# Architektura Obiektowa w Praktyce

**Gdzie jesteśmy?**

- **Mechanika OOP (K4-1, K4-2):** Opanowaliśmy tworzenie klas, dziedziczenie, metody specjalne, dataclasses. Wiemy, jak budować.

**Cel na dziś:** Połączyć teorię z praktyką. Zaimplementować kluczowe wzorce projektowe i zobaczyć zasady SOLID w akcji.

**Plan Działania:**

a. **Zadanie 1: Wzorzec Fabryka (krecyjny).**
   Zbudujemy centralny punkt do tworzenia naszych postaci.

b. **Zadanie 2: Wzorzec Strategia (behawioralny).**
   Umożliwimy dynamiczną zmianę zachowania obiektów (np. ataku).

c. **Zadanie 3: Wzorzec Dekorator (strukturalny).**
   Nauczymy się "opakowywać" obiekty, dodając im nowe funkcje.

d. **Zadanie 4 : Wzorzec Adapter.**
   Zintegrujemy "stary" system z "nowym" bez zmiany ich kodu.

e. **Zadanie 5 : Refaktoryzacja do DIP.**
   "Odwrócimy zależności" i użyjemy wstrzykiwania zależności.

f. **Zadanie 6 : Wzorzec Singleton (kreacyjny).**
   Zapewnimy istnienie tylko jednej instancji danej klasy.

g. **Zadanie 7 : Wzorzec Obserwator (behawioralny).**
   Zbudujemy system powiadomień o zmianach stanu.

h. **Zadanie 8 : Prawo Demeter ("Mów, nie Pytaj").**
   Zrefaktoryzujemy kod, aby zmniejszyć jego powiązania.

i. **Podsumowanie.**

# Zadanie 1 - Wzorzec Fabryka (Kreacyjny)

**Problem:** W głównym kodzie gry, tworzenie postaci wygląda tak:

```
if wybor_gracza == "wojownik":
    gracz = Wojownik("Aragorn", hp=120, sila=25)
elif wybor_gracza == "mag":
    gracz = Mag("Gandalf", hp=80, mana=50)
# ... co jeśli dodamy Łucznika, Kapłana, Łotra?
```

- Kod klienta jest zaśmiecony logiką tworzenia.
- Złamanie **Zasady Jednej Odpowiedzialności** (SRP): główna pętla gry nie powinna wiedzieć, **jak** tworzyć wojowników.

**Rozwiązanie:** Wzorzec Fabryka. Przenosimy logikę tworzenia do dedykowanej funkcji (lub klasy).

**Zadanie:**

a. Przygotuj klasy **Gracz, Wojownik(Gracz)** i **Mag(Gracz)** z poprzednich zajęć. Upewnij się, że mają **_init_ i _repr_.**

b. Stwórz funkcję **fabryka_postaci(typ: str, imie: str) -> Gracz.**

c. **Wewnątrz funkcji, zaimplementuj logikę if/elif/else:**

- Jeśli **typ == "wojownik",** zwróć **Wojownik(imie, hp=120, sila=25).**
- Jeśli **typ == "mag",** zwróć **Mag(imie, hp=80, mana=50).**
- W przeciwnym razie, podnieś wyjątek **ValueError** z komunikatem.

d. **Przetestuj:**

- Użyj fabryki do stworzenia wojownika i maga.
- Wyświetl stworzone obiekty, aby sprawdzić, czy są poprawne.
- Spróbuj stworzyć nieznany typ postaci w bloku **try...except**.

