Wahlpflichtfach Design Pattern

- Dipl.-Inf., Dipl.-Ing. (FH) Michael Wilhelm
- Hochschule Harz
- FB Automatisierung und Informatik
- miwilhelm@hs-harz.de
- http://www.miwilhelm.de
- Raum 2.202
- Tel. 03943 / 659 338

Inhalt

1. Beispiel: Singleton

Problem:

- **Sicherstellen, daß nur eine Instanz existiert.**
- **Allgemeiner Zugriff auf diese Instanz muss sichergestellt werden.**

Singleton, Java

```
public class Singleton {

int value = 0;

public Singleton(int value){
        this.value = value;
}


static public void main(String[] args) {
        Singleton a = new Singleton(1);
        Singleton b = new Singleton(2);
        Singleton c = new Singleton(3);
 }

}
```

Singleton, Java

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

  public static Singleton getInstance(){
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```


Singleton, Java

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

  public static synchronized Singleton getInstance(){
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Singleton, Java

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

  public static Singleton getInstance(){
          synchronized(Singleton.class) {
            if (instance == null) {
               instance = new Singleton();
            }
          }

       return instance;
     }
}
```


Singleton, Java, Double-Checked

```
public class Singleton {
    private static Singleton instance = null;
    protected Singleton(){
    }

  public static Singleton getInstance(){
       if (instance == null) {
         synchronized(Singleton.class) {
            if (instance == null) {              // ???
               instance = new Singleton();
    }
          }
       }
       return instance;
     }
}
```

Singleton, Java, Double-Checked

```
public class Singleton {
    private static Singleton instance = null;
    private boolean okay=false;
    protected Singleton(){
    }
   public static Singleton getInstance(){
        if (instance == null  || !okay) {
          synchronized(Singleton.class) {
            if (instance == null) {              // ???
              instance = new Singleton();
              okay=true;
            }
          }
        }
        return instance;
      }
}
```


Singleton, Java, Double-Checked

```
public class Singleton {
    private static Singleton instance = null;
    private static boolean okay=false;
    protected Singleton(){
    }
   public static Singleton getInstance(){
        if (!okay) {
          synchronized(Singleton.class) {
            if (instance == null) {              // ???
              instance = new Singleton();
              okay=true;
            }
          }
        }
        return instance;
      }
}
```

Singleton, Java, Double-Checked

- Das könnte doch funktionieren.
- Das Speichermodell der Java-Plattform kann zu Problemen führen.
- Die zweite Bedingung if (instance == null) kann nämlich zu true evaluieren, ohne dass „new Singleton()“ aufgerufen, und das instanziierte Objekt dem Klassenattribut instance zugewiesen wurde!
- Um das exemplarisch zu verdeutlichen, betrachtet man den folgenden Pseudo-Bytecode für den Befehl:
  - instance = new DoubleCheckedLockingSingleton() an:

Quelle:
http://www.theserverside.de/singleton-pattern-in-java/

Singleton, Java, Double-Checked

```
01.    // Speicher alloziieren
02.    ptrMemory = allocateMemory()
03.    // Den Speicher dem Klassenattribut zuweisen, ab hier gilt: instance != null
04.    assignMemory(instance, ptrMemory)
05.    // Den Konstruktor aufrufen, das Objekt ist dann ab hier korrekt instanziiert
06.    callDoubleCheckedLockingSingletonConstructor(instance)
```

- Zwischen der 4. und der 6. Zeile könnte die Ausführung des Threads durch die Java Virtual Machine unterbrochen, und die Ausführung eines zweiten Threads vorgezogen werden.
- Die zweite Bedingung if (instance == null) würde damit zu true evaluieren, ohne dass bisher der Konstruktor (Zeile 6) aufgerufen worden wäre. Das double-checked locking ist damit nicht sicher.
- Konstrukte wie diese sind in JIT-Compilern nicht unüblich und da man nicht immer voraussagen kann, wo der Code läuft sollte man das double-checked locking vermeiden, um auf der sicheren Seite zu stehen.

Quelle:
http://www.theserverside.de/singleton-pattern-in-java/

Singleton, Java: Korrekte Lösung

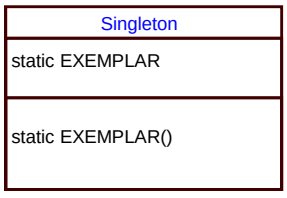
```
public class Singleton {
    private static Singleton instance = new Singleton();
    protected Singleton(){
    }

