

# Proseminar: Paralleles und verteiltes Rechnen

Thema: Parallele Programmierung in Java

28.07.2003

Gunnar Spieß

Diese Ausarbeitung bezieht sich auf den am 06.06.2003 gehaltenen Vortrag über parallele Programmierung in Java im Proseminar paralleles und verteiltes Rechnen. Diese Proseminar wurde vom Wilhelm-Schickard-Institut für Informatik - Abteilung Technische Informatik – angeboten.

Der Ausarbeitung lag hauptsächlich das Buch „Concurrent Programming in Java“ von Doug Lea zugrunde, welches vom Addison-Wesley Verlag veröffentlicht wurde.

Es wurde teilweise auch das Buch „Java in a Nutshell“ von David Flanagan (Deutsche Übersetzung von Konstantin Agouros) vom O'Reilly Verlag verwendet.

## **1. Einleitung / Motivation:**

In der heutigen Zeit treten immer wieder neue und vor allem größere, komplexere Probleme auf.

Die bestehenden Programme bzw. die Algorithmen, die in diesen Programmen vorkommen, lassen sich nur bedingt optimieren.

Somit wäre schnellere Hardware nötig, um Probleme schneller als bisher lösen zu können.

Neue Hardwarebauteile sind meist sehr teuer.

Beispielsweise kostet ein neuer Pentium IV 3-GHz-Prozessor heute ca. 420 € ein Pentium IV 1,6-GHz-Prozessor hingegen nur ca. 150 € (Preise vom K&M Elektronik).

Es wäre also billiger Probleme mit 2 Prozessoren zu lösen, anstatt mit einem. Man kann

Probleme also schneller lösen, indem man Berechnungen auf mehrere Prozessoren verteilt.

Es werden außerdem oft Programme benötigt, die mehrere Dinge parallel erledigen sollen.

Ein Beispiel hierfür wäre ein WWW-Browser. Er kann z.B. gleichzeitig eine neue Seite laden, einen Audio-Clip abspielen und die bereits empfangenen Daten anzeigen.

Dieses Problem kann durch parallele Programmierung (z.B. in Java) gelöst werden.

Die Ausarbeitung ist im Wesentlichen in vier Teile gegliedert:

1. **Einleitung/Motivation:** Hier wird kurz dargestellt, warum überhaupt parallel programmiert wird bzw. warum Parallelität benötigt wird.
2. **Grundlagen:** In diesem Teil werden zuerst einige Grundlagen zu Java bzw. zur Parallelität in Java erläutert. Es wird außerdem beschrieben, was „Threads“ sind und wie diese erzeugt/gestartet werden. Im Anschluss wird noch erklärt, was „Scheduling“ bedeutet.
3. **Paralleles Programmieren:** Im diesem Teil wird beschrieben, wie sich Parallelität in Java realisieren lässt und welche Probleme bei dieser Art von Programmierung auftreten können. Weiterhin werden verschiedene Lösungsstrategien für diese genannten Probleme dargestellt.
4. **Fazit/Zusammenfassung:** Im letzten Teil werden die auftretenden Probleme bzw. Vorteile von Parallelität noch einmal kurz aufgegriffen bzw. erwähnt.

## 2. Grundlagen:

Zuerst werden einige allgemeine Begriffe zu Java erläutert.

Java ist eine objektorientierte Programmiersprache. Hierbei treten Begriffe, wie z.B. Methoden oder Objekte auf. Eine **Methode** ist ein objektorientierter Begriff für eine Funktion oder Prozedur.

Ein **Objekt** ist definiert durch seinen Zustand bzw. seine Attribute und seine jeweiligen Methoden.

In Java wird vom Compiler ein sogenannter **Bytecode** erzeugt. Er unterscheidet sich von anderen Compilern, die Maschinencode erzeugen, welcher nur auf dieser speziellen Plattform ausgeführt werden kann. Dieser Bytecode ist somit **plattformunabhängig**.

Für die Ausführung eines Java-Programms wird ein Interpreter und ein Laufzeitsystem benötigt. Diese Funktion übernimmt die sogenannte **Java-Virtual-Machine (JVM)**.

