

Softwaretechnik 1 - 5. Tutorium

Tutorium 17 Felix Bachmann | 09.07.2019

KIT - INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION (IPD)

- 1 Feedback
- Überblick
- Parallelität
- Testen
- Tipps

Allgemeines



Nächstes Mal ist schon letztes Tutorium

geplant: grobe Wiederholung + viele Klausuraufgaben

Überblick

Allgemeines



Nächstes Mal ist schon letztes Tutorium

- geplant: grobe Wiederholung + viele Klausuraufgaben
- ansonsten irgendwelche Wünsche?
 - etwas bestimmtes wiederholen?
 - nochmal irgendwas erklären?
 - Beispiel für XY?
 - falls euch noch was einfällt, schreibt mir eine Mail
 - ⇒ felix.bachmann@ewetel.net

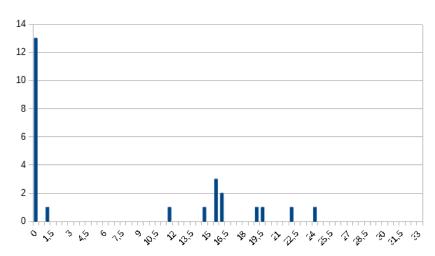
Zusatztermin



Zusatztermin Entwurfmuster

5. Übungsblatt Statistik





Tipps

Überblick



Allgemein

- (mal wieder...) CheckStyle und JavaDoc
- zu späte Abgabe = nur Korrektur, keine Punkte



Aufgabe 1: Architekturstile

Felix Bachmann - SWT1



Aufgabe 2: Entwurfsmuster CacheProvider

Felix Bachmann - SWT1

Testen



Aufgabe 2: Entwurfsmuster CacheProvider

Aufgabe 3: Entwurfsmuster impl. (Iterator, Besucher)

Überblick



Aufgabe 4: Code → Klassendiagramm



Aufgabe 4: Code → Klassendiagramm

Aufgabe 5: Zustandsmuster(Kaffeemaschine) impl.

- abstrakte Oberklasse/Interface Kaffeemaschine fehlte
 - für Hinzufügen neuer Kaffeemaschine sinnvoll



Aufgabe 6: git merge vs git rebase

Überblick

Jetzt letztes Übungsblatt..



Falls ihr noch Punkte braucht, gebt ab!

Überblick

Überblick



Wo sind wir?

Felix Bachmann - SWT1

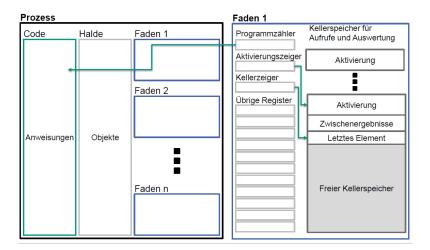
Überblick

Programm vs. Prozess vs. Thread



- Programm = ausführbare Datei
- Prozess = Programm in Ausführung
- Thread = Ausführungseinheit innerhalb eines Prozesses







Prozess = Programm in Ausführung

Überblick



- 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

Felix Bachmann - SWT1



- 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 globalen Variablen 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 globalen Variablen und demselben Code
 - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)

Felix Bachmann - SWT1

15/60



- 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
 - \implies alle Threads innerhalb eines Prozesses arbeiten mit denselben globalen Variablen und demselben Code
 - Threads haben eigene Stacks und Befehlszeiger (Programmzähler)
 - Threads haben eigene lokale Variablen und können beliebigen Code des Prozesses ausführen

15/60

Motivation



```
// 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
});
```

Motivation



```
// 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
    }
});
```

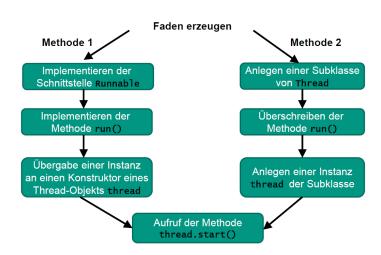
- "normales" sequentielles Programm = 1 Prozess mit 1 Thread
- paralleles Programm = 1 Prozess mit mehreren Threads



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

Felix Bachmann - SWT1







```
// 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

Synchronisation



- Problem: Zugriff auf globale Variablen passiert nicht "gleichzeitig", Unterbrechungen möglich
- Folge: ggf. falsche Ergebnisse

Faden 1

Faden 2

Nicht nur ein theoretisches Problem



