Unit Tests mit TestNG oder JUnit und Spring Unterstützung

Unit Tests mit TestNG oder JUnit und Spring Unterstützung

Nuclos unterstützt seit einiger Zeit das Schreiben von Unit Tests. Die Testunterstützung entwickelt sich jedoch nur langsam weiter, und wird (hauptsächlich) im FDM Projekt verwendet. Dort finden sich also auch die besten Beispiele, allerdings sind diese nur für Nuclos Mitarbeiter zugänglich. Diese Seite gibt (auch allen Anderen) einen Überblick, was möglich ist.

Grundsätzliches zu Unit Tests

Bei Unit Tests im engeren Sinne geht es darum, eine einzelne Methode/Klasse *unabhängig* vom Rest des Quellcodes zu testen. Dies ist natürlich nur dann interessant, wenn die Methode/Klasse irgend etwas **tut** (also kein reines *Value Object* ist, wie etwa eine einfache *Java Bean*). Normalerweise möchte man Berechnungen testen, es ist jedoch auch möglich, Methoden/Klassen zu testen, die hauptsächlich IO machen.

Ein *Unit Test Framework* liefert eine grundlegende Basis, die es ermöglicht, Tests auszuführen, die Testergebnisse festzustellen, zu bewerten und dem Tester anzuzeigen. Insbesondere ermöglicht es es, im Rahmen des Build Prozesses und der CI, alle Tests automatisch auszuführen.

Das bekannteste Unit Test Framework ist *JUnit*. Dieses wird auch von Nuclos unterstützt. Daneben erlaubt Nuclos auch das Erstellen von *TestNG* Tests.

> Während eclipse schon mit JUnit Support kommt, muss für TestNG ein weiteres Plugin installiert werden.

Praktischer Testaufbau

Ziel von Unit Tests ist es, einzelne Klassen möglichst unabhängig voneinander zu testen. Oft werden die Testfälle, die zu einer Klasse gehören, zwar an einer anderen Stelle vorgehalten wie der 'normale' Quellcode (in maven z.B. unter src/test/java). Um aber auch das Testen von 'package-private' Methoden zu unterstützen, wird der Unit Test von Klasse A im gleichen Paket wie die Klasse A angelegt. So muss die zu testende Klasse nicht modifiziert werden und ist trotzdem weitgehend (bis auf 'private' Methoden und Felder) zugänglich.

Mock Objekte in Unit Tests

Für jeden nicht-trivialen Unit Tests benötigt man Eingabedaten, die von der zu testenden Methode/Klasse verarbeitet werden sollen. Abhängigkeiten der Testklasse zu anderen Klassen werden durch entsprechende *Mock Objekt*e ersetzt. Dieses *Mock Objekte* haben die gleichen Methoden, wie die entsprechende Instanz der Abhängigkeit der Testklasse, *aber alle Methoden sind so überschrieben, dass man **im Testfall** selbst bestimmen kann, wie sich das* Mock Object *verhält*. (Technisch wird das in Java so gelöst, dass die Mock Klasse von der Klasse der 'richtigen' Abhängigkeit erbt. Damit sich das Ganze einfacher benutzen lässt, wird für *Mock Objekte* meist ein *Mock Framework* verwendet.)

Prinzipiell ist es möglich, auch für die Eingabedaten *Mock Objekte* zu verwenden. In der Praxis zeigt sich jedoch, dass es meist erheblich einfacher ist, richtige Objekte als Eingabedaten zu erzeugen, z.B. wenn es sich bei den Eingabedaten um reine *Value Objekte* handelt.

Nuclos unterstützt das klassische Easymock und das neuere Mockito.

Spring Injection in Unit Tests

Prinzipiell können auch Klassen getestet werden, deren Abhängigkeiten mittels Spring Injection im 'Normalbetrieb' von Spring verwaltet werden. In einem solchen Fall könnte man die Klasse selbst instanzieren und für die Injections jeweils *Mock Objekte* erzeugen. Das dies jedoch für sehr viele Tests erforderlich ist (da ja viele Klassen Spring Injection verwenden), wird die Sache natürlich einfacher, wenn man für diese Aufgabe auch Support hat.

Dieser Support wird von Spring-Test und Springockito geliefert und ist für das Schreiben von Tests in Nuclos verfügbar.

Die Idee hinter diesen beiden Libraries ist, dass auch zum Testen ein *Spring (Bean) Context* erzeugt wird, der dann während des Tests zu Verfügung steht. Aus diesen Context heraus können dann *Spring Injections* des Tests (wie z.B. mittels `@Autowired`) aufgelöst werden. Dabei kann es sich bei den *Spring Beans* im Context sowohl um 'echte' Implementierungen des 'Normalbetriebs' handeln als auch um *Mock Objekte*. (Falls man im Context viele 'echte' *Spring Beans* verwendet, so wendet man sich zwar etwas von dem *Unit Test* Gedanken ab, aber es eröffnet sich eine einfache Möglichkeit, begrenzte Integrationstests zu schreiben.)

