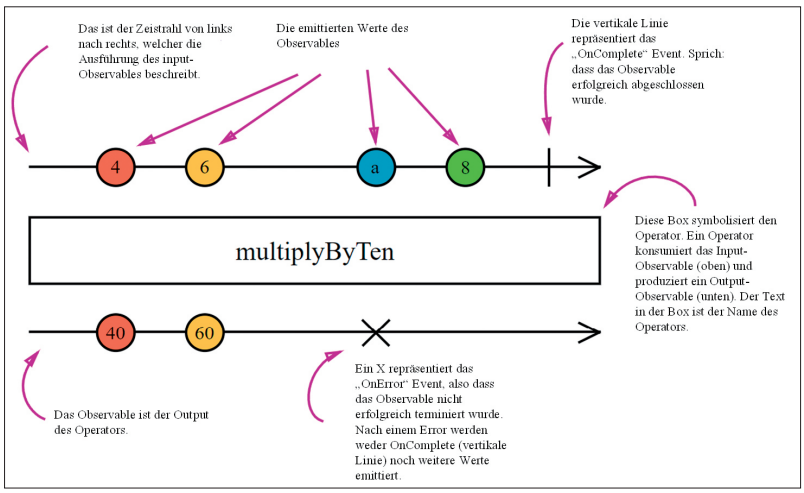
REAKTIV PROGRAMMIEREN, TEIL 2

LINQ für Events

Die wichtigsten Operatoren für die Praxis.

Der erste Teil dieser Artikelserie [1] hat die Open-Source-Bibliothek Rx.NET als Implementierung der Reactive Extensions [2] vorgestellt. Sie ist eine Implementierung des Observer-Design-Patterns, das die Programmierung Push-basierter Benachrichtigungen erleichtert. Rx.NET [3] kann dazu verwendet werden, native .NET-Events zu ersetzen, die mit einigen Schwächen behaftet sind. Dabei wird ein funktionaler Ansatz verfolgt.

Inspiriert vom Objekt *IEnumerable* wird das Objekt *IObservable* verwendet. Dieses informiert über neue Ereignisdaten (*OnNext*), ob es abgeschlossen ist (*OnCompleted*) oder ob ein Fehler aufgetreten ist (*OnError*). Es werden so lange neue Daten per *OnNext* emittiert, bis *OnCompleted* oder *OnError* aufgerufen wird.



Gängige Notation für Observables und Operatoren (Bild 1)

Ein *Observer*-Objekt wird bei einem *IObservable* registriert und verarbeitet die Daten ähnlich wie ein von herkömmlichen Anwendungen bekannter Eventhandler. Ein *Subject* ist beides: sowohl ein *Observable* als auch ein *Observer*. Das bedeutet, dass ein *Subject* sowohl Daten empfangen (*Observer*) als auch Daten an seine Abonnenten weitergeben kann (*Observable*). Deshalb werden *Subjects* in der Praxis genutzt, um Daten manuell zu emittieren.

Zwischen *Observer* und *Observables* befinden sich die modularen Operatoren, die nach dem Vorbild von LINQ beliebig aneinandergereiht werden können. In diesem zweiten Teil der Artikel-Trilogie über Rx.NET sollen die wichtigsten Bausteine zur Programmierung mächtiger Event-Ketten vorgestellt werden.

Operator-Notation

Operatoren lassen sich nur schwer mit Worten beschreiben. Deswegen hat sich eine gängige Diagramm-Notation etabliert, die man häufig in Dokumentationen antrifft. Bild 1 zeigt, wie ein Observable durch einen Operator *multiplyByTen* in ein anderes Observable umgewandelt wird.

