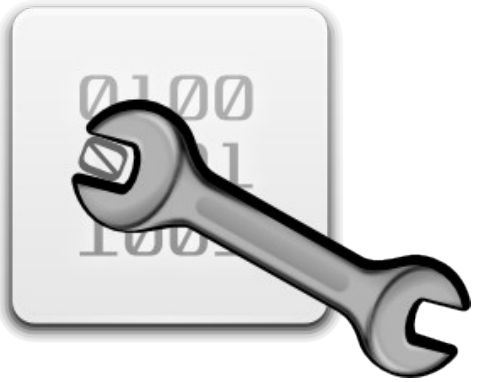
Podstawy inżynierii oprogramowania



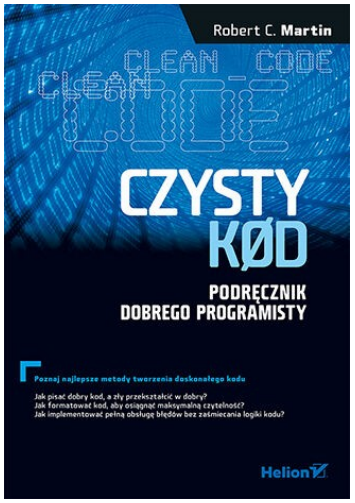
Zasady są po to, żeby się ich trzymać

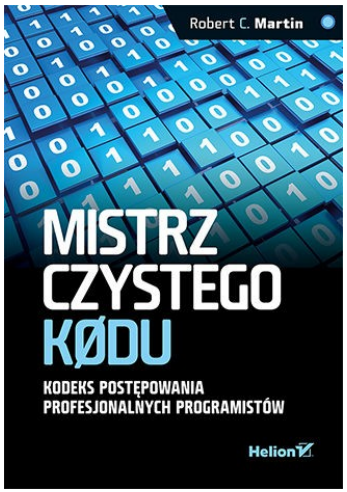
Aleksander Lamża
Wydział Nauk Ścisłych i Technicznych
Uniwersytet Śląski w Katowicach

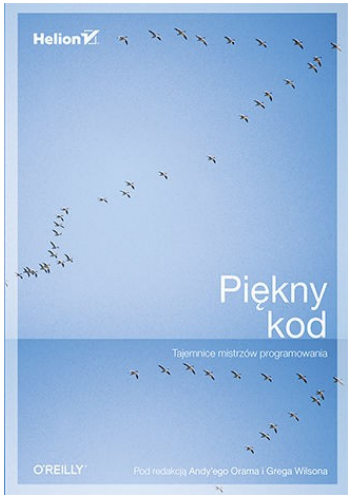
aleksander.lamza@us.edu.pl

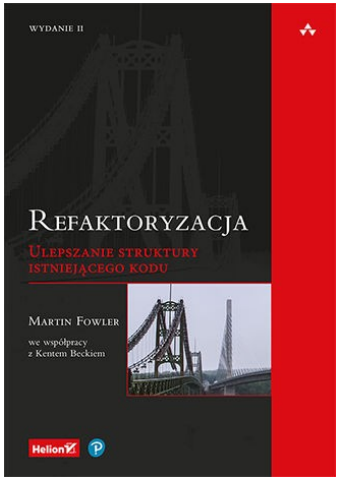
Zawartość

Literatura









Dużo przykładów w prezentacji pochodzi z tej książki.

SRP

Single Responsibility Principle

Reguła jednej odpowiedzialności

Często ta zasada jest rozumiana w taki sposób:

dany moduł powinien wykonywać tylko jedno zadanie

W zależności od kontekstu może to być funkcja, klasa, plik źródłowy...

Niezupełnie o to chodzi.

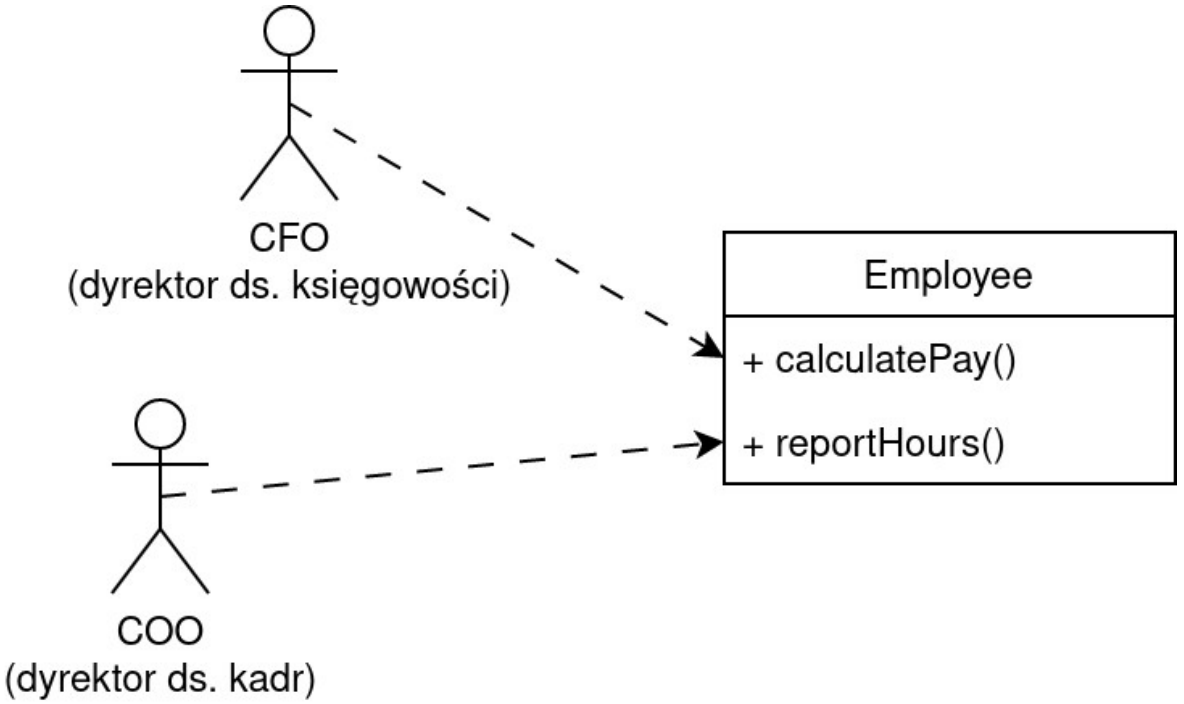
SRP

*Nigdy nie powinien istnieć więcej niż jeden **powód do zmiany**.*

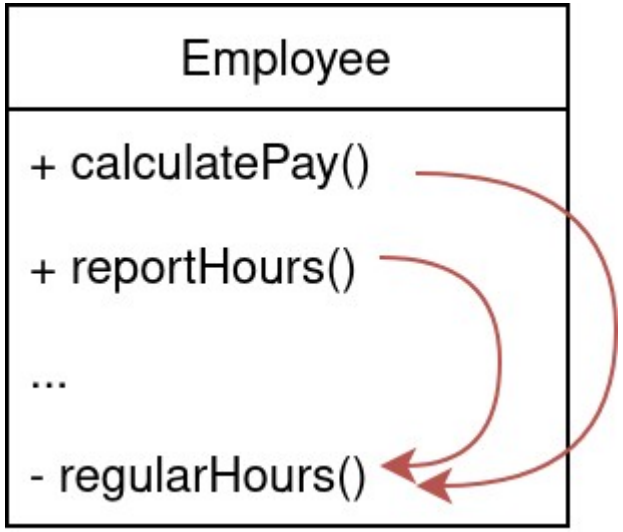
albo

*Każdy moduł powinien **odpowiadać przed jednym** i tylko jednym **aktorem**.*

Diagram przypadków użycia, UML... Pamiętacie, prawda?



SRP



DRY!

Jest bardzo prawdopodobne, że programiści – żeby nie powtarzać kodu – umieszczą kod wyliczający godziny pracy w prywatnej metodzie.

