DDTUnit v.0.8.5

Project Documentation

Hosted by SourceForge **02 July 2007**

Table of Contents

1.1 Summary

DDTUnit

DDTUnit - A Data Driven Approach to Unit Testing
For setting up module/component tests during development cycle we found it extremely useful and productive to take a data centric approach to cover multiple testcases on coded testmethods.
The basic idea of DDTUnit is to provide a XML description (XML Schema based) of testdata and combine it with the simplicity of JUnit. All program flow is coded in plain old Java. A wide range of object types can be expressed through the definition in xml resource (not just the simple String constructor based objects.

Switched to Subversion repository

- New! Add: date format selection for date type hint="date". See Details
- New! Add: DDTUnit configuration properties added. See Configuration Details
- New! Fix: Execution order of tests identical to order in xml resource.
- New! Fix: Support Java 5 enum as hint="constant".
- New! Fix: Hashtable not identified as Map.
- New! Fix: Defining primitive array fields.

More detailed information in section DDTUnit - Changes

Request for Comment:

- Ask for new features and usage, report bugs
- Ask the Developer

1.2 Ideas and Features

Ideas and Features

For setting up module/component tests during development cycle we found it quite helpful and efficient to take a data centric approach. This provides a means of easily covering all specified testcases. Most commercial tools provide a special dialect of C or a scripting language to define parametrized test scripts. In our experience this approach was difficult to adapt in teams that do development in Java under a short schedule. - "Oh not jet another language or tool to distract us from work". And in projects with rather low budget you even won't get the money to buy these expensive tools.

The basic idea of DDTUnit:

- To achieve an easy integration of this framework into existing <JUnit> infrastructure the class `DDTTestCase` is directly extended from <JUnit> `TestCase` class. This approach provides combination of XML data definitions with the simplicity of use of <JUnit>.
  And keep highly compatible with <JUnit> - that means existing TestRunners like Ant Optional task `<junit>` can be used for test execution and result presentation.
- The new class `DDTTestCase` contains an implicite loop-execution of test resources collected for one <Java> / <JUnit> testmethod to process parameters as provided by xml data resource. So there is no need of recompilation if new testcases were found. These can easily be added to the xml data resource.
- Mixing simple <JUnit> testmethods (no external testdata) with DDTUnit methods. This is highly useful for a simple migration to the new framework.
- The xml parsing is done by providing a SAX parser that can do validation based on a XML Schema definition. The SAX API was chosen for speed and small memory footprint.
- Use of XML schema based definition of testdata. The XML is designed to define data structures and no program flow information. The structure itself is oriented on grouping of data on testmethods (called <group>) and testclasses (called <cluster>). This reflects the grouping of data blocks on an execution and the clustering reflects more loosly collected data on a specific theme.
- Defining a representation of objects as test parameters not just String based ones, but different container classes and more complex structures as used for value objects in distributed environments. To fulfill this requirement <DDTUnit> provides a lot of different data types out of the box. For a detailed description refer to the chapter Defining Objects. A few examples can be found there as well.
- Defining assertions on expected objects or exceptions inside of xml resource. This feature becomes more interesting if inputdata changes the behavior of the method under test, e.g. throw an exception instead of a value. And will play a special role in future reporting features of <DDTUnit>

To give an impression of data definition structures using xml resources here follows an example of xml resource structure as used by <DDTUnit>.

```
    Simple XML Example
<?xml version="1.0" ?>
<ddtunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="http://ddtunit.sourceforge.net/ddtunit.xsd">
  <cluster id="SimpleDDTUnitTest">
    <!-- group associated to a testmethod of implemented test -->
```

```
    <group id="testRetrieveAssertData">
      <!-- test associated to one testmethod execution -->
      <test id="myFirstTestCase">
        <objs>
          <obj id="myObj" type="string">My first String </obj>
           ...
        </objs>
        <asserts>
          <assert id="result" type="string" action="isEqual">
            My first String </assert>
          <exception id="expected" type="java.lang.Exception"
            action="isSimilar" />
           ...
        </asserts>
      </test>
      <test id="mySecondTestCase"> ... </test>
    </group>
  </cluster>
</ddtunit>
```

The interesting structures are

- <obj> that contains plain old java objects that can be used as parameters of functions under test (FUT) and objects to check against output of FUT by using <Java> or <JUnit> assert functions.
- <assert> is used to define assertions as provided by <JUnit> but positioned inside of the xml data file to provide a single point of information to basic testing activities. This will be especially useful when combining execution results with test definition information to generate test protocols.
- <exception> to define expected exceptions that should be caught during test method execution.

There are two specialties to easily map <DDTUnit> on <JUnit> processing.

- The attribute `cluster@id` is a free selectable identifier. This gives the opportunity to reuse one xml data file in multiple test classes.
- The attribute `group@id` must be equal to the name of an existing testmethod inside of the defined testclass.
  This restriction allows an easy mapping between xml data and methods and a free mix of standard <JUnit> testmethods and <DDTUnit> methods inside of one testclass definition.

For future development it is under plan to incorporate the best of bread testing frameworks like DBUnit, Jakarta-Cactus, EasyMock and perhaps other testing frameworks to support the data centric approach. A good Ant integration with strong reporting features is planned as well.

Even the integration with the loadtest tool Grinder3 is under consideration, so it will be possible to fire testdata randomly accessed from datafiles against an environment under loadtest with grinder.

All functionality should be verified by <JUnit> or later by using <DDTUnit> itself. The Test Driven approach will be adopted as far as possible during development of this framework.

1.3 A short History

A short History

The first steps on developing a datacentric testing framework began in summer 2002 after supporting a diploma thesis on Testing in a J2EE environment and its special requirements on basis of <JUnit> and it's extensions.
Until then I had covered good ground in using <JUnit> in standard application development and in testing my selfwritten service-based framework to support development of J2EE components and standard applications alike - This was done as inhouse project only.

Even in this first attempt to make developer life a little easier my major goal was to combine different very good frameworks under the hood of an easy to use framework with a rather simple API. - Why invent logging facilities when there is such an excellent logging framework as LOG4J of the Apache Jakarta project or why try to invent templating mechanisms if you can use a rock solid Velocity from Apache Jakarta project.

Why not use an existing framework then?
After a research over the web I found two frameworks that apealed to me by the basic ideas and goals they aimed - a data centric approach to unit testing.

- JXUnit on Sourceforge
- JTestCase on Sourceforge

In my opinion the major drawback in JXUnit was the complexity of usage.

1. Define a data file - thats fine with me.
2. Define a mapping file (in XML) to specify the mapping between xml and Java.
3. Compile the mapping definitions by using jet another framework: Quick4 .
   As developers with a rather short schedule my colleagues didn't want to learn this extra things. They just wanted to write test data as simple and understandable as writing <JUnit> TestCases.

