Simplifying UIMA Component Development and Testing with Java Annotations and Dependency Injection

Christophe Roeder, Philip V. Ogren, William A. Baumgartner Jr., and Lawrence Hunter

Center for Computational Pharmacology
University of Colorado Denver School of Medicine
MS 8303, PO Box 6511, Aurora, CO 80045 USA
Chris.Roeder@ucdenver.edu, philip@ogren.info,
William.Baumgartner@ucdenver.edu, Larry.Hunter@ucdenver.edu

Abstract. Developing within the Apache UIMA project framework presents challenges when writing and testing components in Java. Challenges stem from the relationship between the Java source code implementing the components and the corresponding UIMA XML descriptor files describing configuration and deployment settings. Java Annotations and Dependency Injection can be used to establish a stronger separation of concerns between framework integration and core component implementation, thus freeing the developer from commonly repeated tasks and allowing simplified development and testing.

1 Introduction

UIMA [1] components are defined by a pair of files: a source code file implementing component functionality and an XML descriptor file that stores component metadata consisting of component description information, e.g. information about input parameters (names, types, default values, etc.). At runtime, a third file specifies values for those input parameters: either the CPE descriptor when running in the CPM or an aggregate descriptor when running the AS. The current UIMA Java implementation burdens the developer with keeping the first two in synch, and it burdens the developer with writing and testing code to query the third. We propose two solutions: Java Annotations [2] to eliminate the duplication of component descriptor metadata, and a completed Dependency Injection implementation [3] to eliminate redundant configuration metadata extraction code.

2 Java Annotations

Java Annotations are a facility that appeared in Java 1.5. They allow metadata to be defined in the Java source code and retained during the compilation process.

As they are part of the Java class files, Java Annotations are retrievable at runtime via the Java Reflection API [4].

We propose that Java Annotations are a viable mechanism for representing the metadata currently defined in UIMA component descriptor XML files. Positioning the metadata within the source code would remove the metadata duplication that currently exists, thereby easing the burden of coding and testing on the developer. Under this scenario, changing an input parameter requires a change only to a single file, reducing the chance of creating inconsistencies. A UIMA version supporting Java Annotations would also support the current XML representation in support of older components. It is important to point out that the use of Java Annotations does not preclude the use of XML descriptor files in different components.

Java Annotations require a dedicated class to represent the annotation. Below is an example Annotation class that could be used to represent data about a file parameter. The functions in the Annotation declaration work as getters for the values of the Annotation's parameters.

```
@Retention(RetentionPolicy.RUNTIME)
public @interface FileParameterAnnotation {
        String description() default "no description";
        boolean multiValued() default false;
        boolean mandatory() default false;
        String defaultValue() default "";
}
```

With the Annotation defined, it could be used to mark a Java class member variable as an input parameter for a particular UIMA component. The Java syntax for invoking an Annotation is the @ symbol followed by the name of the Annotation, optionally including named values for its parameters. To minimize duplication, the name appearing in the deployment configuration is assumed to be the same as the variable name. Below is an example of how an input parameter indicating the path to an output file could be defined for a fictitious UIMA component.

```
public class Example_AE extends JCasAnnotator_ImplBase {
    @FileParameterAnnotation(description="output file path",  multiValued=false,
        mandatory=true)
        private String outputFilePath;
        public void setOutputFilePath(String s) { outputFilePath = s; }
        public void process() { FileWriter fr = new FileWriter(outputFilePath)); }
}
```

Once annotated, configuration information for a UIMA component can easily be extracted at runtime by reading from the Java Annotation using Reflection.

3 Dependency Injection

Dependency Injection [3] is a name given to the process of assembling a runtime configuration from a configuration file. Assembling the processing pipeline of components from a CPE descriptor is a form of Dependency Injection. The components are injected into the framework by a process driven by the CPE

descriptor. In UIMA the injection doesn't include setting the parameter values as in other implementations, rather the components query the UIMAContext, and this must be written for each component. A complete Dependency Injection implementation would have the framework set the parameters on the component objects, eliminating the need to write and test such code for each component.

