Testen

Weiterführende Literatur:

- Hoffmann, *Software-Qualität*, Kapitel 4 „Software-Text“, Seite 157-246

Testgetriebene Entwicklung (Test-Driven Development)[1]

Bei der testgetriebenen Entwicklung erstellt der/die ProgrammiererIn Softwaretests konsequent vor den zu testenden Komponenten.[2] Test-Driven Development (TDD) hat sich im Rahmen der agilen Software-Entwicklung als effiziente agile Praxis etabliert[3]

Black-Box-Test[4]

Die Grundlage für das Black-Box Testverfahren sind *Anforderungen und Spezifikationen*, für die der innere Aufbau der Komponenten oder des Systems (also die konkrete Umsetzung und Implementierung) zum Zeitpunkt des Tests *nicht bekannt* sein muss. Die Testobjekte werden daher *unabhängig von ihrer Realisierung getestet*. Aufgrund der Betrachtung des zu testenden Objekts als Black-Box werden die Testfälle von *Daten getrieben* (*Data-Driven*) und beziehen sich auf Anforderungen und das spezifizierte Verhalten der Testobjekte. Das Testobjekt wird mit definierten Eingangsparametern aufgerufen. Die Ergebnisse - nach Abarbeitung durch das Testobjekt - werden mit den erwarteten Ergebnissen verglichen. Stimmen die tatsächlichen Ergebnisse nicht mit den erwarteten Ergebnissen überein, liegt ein Fehler vor. Ziel ist es dabei, eine *möglichst hohe Anforderungsüberdeckung* zu erreichen, also möglichst alle Anforderungen zu testen. Um die Anzahl der Tests bei gleichbleibender Testintensität zu reduzieren, bedient man sich Techniken, wie *Äquivalenzklassenzerlegung* und *Grenzwertanalyse*. Dabei versucht man, die Menge an Eingabe- und Ausgabedaten zu beschränken, um mit minimalem Aufwand möglichst alle Testfälle abzudecken.[5]

Anforderungen und Spezifikationen
nicht bekannt
unabhängig von ihrer Realisierung getestet
Daten getrieben
Data-Driven
möglichst hohe Anforderungsüberdeckung
Äquivalenzklassenzerlegung
Grenzwertanalyse

Äquivalenzklassenzerlegung[6]

Das *vollständige Testen* einer Komponente oder eines Software-Systems ist in der Regel aufgrund der Vielzahl an unterschiedenen Werten, die sowohl Eingangsparameter als auch Ausgangsparameter annehmen können, kaum realisierbar und auch meist nicht sinnvoll. Um die Anzahl der Testdaten auf ein vernünftiges Minimum zu reduzieren, bedient man sich der Äquivalenzklassenzerlegung. Die Äquivalenzklassenzerlegung hilft dabei, *Bereiche von Eingabewerten* zu identifizieren, die jeweils *dieselben Ergebnisse* liefern. Aus diesen Klassen von

vollständige Testen
dieselben Ergebnisse

[1] Schatten, *Best Practice Software-Engineering*, Kaptiel 5.7 Seite 150-145.
[2] *Wikipedia-Artikel „Testgetriebene Entwicklung“*.
[3] Schatten, *Best Practice Software-Engineering*, Seite 151.
[4] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 32.
[5] Schatten, *Best Practice Software-Engineering*, Seite 140-141.
[6] Schneider, *Taschenbuch der Informatik*, Seite 250.

Vertreter (Repräsentanten)

Eingabewerte wählt man jeweils einen *Vertreter* (*Repräsentanten*) aus, der dann für den konkreten Testfall verwendet wird.

