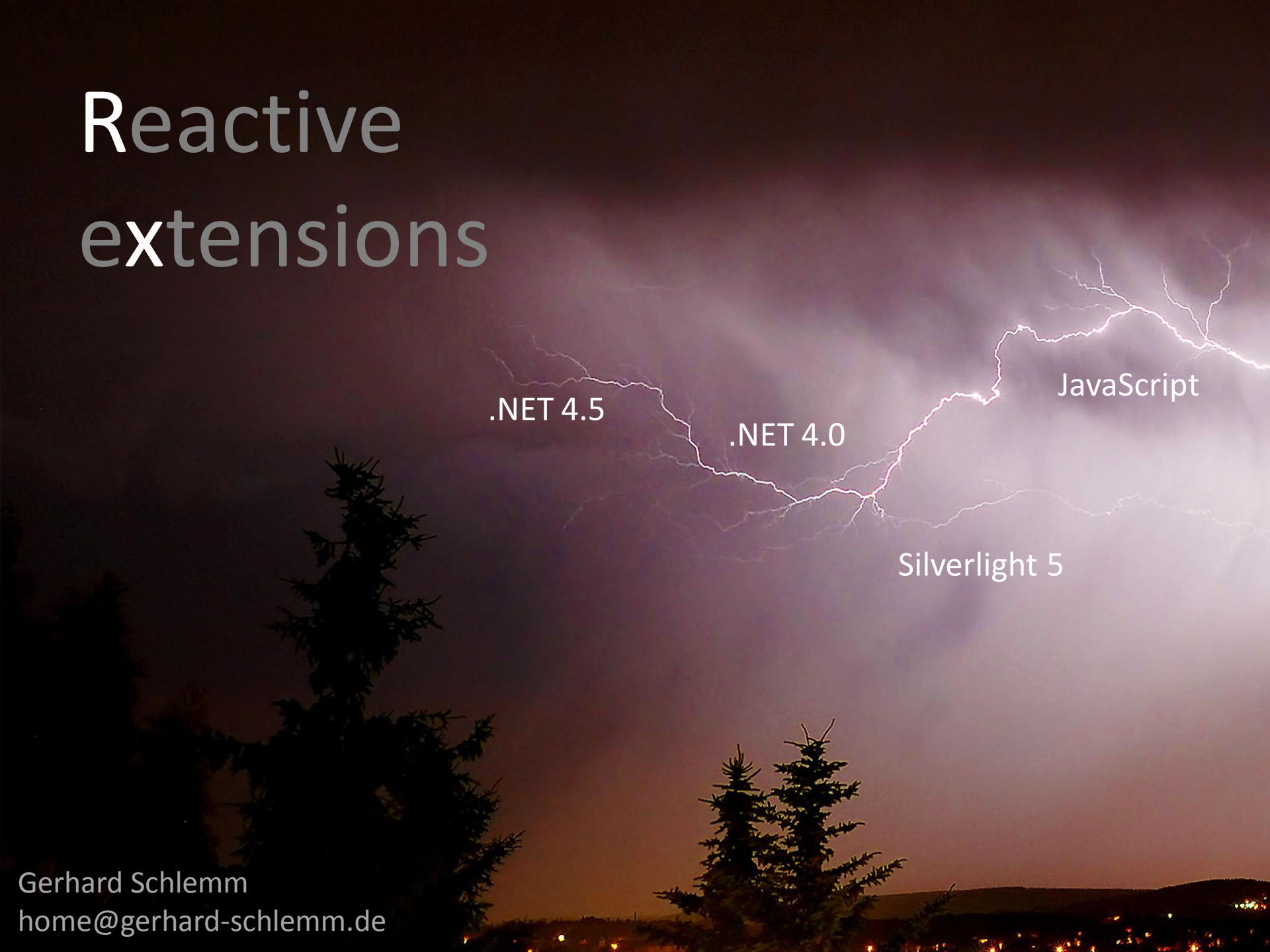
# Reactive extensions




Gerhard Schlemm
home@gerhard-schlemm.de

# Rx

"The Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. Using Rx, developers ***represent*** asynchronous data streams with **Observables**, ***query*** asynchronous data streams using LINQ **operators**, and ***parameterize*** the concurrency in the asynchronous data streams using **Schedulers**. Simply put, Rx = Observables + LINQ + Schedulers." (Microsoft)

http://rx.codeplex.com/

# Hier vorgestellt

- **Reactive Extension** SDK 2.1, Februar 2013
  - .NET Framework 4
  - .NET Framework 4.5
  - .NET Framework 4.5 for Windows Store apps
  - Silverlight 5
  - Windows Phone 7.1
  - Windows Phone 8

# Hier vorgestellt

- Open Source (Apache License 2.0) seit November 2012
- Vorangetrieben durch Microsoft
  - Von Microsoft aktiv mit vielen Konferenz-vorträgen unterstützt (siehe Channel9)
  - Rege Projektaktivität - etwa zeitnaher Release für .NET 4.5

# Agenda

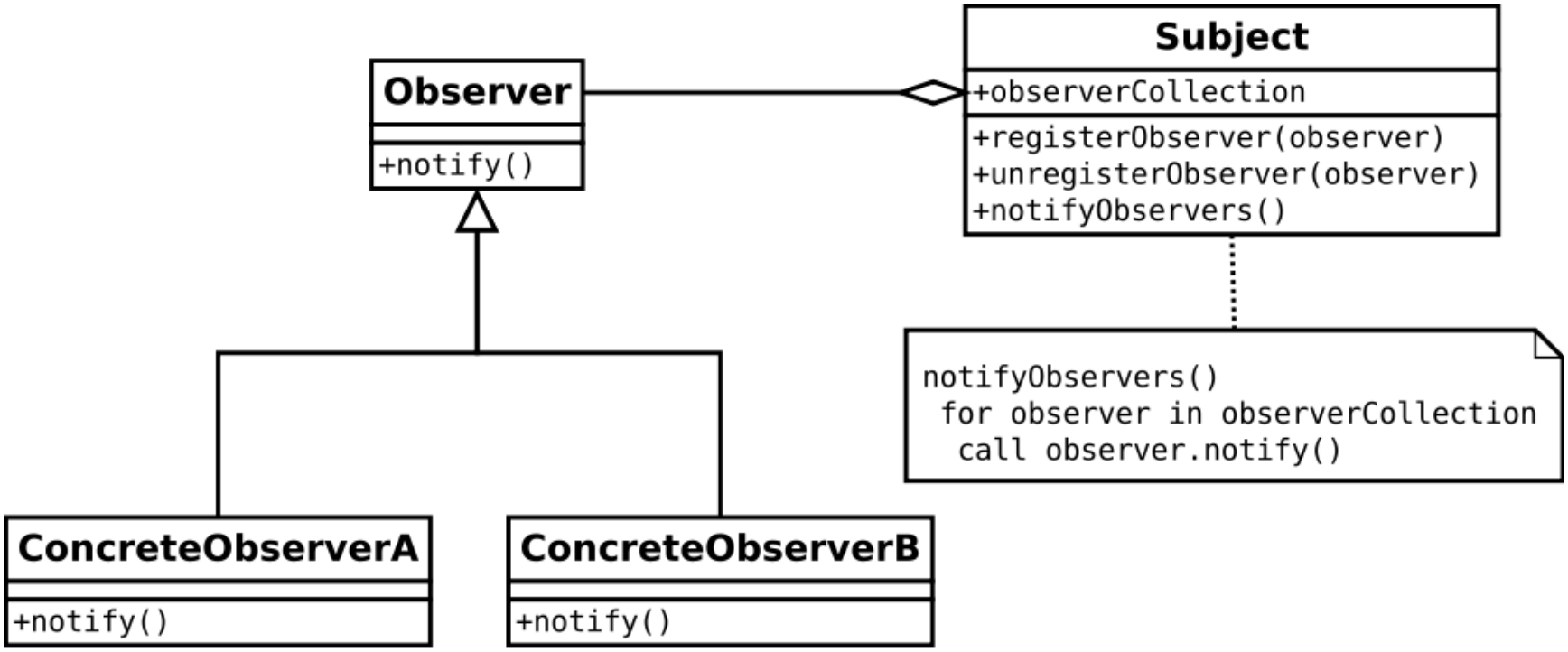
- Das Observer-Pattern
- Observer@Rx
  - Was ist Rx | Rx Beyond Observer | Beispiel 0
- Rx im Detail
  - Operatoren | Klassen | Scheduler | Testing
- Verwendung
- Fazit

# Observer

# Observer

- Behavioural Pattern
- Akteure: **Subject** und (viele) **Observer**
  - Observer können sich am Subject für Benachrichtigungen registrieren und deregistrieren
  - Ändert das Subject etwa seinen State, so benachrichtigt es registrierte Observer
    - Entweder der Observer erhält den geänderten State als Teil der Benachrichtigung (Push) oder
    - Observer ermittelt nach Benachrichtigung hin den geänderten State selektiv (Pull)

# Observer


Observer
+notify()
Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
notifyObservers()
for observer in observerCollection
call observer.notify()
ConcreteObserverA
+notify()
ConcreteObserverB
+notify()

# Observer

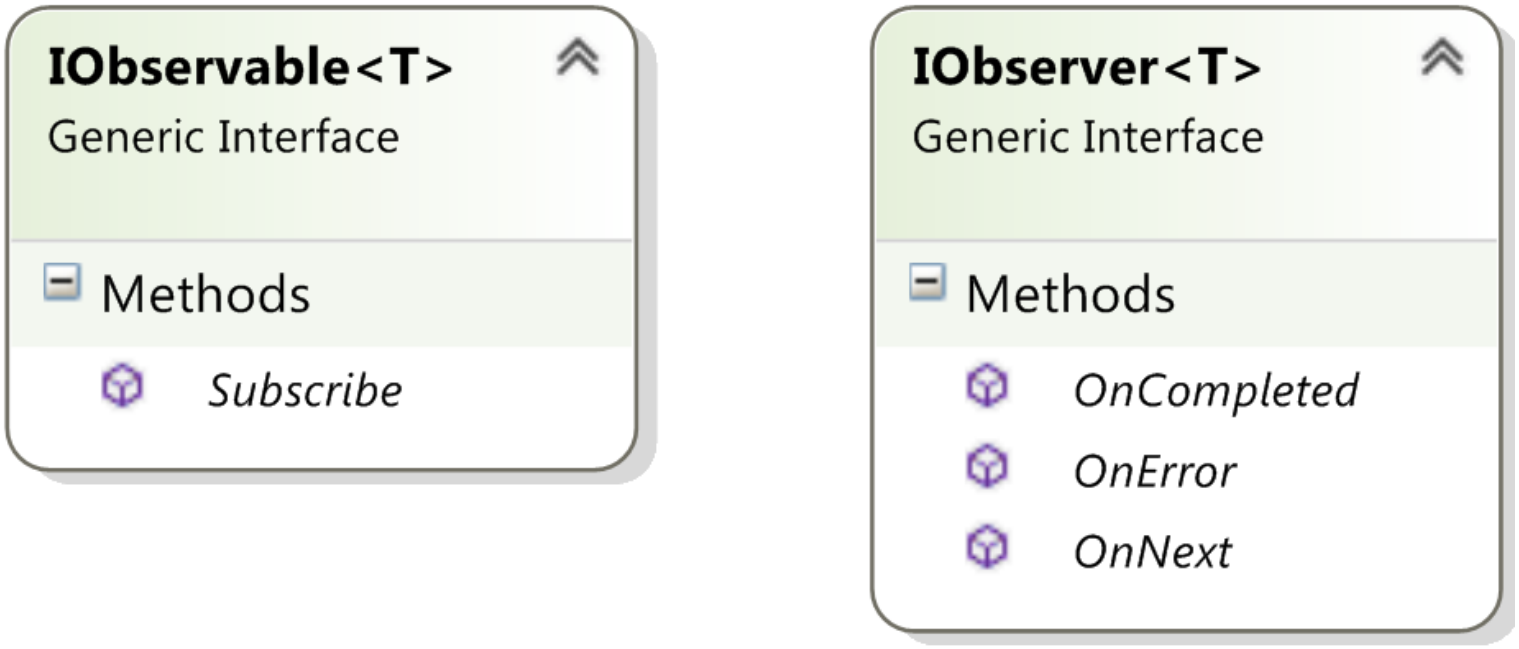
- Ziel: Aufbrechen der engen Kopplung zwischen Subject und Observer
  - z.B. für sauberes Layering (Beispiel: MVVM)
  - Einfacher: Kontrakt
    - Subject: `registerObserver`, `unregisterObserver`
    - Observer: `notify`
- Häufig in Multicast-Szenarien eingesetzt
- Pattern auch bekannt als Publish-Subscribe


