State (stan)

Cel:

Pozwala obiektowi na zmianę zachowania w momencie zmiany stanu wewnętrznego – działa to jak dynamiczna zmiana klasy obiektu.

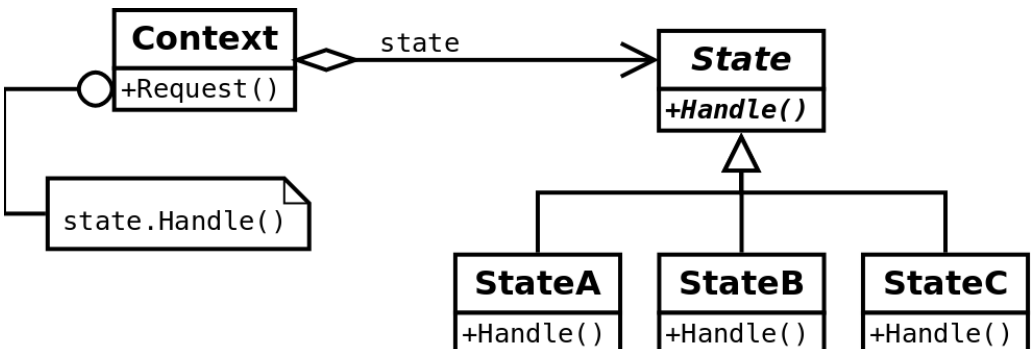
Przykład:

```
class Enemy {
     private int health;
     public void move() {
          if(health < 5)
               // uciekaj
          else if(distance(hero) < 10)
               // atakuj
          else
               // rozglądaj się
     }
}
```

Zastosowanie:

- Gdy zachowanie obiektu zależy od jego stanu i powinno zmieniać się w trakcie działania, zależnie od tego stanu.
- Gdy w kodzie pojawia się dużo złożonych instrukcji warunkowych (być może w wielu operacjach) różnicujących zachowanie na podstawie stanu obiektu. Wzorzec State każdą z gałęzi warunku reprezentuje osobną klasą.

Struktura:



Składniki:

- Context:
  - definiuje interfejs, którym posłuży się klient
  - przechowuje instancję konkretnego Stanu (stan aktualny)
- State:
  - definiuje interfejs zachowań związanych z aktualnym stanem Kontekstu
- klasy ConcreteState:
  - każda z klas implementuje zachowanie związane z pewnym konkretnym stanem Kontekstu

Zależności:

- Kontekst oddelegowuje żądanie do aktualnego obiektu Stanu.
- Może też przekazać siebie jako argument, aby Stan mógł sięgać do danych kontekstu.
- Kontekst to podstawowy interfejs dla klienta – może on skonfigurować kontekst stanem początkowym, ale później już nie ma do niego dostępu.
- Zarówno Kontekst, jak i dany Stan może decydować o zmianie stanu na inny.

```
class Enemy {
	private State state = new StateNormal(); // stan początkowy
	public void move() {
		state.makeMove();
		// sprawdź, czy należy zmienić stan
	}
}
interface State {
	void makeMove();
}
class StateRanny implements State {
	public void makeMove() {
		// uciekaj
	}
}
class StatePogoń implements State {
	public void makeMove() {
		// atakuj
	}
}
class StateNormal implements State{
	public void makeMove() {
		// rozglądaj się
	}
}
```

Konsekwencje:

1. Zachowania związane ze stanami przeniesione do odrębnych klas – dzięki temu dodanie nowego stanu sprowadza się do napisania kolejnej klasy. Jest to prostsze w utrzymaniu i zrozumieniu, niż przerośnięte procedury z instrukcjami warunkowymi. Problemem może być rozbicie definicji operacji na wiele stanów i duża liczba klas.
2. Zmiana stanu bardziej czytelna i jawna (nie polega na zmianie wartości pewnej zmiennej lub zmiennych, których będą gdzieś później sprawdzane) – a zatem prostsza ochrona przed nieprzewidzianymi stanami, etc.
3. Współdzielone obiekty stanów – jeśli jest wiele kontekstów, a Stany nie mają zmiennych składowych (stanu wewnętrznego).

Implementacja:

1. Kto determinuje zmianę stanu? – jeśli kryterium jest ustalone, może zająć się tym Kontekst. Bardziej elastyczne rozwiązanie: powierzyć to Stanowi (każdy Stan będzie decydował o swoim następniku). Wymaga to interfejsu przez który będzie mógł powiadamiać kontekst o zmianach. Dzięki temu łatwiej można modyfikować i rozbudowywać diagram stanów, ale stany muszą wiedzieć o sobie nawzajem.

```
class Enemy {
     private State state = new StateNormal();
     public void move() {
          state.makeMove();
          // sprawdź, czy należy zmienić stan
          if(health < 5)
               if(!state instanceof StateRanny)
                    state = new StateRanny();
          else if(distance(hero) < 10)
               if(!state instanceof StatePogoń)
                    state = new StatePogoń();
          else
               if(!state instanceof StateNormal)
                    state = new StateNormal();
     }
}
```

```
class Enemy {
     private State state = new StateNormal();
     public void move() {
          state = state.makeMove(this);
     }
}
interface State {
     State makeMove(Enemy en);
}
class StateNormal implements State {
     public State makeMove(Enemy en) {
          // rozglądaj się ...
          if(en.health < 5)
               return new StateRanny();
          if(en.distance(hero) < 10)
               return new StatePogoń();
          return this;
     }
}
```

2. Do definiowania przejść można użyć tabel – dla każdego stanu tablica mapująca możliwe wejścia w następujące po nich stany. Korzyść to regularność tego rozwiązania, ale: narzut związany z odwołaniami do tablicy, logika przejść jest niekiedy mniej czytelna, trudniej dodawać akcje wykonywane w momencie przejść.

3. Tworzenie i usuwanie Stanów:
   - tworzyć, gdy są potrzebne i usuwać gdy nie są (lepsze, gdy nie znamy stanów z góry i zmieniają się nieregularnie, być może przechowują dużo danych – dzięki temu nie tworzymy tych, które nie będą potrzebne)
   - tworzyć wszystkie z góry i nie usuwać nigdy (dobre, gdy stany zmieniają się często; przechowywaniem nieużywanych stanów może zająć się kontekst)

```
class Enemy {
     public static final State stRanny = new StateRanny();
     public static final State stPogoń = new StatePogoń();
     public static final State stNormal = new StateNormal();
     // ...
}
```

Powiązania:

- Obiekty stanu mogą być współdzielone – Flyweight
- Stan może być też Singletonem