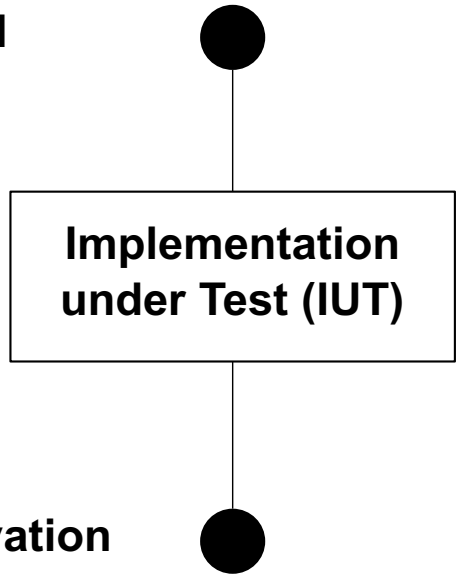
Test objektorientierter Programme

Dynamische Prüfung

Test objektorientierter Programme

– Die bisher betrachteten funktionalen Testverfahren waren auf die prozedurale Programmierung ausgerichtet



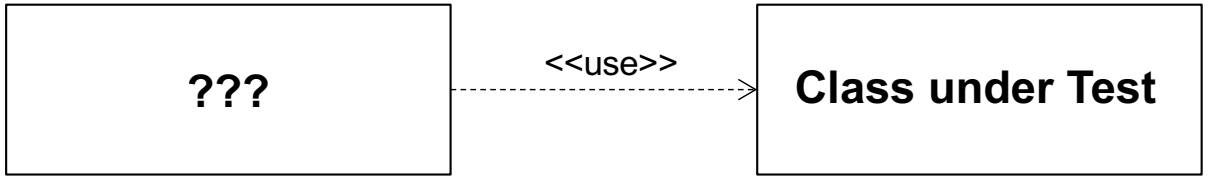

```
int q = quadrat(2);
```

```
int quadrat(int a){
   return a*a;
}
```

```
assertEquals(4, q);
```

- Bei objektorientierten Programmen sind zusätzliche Eigenschaften zu beachten
  - Das Ergebnis einer Methode kann zusätzlich zu den Parametern auch vom Zustand des Objekts (den Attributen) abhängen (Objekt ist immer implizit ein Parameter)
  - Parameter und Rückgabewerte können Objekte sein
  - Vererbung (überschreiben von Methoden) spielt eine Rolle
  - Assoziationen (Aggregation, Komposition)
  - Polymorphismus

- Wiederverwendbarkeit von Klassen erschwert das Testen
  - Einsatzzweck von Klassen nicht immer klar definiert
  - Allgemeinheit führt zu vielen möglichen Testfällen



  - Die Reihenfolge der Methodenaufrufe kann über den Zustand das Ergebnis beeinflussen
  - Aufrufreihenfolge bei Verwendung in einem Framework unbekannt

- Es sind die verschiedenen Arten von Klassen zu unterscheiden
  - normale Klassen
  - abstrakte Klassen und Schnittstellen
  - parametrisierte (generische) Klassen
  - Unterklassen

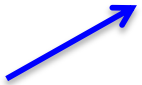
- Elemente der Objektorientierung erschweren das Testen
  - Vererbung von Attributen und Methoden
    - Redundanz wird eliminiert zu Lasten von zusätzlichen Abhängigkeiten
  - Polymorphismus und dynamische Bindung
    - Test jeder möglichen Bindung nötig

– Testen normaler Klassen

Addierer
- summe : int
+ setSumme(n : int) + getSumme() : int + add(n : int)

Diskutieren Sie das Testen von getter- und setter-Methoden

- Test zustandsverändernder Methoden
  1. Testfälle herleiten
  2. Operationen testen, die den Objektzustand nicht ändern
  3. Operationen testen, die den Objektzustand ändern
     Zustandsräume sind lokal an Objekt gebunden. Initialisierung und Auswertung der Tests erfolgt deshalb am Objekt
  4. Jede Folge abhängiger Operationen in der gleichen Klasse testen