With this in mind I stumbled over JTestCase on Sourceforge.
This project used direct data structure description using <Java> datatypes and a mapping of XML to <JUnit> class structure by supporting basic object instanciations. The following example is taken from the JTestCase project version 2.1.0 on Sourceforge

```
<?xml version ="1.0" encoding = "UTF-8"?>
<tests xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="c:/Programme/eclipse/workspace/JTESTCASE/config/jtestcase.xsd">
    <class name="JTestCaseTest">
        <method name="testDateType" test-case="first-testcase">
            <params>
                <param name="a_JavaUtilDate"
type="java.util.Date">01.08.2003</param>
            </params>
        </method>
        <method name="testParamGroups" test-case="second-testcase">
            <params>
                <param name="key" type="java.lang.String">key</param>
```

```
                ...
            </params>
        </method>
        ...
    </class>
</tests>
```

Because of the simple structure and the ease of use JTestCase set the foundation for further research and development.

The major points JTestCase was missing then were

- no direct extension from <JUnit> TestCase class. Just a set of helper classes to be used for data retrieval.
- A missing mapping for arbitrary 'value object' like <Java> structures.
  We are using **a lot** of them in actual projects. A good part of these objects are write protected, that means only creatable with special constructor or through <Java> reflection.
- No referencing between objects like used in complex structures as order objects or other business structures.

The lack of this features set the gound for the basic ideas and features of <DDTUnit>.

2.1 Defining Objects

Defining Objects

```
<objs><obj id="myId" type="my.class.Type" hint="myParseHint">...</obj><objs>
```

Defining objects inside of the xml data files of <DDTUnit> should be as clear and easy as to write the appropriate code in Java.
To achieve this the object structure was mapped onto the xml definition as straight as possible.

As a starting point the following object types are supported:

- all objects that contain a constructor of the form `public MyClass(java.lang.String)`.
- all objects containing a default constructor and using (even private) field definitions can be generated. <Java> reflection is used to map xml field names on objects field definitions.
  (See Generic Structures)
- all objects containing a default constructor and using setter methods according to JavaBean spec can be generated.
  (See Generic Bean Structures)
- For now two container object implementations are supported. (See Container Structures).
- Calling constructor or static methods of object is supported in a first implementation. (See Constructor Structures).
- Constants are supported as defined by static fields of classes and typesafe enumerations. (See Constant Structures)
- Special data types and keywords. (See Object Specialties)
- Shortcut notation using object type abbreviation. (See Object Specialties)

The testdata can be accessed by using the following API called from `DDTTestCase` class:

```
DDTTestCase
  public Object getObject(String id); // or
  public Object getObject(String id, String type);
  public Object getGlobalObject(String id); // or
  public Object getGlobalObject(String id, String type);
  public Object getResourceObject(String id); // or
  public Object getResourceObject(String id, String type);
```

A general remark on XML structure used for object definition. Basically every object definition possesses three attributes to describe it.
These are:

1. `obj@id` and `obj@type` allow identification of objects. If there are only a few objects defined per test and these are uniquely identified by using the `obj@id` the first method can be used. If multiple objects were found this method will throw a `DDTException` and the second method definition

must be used.

2. `obj@hint` - (Default: fields) specifies parsing information for internal processing

2.1.1 Specialties

Object Specialties

This section contains a set of loosly coupled information that will not take a complete section on its own.

Special Keywords

Name	Description
!NULL!	Used to specify that a certain field/object should be null
!EMPTY!	Used to specify an empty String object. - Restricted to type String!
!BLANK!	Used to specify a blank String (one Space char). - Restricted to type String!

Special Keywords

Here is an example

```
<obj id="myExample" type="junitx.ddtunit.resources.SimpleVO">!NULL!</obj>
```

will initialize the instance of SimpleVO with null. So use special key '!NULL! every time you want to explicitly assign **null** to an object.

Using Simple Date

Processing information about date - `java.util.Date, java.sql.Date, java.text.Date`
Three examples

```
<obj id="myDate" hint="date" type="java.util.Date">01.05.2005 00:00:00.000</obj>
```

```
<obj id="myDate" hint="date" type="date">01.05.2005 00:00:00</obj>
```

```
<obj id="myDate" hint="date" type="udate">01.05.2005</obj>
```

The predefined date formats are specified in the resource /junitx/ddtunit/ddtunitConfig.properties.
For specifiing your own format the new xml attributes 'dateformat' and 'locale' were introduced. A simple example

```
<obj id="myDate" hint="date" type="date" dateformat="EEE MMM dd HH:mm:ss zzz yyyy"
locale="en_US">
  Thu Dec 06 12:15:00 CET 2007</obj>
```

More examples can be found in the test junitx.ddtunit.functest.ProcessDateDataTest. Because

`java.util.Date` implements `java.lang.Comparable` the <DDTUnit> assert actions on this type can be used.

To specify an actual date or time new keywords are introduced

Name	Description
!SYSDATE!	Specifies the actual date of test execution, time is set to zero
!SYSTIME!	Specifies the actual date and time of test execution.

Special Keywords

Using Selfdefined Date Formats

Because there are

2.1.2 TypeAbbreviator

Abbreviation of Object Type Definitions

To allow a shorthand on <Java> types the `junitx.ddtunit.data.processing.TypeAbbreviator` is introduced to the API. Executed as standalone application it lists all known type abbreviations to `System.out`.
More on configuration. Here is the list as defined in <DDTUnit> version 0.6.6:

Type Shortcut	Long Expression
byte, Byte	java.lang.Byte
boolean, Boolean	java.lang.Boolean
char, Character	java.lang.Character
string, String	java.lang.String
short, Short	java.lang.Short
int, Integer	java.lang.Integer
long, Long	java.lang.Long
bigint, BigInteger	java.math.BigInteger
float, Float	java.lang.Float
double, Double	java.lang.Double
bigdecimal, BigDecimal	java.math.BigDecimal
arraylist, ArrayList	java.util.ArrayList
vector, Vector	java.util.Vector
hashmap, HashMap	java.util.HashMap
hashtable, Hashtable	java.util.Hashtable
udate	java.util.Date
date, Date	java.util.Date
sdate	java.sql.Date
tdate	java.text.Date

Type Abbreviator Dictionary

2.1.3 Using Default Constructor and Fields

Defining Objects By Default Constructor and Field Definitions

```
<objs><obj id="myId" type="my.class.Type" hint="fields">...</obj><objs>
```

To define value objects by field names you need to know the internal naming of the fields you want to populate. For mapping <Java> reflection is used. The access on private and protected fields is done using <JUnit Addons> `PrivateAccessor`. - Implementation details might be subject to change.
Here follows an example:

```
...
  <test id="myFirstTestCase">
   <objs>
    <obj id="myObj" type="junitx.ddtunit.resources.SimpleVO">
      <doubleValue>12.4</doubleValue>
      <integerValue>4711</integerValue>
      <stringValue>My Text</stringValue>
    </obj>
   </objs>
  </test>
...
```

Object `myObj` reflects the Java object definition

```
public class SimpleVO {
    private Integer integerValue;
    private String stringValue;
    private Double doubleValue;

    /**
     * Default constructor.
     */
    public SimpleVO() {
        // no special initialization neccessary
    }
...
```

If you are using an empty object description

```
    <obj id="myObj" type="junitx.ddtunit.resources.SimpleVO" />
```

you will get the spcified object as long as the default constructor is accessible (public).

