Reactive Programming

What is Reactive Programming[1]

- Reactive Programming (RP) is programming with asynchronous data streams.
- Streams are cheap and ubiquitous, anything can be a stream: variables, user inputs, properties, caches, data structures, etc.
- We listen to a stream and react accordingly.
- RP provides a toolbox of functions to create, combine and filter any streams:
  - One or more streams can be used as input to another stream. We can *merge* two streams.
  - A stream can be *filtered to* get another stream that has only those events we have specified.
  - We can *map* data values from one stream to another new stream.
  - See http://reactivex.io/documentation/operators.html
  - See http://rxmarbles.com

Data Streams

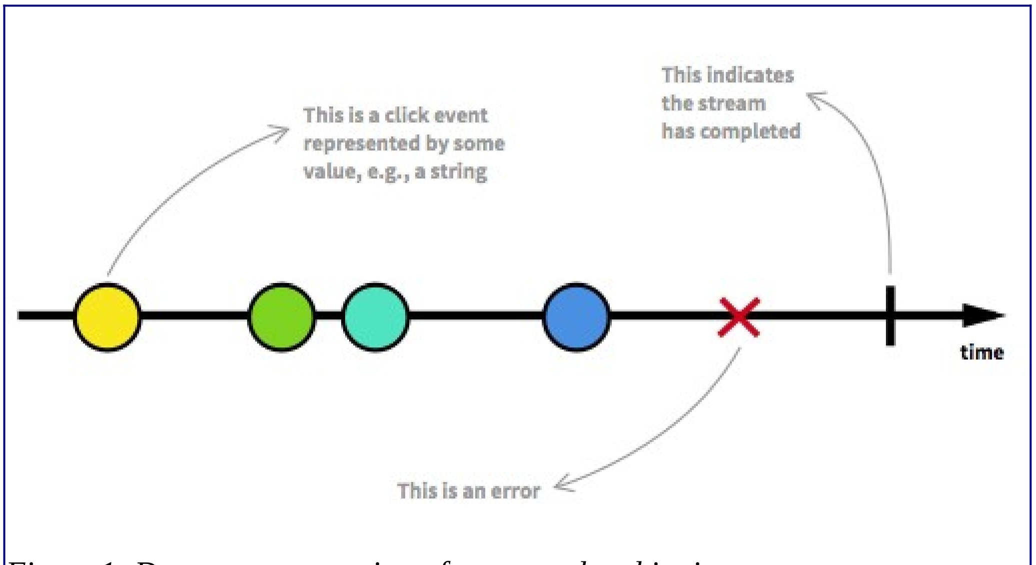
This indicates the stream has completed
This is a click event represented by some value, e.g., a string
time
This is an error

Figure 1: Data streams consists of events ordered in time.

A stream is a sequence of ongoing events ordered in time. It can emit three different things:

1. A value (of some type),
2. An error
3. A completed signal.

[1] See https://gist.github.com/staltz/868e7e9bc2a7b8c1f754

We capture these emitted events only asynchronously, by defining a function that will execute when a value is emitted, another function when an error is emitted, and another function when 'completed' is emitted.

The Observer Design Pattern

The "listening" to the stream is called **subscribing**. The functions we are defining are **observers**. The stream is the subject or **observable** being observed. This is precisely the Observer Design Pattern.

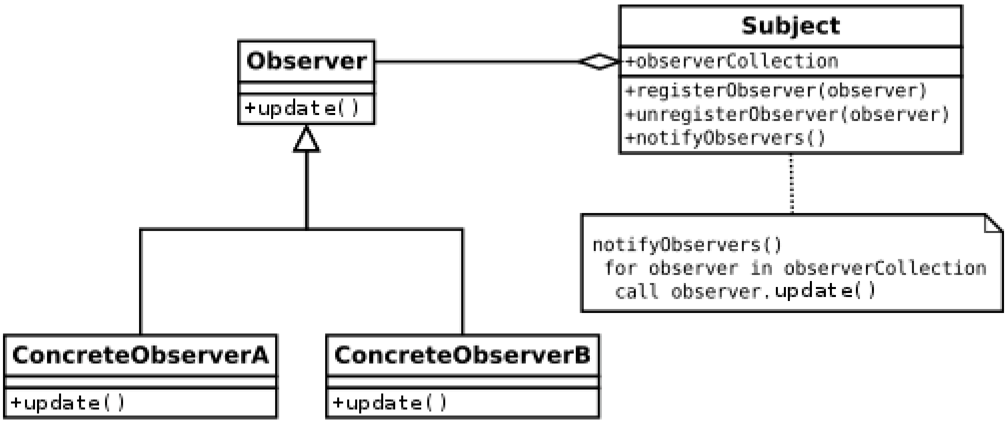

Figure 2: Classic Observer Design Pattern.

