# Tworzenie SOLIDnego kodu obiektowego w C++

Praktyczne Aspekty Rozwoju Oprogramowania


- Piotr Kowalczyk, Piotr Śliwa

- 13-04-2016




NOKIA

# Kwestie organizacyjne

- Kurs Praktyczne Aspekty Rozwoju Oprogramowania został przygotowany przez pracowników firmy Nokia Networks.
- Uczestnicy kursu otrzymają dawkę wiedzy z zakresu rozwoju oprogramowania wraz z przykładami zastosowań praktycznych.
- Harmonogram oraz materiały dodatkowe są dostępne w serwisie Moodle:
  - PWr: http://pst.pwr.wroc.pl/moodle/
  - UWr: http://kno.ii.uni.wroc.pl/ii/
- Prosimy o wyciszenie telefonów oraz wpisanie się na listę obecności.

# Agenda

- Wartości oprogramowania
- Dobre praktyki projektowania zorientowanego obiektowo
- Zasada odwrócenia zależności
- Zasada pojedynczej odpowiedzialności
- Zasada segregacji interfejsów
- Przerwa (10min)
- 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 opiujący 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)

# Dependency inversion principle

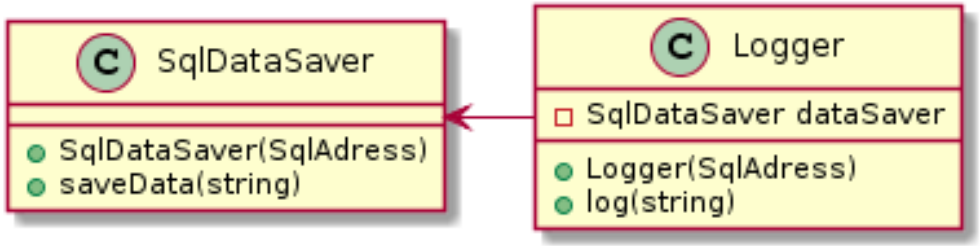
# (Zasada odwrócenia zależności)




NOKIA

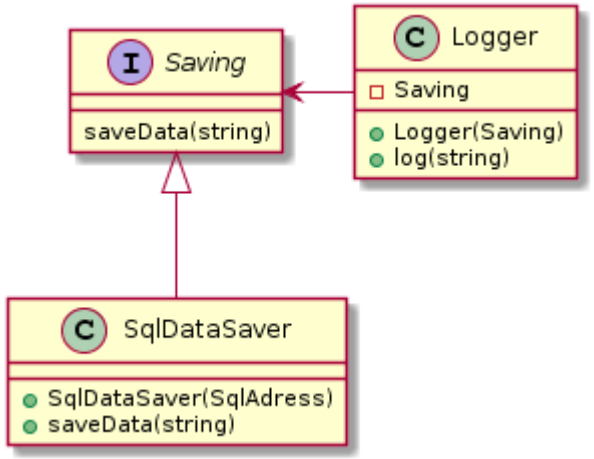
# Dependency inversion principle
## Problem

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



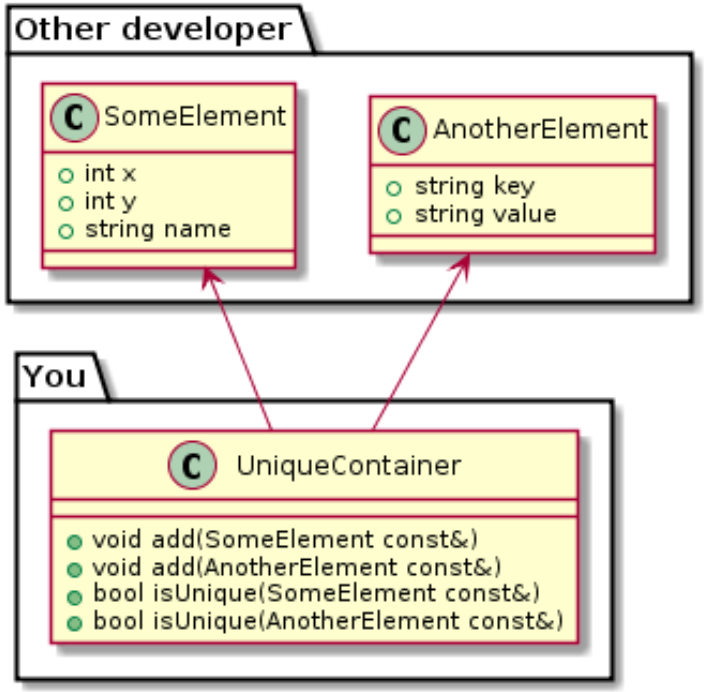
# Dependency inversion principle
## Rozwiązanie