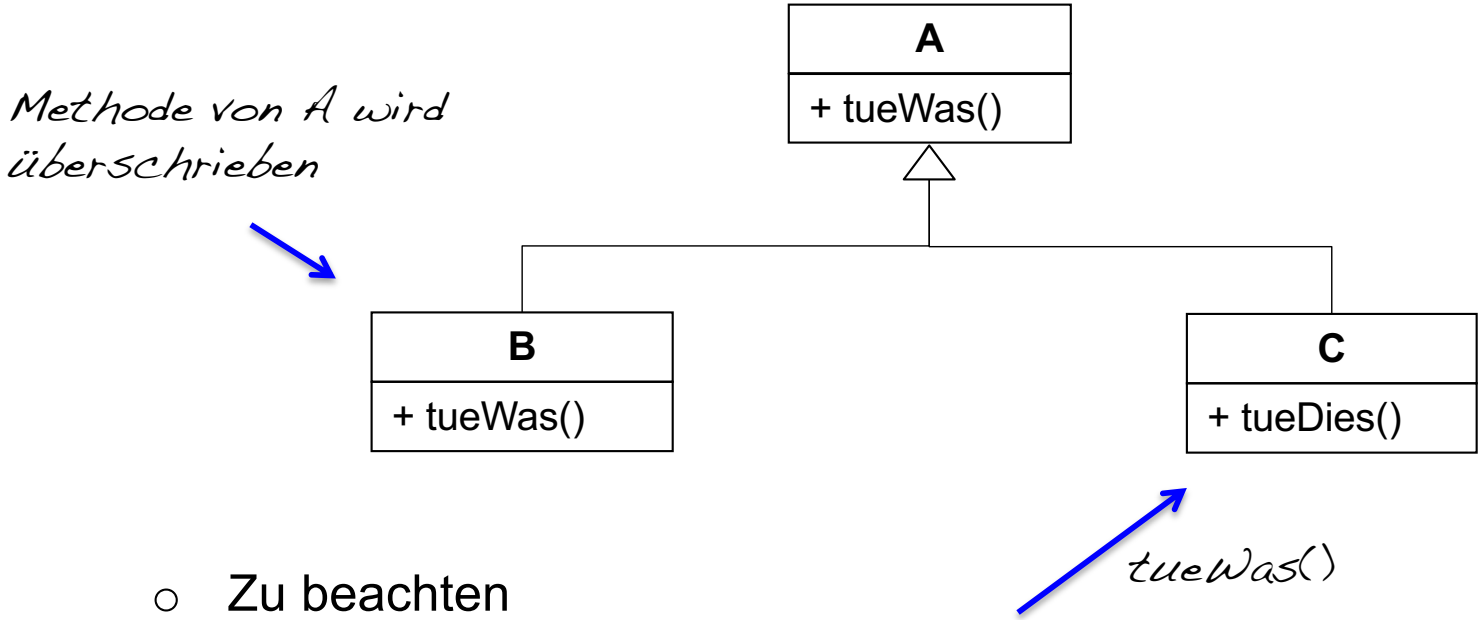


die Verwendung einer Operation sollte unter allen praktisch relevanten Bedingungen getestet werden

