eclipse MAGAZIN

2.13 Deutschland € 9,80 Österreich € 10,80, Schweiz sFr 19,20 S&S

eclipse MAGAZIN

www.eclipse-magazin.de

JavaFX & Eclipse

Datenträger enthält Info- und Lehrprogramme gemäß § 14 JuSchG

4 196603 209806 02

MagicTest: Visuelles Testen mit Java

It's fun! It's magic! It's testing!?

MagicTest ist ein neues Testframework für Java. Es zeichnet sich durch einen visuellen Ansatz aus, der die Unzulänglichkeiten bekannter Testframeworks wie TestNG oder JUnit behebt. So sind keine Assert-Statements notwendig, um die erwarteten Resultate zu prüfen, und negative Tests brauchen keine Hilfskonstrukte. Komplexe Objekte und große Datenmengen können einfach verarbeitet werden, der Unterhalt der Tests wird entscheidend vereinfacht, und ein übersichtlicher HTML-Report macht die Tests für Dritte nachvollziehbar. Damit wird das Testen nicht nur effizienter, sondern macht auch endlich Spaß!

von Thomas Mauch

Testen bzw. das Schreiben von Tests gehört heute genauso zum täglichen Brot eines Java-Entwicklers wie das Schreiben des eigentlichen Sourcecodes. Diese Entwicklung ist einerseits wohl der Popularität der im Java-Umfeld vorhandenen Testframeworks wie TestNG oder JUnit zu verdanken, die die automatisierte Abarbeitung der Tests erlauben. Andererseits gehen Methoden wie TDD (Test-driven Development) oder BDD (Behavior-driven Development) noch einen Schritt weiter und verbinden das Schreiben von Sourcecode und das Testen zu einem umfassenden Ansatz.

Trotzdem bleibt der Eindruck, dass die Testerei von den Entwicklern noch immer mehr als lästige Pflicht denn als gleichwertige Arbeit gesehen wird. So würden wohl die meisten Entwickler der Behauptung zustimmen, dass sie gerne Sourcecode schreiben. Bei der gleichen Frage nach dem Testen würde man aber eher Kopfschütteln ernten. Und hier wollen wir ansetzen: Weshalb schreiben wir denn nicht gerne Tests?

Zuerst gehen wir aber noch kurz auf die fundamentale Kritik ein, dass das Testen grundsätzlich (zu viel) Zeit beansprucht, die dann für das Entwickeln der eigentlichen Software fehlt. Auch wenn heute allgemein davon ausgegangen wird, dass das Erstellen und Ausführen von Unit Tests die Softwareentwicklung auf lange Sicht effizienter macht, weil dann spätere Erweiterungen oder Refactoring einfacher und vor allem fehlerfreier durchzuführen sind, hat die Aussage einen gewissen Wahrheitsgehalt. Denn die Tatsache bleibt, dass man beim Entwickeln einer Komponente langsamer sein wird, wenn neben der eigentlichen Methode auch noch Tests geschrieben werden müssen.

Aus dieser Betrachtung leiten wir nun die Hauptaufgabe des perfekten Testframeworks ab: Es muss das Erstellen und auch Unterhalten von Tests so effizient wie möglich machen – damit wir mehr Zeit für das Entwickeln der eigentlichen Software aufwenden können. Und wünschenswert wäre es natürlich auch, wenn die Testerei ein bisschen mehr Spaß machen würde. Neben dem reinen Zeitaufwand schreckt häufig auch ab, dass viele Arbeiten im Testumfeld schlicht und einfach langweilig und langwierig sind.

Testen heute

Um zu verstehen, wo die Probleme mit den heutigen Tests liegen, betrachten wir ein einfaches Beispiel, das eine *join*-Methode (Listing 1) testet und aus einem positiven und einem negativem Test besteht. Wenn dies mit einem Framework wie TestNG realisiert wird, sieht der Code wie in Listing 2 aus.

Bereits auf den ersten Blick fällt auf, dass positive und negative Tests unterschiedlich formuliert werden müssen:

- Bei positiven Tests geschieht die Überprüfung der Korrektheit mit Assertions. Das bedeutet, dass das erwartete Resultat in den Testcode integriert werden muss.
- Bei negativen Tests, wo eine Exception erwartet wird, muss diese explizit mit *try-catch* abgefangen werden, um zu verhindern, dass das Testprogramm vorzeitig beendet wird.

