Automatisches Testen

Automatische Tests

- Test-Code in eigens dafür vorgesehenem Projektbereich
- Ausführung wie „Main-Methoden“, aber Tool-Unterstützung für:
  - Vergleich mit erwartetem Verhalten
  - automatische Durchführung aller Tests
  - Statistiken über bestandene, nicht bestandene Tests

Vorteile

- Trennung Funktionalität und Tests
- Tests bleiben erhalten
- Tests können regelmäßig, automatisch ausgeführt werden
  $\rightarrow$ Regressionen werden erkannt
- Tests als „Quality Gateway“ für Integration von neuem Code in den Master Branch bzw. Master Repository

Automatisches Testen: JUnit

JUnit

- Erstveröffentlichung 2000
- eclipse-Unterstützung seit 2002
- Einbinden bsp. mit Maven

JUnit in Softwaretechnik

- Struktur und Beispiel in Vorlesung
- Details in Übung
- Eure Abgaben: JUnit-Tests gefordert (Quality-Gateway bei Korrektur)

Eine Beispiel-Funktion

die getestet werden sollte ...

```
package com.example.project;

public class Calculator {

        public int add(int a, int b) {
                return a - b;
        }

}
```

Schauen wir uns das einmal in eclipse an.

Eine Beispiel-Test

```
package com.example.project;

import static
        org.junit.jupiter.api.Assertions.assertEquals;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

class MyTest {

        @Test
        @DisplayName("1 + 1 = 2")
        void addsTwoNumbers() {
            Calculator calculator = new Calculator();
            assertEquals(2, calculator.add(1, 1),
                                    "1 + 1 should equal 2");
        }

}
```

https://github.com/junit-team/junit5-samples/

Automatisches Testen: Resümee

Evaluation

- Automatische Tests machen kaum zusätzliche Arbeit!
- Helfen, Qualitätsprobleme früh zu erkennen

Fragen

- Wie viele Tests sollte ich schreiben?
- Welche Teile des Programms sollten getestet werden?
- Und mit welchen Beispieldaten?
- Woher weiß ich die „erwarteten“ Ausgaben?
- ...

Später ...
Mehr dazu im Abschnitt „Qualitätssicherung“

Mocking: Motivation

Unit Testing

- Code verwendet andere Objekte des Systems
- Objekte müssen für Test verfügbar sein

Schwierigkeiten

- Unit-Testing! Kein komplettes System starten!
- Andere Objekte / Klassen eventuell noch nicht verfügbar!
- Performance
- Echtes Objekt schwierig bereitzustellen:
  - Spezifische Werte (Uhrzeit, Sensorwerte)
  - Fehlerzustände (Netzwerkfehler, . . .)

Beispiele

- Time-Server
- Datenbank
- Temperatur-Sensor
- Netzwerkdienste

Mocking ist insbesondere für „Test Driven Development“ relevant.

Ansatz: Mock-Objekte

Alternative: Mock-Objekte (auch: Test Double)

- Objekte, die echtes Verhalten nachahmen
- Können im Test anstelle des echten Objekts verwendet werden

Interface

- Müssen anstelle des „echten Objekts“ verwendet werden können
  → Gleiches Interface (oder Teilmenge) wie echtes Objekt

Verhalten

- „Angelehnt“ an echtes Objekt
  → für zu testenden Code: wie echtes Objekt
- Explizites Herbeiführen von Werten, Zuständen, ...
  → für Test-Code: konfigurierbar!

Mock-Objekte: Spektrum

Einfachste Version: Stub-Objekte

- Liefert immer feste Antworten auf Anfragen
- Antworten können eventuell vom Test-Code gesetzt werden

...

Komplexeste Version: Fake-Objekte

- Funktionierende Implementierung
- Einfacher als tatsächlich verwendetes Objekt

Mock-Objekte: Kategorien

`https://martinfowler.com/articles/mocksArentStubs.html`

Dummy objects are passed around but never actually used. Usually they are just used to fill parameter lists.

Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production (an in-memory database is a good example).

Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

Spies are stubs that also record some information based on how they were called.

Mocks are [...] objects pre-programmed with expectations which form a specification of the calls they are expected to receive.

Framework: Mockito

Übersicht

- `https://site.mockito.org/`
- Java-Framework zum Erstellen von Mock-Objekten
- Erste Version: 2008

If you use Mockito in tests you typically:

1. Mock away external dependencies and insert the mocks into the code under test
2. Execute the code under test
3. Verify that the code executed correctly

For example, you can verify that a method has been called with certain parameters. This kind of testing is sometimes called behavior testing. Behavior testing does not check the result of a method call, but it checks that a method is called with the right parameters.

Mockito mit JUnit I

```
import static org.mockito.Mockito.*;

public class MockitoTest {

  @Mock
  MyDatabase databaseMock;                                    // 1

  @Rule public MockitoRule mockitoRule
                          = MockitoJUnit.rule();              // 2

  @Test
  public void testQuery() {
    ClassToTest t = new ClassToTest(databaseMock);            // 3
    boolean check = t.query("* from t");                      // 4
    assertTrue(check);                                        // 5
    verify(databaseMock).query("* from t");                   // 6
  }
}
```

Mockito mit JUnit II

1. Tells Mockito to mock the databaseMock instance
2. Tells Mockito to create the mocks based on the `@Mock` annotation
3. Instantiates the class under test using the created mock
4. Executes some code of the class under test
5. Asserts that the method call returned true
6. Verify that the query method was called on the `MyDatabase` mock

Siehe auch `https://www.vogella.com/tutorials/Mockito/article.html`

Mocking: Resümee

Motivation: Komplexe Objekte für Test „ersetzen“

- Simulieren
- Durch Dummies ersetzen
- Zusätzlich Aufruf-Informationen sammeln

Verhältnis zu Unit-Testing

- Ergänzt assertions durch Verification: Testen von Interaktionen mit dem Mock-Objekt

Verifikation

- Mock-Objekte speichern Aufrufinformationen. Dies erlaubt eine Überprüfung von Eigenschaften der *Aufruf-History* der Mock-Objekte
- **Achtung**: Nicht verwechseln mit formaler Verifikation!

Unit Tests *vs.* Mock-Verification?
Ergänzung, nicht Ersatz: Überprüfen unterschiedliche Aspekte!
Mehr Details und Beispiele: Übung!

Testen zur Verifikation

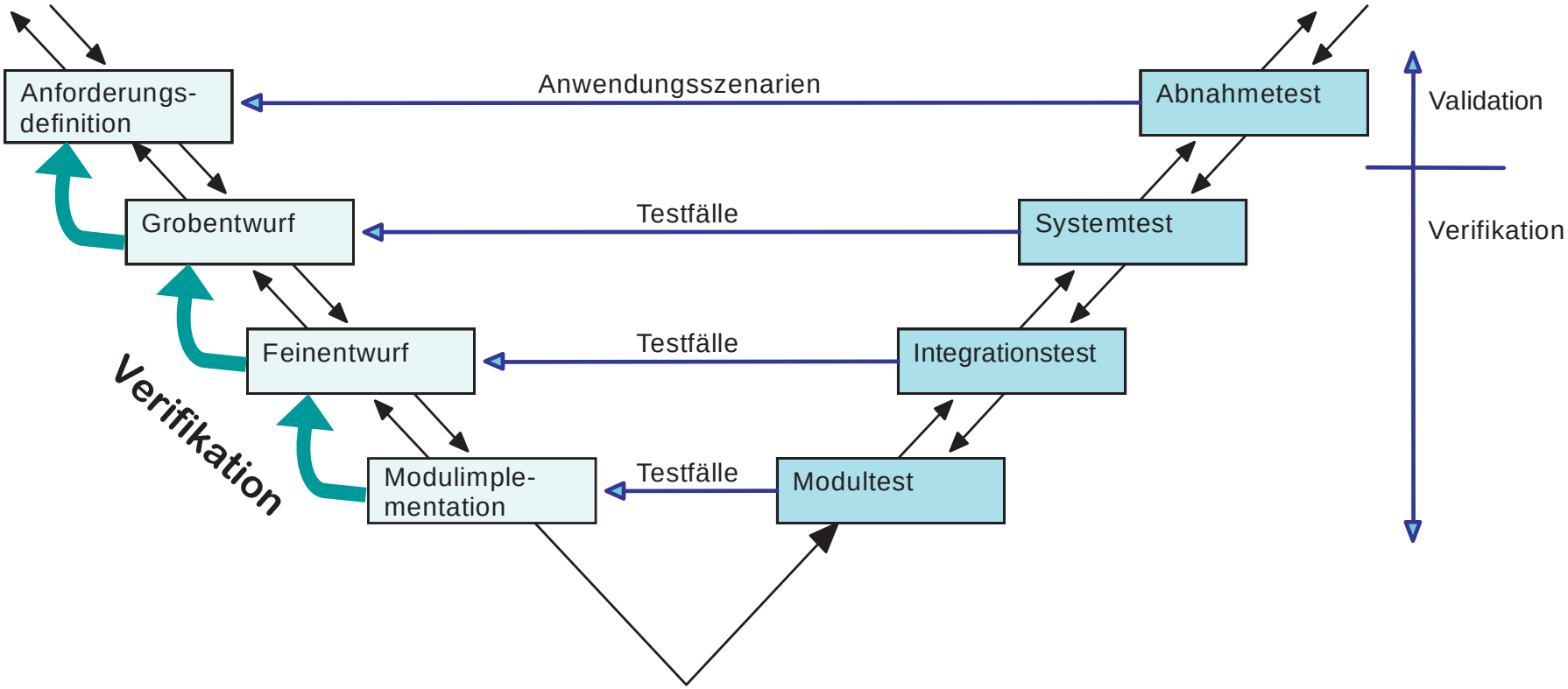
Unit-Test	Einzelne Funktionen / Komponenten werden getestet.
Integrationstest	Abhängige Komponenten werden gemeinsam getestet. Betonung auf Schnittstellentest. Komponenten werden zu Subsystemen integriert und gemeinsam getestet.
Systemtest	Test des Gesamtsystems