```
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 Problem



```
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

Testen

Synchronisation



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
 - Ziel: Bereich im Code markieren,
 - den nur ein Thread gleichzeitig ausführen kann
 - in dem der Thread nicht unterbrochen werden kann



- Idee: Monitor einführen
 - Ziel: Bereich im Code markieren,
 - den nur ein Thread gleichzeitig ausführen kann
 - in dem der Thread nicht unterbrochen werden kann

Schlüsselwort in Java synchronized



- Idee: Monitor einführen
 - Ziel: Bereich im Code markieren,
 - den nur ein Thread gleichzeitig ausführen kann
 - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert
 - synchronized(Object o){/* zu schützender Code */ }



- Idee: Monitor einführen
 - Ziel: Bereich im Code markieren,
 - den nur ein Thread gleichzeitig ausführen kann
 - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert
 - synchronized(Object o){/* zu schützender Code */ }
- Thread t kommt an eine mit synchronized(Object o){...} markierte Stelle



- Idee: Monitor einführen
 - Ziel: Bereich im Code markieren,
 - den nur ein Thread gleichzeitig ausführen kann
 - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert
 - synchronized(Object o){/* zu schützender Code */ }
- Thread t kommt an eine mit synchronized(Object o){...} markierte Stelle
 - es wird geprüft, ob der Monitor gerade frei ist



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

23/60



- Idee: Monitor einführen
 - Ziel: Bereich im Code markieren,
 - den nur ein Thread gleichzeitig ausführen kann
 - in dem der Thread nicht unterbrochen werden kann
- Schlüsselwort in Java synchronized
- es wird immer an einem Objekt synchronisiert
 - synchronized(Object o){/* zu schützender Code */ }
- Thread t kommt an eine mit synchronized(Object o){...} markierte Stelle
 - es wird geprüft, ob der Monitor 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();
          for (int i = 0; i < 100000; i++) {
9
100
               Thread method2Thread = new Thread() {
                   @Override
12
                   public void run() {
13
                       synchronized (o) {
14
                           if (x > 0) {
15
                               x--;
16
18
19
200
               Thread method1Thread = new Thread(new Runnable() {
21⊕
                   @Override
                   public void run() {
23
                       synchronized (o) {
                           if (x > 0) {
24
25
                               x--:
26
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
               x = 1;
38
```

Überblick



```
private Object o = new Object();
          for (int i = 0; i < 100000; i++) {
9
100
               Thread method2Thread = new Thread() {
                   @Override
12
                   public void run() {
13
                       synchronized (o) {
14
                           if (x > 0) {
15
                               x--;
16
18
19
200
               Thread method1Thread = new Thread(new Runnable() {
21⊕
                   @Override
                   public void run() {
23
                       synchronized (o) {
                           if (x > 0) {
24
25
                               x--;
26
28
               });
29
30
               method2Thread.start():
31
              method1Thread.start();
32
               method2Thread.join(); // wait until completion
33
              method1Thread.join();
34
               if(x != 0) {
                   System.out.println(x);
               x = 1;
38
```

<terminated> Test (2) [Java Application] C:\Program Files\Java\jdk1.8.0_91\bin\javaw.exe (30.06.2017, 16:40:24)

Parallelität



- synchronized an Methoden
 - nicht-statische Methode = synchronized(this){Methoden-Rumpf}
 - statische Methode = synchronized(X.class){Methoden-Rumpf}



- synchronized an Methoden
 - nicht-statische Methode = synchronized(this) {Methoden-Rumpf}
 - statische Methode = synchronized(X.class) {Methoden-Rumpf}

Aber Achtung

- synchronized killt Parallelität
- immer nur ein Thread gleichzeitig im kritischen Abschnitt
- also nur dann benutzen wenn wirklich nötig
 - sonst wird alles wieder sequentiell
- außerdem kritische Abschnitte möglichst klein halten
 - möglichst wenig Arbeit im Abschnitt machen



```
synchronized void produce(Product p) {
   while(buffer.isFull()) {
                                                Thread 1
       //Tue nichts
   buffer.add(p);
synchronized void consume() {
   while(buffer.isEmpty()) {
                                                Thread 2
       //Tue nichts
   buffer.remove();
```



Probleme



Probleme

"busy waiting" verschwendet Rechenzeit



Probleme

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



- Idee: brauchen Mechanismus,
 - der es erlaubt den Monitor freizugeben, w\u00e4hrend man auf etwas wartet