Der Grund, wieso die Tests so unterschiedlich formuliert werden müssen, hängt mit der Definition zusammen, wann ein Test bei der Ausführung als korrekt beurteilt wird: Ein Test gilt dann als korrekt, wenn er abgearbeitet wird, ohne dass eine Assertion fehlschlägt bzw. ohne dass

eine nicht erwartete Exception auftritt. Der Test ist damit – und das ist der große Vorteil der Testframeworks – automatisch ausführbar. Das Aussehen des Testcodes wird aber offensichtlich weitgehend von den Möglichkeiten der Sprache bestimmt, denn in einem Testplan unterscheiden sich die beiden Testfälle nicht grundlegend (Tabelle 1).

Um diese umständlichen negativen Tests in den Griff zu bekommen, wurden verschiedene Anstrengungen unternommen:

- Beide Frameworks bieten die Möglichkeit, negative Tests in separate Testmethoden auszulagern, wobei dann die Annotation *@Test* über das Attribut *expectedExceptions* bzw. *expected* verfügen muss. Dann muss aber für jeden negativen Test eine eigene Testmethode verwendet werden.
- JUnit hat so genannte Rules eingefügt, wobei eine Standardimplementierung der *ErrorCollector* ist – ein Art Soft-Assert [1].
- Für TestNG wird eine *Soft Assert*-Klasse zur Diskussion gestellt [2].

Aber welchen Ansatz man auch immer wählt, der Testcode bleibt durch die Verwendung der *Assert*-Statements oder anderer Hilfsmittel sehr schwerfällig.

Testen gestern

Im Zeitalter vor den Testframeworks gab es grundsätzlich zwei Möglichkeiten, die erstellten Programme auf Ebene der Unit Tests zu testen:

- Man überprüfte die Korrektheit mit dem Debugger, indem man die vom Programm unterhaltenen Variablen und Zustände während der Abarbeitung prüfte.
- Man fügte an den relevanten Stellen *Trace*-Statements ein, die den Ablauf des Programms dokumentierten. Nach Beendigung des Programmlaufs prüfte man den erzeugten Output auf Korrektheit. Und wenn die Tests abgeschlossen waren, musste man nur noch daran denken, alle *Trace*-Statements wieder zu entfernen...

Beide Ansätze beruhen auf dem visuellen Vergleich durch den Entwickler. Das hat den Vorteil, dass kein Testcode geschrieben werden muss, der produktive Code reicht zum Testen. Der große Nachteil ist aber, dass das Testen so nicht automatisierbar ist. Wenn also ein Test nach einer Änderung erneut durchgeführt werden musste, musste der Entwickler die Testausführung noch einmal mit exakt gleicher Präzision prüfen – und das wird wohl mit jeder Testiteration unwahrscheinlicher.

MagicTest

MagicTest macht nun diesen visuellen Ansatz automatisierbar. Die Idee ist, dass der Entwickler die visuelle Prüfung nur beim ersten Mal manuell durchführen muss – anschließend soll dies das Testframework automatisch machen. Dazu muss das Testprogramm während der Ausführung die notwendigen Informationen für diese Prüfung ausgeben. Welche Informationen das sind, entnehmen wir unserem Testplan (Tabelle 1). Damit erhalten wir für den ersten Aufruf der Methode *join* in unserem Beispiel den Code in Listing 3.

Allerdings sieht dies noch deutlich umständlicher aus als unser ursprünglicher Testcode. Aber wir sind fast am Ziel: Mit ein bisschen Instrumentierungsmagie auf Bytecode-Ebene können wir auf all den Boilerplate-Code im Test verzichten und müssen nur noch die eigentlichen Methodenaufrufe schreiben:

```
@Trace
public static void testJoin() {
  join("Hello", "MagicTest");
  join("Hello", null);
}
```

Die Annotation *@Trace* weist MagicTest an, die zu testende Methode zu instrumentieren. Welches die zu

