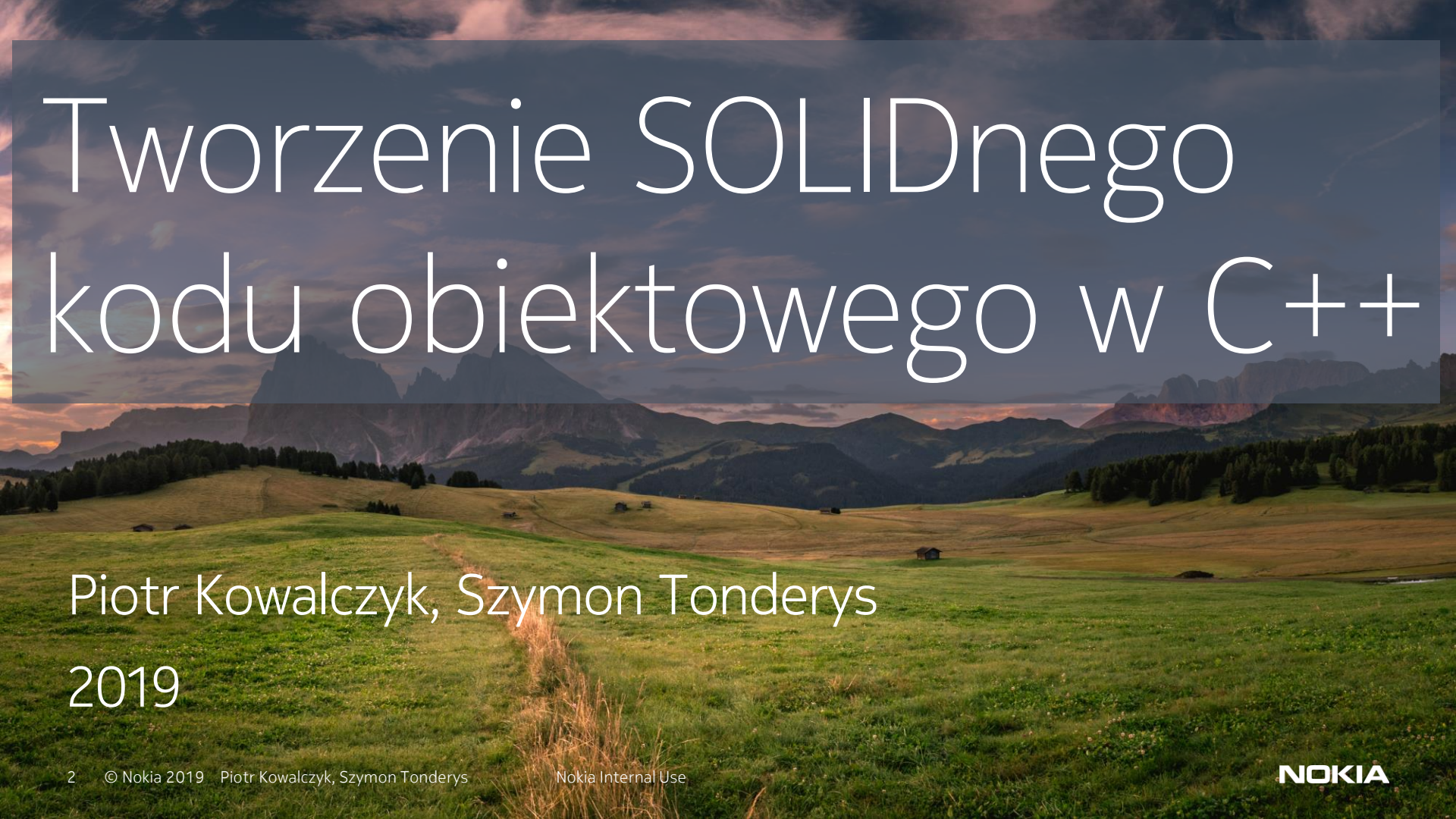

  Piotr Kowalczyk, Szymon Tonderys



# Tworzenie SOLIDnego kodu obiektowego w C++

Piotr Kowalczyk, Szymon Tonderys

2019



NOKIA

# Agenda

- Wartości oprogramowania
- Dobre praktyki projektowania zorientowanego obiektowo
- Zasada odwrócenia zależności
- Zasada pojedynczej odpowiedzialności
- Zasada segregacji interfejsów
- Zasada otwarte-zamknięte
- Zasada podstawienia Liskov

# Wartości oprogramowania

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

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

- **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.

