Faktor-IPS Test-Konzept

von Alexander Weickmann

Motivation

Die momentane Vorgehensweise bei der Entwicklung von Unit Tests für Faktor-IPS bringt verschiedene Nachteile mit sich.

- **Starten von Eclipse zur Ausführung der Tests**
  Beim Starten eines Eclipse Plug-in Tests muss zunächst eine neue Eclipse-Instanz gestartet werden. Das dauert bei jedem Start mehrere Sekunden und verringert die Produktivität der Entwicklung.

- **Enorme Laufzeiten**
  Die Laufzeiten der Unit Tests sind enorm. Entwickler können die Tests daher auf ihrer lokalen Maschine nicht mehr wirklich ausführen. Die Hudson-Builds dauern aus diesem Grund auch enorm lange, was sich abermals auf die Produktivität der Entwicklung negativ auswirkt. Die enormen Laufzeiten kommen hauptsächlich dadurch zustande, dass Klassen wie `IpsProject` nicht gemockt werden und beispielsweise bei jedem Testfall tatsächlich eine Projektstruktur auf der Festplatte angelegt wird.

- **Schlechte Skalierbarkeit**
  Die Probleme aufgrund der langen Laufzeiten der Unit Tests werden sich verschlimmern, umso umfangreicher Faktor-IPS und damit auch die Tests werden. Insgesamt wurde bisher nur verhältnismäßig wenig an Faktor-IPS entwickelt. Stellt man sich vor, dass z.b. 10 Personen Vollzeit an Faktor-IPS arbeiten, dann würden die Laufzeiten der Hudson-Builds ins Unerträgliche ansteigen.
  Es ist eine Best Practice, für jeden Testfall auch wirklich eine eigene Testmethode im Code zu haben. Dies erhöht nicht nur die Übersichtlichkeit und Wartbarkeit, sondern stellt auch sicher, dass die verschiedenen Testfälle unabhängig voneinander behandelt werden können. Bei Faktor-IPS sind die meisten Testmethoden ein Zusammenschluss von mehreren Testfällen. Würde man diese adäquat aufsplitten wollen, verschlimmert sich das Problem der enormen Laufzeiten weiter.

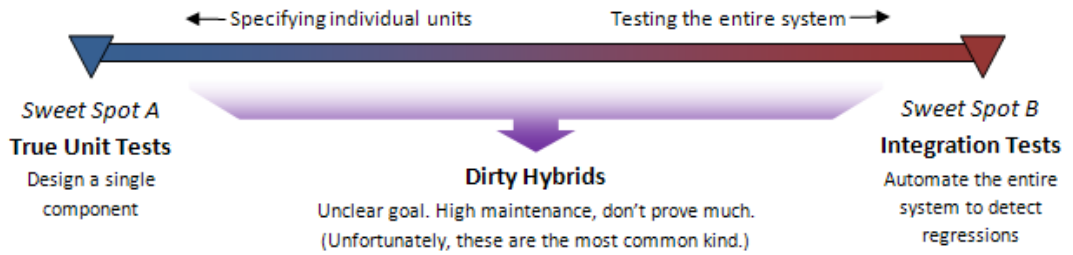
- **Schlechte Isolation**

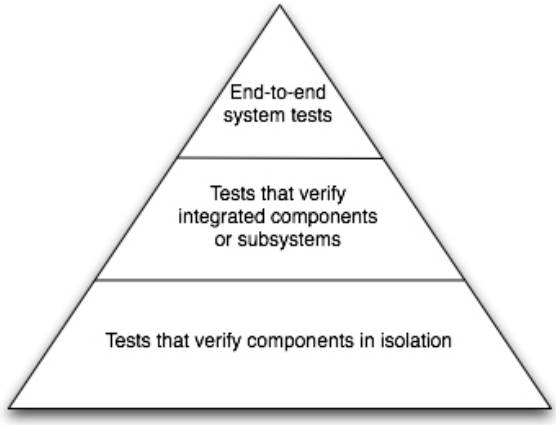
  Da bei den Unit Tests die Klassen, mit denen eine zu testende Klasse kooperiert, nicht gemockt werden, ist der Isolationsgrad dieser Tests gering. Dies führt dazu, dass bei einer Änderung einer Klasse häufig Tests fehlschlagen, die mit der veränderten Klasse gar nichts zu tun haben. Aufgrund der langen Laufzeiten der Tests stellt der Entwickler dies auch erst bei einem Hudson Build Failed fest. Nun beginnt die Fehlersuche in den Tests (meist ein False-Positive da nur Testcode fehlerhaft ist, nicht aber wirklicher Code). Andere Entwickler sollen während dieser Zeit nicht mehr einchecken.

