Parallele Programmierung

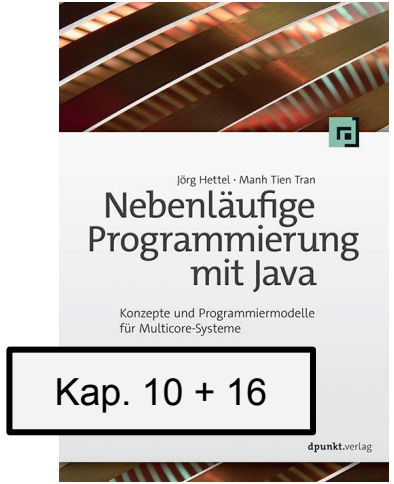
10. Exchanger und BlockingQueue, Anwendung: Logging

Ankündigungen

- Go nicht prüfungsrelevant

Überblick

- `Exchanger<T>`
- `BlockingQueue<T>`
  - Erzeuger/Verbraucher
  - Filter-Verkettung
- Channel-Konzept in Go
- Logging



Kap. 10 + 16

Do not communicate
by sharing memory;

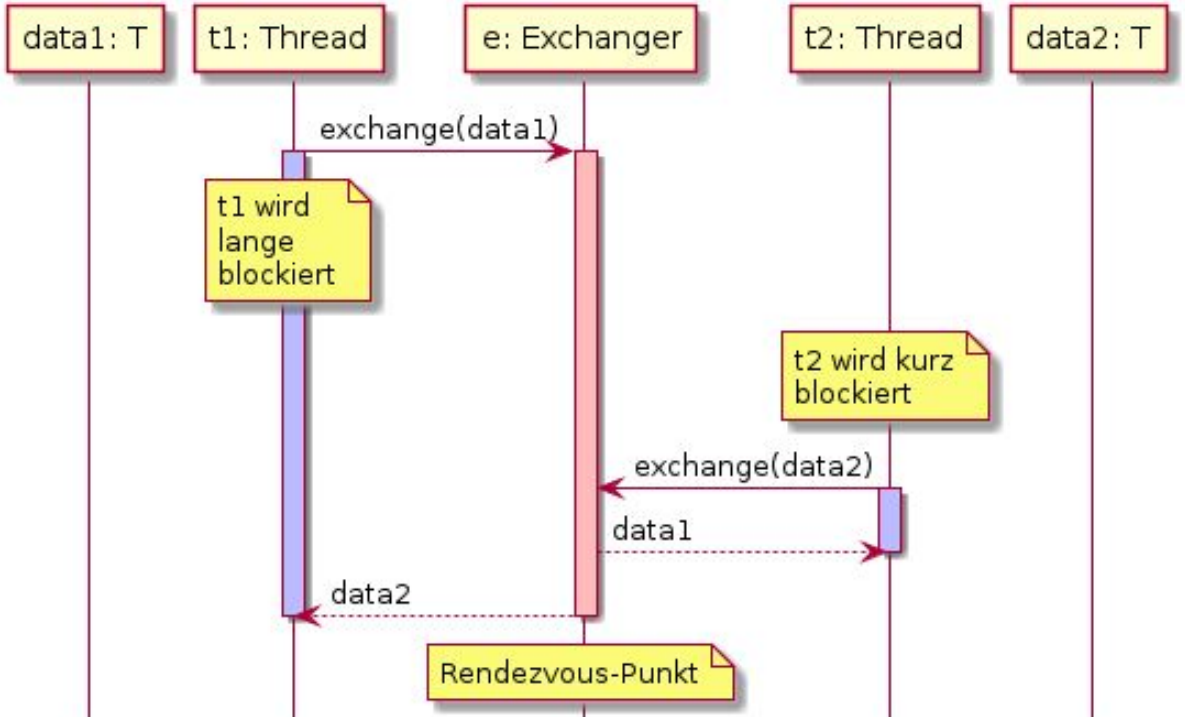
instead,
share memory
by communicating

Exchanger

Exchanger: Synchroner Austausch

- synchroner, typisierter Austauschkanal zwischen zwei Teilnehmern
- zeitgleicher Austausch von Objekten zwischen zwei Threads
- erster am Rendezvous-Punkt (hier `t1`) wird angehalten, bis Partner geliefert hat

```
T data1, data2, x; /* … */
var e = new Exchanger<E>();
x = e.exchange(data1);
```