Der große Vorteil eines *Test Spring Contextes* ist, dass man die zu testenden Klassen nicht verändern muss und auch keinen *Boilerplate Code* benötigt. Die *Spring Injection* funktioniert auch in den Tests weiterhin wie gewohnt, man muss 'nur' im Auge behalten, das der *Test Spring Context* (meist) viele *Mock Objekte* beinhaltet.

Nuclos und Unit Tests

Bei der praktischen Umsetzung von Unit Tests für Nuclos sind mir noch ein paar spezielle Dinge aufgefallen.

Nuclos Client, Common und Server

Da Nuclos ein Client-Server System ist, besteht es grob gesagt aus 3 Teilen:

1. Dem Nuclos Server, der auf dem Tomcat läuft
2. Dem Nuclos Client, der mittels Web Start gestartet wird und eine Swing GUI besitzt
3. Code, der in Nuclos Server und Client verwendet wird (beispielsweise die *Value Objekte*, die zwischen Server und Client ausgetauscht werden (RPC))

Diese Struktur findet man ähnlich auch in den Nuclos Maven Modulen wieder.

Unit Tests können ihrem Wesen nach nur in *einem* dieser Teile laufen (da die Unit Frameworks nur *einen* (Java-VM) Prozess erzeugen). Daher müssen im Server alle Anfragen des Client gemockt werden, und umgekehrt im Client alle Antworten des Servers. In der Praxis ist es praktisch unmöglich, Unit Tests für Swing Klassen zu schreiben, da sich die User Eingaben (per Maus und Tastatur) nicht auf einfachen Wege mocken lassen. Darüber hinaus ist die Trennung zwischen GUI und Logik im Nuclos Client Code nur sehr rudimentär vorhanden. Da sich ferner im gemeinsamen Code (fast) nur *Value Objekte* befinden, lassen sich Unit Test mit vertretbaren Aufwand hauptsächlich *für den* ***Nuclos Server*** *erzeugen*.

Metadaten von BOs und Feldern/Attributen

Konkrete Tests bedingen praktisch immer, dass die Metainformationen der als Eingabedaten verwendeten Objekte vorliegen. Für Nuclos bedeutet dies: Möchte ich einen Test für eine Klasse A schreiben, die als Eingabedaten Instanzen des BOs B verwendet (sei es als EntityObjectVO, MasterDataVO oder GenericObjectVO), dann muss ich die entsprechenden Felder des BOs B setzen können und die Metainformationen (MetaProvider, EntityMeta, FieldMeta) für das BO B müssen zum Testzeitpunkt abrufbar sein. Dies ist notwendig, da sehr viele zu testende Klassen Metainformationen über BOs abrufen und verarbeiten (und es sehr unpraktisch ist, all diese Metainformation für jeden Aufruf zu *mocken*).

Die Metainformationen werden jedoch bei Nuclos *konfiguriert* (und liegen daher nicht in Code-Form vor). Daher war es an dieser Stellen notwendig, speziell für Nuclos Unit Tests einen *speziellen MetaProvider* (`org.nuclos.server.common.TestMetaProvider`) zur Verfügung zu stellen. Dieser `TestMetaProvider` liest die Metainformationen in Form einer XML Datei, von einem Nuclos Server zur Runtime mittels *JMX Operation* (`MetaProvider#dumpAllEntities(boolean)`) erzeugt wurde. Durch diesen Trick stehen 'richtige' Metainformationen einer Nuclos Server Instanz auch für Unit Tests zur Verfügung.

TODO: Zeigen, wie man die XML Datei, mittels JMX dumped.

Ein Beispiel mit TestNG und Mockito

Im folgenden ein Überblick, wie man Unit Test in einer Extension mittels TestNG und Mockito aufsetzt.

Anpassung in pom.xml der Extension

Da Testabhängigkeiten in maven nicht so wie normale Abhängigkeiten vererbt werden, muss im pom.xml in jedem (Maven) Modul, welches Unit Tests verwenden möchte, folgendes bei den Abhängigkeiten ergänzt werden:

<extension-server>/pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:
schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">

    [...]

    <dependencies>

    [...]

        <!-- testing -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.testng</groupId>
            <artifactId>testng</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.easymock</groupId>
            <artifactId>easymock</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.mockito</groupId>
            <artifactId>mockito-core</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-test</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.kubek2k</groupId>
            <artifactId>springockito</artifactId>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.kubek2k</groupId>
            <artifactId>springockito-annotations</artifactId>
            <scope>test</scope>
        </dependency>

    </dependencies>

    [...]

