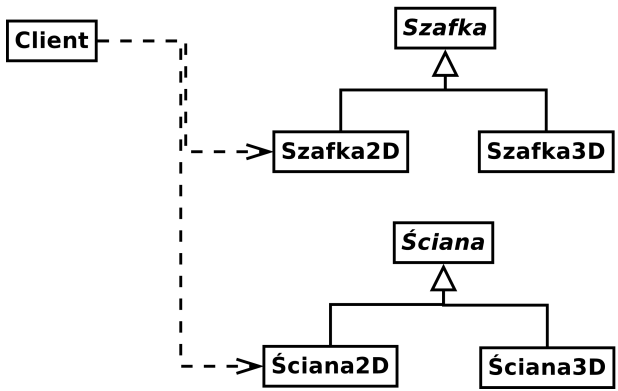
Abstract Factory (fabryka abstrakcyjna)

Cel:

Zapewnienie interfejsu do tworzenia rodzin powiązanych obiektów bez specyfikacji konkretnej klasy.

Przykład:



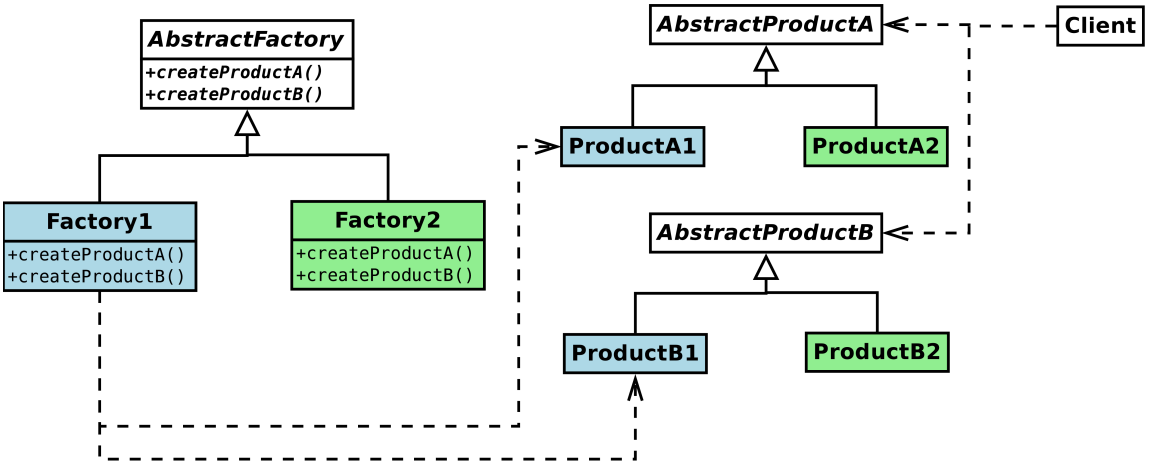
- Aplikacja do ustawiania mebli: osobne rodziny produktów: klas 2D i 3D

```
abstract class Szafka { /* ... */ }
class Szafka2D extends Szafka { /* ... */ }
class Szafka3D extends Szafka { /* ... */ }

abstract class Ściana { /* ... */ }
class Ściana2D extends Ściana { /* ... */ }
class Ściana3D extends Ściana { /* ... */ }

class Client {
     public static void main(String[] args) {
          // ...
          if(tryb == Tryb.2D) {
               model.add(new Ściana2D(...));
               // ...
          } else if (tryb == Tryb.3D) {
               model.add(new Ściana3D(...));
               // ...
          }
          // ...
     }
}
```

Struktura:



Składniki:

- AbstractFactory – deklaruje interfejs tworzenia abstrakcyjnych produktów
- Factory1, Factory2 – implementują operacje tworzenia konkretnego produktu
- AbstractProduct – deklaruje interfejs rodzaju produkowanych obiektów
- Product* – implementują interfejs AbstractProduct, definiują rodzaj produktu
- **Client** – używa wyłącznie interfejsu **AbstractFactory** i **AbstractProduct**

Zależności:

Zazwyczaj w trakcie działania programu tworzona jest pojedyncza instancja ConcreteFactory. Tworzy ona obiekty produktów określonego rodzaju. Aby tworzyć inny typ obiektów, należy użyć innej fabryki.

