LINQ, Take Two

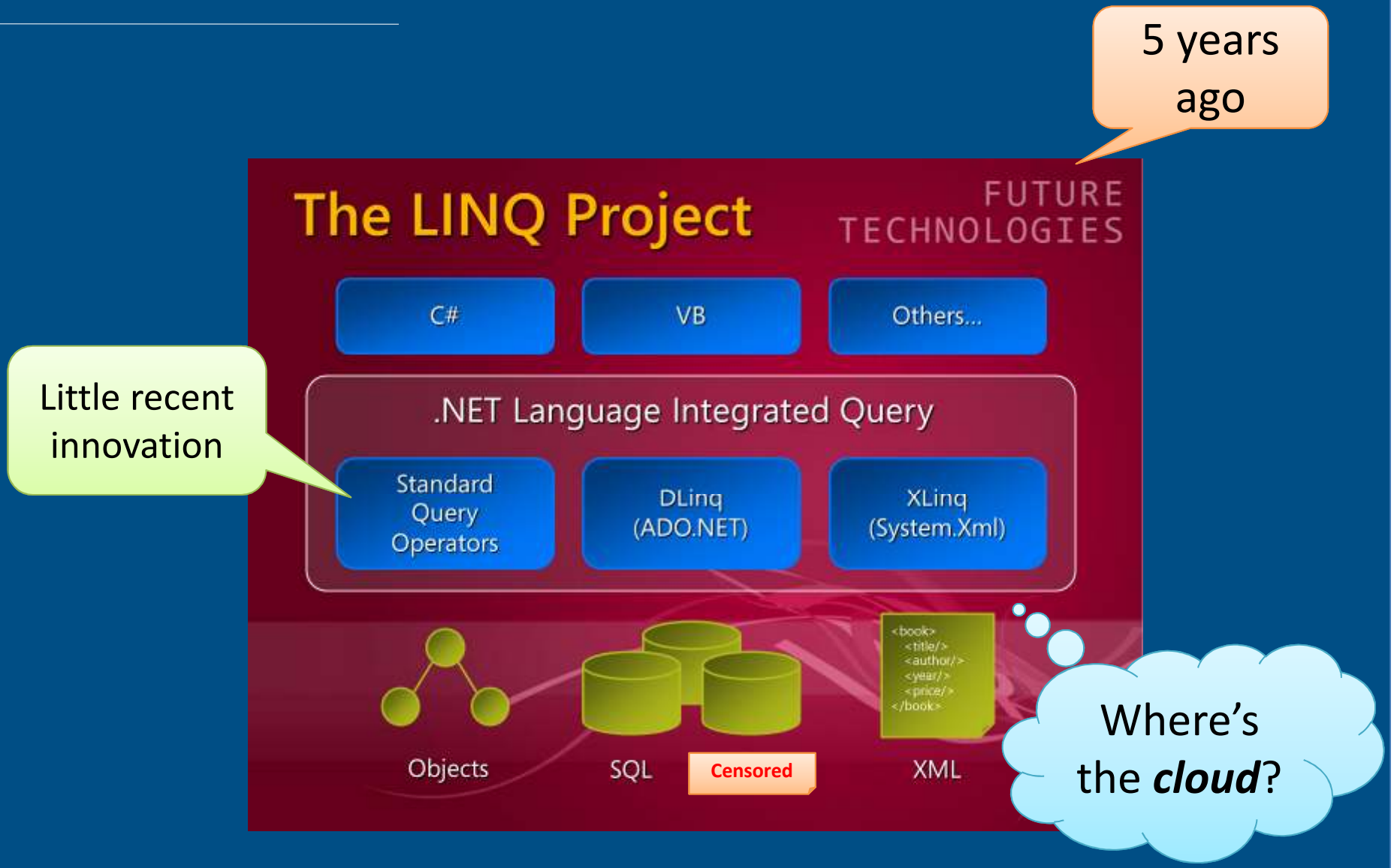
Realizing the LINQ to Everything Dream

Bart J.F. De Smet

Software Development Engineer

bartde@microsoft.com

A Historical Perspective



Language Integrated ~~Query~~ Monads

Monad (functional programming)

