FRUIZIONE E UTILIZZO DEI MATERIALI DIDATTICI

- E' vietata la **copia**, la **rielaborazione**, la **riproduzione** dei contenuti e immagini presenti nelle lezioni in qualsiasi forma

- E' inoltre vietata la **diffusione**, la **redistribuzione** e la **pubblicazione** dei contenuti e immagini, incluse le registrazioni delle videolezioni con qualsiasi modalità e mezzo non autorizzati espressamente dall'autore o da Unica

Observer Design Pattern

Observer Pattern

The observer pattern is useful for state monitoring and event handling situations. This pattern allows a given object to be monitored by an unknown and dynamic group of "observer" objects.

Whenever a value on the core object changes, it lets all the observer objects **know that a change has occurred**, by calling an update() method.

The idea is that you have an object, the **observable**, and a group of other objects (**observers**) that want to be notified when the (inner state of the) **observable** changes.

The **observed object** must maintain a **list of observers** and must warn them of the changes.

Each observer may be responsible for different tasks whenever the observable object changes; the core object doesn't know or care what those tasks are, and the observers don't typically know or care what other observers are doing.

We can adopt another terminology. Instead of **observable** and **observers** we can talk about **publisher** and **subscribers**.

- **observable -> publisher**
- **observers -> subscribers**

Simple Observer

publisher is the observed object. **bob**, **alice** and **john** are the observers.

- **bob**, **alice** and **john** are interested on the state of the publisher.
- When the publisher changes its state, it send a message to all the interested observer using the method `dispatch()`. (Here we simulate the action calling the `publisher.dispatch('message')` method.)
- **bob**, **alice** and **john** receive the message.

- **john** is no more interested on the object publisher
- The next time that the publisher calls the `dispatch('message')` method, the messge is received only by **bob** and **alice**.

Simple Observer

```
publisher=Publisher() # the observed object
bob=Subscriber('Bob') # an observer
alice=Subscriber('Alice') # an observer
john=Subscriber('John') # an observer

# add the subscribers (bob, alice, john)
# to the the subscribers' set of the Publisher

publisher.register(bob)
publisher.register(alice)
publisher.register(john)
```

```
# send a message
publisher.dispatch('Lunchtime!')

# Bob  received the message  **Lunchtime!**
# John  received the message  **Lunchtime!**
# Alice  received the message  **Lunchtime!**

publisher.unregister(john)
publisher.dispatch('Happy hour!')

# Bob  received the message  **Happy hour!**
# Alice  received the message  **Happy hour!**
```

How to implement `Subscriber` and `Publisher`:

Subscriber

- has an instance method `update()` that receives a message from the publisher and prints it.

Publisher

- has a `set`[1] attribute that collects all the subscribers.
- has `register()` and `unregister()` methods that add and remove subscribers from the publisher' set.
- has a `dispatch()` method that send a message to all the subscibers in the set.

(code: `observer/observer1.py`)

---

[1] https://docs.python.org/3/library/stdtypes.html#set. The set has the methods `add()` and `discard()`

Simple Observer (2)

The subscriber can specify not only

"I want to be notified when this interesting event happens"

but also

"I want you to notify me *in this specific way*, that is, *using this specific method*"

For example, we can have
- **bob** and **alice** (`SubscriberOne`) with method `update()`
- **john** (`SubscriberTwo`) with method `receive()`

Simple Observer (2)

```
publisher=Publisher()

bob=SubscriberOne('Bob')
alice=SubscriberOne('Alice')
john=SubscriberTwo('John')

# explicitly uses the 'update()' method
publisher.register(bob, bob.update)

# implicitly uses the 'update()' method
publisher.register(alice)

# explicitly uses the 'receive()' method
publisher.register(john, john.receive)


# send a message
publisher.dispatch('Lunchtime!')
publisher.unregister(john)
publisher.dispatch('Happy hour!')
```

```
# send a message
publisher.dispatch('Lunchtime!')

# Bob  received the message  **Lunchtime!**
# John  received the message  **Lunchtime!**
# Alice  received the message  **Lunchtime!**

publisher.unregister(john)
publisher.dispatch('Happy hour!')

# Bob  received the message  **Happy hour!**
# Alice  received the message  **Happy hour!**
```

How to implement `Subscriber` and `Publisher`:

Subscriber
- has an instance method that receives a message from the publisher and prints it. `update()` could be the default name, but other names are possible, i.e. `receive()`

Publisher
- has a `dictionary` [2] attribute that collects subscribers and their methods.
- has `register()` and `unregister()` methods that add and remove subscribers from the publisher's dictionary.
- has a `dispatch()` method that send a message to all the subscibers in the dictionary, using the appropriate method.

(code: `observer/observer2.py`)

---

[2] https://docs.python.org/3/library/stdtypes.html#dict ; `del d[key]` removes the element identified by `key`. `d.items()` returns all the `(key, value)` couples.

Observing EVENTS

We may want to have an observer that can actually notify
- one group of subscribers for one kind of situation
- a different group of subscribers for a different situation

moreover
- we can even have subscribers that are in both groups

We can call these different situations **events**.

A subscriber can not only register with the observer but register for **a specific event** that the observer can announce.

Observing EVENTS - How to implement `Subscriber` and `Publisher`:

Subscriber has a method that receives a message from the publisher and prints it.

Publisher has

- a `dictionary` attribute `subscribes` with:
    - key: an event
    - value: a **dictionary** that contains subscribers and their methods.
- `register()` and `unregister()` methods that add and remove subscribers from the publisher's dictionary, ***for the specific event only.***
- Publisher has a `dispatch()` method that send a message to all the subscibers in the sub-dictionary ***of that specific event***, using the appropriate method.

(code: `observer/observer3.py`)

Exercises

Define a class `Segment` that contains two points.

Define a set of observers (i.e. `oss1, oss2, oss3`) interested in the 'too small segment' event.

and another set of observers (i.e. `oss1, oss4, oss5`) interested in the 'too big segment' event.