

Softwaretechnik 1 - 5. Tutorium

Tutorium 18

Felix Bachmann | 03.07.2018

KIT - INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)



- 1 Orga
- 2 Überblick
- 3 Parallelität
- 4 Testen
- 5 Tipps

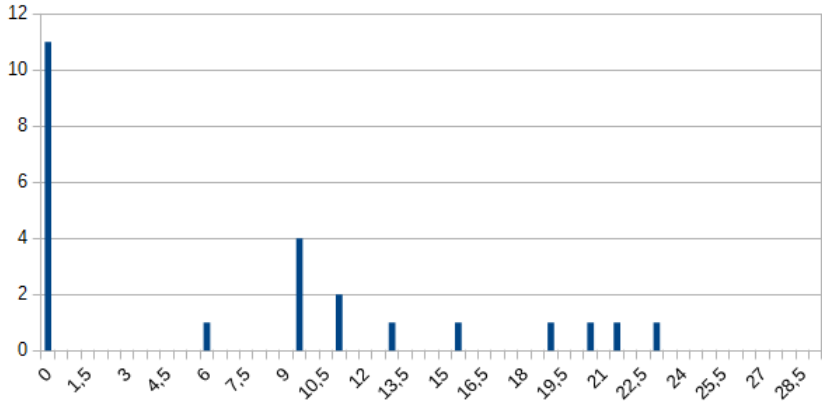
Nächstes Mal letztes Tutorium

- irgendwelche Wünsche für das letzte Tut?
 - etwas bestimmtes wiederholen?
 - falls euch noch was einfällt, schreibt mir eine Mail
⇒ felix.bachmann@ewetel.net

Gut	Schlecht/Verbesserungswürdig
Erklärungen (4) schnelle Korrektur (3) Beispiele (3) Folien (3) Gruppenarbeiten (1)	Feedback genauer/lesbarer (2) Gruppenarbeit generell (1) Gruppenarbeit zu lang (1) Gruppenarbeit zu kurz (1) Gliederungsfolie (1) Evaluation zum Ankreuzen (1)

5. Übungsblatt Statistik

n=24



Ø 7 bzw. 14 von 27+2

Allgemein

- (mal wieder...) CheckStyle und JavaDoc

Aufgabe 1: Shutterpile: Refaktorisierung + Entwurfsmuster

Aufgabe 1: Shutterpile: Refaktorisierung + Entwurfsmuster



Aufgabe 2: cmd-Programm für Pipeline

Aufgabe 2: cmd-Programm für Pipeline



Aufgabe 2: cmd-Programm für Pipeline



Aufgabe 3: Wo sind Entwurfsmuster in Shutterpile?



Aufgabe 4: Entwurfsmuster in Java-API

Aufgabe 4: Entwurfsmuster in Java-API



Aufgabe 4: Entwurfsmuster in Java-API



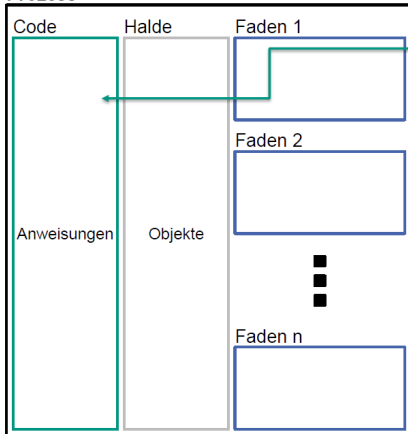
Aufgabe 5: Entwurfsmuster - Kaffeemaschine



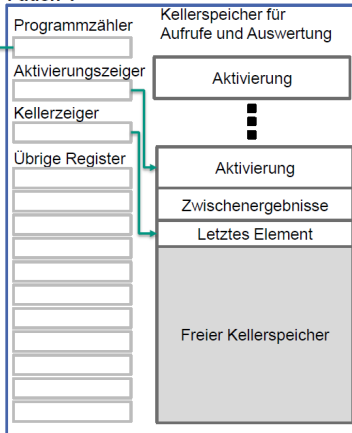
Falls ihr noch Punkte braucht, gebt ab!

Wo sind wir?

