Fachhochschule
Münster University of
Applied Sciences

Java Thread Synchronisierung

---

wait – notify und Semaphor Synchronisierung

Prof. Dr. Nikolaus Wulff

Das Philosophen Problem

- Das „essende Philosophen Problem“ wurden 1968 von Dijkstra formuliert und illustriert ein klassisches Beispiel der nebenläufigen Programmierung.
- N Philosophen denken, werden hungrig und essen an einer Tafel, mit lediglich M ≤ N Löffeln und Gabeln.
- Hat ein Philosoph einen Löffel, so hält er diesen fest und wartet auf eine frei werdende Gabel oder er hält eine Gabel und wartet auf einen Löffel.
- Je nach Wahl von N und M kann es passieren, dass einige Philosophen einen Löffel und Andere eine Gabel besitzen. Keiner kann essen, niemand wird satt und gibt sein Besteck zurück. => **Deadlock**

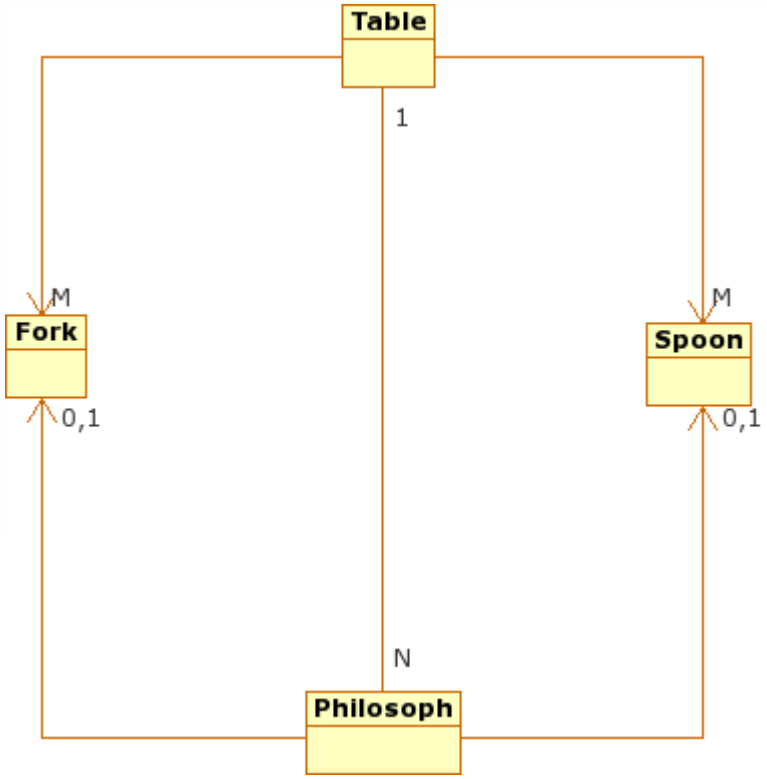
Sequenz der Philosopheninteraktion

- Jeder Philosoph wird als eigenständiger Thread gestartet, so dass alle unabhängig laufen.
- Um zu Essen müssen nacheinander zwei Ressourcen - Gabel und Löffel -, vom Tisch genommen werden.
- Erhält der Philosoph keine von beiden so **wartet** er.
- Erhält er nur eine Ressource, so **blockiert** er diese und **wartet** auf die Zweite.
- Erst wenn er beide Ressourcen hat isst er und legt anschließend Gabel und Löffel auf den Tisch zurück, um wieder zu denken.
- Der nächste blockierte Philosoph wird geweckt...

Prozessalgebra

- FORK = (get → put → FORK).
- SPOON = (get → put → SPOON).
- PHIL = (think
  → (fork.get → spoon.get | spoon.get → fork.get)
  → eat
  → (fork.put → spoon.put | spoon.put → fork.put)
  → PHIL).
- Der versteckte Fehler in dieser Prozessalgebra ist schwer zu erkennen und der Zustandsraum von (p[1..N]:PHIL|| f[1..M]:FORK||s[1..M]:SPOON) wird sehr schnell unübersichtlich...