Ziel einer Testplanung

Testfälle so auswählen, dass die Wahrscheinlichkeit groß ist, Fehler zu finden oder diese auszuschließen.

Das V-Modell

Nutzung insbesondere bei der Bundeswehr und Behörden



Verifikation und Validation der Teilprodukte sind wesentliche Bestandteile des V-Modells.

Probleme beim Testen von Software

- Testfälle beschreiben das Verhalten meist genauer als die Spezifikation:
  $\rightarrow$ Testfälle vor Implementierung entwickeln
- Grundsätzliches Problem:
  "Program testing can be used to show the presence of bugs, but never to show their absence." [Dijkstra 1972]
- Zu beantwortende Fragen:
  - Wie geeignete Testfälle und Testpläne bestimmen?
  - Wie Korrektheit der Testdurchführung überprüfen?
  - Wer testet?

Herausforderungen der Schritte im Testprozess

- Identifikation von Testszenarien
  - Was ist zu Testen?
- Festlegung konkreter Testszenarien
  - Wie wird der Testling ausgeführt?
  - Welches Verhalten ist korrekt?
- Formalisierung von Tests
  - Wie wird der Test spezifiziert?
- Ausführung von Tests
  - Automatisierung!
- Pflege und Wartung von Tests
  - Synchronisiert mit den Änderungen der fachlichen Software müssen (zumeist) die Tests angepasst und geändert werden.

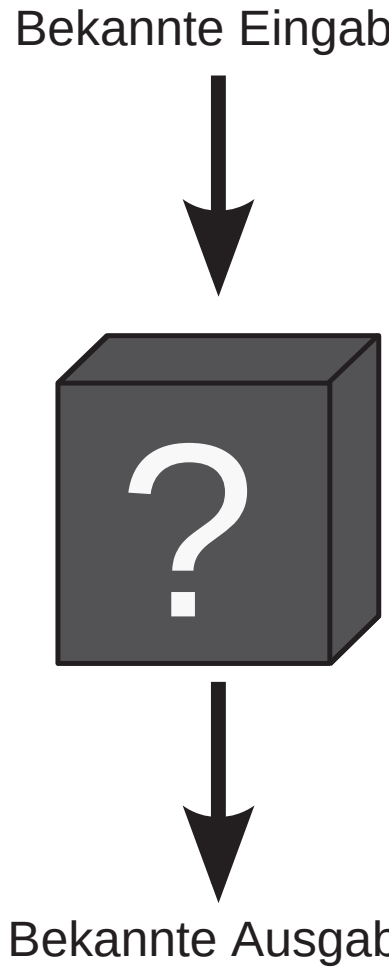
Zielvorgaben beim Testen von Software

- Es muss klar sein, welche Resultate beim Testen erwartet werden.
- Einsatz *systematischer* Techniken:
  - Auswahl der Testfälle möglichst nicht (nur) intuitiv oder zufällig.
- Die Ergebnisse sollten reproduzierbar sein.
  - Das ist insbesondere beim Testen paralleler Programme ein großes Problem.
- Fehler sollten nicht nur erkannt, sondern auch lokalisiert und behoben werden.
  - ‚Known bugs‘ zeugen nicht von hoher Qualität.
- Qualitäts-Anforderungen müssen mit möglichst hoher Genauigkeit geprüft werden.

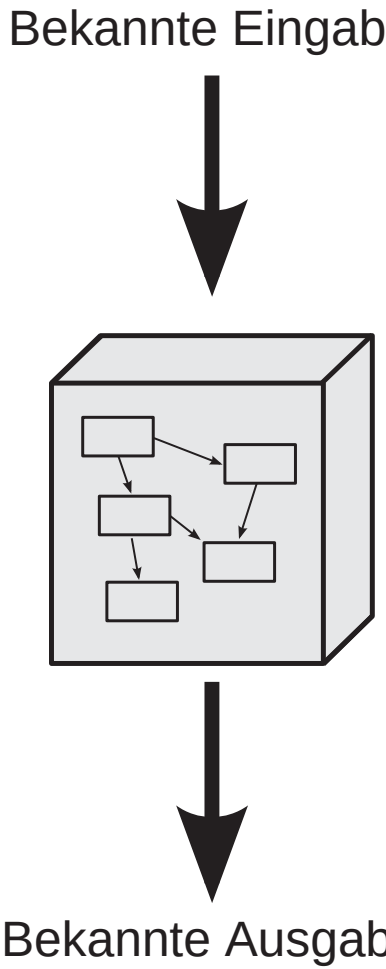
Pragmatische Sicht auf das Testen

- Testen erlaubt Aussagen über die Qualität eines Softwaresystems
- Quantitative Metriken stützen Aussagen
  - Testabdeckung etc.
- Automatisierte Regressionstests bieten eine Absicherung gegen Seiteneffekte bei Änderungen
  - Unter einem Regressionstest versteht man die Wiederholung von Testfällen, um sicherzustellen, dass Modifikationen in bereits getesteten Teilen der Software keine neuen Fehler ("Regressionen") verursachen.

Black Box vs. White Box Testen



Blackbox Test



Whitebox Test

White-box Testen von Modulen

(testing in the small)

C0: Anweisungsüberdeckung: Jede Anweisung soll durch eine geeignete Testmenge mindestens einmal durchlaufen werden (Anweisungsüberdeckungskriterium).

C1: Kantenüberdeckung: Jede Kante eines Kontrollflussgraphen soll mindestens einmal durchlaufen werden (Kantenüberdeckungskriterium).

C2: Pfadüberdeckung: Jeder Pfad vom Start- zum Endknoten soll durch eine geeignete Testmenge mindestens einmal durchlaufen werden (Pfadüberdeckungskriterium).

C3: Bedingungsüberdeckung: Jede mögliche Kombination der Teilbedingungen muss mindestens einmal durchlaufen werden (Bedingungsüberdeckungskriterium).

Probleme mit der Anweisungsüberdeckung

- Ist ein Programm ausreichend getestet, wenn jede Anweisung mindestens einmal ausgeführt wurde?
- Wie bestimmt man eine minimale Testmenge?
- Ist es sinnvoll, auch leere Anweisungen (z.B. fehlendes `else`) zu überdecken?
- Die Struktur des Programms bleibt unberücksichtigt
- Grundsätzlich:
  Überdeckung ist nicht entscheidbar!

