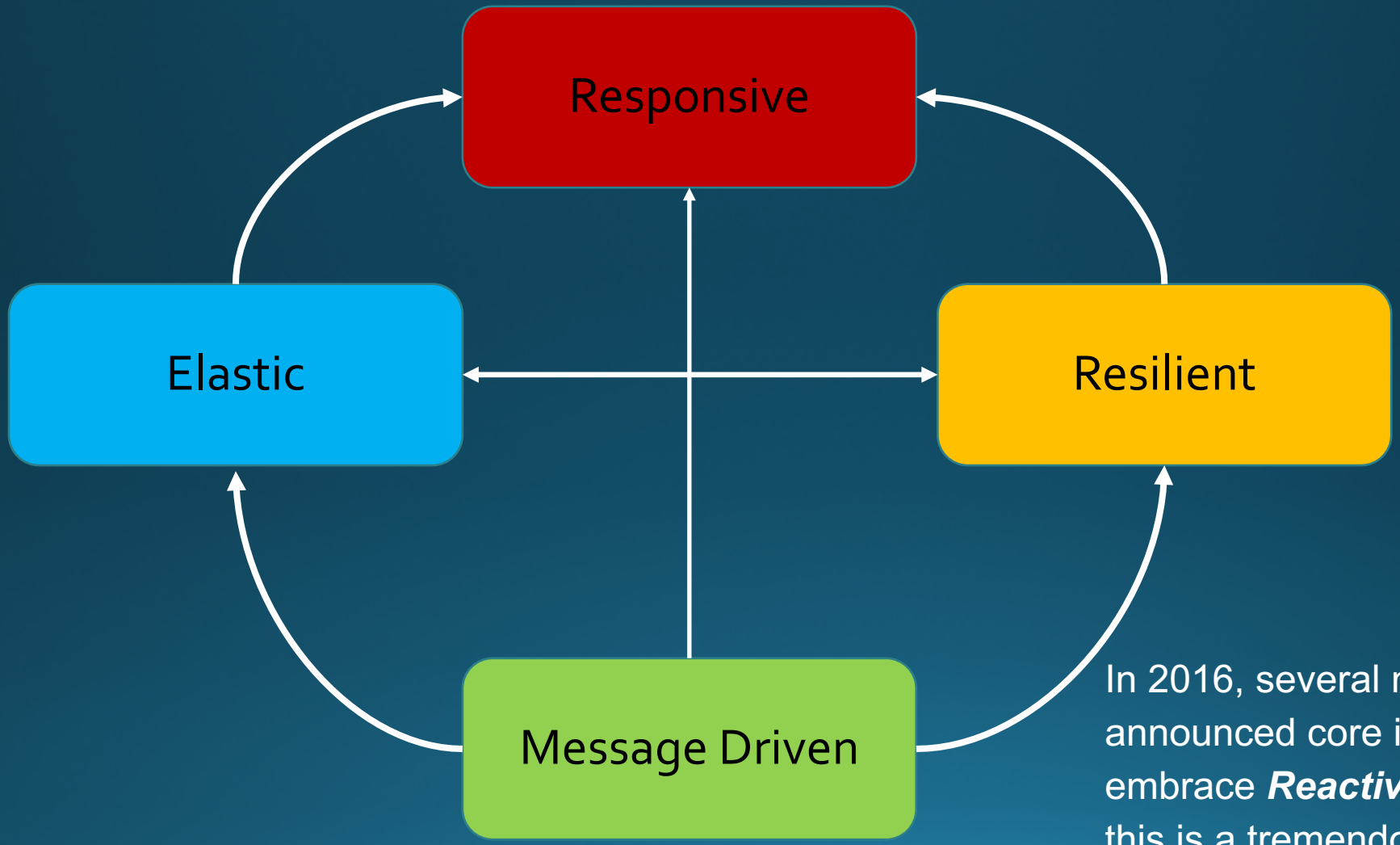
Reactive Programming

Objectives

- Stream processing motivation
- Real time stream analysis
- Simplify complex logic with reactive programming
- Composing events

Reactive Manifesto



In 2016, several major vendors have announced core initiatives to embrace ***Reactive Programming*** this is a tremendous validation of the problems faced by companies today.

http://www.reactivemanifesto.org

Functional Reactive Programming

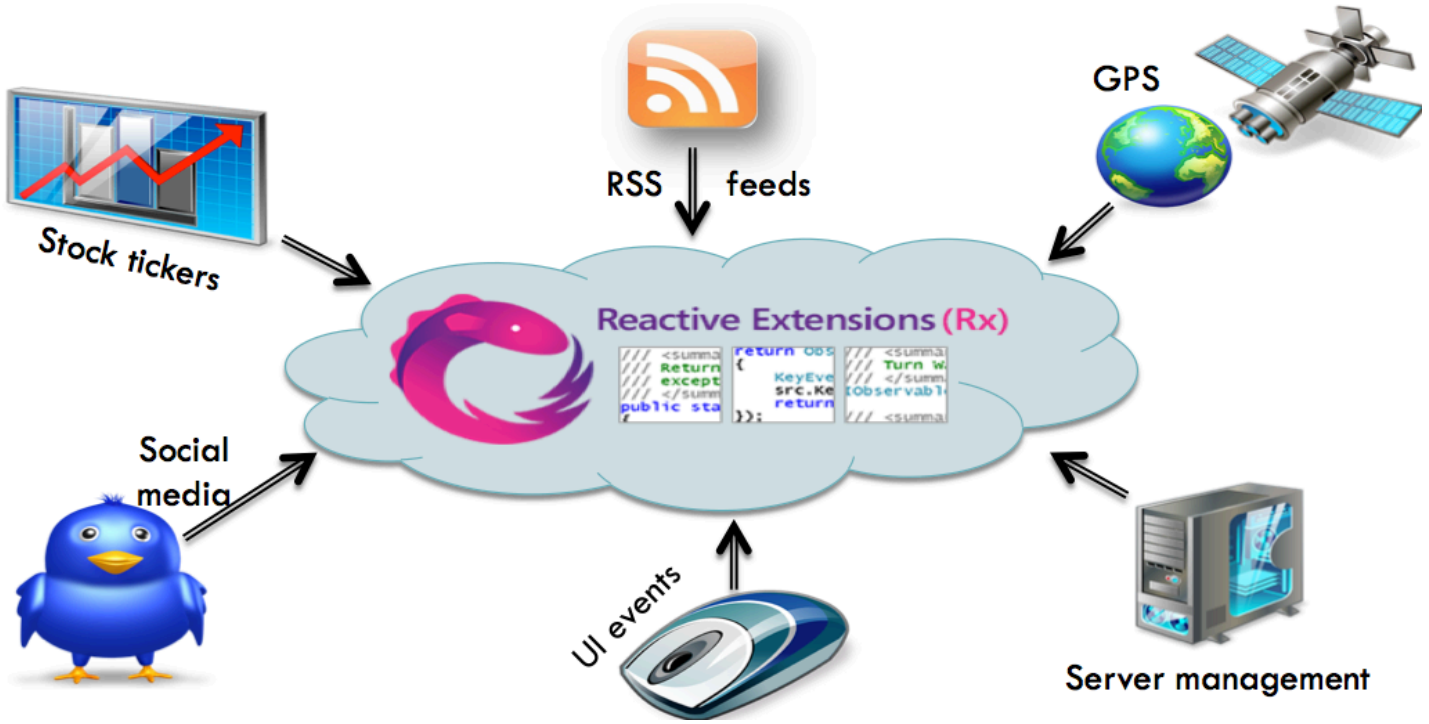
Functional ~~Reactive~~ Programming

- declarative
- functions as values
- side-effects free
- referential transparency
- immutable
- composition