Die Parallelität wird durch eine **nebenläufige** Ausführung verschiedener Prozesse erreicht. Es werden also zu bestimmten Zeitpunkten im Programm neue Prozesse gestartet. Dies kann beliebig oft passieren.

Diese Parallelität bzw. Nebenläufigkeit wird in Java durch sogenannte **Threads** erreicht.

### Was sind überhaupt Threads?

Threads sind parallel ablaufende Steuerflüsse innerhalb eines Programms.

Jedes Programm in Java hat zumindest einen Thread. Mit dem Befehl „new“ können neue Threads erzeugt werden. Mit dem Aufruf von „start“ wird die „run“-Methode des jeweiligen Threads ausgeführt. Somit können zu jedem Zeitpunkt beliebig viele Threads erzeugt werden. Diese erzeugten und lauffähigen Threads werden nun der Virtual Machine zur Ausführung zugeteilt.

Als Ergebnis haben wir eine nebenläufige Ausführung von mindestens zwei Threads, nämlich das Programm und die darin erzeugten Threads.

Nach Beendigung eines solchen Threads laufen die anderen Threads ungestört weiter. Zur Veranschaulichung vergleiche Abbildung 1.



Abbildung 1: Threads

## Scheduler:

Die Threads werden in Java durch den sogenannten Scheduler verwaltet.

Der Scheduler simuliert die Nebenläufigkeit, falls weniger Prozessoren als Threads zur Verfügung stehen. Jeder Thread bekommt bei seiner Erzeugung eine Priorität zugeordnet.

Falls mehrere Threads mit gleicher Priorität zur Ausführung bereit sind, wird vom Scheduler zufällig einer ausgewählt und dann der Virtual Machine übergeben. Ansonsten wird normalerweise der Thread mit der höchsten Priorität ausgewählt. Es gibt aber hierfür keine Garantie.

Threads mit langen Berechnungen bekommen eher niedrige Priorität, da sie sonst den Rechner zu lange blockieren würden. Das Scheduling wird automatisch nach jeder Zustandsänderung (siehe Abschnitt 3.1.1.) eines Threads durchgeführt.

Da der Scheduler (bei gleicher Priorität) immer zufällig Threads auswählt, müssen verschiedene Programmausführungen des gleichen Programms nicht immer identisch sein. Es ist also nicht möglich für den Programmierer vorauszusagen, welcher Thread als nächstes ausgeführt wird. Somit ist der gesamte Programmablauf schwierig vorzuberechnen. Dieses Problem wird auch **Nichtdeterminismus** (engl.: nondeterminism) genannt.



Nachfolgend wird dieses Schaubild erläutert.

Zuallererst wird der Thread mit **new** erzeugt. Danach wird mit dem Aufruf von **start** seine **run**-Methode ausgeführt. Dann befindet sich der Thread in einem rechenwilligen oder ausführbaren Zustand. Nun kann der Thread vom Scheduler ausgewählt und damit ausgeführt werden.

Es gibt die Möglichkeit mit der **yield**-Methode vom ausgeführten Zustand in den rechenwilligen Zustand überzugehen. Es wird dann geschaut, ob ein weiterer rechenwilliger Thread zur Verfügung steht. Ist dies der Fall, dann wird dieser Thread ausgeführt. Ansonsten wird der angehaltene Thread wieder fortgesetzt.

Ein Thread kann auch mit **sleep** in den schlafenden Zustand überführt werden. Der Thread wird für eine bestimmte Zeit „schlafen gelegt“ und nach Ablauf dieser Zeit automatisch fortgesetzt.

Ein Thread kann außerdem mit **join** in einen blockierten Zustand übergehen. Es wird dann solange gewartet bis der Ziel - Thread fertig abgearbeitet ist. Join wird zum Beispiel folgendermaßen aufgerufen:

```
Thread2.join();
```

Hierbei ist Thread2 der Zielthread.

Danach wird der blockierte Thread fortgesetzt. Falls der Thread, auf den gewartet wird, schon beendet ist, wird der blockierte Thread sofort fortgesetzt.

Java stellt darüber hinaus eine **isAlive**-Methode bereit, um zu testen, ob ein Thread noch im ausführbaren/ausgeführten Zustand befindet.