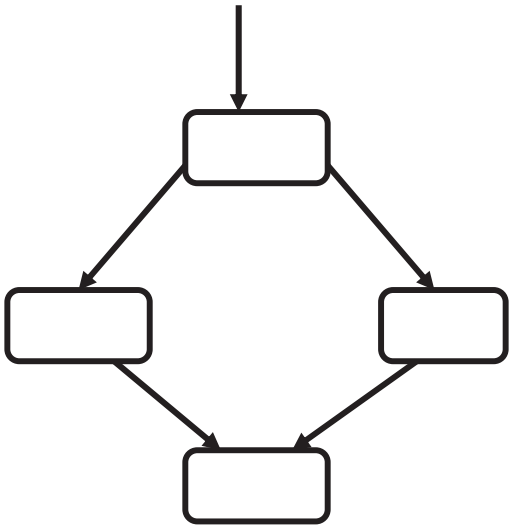
Überdeckung aller Kanten eines Kontrollflussgraphen

C1: Kantenüberdeckung

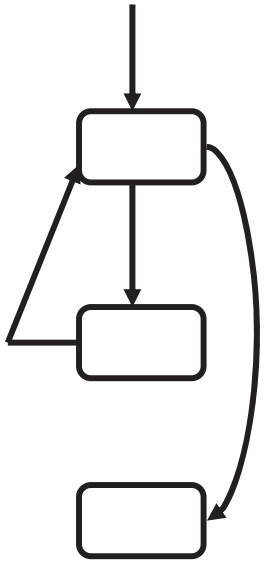
Voraussetzung ist die Konstruktion eines Kontrollflussgraphen für ein (imperatives) Programm:

- Jede einfache Anweisung, die keine weiteren Anweisungen enthält, wird als Knoten dargestellt.
- Bedingte Anweisungen werden als Verzweigungen dargestellt.
- Schleifen werden als Zyklen und
- Sequenzen durch Kanten zwischen Knoten dargestellt.

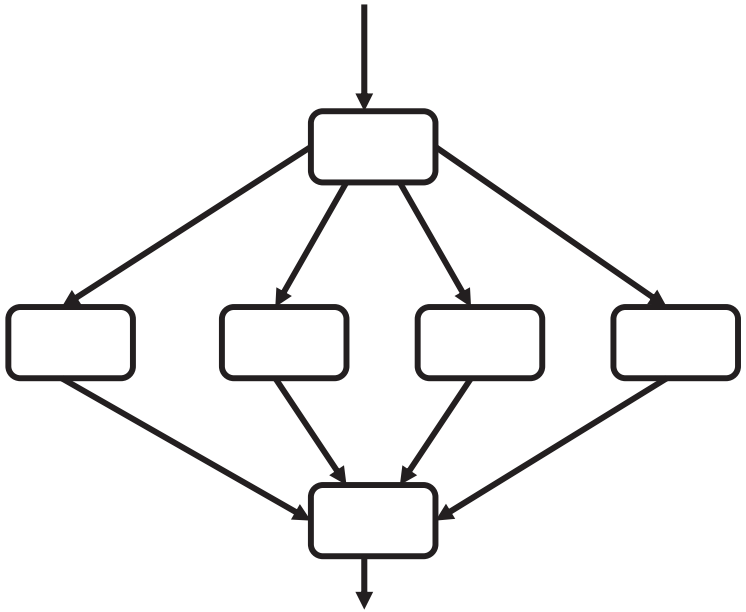
Konstruktion von Kontrollflussgraphen



If-then-else



while-do

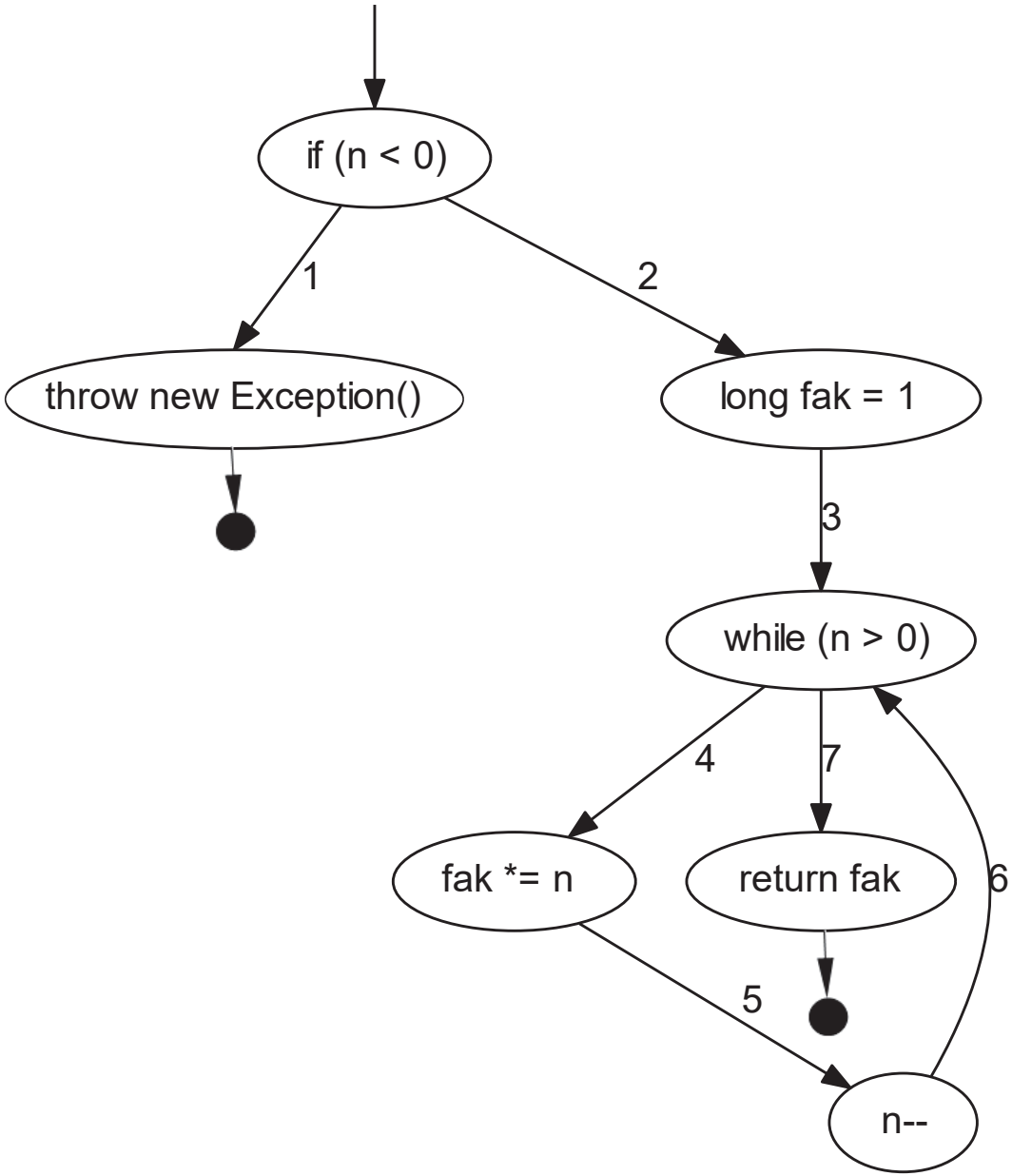


Case-of

Beispiel: Programm zur Fakultätsberechnung

```
public long fakultaet
             (int n) {
  if (n < 0) {
    throw new
      Exception();
  }
  long fak = 1;

  while (n > 0) {
    fak *= n;
    n--;
   }
  return fak;
}
```



Testfälle zur Kantenüberdeckung

Nr	Klasse	Beschreibung des Testfalls	Erwartetes Ergebnis	Beispielhafte Eingabedaten
1	Normal	Eingabeparameter ist gültig	Fakultät des Eingabeparameters	42
	Überprüfte Kanten: 2, 3, 4, 5, 6, 7			
2	Fehler	Eingabeparameter ist nicht gültig	Exception	-1
	Überprüfte Kanten: 1			

