Objektorientiertes Programmieren

Einführung in objektorientiertes Programmieren

`13 - Referenzen, Design Pattern II, Parallele Verarbeitung`

Bachelor Wirtschaftsinformatik

Marcel Tilly

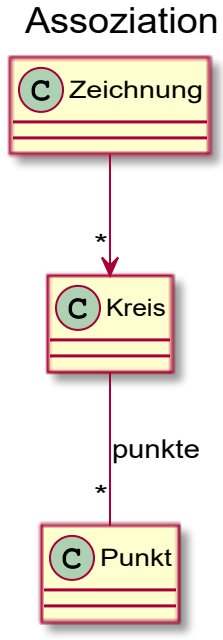
Fakultät Informatik, Cloud Computing

Vererbung (Wiederholung)

- Vererbung bildet in der OOP eine *ist-ein*-Beziehung ab
  - Der Mensch ist ein Säugetier
- Systematische Spezialisierung von oben nach unten
- Die *Unterklasse* erbt damit alle Merkmale der *Oberklasse*
- Unterklassen erben alle zugreifbaren Member der Basisklasse:
  - Konstruktor und Destruktor
  - Attribute / Klassenvariablen
  - Methoden und Operatoren
- Was wird nicht vererbt?
  - Member die als *private* deklariert sind

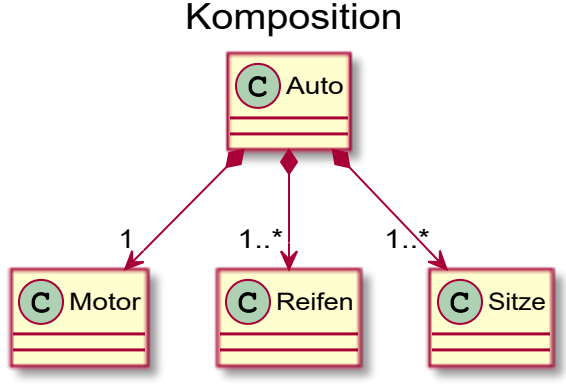
Assoziation

- lose Verbindung zwischen Objekten (Referenz)
- Objekte kennen sich (Richtung der Referenz)
- Kardinalitäten
- Name



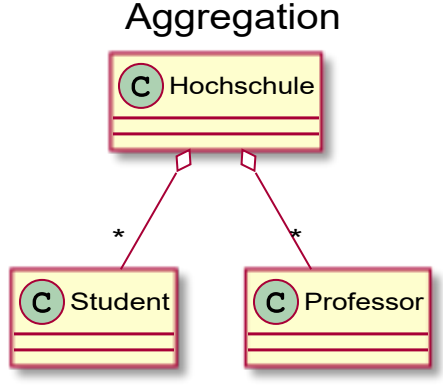
Komposition

- reale und komplexe Objekte bestehen meist aus kleinen und einfachen Objekte
  - Auto besteht aus Reifen, Motor, Sitzen ...
  - PC besteht aus CPU, Motherboard, RAM, ...
- im objektorientierten Paradigma nennt man diese Beziehung: **Komposition**
- bildet eine *besteht aus* oder *hat ein* Beziehung ab



Aggregation

- Aggregation ist eine spezielle Form der Komposition
- bildet auch eine *hat ein* Beziehung ab
- 'Besitz'-Klasse hat jedoch keine Besitzansprüche
  - Referenzierten Klassen leben weiter und werden nicht zerstört wenn die Klasse zerstört wird
  - Referenzierten Klassen werden auch nicht automatisch erstellt wenn die referenzierende Klasse erstellt wird



