

Black

/

- Äquivalenztest
- Grenztest

White

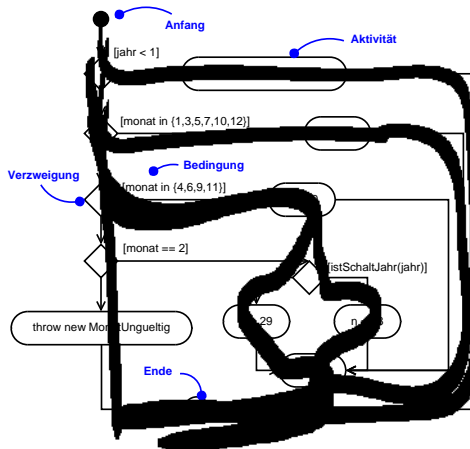
/

Pfadtest

Zustandsbasiert

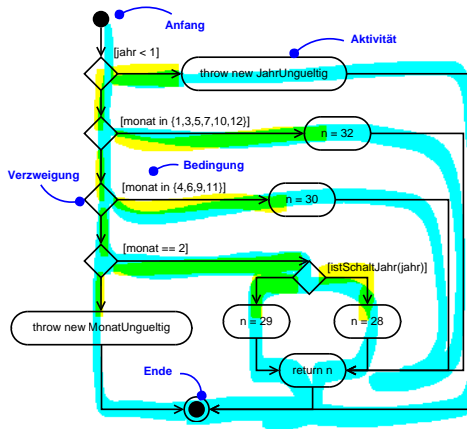
# Maße der Testabdeckung

C0, Anweisungsüberdeckung (Befehlsabdeckung/Statement-Coverage):  
Verhältnis von Anzahl der mit Testdaten durchlaufenen  
Anweisungen zur Gesamtanzahl der Anweisungen.



# Maße der Testabdeckung

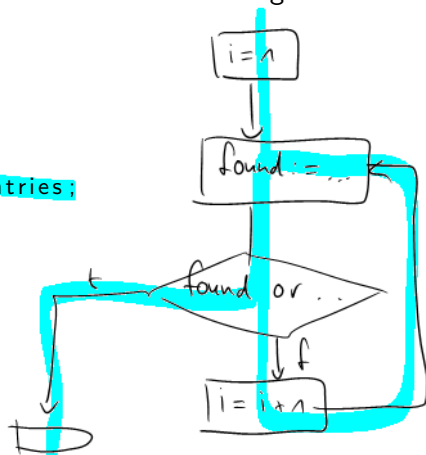
C1, Zweig-/Entscheidungsüberdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Zweige zur Gesamtanzahl der Zweige.



# Maße der Testabdeckung

C2, Bedingungsabdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Terme innerhalb von Entscheidungen zur Gesamtanzahl der Terme.

```
i := 1;  
loop  
  found := a(i) = key;  
  exit when found or i = Max_Entries;  
  i := i + 1;  
end loop;
```

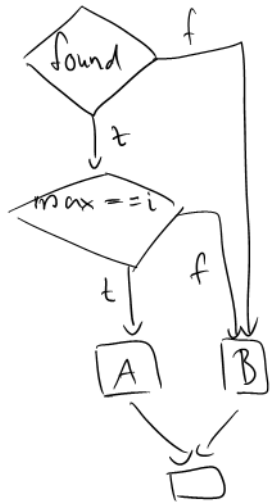
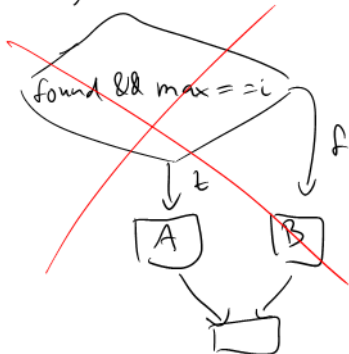


if ( found && max == i ) {

A;

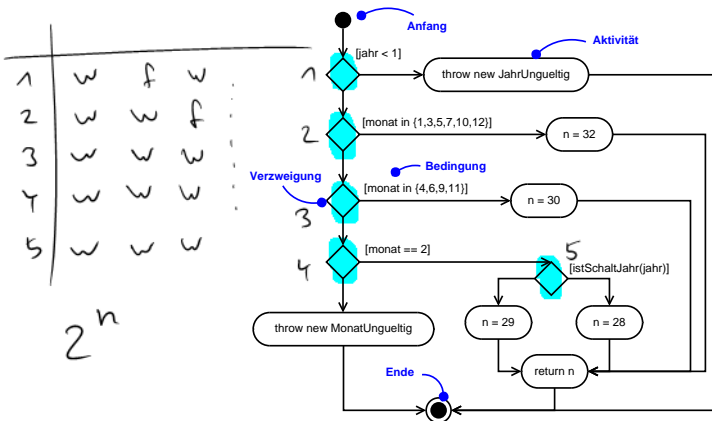
}else

B;