Observer@Rx

# Observer@Rx

- Rx definiert wenige, sehr einfache Schnittstellen aus dem Observer-Pattern
  - `IObservable<T>`
    - Etwas, das observiert werden kann `Subscribe(IObserver<T>)` (registerObserver) Ergebnis von `Subscribe` ist ein `IDisposabe` (`Dispose()` als unregisterObserver)
  - `IObserver<T>`
    - Eine Instanz, die Benachrichtigungen vom IObservable entgegennimmt

# Observer@Rx


IObservable<T>
Generic Interface
Methods
Subscribe
IObserver<T>
Generic Interface
Methods
OnCompleted
OnError
OnNext

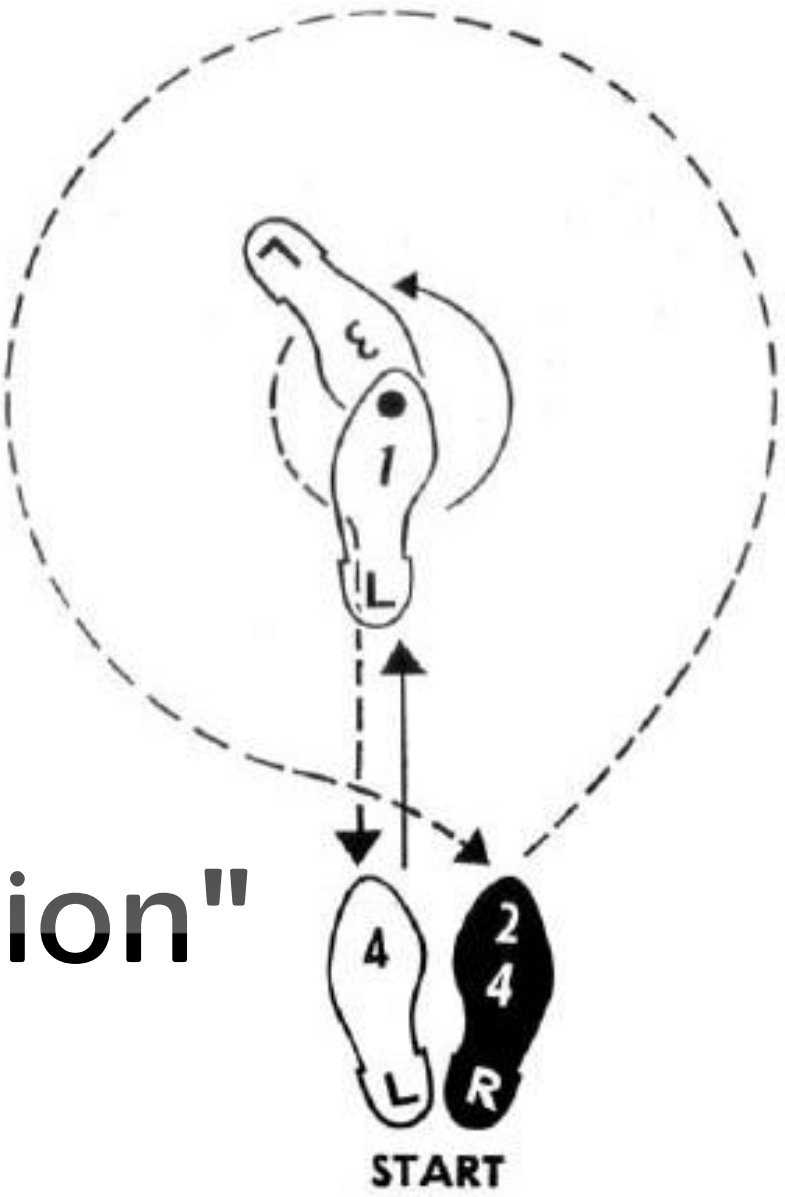
# Observer@Rx

- `OnNext(T data)`
  - Observer erhält neue Daten
  - Wenn kein T erwünscht, `Unit` verwenden
- `OnError(Exception)`
  - Einheitliche Fehlerbehandlung
- `OnComplete()`
  - Sequenz der Benachrichtigungen beendet
  - Was ist die Semantik dieser Sequenz?

# Observer@Rx

- Wesentlich: Die Semantik der Sequenz, die ein IObservable liefert
  - Strom (unkorrelierter) Ereignisse / Daten
    - Sensordaten: Gesten, Maus, Tastatur, GPS, Beschleunigung, fortlaufende Kursdaten für einen Börsenwert.
      Property-Charakter:  OnComplete kommt möglicherweise nie
    - WMI-Ereignisse, TFS Build Notifiaction, "LogonCompleted"
  - Alle Ereignisse bilden eine semantisch zusammenhängende Einheit
    - Teure Kennzahlen für Börsenwerte werden nach und nach angeliefert: "Future Collections"

Rx ist
"Data in Motion"



# Observer@Rx

- In Design Pattern-Büchern gibt es viele Beispiele mit Events zur Benachrichtigung
- Events sind kein "first class citizens" in .NET
  - Können nur aufwändig weitergegeben werden (add/remove)
- IObservable und IObserver können problemlos überall hin transferiert werden!

# Rx Beyond Observer

- Erwartungen an ein Framework:
  - Schneller Code verstehen und entwickeln durch bekannte Schnittstellen

- Aber was sind die wirklichen Stärken von Rx?

# Rx Beyond Observer

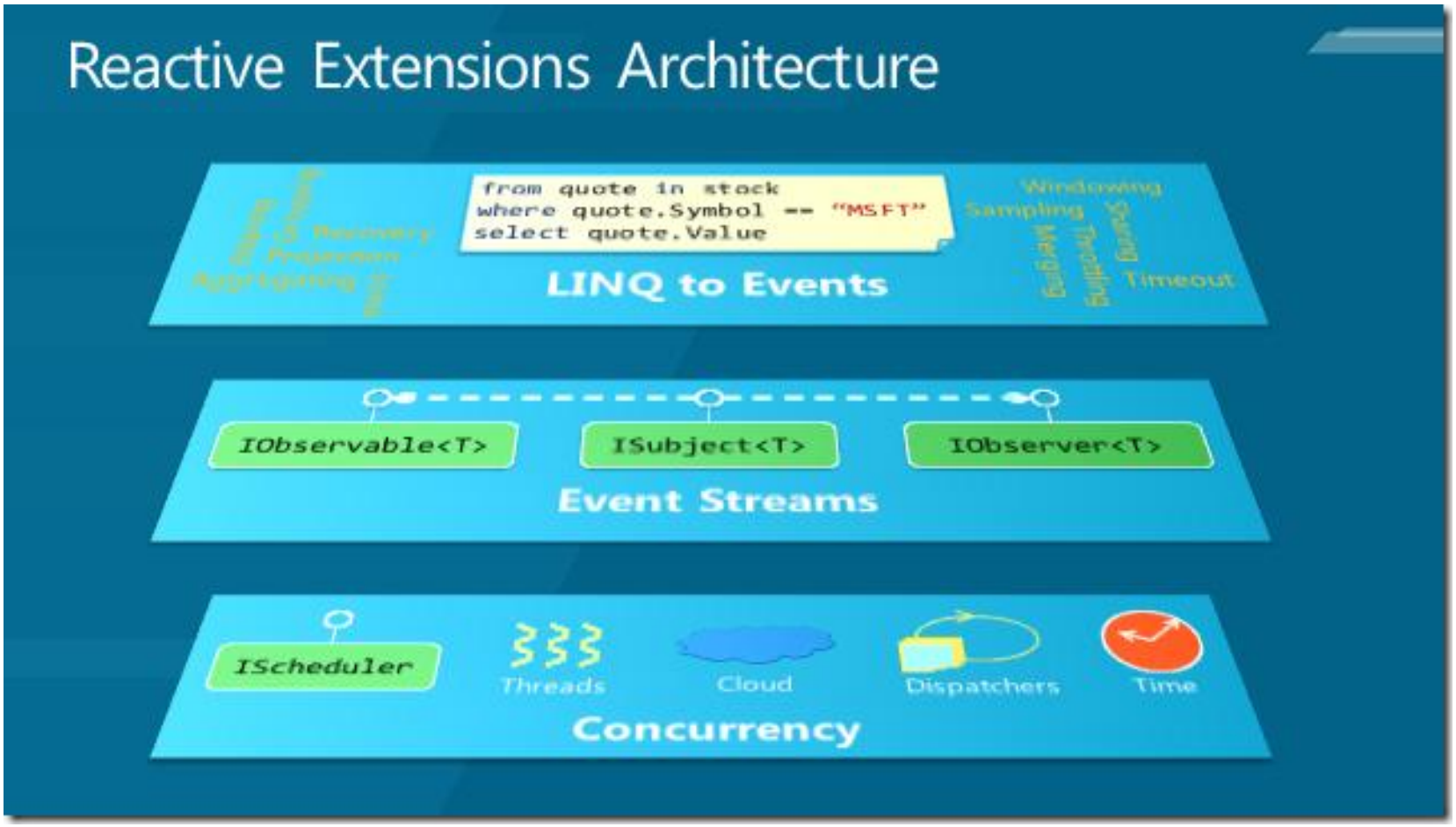
"The Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. Using Rx, developers ***represent*** asynchronous data streams with **Observables**, ***query*** asynchronous data streams using LINQ **operators**, and ***parameterize*** the concurrency in the asynchronous data streams using **Schedulers**. Simply put, Rx = Observables + LINQ + Schedulers." (Microsoft)

http://rx.codeplex.com/

# Rx Beyond Observer

- in breve
  - *"represent* asynchronous data streams with observables"
    - Framework-Implementierungen der Schnittstellen für viele Situationen
  - "*query* asynchronous data streams"
    - LINQ to Events
  - "*parametrize* concurrency using schedulerers"