Ein Observable wird als Zeitstrahl von links nach rechts dargestellt. Die einzelnen Daten, die per *OnNext* geliefert werden, sind als Elemente auf dem Zeitstrahl aufgetragen (hier *4, 6, a, 8.*). Am Ende eines Observables kann eine vertikale Linie für das *OnComplete*-Ereignis stehen. Alternativ symbolisiert ein *X*, dass es zu einem Fehler kam (*OnError*). Nach *OnComplete* beziehungsweise *OnError* werden keine Daten mehr geliefert. Es gibt auch Observables, die weder *OnComplete* noch *OnError* erzeugen. Diese würden unendlich lang weitere Daten liefern.

Aktiv werden

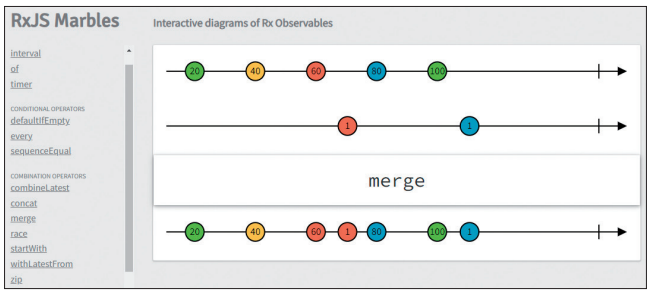
RxMarbles [4] bietet eine schöne Möglichkeit, interaktiv mit Operatoren zu experimentieren. Die Website ist zwar auf RxJS fokussiert, aber die gängigen Operatoren gibt es sowohl in Rx.NET als auch in RxJS. Es lohnt sich daher auch für Rx.NET-Anwender, einen Blick darauf zu werfen. Auf der Website RxJS Marbles können die Events, welche – wie bei der gerade besprochenen Notation – als Kugeln dargestellt werden, per Drag-and-drop zeitlich verschoben werden, und man sieht auf der unteren Leiste in Echtzeit, wie sich das Ergebnis verändert (Bild 2).

So richtig interaktiv wird es mit RxJS-fruits [5] von Gregor Biswanger. Diese Website richtet sich ebenfalls in erster Linie an die JavaScript/TypeScript-Version von Rx, ist aber auch für Rx.NET-Entwickler sehr interessant, um erste Gehversuche zu wagen. In 16 kurzweiligen Levels besteht die Aufgabe darin, Obst (sinnbildlich für Events auf dem Observable) per Operatoren zu filtern, zu manipulieren, zu vereinen et cetera (Bild 3).

Tipp: Auch nach einiger Erfahrung in Rx.NET und dessen Operatoren kann man unerwartete Überraschungen erleben. Es empfiehlt sich, ein kleines Konsolenprojekt zum Experimentieren zu erstellen.

Die wichtigsten Operatoren

In Rx gibt es Dutzende Operatoren, die vergleichbar mit LINQ beliebig hintereinandergereiht werden können. Die Operatoren ermöglichen es, mit nur wenig Code sehr ▶



RxJs Marbles: Website zum interaktiven Experimentieren mit Operatoren (Bild 2)

mächtige Verarbeitungsketten zu definieren. Auf der Website von ReactiveX [6] werden einige dieser Operatoren zusammengetragen. Um alles aus Rx.NET herauszuholen, ist es von Bedeutung, die wichtigsten Operatoren zu kennen und diese sicher einsetzen zu können. In Tabelle 1 finden Sie eine Übersicht der Operatoren. Nachfolgend werden die gemäß der Erfahrung des Autors in der Praxis relevantesten Operatoren vorgestellt. Für alle dabei verwendeten Code-Sequenzen wird die Extension-Methode aus Listing 1 vorausgesetzt.

Where (Find)

Where filtert alle Events mit dem übergebenen Predicate. In Rx-Implementierungen für andere Programmiersprachen wird *Where* [7] oftmals als *Filter* bezeichnet. Da in .NET aber *Where* durch LINQ bereits etabliert war, heißt *Filter* in Rx.NET *Where*.

