

---

# Obiektowe projektowanie systemów

— Jacek Dajda [[dajda@agh.edu.pl](mailto:dajda@agh.edu.pl)] —

---

1

## Agenda

- Wprowadzenie
  - Rodzaje diagramów
  - Modelowanie obiektowe
  - Reguły SOLID
- 

2

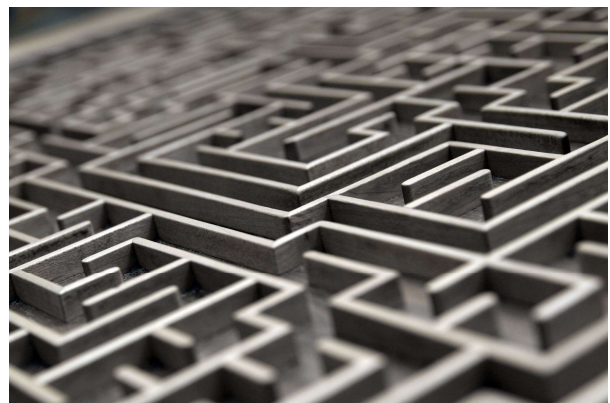
# Wprowadzenie



3

## Problemy dużych systemów informatycznych

- Wysoka złożoność
  - Poszczególne elementy projektowane niezależnie
  - Tworzone przez wiele osób
- Długi czas powstawania
  - Zapominanie o poszczególnych częściach
  - Rotacja pracowników
- Dokumentacja
  - Sprzedawane zewnętrznym podmiotom
  - „Zamrażane” projekty/moduły



4

## Problemy zespołów programistów

- Wybór „najlepszego” sposobu implementacji danej funkcjonalności
- Uzgadnianie pomysłów wewnątrz zespołu
- Utrzymanie jak najprostszej architektury
- Rotacja pracowników
- Wprowadzanie nowych ludzi do projektu
- Zapamiętywanie „starych” decyzji



5

## Rozwiązanie

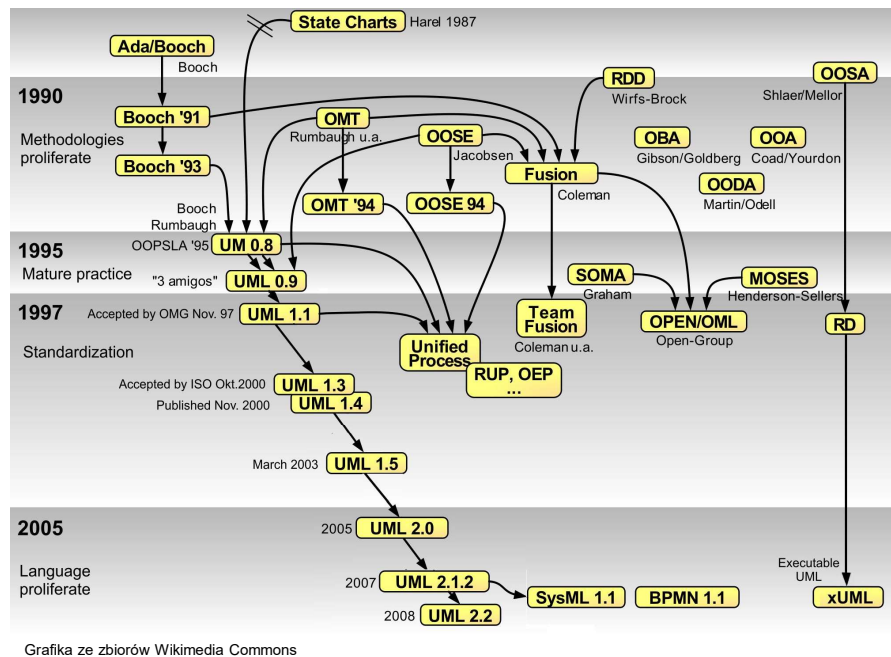
**Ustandaryzowany format  
modelowania systemów  
informatycznych!**



6

# Języki modelowania

## Historia



7

# UML

## Definicja wg OMG

*The Unified Modeling Language (UML) is a graphical language for visualizing, specifying, constructing and documenting the artifacts of a software-intensive system. The UML offers a standard way to write a system's blueprints, including conceptual things such as business processes and system functions as well as concrete things such as programming language statements, database schemas, and reusable software components.*

8

## UML

### Co to?

- *Unified Modeling Language*
- Ustandaryzowany język modelowania wykorzystywany w informatyce
- Stworzony w latach 1994-1997 przez *three amigos* (G. Booch, I. Jacobson, J. Rumbaugh)
- Rozwijany przez *Object Management Group* (OMG)
- Czerwiec 2005 – UML 2.0
- Obecna wersja 2.5.1 (Grudzień 2017)
- Spotyka się także języki wykonywalne jak xUML, fUML, ALF, które są podzbiorami UML



## Bibliografia

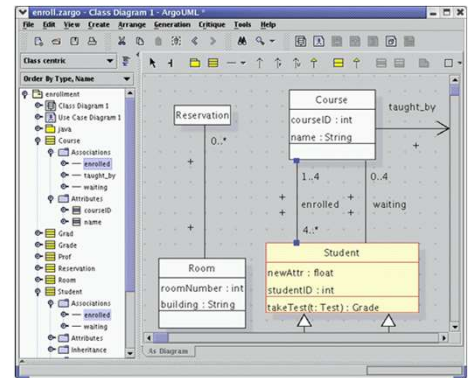
- Brett D. McLaughlin, Gary Pollice, David West  
„Head First Object-Oriented Analysis and Design”
- Martin Fowler  
„UML Distilled” („UML w kropelce”)
- Grady Booch, James Rumbaugh, Ivar Jacobson  
„UML – przewodnik użytkownika”
- James Martin, James J. Odell  
„Podstawy metod obiektowych”



# Narzędzia do projektowania UML

## Darmowe

- Desktopowe
  - ArgoUML
  - Violet UML
- Online
  - <https://www.draw.io>
- Text-to-UML (online)
  - UMLet, <http://www.umlet.com/umletino>
  - yuml.me, <http://yuml.me>
  - PlantUML, <http://plantuml.com>

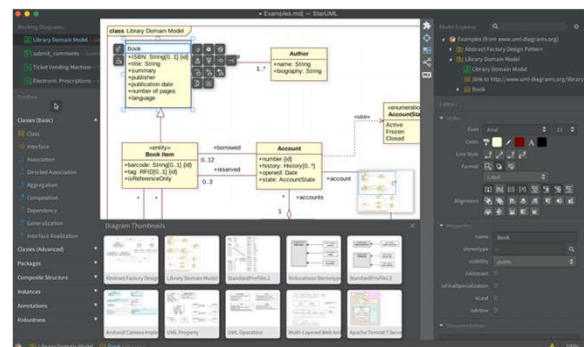


11

# Narzędzia do projektowania UML

## Płatne

- Visual Paradigm (ale jest darmowa też wersja Community)
- Enterprise Architect
- StarUML (dostępny bezterminowy trial)
- Gliffy (online): [www.gliffy.com](http://www.gliffy.com)
- Microsoft Visio (częściowo)
- IntelliJ (diagram klas)



12

# Diagramy

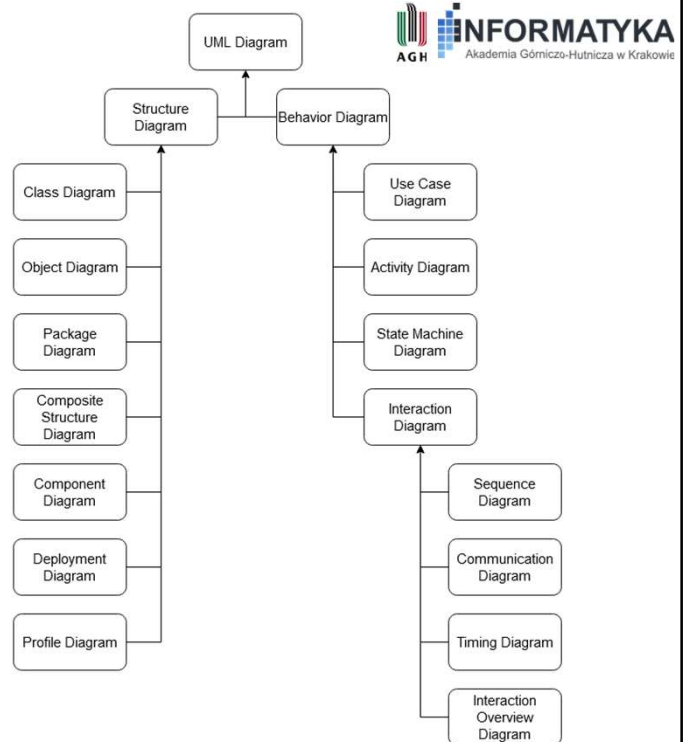


13

## Diagramy UML

### Podział

- UML 2.5.1 definiuje 14 rodzajów diagramów.
- Można je pogrupować w kategorie jak na rysunku obok.



14

## Podstawowe diagramy UML

- modelowanie wymagań – diagramy przypadków użycia
- modelowanie struktury – diagramy klas
- modelowanie zachowań (obiektów) – diagramy interakcji (sekwencji, komunikacji)



15

## Diagramy struktur

- **Klas (najczęściej spotykane, ang. class diagram)**
- Obiektów (ang. object diagram)
- Komponentów (ang. component diagram)
- Wdrożenia (ang. deployment diagram)
- (od UML 2.0) Struktur złożonych (ang. composite structure diagram)
- (od UML 2.0) Pakietów (ang. package diagram)
- (od UML 2.2) Profili (ang. profile diagram)



16



## Diagramy zachowań

- Czynności (ang. activity diagram)
- **Przypadków użycia (ang. use case diagram)**
- Maszyny stanów (ang. state machine diagram) (dla UML 1.x Stanów, ang. statechart diagram)
- Interakcji (diagram abstrakcyjny)
- Komunikacji (ang. communication diagram) (dla UML 1.x Współdziałania, ang. collaboration diagram)
- **Sekwencji (ang. sequence diagram)**
- (od UML 2.2) Czasowe (ang. timing diagram)
- (od UML 2.2) Przeglądu interakcji (ang. interaction overview diagram)

17

## Diagramy UML

### Wprowadzenie

Diagramy składają się z trzech kategorii jednostek:

- Klasyfikatory (*Classifiers*) – zbiory *obiektów*,
  - Obiekt (*object*) – jednostka mająca stan i będąca w relacji do innych obiektów.
- Wydarzenia (*Events*) – zbiory *wystąpień*,
  - Wystąpienie (*occurrence*) – coś, co się dzieje i ma konsekwencje dla systemu.
- Zachowania (*behaviors*) – zbiory możliwych *wykonań*,
  - Wykonanie (*execution*) – wydarzenie się zbioru akcji w pewnym czasie, które mogą wygenerować lub odpowiedzieć na *wystąpienia*, w tym również zmienić stan *obiektów*.