# Rx Beyond Observer




http://blogs.msdn.com/b/rxteam/archive/2012/06/14/testing-rx-queries-using-virtual-time-scheduling.aspx

FileSystemWatcher

# BEISPIEL 0

# Beispiel 0

- FileSystemWatcher erzeugen und dessen Ereignisstrom in ein IObservable überführen
  - Pattern: Subject ist FileSystemWatcher
- Observer erzeugen, Ereignisse des IObservable subskribieren und verarbeiten
  - Pattern: Observer, registerObserver, notify
- Observer deregistrieren
  - Pattern: unregisterObserver

# Beispiel 0

```
var watcher = new FileSystemWatcher(BaseDirectory, "*.txt")
    {
        EnableRaisingEvents = true, IncludeSubdirectories = true,
        NotifyFilter = NotifyFilters.FileName
    };

IObservable<EventPattern<FileSystemEventArgs>> created = Observable.
    FromEventPattern<FileSystemEventHandler, FileSystemEventArgs>(
        handler => watcher.Created += handler,
        handler => watcher.Created -= handler);

IDisposable subscriptionDisposable = created.Subscribe(
    data => Console.Out.WriteLine("created " + data.EventArgs.FullPath));

/* ... */

subscriptionDisposable.Dispose();
```

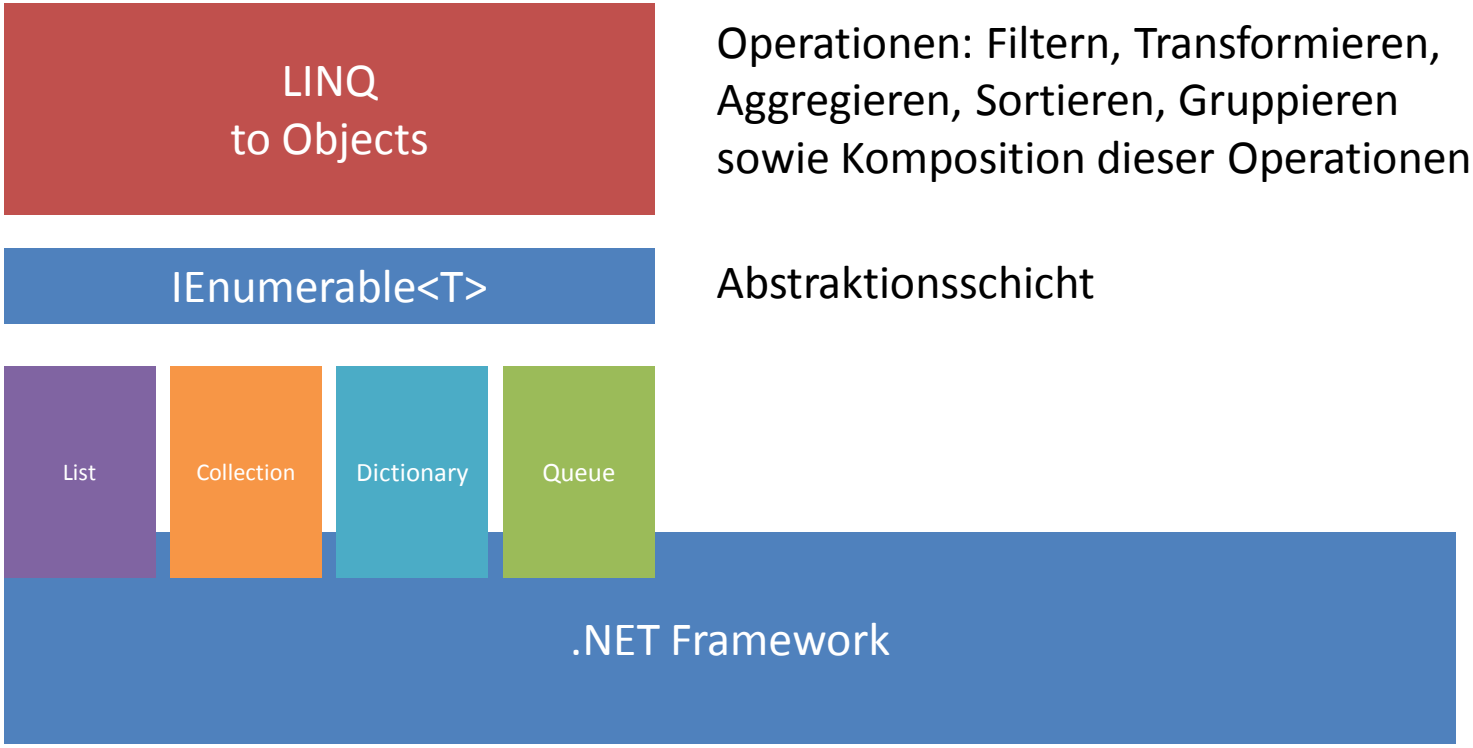
# Beispiel 0

- Scheinbar unspektakulär, aber
  - Viel Code für einen Adapter der "alten" Event-Infrastruktur zur Rx-Welt
    - Danach die Stärken von Rx verwenden!
  - Bei Brown Field-Entwicklung wird dies häufig notwendig sein
- Technisches
  - Bei Subscribe wird hinter den Kulissen ein Observer erzeugt

"*query* asynchronous data streams"

# VON "LINQ TO OBJECTS" ZU RX-"LINQ TO EVENTS"

# .NET 3.5: LINQ to Objects



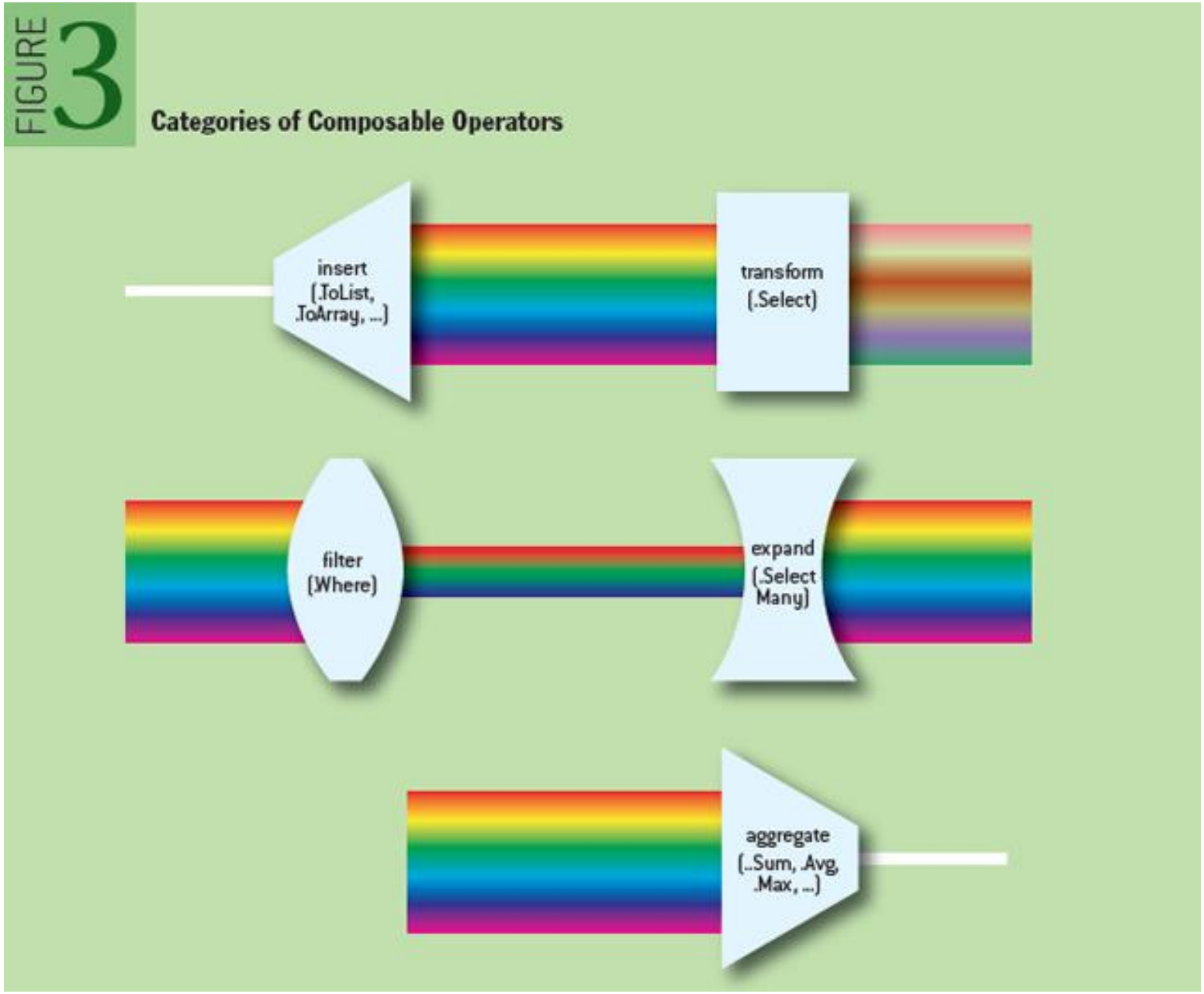

Operationen: Filtern, Transformieren, Aggregieren, Sortieren, Gruppieren sowie Komposition dieser Operationen

Abstraktionsschicht

Enumerator Pattern

# .NET 3.5: LINQ to Objects

- LINQ brachte ein
  - komfortables, einheitliches Programmiermodell für *Operationen* vom Typ Filtern, Transformieren, Aggregieren, Sortieren, Gruppieren sowie *Komposition* dieser Operationen
    (I steht für integrated!)
  - und revolutionierte die .NET Entwicklung im Bereich der Entwicklungsgeschwindigkeit und Verständlichkeit




**Aus: Why LINQ Matters: Cloud Composability Guaranteed, Bryan Beckmann 2012**

# .NET 3.5: LINQ - Just For the Record

- Der Vollständigkeit halber:
  - Andere wichtige Neuerungen von LINQ werden hier *nicht* ausgeführt, sind aber gleichwohl revolutionär
    - IQueryable für Expression Trees, die dann von LINQ-Providern "nativ" ausgewertet werden können.
    - LINQ verschiebt damit die "composability" in einzelne Domänen und aus dem Client heraus.

