Warum White-Box-Test kein Test ist

Prof. Dr. Andreas Spillner
Hochschule Bremen, Flughafenallee 10, 28199 Bremen
Andreas.Spillner@hs-bremen.de

Abstract: Testentwurfsverfahren werden in die beiden Kategorien White-Box und Black-Box eingeteilt. Die Black-Box-Verfahren setzen auf der Spezifikation auf, um daraus systematisch Testfälle abzuleiten. Die White-Box-Verfahren nutzen hierfür zusätzlich den Programmtext. Oft werden beide als gleichwertig angesehen, wobei die White-Box-Verfahren eher auf den unteren Teststufen (Komponenten- und Integrationstest) angesiedelt werden. In der Praxis wird häufig eine angestrebte Code-Überdeckung als Endekriterium für den (White-Box-)Test verwendet. Warum dies ein kritisches Vorgehen ist, wird an mehreren Beispielen erläutert. Es wird dargelegt, warum White-Box-Test kein Test im engeren Sinne ist.

Stichworte: Testentwurfsverfahren, White-Box-Test, Black-Box-Test, White-Box-Control

1. Einleitung

Mit der zunehmenden Verbreitung von agilen Praktiken in der Softwareentwicklung hat auch der Test an Bedeutung gewonnen und wird nicht mehr als späte Phase der Entwicklung und notwendiges *Übel* angesehen. Allerdings wird meist wenig Wert auf die systematische Herleitung der Testfälle gelegt. Sind beispielsweise beim Komponententest alle Anweisungen getestet, wird oft kein Grund für weitere Überlegungen zu ergänzenden Testfällen gesehen.

Eine der agilen Praktiken ist die *Testgetriebene Entwicklung.* Auf den Internetseiten der it-agile, den „Experten für agile Softwareentwicklung“, findet sich folgendes Zitat: „Testgetriebene Entwicklung (Test-Driven Development) bedeutet, dass Tests vor dem Produktivcode geschrieben werden, um so die Softwareentwicklung zu steuern. So entsteht Qualitätssoftware mit sehr hoher Testabdeckung, und es wird nur das entwickelt, was auch tatsächlich benötigt wird.“ [it-agile AT]

Da zuerst die Testfälle vor dem Code spezifiziert werden und der Code dann genau *passend* zu den Testfällen muss jede Anweisung durch die Testfälle zur Ausführung gebracht werden, ansonsten ist nicht testgetrieben entwickelt worden.

Ein weiteres Zitat:
„Vorteile von TDD auf einen Blick
wartbare Qualitätssoftware:
• kein ungetesteter Code ...“ [it-agile TDD]

Was bedeutet nun getesteter Code in dem Zusammenhang? Vermutlich bezieht sich das *getestet* auf die Anweisungs- oder Zweig- bzw. Entscheidungsüberdeckung. Es wurde also jede Anweisung durch die Testfälle mindestens einmal zur Ausführung gebracht oder jede Bedingung wurde mindestens einmal zu *wahr* und einmal zu *falsch* ausgewertet. Der Kontrollfluss steht demnach im Mittelpunkt.

Weitere White-Box-Tests berücksichtigen den Datenfluss im Programm. Diese Ansätze sind in der Praxis und auch in der agilen Welt allerdings kaum verbreitet und werden hier nicht näher betrachtet.

2. Beispiele

Warum die kontrollflussorientierten White-Box-Kriterien nicht ausreichend zum Qualitätsnachweis sind, wird im Folgenden an einigen konkreten Beispielen erörtert.

2.1 Maximalwert von drei Zahlen

Ein sehr einfaches Beispiel: Aus drei Werten soll das Maximum ermittelt werden (in Anlehnung an [Kleuker 11]:

```
1.   public int max
        (int x,int y,int z) {

2.   int max=0;
3.   if (x>z)
4.           max=x;
5.   if (y>x)
6.           max=y;
7.   if (z>y)
8.           max=z;
9.   return max;
10.  }
```

Das Programm wird mit folgenden Testdaten ausgeführt:

TF1: `max(7,5,4), max=7`
TF2: `max(5,7,4), max=7`
TF3: `max(4,5,7), max=7`

Das erwartete Ergebnis (7) wird für alle drei Testfälle bei deren Durchführung bestätigt.