From Wikipedia, the free encyclopedia

In functional programming, a **monad** is a kind of abstract data type constructor used to represent computations (instead of data in the domain model). Monads allow the programmer to chain actions together to build a pipeline, in which each action is decorated with additional processing rules provided by the monad. Programs written in functional style can make use of monads to structure procedures that include sequenced operations,[1][2] or to define arbitrary control flows (like handling concurrency, continuations, or exceptions).

Formally, a monad is constructed by defining two operations (*bind* and *return*) and a type constructor *M* that must fulfill several properties to allow the correct composition of *monadic* functions (i.e. functions that use values from the monad as their uments). The *return* operation takes a value from a plain type and puts it into a monadic container of type eration performs the reverse process, extracting the original value from the container and passing it to the as in the pipeline.

A programmer will compose monadic functions to define a data-processing pipeline. The monad acts as a framework, as it's a reusable behavior that decides the order in which the specific monadic functions in the pipeline are called, and manages all the undercover work required by the computation.[3] The bind and return operators interleaved in the pipeline will be executed after each monadic function returns control, and will take care of the particular aspects handled by the monad.

The name is taken from the mathematical monad construct in category theory.

Why?

What?

How?

The Monadic Bind Operator Revealed

Could there be ***more***?

IEnumerable<T>
IQueryable<T>

new[]
{ 42 }

Definition [edit]

A *monad* is defined by three things:

- a way to produce types of "actions" from the types of their result; formally, a type constructor `M`,
- a way to produce actions which simply produce a value; formally a function named `return`:

```
return :: a -> M a
```

- and a way to chain "actions" together, while allowing the result of an action to be used for the second action; formally, an operator `(>>=)`, which is pronounced "bind":

```
(>>=)  :: M a -> ( a -> M b ) -> M b
```

SelectMany

```
IEnumerable<R> SelectMany<T, R>(
    this IEnumerable<T> source,
    Func<T, IEnumerable<R>> selector)
```

Also see www.codeplex.com/LINQSQO for “Project MinLINQ”

Building the Maybe Monad (For Fun and No Profit)

Null-propagating dot

```
string s =
name?.ToUpper();
```

Syntactic sugar

```
from _ in name
from s in _.ToUpper()
select s
```

Compiler

```
name.SelectMany(
    _ => _.ToUpper(),
    s => s)
```

One single ***library function*** suffices

DEMO

Essential LINQ

Query Providers Revisited – Do We Need IQueryable<T>?

Implements **IQueryable<T>**

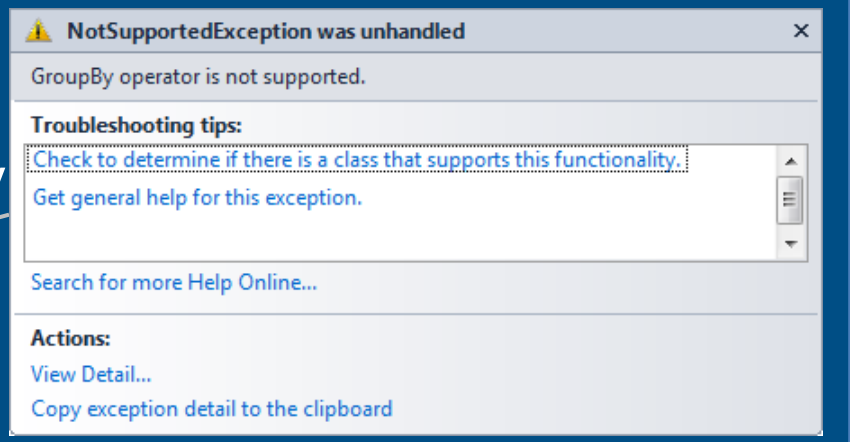
Does it really have to be a runtime check?

```
var src = new Source<Product>();

var res = from p in src
          where p.Price > 100m
          group p by p.Category

foreach (var p in res)
    Console.WriteLine(p);
```