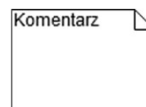
18

# Diagramy UML

## Rozszerzenia

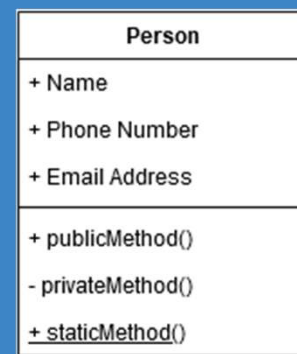
- Stereotypy – sposób na rozbudowanie możliwości języka przez dookreślenie jego elementów. Zapisywane w <<...>>
- Komentarze – pozwalają dopowiedzieć pewne rzeczy, które na podstawie samego diagramu mogłyby być nieoczywiste

-----<<include>>----->



19

# Diagram klas



20

## Model obiektowy

- Połączenie struktur danych z operacjami z nimi związanymi
- Każdy obiekt:
  - wie czym jest (tożsamość)
  - wie jaki jest (atrybuty)
  - wie co potrafi (zachowania)
- Model składa się z definicji obiektów oraz zależności i zachowań pomiędzy nimi



21

## Modelowanie obiektowe - pojęcia

- Obiekt – pojedynczy byt o konkretnym stanie
- Klasa – abstrakcja nad obiektami o tej samej tożsamości
- Metoda – rodzaj zachowania pewnego obiektu
- Atrybut – cecha stanu obiektu

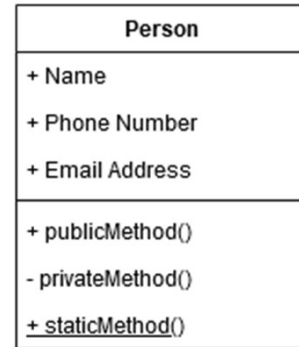


22

## Diagram klas

### Wygląd elementu

- Podstawowy element diagramu klas – klasa składa się z trzech części:
  - Nazwa – obowiązkowa, zazwyczaj pogrubiona,
  - Pola – opcjonalne, reprezentują stan obiektu,
  - Metody – opcjonalne, reprezentują funkcjonalności obiektu.
- Elementy dwóch ostatnich korzystają z modyfikatorów:
  - + – elementy publiczne,
  - # – elementy zastrzeżone (*protected*),
  - ~ – elementy widoczne wewnątrz pakietu,
  - - – elementy prywatne.
- Elementy przynależące do klasyfikatora zamiast do obiektu (tj. statyczne) oznacza się podkreśleniem.



23

## Diagram klas

### Klasy abstrakcyjne i interfejsy

- Klasy abstrakcyjne (z niepełną implementacją) są zapisywane kursywą.
- Interfejsy są zwykle oznaczane słowem kluczowym <<interface>> przed nazwą.


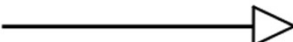
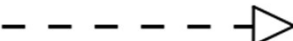



*AbstractWriter*

«interface»  
Computable

24

## Diagram klas

### Rodzaje połączeń

- Asocjacja 
- Dziedziczenie 
- Implementacja 
- Zależność 
- Agregacja 
- Kompozycja 

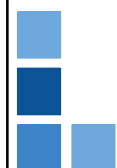
Grafika ze zbiorów Wikimedia Commons



## Diagram klas

### Rodzaje połączeń

- Asocjacja – Połączenie pomiędzy dwoma klasami. Jest nadzbiorem kompozycji i agregacji. Może być jedno lub dwustronne (oznaczane wtedy linią bez grotu na żadnym z końców)
- Dziedziczenie – Strona wskazywana przez grot strzałki jest nadklasą w stosunku do tej, z której strzałka wychodzi
- Implementacja – Grot strzałki wskazuje na interfejs, który jest implementowany przez daną klasę



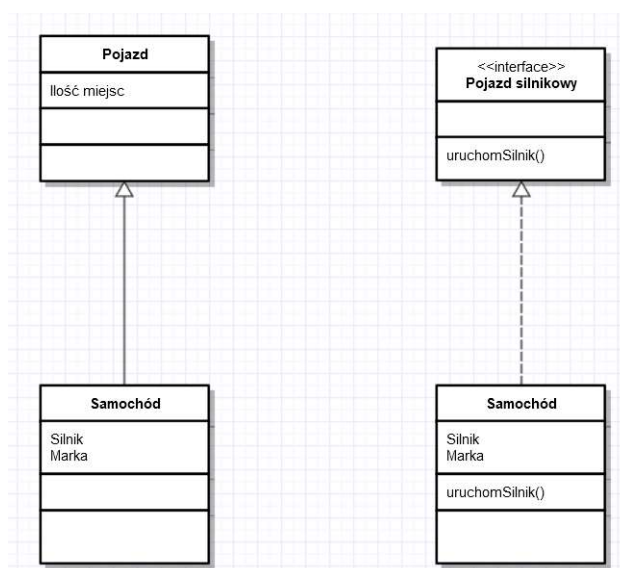
## Diagram klas

### Rodzaje połączeń

- Zależność – Połączenie między dwoma elementami modelu. Oznacza zależność: jeśli zmieni się główny z obiektów, drugi też może się zmienić.
- Agregacja – Relacja posiadania. Obiekt agregujący posiada wskazanie na agregowany, jednak ich cykle życia są niezależne.
- Kompozycja – Połączenie reprezentujące relację bycia częścią większej całości. Cykl życia „części”, tj. elementu, z którego wychodzi „strzałka” jest zależny od obiektu wskazywanego przez nią – całości. Przykład: Wydział jest częścią uniwersytetu.

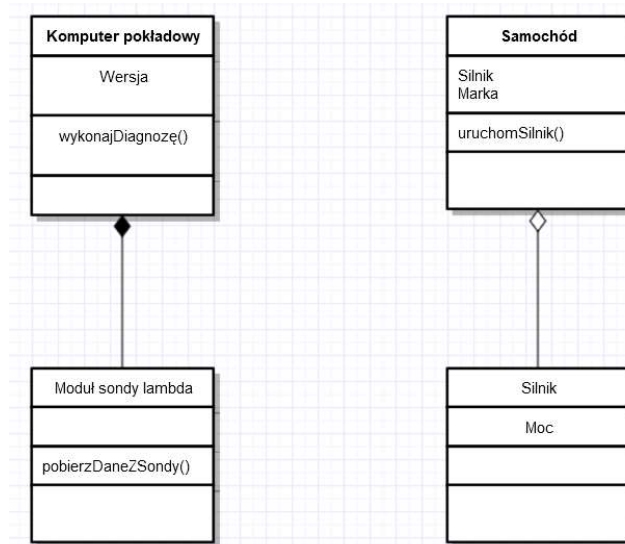
27

## Dziedziczenie i implementacja



28

## Agregacja i kompozycja

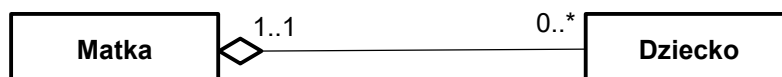


29

## Diagram klas

### Krotności połączeń

- W relacjach pomiędzy obiektami często określamy, ilu obiektów poszczególnych klas one dotyczą.



30

## Diagram klas

### Przykład

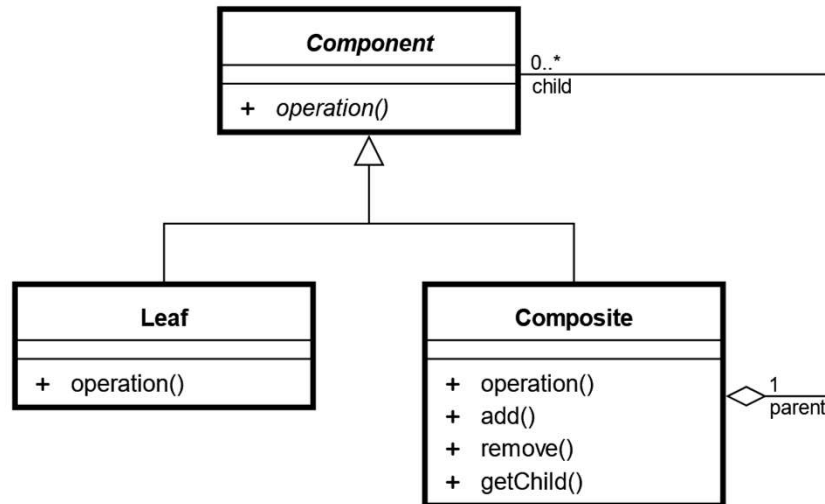
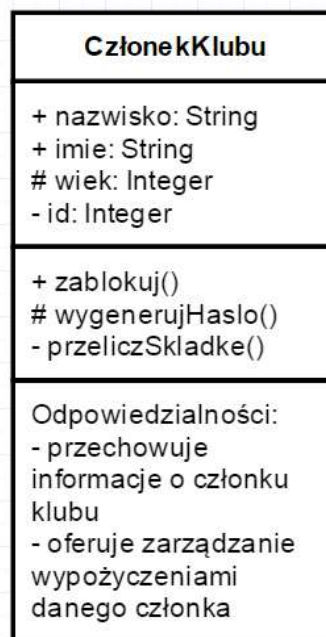


Diagram klas wzorca Kompozyt. Grafika ze zbiorów Wikimedia Commons

31

## Diagram klas

### Kontrola dostępu

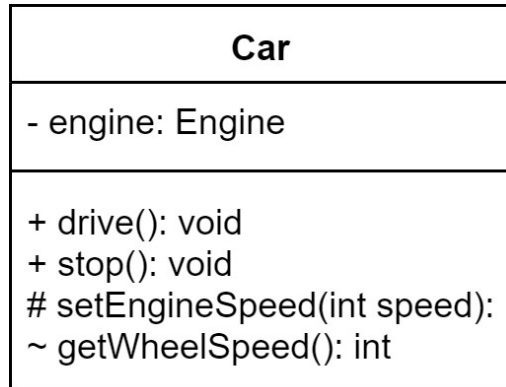


32



## Diagram klas

### Kontrola dostępu



```
public class Car {
    private Engine engine;

    public void drive() {...}

    public void stop() {...}

    private void stopEngine() {...}

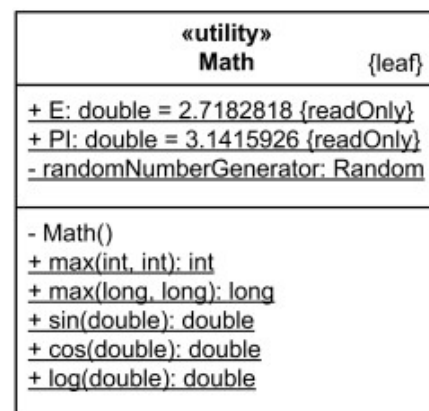
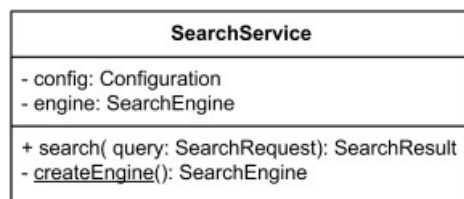
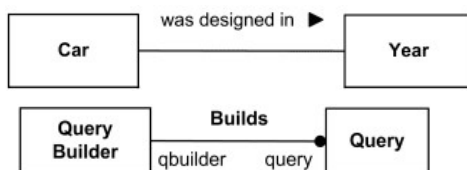
    protected void setEngineSpeed(int speed)
        {...}

    void getWheelSpeed() {...}
}
```

