



# CASTOR TESTING FRAMEWORK USER DOCUMENT

Authors:

Arnaud Blandin [[blandin@intalio.com](mailto:blandin@intalio.com)]

Sebastien Gignoux [[gignoux@kernelcenter.org](mailto:gignoux@kernelcenter.org)]

Revision: March 21, 2002

Abstract	3
1. Introduction .....	4
2. Overview of CTF.....	6
2.1 Creating a JUnit test case .....	6
2.2 Repository architecture .....	7
2.3 Test patterns .....	7
3 Writing Tests or reporting bugs using the CTF .....	11
3.1 The Jar file .....	11
3.2 The directory .....	12
3.3 TestDescriptor.xml .....	12
3.5 ObjectBuilder .....	22
3.6 Reporting a bug.....	23
4 Example.....	24
4.1 TestDescriptor.xml .....	24
4.2 Creating PrimitivesBuilder.java .....	26
4.3 Launching the tests .....	26
5 Status of CTF .....	28
6 References .....	29
7 Appendix A : XML Schema for TestDescriptor .....	30
8 Appendix B: the 'gold' file .....	35
9 Appendix C: Check-list to write a test .....	36

## Abstract

Test Automation is often seen as very useful but also as a tedious task.

However the limited number of 'test templates' for Castor makes it easy to implement.

The aim of this document is to describe the Castor Testing Framework by giving an overview of its functionalities as well as providing a detailed example.

# 1. Introduction

One of the main processes in the life cycle of software is the 'validation and verification' process. It is a control process which checks that the requisites functions are finally obtained and eliminates faults. It is also known as the non-popular 'testing' process.

The reliability and the integrity of a software project are based on tests. Being 'open-source' does not mean avoiding writing tests.

However, software testing consumes in general 30 to 70 percent of software development resources, does not find all the faults and can be as tedious as writing the code itself.

Moreover when involved in an 'open-source' project which includes several developers and contributors, a developer needs to write tests that can be understood by the other developers, that highlight a specific feature and without spending too much time on it.

One solution is to choose to automated tests: as described in 'Seven Steps to Automation Steps' by Bret Pettichord ([1]) there are several reasons to choose automate tests:

- Allow testing to happen more frequently
- Improve test coverage
- Ensure consistency
- Improve the reliability of testing

For all these reasons, we have chosen to give to Castor an Automation Testing Framework called the Castor Testing Framework (referenced as CTF in this document).

This framework is built with JUnit (2)) and will help Castor developers but also users in several ways:

- Improve Castor reliability
- Monitoring bugs will be easy since writing bug reports will be easy
- Building an archive of fixed bugs
- Test writing is easier.

## 2. Overview of CTF

In this section you will be given a quick overview of CTF: what is the main idea behind it? Where are the tests located? What are the test patterns? How to run the tests?

Note: currently CTF is written to test Castor XML; a future could contain the entire tests of Castor.

### 2.1 Creating a JUnit test case

Built on top of JUnit ([2]), CTF simply creates test cases to be run under the JUnit Framework.

The main idea is to provide to the CTF the minimal information needed to build test cases (such as XML Schemas, Castor mapping files, XML documents or some initialization codes for the java object model used) and let it building as much tests as possible.

This necessary information is located in a jar or in a directory structure (see section 3 for more details).

A directory will contain all the directories and jars and CTF will create a global test case with these individual test cases.

CTF will build dynamically the test cases located in the different jar files and pass them to JUnit.

## 2.2 Repository architecture

The code of Castor Testing Framework is located in the following packages:

```
org.exolab.castor.tests.framework
```

```
org.exolab.castor.tests.framework.testDescriptor
```

The tests for testing the behavior of Castor and its main functionalities are located under `src/tests/MasterTestSuite` (and its subdirectories).

The tests needed to keep tracks of reported and hopefully fixed bugs are located under: `src/tests/RegressionTestSuite`.

The hierarchy of these directories follows the main modules of Castor:

(MasterTestSuite | RegressionTestSuite) / xml / Mapping

/ Introspection

/ SourceGenerator

## 2.3 Test patterns

“Testing Castor XML” means, not only testing the Marshalling Framework but also the mapping framework as well as the Source Generator.