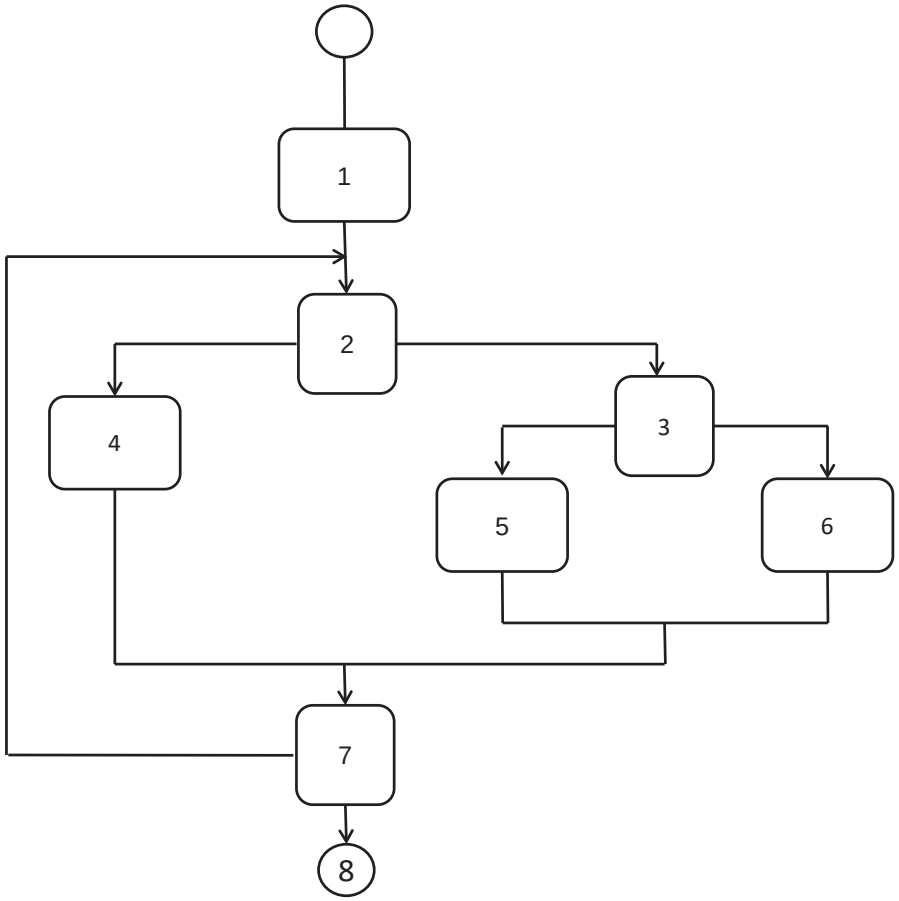
Zyklomatische Komplexität

McCabe [1976], Ebert and Cain [2016]

- Hinter dieser Software-Metrik von McCabe steckt der Gedanke, dass ab einer bestimmten Komplexität ein Modul für Menschen nicht mehr begreifbar ist.
- Die zyklomatische Komplexität ist definiert als die Anzahl unabhängiger Pfade auf dem Kontrollflussgraphen eines Moduls.
- Die zyklomatische Komplexität gibt an, wie viele Testfälle nötig sind, um eine Kantenüberdeckung zu erreichen.
- Das Komplexitätsmaß nach McCabe ist gleich der Anzahl der binären Verzweigungen plus 1.
- Laut McCabe sollte die zyklomatische Zahl eines in sich abgeschlossenen Teilprogramms nicht höher als 10 sein, da sonst das Programm zu komplex und zu schwer zu testen ist.

Zyklomatische Komplexität: Beispiel

McCabe [1976], Ebert and Cain [2016]



Path 1: 1,2,3,6,7,8
Path 2: 1,2,3,5,7,8
Path 3: 1,2,4,7,8
Path 4: 1,2,4,7,2,4,...7,8

Zyklomatische Komplexität $= 4$

Anmerkungen zum Kantenüberdeckungskriterium

- Es wird erzwungen, dass jede Bedingung jeweils mit true und false durchlaufen wird.
- Das Kantenüberdeckungskriterium ist somit *schärfer* als das Anweisungsüberdeckungskriterium.
- Gelegentlich jedoch nicht scharf genug:
  - Zusammengesetzte Bedingungen werden nicht hinreichend berücksichtigt.
- Unzureichend für den Test von Schleifen

Pfadüberdeckung

C2: Pfadüberdeckung

Auch bei einfachen Bedingungen kann die Kantenüberdeckung noch unzureichend sein:

```
if (x != 0) {
    y = 5;
}
else {
    y = 6;
}
if (z > 1) {
    z = z / x;
}
else {
    z = 0;
}
```

- Die Testmenge {(x=0, z=1), (x=1, z=3)} erfüllt zwar das Kantenüberdeckungskriterien, erkennt aber nicht die Division durch 0.
- Abhängigkeiten bleiben unerkannt!

Pfadüberdeckung (Fortsetzung)

- Die Testmenge
  {(x=0, z=1), (x=1, z=3), (x=0, z=3), (x=1, z=1)}
  erfüllt das Pfadüberdeckungskriterium.
- Problem: Die Anzahl der Pfade wächst exponentiell mit der Länge des Programmcodes, so dass ein Testen mit dem Pfadüberdeckungskriterium unrealistisch ist.
- Weiteres Problem: Anzahl Pfade für Schleifen.
  Lösungsansatz: Heuristiken.
  Beispielheuristik für die Anzahl von Schleifendurchläufen:
  - 0 mal
  - *mittelmäßig* oft
  - maximale Anzahl

  Grenzwerte sind wichtig!

Bedingungsüberdeckung

C3

Hierbei wird z.B.

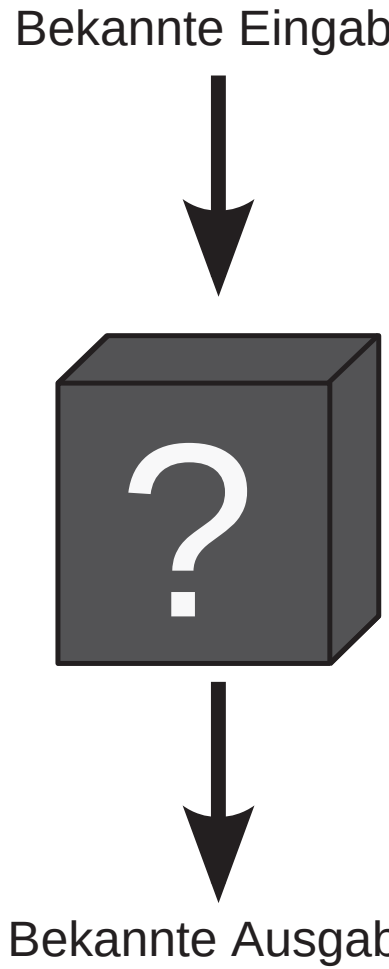
if (C1 and C2) ST;
else SF;

in die folgende Anweisung transformiert:

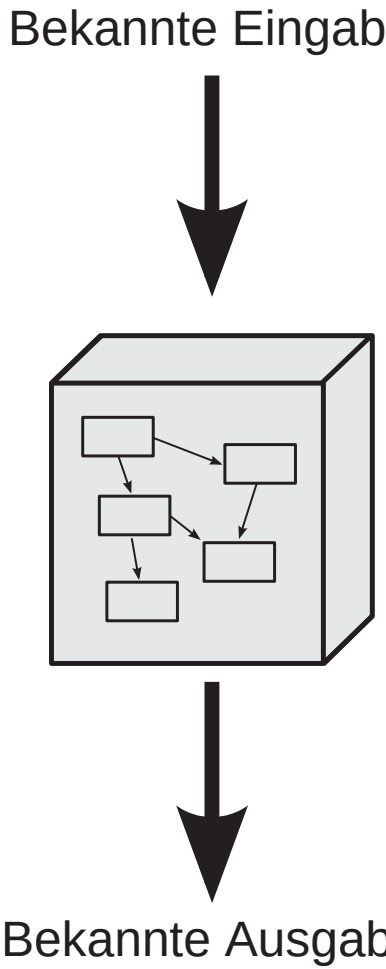
if (C1)
if (C2) ST
else SF
else SF

- Der neue Kontrollflussgraph überdeckt alle elementaren (Teil-)Bedingungen und macht damit *versteckte* Kanten sichtbar.
- Das Bedingungsüberdeckungskriterium ergibt somit ggf. noch schärfere Testmengen.

Black Box vs. White Box Testen



Blackbox Test



Whitebox Test

Blackbox Testen von Modulen

- Idee: Ein Programmteil wird ohne Ansicht des Codes getestet.
- Problem: Wie bestimmt man dann die Testmenge?
- Zur Bestimmung der Testmenge kann nur die Spezifikation herangezogen werden.
- Voraussetzung zur systematischen Generierung von Testmengen sind dann formale Spezifikationen.

