Böse Überraschungen mit Nebenläufigkeit vermeiden: Am Beispiel von .NET

Prof. Dr. Luc Bläser

Hochschule für Technik Rapperswil

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

INSTITUT
FÜR
SOFTWARE

Parallel 2014 Konferenz
Karlsruhe, 6 Mai 2014

Nebenläufigkeit wird unvermeidbar

- Vom Design auferlegt
  - GUIs reaktionsfähig machen
  - Externe Bibliotheken
  - Verteilte Systeme
  - Etliche Sprachfeatures
- Performance-Beschleuniger
  - Taktraten-Stagnation
  - Parallelisierung auf Multi-Cores
  - Expliziter Programmieraufwand nötig!
  - „The free lunch is over” – H. Sutter

Vieles kann schieflaufen

- Ganz neue Fehlerarten
  - Race Conditions
  - Deadlocks & Livelocks
  - Starvation
- Nicht-deterministische Fehler
  - Treten sporadisch auf
  - Schwierig zu finden
  - Schwierig zu testen

Therac-25 RT Linac

Patienten durch Verstrahlung gestorben. Grund: Race condition

N. G. Leveson, C. S. Turner, *An Investigation of the Therac-25 Accidents*, IEEE Computer, Volume 26, Issue 7 (Jul 1993), pp. 18-41

Indikatoren für Alarmierung:

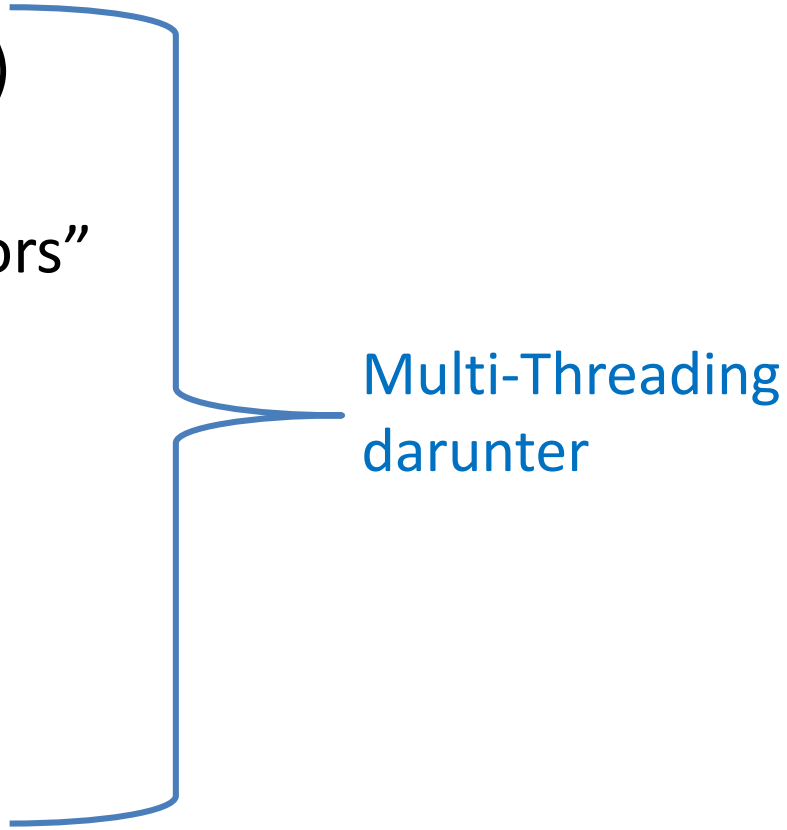
- „Keine Ahnung, wieso der Test fehlschlug. Seit gestern ist er wieder dauernd grün."
- „Die Software hängt in sehr seltenen Fällen. Nichts schlimmes, einfach neustarten."

Vortragsinhalt

- Essentielles zur Nebenläufigkeit
  - Multi-Threading in a Nutshell
  - Concurrency Fehler
- Strukturierung der Nebenläufigkeit
  - Design in Architekturen
  - Szenarien und Muster
- Schlussfolgerungen

