Die Intranet-Agentur.

IPI

`async` und `await`
Warum man den Verlockungen von Keywords manchmal widerstehen sollte

Matthias Unruhe
Track „MacGyver“

`async` und `await` – Warum man den Verlockungen von Keywords manchmal widerstehen sollte

- Mit den (nicht mehr ganz so) neuen Keywords `async` und `await` macht es Microsoft den .NET-Entwicklern einfach, nebenläufigen Code zu implementieren und so die Rechenleistung von Multi-Core-Prozessoren besser auszuschöpfen.

- Doch der Teufel steckt im Detail.
  Leider ist nicht immer garantiert, dass ein asynchroner Aufruf, der via `await` erwartet wird, auch tatsächlich zurück kommt.

- Und was machen wir eigentlich mit unserem alten synchronen Code?
  Müssen wir den als "Legacy Code" betrachten und Big Bang-Refactorings angehen?

- Dieser Vortrag adressiert die vorgenannten Probleme und stellt eine Lösung vor.

Agenda

- Was passiert bei der Verwendung von `async` und `await`?
- Best Practices bei der Verwendung von `async`, `await` und `Task`
- Wie verwendet man asynchronen Code
  ohne seine (synchrone) Abstraktion zu „zerstören“?

IPI

Was passiert bei der Verwendung von `async` und `await`?

Was passiert bei der Verwendung von async und await?

- Bei async und await geht es um den Umgang mit dem „Managed Thread Pool" und dem „Synchronization Context"
- Code startet auf Thread 1
- Der asynchrone Aufruf wird auf Thread 2 ausgeführt
- Wenn Thread 2 fertig ist, benachrichtigt er den Synchronization Context.
- Der Synchronization Context benachrichtigt Thread 1, um den restlichen Code auszuführen

```
1  async Task ReadDataFromUrl(string url)
2  {
3      WebClient client = new WebClient();
4      byte[] result = await client.DownloadDataTaskAsync(url);
5      string data = Encoding.UTF8.GetString(result);
6      this.LoadData(data);
7  }
```

Was passiert bei der Verwendung von `async` und `await`?

- `async`
  - Wird nur im Kopf einer Methode verwendet
  - Ist nur eine Information an den Compiler: „Diese Methode beabsichtigt `await` zu verwenden“
    Das heißt im Umkehrschluss aber nicht, dass in der Methode das `await`-Keyword verwendet werden muss
  - Ändert **nicht** die Methodensignatur, sondern nur wie die Methode kompiliert wird
    Der Compiler macht aus dem Methodenrumpf eine eigene Klasse, deren Code aus der Methode heraus aufgerufen wird
  - Macht die Methode **nicht** asynchron

- `await`
  - Wird nur im Rumpf einer Methode verwendet
  - Kann nur verwendet werden, wenn der Methodenkopf das Keyword `async` enthält
  - Informiert den Compiler, wie der Code in der „asynchronen Klasse“ aufgeteilt werden soll

Was passiert bei der Verwendung von async und await?

```
async Task ReadDataFromUrl(string url)
{
  WebClient client = new WebClient();

  byte[] result = await client
    .DownloadDataTaskAsync(url);

  string data = Encoding.UTF8
    .GetString(result);

  this.LoadData(data);
}
```

Under the Hood

```
[AsyncStateMachine(typeof(ReadDataFromUrl))]
Task ReadDataFromUrl(string url)
{
    ReadDataFromUrl stateMachine = new ReadDataFromUrl();

    stateMachine._this = this;
    stateMachine.url = url;
    stateMachine.builder = AsyncTaskMethodBuilder.Create();
    stateMachine.state = -1;

    AsyncTaskMethodBuilder builder = stateMachine.builder;
    builder.Start(ref stateMachine);

    return stateMachine.builder.Task;
}
```

Was passiert bei der Verwendung von async und await?

```
async Task ReadDataFromUrl(string url)
{
  WebClient client = new WebClient();

  byte[] result = await client
    .DownloadDataTaskAsync(url);

  string data = Encoding.UTF8
    .GetString(result);

  this.LoadData(data);
}
```


Under the Hood

```
[CompilerGenerated]
sealed class ReadDataFromUrl : IAsyncStateMachine
{
  public int state;
  // Weitere Felder
  public string url;
  private WebClient client;
  private byte[] result;
  private string data;

  public void MoveNext()
  {
    try
    {
      // Asynchroner Code
    }
    catch (Exception exception)
    {
      this.state = -2;
      this.builder.SetException(exception);
      return;
    }

    this.state = -2;
    this.builder.SetResult();
  }

}
```

