Daily-Soaps der Software:

Kommunikation, Kooperation, und Konkurrenz in Programmen

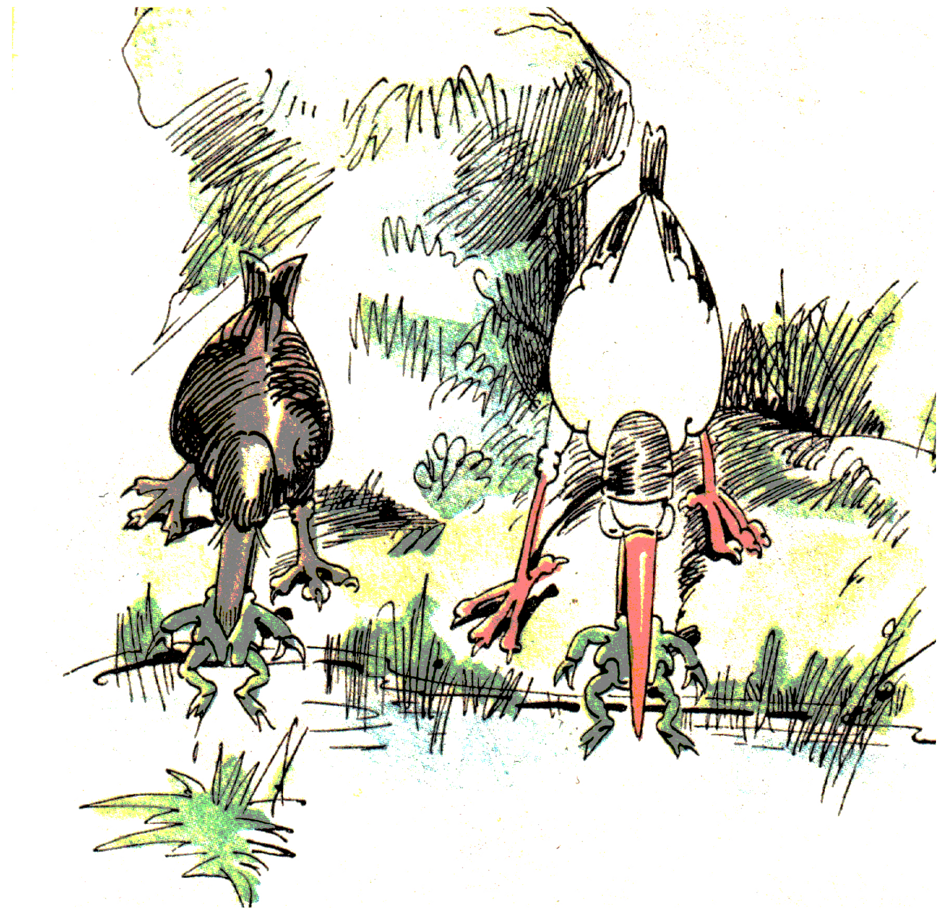
Thomas Letschert

Jeder für sich – ist oft das Beste

Interaktionen machen das Leben interessanter.

Jeder für sich – ist oft das Beste

Oft ist es aber besser, jeder tut das Seine und (fast) alle sind zufrieden.



Kooperation

Manches geht aber nicht alleine.

Kooperation

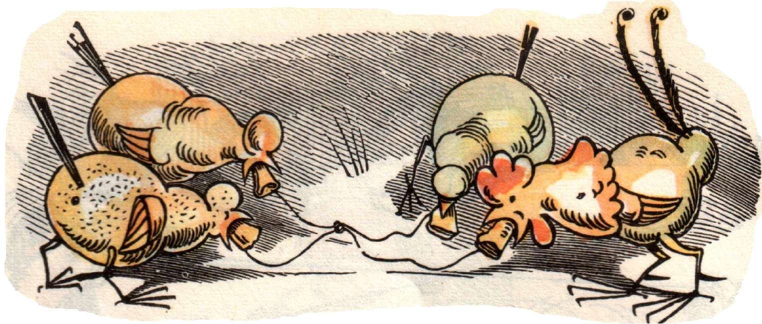
- Bei Kooperationen kann es schnell mal zu Problemen und Enttäuschungen kommen

Kooperationsprobleme

Manchmal entstehen Probleme weil genutzte Ressourcen sich während der Nutzung plötzlich ändern.