To specify a "null" assignment use the explizit description

```
    <obj id="myObj" type="junitx.ddtunit.resources.SimpleVO">!NULL!<obj>
```

2.1.4 Using Default Constructor and Bean

Defining Objects By Default Constructor and Bean Definitions

```
<objs><obj id="myId" type="my.class.Type" hint="bean">...</obj><objs>
```

To define bean objects you need to know the naming of the field setter method (without set prefix) and the datatype of the argument used for this method.
To use this functionality could be interesting in the context of providing special validation processing or calculation inside of the setter methods.
The structure looks like this:

```
...
<test id="myFirstTestCase">
  <objs>
    <obj id="myObj" type="junitx.ddtunit.resources.SimpleVO" hint="bean">
      <doubleValue>12.4</doubleValue>
      <integerValue>4711</integerValue>
    </obj>
  </objs>
</test>
...
```

where object `obj@id="myObj"` reflects a Java object definition like the following for example

```
public class SimpleVO {
  private Integer integerValue;
  private String stringValue;
  private Double doubleValue;

  /**
   * Default constructor.
   */
  public SimpleVO() {
    // no special initialization neccessary
  }

  public void setDoubleValue(Double arg){
    // whatever check or calculation you like to process ...
    this.doubleValue = arg;
  }

  public void setIntegerValue(Integer arg){
    // whatever check or calculation you like to process ...
    this.integerValue = arg;
  }
...
```

If you are using an empty object description

```
<obj id="myObj" type="junitx.ddtunit.resources.SimpleVO" hint="bean"/>
```

you will get the specified object as long as the default constructor is accessible (public). This is the same behavior like for `hint="fields"`.

To specify a "null" assignment use the explizit description

```
<obj id="myObj" type="junitx.ddtunit.resources.SimpleVO" hint="bean">!NULL!<obj>
```

2.1.5 Using Constructor and Method Calls

Defining Objects By Constructor and Method Call

```
<objs><obj id="myId" type="my.class.Type" hint="call" calltype="my.call.Type"
>...</obj><objs>
```

To define objects according to implemented specifications you must be able to call the constructor of this object. Or even call a static method of a factory class. This is just the way you do it in real application code.
Here is an example:

```
...
  <test id="mySecondTestCase">
   <objs>
    <obj id="myObj" type="junitx.ddtunit.resources.SimpleVO" hint="call">
      <item type="string">My Text</item>
      <item type="int">4711</item>
      <item type="double">12.4</item>
    </obj>
   </objs>
  </test>
...
```

The definition of object `id="myObj"` reflects the use of the Java object constructor as defined below

```
public class SimpleVO {
...
    public SimpleVO(String text, Integer intValue, Double doubValue) {
        // whatever your class should do on instanciation
    }
...
```

This is equivalent to

```
<obj id="myObj" type="junitx.ddtunit.resources.SimpleVO"
  calltype="junitx.ddtunit.resources.SimpleVO" hint="call" method="constructor">
  <item type="string">My Text</item>
  <item type="int">4711</item>
  <item type="double">12.4</item>
</obj>
```

So you can see that there are three attributes to specify a method/constructor invokation:

- hint="call" - to specify the use of method/constructor call
- method="methodName" - specifying the method to call. If you want to use a constructor you can specify method="constructor" or just leave this attribute empty.
- calltype="my.call.Type" - specifies the class to call the method from. If it is the same as the resulting

type specified by type attribute you can leave it empty as well.
If you want to call a method having no arguments write it like this:

```
<obj id="myObj" type="junitx.ddtunit.resources.SimpleVO"
  hint="call" method="toString" />
```

If you specify non static methods the framework will try to instanciate a calltype object by using default constructor.

Actually it is not important to use the `<item/>` tag name to specify the parameters of the constructor. You can use any name that supports an expressive meaning for the declaration.
To resolve the parameters only the `type` attribut and the order of the parameters is used to specify the constructor call.
So the example above is equivalet to:

```
<obj id="myObj" type="junitx.ddtunit.resources.SimpleVO"
  calltype="junitx.ddtunit.resources.SimpleVO" hint="call" method="constructor">
  <stringValue type="string">My Text</stringValue>
  <integerValue type="int">4711</integerValue>
  <doubleValue type="double">12.4</doubleValue>
</obj>
```

This declatation reflects the mapping of the constructor parameters on the instance valiables of the class `SimpleVO`.

2.1.6 Using Constant Objects

Defining Constant Objects

```
<objs><obj id="myId" type="my.class.Type" hint="constant">...</obj><objs>
```

To define constants you need to know the naming of the fields you want to reference. The attribute `obj@type` is that of the class containing the constant field you want to reference. In general this is not the type of the field itself.
The only restriction stated: the field must be defined static.
The Java 5 Enumeration type is supported as well. Here is an example:

```
<test id="myFirstTestCase">
   <objs>
     <obj id="myObj" type="junitx.ddtunit.resources.SimpleConstants"
       hint="constant">MY_STRING_CONSTANT</obj>
   </objs>
 </test>
```

where the object identified by `obj@id="myObj"` reflects a Java object definition like in the next example

```
 public class SimpleConstants {
  public final static String MY_STRING_CONSTANT = "Hallo World";
  /**
   * Default constructor.
   */
  private SimpleConstants() {
    // no special initialization neccessary
  }
```

```
<test id="mySecondTestCase">
   <objs>
     <obj id="myObj" type="junitx.ddtunit.resources.MyEnumerator"
       hint="constant">FIRST_ENUMERATOR</obj>
   </objs>
 </test>
```

where the object identified by `obj@id="myObj"` reflects a Java object definition like in the next example

```
 public public enum MyEnumerator {
        FIRST_ENUMERATION, SECOND_ENUMERATION, NULL
}
```

2.1.7 Using Containers

Defining Object Containers

```
<objs><obj id="myId" type="my.class.Type" hint="valid container
type">...</obj><objs>
```

Because containers play a vital part in the object life of projects this section is dedicated to the creation of different conainer types known by <Java>.
Headstart takes

1. the group of `Collection` classes.
2. and the group of dictionary like `Map` classes.

2.1.7.1 Using Containers - Collection

Using Containers - Collections

```
<objs><obj id="myId" type="my.class.Type" hint="collection">...</obj><objs>
```

All collection classes in <Java> are derived from `java.util.Collection`.
This specimen is used to store a set of values (whatever type you like) in a box. The general structure in <DDTUnit> looks like this:

```
<obj id="myVector" type="java.util.Vector" hint="collection">
  <item type="string">firstEntry</item>
  <item type="string">secondEntry</item>
  <item type="string">thirdEntry</item>
</obj>
```

And here is the equivalent example in shortcut notation:

```
<obj id="myVector" type="vector" hint="collection" valuetype="string">
  <item>firstEntry</item>
  <item>secondEntry</item>
  <item>thirdEntry</item>
</obj>
```

and at last a mixed value example for collection:

```
<obj id="myVector" type="vector" hint="collection" valuetype="string">
  <item>firstEntry</item>
  <item type="int">4711</item>
  <item>thirdEntry</item>
</obj>
```

This will instanciate to a sequence of `[String, Integer, String]`.