Listing 1

```
public static String join(String s1, String s2) {
  if (s1 == null || s2 == null) {
    throw new IllegalArgumentException("Null strings are not allowed");
  }
  return s1 + " " + s2;
}
```

Listing 2

```
@Test
public void testJoin() {
  assertEquals("Hello MagicTest", join("Hello", "MagicTest"));
  try {
    join("Hello", null);
    fail();
  }
  catch (IllegalArgumentException e) {
  }
}
```

Listing 3

```
try {
  printDescription("join");
  printParameters("Hello", "MagicTest");
  String result = join("Hello", "MagicTest")
  printResult(result);
}
catch (Throwable t) {
  printError(t);
}
```

Methode	Argumente	Resultat
join	"Hello", "MagicTest"	"Hello MagicTest"
join	"Hello", null	IllegalArgumentException

Tabelle 1: Testplan

Abb. 1: Report nach erstmaliger Ausführung des Tests

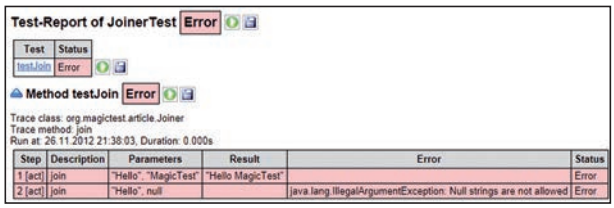
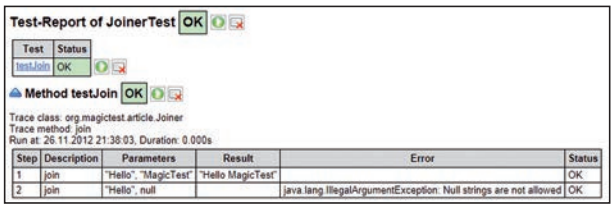


Abb. 2: Report nach dem Speichern des aktuellen Outputs



testende Methode ist, wird per Default über Namenskonventionen ermittelt: Heißt die Testmethode also *testJoin*, wird nach einer zu testenden Methode mit Namen *join* gesucht, und diese Aufrufe werden instrumentiert. Diese Konvention kann aber auch mit Annotationsparametern übersteuert werden.

Wenn wir den Test nun ausführen, schickt der instrumentierte Testcode die gewünschten Informationen an das Testframework, das diese sammelt und als Datei speichert. Diese Referenzdateien können dann mit dem Sourcecode zusammen unter *Versionsverwaltung* abgelegt werden. Wir haben also folgenden Ablauf bei Ausführung eines Tests:

- Das Testprogramm gibt während der Abarbeitung Daten aus, die vom Testframework gesammelt werden.
- Nach Beendigung des Tests vergleicht das Testframework die Daten der aktuellen Ausgabe mit der gespeicherten Referenzausgabe.
- Wenn die Ausgaben übereinstimmen, ist der Test erfolgreich. Stimmen sie nicht überein oder wirft die Testmethode eine nicht abgefangene Exception (es werden nur die Exceptions automatisch abgefangen, die beim Aufruf der zu testenden Methode generiert werden), ist der Test fehlgeschlagen.

Mit dem visuellen Ansatz hat sich also auch die Definition geändert, wann ein Test als erfolgreich beurteilt wird. Wenn ein Test nicht erfolgreich war, muss der Entwickler die aktuelle Ausgabe prüfen, indem er sie mit der Referenzausgabe vergleicht, die schließlich korrekt sein sollte. In folgenden Fällen kann aber auch die Referenzausgabe falsch sein:

- Die Testmethode wurde verändert.
- Die Spezifikation der zu testenden Methode hat sich geändert, d. h. sie muss für den gleichen Aufruf ein anderes Resultat liefern.
- Es gibt noch keine Referenzausgabe, da der Test zum ersten Mal ausgeführt wird.

Auch wenn klar ist, dass die Referenzausgabe falsch ist, muss die aktuelle Ausgabe sorgfältig auf Korrektheit geprüft werden – sie könnte ja ebenfalls falsch sein! Wenn sie aber korrekt ist, kann die aktuelle Ausgabe als neue Referenzausgabe gespeichert werden. In allen anderen Fällen ist die Referenzausgabe korrekt, d. h. die zu testende Methode muss angepasst werden, damit sie das erwartete Referenzresultat liefert.

