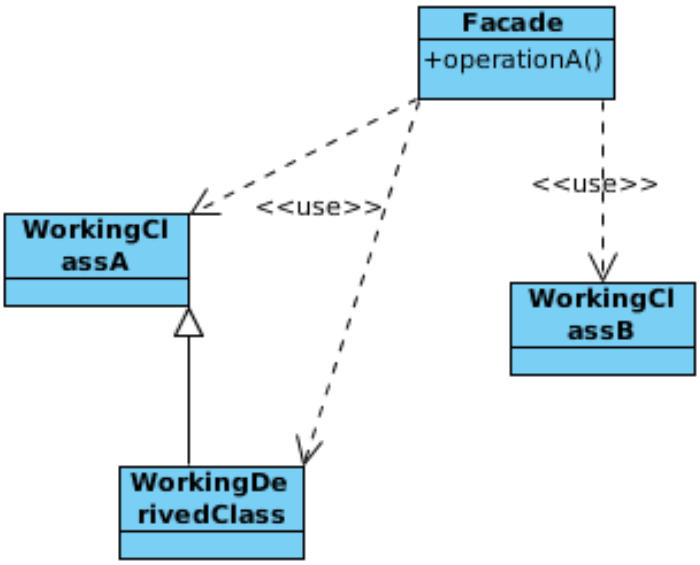


Rysunek 1: Struktura Fasady

1 Fasada

Fasada - obiekt interfejsowy, który umożliwia wygodniejszy dostęp do podsystemu.

Elementy i połączenia :

- Klasy podsystemu, do którego dostęp ma być ułatwiony.
- Klasa fasady, klasa, która ułatwia użycie podsystemu.

```
class Facade{
  public double complicatedOperation(... args){
    SubsystemClassA arguments = ....
    SubsystemConfigClass config = ...// get Config
    SubsystemClassB doer = ... // na przykład pobrać z fabryki,
   użyć wzorca builer
    if (doer.doSomething(arguments) == null){
      return 0;
    }
```

```
9       else{
          return doer.calculate(...);
11      }
    }
13  }
```

- Podsystem nie powinien wiedzieć o fasadzie (czyli zależeć od niej).
- Fasada stanowi interfejs wysokiego poziomu i jest zasadniczo opcjonalna. Klient może odnieść się do klas ukrytych za fasadą (czyli interfejsu niższego poziomu abstrakcji). Jeżeli jest to jednak sytuacja typowa - być może fasada w ogóle nie ma racji bytu w tym przypadku.
- Jeżeli kod klienta używa tylko fasady to można implementację podsystemu podmienić bez zmiany klienta. Jednak taka izolacja nie jest częścią wzorca.

Zadanie 1. Na kopalni za wymianę powietrza odpowiedzialne są ogromne wentylatory (zwykle instalowane parami). Sterowanie nimi uwzględnia różne przypadki awarii, równoważenia obiążeń, pożary itp. Podstawową funkcją jednak jest zatrzymywanie i uruchamianie. Ale i to też nie jest takie proste.

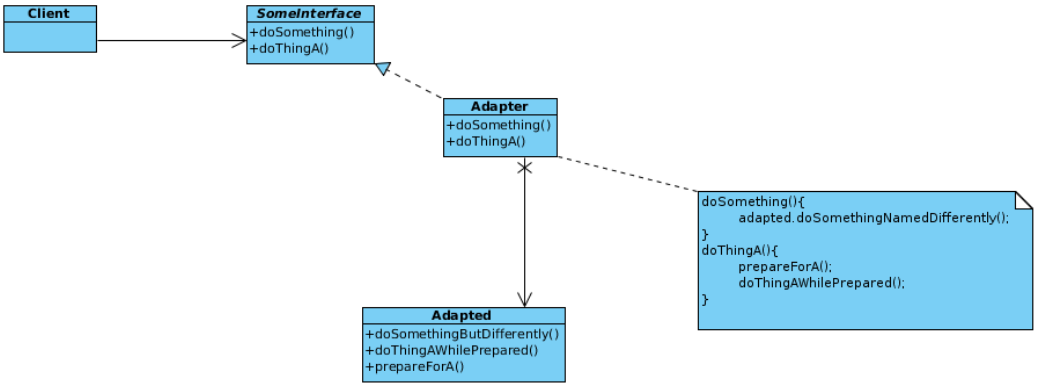
Procedura uruchamiania (w uproszczeniu) wygląda tak:

1. Uruchomić pompy oleju (pompy są 2 na wentylator, ich sterownik (klasa `SterownikPompyOleju`) ma, powiedzmy, tylko metody `wlacz` i `wylacz`.
2. Zwolnić hamulec (znów - sterownik hamulca ma tylko dwie metody).
3. Zewrzeć główny włącznik prądowy.
4. Zewrzeć włącznik prądu wzbudzenia.
5. Odczekać zadany czas (to w zasadzie można pominąć w uproszczeniu).
6. Rozewrzeć wyłącznik prądu wzbudzenia.