Wurde ein Thread bis zu seinem Ende ausgeführt (letzter Befehl in run-Methode wurde abgearbeitet), dann wird dieser Thread beendet.

Bis zur Java Version 1.2 gab es noch 3 andere Methoden:

Es gab die **suspend**-Methode, die einen Thread anhält bis **resume** aufgerufen wird.

Es war außerdem **stop** vorhanden. Diese Methode beendete einen Thread sofort. Es spielte keine Rolle, ob der Thread seine Operation bereits beendet hatte.

## Monitoring:

Es sind noch 4 weitere Methoden vorhanden, um Threads zu kontrollieren, die speziell bei Synchronisationen verwendet werden.

Zum einen gibt es die **wait**-Methode. Sie unterbricht die Ausführung und stellt den Thread in eine Warteschlange. Es wird außerdem die Synchronisationssperre für das Zielobjekt aufgehoben.

Dann gibt es noch die **notify**-Methode, welche einen zufälligen ausgewählten Thread aus der Warteschlange entfernt und die Synchronisationssperre für das Zielobjekt wieder aufbaut. Dieser Thread wird dann an der Stelle, wo wait aufgerufen wurde, wieder fortgesetzt.

Außerdem gibt es **notifyAll**. Diese Methode macht dasselbe wie notify, nur gilt dies für alle Threads in der Warteschlange.

Falls die wait-Methode mit einer Zeitangabe (in ms) aufgerufen wurde, wird nach Ablauf dieser Zeit automatisch notify aufgerufen.

Diese wait und notify-Methoden sind sogenannte **Monitoring**-Methoden. Sie sind dazu da, dass man die Zustände eines Threads besser überwachen kann. Außerdem dienen sie auch zur besseren Ablaufkontrolle / Ablaufsteuerung eines nebenläufigen Programms.

Wenn man z.B. möchte, dass ein nebenläufiges Programm zuerst Punkt1 und dann Punkt2 erledigt, kann man dies mit den oben genannten Methoden steuern.

Man könnte Punkt2 mit wait() in den blockierenden Zustand setzen. Nach der Abarbeitung von Punkt1 könnte dann von Punkt1 aus notify() aufgerufen werden. Dann wird Punkt2, an der Stelle an der es blockiert wurde, fortgesetzt.

Ohne diese Monitoring-Methoden gäbe es in Java keine effiziente nebenläufige Programmierung. Es sind einfach Methoden nötig, um die Zustände anderer Thread abfragen oder auch steuern zu können.

Ohne diese Hilfsmittel käme beim nebenläufigen Programmieren oft nur Datenmüll oder Ausnahmefehler zustande, wie z.B. Zugriffe auf nicht mehr vorhandene Teile eines Arrays (siehe Programm 2).

### **3.1.2. Die zwei Eigenschaften eines parallelen Programms:**

In Java gibt es folgende zwei Eigenschaften, die ein paralleles Programm beinhalten sollten. Zum einen gibt es die **Sicherheit**. Das bedeutet, dass nichts Schlimmes im Programm passiert. Sicherheitsfehler führen zu unbeabsichtigtem Verhalten während der Laufzeit. Zum anderen gibt es **Liveness**. Liveness bedeutet, dass überhaupt etwas passiert. Liveness-Fehler führen zu keinem Verhalten. Alles kann anhalten.

Man sollte darauf achten, dass ein Programm eher weniger macht oder auch langsamer ist, aber dafür sicher ist.

Es ist sehr schwierig eine gute **Balance** zwischen Sicherheit und Liveness zu finden.

## **3.2. Sicherheit:**

### **3.2.1 Sichere Objekte:**

Sichere Objekte sind vergleichbar zu Typen-Sicherheit. Es kommt nicht vor, dass z.B. Bits, die ein float oder ein double repräsentieren als Objektreferenz aufgefasst werden.

Sicherheit heißt auch, dass Objekt in einem konsistenten („korrekten“) Zustand gehalten werden.

Sicherheit kann leider nicht vom Compiler überprüft werden. Man muss sich ganz auf die Kompetenz des Programmierers verlassen.