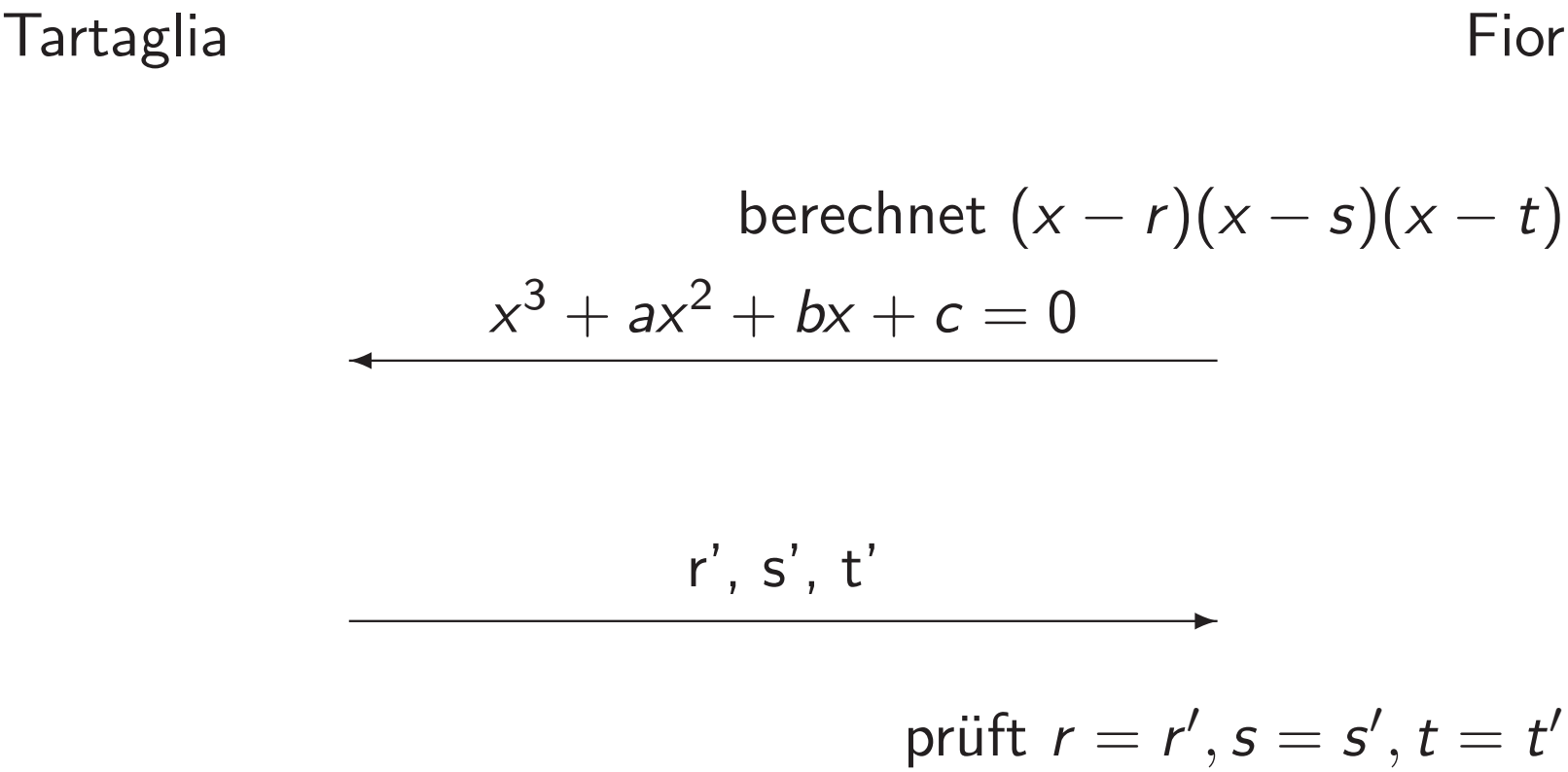
Mittelalterliches Beispiel

Lösungen für Gleichungen der Form:

$$x^3 + ax^2 + bx + c = 0$$

- 1535: Tartaglia hat eine Lösung gefunden, aber nicht verraten.
- Fior legte 30 Aufgaben zur Überprüfung vor.
- Siehe: [Wußing and Arnold 1975]

Mittelalterlicher Blackboxtext



Black Box Testverfahren

- Testfälle aus der Programmspezifikation ableiten.
- Programmstruktur wird nicht betrachtet.
- Möglichst umfassende, aber redundanzarme Prüfung der spezifizierten Funktionalität.
- Funktionsüberdeckung ist das Ziel
- Testfallbestimmung:
  - Funktionale Äquivalenzklassenbildung
  - Grenzwertanalyse
  - Test spezieller Werte

Funktionale Äquivalenzklassenbildung

Ziel
Definitionsbereiche der Eingabeparameter und Wertebereiche der Ausgabeparameter werden in Äquivalenzklassen zerlegt.

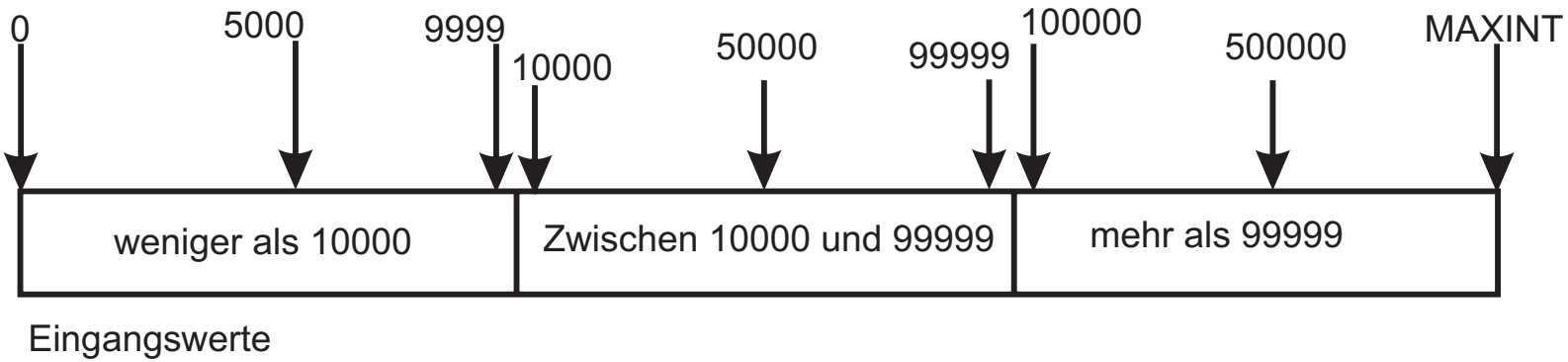
Annahme
Ein Programm reagiert bei der Verarbeitung eines Repräsentanten aus einer Äquivalenzklasse so, wie bei allen anderen Werten aus dieser Äquivalenzklasse.

Repräsentant für einen Testfall
Irgendein Element aus der Klasse auswählen.

Grenzwertanalyse

- Testfälle, die die Grenzwerte der Äquivalenzklassen abdecken oder in der unmittelbaren Umgebung dieser Grenzen liegen, decken besonders häufig Fehler auf.
- Es wird nicht irgendein Element aus der Äquivalenzklasse als Repräsentant ausgewählt.
- Es werden ein oder mehrere Elemente ausgesucht, so dass jeder Rand der Äquivalenzklasse getestet wird.

Äquivalenzklassen mit Grenzwerten



Spezifikation einer Suchfunktion

procedure Search (Key : ELEM ; T: ELEM_ARRAY;
Found : **in out** BOOLEAN; L: **in out** ELEM_INDEX) ;

Pre-condition

– the array has at least one element
T'FIRST <= T'LAST

Post-condition

– the element is found and is referenced by L
(Found and T (L) = Key)
or
– the element is not in the array
(**not** Found **and**
not (**exists** i, T'FIRST >= i <= T'LAST, T (i) = Key))

Äquivalenzklassenbildung

auf Basis der Spezifikation

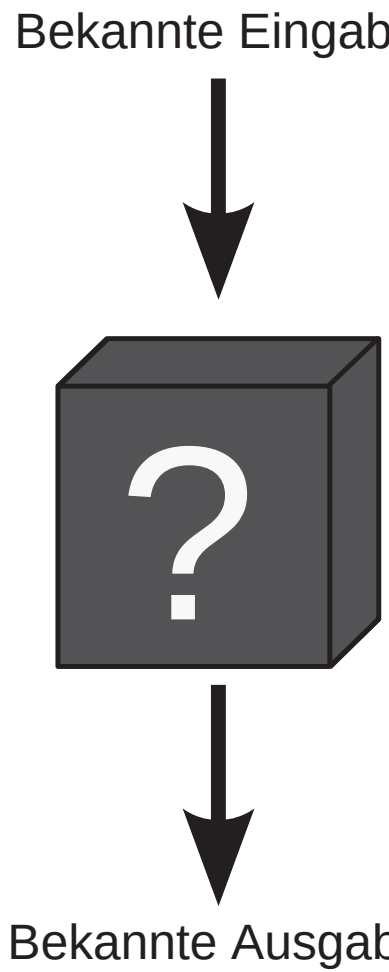
- Eingaben, die die Vorbedingung erfüllen
- Eingaben, die die Vorbedingung nicht erfüllen
- Eingaben, bei denen das Schlüsselelement im Array ist
- Eingaben, bei denen das Schlüsselelement nicht im Array ist
- Eingabemengen
  - Mit 0 Elementen
  - Mit 1 Element
  - Mit mehreren Elementen

