# 3. VDMJUnit Testing

Section 2.6 describes the debugging of specifications using features built into the command line tools, either directly in VDMJ or indirectly using the VDMJC client. But once a specification is working, it is also possible to automatically execute batches of tests by using VDMJUnit, which adds VDM interpreter support to the Java JUnit 4 framework [12].

JUnit tests are written in Java, and are usually executed as part of an automated build of a larger system, or executed piecemeal in a development IDE. The JUnit framework allows Java classes and methods to be annotated to identify them as tests, and then provides the means to execute all of the tests in a give class or package, reporting any errors encountered.

The VDMJUnit package includes base classes that provide tests with the means to load a specification from file(s), (re)initialize them, and make test evaluations against the specification loaded, including tests which look for specific runtime errors, such as invariant failures.

Here is a minimal example:

```
public class SpecTest extends VDMJUnitTestPP
{
	@BeforeClass
	public static void start() throws Exception
	{
		setRelease(Release.VDM_10);
		readSpecification("test.vpp");
	}

	@Before
	public void setUp()
	{
		init();
	}

	@Test
	public void test() throws Exception
	{
		assertEquals(10, runInt("new A().f(9)"));
	}
}
```

Where the file "test.vpp" might contain the following VDM class definition:

```
class A
functions
	public f: nat -> nat
	f(i) == i + 1
	pre i < 10;
end A
```

Since this test is of a VDM++ class, the JUnit test class extends *VDMJUnitTestPP*; similar base classes are provided for VDM-SL and VDM-RT.

@BeforeClass is a standard JUnit 4 annotation which indicates that this static method is to be executed once before all tests in the class. This is used to execute the *setRelease* and *readSpecification* methods, inherited from *VDMJUnitTestPP*, which loads the file name(s) passed, and parses and type checks them using the *VDM10* language extensions. There must be no parse or type checking errors. The *readSpecification* method takes a varargs list of filenames or directories which are located relative to the Java classpath. Directories are searched (shallow) for all VDM source files. It is permissible to pass no files and evaluate bare expressions in the tests (like "1+1"), but the *readSpecification* method must always be called, and called only once per test class.

  

The @Before annotation is also from JUnit 4, and indicates that this method should be executed before each test defined in the class. The *init* method shown initializes the specification (eg. setting the state data back to the original settings). Without an @Before method to do this, tests will see the effect of earlier tests on the specification state (which may be useful). If the specification is not initialized per test, then it must be initialized once in the @BeforeClass method, after the specification is loaded.

Lastly, the @Test annotation identifies test methods, in this example a test asserting that a particular expression should produce a given (integer) value. The string argument to *runInt* can be any valid VDM expression that returns an integer value. The simpler *run* method returns a raw VDMJ *Value* object, which can return arbitrarily complex values (sets, records, objects etc).

A method called *assertVDM* is provided to make assertions about VDM values by using VDM expressions. This can be useful if it is too difficult to unpick the Value objects returned, when a VDM expression would be far simpler. The methods set a “RESULT” variable from the value passed in (or as a result of evaluating the expression passed), and this should be used to create a boolean VDM assertion expression. Assertion expressions can use global constants defined in the specification.

```
@Test
public void one() throws Exception
{
	run("setValue(123)");
	assertVDM("getValue()", "RESULT = 123");

	try
	{
		Value r = run("getValue()");
		assertVDM("Testing!", r, "RESULT = 456");
		fail("Expected failure");
	}
	catch (AssertionError e)
	{
		assertEquals(e.getMessage(), "Testing!");
	}
}
```

Tests can catch and check exceptions if they believe a VDM runtime error should occur, for example:

```
@Test
public void one() throws Exception
{
	try
	{
		create("object", "new A()");
		run("object.f(100)");
		fail("Expecting precondition failure");
	}
	catch (ContextException e)
	{
		// Error 4055: Precondition failure: pre_f in 'A' at line 5:11
		assertEquals(4055, e.number);
		assertEquals("A", e.location.module);
		assertEquals("test.vpp", e.location.file.getName());
		assertEquals(5, e.location.startLine);
		assertEquals(11, e.location.startPos);
	}
}
```

In this case, a VDMJ *ContextException* contains all of the information about the error that occurred, which can be checked by the test. Note that an *init* call should be made after an error to reset the specification to a known state, perhaps in the @Before method.