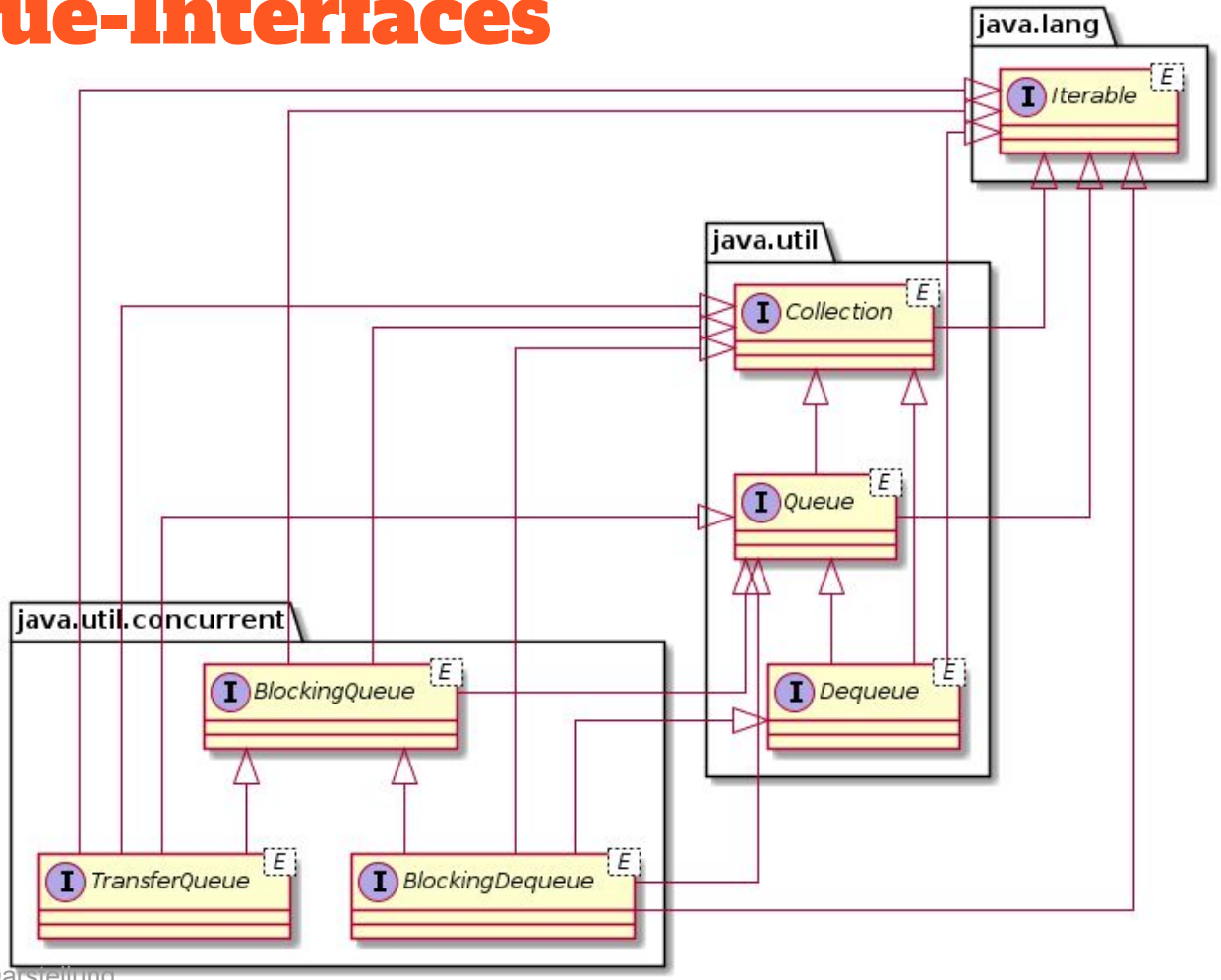


Bildquelle: eigene Darstellung

BlockingQueue

Queue-Interfaces

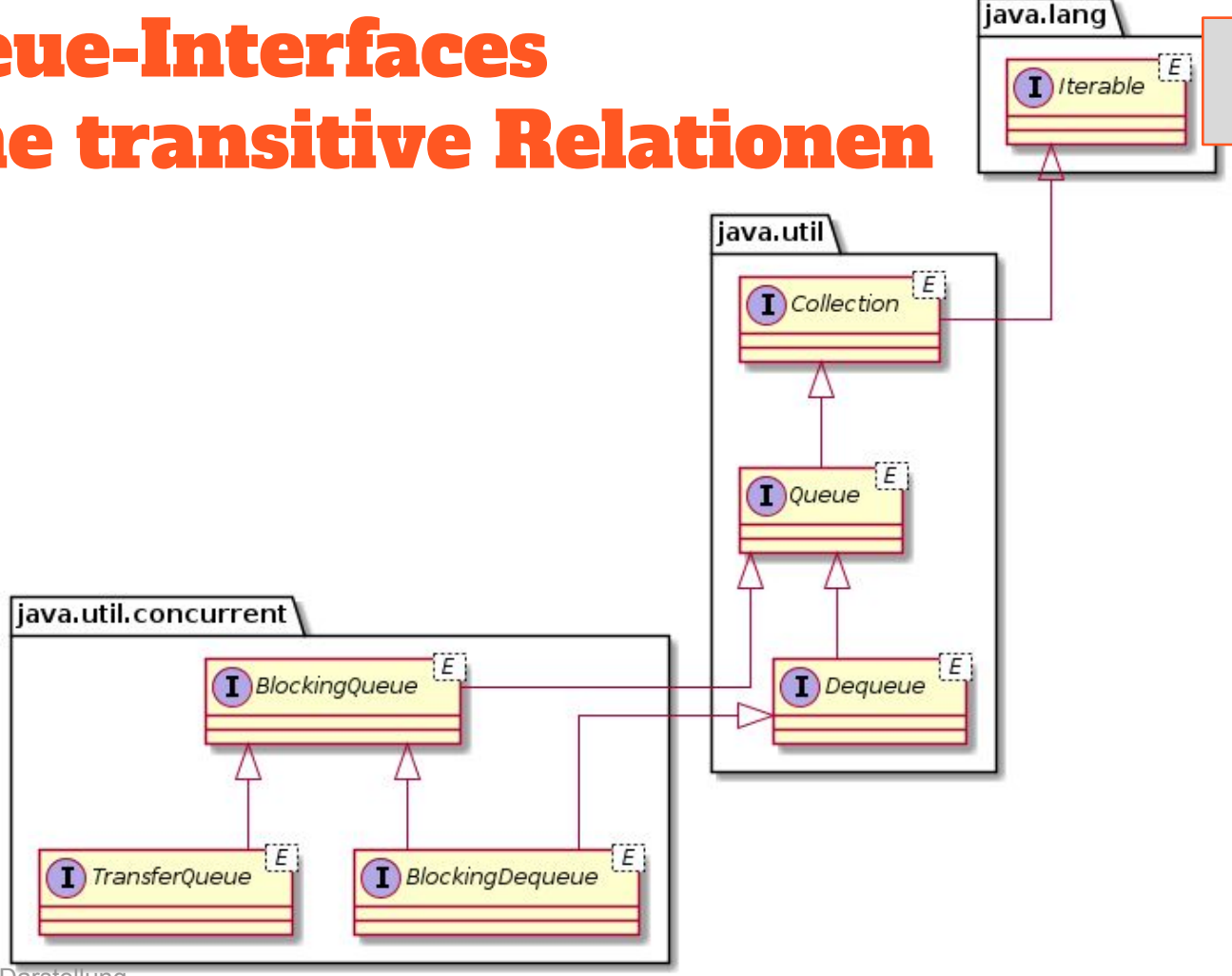
Folie mit Anmerkungen



Bildquelle: eigene Darstellung

Queue-Interfaces ohne transitive Relationen

Folie mit Anmerkungen



Bildquelle: eigene Darstellung