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



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



- Idee: brauchen Mechanismus.
 - der es erlaubt den Monitor freizugeben, w\u00e4hrend man auf etwas wartet
 - der es erlaubt wartende Threads aufzuwecken
- in Java: wait() und notify() bzw. notifyAll()

```
synchronized void produce(Product p) {
   while(buffer.isFull()) {
                                                 Thread 1
       this.wait():
   buffer.add(p):
   this.notifyAll();
synchronized void consume() {
                                                 Thread 2
   while(buffer.isEmpty()) {
       this.wait():
   buffer.remove();
   this.notifvAll():
```



```
synchronized void produce(Product p) {
   while(buffer.isFull()) {
                                                Thread 1
       this.wait();
   buffer.add(p);
   this.notifyAll();
synchronized void consume() {
                                                Thread 2
   while(buffer.isEmpty()) {
       this.wait();
   buffer.remove();
   this.notifyAll();
```

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



```
synchronized void produce(Product p) {
   while(buffer.isFull()) {
                                                Thread 1
       this.wait();
   buffer.add(p);
   this.notifyAll();
svnchronized void consume() {
                                                Thread 2
   while(buffer.isEmpty()) {
       this.wait():
   buffer.remove():
   this.notifyAll();
```

- 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.



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.







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





- 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();
    }
}
```

klassicher Deadlock!



Lösungsansatz: Monitore immer in gleicher Reihenfolge anfordern

Thread 1:

```
synchronized(Papier) {
   synchronized(Stift) {
       maleMandala();
Thread 2:
synchronized(Papier) {
   synchronized(Stift) {
       maleMandala();
```

Felix Bachmann - SWT1



- Schlüsselwort volatile an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
 - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen



- Schlüsselwort volatile an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
 - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren



- Schlüsselwort volatile an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
 - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren

Löst es das folgende Problem?

i++ ausgeführt von 2 Threads. volatile i ist initial 0. Ist i nach der Ausführung immer 1?



- Schlüsselwort volatile an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
 - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren

Löst es das folgende Problem?

i++ ausgeführt von 2 Threads. volatile i ist initial 0. Ist i nach der Ausführung immer 1?

Beispiel-Code



- Schlüsselwort volatile an Variablen
- Zugriff auf Variablen immer direkt über den Hauptspeicher
 - Caches werden umgangen
- sorgt dafür, dass alle Threads den gleichen Wert sehen
- Compiler darf weniger optimieren

Löst es das folgende Problem?

i++ ausgeführt von 2 Threads. volatile i ist initial 0. Ist i nach der Ausführung immer 1?

Beispiel-Code

- Nein!
 - volatile kümmert sich nur darum, dass alle Threads die gleiche Sicht auf den Speicher haben
 - kein Einfluss auf Ausführung
 - kein Schutz vor race conditions

32/60

Klausuraufgabe SS14

Überblick

09.07.2019

Testen

Parallelität üben



https:
//deadlockempire.github.io/

Überblick

Swing Dispatch Thread



```
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

Parallelität in GUI nutzen



```
// 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();
});
```

Wie benutzt man nun Parallelität?



- großes Problem
- ??? } Parallelität?
- Profit

Felix Bachmann - SWT1

Wie benutzt man nun Parallelität?



- großes Problem in Teilprobleme aufteilen
- Teilprobleme durch verschiedene Threads parallel lösen lassen
- Teil-Lösungen zu Gesamt-Lösung zusammensetzen
 - auf critical sections/ race conditions achten
- Profit = Laufzeit-Reduktion (falls alles richtig gemacht)



Ziel: "sinnvolle" Testfälle finden

Vorgehen:

gegeben: zu testender Code

Felix Bachmann - SWT1



Ziel: "sinnvolle" Testfälle finden

Vorgehen:

- gegeben: zu testender Code
- ② Code ⇒ Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen



Ziel: "sinnvolle" Testfälle finden

Vorgehen:

- gegeben: zu testender Code
- ② Code ⇒ Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen



Ziel: "sinnvolle" Testfälle finden

Vorgehen:

- gegeben: zu testender Code
- ② Code ⇒ Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen
- am Kontrollflussgraphen Testfälle finden



Ziel: "sinnvolle" Testfälle finden

Vorgehen:

- gegeben: zu testender Code
- ② Code ⇒ Zwischensprache
 - Sprünge umwandeln
 - Grundblöcke finden
 - Grundblöcke prüfen
- Zwischensprache Kontrollflussgraph
- am Kontrollflussgraphen Testfälle finden
 - Anweisungsüberdeckung
 - Zweigüberdeckung
 - Pfadüberdeckung