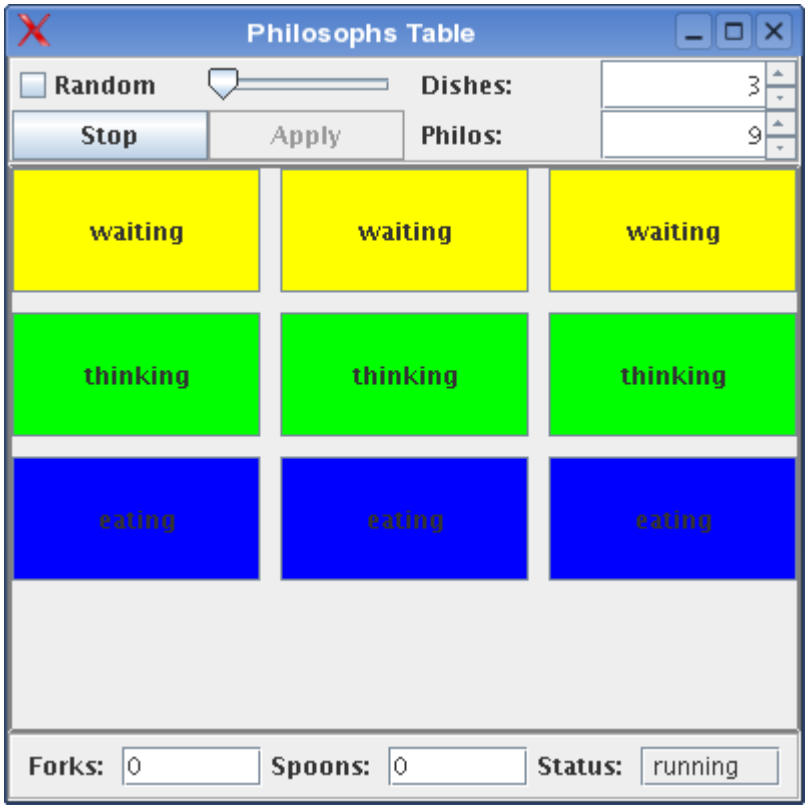
Philosophen Runde



Forks und Spoons werden durch zwei Semaphore synchronisiert.

- N Philosophen wetteifern um die M Ressourcen.
  - Ohne Synchronisierung kommt es zur **Race Condition**.
  - Mit (falscher) Synchronisierung zum **Deadlock**.

Philosophen Deadlock



- Bei zufälliger Reihenfolge von Gabel/Löffel-Zugriff kommt es nach einiger Zeit zur Verklemmung. Solche Fehler sind sehr schwer durch Testen zu entdecken!

Labeled Transition System

- Das Lehrbuch Concurrency von J. Magee bietet auf seiner Web-Site das LTSA Werkzeug an und kann eine Prozessalgebra auswerten und analysieren.
- Vereinfachte Problembeschreibung mit N=2 und M=1 in der LTSA Skript Sprache:

