Asynchronous Programming

with the Reactive Extensions

About me

MICHAEL LEHMANN

Lead Software Architect @ Zühlke

@lehmamic

Agenda

- Problematics in Asynchonous Programming
- Recipes to simplify asynchronous programming
- The Reactive Extensions
- Exerice

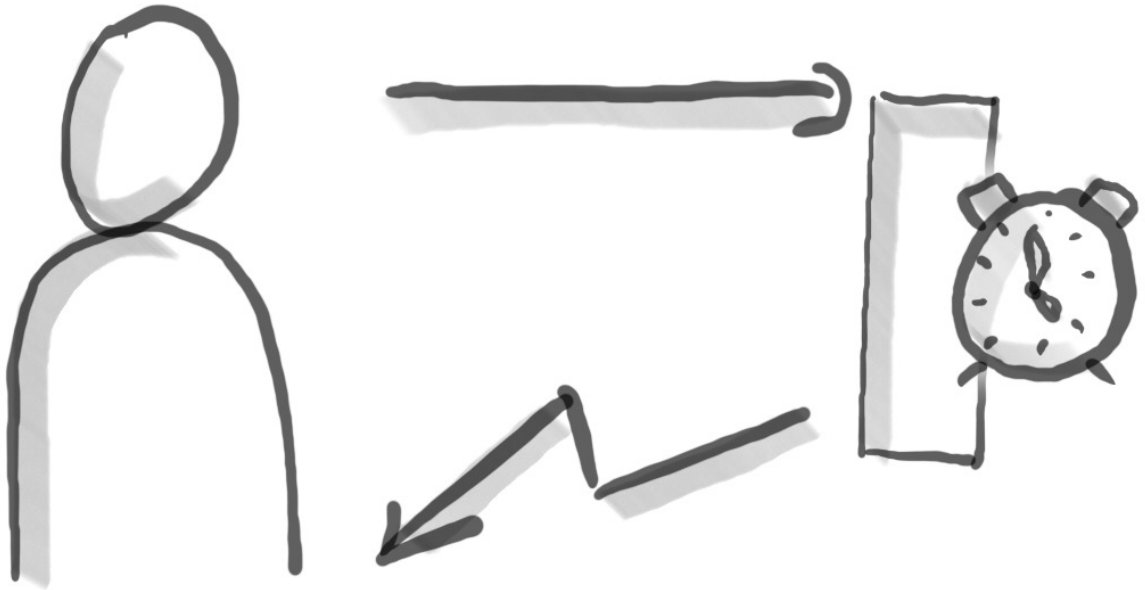
Problematics in Asynchonous Programming

Task: What makes async programming complex?

Asynchronous Programming is non-trivial

```
window.MouseMoved += (sender, e) =>
{
    Console.WriteLine($”X: {e.X}, Y: {e.Y}”);
};
```

We are **reacting** on events

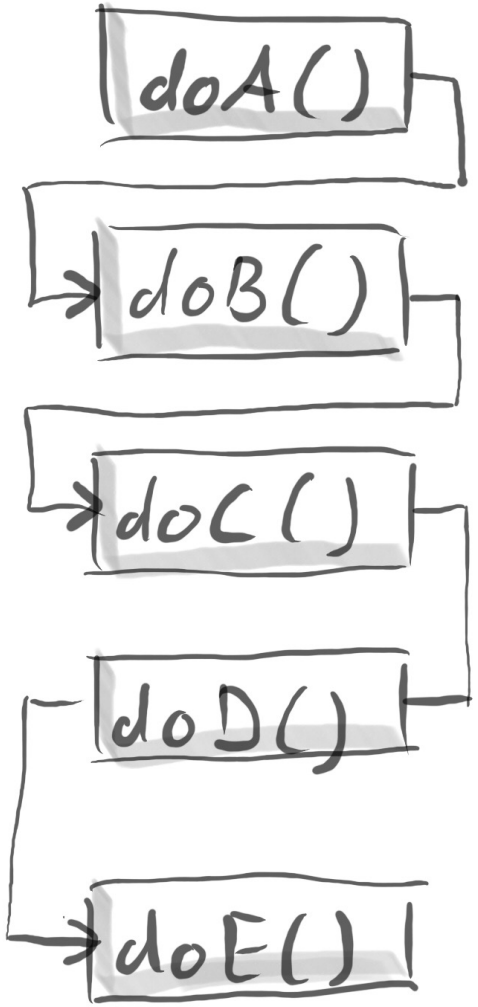


```
doB()
doB()
doC()
doD()
doD()
doE()
```

“doA() **then** doB() **then** doC() **then** doD() **then** doE()”

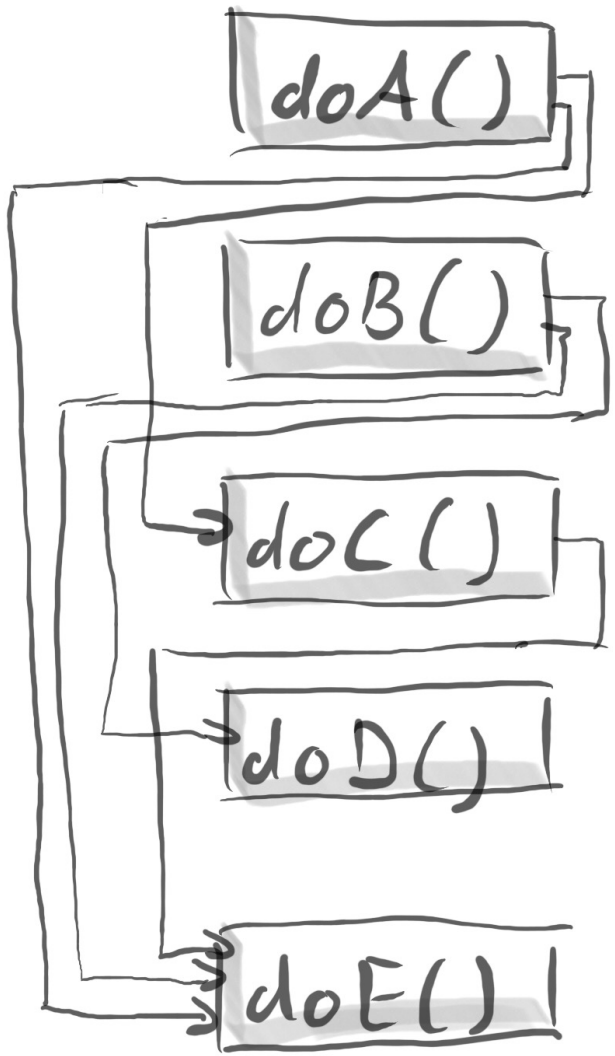
```
doB()
   .then(doB()
      .then(doC()
         .then(doD()
            .then(doE()))
         )
      )
   )
```

Sequential asyncronous calls



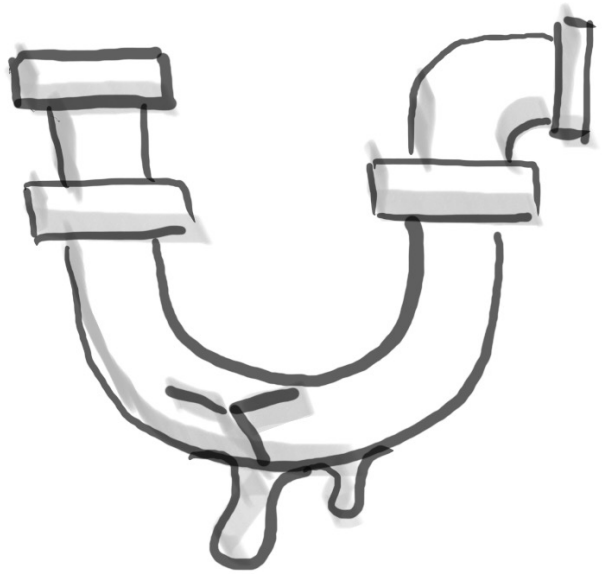
"doA() **then** doC(), **simultaneously** doB() **and then** doD(),
and after doA(), doB() **and** doC() **are done**, doE()"

Welcome in the **Callback Hell**

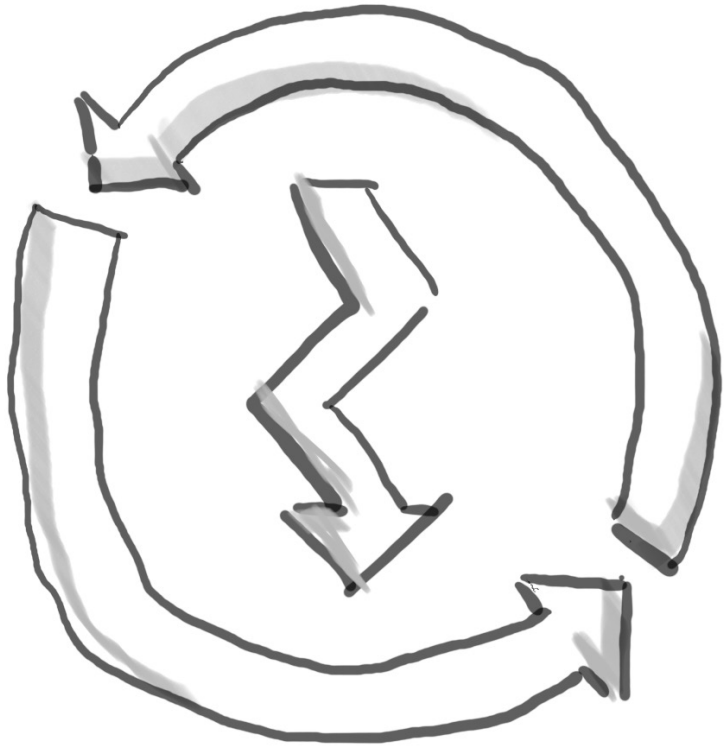


How do we **control** an asynchronous call sequence?

