


SOLID / GRASP

Projektowanie oprogramowania

dr inż. Gabriel Rojek

1

Parę słów o historii (1)

- **1950-1960** – języki strukturalne (if/then/else, for, ...)
 - **1960-1980** – object oriented programming (C++ 1979-1982, Java 1991)
 - **1990-1995** – liczne metody modelowania (projektowania) obiektowego
 - **1994** - „Design Patterns: Elements of Reusable Object-Oriented Software” **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides**
 - **1995-2005** – UML (0.8-2.2)
- 

2

Parę słów o historii (2)

- **1998** – „AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis” **William Brown, Raphael Malveau, Skip McCormick, Tom Mowbray** (powstanie/popularyzacja terminu antywzorca)
- **2000** – „Design Principles and Design Pattern” **Robert C. Martin** (paradygmaty SOLID)
- **2004 (?)** – GRASP
- **????** – inne paradygmaty (YAGNI, KISS, DRY)

3

Parę słów o historii (3)

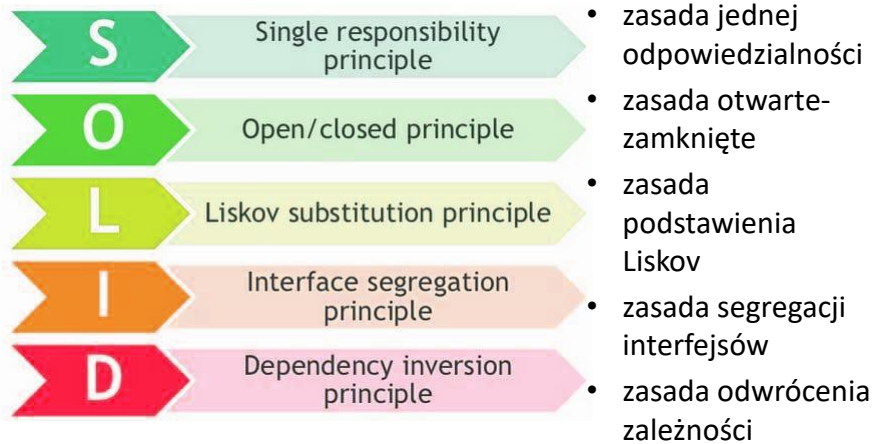
- Ale jeszcze wzorce architektoniczne...
 - Client-server, 1960-1970
 - MVC, powstały w końcówce lat 1980, ale spopularyzowany w latach 2000
 - MVP, powstały na początku lat 1990, ale spopularyzowany po 2005 roku
 - MVVM, 2005
 - SOA, 2009

4



5

SOLID – mnemonik zaproponowany przez Roberta C. Martina, opisujący **pięć podstawowych założeń programowania obiektowego**. Przestrzeganie tych zasad znacznie poprawia **czytelność kodu oraz możliwości konserwacji i utrzymania tworzonego oprogramowania**.



6



7

Single Responsibility Principle

- **„Nigdy nie powinno być więcej niż jednego powodu do modyfikacji klasy.”**
- W teorii **chodzi o nieprzypisywanie klasom więcej niż jednego konkretnego zadania (odpowiedzialności)**, patrząc z perspektywy logiki aplikacji.
- W rozumieniu praktycznym, nie jest to kwestia ograniczania ilości metod lub pól w danej klasie, ale zachowania ich spójności względem danej odpowiedzialności.
 - Klasy powinny zawierać w sobie funkcjonalności powiązane z ich nazwą i przeznaczeniem, przy założeniu że owe przeznaczenie jest jasne i klarowne, a przede wszystkim – jedno.

8

Single Responsibility Principle

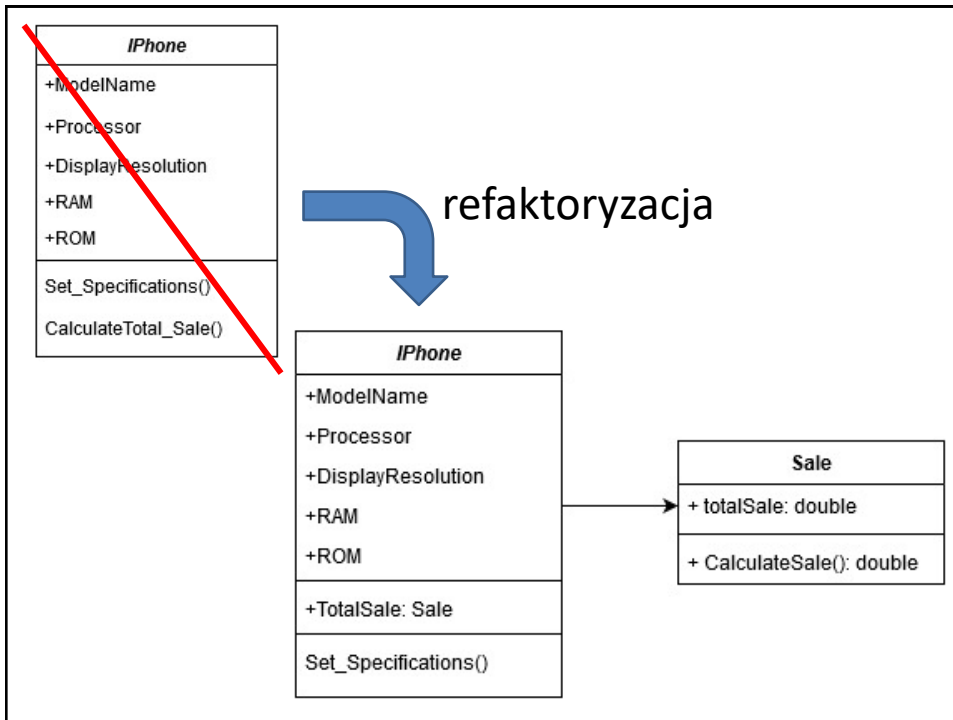
- Jeśli jakaś klasa jest odpowiedzialna za więcej niż jeden obszar naszego projektu (odpowiedzialność), to jest to bardzo niekorzystne.
 - Może się zdarzyć, że modyfikując jeden obszar mimowolnie wpłyniemy na zupełnie inny, który nie jest w żaden sposób związany z pierwszym.
 - Każdy taki związek wpływa negatywnie na cały projekt poprzez zmniejszenie jego elastyczności, a co za tym idzie – prowadzi do nieoczekiwanych rezultatów wprowadzanych przez nas zmian