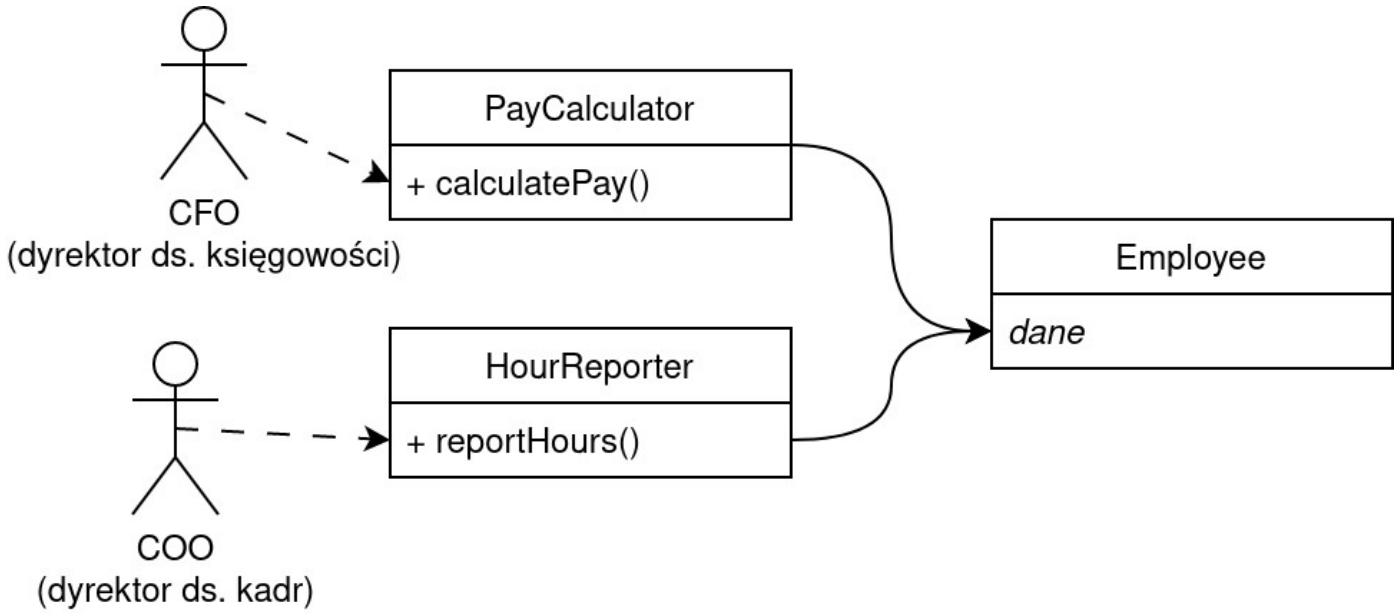
I bardzo dobrze!

Tylko co się stanie, jeżeli trzeba będzie zmienić sposób wyliczania godzin w związku ze zmianami w module księgowym?

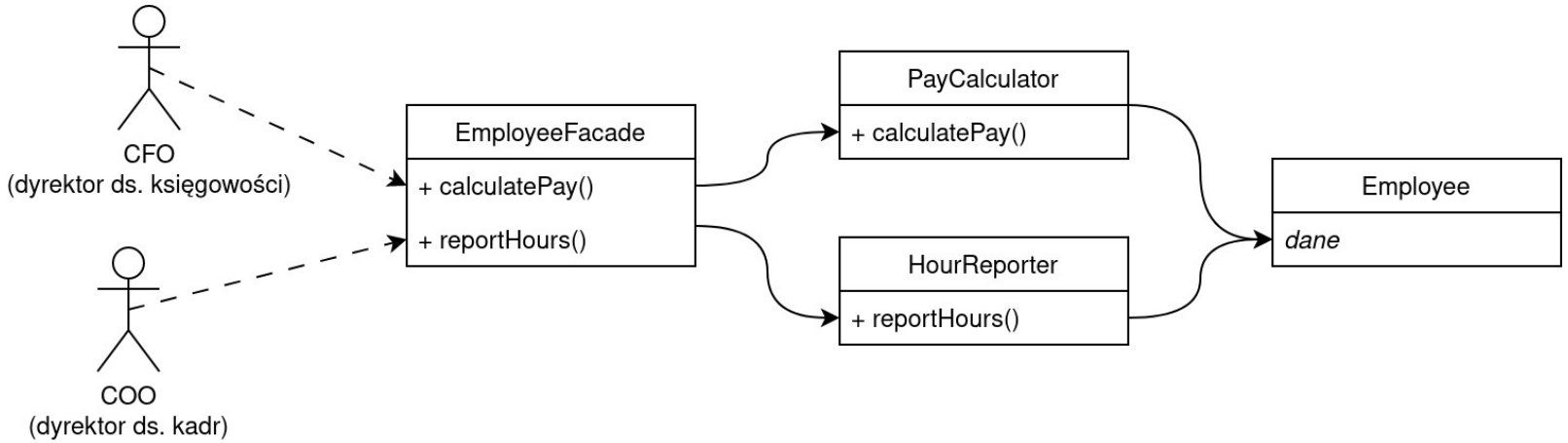
• • •

SRP

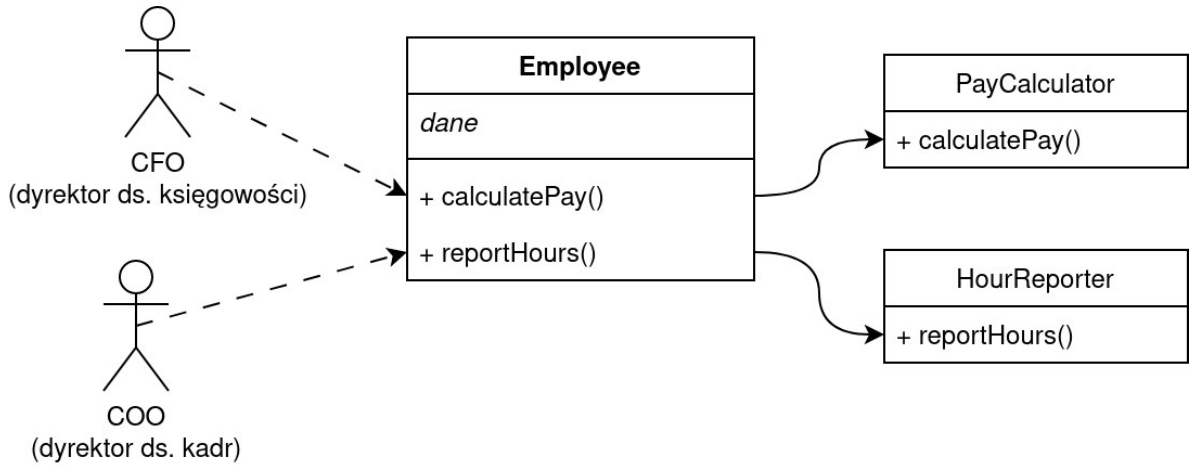
Rozwiązaniem może być oddzielenie metod od danych i przeniesienie metod do osobnych klas:



Modyfikacją ułatwiającą korzystanie z takiej struktury jest dodanie **fasady**:

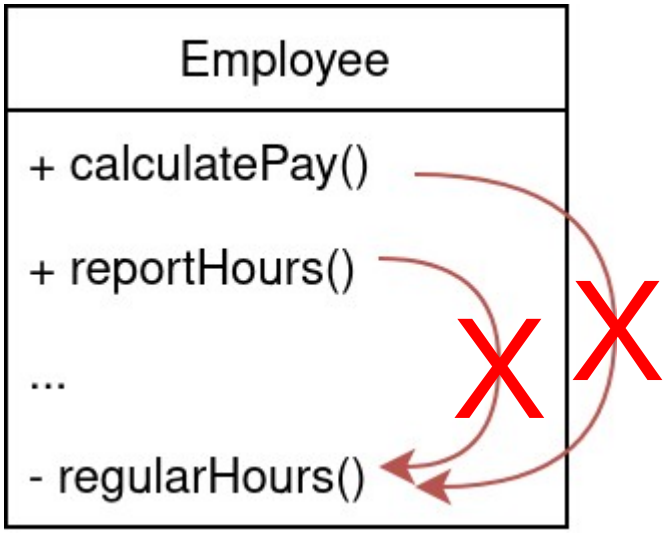


Można to uprościć, jeśli klasę Employee potraktujemy po części jak fasadę:



SRP vs DRY

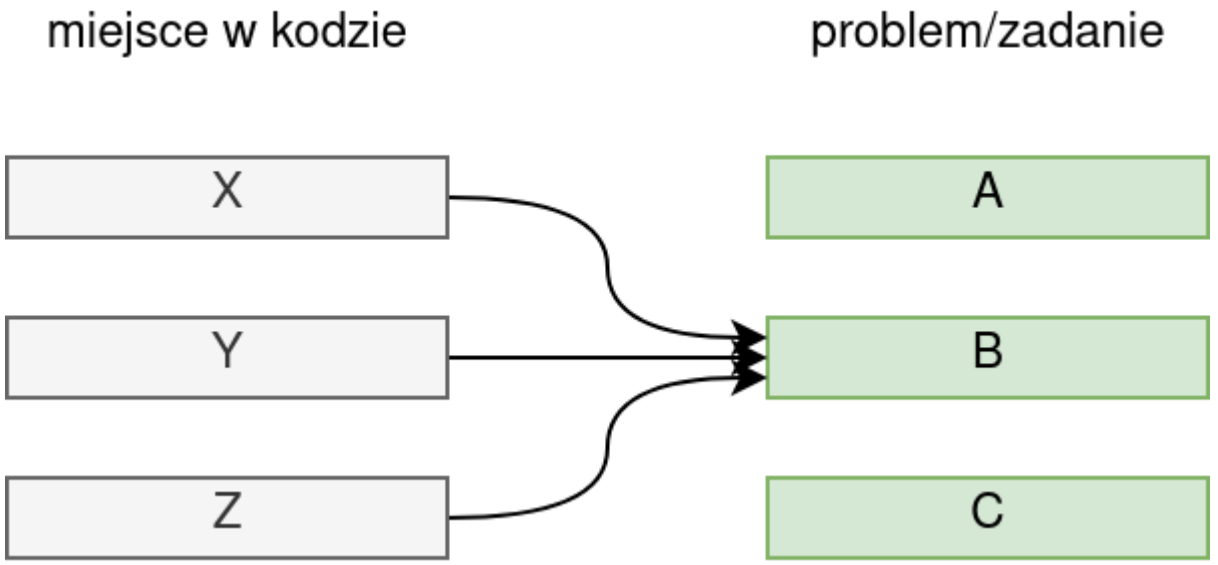
Na pierwszy rzut oka może się wydawać, że SRP stoi w sprzeczności z DRY.



Czy na pewno?

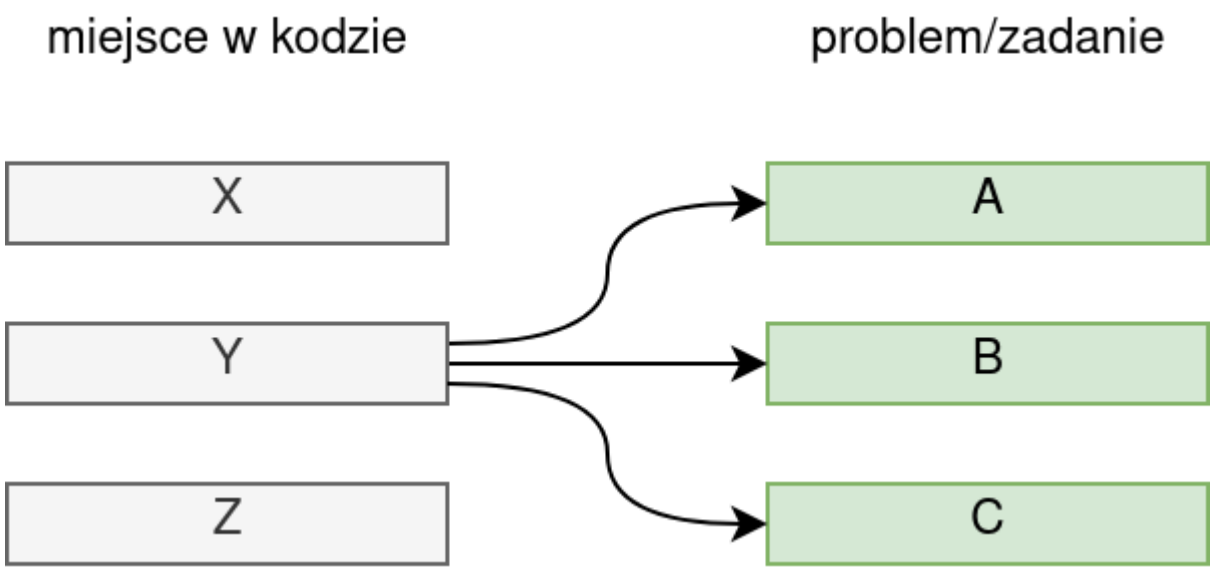
SRP vs DRY

Tak można sobie wyobrazić **złamanie reguły DRY**:



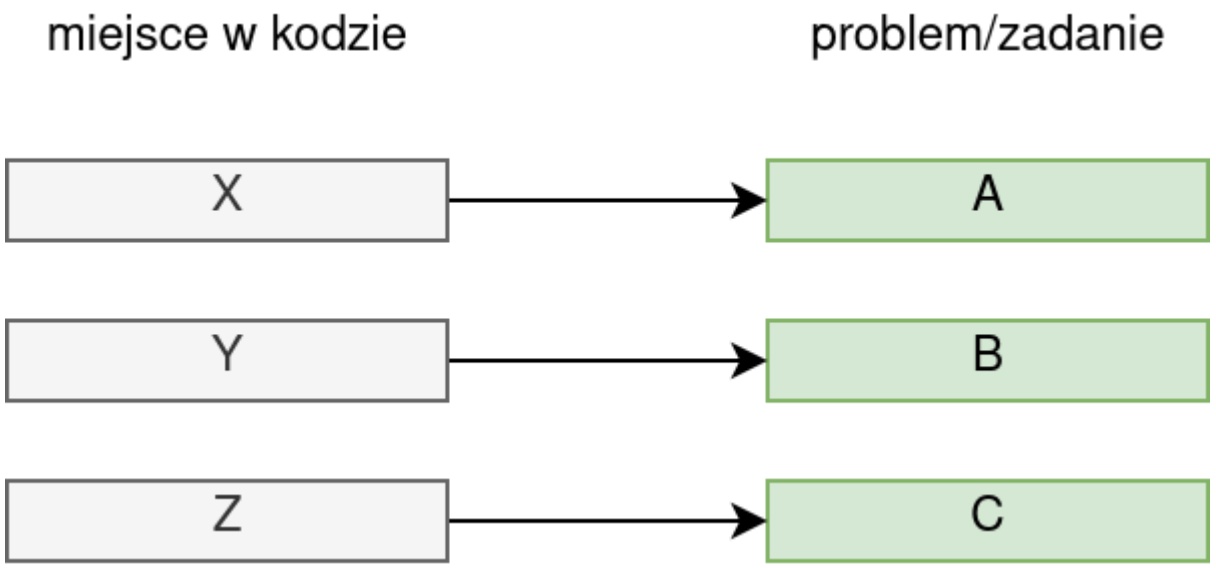
SRP vs DRY

A tak możne wyglądać **złamanie reguły SRP**:



SRP vs DRY

Idealną sytuację (pod kątem SRP i DRY) można sobie wyobrazić tak:



OCP

Open-Close Principle

Reguła otwarte-zamknięte

Elementy oprogramowania powinny być

otwarte na rozbudowę,

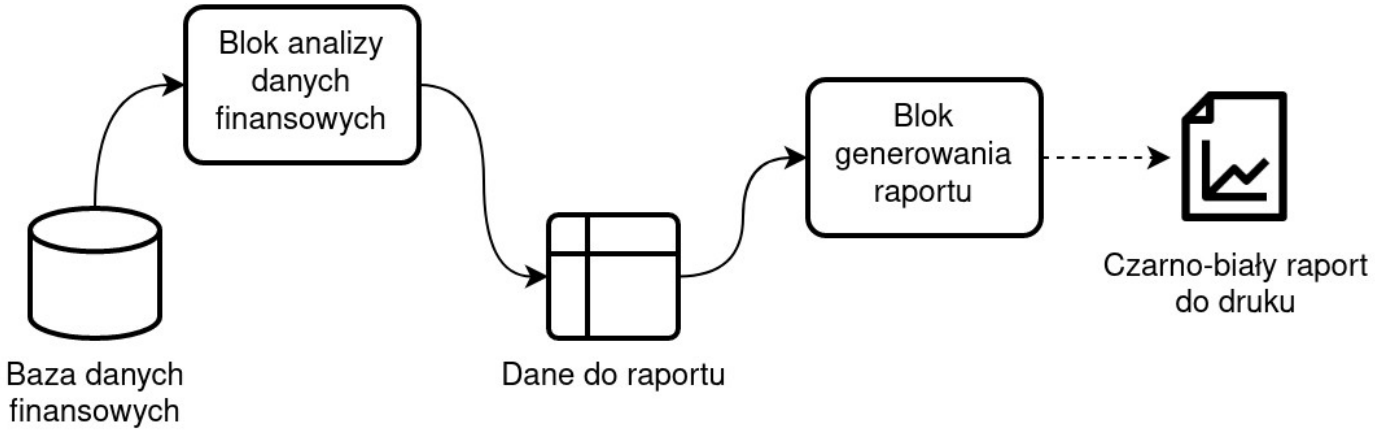
ale

zamknięte na modyfikacje.

