Mit Tests zurück in die Zukunft

von Claas Thiele

im Oktober 2009

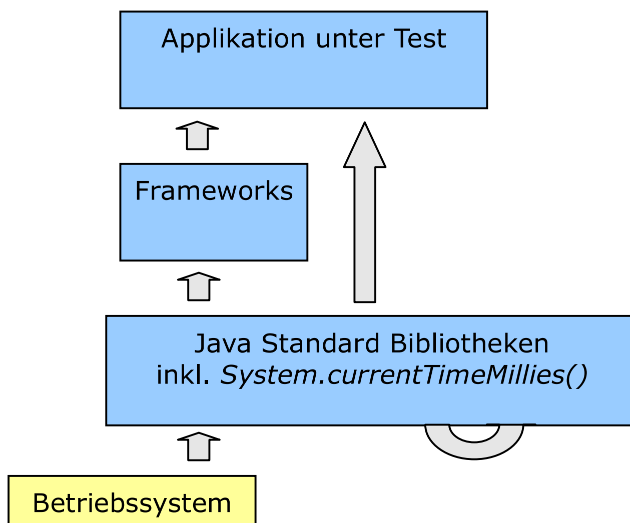
Eigentlich ist Testen ja eine einfache Sache: Man isoliert den zu testenden Teil mit *Mocks*, schafft sich ein *Testfixture* und los geht's.
Was aber, wenn das Verhalten des Systems, welches unsere zu testende Komponente umgibt, nur teilweise bekannt ist? Dann basiert ein Mock nur auf Annahmen.
Umso wichtiger wird der *Integrationstest*. Er muss die Funktionsfähigkeit unserer Komponente unter realen Bedingungen sicherstellen. Mit jeder Veränderung an unserer Komponente oder dem umgebenden System sollte ein *Regressionstest* durchgeführt werden. Verhält sich das System anders als im Mock angenommen, so muss korrigiert werden: unsere Komponente und natürlich den Mock.

Das konkrete Problem

Getestet werden muss ein über längerer Zeit andauernder Prozess, bei dem die Zeit auch noch direkt in Berechnungen und Entscheidungen einfließt.
Konkret wollen wir eine Erweiterung, einen *Listener* für das *Issue-Tracking-System „JIRA“* der Firma *Atlassian* erstellen. Dieser Listener reagiert auf Ereignisse aus dem System und soll diese Ereignisse protokollieren.
Die zeitliche Abfolge und der zeitliche Abstand der Ereignisse spielt dabei eine wesentliche Rolle.

Die klassische Lösung

Ein Mock setzt immer an der Schnittstelle an, die die Daten in das zu testende System übergibt und ersetzt das daten-liefernde System. Für die üblichen Verdächtigen wie Dateien, Datenbanken oder Webservices gibt es bereits Lösungen, aber für die Zeit?



Datenfluss für die Zeit in einer Java-Anwendung

Um ein System oder eine Komponente ersetzen zu können, brauchen wir erst einmal ein frei zugängliches *(Java) Interface*. Dies scheint für die Zeit nicht zu existieren. Aber was nicht da ist, kann man ja erschaffen. Wir könnten einen *TimeService* bereitstellen, der jedwede Zeit liefert, die in unserer Ap-

plikation verwendet wird. Diesen *TimeService* kann man dann auf dem üblichen Weg mocken. Ein solches Vorgehen würde man wohl als „Test Driven Design" bezeichnen.
In unserem konkreten Fall ist das nicht möglich, da uns der Zugriff auf das System verwehrt ist, wir haben nur Zugriff auf die von uns erstellte Erweiterung. Aber das Zusammenspiel dieser Erweiterung mit dem System ist gerade der für den Test interessante Punkt. Also muss eine andere Lösung her.

Die nahe liegende, aber fatale Lösung

Wenn wir die Zeit für unsere Applikation nicht beeinflussen können, warum verstellen wir die Zeit nicht gleich für das gesamte Betriebssystem? Dies ist technisch sicherlich möglich, aber würde fatale Folgen auf andere zeitabhängige Prozesse des Systems haben: Redo-Logs der Datenbanken können durcheinander geraten, Backup-Prozesse ebenso, Netzwerk-Dateisysteme oder Timeout-Prozesse im Allgemeinen sowie Log-Rotation geraten aus dem Tritt. Von diesem Vorhaben sollten wir also Abstand nehmen.

