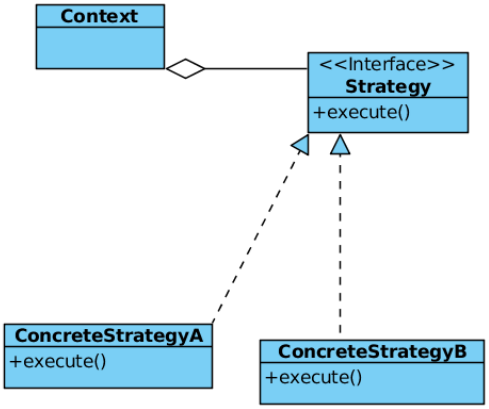
1 Strategia

W skrócie: Zbiór algorytmów realizujących ten sam ogólny cel jest mapowany na rodzinę obiektów - obiekt zawiera jeden (niekiedy więcej powiązanych) konkretny algorytm np. jako metodę. Dzięki "wyciągnięciu" wspólnej abstrakcji algorytmy są nawzajem zastępowalne i mogą się zmieniać zasadniczo niezależnie od kontekstu użycia.

Elementy i połączenia :

- Klient - żąda usługi od *kontekstu*, czasem wybiera strategię (w jakiś sposób, niekoniecznie przekazując obiekt do kontekstu).
- Kontekst - dostarcza jakiejś usługi *klientowi*, niektóre (być może wszystkie) operacje potrzebne do wykonania tej usługi deleguje do obiektu *strategii*, którą przechowuje w zmiennej typu abstrakcyjnego. Implementacja tej operacji zależy od konkretnej klasy *strategii*.
- Abstrakcyjna strategia/interfejs strategii - definiuje interfejs strategii potrzebny do wywołania operacji.
- Konkretna strategia - implementuje konkretny algorytm w odpowiednim interfejsie.

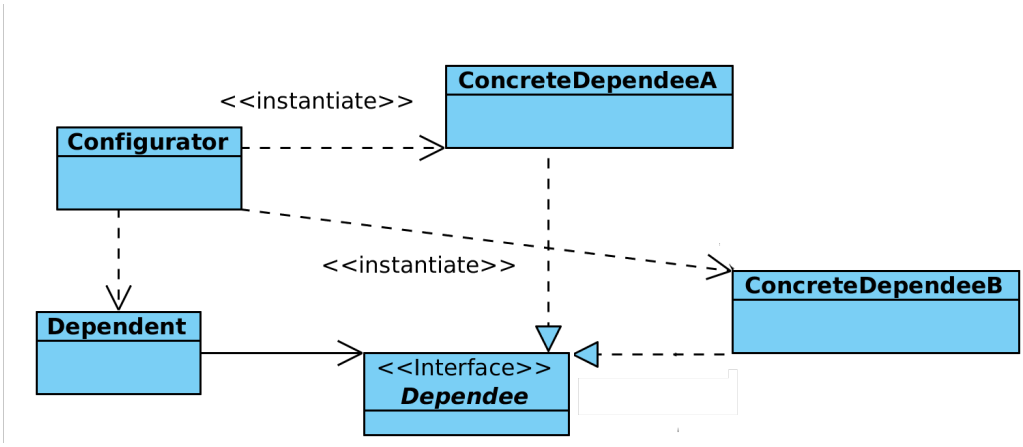
```
interface AbstractStrategy{
    runAlgorithm(AlgorithmInput input);
}
class StrategyA implements AbstractStrategy{
  runAlgorithm(AlgorithmInput input){
  ...
  }
}

class StrategyB implements AbstractStrategy{
  runAlgorithm(AlgorithmInput input){
  ...
  }
}

class Context{
      AbstractStrategy strategy;
```




```
        public setStrategy(AbstractStrategy strategy){
19  this.strategy = strategy;
        }
21      doWork(){
    ...   // czesc algorytmu przed łwywoaniem strategii
23    strategy.runAlgorithm();
    ... // czesc algorytmu po łwywoaniu strategii
25      }
}
```

Uwagi :

- Przekazywanie parametrów można zaimplementować przez argumenty metody, ale też można ustawić odpowiednie pola w obiekcie lub nawet przekazać kontekst przez referencję, ale to niestety uzależnia strategię od kontekstu.
- Jeżeli algorytmy mają ten sam ogólny cel, ale wymagają różnych danych, to jest to pewien problem. Można np. zdefiniować wiele interfejsów i "pytać" o nie, ale to zwiększa powiązania.
- Kontekst nie powinien zależeć od konkretnej strategii (choć wyjątki też istnieją). W takim razie nie powinien jej tworzyć: strategię można otrzymywać przez ustawienie pola z zewnątrz (Dependency Injection) albo przez tzw. fabrykę (ew. metodę fabryczną).



Rysunek 1: Struktura typowego Dependency Injection (

2 Dependency Injection

Prosty wzorzec oddzielenia używania obiektu od jego tworzenia - obiekt (klient) zależy tylko od abstrakcji, nie tworzy konkretnego obiektu, który jest ustawiany z zewnątrz ale tworzy ktoś inny (np. zarządca).