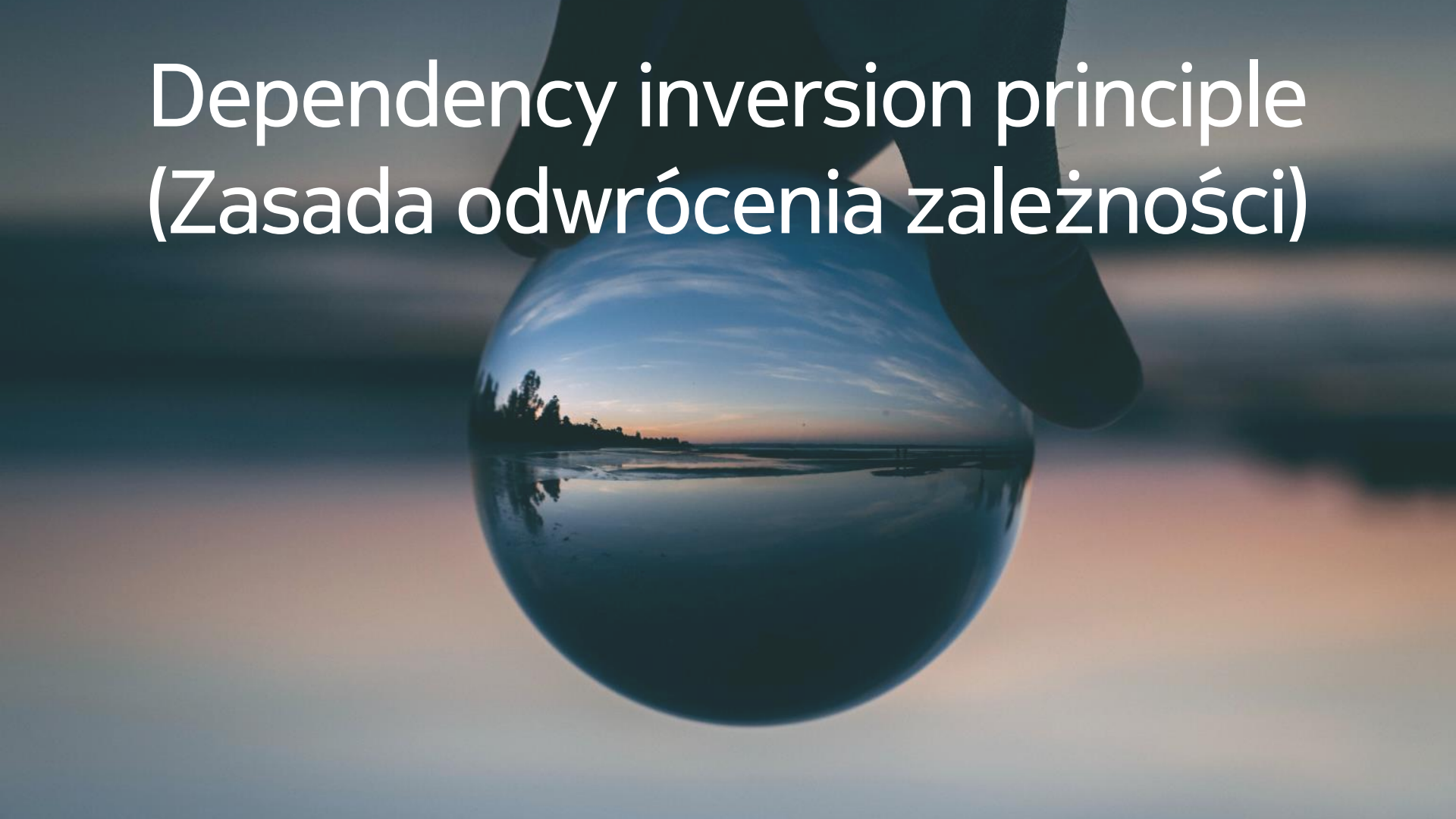
# SOLID

- 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

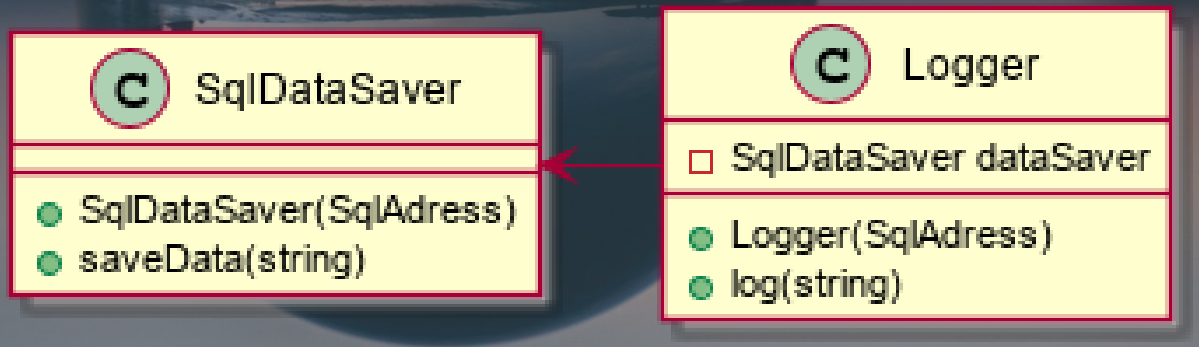
- **S**ingle responsibility principle (Zasada jednej odpowiedzialności)
- **O**pen/closed principle (Zasada otwarte/zamknięte)
- **L**iskov substitution principle (Zasada podstawienia Liskov)
- **I**nterface segregation principle (Zasada segregacji interfejsów)
- **D**ependency inversion principle (Zasada odwrócenia zależności)

# Dependency inversion principle (Zasada odwrócenia zależności)



# Problem

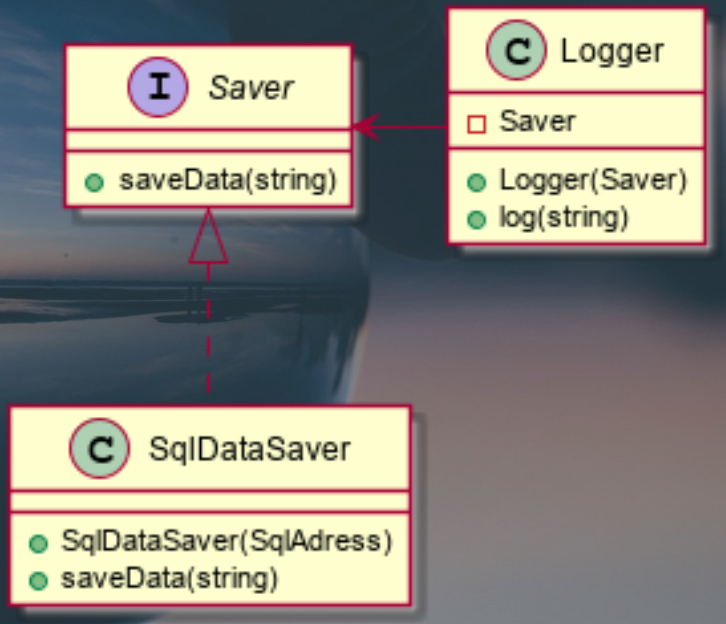
- Projekt wymaga logowania zdarzeń do bazy danych.
- Programiści zdecydowali korzystać z SQL.
- Szczegóły są enkapsulowane w klasie `SqlDataSaver`.
- Klasa `Logger` bezpośrednio wykorzystuje `SqlDataSaver`.



# Rozwiązanie

- Pozbywamy się niepotrzebnej zależności poprzez wprowadzenie interfejsu (klasy abstrakcyjnej).
- Teraz Logger nie zalezy od niskopoziomowych szczegółów implementacji.



# Posumowanie

- „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).
  - Podczas projektowania podejmowane są wybory bibliotek niskopoziomowych.
  - Za mało interfejsów. Zmiany w kodzie pociągają za sobą wyrównywanie UT.
- Cechy kodu nienaruszającego zasady odwrócenia zależności:
  - Kod jest modułowy
  - Brak zależności pomiędzy modułami, klasami.
  - Obiekty komunikują się poprzez interfejsy.
  - Implementacja zależy 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.”