# Reactive Extensions

- Versuchen die Erfolgsgeschichte von LINQ to Objects auf asynchrone Ereignisströme zu übertragen!
  - LINQ to Events
    - Event im Sinne von asynchron eintreffenden Benachrichtigung, nicht als .NET-Sprachmittel!
  - Viele Operationen und deren Komposition
  - Programmiermodell vom Ansatz her bereits durch LINQ to Objects bekannt

Exkurs

# ENUMERATOR UND OBSERVER

# Enumerator: Blocking

- Enumerable-Enumerator:
  - Zwischen zwei **`yield return`** - Statements blockiert der Consumer
  - **`yield return`** hat Einschränkungen bei der Verwendung - by design!
    - Nur in Methoden, die ein IEnumerable zurückgeben
    - Nicht in anonymen Methoden
    - Nicht in try { } catch { }

# Observer: Non-Blocking

- Observable
  - Ereignisse treffen unabhängig vom Kontrollfluss ein
  - Kontrolfluss darf beim Warten nicht blockieren, muss immer in der Lage sein, andere Ereignisse zu verarbeiten und darauf zu reagieren
    - Stay responsive!
  - "non-blocking " und "asynchron" werden häufig synonym verwendet

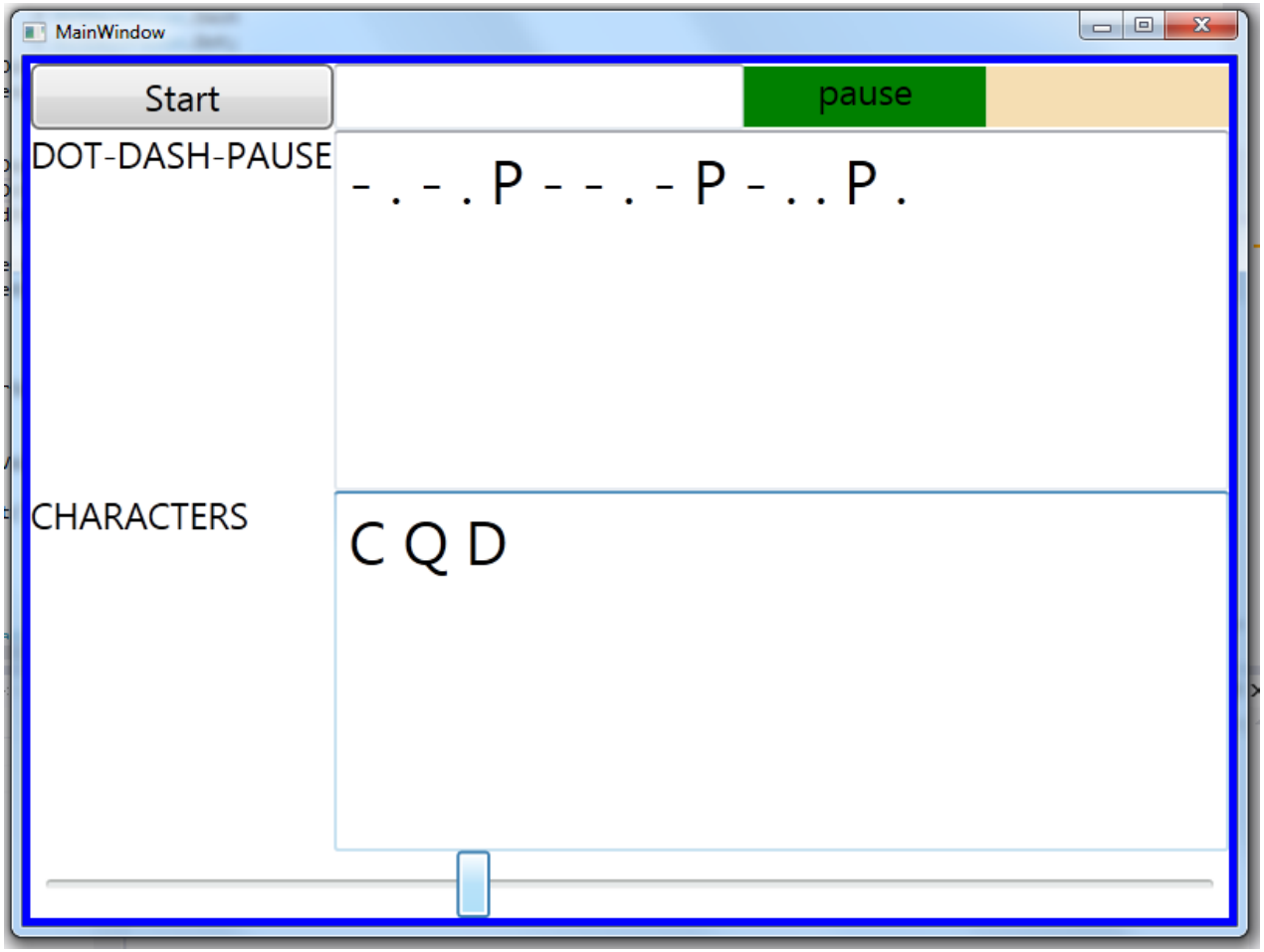
# BEISPIEL 1: MORSEN

# Beispiel 1

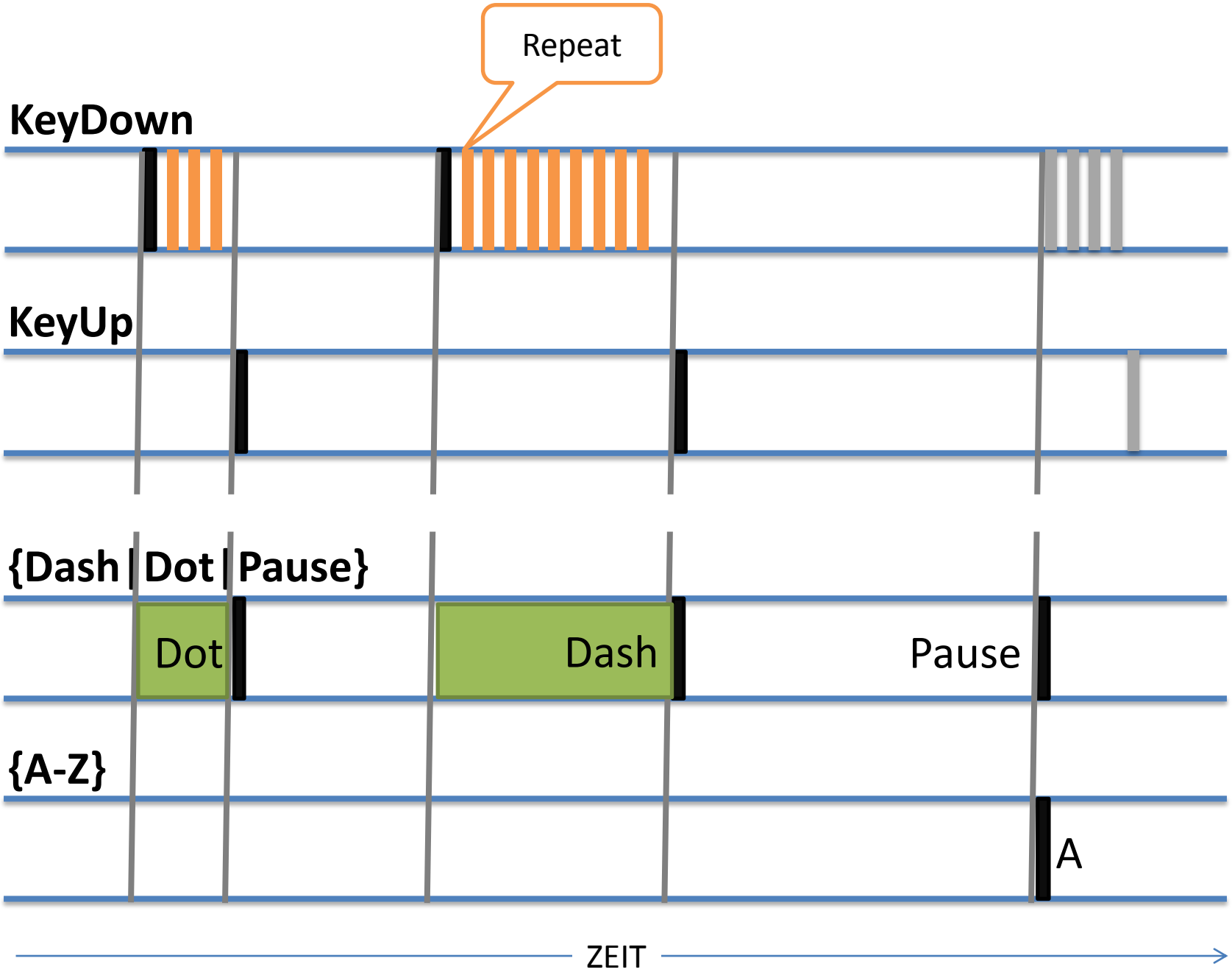
- Mit der Tastatur morsen (beliebige Taste)
  - Input: KeyDown und KeyUp Events (WPF)
  - Transformation zu {Dot, Dash, Pause}
  - Transformation in Buchstaben

- Entwicklung mit Rx-Bordmitteln
  - Standard-Methoden zur Komposition von IObservables: Filtern, Anreichern, Projezieren, Vereinigen

# Beispiel 1




run .-. -..-


Repeat
KeyDown
KeyUp
{Dash | Dot | Pause}
Dot
Dash
Pause
{A-Z}
A
ZEIT

# Beispiel 1

**Erzeuge** je ein IObservable für WPF-KeyDown, KeyUp-Events
**Ausfiltern** der Repeat-Events
**Transformation** der Events in eigenes KeyPressed (Up/Down)

```
// create an observable for key down (without repeat events)
IObservable<KeyPressed> keyDowns =
    Observable.FromEventPattern<KeyEventHandler, KeyEventArgs>(
        handler => theWindow.KeyDown += handler,
        handler => theWindow.KeyDown -= handler).
               Where(e => !e.EventArgs.IsRepeat).
               Select(e => KeyPressed.Down);

// create an observable for key up
IObservable<KeyPressed> keyUps =
    Observable.FromEventPattern<KeyEventHandler, KeyEventArgs>(
        handler => theWindow.KeyUp += handler,
        handler => theWindow.KeyUp -= handler).
            Select(e => KeyPressed.Up);
```

# Beispiel 1

**Zusammenführen** der Ereignisströme
**Anreichern** der Ereignisse um den zeitlichen Abstand zum vorhergehenden Ereignis

```
IObservable<KeyPressed> downAndUps = keyUps.Merge(keyDowns);

IObservable<TimeInterval<KeyPressed>> withInterval =
    downAndUps.TimeInterval();
```

Ergebnis: Zweitupel
(Down, 0),(Up, 100),(Down, 100),Up(2100),Down(3100)

# Beispiel 1

- "Fachlichkeit"
  - KeyUp: Dauer des Tastendrucks identifiziert Dash (>= 2 s) und Dot (< 2 s)
  - KeyDown: nur für Pausenerkennung (>= 3 s) wichtig. Deshalb können KeyDown-Events mit einem Intervall kürzer als die Pausenzeit ignoriert werden.