OCP

Przykład:

W rozwijanym przez nas systemie mamy moduł odpowiedzialny za raporty.



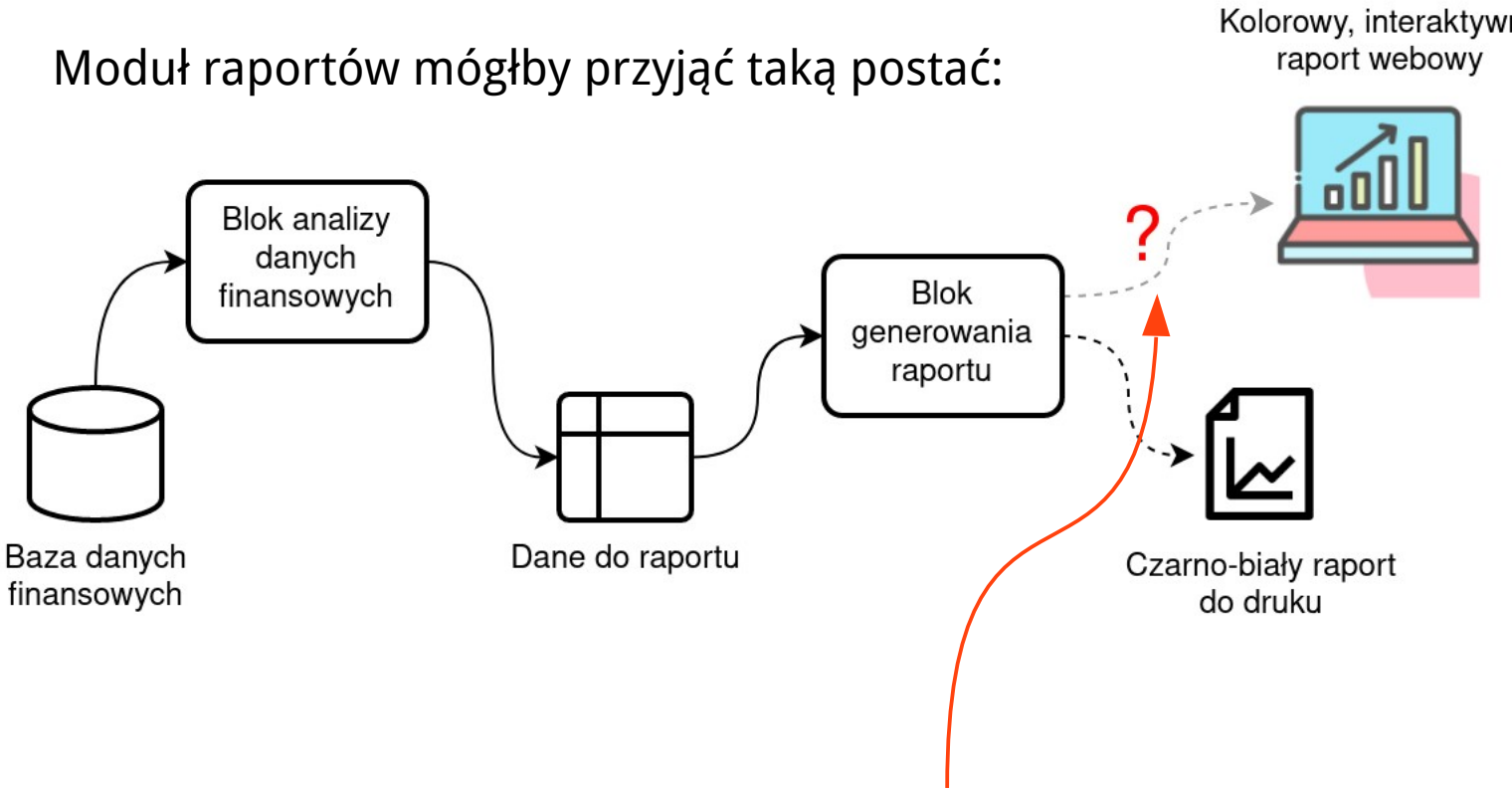
Klienci są znudzeni smętnymi czarno-białymi raportami: *Kto w tych czasach drukuje raporty!? Wskaźnik polityk proekologicznych w naszej dywizji osiąga levele poniżej dyrektyw. Mitygacją problemu może być tylko dedykowane rozwiązanie webowe!*

Po przetłumaczeniu z bełkotu na nasze: chodzi o **rozbudowę** modułu raportów o możliwość wyświetlania zestawień w przeglądarce internetowej.

Raporty muszą być oczywiście kolorowe i interaktywne.

OCP

Moduł raportów mógłby przyjąć taką postać:



Jeżeli zdecydujemy się na taką rozbudowę, musimy się przygotować na spore **modyfikacje istniejącego kodu**.

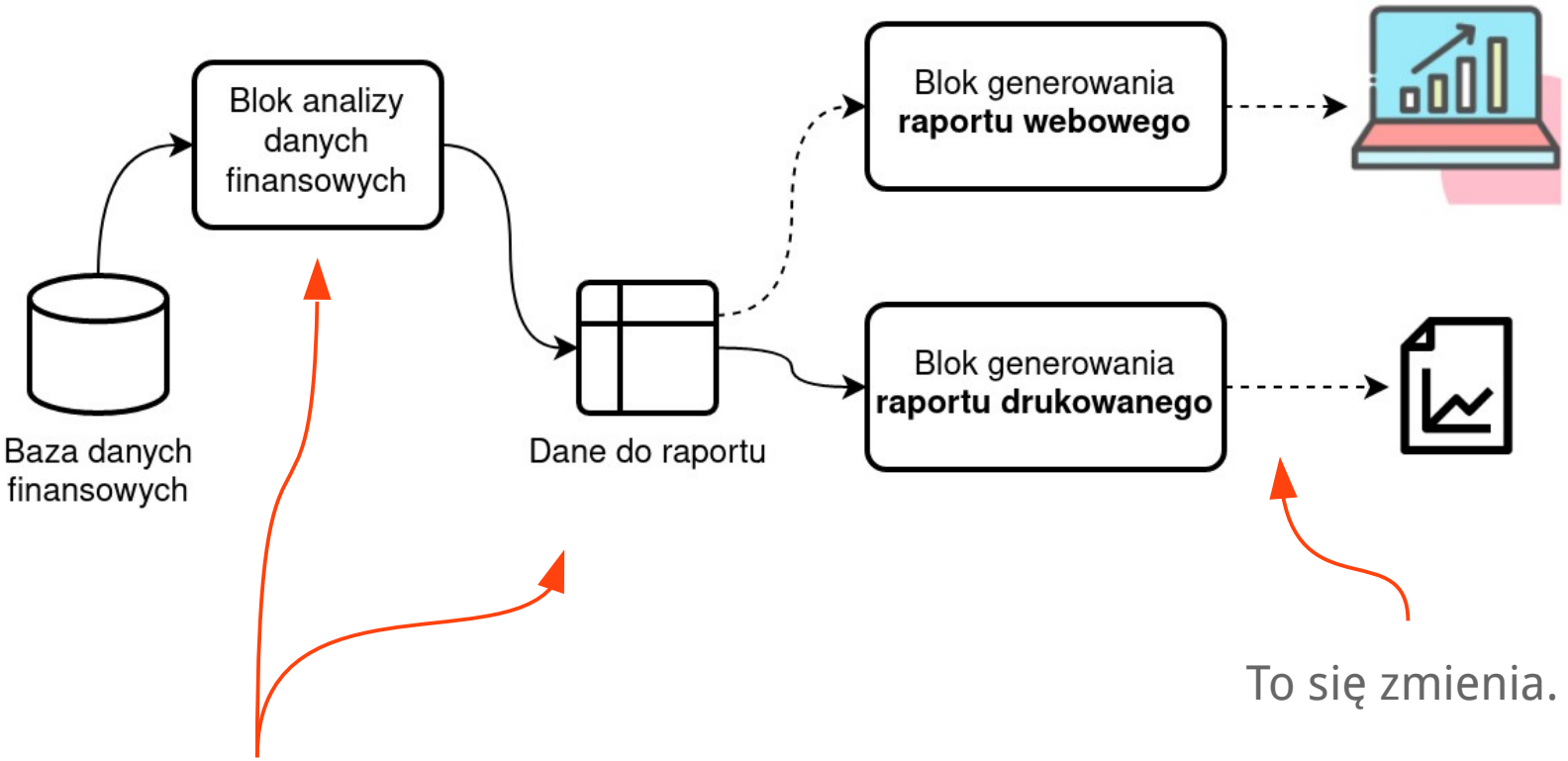
Musielibyśmy dostosować kod w bloku generowania raportu do nowego typu zestawień. Mnóstwo fajnych if-ów... A przy okazji łamiemy SRP!