Thus tests fall into one of these 3 categories and follow several patterns defined in this section.

From now on, we will use the term ‘gold file’ to designate a file that represents the expected result.

## Marshalling Framework testing

2 main patterns are used to test the marshalling framework.

### Marshal/Unmarshal process

- Instantiate an object model with random data or hardcoded data if any.
- Marshal it
- Compare the marshalled object to a 'gold' file if any
- Unmarshal the document obtained from the marshalling process
- Compare the unmarshal object to the original object model

### Unmarshal/Marshal process

- Unmarshal a provided XML document (input document)
- Compare the content of the unmarshalled object with hardcoded data
- Marshal the object model obtained
- Compare the obtained document with the input document

## Mapping

Testing the mapping functionalities of Castor is simply testing the Marshalling framework by using a mapping file so the patterns are exactly the same as the ones for the Marshalling Framework.



## Source Generator

Testing the Source Generator requires testing the validity of an XML Schema, checking the content of the generated files and using these files to marshal/unmarshal documents

This can be summarized as follow:

- Generate classes from a given XML Schema
- Initializes the created Object Model (randomly or by using hardcoded data)
- Marshal the Object Model
- Compare it with a gold file if any
- Unmarshals the obtained document
- Compares the two object models
- Unmarshals a given input file

Note: CTF will automatically generate an Object Model from the given XML schema and will compile it in order to use it in the marshalling framework.

## 2.4 Running the test

You can run the tests by using the script CTFRun.bat under Windows platform or CTFRun.sh under Linux/UNIX platform.

CTFRun path of a Castor Testing Framework test (JAR or Directory).

You can set a number of different options:

Option	Args	Description	Status
-verbose	<i>N/A</i>	Give detailed information on the execution of each test.	Optional
-text	<i>N/A</i>	Run the tests without using the JUnit GUI	Optional
-seed	<i>Value</i>	Specifies the use of a specific seed for the pseudo-random generator	Optional

For instance to run the tests for the Source Generator:

```
CTFRun -verbose pathofMasterTestSuite/xml/SourceGenerator
```

This command will execute in GUI mode (default mode) all the test cases written for the Source Generator and will print detailed messages about the execution of the tests.

## 3 Writing Tests or reporting bugs using the CTF

Writing test consist in providing a jar file or a directory with all the necessary information.

This section will detail the necessary steps to write an efficient test case.

You will find in the appendix section a checklist, which summarize all the important points.

### 3.1 The Jar file

The jar file must follow this architecture:

```
./META-INF/TestDescriptor.xml  
./All necessary files
```

For instance the following represents a CTF jar file:

```
MappingTesting/META-INF/TestDescriptor.xml  
  
    /Data.java  
    /Root.java  
    /ObjectBuilder.java  
    /mapping.xml  
    /xmlexample.xml
```

## 3.2 The directory

When using a directory there is no specific architecture to follow, all the necessary files must be put under the directory.

For instance the following is a valid CTF test:

```
MappingTesting/TestDescriptor.xml  
  
    /Data.java  
  
    /Root.java  
  
    /ObjectBuilder.java  
  
    /mapping.xml  
  
    /xmlexample.xml
```

As you may have noticed, one file must be present in both cases: TestDescriptor.xml.

## 3.3 TestDescriptor.xml

This file is located under the META-INF directory when using a jar file or directly under the directory that contains the test case files. This is a summary of the test cases contained in the jar file or in the directory.

It can be divided as follows:

Note: the whole XML Schema used to write TestDescriptor.xml is provided in the appendix.

## Header

The header of the xml file contains all the generic information about a test:

Tag	Content	Status
Name	The name of the test	Required
Comment	Some comment on the test	Required
Category	The category in which this test falls: - basic capability - special case	Required
BugFix	Used to report a bug and keep track of it It is defined as follows: <ul style="list-style-type: none"> <li>• Name of the reporter</li> <li>• Date of the report</li> <li>• Name of the 'fixer'</li> <li>• Date of the fix</li> </ul>	Optional

For instance,

```
<?xml version='1.0'?>
<TestDescriptor>
  <Name>Bug in the support of foo</Name>
  <Comment>This test case illustrates the bad handling of the foo element in the
Marshalling Framework</Comment>
  <Category>basic capability</Category>
  <BugFix>
    <Reporter>Desperate User</Reporter>
    <Date_Report>2001-03-11T12:00:00</Date_Report>
    <Fixer>Comprehensive Contributor</Fixer>
    <Date_Fix>2053-12-12T15:03:00</Date_Fix>
  </BugFix>
...