With Dependency Injection, generic framework code searches the component for setter functions using Reflection based on the parameter name, and puts the configuration values obtained from the metadata into the component. For example, if you have the Example_AE from above, code in the depths of UIMA that creates it from the parsed deployment descriptor as represented in the UIMAContext object would look roughly like the following code. It would read the metadata (below), extracting the method name and the value (the path), inspect the annotator, and set the file path on it:

```
<casProcessor deployment="integrated" name="Example_AE">
<configurationParameterSettings><nameValuePair>
<name>OutputFilePath</name><value><string>/home/roederc/outputfile</string></value>
</nameValuePair></configurationParameterSettings>

JCasAnnotator ae = new Example_AE();
// ae.setOutputFilePath("/home/roederc/outputfile");
Class aeClass = Class.forName("Example_AE");
Method method = aeClass.getMethod("set" + "OutputFilePath", String.class);
method.invoke(ae, "/home/roederc/outputfile");
```

The three lines that follow the commented call to setInputFilePath() do essentially the same thing as the commented call, but in a way that can be driven by metadata. The function name is derived from the parameter name. Other code in UIMA would supply that string and this code would be used generically for any component. Since the framework is responsible for transferring the configuration data from the UIMAContext object to the component, the component does not have to do it, thereby reducing the developer of both writing and testing this code for each new component.

4 Related Work

UUTUC [5] is a set of convenience classes written to simplify UIMA testing. It allows a component and related UIMA infrastructure to be created without the overhead of a UIMA pipeline and related XML [6]. It does this by providing factory classes with methods that create various UIMA framework components from configuration data supplied in the Java code. UUTUC was written to reduce the dependence of test code on XML configuration data from deployment descriptors. Although the configuration metadata is not required in XML form components still need to be written with queries into the UIMAContext object to get configuration data when UUTUC alone is used.

5 Conclusion

Adding Java Annotations to and improving the Dependency Injection in UIMA is the logical next step after UUTUC in making developing and testing compo-

nents easier. The techniques described here not only reduce the XML required, but reduce the code and related testing required. Improving Dependency Injection is separate from XML issues, but reduces the amount of code involved in maintenance and testing when changing parameters. Java Annotations eliminate the need for XML component descriptors in the broader UIMAContext object. UUTUC, in contrast, while eliminating the need for component and deployment descriptors when testing, does not change the need for XML component descriptors during deployment under AS or CPM (UUTUC does not prevent AnalysisEngineDescription.toXML() from being called, so the descriptors can be generated in that environment). The component descriptors are used to accurately create deployment descriptors, so delivering a component without them hobbles deployment in these environments. Java Annotations provide an alternative way of representing this information, and with modification to some or all of CPM, AS, and tools used to create the deployment descriptors, can replace the XML form of component descriptors.

Moving UIMA in this direction would involve two major steps. Incorporating Java Annotations requires identifying and modifying code that reads the component descriptors. This does not have to preclude continued use of component descriptors for existing components. Improving Dependency Injection would require modification of the CPM and/or AS to inject values from the UIMAContext object to the components. Such a modification, like that for Java Annotations, does not preclude existing components from querying the UIMAContext object as they do today. The changes can be made so the benefits are there for the future, while not requiring change for existing components.

6 Acknowledgments

The authors thank Kevin Bretonnel Cohen, Helen L. Johnson, and Karin Verspoor for careful review.

This work was supported by NIH grants R01LM009254, R01GM083649, and R01LM008111 to Lawrence Hunter and T15LM009451 to Philip Ogren.

References

1. Apache UIMA project. http://incubator.apache.org/uima
2. Bloch, J. and others (2004): JSR 175: A metadata facility for Java programming language, Technical Report Final Release, Sun Microsystems Inc
3. Fowler, Martin http://www.martinfowler.com/articles/injection.html
4. http://java.sun.com/j2se/1.4.2/docs/api/java/lang/reflect/package-summary.html
5. Ogren, Philip and Bethard, Steven (2009): Building Test Suites for UIMA Components, in *Proceedings of the Workshop on Software Engineering, Testing and Quality Assurance for Natural Language Processing* (SETQA-NLP 2009)
6. UUTUC Getting Started Wiki Page http://code.google.com/p/uutuc/wiki/GettingStarted