

Software Qualitätssicherung

HTWG Konstanz

Komponententests (Modultests) und
Testabdeckung



Christian Baranowski

Einschub

Was sind Java Annotation?

Als Annotation wird im Zusammenhang mit der Programmiersprache Java ein Sprachelement bezeichnet, das die Einbindung von Metadaten in den Quelltext erlaubt. - Wikipedia

Wiederholung

Grundlagen Software Qualitätssicherung

Was ist Software Qualität ?



Was soll sichergestellt werden?

Was soll sichergestellt werden?

Fehlerwirkung (Failure)

Nach außen sichtbare Fehlverhalten



Was soll sichergestellt werden?

Fehlerwirkung (Failure)

Nach außen sichtbare Fehlverhalten



Fehlerzustand (Bug)

Zustand in der Anwendung der zu einer Fehlerwirkung führen kann.



Was soll sichergestellt werden?

Fehlerwirkung (Failure)

Nach außen sichtbare Fehlverhalten



Fehlerzustand (Bug)

Zustand in der Anwendung der zu einer Fehlerwirkung führen kann.



Fehlhandlung (Error, Misstake)

Irrtum bei der Software-Entwicklung.

Was soll sichergestellt werden?

Fehlerwirkung (Failure)

Nach außen sichtbare Fehlverhalten



Fehlerzustand (Bug)

Zustand in der Anwendung der zu einer Fehlerwirkung führen kann.



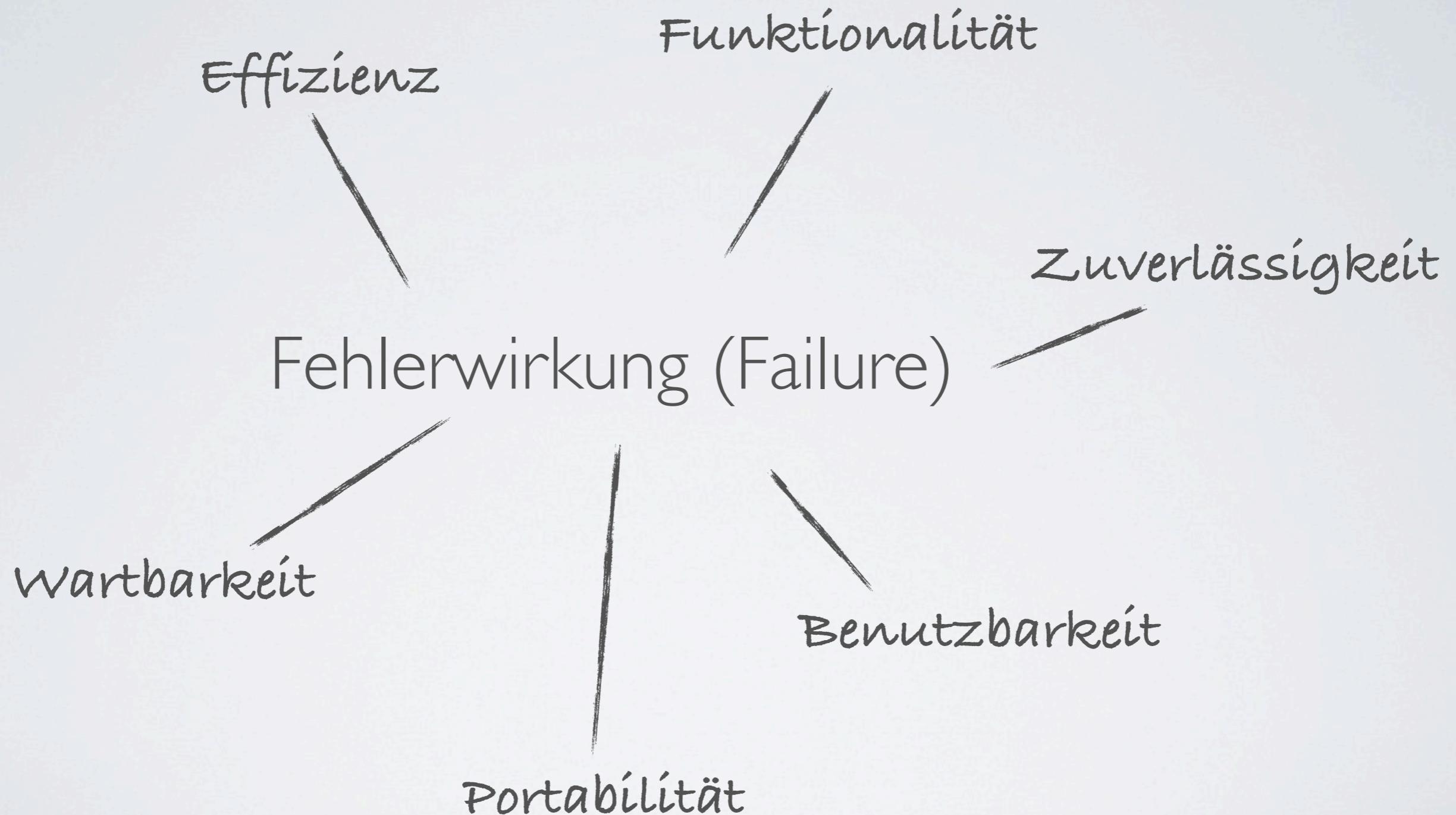
Fehlhandlung (Error, Misstake)

Irrtum bei der Software-Entwicklung.

Fehlermaskierung

Wirkung eines Fehlers (Fehlerzustand) ist nach außen nicht sichtbar, weil er durch einen weiteren Fehler verborgen (überlagert / maskiert) wird.

Was soll sichergestellt werden? Fehlerwirkungen für alle Attribute der ISO 9126



Wege zu guter Software

Wege zu guter Software

analytische Qualitätssicherung

Methoden die dazu dienen eine Anwendung strukturiert (analytisch) nach Fehlerwirkungen, Fehlerzuständen und Fehlermaskierung zu untersuchen. Die Qualität wird gemessen. Anzahl der Fehlerwirkungen, Fehlerzuständen und Fehlermaskierung bestimmt.

Wege zu guter Software

analytische Qualitätssicherung

Methoden die dazu dienen eine Anwendung strukturiert (analytisch) nach Fehlerwirkungen, Fehlerzuständen und Fehlermaskierung zu untersuchen. Die Qualität wird gemessen. Anzahl der Fehlerwirkungen, Fehlerzuständen und Fehlermaskierung bestimmt.

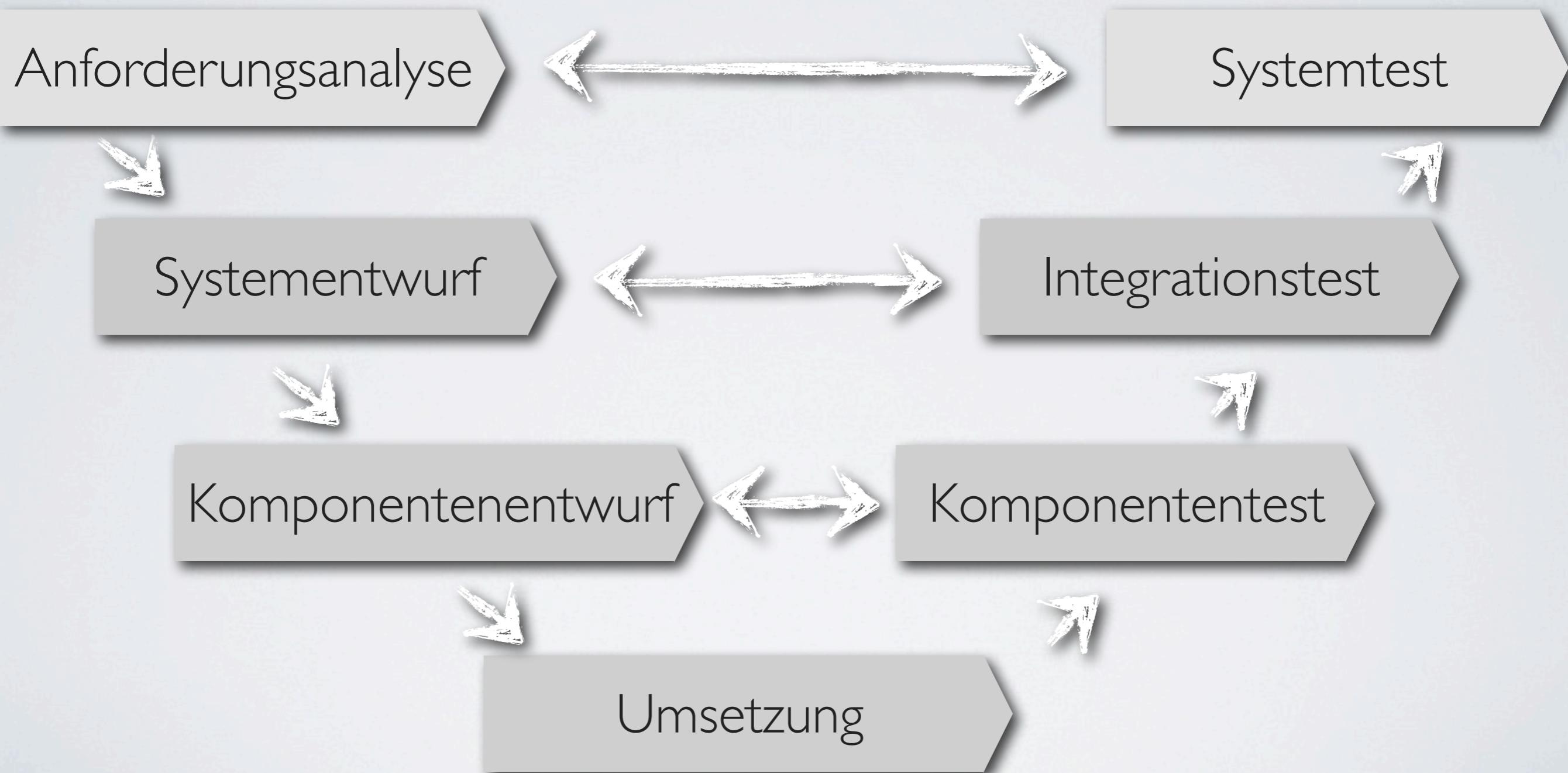
konstruktive Qualitätssicherung



Methoden die dazu dienen die Wahrscheinlichkeit für Fehlhandlungen zu minimieren. D.h. die Methoden führen dazu dass es zu einer geringeren Fehlerwahrscheinlichkeit im Projekt kommt.