In Java gibt es 4 Strategien, wie Sicherheit realisiert werden kann:

- Unveränderlichkeit
- Synchronisation: Hierbei wird dynamisch garantiert, dass exklusiver Zugriff vorliegt

- Einkapselung: Hierbei wird strukturell garantiert, dass exklusiver Zugriff vorliegt
- verwalteter Besitz (managed ownership)

Die erste Strategie wird folgendermaßen realisiert:

Die Zustände eines Objektes werden einfach mehr nicht verändert. Falls man einen neuen Objektzustand braucht, wird stattdessen ein neues Objekt mit dem neuen Zustand erzeugt.

Im folgenden wird die zweite Strategie erläutert:

### **3.2.2 Synchronisation**

Falls keine Synchronisation verwendet werden würde, gäbe es Ergebnisse, wie z.B.: „HelGoodlobye“

Dieses Ergebnis könnte das Resultat des nachstehenden Programms sein:

```
public class t extends Thread{
    private int was;
    public t(int w){
        was=w;
        start();
    }
    public void run(){
        if (was==0){
            System.out.print(„Hel“);
            System.out.print(„lo“);
        };
        else {System.out.print(„Good“);
            System.out.print(„bye“);
        };
    }
    public static void main(String[] args){
        t t1=new t(0);
        t t2=new t(1);
    }
}
```

Programm 1

Quelle: Concurrent Programming in Java

In der main-Methode werden 2 Threads erzeugt. Dem einen wird eine 0 und dem anderen eine 1 mit übergeben. In einem Thread wird die integer-Variable auf 0 und im anderen auf 1 gesetzt.

In der run-Methode wird überprüft, ob was = 0 ist. Falls dem so ist, wird zuerst „Hal“ und dann „lo“ ausgegeben. Im anderen Fall zuerst „Good“, dann „bye“.

Nun könnte es aber vorkommen, dass der Scheduler den ersten Thread nach der Ausgabe von „Hal“ unterbricht und den zweiten ausführt und „Good“ ausgibt. Danach könnte der erste wieder an der Reihe sein und „lo“ ausgeben und später der zweite „bye“.

Dies wäre ein Beispiel, wie Java ohne Synchronisation Unsinn ausgibt.  
Dies kann vermieden werden, indem man die run-Methode synchronisiert.

Bei Zustandsänderungen von Objekten ist Synchronisation notwendig.  
Sonst könnten folgende zwei Konflikte auftreten:

### **3.2.2.1 read/write-Konflikt:**

Es könnte vorkommen, dass auf illegale, flüchtige Zustände zugegriffen wird, die nur während des Updatens erscheinen.

Ein Beispiel hierfür wären diese 2 Methoden:

```
public synchronized String at(int i){
    if(i>=0&& i<size) return data[i]}
}
public synchronized void removeLast(){
    if(size!=0) data[size-1]=null;
    --size;
}
```

Programm 2

Quelle: Concurrent Programming in Java

Die „at“-Methode gibt ein Element *i* in einem Array zurück. Die Methode `removeLast` entfernt das letzte Element aus einem Array.

Wenn man das Wort „synchronized“ weglassen würde, könnte es vorkommen, dass während das letzte Element im Array mit `removeLast` entfernt wird, auf das letzte Element zugegriffen wird. Denn die Variable `size` wird erst nach dem Entfernen des letzten Elements dekrementiert.

Das es aber nicht mehr vorhanden ist, führt das zu einem Fehler.

### **3.2.2.2. write/write-Konflikt:**

Es könnte eine inkonsistente Zuweisung von parallel laufenden Update-Methoden geben. Zum Beispiel wenn zwei Methoden die Operation `i=i+1` ausführen. Im sequentiellen Fall wird *i* um 2 erhöht. Hingegen kann es im parallelen Fall vorkommen, dass gleichzeitig beide Methoden den Wert *i* auslesen, beide den Wert inkrementieren und beide den inkrementierten Wert in *i* schreiben. Somit wäre *i* nur um 1 erhöht.

