Lebendigkeit

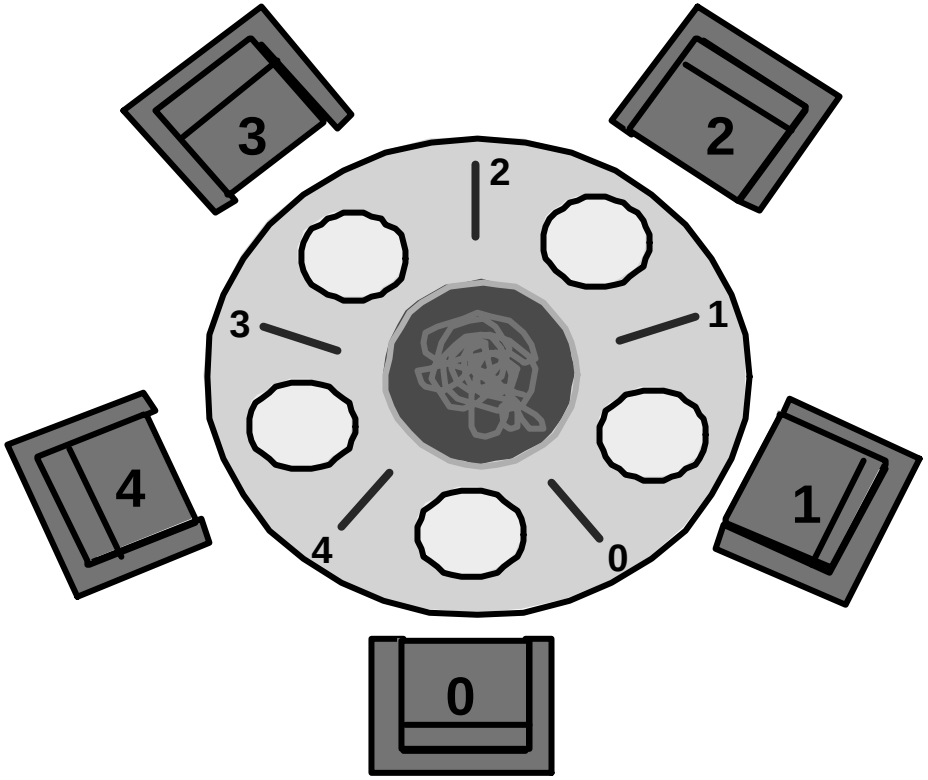
Nach D. Lea, Kap. 3

Sicherheits - und Lebendigkeitseigenschaften

- Sicherheit:
  - eine Eigenschaft, die zusichert, daß nichts „Böses“ passieren wird
  - „Böses“ meist „fehlerhaftes Verhalten“
  - falls eine Sicherheitseigenschaft verletzt wird, so ist der Effekt nicht mehr reparierbar
  - „things just start going wrong“
- Lebendigkeit:
  - eine Eigenschaft, die zusichert, daß etwas „Gutes“ irgendwann eintreten kann
  - umgekehrte Sichtweise, alternative Formulierung: es ist nícht möglich, eine Situation herbeizuführen, in der das „Gute“ für alle Zeit ausgeschlossen ist.
  - „things just stop running“
- viele Eigenschaften sind Mischformen dieser Extrema
- formale Behandlung: B. Alpern, F. Schneider: Defining Liveness, Information Processing Letters 21 (1985), 181-185

Ein Klassiker: Dining Philosophers Problem

5 Philosophen verbringen ihr Leben mit Essen und Denken in stetigem Wechsel. Am gemeinsamen Tisch finden sich jedoch nur 5 Gabeln, von denen jeder Philosoph jeweils die Gabel zu seiner rechten und zu seiner linken Seite zum Essen verwendet.



Gesucht: Algorithmus mit 5 Threads (Philosophen) und 5 geteilten Objekten (Gabeln) der unbegrenzten Ablauf ohne Deadlocks und Starvation erlaubt.