W rzeczywistości to jest znacznie bardziej skomplikowane. Wentylator obsługiwany jest przez więcej urządeń, w tym wiele pomiarowych, na których trzeba wykonać diagnostykę. Ale nie o to chodzi w zadaniu.

Również w uproszczeniu wyłączanie wygląda następująco:

1. Rozewrzeć główny wyłącznik.



Rysunek 2: Struktura wzorca Adapter (wersja “obiektowa”)

2. Załączyć hamulec.

3. Odczekać aż wentylator się zatrzyma (w zadaniu można pominąć).

4. Wyłączyć pompy oleju.

Kod klienta natomiast chciałby mieć 2 metody (dla uproszczenia powiedzmy, że czekające na zakończenie algorytmu - wywoływane synchronicznie):
`wlaczWentylator(int numerWentylatora)` `wylaczWentylator(int numerWentylatora)`,
ale również możliwość sterowania każdym z urządzeń oddzielnie. Proszę zaimplementować tą strukturę klas (sterowniki wspomianych urządzeń) z fasadą do włączania i wyłączania.

Każdy sterownik to oddzielna klasa. Raczej nie dziedziczą po wspólnej klasie abstrakcyjnej. Oczywiście funkcje na ”dole” wywołań po prostu tylko wypisują, co mają zrobić np. “włączam pompę oleju” albo “czekam na zatrzymanie wentylatora”.

2 Adapter

Adapter pozwala na użycie w kliencie klasy robiącej to, czego wymaga klient, ale prezentującej technicznie niepasujący interfejs.

Elementy i połączenia :

- Klasa adaptująca - posiada wymagany interfejs, zasadniczo przekazuje wywołania do obiektu klasy adaptowanej.

- Klasa adaptowana - dostarcza podstawowej funkcjonalności, ale w nieodpowienim interfejsie.
- Interfejs, który ma być adaptowany.
- Klient - używa interfejsu, nie musi wiedzieć, że typem konkretnym jest adapter.

```
Adapter implements SomeInterface{

  Adapted adapted;

  T interfaceOperationA(){

    adapted.nonInterfaceOperationA();

  }

  T interfaceOperationB(){

      adapted.prepareForB();

      adapted.doB();

  }

}
```

- Klasyczny adapter może po prostu zamieniać nazwy, kolejność parametrów itp. ale też wykonywać inne operacje niezbędne do użycia klasy adaptowanej - np. przetwarzać argumenty i wynik, zmieniać stan obiektu, nawet tworzyć obiekty klasy adaptowanej. Jeżeli dodamy odpowiednio dużo funkcjonalności do adaptera (tyle, że "adaptowany" obiekt już nie dostarcza jej większości) to mamy po prostu jedną klasę używającą innej (a nie właściwy "adapter"). Czasem może to skutkować wzorcem *Most*.
- Używany dość często, ale w specyficznej sytuacji - zasadniczo przy łączeniu z bibliotekami, innymi systemami, starszymi częściami systemu itp. Nowoprojektowany system raczej nie powinien wymagać adapterów *wewnątrz* - trzeba dobrze zaprojektować interfejsy.

- Ma też dość duże znaczenie, ponieważ pozwala na zwiększenie ponownego użycia istniejącego kodu – zazwyczaj zmniejszając koszty i prawdopodobieństwo błędów w stosunku do pisania od nowa.
- Może być zrobiony przez kompozycję (jak powyżej) albo przez dziedziczenie - dziedziczy wtedy implementację po klasie adaptowanej jednocześnie implementując wymagany interfejs (ew. wielodziedziczy np. w C++).
- W Javie np. InputStreamReader (przykład adaptera robiącego trochę poważniejsze zmiany).
- Wspiera użycie istniejących klas w innych wzorcach, z których wiele wymaga wspólnego interfejsu dla jakiejś grupy części.
- W przeciwieństwie do Fasady zarówno interfejs dostarczany jak i używany istnieją (koncepcyjnie) przed użyciem adaptera.