

Create a UIMA component Web service, Part 1: Create a UIMA application using Eclipse

Use wizards to simplify component creation

Skill Level: Intermediate

[Nicholas Chase \(ibmquestions@nicholaschase.com\)](mailto:ibmquestions@nicholaschase.com)

Freelance writer

Backstop Media

28 Jul 2005

Search word processing documents, emails, video, and other unstructured information for specific text or even for concepts using the Unstructured Information Management Architecture (UIMA). Part 1 of this tutorial explains how to install and use the UIMA Eclipse plug-ins to create a simple UIMA application.

Section 1. Before you start

In this tutorial, you learn about UIMA type systems and their descriptors and how to create them in Eclipse. You also learn about Annotators and Annotations and Analysis Engines and their descriptors. Then find out how to access Analysis Engines and the Common Analysis Structure (CAS) using a Java™ application. Finally, you learn about the CAS Visual Debugger.

About this series

This series chronicles the creation of a UIMA component that can be accessed as a Web service. Part 1 describes the actual creation of the component using Eclipse, and Part 2 converts the component into a Web service and discusses the creation of a client to use that component.

About this tutorial

This tutorial is for developers who may or may not have a general idea of the concepts behind UIMA, but who are ready to start building an actual application. Using Eclipse plug-ins and other tools included as part of a UIMA SDK, you create a UIMA type system, Annotator, and Analysis Engine, and you write an application that ties them all together.

You build an application for searching unstructured text files for specific patterns of text. Your application uses this information to create a common analysis structure, which you then analyze using a Java application. In the course of all this, you learn to do the following:

- Install the UIMA SDK for Eclipse
- Create a type system
- Generate Java classes from the type system
- Create an Annotator
- Create an Analysis Engine descriptor
- Use UIMA tools such as the CAS Visual Debugger from within Eclipse
- Create an application that programmatically calls a UIMA Analysis Engine
- Programmatically access UIMA analysis data

Prerequisites

In order to make use of this tutorial, you should have a general familiarity with working in the Eclipse IDE, but steps directly related to the UIMA tools will be discussed in detail. You should also be familiar with Java programming, but all code will be discussed in sufficient detail so that beginning Java programmers should be able to follow along.

Familiarity with the UIMA in general would be helpful, but is not required. Important UIMA concepts will be covered in the tutorial.

To follow along with this tutorial, you should have the following tools installed and tested prior to beginning:

- **Java 2 Standard Edition SDK.** Before you can even install Eclipse, you need to have a working installation of the Java SDK. You can download version 1.4.x or higher from the following location: <http://java.sun.com/j2se/1.5.0/download.jsp>.
- **The Eclipse IDE.** The UIMA SDK works with both versions 2 and 3 of the Eclipse development environment, but the steps in this tutorial assume that you are using version 3.1.1. Later versions should also work, although specific steps might change slightly. You can download Eclipse from the following location: <http://www.eclipse.org/downloads/index.php>.

- **The UIMA SDK.** The UIMA SDK comes in several different varieties, with installers for both Windows and UNIX. This tutorial assumes that you have downloaded the platform-independent zip file, available at <http://alphaworks.ibm.com/tech/uima/download>, which provides specific installation instructions.
-

Section 2. What is UIMA?

The first thing you need to understand is the "big picture" behind UIMA. What are you searching, and why is it important? What's more, how does the UIMA SDK help?

How UIMA works

The key to understanding how the Unstructured Information Management Architecture (UIMA) works is to focus on the "U" and the "M" in the acronym. The overall idea is that there is an almost unimaginable amount of data locked up in unstructured documents such as word processing files, e-mails, video, and audio. In a database, it's easy to get just the information you want; the SQL statement allows you to pull a specific column from a specific row. Pulling information such as a name or idea out of a mass of text is not quite so simple.

The UIMA SDK provides a standard way of organizing the search through this information and recording the results so that it can be passed on for further analysis. For example, a company might wish to preprocess emails from customers in order to route them to the appropriate department. A researcher might want to analyze hundreds of hours of video in order to find patterns relating to human or animal behavior. A publicist might want to search the Web for particular favorable or unfavorable remarks about companies in a particular industry.

To accomplish this, UIMA uses the concept of an Analysis Engine, which analyzes the data (possibly in conjunction with other Analysis Engines) and saves the information in a comment analysis structure (CAS) object. Because the CAS object is a standard structure, any application that understands it, no matter what the platform or development environment, can use it. This capability makes it especially useful in conjunction with Service-Oriented Architecture (SOA), where different pieces of the same puzzle might not share anything more than a common goal. As long as all parts of a distributed application are using the same structures for their data, programmers can mix and match components at will. UIMA is designed specifically with this goal in mind.

The UIMA SDK

UIMA consists of the two following main parts:

1. A platform-independent framework in which you can run solutions that embody a standard interface.
2. A software development kit that enables you to write applications for that framework.

The framework is a platform-independent run-time environment into which you can plug your UIMA components in order to create a solution. The UIMA SDK includes a Java implementation of that environment. It also includes a number of tools for facilitating development, including Eclipse plug-ins.

These tools include a visual debugger for looking at the "annotations" an Analysis Engine finds, the Document Analyzer to run an Analysis Engine against the series of documents (especially good for testing), and plug-ins that assist in the creation of the "descriptors" many UIMA components need.

In this tutorial, you will learn how to use many of these tools.

Section 3. The project

During the course of this tutorial, you build an application that looks for specific kinds of information. Let's take a more detailed look at what you can accomplish in this tutorial.

The goal

The purpose of this project is to create an application that searches unstructured information such as text files for specific kinds of information and then displays the information. In practice, how you search for information depends on what it is you're searching; obviously, you use different techniques for searching text than you would use for searching video or audio. In this case, you use regular expressions to search text files.

You build a system that analyzes internal corporate documents looking for references to product numbers. When it finds them, it notes them, and associates them with the appropriate product line. In this tutorial, you build an application that simply lists the products found in one document, but the principle is similar for searching collections of documents and doing more intensive analysis.

The document

A UIMA application typically analyzes a collection of documents, but the application you build here analyzes just one: a company report regarding consumer contacts.

The document is a narrative but includes product numbers with the text. See [Listing 1](#).

Listing 1. Company report for analyzing

October Survey Report

This document reports on consumer contacts regarding our "Universe" and "Beyond" new-age vacuum cleaner product lines for the month of October, 2005.

There were 130 contacts this month. Twenty consumers sent comments on the Space Age Power Suction Cleaner (BNA-233). Of the 20, 17 like the product. There were three complaints, however; two that there was an unpleasant odor coming from the machine and one that the machine was too easily clogged. The first complaints were solved when it was discovered that consumers were not changing the filters on the machines. The last complaint turned out to be caused by the consumer, who had tried to vacuum up her children's socks.

As a result of 106 complaints about the new Heavy Duty Mega Super Sonic Vacuum (UNA-87322), the product has been recalled. The recall was initiated because the vacuum was so powerful. It was destroying carpets by suctioning them up. One household pet was also injured as a result of supersonic suctioning. The company has paid for the vet bill and the fur transplants.

There were only four responses to our Mini Laser Little Wizard Vacuum (BOA-549). Three were favorable, but one involved a complaint about a rude service manager. The complaint was investigated, and after being reprimanded, the service manager apologized for telling the client he was "simply too dumb to turn on a switch."

You build an application that simply notes the product number occurrences and classifies them by product line based on the pattern of the product number. A more extensive UIMA application might also look for the product name or even a sense of whether comments were positive or negative.

The Analysis Engine's job

At its heart, an Analysis Engine is simply an Annotator that has been wrapped with a descriptor, enabling it to be used in the context of the UIMA application. The Annotator's job is to find instances of a particular kind of data (in this case, product numbers) and create Annotations, or instances of that data. An Annotation includes the actual data as well as its position within the document.

For example, the first Annotation in this document involves product number BNA-233, which begins at character 281 and ends at character 287. All of this information is included in the Annotation.

More than that, however, the Analysis Engine can add additional information based on the type. In this case, the ProductNumber type also has an attribute for the product line. It is the Analysis Engine's job to add this information as well.

The Analysis Engine stores its information in the CAS object.

The CAS

When the Analysis Engine finds an Annotation, it adds the information to a structure called the Common Analysis Structure, or CAS. The CAS object contains all of the Annotations for a specific artifact, such as a file being analyzed, as well as the artifact itself. In a sense, you can think of the Annotations as metadata included with the actual file.

The advantage of the CAS object is two-fold. First, it provides a standard way for you to pass results around. It means that you can pass the results of one Analysis Engine into a second Analysis Engine, and from there into a third and so on. Second, it means that you have a standard way to analyze the final results. It is the CAS object with which the application interacts in order to get the final results.

The overall procedure

Developing a UIMA solution generally involves the following steps:

1. Define the CAS types: These are the types of data for which you search. For example, you create Annotations for product numbers, so you define a `ProductNumber` type for the CAS object.
2. Generate Java classes for the CAS types: in order to work with the data, the application uses Java representations of each type. The UIMA SDK includes a tool, JCasGen, for generating classes with the appropriate getter and setter methods for each type.
3. Create the Annotator: This is the class that actually performs the analysis on your documents.
4. Create the Analysis Engine: In this step, you combine the Annotator with a descriptor that enables the SDK to use it as an Analysis Engine.
5. Test the Analysis Engine: The UIMA SDK includes several tools that make it easy to test your Analysis Engine. Specifically, you use the CAS Visual Debugger.

Once you have a working Analysis Engine, you incorporate it into the actual application.

Section 4. Prepare the environment

Before you can start building anything, you need to prepare your environment by setting up the UIMA toolkit and its Eclipse-specific tools.

Install the UIMA SDK

You can obtain the UIMA SDK in a number of different forms, including installers for Windows and Linux. If you're using one of these installers, simply execute it and follow the instructions. If, on the other hand, you have downloaded the platform-independent (and much smaller) zip file, execute the following steps:

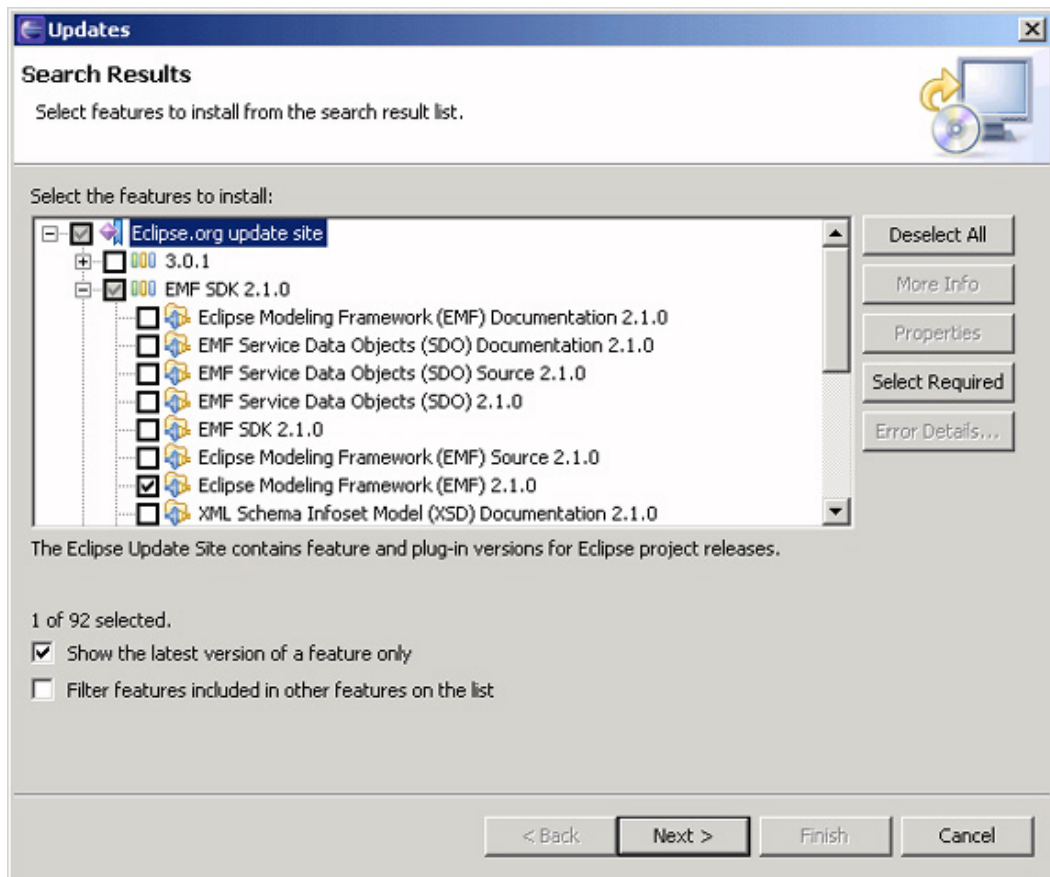
1. Extract the files into a location that you designate as UIMA_HOME. For example, C:\uima1.2.1.
2. Set the value of UIMA_HOME to this location. For example, on Windows, right-click **My Computer** and choose Properties > Advanced > Environment Variables > New to set the value.
3. Append UIMA_HOME\bin to your PATH.
4. Make sure that your JAVA_HOME environment variable points to your JRE installation, such as C:\j2sdk1.4.2_05.
5. Open a command window and execute:
UIMA_HOME\bin\adjustExamplePaths.bat
or:
UIMA_HOME/bin/adjustExamplePaths.sh
depending on your operating system. This script runs a Java program that fixes hardcoded paths in the examples.