Design Pattern: Oberserver-Pattern

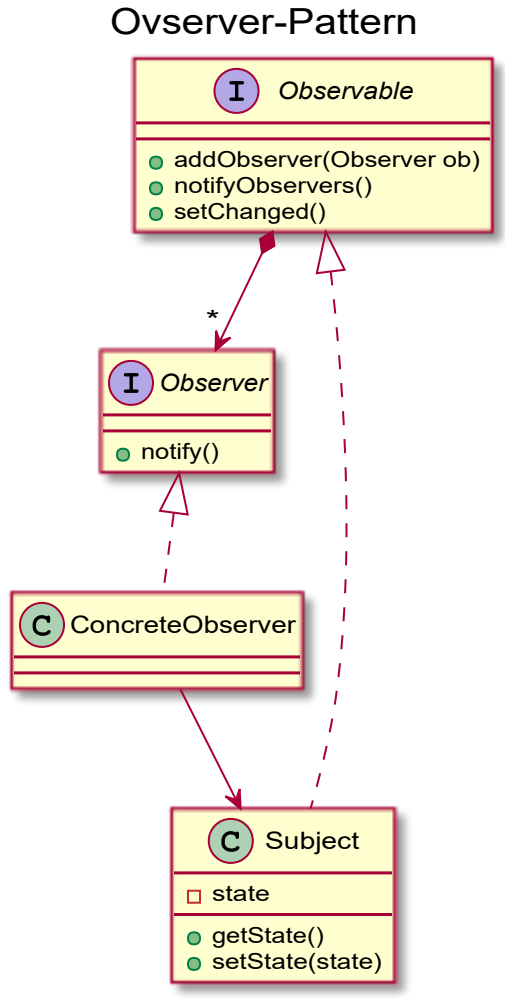
- Das Observer-Pattern ist eines der am meisten genutzten und bekanntesten Patterns
- In diesem Muster gibt es zwei Akteure: Ein Subjekt, welches beobachtet wird und ein oder mehrere Beobachter, die über Änderungen des Subjektes informiert werden wollen
- Die Idee des Observer-Patterns ist es, dem zu beobachtenden Subjekt die Aufgabe aufzutragen, die Beobachter bei einer Änderung über die Änderung zu informieren
- Die Beobachter müssen nicht mehr in regelmäßigen Abständen beim Subjekt anfragen, sondern können sich darauf verlassen, dass sie eine Nachricht über eine Änderung erhalten

Observable and Observer

```
public interface Observable {
    void addObserver(Observer beobachter);
    void removeObserver(Observer beobachter);
    void notifyObservers();
}
```

```
public interface Observer {
    void update();
}
```

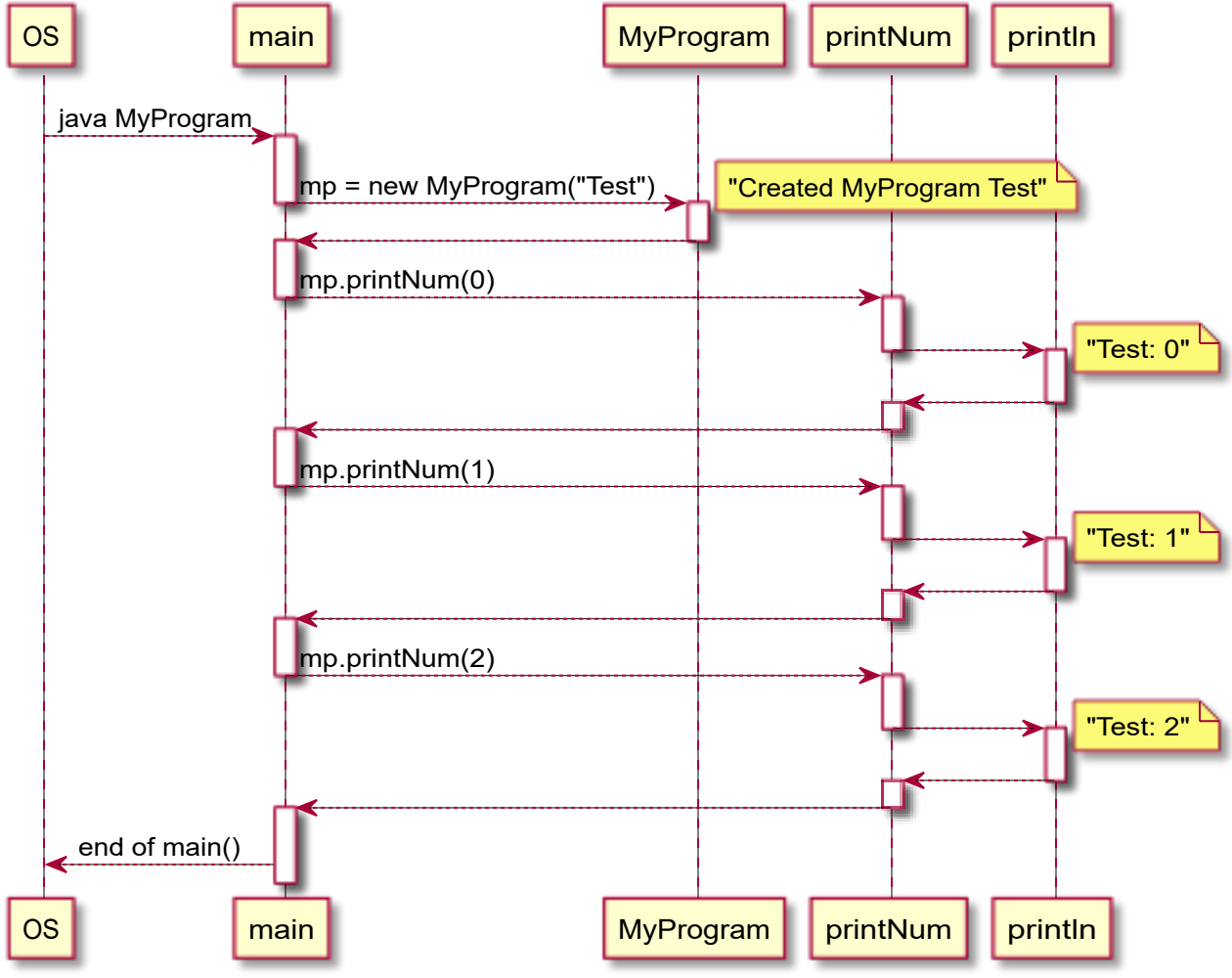
Design Pattern: Oberserver-Pattern (II)



