# Command (action, transaction, polecenie)

## *Cel:*

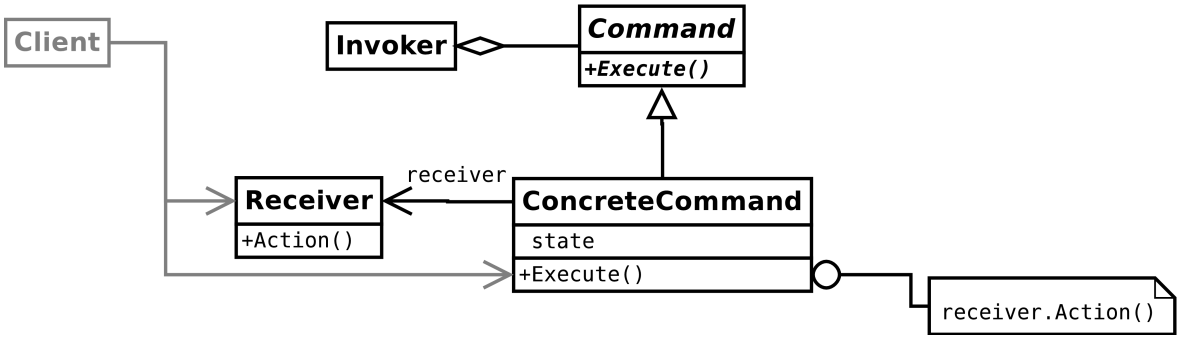
Obiekt funkcyjny. Pozwala operować żądaniem jako obiektem – wysyłać jako parametr, buforować, kolejkować, składować (dzienniki lub undo).

## *Przykład:*

```
class Application {
     public Application() {
          Button[] toolbar = { new Button("New"),
                               new Button("Open"),
                               new Button("Save"), ... };
     }
     public void action(Button btn) {
          if(btn.action == "New") {
               document = newDocument();
          }
          else ...
     }
}
```

```
class Application {
     public Application() {
          Button[] toolbar = {
               new Button(new AkcjaNewDocument()),
               new Button(new AkcjaOpenDocument()),
               new Button(new AkcjaSaveDocument()), ... };
     }
     public void action(Button btn) {
          btn.action.execute();
     }
}
class AkcjaNewDocument {
     public void execute() {
          document = newDocument();
     }
}
class AkcjaSaveDocument {
     public void execute() {
          document.save();
     }
}
...
```

## *Struktura:*




## *Składniki:*

- ***Command*** – deklaruje interfejs wywołania operacji
- ***ConcreteCommand*** – definiuje powiązanie między odbiorcą a poleceniem i implementuje metodę Execute (wywołując tam odpowiednie operacje odbiorcy)
- ***Client*** (np. aplikacja) – tworzy obiekt ConcreteCommand i ustawia jego odbiorcę (Receiver)
- ***Invoker*** (np. menu aplikacji) – wywołuje operację przy pomocy Commanda
- ***Receiver*** – wie, jak wykonać operację (odbiorca operacji)

## *Zależności:*

- Klient tworzy obiekt konkretnego polecenia i określa jego odbiorcę.
- Invoker przechowuje obiekty poleceń.
- Invoker wywołuje polecenie (metoda Execute z obiektu polecenia), w wypadku operacji cofalnych obiekt polecenia zachowuje stan sprzed wywołania metody
- Obiekt polecenia wywołuje operacje swego odbiorcy, wypełniając żądanie.

```
interface Command {
     void execute();
}
class ConcreteCommand implements Command {
     public void execute() {
          // ...
     }
}
public class Client {
     public static void main(...) {
          ConcreteCommand cmd = new ConcreteCommand();
          // ...
          cmd.execute();
     }
}
```