Äquivalenzklassen für die Suchfunktion

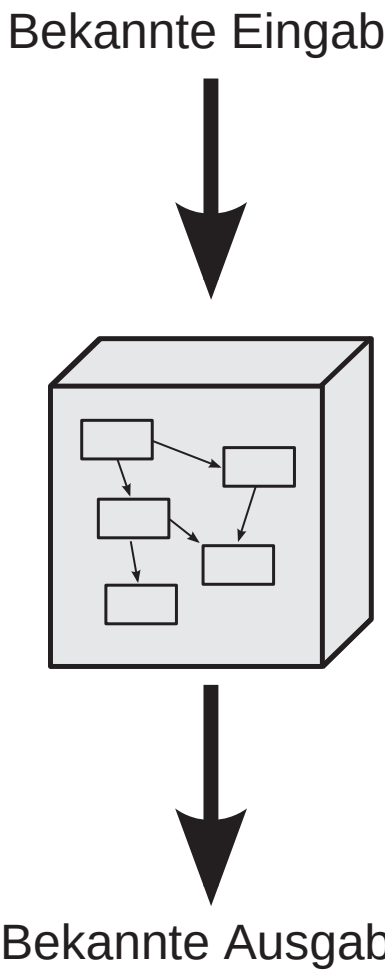
Array	**Element**
Single value	In sequence
Single value	Not in sequence
More than 1 value	First element in sequence
More than 1 value	Last element in sequence
More than 1 value	Middle element in sequence
More than 1 value	Not in sequence

Input sequence (T)	**Key**	**Output** (Found, L)
17	17	true, 1
17	0	false, *
17, 29, 21, 23	17	true, 1
41, 18, 9, 31, 30, 16, 45	45	true, 7
17, 18, 21, 23, 29, 41, 38	23	true, 4
21, 23, 29, 33, 38	25	false, *

Black Box und White Box Testen



Blackbox Test



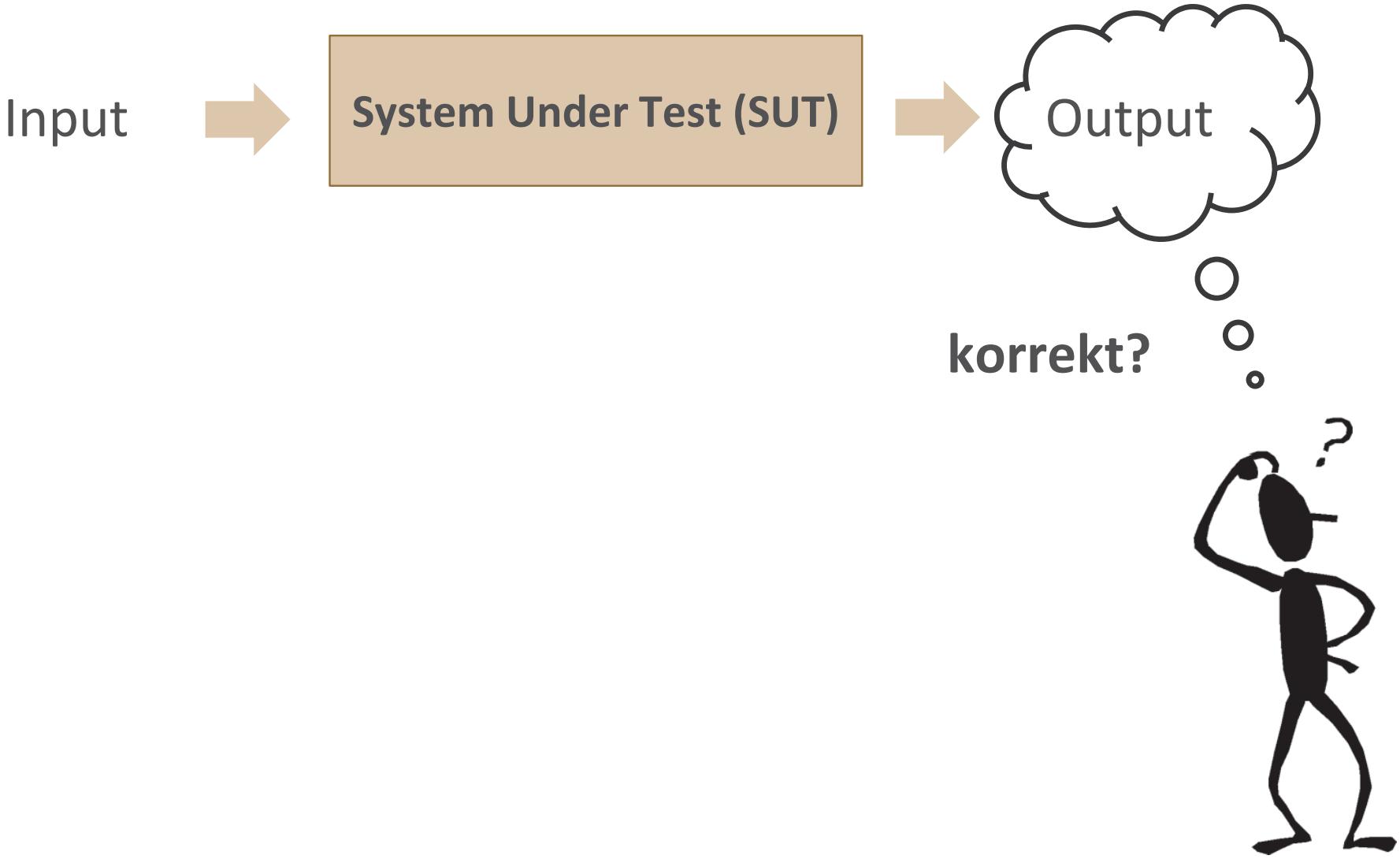
Whitebox Test

Fuzzy Testing

Fuzzing:

- Anders als bei Unit-Tests werden beim Fuzzy-Testing Testfälle nicht manuell definiert, sondern anhand statistischer Funktionen zufällig erzeugt.
- Durch eine hohe Anzahl der so generierten Tests wird die Software auch auf außergewöhnliche Eingabeparameter getestet.
  - Das ist insbesondere zur Aufdeckung von Sicherheitslücken interessant.
- Fuzzing wird in der Regel im Rahmen eines Black-Box-Tests durchgeführt, um neue Software auf Fehleranfälligkeit zu prüfen sowie um Sicherheitslücken aufzuspüren.
- Wenn das Programm bei bestimmten vom Fuzzer generierten Daten reproduzierbar ein Problem verursacht (z. B. abstürzt), kann darauf aufbauend anhand von White-Box-Tests die genaue Ursache gesucht werden.

Test-Orakel



Für welche Art von Software gibt es keine Test-Orakel?

https://monti.com

Das Orakel-Problem

Es ist nicht immer möglich, ein Orakel zu definieren

Was tun?

Segura et al. [2016; 2020], Kanewala and Yueh Chen [2019]

Anwendungsbereiche für metamorphisches Testen

Segura et al. [2020]

Integrationstest

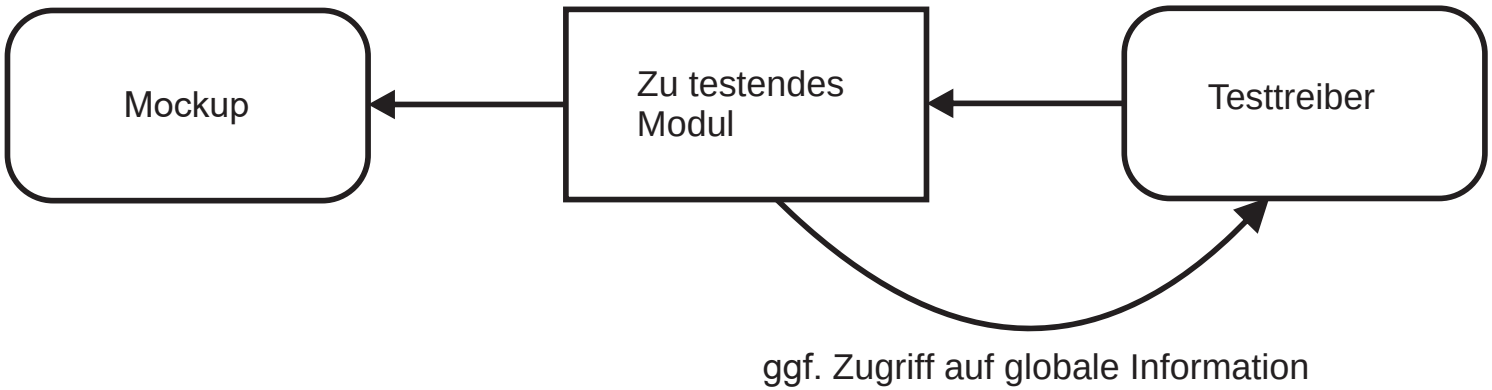
Testing in the large

- Der Test einzelner Module garantiert nicht, dass die Zusammenarbeit korrekt funktioniert.
- Viele Module können gar nicht isoliert getestet werden.
- Also: Vorgehensweise zum Test komplexer Systeme (modulares Testen):
  - Modultests (a.k.a. Unit Tests & Components Tests)
    Testen eine Komponente ohne Kontext.
  - Inkrementeller Integrationstest mehrer Komponenten im Kontext.
  - Systemtest: Test des gesamten Systems in der Anwendungsumgebung (end-to-end).
- Akzeptanztests (a.k.a. Funktionstests)
  Focus auf das Testen von ‘cross-cutting’ Funktionalität.