Dining Philosophers

- Deadlock = Verklemmung, partieller oder totaler Stillstand
- Deadlock möglich, wenn jeder Philosoph Gabeln nacheinander aufnimmt
- Lösungen ohne Deadlock:
  - Anzahl Philosophen am Tisch wird auf 4 begrenzt
  - One-Shot-Allocation: jeder Philosoph wartet bis beide Gabeln frei sind und nimmt simultan beide Gabeln
  - Definition einer linearen Ordnung: Gabeln werden durchnumeriert und Philosophen greifen auf Gabeln nur in aufsteigender Numerierung zu

  Warum funktioniert das ?

- Starvation möglich, wenn Auflösung der Wartesituation nicht geregelt ist

- Interessante Programmieraufgabe zur Anwendung von Threads und Monitoren (synchronized, wait, notifyAll)

Deadlockbehandlung: Querbezug Betriebssysteme

- 4 Bedingungen für Deadlock
  - wechselseitiger Ausschluß
  - halten und warten
  - keine Unterbrechung, keine erzwungene Ressourcenfreigabe
  - zyklische Wartesituation
- bekannte Lösungen
  - verhindern
    - simultane Allokation von Ressourcen (One-shot-allocation)
    - lineare Ordung in der Ressourcenbelegung beachten
    - bei Wartesituation oder auf Anforderung müssen alle Ressourcen freigegeben werden
  - vermeiden
    - nur bei ausreichender Zahl von Ressourcen Prozeß/Thread starten
    - nur Wechsel zwischen sicheren Zuständen, Bankier Algorithmus
  - entdecken und beseitigen
    - zyklische Wartesituation in Menge blockierter Threads ermitteln

Diskussion

- Strategien, die eher innerhalb der VM realisiert werden könnten
  - entdecken und beseitigen
  - vermeiden

weil diese Strategien einen aktiven Ressourcenverwalter erfordern.

- Innerhalb eines Softwaredesigns realisierbar
  - bei Wartesituation müssen Ressourcen freigegeben werden

  Dies entspricht der Idee aus : wait & notify innerhalb eines synchronized Blocks

  Problematisch ist die Freigabe von allen(!) allokierten Ressourcen, dh bei geschachtelten Aufrufen von synchronisierten Methoden!
  - Definition einer linearen Ordnung auf den Ressourcen und Allokation nur gemäß dieser Reihenfolge.

  Diese Restriktion kann in einigen Anwendungsfällen durchgehalten werden!
    - Falls nicht ? Unterschiedliche Reihenfolgen liefern bereits ersten Hinweis auf möglichen Deadlock
    - weitere Alternative: Ressourcen gruppieren und zumindest Reihenfolge auf Gruppen beachten.

Beispielcode für Philosophenproblem mit linearer Ordnung

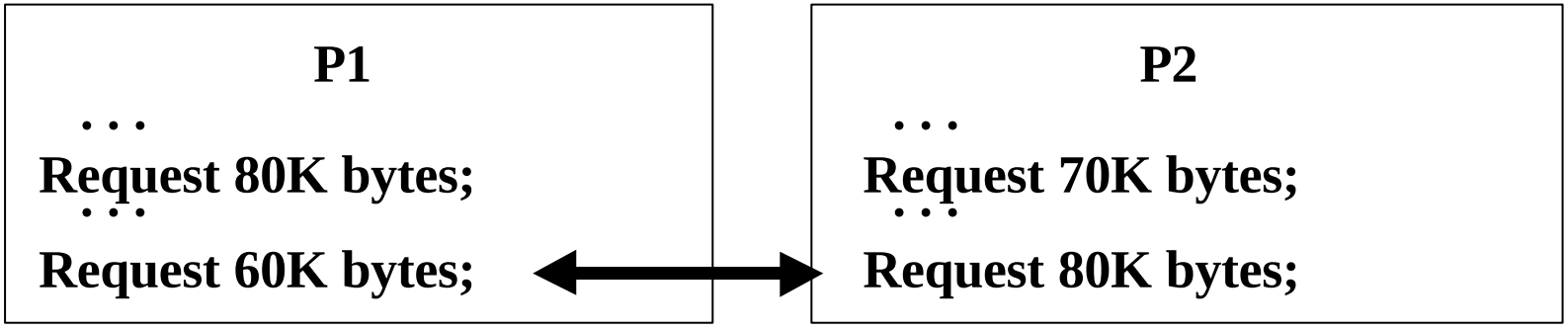
```
Start Methode des Applets
public void start() {
      for (int i =0; i<5; ++i)
          fork[i] = new Fork(display,i);
      for (int i =0; i < 5; ++i) {
          if (i < 4)
             phil[i] = makePhilosopher(this,i,
                    fork[(i-1+5)% 5],
                    fork[i]);
         else
            phil[i] = makePhilosopher(this,i,
                    fork[i],
                    fork[(i-1+5)% 5);
         phil[i].start();
      }}
```

