Subject/Observer is Dual to Iterator (Erik "Head in the Box" Meijer, Microsoft Corporation, emeijer@microsoft.com)

The concept of *duality* is a very powerful trick that provides "buy one, get one free" in mathematics and engineering. For example, the well-known law of De Morgan, exploits the duality between conjunction && and disjunction || to prove that negation ! distributes over both conjunction and disjunction:

```
!(a  && b) == !a || !b
!(a || b)  == !a && !b
```

Other examples in computer science are the duality between call-by-value and call-by-name, between induction and co-induction, between least and greatest fixed points, and between algebras and co-algebras.

We discovered that the archetypal enumerable collections exposed by the `IEnumerable<T>` and `IEnumerator<T>` pair of interfaces are dual to observable collections exposed by the `IObservable<T>` and `IObserver<T>` pair of interfaces. The former embody the protocol for *interactive* computations where the consumer synchronously *pulls* data from the producer, while the latter embody the protocol for *reactive* computations where the producer asynchronously *pushes* data to the consumer. As far as we are aware the fact that the *Observer* and *Iterator* design patterns are mathematical duals has not been observed before; the standard design pattern literature does not even list *Iterator* and *Observer* as related patterns.

To derive observable reactive collections from interactive enumerable collections, we start with the well-known *Iterator* design pattern, as embodied in the .NET framework via the pair of `IEnumerable<out T>` and `IEnumerator<out T>` interfaces (Java has a very similar pair of interfaces `Iterable<T>` and `Iterator<T>`):

```
interface IEnumerable<out T>
{
   IEnumerator<T> GetEnumerator();
}

interface IEnumerator<out T>: IDisposable
{
   bool MoveNext();   // throws Exception
   T Current { get; }
}
```

We ignore the `Reset` method and the legacy non-generic base interfaces `IEnumerable` and `IEnumerator`. Note that the `MoveNext` method can throw an exception (in the equivalent Java `Iterator` interface this fact is apparent in the `throws` clause, strangely enough on the `next()` method), besides returning a proper value. A more precise return type for `MoveNext` would be the disjoint sum `bool+Exception`.

Taking the definition of categorical duality from Wikipedia literally, we completely mechanically swap the arguments and results of all method signatures. The only small twist is that we dualize just the `IEnumerator` interface, but leave the `IDisposable` aspect of `IEnumerator` that is returned from `GetEnumerator` invariant. Doing this, we arrive at the following two intermediate interfaces:

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

interface IObserver<in T>
{
  void OnCompleted(bool done);
  void OnError(Exception exception);
  T OnNext { set; }
}
```

We streamline these unthinkingly derived interfaces by observing that the (implicit) protocol for `IEnumerator` is that the environment signals that there will be no more values by returning `false` from `MoveNext`, so dualizing that operational behavior, the boolean argument of the `OnCompleted` method is redundant and by just calling `OnCompleted` (without any arguments) the environment signals that there will be no more values. Instead of a write-only property, we turn `OnNext` into a method.

Surprisingly, after this refactoring, we obtain a pair of interfaces for the familiar *Observer* design pattern:

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

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

Interestingly, the interfaces that we derived guided by mathematics are even better than the typical observer/observable APIs. Operationally, for enumerable collections, we undo the effect of calling `GetEnumerator()` by disposing the returned enumerator. For observable collections, we undo the effect of `Subscribe`-ing an observer by disposing the handle that is returned from the subscription. This seemingly simple technique is essential to allow composition of operations on observable collections, which is hard to achieve using `Add`/`Remove` mechanism of .NET delegates, or the usual `void deleteObserver(Observer o)` method for removing observers from an observable.

For example, consider the filter combinator `IObservable<T> Where<T>(this IObservable<T> source, Func<T, bool> predicate)`, that returns a new observable (using the factory method `Observable.Create`) that when you subscribe to it, will subscribe to the source with an anonymous observer (using the factory method `Observer.Create`) that will filter out all values for which the predicate does not hold:

```
IObservable<T> Where(this IObserver<T> source, Func<T, bool> predicate)
{
   return Observable.Create<T>(observer =>
   {
      return source.Subscribe(Observer.Create<T>(value =>
      {
         try { if(predicate(value)) observer.OnNext(value); }
         catch (Exception e) { observer.OnError(e); }
      });
   };
}
```

Thanks to the fact that subscribing to an observable returns a disposable object to unsubscribe, the above implementation is completely stateless, there is no need for the inner observer keep track of its observers so that they can be removed. That responsibility is simply delegated to the source observable, which ultimately bottoms out into a stateful object such as a .NET event source that maintains a list of observers:

```
IObservable<IEvent<MouseEventArgs>> GetMouseMoves(this Control control)
{
   return Observable.Create<IEvent<MouseEventArgs>>(observer =>
   {
      MouseEventHandler handler = (s,e) => observer.OnNext(Event.Create(s,e));
      control.MouseMove += handler;
      return () => { control.MouseMove -= handler; };
   });
}
```

Based on the `IObserver<T>` and `IObservable<T>` interfaces we derived above, we have built a ~~monadic~~ LINQ-based framework for reactive programming called Rx. It is interesting to compare Rx with the traditional approach to reactive programming in the functional language community. As the Wikipedia entry mentions "the semantic model of FRP […] is typically in terms of *continuous functions* […] *over time*". Even discrete events are typically associated with time, and as a result of that choice (we believe) efficient implementations of FRP are still an open research problem.

In contrast, Rx completely sheds the notion of time from the notion of reactivity, and instead of focusing on time Rx is parameterized over *concurrency*. Not only are pull-based enumerable streams and push-based observable streams each other's dual, they are in fact isomorphic. When converting a pull-based stream into a push-based stream we need to *add* concurrency to prevent the `Subscribe` method from blocking while the values from the source stream are pushed on the target stream, while dually, when converting a push-based stream into a pull-based stream we need to *remove* concurrency to block the `MoveNext` call until the source pushed the next value. We use a similar abstraction for representing execution contexts as Java's ScheduledExecutorService interface.

```
interface IScheduler
{
    DateTimeOffset Now { get; }                          // local absolute time

    IDisposable Schedule(Action work)                    // work will execute eventually.
    IDisposable Schedule(Action work, TimeSpan dueTime)  // work will execute after dueTime.
}
```

It is still an open question where and how the math we used to derive `IObservable<T>` from `IEnumerable<T>` dictates the essential role of concurrency. Also, we are hoping for a similar strike of lightening for `IScheduler<T>` as we experienced when we discovered `IObservable<T>`.