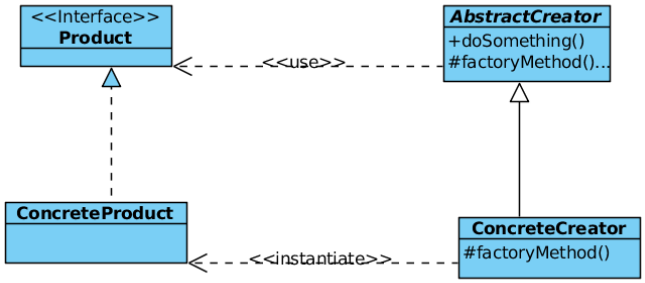


Rysunek 1: Struktura klas przy Metodzie fabrycznej.

1 Metoda fabryczna

Zamiast bezpośrednio tworzyć obiekt, używamy metody, która może być nadpisana przez klasy potomne. Słabsze, ale wymaga mniej dodatkowego kodu niż np AbstractFactory.

Elementy i połączenia :

- Klasa używająca metody fabrycznej.
- Klasa nadrzędna deklarująca metodę fabryczną (czasem, to ta sama klasa, co powyższa).
- Klasy konkretna definiująca metodę fabryczną.

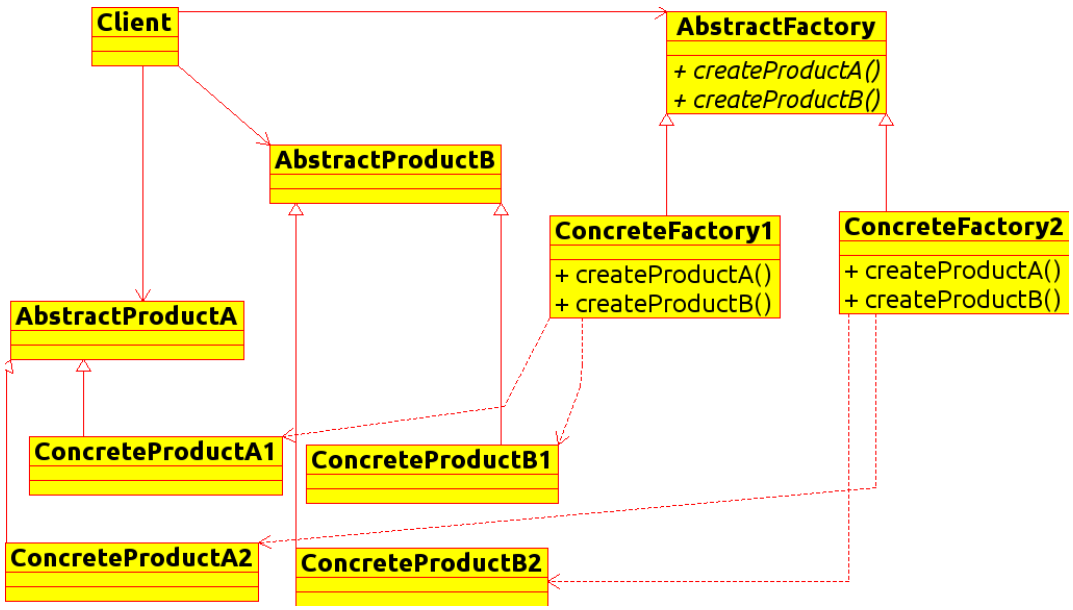
```
class SomeClass{
  someMethod(){
    //zamiast A a = new SpecificA();
    A a = createA();
  }
  A createA(){
    return new SpecificA();
  }
}

  A createA(){
    return new SpecificA();
```

```
13    }
```

```
1  class SomeClass{
     someMethod(){
3      //zamiast A a = new SpecificA();
       A a = createA();
5    }
     A createA(){
7      return new SpecificA();
     }
9  }
```

```
11   A createA(){
       return new SpecificA();
13   }
```