33

## Diagram klas

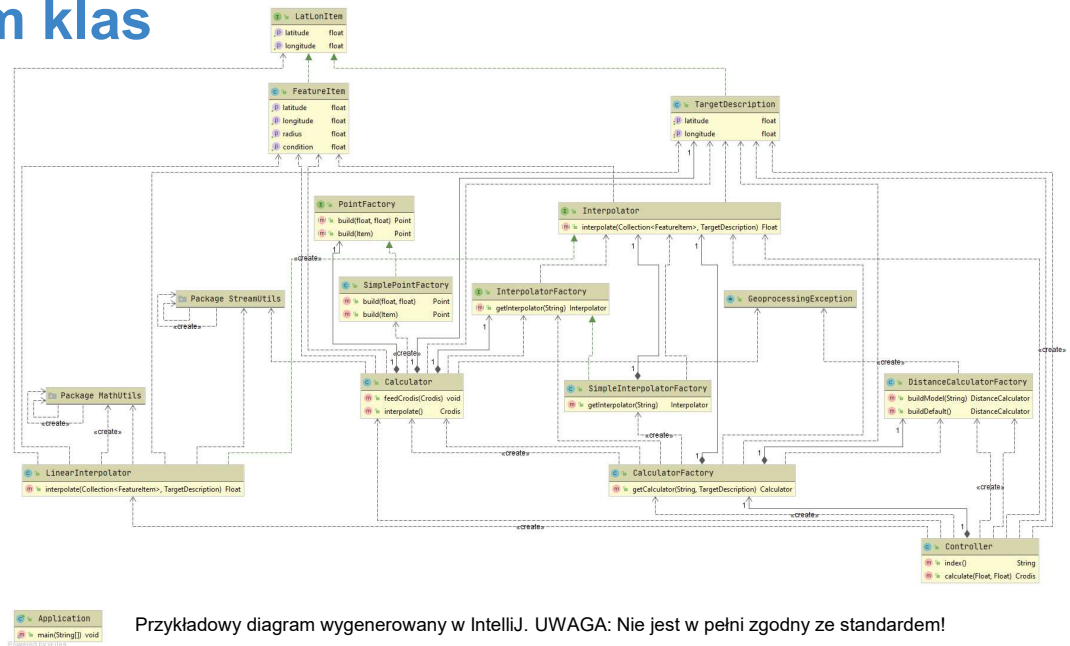
### Opisy powiązań, domyślne wartości, dodatkowe atrybuty



34

# Diagram klas

## Przykład



35

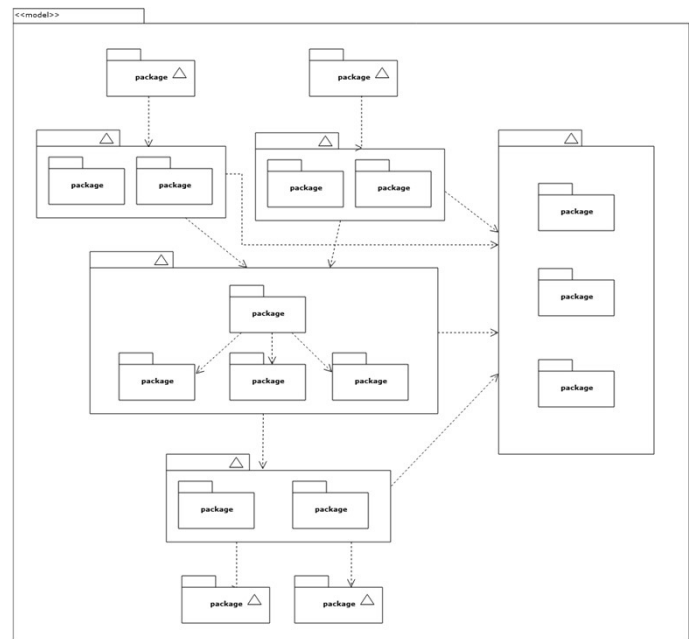
# Diagram pakietów



36

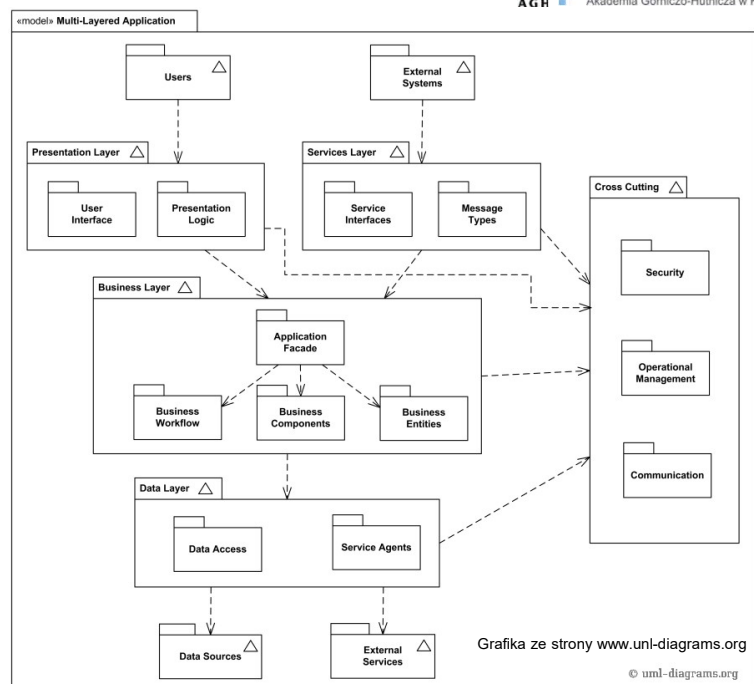
## Diagram pakietów

- Pozwala na spojrzenie na projekt z wyższego poziomu niż poszczególne klasy
- Pomaga uporządkować pakiety. Przydatny przy refactoringu.



37

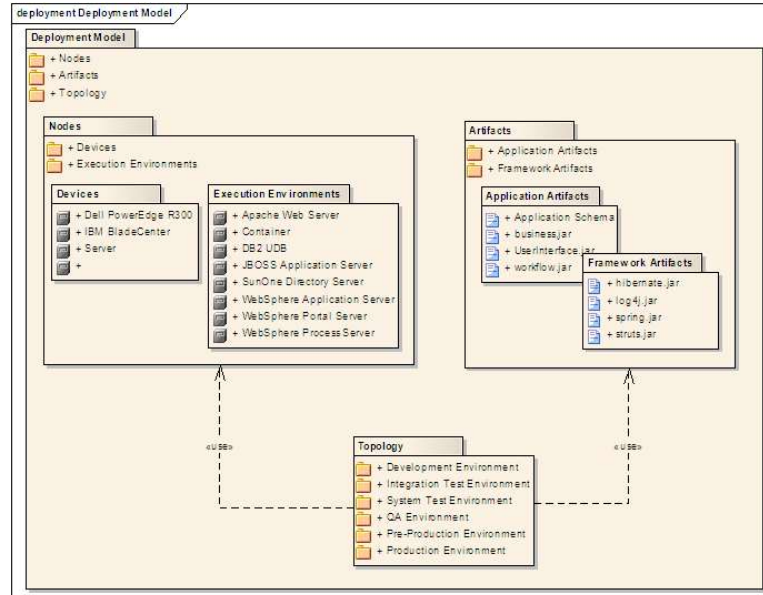
## Diagram pakietów Przykład



38

# Diagram pakietów

## Przykład

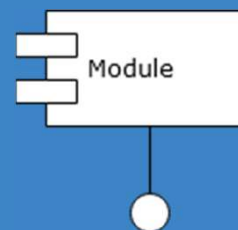


Grafika ze zbiorów Wikimedia Commons

39

# Diagram

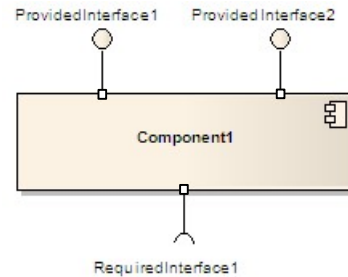
## komponentów



40

## Diagram komponentów

- Pozwala na spojrzenie na system z dużej perspektywy.
- Poszczególne komponenty są zamkniętymi „czarnymi skrzynkami”.
- Możliwe jest zagnieżdżanie elementów.
- Komponenty otwierają się na komunikację jedynie w portach (małe kwadraty na brzegach).
- Z portów wystawione są interfejsy (kulki) lub definicje oczekiwanych interfejsów (półokręgi).
- Wygodne do wyłapywania zbyt złożonej architektury (*spaghetti*) i wdrażaniu nowych osób w projekt.