Idee:
Code unterscheidet rechte und linke Gabel nur durch Variable l und r, die Nutzung ist identisch.
Daher nur bei Aufruf des Konstruktors für den letzten Philosophen die Fork Objekte für rechts und links vertauschen.

Einfacher Sonderfall !!!

Weitere Varianten von Deadlocks

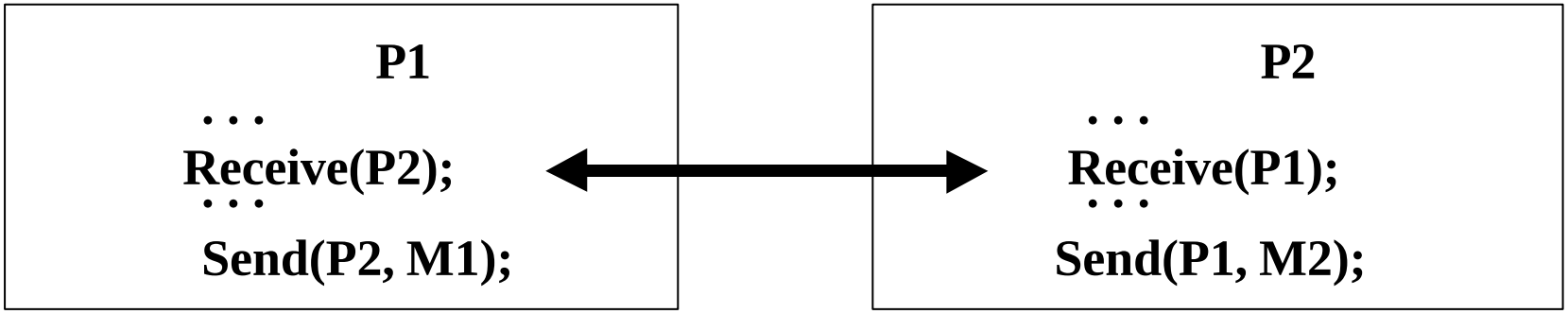
- Inkrementelle Anforderungen von einer Ressource
- Beispiel Speicher: bei 200 kb freiem Speicher folgende Anfragen
- Deadlock entsteht falls beide Prozesse vor zweiter Anforderung stehen



- Speicheranforderung ist wg virtuellem Speicher natürlich unkritisch
- in Java Thread Kontext: inkrementelle Belegung von exklusiv zugreifbaren Elementen aus einer Menge von Objekten, wobei die Menge als Ressource betrachtet wird

Weitere Varianten von Deadlocks
Verbrauchsressourcen

- Verbrauchsressourcen werden durch einen Prozeß erzeugt, durch einen anderen verbraucht
- z.B. Interrupts, Signale, Nachrichten, Daten in I/O Puffern
- Deadlock kann entstehen, wenn Receive blockierend ist
- Kombination erforderlicher Ereignisse kann selten auftreten
- es liegt ein Mismatch aus send und receive Operationen vor