~~Functional~~ Reactive Programming

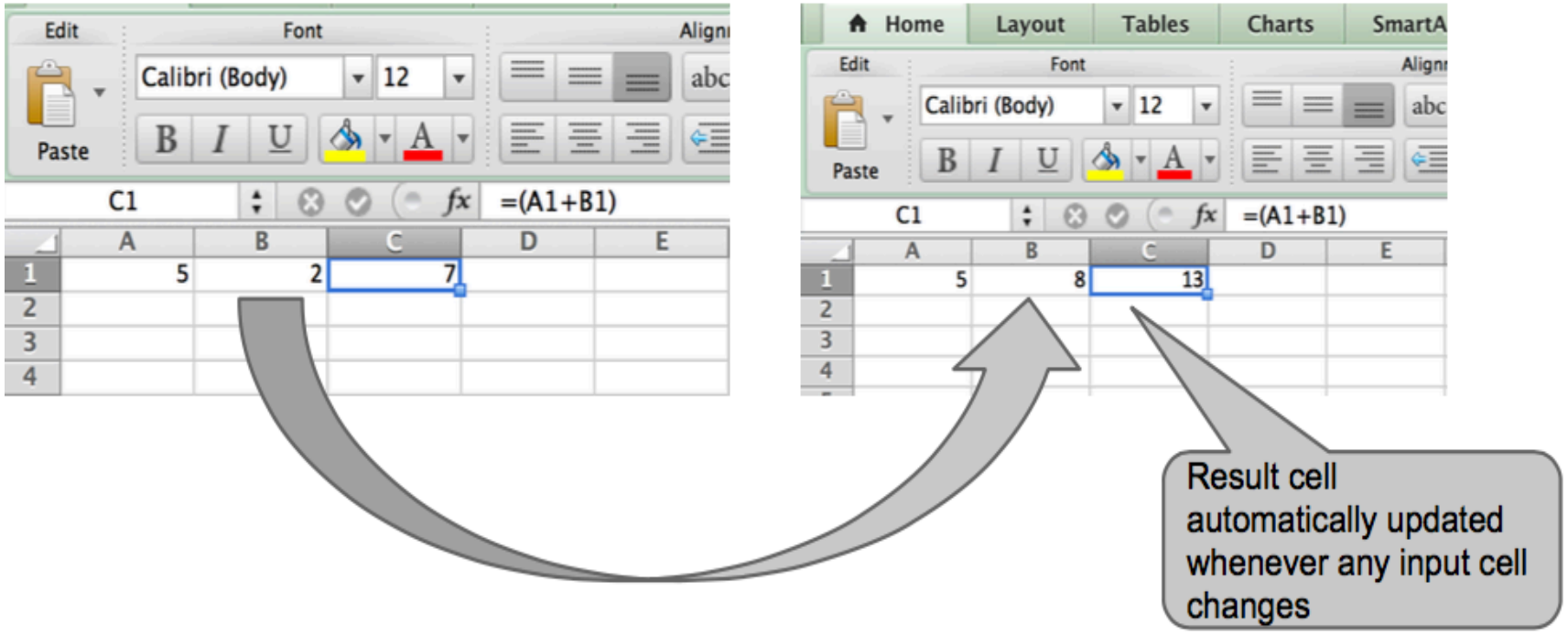
What is Reactive Programming

A programming paradigm oriented around data flows
and the propagation of change.



What is Reactive Programming

SpreadSheet == Mother of All Reactive Programming



Functional Reactive Programming

Push-Pull Functional Reactive Programming

Push-Pull Functional Reactive Programming

Conal Elliott

LambdaPix

conal@conal.net

Abstract

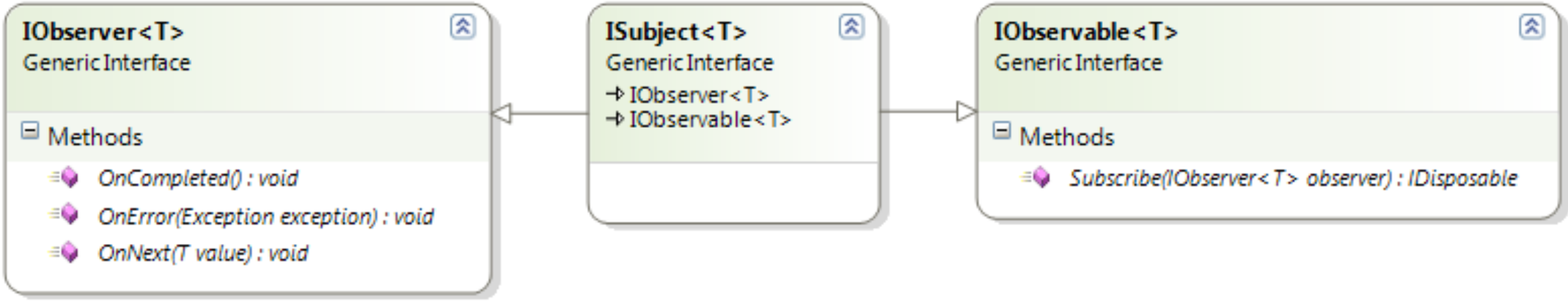
Functional reactive programming (FRP) has simple and powerful semantics, but has resisted efficient implementation. In particular, most past implementations have used demand-driven sampling, which accommodates FRP's continuous time semantics and fits well with the nature of functional programming. Consequently, values are wastefully recomputed even when inputs don't change, more composable than their finite counterparts, because they can be scaled arbitrarily in time or space, before being clipped to a finite time/space window.

While FRP has simple, pure, and composable semantics, its efficient implementation has not been so simple. In particular, past implementations have used demand-driven (pull) sampling of reactive behaviors, in contrast to the data-driven (push) evaluation typ-

Reactive Extensions

What are Reactive Extensions

Rx is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators



IObserver & IObservable

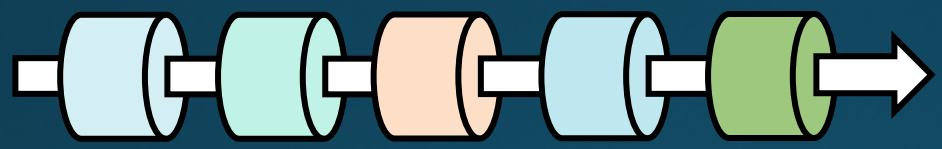
```
type IObserver<'a> = interface
    abstract OnCompleted : unit -> unit
    abstract OnError : exn -> unit
    abstract OnNext : 'a -> unit
end

type IObservable<'a> = interface
    abstract Subsrcribe : IObserver<'a> -> IDisposable
end
```

What is Rx?

Rx **compose pipeline** of operations over **single** or **multiple event's source**

