OpenDMAP Project

MANUAL

Testing Manual

OpenDMAP Project

May 2006

Testing Manual
OpenDMAP Project
May 2006

R. James Firby

Contents

1 Introduction

A relatively general testing framework has been created that allows the OpenDMAP code itself to be tested and also allows users to build their own tests for larger systems. This framework does not rely on OpenDMAP and can be used for other projects as well. However, the testing framework is included in the OpenDMAP jar file so no additional libraries are required to run OpenDMAP tests.

2 Testing OpenDMAP

While internal unit tests are an effective means for testing and validating basic functionality in the OpenDMAP system, they are difficult to use when testing the highly recursive overall behavior of the system. Therefore OpenDMAP includes a regression testing system designed to run multiple suites of tests against the OpenDMAP library as a whole.

Each regression test consists of an input phrase to be parsed by OpenDMAP and a set of expected responses encoded as serialized strings. The test system initializes a parser and hands it each input phrase. The parser results are then compared against those encoded in the test and discrepancies are noted.

A number of test suites are included with the OpenDMAP system and others can be added easily.

2.1 Supplied OpenDMAP Test Suites

All test suites supplied with OpenDMAP are in the `test` directory. This directory holds the test suites files and the Protégé projects and pattern files they require.

The existing suites are:

- `pattern-tests.xml`
  This test suite includes a number of tests to check that OpenDMAP patterns are being read in properly. These tests are aimed primarily at the "JavaCC" parser used internally to OpenDMAP. If the pattern language is changed, additional tests should be added to this suite.

- `pattern-rule-tests.xml`
  This test suite goes a step beyond the pattern tests and makes sure that OpenDMAP is building its own internal pattern structures correctly from the pattern definitions supplied. These tests are aimed at the generation of anytime predictions within the OpenDMAP system. If the basic pattern objects in the OpenDMAP Java code are changed, additional tests should be added to this suite.

- `basic-tests.xml`
  This test suite includes basic concept recognition tests. It is aimed at making sure that OpenDMAP is recognizing complex patterns correctly (with no slots). When new features are added to the pattern language or any of the internal OpenDMAP algorithms are changed, new tests should be added to this suite to cover changes to the basic pattern recognition process.

- `slot-tests.xml`
  This test suite includes tests of patterns that include simple slot references. It is aimed at making sure that OpenDMAP is properly recognizing and binding slot fillers. If those algorithms are ever changed new tests should be added to this suite.
- `slot-nesting-tests.xml`
  This test suite includes tests where recognition should generate slots fillers that include more complex concepts that in turn have their own slot fillers. Then OpenDMAP algorithms for reference building are changed new tests should be added to this suite.
- `slot-property-tests.xml`
  This test suite is aimed at testing the additional property tags that can be attached to slots in patterns. It also tests the use of denotation markers in patterns. If these algorithms change, new tests should be added to this suite.

In addition, a customized test suite is included in the `test/people-loving-people` directory. This test suite name:

- `love-hate-tests.xml`
  This test suite shows how the testing system can be customized to deal with the output from processing steps that come after OpenDMAP parsing. In this case, expected the expected outputs are English sentences.

2.2 Adding New OpenDMAP Tests

The OpenDMAP system comes with three different types of tests predefined: pattern tests, pattern rule tests, and parsing tests. Each one takes a different sort of input and generates a different sort of output.

Pattern tests use the `pattern` form:

```
<pattern name="name">
  <input>pattern component</input>
  <output>pattern expansion</output>
  ... alternate pattern expansions ...
</pattern>
```

The `name` attribute assigns this test a name to be used in output messages. The `input` element holds a valid component from an OpenDMAP pattern. This component can be as simple or as complex as desired. The expected `output` elements are all of the different ways that this pattern component can be expanded and begin with a constant token. Each output begins with the constant that needs to be recognized first for this expansion followed by the expansion itself.

