

# Asynchrone Programmierung mit Async und Await in C# 5 unter der Lupe

Prof. Dr. Luc Bläser

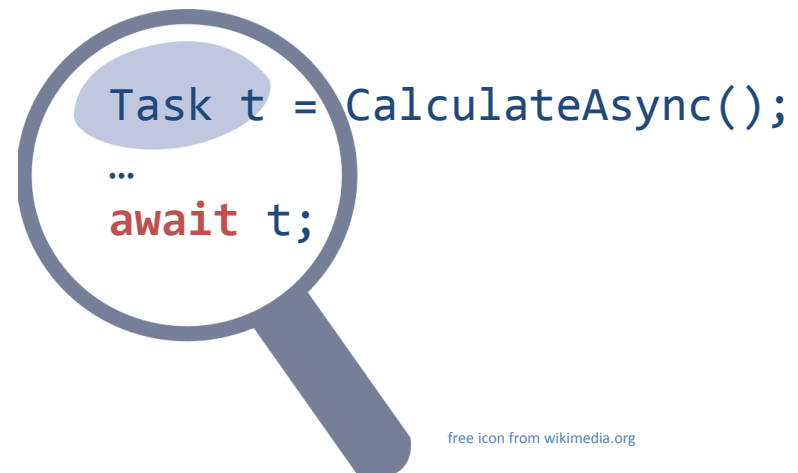
Hochschule für Technik Rapperswil

# Ausgangslage

---

- Async/Await Sprachkonstrukte
  - Institutionalisiert in C# 5 (und VB.NET)
  - Basierend auf Task Parallel Library (TPL) in .NET 4.5/WinRT
- First-Class asynchrones Programmiermodell
  - Substanziell anders als gewöhnliche asynchrone Aufrufe
  - Diverse Besonderheiten und Fallstricke

```
async Task CalculateAsync() {  
    ...  
}
```



# Überblick

---

- Async/Await Funktionsweise
- Ausführungsmodell
- Fallstricke
- Architekturüberlegungen
- Schlussfolgerung

# Asynchrone Programmierung: Zweck

---

- Methoden nebenläufig zum Aufrufer ausführen
  - Aufrufer soll nicht unnötig blockiert werden
  - Bei I/O Calls, Service Aufrufe, Berechnungen

```
int result = LongOperation(...);  
OtherWork();  
Process(result);
```

Unnötig blockierend

Könnte parallel zu  
LongOperation() laufen

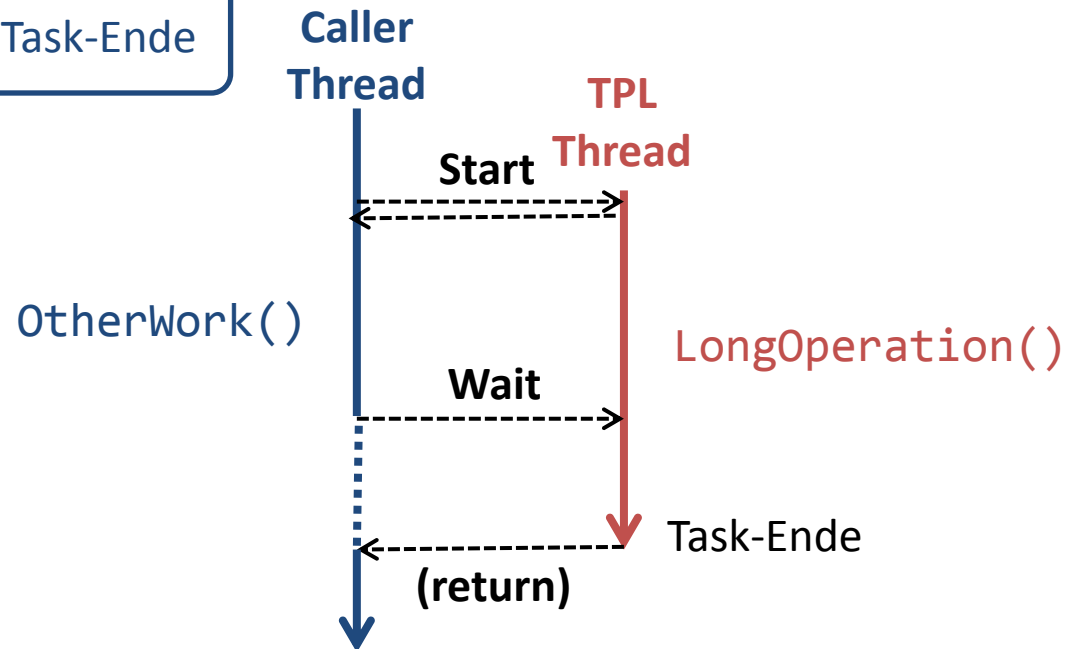
Erst hier muss  
LongOperation() fertig sein

# Klassische Asynchronität mit TPL

```
...  
Task<int> task =  
Task.Factory.StartNew<int>(LongOperation);  
OtherWork();  
int result = task.Result;  
...
```

In Task auslagern

Warte auf Task-Ende



Alternative: Task ruft «Completion Callback»/Folgearbeit nach Beendigung selber auf

# Asynchronität mit Async/Await

---

Async Methode

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

```
...
```

```
Task<int> task = LongOperationAsync();
```

```
OtherWork();
```

```
int result = await task;
```

```
...
```

Warte auf Beendigung  
der Async Methode

# Async und Await

---

- Schlüsselwort **async** für Methode
  - Aufrufer ist nicht zwingend während der gesamten Ausführung der Async Methode blockiert
  - Mögliche Rückgabetypen
    - **void**: «fire-and-forget»
    - **Task**: Keine Rückgabe, aber erlaubt Warten auf Ende
    - **Task<T>**: Für Methode mit Rückgabotyp T
  - Keine **ref** oder **out** Parameter
- Schlüsselwort **await** für Tasks
  - Warte auf Ende eines TPL Tasks (sog. „Awaiter“)
  - Liefert Resultat des Tasks (sofern Rückgabotyp definiert)