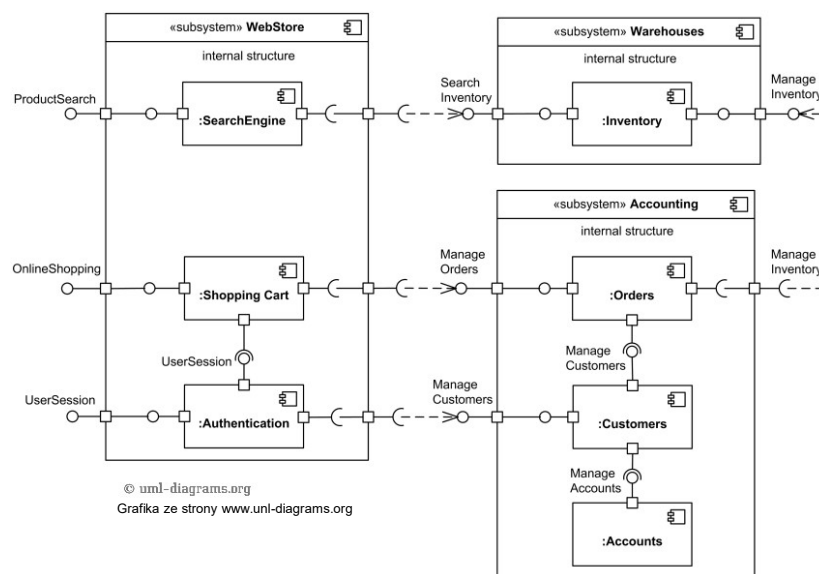


Grafika ze zbiorów Wikimedia Commons

41

## Diagram komponentów

### Przykład



© uml-diagrams.org  
 Grafika ze strony www.uml-diagrams.org

42

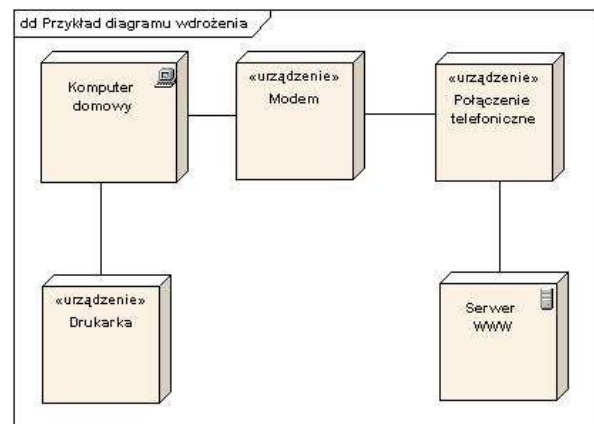
# Diagram wdrożenia



43

## Diagram komponentów

- Ilustruje sposób instalacji systemu na fizycznych urządzeniach
- Pokazuje niezbędne zależności i schematy komunikacji w kontekście urządzeń i tzw. peryferiów



44

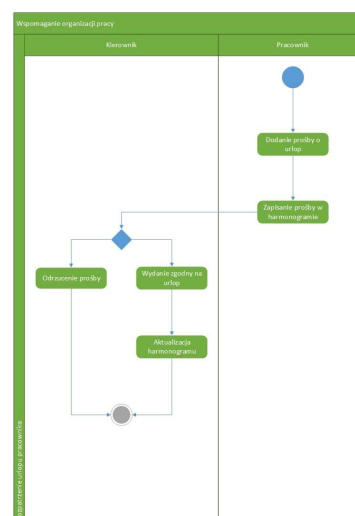
# Diagram aktywności



45

## Diagram aktywności

- Diagram behawioralny – pozwala na śledzenie dynamicznych aspektów systemu.
- Umożliwia rozbudowaną analizę poszczególnych przypadków użycia systemu.
- Elementy diagramu stanowią nierozbijalne na mniejsze elementy zdarzenia
- Przydatny do rozbijania na kroki złożonych procesów, które ma wykonać system.

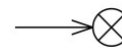
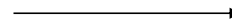
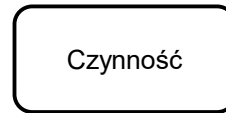


46

## Diagram aktywności

### Elementy

- Czynność (*action*) – podstawowy element diagramu, oznacza niepodzielny fragment zdarzenia wykonywany w trakcie działania systemu. Może wpływać na stan obiektów i generować inne zdarzenia.
- Przepływ (*flow*) – reprezentuje przejście do kolejnego węzła.
- Węzeł rozpoczynający (*initial node*) – moment startu aktywności.
- Węzeł kończący aktywność (*activity final node*) – moment zakończenia aktywności.
- Węzeł przerywania aktywności (*flow final node*) – aktywność, która do niego doszła zostaje przerywana.

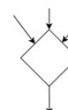
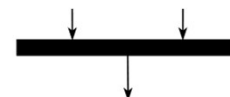
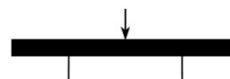


47

## Diagram aktywności

### Elementy

- Węzeł rozdzielający (*fork node*) – pozwala na rozdzielenie procesu na dwie osobne ścieżki biegnące niezależnie od siebie.
- Węzeł łączący (*join node*) – pozwala połączenie biegnących niezależnie ścieżek. Implikuje czekanie aż wszystkie z nich się wykonają.
- Węzeł decyzyjny (*decision node*) – pozwala na sprawdzenie warunku i wybranie ścieżki w zależności od niego.
- Węzeł scalający (*merge node*) – mogą do niego wchodzić aktywności, które zmierzały różnymi ścieżkami. Wychodzi z niego tylko jedna.

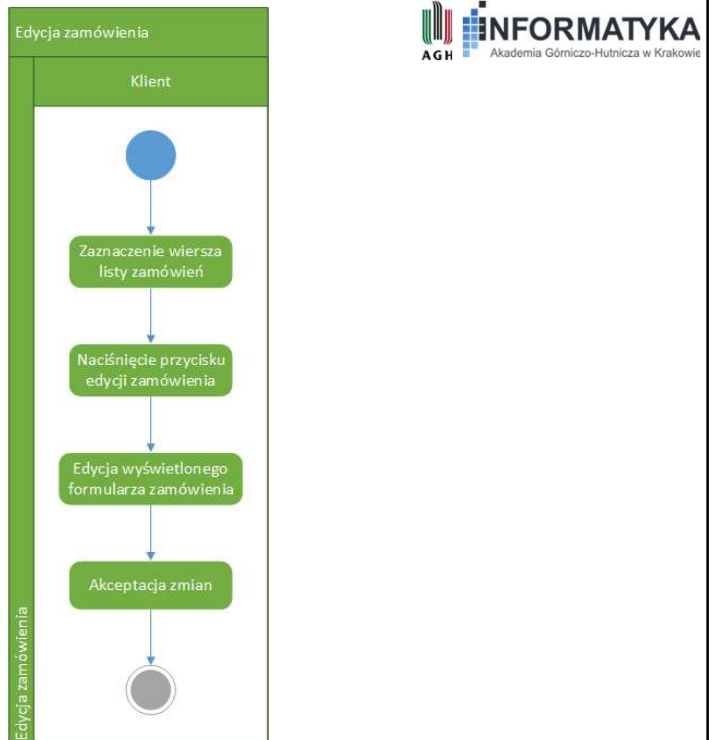


48



## Diagram aktywności

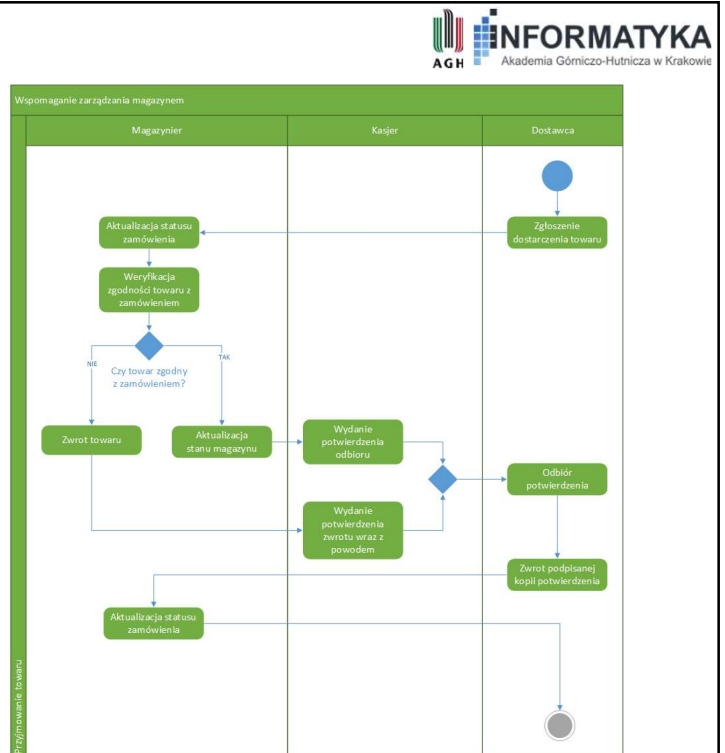
### Przykład



49

## Diagram aktywności

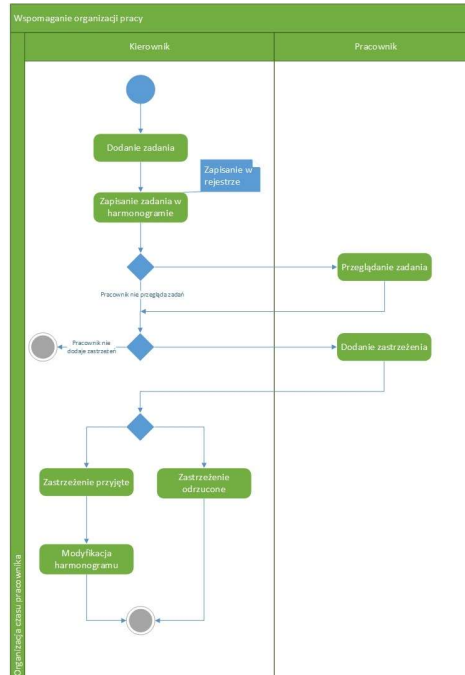
### Przykład



50

## Diagram aktywności

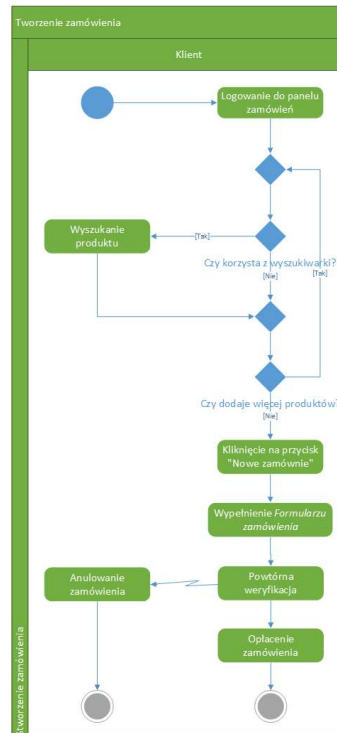
### Przykład



51

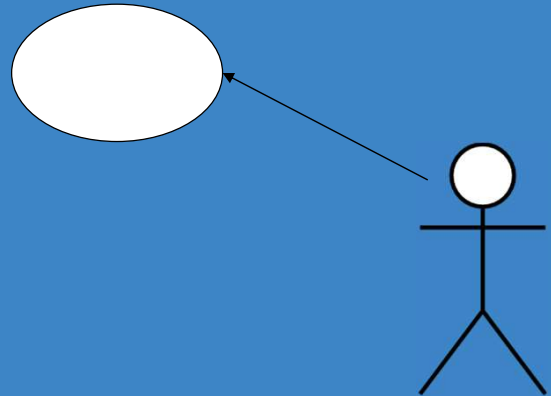
## Diagram aktywności

### Przykład



52

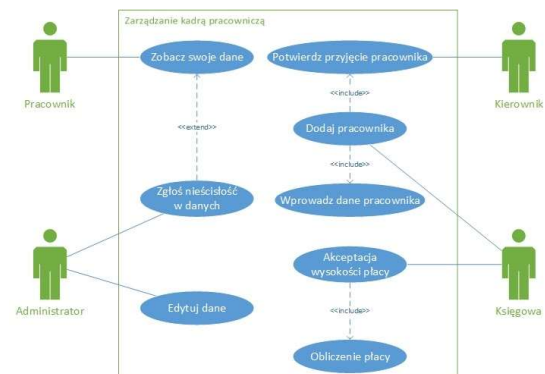
# Diagram przypadków użycia



53

## Diagram przypadków użycia

- Pozwala na spojrzenie na system z perspektywy jego użytkowników.
- Pokazuje jego interakcje z zewnętrznymi aktorami (niekoniecznie ludźmi).
- Przydatny przy podejściu Behavior Driven Development
- Każdy przypadek użycia (*use case*) może być opisany np. za pomocą diagramu aktywności.