For example:

```
<pattern name="Test-7a">
  <input>Kelly* Mary</input>
  <output>"Mary" . "Mary"</output>
  <output>"Kelly" . ("Kelly" "Mary")</output>
  <output>"Kelly" . (("Kelly" "Kelly"+) "Mary")</output>
</pattern>
```

The input for this test is a pattern that specifies zero or more "Kelly" tokens followed by a "Mary" token. Internally, this is turned into three different expansions that start with a constant token. The first one is just the pattern "Mary". The second is the pattern "Kelly" followed by "Mary" and the third is one "Kelly" followed by one or more "Kelly" tokens followed by a "Mary" token.[1]

Pattern rule tests use the `pattern-rule` the form:

```
<pattern-rule name="name">
  <input>complete pattern rule</input>
  <output>pattern expansion</output>
  ... disjunctive pattern expansions ...
</pattern-rule>
```

The `name` attribute assigns this test a name to be used in output messages. The `input` element holds a complete OpenDMAP pattern as would appear in a pattern file. This component can be as simple or as complex as desired. The expected `output` elements are all of the disjunctive components of this pattern.

For example:

```
<pattern-rule name="Test-4">
  <input>a := Kelly+;</input>
  <output>"Kelly"+</output>
</pattern-rule>
```

The input pattern rule says that frame "a" should be assigned the pattern "Kelly+". The output includes only one disjunctive component, which is the pattern itself. If the pattern was several disjuncts separated by commas, each disjunct would show up as a separate output.

Parser tests use the `test` form:

```
<test name="name" [includeSubsumed="false"] [includeText="true"]>
  <input>input sentence</input>
  <output>serialized output reference</output>
  ... more serialized output references ...
</test>
```

The `name` attribute assigns this test a name to be used in output messages. The `input` element holds an input sentence to be parsed by OpenDMAP. Each `output` element holds a reference recognized during the processing of this input sentence. The `includeSubsumed` attribute can be set to *false* which tells the testing program to only look at references that are not subsumed by other references. The `includeText` attribute can be set to *true* to tell the testing program to look at every individual text token recognized during processing of the input as well as recognized references.

For example:

```
<test name="Test-2" includeSubsumed="true">
  <input>Charlie loves Mary</input>
  <output>Charlie from 0 to 0</output>
  <output>Mary from 2 to 2</output>
  <output>Loves [lover]=Charlie [beloved]=Mary from 0 to 2</output>
</test>
```

[1]A useful way to get these output patterns is to add a new test with no output and then run the test and see what the testing system says it produces. If these make sense, then add them to the test as valid outputs.

The input is "Charlie loves Mary". The test says that after parsing this input, OpenDMAP should have recognized a reference to the frame called `{Charlie}`, a reference to the frame called `{Mary}`, and a reference to the `Loves` frame where the `[lover]` slot is filled with `{Charlie}` and the `[beloved]` slot is filled by `{Mary}`. The additional number supplied give the starting and ending token numbers for the input words used to recognize the reference. Note that this test sets the `includeSubsumed` to be *true* so that `{Charlie}` and `{Mary}` will be checked as separate outputs. They are "subsumed" by the third output and would normally not be considered.

3 Running Test Suites

Test suites are run using the Java class `TestMain`:

```
edu.uchsc.ccp.testing.TestMain
```

The `TestMain`main method in this class expects to be passed a set of test suite file names. It reads each tests suite into memory and then runs all the tests. It prints out a summary of each suite once it runs.

A test suite summary includes the number of tests run, the number that passed, the number that failed, and for each failing test a description of why it failed.

For example,

```
edu.uchsc.ccp.testing.TestMain test/pattern-tests.xml test/slot-tests.xml
```

will run the test suites described in `'test/pattern-tests.xml'` and `'test/slot-tests.xml'` and print the results.

This program can also be given the command line switch '-b' which tells it to stop as soon as it encounters a test that fails. This can make the output a lot easier to understand when debugging things. For example:

```
edu.uchsc.ccp.testing.TestMain -b test/slot-tests.xml
```

will run the test suite in the file but stop at the first test that fails.

4 Building a Custom Testing Framework

When building an application that uses OpenDMAP, the testing framework can be expanded and customized to test the new application end-to-end.

