
NOKIA

Tworzenie SOLIDnego kodu obiektowego w C++

Piotr Kowalczyk, Sebastian Zdanowicz

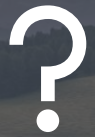
2023

NOKIA

Kontrakt

Ile planujemy przerw?

Jak się do siebie zwracamy?



Pytania zadajemy na bieżąco (chat lub mikrofon)
Nie ma „głupich pytań”

W czasie zajęć liczymy na aktywność uczestników

Zadania

Paczka z zadaniami do pobrania:

http://tiny.cc/paro23solid

Agenda

- Wartości oprogramowania
- Czym jest SOLID?
- Ćwiczenia praktyczne
  - DIP
  - ISP
  - OCP
  - LSP
  - SRP

Czym się różni projekt studencki od komercyjnego?

Projekt studencki nie musi być utrzymywany.

Wartości oprogramowania

- **Wartość pierwotna**: wyraża się poprzez zdolność do szybkiego wprowadzania zmian oraz dodawania nowych funkcjonalności.

- **Wartość wtórna**: wyraża się przez istniejącą funkcjonalność oprogramowania, zgodność z wymaganiami oraz brak błędów.

- **Dług techniczny**: jest zaciągany w momencie tworzenia oprogramowania bez uwzględnienia możliwego kierunku jego rozwoju. Może się zdarzyć, że wprowadzenie zmian w oprogramowaniu jest tak kosztowne, że bardziej opłacalne jest wykonanie projektu od nowa.

SOLID

- Mnemonik opisujący 5 zasad dobrego kodu obiektowego.
- Stosowanie tych zasad ułatwia w przyszłości rozwijanie oprogramowania.
- Zaproponowany przez Roberta C. Martina
  (ale nie jest on autorem wszystkich zasad).
- Nie jest związany z konkretnym językiem programowania.
- SOLID został opracowany na podstawie wieloletnich doświadczeń programistów.

SOLID

- Single responsibility principle (Zasada jednej odpowiedzialności)
- Open/closed principle (Zasada otwarte/zamknięte)
- Liskov substitution principle (Zasada podstawienia Liskov)
- Interface segregation principle (Zasada segregacji interfejsów)
- Dependency inversion principle (Zasada odwrócenia zależności)

Podstawowe relacje UML

Dependency inversion principle (Zasada odwrócenia zależności)

Problem

- Projekt wymaga aby zachodzące zdarzenia były logowane.
- Zdecydowano że te informacje będą zapisywane do bazy danych SQL.
- Sposób tworzenia raportów zaimplementowano w klasie `EventLog`.
- Szczegóły komunikacji z serwerem zamknięto w klasie `SqlStorage`.

Pytanie

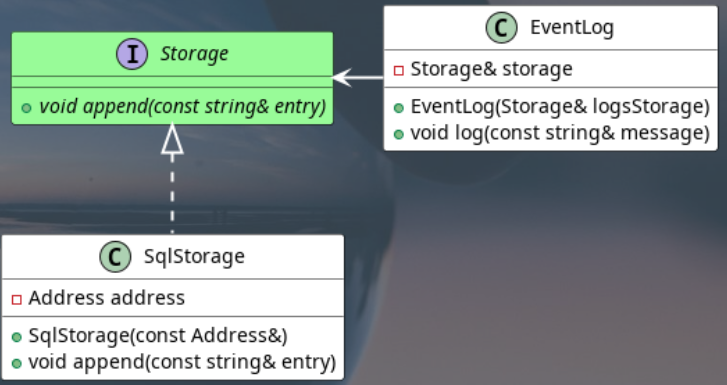
Jaki problem mogą napotkać programiści w przyszłości przy takim designie?

Pytanie

Jak rozwiązać problem zależności klasy `EventLog` od klasy `SqlStorage`?