</project>
```

Erstellung von testng.xml

Für TestNG Unit Tests ist die Datei testng.xml verpflichtend. In ihr wird definiert, welche Tests während des CI Build standardmäßig ausgeführt werden. Um alle Tests auszuführen und Log4j für die Test zu konfigurien, kann die Datei wie folgt aussehen:

<extension-server>/testng.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd">
<suite name="Suite" parallel="false">
    <listeners>
        <listener class-name="de.fdm.testng.LoggingListener" />
    </listeners>
    <test name="Test">
        <packages>
            <package name="de.fdm.*" />
        </packages>
    </test>
    <!-- Test -->
</suite>
<!-- Suite -->
```

Bei der Definition dieser XML Datei muss man leider einige Stolperfallen beachten.

Log4j für Unit Tests konfigurien

Da viele zu testende Klassen mit Logging arbeiten, ist es praktisch immer notwendig, dieses auch zu konfigurieren. In Nuclos wird Log4j als Logging Library verwendet. Daher:

<server-extension>/src/test/java/log4j.properties

```
log4j.appender.logfile=org.apache.log4j.RollingFileAppender
log4j.appender.logfile.File=nuclos-test.log
log4j.appender.logfile.MaxBackupIndex=10
log4j.appender.logfile.MaxFileSize=2GB
log4j.appender.logfile.layout=org.apache.log4j.PatternLayout
#log4j.appender.logfile.layout.ConversionPattern=%d %p [%c] - %m%n
log4j.appender.logfile.layout.ConversionPattern=%d %-5p %-9t [%c{3}] - %m%n
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
#log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
log4j.appender.stdout.layout.ConversionPattern=%d %-5p %-9t [%c{3}] - %m%n
log4j.rootLogger=INFO, stdout, logfile

# http://www.benmccann.com/dev-blog/sample-log4j-properties-file/

# Adjust log levels

# SQL logging
log4j.logger.org.nuclos.server.dblayer=DEBUG

#log4j.logger.org.springframework=DEBUG
```

Die Konfiguration von Log4j ermöglicht es auch, im Unit Test selbst das Logging zu verwenden. Das ist bei der Testentwicklung meist recht praktisch.

Unit Test in TestNG

Klassendefinition der Unit Test Klasse

Die Klassendefinition für einen TestNG Unit Test könnte wie folgt aussehen:

/<extension-server>/src/test/java/de/fdm/server/explorer/PdmcaeExplorerFacadeBeanTest.java

```
package de.fdm.server.explorer;

import static org.testng.Assert.*;
import static org.mockito.Matchers.*;
import static org.mockito.Mockito.*;
import static de.bmw.fdm.common.FdmCommon.*;

import org.apache.log4j.Logger;
import org.kubek2k.springockito.annotations.SpringockitoContextLoader;
import org.nuclos.common.EntityMeta;
import org.nuclos.common.UID;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;
import org.testng.annotations.AfterTest;
import org.testng.annotations.BeforeClass;
import org.testng.annotations.Test;

[...] // weitere imports

@ContextConfiguration(loader=SpringockitoContextLoader.class,
    locations= {"classpath:de/bmw/fdm/server/explorer/spring.xml"})
public class PdmcaeExplorerFacadeBeanTest extends AbstractTestNGSpringContextTests {

    private static final Logger LOG = Logger.getLogger(PdmcaeExplorerFacadeBeanTest.class);

        [...] // siehe weiter unten

}
```

Diese Beispiel benutzt org.springframework.test.context.testng.AbstractTestNGSpringContextTests, um einen TestNG zu erzeugen, der mit einem *Spring Test Context* arbeitet. Soll der Test zudem transaktional ablaufen, müsste statt dessen von AbstractTransactionalTestNGSpringContextTests geerbt werden.

Die `@ContextConfiguration` Annotation stellt sicher, dass innerhalb des *Spring Test Contextes* ganz einfach *Mocks* erzeugt werden können. Und sie gibt die XML Datei an, aus der der Spring Test Context erzeugt werden soll. Oft kann man eine XML Context Definition für mehrere Tests verwenden. (Meist versuche ich nur einen XML Context Definition pro zu testenden (Java) Paket zu verwenden.)

Spring Injection in der Unit Test Klasse

Im Quellcode folgen nun die Spring Injections:

/<extension-server>/src/test/java/de/fdm/server/explorer/PdmcaeExplorerFacadeBeanTest.java

```
[...] // siehe oben

    private final DataFactory dataFactory = new DataFactory();

    @Autowired
    private PdmcaeExplorerFacadeRemote dut;

    // mocks from spring context

    @Autowired
    private MasterDataFacadeLocal masterDataFacade;

    @Autowired
    private CollectableEOEntityProvider collectableEntityProvider;

    @Autowired
    private MasterDataTreeNodeFactory mdTreeNodeFactory;

    @Autowired
    private MetaProvider metaProvider;

    // end of mocks from spring context
```

Zunächst wird eine Hilfsklasse erzeugt, die für die Erzeugung der für den Test nötigen *Input Value Objects* verwendet wird (s.u.). Oft reicht es eine solche Hilfsklasse pro zu testenden (Java) Pakets zu erzeugen. Es ist üblich, diese Klasse `DataFactory` zu nennen. Eine alternative zur direkten Erzeugung wäre die *Spring Injection* auch der `DataFactory`. Dies hätte zudem den Vorteil, dass auch innerhalb der `DataFactory` *Spring Injektion* verwendet werden könnte.