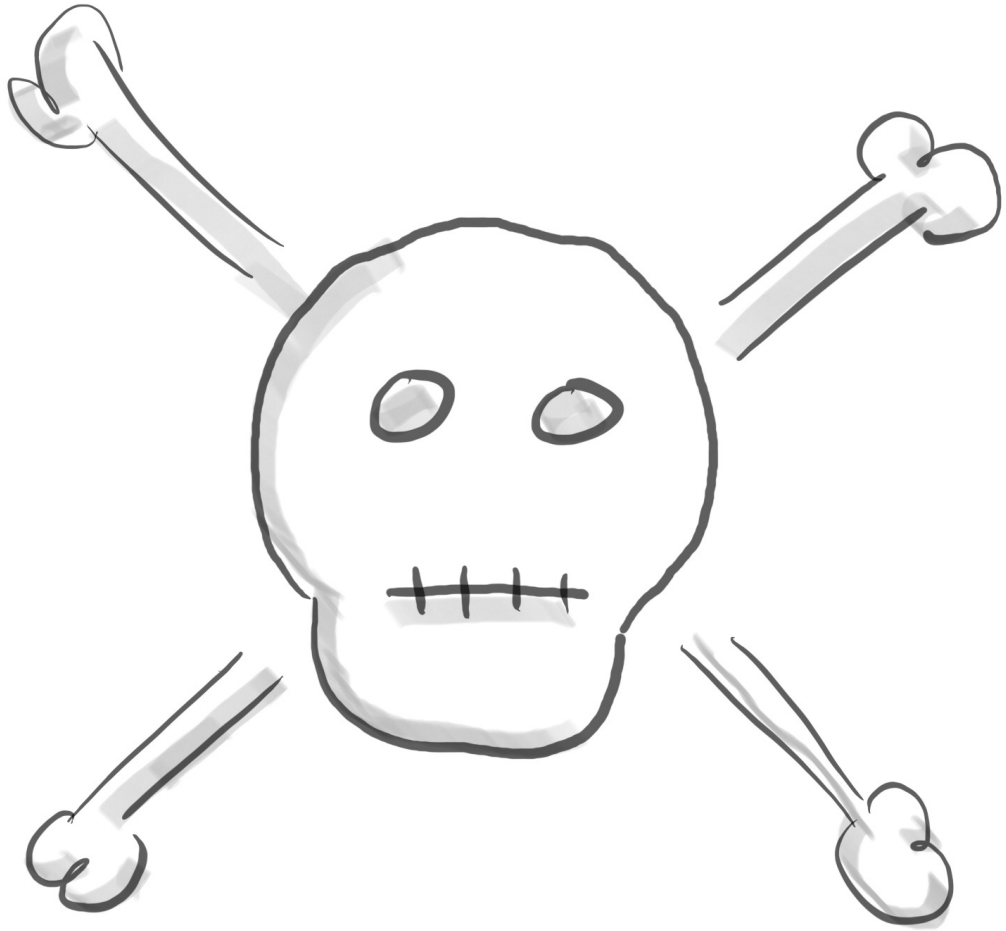
High chance for **Memory Leaks**



Most likely we'll have **Concurrency Issues**

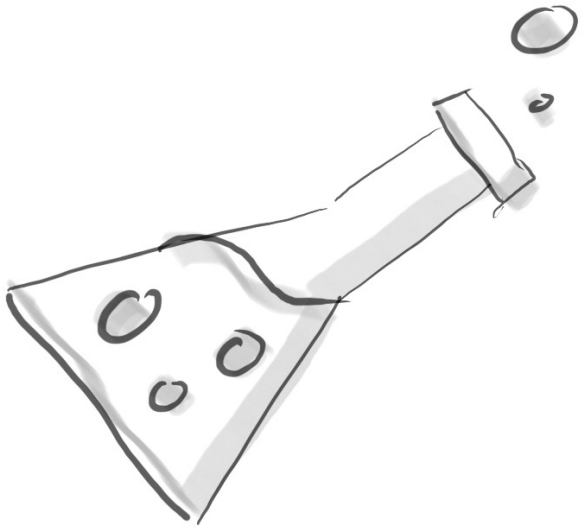


And … Deadlocks …



Debugging is a pain

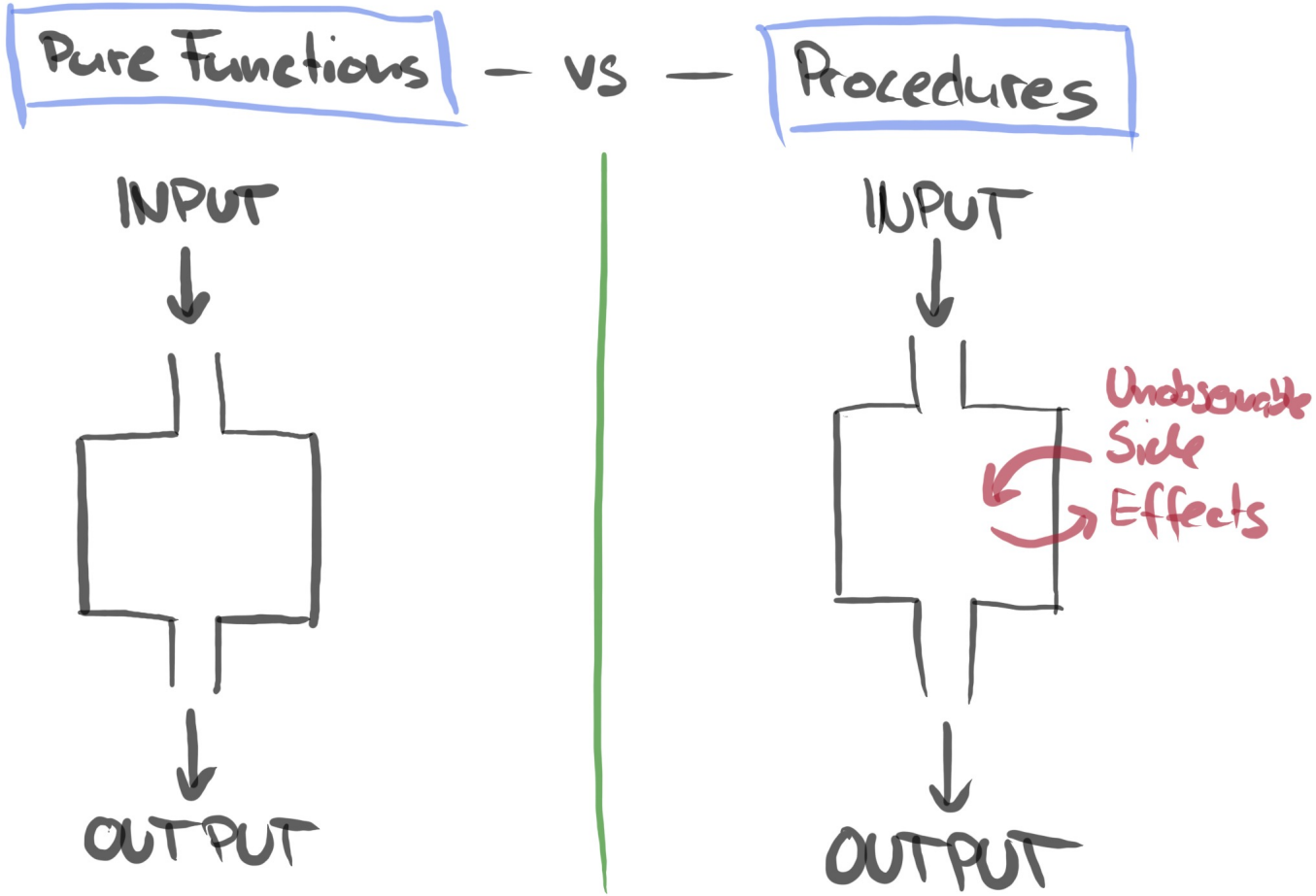
It is hard to **test** asynchronous code



Recipes to simplify asynchronous programming

Task: What can we do to simplify async programming?

Functional programming



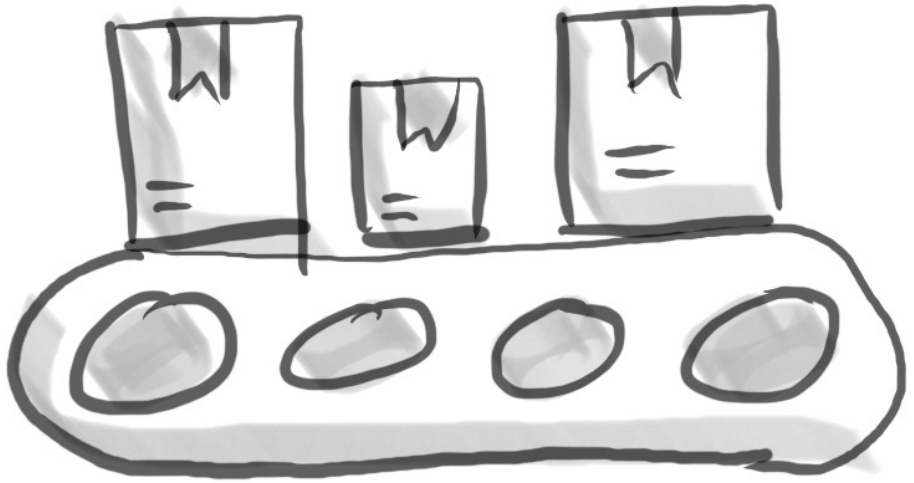
Procedure

```
private int oldDigit = 5;


public void AddNumber(int newValue)
{
    this.oldDigit += newValue;
}


this.AddNumber(10);
```

