Building Test Suites for UIMA Components

Philip V. Ogren
Center for Computational Pharmacology
University of Colorado Denver
Denver, CO 80217, USA
philip@ogren.info

Steven J. Bethard
Department of Computer Science
Stanford University
Stanford, CA 94305, USA
bethard@stanford.edu

Abstract

We summarize our experiences building a comprehensive suite of tests for a statistical natural language processing toolkit, ClearTK. We describe some of the challenges we encountered, introduce a software project that emerged from these efforts, summarize our resulting test suite, and discuss some of the lessons learned.

1 Introduction

We are actively developing a software toolkit for statistical natural processing called ClearTK (Ogren et al., 2008) [1], which is built on top of the Unstructured Information Management Architecture (UIMA) (Ferrucci and Lally, 2004). From the beginning of the project, we have built and maintained a comprehensive test suite for the ClearTK components. This test suite has proved to be invaluable as our APIs and implementations have evolved and matured. As is common with early-stage software projects, our code has undergone number of significant refactoring changes and such changes invariably break code that was previously working. We have found that our test suite has made it much easier to identify problems introduced by refactoring in addition to preemptively discovering bugs that are present in new code. We have also observed anecdotally that code that is more thoroughly tested as measured by code coverage has proven to be more reliable and easier to maintain.

While this test suite has been an indispensable resource for our project, we have found creating tests for our UIMA components to be challenging for a number of reasons. In a typical UIMA processing pipeline, components created by developers are instantiated by a UIMA container called the Collection Processing Manager (CPM) which decides at runtime how to instantiate components and what order they should run via configuration information provided in descriptor files. This pattern is typical of programming frameworks: the developer creates components that satisfy some API specification and then these components are managed by the framework. This means that the developer rarely directly instantiates the components that are developed and simple programs consisting of e.g. a main method are uncommon and can be awkward to create. This is indeed consistent with our experiences with UIMA. While this is generally a favorable approach for system development and deployment, it presents challenges to the developer that wants to isolate specific components (or classes that support them) for unit or functional testing purposes.

2 Testing UIMA Components

UIMA coordinates data generated and consumed by different components using a data structure called the Common Analysis Structure (CAS). The

[1] http://cleartk.googlecode.com

CAS represents the current state of analysis that has been performed on the data being analyzed. As a simple example, a UIMA component that performs tokenization on text would add token annotations to the CAS. A subsequent component such as a part-of-speech tagger would read the token annotations from the CAS and update them with part-of-speech labels. We have found that many of our tests involve making assertions on the contents of the CAS after a component or series of components has been executed for a given set of configuration parameters and input data. As such, the test must obtain an instance of a CAS after it has been passed through the components relevant to the tests.

For very simple scenarios a single descriptor file can be written which specifies all the configuration parameters necessary to instantiate a UIMA component, create a CAS instance, and process the CAS with the component. Creating and processing a CAS from such a descriptor file takes 5-10 lines of Java code, plus 30-50 lines of XML for the descriptor file. This is not a large overhead if there is a single test per component, however, testing a variety of parameter settings for each component results in a proliferation of descriptor files. These descriptor files can be difficult to maintain in an evolving codebase because they are tightly coupled with the Java components they describe, yet most code refactoring tools fail to update the XML descriptor when they modify the Java code. As a result, the test suite can become unreliable unless substantial manual effort is applied to maintain the descriptor files.

Thus, for ease of refactoring and to minimize the number of additional files required, it made sense to put most of the testing code in Java instead of XML. But the UIMA framework does not make it easy to instantiate components or create CAS objects without an XML descriptor, so even for relatively simple scenarios we found ourselves writing dozens of lines of setup code before we could even start to make assertions about the expected contents of a CAS. Fortunately, much of this code was similar across test cases, so as the ClearTK test suite grew, we consolidated the common testing code. The end result was a number of utility classes which allow UIMA components to be instantiated and run over CAS objects in just 5-10 lines of Java code. We decided that these utilities could also ease testing for projects other than ClearTK, so we created the UUTUC project, which provides our UIMA unit test utility code.

