Chapter 11 - The Observer Pattern

September 18, 2021

0.1 Overview

- when we need to update a group of objects when the state of another object changes, a popular solution is offered by the `Model-View-Controller (MVC)` pattern
- assume that we are using the data of the same `model` in two `views`, for instance in a pie chart and in a spreadsheet
- whenever the model is modified, both the vieews need to be updated

- the observer pattern describes a publish-subscribe relationship between a single object, the publisher, which is also known as the `subject` or the `observable`, and one or more objects, the subscribers, also known as `observers`
- in the case of `MVC`, the publisher is the model and the subscribers are the views

- the ideas behind Observer are the same as those behind the separation of concerns principle, that is, to increase decoupling between the publisher and subscribers, and to make it easy to add/remove subscribers at runtime

0.2 Real-world examples

- `RabbitMQ` library can be used to add asynchronous messaging support to an application
- several messaging protocols are supported such as `HTTP` and `AMQP`
- `RabbitMQ` can be used in a Python application to implement a publish-subscribe pattern, which is nothing more than the Observer design pattern

0.3 Use cases

- we generally use the Observer pattern when we want to inform/update one or more objects (observers/subscribers) about a change that happened on a `given object` (subject/publisher/observer)
- the number of observers, as well as who those observer are may vary and can be changed dynamically

- we can think of many cases where Observers can be useful
- one such use case is `news feeds`
- with RSS, Atom, or other related formats, you follow a feed, ans everytime it is updated, you receive a notification about the update

- `Event-driven systems` are another example where `Observer` is usually used
- in such sysems, you have `listeners` that `listen` for specific events
- the listeners are triggered when an event thay are listening to is created
- this can be typing a specific key, moving the mouse, etc

- the even plays the role of the publisher and the listeners play the role of the observers
- the key point in this case is that multiple listeners (observers) can be attached to a single event (publisher)

0.4 Implementation

- we will be implementing a data formatter
- the ideas described here are based on the `AtiveState` Python Observer code recipe
- there is a default formatter that shows a value in decimal format
- howerver, we can add/register more formatters
- in this example, we will add a hex and binary formatter
- every time the value of the default formatter is updated, the registered formatters are notified and take action
- in this case, the action is to show the new value in the relevant format

- `Observer` is actually one of the patterns where inheritance makes sense
- we can have a base `Publisher` class that contains the common functionality of adding, removing and notifying observers
- our `DefaultFormatter` class derives from `Publishers` and adds the formatter-specific functionality
- and we can dynamically add and remove observers on demand

- we begin with the `Publisher` class
- the observers are kept in the observers list
- the `add()` method registers a new observer, or throws an error it is already exists
- the `remove()` method unregisters an existing observer, or thows an exception if it does not exist
- finally the `notify()` method informs all observers about a change

[8]:

```
class Publisher:
    def __init__(self):
        self.observers = []

    def add(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)
        else:
            print(f'Failed to add {observer}')

    def remove(self, observer):
        try:
            self.observers.remove(observer)
        except ValueError:
            print(f'Failed to remove: {observer}')

    def notify(self):
        [o.notify(self) for o in self.observers]
```

- lets continue with the `DefaultFormatter` class

- the first thing that its `__init__()` does is call the `__init__()` method of the base class, since this is not odne automatically in python
- a `DefaultFormatter` instance has a name to make it easier for us to track its status
- we use name mangling in the `_data` variable to state that it should not be accessed directly
- note that this is always possible in Python but fellow developers have no excuse for doing so, since the code already states that they shouldent
- there is a serious reason for using name mangling in this case
- `DefaultFormatter` treats the `_data` variable as an integer, and the defualt value is zero

- the `__str__()` method returns information about the name of the publisher and the v alue of the `_data` attribute
- `type(self).__name__` is a handy trick to get the name of a class without hardcoding it

- there are two `data()` methods
- the first one uses the `@property` decorator to give read access to the `_data` variable
- using this, we can just execute `object.data` instead of `object.data()`

- the second `data()` method is more interesting
- it uses the `@setter` decorator, which is called every time the assignment (`=`) operator is used to assign a new value to the `_data` variable
- this method also tries to cast a new value to an integer, and does exception handling in case this operation fails

[9]:

```
class DefaultFormatter(Publisher):
    def __init__(self, name):
        Publisher.__init__(self)
        self.name = name
        self._data = 0

    def __str__(self):
        return f"{type(self).__name__}: '{self.name}' has data = {self._data}"

    @property
    def data(self):
        return self._data

    @data.setter
    def data(self, new_value):
        try:
            self._data = int(new_value)
        except ValueError as e:
            print(f'Error: {e}')
        else:
            self.notify()
```

- the next step is to add the observers
- the functionality of `HexFormatter` and `BinaryFormatter` is very similar
- the only differnece between them is how they format the value of data received by the publisher, that is, in hexadecimal and binary

[10]:

```
class HexFormatterObs:
    def notify(self, publisher):
        value = hex(publisher.data)
        print(f"{type(self).__name__}: '{publisher.name}' has now hex dat =␣
 ↪{value}")

class BinaryFormatterObs:
    def notify(self, publisher):
        value = bin(publisher.data)
        print(f"{type(self).__name__}: '{publisher.name}' has now bin data =␣
 ↪{value}")
```

- to help us use those classes, the `main()` function initially creates a `DefaultFormatter` instance named `test1` and afterwords, attaches (and detaches) the two available observers

[11]:

```
def main():
    df = DefaultFormatter('test1')
    print(df)

    print()
    hf = HexFormatterObs()
    df.add(hf)
    df.data = 3
    print(df)

    print()
    bf = BinaryFormatterObs()
    df.add(bf)
    df.data = 21
    print(df)

main()
```

```
DefaultFormatter: 'test1' has data = 0

HexFormatterObs: 'test1' has now hex dat = 0x3
DefaultFormatter: 'test1' has data = 3

HexFormatterObs: 'test1' has now hex dat = 0x15
BinaryFormatterObs: 'test1' has now bin data = 0b10101
DefaultFormatter: 'test1' has data = 21
```