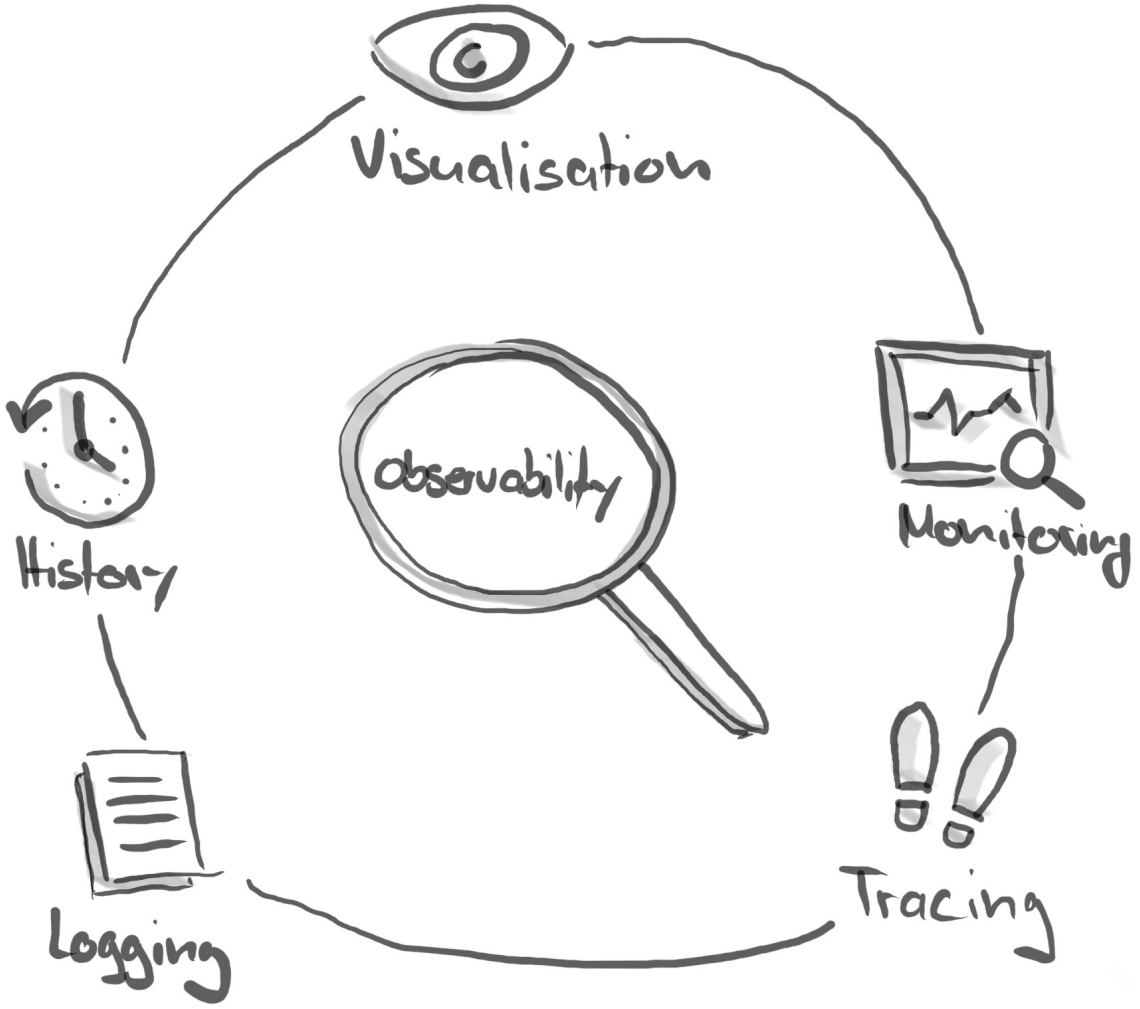
Pure function

```
private int oldDigit = 5;


public int AddNumber(int oldValue, int newValue)
{
    return oldValue + newValue;
}


this.oldDigit = AddNumber(this.oldDigit, 10);
```

Processing events in **Queues**



Ensure **observability**



The Reactive Extensions

How to program **without loops**?

ForEach

```
> [1, 2, 3].ForEach(x => Console.WriteLine(x));


> 1
> 2
> 3
```

Map / Select

```
> [1, 2, 3].Select(x => x + 1)

> [2, 3, 4]
```

Filter / Where

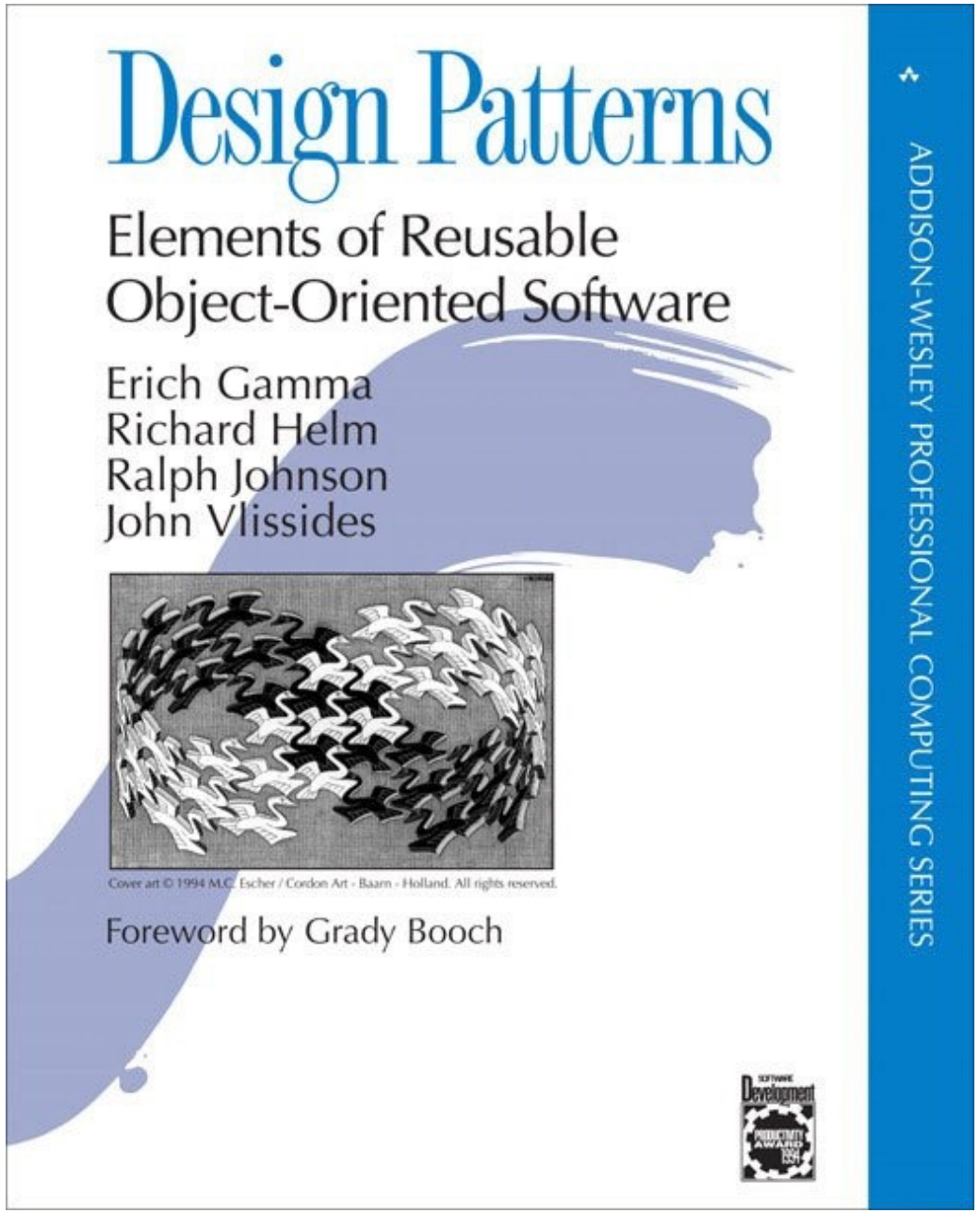
```
> [1, 2, 3].Where(x => x > 1)

> [2, 3]
```

ConcatAll / SelectMany

```
> [[1], [2, 3], [], [4]].SelectMany(x => x)

> [1, 2, 3, 4]
```