  Da kooperierende Klassen nicht gemockt werden, kann man sich nie ausschließlich nur auf die zu testende Klasse konzentrieren, sondern muss immer auch sehen wie man die kooperierenden Objekte in einen validen Zustand bringt.

- **Unklares Ziel**

  Es wird nicht angemessen zwischen den Konzepten „Unit Test“ und „Integration Test“ unterschieden. Somit ist neben den oben genannten Nachteilen auch unklar, was ein Test überhaupt zeigen soll. Das Ziel eines Unit Tests ist es, zu zeigen dass eine Klasse sich so verhält, wie es in den Testfällen spezifiziert wurde. Das ermöglicht sicheres Refactoring und bietet eine solide Basis für Weiterentwicklung. Es ist **nicht** das Ziel eines Unit Tests, Bugs zu finden – dies haben Integrationstests und manuelle Tests zum Ziel. Bei den derzeitigen Faktor-IPS Tests sind die Konzepte vermischt. Somit wird weder das eine, noch das andere Ziel adäquat erreicht.





Unit Tests für bereits existierenden Code schreiben bringt nichts

Bei Faktor-IPS werden meistens die Tests erst nach dem Code geschrieben. Dies ist nicht TDD! Wie bereits oben erwähnt, ist es das Ziel eines Unit Tests sicherzustellen, dass sich eine Klasse so verhält wie ihr Verhalten im Test spezifiziert wurde. Indem man zuerst einen Testfall (die Spezifikation einer Methode) schreibt, kann man sich nachher bei der Implementierung sicher sein, dass man genau das programmiert, was zur Erfüllung der Spezifikation notwendig ist – nicht weniger und auch nicht mehr. Wenn man den Test nicht vor der Implementierung schreibt, woher weiß man dann überhaupt was man implementieren muss? Man macht sich darüber während der Implementierung Gedanken. Dadurch vermischen sich die Aktivitäten Spezifizierung (oder auch Design) und Implementierung, was z.B. oft dazu führt, dass die Implementierung komplizierter geschrieben wird als nötig..

Indem man zuerst einen Testfall (die Spezifikation) schreibt, erzeugt man ein hohes Maß an Sicherheit bei der Implementierung. Die Implementierungsphase verläuft dann immer mit klarem und messbarem Ziel. Ist man einmal mit dieser Methode vertraut, so kommt einem alles andere als „unsicher“ vor. Darüber hinaus stellt die Methode stetigen Fortschritt in kleinen Schritten sicher.

Gegenargument: *„Wenn ich zuerst versuche die Tests zu schreiben, dann habe ich doch gar keinen Überblick über den Code und weiß gar nicht, was ich testen soll.“*

Das ist eben genau falsch. Bei TDD muss man entgegen der offenbar weit verbreiteten Meinung **nicht** eine komplette Test-Klasse schreiben, bevor man mit der Implementierung beginnt. Man schreibt immer einen einzelnen Testfall und dann die minimal notwendige Implementierung um diesen einen Testfall zu erfüllen. Dadurch ist der Entwickler auch jederzeit mit der Implementierungs-Seite vertraut und begibt sich mit der Spezifikation nicht auf Glatteis. Nachdem man ein Klassendiagramm hat, kann man beginnen eine Klasse anzulegen und eine einzelne Methode zu definieren, die man auch sofort dokumentiert (wenn nicht dokumentiert ist, was eine Methode überhaupt machen soll, was soll man dann testen?). Nun beginnt der TDD-Prozess: Testfall, Implementierung, Refactoring, wiederholen – so lange bis die Klasse komplett spezifiziert und implementiert ist.