Rozwiązanie

- Usuwamy niechcianą zależność przez wprowadzenie interfejsu (klasy abstrakcyjnej).
- Dzięki temu `EventLog` zależy tylko od tego interfejsu, nie od konkretnej implementacji i jej niskopoziomowych szczegółów.



Podsumowanie

- „Zapachy” towarzyszące naruszeniu zasady odwrócenia zależności:
  - dużo zależności w kodzie - kod jest „sztywny” (trudno wprowadzić zmiany) i „kruchy” (zmiana w jednej klasie może popsuć wiele innych),
  - kod wysokiego poziomu zależy od niskopoziomowych szczegółów,
  - podczas projektowania podejmowane są wybory bibliotek niskopoziomowych,
  - niewielka liczba interfejsów.
- Cechy kodu nienaruszającego zasady odwrócenia zależności:
  - kod jest modułowy – łatwo zastąpić jedne implementacje innymi,
  - brak bezpośrednich zależności pomiędzy implementacjami,
  - obiekty i moduły komunikują się poprzez interfejsy,
  - implementacje zależą od ogólnych interfejsów – nigdy na odwrót.

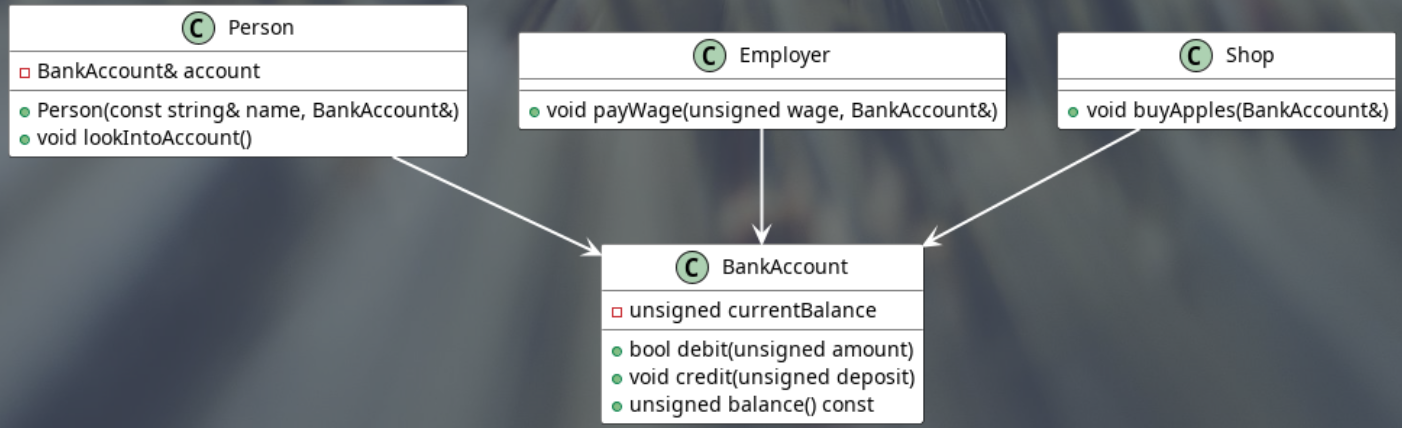
Dependency inversion principle (Zasada odwrócenia zależności)

„Moduły wysokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Jedne i drugie powinny zależeć od abstrakcji.”

Interface segregation principle (Zasada segregacji interfejsów)