Die solide Lösung

Wenn wir uns mit klassischen, programmiertechnischen Methoden nicht in den Datenfluss der Zeit schalten können, müssen stärkere Geschütze aufgefahren werden. Die Verwendung *Aspektorientierter Programmierung* für das Mocken ist nicht neu, aber ist sie auch hier anwendbar?
Ja, es ist möglich, aber wir werden einige Hürden umschiffen müssen:
Bei der näheren Analyse des Problems fällt auf, dass die entscheidende Methode, `java.lang.System.currentTimeMillies()` natürlich einer Standard *Core-Klasse* angehört und zudem noch eine *native Methode* ist. Das heißt, ihre Implementierung liegt nicht in *Java Bytecode* vor und ist somit für gängige AOP Frameworks nicht erreichbar.
Um die Implikationen unseres Vorhabens besser verstehen zu können, führen wir uns noch einmal einige Grundlagen der AOP vor Augen. Für die konkrete Ausführung wurde hier *AspectJ* gewählt.
AOP verändert meist den Bytecode existierender Java Klassen. Dies kann auf unterschiedliche Art und Weise geschehen. Man unterscheidet *static, loadtime und runtime weaving*. Loadtime und runtime weaving sind elegant, da sie seit Java5 lediglich den Start der JVM mit einem zusätzlichen Parameter (-

javaagent:) erfordern. Weitere Modifikationen an der Laufzeitumgebung sind nicht nötig. Sie benötigen allerdings einen modifizierten Classloader, um den Bytecode der zu ladenden Klassen verändern zu können. Der Classloader für Standard Core Klassen wie java.lang.System lässt sich allerdings, schon aus Sicherheitsgründen, nicht so einfach modifizieren. Daher kommt hier nur statisches Weaving in Frage. Das statische Weaving verändert den Bytecode, bevor er auf den Klassenpfad gestellt wird.
In unserem Fall heißt das, dass das *rt.jar* des JRE verändert wird und dieses veränderte JAR das Original ersetzen muss. Das Verändern (Weaving) übernimmt das Tool *ajc*, das möglichst minimal-invasive Ersetzen des Originals erreichen wir dann durch den JVM Parameter -Xbootclasspath/p:. Dieser Parameter erlaubt es uns, Klassen zuvorderst auf den Klassenpfad zu legen, so dass sie beim *Bootstrapping* der JVM als erstes geladen werden und die Originalklassen nicht mehr zum Zuge kommen:

```
java
  -Xbootclasspath/p:timejump_rt.jar:aspectjrt.jar
  -Dorg.ct42.timejump=3600000
  org.ct42.TimeJumpTest
```

Es wird das modifizierte *rt.jar* und die *AspectJ* Runtime auf den Bootclasspath gelegt. Mit dem Setzen des System Properties reisen wir eine Stunde in die Zukunft und führen unser kleines Testprogramm aus.
Die *AspectJ* Implementierung unseres Aspektes sieht wie folgt aus:

```
package lib;

public aspect TimeJump {

        long around(): call(long System.currentTimeMillis()) {

                long timeJump = Long.getLong("org.ct42.timejump", 0L).longValue();

                long timestamp = proceed();

                return timestamp + timeJump;

        }

}
```

Die Implementierung ist recht schlicht gehalten. Dies hat gute Gründe, denn

der Bootstrapping-Prozess der JVM ist ein sensibler Prozess: Sollten wir zum Beispiel versuchen, eine Log-Ausgabe über System.out abzusetzen, so werden wir feststellen, dass dies mit einer NullpointerException quittiert wird. Die Methode `System.currentTimeMillies()` wird nämlich sehr früh im Bootstrapping Prozess angesprungen, zu einer Zeit, zu der noch nicht einmal die Standard IN/OUT Streams initialisiert sind. Den Zeitsprung steuern wir über ein System Property, welches die Differenz zur Realzeit in Millisekunden angibt.
Unser kleines Testprogramm sieht wie folgt aus:

```
package org.ct42;

import java.text.SimpleDateFormat;
import java.util.Date;

public class TimeJumpTest {
	public static void main(String[] args) {
		SimpleDateFormat formatter = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
		Date today = new Date();
		System.out.println( formatter.format(today) );
		System.out.println( formatter.format( new Date( System.currentTimeMillis() ) ) );
	}
}
```

