

# **Strategia i Polecenie – wzorce projektowe**

Paweł Tymoczko

# Behawioralne wzorce projektowe

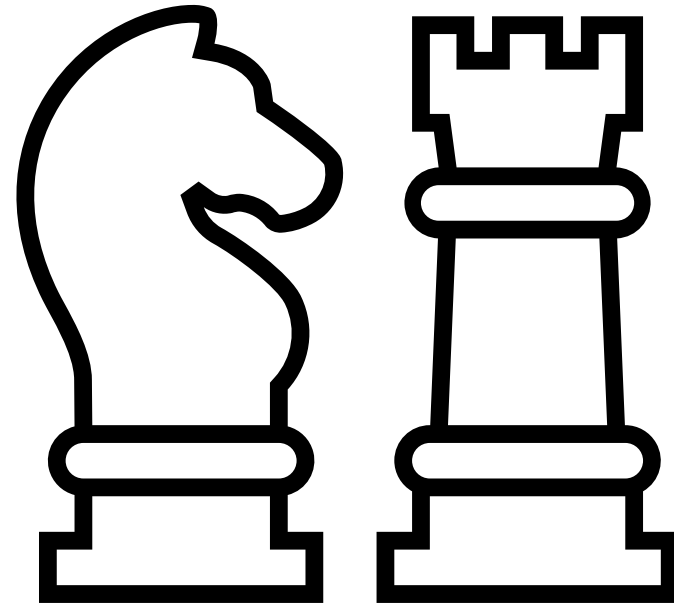
## **Wzorce behawioralne**

dotyczą algorytmów i rozdzielania odpowiedzialności pomiędzy obiektami.