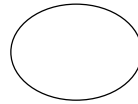


54

## Diagram przypadków użycia

### Elementy

- Przypadek użycia (*use case*) – zestaw funkcjonalności systemu, które spełniają konkretną potrzebę użytkownika. Każdy z nich musi być połączony z aktorem (lub rozszerzać / zawierać się w innym)
- Aktor – byt zewnętrzny w stosunku do systemu, który wchodzi z nim w interakcję.

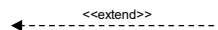
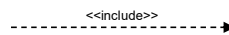


55

## Diagram przypadków użycia

### Elementy

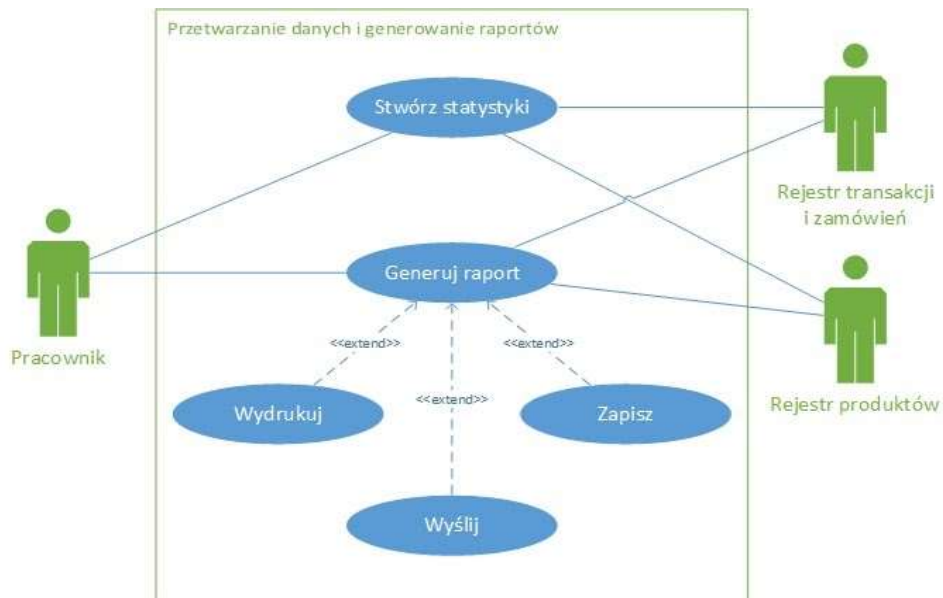
- Zawieranie – sposób na rozbicie przypadku na mniejsze części. Oznaczamy w ten sposób te przypadki, które stanowią część przypadku połączonego z aktorem. Grot strzałki wskazuje na nadrzędny.
- Rozszerzenie – jeśli przy okazji wykonywania jednego przypadku opcjonalnie może być wykonany inny korzystamy z tej relacji. Grot wskazuje na przypadek nadrzędny.



