Jednostka 11 - Paradygmat zdarzeniowy i reaktywny

Joanna Dagil

Grupa TCH-1

30 listopada 2025

1 Cel ćwiczenia

Celem ćwiczenia jest zapoznanie się z podstawami paradygmatu zdarzeniowego i reaktywnego. W szczególności oddzielenie źródła zdarzeń od ich obsługi (handlery, obserwatorzy, funkcje przetwarzające strumień).

2 Zadania

Zadanie 1

```python
import time
import random

class EventEmitter:
    def __init__(self):
        self.handlers = {}

    def on(self, event, handler):
        """Rejestruje handler dla danego zdarzenia."""
        if event not in self.handlers:
            self.handlers[event] = []
        self.handlers[event].append(handler)

    def emit(self, event, *args, **kwargs):
        """Wywołuje wszystkie handlery przypisane do zdarzenia."""
        if event in self.handlers:
            for h in self.handlers[event]:
                h(*args, **kwargs)

# ------ definiowanie i rejestrowanie handlerów ------

emitter = EventEmitter()

def zglos_burze(poziom):
    if poziom > 50:
        print(f"Uwaga! Burza o poziomie {poziom}!")

emitter.on(0, zglos_burze)

def zglos_huragan(poziom):
    print(f"Uwaga! Huragan o poziomie {poziom}!")

emitter.on(1, zglos_huragan)

def zglos_pozar(poziom):
    print(f"Uwaga! Pożar o poziomie {poziom}!")

emitter.on(2, zglos_pozar)

# ------ symulacja strumienia zdarzeń ------
```

```
41
42 def strumien():
43     while True:
44         yield (random.randint(0,4), random.randint(0, 100))
45         time.sleep(0.5)
46
47 def monitor(strumien):
48     for zdarzenie, poziom in strumien:
49         if zdarzenie < 3:
50             emitter.emit(zdarzenie, poziom)
51
52 if __name__ == "__main__":
53     stream = strumien()
54     monitor(stream)
```

Przykładowy przebieg programu:

```
Uwaga! Huragan o poziomie 24!
Uwaga! Pożar o poziomie 12!
Uwaga! Huragan o poziomie 18!
Uwaga! Pożar o poziomie 28!
Uwaga! Pożar o poziomie 32!
Uwaga! Burza o poziomie 59!
```

Zadanie 2

```
1  import time
2  import random
3
4  class Observer:
5      def update(self, dane):
6          raise NotImplementedError
7
8  class Subject:
9      def __init__(self):
10         self.observers = []
11
12     def attach(self, obs):
13         self.observers.append(obs)
14
15     def detach(self, obs):
16         self.observers.remove(obs)
17
18     def notify(self, dane):
19         for o in self.observers:
20             o.update(dane)
21
22     def generate_data(self):
23         """Symuluje generowanie danych pogodowych."""
24         temp = random.randint(-20, 40)
25         humidity = random.randint(0, 100)
26         pressure = random.randint(950, 1050)
27         return (temp, humidity, pressure)
28
29 # Przykład użycia
30 class Logger(Observer):
31     def update(self, dane):
32         print(f"[LOG]")
33
34 class Alarm(Observer):
35     def update(self, dane):
36         temp, humidity, pressure = dane
37         if temp > 30:
38             print("[Alarm] Temperatura przekroczyła 30 C ")
39         if humidity < 20:
40             print("[Alarm] Wilgotność spadła poniżej 20%")
41         if pressure < 980:
42             print("[Alarm] Ciśnienie spadło poniżej 980 hPa")
43
44 class UI(Observer):
```

```
    def update(self, dane):
        print(f"[UI] Aktualizacja danych:")
        print(f"     Temperatura: {dane[0]} C ")
        print(f"     Wilgotność: {dane[1]}%")
        print(f"     Ciśnienie: {dane[2]} hPa")

stacja_pogodowa = Subject()
stacja_pogodowa.attach(Logger())
stacja_pogodowa.attach(Alarm())
stacja_pogodowa.attach(UI())

while True:
    stacja_pogodowa.notify(stacja_pogodowa.generate_data())
    time.sleep(1)
```

Przykładowy przebieg programu:

```
[LOG]
[UI] Aktualizacja danych:
     Temperatura: 27 C
     Wilgotność: 57%
     Ciśnienie: 1031 hPa
[LOG]
[Alarm] Temperatura przekroczyła 30 C
[Alarm] Wilgotność spadła poniżej 20%
[UI] Aktualizacja danych:
     Temperatura: 37 C
     Wilgotność: 7%
     Ciśnienie: 986 hPa
```