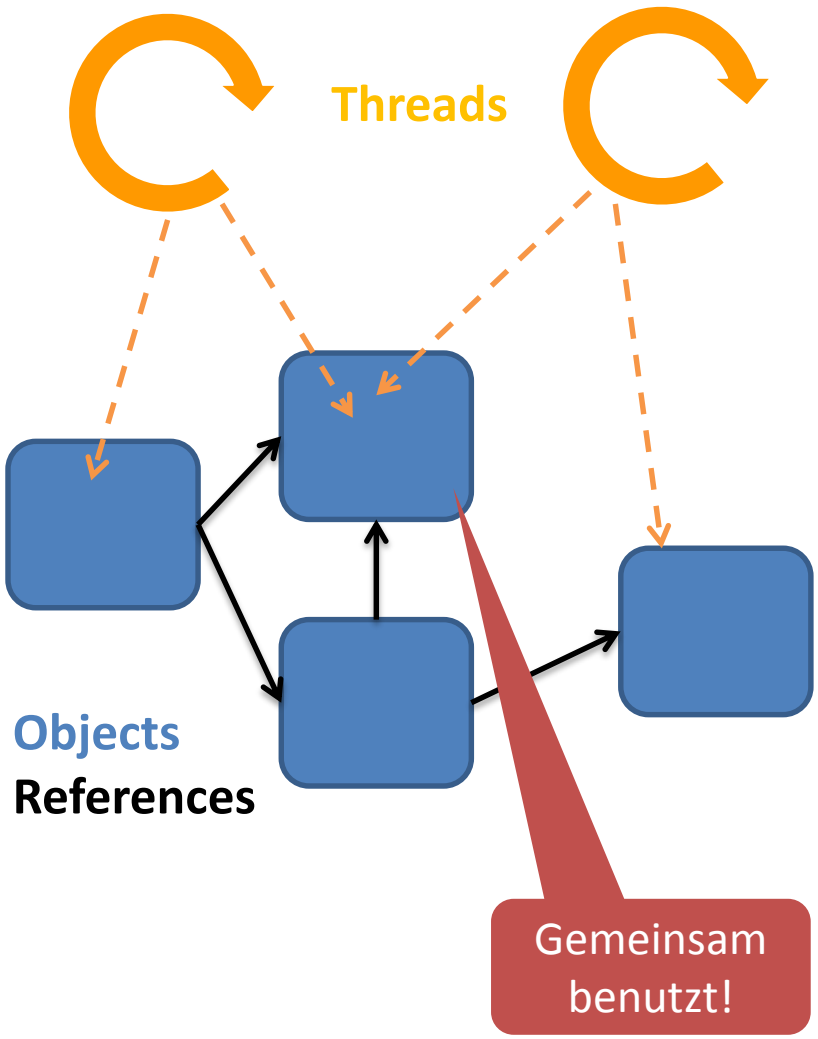
Quellen der Nebenläufigkeit in .NET

- Explizite Threads
- TPL Tasks (Thread Pools)
- Parallel LINQ
- Finalizers, “C# Destructors”
- Web services, sockets
- Background-Worker
- Asynchrone Aufrufe
- async/await
- Timers
- External calls/callbacks
- Zwischen Prozessen (via DB, files, network etc.)



Nebenläufiges Programmiermodell

- Threads operieren auf passiven Objekten
  - Via direkte / indirekte Methodenaufrufe
  - Parallele oder beliebige Verzahnung
  - By default: unkontrolliert
  - Muss explizit synchronisieren
- Es gibt bessere Modelle
  - Actors, (STM), funktional



Prädestinierte Race Condition Stellen

```
int Next() {
  return counter++;
}
```

```
if (!loaded) {
   loaded = true;
   Internalize();
}
```

```
void Swap() {
   t = y; y = x; x = t;
}
```

```
foreach (T item in threadSafeCollection) {
   …
}
```

```
while (safeBuffer.Size > 0) {
   safeBuffer.RemoveFirst();
}
```

```
if (weakReference.IsAlive) {
  weakReference.Value.Op();
}
```

Und so weiter…

Race Condition

- Ungenügend synchronisierter Zugriff auf gemeinsame Ressourcen
  - Fehlerhaftes oder undefiniertes Verhalten
  - Hängt von Timing/Interleaving/Maschine ab
- Low Level: Data Races
  - Nebenläufiger Zugriff ohne Synchronisation
  - Selbe Variable oder Array Element
  - Read-write, write-read, write-write
- High Level: Unsynchronisierte Sequenzen
  - Kritische (atomare) Abschnitte nicht sichergestellt

Synchronisations-Abstraktionen in .NET

- Monitor (aka C# lock mit Wait & Pulse)
- Reader-Writer Lock
- Concurrent Collections
- Primitives: Semaphore, Barrier, Mutex, CountDownEvent, Wait Handles, …
- Thread/Task Joins
- Memory Level: Interlocked, volatile, Barriers/Fences

Races behoben, Deadlocks verursacht

```
class Repository { …
  void CopyTo(Repository target) {
    lock(this) {
      // get
      target.Add(content);
    }
  }

  public void Add(T content) {
    lock(this) {
      // add
    }
  }
}
```

Geschachtelter Lock

Deadlock

T1 sperrt a
T2 sperrt b
T1 will b
T2 wil a

lock a
lock b

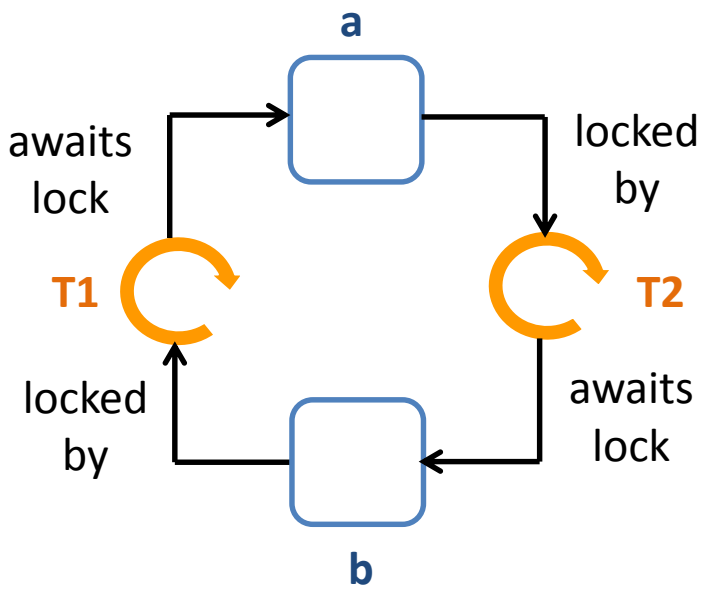
Thread T1
a.CopyTo(b);

lock b
lock a

Thread T2
b.CopyTo(a);

