Chapter 14 - The Observer Pattern in Reactive Programming

September 18, 2021

0.1 Overview

- the `Observer` pattern is useful for notifying an object or a gorup of objects when the state of a given object changes
- this type of traditional Observer applies the publish-subscribe principle, which allows us to react to some object change events
- it provides a nice solution for many cases, but in a situation here we have to deal with many events, some of them depending on each other, the traditional way could lead to complicated, difficult-to-maintain code
- this is where another paradigm called `reactive programming` gives us an intersting option
- in simple terms, the concept of reactive programming is to react to many events `streams of events` while keeping our code clean

- we will focus on the framework `ReactiveX` as part of reactive programming
- the core entity in reactiveX is called an `Observable`
- `ReactiveX` is defined as an API for asychronous programming with observable streams
- in addition to that, we also have the Observer

- you can think of an observable as a stream that can push or emit data to the observer
- and it can also emit events

0.2 Real-World Examples

- streams of water that accumulates somewhere resembles an Observable
- a spreadsheet application can be seen as an example of reactive programming, based on its internal behavior
- all spreadsheet applications, interactively changing any one cell in the sheet will result in immediately reevaluating all formulas that directly or indirectly depend on that cell and updating the display to reflect these reevaluations

0.3 Use Cases

- one use case is the idea of the `Collection Pipeline`
- collection pipelines are programming patterns where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next

- we can use an observable to do operations such as `map` and `reduce` or `groupby` on sequences of objects when processing data
- observables can be created for diverse functions such as button events, requests and RSS/Twitter feeds

0.4 Implementation

0.4.1 Example 1

[1]:
```
import contextlib, io
zen = io.StringIO()
with contextlib.redirect_stdout(zen):
    import this
print(zen.getvalue())
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

[ ]:
```
def get_quotes():
    from rx import Observable, Observer

    import contextlib, io
    zen = io.StringIO()
    with contextlib.redirect_stdout(zen):
    import this
    quotes = zen.getvalue().split('\n')[1:]
    return quotes
```

to create an observable from the list of quotes we get, the way to do it is 1. define a function that hands data items to the `Observer` 2. use an `observable.create()` factory, and pass it that function, to set up the source or stream of data 3. make the `obsserver` subscribe to the source

the observer class itself has three methods used for this type of communication - the `on_next()` is used to pass items - the `on_completed()` will signal that no more items are comming - the `on_error()` signals an error

- we have a function `push_quotes()` that takes an `Observer` object `obs` as input and using a the sequence of quotes, sends each quote using the `on_next()` and signals the end using `on_completed()`

[5]:
```
def push_quotes(obs):
    quotes = get_quotes()
    for q in quotes:
        if q: # skip empty
            obs.on_next(q)
    obs.on_complete()
```

- we implement the `Observer` to be used, u sing a subclass of `Observer` base class

[8]:
```
'''
from rx import Observable, Observer


class ZenQuotesObserver(Observer):
    def on_next(self, value):
        print(f"Received: {value}")

    def on_complete(self):
        print("Done!")

    def on_error(self, error):
        print(f"Error Occurred: {error}")
'''
```

[8]:
```
'\nfrom rx import Observable, Observer\n\n\nclass ZenQuotesObserver(Observer):\n
def on_next(self, value):\n        print(f"Received: {value}")\n        \n
def on_complete(self):\n        print("Done!")\n        \n    def on_error(self,
error):\n        print(f"Error Occurred: {error}")\n'
```

- next, we define the source to be observed:

  `source = Observable.create(push_quotes)`

- finally we define the subscription to the `Observable`, without nothing would happend

  `source.subscribe(ZenQuotesObserver())`

0.4.2 Second Example

- we will rewrite the code from example one to obtain a similar result
- we adopt the `get_quotes()` function in order to return an enumeration of the sequence, using Python's built-in `enumerate()` function

[9]:
```
def get_quotes():
    import contextlib, io
    zen = io.StringIO()
```

```
    with contextlib.redirect_stdout(zen):
        import this

    quotes = zen.getvalue().split('\n')[1:]
    return enumerate(quotes)
```

- we then call that function and store its result in a variable `zen_quotes`