Komplizierter wird die Äquivalenzklassenzerlegung, falls Bedingungen mit mehreren Parametern zu testen sind, da sämtliche Kombinationen von Parameterklassen getestet werden müssen. Allerdings gilt auch hier wieder, dass

pro Äquivalenzklassenkombination nur ein Repräsentant

pro Äquivalenzklassenkombination nur ein Repräsentant ausgewählt werden muss und die Anzahl der Testfälle auf diese Weise beschränkt werden kann.[7]

Grenzwertanalyse[8]

Bereichsgrenzen von Äquivalenzklassen

Einen besonderen Stellenwert nehmen *Bereichsgrenzen von Äquivalenzklassen* ein, da diese häufige Ursachen für Fehler sind, die beispielsweise durch *Tippfehler* verursacht werden. Statt eines „<=“ wird ein „<“ geschrieben, was speziell an den Systemgrenzen zu einem Fehlverhalten des Systems führt. Diese Fehlerquelle kann beispielsweise mit der Grenzwertanalyse erkannt werden. Die Kernidee bei der Grenzwertanalyse ist, (a) *Grenzbereiche zu identifizieren* und (b) *Testdaten aus dem nahen Umfeld* dieser Bereichsgrenzen auszuwählen. Dabei empfiehlt es sich, jeweils einen Wert aus dem Grenzbereich der einen und der anderen Klasse auszuwählen. Optional kann natürlich auch der exakte Grenzwert in einem Testfall spezifiziert werden, der aber - bei genauer Analyse - ohnehin einer der beiden Klassen zugeordnet werden kann.[9]

Tippfehler

Grenzbereiche zu identifizieren

Testdaten aus dem nahen Umfeld

White-Box-Testtechniken[10]

Weiterführende Literatur:

- *Wikipedia-Artikel „White-Box-Test“*
- Hoffmann, *Software-Qualität*, Seite 199-246

Bei White-Box-Tests ist der Quellcode bekannt und wird zum Testen benutzt.[11]

Kontrollflussorientiertes Testen[12]

Weiterführende Literatur:

- *Wikipedia-Artikel „Kontrollflussorientierte Testverfahren“*
- Schneider, *Taschenbuch der Informatik*, Kapitel 8.6.3 „Methoden zur Testfallermittlung“, Seite 251-252

Dynamische White-Box-Tests, die sich am Kontrollflussgraphen des Programms orientieren.

- Anweisungsüberdeckungstest
- Zweigüberdeckungstest

[7] Schatten, *Best Practice Software-Engineering*, Seite 142.
[8] Schneider, *Taschenbuch der Informatik*, Seite 251.
[9] Schatten, *Best Practice Software-Engineering*, Seite 142.
[10] Hoffmann, *Software-Qualität*, Seite 199-246.
[11] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 32.
[12] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 34.

- Pfadüberdeckungstest

- Bedingungsüberdeckungstest

Kontrollflussgraph

Ein Kontrollflussgraph (englisch: *control-flow graph* (CFG)) bezeichnet einen *gerichteten Graphen* der dazu dient, den Kontrollfluss eines Computerprogramms zu beschreiben. Sie werden unter anderem zur Programmoptimierung eingesetzt.[13]

control-flow graph
gerichteten Graphen

Jeder Graph enthält genau *einen Startknoten* und *einen Endknoten*. *Verschiedenen Aussprungspunkte* (z. B. mehrere return-Anweisungen) werden in einem neuen, *zusätzlich eingeführten Knoten* zusammengeführt.[14]

einen Startknoten
einen Endknoten
Verschiedenen Aussprungspunkte
zusätzlich eingeführten Knoten

Der *kantenmarkierter* Kontrollflussgraphen ist ein Graph, indem die *Anweisungen den Knoten* und die *Verzweigungsbedingungen den Kanten* zugeordnet werden. Die meisten White-Box-Tests basieren auf dieser Art der Kontrollflussmodellierung.[15]

kantenmarkierter
Anweisungen den Knoten
Verzweigungsbedingungen den Kanten

Expandierte Kontrollflussgraphen enthalten für jeden Befehl einen separaten Knoten.[16]

Expandierte Kontrollflussgraphen

Kollabierte Kontrollflussgraphen zeichnen sich dadurch aus, dass verzweigungsfreie Befehlsblöcke vollständig in einem einzigen Knoten zusammengefasst werden.[17]

Kollabierte Kontrollflussgraphen

Der Kontrollflussgraph wird in manchen Aufgaben auch *Ablaufdiagramm* genannt. Gestrichelte Kanten können eingezeichnet werden, um einen `false`-Zweig von einem `true`-Zweig besser unterscheiden zu können. Oftmals sind die Knoten mit dem Präfix „n“ (für node) benannt. *S* steht für Start-Knoten und *E* steht für End-Knoten.