56

## Diagram przypadków użycia

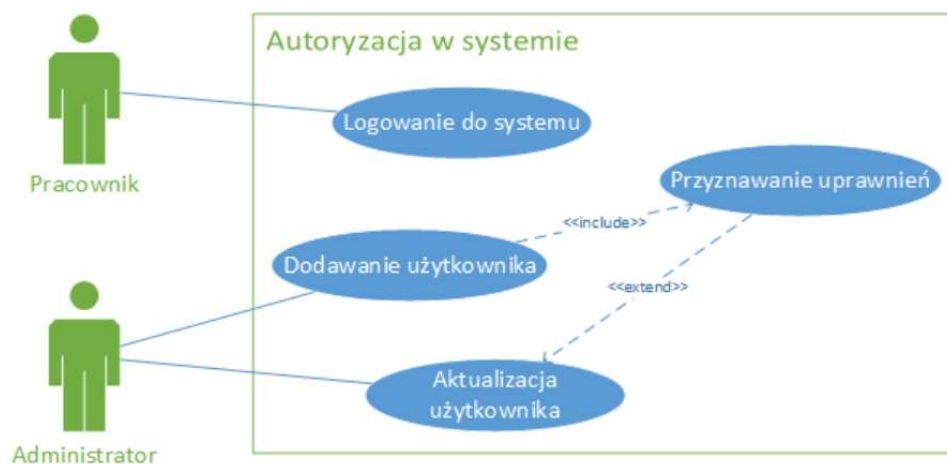
### Przykład



57

## Diagram przypadków użycia

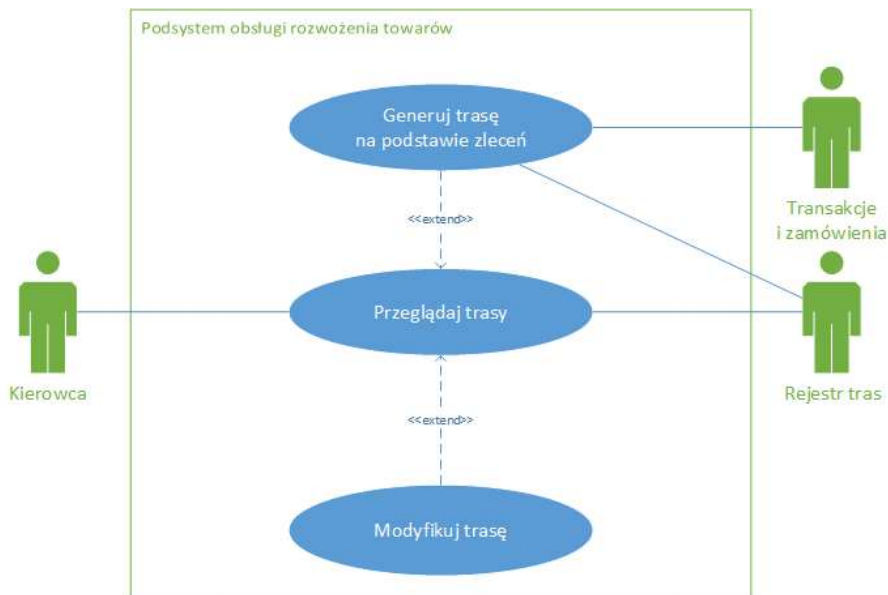
### Przykład



58

## Diagram przypadków użycia

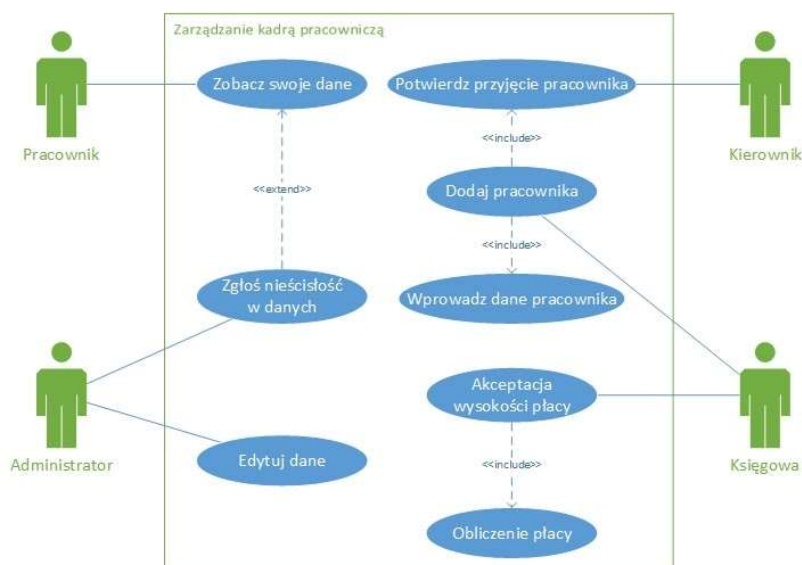
### Przykład



59

## Diagram przypadków użycia

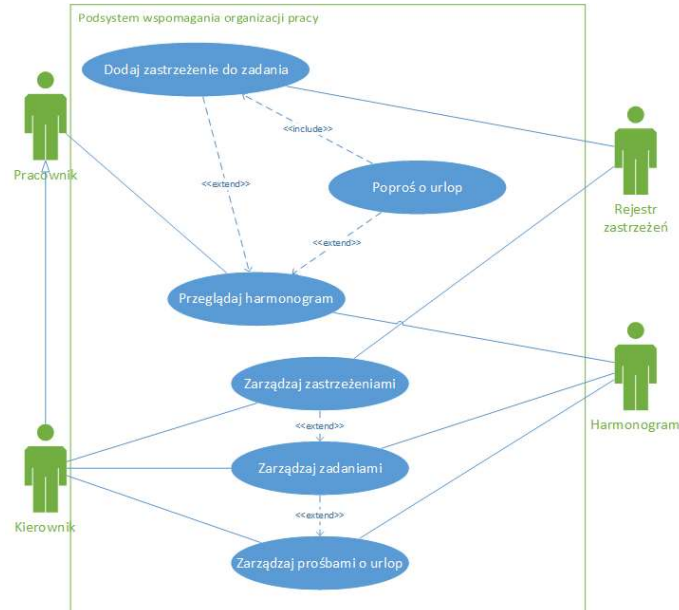
### Przykład



60

## Diagram przypadków użycia

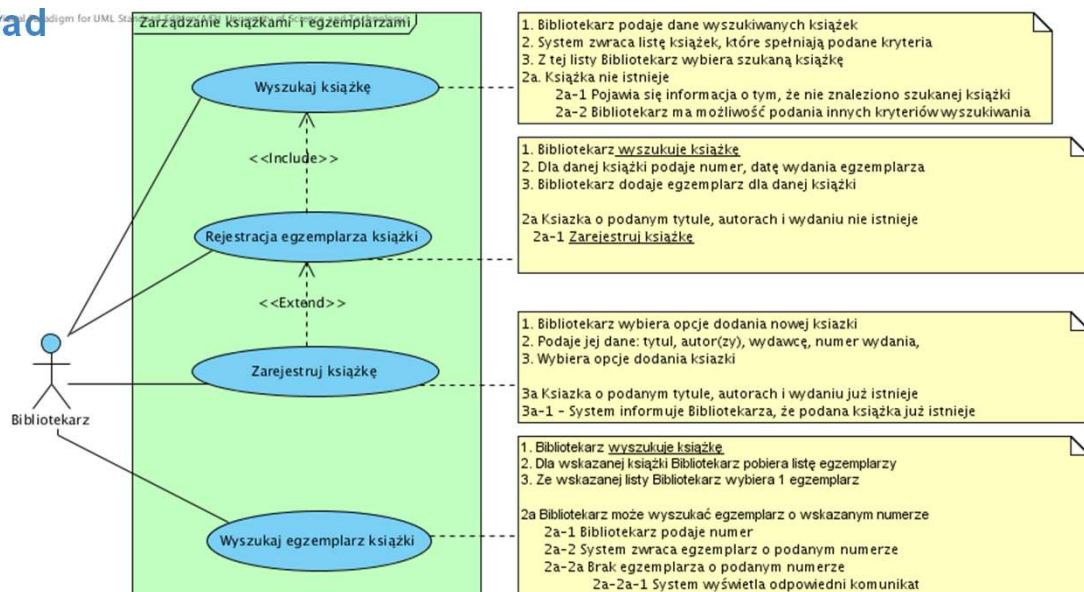
### Przykład



61

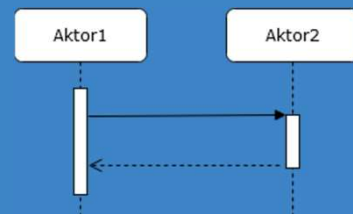
## Diagram przypadków użycia

### Przykład



62

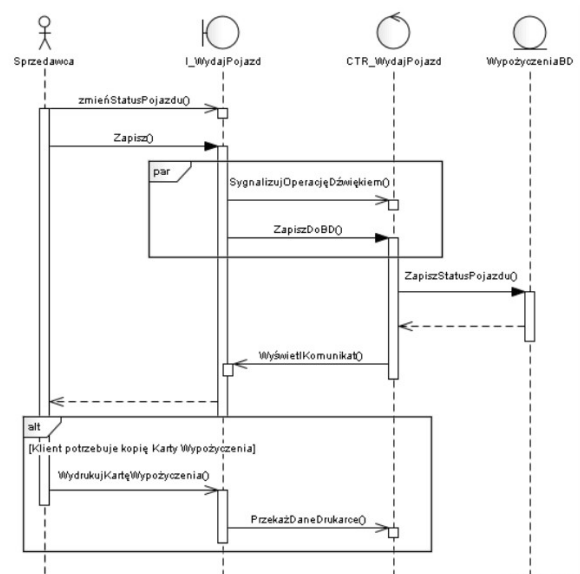
# Diagram sekwencji



63

## Diagram sekwencji

- Ocena komunikatów krążących między obiektami
- Pozwala śledzić czas życia obiektu/aktora/modułu/encji danych
- W pionowych liniach zapisujemy aktorów.
- Upływ czasu sygnalizuje przemieszczanie się w dół diagramu.
- Komunikaty zapisujemy jako poziome linie.
- W trakcie aktywności aktorów są oni zaznaczeni prostokątem. Przed i po aktywowaniu się linią przerywaną.



Grafika ze strony wolski.pro

© Michał Wolski

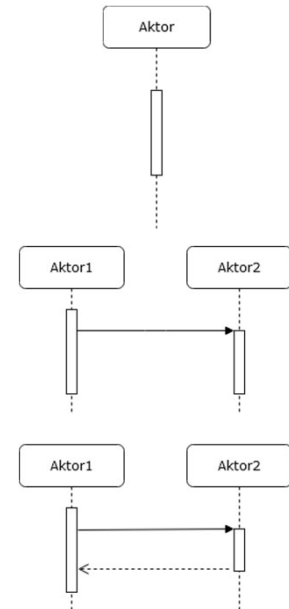
64



## Diagram sekwencji

### Elementy

- Linia życia (*lifeline*) – linia przerywana, widoczna tak długo jak żyje aktor (zazwyczaj cały diagram)
- Czas aktywacji (*activation*) – podłużny prostokąt, widoczny tak długo jak aktor wysyła wiadomości lub czeka na odpowiedź
- Wiadomość asynchroniczna (*asynchronous call*) – komunikacja jednostronna, nadawca nie czeka na odpowiedź
- Wiadomość synchroniczna – komunikacja dwustronna, nadawca czeka na odpowiedź (linia przerywana)

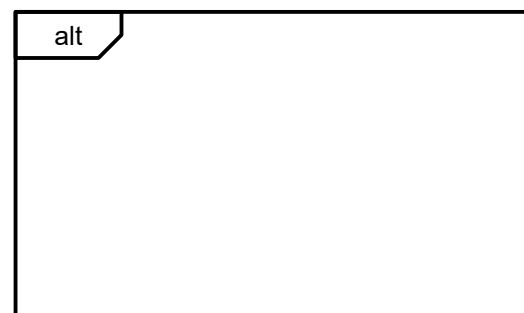


65

## Diagram sekwencji

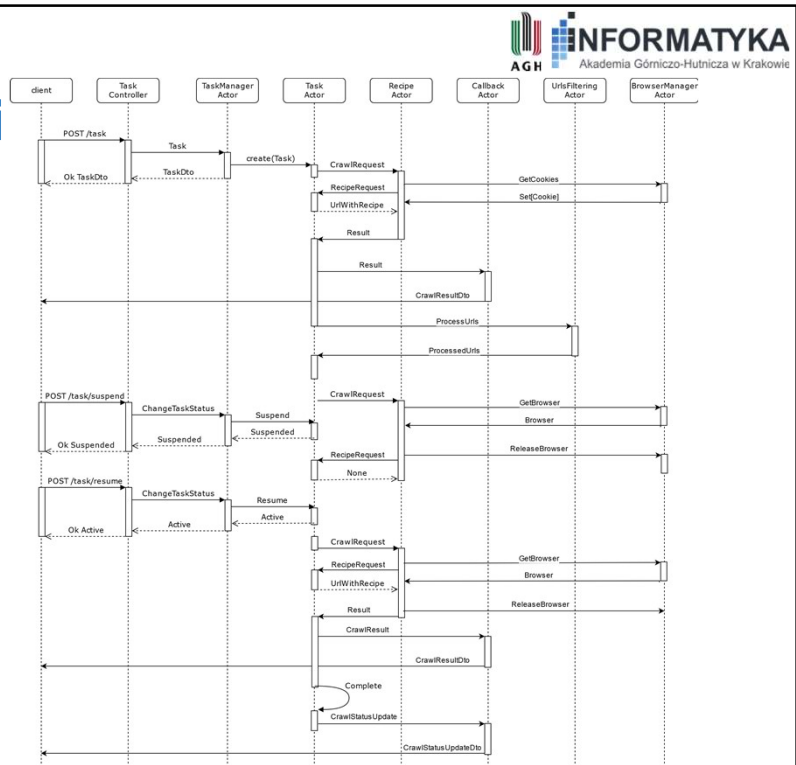
### Bloki specjalne

- alt – ścieżki alternatywne
- opt – fragment warunkowy
- par – fragment współbieżny
- loop – fragment wykonywany w pętli



66

## Diagram sekwencji



67

## Modelowanie obiektowe



68

## Modelowanie obiektowe

Opanowywanie złożoności problemu opiera się na powszechnie (choć nie zawsze świadomie) stosowanych zasadach:

- rozróżnianie i opisywanie poszczególnych obiektów – ich cech i możliwości wykorzystania
- klasyfikowanie obiektów i definiowanie pojęć
- określanie relacji między obiektami (powiązania)
- znajdowanie zależności między pojęciami (skojarzenie)
- wyprowadzanie nowych (generalizacja)



69

## Wprowadzenie Analiza vs. projekt

### OOAD - Object-oriented analysis and design

*Object-oriented analysis and design (OOAD) is a popular technical approach for analyzing and designing an application, system, or business by applying object-oriented programming, as well as using visual modeling throughout the development life cycles to faster better stakeholder communication and product quality*



70

## Wprowadzenie

# Jak opracować i utrzymać dobry projekt?

- Projektować, projektować, projektować...
  - praca zespołowa
  - UML
- Stosować dobre i sprawdzone standardy
  - zasady dobrego projektowania
  - wzorce projektowe



71

## Wprowadzenie

# Jak/kiedy/w jaki sposób używać UML

- “a priori” dokładne jako specyfikacja do implementacji - rzadziej, bardzo kosztowne
- “a posteriori” w celu opisu implementacji na potrzeby dokumentacji - częściej, mniej kosztowne
- w trakcie całego procesu - jako narzędzie do komunikacji i dyskusji (np. przy tablicy), najczęściej, niski koszt, duże zyski :-)



72

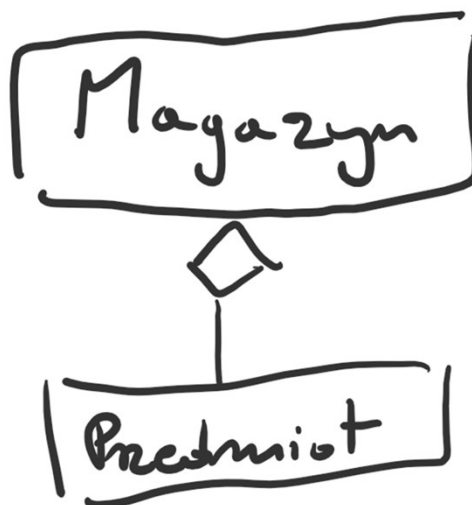
## Projekt koncepcyjny

- Narzędzie dyskusji i pracy zespołowej
- Szybkie i tanie
- Ogólny widok na system lub dany problem
- Niekonieczne ze szczegółami



73

## Projekt koncepcyjny



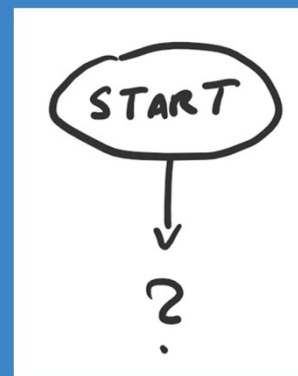
74

## Projekt koncepcyjny



75

## Projektowanie: jak zacząć?



76

Jak zacząć?

## #1: Wstępna identyfikacja bytów

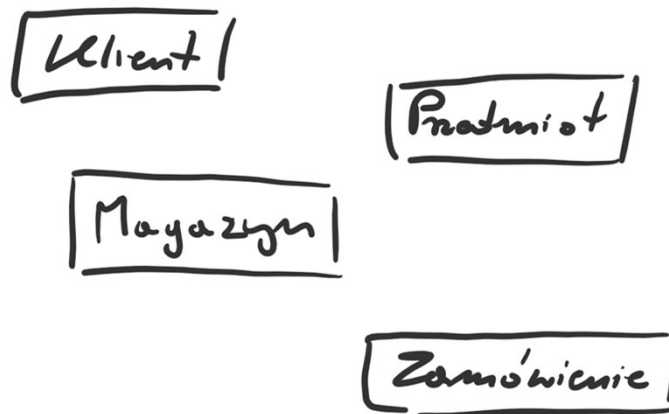
- Często podstawowy krok
- Rozbijamy problem na klasy
- Szczegóły na razie nieistotne
- Identyfikacja bytów obecna w całym procesie projektowania



77

Jak zacząć?