- Produkt może być używany bezpośrednio przez twórcę, lub inną klasę.
- W niektórych przypadkach można wyodrębnić tą metodę w strategię, wtedy otrzymujemy wzorzec fabryki (z tym, że bez "rodziny" obiektów) i możemy podmieniać sposób tworzenia obiektu w czasie wykonania.
- Używany, gdy jest istotne, by klasy pochodne mogły "podmieniać" implementację tworzonych obiektów lub chcemy wyodrębnić funkcjonalność tworzenia tych obiektów, bo może zostać ona zarówno użyta w więcej niż jednym miejscu, jak i potencjalnie jest podatna na zmiany.
- Podobnie jak przy template method, twórca może dostarczyć implementację domyślną albo być abstrakcyjny.
- Brak dostępu do tworzenia obiektów może być realizowany przez np. przez konstruktor prywatny (jeśli nie chcemy rozszerzać podstawowej klasy produktu) lub chroniony, albo wręcz przez produkt będący np. niepubliczną klasą.
- Np. Java `AbstractMap.entrySet();`.



Rysunek 2: Struktura fabryki abstrakcyjnej

2 Fabryka abstrakcyjna

Dostarczamy interfejsu pozwalającego tworzyć rodziny powiązanych obiektów. Szczegóły dostarczanych obiektów (np. konkretne typy) są różne w różnych implementacjach.

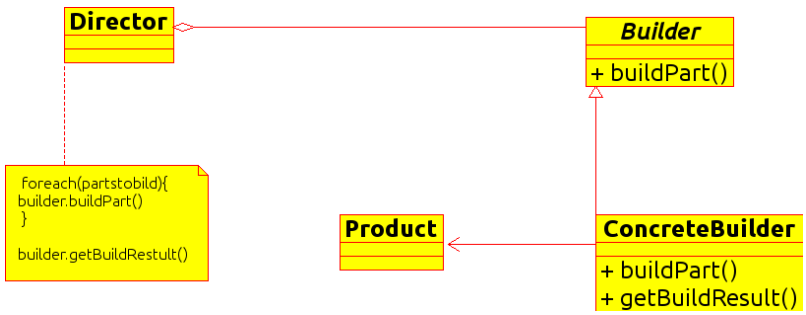
Elementy i połączenia :

- Fabryka abstrakcyjna - dostarcza interfejsu pozwalającego tworzyć rodziny powiązanych obiektów.
- Fabryka konkretna - implementuje wyżej wspomniany interfejs.
- Abstrakcyjny produkt - klasa nadrzędna wszystkich produktów.
- Produkt konkretny.

```
interface XFactory{
  A getA();
  B getB();
}

class SpecificFactory{
```

```
7     A getA(){
          return new SpecificA();
9     }
      B getB(){
11      // np. pobieramy b z konfiguracji, o której użytkownicy nie wiedzą
        return new SpecificB(b);
13    }
  }
15
  class OtherSpecificFactory{
17    A getA(){
          return new SpecificA1();
19    }
      B getB(){
21      return new SpecificB1();
      }
23  }


25
  class FactoryUser{
27   XFactory xfact;
        someMethod(){
29     xfact.getA();
       xfact.getB();
31
       //zamiast np. new SpecificA() ...
33       }
  }
```

- Niekiedy przydatne również kiedy jest tylko jeden obiekt, a nie cała rodzina. Ale przy rodzinie mamy dodatkowo zapewnioną zgodność obiektów.
- Użytkownicy znają tylko abstrakcje - nie muszą w ogóle znać klas konkretnych - dość istotna izolacja.
- Umożliwia podmianę całej rodziny obiektów zmianą właściwie jednej linii kodu (albo łatwe odczytanie z konfiguracji).



Rysunek 3: Struktura budowniczego

Budowniczy Złożony proces tworzenia pewnego obiektu zostaje wydelegowany do innego obiektu.