Ablaufdiagramm

Kontrollflussgraphen der elementaren Verzweigungs- und Schleifenkonstrukte[18]

- `if (B) X;`

- `if (B) X; else Y;`

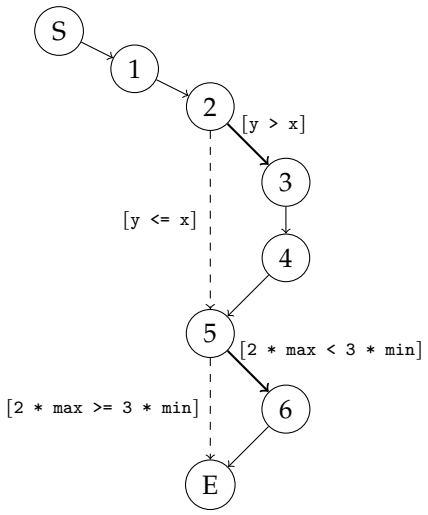
- `while (B) X;`

- `do X; while (B);`

[13] *Wikipedia-Artikel „Kontrollflussgraph“*.
[14] Hoffmann, *Software-Qualität*, Seite 203.
[15] Hoffmann, *Software-Qualität*, Seite 203.
[16] Hoffmann, *Software-Qualität*, Seite 204.
[17] Hoffmann, *Software-Qualität*, Seite 204.
[18] Hoffmann, *Software-Qualität*, Seite 205.

```java
int max(int x, int y) {
  // S
  int max = x; // 1
  int min = y; // 1
  if (y > x) { // 2
    max = y; // 3
    min = x; // 4
  }
  int z = 2 * max; // 5
  if (2 * max < 3 * min) {
    z = 3 * min; // 6
  }
  return z; // E
}
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/testen/Beispiel.java`



C_0-Test: Anweisungsüberdeckung (Statement Coverage)[19]

jede Anweisung mindestens ein Mal

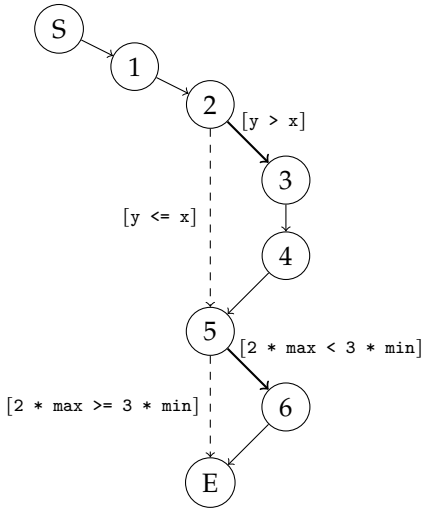
vollständiger Anweisungsüberdeckung

kein toter Code

Anweisungsüberdeckungstests, auch C_0-Test genannt, testen *jede Anweisung mindestens ein Mal*. Wurde jede Anweisung in einem Programm mindestens einmal ausgeführt, spricht man von *vollständiger Anweisungsüberdeckung*. Wurde vollständige Anweisungsüberdeckung erreicht, dann steht fest, dass *kein toter Code* (Anweisungen, die niemals durchlaufen werden) im Programm existiert.[20]

```java
int max(int x, int y) {
  // S
  int max = x; // 1
  int min = y; // 1
  if (y > x) { // 2
    max = y; // 3
    min = x; // 4
  }
  int z = 2 * max; // 5
  if (2 * max < 3 * min) {
    z = 3 * min; // 6
  }
  return z; // E
}
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/testen/Beispiel.java`



`max(4, 5);` : (S) - (1) - (2) - (3) - (4) - (5) - (6) - (E)

[19] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 36.