# Beispiel: Async Methode

---

Rückgabebetyp string

Suffix „Async“ als  
Namenskonvention

```
async Task<string> ConcatWebSitesAsync(string url1, string url2)
{
    HttpClient client = new HttpClient();
    Task<string> download1 = client.GetStringAsync(url1);
    Task<string> download2 = client.GetStringAsync(url2);
    string site1 = await download1;
    string site2 = await download2;
    return site1 + site2;
}
```

Direkte Rückgabe  
des String

`async Task<string>`  
`GetStringAsync(string url)`



# Spezielle Regeln

---

- `async` Methode
  - Muss `await` enthalten => Sonst Compiler-Warnung
- `await` Anweisung
  - Nur in `async` Methode erlaubt => Sonst Compiler-Fehler



Grund: Spezielles Ausführungsmodell

# Ausführungsmodell

---

- Async Methode läuft teilweise **synchron**, teilweise **asynchron**
  - Aufrufer führt Methode solange **synchron** aus, bis ein blockierendes `await` anliegt
  - 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**  
(evtl. anderer  
Thread)

# Mechanismus

---

- Compiler zerlegt Methode in Abschnitte
  - Erster Abschnitt vor `Await`: Synchron durch Aufrufer
  - Abschnitt nach `Await`: Läuft später nach Task-Ende («Continuation»)

Synchron durch Aufrufer

```
async Task OpAsync() {  
    ...  
    ...  
    Task t = OtherAsync();  
}
```

```
    await t;  
    ...  
    ...  
}
```

Asynchron nach Task-Ende

# Async Methodenaufruf

---

- Methode läuft synchron bis zu blockierenden Await
  - Bei Warten auf anderen Thread bzw. externes I/O
- Bei blockierendem Await Rücksprung zum Aufrufer

Aufrufer  
Thread



Aufruf



Rücksprung



```
async Task OpAsync() {  
    ...  
    ...  
    Task t = OtherAsync();  
}
```

```
await t;
```

```
...
```

```
...
```

```
}
```

Später nach Task-Ende

# Abschnitt nach blockierendem Await

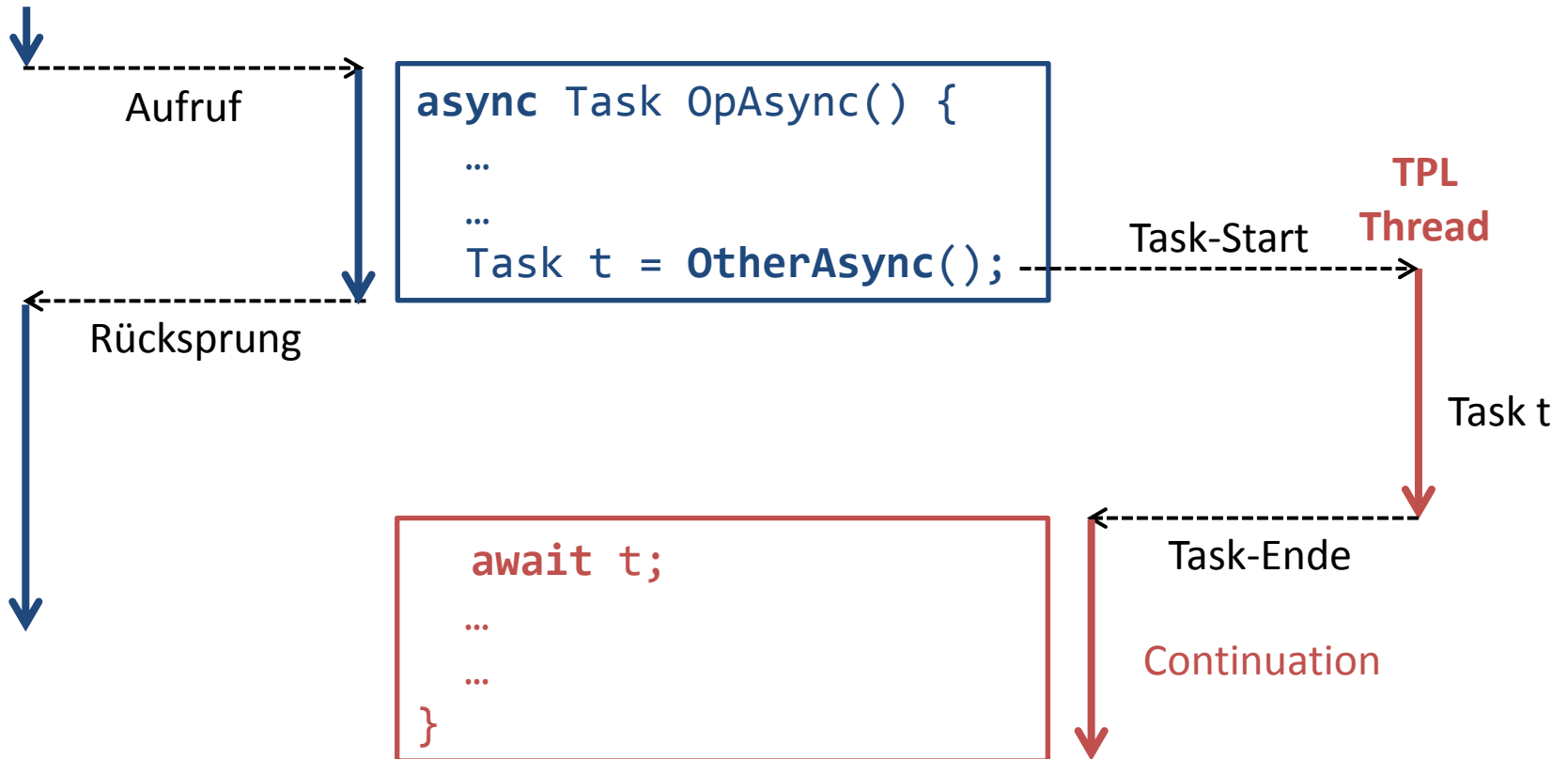
---

- Fall 1: Aufrufer ohne Synchronisationskontext
  - `SynchronizationContext.Current == null`
    - Meiste Threads (Konsole, TPL etc.)
  - Abschnitt wird durch Thread des erwarteten Tasks ausgeführt
- Fall 2: Aufrufer mit Synchronisationskontext
  - Z.B. GUI-Thread
  - Abschnitt wird an diesen Kontext «dispatched»
    - Wird z.B. als Event vom GUI-Thread ausgeführt