3 UUTUC

UUTUC[2] provides a number of convenience classes for instantiating, running, and testing UIMA components without the overhead of the typical UIMA processing pipeline and without the need to provide XML descriptor files.

Note that UUTUC cannot isolate components entirely from UIMA – it is still necessary, for example, to create AnalysisEngine objects, JCas objects, Annotation objects, etc. Even if it were possible to isolate components entirely from UIMA, this would generally be undesirable as it would result in testing components in a different environment from that of their expected runtime. Instead, UUTUC makes it easier to create UIMA objects entirely in Java code, without having to create the various XML descriptor files that are usually required by UIMA.

Figure 1 provides a complete code listing for a test of a UIMA component we wrote that provides a simple wrapper around the widely used Snowball stemmer[3]. A complete understanding of this code would require detailed UIMA background that is outside the scope this paper. In short, however, the code creates a UIMA component from the `SnowballStemmer` class, fills a CAS with text and tokens, processes this CAS with the stemmer, and checks that the tokens were stemmed as expected. Here are some of the highlights of how UUTUC made this easier:

Line 3 uses `TypeSystemDescriptionFactory` to create a `TypeSystemDescription` from the user-defined annotation classes `Token` and `Sentence`. Without this factory, a 10 line XML descriptor would have been required.

Line 5 uses `AnalysisEngineFactory` to create an `AnalysisEngine` component from the user-defined annotator class `SnowballStemmer` and the type system description, setting the stemmer name parameter to `"English"`. Without this factory, a 40-50 line XML descriptor would have been required (and near duplicate descrip-

[2] http://uutuc.googlecode.com – provided under BSD license
[3] http://snowball.tartarus.org

```
 1 @Test
 2 public void testSimple() throws UIMAException {
 3     TypeSystemDescription typeSystemDescription = TypeSystemDescriptionFactory
 4         .createTypeSystemDescription(Token.class, Sentence.class);
 5     AnalysisEngine engine = AnalysisEngineFactory.createAnalysisEngine(
 6         SnowballStemmer.class, typeSystemDescription,
 7         SnowballStemmer.PARAM_STEMMER_NAME, "English");
 8     JCas jCas = engine.newJCas();
 9     String text =   "The brown foxes jumped quickly over the lazy dog.";
10     String tokens = "The brown foxes jumped quickly over the lazy dog .";
11     TokenFactory.createTokens(jCas, text, Token.class, Sentence.class, tokens);
12     engine.process(jCas);
13     List<String> actual = new ArrayList<String>();
14     for (Token token: AnnotationRetrieval.getAnnotations(jCas, Token.class)) {
15         actual.add(token.getStem());
16     }
17     String expected = "the brown fox jump quick over the lazi dog .";
18     Assert.assertEquals(Arrays.asList(expected.split(" ")), actual);
19 }
```
Figure 1: A complete test case using UUTUC.

tor files would have been required for each additional parameter setting tested).

Line 11 uses `TokenFactory` to set the text of the CAS object and to populate it with `Token` and `Sentence` annotations. Creating these annotations and adding them to the CAS manually would have taken about 20 lines of Java code, including many character offsets that would have to be manually adjusted any time the test case was changed.

While a Python programmer might not be impressed with the brevity of this code, anyone who has written Java test code for UIMA components will appreciate the simplicity of this test over an approach that does not make use of the UUTUC utility classes.

