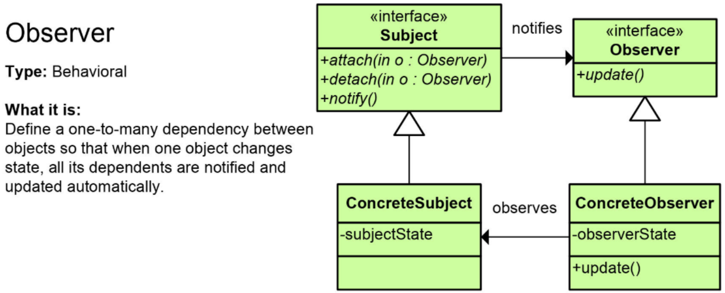
Lecture 18: Observer Pattern
IN628: Programming 4
Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Observer Pattern: GoF

- GoF definition & UML



Observer

Type: Behavioral

What it is:
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Observer Pattern: Definition

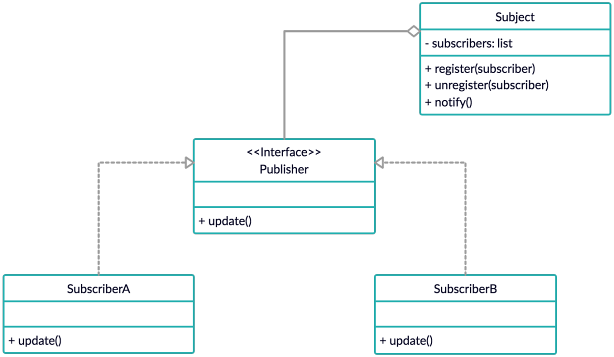
- Behavioural pattern
- An object (subject) maintains a list of its dependents (observers)
- The subject automatically notifies the observers of any state changes
- Mainly used to implement event handling systems
    - The subject is usually called stream of events
    - The observers are called sink of events
- Suits any process where data arrives through I/O
- Most modern programming languages have built-in event constructs

Observer Pattern: Problem

- Purchasing the new Tesla Cybertruck

Observer Pattern: Solution

- Subject class
- Publisher/observer class
- Subscriber/observerable classes

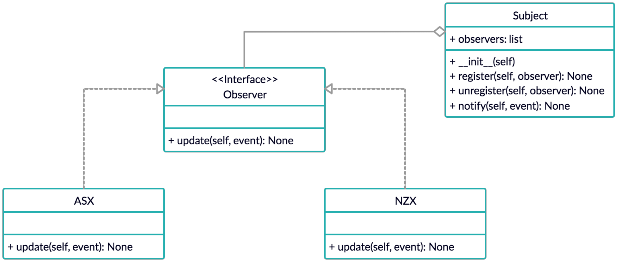


Observer Pattern: Real World Analogy

- Subscription to Time Magazine
- Publisher sends a new issue directly to your address
- Publisher maintains a list of subscribers
- Subscribers can unsubscribe at anytime

Observer Pattern: UML

- Consider the following UML diagram:


Subject
+ observers: list
+ __init__(self)
+ register(self, observer): None
+ unregister(self, observer): None
+ notify(self, event): None
<<Interface>>
Observer
+ update(self, event): None
ASX
+ update(self, event): None
NZX
+ update(self, event): None

Observer Pattern: Implementation

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

    def register(self, observer):
        if not observer in self.observers:
            self.observers.append(observer)

    def unregister(self, observer):
        if observer in self.observers:
            self.observers.remove(observer)

    def notify(self, event):
        for o in self.observers:
            o.update(event)

class Observer:
    def update(self, event):
        pass

class ASX(Observer):
    def update(self, event):
        print(f'ASX — {event}')

class NZX(Observer):
    def update(self, event):
        print(f'NZX — {event}')

def main():
    subject = Subject()
    nzx = NZX()
    subject.register(nzx)
    subject.notify('Update: CEO of NZX has resigned effective immediately.')

if __name__ == '__main__':
    main()  # NZX — Update: CEO of NZX has resigned effective immediately.
```

Observer Pattern: Implementation

```
from abc import ABC, abstractmethod

class Subject(ABC):
    @abstractmethod
    def register(self, observer):
        pass

    @abstractmethod
    def unregister(self, observer):
        pass

    @abstractmethod
    def notify(self, event):
        pass

class ConcreteSubject(Subject):
    def __init__(self):
        self.observers = []

    def register(self, observer):
        if not observer in self.observers:
            self.observers.append(observer)

    def unregister(self, observer):
        if observer in self.observers:
            self.observers.remove(observer)

    def notify(self, event):
        for o in self.observers:
            o.update(event)
```

Observer Pattern: Implementation

```
class Observer(ABC):
    @abstractmethod
    def update(self, event):
        pass

class ASX(Observer):
    def update(self, event):
        print(f'ASX - {event}')

class NZX(Observer):
    def update(self, event):
        print(f'NZX - {event}')

def main():
    concrete_subject = ConcreteSubject()
    nzx = NZX()
    concrete_subject.register(nzx)
    concrete_subject.notify('Update: CEO of NZX has resigned effective immediately.')

if __name__ == '__main__':
    main()  # NZX - Update: CEO of NZX has resigned effective immediately.
```

Observer Pattern: Pros

- New subscribers can be introduced without having to change the publisher's code
- Relations are established between object at runtime

Observer Pattern: Cons

- Subscribers/observerables are notified in random order

Practical

- Series of tasks covering today's lecture
- Worth 1% of your final mark for the Programming 4 course
- Deadline: Friday, 12 June at 5pm