# Omówienie Zadania 1 - Wzorzec Fabryka

**Cel:** Analiza rozwiązania i utrwalenie, jak Fabryka realizuje Zasadę Jednej Odpowiedzialności (SRP).

**Kluczowe korzyści:**

- **Czysty kod klienta:** Główna logika gry nie jest zaśmiecona if-ami.
- **Centralizacja:** Logika tworzenia jest w jednym miejscu.
- **Elastyczność:** Aby dodać Łucznika, modyfikujemy tylko fabrykę.

```
# Zakładamy, że klasy Gracz, Wojownik i Mag istnieją z
poprzednich zajęć
# i mają zaimplementowane __repr__ dla czytelności.

# 1. Funkcja-fabryka - jedyne miejsce z logiką tworzenia
def fabryka_postaci(typ: str, imie: str) -> Gracz:
    """Centralny punkt do tworzenia obiektów postaci."""
    if typ == "wojownik":
        return Wojownik(imie, hp=120, sila=25)
    elif typ == "mag":
        return Mag(imie, hp=80, mana=50)
    else:
        raise ValueError(f"Nieznany typ postaci: {typ}")

# 2. Kod klienta - czysty i niezależny od szczegółów
print("--- Testowanie fabryki ---")
try:
    wojownik = fabryka_postaci("wojownik", "Aragorn")
    mag = fabryka_postaci("mag", "Gandalf")

    print(f"Stworzono: {wojownik}")
    print(f"Stworzono: {mag}")

    # Próba stworzenia nieznanego typu
    nieznany = fabryka_postaci("lotr", "Bilbo")

except ValueError as e:
    print(f"\nZłapano oczekiwany błąd: {e}")
```

# Zadanie 2 - Wzorzec Strategia (behawioralny)

**Problem:** Nasz **Wojownik** ma metodę **atakuj**. Ale co, jeśli chcemy **dynamicznie zmieniać sposób ataku** w trakcie gry? Np. wojownik podnosi magiczny miecz i teraz atakuje ogniem. Nie możemy zmienić jego klasy w trakcie działania programu!

**Rozwiązanie: Wzorzec Projektowy "Strategia"** - "Preferuj kompozycję nad dziedziczeniem".

- Zamiast umieszczać logikę ataku w klasie gracza, "wyciągamy" ją do osobnych, wymiennych obiektów-strategii.
- Gracz **ma** strategię ataku (kompozycja), a nie **jest** strategią (dziedziczenie).

**Zadanie:** Zrefaktoryzuj system walki z użyciem wzorca Strategia.

a. **Zdefiniuj "kontrakt":** Stwórz klasę abstrakcyjną **StrategiaAtaku** z modułu **abc**, z jedną abstrakcyjną metodą **atakuj(self) -> str**.

b. **Stwórz konkretne strategie:** Stwórz klasy **AtakMieczem** i **AtakKulaOgnia**, które dziedziczą po **StrategiaAtaku** i implementują metodę **atakuj**, zwracając odpowiedni komunikat.

c. **Zmodyfikuj klasę Gracz:**
   - W konstruktorze **_init_** dodaj nowy parametr **strategia: StrategiaAtaku.**
   - Przechowaj go jako atrybut **self.strategia = strategia**.
   - Stwórz metodę **wykonaj_atak(self),** która **deleguje** zadanie do obiektu strategii: **print(f"{self.imie} atakuje: {self.strategia.atakuj()}").**
   - (Opcjonalnie) Dodaj metodę **zmien_strategie(self, nowa_strategia).**

d. **Przetestuj:**
   - Stwórz gracza, przekazując mu w konstruktorze strategię **AtakMieczem**.
   - Wywołaj **wykonaj_atak().**
   - Zmień strategię gracza na **AtakKulaOgnia**.
   - Ponownie wywołaj **wykonaj_atak()** i zaobserwuj zmianę zachowania.