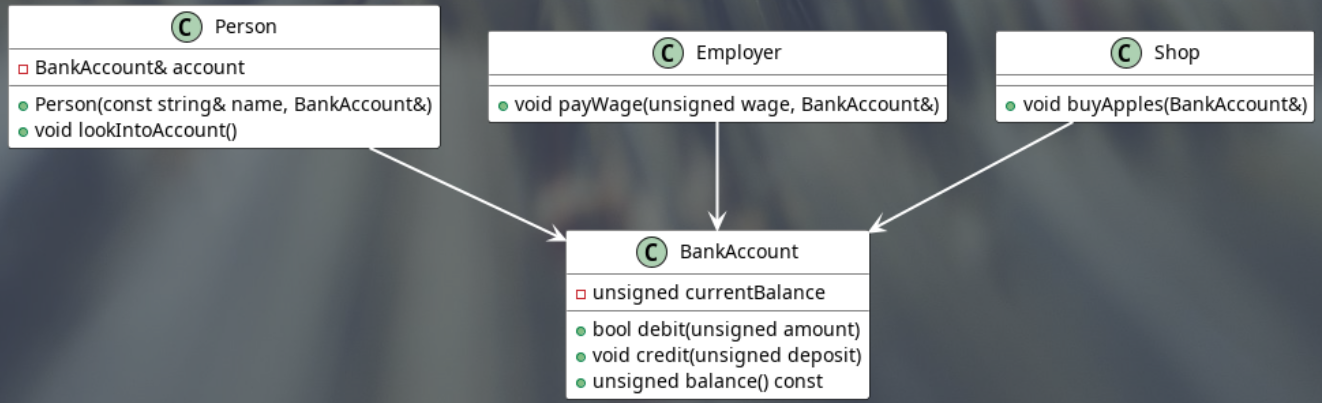
Problem

Program modeluje konto bankowe wykorzystywane przez właściciela, jego pracodawcę i sklep w którym robi on zakupy



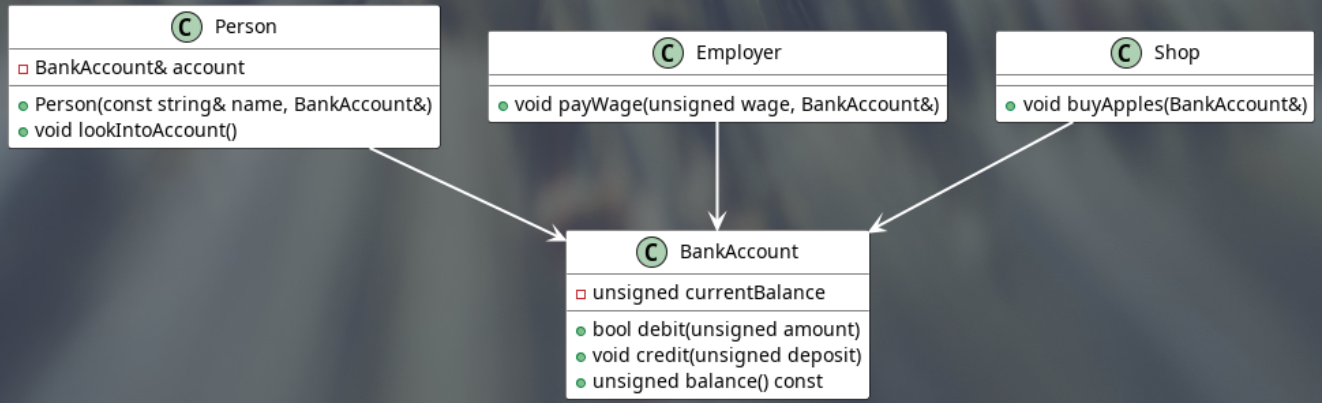
Pytanie

Jaki problem zauważamy w obecnym designie?