Prozesse

```
class MyProgram {
        String name;
        MyProgram(String name) {
                this.name = name;
                System.out.println("Created MyProgram: " + name
        }
        void printNum(int n) {
                System.out.println(name + ": " + n);
        }
        public static void main(String[] args) {
                MyProgram mp = new MyProgram("Test");
                for (int i = 0; i < 3; i++)
                        mp.printNum(i);
        }
}
```

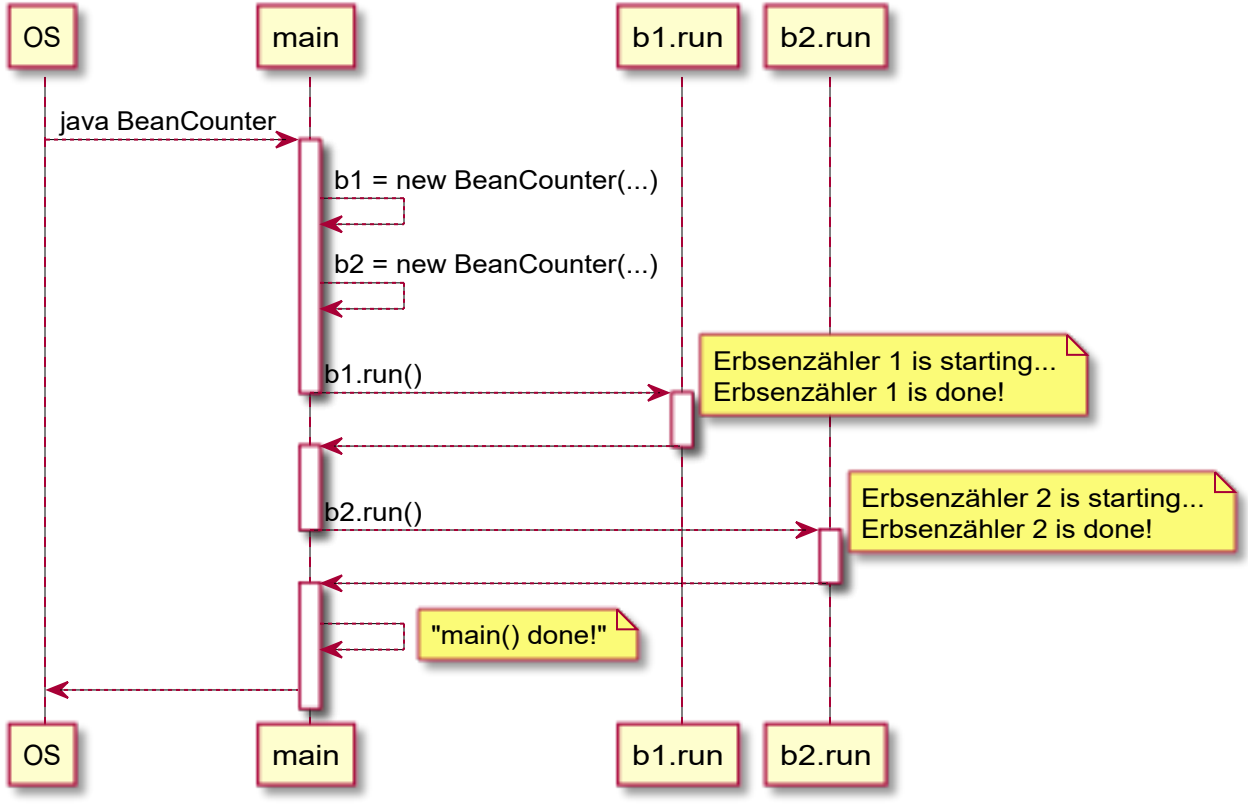
Prozesse


OS
main
MyProgram
printNum
println
java MyProgram
mp = new MyProgram("Test")
"Created MyProgram Test"
mp.printNum(0)
"Test: 0"
mp.printNum(1)
"Test: 1"
mp.printNum(2)
"Test: 2"
end of main()
OS
main
MyProgram
printNum
println

Erbsenzähler

```
class BeanCounter implements Runnable {
        private final String name;
        private final double[] data;
        BeanCounter(String name, int n) {
                this.name = name; this.data = new double [n];
        }
        public void run() {
                System.out.println(name + " is starting...");
                Arrays.sort(data);
                System.out.println(name + " is done!");
        }
        public static void main(String... args) {
                BeanCounter b1 = new BeanCounter("Erbsenzähler
                BeanCounter b2 = new BeanCounter("Erbsenzähler
                b1.run();
                b2.run();
                System.out.println("main() done!");
        }
}
```