Focus on what happens **between** the **Producer** and the **Consumer**

Rx is the **Pipeline**



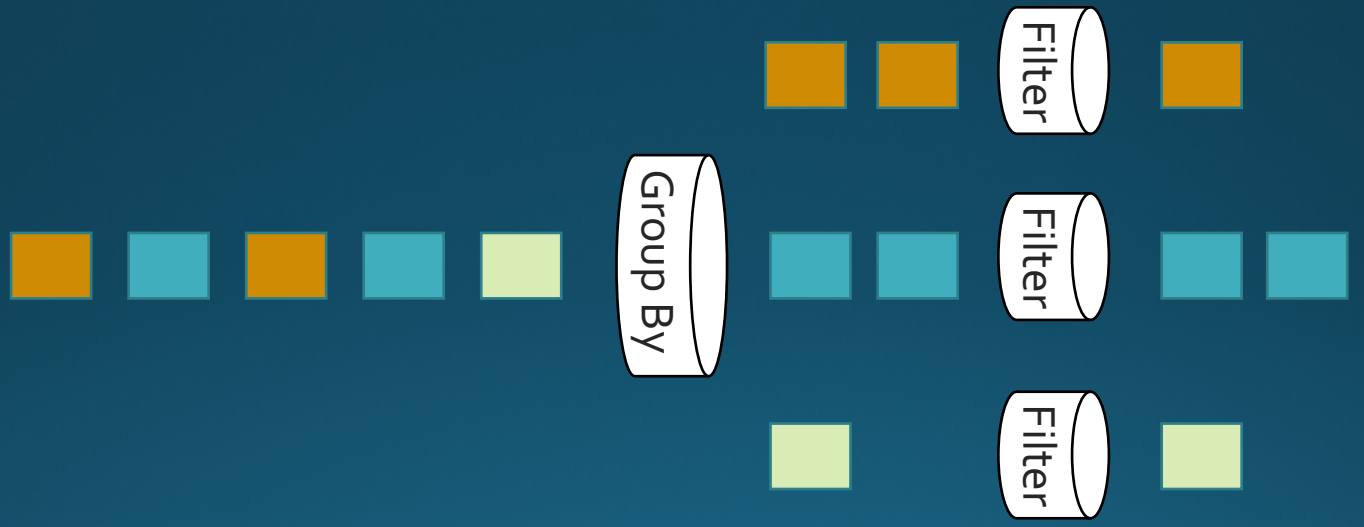
Rx = Async Data Stream Composition

Rx = LINQ to Async Data Stream

What is Rx?

Rx **compose pipeline** of operations over **single** or **multiple event's source**

Focus on what happens **between** the **Producer** and the **Consumer**



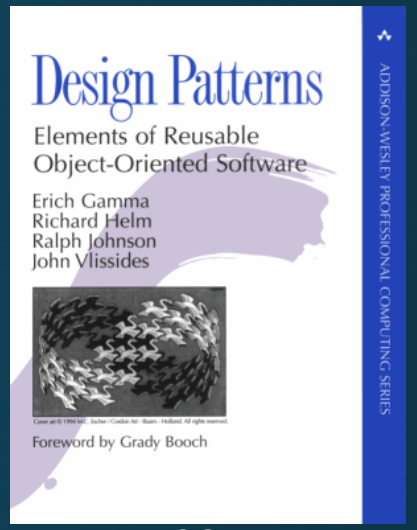
Rx = Async Data Stream Composition

Rx = LINQ to Async Data Stream

Fundamental Abstractions

- Adapting the observer pattern
  - Ensuring duality with the enumerator pattern
  - More compositional approach

```
interface IObservable<out T>
{
    IDisposable Subscribe(IObserver<T> observer);
}

interface IObserver<in T>
{
    void OnNext(T value);
    void OnError(Exception error);
    void OnCompleted();
}
```

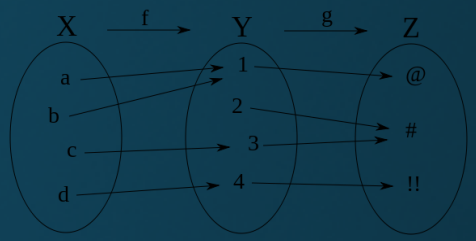


"Gang of four" book
Addison-Wesley

Notification grammar

```
OnNext* (OnError | OnCompleted)?
```

Highly Compositional



Function composition

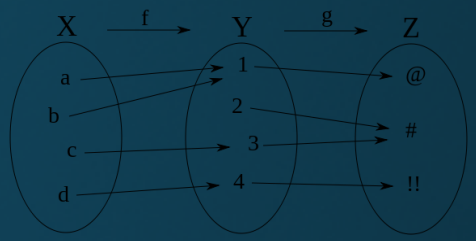
- LINQ-style query operators over IObservable<T>
  - Composition of 0-N input sequences

```
static class Observable
{
    static IObservable<T> Where<T>(this IObservable<T> source, Func<T, bool> f);
    static IObservable<R> Select<T, R>(this IObservable<T> source, Func<T, R> p);
    …
}
```

  - Composition of disposable subscriptions and scheduler resources

```
interface IScheduler
{
    IDisposable Schedule(Action work);
    …
}
```

Highly Compositional



Function composition

- Building a binary Merge operator

First-class: can build extension methods

```
public static class Observable
{
    public static IObservable<T> Merge<T>(this IObservable<T> xs, IObservable<T> ys)
    {
        return Create<T>(observer =>
        {
            var gate = new object();

            return new CompositeDisposable
            {
                xs.Subscribe(x => { lock (gate) { observer.OnNext(x); } }, …),
                ys.Subscribe(y => { lock (gate) { observer.OnNext(y); } }, …),
            };
        });
    }
}
```

Composition of resource management

The Role of Schedulers

- Pure architectural layering of the system
  - Logical query operators (~ relational engine)
  - Physical schedulers (~ operating system)

```
public static IObservable<T> Return<T>(T value, IScheduler scheduler)
{
    return Create<T>(obs => scheduler.Schedule(() => { obs.OnNext(value);
                                                       obs.OnCompleted(); }));
}
```

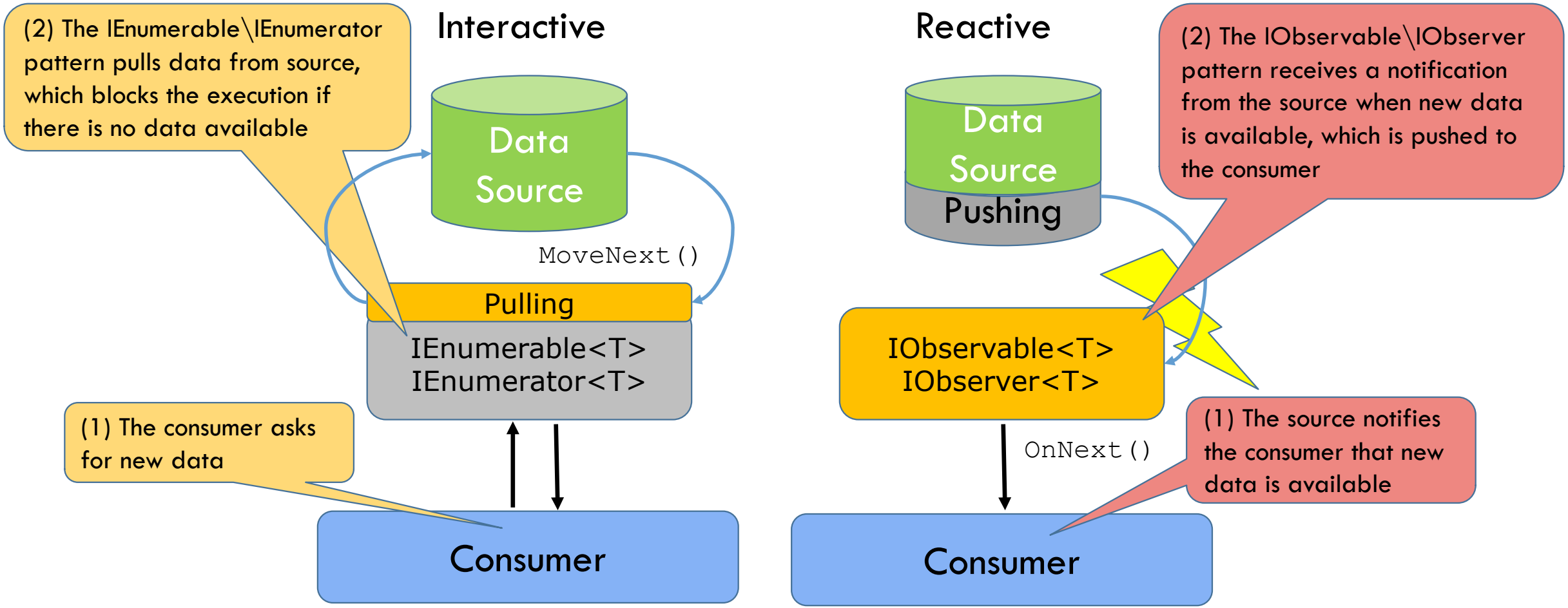
- Abstract over sources of asynchrony and time
  - Threads, thread pools, tasks, message loops
  - DateTimeOffset.UtcNow, timers