# Fall 1: Kein Synchronisationskontext

- Task-Thread führt Abschnitt nach Await aus

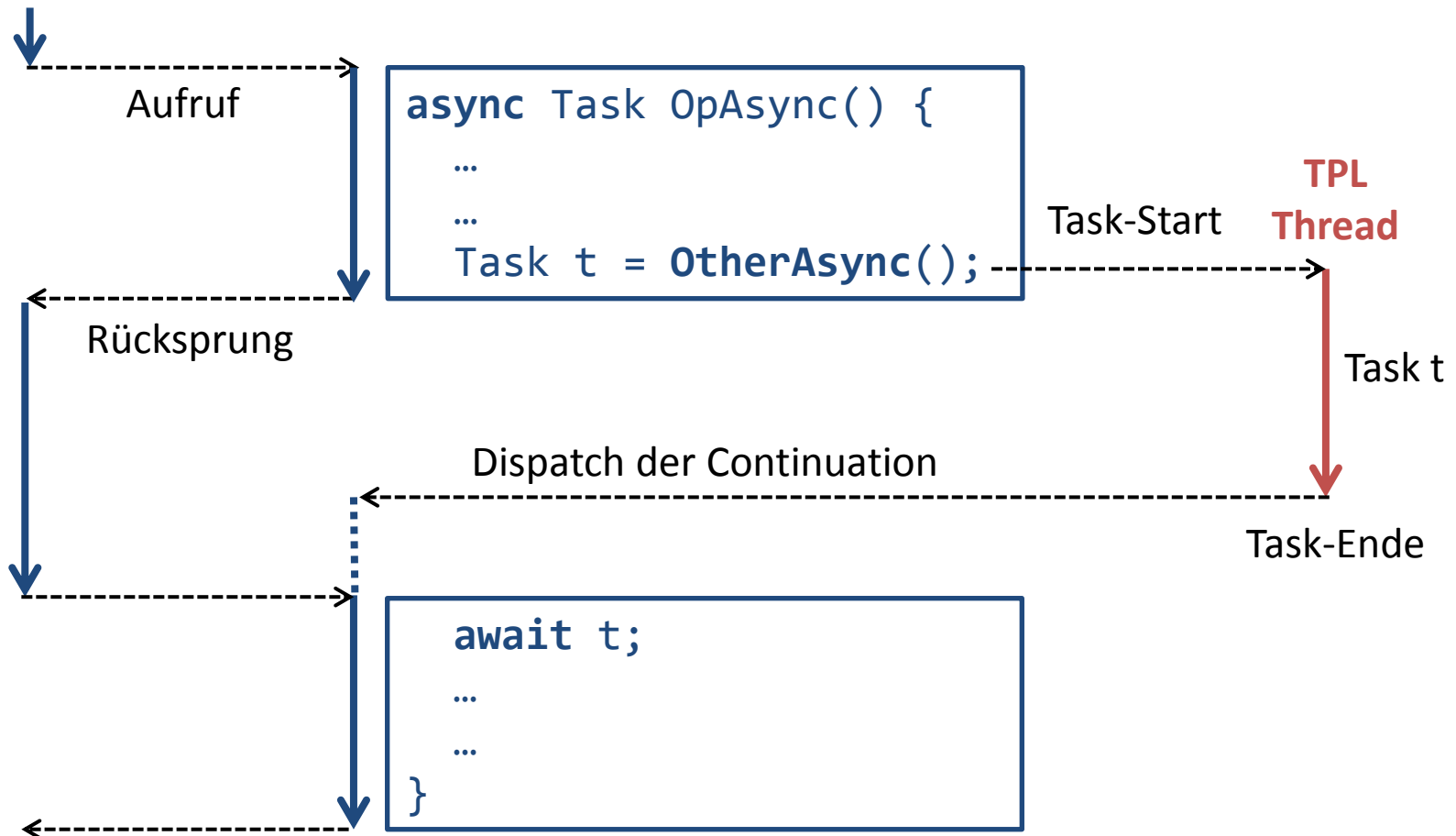
Aufrufer  
Thread



# Fall 2: Mit Synchronisationskontext

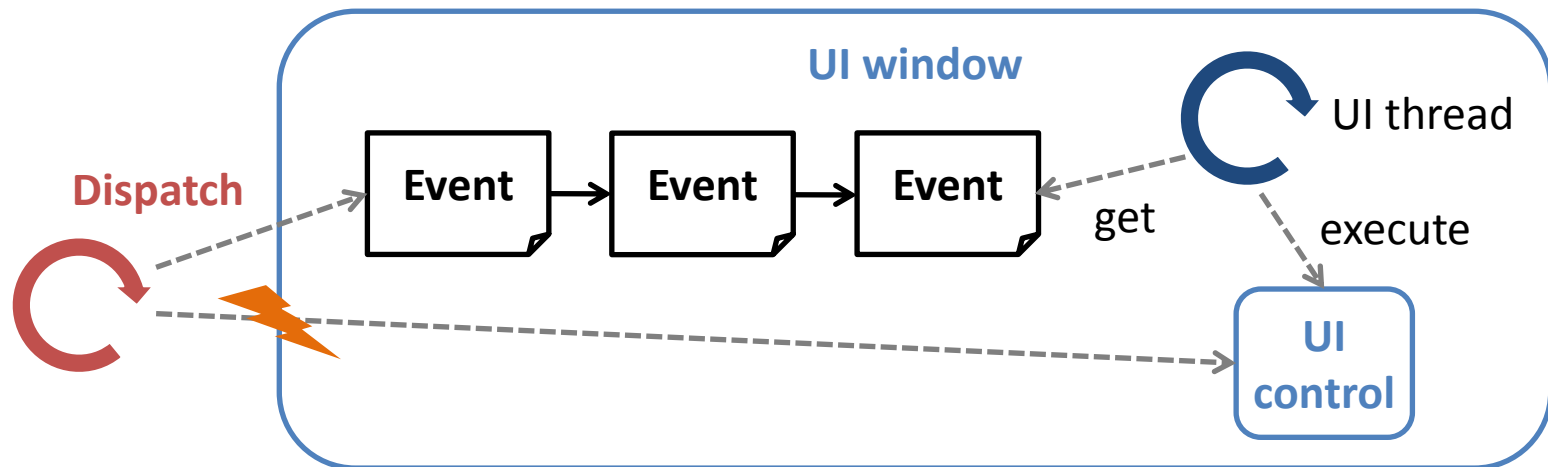
- Beispiel GUI-Thread als Aufrufer: Dispatch

GUI Thread



# Wieso diese Komplexität?

- GUI Frameworks sind Single-Threaded
  - Nur dedizierter UI Thread darf UI-Controls zugreifen
  - Threads können Arbeiten als UI-Events beim UI einreihen
  - UI Thread verarbeitet sukzessive diese UI-Events





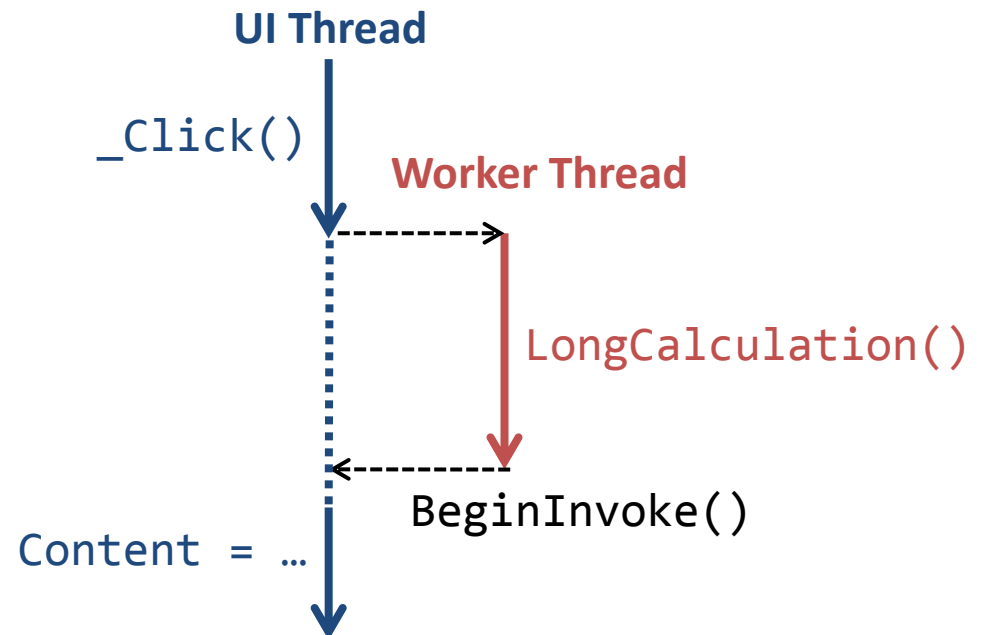
# Klassische UI-Programmierung

```
void startCalculationButton_Click(...) {  