Prozess



Faden 1



- Prozess = Programm in Ausführung

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
 - Threads haben den gleichen Heap und Code

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
 - Threads haben den gleichen Heap und Code
⇒ alle Threads innerhalb eines Prozesses arbeiten mit denselben Objekten und demselben Code

- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
 - Threads haben den gleichen Heap und Code
⇒ alle Threads innerhalb eines Prozesses arbeiten mit denselben Objekten und demselben Code
 - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)

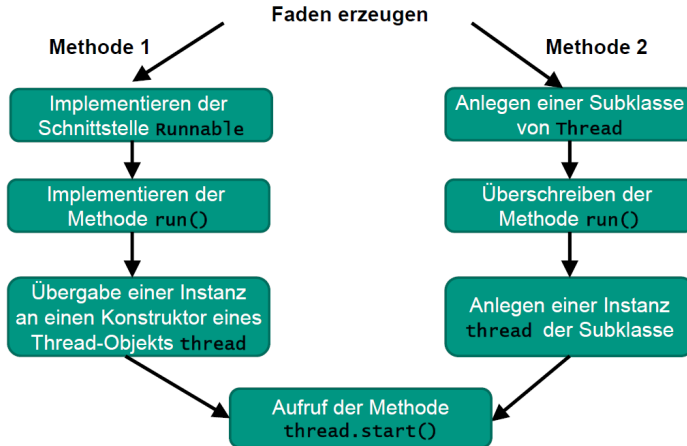
- Prozess = Programm in Ausführung
- jeder Prozess hat eigenen Adressraum (= Speicherbereich im Arbeitsspeicher)
- jeder Prozess hat mindestens einen Thread
- Threads existieren innerhalb eines Prozesses
 - Threads haben den gleichen Heap und Code
⇒ alle Threads innerhalb eines Prozesses arbeiten mit denselben Objekten und demselben Code
 - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)
⇒ Threads haben eigene lokale Variablen und können beliebigen Code des Prozesses ausführen

```
// will freeze the gui when the button is clicked
JButton heavy = new JButton("Freeze");
heavy.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // heavy work
    }
});
```

```
// will freeze the gui when the button is clicked
JButton heavy = new JButton("Freeze");
heavy.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // heavy work
    }
});
```

- “normales“ sequentielles Programm = 1 Prozess mit 1 Thread
- paralleles Programm = 1 Prozess mit mehreren Threads

- in Java zwei Möglichkeiten einen Thread zu erstellen
- bereits in Java enthalten:
 - Interface `java.lang.Runnable`
 - Klasse `java.lang.Thread`



```
// method 2
Thread method2Thread = new Thread() {
    @Override
    public void run() {
        // do stuff
    }
};
method2Thread.start();

// method 1
Thread method1Thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // do stuff
    }
});
method1Thread.start();

method2Thread.join(); // wait until completion
method1Thread.join();
```

```
// method 2
Thread method2Thread = new Thread() {
    @Override
    public void run() {
        // do stuff
    }
};
method2Thread.start();

// method 1
Thread method1Thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // do stuff
    }
});
method1Thread.start();

method2Thread.join(); // wait until completion
method1Thread.join();
```

Wichtig!

- immer Thread.start() aufrufen, nicht Thread.run()

```
// method 2
Thread method2Thread = new Thread() {
    @Override
    public void run() {
        // do stuff
    }
};
method2Thread.start();

// method 1
Thread method1Thread = new Thread(new Runnable() {
    @Override
    public void run() {
        // do stuff
    }
});
method1Thread.start();

method2Thread.join(); // wait until completion
method1Thread.join();
```

Wichtig!

- immer Thread.start() aufrufen, nicht Thread.run()
- Thread.run() würde run() sequenziell aufrufen, start() kehrt direkt zurück, nachdem Thread gestartet wurde

- Problem: Zugriff auf globale Variablen/ Objekte passiert nicht parallel, Unterbrechungen möglich
- Folge: ggf. falsche Ergebnisse

Faden 1

```
                // globalVar == 1  
  
if (globalVar > 0) {  
  
    globalVar--;  
}
```