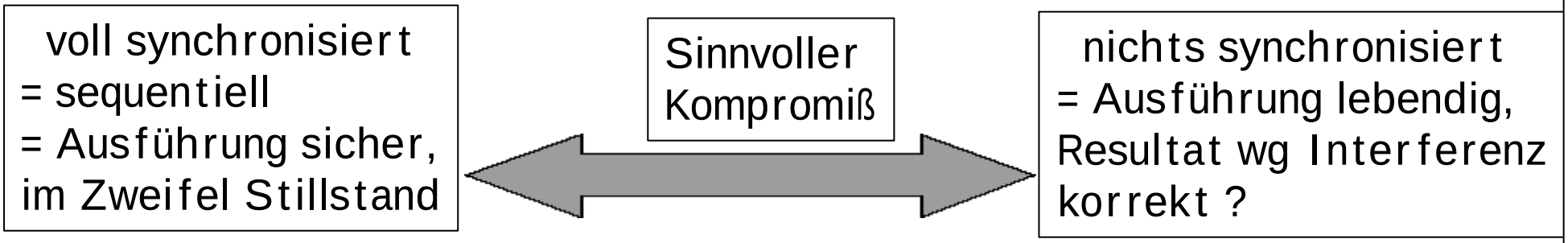
Verletzungen von Lebendigkeitsanforderungen

- Bisher: Deadlock
  - n > 1 Threads warten im Kreis auf Freigabe von Locks (synchronized) oder auf das Eintreffen von Nachrichten
- Contention, Starvation
  - Thread ist runnable, wird aber nicht bearbeitet, weil die CPU von anderen Threads oder anderen Prozessen permanent belegt wird.
- Dormancy
  - Thread ist non-runnable und erlangt Status runnable nicht mehr.

  Mögliche Ursachen:
    - suspend ohne resume,
    - wait ohne notify/notifyAll
    - joint ohne Terminierung des entsprechenden Threads
- Premature Termination
  - Thread ist unvorhergesehenerweise terminiert:
    - Exception, Error
    - durch anderen Thread stimuliert: stop

Lebendigkeit, Sichtweise nach D. Lea

- Lebendigkeit als Extremfall von Effizienzbetrachtungen
- Extrema:
  - volle Synchronisation -> Sicherheit
  - keine Synchronisation -> Lebendigkeit



- Lebendigkeit durch Verringerung der Synchronisationsanforderungen
  - auch Effizienzsteigerung, weil Synchronisation Performanz senkt
    - Belegen und Freigeben von Locks erfordert Ausführung von zusätzlichem Code
    - Blockieren und Freigeben von Threads kostet Verwaltungsaufwand im Betriebssystem oder im Runtime Environment

Beispiel: Klasse mit Selbstreferenz

- Zyklische Referenz zwischen Objekten kann Deadlock bewirken wenn Synchronisation im Spiel ist und Aufrufreihenfolge passt
- Beispiel:
  - Klasse für Dokumente, die neben eigenen Inhalten, Referenzen auf weitere Dokumente beinhalten
  - sei eine Funktion printAll, die alle Dokumente ausgibt, synchronisiert
  - sei eine Funktion print, die das eigene Teildokument ausgibt, synchronisiert
  - dann kann bei 2 Threads, die beide das gleiche Dokument ausgeben, ein Deadlock auftreten,
    - wenn das Dokument aus mindestens 2 Teilen A und B besteht
    - wenn ein Thread A.printAll() aufruft und der andere Thread B.printall()
    - die beiden Aufrufe allokieren jeweils den Lock von A und B und versperren sich dann damit gegenseitig den Zugang zu A.print() bzw B.print()
- andere Semantik: doppelt verkettete Liste mit 2 Elementen und printAll()

Zwischenfazit

- Synchronized muß mit viel Bedacht genutzt werden !!!

