as a parameter to your annotator's typeSystemInit method by the UIMA framework, or you can obtain it from a CAS reference using the method `getTypeSystem`.) From the type system, we can get an iterator over all known types. If you run this against a CAS created with no additional user-defined types, we should see something like this on the console:

Types in the type system:

```
uima.cas.TOP
uima.cas.Integer
uima.cas.Float
uima.cas.String
uima.cas.ArrayBase
uima.cas.FSArray
uima.cas.IntegerArray
uima.cas.FloatArray
uima.cas.StringArray
uima.cas.ListBase
uima.cas.IntegerList
uima.cas.EmptyIntegerList
uima.cas.NonEmptyIntegerList
uima.cas.FloatList
uima.cas.EmptyFloatList
uima.cas.NonEmptyFloatList
uima.cas.StringList
uima.cas.EmptyStringList
uima.cas.NonEmptyStringList
uima.tcas.Annotation
```

Here we only see the built-in types; more would show up if the type system had user-defined types. Note that some of these types are not directly creatable – they are types used by the framework in the type hierarchy (e.g. uima.cas.ArrayBase).

CAS type names include a name-space prefix. The components of a type name are separated by the dot (.). A type name component must start with a Unicode letter, followed by an arbitrary sequence of letters, digits and the underscore (_). By convention, the last component of a type name starts with an uppercase letter, the rest start with a lowercase letter.

Listing the type names is mildly useful, but it would be even better if we could see the inheritance relation between the types. The following code prints the inheritance tree in indented format.

```
  private static final int INDENT = 2;
  private void listTypes2(TypeSystem ts) {
    // Get the root of the inheritance tree.
    Type top = ts.getTopType();
    // Recursively print the tree.
    printInheritanceTree(ts,top, 0);
  }


private void printInheritanceTree(TypeSystem ts, Type type, int level) {
    indent(level); // Print indentation.
    System.out.println(type.getName());
    // Get a vector of the immediate subtypes.
    Vector subTypes =
      ts.getDirectlySubsumedTypes(type);
    ++level; // Increase the indentation level.
    for (int i = 0; i < subTypes.size(); i++) {
      // Print the subtypes.
      printInheritanceTree(ts, (Type) subTypes.get(i), level);
    }
  }
  // A simple, inefficient indenter
  private void indent(int level) {
    int spaces = level * INDENT;
    for (int i = 0; i < spaces; i++) {
      System.out.print(" ");
    }
  }
```

This example shows that you can traverse the type hierarchy by starting at the top with TypeSystem.getTopType and by retrieving subtypes with `TypeSystem.getDirectlySubsumedTypes`.

The JavaDocs also have APIs that allow you to access the features, as well as what the allowed value type is for that feature. Here is sample code which prints out all the features of all the types, together with the allowed value types (the feature "range"). Each feature has a "domain" which is the type where it is defined, as well as a "range".

```
  private void listFeatures2(TypeSystem ts) {
    Iterator featureIterator = ts.getFeatures();
    Feature f;
    System.out.println("Features in the type system:");
    while (featureIterator.hasNext()) {
      f = (Feature) featureIterator.next();
      System.out.println(
        f.getShortName() + ": " +
        f.getDomain() + " -> " + f.getRange());
    }
    System.out.println();
  }
```

We can ask a feature object for its domain (the type it is defined on) and its range (the type of the value of the feature). The terminology derives from the fact that features can be viewed as functions on subspaces of the object space.

23.3.2 Using the CAS APIs to create and modify feature structures

Assume a type system declaration that defines two types: Entity and Person. Entity has no features defined within it but inherits from uima.tcas.Annotation – so it has the begin and end features. Person is, in turn, a subtype of Entity, and adds firstName and lastName features. CAS type systems are declaratively specified using XML; the format of this XML is described in *Chapter 20* .

```
<!-- Type System Definition -->
<typeSystemDescription>
  <types>
    <typeDescription>
      <name>com.xyz.proj.Entity</name>
      <description />
      <supertypeName>uima.tcas.Annotation</supertypeName>
    </typeDescription>
    <typeDescription>
      <name>Person</name>
      <description />
      <supertypeName>com.xyz.proj.Entity </supertypeName>
      <features>
        <featureDescription>
          <name>firstName</name>
          <description />
          <rangeTypeName>uima.cas.String</rangeTypeName>
        </featureDescription>
        <featureDescription>
          <name>lastName</name>
          <description />
          <rangeTypeName>uima.cas.String</rangeTypeName>
        </featureDescription>
      </features>
    </typeDescription>
  </types>
</typeSystemDescription>
```

To use these types in annotator code, the CAS APIs require "handles" which are references to the specific type and feature objects corresponding to each type and feature (note that these are not required when using the JCas APIs to the CAS). These are setup by CAS TypeSystem API calls that are passed the official external names of the types and features. The CAS APIs provide string constants for the official names of all the built-in types and features that you might use.

```
/** Entity type name constant. */
public static final String ENTITY_TYPE_NAME = "com.xyz.proj.Entity";
```

```
/** Person type name constant. */
public static final String PERSON_TYPE_NAME = "com. xyz.proj.Person";

/** First name feature name constant. */
public static final String FIRST_NAME_FEAT_NAME = "firstName";

/** Last name feature name constant. */
public static final String LAST_NAME_FEAT_NAME = "lastName";
```

We define type and feature member variables; these will hold the values of the type and feature objects needed by the CAS APIs.

```
// Type system object variables
private Type entityType;
private Type personType;
private Feature firstNameFeature;
private Feature lastNameFeature;
private Type stringType;
```

The type system does not consider it to be an error if we ask for something that is not known, it simply returns null; therefore the code checks for this.

```
// Get a type object corresponding to a name.
// If it doesn't exist, throw an exception.
private Type initType(String typeName)
  throws AnnotatorInitializationException {
  Type type = ts.getType(typeName);
  if (type == null) {
    throw new AnnotatorInitializationException(
      AnnotatorInitializationException.TYPE_NOT_FOUND,
      new Object[] { this.getClass().getName(), typeName });
  }
  return type;
}
We add similar code for retrieving feature objects.
// Get a feature object from a name and a type object.
// If it doesn't exist, throw an exception.
private Feature initFeature(String featName, Type type)
  throws AnnotatorInitializationException {
  Feature feat = type.getFeatureByBaseName(featName);
  if (feat == null) {
    throw new AnnotatorInitializationException(
      AnnotatorInitializationException.FEATURE_NOT_FOUND,
      new Object[] { this.getClass().getName(), featName });
  }
  return feat;
}
```

Using these two functions, code for initializing the type system described above would be:

```
public void typeSystemInit(TypeSystem aTypeSystem)
  throws AnnotatorConfigurationException,
         AnnotatorInitializationException
{
```

```
    this.typeSystem = aTypeSystem;
    // Set type system member variables.
    this.entityType = initType(ENTITY_TYPE_NAME);
    this.personType = initType(PERSON_TYPE_NAME);
    this.firstNameFeature =
      initFeature(FIRST_NAME_FEAT_NAME, personType);
    this.lastNameFeature =
      initFeature(LAST_NAME_FEAT_NAME, personType);
    this.stringType = initType(CAS.TYPE_NAME_STRING);
  }
```

Note that we initialize the string type by using a type name constant from the CAS.

23.4 Creating feature structures

To create feature structures in JCas, we use the Java "new" operator. In the CAS, we use one of several different API methods on the CAS object, depending on which of the 5 basic kinds of feature structures we are creating (a plain feature structure, or an instance of the built-in StringArray, FloatArray, IntegerArray, or FSArray).

If a TCAS is provided, it has a method to create an instance of a `uima.tcas.Annotation`, setting the begin and end values.

Once a feature structure is created, it needs to be added to the CAS indexes (unless it will be accessed via some reference from another accessible feature structure). The API to add a feature structure to the CAS indexes is found by starting with the CAS object, getting a reference to the index repository, and then calling the index repository's addFS method. (There is also a removeFS method to remove a feature structure from the index). Assuming aCAS holds a reference to a CAS, and token holds a reference to a newly created feature structure, here's the code to add that feature structure to all the relevant CAS indexes:

```
// Add the token to the index repository.
aCAS.getIndexRepository().addFS(token);
```

23.5 Accessing or modifying features of feature structures

Values of individual features for a feature structure can be set or referenced, using a set of methods that depend on the type of value that feature is declared to have. There are methods getIntValue, getFloatValue, getStringValue, and getFeatureValue (which means to get a value which in turn is a reference to a feature structure). There are corresponding "setter" methods, as well. These are methods on the feature structure object, and take as arguments the feature object retrieved earlier in the typeSystemInit method.