Faden 2

```
if (globalVar > 0) {  
    globalVar--;  
}
```

■ nicht nur ein theoretisches Beispiel!

```
for (int i = 0; i < 100; i++) {  
    Thread method2Thread = new Thread() {  
        @Override  
        public void run() {  
            if (x > 0) {  
                x--;  
            }  
        }  
    };  
    Thread method1Thread = new Thread(new Runnable() {  
        @Override  
        public void run() {  
            if (x > 0) {  
                x--;  
            }  
        }  
    });  
    method2Thread.start();  
    method1Thread.start();  
    method2Thread.join(); // wait until completion  
    method1Thread.join();  
    if (x != 0) {  
        System.out.println(x);  
    }  
    x = 1;  
}
```

■ nicht nur ein theoretisches Beispiel!

```
for (int i = 0; i < 100; i++) {  
    Thread method2Thread = new Thread() {  
        @Override  
        public void run() {  
            if (x > 0) {  
                x--;  
            }  
        }  
    };  
    Thread method1Thread = new Thread(new Runnable() {  
        @Override  
        public void run() {  
            if (x > 0) {  
                x--;  
            }  
        }  
    });  
    method2Thread.start();  
    method1Thread.start();  
    method2Thread.join(); // wait until completion  
    method1Thread.join();  
    if (x != 0) {  
        System.out.println(x);  
    }  
    x = 1;  
}
```

<terminated> Test (2) [Java Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (30.06.2017, 15:50:29)

-1
-1

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
 - kritische Abschnitte schützen

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
 - kritische Abschnitte schützen
 - Wettlaufsituationen vermeiden

Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfinden

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
 - kritische Abschnitte schützen
 - Wettlaufsituationen vermeiden

Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfinden

Wettlaufsituation (race condition)

Verhalten des Programms hängt von der zeitlichen Abfolge der Operationen ab (Wann wird welcher Thread abgebrochen?)

- Ziel: Zugriff auf gemeinsam genutzte Daten synchronisieren
 - kritische Abschnitte schützen
 - Wettlaufsituationen vermeiden

Kritischer Abschnitt (critical section)

Codeabschnitt, wo Zugriffe auf gemeinsam genutzte Daten stattfinden

Wettlaufsituation (race condition)

Verhalten des Programms hängt von der zeitlichen Abfolge der Operationen ab (Wann wird welcher Thread abgebrochen?)

- Idee: Monitor einführen

- Idee: Monitor einführen

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert, als Argument bei `synchronized`

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert, als Argument bei `synchronized`
- Thread `t` kommt an eine mit `synchronized(Objekt){...}` markierte Stelle

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert, als Argument bei `synchronized`
- Thread `t` kommt an eine mit `synchronized(Objekt){...}` markierte Stelle
 - es wird geprüft, ob der Monitor Objekt gerade frei ist

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert, als Argument bei `synchronized`
- Thread `t` kommt an eine mit `synchronized(Object){...}` markierte Stelle
 - es wird geprüft, ob der Monitor Objekt gerade frei ist
 - ist der Monitor frei, kommt `t` in den kritischen Abschnitt und der Monitor ist besetzt, bis `t` den Abschnitt wieder verlässt

- Idee: Monitor einführen
- Bereich im Code markieren, den nur ein Thread gleichzeitig ausführen kann und dabei nicht unterbrochen werden kann
- Schlüsselwort in Java `synchronized`
- es wird immer an einem Objekt synchronisiert, als Argument bei `synchronized`
- Thread `t` kommt an eine mit `synchronized(Objekt){...}` markierte Stelle
 - es wird geprüft, ob der Monitor Objekt gerade frei ist
 - ist der Monitor frei, kommt `t` in den kritischen Abschnitt und der Monitor ist besetzt, bis `t` den Abschnitt wieder verlässt
 - ist der Monitor besetzt, wird `t` blockiert, bis der kritische Abschnitt frei ist


```
private Object o = new Object();
```

```
9      for (int i = 0; i < 100000; i++) {
10          Thread method2Thread = new Thread() {
11              @Override
12              public void run() {
13                  synchronized (o) {
14                      if (x > 0) {
15                          x--;
16                      }
17                  }
18              }
19          };
20          Thread method1Thread = new Thread(new Runnable() {
21              @Override
22              public void run() {
23                  synchronized (o) {
24                      if (x > 0) {
25                          x--;
26                      }
27                  }
28              }
29          });
30          method2Thread.start();
31          method1Thread.start();
32          method2Thread.join(); // wait until completion
33          method1Thread.join();
34          if (x != 0) {
35              System.out.println(x);
36          }
37          x = 1;
38      }
```

```
private Object o = new Object();
```

```
9      for (int i = 0; i < 100000; i++) {
10          Thread method2Thread = new Thread() {
11              @Override
12              public void run() {
13                  synchronized (o) {
14                      if (x > 0) {
15                          x--;
16                      }
17                  }
18              }
19          };
20          Thread method1Thread = new Thread(new Runnable() {
21              @Override
22              public void run() {
23                  synchronized (o) {
24                      if (x > 0) {
25                          x--;
26                      }
27                  }
28              }
29          });
30          method2Thread.start();
31          method1Thread.start();
32          method2Thread.join(); // wait until completion
33          method1Thread.join();
34          if (x != 0) {
35              System.out.println(x);
36          }
37          x = 1;
38      }
```