Anschließend wird die zu testende Klasse injiziert (`dut` ist hier der kanonische Name *Device under Test*). Die Methoden dieser Klasse sollen von Unit Test überprüft werden.

Darauf folgend werden verschiedene *Spring Beans* des *Spring Test Contextes* injiziert. Dabei handelt es sich in diesem Fall ausschließlich um *Mock Objekte*.

Boilerplate Code

/<extension-server>/src/test/java/de/fdm/server/explorer/PdmcaeExplorerFacadeBeanTest.java

```
[...] // siehe oben

    @BeforeClass(dependsOnMethods={"springTestContextPrepareTestInstance"})
    public void setupParamValidation(){
        // Test class setup code with autowired classes goes here
    }

    @AfterTest(alwaysRun=true)
    public void afterTest() {
        reset(masterDataFacade);
    }
```

Ein Beispiel für Boilerplate Code, den man recht oft benötigt. Die Method `setupParamValidation` hat Platz vor Code, der nach der Spring Context Erzeugung und vor allen Tests ausgeführt werden soll. Wichtig ist hier nicht der Methodenname, sondern die Annotierung. Diese Methode hat für den TestNG ungefähr die Funktion, die eine mit `@PostConstruct` annotierte Methode für Spring Beans hat.

Mock Objekte muss man nach jeden Tests auf den Ausgangszustand zurücksetzen, damit Problem, die sich in einem Test ergeben, nicht in den danach ausgeführten Test propagieren. In TestNG gelingt dies mit der Annotation `@AfterTest(alwaysRun=true)`. Typischerweise wird das `reset` von Mockito für die Mock Objekte aufgerufen, die mittels Spring injiziert wurden.

Typischer Test

/<extension-server>/src/test/java/de/fdm/server/explorer/PdmcaeExplorerFacadeBeanTest.java

```
[...] // siehe oben

    @Test
    public void testGetSubNodesForRootSubnodes1() throws CommonBusinessException {
        // u.U. wegen afterTest Method nicht notwendig
        reset(masterDataFacade);

        // prepare input data
        final MasterDataVO<Long> sub1 = dataFactory.getBO_Fahrzeugbaum(null, "sub1");
        final Long sub1Id = sub1.getPrimaryKey();
        final PdmcaeTreeNode sub1Node = dataFactory.asPdmcaeTreeNode(sub1);
        final MasterDataVO<Long> sub2 = dataFactory.getBO_Fahrzeugbaum(null, "sub2");
        final Long sub2Id = sub2.getPrimaryKey();
        final PdmcaeTreeNode sub2Node = dataFactory.asPdmcaeTreeNode(sub2);
        final MasterDataVO<Long> subsub = dataFactory.getBO_Fahrzeugbaum(sub2, "subOfSub2");
        final Long subsubId = subsub.getPrimaryKey();
        final PdmcaeTreeNode subsubNode = dataFactory.asPdmcaeTreeNode(subsub);

        final PdmcaeRootTreeNode root = PdmcaeRootTreeNode.getInstance();
        root.setSearchResultSubnodeIds((Set) dataFactory.set(sub1Id, null, sub2Id));

        // prepare mocks
        // for getSubnodesByIds
        when(masterDataFacade.getMasterData(eq(BO_Fahrzeugbaum), (CollectableSearchCondition) any(), eq
(true))).thenReturn(
                (TruncatableCollection) TruncatableCollectionDecorator.createUntruncated(dataFactory.list
(sub1, sub2)));

        // Method to test
        final List<TreeNode> result = dut.getSubNodes(root);
        LOG.info("getSubNodesForRootSubnodes1: result is " + result);

        // return value verification
        assertEquals(result.size(), 2);
        assertTrue(result.contains(sub1Node));
        assertTrue(result.contains(sub2Node));
        assertFalse(result.contains(subsubNode));

        // mock verification
        verify(masterDataFacade).getMasterData(eq(BO_Fahrzeugbaum), (CollectableSearchCondition) any(), eq
(true));
        verifyNoMoreInteractions(masterDataFacade);
    }
```

Der eigentliche Unit Test gliedert sich in folgende Teile:

1. `@Test` Annotierung
2. `reset` Mock Objekte (optional)
3. Aufbau der/des *Input Value Object(s)*
4. Definition, was von den Mock Objekten zur Laufzeit erwartet wird (d.h. Definition der erwarteten Methodenaufrufe und deren Rückgabewerte)
5. Aufruf der zu testenden Methode
6. Verifikation des Ergebnisses (d.h. des Rückgabewertes und der Änderungen an den *Input Value Objects*)
7. Verifikation der Mock Objekte

An diesem einfachen Testfall kann man bereits erkennen, das er Aufruf der zu testenden Methode nur ein ganz kleiner Teil des Unit Testes ist. Den größten Teil eines Tests nimmt meist die Behandlung der Mock Objekte in Anspruch. Es ist daher notwendig, sich mit dem verwendeten Mock Framework gut auszukennen.

