Zasady projektowania obiektowego

SOLID i GRASP

Wprowadzenie

Nie każdy, kto ma młotek,
może nazywać się architektem.

- wzorce projektowe
- UML
- SOLID
  - Robert C. Martin
    http://www.objectmentor.com/resources/publishedArticles.html
  - Strategia w metodyce Agile
- GRASP – Responsibility Driven-Design

S. O. L. I. D.

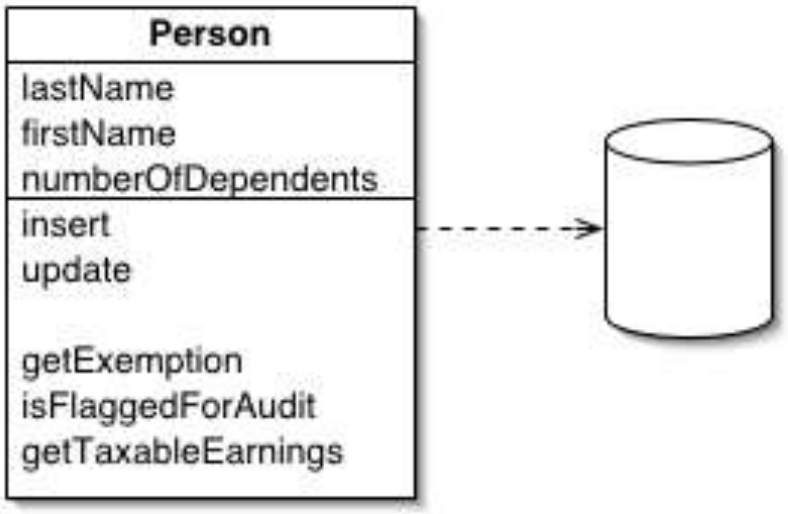
S O L I D

Outline

- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

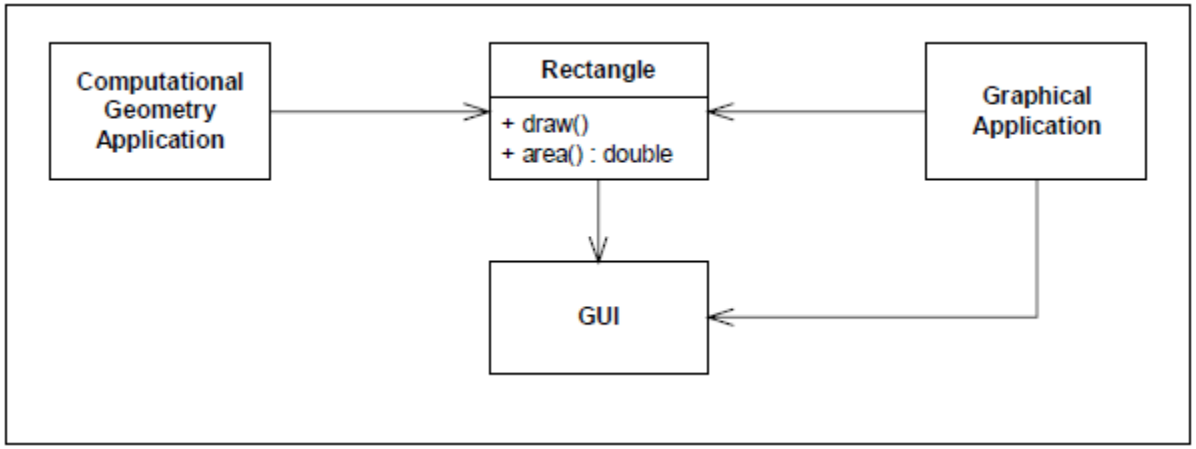
Przykład – wzorzec Active Record

Obiekt, który osłania wiersz tabeli lub widoku bazy danych,
hermetyzuje operacje dostępu
i realizuje elementy logiki dziedziny.

