2 Design Patterns
Fasada

Wstęp

Fasada jest jednym ze strukturalnych wzorców projektowych. Przede wszystkim zapewnia ona użytkownikowi uproszczony interfejs, który będzie delegował realizację złożonych funkcjonalności.

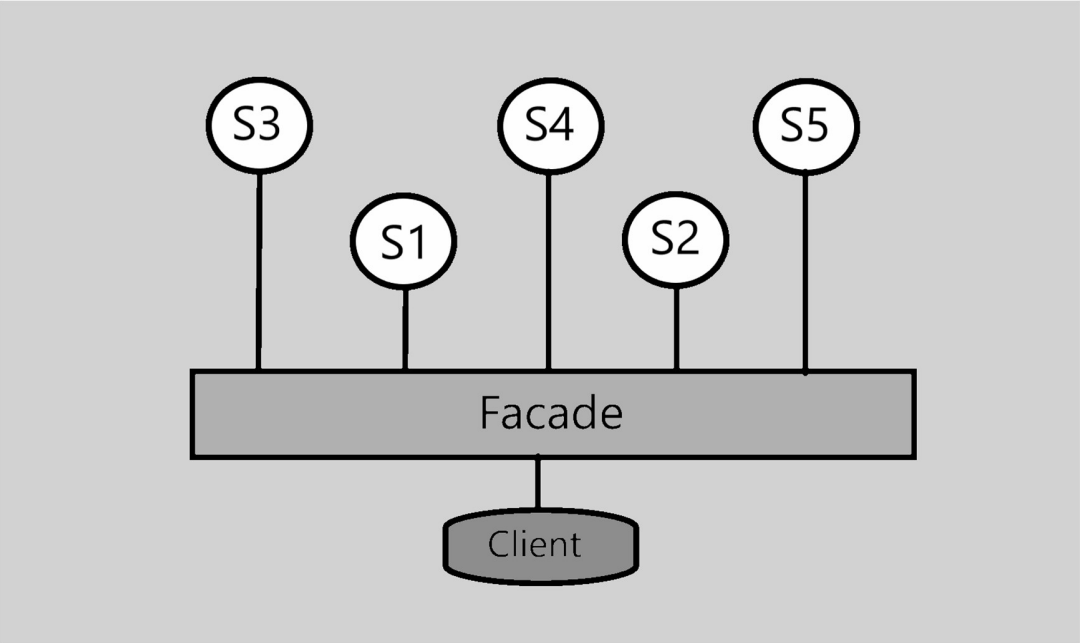
Tak więc celem wprowadzenia wzorca fasadowego jest oddzielenie klienta od całej wewnętrznej logiki biznesowej projektu (*logika biznesowa - to inaczej różne funkcjonalności projektu, np. system rejestracji, system walidacji użytkownika etc.*). Po co wprowadzać Fasadę? Właśnie po to, aby uprościć interakcję użytkownika z systemem (wówczas klient bazuje na jednym interfejsie, który zleca odpowiednie działania swoim komponentom, dzięki czemu user nie czuje się przytłoczony skomplikowaną obsługą komponentów).

Dodatkowo, dzięki wydzieleniu w systemie jednego głównego komponentu, który zlecać będzie odpowiednie operacje subkomponentom, dla użytkownika dostępne i widoczne stają się tylko niektóre funkcjonalności. Użytkownik nie musi znać ani widzieć dokładnie działania systemu wewnętrznego. Tylko potrzebne funkcjonalności są udostępniane na zewnątrz.

Jako przykład fasadowego wzorca projektowego z życia codziennego, często przytaczam przykład bankomatu. Zauważ, że użytkownik, chcąc wypłacić/wpłacić czy dokonać dowolną transakcję pieniężną, korzysta z ujednoliconego interfejsu (całego pudła, zwanego bankomatem :)), a nie rozdrabnia się na poszczególne podzespoły.

Gdyby tak nie było, to klient w zależności od potrzeby, korzystałby oddzielnie z jakiegoś urządzenia, który odpowiadałoby za, np. wpłatę pieniędzy, jeszcze z jakiegoś innego związanego z wypłatą gotówki, czy na końcu sprawdzeniem stanu konta. Stanowczo takie zaprojektowanie funkcjonalności bankomatu byłoby totalnie niepraktyczne i dość uciążliwe.