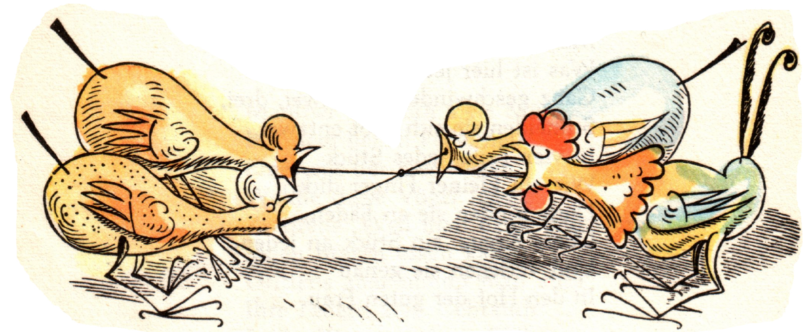
Unveränderliche Partner sind darum zu bevorzugen. – Auch in der Software !

Kooperationsprobleme



Meist entstehen Konflikte durch eine konkurrierende Nutzung von Ressourcen, ...

... die dafür nicht geeignet sind.



Das kann schlimme Folgen haben.

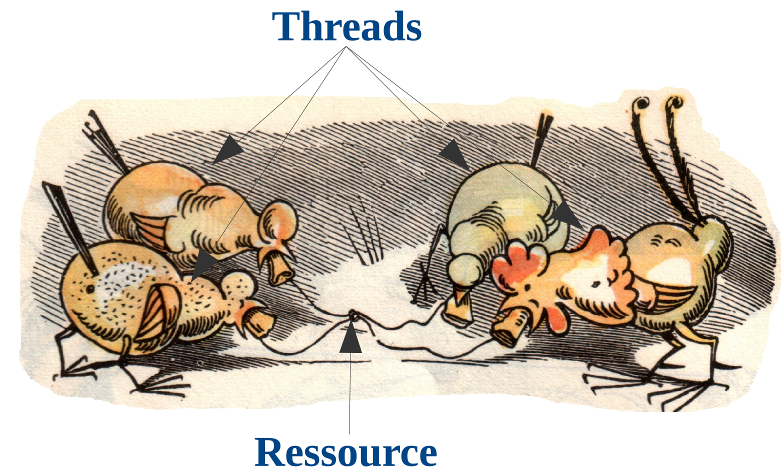
Thread – Threadsave

Thread : Aktive Komponente in einem Programm.

Ressource : Passive Komponente die von Threads genutzt wird.

Threadsave : Kann problemlos von mehreren Akteuren gleichzeitig genutzt werden.

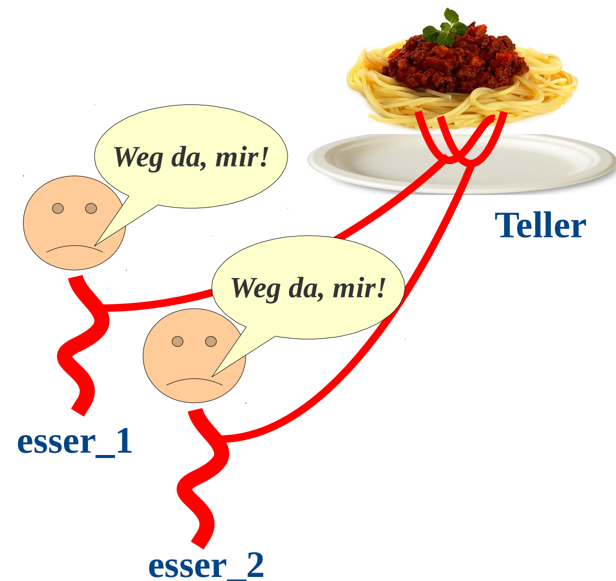
Nicht Threadsave : Gleichzeitige Nutzung führt zu Problemen.