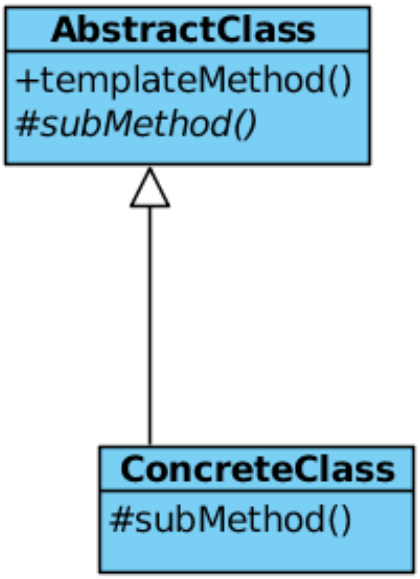
Przykładowy kod:

```
class Director{
  constructClient(){
    x;
    Client c = new Client();
    c.setDependencyX(x);
  }
}

}
class Client{
      x;
      setDependencyX(x){
this.x = X; //Zamiast x= new ConcreteX(); w konstruktorze
      }
      someMethod(){
x.doSomething();
      }
}
```

Odrobinę bardziej specyficzny

```
class Director{

  constructClient(){
    ConnectionStrategy cs = new WebConnection();
    Client c = new Client();
    c.setConnectionStrategy(cs);
  }
}

}
class Client{
      ConnectionStrategy cs;
      setConnectioStrategy(cs){
this.cs = cs;
      }
      connect(){
cs.connect(presetAddress);
      }
}
```

- Może trochę mniej niż "wzorzec", ale warto wiedzieć.
- Można ustawiać przede wszystkim w konstruktorze i przez setter. Setter może być wydzielony do osobnego interfejsu, co wydziela tą cechę i pozwala na stosunkowo łatwe odróżnienie obiektów potrzebujących danego "wstrzyknięcia". W bardzo szerokim rozumieniu odnosi się to w zasadzie do jakiejkolwiek implementacji zasady dependency inversion.
- Pozwala nie zależeć obiektom od klas konkretnych, tylko od abstrakcji. Zależności od klas konkretnych można przenieść do zarządcy, klientów (lub fabryk).
- Pozwala wynieść specjalny przypadek "setup" systemu do innej, ograniczonej hierarchii klas, zamiast trzymać go w tych obługujących "normalne działanie" (przykład Inversion of Control).
- Ustawianie może być sterowanie kodem zarządcy albo np. plikami konfiguracyjnymi. Wtedy zaletą jest, że plik konfiguracyjny przetwarzany jest w (bardziej) określonym miejscu.
- Przydatny głównie do inicjalizacji i to nawet niekoniecznie do pełnej inicjalizacji, a np. tylko do podłączenia obiektu to struktur *umożliwia-*



Rysunek 2: Struktura template method.

jących pełną inicjalizację. Np. ustawianie Fabryk (o których później), ew. stałych (bo np. ustawionych w konfiguracji) Strategii itp.

- Ogólnie przydatny przede wszystkim do ustawiania czegokolwiek, co obiekt ma zamiar przechowywać (skojarzenie a nie zwykła zależność) lub do ustawiania producentów obiektów, od których obiekt zależy.

3 Szablon metody – Template Method

Definiujemy ogólny zarys algorytmu pozostawiając implementację szczegółów klasom potomnym.

Elementy i ich połączenia

- Klasa bazowa definiuję metodę, która z kolei używa metod pozostawionych klasom potomnym do nadpisania.
- Klasy potomne definiują te metody, zmieniając szczegóły zachowania.

```
abstract class AClass{
        doBigThing(){
   ...
   doSomething();
   doOtherThing();
   ...
        }
        // Ponizsze metody nie musza byc abstrakcyjne,
    moga miec implementacje domyslne
        abstract doSomething();
        abstract doOtherThing();
}

class DerivedClass1 extends AClass{
     doSomething(){
   //1 way
     }
     doOtherThing(){
   //1 way
     }
}

class DerivedClass2 extends AClass{
     doSomething(){
   //way 2
     }
     doOtherThing(){
   //way2
     }
}
```

Przykład z biblioteki Javy:

```
public abstract class AbstractList<E> extends
   AbstractCollection<E> implements List<E> {
 //...
      public boolean addAll(int index, Collection<?
    extends E> c) {
          rangeCheckForAdd(index);
          boolean modified = false;
          for (E e : c) {
```

```
                add(index++, e); //Domyslnie rzuca
    UnsupportedOperationException, get jest abstrakcyjna
8               modified = true;
            }
10          return modified;
        }
12  //...
```

- Metody do nadpisania mogą być abstrakcyjne, albo mieć domyślne implementacje, nawet puste, o ile to ma sens. Wtedy klasy potomne nie muszą nadpisywać metod, które i tak by nic nie robiły.
- Jeżeli jest więcej niż jedna metoda, to algorytmy są powiązane. Jeżeli nie są powiązane, to trzeba raczej użyć strategii, aby uniknąć kombinatorycznej eksplozji wariantów.

Strategia, Stan, i Most mają nieco podobne struktury, ale inne cele i nieco inne wzorce interakcji (zwł most).

Zadanie 1. Napisać program, który:

1. Wczyta z pliku podanego jako parametr uruchomienia serię liczb.
2. Posortuje je.
3. Zapisze w pliku (podanym jako parametr uruchomienia).

Zmienne zachowanie:

1. Format odczytu liczb: w jednej linii oddzielone spacjami lub w prostym "niby-XML" – każda liczba w elemencie <value> </value>. Użytkownik może format wybrać. Jeśli wybierze błędnie, to błąd otrzyma.
2. Format zapisu - podawany przez użytkownika (jeden z dwóch wyżej wymienionych).
3. Użyte sortowanie - dwa dowolne rodzaje, same algorytmy mogą być wzięte z bibliotek systemowych (przynajmniej 1). Sortowanie jest wybierane przez użytkownika. Żeby ten wybór miał choć trochę sensu, przynajmniej jeden algorytm powinien mieć złożoność liniową dla danych posortowanych (np. sortowanie przez wstawianie).

Proszę użyć poznanych wzorców.