Deadlock

- Threads warten gegenseitig auf Ressourcen-Freigabe
  - Geschachtelte Locks
  - Zyklische Warteabhängigkeiten
- Livelock = Deadlock mit CPU-Verbrauch beim Warten




```
while (!Wait(timeout)) { }
```

Starvation: Nicht viel besser

- Fairness-Problem
  - Ein Thread hat evtl. nie die Chance für Ressourcen-Zugriff
  - Andere kommen ihm andauernd zuvor
- Häufige Kandidaten
  - Timeout und Retry
  - Thread Prioritäten
  - Optimistic Concurrency Control
  - Read/Write Locks im Eigenbau
  - .NET Synchronisationsprimitiven haben keine strikte Fairness

```
a.Lock();
while (!b.TryLock()) {
  a.Unlock();
  // let others continue
  a.Lock();
}
```

Strukturierung der Nebenläufigkeit

- Welche Threads greifen auf welche Objekte zu?
- Welche Objekte müssen Thread-sicher sein?
- Existieren potentiell zyklische Warteabhängigkeiten?

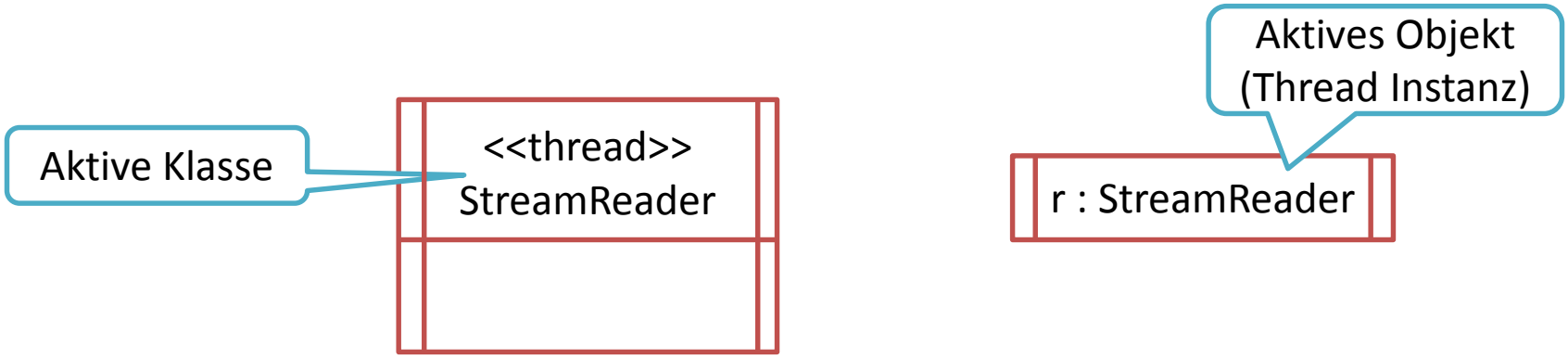
Concurrency Design als Teil der Architektur!