Wenn ein Unit Test aber nun dazu da ist, ein Ziel für die Implementierung vorzugeben, dann bringt es nichts, einen Unit Test erst nach der Implementierung zu schreiben.

Gegenargument: *„Mit den Tests können aber später Bugs gefunden werden.“* ← Das ist nicht das Ziel eines Unit Tests. Ein Unit Test kann dies nicht leisten, da im fertigen System die einzelnen Units nicht isoliert sind sondern zusammenarbeiten müssen.

Fallstudie: MultiLanguageSupportTest

Faktor-IPS bietet seit kurzem die Möglichkeit, Beschreibungen und Labels für Modellelemente in verschiedenen Sprachen zu hinterlegen. In diesem Kontext wurde eine Klasse `MultiLanguageSupport` angelegt, welche über die Klasse `IpsPlugin` erreichbar ist und einfache Methoden anbietet, die quer durch das System benötigt werden. So gibt es z.B. die Methode `String getDefaultDescription(IDescribedElement)`, welche direkt den Text der Beschreibung zurückliefert, die zur „Default-Language“ passt.

Wie in Faktor-IPS üblich wurde für die Klasse eine Eclipse Plug-in Testklasse `MultiLanguageSupportTest` im Testprojekt angelegt. Eine `Description` ist bei Faktor-IPS immer Teil eines `DescribedElement`. Der Test umfasst aber nicht nur Testfälle bezüglich Descriptions, daher wurde in der Methode `setUp()` ein `IpsObjectPartContainer` erzeugt.

```
@Override
protected void setUp() throws Exception {
	super.setUp();

	support = IpsPlugin.getMultiLanguageSupport();
	ipsProject = newIpsProject();

	productCmptType = newProductCmptType(ipsProject, "TestPolicy");
	testContainer = new TestContainer(productCmptType, "id");

	germanDescription = testContainer.getDescription(Locale.GERMAN);
	germanDescription.setText(GERMAN_DESCRIPTION);
	usDescription = testContainer.getDescription(Locale.US);
	usDescription.setText(US_DESCRIPTION);
}

private static class TestContainer
	extends AtomicIpsObjectPart implements IDescribedElement, ILabeledElement {

	private static final String LAST_RESORT_CAPTION = "Last Resort Caption";

	private static final String LAST_RESORT_PLURAL_CAPTION =
		"Last Resort Plural Caption";

	public TestContainer(IIpsObjectPartContainer parent, String id) {
		super(parent, id);
	}

	@Override
	protected Element createElement(Document doc) {
		return null;
	}

	@Override
	public String getCaption(Locale locale) {
		return "Caption for " + locale.getLanguage();
	}

	@Override
	public String getPluralCaption(Locale locale) {
		return "Plural Caption for " + locale.getLanguage();
	}

	@Override
	public String getLastResortCaption() {
		return LAST_RESORT_CAPTION;
	}

	@Override
	public String getLastResortPluralCaption() {
		return LAST_RESORT_PLURAL_CAPTION;
	}

	@Override
	public String getName() {
		return "TestContainer";
	}

	@Override
	public boolean isPluralLabelSupported() {
		return true;
	}

}
```

Da `IpsObjectPartContainer` eine abstrakte Klasse ist (bzw. `IIpsObjectPartContainer` ein Interface), benötigen wir eine Test-Implementierung (nicht zuletzt auch um die Rückgabewerte

der zu testenden Methoden zu kontrollieren). Dazu ist bereits relativ viel Quelltext notwendig, der nicht gerade zum Verständnis des Tests beiträgt.