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



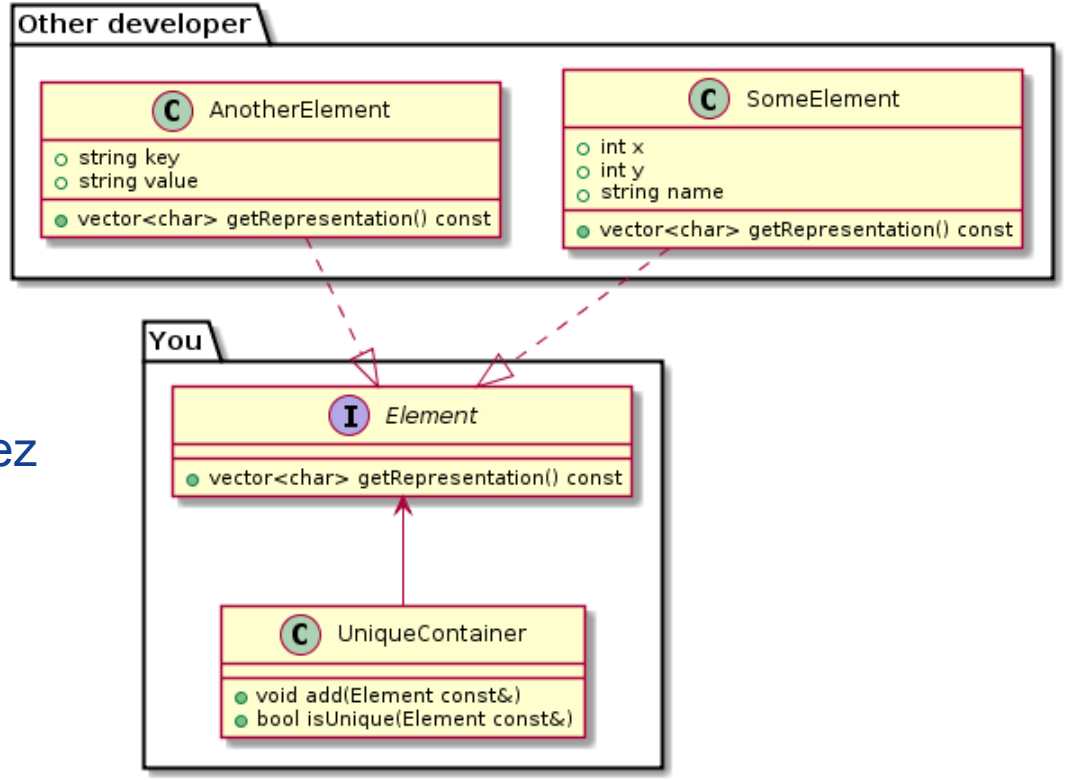
# Dependency inversion principle
## Inny przykład

- Dwóch programistów rozwija projekt.
- Kod rozwijany wspólnie jest od siebie zależny.
- Dodawanie kolejnych elementów powoduje konieczność modyfikacji klasy `UniqueContainer`.



# Dependency inversion principle

## Inny przykład




- Każdy z programistów może rozwijać swoją część kodu bez wpływu na inne.
- Wszystkie klasy zależą teraz od abstrakcji.

# Dependency inversion principle
## Posumowanie

- „zapachy” towarzyszące naruszeniu zasady odwrócenia zależności:
  - Dużo zależności w kodzie.
  - Kod staje się sztywny.
  - Podczas projektowania podejmowane są wybory bibliotek niskopoziomowych.
  - Brak/mało interfejsów.
- 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 wyskokopoziomowe nie powinny zależeć od modułów niskopoziomowych. Jedne i drugie powinny zależeć od abstrakcji.”