Thread - Konkurrenz

Beispiel: 2 Threads in einem Programm wollen vom gleichen Teller essen

```
case class Speise(art: String)

object Teller {
  var essen: Option[Speise] = Some(Speise("Spaghetti"))
  def leere : Speise = {
    var gabel = essen.get
    essen = None
    gabel
  }
}

object EssenApp extends App {

  val esser_1 = new Thread {
    var mund : Option[Speise] = None
    override def run () {
      mund = Some(Teller.leere)
    }
  }

  val esser_2 = new Thread {
    var mund : Option[Speise] = None
    override def run () {
      mund = Some(Teller.leere)
    }
  }

  esser_1.start
  esser_2.start
}
```



Zugriffssynchronisation

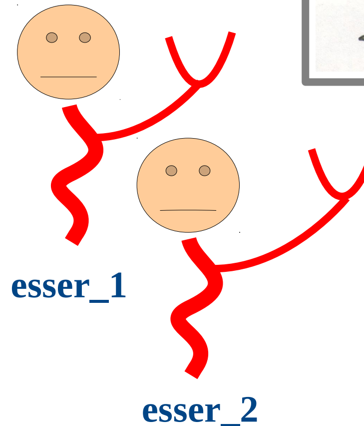
Synchronisation : in geordnete Reihenfolge bringen

Monitor : Passive Komponente die Zugriffe synchronisiert

Mutex : Mechanismus zur Synchronisation der Zugriffe

```
case class Speise(art: String)

object Teller {
  var essen: Option[Speise] = Some(Speise("Spaghetti"))

  def leere : Speise = synchronized {
    var gabel = essen.get
    essen = None
    gabel
  }
}

object EssenApp extends App {

  val esser_1 = new Thread {
    var mund : Option[Speise] = None
    override def run () {
      mund = Some(Teller.leere)
    }
  }
  val esser_2 = new Thread {
    var mund : Option[Speise] = None
    override def run () {
      mund = Some(Teller.leere)
    }
  }
  esser_1.start
  esser_2.start
}
```

Aktion ist bei der Teller-Polizei anzumelden!.

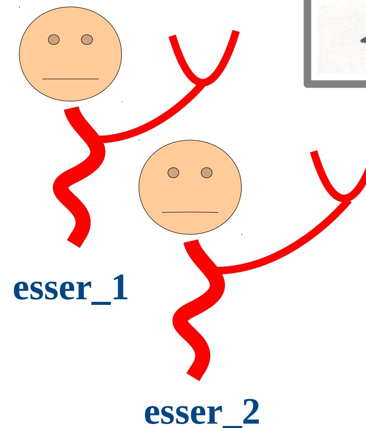


Zugriffssynchronisation

Ein Koch füllt den Teller, auch der Koch wird synchronisiert.

```
case class Speise(art: String)

object Teller {
  var essen: Option[Speise] = None

  def leere : Speise = synchronized {
    var gabel = essen.get
    essen = None
    gabel
  }

  def fuelle (speise: Speise) = synchronized{
    essen = Some(speise)
  }

}

object EssenApp extends App {

  val koch = new Thread {
    override def run () {
       val speise = Speise("Spaghetti")
       Teller.fuelle(speise)
    }
  }

  val esser_1 = new Thread { . . . }
  val esser_2 = new Thread { . . . }

  esser_1.start
  esser_2.start
  koch.start
}
```

Immer schön einer nach dem anderen.

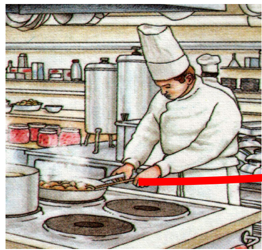


Bedingungssynchronisation

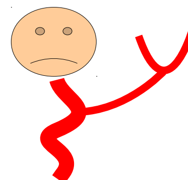
Nicht gleichzeitig **reicht nicht** für eine geordnete Nutzung.

Die *richtige Reihenfolge* ist auch wichtig.

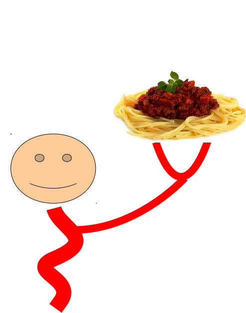
Stopp!
Der Teller ist voll.