Achtung: Durch die Bezeichnung *Where* ist Rx.NET zwar konsistent zu LINQ, aber gleichzeitig kann man bei komplexeren Queries, insbesondere dann, wenn man eine Collection aus Observables verarbeitet, nicht mehr klar erkennen, ob es sich um ein *Where* auf einer Collection oder um einen Rx.NET-Operator handelt. Hier empfiehlt es sich, im Code einen Kommentar zu spendieren, wie im folgenden Code-Schnipsel gezeigt wird. Darin wird eine Sequenz der Zahlen von 0 bis 5 nach ungeraden Werten gefiltert:

```
Observable.Range(0, 5)
  .Where(x =>
   x % 2 == 1)
   // IObservable<int>
  .Print();

Ausgabe:
OnNext(1)
OnNext(3)
Completed
```

Das Beispiel verwendet *Observable.Range(start, count)* – ein sogenanntes Cold-Observable (mehr dazu später), das zum Zeitpunkt der Registrierung Werte ab dem Parameter *start* mit der Anzahl *count* ausgibt und abschließend *completed*.

Select (Map)

Select projiziert alle Events mit der übergebenen Transformations-Funktion. In anderen Rx-Implementierungen wird *Select* oftmals als *Map* [8] bezeichnet. Da in .NET aber *Select* durch LINQ bereits etabliert war, heißt *Map* auch in Rx.NET *Select*.

Achtung: Analog zum Hinweis zu *Where* ist es auch bei *Select* ratsam, einen Kommentar anzufügen. Im folgenden Beispiel wird eine Sequenz der Zahlen von 0 bis 5 mit 10 multipliziert.

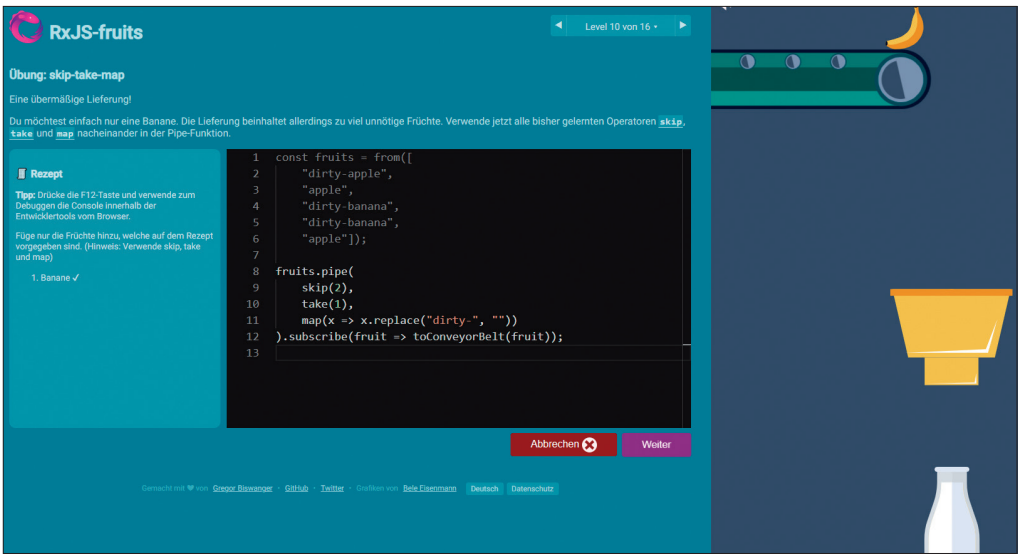
```
Observable.Range(0, 5)
  .Select(x => x * 10) // IObservable<int>
  .Print();

Ausgabe:
OnNext(0)
OnNext(10)
OnNext(20)
OnNext(30)
OnNext(40)
Completed
```

CombineLatest

CombineLatest ist ein sehr nützlicher und in der Praxis häufig anzutreffender Operator. Er dient dazu, mehrere Observables in einem einzigen Observable zusammenzufassen, vergleiche Bild 4. Dieser Fall tritt auf, wenn man mehrere Datenquellen verarbeitet, welche unabhängig voneinander Informationen veröffentlichen.