Ausgleichsmechanismen, Entwurfsmechanismen

- Top-Down (Sicherheit zuerst)
  - beginne mit voller Synchronisation
  - reduziere Synchronisation soweit wie möglich, um Lebendigkeit und Effizienz herzustellen
  - kann als Vorgehen zur Optimierung betrachtet werden
  - paßt z.B. zum Einsatz von „guarded methods“, Methoden mit Wächtern, die stets synchronisiert sind (später mehr ...)
- Bottom-Up (Lebendigkeit zuerst)
  - beginne Design ohne Rücksicht auf Synchronisation
  - füge Synchronisation durch Komposition, Subclassing und Schichten ein
  - erlaubt den Einsatz spezieller Synchronisationstechniken unter Berücksichtigung der Kooperation von Objekten, weil ein komplettes Design vorliegt
  - paßt zu Synchronisationstechniken, die von einem Schichtenmodell ausgehen (später mehr ...)

Lebendigkeit und Wiederverwendbarkeit

- Wiederverwendung von SW Komponenten in unbekannten Umgebungen ?
  - Erforderlich für Nützlichkeit der Software
  - Heikel für Lebendigkeit, deutlich schwieriger als im sequentiellen Fall
- Policies
  - Grundsatzvereinbarungen, die in einem Framework oder Paket beachtet werden müssen
  - in verteilten, offenen Umgebungen zum Umgang mit Verletzungen von Lebendigkeitsannahmen üblich

Lebendigkeit und Wiederverwendbarkeit

- Kriterien für die Wiederverwendbarkeit von Entities
  - wenig gekoppelt: Entity ohne zahllose weitere Spezialklassen nutzbar
  - modifizierbar: Policies + Implementierung dynamisch veränderbar
  - nützlich: leisten Dienste, die tatsächlich gebraucht werden
  - interoperabel: Funktionalität und Implementierung getrennt, Implementierung austauschbar
  - erweiterbar: durch Klassen und Interfaces so strukturiert, daß Variationen und Erweiterungen vorgenommen werden können
  - regulär: basieren auf einfachen Policies für Kopplung und Kommunikation
  - lokal: keine unkontrollierte Interaktion mit anderen sichtbaren Objekten
  - robust: sicheres Funktionieren auch im Mißbrauchsfall
  - dokumentiert: nicht kodierte Policies, Constraints, Limitations müssen textuell dokumentiert sein

Instance Variable Analysis

- Vorüberlegung: wechselseitiger Ausschluß schützt letztlich Zugriff auf Speicherbereiche, die Daten speichern
- Grundidee: bei Vorliegen von Constraints, Restriktionen bzgl der Werte von Variablen und der Methoden, die darauf zugreifen, kann in einigen Fällen auf den wechselseitigen Ausschluß verzichtet werden
- Randbedingungen:
  - die Zuweisungsoperation bei einfachen Datentypen in Java ist atomar
  - dies gilt jedoch bereits für double oder long Werte nicht mehr !!!
- Analyse muß die Semantik der Klasse mitberücksichtigen

Accessors - Zugriffsfunktionen

- Eine Accessor Funktion liefert den Wert einer Variable aus einer Instanz eines Objektes oder einen daraus abgeleiteten Wert (durch eine zustandslose Funktion)
- Accessors müssen nicht immer wechselseitigen Ausschluß garantieren
- Voraussetzung: keine Lese/Schreib Konflikte
- notwendige Bedingungen
  - die Variable kann keine illegalen Übergangswerte einnehmen. Sie beinhaltet auch innerhalb der Operationen jeder Methode stets einen sinnvollen, nach außen exportierbaren Wert. Dies schließt stets Variable aus, deren Wert Constraints mit anderen Variablenwerten unterliegt, und falls Updates auf diese Menge von Variablen unabhängig erfolgen können.
  - Die Zuweisung von Werten muss atomar erfolgen.