One versus Many

- When Task and Task<T> were born…
  - Single-value specializations
  - Await-ability of sequences (for aggregates)

	Synchronous	Asynchronous
One	`Func<T> f` `var x = wait f();`	`Task<T> t` `var x = await t;`

Many	`IEnumerable<T> xs`
	`foreach (var x in xs) {`
	`  f(x);`
	`}`
		`IObservable<T> xs`
		`xs.Subscribe(x => {`
		`  f(x);`
		`});`


Interactive
(2) The IEnumerable\IEnumerator pattern pulls data from source, which blocks the execution if there is no data available
Data Source
MoveNext()
Pulling
IEnumerable<T>
IEnumerator<T>
(1) The consumer asks for new data
Consumer
Reactive
Data Source
Pushing
(2) The IObservable\IObserver pattern receives a notification from the source when new data is available, which is pushed to the consumer
IObservable<T>
IObserver<T>
OnNext()
(1) The source notifies the consumer that new data is available
Consumer

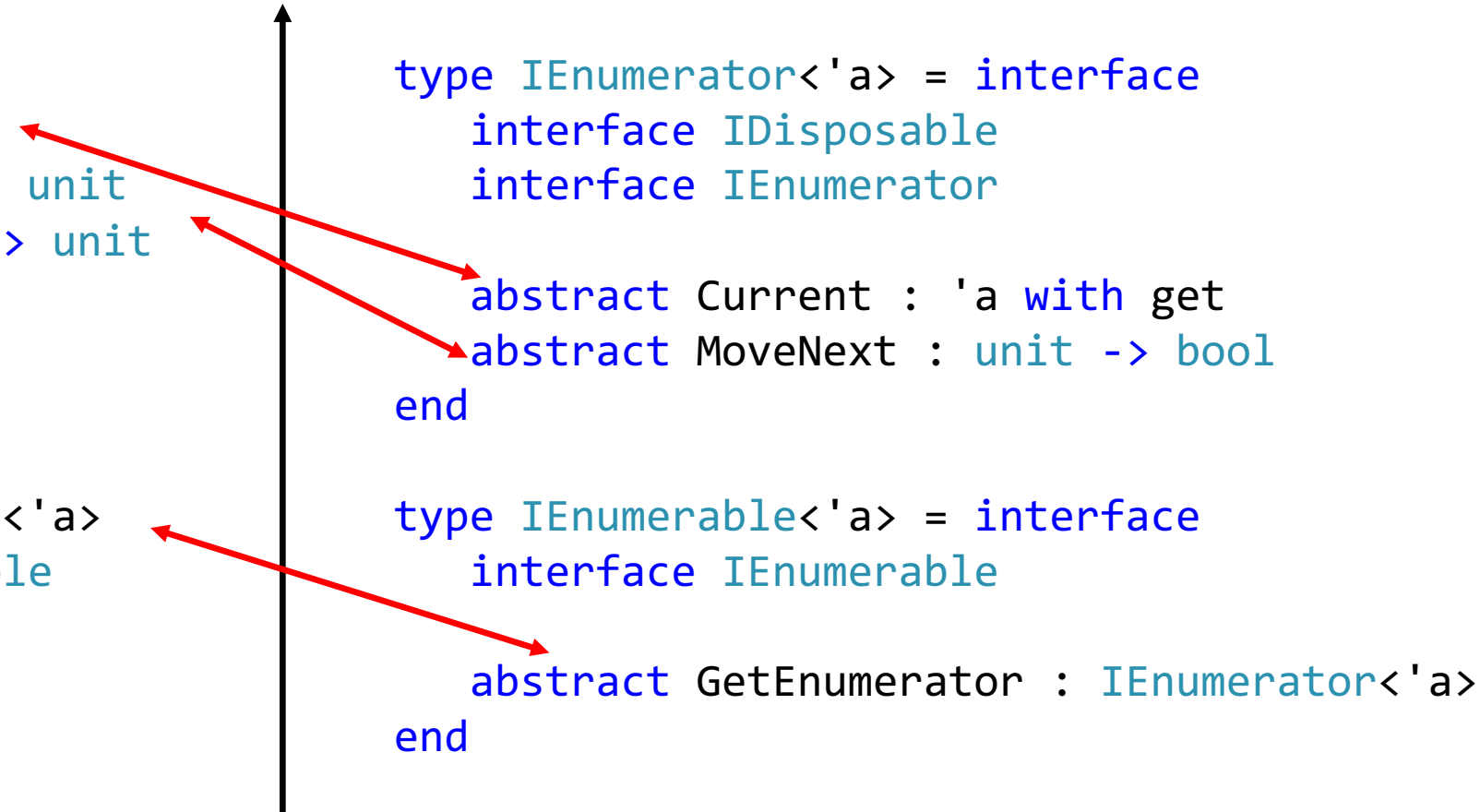
IObservable the dual of IEnumerable

Category theory to the rescue (Bierman, Meijer)

Observable/observer (push) is dual to enumerable/enumerator (pull)

Cross-influence of both domains

```
type IObserver<'a> = interface
  abstract OnNext : 'a with set
  abstract OnCompleted : unit -> unit
  abstract OnError : Exception -> unit
end


type IObservable<'a> = interface
  abstract Subscribe : IObserver<'a>
                    -> IDisposable
end
```



```
type IEnumerator<'a> = interface
   interface IDisposable
   interface IEnumerator

   abstract Current : 'a with get
   abstract MoveNext : unit -> bool
end

type IEnumerable<'a> = interface
   interface IEnumerable

   abstract GetEnumerator : IEnumerator<'a>
end
```

Stream processing basics - Event

- Not first class in C#
- Hard to compose

C#

```
button.Click += OnClick;

private void OnClick(object sender, RoutedEventArgs e)
{
    …
}
```

Stream processing basics – F# Event

- First class in F#
- Can have basic composition

```
button1.Click
    |> Event.merge button2.Click
    |> Event.add (fun a -> MessageBox.Show "Hii"
                           |> ignore)
```

F#

Stream processing basics – IObservable/IObserver

C#

```
var click1 = Observable
    .FromEventPattern(button1, nameof(button1.Click));

var click2 = Observable
    .FromEventPattern(button2, nameof(button2.Click));

Observable.Merge(click1, click2).Subscribe(OnClick);
```

Stream processing basics – IObservable/IObserver

F#

```
let button1 = Button()
let button2 = Button()

button1.Click
|> Observable.merge button2.Click
|> Observable.add (fun a -> MessageBox.Show "Hii“
                            |> ignore)
```

Stream processing basics – Combinators

F#

```
let observable = Observable
    .Interval(TimeSpan.FromSeconds(1.0))

observable.Subscribe (printfn "%d")
```