terminated> Test (2) [Java Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.

Parallelität - wait() und notify()

`synchronized` an Methoden = `synchronized(this){Methoden-Rumpf}`

Parallelität - wait() und notify()

synchronized an Methoden = synchronized(this){Methoden-Rumpf}

```
synchronized void produce(Product p) {  
    while(buffer.isFull()) {  
        //Tue nichts  
    }  
    buffer.add(p);  
}
```

Thread 1

```
synchronized void consume() {  
    while(buffer.isEmpty()) {  
        //Tue nichts  
    }  
    buffer.remove();  
}
```

Thread 2

Probleme

Parallelität - wait() und notify()

synchronized an Methoden = synchronized(this){Methoden-Rumpf}

```
synchronized void produce(Product p) {  
    while(buffer.isFull()) {  
        //Tue nichts  
    }  
    buffer.add(p);  
}  
  
synchronized void consume() {  
    while(buffer.isEmpty()) {  
        //Tue nichts  
    }  
    buffer.remove();  
}
```

Thread 1

Thread 2

Probleme

- “busy waiting“ verschwendet Rechenzeit

Parallelität - wait() und notify()

synchronized an Methoden = synchronized(this){Methoden-Rumpf}

```
synchronized void produce(Product p) {  
    while(buffer.isFull()) {  
        //Tue nichts  
    }  
    buffer.add(p);  
}
```

Thread 1

```
synchronized void consume() {  
    while(buffer.isEmpty()) {  
        //Tue nichts  
    }  
    buffer.remove();  
}
```

Thread 2

Probleme

- “busy waiting“ verschwendet Rechenzeit
- wartender Produzent blockiert Konsument, der dann nichts konsumieren kann

Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet

Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet
- dazu braucht man natürlich auch einen Mechanismus, der es erlaubt wartende Threads aufzuwecken

Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet
- dazu braucht man natürlich auch einen Mechanismus, der es erlaubt wartende Threads aufzuwecken
- in Java: wait() und notify() bzw. notifyAll()

Parallelität - wait() und notify()

- Idee: brauchen Mechanismus, der es erlaubt den Monitor freizugeben, während man auf etwas wartet
- dazu braucht man natürlich auch einen Mechanismus, der es erlaubt wartende Threads aufzuwecken
- in Java: wait() und notify() bzw. notifyAll()

```
synchronized void produce(Product p) {  
    while(buffer.isFull()) {  
        this.wait();  
    }  
    buffer.add(p);  
    this.notifyAll();  
}
```

Thread 1

```
synchronized void consume() {  
    while(buffer.isEmpty()) {  
        this.wait();  
    }  
    buffer.remove();  
    this.notifyAll();  
}
```

Thread 2

Parallelität - wait() und notify()

```
synchronized void produce(Product p) {  
    while(buffer.isFull()) {  
        this.wait();  
    }  
    buffer.add(p);  
    this.notifyAll();  
}
```

Thread 1

```
synchronized void consume() {  
    while(buffer.isEmpty()) {  
        this.wait();  
    }  
    buffer.remove();  
    this.notifyAll();  
}
```

Thread 2

- Kann man die while-Schleifen jetzt nicht durch eine if-Abfrage ersetzen?