Compiles fine



Leveraging The Query Pattern

```
var res = from p in src
          where p.Price > 100m
          group p by p.Category;
```

Syntactic sugar

```
var res = src
          .Where(p => p.Price > 100m)
          .GroupBy(p => p.Category);
```



Can be instance methods

No **GroupBy** "edge"

Taking It One Step Further

```
var res = from tweet in twitter
          where tweet.From == "B
          where tweet.About == "
          select tweet;
```

From
About
Location

Query *"learns"*

```
class TweetAboutFromLoc
{
  public FromString From;
  public AboutString About;
  public LocString Location;
}

class FromString
{
  FilterFrom operator ==(
    FromString f, string s)
}
```

"Has a" type

```
class TwitterByFrom
{
    public TwitterByAboutWhere(Where(Tweet<TweetAboutLoc, FilterFAbout> filter);
    // Other filter methods
}   // Fields with current filters
}
```

Custom *syntax trees*

DEMO

Query Providers Revisited

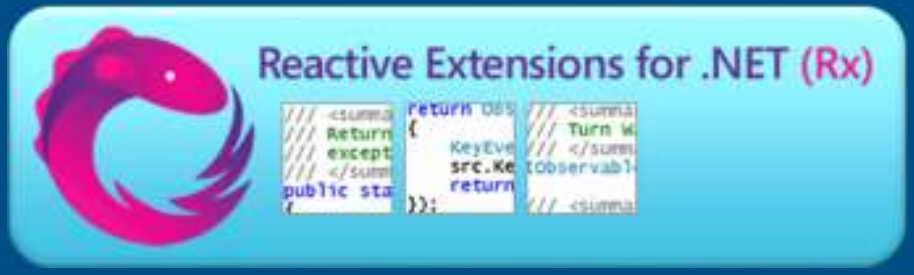
Event Processing Systems

Way ***simpler*** with Rx

$$(f \circ g)(x) = f(g(x))$$

Rx is a library for composing asynchronous and event-based programs using observable sequences.

Queries! **LINQ!**



- .NET 3.5 SP1, 4.0, and 4.5
- Silverlight 4, and 5
- Windows Phone 7 and 7.5
- JavaScript (RxJS)