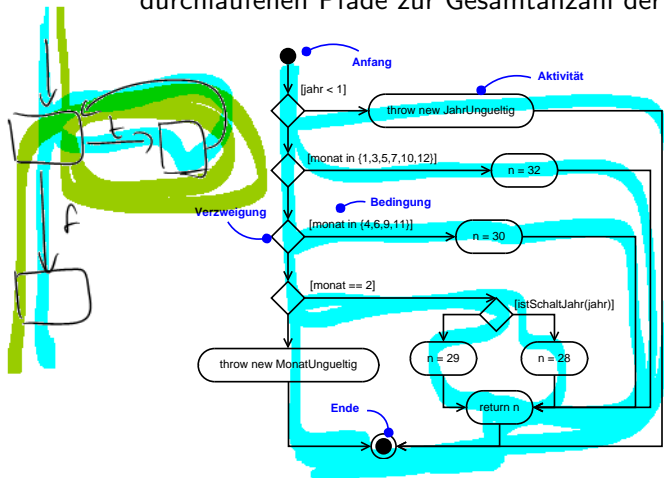
# Maße der Testabdeckung

C3, Abdeckung aller Bedingungskombinationen Verhältnis von Anzahl der mit Testdaten durchlaufenen Bedingungskombinationen zur Gesamtanzahl der Bedingungskombinationen.



# Maße der Testabdeckung

C4, Pfadabdeckung Verhältnis von Anzahl der mit Testdaten durchlaufenen Pfade zur Gesamtanzahl der Pfade.



# Probleme beim Pfadtest

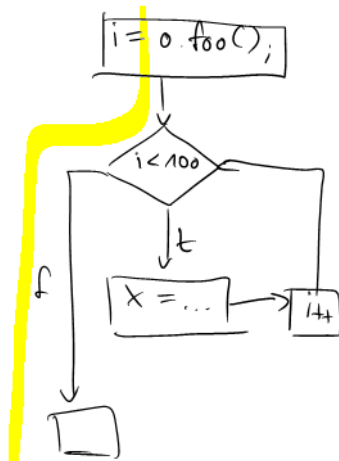
## Unrealisierbare Pfade:

```
for (i = o.foo(); i < 100; i++) {  
    x = ...  
}
```

...x... // hat x definierten Wert?

```
if (p) {...}
```

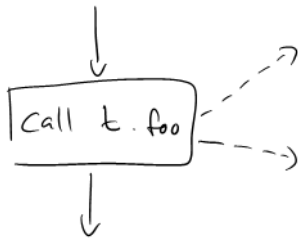
```
if (q) { //  $p \Rightarrow \text{not } q??$   
    ...  
}
```





# Polymorphismustest

```
class T {public int foo();}  
class NT extends T {public int foo();}  
class Factory { T create(int i);}  
  
class UnderTest {  
    void bar (int i)  
    { T t = (new Factory).create(i);  
      if (t.foo() > 0)  
        doSomething();  
    }  
}
```





Quizz

Woher kommt der englische Begriff *Bug* für Fehler?

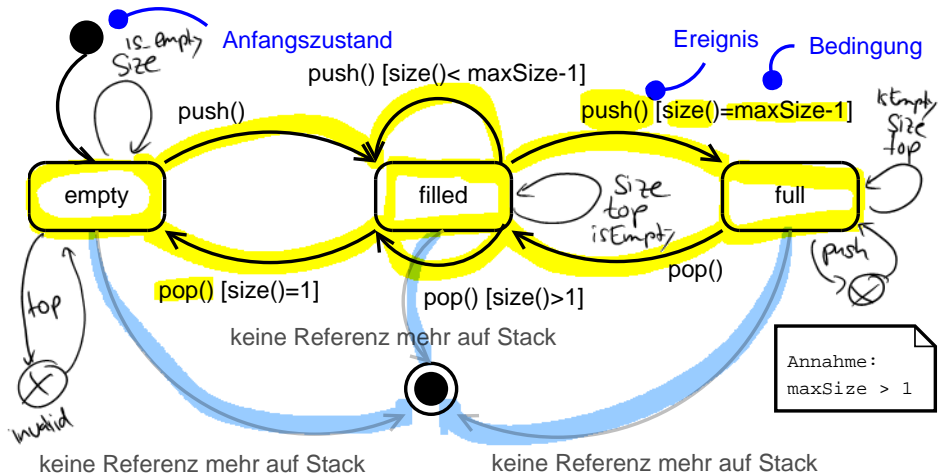
# Zustandsbasiertes Testen

Verhalten von Komponente hängt ab von

- der Eingabe
- ihrem Zustand

```
class Stack {  
    public static class EmptyStack extends Exception {};  
    public static class FullStack extends Exception {};  
  
    public Stack(int maxSize) {...};  
  
    public boolean isEmpty() {...};  
    public int size() {...};  
  
    public Object top () throws EmptyStack {...};  
    public void push (Object o) throws FullStack {...};  
    public void pop () throws EmptyStack {...};  
}
```