The testing framework consists of several pieces in the Java package *edu.uchsc.ccp.testing*:

- The Test Manager
- Test Suites
- Test Contexts
- Test Classes

- Test Suite XML Files

There are two key components that can to be extended to create a custom testing framework:

- A Test Context Java class
- One or more Test Java classes

One test context object is created and initialized for each test suite and it is passed to each test run in that suite. The test context is used to hold any shared state that is required between tests.

A test class defines a *type* of test that can appear in a test suite. An instance of a test class is created for each test that appears in the test suite. Different types of test require different test classes.

4.1 Creating a Test Context

A `TestContext` contains any global state that should be shared across the tests in a test suite. For example, for most OpenDMAP test suites the test context holds a shared `Parser` object. This parser is created and initialized once and then used by every test in the suite.

A Java class to be used as a test context must include a constructor with no arguments and implement the `TestContext` interface:

```
edu.uchsc.ccp.testing.TestContext
```

which defines the two methods `initialize` and `terminate`:

```
public boolean initialize(TestSuite suite, Collection<Object> errors);
public boolean terminate(Collection<Object> errors);
```

When the testing framework is ready to run a test suite, it creates an instance of the test context and calls the `initialize` method. This method should set up any state that tests within the suite will need to run.

The *suite* argument to `initialize` is a `TestSuite` holding the test suite that is about to run. Any context values defined in the test suite XML file can be extracted from the *suite* using the methods `getValue` and `getValues`:

```
suite.getValue(name)  => string value
suite.getValues(name) => collection of string values
```

The first method can be used to fetch the first value with that name defined in the XML file and the second method returns all values with that name defined in the XML file.

These values are available so that initialization of the testing state can be customized for different suites. For example, the DMAP test context gets the appropriate Protégé project name and pattern files to load into the parser using these methods.

The *errors* argument to `initialize` and `terminate` is a collection that errors should be added to if encountered. These errors will be reported in the test suite transcript.

The `initialize` method should return *true* if the test context can now be used to run tests. The `terminate` method should return *true* if it succeeds.

4.2 Creating a Test Class

A test class does two things:

- It parses the XML description of itself, and
- It runs the test defined by that XML.

A new Java test class must extend the `Test` class found in:

```
edu.uchsc.ccp.testing.Test
```

This abstract class requires the definition of the three methods `startElement`, `endElement`, and `run`:

```
boolean startElement(String uri, String localName, String qName,
                     Attributes atts) throws SAXException
boolean endElement(String uri, String localName, String qName)
         throws SAXException;
boolean run(TestContext context, PrintStream stream)
```

The first two of these are used when the test is being read from an XML file. The third one is used to run the test.

Parsing an XML Test Definition

When a test is read from an XML test suite file, several things happen. First, the test suite uses the XML definition of the test to determine the *type* of test being read. Second, the test suite creates a new instance of the Java test class for that *type* of test. Third, that instance is used to parse the XML elements internal to the XML test definition. This is done by calling `startElement` when each internal element begins and `endElement` when each internal element is closed. Finally, when the test definition itself is closed, the test suite saves away the test instance to be run later. The instance is responsible for saving any state it needs from its XML definition.

The `startElement` and `endElement` methods are patterned after the methods with the same names in the Java SAX class. The differences are:

- These methods are only called within the definition of a single test. The larger context of the XML test suite file is managed by the test suite reading the file.
- These methods should return *true* if they consume the XML element. If the XML element is not valid for this test, or comes out of order, these methods should return *false* so appropriate error messages can be generated.

To aid in processing the XML elements within a test, the `Test` abstract class includes the methods `startBuffering` and `getBuffer`:

```
this.startBuffering() => void
this.getBuffer() => string containing all characters read since startBuffering
```

These methods can be called to start and stop gathering the characters read between XML elements.

A `Test` derived class may also define a constructor with the following signature:

```
Test(String kind, Attributes attributes) throws SAXException
```

The *kind* argument will be passed the XML element name used in the test suite file to define this test. (See Section 4.2 below on writing an XML test suite file).