Im nachfolgenden Code-Schnipsel werden zunächst ein Temperatur- und ein Luftfeuchtigkeitssensor simuliert und die Ergebnisse anschließend miteinander in einem String kombiniert. Aus Platzgründen wird hier darauf verzichtet, den Heat-Index zu berechnen.



RxJS-fruits: Die Website bietet Übungsmöglichkeiten im Umgang mit Operatoren (Bild 3)

Tabelle 1: Übersicht der Operatoren

Erzeugende Operatoren	
Return	Erstellt ein Observable, das einen einzigen Wert liefert und dann abgeschlossen wird.
Empty	Erstellt ein Observable, das sofort abgeschlossen wird, ohne einen Wert zu liefern.
Never	Erstellt ein Observable, das nie Werte liefert und nie abgeschlossen wird.
Throw	Erstellt ein Observable, das sofort einen Fehler ausgibt.
Interval	Gibt periodisch aufeinander folgende ganze Zahlen in einem angegebenen Intervall aus.
Timer	Gibt einen Wert nach einer Verzögerung aus und kann optional periodisch Werte ausgeben.
Range(Start, Count)	Erstellt ein Observable, das die in *Count* festgelegte Anzahl Werte ab *Start* ausgibt.
Filternde Operatoren	
Where	Filtert Elemente basierend auf einer Bedingung.
Distinct	Unterdrückt aufeinander folgende doppelte Elemente.
DistinctUntilChanged	Unterdrückt aufeinander folgende doppelte Elemente, erlaubt jedoch unterschiedliche aufeinander folgende Werte.
Take	Gibt die ersten n Elemente aus einer Sequenz zurück.
Skip	Überspringt die ersten n Elemente aus einer Sequenz.
TakeWhile	Gibt Elemente aus, solange eine Bedingung wahr ist.
SkipWhile	Überspringt Elemente, solange eine Bedingung wahr ist.
Transformierende Operatoren	
Select	Projiziert jedes Element einer Sequenz.
SelectMany	Projiziert jedes Element einer Sequenz in eine Observable-Sequenz und gibt die abgeflachten Ergebnisse zurück. Tipp: Eine Überladung erlaubt es, für jedes Element einen Task auszuführen und die Ergebnisse in Form eines Observables zu verarbeiten.
Buffer	Gruppiert Elemente in Arrays der angegebenen Größe.
Window	Teilt eine Observable-Sequenz in mehrere kleinere Observable-Sequenzen auf.
Kombinierende Operatoren	
Merge	Kombiniert mehrere Observable-Sequenzen in eine einzige.
Zip	Kombiniert mehrere Observable-Sequenzen, indem es jeweils ein Element aus jeder Sequenz in ein Element zusammenfasst.
CombineLatest	Kombiniert die neuesten Werte aus mehreren Observable-Sequenzen.
Join	Kombiniert Elemente aus zwei Observable-Sequenzen basierend auf Selektoren.
GroupJoin	Kombiniert Elemente aus zwei Observable-Sequenzen basierend auf einem Schlüssel und gruppiert die Ergebnisse.
Switch	Wechselt zu einer neuen Observable-Sequenz, wenn eine neue Observable-Sequenz emittiert wird, und deregistriert die vorherige.
Fehlerbehandlung	
OnErrorResumeNext	Wirft ein Observable eine Exception, wird mit dem nachfolgenden Observable fortgefahren.
Catch	Verhält sich wie *OnErrorResumeNext*, bietet aber die Möglichkeit, auf den Fehler zu reagieren.
Retry	Wiederholt die Ausführung eines Observables bei Fehlern so lange, bis sie erfolgreich beendet ist.
Zeitverhalten	
Interval	Erzeugt eine Sequenz, die nach jedem Zeitraum einen aufsteigenden Wert produziert.
Delay	Verschiebt die beobachtbare Sequenz um die angegebene Zeitdauer. Die Zeitabstände zwischen den Werten bleiben erhalten.
Throttle	Verwirft Elemente so lange, bis eine gewisse Zeit keine Elemente ausgegeben werden (Beispiel: Autocomplete)
Nebenläufigkeit	
SubscribeOn	Bestimmt, auf welchem Scheduler die Logik des Registrierens und Deregistrierens ausgeführt wird.
ObserveOn	Bestimmt, auf welchem Scheduler Beobachtungen empfangen werden.
SelectMany	Kann dazu genutzt werden, um Tasks für jedes Element in der Sequenz auszuführen.

```
var temperatureSensor = new Subject<double>();
var humiditySensor = new Subject<double>();

Observable
  .CombineLatest(temperatureSensor,
     humiditySensor)
  .Select(x => $"temperature = {x[0]}°C ;
     humidity = {x[1]}%")
  .Print();

temperatureSensor.OnNext(35);
temperatureSensor.OnNext(28);

humiditySensor.OnNext(45);
humiditySensor.OnNext(55);

temperatureSensor.OnNext(25);
temperatureSensor.OnCompleted();
humiditySensor.OnCompleted();

Ausgabe:
OnNext(temperature = 28°C ; humidity = 45%)
OnNext(temperature = 28°C ; humidity = 55%)
OnNext(temperature = 25°C ; humidity = 55%)
Completed
```

