Observer Design Pattern

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

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.

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 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.

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.

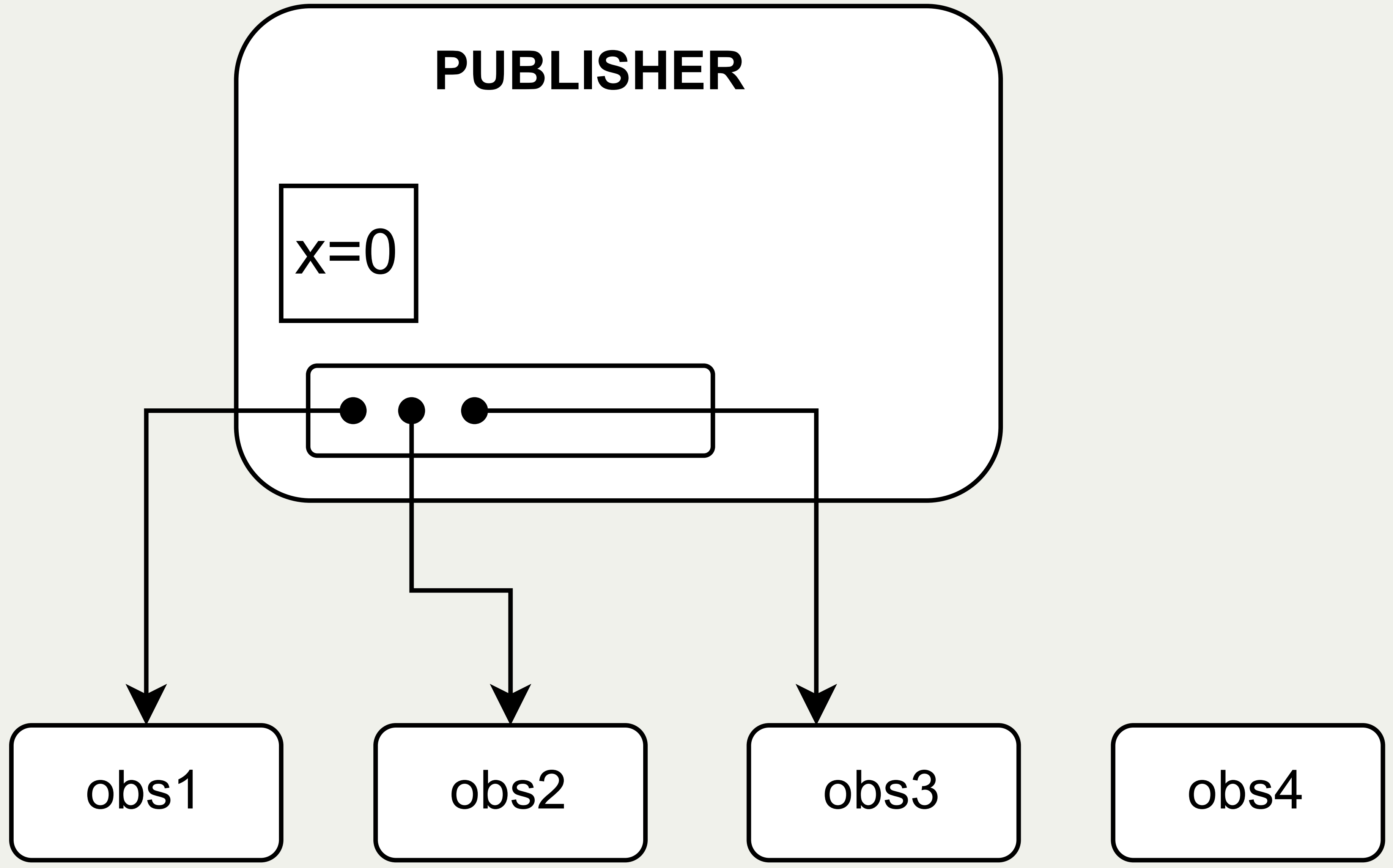
Terminology:

- **observable** or **publisher**
- **observers** or **subscribers**

Simple Observer

In the context of the **Simple Observer** pattern, there exists a **Publisher** and a group of **observers**.

The `Publisher` keeps track of a list of `observers` who are keen on being notified of any changes in the `Publisher`'s state, such as updating a variable or a database table.
When a state change occurs, the `Publisher` signals the observers.