The *attributes* argument will hold the attributes specified for the XML element defining this test. The default constructor extracts the attributes "`name`" and "`fail`" and makes them available through the `Test` class methods `getName` and `isFailTest`:

```
getName() => name string
isFailTest() => boolean true if this test is supposed to fail
```

If a test class defines its own constructor, it may extract any additional attributes that the test needs.

Running a Test

When a test suite runs a test instance, it calls the `run` method defined for that test:

```
boolean run(TestContext context, PrintStream stream)
```

This method is passed the test *context* object created for this test suite and a print *stream* where this test can describe the reason that a test failed (if it does).

This method should run the test that it defines. It may access state made available by the context object (such as an OpenDMAP parser to process an input string) to run the test. It must then decide if the test succeeded or failed. If it succeeds then nothing should be output and the method should return *true*. If the test fails, then a descriptive message should be written to *stream* so that the user will know why the test failed, and the method should return *false*.

A trick to remember here is that the XML test suite may define tests that are expected to fail. If a test fails and it supposed to, then the `run` method should not generate any output and it should return *true*. The `run` method can check whether it is supposed to fail using the `isFailTest` method defined by its parent `Test` class.

4.3 Input Output Tests

A great many tests used in natural language processing systems consist of parsing an input string and then checking that the correct results are generated. To simplify the definition of custom tests of this sort, the testing framework includes a subclass of `Test` called `InputOutputTest` in the package:

```
edu.uchsc.ccp.testing.InputOutputTest
```

This class extends `Test` to include `InputOutputTest`startElement and `InputOutputTest`endElement methods that look for one `<input>` and any number of `<output>` elements in a test definition.

It also includes a definition for the `InputOutputTest`run method that runs a test on the input and decides if it succeeds by comparing the output it generates with the output strings in the test.

If all test strings are generated and no other strings are generated, then the test succeeds. If the test fails, it prints out a complete list of expected versus generated strings.

This `InputOutputTest`run method expects that the test class will define a method called `InputOutputTest`runTest with the signature:

```
String runTest(TestContext context, String input, Collection<String> output)
```

This method should run the test given the test suite *context* and the test ¡*input* string. It should then write a string representing each output from the test into the *output* collection. These strings will be compared verbatim to the strings supplied in the `<output>` XML elements defining the test.

The `InputOutputTest`runTest method should return *null* if the test succeeds and an informative error string if the test fails. This method need not worry about whether or not the test was supposed to fail, that processing will be managed by the `InputOutputTest` `InputOutputTest`run method.

To create a new test class based on the `InputOutputTest` class, create a Java class that extends `InputOutputTest` and implements the `InputOutputTest`runTest method. If desired, the new class can also implement `startElement` and `endElement` methods to capture additional elements with the test definition.

All of the OpenDMAP standard test classes extend the `InputOutputTest` class. They can be found as examples in the Java package:

```
edu.uchsc.ccp.dmap.test
```

In addition, some OpenDMAP example classes extend the `InputOutputTest` class. See for example:

```
edu.uchsc.ccp.dmap.example.LoveHateTester
```

5 Creating a Custom XML Test Suite File

Test suites are typically written as XML files rather than as Java classes directly. This makes it much simpler to add more tests and to find tests that fail.

The general format for an XML test suite file is:

```
<test-suite name="name">
 <test-context>Java-Test-Context-Class-Name</test-context>
 <test-type name="mytest">Java-Test-Class-Name</test-type>
 ...context values...
 ...test forms...
</test-suite>
```

The overall test file format holds one ¡test-suite¿ declaration. This declaration must include a name that distinguishes it from other test suites. If the `TestMain` class is used to load and run test suites, it will attempt to merge together test suites with the same name. This allows a test suite to be split across multiple files if desired.

The test suite may also include a `<test-context>` element that declares the class of object that should be used as the context for the tests in this suite.

If the test suite includes any tests, it must also include one or more `<test-type>` definitions to map the test elements to Java `Test` class names.

Finally, the test suite may include context value definitions and the tests themselves.