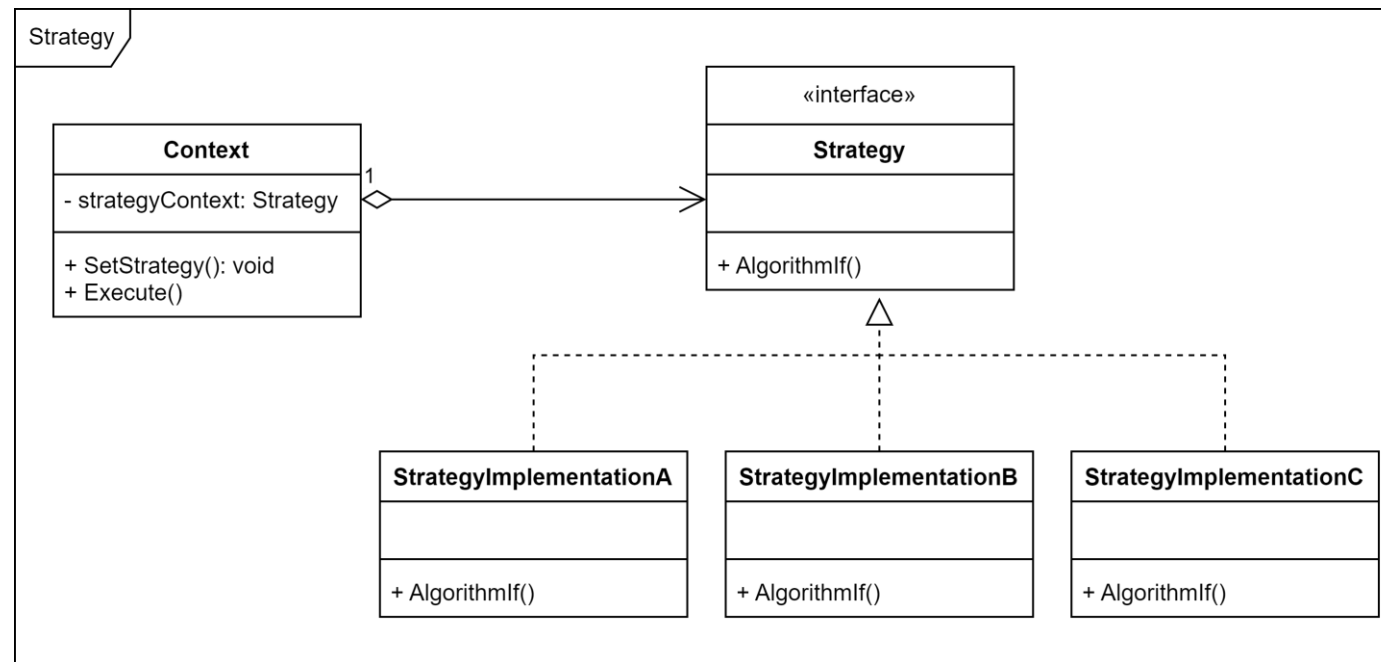
## **Wybrane wzorce behawioralne:**

- Łańcuch zobowiązań,
- *Polecenie*,
- Iterator,
- Mediator,
- Pamiętka,
- Obserwator,
- Stan,
- *Strategia*,
- Metoda szablonowa,
- Odwiedzający,

# **Strategia** *ang. Strategy*



# Strategia - ogólna koncepcja



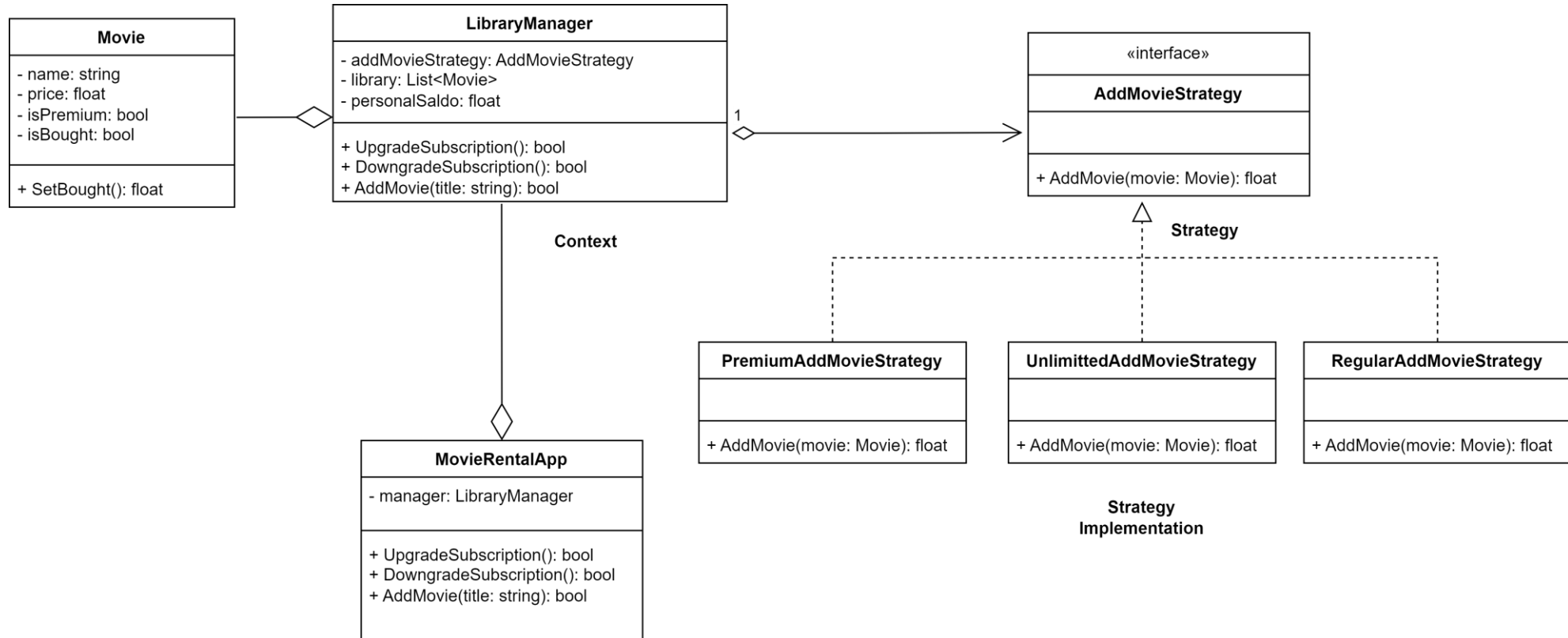
Rys. 1 Ogólny diagram klas wzorca Strategia