# Zadanie 2 - Wzorzec Strategia (behawioralny)

**Cel:** Analiza rozwiązania i zrozumienie korzyści płynących z wzorca Strategia.

**Kluczowe korzyści:**

**Elastyczność:** Możemy dodawać nowe strategie bez modyfikacji klasy Gracz.

**Czysty kod:** Unikamy skomplikowanych instrukcji if/elif/else.

**Zasady SOLID:** Realizuje Zasadę Otwarte/Zamknięte (OCP).

```
from abc import ABC, abstractmethod

# 1. Kontrakt (interfejs) dla wszystkich strategii
class StrategiaAtaku(ABC):
    @abstractmethod
    def atakuj(self) -> str:
        pass

# 2. Konkretne, wymienne strategie
class AtakMieczem(StrategiaAtaku):
    def atakuj(self) -> str:
        return "wykonuje szybki cios mieczem!"


class AtakKulaOgnia(StrategiaAtaku):
    def atakuj(self) -> str:
        return "ciska potężną kulą ognia!"


# 3. Kontekst, który UŻYWA strategii (przez kompozycję)
class Gracz:
    def __init__(self, imie, strategia: StrategiaAtaku):
        self.imie = imie
        self._strategia = strategia # Gracz MA strategię

    def zmien_strategie(self, nowa_strategia: StrategiaAtaku):
        print(f"-> {self.imie} zmienia strategię walki.")
        self._strategia = nowa_strategia

    def wykonaj_atak(self):
        # 4. Delegowanie zadania do obiektu strategii
        wynik_ataku = self._strategia.atakuj()
        print(f"{self.imie} {wynik_ataku}")

print("\n--- Testowanie wzorca Strategia ---")
rycerz = Gracz("Artur", AtakMieczem())
rycerz.wykonaj_atak()
print("-" * 20)
rycerz.zmien_strategie(AtakKulaOgnia())
rycerz.wykonaj_atak()
```

# Zadanie 3 - Wzorzec Dekorator (Structural)

**Problem:** Jak dodać nowe funkcjonalności (np. z magicznych przedmiotów) do obiektu **dynamicznie**, bez tworzenia dziesiątek podklas? (Np. **WojownikZMagicznymMieczem**, **WojownikZOgnistaZbroja**, **WojownikZObomaPrzedmiotami**...).

**Rozwiązanie:** "Opakowywanie" (wrapping) obiektu w inne obiekty (dekoratory), które mają ten sam interfejs i dodają nowe zachowanie.

**Zadanie:** Zaimplementuj system ekwipunku dodającego bonusy do ataku.

a. **Zdefiniuj "komponent":** Stwórz klasę abstrakcyjną **Bohater** z modułu **abc** z dwiema metodami: **atak()** (zwraca **int**) i **opis()** (zwraca **str**).

b. **Stwórz konkretny komponent:** Stwórz klasę **Wojownik(Bohater),** która implementuje interfejs. **atak()** zwraca **10**, a **opis()** zwraca **"Wojownik".**

c. **Zdefiniuj "kontrakt" dekoratora:** Stwórz klasę abstrakcyjną **DekoratorBohatera(Bohater).**

   - W **__init__** przyjmij obiekt **opakowywany_bohater: Bohater** i zapisz go.
   - Zaimplementuj **atak()** i **opis()** tak, aby **delegowały** wywołanie do opakowywanego obiektu.

d. **Stwórz konkretne dekoratory:** Stwórz klasy **MagicznyMiecz(DekoratorBohatera)** i **OgnistaZbroja(DekoratorBohatera).**

   - Nadpisz w nich metody **atak()** i **opis().**
   - Metoda **atak()** powinna wywołać **super().atak()** i dodać bonus (np. Miecz: +5, Zbroja: +2).
   - Metoda **opis()** powinna wywołać **super().opis()** i dokleić opis przedmiotu (np. ", z magicznym mieczem").

e. **Przetestuj:**

   - Stwórz bazowego wojownika: **bohater = Wojownik().**
   - "Udekoruj" go, opakowując w kolejne warstwy: **bohater = MagicznyMiecz(bohater), bohater = OgnistaZbroja(bohater).**
   - Wyświetl ostateczny opis i siłę ataku.

