



# Lecture 06: Observer Pattern

## IN710: Object-Oriented Systems Development

### Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Thursday, 5 March

# LECTURE 05: STRATEGY PATTERN RECAP

- ▶ Design pattern 01: strategy pattern
  - ▶ Definition
  - ▶ Problem & solution
  - ▶ Real world analogy
  - ▶ UML & implementation
  - ▶ Open-closed principle
  - ▶ Pros & cons

# LECTURE 06: OBSERVER PATTERN TOPICS

- ▶ Design pattern 02: observer pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ Real world analogy
  - ▶ UML & implementation
  - ▶ Strong vs. weak reference
  - ▶ Pros & cons

# OBSERVER PATTERN: DEFINITION

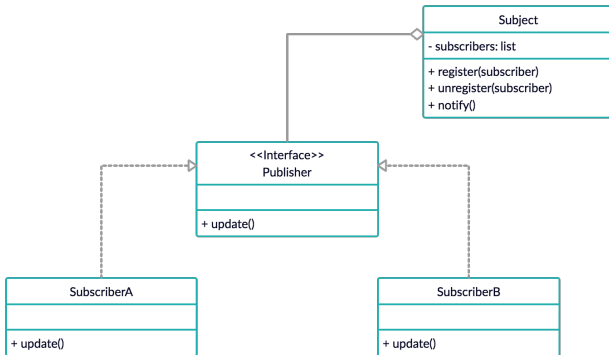
- ▶ Behavioural pattern
- ▶ An object (subject) maintains a list of its dependents (observers)
- ▶ The subject automatically notifies the observers of any state changes
- ▶ Mainly used to implement event handling systems
  - ▶ The subject is usually called stream of events
  - ▶ The observers are called 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/observable classes

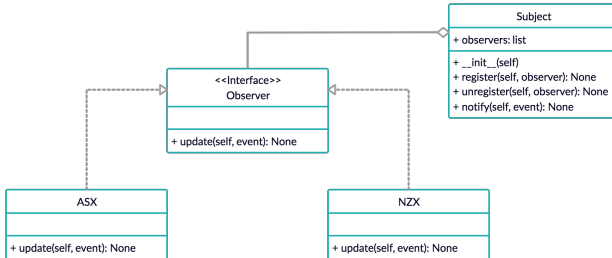


# OBSERVER PATTERN: REAL WORLD ANALOGY

- ▶ 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:





# 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: STRONG Vs. WEAK REFERENCE

- ▶ Memory leaks - lapsed listener problem
- ▶ Basic implementation requires both explicit registration & unregistration
- ▶ Subject holds strong references to the observers
- ▶ Can be prevented by the subject holding weak references to the observers

# 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/observables are notified in random order

# PRACTICAL

- ▶ Series of tasks covering today's lecture
- ▶ Worth 1% of your final mark for the Object-Oriented Systems Development course
- ▶ Deadline: Tuesday, 17 March at 5pm

# REMINDER: EXAM 01

- ▶ Series of tasks covering lectures 01-04
- ▶ Worth 6% of your final mark for the Object-Oriented Systems Development course
- ▶ Deadline: Today at 5pm



# LECTURE 07: FACTORY PATTERN TOPICS

- ▶ Design pattern 03: factory pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ UML & implementation
  - ▶ Applicability
  - ▶ Pros & cons
  - ▶ Relationship with other design patterns