Vorgehen zur Strukturierung

1. Aktive Instanzen bestimmen
2. Interaktionen spezifizieren
3. Synchronisation definieren
4. Korrektheit prüfen

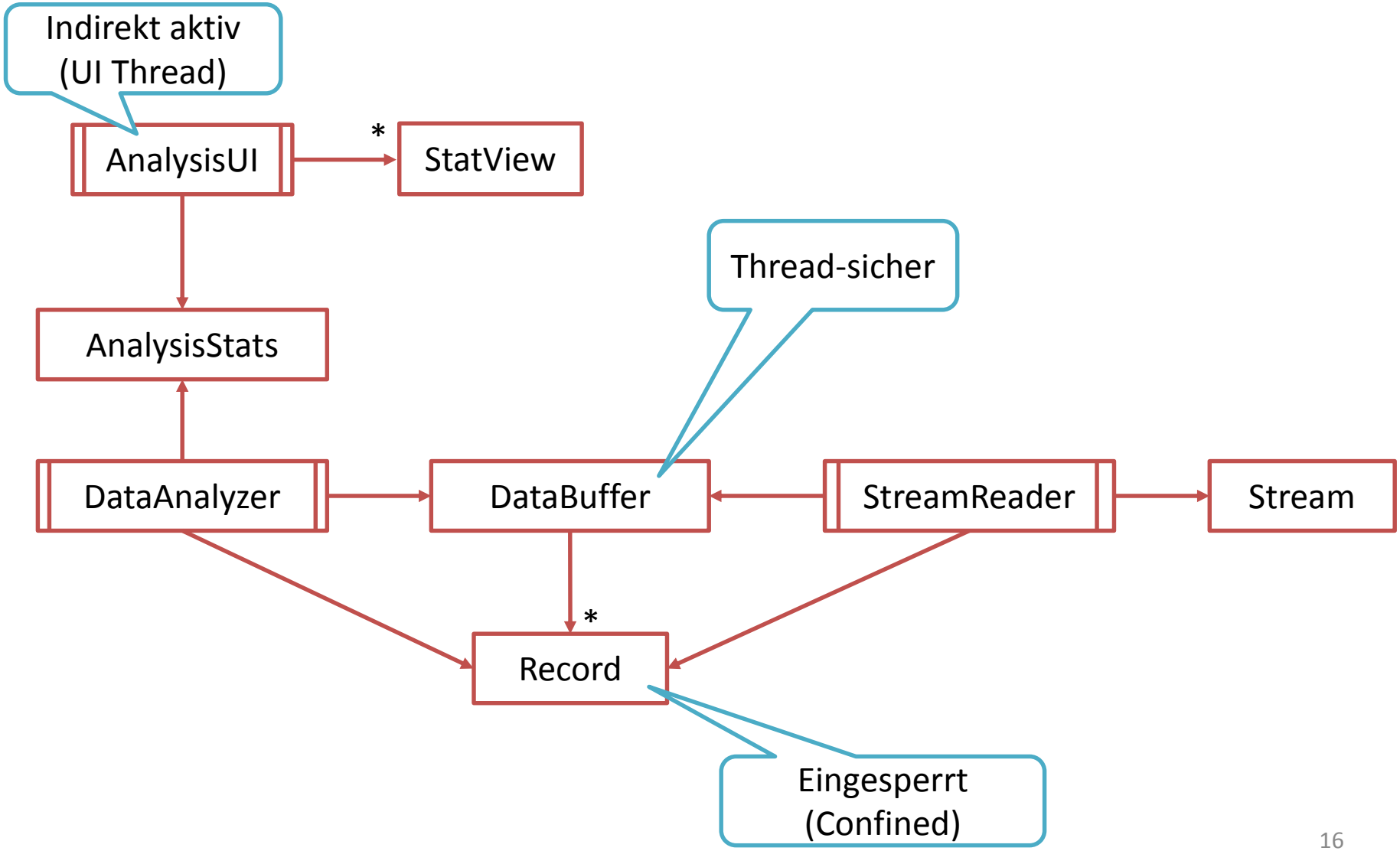
1. Aktive Instanzen bestimmen

- Threads, Parallel Tasks etc.
  - Selbst entworfen
    - Objekte mit intrinsischer Nebenläufigkeit
  - Extern vorgegeben
    - Single UI Thread, Service Worker Threads, …



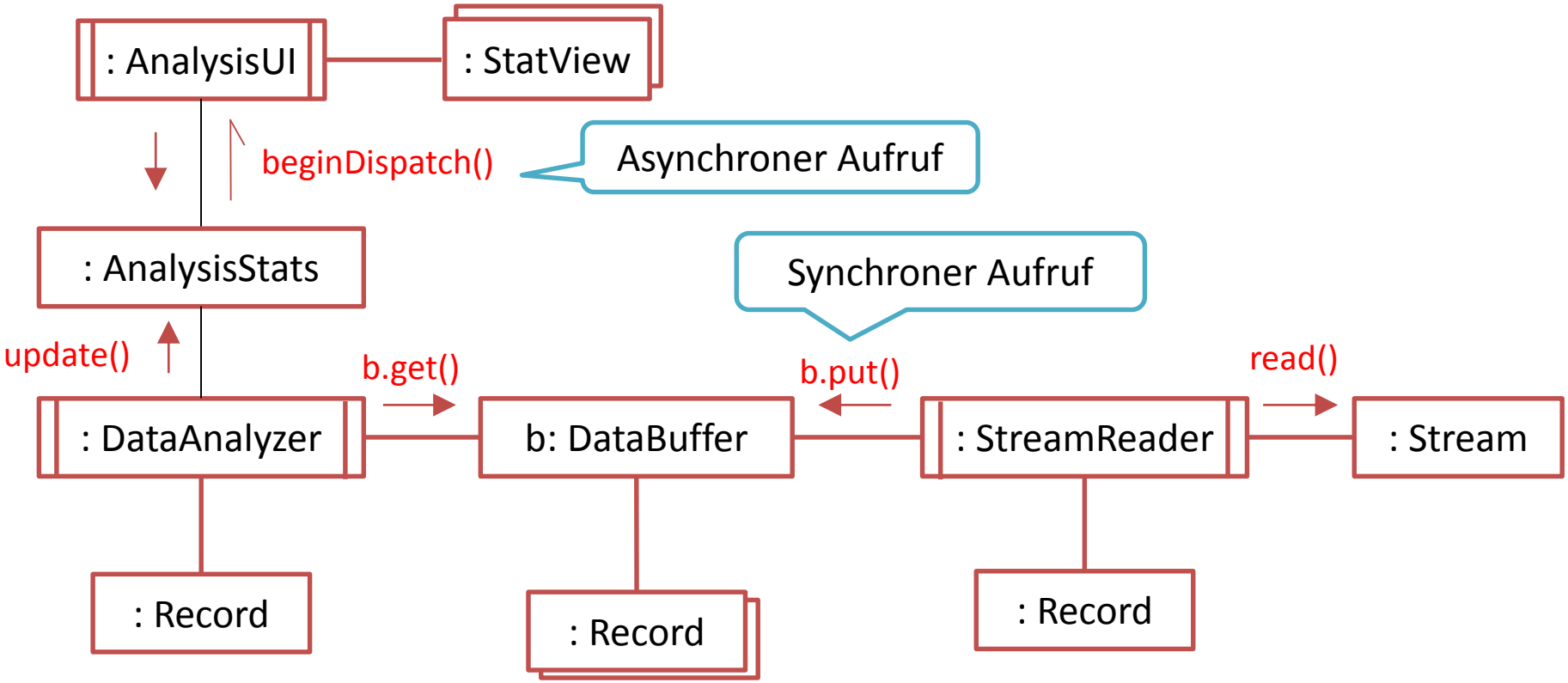
Kann konzeptionelle aktive Instanzen modellieren (Objekte mit inneren Thread-Logik)