Now it's time to prepare Eclipse for the SDK.

Get the Eclipse Modeling Framework

The Eclipse plug-ins that ship with the UIMA SDK are designed to work with the Eclipse Modeling Framework (EMF), which is not part of a standard Eclipse installation. Fortunately, it is not difficult to get. Execute the following steps:

1. Open Eclipse, and select Help > Software Updates > Find and Instal.
2. Select Search for new features to install and click **Next**.
3. Make sure that the Eclipse.org update site and Ignore features not applicable to this environment checkboxes are selected and click **Finish**.
4. If prompted, choose an appropriate mirror site and click **OK**.
5. Navigate to the EMF to make sure that it is selected. See [Figure 1](#).
Figure 1. EMF Framework



6. Click **Next**.
7. Read and accept the license agreement.
8. Click **Finish**.
9. Click **Install** to confirm that you want to install the feature, even though it is unsigned.
10. You will need to restart Eclipse after the plug-in installation, so don't bother restarting now. Click **No** in the dialog box and close Eclipse.

Install the plug-ins

Installing the plug-ins is a much simpler process. In the UIMA_HOME directory you'll find a directory called eclipsePlugin. Inside this directory are two zip files, one each for version 2 and version 3 of Eclipse. Choose the appropriate version for your environment and extract it into the plugins directory of your Eclipse installation.

To make sure that Eclipse recognizes the new plug-in, open a new command window and start the program with the -clean option, as in:

```
eclipse -clean
```

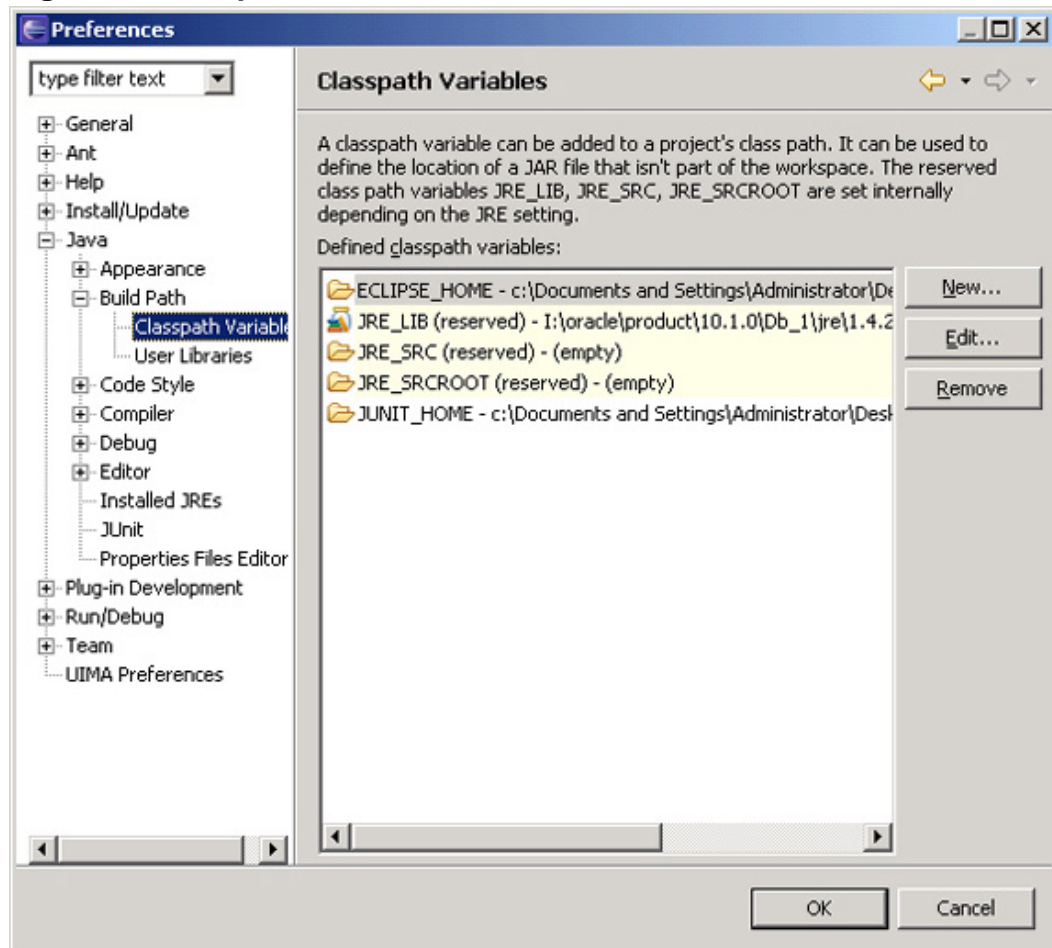

You only need to do this once, so Eclipse knows to check for any configurational changes. Thereafter, you can start the application as you normally would.

Set the UIMA_HOME variable

For Eclipse to know where to find the appropriate classes for the UIMA framework, you need to add the `UIMA_HOME` variable to the Java "build path." To do that, do the following:

1. Make sure that the Java perspective is open by choosing Window > Open Perspective > Java
2. Choose Window > Preferences > Java > Built Path > Classpath Variables, as shown in [Figure 2](#).

Figure 2. Classpath variables



3. Click **New**.
4. Set the name of the new variable to `UIMA_HOME` and the value to the directory in which you installed the SDK. (In other words, the same value set to in the operating system).

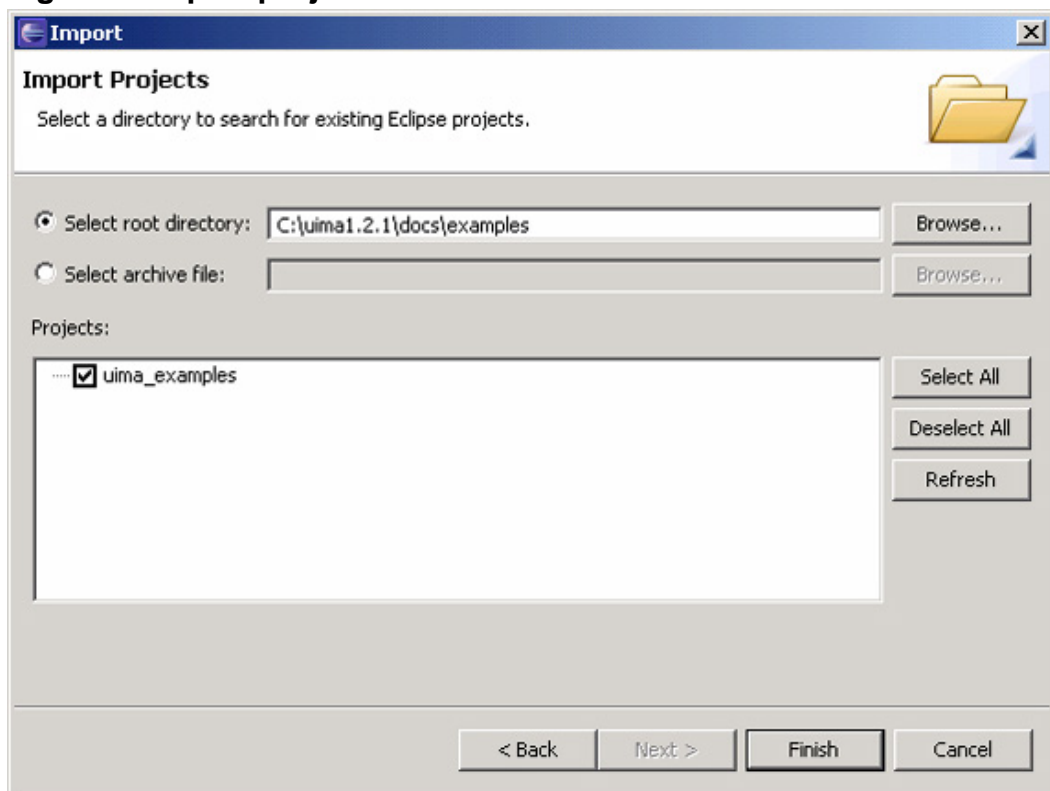
5. Click **OK** twice to return to the Java perspective.

Import the examples

A number of UIMA tools can be run from within Eclipse, as long as you have the appropriate run-time configuration. In order to simplify this set up, import the example project that comes with the UIMA SDK. Perform the following steps:

1. Choose File > Import.
2. As an import source, select Existing Projects into Workspace and click **Next**.
3. Choose Select root directory and click **Browse**.
4. Navigate to the docs\examples directory in UIMA_HOME.
5. Make sure uima_examples is selected and click **Finish**. See [Figure 3](#).

Figure 3. Import projects



Depending on the speed of your machine, the import might take several moments. Errors might appear in the Problems pane while Eclipse builds the project, but they should all disappear when building is complete.

When building is complete, you should see a complete project hierarchy in the

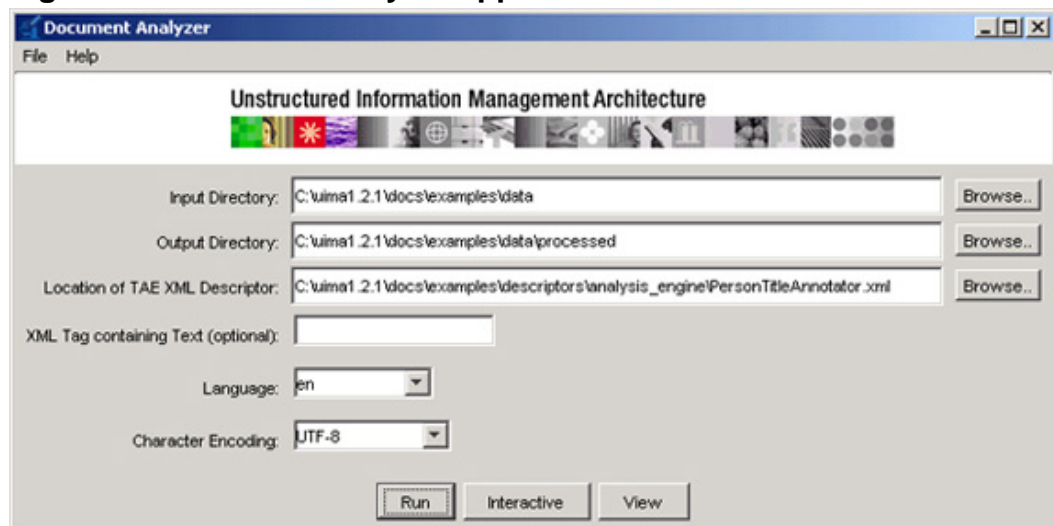
Package Explorer pane.

Check the configuration

To make sure that everything has been installed and configured properly, you can run the Document Analyzer tool that comes with the SDK. Execute the following steps:

1. Select **Run > Run** from the Eclipse menu.
2. Expand the Java Application node if necessary, and click **Document Analyzer**.
3. Click **Run**.
4. After a few moments you will see a new window with a Document Analyzer application. If you have never used this application before, the inputs will look something like what you see in [Figure 4](#).

Figure 4. Document Analyzer application window



5. Click **Run** to execute the search.

At this point, everything should be working. You can also get a good idea of what kinds of applications you can put together by looking at the results of this search. When you're finished, close the extra windows to return to Eclipse.

Section 5. Create the type description

Now that everything is working, it's time to begin building the application. Start by

defining the information the application will look for.

The ProductNumber type

The basic unit of information which the application is going to search is the product number. In this case, product numbers fall into one of two patterns.