Events and Arrays are both

collections


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

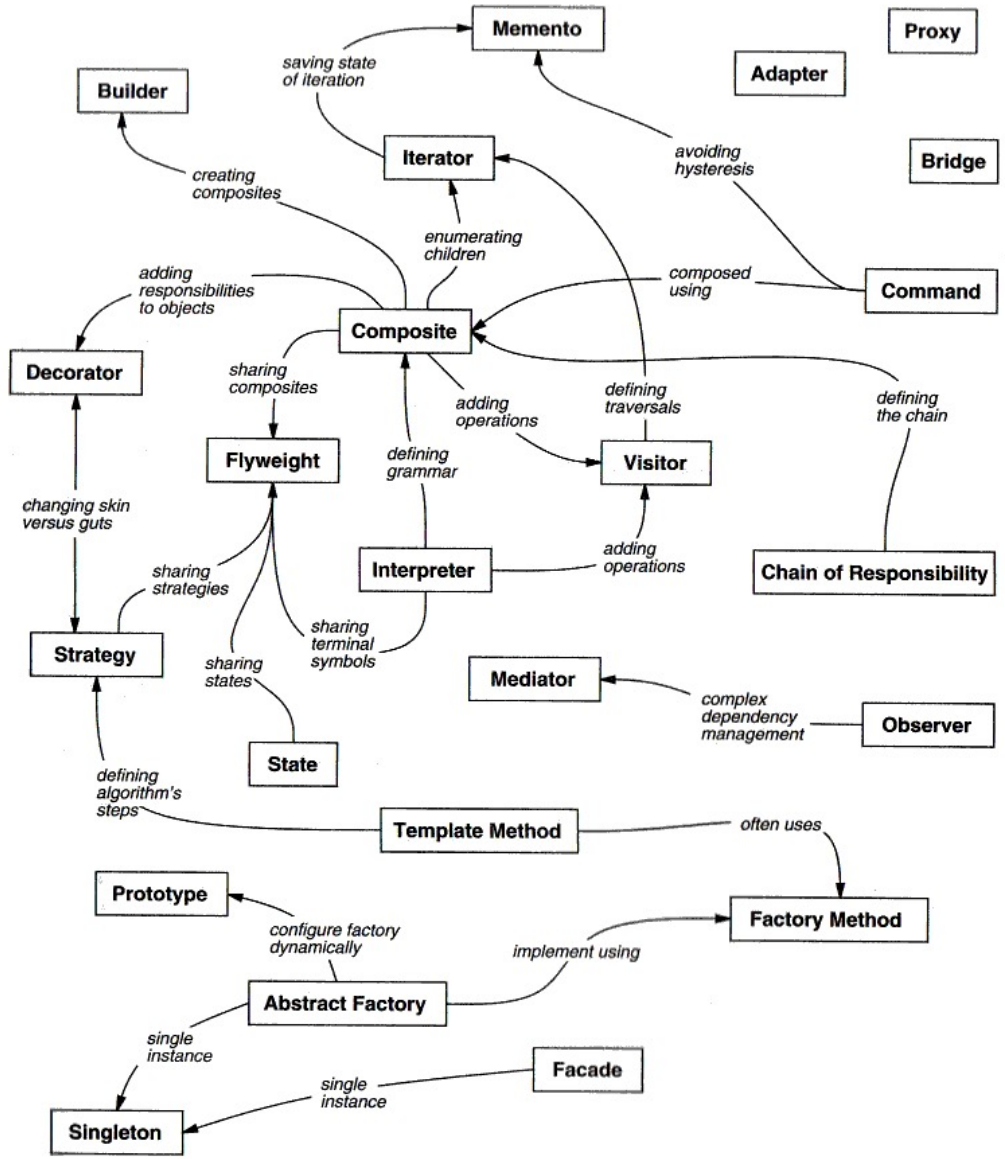


Figure 1.1: Design pattern relationships

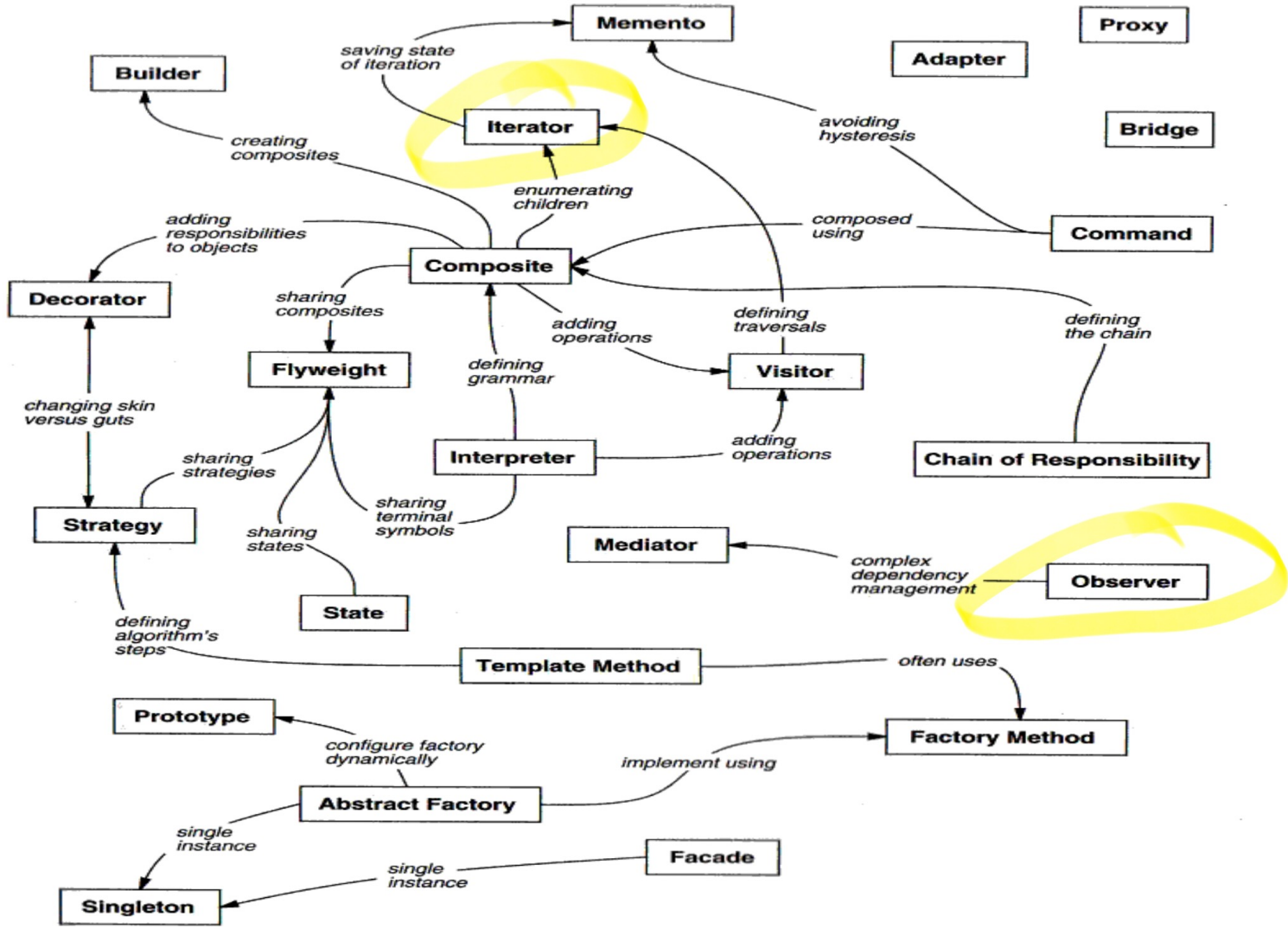


Figure 1.1: Design pattern relationships

Iterator / Enumerator

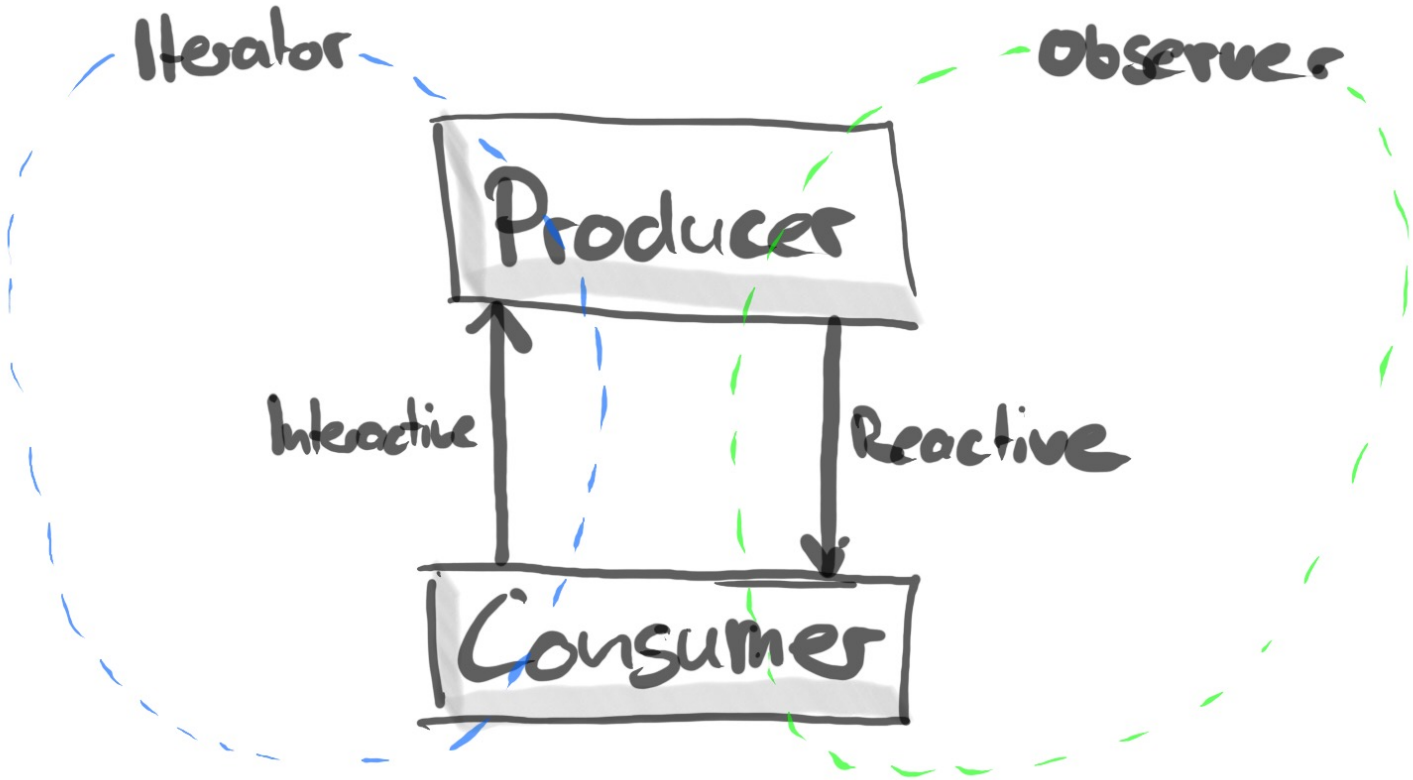
```
> var enumerator = [1, 2, 3].GetEnumerator()
> var hasMore = enumerator.MoveNext()
> Console.WriteLine(
   $”Value: {enumerator.Current}, HasMore: {hasMore}”)
> Value: 1, HasMore: true

    …
> var hasMore = enumerator.MoveNext()
> Console.WriteLine(
   $”Value: {enumerator.Current}, HasMore: {hasMore}”)
> Value: 3, HasMore: false
```

Select, **Where**, **Concat** can all be implemented using an iterator

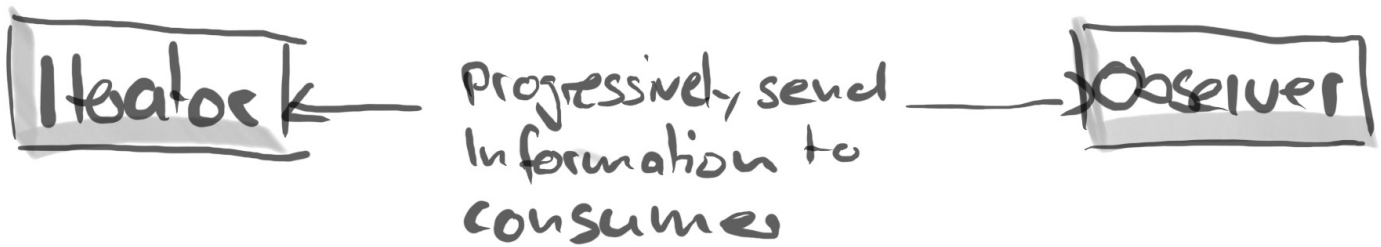
Observer Pattern

```
> window.MouseMoved += (sender, e) =>
{
    Console.WriteLine($”X: {e.X}, Y: {e.Y}”);
};

> X: 425, Y: 543
> X: 450, Y: 558
> X: 455, Y: 562
> X: 460, Y: 743
> X: 476, Y: 760
```

Interactive vs. Reactive



The Gang of Four made a failure



The Interator and Observer pattern are **symmetrical**

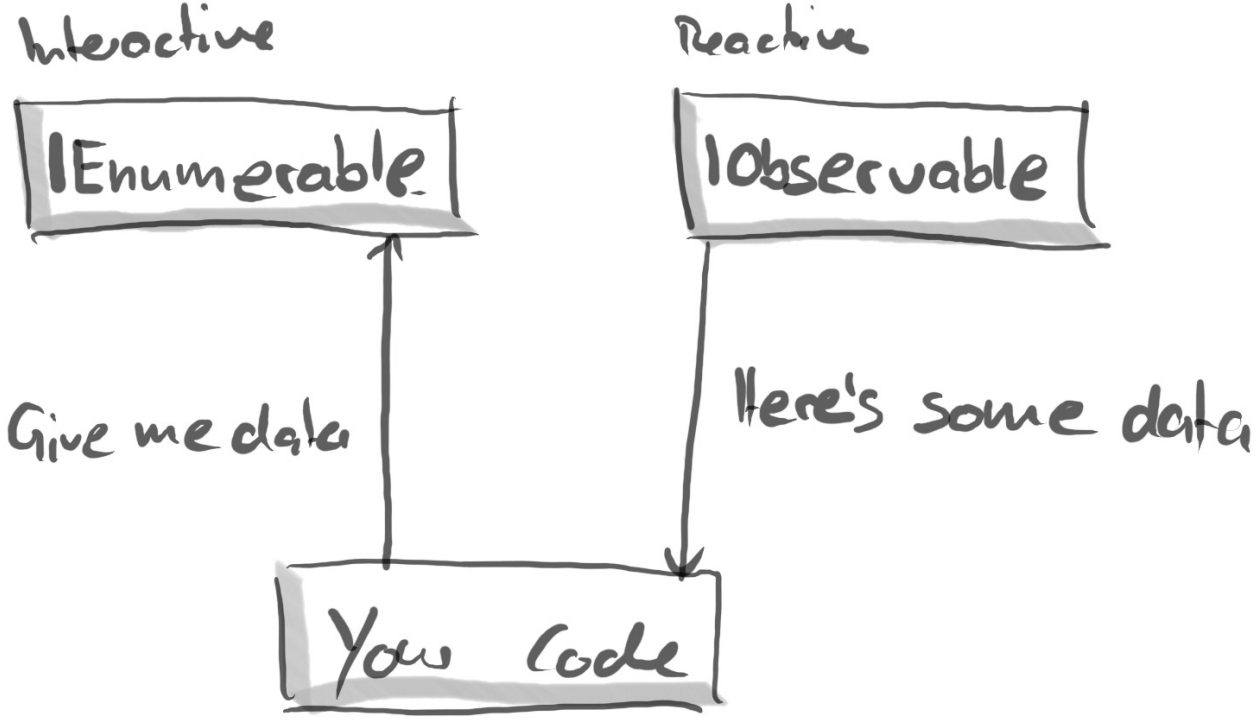
As a result, they gave the Iterator and Observer pattern **different semantics**…