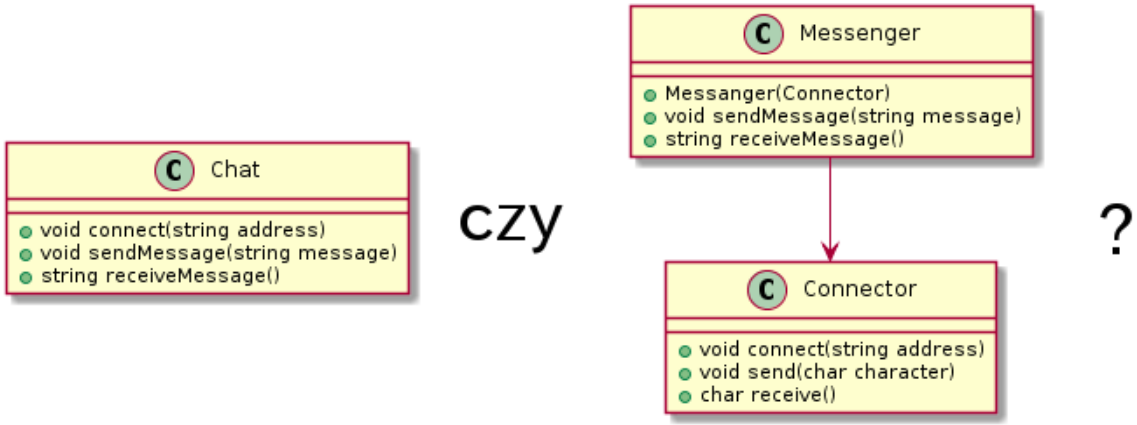
# Single Responsibility Principle

# (Zasada jednej odpowiedzialności)



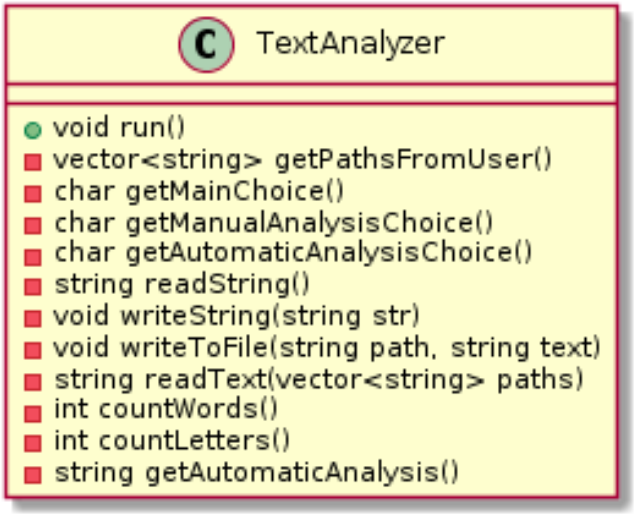

NOKIA

# Single responsibility principle
## Problem



# Single responsibility principle
## Problem

- Klasa `TextAnalyzer` ma zbyt wiele odpowiedzialności.
- Zmiana dowolnej metody wymaga rekompilacji całej klasy `TextAnalyzer`, oraz wszystkich klas, które niej zależą.
- Brak logicznego podziału zadań na klasy.