Product numbers in the "Universe" product line consist of three capital letters, starting with a capital U, followed by a hyphen (-) and then five digits. Product numbers in the "Beyond" product line consist of three capital letters, starting with a capital B, followed by a hyphen and three digits. Later, when you create the actual Annotator, you express those patterns as regular expressions.

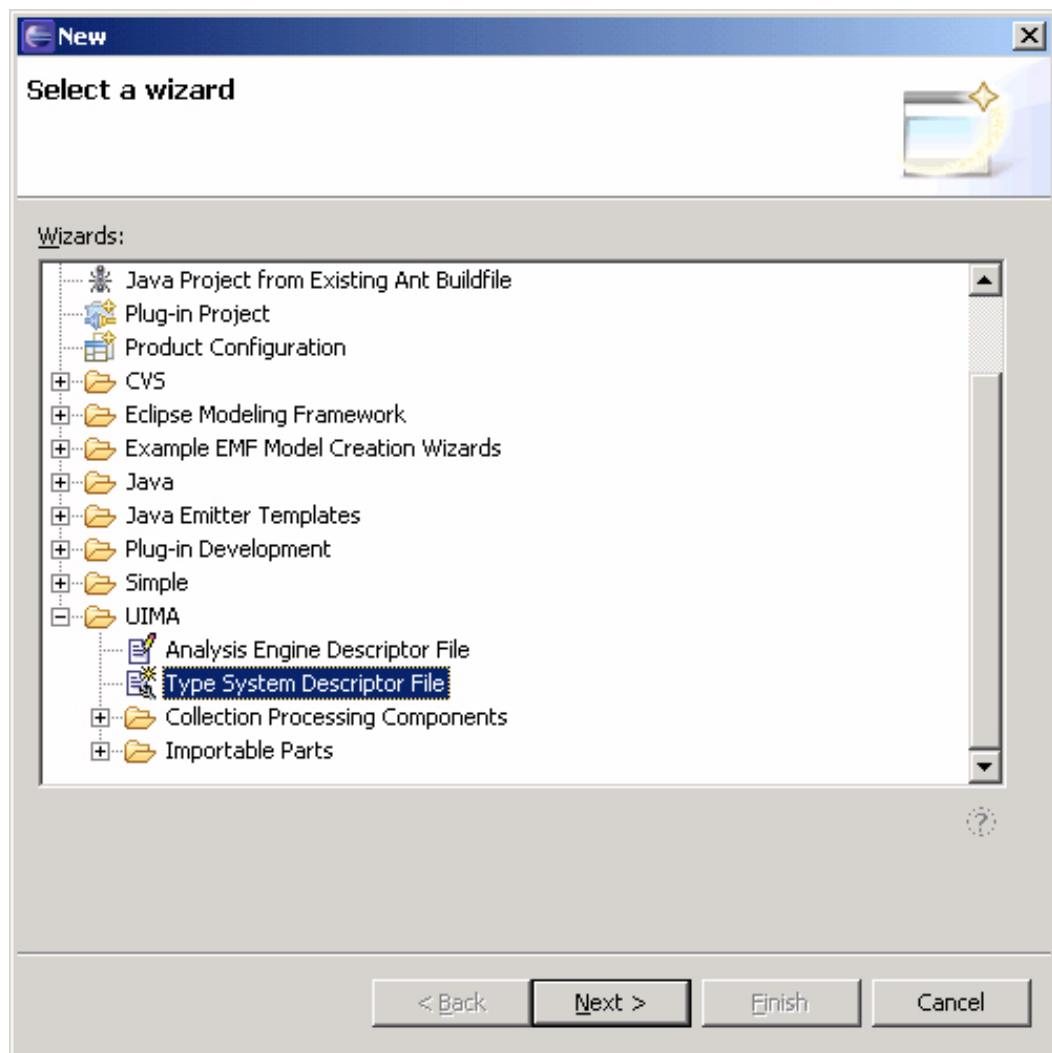
When you find a product number in the document, you want to record the actual information about it, and also the product line to which it belongs. The product line is a "feature" of the product number.

Create the type descriptor file

Start by creating the type descriptor file, which includes all of the information about the structure of the type. The UIMA SDK includes an Eclipse plug-in for editing descriptors. To create a new file in any editor, do the following:

1. In the Package Explorer pane, click the plus sign (+) to expand the uima_examples project.
2. Right-click the descriptors folder and choose New > Other.
3. Expand the UIMA node and select Type System Descriptor File, as shown in [Figure 5](#).

Figure 5. Create a new type descriptor



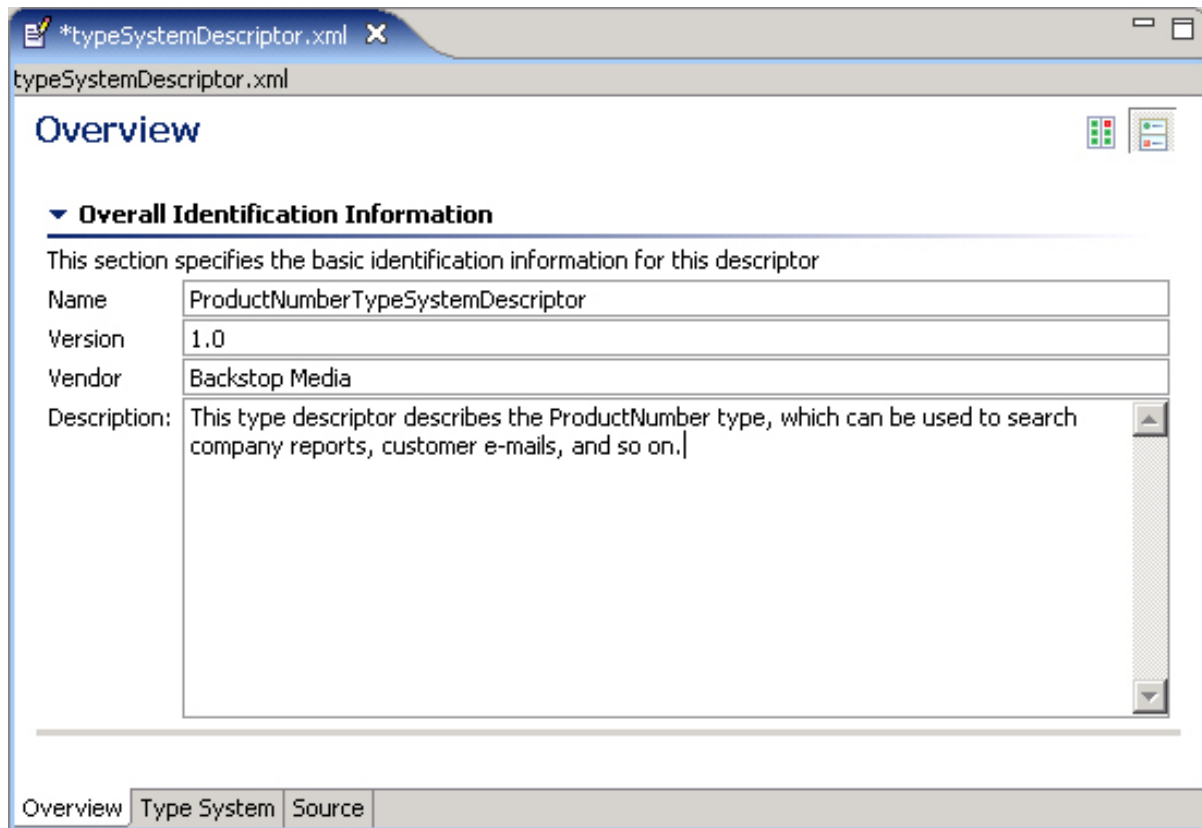
4. Click **Next**.
5. Click **Finish**.

Eclipse now creates the file and opens it in the editor.

Enter the basic information

The first step in creating a new type descriptor is to enter the basic information about it in the editor, as shown in [Figure 6](#).

Figure 6. Entering basic information about the new type descriptor



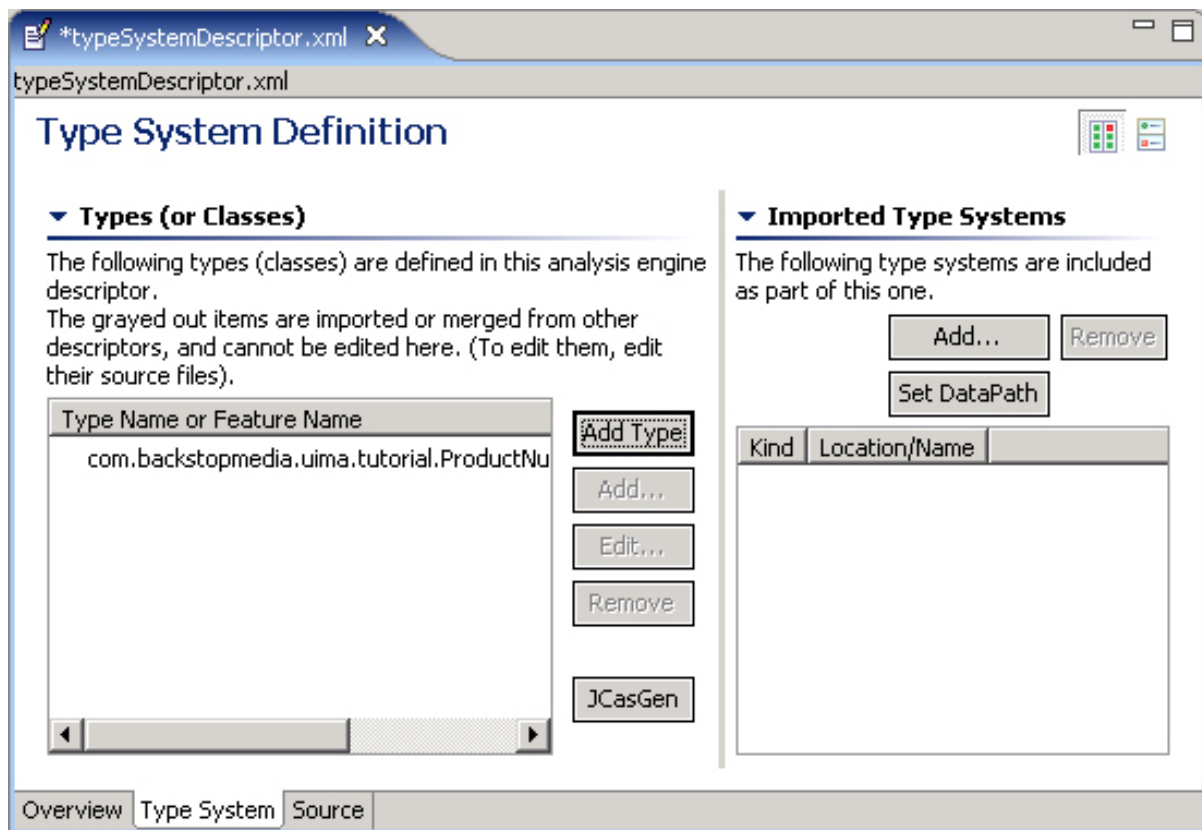
The actual information here is fairly arbitrary. Just make sure it's descriptive so you know what it means. Next you add the actual type.

Add a new type

Add the new `ProductNumber` type as follows:

1. Click the **Type System** tab, shown in [Figure 7](#), at the bottom of the editor pane.
2. Click **Add Type**.
3. Add the type name. Type names are distinguished by name spaces that have the same syntax as Java packages. For example, you can enter the type as `com.backstopmedia.uima.tutorial.ProductNumber`.
4. Your type will be an extension of the `uima.tcas.Annotation` supertype, so leave that as is. In future projects, you can actually create types by extending other types.
5. Click **OK**.

