Maven Plugin Harness

jira issue

Read this jira issue for some details.

http://jira.codehaus.org/browse/MNG-32

current location of harness

mojo/mojo-sandbox/maven-plugin-testing/maven-plugin-testing-harness

what it is and isn't

The harness provides an abstract base class for testing plugins and an increasing collection of stub implementations of key maven classes. Many of your plugins contain plexus components which should be populated by the container so in your test cases you will be 'looking up' a mojo and then either using getters and setters on it to setup the mojo or make use of the PlexusConfiguration step that lets an xml snippet on disk configuration everything for you.

This plugin harness is not the be all end all solution to mojo testing. It does not do integration testing and the primary scope of the harness is to provide a way for plugin authors to create a fully populated instance of their mojo only with stubs in the place of critical maven classes. This is an attempt to all for some level of mojo testing without needing a full fledged maven execution environment.

how to basically use it

```
import org.codehaus.plexus.util.FileUtils;
import org.apache.maven.plugins.testing.AbstractMojoTestCase;

import java.io.File;

public class CompilerMojoTest
    extends AbstractMojoTestCase
{
    protected void setUp() throws Exception {

        // required for mojo lookups to work
        super.setUp();

    }

    /**
     * tests the proper discovery and configuration of the mojo
     *
     * @throws Exception
     */
    public void testCompilerTestEnvironment() throws Exception {

        File testPom = new File( getBasedir(),
"target/test-classes/unit/compiler-basic-test/plugin-config.xml" );

        CompilerMojo mojo = (CompilerMojo) lookupMojo ("compile", testPom );

        assertNotNull( mojo );
    }
}
```

The first step is to make a test case in src/test/java and extend the AbstractMojoTestCase class. You *must* make sure that your setUp() method called the super.setUp() otherwise looking up your components will fail. Then to get your mojo it is a simple matter of looking up the mojo and passing in the configuration plugin pom.

```
<project>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <compileSourceRoots>

<compileSourceRoot>${basedir}\target\test-classes\unit\compiler-basic-test\src\main\ja
va</compileSourceRoot>
          </compileSourceRoots>
          <compilerId>javac</compilerId>
          <debug>true</debug>

<outputDirectory>${basedir}\target\test\unit\compiler-basic-test\target\classes</outpu
tDirectory>

<buildDirectory>${basedir}\target\test\unit\compiler-basic-test\target</buildDirectory
>
          <projectArtifact
implementation="org.apache.maven.plugins.testing.stubs.StubArtifact"/>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

This a sample plugin configuration file. If custom expressions are needed then we can add them to the StubResolverExpressionEvaluator class in the plugin harness so things like the

```
${basedir}
```

above are taken care of. Notice the *projectArtifact* element above with the implementation="" attribute. This is how you will inject the StubArtifact implementation into that projectArtifact private variable on the mojo. Additional elements can be inserted as child elements and they will in turn be assigned to the variables inside that object. An example would be setting the groupId variable on the StubMavenProject. Note: these internal variable might also need implementation="" attributes.

another way to use the harness

There is another way to use the harness if the above solution isn't desired, but it is a bit more dangerous perhaps and requires pretty good legwork to make sure it doesn't blow up, but in some circumstances it might just make life easier.

The AsbtractMojoTestCase has another utility method on it for setting a value of variables in an object that might not have a setter.

```
   protected setVariableValueInObject( Object object, String variable, Object value)
throws IllegalAccessException
```

This should allow you to populate most, if not all of the mojo as you need, depending on the depth to which it needs to be configured. Here is an example on how it could be used.

```
    public void testSettingMojoVariables()
        throws Exception
    {
        SimpleMojo mojo = new SimpleMojo();

        setVariableValueToObject( mojo, "keyOne", "myValueOne" );

        assertEquals( "myValueOne", (String)getVariableValueFromObject( mojo, "keyOne"
) );

    }
```

Or I would even be used in conjuction with the above mechanism to tweak a value.

```
    public void testSettingMojoVariables()
        throws Exception
    {

        SimpleMojo mojo = lookupMojo( "simple", "path-to-test-pom" );

        setVariableValueToObject( mojo, "keyOne", "myValueOne" );

        assertEquals( "myValueOne", (String)getVariableValueFromObject( mojo, "keyOne"
) );

    }
```

Great care should be taken in the writing of tests using the approach since there are none of the safeguards in place that plexus provides for this sort of variable injection. However, this should help us in sticky situations were we need to change/set the value of a variable in an object that might not have a setter. Note: this setVariableValueToObject is generic and can work in all sort of objects that you might need. Also, whereas plexus will use the objects setter for the variable if it exists, this will not.

useful things to know

There are a couple of utility methods on the AbstractMojoTestCase that should make the asserts easier to work with.

```
   protected Object getVariableValueFromObject( Object object, String variable )
throws IllegalAccessException

   protected Map getVariablesAndValuesFromObject( Object object ) throws
IllegalAccessException
```

These two methods wrap functionality in the ReflectionUtils class in plexus utils that will let you retrieve the value of variable on the mojo. Since most of the mojo's variables are private scope and most mojo's don't make use of getters and setters this will let you assert things about the internal state of the mojo. Both methods return objects that need to be casted to what you want to look at, and as testers of the mojo in question it should be perfectly fine to just cast away based on your knowledge of the mojo internals.

For example if I needed to check the value of the groupId in the project instance in a given mojo this is what I would do.

```
// get the project variable
MavenProject project = (MavenProject)getVariableValueFromObject( mojo, "project" );

// get the groupId from inside the project
String groupId = (String)getVariableFromObject( project, "groupId" );

// assert
assertEquals("foo", groupId);
```

things to watch out for

I have noticed that since the declaration of the patchs are not directly in the same location as the verification (one being with the plugin configuration and the other in the assert clauses) that it is very easy to get them mixed up. And if you are validating that the file doesn't exist at the end of the operation as with the clean plugin, if you are not testing the absolute right thing then your assert will be working when the test itself might have failed. So make sure you check these conditions, perhaps even asserting something is there before the mojo execution and then asserting it isn't there at the end.

this might actually be made easier with the addition of the ability to set and get variables in the instantiated mojo from the java side of the fence