Zum Instanziieren des Test Containers wird ein Eltern-Container benötigt. Hier verwenden wir einfach einen `ProductCmptType`. Dieser wiederum benötigt ein `IpsProject`. Das hat zur Folge, dass bei jedem Aufruf der Methode `setUp()` – also vor jedem Testfall – fleißig auf der Festplatte gearbeitet wird. In diesem Moment haben wir unseren Testfall schon von der Implementierung einiger anderer Klassen abhängig gemacht: `IpsProject`, `ProductCmptType`, `AtomicIpsObjectPart`, ...

Es folgen einige beispielhafte Testfälle.

```
public void testGetDefaultDescription() {
	assertEquals(GERMAN_DESCRIPTION, support.getDefaultDescription(testContainer));
}
```

Dieser Test überprüft, ob die Methode im "OK-Szenario" die richtige `Description` zurückliefert. Hierzu muss man bereits wissen, das die Methode `newIpsProject()` von `AbstractIpsPluginTest` das deutsche Locale als Default-Language einrichtet.

```
public void testGetDefaultDescriptionNotExsitent() {
	germanDescription.delete();
	assertEquals("", support.getDefaultDescription(testContainer));
}
```

Dieser Test überprüft, ob ein Leerstring zurückgeliefert wird, wenn keine Default-Description existiert. Dieser Test ist abhängig von der Implementierung der Methode `delete()` von `Description`. Eine Änderung dort könnte diesen Test zum Fehlschlagen bringen.

```
public void testSetDefaultDescription() {
	support.setDefaultDescription(testContainer, "foo");
	assertEquals("foo", germanDescription.getText());
}
```

Dieser Test überprüft, ob nach dem Setzen des Beschreibungstextes der Default-Description der entsprechende Beschreibungstext angepasst wurde. Der Test ist komplett von der Implementierung der Klasse `Description` abhängig.

Benchmark

Dauer bis Test startet:	ca. 18 Sekunden
Dauer der Test-Durchführung:	57,8 Sekunden
Isolationsgrad:	Niedrig
Lines of Code:	700

MultiLanguageSupportTest re-engineered

- Alle kooperierenden Klassen wurden gemockt
- Der Test wurde zu einem gewöhnlichen JUnit Test umgewandelt (wurde möglich, da kooperierende Klassen gemockt wurden)

Der Code der Methode `setUp()` sieht nun folgendermaßen aus.

```
@Override
protected void setUp() throws Exception {
	super.setUp();

	mockLocale();
	mockIpsProject();
	mockDescriptions();
	mockDescribedElement();

	support = new MultiLanguageSupport(localizationLocale);
}

private void mockLocale() {
	defaultLocale = mock(Locale.class);
}

private void mockIpsProject() {
	ISupportedLanguage defaultLanguage = mock(ISupportedLanguage.class);
	when(defaultLanguage.getLocale()).thenReturn(defaultLocale);

	ipsProjectProperties = mock(IIpsProjectProperties.class);
	when(ipsProjectProperties.getDefaultLanguage()).thenReturn(defaultLanguage);

	ipsProject = mock(IIpsProject.class);
	when(ipsProject.getProperties()).thenReturn(ipsProjectProperties);
}

private void mockDescriptions() {
	defaultDescription = mock(IDescription.class);
	when(defaultDescription.getText()).thenReturn(DEFAULT_DESCRIPTION);
}

private void mockDescribedElement() {
	describedElement = mock(IDescribedElement.class);
	when(describedElement.getIpsProject()).thenReturn(ipsProject);
	when(describedElement.getDescription(defaultLocale)).thenReturn(
		defaultDescription);
}
```

Zunächst wird das `Locale` gemockt, welches als Default-Locale gelten soll.

Im zweiten Schritt wird das `IpsProject` gemockt. Dies ist notwendig, da die Klasse `MultiLanguageSupport` auf die Methode `getIpsProject()` des `DescribedElement` zugreift, um von diesem die `SupportedLanguage` abzufragen, welche als Default eingestellt ist. Aus diesem Grund mocken wir auch die `IpsProjectProperties` des Projekts und die zurückgegebene `SupportedLanguage`.