Figure 7. Type System tab



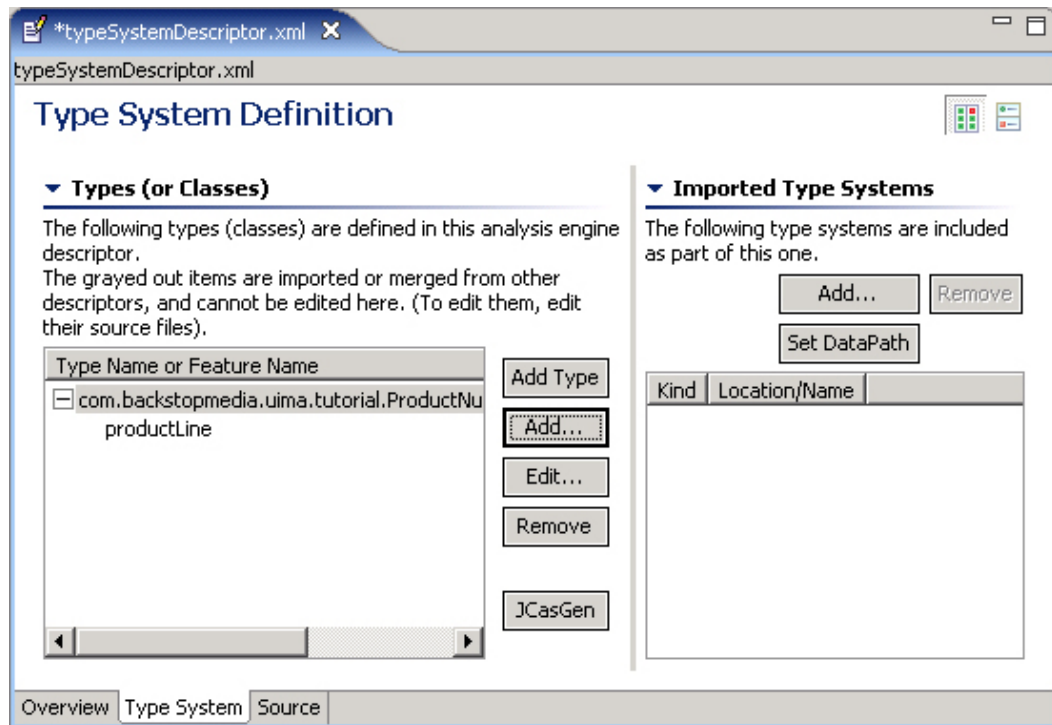
Now you can add the product line feature to this type.

Add a feature

The product line is considered a "feature" of the type, just as the engine size is considered a feature of a car. To make it available to the system, you will need to edit the type, as follows:

1. Click the new class to highlight it. This also makes the **Add** button active.
2. Click the **Add** button.
3. Add the Feature Name. In this case, the feature name is productLine.
4. The Range Type represents the type of data this feature can hold. For example, you put string data into the productLine. Click **Browse** and select uima.cas in the bottom of the resulting window, and String in the top. Click **OK**.
5. Click **OK** to save the new feature. See [Figure 8](#).

Figure 8. Adding a feature



6. Save the file by pressing <ctrl>-S.

You now have a complete, if simple, type descriptor.

The XML source

All this is just a more convenient way to create an XML file. The XML itself is not complicated, and you can also create it with a simple text editor. If you click the Source tab, you can see the actual XML that represents what you just created. See [Listing 2](#).

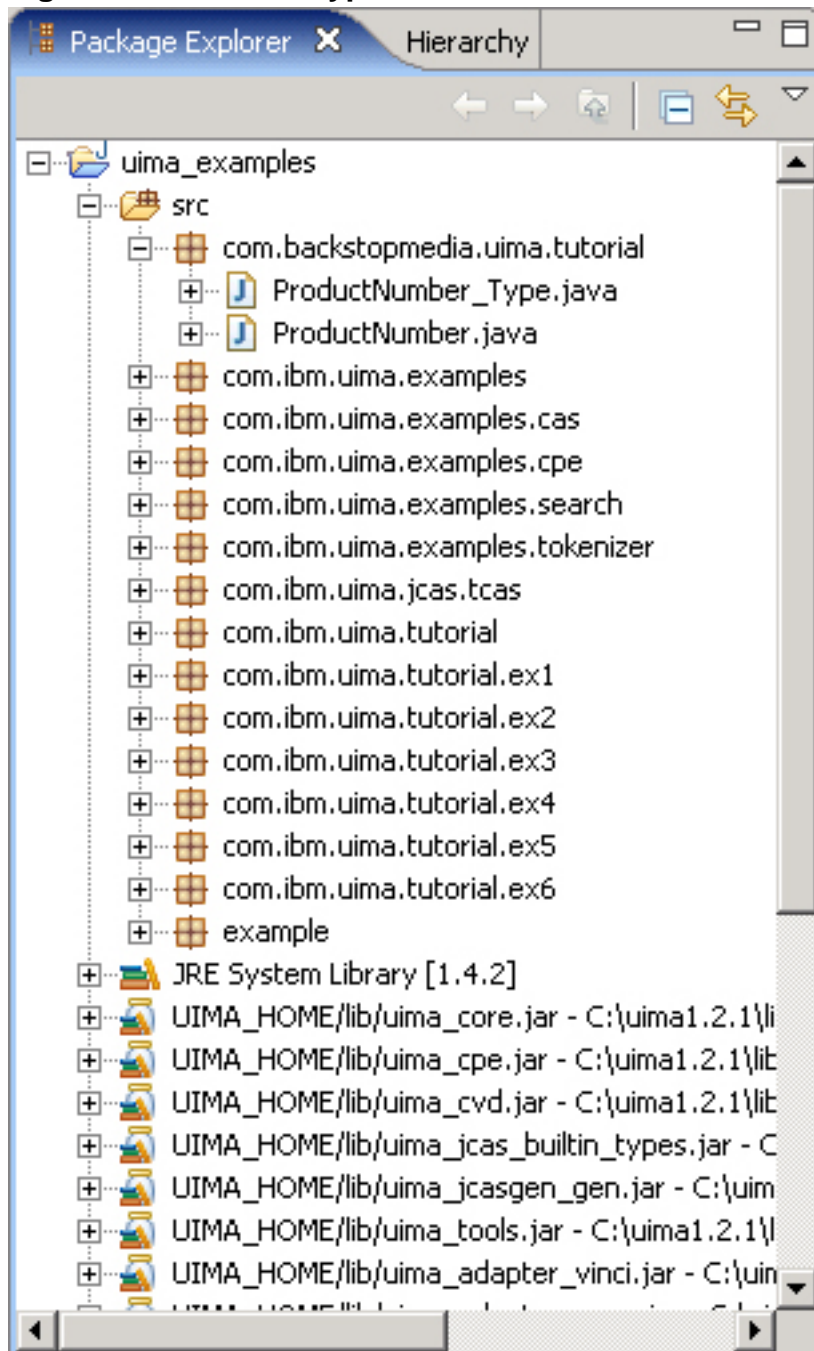
Listing 2. The XML source file

```
<?xml version="1.0" encoding="UTF-8"?>
<typeSystemDescription xmlns="http://uima.watson.ibm.com/resourceSpecifier">
  <name>ProductNumberTypeSystemDescriptor</name>
  <description>This type descriptor describes the ProductNumber type, which
  can be used to search company reports, customer e-mails, and so on.</description>
  <version>1.0</version>
  <vendor>Backstop Media</vendor>
  <types>
    <typeDescription>
      <name>com.backstopmedia.uima.tutorial.ProductNumber</name>
      <description/>
      <supertypeName>uima.tcas.Annotation</supertypeName>
      <features>
        <featureDescription>
          <name>productLine</name>
          <description/>
          <rangeTypeName>uima.cas.String</rangeTypeName>
        </featureDescription>
      </features>
    </typeDescription>
  </types>
</typeSystemDescription>
```

Generate the Java classes

As long as you have not disabled the functionality, Eclipse automatically generates the Java classes when you save the type descriptor. You can tell whether that occurred by expanding the `uima_examples/src` node in the Package Explorer pane and looking for your type. See [Figure 9](#).

Figure 9. Generated types



You should see two Java files -- `ProductNumber.java` and

ProductNumber_Type.java.

If you don't see these classes, you can generate them manually by clicking on the Type System tab in the typeSystemDescriptor.xml editor pane, and then clicking the JCasGen button. JCasGen is actually a separate utility that you can use to generate Java class files from type descriptors independently of Eclipse.

Let's take a quick look at these classes.

ProductNumber.java

The `ProductNumber` class is used for setting general Annotation information, such as the start and end position of the Annotation, as shown in [Listing 3](#).

Listing 3. ProductNumber.java

```
package com.backstopmedia.uma.tutorial;

import com.ibm.uma.jcas.impl.JCas;
import com.ibm.uma.jcas.cas.TOP_Type;

import com.ibm.uma.jcas.tcas.Annotation;

public class ProductNumber extends Annotation {

    public final static int typeIndexID = JCas.getNextIndex();

    public final static int type = typeIndexID;

    public int getTypeIndexID() {return typeIndexID;}

    protected ProductNumber() {}

    public ProductNumber(int addr, TOP_Type type) {
        super(addr, type);
        readObject();
    }

    public ProductNumber(JCas jcas) {
        super(jcas);
        readObject();
    }

    public ProductNumber(JCas jcas, int begin, int end) {
        super(jcas);
        setBegin(begin);
        setEnd(end);
        readObject();
    }

    private void readObject() {}

    public String getProductLine() {
        if (ProductNumber_Type.featoKtst &&
            ((ProductNumber_Type)jcasType).casFeat_productLine == null)
            JCas.throwFeatMissing("productLine",
                "com.backstopmedia.uma.tutorial.ProductNumber");
        return jcasType.ll_cas.ll_getStringValue(addr,
            ((ProductNumber_Type)jcasType).casFeatCode_productLine);
    }

    public void setProductLine(String v) {
        if (ProductNumber_Type.featoKtst &&
            ((ProductNumber_Type)jcasType).casFeat_productLine == null)
            JCas.throwFeatMissing("productLine",
                "com.backstopmedia.uma.tutorial.ProductNumber");
    }
}
```

```

        jcasType.ll_cas.ll_setStringValue(addr,
            ((ProductNumber_Type)jcasType).casFeatCode_productLine, v);
    }
}

```

As you can see, these are just general utility methods. You can certainly go in and add additional methods, customize methods, and so on, if you like. The `setBegin()` and `setEnd()` methods are inherited from the `Annotation` class.

ProductNumber_Type.java

The `ProductNumber_Type` class shows methods more specific to the internal workings of the UIMA framework. See [Listing 4](#).

Listing 4. The `ProductNumber_Type` class

```

package com.backstopmedia.uima.tutorial;

import com.ibm.uima.jcas.impl.JCas;
import com.ibm.uima.cas.impl.CASImpl;
import com.ibm.uima.cas.impl.FSGenerator;
import com.ibm.uima.cas.FeatureStructure;
import com.ibm.uima.cas.impl.TypeImpl;
import com.ibm.uima.cas.Type;
import com.ibm.uima.cas.impl.FeatureImpl;
import com.ibm.uima.cas.Feature;
import com.ibm.uima.jcas.tcas.Annotation_Type;

public class ProductNumber_Type extends Annotation_Type {

    protected FSGenerator getFSGenerator() {return fsGenerator;};

    private final FSGenerator fsGenerator =
        new FSGenerator() {
            public FeatureStructure createFS(int addr, CASImpl cas) {
                if (instanceOf_Type.useExistingInstance) {
                    // Return eq fs instance if already created
                    FeatureStructure fs =
                        instanceOf_Type.jcas.getJfsFromCaddr(addr);
                    if (null == fs) {
                        fs = new ProductNumber(addr, instanceOf_Type);
                        instanceOf_Type.jcas.putJfsFromCaddr(addr, fs);
                        return fs;
                    }
                    return fs;
                } else return new ProductNumber(addr, instanceOf_Type);
            }
        };

    public final static int typeIndexID = ProductNumber.typeIndexID;

    public final static boolean featOkTst =
        JCas.getFeatOkTst(
            "com.backstopmedia.uima.tutorial.ProductNumber");

    final Feature casFeat_productLine;

    final int    casFeatCode_productLine;

    public String getProductLine(int addr) {
        if (featOkTst && casFeat_productLine == null)
            JCas.throwFeatMissing("productLine",
                "com.backstopmedia.uima.tutorial.ProductNumber");
        return ll_cas.ll_getStringValue(addr, casFeatCode_productLine);
    }

    public void setProductLine(int addr, String v) {
        if (featOkTst && casFeat_productLine == null)
            JCas.throwFeatMissing("productLine",
                "com.backstopmedia.uima.tutorial.ProductNumber");
    }
}

```

```
        ll_cas.ll_setStringValue(addr, casFeatCode_productLine, v);
    }

    public ProductNumber_Type(JCas jcas, Type casType) {
        super(jcas, casType);
        casImpl.getFSClassRegistry()
            .addGeneratorForType((TypeImpl)this.casType, getFSGenerator());

        casFeat_productLine = jcas.getRequiredFeatureDE(casType,
            "productLine", "uima.cas.String", featOkTst);
        casFeatCode_productLine =
            (null == casFeat_productLine) ? JCas.INVALID_FEATURE_CODE :
            ((FeatureImpl)casFeat_productLine).getCode();
    }
}
```

Now you're ready to use the new type in the creation of an Annotator.