Ende der Wiederholung
Grundlagen Software Qualitätssicherung

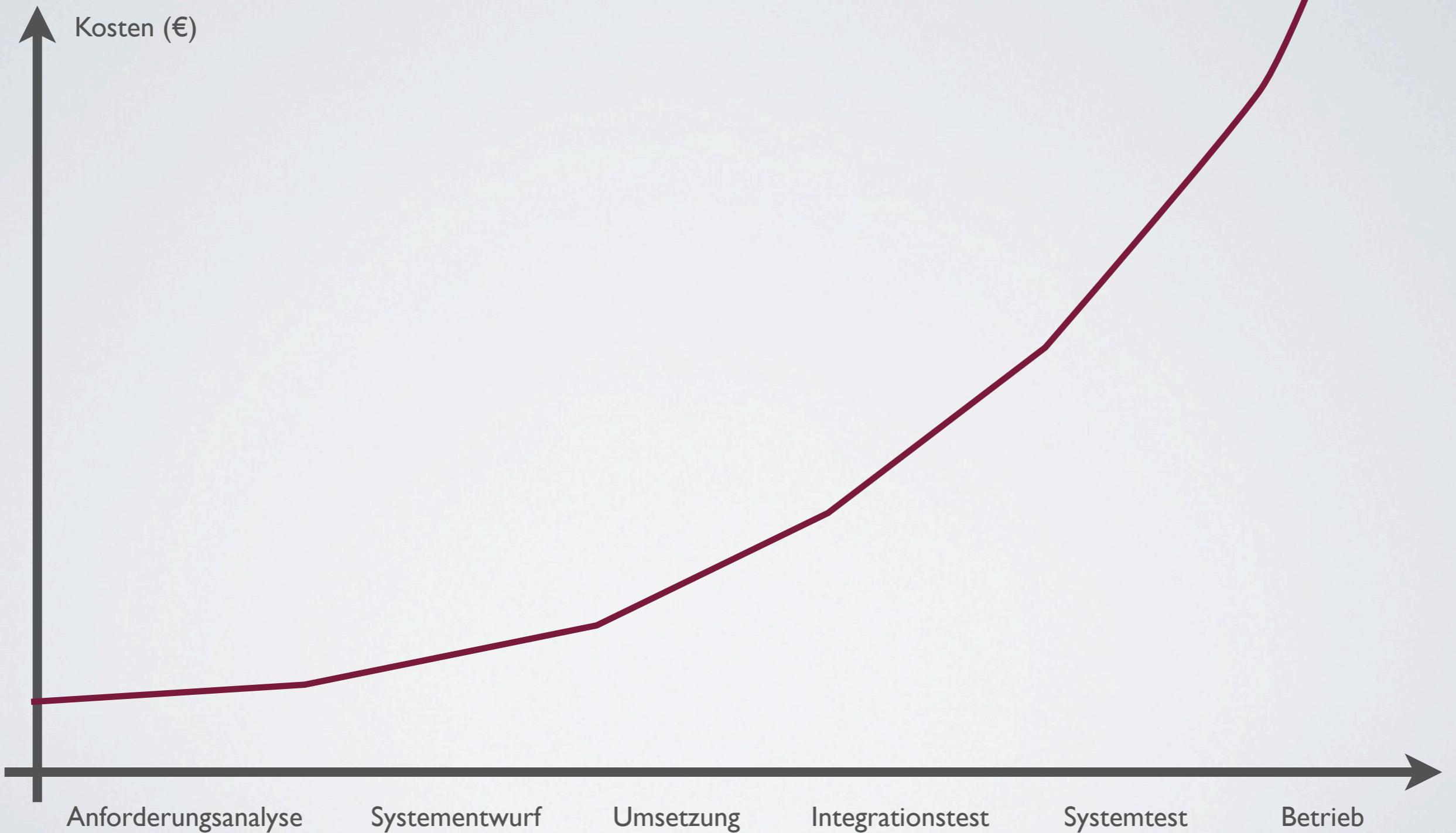
Testen im Software Lebenszyklus



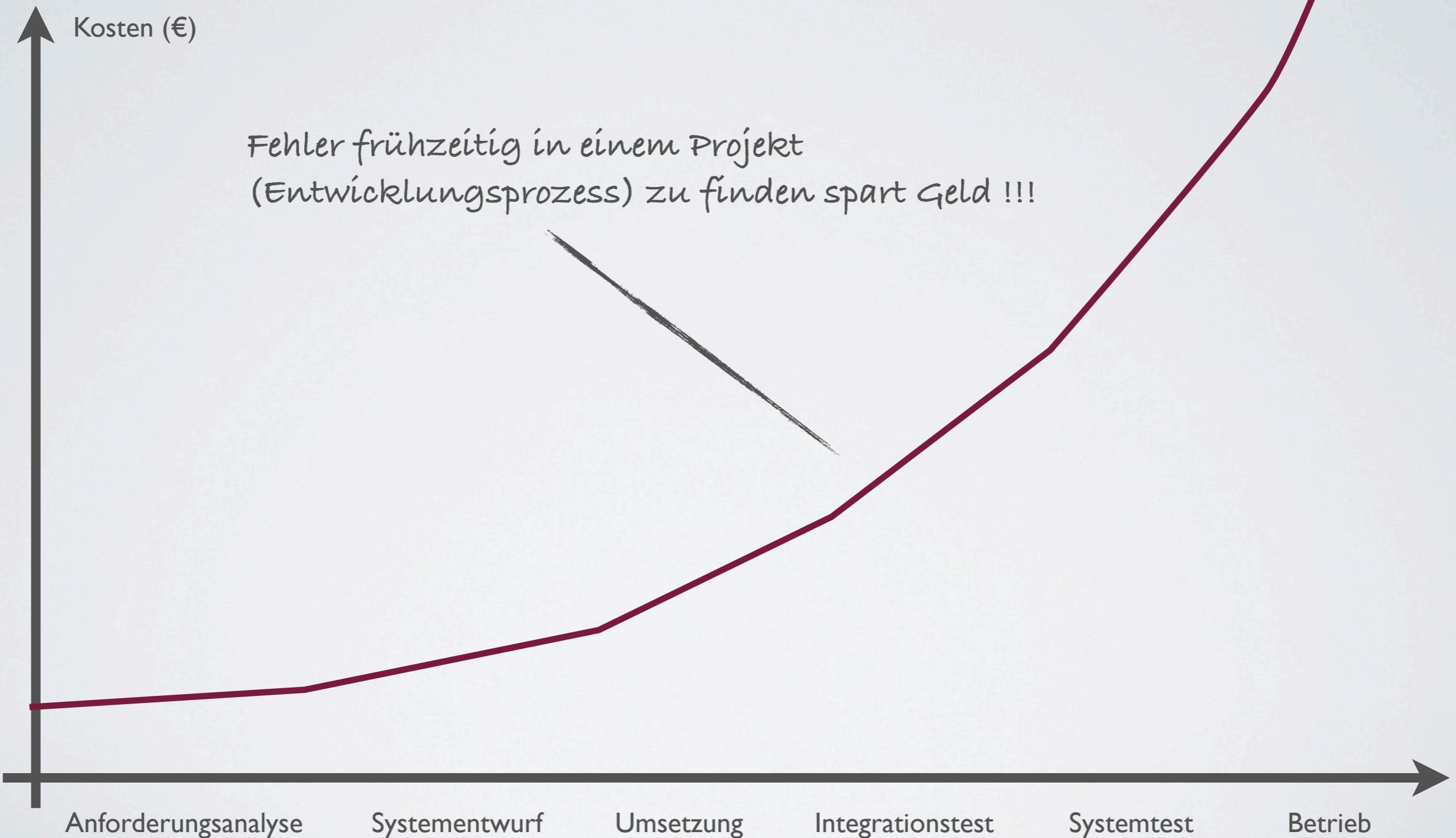
Testen im Software Lebenszyklus



Kosten zur Fehlerbehebung



Kosten zur Fehlerbehebung



Test Charakterisierung

- Was wird getestet ? (Testobjekt)
- Wer testet? (Tester)
- Welche Qualitätseigenschaft (ISO9126) wird geprüft?
- Mit welchen Werkzeugen wird getestet?

Komponententests (Modultests) und Testabdeckung



Komponententest Charakter

- Was wird getestet ? (Testobjekt)
 - Klasse / Komponente (mehrere Klassen mit definierter Schnittstelle)
- Wer testet? (Tester)
 - Entwickler (White-Box / Gray-Box) / Tester (Black-Box)
- Welche Qualitätseigenschaft (ISO9126) wird geprüft?
 - Funktionalität, Robustheit, Effizienz
- Mit welchen Werkzeugen wird getestet?
 - xUnit Frameworks und Coverage Messung z.B. Cobertura, Emma

Wiederholung

Test Design Pattern vier Phasen Test

Wiederholung

Test Design Pattern vier Phasen Test

Setup

SUT in einen definierten Zustand bringen.

Wiederholung

Test Design Pattern vier Phasen Test

Setup

SUT in einen definierten Zustand bringen.



Exercise

SUT aufrufen mit Test Parametern (Daten).

Wiederholung

Test Design Pattern vier Phasen Test

Setup

SUT in einen definierten Zustand bringen.



Exercise

SUT aufrufen mit Test Parametern (Daten).



Verify

Prüfen ob das SUT im erwarteten Zustand ist.

Wiederholung

Test Design Pattern vier Phasen Test

Setup

SUT in einen definierten Zustand bringen.



Exercise

SUT aufrufen mit Test Parametern (Daten).



Verify

Prüfen ob das SUT im erwarteten Zustand ist.



Teardown

SUT und Testumgebung aufräumen.

Wiederholung JUnit

```
public class QuicksortTest {  
  
    Quicksort<Integer> quicksortSUT;  
  
    @Before  
    public void setUp() {  
        quicksortSUT = new Quicksort<Integer>(new IntComperator());  
    }  
  
    @Test  
    public void testSort() throws Exception {  
        quicksortSUT.sort(new Integer[] { 5, 8, 2, 4, 7 });  
        assertEquals(new Integer[] { 2, 4, 5, 7, 8 }, values);  
    }  
  
    @After  
    public void tearDown() {  
        quicksortSUT = null;  
    }  
}
```

Wiederholung JUnit

```
public class QuicksortTest {  
  
    Quicksort<Integer> quicksortSUT;          Phase 1: Setup  
  
    @Before  
    public void setUp() {  
        quicksortSUT = new Quicksort<Integer>(new IntComperator());  
    }  
  
    @Test  
    public void testSort() throws Exception {  
        quicksortSUT.sort(new Integer[] { 5, 8, 2, 4, 7 });  
        assertEquals(new Integer[] { 2, 4, 5, 7, 8 }, values);  
    }  
  
    @After  
    public void tearDown() {  
        quicksortSUT = null;  
    }  
}
```

Wiederholung JUnit

```
public class QuicksortTest {
```

```
    Quicksort<Integer> quicksortSUT;
```

Phase 1: Setup

```
@Before
```

```
public void setUp() {
```

```
    quicksortSUT = new Quicksort<Integer>(new IntComperator());
```

```
}
```

```
@Test
```

```
public void testSort() throws Exception {
```

```
    quicksortSUT.sort(new Integer[] { 5, 8, 2, 4, 7 });
```

```
    assertEquals(new Integer[] { 2, 4, 5, 7, 8 }, values);
```

```
}
```

Phase 2: Exercise

```
@After
```

```
public void tearDown() {
```

```
    quicksortSUT = null;
```

```
}
```

```
}
```

Wiederholung JUnit

```
public class QuicksortTest {  
  
    Quicksort<Integer> quicksortSUT;           Phase 1: Setup  
  
    @Before  
    public void setUp() {  
        quicksortSUT = new Quicksort<Integer>(new IntComperator());  
    }  
  
    @Test  
    public void testSort() throws Exception {           Phase 2: Exercise  
        quicksortSUT.sort(new Integer[] { 5, 8, 2, 4, 7 });  
        assertEquals(new Integer[] { 2, 4, 5, 7, 8 }, values);  
    }  
    @After  
    public void tearDown() {  
        quicksortSUT = null;  
    }  
}
```

Phase 3: Verify

Wiederholung JUnit

```
public class QuicksortTest {  
  
    Quicksort<Integer> quicksortSUT;           Phase 1: Setup  
  
    @Before  
    public void setUp() {  
        quicksortSUT = new Quicksort<Integer>(new IntComperator());  
    }  
  
    @Test  
    public void testSort() throws Exception {  
        quicksortSUT.sort(new Integer[] { 5, 8, 2, 4, 7 });  
        assertEquals(new Integer[] { 2, 4, 5, 7, 8 }, values);  
    }  
    @After  
    public void tearDown() {  
        quicksortSUT = null;           Phase 4: tearDown  
    }  
}
```

Phase 2: Exercise

Phase 3: Verify

Test Coverage
Testabdeckung

Testabdeckung

„Als Testabdeckung bezeichnet man das Verhältnis an tatsächlich getroffenen Aussagen eines Tests gegenüber den theoretisch möglich treffbaren Aussagen bzw. der Menge der gewünschten treffbaren Aussagen.“

- Wikipedia

Testabdeckung (100 %)

```
public boolean aAndbOrC(boolean A, boolean B, boolean C) {  
    return (A && B) || C;  
}
```

Erwartet	A	B	C
false	false	false	false
true	false	false	true
false	false	true	false
true	false	true	true
false	true	false	false
true	true	false	true
true	true	true	false
true	true	true	true

vollständige Testabdeckung

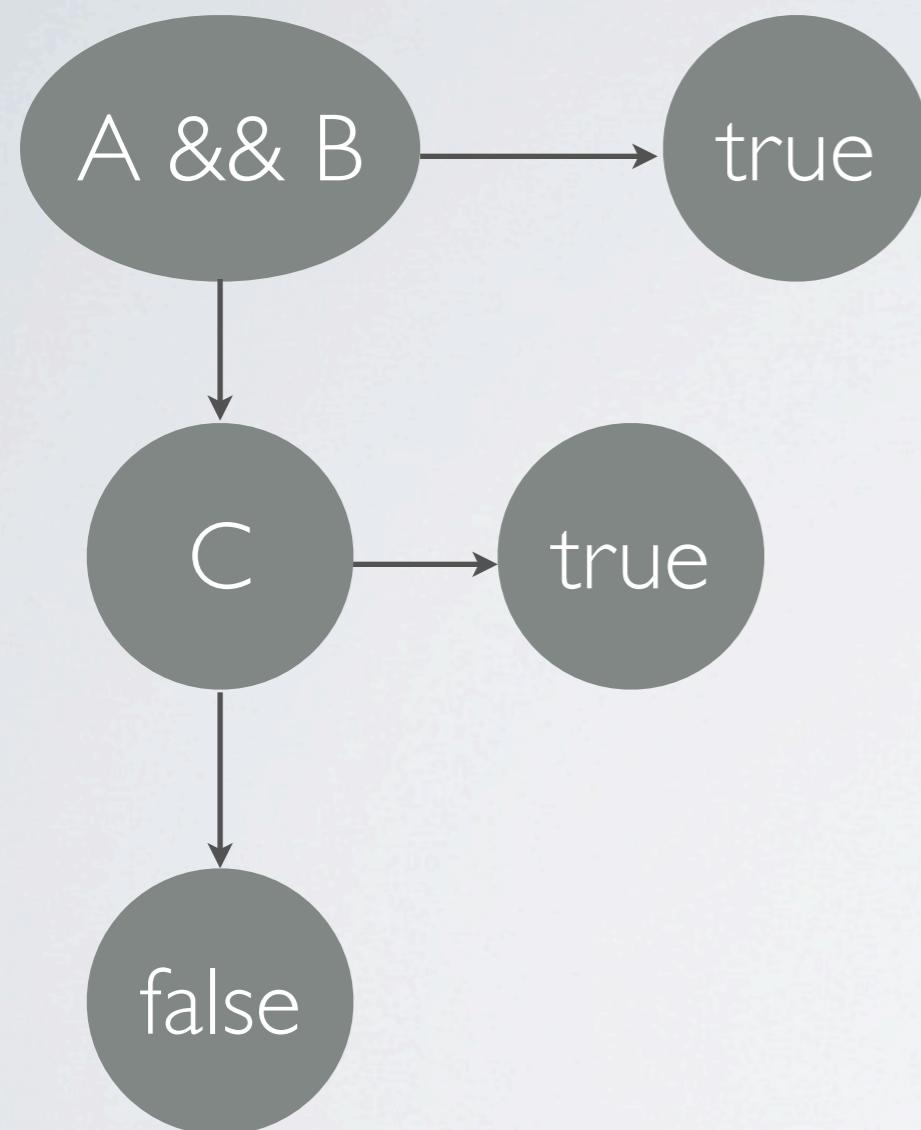
„Eine vollständige Testabdeckung stellt eine Ausnahme dar, weil die Anzahl möglicher Testfälle sehr schnell ungeheuer groß wird (durch kombinatorische Explosion). Ein vollständiger Funktionstest für eine einfache Funktion, die zwei 16-Bit-Werte als Argument erhält, würde schon $2^{(16+16)}$, also ca. 4 Milliarden Testfälle bedeuten, um die Spezifikation vollständig zu testen.“
- Wikipedia