Erbsenzähler

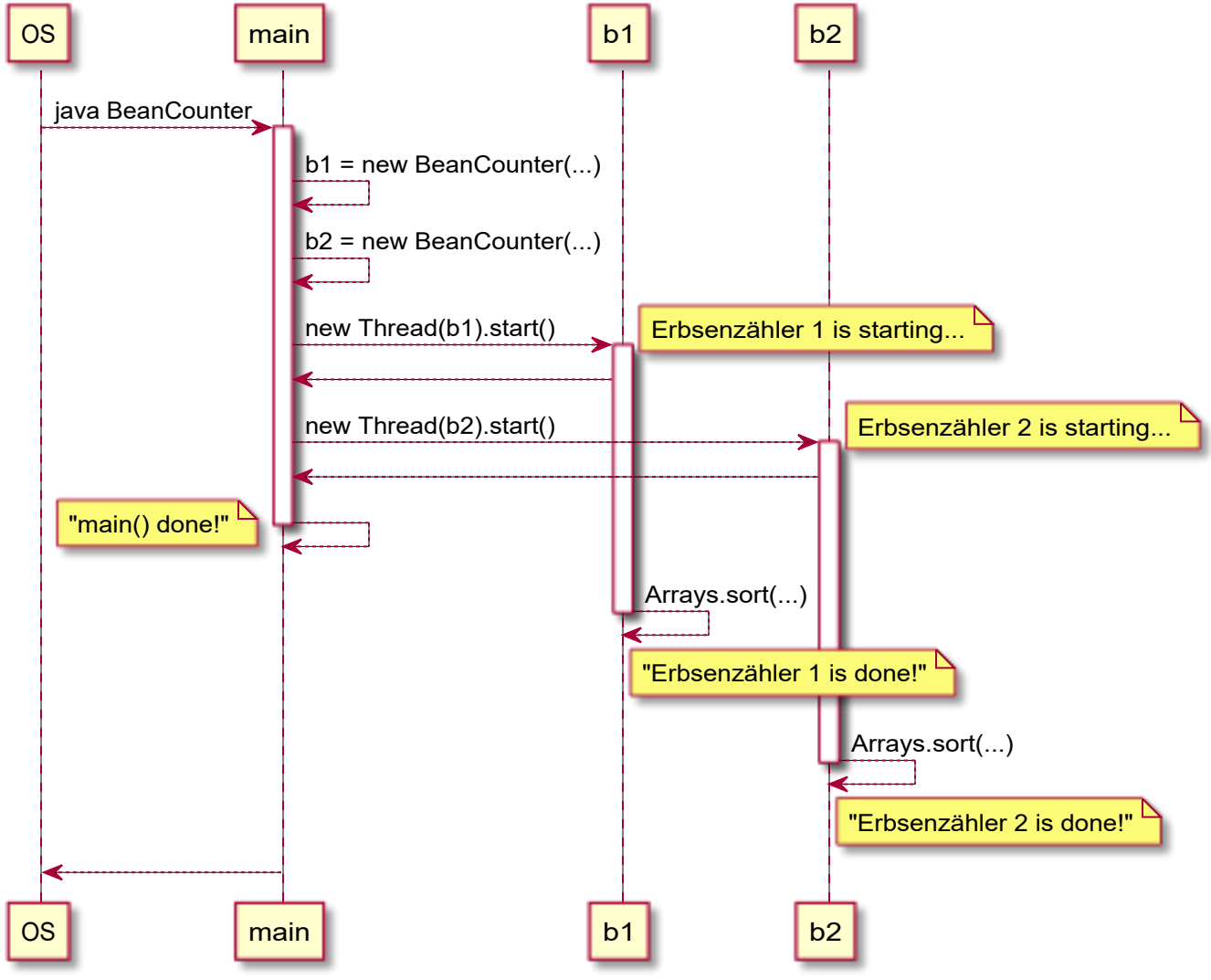


Fleissige Erbsenzähler mit Threads

```
public static void main(String[] args) {
        BeanCounter b1 = new BeanCounter("Erbsenzähler 1", 1000
        BeanCounter b2 = new BeanCounter("Erbsenzähler 2", 1000
        new Thread(b1).start();
        new Thread(b2).start();
        System.out.println("main() done!");
}
```

Fleissige Erbsenzähler mit Threads

.w80[



]

Threads: Beispiele

Nebenläufige Programmierung mit Threads ist in allen modernen Applikationen zu sehen:

- Browser: gleichzeitiges Laden und Rendern von Ressourcen auf einer Webseite
- Gleichzeitiges Rendering mehrerer Animationen
- Behandlung von Benutzerinteraktionen wie Klicks oder Wischen
- Sortieren mit Teile-und-Herrsche-Verfahren
- Gleichzeitige Datenbank-, Netzwerk- und Dateioperationen
- Steuerbarkeit von langlaufenden Prozessen

Geteilte Ressourcen

```