Stopp!
Der Teller ist leer.



esser_1



esser_2

Bedingungssynchronisation

Nicht gleichzeitig **reicht nicht für eine geordnete Nutzung.**

Die *richtige Reihenfolge* ist auch wichtig.

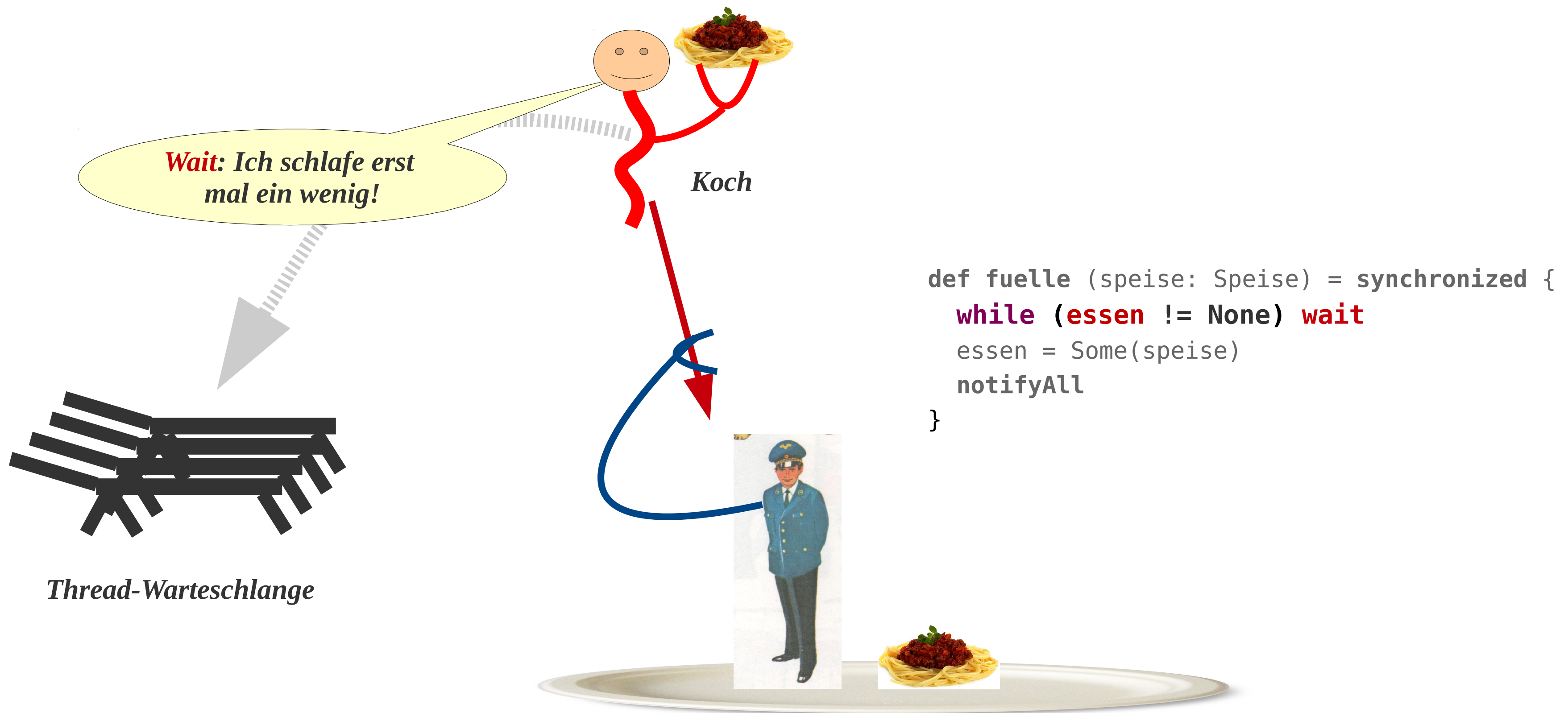
```scala
case class Speise(art: String)

object Teller {

  var essen: Option[Speise] = None

  def leere : Speise = synchronized {
    while (essen == None) wait
    var gabel = essen.get
    essen = None
    notifyAll
    gabel
  }

  def fuelle (speise: Speise) = synchronized {
    while (essen != None) wait
    essen = Some(speise)
    notifyAll
  }

}
```

```scala
object EssenApp extends App {

  val koch = new Thread {
    override def run () {
      while (true) {
            val speise = Speise("Spaghetti")
            Teller.fuelle(speise)
            println("Koch hat Teller gefüllt")
            Thread.sleep(1000)
      }
    }
  }
  val esser_1 = new Thread {
    var mund : Option[Speise] = None
    override def run () {
      while (true) {
          println("Esser 1 ist hungrig")
          mund = Some(Teller.leere)
          println("Esser 1 hat gegessen")
      }
    }
  }
  val esser_2 = new Thread {
    var mund : Option[Speise] = None
    override def run () {
      while (true) {
          println("Esser 2 ist hungrig")
          mund = Some(Teller.leere)
          println("Esser 2 hat gegessen")
      }
    }
  }

  esser_1.start
  esser_2.start
  koch.start
}
```