   public static Singleton getInstance(){
        return instance;
     }
}
```

Threadsicher ?

Singleton: UML

Problem:

- **Sicherstellen, daß nur eine Instanz existiert.**
- **Allgemeiner Zugriff auf diese Instanz muss sichergestellt werden.**



Singleton

Zweck:

- Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.

Motivation:

- Eindeutiger Druckerspooler, ein Dateisystem, ein Fenstersystem
- Globale Variable ermöglicht Zugriff auf das Objekt, eindeutig?
- Einzigartigkeit in der Klasse abgefangen

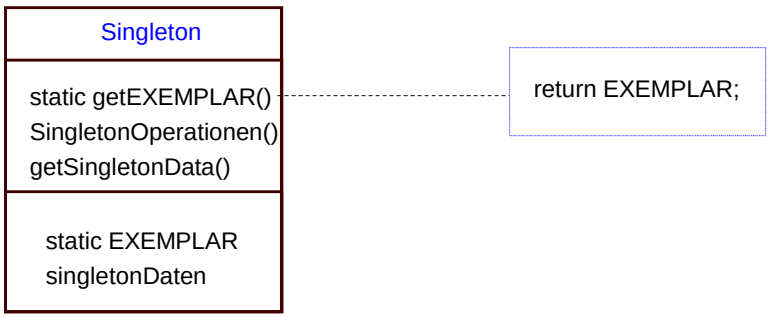
Anwendbarkeit:

Verwenden Sie das Singletonmuster, wenn

- es genau ein Exemplar einer Klasse geben muss und der Zugriff eindeutig sein soll.
- das einzige Exemplar durch Bildung von Unterklassen erweiterbar sein soll. Anwender sollen die Erweiterungen ohne Modifikation weiter verwenden.

Singleton mit UML-Notation

Struktur:



Singleton

Teilnehmer:
- definiert eine Exemplaroperation, die es Klienten ermöglicht, auf sein einziges Exemplar zuzugreifen.
- ist potenziell für die Erzeugung seines einzigen Exemplars zuständig.

Interaktion:
- Klienten greifen ausschließlich durch die getExemplar-Operation der Singletonklasse zu.

Konsequenzen / Vorteile:
- *Zugriffskontrolle auf das Exemplar.* Aus der Kapselung des Konstruktors folgt die strikte Kontrolle.
- *Kleinerer Namensraum.* Das Singletonmuster ist eine Verbesserung gegenüber globalen Variablen

Singleton

Konsequenzen / Vorteile:
- *Verfeinerung von Operationen und Repräsentation*. Die Klasse kann abgeleitet und spezialisiert werden. Die Konfiguration zur Laufzeit ist möglich.
- *Variable Anzahl von Exemplaren*. Änderung der Methode getExemplar() ist einem Konstruktor stellt den Normalzustand ohne weitere Codeänderung her.

Implementierung:
- *Problem Ableitung neuer Klassen.*
- Damit können mehrere Instanzen existieren.
- Problem Serialize

Singleton mit enums, Java

```
enum Singleton {
    INSTANCE;
  public int i=33;
  private Singleton() {
    this(0);
  }
  private Singleton(int i) {
    this.i = i;
  }
} // Singleton6
```

- Der Vorteil dieser Variante ist, dass sie selbst aufwändigen Manipulationsversuchen durch Serialisierung und/oder Introspektion um eine Mehrfachinstanziierung zu ermöglichen widersteht.
- Es ist die derzeit beste Methode, Singletons zu implementieren.
- Der Nachteil ist jedoch, dass diese Methode erst ab Java Version 1.5 funktioniert.

Singleton mit enums, Java

```
static public void main(String[] args) {
    System.out.println(" ");

    Singleton  a  = Singleton.INSTANCE;
    System.out.println("a: "+ a.getValue()  );
    a.setValue(2222);
    System.out.println(" ");

