# Flyweight (pyłek)

## *Cel:*

Zastąpienie dużej liczby drobnych obiektów niewielką liczą obiektów współdzielonych. Obniża to koszt składowania obiektów.

## *Przykład:*

- Z punktu widzenia projektu, dobrze jest nawet najmniejsze elementy systemu reprezentować w postaci obiektów (dzięki temu mogą one np. mieć swoje operacje).
- Niekiedy oznacza to niepotrzebną powtarzalność danych (gdy wiele obiektów ma identyczny lub podobny stan). Jeśli obiektów jest bardzo dużo i są bardzo małe, koszt składowania jest duży w stosunku do pamięci przechowującej dane.
- Rozwiązanie: wydzielenie stanu wewnętrznego i zewnętrznego. Stan wewnętrzny jest przechowywany w obiekcie; składa się z informacji niezależnych od kontekstu, a zatem takich, które mogą być współdzielone. Stan zewnętrzny zależy od kontekstu – klient jest odpowiedzialny za jego przechowywanie i przekazywanie obiektowi, gdy ten tego potrzebuje.

```
// przykład:
class Znak {
     char znak;
     void wypisz() {
          System.out.print(znak);
     }
     // ...
}

// ... Client:
Znak[] txt = { new Znak('A'), new Znak('b'), new Znak('b'),
               new Znak('B'), new Znak('a'), new Znak('a'),
               new Znak('b'), new Znak('a'), new Znak('B') };
for(Znak z : txt)
     z.wypisz();
```

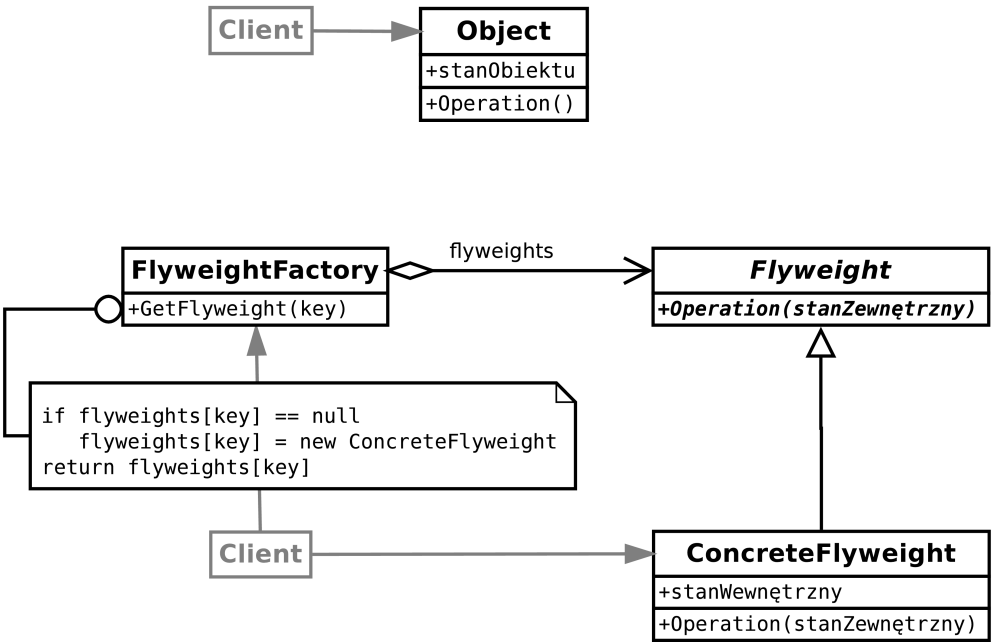
```
// wersja z obiektami współdzielonymi:
class Znak {
     char znak; // stan wewnętrzny
     void wypisz(boolean big) {
          System.out.print(big?(char)(znak-32):znak);
     }
     // ...
}

// ... Client:
Znak a = new Znak('a'), b = new Znak('b');
Znak[] txt = { a, b, b, b, a, a, b, a, b };
boolean[] big = { true, false, false, true, false, false,
                  false, false, true };
for(int i = 0; i < txt.length; ++i)
     txt[i].wypisz(big[i]);
```

## *Zastosowanie:*

Efektywność Flyweighta zależy od sposobu i miejsca jego użycia, dlatego będziemy go stosować, gdy wszystkie warunki są spełnione:

- aplikacja korzysta z dużej liczby obiektów,
- koszty składowania obiektów są wysokie (z powodu dużej ich liczby),
- można przenieść na zewnątrz większość stanu obiektu,
- po usunięciu stanu zewnętrznego grupa obiektów może być zastąpiona jednym obiektem współdzielonym.