Nun wollen wir das Testprogramm erstmals mit MagicTest ausführen. Am einfachsten geht dies mit dem MagicTest-Plug-in direkt in Eclipse (Run As | MagicTest). Nach Ausführung des Tests wird uns ein HTML-Report angezeigt (**Abb. 1**).

Das sieht unserem ursprünglichen Testplan sehr ähnlich! Allerdings ist noch alles rot, was darauf hindeutet, dass der Test fehlgeschlagen ist. Das ist aber korrekt, da wir den Test zum ersten Mal ausführen und das Testframework das richtige Resultat noch gar nicht kennen kann. Es ist nun unsere Aufgabe als Entwickler, den Report visuell zu prüfen. Da er offensichtlich korrekt ist, speichern wir die aktuelle Ausgabe als Referenzausgabe. Dies können wir direkt im HTML-Report über den Save-Button tun. Anschließend erkennt MagicTest den Test als erfolgreich (**Abb. 2**).

Damit haben wir unseren ersten Test mit MagicTest geschrieben und ausgeführt. Nach diesem einfachen Beispiel wollen wir nun sehen, wie sich der Ansatz bei komplexeren Beispielen und über den gesamten Lebenszyklus der Softwareentwicklung verhält.

Komplexe Tests

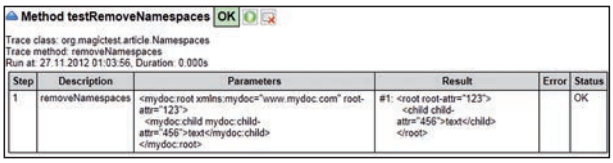
Wie bereits gesehen, müssen beim traditionellen Ansatz die erwarteten Resultate in den Testcode integriert werden. Während bei einfachen statischen Funktionen offensichtlich ist, was geprüft werden soll (nämlich das Resultat), ist das bei komplexen Objekten, auf denen Instanzmethoden aufgerufen werden, schnell nicht mehr so klar.

Wir schauen dazu folgendes Beispiel an: Ich implementiere eine Methode, die alle Namespaces von einem XML-Element entfernen soll. Was soll ich nun alles testen, damit ich sicher bin, dass meine Funktion korrekt arbeitet? Sicher muss ich prüfen, dass das übergebene Element und alle rekursiv erreichbaren Kinder keine Namespaces mehr enthalten. Gleichzeitig sollte ich aber auch sicherstellen, dass alle im übergebenen Element vorhandenen Elemente und Attribute auch nachher noch vorhanden sind – schließlich soll meine Methode ja nur die Namespaces und nicht noch ein paar Elemente oder Attribute entfernen.

Beim traditionellen Ansatz muss der Entwickler nun alle diese Fakten per Assertions prüfen. Im genannten Fall kann es gut sein, dass die vollständige Prüfung eher noch komplexer ist als das eigentliche Entfernen der Namespaces, sodass sich der Entwickler vielleicht darauf beschränkt, nur zu prüfen, dass im Resultat keine Namespaces mehr vorkommen.

Der visuelle Ansatz von MagicTest vereinfacht das Prüfen der gewünschten Eigenschaften entscheidend. Durch die Ausgabe der gewünschten Daten werden diese automatisch in den Test einbezogen. Für unser Beispiel

Abb. 3: Beispiel für einen Test mit komplexen Objekten

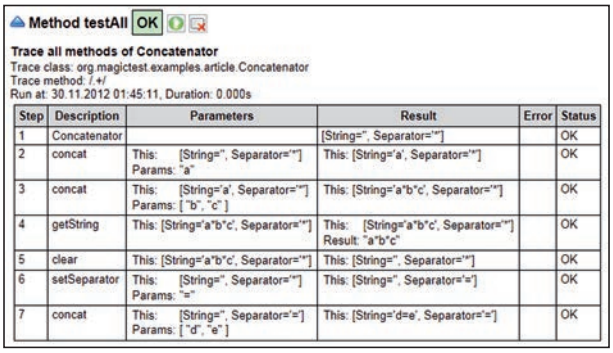


Method testRemoveNamespaces OK

Trace class: org.magictest.article.Namespaces
Trace method: removeNamespaces
Run at: 27.11.2012 01:03:56, Duration: 0.000s