The `item` tags can contain every structure that is allowed for standard `obj` tags as well.
To specify an empty collection just use an empty xml tag:

```
<obj id="myVector" type="vector" hint="collection" valuetype="string" />
```

To specify the collection object as "null" use the special key `!NULL!`

```
<obj id="myVector" type="vector" hint="collection" valuetype="string">!NULL!</obj>
```

Here is an example from the <DDTUnit> test suite.

2.1.7.2 Using Containers - Map

Defining Object Containers - Maps

```
<objs><obj id="myId" type="my.class.Type" hint="map">...</obj><objs>
```

All map or dictionary like classes in <Java> are derived from `java.util.Map`.
This class can be used to store a set of key/value pairs. The general structure looks like this:

```
<obj id="myMap" type="java.util.HashMap" hint="map">
  <item>
    <key type="java.lang.String">firstkey</key>
    <value type="java.lang.String">firstValue</value>
  </item>
  <item>
    <key type="java.lang.String">secondkey</key>
    <value type="java.lang.String">secondValue</value>
  </item>
</obj>
```

The `key` and `value` tags can contain every structure that is allowed for standard `obj` tags as well.
To specify a mixed key/value constallation just provide different type attributes to the `key` or `value`.
If you prefere shortcut notation with homogenous key/value pairs define it like this:

```
<obj id="myMap" type="java.util.HashMap" hint="map"
  keytype="java.lang.String" valuetype="string">
  <item>
    <key>firstkey</key>
    <value>firstValue</value>
  </item>
  <item>
    <key>secondkey</key>
    <value>secondValue</value>
  </item>
</obj>
```

As in all the other examples you can use type shortcuts for all types as defined in Object Type Abbreviation.

To specify an empty map just use an empty xml tag:

```
<obj id="myMap" type="hashmap" hint="map" keytype="string" valuetype="string" />
```

To specify the map object as "null" use the special key `!NULL!`

```
<obj id="myMap" type="hashmap" hint="map" keytype="string"
valuetype="string">!NULL!</obj>
```

2.1.8 Using Arrays of Objects

Defining Arrays of Objects

```
<objs><obj id="myId" type="my.class.Type" hint="array" >...</obj><objs>
```

To define an array of objects you have to specify the `hint="array"`. The structure is the same as used in collection datatype definition.
Here comes an example:

```
<test id="myFirstTestCase">
  <objs>
    <obj id="myObj" type="int" hint="array">
      <item>4711</item>
      <item>4712</item>
    </obj>
  </objs>
</test>
```

Using this object is equivalent to use:

```
Integer [] myObj = { new Integer(4711), new Integer(4712)};
```

If you want to specify an empty array of an array of a certain size you can proceed like this:

```
<obj id="myObj" type="int" hint="array" />
```

will specify an integer array of size 1 with null element.

```
<obj id="myObj" type="int" hint="array">3<obj>
```

will specify an array of size 3 will "null" elements.

So the only way to assigning an array object to "null" is the explizit way

```
<obj id="myObj" type="int" hint="array">!NULL!<obj>
```

2.1.9 Using Object References

Object References

```
<objs><obj id="myId" type="my.class.Type" refid="myRefClass"</obj><objs>
```

This implementation is just a first draft and will be subject to change in the future.
Any ideas, feature requests and error reports are welcome on this subject!

Why using References anyway?

One question I was ask during a short presentation of the framework
Why use references anyway? - Is'nt it just to complicated and error prone to use ? I think there are always two sides of a medal. On one hand you have the possibility to do a lot of shorthand writing through the use of referencing. And on the other hand you have to know what you are dealing with in terms of implementation details and their influence on the execution model and the lifespan of object instances.

Upside of the Medal

So lets start with the obvious benefit of extracting common object definitions to the global `cluster/objs` section and reference these objects throughout different tests.
The idea for this example was provided by Ted Velkoff who provided to me a lot of valuable input and discussion on the use of <DDTUnit>.
Lets say we are using a complex data structure to manage calendar date/time functionality by with range considerations. E.g.

```
public class CompositeDateTime{
  private CompositeDate date;
  private CompositeTime time;
// a lot of special processing methods on date and time
...}
```

The first field splits up into a number of fields as well:

```
public class CompositeDate{
  public CompositeDate(){}

  private int day;
  private int month;
  private int year;
// a lot of specific methods on date processsing
...}
```

So if I have to specify such a nested structure in an <DDTUnit> xml resource it could become quite tiring to do so without using references. Especially if you have to use specific date/time objects again and again because these where the ones you identified during equivalence class analysis of your test data.

The idea now is to specify a set of objects in the `resources/objs` or `cluster/objs` section of the <DDTUnit> xml resource and reference these.
If you change your mind on the specified objects later, you only have to change these globally defined objects but not the whole xml resource.
To get back to the example:

```
...
  <cluster id="RangeCompositeDateTimeDDTUnitTest">
    <objs>
      <obj id="date20060401"
type="com.velkoff.articles.ddtunit.example.CompositeDate">
       <year>2005</year>
       <month>4</month>
       <day>1</day>
      </obj>
...
   <test id="testUpperLower">
    <objs>
     <obj refid="upper"
type="com.velkoff.articles.ddtunit.example.CompositeDateTime">
      <date refid="date20060401"
type="com.velkoff.articles.ddtunit.example.CompositeDate" />
      <time reftype="" com.velkoff.articles.ddtunit.example.CompositeTime" />
       <hours type="int">15</hours>
       <minutes type="int">3</minutes>
       <seconds type="int">27</seconds>
       <millis type="int">348</millis>
      </time>
     </obj>
...
```

Downside of the Medal

The other side of the medel contains details about the lifespan of object instances.
Because <DDTUnit> implements an object repository that can contain objects that have longer lifespans than a single `test` it is important to check the use of objects that are used as `global` or even `resource`.

If you are sure these objects will not be modified during test execution there will be no problem at all. But if you are not sure about this of sometimes willingly provide objects that should be modified during a number of test executions - like in the standard case of a functional test - you have to double check if your expectations on the provided object hold true any more.