~~(Down, 0)~~,(Up, 100~~),(Down, 100)~~,Up(2100),Down(3100)

# Beispiel 1

**Herausfiltern** von KeyDown-Ereignissen, die keine Pause beenden. **Umwandeln** in {Dot, Dash, Pause}

```
IObservable<DotDashPause> dotDashPauses = withInterval.
    Where(e => e.Value.IsUp || e.Interval >= s_minPauseLength).
    Select(e => {
        if (e.Value.IsDown)
            return DotDashPause.Pause;
        return (e.Interval >= s_minDashLength)
            ? DotDashPause.Dash
            : DotDashPause.Dot;
    });
```

# Beispiel 1

**Segmentierung** des Ereignisstroms aus {Dot, Dash, Pause} an den Pausen

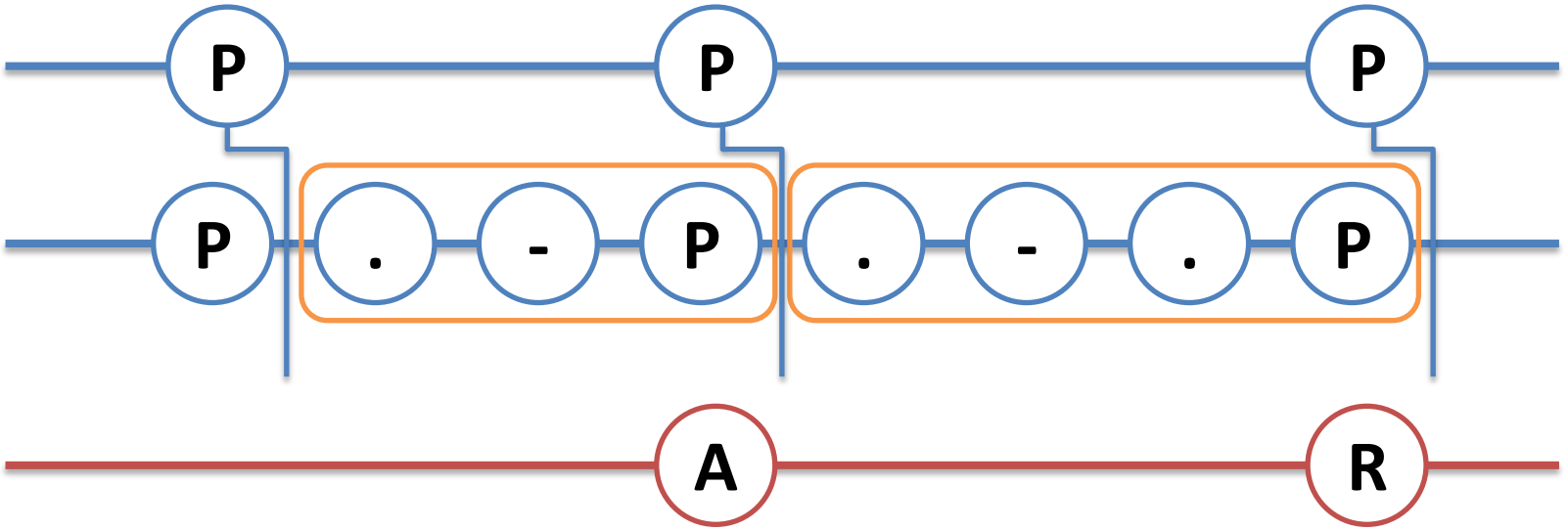
**Projektion** der Segmente in Zeichen

```
IObservable<string> characters = dotDashPauses.
    Buffer(
        // pauses are boundary markers
        m_dotDashPauses.Where(ddp => ddp == DotDashPause.Pause)
    ).
    Where(segment => segment.Count > 1).
    Select(GetCharForSequence);

private string GetCharForSequence(IList<DotDashPause> sequence)...
```

# Beispiel 1

- **Buffer** zerlegt den Ereignisstrom an Schnittpunkten: zweites Observable als Parameter; Schnittpunkt wenn dieses "feuert"



# Beispiel 1 - Rückblick

- Wrapper für Events als IObservable erzeugt
- Viel Operatoren verwendet
  - Where (Filtern), Select (Transformieren), Merge (Zusammenführen), TimeInterval (Anreichern), Buffer (Segmentieren)
  - Komposition von Operatoren
- Denselben Strom auf mehrere Arten weiterverarbeitet (`Buffer`)

# Beispiel 1

- Wenig Code durch aussagekräftige und mächtige Operatoren

```
// create an observable for key down (without repeat events)
IObservable<KeyPressed> keyDowns =
    Observable.FromEventPattern<KeyEventHandler, KeyEventArgs>(
        handler => theWindow.KeyDown += handler,
        handler => theWindow.KeyDown -= handler).
               Where(e => !e.EventArgs.IsRepeat).
               Select(e => KeyPressed.Down);

// create an observable for key up
IObservable<KeyPressed> keyUps =
    Observable.FromEventPattern<KeyEventHandler, KeyEventArgs>(
        handler => theWindow.KeyUp += handler,
        handler => theWindow.KeyUp -= handler).
            Select(e => KeyPressed.Up);

IObservable<DotDashPause> dotDashPauses =
    keyDowns.Merge(keyUps).
    TimeInterval().
    Where(e => e.Value.IsUp || e.Interval > s_minPauseLength).
    Select(e => {
            if (e.Value.IsDown)
                return DotDashPause.Pause;
            return (e.Interval > s_minDashLength)
                       ? DotDashPause.Dash
                       : DotDashPause.Dot;
    });

 IObservable<string> characters = dotDashPauses.Buffer(
        dotDashPauses.Where(ddp => ddp == DotDashPause.Pause)
     ).
     Where(sequence => sequence.Count > 1).
     Select(GetCharForSequence);
```

# Beispiel 1

- Technische Details
  - In `Buffer` verwenden wir den Strom über {Dot, Dash, Pause} zweimal:
    - Gefiltert (nur Pausen) zur Bestimmung der Schnittpunkte für die Segmentierung
    - Als zu segmentierenden Strom
  - In der Buffer-Implementierung wird erst das Pause-Zeichen im Buffer aufgenommen, dann die Segmentierung ausgewertet

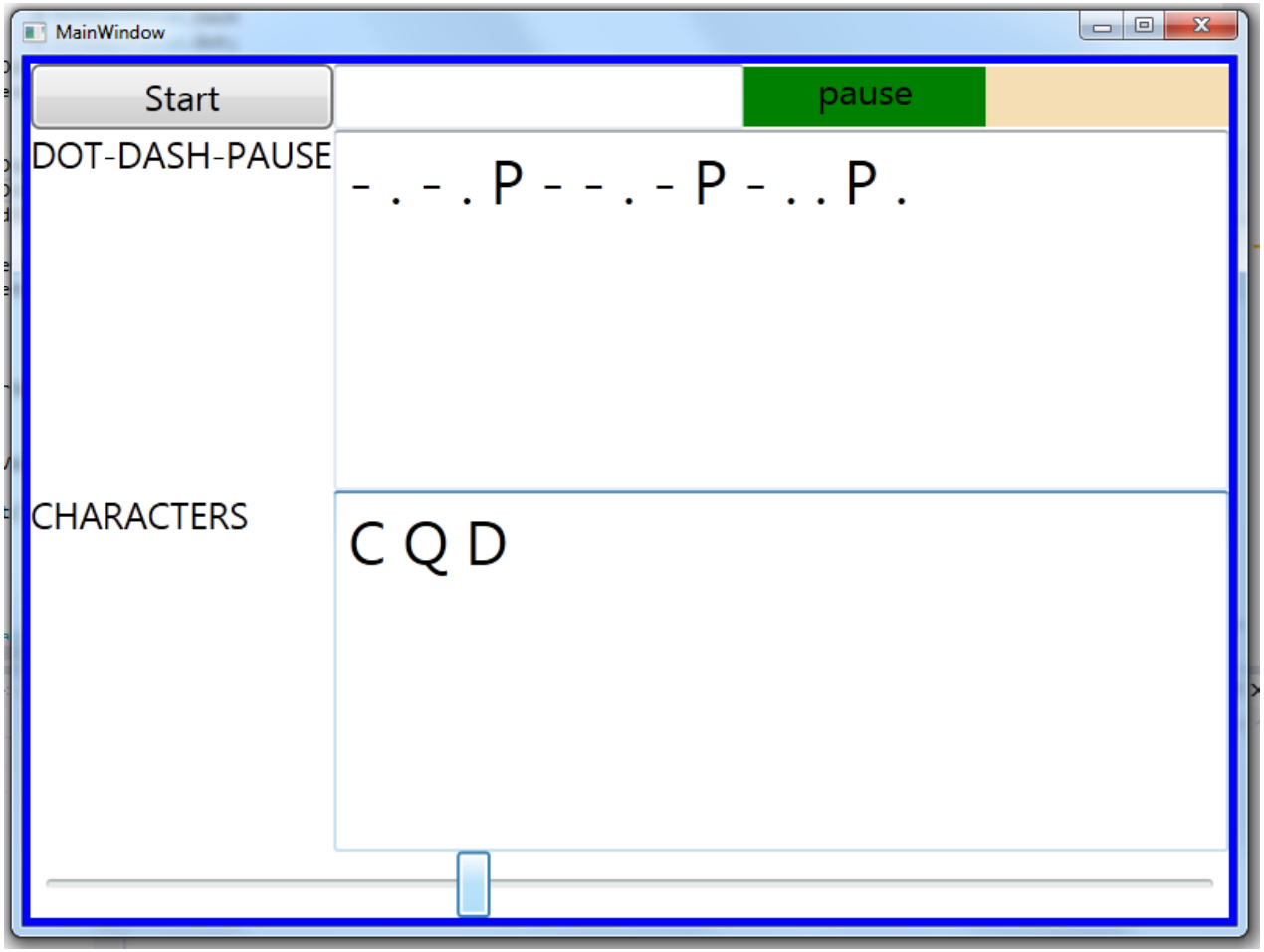
# Beispiel 1

- Zwei Hilfen für Morseanfänger
  - Wird eine Taste länger als 2 s gedrückt, gibt der DotDash-Indicator einen Hinweis
    - Strom downAndUps wird in einen Strom von {DoesNotApply, Dot, Dash} transformiert
  - Bleibt länger als 3 s keine Taste gedrückt, gibt der Pausenindikator einen Hinweis
    - Strom downAndUps wird in einen Strom von {DoesNotApply, NoPause, Pause} transformiert
  - Die beiden neuen Ströme steuern die Indikatoren im User Interface

(echte)