# Omówienie Zadania 3 - Wzorzec Dekorator

**Cel:** Analiza jak Dekorator pozwala dynamicznie dodawać funkcjonalności.

**Kluczowe korzyści:**

- **Unikamy "eksplozji klas":** Nie potrzebujemy klasy **WojownikZMagicznymMieczem**.
- **Dynamiczne rozszerzanie:** Możemy dodawać i usuwać przedmioty w trakcie działania programu.
- **Zasada Otwarte/Zamknięte (OCP):** Możemy dodawać nowe przedmioty (dekoratory) bez modyfikacji istniejących klas.

```
from abc import ABC, abstractmethod

# 1. Abstrakcyjny, "opakowywany" komponent
class Bohater(ABC):
    @abstractmethod
    def atak(self) -> int: pass
    @abstractmethod
    def opis(self) -> str: pass

# 2. Konkretny komponent
class Wojownik(Bohater):
    def atak(self) -> int: return 10
    def opis(self) -> str: return "Wojownik"

# 3. Abstrakcyjny "kontrakt" dla wszystkich dekoratorów
class DekoratorBohatera(Bohater):
    def __init__(self, opakowywany_bohater: Bohater):
        self._bohater = opakowywany_bohater

    def atak(self) -> int: return self._bohater.atak()
    def opis(self) -> str: return self._bohater.opis()

# 4. Konkretne dekoratory dodające funkcjonalność
class MagicznyMiecz(DekoratorBohatera):
    def atak(self) -> int: return super().atak() + 5
    def opis(self) -> str: return super().opis() + ", z magicznym
mieczem"

class OgnistaZbroja(DekoratorBohatera):
    def atak(self) -> int: return super().atak() + 2
    def opis(self) -> str: return super().opis() + ", w ognistej
zbroi"

print("\n--- Testowanie wzorca Dekorator ---")
bohater = Wojownik()
print(f"1. {bohater.opis()} -> Atak: {bohater.atak()}")

# "Ubieramy" obiekt w kolejne warstwy
bohater = MagicznyMiecz(bohater)
print(f"2. {bohater.opis()} -> Atak: {bohater.atak()}")

bohater = OgnistaZbroja(bohater)
print(f"3. {bohater.opis()} -> Atak: {bohater.atak()}")
```

# Zadanie 4 - Wzorzec Adapter (strukturalny)

**Problem:** Jak sprawić, by obiekty o niekompatybilnych interfejsach mogły ze sobą współpracować?

**Przykład:** Nasz nowy system **(Gracz)** używa obiektów-strategii z metodą **atakuj().** Chcemy zintegrować postać ze starej biblioteki, która ma metodę **wykonaj_uderzenie().** Nie możemy zmieniać kodu starej biblioteki.

**Rozwiązanie**: Stworzyć klasę **Adaptera,** która "tłumaczy" wywołania.

**Zadanie:** Zintegruj "starego" bohatera z naszym nowym systemem walki.

a. **Stwórz "stary system":** Stwórz klasę **StaryBohater** z jedną metodą **wykonaj_uderzenie(),** która zwraca string ***"Stary bohater wykonuje potężne uderzenie!".***

b. **Stwórz Adapter:** Stwórz klasę **AdapterBohatera,** która dziedziczy po **StrategiaAtaku** (z zadania o Strategii).

- W **_init_** przyjmij obiekt **stary_bohater: StaryBohater** i zapisz go.
- Zaimplementuj wymaganą metodę **atakuj().**
- Wewnątrz **atakuj(), przetłumacz** wywołanie: wywołaj i zwróć wynik metody **wykonaj_uderzenie()** na opakowanym obiekcie.

c. **Przetestuj:**

- Stwórz obiekt **stary_heros = StaryBohater().**
- Stwórz adapter: **adapter = AdapterBohatera(stary_heros).**
- Stwórz obiekt **Gracz** z zadania o Strategii, przekazując mu adapter jako strategię ataku: **nowy_gracz = Gracz("Zadaptowany", adapter).**
- Wywołaj **nowy_gracz.wykonaj_atak()** i zaobserwuj, że wywoływana jest metoda ze **StaregoBohatera.**

