ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Basic Combinators on Observable Collections

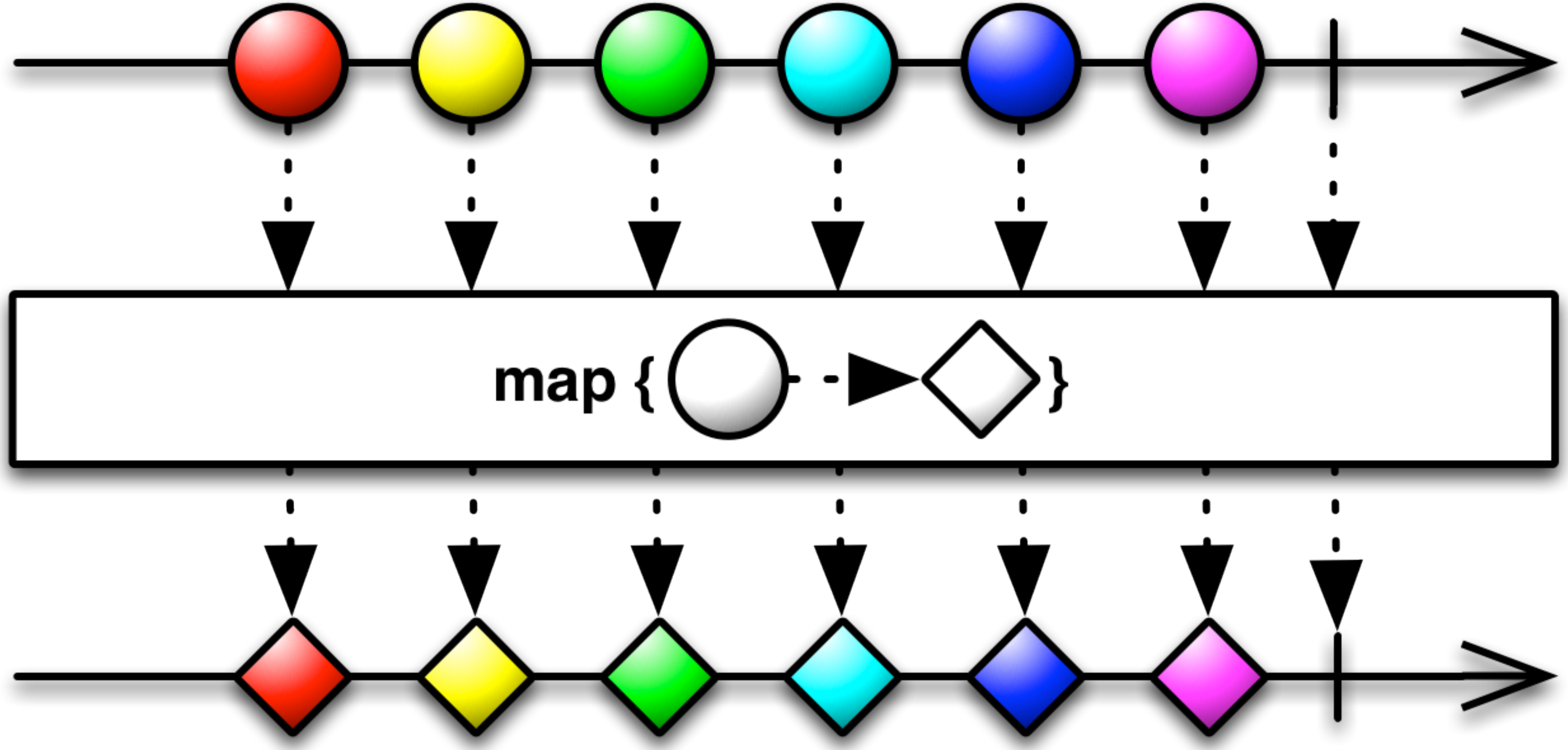
Principles of Reactive Programming

Erik Meijer

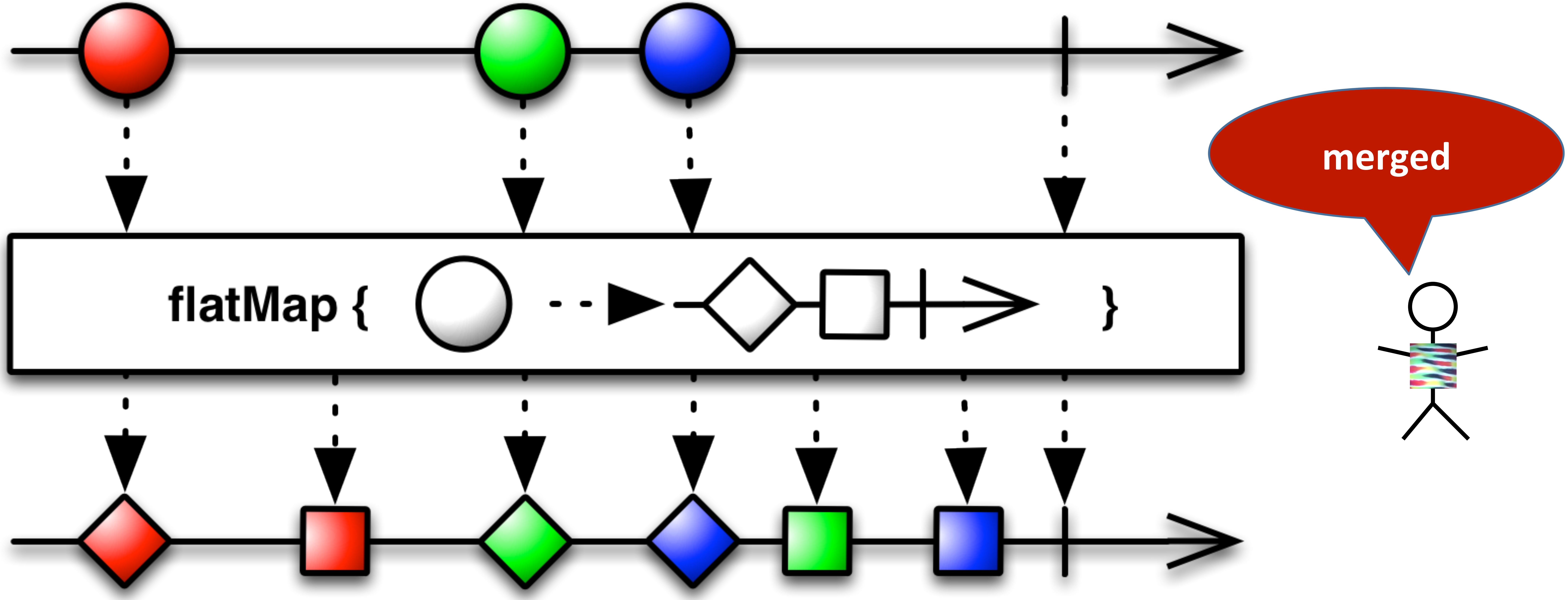
Higher-order Function to manipulate Observable[T]

```
def flatMap[B](f: A⇒Observable[B]): Observable[B]

def map[B](f: A⇒B): Observable[B]

def filter(p: A⇒Boolean): Observable[A]

def take(n: Int): Observable[A]

def takeWhile(p: A⇒Boolean): Observable[A]

def toList(): List[A]

def zip[B](that: Observable[B]): Observable[(A, B)]
```