…and this leads to so many **different push API's**

- DOM Events
- Server Request Responses
- WebSocket
- Service Workers
- Message Queue Handlers
- Streams
- IO Events
- …

Reactive Extensions (Rx)

And this is where the **Observable** comes into place…



IObservable<T>

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

IObserver<T>

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

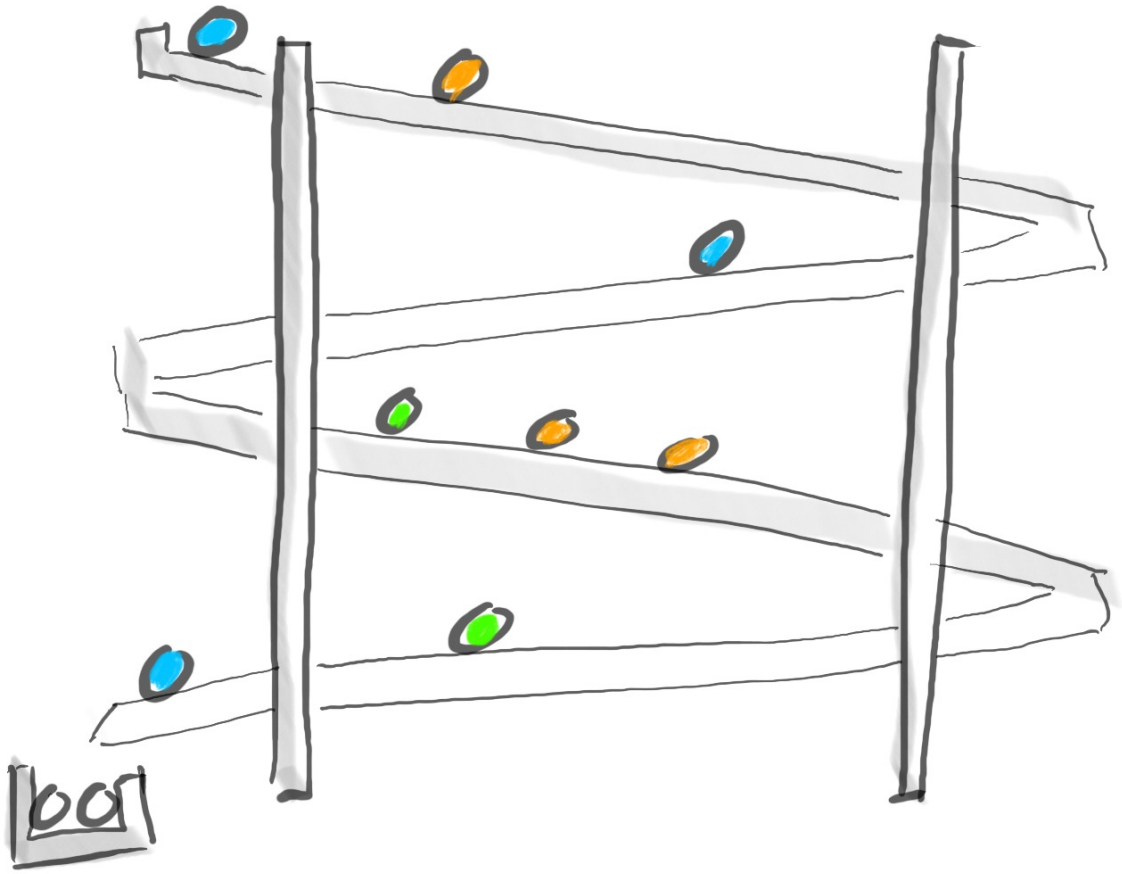
Examples

Type	Sequence	Example
IObservable<int>	OnCompleted	A timer
IObservable<SomeDTO>	OnNext, OnCompleted	API invocation
IObservable<SomeDTO>	OnError	Failed API invocation
IObservable<MouseMoveEvent>	OnNext, OnNext, OnNext	Continues UI events
IObservable<Customer>	OnNext, OnNext, OnNext, OnCompleted	Customers
IObservable<Customer>	OnError	Customers, immediate failure
IObservable<Customer>	OnNext, OnNext, OnError	Customers, later failure

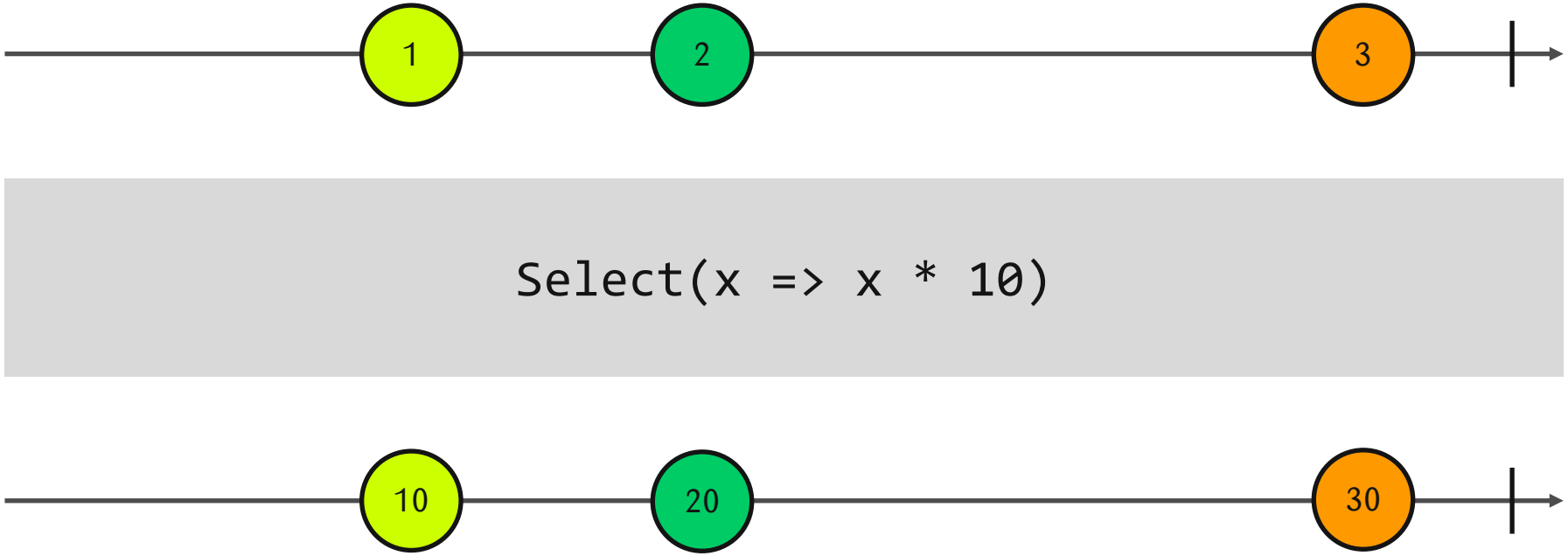
Observable = Collection + Time

The interfaces from iterators and observables **are the same** and observables are collections, why don't we **use the same** functional operators on observables?

The **Marble** syntax



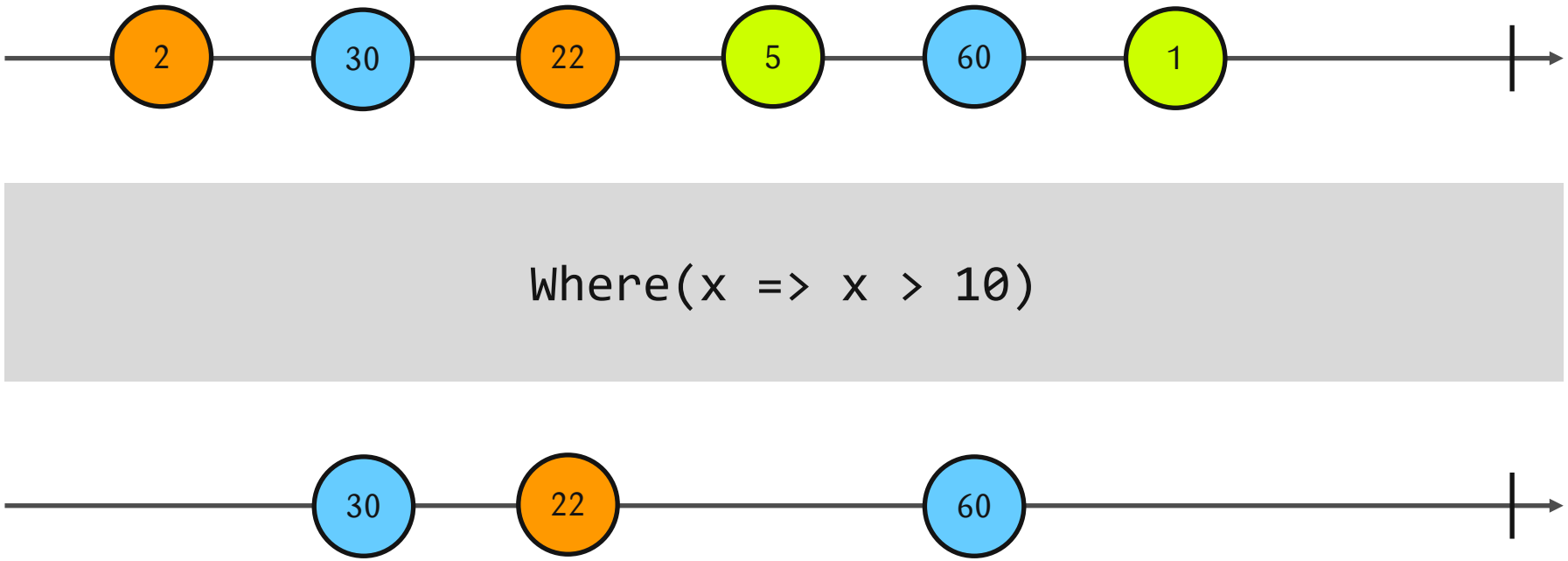
Select



Select

```
IObservable<SomeModel> models = …;

var viewModels = models.Select(model => new ViewModel(model));
```

