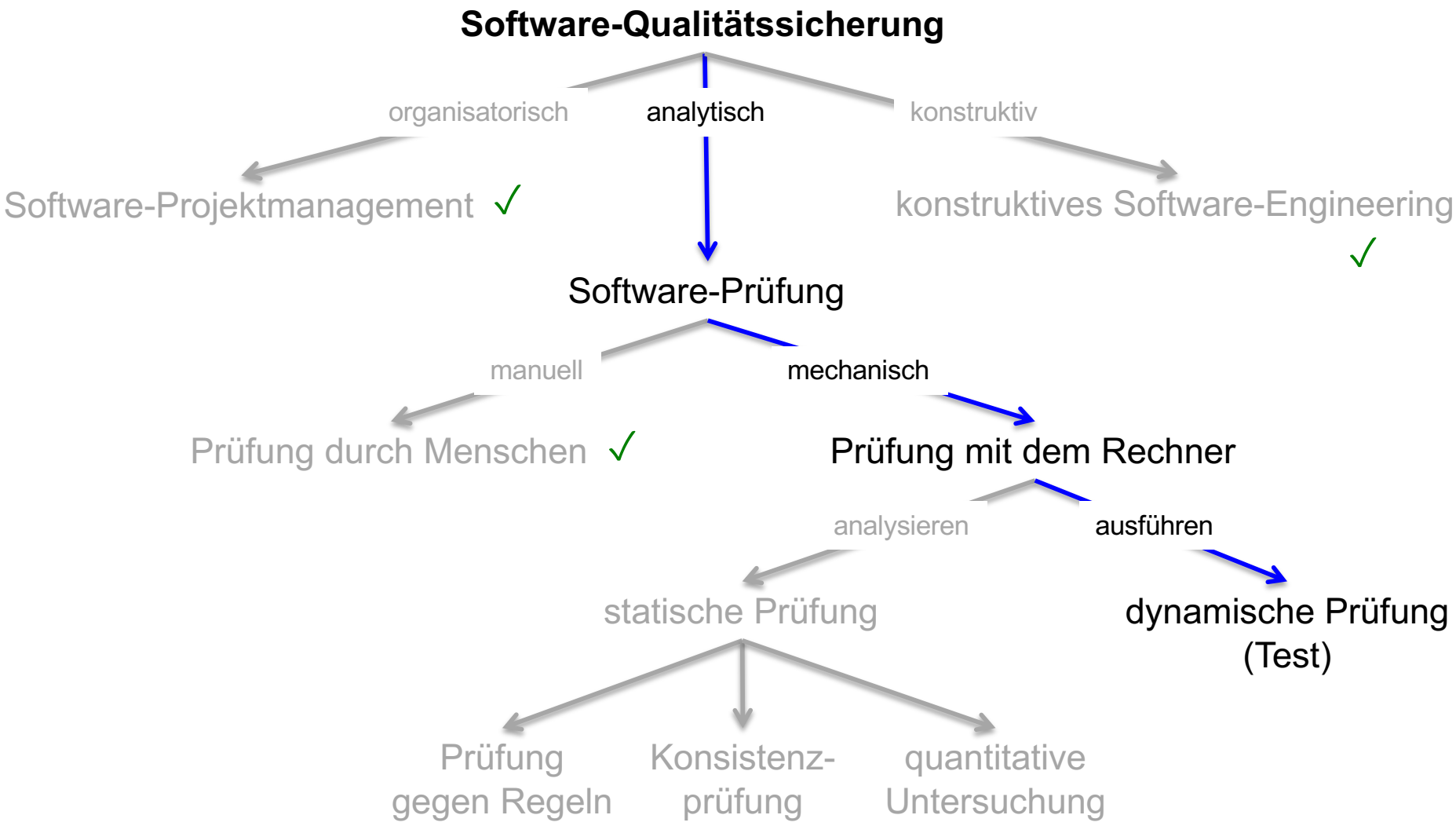
# Programmtest

## Dynamische Prüfung




**Gliederung der Software-Qualitätssicherung** nach [LL10]

## Programmtest

- Wir müssen die Fehlerfreiheit des fertigen Software-Systems nachweisen
- Aufgrund des erforderlichen Aufwands sind wir dazu in der Praxis nicht in der Lage
- Unser Ziel ist es daher, möglichst viele der vorhandenen Fehler durch Programmtests zu finden