Marble Diagrams



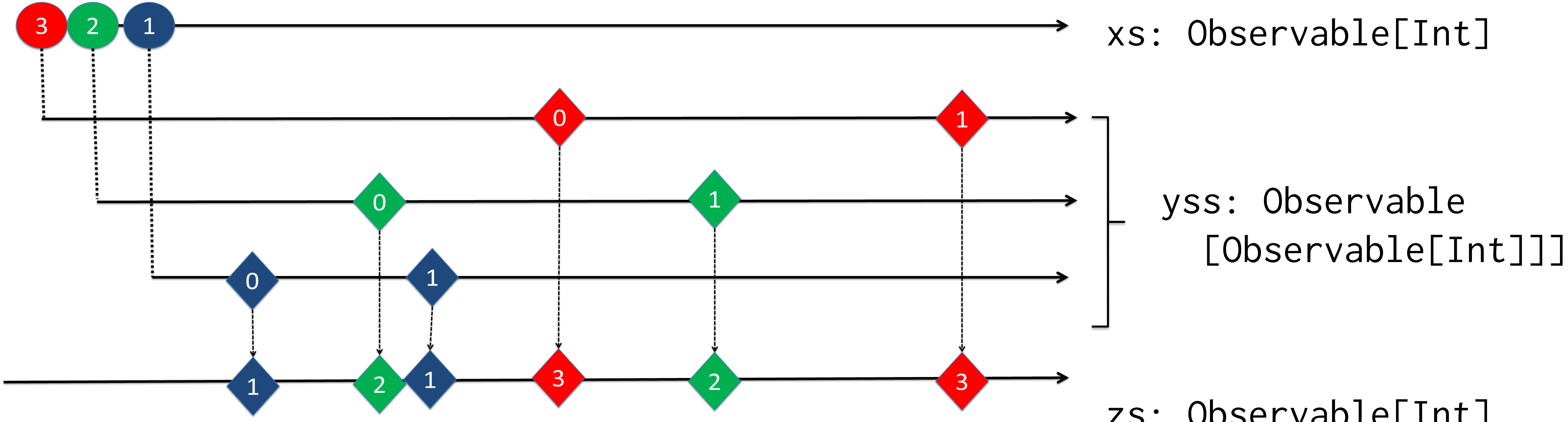
Marble Diagrams



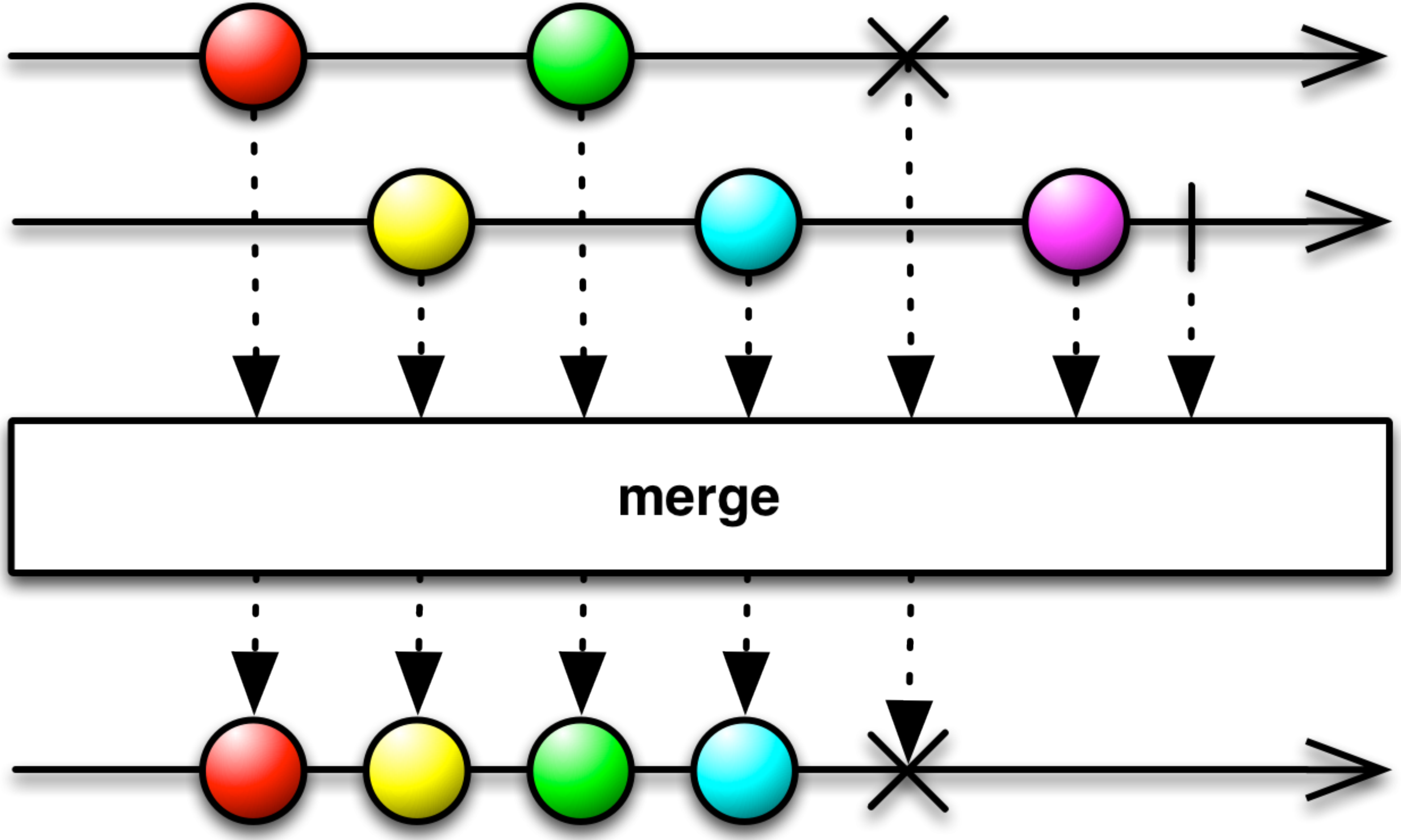
```
def flatMap(f: T=>Observable[S]): Observable[S] = { map(f).flatten() }
```

Flatten nested streams

```
val xs: Observable[Int] = Observable(3,2,1)
val yss: Observable[Observable[Int]] =
   xs.map(x => Observable.Interval(x seconds).map(_=>x).take(2))
val zs: Observable[Int] = yss.flatten()
```