Parallelität - wait() und notify()

```
synchronized void produce(Product p) {  
    while(buffer.isFull()) {  
        this.wait();  
    }  
    buffer.add(p);  
    this.notifyAll();  
}
```

Thread 1

```
synchronized void consume() {  
    while(buffer.isEmpty()) {  
        this.wait();  
    }  
    buffer.remove();  
    this.notifyAll();  
}
```

Thread 2

- Kann man die while-Schleifen jetzt nicht durch eine if-Abfrage ersetzen?
 - Nein, dann würde nach dem Aufwecken nicht nochmal geprüft werden, ob die Bedingung mittlerweile falsch ist.

Parallelität - Verklemmung (deadlock)

You release the lock first
Once I have finished
my task, you can continue.

Why should I?
You release the lock first
and wait until
I complete my task.



Parallelität - Verklemmung (deadlock)



- Thread A hält Monitor X und benötigt Monitor Y
- Thread B hält Monitor Y und benötigt Monitor X

Thread 1:

```
synchronized(Papier) {  
    synchronized(Stift) {  
        maleMandala();  
    }  
}
```

Thread 2:

```
synchronized(Stift) {  
    synchronized(Papier) {  
        maleMandala();  
    }  
}
```

klassischer Deadlock!

Parallelität - Verklemmung (deadlock)

Lösungsansatz: Monitore immer in gleicher Reihenfolge anfordern

Thread 1:

```
synchronized(Papier) {  
    synchronized(Stift) {  
        maleMandala();  
    }  
}
```

Thread 2:

```
synchronized(Papier) {  
    synchronized(Stift) {  
        maleMandala();  
    }  
}
```


Klausuraufgabe SS14

4. Übungsblatt - A3 mit Threads

Aufgabe 3 (GUI für Geometrify): 6,56 bzw. 11,25 von 10+7

```
public Test() {  
    // calling Swing methods from arbitrary threads may result in unexpected behaviour  
    // because most Swing Components are not thread safe!  
    new JFrame("HelloWorld").setVisible(true);  
  
    // instead use the swing event dispatch thread every time you paint, build,... Swing components  
    SwingUtilities.invokeLater(new Runnable() {  
        @Override  
        public void run() {  
            new JFrame("HelloWorld").setVisible(true);  
        }  
    });  
}
```

siehe auch: <https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>

4. Übungsblatt - A3 mit Threads

Aufgabe 3 (GUI für Geometrify): 6,56 bzw. 11,25 von 10+7

```
// will freeze the gui when the button is clicked
JButton heavy = new JButton("Freeze");
heavy.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // heavy work
    }
});

// will not freeze the gui
JButton light = new JButton("Don't freeze");
light.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        // use a new thread to handle heavy work
        Thread t = new Thread(new Runnable() {
            @Override
            public void run() {
                // heavy work
            }
        });
        // starts the thread, "main thread" will return immediately
        t.start();
    }
});
```

- Was verursacht was?
- Defekt, Irrtum, Versagen

- Was verursacht was?
- Defekt, Irrtum, Versagen
- “Testing shows the presence of bugs, not their absence.” (Edsger W. Dijkstra)

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
 - kontrollflussorientiert
 - datenflussorientiert
- black box testings

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
 - kontrollflussorientiert
 - datenflussorientiert
- black box testings
 - funktionale Tests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
 - kontrollflussorientiert
 - datenflussorientiert
- black box testings
 - funktionale Tests
 - Leistungstests

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
 - kontrollflussorientiert
 - datenflussorientiert
- black box testings
 - funktionale Tests
 - Leistungstests

Statische Verfahren

- Inspektion

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
 - kontrollflussorientiert
 - datenflussorientiert
- black box testings
 - funktionale Tests
 - Leistungstests

Statische Verfahren

- Inspektion
- statische Analyse mit Tools

Dynamische Verfahren

- Testfälle schreiben und ausführen (z.B. mit JUnit)
- white box testing
 - kontrollflussorientiert
 - datenflussorientiert
- black box testings
 - funktionale Tests
 - Leistungstests

Statische Verfahren

- Inspektion
- statische Analyse mit Tools
- Programm wird nicht ausgeführt!