```

```
</TestDescriptor>
```

Is a valid header for a TestDescriptor file.

You can wonder why CTF needs so much information.

Actually, this information can be used to generate acceptance documents or to archive properly the different bugs.

After the definition of the header comes the main part: the definition of the test.

This can be either a SourceGeneratorTest or a MarshallingTest.

## MarshallingTest

The MarshallingTest element may contain the following tags:

Tag	Attribute	Content	Status
Root_Object	<ul style="list-style-type: none"> <li><i>dump</i> : a boolean indicating that dumpFields() method has been implemented in Root_Object.</li> <li><i>random</i> : a boolean indicating that randomizeFields() method has been implemented in Root_Object.</li> </ul>	The qualified name of the Root object in the generated object model	Required
Mapping_File	N/A	The name of the mapping file to (if any)	Optional

Then you can define several UnitTestCase elements (see section 3.2.4).

## SourceGeneratorTest

The SourceGeneratorTest element may contain the following elements:

Tag	Attribute	Content	Status
Schema	N/A	The name of the schema form which we generate sources	Required
Property_File	N/A	The name of the Source Generator property file to use	Optional
FieldInfoFactory	N/A	The collection type to use (Vector or ArrayList)	Optional
Root_Object	<ul style="list-style-type: none"> <li><i>dump</i>: a boolean that indicates if the result are to be dumped into specific files</li> <li><i>random</i>: a boolean that indicates if the object model should be randomly initialized</li> </ul>	The qualified name of the Root object in the generated object model	Required

Then you can define several UnitTestCase elements (see next section).

Note: for more details on 'dump' and 'random', please refer to the next section.

## UnitTestCase

This is the core of the test, this element provides the input file and output file used for a specific test and the name of the class used to instantiate the Object Model.

Once again remember that a TestDescriptor.xml describes a test for a general behavior of Castor which can imply several test cases.

Tag	Content	Status
Name	The name one specific test case	Required
Input	The name of the input file used to create a specific Object Model while unmarshalling this file	Optional
Output	The name of the output file which can be seen as a 'gold file'	Optional
ObjectBuilder	The name of the class used to instantiate the Object Model used.	Optional
Failure	A boolean that indicates if set to true that the test case intents to fail (Exception thrown)	Optional



## Example file

Here is a complete TestDescriptor.xml

```
<?xml version='1.0'?>
<TestDescriptor>
    <Name>Bug in the support of foo</Name>
    <Comment>This test case illustrates the bad handling of the foo element
in the Marshalling Framework</Comment>
    <Category>basic capability</Category>
    <BugFix>
        <Reporter>Angry User</Reporter>
        <Date_Report>2001-03-11T12:00:00</Date_Report>
        <Fixer>Comprehensive Contributor</Fixer>
        <Date_Fix>2053-12-12T15:03:00</Date_Fix>
    </BugFix>
    <MarshallingTest>
        <Root_Object random="true" dump="true">Root</Root_Object>
        <Mapping_File>mapping.xml</Mapping_File>
        <UnitTestCase>
            <Name>Test foo as an attribute</Name>
            <Input>input1.xml</Input>
            <ObjectBuilder>input1_Builder</ObjectBuilder>
            <Output>input1.xml</Output>
        </UnitTestCase>
        <UnitTestCase>
            <Name>Test foo as an element</Name>
            <Input>input2.xml</Input>
            <Failure>true</Failure>
        </UnitTestCase>
    </MarshallingTest>
</TestDescriptor>
```

## 3.4 Implementing CastorTestable

As you may have noticed, while testing the SourceGenerator or the Marshalling Framework you have to provide the name of the 'Root Object' of your Object Model.

The Root Object simply represents the mapping of the root element of the XML document.