In Abbildung 1 ist der Kontrollflussgraph für das Programm abgebildet. Durch Ausführung der drei

Testfälle werden sowohl alle Anweisungen (Knoten in Graphen) als auch alle Zweige (Kanten im Graphen) erreicht:

Pfad für TF1: 1,2,3,4,5,7,9,10
Pfad für TF2: 1,2,3,4,5,6,7,9,10
Pfad für TF3: 1,2,3,5,6,7,8,9,10

Also kann der Test als erfolgreich angesehen und beendet werden, wenn 100% Zweigüberdeckung als Testendekriterium vereinbart wurde.

Da im Programm keine Schleifen vorkommen, sollte auch eine Pfadüberdeckung zu 100% erreichbar sein (insgesamt acht Pfade). Allerdings ist der Pfad, bei dem alle Bedingungen `wahr` sind, nicht ausführbar, da die Bedienung (`x>z ∧ y>x ∧ z>y`) nicht erfüllbar ist.

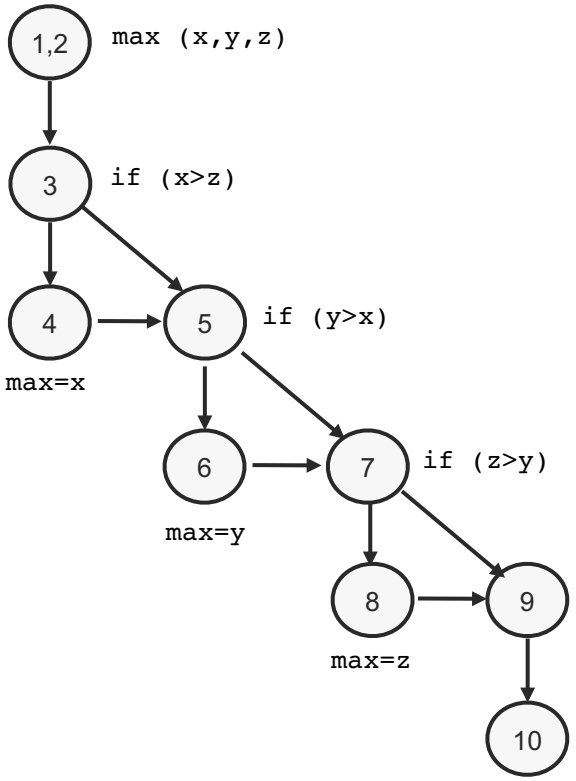

Abb.1: Kontrollflussgraph Maximum

Obwohl die drei Testfälle mit 100%-Zweigüberdeckung keine fehlerhaften Ergebnisse liefern, ist das Programm nicht fehlerfrei:

Pfad: 1,2,3,4,5,7,8,9,10
mit dem Testfall `max(7,4,5) max=5` und
Pfad: 1,2,3,5,7,9,10
mit dem Testfall `max(7,7,7) max=0`

führen zu Fehlerwirkungen.

Werden die Testfälle mittels der Äquivalenzklassenbildung ermittelt, ergeben sich folgende dreizehn Äquvalenzklassen mit den entsprechenden Testfällen:

ÄK1: x > y > z, ÄK2: x > z > y
ÄK3: y > x > z, ÄK4: y > z > x
ÄK5: z > x > y, ÄK6: z > y > x
ÄK7: x = y > z, ÄK8: x = y < z
ÄK9: x = z > y, ÄK10: x = z < y
ÄK11: y = z > x, ÄK12: y = z < x
ÄK13: x = y = z

Werden die dreizehn Testfälle (und nicht nur die drei oben angegebenen) ausgeführt, werden die beiden Fehler aufgedeckt. Die dreizehn Testfälle beinhalten die sieben ausführbaren Pfadtestfälle, diese sieben berücksichtigen aber nicht die mögliche Gleichheit von zwei der drei oder aller drei Parameter. Weitere Testfälle (negative, große, kleine Zahlen, keine Zahlen, usw.) sind ggf. ergänzend durchzuführen.