# Omówienie Zadania 4 - Wzorzec Adapter

**Cel:** Analiza rozwiązania i zrozumienie, jak Adapter pozwala współpracować niekompatybilnym interfejsom.

**Kluczowe korzyści:**

- **Integracja:** Pozwala na współpracę klas, które nie zostały do tego zaprojektowane.
- **Brak modyfikacji:** Nie wymaga zmiany istniejącego kodu (ani nowego, ani starego).
- **Zasada Otwarte/Zamknięte (OCP):** Rozszerzamy system o nową funkcjonalność bez modyfikacji istniejących klas.

```
# Zakładamy, że klasy StrategiaAtaku i Gracz istnieją z zadania 2

# 1. "Stary system" - klasa z niekompatybilnym interfejsem
class StaryBohater:
    def wykonaj_uderzenie(self) -> str:
        return "wykonuje potężne uderzenie ze starego systemu!"

# 2. Adapter - "tłumacz"
# Dziedziczy po NOWYM interfejsie, a w środku "opakowuje" STARY obiekt.
class AdapterBohatera(StrategiaAtaku):
    def __init__(self, stary_bohater: StaryBohater):
        self._stary_bohater = stary_bohater

    def atakuj(self) -> str:
        # Tłumaczenie wywołania: nowy interfejs -> stary interfejs
        return self._stary_bohater.wykonaj_uderzenie()

# --- Testowanie ---
print("\n--- Testowanie wzorca Adapter ---")
stary_heros = StaryBohater()
adapter = AdapterBohatera(stary_heros)

# Nasz nowy system (Gracz) może teraz używać starego obiektu przez
adapter,
# nie wiedząc nic o jego prawdziwej implementacji.
nowy_gracz = Gracz("Zadaptowany Heros", adapter)
nowy_gracz.wykonaj_atak()
```

# Zadanie 5 - Zasada Odwrócenia Zależności (DIP)

**Problem:** Nasz kod wysokiego poziomu (np. główna pętla gry) jest "przyklejony" do konkretnych implementacji niskiego poziomu (np. do **ProstaFabrykaPostaci**).

```
class ManagerGry:
    def __init__(self):
        # Zależność od KONKRETNEJ implementacji!
        self.fabryka = ProstaFabrykaPostaci()

    def stworz_bohatera(self):
        return self.fabryka.stworz_postac("wojownik", "Domyslny")
```

**Rozwiązanie: Odwrócenie Zależności i Wstrzykiwanie Zależności.**

a. Moduły wysokiego poziomu nie tworzą swoich zależności.

b. Zamiast tego, **otrzymują je z zewnątrz** (np. w konstruktorze).

c. Zależą od **abstrakcji** (interfejsu), a nie od konkretnej klasy.

**Zadanie:** Zrefaktoryzuj system tworzenia postaci, aby był zgodny z DIP.

**a. Stwórz abstrakcję fabryki:** Stwórz klasę abstrakcyjną IFabrykaPostaci z modułu abc z jedną metodą abstrakcyjną stworz_postac(self, typ: str, imie: str) -> Gracz.

**b. Stwórz konkretną fabrykę:** Zmień swoją funkcję fabryka_postaci w klasę ProstaFabrykaPostaci, która dziedziczy po IFabrykaPostaci i implementuje jej metodę.

c. Stwórz moduł wysokiego poziomu: Stwórz klasę ManagerGry.

d. Jej _init_ powinien przyjmować argument fabryka: IFabrykaPostaci.

e. Stwórz metodę rozpocznij_gre(self), która użyje wstrzykniętej fabryki do stworzenia postaci.

f. Przetestuj (Wstrzykiwanie Zależności):

g. Na zewnątrz, stwórz instancję ProstaFabrykaPostaci.

h. Stwórz instancję ManagerGry, "wstrzykując" mu fabrykę.

i. Wywołaj metodę rozpocznij_gre() i zaobserwuj działanie.

# Omówienie Zadania 5 - Zasada Odwrócenia Zależności (DIP)

**Cel:** Analiza rozwiązania i zrozumienie, jak DIP i Wstrzykiwanie Zależności prowadzą do elastycznego i testowalnego kodu.

