Wzorce Projektowe, implementacje Obserwatora i Singletonu

Szymon Kuliński, Piotr Głownia,
Jakub Jurczak, Janusz Utrata,
Bartłomiej Sroczyński

August 31, 2020

1 Teoria

W poniższym raporcie omówione zostały dwa wzorce:

- Observer (wraz z różnymi odmianami)
- Singleton

1.1 Observer

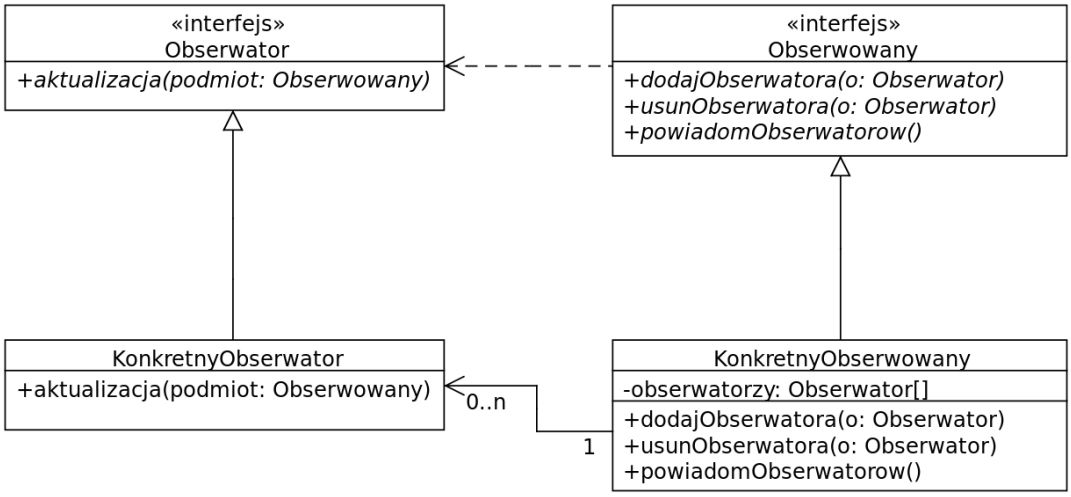
1.1.1 Przeznaczenie

Określa zależność jeden do wielu między obiektami. Kiedy zmieni się stan jednego z obiektów, wszystkie obiekty zależne od niego są o tym automatycznie powiadamiane i aktualizowane.

1.1.2 Warunki stosowania:

- Kiedy abstrakcja ma dwa aspekty, a jeden z nich zależy od drugiego. Zakapsułkowanie tych aspektów w odrębnych obiektach umożliwia modyfikowanie i wielokrotne użytkowanie ich niezależnie od siebie.
- Jeśli zmiana w jednym obiekcie wymaga zmodyfikowania drugiego, a nie wiadomo, ile obiektów trzeba przekształcić.
- Jeżeli obiekt powinien móc powiadamiać inne bez określania ich rodzaju. Oznacza to, że obiekty nie powinny być ściśle powiązane.

1.1.3 Struktura



1.2 Singleton

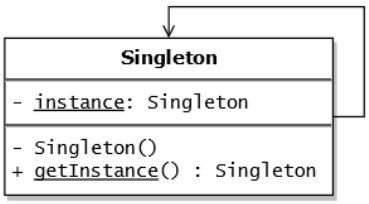
1.2.1 Przeznaczenie

Gwarantuje, że klasa będzie miała wyłącznie jeden egzemplarz, oraz zapewnia do niej globalny dostęp z każdego poziomu programu lub aplikacji.

1.2.2 Warunki stosowania:

- Jeśli musi istnieć dokładnie jeden egzemplarz klasy dostępny klientom w znanym miejscu.
- Kiedy potrzebna jest możliwość rozszerzania jedynego egzemplarza przez tworzenie podklas, a klienty powinny móc korzystać ze wzbogaconego egzemplarza bez konieczności wprowadzania zmian w ich kodzie.

1.2.3 Struktura



2 Implementacja

[1]:
```
from abc import ABC, abstractmethod
```

2.1 Przykład obserwatora

Tworzone są dwie abstakcyjne klasy (interfejsy), które definiują sposób ich komunikacji między sobą. Jest to komunikacja jednostronna.

[2]:
```
class Observer(ABC):

    def __init__(self):
        super().__init__()

    @abstractmethod
    def update(self, value):
        pass


class Observable(ABC):

    def __init__(self):
        super().__init__()

    @abstractmethod
    def subscribe(self, observer:Observer):
        pass

    @abstractmethod
    def unsubscribe(self, observer:Observer):
        pass

    @abstractmethod
    def notify(self):
        pass
```

2.2 Implementacja abstrakcyjnych klas

[3]:
```
class DataProvider(Observable):

    __observers :[Observer] = []
    __result = 0

    def subscribe(self, observer:Observer):
       self.__observers.append(observer)

    def unsubscribe(self, observer:Observer):
        self.__observers.remove(observer)

    def notify(self):
        for observer in self.__observers:
```

```
            observer.update(self.__result)

    def trapez_rule(self,f, a, b, n=100):
        delta_x = ((b - a) / n)
        approx = 0.0
        y_a = f(a)
        y_b = f(b)
        i = 1
        approx_end_start = delta_x * ((y_a + y_b) / 2)
        while i < n:
            approx += delta_x * f(a + (i * delta_x))
            i += 1
            self.__result = approx_end_start + approx
        self.notify()



class MultiplyObserver(Observer):

    def __init__(self, n):
        self.__n = n
        super().__init__()

    def update(self, value):
        result = value * self.__n
        print(result)

class DivideObserver(Observer):

    def __init__(self, n):
        self.__n = n
        super().__init__()

    def update(self, value):
        result = value / self.__n
        print(result)
```