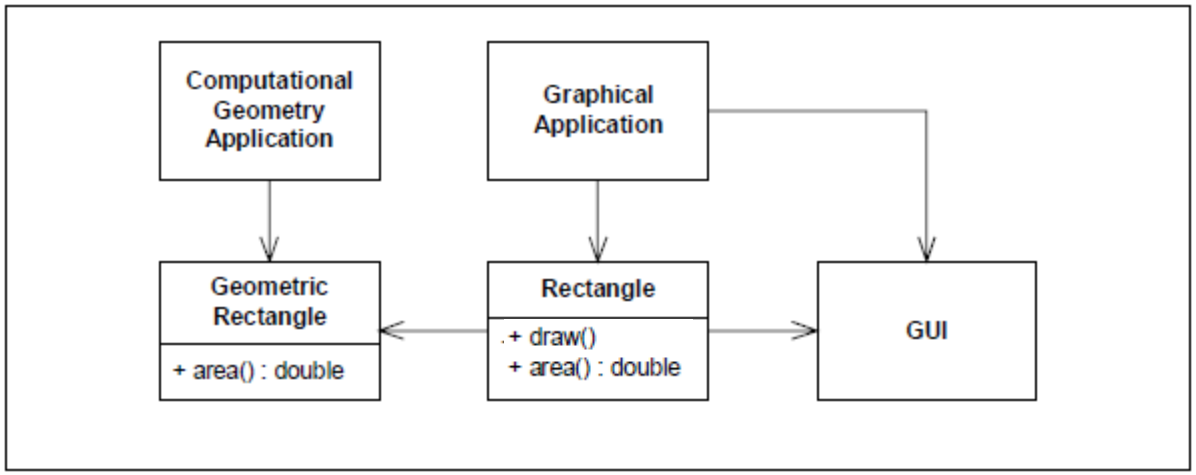


Signle Responsibility Principle

SRP – przykład Rectangle

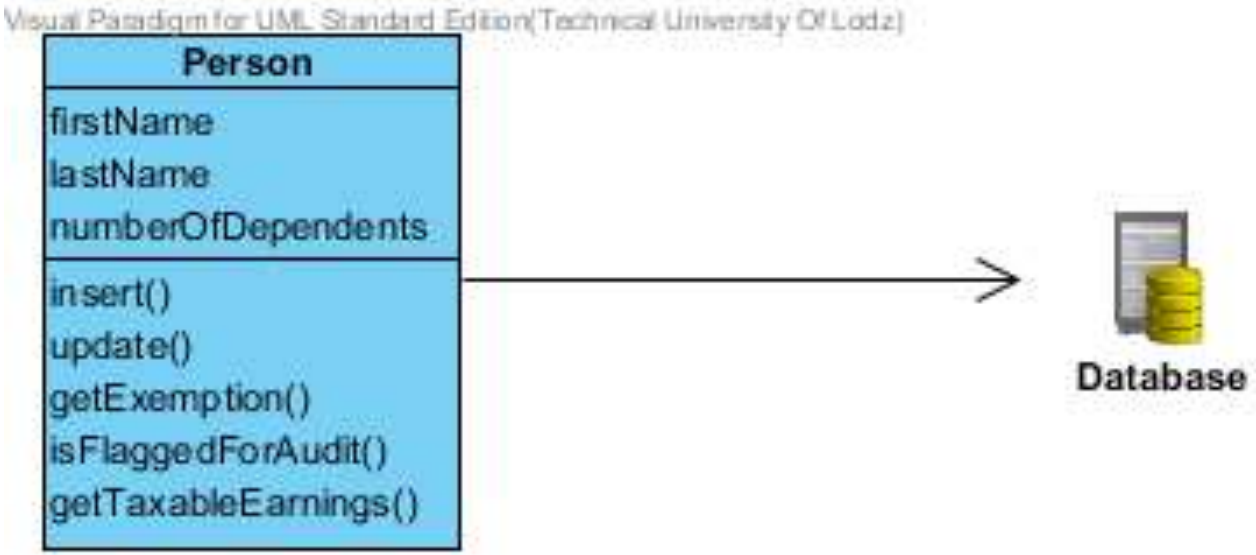


SRP