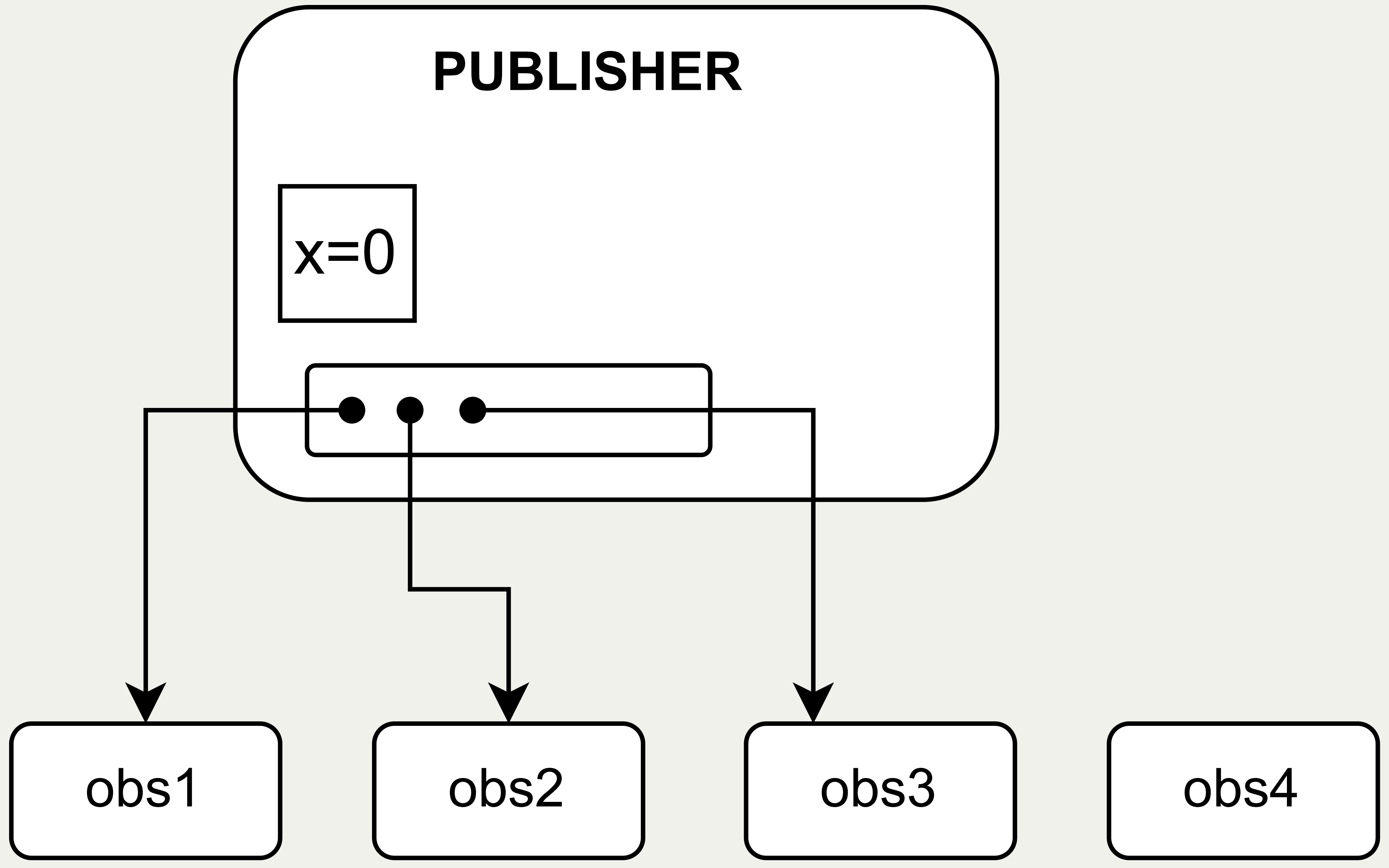


Simple Observer

In the context of the **Simple Observer** pattern, there exists a **Publisher** and a group of **observers**.

The `Publisher` keeps track of a list of `observers` who are keen on being notified of any changes in the `Publisher`'s state, such as updating a variable or a database table.
When a state change occurs, the `Publisher` signals the observers.

Each `observer` has the ability to **register** or **unregister** from the list.



```python
# client code 001
obs1 = Subscriber("obs1")
obs2 = Subscriber("obs2")
obs3 = Subscriber("obs3")
publisher = Publisher(0)

# obs1, obs2 are interested in the publisher's state
publisher.register(obs1)
publisher.register(obs2)

# x does not change. No message from the observers
publisher.x = 0

# x changes. Message from obs1, obs2
publisher.x = 10

# obs1 is no longer interested in the publisher's state.
publisher.unregister(obs1)

# x changes. Message from obs2 only
publisher.x = 20
```

```
Output
--------------------
"obs2  received the message  The new value of x is 10"
"obs1  received the message  The new value of x is 10"
"obs2  received the message  The new value of x is 20"
```

Publisher

- has a `set`(1) attribute that collects all the subscribers. We utilize a `set` instead of a `list` to prevent duplicates in case a subscriber signs up more than once.
- 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 subscribers in the set.

