Minimal? – Vom Testen eines Textsatzsystems

Gerd Neugebauer

Ein System wie $\varepsilon_{\mathcal{X}}$TEX muss getestet werden. Dabei kommt die Kunst der Minimalbeispiele zum Einsatz. Diese treten hier in der Form von Testfällen auf. $\varepsilon_{\mathcal{X}}$TEX soll durch Testfälle so abgesichert werden, dass während der Weiterentwicklung keine ungewollten Veränderungen an den implementierten Funktionalitäten auftreten.

Einleitung

In Mailing-Listen wie TEX-D-L und News-Gruppen wie de.comp.text.tex wird auf Fragen immer wieder mit einem Hinweis auf ein Minimalbeispiel geantwortet. Ein Minimalbeispiel dient dabei dazu, die Essenz des Effekts, den man zeigen will herauszuarbeiten. Das hilft auf der einen Seite zu verstehen, welche Elemente daran beteiligt sind – wenn man noch etwas weglassen kann, dann ist es noch nicht minimal. Auf der anderen Seite erlaubt es anderen, den Effekt zu reproduzieren. Damit kann nachgewiesen werden, dass der Effekt immer noch auftritt, auch wenn ein anders TEX-System benutzt wird.

Das Konzept des Minimalbeispiels ist auch noch an anderer Stelle relevant, nämlich in der Software-Entwicklung. Hier kommt es in der Form von Testfällen vor. Eine Software sollte normalerweise eine Reihe von Anforderungen erfüllen. Um dies zu testen werden die Anforderungen in Testfälle gegossen. Damit hat man eine – im besten Fall – große Liste von Testfällen, die es gestattet detailliert abzuschätzen, in welchem Maß die Anforderungen erfüllt sind.

Im Gegensatz dazu steht ein einzelner, großer Testfall. Entweder er geht durch, oder eben nicht. Hier ist eine detaillierte Bewertung schwieriger. Auch ist es schwieriger, einen Fehler zu lokalisieren und gegebenenfalls zu beheben.

Während der Entwicklung können Testfälle dabei helfen das Funktionieren von Code zu beurteilen. Oftmals wird Code geändert – sei es, dass er um neue Funktionalität erweitert wird, oder sei es, dass er verbessert wird. Das wird heute oft als Refactoring bezeichnet [3]. Zum Extrem wird dies beim Test-Driven Development getrieben [1, 6]. Hier werden als erstes die Testfälle erstellt und danach die Software implementiert.

In $\mathcal{NTS}$ wurde im Gegensatz zu dem bisher gesagten im Wesentlichen ein großer Testfall benutzt – das TEX Book [5]. Das zeigt zwar viele Eigenschaften von TEX, aber eben nicht alle. Und wenn dann ein Fehler auftritt kann man nur schwer beurteilen, wie schwerwiegend er ist.

Auch der Trip-Test von TEX testet einige Eigenschaften ab. Dies sind vor allem das Verhalten von TEX in Randbereichen des Einsatzes. Es ist denkbar ein – relativ einfaches – Programm zu schreiben, das den Trip-Test bestehen würde, aber damit noch bei weitem nicht geeignet wäre, damit Texte zu setzen. Auf der anderen Seite werden im Trip-Test auch Eigenschaften abgetestet, die für ein TEX-kompatibles System nicht relevant sind.

$\varepsilon_{\mathcal{X}}$TEX (`http://www.extex.org`) hat als ein Ziel, in einem Kompatibilitäts-Modus, die gleiche Funktionalität wie TEX anzubieten. Deshalb ist es sinnvoll, hier eine Test-Suite aufzubauen, die sowohl die Kompatibilität sicher stellt, wie auch die Stabilität des bereits erstellten Codes überprüft.

Hyphenation – ein Beispiel

Um das Testen etwas anschaulicher zu gestalten betrachten wir ein Beispiel. Dies ist aus dem Bereich der Silbentrennung.