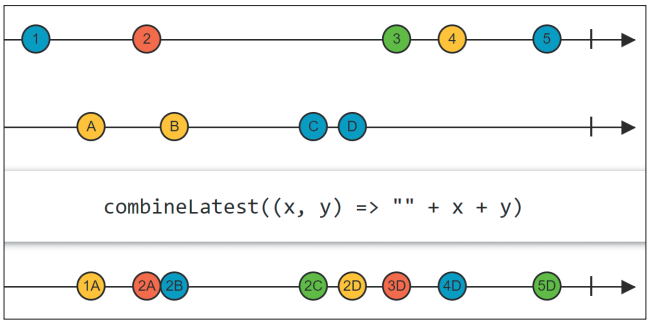
Der erste Temperaturwert (*35*) wird hier nicht ausgegeben, da der Operator *CombineLatest* so lange wartet, bis alle Observables mindestens einen Wert geliefert haben.

Wie findet man den passenden Operator?

Es ist wichtig, den zur Aufgabe passenden Operator zu finden, um Rx.NET maximal effizient einsetzen zu können. Hierzu gibt es für Rx.JS einen interaktiven Entscheidungsbaum [9], mit dem man durch Beantwortung einfacher Fragen den richtigen Operator finden kann. Ein nicht interaktiver Entscheidungsbaum existiert auch für Rx [10].

Listing 1: PrintExtension

```
public static class PrintExtension
{
  public static void Print<T>(
      this IObservable<T> source)
  {
    source.Subscribe(
      value => Console.WriteLine(
      $"OnNext({value})"),
      ex => Console.WriteLine(
      $"OnError() lead to exception:
      {ex.Message}"),
      () => Console.WriteLine("Completed"));
  }
}
```



Schematische Darstellung von CombineLatest (Bild 4)

Achtung: Bei beiden Angeboten ist zu beachten, dass man in der Dokumentation den entsprechenden Operator für Rx.NET verwendet. Gerade Rx.NET weicht von der Rx-Namensgebung einige Male ab, wie bereits bei *Map* und *Select* gesehen.

Hot- und Cold-Observables

Ein sehr wichtiges Konzept ist die Unterscheidung zwischen Hot- und Cold-Observables, auch Hot- und Cold-Sources genannt. Diese Eigenschaft beschreibt, wie sich das Observable im Zusammenspiel mit dem Subscriber verhält. Da *OnNext()* die Daten zeitlich verteilt an das Observable liefert, kann es relevant sein, zu welchem Zeitpunkt sich der Subscriber am Observable registriert.

Registriert sich der Observer bei einem Hot-Observable, bekommt der Subscriber neue Daten erst ab dem Zeitpunkt, zu dem er sich registriert hat. Das bedeutet im Umkehrschluss, dass alle Daten vor der Registrierung für den Subscriber verloren sind.