Section 6. Create the Annotator

The Annotator is the Java class that does the actual searching. Let's create it now.

Create the Annotator class

The purpose of the Annotator is to take the data and look for instances of a specific type. When it finds any, it makes a note of the start and end position of the data as well as the data itself. All of this information is stored in the Common Analysis Structure, or CAS object.

Technically speaking, an Annotator receives a CAS object, which includes the actual artifact to search and any Annotations that have already been made, such as the data, and inserts any additional Annotations back into the CAS object.

Start by creating the new Java class, as follows:

1. Right-click the src node in the Package Explorer pane.
2. Choose New > Class.
3. For simplicity's sake, choose a package name to match the namespace for your ProductNumber type.
4. Enter the class name as `ProductNumberAnnotator`. This is actually an arbitrary name, but it helps to make it something descriptive.
5. Click **Finish**.

The basic Annotator

Add the code shown in [Listing 5](#) to the `ProductNumberAnnotator.java` file.

Listing 5. The basic `ProductNumberAnnotator` class

```
package com.backstopmedia.uima.tutorial;

import java.util.regex.Matcher;
import java.util.regex.Pattern;

import com.ibm.uima.analysis_engine.ResultSpecification;
import com.ibm.uima.analysis_engine.annotator.AnnotatorProcessException;
import com.ibm.uima.analysis_engine.annotator.JTextAnnotator_ImplBase;
import com.ibm.uima.jcas.impl.JCas;

public class ProductAnnotator extends JTextAnnotator_ImplBase {

    public void process(JCas aJCas, ResultSpecification aResultSpec)
        throws AnnotatorProcessException {

        String txt = aJCas.getDocumentText();

    }
}
```

Starting with the class definition, notice that the Annotator extends the `JTextAnnotator_ImplBase` class. This class handles any additional methods required beyond the `process()` method.

The `process()` method is where all the magic happens. It receives a Java version of the CAS object, which contains the data to search, and an optional `ResultSpecification`, which you do not need for this tutorial. The first thing the `process()` method does is to obtain a string representation of the actual text to be searched by requesting it from the CAS object that has been passed in as a parameter.

It processes this string using regular expressions.

Using regular expressions in Java

Unfortunately, a thorough discussion of regular expressions is beyond the scope of this tutorial, but understand this: a regular expression is a pattern, such as "three capital letters, a dash, and then five numbers." In Java, you search text by matching the patterns shown in [Listing 6](#).

Listing 6. Adding regular expressions

```
...
    public void process(JCas aJCas, ResultSpecification aResultSpec)
        throws AnnotatorProcessException {

        String txt = aJCas.getDocumentText();

        Pattern UniverseProductNumbers =
            Pattern.compile("\\b[U][A-Z][A-Z]-\\d\\d\\d\\d\\d\\b");
        Matcher matcher = UniverseProductNumbers.matcher(txt);
        int pos = 0;
        while (matcher.find(pos)) {

            pos = matcher.end();

        }
    }
}
```

```

    Pattern BeyondProductNumbers =
        Pattern.compile("\\b[B][A-Z][A-Z]-\\d\\d\\d\\d\\b");
    matcher = BeyondProductNumbers.matcher(txt);
    pos = 0;
    while (matcher.find(pos)) {
        pos = matcher.end();
    }
}

```

In this case, you match against two patterns. Starting with the first, `UniverseProductNumbers`, you create the `Pattern` object by compiling the appropriate regular expression. (See [Resources](#) for links to more information on the actual regular expressions themselves.) Once you've obtained the `Pattern`, use it to request a `Matcher` that compares the pattern to the text you actually want to search.

The `find()` method starts at the given position and returns true if it has found a match. In that case, you are currently resetting the starting position to the end of the found pattern and looping through once again.

You do this twice; once for each pattern, and you can see what to actually do with the found pattern next.

Create the Annotation

Once you've located instances of the pattern, you need to create an `Annotation` and add it to the CAS object. See [Listing 7](#).

Listing 7. Create the Annotations

```

...
    Pattern UniverseProductNumbers =
        Pattern.compile("\\b[U][A-Z][A-Z]-\\d\\d\\d\\d\\d\\d\\b");
    Matcher matcher = UniverseProductNumbers.matcher(txt);
    int pos = 0;
    while (matcher.find(pos)) {
        ProductNumber productNumberAnnotation =
            new ProductNumber(aJCas);

        productNumberAnnotation.setProductLine("Universe");
        productNumberAnnotation.setBegin(matcher.start());
        productNumberAnnotation.setEnd(matcher.end());

        productNumberAnnotation.addToIndexes();
        pos = matcher.end();
    }

    Pattern BeyondProductNumbers =
        Pattern.compile("\\b[B][A-Z][A-Z]-\\d\\d\\d\\d\\d\\b");
    matcher = BeyondProductNumbers.matcher(txt);
    pos = 0;
    while (matcher.find(pos)) {
        ProductNumber productNumberAnnotation =
            new ProductNumber(aJCas);

        productNumberAnnotation.setProductLine("Beyond");
        productNumberAnnotation.setBegin(matcher.start());
        productNumberAnnotation.setEnd(matcher.end());

        productNumberAnnotation.addToIndexes();
        pos = matcher.end();
    }
}

```



```
}  
}  
}
```

Each time the Annotator finds a match on the pattern, it creates a new Annotation in the CAS object. Remember, `ProductNumber` extends `Annotation`, so in addition to the `setProductLine()` method, you also have access to the `setBegin()` and `setEnd()` methods.

Once you've created the `ProductNumber` Annotation, you must add it to the CAS indexes, or you won't be able to find it later.

Now you have a working Annotator, so you need to turn it into a full-fledged Analysis Engine.

Section 7. Create the Analysis Engine descriptor

The Annotator is the heart of a basic Analysis Engine, and the descriptor is its spine. In this section you build that spine.

Create a new descriptor

The process of creating a new Analysis Engine descriptor is similar to that of creating a new type system descriptor, as follows:

1. In the Package Explorer pane, right-click the descriptors node and choose **New > Other**.
2. Expand the UIMA node and select Analysis Engine Descriptor File. Click **Next**.
3. Set the Parent Folder to the `/uima_examples/descriptors` folder.
4. Enter a new filename, such as `ProductNumberAEDDescriptor.xml`. Again, this name is completely arbitrary, but should be descriptive.
5. Click **Finish**.

The new file should open in the Component Descriptor Editor.

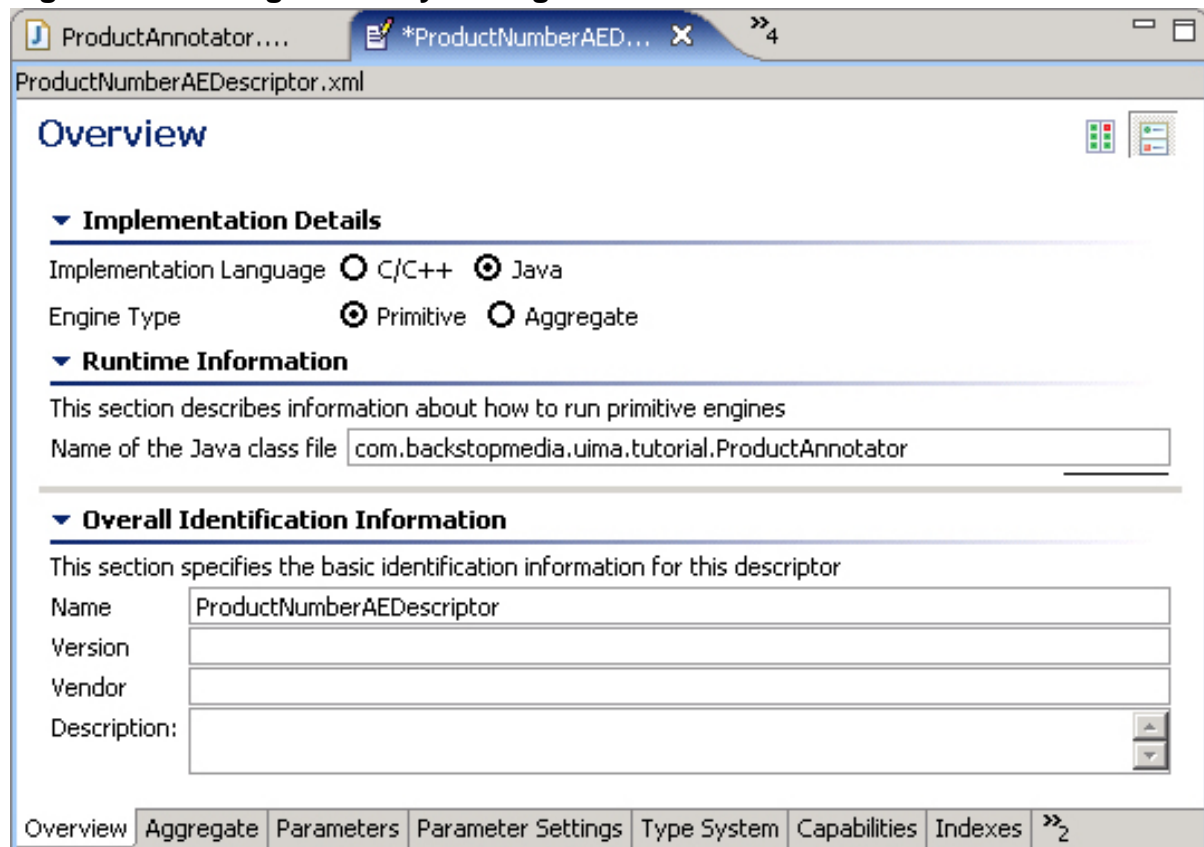
Assign the Annotator

Once you've created the new descriptor, you need to tell it what kind of Analysis Engine you are trying to create. Enter that information in the Overview tab. See

Figure 10.

1. Make sure that the Implementation Language is set to Java. You also have the ability to build components in C or C++, but I don't cover that here.
2. Make sure that the Engine Type is set as Primitive. Aggregate engines enable you to chain Analysis Engines together, feeding the results from one as the input to the other. Building such an engine is beyond the scope of this tutorial, but it is not tremendously complex (See [Resources](#) and the Users Guide for more information).
3. Under Runtime Information, add the fully-qualified name of the Annotator class you just created.
4. Finally, enter any additional information, such as the version, vendor, and so on.

Figure 10. Linking the Analysis Engine to the Annotator



Next you need to tell the Analysis Engine what types of data it will be working with.