KFO: Code → Zwischensprache



Sprünge umwandeln

```
int a = 9;
System.out.println("blubb");
while (a == 9) {
  int z = 0;
  for (int i = 0; i <= b; i++) {
    n++;
  }
  z++;
  if (a == z) {
    a = 8:
```

KFO: Code → Zwischensprache



- Sprünge umwandeln (while, for, if, x ? a : b,...)
 - wie geht das für while, if, for? (Tafel)
 - Ziel: nur if, not, goto verwenden. Kein {...}

```
int a = 9;
System.out.println("blubb");
while (a == 9) \{ <- hier \}
  int z = 0:
  for (int i = 0; i \le b; i++) { <- hier
    n++:
  z++:
  if (a == z) \{ <- hier \}
    a = 8;
```

Sprünge wurden umgewandelt



```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;
04 \text{ int } z = 0;
05 \text{ int i = 0};
06 if not (i <= b) goto 10;
07 n++;
08 i++:
09 goto 06;
10 z++;
11 if not (a == z) goto 13;
12 a = 8;
13 goto 03;
14
```

KFO: Code → **Zwischensprache**



- nächster Schritt: Grundblöcke finden
- Code bis goto ist ein Grundblock

```
03 if not (a == 9) goto 14;
04 \text{ int } z = 0;

    haben erste Grundblöcke

05 \text{ int } i = 0;
06 if not (i <= b) goto 10;
                                                gefunden
07 n++;
08 i++;
09 goto 06;
10 z++;
11 if not (a == z) goto 13;
12 a = 8;
13 goto 03;
14
```

Testen

09.07.2019

44/60

01 int a = 9;

Feedback

Felix Bachmann - SWT1

02 System.out.println("blubb");

Überblick

```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;
```

- 04 int z = 0; 05 int i = 0; 06 if not (i <= b) goto 10;
- 20 11 1100 (1) 2, 8000 10
- 08 i++; 09 goto 06;

07 n++:

- 10 z++; 11 if not (a == z) goto 13;
- 12 a = 8; 13 goto 03;

- haben erste Grundblöcke gefunden
- jetzt: Grundblöcke prüfen
- goto d\u00fcrfen nur auf Start eines Blocks verweisen
- daher: Blöcke ggf. noch weiter unterteilen

```
01 int a = 9:
02 System.out.println("blubb");
03 if not (a == 9) goto 14;
04 \text{ int } z = 0:
05 \text{ int } i = 0:

    haben erste Grundblöcke

06 if not (i <= b) goto 10;
                                              gefunden
                                            jetzt: Grundblöcke prüfen
07 n++:
08 i++:
                                            goto dürfen nur auf Start eines
09 goto 06; <-- kaputt!
                                               Blocks verweisen
                                            daher: Blöcke agf. noch weiter
10 z++;
                                              unterteilen
11 if not (a == z) goto 13; <-- kaputt!
12 a = 8:
13 goto 03; <-- kaputt!
```

14

```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;
04 \text{ int } z = 0;
05 \text{ int } i = 0;
06 if not (i <= b) goto 10;
07 n++:
08 i++;
09 goto 06;
10 z++;
11 if not (a == z) goto 13;
12 a = 8;
13 goto 03;
14
Feedback
                Überblick
                             Parallelität
                                                                Testen
                                                                                          Tipps
                                                                Felix Bachmann - SWT1
                                                                         09.07.2019
                                                                                       46/60
```

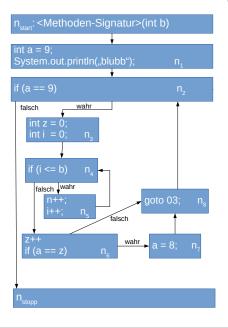
```
03 if not (a == 9) goto 14;
04 \text{ int } z = 0:
05 \text{ int } i = 0;
                                          jetzt sind wir fertig mit den Grundblöcken
06 if not (i <= b) goto 10;
07 n++:
08 i++;
09 goto 06;
10 z++;
11 if not (a == z) goto 13;
12 a = 8:
13 goto 03;
14
Feedback
                Überblick
                             Parallelität
                                                                Testen
                                                                                          Tipps
                                                                Felix Bachmann - SWT1
                                                                         09.07.2019
                                                                                       46/60
```

01 int a = 9;