Bedingungssynchronisation

Die Ressource Teller regelt ihre Benutzung in der richtigen Reihenfolge

- mit **Warteschlange**: Threads die warten müssen bis die Ressource in einem Zustand ist, in dem sie von diesem Benutzer genutzt werden kann

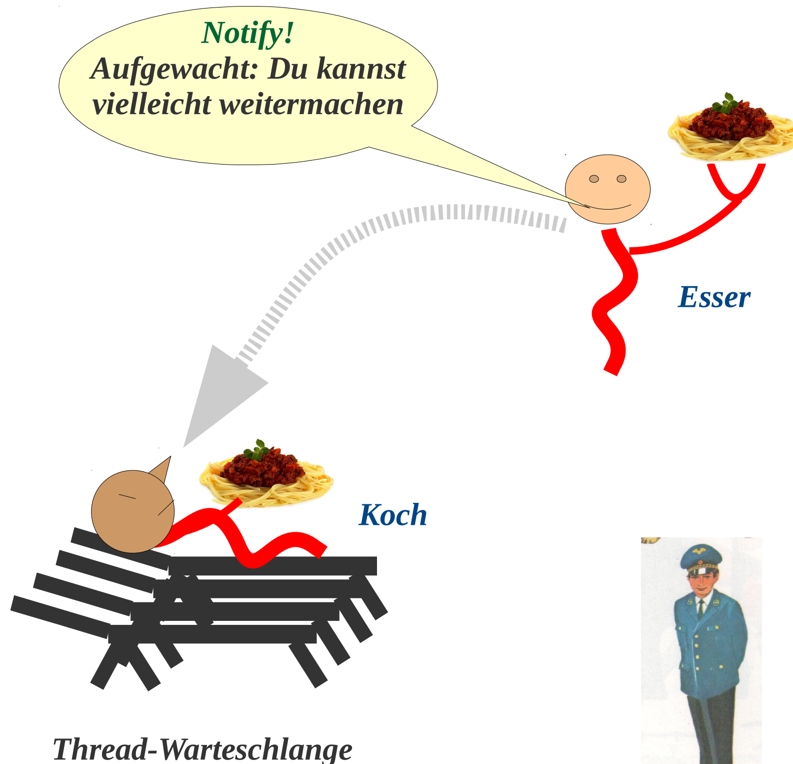



```
def fuelle (speise: Speise) = synchronized {
  while (essen != None) wait
  essen = Some(speise)
  notifyAll
}
```

Bedingungssynchronisation

Die Ressource Teller regelt ihre Benutzung in der richtigen Reihenfolge

- mit **Warteschlange**: Threads die warten müssen, bis die Ressource in einem Zustand ist, in dem sie von diesem Benutzer genutzt werden kann.




```
def leere : Speise = synchronized {
  while (essen == None) wait
  var gabel = essen.get
  essen = None
  notifyAll
  gabel
}
```

`notifyAll`: *Alle wecken. Wenn nur einer geweckt wird, dann könnte es der falsche sein. (Z.B.: ein anderer Esser.)*