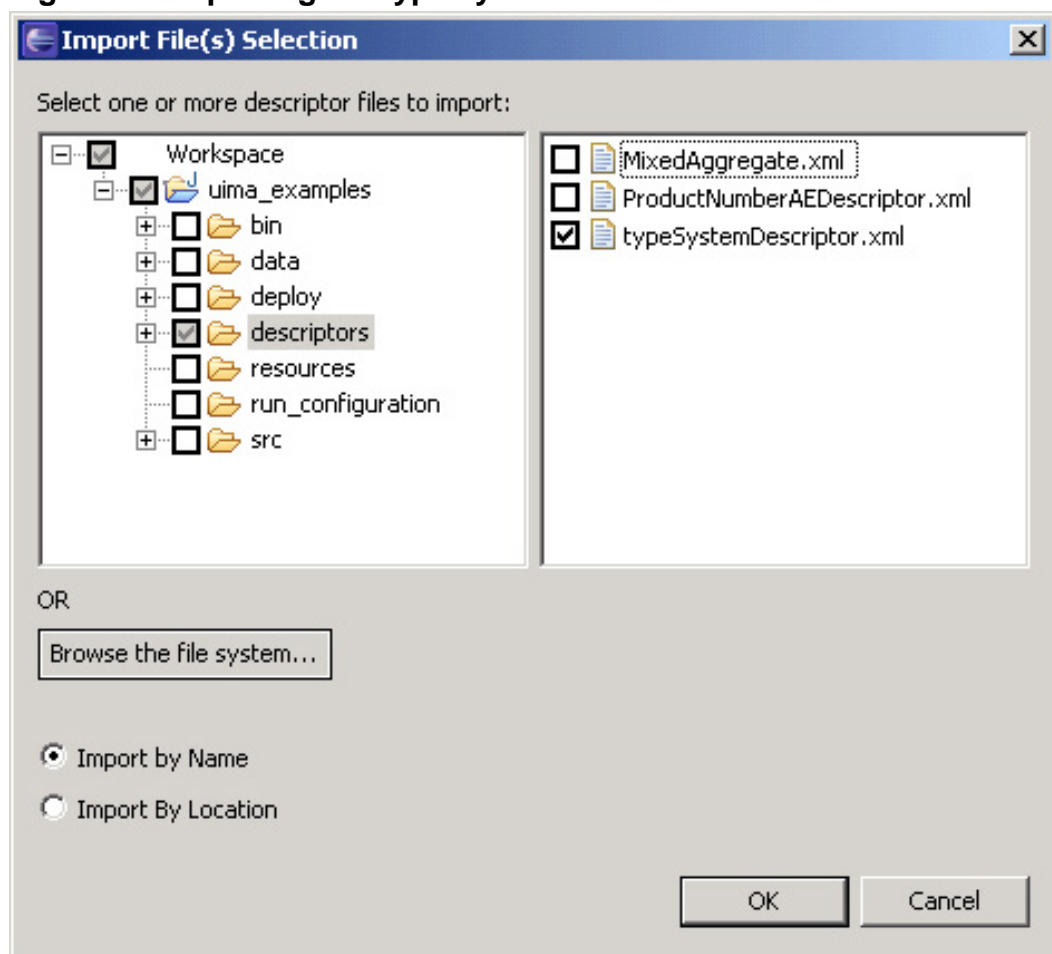
Setting types

You do have the option of creating types directly during this process, but instead you

can simply import the types you've already created. Execute the following steps:

1. Click the **Type System** tab at the bottom of the component editor pane.
2. Click the **Set DataPath** button under Imported Type Systems.
3. Set the data path to the absolute location of the descriptors directory, in other words, C:\uima1.2.1\docs\examples\descriptors.
4. Also under Imported Type Systems, click **Add**.
5. Expand the node on the left-hand side and click the descriptors folder (see [Figure 11](#)). In the right hand panel, select the check box next to your type descriptor file.

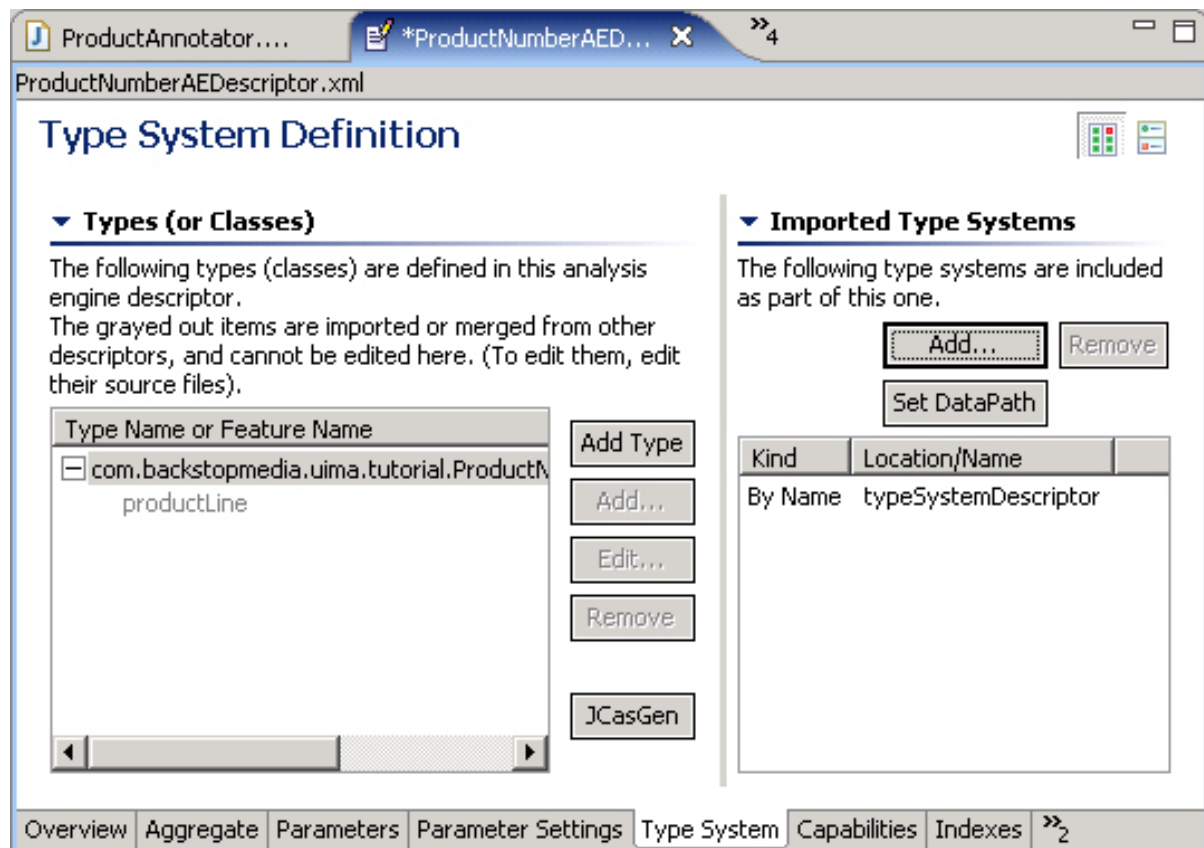
Figure 11. Importing the type system



6. Click **OK**.

The resulting window, shown in [Figure 12](#), should show you the `ProductNumber` type you created earlier.

Figure 12. Importing the Type System



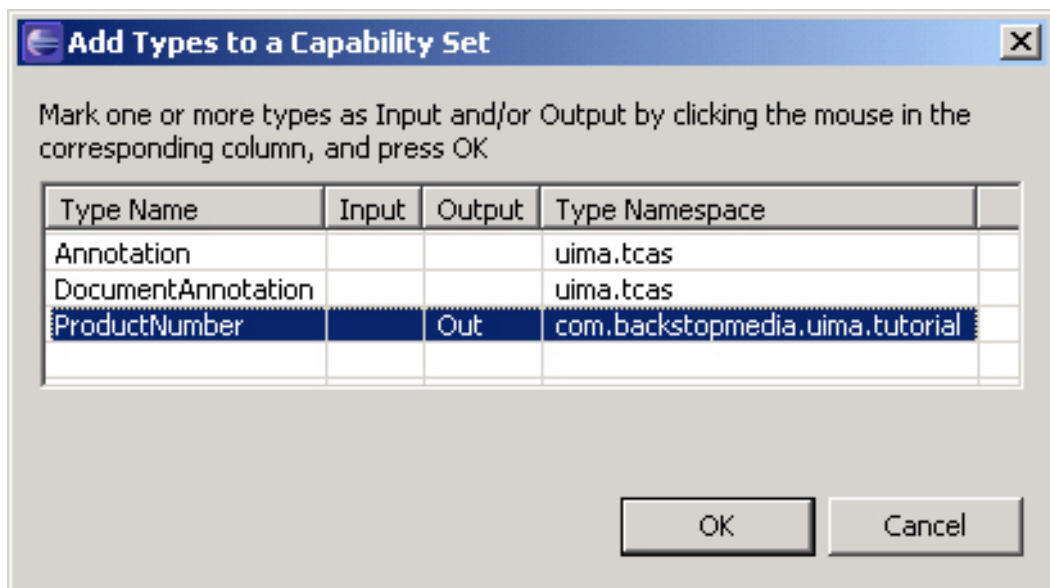
Now you just have to tell the Analysis Engine what kind of data to expect and to produce.

Capabilities

The Capabilities tab enables you to determine the input and output for the Analysis Engine. Perform the following steps:

1. Click the **Capabilities** tab at the bottom of the component editor pane.
2. Select the existing Capability by clicking the first line.
3. Click **Add Type**.
4. In the resulting window, you can toggle Input and Output for each of the existing types. Click the Output column for `ProductNumber` to turn it on. See [Figure 13](#).

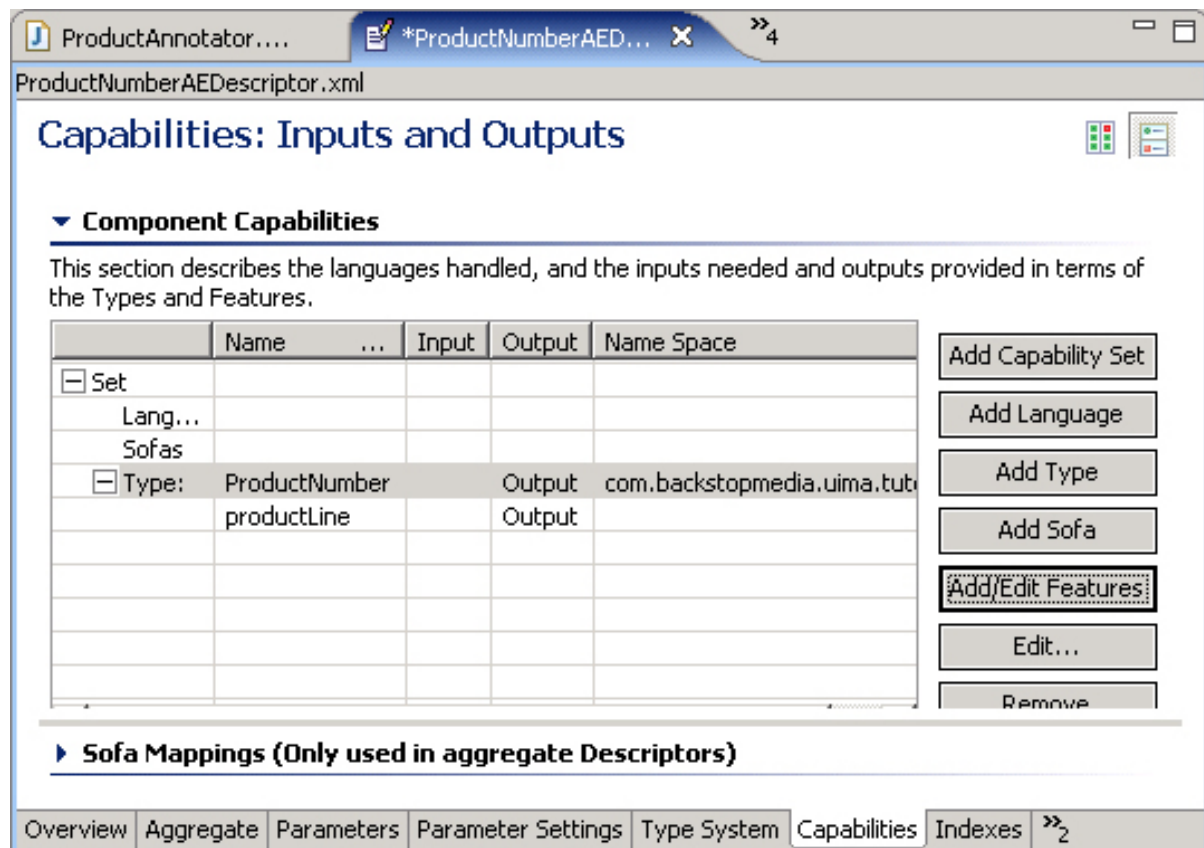
Figure 13. Specifying input and output capabilities



5. By default, all features out of the added type are set to be output. To restrict this to just the `productLine` feature, select All Features > Add/Edit Features.
6. Then in the resulting window, turn off output for All Features and turn it on for `productLine`.
7. Click **OK**.

You should see a window similar to the one shown in [Figure 14](#).

Figure 14. Capabilities in place



Save the descriptor file by pressing <ctrl>-S.

Now you're ready to test the Analysis Engine.