Zadanie 3

```
import time
import random

def stream():
    while True:
        yield random.randint(0, 1000)
        time.sleep(0.5)

def filter(stream, predicate):
    for x in stream:
        if predicate(x):
            yield x

def map(stream, fun):
    for x in stream:
        yield fun(x)

stream = stream()
filtered = filter(stream, lambda x: x > 500)
mapped = map(filtered, lambda x: str(x))

for x in mapped:
    print(x)
```

Przykładowy przebieg programu:

```
973
651
535
```

3 Wnioski

3.1 Zalety podejścia zdarzeniowego

- **Modularność i luźne powiązanie** – nowe handlery lub obserwatorów można łatwo dopinać lub odłączać bez zmiany kodu źródła zdarzeń. W zadaniu 1 wystarczy zarejestrować kolejną funkcję

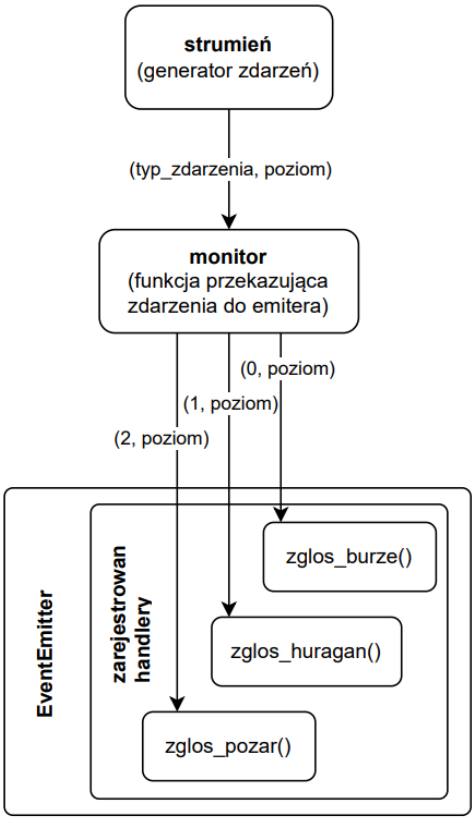
w `EventEmitter`, a w zadaniu 2 dodać nową klasę obserwatora, aby reagować na dodatkowe warunki pogodowe.

- **Naturalne modelowanie strumieni danych** – w zadaniu 3 kolejne elementy strumienia są przetwarzane bez potrzeby trzymania całej kolekcji w pamięci.
- **Podział wywoływań i obsługi** – rozdzielenie mechanizmów wywołania zdarzenia od jego obsługi.

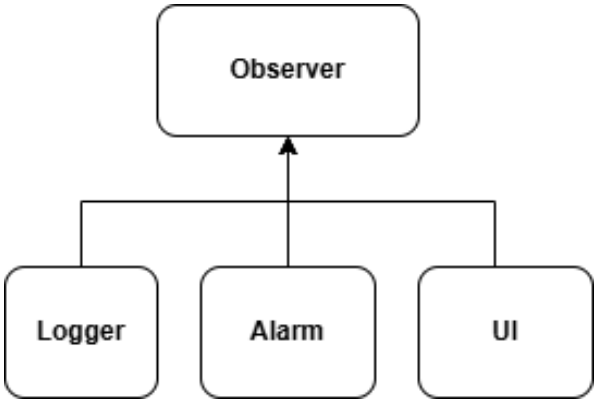
3.2 Ograniczenia podejścia zdarzeniowego

- **Trudniejsze śledzenie przepływu sterowania** – kod reaguje na zdarzenia asynchronicznie, przez co trudniej prześledzić, co zostanie wykonane po kolei, co utrudnia testowanie i debugowanie.
- **Ryzyko braku obsługi zdarzeń** – jeśli dla danego typu zdarzenia nie zarejestruje się handlera/obserwatora, zdarzenie jest po prostu ignorowane, co może prowadzić do trudnych do wykrycia błędów.

3.3 Schematy i diagramy przebiegu zdarzeń w zadaniu 1 i 2



Rysunek 1: Schemat zdarzeń w zadaniu 1



Rysunek 2: Diagram klas obserwatora w zadaniu 2