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

- ① gegeben: zu testender Code

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

- 1 gegeben: zu testender Code
- 2 Code \implies Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

- ① gegeben: zu testender Code
- ② Code \implies Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen
- ③ Zwischensprache \implies Kontrollflussgraph

Kontrollflussorientiertes Testverfahren (KFO)

- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

- ① gegeben: zu testender Code
- ② Code \implies Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen
- ③ Zwischensprache \implies Kontrollflussgraph
- ④ am Kontrollflussgraphen Testfälle finden:

Kontrollflussorientiertes Testverfahren (KFO)

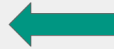
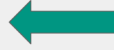
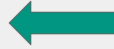
- Ziel: “sinnvolle“ Testfälle finden

Vorgehen:

- 1 gegeben: zu testender Code
- 2 Code \implies Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen
- 3 Zwischensprache \implies Kontrollflussgraph
- 4 am Kontrollflussgraphen Testfälle finden:
 - Anweisungsüberdeckung
 - Zweigüberdeckung
 - Pfadüberdeckung

■ Sprünge umwandeln

```
1  int a = 9;
2  System.out.println("Blahblah");
3  while(a == 9) {
4      int z = 0;
5      for(int i = 0; i <= 8; i++) {
6          z++;
7      }
8      int k = 0;
9      if(a == z + k) {
10         a = 8;
11     }
12 }
```



■ Sprünge umwandeln

```
1  int a = 9;
2  System.out.println("Blahblah");
3  if not (a == 9) goto 14;
4      int z = 0;
5      int i = 0;
6      if not (i <= 8) goto 10;
7          z++;
8          i++;
9      goto 6;
10     int k = 0;
11     if not (a == z + k) goto 13;
12         a = 8;
13 goto 3;
14
```

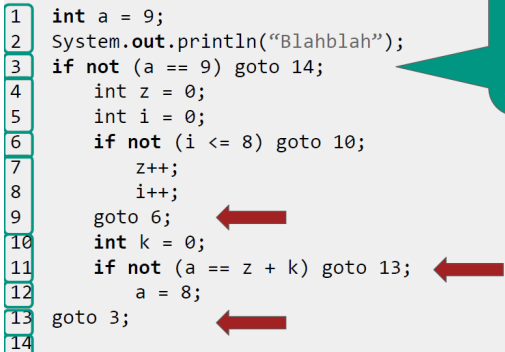
- Grundblöcke finden (Code bis goto ist ein Grundblock)

```
1  int a = 9;  
2  System.out.println("Blahblah");  
3  if not (a == 9) goto 14;  
4      int z = 0;  
5      int i = 0;  
6      if not (i <= 8) goto 10;  
7          z++;  
8          i++;  
9      goto 6;  
10     int k = 0;  
11     if not (a == z + k) goto 13;  
12         a = 8;  
13 goto 3;  
14
```

Grundblöcke dürfen
nur am Ende einen
Sprung (goto)
haben (müssen
aber nicht)

- Grundblöcke prüfen (goto dürfen nur an Anfang eines Grundblocks verweisen)

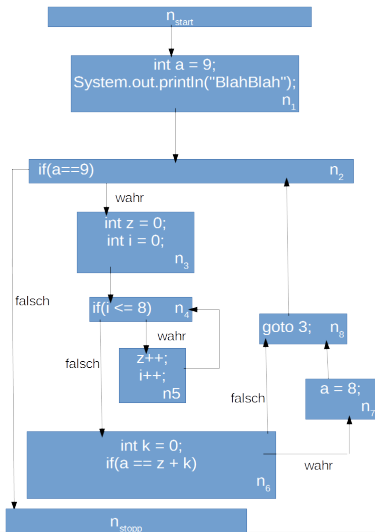
```
1  int a = 9;  
2  System.out.println("Blahblah");  
3  if not (a == 9) goto 14;  
4      int z = 0;  
5      int i = 0;  
6      if not (i <= 8) goto 10;  
7          z++;  
8          i++;  
9      goto 6;  
10     int k = 0;  
11     if not (a == z + k) goto 13;  
12         a = 8;  
13     goto 3;  
14
```



KFO: Zwischensprache nach Kontrollflussgraph

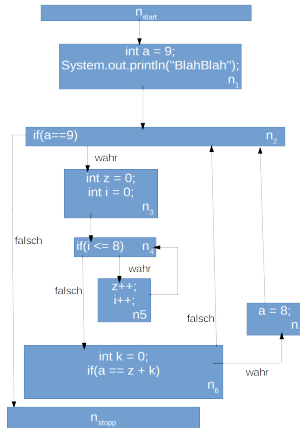
- Grundblöcke benennen
- Grundblöcke und Verzweigungen hinzeichnen
- Start- und Endzustand hinzufügen