For instance given the following XML document:

```
<?xml version='1.0'?>
<Invoice>
  <Customer>
    ...
  </Customer>
  <Items>
    ...
  </Items>
  ...
</Invoice>
```

The class representing the Root Object will be Invoice.java.

In order to be correctly used in CTF, the provided Root Object must implement the interface `org.exolab.castor.tests.framework.CastorTestable`.

Implementing this interface is simply implementing two methods: `dumpFields()` and `randomizeFields()`.

Here is the javadoc of these methods:

```
/**
 * Return a recursive dump of the content of the
 * object fields in a user readable format.
 * This is used to retrieve the state of the object if
 * castor fail to marshal the object for any reason.
 *
 * We don't rely on the toString() function as it could have
 * been already implemented with another semantic.
```

```

*/

public String dumpFields();

/**
 * The instance of the object will randomize the content
 * of its field. This is used to create an instance
 * of the object model without having to
 * unmarshal anything.
 */

public void randomizeFields()

    throws InstantiationException, IllegalAccessException ;

```

In order to tell the CTF that one (or both) of these methods is implemented you have to set the attributes 'dump' and 'random' to true in the Root\_Object element:

```
<Root_Object dump='true' random='true'>Root.java</Root_Object>
```

CTF needs also your Root Object to override the equals() method of the java.lang.Object class.

This is required in order to compare properly two Object Models.

The following is an example of a Root Object:

```

import java.util.Vector;
import java.util.Enumeration;

//CTF specific
import org.exolab.castor.tests.framework.CastorTestable;
//Provide the necessary methods to randomly create //object fields
import org.exolab.castor.tests.framework.RandomHelper;

public class Root implements CastorTestable{

    private String _name;

    private Vector _data;

```

```

public void setName(String name) {
    _name = name;
}

public String getName() {
    return _name;
}

public void setData(Vector data) {
    _data = data;
}

public Vector getData() {
    return _data;
}

// --- CastorTestable -----
public String dumpFields() {
    String fields = "name=" + _name + "; data=\n";
    for (Enumeration e=_data.elements(); e.hasMoreElements(); )
        fields += "[" + ((CastorTestable)e.nextElement()).dumpFields() + "]\n";
    return fields;
}

public void randomizeFields()
    throws InstantiationException, IllegalAccessException
{
    _name = RandomHelper.rndString();
    _data = RandomHelper.rndVector(_data, Data.class);
}

/**
 * Note: hashCode() has not been overridden
 * @param obj
 */

```

```

public boolean equals(java.lang.Object obj)
{
    if ( this == obj )
        return true;

    if (obj instanceof Root) {

        Root temp = (Root)obj;
        if (!(this._name.equals(temp._name)))
            return false;
        if (!(this._data.equals(temp._data)))
            return false;
        return true;
    }
    return false;
} //-- boolean equals(java.lang.Object)
} //Root

```

Note: when using the Source Generator you can automatically generate classes that implements CastorTestable by using the command line option “-testable”.

For more details on the Source Generator you can refer to the Source Generator Documentation ([4]).

## 3.5 ObjectBuilder

When you unmarshal an XML document it could happen that you want to compare the resulting Object Model with an expecting Object Model instance.

You can provide to CTF a way to build this expected Object Model instance by giving it a class that implements

`... tests.framework.ObjectBuilder`

This class should provide an implementation of the method `buildInstance()`.

Here is a simple implementation:

```
import java.util.Vector;

import org.exolab.castor.tests.framework.ObjectBuilder;

public class Builder implements ObjectBuilder
{

    /**
     * Build the expected object model while
     * unmarshalling the given input file.
     */
    public Object buildInstance() {
        Root r = new Root();
        r.setName("object name");
        Vector v = new Vector(3);
        v.add(new Data("CASTOR-XML", "good"));
        v.add(new Data("CASTOR-JDO", "good"));
        v.add(new Data("CASTOR-DAX", "to be improved"));
        r.setData(v);
        return r;
    }
} //Builder
```

## 3.6 Reporting a bug

With CTF reporting a bug is easier, you should not find anymore yourself in endless discussion in describing the bug you encounter.