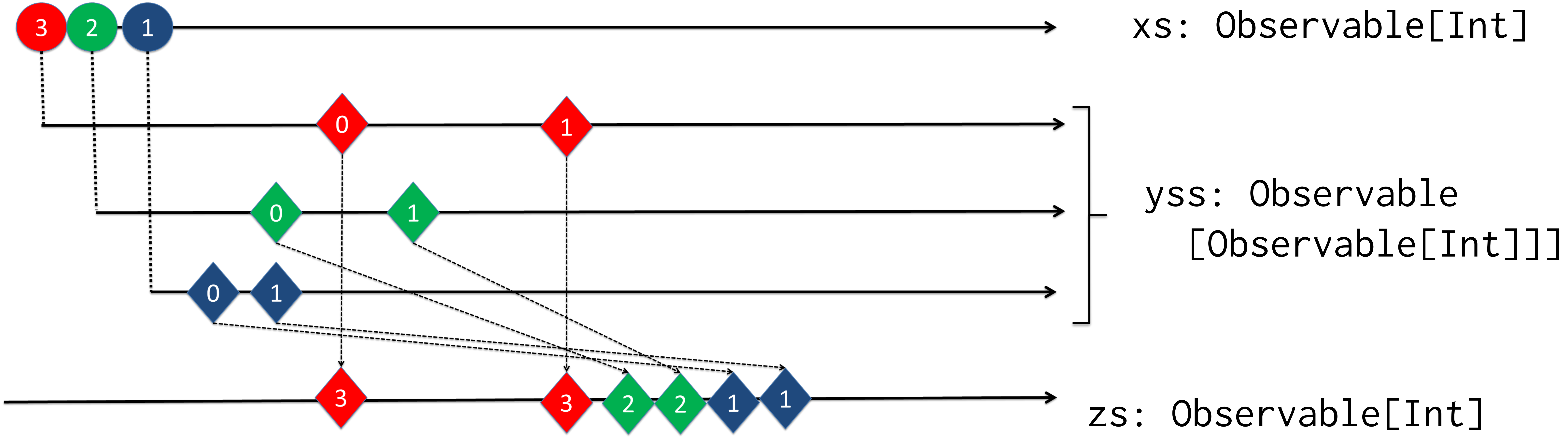


Marble Diagrams

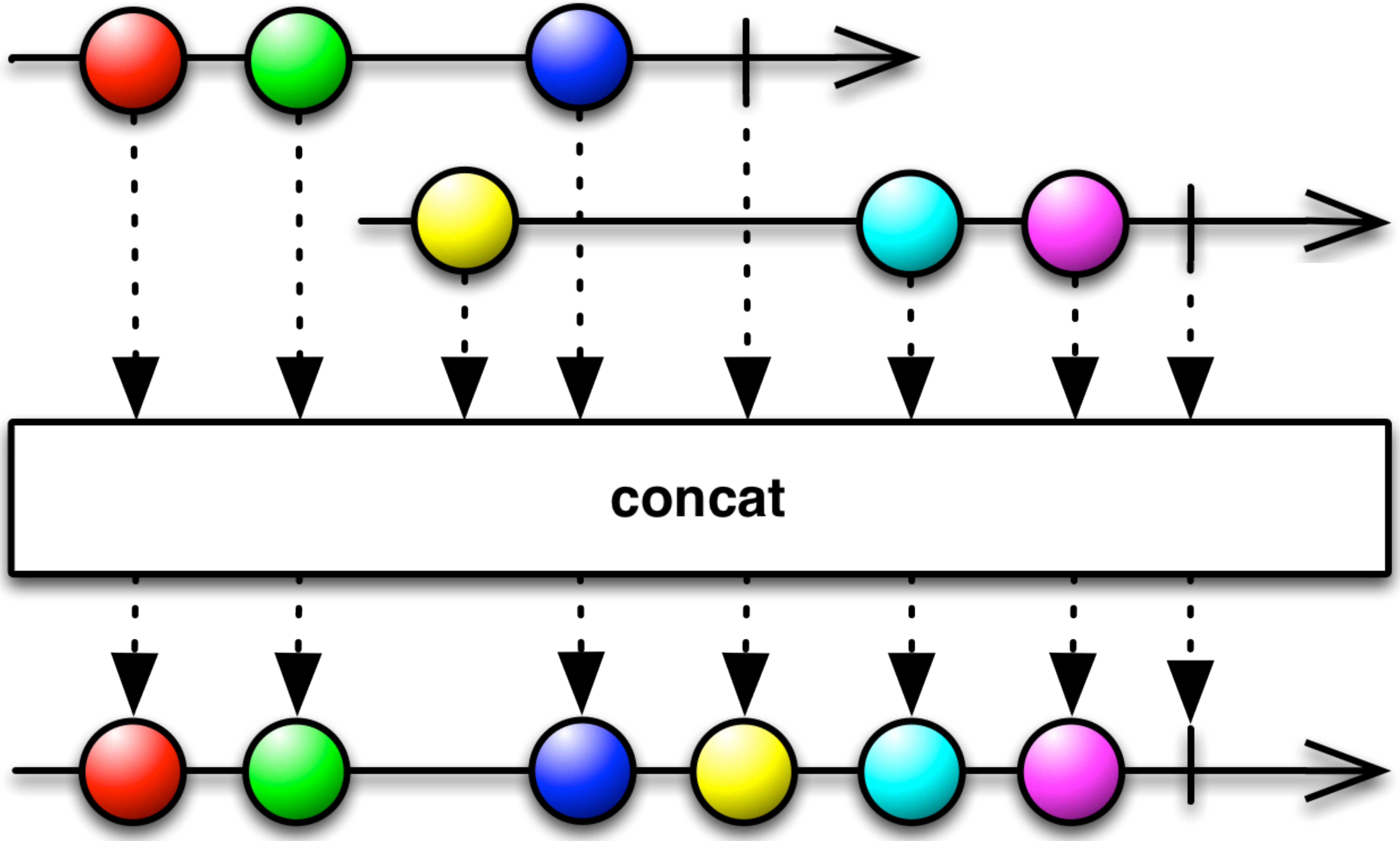


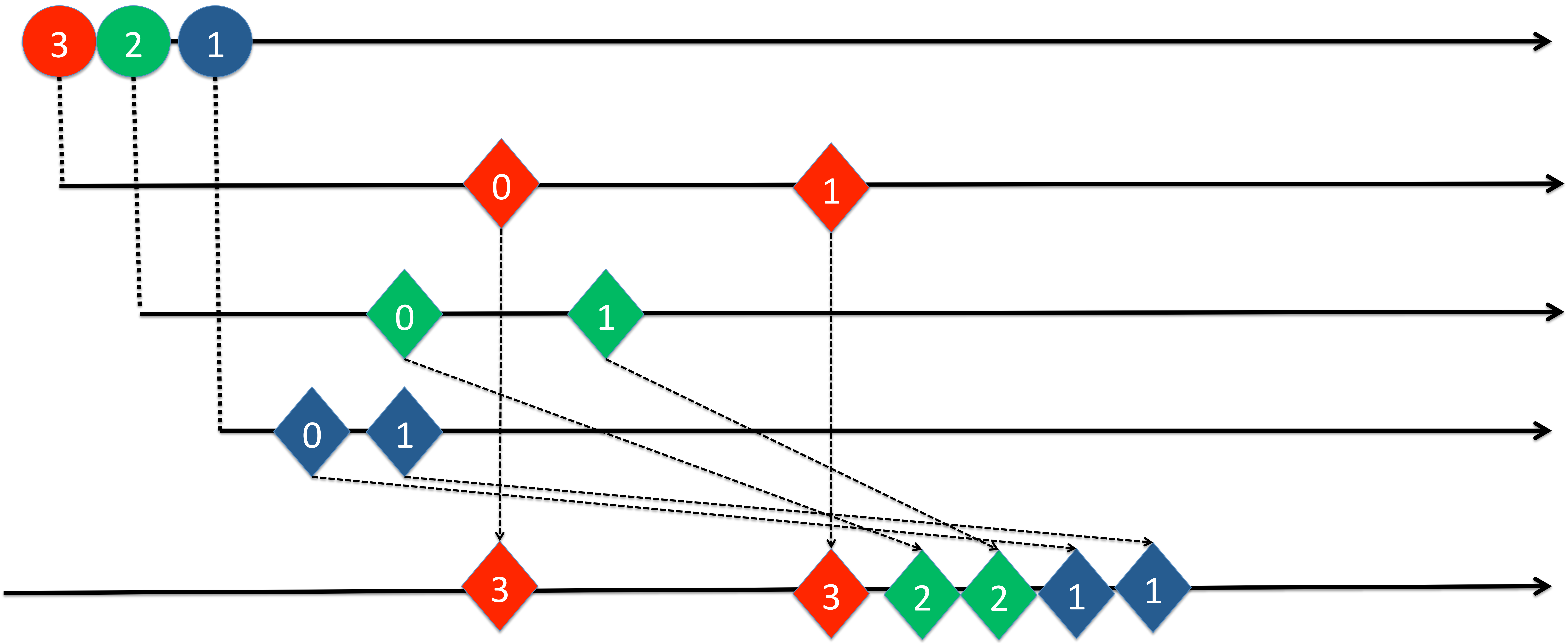
Flatten nested streams

```
val xs: Observable[Int] = Observable(3,2,1)
val yss: Observable[Observable[Int]] =
   xs.map(x => Observable.Interval(x seconds).map(_=>x).take(2))
val zs: Observable[Int] = yss.concat()
```



Marble Diagrams




3
2
1
0
1
0
1
0
1
3
3
2
2
1
1

Earthquakes

```
def usgs(): Observable[EarthQuake] = {…}

class EarthQuake {
   …
   def magnitude: Double
   def location: GeoCoordinate
}

object Magnitude extends Enumeration {
  def apply(magnitude: Double): Magnitude = { … }
  type Magnitude = Value
  val Micro, Minor, Light, Moderate, Strong, Major, Great = Value
}
```