2.2 Asserts on Objects

Asserts on Objects

```
<asserts><assert id="myId" type="my.class.Type" hint="myParseHint"
action="MYACTION">...</assert><asserts>
```

For implementing a solid support of assertion features the <JUnit Addons> library from Sourceforge is added as utility archive under the terms of the Apache Software License. The possibilities of defining asserts is mapped directly on the assert methods as defined inside of <JUnit> and <JUnit Addons>. The following actions are implemented for now:

Name	Description
isEqual	Maps to <JUnit> Assert.assertEquals().
isNotEqual	Maps to <JUnit Addons> Assert.assertNotEquals().
isSame	Maps to <JUnit> Assert.assertSame().
isNotSame	Maps to <JUnit Addons> Assert.assertNotSame().
isNull	Maps to <JUnit> Assert.assertNull().
isNotNull	Maps to <JUnit Addons> Assert.assertNotNull().
isTrue	Maps to <JUnit> Assert.assertTrue().
isFalse	Maps to <JUnit Addons> Assert.assertFalse().
isContainedIn	Maps to <JUnit Addons> Assert.assertTrue(((Collection)obj).contains(actual)).
isNotContainedIn	Maps to <JUnit Addons> Assert.assertFalse(((Collection)obj).contains(actual)).
	used for `java.lang.Compareable`
isLT	actual is smaller as expected.
isNotLT	actual is not smaller (greater or equal) as expected.
isGT	actual is greater as expected.
isNotGT	actual is not greater (smaller or equal) as expected.
isInRange	actual is in specified range according to Comparable rules.
isNotInRange	actual is not in specified range according to Comparable rules.

Valid Assert Actions

The possibility of defining asserts inside of xml gives two advantages over directly using the `obj` tag and plain %junit; asserts inside the code.

- You clearly distinguish between object definitions and assert information.
- You can use these definitions for reports on test execution and what was checked during test. And therefore can provide an active specification document of your class under test.

All object structures that can be defined under `<objs>` can be used for defining expected objects inside of `<assert>`. Just remember the importance of overwriting the `equals()` and `hashCode()` methods of the objects to make them even more valuable to use under <JUnit> / <DDTUnit>.
Here is a short example:

```
<assert id="complete" action="isEqual" type="junitx.ddtunit.resources.SimpleVO">
  <doubleValue>12.4</doubleValue>
  <integerValue>4711</integerValue>
  <stringValue>My Text</stringValue>
</assert>
```

Which is interpreted like this:
Check an actual object provided during test execution against the expected object stored under `assert@id` **complete** and `assert@type` **junitx.ddtunit.resources.SimpleVO**.
The <DDTUnit> testmethod would look something like this:

```
public void testMyService(){
  SimpleVO simpleVO = MyServiceUnderTest.getSimpleVO();
  addResultToAssert("complete", simpleVO);
}
```

To make this work right the object `simpleVO` must implement the methods `equals()` and `hashCode()` to meet a clear expectation of an equality comparison of this type. The same holds true for the use of <JUnit> as well.

To be complete: There are three methods defined under `DDTTestCase`

- public addObjectToAssert(String id, Object toAssertAgainst);
- public assertObject(String id, Object toAssertAgainst, boolean mark);
- and public assertObject(String id, Object toAssertAgainst);

The first method provides information for assertions at the end of the <JUnit>/<Java> testmethod execution.
The second and third are used for direct evaluation of the defined assertion. If the boolean mark is set, no double validation of this assertion will be done at the end of the test method.

As the class `DDTTestCase` is a direct derivation of <JUnit> `TestCase` it is possible to use all assert methods of `junit.framework.Assert` directly in the code. It is also possible to use your own assertion extensions in the testmethod.
Because the provided testdata change, so do the required assertions.
As a rule of thump use this kind of assertions every time you have correlation with parameter data of xml resources. This provides information about the contract or specification you want to test.

What are <JUnit> assert methods good for?
Every time you want to check pre or post conditions of a data centric check use <JUnit> assert methods. These should be used similar to the <Java> assert extension. For example if you want to check the result of a customer list returned by a business function.

- First check if the list is not null by using <JUnit> `assertNull(...)` .
- Evaluate the correct list content by using <DDTUnit> assert definition.

As by using every rule it is up to you to select the appropriate solution.

The assert action `isContainedIn` is used to validate if an object is in a specified set of objects. To define such an assert you must specify a `Collection` containing expected values. For example:

```
<assert id="testResult" type="java.util.Vector" hint="collection"
        valuetype="boolean" action="isContainedIn">
        <item>true</item>
        <item>false</item>
</assert>
```

The assert action `isInRange` is used on `java.lang.Comparable` datatypes to validate if an object is in a specified range. To define such an assert you proceed as in the following example.

```
<assert id="result" type="range" action="isInRange">
  <startIncluded>false</startIncluded>
  <start type="long">4711</start>
  <endIncluded>false</endIncluded>
  <end type="long">4759</end>
</assert></asserts>
```

For now you must specify `start@type` and `end@type`. This behavior will be replaced by `assert@valuetype` in future.
By default the end points of the interval are included in the specified range. To exclude one end of the range specify the appropreate tag `<startIncluded>` or `<endIncluded>`.

2.3 Catching Exceptions

Catching Expected Exceptions

```
<asserts><exception id="myId" type="my.exception.Type" hint="myParseHint"
action="MYACTION">...</exception><asserts>
```

In fact a lot of testing goes into checking if certain exceptions are raised as expected error conditions. To support this feature in <JUnit> you would use a code snipplet like this

```
  public void testExpectedException(){
    try{
      instanceUnderTest.methodUnderTest(myParam);
      fail("An exception was expected");
    } catch (MyExpectedException e){
      // whatever special checks you want to add
      // if pass, all is fine
    }
  }
```

In <DDTUnit> this task would look as following:

```
...
  public void testExpectedException(){
      my.param.Clazz myParam = (my.param.Clazz) getObj("myParam");
      instanceUnderTest.methodUnderTest(myParam);
  }
...
== with the xml definition:
  <test id="testExpectedException">
   <objs>
     <obj id="myParam" type="my.param.Clazz">Content</obj>
   </objs>
   <asserts>
     <exception id="expectedException" type="my.expected.Exception"
        action="isEqual">Message of Exception</exception>
   </asserts>
  </test>
```

If no exception is raised during method execution but one was defined, a `junit.framework.AssertFailedError` is thrown.
Exception definition classes must be derived from type `java.lang.Throwable`.
Possible actions are

Name	Description
isEqual	Two exceptions are equal if they are of the same type and they contain the same method as defined by `getMessage()`
isSimilar	Two exceptions are similar if they are of the same type and the message of the defined exception is contained in the one of the actual exception.

Valid Assert Actions

Name	Description
isInstanceOf	One exception object is of the same instance as a specified class type if it has the same type or a derived type as the specified class.

Valid Assert Actions

Because expected exceptions are a special form of assertions they are placed near the assert definitions under `<asserts>`.

Here you can see an example taken from the test suite.

3.1 Cookbook

A Cookbook to DDTTestCase

This cookbook is related to the <JUnit> Cookbook by Gamma and Beck. I selected the same document structure to clearly show the similarities and the differences between these two frameworks.

The major differences of the two are

- you do not write tests by overwriting the method `runTest()` of the subclassed `DDTTestCase`. This way of test writing is not supported by <DDTUnit>. So in the following sections all examples from <JUnit> cookbook containing this technic will be omitted.
- you need to specify a xml resource for every `DDTTestCase` class by overwriting the abstract method `void initContext()`.