Folie mit Anmerkungen

Methoden von BlockingQueue

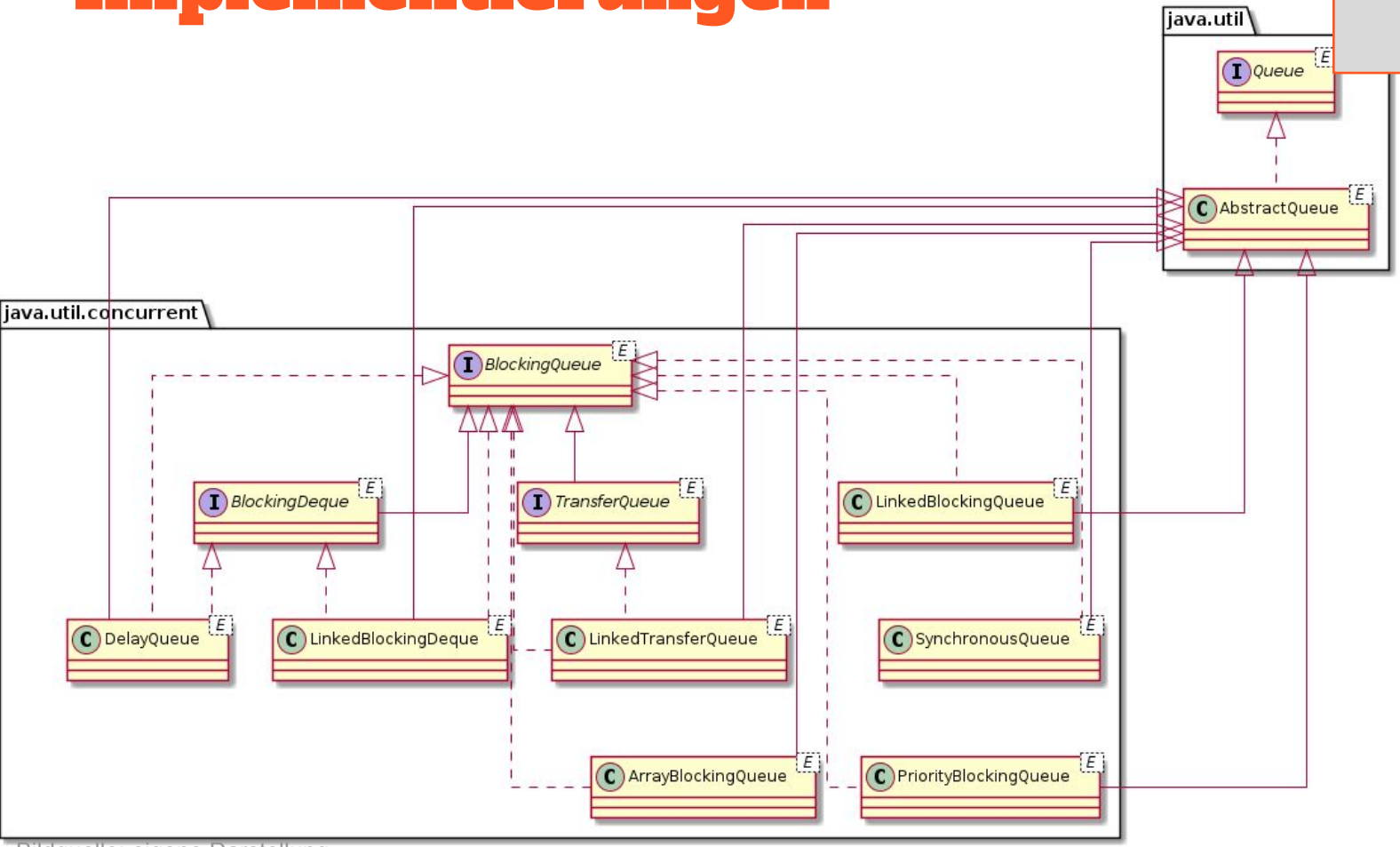
	Mit Exception	Nicht blockierend	Blockierend	Timeout
Einfügen	`add(e)`	`offer(e)`	`put(e)`	`offer(e, time, unit)`
Auslesen	`remove()`	`poll()`	`take()`	`poll(time, unit)`
Überprüfen	`element()`	`peek()`	nicht def.	nicht def.