Elementy i połączenia :

- Budowniczy - definiuje interfejs do budowania pewnego obiektu.
- Konkretny budowniczy - implementuje powyższy interfejs.
- Nadzorca - używa budowniczego do wybudowania produktu.
- Produkt - budowany przez budowniczych.

Używany gdy: Chcemy użyć tego samego ogólnego (nietrywialnego) schematu tworzenia, dla różnych wariantów obiektu budowanego. Wariantowanie może odbywać się poprzez zmianę typu obiektu budowanego zarówno na podtyp, jak i klasę formalnie niezwiązaną, ew. przez zmiany poszczególnych części.

Na przykład:

```
interface ABuilder{
  buildPartA();
  buildPartB();
  buildPartC(OtherClass c);
  X getResult();
}

//Czasem dostarczana klasa z pustymi implementacjami, jako domyślnymi
    do podklas obok albo zamiast interfejsu

class ConcreteXBuilder{
  X beingBuilt;
```

```
12    ...
      buildPartA(){
14      // np. ¡Oblicz a¿
        beingBuilt.setA(a);
16    }
      buildPartB(){
18      //np. pobierz b z konfigruacji
        beingBuilt.setB(b);
20    }
      buildPartC(OtherClass c){
22      beingBuilt.addC(c)
      }
24    X getResult(){
        return beingBuilt;
26    }

28  }


30
    BuildingClass{
32    someMethod(Builder b){
        b.buildPartA();
34      b.buildPartB();
        b.buildPartC(c);
36      b.buildPartC(c2);
        X built = b.getResult();
38      built.doSomething(); //albo return
      }
40  }
```

Inne implementacje budowniczego mogłyby robić w metodach coś znacznie innego, ważny jest schemat budowy widoczny dla kodu "klienta".

Zdegenerowana wersja tego wzorca (tylko jeden produkt, jak np. w StringBuilder w Javie) może być użyta także np. kiedy obiekt końcowy ma być niezmienny (immutable) ale zawiera sporo danych i niewygodnie byłoby przekazywać je na raz.

Zadanie 1. Mamy miejsce podzielone na 9 (3x3) mniejszych pól z granicami między nimi. W strumieniu (dla uproszczenie powiedzmy, zapisanym w pliku) mamy wpisy typu r 3 5 4. Litera oznacza "kolor", pierwsza współrzędna

oznacza rząd, druga kolumnę, trzecia "wysokość". Tych wpisów może być dowolna liczba, na końcu trzeba wyświetlić (albo zapisać w pliku): wspomnianą planszę, w której każde pole ma być pokolorowane takim kolorem, jaki jest "najwyżej". Kolory to mogą być r (czerwony) g (zielony) lub b (niebieski). Kolory granic to też rgb ale oczywiście kolor granicy musi być inny niż kolor sąsiadujących pól (w "rogach, kolor granicy jest nieistotny). Zakładamy, że na każdym polu znajdzie się chociaż jeden kolor.

Nie należy zapamiętywać serii wpisów ani stosu "kolorów".

Proszę zrobić "parser" zdarzeń oraz dwóch budowniczych:

1. Jeden tworzy reprezentację złożoną np. z elementów gui, kolorowych kwadratów w html lub tworzy obrazek (dowolnie). Jest to później wyświetlane przez klienta (albo zapisywane do pliku)

2. Drugi tworzy reprezentację złożoną z Ascii, która wygląda np. tak:

   r r g b b r g g
   r r g b b r g g
   g g g g g g b b
   b b g r r g r r
   b b g r r g r r
   r r r b b b g g
   b b r g g r b b
   b b r g g r b b

   (proszę pamiętać - w "rogach" kolor musi być jednym z 3, ale dowolnym).

Uwagi - "konsumowanie" produktu może odbywać się w całkiem innym kodzie. Klasa "nadrzędna" tworzy "parser" i ustawia mu "budowniczego" reprezentacji, jednak sama jest odpowiedzialna za zrobieniem coś z "produktem" (np. zapisanie pliku, wyświetlenie).