Strategy (strategia)

Cel:

Definiuje i wydziela rodzinę wymiennych algorytmów. Strategia pozwala zmieniać się im niezależnie od używającego ich klienta.

Przykład:

```
enum Renderer { DirectX, OpenGL };
class Geometry3D {
     Renderer renderer;
     public void render() {
          if(renderer == Renderer.DirectX) {
               System.out.println("using DirectX"); /* ... */
          }
          else if(renderer == Renderer.OpenGL ) {
               System.out.println("using OpenGL"); /* ... */
          }
          else
               throw new UnsupportedOperationException();
     }
}
```

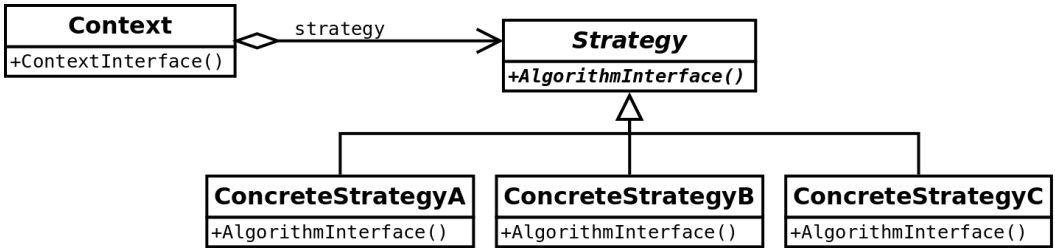
```
interface RenderStrategy {
     void render(Geometry3D geometry);
}
class RenderStrategyDirectX implements RenderStrategy {
     public void render(Geometry3D geometry) {
          System.out.println("using DirectX");
          // ...
     }
}
class RenderStrategyOpenGL implements RenderStrategy {
     public void render(Geometry3D geometry) {
          System.out.println("using OpenGL");
          // ...
     }
}
class Geometry3D {
     private RenderStrategy strategy;
     public Geometry3D(RenderStrategy strategy) {
          this.strategy = strategy;
     }
     public void render() {
          strategy.render(this);
     }
}
```

```
class Client {
     public static void main() {
          Geometry3D g3d;
          g3d = new Geometry3D(new RenderStrategyDirectX());
          g3d.render();
     }
}
```

Zastosowanie:

- Gdy wiele powiązanych klas różni się jedynie zachowaniem. Strategia pozwala skonfigurować jedną klasę jednym z kilku zachowań.
- Gdy potrzebujemy jednego z wariantów jakiegoś algorytmu, różniących się np. kosztami czasu i zajętości pamięci. Strategia może być wykorzystana, gdy te warianty są zaimplementowane jako hierarchia algorytmów.
- Gdy algorytm używa danych, o których klient nie powinien wiedzieć. Strategia pozwala ukryć złożone, specyficzne dla algorytmu struktury danych.
- Gdy klasa definiuje wiele różnych zachowań – obecnych w kodzie jako instrukcje warunkowe. Lepiej jest przenieść je do odrębnych klas Strategii.

Struktura:



Składniki:

- **Strategy**
  - deklaruje i udostępnia wspólny interfejs wszystkich oferowanych algorytmów
- **ConcreteStrategies**
  - implementuje algorytm używając interfejsu Strategii
- **Context**
  - jest konfigurowany obiektem konkretnej Strategii
  - przechowuje referencję do Strategii i wywołuje jej metody
  - może definiować interfejs, przez który Strategia sięgnie po dane

Zależności:

- Strategy i Context wspólnie implementują wybrany algorytm. Dane, których wymaga algorytm: Context może przekazywać je w momencie wywołania lub przekazać Strategii swoją referencję, aby sama po nie sięgała.

```
interface RenderStrategy {
     void renderTriangle(Point3D[] vertices,
                         Vector3D[] normals, Image texture);
}
class Geometry3D {
     ...
     public void render() {
          strategy.renderTriangle(...);
     }
}
```

- Kontekst odsyła żądania klienta do Strategii. Klient zazwyczaj stworzy i przekaże Konkretną Strategię do kontekstu, później komunikując się wyłącznie z nim. Przeważnie ma też kilka Strategii do wyboru.

Konsekwencje:

1. Rodzina pokrewnych algorytmów – definiowana przez hierarchię strategii, wykorzystywana przez kontekst. Dziedziczenie pomaga wyodrębnić elementy wspólne poszczególnych algorytmów.
2. Alternatywa dziedziczenia – zamiast dziedziczyć z Kontekstu, by zmienić jakieś zachowanie, można zamknąć je w odrębnej klasie, co pozwoli m. in. wymieniać dynamicznie i rozwijać niezależnie od kontekstu.

```
abstracte class Enemy {
     public abstract void move();
}
class AggressiveEnemy extends Enemy {
     public void move() { ... }
}
class DefensiveEnemy extends Enemy {
     public void move() { ... }
}
```

```
class Enemy {
     Strategy strategy;
     public void move() { strategy.makeMove(...); }
}
class AggressiveStrategy implements Strategy {
     public void makeMove(...) { ... }
}
class DefensiveStrategy implements Strategy {
     public void makeMove(...) { ... }
}
```

3. Zamiast wyrażeń warunkowych – które miałyby służyć wybraniu tego, czy innego zachowania.

```
class Enemy {
     public void move() {
          if (...) {
               ...
          }
          else if (...) {
               ...
          } else ...
}
```

4. Wybór implementacji – strategie mogą udostępniać różne implementacje tego samego zachowania, klient może wybrać opcje różniące się zajętością pamięci, czy czasem wykonania.
5. Potencjalny minus – ujawnienie implementacji. Klient wybiera strategię, a zatem powinien wiedzieć czym się one różnią (różnice w zachowaniach powinny mieć dla niego znaczenie).
6. Komunikacja między strategią a kontekstem – algorytm będzie wymagał danych wejściowych, ale algorytm prosty może wymagać ich mniej; skoro interfejs wszystkich strategii jest wspólny, będziemy mu przekazywać argumenty, których nie wykorzysta. Alternatywą jest ściślej związać kontekst i strategię.
7. Większa liczba obiektów – można temu zaradzić implementując strategie jako współdzielone obiekty bezstanowe (*Flyweight*).

Implementacja:

1. Przekazywanie danych Strategii:

```
class EnemyAI {
     Strategy strategy;
     public void move() {
          decision = strategy.makeDecision(
               factories, army, resources, money);
          // perform action
     }
}
```

2. Przekazywanie referencji na Kontekst:

```
class EnemyAI {
     Strategy strategy;
     public void move() {
          decision = strategy.makeDecision(this);
          // perform action
     }
}
```

3. Strategia, jako opcja – kontekst może oferować pewnie domyślne działanie, dzięki czemu może obyć się bez strategii. Pozwala to klientowi nie troszczyć się o strategie, jeśli nie ma szczególnej potrzeby.

Powiązania:

- Flyweight – Strategia to dobry kandydat na flyweighta