Testfälle entstehen aus Anforderungen. Anforderungen an TEX finden sich in [5, 4]. Also list man, was dort steht und versucht sich etwas auszudenken, wie das überprüft werden kann. Selbst ohne allzu große Kenntnisse in TEX kann man auf diese Weise Schritt für Schritt die Anforderungen erfassen und in Testfälle übersetzen.

Beipielsweise findet sich in [4] die folgende Aussage:[1]

> **902.** TEX will never insert a hyphen that has fewer than `\lefthyphenmin` letters before it or fewer than `\righthyphenmin` after it; hence, a short word has comparatively little chance of being hyphenated.

Die Silbentrennung in TEX wird dann benutzt, wenn der aktuelle Absatz nicht ohne Trennstellen „gut“ umbrochen werden kann. In diesem Fall werden zusätzliche, optionale Trennstellen eingefügt. Dazu gibt es einen Algorithmus,

[1] Übersetzung: TEX wird keine Trennstelle einfügen, die weniger als `\lefthyphenmin` Buchstaben davor oder weniger als `\righthyphenmin` danach haben; damit hat ein kurzes Wort eine relativ geringe Chance, getrennt zu werden.

der aus `\pattern{}` gespeist wird. Zusätzlich gibt es noch die Ausnahmen, die über `\hyphenation{}` eingegeben werden.

Für den ersten Testfall wollen wir uns auf die Werte aus `\hyphenation` konzentrieren. Gilt die Aussage auch für diese, oder nur für die Muster-basierten Trennstellen?

Um das zu überprüfen brauchen wir ein Minimalbeispiel. Da wir wirklich minimalistisch arbeiten wollen, nutzen wir zuerst einmal nur iniTEX. Dies ist der eigentliche Programmkern, ohne dass irgendwelche Definitionen aus Formaten gelesen werden.

Damit müssen wir uns um fast alles auch wirklich selbst kümmern. Das fängt damit an, dass die Klammern noch nicht als solche behandelt werden – die Kategorien müssen erst mit `\catcode` eingestellt werden. Auch gibt es noch keinen Font. Also laden wir cmr10 und benutzen ihn im Folgenden. Für diesen Font stellen wir auch noch den Hyphen-Charakter ein. Der Wert ist der ASCII-Code des Zeichens, das bei der Trennung eingefügt werden soll. Der Wert 45 ist dabei der Code für das Zeichen „-“. Das führt zu folgenden Zeilen Code:

```
\catcode`{=1
\catcode`}=2
\font\f cmr10 \f
\hyphenchar\f=45
```

Jetzt können wir mit dem eigentlichen Test beginnen. Wir wollen einem Wort mit `\hyphenation` eine Trennstelle zuweisen. Da es nicht relevant ist, welches Wort wir nehmen haben wir uns für `abcdef` entschieden, das in der Mitte trennbar sein soll. Damit auch getrennt werden muss stellen wir die Seitengröße auf eine Breite von 25 Punkten und eine Höhe von 100 Punkten. Schließlich setzen wir noch `\lefthyphenmin` und `\righthyphenmin` jeweils auf 0, damit auch wirklich getrennt werden kann.

```
\hyphenation{abc-def}
\hsize=25pt
\vsize=100pt
\lefthyphenmin=0
\righthyphenmin=0
```

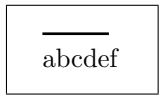
Um die Breite zu sehen, zeichnen wir mit `\hrule` eine Linie der Textbreite und können nun unser Wort „abcdef“ ausgeben. Mit `\end` schließen wir die Verarbeitung ab.

```
\hrule width \hsize height 1pt

abcdef

\end
```

Wenn wir die obigen Zeilen in einer Datei `htest.tex` haben, können wir auch schon ansehen, was TEX daraus macht:

```
initex htest.tex
```

Im Preview sehen wir dann in etwa das Folgende – allerdings ohne den Rahmen:

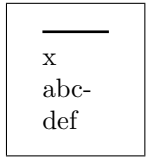


Das Wort wurde nicht getrennt, obwohl wir das erwartet haben. Irgend etwas stimmt da nicht. Vielleicht wird die Regel doch nicht auf Worte aus `\hyphenation` angewendet?