Using the previous example, with the type system initialized with type personType and feature lastNameFeature, here's a sample code fragment that gets and sets that feature:

```
// Assume aPerson is a variable holding an object of type Person
// get the lastNameFeature value from the feature structure
String lastName = aPerson.getStringValue(lastNameFeature);
// set the lastNameFeature value
aPerson.setStringValue(lastNameFeature, newStringValueForLastName);
```

The getters and setters for each of the primitive types are defined in the JavaDocs as methods of the FeatureStructure interface.

23.6 Indexes and Iterators

Each CAS can have many indexes associated with it. Each index is represented by an instance of the type com.ibm.uima.cas.FSIndex. You use the object com.ibm.uima.cas.FSIndexRepository, accessible via a method on the basic CAS object, to retrieve instances of the index object. There are methods that let you select the index by name, or by name and type. Since each index is already associated with a type, the passing of an additional type parameter is valid only if the type passed in is the same type or a subtype of the one declared in the index specification for this index. If you pass in a subtype, the returned FSIndex object refers to an index that will return only items belonging to that subtype (or subtypes of that subtype).

The returned FSIndex objects are used, in turn, to create iterators. The iterators created can be used like common Java iterators, to sequentially retrieve items indexed. If the index represents a sorted index, the items are returned in a sorted order, where the sort order is specified in the XML index definition. This XML is part of the Component Descriptor, see *Chapter 20* .

Feature structures should not be added to or removed from indexes while iterating over them; results are undefined if this is done.

23.6.1 Iterators

Iterators are objects of class `com.ibm.uima.cas.FSIterator`. This class implements the normal Java iterator methods, plus additional ones that allow moving both forwards and backwards.

23.6.2 Special iterators for Annotation types

The built-in index over the `uima.tcas.Annotation` type named `"AnnotationIndex"` has additional capabilities. To use them, you first get a reference to this built-in index using either the `getAnnotationIndex` method on a TCAS object, or by asking the `FSIndexRepository` object for an index having the particular name "AnnotationIndex". You then must cast the returned FSIndex object to `AnnotationIndex`. Here's an example showing the cast:

```
AnnotationIndex idx = (AnnotationIndex) aTCAS.getAnnotationIndex();
```

This object can be used to produce several additional kinds of iterators. It can produce unambiguous iterators; these skip over elements until it finds one where the start position of the next annotation is equal to or greater than the end position of the previously returned annotation.

It can also produce several kinds of subiterators; these are iterators whose annotations fall within the span of another annotation. This kind of iterator can also have the unambiguous property, if desired. It also can be "strict" or not; strict means that the returned annotation lies completely within the span of the controlling annotation. Non-strict only implies that the beginning of the returned annotation falls within the span of the controlling annotation.

There is also a method which produces an `AnnotationTree` object, which contains nodes representing the results of doing a strict, unambiguous subiterator over the span of some controlling annotation. For more details, please refer to the JavaDocs for the `com.ibm.uima.cas.text` package.

23.6.3 Constraints and Filtered iterators

There is a set of API calls that build constraint objects. These objects can be used directly to test if a particular feature structure matches (satisfies) the constraint, or they can be passed to the createFilteredIterator method to create an iterator that skips over instances which fail to satisfy the constraint.

It is possible to specify a feature value located by following a chain of references starting from the feature structure being tested. Here's a scenario to explore this concept. Let's suppose you have the following type system (namespaces are omitted for clarity):

Token, having a feature PartOfSpeech which holds a reference to another type (POS)

POS (a type with many subtypes, each representing a different part of speech)

Noun (a subtype of POS)

ProperName (a subtype of Noun), having a feature Class which holds an integer value encoding some information about the proper noun.

If you want to filter Token instances, such that only those tokens get through which are proper names of class 3 (for example), you would need a test that started with a Token instance, followed its PartOfSpeech reference to another instance (the

ProperName instance) and then tested the Class feature of that instance for a value equal to 3.

To support this, the filtering approach has components that specify tests, and components that specify "paths". The tests that can be done include testing references to type instances to see if they are instances of some type or its subtypes; this is done with a FSTypeConstraint constraint. Other tests check for equality or, for numeric values, ranges.

Each test may be combined with a path – to get to the value to test. Tests that start from a feature structure instance can be combined with and and or connectors. The JavaDocs for these are in the package com.ibm.uima.cas in the classes that end in Constraint, plus the classes ConstraintFactory, FeaturePath and CAS. Here's an example; assume the variable cas holds a reference to a CAS instance.

```
// Start by getting the constraint factory from the CAS.
ConstraintFactory cf = cas.getConstraintFactory();


// To specify a path to an item to test, you start by
// creating an empty path.
FeaturePath path = cas.createFeaturePath();


// Add POS feature to path, creating one-element path.
path.addFeature(posFeat);
// You can extend the chain arbitrarily by adding additional
// features.


// Create a new type constraint.
// Type constraints will check that structures
// they match against have a type at least as specific
// as the type specified in the constraint.
FSTypeConstraint nounConstraint = cf.createTypeConstraint();
// Set the type (by default it is TOP).
// This succeeds if the type being tested by this constraint
// is nounType or a subtype of nounType.
nounConstraint.add(nounType);


// Embed the noun constraint under the pos path.
// This means, associate the test with the path, so it tests the
// proper value.
// The result is a test which will
// match a feature structure that has a posFeat defined
// which has a value which is an instance of a nounType or
// one of its subtypes.
```

```
FSMatchConstraint embeddedNoun = cf.embedConstraint(path, nounConstraint);

// Create a type constraint for token (or a subtype of it)
FSTypeConstraint tokenConstraint = cf.createTypeConstraint();
// Set the type.
tokenConstraint.add(tokenType);

// Create the final constraint by conjoining the two constraints.
FSMatchConstraint nounTokenCons = cf.and(nounConstraint, tokenConstraint);

// Create a filtered iterator from some annotation iterator.
FSIterator it = cas.createFilteredIterator(annotIt, nounTokenCons);
```

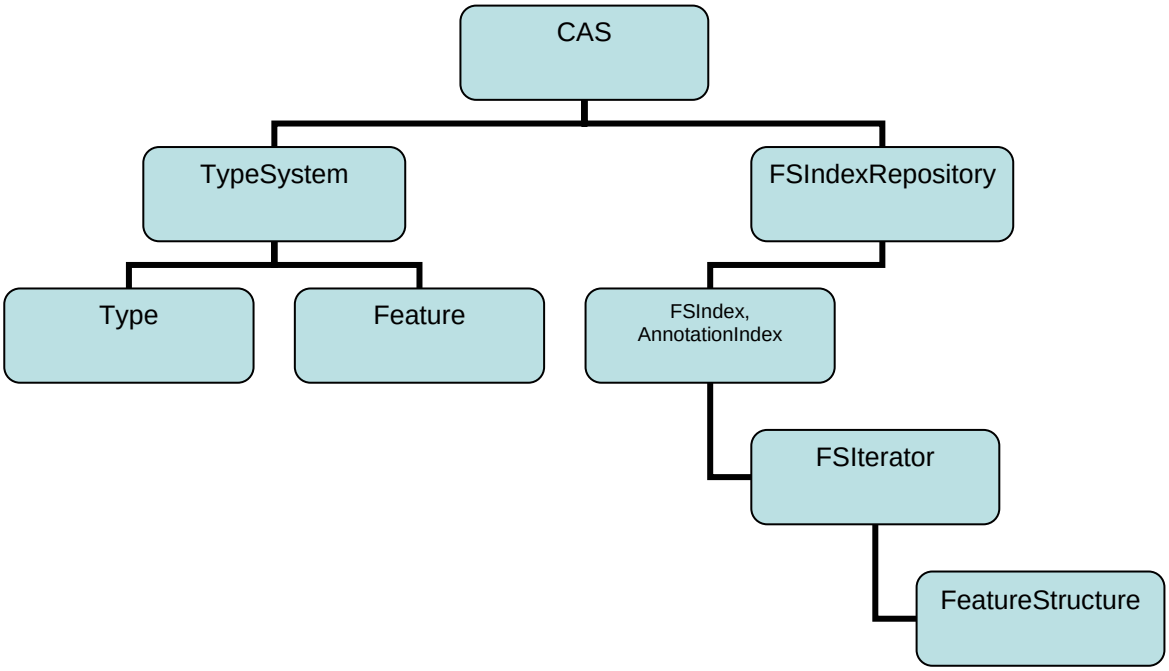
23.7 The CAS APIs – a guide to the JavaDocs

The CAS APIs are organized into 3 Java packages: cas, cas.impl, and cas.text. Most of the APIs described here are in the cas package. The cas.impl package contains classes used in serializing and deserializing (reading and writing to external strings) the XCAS form of the CAS (XCAS is an XML serialization of the CAS). The XCAS form is used for transporting the CAS among local and remote annotators, or for storing the CAS in permanent storage. The cas.text contains the APIs that extend the CAS to support artifact (including "text") analysis.