Updates - Aktualisierungsfunktionen

- Schreiboperationen auf Variable einer Instanz sollten normalerweise im wechselseitigen Ausschluß erfolgen (Write/Write Konflikte)
- Möglicher Sonderfall:
  - keine illegalen Übergangswerte möglich
  - Zuweisung ist atomare Operation
  - Werte bei beliebigem Interleaving von Lese/Schreibmethoden sind stets zulässig und sinnvoll (wenn auch nicht aktuell)
  - Variable haben nur Wertebereich aus spezieller Wertemenge

Reduktion von Synchronisationsanforderungen

- Falls Instance Variable Analysis zeigt, dass keine Synchronisation erforderlich ist:
  - „synchronized“ an Methoden weglassen, da dann durch Subclassing bei Wiederverwendung bei Bedarf Synchronisation wiederherstellbar ist
  - Variable deshalb nicht als public definieren sondern private protected

  (d.h. innerhalb der Klasse, für daraus abgeleitete Klassen und innerhalb des Package sichtbar)
- Achtung: synchronized ist nicht atomar, falls auch nicht-synchronisierte Methoden vorkommen!
- Annahmen bzgl Ausführungsgeschwindigkeiten und Reihenfolgen in der Bearbeitung von Threads sind unzulässig
- Annahme: Threads können jederzeit an jedem Statement unterbrochen werden !!!

Volatile Variable

- Gültige Annahme in JAVA für den Compiler und die Laufzeitumgebung:

„Zwei Referenzen auf dieselbe Instanzvariable in derselben Methode liefern den gleichen Wert.“

- Dies wird zur Optimierung von Zugriffen u.a. in Verbindung mit Caches, Registern genutzt.
- Für sequentielle oder voll synchronisierte Klassen stets richtig.
- Bei Klassen mit synchronisierten und unsynchronisierten Methoden und mehreren Threads muß das nicht stimmen.
- Schlüsselwort: volatile kennzeichnet Variable, deren Wert jeweils aktuell aus dem Hauptspeicher gelesen werden muß.
- Prinzipiell anzuwenden bei
  - Instanzvariable, die in native Methoden genutzt werden
  - Instanzvariable, die in busy-wait Loops genutzt werden

Regrouping

- Regrouping ist eine Möglichkeit zur Berücksichtigung von Constraints, von Abhängigkeiten zwischen Werten
- Variable, deren Werte aufgrund der Semantik zusammenhängen, werden in einer eigenen Hilfsklasse gekapselt
- Beispiel: Position eines graphischen Objektes auf einem Canvas
  - werden x, y Koordinaten (integer) durch accessors und updates jeweils einzeln gesetzt, so sind die Werte jeweils zulässig, aber die Kombination ist nicht notwendigerweise sinnvoll
  - z.B. bei Unterbrechung einer moveTo(X2,Y2) Methode durch accessor Funktionen kann dies zu sinnlosen Kombinationen aus Ursprungskoordinate Y und neuer Zielkoordinate X2 führen

Regrouping

- Strategie
  - definiere Hilfsklasse Value für betroffene Variable, read-only Accessor Funktionen und Konstruktur, der alle Variable initialisiert
  - in der Host Klasse, definiere
    - Referenz val auf neues Hilfsobjekt vom Typ Value
    - eine Methode updateValue(), die atomar eine neue Instanz von Value erzeugt und auf val zuweist
    - eine Accessor Methode value(), die den aktuellen Wert ausgibt
  - wg Konsistenz mit allen internen und externen Anwendungen, greife auf Werte innerhalb eines Objektes vom Typ Value nur nach einer Zuweisung auf eine lokale Variable (current = value()) und auf dieser lokalen Kopie zu.

Beispiel für Regrouping: Verwaltung eines Punktes

```
Public class Point { private int x_, y_ ;
public Point (int x, int y) { x_ = x ; y_ = y ; }
public int x() { return x_ ; }
public int y() { return y_ ; }
```

} Unveränderlich!

```
public class Dot { protected Point loc_ ;
public Dot(int x, int y) { loc_ = new Point(x, y) ; }
public Point location() { return loc_ ; }
protected synchronized void updateLoc( Point loc) { loc_ = loc ;}
public synchronized void moveTo( int x, int y ) { updateLoc(new Point(x,y)); }
public synchronized void shiftX( int deltaX) {
        Point currentLoc = location() ;
        updateLoc(new Point(currentLoc.x()+deltaX,currentLoc.y())) ; }
}
```

