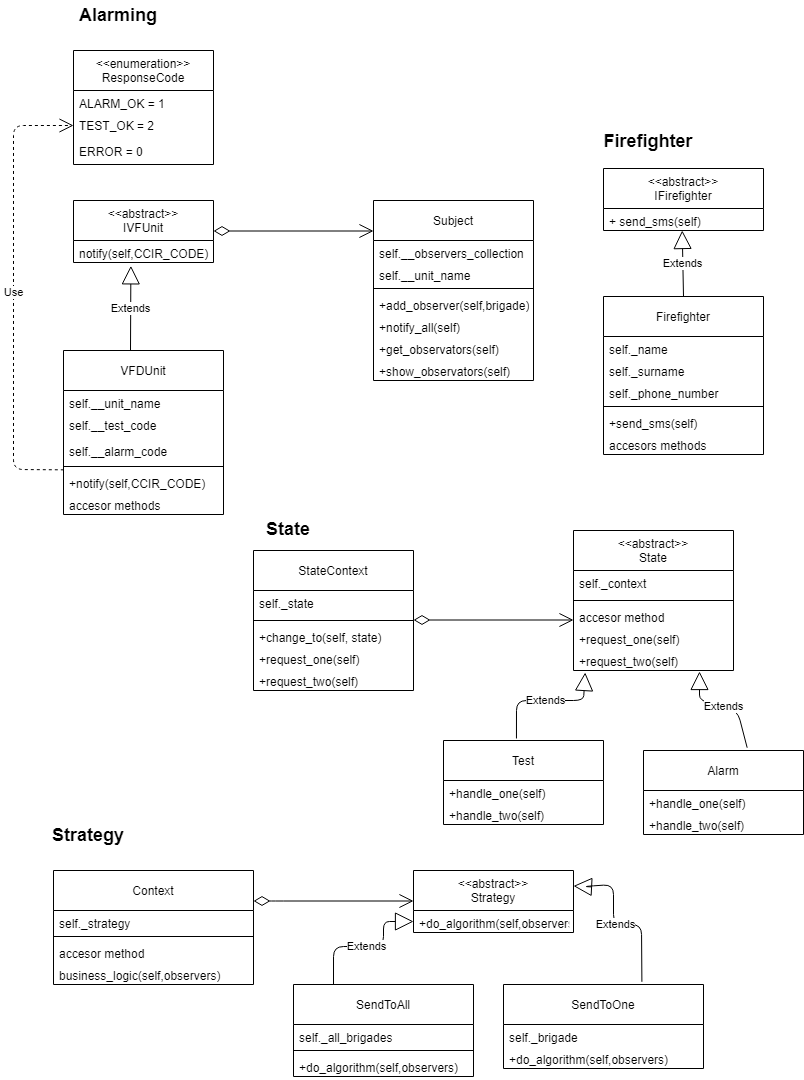
|  |  |  |
| --- | --- | --- |
| Aleksandra Duda | Gr. 31 lab 02 | Techniki Obiektowe |
| Nr indeksu: 130510 | Laboratorium nr 6 | 8.12.20r. |

**Sprawozdanie z labolatorium nr 6**

1. **Cel ćwiczenia**

Modelowanie procesu alramowania dla jednostek ochrony przeciwpożarowej w obszarze powiatu z wykorzystaniem wzorców projektowych.

1. **Model UML**



1. **Przebieg ćwiczenia**

**Alarming**

Klasa ResponseCode jest typem Enum, ma zbiór nazw, którym przyporządkowany jest odpowiedni numer. Definiuje odpowiedź urządzenia DSP\_50 do stanowiska kierowania.



Następnie zdefiniowałam klasę abstrakcyjną **IVFUnit** (w języku Python nie praktykuje się interfejsów). Ma w sobie metodę **notify** , która będzie służyła powiadomieniu jednostki.

Następnie mamy klasę **VFDUnit**, która dziedziczy po **IVFDUnit** , w której jest implemetnacja wspomnianej wcześniej metody **notify** . Zawarłam też settery i gettery dla atrybutów klasy, aby w łatwy sposób je pozyskać.

Tutaj jest zastosowany wzorzec obserwatora, gdzie jest nim **IVFDUnit.**



Następnie mamy clasę **Subject,** która ma referencję do obserwatora (**VFDUnit**) i odpowiada za tworzenie listy jednostek straży. Ma w sobie metody takie jak:

**Add\_observer**  dzięki której możemy dodać nową jednostkę straży, **notify\_all** metoda usprawniająca mechanizm powiadomienia wszystkich jednostek, **get\_observators** zwracanie listy wszystkich zapisanych jednostek, **show\_observators**  - pokazywanie wszystkich zapisanych jednostek.

****

**Firefighter**

Tutaj zdefioniowałam wszystkie dane potrzebne do utworzenia strażaka. Najpierw następuje implemntacja klasy abstrakcyjnej **IFirefighter** gdzie implementujemy metidę **send\_sms** .

Klasa Firefighter dziedziczy po **IFirefighter,** tworząc obiekt musimy podać imię, nazisko i numer kontaktowy do strażaka. Następnie metoda **send\_sms**  odpowiada za wysłanie SMS o alarmie strażakowi. Zdefuniowałam tutaj również settery i gettery, zeby był łatwiejszy dostęp do zmiennych klasowych.