A tak w rzeczywistości, korzystamy po prostu z prostego “monitora”, który jedynie zleca pewne operacje (np. wpłatę gotówki) do odpowiedniego komponentu bankomatu. Użytkownik wcale nie musi mieć pojęcia o jego istnieniu.



Reasumując zatem:

Zalety:

- Fasada rozdziela klienta od podsystemów danego systemu. Może więc zapewniać bezpieczeństwo, bo klient nie ma bezpośredniego dostępu do metod podsystemu.

- Klient jest oddzielony od niepotrzebnej dla niego wiedzy złożoności działania danego podsystemu.

Z tego powodu właśnie, wzorzec projektowy nazywany fasadą idealnie sprawdza się choćby przy tworzeniu logiki prostych gier zręcznościowych. Programiści wówczas tworzą Managera gry, który obsługuje kolejne komponenty gry (np. Menu i jego wyświetlanie, tworzenie i operowanie na Boardzie gry itd.)

Wady:

- Utworzona przez Nas fasada zależy niejako od komponentów, z których korzysta. Tak więc, dodanie czy zmodyfikowanie pewnej istniejącej funkcjonalności poza "klasą sterującą", będzie również wymagało od Nas wprowadzenie zmian do fasady.
- Jeżeli okaże się, że fasada nagle przestała działać, to użytkownik odnosi wrażenie, jakoby wraz z nią "padł" cały system.

Implementacja

Aby dobrze zobrazować praktyczne wykorzystanie wzorca Fasady, zaimplementujemy symulację "inteligentnego" systemu zarządzania domem. Fasadą będzie klasa RemoteControl, do której użytkownik będzie wydawał polecenia i będzie w odpowiedni sposób przekierowywał działanie do komponentów (np. obsługa światła w korytarzu, sterowanie drzwiami domu).

Dzięki takiemu podejściu, user będzie jedynie wydawał proste polecenia do klasy sterującej bez rozdrabniania się na obsługę poszczególnych komponentów systemu. Mega praktyczne i wygodne rozwiązanie!

```
class HallLight:
    def __init__(self):
        self.name = "Hall Light 2.0"
        print("Hall Light's been added to the system!")

    def light(self):
        print(f"{self.name}'s been lighted!")

    def switch_off(self):
        print(f"{self.name}'s been switched off!")


class MainDoor:
    def __init__(self):
        self.name = "Wooden Main Door"
        print("Wooden Main Door's been added to the system!")

    def open(self):
        print(f"{self.name}'s been opened!")

    def close(self):
        print(f"{self.name}'s been closed!")


class GarageDoor:
    def __init__(self):
        self.name = "Solid Garage Door"
        print("Garage Door's been added to the system")

    def open(self):
        print(f"{self.name}'s been opened!'")

    def close(self):
        print(f"{self.name}'s been closed!")
```

```
class RemoteControl:
    def __init__(self):
        self.hall_light = HallLight()
        self.doors = {"main_door": MainDoor(), "garage_door": GarageDoor()}

    def close_doors(self):
        print("Closing all doors!")
        for door in self.doors:
            self.doors[door].close()

        print()

    def open_doors(self):
        print("Opening all doors!")
        for door in self.doors:
            self.doors[door].open()
        print()

    def light_hall(self):
        self.hall_light.light()
        print()

    def switch_off_hall(self):
        self.hall_light.switch_off()
        print()

remote = RemoteControl()
remote.open_doors()
remote.light_hall()
remote.switch_off_hall()
remote.close_doors()
```

Output:

```
Hall Light's been added to the system!
Wooden Main Door's been added to the system!
```

```
Garage Door's been added to the system
Opening all doors!
Wooden Main Door's been opened!
Solid Garage Door's been opened!'

Hall Light 2.0's been lighted!

Hall Light 2.0's been switched off!

Closing all doors!
Wooden Main Door's been closed!
Solid Garage Door's been closed!
```

Powyższy przykład nie jest mocno skomplikowany i zauważ, że RemoteControl możemy dowolnie dostosowywać do swoich potrzeb. Mógłbyś dodać choćby do niego inne metody - takie jak zamknięcie i otwarcie pojedynczych drzwi, zapalenie światła po otwarciu poszczególnych drzwi itd.