

Software- Qualitätsmanagement

**Vorlesung im Modul 10-202-2319
Software-Management**

Sommersemester 2015

Prof. Dr. Hans-Gert Gräbe

<http://bis.informatik.uni-leipzig.de/HansGertGraebe>

Aufbau eines Testwerkzeugs am Beispiel von JUnit

Üblicher Ansatz für Tests und Fehlersuche:

- Print-Befehle, Debugger-Ausdrücke, Test-Skripte
- möglichst über globale Variable *debug* steuerbar

Umsetzung in einem OO-Ansatz

Command Pattern

Idee: Objekte mit gemeinsamer *run*-Methode, in welcher die Test-Aktionen gekapselt sind.

```
public class TestCase implements Test {  
    private final String fName; // identifiziert Test  
    public void run(); // zu überladende Methode  
}
```

Wie hängt der Programmierer seinen Testcode ein?

Tests haben eine gemeinsame Struktur:

Aufsetzen der Testumgebung -> Code gegen diese Umgebung laufen lassen -> Ergebnisse mit den Erwartungen vergleichen -> Testumgebung auflösen

Template Method Pattern

Idee: Skelett eines Algorithmus vorgeben, die Methoden werden in Subklasse konkretisiert.

```
public void run() {  
    setUp(); /* jeweils protected und leere Methodenrumpfe */  
    runTest();  
    tearDown();  
}
```

Wie werden die Testergebnisse eingesammelt?

Ausgabe ist unsymmetrisch: Von erfolgreichen Tests ist nur Statistik interessant, sonst genauere Informationen über die Fehlerstelle.

Collecting Result Pattern

Idee: Der Methode ein Objekt übergeben, welches die Ergebnisse einsammelt.

```
public class TestResult extends Object {
    protected int fRunTests; /* Zähler der Testläufe */
    ... }
    public void run(TestResult result) {
        result.startTest(this);
        setUp(); runTest(); tearDown();
    }
    public synchronized void starttest(Test test) { fRunTest++; }
    /* synchronized, da verschiedene Tests auf dasselbe Resultat
       schreiben könnten */
```

Der Test hat einen Fehler entdeckt. Was weiter?

Fehler können planmäßig und unerwartet auftreten. JUnit unterscheidet deshalb *failure* und *error*. Realisierung durch Ausnahmebehandlung mit spezieller Ausnahmeklasse *AssertionFailedError* für failures.

```
public void run(TestResult result) {  
    result.startTest(this);  
    setUp();  
    try { runTest(); }  
    catch (AssertionFailedError e) // planmäßige Ausnahmen  
        { result.addFailure(this, e); }  
    catch (Throwable e) // unplanmäßige Ausnahmen  
        { result.addError(this, e); }  
    /* An der Stelle sind alle Ausnahmen abgefangen! Keine Ausnahme  
       wird aus TestCase.run herausgereicht! */  
    finally { tearDown(); }  
}
```

Wo kommen planmäßige *AssertionFailedErrors* her?

Werden von *assert*-Methoden aus der Klasse *TestCase* ausgelöst (es gibt noch mehr *assertXXX*-Methoden)

```
protected void assert (boolean condition) {  
    if (!condition) throw new AssertionFailedError();  
}
```

Aufsammeln in entsprechenden Aggregationen in *TestResult*

```
public synchronized void addError(Test test, Throwable t) {  
    fErrors.addElement(new TestFailure(test,t));  
}  
public synchronized void addFailure(Test test, Throwable t) {  
    fFailures.addElement(new TestFailure(test,t));  
}  
public class TestFailure extends Object { // Wrapper-Klasse  
    protected Test fFailedTest; protected Throwable fThrownException;  
}
```

JUnit kommt mit verschiedenen fertigen Subklassen von *TestResult*, z.B. *TextTestResult* für textuelle Darstellung oder *UITestResult* für Einbindung in grafische Testumgebung. Erweiterungen sind möglich, z.B. *HTMLTestResult*.

TestCase: viele, aber nur wenig variierende Klassen

Lösung in JUnit: Verwendung innerer Klassen erspart das Erfinden von Klassennamen (Adapter Pattern)

```
TestCase test = // Wiederverwendung der generischen Klasse MoneyTest
    new MoneyTest(„testMoneyEquals“) {
        // neue innere Klasse als Subklasse
        protected void runTest() { testMoneyEquals(); }
        // Methode runTest überschrieben
    }
```

Anderer möglicher Lösungsansatz: Parametrisierte Klassen, wird erst von Java 5 unterstützt. Siehe Junit 4

Kann auch über Reflection und Stringmanipulation simuliert werden.