Testabdeckung beim White-Box Testen

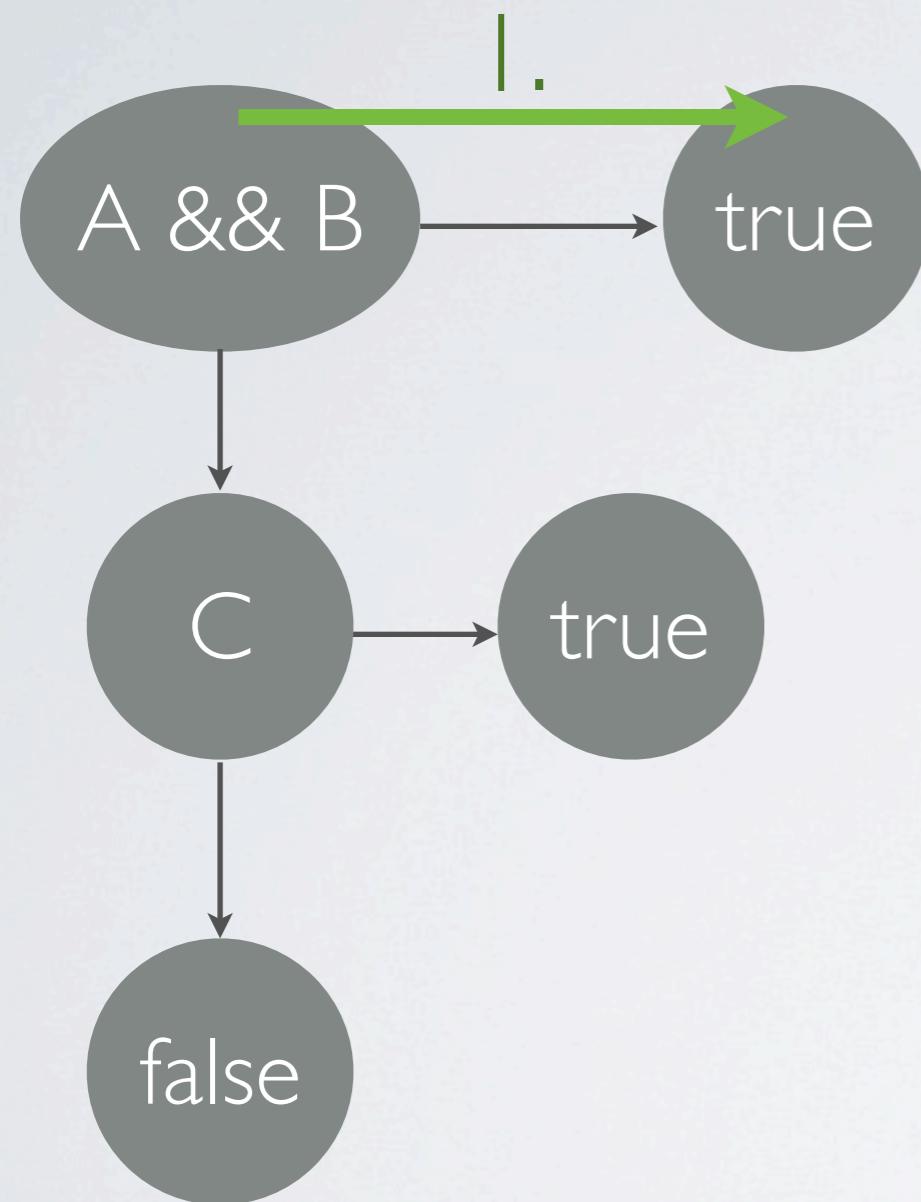
Statement Coverage (Anweisungsüberdeckung)

```
public static boolean aAndBOrC(boolean A, boolean B, boolean C) {  
    if(A && B){  
        return true;  
    }  
    else if(C) {  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

Statement Coverage (Anweisungsüberdeckung)

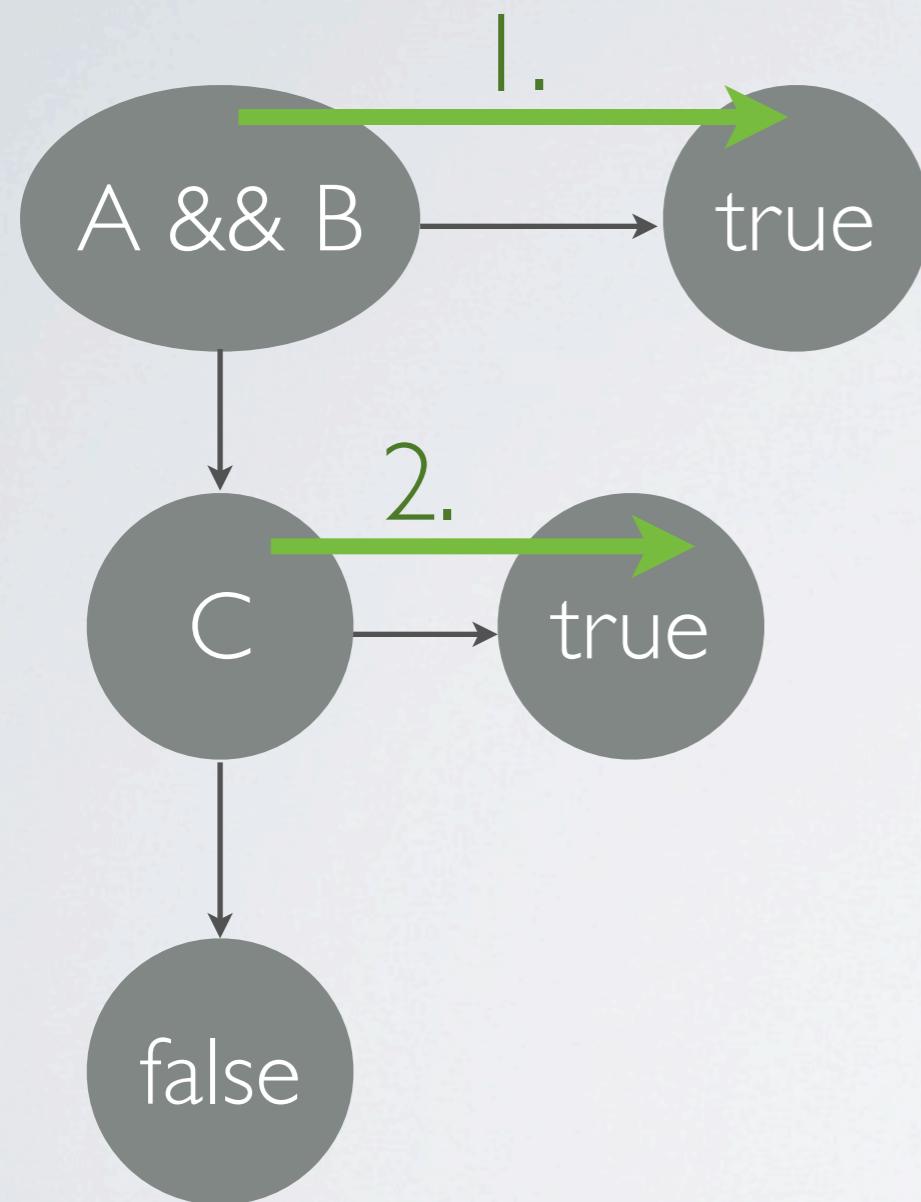


Statement Coverage (Anweisungsüberdeckung)



Erwartet	A	B	C
true	true	true	false

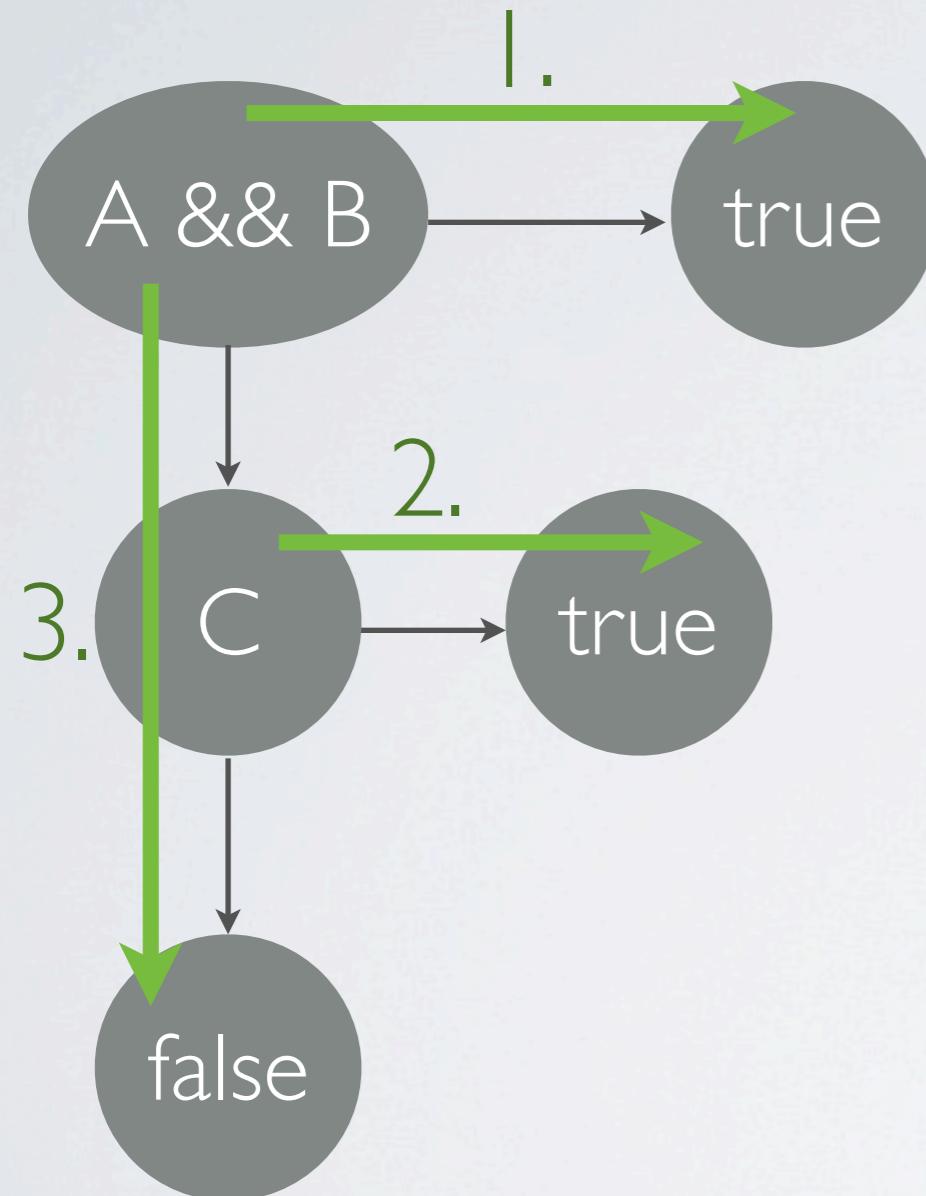
Statement Coverage (Anweisungsüberdeckung)



Erwartet	A	B	C
true	true	true	false
2.	true	false	true

Statement Coverage (Anweisungsüberdeckung)

100 %



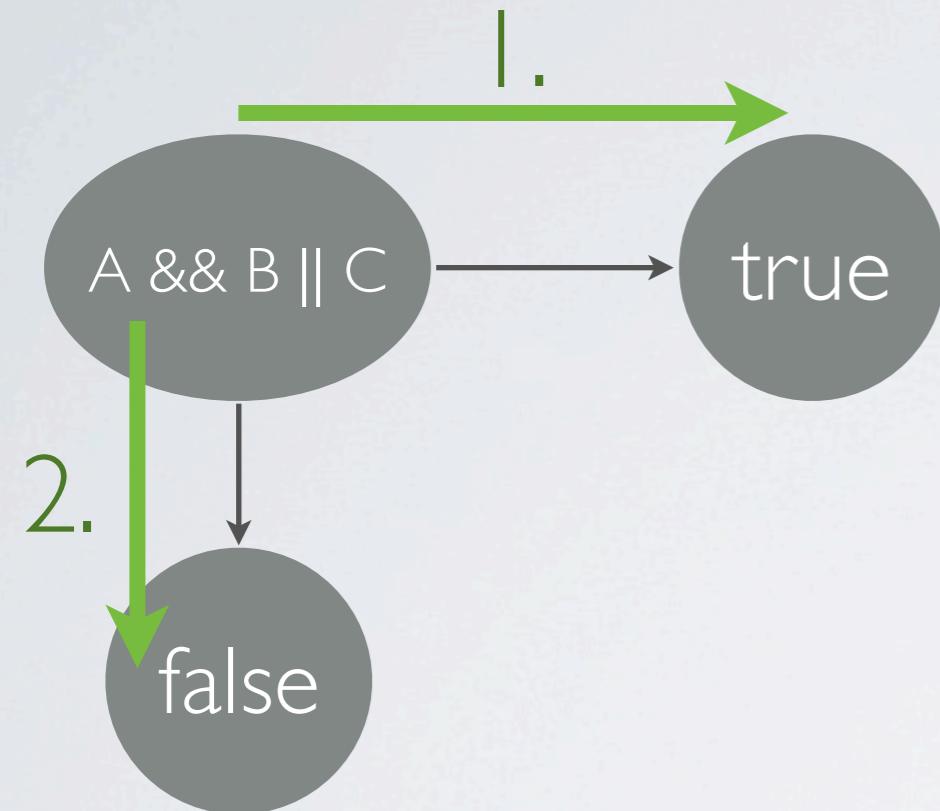
Erwartet		A	B	C
1.	true	true	true	false
2.	true	false	false	true
3.	false	false	false	false

Statement Coverage (Anweisungsüberdeckung)