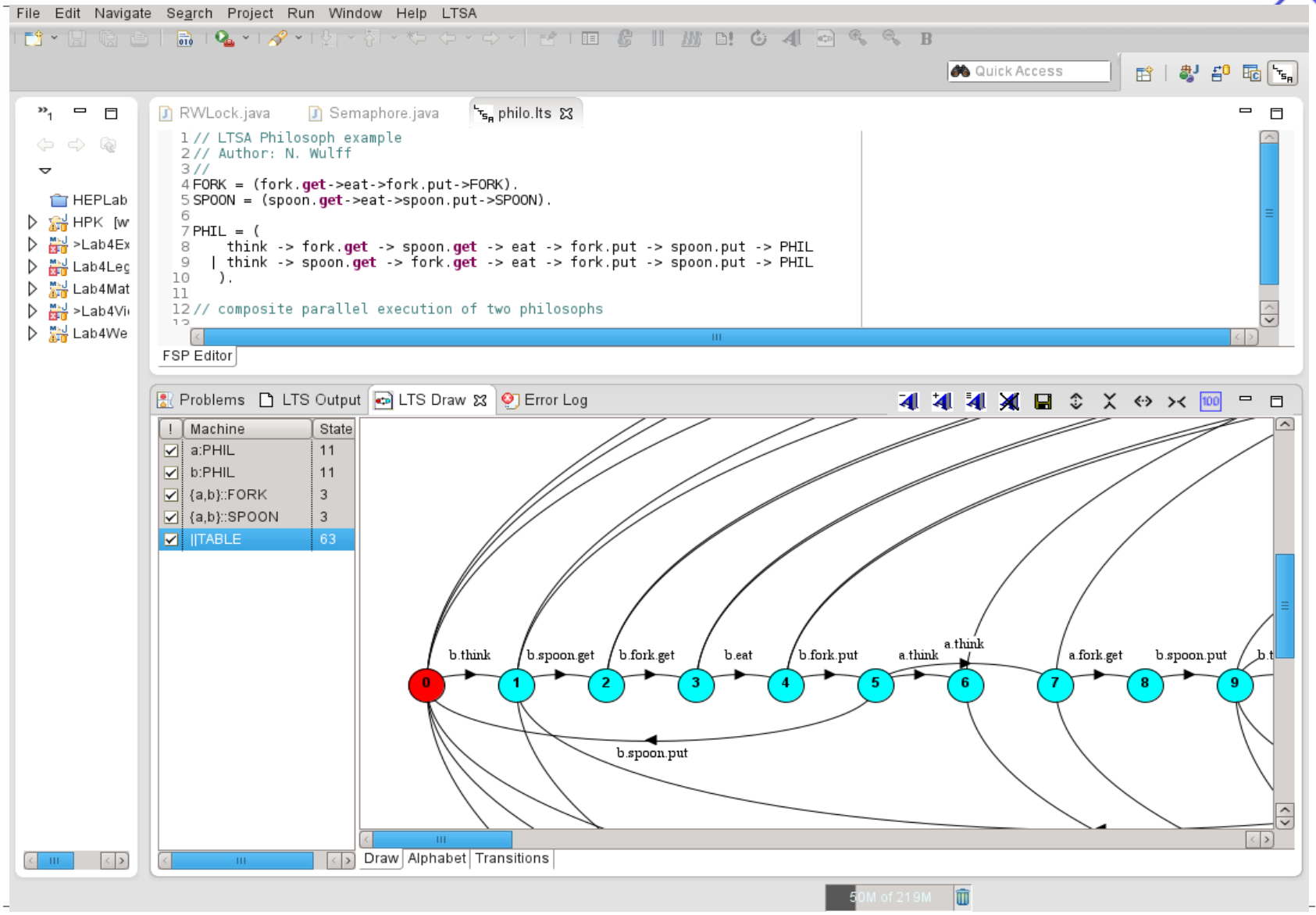
```
// LTSA Philosoph example
// Author: N. Wulff
//
FORK = (fork.get->eat->fork.put->FORK).
SPOON = (spoon.get->eat->spoon.put->SPOON).

PHIL = (
    think -> fork.get -> spoon.get -> eat -> fork.put -> spoon.put -> PHIL
  | think -> spoon.get -> fork.get -> eat -> fork.put -> spoon.put -> PHIL
   ).

// composite parallel execution of two philosophs

||TABLE = (a:PHIL || b:PHIL || {a,b}::FORK || {a,b}::SPOON).
```

LTSA Eclipse Plugin



LTS Analyse

- Eine LTS Analyse der Prozessalgebra zeigt sofort den möglichen Deadlock auf:

```
Composition:
TABLE = a:PHIL || b:PHIL || {a,b}::FORK || {a,b}::SPOON
State Space:
 11 * 11 * 3 * 3 = 2 ** 12
Analysing...
Depth 5 Trace to DEADLOCK:
a.think
a.fork.get
b.think
b.spoon.get
Analysed in: 2ms
```

- Schon bevor die Anwendung programmiert wird kann ein potentieller Fehler durch die Analyse des LTS Prozess erkannt werden...