Composite Pattern - Ausführung mehrerer Tests „im Stück“

Idee: Anordnung der Objekte in einer Baumstruktur, um Teil-Ganzes-Hierarchien auszudrücken. Einzelne Objekte und Objekt-Aggregationen werden auf dieselbe Weise behandelt.

Bestandteile:

- Komponente: Schnittstellendefinition, mit welcher unsere Tests interagieren sollen. (*interface Test*)
- Komposition: Implementierung dieser Schnittstelle samt Management von Test-Sammlungen. (*class TestSuite implements Test*)
- Blatt: Repräsentation eines TestCase in einer solchen Komposition, welcher die Komponenten-Schnittstelle implementiert.

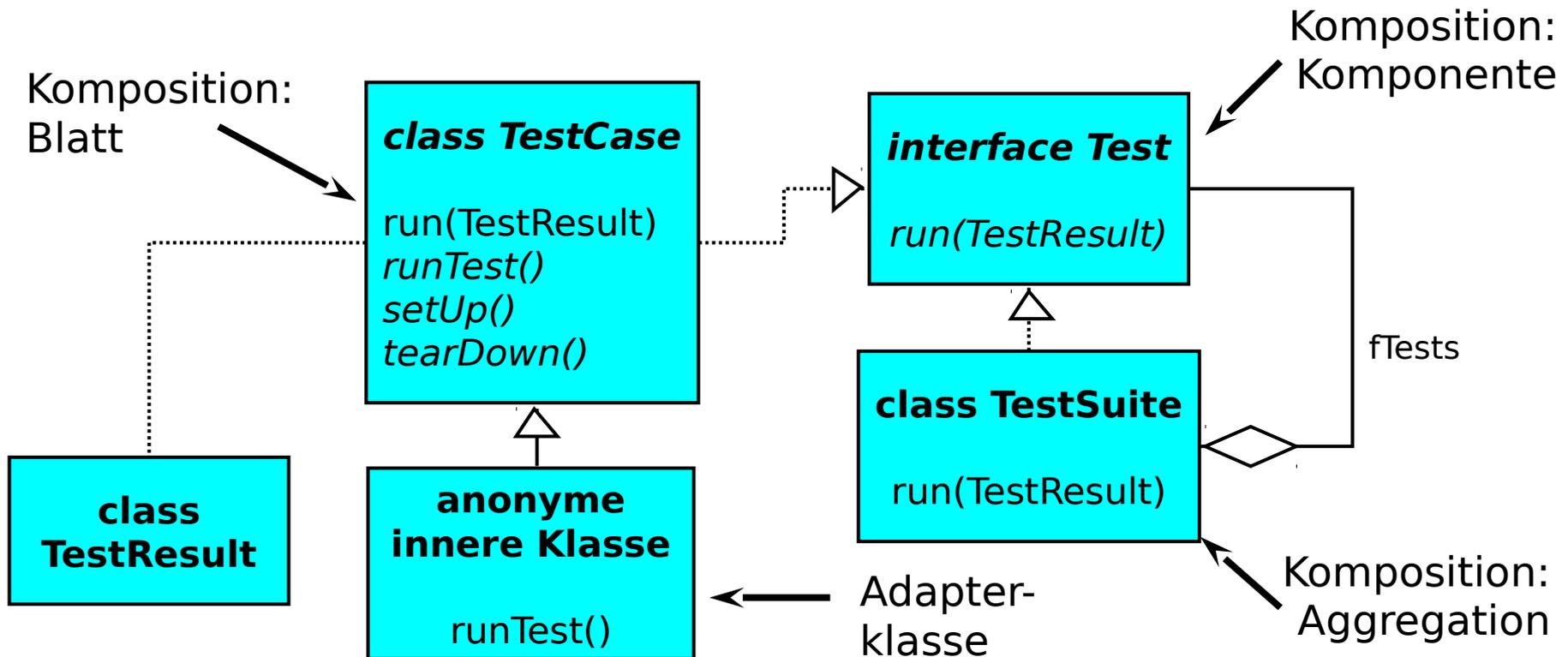
```
public interface Test {
    public void run (TestResult result);
}
public class TestSuite implements Test {
    private Vector fTests = new Vector();
    public void run (TestResult result) {
        // Delegiert Testausführung auf die Kinder
        for (Enumeration e=fTests.Elements(); e.hasMoreElements;) {
            Test test = (Test) e.nextElement();
            test.run(result);
        }
    }
    public void addTest(Test test) { // Clients können neue Tests hinzufügen
        fTests.addElement(test);
    }
}
```