Doch halt. da war doch noch etwas. TEX trennt niemals das erste Wort eines Absatzes. Also müssen wir dafür sorgen, dass unser Wort nicht das erste ist. Dazu schreiben wir einfach ein x davor:

```
x abcdef
```

Damit erhalten wir:



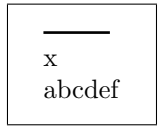
Genau das ist der erwartete Effekt. Also haben wir unser Minimalbeispiel erreicht. Daraus können wir noch einige weitere Testfälle ableiten, indem wir

`\lefthyphenmin` und `\righthyphenmin` variieren oder den Bindestrich nach links oder rechts verschieben.

Dann ergibt beispielsweise der folgende Code das Ergebnis, dass keine Trennung stattfindet:

```
\lefthyphenmin=4
\righthyphenmin=4
```

Das sieht dann folgendermaßen aus:



Die Test-Umgebung

Die Test-Suite von $\varepsilon_{\mathcal{X}}$TEX hat das Ziel, die Testfälle automatisiert ablaufen lassen zu können. Damit wird es möglich, ohne menschlichen Eingriff feststellen zu können in welchem Maß die Anforderungen schon erfüllt sind. Und schließlich ist es bei einer größeren Anzahl von Testfällen aufwändig, jedes Mal im Preview ein Ergebnis zu überprüfen.

$\varepsilon_{\mathcal{X}}$TEX wird in Java entwickelt. Damit liegt es nahe, die Werkzeuge aus der Java-Welt einzusetzen. Als ein Standard-Werkzeug für Unit-Tests hat sich hier JUnit [2] durchgesetzt. JUnit ist im Kern recht einfach – sowohl was die dahinter liegenden Konzepte als auch, was die Anwendung betrifft. Jeder Testfall ist eine Java-Methode; die umschließende Java-Klasse stellt eine Test-Suite dar.

Damit das funktioniert muss die Klasse von einer Basisklasse abgeleitet sein und die Methoden müssen einer Namenskonvention genügen. Die Basisklasse ist `junit.framework.TestCase` und der Klassenname muss mit „Test“ enden. Die Konvention sagt, das die Methoden `public void` sein müssen und mit der Zeichenkette „test“ beginnen.

In den Testfall-Methoden sollte dann auf einige der vielen „assert“-Methoden der Basis-Klasse zurückgegriffen werden. Diese dienen sowohl dazu, Nachbedingungen zu überprüfen, als auch, dem Test-Framework Erfolg oder Miss0-11.

```
public class MyTest extends TestCase {

    public MyTest(final String name) {
        super(name);
    }

    public void test1() throws Exception {

      // Perform tests here
      assertEquals(2, 1 + 1);
    }
}
```

Soviel als kurzer Ausflug in die „Tiefen“ von Java. Testfälle kann man aber auch erstellen und laufen lassen, wenn man kein Java kennt und nur die passenden Werte an die richtigen Stellen schreibt.

Die Aufgabe für ε_χTEX ist vielfach die Gleiche: Der Interpreter wird mit einer Eingabe gefüttert. Die Ausgabe ist einerseits der Strom der „Nodes“, die zur Ausgabe gehen und der Fehler-Kanal, der am Ende in der Log-Datei landet. Diese beiden Ströme können abgefangen und mit vorgegebenen Werten verglichen werden. Dafür gibt es die Klasse `ExTeXLauncher` und die Methode `assertOutput`. Dieser Methode gibt man drei Zeichenketten mit:

- den Code, der ausgeführt werden soll,
- das Ergebnis im Ausgabekanal und
- das Ergebnis in der Log-Datei.