NOKIA

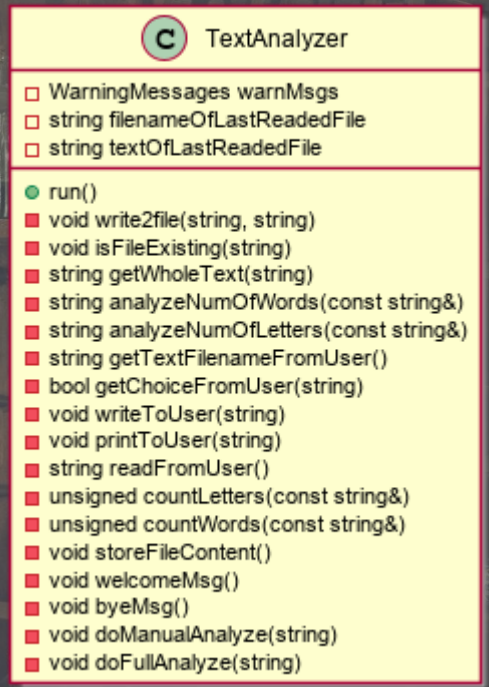
# Single responsibility principle (Zasada jednej odpowiedzialności)



NOKIA

# Problem

- Klasa `TextAnalyzer` zlicza słowa lub litery w pliku tekstowym.
- Wynik analizy może być zapisany do pliku lub wyświetlony na konsoli.
- Brak logicznego podziału zadań na komunikujące się obiekty.
- Antywzorzec The Blob.



# Rozwiązanie

NOKIA

# Posumowanie

- „zapachy” towarzyszące naruszeniu zasady jednej odpowiedzialności:
  - Pojawienie się w kodzie antywzorca „The God class”.
  - Kod jest „sztywny”
  - Kod staje się „kruchy”

- Cechy kodu nienaruszającego zasady jednej odpowiedzialności:
  - Nie istnieje więcej niż jeden powód do modyfikacji danej klasy
  - Jedna odpowiedzialność klasy nie znaczy że musi ona mieć tylko jedną metodę!
  - Kod jest łatwo rozwijalny przez dużą grupę programistów (małe prawdopodobieństwo konfliktów).
  - Ułatwione ponowne wykorzystanie klas w innym miejscu.

# Single responsibility principle (Zasada jednej odpowiedzialności)

„Powód do modyfikacji klasy powinien być tylko jeden.”

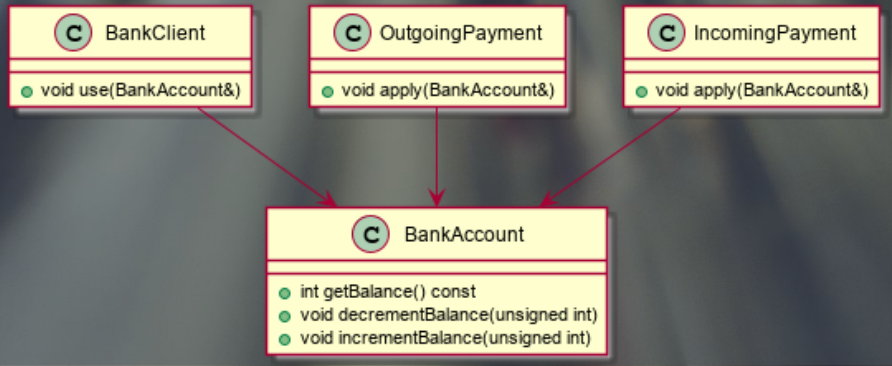
# Interface segregation principle (Zasada segregacji interfejsów)