Writing Tests

<DDTUnit> like <JUnit> tests do not require human judgment to interpret, and it is easy to run many of them at the same time. For testing requirements here is what you have to do:

1. Create an instance of DDTTestCase by subclassing it.
2. Create a constructor which accepts a String as a parameter and passes it to the superclass. So this is simply the same as <JUnit> test case.
3. Implement the abstract method `initContext()` of `DDTTestCase` to specify the resource containing the xml description of test data and the `clusterId` under which the data is accessible. For convenience the methods
   - `void initTestData(String resource, String clusterId)` or
   - `void initTestData(String clusterId)`

   can be used as body of the `initContext()` method. The later is mapped onto the first by duplicating the parameter.
4. Write test methods that start with prefix `test`. E.g. `void testMyService()`. Here you can access xml defined objects by using method `getObject(String)` e.a. as provided by DDTTestCase. Where the parameter denotes the name of the object to retrieve from internal cache as specified in xml resource.
5. If you want to check a value, call `assert...()` method as of standard <JUnit>. Or specify asserts inside of the xml resource on a per <test> basis and provide the actual object through `addObjectToAssert();`

For example, to test that the sum of two Moneys with the same currency contains a value which is the sum of the values of the two Moneys, write:

```
public void initContext(){
  initTestData("/DDT-TestResource.xml", "MyClassTest");
}

public void testSimpleAdd() {
  Money m12CHF= (Money)getObject("m12CHF");
  Money m14CHF= (Money)getObject("m14CHF");
```

```
  Money result= m12CHF.add(m14CHF);
  // here is the JUnit assert way of testing
  Money expected= (Money)getObject("m26CHF");
  assertTrue(expected.equals(result));
  // here is part of DDTUnit way of testing
  addObjectToAssert("result", result);
}
```

Whats missing now is the xml testdata description of the requested objects like `m12CHF`. Here it comes

```
<?xml version="1.0">
<ddtunit>
  <cluster id="MyClassTest">
    <group id="testSimpleAdd">
      <test>
       <objs>
        <obj id="m12CHF" type="Money">
          <amount>12</amount>
          <currency>CHF</currency></obj>
        <obj id="m14CHF" type="Money">
          <amount>14</amount>
          <currency>CHF</currency></obj>
       </objs>
       <asserts>
        <assert id="result" type="Money" action="isEqual">
          <amount>26</amount>
          <currency>CHF</currency></assert>
       </asserts>
      </test></group>
  </cluster>
</ddtunit>
```

If you want to write a test similar to one you have already written, write a Fixture or if only input and expected output changes just add a new <test> to the xml resource.
When you want to run more than one test, just create a Suite like under <JUnit>.

Fixture

What if you have two or more tests that operate on the same or similar sets of objects?
Tests need to run against the background of a known set of objects. This set of objects is called a test fixture. When you are writing tests you will often find that you spend more time writing the code to set up the fixture than you do in actually testing values.

To some extent, you can make writing the fixture code easier by paying careful attention to the constructors you write. However, a much bigger savings comes from sharing fixture code. Often, you will be able to use the same fixture for several different tests. Each case will send slightly different messages or parameters to the fixture and will check for different results.

When you have a common fixture, here is what you do:

1. Create a subclass of DDTTestCase
2. Create a constructor which accepts a String as a parameter and passes it to the superclass.

3. Implement the abstract method `initContext()` of `DDTTestCase` to specify the resource containing the xml description of test data and the `classId` under which the data is accessible.
4. Add an <obj> entry in the global section under <cluster> for each part of the fixture and include an instance variable in the <Java> test code.
5. Override `setUp()` to initialize the test conditions by using the globally defined objects.
6. Override `tearDown()` to release any permanent resources you allocated in `setUp`

For example, to write several test cases that want to work with different combinations of 12 Swiss Francs, 14 Swiss Francs, and 28 US Dollars, first create a fixture:

```
public class MoneyTest extends DDTTestCase {
    private Money f12CHF;
    private Money f14CHF;
    private Money f28USD;

    public void initContext(){
      initTestData("/DDT-TestResource.xml", "MyClassTest");
    }

    protected void setUp() {
        f12CHF= (Money)getObject("f12CHF");
        f14CHF= (Money)getObject("f14CHF");
        f28USD= (Money)getObject("f28USD");
    }
}
```

And the xml resource (in part):

```
<?xml version="1.0">
<ddtunit>
  <cluster id="MyClassTest">
   <group id="MyMethod">
    <objs>
      <obj id="f12CHF" type="Money">
        <amount>12</amount>
        <currency>CHF</currency></obj>
      ...
    </objs>
    <group id="testSimpleAdd">
    ...
</ddtunit>
```

The architecture of <DDTUnit> garanties that the Fixture is executed before and after every entry of `<test>` inside of `<group>`. This is conformant to the behavior of <JUnit>. Once you have the Fixture in place, you can write as many tests as you like for one test method just by adding new data to the xml resource.

DDTTestCase

How do you write and invoke an individual test case when you have a Fixture?
Writing a test case without a fixture is simple - just look at the example above. Writing one with a Fixture

is quit as easy.
For example, to test the addition of a Money and a MoneyBag, write:
Remember we allready defined a Fixture above.

```
public void testMoneyMoneyBag() {
    // [12 CHF] + [14 CHF] + [28 USD] == {[26 CHF][28 USD]}
    Money bag[]= { f26CHF, f28USD };
    MoneyBag expected= new MoneyBag(bag);
    assertEquals(expected, f12CHF.add(f28USD.add(f14CHF)));
}
```

Create an instance of MoneyTest that will run this test case like this:

```
  new MoneyTest("testMoneyMoneyBag")
```

When the test is processed, the name of the test is used to look up the method to run. Once you have several tests, organize them into a Suite.

Suite

How do you run several tests at once?
As soon as you have two test methods, you'll want to run them together. You could run the test methods one at a time yourself, but you would quickly grow tired of that. Instead, <DDTUnit> as <JUnit> provides an object, TestSuite which runs any number of test cases together.

For example, to run a single test case, you execute:

```
DDTTestResult result= (new MoneyTest("testMoneyMoneyBag")).run();
```

To create a suite of two test cases concerning two different <Java> test methods and run them together, execute:

```
TestSuite suite= new TestSuite();
suite.addTest(new MoneyTest("testMoneyEquals"));
suite.addTest(new MoneyTest("testSimpleAdd"));
TestResult result= suite.run();
```

If you want to execute a test scenario with different input/output information for the same java test method just add a new `<test>` in the xml test resource.

Another way is to let <DDTUnit> extract a suite from a DDTTestCase class. This is actually the same behavior as of <JUnit>. To do so you pass the class of your DDTTestCase to the TestSuite constructor.

```
TestSuite suite= new TestSuite(MoneyTest.class);
TestResult result= suite.run();
```

Use the manual way when you want a suite to only contain a subset of the test cases. Otherwise the automatic suite extraction is the preferred way. It avoids you having to update the suite creation code when you add a new test method.