9

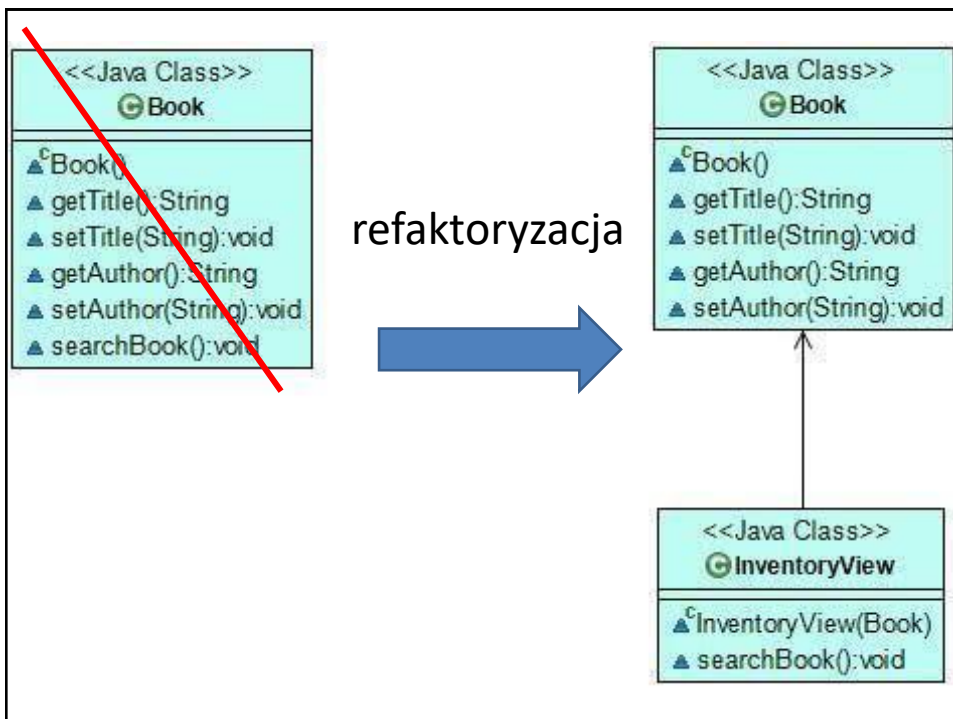
Single Responsibility Principle

- Adekwatny podział odpowiedzialności ma wiele zalet, zwłaszcza z perspektywy przyszłej konserwacji oprogramowania.
- Klasy łamiące tą zasadę, pomimo iż w praktyce są szybsze w implementacji i mają szersze spectrum zastosowania, z perspektywy czasu i ewentualnych zmian, są trudniejsze w utrzymaniu i modyfikacji, niż klasy o jednej odpowiedzialności i wysokiej spójności.

