SoftConEx
Software Consulting Experts

SoftConEx Assessment

Berlin, 27. September 2016

Holger Haag
holger.haag@softconex.de
+49 (0) 30 8145638-12

SoftConEx GmbH
Charlottenstr. 81
10961 Berlin

http://www.SoftConEx.de

Table Of Contents

1 Introduction

This document describes the SoftConEx assessment project. The assessment project is used to ensure consistency in the recruiting process and to aid both sides – the future employee and SoftConEx – assessing each other. The assessment is not a one-way street, it is supposed to help both sides in understanding whether the applicants requirements match SoftConExs' requirements and vice versa.

1.1 Important note

Software development is inherently difficult, especially when – which is the case here – somebody enters a project that has already been going on for a while. It is virtually impossible to know everything first place. **Do not hesitate to ask any questions**. At SoftConEx we cultivate an open style of discussing problems and questions. Do not waste your time (and our money) when you've come to a dead end at your current task (be it assessment or „real“ development) just because you don't want to ask too many questions.

2 Revisions

Date	*Update*
10.04.07	Document created
12.04.07	Download section
27.04.09	FTP Credentials fixed
05.09.11	FTP → FTP-Secure
07.08.13	Remove FTP section, some adjustments
09.08.13	ANT usage of a different version
18.09.15	Removed PMD/checkstyle
30.09.15	Removed reference to ant, firstPerCent clarified

3 The assessment project

3.1 Installation

First install the Java SE Development Kit (JDK), Version 8. It can be downloaded from http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Next install the Eclipse IDE (latest version) from http://www.eclipse.org/downloads/.

Import the assessment.zip as project into your workspace.

3.2 Java 5++

The following Java 5 constructs are used wherever possible:

- Varargs (http://docs.oracle.com/javase/1.5.0/docs/guide/language/varargs.html)
- For-Each loop (http://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html)
- Custom class for list:

```
public class SomeObjectList extends ArrayList<SomeObject> {
}

SomeObjectList list = new SomeObjectList();

instead of

SomeObjectList list = new ArrayList<SomeObject>();

This allows to add methods to the SomeObjectList class which will be
readily available througout the code without any other changes.
```

3.3 The tools used

3.3.1 Commons IO

Jakarta Commons IO (http://jakarta.apache.org/commons/io/), version 1.3.1 is part of the projects' classpath to make some I/O related tasks easier.

3.3.2 Commons Logging

Although the „regular“ SoftConEx projects use a different approach to logging, the assessment project uses the Jakarta Commons Logging API (http://jakarta.apache.org/commons/logging/).

The simplest usage form is really very simple. First instantiate a LOG object. This is done typically as a final static variable which can be used throughout the class:

```
private final static Log LOG = LogFactory.getLog(SimpleTest.class);
```

Logging a message in the assessment project is always done on the so-called INFO level:

```
LOG.info("Caught expected exception " + ex.getMessage());
```

Logging an exception is done on the ERROR level:

```
LOG.error("Caught expected exception " + ex.getMessage(), ex);
```

Important notes on logging:

- Never use System.out.print to log messages on the console, but LOG.info
- Never use ex.printStackTrace() to log an exception, but LOG.error

3.3.3 A few notes on catching and logging exceptions

Never catch exceptions silently:

```
try {
      final String testString = "Hello World";
      assertEquals("Hello", testString.substring(0, 5));
} catch (final Exception ex) {
}
```

This is a good way to have your head ripped off when something goes wrong in production and a colleage spends hours to find out that an exception is being ignored unless you have a very good reason to code this way. ALWAYS log exceptions, at least a message on DEBUG level.

But even this makes sense only in a very few cases, typically exceptions will be rethrown in some way.

There are three accepted ways of handling an exception:

- In most cases the exception will be logged on ERROR level. Depending on the use case, a new exception might be thrown.
- Logging an exception on ERROR level will print the stack trace to the application log. This can blow up the log extensively and it might be advisable not to use the ERROR level. In that case consider logging just the exception message (not the stack trace) on WARN level.
- In rare cases the exception must be caught and ignored „silently“. A typical case is a NumberFormatException which can be thrown e.g. if an invalid number is passed to the Integer(String) constructor and such an exception represents an invalid user input. In that case log the exception message on DEBUG level so that in production systems the administrator can see what is going on by increasing the log level. Also make sure that you get approval for doing so.

Why bother about exceptions in this assessment document ? Because we at SoftConEx have wasted too much productivity in the last years on that issue and are very very picky about it.

3.3.4 Junit

Junit (http://www.junit.org) is used for unit testing. A junit plug-in is bundled with the standard eclipse distribution. All classes which extend `junit.framework.TestCase` can be executed as a unit test case through the Eclipse Menu (Run / Run As / Junit)..