Bei Cold-Observables sieht es anders aus. Sie können Informationen intern speichern und beispielsweise zum Zeitpunkt der Registrierung ausgeben.

Hot-Observables

Typische Beispiele für Hot-Observables sind aktuelle Sensordaten, Click-Streams oder Börsenticker. Hot-Observables taugen also immer dann, wenn man nur an zukünftigen Ereignissen interessiert ist.

Das folgende Code-Beispiel simuliert einen Temperatursensor, der kontinuierlich Daten liefert. Der Subscriber registriert sich hier erst, nachdem die Temperaturen *24°C* und *28°C* emittiert wurden. Deswegen werden nur die danach folgenden Temperaturen *30°C* und *35°C* ausgegeben.

```
Subject<int> temperatureSubject = new Subject<int>();
// let's emit some values
temperatureSubject.OnNext(24);
temperatureSubject.OnNext(28);

// now we register
temperatureSubject.Select(x =>
  $"Aktuelle Temperatur: {x}°C").Print();

// continue comitting data
temperatureSubject.OnNext(30);
temperatureSubject.OnNext(35);
```

Listing 2: Await und Observables

```
var rangeObservable = Observable.Range(0, 5);

// Liefert das erste Element der Observable-Sequenz.
int result1 = await rangeObservable.FirstAsync();

// Liefert das erste Element der Observable-
// Sequenz oder einen Default-Wert.
int result2 =
  await rangeObservable.FirstOrDefaultAsync();

// Liefert das letzte Element der Observable-Sequenz.
// Das ist der Standardfall, wenn man await benutzt.
int result3_1 = await rangeObservable.LastAsync();
int result3_2 = await rangeObservable;

// Liefert das letzte Element der Observable-
// Sequenz oder einen Default-Wert.
int result5 =
  await rangeObservable.LastOrDefaultAsync();

// Ruft eine Aktionsmethode für jedes Element in
// der Observable-Sequenz auf und gibt einen Task
// zurück, der signalisiert, dass die Sequenz
// beendet ist.
await rangeObservable.ForEachAsync(
  x => Console.WriteLine(x));

var singleObservable = Observable.Return(1);
// Gibt das einzige Element der Observable-
// Sequenz zurück oder wirft eine Exception.
int result6 =
  await singleObservable.SingleAsync();

// Gibt das einzige Element der Observable-
// Sequenz oder einen Default-Wert zurück.
// Gibt es mehr als ein Element, wird eine
// Exception geworfen.
int result7 =
  await singleObservable.SingleOrDefaultAsync();
```

Bei Hot-Observables teilen sich alle Subscriber denselben Datenstrom. Ein späterer Abonnent verpasst somit all diejenigen Ereignisse, die schon aufgetreten sind, bevor er sich registriert hat.

Cold-Observables

Cold-Observables sind immer dann die richtige Wahl, wenn man unabhängig vom Zeitpunkt der Registrierung sein muss. Möchte man sich irgendwann in der Zukunft beim Temperatur-Sensor registrieren und alle vorherigen Temperaturwerte geliefert bekommen, geht das wie folgt:

```
ReplaySubject<int> temperatureSubject =
  new ReplaySubject<int>();

// let's emit some values
temperatureSubject.OnNext(24);
temperatureSubject.OnNext(28);

// now we register
temperatureSubject.Select(
  x => $"Aktuelle Temperatur: {x}°C").Print();

// continue comitting data
temperatureSubject.OnNext(30);
temperatureSubject.OnNext(35);
```

Das Beispiel verwendet anstelle eines *Subjects* ein *ReplaySubject*. Dieses speichert alle bereits emittierten Werte ab. Sobald sich ein Abonnent bei einem *ReplaySubject* registriert, werden alle bereits emittierten Werte ausgegeben und danach alle zukünftigen Daten. Das Ergebnis hier wäre also *24°C, 28°C, 30°C, 35°C,* unabhängig davon, dass der Observer sich erst registriert hat nachdem die Werte *24°C* und *28°C* emittiert wurden.