Auch das `Description` Objekt für die Default-Description wird gemockt.

Im letzten Schritt wird schließlich noch das `DescribedElement` gemockt. Falls die Methode `getIpsProject()` an diesem Mock aufgerufen wird, geben wir das gemockte Projekt zurück. Analog geben wir die gemockte `Description` zurück, falls die Methode `getDescription(Locale)` mit dem Default-Locale aufgerufen wird.

Da wir stets Mocks verwendet haben ist jetzt sichergestellt, dass der Test von keiner Implementierung einer anderen Klasse mehr abhängt. Auch wird nicht mehr tatsächlich ein Projekt auf der Festplatte angelegt. Das entsprechende Mock ist einfach eine leere Implementierung des `IIpsProject` Interfaces.

Die Implementierung des Test Containers ist komplett weggefallen, da wir das Verhalten der Mocks über das Mocking-Framework steuern können.

Die bereits vorgestellten Testfälle sehen jetzt folgendermaßen aus.

```
public void testGetDefaultDescription() {
	assertEquals(DEFAULT_DESCRIPTION,
		support.getDefaultDescription(describedElement));
}
```

Bei dem ersten Test ist es nicht mehr notwendig zu wissen, wie die Methode `newIpsProject()` von `AbstractIpsPluginTest` funktioniert. Diese wird gar nicht mehr aufgerufen. Die Konstante `DEFAULT_DESCRIPTION` wird genau dann zurückgegeben, wenn an dem Mock für die Default-Description die Methode `getText()` aufgerufen wird (siehe `mockDescriptions()` oben).

```
public void testGetDefaultDescriptionNotExsitent() {
	when(describedElement.getDescription(defaultLocale)).thenReturn(null);
	assertEquals("", support.getDefaultDescription(describedElement));
}
```

Um die Nicht-Existenz der Default-Description zu simulieren, weißen wir das Mock des `DescribedElement` einfach an, `null` zurückzugeben, falls die Description für das Default-Locale angefordert wird. Die Abhängigkeit zur Methode `delete()` von `Description` entfällt.

```
public void testSetDefaultDescription() {
	String newDescription = "foo";
	support.setDefaultDescription(describedElement, newDescription);
	Mockito.verify(defaultDescription).setText(newDescription);
}
```

Beim Test zum Setzen des Textes der Default-Description sind wir nicht länger auf die Implementierung der Klasse `Description` angewiesen. Wir verifizieren nur noch, dass am Mock für die Default-Description die Methode `setText(String)` mit dem gewünschten Text aufgerufen wurde. Da wir hier lediglich die Interaktion mit der Klasse `Description` testen wollen

und nicht die Veränderung des Zustands des Objekts der Klasse `Description`, spricht man auch von ***Behaviour Verification*** (= `verify(...)`). Dies steht im Gegensatz zu ***State Verification*** (= `assertEquals(...)`).

Benchmark der überarbeiteten Version

Dauer bis Test startet:	ca. 1 Sekunde	(vormals ca. 18 Sekunden)
Dauer der Test-Durchführung:	2,6 Sekunden	(vormals 57,8 Sekunden)
Isolationsgrad:	Maximal	(vormals Niedrig)
Lines of Code:	600	(vormals 700)