(1) Built-in Types — Python 3.10.1 documentation](https://docs.python.org/3/library/stdtypes.html#set). The set has the methods `add()` and `discard()`

Publisher

- has a `set`[1] attribute that collects all the subscribers. We utilize a `set` instead of a `list` to prevent duplicates in case a subscriber signs up more than once.
- 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 subscribers in the set.

Subscriber

- has an instance method called `update()` which accepts a message from the publisher and prints it.

---

[1] Built-in Types — Python 3.10.1 documentation](https://docs.python.org/3/library/stdtypes.html#set). The set has the methods `add()` and `discard()`

Solution

```python
# The Observer
class Subscriber:

  def __init__(self, name):
    self.name = name

  def update(self, message):
    print(self.name, " received the message ", message)
```

```python
# The Observable
class Publisher:

  def __init__(self, x):
    self.x = x
    self.subscribers = set()

  @property
  def x(self):
    return self._x

  @x.setter
  def x(self, x):
    if not hasattr(self, "_x"):
      # first assignment
      self._x = x
    else:
      if self._x != x:
        # it is a true update!
        self._x = x
        message = "The new value of x is " + str(x)
        self.dispatch(message)

  def register(self, an_observer):
    self.subscribers.add(an_observer)

  def unregister(self, an_observer):
    self.subscribers.discard(an_observer)

  def dispatch(self, message):
    for subscriber in self.subscribers:
      subscriber.update(message)
```

Simple Observer (2)

The observer needs to specify which method the publisher should invoke.

Example:

- **obs1** with method `update()` (the default method)
- **obs3** with method `receive()`

Publisher

- has a **dictionary** attribute that collects all the **subscribers** and the corresponding **method** to be invoked.
- has `register()` and `unregister()` methods that add and remove subscribers from the publisher' **dictionary**.
- has a `dispatch()` method that sends a message to all the subscribers in the dictionary, using the appropriate method.

Publisher

- has a **dictionary** attribute that collects all the **subscribers** and the corresponding **method** to be invoked.
- has `register()` and `unregister()` methods that add and remove subscribers from the publisher' **dictionary**.
- has a `dispatch()` method that sends a message to all the subscribers in the dictionary, using the appropriate method.

Subscriber

- has an instance method that accepts a message from the publisher and prints it. `update()` is the default method name, but the **observer has the flexibility to specify the method that should be invoked**.

```python
# client code 002
obs1 = Subscriber("obs1")
obs2 = Subscriber("obs2")
publisher = Publisher(0)

# obs1, obs2 are interested in the publisher's state
# obs1 implicitly uses the `update()` method
# obs2 explicitly indicate the `receive()` method
publisher.register(obs1)
publisher.register(obs2, obs2.receive)

# x does not change. No message from observers
publisher.x = 0
# x changes. Message from obs1, obs2
publisher.x = 10
# obs1 is no longer interested in the publisher's state.
publisher.unregister(obs1)
# x changes. Message from obs2 only
publisher.x = 20
```

Output

```
--------------------
"obs1  (update method) received the message  The new value of x is 10"
"obs2  (receive method) received the message  The new value of x is 10"
"obs2  (receive method) received the message  The new value of x is 20"
```

Solution

```python
# The Observer
class Subscriber:

  def __init__(self, name):
    self.name = name

  def update(self, message):
    print(self.name, " (update method) received the message ", message)

  def receive(self, message):
    print(self.name, " (receive method) received the message ", message)


# The Observable
class Publisher:

  default_method_name = "update"

  def __init__(self, x):
    self.x = x
    self.subscribers = dict()
    # subscribers is a dictionary.
    #   key: an observer
    #   value: the method to be invoked

  # property, setter as in the previous code
  @property
  def x(self):
    return self._x
```

```python
# class Publisher: (continue)

  @x.setter
  def x(self, x):
    if not hasattr(self, "_x"):
      # first assignment
      self._x = x
    else:
      if self._x != x:
        # it is a true update!
        self._x = x
        message = "The new value of x is " + str(x)
        self.dispatch(message)

  def register(self, an_observer, method_to_invoke=None):
    if method_to_invoke is None:
      # use the 'default' method
      method_to_invoke = getattr(an_observer, self.default_method_
      # store observer and method
      self.subscribers[an_observer] = method_to_invoke

  def unregister(self, an_observer):
    if an_observer in self.subscribers:
      del self.subscribers[an_observer]

  def dispatch(self, message):
    for method_to_invoke in self.subscribers.values():
     method_to_invoke(message)
```

Observing EVENTS

We might desire an `observable` capable of notifying:

- one set of subscribers for a particular situation,
- a distinct set of subscribers for another situation.