Nebenläufiges Klassendiagramm

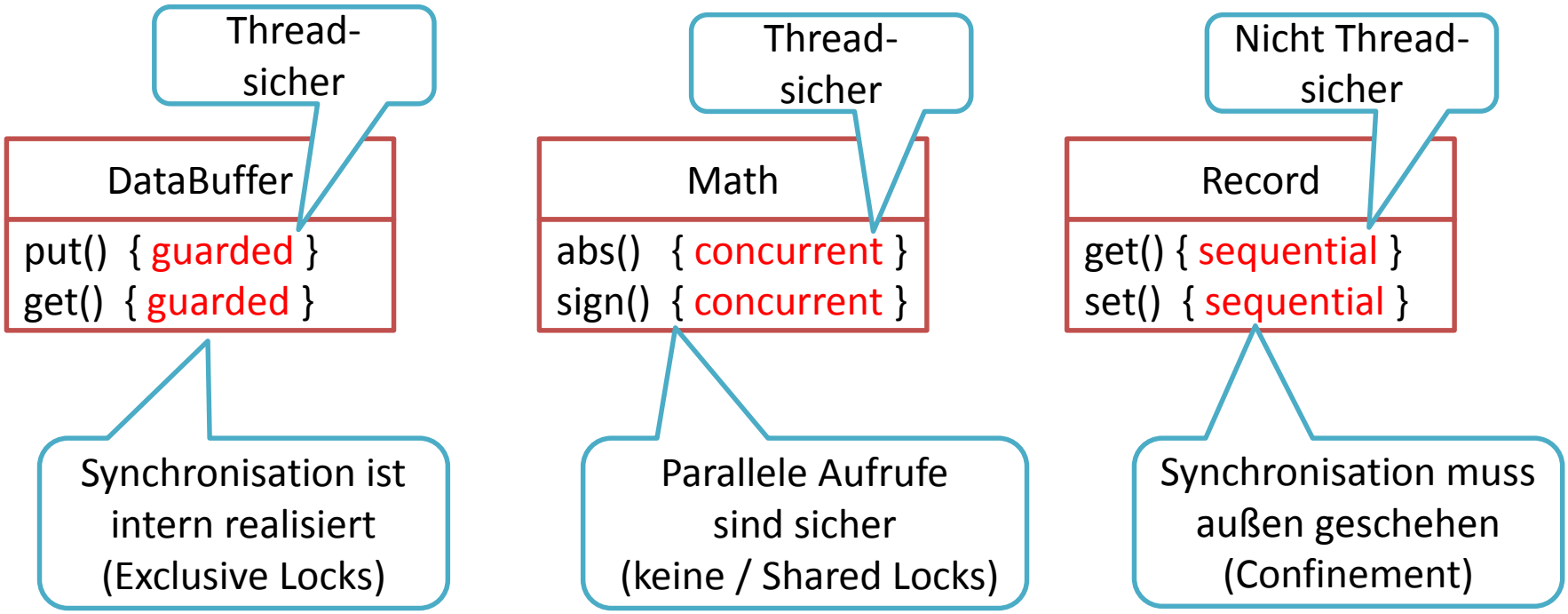


2. Interaktionen spezifizieren

- Kommunikationsdiagramm



3. Synchronisation definieren



4. Korrektheit prüfen

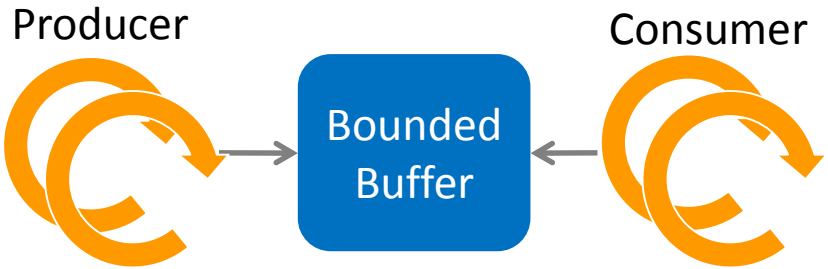
- Keine Race Conditions
  - ❑ Ist Synchronisation oder Confinement pro Klasse definiert?
  - ❑ Sind die kritischen Abschnitte definiert und geschützt?
- Keine Deadlocks
  - ❑ Geschachtelte Locks: Gibt es eine lineare Sperr-Ordnung?
- Keine Starvation
  - ❑ Faire Synchronisation (falls notwendig)?
  - ❑ Keine Prioritätsinversion bei mehreren Thread-Prioritäten

Häufige Muster und Szenarien

- Lose gekoppelte Aktivitäten
  - Producer-Consumer
- Reaktionsfähige UIs/Logiken
  - Asynchrone Ausführung
- Algorithmische Parallelisierung
  - Divide & Parallel Conquer