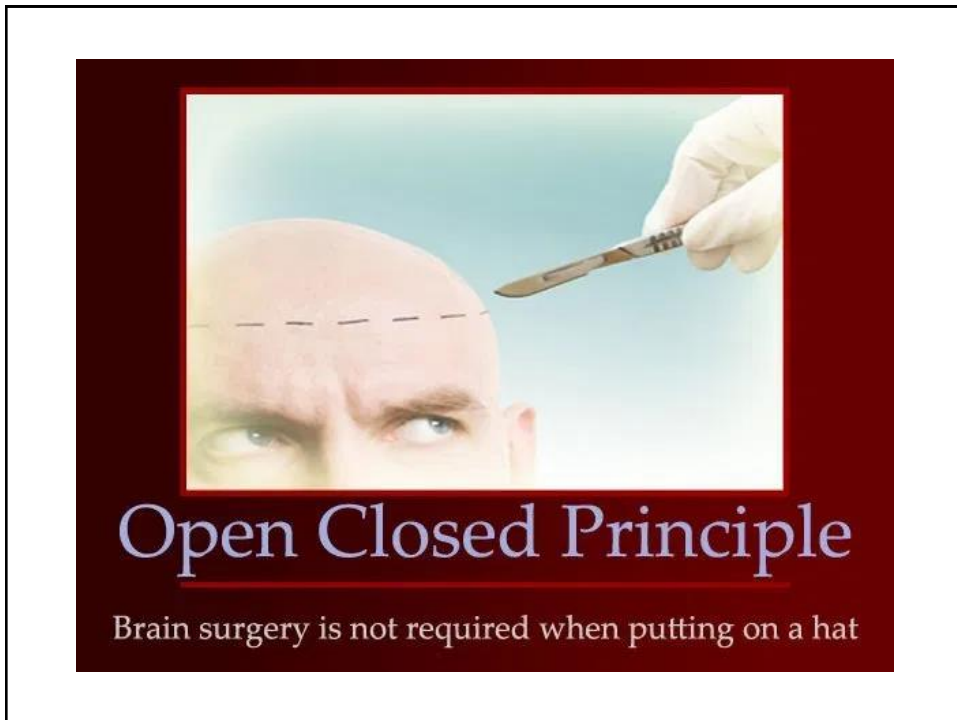
10



11



12



13

Open/Closed Principle

- ***„Elementy systemu (klasy moduły i funkcje, etc.) powinny być otwarte na rozszerzenie, ale zamknięte na modyfikacje.”***
- W przypadku poszerzenia logiki aplikacji o nowe funkcjonalności, stary kod nie będzie wymagał modyfikacji (ale będzie rozszerzany).
 - Oznacza to, iż można zmienić zachowanie takiego elementu bez zmiany jego kodu. Jest to szczególnie ważne w środowisku produkcyjnym, gdzie zmiany kodu źródłowego mogą być niewskazane i powodować ryzyko wprowadzenia błędu. Program, który trzyma się tej zasady, nie wymaga zmian w kodzie, więc nie jest narażony na powyższe ryzyko.

14

Open/Closed Principle

- „Składniki oprogramowania (klasy, moduły, funkcje itp.) powinny być otwarte na rozbudowę, ale zamknięte dla modyfikacji.”
- „otwarte na rozbudowę” - musi istnieć stosunkowo prosty sposób rozbudowy zachowań takiego modułu.
- „zamknięte dla modyfikacji” - rozbudowa modułu nie może być przeprowadzona w sposób, który spowoduje zmianę istniejącego kodu źródłowego.

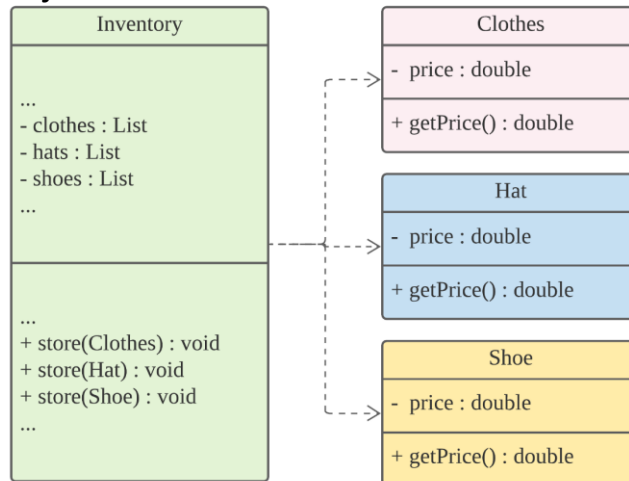
15

Open/Closed Principle

- Każdą klasę powinniśmy pisać tak, aby możliwe było dodawanie nowych funkcjonalności, bez konieczności jej modyfikacji.
 - Modyfikacja jest surowo zabroniona, ponieważ zmiana deklaracji jakiegokolwiek metody może spowodować awarię systemu w innym miejscu.

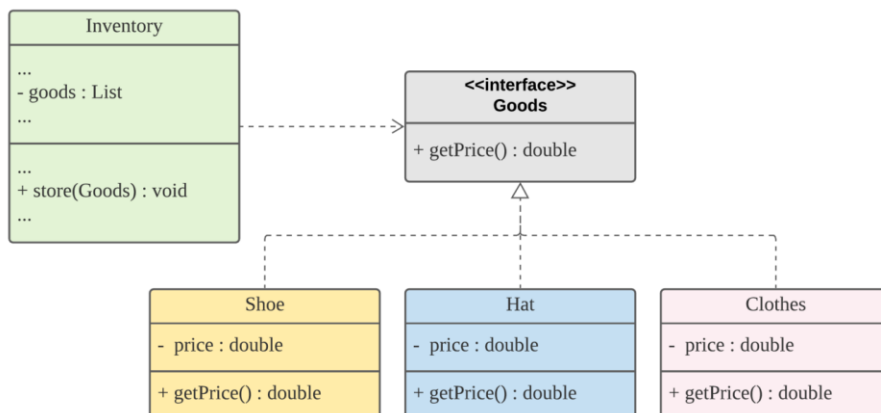
16

- Czy poniższe klasy spełniają open/closed principle?
- Dlaczego nie? Na czym będzie polegała refaktoryzacja?

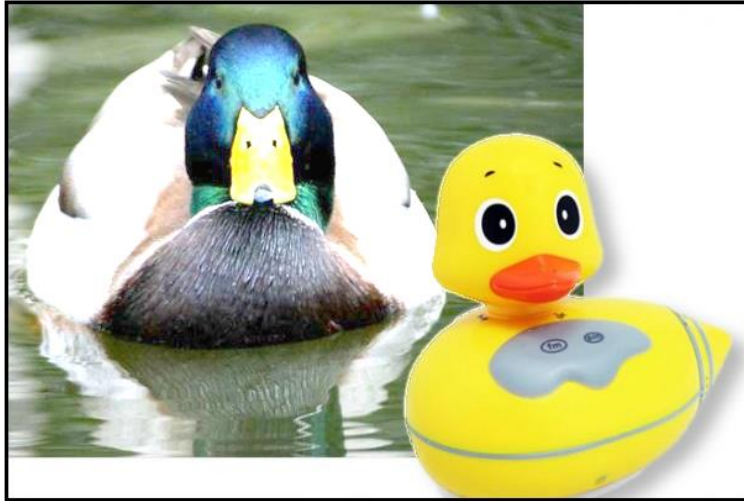


17

- A te klasy (i interfejs) spełniają open/closed principle?
- Dlaczego tak?