    new Thread(LongCalculation).Start();  
}
```

Langlaufende Operation  
an Thread outsourcen

```
void LongCalculation() {  
    ...calculation...  
    Dispatcher.BeginInvoke(new Action(() => {  
        resultLabel.Content = ...;  
    }));  
}
```

Fremder Thread darf UI nicht  
zugreifen => Dispatch zu UI



# Nicht-Blockierende UIs

---

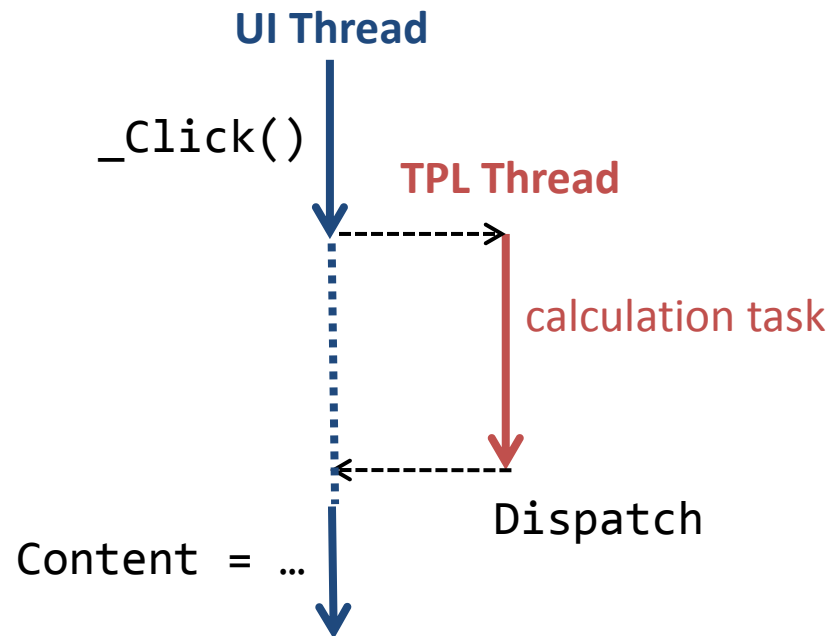
- Klassisch: Zerstückelung der Logik
  - Kette von Dispatch (UI Thread/fremder Thread)
  - Hilfsklasse BackgroundWorker
- Leserlicher Code mit Async/Await
  - Logik in einem Guss (eine Methode)
  - Zerstückelung in mehrere UI-Events hinter Kulissen