Vorschlag für Faktor-IPS Test-Konzept

- Alle Unit Tests kommen in das **test** Verzeichnis des entsprechenden Projektes als gewöhnliche JUnit Tests (-> Problem: Test-spezifische Warning-Einstellungen)
- Kooperierende Klassen werden wie im gezeigten Beispiel gemockt
- Als Mocking-Framework sticht Mockito besonders heraus, da es ein besonders einfaches und intuitives API bietet
- Mit dem Erweiterungs-API PowerMockito können auch statische Methoden und finale Klassen gemockt werden. Das API bietet zudem noch viele andere Extras – es wird sehr schnell unverzichtbar. PowerMock ist mit JUnit3 aber nur umständlich einzusetzen (-> Upgrade auf JUnit4 wäre angebracht, auch weil alle Beispiele auf JUnit4 basieren)
- In den Test-Projekten finden sich weiterhin Eclipse Plug-in Tests. Diese sind aber ab jetzt als Integrationstests zu sehen. Ein Integrationstest wäre z.B. für die Refactoring-Unterstützung ein Modell mit den tatsächlichen Faktor-IPS Objekten anzulegen, dort Refactorings anzuwenden und nachher die Objekte auf die entsprechenden Änderungen hin zu testen.
- Man sollte evaluieren, ob es sich nicht lohnen würde zusätzlich UI Tests anzulegen
- Man sollte sich noch einmal über den Test-First Ansatz (TDD) Gedanken machen – ich denke die konsequente Anwendung würde die Effizienz der Entwickler und die Qualität des Codes um den Faktor Zehn ;-) verbessern. Es macht aber nur Sinn wenn es auch konsequent immer und von jedem Entwickler angewendet wird (-> 1 Tag im Büro Schulung?)
- Es stellt sich natürlich die Frage, was mit den existierenden Tests geschehen soll. Diese müssten fast durch neue Tests ersetzt werden, sobald ein Entwickler etwas an einer Klasse ändert (-> das wird teuer, aber wenn sich nichts verändert wird später alles nur noch teurer)

Ein etwas komplizierteres Beispiel

Da in Vorabdiskussionen bereits anklang, dass Zweifel darüber existieren, ob man immer so einfach kooperierende Klassen mocken kann, habe ich hier noch ein Beispiel angefügt, bei dem es etwas mehr zur Sache geht als beim ziemlich einfachen `MultiLanguageSupportTest`. Dies soll beweisen, dass man diesbezüglich eigentlich mit jeder Situation fertig werden kann.

Getestet werden soll die Implementierung der Methode `validateThis(...)` von `EnumLiteralNameAttributeValue`. In diesem Beispiel existiert der Code bereits bevor wir den Test schreiben – so sollte es in Wirklichkeit natürlich genau nicht sein.

```
@Override
protected void validateThis(MessageList list, IIpsProject ipsProject
        throws CoreException {

        super.validateThis(list, ipsProject);

        if (getValue() != null) {
                char[] charArray = getValue().toCharArray();
                if (charArray.length > 0 && Character.isDigit(charArray[0])) {
                        String text =
                                NLS.bind(
                                        Messages.EnumLiteralNameAttributeValue_ValueIsNumber,
                                                getValue());
                        Message msg = new
                                Message(MSGCODE_ENUM_LITERAL_NAME_ATTRIBUTE_VALUE_IS_NUMBER,
                                text, Message.ERROR, this, PROPERTY_VALUE);
                        list.add(msg);
                }
        }
}
```

Die Methode ruft zunächst die Super-Implementierung auf. Dann wird geprüft ob ein Value gesetzt ist. Ist dies der Fall wird überprüft, ob der Wert mindestens 1 Zeichen lang ist. Wenn dem so ist, wird noch geprüft, ob das erste Zeichen eine Zahl ist. Ist das erste Zeichen eine Zahl, wird eine Validierungs-Message in die per Parameter übergebene `MessageList` eingefügt.

Schreiben wir nun einen Test für diese Methode, könnte dieser zunächst wie im Folgenden abgedruckt aussehen.

```
public class EnumLiteralNameAttributeValueTest extends TestCase {

	private EnumLiteralNameAttributeValue literalAttributeValue;

	@Override
	protected void setUp() throws Exception {
		super.setUp();

		IEnumValue enumValue = mock(IEnumValue.class);
		literalAttributeValue = new EnumLiteralNameAttributeValue(
			enumValue, "AnyID");
	}

	public void testValidateThisStartsWithNumber() throws CoreException {
		literalAttributeValue.setValue("123LITERAL");

		MessageList validationMsgList = new MessageList();
		literalAttributeValue.validateThis(
			validationMsgList, mock(IIpsProject.class));

		assertEquals(1, validationMsgList.size());
		assertNotNull(validationMsgList.getMessageByCode(
IEnumLiteralNameAttributeValue.MSGCODE_ENUM_LITERAL_NAME_ATTRIBUTE_VALUE_IS_NUMBER));
	}
}
```