alle Äquivalenzklassen von Objektzuständen sind zu betrachten

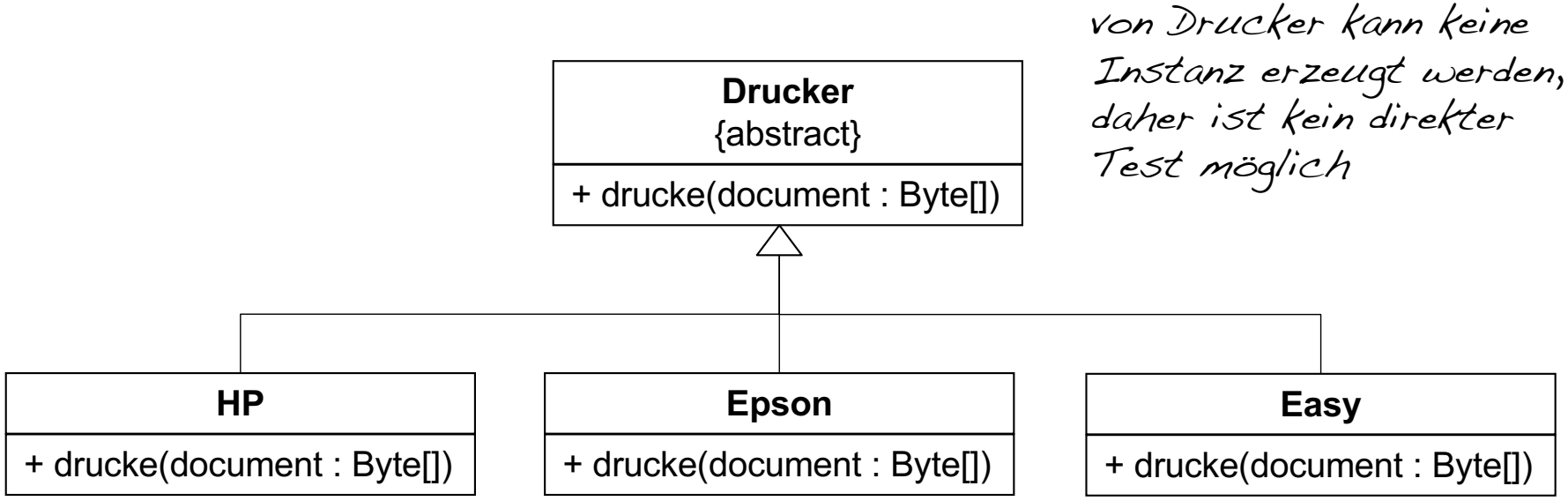
- Testen des Objektlebenszyklus
  - Falls Operationen abhängig vom Objektzustand sind, dann kann die Reihenfolge der Aufrufe bedeutend sein
  - Es sind dann ein zustandsbezogener Test mit den entsprechenden Überdeckungen durchzuführen
  - Verschiedene Laufzeitfehler lassen sich durch eine Datenflussanalyse aufdecken
    - Array-Grenzen
    - Division durch Null
    - Typecast
    - I/O
  - Eine Datenflussanalyse muss über Objektgrenzen hinweg erfolgen
  - Polymorphismus
    - Aufrufsignatur (dynamische Bindung)
    - Aufrufreihenfolge
    - Wertebereiche

- Testen von Unterklassen
  1. Systematischer Test aller Methoden der Oberklasse
  2. Systematischer Test aller Unterklassen



  - Zu beachten
    - Alle Testfälle für geerbte und nicht redefinierte Operationen der Oberklasse müssen erneut ausgewertet werden (Unterklasse definiert neuen Kontext)
    - Für redefinierte Operationen sind vollständig neue Testfälle zu erstellen

- Testen abstrakter Klassen
  - Möglichst einfache Unterklasse testen, die die abstrakte Klasse implementiert



- Testen nicht sichtbarer Bestandteilte
  - Test privater Methoden
  - Überprüfen privater Attribute (Kontrolle des Zustandes)

  - Alternativen
    - Es wird nur über die (öffentliche) Schnittstelle einer Klasse getestet. Ein Test der privaten Methoden und eine Kontrolle des Zustandes findet nur indirekt statt.
    - Lockerung der Sichtbarkeit auf Paketebene (*package scope*). Erfordert entsprechende Richtlinie für die Einstellung von Sichtbarkeiten. Testen im gleichen Paket ist möglich
    - Extra Testzugang (z.B. innere Testklassen)
    - Zugriff über Reflection

- Verwendung von Reflection für das Testen von privaten Attributen und Methoden

```
public class Calculator {
   private int methodCalls = 0;

   private boolean istNull(int i){
      methodCalls++;
      return (i==0);
   }

   public int divide(int n, int d){
      methodCalls++;
      if(istNull(d)) throw new ArithmeticException("Division durch 0");
      return n/d;
   }
}
```