Das sieht dann beispielsweise folgendermaßen aus:

```
public class RelaxTest extends ExTeXLauncher {

    public RelaxTest(final String name) {
        super(name);
    }

    public void test1() throws Exception {

      assertOutput(
          // --- Input code
          "\\relax\\end",
```

```
            // --- Log channel
            "",
            // --- Output channel
            ""
        );
    }
}
```

Man muss nur wissen, dass man den Backslash in der Zeichenkette verdoppeln muss und andere Sonderzeichen durch voranstellen eines Backslash kenntlich gemacht werden. Insbesondere der String-Begrenzer (\") und das Newline-Zeichen (\n) zu nennen.

Das war alles. Schon kann man die Tests laufen lassen. Damit alles noch mit einer netten, grafischen Oberfläche versehen wird, genügen die folgenden drei Zeilen:

```
    public static void main(final String[] args) {
        junit.swingui.TestRunner.run(RelaxTest.class);
    }
```

Das Ergebnis sieht dann wie das Beispiel in Abbildung 1 aus. Hier wird angezeigt, wieviele Testfälle enthalten sind und wieviele erfolgreich oder mit einem Fehler beendet wurden. Der Gesamterfolg wird durch den grünen Balken signalisiert; anderenfalls wäre er rot. In einem solchen Fall gilt es dann herauszufinden, wo die Ursache für den Fehlschlag zu suchen sein könnte.

Verschwiegen haben wir bisher, wie man die Tests laufen lasen kann. Und auch das geht recht einfach. Hierfür ist in $\varepsilon\chi$TEX das Skript `develop/juint` vorgesehen. Hier werden alle Einstellungen passend vorgenommen und der Test gestartet:

```
develop/junit de.dante.extex.interpreter.primitives.RelaxTest
```

Das funktioniert auf die gleiche Weise sowohl unter Unix wie auch unter Windows. Man muss also kein Unix-affiner Hacker sein, um Testfälle für $\varepsilon\chi$TEX machen und ausführen zu können

Genauso gut lassen sich die Testfälle aber auch – ohne diese Zusatzzeilen – aus einer IDE wie Eclipse (http://www.eclipse.org) heraus abarbeiten. Das ist an den gleichen Beispiel in Abbildung 2 zu sehen.