DataFactory

/<extension-server>/src/test/java/de/fdm/server/explorer/DataFactory.java

```
package de.bmw.fdm.server.explorer;

import static de.fdm.common.FdmCommon.BO_Datenbedarf;
import static de.fdm.common.FdmCommon.BO_Datenpakettyp;
```

```
import static de.fdm.common.FdmCommon.BO_Fahrzeugbaum;
import static de.fdm.common.FdmCommon.BO_Knoten;
import static de.fdm.common.FdmCommon.F_Datenbedarf_fahrzeugbaum;
import static de.fdm.common.FdmCommon.F_Datenbedarf_link;
import static de.fdm.common.FdmCommon.F_Datenbedarf_modell;
import static de.fdm.common.FdmCommon.F_Datenbedarf_pfad;
import static de.fdm.common.FdmCommon.F_Datenbedarf_produktlinie;
import static de.fdm.common.FdmCommon.F_Datenbedarf_sachnummer;
import static de.fdm.common.FdmCommon.F_Datenpakettyp_bkrelevant;
import static de.fdm.common.FdmCommon.F_Datenpakettyp_knoten;
import static de.fdm.common.FdmCommon.F_Fahrzeugbaum_fahrzeug;
import static de.fdm.common.FdmCommon.F_Fahrzeugbaum_knotenname;
import static de.fdm.common.FdmCommon.F_Fahrzeugbaum_uebergeordnknoten;
import static de.fdm.common.FdmCommon.F_Knoten_bezeichnung;
import static de.fdm.common.FdmCommon.F_Knoten_uebergeordnknoten;

import static org.mockito.Mockito.mock;
import static org.mockito.Mockito.when;

import java.util.ArrayList;
import java.util.Date;
import java.util.HashSet;
import java.util.List;
import java.util.Random;
import java.util.Set;
import org.apache.log4j.Logger;

import org.nuclos.common.UID;
import org.nuclos.common.UsageCriteria;
import org.nuclos.common.collection.Pair;
import org.nuclos.common.dal.vo.EntityObjectVO;
import org.nuclos.server.dal.DalSupportForGO;
import org.nuclos.server.genericobject.valueobject.GenericObjectVO;
import org.nuclos.server.masterdata.valueobject.MasterDataVO;
import org.nuclos.server.navigation.treenode.DefaultMasterDataTreeNode;
import org.nuclos.server.navigation.treenode.DefaultMasterDataTreeNodeParameters;
import org.nuclos.server.navigation.treenode.GenericObjectTreeNode.RelationDirection;
import org.nuclos.server.navigation.treenode.GenericObjectTreeNode.SystemRelationType;

import de.fdm.common.FdmCommon;
import de.fdm.common.explorer.FolderTreeNode;
import de.fdm.common.explorer.PdmcaeTreeNode;

class DataFactory {

    private static final Logger LOG = Logger.getLogger(DataFactory.class);

    //

    private final Random rand = new Random(9132278275L);

    DataFactory() {
    }

    EntityObjectVO<Long> finishEo(EntityObjectVO<Long> eo) {
        eo.setPrimaryKey(rand.nextLong());
        return eo;
    }

    GenericObjectVO getBO_KnotenAsGO(GenericObjectVO parent, String name) {
        return DalSupportForGO.getGenericObjectVOForUnitTest(getBO_Knoten(parent, name));
    }

    EntityObjectVO<Long> getBO_Knoten(GenericObjectVO parent, String name) {
        final EntityObjectVO<Long> eo = new EntityObjectVO<Long>(BO_Knoten);

        eo.setFieldValue(F_Knoten_bezeichnung, name + "Bez");
        if (parent != null) {
            eo.setFieldId(F_Knoten_uebergeordnknoten, parent.getPrimaryKey());
        } else {
            eo.setFieldId(F_Knoten_uebergeordnknoten, null);
```

```
        }
        finishEo(eo);
        // final MasterDataVO<Long> der = new MasterDataVO<Long>(finishEo(eo));
        return eo;
    }

    MasterDataVO<Long> getBO_Datenpakettyp(GenericObjectVO parent, String name) {
        final EntityObjectVO<Long> eo = new EntityObjectVO<Long>(BO_Datenpakettyp);

        // BO_Knoten
        eo.setFieldValue(F_Knoten_bezeichnung, name + "Bez");
        if (parent != null) {
            eo.setFieldId(F_Datenpakettyp_knoten, parent.getPrimaryKey());
        } else {
            eo.setFieldId(F_Datenpakettyp_knoten, null);
        }

        // BO_Datenpakettyp
        eo.setFieldValue(F_Datenpakettyp_bkrelevant, true);

        final MasterDataVO<Long> der = new MasterDataVO<Long>(finishEo(eo));
        return der;
    }

    MasterDataVO<Long> getBO_Fahrzeugbaum(MasterDataVO<Long> parent, String name) {
        final EntityObjectVO<Long> eo = new EntityObjectVO<Long>(BO_Fahrzeugbaum);

        eo.setFieldValue(F_Fahrzeugbaum_knotenname, name);
        eo.setFieldValue(F_Fahrzeugbaum_fahrzeug, name + "Vehicle");
        if (parent != null) {
            eo.setFieldId(F_Fahrzeugbaum_uebergeordnknoten, parent.getPrimaryKey());
        } else {
            eo.setFieldId(F_Fahrzeugbaum_uebergeordnknoten, null);
        }

        final MasterDataVO<Long> result = new MasterDataVO<Long>(finishEo(eo));