Section 8. Test the Analysis Engine

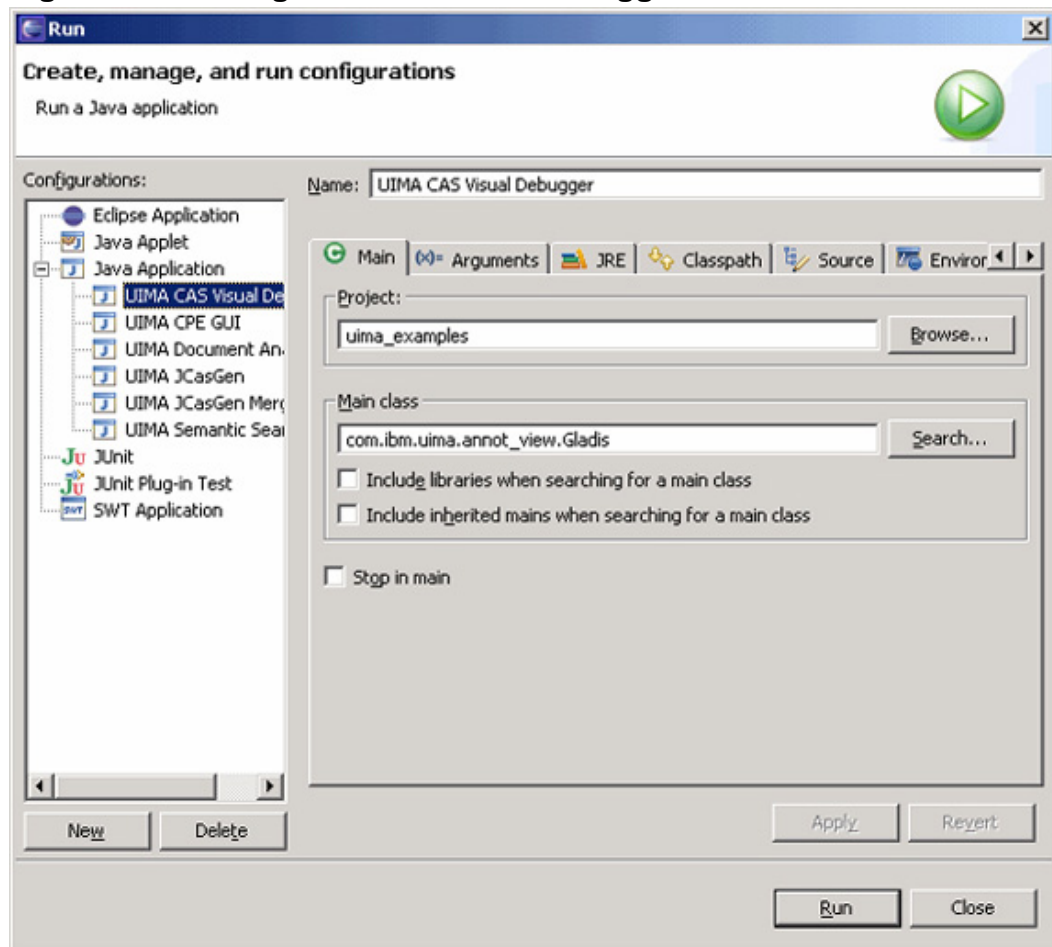
Ultimately, you run your Analysis Engine from within your application, but it helps to be able to test it independently. Fortunately, the UIMA SDK includes tools for just such an occasion.

Start the CAS Visual Debugger

The UIMA SDK actually comes with several tools you can use to test your Analysis Engine, but to really see what's going on, use the CAS Visual Debugger, which lets you see exactly what Annotations are being added to the CAS object. To start the debugger, do the following:

1. In the main Eclipse window, select Run > Run.
2. Expand the Java node and select UIMA CAS Visual Debugger. See

Figure 15.

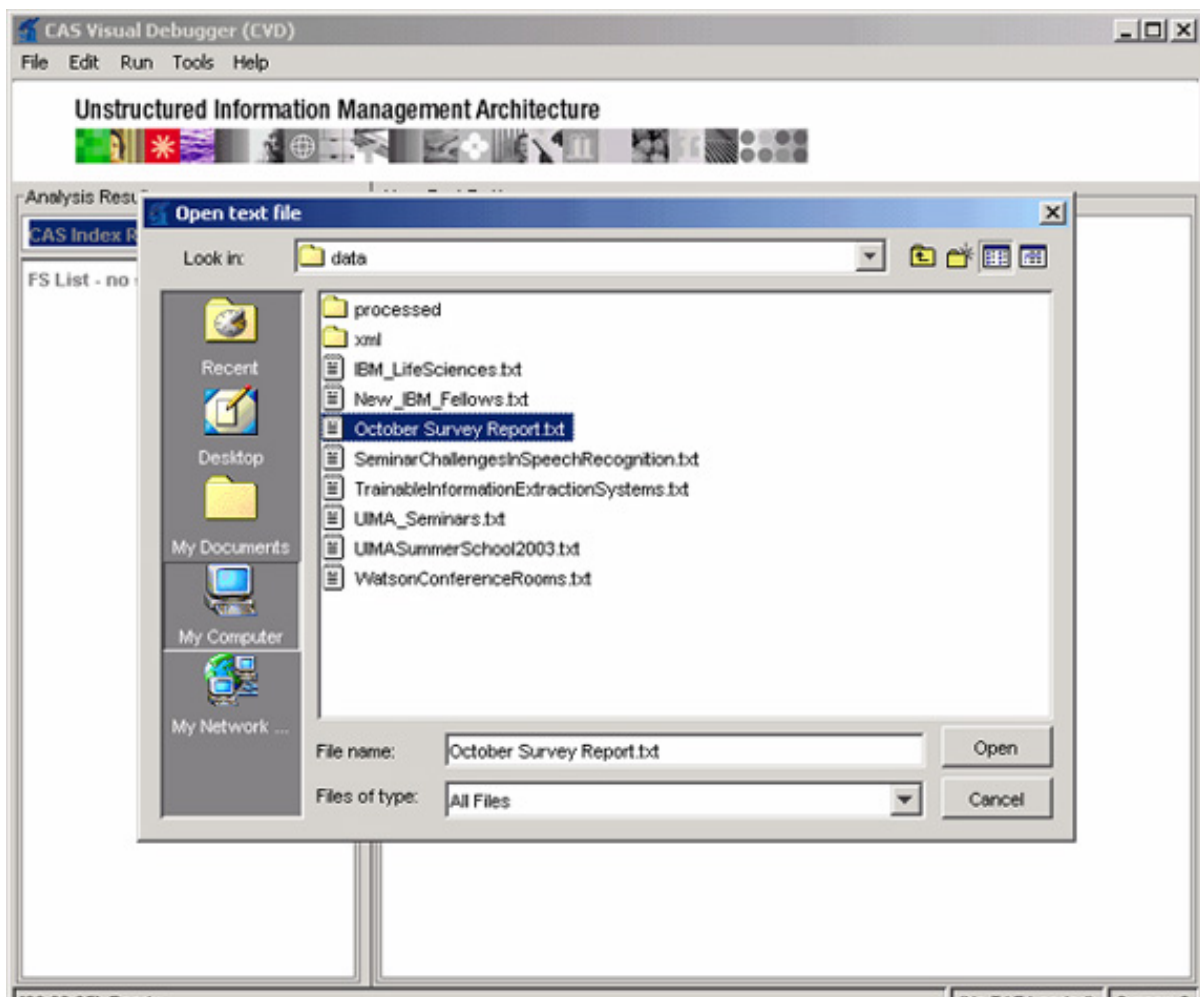
Figure 15. Starting the CAS Visual Debugger

3. Click Run

Specify the document

The next step is to specify the text you want to analyze. The CAS Visual Debugger, shown in [Figure 16](#), lets you enter text directly or load a text file. Click **File > Open text file** and specify the text version of the company report you want to analyze.

Figure 16. Specifying the document to analyze



Click **Open** to load the document.

Specify the Analysis Engine descriptor

Once you've got the text, you need to specify the Analysis Engine. Before you can do that, however, you need to compensate for some classpath issues. Even though you set the data path to point to the descriptors folder when you imported the type system into the descriptor, the CAS Visual Debugger won't know to look for it there, looking instead in the resources directory.

To solve the problem, copy the `typeSystemDescriptor.xml` to the resources directory. Right-click the file and choose Copy, then right-click the resources folder and choose Paste.

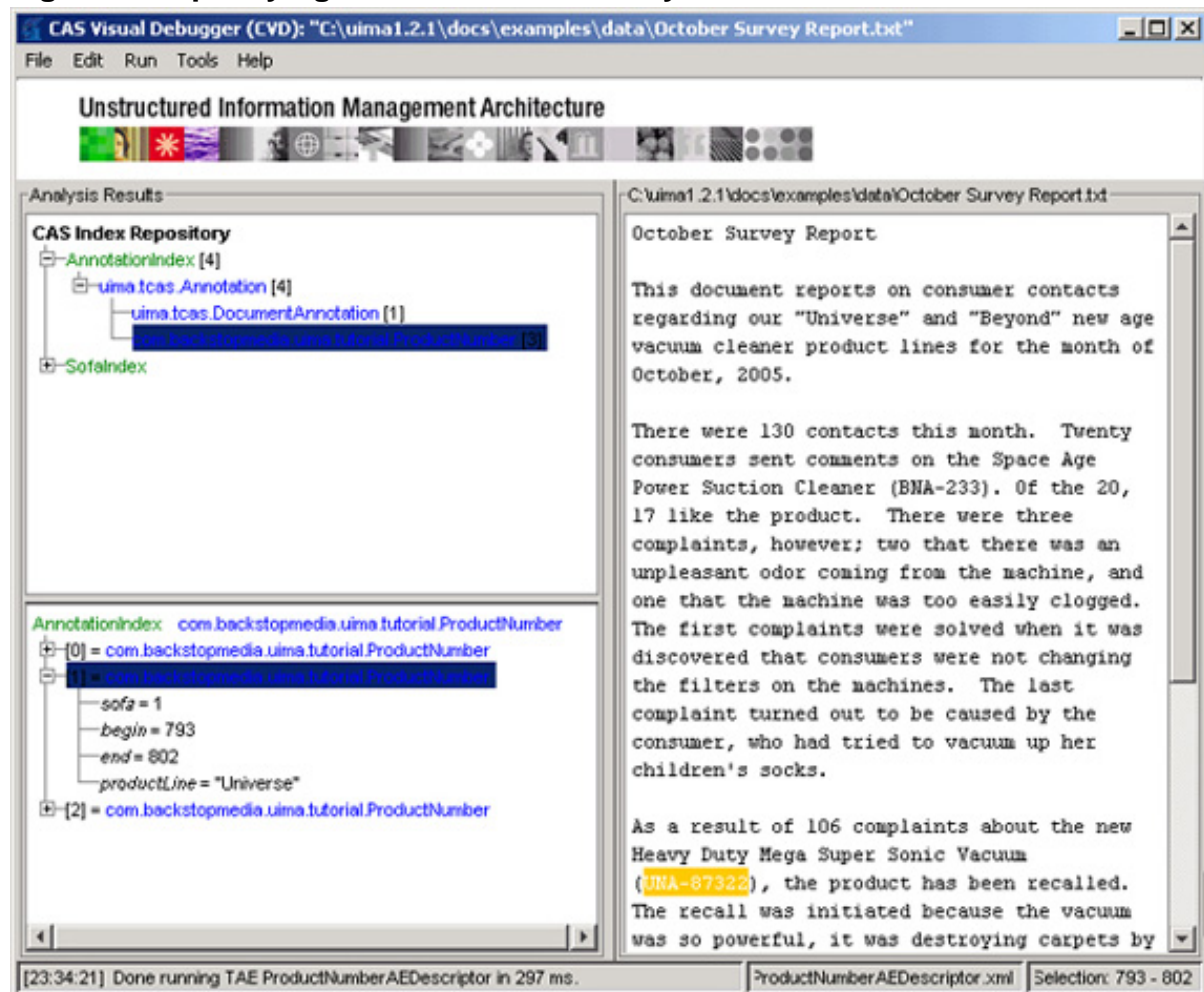
Now you can load the Analysis Engine as follows:

1. In the CAS Visual Debugger, select Run > Load TAE. (TAE stands for Text Analysis Engine.)
2. Navigate to the `ProductNumberAEDescriptor.xml` file and click **Open**.

Run the debugger

Finally, it's time to test the Analysis Engine. In the CAS Visual Debugger window, select **Run > Run ProductNumberAEDescriptor**. The results appear in the upper left-hand section of the window, under **Analysis Results**. See [Figure 17](#). If this pane is too small to see the results, you can drag its borders to expand it.

Figure 17. Specifying the document to analyze



Expand the `AnnotationIndex` node until you see the `ProductNumber` Annotations. With this document, there should be three. (The `DocumentAnnotation` Annotation refers to the document itself.) If you select the `ProductNumber` Annotation, you see all of the Annotations of that type in the lower pane.

Click each of the Annotations to see them highlighted in the actual document. You can also expand the Annotation to see its features, such as its starting and ending position, and the `productLine`.

Now that you know your Analysis Engine works, it's time to incorporate it into the actual application.

Section 9. Create an application

At this point you leave the realm of creating classes and components for the UIMA and begin creating applications that simply use those classes and components.