23.7.1 APIs in the CAS package

The main objects implementing the APIs discussed here are shown in the diagram below. The hierarchy represents that there is a way to get from an upper object to an instance of the lower object, usually by using a method on the upper object; this

is not an inheritance hierarchy.



The main Interface is the CAS interface. This has most of the functionality of the CAS, except for the type system metadata access, and the indexing access. JCas and CAS are alternative representations and API approaches to the CAS; each has a method to get the other. You can mix JCas and CAS APIs in your application as needed. To use the JCas APIs, you have to create the Java classes that correspond to the CAS types, and include them in the Java class path of the application. If you have a CAS or TCAS object, you can get a JCas object by using the getJCas() method call on the CAS object; likewise, you can get the CAS (which could be a TCAS) object from a JCas by using the getCAS() method call on the JCas object. There is also a low level CAS interface that is not part of the official API, and is intended for internal use only – it is not documented here.

The type system metadata APIs are found in the TypeSystem interface. The objects defining each type and feature are defined by the interfaces Type and Feature. The Type interface has methods to see what types subsume other types, to iterate over the types available, and to extract information about the types, including what features it has. The Feature interface has methods that get what type it belongs to, its name, and its range (the kind of values it can hold).

The FSIndexRepository gives you access to methods to get instances of indexes. The FSIndex and AnnotationIndex objects give you methods to create instances of iterators.

Iterators and the CAS methods that create new feature structures return FeatureStructure objects. These objects can be used to set and get the values of defined features within them.

Chapter 24 JCas Reference

The CAS is a system for sharing data among annotators, consisting of data structures (definable at run time), indexes over these data, metadata describing these, and a high performance serialization/deserialization mechanism. JCas is a Java approach to accessing CAS data, based on using generated, specific Java classes for each CAS type.

Annotators process one CAS per call to their process method. During processing, annotators can retrieve feature structures from the passed in CAS, add new ones, modify existing ones, and use and update CAS indexes. Of course, an annotator can also use plain Java Objects in addition; but the data in the CAS is what is shared among annotators within an application.

All the facilities present in the APIs for the CAS are available when using the JCas APIs; indeed, you can use the getCas() method to get the corresponding CAS (which could be a TCAS) object from a JCas (and vice-versa). The JCas APIs often have helper methods that make using this interface more convenient for Java developers, however.

The data in the CAS are typed objects having fields. JCas uses a set of generated Java classes (each corresponding to a particular CAS type) with "getter" and "setter" methods for the features, plus a constructor so new instances can be made. The Java classes don't actually store the data in the class instance; instead, the getters and setters forward to the underlying CAS data representation. Because of this, applications which use the JCas interface can share data with annotators using plain CAS (i.e., not using the JCas approach). Users can modify the JCas generated Java classes by adding fields to them; this allows arbitrary non-CAS data to also be represented within the JCas objects, as well; however, the non-CAS data stored in the JCas object instances cannot be shared with annotators using the plain CAS.

Data in the CAS initially has no corresponding JCas type instances; these are created as needed at the first reference. This means, if your annotator is passed a large CAS having millions of CAS feature structures, but you only reference a few of them, and no previously created Java JCas object instances were created by upstream annotators, the only Java objects that will be created will be those that correspond to the CAS feature structures that you reference.

The JCas class Java source files are generated from XML type system descriptions. The JCasGen utility does the work of generating the corresponding Java Class Model for the CAS types. There are a variety of ways JCasGen can be run; these are described later. You include the generated classes with your UIMA component, and you can publish these classes for others who might want to use your type system.

The specification of the type system in XML can be written using a conventional text editor, an XML editor, or using the Eclipse plug-in that supports editing UIMA descriptors.

Changes to the type system are done by changing the XML and regenerating the corresponding Java Class Models. Of course, once you've published your type system for others to use, you should be careful that any changes you make don't adversely impact the users. Additional features can be added to existing types without breaking other code.

A separate Java class is generated for each type; this type implements the CAS FeatureStructure interface, as well as having the special getters and setters for the included features. In the current implementation, an additional helper class per type is also generated. The generated Java classes have methods (getters and setters) for the fields as defined in the XML type specification. Descriptor comments are reflected in the generated Java code as Java-doc style comments.

Type names used in the CAS correspond to the generated Java classes directly. If the CAS name is com.myCompany.myProject.ExampleClass, the generated Java class is in the package com.myCompany.myProject, and the class is ExampleClass.

24.1 Name Spaces

Full Type names consist of a "namespace" prefix dotted with a simple name. Namespaces are used like packages to avoid collisions between types that are defined by different people at different times. The namespace is used as the Java package name for generated Java files. An exception to this rule is the built-in types starting with `uima.cas` and `uima.tcas`; these names are mapped to Java packages named `com.ibm.uima.jcas.cas` and `com.ibm.uima.jcas.tcas`.

24.2 XML source description tags

Each XML type specification can have <description ... > tags. The description for a type will be copied into the generated Java code, as a JavaDoc style comment for the class. When writing these descriptions in the XML type specification file, you might want to use html tags, as allowed in JavaDocs.

If you use the Component Description Editor, you can write the html tags normally, for instance, "<h1>My Title</h1>. The Component Descriptor Editor will take care of coverting the actual descriptor source so that it has the leading "<" character written as "<", to avoid confusing the XML type specification. For example, <p> would be written in the source of the descriptor as <p>. Any characters used in the JavaDoc comment must of course be from the character set allowed by the XML type specification. These specifications often start with the line <?xml version="1.0" encoding="UTF-8" ?>, which means you can use any of the UTF-8 characters.

24.3 Mapping built-in CAS types to Java types

The built-in primitive CAS types map to Java types as follows:

```
uima.cas.Integer >> int
uima.cas.Float   >> float
uima.cas.String  >> String
```

24.4 Augmenting the generated Java Code

The Java Class Models generated for each type can be augmented by the user. Typical augmentations include adding additional (non-CAS) fields and methods, and import statements that might be needed to support these. Commonly added methods include additional constructors (having different parameter signatures), and implementations of toString().

To augment the code, just edit the generated Java source code for the class named the same as the CAS type. Here's an example of an additional method you might add; the various getter methods are retrieving values from the instance:

```
public String toString() { // for debugging
  return "XsgParse "
    + getslotName() + ": "
    + getheadWord().getCoveredText()
    + " seqNo: " + getseqNo()
    + ", cAddr: " + id
    + ", size left mods: " + getlMods().size()
    + ", size right mods: " + getrMods().size();
}
```

24.4.1 Keeping hand-coded augmentations when regenerating

If the type system specification changes, you have to re-run the JCasGen generator. This will produce updated Java for the Class Models that capture the changed specification. If you have previously augmented the source for these Java Class Models, your changes must be merged with the newly (re)generated Java source code for the Class Models. This can be done by hand, or you can run the version of JCasGen that is integrated with Eclipse, since the merging depends on Eclipse's EMF plug-in. You can obtain Eclipse and the needed EMF plug-in from http://www.eclipse.org.