**Strategy** - Wspólny interfejs dla wszystkich strategii, udostępnia wirtualną metodę, którą wykonywany jest algorytm.

**StrategyImplementationX** - Klasa dziedzicząca po interfejsie *Strategy*, implementująca konkretny algorytm.

**Context** - Klasa przechowująca odniesienie do jednej (lub zestawu) strategii, dzięki czemu jest w stanie ją wykonać. Przechowywanie referencji do interfejsu *Strategy* pozwala na dynamiczną zmianę implementacji strategii.

# Strategia - Przykład



Rys. 2 Diagram klas demonstracyjnego projektu MovieRentalApp.

Implementacja projektu w języku c# oraz C++ znajduje się w repozytorium autora:  
[https://github.com/Pawlicho/design\\_patterns\\_demo](https://github.com/Pawlicho/design_patterns_demo)

# Strategia – zalety

**Wprowadzenie dodatkowej złożoności projektu** – Wprowadzenie dodatkowej warstwy relacji między nadawcą a odbiorcą może skomplikować projekt oraz utrudnić jego analizę.

**Open-Closed** – wprowadzenie wspólnego interfejsu dla każdej strategii pozwala na dodawanie kolejnych implementacji bez konieczności zmiany projektu.

**Zmiana algorytmu w czasie trwania programu** – zmiana algorytmu, który wykorzystuje kontekst wymaga jedynie zmiany referencji obiektu interfejsu.

**Zmniejszenie ilości wyrażeń warunkowych** – zmiana zachowania klasy wymaga jedynie ustawienia odpowiedniej strategii, raz wybrana strategia może być wykonywana bez konieczności sprawdzania wyrażeń warunkowych.

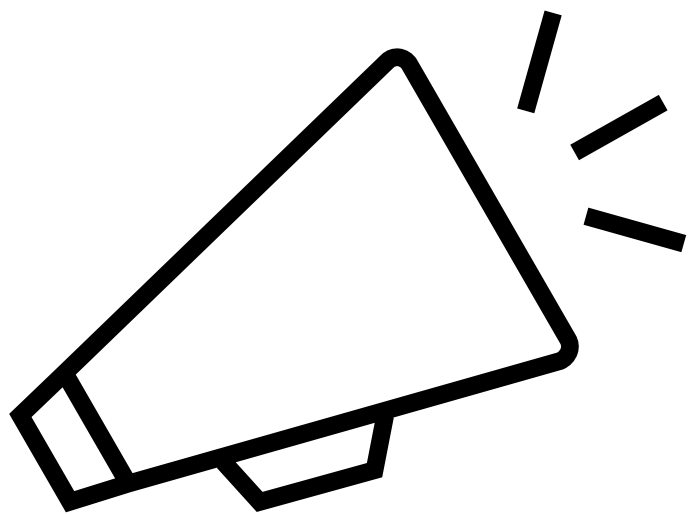
# Strategia – wady

**Klient musi być świadomy istnienia różnych strategii** – aby wybrać właściwą strategię, niezbędne jest poznanie szczegółów implementacyjnych dostępnych strategii.

**Zwiększenie liści obiektów** – wykonanie strategii wymaga, aby w pamięci programu istniał przynajmniej jeden taki obiekt.

**Dodatkowy narzut parametrów w komunikacji między Strategią a Contextem** – wykonanie konkretnej implementacji parametrów wymaga przekazania przez Context zestawu danych do interfejsu Strategy. Oznacza to, że za każdym razem strategia otrzymuje maksymalny zestaw danych i parametrów, niezależnie czy jest trywialna czy skomplikowana.