**Kluczowe korzyści:**

- **Luźne powiązania: ManagerGry** nie wie o istnieniu ProstaFabrykaPostaci.
- **Elastyczność:** Możemy łatwo podmienić fabrykę na inną (np. **FabrykaPotworow**) bez zmiany **ManageraGry**.
- **Testowalność:** W testach możemy "wstrzyknąć" fałszywą fabrykę (**FalszywaFabrykaDlaTestow**).

```python
from abc import ABC, abstractmethod

# 1. Abstrakcja ("gniazdko"), od której zależą wszyscy
class IFabrykaPostaci(ABC):
    @abstractmethod
    def stworz_postac(self, typ: str, imie: str) -> Gracz:
        pass

# 2. Konkretna implementacja ("wtyczka")
class ProstaFabrykaPostaci(IFabrykaPostaci):
    def stworz_postac(self, typ: str, imie: str) -> Gracz:
        if typ == "wojownik":
            return Wojownik(imie, hp=120, sila=25)
        elif typ == "mag":
            return Mag(imie, hp=80, mana=50)
        else:
            raise ValueError(f"Nieznany typ postaci: {typ}")

# 3. Moduł wysokiego poziomu, który zależy od ABSTRAKCJI
class ManagerGry:
    def __init__(self, fabryka: IFabrykaPostaci):
        # Zależność jest "wstrzykiwana" z zewnątrz
        self._fabryka = fabryka

    def rozpocznij_gre(self):
        print("Manager Gry: Tworzę bohatera...")
        bohater = self._fabryka.stworz_postac("wojownik", "Geralt")
        print(f"Manager Gry: Stworzono postać: {bohater}")

print("\n--- Testowanie zasady DIP ---")
moja_fabryka = ProstaFabrykaPostaci()
manager = ManagerGry(fabryka=moja_fabryka)
manager.rozpocznij_gre()
```

# Zadanie 6 - Wzorzec Singleton (kreacyjny)

**Problem:** Jak zagwarantować, że w całym programie będzie istniała tylko jedna instancja danej klasy?

**Przykłady:** Menedżer konfiguracji gry, połączenie z bazą danych, globalny stan gry.

**Rozwiązanie:** Klasa sama kontroluje proces tworzenia instancji, używając metody **_new_.**

**Zadanie:** Stwórz klasę **MenedzerKonfiguracji** jako Singleton.

a. Stwórz klasę **MenedzerKonfiguracji**.

b. Dodaj do niej atrybut klasy **_instancja = None**.

c. Zaimplementuj metodę **_new_(cls).**
   - Wewnątrz sprawdź, czy **cls._instancja is None**.
   - Jeśli tak, stwórz nową instancję **(super()._new_(cls))** i przypisz ją do **cls._instancja**.
   - Zawsze zwracaj **cls._instancja**.

d. Dodaj prosty konstruktor **_init_,** który ustawi jakieś dane, np. **self.ustawienia = {"trudnosc": "normalna"}.**

e. **Przetestuj:**
   - Stwórz dwie instancje.
   - Sprawdź, czy to ten sam obiekt: **print(config1 is config2).**
   - Zmień ustawienie w **config1** (**config1.ustawienia["trudnosc"] = "trudna"**).
   - Wyświetl ustawienie z **config2** i zaobserwuj, że też się zmieniło.

**Uwaga:** Prosta implementacja _init_ będzie wywoływana za każdym razem. Jak temu zapobiec? (Pytanie do przemyślenia).

# Omówienie Zadania 6 - Wzorzec Singleton

**Cel:** Analiza rozwiązania i zrozumienie pułapek implementacyjnych
(np. wielokrotne wywoływanie **_init_**).

**Kluczowe korzyści i wady:**

- **Korzyść:** Gwarancja jednej instancji, globalny punkt dostępu.
- **Wada:** Tworzy globalny stan, utrudnia testowanie, często jest antywzorcem.

```
class MenedzerKonfiguracji:
    _instancja = None
    _zainicjalizowany = False # Flaga do kontroli __init__

    def __new__(cls, *args, **kwargs):
        if cls._instancja is None:
            print("-> Tworzę nową instancję...")
            cls._instancja = super().__new__(cls)
        return cls._instancja

    def __init__(self):
        if self._zainicjalizowany:
            return # Nie inicjalizuj ponownie

        print("-> Inicjalizuję instancję po raz pierwszy...")
        self.ustawienia = {"trudnosc": "normalna"}
        self._zainicjalizowany = True

# --- Testowanie ---
print("--- Testowanie wzorca Singleton ---")
config1 = MenedzerKonfiguracji()
config2 = MenedzerKonfiguracji()

print(f"\nCzy config1 to ten sam obiekt co config2? {config1 is
config2}")

print(f"Początkowe ustawienia: {config1.ustawienia}")
config2.ustawienia["trudnosc"] = "trudna"
print(f"Ustawienia po zmianie w config2: {config1.ustawienia}")
```

# Zadanie 7 - Wzorzec Obserwator (behawioralny)

**Problem:** Jak powiadomić wiele obiektów o zmianie stanu jednego obiektu, nie tworząc między nimi ścisłych powiązań?

**Przykład:** Gdy na forum pojawia się nowy post, wszyscy subskrybenci powinni dostać powiadomienie.

**Rozwiązanie:**

- Obserwowany (Subject): Utrzymuje listę obserwatorów i powiadamia ich o zdarzeniach.
- Obserwator (Observer): Definiuje interfejs (metodę) do otrzymywania powiadomień.