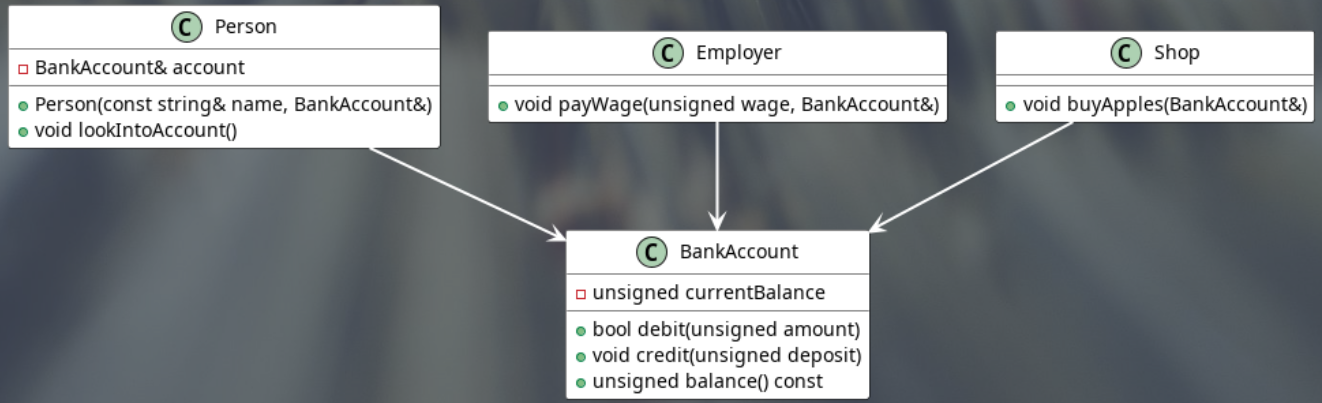
Pytanie

Klienci BankAccount mają dostęp do wszystkich jej metod – też do tych do których nie powinni.

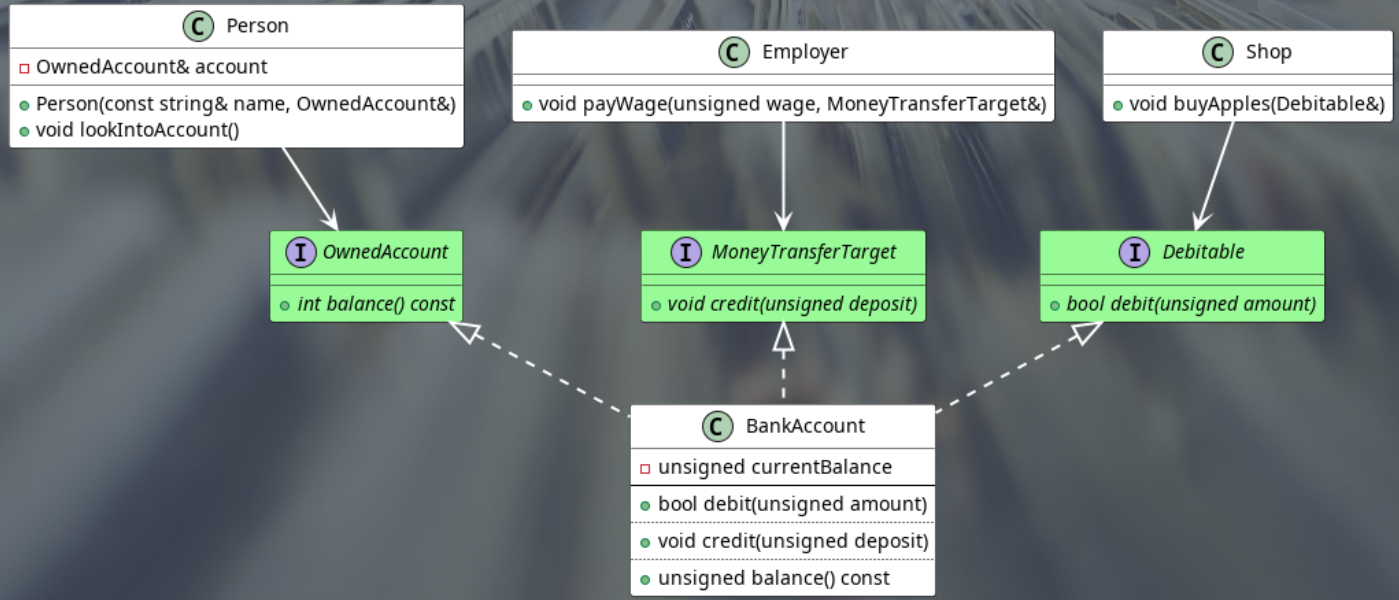


Pytanie

Jak możemy rozwiązać problem dostępu do wszystkich metod klasy BankAccount?