class Counter {
        private int c = 0;
        int getCount() { return c; }
        void increment() {
                c = c + 1;
        }
}
class TeamBeanCounter implements Runnable {
        Counter c;
        TeamBeanCounter(Counter c) {
                this.c = c;
        }
        public void run() {
                for (int i = 0; i < 100000; i++)
                        c.increment();
                System.out.println("Total beans: " + c.getCount
        }
}
```

Geteilte Ressourcen

```
public static void main(String[] args) {
        Counter c = new Counter();
        new Thread(new TeamBeanCounter(c)).start();
        new Thread(new TeamBeanCounter(c)).start();
        new Thread(new TeamBeanCounter(c)).start();
        new Thread(new TeamBeanCounter(c)).start();
}
```

```
Total beans: 362537
```

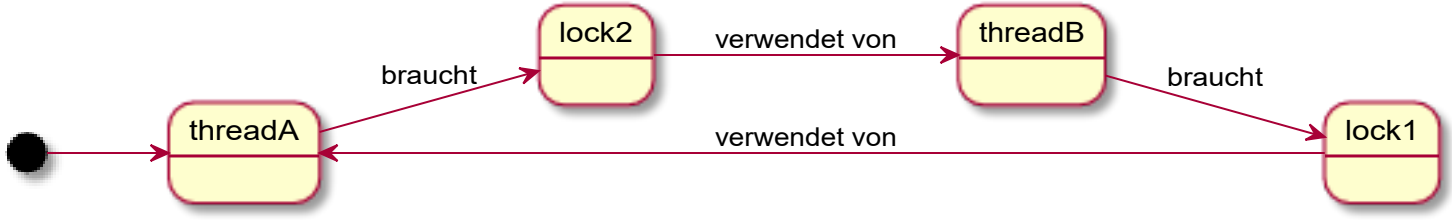
Geteilte Ressourcen: Inkonsistenz!