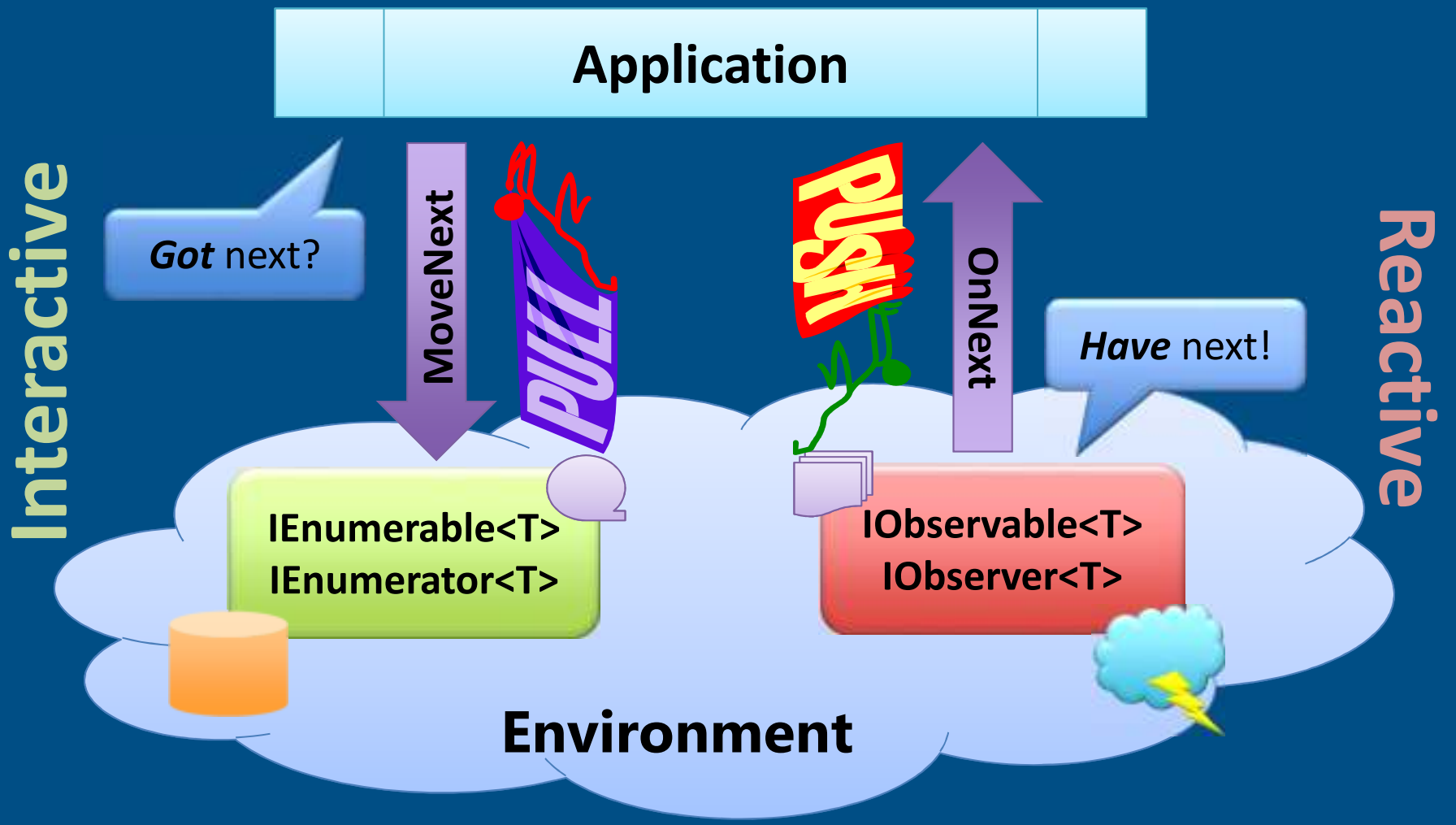
Download at **MSDN Data Developer Center** or use **NuGet**