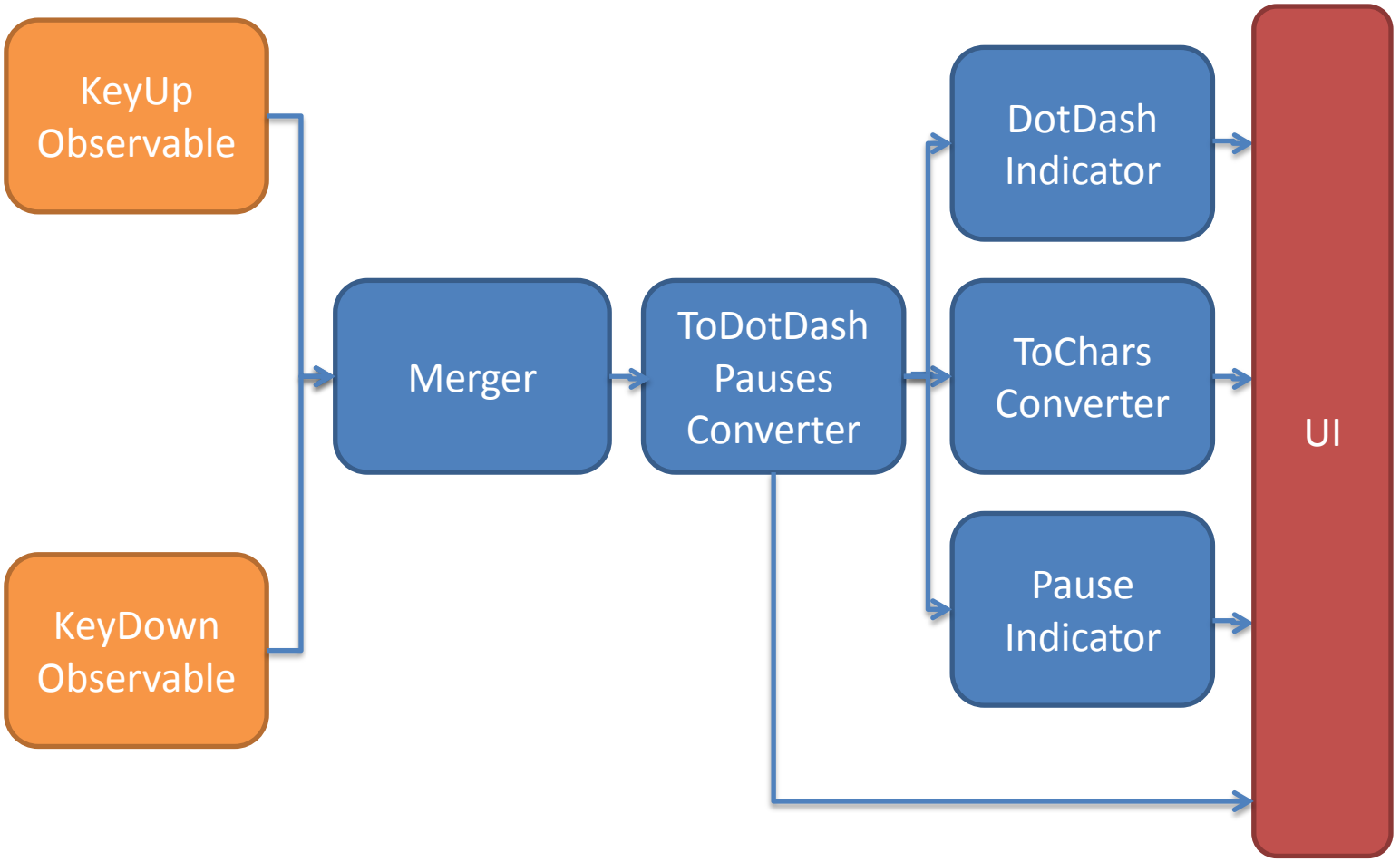
# PAUSE

# Beispiel 1




run .-. -..-

# Beispiel 1


KeyUp Observable
KeyDown Observable
Merger
ToDotDash Pauses Converter
DotDash Indicator
ToChars Converter
Pause Indicator
UI

# Beispiel 1

- Funktionalität in Komponenten schneiden
- Komponenten verdrahten
- Wäre dies noch imperative Programmierung?

# Beispiel 1

- Paradigmen
  - Event-Driven Programming
    "Program flow is determined mainly by events, such as mouse clicks or interrupts including timer"
  - Dataflow
    "Es wird von einem kontinuierlichen Datenfluss ausgegangen (meist Audio- oder Videodaten), der (oft in Echtzeit) verändert und ausgegeben wird."

LINQ to Events

# OPERATIONEN

# Rx-Operatoren

Aggregate
All
Amb
And
Any
AsObservable
Average
**Buffer**
Case
Cast
Catch
Chunkify
Collect
CombineLatest
Concat
Contains
Count
Create
DefaultIfEmpty
Defer
DeferAsync
Delay
DelaySubscription
Dematerialize
Distinct
DistinctUntilChanged
Do
DoWhile
ElementAt
ElementAtOrDefault
Empty
Equals
Finally
First
FirstAsync
FirstOrDefault
FirstOrDefaultAsync
For
ForEach
ForEachAsync
FromAsync
FromAsyncPattern
FromEvent
**FromEventPattern**
Generate
GetAwaiter
GetEnumerator
GetHashCode
GetType
GroupBy
GroupByUntil
GroupJoin
If
IgnoreElements
Interval
IsEmpty
Join
Last
LastAsync
LastOrDefault
LastOrDefaultAsync
Latest
LongCount
Materialize
Max
MaxBy
**Merge**
Min

# Rx-Operatoren

MinBy
MostRecent
Multicast
Never
Next
ObserveOn
OfType
OnErrorResumeNext
Publish
PublishLast
Range
RefCount
Repeat
Replay
Retry
Return
RunAsync
Sample
Scan
**Select**
SelectMany
SequenceEqual
Single
SingleAsync
SingleOrDefault
SingleOrDefaultAsync
Skip
SkipLast
SkipUntil
SkipWhile
Start
StartAsync
StartWith
**Subscribe**
SubscribeOn
Sum
Switch
Synchronize
Take
TakeLast
TakeLastBuffer
TakeUntil
TakeWhile
Then
Throttle
Throw
**TimeInterval**
Timeout
Timer
Timestamp
ToArray
ToAsync
ToDictionary
ToEnumerable
ToEvent
ToEventPattern
ToList
ToLookup
ToObservable
ToString
Using
Wait
When
**Where**
While
Window
Zip

# Rx-Operatoren

- Unmöglich, hier alle Operationen vorzustellen
- Einteilung in Funktionsgruppen
  - Create
  - Reduce
  - Inspect
  - Aggregate
  - Segmentate
  - Transform

(aus: Introduction to Rx, Lee Campbell)

# Rx-Operatoren

- Andere Klassifikationen
  - Zeitsteuerungs-Methoden (Throttle, Delay)
  - Nach Art des Rückgabewerts
    - IObservable<T>, IEnumerable<T>, T
  - Blocking oder non-blocking?
    - Methoden, die etwa ein T zurückgeben, können blockieren (etwa First(), Async-Varianten)
    - Methoden, die ein IObservable zurückgeben, blockieren i.d.R. erst einmal nicht (Beispiel: Take())

" represent asynchronous data streams"

# SUBJECTS

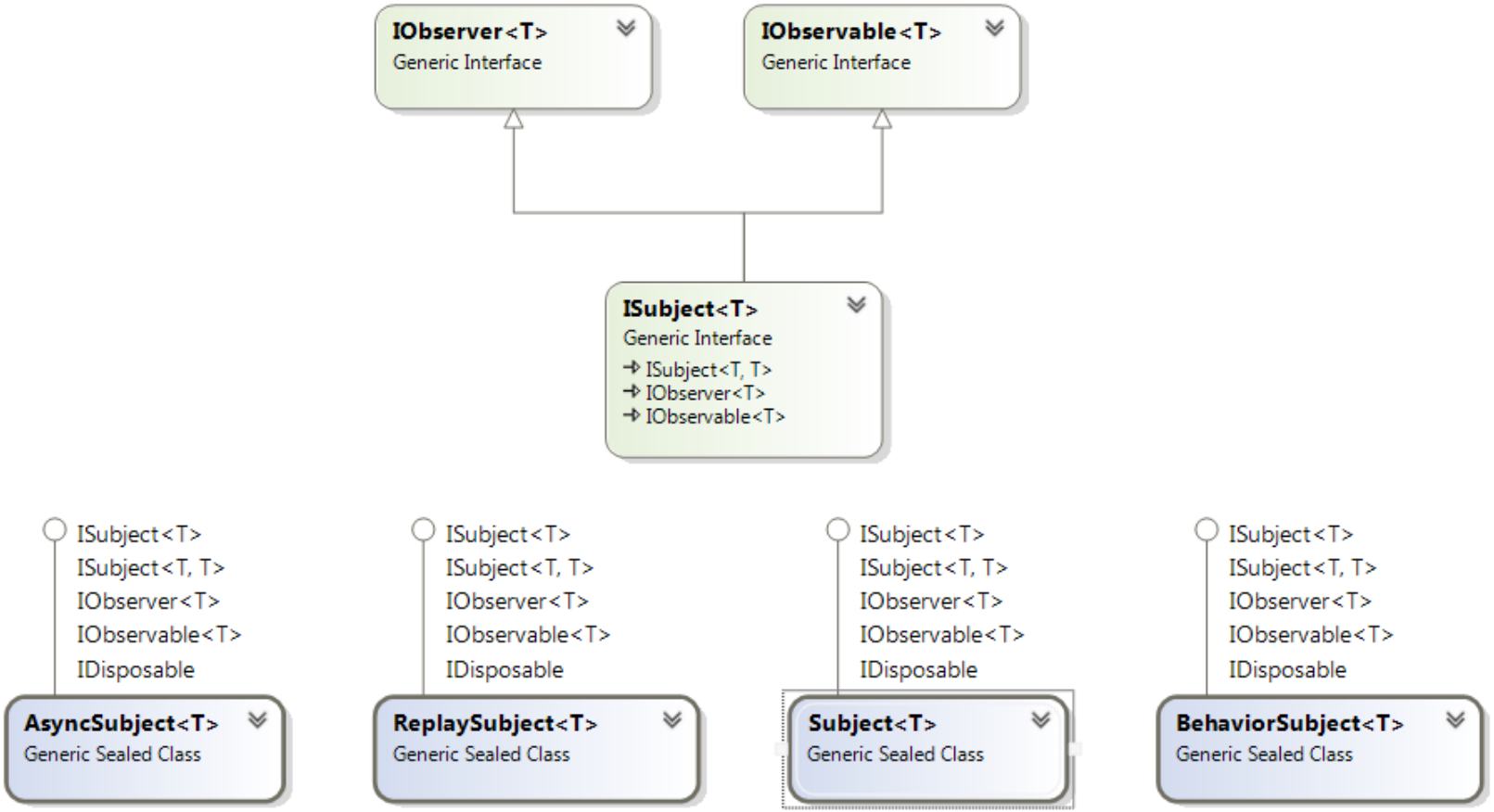
# Rx Subjects

- Subject<T> implementiert IObservable<T> und auch IObserver<T>
- Wann brauche ich ein Subject?
  - Proxy-Funktion: Subject vorab erzeugen, Subject mit Observer (ggf. mehrere) verbinden, irgendwann Subject an ein IObservable ankoppeln
    - Wenn man Beispiel 1 wie skizziert zerlegt, werden Subjects genau so verwendet!