Rozwiązanie



Podsumowanie

- „Zapachy” towarzyszące naruszeniu zasady segregacji interfejsów:
  - klasa kliencka ma dostęp do metod, do których nie powinna mieć dostępu,
  - brak interfejsów (klas abstrakcyjnych),
  - jeśli interfejsy (klasy abstrakcyjne) istnieją, to mają dokładnie ten sam zbiór metod co implementujące je klasy.
- Cechy kodu nienaruszającego zasady segregacji interfejsów:
  - interfejsy są „skrojone na miarę” pod przypadki użycia, dzięki czemu klasy klienckie nie zależą od metod oraz typów których nie potrzebują.
- Sugestia:
  - wydzielenie interfejsów zgodnie z ISP może być pierwszym krokiem podczas refaktoryzacji kodu w którym występuje antywzorzec „The God class”.

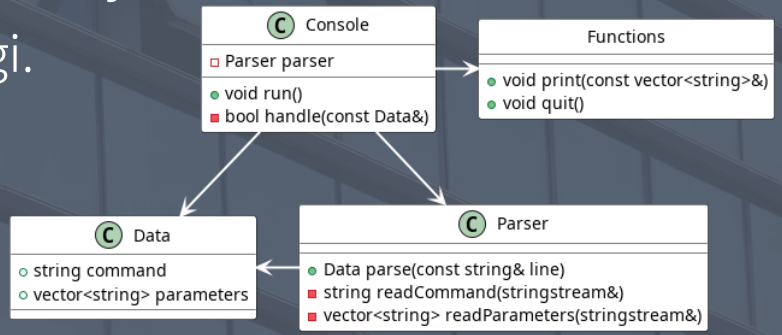
Interface segregation principle (Zasada segregacji interfejsów)

„Klienci nie powinni być zależni od metod, których nie używają.”

Open - closed principle
(Zasada otwarte - zamknięte)

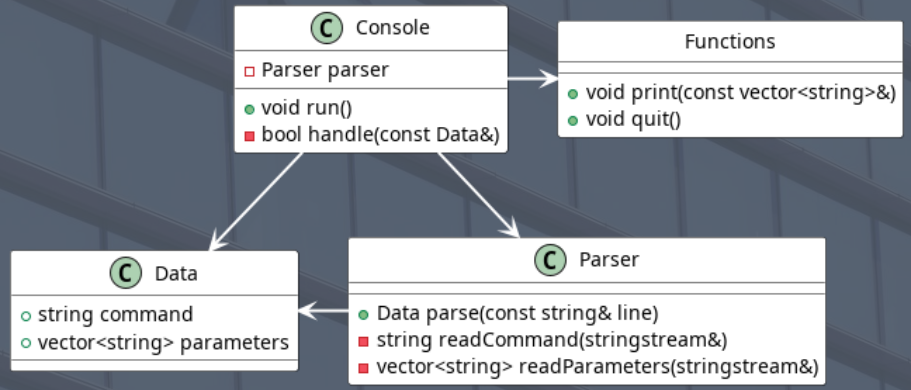
Problem

- Aplikacja `Console` obsługuje komendy: „**print**” oraz „**quit**”.
- Klasa `Parser` rozpoznaje komendę oraz oddziela ją od argumentów.
- Metoda `Console::handle()` dokonuje wyboru właściwej funkcji do obsługi.