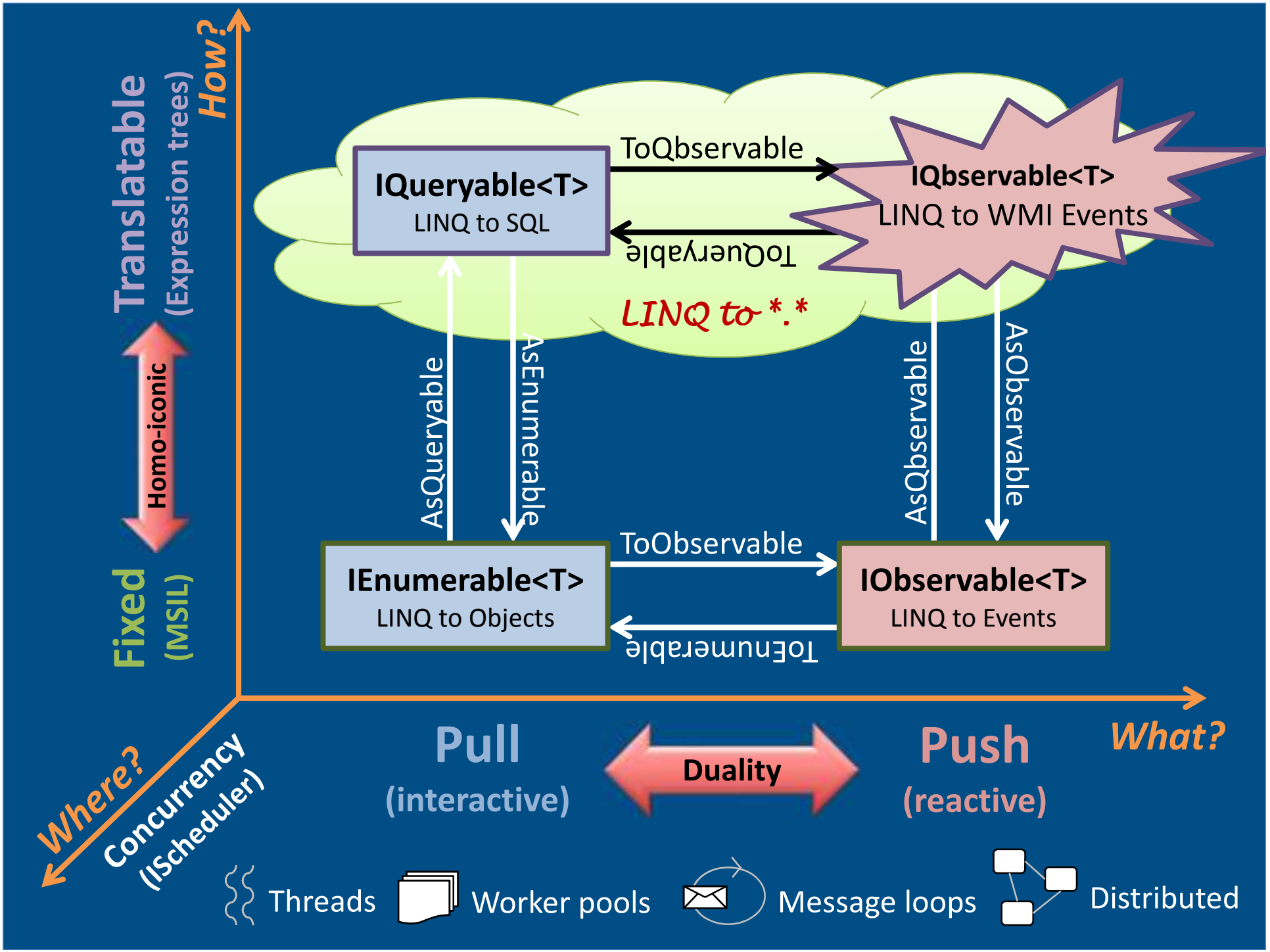
Observable Sequences

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

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

Push-Based Data Retrieval




How?
Translatable
(Expression trees)
Homo-iconic
Fixed
(MSIL)
IQueryable<T>
LINQ to SQL
ToQbservable
ToQueryable
IQbservable<T>
LINQ to WMI Events
LINQ to *.*
AsQueryable
AsEnumerable
AsQbservable
AsObservable
IEnumerable<T>
LINQ to Objects
ToObservable
ToEnumerable
IObservable<T>
LINQ to Events
What?
Pull
(interactive)
Duality
Push
(reactive)
Where?
Concurrency
(IScheduler)
Threads
Worker pools
Message loops
Distributed

IQbservable<T> – The Dual of IQueryable<T>

```
interface IQbservable<out T> : IObservable<T>
{
    Expression           Expression  { get; }
    Type                 ElementType { get; }
    IQbservableProvider  Provider    { get; }
}
```

We welcome ***semantic*** discussions

Extended role for some operators

No **Execute** method

```
interface IQbservableProvider
{
    IQbservable<R> CreateQuery<R>(Expression expression);
}
```