# Rx Subjects

- Wann brauche ich ein Subject?
  - Subject-Implementierung mit State: Ereignisstrom-Cache (s.u.)
  - Eigene Quelle für asynchronen Ereignisstrom, der manuelle gesteuert wird (Brownfield-Entwicklung)

# Rx Subjects


IObserver<T>
Generic Interface
IObservable<T>
Generic Interface
ISubject<T>
Generic Interface
ISubject<T, T>
IObserver<T>
IObservable<T>
ISubject<T>
ISubject<T, T>
IObserver<T>
IObservable<T>
IDisposable
AsyncSubject<T>
Generic Sealed Class
ISubject<T>
ISubject<T, T>
IObserver<T>
IObservable<T>
IDisposable
ReplaySubject<T>
Generic Sealed Class
ISubject<T>
ISubject<T, T>
IObserver<T>
IObservable<T>
IDisposable
Subject<T>
Generic Sealed Class
ISubject<T>
ISubject<T, T>
IObserver<T>
IObservable<T>
IDisposable
BehaviorSubject<T>
Generic Sealed Class

# Rx Subjects

- Subject<T>
  - Bei Subscribe erhält der Observer ab dem aktuellen Zeitpunkt alle Notifications (Next, Error, Complete)
- ReplaySubject<T>
  - Bei Subscribe erhält der neue Observer zusätzlich alle vergangenen Notifications
  - Subject mit Cache - gegen Race Conditions!

# Rx Subjects

- BehaviorSubject<T>
  - Dieses Subject liefert bei Subscribe immer und sofort einen Wert
    - Den zuletzt mit OnNext empfangene Wert (i.d.S. wie ReplaySubject) oder einen im Konstruktor spezifizierten Default-Wert
    - Blockiert also nie!
  - Kein Wert wenn bei Subscribe bereits Error oder Completed vorliegt

# Rx Subjects

- BehaviorSubject<T>
  - Analogie .NET-Property
    Siehe Ansätze wie ReactiveUi, die in MVVM versuchen, soweit wie möglich mit IObservable<T> in Richtung View zu kommen (ReactiveObject, ReactiveCollection, ReactiveCommand, http://www.reactiveui.net/)

# Rx Subjects

- AsyncSubject<T>
  - Spezial-Subject für Operationen, die ein oder kein Ereignis liefern (wie etwa ein "normaler" Web Service-Zugriff)
  - Gibt den letzten gelieferten Wert weiter, aber erst, wenn Completed empfangen wurde.

"parameterize the concurrency"

# SCHEDULING

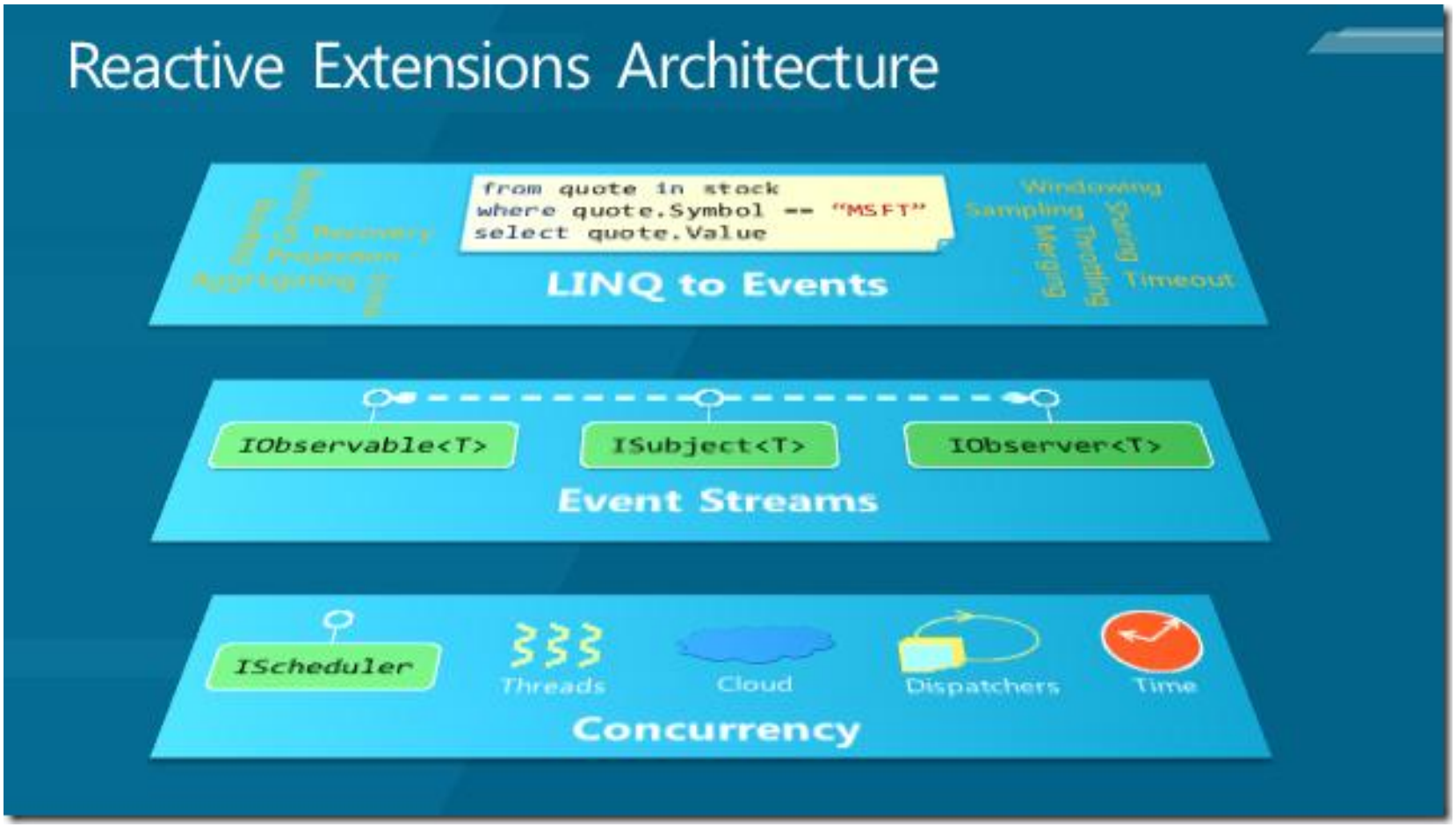
# Scheduling

"The Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators. Using Rx, developers ***represent*** asynchronous data streams with Observables, ***query*** asynchronous data streams using LINQ operators, **and *parameterize* the concurrency in the asynchronous data streams using Schedulers**. Simply put, Rx = Observables + LINQ + Schedulers."

# Scheduling

- Asynchronität bedeutet Concurrency
  - In Rx sind Scheduler die Klassen, die Concurrency parametrisieren
    - ExecutionContext (where to do work)
    - ExecutionPolicy (in which order)
    - Clock (when to do work)
  - Rx bietet weitreichende Mittel
    - Aber auch für Rx gilt der Default: zunächst alles auf einem Thread!

# Scheduling




http://blogs.msdn.com/b/rxteam/archive/2012/06/14/testing-rx-queries-using-virtual-time-scheduling.aspx

# Scheduling

```
var observable = Observable.Create<int>(
    s => {
        s.OnNext(3);
        s.OnCompleted();
        return Disposable.Create(() => { });
    });

var subscribeDisposable = observable.
    Subscribe(
        LogNewData, LogError, LogCompleted
    );
```

Subscribe build-up

Observe

# Scheduling

- Der Notification-Code (Next, Error, Complete) muss z.B. auf dem Dispatcher eines Control aufgerufen werden, mit dem er arbeitet.
  - `ObserveOn`
- Seltener: Der Buildup-Code soll möglicherweise auf einem anderen Thread ausgeführt werden, da er sonst den folgenden Subscribe-Aufruf blockieren würde.
  - `SubscribeOn`

# Scheduling

```
var observable = Observable.Create<int>(
    s => {
        s.OnNext(3);
        s.OnCompleted();
        return Disposable.Create(() => { });
    });

var subscribeDisposable = observable.
    SubscribeOn(new NewThreadScheduler()).
    ObserveOn(targetControl).
    Subscribe(
        LogNewData, LogError, LogCompleted
    );
```

# Scheduling

- Beispiele für Scheduler
  - DefaultScheduler
  - DispatcherScheduler
  - ImmediateScheduler
  - TaskPoolScheduler
  - ThreadPoolScheduler
  - ...
  - TestScheduler (mehr später)

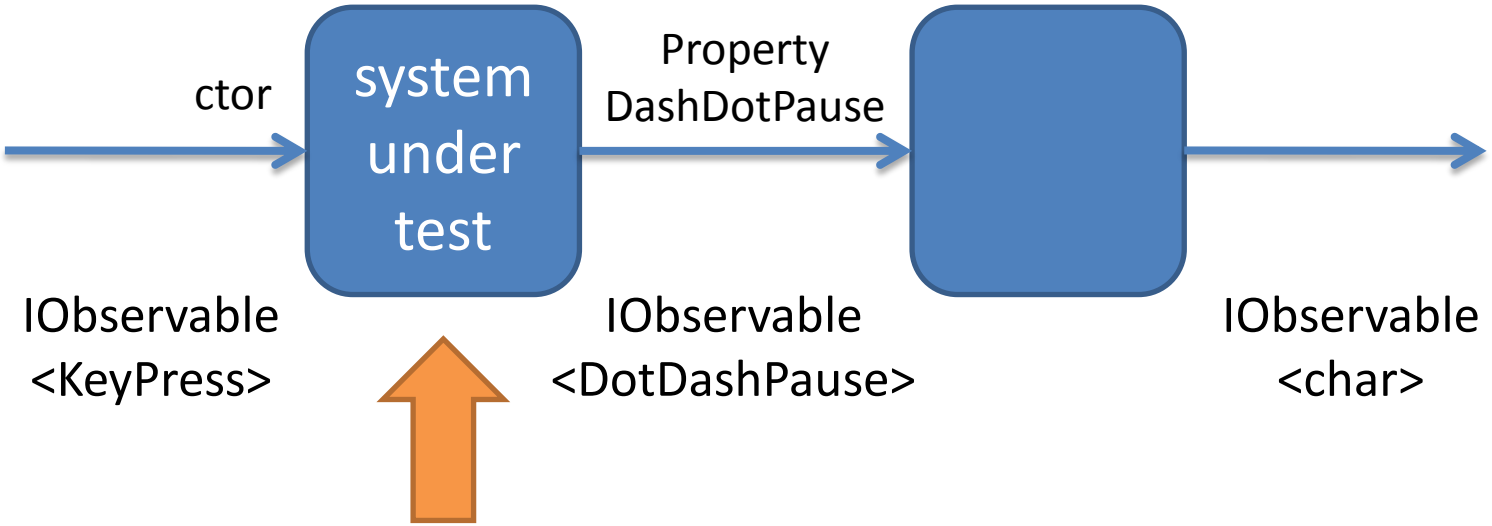
# Scheduling

- Auch wenn vieles steuerbar ist,
  haben die Reactive Extensions eine gezielte Parallelisierung nicht im Fokus
- Dafür gibt es andere Bibliotheken