**Zadanie:** Zaimplementuj prosty system powiadomień w grze.

a. **Stwórz interfejs Obserwatora:** Stwórz klasę abstrakcyjną **IObserwator** z metodą **aktualizuj(self, wiadomosc: str).**

b. **Stwórz konkretnego Obserwatora:** Stwórz klasę **Gracz**, która dziedziczy po **IObserwator**.
   - W **__init__** przyjmij **imie**.
   - Zaimplementuj **aktualizuj**, aby wyświetlała np. **Gracz [imię] otrzymał powiadomienie: [wiadomosc] .**

c. **Stwórz Obserwowanego (Subject):** Stwórz klasę **SerwerGry**.
   - W **__init__** stwórz prywatną listę **self._obserwatorzy = [] .**
   - Zaimplementuj metody **dodaj_obserwatora** i **usun_obserwatora**.
   - Zaimplementuj prywatną metodę **_powiadom_wszystkich**, która iteruje po obserwatorach i wywołuje ich metodę **aktualizuj**.
   - Stwórz publiczną metodę **oglos_wydarzenie(self, wydarzenie),** która wywoła **_powiadom_wszystkich**.

d. **Przetestuj:**
   - Stwórz serwer i kilku graczy.
   - Zarejestruj graczy jako obserwatorów serwera.
   - Wywołaj **oglos_wydarzenie** i sprawdź, czy wszyscy dostali powiadomienie.
   - Usuń jednego gracza z listy obserwatorów i ponownie ogłoś wydarzenie.

# Omówienie Zadania 7 - Wzorzec Obserwator

**Cel:** Analiza rozwiązania i zrozumienie, jak Obserwator tworzy luźne powiązania między obiektami.

**Kluczowe korzyści:**

- **Luźne powiązania: SerwerGry** nie wie nic o klasie **Gracz.**
- **Dynamiczne relacje:** Obiekty mogą być dodawane i usuwane w trakcie działania programu.

```
class MenedzerKonfiguracji:
    _instancja = None
    _zainicjalizowany = False # Flaga do kontroli __init__

    def __new__(cls, *args, **kwargs):
        if cls._instancja is None:
            print("-> Tworzę nową instancję...")
            cls._instancja = super().__new__(cls)
        return cls._instancja

    def __init__(self):
        if self._zainicjalizowany:
            return # Nie inicjalizuj ponownie

        print("-> Inicjalizuję instancję po raz pierwszy...")
        self.ustawienia = {"trudnosc": "normalna"}
        self._zainicjalizowany = True

# --- Testowanie ---
print("--- Testowanie wzorca Singleton ---")
config1 = MenedzerKonfiguracji()
config2 = MenedzerKonfiguracji()

print(f"\nCzy config1 to ten sam obiekt co config2? {config1 is
config2}")

print(f"Początkowe ustawienia: {config1.ustawienia}")
config2.ustawienia["trudnosc"] = "trudna"
print(f"Ustawienia po zmianie w config2: {config1.ustawienia}")
```

# Zadanie 8 - Prawo Demeter ("Mów, nie Pytaj")

**Problem:** Nasz kod tworzy długie łańcuchy wywołań, co oznacza, że wie za dużo o wewnętrznej strukturze innych obiektów.

**Antywzorzec ("Wypadek kolejowy" - Train Wreck):**

```
# Wiemy, że postać ma ekwipunek, a ekwipunek ma przedmioty...
atak_gracza =
gracz.get_ekwipunek().get_przedmioty()["miecz"].get_atak()
```

**Rozwiązanie: delegowanie odpowiedzialności.** Zamiast "pytać" obiekty o ich wewnętrzne części, "mówimy" im, co mają zrobić.