If you run the generator version that works outside of Eclipse, it will not merge Java source changes you may have previously made; if you want them retained, you'll have to do the merging by hand.

The Java source merging will keep additional constructors, additional fields, and any changes you may have made to the readObject method (see below). Merging

will not delete classes in the target corresponding to deleted CAS types, which no longer are in the source – you should delete these by hand.

24.4.2 Additional Constructors

Any additional constructors that you add must include the JCas argument. The first line of your constructor is required to be

```
this(jcas);        // run the standard constructor
```

where jcas is the passed in JCas reference. If the type you're defining extends `uima.tcas.Annotation`, JCasGen will automatically add a constructor which takes 2 additional parameters – the begin and end Java int values, and set the `uima.tcas.Annotation` `begin` and `end` fields.

Here's an example: If you're defining a type MyType which has a feature parent, you might make an additional constructor which has an additional argument of parent:

```
MyType(JCas jcas, MyType parent) {
    this(jcas);        // run the standard constructor
    setParent(parent);   // set the parent field from the parameter
  }
```

Using readObject

Fields defined by augmenting the Java Class Model to include additional fields represent data that exist for this class in Java, in a local JVM (Java Virtual Machine), but do not exist in the CAS when it is passed to other environments (for example, passing to a remote annotator).

A problem can arise when new instances are created, perhaps by the underlying system when it iterates over an index, which is: how to insure that any additional non-CAS fields are properly initialized. To allow for arbitrary initialization at instance creation time, an initialization method in the Java Class Model, called readObject is used. The generated default for this method is to do nothing, but it is one of the methods that you can modify – to do whatever initialization might be needed. It is called with 0 parameters, during the constructor for the object, after the basic object fields have been set up. It can refer to fields in the CAS using the getters and setters, and other fields in the Java object instance being initialized.

A pre-existing CAS feature structure could exist if a CAS was being passed to this annotator; in this case the JCas system calls the readObject method when creating the corresponding Java instance for the first time for the CAS feature structure. This can happen at two points: when a new object is being returned from an iterator over a CAS index, or a getter method is getting a field for the first time whose value is a feature structure.

24.4.3 Modifying generated items

The following modifications, if made in generated items, will be preserved when regenerating.

The public/private etc. flags associated with methods (getters and setters). You can change the default ("public") if needed.

"final" or "abstract" can be added to the type itself, with the usual semantics.

24.5 Merging types from different type system specifications

24.5.1 Aggregate AEs and CPEs as sources of types

When running aggregate AEs (Analysis Engines), or a set of AEs in a collection processing engine, a merged type system is built. (Note: this "merge" is merging types, not to be confused with merging Java source code, discussed above). This merged type system has all the types of every component used in the application. It is possible that there may be multiple definitions of the same CAS type, each of which might have different features defined; the merged type result is created by accumulating all the defined features for a particular type into that type's type definition.

If no type merging is needed, then each type system can have its own Java Class Models generated individually, perhaps at an earlier time, and the resulting class files (or .jar files containing these class files) can be put in the class path to enable JCas.

JCasGen support for type merging

If type merging is needed, the input to the JCasGen generation process, rather than being a simple type system or a primitive AE specification, is instead, an aggregate AE specification or a CPE (Collection processing engine) specification, which specifies a set of type systems that need to be combined. The generation process will merge the type systems, and the generated output will reflect the merged types. This generated Java source code can be, in turn, merged with hand-done changes to previously generated versions for this aggregate or CPE, as described above. To do this Java source merge, the source for the (hand-modified) generated JCas types must be put into the file system where the generated output will go.

Directions for running JCasGen can be found in *Chapter 16* ***JCasGen User Guide***.

24.6 Using JCas within an Annotator

To use JCas within an annotator, you must include the generated Java classes output from JCasGen in the class path.

An annotator written using JCas is built by defining a class for the annotator that implements JTextAnnotator. The process method for this annotator is written

```
public void process(JCas jcas, ResultSpecification aResultSpec)
     throws AnnotatorProcessException {
    ... // body of annotator goes here
}
```

The process method is passed the JCas instance to use as the first parameter.

The JCas reference is used throughout the annotator to refer to the particular JCas instance being worked on. In pooled or multi-threaded implementations, there will be a separate JCas for each thread being (simultaneously) worked on.

You can do several kinds of operations using the JCas APIs: create new feature structures (instances of CAS types) (using the new operator), access existing feature structures passed to your annotator in the JCas (for example, by using the next method of an iterator over the feature structures), get and set the fields of a particular instance of a feature structure, and add and remove feature structure instances from the CAS indexes. To support iteration, there are also functions to get and use indexes and iterators over the instances in a JCas.

24.6.1 Creating new instances using the Java "new" operator

The new operator creates new instances of JCas types. It takes at least one parameter, the JCas instance in which the type is to be created. For example, if there was a type Meeting defined, you can create a new instance of it using:

```
Meeting m = new Meeting(jcas);
```

Other variations of constructors can be added in custom code; the single parameter version is the one automatically generated by JCasGen. For types that are subtypes of Annotation, JCasGen also generates an additional constructor with additional "begin" and "end" arguments.

24.6.2 Getters and Setters

If the CAS type Meeting had fields location and time, you could get or set these by using getter or setter methods. These methods have names formed by splicing

together the word "get" or "set" followed by the field name, with the first letter of the field name capitalized. For instance

```
getLocation()
```

The getter forms take no parameters and return the value of the field; the setter forms take one parameter, the value to set into the field, and return void.

There are built-in CAS types for arrays of integers, strings, floats, and feature structures. For fields whose values are these types of arrays, there is an alternate form of getters and setters that take an additional parameter, written as the first parameter, which is the index in the array of an item to get or set.

24.6.3 Obtaining references to Indexes

The only way to access instances (not otherwise referenced from other instances) passed in to your annotator in its JCas is to use an iterator over some index. Indexes in the CAS are specified in the annotator descriptor. Indexes have a name; text annotators have a built-in, standard index over all annotations.

To get an index, first get the JFSIndexRepository from the JCas using the method jcas.getJFSIndexRepository(). Here are the calls to get indexes:

```
JFSIndexRepository ir = jcas.getJFSIndexRepository();

ir.getIndex(name-of-index) // get the index by its name, a string
ir.getIndex(name-of-index, Foo.type) // filtered by specific type

ir.getAnnotationIndex()      // get AnnotationIndex
ir.getAnnotationIndex(Foo.type)      // filtered by specific type
```

Filtering types have to be a subtype of the type specified for this index in its index specification. They can be written as either Foo.type or if you have an instance of Foo, you can write

```
fooInstance.jcasType.casType.
```

Foo is (of course) an example of the name of the type.

24.6.4 Adding (and removing) instances to (from) indexes

CAS indexes are maintained automatically by the CAS. But you must add any instances of feature structures you want the index to find, to the indexes by using the call:

```
myInstance.addToIndexes();
```

Do this after setting all features in the instance ***which could be used in indexing***, for example, in determining the sorting order. After indexing, do not change the values of these particular features because the indexes will not be updated. If you need to change the values, you must first remove the instance from the CAS indexes, change the values, and then add the instance back. To remove an instance from the indexes, use the method:

```
myInstance.removeFromIndexes();
```

Note: It's OK to change feature values which are not used in determining sort ordering (or set membership), without removing and re-adding back to the index.

24.6.5 Using Iterators

Once you have an index obtained from the JCas, you can get an iterator from the index; here is an example:

```
FSIndexRepository ir = jcas.getFSIndexRepository();
FSIndex myIndex = ir.getIndex("myIndexName");
FSIterator myIterator = myIndex.iterator();

JFSIndexRepository ir = jcas.getJFSIndexRepository();
FSIndex myIndex = ir.getIndex("myIndexName", Foo.type); // filtered
FSIterator myIterator = myIndex.iterator();
```

Iterators work like normal Java iterators, but are augmented to support additional capabilities. Iterators are described in the CAS Reference, *Section 23.6* ***Indexes and Iterators***.

24.6.6 Class Loaders in UIMA

The basic concept of a UIMA application includes assembling engines into a flow. The applications made up of these Engines are run within the UIMA Framework, either by the Collection Processing Manager, or by using more basic UIMA Framework APIs.