Create the class

You can create a plain old Java class that loads the Analysis Engine, instructed to process the document, and then extracts information from the resulting CAS object, outputting it to the command line. That might not sound very impressive, but it is the heart of what any UIMA application does: process the data and examine the results.

Start by creating the new Java class:

1. In the Package Explorer pane, right-click the src folder and select New > Class.
2. Choose the same package you used for the `ProductNumber` class. This is not required; it is merely convenient.
3. Choose a class name. Because this is the final application, this name is truly arbitrary. I use `ProductFinder` in these examples.

Now let's add some code.

Create the class

Once Eclipse creates the new class, add the following code. See [Listing 8](#).

Listing 8. Creating the `ProductFinder` class

```
package com.backstopmedia.uima.tutorial;

import java.io.File;
import java.io.FileInputStream;
import com.ibm.uima.UIMAFramework;
import com.ibm.uima.analysis_engine.TextAnalysisEngine;
import com.ibm.uima.cas.FSIterator;
import com.ibm.uima.cas.FeatureStructure;
import com.ibm.uima.cas.Type;
import com.ibm.uima.cas.text.TCAS;
import com.ibm.uima.resource.ResourceSpecifier;
import com.ibm.uima.util.XMLInputSource;

public class ProductFinder {

    public static void main(String[] args) {

        try {
            File taeDescriptor = new File("C:\\uima1.2.1\\docs\\example
```

```
s\\descriptors\\ProductNumberAEDescriptor.xml");
    File inputFile = new File("C:\\uima1.2.1\\docs\\examples\\d
ata\\October Survey Report.txt");

    } catch(Exception e) {
        e.printStackTrace();
    }
}
}
```

Just as in the case of the CAS Visual Debugger, you need to specify the Analysis Engine descriptor and the file to be analyzed. Notice that these are absolute locations. Make sure to specify the actual locations is your installation.

Create the Analysis Engine

The first step is to actually create the Analysis Engine:

Listing 9. Creating the Analysis Engine

```
...
    try {
        File taeDescriptor = new File("C:\\uima1.2.1\\docs\\example
s\\descriptors\\ProductNumberAEDescriptor.xml");
        File inputFile = new File("C:\\uima1.2.1\\docs\\examples\\d
ata\\October Survey Report.txt");

        XMLInputSource in = new XMLInputSource(taeDescriptor);
        ResourceSpecifier specifier =
            UIMAFramework.getXMLParser().parseResourceSpecifier(in);

        TextAnalysisEngine tae = UIMAFramework.produceTAE(specifier);

        tae.destroy();
    } catch(Exception e) {
        e.printStackTrace();
    }
...
}
```

First create a new `XMLInputSource` to represent the descriptor file. From there, you can use the UIMA framework itself to read that file for information on the Analysis Engine you're trying to create. Once you have the specifier for the engine, you can use it to create the actual `TextAnalysisEngine` object.

Finally, when all is said and done, you should destroy the `TextAnalysisEngine` to free up the memory it occupied.

Process the document

Once you have the Analysis Engine, you can actually process the document, as shown in [Listing 10](#).

Listing 10. Processing the document

```
...
public class ProductFinder {

    public static void main(String[] args) {
```

```

    try {
        File taeDescriptor = new File("C:\\uima1.2.1\\docs\\example
s\\descriptors\\ProductNumberAEDescriptor.xml");
        File inputFile = new File("C:\\uima1.2.1\\docs\\examples\\d
ata\\October Survey Report.txt");

        XMLInputSource in = new XMLInputSource(taeDescriptor);
        ResourceSpecifier specifier =
            UIMAFramework.getXMLParser().parseResourceSpecifier(in);

        TextAnalysisEngine tae = UIMAFramework.produceTAE(specifier);
        TCAS tcas = tae.newTCAS();

        FileInputStream fis = new FileInputStream(inputFile);
        byte[] contents = new byte[(int)inputFile.length()];
        fis.read( contents );
        fis.close();
        String document =new String(contents );

        tcas.setDocumentText(document);
        tae.process(tcas);

        tae.destroy();
    } catch(Exception e) {
        e.printStackTrace();
    }
}

```

The first step is to obtain a new CAS object from the engine. It is this CAS object that will receive any Annotations discovered for this document. Next, get the contents of the actual file as a string.

Remember, the CAS object contains not just the Annotations, but the data itself. Set that data in the CAS object using the `setDocumentText()` method.

Finally, feed the newly populated CAS object to the `process()` method. This method searches the data and adds any Annotations to the CAS object.

That takes care of getting the data in. Now you have to get it out again.

Get the Annotations

Using the classes provided in the UIMA framework and the classes you generated earlier, you can directly access the information in the newly populated CAS object. See [Listing 11](#).

Listing 11. Retrieving the Annotations

```

...
public class ProductFinder {

    public static void printProducts(TCAS tcas) {

        Type productType = tcas.getTypeSystem()
            .getType("com.backstopmedia.uima.tutorial.ProductNumber");
        System.out.println("Type is " + productType.getName() + ".");
        System.out.println("It has " + productType.getNumberOfFeatures()
            + " features.");

        FSIterator iter =
            tcas.getAnnotationIndex(productType).iterator();
        while (iter.isValid()) {
            FeatureStructure fs = iter.get();

```

```

        ProductNumber annot = (ProductNumber)fs;

        iter.moveToNext();
    }
}

public static void main(String[] args) {
    try {
        ...
        tcas.setDocumentText(document);
        tae.process(tcas);

        printProducts(tcas);

        tae.destroy();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

First, in the `printProducts()` method, get a feel for how things are working by obtaining a reference to the definition of the `ProductNumber` type by extracting it from the CAS object. You can then output attributes such as the name and number of features to the command line.

But the real task is to see the data that's in the CAS object. To do that, you can obtain a `FSIterator` object to iterate over the feature structures present. Once you have that, you can loop through each item in the iterator, each time retrieving the current `FeatureStructure` and casting it as a `ProductNumber` Annotation.

If you run this application, you should see the following type information:

```

Type is com.backstopmedia.uima.tutorial.ProductNumber.
It has 4 features.

```

Get the Annotation features

Once you have the Annotations, you can get at their data, shown in [Listing 12](#).

Listing 12. Extracting the Annotation features

```

...
    FSIterator iter = tcas.getAnnotationIndex(productType).iterator();
    while (iter.isValid()) {
        FeatureStructure fs = iter.get();

        ProductNumber annot = (ProductNumber)fs;
        String coveredText = annot.getCoveredText();
        System.out.println("The product number is " + coveredText);
        System.out.println("The product line is " +
                           annot.getProductLine());
        System.out.println("Annotation found from " +
                           annot.getStart() + " to " + annot.getEnd() + ".");
        System.out.println("");

        iter.moveToNext();
    }
}
...

```


Remember when you created the `ProductNumber` class? It had getters and setters for the `productLine` and other information such as the start and end positions. Now you can make use of those methods to retrieve the actual information. You can also retrieve the data being annotated using the `getCoveredText()` method.

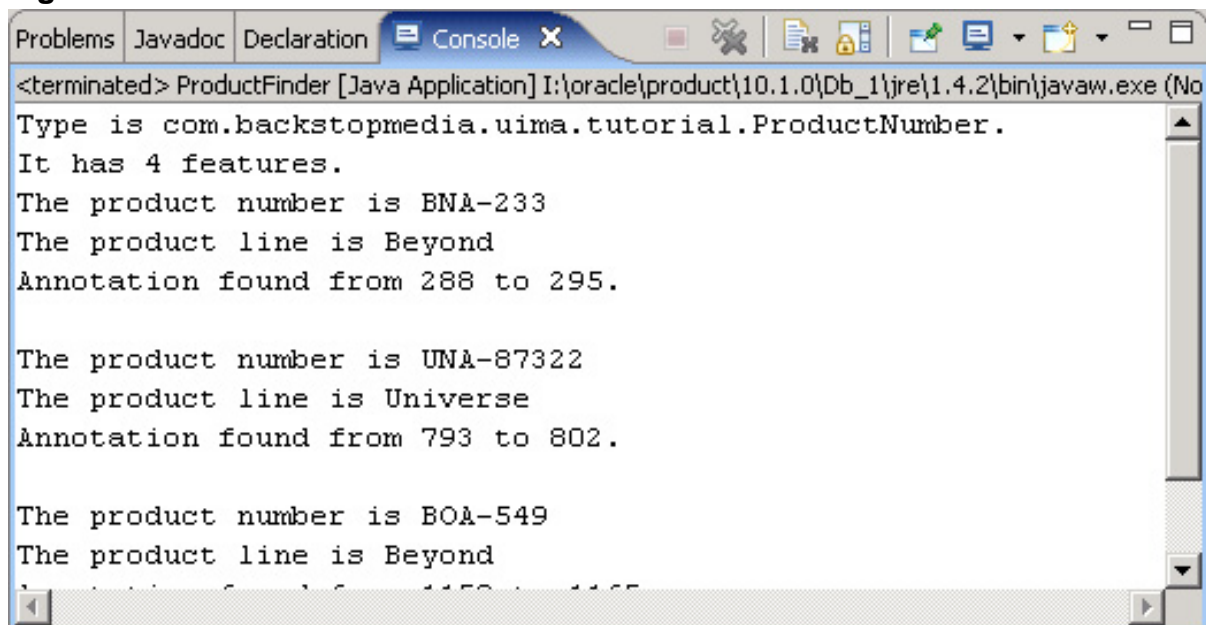
Now let's run it.

Run the application

Running an application in Eclipse is fairly straightforward. Right-click the appropriate `.java` file -- in this case, `ProductFinder.java` -- and choose `Run As > Java Application`.

The results appear in the Console window, which appears below the editors (unless you've moved it, of course). You should see results similar to [Figure 18](#).

Figure 18. The final results

The screenshot shows the Eclipse IDE's Console window. The title bar includes tabs for 'Problems', 'Javadoc', 'Declaration', and 'Console'. The console output is as follows:

```
<terminated> ProductFinder [Java Application] I:\oracle\product\10.1.0\Db_1\jre\1.4.2\bin\javaw.exe (No
Type is com.backstopmedia.uima.tutorial.ProductNumber.
It has 4 features.
The product number is BNA-233
The product line is Beyond
Annotation found from 288 to 295.

The product number is UNA-87322
The product line is Universe
Annotation found from 793 to 802.

The product number is BOA-549
The product line is Beyond
Annotation found from 1150 to 1165
```

If there are any run-time errors, they also appear in this window.

And that's all there is to it.

Section 10. Summary

Unstructured Information Management Architecture (UIMA) has the potential to discover a gold mine of information in the mountains of unstructured information most companies already have. Designed to provide a standard way of storing data known as Annotations, UIMA also provides a standard way of using individual pluggable components to perform each step.

In this tutorial, you learned how to create type system to define a particular kind of data, how to program an Annotator to look for that data, and how to turn it into an Analysis Engine that other applications can use. You also learned how to create an application that programmatically controls this process, as well as retrieving information once it's been stored.

In short, you have learned the foundation of the UIMA. Any application, no matter how complex, no matter what type of media you're searching, no matter how geographically diverse your systems, uses the same basic principles to accomplish its mission.

In Part 2 of this series, you will take the ProductNumber Analysis Engine and deploy it as a Web service, enabling UIMA applications to access it from anywhere.

Resources

Learn

- Get a general [overview of the UIMA](#).
- Join the [UIMA Forum](#) to get a sense of how other people are using it.
- For an in-depth look at all things UIMA related, be sure to read the [UIMA SDK Users Guide Reference](#).
- This tutorial talked about a simple pattern search, but where UIMA really shines is a [semantic search tool](#).
- Get an introduction to using the [Eclipse development platform](#).
- See the [Introduction to XML](#) tutorial for some background.
- Learn how to get more out of your regular expressions with the [Using regular expressions](#) tutorial. Eclipse hides the actual descriptors from you, but if you use UIMA for any length of time you are virtually guaranteed to need to have a look at the XML behind these editors.

Get products and technologies

- Download the [UIMA SDK](#) from alphaWorks.

About the author

Nicholas Chase

Nicholas Chase has been involved in Web site development for companies such as Lucent Technologies, Sun Microsystems, Oracle, and the Tampa Bay Buccaneers. Nick has been a high school physics teacher, a low-level radioactive waste facility manager, an online science fiction magazine editor, a multimedia engineer, an Oracle instructor, and the chief technology officer of an interactive communications company. He is the author of several books, including [XML Primer Plus](#) (Sams).