> ***Testen*** *ist die Ausführung eines Programms auf einem Rechner mit dem Ziel, Fehler zu finden.*

nach [LL10]

- Das übersetzte Programm wird dazu ausgeführt und es werden konkrete Werte eingegeben
- Die Ausgaben (Ist-Werte) werden mit den Soll-Werten (aus der Spezifikation abgeleitet) verglichen
- Bei einer Abweichung zwischen Ist- und Soll-Wert liegt ein Fehler vor

- Ein spontanes Ausprobieren von Eingaben ist weder effizient noch sonderlich effektiv

- Wir konzentrieren uns daher auf **systematische Tests**:
  1. Randbedingungen (z.B. Systemumgebung) sind definiert
  2. Eingaben werden systematisch ausgewählt
  3. Soll-Werte werden vor dem Test festgelegt
  4. Der Testverlauf wird dokumentiert
  5. Test und Korrektur finden getrennt statt

- Ein systematischer Test ist reproduzierbar (vorausgesetzt, der Startzustand ist reproduzierbar). Damit ist der Test objektiv und wiederholbar

- Tests lassen sich nach der Prüfebene in verschiedene Teststufen einteilen
  - Unit-Test (Modultest, Komponententest)
  - Integrationstest
  - Systemtest
  - Abnahmetest

- Unit-Test (Modultest, Komponententest)
  - Es werden Programmteile getestet, die eine ausreichende Größe für einen eigenständigen Test haben
    - Funktionen/Methoden
    - Klassen
    - Komponenten / Verbund von Klassen
  - Wird in der Implementierung durchgeführt
  - Findet in der Entwicklungsumgebung statt

- Ein Testtreiber versorgt das Modul über Aufrufe der Schnittstelle mit Testdaten und nimmt die Antwort des Moduls entgegen
- Im Mittelpunkt steht der funktionale Test
- Optional
  - Test auf Robustheit
  - Test auf Effizienz

– Integration
  - Nachdem einzelne Module / Komponenten fertiggestellt und getestet wurden, müssen sie zu einem lauffähigen System zusammengebaut werden

> ***integration*** – *The process of combining software components, hardware components, or both into an overall system.*

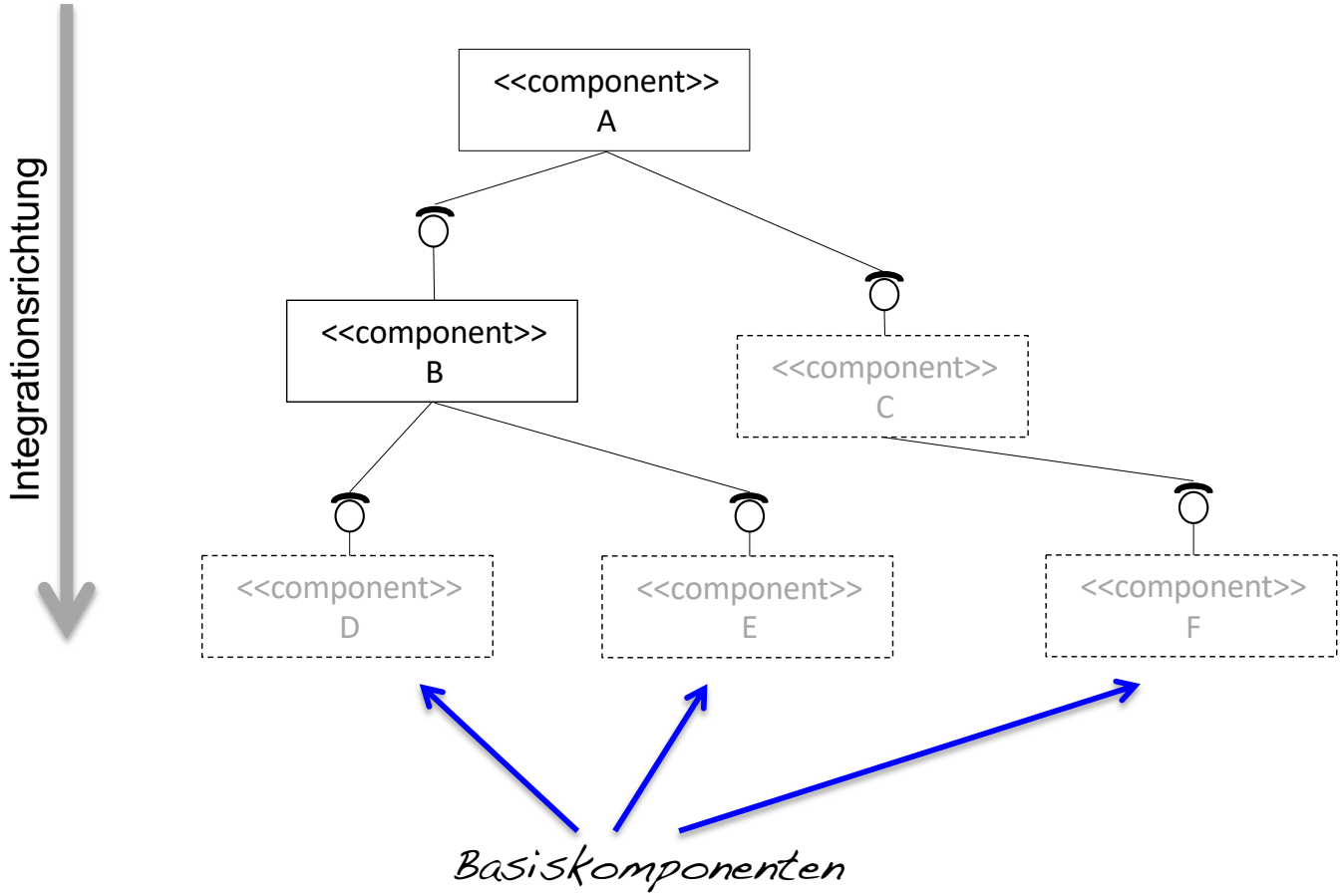
IEEE Std 610.12 (1990)