[20] *Wikipedia-Artikel „Kontrollflussorientierte Testverfahren“.*

C_1-Test: Zweigüberdeckung (Branch Coverage)[21]

Die Zweigüberdeckung (auch Kantenüberdeckung) fordert, dass jede *Kante* des Kontrollflussgraphen von mindestens einem Testfall durchlaufen werden muss. Um das Kriterium zu erfüllen, müssen die Testfälle so gewählt werden, dass jede Verzweigungsbedingung mindestens *einmal wahr* und mindestens *einmal falsch* wird. Da hierdurch alle Knoten ebenfalls mindestens einmal besucht werden müssen, ist die *Anweisungsüberdeckung* in der Zweigüberdeckung *vollständig enthalten*.[22]

Kante

einmal wahr

einmal falsch

Anweisungsüberdeckung

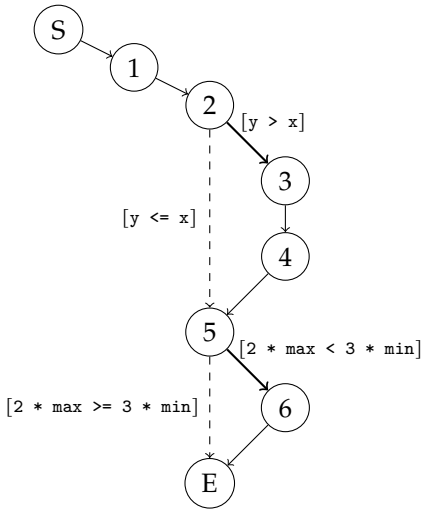
vollständig enthalten

Der Zweigüberdeckungstest wird auch *Entscheidungsüberdeckungstest* genannt, da die Hilfsvariable mindestens einmal mit dem Wert `true` und `false` durchlaufen werden muss. In diesem Fall muss die While-Schleife mindestens zweimal durchlaufen werden.[23]

Entscheidungsüberdeckungstest

```
4   int max(int x, int y) {
    ↪  // S
5     int max = x; // 1
6     int min = y; // 1
7     if (y > x) { // 2
8       max = y; // 3
9       min = x; // 4
10    }
11    int z = 2 * max; // 5
12    if (2 * max < 3 * min)
      ↪  {
13      z = 3 * min; // 6
14    }
15    return z; // E
16  }
```

Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/testen/Beispiel.java`



`max(4, 5);` : (S) - (1) - (2) - (3) - (4) - (5) - (6) - (E)
`max(3, 2);` : (S) - (1) - (2) - (5) - (E)

Zyklomatische Komplexität[24]

- gibt an, wie viele Testfälle höchstens nötig sind, um eine Zweigüberdeckung zu erreichen
- i.A. Metrik für „Testbarkeit/Komplexität“

Berechnung durch Anzahl Binärverzweigungen b (p Anzahl der Zusammenhangskomponenten des Kontrollflussgraphen)

$$M = b + p$$

$\rightarrow M = 2 + 1 = 3$

[21] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 37.
[22] Hoffmann, *Software-Qualität*, Seite 209.
[23] *Wikipedia-Artikel „Kontrollflussorientierte Testverfahren“*.
[24] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 38.

oder durch Anzahl Kanten e und Knoten n

$$M = e - n + 2p$$

$$\rightarrow M = 9 - 8 + 2 \cdot 1 = 3^{25}$$

C_2-Test: Pfadüberdeckung (Path Coverage)[26]

immensen Komplexität
geringe Praxisbedeutung

Die Pfadüberdeckung ist die mit Abstand mächtigste White-Box-Prüftechnik, besitzt aufgrund ihrer *immensen Komplexität* aber nur eine äußerst *geringe Praxisbedeutung*.

C_{2a} Vollständige Pfadüberdeckung (Full Path Coverage):

jeden möglichen Pfad

Das Kriterium der vollständigen Pfadüberdeckung wird erst dann erfüllt, wenn für *jeden möglichen Pfad*, der den Eingangsknoten des Kontrollflussgraphen mit dem Ausgangsknoten verbindet, ein separater Testfall existiert.[27]

kombinatorische Explosion
Boundary-Interior-Pfadüberdeckung
strukturierte Pfadüberdeckung

Um die Limitierungen der vollständigen Pfadüberdeckung zu überwinden, wurden in der Vergangenheit mehrere Variationen vorgeschlagen, mit deren Hilfe sich die *kombinatorische Explosion* zumindest teilweise eindämmen lässt. Von Bedeutung sind in diesem Zusammenhang insbesondere die *Boundary-Interior-Pfadüberdeckung* und die *strukturierte Pfadüberdeckung*.