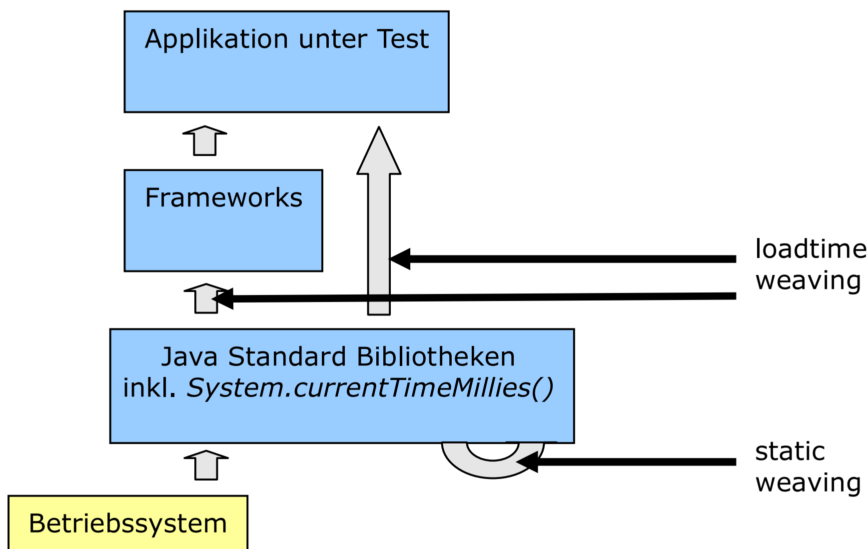
Bringen wir es zur Ausführung, werden wir feststellen müssen, dass die erste Log-Ausgabe mit korrigierter Zeit erfolgt, die zweite allerdings nicht. Die Erklärung dafür führt uns wieder ein wenig in die Tiefen der AOP:
Wie schon erwähnt, ist die Methode `currentTimeMillies()` eine *native* Methode und kann nicht durch Weaving verändert werden. Es ist also nicht möglich, den Rückgabewert der Methode zu verändern, bevor er diese verlässt. Stattdessen müssen wir jeden Aufruf der Methode verändern und den Rückgabewert anpassen, bevor er im Laufe des Programms weiter verwendet wird. Dies erledigt der around() Aspekt für uns.
Allerdings haben wir ja nur ein statisches Weaving angesetzt und damit nur das *rt.jar* verändert. Unsere Testklasse verwendet noch den original Rückgabewert. Dies würde auch auf alle anderen Aufrufe aus nachgeladenen JARs von Frameworks bzw. Bibliotheken Dritter zutreffen.
Damit wir nun nicht umständlich alle weiteren Klassen und JARs statisch weaven müssen, werden wir hier das loadtime Weaving anwenden. Dafür muss eine Konfiguration in *META-INF/aop.xml* hinterlegt werden:

```
<aspectj>
        <aspects>
                <aspect name="lib/TimeJump" />
        </aspects>
</aspectj>
```

Unser JVM Aufruf sieht dann so aus:

```
java
  -Xbootclasspath/p:timejump_rt.jar:aspectjrt.jar:timejumpaspects.jar
  javaagent:aspectjweaver.jar
  -Dorg.ct42.timejump=3600000
  org.ct42.TimeJumpTest
```

Die Konfiguration legen wir praktischerweise mit in das *timejumpaspects.jar*. Nun sollten beide Log-Ausgaben einer Zeitreise unterliegen.



Weaving Punkte und Typen

Die Zeit steuern

Das obige einfache Beispiel erzwingt die Zeitreise einmalig beim Start des Programms.
Nun wollen wir unseren Test allerdings im Zeitraffer durchlaufen. Dazu müssen wir nur das Systemproperty `org.ct42.timejump` auch zur Laufzeit verändern können. Dies geschieht am besten mit Bordmitteln des umgebenden Systems. Möglich wäre eine JMX Konsole oder ähnliches. In unserem Fall bedienen wir uns eines *Jelly Scripts*, für das JIRA eine Möglichkeit der Ausführung anbietet.

Einschränkungen

Der around() Aspekt ist recht laufzeitintensiv. Da die currentTimeMillies() Methode oft auch für das Profiling herangezogen wird, sollte man bei Anwendung unseres TimeJump Aspekts Verfälschungen von Profiling-Ergebnissen erwarten dürfen.
Das loadtime Weaving ist vom Classloader abhängig, sollte die Methode currentTimeMillies() aus Code heraus angesprungen werden, der auf irgendeine Weise am Standard-Classloader vorbei geladen wurde, fehlt natürlich unser Aspekt. Das könnte zu einigem Durcheinander in der Applikation unter Test führen.
Aber auf die Zeit kann auch in anderen Systemteilen zugegriffen werden, die nicht durch den Aspekt beeinflusst sind. Da wäre zum Beispiel die Verwendung der Funktion *SYSDATE* in SQL-Statements oder Zeitangaben in Daten, die die Applikation von außen über Dateien, Datenbanken oder Webservices erreichen. Hier müssen dann die entsprechenden Mocks mit der Zeitreisenkonfiguration unseres TimeJump Aspekts synchronisiert werden.
Wir müssen auch mit Nebeneffekten rechnen. Zum Beispiel wird wohl die Websession ausgelaufen sein, nachdem wir den Zeitraffer aktiviert haben und uns weit in die Zukunft begeben.