Das Beispiel zeigt sehr drastisch, dass eine 100%ige Anweisungs- oder Zweigüberdeckung keine Aussage über Fehlerfreiheit zulässt. Es ist nur sichergestellt, dass alle Programmteile zur Ausführung gekommen sind bzw. jede Bedingung zu `wahr` und `falsch` ausgewertet wurde.

Die folgenden Beispiele werden nicht mehr so ausführlich dargestellt.

2.2 Größter gemeinsamer Teiler

Als nächstes Beispiel wird die Ermittlung des größten gemeinsamen Teilers von zwei ganzen positiven Zahlen betrachtet.

```
1.   public int gcd(int m, int n) {
     // pre: m > 0 and n > 0
     // post: return > 0 and
     // m@pre.mod(return) = 0 and
     //...
2.          int r;
3.          if (n > m) {
4.             r = m;
5.             m = n;
6.             n = r;
            }
7.          r = m % n;
8.          while (r != 0) {
9.             m = n;
10.            n = r;
11.            r = m % n;
            }
12.         return n;
13.   }
```

TF1: `gcd(4,6), gcd=2`
TF2: `gcd(6,4), gcd=2`

Mit der Ausführung von Testfall1 werden bereits alle Anweisungen zur Ausführung gebracht. Beim Testfall2 wird die Bedingung `(n>m)` zu `falsch` ausgewertet und somit 100% Zweiüberdeckung erzielt. 100% Pfadüberdeckung ist durch die vorhandene Schleife mit ihren beliebig vielen

Schleifendurchläufen (und damit unterschiedlichen Pfaden) praktisch nicht erreichbar.

Für die Prüfung von Schleifen gibt es folgenden White-Box-Ansatz, der drei Testfälle erfordert:

a) die Abweisung (nicht Ausführung) der Schleife, falls möglich,
b) einen einmaligen Durchlauf und
c) mindestens eine Wiederholung des Durchlaufs.

Folgende ergänzenden Testfälle sind somit erforderlich:

a) `TF3: gcd(4,4), gcd=4 (neu)`
b) `TF1: gcd(4,6), gcd=2`
c) `TF4: gcd(104,169), gcd=13 (neu)`

Vier Testfälle erfüllen neben der Zweigüberdeckung auch die geforderte Prüfung der Schleife.

Der für die Funktionalität wichtige Testfall, dass es keinen gemeinsamen Teiler > 1 gibt, ist nicht berücksichtigt. Auch hier muss die Spezifikation und nicht der Programmtext herangezogen werden. Die Bildung von Äquivalenzklassen ist in diesem Fall für die Ausgaben zu erstellen. Ein Testfall muss durchgeführt werden, bei dem das Ergebnis, der gemeinsame Teiler `1` ist:

`TF5: gcd(5,4), gcd=1.`

2.3 Rabattberechnung

Das dritte Beispiel (aus [Spillner 12], S. 46) enthält ein unerreichbares Programmstück und eine vollständige Code-Überdeckung ist nicht möglich.

```
1.  double calculate_price
    (double baseprice, double
    specialprice, double extraprice,
    int extras, double discount)
2.  {
3.  double addon_discount;
4.  double result;
5.  if (extras≥3) addon_discount=10;
6.   else if (extras ≥ 5)
7.      addon_discount=15;
8.      else addon_discount=0;
9.  if (discount>addon_discount)
10.   addon_discount=discount;
11. result=
       baseprice/100.0*(100-discount)
       + specialprice + extraprice/
       100.0*(100-addon_discount);
12. return result;
```

Die Anweisung in Zeile 7 (Rabatt auf 15% setzen) wäre nur dann erreichbar, wenn die Bedingungen `(extras≥3) falsch` und die Bedingungen `(extras≥5) wahr` ist. Eine 100%ige Anweisungsüberdeckung ist somit nicht erreichbar. Eine fehlerträchtige Situation liegt möglicher Weise vor.