# UI mit Async/Await

---

```
async void startCalculationButton_Click(...) {  
    await LongCalculationAsync();  
    resultLabel.Content = ...;  
}
```

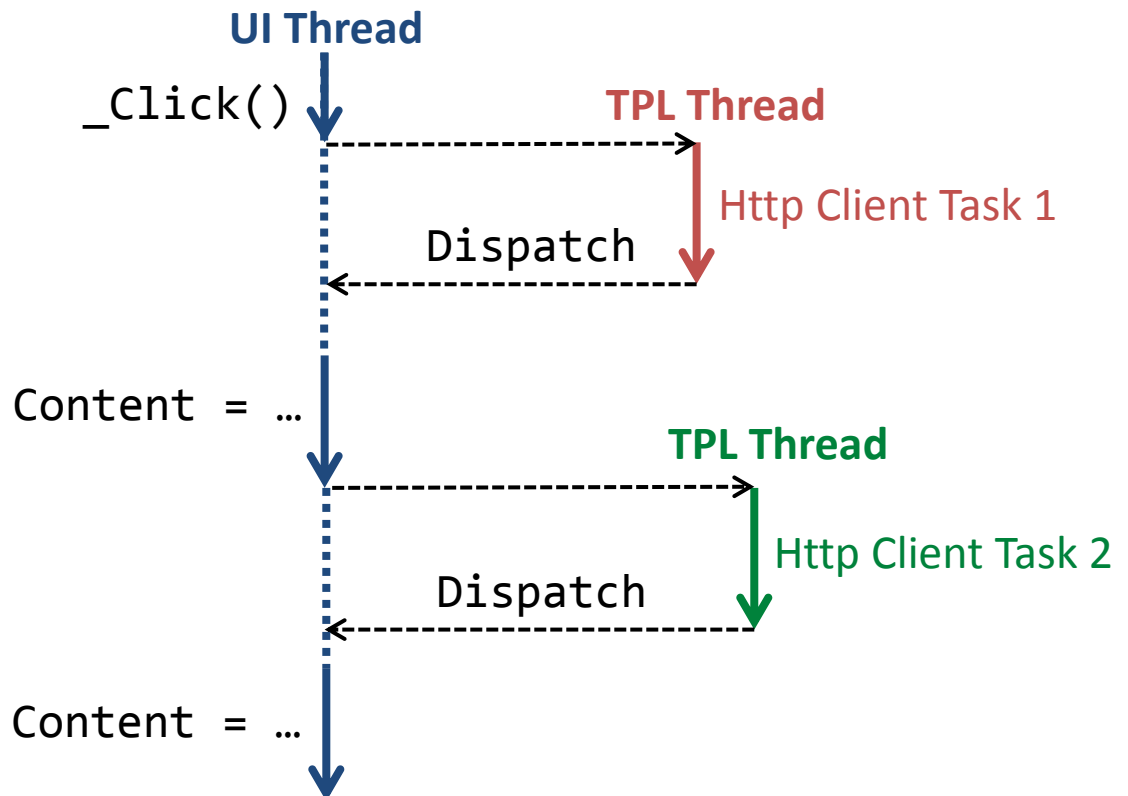
```
async Task LongCalculationAsync() {  
    ...start and await calculation task...  
}
```



# Async/Await Sequenz

```
async void startDownload_Click(...) {  
    HttpClient client = new HttpClient();  
    foreach (var url in collection) {  
        var data = await client.GetStringAsync(url);  
        textArea.Content += data;  
    }  
}
```

API bietet diverse  
async Methoden



# Tipps zu Async/Await (1)

---

- **async** Methode in synchroner Methode aufrufen

- `task.Wait()` oder `task.Result`

- ```
void Test() {  
    Task myTask = LongOpAsync();  
    myTask.Wait();  
}
```

- **async** Lambda

- ```
async () => {  
    await SomeOpAsync();  
    ...  
}
```

# Spezialfall

---

- `configureAwait(false)` unterbindet Dispatch an Synchronisationskontext
  - Verhalten wie ohne Synchronisationskontext

```
async void button_Click() {  
    ...  
    ...  
    Task t = CalculationAsync().configureAwait(false);  
    await t;  
    ...  
    ...  
}
```



UI Thread

Anderer Thread

# Achtung Fallstricke

---



1. Async Methoden sind nicht per se asynchron
2. Thread-Wechsel innerhalb Methode
3. Quasi-Parallelität der UI-Event-Handler
4. Race Conditions bleiben möglich
5. UI-Deadlocks wegen Async Task-Zugriff
6. Exceptions werden teilweise ignoriert
7. Frühzeitiger Abbruch von «Fire-And-Forget»

# Pitfall 1

---

- Async Methoden sind nicht per se asynchron

```
public async Task<bool> IsPrimeAsync(long number) {  
    for (long i = 2; i <= Math.Sqrt(number); i++) {  
        if (number % i == 0) { return false; }  
    }  
    return true;  
}
```

Läuft vollständig synchron



**Aufrufer ist während gesamter Ausführung blockiert**

**Compiler-Warnung: kein Await in Async Methode**



# Pitfall 1: Workaround

---

- Rechenintensive Operation explizit als Task ausführen

```
public async Task<bool> IsPrimeAsync(long number) {  
    return await Task.Run(() => {  
        for (long i = 2; i <= Math.Sqrt(number); i++) {  
            if (number % i == 0) { return false; }  
        }  
        return true;  
    });  
}
```