5. Testende Verfahren

6. Aufbau eines Testwerkzeugs

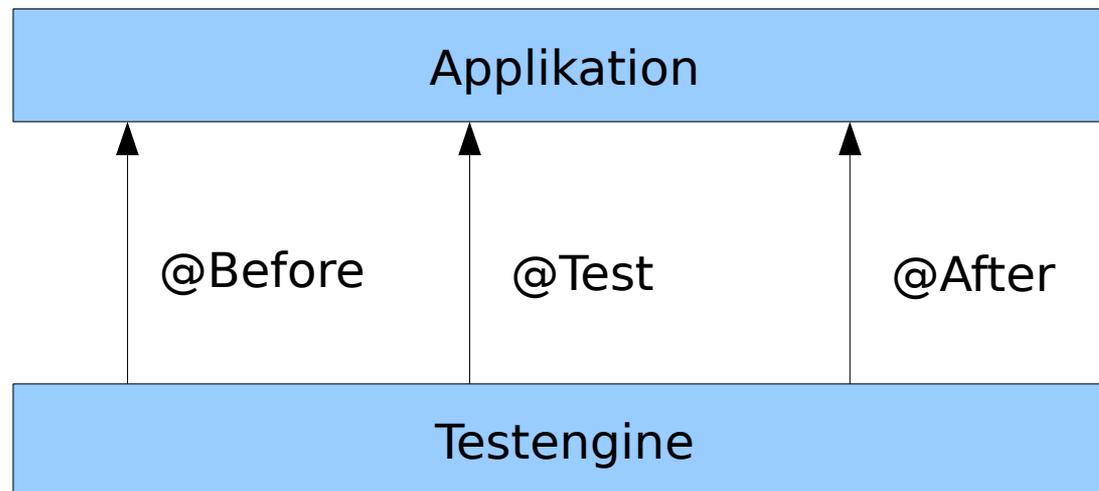
Nachteil dieser Methode: Alle Tests müssen per Hand in eine entsprechende TestSuite eingetragen werden (statischer JUnit-Zugang)

Alternative Lösung: Java sucht mit Reflection-Methoden nach Methoden mit entsprechendem Namen und fügt diese selbst zu einer TestSuite zusammen (dynamischer JUnit-Zugang).



JUnit 4

- Neuer Zugang über Annotationen und Dependency Injection
- Annotationen als neues Ausdrucksmittel der Metaprogrammierung.
- Zugriff auf markierte Codeelemente über Reflection-Mechanismus
- JUnit 4 verwendet dieses Sprachkonstrukt, um beliebige Methoden beliebiger Klassen als ausführbaren Testfall kennzeichnen zu können.



JUnit 4

- Kein Ableiten von TestCase mehr.
- Import static = Import der statischen Methoden einer Klasse in eine andere Klasse
- Testmethoden als @Test public void *beliebigerName()* ohne Parameter.

```
import junit.framework.TestCase;  
import org.junit.Test;  
import static org.junit.Assert.*;  
  
public class EuroTest extends TestCase {  
  @Test public void testadding() {  
    Euro two = new Euro(2.00);  
    Euro sum = two.add(two);  
    assertEquals("sum", new Euro(4.00), sum);  
    assertEquals("two", new Euro(2.00), two);  
  }  
}
```

JUnit 4

@Before und **@After**: Setup- und Teardown-Methoden werden wie Testfälle via Annotation gekennzeichnet

- @Before Methoden werden vor jedem Testfall ausgeführt
- @After Methoden nach jedem Testfall
- @Before Methoden der Oberklasse vor denen der Unterklasse
- @After Methoden der Unterklasse vor denen der Oberklasse

@BeforeClass und **@AfterClass**: Für kostspielige Test-Setups, die nicht für jeden einzelnen Testfall neu auf- und abgebaut werden können,