Kommunikation statt Synchronisation

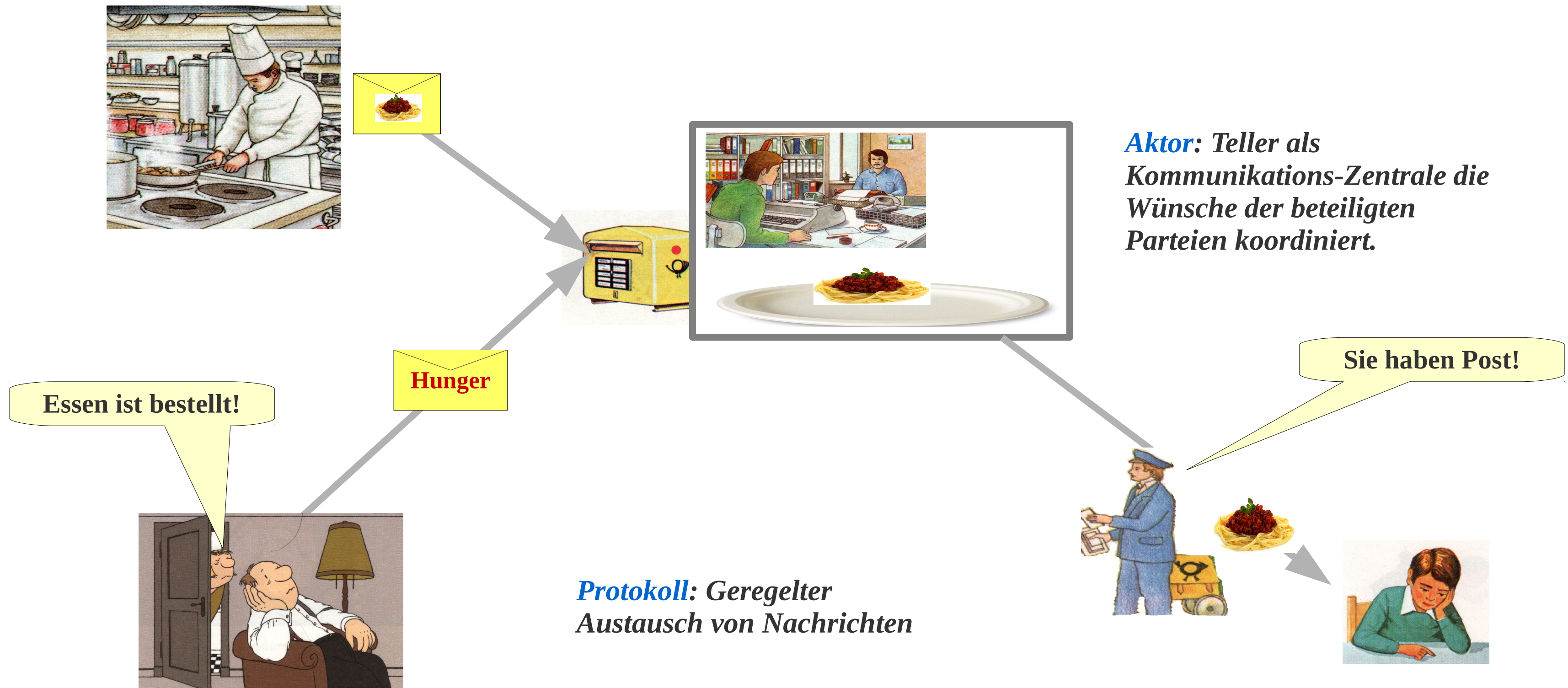
Direkte Begegnungen sind also oft problematisch, vielleicht ist es besser, man begegnet sich nicht direkt ...

… sondern formuliert seine Wünsche als Nachricht, ...

… die dem Partner zugesendet wird.

Kommunikation statt Synchronisation

Konflikte durch direkte Begegnungen bleiben aus, wenn alle nur über Nachrichten Kommunizieren.



Aktor: Teller als Kommunikations-Zentrale die Wünsche der beteiligten Parteien koordiniert.