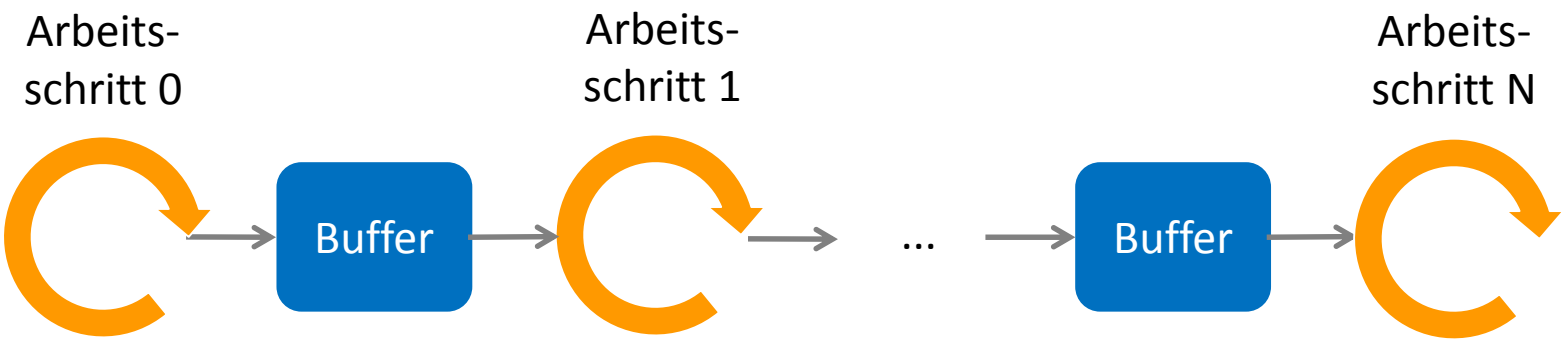
Producer-Consumer

- Partiell entkoppelte Arbeitsschritte
  - Bedienung von IO Kanälen, verzögerter Logger etc.
  - Warten nur, falls Buffer leer oder voll ist
- Buffer muss Thread-sicher sein
  - Concurrent Blocking Collection, Monitor Sync. etc.



Nebenläufige Pipelines

- Eventuell schneller als serielle Verarbeitung



- Arbeitsschritte als Threads oder Tasks (Thread Pool)?

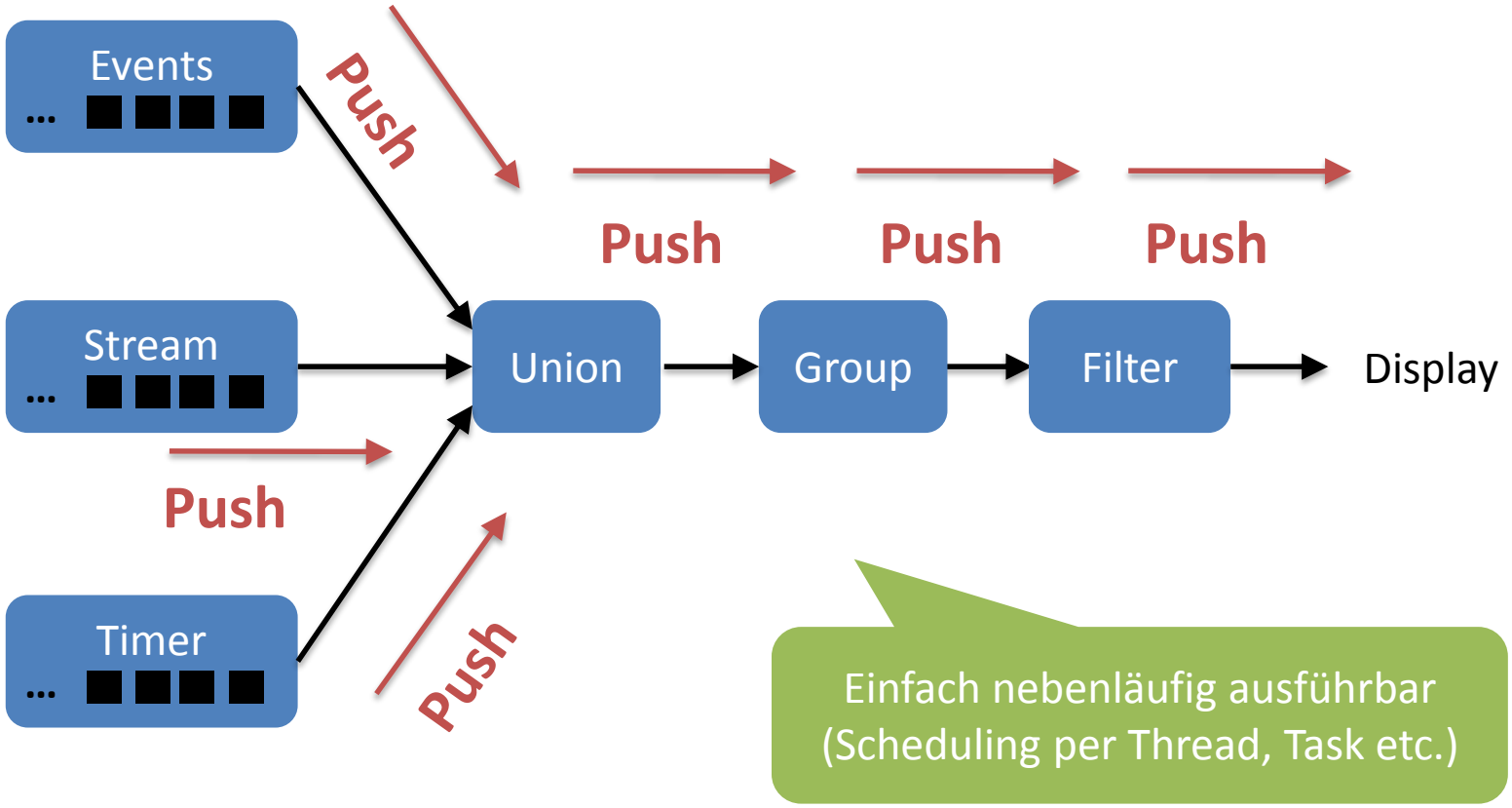
Implementation der Arbeitsschritte

- Threads
  - Für zustandsbehaftete Abarbeitung
  - Bei Abhängigkeiten zwischen den Elementen
- Tasks
  - Für zustandslose Abarbeitung
  - Unabhängige Arbeit pro Element (Run-to-Completion)