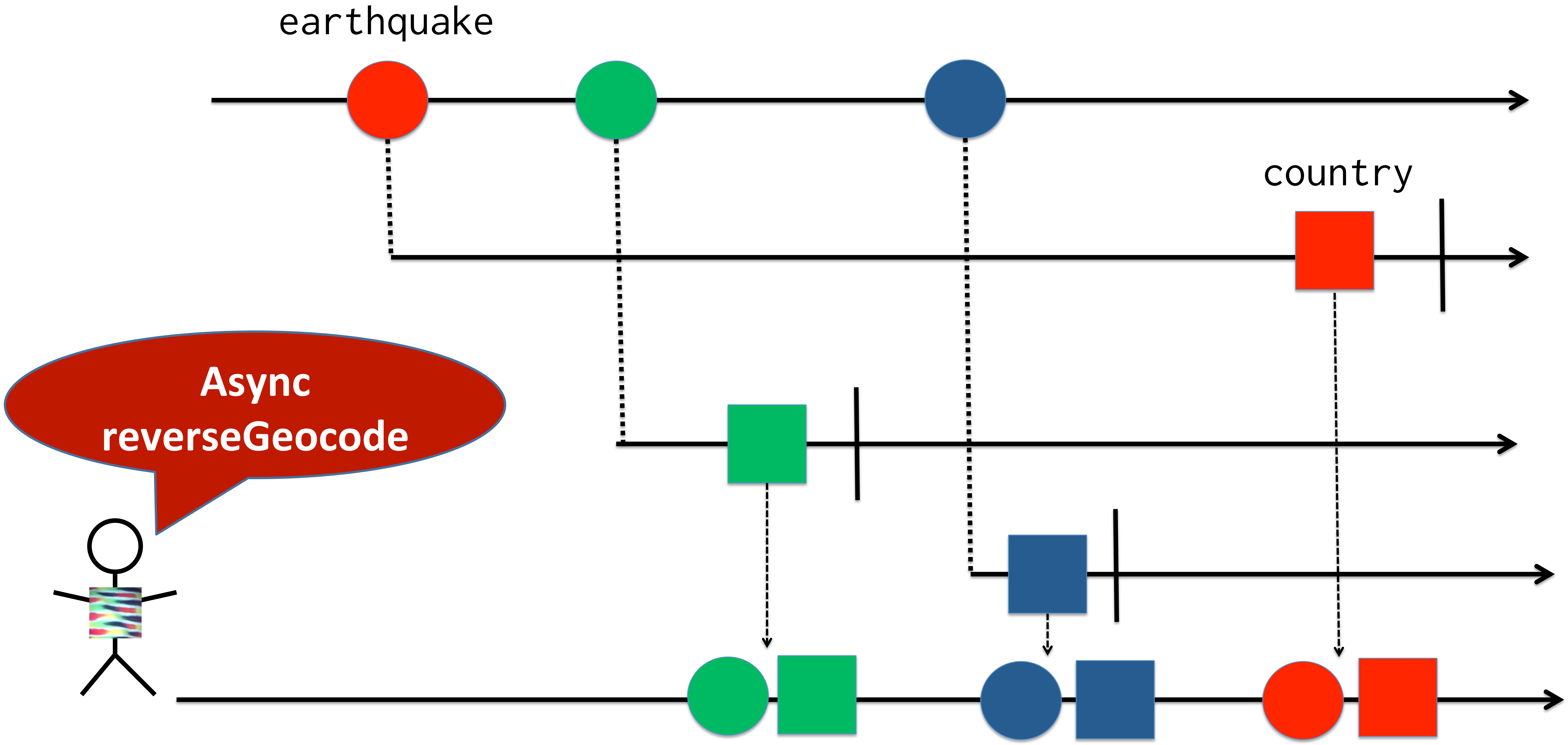
Mapping and filtering asynchronous streams

```
val quakes = usgs()

val major = quakes.
   map(q⇒(q.Location, Magnitude(q.Magnitude))).
   filter{ case (loc,mag) ⇒ mag >= Major }

major.subscribe({ case (loc, mag) ⇒ {
   println($“Magnitude ${ mag } quake at ${ loc }”)
})
```