# Pitfall 1: Diskussion

---

- Muss situativ explizit Task in async lancieren
  - Um Asynchronität zu erhalten
- Compiler-Warnung ist nicht hinreichend

```
public async Task ComputeAsync() {  
    var result = VeryLongCalculation();  
    await UploadResultAsync(result);  
}
```

Läuft immer noch auf  
Kosten des Aufrufers

**Keine Compiler-Warnung**

# Pitfall 2

---

- Thread-Wechsel innerhalb Methode
  - Falls kein UI / Synchronisationkontext mit Dispatcher

```
public async Task BlockingOperation() {  
    Console.WriteLine("BEFORE " + Thread.CurrentThread.ManagedThreadId);  
    HttpClient client = new HttpClient();  
    Task<string> task = client.GetStringAsync("...");  
    string result = await task;  
    Console.WriteLine("AFTER " + Thread.CurrentThread.ManagedThreadId);  
}
```



**Partielle Nebenläufigkeit berücksichtigen  
Achtung bei Thread-lokalen Variablen**

BEFORE 10

...

AFTER 14

# Pitfall 3

---

- Quasi-Parallelität der UI-Event-Handler
  - Im UI: Bei jedem `Await` können natürlich andere UI-Events dazwischen laufen

```
async void startDownload_Click(...) {  
    HttpClient client = new HttpClient();  
    foreach (var url in collection) {  
        var data = await client.GetStringAsync(url);  
        textArea.Content += data;  
    }  
}
```

User-Click dazwischen =>  
collection evtl. verändert



## Muss Interferenzen unterbinden

- Snapshots / Isolation von anderen Events
- Zwischenevent-Ausführung einschränken

# Pitfall 4

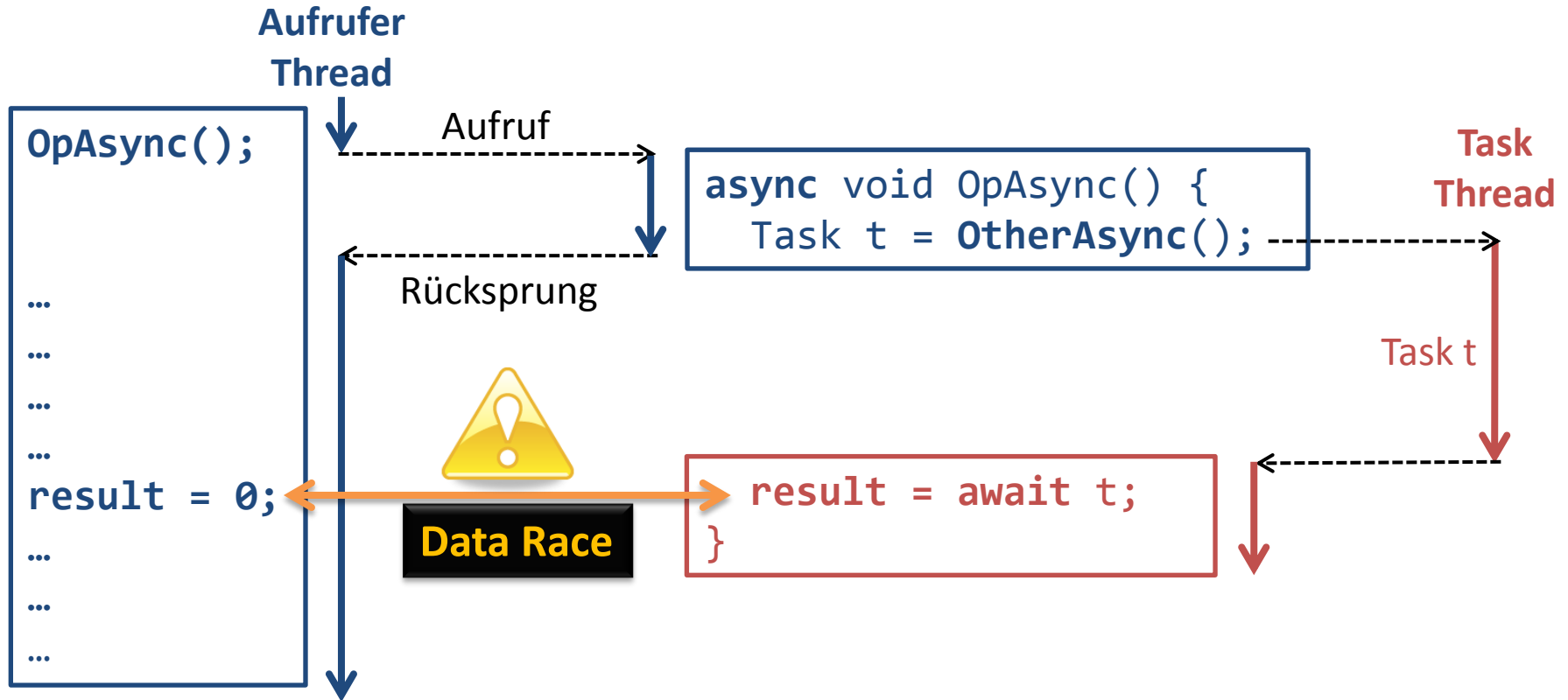
---

“The async-based approach to asynchronous programming is preferable to existing approaches in almost every case. In particular, this approach is better [...] because the code is simpler and you don't have to guard against race conditions”

- <http://msdn.microsoft.com/en-us/library/vstudio/hh191443.aspx>

- Race Conditions bleiben natürlich möglich!
  - Abschnitt nach await läuft potentiell parallel zum Aufrufer
    - Falls ohne Synchronisationkontext (z.B. kein UI-Thread)
    - Falls `ConfigureAwait(false)`
  - Explizite gestartete Task (`Task.Run()`, `Threads` etc.)