# Zustände eines Stacks



Für alle Zustände einer Komponente:

- Komponente wird in zu testenden Zustand gebracht
- Test für alle möglichen Stimuli:
  - korrekte Eingaben,
  - fehlerhafte Eingaben
  - und Aktionen, die Zustandsübergänge bewirken

# Zustandsbasiertes Testen I

```
public void testStateFull() {  
    // test of State full  
    final int max = 100;  
    Stack s = new Stack(max);  
    Object last = new Object();  
  
    // put Stack into state full  
    for (int i = 1; i < max; i++) {  
        last = new Object ();  
        try {s.push (last);}   
        catch (Exception e) {assertTrue (false);};  
    }  
  
    // test legal action 'size'; no transition  
    assertEquals (max, s.size());  
  
    // test legal action 'top'; no transition  
    try {assertEquals (last, s.top ());}  
    catch (Exception e) {assertTrue (false);};  
}
```

# Zustandsbasiertes Testen II

```
// test legal action 'isEmpty'; no transition
```

```
assertEquals (false , s.isEmpty());
```

```
// test illegal action 'push'
```

```
try {s.push(new Object()); assertTrue (false);}
catch (Stack.FullStack e) {assertTrue (true);}
```

```
// test legal action/transition 'pop'
```

```
try {s.pop();}
```

```
catch (Exception e) {assertTrue (false);};
```

```
// reverse transition
```

```
try {s.push(new Object());}
```

```
catch (Exception e) {assertTrue (false);};
```

```
}
```

# Vergleich von White- und Black-Box-Tests

Black-Box-Tests (Funktionstests) betrachten den Prüfling als schwarze Box. Sie setzen keine Kenntnisse über die Interna voraus.

White-Box-Tests (Strukturtests, Glass-Box-Tests) betrachten Interna des Prüflings, um Testfälle zu entwickeln.

Eigenschaft	Black-Box-Test	White-Box-Test
Test auf Basis von	Schnittstellen-spezifikation	Lösung
Wiederverwendung bei Änderung der Struktur	ja	eingeschränkt
Geeignet für Testart	alle	Komponententest
Finden Fehler aufgrund von	Abweichung von Spez.	eher Kodierfehler



# Black- versus White-Box-Tests

Nicht entweder-oder, sondern sowohl-als-auch!

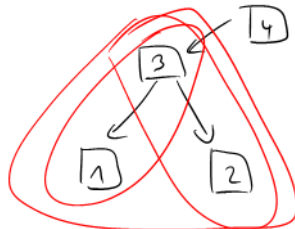
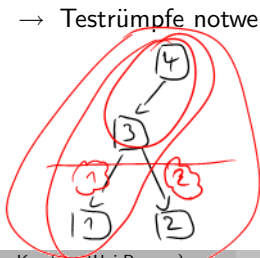
Komplementäre Verwendung:

- ① Erstelle Funktionstest
- ② Messe Abdeckung
- ③ Wenn Abdeckung nicht ausreichend; ergänze durch Strukturtest;  
zurück zu 2.

White-Box-Test/

# Strategien des Integrationstests I

- Urknalltest: alle Komponenten werden einzeln entwickelt und dann in einem Schritt integriert  
→ erschwert Fehlersuche
- Bottom-Up: Integration erfolgt inkrementell in umgekehrter Richtung zur Benutzt-Beziehung  
→ keine Testrumpfe notwendig (aber Testtreiber)  
→ Fehler in der obersten Schicht werden sehr spät entdeckt; die enthalten jedoch die Applikationslogik
- Top-Down: Integration erfolgt in Richtung der Benutzt-Beziehung  
→ Fehler in der obersten Schicht werden sehr früh entdeckt  
→ Testrumpfe notwendig



# Strategien des Integrationstests II

- Sandwich-Test-Strategie: Kombination von Top-Down und Bottom-Up
  - Identifikation der zu testenden Schicht: Zielschicht
  - zuerst: individuelle Komponententests
  - dann: kombinierte Schichttests:
    - Oberschichttest mit Zielschicht
    - Zielschicht mit Unterschicht
    - alle Schichten zusammen



- Härtetest: viele gleichzeitige Anfragen
- Volumentest: große Datenmengen
- Sicherheitstests: Sicherheitslücken aufspüren
- Zeitvorgabentests: werden spezifizierte Antwortzeiten eingehalten?
- Erholungstests: Tests auf Erholung von fehlerhaften Zuständen