- Der Aufwand für die Integration darf nicht unterschätzt werden, da Fehler an den Schnittstellen auftreten können
- Der Grund für diese Fehler ist häufig eine unvollständige oder inkonsistente Spezifikation
- Bezüglich des Ablaufs unterscheidet man
  - die Integration in einem Schritt (Big-Bang-Integration)
  - die Inkrementelle Integration

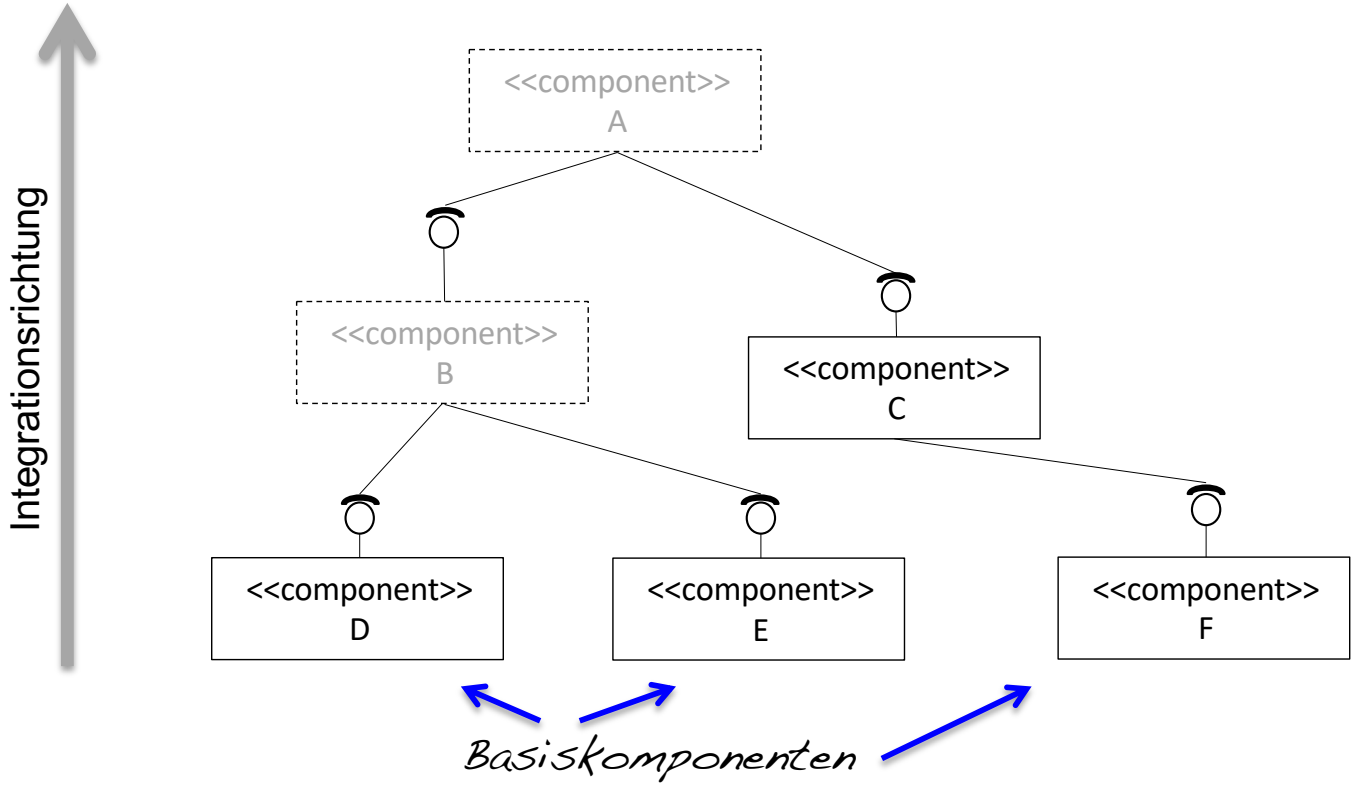
*Warum kann in der Regel die Integration in einem Schritt in der Praxis nicht durchgeführt werden?*