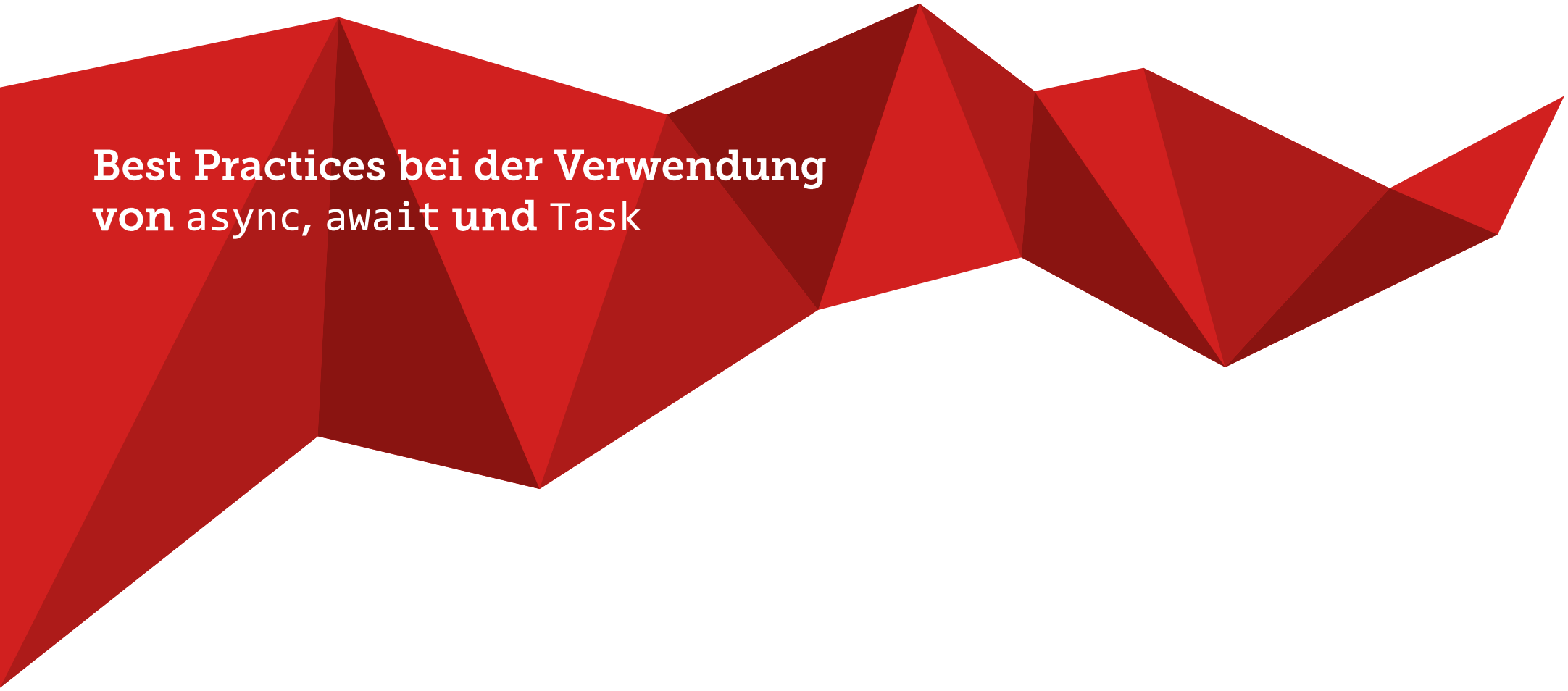
Was passiert bei der Verwendung von async und await?

```
async Task ReadDataFromUrl(string url)
{
  WebClient client = new WebClient();

  byte[] result = await client
    .DownloadDataTaskAsync(url);

  string data = Encoding.UTF8
    .GetString(result);

  this.LoadData(data);
}
```


Under the Hood – Asynchroner Code

```
[CompilerGenerated]
sealed class ReadDataFromUrl : IAsyncStateMachine {
  public void MoveNext() {
    TaskAwaiter<byte[]> awaiter;
    if (this.state != 0) {
      this.client = new WebClient();
      awaiter = client.DownloadDataTaskAsync(url).GetAwaiter();
      if (!awaiter.IsCompleted) {
        this.state = 0;
        this.awaiter = awaiter;
        ReadDataFromUrl code = this;
        this.builder.AwaitUnsafeOnCompleted(ref awaiter, ref code);
        return;
      }
    }
    else {
      awaiter = this.awaiter;
      this.awaiter = default(TaskAwaiter<byte[]>);
      this.state = -1;
    }
    this.result = awaiter.GetResult();
    this.data = Encoding.UTF8.GetString(this.result);
    this._this.LoadData(this.data);
  }
}
```

Best Practices bei der Verwendung von `async`, `await` und `Task`



IPI

Die Intranet-Agentur.