- @BeforeClass läuft für jede Testklasse nur ein einziges Mal und noch vor allen @Before Methoden,
- @AfterClass entsprechend für jede Testklasse nur einmal und zwar nach allen @After Methoden.
- Die so markierten Methoden müssen statisch sein

Junit 4

Erwartete Exceptions: Zum Testen von Exceptions kann der `@Test` Annotation mitgeteilt werden, dass die Ausführung Ihres Testfalls zu einer gezielten Exception führen soll:

```
public class EuroTest...  
@Test(expected = IllegalArgumentException.class)  
public void negativeAmount() {  
    final double NEGATIVE_AMOUNT = -2.00;  
    new Euro(NEGATIVE_AMOUNT); // should throw the exception  
}  
}
```

Ähnlich mit anderen Parametern, etwa `timeout`

Testmethodik

Jede der Testmethoden hat ihre Vor- und Nachteile.

- **Strukturtests** sind nicht in der Lage, fehlende oder fehlerhafte Funktionalität (Semantik) zu erkennen.
- **Funktionstests** sind nicht in der Lage, Implementationsdetails zu prüfen (Syntax).
 - Spezifikation besitzt ein höheres Abstraktionsniveau als die Implementierung.

Üblich sind deshalb Kombinationen der Verfahren zu einer stufenweise aufgebauten **Testmethodik**.

Anforderungen an eine stufenweise aufgebaute **Testmethodik**

- Erfüllung anerkannter Minimalkriterien (Zweigüberdeckung, Funktionstest)
- Fehlersensitive Testdaten
- Testdaten für funktionelle Korrektheitsprüfung
- Systematik, Wirtschaftlichkeit, Nachvollziehbarkeit
- Verwendung geeigneter Werkzeuge
 - Protokoll, Testmetriken erstellen, Regressionstests

Problem der Verteilung der Verantwortung zwischen Programmierer und Tester.

- Test durch Programmierer: Eigenes Code-Review
 - Prinzip der frühen Fehlerentdeckung
 - Prinzip der entwicklungsbegleitenden QS
- Test durch Testabteilung: Tests unter anderen als den Entwicklungsbedingungen möglich

Verschiedene Sichten auf den Testprozess

Programmierer: strukturelle Aspekte stehen im Vordergrund

- **Frage:** Durchlaufen die Testdaten die richtigen Wege
- **Aspekte:** Test von Programmteilen, Tests zu Optimierungszwecken

Endprüfung: funktionale Aspekte stehen im Vordergrund

- **Frage:** Ist der vereinbarte Leistungsumfang vollständig und korrekt realisiert?
- **Aspekte:** alle Teilfunktionen, Abdecken des gesamten Eingabebereichs, Spezialfälle korrekt behandelt

Management: organisatorische Aspekte stehen im Vordergrund

- **Aspekte:** Abschätzung des Testaufwands, Beurteilung des Projektfortschritts, Beurteilung der Testgüte

Tests für Moduln mit hoher Kontrollflusskomplexität

Voraussetzungen: Testwerkzeug, Spezifikation, Quellcode

1. Funktionstest

- Testling wird instrumentiert
 - Ziel: Überdeckung im Hintergrund protokollieren
- Klassenbildung an Hand der Spezifikation, Auswahl von Grenzfällen, Spezialfällen und anderen Testfällen
 - Ohne Hinzuziehen des Quellcodes!
- Testfälle mit instrumentiertem Testling ausführen
 - Vergleich der Ist- mit den Soll-Werten

Ergebnis: Funktions- und Leistungsumfang wurde systematisch geprüft

2. Strukturtest

- Die in Phase 1 erstellte Überdeckungsstatistik wird ausgewertet.
 - Ursachen für nicht überdeckte Anweisungen, Pfade, Bedingungen sind zu ermitteln
- Testfälle für die noch nicht durchlaufenen Zweige etc. erstellen bzw. diese entfernen, bis die geplante Überdeckungsrate erreicht ist.
- Problem gegen Ende:
 - Testsituation schwer herstellbar (etwa „Platte voll“)
 - Testfälle schwer herleitbar (relevanter Eingabebereich schwer zu bestimmen)
 - Für beide Fälle ist eine spezialisierte Testabteilung oft besser gerüstet

Ergebnis: Struktur wurde systematisch geprüft und optimiert

3. Regressionstest

- Nochmaliger automatischer Durchlauf aller Testfälle mit Soll/Ist-Vergleich nach jeder Fehlerkorrektur.