```
zen_quotes = get_quotes()
```

- we create the observable using the special `Observable.from_()` function and chain operations such as `filter()` on the sequence, and finally use the `subscribe()` to subscribe to the observable

Observable.from_(zen_quotes)
.filter(lambda q: len(q[1]) > 0)
.subscribe(lambda value: print(f“Received: {value[0]} - {value[1]}”))

0.4.3 Third Example

- in this third example, we react to the `Observable` (the stream of quotes created using the `get_quotes()` function, using a chain of `flat_map()`, `filter()` and `map()` operations
- the main difference from the previous example is that we schedule the streaming of items so that a new item is emitted every five seconds (`the interval`) using the `Observable.interval()` function
- furthermore, we use the `flat_map()` method to map each emission to an Observable and merge their emissions together into a single Observable
- the main part of the code is follows

```
Observable.interval(5000) \
    .flat_map(lambda seq: Observable.from_(zen_quotes)) \
    .flat_map(lambda q: Observable.from_(q[1].split())) \
    .filter(lambda s: len(s) > 2) \
    .map(lambda s: s.replace('.', '').replace(',', '').replace('!',
    '').replace('-', '')) \
    .map(lambda s: s.lower()) \
    .subscribe(lambda value: print(f"Received: {value}"))
```

- we also add te following line at the end, using the `input()` function to make sure we can stop the execution when the user wants

```
input("Starting... Press any key to quit\n")
```

0.4.4 Fourth Example

- we will use `Faker` to build a stream of the list of people and an Observable based on it

[11]:
```
def populate_people():
    from faker import Faker
    fake = Faker()
```

```
    def populate():
        persons = []
        for _ in range(0, 20):
            p = {'firstname': fake.first_name(), 'lastname': fake.last_name()}
            persons.append(p)
        return iter(persons)
```

- in the main part of the program, we write the names of the people in the list that were generated in the `text` file

```
if __name__ == '__main__':
    new_persons = populate()
    new_data = [f"{p['firstname']} {p['lastname']}" for p in new_persons]
    new_data = ", ".join(new_data) + ", "
    with open('people.txt', 'a') as f:
    f.write(new_data)
```

- we define a function, `firstnames_from_db()` which returns a Observable from text file containing the names, with transformations using `flat_map()`, `filter()` and `map()` methods and a new operation, `group_by()` to emit items from another sequence

[13]:
```
def first_name_db_wrapper():
    from rx import Observable

    def firstnames_from_db(file_name):
        file = open(file_name)
        # collect and push stored people firstnames
        return Observable.from_(file) \
            .flat_map(lambda content: content.split(', ')) \
            .filter(lambda name: name!='') \
            .map(lambda name: name.split()[0]) \
            .group_by(lambda firstname: firstname) \
            .flat_map(lambda grp: grp.count().map(lambda ct: (grp.key, ct)))
```

- then we define an Observable, as in the previous example, which emits data every five seconds, merging its emission with what is returned from `firstnames_from_db(db_file)` after setting the `db_file` to the `people.txt` file as:

```
db_file = "people.txt"
# Emit data every 5 seconds
Observable.interval(5000) \
 .flat_map(lambda i: firstnames_from_db(db_file)) \
 .subscribe(lambda value: print(str(value)))
input("Starting... Press any key to quit\n")
```

- to improve the code above, we will try to get an emission of only the first names that are present at least four times
- we need another observable and filter it
- compared to ones we used in the first version, we have to use the `filter()` operation to only

keep the first name groups for which the count of occurrences (`ct`) value is bigger than three

- if you check the code again, based on the group obtained, we get a tuple containing the group's key as the first element and the count as the second element using the lambda function `n lambda grp: grp.count().map(lambda ct: (grp.key, ct))`, which is emitted thanks to the `.flat_map()` operator
- the next thing to do is further filter using `.filter(lambda name_and_ct: name_and_ct[1] > 3)` in order to only get first names that currently appear at leadt four times

[15]:
```
def frequent_firstnames_from_db(file_name):
    file = open(file_name)
    # collect and push only the frequent firstnames
    return Observable.from_(file) \
        .flat_map(lambda content: content.split(', ')) \
        .filter(lambda name: name!='') \
        .map(lambda name: name.split()[0]) \
        .group_by(lambda firstname: firstname) \
        .flat_map(lambda grp: grp.count().map(lambda ct: (grp.key, ct))) \
        .filter(lambda name_and_ct: name_and_ct[1] > 3)
```

- we add almost the same code for the Observable
- we change only the name of the referenced function accordingly

```
# Emit data every 5 seconds
Observable.interval(5000) \
    .flat_map(lambda i: frequent_firstnames_from_db(db_file)) \
    .subscribe(lambda value: print(str(value)))
# Keep alive until user presses any key
input("Starting... Press any key to quit\n")
```