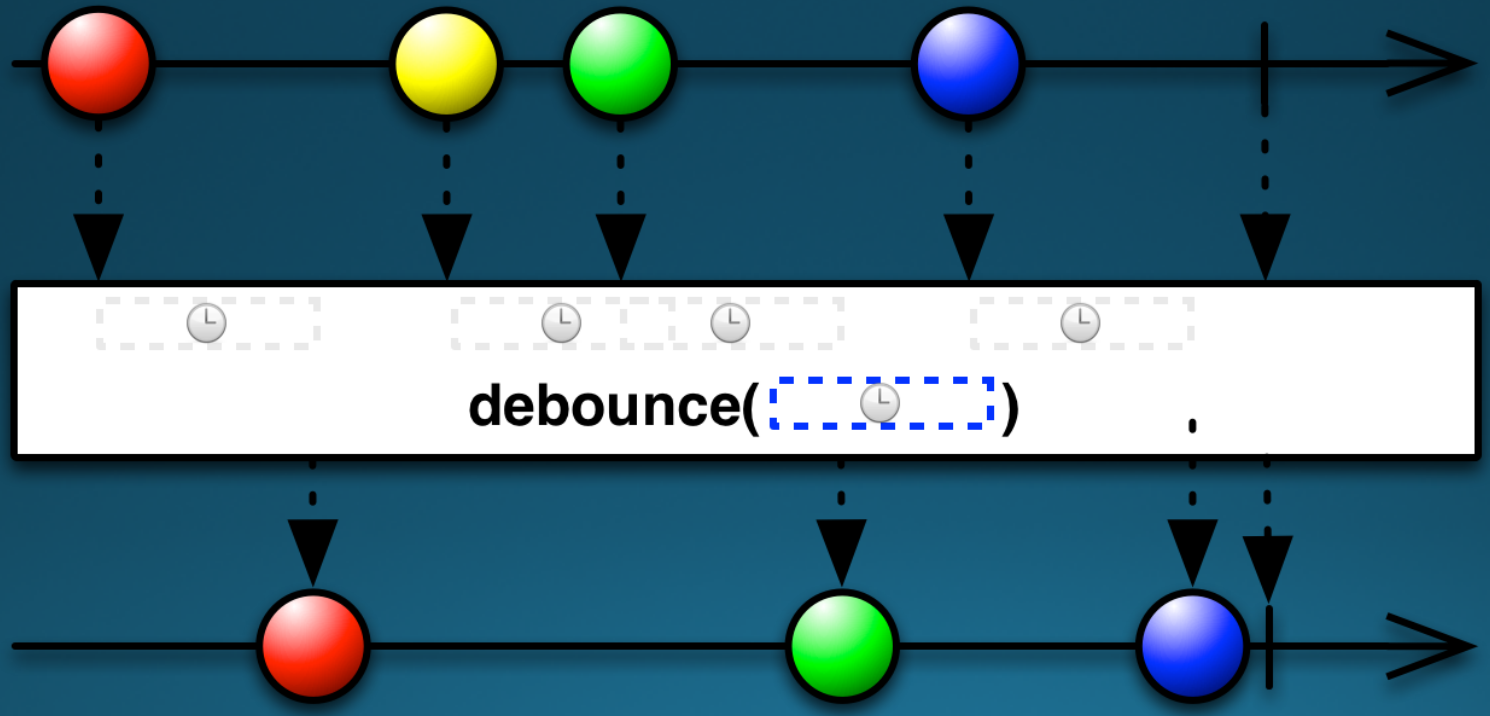
Stream processing basics – Combinators

F#

```
let observable = Observable
    .Interval(TimeSpan.FromSeconds(1.0))
    .GroupBy(fun x -> x % 3L)

let printKeyValue key value =
    printfn "%s %d" (String('*', key)) value
    ()

observable.Subscribe(fun obs ->
    obs.Subscribe (printKeyValue (int obs.Key)) |> ignore)
```

Stream processing basics – Combinators

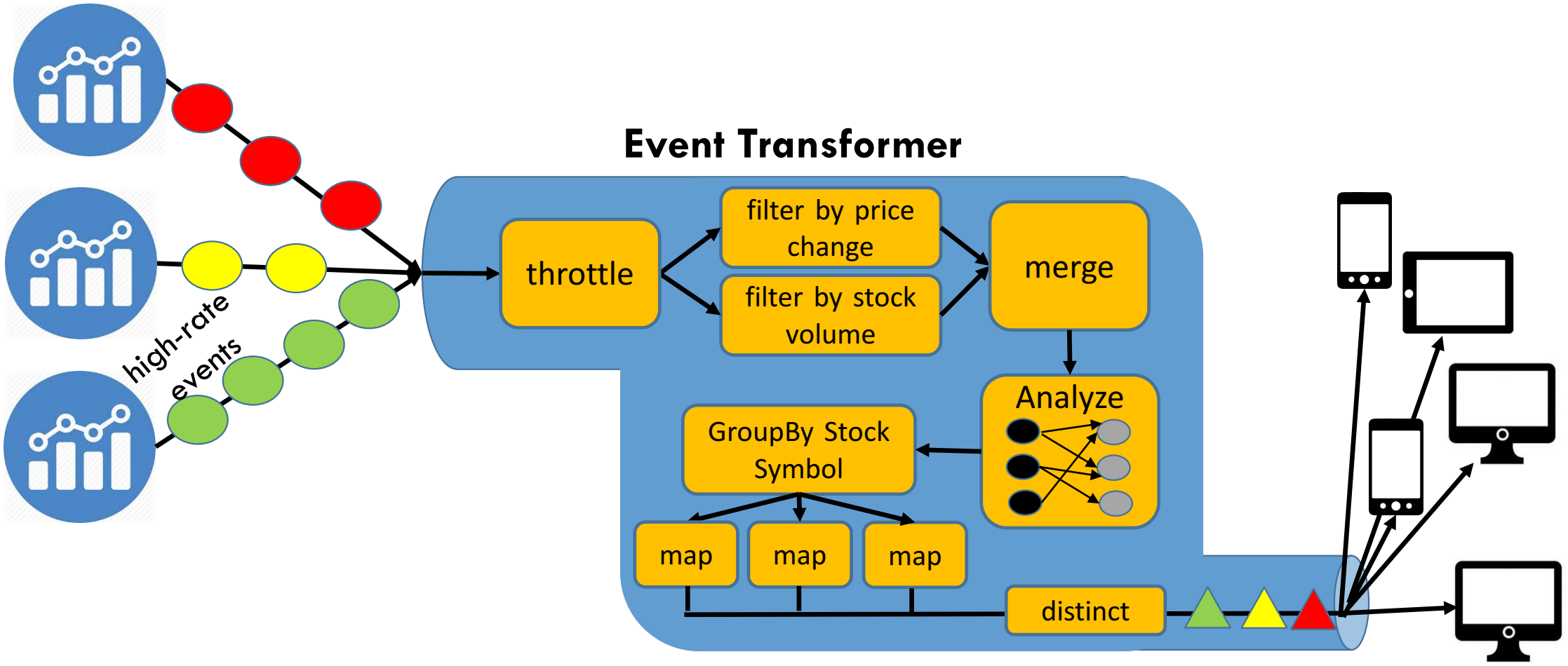
- **Throttle** — only emit an item from an Observable if a particular timespan has passed without it emitting another item