18



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

19

Liskov Substitution Principle

- *„Funkcje które używają wskaźników lub referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych, bez dokładnej znajomości tych obiektów.”*
- *„Muszą istnieć możliwości zastępowania typów bazowych ich podtypami”.*



BARBARA LISKOV

Developed the Liskov substitution principle

20

Liskov Substitution Principle

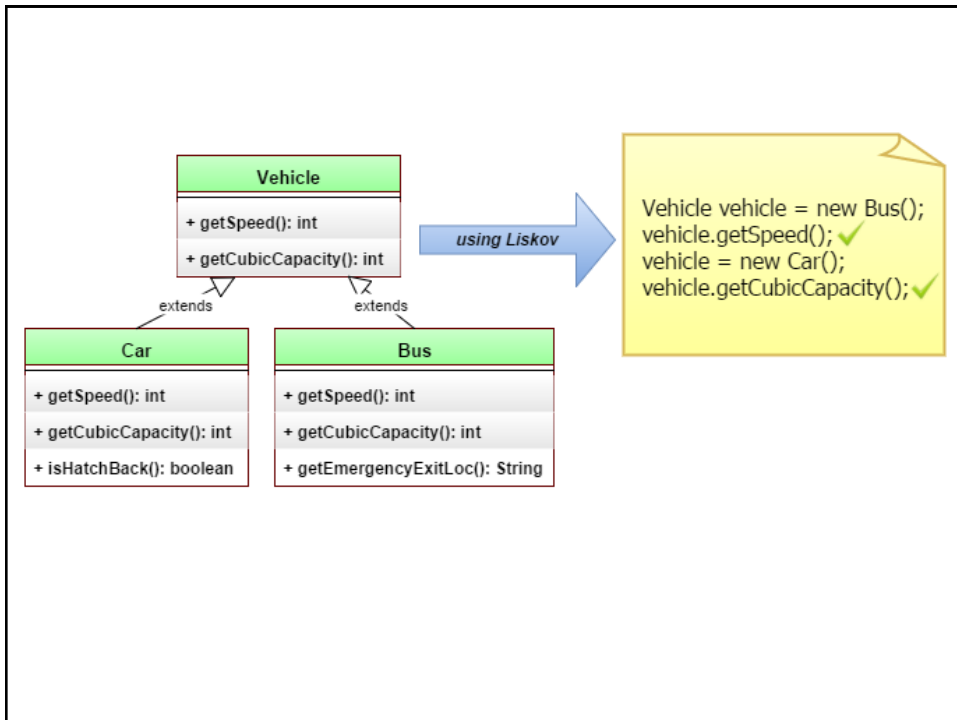
- Jeśli używasz typu bazowego, powinieneś mieć możliwość postawienia w jego miejsce typów pochodnych.
- Funkcje operujące na klasie bazowej, powinny działać odpowiednio także w przypadku przekazania instancji klasy pochodnej.
 - Twój kod powinien współpracować poprawnie z klasą, jak i wszystkimi jej podklasami.
 - Innymi słowy, jeśli zależyś od jakiegoś interfejsu to wszystkie jego implementacje powinny poprawnie działać z Twoją klasą/metodą.

21

Liskov Substitution Principle

- Zasada podstawiania Liskov jest jednym z warunków zasady otwarte – zamknięte:
 - możliwość zastępowania podtypów umożliwia rozbudowę modułów (klas, typów bazowych) bez konieczności ich bezpośredniego modyfikowania.

22



23



24

Interface Segregation Principle

- „Wiele dedykowanych interfejsów jest lepsze niż jeden ogólny.”
- Interfejsy powinny być konkretne i jak najmniejsze,
 - nie tworzyć interfejsów z metodami, których nie używa klasa.
- Klasa nie powinna być zależna od metod, których nie używa.

25

Interface Segregation Principle

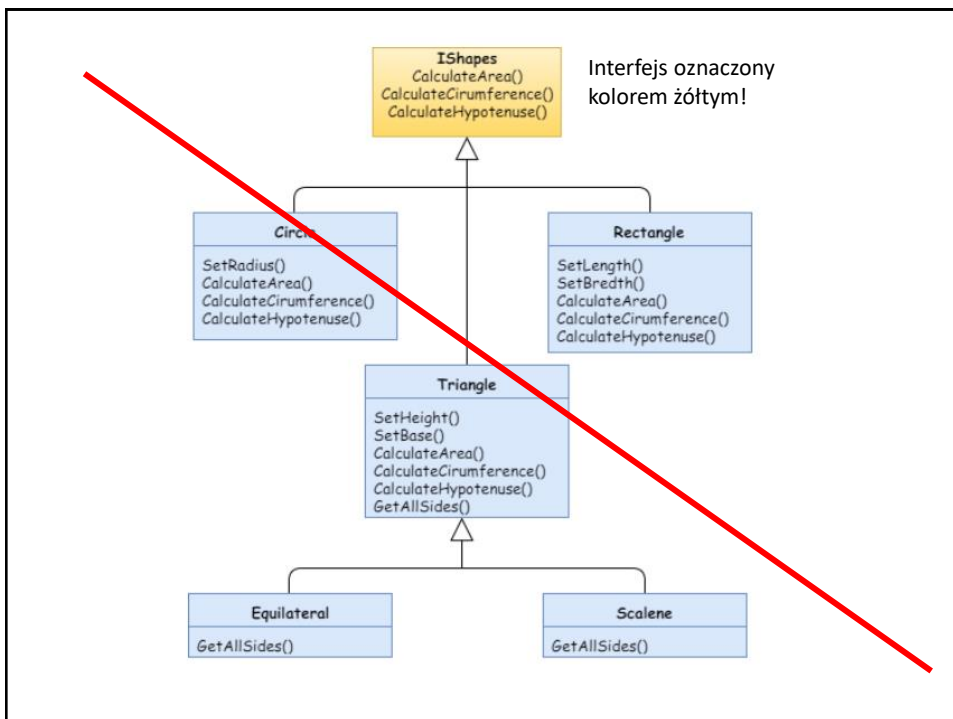
- Zmuszanie obiektów do implementacji metod, których nie będą wykorzystywały nie jest dobrym rozwiązaniem, ponieważ w ten sposób będziemy dodawać do naszej aplikacji zbędny kod.
- Znacznie korzystniej będzie rozbić powyższy protokół (interfejs) na kilka mniejszych i przypisywać je tylko w razie faktycznej potrzeby.