## *Struktura:*


Client
Object
+stanObiektu
+Operation()
FlyweightFactory
+GetFlyweight(key)
flyweights
Flyweight
+Operation(stanZewnętrzny)
if flyweights[key] == null
flyweights[key] = new ConcreteFlyweight
return flyweights[key]
Client
ConcreteFlyweight
+stanWewnętrzny
+Operation(stanZewnętrzny)

## *Składniki:*

- ***Flyweight***
  - deklaruje interfejs, przez który obiekt otrzyma stan zewnętrzny
- ***ConcreteFlyweight***
  - obiekt współdzielony, przechowuje stan wewnętrzny
- ***FlyweightFactory***
  - produkuje flyweighty i upewnia się, że będą właściwie współdzielone – na żądanie dostarcza istniejący obiekt lub tworzy nowy, gdy go nie ma
- ***Client***
  - przechowuje referencje do flyweightów
  - oblicza lub przechowuje stany zewnętrzne

## *Zależności:*

- Stan flywieghta może być wewnętrzny (przechowywany w konkretnym flyweighcie) lub zewnętrzny (przechowywany lub obliczany przez klienta i przekazywany w momencie wywołania operacji).
- Klienci nie tworzą flyweightów samodzielnie, lecz dostają je z fabryki, która dba, by były one właściwie współdzielone.
- Nie wszystkie flyweighty muszą być współdzielone.

## *Konsekwencje:*

1. Dodatkowy koszt odnajdywania i przekazywania stanu zewnętrznego, ale w zamian uzyskujemy oszczędność miejsca, zależną od:
   - zmniejszenia ilości instancji, wynikającej ze współdzielenia,
   - wielkości stanu wewnętrznego,
   - tego, czy stan zewnętrzny liczymy, czy przechowujemy.
2. Im więcej (ilościowo i jakościowo) współdzielimy, tym większa oszczędność. Dodatkowa korzyść, gdy jesteśmy w stanie obliczać stan zewnętrzny, zamiast go przechowywać.
3. Często Flyweight jest łączony z Compositem (struktura ze współdzielonymi liśćmi). W takim wypadku nie mogą one przechowywać referencji do ojca – to będzie ich stan zewnętrzny.

## *Implementacja:*

1. Usunięcie stanu zewnętrznego. Możliwość użycia flyweighta zależy od tego, jak łatwo jesteśmy w stanie zidentyfikować i usunąć stan zewnętrzny. Powinniśmy mieć korzyść z przechowywania stanu poza obiektem (najlepiej jeśli jesteśmy go w stanie obliczać). Niekorzystna jest duża różnorodność stanu zewnętrznego.

```
class Znak {
     char znak;
     boolean bold, italic, underline;
     void wypisz() { ... }
     // ...
}
// zamieniamy na:
class Znak {
     char znak;
     void wypisz(boolean bold, boolean it, boolean under) { }
     // ...
}
class BlokTekstu {
     Znak[] tekst;
     boolean bold, italic, underline;
     void wypisz() {
          for(Znak z : tekst)
               z.wypisz(bold, italic, underline);
     }
}
...

BlokTekstu blok = new BlokTekstu(true, false, false, ...);
```

```
// wersja z kompozytem:
class BlokTekstu {
     BlokTekstu[] tekst;
     boolean bold;
     boolean italic;
     boolean underline;
     void wypisz(boolean bold, boolean it, boolean under) {
          // blok tekstu ignoruje otrzymane parametry
          for(BlokTekstu z : tekst) {
               z.wypisz(bold, italic, underline);
          }
     }
}
```

2. Zarządzanie współdzielonymi obiektami. Klient sam ich nie tworzy, lecz dostaje z fabryki na odpowiednio sformułowane żądanie. Jeśli obiektów jest wiele, warto niekiedy troszczyć się o ich usuwanie, gdy przestaną być potrzebne (licznik referencji).

```
class FlyweightFactory {
     private HashMap<Character, Znak> flyweights;
     public FlyweightFactory() {
          flyweights = HashMap<Character, Znak>();
     }
     public Znak getZnak(char z) {
          if(!flyweights.containsKey(z))
               flyweights.put(z, new Znak(z));
          return flyweights.get(z);
     }
}
...
FlyweightFactory ff = new FlyweightFactory();
BlokTekstu tekst = new BlokTekstu(true, false, false);
tekst.add(ff.getZnak('a'));
tekst.add(ff.getZnak('b'));
tekst.add(ff.getZnak('c'));
tekst.add(ff.getZnak('.'));
```

### *Powiązania:*

- Często łączony z Kompozytem (współdzielone liście).
- Obiekty State czy Strategy często warto implementować jako flyweighty.