### **3.2.2.3. Partielle Synchronisation:**

Es reicht oft aus in Methoden nur gewisse Teile zu synchronisieren. Es ist möglich ,dass parallel zu unsynchronisierten Methoden synchronisierte Methoden laufen können. Somit könnte man sehr viel **Zeit einsparen**. Man muss aber darauf achten, dass Lese und Schreibzugriffe auf Objekte synchronisiert sind, um oben genannte Konflikte zu vermeiden.

Ein Beispiel für ein solches Programm wäre:

```
public synchronized void setValue(double v){
    value=v;
}
public boolean Abfrage(double x){ //search for x
    synchronized(this){
        if(value==x) return true;
    }
    System.out.println(„Wert nicht gefunden!“);
    //.....
}
```

Porgramm 3

Quelle: Concurrent Programming in Java

Das this-Argument ist eine Referenz auf das jeweilige Objekt, das synchronisiert wird. In diesem Fall auf value. Die partielle Synchronisation findet nur in dem markierten Bereich statt, also nur in dem Bereich von „synchronized(this)“.

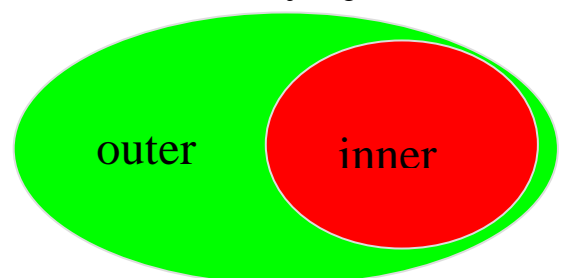
Hier kann mit der Methode setValue value auf einen bestimmten Wert gesetzt werden. Mit der Methode Abfrage kann überprüft werden, ob value mit x übereinstimmt. In Abfrage muss nur vermieden werden, dass der Zugriff auf value gleichzeitig mit dem Aufruf der Methode setValue vorkommt. Der Rest der Methode kann unsynchronisiert weiterlaufen, da keine Zugriffe auf value mehr folgen.

Im folgenden wird die dritte Strategie erläutert:

### **3.2.3. contained objects:**

Bei dieser Strategie werden Objekte in andere Objekte eingebettet. Es verweisen hier Referenzen vom äußeren Objekt outer auf das innere Objekt inner. Das äußere Objekt gibt die Referenzen nie als Argumente oder Rückgabewerte weiter. Somit wird durch volle Synchronisation des äußeren Objekts auch das innere komplett synchronisiert. Es können also Lese- und Schreibzugriffe bedenkenlos im Programm vorkommen.

Es werden somit Klassen konstruiert, in denen nur ein Thread Zugriff auf ein Objekt hat.



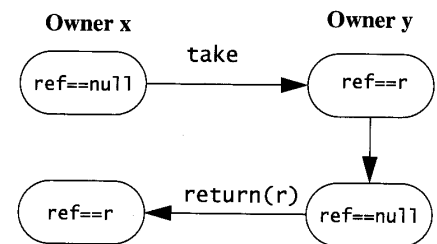
→Somit liegt eine Garantie von **exklusivem** Zugriff vor

### 3.2.4 verwalteter Besitz (managed ownership)

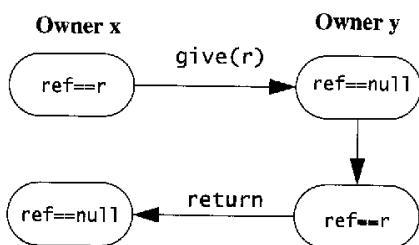
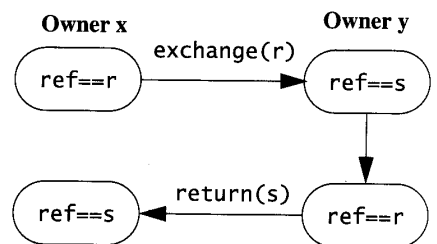
Es besteht auch die Möglichkeit durch verwalteten Besitz exklusiven Zugriff zu garantieren. Die Strategie hierfür ist, dass eine Ressource immer nur einen Besitzer zu einem bestimmten Zeitpunkt haben kann. Der Besitz kann im Laufe der Zeit aber durchaus wechseln.