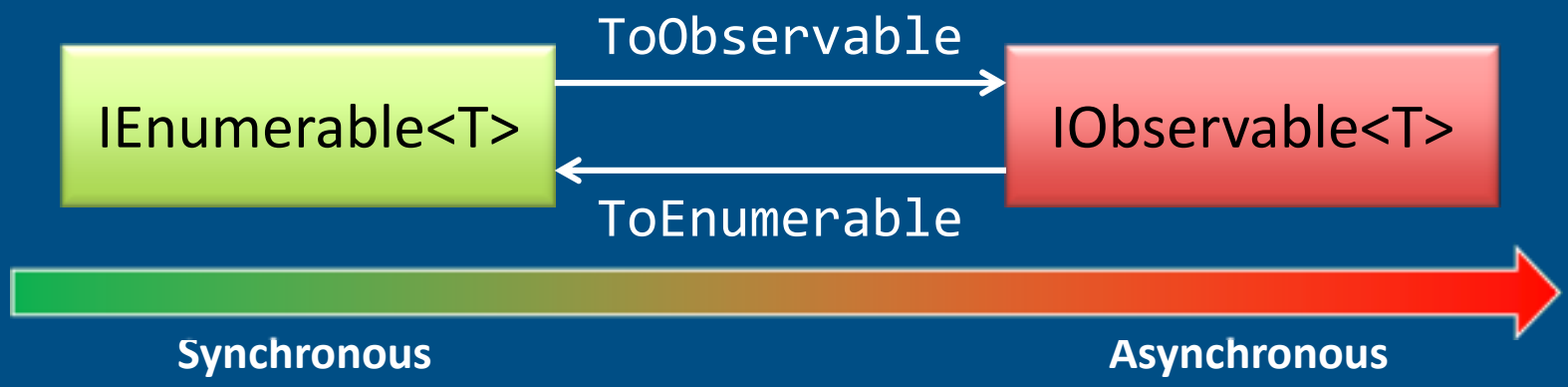
DEMO

LINQ to WMI Events (WQL)

Where Execution Happens – Introducing Schedulers

```
interface IScheduler
{
    DateTimeOffset Now { get; }
    IDisposable Schedule<T>(T state,
                            Func<IScheduler, T, IDisposable> action);
    IDisposable Schedule<T>(T state, TimeSpan dueTime,
                            Func<IScheduler, T, IDisposable> action);
    IDisposable Schedule<T>(T state, DateTimeOffset dueTime,
                            Func<IScheduler, T, IDisposable> action);
}
```

Rx's unified way to **introduce concurrency**



IScheduler Specifies Where Things Happen

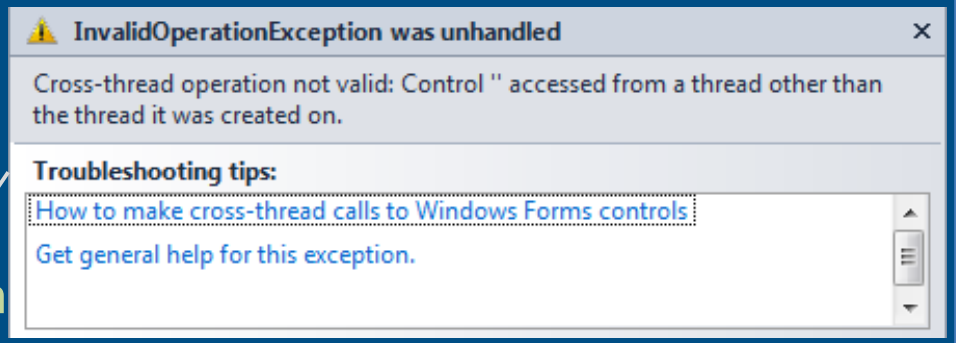
Imported TextBox **TextChanged** event

```
var res = from word in input.DistinctUntilChanged()
                            .Throttle(TimeSpan.FromSeconds(0.5))
          from words in lookup(word)
          select words;
```