- Die inkrementelle Integration kann weiter differenziert werden
  - Top-down-Integration
  - Bottom-up-Integration
  - Outside-In-Integration
  - Kontinuierliche Integration

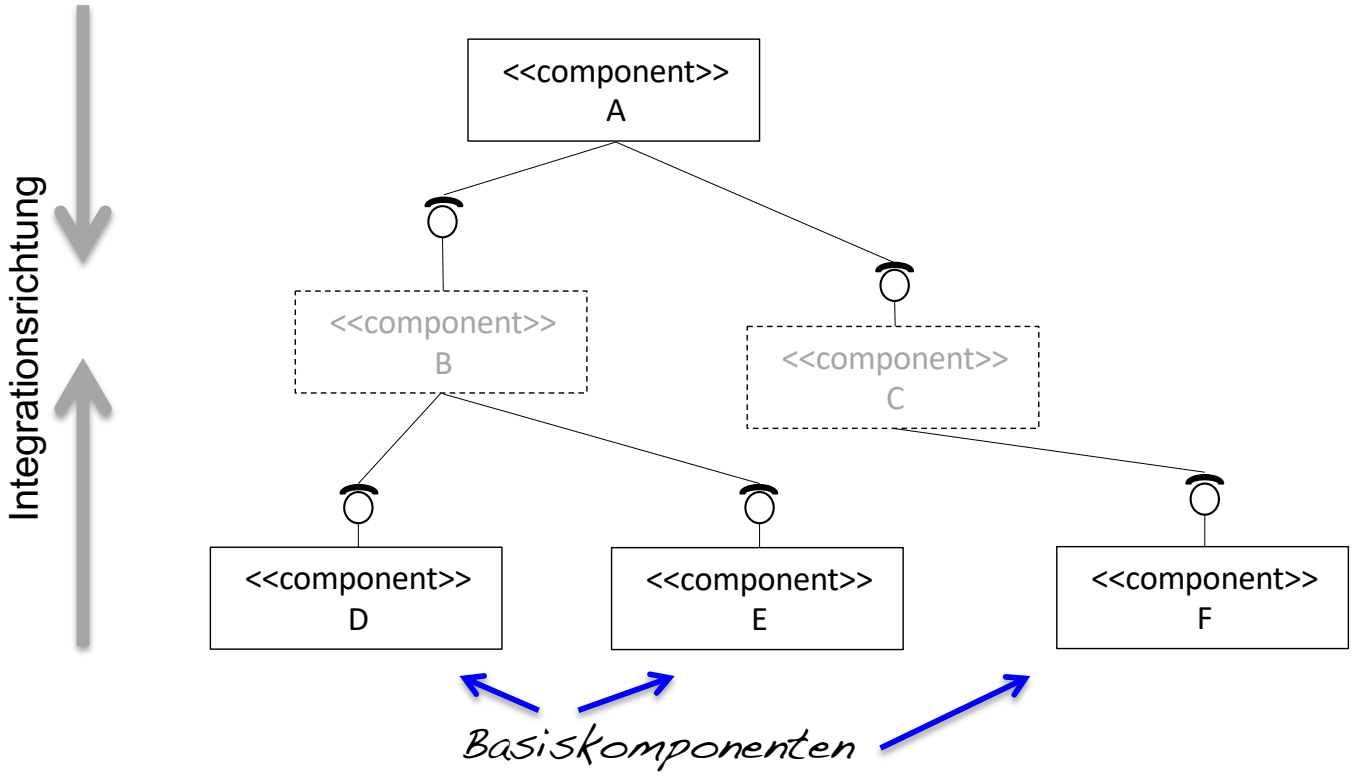
- Top-down-Integration
  - Die Integration beginnt auf der höchsten Hierarchieebene und arbeitet sich nach unten vor



