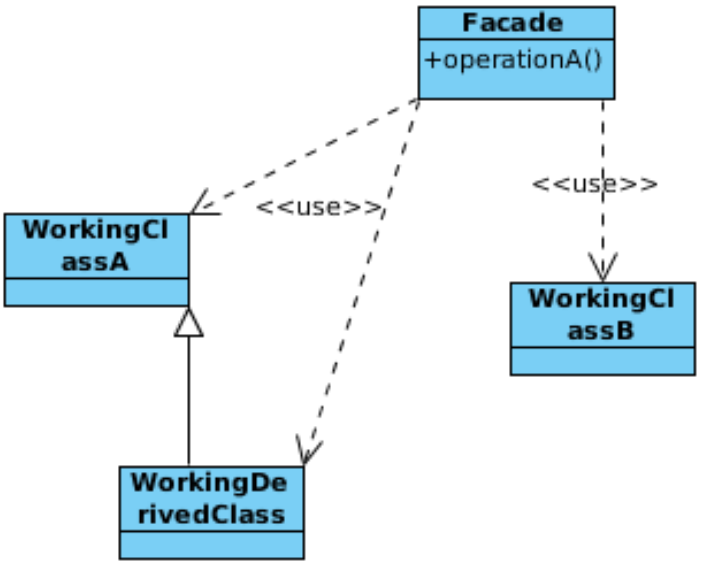


Rysunek 1: Struktura Fasady

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 praktycznie podmienić (z punktu widzenia tego klienta).

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.

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.

Oczywiście funkcje na "dole" wywołań po prostu tylko wypisują, co mają zrobić np. "włączam pompę oleju" albo "czekam na zatrzymanie wentylatora".

2 Dekorator (decorator)

. W skrócie: używamy kompozycji zamiast tworzenia podklas do zmiany zachowania obiektu (przede wszystkim do elastycznego dodania nowej odpowiedzialności) Wciąż używamy dziedziczenia interfejsu.

Elementy i połączenia :

- Obiekt dekorowany, przechowywany przez dekorator, jego metody są zwykle wywoływanie jako część wywołań metody dekoratora.

- Dekorator - przechowuje dekorowany obiekt, dostarcza zasadniczo takiego samego interfejsu, ale dodaje coś do operacji (zwykle wywołuje operacje obiektu dekorowanego).

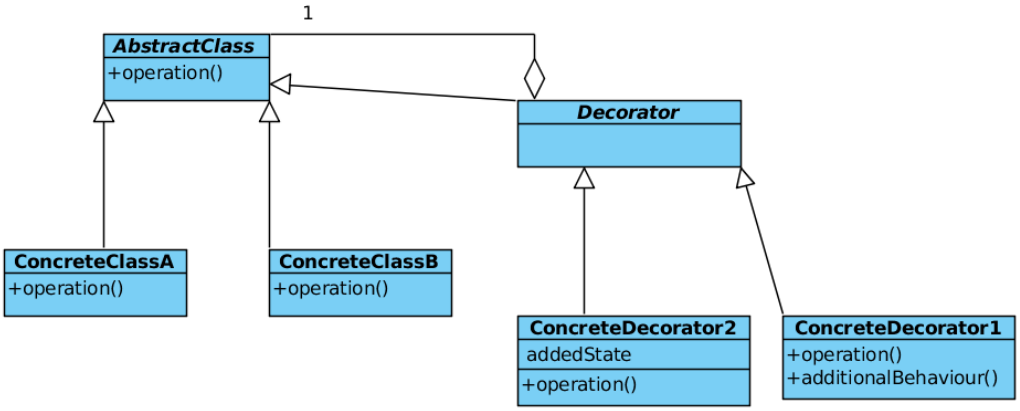
Kod wygląda mniej więcej tak:

```
interface Dekorowany{
  operacja();
}
class DekorowanyKonkretnyA implements Dekorowany{
  operacja(){
    //...
    }
}
class DekorowanyKonkretnyB implements Dekorowany{
  operacja(){
```

```
11      //...
        }
13  }
    class Dekorator implements Dekorowany{
15      Dekorowany dekorowany;
        operacja(){
17        // Zrób coś jeszcze...
          dekorowany.operacja();
19        // Zrób coś innego
        }
21      Dekorator(Dekorowany d){
          this.dekorowanyu = d;
23      }
    }
25
    class KlientDekoratora{
27  //...
      zrobCos(){
29    //...
        Dekorator dr = new Dekorator(new
       DekorowanyKonkretnyA());
31      Dekorator dr2 = new Dekorator(new
       DekorowanyKonkretnyB());
        dr.operacja();
33      dr2.operacja();
      }
35  }
```

Nieco bardziej konkretnie to np.:

```
1   abstract class GraphicsObject{
      paint();
3   }
    class Button extends GraphicsObject{
5     paint(){
        // paint a button
7       }
    }
9   class TextField extends GraphicsObject{
      operacja(){
11      // paint a text field
```



Rysunek 2: Strutkura Dekoratora

```
    }
}
class GracphisObjectWithBorder extends GraphicsObject{
    GraphicsObject primary;
    operacja(){
  paintBorders(primary);
  primary.paint();
    }
}
```

- Dekorator raczej nie powinien wymagać informacji, jaki dokładnie obiekt dekoruje, tylko polegać na abstrakcji - interfejsie. Można stosować wtedy dekorowanie wielopoziomowe.
- Można wymieniać dekoratory niezależnie od klas dekorowanych – duża elastyczność.
- Można stosować dekorację wielopoziomową.
- Można zrobić dużo rzeczy, nawet całkowicie nadpisać metodę, jednak mamy ograniczony dostęp do wewnętrznego stanu obiektu dekorowanego (tyle, na ile pozwala interfejs) i nie możemy ingerować w wykonanie metody obiektu bazowego (np. jeśli tam była template method) - dlatego możliwości zmian są mniejsze niż np. w dziedziczeniu.

Zadanie 2. Proszę w poprzednim zadaniu napisać dekoratory zliczające cykle włączeń i wyłączeń (lub użyć) poszczególnych urządzeń i zademonstrować ich działanie.