Integrationstest (Forts.)

- Bei eingebetteten Realzeitsystemen muss auch das Zusammenspiel von Hard- und Software getestet werden.
  - Digital Twins
- Inkrementelles Testen ist dem ‚Big-Bang'-Testen, wobei nach den Modultests direkt das Gesamtsystem getestet wird, vorzuziehen.
- Die Trennung zwischen Schnittstelle und Implementierung erleichtert den Integrationstest erheblich.
  - Leichtes Ersetzen von ‚Mockups' durch ‚richtige' Module. Siehe z.B. Mockito `https://github.com/mockito/`

Inkrementelles Testen



„Testgerüst“

- Inkrementelle Tests können entsprechend den Benutzt- und Kompositions-Hierarchien bottom-up oder top-down erfolgen; natürlich auch jo-jo.
- Eine hierarchische Architektur ist dabei sehr förderlich.

Regressionstest

- Testwerkzeug speichert alle durchgeführten Testfälle
- Erlaubt die automatische Wiederholung aller bereits durchgeführten Tests nach Änderungen des Prüflings
- Führt Soll/Ist-Ergebnisvergleich durch
  - Eingabedaten in das Testobjekt
  - Erwartete Ausgabedaten oder Ausgabereaktionen (Soll-Ergebnisse)

Testgetriebene Entwicklung und Refactoring

- Testgetriebene Entwicklung besteht aus der Kombination von zwei Techniken:
  - Testgetriebene Programmierung zur externen Evolution,
  - Refactoring zur internen Evolution.
- Refactoring
  Programm-Änderungen zur Verbesserung der internen Struktur, ohne die Funktionalität zu ändern.
- Ein Refactoring ist dann erlaubt, wenn alle Tests erfüllt sind.
- Tests sind hier auch eine Form von ausführbarer Anforderungsspezifikation.

Continuous Integration

Aufgaben, u.a.:

- Ausführung statischer Analysen (z.B. SpotBugs, Checkstyle)
- Bauen des Systems (z.B. Gradle)
- Automatisierte Ausführung von Tests (z.B. JUnit, Selenium) [Polo et al. 2013]
- Überprüfung von Log- und Monitoring-Daten (z.B. Kieker) [Waller et al. 2015]

Siehe Gastvorlesung von Christian Zirkelbach, Alexander Krause und Marcel Bader.

Literatur I

H. Balzert. *Lehrbuch der Softwaretechnik: Entwurf, Impementierung, Installation und Betrieb*. Spektrum Akademischer Verlag, 3. edition, 2011.

H. Benington. Production of large computer programs. In *Proceedings of the ONR Symposium on Advanced Program Methods for Digital Computers*, 1956.

E. V. Berard. *Essays on object-oriented software engineering*. Prentice Hall, Englewood Cliffs, N.J., 1993.

R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley, 1999.

J. Bloch. Nearly all binary searches and mergesorts are broken, 2006. http://googleresearch.blogspot.de/2006/06/extra-extra-read-all-about-it-nearly.html.

B. Boehm. *Software Engineering Economics*. Prentice-Hall, 1981.

B. Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, 1984.

B. Boehm. A spiral model of software development and enhancement. *IEEE Computer*, 21(5):61–72, 1988.

B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy. Using the WinWin spiral model: a case study. *IEEE Computer*, 31(7):33–44, 1998.

G. Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, 2nd edition, 1994.

G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1998.

G. Booch, R. Maksimchuk, M. Engel, B. Young, J. Conallen, and K. Houston. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 3nd edition, 2007. ISBN 0-201-89551-X.

C. Brindescu, M. Codoban, S. Shmarkatiuk, and D. Dig. How do centralized and distributed version control systems impact software changes? In Jalote et al. [2014], pages 322–333. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568322.

M. Broy and D. Rombach. Software engineering. *Informatik Spektrum*, 25(6):438–451, 2002.

B. Brügge and A. Dutoit. *Object Oriented Software Engineering Using UML, Patterns, and Java*. Prentice Hall, 3rd edition, 2010. ISBN 978-0138152215.

Bundestag. Verfügbarkeit der webbasierten hartz-iv-software a2ll, 2006. http://www.heise.de/newsticker/meldung/88318, http://dip.bundestag.de/btd/16/049/1604938.pdf.

Literatur II

J. Caumanns, H. Weber, A. Fellien, H. Kurrek, O. Boehm, J. Neuhaus, J. Kunsmann, and B. Struif. Die eGK-Lösungsarchitektur Architektur zur Unterstützung der Anwendungen der elektronischen Gesundheitskarte. *Informatik-Spektrum*, 29(5):341–348, Oct. 2006.

H. B. Christensen and K. M. Hansen. An empirical investigation of architectural prototyping. *Journal of Software and Systems*, 83(1):133–142, Jan. 2010. doi: 10.1016/j.jss.2009.07.049.

R. A. De Millo, R. J. Lipton, and A. J. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5):271–280, May 1979. doi: 10.1145/359104.359106.

T. DeMarco. *Controlling Software Projects*. Yourdon Press, New York, 1982.

S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann, 2003.

Detienne. *Software-Design: Cognitive Aspects*. Springer-Verlag, 2002.

E. Dijkstra. Notes on structured programming. In O. Dahl et al., editors, *Structured Programming*, pages 1–82. Academic Press, London, 1972.

C. Ebert and J. Cain. Cyclomatic complexity. *IEEE Software*, 33(6):27–29, Nov. 2016. doi: 10.1109/MS.2016.147.

J. Ehlers, A. van Hoorn, J. Waller, and W. Hasselbring. Self-adaptive software system monitoring for performance anomaly localization. In *Proceedings of the 8th IEEE/ACM International Conference on Autonomic Computing (ICAC 2011)*, pages 197–200. ACM, June 2011.

A. Endres and D. Rombach. *A Handbook of Software and Systems Engineering: Empirical Observations, Laws and Theories*. Addison Wesley, 2003.

M. S. Fisher. *Software Verification and Validation: An Engineering and Scientific Approach*. Springer-Verlag, 2007. ISBN 978-0-387-32725-9.

C. Floyd. A systematic look at prototyping. In R. Budde, K. Kuhlenkamp, L. Mathiassen, and H. Züllighoven, editors, *Approaches to Prototyping*, pages 1–18. Springer-Verlag, 1984.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0201485672.

Literatur III

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1996.

C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2nd edition, 2003.

T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

T. Grams. *Denkfallen und Programmierfehler*. Springer-Verlag, Berlin, 1990.

T. Grechenig, M. Bernhart, R. Breiteneder, and K. Kappel. *Softwaretechnik*. Pearson Studium, 2010.

D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

W. Hasselbring, editor. *Betriebliche Informationssysteme: Grid-basierte Integration und Orchestrierung*. GITO-Verlag, 2010. ISBN 978-3-942183-20-8.

W. Hasselbring and G. Steinacker. Microservice architectures for scalability, agility and reliability in e-commerce. In *Proceedings 2017 IEEE International Conference on Software Architecture Workshops (ICSA 2017)*, Gothenburg, Sweden, Apr. 2017. IEEE.

W. Hasselbring and A. van Hoorn. Kieker: A monitoring framework for software engineering research. *Software Impacts*, 5, June 2020. doi: 10.1016/j.simpa.2020.100019.

J. Hielscher, R. Kazhamiakin, A. Metzger, and M. Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In P. Mähönen, K. Pohl, and T. Priol, editors, *Towards a Service-Based Internet, First European Conference, ServiceWave 2008, Madrid, Spain, December 10-13, 2008. Proceedings*, volume 5377 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2008.

G. Hohpe, R. Wirfs-Brock, J. W. Yoder, and O. Zimmermann. Twenty years of patterns' impact. *Software, IEEE*, 30(6):88–88, Nov. 2013. doi: 10.1109/MS.2013.135.