In der `setUp()` Methode erzeugen wir ein Objekt der zu testenden Klasse `EnumLiteralNameAttributeValue`. Als Parent übergeben wir ein Mock des Typs `IEnumValue`.

In der Test-Methode setzen wir einen illegalen Wert und rufen die zu testende Methode `validateThis(...)` auf. Die Klasse `MessageList` wurde hier nicht gemockt. Das ist aber bereits eine große Ausnahme die gemacht wurde, weil es sich hier lediglich um eine Utility-Klasse handelt – irgendwo muss man die Grenze ziehen, man würde z.B. auch nicht `java.lang.String` mocken.

Führen wir den Test aus bekommen wir eine `NullPointerException`. Wir stellen fest, dass die Super-Implementierung der Methode `setValue(...)` auf die Methode `getEnumValueContainer()` des Parent `IEnumValue` zugreifen muss. Hier sieht man schön, wie Vererbung das Konzept der Kapselung kaputt macht – das macht das Testen an dieser Stelle natürlich auch etwas schwieriger. Das Problem hat man beim Testen ohne Mocks aber genauso. Mit PowerMock besteht zwar die Möglichkeit, alle Methoden der Super-Klasse zu unterdrücken, aber wir sind ja auf das Setzen des Value-Feldes angewiesen. Ein einzelner super – Call lässt sich übrigens nicht unterdrücken (die Entwickler von PowerMock sind drüber ;-))

Die Super-Implementierung führt zusätzlich noch einen Cast auf `EnumValueContainer` durch (weil eine interne Methode aufgerufen wird, die nicht im Published API erscheinen soll).

Um das Problem zu lösen mocken wir den Aufruf `getEnumValueContainer()` des Mocks für `IEnumValue` und geben ein weiteres Mock – diesmal vom Typ `EnumValueContainer` – zurück.

```
public class EnumLiteralNameAttributeValueTest extends TestCase {

	private EnumLiteralNameAttributeValue literalAttributeValue;

	private IEnumValue enumValue;

	@Override
	protected void setUp() throws Exception {
		super.setUp();

		enumValue = mock(IEnumValue.class);
		literalAttributeValue = new EnumLiteralNameAttributeValue(
			enumValue, "AnyID");
	}

	public void testValidateThisStartsWithNumber() throws CoreException {
		when(enumValue.getEnumValueContainer()).thenReturn(
			mock(EnumValueContainer.class));

		literalAttributeValue.setValue("123LITERAL");

		MessageList validationMsgList = new MessageList();
		literalAttributeValue.validateThis(
			validationMsgList, mock(IIpsProject.class));

		assertEquals(1, validationMsgList.size());
		assertNotNull(validationMsgList.getMessageByCode(
IEnumLiteralNameAttributeValue.MSGCODE_ENUM_LITERAL_NAME_ATTRIBUTE_VALUE_IS_NUMBER));
	}
}
```

Führen wir den Test nun aus, bekommen wir einen `ExceptionInitializerError`. Dies liegt daran, dass die Super-Implementierung von `setValue(...)` die Methode `valueChanged(...)` aufruft und diese wiederum noch viele weitere Methoden. Irgendwann landen wir beim `IpsPlugin`, wo der Fehler schließlich aus irgendwelchen Gründen im Kontext eines Static Initializers auftritt.