4 Results

The test suite we created for ClearTK was built using UUTUC and JUnit version 4[4] and consists of 92 class definitions (i.e. files that end in *.java*) containing 258 tests (i.e. methods with the marked with the annotation *@Test*). These tests contain a total of 1,943 individual assertions. To measure code coverage of our unit tests we use EclEmma[5], a lightweight analysis tool available for the Eclipse development environment, which counts the number of lines that are executed (or not) when a suite of unit tests are executed. While this approach provides only a rough approximation of how well the unit tests "cover" the source code, we have found anecdotally that code with higher coverage reported by EclEmma proves to be more reliable and easier to maintain. Overall, our test suite provides 74.3% code coverage of ClearTK (5,391 lines covered out of 7,252) after factoring out automatically generated code created by JCasGen. Much of the uncovered code corresponds to the blocks catching rare exceptions. While it is important to test that code throws exceptions when it is expected to, forcing test code to throw all exceptions that are explicitly caught can be tedious and sometimes technically quite difficult.

5 Discussion

We learned several lessons while building our test suite. We started writing tests using Groovy, a dynamic language for the Java Virtual Machine. The hope was to simplify testing by using a less verbose language than Java. While Groovy provides a great syntax for creating tests that are much less verbose, we found that creating and maintaining these unit tests was cumbersome using the Eclipse plug-in that was available at the time (Summer 2007). In particular, refactoring tasks such as changing class names or method names would succeed in the Java code, but the Groovy test code would not be updated, a similar problem to that of UIMA's XML descriptor files. We also found that Eclipse became less responsive because user actions would often wait for the Groovy compiler to

[4] http://junit.org
[5] http://www.eclemma.org

complete. Additionally, Groovy tests involving Java's Generics would sometimes work on one platform (Windows) and fail on another (Linux or Mac). For these reasons we abandoned using Groovy and converted our tests to Java. It should be noted that the authors are novice users of Groovy and that Groovy (and the Eclipse Groovy plug-in) may have matured significantly in the intervening two years.

Another challenge we confronted while building our test suite was the use of licensed data. For example, ClearTK contains a component for reading and parsing PennTreebank formatted data. One of our tests reads in and parses the entire PennTreebank corpus, but since we do not have the rights to redistribute the PennTreeBank, we could not include this test as part of the test suite distributed with ClearTK. So as not to lose this valuable test, we created a sibling project of ClearTK which is not publicly available, but from which we could run tests on ClearTK. This sibling project now contains all of our unit tests which use data we cannot distribute. We are considering making this project available separately for those who have access to the relevant data sets.

We have begun to compile a growing list of best practices for our test suite. These include:

Reuse JCas objects. In UIMA, creating a JCas object is expensive. Instead of creating a new JCas object for each test, a single JCas object should be reused for many tests where possible.

Refer to descriptors by name, not location. UIMA allows descriptors to be located by either "location" (a file system path) or "name" (a Java-style dotted package name). Descriptors referred to by "name" can be found in a .jar file, while descriptors referred to by "location" cannot. This applies to imports of both type system descriptions (e.g. in component descriptors) and to imports of CAS processors (e.g. in collection processing engine descriptors).

Test loading of descriptor files. As discussed, XML descriptor files can become stale in an evolving codebase. Simply loading each descriptor in UIMA and verifying that the parameters are as expected is often enough to keep the descriptor files working if the actual component code is being properly checked through other tests.

Test copyright and license statements. We found it useful to add unit tests that search through our source files (both Java code and descriptor files) and verify that appropriate copyright and license statements are present. Such statements were a requirement of the technology transfer office we were working with, and were often accidentally omitted when new source files were added to ClearTK. Adding a unit test to check for this meant that we caught such omissions much earlier.

As ClearTK has grown in size and complexity its test suite has proven many times over to be a vital instrument in detecting bugs introduced by extending or refactoring existing code. We have found that the code in UUTUC has greatly decreased the burden of maintaining and extending this test suite, and so we have made it available for others to use.

References

Philip V. Ogren, Philipp G. Wetzler, and Steven Bethard. 2008. ClearTK: a UIMA toolkit for statistical natural language processing. In UIMA for NLP workshop at LREC.

David Ferrucci and Adam Lally. 2004. UIMA: an architectural approach to unstructured information processing in the corporate research environment. Natural Language Engineering, 10(3-4):327–348.