KFO: Zwischensprache nach Kontrollflussgraph



KFO: Zwischensprache nach Kontrollflussgraph

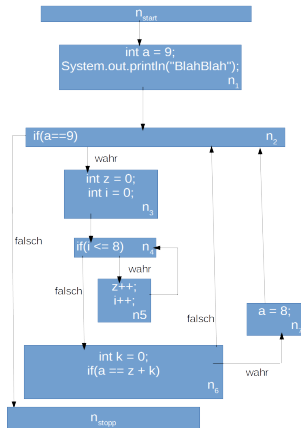
- goto-Knoten kann man auch weglassen



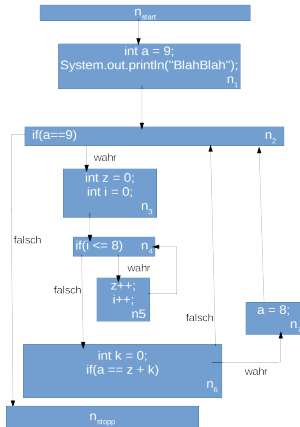
- Pfade finden, sodass jeder Grundblock traversiert wird

- Pfade finden, sodass jeder Grundblock traversiert wird
⇒ Entdeckung nicht erreichbarer Code-Abschnitte

- Pfade finden, sodass jeder Grundblock traversiert wird
⇒ Entdeckung nicht erreichbarer Code-Abschnitte
- aber: kein ausreichendes Testkriterium



■ Pfad für Anweisungsüberdeckung?

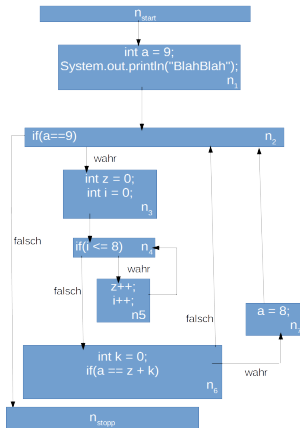


- Pfad für Anweisungsüberdeckung?
 $(n_{start}, n_1, n_2, n_3, n_4, n_5, n_4, n_6, n_7, n_2, n_{stopp})$

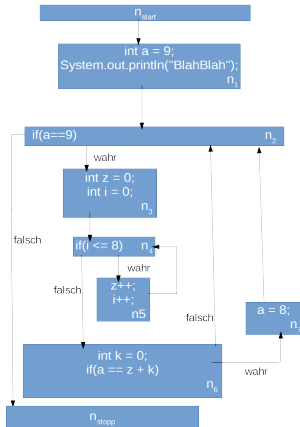
- Pfade finden, sodass jeder Zweig (=Kante) traversiert wird

- Pfade finden, sodass jeder Zweig (=Kante) traversiert wird
⇒ Entdeckung nicht erreichbarer Kanten

- Pfade finden, sodass jeder Zweig (=Kante) traversiert wird
⇒ Entdeckung nicht erreichbarer Kanten
- aber: Schleifen werden nicht ausreichend getestet



■ Pfad für Zweigüberdeckung?



■ Pfad für Zweigüberdeckung?

$(n_{start}, n_1, n_2, n_3, n_4, n_5, n_4, n_6, n_2, n_3, n_4, n_5, n_4, n_6, n_7, n_2, n_{stopp})$

- Finde **alle** vollständige, unterschiedlichen Pfade

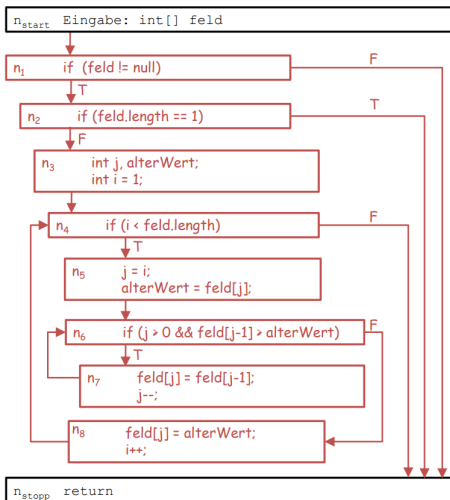
- Finde **alle** vollständige, unterschiedlichen Pfade
- vollständiger Pfad = Anfang bei n_{start} , Ende bei n_{stopp}