        return result;
    }

    MasterDataVO<Long> getBO_Datenbedarf(MasterDataVO<Long> parentFahrzeugbaum, String name) {
        final EntityObjectVO<Long> eo = new EntityObjectVO<Long>(BO_Datenbedarf);

        eo.setFieldValue(F_Datenbedarf_sachnummer, name + "Sachnummer");
        eo.setFieldValue(F_Datenbedarf_pfad, name + "Pfad");
        eo.setFieldValue(F_Datenbedarf_modell, name + "Modell");
        eo.setFieldValue(F_Datenbedarf_produktlinie, name + "PL");
        eo.setFieldValue(F_Datenbedarf_link, name + "Link");
        if (parentFahrzeugbaum != null) {
            eo.setFieldId(F_Datenbedarf_fahrzeugbaum, parentFahrzeugbaum.getPrimaryKey());
        } else {
            eo.setFieldId(F_Datenbedarf_fahrzeugbaum, null);
        }

        final MasterDataVO<Long> result = new MasterDataVO<Long>(finishEo(eo));
        return result;
    }

    PdmcaeTreeNode asPdmcaeTreeNode(MasterDataVO<Long> md) {
        final String name = (String) md.getFieldValue(FdmCommon.F_Fahrzeugbaum_knotenname);
        final String vehicle = (String) md.getFieldValue(FdmCommon.F_Fahrzeugbaum_fahrzeug);
        final Number id = md.getPrimaryKey();
        final PdmcaeTreeNode result = new PdmcaeTreeNode(id, name, true, id, name, vehicle);

        return result;
    }

    DefaultMasterDataTreeNode asDefaultMasterDataTreeNode(MasterDataVO<Long> md, String name) {
        final Long id = md.getPrimaryKey();
        final DefaultMasterDataTreeNodeParameters<Long> params = new
DefaultMasterDataTreeNodeParameters<Long>(md.getEntityObject().getDalEntity(),
                id, null, null, name + "Label", name + "Description");
```

```
        final DefaultMasterDataTreeNode<Long> result = new DefaultMasterDataTreeNode<Long>(params);
        return result;
    }

    FolderTreeNode asFolderTreeNode2(int level, GenericObjectVO go, List subNodes) {
        final FolderTreeNode result = mock(FolderTreeNode.class);
        when(result.getId()).thenReturn(go.getId());
        when(result.getEntityUID()).thenReturn(FdmCommon.BO_Knoten);
        when(result.hasSubNodes()).thenReturn(subNodes != null && !subNodes.isEmpty());
        when(result.getSubNodes()).thenReturn(subNodes);
        when(result.getLevel()).thenReturn(level);
        return result;
    }

    FolderTreeNode asFolderTreeNode(int level, Long id, UsageCriteria usagecriteria,
            String sIdentifier, Long relationId,
            SystemRelationType relationtype, RelationDirection direction,
            String sUserName, UID state, UID uidNode, Long idRoot) {
        if (usagecriteria == null) {
            usagecriteria = new UsageCriteria(null, null, null, "pseudo-mocked usagecriteria");
        }
        final FolderTreeNode result = new FolderTreeNode(id, usagecriteria,
                sIdentifier, relationId,
                relationtype, direction,
                sUserName, state, uidNode, idRoot);
        result.setLevel(level);
        return result;
    }

    Pair<Date,Date> interval() {
        final long now = System.currentTimeMillis();
        Date start = new Date(now + rand.nextInt(1234567890));
        Date end = new Date(now + rand.nextInt(1234567890));
        if (start.after(end)) {
            final Date tmp = start;
            start = end;
            end = tmp;
        }
        return new Pair<Date, Date>(start, end);
    }

    void reset() {
    }

    Random getRand() {
        return rand;
    }

    <S,T> List<T> list(S... ts) {
        final List<T> result = new ArrayList<T>(ts.length);
        for (S t: ts) {
            result.add((T) t);
        }
        return result;
    }

    <S,T> Set<T> set(S... ts) {
        final Set<T> result = new HashSet<T>(ts.length);
        for (S t: ts) {
            result.add((T) t);
        }
        return result;
    }
}
```

Die `DataFactory` stellt die *Value Objekte* als Eingabe für den Test zur Verfügung. Bei Nuclos handelt es sich dabei oft um `EntityObjectVO`s, `MasterDataVO`s und/oder `GenericObjectVO`s. Das obrige Beispiel benutzt alle 3 Objektarten. Entscheiden für die Tests ist es, (zumindest) für jedes *Value Objekt* das referenziert wird, auch einen *Primary Key* zu hinterlegen. Dies gelingt besonders einfach mit einem Zufallsgenerator, dessen Startbedingung festgelegt wird. Ein solcher Generator liefert nämlich (bei jedem Test) immer wieder die gleiche (Pseudo-Zufalls-)Zahlenfolge. Nummern dieser Folge werden im Beispiel als *Primary Key* benutzt.