Observing EVENTS

We might desire an `observable` capable of notifying:

- one set of subscribers for a particular situation,
- a distinct set of subscribers for another situation.

Furthermore, some subscribers may be part of both sets.
We can refer to these distinct situations as **events**.

Observing EVENTS

We might desire an `observable` capable of notifying:

- one set of subscribers for a particular situation,
- a distinct set of subscribers for another situation.

Furthermore, some subscribers may be part of both sets.
We can refer to these distinct situations as **events**.

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

Publisher

- has a **dictionary** attribute that collects all the **events** as *key*. the correspondent *value* is another dictionary that collects all **subscribers** and the corresponding **method** to be invoked.
- has `register()` and `unregister()` methods that add and remove subscribers from the publisher' **dictionary**, *for the specific event only*.
- has a `dispatch()` method that sends a message to all the subscribers in the dictionary *of that specific event*, using the appropriate method.

Publisher

- has a **dictionary** attribute that collects all the **events** as *key*. the correspondent *value* is another dictionary that collects all **subscribers** and the corresponding **method** to be invoked.
- has `register()` and `unregister()` methods that add and remove subscribers from the publisher' **dictionary**, *for the specific event only*.
- has a `dispatch()` method that sends a message to all the subscribers in the dictionary *of that specific event*, using the appropriate method.

Subscriber

- has an instance method that accepts a message from the publisher and prints it.

```python
# client code 003
obs1 = Subscriber("obs1")
obs2 = Subscriber("obs2")
publisher = Publisher(["x", "y"], 0, 0)  # it accepts in input a list of events

# obs1, obs2 are interested in the publisher's state
# obs1 implicitly uses the `update()` method and it is interested on events 'x changes' and 'y changes'
publisher.register("x", obs1)
publisher.register("y", obs1 )

# obs2 explicitly indicate the `receive()` method and it is interested on event 'y changes'
publisher.register("y", obs2, obs2.receive)

print("# 1 - x changes. Message from obs1")
publisher.x = 10

print("\n# 2 - y changes. Message from obs1 and obs2 ")
publisher.y = 20

print("\n# 3 - no message")  # obs1 is no longer interested in 'x'
publisher.unregister("x", obs1)
publisher.x = 11  # NO MESSAGE

print("\n# 4 - y changes. Message from obs1 and obs2 ")
publisher.y = 30
```

```
Output
--------------------
"# 1 - x changes. Message from obs1"
"obs1  (update method) received the message  The new value of x is 10"

"# 2 - y changes. Message from obs1 and obs2"
"obs1  (update method) received the message  The new value of y is 20"
"obs2  (receive method) received the message  The new value of y is 20"

"# 3 - no message"

"# 4 - y changes. Message from obs1 and obs2"
"obs1  (update method) received the message  The new value of y is 30"
"obs2  (receive method) received the message  The new value of y is 30"
```

Solution

```python
class Publisher:

  default_method_name = "update"

  def __init__(self, events, x, y):
    # init accepts in input a list of events
    #   and uses this list to initialize a dictionary
    self.subscribers = {event: dict() for event in events}
    self._x = x
    self._y = y

  # property, setter as in the previous code
  @property
  def x(self):
    return self._x

  @x.setter
  def x(self,x):
    if self._x != x:
      # it is a true update!
      self._x = x
      message = "The new value of x is " + str(x)
      self.dispatch("x", message)

  @property
  def y(self):
    return self._y
```

```python
# class Publisher: (continue)

  @y.setter
  def y(self,y):
    if self._y != y:
      # it is a true update!
      self._y = y
      message = "The new value of y is " + str(y)
      self.dispatch("y",message)

  def register(self, event, an_observer, method_to_invoke=None):
    if method_to_invoke is None:
      # use the 'default' method
      method_to_invoke = getattr(an_observer, self.default_method_name)
    # store observer and method
    self.subscribers[event][an_observer] = method_to_invoke

  def unregister(self,event, an_observer):
    if an_observer in self.subscribers[event]:
      del self.subscribers[event][an_observer]

  def dispatch(self, event, message):
    for method_to_invoke  in self.subscribers[event].values():
      # returns all values FOR THE DICTIONARY RELATED TO THIS EVENT
      method_to_invoke(message)
```

Monitor the events 'increase' or 'decrease'.