`InterruptedException`

Bildquelle: Hettel & Tran 2016

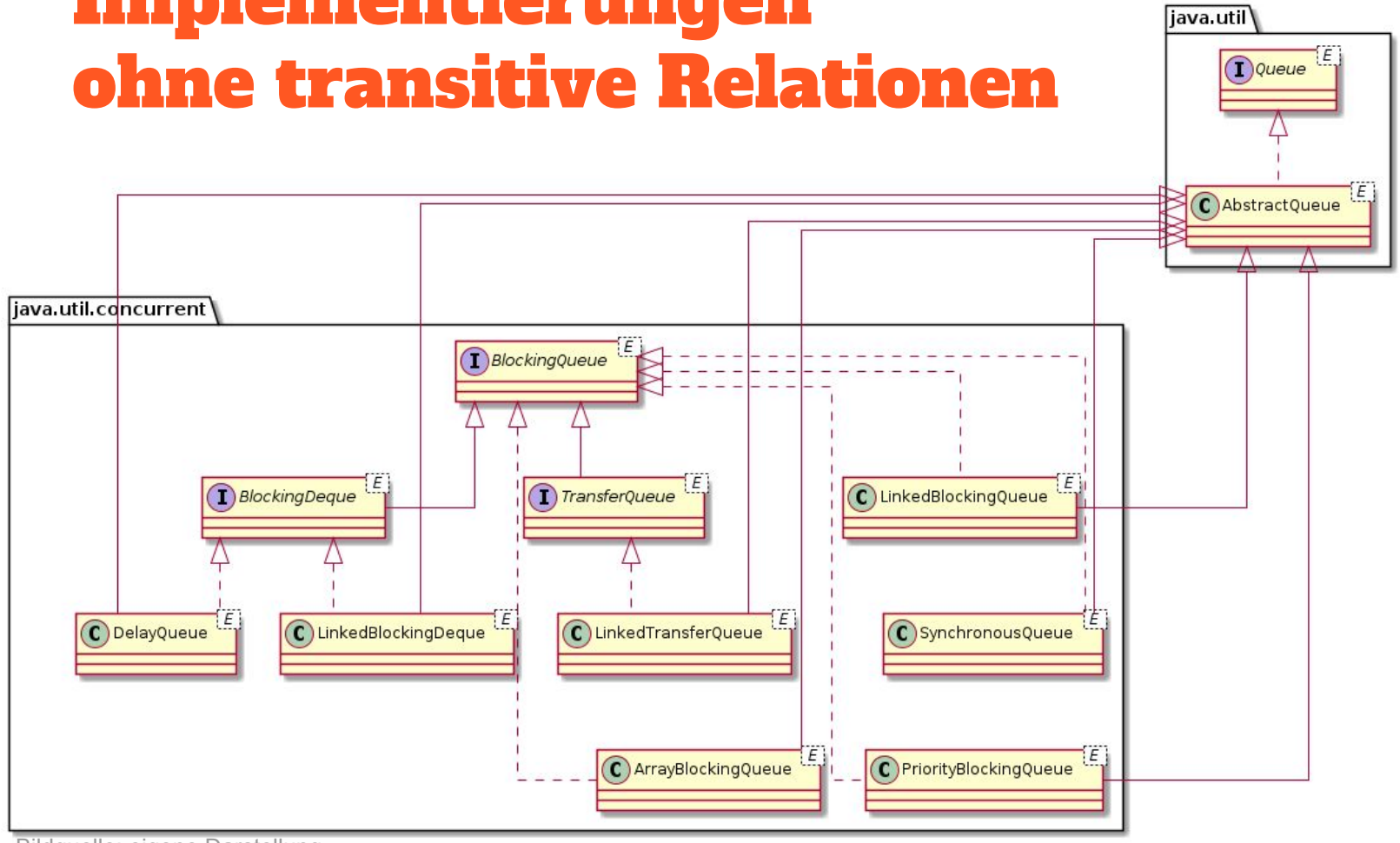
Implementierungen

Folie mit Anmerkungen



Bildquelle: eigene Darstellung

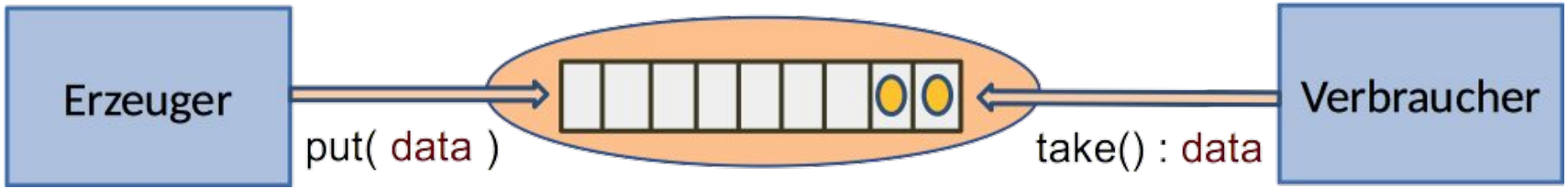
Implementierungen ohne transitive Relationen



Bildquelle: eigene Darstellung

Entkopplung bei Erzeuger/Verbraucher

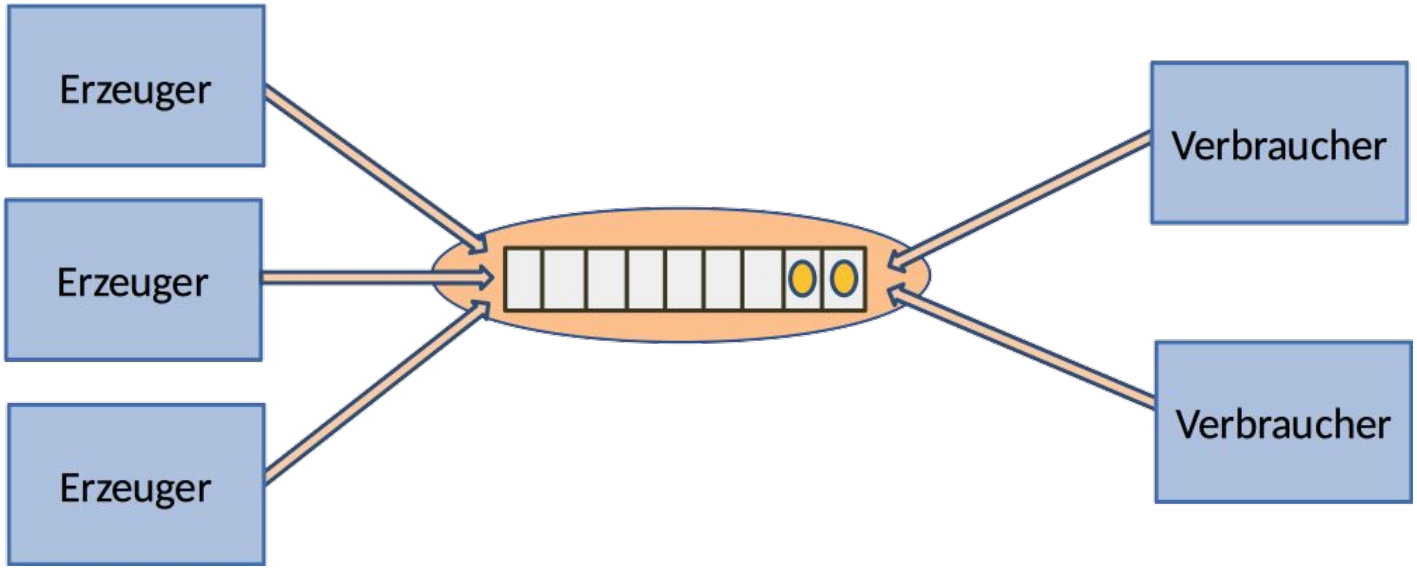
- komplette Entkopplung bei asynchronen Methoden (`offer()`/`poll()`)
- Synchronisierung von Erzeuger/Verbraucher mit blockierenden Methoden (`put()`/`take()`)