- Bottom-up-Integration
  - Es wird mit den Komponenten begonnen, die keine Dienstleistungen anderer Komponenten benötigen (bis auf Betriebssystem-Dienste)
  - Danach werden die Komponenten integriert, die auf diese Basiskomponenten zugreifen



- Outside-in-Integration
  - Es werden die Komponenten der obersten und der untersten Schicht zuerst integriert
  - Danach arbeitet man sich von beiden Seiten zur Mitte vor

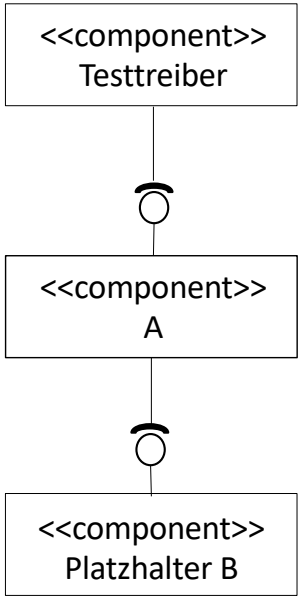


- Kontinuierliche Integration
  - Jede Komponente wird sofort integriert (siehe XP)
  - Es soll immer eine lauffähige (Teil-)Version in der Integrationsumgebung zur Verfügung stehen

- Auch die Integration muss geplant werden (Aufwand, Ablauf)
  - Termingetriebene Integration
  - Risikogetriebene Integration
  - Testgetriebene Integration

- Integrationstest
  - Es wird getestet, ob das Zusammenspiel der integrierten Module / Komponenten funktioniert
  - An den Schnittstellen können verschiedene Fehler auftreten
    - Inkompatible Schnittstellenformate
    - Protokollfehler
    - Semantische Fehler (unterschiedliche Einheiten)
    - Timing-Probleme
    - Kapazitäts- und Lastprobleme

- Falls eine Komponente *A* getestet werden soll, wird ggf. ein **Testtreiber** benötigt, der die Schnittstelle der Komponente *A* mit Testdaten versorgt
- Falls die Komponente *A* Dienste einer Komponente *B* nutzt, die noch nicht integriert ist, wird ein **Platzhalter** (*stub*) für *B* benötigt
- Der Platzhalter vertritt die fehlende Komponente und liefert entweder konstante Werte, oder simuliert das Verhalten der späteren Komponente in Ausschnitten



- Der Aufwand für die Erstellung von Platzhaltern und Testtreibern muss bei der Planung der Integration und der Integrationstests berücksichtigt werden

– Systemtest
  - Test des komplett integrierten Systems
  - Test aus Sicht des Anwenders
  - Validation, ob die Anforderungen vollständig und angemessen umgesetzt wurden
  - Testumgebung sollte der späteren Produktivumgebung möglichst ähnlich sein
  - Testtreiber und Platzhalter sind durch reale Komponenten ersetzt

- Abnahmetest
  - Erfolgt als letzter Test vor der Inbetriebnahme
  - Die Abnahmekriterien sind im Entwicklungsvertrag festgeschrieben (Individualsoftware)
  - Kunde ist in die Festlegung der Akzeptanztests involviert
  - Test aus Anwendersicht in der Abnahmeumgebung des Kunden

## Grenzen von Programmtests

- Falls es für ein Programm *n* mögliche Eingaben gibt, und wir alle *n* Eingaben testen, dann haben wir einen vollständigen Test durchgeführt
- Treten bei einem vollständigen Test keine Fehler auf, dann ist die Korrektheit des Programms nachgewiesen