## *Zastosowanie:*

- By parametryzować obiekty akcją, jaką mają wykonać (np. Menu). W języku C polegałoby to na użyciu wskaźnika na funkcję – możemy go składować, a funkcję wywołać później. Command zapewnia to w sposób obiektowy.
- By specyfikować, kolejkować i wykonywać polecenia w różnym czasie – czas życia obiektu Command jest niezależny od czasu życia oryginalnego polecenia. Można też przesłać obiekt żądania do innej przestrzeni adresowej.
- Implementacja undo – pozwala zachować dane konieczne do odwrócenia operacji. Command powinna wówczas zawierać operację Unexecute, cofającą operację Execute. Wykonane operacje przechowywane na liście „historii”.
- Dziennik operacji – lista wykonanych Poleceń może być zapisana, by móc np. odtworzyć ją jeszcze raz w razie awarii. Może to też służyć jako proste makro.

```
class Matrix {
     private double[][] matrix;
     ...
     public void set(int row, int col, double value {
          matrix[row][col] = value;
     }
}
```

```
class SetCommand implements Command {
     private int row, col;
     private double value;
     public SetCommand(int row, int col, double value) {
          this.row = row; this.col = col; this.value = value;
     }
     public void execute(Matrix matrix) {
          matrix.set(row, col, value);
     }
}
public class Client {
     public static void main(...) {
          // ...
          SetCommand cmd = new SetCommand(2, 3, 2.5);
          // ...
          Matrix matrix1 = new Matrix(5, 5);
          cmd.execute(matrix1);
          // ...
          Matrix matrix2 = new Matrix(3, 4);
          cmd.execute(matrix2);
     }
}
```

```
// inne rozwiązanie:
class SetCommand implements Command {
     private int row, col;
     private double value;
     private Matrix receiver;
     public SetCommand(Matrix receiver, int, int, double) {
          this.receiver = receiver;
          ...
     }
     public void execute() {
          receiver.set(row, col, value);
     }
}
public class Client {
     public static void main(...) {
          Matrix mat1 = new Matrix(5, 5);
          // ...
          SetCommand cmd = new SetCommand(mat1, 2, 3, 2.5);
          // ...
          cmd.execute();
          // ...
     }
}
```

## *Konsekwencje:*

1. Oddziela obiekt wywołujący operację (zleceniodawcę) od obiektu wiedzącego jak ją wykonać (wykonawcy).
2. Operację możemy traktować jak obiekt – manipulować nią, rozszerzać, etc.
3. Można składać polecenia w bardziej złożone, np. makra (wzorzec *Composite*)
4. Łatwo można dodawać kolejne polecenia.

## *Implementacja:*

1. Jak bardzo inteligentne powinno być polecenie? – może tylko wywołuje operację odbiorcy (definiując jedynie połączenie między odbiorcą a akcją), a może samo wszystko implementuje, odbiorcy nie angażując w ogóle (przydatne, gdy nie chcemy polegać na istniejących klasach, nie ma odbiorcy lub jest bardzo dobrze znany, np. polecenie utworzenia okna może je spełnić równie dobrze, jak każdy inny obiekt).
2. Implementacja undo/redo – *Command* powinien umieć odwrócić swe działanie, czasem będzie musiał przechować dodatkowe informacje:
   - odbiorcę, który wykonuje operację,
   - argumenty operacji,
   - oryginalne wartości, które mogły zostać zmienione w odbiorcy jako efekt wykonania operacji (aby móc przywrócić mu jego pierwotny stan),
   - potrzebujemy całej listy Commandów, aby mieć wielopoziomowe undo.

```
class SetCommand implements Command {
     private int row, col;
     private double value, oldvalue;
     private Matrix receiver;
     public void execute() {
          oldvalue = receiver.get(row, col);
          receiver.set(row, col, value);
     }
     public void unexecute() {
          receiver.set(row, col, oldvalue); }
}
public class Client {
     public static void main(...) {
          Matrix mat1 = new Matrix(5, 5);
          Stos<Command> undo = new Stos<Command>();
          // ... wykonanie akcji
          SetCommand cmd = new SetCommand(mat1, 2, 3, 2.5);
          cmd.execute();
          undo.push(cmd);
          // ... cofnięcie akcji
          undo.pop().unexecute();
     }
}
```

3. Kumulacja błędów w undo/redo – zaokrąglenia w wielokrotnym cofaniu operacji będą się nakładać i nie wrócimy do oryginalnego stanu, zatem będziemy niekiedy potrzebowali więcej danych, by przywrócić poprzedni stan – wzorzec Memento.

## *Powiązania:*

- Composite – do budowy makr
- Memento – może przechować stan potrzebny do cofnięcia operacji
- polecenie, które kopiujemy przed umieszczeniem w historii działa jak Prototyp

```
class MacroCommand implements Command {
     private ArrayList<Command> commands;
     public MacroCommand() {
          commands = new ArrayList<Command>();
     }
     public void add(Command cmd) {
          commands.add(cmd);
     }
     public void execute() {
          for(Command cmd : commands)
               cmd.execute();
     }
}
```