**Zadanie:** Zrefaktoryzuj poniższy kod, aby był zgodny z Prawem Demeter.

**Kroki refaktoryzacji:**

- W klasie **Gracz** stwórz nową metodę **zaplac(self, kwota_do_zaplaty)**.
- Przenieś całą logikę **if** i modyfikacji portfela do wnętrza metody **zaplac**.
- Metoda **zaplac** powinna zwracać **True** (jeśli płatność się udała) lub **False** (jeśli nie).
- Zmień kod klienta tak, aby wywoływał tylko **gracz.zaplac(50).**

**Kod do refaktoryzacji:**

```
class Portfel:
    def __init__(self, kwota: float):
        self.kwota = kwota

class Gracz:
    def __init__(self, imie: str, portfel: Portfel):
        self.imie = imie
        self.portfel = portfel # Gracz MA portfel

gracz = Gracz("Geralt", Portfel(100))
if gracz.portfel.kwota >= 50:
    gracz.portfel.kwota -= 50
    print(f"Pobrano 50 zł. Pozostało: {gracz.portfel.kwota}")
```

# Omówienie Zadania 8 - Prawo Demeter ("Mów, nie Pytaj")

**Cel:** Analiza refaktoryzacji i zrozumienie, jak delegowanie odpowiedzialności zmniejsza powiązania między obiektami.

**Kod "Przed" (Pytamy o stan):**

```
# Kod klienta wie o istnieniu Portfela i jego atrybucie
'kwota'
if gracz.portfel.kwota >= 50:
    gracz.portfel.kwota -= 50
```

**Kluczowe korzyści:**

- **Mniejsza wiedza, mniejsze powiązania:** Kod klienta nie wie już, że **Gracz** używa **Portfela.**
- **Większa elastyczność:** Możemy zmienić **Portfel** na **KarteKredytowa** wewnątrz **Gracza**, a kod klienta nie ulegnie zmianie.
- **Lepsza enkapsulacja:** Logika związana z płatnością jest zamknięta wewnątrz **Gracza.**

**Kod "Po" (Mówimy, co ma zrobić):**

```
class Gracz:
    # ...
    def zaplac(self, kwota_do_zaplaty: float) -> bool:
        """Deleguje płatność do portfela."""
        if self.portfel.kwota >= kwota_do_zaplaty:
            self.portfel.kwota -= kwota_do_zaplaty
            print(f"-> {self.imie} “
                  “zapłacił {kwota_do_zaplaty} zł.")
            return True
        else:
            print(f"-> {self.imie} nie “
                  ”ma wystarczających środków.")
            return False

# Kod klienta jest teraz prostszy i nie zależy od Portfela
gracz.zaplac(50)
```

# Podsumowanie Laboratorium 4 (cz. 1, 2 i 3)

**Co osiągnęliśmy w trakcie modułu 4?**

**Poziom 1: Opanowaliśmy Mechanikę OOP (K4-1, K4-2)**

- Budowaliśmy klasy, hierarchie (**dziedziczenie**) i chroniliśmy dane **(@property**).
- Nauczyliśmy obiekty "mówić" językiem Pythona **(_str_, _add_, _iter_).**
- Używaliśmy nowoczesnych narzędzi (**dataclasses**, **_slots_**).

**Poziom 2: Zastosowaliśmy Zasady Architektury (K4-3)**

- Zobaczyliśmy SOLID w akcji: SRP (Fabryka), OCP (Strategia, Dekorator), DIP (Wstrzykiwanie Zależności).
- Zastosowaliśmy Prawo Demeter, delegując odpowiedzialność.

**Poziom 3: Zaimplementowaliśmy Wzorce Projektowe (K4-3)**

- **Kreacyjne**: Fabryka, Singleton.
- **Strukturalne**: Dekorator, Adapter.
- **Behawioralne**: Strategia, Obserwator.

**Gdzie jesteśmy?** Przeszliśmy drogę od bycia programistą klas do myślenia jak architekt systemów.

**Co dalej?** Jesteśmy gotowi, by używać tych wzorców do budowania prawdziwych, solidnych aplikacji.