Step	Description	Parameters	Result	Error	Status
1	removeNamespaces	<mydoc:root xmlns:mydoc="www.mydoc.com" root-attr="123"><mydoc:child mydoc:child-attr="456">text</mydoc:child></mydoc:root>	#1: <root root-attr="123"><child child-attr="456">text</child></root>		OK

Abb. 4: Beispiel für einen Test, der alle Methoden einer Klasse tract



Method testAll OK

Trace all methods of Concatenator
Trace class: org.magictest.examples.article.Concatenator
Trace method: /.+/
Run at: 30.11.2012 01:45:11, Duration: 0.000s

Step	Description	Parameters	Result	Error	Status
1	Concatenator		[String='', Separator='*']		OK
2	concat	This: [String='', Separator='*'] Params: "a"	This: [String='a', Separator='*']		OK
3	concat	This: [String='a', Separator='*'] Params: ["b", "c"]	This: [String='a*b*c', Separator='*']		OK
4	getString	This: [String='a*b*c', Separator='*']	This: [String='a*b*c', Separator='*'] Result: "a*b*c"		OK
5	clear	This: [String='a*b*c', Separator='*']	This: [String='', Separator='*']		OK
6	setSeparator	This: [String='', Separator='*'] Params: "="	This: [String='', Separator='=']		OK
7	concat	This: [String='', Separator='='] Params: ["d", "e"]	This: [String='d=e', Separator='=']		OK

ist es die einfachste Lösung, das XML-Element als Text auszugeben. Normalerweise gibt MagicTest für Parameter und Resultat den Wert aus, der die *toString*-Methode liefert, mit der Definition von *Formatter* für bestimmte Objekttypen können wir dies aber übersteuern. Listing 4 zeigt den entsprechenden Testcode und **Abbildung 3** den erzeugten Report bei Ausführung des Tests.

Die Stärken des visuellen Ansatzes sind ebenfalls sichtbar, wenn große Datenmengen geprüft werden müssen. Betrachten wir als Beispiel eine Methode, die Informationen aus einer Datenbank abfragt. Um die Korrektheit der Methode zu garantieren, müssen die zurückgegebenen Daten geprüft werden. Schnell wird aber offensichtlich, dass dies mit dem Schreiben von Assertions nicht vernünftig zu bewältigen ist, da man dazu die ganzen Daten in den Testcode einbetten müsste. Um das Problem in den Griff zu bekommen, müsste man die Queries manuell oder mit einem Hilfsprogramm ausführen und die Resultate in Dateien speichern. Damit diese dann für den Vergleich verwendet werden können, könnte man eine Hilfsklasse wie *FileAssert* benutzen, die das Vergleichen von Dateiinhalten unterstützt – kurz, ein ziemlicher Aufwand. Bei der Verwendung von MagicTest hingegen sind solche Umwege nicht nötig: Das einzige, was man braucht, ist ein Formatter, um die Resultatdaten als Text auszugeben, und schon sind diese in den Testreport integriert.

Bis jetzt hat unsere Testmethode immer nur eine einzelne Methode getracet. Es kann aber auch durchaus sinnvoll sein, dies für mehrere Methoden einer Klasse gleichzeitig zu tun. In unserem nächsten Beispiel (Listing 5) wählen wir mit einer Regular Expression gleich alle Methoden einer Klasse auf einmal zum Tracen aus. Damit ist es möglich, ganze Use Cases in einer Testmethode abzubilden. Im erzeugten Report (**Abb. 4**) kann man dann die Auswirkungen der einzelnen Testschritte verfolgen.