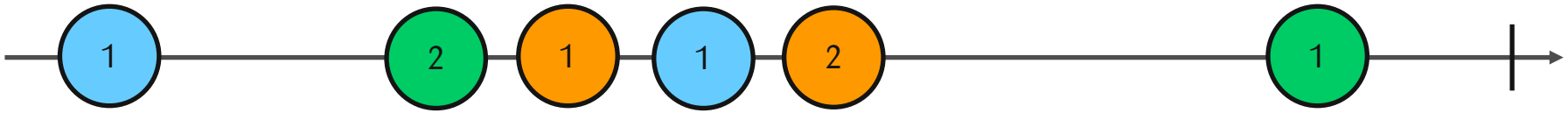
Where



Where

```
IObservable<SomeModel> models = …;

var filteredModels = models.Where(model =>!model.IsDone);
```

Throttle

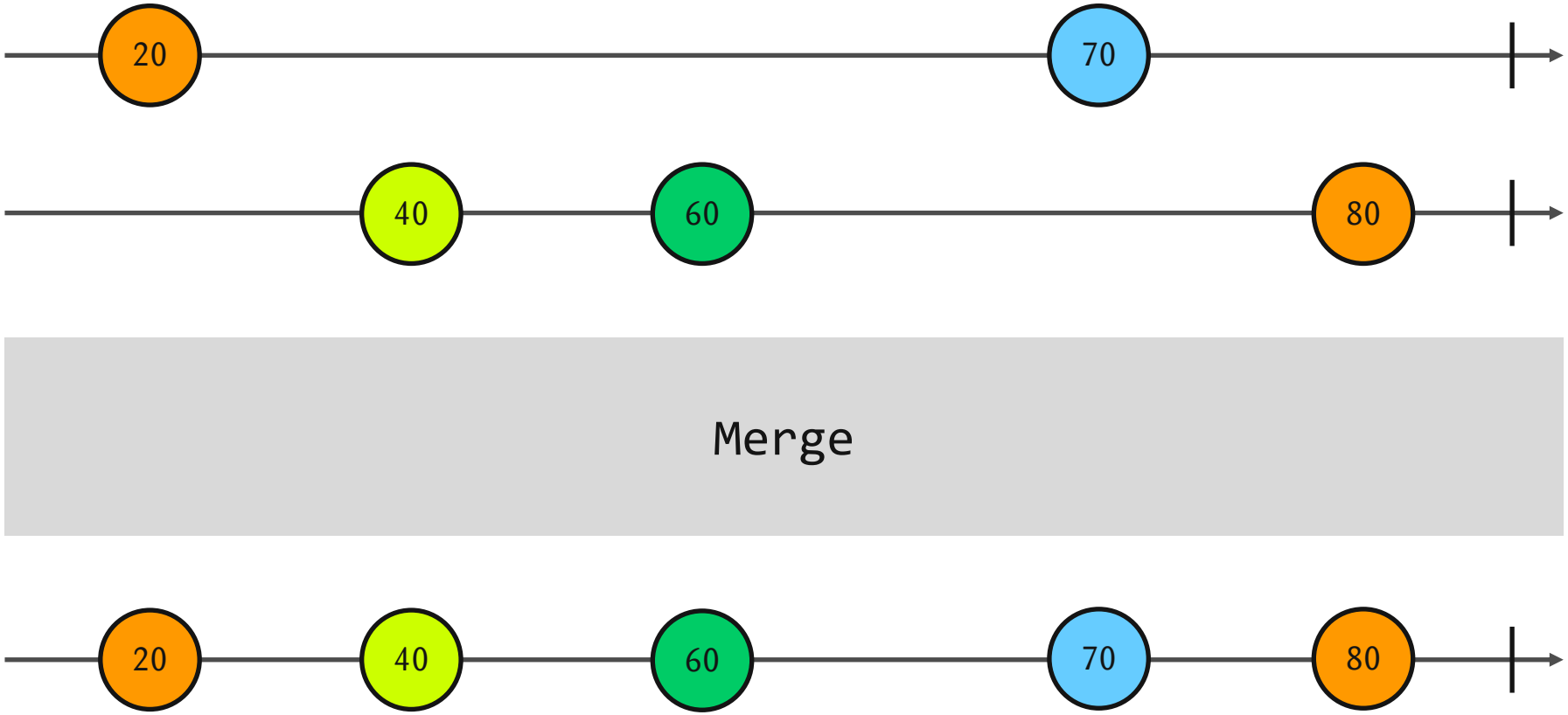



```
Throttle(TimeSpan.FromMilliseconds(400))
```

Throttle

```
IObservable<Action> actions = …;

var throttledActions =
      actions.Throttle(TimeSpan.FromMilliseconds(200));
```

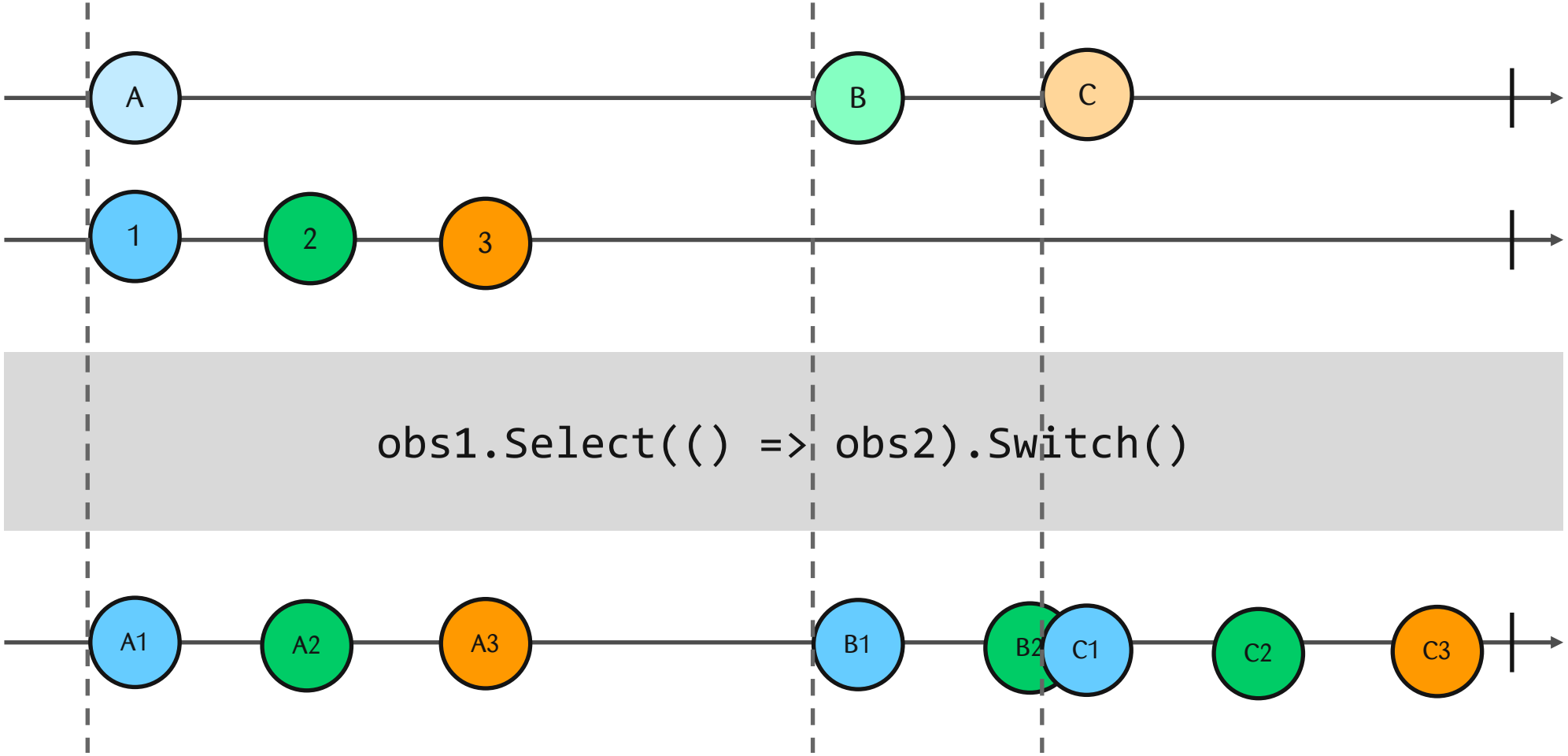
Merge



Merge

```
IObservable<SomeModel> models = …;
IObservable<SomeModel> moreModels = …;

var allModels = models.Merge(moreModels);
```