26

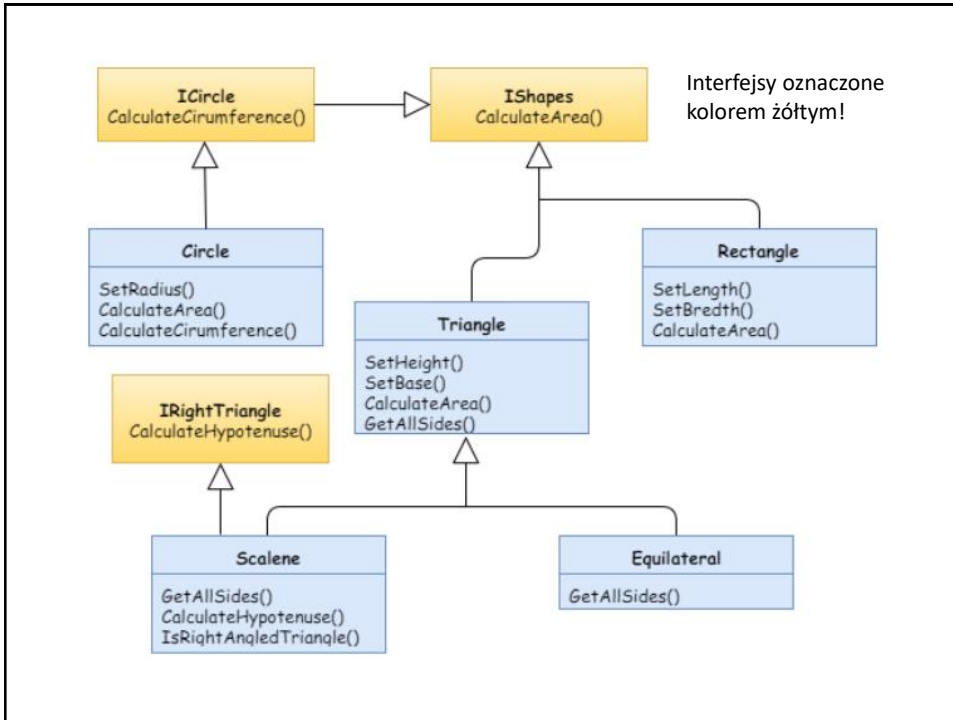
Interface Segregation Principle

- Załóżmy, że posiadamy pewien interfejs oraz dwie klasy, które go implementują.
 - Klasa A implementuje nasz przykładowy interfejs w całości
 - wszystkie funkcje,
 - Klasa B posiada pełną implementację tylko części z nich, pozostałe, które są niepotrzebne pozostają puste lub zwracają domyślnie wartości.
- Jeśli mamy do czynienia z taką sytuacją, powinniśmy rozważyć **rozdzielenie interfejsu na mniejsze tak, aby każdy z nich deklarował tylko te funkcje, które rzeczywiście są wywoływane przez danego klienta** lub grupę klientów (klasa/grupy klas implementujące dany interfejs).