Definition Spring Context

/<extension-server>/src/test/resources/de/bmw/fdm/server/explorer/spring.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:jms="http://www.springframework.org/schema/jms"
    xmlns:amq="http://activemq.apache.org/schema/core"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:sec="http://www.springframework.org/schema/security"
    xmlns:jee="http://www.springframework.org/schema/jee"
    xmlns:mockito="http://www.mockito.org/spring/mockito"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
                        http://www.springframework.org/schema/beans/spring-beans.xsd
                        http://www.springframework.org/schema/context
                        http://www.springframework.org/schema/context/spring-context.xsd
                        http://www.springframework.org/schema/jms
                        http://www.springframework.org/schema/jms/spring-jms.xsd
                        http://activemq.apache.org/schema/core
                        http://activemq.apache.org/schema/core/activemq-core.xsd
                        http://www.springframework.org/schema/tx
                        http://www.springframework.org/schema/tx/spring-tx.xsd
                        http://www.springframework.org/schema/aop
                        http://www.springframework.org/schema/aop/spring-aop.xsd
                        http://www.springframework.org/schema/security
                        http://www.springframework.org/schema/security/spring-security.xsd
                        http://www.springframework.org/schema/jee
                        http://www.springframework.org/schema/jee/spring-jee.xsd
                        http://www.mockito.org/spring/mockito
                        https://bitbucket.org/kubek2k/springockito/raw/tip/springockito/src/main/resources
/spring/mockito.xsd">

    <context:annotation-config />
    <context:spring-configured />

    <!-- No component scanning! - we place any bean needed here by hand. (tp) -->

    <!-- context:component-scan base-package="org.nuclos">
        <context:exclude-filter expression=".*_Roo_.*"
        type="regex" />
        <context:exclude-filter expression="org.springframework.stereotype.Controller"
        type="annotation" />
        </context:component-scan -->

    <!-- Avoid scanning for nuclos-common (this is important for *client* performance) -->
    <bean class="org.nuclos.common.ApplicationProperties"/>
    <bean id="appContext" class="org.nuclos.common.SpringApplicationContextHolder" />
    <bean id="domPreferencesFactory" class="org.nuclos.common.preferences.DOMPreferencesFactory" />

    <!-- Services/Beans -->
    <mockito:mock id="masterDataService" class="org.nuclos.server.masterdata.ejb3.MasterDataFacadeBean"/>
    <mockito:mock id="genericObjectService" class="org.nuclos.server.genericobject.ejb3.
GenericObjectFacadeBean" />
    <mockito:mock id="parameterService" class="org.nuclos.server.common.ejb3.ParameterFacadeBean" />
    <mockito:mock id="stateService" class="org.nuclos.server.statemodel.ejb3.StateFacadeBean" />
    <mockito:mock id="serverMetaService" class="org.nuclos.server.servermeta.ejb3.ServerMetaFacadeBean" />
    <mockito:mock id="localeService" class="org.nuclos.server.common.ejb3.LocaleFacadeBean" />
    <mockito:mock id="preferencesService" class="org.nuclos.server.common.ejb3.PreferencesFacadeBean" />
    <mockito:mock id="metaDataService" class="org.nuclos.server.masterdata.ejb3.MetaDataFacadeBean"/>
    <mockito:mock id="resourceService" class="org.nuclos.server.resource.ejb3.ResourceFacadeBean" />
    <mockito:mock id="entityService" class="org.nuclos.server.masterdata.ejb3.EntityFacadeBean" />
    <mockito:mock id="entityObjectService" class="org.nuclos.server.common.ejb3.EntityObjectFacadeBean" />
    <mockito:mock id="treeNodeService" class="org.nuclos.server.navigation.ejb3.TreeNodeFacadeBean" />
    <mockito:mock id="lookupService" class="org.nuclos.server.common.ServerLocaleDelegate" />
    <mockito:mock id="parameterProvider" class="org.nuclos.server.common.ServerParameterProvider"/>
```

```
    <mockito:mock id="attributeProvider" class="org.nuclos.server.common.AttributeCache" />
    <mockito:mock id="nodeProvider" class="org.nuclos.server.common.NodeCache" />
    <mockito:mock id="moduleProvider" class="org.nuclos.server.genericobject.Modules" />
    <mockito:mock id="stateCache" class="org.nuclos.server.common.StateCache" />