Splitting Synchronisation

- Voraussetzung: innerhalb einer Klasse lassen sich Teile identifizieren, deren Verhalten unabhängig, ohne Interaktion oder zumindest ohne Konflikte abläuft
- Idee: delegiere Verhalten an Hilfsklassen mit Objekten feinerer Granularität
- Grundregel in OO Design, hier mit interessanten Aspekten
  - Hilfsobjekte schaffen zusätzliche Locks, über die synchronisiert werden kann
  - dies kann Deadlocks vermeiden helfen
  - dies erhöht das Potential für eine parallele Bearbeitung

Design Schritte

- Verschiebe Funktionalität aus der Host Klasse in eine eigenständige Klasse Helper
- Erzeuge innerhalb der Host Klasse eine eindeutige Instanzvariable der Helper Klasse, die im Konstruktor einmalig belegt wird. (Strict Containment)
- Reiche alle betroffenen Methodenaufrufe innerhalb der Host Klasse ohne Synchronisation an entsprechende, aber synchronisierte Methoden der Helper Klasse. (durchreichende Methoden sind stateless)

- Konzept ist Umkehrung zu Containment
  - Containment: synchronisierterHost kapselt unsynchronisierte Klassen

  Dies impliziert grobe Granularität bzgl Synchronisation.
  - Splitting Classes: unsynchronisierter Host nutzt synchronisierten Helper

  Dies impliziert feine Granularität bzgl Synchronisation, für unabhängige Teilaufgaben innerhalb der Host Klasse können unterschiedliche Helper zum Einsatz kommen.

Splitting Locks

- Wenn Klassen nicht geteilt werden sollen, aber Funktionalität teilbar, können auch mehrere Locks eingesetzt werden.
- Design Schritte
  - Für jeden Teil der Host Klasse definiere eigenes Objekt Lock, das im Host Konstruktor erzeugt und niemals verändert wird.
    - Recycling von Locks aus bestehenden Objekten möglich, z.B. this oder von Instanzvariable, die innerhalb der Klasse eingeschlossen sind. Die jeweilige Klasse spielt i.w. keine Rolle, weil der Lock aus der Objekt Klasse geerbt wird.
  - Für jeden Teil der Host Klasse, deklariere Methoden als unsynchronized, aber innerhalb der Methoden kapsele Code mit synchronized (lock) { ... }

Beispiel: FiFO Queue unbegrenzter Länge mit 2 Locks

- Idee: jeweils 1 Lock (llock) für put Operation am Listenende und 1 Lock (this) für take Operation am Listenanfang
- Elemente der Queue werden in Objekten der Klasse TLQ verwaltet,
  - typisches Konstrukt zur Verwaltung von beliebigen Objekten in einer einfach verketteten Liste

```
final class TLQ {
Object value ; TLQ next ;

TLQ(Object x, TLQ n) {
    value = x ;
    next = n ;
}
}
```

- am Anfang der Liste befindet sich ein leeres Dummy Element mit x = null

Beispiel: FiFO Queue unbegrenzter Länge mit 2 Locks

```
Public class TLQueue {
private TLQ head_ ;      private TLQ last_;      private Object llock ;

public TLQueue() { head = last = new TLQ(null,null) ;
                   llock = new Object() ; }

public void put(Object x) {
        TLQ n = new TLQ(x,null) ;
        synchronized (llock) { last_.next = n ; last_ = node ; } }

public synchronized Object take() {
        Object x = null ;
        TLQ first = head_.next ;
        if (first != null) { x = first.value ; head_ = first ; }
        return(x) ; }
}
```

Zusammenfassung

- Lebendigkeit vs Sicherheit
- Deadlock aus Betriebssystemumfeld
- Strategien nach D. Lea
  - Bottom Up (Lebendigkeit zuerst) vs Top Down (Sicherheit zuerst)
  - Instance Variable Analysis: Accessor, Update Funktionen
  - Reduktion von Synchronisation
    - Volatile Variable
  - Regrouping
  - Splitting