Der Tester kann nun versuchen, Testfälle zu spezifizieren und auszuführen, welche diese Anweisung zur Ausführung bringen sollen, was aber nicht gelingen wird. Ein Review des Programmstücks, *debugging* oder auch symbolische Ausführung (mit *Aufsammeln* der einzelnen Bedingungen) bis zur unausführbaren Anweisung führen zur Aufdeckung und zur Klärung des Problems, weitere Tests nicht.

3. Prüfung von Bedingungen

Zu den White-Box-Test zählen auch die Verfahren, die zusammengesetzte Bedingungen prüfen. Zusammengesetzte Bedingungen bestehen aus mehreren sogenannten atomaren Bedingungen (zB `(a>b)`) die mit logischen Operatoren (zB `AND`) verknüpft werden. Im Fokus der Prüfung der zusammengesetzten Bedingungen stehen dabei die atomaren Bedingungen und das Ergebnis der Gesamtbedingung. Motivation für die Ansätze ist die Einsicht, dass Tests nicht ausreichen, die nur den Gesamtwert der zusammengesetzten Bedingung (`wahr` und `falsch`) betrachten.

Ähnlich den Ansätzen zur Prüfung von Schleifen (s. Beispiel `gcd`) werden die jeweiligen zusammengesetzten Bedingungen separat untersucht. Somit wird eine komplexere Stelle im Programmtext näher betrachtet.

Im *Certifited Tester* Lehrplan »*Advanced Level Syllabus Technical Test Analyst*« [CTAL-TTA] werden folgende Verfahren aufgeführt:

- Einfacher Bedingungstest
- Bedingungs-/Entscheidungstest
- Modifizierter Bedingungs-/Entscheidungstest
- Mehrfachbedingungstest

Beim einfachen Bedingungstest wird gefordert, dass jede atomare Bedingung mindestens einmal den Wert `wahr` und einmal den Wert `falsch` ergibt. Das Ergebnis der Gesamtbedingung wird hierbei nicht betrachtet. Eine ungeschickter Wahl der Variablenwerte in den atomaren Bedingungen kann dazu führen, dass die Gesamtbedingung jeweils nur zu einen Wahrheitswert ausgewertet wird.

Der Bedingungs-/Entscheidungstest hebt diesen Nachteil dadurch auf, dass neben den atomaren Bedingungen auch die Gesamtbedingung bei der insgesamt betrachteten Ausführung der Testfälle beide Wahrheitswerte annehmen muss.

Beim modifizierter Bedingungs-/Entscheidungstest sind die Änderungen der Wahrheitswerte der einzelnen atomaren Bedingungen zu testen, deren Änderung zu einer Änderung des Wahrheitswerts der Gesamtbedingung führt.

Der Mehrfachbedingungstest fordert, dass alle Kombinationen der Wahrheitswerte der atomaren Bedingungen getestet werden. Sobald Abhängigkeiten zwischen den atomaren Bedingungen existieren, kann diese Forderung oft nicht erfüllt werden, wie folgendes Beispiel zeigt:

```
(x>0) ∧ (x<101)
```

Einen Wert für `x`, bei dem die beiden atomaren Bedingungen zu `falsch` ausgewertet werden, existiert nicht.

Die Schwierigkeit bei den White-Box-Tests zur Prüfung von zusammengesetzten Bedingungen liegt darin, wie die jeweiligen Testeingaben (Parameterwerte) zu wählen sind, damit die untersuchten atomaren und Gesamt-Bedingungen wie gefordert ausgewertet werden. Die Testdaten dienen meist ausschliesslich dem Nachweis der Überdeckung und sind oft ohne jede praktische Relevanz.

Der Ablauf zum Nachweis der erzielten Bedingungsüberdeckungen unterscheidet sich nicht von den anderen White-Box-Ansätzen. Bei der Ausführung der bereits spezifizierten Testfälle ist zu ermitteln, welche atomaren und Gesamt-Bedingungen welche Wahrheitswerte mit welchen Auswirkungen angenommen haben. Dann ist zu prüfen, in wieweit die geforderten Bedingungsüberdeckungen bereits erreicht sind und welche Testfälle ggf. noch zusätzlich zu spezifizieren sind.