```
public static boolean aAndBOrC(boolean A, boolean B, boolean C) {  
    if(A && B || C){  
        return true;  
    }  
    else {  
        return false;  
    }  
}
```

Statement Coverage (Anweisungsüberdeckung)

100%



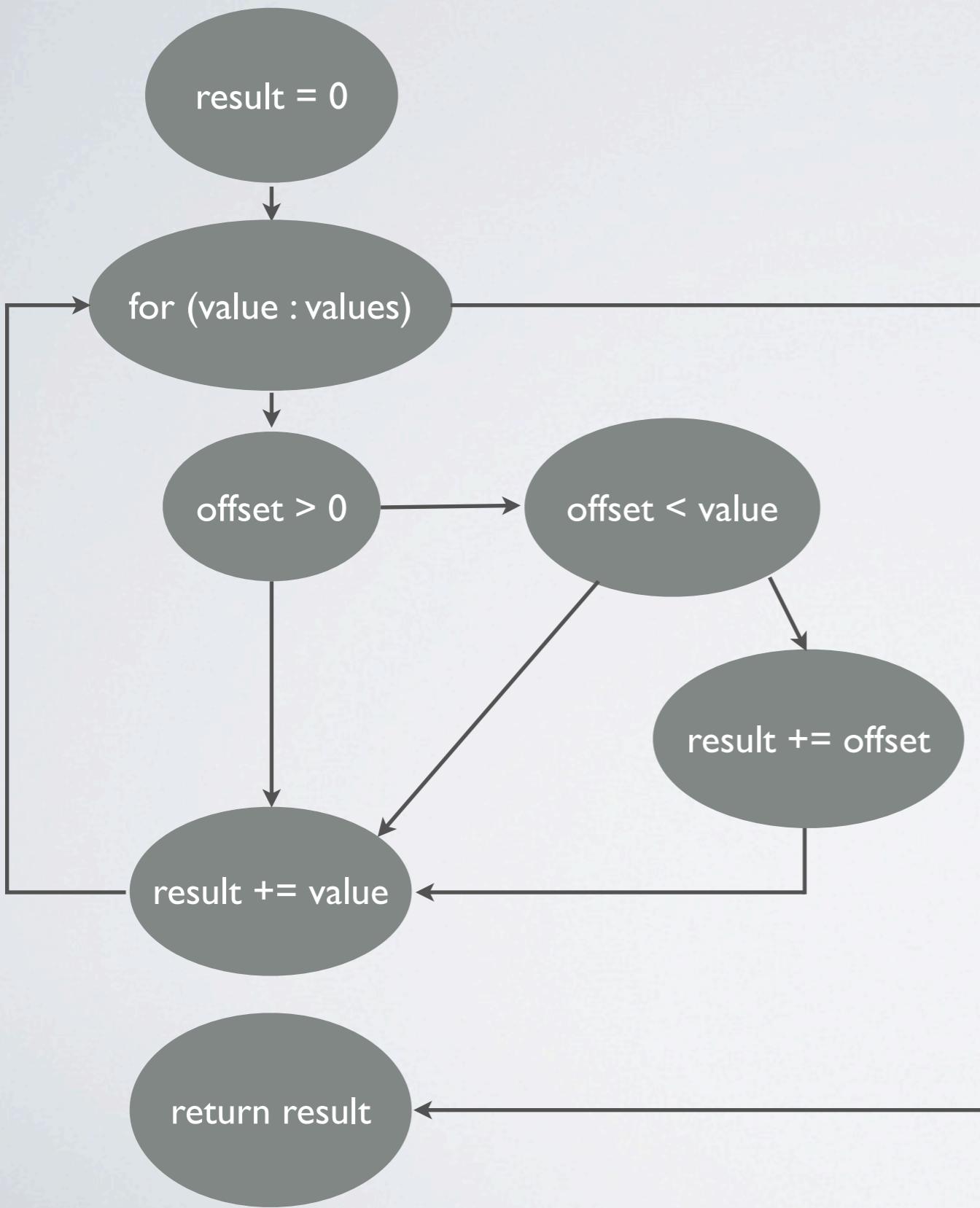
Erwartet	A	B	C	
1.	true	false	false	true
2.	false	false	false	false

Statement Coverage (Anweisungsüberdeckung)