Im folgenden werden einige Möglichkeiten des Besitzerwechsels aufgezeigt:

Eine Möglichkeit ist es den Besitz zu **übernehmen**. Besitzer x sagt Besitzer y, dass er r haben möchte. Also setzt y seinen Besitz auf null und übergibt r an Besitzer x. Dieser kann nun seinen Besitz auf r setzen.



Eine weitere Möglichkeit ist es den Besitz **auszutauschen**. Besitzer x übergibt r an y. Dieser setzt dann seinen Besitz auf r und übergibt s an Besitzer x, welcher dann seinen Besitz auf s setzen kann.



Eine dritte Möglichkeit ist es den Besitz von Besitzer x zu Besitzer y zu **übergeben**. Besitzer x übergibt Ressource r an Besitzer y. Besitzer y setzt dann seinen Besitz auf r und Besitzer x setzt ihn auf null.

Quelle: Concurrent Programming in Java

### 3.3. Liveness-Failures:

Liveness-Failures sind schwierig zu identifizieren und zu vermeiden. Diese Fehler treten in 4 verschiedenen Zusammenhängen auf:

- Contention (Streit): Ein Thread läuft nicht, ist aber ausführbar, weil ein anderer Thread die CPU Ressourcen übernommen hat.
- Dormancy: Ein nicht-runnable Thread wird nicht zu runnable, weil nie ein resume oder nach wait nie ein notify aufgerufen wird. Der Thread bleibt also im schlafenden oder blockierten Zustand.

- Premature Termination (verfrühte Beendigung) : Ein Thread wird mit Stopp früher beendet als geplant. Er kann also eine gedachte Operation nicht zuende ausführen.
- Deadlock (Verklemmung): siehe nächster Abschnitt

### **3.3.1. Deadlock / Verklemmung:**

Einer der zentralen Liveness-Fehlern ist Deadlock. Wenn mehrere kooperierende Objekte synchronisiert werden, kann es vorkommen, dass Deadlock auftritt.

Ein Beispiel hierfür wäre nachfolgendes Programm:

```
class Document{
    Document otherPart; //verweist auf anderes Dokument
    synchronized void print(){
        System.out.println(„first line“);
        //.....
        System.out.println(„last line“);
    }
    synchronized void printAll(){
        otherPart_.print();
        print();
    }
}
```

Programm 4

Quelle: Concurrent Programming in Java

Hier haben wir eine Dokumenten-Klasse. Die Klasse besitzt eine Referenz auf ein anderes Dokument (other part). Sie besitzt außerdem eine Methode, die das eigene Dokument von der ersten bis zur letzten Zeile ausgibt. Es gibt auch noch eine zweite Methode, die zuerst das andere Dokument ausdrückt und danach das eigene Dokument.

Hier kann es folgendermaßen zum Deadlock kommen. Nehmen wir an wir hätten zwei Threads. Der eine mit einem Dokument Brief1 und der andere mit einem Dokument Brief2.

Dann könnte es nach dem Schema, das auf der nächsten Seite aufgeführt und erläutert wird, zur Verklemmung kommen.

## Thread1

Brief1.printAll  
(Brief1 ist nun gesperrt)

Brief1.otherPart.print  
(wartet bis Brief2 freigegeben wird)

## Thread2

Brief2.printAll  
(Brief2 ist nun gesperrt)

Brief2.otherPart.print  
(wartet bis Brief1 freigegeben wird)

In Thread1 wird printAll aufgerufen und somit Brief1 für andere Zugriffe gesperrt. Nun wird vom Scheduler Thread2 ausgewählt. In Thread2 wird zuerst auch printAll aufgerufen und somit auch Brief2 für andere Zugriffe gesperrt. In printAll wird ja zu Beginn versucht das andere Dokument auszugeben. Also versucht Thread1 Brief2 auszudrucken. Da aber eine Synchronisationssperre von Thread2 vorliegt, muss Thread1 warten bis Brief2 freigegeben wird. Genauso versucht Thread2 Brief1 auszudrucken. Aber auch hier liegt eine Sperre auf Brief1 vor. Thread2 wartet also bis Brief1 freigegeben wird. Die Dokumente werden aber erst nach kompletter Abarbeitung der Methode wieder freigegeben. Hier wird es nie zur Abarbeitung einer Methode kommen, also liegt nun eine Verklemmung vor. In diesem Fall verursacht die Aufrufreihenfolge den Deadlock. Es kann durchaus vorkommen, dass dieses Programm normal abläuft und den gewünschten Effekt erzielt. Diese Methoden sind zwar sicher. Es kann aber vorkommen, dass es zu keiner Ausgabe kommt. Das Programm kann sich also verklemmen.