Sonst Thread Pool untauglich =>
Thread Injection oder Deadlock

.NET Rx (Reactive Extensions)

- Modellierung von Datenfluss-Pipelines



Reaktionsfähige UIs/Logiken

- GUI ist Single-Threaded
  - Muss blockierende bzw. langlaufende Aufgaben an andere Threads outsourcen (sonst blockiert es)
  - Aber: Nur der UI Thread darf die GUI Controls zugreifen
- Ideal für C# 5 async/await Modell

```
public async Task<int> LongOperationAsync() { … }

…
Task<int> task = LongOperationAsync();
// other work
int result = await task;
// continue
```

Achtung: Async/Await Ausführung

- Async Methoden sind halb-synchron, halb-asynchron
  - Aufrufer führt Methode synchron bis zum blockierenden Await aus
  - Erst danach läuft die Methode asynchron

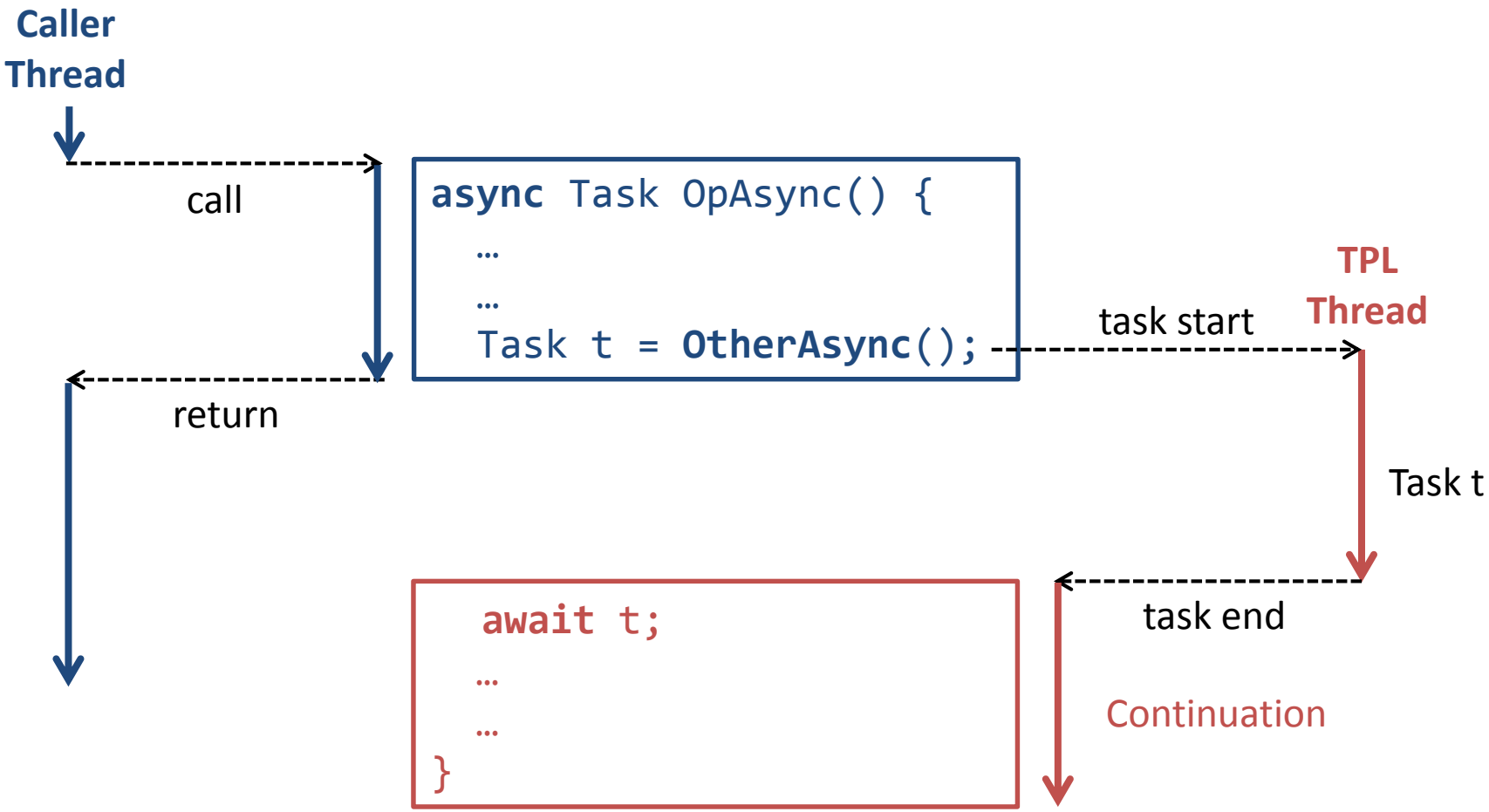
```
async Task<int> GetSiteLengthAsync(string url) {
  HttpClient client = new HttpClient();
  Task<string> task = client.GetStringAsync(url);
  string site1 = await task;
  return site1.Length;
}
```

Synchron
(Aufrufer Thread)