The UIMA Framework exists within a JVM (Java Virtual Machine). A JVM has the capability to load multiple applications, in a way where each one is isolated from the others, by using a separate class loader for each application. For instance, one set of UIMA Framework Classes could be shared by multiple sets of application - specific classes.

Use of Class Loaders is optional

The UIMA framework will use a specific ClassLoader, based on how ResourceManager instances are used. Specific ClassLoaders are only created if you specify an ExtensionClassPath as part of the ResourceManager. If you do not need

to support multiple applications within one UIMA framework within a JVM, don't specify an ExtensionClassPath; in this case, the classloader used will be the one used to load the UIMA framework - usually the overall application class loader.

Of course, you should not run multiple UIMA applications together, in this way, if they have different class definitions for the same class name. This includes the JCas "cover" classes. This case might arise, for instance, if both applications extended `uima.tcas.DocumentAnnotation` in differing, incompatible ways. Each application would need its own definition of this class, but only one could be loaded (unless you specify ExtensionClassPath in the ResourceManager which will cause the UIMA application to load its private versions of its classes, from its classpath).

24.6.7 Issues around DocumentAnnotation

The built-in type, `uima.tcas.DocumentAnnotion`, is frequently extended by applications. The JCas provides a method, `getDocumentAnnotation()`, to get the special instance of this type which associated with each TCAS. Currently this method returns an instance of the JCas cover class for this. Because there can be multiple definitions of this class, this method is deprecated. It will continue to work, as long as the ExtensionClassPath is not being used. If it is being used, the user will see some pretty strange errors, something like

```
ClassCast Exception: Cannot cast "uima.tcas.DocumentAnnotation" to
"uima.tcas.DocumentAnnotation"
```

What's really going on is that the JCas method for this loads a version of the `DocumentAnnotation` class from the UIMA Framework loader, while the Application trying to use it loads a different version of the `DocumentAnnotation` class from its ExtensionClassLoader.

If only one definition of `DocumentAnnotation` will be used for the complete set of UIMA applications being run in the JVM, then you can replace the definition of `DocumentAnnotation` in the Jar that the UIMA Framework loader is using with your definition, and not have this definition findable in the ExtensionClassPath.

This approach is enabled by putting all the extendable, built-in classes for UIMA into a separate JAR file.

The method `getDocumentAnnotationFs()` replaces the deprecated `getDocumentAnnotation()`. It has the same function, except its return type is TOP, which means your code will have to "cast" it to your particular loaded version of `DocumentAnnotation`.

```
   /* deprecated */
DocumentAnnotation docAnn = aJcas.getDocumentAnnotation();
```

```
   /* new way */
DocumentAnnotation docAnn =
(DocumentAnnotation)aJcas.getDocumentAnnotationFs();
```

24.6.8 Issues accessing JCas objects outside of UIMA Engine Components

If you are using the ExtensionClassPaths, the JCas cover classes are loaded under a class loader created by the ResourceManager. If you reference the same JCas classes outside of any UIMA component, for instance, in top level application code, the JCas classes used by that top level application code must be loaded under the same class loader, in order to avoid class cast exceptions. Currently, there is no supported way to do this if you are using ExtensionClassPaths.

The workaround is to do all the JCas processing inside a UIMA component (no processing using JCas outside of the UIMA pipeline), or to put the JCas classes only in the main classpath for the UIMA Framework, and insure they are not findable in the ExtensionClassPaths. This latter approach of course limits you to one set of JCas class definitions per UIMA framework.

24.7 Setting up Classpath for JCas

The JCas Java classes generated by JCasGen are typically compiled and put into a JAR file, which, in turn, is put into the application's class path.

This JAR file must be generated from the application's merged type system. This is most conveniently done by opening the top level descriptor used by the application in the Component Descriptor Editor tool, and pressing the Run-JCasGen button on the Type System Definition page.

Chapter 25 Semantic Search Engine Reference

The documentation describing how Semantic Search and how to write queries using the XML Fragments language can be found in the docs/ directory: SIAPI.pdf . The complete specification of the standard Search and Indexing API is in the docs/ directory: Programming_Guide_and_API_Reference_for_OmniFind.pdf.

Chapter 26 PEAR Reference

A PEAR (Processing Engine ARchive) file is a standard package for UIMA (Unstructured Information Management Architecture) components. This chapter describes the PEAR 1.0 structure and specification.

The PEAR package can be used for distribution and reuse by other components or applications. It also allows applications and tools to manage UIMA components automatically for verification, deployment, invocation, testing, etc.

Currently, the PEAR Eclipse Plugin is available as a tool to create PEAR files for standard UIMA components. Please refer to *Chapter 11* ***PEAR Packager User's Guide*** for more information about this tool.

26.1 Packaging a UIMA component

For the purpose of describing the process of creating a PEAR file and its internal structure, this section describes the steps used to package a UIMA component as a valid PEAR file. The PEAR packaging process consists of the following steps:

- Creating the PEAR structure
- Populating the PEAR structure
- Creating the installation descriptor
- Packaging the PEAR structure into one file

26.1.1 Creating the PEAR structure

The first step in the PEAR creation process is to create a PEAR structure. The PEAR structure is a structured tree of folders and files, including the following elements:

- Required Elements:
  - The **metadata** folder which contains the PEAR installation descriptor and properties files.
  - The installation descriptor (**metadata/install.xml**)
  - A UIMA analysis engine descriptor and its required code, delegates (if any), and resources
- Optional Elements:
  - The desc folder to contain descriptor files of analysis engines, delegates analysis engines (all levels), and other components (Collection Readers, CAS Consumers, etc).
  - The src folder to contain the source code

- The bin folder to contain executables, scripts, class files, dlls, shared libraries, etc.
- The lib folder to contain jar files.
- The doc folder containing documentation materials, preferably accessible through an index.html.
- The data folder to contain data files (e.g. for testing).
- The conf folder to contain configuration files.
- The resources folder to contain other resources and dependencies.
- Other user-defined folders or files are allowed, but should be avoided.

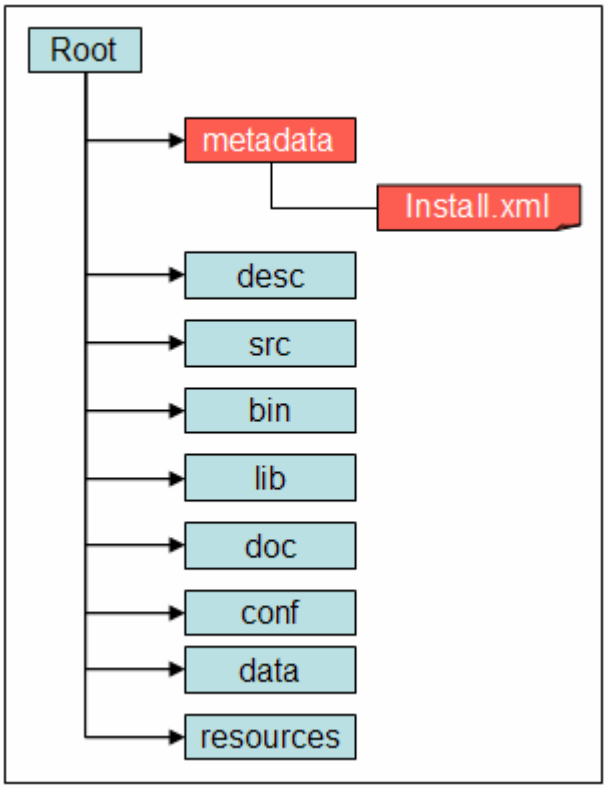


Figure 18. The PEAR Structure

26.1.2 Populating the PEAR structure

After creating the PEAR structure, the component's descriptor files, code files, resources files, and any other files and folders are copied into the corresponding folders of the PEAR structure. The developer should make sure that the code would work with this layout of files and folders, and that there are no broken links. Although it is strongly discouraged, the optional elements of the PEAR structure can be replaced by other user defined files and folder, if required for the component to work properly.