**State**

Tutaj mamy do czynienia z kolejnym wzorcem, mianowicie **State**. Najpierw definiuję kontekst, czyli klasę **StateContext,** gdzie w konstruktorze wywołuję metodę **change\_to**, która zmienia stan odwołując się do klasy **State.** Ma również dwie metody **request\_one** oraz **request\_two**, które stanowią żądania do wykonania. W przypadku, gdy na przykład wywołamy na obiekcie metodę **request\_one** jest wykonywane żądanie przy pomocy wywołania funkcji **handle\_one** w ciele funkcji **request\_one.** Analogicznie jest w przypadku **request\_two.**

****

Następnie zdefuniowałam klasę abstrakcyjną **State** gdzie zdefioniowałam setter i getter dla contextu. Metody **handle\_one** oraz **handle\_two** są abstrakcyjne i są również zadaklrwoane w **State.**



Mamy dwie klasy dziedziczące po **State**, a są to klasy **Test** oraz **Alarm.**  Są to nasze dwa możliwe typy powiadamiania - alarmowe i testowe. W zależności jaki stan chcemy wywołać, bądź zmienić musimy wywołać na obiekcie żądanie, następnie jest ono realizowane metodami **handle\_one** oraz **handle\_two**. Kontekst przekazuje wywołanie do konkretnego stanu – w naszym przypadku jest to Alarm lub Test. Stan możemy zmienić w kontekście. Przy inicjalizacji kontekstu inizjalizuję stan początkowy, ale to dopiero przy wywołaniu funkcji - co opiszę później.

class Test(State):

    def handle\_one(self)->None:

        print("Test handles request\_one")

        print("Changing state to Alarm...")

        self.context.change\_to(Alarm())

    def handle\_two(self)->None:

        print("Test handles request\_two")

class Alarm(State):

    def handle\_one(self)->None:

        print("Alarm handles request\_one")

    def handle\_two(self)->None:

        print("Alarm handles request\_two")

        print("Changing state to Test..")

        self.context.change\_to(Test())

**Strategy**

W strategii najpierw zdefinowałam klasę **Context** , która ma referencję do klasy **Strategy.** Korzystamy z niej zarówno w konstrukotrze, jak i w stworzonych setterach i getterach w tej klasie. Mamy taką funkcję jak **business\_logic** , która pomaga nam wybrać popranwny algorytm w trakcie działania programu. Klasa **Strategy** jest klasą abstrakcyjną dla algorytmów rozwiązujących daną klasę problemów. Implementację możemy przekazać do konkretnej klasy, w moim przypadku do **SendToAll** lub **SendToOne**.



**Main**

W pliku main kolejno wywołuję funkcje dzięki którym mogę zaprezentować działanie programu. Najpierw stworzyłam niewielką listę jednostek oraz strażaków, aby mieć na czym testować swój program



Następnie ustawiam typ powiadomienia na testowy:



Wysyłaniem komunikatów do jednostek zajmuje się komendant, więc pomocniczo zdefioniowałam klasę **Commander:**



Następnie po ustawieniu typu powiadomienia komendant wysyła w tym wypadku do jednej brygady i czeka na odpowiedź jednostek. Gdy jednostki odpowiedzą jest załączana syrena i klasa **FirefighterSender** pełni rolę u nas **DTG-53**:



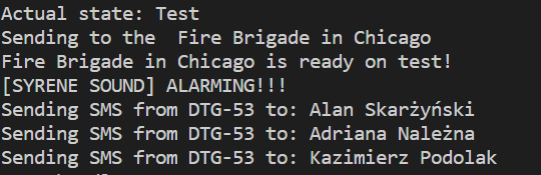


Następnie testuję zmianę stanu na powiadomienie typu Alarm – tym razem wysyłam wszystkim jednostkom i, żeby sprawdzić działanie funkcji **notify**  w klasie **VFDUnit,** czy wyrzuca ResponseCode.ERROR w przypadku nie podania odpowiedniego kodu albo w ogóle nie podania odpowiedzi. Używam do tego funkcji random, która czasem daje odpowiedź jednostki poprawną, a czasem złą. Jeśli zostanie zwrócony błąd to pokazuje się komunikat „ No or wrong response from [nazwa jdnostki] unit!” i w tej jednostce nie jest wtedy włączana syrena, ani nie są wysyłane SMSy do strażaków.

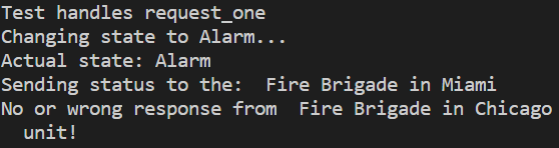
1. **Wnioski i działanie**

* Implementując wzorzec obserwatora możemy w łatwy sposób stworzyć sobie kolekcję jakichś danych, dodawać coś do niej, czy usuwać.
* W przypadku wzorca Strategy strategia nie może się zmienić i jest przekazywana przez konstruktor.
* Poprzez wzorzec Stanu stan może zmienić sam siebie w kontekście. Co oznacza, że nie musimy tworzyć obiektu konkretnego stanu, a wystarczy, że mamy jeden stworzony kontekst i możemy przekazać wywołanie do konkretnego stanu.

Wysyłanie powiadomienia typu test do jednej jednostki. Jednostka przyjęła zgłoszenie, więc włączony został dźwięk syreny i SMSy zostały wysłane do strażaków.



Następnie jest zmiana typu powiadamiania na Alarm i powiadamiane już są wszystkie jednostki.



Jak widzimy kod odpowiedzi od” Fire Bfigade in Chicago „ został albo źle wysłany, źle przetworzony albo nie wysłany, więc syrenę włączamy tylko w jednostce w Miami, bo odpowiedziała na alarm poprawnym kodem i został on dobrze zdekodowany. Zostały wysłane SMS’y do strażaków.