NOKIA

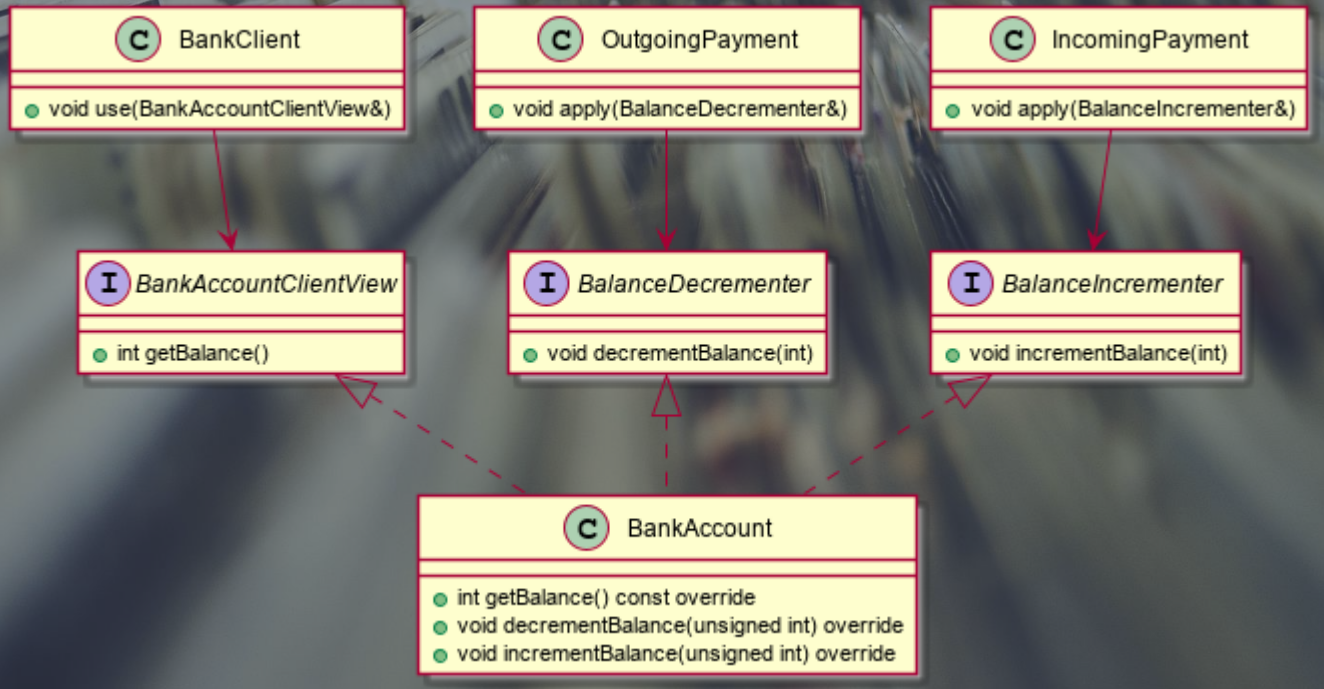
# Problem

- Klasa `BankAccount` ma wiele metod używanych przez wielu klientów.
- Każdy z klientów wykorzystuje tylko część metod.
- Modyfikując dowolną metodę musimy przekompilować wszystkich klientów.
- Nie możemy jej podzielić ze względu na implementację.



# Rozwiązanie



# Posumowanie

- „zapachy” towarzyszące naruszeniu zasady segregacji interfejsów:
  - Klasy zależą od metod, których nie używają.
  - Duża podatność na błędy jeśli klasa kliencka ma dostęp do metod, do których nie powinna mieć dostępu. (Kod jest kruchy)
  - Brak interfejsów (klas abstrakcyjnych)
  - Jeśli interfejsy (klasy abstrakcyjne) istnieją, to są dokładnie tym samym zbiorem metod co implementujące je klasy.
- Cechy kodu nienaruszającego zasady segregacji interfejsów:
  - Żadna klasa nie jest zależna od metod z których nie korzysta.
- Uwaga:
  - ISP często powinien być pierwszym krokiem podczas refaktoryzacji antywzorca „The God class”.

# Interface segregation principle (Zasada segregacji interfejsów)

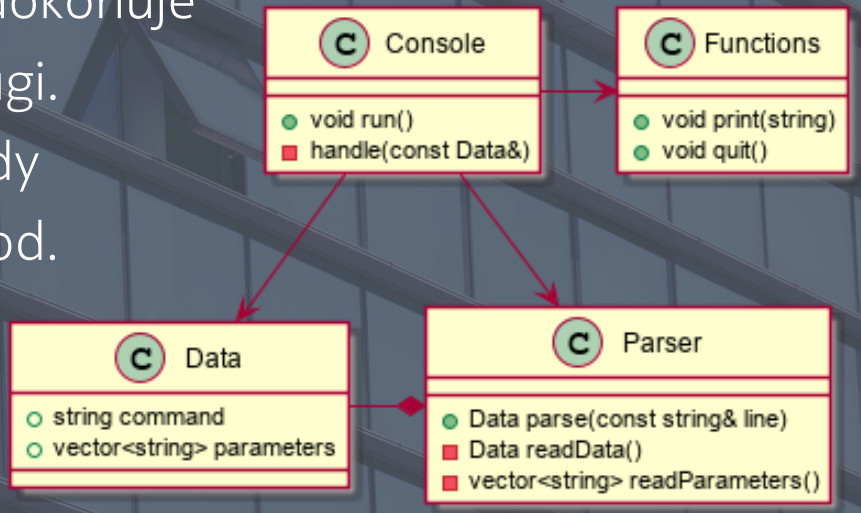
„Klienci nie powinni być zależni od metod, których nie używają.”