Asynchronous ***web service call***

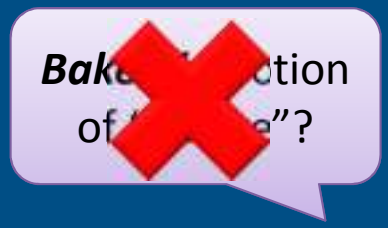
Indicates ***where*** things run

Use of scheduler to ***synchronize***




```
res.ObserveOn(wndControlSch
{
    lst.Items.Clear();
    lst.Items.AddRange((from word in words
                        select word.Word).ToArray());
});
```

IScheduler Parameterization



Entry-point for the ***schema***

```
var ctx = new NorthwindDataContext();
var res = from product in ctx.Products
          where product.Price > 100m
          select product.Name;
```

Custom schedulers could be ***very rich*** (e.g. server farm)

Decoupled “what” from “where”

```
foreach (var p in res.RemoteOn(new SqlScheduler(“server”)))
    // Process product
```

Expression Tree Remoting

Rx .NET

```
from ticker in stocks
where ticker.Symbol == “MSFT”
select ticker.Quote
```

Observable data source

Retargeting to AJAX

JSON serializer

RxJS

```
stocks
.Where(function (t) { return t.Symbol == “MSFT”; })
.Select(function (t) { return t.Quote; })
```

DEMO

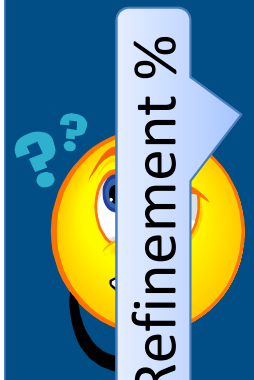
Remoting of Query Expressions

LINQ to the Unexpected

Microsoft® Solver Foundation

```
Model[
  Decisions[Reals, SA, VZ],
  Goals[
    Minimize[20 * SA + 15 * VZ]
  ],
  Constraints[
    C1 ->   0.3 * SA
          + 0.4 * VZ >= 2000,
    C2 ->   0.4 * SA
          + 0.2 * VZ >= 1500,
    C3 ->   0.2 * SA
          + 0.3 * VZ >= 500,
    C4 -> SA <= 9000,
    C5 -> VZ <= 6000,
    C6 -> SA >= 0,
    C7 -> VZ >= 0
  ]
]
```

Cost / barrel



Refinement %

Max # barrels

Min # barrels

Non-persistent

```
from m in ctx.CreateModel(new {
    vz = default(double),
    sa = default(double)
})
where    0.3 * m.sa
       + 0.4 * m.vz >= 2000
    &&   0.4 * m.sa
       + 0.2 * m.vz >= 1500
    &&   0.2 * m.sa
       + 0.3 * m.vz >= 500
where    0 <= m.sa && m.sa <= 9000
    &&   0 <= m.vz && m.vz <= 6000
orderby 20 * m.sa + 15 * m.vz
select m
```

To compute call **Solve**

LINQ to the Unexpected

SelectMany

```
from costSA in GetPriceMonitor("SA")
from costVZ in GetPriceMonitor("VZ")

from m in ctx.CreateModel(
                    new { vz = default(double),
                          sa = default(double) })
where 0.3 * m.sa + 0.4 * m.vz >= 2000
   && 0.4 * m.sa + 0.2 * m.vz >= 1500
   && 0.2 * m.sa + 0.3 * m.vz >= 500
where 0 <= m.sa && m.sa <= 9000
   && 0 <= m.vz && m.vz <= 6000

orderby costSA * m.sa + costVZ * m.vz
select m
```