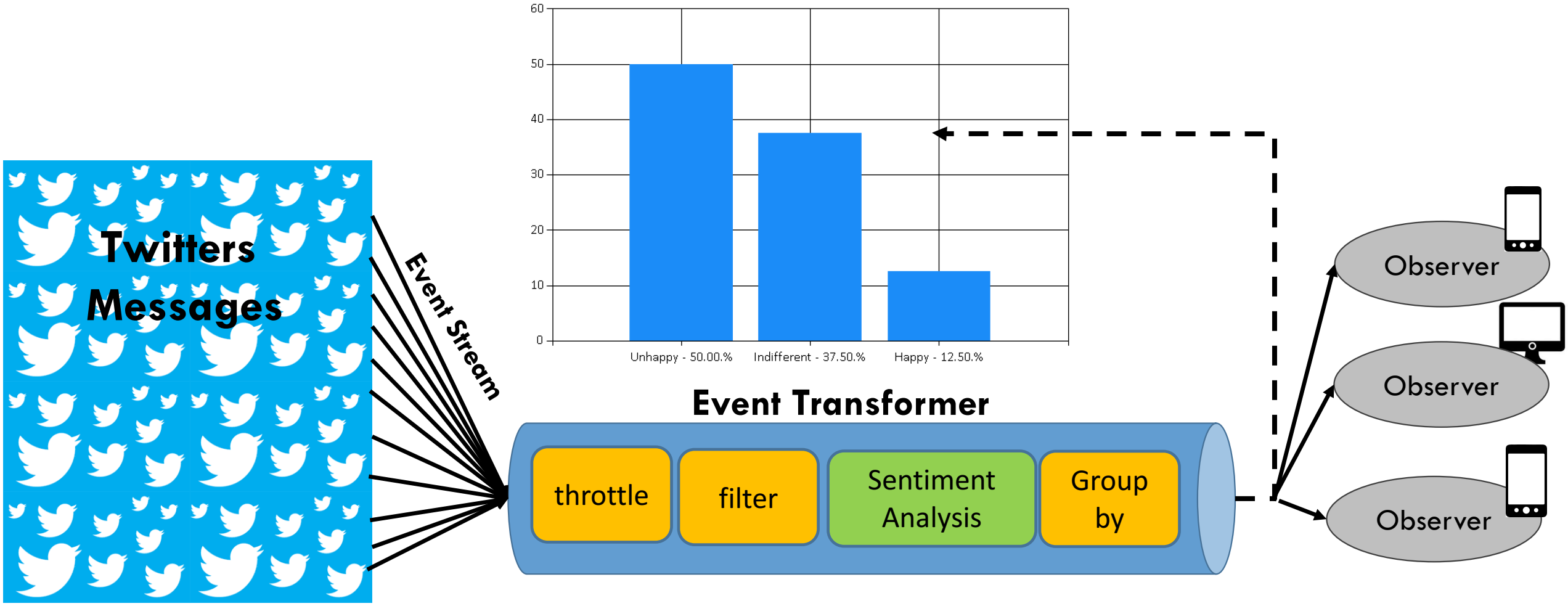


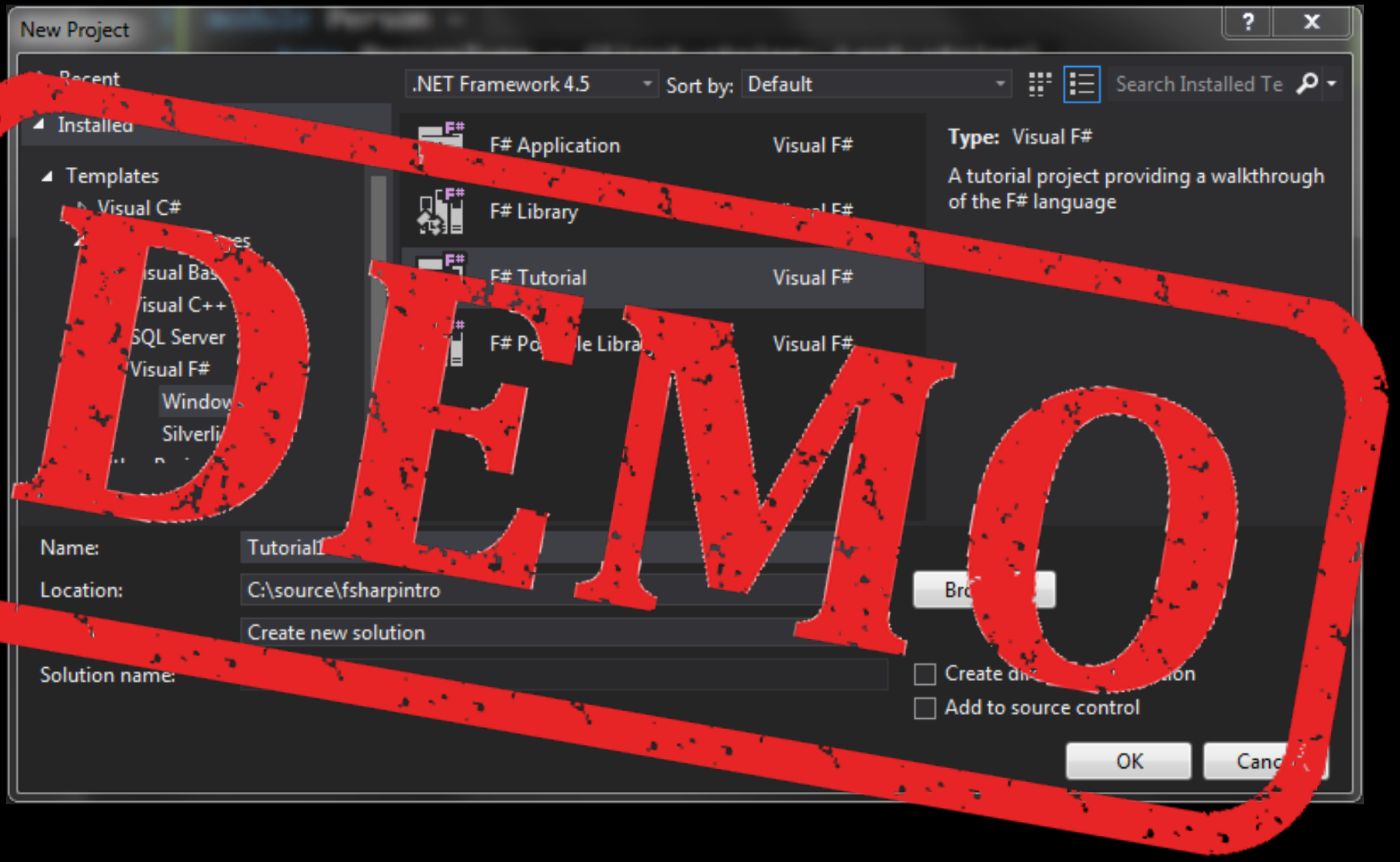
Stream processing basics – Combinators

F#

```
let observable = Observable
    .FromEventPattern(txt, "TextChanged")
    .Throttle(TimeSpan.FromSeconds(1.0))

observable
    .ObserveOn(SynchronizationContext.Current)
    .Subscribe (fun text -> console.Text <- console.Text +
                            Environment.NewLine + txt.Text )
```


Event Transformer
high-rate events
throttle
filter by price change
filter by stock volume
merge
Analyze
GroupBy Stock Symbol
map
map
map
distinct

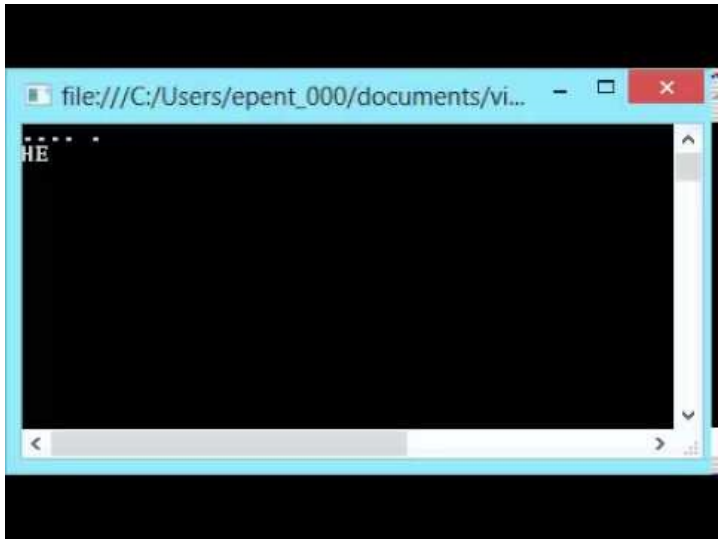

Twitters Messages
Event Stream
60
50
40
30
20
10
0
Unhappy - 50.00.%
Indifferent - 37.50.%
Happy - 12.50.%
Event Transformer
throttle
filter
Sentiment Analysis
Group by
Observer
Observer
Observer