Semaphore & ResourceManger

- Eine Implementierung des ResourcePools wird synchronisierte Methoden verwenden, um die Resourcen zu schützen. (Siehe Übung).

```
public interface ResourcePool<T> {
    /**
     * Acquire a resource of type T,
     * this call might block if none available.
     * @return T resource
     */
     T require();
    /**
     * Release a previously acquired resource back to the pool.
     * @param o T resource to release.
     */
     void release(final T o);
}
```

wait-notify Synchronsierung

- Eine einfache Pool Implementierung mit einem T-Feld synchronisiert per wait-notify.

```
private int size;
private T[]  pool;
 public synchronized void release(T resource) {
     pool[size++] = resource;
     notify();
 }
 public synchronized T require() {
     try {
         while (size == 0) {
             wait();
         }
     } catch (InterruptedException e) {
         e.printStackTrace();
     }
     return pool[--size];
 }
```

- Die await-Bedingung wird in einer Schleife überprüft.
- Synchronisierung und fachliche Logik sind eng gekoppelt und schwieriger zu verstehen...

Explizite Object Synchronisierung

```
public void release(T resource) {
    synchronized (pool) {
        pool.add(resource);
        pool.notify();
    } // release monitor lock
}
public T require() {
    synchronized (pool) {
        try {
            while (pool.size() == 0) {
                pool.wait();
            }
        } catch (InterruptedException e) {
        }
        return pool.remove(0);
    } // release monitor lock
}
```

Pool ist jetzt eine ArrayList<T>

Disassemblierter Java Byte-Code

```
public void release(java.lang.Object);
  Code:
   0:   aload_0
   1:   getfield        #21; //Field pool:Ljava/util/ArrayList;
   4:   dup
   5:   astore_2
   6:   monitorenter
   7:   aload_0
   8:   getfield        #21; //Field pool:Ljava/util/ArrayList;
   11:  aload_1
   12:  invokevirtual   #23; //Method java/util/ArrayList.add:
                                    (Ljava/lang/Object;)Z
   15:  pop
   16:  aload_0
   17:  getfield        #21; //Field pool:Ljava/util/ArrayList;
   20:  invokevirtual   #43; //Method java/lang/Object.notify:()V
   23:  aload_2
   24:  monitorexit
   25:  goto    31
   28:  aload_2
   29:  monitorexit
   30:  athrow
   31:  return
```

```
synchronized (pool) {
    pool.add(resource);
    pool.notify();
}
```

Simple Semaphore Implementierung

```
@Deprecated
public class Semaphore {
    private volatile int free;
    public Semaphore(int size) {
        free = size;
    }
    public synchronized void release() {
        free++;
        notify();
    }
    public synchronized void acquire() {
        while (free <= 0) {
            try {
                wait();
            } catch (InterruptedException e) { /*ignore */
               e.printStackTrace();
            }
        }
        free--;
    }
}
```

Diese einfache, unfaire Implementierung sollte seit JDK 1.5 nicht mehr verwendet werden.

Pool mit Semaphore

- Häufig wird die Synchronisierung in einer Semaphore Instanz ausgelagert.

```
private List<T>  pool;

public void release(T resource) {
    pool.add(resource);
    semaphore.release();
}

public T require() throws InterruptedException {
    semaphore.acquire();
    return pool.remove(0);
}
```

- Dieser Pool verwendet intern eine List<T> und delegiert die komplette Synchronisierung an die Semaphore Instanz.

ResourcePool mit BUG!!!

```
public synchronized void release(T resource) {
    pool.add(resource);
    semaphore.release();
}

public synchronized T require() throws InterruptedException {
    semaphore.acquire();
    return pool.remove(0);
}
```

- Dieser fehlerhafte ResourcePool enthält einen schwerwiegenden Bug und führt zu einem Deadlock!
- Ein JUnit Test zum Entdecken des Fehlers kostet einiges an Aufwand...
  - An welchen Objekten existiert eine „Monitorsperre“?

Achtung

- Ein Semaphor eliminiert Race-Condition. Der Pool wird einen Client solange blockieren, bis eine freie Ressource zur Verfügung steht.

Dies generiert leider weitere Probleme:

- Wer garantiert, dass der Client die Ressource in den Pool zurückgibt?
- Wer verhindert, dass nach der Rückgabe an den Pool diese Ressource nicht weiterhin vom ursprünglichen Client verwendet wird?