Best Practices bei der Verwendung von `async`, `await` und `Task`

- **Niemals** `Task.Wait()` oder `Task<T>.Result` verwenden

- Der asynchrone Aufruf wird auf einem eigenen Thread ausgeführt
- Der aufrufende Thread blockiert bis der asynchrone Aufruf zurückkehrt
- Exceptions aus der asynchronen Methode werden als `AggregateException` geworfen

- Was ist, wenn der wartende Thread der Main Thread ist?
  - **Herzlich Willkommen, UI-Freeze!**
- Was ist, wenn der asynchrone Aufruf nicht zurückkehren sollte?
  - **Herzlich Willkommen, Deadlocks!**

- Notlösung: `Task.GetAwaiter().GetResult()`
  - Das Blocking-Problem und das zwei Threads gleichzeitig laufen besteht weiterhin
  - Aber: Exceptions aus der asynchronen Methode werden nicht behandelt und so 1:1 an den Aufrufer geworfen

Best Practices bei der Verwendung von `async`, `await` und `Task`

- `Task.ConfigureAwait(false)` verwenden bei `await`-Aufrufen
- `Task.ConfigureAwait(true)` zwingt den Synchronization Context, den nachfolgenden Code auf dem aufrufenden Thread auszuführen
- `Task.ConfigureAwait(true)` ist der Default

- Was ist, wenn der aufrufende Thread der Main-Thread ist?
  - **Herzlich Willkommen, UI-Freeze!**
- Was ist, wenn der aufrufende Thread gerade anderen Code ausführt?
  - **Herzlich Willkommen, Deadlocks!**

- `Task.ConfigureAwait(false)` erlaubt dem Synchronization Context, den nachfolgende Code auf dem nächsten freien Thread auszuführen

Best Practices bei der Verwendung von `async`, `await` und `Task`

- `async void` vermeiden
- „Fire and Forget“ für den Aufrufer
  - Keine Chance für Exception Handling
  - Keine Chance festzustellen, ob die asynchrone Methode ausgeführt ist

- Was ist, wenn die asynchrone void-Operation den Zustand des aktuellen Objekts verändert?
  - **Herzlich Willkommen, Race Conditions!**

Best Practices bei der Verwendung von `async`, `await` und `Task`

- `return await` vermeiden
  - Task durchreichen, den die asynchrone Methode zurückgibt

- Die Verwendung von `await` erzwingt die Verwendung von `async` im Methodenkopf
- `async` wandelt die Methode in eine Klasse um

- Zwei Ausnahmen:
  - In `try-catch`-Blöcken
  - In `using`-Blöcken

IPI

Wie verwendet man asynchronen Code ohne seine (synchrone) Abstraktion zu „zerstören“? - Das Problem

Beispiel Freiwilliger Einbehalt

- Kunde bietet seinen Mitarbeitern an, einen Teil des Gehalts zurückzubehalten
- Mitarbeiter kann jederzeit seine Einbehalts-Modalitäten ändern
- Mitarbeiter kann jederzeit sein Guthaben prüfen
- Mitarbeiter kann sich jederzeit Guthaben auszahlen lassen – ganz oder teilweise

- Mitarbeiterportal bietet die Möglichkeit, diese Funktionen online zu nutzen

Das Problem – Die Domänenlogik

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackRepository repo;

    public decimal GetAccountBalance(Employee e)
        => repo.LoadBalance(e.ID);

    public decimal PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");
        if (repo.LoadBalance(e.ID) - Math.Abs(amount) < 0m)
            throw new Exception("Amount too high");
        return repo.PostAmount(e.ID, amount);
    }
}
```

Beispiel Freiwilliger Einbehalt

- Kunde bietet seinen Mitarbeitern an, einen Teil des Gehalts zurückzubehalten
- Mitarbeiter kann jederzeit seine Einbehalts-Modalitäten ändern
- Mitarbeiter kann jederzeit sein Guthaben prüfen
- Mitarbeiter kann sich jederzeit Guthaben auszahlen lassen – ganz oder teilweise

- Mitarbeiterportal bietet die Möglichkeit, diese Funktionen online zu nutzen

Das Problem – Das Repository wird asynchron