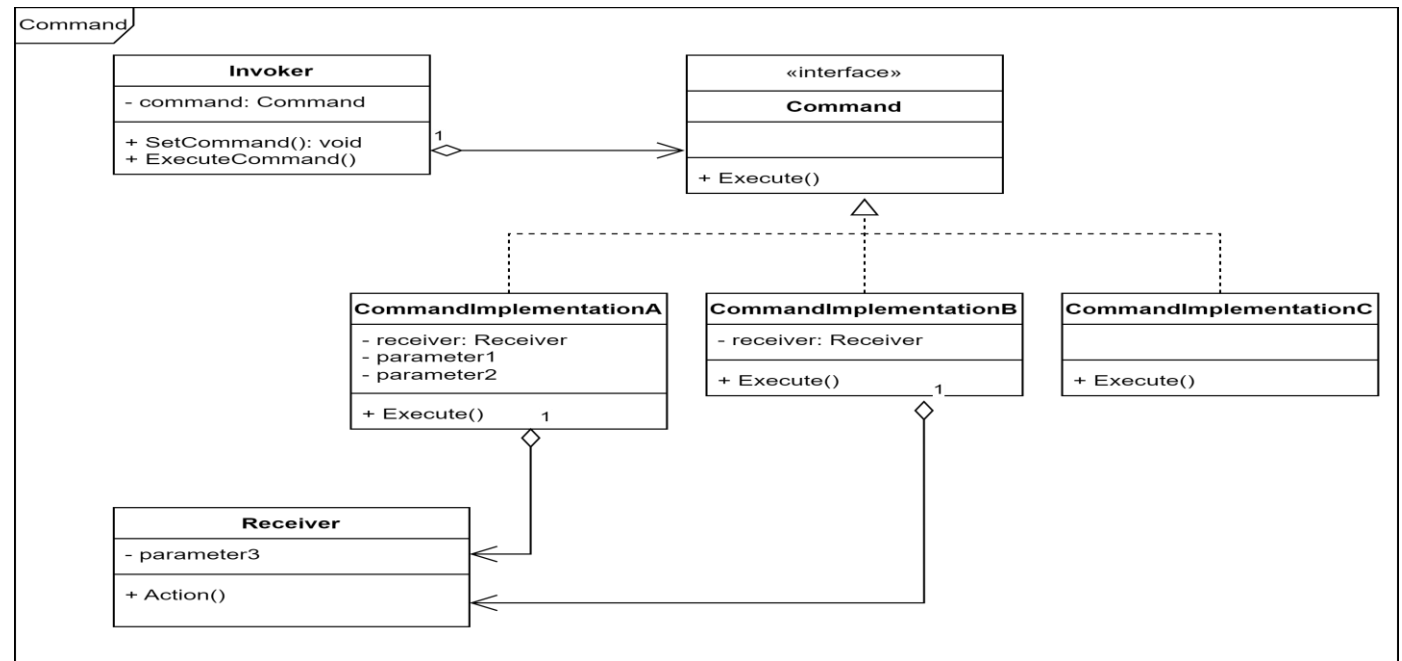




**Polecenie**  
ang. Command



# Polecenie - ogólna koncepcja



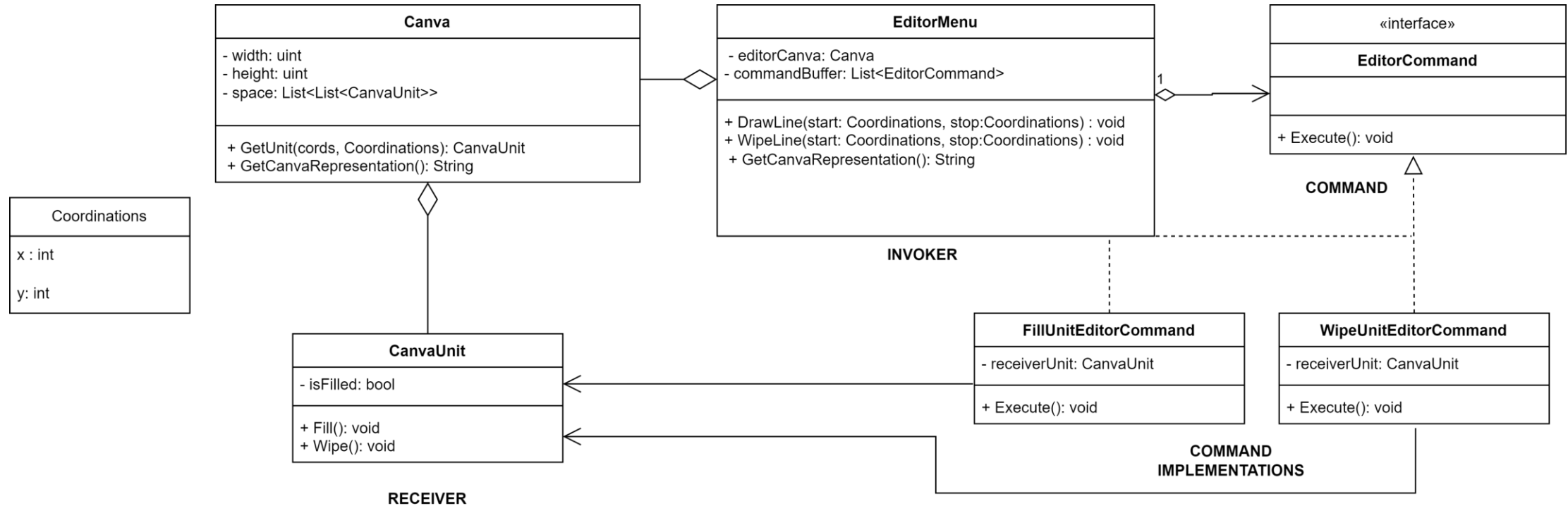
Rys. 3 Ogólny diagram klas wzorca Polecenie.

**Command** - Wspólny interfejs dla wszystkich poleceń, udostępnia wirtualną metodę, którą wykonywane jest zadanie.

**CommandImplementationX** - Klasa dziedzicząca po interfejsie *Strategy*, implementująca konkretne polecenie. Najczęściej posiada referencje do obiektu odbiorcy **Receiver**.

**Receiver** - Odbiorca polecenia, obiekt, którego metoda **Action** jest wywoływana podczas przetwarzania polecenie (tj. po wykonaniu **Execute**)

# Polecenie - Przykład



Rys. 4 Diagram klas demonstracyjnego projektu GraphicEditor.

Implementacja projektu w języku c# oraz C++ znajduje się w repozytorium autora:  
[https://github.com/Pawlicho/design\\_patterns\\_demo](https://github.com/Pawlicho/design_patterns_demo)

# Polecenie – zalety


**Single-responsibility** - wprowadza izolację między klasą wykonującą polecenie od klasy wywołującej polecenie

**Open-Closed** - wprowadzenie wspólnego interfejsu dla każdego polecenia pozwala na dodawanie kolejnych implementacji bez konieczności zmiany projektu.