Zustand soll geprüft werden

private Methode soll getestet werden

```
public class CalculatorTest {
   private Calculator cal;

   @Before
   public void erzeugeCalculator(){
      cal = new Calculator();
   }

   @Test
   public void testState(){
      final Class<?> clazz = cal.getClass();
      try {
         final Field privateAttr = clazz.getDeclaredField("methodCalls");
         privateAttr.setAccessible(true);
         assertEquals(0, ((Integer)privateAttr.get(cal)).intValue());
      } catch (Exception e) {
            e.printStackTrace();
      }
   }

   // Testmethode für private Methode auf der nächsten Seite
}
```

Achtung: Attributname muss bei Änderung angepasst werden

Differenziertes Exception-Handling erforderlich: NoSuchFieldException, SecurityException, IllegalArgumentExceptio, IllegalAccessException

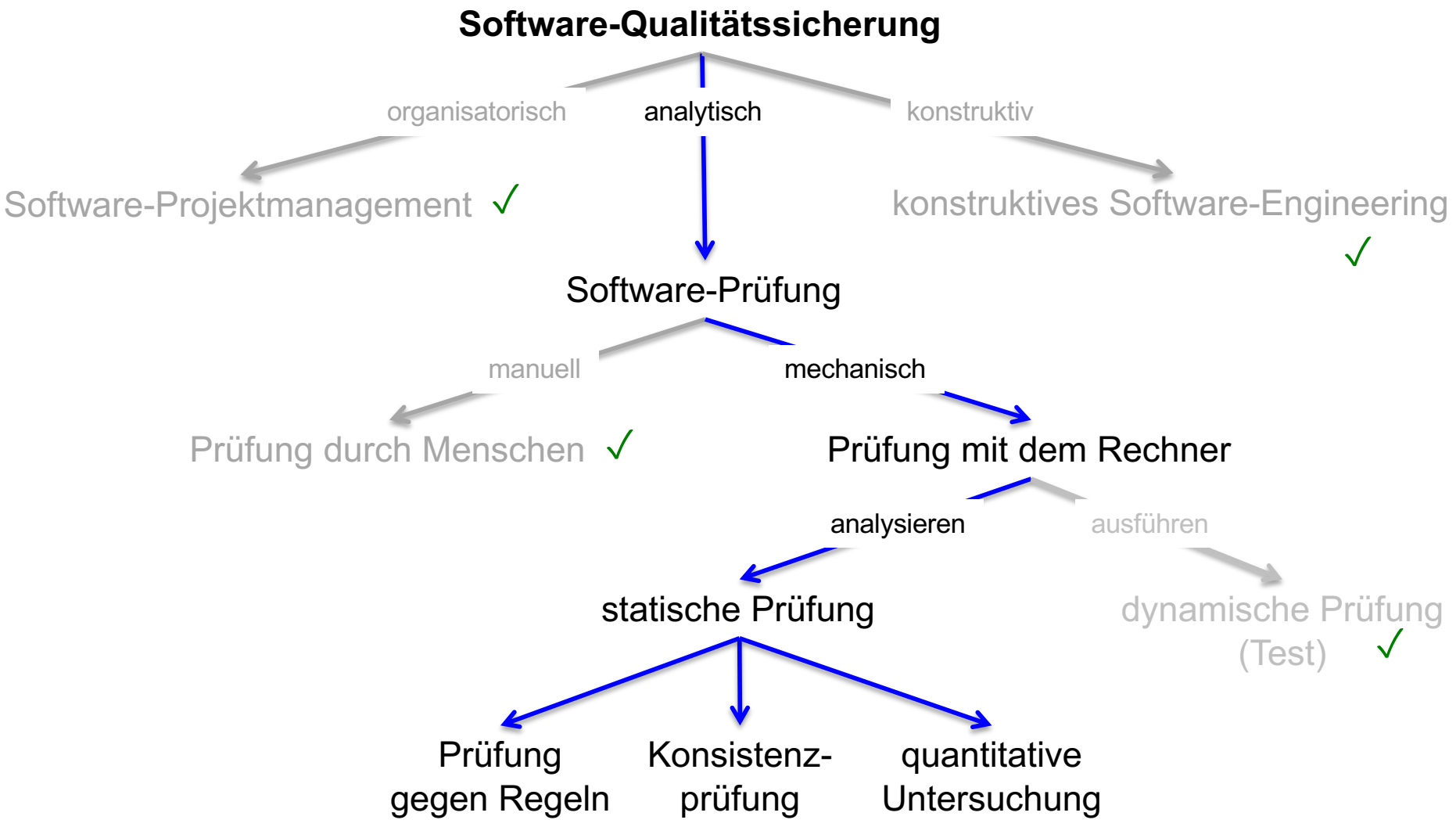
```
@Test
public void testIstNull(){
   final Class<?> clazz = cal.getClass();
   try {
      Class<?>[] param = {int.class};
      Method privat = clazz.getDeclaredMethod("istNull", param);
      privat.setAccessible(true);
      assertTrue(((Boolean)privat.invoke(cal, 0)).booleanValue());
   } catch (Exception e) {
      e.printStackTrace();
   }
}
```