## #1: Wstępna identyfikacja bytów



78

## Jak zacząć?

### #2: Projektowanie z punktu widzenia struktury

- Analizujemy dziedzinę i przypisujemy bytom atrybuty
- Zalety:
  - Naturalne podejście
  - Dobry punkt wyjścia
  - Dobrze do systemów bazodanowych



## Jak zacząć?

### #2: Projektowanie z punktu widzenia struktury

- Wady:
  - Koncentracja na nieistotnych lub oczywistych szczegółach zamiast istocie problemu
  - Złożona logika wymaga spojrzenia od strony funkcji systemu i scenariuszy





## Jak zacząć?

### #2: Projektowanie z punktu widzenia zachowań/funkcji

- Spojrzenie od strony funkcjonalnej systemu
- Polega na przypisaniu odpowiedzialności/funkcji poszczególnym klasom i określenie relacji pomiędzy nimi
- Przydatne jest wyspecyfikowanie przykładowych scenariuszy użycia systemu i na ich podstawie weryfikowanie kompletności modelu
- Podejście to realizuje technika kart CRC

81

## Istota modelowania obiektowego

- hermetyzacja stanu i zachowań jednostek
- które ze sobą współpracują poprzez wykorzystanie udostępnianych usług
- w realizacji określonych scenariuszy działania

Dwie warstwy modelowania:

1. obiekty i relacje między nimi (model wybranej sytuacji – „migawka” z życia systemu)
2. pojęcia i relacje między nimi (model dziedziny – ogólna struktura systemu)

82

## Od pojęć do klas

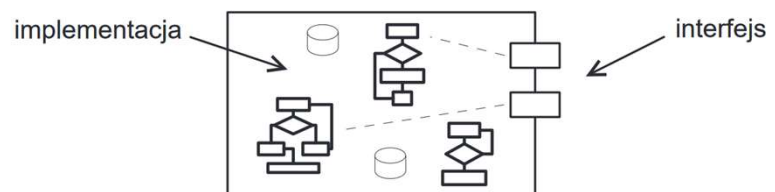
- pojęcia jako klasy
- pojęcia jako atrybuty
- pojęcia jako usługi
- pojęcia jako skojarzenia



83

## Hermetyzacja: zamykanie złożoności

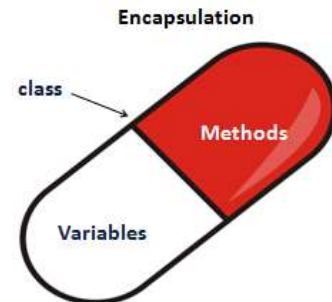
- podział na elementy składowe (modularyzacja) ułatwia modelowanie struktury
- hermetyzacja oznacza zamykanie struktury wewnętrznej składowej (implementacji) za dobrze zdefiniowanymi granicami (interfejsem)
- celem jest zwiększenie odporności na zmiany (zmiany implementacji ograniczają się do składowej, zależności zewnętrzne specyfikowane są explicite przez interfejs)



84

## Hermetyzacja: zamykanie złożoności

- Pozwala na projektowanie łatwego do użycia interfejsu programistycznego (API) dla innych programistów – użytkowników naszej klasy
- Ułatwia dalszy rozwój systemu poprzez czytelne API i upraszczanie interfejsów klas



Source: <https://www.quora.com/How-does-a-class-enforce-data-hiding-abstraction-and-encapsulation>

85

## Hermetyzacja: przykład

Car
- engine: Engine
+ drive(): void + stop(): void - stopEngine(): void

86

## Hermetyzacja: przykład

```
public class Car {  
    private Engine engine;  
    public void drive() {...}  
    public void stop() {...}  
    private void stopEngine() {...}  
}
```



87

## Zależności pomiędzy klasami

- stosowanie hermetyzacji wiąże się ze stopniem rozdrobnienia struktury systemu
- miary zależności specyfikacji (implementacji): spójność i sprzężenie
- zasada jednej odpowiedzialności (ang. SRP: Single Responsibility Principle)
- zasada segregacji interfejsów (ang. ISP: Interface Segregation Principle)



88

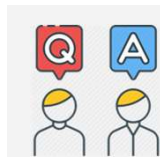


# Reguły SOLID



91

## Reguły SOLID Co to jest dobry projekt?



**Pytanie:**

Jakie są symptomy złego projektu?

- Rigidity (pol. sztywność)
- Immobility (pol. trudność w zmianie)
- Fragility (pol. delikatność)
- Viscosity (pol. lepkość)

92

## Reguły SOLID

### Rigidity

- wpływ zmian jest nieprzewidywalny
- każda zmiana wywołuje kaskadę zmian w zależnych modułach
- miła “krótka robótka” staje się rodzajem niekończącego się maratonu
- koszt staje się nieprzewidywalny



93

## Reguły SOLID

### Fragility

- oprogramowanie ma tendencję do psucia się w wielu miejscach przy każdej zmianie
- błędy pojawiają się w miejscach niepowiązanych koncepcyjnie
- przy każdej naprawie oprogramowanie psuje się w niespodziewany sposób



94

## Reguły SOLID Immobility

- **jest niemal niemożliwe ponowne użycie interesujących fragmentów oprogramowania**
- użyteczne moduły mają za dużo zależności
- koszt przepisania/skopiowania jest mniejszy w porównaniu do ryzyka związanego z wydzielaniem tych części
- koszt staje się nieprzewidywalny



## Reguły SOLID Viscosity

- “hack” jest łatwiejszy w implementacji niż rozwiązanie w ramach projektu
- działania chroniące projekt są trudne do opracowania i wdrożenia
- jest dużo łatwiej robić rzeczy niewłaściwe niż te właściwe... :-)





## Reguły SOLID

### Przyczyny złego projektu

- Jaka jest przyczyna, że projekt staje się sztywny (Rigidity), trudny w zmianie (Immobility), delikatny (Fragility) i lepki (Viscosity)?

nieodpowiednie zależności  
pomiędzy modułami



97

## Reguły SOLID

### Dobry projekt

- Więc, jakie są cechy dobrego projektu?

wysoka spójność

małe zazębianie się/  
sprzężenie



98

## Reguły SOLID

### Dobry projekt

- Jak możemy osiągnąć (stworzyć) dobry projekt?



Bądźmy **SOLIDni** !!!



## Reguły SOLID

- **S** RP - Single Responsibility Principle
- **O** CP - Open Closed Principle
- **L** SP - Liskov Substitution Principle
- **I** SP - Interface Segregation Principle
- **D** IP - Dependency Inversion Principle



## Reguły SOLID

### Single Responsibility Principle

- Dana jednostka oprogramowania powinna posiadać tylko jedną odpowiedzialność/zadanie
- Mówi się też, że “powinna posiadać tylko jeden powód do zmiany”
- Dostrzeganie różnych odpowiedzialności bywa trudne



101

## Reguły SOLID

### Single Responsibility Principle - przykład #1



**Pytanie:**

Czy SRP jest tutaj spełnione?

**<<interface>>**  
**Modem**

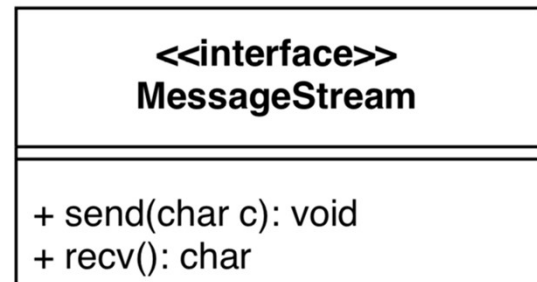
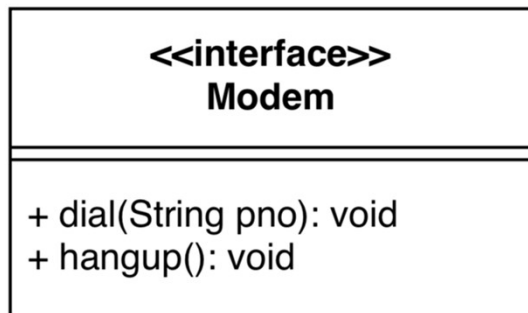
```
+ dial(String pno): void
+ hangup(): void
+ send(char c): void
+ recv(): char
```



102

## Reguły SOLID

### Single Responsibility Principle - przykład #1



103

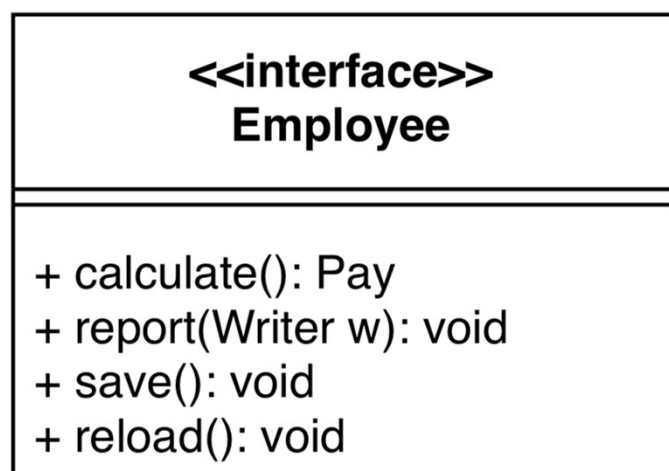
## Reguły SOLID

### Single Responsibility Principle - przykład #2



**Pytanie:**

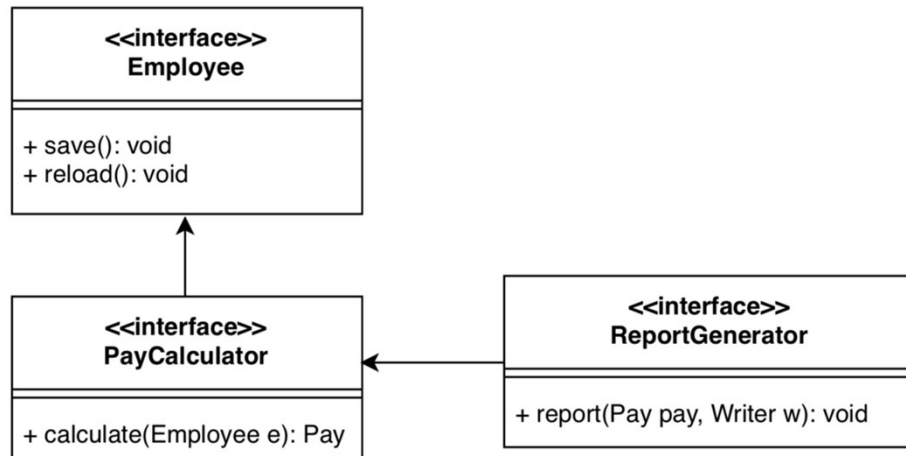
Czy SRP jest tutaj spełnione?