  

It is also possible to execute combinatorial tests from within the VDMJUnit framework (see section 4.). As with the command line, all the tests generated from a trace may be executed, or a contiguous range of tests, or a reduced subset of the tests. Three *runTrace* methods are provided, as follows:

```
@Test
public void one() throws Exception
{
	assertTrue(runTrace("T1"));
}

@Test
public void two() throws Exception
{
	assertTrue(runTrace("T1", 1, 3));
}

@Test
public void three() throws Exception
{
	assertTrue(runTrace("T1", 0.5, TraceReductionType.RANDOM, 123));
}
```

The first method executes all the tests generated from the "T1" trace. The second method runs a range of tests from those generated (tests are numbered from 1). The third method reduces the number of tests generated to 50% (ie. 0.5) using a RANDOM reduction technique, seeded with the value 123 (for reproducible results). There is a description of test reduction in section 4.2..

In all three cases, if all of the tests pass, the *runTrace* method returns true else false. Note that INCONCLUSIVE results are regarded as a failure. The expansion of the individual test cases, including test numbers and the output from each test execution, is sent to stdout in the same format as they are from the command line (see section 4.).

The VDMJUnitTest framework provides the following methods to tests:

- **setRelease(Release)** to specify the language release to parse. The default is VDM classic.
- **readSpecification([Charset], file/dirs...)** to parse and type check specification files.
- **getErrors()** to get a List<VDMError> of syntax or type checking errors after readSpecification
- **getWarnings()** to get a List<VDMWarning> after readSpecification
- **init** to (re)initialize a loaded specification.
- **setDefault(Name)** to set the default module or class name.
- **create(Name, Expr)** for VDM++ and VDM-RT only, to create temporary object values
- **run(Expr)** to parse and evaluate the expression and return a VDMJ Value.
- **runInt(Expr)** same as run, but return a long.
- **runReal(Expr)** same as run, but return a double.
- **runBool(Expr)** same as run, but return a boolean.
- **runTrace(Name)** run all the tests from the named combinatorial trace.
- **runTrace(Name, first, last)** run the specific test range from a trace.
- **runTrace(Name, subset, method, seed)** run a subset of tests, reduced by the given method.
- **assertVDM([Message], Value, VDM assertion)** test value against a VDM assertion
- **assertVDM([Message], Expr, VDM assertion)** test VDM expression against VDM assertion

  

Note that a Java *Charset* can be passed to *readSpecification*, if the files are not in the default character encoding for your locale. See the Javadocs for full details.

A subclass of the VDMJ *Value* class is returned from the *run* method, which allows any value to be returned. Such *Value* objects either have fields or getter methods to allow them to be examined, though note that many *Value* subtypes contain other *Values* – for example, a *SetValue* contains a set of *Values*, and a *RecordValue* contains a map of String field names to *Values*. *Values* can be converted to Java primitives using (for example) the *boolValue* or *realValue* methods.

To execute JUnit tests with VDMJUnit, you need to have the VDMJ jar as well as the Junit 4 and VDMJUnit jars on the classpath.

Although *VDMJUnitTestSL* and related classes are intended to be used with JUnit tests, the same methods can be used in stand-alone Java applications. For example, the following code will load a specification and execute an expression passed from the command line.

```
public class NonJUnitExample extends VDMJUnitTestSL
{
	public static void main(String[] args) throws Exception
	{
		new NonJUnitExample().execute(args);
	}

	public void execute(String[] args) throws Exception
	{
		setRelease(Release.VDM_10);
		readSpecification(args[0]);
		init();
		System.out.println(run(args[1]));
	}
}
```

  