Achtung: Methodenname und Parametertypen müssen bei Änderung angepasst werden

Aufruf der privaten Methode

Statische Prüfung

Metriken



Gliederung der Software-Qualitätssicherung nach [LL10]

Statische Prüfung

- Durch eine statische Prüfung sollen vorhandene Fehler oder fehlerträchtige Stellen in einem Dokument gefunden werden
- Im Gegensatz zum Review wird die statische Prüfung durch Werkzeuge vorgenommen
- Die Prüfung durch Werkzeuge setzt voraus, dass das Dokument eine formale Struktur besitzt
- Beispiele:
  - Anforderungsdokument (Prosa): Rechtschreib- und Grammatikprüfung
  - UML-Diagramme: Prüfung auf Einhaltung der UML-Syntax
  - Programmcode : Syntaxanalyse, Konformitäts-/Konventionsanalyse, Prüfung auf Kontrollfluss- und Datenflussanomalien

Wir konzentrieren uns später auf die Analyse des Programmcodes

Die Bezeichnung „statische Prüfung" weist darauf hin, dass der Programmcode für die Prüfung nicht ausgeführt werden muss

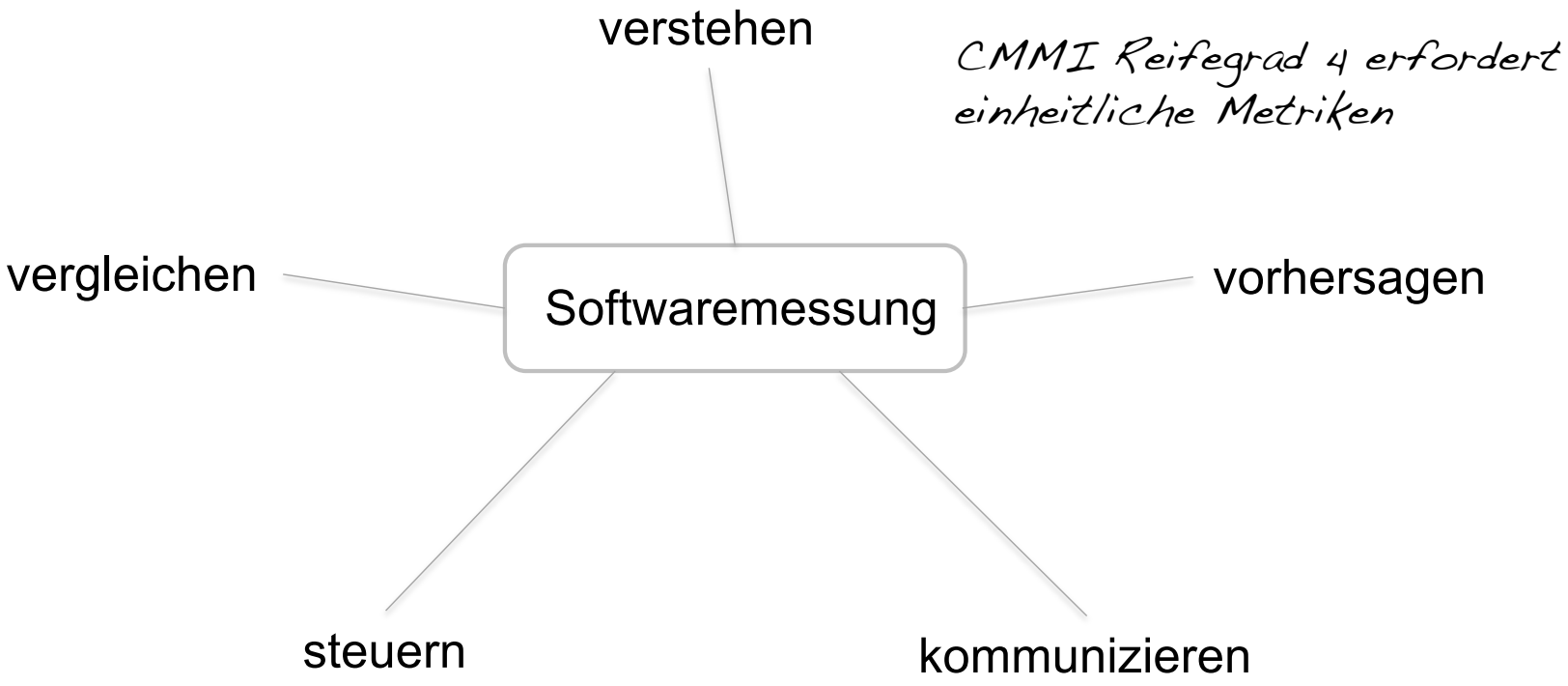
- Zeitpunkt einer statischen Prüfung (von Programmcode)
  - vor einer manuellen Prüfung (z.B. Review)
  - vor einem Komponententest
  - vor einem Integrationstest
- Die statischen Analysewerkzeuge werden von den Entwicklern/Testern eingesetzt
- Die statischen Analysewerkzeuge liefern häufig auch Messwerte, die Hinweise auf die Quantität und die Qualität geben können
- Wir betrachten daher zunächst Softwaremetriken

Metriken

- Vermessung und Prüfung des Quellcodes mit dem Ziel, die Qualität und Quantität zu bestimmen
- Für Software stehen leider keine aussagekräftigen physikalischen Maßeinheiten, wie z.B. Länge, Gewicht, Spannung, Widerstand, zur Verfügung
- Es stehen aber eine Vielzahl von Software-Metriken zur Verfügung
- Trotz ihres heuristischen Ansatzes können sinnvoll gewählte Software-Metriken hilfreiche Aussagen liefern

> *Quantitative Bewertung von Software-Systemen mit Hilfe von Kenngrößen*

- Neben dem Produkt kann auch der Prozess vermessen werden (z.B. Aufwand, Kosten, Dauer, ...)
- Was ist Sinn und Zweck der Softwaremessung? [SSB10]