02 System.out.println("blubb");

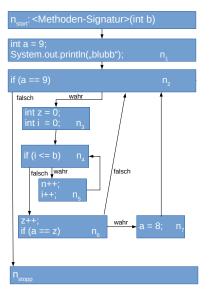
```
01 int a = 9;
02 System.out.println("blubb");
03 if not (a == 9) goto 14;
04 \text{ int } z = 0:
05 \text{ int } i = 0;
06 if not (i <= b) goto 10;
07 n++:
08 i++:
09 goto 06;
10 z++;
11 if not (a == z) goto 13;
12 a = 8:
13 goto 03;
14
Feedback
                Überblick
```

- jetzt sind wir fertig mit den Grundblöcken
- als n\u00e4chstes: Graphen hinzeichnen
- dazu Start- und Endblock hinzufügen
- Blöcke benennen $(n_{start}, n_1, \dots, n_x, n_{stopp})$
- Knoten und Kanten hinzeichnen



Überblick

goto-Knoten kann man auch weglassen



KFO: Anweisungsüberdeckung



wähle Testfälle so, dass jeder Grundblock traversiert wird

KFO: Anweisungsüberdeckung



- wähle Testfälle so, dass jeder Grundblock traversiert wird
 - ⇒ Entdeckung nicht erreichbarer Code-Abschnitte

KFO: Anweisungsüberdeckung



- wähle Testfälle so, dass jeder Grundblock traversiert wird
 - ⇒ Entdeckung nicht erreichbarer Code-Abschnitte
- aber: kein ausreichendes Testkriterium
 - Schleifen einmal durchlaufen reicht schon

KFO: Zweigüberdeckung



wähle Testfälle so, dass jeder Zweig (=Kante) traversiert wird

KFO: Zweigüberdeckung



wähle Testfälle so, dass jeder Zweig (=Kante) traversiert wird
 Entdeckung nicht erreichbarer Kanten

KFO: Zweigüberdeckung



- wähle Testfälle so, dass jeder Zweig (=Kante) traversiert wird
 - ⇒ Entdeckung nicht erreichbarer Kanten
- wie bei Anweisungsüberdeckung
 - Schleifen werden ggf. nicht ausreichend getestet

KFO: Pfadüberdeckung



- Finde alle vollständigen, unterschiedlichen Pfade
 - wähle Testfälle so, dass alle gefundenen Pfade durchlaufen werden
- vollständiger Pfad = Anfang bei n_{start}, Ende bei n_{stopp}

KFO: Pfadüberdeckung



- Finde alle vollständigen, unterschiedlichen Pfade
 - wähle Testfälle so, dass alle gefundenen Pfade durchlaufen werden
- vollständiger Pfad = Anfang bei n_{start}, Ende bei n_{stopp}
- nicht praktikabel, da
 - Schleifen die Anzahl der möglichen Pfade stark erhöhen
 - sogar "unendlich viele", wenn Iterationen von Eingabe abhängig

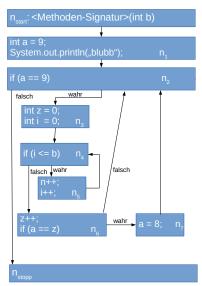
KFO: Pfadüberdeckung



- Finde alle vollständigen, unterschiedlichen Pfade
 - wähle Testfälle so, dass alle gefundenen Pfade durchlaufen werden
- vollständiger Pfad = Anfang bei n_{start} , Ende bei n_{stopp}
- nicht praktikabel, da
 - Schleifen die Anzahl der möglichen Pfade stark erhöhen
 - sogar "unendlich viele", wenn Iterationen von Eingabe abhängig
 - manche Pfade nicht ausführbar sind
 - z.B. sich gegenseitig ausschließende Bedingungen

"Sinnvolle" Testfälle

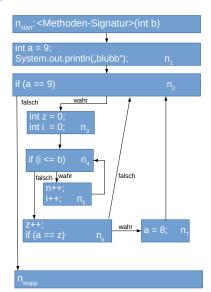




- was ist nun unsere Testfallmenge?
 - = welche b testen?
- für Anweisungsüberdeckung?
- für Zweigüberdeckung?