C_{2b} Schleife-Inneres-Pfadüberdeckung (Boundary-Interior Path Coverage):

Der Boundary-Interior-Pfadtest konstruiert für jede Schleife des Programms drei Gruppen von Testfällen, die alle erfolgreich abgearbeitet werden müssen.

Schleifenkörper nicht betreten

Äußere Pfade Die Testfälle durchlaufen das Programm auf Pfaden, die den *Schleifenkörper nicht betreten*. Für Schleifen fester Lauflänge sowie für While-Schleifen, die erst am Ende des Schleifenkörpers die Wiederholungsbedingung auswerten, ist diese Testfallgruppe leer.

Schleifenkörper betreten
keiner Wiederholung
alle möglichen Pfade getestet

Grenzpfade (boundary paths, boundary test) Die Testfälle durchlaufen das Programm auf Pfaden, die den *Schleifenkörper betreten*, jedoch zu *keiner Wiederholung* führen. Innerhalb des Schleifeninneren müssen *alle möglichen Pfade getestet* werden. Für Schleifen fester Lauflänge ist diese Testfallgruppe leer.

mindestens eine weitere Iteration
innerhalb der ersten beiden Ausführungen alle möglichen Pfade abgearbeitet

Innere Pfade (interior test) Die Testfälle durchlaufen das Programm auf Pfaden, die den Schleifenkörper betreten und *mindestens eine weitere Iteration* ausführen. Die Testfälle werden so gewählt, dass *innerhalb der ersten beiden Ausführungen alle möglichen Pfade abgearbeitet* werden.[28]

[25] Hoffmann, *Software-Qualität*, Kapitel 4.4.6 McCabe-Überdeckung, Seite 216-220.
[26] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 39.
[27] Hoffmann, *Software-Qualität*, Seite 210.
[28] Hoffmann, *Software-Qualität*, Seite 212.

C_{2c} Strukturierte Pfadüberdeckung (Structured Path Coverage):

Anzahl der Schleifendurchläufe auf eine Zahl n reduziert

vollständige Pfadüberdeckung

`(S, 1, 2, 3, 4, 5, 6, E), (S, 1, 2, 5, E), (S, 1, 2, 3, 4, 5, E), (S, 1, 2, 5, 6, E)`

C_3-Test: Bedingungsüberdeckung (Condition Coverage)[29]

Problem: zusammengesetzte, hierarchische Bedingungen nicht ausreichend getestet

Einfachbedingungsüberdeckung: Alle atomaren Prädikate müssen mindestens einmal beide Wahrheitswerte annehmen. ($\rightarrow$ A = 0, B = 1; A = 1, B = 0) Mehrfachbedingungsüberdeckung: Alle Wahrheitskombinationen müssen getestet werden. ($\rightarrow$ A = 0, B = 1; A = 1, B = 0; A = 1, B = 1; A = 0, B = 0) Minimaler Mehrfachbedingungsüberdeckung: („Zwischenlösung“) Alle atomaren und zusammengesetzten Prädikate müssen einmal beide Wahrheitswerte annehmen.

		erfüllte Bedingung	**Durchführbarkeit**
Anweisungsüberdeckungstest	C0	jede Anweisung wird mindestens einmal ausgeführt	relativ einfach
Zweigüberdeckungstest	C1	jede Kante im Kontrollflussgraph (KFG) wird mindestens einmal durchlaufen	realistische Mindestanforderung, vertretbarer Aufwand
Pfadüberdeckungstest	C2		
Vollständig	C2a	Alle möglichen Pfade werden durchlaufen	unmöglich bei Schleifen
Boundary-Interior	C2b	wie C2a, Schleifen werden jedoch nach speziellen Regeln durchlaufen	aufwendig
Strukturiert	C2c	wie C2b, Schleifen werden jedoch genau n-mal durchlaufen	aufwendig
Bedingungsüberdeckungstest	C3		
Einfachbedingung	C3a	jede atomare Bedingung wird einmal mit true und false getestet	
Mehrfachbedingung	C3b	jede true/false Kombination der atomaren Bedingungen wird getestet	sehr hoher Aufwand
Minimale Mehrfachbedingung	C3c	jede atomare Bedingung und die Gesamtbedingung wird mit true und false getestet	hoher Aufwand