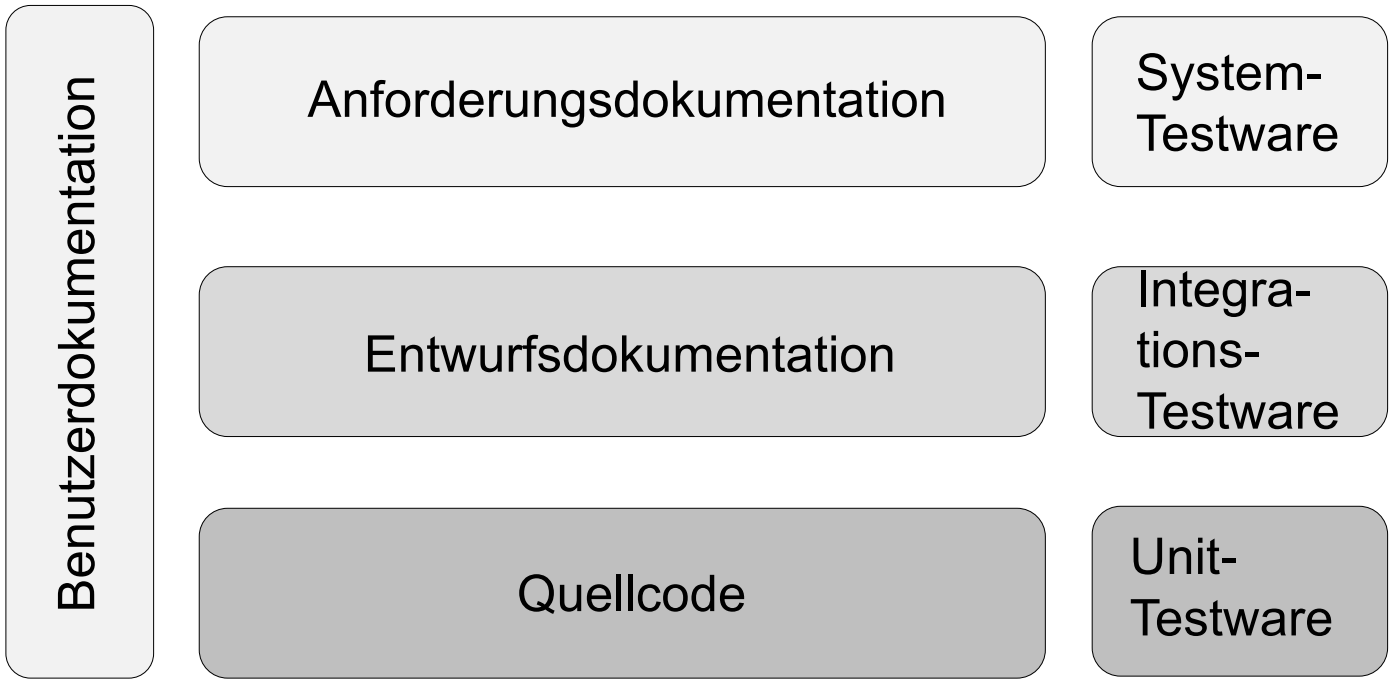
„You cannot control what you cannot measure" [DeMarco]

- Unterteilung von Software-Metriken
  - Quantitätsmetriken
  - Komplexitätsmetriken
  - Qualitätsmetriken
- Wir wissen bereits: Software ist mehr als nur das Programm
  - Anforderungsdokumentation/Spezifikation
  - Entwurfsdokumentation
  - Code
  - Testware
  - Benutzerhandbücher
- Daraus ergeben sich die unterschiedlichen Objekte der Softwaremessung
- Man kann unterschiedliche Sichten auf die Objekte einnehmen

- Sicht der Typen
  - Natürlichsprachliche Texte
  - Diagramme
  - Tabellen
  - Code
- Sicht des Zwecks
  - Anforderungsartefakt
  - Entwurfsartefakt
  - Codeartefakt
  - Testartefakt
  - Beschreibungsartefakt
- Sicht des Anwenders
  - System/Benutzerinteraktion / Systemausgabe
  - Benutzerdokumentation

Quantitätsmetriken

– Quellen



- Entwurfsgrößen
  - Strukturierte Entwurfsgrößen
    - Module
    - Funktionen
    - Datenobjekte
    - Schnittstellen
  - Objektmodellgrößen
    - Klassen / Klasseninteraktionen
    - Methoden
    - Attribute
  - Datenmodellgrößen
    - Datenentitäten
    - Datenattribute
    - Datenschlüssel
    - Datenbeziehungen
    - Datensichten

- Codegrößen:
  - Codedateien
  - Codezeilen
  - Anweisungen
  - Prozeduren / Funktionen / Methoden
  - Module / Klassen
  - Entscheidungen
  - Logikzweige
  - Aufrufe
  - Definierte Datenelemente
  - Benutzte Datenelemente
  - Benutzte Operanden
  - Datenobjekte
  - Datenzugriffe
  - Interaktionselemente
  - Kommentare