"Sinnvolle" Testfälle





- b=0 erfüllt bereits beides
- Pfad $(n_{start}, n_1, n_2, n_3, n_4, n_5, n_4, n_6, n_2, \dots, n_6, n_7, n_2, n_{stopp})$
- i.A. aber mehrere Testfälle nötig

Klausuraufgabe SS11



```
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
           i = i:
09
           alterWert = feld[i];
           while (i > 0 \&\& feld[i - 1] > alterWert) {
10
11
             feld[i] = feld[i - 1]:
12
13
14
           feld[i] = alterWert;
15
16
17
18
```

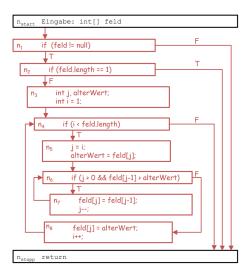
Erstellen Sie den Kontrollflussgraphen und geben Sie einen Pfad an, der

Anweisungsüberdeckung erzielt.

Felix Bachmann -	SWT1
000000000	
Feedback	Ü

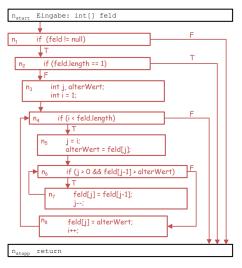
MuLö KFO





MuLö KFO





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

55/60

Kurzschlussauswertung



Achtung, für ÜBs/Klausur

- MuLö ist alt, nicht ganz korrekt
- Kurzschlussauswertung (&& bzw. | |) auflösen (in zwei if)



Aufgabe 1: Parallelisierung von HDrize

Berechnung der HDR-Bilder parallelisieren



Aufgabe 1: Parallelisierung von HDrize

- Berechnung der HDR-Bilder parallelisieren
- Thread-Pool spart euch die manuelle Verwaltung der Threads
 - ExecutorService es =
 Executors.newFixedThreadPool(amountOfThreads);
 - es.execute(myRunnable); (beliebig viele)
 - es.shutdown();



Aufgabe 1: Parallelisierung von HDrize

- Berechnung der HDR-Bilder parallelisieren
- Thread-Pool spart euch die manuelle Verwaltung der Threads
 - ExecutorService es =
 Executors.newFixedThreadPool(amountOfThreads);
 - es.execute(myRunnable); (beliebig viele)
 - es.shutdown();
- Tests schreiben



Aufgabe 1: Parallelisierung von HDrize

- Berechnung der HDR-Bilder parallelisieren
- Thread-Pool spart euch die manuelle Verwaltung der Threads
 - ExecutorService es =
 Executors.newFixedThreadPool(amountOfThreads);
 - es.execute(myRunnable); (beliebig viele)
 - es.shutdown();
- Tests schreiben

Aufgabe 2: Parallelisierungswettbewerb

- Aufgabe 1 verbessern und Laufzeit messen
- Teile handschriftlich abgeben
 - Laufzeitprofil und Erklärung des Ansatzes
 - Laufzeitprofil kann man auch ohne weitere Optimierung erstellen ;)

Feedback

Überblick

Parallelität

arallelitat

Testen

Tipps •ooo



Aufgabe 3: Kontrollfluss-orientiertes Testen

- zur Sicherheit Zwischensprache benutzen
 - aber auch ok ohne
 - (hilft aber keine Fehler zu machen)
- Achtung: & vs. &&
- ansonsten "Schema F"



Aufgabe 3: Kontrollfluss-orientiertes Testen

- zur Sicherheit Zwischensprache benutzen
 - aber auch ok ohne
 - (hilft aber keine Fehler zu machen)
- Achtung: & vs. &&
- ansonsten "Schema F"

Aufgabe 4: Funktionales Testen

- Äquivalenzklassen für Parameter finden
- auch ungültige und fehlerhafte Eingaben beachten



Aufgabe 3: Kontrollfluss-orientiertes Testen

- zur Sicherheit Zwischensprache benutzen
 - aber auch ok ohne
 - (hilft aber keine Fehler zu machen)
- Achtung: & vs. &&
- ansonsten "Schema F"

Aufgabe 4: Funktionales Testen

- Äquivalenzklassen für Parameter finden
- auch ungültige und fehlerhafte Eingaben beachten

Aufgabe 5: Codeinspektion

mit Checkliste an eventuellen Problemen Code durchgehen

Denkt dran!



Abgabe

- Deadline am 17.7, um 12:00
- A3-5 und Teile von A2 handschriftlich
- auf jeden Fall abgeben, wenn ihr noch Punkte braucht
 - aus jeder Aufgabe gute Punkte rauszuholen

Bis dann! (dann := 23.07.19)



SIMPLY EXPLAINED