Ergebnis: Konsistenz der Fehlerkorrekturen mit den bisherigen Testergebnissen wird sichergestellt.

Hierfür ist der Einsatz eines entsprechenden Werkzeugs unerlässlich.

Erfahrungen zum Einsatz testender Verfahren

Ohne Testwerkzeuge sowie konstruktive Voraussicht ist eine systematische, effektive und ökonomische Überprüfung der Implementierung nicht möglich.

Funktionstest muss bereits im Entwurf berücksichtigt werden.

- Tests sind bereits in der Entwurfsphase mit zu planen und in der Implementierungsphase parallel zum funktionalen Code zu implementieren.
 - Regel: a little test, a little code, ...
 - Im Moment des Codierens sind die Vorstellungen über das Funktionieren des Codes am detailliertesten. Die rechte Zeit, auch über Tests nachzudenken ...
 - Prinzip der frühzeitigen Fehlerentdeckung!
- gute Regel ist: Tests zuerst schreiben.
 - M. Fowler: „Wenn du eine Print-Anweisung zum Debuggen schreiben willst, dann denke gleich über einen Testfall nach.“

Funktionstests bedürfen einer sorgfältigen Planung.

- Tests sollten auch modelliert werden; Identifizieren gemeinsamer Testinitialisierungen (*setUp()*)
 - mehrfachen ähnlichen Testcode refaktorisieren!
- Alte Tests lauffähig zu halten ist fast so wichtig wie neue Tests zum Laufen zu bringen.
- Tests sollten mehrmals an Tag laufen
 - z.B. in der Frühstückspause

Auf zusätzlichen Strukturtest sollte nicht verzichtet werden.

- Sinnvoll, wenn entscheidende Teile des Codes fertig sind, und auf der Basis der Ergebnisse der während der Funktionstests erstellten Metriken

Integrationstests erst, wenn alle Systemkomponenten die Einzeltests bestanden haben.

Benutzerakzeptanz erfordert für die Tests einen hohen Automatisierungsgrad und komfortable Werkzeuge. (wie JUnit)

- Verwendung integrativer konzeptioneller Testansätze
- Nutzung vorhandener und bereits erstellter Testbibliotheken
 - Nach einiger Zeit ist Einfügen eines neuen Tests so einfach wie das Hinzufügen einer neuen Methode zu einer Klasse.
- Mit Tests im Rücken ändert sich der Zugang zum Programmieren, da laufende Tests ein Gefühl von Sicherheit vermitteln.

Die Testwerkzeuge müssen in die gesamte Entwicklungssystematik eingebettet sein und das Vorgehen durch entsprechende Testrichtlinien inhaltlich und methodisch untersetzt sein.

1. Testen versus Verifizieren
2. Konditionieren von Programmen
 - Zusicherungen
 - Spezifizieren mit Anfangs- und Endebedingung
3. Programmverifikation
 - Verifikationsregeln
 - Termination von Schleifen
 - Entwickeln von Schleifen
4. Symbolisches Testen

Testen = Stichprobenartige Überprüfung des Programms

- Auswahl einer möglichst repräsentativen Menge von Eingabedaten
- Test liefert eine gewisse Plausibilität des korrekten Funktionierens, aber keine Sicherheit
- Mit Tests lassen sich nicht nur die Korrektheit, sondern auch andere Parameter (Profiling) erfassen.

Tests haben Überzeugungskraft, aber keine Beweiskraft im streng deduktiven Verständnis.
Es bleibt immer ein Rest von Unsicherheit.

Verifikation = Formal exakte Methode, um durch theoretische Analyse die Konsistenz zwischen Spezifikation und Implementierung für *alle* möglichen Eingabedaten zu **beweisen**.