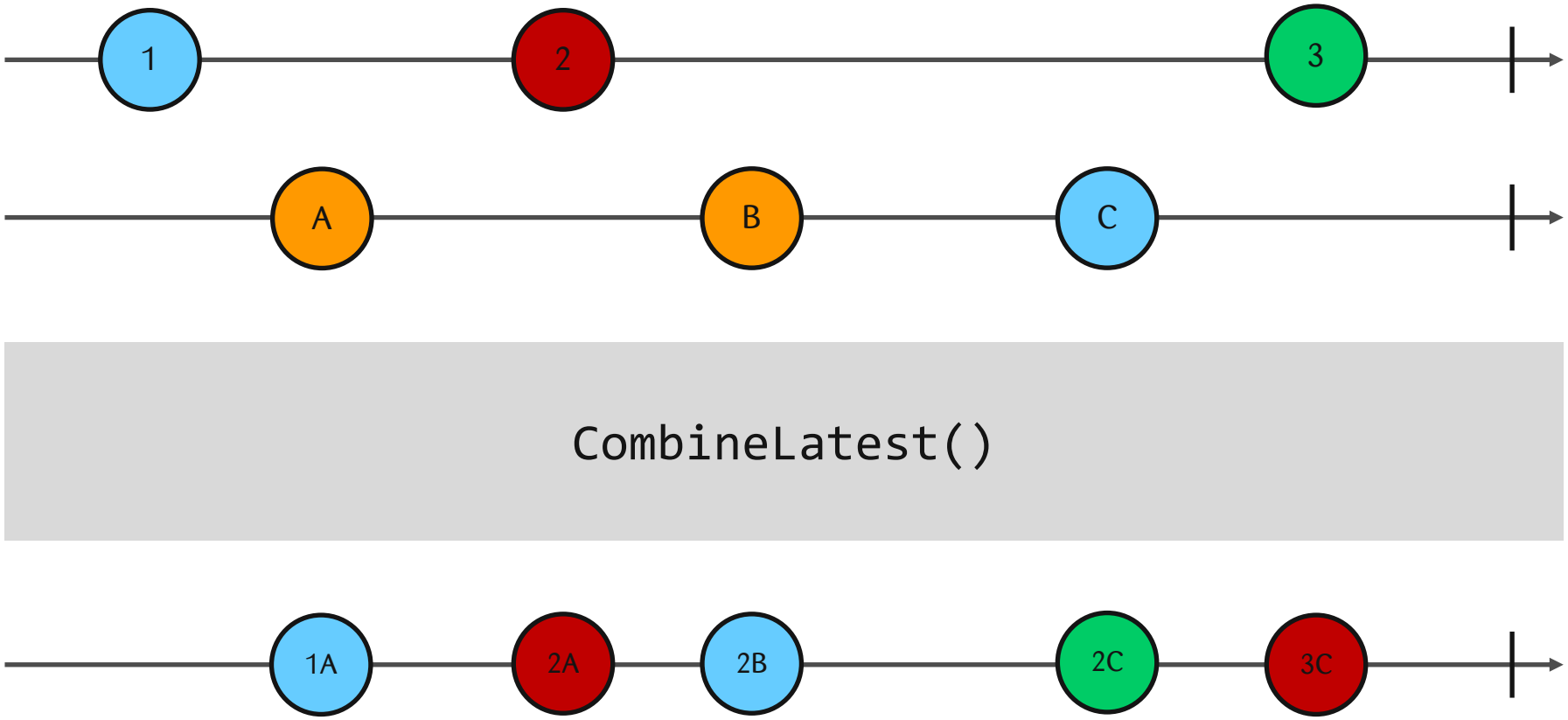
Switch



Switch

```
IObservable<Action> actions = …;
IObservable<SomeDtos> requestResult = …;

var results = actions.Select(() => requestResults).Switch();
```

CombineLatest



CombineLatest

```
IObservable<Action> actions = …;
IObservable<State> state = …;

var actionsCombinedWithState = actions.CombineLatest(state);
```

And many, many more…

- SelectMany
- Amb
- Scan
- Distinct
- Zip
- Delay
- First
- Take
- Count

Example

```
IApi api = …;
IObservable<string> input = …;

var searchResults = input
      .Throttle(TimeSpan.FromMilliseconds(400))
      .Select(i => api.Search(i))
      .Switch();
```

How can we **produce** an Observable?

Subjects

Type	Description
Subject<T>	Subscribers get notifications from the time on when they subscribe
BehaviorSubject<T>	Subscribers get the latest notification at the time when they subscribe
ReplaySubject<T>	Subscribers get all the notifications which have ever been in the subject
AsyncSubject<T>	Similar to Replay and Behavior subjects. It returns the last notification after the subject completed.

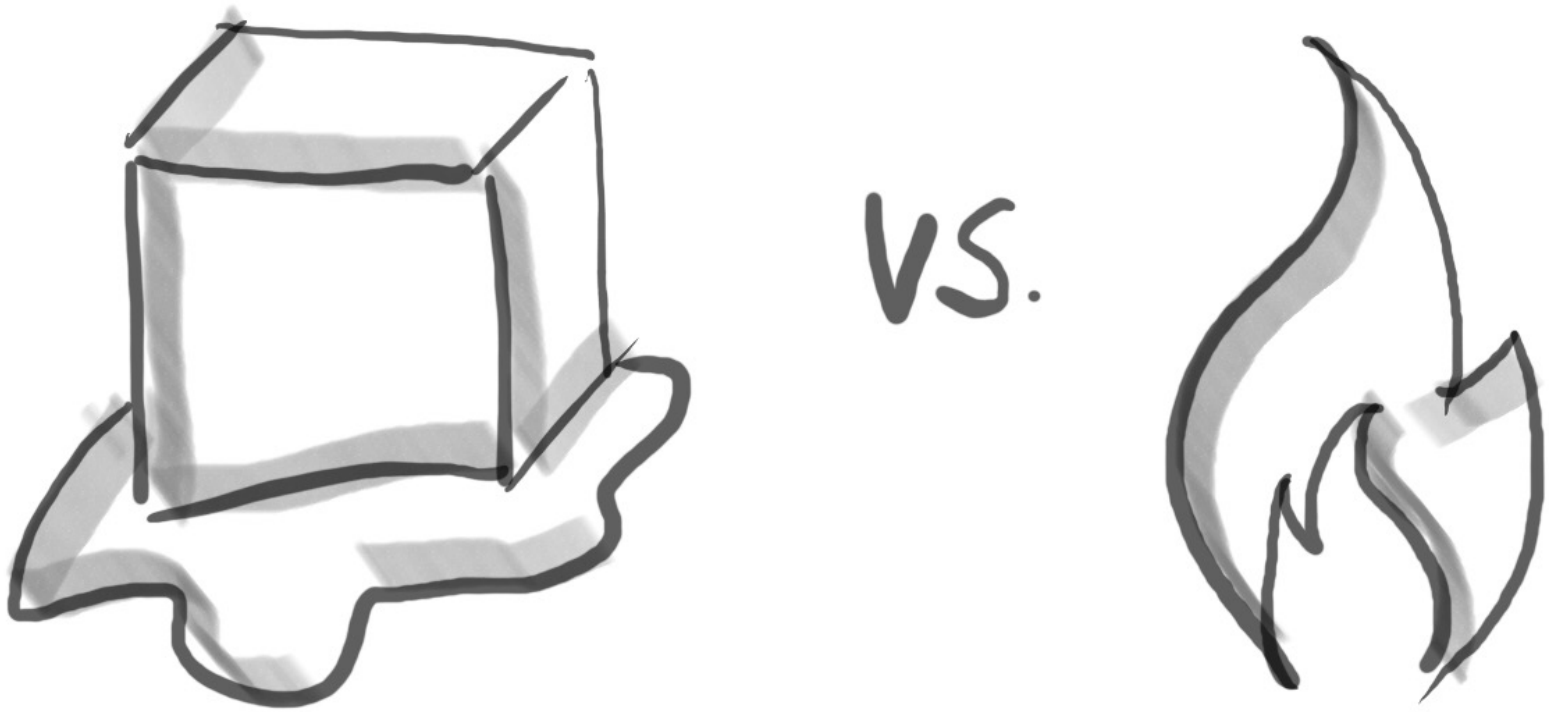
Example

```
var subject = new Subject<string>();

subject.OnNext("a");
subject.OnNext("a");

subject.OnComplete();


subject.OnError(new Exception(“An Error Happened”));
```

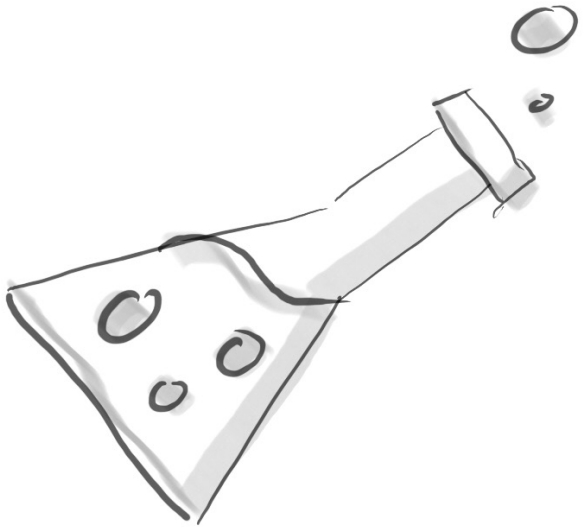
And many more ways…

Type	Description
Observable.Return	Creates a completed objservable with some notifications init.
Observable.Empty	Creates a competed observable without any notification.
Observable.Throw	Creates a completed observable with an error in it.
Observable.Never	Creates an infinite observable without any notification.
From Event	Creates an observable and dispatches events into it.
From Task	Converts a task into a completed single value observable.
From Enumerable	Converts an Enumerable into a completed observable.
Observable.Range	Creates a completed observable with a range of integer values.
Observable.Timer	Creates an observable and publishes a notification after a specified time and complets the observable afterwards.
Observable.Interval	Creates an observable and publeshes periodic notification on a configured interval.

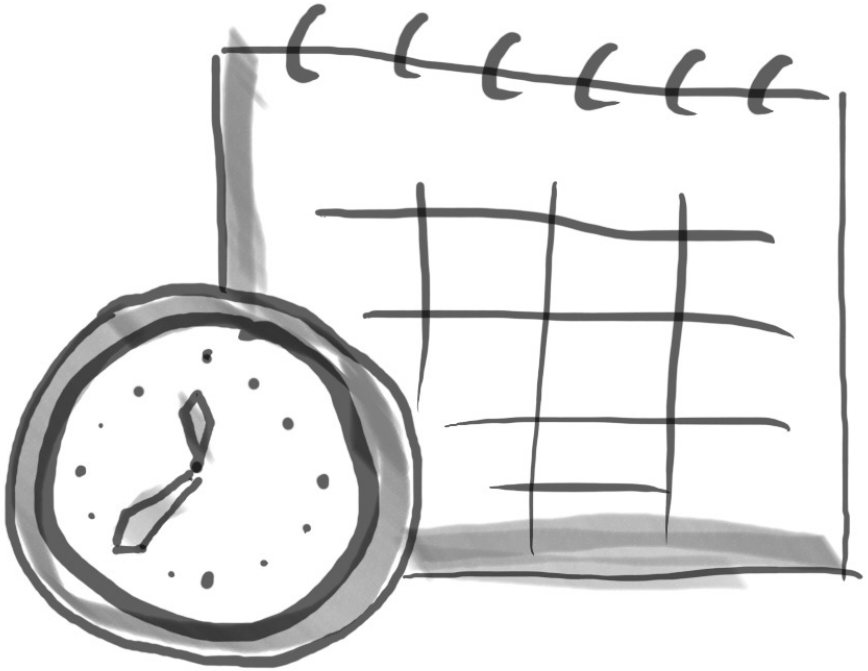
One thing you need to be aware of..