In the Observer design pattern, the responsibility to check whether a value has increased or decreased typically lies with the Publisher, not the Observer.

This is because this design pattern promotes loose coupling, efficiency, and reliability

Monitor the events 'increase' or 'decrease'.

In the Observer design pattern, the responsibility to check whether a value has increased or decreased typically lies with the Publisher, not the Observer.

This is because this design pattern promotes loose coupling, efficiency, and reliability

The main role of the Observer pattern is to allow a set of objects (observers) to be notified of changes in the state of another object (subject or publisher) without being tightly coupled to it.

Discussion

- **Loose Coupling**: The Observer pattern is designed to promote loose coupling between classes, which means that the classes should not depend on each other too tightly. This is important because it makes the code more maintainable and testable. If the Observers were responsible for checking whether a value has increased or decreased, then they would need to know the details of the data being observed, which would make them more tightly coupled to the Publisher.

Discussion

- **Loose Coupling**: The Observer pattern is designed to promote loose coupling between classes, which means that the classes should not depend on each other too tightly. This is important because it makes the code more maintainable and testable. If the Observers were responsible for checking whether a value has increased or decreased, then they would need to know the details of the data being observed, which would make them more tightly coupled to the Publisher.
- **Efficiency**: If the Observers were responsible for checking whether a value has increased or decreased, then they would need to compare the new value to the old value every time they were notified of a change. This could be inefficient, especially if the data changes frequently. It is more efficient for the Publisher to keep track of the previous value and only notify the Observers when the value has actually changed.

Discussion

- **Loose Coupling**: The Observer pattern is designed to promote loose coupling between classes, which means that the classes should not depend on each other too tightly. This is important because it makes the code more maintainable and testable. If the Observers were responsible for checking whether a value has increased or decreased, then they would need to know the details of the data being observed, which would make them more tightly coupled to the Publisher.
- **Efficiency**: If the Observers were responsible for checking whether a value has increased or decreased, then they would need to compare the new value to the old value every time they were notified of a change. This could be inefficient, especially if the data changes frequently. It is more efficient for the Publisher to keep track of the previous value and only notify the Observers when the value has actually changed.
- **Reliability**: If the Observers were responsible for checking whether a value has increased or decreased, then there would be a risk of errors if the comparison was done incorrectly. It is more reliable for the Publisher to do the comparison, as it has access to all of the necessary information.

```python
# client code 004
obs1 = Subscriber("obs1")
obs2 = Subscriber("obs2")
publisher = Publisher(0)

# obs1, obs2 are interested in the publisher's state
# obs1  uses the `it_is_increasing()` method and
#  it is interested on events 'increase'
publisher.register("increase", obs1, obs1.it_is_increasing)

# obs2 uses  the `it_is_decreasing()` method and
#  it is interested on event 'decrease'
publisher.register("decrease", obs2, obs2.it_is_decreasing)

print("# 1 - x increase. Message from obs1")
publisher.x = 10

print("\n# 2 - x decrease. Message from obs2")
publisher.x = 5
```

Solution

```python
class Publisher:

  default_method_name = "update"
  events = ["decrease", "increase"]

  def __init__(self, x):
    self.subscribers = {event: dict() for event in self.events}
    self._x = x

  @property
  def x(self):
    return self._x

  @x.setter
  def x(self, x):
    if self._x != x:
      # it is a true update!
      event = self.events[x > self._x]
      self._x = x
      message = "The new value of x is " + str(x)
      self.dispatch(event, message)

  def register(self, event, an_observer, method_to_invoke=None):
    if method_to_invoke is None:
      # use the 'default' method
      method_to_invoke = getattr(an_observer, self.default_method_name)
    # store observer and method
    self.subscribers[event][an_observer] = method_to_invoke
```

```python
# class Publisher (continue):

  def unregister(self, event, an_observer):
    if an_observer in self.subscribers[event]:
      del self.subscribers[event][an_observer]

  def dispatch(self, event, message):
    for method_to_invoke in self.subscribers[event].values():
      # returns all values FOR THE DICTIONARY RELATED TO THIS EVENT
      method_to_invoke(message)


class Subscriber:

  def __init__(self, name):
    self.name = name

  def it_is_increasing(self, message):
    print(self.name,
          " (`it_is_increasing` method) received the message ",
          message)

  def it_is_decreasing(self, message):
    print(self.name,
          " (`it_is_decreasing` method) received the message ",
          message)
```

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.