    Singleton  b = Singleton.INSTANCE;
    b.setValue(1111);
    System.out.println("a: "+ a.getValue()  );
    System.out.println("b: "+ b.getValue()  );
}
```

Singleton, C#: Beispiel SingletonA

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Singleton1 {

   sealed class Singleton {
      private static Singleton instance = new Singleton();
      protected Singleton() {
      }

      public static Singleton getInstance() {
         return instance;
      }

   }
}
```


Simple Singleton in C#

```
sealed class Singleton {

    private Singleton() {
    }

   public static readonly Singleton Instance = new Singleton();

}
```

Singleton, C#: Beispiel SingletonB

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Singleton1 {

   sealed class Singleton {
      private static Singleton instance = new Singleton();
      protected Singleton() {
      }

          // Eigenschafts-Notation          public int Std {get; private set;}
       public static readonly SingletonB Instance = new SingletonB();

   }
}
```


Double-Check Lock in C#

```
sealed class Singleton {
   private static volatile Singleton _instance = null;

   protected Singleton() {
   }

   public static Singleton Instance() {
      if (_instance == null) {
         lock (typeof(Singleton)) {
            if (_instance == null) {
               _instance = new Singleton();
            }
         }
      }
      return _instance;
   }
}
```

Double-Check Lock in C#

```
sealed class Singleton {
   private static volatile Singleton _instance = null;
   private bool okay=false;

   protected Singleton() {
   }

   public static Singleton Instance() {
      if (!okay)    // _instance == null) {
         lock (typeof(Singleton)) {
            if (_instance == null) {
               _instance = new Singleton();
               okay=true;
            }
         }
      }
      return _instance;
   }
}
```


Singleton mit Lazy Creation

- In den vorherigen Lösungen werden die Instanzen immer sofort beim Starten erzeugt.
- Bei größeren Klassen bzw. bei Klassen, deren Instanz man nicht immer benötigt, wäre eine spätere Instanziierung sinnvoller.
- Diese Technik heißt „lazy creation“
- In Java wäre das das Beispiel „Singleton mit Double-Checked“

Singleton mit Lazy Creation in C#: Beispiel SingletonC

```
class SingletonC {

    protected SingletonC() {    }


   class SingletonCreator {
     static SingletonCreator() {   }

        // internal besser als private
      internal static readonly SingletonC instance = new SingletonC();
    }

     public static SingletonC Instance {
        get { return SingletonCreator.instance; }
     }

  }
```


volatile vs. synchronized

- Bei Multithreading-Anwendungen müssen Variablen gegenüber dem gleichzeigen Schreiben gesichert werden.
- Dazu existieren die beiden Techniken „volatile“ und „synchronized“.
- **synchronized:**
  - Mit „synchronized“ sichert man den Zugriff auf eine Methode Thread-sicher ab.
  - Die Änderung wird aber nicht automatisch weitergeleitet.

- **volatile:**
  - Mithilfe des Schlüsselworts volatile lässt sich der Zugriff auf eine Variable synchronisieren. Ist eine Variable als volatile deklariert, muss die JVM sicherstellen, dass alle zugreifenden Threads ihre Kopien aktualisieren, sobald die Variable geändert wird.
  - Eine Sperre à la „lock“ ist nicht vorgesehen.
  - Aber nun wird beim jedem Zugriff ein Broadcast gesendet.
  - Das volatile Increment bzw. Decrement ist nicht threadsicher.

Volatile und Increment resp. Decrement

Seit Java 5 helfen hier die Klassen des Pakets java.util.concurrent.atomic. Diese besitzen die geeigneten Methoden.

Klassen:
- AtomicBoolean
- AtomicInteger
- AtomicLong

Felder:
- AtomicIntegerArray
- AtomicLongArray.

Reflection an Attribute und Referenzen:
- AtomicIntegerFieldUpdater<T>
- AtomicLongFieldUpdater<T>
- AtomicReference
- AtomicReferenceArray<E>
- AtomicReferenceFieldUpdater<T,V>
- AtomicMarkableReference<V>
- AtomicStampedReference<V>

Volatile und Increment resp. Decrement

```
public class Id  {

   private static final AtomicLong id = new AtomicLong();

   private Id() {
    }

   public long next() {
     return id.getAndIncrement();
   }

}
```

Quelle: com/tutego/insel/thread/atomic/Id.java