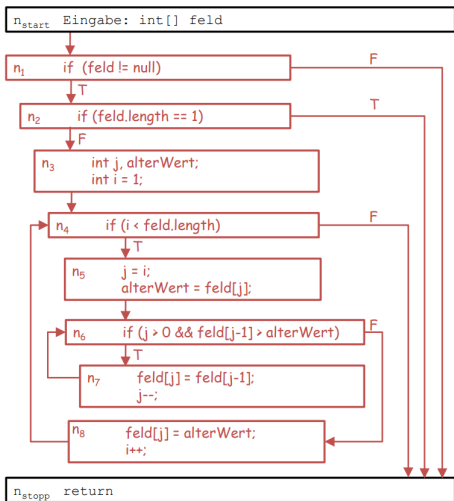
- Finde **alle** vollständige, unterschiedlichen Pfade
- vollständiger Pfad = Anfang bei n_{start} , Ende bei n_{stopp}
- nicht praktikabel, da
 - Schleifen die Anzahl der möglichen Pfade stark erhöhen

- Finde **alle** vollständige, unterschiedlichen Pfade
- vollständiger Pfad = Anfang bei n_{start} , Ende bei n_{stopp}
- nicht praktikabel, da
 - Schleifen die Anzahl der möglichen Pfade stark erhöhen
 - manche Pfade nicht ausführbar sind (sich gegenseitig ausschließende Bedingungen)

```
01 public void sortiere(int[] feld) {  
02     if (feld != null) {  
03         if (feld.length == 1) {  
04             return;  
05         } else {  
06             int j, alterWert;  
07             for (int i = 1; i < feld.length; i++) {  
08                 j = i;  
09                 alterWert = feld[i];  
10                 while (j > 0 && feld[j - 1] > alterWert) {  
11                     feld[j] = feld[j - 1];  
12                     j--;  
13                 }  
14                 feld[j] = alterWert;  
15             }  
16         }  
17     }  
18 }
```

Erstellen Sie den Kontrollflussgraphen und geben Sie einen Pfad an, der Anweisungsüberdeckung erzielt.





Pfad: $(n_{start}, n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8, n_4, n_{stopp})$

Aufgabe 1: Kontrollfluss-orientiertes Testen

- Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

Aufgabe 1: Kontrollfluss-orientiertes Testen

- Zwischensprache benutzen
- Definitionen der verschiedenen Abdeckungen anschauen

Aufgabe 2: Codeinspektion

- an das Format halten

Aufgabe 3: Parallelisierung von Geomtrify

- Berechnung der Samples parallelisieren
- Zahl der benutzten Threads abhängig machen von der Anzahl der Prozessor-Kerne

Aufgabe 3: Parallelisierung von Geomtrify

- Berechnung der Samples parallelisieren
- Zahl der benutzten Threads abhängig machen von der Anzahl der Prozessor-Kerne

Aufgabe 4: Alternative Parallelisierungsverfahren

- theoretische Überlegungen, was man sonst noch so parallelisieren könnte
- Sinnhaftigkeit, Aufwand, etc. prüfen

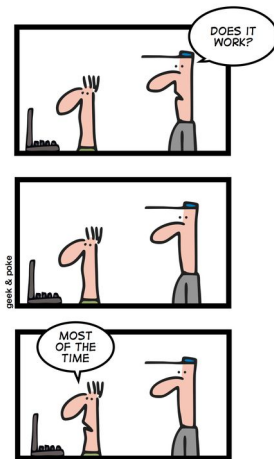
Aufgabe 5: Parallelisierungswettbewerb

- Aufgabe 3 verbessern und Laufzeit messen

Abgabe

- Deadline am 11.8. um 12:00
- Aufgabe 1,2,4 und Beschreibung, Laufzeitprofil von Aufgabe 5 handschriftlich
- auf jeden Fall abgeben, wenn ihr noch Punkte braucht

SIMPLY EXPLAINED



CONCURRENCY