Monitor-Invariante

Eine **Monitor-Invariante (MI)** beschreibt **zulässige Zustände des Monitors**.

Sie gilt, immer wenn ein Prozess den Monitor (wieder) betritt:

- nach der Initialisierung (*),
- am Anfang und am Ende (*) von Entry-Prozeduren,
- vor (*) und nach wait-Aufrufen

Beispiel zum beschränkten Puffer:

MI: 0 <= buf.length() <= n

Für die Stellen (*) muss man zeigen, dass MI gilt.
Dabei kann man voraussetzen, dass MI an den anderen Stellen gilt.

Systematischer Entwurf von Monitoren

1. **Monitor-Zustand** definieren und **Entry-Prozeduren ohne Synchronisation** entwerfen.
   z. B. Beschränkter Puffer mit put und get

2. **Monitor-Invariante** formulieren
   z. B.: MI: 0 <= buf.length() <= n

3. **Wartebedingungen** formulieren und **Bedingungssynchronisation** einfügen
   ```
   while (!Cond) try { wait(); } catch (InterruptedException e) {}
   ```
   an den Programmstellen, die Cond && MI voraussetzen müssen, um MI nicht zu verletzen,
   z. B. vor { buf.length()<n && MI } buf.enqueue(); { MI }

   Eigentlich eine Bedingungsvariable c für jede Bedingung Cond - aber es gibt keine in Java.

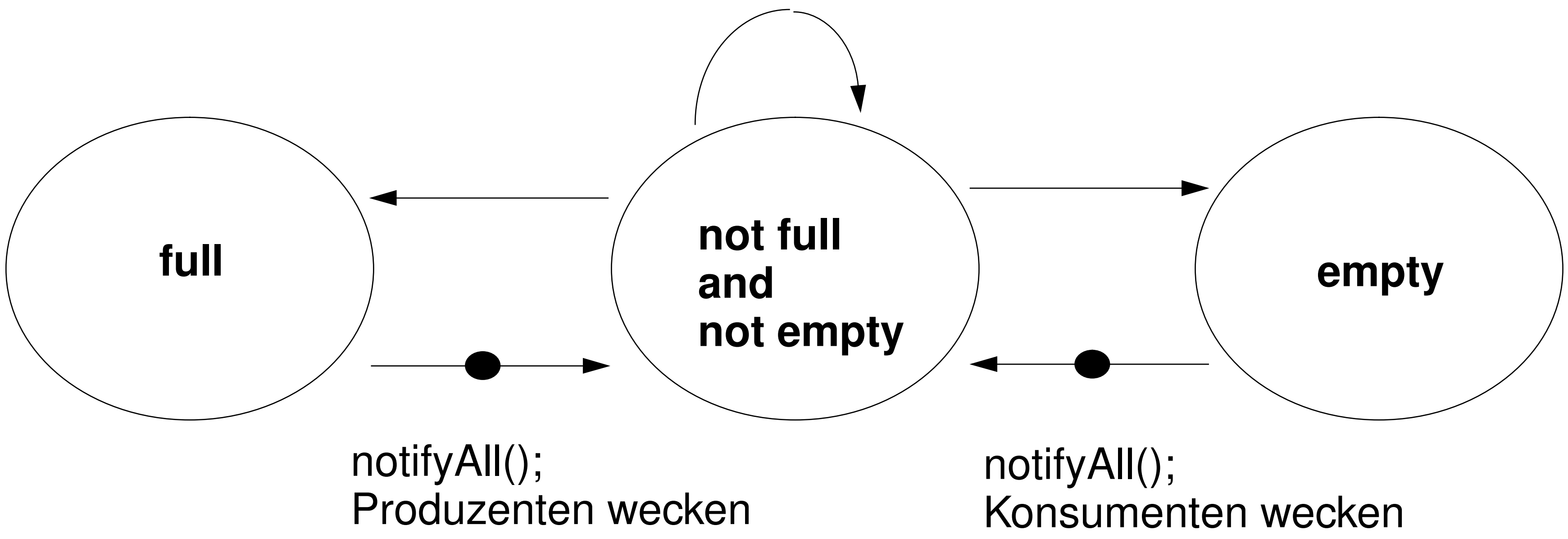
4. **Wecken der Prozesse** einfügen:
   überall wo der Zustand so geändert wird, dass eine Wartebedingung erfüllt sein könnte:
   signal(c) bzw in Java:
   ```
   notifyAll();
   ```
   z. B. nach `buf.dequeue(); notifyAll();`

5. Ggf. überflüssige Aufrufe von `notifyAll` eliminieren. (PPJ-28)
   Zuviele Aufrufe schaden nicht der Korrektheit - nur der Effizienz.
   Zuwenige Aufrufe können Deadlocks verursachen.