**Cofanie operacji** - wykorzystując ten wzorec, niewielkim kosztem można wprowadzić mechanizm cofania poleceń.

**Kolejkowanie i opóźnianie** - poprzez przechowywanie obiektów poleceń można wprowadzić mechanizm ich kolejkowania oraz opóźniania

**Transakcje** - umożliwia łączenie poleceń w celu wykonania ich partiami.



# Polecenie – wady

**Klient musi być świadomy istnienia różnych strategii** – aby wybrać właściwą strategię, niezbędne jest poznanie szczegółów implementacyjnych dostępnych strategii.

**Zwiększenie liści obiektów** – wykonanie strategii wymaga, aby w pamięci programu istniał przynajmniej jeden taki obiekt.

**Dodatkowy narzut parametrów w komunikacji między Strategią a Contextem** – wykonanie konkretnej implementacji parametrów wymaga przekazania przez Context zestawu danych do interfejsu Strategy. Oznacza to, że za każdym razem strategia otrzymuje maksymalny zestaw danych i parametrów, niezależnie czy jest trywialna czy skomplikowana.



# Źródła

- E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.*, Addison-Wesley Professional Computing Series
- <https://refactoring.guru/pl/design-patterns/command>  
dostęp: 12:23 10.03.2024
- <https://refactoring.guru/pl/design-patterns/strategy>  
dostęp: 10:30 10.03.2024



**Dziękuję  
za uwagę!**