TestSuites don't only have to contain TestCases. They contain any object that implements the Test interface. For example, you can create a TestSuite in your code and I can create one in mine, and we can run them together by creating a TestSuite that contains both:

```
TestSuite suite= new TestSuite();
suite.addTest(Joerg.suite());
suite.addTest(Kai.suite());
TestResult result= suite.run();
```

TestRunner

How do you run your tests and collect their results?
For now there is no extra implementation of a `DDTTestRunner`. So you can use a `TestRunner` implementation provided by <JUnit> or other packages like the Apache Ant optional target <junit>. These tools are used to define the suite to be run and to display its results. You make your suite accessible to a `TestRunner` tool with a static method `suite()` that returns a test suite like described above. For example, to make a MoneyTest suite available to a TestRunner, add the following code to MoneyTest:

```
public static Test suite() {
    TestSuite suite= new TestSuite();
    suite.addTest(new MoneyTest("testMoneyEquals"));
    suite.addTest(new MoneyTest("testSimpleAdd"));
    return suite;
}
```

If a TestCase class doesn't define a suite method a TestRunner will extract a suite and fill it with all the methods starting with "test".

3.2 Frequently Asked Questions

Frequently Asked Questions

Configuration

1. How do I initialize the testdata inside of a DDTTestCase?
2. How can I extend shortcut notations for TypeAbbreviator?

All about XML Resources

1. How can I spare time writing xml data types?
2. How should I specify my xml schema definition and where to place it?

All about Testing Strategies

1. How can I test class constructor behavior? - Every time I try to construct my class under test in <DDTUnit> xml resource an exception is thrown during object creation and the processing is stoped.

Problems

1. What can I do to make this and that to work?

Configuration

How do I initialize the testdata inside of a DDTTestCase?

To connect the xml resource to your <Java> testclass you must implement the abstract method `initContext()` in your extension of the `DDTTestCase` class. This method will be executed implicitely during execution preparation.

To facilitate usage there is another method provided my `DDTTestCase`. Here follows an example how to use the two methods in the default behavior.

```
...
public void initContext(){
  initTestData("MyClassResource", "MyClassId");
}
...
```

By providing the two parameters of resource name and classId `DDTTestCase` will search for a resource of name **DDT-**MyClassResource**.xml** in the same ClassPath as the associated testclass . Inside of the resource it will look for the class id provided.
If you provide an absolut resource name starting with `/` the class will use the resource name unchanged.

```
...
public void initContext(){
  initTestData("/my/own/path/to/MyClassResource.xml", "MyClassId");
}
        ...
```

If you like to use the default selection scheme of <DDTUnit> you can simply write

```
...
public void initContext(){
  initTestData("MyClassId");
}
        ...
```

This will be resolved to

```
public void initContext(){
  initTestData("MyClassId", "MyClassId");
}
```

where the resource name is is extended to `/package/name/of/testclazz/DDT-MyClassId.xml`.

Search will be done first on ClassPath then on filesystem.

[top]

How can I extend shortcut notations for TypeAbbreviator?

You will find a properties file `junitx.ddtunit.data.processing.TypeAbbreviator.properties`. This file must be placed in the same package in the classpath as `TypeAbbreviator.class`. To modify it just place a copy into the classpath before searching in the jar. Now you can just add the types you need. - Or send a mail and if it is a class provided along with <DDTUnit> it will be added to the internal properties file.

[top]

All about XML Resources

How can I spare time writing xml data types?

Because it is rather tedious to write something like `java.lang.String` over and over again a special factory was introdused to the package `junitx.ddtunit.data.processing` The class `TypeAbbreviator` contains a dictionary to translate shortcut notatopns used for XML attribute `type` to the correct <Java> type notation. The defined details can be displayed to System.out by just executing this class as <Java> application.

[top]

How should I specify my xml schema definition and where to place it?

Every <DDTUnit> xml resource should start with

```
<?xml version="1.0" ?>
<ddtunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://ddtunit.sourceforge.net/ddtunit.xsd">
  <cluster id="ThisIdShouldBeAdapted">
...
```

There is a good reason not to use jet another local URL to store the xml schema definition. The provided URL is hard coded in the `EntityResolver` of the <DDTUnit> parser and resolves to a resource that resides in the `ddtunit.jar` archive. By doing this

1. you can be sure that you will always use a xml schema that is valid for the active parser
2. and you do not have the need for online connection or the allowance to access the web if you are behind a corporate firewall for example.

[top]

All about Testing Strategies

How can I test class constructor behavior? - Every time I try to construct my class under test in <DDTUnit> xml resource an exception is thrown during object creation and the processing is stoped.

Thanks for the example code and actual question to Ted Velkoff.

The intention of the framework is to provide an environment to validate asserts on a class or service/system contract.
Your investigation is focused on a class or service/system under test (CUT, SUT). To do asserts on the behavior of this CUT, good or bad, you will feed it with different parameters and check against expected results.
With this intention in mind you expect your parameters to be valid in the sense of constructing these. Otherwise you wouldn't even start execution of the cut => no parameters, no run.

With this basic concept of operation <DDTUnit> expects all test parameters provided in the xml resource to be constructable. This will be done before actually executing any of the methods in the test class.
Now you have two choices to phrase the good case execution and a third for exceptional behavior of the example test:

1.
```
public void testConstructor() throws InvalidDateException {
  CompositeDate subject = new CompositeDate((Integer) getObject("day"),
    (Integer) getObject("month"), (Integer) getObject("year"));
  addObjectToAssert("expectedDay", subject.getDay());
  addObjectToAssert("expectedMonth", subject.getMonth());
  addObjectToAssert("expectedYear", subject.getYear());
}
```

with xml resource:

```
<test id="test20060101">
<objs>
  <obj id="day" type="int">1</obj>
  <obj id="month" type="int">1</obj>
  <obj id="year" type="int">2006</obj>
</objs>
<asserts>
  <assert id="expectedDay" type="int" action="isEqual">1</assert>
  <assert id="expectedMonth" type="int" action="isEqual">1</assert>
  <assert id="expectedYear" type="int" action="isEqual">2006</assert>
```

```
</asserts>
</test>
```

2. or (if `CompositeDate.equals()` is defined appropriate)

```
public void testConstructor() throws InvalidDateException {
  CompositeDate subject = new CompositeDate((Integer) getObject("day"),
    (Integer) getObject("month"), (Integer) getObject("year"));
  addObjectToAssert("expectedCompositeDate", subject);
}
```

with xml resource

```
<test id="test20000229">
<objs>
  <obj id="day" type="int">29</obj>
  <obj id="month" type="int">2</obj>
  <obj id="year" type="int">2000</obj>
</objs>
<asserts>
<assert id="expectedCompositeDate"
  type="com.foo.model.calendar.CompositeDate" action="isEqual"
  hint="call">
  <day>29</day>
  <month>2</month>
  <year>2000</year>
</assert>
</asserts>
</test>
```

3. And last but not least the exceptional case:

```
public void testConstructorException() throws InvalidDateException {
  CompositeDate subject = new CompositeDate((Integer) getObject("day"),
    (Integer) getObject("month"), (Integer) getObject("year"));
}
```

with xml resource:

```
<test id="test20060431">
<objs>
  <obj id="day" type="int">31</obj>
  <obj id="month" type="int">4</obj>
  <obj id="year" type="int">2006</obj>
</objs>
<asserts>
  <exception id="expected"
type="com.foo.model.calendar.InvalidDateException"
    action="isInstanceOf" />
</asserts>
</test>
```

So there is no try/catch block neccessary for expected exception behavior. If an expected exception is not caught during execution of test method an appropriate error is thrown (second example for method testConstructorException).
Under the links you will find the Java source and the xml resource.