Bildquelle: Hettel & Tran 2016

mehrere Erzeuger/Verbraucher

im Gegensatz zum Exchanger können beliebig viele Erzeuger und Verbraucher auf die Queue zugreifen



Bildquelle: Hettel & Tran 2016

Benutzungsmuster: Filterverkettung

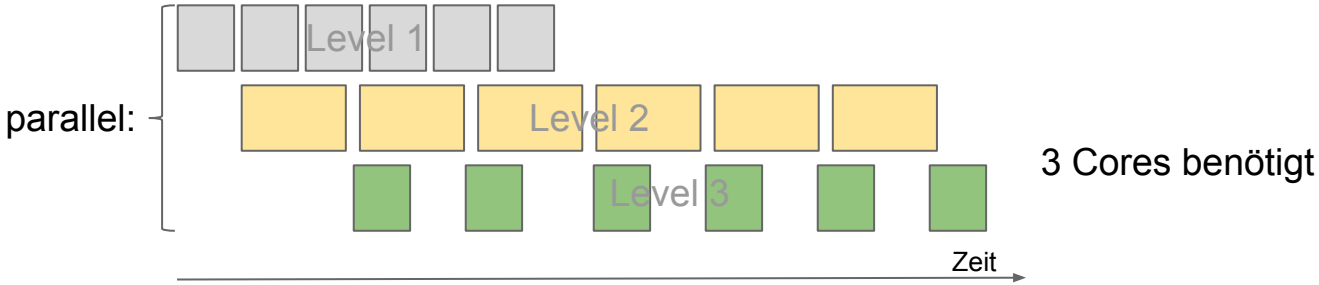
- gerne verwendetes Muster auf der Basis Erzeuger/Verbraucher

1 Core benötigt

Bildquelle oberer Teil: Hettel & Tran 2016, Bildquelle unterer Teil: eigene Darstellung

Benutzungsmuster: Filterverkettung

- gerne verwendetes Muster auf der Basis Erzeuger/Verbraucher
  - alle Verarbeitungsstufen arbeiten nebenläufig

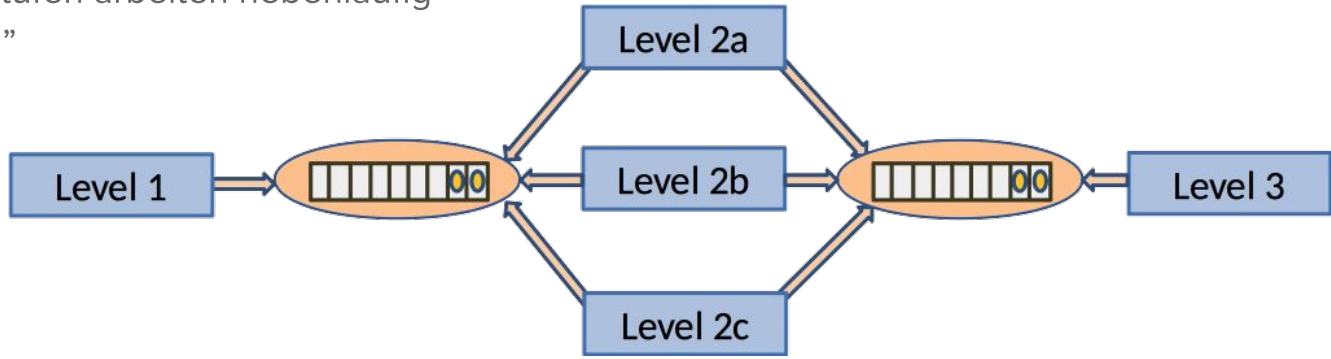


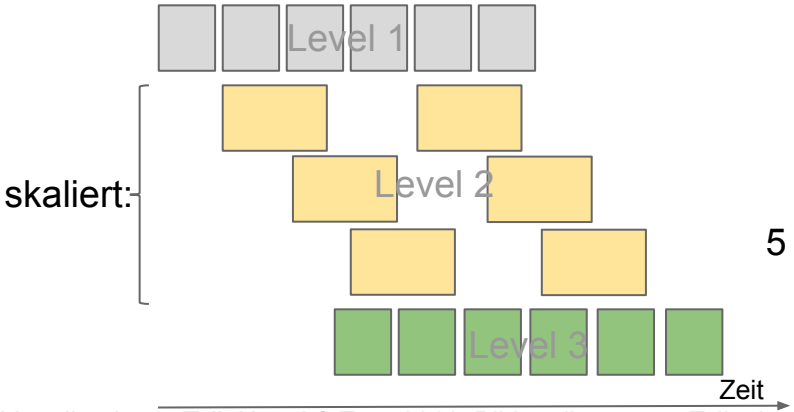
3 Cores benötigt

Bildquelle oberer Teil: Hettel & Tran 2016, Bildquelle unterer Teil: eigene Darstellung