Tego nie chcemy!

Kusząca jest myśl, żeby **zastąpić** poprzedni raport nowym.

Zawsze lepiej się zastanowić dwa razy (*Ogłupiałeś!? CEO musi mieć wszystko na papierze!*).

OCP



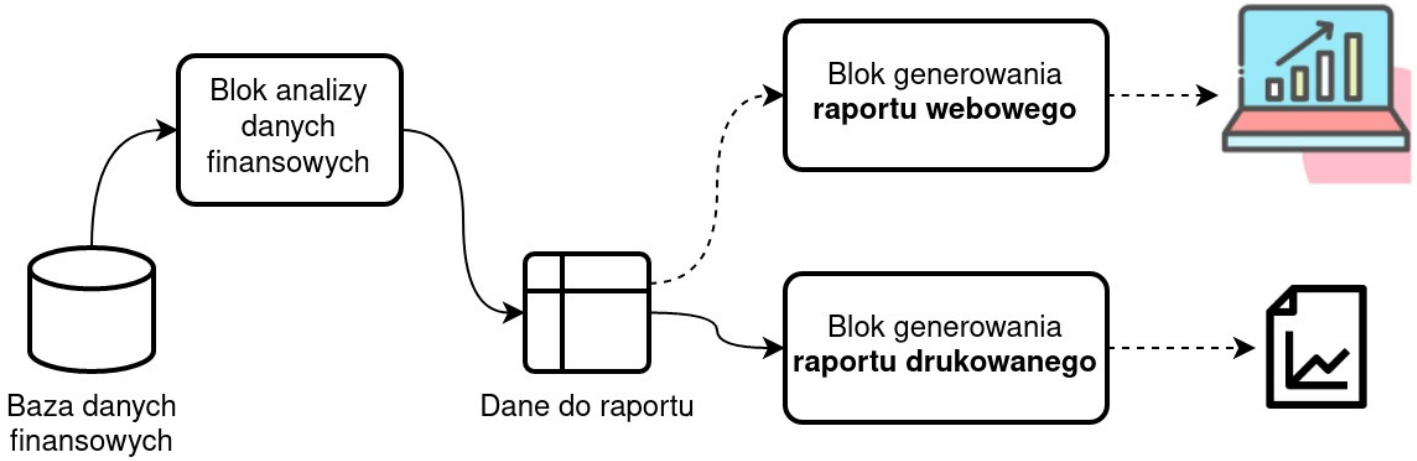
Musimy wyodrębnić elementy, które się zmieniają (zgodnie z SRP)

oraz

ustalić **hierarchię** wszystkich elementów.

Dzięki temu będzie można ustalić zależności między tymi elementami.

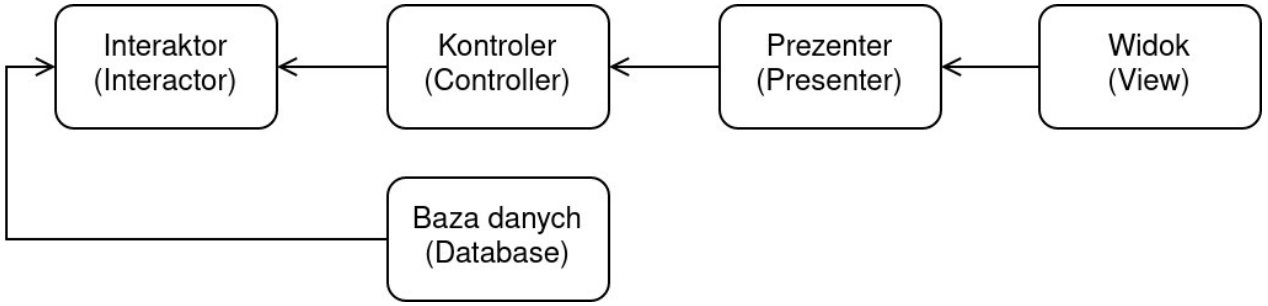
OCP



Uogólniony schemat wygląda następująco:

wysoko w hierarchii
mało zmian

nisko w hierarchii
dużo zmian



LSP

Liskov Substitution Principle

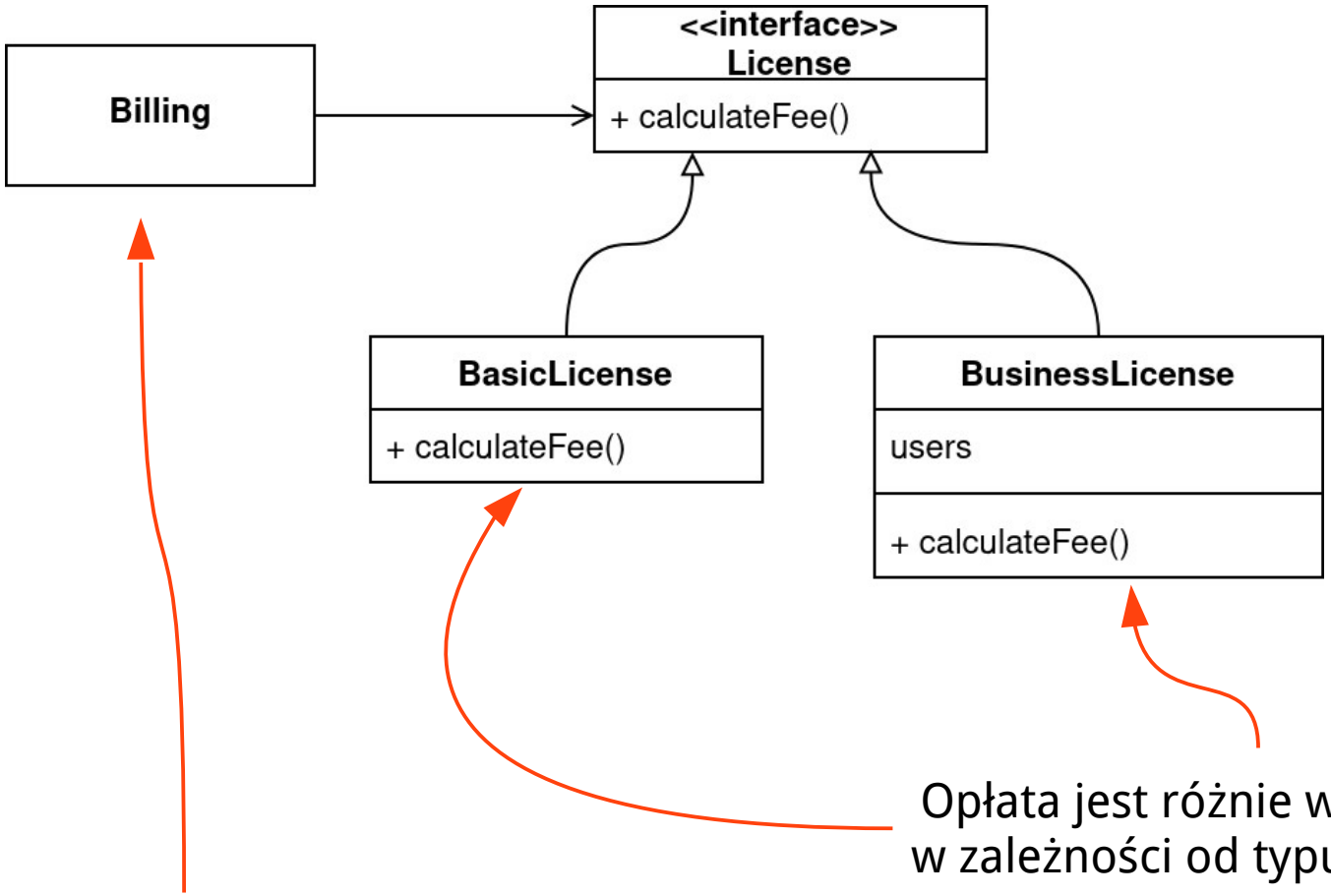
Reguła podstawień Liskov

Chciałam uzyskać coś w rodzaju następującej właściwości podstawień: jeżeli dla każdego obiektu o1 typu S istnieje obiekt o2 typu T, taki, że dla każdego programu P określonego warunkami T zachowanie P nie zmieni się, gdy zamiast o1 zostanie podstawiony o2, to znaczy, że S jest podtypem T.