[top]

Problems

What can I do to make this and that to work?

I can fix it right away, if you go to the Issue Tracking page add a new bug/feature request/support request, and most important, ADD A CODE EXAMPLE! If you do that, i will do my best to sort the problem out as quick as possible. [top]

3.3 Tool Tips

Using Eclipse for Editing

A lot has been written about the use of xml as the new hype in structured data definition and the same amount against xml as not human readable and useless for this purpose.
During the same period of time a lot of development took place as well to support easier creation and maintenance of xml resoures. -- And a lot of people who found it a real nono to use xml for testdata are now big fans of the String framework which does a really big deal on xml configuration stuff ;-)

One way to make your live on creating and maintaining xml resources a little bit easier if you are using Eclipse is based on the xml features introduced by the Web Tools Plattform of Eclipse. and the xml editor extension Eclipse XSLT from Orangevolt.

These components combined provide a feature rich xml editor environment a few years back only expensive commercial suites provided.

After installation (please refere to the appropriate installation details on the provided links) first thing to do is to create a valid **xml catalog** entry of the <DDTUnit> xml schema file.
To do so

1. Ppen Window -> Preferences... -> Web and XML -> XML Catalog
2. Add a new entry under User Entries

   URL: file://<path of local copy of <DDTUnit> xsd resource

   Key Type: Schema Location

   Key: http://ddtunit.sourceforge.net/ddtunit.xsd

Now you can start to create a new <DDTUnit> xml resource by creating a new xml file by using xml schema definition:

1. Open File -> New -> Other... -> XML -> XML (Next)
2. Create XML file from a XML schema file
3. Specify file name
4. Select xml schema file from catalog: key=http://ddtunit.sourceforge.net/ddtunit.xsd
5. Select details level for element creation and the root element

This finishes the creation process. And as example you find a generated xml resource with all optional elements and attributes created with this editor.

```
<?xml version="1.0" encoding="UTF-8"?>
&lt;ddtunit xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation="http://ddtunit.sourceforge.net/ddtunit.xsd"&gt;
 <resources>
  <obj calltype="" hint="fields" id="" keytype="string" method="constructor"
   refid="" type="" valuetype="string">
   <description>description</description>
  </obj>
 </resources>
 <cluster id="">
```

```
  <description>description</description>
  <objs>
   <obj calltype="" hint="fields" id="" keytype="string" method="constructor"
    refid="" type="" valuetype="string">
    <description>description</description>
   </obj>
  </objs>
  <group id="">
   <test id="">
    <objs>
     <obj calltype="" hint="fields" id="" keytype="string" method="constructor"
      refid="" type="" valuetype="string">
      <description>description</description>
     </obj>
    </objs>
    <asserts>
     <assert action="isTrue" hint="fields" id="" keytype="string"
      method="constructor" refid="" type="" valuetype="string">
      <description>description</description>
     </assert>
     <exception action="isEqual" hint="fields" id="" type="">
      <description>description</description>
     </exception>
    </asserts>
   </test>
  </group>
 </cluster>
</ddtunit>
```

The last appetizer is the use of element and attribute completion as defined in the xml schema. So if e.g. selecting the xml element `<assert >` you'll get a list of all possible attributes. - Select `action` and it will complete to

```
<assert action="isEqual">
```

If you select `isEqual` and hit code completion (Ctrl-Space in standard installation) the list of possible attribute values is provided as selectable drop down listbox.

You are not allowed to stop thinking but this is hell of a lot more convenient than writing all by hand.

3.4 Installation Requirements

Requirements

As a basic prerequisite a JAXP 1.1 conformant parser is required - no schema validation.
If xerces 2.4.0 or higher is provided the schema validation support is active. Because this parser is found in many projects it is not provided as part of this project to minimize download resources. Personally I placed it directly in the %java; runtime environment as external library.

The actual list of required archives can be found under Project Dependencies as generated by Maven. Here you see the list for convenience as of %ddtunit; version 0.8.3.

Artifact ID	Type	Version
junit-addons	jar	1.4
junit	jar	3.8.1
junitperf	jar	1.9.1
log4j	jar	1.2.7
xercesImpl	jar	2.6.2

Configuration

The framework contains a set of switches and predefined configuration details. These can be found in the resource /junitx/ddtunit/ddtunitConfig.properties. To change these configurations extract the properties file from jar and provvide it in classpath in front of ddtunit.jar.

Here follows the content as provided by <DDTUnit> 0.8.5

```
# Configuration file of DDTUnit testing framework
# containing all default configuration information
#
# internal test monitor is a DDTTestListener implementation that will
# display test results by using the Apache log4j logger class
# junitx.ddtunit.DDTRunMonitor to be configured as every log4j logger
# Values: true (default) / false
activeRunMonitor = true
#  resource path of log4j configuration file
#  Default: /junitx.ddtunit.log4j.properties
log4jConfigResource = /junitx.ddtunit.log4j.properties
#  Activate xml schema validation
#  Values: true (default) / false
activeXmlValidation = true
#  Activate PARSER validation
#  Values: true / false (default)
activeParserValidation = true
#  Activate assert validation
#  Values: true (default) / false
activeAsserts = true
#  Define Locale to use during tst execution
```

```
# Values: <language>_<country>
activeLocale = en_US
#  Define date / time format
#  You can add new formats by using the naming convention
#  date.<myFormatName>=<format characters as used by DateFormat class of JDK>
#
date.short=dd.MM.yyyy
date.medium=dd.MM.yyyy HH:mm:ss
date.long=dd.MM.yyyy HH:mm:ss.SSS
date.example=EEE MMM dd HH:mm:ss zzz yyyy
#
```

3.5 XML Schema Definition

3.6 Ant

Apache Ant Build Support

The build process will be based on Ant in future.
The first implementation is tested with Ant 1.5, 1.5.3 and 1.6.1 included in Eclipse 3.0. Additionally Ant optional.jar is required.

For now a lot of tags are missing which are provided by maven.

3.7 Maven

Apache Maven Build Support

The build via Maven (tested with Maven-1.0) is in experimental state for the moment.
But for now it is the major facility to do a build at all.
It is used to generate the project including documentation and statistics via `maven site:generate`.
Next steps are to provide a fully functional deployment process to Sourceforge by using `maven sourceforge:deploy-dist`.

A few things are missing for now like the test classes and the Ant build script.

3.8 Download

http://sourceforge.net/project/showfiles.php?group_id=98719

3.9 Browse CVS

http://cvs.sourceforge.net/viewcvs.py/ddtunit/