Protokoll: Geregelter Austausch von Nachrichten

Kommunikation statt Synchronisation

Speisewünsche und Essen via Paket-Post : Teller

```
case object Hunger
case class  Speise(art: String)
case object OK

class Teller extends Actor with Stash {

  var essen: Option[Speise] = None

  def receive = {

    case Hunger =>
      if (essen == None) {
              stash
      } else {
              sender !  Speise(essen.get.art)
              essen = None
              unstashAll
      }

    case Speise(s) if (essen == None) =>
      if (essen != None) {
              stash
      } else {
              essen = Some(Speise(s))
              sender ! OK
              unstashAll
      }
  }

}
```

Die verschiedenen Arten von Nachrichten

Der Teller empfängt Nachrichten ...

***… z.B. von einem Esser die Nachricht* Hunger**
wenn gerade nicht passt wird die Nachricht wieder weg gelegt,
wenn doch etwas da ist, wird die Speise an den Sender geschickt

***… z.B. vom Koch eine* Speise**
wenn gerade nicht passt wird die Nachricht wieder weg gelegt,
***wenn doch etwas da ist, wird die Speise abgelegt und der Sender bekommt ein* OK.**

Kommunikation statt Synchronisation

Speisewünsche und Essen via Paket-Post : Esser

```
object EssenApp extends App {
  val system = ActorSystem("EssenSystem")
  val teller = system.actorOf(
                        Props[Teller].withDispatcher("akka.actor.my-custom-dispatcher"),
                        name = "teller")

  implicit val timeout = Timeout(5 seconds)

  object esser1 extends Thread {
      var mund : Option[Speise] = None
      override def run () {
        while (true) {
              println(s"Esser 1 ist hungrig")
              val futureSpeise = teller ? Hunger
              val speise = Await.result(futureSpeise, timeout.duration)
              println(s"Esser 1 hat $speise gegessen")
        }
      }
  }
  object esser2 extends Thread { … entsprechend … }

  esser1.start()
  esser2.start()
```

*Die Esser senden **Hunger**-Nachrichten, warten auf die Essens-Nachrichten und essen das Essen*

Kommunikation statt Synchronisation

Speisewünsche und Essen via Paket-Post : Koch

```
  object koch extends Thread {

    override def run () {
        while (true) {
              val speise = Speise("Spaghetti")
              val futureOK = teller ? speise
              println("Koch hat Teller gefüllt")
              Await.result(futureOK, timeout.duration)
              Thread.sleep(1000)
        }
    }
  }

  koch.start()
}
```

Der Koch kocht Spaghetti,
sendet Essen-Nachrichten,
wartet auf die OK-Nachricht,
ruht sich kurz aus,
und macht dann weiter.

Zusammenfassung

Kooperationen sind schwierig und riskant

- auch wenn Software-Komponenten kooperieren
- Die Beteiligten müssen synchronisiert werden

Es geht oft um den Streit um eine gemeinsame Ressource

- Lösung: Gegenseitiger Ausschluss

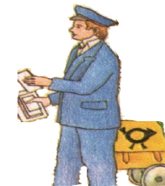
Aber auch Kooperation muss geregelt werden

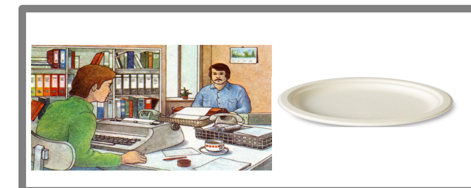
- Aktionen der Partner müssen in die richtige Reihenfolge gebracht werden
- z.B. Teller füllen. dann leer essen

Konzept 1: Monitor / gemeinsame Ressourcen

- Die gemeinsam genutzte Ressource passt darauf auf, dass sie richtig genutzt wird
- Mittel: Akteure mit gerade nicht passenden Absichten schlafen legen
- Wecken, wenn es passt

Konzept 2: Verteiltheit / nur Nachrichten





- Gemeinsames gibt es nicht
- Interaktion nur über Nachrichten
- Nachrichten werden passend versendet und erwartet

Software

Die Beispiele wurden in Scala verfasst

- Scala: Eine moderne Java-basierte Programmiersprache
- mit Schwerpunkt auf Nachrichten-basierter Kooperation
- aber auch allen klassischen Konstrukten zur Synchronisation
- siehe http://www.scala-lang.org/
  oder
  http://de.wikipedia.org/wiki/Scala_%28Programmiersprache%29