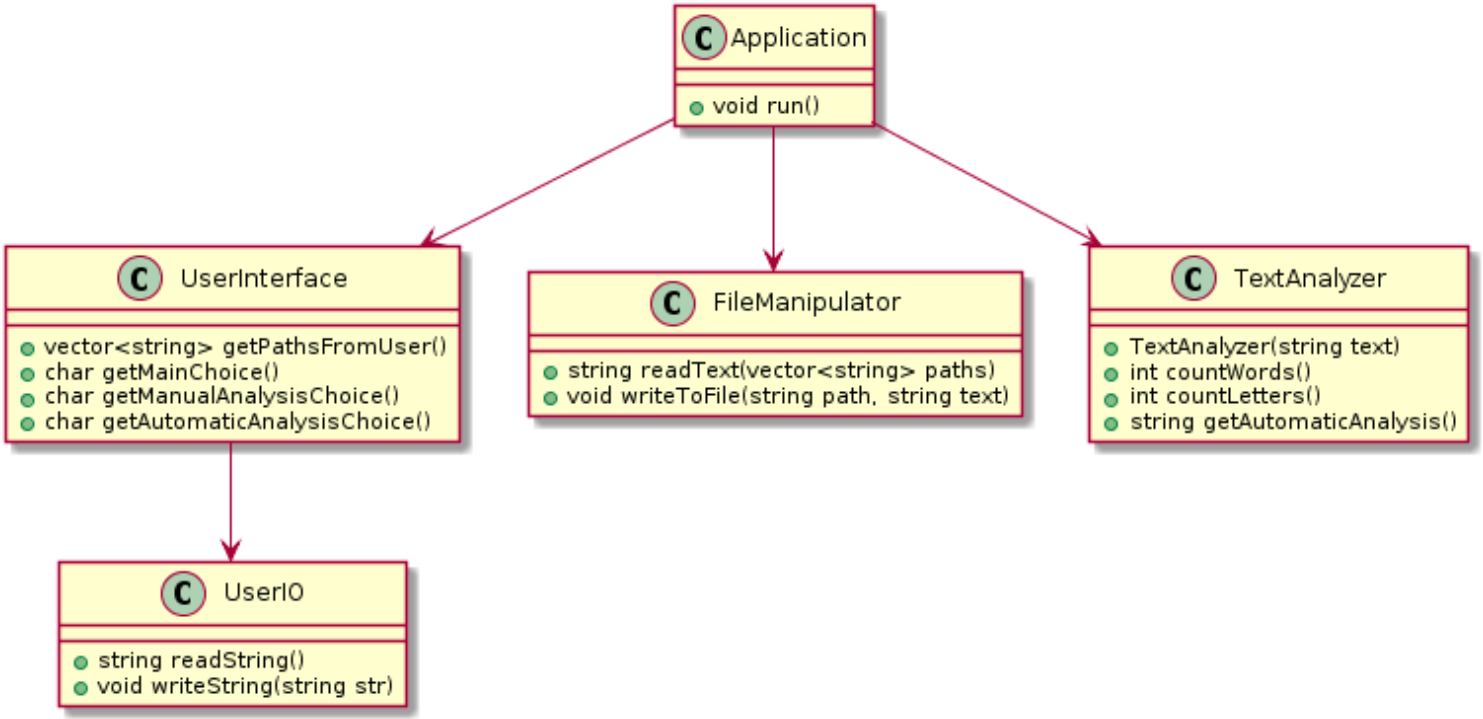
# Single responsibility principle

## Co to jest jedna odpowiedzialność?

- Odpowiedzialność jest to zbiór operacji, które odpowiadają potrzebie jednego aktora.
- Aktorem może być np.
  - logika biznesowa,
  - architektura (połączenie sieciowe)
  - warstwa prezentacji (generowanie raportów, wyświetlanie danych)
  - warstwa utrwalania (baza danych)
- Innymi słowy: klasa posiada jedną odpowiedzialność, jeśli istnieje tylko jeden powód, dla którego będziemy ją zmieniać.

# Single responsibility principle

## Przykładowe rozwiązanie






NOKIA

# Single responsibility principle
## Posumowanie

- „zapachy” towarzyszące naruszeniu zasady jednej odzpowiedzialności:
  - Istnienie antywzorca „The God class”.
  - Kod staje się „sztywny” – bardzo trudne staje wprowadzanie zmian.
  - Kod staje się „kruchy” – drobna zmiana może spowodować wiele błędów w nieoczekiwanych miejscach.
- Cechy kodu nienaruszającego zasady jednej odpowiedzialności:
  - Nie istnieje więcej niż jeden powód do modyfikacji danej klasy
  - Jedna odpowiedzialność klasy nie znaczy że może 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)