Unterhalt von Tests

Einen Aspekt des Testens haben wir bisher außer Acht gelassen: den Unterhalt. Wie der normale Code müssen Tests nicht nur erstellt, sondern anschließend auch unterhalten werden. Unglücklicherweise bietet der traditionelle Assertion-basierte Ansatz hier keine wirkliche Unterstützung. Betrachten wir das Beispiel, das die Spezifikation einer Methode so ändert, dass nach der Änderung alle Testfälle ein anderes Resultat liefern. Das Problem ist nun, dass die erwarteten Resultate mit im Testcode gespeichert sind. Es kann also leicht sein, dass die eigentliche Änderung (dadurch, dass der Code gut strukturiert ist) nur an einer einzigen Stelle vorgenommen werden muss, dass aber anschließend noch alle in den Testmethoden enthaltenen Testresultate angepasst werden müssen. Kurz gesagt: Die eigentliche Codeänderung ist in wenigen Sekunden erledigt, die Anpassung der Tests dauert ein Vielfaches länger! Das motiviert die Entwickler natürlich nicht gerade, möglichst viele Unit Tests zu schreiben.

Ein zweites Problem ist, dass fehlgeschlagene Assertions die Abarbeitung des Tests stoppen: Wenn ich eine Anpassung vergesse oder sie falsch mache, bricht

Listing 4

```
@Formatter(outputType=OutputType.TEXT)
static String formatElement(Element elem) {
      XMLOutputter serializer = new XMLOutputter();
      serializer.setFormat(Format.getPrettyFormat());
      return serializer.outputString(elem);
}

@Trace(parameters=Trace.ALL_PARAMS, result=Trace.ALL_PARAMS)
public void testRemoveNamespaces() {
   Element elem = createElemWithNamespace();
   removeNamespaces(elem);
}
```

Listing 5

```
@Trace(traceMethod="/.+/",
   parameters = Trace.THIS|Trace.ALL_PARAMS,
   result = Trace.THIS|Trace.RESULT,
   title = "Trace all methods of Concatenator")
public static void testAll() {
   Concatenator c = new Concatenator();
   c.concat("a");
   c.concat("b", "c");
   c.getString();
   c.clear();
   c.setSeparator("=");
   c.concat("d", "e");
}

@Formatter
public static String format(Concatenator concatenator) {
   return "[String='" + concatenator.toString() + "', Separator='" +
      concatenator.getSeparator() + "']";
}
```

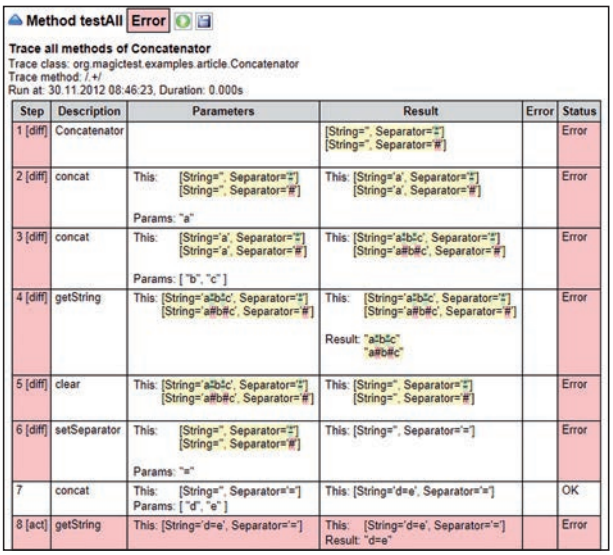
die Testmethode mit einer Exception ab, und ich weiß nicht, ob der Rest meiner Anpassungen korrekt war. Die einzige Methode, dies in Erfahrung zu bringen, ist es, den Fehler zu korrigieren und den Test erneut auszuführen – sofern mich nicht der nächste Fehler stoppt…

Auch hier kann MagicTest helfen: Nachdem die eigentliche Änderung vorgenommen wurde, wird der Test erneut ausgeführt. Die Auswirkungen der Änderungen werden nun alle übersichtlich im HTML-Report dargestellt. Entspricht die Ausgabe den Erwartungen, können alle Testresultate mit einem einzigen Mausklick auf den SAVE-Button bestätigt werden – ohne dass am Testcode eine einzige Änderung gemacht werden muss!

Damit die gemachten Änderungen einfach erkennbar sind, werden sie grafisch ausgezeichnet, wie dies von Diff-Tools bekannt ist. **Abbildung 5** zeigt deshalb nochmals einen Report für die Ausführung des Tests von Listing 5, nachdem wir in der *concat*-Methode das Trennzeichen geändert haben. Diese einfache Änderung hat Auswirkungen auf die meisten Aufrufe der zu testenden Methoden – und der Report macht dies sofort sichtbar. Ebenfalls sieht man, dass der letzte Aufruf neu zum Testprogramm hinzugekommen sein muss.