Relevante Zustandsänderungen

Nicht bei jeder Änderung einer Zustandsvariablen Prozesse wecken sondern nur bei relevanten Zustandsänderungen: **irgendeine Wartebedingung könnte wahr werden**.

z. B. im Beschränkten Puffer nicht bei jedem Aufruf von `put` und `get` `notifyAll` aufrufen, sondern **vergröberte Zustände**:



Schema: Gleichartige Ressourcen vergeben

Ein **Monitor** verwaltet eine Menge von k >= 1 **gleichartigen Ressourcen**.
Prozesse fordern jeweils n Ressourcen an, 1<=n<=k, und geben sie nach Gebrauch zurück.

Monitor-Invariante: Die Summe der freien und vergebenen Ressourcen ist k.
MI: avail + inUse = k and 0<= avail and 0<= inUse

Wartebedingung für das Anforden: Sind n Ressourcen verfügbar? avail >= n

Beispiele:

- Verleih von Fahrrädern an Ausflugsgruppen
- Vergabe von Speicherblöcken an Prozesse
- Taxi-Vermietung, immer n = 1

auch **abstrakte Ressourcen**:
z. B. das Recht, eine Brücke mit n Tonnen Gewicht zu befahren

Je nach Aufgabenstellung werden

- die **Ressourcen nur gezählt** oder
- ihre **Identität** (Nummern der Taxis, Speicherblöcke) in einer Datenstruktur verwaltet.

Sonderfall n = 1: Alle Prozesse warten auf die gleiche Bedingung: avail >= 1;
also **`notifyAll()`** ersetzen durch **`notify()`**.

Monitor zur Vergabe von Ressourcen

Ein **Monitor** verwaltet eine Menge von k >= 1 **gleichartigen Ressourcen**.
Prozesse fordern jeweils n Ressourcen an, 1<=n<=k, und geben sie nach Gebrauch zurück.

Annahme: Prozess gibt nicht mehr zurück als er bekommen hat => vereinfachte Invariante:

```
class Resources
{  private int avail;                                    // Invariante: avail >= 0

   public Resources (int k) { avail = k; }

   synchronized public void getElems (int n)             // n Elemente abgeben
   {  while (avail<n)                                    // negierte Wartebedingung
         try { wait(); } catch (InterruptedException e) {}
      avail -= n;
   }

   synchronized public void putElems (int n)             // n Elemente zurückgeben
   {  avail += n;                                        // wegen Annahme ist Warten nicht nötig
      notifyAll();                                       // nicht notify()!
   }
}
```

Prozesse und Hauptprogramm zu Ressourcen-Monitor

```
import java.util.Random;

class Client extends Thread
{  private Resources mon; private Random rand;
   private int ident, rounds, maximum;

   public Client (Resources m, int id, int rd, int max)
   {  mon = m; ident = id; rounds = rd; maximum = max;
      rand = new Random();                    // Zufallszahlengenerator bestimmt die
   }                                          // in jeder Runde angeforderten Elemente

   public void run ()                         // und die Wartezeit bis zur Rückgabe
   {  while (rounds > 0)
      {  int m = Math.abs(rand.nextInt()) % maximum + 1;
         mon.getElems (m);
         try { sleep (Math.abs(rand.nextInt()) % 1000 + 1); }
            catch (InterruptedException e) {}
         mon.putElems (m);
         rounds--;
      }
   }
}
```

```
public class TestResource
{  public static void main (String[] args)
   {  int avail = 20;
      Resources mon = new Resources (avail);
      for (int i=0; i<5; i++)
         new Client (mon, i, 4, avail).start();
   }
}
```

Leser/Schreiber-Schema

Ein Monitor verwaltet lesende und schreibende Zugriffe von Prozessen auf eine Datenbasis.