Benutzungsmuster: Filterverkettung

- gerne verwendetes Muster auf der Basis Erzeuger/Verbraucher
  - alle Verarbeitungsstufen arbeiten nebenläufig
  - “vertikal Skalierung”



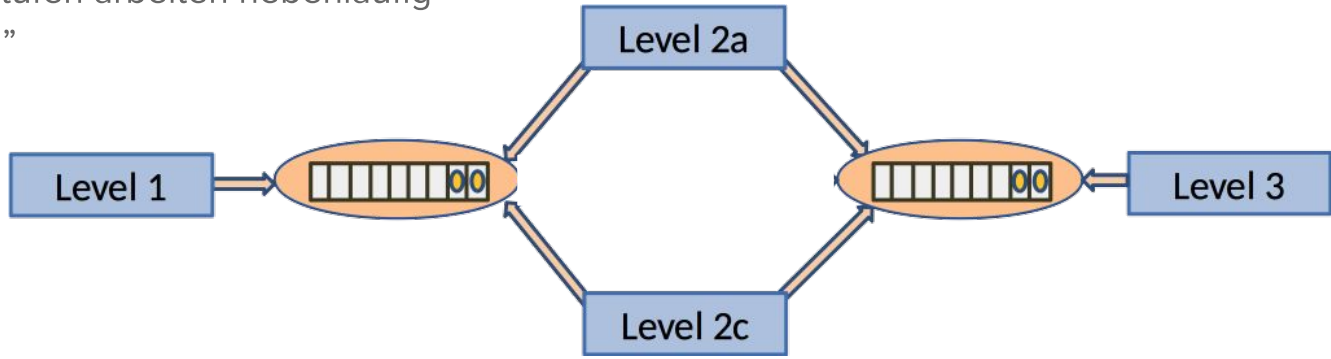


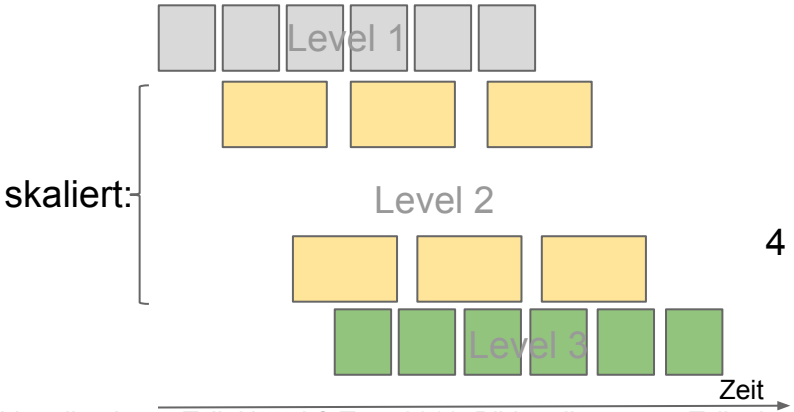
5 Cores benötigt

Bildquelle oberer Teil: Hettel & Tran 2016, Bildquelle unterer Teil: eigene Darstellung

Benutzungsmuster: Filterverkettung

- gerne verwendetes Muster auf der Basis Erzeuger/Verbraucher
  - alle Verarbeitungsstufen arbeiten nebenläufig
  - “vertikal Skalierung”





4 Cores reichen hier

Bildquelle oberer Teil: Hettel & Tran 2016, Bildquelle unterer Teil: eigene Darstellung

nicht prüfungsrelevant!

Kanäle in Go

Communicating Sequential Processes

- Kanal
  - atomare Operationen zum Senden und zum Empfangen
  - Channel-Deklaration: Typbindung
- Empfang von Nachrichten
  - blockiert solange keine Nachricht im Kanal verfügbar ist
- Senden von Nachrichten
  - synchroner Botschaftenaustausch (“Rendezvous”)
    - Kanal puffert Nachrichten nicht, Sender wird solange blockiert, bis Nachricht abgeholt wurde
  - asynchroner Botschaftenaustausch
    - Kanal hat Queue mit endlicher Kapazität > 0
    - wenn Kapazität noch nicht erreicht ist, wird der Sender nicht blockiert
    - wenn die Kapazität erreicht ist, wird der Sender blockiert, bis die Queue wieder Platz hat und die Nachricht dort eingereiht wurde

Communicating Sequential Processes

```
package main
var
    quit chan bool
func f() {
    B
    quit <- true
}
func main() {
    quit = make (chan bool)
    A
    go f() // fork B
    C
    <-quit // wait B
    D
}
```

Deklaration des Kanals `quit` für Elemente vom Typ `bool`

Initialisierung des Kanals `quit` für Elemente vom Typ `bool` mit `make`

hier synchrone Kommunikation (`kapa == 0`)

`make (chan T, kapa)` erzeugt Kanal für Typ `T` mit Kapazität `kapa` (`int`)

Go-Syntax

```
c <- a
```
Senden von Nachricht a auf Kanal c

```
a <- c
```
Nachricht von c empfangen

```
go fun   (...)
```
nebenläufiger Funktionsaufruf von `fun (...)`

blockiert nicht

```
select {
  case x <- c1:
     ...x...
  case <-quit:
     ...
  default:
     ...
}
```
selektives Warten auf mehrere Channels,

normaleweise in **`for`**-Schleife

`default`: falls keine Nachricht an den anderen Kanälen anliegt

Go installieren

Download: https://golang.org/dl/

```
~/go/src/projekt/main.go
cd ~/go/src/projekt/
go build
./main
```