```
public static int sum(int values[], int offset){  
    int result = 0;  
    for (int value : values) {  
        if(offset > 0){  
            if(offset < value)  
                result += offset;  
        }  
        result += value;  
    }  
    return result;  
}
```

Statement Coverage (Anweisungsüberdeckung)

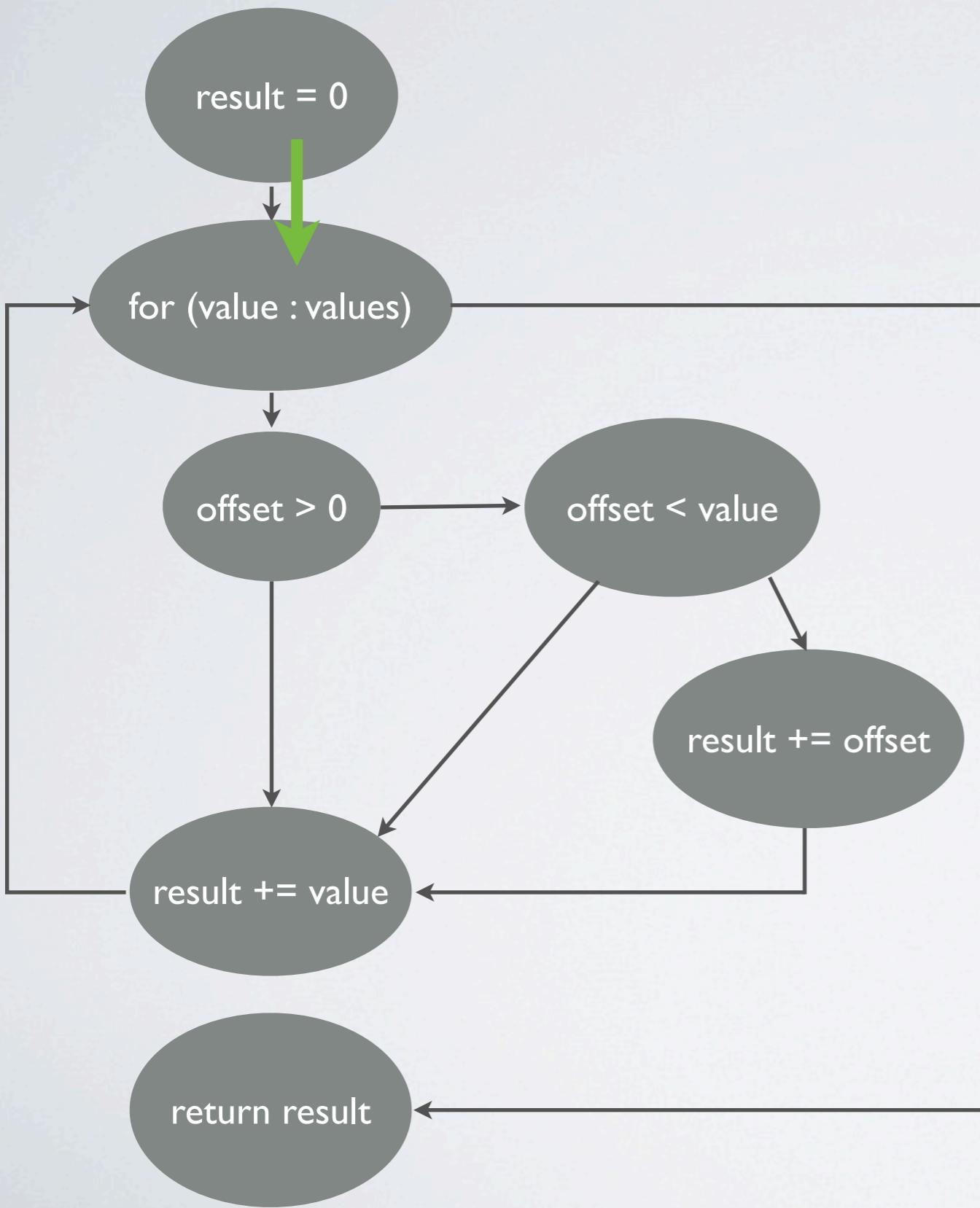
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

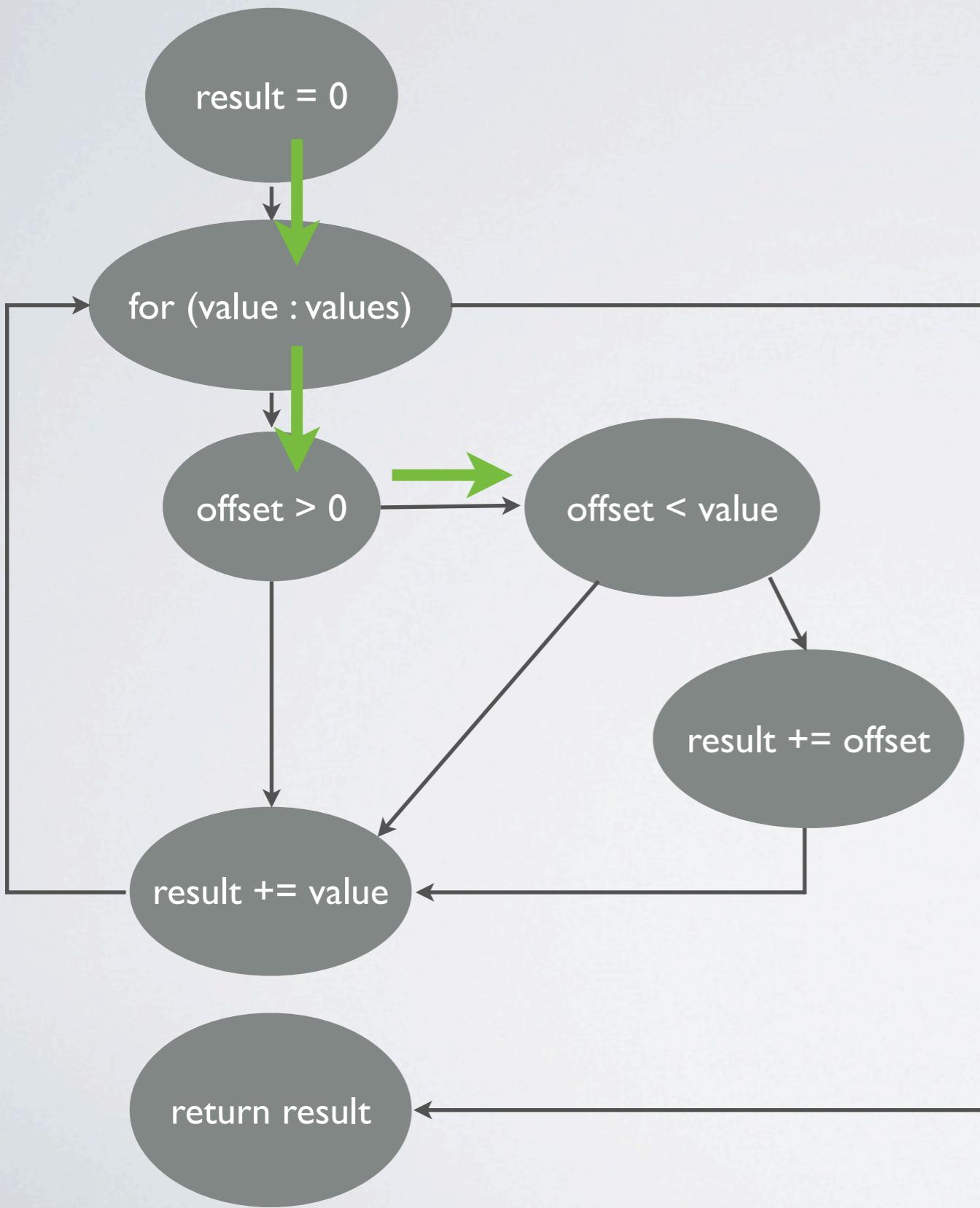
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

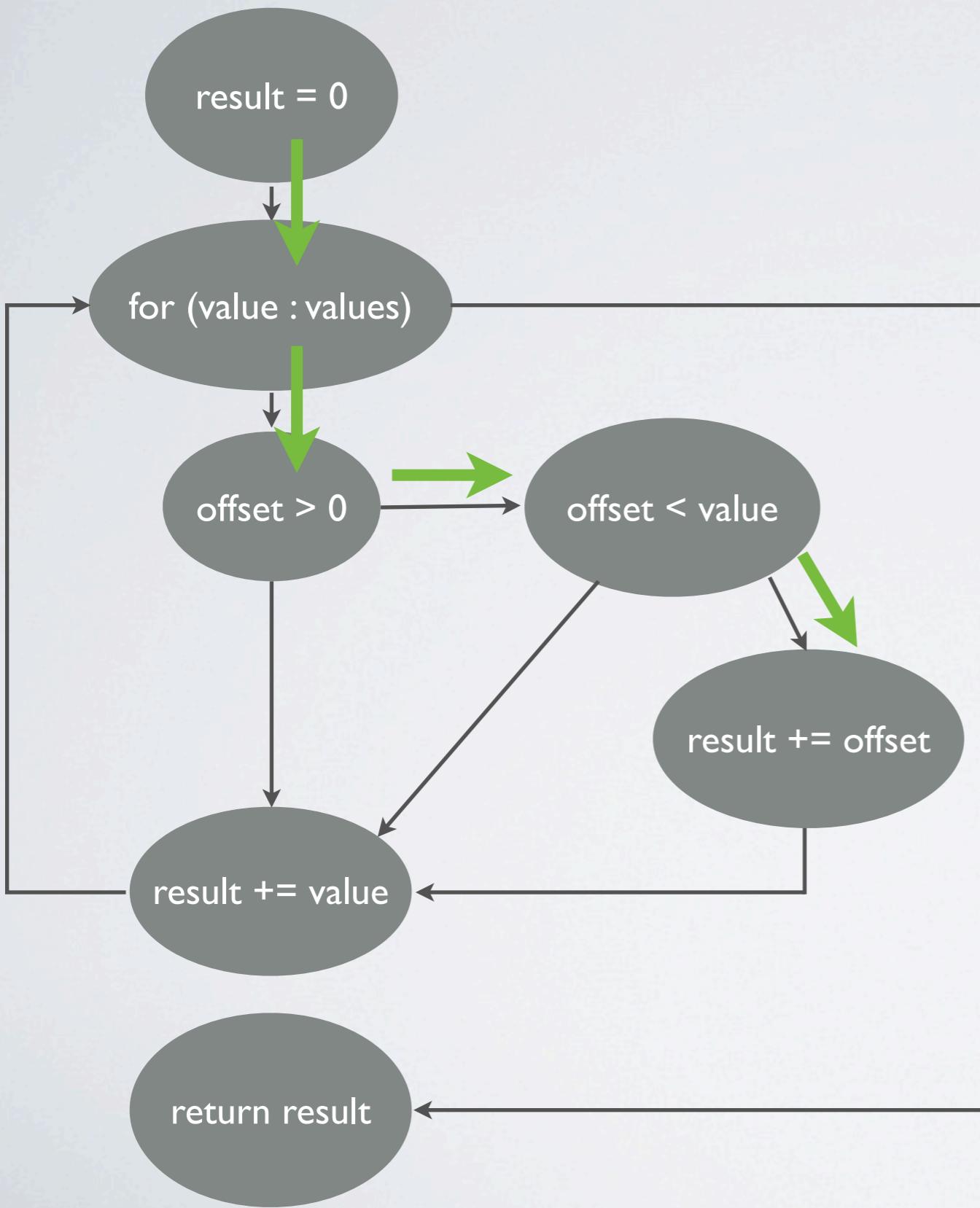
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