G. J. Holzmann. The power of 10: Rules for developing safety-critical code. *IEEE Computer*, 39(6):95–97, June 2006.

G. J. Holzmann. Landing a spacecraft on mars. *Software, IEEE*, 30(2):83–86, 2013. doi: 10.1109/MS.2013.32.

IEEE. *Standard Glossary of Software Engineering*. The Institute of Electrical and Electronics Engineers, New York, ieee std 610.12-1990 edition, 1990.

Literatur IV

ISO/IEC 25010. *ISO/IEC 25010:2011 – Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models*. ISO/IEC, Mar. 2011.

I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering*. Addison-Wesley, 1992.

I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

P. Jalote, L. C. Briand, and A. van der Hoek, editors. *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014. ACM. ISBN 978-1-4503-2756-5.

J. Jézéquel and B. Meyer. Design by contract: The lessons of Ariane. *IEEE Computer*, 30(1):129–130, Jan. 1997.

U. Kanewala and T. Yueh Chen. Metamorphic Testing: A Simple Yet Effective Approach for Testing Scientific Software. *Computing in Science & Engineering*, 21(1):66–72, Jan. 2019. doi: 10.1109/MCSE.2018.2875368.

C. Kecher, A. Salvanos, and R. Hoffmann-Elbern. *UML 2.5: das umfassende Handbuch*. Rheinwerk Verlag, 6., aktualisierte auflage edition, 2018. OCLC: 1004662469.

J. Kerievsky. *Refactoring to Patterns*. Addison-Wesley Professional, 2004. ISBN 0321213351.

S. Koolmanojwong and B. Boehm. The incremental commitment model process patterns for rapid-fielding projects. In *New Modeling Concepts for Today's Software Processes*, pages 150–162. Springer, 2010.

J. Koskinen. Software maintenance costs. Technical report, School of Computing, University of Eastern Finland, Joensuu, Finland, Apr. 2015. https://wiki.uef.fi/display/tktWiki/Jussi+Koskinen.

J. Kramer. Is abstraction the key to computing? *Commun. ACM*, 50(4):36–42, 2007. doi: 10.1145/1232743.1232745.

N. S. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In A. Winter, R. Ferenc, and J. Knodel, editors, *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 47–57. IEEE, Mar. 2009. ISBN 978-0-7695-3589-0. doi: 10.1109/CSMR.2009.15.

T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

Literatur V

A. Metzger, O. Sammodi, K. Pohl, and M. Rzepka. Towards pro-active adaptation with confidence – augmenting service monitoring with online testing. In *Proceedings of the ICSE 2010 Workshop on Software Engineering for Adaptive and Self-managing Systems (SEAMS)*, Cape Town, South Africa, 2-8 May 2010.

B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 2nd edition edition, 2001.

K. Muslu, C. Bird, N. Nagappan, and J. Czerwonka. Transition from centralized to decentralized version control systems: a case study on reasons, barriers, and outcomes. In Jalote et al. [2014], pages 334–344. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568284.

B. Oestereich and A. Scheithauer. *Analyse und Design mit UML 2.5.1: Objektorientierte Softwareentwicklung*. De Gruyter, 12., neu nach essence 1.2 strukturierte und ergänzte auflage edition, 2019. ISBN 978-3-11-062909-5.

I. Ozkaya. Ethics is a software design concern. *IEEE Software*, 36(3):4–8, 2019. doi: 10.1109/MS.2019.2902592.

D. L. Parnas. *Software Engineering or Methods for the Multi-Person Construction of Multi-Version Programs*, volume 23 of *Lecture Notes in Computer Science, Programming Methodology*. Springer-Verlag, 1974. ISBN 3-540-07131-8.

D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering (ICSE 1994)*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. ISBN 0-8186-5855-X.

D. L. Parnas. Software engineering: Multi-person development of multi-version programs. In C. B. Jones and J. L. Lloyd, editors, *Dependable and Historic Computing*, volume 6875 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2011. ISBN 978-3-642-24540-4. doi: 10.1007/978-3-642-24541-1_31.

K. Pohl. *Requirements Engineering*. dpunkt Verlag, 2. edition, 2008.

M. Polo, P. Reales, M. Piattini, and C. Ebert. Test automation. *Software, IEEE*, 30(1):84–89, Jan. 2013. doi: 10.1109/MS.2013.15.

R. Potvin and J. Levenberg. Why Google stores billions of lines of code in a single repository. *Commun. ACM*, 59 (7):78–87, June 2016. doi: 10.1145/2854146.

R. Preißel and B. Stachmann. *Git: Dezentrale Versionsverwaltung im Team – Grundlagen und Workflows*. dpunkt, 4., aktualisierte und erweiterte auflage edition, 2017.

Literatur VI

R. Reussner and W. Hasselbring, editors. *Handbuch der Software-Architektur*. dpunkt Verlag, 2. edition, 2008.

S. Roock and M. Lippert. *Refactorings in großen Softwareprojekten: Komplexe Restrukturierungen erfolgreich durchführen*. dpunkt Verlag, 2004. ISBN 13 978-3-89864-207-1.

W. Royce. Managing the development of large software systems: concepts and techniques. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 328–338, 1987.

J. Rumbaugh, G. Booch, and I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

B. Rumpe. *Modellierung mit der UML*. Springer-Verlag, 2. auflage edition, 2011.

N. B. Ruparelia. Software development lifecycle models. *SIGSOFT Softw. Eng. Notes*, 35(3):8–13, 2010. doi: 10.1145/1764810.1764814.

C. Rupp, S. Queins, et al. *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. Hanser, 4. auflage edition, 2012. ISBN 978-3-446-43057-0.

S. Segura, G. Fraser, A. B. Sanchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016. doi: 10.1109/TSE.2016.2532875.

S. Segura, D. Towey, Z. Q. Zhou, and T. Y. Chen. Metamorphic Testing: Testing the Untestable. *IEEE Software*, 37(3):46–53, May 2020. doi: 10.1109/MS.2018.2875968.

M. Seidl, M. Brandsteidl, C. Huemer, and G. Kappel. *UML@Classroom: Eine Einführung in die objekt-orientierte ModellierungWork*. dpunkt Verlag, 2012.

I. Sommerville. *Software Engineering*. Pearson, 10. auflage edition, 2018. ISBN 978-3-86326-835-0. `http://software-engineering-book.com`.

H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, Wien, 1973.

W. F. Tichy. RCS – A system for version control. *Software – Practice and Experience*, 15(7):637–654, July 1985.

W. F. Tichy. Tools for software configuration management. In J. F. H. Winkler, editor, *Proceedings of the International Workshop on Software Version and Configuration Control (SCM), January 27-29, 1988, Grassau, Germany*, volume 30 of *Berichte des German Chapter of the ACM*, pages 1–20. Teubner, Jan. 1988. ISBN 3-519-02671-6.

Literatur VII

UML 2.5. *OMG Unified Modeling Language Version 2.5*, Mar. 2015. `http://www.omg.org/spec/UML/2.5/`.

UML Infrastructure. *OMG Unified Modeling Language Infrastructure Version 2.2*, Feb. 2009. `http://www.omg.org/spec/UML/2.2/Infrastructure/PDF/`.

UML Superstructure. *OMG Unified Modeling Language Superstructure Version 2.2*, Feb. 2009. `http://www.omg.org/spec/UML/2.2/Superstructure/PDF/`.

A. van Hoorn, M. Rohr, I. A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In N. Medvidovic and T. Tamai, editors, *Proc. of the Warm Up Workshop (WUP 2009) for ACM/IEEE ICSE 2010*, pages 37–40. ACM, Apr. 2009a. ISBN 978-1-60558-565-9.

A. van Hoorn, M. Rohr, and W. Hasselbring. Engineering and continuously operating self-adaptive software systems: Required design decisions. In *Design for Future – Langlebige Softwaresysteme*, pages 52–63. CEUR Workshop Proceedings, Oct. 2009b.

J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable DevOps. *SIGSOFT Softw. Eng. Notes*, 40(2):1–4, Mar. 2015. doi: 10.1145/2735399.2735416.

J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley, second edition, 2003.

G. M. Weinberg. *The Psychology of Computer Programming*. Dorset House, 1998.

K. Wiegers. *Peer Reviews in Software*. Addison-Wesley, 2002.

P. Y. H. Wong and J. Gibbons. A Process Semantics for BPMN. In *Formal Methods and Software Engineering*, volume 5256, pages 355–374. Springer Berlin Heidelberg, 2008. doi: 10.1007/978-3-540-88194-0_22.

H. Wußing and W. Arnold. *Biographien bedeutender Mathematiker*. Aulis Verlag Deubner & Co., Köln, 1975.