What you just have to do is :

- write a simple test case as describe previously.
- create a jar or a directory
- send it to the exolab bugzilla or to a contributor:

<http://bugzilla.exolab.org/>

<http://www.castor.org/contributors.html>

## 4 Example

In this section we are going to detail the writing of a specific test case for the Source Generator: testing the handling of primitives W3C XML Schema type in the Source Generator using the default properties of the Source Generator.

This test just illustrates that the Source Generator can support correctly primitives types.

First we have to create the TestDescriptor xml file:

### 4.1 TestDescriptor.xml

According to section 3, the test descriptor just describes the test case:

```
<?xml version='1.0'?>
<TestDescriptor>
  <Name>Test primitive types with default properties</Name>
  <Comment>Test the W3C XML Schema primitives types handling in the Source Generator</Comment>
  <Comment>The supported types are:
    -string
    -boolean
    -decimal
    -float
    -double
    -duration
    -dateTime
    -time
    -gYearMonth
    -gYear
    -gMonthDay
    -gDay
    -gMonth
    -hexBinary
    -base64Binary
```



```

        -anyURI
        -QName

</Comment>

<Comment>The facets are not tested in this case and java primitives are used.</Comment>

<Comment>hexBinary and base64Binary are not tested</Comment>

<Category>basic capability</Category>

<SourceGeneratorTest>

    <Schema>primitives.xsd</Schema>

    <Root_Object random="true" dump="true">TestPrimitives</Root_Object>

    <UnitTestCase>

        <Name>Test Generation</Name>

        <Input>input1.xml</Input>

    </UnitTestCase>

    <UnitTestCase>

        <Name>Test Marshalling with the generated Descriptors</Name>

        <Input>input1.xml</Input>

        <ObjectBuilder>PrimitivesBuilder</ObjectBuilder>

    </UnitTestCase>

    <UnitTestCase>

        <Name>Test the validation</Name>

        <Input>badinput.xml</Input>

        <Failure>true</Failure>

    </UnitTestCase>

</SourceGeneratorTest>

</TestDescriptor>

```

Once we have the test descriptor we have to provide as stated the 'primitives.xsd' and the 'input1.xml' files. Let's assume we have already written it (the full content of these files can be found in the appendix).

We don't need to implement CastorTestable since the SourceGenerator is doing it for us: it is automatically generating the dumpFields() and randomizeFields() methods.

We need now to provide an instance builder i.e. implementing the ObjectBuilder.

## 4.2 Creating PrimitivesBuilder.java

PrimitivesBuilder.java is a simple implementation of ObjectBuilder.

It contains the hard-coded values of the input xml file, it is used to create a 'gold' Object Model we can use to compare the Object Model created from the unmarshalling of 'input1.xml'.

```
import java.util.Vector;

//the interface we implement
import org.exolab.castor.tests.framework.ObjectBuilder;

public class PrimitivesBuilder implements ObjectBuilder
{
    /**
     * Build the object expected when unmarshalling 'input1.xml'
     */
    public Object buildInstance() {
        TestPrimitives test = new TestPrimitives();
        test.setStringTestAtt("StringAttribute");
        test.setBooleanTestAtt(false);
        test.setFloatTestAtt(3.141526f);
        test.setDoubleTestAtt(1.171077);
        ...
        return test;
    }
}
```

## 4.3 Launching the tests

The last step consists in creating the jar file:

```
jar cvf ../PrimitivesWithoutFacetsTest.jar *.*
```

and commit it in the corresponding directory (in this case we will commit  
PrimitivesWithoutFacets.jar in  
castor/src/tests/MasterTestSuite/xml/sourcegenerator.

Then you can launch it by using the following:

```
CTFRun -verbose src/tests/MasterTestSuite/xml/SourceGenerator/  
PrimitivesHandlingWithoutFacets.jar
```

For more information on CTFRun, please refer to section 2.4

If you were writing a test case to report a bug what you only have to do is send  
your jar to a Castor contributor.

## 5 Status of CTF

The current version of CTF is far from being perfect and clearly needs some improvements.

However it is a solid base to test the behavior of Castor and avoid regression while fixing bugs.

Future versions of the CTF will include:

- Write a tool to generate test acceptance documents.
- Change the architecture of RandomHelper
- Provide a way to launch the test from the web site
- Better exception handling and failure tests.