- Beispiel:
  - Wir wollen einen vollständigen Test für die Methode
    `public static long abs(long a)`
    der Klasse `java.lang.Math` durchführen
  - Da eine `long`-Variable eine Breite von 64-Bit hat, gibt es $2^{64}$ Testfälle
  - Falls ein Test nur eine Mikrosekunde dauern würde, müssten wir über 500000 Jahre auf den Nachweis der Korrektheit warten

nach [Hof13]

- Wir müssen also mit einer kleinen Teilmenge von Eingaben auskommen

> *Programm testing can be used to show the presence of bugs, but never to show their absence!*

E.W. Dijkstra (1970)

- Um mit den wenigen Eingaben trotzdem viele Fehler finden zu können, müssen wir die Eingaben geeignet wählen

- Die Festlegung der Eingabedaten reicht für einen systematischen Test nicht aus

- Ein **Testfall** beschreibt
  - die Ausgangssituation (Vorbedingungen, Randbedingungen)
  - Testdaten, die die vollständige Ausführung des Testobjekts bewirken
  - Sollwerte (das erwartete Ergebnis bzw. Verhalten)

- Die Sollwerte eines Testfalls werden aus der Spezifikation abgeleitet
- Oft können Testszenarien als Folge von Testfällen gebildet werden
- Ein guter Testfall sollte die folgenden Eigenschaften besitzen:
  - fehlersensitiv (er zeigt mit hoher Wahrscheinlichkeit einen Fehler an)
  - repräsentativ (er steht stellvertretend für viele andere Testfälle)
  - redundanzarm (er überdeckt keine anderen Testfälle)

## Black-Box-Test (Funktionstest)

*Die Testfälle werden auf Basis der in der Spezifikation geforderten Eigenschaften gewählt.*
*Die innere Beschaffenheit des Programms spielt keine Rolle.*

- Der Black-Box-Test ist die wichtigste Form des dynamischen Tests
- Die Testfälle können/sollen bereits aufgestellt werden, sobald die entsprechende Spezifikation vorliegt
- Wir betrachten die folgenden Verfahren zur Bestimmung von Testfällen:
  ① Funktionale Äquivalenzklassenbildung
  ② Grenzwertanalyse
  ③ Zustandsbezogener Test
  ④ Test spezieller Werte
  ⑤ Zufallstest

1. Funktionale Äquivalenzklassenbildung
   - Angenommen, wir haben eine Menge $E=\{e_1,e_2,...,e_n\}$ von Eingaben, für die sich das Testobjekt gleich verhält
   - Dann reicht es aus, nur eine Eingabe aus dieser Menge zu testen (z.B. $e_1$)
   - Wenn diese Eingabe fehlerfrei ist, dann kann man davon ausgehen, dass auch die anderen Eingaben korrekt funktionieren
   - Eine solche Menge von Eingaben wird auch als Äquivalenzklasse bezeichnet
   - Jede Eingabe aus einer Äquivalenzklasse ist also repräsentativ für alle anderen Eingaben aus der Äquivalenzklasse
   - **Hinweis:** Der Begriff Äquivalenzklasse ist nicht im streng mathematischen Sinn zu sehen. In der Praxis können wir häufig nur vermuten, dass Eingaben äquivalent sind. Man spricht dann auch von schwacher Äquivalenz

## Aufgabe: Welche Äquivalenzklassen lassen sich für die Methode `long abs(long a)` bilden?

- Neben gültigen Äquivalenzklassen gibt es aber auch ungültige Äquivalenzklassen (evtl. leer)
- Die ungültigen Äquivalenzklassen enthalten ungültige Eingabewerte
- Es ist natürlich auch zu prüfen, wie sich das Testobjekt bei ungültigen Eingaben verhält
  a) Angemessene Reaktion auf die ungültige Eingabe (Meldung, mit Möglichkeit der Korrektur)
  b) Ignorieren der ungültigen Eingabe
  c) Automatische Korrektur der ungültigen Eingabe
  d) Fehlberechnung
  e) Programmabbruch