Chyba łatwiej będzie to zrozumieć na przykładzie...

LSP

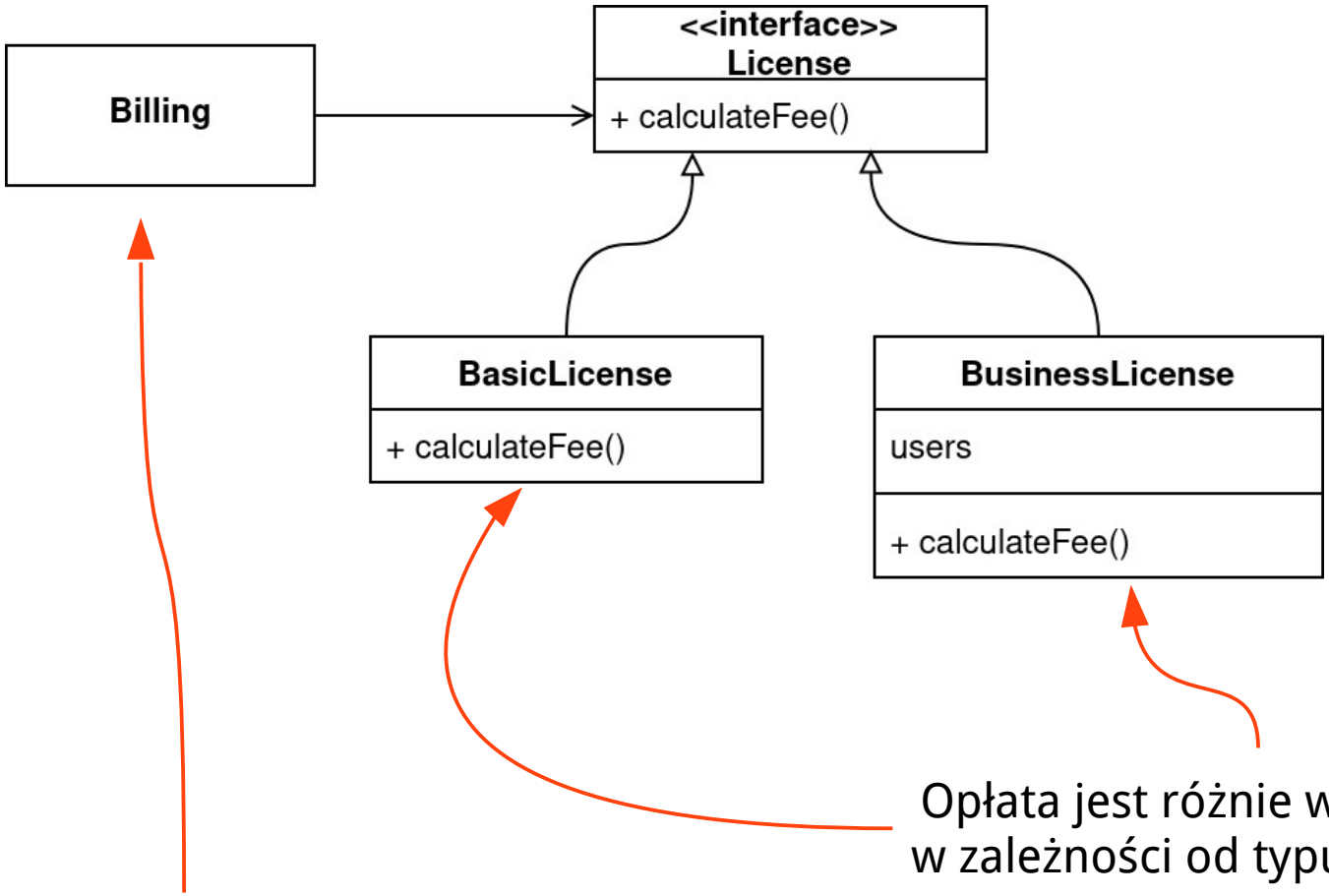
Aplikacja odpowiedzialna za obliczanie opłaty licencyjnej może wyglądać tak:



Zachowanie aplikacji Billing nie zależy od wybranej podklasy licencji – zawsze jest liczona opłata (odpowiednio dla danego typu licencji).

LSP

Aplikacja odpowiedzialna za obliczanie opłaty licencyjnej może wyglądać tak:

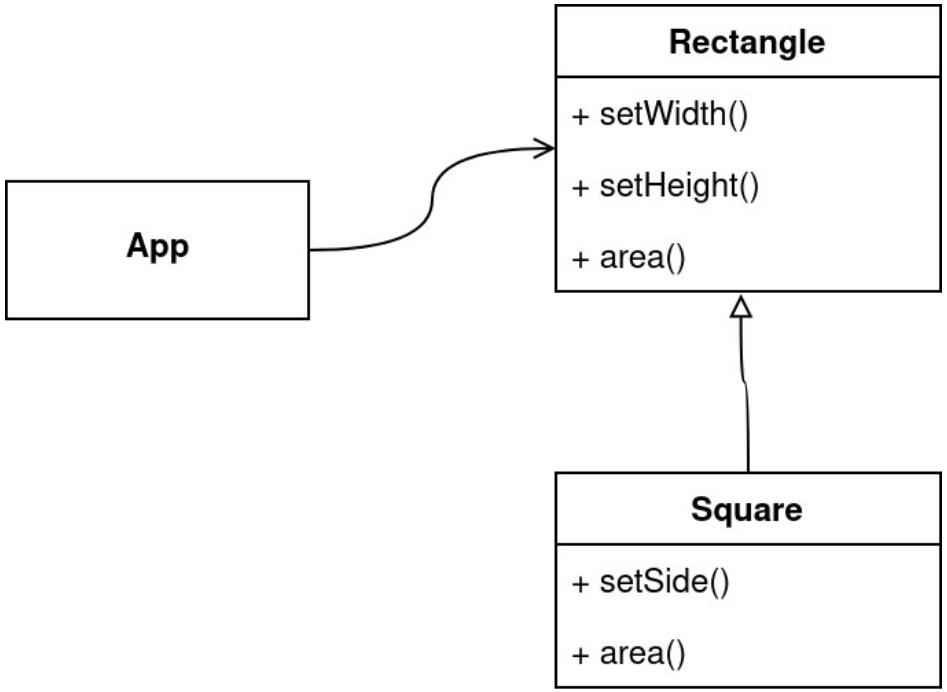


ZGODNE
Z LSP

Zachowanie aplikacji Billing nie zależy od wybranej podklasy licencji – zawsze jest liczona opłata (odpowiednio dla danego typu licencji).

LSP

A tu kolejny przykład – aplikacja wyliczająca pole prostokątów:



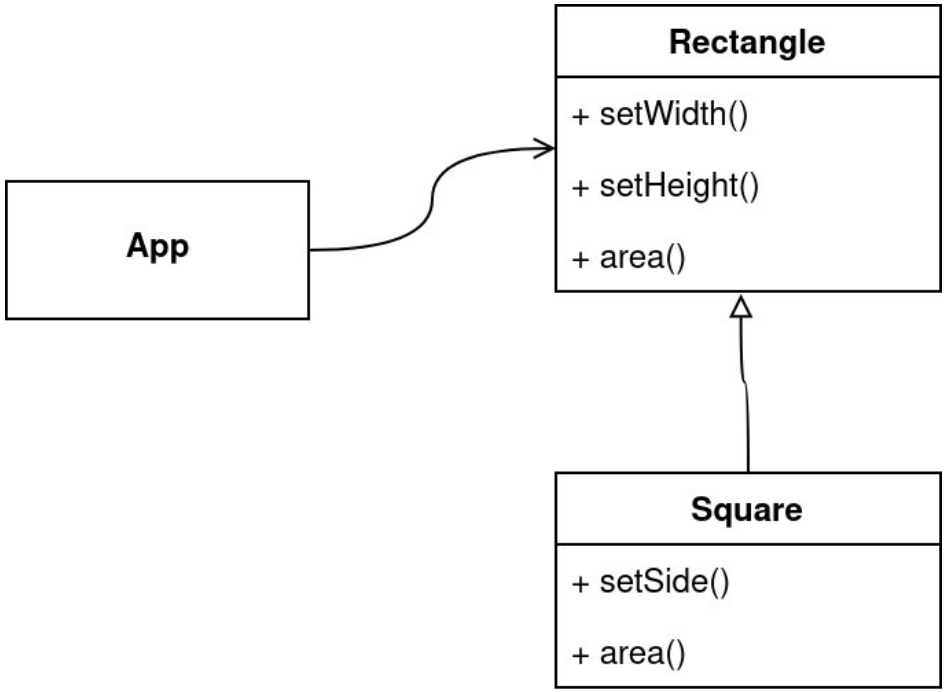

```
Rectangle rect = …
rect.setWidth(10);
rect.setHeight(3);
a = rect.area()
```

Tu może wylądować obiekt klasy Rectangle albo Square.