Cold-Observables erzeugen für jeden Abonnenten einen eigenen, unabhängigen Stream.

Mischform

In der Realität gibt es oftmals kein Schwarz und Weiß. Angenommen, man benötigt nach dem Registrieren beim Temperatur-Sensor sofort den aktuellen (also zuletzt gemessenen Wert) und ab diesem Zeitpunkt alle zukünftigen Messungen. Wie ist das einzuordnen?

Der Fakt, dass man sofort die aktuelle Temperatur bekommt, würde für ein Cold-Observable sprechen. Da es sich aber nur auf den letzten Wert und nicht auf die gesamte Historie bezieht, ist es weder ein reines Hot- noch ein Cold-Observable. Hier hilft ein *BehaviorSubject*:

```
BehaviorSubject<int?> temperatureSubject =
  new BehaviorSubject<int?>(null);

// let's emit some values
temperatureSubject.OnNext(24);
temperatureSubject.OnNext(28);

// now we register
temperatureSubject.Select(x =>
  $"Aktuelle Temperatur: {x}°C").Print();

// continue committing data
temperatureSubject.OnNext(30);
temperatureSubject.OnNext(35);
```

▶

Das *BehaviorSubject* in diesem Beispiel merkt sich den zuletzt emittierten Wert und gibt diesen unmittelbar bei der Registrierung eines Observers aus. BehaviorSubjects müssen mit einem initialen Wert erzeugt werden, daher der Parameter *null*. Die Ergebnisse in diesem Beispiel wären *28°C, 30°C sowie 35°C*. BehaviorSubjects sind äußerst nützlich, wenn man den aktuellen (zuletzt gemessenen) Sensorwert unmittelbar benötigt und nicht erst darauf warten kann, bis der nächste Sensorwert aufgenommen wird.

Observables in Tasks konvertieren

In .NET modelliert man nebenläufige Aktionen am besten mit Tasks, auf deren Ergebnis man mit *await* wartet. Rx.NET bietet eine Menge Wege an [11], um aus der Observable-Welt in die Task-Welt zu gelangen. Listing 2 zeigt einige Beispiele.

Tasks in Observables konvertieren

Ebenso gibt es mehrere Wege, um aus der Task-Welt in die Observable-Welt zu kommen [12]. Hierbei ist es entscheidend zu bedenken, welches Ausführungsverhalten man erzielen möchte, wenn sich ein Subscriber beim Observable registriert. Siehe dazu den Abschnitt zu Hot- und Cold-Observables.

Fall 1: Ein Task soll exakt einmal starten, und unabhängig vom Zeitpunkt der Registrierung soll das Ergebnis im Observable emittiert werden. Jeder Subscriber erhält das Ergebnis, sobald es vorliegt.

```
var observable = Task.Run(() =>
{
  Console.WriteLine("I am executing");
  return 0;
}).ToObservable();
observable.Print();
observable.Print();

Ausgabe:
I am executing
OnNext(0)
Completed
OnNext(0)
Completed
```

Daran, dass *I am executing* nur einmal in der Ausgabe erscheint, sieht man, dass der Task nur einmal ausgeführt wurde. Beide Subscriber erhalten dasselbe Ergebnis.

Fall 2: Jedes Mal, wenn sich ein Subscriber registriert, soll ein neuer, separater Task gestartet werden (Cold-Observable).

```
var observable = Observable.FromAsync(() =>
{
  Console.WriteLine("I am creating a new Task");
  return Task.Run(() =>
  {
    Console.WriteLine("I am executing");
    return 0;
  });
});
observable.Print();
observable.Print();

Ausgabe:
I am creating a new Task
I am executing
OnNext(0)
Completed
I am creating a new Task
I am executing
OnNext(0)
Completed
```

Observable.FromAsync erwartet eine Task-Factory als Parameter. Diese Task-Factory wird jedes Mal aufgerufen, wenn sich ein neuer Subscriber beim Observable registriert.