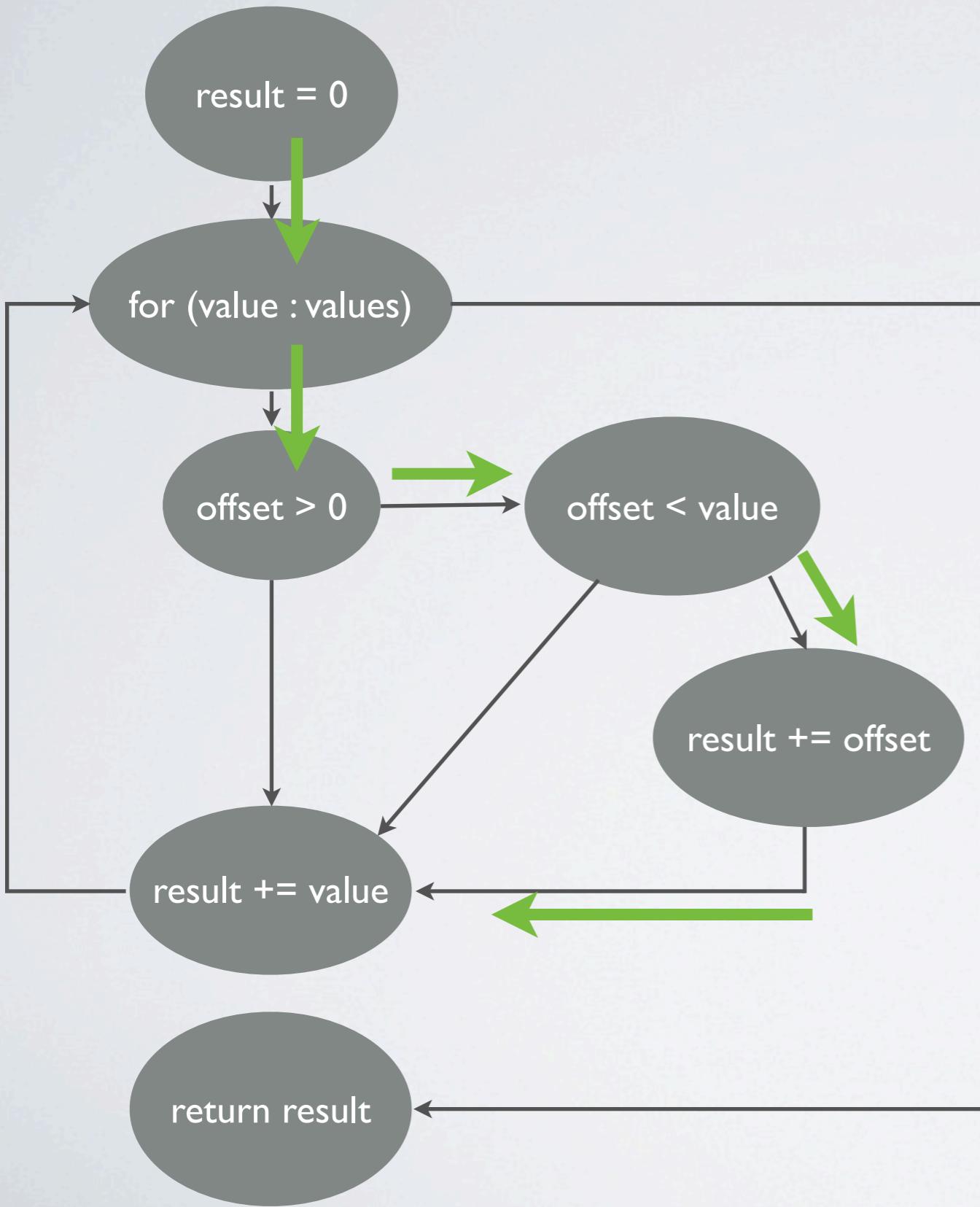
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

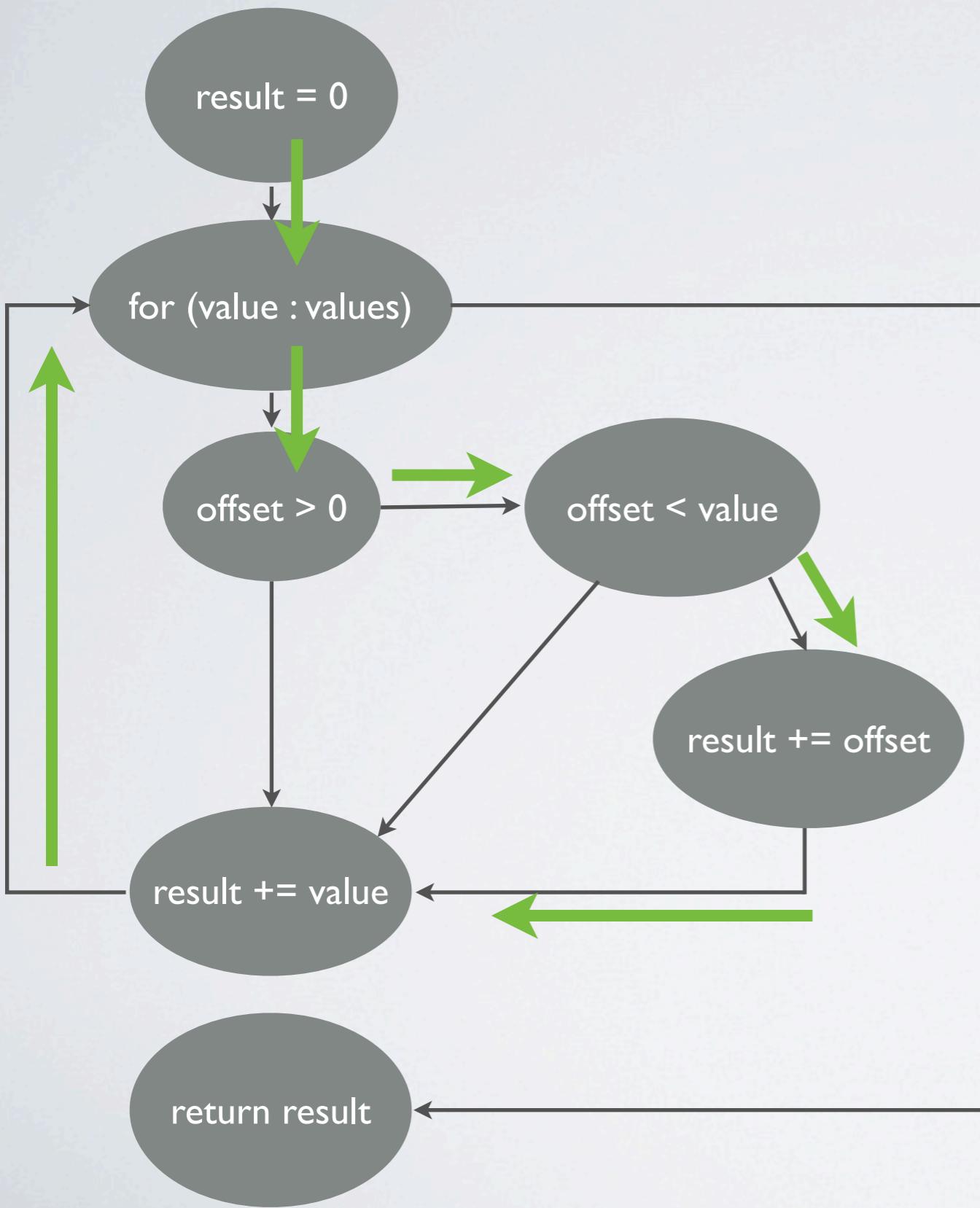
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

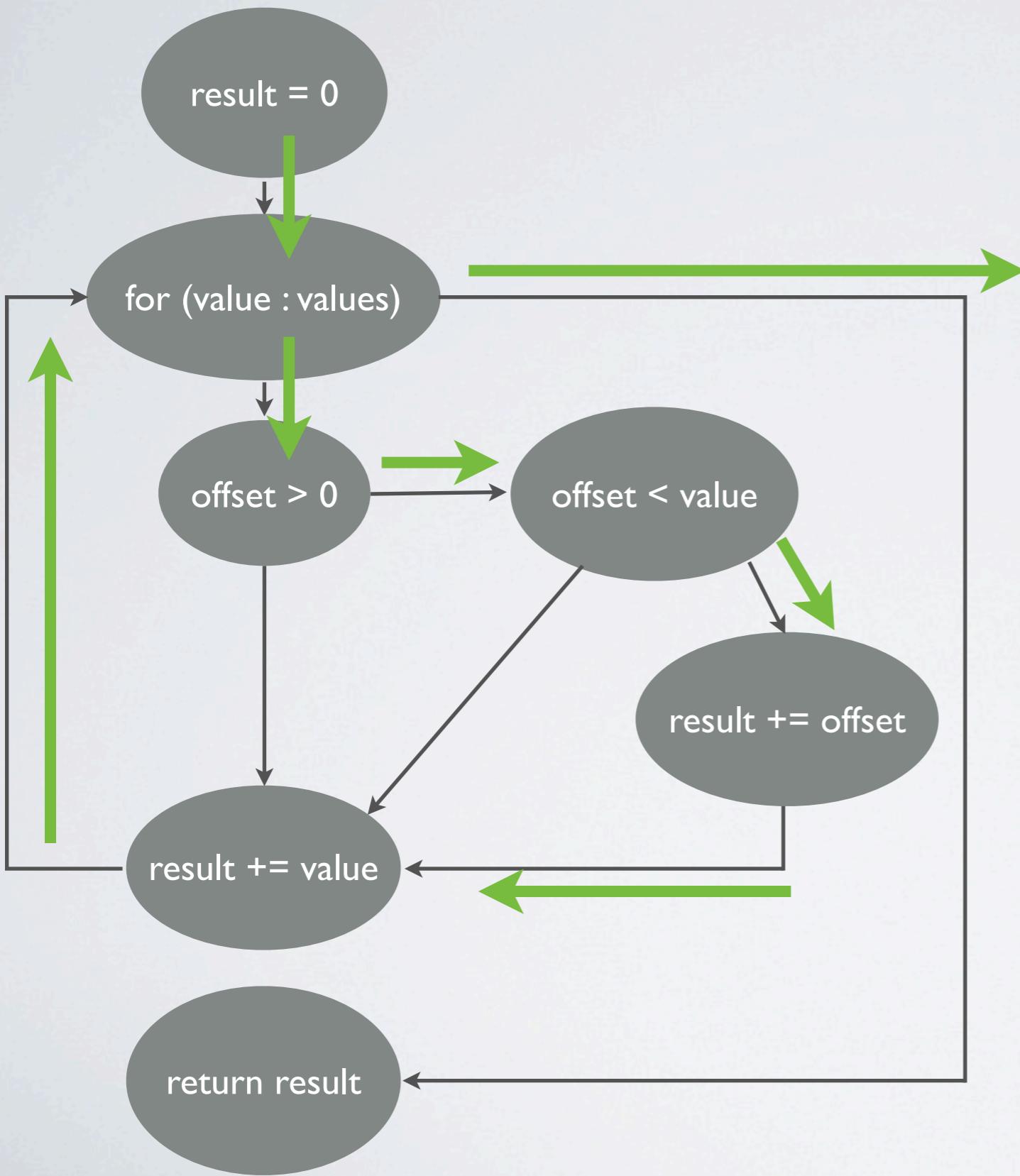
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

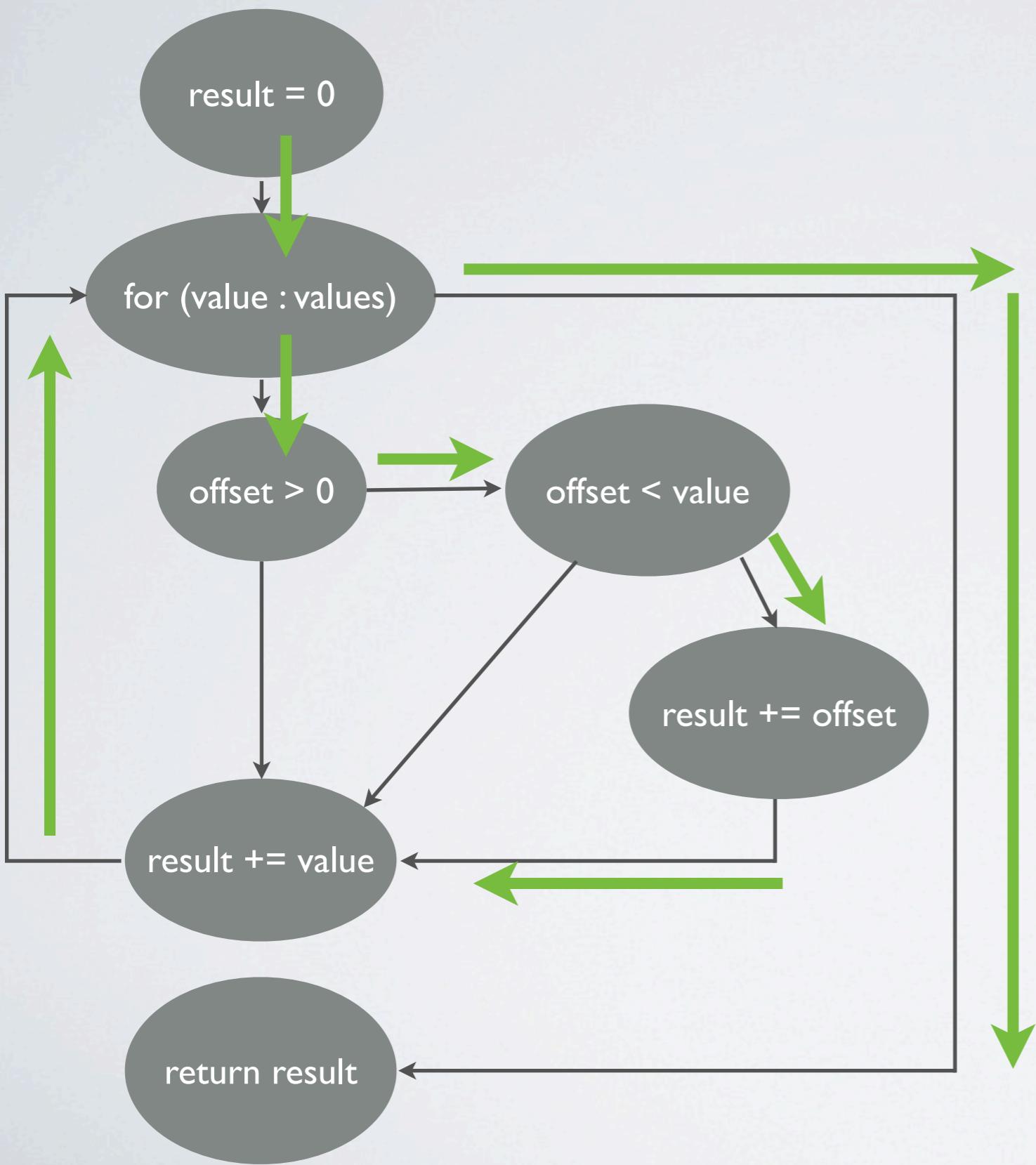
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

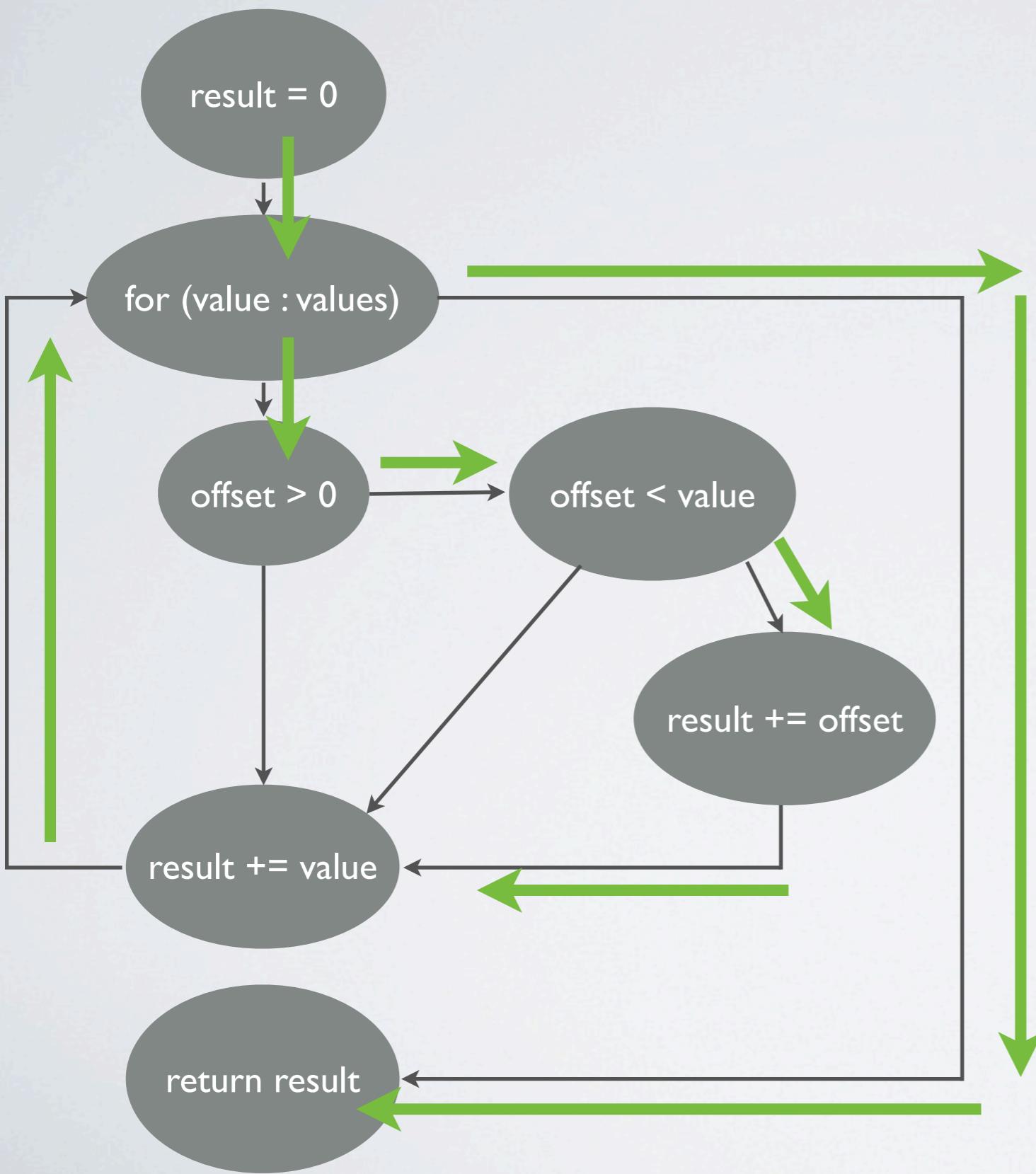
| 100%



Erwartet	values	offset
3	[2]	1

Statement Coverage (Anweisungsüberdeckung)

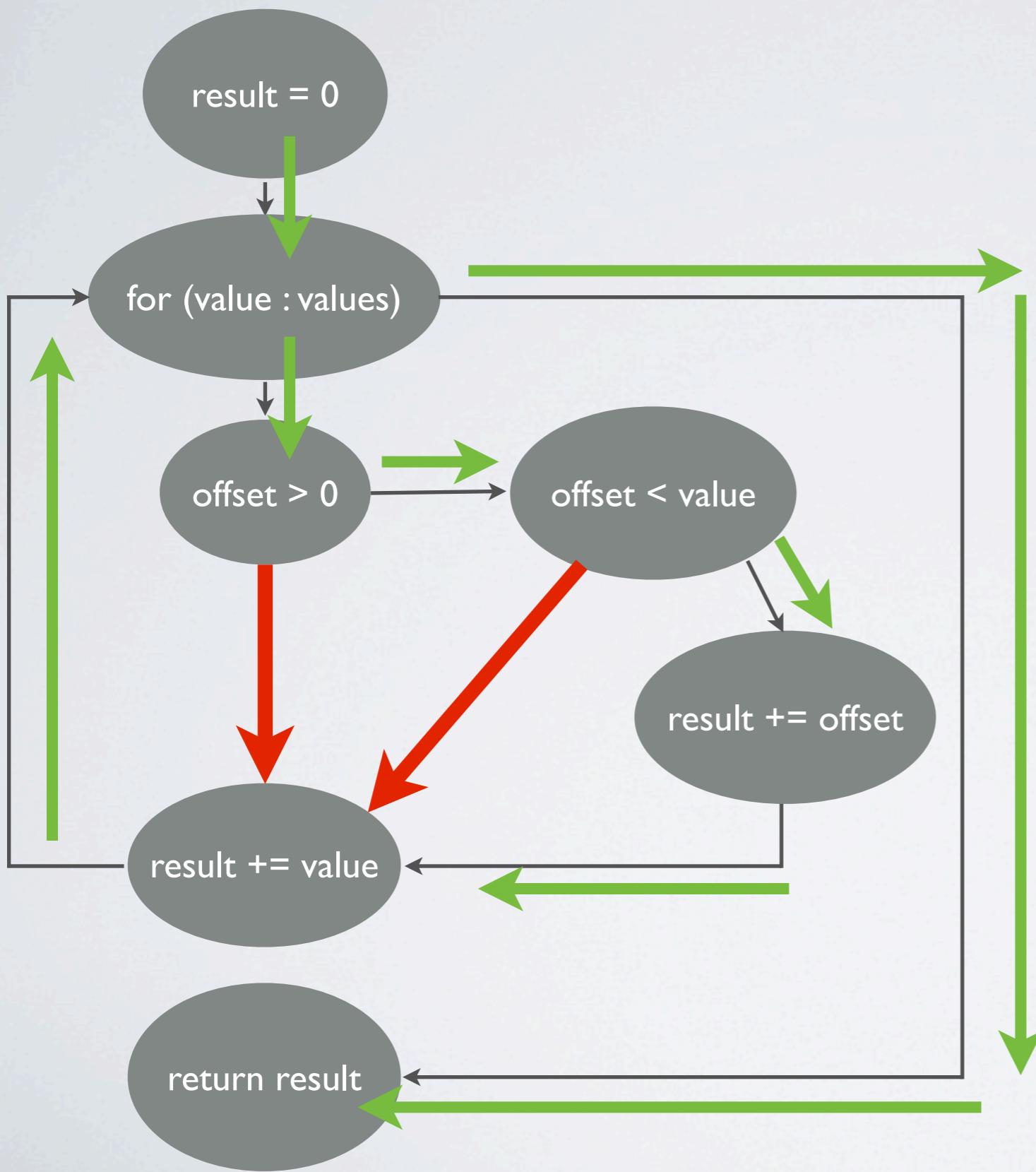
| 100%



Erwartet	values	offset
3	[2]	1