```
public interface IVoluntaryHoldbackRepository
{
    decimal LoadBalance(string employeeID);

    decimal PostAmount(string employeeID, decimal amount);
}
```

```
public interface IVoluntaryHoldbackAsyncRepository
{
    Task<decimal> LoadBalance(string employeeID);

    Task<decimal> PostAmount(string employeeID, decimal amount);
}
```

Beispiel Freiwilliger Einbehalt

- Kunde bietet seinen Mitarbeitern an, einen Teil des Gehalts zurückzubehalten
- Mitarbeiter kann jederzeit seine Einbehalts-Modalitäten ändern
- Mitarbeiter kann jederzeit sein Guthaben prüfen
- Mitarbeiter kann sich jederzeit Guthaben auszahlen lassen – ganz oder teilweise

- Mitarbeiterportal bietet die Möglichkeit, diese Funktionen online zu nutzen

Das Problem – Die Domänenlogik ist betroffen

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackRepository repo;

    public decimal GetAccountBalance(Employee e)
        => repo.LoadBalance(e.ID);

    public decimal PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");
        if (repo.LoadBalance(e.ID) - Math.Abs(amount) < 0m)
            throw new Exception("Amount too high");
        return repo.PostAmount(e.ID, amount);
    }
}
```

Beispiel Freiwilliger Einbehalt

- Kunde bietet seinen Mitarbeitern an, einen Teil des Gehalts zurückzubehalten
- Mitarbeiter kann jederzeit seine Einbehalts-Modalitäten ändern
- Mitarbeiter kann jederzeit sein Guthaben prüfen
- Mitarbeiter kann sich jederzeit Guthaben auszahlen lassen – ganz oder teilweise

- Mitarbeiterportal bietet die Möglichkeit, diese Funktionen online zu nutzen

Das Problem – Die Domänenlogik wird angepasst

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackAsyncRepository repo;

    public async Task<decimal> GetAccountBalance(Employee e)
        => await repo.LoadBalance(e.ID);

    public async Task<decimal> PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");
        if (await repo.LoadBalance(e.ID) - Math.Abs(amount) < 0m)
            throw new Exception("Amount too high");
        return await repo.PostAmount(e.ID, amount);
    }
}
```

Beispiel Freiwilliger Einbehalt

- Kunde bietet seinen Mitarbeitern an, einen Teil des Gehalts zurückzubehalten
- Mitarbeiter kann jederzeit seine Einbehalts-Modalitäten ändern
- Mitarbeiter kann jederzeit sein Guthaben prüfen
- Mitarbeiter kann sich jederzeit Guthaben auszahlen lassen – ganz oder teilweise

- Mitarbeiterportal bietet die Möglichkeit, diese Funktionen online zu nutzen

Das Problem – Die angepasste Domänenlogik vereinfachen

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackAsyncRepository repo;

    public Task<decimal> GetAccountBalance(Employee e)
        => repo.LoadBalance(e.ID);

    public async Task<decimal> PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");
        if (await repo.LoadBalance(e.ID) - Math.Abs(amount) < 0m)
            throw new Exception("Amount too high");
        return repo.PostAmount(e.ID, amount);
    }
}
```

Beispiel Freiwilliger Einbehalt

- Kunde bietet seinen Mitarbeitern an, einen Teil des Gehalts zurückzubehalten
- Mitarbeiter kann jederzeit seine Einbehalts-Modalitäten ändern
- Mitarbeiter kann jederzeit sein Guthaben prüfen
- Mitarbeiter kann sich jederzeit Guthaben auszahlen lassen – ganz oder teilweise

- Mitarbeiterportal bietet die Möglichkeit, diese Funktionen online zu nutzen

Das Problem – Die Abstraktion hat sich geändert

```
public interface IVoluntaryHoldback
{
    decimal GetAccountBalance(Employee e);
    decimal PayOut(Employee e, decimal amount);
}
```

```
public interface IVoluntaryHoldback
{
    Task<decimal> GetAccountBalance(Employee e);
    Task<decimal> PayOut(Employee e, decimal amount);
}
```

IPI

Wie verwendet man asynchronen Code ohne seine (synchrone) Abstraktion zu „zerstören“? - Eine Lösung

Wie macht man bestehenden Code asynchron ohne seine Abstraktion zu „zerstören“?