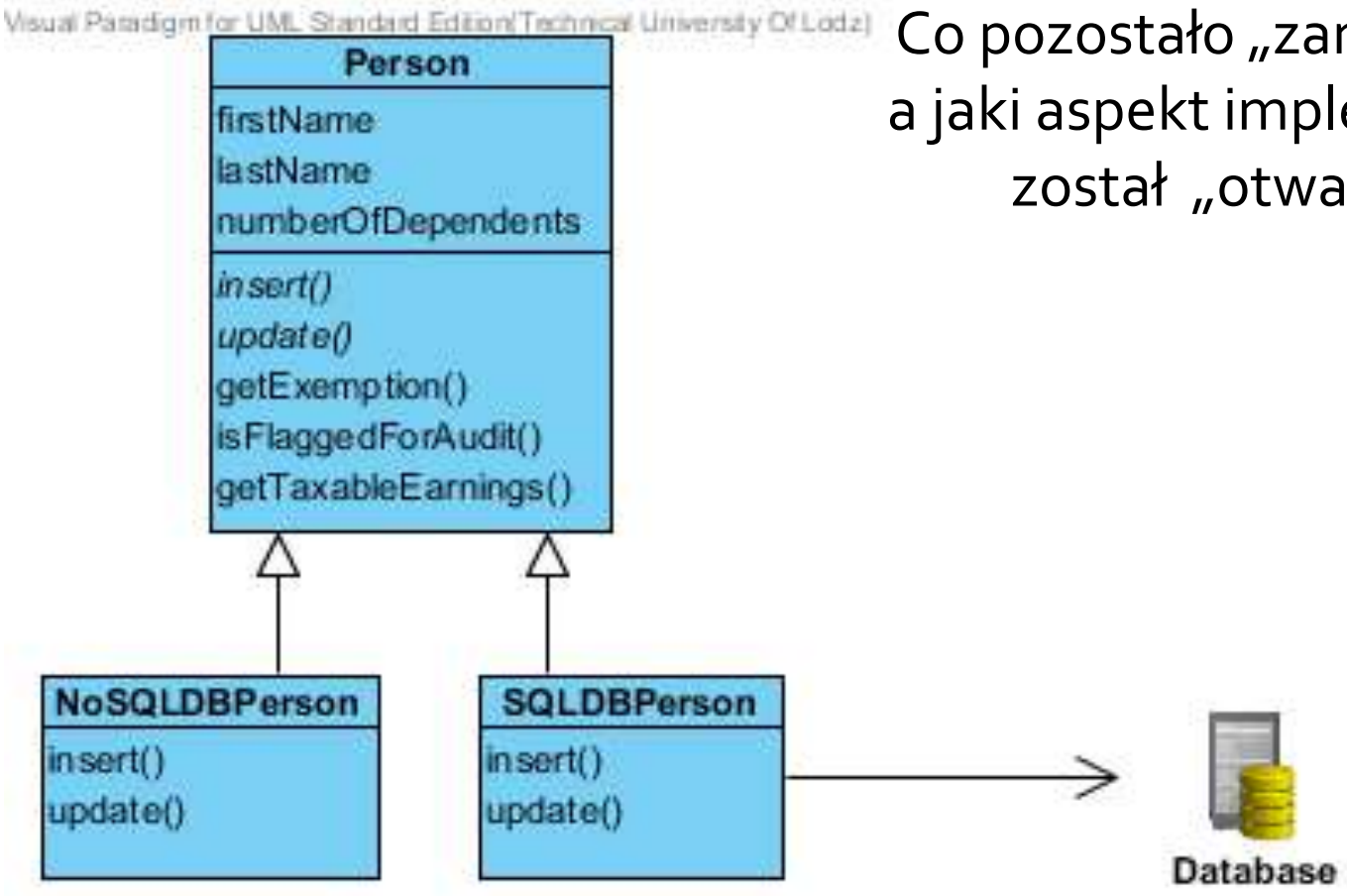
ActiveRecord a SRP

Ile i jakie odpowiedzialności spoczywają na obiekcie *Person*?
Czy ten wzorzec spełnia zasadę SRP?