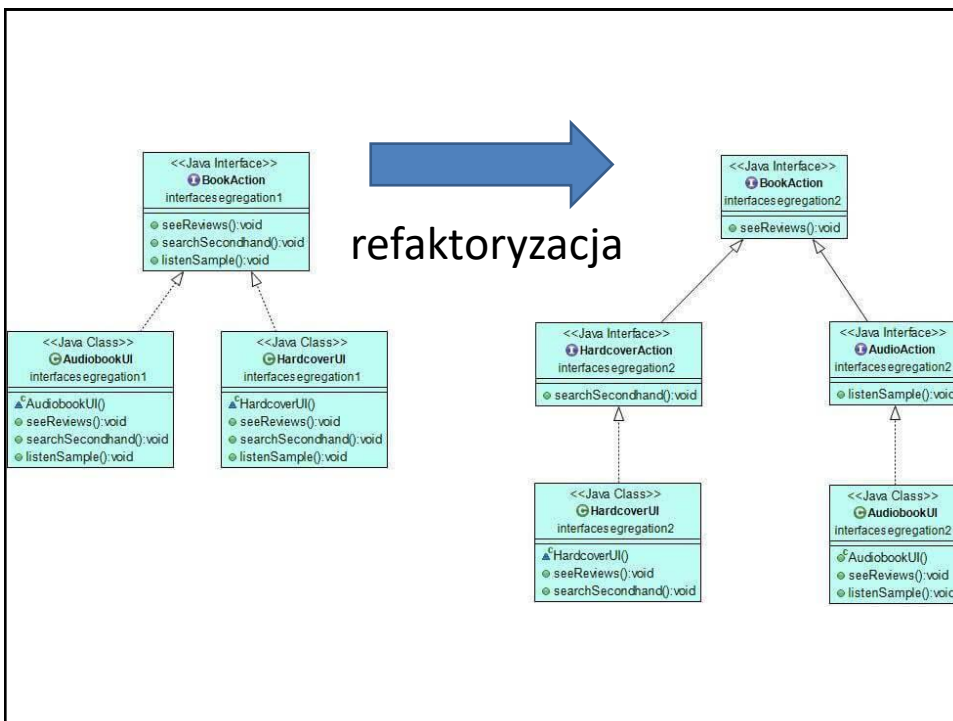
27



28



29



30

Dependency Inversion Principle



31

Dependency Inversion Principle

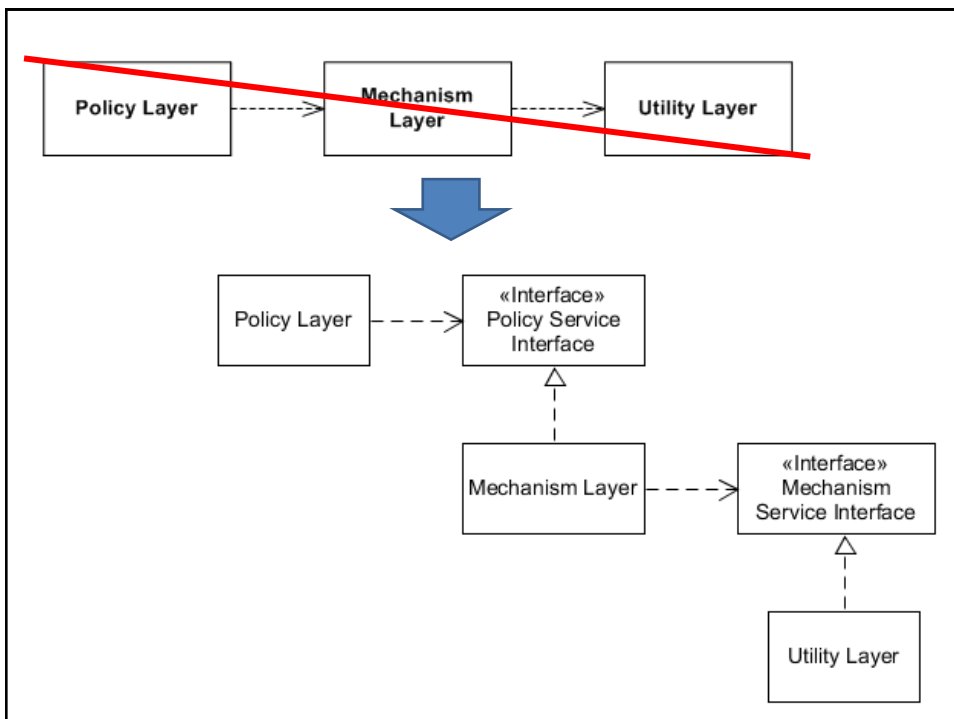
- „Wysokopoziomowe moduły nie powinny zależeć od modułów niskopoziomowych - zależności między nimi powinny wynikać z abstrakcji.”
- ”Abstrakcja nie powinna zależeć od implementacji – implementacja powinny zależeć od abstrakcji.”
- Cel - zmniejszenie zależności od konkretnych implementacji. Możemy to uzyskać za pomocą abstrakcji (interfejsów). Jeśli kod zależy od interfejsu to mamy małą zależność. Dzięki temu nasz kod nie zmienia się lub zmienia się bardzo rzadko.