### 3.4. zwei verschiedene Annäherungsmöglichkeiten:

Es gibt 2 verschiedene Möglichkeiten, um ein paralleles Programm zu entwerfen:

- **Top-down-Strategie** (safety first): Hierbei werden zuerst Methoden mit vollständiger Synchronisation entworfen. Es werden dann nach und nach unnötige Synchronisationen gestrichen, um Effizienz und Liveness zu gewinnen
- **Bottom-up-Strategie** (liveness first): Es werden Methoden ohne Synchronisation entworfen. Danach wird an benötigten Stellen Synchronisation hinzugefügt, z.B. durch Unterklassen oder durch partielle Synchronisation

Das Ziel beider Methoden ist es die größtmögliche Liveness bzw. Effizienz zu erzielen ohne dabei Sicherheit opfern zu müssen.

## 4. Fazit/Zusammenfassung

Bei paralleler bzw. nebenläufiger Programmierung muss man darauf achten, dass ein Programm den **Sicherheits**aspekt beinhaltet.

Man sollte also sein Programm so implementieren, dass auf ein Objekt niemals doppelt zugegriffen werden kann. Dies könnte zu den erläuterten Konflikten führen, oder es könnten inkonsistente Zustände auftreten.

Man sollte außerdem darauf achten, dass ein Programm genügend **Liveness** mit sich führt. Sonst könnte das Programm ausgebremst oder sogar zum größten Problem, nämlich dem Deadlock führen und somit nicht das gewünschte Ergebnis liefern.

Bei parallelen Programmen in Java stellt sich außerdem das Problem des **Nichtdeterminismus**. Das heißt, dass verschiedene Ausführungen des selben Programms nicht immer identisch sein müssen. Der Programmierer kann den Ablauf nicht vorausberechnen.

Um die Parallelität in einem Programm verwalten zu können, ist zusätzlicher Rechenaufwand nötig. Dies hat zur Folge, dass oft sequentielle Programme schneller als parallele Programme sind, die dieselbe Funktion haben.

Ein weiteres Problem ist die **Komplexität**. Ein solches Programm ist weniger übersichtlich. Es werden außerdem zusätzliche Sprachkonstrukte gebraucht, um solche Programme implementieren zu können.

Es ist aber eine gute Kontrolle der Programme möglich. Dies kann mit Methoden, wie z.B. wait, notify, usw. geschehen.

Es ist darüber hinaus möglich eine gute **Verfügbarkeit** zu gewährleisten. Man kann in Programmen für jede Anfrage von Clients einen neuen Thread erzeugen und somit unnötige Engpässe vermeiden.

Dies ist eines der Hauptziele, welches man durch nebenläufiges Programmieren erreichen will (falls ein Prozessor verwendet wird).

Falls sogar mehrere Prozessoren zur Verfügung stehen, kann durch Parallelität außerdem die Rechenzeit verkürzt werden. Man verteilt die einzelnen Prozesse auf mehrere Prozessoren.

### Verwendete Literatur:

Die Beispielprogramme dieser Ausarbeitung wurden größtenteils dem Buch „Concurrent Programming in Java“ von Doug Lea (Addison-Wesley Verlag) entnommen.

Einigen Textpassagen lag auch das Buch „Java in a Nutshell“ von David Flanagan – deutsche Übersetzung von Konstantin Agouros- (O'Reilly Verlag) zugrunde.

Das Schaubild der Threadzustände wurde dem Internet entnommen. Es war aber nicht mehr recherchierbar von welcher URL dieses Schaubild stammt

Die restlichen Diagramme bzw. Schaubilder wurden selber entworfen.