Rendezvous: Addition mit 2 Go-Routinen

```
package main
import "fmt"
func tasker (c, r chan int, d chan bool) {
    c <- 1; c <- 2
    fmt.Println (<-r); d <- true
}
func add (c, r chan int) {
    r <- <-c + <-c
}
func main () {
    c, r:= make(chan int), make(chan int)
    done:= make(chan bool)
    go tasker (c, r, done)
    go add (c, r)
    <-done
}
```

1. 1 auf c schreiben
2. 2 auf c schreiben
3. Ergebnis von r lesen und ausgeben
4. true auf d schreiben

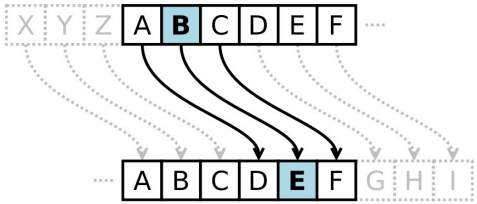
2x von c lesen, addieren, Ergebnis auf r schreiben

erweitertes Rendezvous:
Sender wartet, bis Antwort auf Anfrage kommt

Filterkette durch Channels: Caesar-Crypt

Folie mit Anmerkungen

```
package main
import (
	"bufio"
	"fmt"
	"os"
)
const eol = 13
var input *bufio.Reader
func cap(b byte) byte {
	if b >= 'a' {
		return b - 'a' + 'A'
	}
	return b
}
func dictate(t chan byte) {
	for {
		b, _ := input.ReadByte()
		t <- b
		if b == eol {
			break
		}
	}
}
```




```
func encrypt(t, c chan byte) {
	for {
		b := <-t
		if b == ' ' || b == '.' || b ==

			c <- b
		} else if cap(b) < 'X' {
			c <- b + 3
		} else {
			c <- b - 23
		}
	}
}
func send(c chan byte, d chan bool) {
	b := byte(0)
	for b != eol {
		b = <-c
		fmt.Print(string(b))
	}
	fmt.Println()
	d <- true
}
```

Filterkette durch Channels: Caesar-Crypt

```
func main() {
    input = bufio.NewReader(os.Stdin)
    textchan := make(chan byte)
    cryptchan := make(chan byte)
    done := make(chan bool)
    go dictate(textchan)
    go encrypt(textchan, cryptchan)
    go send(cryptchan, done)
    <-done
}
```

nicht prüfungsrelevant!

Logging

Folie mit Anmerkungen

Logging

- Ausgabe/Speichern von Nachrichten während des (operativen) Betriebs von Systemen.
- Nebenaufgabe: Soll möglichst wenig belasten
- RFC 5424 (Syslog Protocol): Severity-Levels (Emergency – Alert – Critical – Error – Warning – Notice – Informational – Debug)
- In verteilten Systemen: Herausforderung Timing (Uhren auf Knoten können unterschiedlich gehen), Herstellen von Bezügen zwischen Meldungen unterschiedlicher Knoten
- Zentrale Komponente; mehrere Puffer, damit loggende Threads sich nicht wegen der nebenläufigen Nutzung des Pufferspeichers nicht gegenseitig blockieren

SimpleLogger

Interface `io.dama.ffi.logger.Logging`

- zentrale Datenstruktur, die als Puffer dient
- bei mehreren Thread: Es muss synchronized benutzt werden.

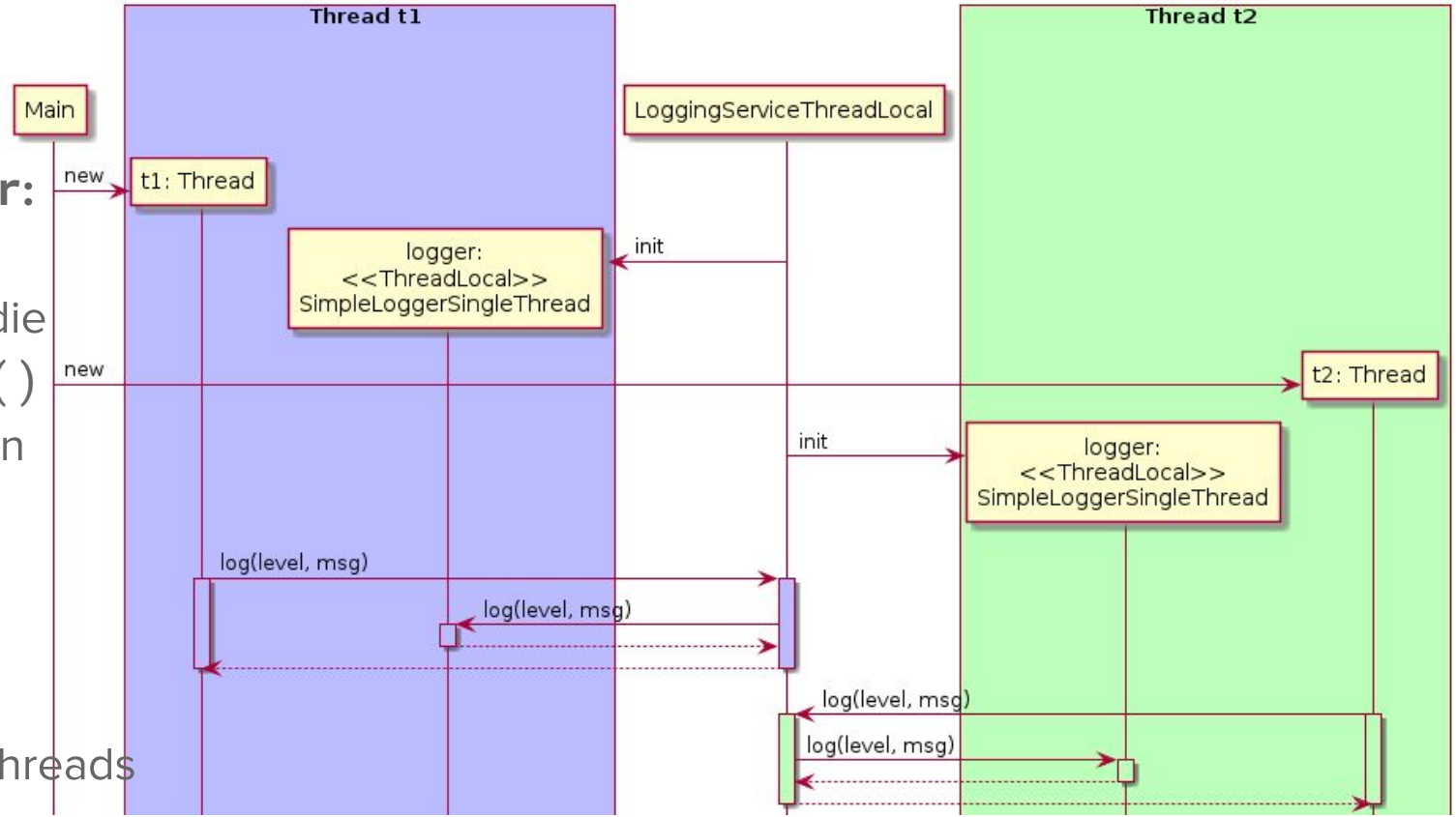
Machen Sie SimpleLoggerSingleThreaded Thread-safe.