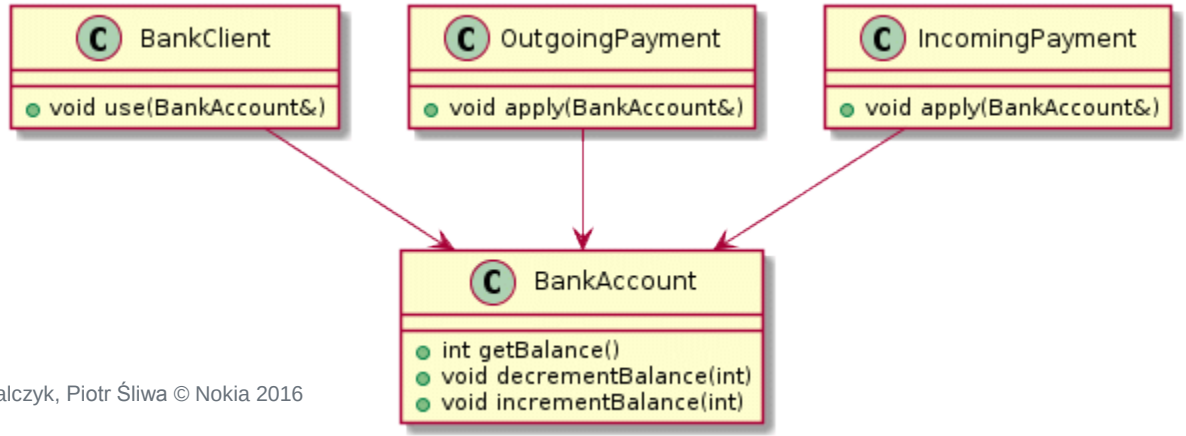
„Powód do modyfikacji klasy powinien być tylko jeden.”

# Interface Segregation Principle

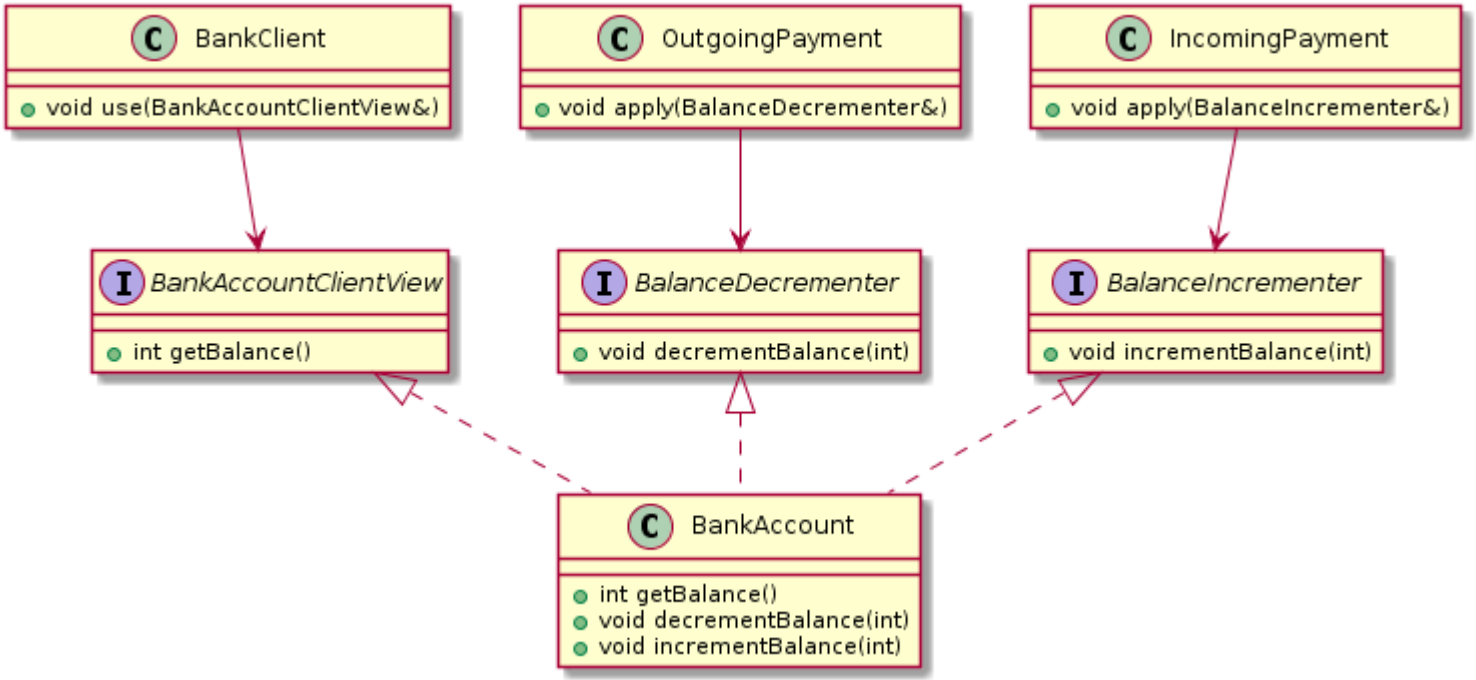
# (Zasada segregacji interfejsów)