[4]:
```
from math import sin
```

[5]:
```
data_provider = DataProvider()
muliply_observer = MultiplyObserver(2)
devided_observer = DivideObserver(10)
data_provider.subscribe(muliply_observer)
data_provider.subscribe(devided_observer)
data_provider.trapez_rule(lambda x: sin(x),0,2,100)
data_provider.unsubscribe(muliply_observer)
```

```
data_provider.unsubscribe(devided_observer)
```

```
2.832199262675777
0.14160996313378885
```

2.2.1 Observable bez metody unsubscribe

Zamiast metody unsubscribe możemy wrócić w metodzie subscribe funkcje, której wywołanie zpowoduje usunięcie observer-a.

[6]:
```
class ObservableWithoutUbsubscribe(ABC):

    def __init__(self):
        super().__init__()

    @abstractmethod
    def subscribe(self, observer:Observer):
        pass

    @abstractmethod
    def notify(self):
        pass
```

[7]:
```
class DataProviderWithUnsubscribe(ObservableWithoutUbsubscribe):

    __subscribers = []

    __result = 0

    def subscribe(self, observer:Observer):
       self.__subscribers.append(observer)
       return lambda : self.__subscribers.remove(observer)

    def notify(self):
        for subscriber in self.__subscribers:
            subscriber.update(self.__result)

    def trapez_rule(self,f, a, b, n=100):
        delta_x = ((b - a) / n)
        approx = 0.0
        y_a = f(a)
        y_b = f(b)
        i = 1
        approx_end_start = delta_x * ((y_a + y_b) / 2)
        while i < n:
            approx += delta_x * f(a + (i * delta_x))
            i += 1
```

```
            self.__result = approx_end_start + approx
        self.notify()
```

[8]:
```
data_provider_with_ubsubscribe = DataProviderWithUnsubscribe()
muliply_observer = MultiplyObserver(2)
devided_observer = DivideObserver(10)
unsubscribe_1 = data_provider_with_ubsubscribe.subscribe(muliply_observer)
unsubscribe_2 = data_provider_with_ubsubscribe.subscribe(devided_observer)
data_provider_with_ubsubscribe.trapez_rule(
lambda x: x*2+3, -1,1,100 )
unsubscribe_1()
print('-----------------------------------------------')
data_provider_with_ubsubscribe.trapez_rule(
lambda x: x*2+3, -1,1,100 )
unsubscribe_2()
```

```
11.999999999999998
0.5999999999999999
-----------------------------------------------
0.5999999999999999
```

2.3 Bardziej elastyczny Observable

który powiadamia o aktualnym stanie nowych subskrybentów, oraz przyjmuje stan początkowy

[9]:
```
class ObservableImp(ObservableWithoutUbsubscribe):

    __observers = []

    def __init__(self, value):
        self.__value = value
        super().__init__()

    def notify(self):
        for i in self.__observers:
            i.update(self.__value)

    def subscribe(self, observer:Observer):
        self.__observers.append(observer)
        observer.update(self.__value)
        return lambda : self.__observers.remove(observer)

    def set_value(self, value):
        self.__value = value
```

2.4 Implementacja dostawcy danych

[10]:
```
class TrapezRule:

    __instance = None

    @staticmethod
    def get_instance():
        if TrapezRule.__instance is None:
            TrapezRule()
        return TrapezRule()

    def __init__(self):
        self.__observable = ObservableImp(0)
        if TrapezRule.__instance is not None:
            raise Exception("This class is a singleton")
        else: TrapezRule.__instance =self

    def calculate(self,f, a, b, n=100):
        delta_x = ((b - a) / n)
        approx = 0.0
        y_a = f(a)
        y_b = f(b)
        i = 1
        approx_end_start = delta_x * ((y_a + y_b) / 2)
        while i < n:
            approx += delta_x * f(a + (i * delta_x))
            i += 1
            result = approx_end_start + approx
            self.__observable.set_value(result)
        self.__observable.notify()

    def subscribe(self, observer:Observer):
      return self.__observable.subscribe(observer)


class Derivative:

    __instance = None

    @staticmethod
    def get_instance():
        if Derivative.__instance is None:
            Derivative()
        return  Derivative.__instance

    def __init__(self):
```

```
        self.__observable = ObservableImp(0)
        if Derivative.__instance is not None:
            raise Exception("This class is a singleton")
        else: Derivative.__instance = self



    def calculate(self, f, x0, h=0.001):
        result = (f(x0+h)-f(x0))/h
        self.__observable.set_value(result)
        self.__observable.notify()

    def subscribe(self, observer:Observer):
        return self.__observable.subscribe(observer)
```

[11]:
```
derivative = Derivative()
trapez_rule = TrapezRule()
```

[12]:
```
muliply_observer = MultiplyObserver(2)
devided_observer = DivideObserver(10)
muliply_ubsubscribe = derivative.subscribe(muliply_observer)
print("----------------------------------------")
derivative.calculate(lambda x: x**x +2, 2)
print("----------------------------------------")
trapez_rule.calculate(lambda x: x**x *x, 0, 5, 100)
print("----------------------------------------")
muliply_ubsubscribe()
derivative.calculate(lambda x: x**x +2, 2)
trapez_rule.calculate(lambda x: x**x *x, 0, 5, 100)
```

```
0
----------------------------------------
13.5586539640844
----------------------------------------
11399.292451376812
----------------------------------------
```