# 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.
- Aby dodać obsługę nowej komendy należy zmodyfikować istniejący kod.



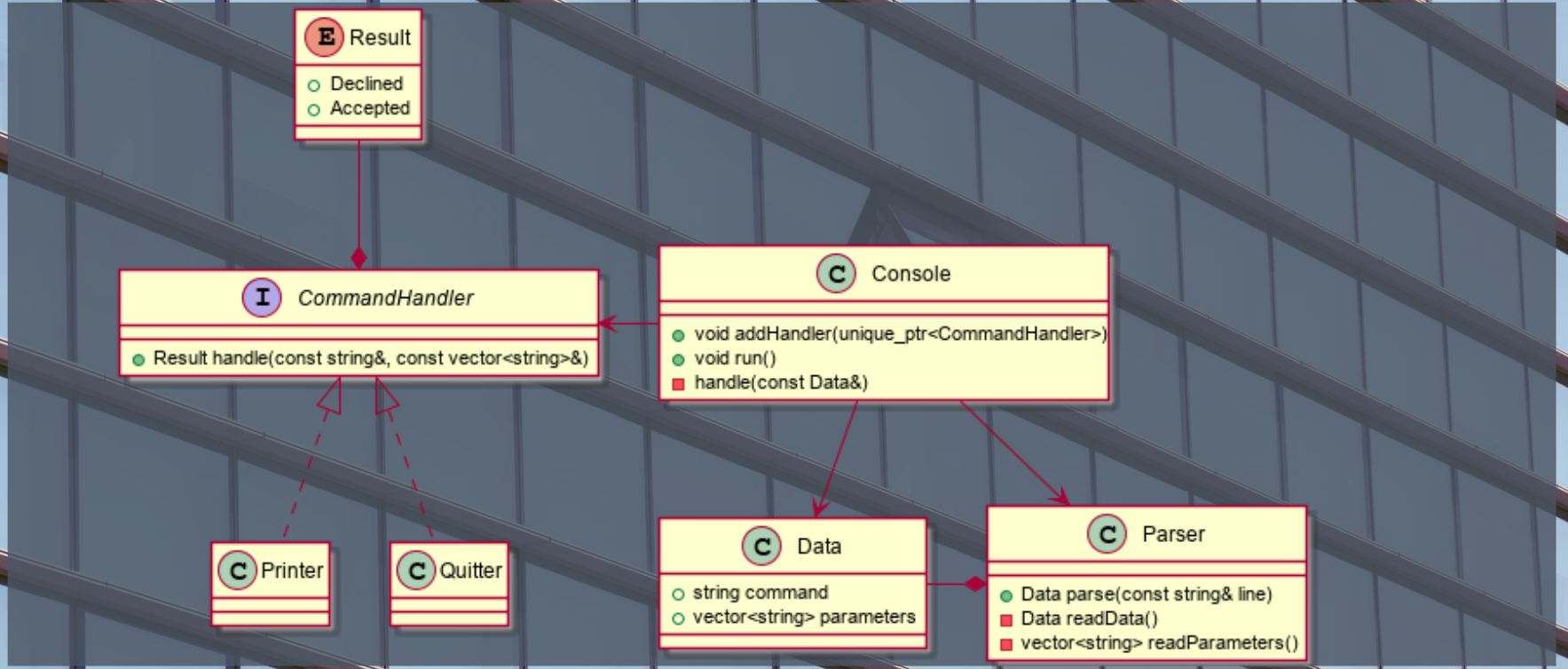

  

# Jak być jednocześnie otwartym i zamkniętym?

- Oprogramowanie powinno być otwarte na dodawanie nowych funkcjonalności.
- Oprogramowanie powinno być zamknięte na zmiany w już istniejącym kodzie.
- Jeśli pojawiają się nowe wymagania powinniśmy móc je 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 zmniejsza czytelność kodu.