Observable data sources

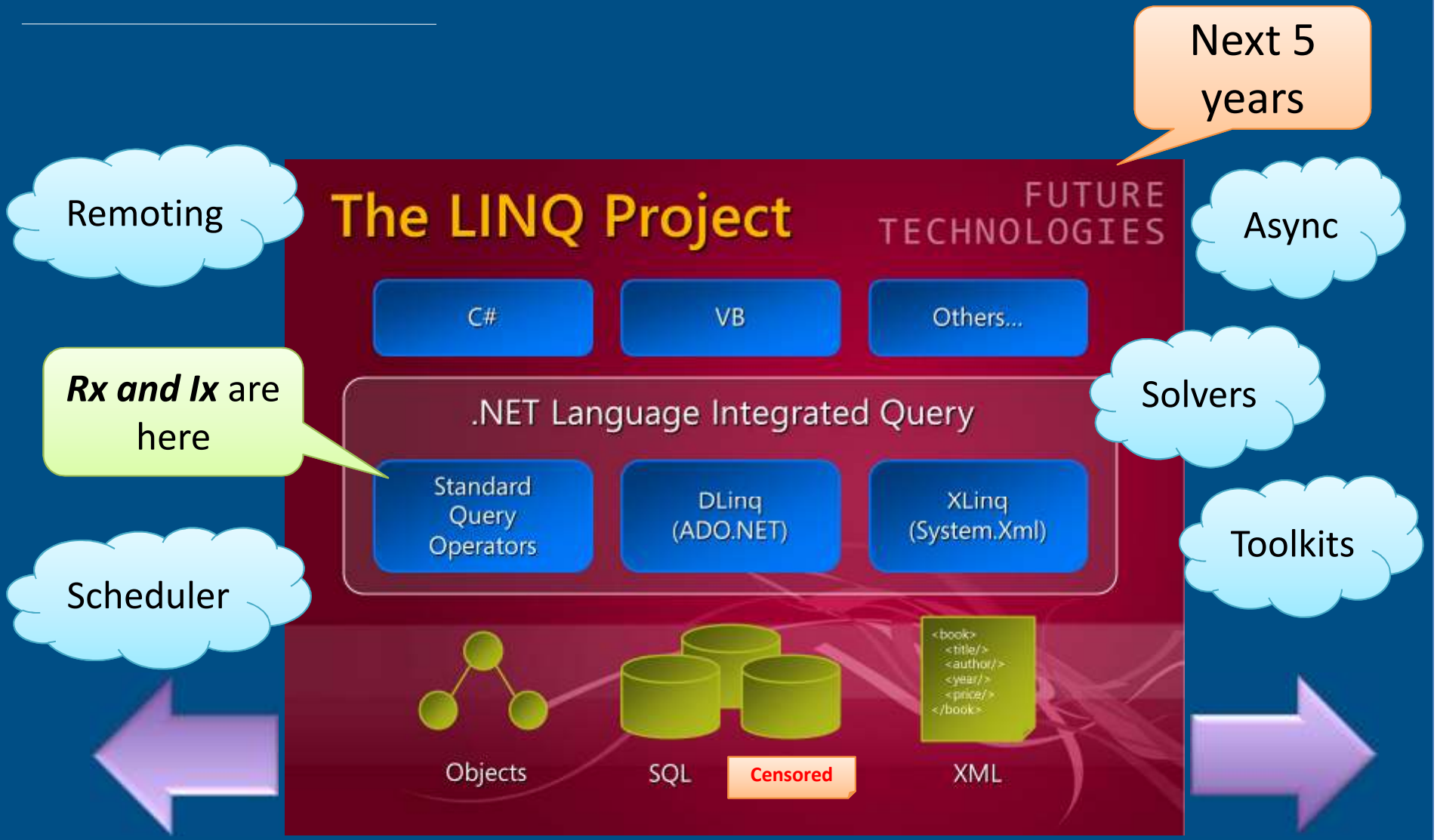
Subscribe here!

Parameters to decision

DEMO

Theorem Solving using Z3

A Futuristic Perspective



THANK YOU


PRAGUE
INTERNATIONAL
SOFTWARE DEVELOPMENT
CONFERENCE 2011
Conference : Nov. 22-23 // Training : Nov. 24
goto;
conference