Quelle: Wikipedia[30]

Datenflussorientiertes Testen

Weiterführende Literatur:

- Schneider, *Taschenbuch der Informatik*, Kapitel 8.6.3 „Methoden zur Testfallermittlung“, Seite 250-251
- Hoffmann, *Software-Qualität*, Kapitel 4.4.7 Def-Uses-Überdeckung, Seite 220-227

[29] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 40.
[30] *Wikipedia-Artikel „Kontrollflussorientierte Testverfahren“.*

Im Gegensatz zu allen bisher betrachteten Überdeckungskriterien, die ausschließlich den Kontrollfluss eines Programms in Betracht ziehen, leiten die Defs-Uses-Kriterien die auszuführenden Pfade aus dem Datenfluss ab. Hierzu werden die *Variablenzugriffe* des untersuchten Programms analysiert und *mit verschiedenen Datenflussattributen versehen*. Die vergebenen Attribute teilen die Variablenzugriffe in *drei Klassen* ein:

Variablenzugriffe

mit verschiedenen Datenflussattributen versehen

drei Klassen

Sie sind besonders geeignet für objektorientiert entwickelte Systeme.

Definitorische Nutzung (`def-use`): In diese Klasse fallen alle Zugriffe, die den aktuellen *Wert einer Variablen überschreiben*. Alle Variablenzugriffe auf der linken Seite einer Zuweisung fallen in diese Kategorie. (Beispiel: `x = 42;`)

Wert einer Variablen überschreiben

Referenzierende Nutzung (`r-use`): In diese Klasse fallen alle Zugriffe, die den Wert einer Variablen *verwenden, jedoch nicht verändern*. Zwischen einer Zuweisung und einer nachfolgenden Referenz, die den zugewiesenen Wert verwendet, besteht eine du-Interaktion. Referenzierende Nutzungen können weiter in berechnende und prädikative Nutzungen unterteilt werden.

verwenden, jedoch nicht verändern

Berechnende Nutzung (`c-use`): In diese Klasse fallen alle Zugriffe, die den *Wert einer Variablen in einer Berechnung verwenden* (*computational use*, kurz `c-use`). Zwischen einer Zuweisung und einer sich anschließenden berechnenden Nutzung besteht eine dc-Interaktion. (Beispiel: `x = x * 2;`) (+ `def-use`)[31]

Wert einer Variablen in einer Berechnung verwenden

computational use

Prädikative Nutzung (`p-use`): In diese Klasse fallen alle Zugriffe, die den Wert einer Variablen *prädikativ*, d. h. innerhalb eines *booleschen Ausdrucks* verwenden (*predicative use*, kurz `p-use`). Zwischen einer Zuweisung und einer sich anschließenden prädikativen Nutzung besteht eine dp-Interaktion. (Beispiel: `if (x > 42)`)[32]

prädikativ

booleschen Ausdrucks

predicative use

33

Datenflussgraph

```
18  int minMax(int min, int max) { // S
19    if (min > max) { // 1
20      int tmp = max; // 2
21      max = min; // 2
22      min = tmp; // 2
23    }
24    return max - min; // E
25  }
```

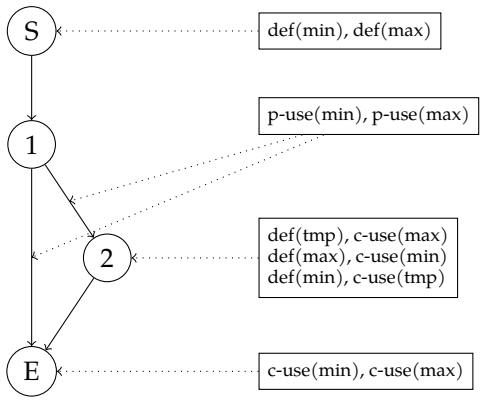
Code-Beispiel auf Github ansehen: `src/main/java/org/bschlangaul/testen/Beispiel.java`

def-use: Knoten

c-use: Knoten

p-use: Kanten[34]

[31] Hoffmann, *Software-Qualität*, Seite 220.
[32] Hoffmann, *Software-Qualität*, Seite 221.
[33] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 42.
[34] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 43.