Note: The PEAR structure must be self-contained. For example, this means that the component must run properly independently from the PEAR root folder location. If the developer needs to use an absolute path in configuration or descriptor files, then he/she should put these files in the "conf" or "desc" and replace the path of the PEAR root folder with the string "$main_root". The tools that deploy and use PEAR files should localize the files in the "conf" and "desc" folders by replacing the string "$main_root" with the local absolute path of the PEAR root folder. The "$main_root" macro can also be used in the Installation descriptor (install.xml)

Currently there are three types of component packages depending on their deployment:

Standard type

A component package with the **standard** type must be a valid Analysis Engine, and all the required files to deploy it locally must be included in the PEAR package.

Service type

A component package with the **service** type must be deployable locally as a supported UIMA service (e.g. Vinci). In this case, all the required files to deploy it locally must be included in the PEAR package.

Network Type

A component package with the network type is not deployed locally but rather in the "remote" environment. It's accessed as a network AE (e.g. Vinci Service). The component owner has the responsibility to start the service and make sure it's up and running before it's used by others (like a webmaster that makes sure the web site is up and running). In this case, the PEAR package does not have to contain files required for deployment, but must contain the network AE descriptor (see *4.1.4 **Creating the XML Descriptor***) and the <DESC> tag in the installation descriptor must point to the network TAE descriptor. For more information about Network Analysis Engines, please refer to *Section 6.6 **Working with Analysis Engine and CAS Consumer Services***.

26.1.3 Creating the installation descriptor

The installation descriptor is an xml file called install.xml under the metadata folder of the PEAR structure. It's also called InsD. The InsD XML file should be created in the UTF-8 file encoding. The InsD should contain the following sections:

- <OS>: This section is used to specify supported operating systems

- <TOOLKITS>: This section is used to specify toolkits, such as JDK, needed by the component.
- <SUBMITTED_COMPONENT>: This is the most important section in the InsD. It's used to specify required information about the component. See section 2.3.2 for detailed information about this section.
- <INSTALLATION>: This section is explained in section 26.1.5 .

Documented template for the installation descriptor:

The following is "documented template" for the content of the installation descriptor install.xml:

```
<? xml version="1.0" encoding="UTF-8"?>
<!-- Installation Descriptor Template -->
<COMPONENT_INSTALLATION_DESCRIPTOR>
  <!-- Specifications of OS names, including version, etc. -->
  <OS>
    <NAME>OS_Name_1</NAME>
    <NAME>OS_Name_2</NAME>
  </OS>
  <!-- Specifications of required standard toolkits -->
  <TOOLKITS>
    <JDK_VERSION>JDK_Version</JDK_VERSION>
  </TOOLKITS>

  <!-- There are 2 types of variables that are used in the InsD:
       a) $main_root , which will be substituted with the real path to the
                 main component root directory after installing the
                 main (submitted) component
       b) $component_id$root, which will be substituted with the real path
          to the root directory of a given delegate component after
          installing the given delegate component -->

  <!-- Specification of submitted component (TAE)             -->
  <!-- Note: submitted_component_id is assigned by developer; -->
  <!--       XML descriptor file name is set by developer.    -->
  <!-- Important: ID element should be the first in the       -->
  <!--            SUBMITTED_COMPONENT section.                -->
  <!-- Submitted component may include optional specification -->
  <!-- of Collection Reader that can be used for testing the  -->
  <!-- submitted component.                                   -->
  <!-- Submitted component may include optional specification -->
  <!-- of CAS Consumer that can be used for testing the       -->
  <!-- submitted component.                                   -->

  <SUBMITTED_COMPONENT>
    <ID>submitted_component_id</ID>
    <NAME>Submitted component name</NAME>
    <DESC>$main_root/desc/ComponentDescriptor.xml</DESC>

    <!-- deployment options:                                    -->
    <!-- a) 'standard' is deploying AE locally                  -->
    <!-- b) 'service'  is deploying AE locally as a service,    -->
    <!--    using specified command (script)                   -->
```

```
  <!-- c) 'network'  is deploying a pure network AE, which   -->
  <!--    is running somewhere on the network                -->

  <DEPLOYMENT>standard | service | network</DEPLOYMENT>

  <!-- Specifications for 'service' deployment option only   -->
  <SERVICE_COMMAND>$main_root/bin/startService.bat</SERVICE_COMMAND>
  <SERVICE_WORKING_DIR>$main_root</SERVICE_WORKING_DIR>
  <SERVICE_COMMAND_ARGS>

    <ARGUMENT>
      <VALUE>1st_parameter_value</VALUE>
      <COMMENTS>1st parameter description</COMMENTS>
    </ARGUMENT>

    <ARGUMENT>
      <VALUE>2nd_parameter_value</VALUE>
      <COMMENTS>2nd parameter description</COMMENTS>
    </ARGUMENT>

  </SERVICE_COMMAND_ARGS>

  <!-- Specifications for 'network' deployment option only   -->

  <NETWORK_PARAMETERS>
    <VNS_SPECS VNS_HOST="vns_host_IP" VNS_PORT="vns_port_No" />
  </NETWORK_PARAMETERS>

  <!-- General specifications                                -->

  <COMMENTS>Main component description</COMMENTS>

  <COLLECTION_READER>
    <COLLECTION_ITERATOR_DESC>
      $main_root/desc/CollIterDescriptor.xml
    </COLLECTION_ITERATOR_DESC>

    <CAS_INITIALIZER_DESC>
      $main_root/desc/CASInitializerDescriptor.xml
    </CAS_INITIALIZER_DESC>
  </COLLECTION_READER>

  <CAS_CONSUMER>
    <DESC>$main_root/desc/CASConsumerDescriptor.xml</DESC>
  </CAS_CONSUMER>

</SUBMITTED_COMPONENT>
<!-- Specifications of the component installation process -->
<INSTALLATION>
  <!-- List of delegate components that should be installed together -->
  <!-- with the main submitted component (for aggregate components)  -->
  <!-- Important: ID element should be the first in each             -->

  <!--            DELEGATE_COMPONENT section.                       -->
  <DELEGATE_COMPONENT>
    <ID>first_delegate_component_id</ID>
    <NAME>Name of first required separate component</NAME>
  </DELEGATE_COMPONENT>

  <DELEGATE_COMPONENT>
    <ID>second_delegate_component_id</ID>
    <NAME>Name of second required separate component</NAME>
  </DELEGATE_COMPONENT>
```

```
    <!-- Specifications of local path names that should be replaced -->
    <!-- with real path names after the main component as well as   -->
    <!-- all required delegate (library) components are installed.  -->
    <!-- <FILE> and <REPLACE_WITH> values may use the $main_root or -->
    <!-- one of the $component_id$root variables.                   -->
    <!-- Important: ACTION element should be the first in each      -->
    <!--            PROCESS section.                                -->

    <PROCESS>
      <ACTION>find_and_replace_path</ACTION>
      <PARAMETERS>
        <FILE>$main_root/desc/ComponentDescriptor.xml</FILE>
        <FIND_STRING>../resources/dict/</FIND_STRING>
        <REPLACE_WITH>$main_root/resources/dict/</REPLACE_WITH>
        <COMMENTS>Specify actual dictionary location in XML component
          descriptor
        </COMMENTS>
      </PARAMETERS>
    </PROCESS>

    <PROCESS>
      <ACTION>find_and_replace_path</ACTION>
      <PARAMETERS>
        <FILE>$main_root/desc/DelegateComponentDescriptor.xml</FILE>
        <FIND_STRING>
local_root_directory_for_1st_delegate_component/resources/dict/
        </FIND_STRING>
        <REPLACE_WITH>
          $first_delegate_component_id$root/resources/dict/
        </REPLACE_WITH>
        <COMMENTS>
          Specify actual dictionary location in the descriptor of the 1st
          delegate component
        </COMMENTS>
      </PARAMETERS>
    </PROCESS>

    <!-- Specifications of environment variables that should be set prior
         to running the main component and all other reused components.
         <VAR_VALUE> values may use the $main_root or one of the
         $component_id$root variables. -->

    <PROCESS>
      <ACTION>set_env_variable</ACTION>
      <PARAMETERS>
        <VAR_NAME>env_variable_name</VAR_NAME>
        <VAR_VALUE>env_variable_value</VAR_VALUE>
        <COMMENTS>Set environment variable value</COMMENTS>
      </PARAMETERS>
    </PROCESS>

  </INSTALLATION>
</COMPONENT_INSTALLATION_DESCRIPTOR>
```

The SUBMITTED_COMPONENT section

The SUBMITTED_COMPONENT section of the installation descriptor (install.xml) is the most important. It's used to specify required information about the UIMA component. Before explaining the details, let's clarify the concept of component ID and "macros" used in the installation descriptor. The component ID element should be the **first element** in the SUBMITTED_COMPONENT section.