How to test Observables



Many **operators** are **time** based



TestScheduler to the rescue

```
var scheduler = new TestScheduler();

scheduler.Schedule(() => Console.WriteLine("A")); //Schedule immediately
scheduler.Schedule(TimeSpan.FromTicks(10), () => Console.WriteLine("B"));
scheduler.Schedule(TimeSpan.FromTicks(20), () => Console.WriteLine("C"));

Console.WriteLine("scheduler.AdvanceTo(1);");
scheduler.AdvanceTo(1);

Console.WriteLine("scheduler.AdvanceTo(10);");
scheduler.AdvanceTo(10);

Console.WriteLine("scheduler.AdvanceTo(15);");
scheduler.AdvanceTo(15);

Console.WriteLine("scheduler.AdvanceTo(20);");
scheduler.AdvanceTo(20);
```

```
scheduler.AdvanceTo(1);
A
scheduler.AdvanceTo(10);
B
scheduler.AdvanceTo(15);
scheduler.AdvanceTo(20);
C
```

Marble Testing

https://github.com/alexvictoor/MarbleTest.Net

Marble ASCII syntax

- Each ASCII character represents what happens during a time interval, by default 10 ticks.
  '-' means that nothing happens
  'a' or any letter means that an event occurs
  '|' means the stream end successfully
  '#' means an error occurs
- So "a-b-|" means:
  At 0, an event 'a' occurs
  Nothing till 20 where an event 'b' occurs
  Then the stream ends at 40
- If some events occurs simultaneously, you can group them using parenthesis.
  So "--(abc)--" means events a, b and c occur at time 20.

Marble test example

```
2 usages  Michael Lehmann
public IObservable<IAction> SearchAlbums => ActionStream // IObservable<IAction>
    .OfType<SearchAlbumsAction>()
    .Throttle(_schedulerProvider.CreateTime(TimeSpan.FromMilliseconds(400)), _schedulerProvider.Scheduler) // IObservable<SearchAlbumsAction>
    .Select(a :SearchAlbumsAction => _albumService.SearchAsync(a.Term)
        .SelectMany(searchResults :IEnumerable<Album> => new IAction[]
        {
            new SetBusyAction(false),
            new SetSearchResultsAction(searchResults.ToArray())
        })
        .StartWith(new SetBusyAction(true))
    ) // IObservable<IObservable<...>>
    .Switch(); // IObservable<IAction>
```

Marble test example

Mock the TimeSpan

```
_mocker.GetMock<ISchedulerProvider>() // Mock<ISchedulerProvider>
    .Setup( expression: s :ISchedulerProvider => s.CreateTime(It.IsAny<TimeSpan>())) // ISetup<ISchedulerProvider,...>
    .Returns( valueFunction: () => _scheduler.CreateTime( marbles: "---|"));
```

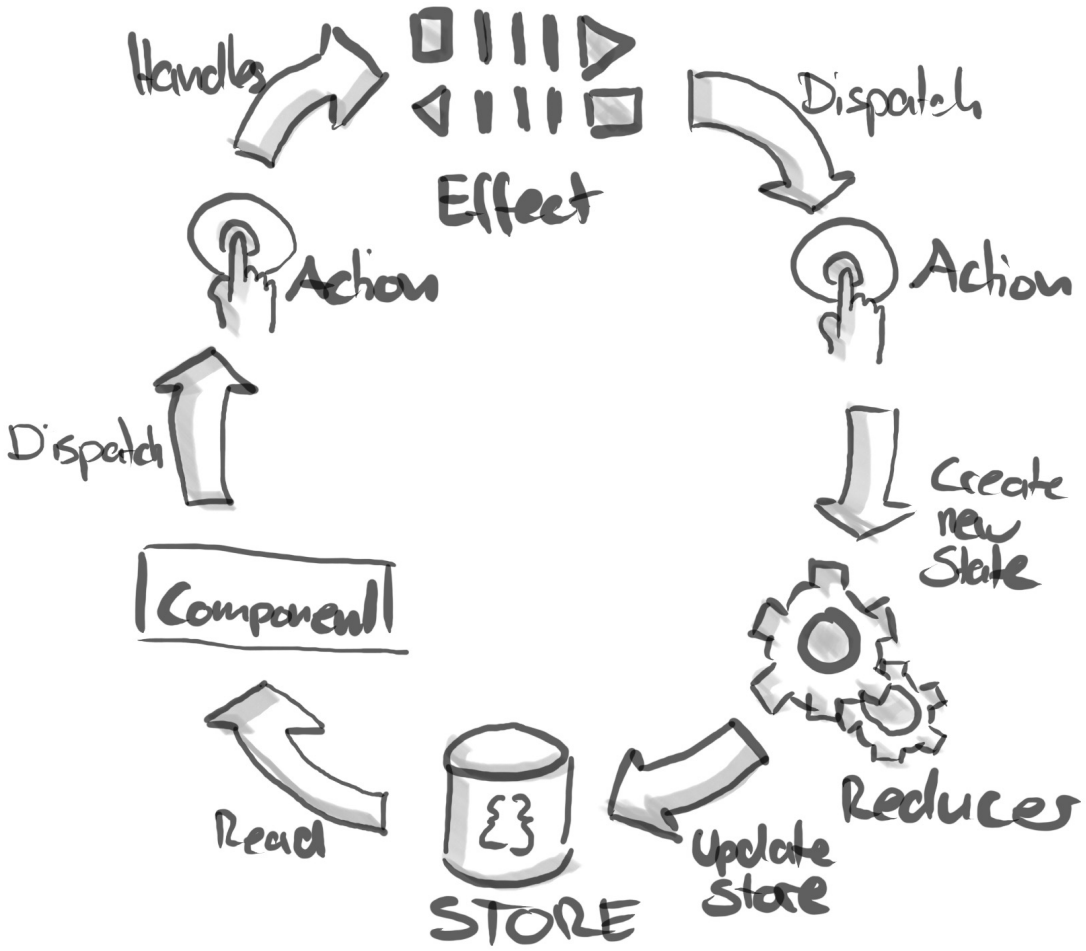
Mock the Album Service

```
_mocker.GetMock<IAlbumService>() // Mock<IAlbumService>
    .Setup( expression: s :IAlbumService => s.SearchAsync( searchTerm: It.IsAny<string>())) // ISetup<IAlbumService,IObservable<...>>
    .Returns( valueFunction: ()=>
    {
        return Observable.Return<IEnumerable<Album>>(new[] { new Album( artist: "Muse", title: "Absolution", coverUrl: "http://localhost/absolution") });
    });
```

Marble test example

```
var effect = _mocker.CreateInstance<ApplicationEffects>();

// when
var sourceEvents :ITestableObservable<StateActionPair<...>>? = _scheduler.CreateColdObservable<StateActionPair<ApplicationState>>(
    marbles: "--a-",
    values: new
    {
        a = new StateActionPair<ApplicationState>(state, action),
    });
effect.Connect(sourceEvents);

// then
_scheduler.ExpectObservable(effect.SearchAlbums) // ISetupTest
    .ToBe(
        marble: "-----(abc)-", // the actions are expected 3 time blocks later (see throtteling with the test scheduler)
        values: new
        {
            a = new SetBusyAction(true),
            b = new SetBusyAction(false),
            c = new SetSearchResultsAction(new[] { new Album( artist: "Muse",  title: "Absolution",  coverUrl: "http://localhost/absolution") }),
        });
_scheduler.Flush();
```

Why is **RX** a great match for **Redux**?



Redux implementation is a **One-Liner**

```
var actions = new Subject<State>();

var store = actions.Scan((state, action) => reduce(state, action), initState)

// subscribe to state (changes)
store.Subscribe(state => …);

// dispatch action
actions.OnNext(new SomeAction());
```

Effects provide an ease frame for the usage of **RX**

```
public IObservable<IAction> SearchAlbums => ActionStream
    .OfType<SearchAlbumsAction>()
    .Throttle(TimeSpan.FromMilliseconds(400)
    .Select(a => _albumService.SearchAsync(a.Term)
        .SelectMany(searchResults => new IAction[]
        {
            new SetBusyAction(false),
            new SetSearchResultsAction(searchResults.ToArray())
        })
        .StartWith(new SetBusyAction(true))
    )
    .Switch();
```

Keep in mind

- We are talking about a **push interface**, the values will arrive **over time**
- An **Observable can get closed**. **Certain operators** (e.g. Take, First, TakeUntil) and an **error** will close the Observable.
- An Observable can be **Hot** or **Cold**. Be aware what you have in your hand.

Learning Source

http://introtorx.com/Content/v1.0.10621.0/01_WhyRx.html#WhyRx

Examples

Converting http call to Observable

```
public IObservable<IEnumerable<string>> Search(string term)
{
    using var client = new HttpClient();

    // making a hot observable from the web request
    var response =  client
        .GetAsync($"http://webcode.me?search={term}")
        .ToObservable();


    // converting it to a cold oberservable
    // (request only on subscribe)
    return Observable.Defer(() => response);
}
```

Throttle input and cancle previous request

```
IApi api = …;
Subject<string> input = new Subject<string>;

var searchResults = input
      .Throttle(TimeSpan.FromMilliseconds(400))
      .Select(i => api.Search(i))
      .Switch();


input.OnNext(“t”);
input.OnNext(“te”);
input.OnNext(“tes”);
input.OnNext(“test”);
```

Poll api endpoint

```
IApi api = …;

var searchResults = Observable
    .Interval(TimeSpan.FromMilliseconds(250))
    .Select(i => api.Search(i))
    .Switch()
```

Cache a hot observable

```
IApi api = …;

IObservable<string> cachedSearchResult = api.Search(“term”)
    .Publish() // redirects the values into a new Subject
    .RefCount(); // connects the new subject on the first Subscribe
```

Cleanup muliple subsccriptions

```
Subject<Unit> disposer = new Subject<Unit>();

IObservable myObservable1 = …;
IObservable myObservable2 = …;


myObservable1
    .TakeUntil(disposer) // this closes the observable
    .Subscribe(…);

myObservable2
    .TakeUntil(disposer) // this closes the observable
    .Subscribe(…);
```

Exercise

Workshop Sources

https://github.com/lehmamic/AsyncProgrammingWorkshop

Exercice: Refactor the code to load the articles via RX.Net

Acceptance Criteria:

1. The input gets throttled (400ms)
2. Ongoing Requests get canceled when a new input triggers a new request
3. Use a Subject to produce “search term” events
4. The marble test is green
5. Compare the code: imperative implementation vs. reactive implementation

Exercice: Introduce a polling to automatically update the search results

Acceptance Criteria:

1. Update the result list automatically when a new matching article has been added
2. The marble tests is green again (question: we have an infinite timer in place, what we need to do?)
3. Compare the code: imperative implementation vs. reactive implementation

Possible extensions:

- Introduce a new “swaerword api” and filter out articles containing searwords
- Also update the result list automatically when the swearword database gets updated
- Display a second result list (with a second subscription) and avoid that the articles get loaded twice

Questions