PowerMock bietet verschiedene Möglichkeiten an, um das Problem in den Griff zu bekommen. Wir könnten z.B. auf den Aufruf `setValue(...)` komplett verzichten und den Wert des privaten Value-Feldes mit PowerMock setzen (Bypass Encapsulation). Wir könnten auch den Static Initializer mit PowerMock unterdrücken. Wir möchten aber unseren Test so gut wie möglich von anderen Klassen isolieren, daher gefällt es uns gar nicht, wenn eine so lange Kette von Methoden die Hierarchie hinauf augerufen wird. Wir entscheiden uns deswegen die Methode `valueChanged(...)` von `IpsObjectPartContainer` zu unterdrücken.

Die zusätzliche Funktionalität, die von PowerMock gewährleistet wird, macht natürlich massiv Gebrauch von Reflection und Byte-Code Modifikation. Um diese Features für den Test freizuschalten müssen wir die Annotation `@RunWith(PowerMockRunner.class)` an die Testklasse schreiben (mit JUnit3 müsste man dafür eine TestSuite anlegen -> umständlich). Außerdem müssen wir PowerMock sagen, dass er die Klasse `IpsObjectPartContainer` für die Byte-Code Manipulation präparieren soll. Das geht mit der Annotation

`@PrepareForTest(IpsObjectPartContainer.class)`, die wir entweder auf Klassen-Ebene oder auf Methoden-Ebene anwenden können. Wir packen hier jetzt unsere schärfsten Geschütze aus. Man sollte die Zusatz-Funkionalitäten von PowerMock immer nur dann verwenden, wenn es keine andere Möglichkeit gibt (die Performance-Kosten für die Byte-Code Manipulation sind nicht zu vernachlässigen und können sich schnell summieren). In der Test-Methode weißen wir PowerMock schließlich an, die problematische Methode zu unterdrücken.

```
@RunWith(PowerMockRunner.class)
public class EnumLiteralNameAttributeValueTest extends TestCase {

	private EnumLiteralNameAttributeValue literalAttributeValue;

	private IEnumValue enumValue;

	@Override
	protected void setUp() throws Exception {
		super.setUp();

		enumValue = mock(IEnumValue.class);
		literalAttributeValue = new EnumLiteralNameAttributeValue(
			enumValue, "AnyID");
	}

	@PrepareForTest(IpsObjectPartContainer.class)
	public void testValidateThisStartsWithNumber() throws CoreException {
		when(enumValue.getEnumValueContainer()).thenReturn(
			mock(EnumValueContainer.class));
		suppress(method(IpsObjectPartContainer.class, "valueChanged",
			Object.class, Object.class));

		literalAttributeValue.setValue("123LITERAL");

		MessageList validationMsgList = new MessageList();
		literalAttributeValue.validateThis(
			validationMsgList, mock(IIpsProject.class));

		assertEquals(1, validationMsgList.size());
		assertNotNull(validationMsgList.getMessageByCode(
IEnumLiteralNameAttributeValue.MSGCODE_ENUM_LITERAL_NAME_ATTRIBUTE_VALUE_IS_NUMBER));
	}
}
```

Jetzt läuft der Test wie erwartet durch. Das Beispiel sollte zum Einen zeigen, dass man so gut wie jede Situation mit dem Mocking-Ansatz lösen kann. Zum Anderen demonstriert das Beispiel auch die Mächtigkeit von PowerMock und zeigt, warum PowerMock eigentlich unverzichtbar ist.

Abschließend betrachtet wäre es hier wahrscheinlich besser gewesen, das Value-Feld anstatt über `setValue(...)` wirklich mit Hilfe von PowerMock zu setzen, da dies eine höhere Test-Isolation bringen würde.

Man könnte die Frage stellen, ob das ganze Methoden-Unterdrücken und Bytecode-Modifzieren nicht unser Programm verändert, das wir doch eigentlich testen wollen. Wir wollen hier allerdings ja nur die Methode `validateThis(...)` von `EnumLiteralNameAttributeValue` testen und

möglichst nichts anderes. Das Ziel ist es, diesen Code zu isolieren. Dazu ist es manchmal notwendig und durchaus legal, umgebenden Code "abzuholzen".