Template method (metoda szablonowa)

Cel:

Definiuje szkielet algorytmu przy pomocy operacji podstawowych. Konkretyzacja poszczególnych kroków składowych pozostawiona klasom potomnym – mogą być one zmieniane bez naruszania ogólnej struktury.

Przykład:

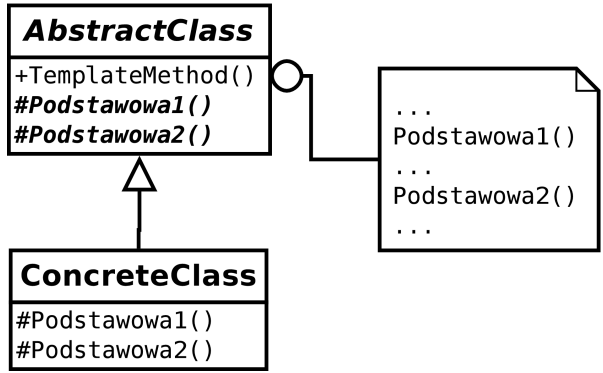
```
abstract class Game {
     public final void run() {
          initialize();
          while(!gameOver()) {
               makeMoves();
               paintScreen();
          }
          onEnd();
     }
     ...
}
```

```
class MyGame extends Game {
     ...
     ...
     ...
     protected void initialize() {
          System.out.println("Przygotowanie do gry");
     }
     protected boolean gameOver() {
          return player.lives <= 0;
     }
     protected void makeMoves() {
          player.move();
          for(Monster m : monster)
               m.move();
     }
     protected void paintScreen() {
          world.draw();
     }
     protected void onEnd() {
          System.out.println("Game Over");
     }
}
```

Zastosowanie:

- Aby niezmienne części algorytmu zaimplementować raz, natomiast klasom potomnym zostawić implementację zachowań, które mogą się zmieniać.
- Aby podobne zachowanie kilku klas przenieść do jednej klasy bazowej (w celu uniknięcia powtarzania kodu). Odnajdujemy różnice w zachowaniu poszczególnych klas, przenosimy je do osobnych metod, a następnie zachowanie wspólne umieszczamy w *metodzie szablonowej*, wywołującej tamte.
- Aby kontrolować, co może być rozszerzane przez dziedziczenie. Szkielet algorytmu pozostaje niezmienny, klasom potomnym pozwalamy dodawać swoją funkcjonalność jedynie w określonym punktach algorytmu (*hook operations*).

Struktura:



Składniki:

- **AbstractClass** – klasa abstrakcyjna
  - deklaruje abstrakcyjne **operacje podstawowe**
  - implementuje **metodę szablonową**, definiującą szkielet algorytmu, która korzysta (między innymi) z operacji podstawowych
- **ConcreteClass** – klasa konkretna
  - implementuje **operacje podstawowe**, definiując specyficzne dla danej klasy kroki algorytmu

Zależności:

- Konkretna klasa pozostawia klasie abstrakcyjnej definicję niezmiennych elementów algorytmu

```
class abstract AbstractClass {
     public void algorytm() {
          int ile = kroki();
          double suma = 0.0;
          for(int i = 0; i < ile; ++i)
               suma += krok(i);
          krokOstatni(suma);
     }
     abstract int kroki();
     abstract double krok(int i);
     abstract void krokOstatni(double i);
}
class ConcreteClass extends AbstractClass{
     int kroki() { return (int)(100*Math.random()); }
     double krok(int i) { return i; }
     void krokOstatni(double d) { System.out.println(d); }
}
```

Konsekwencje:

Podstawowa technika wielokrotnego wykorzystania kodu. Ważne zwłaszcza w bibliotekach, gdyż jest sposobem na parametryzację zachowań klas bibliotecznych.

Prowadzi do odwrotnej interakcji (zwanej czasem „zasadą Hollywoodu” – „Don’t call us, we’ll call you”) – klasa bazowa wywołuje metody z klas potomnych, nie na odwrót.

```
// klasycznie:
abstract class Bazowa {
     public void funkcja() {
          // bazowe działanie
     }
}
class Potomna extends Bazowa {
     public void funkcja() {
          super.funkcja();
          // dodatkowe działanie
     }
}
```

```
// z metodą szablonową:
abstract class Bazowa {
     public void funkcjaSzablonowa() {
          // bazowe działanie
          funkcjaDodatkowa();
     }
     protected void funkcjaDodatkowa() { }
}
class Potomna extends Bazowa {
     protected void funkcjaDodatkowa() {
          // dodatkowe działanie
     }
}
```

Operacje podstawowe są wywoływane przez *metodę szablonową*, a definiowane przez klasę potomną. Można podzielić je na dwa rodzaje: te, które klasa potomna musi nadpisać (konkretyzują działanie algorytmu) i te, które może nadpisać (stanowią miejsce wstawienia opcjonalnych działań). Ten drugi typ metod (*hook operations*) powinien zawierać jakieś działanie domyślne (często: „nic nie rób”).

```
abstract class Bazowa {
     public final void algorithm() {
          beforeAlgorithm();
          // ...
          // przygotowanie środowiska
          // ...
          result = doAlgorithm(data);
          // ...
          // kończenie działań
          // ...
          afterAlgorithm(result);
     }
     protected abstract int doAlgorithm(int data);
     protected void beforeAlgorithm() { }
     protected void afterAlgorithm(int result) { }
}
```

```
abstract class Story {
     public final void tell() {
          System.out.println(bohater() + " wyrusza na " +
               "wyprawę, aby zdobyć " + cel() + ". Na jego" +
               " drodze staje " + wróg() + ". " + bohater() +
               " jest bliski przegranej, ale pomaga mu " +
               przyjaciel() + ". Dzięki wspólnemu wysiłkowi "
               + wróg() + " zostaje pokonany i " + bohater()
               + " zdobywa " + cel() + ".");
     }
     public abstract String bohater();
     public abstract String cel();
     public abstract String wróg();
     public abstract String przyjaciel();
}
class MyStory extends Story {
     public String bohater() { return "Książę"; }
     public String cel() { return "magiczny miecz"; }
     public String wróg() { return "smok"; }
     public String przyjaciel() { return "wiedźma"; }
}
...
     new MyStory().tell()
```

Implementacja:

1. Modyfikatory dostępu. Operacje podstawowe to metody chronione (*protected*) – będą mogły być wołane jedynie przez metodę szablonową. Mogą być abstrakcyjne (*abstract*), wtedy klasa potomna musi dostarczyć ich implementację. Metoda szablonowa nie powinna być przesłaniana, zatem jest finalna (*final*).

```
abstract class Bazowa {
     public final void metodaSzablonowa() {
          krok1();
          krok2();
     }
     protected abstract void krok1();
     protected void krok2() {
          // domyślna implementacja
     }
}
```

2. Dobrze jest minimalizować liczbę operacji podstawowych – a w każdym razie tych, które klasa potomna musi przesłaniać.
3. Wybór konwencji nazewniczej – dodawanie prefiksów do nazw operacji podstawowych i metod do przesłonięcie.

Powiązania:

- Factory Method – są często wywoływane właśnie przez *metody szablonowe.*
- Strategy – *template methods* wykorzystują dziedziczenie, by zróżnicować fragmenty algorytmu, Strategia wykorzystuje delegację, by zmienić cały algorytm.