32

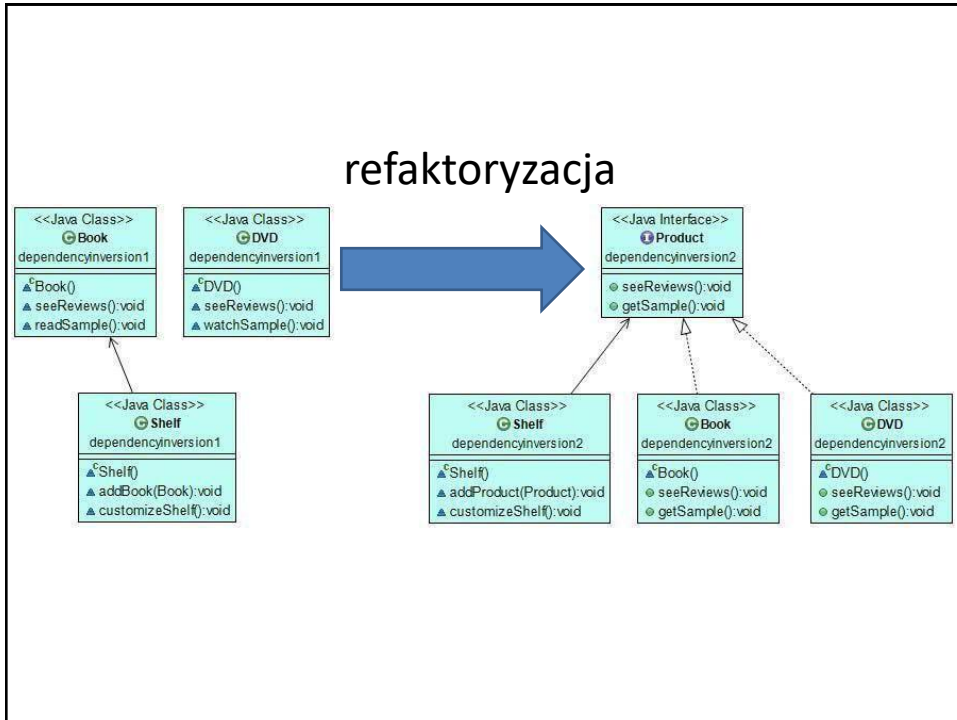
Dependency Inversion Principle

- Wszystkie zależności powinny w jak największym stopniu zależeć od abstrakcji a nie od konkretnego typu.
- Polega to na używaniu interfejsu polimorficznego wszędzie tam gdzie jest to możliwe, szczególnie w parametrach metod.
 - Jeżeli mamy parametr funkcji, który przyjmuje figurę matematyczną, znaczenie lepszym rozwiązaniem będzie przyjęcie interfejsu figur matematycznych niż konkretnej figury.
 - Dzięki temu, nie uzależniamy pojedynczej metody od konkretnego typu, tylko od interfejsu, który mogą implementować duże grupy podtypów.

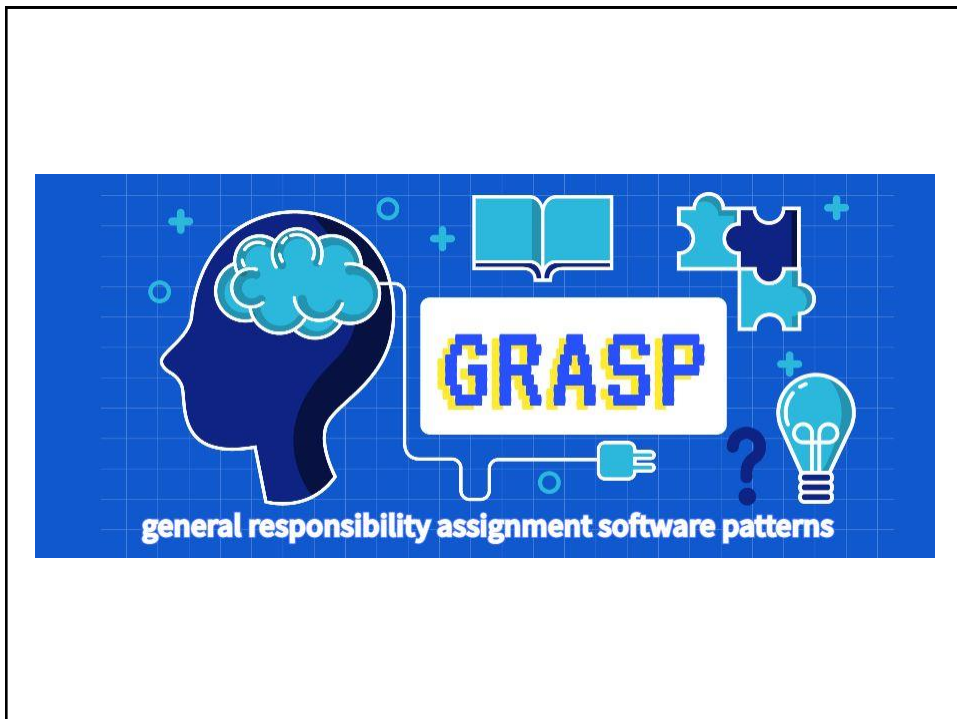
33



34



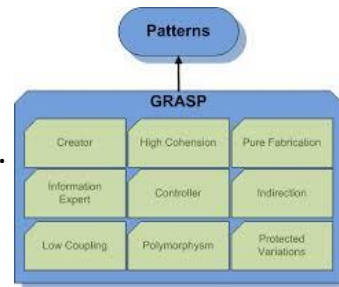
35



36

GRASP - General Responsibility Assignment Software Patterns (or Principles)

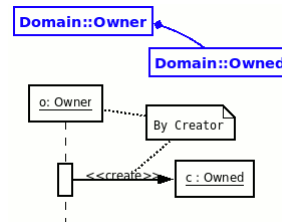
- GRASP jest zbiorem zasad opisujących dobre **praktyki stosowane przy podziale odpowiedzialności** pomiędzy klasami.
 - Stosowanie się do tych reguł na etapie projektowania systemu informatycznego, ułatwia późniejszą konserwację i utrzymanie oprogramowania.



37

Creator

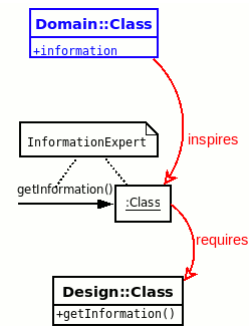
- Problem:** Kiedy obiekt powinien tworzyć inny obiekt?
- Rozwiązanie:** Klasa B powinna otrzymać odpowiedzialność utworzenia obiektu klasy A, jeżeli:
 - klasa B zawiera lub agreguje obiekty klasy A,
 - B „zapisuje/rejestruje życie” instancji klasy A,
 - B blisko współpracuje z A,
 - B ma dane inicjalizacyjne potrzebne przy tworzeniu A.
- Opis:** Przestrzeganie tej reguły zmniejsza prawdopodobieństwo powstawania niepotrzebnych powiązań.