Abhilfe

Dies lässt sich mit Hilfe verschiedener Strategien verhindern:

- Die Referenz wird ungültig nach dem *release*-Aufruf.
- Die Referenz ist eine bestimmte Zeitspanne gültig. Danach generiert ihrer Verwendung durch den Client eine TimeoutException.
- Ein interner GarbageCollector sammelt nicht zurückgegebene Ressourcen ein und gibt diese in den Pool zurück. Die Referenz ist ab dann ungültig.

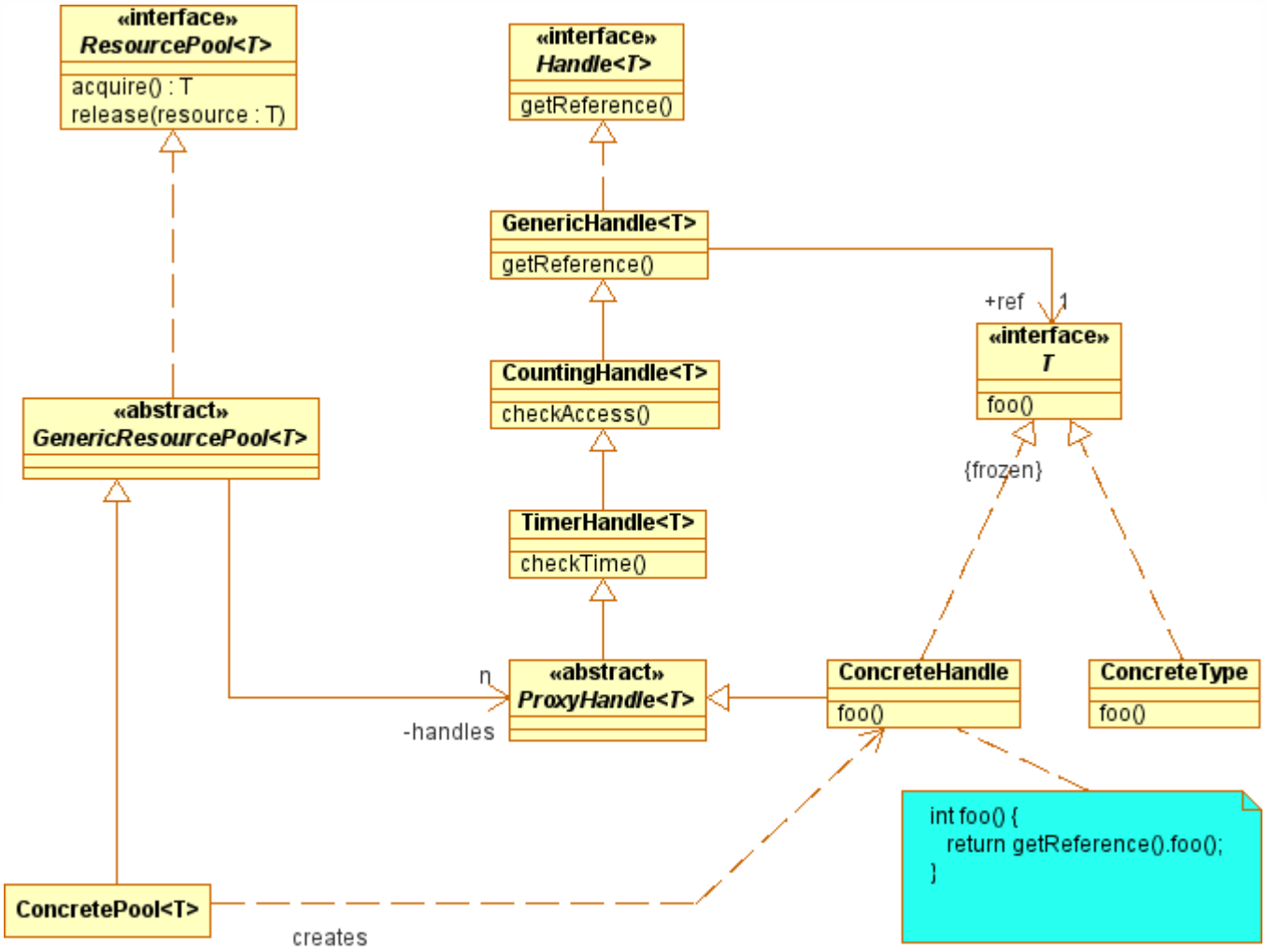
Problem

- Wenn die Referenz entweder ungültig wird oder einen Timeout hervorruft lohnt es sich nicht diese in einem Pool zu cachen...