# Pitfall 4: Race Condition



# Pitfall 5

---

- UI-Deadlocks

- Aufruf von `Task.Wait()` oder `Task.Result` in UI-Thread

```
void calculationButton_Click(...) {  
    Task<string> task = CalculateAsync();  
    label.Content = task.Result;  
}
```

Implizites `Task.Wait()`  
=> Blockiert UI Thread



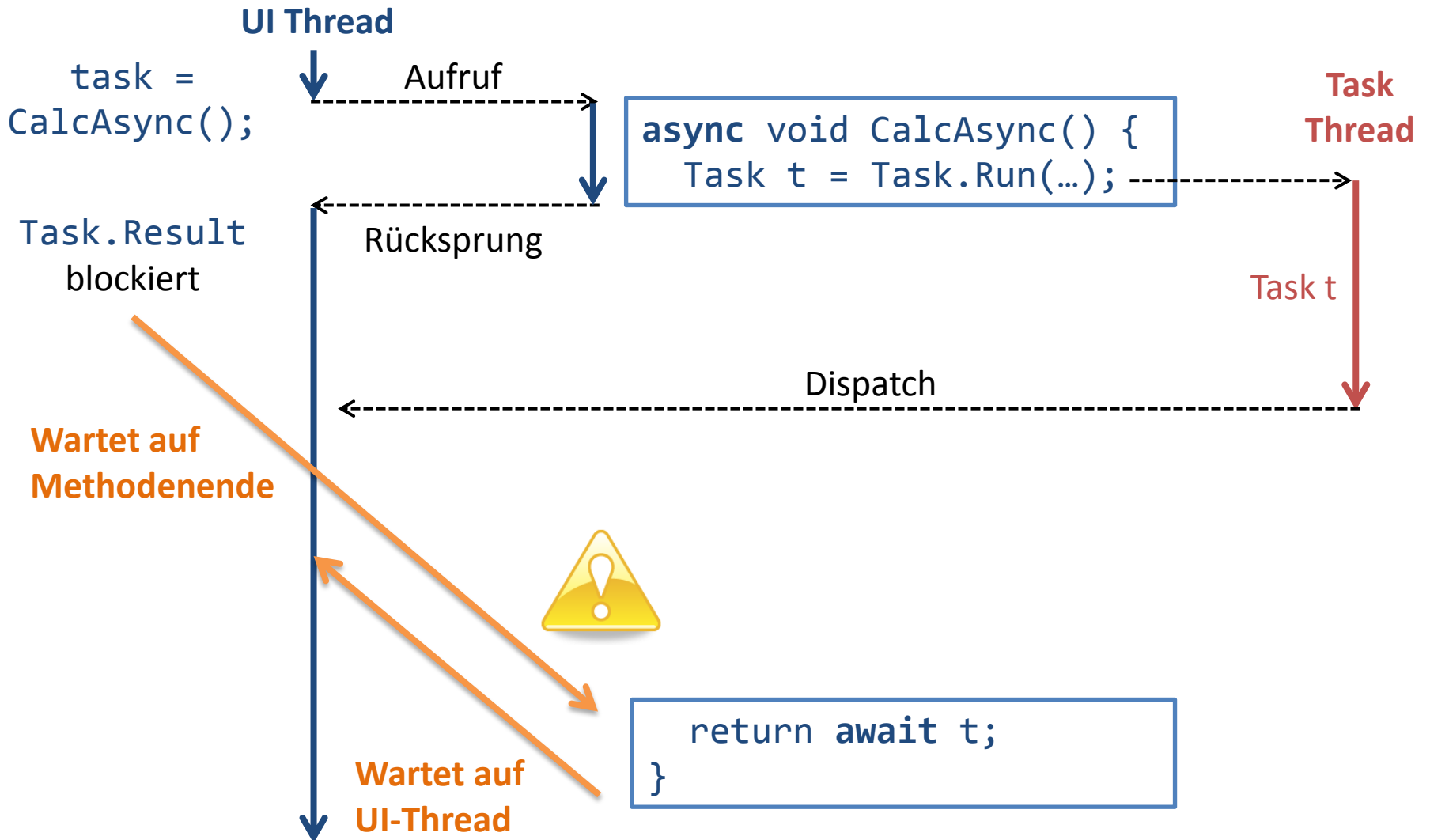
```
async Task<string> CalculateAsync() {  
    return await Task.Run(...);  
}
```

Abschnitt nach `await` kann nicht  
laufen, weil UI-Thread blockiert ist



**Deadlock**

# Pitfall 5: Analyse





# Pitfall 6

---

- Exceptions werden unter Umständen ignoriert
  - Aufrufer ohne Synchronisationskontext
  - Für async void Methoden
  - Für async Methoden ohne await/Wait/Result-Abfrage

```
void button_Click(...) {  
    LongCalculationAsync().ConfigureAwait(false);  
}
```

```
async Task LongCalculationAsync() {  
    await Task.Run(() => { ... });  
    throw new Exception("TEST");  
}
```



**Exception wird ignoriert**

# Pitfall 6: Analyse

---

- Exceptions werden nur bei Task-Zugriff propagiert
  - Seit 4.5 werden Exceptions in verwaisten Tasks ignoriert
    - Vorher per Garbage Collector ausgelöst
  - `TaskScheduler.UnobservedTaskException` Event zum Abfangen

# Pitfall 7

---

- «Fire-and-Forget» Async-Methode läuft unter Umständen nicht fertig
  - Main-Methode ist fertig und keine sonstige «normalen» (Foreground) Threads laufen

```
static void Main() {  
    ComputeAsync();  
}
```