Reporting

Es gibt noch einen weiteren Kritikpunkt bei den traditionellen Tests: Die wesentlichen Informationen sind nur



Method testAll Error

Trace all methods of Concatenator
Trace class: org.magictest.examples.article.Concatenator
Trace method: /.+/
Run at: 30.11.2012 08:46:23, Duration: 0.000s

Step	Description	Parameters	Result	Error	Status
1 [diff]	Concatenator		[String='', Separator='‡'] [String='', Separator='#']		Error
2 [diff]	concat	This: [String='', Separator='‡'] [String='', Separator='#'] Params: "a"	This: [String='a', Separator='‡'] [String='a', Separator='#']		Error
3 [diff]	concat	This: [String='a', Separator='‡'] [String='a', Separator='#'] Params: ["b", "c"]	This: [String='a‡b‡c', Separator='‡'] [String='a#b#c', Separator='#']		Error
4 [diff]	getString	This: [String='a‡b‡c', Separator='‡'] [String='a#b#c', Separator='#']	This: [String='a‡b‡c', Separator='‡'] [String='a#b#c', Separator='#'] Result: "a‡b‡c" "a#b#c"		Error
5 [diff]	clear	This: [String='a‡b‡c', Separator='‡'] [String='a#b#c', Separator='#']	This: [String='', Separator='‡'] [String='', Separator='#']		Error
6 [diff]	setSeparator	This: [String='', Separator='‡'] [String='', Separator='#'] Params: "="	This: [String='', Separator='=']		Error
7	concat	This: [String='', Separator='='] Params: ["d", "e"]	This: [String='d=e', Separator='=']		OK
8 [act]	getString	This: [String='d=e', Separator='=']	This: [String='d=e', Separator='='] Result: "d=e"		Error

Abb. 5: Report, der alle Änderung als Diff anzeigt

für Entwickler zugänglich. Der Grund ist, dass die von den Testframeworks erstellten Reports zu wenig detailliert sind, da man nur sieht, welche Tests erfolgreich waren, aber eben nicht, was genau getestet wurde. Daher ist es häufig notwendig, zusätzlich zu den Tests auch noch einen separaten Testplan zu schreiben, um dies für Kollegen, Vorgesetzte oder Kunden zu dokumentieren. Hier kann der detaillierte Report von MagicTest helfen, wo für jeden einzelnen Methodenaufruf genau ersichtlich ist,

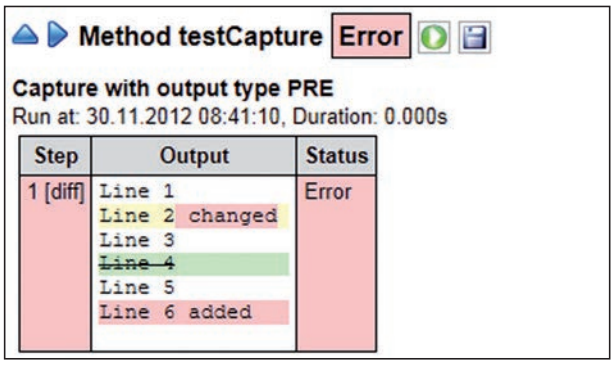


Abb. 6: Report eines @Capture-Tests mit Differenzen

was getestet wurde. Dieser Report kann natürlich auch für den Entwickler selbst hilfreich sein: wenn er später wegen laxer Programmierkonventionen beim Schreiben von Tests selber nicht mehr weiß, was er da genau testen wollte – von seinem Kollegen ganz zu schweigen.

Unit und Characterization Tests

Die Annotation *@Trace*, die wir bisher verwendet haben, ist für den Einsatz in Unit Tests ausgelegt. Der Ansatz des visuellen Vergleichs eignet sich aber ebenfalls sehr gut für Characterization Tests. Diese zwei Arten von Tests unterscheiden sich wie folgt:

- Ein Unit Test testet die Funktionalität eines einzelnen Softwaremoduls.
- Ein Characterization Test beschreibt das tatsächliche Verhalten eines existierenden Software-Packages; ein Package kann ein einzelnes Modul oder aber eine ganze Applikation umfassen.