The component id is a string that uniquely identifies the component. It should use the JAVA naming convention (e.g. ibm.uima.mycomponent).

Macros are variables such as $main_root, used to represent a string such as the full path of a certain directory.

These macros should be defined in the PEAR.properties file using the local values. The tools and applications that use and deploy PEAR files should replace these macros with the corresponding values in the local environment as part of the deployment process in the files included in the conf and desc folders.

Currently, there are two types of macros:

- $main_root, which represents the local absolute path of the main component root directory after deployment.
- $*component_id*$root, which represents the local absolute path to the root directory of the component which has *component_id* as component ID. This component could be, for instance, a delegate component.

For example, if some part of a descriptor needed to have a path to the data subdirectory of the PEAR, you would write `$main_root/data`. If your PEAR refers to a delegate component having the ID "`my.comp.Dictionary`", and you need to specify a path to one of this component's subdirectories, say `resource/dict`, you would write `$my.comp.Dictionary$root/resources/dict`.

The ID, NAME, and DESC tags

These tags are used to specify the component ID, Name, and descriptor path using the corresponding tags as follows:

```
<SUBMITTED_COMPONENT>
  <ID>submitted_component_id</ID>
  <NAME>Submitted component name</NAME>
  <DESC>$main_root/desc/ComponentDescriptor.xml</DESC>
```

Tags related to deployment types

As mentioned before, there are currently three types of PEAR packages, depending on the following deployment types:

Standard type

A component package with the **standard** type must be a valid UIMA Analysis Engine, and all the required files to deploy it must be included in the PEAR package. This deployment type should be specified as follows:

```
<DEPLOYMENT>standard</DEPLOYMENT>
```

Service type

A component package with the **service** type must be deployable locally as a supported UIMA service (e.g. Vinci). The installation descriptor must include the path for the executable or script to start the service including its arguments, and the working directory from where to launch it, following this template:

```
<DEPLOYMENT>service</DEPLOYMENT>
<SERVICE_COMMAND>$main_root/bin/startService.bat</SERVICE_COMMAND>
<SERVICE_WORKING_DIR>$main_root</SERVICE_WORKING_DIR>
<SERVICE_COMMAND_ARGS>
<ARGUMENT>
<VALUE>1st_parameter_value</VALUE>
<COMMENTS>1st parameter description</COMMENTS>
</ARGUMENT>
<ARGUMENT>
<VALUE>2nd_parameter_value</VALUE>
<COMMENTS>2nd parameter description</COMMENTS>
</ARGUMENT>
</SERVICE_COMMAND_ARGS>
```

Network Type

A component package with the network type is not deployed locally, but rather in a "remote" environment. It's accessed as a network AE (e.g. Vinci Service). In this case, the PEAR package does not have to contain files required for deployment, but must contain the network AE descriptor. The <DESC> tag in the installation descriptor (See section 2.3.2.1) must point to the network AE descriptor. Here is a template in the case of Vinci services:

```
<DEPLOYMENT>network</DEPLOYMENT>
```

```
<NETWORK_PARAMETERS>
<VNS_SPECS VNS_HOST="vns_host_IP" VNS_PORT="vns_port_No" />
</NETWORK_PARAMETERS>
```

The Collection Reader and CAS Consumer tags

These sections of the installation descriptor are used by any specific Collection Reader or CAS Consumer to be used with the packaged analysis engine. See the template in section 2.3.1.

The INSTALLATION section

The <INSTALLATION> section specifies the external dependencies of the component and the operations that should be performed during the PEAR package installation.

The component dependencies are specified in the <DELEGATE_COMPONENT> sub-sections, as shown in the installation descriptor template above.

Important: The ID element should be the first element in each <DELEGATE_COMPONENT> sub-section.

The <INSTALLATION> section may specify the following operations:

- Setting environment variables that are required to run the installed component.

  Note: Note that you can use "macros", like $main_root or $component_id$root in the VAR_VALUE element of the <PARAMETERS> sub-section.

- Finding and replacing string expressions in files.

  Note: Note that you can use the "macros" in the FILE and REPLACE_WITH elements of the <PARAMETERS> sub-section.

Important: the ACTION element always should be the 1st element in each <PROCESS> sub-section.

By default, the PEAR Installer will try to process every file in the desc and conf directories of the PEAR package in order to find the "macros" and replace them with actual path expressions. In addition to this, the installer will process the files specified in the <INSTALLATION> section.

Important: all XML files which are going to be processed should be created using UTF-8 or UTF-16 file encoding. All other text files which are going to be processed should be created using the ASCII file encoding.

26.1.4 Packaging the PEAR structure into one file

The last step of the PEAR process is to simply **zip** the content of the PEAR root folder (**not including the root folder itself**). The PEAR file must have a ".pear" extension.

26.1.5 Installing a PEAR file

For information about the installation of a PEAR file and the PEAR Installer tool, please refer to the "PEAR Installer" Chapter.

Chapter 27 XMI CAS Serialization Reference

This is the specification for the mapping of the UIMA CAS into the XMI (XML Metadata Interchange5) format. XMI is an OMG standard for expressing object graphs in XML. The UIMA SDK provides support for XMI through the classes com.ibm.uima.cas.impl.XmiCasSerializer and com.ibm.uima.cas.impl.XmiCasDeserializer.

27.1 XMI Tag

The outermost tag is <XMI> and must include a version number and XML namespace attribute:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi=http://www.omg.org/XMI>
  <!-- CAS Contents here -->
</xmi:XMI>
```

XML namespaces[6] are used throughout. The "xmi" namespace prefix is used to identify elements and attributes that are defined by the XMI specification. The XMI document will also define one namespace prefix for each CAS namespace, as described in the next section.

27.2 Feature Structures

UIMA Feature Structures are mapped to XML elements. The name of the element is formed from the CAS type name, making use of XML namespaces as follows.

The CAS type namespace is converted to an XML namespace URI by the following rule: replace all dots with slashes, prepend http:///, and append .ecore.

This mapping was chosen because it is the default mapping used by the Eclipse Modeling Framework (EMF)[7] to create namespace URIs from Java package names. The use of the http scheme is a common convention, and does not imply any HTTP communication. The .ecore suffix is due to the fact that the recommended type system definition for a namespace is an ECore model, see ***XMI and EMF Interoperability*** on page 8-173.

Consider the CAS type name "org.myproj.Foo". The CAS namespace ("org.myorg.") is converted to the XML namespace URI is http:///org/myproj.ecore.

[5] For details on XMI see Grose et al. *Mastering XMI. Java Programming with XMI, XML, and UML.* John Wiley & Sons, Inc. 2002.

[6] http://www.w3.org/TR/xml-names11/

[7] For details on EMF and Ecore see Budinsky et al. *Eclipse Modeling Framework 2.0*. Addison-Wesley. 2006.

The XML element name is then formed by concatenating the XML namespace prefix (which is an arbitrary token, but typically we use the last component of the CAS namespace) with the type name (excluding the namespace).

So the example "org.myproj.Foo" FeatureStructure is written to XMI as:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:myproj="http:///org/myproj.ecore">
  ...
  <myproj:Foo xmi:id="1"/>
  ...
</xmi:XMI>
```

The xmi:id attribute is only required if this object will be referred to from elsewhere in the XMI document. If provided, the xmi:id must be unique for each feature.

All namespace prefixes (e.g. "myproj") in this example must be bound to URIs using the "xmlns..." attribute, as defined by the XML namespaces specification.