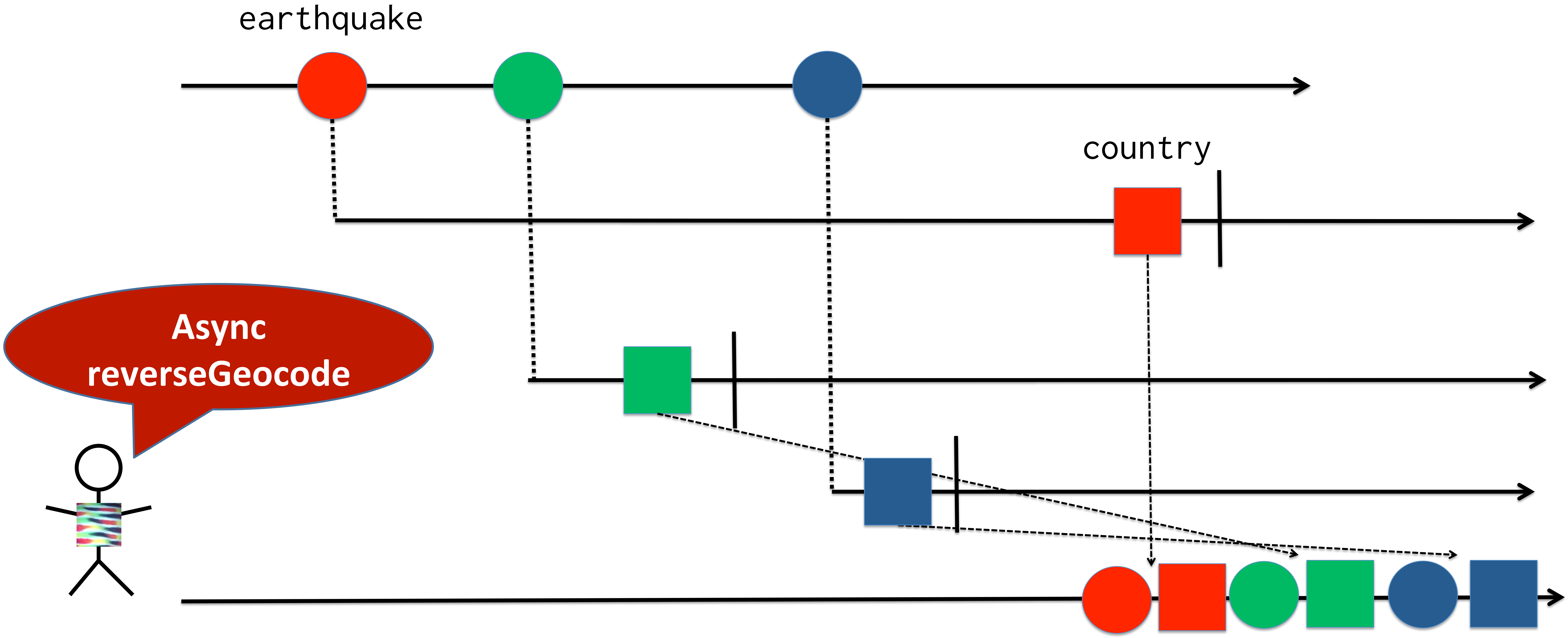
Reverse GeoCode

```
def reverseGeocode(c: GeoCoordinate): Future[Country] = { … }

val withCountry: Observable[Observable[(EarthQuake, Country)]] =
   usgs().map(quake ⇒ {
     val country: Future[Country] = reverseGeocode(q.Location)
     Observable(country.map(country⇒(quake,country)))
})

val merged: Observable[(EarthQuake, Country)] = withCountry.flatten()

val merged: Observable[(EarthQuake, Country)] = withCountry.concat()
```