Czy wynik będzie prawidłowy dla kwadratu?

LSP

A tu kolejny przykład – aplikacja wyliczająca pole prostokątów:




```
Rectangle rect = …
rect.setWidth(10);
rect.setHeight(3);
a = rect.area()
```

Tu może wylądować obiekt klasy Rectangle albo Square.

Czy wynik będzie prawidłowy dla kwadratu?

NIEZGODNE Z LSP

Cały problem polega na tym, że zakładamy (ze strony App), że mamy do czynienia z prostokątem.

Aby naprawić* kod, musielibyśmy dodać instrukcje warunkowe (nasze kochane if-y) sprawdzające, z czym tak naprawdę mamy do czynienia.

* w tym przypadku raczej zepsuć...

ISP

Interface Segregation Principle

Reguła rozdzielania interfejsów

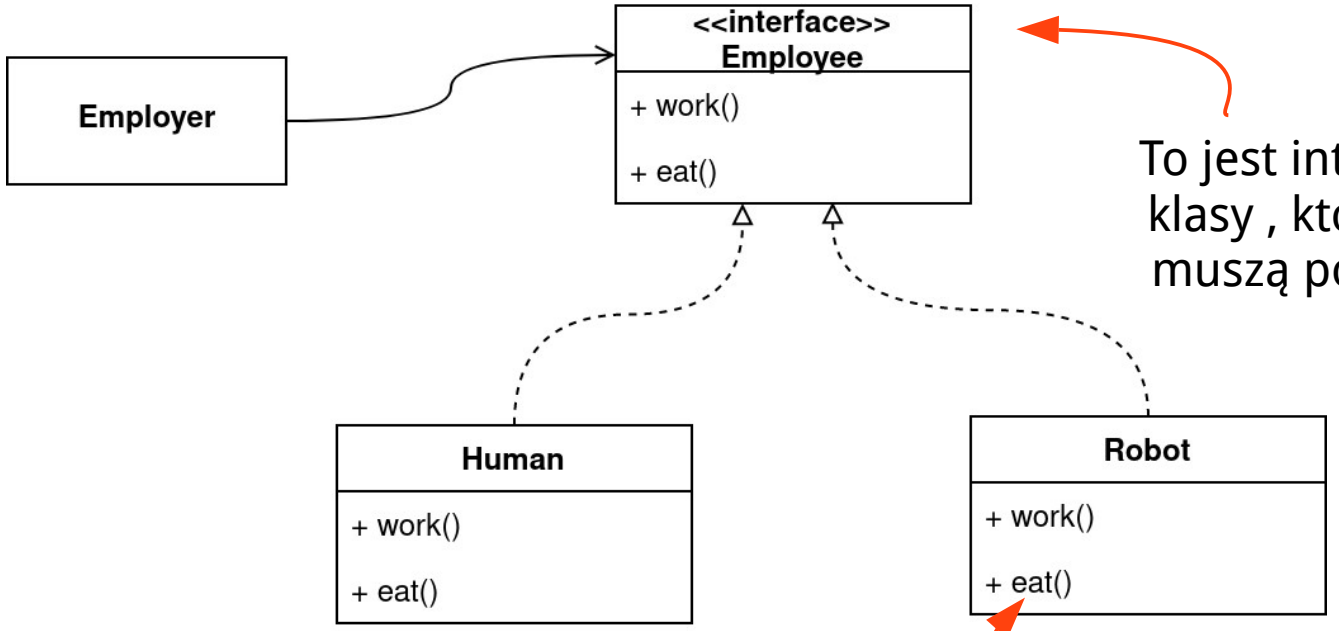
W skrócie:

Trzeba unikać sytuacji, w których wprowadzamy zależność od czegoś, czego dany kod (np. klasy) nie używa.

I znów najlepiej omówić to na przykładzie.

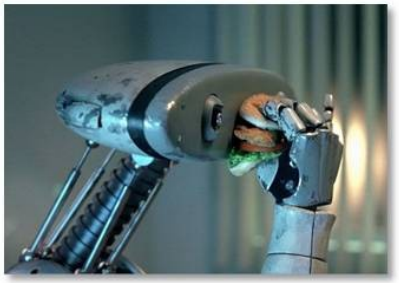
ISP

Tworzymy system obsługujący pracowników w pewnej fabryce (w już nie tak odległej przyszłości):



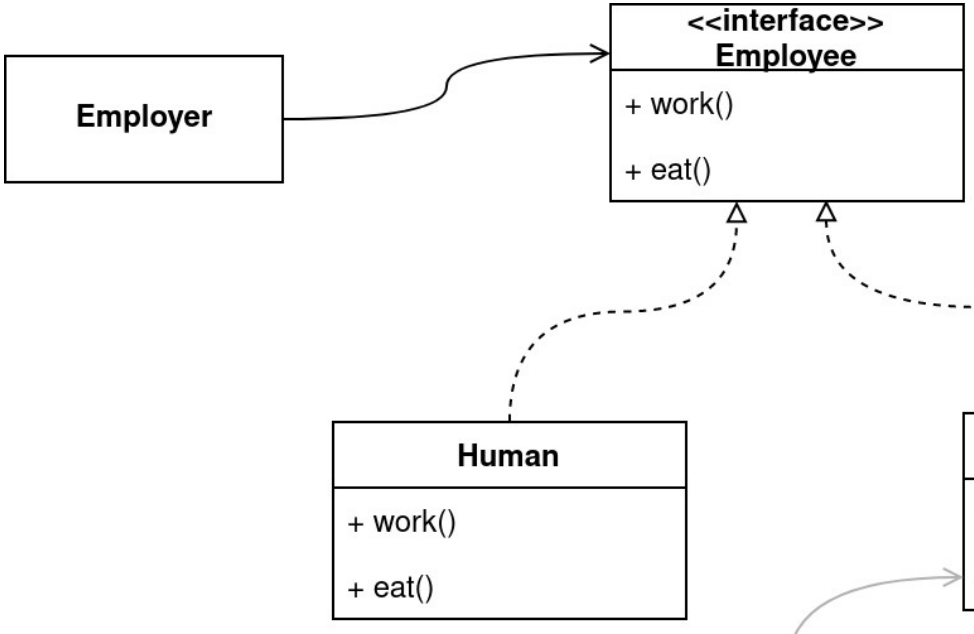
To jest interfejs, więc wszystkie klasy , które go implementują, muszą posiadać obie metody.

Co jadają roboty?



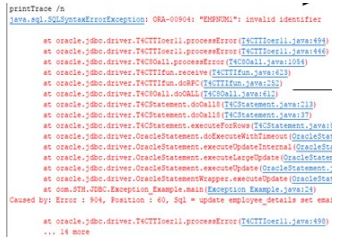
ISP

Tworzymy system obsługujący pracowników w pewnej fabryce (w już nie tak odległej przyszłości):



To jest interfejs, więc wszystkie klasy , które go implementują, muszą posiadać obie metody.