.pure-table.pure-table-bordered.pure-table-striped[

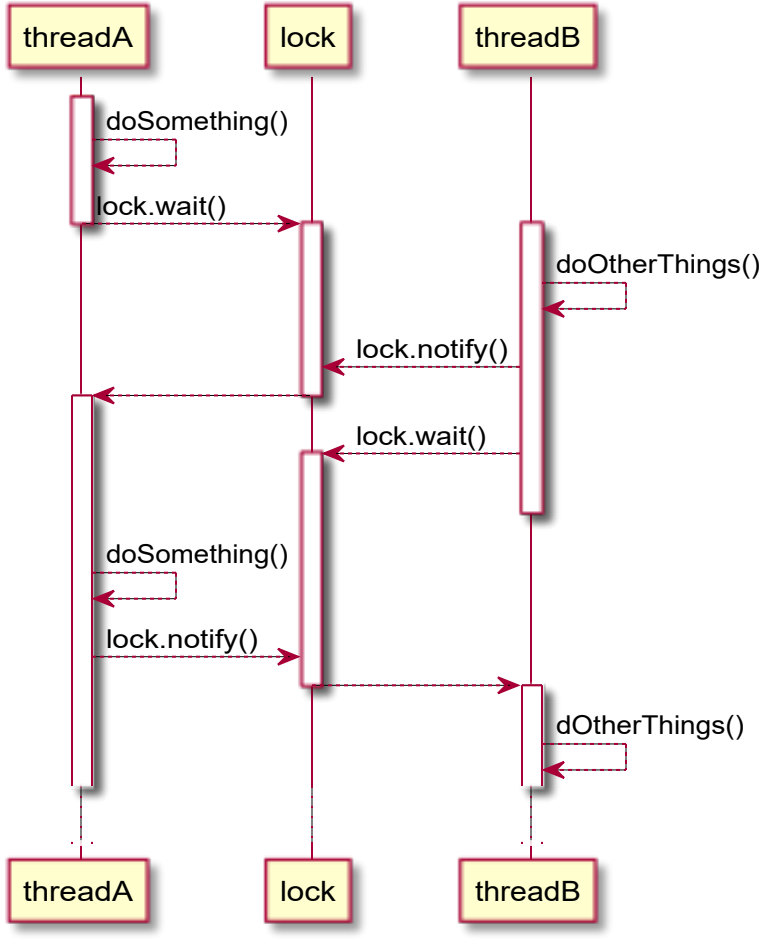
	Thread 1	Thread 2	*result*
1	tmp1 = c		tmp1 = 0
2		tmp2 = c	tmp2 = 0
3	++tmp1		tmp1 = 1
4		++tmp2	tmp2 = 1
5	c = tmp1		c = 1
6		c = tmp2	**c = 1 !**

]

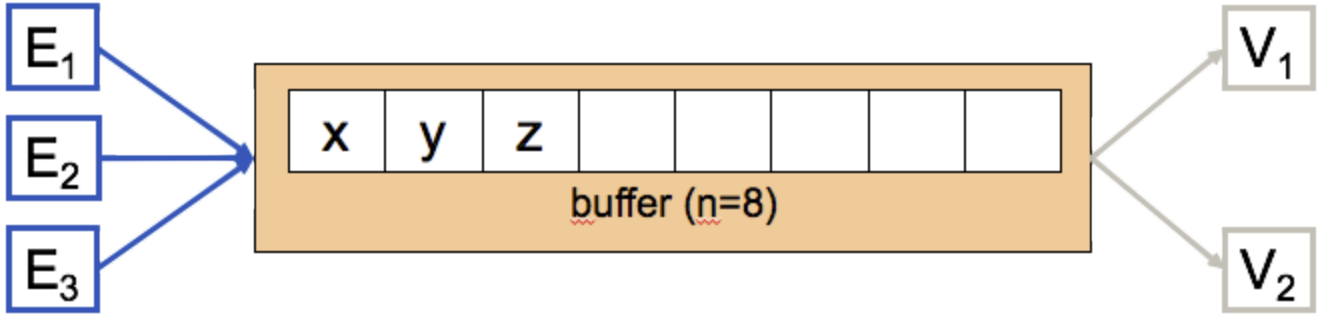
Deadlock


lock2
verwendet von
threadB
braucht
braucht
threadA
verwendet von
lock1

`wait` und `notify`



Erzeuger-Verbraucher-Problem



Threads: Lebenszyklus