The reactive observer design pattern is slightly different from the classical observer pattern.

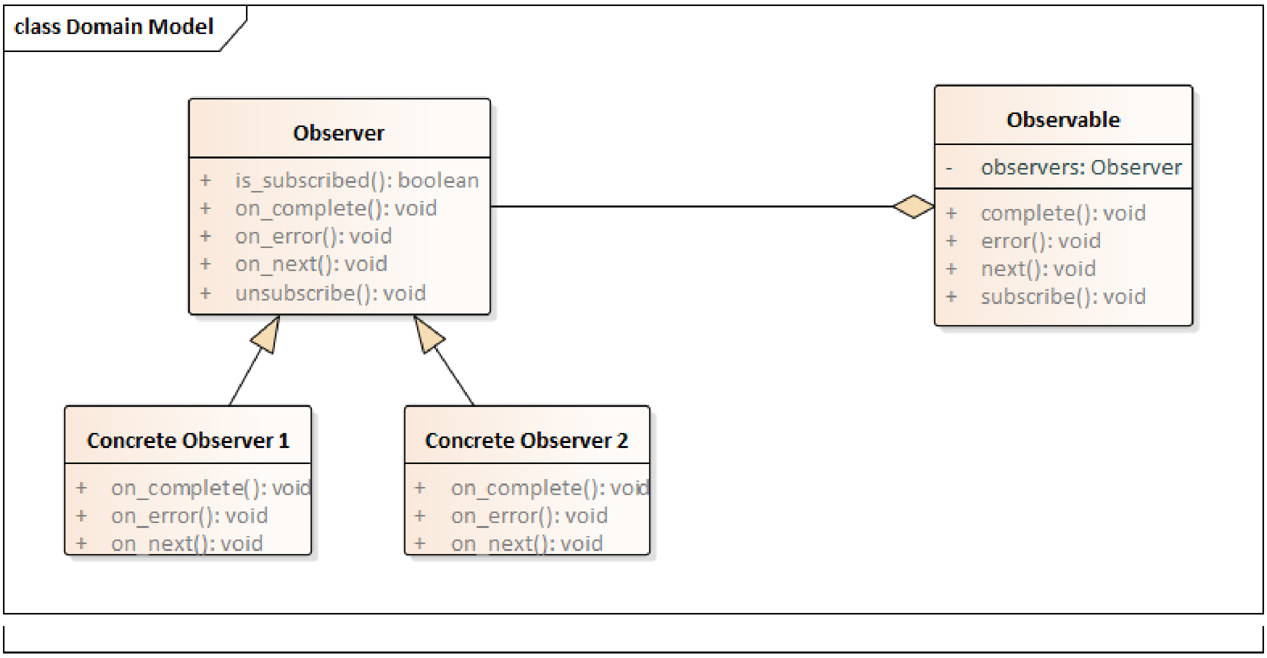

Figure 3: Reactive Observer Design Pattern

Reactive Programming != Reactive System[2]

Probably the most confusing part. Using reactive programming does not build a reactive system. Reactive systems, as defined in the reactive manifesto, are an architectural style to *build responsive distributed systems*. Reactive Systems could be seen as distributed systems done right. A reactive system is characterized by four properties:

- **Responsive**: a reactive system needs to handle requests in a reasonable time (I let you define reasonable).
- **Resilient**: a reactive system must stay responsive in the face of failures (crash, timeout, 500 errors…), so it must be designed for failures and deal with them appropriately.
- **Elastic**: a reactive system must stay responsive under various loads. Consequently, it must scale up and down, and be able to handle the load with minimal resources.
- **Message driven**: components from a reactive system interacts using asynchronous message passing.

The Promises of Reactive Programming[3]

- **Functional**
  Avoid intricate stateful programs, using clean input/output functions over observable streams.

- **Less is more**
  ReactiveX's operators often reduce what was once an elaborate challenge into a few lines of code.

- **Async error handling**
  Traditional try/catch is powerless for errors in asynchronous computations, but ReactiveX is equipped with proper mechanisms for handling errors.

- **Concurrency made easy**
  Observables and Schedulers in ReactiveX allow the programmer to abstract away low-level threading, synchronization, and concurrency issues.

Examples

See https://github.com/henrik7264/RxROS/blob/master/src/rxros_lang/src/rxcpp_examples.cpp

[2] See https://dzone.com/articles/5-things-to-know-about-reactive-programming

[3] See http://reactivex.io/