Pytanie

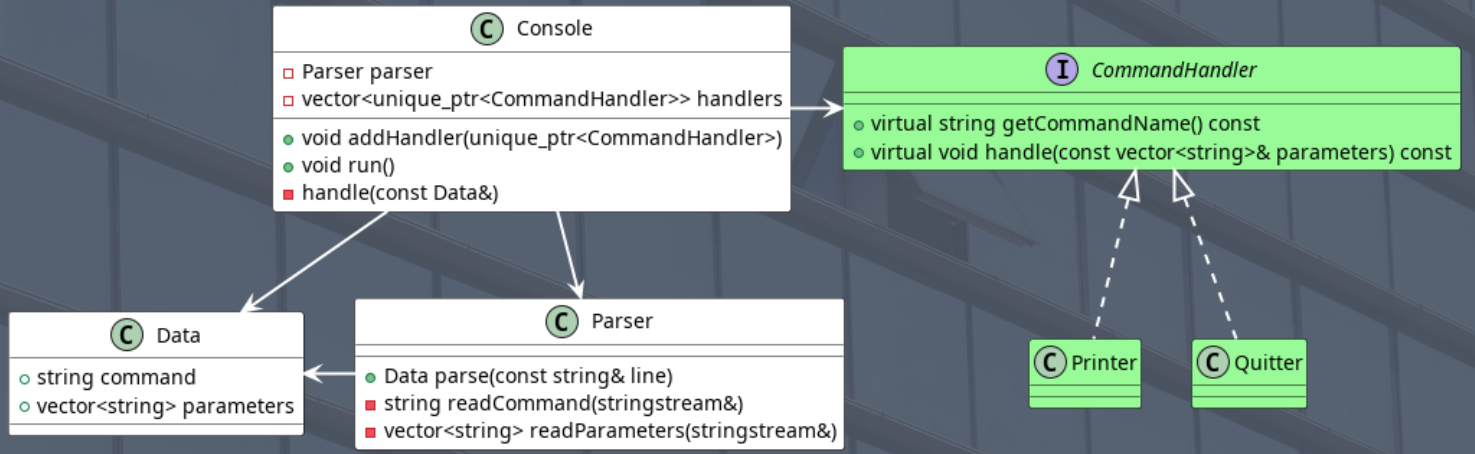
W którym miejscu w kodzie należy dokonać zmian aby dodać obsługę nowej komendy?



Jak być jednocześnie otwartym i zamkniętym?

- Oprogramowanie powinno być otwarte na dodawanie nowych funkcjonalności.
- Oprogramowanie powinno być zamknięte na zmiany w już istniejącym kodzie.
- Nowe wymagania powinniśmy móc zaimplementować tworząc nowy kod, nie modyfikując istniejącego.
- Aby zaprojektować kod w taki sposób musimy przewidzieć przyszłe zmiany.
- „Oś zmian jest osią zmiany, tylko wówczas, gdy zmiany rzeczywiście występują”.

Rozwiązanie



Podsumowanie

- „Zapachy” towarzyszące naruszeniu zasady otwarte/zamknięte:
  - kod jest „sztywny”,
  - za mało interfejsów.
- Cechy kodu nienaruszającego zasady otwarte/zamknięte:
  - logika biznesowa jest enkapsulowana w pojedynczych, polimorficznych klasach,
  - nowe funkcjonalności dodajemy na zasadzie „pluginów”,
  - ułatwione powtórne wykorzystanie kodu,
  - zredukowanie złożoności metod (brak konieczności używania konstrukcji `switch case`),
  - dodanie lub zmiana wymagań nie narusza już istniejącego (i działającego) kodu.

Ważna uwaga

- Musimy wcześnie przewidzieć kierunek rozwoju oprogramowania.
- Nadużywanie tej zasady zwiększa złożoność oraz zmniejsza czytelność kodu.

Open - closed principle (Zasada otwarte - zamknięte)

„Nowe wymagania powinniśmy móc zaimplementować tworząc nowy kod, nie modyfikując istniejącego.”

Dziękujemy za uwagę.

Kontakt do autorów:
- piotr.1.kowalczyk@nokia.com
- sebastian.zdanowicz@nokia.com


NOKIA