- Das eigentliche Problem ist nicht die Verwendung von `async` und `await`
- Das eigentlich Problem ist der Rückgabewert `Task` (statt `void`) bzw. `Task<T>` (statt `T`)

- Lösung 1: Hinnehmen!
  Das Drittsystem gibt asynchronen Code vor, also ziehen wir es durch!
  - Frage: Hat ein Drittsystem „das Recht“ uns vorzuschreiben, wie wir programmieren?
  - Mal kurz überlegen … ähhh … NEIN!

- Lösung 2: Asynchronen Code synchronisieren!
  - Konsequenz: wir verlieren die Benefits von Nebenläufigkeit
  - Notwendiges Übel, um die Abstraktion zu behalten

Nochmal zum Problem

```
public interface IVoluntaryHoldback
{
    decimal GetAccountBalance(Employee e);
    decimal PayOut(Employee e, decimal amount);
}
```

```
public interface IVoluntaryHoldback
{
    Task<decimal> GetAccountBalance(Employee e);
    Task<decimal> PayOut(Employee e, decimal amount);
}
```

Wie macht man bestehenden Code asynchron ohne seine Abstraktion zu „zerstören“?

- `async` ist tot, weil das nur drei Rückgabetypen vorschreibt: `Task`, `Task<T>` oder `void`
- `await` ist tot, weil das die Verwendung von `async` im Methodenkopf vorschreibt
- `Task` ist tot, weil das unsere Abstraktion zerstört

- `Task` ist nicht tot!
- `Task` kennt eine Methode, die weiterhilft
  - `.GetAwaiter().GetResult()`

`async` ist tot! `await` ist tot! `Task` ist tot! Und nu?

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackAsyncRepository repo;

    public decimal GetAccountBalance(Employee e)
        => repo.LoadBalance(e.ID).GetAwaiter().GetResult();

    public decimal PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");
        var balance = repo.LoadBalance(e.ID).GetAwaiter().GetResult();
        if (balance - Math.Abs(amount) < 0m)
            throw new Exception("Amount too high");
        return repo.PostAmount(e.ID, amount).GetAwaiter().GetResult();
    }
}
```

Wie macht man bestehenden Code asynchron ohne seine Abstraktion zu „zerstören“?

- Für triviale Einzeiler ist die Lösung akzeptabel
  - `GetAccountBalance()`
- Man hat zwar immer noch Blocking Calls, aber das ist keine Verschlechterung gegenüber den ursprünglichen Entwicklungs-Stand.
- Aber wenn man mehrere asynchrone Aufrufe in einer Methode hat, verliert man DEN Vorteil der `IAsyncStateMachine`:
  - Vermeidung von Kontextwechseln
  - Besser: wenn ich einmal in einem asynchronen Kontext bin, dann bleibe ich auch da bis alle asynchronen Operationen abgeschlossen sind
- Auch hier kennt `Task` eine Methode, die weiterhilft
  - `.ContinueWith()`

Nice! Aber `IAsyncStateMachine` macht es besser!

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackAsyncRepository repo;

    public decimal PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");

        var newBalance = repo.LoadBalance(e.ID)
            .ContinueWith(loadTask => {
                if (loadTask.IsFaulted)
                    throw loadTask.Exception.Flatten()
                                  .InnerExceptions[0];
                if (loadTask.Result - Math.Abs(amount) < 0m)
                    throw new Exception("Amount too high");
                return repo.PostAmount(e.ID, amount)
                           .GetAwaiter().GetResult();
            })
            .GetAwaiter().GetResult();

        return newBalance;
    }
}
```

Wie macht man bestehenden Code asynchron ohne seine Abstraktion zu „zerstören“?

- Die Verwendung von `ContinueWith()` bringt zwei Probleme mit sich:
  - Kein Error Handling
  - Der Rückgabewert der Lambda wird immer in einen neuen `Task` gesteckt
- Konsequenz: Jede Menge Boilerplate-Code!
- Schön wäre eine Methode, die den Boilerplate-Code in sich kapselt, sodass wir uns auf die eigentliche Programmlogik konzentrieren können.
  - Lösung: Eine Extension-Method für `Task`, die einer Lambda im Erfolgsfall `Task.Result` übergibt und den zurückgegebenen `Task` der Nachfolge-Operation direkt durchreicht.
  - Im Fehlerfall wird die Exception in einem passend typisierten `Task` weitergegeben.