- Beweis im mathematisch deduktiven Verständnis

Grundstruktur mathematischer Beweise

- modularer Aufbau aus Aussage-Bausteinen mit „Wenn..., dann...“ Struktur
- Verifikation eines neuen Bausteins durch **Beweis** = Zusammensetzen einer Argumentation aus bereits verifizierten Bausteinen nach den Regeln der (math.) Logik

Programm = schrittweise Transformation der Eingabe- in die Ausgabedaten nach einem vorgegebenen Algorithmus

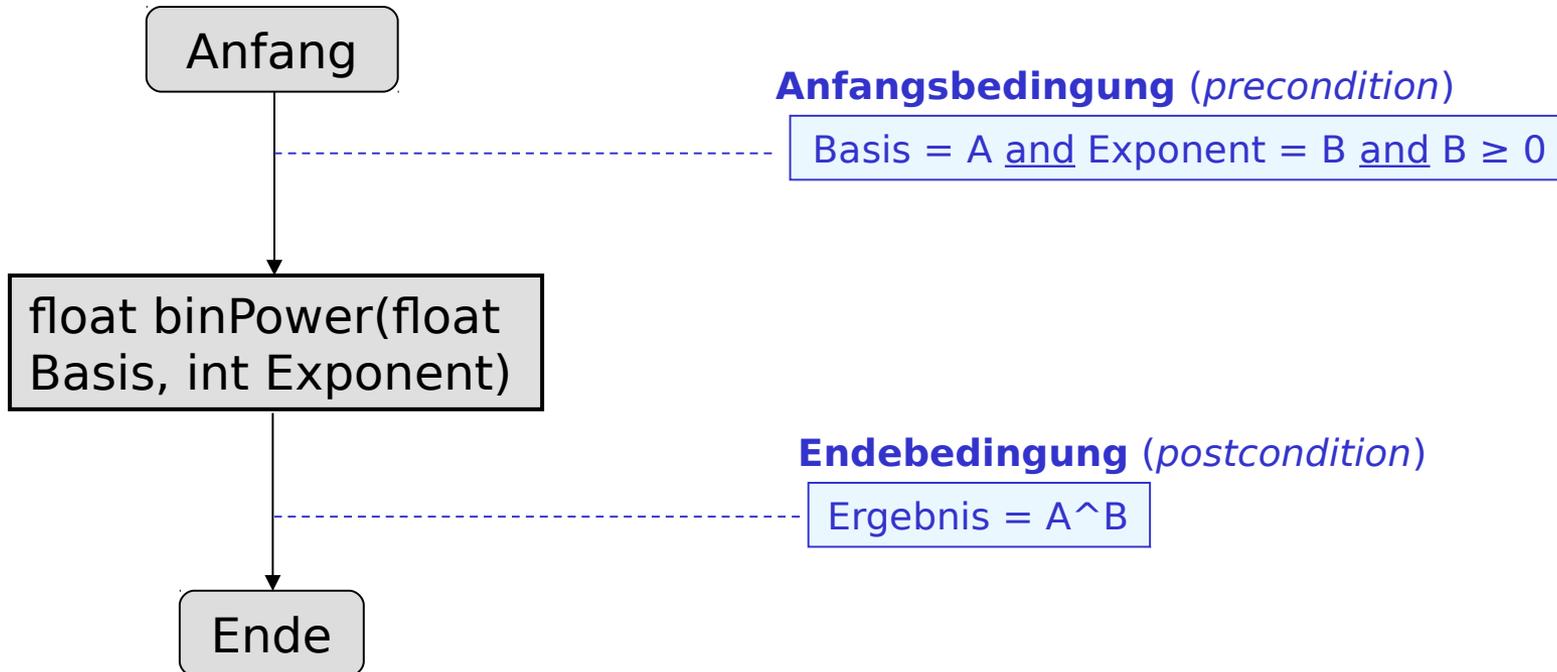
- modularer Aufbau, der für Zwecke der Verifikation entsprechend **konditioniert** werden muss.

Zusicherungen

- Konditionierung bedeutet, das Programm in Einheiten von Wenn-Dann-Aussagen zu zerlegen.
- **Zusicherungen** beschreiben dazu bestimmte **Eigenschaften** der Datenlandschaft an vorgegebenen Kontrollpunkten im Programmfluss.
- Logische Aussagen über Werte von Variablen im Programm
 - Formulierung auf unterschiedliche Art und Weise:
 - umgangssprachlich, z. B. x ist nicht negativ
 - formal, z. B. $x \geq 0$
 - gebräuchliche Notationen :
 - Annotation durch gestrichelte Linien am Programmablaufplan
 - Kommentare oder Makros in Programmiersprachen, z. B. `assert (x >= 0); //Zusicherung ist ungültig, wenn x negativ ist.`
 - Ergänzung von Struktogrammen durch abgerundete Rechtecke

6. Verifizierende Verfahren

2. Konditionierung von Programmen



Beispiel: Binäres Potenzieren, Spezifikation