    <!-- API beans -->
    <bean id="clientPreferences" class="org.nuclos.common2.ClientPreferences">
        <property name="preferencesFactory" ref="nuclosPreferencesSingleton" />
    </bean>
    <bean id="nuclosPreferencesSingleton"
        class="org.nuclos.common.preferences.NuclosPreferencesSingleton">
        <property name="preferencesFacadeRemote" ref="preferencesService" />
        <property name="preferencesFactory" ref="domPreferencesFactory" />
    </bean>

    <!-- mockito:mock id="treeService" class="org.nuclos.common.metadata.TreeMetaProvider" / -->
    <bean id="treeService" class="org.nuclos.common.metadata.TreeMetaProvider" />

    <mockito:mock id="mdTreeNodeFactory"
        class="org.nuclos.server.navigation.treenode.MasterDataTreeNodeFactory" />

    <!-- generic support for *FacadeLocal proxies -->

    <bean id="facadeLocalProxyFactoryBean" class="org.nuclos.server.spring.FacadeLocalProxyFactoryBean" />
    <bean id="facadeLocalProxyBeanFactoryPostProcessor" class="org.nuclos.server.spring.
FacadeLocalProxyBeanFactoryPostProcessor" />

    <!-- entity/field meta data support -->

    <bean id="xstreamSupport" class="org.nuclos.common2.XStreamSupport"/>

    <bean id="metaDataProvider" class="org.nuclos.server.common.TestMetaProvider">
        <constructor-arg>
            <!--
                this has been dump before by JMX with
                org.nuclos.server.common.MetaProvider.dumpAllEntities(boolean false)
            -->
            <value>classpath:bmw-fdm-meta.xml</value>
        </constructor-arg>
    </bean>

    <!-- Unavoidable dependencies of *ExplorerFacadeBeans -->

    <mockito:mock id="securityCache" class="org.nuclos.server.common.SecurityCache" />
    <mockito:mock id="dynamicMetaDataProcessor" class="org.nuclos.server.dal.processor.jdbc.impl.
DynamicMetaDataProcessor" />
    <mockito:mock id="chartMetaDataProcessor" class="org.nuclos.server.dal.processor.jdbc.impl.
ChartMetaDataProcessor" />
    <mockito:mock id="nucletDalProvider" class="org.nuclos.server.dal.provider.NucletDalProvider" />
    <mockito:mock id="recordGrantUtils" class="org.nuclos.server.common.RecordGrantUtils" />
    <mockito:mock id="nuclosUserDetailsContextHolder" class="org.nuclos.server.common.
NuclosUserDetailsContextHolder" />
    <mockito:mock id="nuclosRemoteContextHolder" class="org.nuclos.server.common.NuclosRemoteContextHolder"
/>
    <mockito:mock id="masterDataFacadeHelper" class="org.nuclos.server.masterdata.ejb3.
MasterDataFacadeHelper" />
    <mockito:mock id="springDataBaseHelper" class="org.nuclos.server.database.SpringDataBaseHelper" />
    <mockito:mock id="collectableEOEntityProvider" class="org.nuclos.common.entityobject.
CollectableEOEntityProvider" />

    <!-- Unavoidable local facades -->

    <mockito:mock id="entityObjectFacadeLocal" class="org.nuclos.server.common.ejb3.EntityObjectFacadeLocal"
/>
    <mockito:mock id="masterDataFacadeLocal" class="org.nuclos.server.masterdata.ejb3.MasterDataFacadeLocal"
/>
    <mockito:mock id="genericObjectFacadeLocal" class="org.nuclos.server.genericobject.ejb3.
GenericObjectFacadeLocal" />
    <mockito:mock id="treeNodeFacadeLocal" class="org.nuclos.server.navigation.ejb3.TreeNodeFacadeLocal" />

    <!-- Classes subject to test -->
```

```
    <bean class="de.bmw.fdm.server.explorer.VehicleExplorerFacadeBean" depends-on="
facadeLocalProxyBeanFactoryPostProcessor"/>
    <bean class="de.bmw.fdm.server.explorer.PdmcaeExplorerFacadeBean" depends-on="
facadeLocalProxyBeanFactoryPostProcessor"/>

</beans>
```

Der XML Definition für den *Test Spring Context* erscheint auf den ersten Blick recht kompliziert. Allerdings sind viele Teile des Test Kontext stets gleich, so dass sich der Aufwand für die Erstellung doch in Grenzen hält. Der Kontext*:*

1. Eine große Menge unvermeidbarer *Mock Objekte*. Diese werden durch `mockito:mock` Tags erzeugt.
2. Boilerplate Code zur Konfiguration von Spring.
3. Die beiden zu testenden Klassen PdmcaeExplorerFacadeBean und VehicleExplorerFacadeBean.
4. Den TestMetaProvider, der die Metadaten für die BOs bereitstellt (s.o.).