## 6 References

1. Seven Steps to Test Automation Success

Version of July 2000

Bret Pettichord

See [http://www.io.com/~wazmo/papers/seven\\_steps.html](http://www.io.com/~wazmo/papers/seven_steps.html)

2. JUnit Framework

See <http://www.junit.org>

3. Castor XML

Castor XML.

Exolab Castor XML Team

See <http://www.castor.org/>

4. Source Generator User Documentation

Castor XML

Exolab Castor XML Team

See <http://www.castor.org/SourceGeneratorUser.pdf>

## 7 Appendix A : XML Schema for TestDescriptor

Here is the whole W3C XML Schema for the TestDescriptor document (described in section 3.2)

```
<?xml version='1.0'?>
<!--
    Castor Testing Framework Test Descriptor XML Schema
    Namespace: http://castor.exolab.org/Test

    This schema is used to generate the
    org.exolab.castor.tests.framework.testdescriptor package
    *Note*: This schema is under evolution and subject to change.
        This schema is under the Exolab license
-->
<!-- $Id: TestDescriptor.xsd,v 1.6 2002/03/12 16:20:42 blandin Exp $ -->
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://castor.exolab.org/Test"
            elementFormDefault="qualified">
<!-- The root element which contains an header and a test element-->
    <xsd:element name="TestDescriptor">
        <xsd:complexType>
            <xsd:sequence>
                <!-- The name of the test -->
                <xsd:element name="Name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
                <!-- Some comments for describing the test -->
                <xsd:element name="Comment" type="xsd:string" minOccurs="1" maxOccurs="unbounded"/>
                <!-- Define the category of the test --
                <xsd:element name="Category" type="CatgoryType" minOccurs="1" maxOccurs="1"/>
                <!-- Is it a bug fix?--
                <xsd:element ref="BugFix" minOccurs="0" maxOccurs="1"/>
            </xsd:sequence>
        </xsd:complexType>
    </xsd:element>
</-->
</xsd:schema>
```

```
<!-- Test for the SourceGenerator OR the Marshalling Framework-->
<xsd:choice>
    <!-- Test case for the SourceGenerator-->
    <xsd:element ref="SourceGeneratorTest" minOccurs="0" maxOccurs="1"/>
    <!-- Test case for the Marshalling Framework-->
    <xsd:element ref="MarshallingTest" minOccurs="0" maxOccurs="1"/>
</xsd:choice>
</xsd:sequence>
</xsd:complexType>
</xsd:element>

<xsd:element name="BugFix">
    <xsd:complexType>
        <xsd:sequence>
            <!-- the reporter name -->
            <xsd:element name="Reporter" type="xsd:string" minOccurs="1" maxOccurs="1"/>
            <!-- date of the report-->
            <xsd:element name="Date_Report" type="xsd:date" minOccurs="1" maxOccurs="1"/>
            <!-- the 'fixer' name -->
            <xsd:element name="Fixer" type="xsd:string" minOccurs="1" maxOccurs="1"/>
            <!-- date of the fix-->
            <xsd:element name="Date_Fix" type="xsd:date" minOccurs="1" maxOccurs="1"/>
            <!-- Some comments on the fix or the bug -->
            <xsd:element name="Comment" type="xsd:string" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="SourceGeneratorTest">
    <xsd:complexType>
        <xsd:sequence>
            <!-- the name of the schema file -->
```

```
<!-- assume that the Testing Framework will try to match the name -->
<!-- by looking in all the JARs - directories -->
<xsd:element name="Schema" type="xsd:string" minOccurs="1" maxOccurs="1"/>
<!-- The name of the properties file used with this SourceGenerator test case-->
<xsd:element name="Property_File" type="xsd:string" minOccurs="0" maxOccurs="1"/>
<!-- The name of the collection type used with this SourceGenerator test case-->
<xsd:element name="FieldInfoFactory" type="xsd:string" minOccurs="0" maxOccurs="1"/>
<!-- the qualified name of the root Object -->
<!-- later: define a pattern to describe a Java qualified name-->
<xsd:element name="Root_Object" type="RootType" minOccurs="1" maxOccurs="1"/>
<!-- the test cases for the SourceGenerator-->
<!-- for the moment it is shared with the test cases-->
<xsd:element ref="UnitTestCase" minOccurs="1" maxOccurs="unbounded"/>

</xsd:sequence>

</xsd:complexType>

</xsd:element>

<xsd:element name="MarshallingTest">
  <xsd:complexType>
    <xsd:sequence>
      <!-- the qualified name of the root Object -->
      <!-- later: define a pattern to describe a Java qualified name-->
      <xsd:element name="Root_Object" type="RootType" minOccurs="1" maxOccurs="1"/>
      <!-- the mapping file used (if any) -->
      <xsd:element name="Mapping_File" type="xsd:string" minOccurs="0" maxOccurs="1"/>
      <!-- the test cases for the Marshalling Framework-->
      <!-- for the moment it is shared with the test cases-->
      <!-- for the SourceGenerator-->
      <xsd:element ref="UnitTestCase" minOccurs="1" maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```