} unerwünschtes Verhalten

- Beispiele für die Bildung von gültigen und ungültigen Äquivalenzklassen

| Eingabe | gültige Äquivalenzklasse | ungültige Äquivalenzklasse |
|---|---|---|
| eine ganze Zahl x zwischen 1 und 31 | ganze Zahl x mit $1 \leq x \leq 31$ | 1. ganze Zahl $x < 1$<br>2. ganze Zahl $x > 31$ |
| Zeichenketten, die mit einem großen Buchstaben beginnen | Zeichenketten, die mit einem großen Buchstaben beginnen | 1. Zeichenketten, die mit einem kleinen Buchstaben beginnen<br>2. Zeichenketten, die mit einem Sonderzeichen beginnen<br>3. Zeichenketten, die mit einer Zahl beginnen |
| Rot, Grün, Blau | {Rot, Grün, Blau} | Eingabe nicht aus {Rot, Grün, Blau} |

- Konstruktion von Testfällen
  - Für jeden Eingabeparameter werden Äquivalenzklassen gebildet
  - Für jeden einzelnen Parameter gibt es mindestens zwei Äquivalenzklassen (eine gültige und eine ungültige)
  - Die Repräsentanten aller gültiger Äquivalenzklassen werden zu Testfällen kombiniert. Alle Kombinationen sind zu bilden
  - Ein Repräsentant einer ungültigen Äquivalenzklasse wird nur mit gültigen Repräsentanten kombiniert. Pro ungültige Äquivalenzklasse (mindestens) 1 Testfall
  - Zu jedem Testfall wird der Sollwert bestimmt

- Aufgabe: An der Schnittstelle einer Komponente können die drei Parameter *a*, *b* und *c* übergeben werden. Für *a* wurden vier Äquivalenzklassen gebildet. Davon sind zwei Äquivalenzklassen ungültig. Für *b* existieren zwei gültige und eine ungültige Äquivalenzklasse. Für *c* existiert eine gültige und eine ungültige Äquivalenzklasse. Wie viele Testfälle sind nach der obigen Regel zu konstruieren?

- Abhängig von der Anzahl der Parameter und der (gültigen) Äquivalenzklassen kann die Anzahl der betrachteten Testfälle sehr groß werden
- Es gibt verschiedene Möglichkeiten, die Anzahl der Testfälle geeignet zu reduzieren
  - Äquivalenzklassen verschiedener Parameter, die sich überlagern, können verschmolzen werden
  - Typische Testfälle bevorzugen
  - Paarweise Kombination
  - Jeder Repräsentant einer Äquivalenzklasse kommt in mindestens einem Testfall vor

- Ein Testende-Kriterium kann über eine zu erreichende Äquivalenzklassen-Überdeckung vorgegeben werden

  *Überdeckung* = (*Anzahl getesteter ÄK* / *Gesamtzahl ÄK*) x 100

2. Grenzwertanalyse
   - Falls die Elemente einer Äquivalenzklasse eine natürliche Ordnung besitzen, dann sollten Elemente aus der Äquivalenzklasse gewählt werden, die an den Grenzen liegen
   - Die Erfahrung hat gezeigt, dass Eingaben, die die Grenzen von Äquivalenzklassen bilden, häufig Fehler aufdecken
   - Die Annäherung an die Grenzen kann vom gültigen und vom ungültigen Bereich der Äquivalenzklasse aus erfolgen
   - In einigen Situationen können Fehler nur gefunden werden, wenn die Grenzen mit drei Werten geprüft werden
     ① der exakte Grenzwert
     ② der zum Grenzwert benachbarte Wert innerhalb der Äquivalenzklasse
     ③ der zum Grenzwert benachbarte Wert außerhalb der Äquivalenzklasse

## Übungsaufgabe

Die Methode

`public static boolean istErstesHalbjahr(int monat)`

bekommt einen Monat als ganze Zahl übergeben und entscheidet, ob es sich um einen Monat aus dem ersten Halbjahr handelt. Bilden Sie die Äquivalenzklassen und bestimmten Sie die Testfälle mit Hilfe der Grenzwertanalyse.

Setzen Sie an den markierten Stellen im Quellcode fehlerhafte Vergleichsoperatoren (<,>,<=,>=,!=,==) ein. Mit welchem Testfall wird der Fehler erkannt?

```
public static boolean istErstesHalbjahr(int monat){
   if ((monat □ 1) || (monat □ 12)) throw new IllegalArgumentException();
   if(monat □ 7) return true;
   return false;
}
```