Lösung:

- Der Pool gibt keine „echte Referenz“ an den Client heraus, sondern ein Stellvertreterobjekt, ein Proxy.
- Dieses Proxy hat dieselbe Schnittstelle wie die echte Referenz, kann aber durch den Pool administriert werden, um *invalide* oder per timeout *disabled* oder garbage-collected zu werden.

ResourcePool Proxy



Prinzip des Proxy

- Ressource und Proxy implementieren die selbe Schnittstelle <T>.
- Jeder Methodenaufruf wird vom Proxy an die Reference weiter delegiert:

```
public int foo() {
    return getReference().foo();
}
```

- Die getReference Method bietet die Möglichkeit zu überprüfen, ob die Reference noch gültig ist!

```
public T getReference() {
      checkAccess();
      return super.getReference();
 }
```

Resource Recycling

- Der ResourcePool invalidiert ein Proxy wenn es per release zurückgegeben wird. Ab dann ist die Ressource mit diesem Proxy nicht mehr erreichbar, da jeder Aufruf zu einer Exception beim Client führt.
- Sollte das Proxy nicht zurückgegeben werden, so hat dies einen Timestamp für seine Lebensdauer, danach führt jede weitere Verwendung zu einer Exception.
- Der ResourcePool hat einen Garbage-Collector Thread, der alle herausgegebenen Proxys nach einer vorgegebener Zeit invalidiert und die Resourcen in den Pool zurückgibt. So werden sie dem Client effektiv entzogen falls er den *release*-Aufruf vergißt...

Timeout

- Überprüfung der Lebensdauer:

```
/**
 * check that the access is within the
 * time frame, otherwise an exception
 * will be thrown.
 */
private void checkAccessTime() {
    long now = System.currentTimeMillis();
    if (now>endTime) {
        String msg = String.format("time excess: %d mSec delay",
                                   now-endTime);
        throw new TimerException(msg);
    }
}
```

- Wird dies in der getReference() Methode aufgerufen, so ist das Proxy nach dem Timeout für den Client effektiv nicht mehr zu gebrauchen...

Proxy Vor- und Nachteile

- Mit Hilfe des Proxy Ansatz lassen sich Resourcen effektiv und sicher wieder an den ResourcePool zurückgeben.
- Ein Nachteil ist, dass hierzu eine gemeinsame Schnittstelle vorliegen muss, oder die Proxys müssen von der Ressource ableitbar sein. (Dann sind es keine eigentlichen Proxys mehr, sondern Dekorierer...)
- C^{++} gestattet es die Referenzierungsoperatoren „->“ und „*“ sowie den Zuweisungsoperator „=“ zu überladen, so dass mit einem „intelligenten Zeiger“ eine effektive Überprüfung der Zugriffe erfolgt.

C++ Handle

```
template <class T> class Handle {
  private: T* ptr;

  public:
   Handle() :ptr(0) { }
   Handle(T* ref) :ptr(ref) { assert(ptr != 0); }
  ~Handle() { if(ptr) delete ptr; ptr=0;}
   // assignment
   Handle& operator=(const Handle&);
   // pointer access
   T& operator*() const;
   T* operator->() const;
   operator bool() const {return ptr!=0;}
   // invalidate the handle
   void invalidate() { ptr = 0;}
};
```

Daumenregeln gegen Deadlocks

- Lokale Methodenvariablen sind immer Thread-sicher und benötigen keine Synchronisierungsmechanismen.
- Globale read-only Variablen sind Thread-sicher
  - also entweder final primitive Attribute
  - oder Objekte, die selber wiederum nicht veränderbar sind.
- Synchronisierte Methoden vermeiden oder aber sicherstellen, das diese immer! in der selben Reihenfolge Lock-Objekte alloziieren.
  - letzte Bedingung erfordert Analyse des WFG.
  - => UML Sequenzdiagramm kann helfen
  - oder prinzipielle LTS Analyse...