38

Information Expert

- **Problem:** Której klasie przypisać daną odpowiedzialność lub zadanie?
- **Rozwiązanie:** Przypisz to zadanie klasie, która ma informacje niezbędne do tego aby móc je wykonać.
- **Opis:** Należy zacząć od określenia danych potrzebnych do wykonania nowego zadania. Wybór dokonany w oparciu o tę zasadę, ogranicza konieczność tworzenia dodatkowych interfejsów.



39

Controller

- **Problem:** Który obiekt poza GUI (interfejsem użytkownika) powinien obsłużyć żądania systemu?
- **Rozwiązanie:** Przydziel odpowiedzialność do obiektu spełniającego jeden z warunków:
 - klasa reprezentuje cały system,
 - klasa reprezentuje przypadek użycia systemu, w którym wykonywana jest dana operacja.
- **Opis:** Zasada ta mówi o potrzebie istnienia kontrolera, pośredniczącego pomiędzy GUI a modelem aplikacji. Przykładem wykorzystania tej idei jest wzorzec **MVC**.

40

Low Coupling

- **Problem:** Jak ograniczyć zakres zmian w systemie w momencie zmiany fragmentu systemu?
- **Rozwiązanie:** Klasy A i B powinny być ze sobą powiązane, gdy:
 - obiekt A ma atrybut typu B lub typu C związanego z B,
 - obiekt A wywołuje metody obiektu typu B,
 - obiekt A ma metodę związaną z typem B (zmienna lokalna, typ wartości/parametru),
 - obiekt A dziedziczy po B.
- **Opis:** Przestrzeganie tej reguły ogranicza ilość niepotrzebnych powiązań pomiędzy klasami. Dzięki temu poprawia modułowość oraz przenośność aplikacji. Klasy łamiące tę zasadę, prawdopodobnie łamią również zasadę *High Cohesion*.

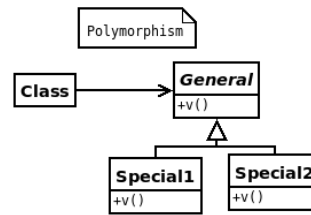
41

High Cohesion

- **Problem:** Jak sprawić by obiekty miały jasny cel, były zrozumiałe i łatwe w utrzymaniu?
- **Rozwiązanie:** Przypisuj odpowiedzialności do obiektu tak, aby spójność była możliwie największa.
- **Opis:** Nie należy tworzyć obiektów o zbyt szerokiej odpowiedzialności. Funkcjonalności danej klasy powinny być ze sobą jak najbardziej spójne, co zmniejsza prawdopodobieństwo złamania zasady *Low Coupling*.

42

Polymorphism



- **Problem:** Co zrobić, gdy odpowiedzialność różni się w zależności od typu?
- **Rozwiązanie:** Przydziel zobowiązania przy użyciu polimorfizmu, typom dla których to zachowanie jest różne.
- **Opis:** Zastosowanie polimorfizmu zwiększa czytelność i ułatwia utrzymanie kodu.

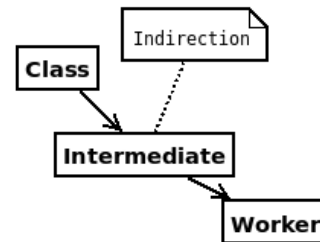
43

Pure Fabrication

- **Problem:** Jak przydzielić odpowiedzialność, by nie naruszyć zasad *High Cohesion* i *Low Coupling*, a nie odpowiada nam rozwiązanie sugerowane przez *Information Expert*?
- **Rozwiązanie:** Przypisz zakres odpowiedzialności sztucznej lub pomocniczej klasie, które nie reprezentuje żadnego problemu domenowego. Nie narusz zasad *High Cohesion* i *Low Coupling*.
- **Opis:** W pewnych przypadkach powyższe zasady mogą się wykluczać. Z tego powodu lepiej stworzyć dodatkową klasę pomocniczą pośredniczącą pomiędzy analizowanymi funkcjonalnościami, niż złamać jedną z powyższych reguł.

44

Indirection



- **Problem:** Komu przydzielić zobowiązanie, jeśli zależy nam na uniknięciu bezpośredniego powiązania pomiędzy klasami?
- **Rozwiązanie:** Przypisz te odpowiedzialności do nowego pośredniego obiektu. Obiekt ten będzie służył do komunikacji innych klas/komponentów/usług/pakietów tak, że nie będą one zależne bezpośrednio od siebie.
- **Opis:** Stworzenie obiektu pośredniczącego w komunikacji, w wielu przypadkach jest opłacalnym rozwiązaniem. Przykładem wykorzystania tej zasady jest wzorec projektowy *Mediator*.

45

Protected Variations

- **Problem:** Jak projektować obiekty, by ich zmiana nie wywierała szkodliwego wpływu na inne obiekty?
- **Rozwiązanie:** Zidentyfikuj punkty przewidywanego różnicowania czy niestabilności i przypisz odpowiedzialności do wspólnego stabilnego interfejsu.
- **Opis:** Wyodrębnienie punktów niestabilności znacząco ułatwia ich późniejszą modyfikację.

46

Inne zasady

47

- **YAGNI:** „*You Aren't Gonna Need It*”
- **Opis:** Zasada ta zwraca uwagę, by nie tworzyć niepotrzebnych funkcjonalności.
- **KISS:** „*Keep It Simple, Stupid*”
- **Opis:** Nie należy niepotrzebnie komplikować implementacji.
- **DRY:** „*Don't Repeat Yourself*”
- **Opis:** Błędem jest załączanie tych samych funkcjonalności w różnych klasach.

48