New Project
.NET Framework 4.5
Sort by:
Default
Search Installed Te
Installed
Templates
Visual C#
Visual C++
SQL Server
Visual F#
Silverli
F# Application
Visual F#
F# Library
F# Tutorial
Visual F#
Visual F#
Type: Visual F#
A tutorial project providing a walkthrough of the F# language
Name:
Tutorial1
Location:
C:\source\fsharpintro
Create new solution
Solution name:
Add to source control
OK
DEMO

Lab – Stock market Twitter emotion

Exercise

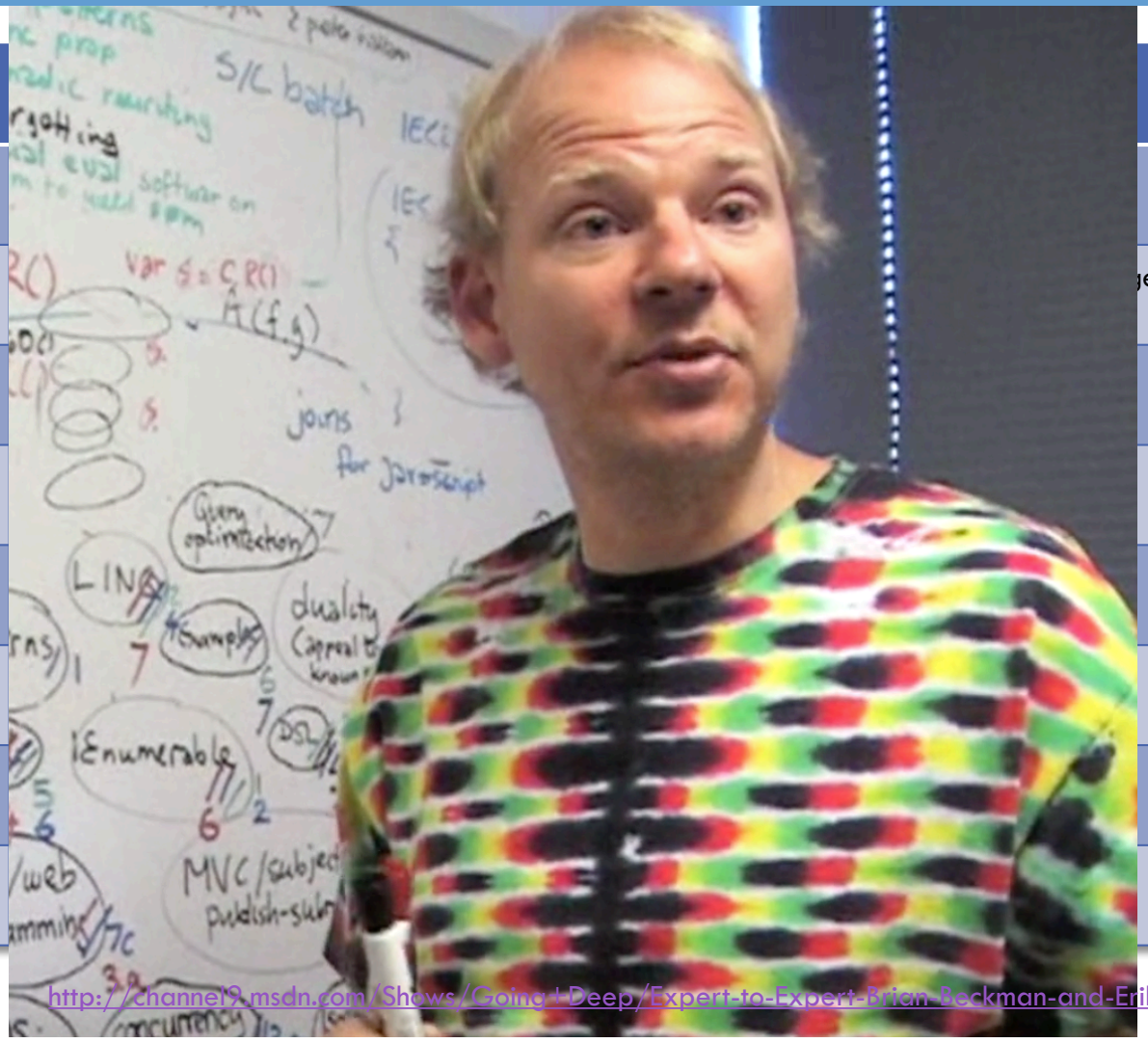
- Write console app
- The user input is English text
- While the user is typing print in the next line the The translation to Morse code
- http://en.wikipedia.org/wiki/Morse_code
- **Bonus:** Do it the opposite way



F# & .Net RX Api

F# Observable
add
choose
filter
merge
pairwise
partition
scan
subscribe

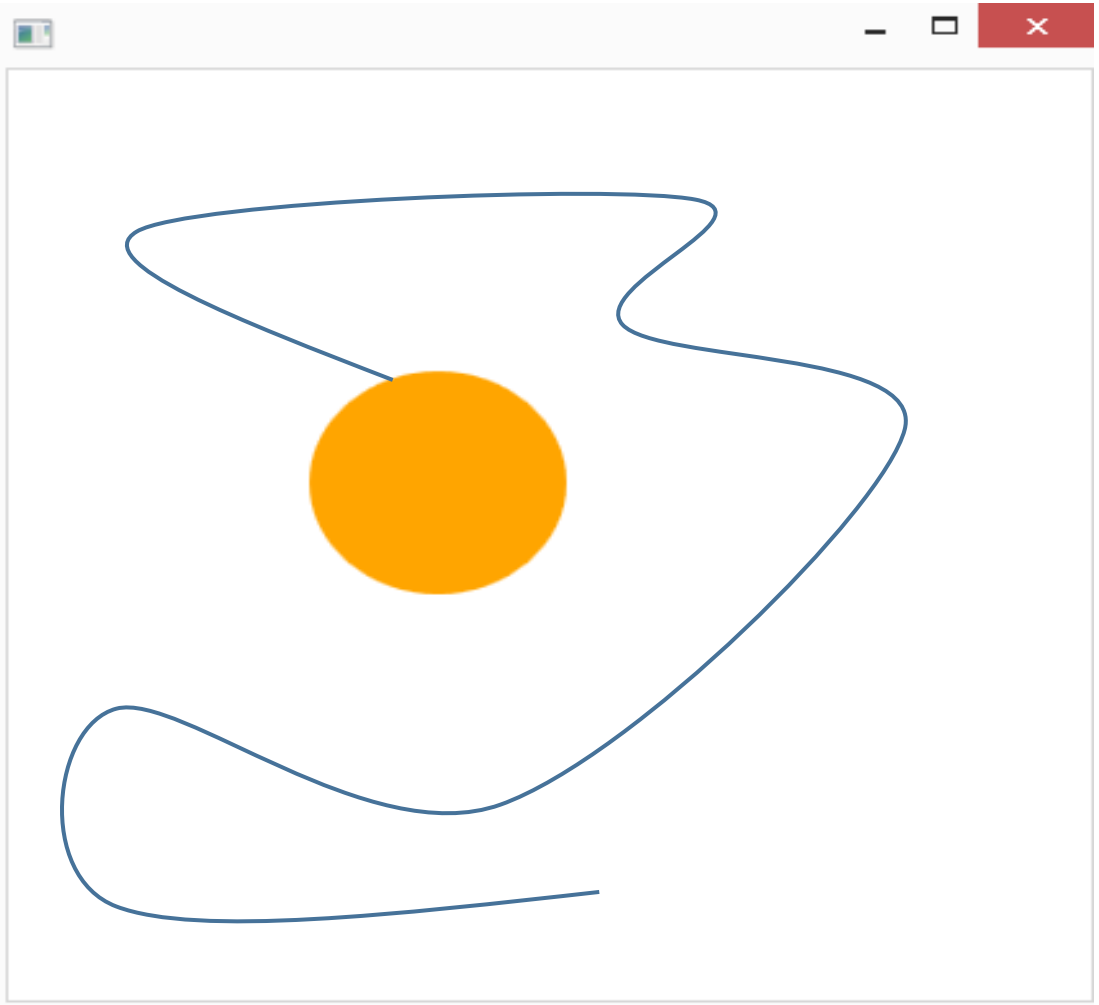
.NET RX Api	
All	Amb
Combine	Count
Finally	First
Last	Latest
Multicast	Range
Switch	Select
Take	TakeLast
Window	When