27.3 Primitive Features

CAS features of primitive types (currently String, Integer, or Float, but others are possible) can be mapped either to XML attributes or XML elements. For example, a CAS FeatureStructure of type org.myproj.Foo, with features:

```
begin   = 14
end     = 19
myFeature = "bar"
```

could be mapped to:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:myproj="http:///org/myproj.ecore">
  ...
  <myproj:Foo xmi:id="1" begin="14" end="19" myFeature="bar"/>
  ...
</xmi:XMI>
```

or equivalently:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI
  xmlns:myproj="http:///org/myproj.ecore">
  ...
  <myproj:Foo xmi:id="1">
    <begin>14</begin>
    <end>19</end>
    <myFeature>bar</myFeature>
  </myproj:Foo>
  ...
</xmi:XMI>
```

The attribute serialization is preferred for compactness, but either representation is allowable. Mixing the two styles is allowed; some features can be represented as attributes and others as elements.

27.4 Reference Features

CAS features that are references to other feature structures (excluding arrays and lists, which are handled separately) are serialized as ID references.

If we add to the previous CAS example a feature structure of type org.myproj.Baz, with feature "myFoo" that is a reference to the Foo object, the serialization would be:

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
  xmlns:myproj="http:///org/myproj.ecore">
  ...
  <myproj:Foo xmi:id="1" begin="14" end="19" myFeature="bar"/>
  <myproj:Baz xmi:id="2" myFoo="1"/>
  ...
</xmi:XMI>
```

As with primitive-valued features, it is permitted to use an element rather than an attribute. However, the syntax is slightly different:

```
<myproj:Baz xmi:id="2">
   <myFoo href="#1"/>
<myproj.Baz>
```

Note that in the attribute representation, a reference feature is indistinguishable from an integer-valued feature, so the meaning cannot be determined without prior knowledge of the type system. The element representation is unambiguous.

27.5 Array and List Features

For a CAS feature whose range type is one of the CAS array or list types (currently uima.cas.IntegerArray, uima.cas.FloatArray, uima.cas.StringArray, uima.cas.FSArray, uima.cas.IntegerList, uima.cas.FloatList, uima.cas.StringList, and uima.cas.FSList), the XMI serialization depends on the setting of the "multipleReferencesAllowed" attribute for that feature in the UIMA Type System Description (see ***Features*** on page 20-263.

An array or list with multipleReferencesAllowed = false (the default) is serialized as a "multi-valued" property in XMI. An array or list with multipleReferencesAllowed = true is serialized as a first-class object. Details are described below.

27.5.1 Arrays and Lists as Multi-Valued Properties

In XMI, a multi-valued property is the most natural XMI representation for most cases. Consider the example where the FeatureStructure of type org.myproj.Baz has a feature myIntArray whose value is the integer array {2,4,6}. This can be mapped to:

```
<myproj:Baz xmi:id="3" myIntArray="2 4 6"/>
```

or equivalently:

```
<myproj:Baz xmi:id="3">
  <myIntArray>2</myIntArray>
  <myIntArray>4</myIntArray>
  <myIntArray>6</myIntArray>
</myproj:Baz>
```

Note that String arrays whose elements contain embedded spaces MUST use the latter mapping.

FSArray or FSList features are serialized in a similar way. For example an FSArray feature that contains references to the elements with xmi:id's "13" and "42" could be serialized as:

```
<myproj:Baz xmi:id="3" myFsArray="13 42"/>
```

or:

```
<myproj:Baz xmi:id="3">
  <myFsArray href="#13"/>
  <myFsArray href="#42"/>
</myproj:Baz>
```

27.5.2 Arrays and Lists as First-Class Objects

The multi-valued-property representation described in the previous section does not allow multiple references to an array or list object. Therefore, it cannot be used for features that are defined to allow multiple references (i.e. features for which multipleReferencesAllowed = true in the Type System Description).

When multipleReferencesAllowed is set to true, array and list features are serialized as references, and the array or list objects are serialized as separate objects in the XMI. Consider again the example where the FeatureStructure of type org.myproj.Baz has a feature myIntArray whose value is the integer array {2,4,6}. If myIntArray is defined with multipleReferencesAllowed=true, the serialization will be as follows:

```
<myproj:Baz xmi:id="3" myIntArray="4"/>
```

or:

```
<myproj:Baz xmi:id="3">
  <myIntArray href="#4"/>
</myproj:Baz>
```

with the array object serialized as:

```
<cas:IntegerArray xmi:id="4" elements="2 4 6"/>
```

or:

```
<cas:IntegerArray xmi:id="4">
  <elements>2</elements>
  <elements>4</elements>
  <elements>6</elements>
</cas:IntegerArray>
```

Note that in this case, the XML element name is formed from the CAS type name (e.g. "uima.cas.IntegerArray") in the same way as for other FeatureStructures. The elements of the array are serialized either as a space-separated attribute named "elements" or as a series of child elements named "elements".

List nodes are just standard FeatureStructures with "head" and "tail" features, and are serialized using the normal FeatureStructure serialization. For example, an IntegerList with the values 2, 4, and 6 would be serialized as the four objects:

```
<cas:NonEmptyIntegerList xmi:id="10" head="2" tail="11"/>
<cas:NonEmptyIntegerList xmi:id="11" head="4" tail="12"/>
<cas:NonEmptyIntegerList xmi:id="12" head="6" tail="13"/>
<cas:EmptyIntegerList xmi:id"13"/>
```

This representation of arrays allows multiple references to an array of list. It also allows a feature with range type TOP to refer to an array or list. However, it is a very unnatural representation in XMI and does not support interoperability with other XMI-based systems, so we instead recommend using the multi-valued-property representation described in the previous section whenever it is possible.

27.6 Null Array/List Elements

In UIMA, an element of an FSArray or FSList may be null. In XMI, multi-valued properties do not permit null values. As a workaround for this, we will use a dummy instance of the special type cas:NULL, which has xmi:id 0. For example, in the following example the "myFsArray" feature refers to an FSArray whose second element is null:

```
<cas:NULL xmi:id="0"/>
<myproj:Baz xmi:id="3">
  <myFsArray href="#13"/>
  <myFsArray href="#0"/>
  <myFsArray href="#42"/>
</myproj:Baz>
```

27.7 Subjects of Analysis (Sofas) and Views

A UIMA CAS contain one or more subjects of analysis (Sofas). These are serialized no differently from any other feature structure. For example:

```
<?xml version="1.0" encoding="ASCII"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi=http://www.omg.org/XMI
                           xmlns:cas="http:///uima/cas.ecore">
  <cas:Sofa xmi:id="1" sofaNum="1"
            text="the quick brown fox jumps over the lazy dog."/>
</xmi:XMI>
```

Each Sofa defines a separate View. Feature Structures in the CAS can be members of one or more views. (A Feature Structure that is a member of a view is indexed in its IndexRepository, but that is an implementation detail.)

In the XMI serialization, views will be represented as first-class objects. Each View has an (optional) "sofa" feature, which references a sofa, and multi-valued reference to the members of the View. For example:

```
<cas:View sofa="1" members="3 7 21 39 61"/>
```

Here the integers 3, 7, 21, 39, and 61 refer to the xmi:id fields of the objects that are members of this view.

If the sofa feature is omitted, then this is interpreted as the "base" view, whose members pertain to the artifact as a whole rather than any individual Sofa.

27.8 Linking an XMI Document to its Ecore Type System

If the CAS Type System has been saved to an Ecore file (which is the subject of a different spec), it is possible to store a link from an XMI document to that Ecore type system. This is done using an xsi:schemaLocation attribute on the root XMI element.

The xsi:schemaLocation attribute is a space-separated list that represents a mapping from namespace URI (e.g. http:///org/myproj.ecore) to the physical URI of the .ecore file containing the type system for that namespace. For example:

```
xsi:schemaLocation=
"http:///org/myproj.ecore file:/c:/typesystems/myproj.ecore"
```

would indicate that the definition for the org.myproj CAS types is contained in the file `c:/typesystems/myproj.ecore`. You can specify a different mapping for each of your CAS namespaces, using a space separated list. For details see Budinsky et al. *Eclipse Modeling Framework*.

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Deutschland Informationssysteme GmbH
Department 0790
Pascalstrasse 100
70569 Stuttgart
Germany

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided

by IBM under terms of the IBM Customer Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information is for planning purposes only. The information herein is subject to change before the products described become available.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. _enter the year or years_. All rights reserved.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States, other countries, or both.

IBM

The following terms are trademarks or registered trademarks of other companies:

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Eclipse is a trademark of Eclipse Foundation, Inc.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.