4. Resümee

Die aufgeführten Beispiele zeigen, dass eine erreichte Code-Überdeckung kein Garant für die Qualität des Programms ist. Auch werden nicht alle funktional notwendigen Testfälle durch die White-Box-Tests erzwungen. Vorteil ist die Aufdeckung von bisher nicht ausgeführten Programmteilen, diese können erkannt werden, wenn eine 100%ige Anweisungs- oder Zweigüberdeckung nachzuweisen ist.

Beim agilen Vorgehen, wie zB beim TDD, werden die Testfälle spezifiziert und dann der *passende* Code realisiert. Eine Code-Überdeckung wird somit durch das Vorgehen impliziert. Als Qualitätsnachweis kann die Überdeckung nicht herangezogen werden. Der Code wird geprüft. Eine Hauptaufgabe der Tests ist aber der Nachweis der Erfüllung der Anforderungen an das Programm.

Dass dies keine neue Erkenntnis ist, soll durch das folgende Zitat belegt werden: »*... parameter testing*[1] *is guided by the code specifications instead of by the coded program. ... in other words, a programmer must prove that he satisfied his specifications, not that his program will perform as coded*« [Benington 56]

Entscheidend für den Qualitätsnachweis ist die Vorgehensweise bei der Spezifikation der Testfälle, diese wird allerdings all zu oft - und nicht nur bei agilen Projekten - weder methodisch noch systematisch durchgeführt.

Ein zielführendes Vorgehen ist die Verwendung von Black-Box-Testspezifikationsverfahren (zB die Bildung von Äquivalenzklassen), die Ausführung der systematisch ermittelten Testfälle und die anschliessende Kontrolle, welche Programmteile durch diese Testfälle zur Ausführung gekommen sind und welche noch nicht. Danach sind ggf. ergänzende Testfälle zu spezifizieren oder der Code ist mit anderen Techniken zu analysieren.

Der Lehrplan zum *Certified Tester - Foundation Level* [CTFL] trägt dem bereits Rechnung. Für die Black-Box-Verfahren werden in den Kursen 150 Minuten veranschlagt und es werden fünf unterschiedliche Verfahren aufgeführt. Für zwei White-Box-Verfahren sind 60 Minuten vorgesehen.

Der Begriff White-Box-*Test* ist irreführend, die Verfahren kontrollieren eher die Güte der bisher durchgeführten Testfälle in Bezug auf die erreichte Überdeckung des Programmcodes. Dinge sollen beim *richtigen* und nicht irreführenden Namen genannt werden:

White-Box-Control

ist die passendere Bezeichnung, auch wenn ein weißer Kasten nicht wirklich durchsichtiger ist als ein schwarzer. Die assoziativere Bezeichnung *Glass-Box* hat sich nicht durchgesetzt, daher ginge mit dem Begriff *Glass-Box-Control* der Bezug zur bisherigen Bezeichnung zu sehr verloren.

Referenzen

[Benington 56] H.D. Benington: Production of Lagre Computer Programs, Proc. of Symposium On Advanced Computer Programs for Digital Computers, June 1956, Washington, D.C., USA

[CTAL-TTA] Certified Tester - Advanced Level Syllabus Technical Test Analyst, Version 2012 http://www.german-testing-board.info/service/information/lehrplaene.html#c72 10.9.2013

[CTFL] Certified Tester Foundation Level Syllabus, Version 2011 1.0.1, deutschsprachige Ausgabe http://www.german-testing-board.info/service/information/lehrplaene.html#c72 10.9.2013

[it-agile TDD] http://www.it-agile.de/wasisttdd.html, 10.9.2013

[it-agile AT] http://www.it-agile.de/agiles-testen.html, 10.9.2013

[Kleuker 11]: S. Kleuker: Open Source Testwerkzeuge für alle Testphasen. 6. Treffen Softwaretest und Qualitätssicherung, 12. & 13.9.2011, Leipzig

[Spillner 12] Spillner, A.; Linz, T.: Basiswissen Softwaretest, 5. Auflage, dpunkt Verlag, 2012

[1] *parameter testing* ist die erste Teststufe und wird heute als Unit- oder Komponenten-Test bezeichnet