AbstractFactory oddelegowuje zadanie tworzenia produktów do potomnych klas konkretnych fabryk.

```
abstract class AbstractFactory {
     public abstract AbstractProductA createProductA();
     public abstract AbstractProductB createProductB();
}
class Factory1 extends AbstractFactory {
     public AbstractProductA createProductA() {
          return new ProductA1(); }
     public AbstractProductB createProductB() {
          return new ProductB1(); }
}
class Factory2 extends AbstractFactory {
     public AbstractProductA createProductA() {
          return new ProductA2(); }
     public AbstractProductB createProductB() {
          return new ProductB2(); }
}
```

```
class Client {
     public static void main(String[] args) {
          AbstractFactory factory;
          if(/* ... */) factory = new Factory1();
          else          factory = new Factory2();
          // ...
          AbstractProductA pa = factory.createProductA();
          AbstractProductB pb = factory.createProductB();
          model.addA(factory.createProductA());
          model.addB(factory.createProductB());
          // ...
     }
}
```

Zastosowanie:

- Gdy system nie powinien zależeć od tego w jaki sposób jego produkty są tworzone, składane i reprezentowane
- Gdy system może być skonfigurowany z jedną z kilku rodzin produktów
- Gdy produkty są tak zaprojektowane, by używać ich razem (czyli tylko produkty z jednej rodziny)
- Gdy chcemy dostarczyć bibliotekę produktów i udostępniać tylko ich interfejsy, nie implementację

Konsekwencje:

1. izoluje konkretne klasy – ich nazwy nie są widoczne w kodzie klienta, gdyż ich tworzeniem zajmuje się konkretna fabryka;
2. ułatwia wymianę rodziny produktów – aby użyć innej rodziny wymieniamy tylko konkretną fabrykę, która przeważnie pojawia się tylko w jednym miejscu aplikacji;
3. wspomaga spójność między produktami – niekiedy produkty danej rodziny są pomyślane tak, by używać ich razem (czyli aplikacja powinna używać tylko produktów jednej rodziny)
4. dostarczenie nowego rodzaju produktów jest trudne – musimy rozszerzyć interfejs fabryki, czyli zmienić AbstractFactory i wszystkie potomne;

Implementacja:

1. fabryka jako Singleton – zazwyczaj potrzebujemy tylko jednej instancji fabryki
2. tworzenie produktu – abstrakcyjna fabryka deklaruje tylko interfejs do tworzenia produktów, ich stworzenie jest zadaniem konkretnych fabryk – najczęściej robi się to definiując factory method dla każdego produktu, co wymaga nowej fabryki dla każdej rodziny produktów;
3. definiowanie rozszerzalnych fabryk – aby nie mieć odrębnej operacji dla każdego produktu można użyć jednej metody z parametrem oznaczającym rodzaj produktu; jest to używane w fabrykach typu prototyp; produkty muszą mieć wspólny interfejs, albo klient będzie musiał je rzutować w dół;

```
abstract class AbstractFactory {
     public enum Type { szafka, ściana };
     public public Object create(Type type);
}

class Factory2D extends AbstractFactory {
     public Object create(Type type) {
          switch(type) {
               case szafka: return new Szafka2D();
               case ściana: return new Ściana2D();
          }
          throw new IllegalArgumentException();
     }
}

//... Client
AbstractFactory factory = new Factory2D();
Szafka sz =
(Szafka)factory.create(AbstractFactory.Type.szafka);
```

Inny przykład:

```
interface Window { … }
interface Label { … }
interface Button { … }
//Abstract Factory
inteface GUIEnv {
   Window getWindow();
   Label getLabel(String text);
   Button getButton(String text);
}
//Windows Factory
class WindowsGUI implements GUIEnv{
   public Window getWindow() {
     return new Windows.Frame();
   }
   public Label getLabel(String text) {
     return new Windows.TextBlock(text);
   }
   public Button getButton(String text) {
     return new Windows.PressButton(text);
   }
}
```

```
//My Factory
class MyGUI implements GUIEnv{
   public Window getWindow() {
     return new MyDialogWindow();
   }
   public Label getLabel(String text) {
     return new MyIcon(new Text(text));
   }
   public Button getButton(String text) {
     return new MyButton(new MyIcon(new Text(text)));
   }
}
// Client
     GUIEnv factory = new MyGUI();
     Window window = factory.getWindow()
     window.add(factory.getLabel("Hello"));
     window.add(factory.getButton("OK"));
     window.add(factory.getButton("Anuluj"));
```

Powiązania:

- AbstractFactory – przeważnie implementowana z użyciem Factory Method, może korzystać z Prototypów, przeważnie jest Singletonem