Nicer! Aber die `async`/`await`-Syntax ist lesbarer!

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackAsyncRepository repo;

    public decimal PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");

        var newBalance = repo.LoadBalance(e.ID)
            .Bind(oldBalance => {
                if (oldBalance - Math.Abs(amount) < 0m)
                    throw new Exception("Amount too high");
                return repo.PostAmount(e.ID, amount);
            })
            .GetAwaiter().GetResult();

        return newBalance;
    }
}
```

```
public static class TaskExtensions
{
    public static Task<U> Bind<T, U>(this Task<T> t, Func<T, Task<U> f);
}
```

Wie macht man bestehenden Code asynchron ohne seine Abstraktion zu „zerstören“?

- Mit `Bind()` kann man asynchrone Operationen miteinander verketten ohne `async`/`await`, ohne Kontextwechsel, ohne nervigen Boilerplate-Code und mit Built-In-Exception Handling

- Aber es gibt immer noch ein Problem!
  - Was tun, wenn die Kette auch synchrone Operationen enthält?

- Variante 1: Den synchronen Code in einem `Task.Run()` ausführen!
  - Problem: neuer Boilerplate-Code

- Variante 2: Eine weitere Extension-Method für `Task`, die genauso funktioniert wie `Bind()` aber bei der die Lambda-Funktion keine `Task<T>` sondern nur `T` zurückgibt.

Cool! Aber das geht noch besser!

```
public class VoluntaryHoldback : IVoluntaryHoldback
{
    private readonly IVoluntaryHoldbackAsyncRepository repo;

    public decimal PayOut(Employee e, decimal amount)
    {
        if (amount >= 0m)
            throw new Exception("No positive amount");

        var newBalance = repo.LoadBalance(e.ID)
            .Map(oldBalance =>
                oldBalance - Math.Abs(amount) >= 0m)
                ? true : throw new Exception("Amount too high");
            )
            .Bind(canPayOut => repo.PostAmount(e.ID, amount))
            .GetAwaiter().GetResult();

        return newBalance;
    }
}
```

```
public static class TaskExtensions
{
    public static Task<U> Bind<T, U>(this Task<T> t, Func<T, Task<U> f);
    public static Task<U> Map<T, U>(this Task<T> t, Func<T, U> f);
}
```

Wie macht man bestehenden Code asynchron ohne seine Abstraktion zu „zerstören“?

- Mit `Bind()` kann man asynchrone Operationen miteinander verketten ohne `async`/`await`, ohne Kontextwechsel, ohne nervigen Boilerplate-Code und mit Built-In-Exception Handling
- Mit `Map()` kann man synchrone Operationen zu einer Kette asynchroner Operationen hinzufügen ohne `async`/`await`, ohne Kontextwechsel, ohne nervigen Boilerplate-Code und mit Built-In-Exception Handling

- Wenn man die Methoden `Map()` und `Bind()` anders benennt …
  - … nämlich `Select()` und `SelectMany()`
  - … mit Überladungen analog `System.Linq`
- Dann funktioniert `Task` auch mit C#-Query-Syntax

```
Task<bool> canPayOutTask =
    from old in repo.GetAccountBalance()
  select old - amount >= 0;
```


Das Sahnehäubchen!

```
public static class TaskExtensions
{
    public static Task<U> Map<T, U>(
        this Task<T> task,
        Func<T, U> mapping
    ) { ... }
    public static Task<U> Bind<T, U>(
        this Task<T> task,
        Func<T, Task<U> binding
    ) { ... }
}
```

```
public static class TaskExtensions
{
    public static Task<U> Select<T, U>(
        this Task<T> task,
        Func<T, U> selection
    ) { ... }
    public static Task<U> SelectMany<T, U>(
        this Task<T> task,
        Func<T, Task<U> selection
    ) { ... }
}
```

IPI

Matthias Unruhe
Software-Architekt

Telefon +49 (9827) 927 87 - 353
E-Mail m.unruhe@ipi-gmbh.com

IPI GmbH
Untere Industriestraße 5
91586 Lichtenau

Telefon +49 (9827) 927 87 - 0
Telefax +49 (9827) 927 87 - 9000
E-Mail info@ipi-gmbh.com

www.ipi-gmbh.com