Während Unit Tests typischerweise gleichzeitig mit dem Modul selbst geschrieben werden, werden Characterization Tests als Sicherheitsnetz eingesetzt, wenn Software, die nicht über genügend Unit Tests verfügt, erweitert oder angepasst werden muss. Sie werden dann geschrieben, um den aktuellen Stand so zu dokumentieren, dass Fehler mit Regressionstests entdeckt werden können.

MagicTest unterstützt Characterization Tests mit der Annotation *@Capture*. Damit werden die vom Programm auf einen Ausgabekanal wie *System.out* geschriebenen Daten gesammelt und als aktuelle Ausgabe gespeichert. Listing 6 zeigt ein zugegebenermaßen nicht sehr sinnvolles Minibeispiel, in dem das Testprogramm einfach ein paar statische Texte ausgibt. **Abbildung 6** lässt aber die Möglichkeiten erahnen, die sich dahinter verbergen, indem MagicTest nun allfällige Änderungen in der Ausgabe erkennt und markiert. Um dies für existierende Software zu nutzen, genügt es also, an den wichtigen Stellen *Trace*-Aufrufe einzufügen, die die für den Test relevanten Daten ausgeben.

Listing 6

```
@Capture(outputType=OutputType.PRE,
    title="Capture with output type PRE")
public static void testCapture() {
  System.out.println("Line 1");
  System.out.println("Line 2");
  System.out.println("Line 3");
  System.out.println("Line 4");
  System.out.println("Line 5");
}
```

Fazit

Wir haben gesehen, dass der traditionelle Assertion-basierte Ansatz beim Testen zwar den Vorteil der Automatisierung bietet, ansonsten aber auch etliche Probleme mit sich bringt. Diese Unzulänglichkeiten behebt MagicTest mit einem neuartigen visuellen Ansatz. Damit wird das Testen effizienter und macht auch wieder mehr Spaß. MagicTest bietet in folgenden Bereichen Verbesserungen:

- **Positive Tests**: Es sind keine Assert-Statements mehr nötig, um das erwartete Resultat in den Testcode zu integrieren. Das reduziert nicht nur die Tipparbeit, sondern erleichtert auch den Unterhalt der Tests.
- **Negative Tests**: Negative Tests sind so einfach wie positive Tests, d. h. es werden keine Hilfskonstrukte wie *try-catch* oder separate Testmethoden mehr benötigt.
- **Komplexe Tests**: Komplexe Objekte und große Datenmengen können einfach durch Ausgabe der relevanten Informationen getestet werden.
- **Unterhalt der Tests**: Die Auswirkungen aller Änderungen werden übersichtlich im Report dargestellt und können visuell geprüft und bestätigt werden.
- **Reporting**: Der HTML-Report enthält die Details zu jedem Methodenaufruf. Damit ist auch für Dritte nachvollziehbar, was genau getestet wurde.

MagicTest kann auf verschiedene Arten verwendet werden. Die komfortabelste Variante während der Entwicklung ist sicherlich der Einsatz des Eclipse-Plugins. Daneben stehen Kommandozeilenprogramme zur Auswahl, die z. B. für die Einbindung in Continuous-Integration-Systeme verwendet werden können. Eine Integration mit TestNG ermöglicht die Weiterverwendung von bestehenden Testsuiten. Weitere Informationen zu MagicTest inklusive Downloads findet man auf der Website [3]. Für erste Gehversuche startet man am besten mit dem dort vorhandenen Beispielprojekt und kann so in wenigen Minuten die neue Magie des Testens genießen.

Thomas Mauch arbeitet als Software Architect bei Swisslog in Buchs, Schweiz. Er interessiert sich seit den Zeiten des C64 für alle Aspekte der Softwareentwicklung.

Links & Literatur

[1] https://github.com/KentBeck/junit/wiki/Rules

[2] http://beust.com/weblog/2012/07/29/reinventing-assertions/

[3] http://www.magicwerk.org/magictest