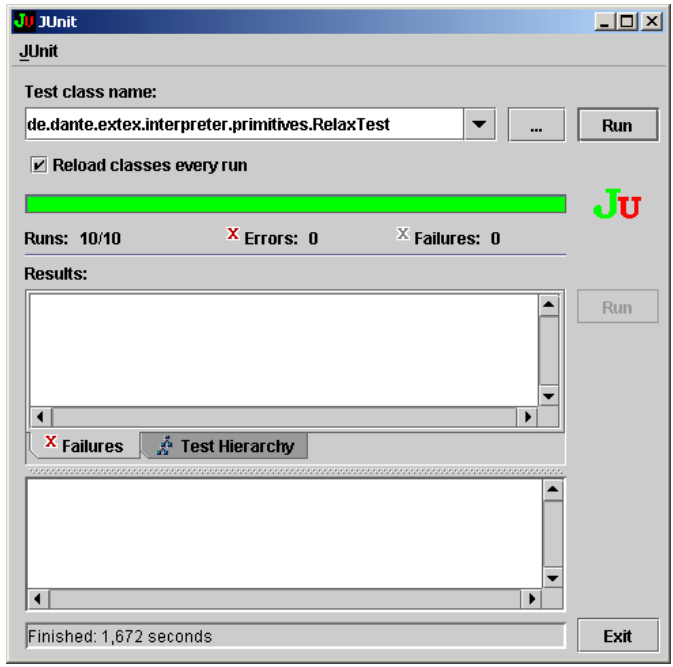


Abbildung 1: Ein Test-Bericht von JUnit

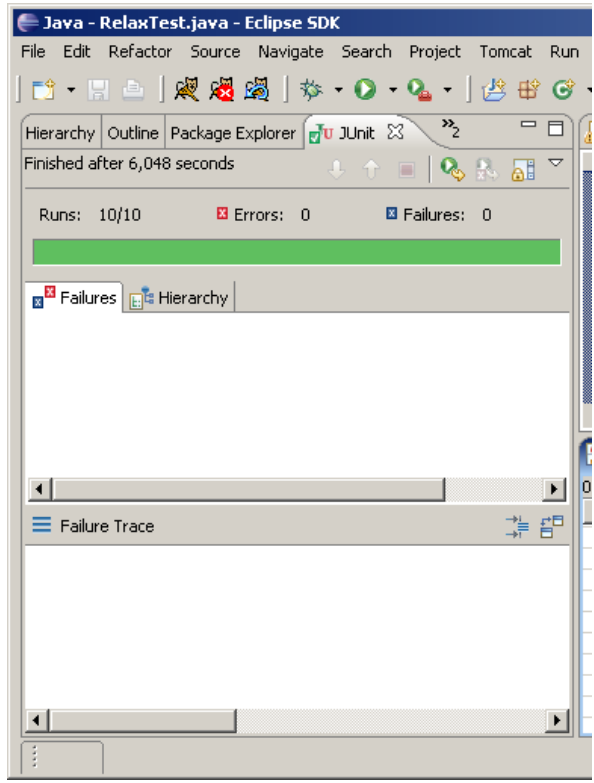


Abbildung 2: JUnit in Eclipse

```
.........
Time: 10,054

OK (10 tests)
```

Abbildung 3: JUnit, textbasiert

Schließlich gibt es auch noch die textbasierte Variante. Eine Ausgabe kann dann wie in Abbildung 3 aussehen. Diese wird auch benutzt, um alle Testfälle en-bloc abarbeiten zu lassen.

Zusammenfassung und Ausblick

Im Rahmen von $\varepsilon_{\mathcal{X}}$TEX wurde bisher eine Test-Suite in einer Basis-Version erstellt. Diese enthält bereits über 7000 Testfälle. Da die Testfälle immer nur einen kleinen Aspekt überprüfen, ist zu erwarten, dass es sehr viele davon geben muss. Eine einigermaßen vollständige Test-Suite wird sicher auf mehr als die doppelte Größe des bisher erreichten anwachsen müssen.

Wer bisher die Kunst des Minimalbeispiels auf der Ebene von LATEX verwendet hat, kann sich hier in einem neuen Minimalismus üben. Durch den tief gewählten Startpunkt muss man auch nicht viele Pakete kennen. Dafür muss man sich um einige Dinge kümmern, die sonst schon geregelt sind. Aus diese Weise kann man auf einfache Weise tiefe Einblicke in die Arbeitsweise von TEX – und $\varepsilon_{\mathcal{X}}$TEX – erhalten.

Für $\varepsilon_{\mathcal{X}}$TEX wäre es sehr hilfreich, wenn sich der eine oder andere finden würde, der sich in solchen Minimalbeispielen üben will und die Ergebnisse für $\varepsilon_{\mathcal{X}}$TEX beisteuert. Eine Mitarbeit in diesem oder einem anderen Bereich von $\varepsilon_{\mathcal{X}}$TEX ist jederzeit willkommen.

Literatur

[1] Kent Beck: *Test-Driven Development*; Addison-Wesley; Boston, MA; 2003.

[2] Kent Beck: *JUnit kurz & gut*; O'Reilly; Köln; 2005.

[3] Martin Fowler: *Refactoring – Wie Sie das Design vorhandener Software verbessern*; Addison-Wesley; 2000.

[4] Donald E. Knuth: *TEX – the Program*; Bd. B von *Computers and Typesetting*; Addison-Wesley; Boston, MA; 1986.

[5] Donald E. Knuth: *The TEXbook*; Bd. A von *Computers and Typesetting*; Addison-Wesley; Boston, MA; 1996.

[6] Johannes Link: *Unit Tests mit Java – der Test-First-Ansatz*; dpunkt.verlag; Heidelberg; 2002.