Listing 3: Task und Observables kombiniert

```
Task<string> download1 =
  Task.Run(async () =>
  { await Task.Delay(1000);
  return "Hallo"; });

Task<string> download2 =
  Task.Run(async () =>
  { await Task.Delay(2000);
  return "Welt"; });

var downloadResult =
  await Task
  .WhenAll(download1, download2)
  // Operation auf Task<string>[]

  .ToObservable()
  // Umwandlung in Observable<string[]>

  .Timeout(TimeSpan.FromSeconds(10))
  // Wirft eine Execption, wenn die Observable-
  // Sequenz nicht innerhalb von 10 Sekunden
  // terminiert.

  .FirstAsync();
  // Gibt ein Array mit zwei Elementen und den
  // jeweiligen Downloads zurück.

Console.WriteLine(
  $"{downloadResult[0]} {downloadResult[1]}");
```

Hin und zurück

Kombiniert man Tasks und Observables, kann man mächtige, nebenläufige Event-Ketten verarbeiten. Das in Listing 3 gezeigte Beispiel simuliert zwei Downloads, die beide klassisch als Task modelliert sind. Die Aufgabe besteht darin, beide

Downloads parallel zu starten, maximal zehn Sekunden zu warten und dann entweder eine Exception zu werfen oder das Ergebnis zurückzugeben.

Im Beispiel werden die beiden Tasks in ein Objekt vom Typ *Observable<string[]>* umgewandelt, dann mit dem Timeout-Operator [13] kombiniert und das Ergebnis als *downloadResult*-Task zurückgegeben.

Fazit und Ausblick

In dieser zweiten von drei Serienfolgen wurden Operatoren und deren Verwendung mit Rx.NET vorgestellt. Operatoren sind die wesentliche Zutat, die es Entwicklern erlaubt, mit Rx.NET effiziente Datenverarbeitungsketten zu gestalten. Zusätzlich wurden Hot- und Cold-Observables erläutert und gezeigt, wie man diese durch die Verwendung verschiedener *Subject*-Typen erzeugt. Den Abschluss bildetet der Übergang von der Observable- in die Task-Welt und zurück.

Die kommende dritte und letzte Folge der Artikelreihe wird sich mit fortgeschrittenen Themen wie Threading, Scheduling und Testen beschäftigen. ■

[1] Tim Borowski, Reaktiv programmieren, Teil 1, Einführung in Rx.NET, dotnetpro 10/2024, Seite 51 ff., www.dotnetpro.de/A2410RxNET

[2] ReactiveX, https://reactivex.io

[3] Reactive Extensions for .NET, www.dotnetpro.de/SL2411RxNET1

[4] RxJS Marbles, https://rxmarbles.com

[5] RxJS-fruits, https://www.rxjs-fruits.com

[6] Operators in der Rx-Dokumentation, www.dotnetpro.de/SL2411RxNET2

[7] Filter-Operator, www.dotnetpro.de/SL2411RxNET3

[8] Map-Operator, www.dotnetpro.de/SL2411RxNET4

[9] Entscheidungsbaum für RxJS, www.dotnetpro.de/SL2411RxNET5

[10] Entscheidungsbaum für Rx, www.dotnetpro.de/SL2411RxNET6

[11] Weitere Informationen zu Observables, www.dotnetpro.de/SL2411RxNET7

[12] Weitere Informationen zum Konvertieren von Tasks, www.dotnetpro.de/SL2411RxNET8

[13] Timeout in der Rx-Dokumentation, www.dotnetpro.de/SL2411RxNET9

Tim Borowski
ist freiberuflicher Softwareentwickler und –berater. Seine Schwerpunkte sind die .NET/C#–Entwicklung, Angular sowie AWS, K8s und Istio Service Mesh. Zudem ist er Fachartikelautor und Konferenzsprecher. Sie erreichen ihn unter **tim@borowski-software.de**.

dnpCode A2411RxNET