# Interface segregation principle
## Problem

- Klasa `BankAccount` ma wiele metod używanych przez wielu klientów.
- Każdy z klientów wykorzytuje tylko część metod, a z pozostałych nie powinien korzystać.
- Modyfikując dowolną metodę musimy przekompilować wszystkich klientów.
- Nie możemy jej podzielić ze względu na implementację.



# Interface segregation principle
## Rozwiązanie



# Interface segregation principle
## Posumowanie

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

# Interface Segregation Principle
# (Zasada segregacji interfejsów)

„Klienci nie powinni być zmuszeni do zależności od metod, których nie używają.”



NOKIA

# Open/Closed Principle

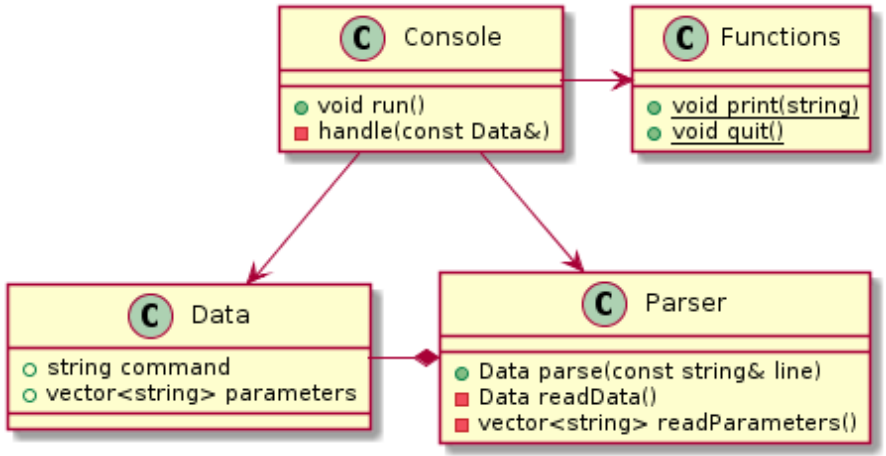
# (Zasada otwarte/zamknięte)




NOKIA

# Open/closed principle
## Problem

- Aplikacja `Console` obsługująca komendy: „**print**” oraz „**quit**”.
- Klasa `Parser` rozpoznaje rodzaj komendy oraz oddziela ją od argumentów.
- Metoda `Console::handle()` dokonuje wyboru właściwej funkcji obsługującej komendę.
- Pojawia się nowe wymaganie. Dodajemy nową komendę.
- Aby to zrobić należy zmodyfikować istniejący kod.