5.1 Specifying the Test Context

The test context for a test suite is declared using the `<test-context>` XML element. For example, the declaration needed in OpenDMAP test suites is:

```
<test-context>edu.uchsc.ccp.dmap.test.ParserTestContext</test-context>
```

This tells the test suite that before it can run any tests, it must create an instance of the Java class `ParserTestContext`. This context object will then be passed to each test as it is run.

Different test suites can, of course, use different context objects. However, it is important that the test classes used by the suite understand the type of context being used.

5.2 Specifying the Test Types

The XML test forms that are to be included in the test suite file must be declared using the element:

```
<test-type name="mytest">Java-Test-Class-Name</test-type>
```

This tells the test suite that XML `<mytest>` forms will be used to declare tests within this file and that `<mytest>` tests should be instantiated using the Java class named *Java-Test-Class-Name*.

For example, in OpenDMAP test files the following declarations are often used:

```
<test-type name="test">edu.uchsc.ccp.dmap.test.ParserTest</test-type>
<test-type name="pattern">edu.uchsc.ccp.dmap.test.PatternTest</test-type>
```

The first tells the enclosing test suite that tests declared using `<test>` should be instantiated as `ParserTest` objects and tests declared using `<pattern>` should be instantiated as `PatternTest` objects.

For example, since `ParserTest` and `PatternTest` extend the `InputOutputTest` class, a test suite that includes both of the declarations above might include tests like:

```
<test name="Test-1" includeText="true" includeSubsumed="true">
  <input>John loves Mary</input>
  <output>"John" from 0 to 0</output>
  <output>John from 0 to 0</output>
  <output>"loves" from 1 to 1</output>
  <output>"Mary" from 2 to 2</output>
  <output>Mary from 2 to 2</output>
  <output>Loves [lover]=John [beloved]=Mary from 0 to 2</output>
</test>

<pattern name="Test-5">
  <input>(Kelly Mary)+</input>
  <output>"Kelly" . ("Kelly" "Mary")</output>
  <output>"Kelly" . (("Kelly" "Mary") ("Kelly" "Mary")+)</output>
</pattern>
```

Any number of `<test-type>` forms can be included in a test suite to declare that different sorts of tests will be defined. The same Java class can be used in multiple `<test-type>` forms if the same types of tests are defined using different XML elements.

If no `<test-type>` elements are included in the test suite, then no tests will be defined by the elements it contains. All internal XML elements will default to context value definitions.

5.3 Declaring Context Values

Any top-level XML elements found in a test suite that have not been declared as test types, are assumed to be declaring values that will be used by the context object. Everything within the element is turned into a string and added to the test suite as a value with the name of the element.

For example, in OpenDMAP test suites, the following values often appear:

```
<protege-project>protege-project-file-name</protege-project>
<pattern-file>pattern-file-name</pattern-file>
```

These specify that that the included file names should be made available by the test suite as values named "`protege-project`" and "`pattern-file`". If more than one element of the same type is included in the suite, all of the values with that name will be represented as a collection.

As mentioned above, the `getValue` and `getValues` methods can be used in the `initialize` method of the context object to get access to these values.

Any number of context values with any number of names can be included in an XML test suite.

Note: once a specific test context has been decided upon for a particular type of test suite, a DTD for that particular type of suite can be defined and used with an editor to help flag inappropriate context value definitions.

5.4 Writing the Tests

Tests within the testing framework all have the basic form:

```
<mytest name="name" [fail="true"]>
  ...
</mytest>
```

Where `mytest` is defined using a `<test-type name="mytest">` element earlier in the test suite file.

All tests must be given a `name` attribute and may be declared as tests that are supposed to fail using the `fail` attribute.

The internal elements that should appear inside a test depend on the internals of the `Test` class paired with the test type in the `<test-type>` element that defines it.

Additional attributes may also be used by some types of test class.

Examples of test classes and test suite files can be found in:

```
edu.uchsc.ccp.dmap.test - Package holding OpenDMAP test classes
test                    - Directory holding OpenDMAP test suites
```

and

```
edu.uchsc.ccp.example.LoveHateTester
test/projects/people-loving-people/love-hate-test.xml
```

Index