104

## Reguły SOLID

### Single Responsibility Principle - przykład #2



105

## Reguły SOLID

### Single Responsibility Principle

- Identyfikuj i grupuj elementy, które wspólnie się zmieniają (zależą od siebie) z tego samego powodu
- Bądź podejrzliwy w stosunku do wszelkich klas typu \*Manager, \*Controller, \*Handler
- Bądź precyzyjny w nazewnictwie klas i ich metod

106

## Reguły SOLID

### Open Closed Principle

- jednostki oprogramowania (klasy) powinny być otwarte na rozszerzanie a zamknięte na zmianę
- opracowane w 1998 przez Bertrand Meyer
- tak projektuj swoje klasy by bez zmiany ich budowy można było rozszerzać ich zachowanie/możliwości



107

## Reguły SOLID

### Open Closed Principle - przykład #1



**Pytanie:**

Czy kod spełnia OCP?

```
public void draw(Shape[] shapes) {
    for( Shape shape : shapes ) {
        switch (shape.getType()) {
            case Shape.SQUARE:
                draw((Square)shape);
                break;
            case Shape.CIRCLE:
                draw((Circle)shape);
                break;
        }
    }
}
```



108

## Reguły SOLID

### Open Closed Principle - przykład #2



**Pytanie:**

Czy kod spełnia OCP?

```
public void draw(Shape[] shapes) {
    for( Shape shape : shapes ) {
        shape.draw();
    }
}
```



109

## Reguły SOLID

### Liskov Substitution Principle

Definition #1

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, then S is a subtype of T



110

## Reguły SOLID

# Liskov Substitution Principle

### Definition #2

Given an entity with a behavior and some possible subentities that could implement the original behavior, the caller should not be surprised by anything if one of the sub entities are substituted to the original entity

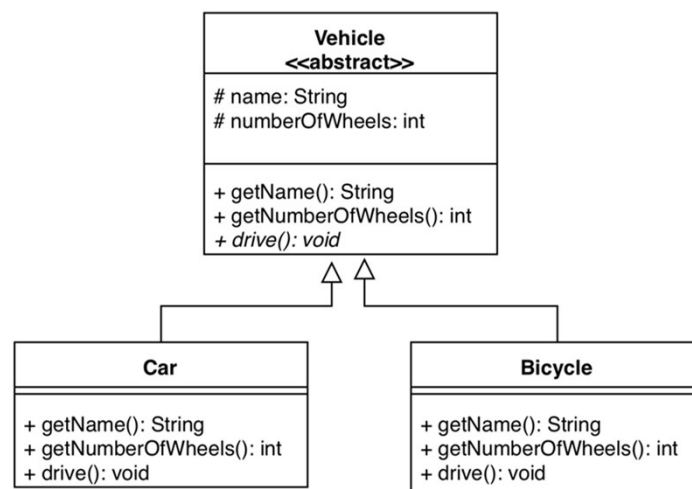


#### Pytanie:

Jakie znane mechanizmy Javy to przypomina?

111

## Liskov Substitution Principle - przykład



112



## Liskov Substitution Principle - przykład

```
public void listParkingLot(Vehicle[] vehicles) {  
    for( Vehicle vehicle : vehicles ) {  
        vehicle.getName();  
    }  
}
```



113

## Reguły SOLID

### Interface Segregation Principle

- Często podczas tworzenia oprogramowania niektóre klasy rozrastają się do sporych rozmiarów ("grube klasy") i trudno nam jest coś z nimi poradzić :(
- Zwykle mają wiele klientów-klas które odwołują się tylko do pewnych grup metod, a innych nie potrzebują, ale niestety od nich zależą
- Niestety zmiana którejś z nich wymaga również zmiany wszystkich zależnych modułów/klas



114

## Reguły SOLID

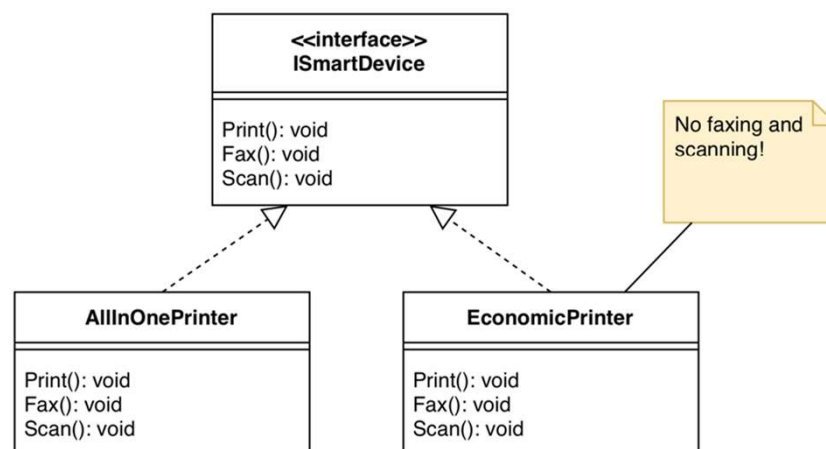
### Interface Segregation Principle

- ISP polega więc na tym, by jednostki nie zależały od całych “grubych klas”, a tylko od przejrzystych i spójnych interfejsów, których używają
- Wówczas zmiana metod w “grubej klasie”, których nie używają nie wpływa na nie
- Minimalizujemy ilość zmian i zależności



115

### Interface Segregation Principle - przykład #1



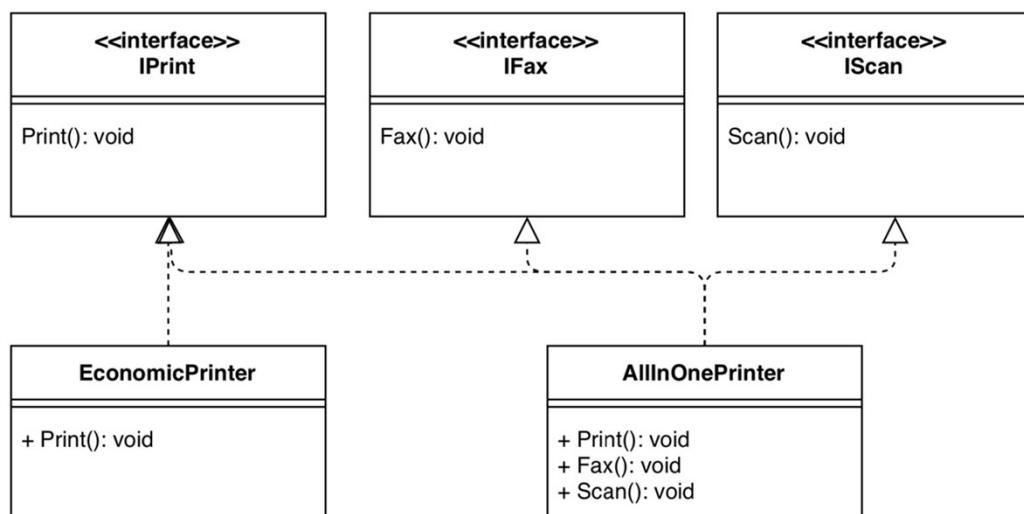
116

## Interface Segregation Principle - przykład #1

```
class EconomicPrinter : ISmartDevice
{
    public void Print()
    {
        //Yes I can print.
    }
    public void Fax()
    {
        throw new NotSupportedException();
    }
    public void Scan()
    {
        throw new NotSupportedException();
    }
}
```

117

## Interface Segregation Principle - przykład #2



118

## Reguły SOLID

# Dependency Inversion Principle

- Jednostki wyższego rzędu nie powinny zależeć od jednostek niższego rzędu
- Jednostki powinny zależeć od abstrakcji
- Abstrakcja nie powinna zależeć od szczegółów, a szczegóły nie powinny zależeć od abstrakcji => utrzymanie odpowiedniego poziomu abstrakcji



119

## Reguły SOLID

# Dependency Inversion Principle - przykład #1



**Pytanie:**

Czy poniższy kod spełnia DIP?

```
class Logger {
    private NtfsFileSystem _fileSystem = new NtfsFileSystem ();
    public void Log (string text) {
        var fileStream = _fileSystem.OpenFile ("log.txt");
        fileStream.Write (text);
        fileStream.Dispose ();
    }
}
```

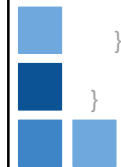


120

## Dependency Inversion Principle

```
public interface ILoggable {
    void Log(string textToLog);
}

class NtfsFileSystem : ILoggable {
    public void Log (string textToLog) {
        //file handling, writing and disposing.
    }
}
```



121

## Dependency Inversion Principle

```
class Logger {
    private ILoggable _logService;

    public Logger (ILoggable logService) {
        if (logService == null) throw new ArgumentNullException ();
        _logService = logService;
    }

    public void Log (string text) {
        _logService.Log (text);
    }
}
```



122

## Dependency Inversion Principle

```
class Program () {
    public static void main(String[] vars) {
        var ntfsLogger = new Logger(new NtfsFileSystem ());
        var noSqlLogger = new Logger(new DbNoSql ());
        ntfsLogger.Log("some text");
        noSqlLogger.Log("other text");
    }
}
```



123

## Reguły SOLID

### Dependency Inversion Principle

- Staraj się utrzymać zależności tylko od abstrakcyjnych jednostek
- Jednostki wyższego rzędu nie powinny się zmieniać w wyniku zmian w jednostkach niższego rzędu (np. wskutek zmian technologicznych)
- Ta ostatnia reguła SOLID prowadzi bezpośrednio do niskiego sprzężenia



124

# Podsumowanie



125

## Potencjalne zalety podejścia obiektowego

- Bezpośrednie odwzorowanie dziedziny (problemu) na system (rozwiązanie) → bardziej ambitne dziedziny zastosowań.
- Metody organizacji wzorowane na myśleniu człowieka → lepsze zrozumienie użytkownika i eksperta.
- Traktowanie danych (atrybutów) i procesów (usług) jako naturalnej całości → zwiększenie spójności modelu.
- Jawna reprezentacja wspólnych cech (dziedziczenie) → uproszczenie modelu.
- Ukrywanie cech nietrwałych (hermetyzacja) → łatwość modyfikacji i rozszerzania systemu.
- Modele odpowiadające rzeczywistości, takie same metody organizacji → wielokrotne wykorzystanie wyników.
- Ciągłość reprezentacji w kolejnych etapach budowy systemu → łatwy powrót do wcześniejszych etapów



126