Entwurfsschritte:

1. **Zustand**: Zahl der gerade lesenden (nr) und schreibenden (nw) Prozesse
   Operationen: requestRead, releaseRead, requestWrite, releaseWrite

2. **Monitor-Invariante** RW: (nr == 0 or nw == 0) and nw <= 1

3. **Wartebedingungen** vor Erhöhen der Zähler:
   in requestRead: { nw == 0 and RW } nr++; { RW }
   in requestWrite: { nr == 0 and nw == 0 and RW } nw++; { RW }

   verschiedene, überlappende Wartebedingungen - typisch für Leser/Schreiber-Schema

4. **Prozesse wecken** nach Erniedrigen der Zähler:
   in releaseRead: nr--; notifyAll();
   in releaseWrite: nw--; notifyAll();

5. **nur relevante Zustandsänderungen**:
   in releaseRead: nr--; if (nr==0) notify();
   Wecken eines wartenden Schreiber genügt.

Leser/Schreiber-Monitor

```
class ReaderWriter
{  private int nr = 0, nw = 0;
                   // Monitor-Invariante RW: (nr == 0 || nw == 0) && nw <= 1
   synchronized public void requestRead ()
   {  while (nw > 0)                                    // negierte Wartebedingung
         try { wait(); } catch (InterruptedException e) {}
      nr++;
   }
   synchronized public void releaseRead ()
   {  nr--;
      if (nr == 0) notify ();                     // einen Schreiber wecken genügt
   }

   synchronized public void requestWrite ()
   {  while (nr > 0 || nw > 0)                          // negierte Wartebedingung
         try { wait(); } catch (InterruptedException e) {}
      nw++;
   }
   synchronized public void releaseWrite ()
   {  nw--;
      notifyAll ();          // 1 Schreiber und alle Leser zu wecken würde genügen!
   }
}
```

Schema: Begegnungen von Prozessen

Prozesse durchlaufen **verschiedene Stationen** und interagieren dabei.
Ein Monitor organisiert die **Begegnungen in der vorgeschriebenen Abfolge**.

Entwurfsmethode:

Zustände durch **Zählvariable** beschreiben;
zulässige Zustände durch **Invariante** über Zählvariable charakterisieren;
daraus **Wartebedingungen** der Monitor-Operationen herleiten;
Zählvariable durch binäre **Variable substituieren**.

Beispiel **Sleeping Barber**:

In einem verschlafenen Ort im südlichen Ostwestfalen wartet ein Friseur in seinem Laden auf Kundschaft. Wenn ein Kunde im Frisierstuhl platzgenommen hat, schneidet er ihm die Haare. Wenn er den Haarschnitt beendet hat, kassiert er und verabschiedet den Kunden.