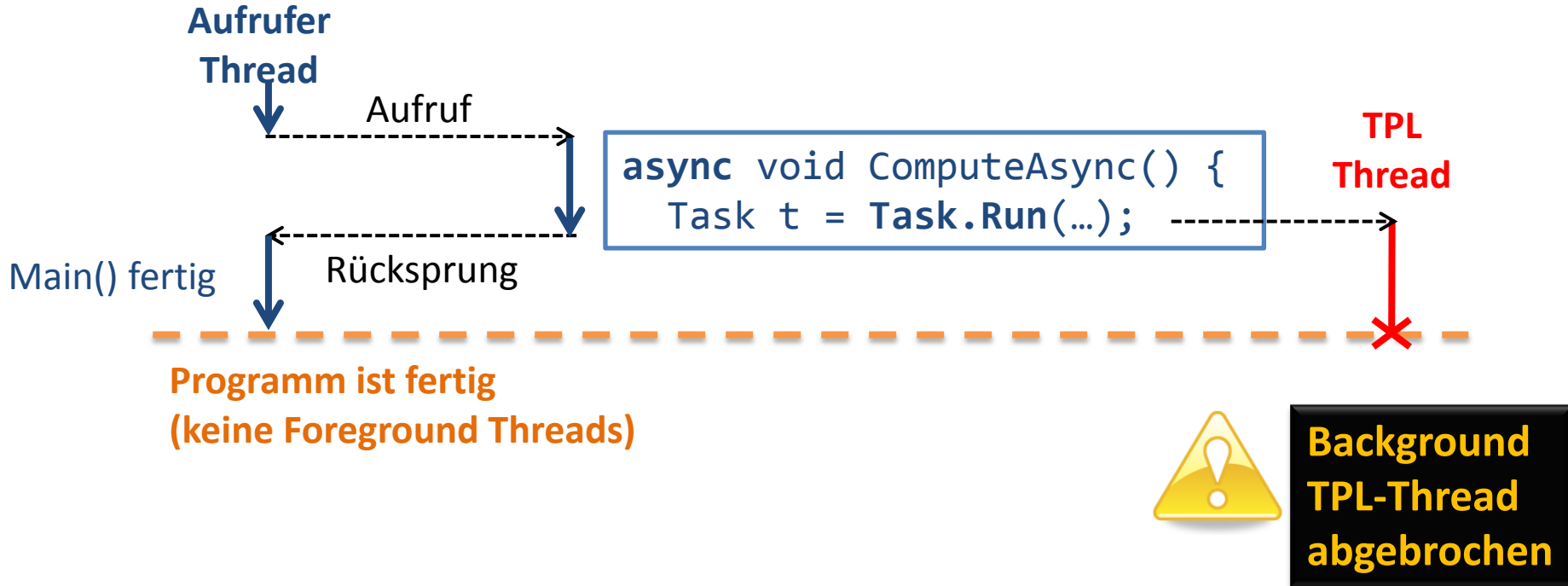
```
async void ComputeAsync() {  
    Task task = Task.Run(...);  
    Console.WriteLine(await task);  
}
```



**Eventuell nicht mehr ausgeführt**

# Pitfall 7: Hintergrund

- TPL Thread Pool benutzt Background-Threads
  - Erlauben jederzeit Programm-Beendigung
- Programm läuft, solange es Foreground-Threads gibt



# Pitfall 7: Lösung

---

- Auf Ende von Async-Methoden warten
  - «Normaler» Foreground Thread existiert, solange Background Thread läuft

```
static void Main() {  
    Task t = ComputeAsync();  
    t.Wait();  
}
```

```
async Task ComputeAsync() { ... }
```

# Architekturüberlegungen (1)

---

- Eine Async-Methode => Zwei Aufruffälle
    - Mit und ohne Synchronisationskontext
    - Zwei unterschiedliche Verhalten => beides testen
  - Viraler Effekt: async/await-Aufrufer wird auch async
    - Unnötig komplexe Methodensignaturen mit Tasks
    - Kosten wegen interner Methodenzerstückelung
  - Synchrone/asynchrone Verwendung sollte meist orthogonal sein
    - «Caller» sollte entscheiden, nicht «Callee»
- => Risiko von unnötig hoher Komplexität

# Architekturüberlegungen (2)

---

- Primär im UI Layer sinnvoll
  - Muss aber Zerstückelungs-Mechanismus verinnerlichen
  - Sowieso meist Top-Layer in Architektur
- In darunterliegenden Layer weniger
  - Limitiert auf inhärent blockierenden Operationen (I/O)
    - Appellation an Caller, asynchrones Design zu wählen
  - Problematisch, wenn Methoden ad-hoc async werden
    - Nebenläufigkeit/Verzahnung kommt ins Spiel
  - Aufrufer kann sowieso leicht eine gewöhnliche Methode asynchron aufrufen (`Task.Run()`)

# Unterschiedliche Auffassungen

---

- MS Ansatz: «Bottom-Up» Async-Design
  - Schon einfache blockierende Methoden sind async
  - Direkte / indirekte Aufrufer werden auch async
  - Aufrufer muss auf Wunsch selber synchron machen
- Empfehlung: «Top-Down» Async-Design
  - Kleine Methoden / in Basislayers eher synchron designen
  - Top-Level Methoden bei Bedarf asynchron machen
  - Async-Komplexität auf Top-Level beschränkt



# Schlussfolgerungen

---

- Komplizierter Mechanismus
  - Anders als gewöhnliche asynchrone Programmierung
  - Genaues Verständnis unabdingbar
  - Diverse Fallstricke
- Wohlüberlegter Einsatz
  - Primär für UI-Layer angemessen
  - Sonst schnell hohe Komplexität

# Danke für Ihr Interesse

---

- .NET Concurrency Industriekurse
  - <http://concurrency.ch/Training>
- Kontakt
  - **Prof. Dr. Luc Bläser**  
**HSR Hochschule für Technik Rapperswil**  
IFS Institut für Software  
Rapperswil, Schweiz
  - [lblaeser@hsr.ch](mailto:lblaeser@hsr.ch)
  - <http://ifs.hsr.ch>  
<http://concurrency.ch>