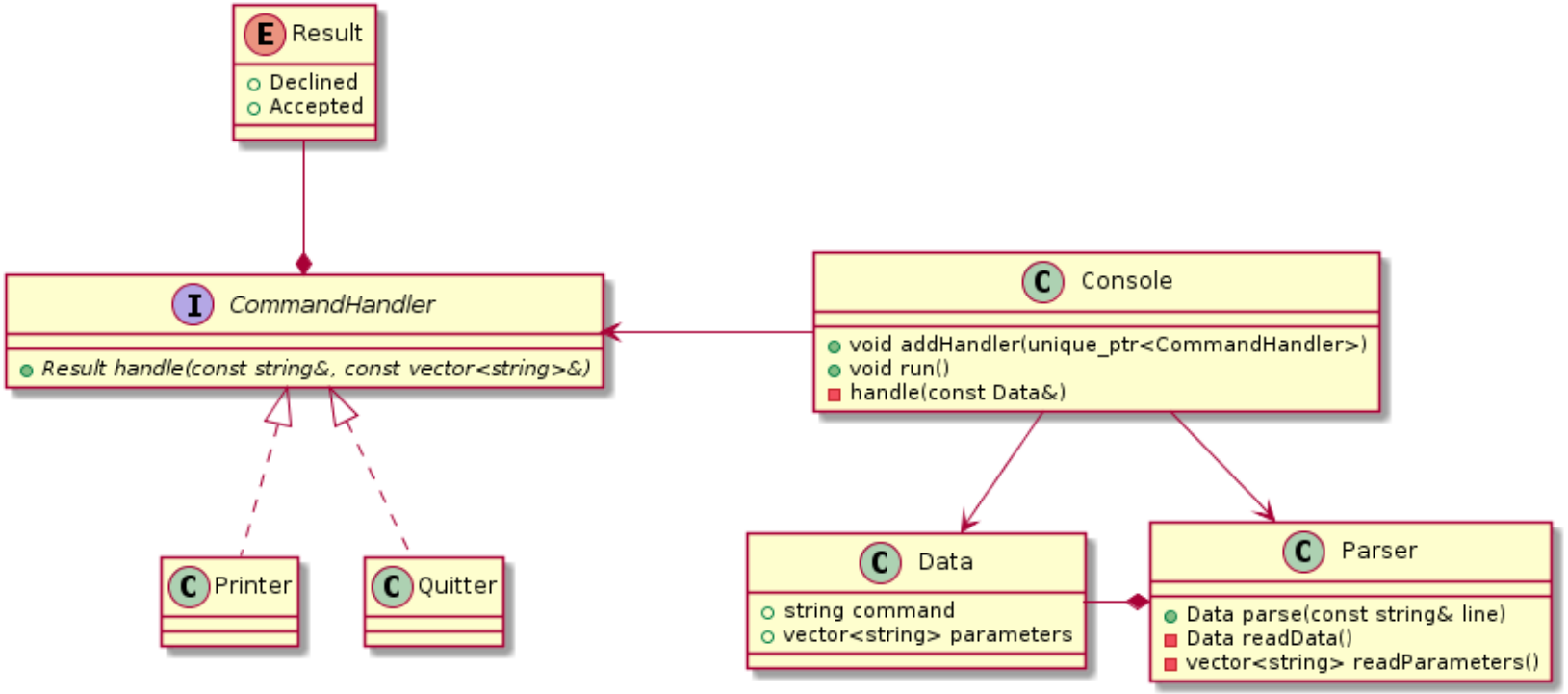


# Open/closed principle

## 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 już istniejącego.
- Aby zaprojektować kod w taki sposób musimy przewidzieć przyszłe zmiany.
- „Oś zmian jest osią zmiany, tylko wówczas, gdy zmiany rzeczywiście występują”.

# Open/closed principle
## Rozwiązanie



# Open/closed principle
## Posumowanie

- Cechy kodu nienaruszającego zasady otwarte/zamknięte:
  - Logika biznesowa jest enkapsulowana w pojedyńczych, 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.

- Wady:
  - Musimy wcześnie przewidzieć kierunek rozwoju oprogramowania.
  - Nadużywanie tej zasady wprowadza niepotrzebną złożoność kodu.

# 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

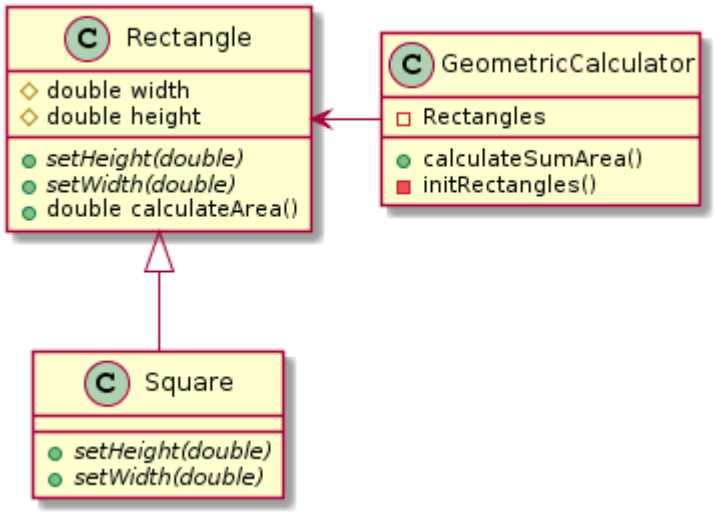
# (Zasada podstawienia Liskov)