Überdeckungskriterium

Definitionsfreier Pfad: Wird x im Knoten 0 definiert, so ist der Pfad 0, 1, 2, 3, 4, 5 definitionsfrei, wenn x in keinem weiteren Knoten 1, 2, 3, 4, 5 erneut definiert wird.

Testfälle durchlaufen für jede Definition einer Variable einen definitionsfreien Pfad zu ...

all definitions: mindestens einem p- oder c-use → S, 1, 2, E

all c-uses: allen erreichbaren c-uses → S, 1, 2, E und S, 1, E

all p-uses: allen erreichbaren p-uses → S, 1, 2, E und S, 1, E

all uses: zu allen erreichbaren p- und c-uses → S, 1, 2, E und S, 1, E

all c-uses/some p-uses bzw. all p-uses/some c-uses: falls zu manchen Variablendefinitionen kein c- bzw. p-use vorhanden ist, stattdessen mindestens einen p-use bzw. c-use testen.

35

Design by Contract[36]

Ziel des Design by contract ist das reibungslose Zusammenspiel einzelner Programmmodule durch die Definition formaler Verträge zur Verwendung von Schnittstellen, die über deren statische Definition hinausgehen.

Das reibungslose Zusammenspiel der Programmmodule wird durch einen „Vertrag“ erreicht, der beispielsweise bei der Verwendung einer Methode einzuhalten ist. Dieser besteht aus

Vorbedingungen (englisch precondition), also den Zusicherungen, die der Aufrufer einzuhalten hat

[35] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*, Seite 44.
[36] *Wikipedia-Artikel „Design by contract“*.

Nachbedingungen (englisch postcondition), also den Zusicherungen, die der Aufgerufene einhalten wird, sowie den

Invarianten (englisch class invariants), über alle Instanzen einer Klasse hinweg geltende Grundannahmen.

Der Vertrag kann sich auf die gesamte verfügbare Information beziehen, also auf Variablen- und Parameter-Inhalte ebenso wie auf Objektzustände des betroffenen Objekts oder anderer zugreifbarer Objekte. Sofern sich der Aufrufende an Vorbedingungen und Invarianten hält, können keine Fehler auftreten und die Methode liefert garantiert keine unerwarteten Ergebnisse.

Unit-Test

atomare Programmeinheit

Modultest
Komponententest

Mit dem Begriff Unit wird eine *atomare Programmeinheit* bezeichnet, die groß genug ist, um als solche eigenständig getestet zu werden. Der Unit-Test wird nicht selten auch als *Modultest* oder noch allgemeiner als *Komponententest* bezeichnet. Wird die Methodik des Unit-Tests auf größere Klassen- oder Modulverbünde angewendet, so geht der Testprozess fast nahtlos in den Integrationstest über.[37]

Literatur

[1] Dirk W. Hoffmann. *Software-Qualität*. 2013.

[2] Alexander Schatten. *Best Practice Software-Engineering. Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. 2010.

[3] Uwe Schneider. *Taschenbuch der Informatik*. 7. Aufl. Hanser, 2012. ISBN: 9783446426382.

[4] *Softwaresysteme: Präsenztag 5: Foliensatz: Formale Verifikation und Testen*. `https://www.studon.fau.de/file2800509_download.html`.

[5] *Wikipedia-Artikel „Design by contract“*. `https://de.wikipedia.org/wiki/Design_by_contract`.

[6] *Wikipedia-Artikel „Kontrollflussgraph“*. `https://de.wikipedia.org/wiki/Kontrollflussgraph`.

[7] *Wikipedia-Artikel „Kontrollflussorientierte Testverfahren“*. `https://de.wikipedia.org/wiki/Kontrollflussorientierte_Testverfahren`.

[8] *Wikipedia-Artikel „Testgetriebene Entwicklung“*. `https://de.wikipedia.org/wiki/Testgetriebene_Entwicklung`.

[9] *Wikipedia-Artikel „White-Box-Test“*. `https://de.wikipedia.org/wiki/White-Box-Test`.

[37] Hoffmann, *Software-Qualität*, Seite 159 Kapitel 4.2.1.1 Unit Tests.