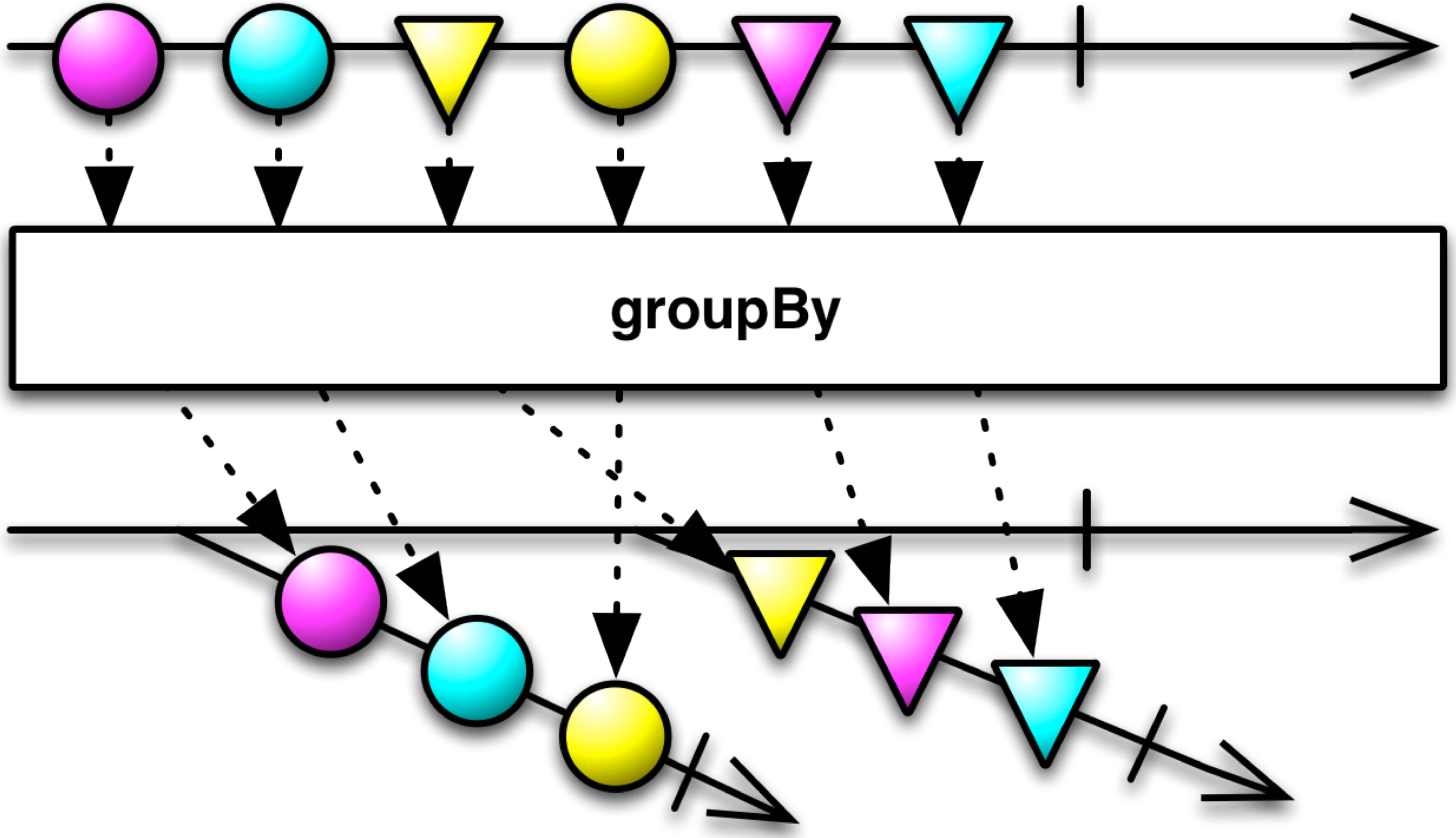
Reverse Geocode merge



Reverse Geocode concat



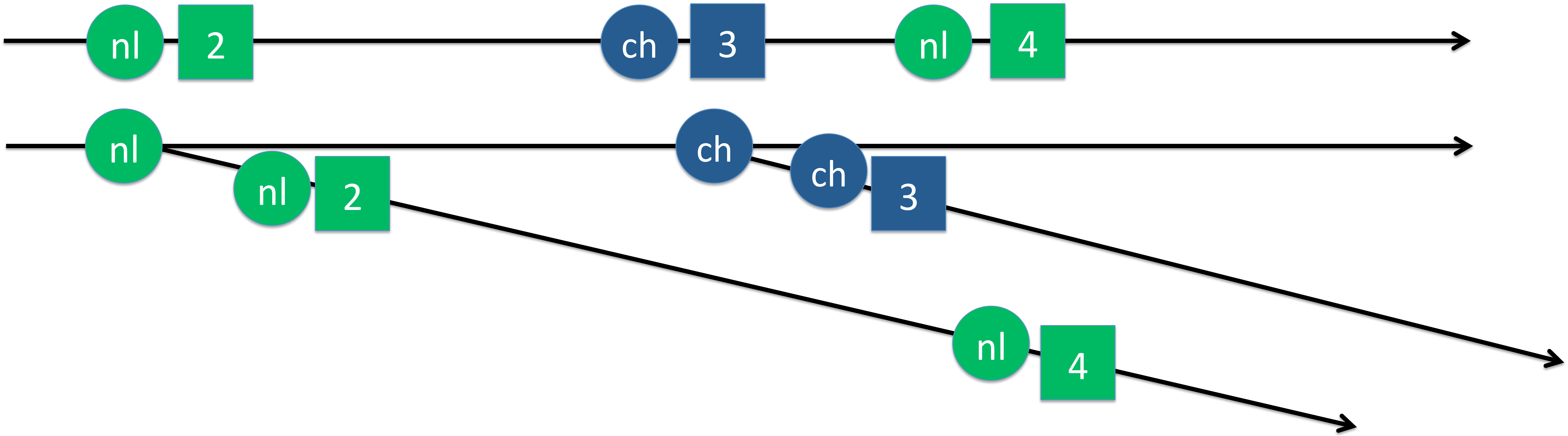
Groupby



```
def groupBy[K](keySelector: T⇒K): Observable[(K,Observable[T])]
```

Grouping by region

```
val merged: Observable[(EarthQuake, Country)] = withCountry.flatten()

val byCountry: Observable[(Country, Observable[(EarthQuake, Country)]] =
   merged.groupBy{ case (q,c) ⇒ c }
```



Quiz

```
val byCountry: Observable[(Country, Observable[(EarthQuake, Country)]]
def runningAverage(s : Observable[Double]): Observable[Double] = {…}
val runningAveragePerCountry : Observable[(Country, Observable[Double])]

a) val runningAveragePerCountry = byCountry.map{
   case (country, quakes) ⇒ (country, runningAverage(quakes))
}

b) val runningAveragePerCountry = byCountry.map{
   case (country, quakes) ⇒ (country, runningAverage(quakes.map(_.Magnitude))
}

c) val runningAveragePerCountry = byCountry.map{
   case (country, cqs) ⇒ (country, runningAverage(cqs.map(_._1.Magnitude))
}
```