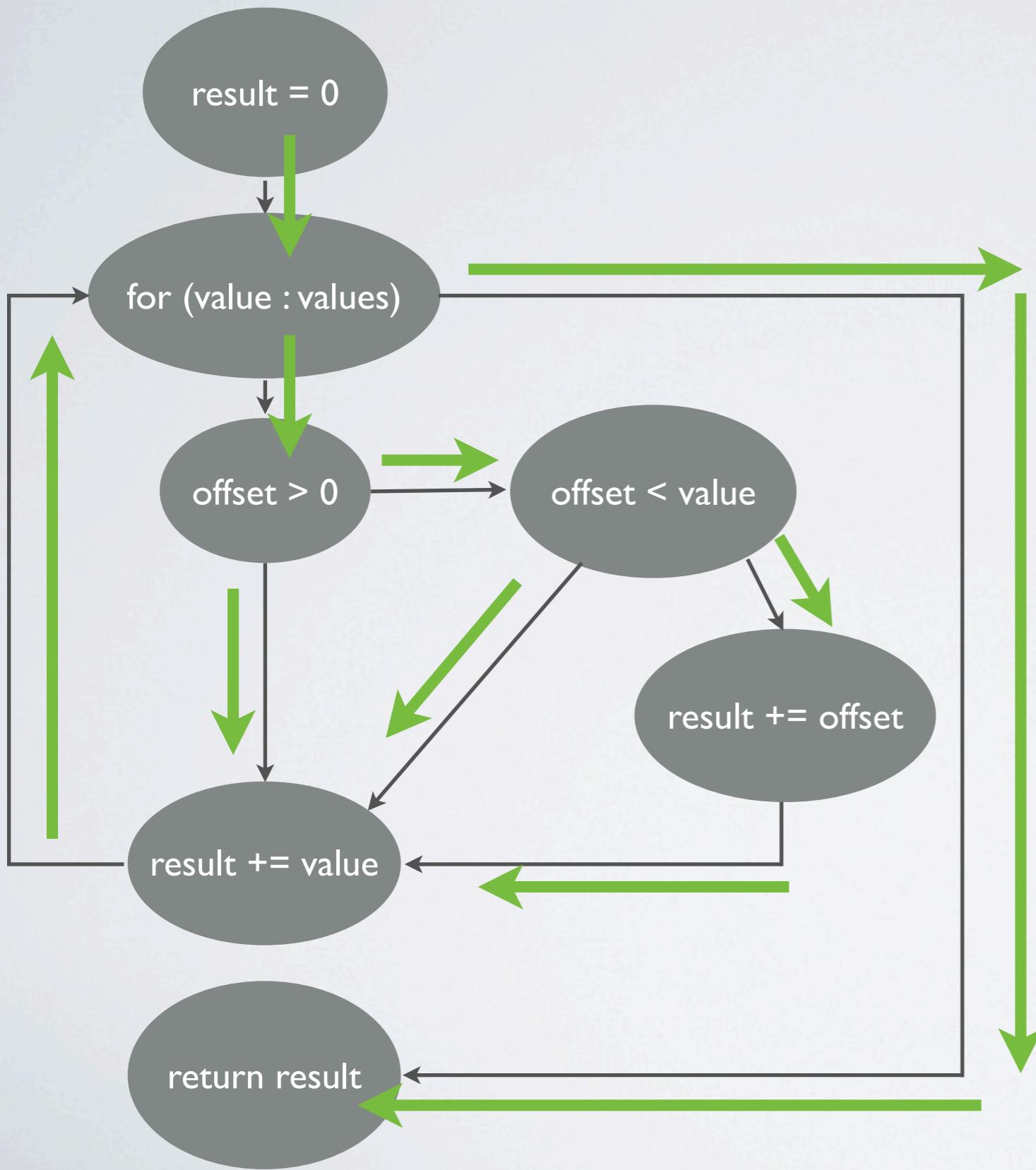
Statement Coverage (Anweisungsüberdeckung)

100 %



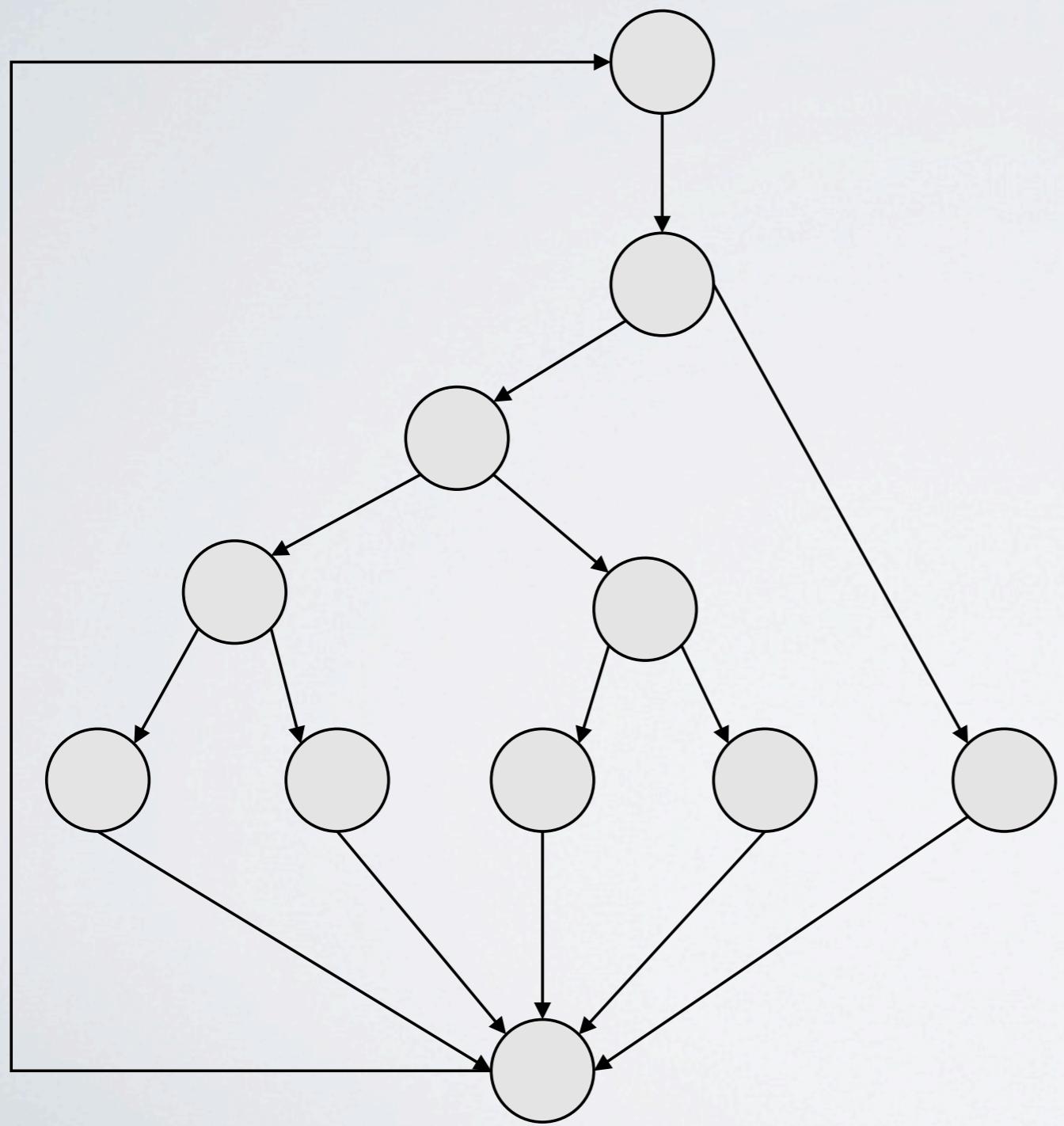
Erwartet	values	offset
3	[2]	1

Branch Coverage 100 % (Zweigabdeckung)

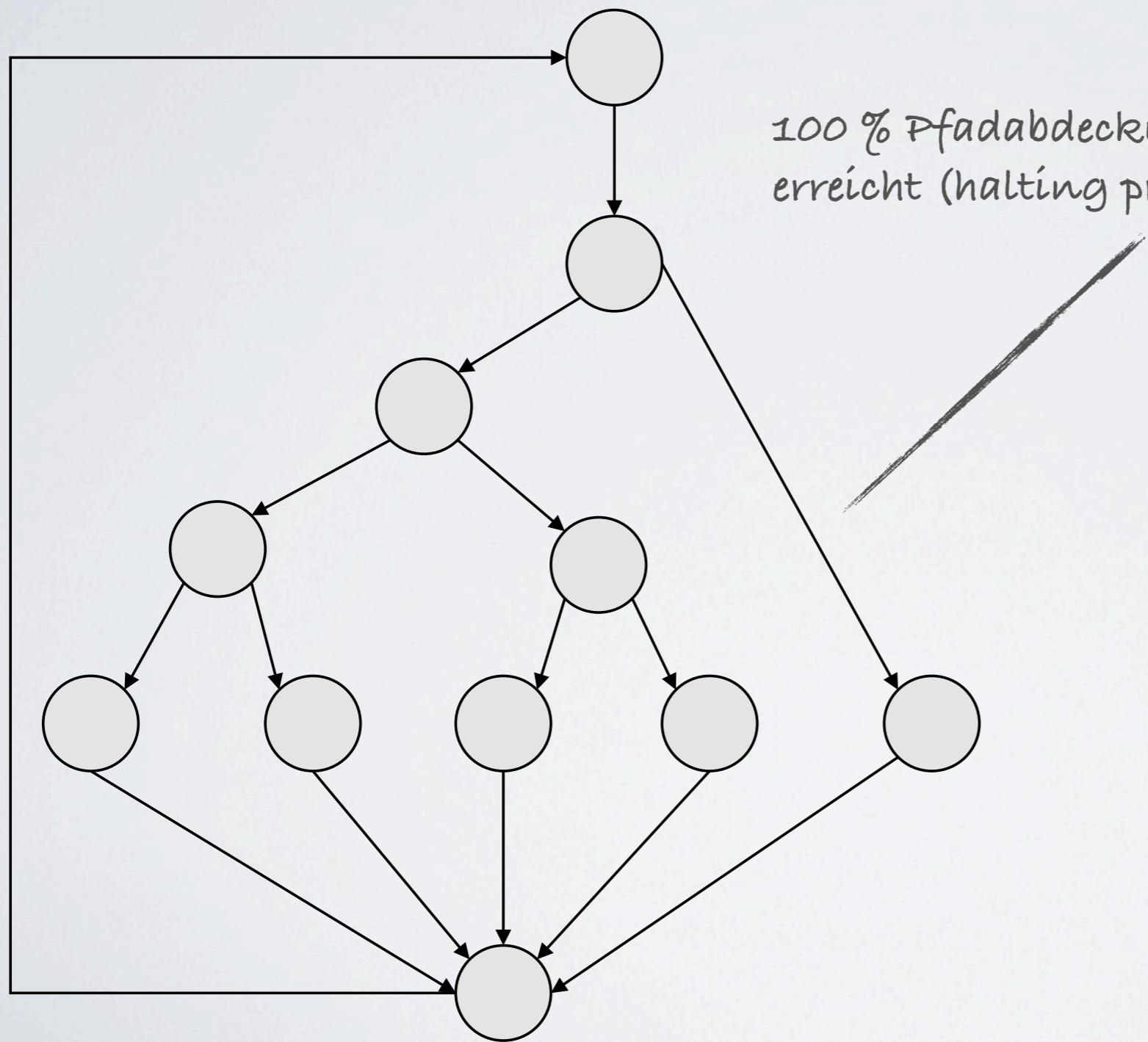


Erwartet	values	offset
3	[2]	1
2	[2]	0
2	[2]	2

100% Pfadabdeckung



100% Pfadabdeckung



100 % Pfadabdeckung wird in der Regel nicht erreicht (halting problem)

Testabdeckung automatisiert bestimmen

The screenshot shows a Java IDE interface with two main windows. The top window displays the `SumUtil.java` file, which contains the following code:

```
package swqs.coverage;

public class SumUtil {

    public static int sum(int values[], int offset){
        int result = 0;
        for (int value : values) {
            if(offset > 0){
                if(offset < value)
                    result += offset;
            }
            result += value;
        }
        return result;
    }
}
```

The code is annotated with color-coded highlights: green for statements, yellow for loops, red for conditionals, and blue for comments. The bottom window is a coverage analysis tool with the following interface:

- Toolbar: Test Sessions, Coverage (selected), Boolean Analyzer, Pick Test Cases, Correlation, Problems.
- Filter: Show methods with Statement Coverage ≤ 90,5 %
- Table:

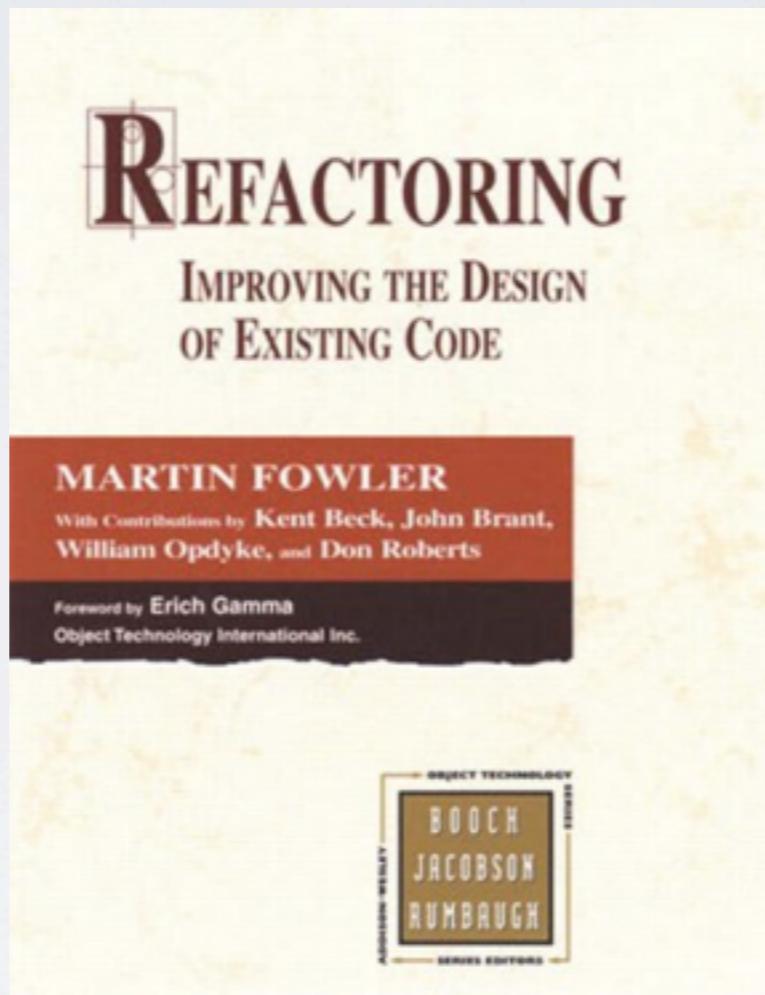
Name	Statement	Branch	Loop	Term	?-Operator	Synchroni
Quicksort	100,0 %	100,0 %	33,3 %	0,0 %	-	-
swqs	100,0 %	100,0 %	33,3 %	0,0 %	-	-
coverage	100,0 %	100,0 %	33,3 %	0,0 %	-	-
SumUtil	100,0 %	100,0 %	33,3 %	0,0 %	-	-

Übung I

- Messen Sie in Eclipse die Code Abdeckung Ihres Quicksort-Tests
- Versuchen Sie den Test so zu erweitern dass Sie eine besser Abdeckung bekommen.



Refactoring Grundlagen



Refactoring

„Refactoring bezeichnet in der Software-Entwicklung die manuelle oder automatisierte Strukturverbesserung von Quelltexten unter Beibehaltung des beobachtbaren Verhaltens. Dabei sollen die Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit verbessert werden, mit dem Ziel, den jeweiligen Aufwand für Fehleranalyse und funktionale Erweiterungen deutlich zu senken.“

- Wikipedia

Fundamentaler Refactoring Prozess

- 1.) Erstelle einen Test (Unit-Test) mit möglich 100% Abdeckung
- 2.) Führe den Test aus (Green bar)
- 3.) Refactoring
- 4.) Führe den Test aus (Green bar)

Refactoring Patterns

- Rename Class, Field, Method
- Extrahiere Methode
- Extrahiere lokale Variable
- Extrahiere Konstante
- Extrahiere Interface
- ...

Refactoring Patterns in Eclipse

The screenshot shows the Eclipse IDE interface. The top menu bar includes Source, Refactor (which is highlighted in blue), Navigate, Search, Project, Run, Window, and Help. The Refactor menu is open, displaying various refactoring options with their keyboard shortcuts. Below the menu, a code editor displays Java code for a Quicksort algorithm. The code uses a bubble sort as a temporary solution for sorting integers.

```
Quicksort.java - E
protected void bubblesort(Integer[] values) {
    for (int i = 0; i < values.length - 1; i++) {
        for (int j = 0; j < values.length - i - 1; j++) {
            if (values[j] > values[j + 1]) {
                int temp = values[j];
                values[j] = values[j + 1];
                values[j + 1] = temp;
            }
        }
    }
}

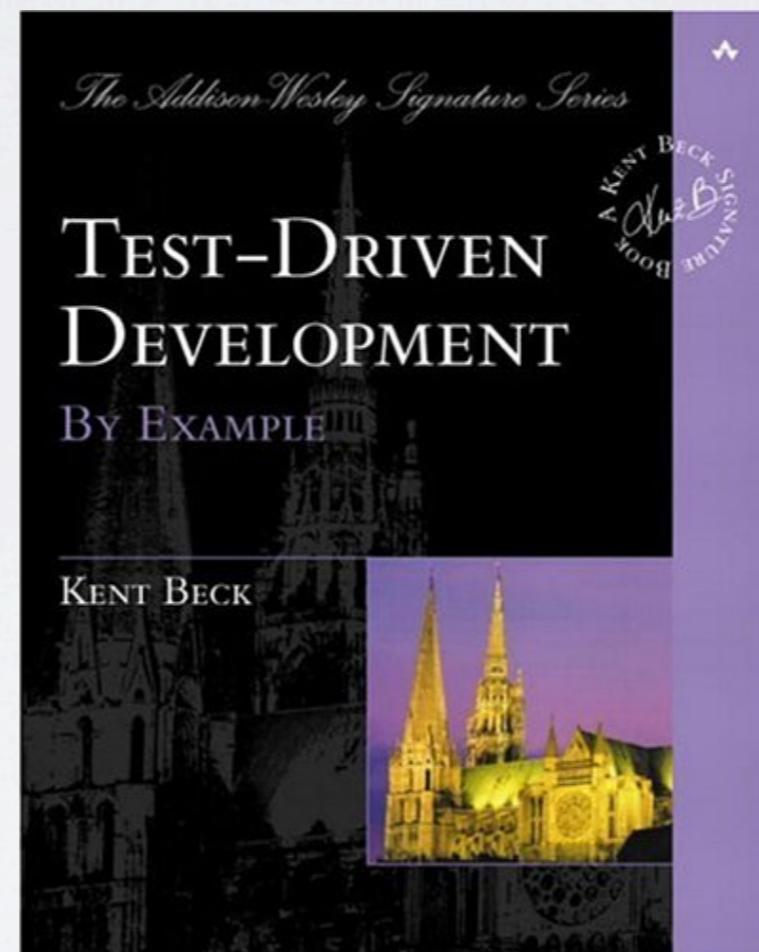
void quicksort(int low, int high) {
    if (low > high)
        return;
    int i = low, j = high;
    Integer pivot = elements[low + (high - low) / 2];
    while (i <= j) {
        while (comparator.compare(elements[i], pivot) < 0)
            i++;
        while (comparator.compare(elements[j], pivot) > 0)
            j--;
        if (i <= j) {
            T temp = elements[i];
            elements[i] = elements[j];
            elements[j] = temp;
            i++;
            j--;
        }
    }
    if (j < high)
        quicksort(j + 1, high);
    if (i < low)
        quicksort(i, low);
}
```

Übung 2

- Probieren Sie die Refactoring Methoden:
Extrahiere Methode, Extrahiere lokale Variable,
Extrahiere Konstante, Extrahiere Interface
In der Quicksort Beispiel Klasse aus
- Nutzen Sie Ihren Test um sicherzustellen
dass Sie lediglich die Struktur ändern



Einführung TDD



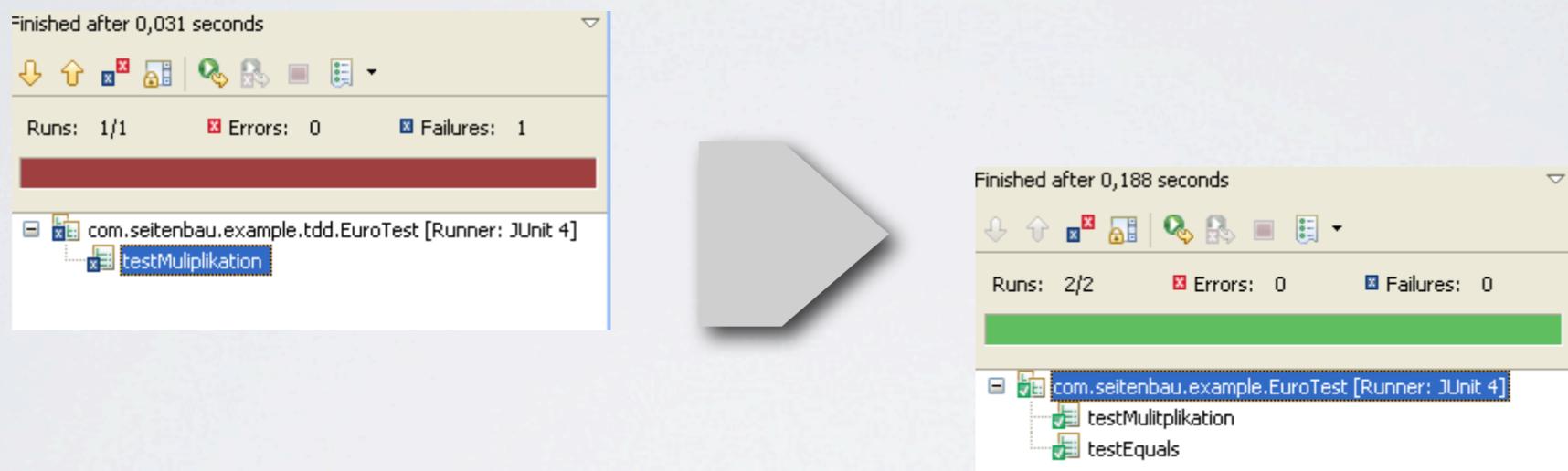
Test Driven Development

Testgetriebene Entwicklung bezeichnet man eine Agile Methode zur Entwicklung einer Software, bei der Software-Tests vor den zu testenden Komponenten entwickelt werden.

Test Driven Development Prozess

- 1.) Erstelle einen kleinen Test
- 2.) Führe alle Tests aus wobei der neue nicht funktioniert (Red bar)
- 3.) Erfülle den Test mit möglichst wenig Code
- 4.) Führe alle Tests aus, wobei alle Grün (Green-Bar)
- 5.) Refactoring - z.B. Entferne duplizierten Code

Red Bar to Green Bar Pattern



Test List Pattern

- Es empfiehlt sich beim TDD eine TODO Liste zu führen mit den Funktionen die noch umgesetzt werden soll
- Man spricht im TDD hier vom Test List Pattern - Test Driven Development by Example Kent Beck

Geld Beispiel

Produkt	Anzahl	Preis	Summe
Füller	10	10 €	100 €
Schokolade	5	25 €	125 €

TODO Liste:

- $10 \text{ €} * 10 = 100 \text{ €}$

Geld Beispiel

TODO Liste:

- ~~+10 € * 10 = 100 €~~
- Betrag sollte private sein
- Euro Nebeneffekte

Geld Beispiel

TODO Liste:

- ~~+10 € * 10 = 100 €~~
- Betrag sollte private sein
- ~~Euro Nebeneffekte~~
- **equals()**
- **hashCode()**

Geld Beispiel

TODO Liste:

- ~~+10 € * 10 = 100 €~~
- Betrag sollte private sein
- ~~Euro Nebeneffekte~~
- ~~equals()~~
- ~~hashCode()~~
- Equals null und object

TDD Vorteile

- Tests
- Hohe Code Abdeckung
- Gray-Box Testing statt White-Box
- Erfüllung der Anforderungen ist messbar
- Durchführung eines Refactoring ist mit wenig Fehlern behaftet
- Die Unit-Testsuite stellt eine „ausführbare Spezifikation“ dar

TDD Nachteile

- Entwickler die keine Erfahrung in der testgetriebenen Entwicklung besitzen ist die Umsetzung schwierig
- Hohes Test KnowHow bei den Entwickler nötig (kann man auch als Vorteil sehen)
- Hohe Anforderungen an Werkzeuge z.B. IDE

Übung 3



- Erstellen Sie Testgetrieben eine Software Komponente die die Anzahl der Wörter in einem Text zählt
- Beispiel: „Hallo hallo Beispiel TEXT für Text.“
Ergebnis:

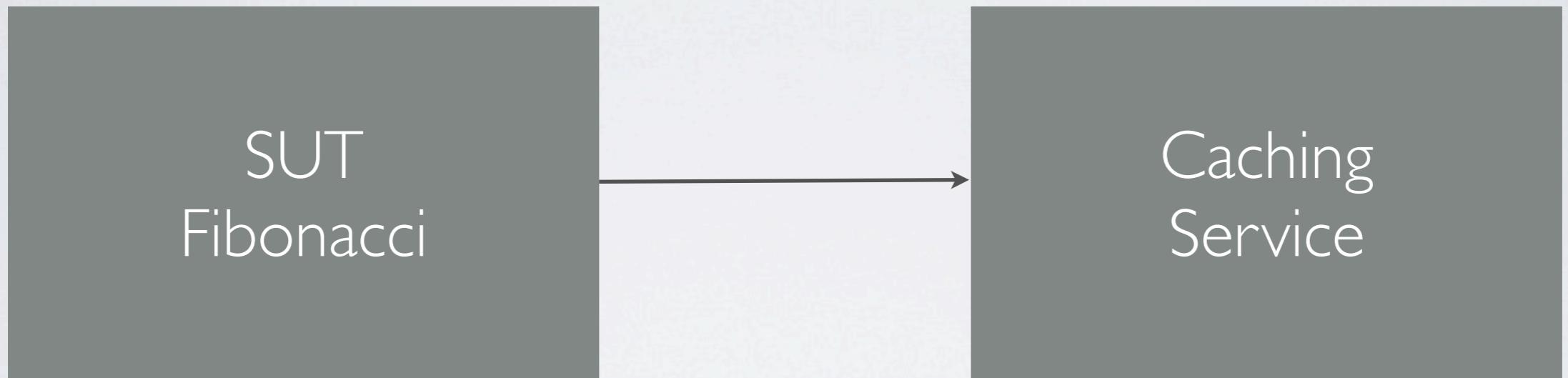
Hallo	Beispiel	Text	für
2		2	

Übung 3

- Messen Sie die Code Abdeckung



Test Dummy und Mock Objekte



Übung 4



- Erstellen Sie für den Caching Service eine Dummy Implementierung und nutzen diese in Ihrem Test.
- Erstellen Sie für den Caching Service ein Mock Objekt via JMock und nutzen diese in Ihrem Test.

Software Qualitätssicherung

HTWG Konstanz

Statische Testverfahren
am 11.11.2010 17:30 bis 20:45