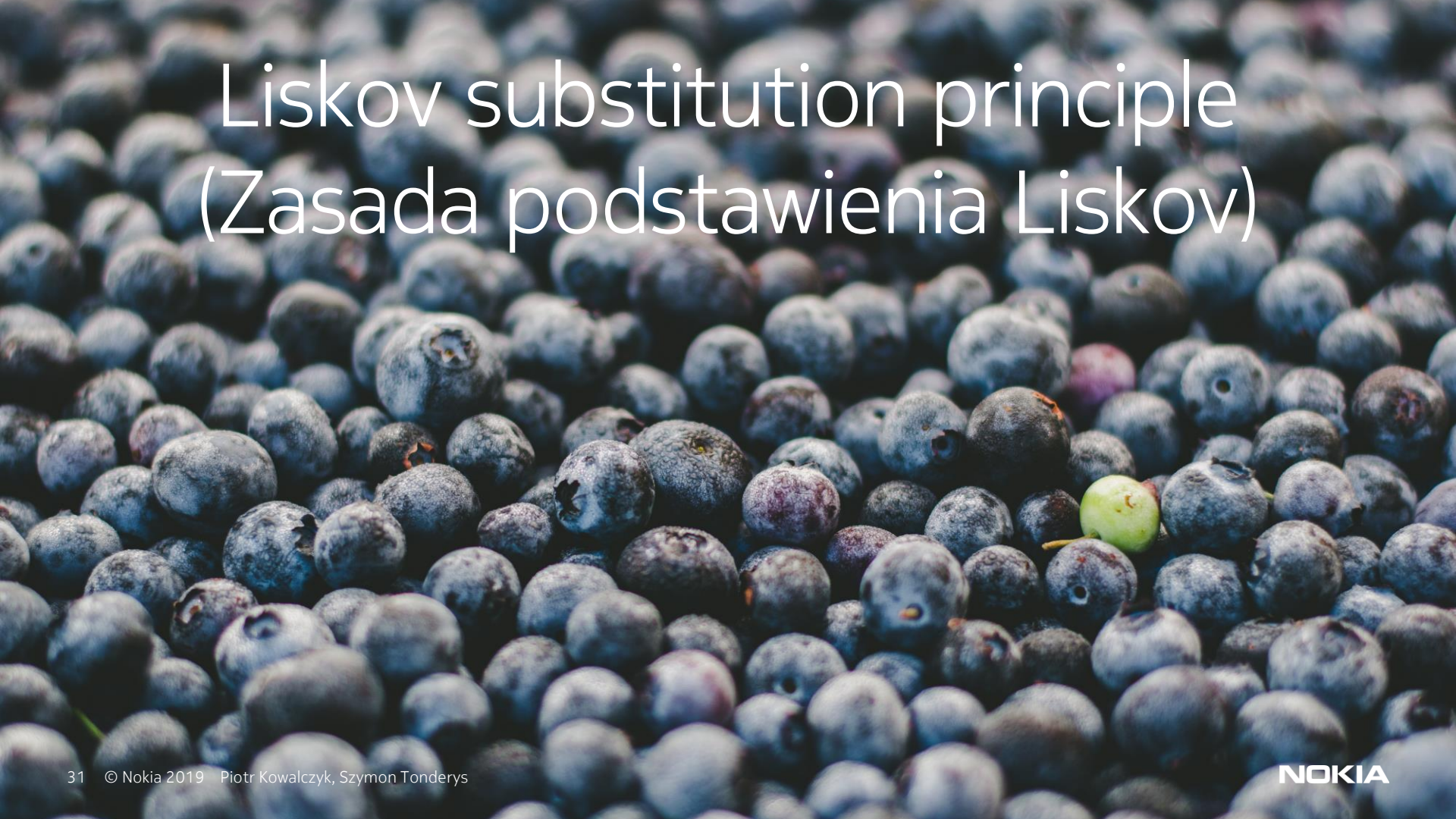
- Tyczy się to każdej reguły SOLID'a, jednak w przypadku Open/Closed Principle najłatwiej „przekroczyć granicę".

# Open - closed principle (Zasada otwarte - zamknięte)

„Przy zmianie wymagań nie powinien być zmieniany stary, działający kod, ale dodawany nowy, który rozszerza zachowania.”