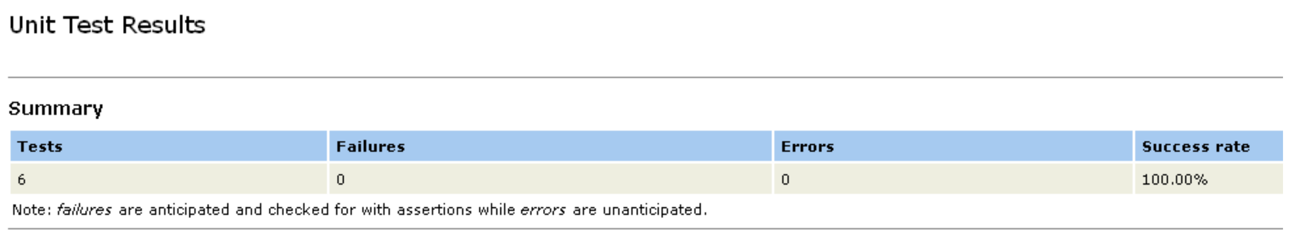
A simple test case might look like this:

```
public class SimpleTest extends TestCase {
        public final static Log LOG = LogFactory.getLog(SimpleTest.class);

        public void testEquals() {
                final String testString = "Hello World";

                assertEquals("Hello", testString.substring(0, 5));
        }

        public void testForException() {
                final String testString = "Hello World";

                try {
                        assertEquals("Hello", testString.substring(100));
                        fail();
                } catch (final Exception ex) {
                        LOG.info("Caught expected exception " + ex.getMessage());
                }
        }
}
```

All public methods which start with „test" are considered test methods and will be executed. The junit window in eclipse will show the results:

As shown in the example above, there are two typical ways of structuring a test method. The first case (testEquals()) instantiates objects and makes assertions on method outputs. The second case (testForException()) ensures that an exception is thrown.

The default Ant build target also runs all unit tests in the assessment project. The output is stored in the file /temp/junit/junit-noframes.html:



Unit Test Results

Summary

Tests	Failures	Errors	Success rate
6	0	0	100.00%

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

3.3.5 Dom4j

According to its own web site (http://www.dom4j.org/), dom4j *"is an easy to use, open source library for working with XML, XPath and XSLT on the Java platform using the Java Collections Framework and with full support for DOM, SAX and JAXP".*

In simpler words, it can be used for reading and writing XML documents but is much nicer to use than the XML packages in the standard JDK.

A quick guide with all necessary information needed for the assessment project can be found in the „Quick Guide“ (http://www.dom4j.org/guide.html). The current version (1.6.1) is already contained in the project's library path.

Some helper methods which are useful e.g. in test cases can be found in the `de.softconex.assessment.XmlUtils` class.

3.3.6 XSD Validator

To validate generated XML documents XSD schemas (http://de.wikipedia.org/wiki/XSD) are used. A helper method in `de.softconex.assessment.XmlUtils#validate` exists for easy XSD validation of a dom4j document. Sample usage is documented in the `de.softconex.assessment.XmlUtilsTest` test case.

3.4

3.5 Assessment

3.5.1 Calculation Model

A calculation model calculates a markup for a given price, **e.g.**

- If price is between 0 and 99€, a markup of 10€ will be applied
- If price is between 100 and 199€, a markup of 15€ will be applied
- If price is higher than 199€, a markup of 8% will be applied

The implemented classes shall provide all necessary attributes and methods to perform this **kind of logic**.

The following classes will be created:

- **Price**: A simple price object (without currency)
- **PriceList**: A list of price objects
- **PriceRange**: A price range (price from/to)
- **CalculationModelDetail**: A calculation model detail which defines the markup for a given price range
- **CalculationModelDetailList**: A list of calculation model details
- **CalculationModel**: A calculation model holds an instance of CalculationModelDetailList

The classes will have methods to convert objects to XML and to parse XML structures. Testsfor all methods will be provided.

3.5.2 Assessment Task

- Implement CalculationModelDetail#calculate completely (the TODOs are described inside the method body itself) and add the necessary test methods.
- Implement the toXml() and parse() methods in CalculationModelDetail, CalculationModelDetailList and CalculationModel
- Implement the equals() method in CalculationModelDetail and add the necessary test methods to CalculationModelDetailTest. A sample implementation of a non-trivial equals() method can be found in the PriceRange class.
- Implement a new method sortByMinimumAscending() in CalculationModelDetailList. The sort logic will be similar to the sorting done in PriceList (already implemented) with one exception – a detail with no priceRange (getPriceRange() == null) should always be the last element of a sorted list.
- Implement CalculationModel class completely

Note on firstPerCent:

This defines whether percentage is applied first or absolute markup, e.g.

firstPerCent=true: (100 + 10%) + 10 = 120

firstPerCent=false: (100 + 10) + 10% = 121