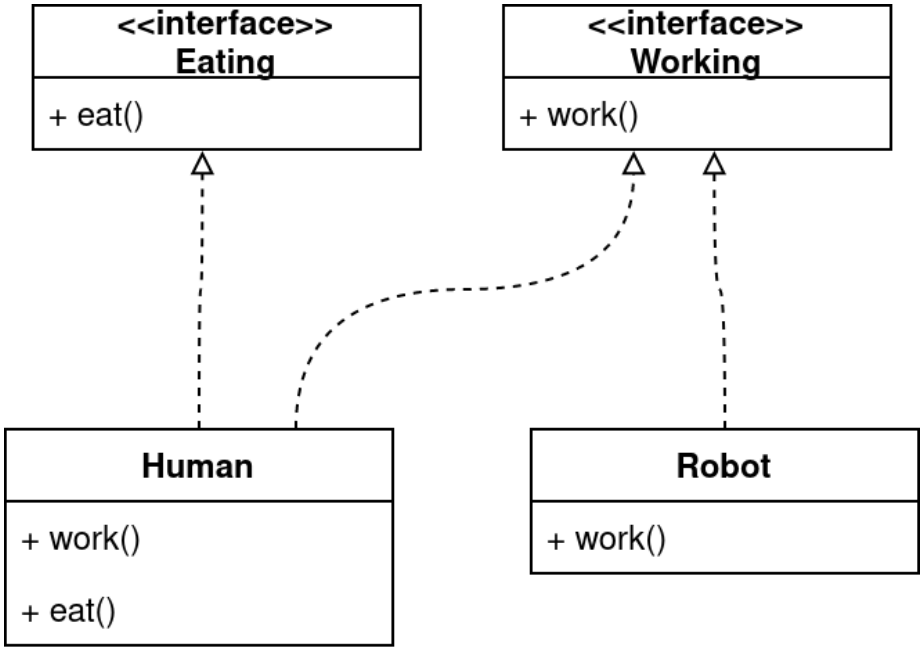
```
public void eat() {
   throw new UnsupportedOperationException();
}
```



ISP

Zawsze powinniśmy unikać tworzenia „dużych” interfejsów.

Lepiej zdefiniować wiele małych „wyspecjalizowanych” interfejsów.



W rzeczywistych sytuacjach należałoby jeszcze zapewnić wygodny sposób korzystania z obiektów konkretnych klas, ale to przy innej okazji...
Poczytajcie na przykład o wzorcu projektowym „Adapter”.

DIP

Dependency Inversion Principle

Reguła odwrócenia zależności

Główna myśl:

Elastyczne systemy to takie, w których zależności w kodzie źródłowym odnoszą się wyłącznie do abstrakcji, a nie do konkretnych elementów.

Bardziej konkretnie (a właściwie abstrakcyjnie...):

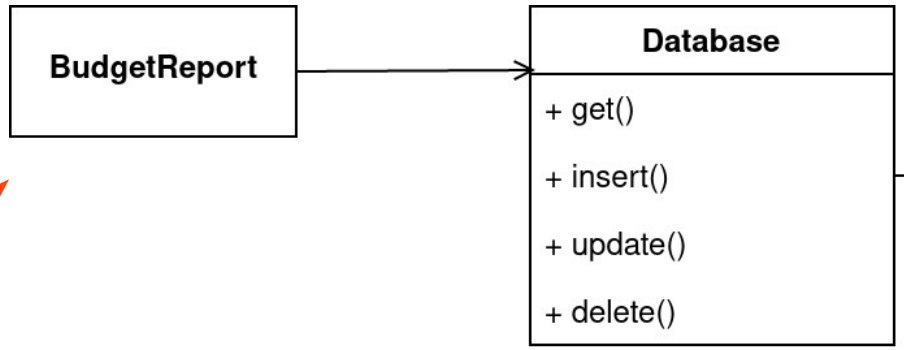
– Wysokopoziomowe moduły nie mogą zależeć od niskopoziomowych modułów. Moduły obu typów powinny zależeć od abstrakcji.

– Abstrakcje nie mogą zależeć od konkretów. Konkrety powinny zależeć od abstrakcji.

No dobra, przykład...

DIP

Kolejny nudny, korporacyjny system...



To jest **konkretna** baza danych, np. MySQL.

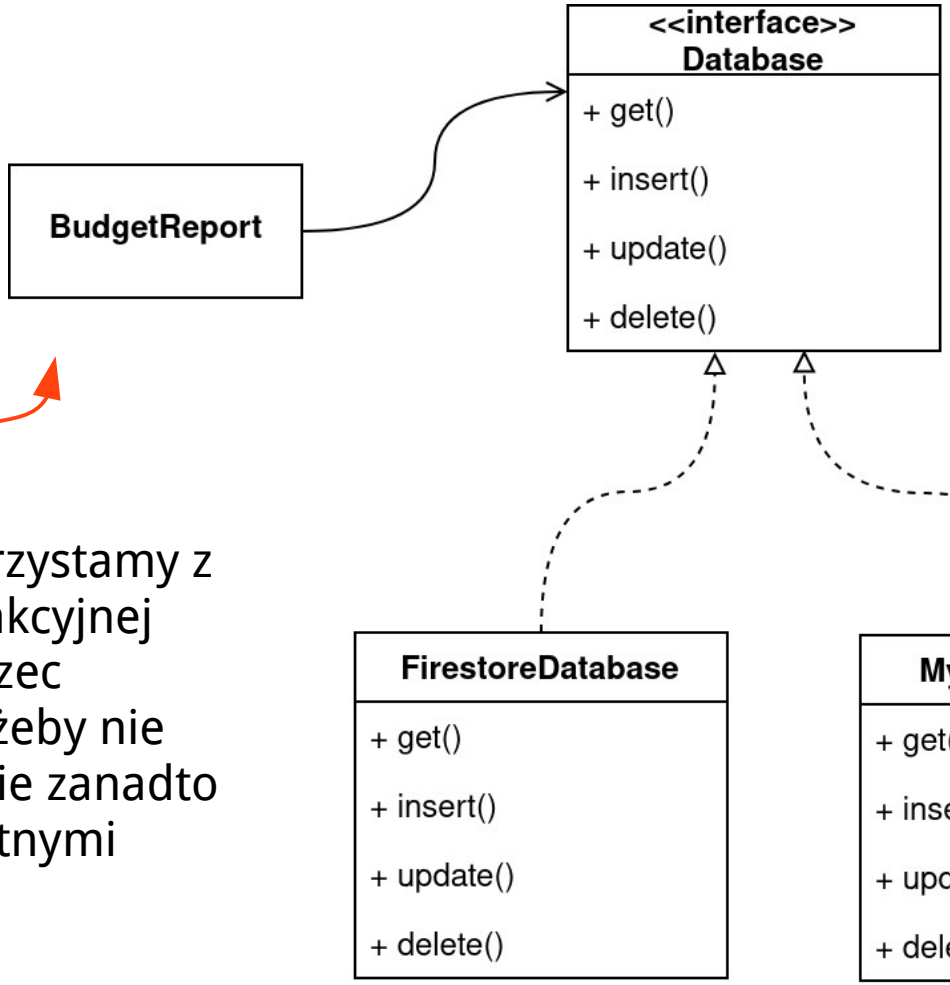
W związku z tym w metodach tej klasy znajdują się **konkretne** implementacje powiązane z bazą MySQL.

Z perspektywy obiektu generującego raporty nie ma znaczenia, z jakiej konkretnie bazy danych korzystamy.

A co, jeśli trzeba będzie zmienić typ bazy danych?

DIP

Musimy wprowadzić **abstrakcję**:



Czysta **abstrakcja** (interfejs).

Mówimy tylko, co można zrobić, ale nie jak to zrobić.

Zazwyczaj korzystamy z fabryki abstrakcyjnej (kolejny wzorzec projektowy), żeby nie zawracać sobie zanadto głowy konkretnymi klasami.

Tutaj mówimy, jak to zrobić (w każdym **konkretnym** przypadku bazy danych).

Podsumowanie

Omówiliśmy pięć reguł:

SRP

Każdy moduł oprogramowania powinien mieć jeden i tylko jeden powód do zmiany.

OCP

Zmiana zachowania systemu ma być możliwa przez dodanie nowego kodu, a nie modyfikowanie istniejącego.

DIP

Kod wyższego poziomu nie może zależeć od kodu niższego poziomu. Ma być na odwrót.

LSP

Jeżeli chcemy umożliwić wymianę części na inne, musimy sprawić, żeby stosowały się do tego samego kontraktu.

ISP

Musimy unikać tworzenia zależności od elementów, których nie potrzebujemy.

Jeżeli ustawimy je w odpowiedniej kolejności, ich pierwsze litery utworzą...

Podsumowanie

SRP **O**CP **L**SP **I**SP **D**IP