Test Last

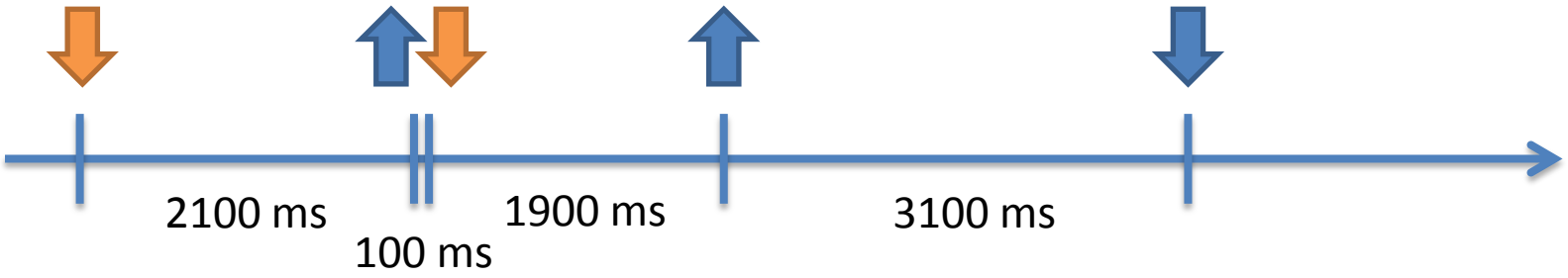
# TESTING

# Testing


ctor
system under test
Property DashDotPause
IObservable <KeyPress>
IObservable <DotDashPause>
IObservable <char>

# Testing

- Einfacher Testfall:




| DASH | DOT | PAUSE |
| --- | --- | --- |
| >2000 ms | <2000 ms | >3000 ms |

# Testing

```
var downsAndUps = Observable.Create<KeyPressed>(o =>
    {
        o.OnNext(KeyPressed.Down);  // DASH > 2000 ms
        Thread.Sleep(2100);
        o.OnNext(KeyPressed.Up);
        Thread.Sleep(100);
        o.OnNext(KeyPressed.Down);  // DOT < 2000 ms
        Thread.Sleep(1900);
        o.OnNext(KeyPressed.Up);
        Thread.Sleep(3100);         // PAUSE > 3000 ms
        o.OnNext(KeyPressed.Down);
        return Disposable.Empty;
    });

var systemUnderTest = new Morse(downsAndUps);

DotDashPause[] result = systemUnderTest.DotDashPauses.
    Take(2).ToEnumerable().ToArray();
Assert.AreEqual(DotDashPause.Dash, result.First());
Assert.AreEqual(DotDashPause.Dot, result.Last());
```

# Testing

- Testdauer: > 7200 ms
- Wie geht es besser?
  Schneller testen mit dem Rx TestScheduler und virtueller Zeit!
  - TestScheduler hat sein eigenes Zeitgefüge
  - Operationen können in diesem Zeitgefüge platziert werden
  - Dadurch erhebliche Testbeschleunigung und Robustheit

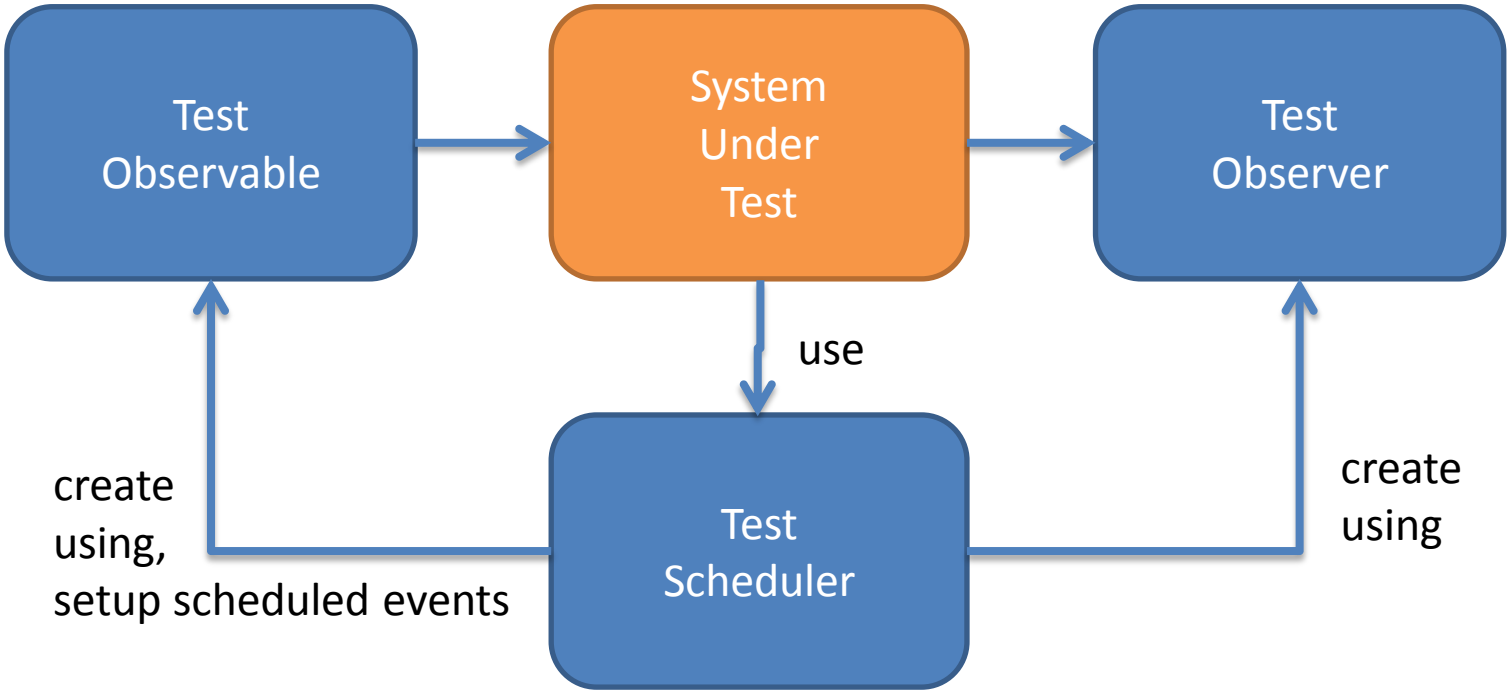
# Testing

- TestScheduler verwenden und damit (Test)Observable für Input aufsetzen

```
var scheduler = new TestScheduler();

// create input for system under test (scheduled on virtual time)
IObservable<KeyPressed> observable = scheduler.CreateHotObservable(
    new Recorded<Notification<KeyPressed>>(TimeSpan.FromMilliseconds(0).Ticks,
        Notification.CreateOnNext(KeyPressed.Down)),
    new Recorded<Notification<KeyPressed>>(TimeSpan.FromMilliseconds(2100).Ticks,
        Notification.CreateOnNext(KeyPressed.Up)),
    new Recorded<Notification<KeyPressed>>(TimeSpan.FromMilliseconds(2200).Ticks,
        Notification.CreateOnNext(KeyPressed.Down)),
    new Recorded<Notification<KeyPressed>>(TimeSpan.FromMilliseconds(4100).Ticks,
        Notification.CreateOnNext(KeyPressed.Up)),
    new Recorded<Notification<KeyPressed>>(TimeSpan.FromMilliseconds(7200).Ticks,
        Notification.CreateOnNext(KeyPressed.Down)));
```

# Testing


Test
Observable
System
Under
Test
Test
Observer
use
create
using,
setup scheduled events
Test
Scheduler
create
using

# Testing

```
// create output observer for system under test
ITestableObserver<DotDashPause> observer = scheduler.CreateObserver<DotDashPause>();

// our system under test takes an IObservable<KeyPressed> and
// implements IObservable<DotDashPause>
IObservable<DotDashPause> systemUnderTest =
    new KeyPressedToDotDashPause(observable, scheduler);
systemUnderTest.Subscribe(observer);

// act (play schedule of observable)
scheduler.Start();

// assert
CollectionAssert.AreEqual(
    new[] {DotDashPause.Dash, DotDashPause.Dot, DotDashPause.Pause },
    observer.Messages.Take(3).Select(m => m.Value.Value).ToArray());
```

# Testing

- system under test aufsetzen
- Observer erzeugen, der zu validierende Ergebnisse am system under test abgreift
- (Test)Obervable veranlassen, die Ereignisse im virtuellen Zeitgefüge abzuspielen
- Ergebnisse, die der Observer empfangen hat, validieren

# Testing

- Unschön: innerhalb des System Under Test gibt es Operationen, die ebenfalls den TestScheduler verwenden müssen
  - Etwa TimeInterval, Timeout
  - Design for Testability: Scheduler injizieren
    Ggf. später per Factory

- Gewinn: Testdauer **190 ms** vs **7200 ms**

Fast geschafft ...

# WRAP-UP

# Verwendung

- Wann soll Rx eingesetzt werden (MoSCoW)?
  - Must: <left blank intentionally>
  - Should: Events, die spontan von Sensoren oder aus anderen Domänen kommen, fachliche Events wie "LogonFailed", Infrastruktur-Events u.ä. - Vieles!
  - Could: Ergebnisse asynchroner Aufrufe entgegennehmen und verbreiten
  - Won't: kein Ersatz für IEnumerable
- Besonders dort, wo Eleganz und Ausdrucksstärke von LINQ to Events zur vollen Entfaltung gelangt!

# Fazit

- Rx ist mächtig, vielfältig und elegant
  - Will man die gesamte Funktionalität ausloten, so benötigt man Zeit. Zu Beginn ist die Lernkurve sehr angenehm, mit zunehmender Tiefe und Breite (großen Bibliothek) wird es schwieriger.
- Rx lebt und ist gut dokumentiert
  - Ausnahme: Testing
- Rx macht Lust ...
  ... auf Event-Based Programming!

# FRAGEN?


Danke!
[Fertig]

# Links

- http://www.introtorx.com/
  Hervorragendes Online-Buch von Lee Campell
- http://channel9.msdn.com/
  Etliche Beiträge, die Rx bis in die Tiefe ausloten
- "Programming Reactive Extensions and LINQ"
  von Jesse Liberty und Paul Betts ist weniger zu empfehlen

# Bilder

- Fotos
  - "Wolke-Wolke Blitz": Karl-Heinz Laube / pixelio.de
  - "Guckste Du Sterne": Gerd Altmann / pixelio.de
  - http://www.flickr.com/photos/markmorgantrinidad/5298000054/sizes/l/in/photostream/ - (Tanzschritte)
  - "Müder Löwe": Sw / pixelio.de

- Übersichten
  - http://en.wikipedia.org/wiki/File:Observer.svg (by WikiSolved)
  - LINQ-Operationen: http://queue.acm.org/detail.cfm?id=2141937
  - Rx-Architektur: http://blogs.msdn.com/b/rxteam/archive/2012/06/14/testing-rx-queries-using-virtual-time-scheduling.aspx