NOKIA

# Liskov substitution principle
## Problem

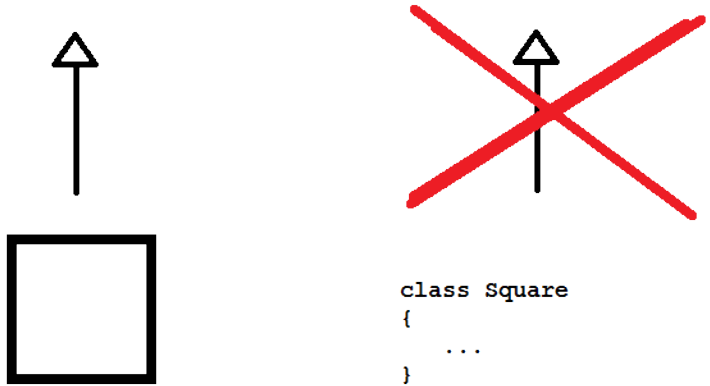
- Klasa `GeometricCalculator` wykorzystuje polimorficznie 2 obiekty: `Rectangle` oraz `Square`.
- Klasa `Square` dziedziczy z `Rectagle`, ponieważ każdy kwadrat jest prostokątem.



# Liskov substitution principle
## Zasada reprezentowalności

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

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



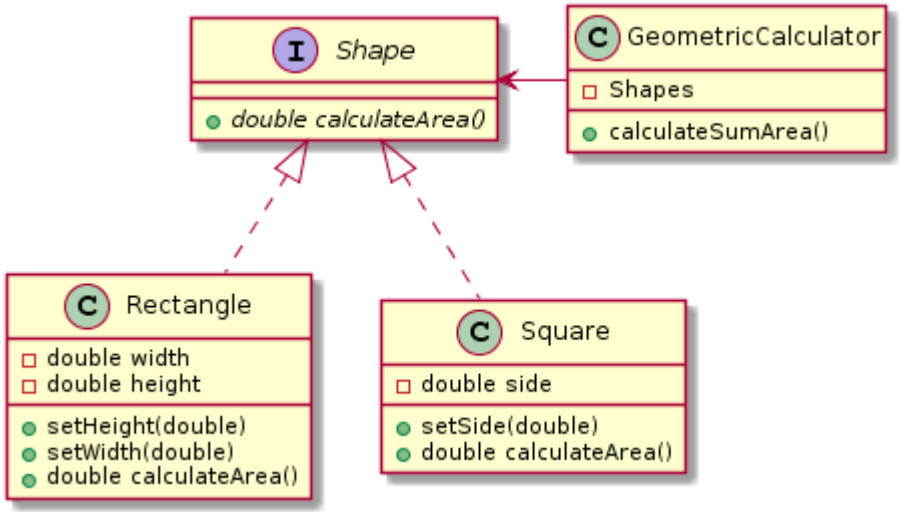
```
class Square
{
   ...
}
```



NOKIA

# Liskov substitution principle
## Rozwiązanie

- Ponieważ `Square` nie może dziedziczyć po `Rectangle` tworzymy interfejs `Shape`, który jest implementowany przez obydwie klasy.
- Tworzenie obiektów nie powinno być odpowiedzialnością klasy `GeometricCalculator`.
- Przenosimy tworzenie obiektów do „main part” lub dedykowanej fabryki.



# Liskov substitution principle
## Posumowanie

- „zapachy” towarzyszące naruszeniu zasady podstawienia Liskov:
  - RTTI.
  - Używanie instrukcji `switch/case`.
  - 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)

*„Poszukujemy następującej właściwości podstawiania: jeśli dla każdego obiektu $O_1$ typu S istnieje obiekt $O_2$ typu T taki, że dla wszystkich programów P zdefiniowanych w kategoriach T zachowanie P pozostanie niezmienione, gdy $O_1$ zostanie podstawione przez $O_2$, to S jest podtypem T.”*

Barbara Liskov

# Liskov Substitution Principle
# (Zasada podstawienia Liskov)

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