Asynchron
(Potentiell anderer Thread)

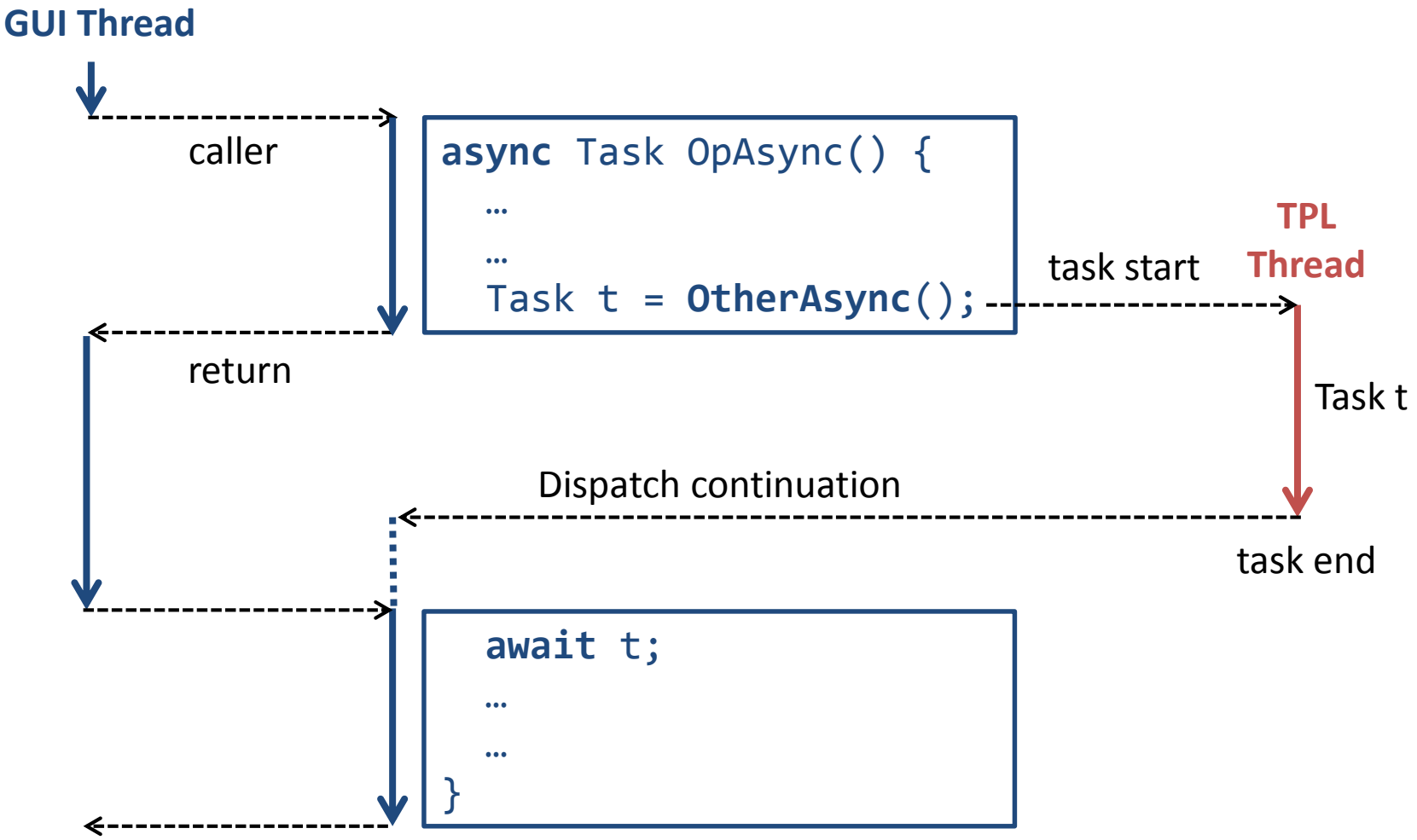
Fall 1: Kein Synchronisations-Kontext

- TPL Thread führt Teil nach (blockierendem) Await aus



Fall 2: Mit Synchronisations-Kontext

- Z.B. GUI Thread als Aufrufer: Dispatch



Divide & Parallel Conquer

- Klassische Parallelisierung zur Beschleunigung
- Fall für den TPL Thread Pool

```
void MergeSort(l, r) {
  long m = (l + r)/2;
  MergeSort(l, m);
  MergeSort(m, r);
  Merge(l, m, r);
}
```

```
Parallel.Invoke(
  () => MergeSort(l, m),
  () => MergeSort(m, r)
);
```

```
void Convert(IList<File> files) {
  foreach (File f in files) {
    Convert(f);
  }
}
```

```
Parallel.Foreach(files,
  f => Convert(f)
);
```

Schlussfolgerungen

- Nebenläufigkeit wird zunehmend wichtiger
  - Relevanz für praktisch jeden Programmierer
- Gefahr der nicht-deterministischen Fehler
  - Nebenläufigkeits-Design in Architektur ist entscheidend
- Diverse Fallstricke lauern in den Technologien
  - Bewusstsein dafür nötig, so lange Tools hier nicht helfen

Danke für Ihr Interesse

- Concurrency Forschung, Consulting, und Kurse
  - http://concurrency.ch
- Kontakt
  - **Prof. Dr. Luc Bläser**
    HSR Hochschule für Technik Rapperswil
    IFS Institut für Software
    Rapperswil, Schweiz
  - lblaeser@hsr.ch