```
<xsd:element name="UnitTestCase">
  <xsd:complexType>
    <xsd:sequence>
      <!-- The name of the test -->
      <xsd:element name="Name" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      <!-- the input XML file for unmarshalling-->
      <xsd:element name="Input" type="xsd:string" minOccurs="0" maxOccurs="1"/>
      <!-- the output file for marshalling -->
      <xsd:element name="Output" type="xsd:string" minOccurs="0" maxOccurs="1"/>
      <!-- the ObjectBuilder class used for this test case -->
      <xsd:element name="ObjectBuilder" type="xsd:string" minOccurs="0" maxOccurs="1"/>
      <!-- a boolean that indicates if the test case intends to fail (Exception thrown)-->
      <xsd:element name="Failure" type="xsd:boolean" default="false" minOccurs="0"
        maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:simpleType name="CategoryType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="basic capability"/>
    <xsd:enumeration value="special case"/>
    <!-- Extensible-->
  </xsd:restriction>
</xsd:simpleType>
```

```
<!--A root object in an object model-->
<xsd:complexType name="RootType">
  <xsd:complexContent>
    <xsd:extension base="StringType">
      <!--set to true to generate randomly the given Object Model-->
      <xsd:attribute name="random" type="xsd:boolean" default="false"/>
      <!--set to true to dump the given Object Model states in specific files-->
      <xsd:attribute name="dump" type="xsd:boolean" default="false"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="StringType">
  <xsd:simpleContent>
    <xsd:extension base="xsd:string"/>
  </xsd:simpleContent>
</xsd:complexType>

</xsd:schema>
```

## 8 Appendix B: the 'gold' file

Here is the 'gold' file used in our test case example:

```
<?xml version="1.0"?>
<TestPrimitives StringTestAtt="StringAttribute" booleanTestAtt="false"
    floatTestAtt="3.141526" doubleTestAtt="1.171077"
    decimalTestAtt="123456789.987654321"
    uriReferenceTestAtt="http://www.castor.org"
    IDTestAtt="Castor0.91" QNameTestAtt="xsd:type">
  <StringTestEle>StringElement</StringTestEle>
  <booleanTestEle>true</booleanTestEle>
  <floatTestEle>1234567899876543210</floatTestEle>
  <doubleTestEle>0.6385682166079459</doubleTestEle>
  <decimalTestEle>0.2693678757526658529286578414030373096466064453125</decimalTestEle>
  <timeDurationTestEle>P1Y2M3DT4H5M6S</timeDurationTestEle>
  <uriReferenceTestEle>http://castor.exolab.org</uriReferenceTestEle>
  <IDTestEle>ID9</IDTestEle>
  <QNameTestEle>Test:test</QNameTestEle>
</TestPrimitives>
```

## 9 Appendix C: Check-list to write a test

- Create a directory (the test directory) to put the files needed for the test case.
- Create the META-INF directory in this directory.
- Create your TestDescriptor.xml.
- Write the XML schema, mapping file or XML documents needed and put it under your test directory.
- If you provide an Object Model don't forget to override the equals() method
- Implement if necessary the dumpFields() and randomizeFields() method.
- Compile the provided object model
- Create a jar representing the 'test directory'
- Run the test