Implementierung Thread-lokal

Folie mit Anmerkungen

Vorteil ggü. `SimpleLogger`:
keine zentrale Datenstruktur, die bei jedem `log()` gesperrt werden muss

Nachteil:
Logging in den jeweiligen Anwendungs-Threads



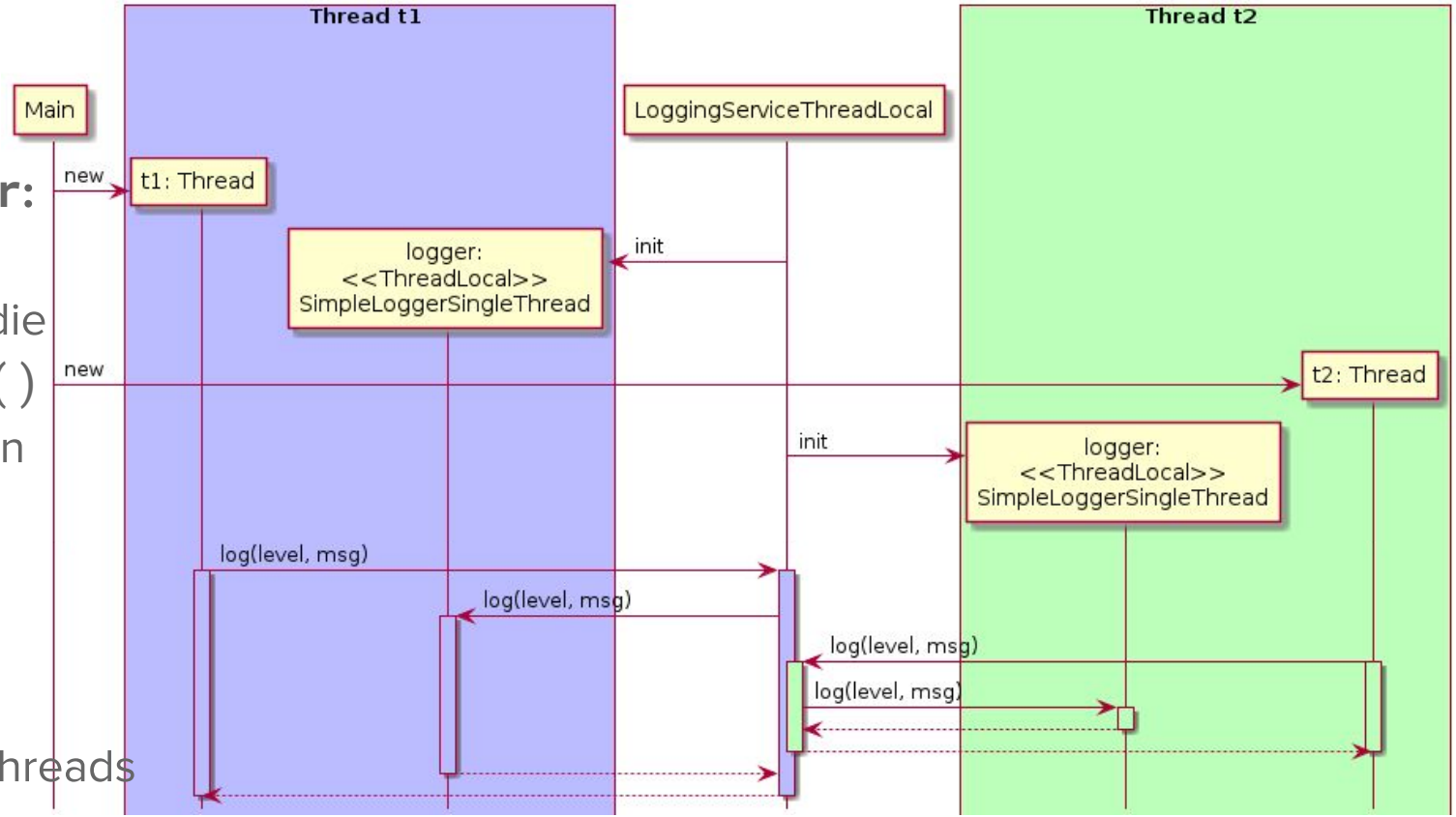
Bildquelle: eigene Darstellung

Implementierung Thread-lokal

Folie mit Anmerkungen

Vorteil ggü. `SimpleLogger`:
keine zentrale Datenstruktur, die bei jedem `log()` gesperrt werden muss

Nachteil:
Logging in den jeweiligen Anwendungs-Threads



Bildquelle: eigene Darstellung