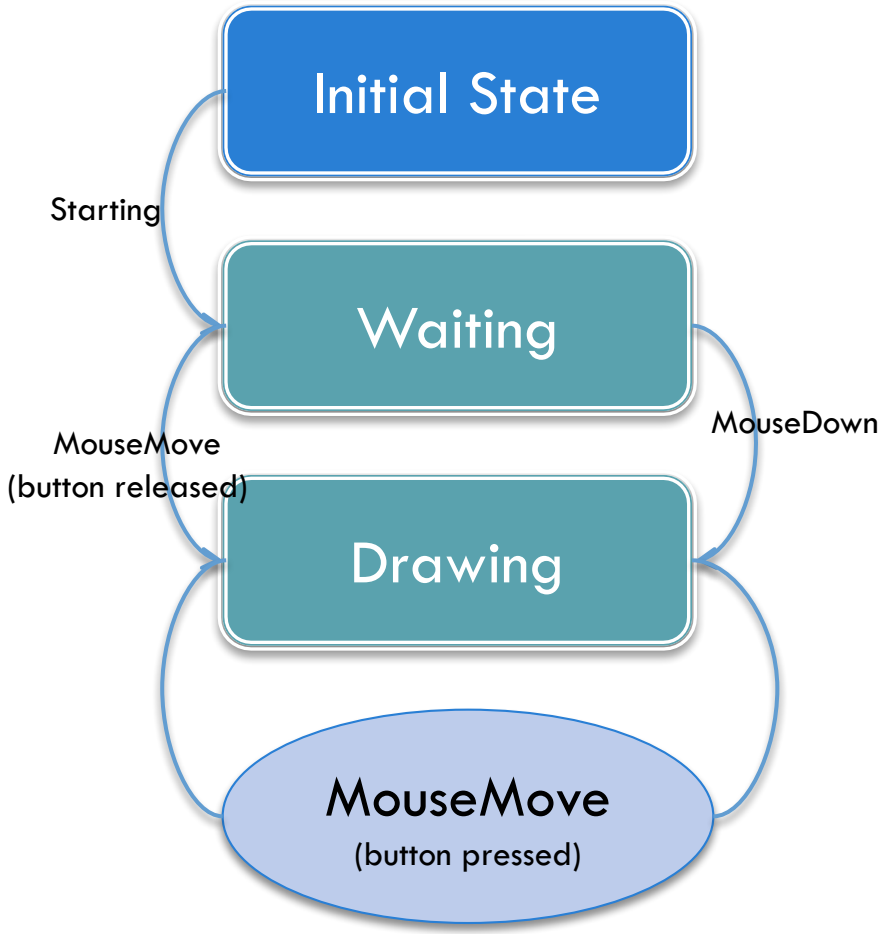


	Catch	Chunhify
ged	ElemntAt	ElementAtOrDefault
	If	IgnoreElement
	MinBy	MostRecent
	Never	Sum
	SkipUntill	Then
	While	When

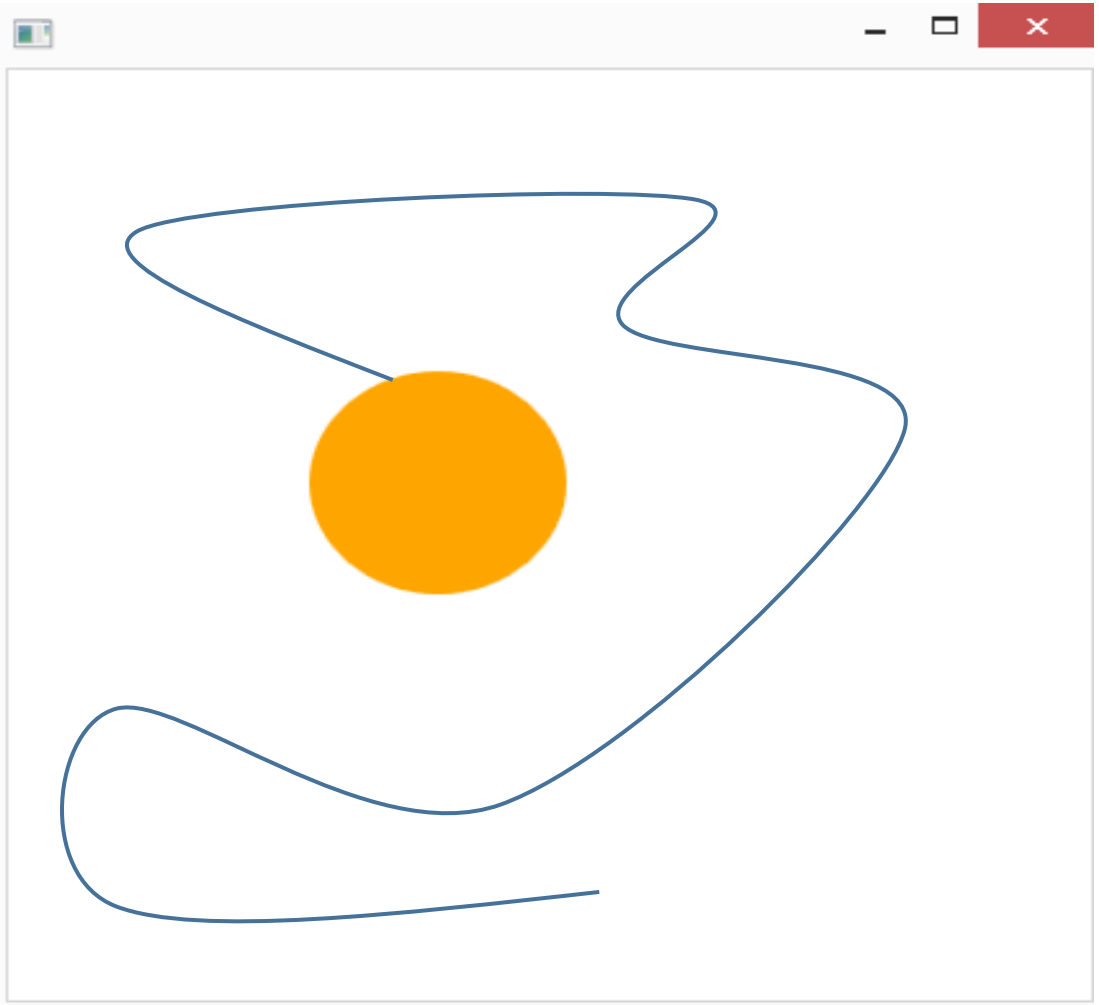
http://channel9.msdn.com/Shows/Going+Deep/Expert-to-Expert-Brian-Beckman-and-Erik-Meijer-Inside-the-NET-Reactive-Framework-Rx

Reactive Ball with undo





Reactive Ball with undo



Tasks

1. Add drawing line functionality when mouse is pressed
2. While drawing the line collect coordinates
3. Create undo (memento pattern) functionality.
    1. When the mouse is release, the Ball should replay backward the path following the line (slowly for animation)

The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities.

-- Edsger Dijkstra