# Liskov substitution principle (Zasada podstawienia Liskov)

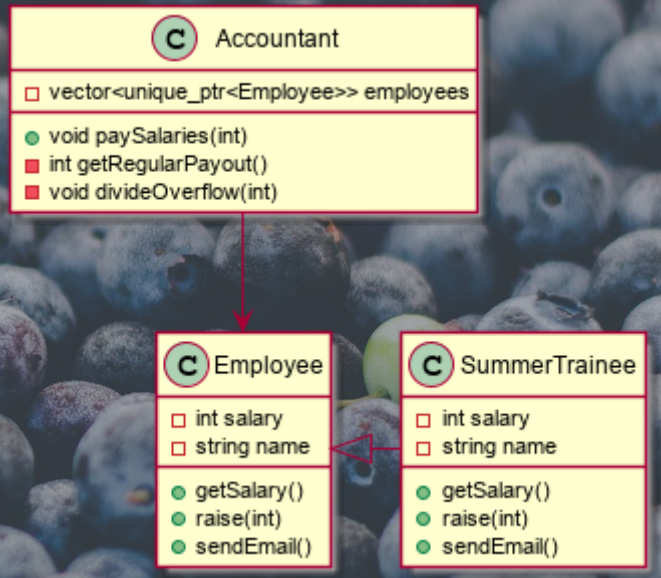


NOKIA

# Problem

- Każdy obiekt `Employee` zna swoje wynagrodzenie.
- Obiekty `SummerTrainee` nie otrzymują wynagrodzenia.
- Klasa `Accountant` zarządza wypłatami dla pracowników firmy.
- Jeśli kwota przeznaczona na wypłaty jest większa niż suma wynagrodzeń `Accountant` rozdziela nadwyżkę na wszystkich pracowników.



# Problem

- Kod reprezentuje prawdziwe obiekty.
- Reprezentacja (kod) nie musi współdzielić relacji, które istnieją pomiędzy reprezentowanymi obiektami.

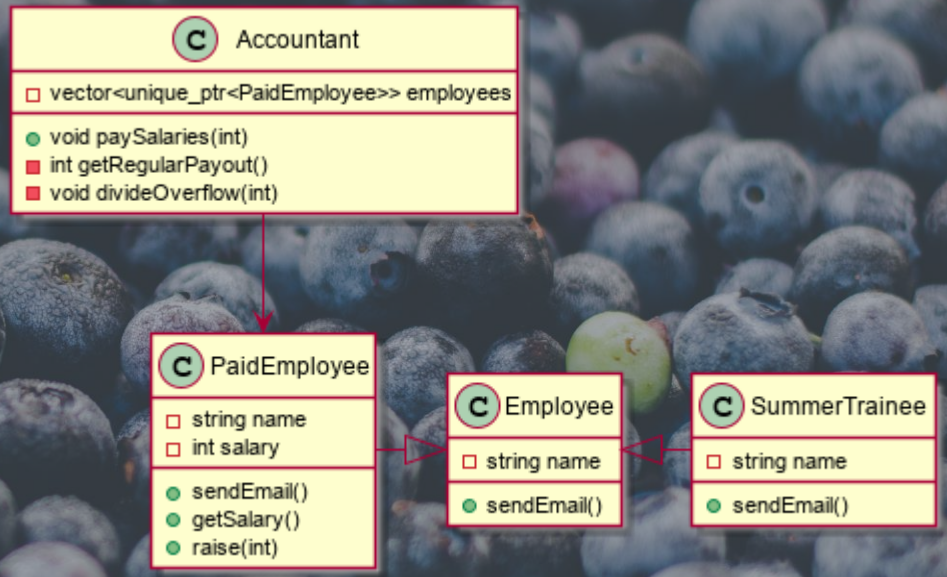
```
class Rectangle
{
    ...
}
```

```
class Square
{
    ...
}
```

# Rozwiązanie

- SummerTrainee oraz PaidEmployee dziedziczy ze wspólnego interfejsu Employee.
- SummerTrainee nie posiada zdegenerowanych metod `raise()` i `getSalary()`



# Podsumowanie

- „zapachy” towarzyszące naruszeniu zasady podstawienia Liskov:
  - RTTI
  - Pojawianie się niepożądanych zależności między klasami.

- Cechy kodu nienaruszającego zasady podstawienia Liskov:
  - Korzystanie z klasy pochodnej jest takie samo jak korzystanie z klasy bazowej - kod zachowuje się poprawnie po podstawieniu dowolnego typu pochodnego w miejsce bazowego.
  - Brak potrzeby znajomości typu.
  - Brak zależności pomiędzy klasami pochodnymi a klientem.

# Liskov substitution principle (Zasada podstawienia Liskov)

„Musi być możliwość podstawienia typów pochodnych za ich typy bazowe.”

NOKIA