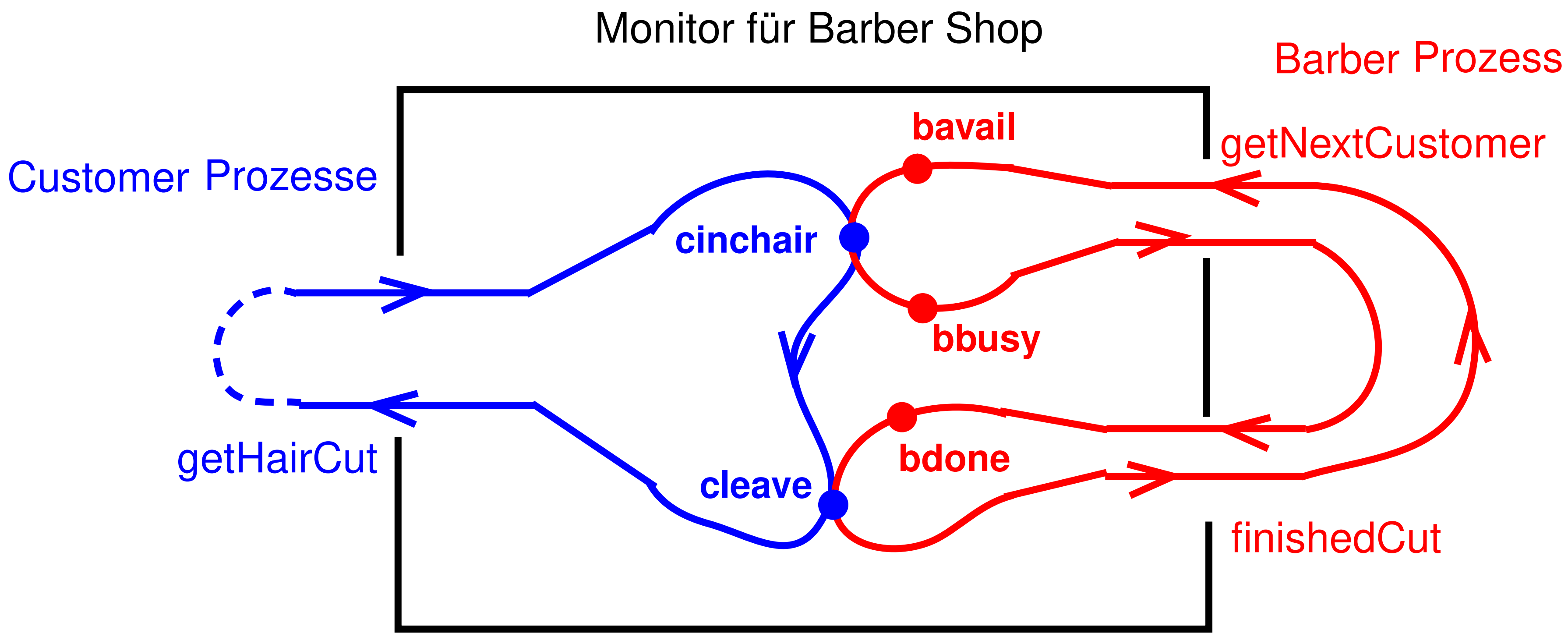
Ein Kunde, der den Friseurladen betritt, setzt sich in den Frisierstuhl, falls er frei ist, und wartet auf seinen Haarschnitt. Falls der Frisierstuhl besetzt ist, wartet er auf einem anderen Platz bis er dran ist.

2 Arten von Prozessen: Barber (1 Exemplar), Customer (viele Exemplare)

2 Begegnungen: Haarschnitt und verabschieden

Die Aufgabe ist auch ein Beispiel für das Client/Server-Schema.

Monitor-Entwurf zum Sleeping Barber



Invariante über Zählvariable:

C1: cinchair >= cleave and bavail >= bbusy >= bdone wird durch die Prozesse erfüllt.

C2: bavail >= cinchair >= bbusy führt zu 2 Wartebedingungen

C3: bdone >= cleave führt zu 1 Wartebedingung

Monitor-Invariante BARBER: C1 and C2 and C3

Grobentwurf mit Zählvariablen und Wartebedingungen

Monitor-Invariante

BARBER: cinchair >= cleave && bavail >= bbusy >= bdone &&
bavail >= cinchair >= bbusy &&
bdone >= cleave

	Wartebedingung:	**Zählvariable erhöhen:**	
getHairCut			Customer Prozesse:
	cinchair < bavail	cinchair++;	
	cleave < bdone	cleave++;	
getNextCustomer			Barber Prozess:
		bavail++;	
	busy < cinchair	bbusy++;	
finishedCut			
		bdone++;	
	bdone == cleave	Kunde verabschiedet	

Zählvariable substituieren

neue binäre Variable: barber = bavail - cinchair
chair = cinchair - bbusy
open = bdone - cleave

alte Invariante:
bavail >= cinchair >= bbusy &&
bdone >= cleave

substituierte Invariante:
barber >= 0 && chair >= 0 &&
open >= 0

Wartebedingung:	**Zustandswechsel:**	
getHairCut		Customer Prozesse:
barber > 0	barber--;	
	chair++; wecken	
open > 0	open--;	
getNextCustomer		Barber Prozess:
	barber++; wecken	
chair > 0	chair--;	
finishedCut		
	open++; wecken	
open == 0	Kunde verabschiedet	