Open Closed Principle

OCP przez dziedziczenie



Co pozostało „zamknięte”, a jaki aspekt implementacji został „otwarty”?

OCP a Agile

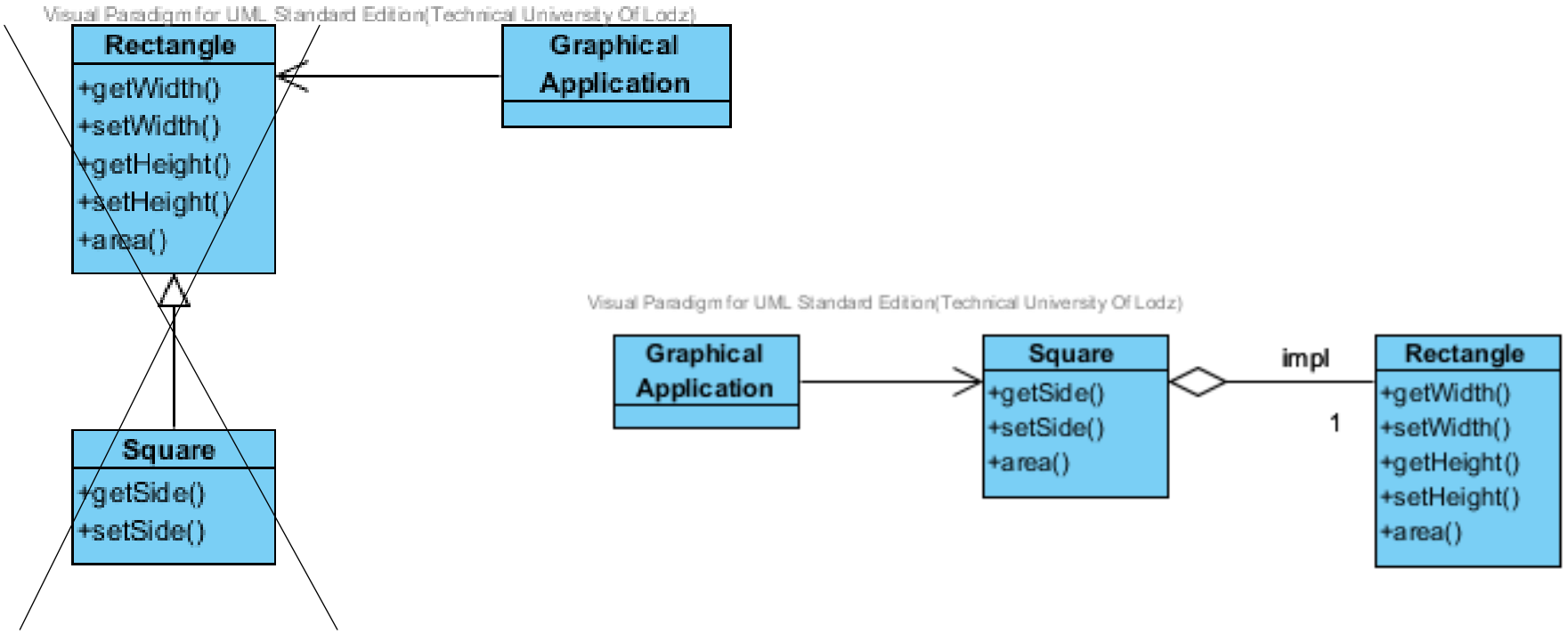
- „Otwarcie” na wszystkie rodzaje zmian nie jest możliwe.
- Decyzja programisty w tym zakresie może okazać się niewłaściwa.

- Metodyka Agile może pomóc ujawnić istotne pola dla zmian na wczesnym etapie.
- Można wtedy dokonać „otwarcia” pod kątem *tych zmian* (refaktoryzacja).

Liskov Substitution Principle

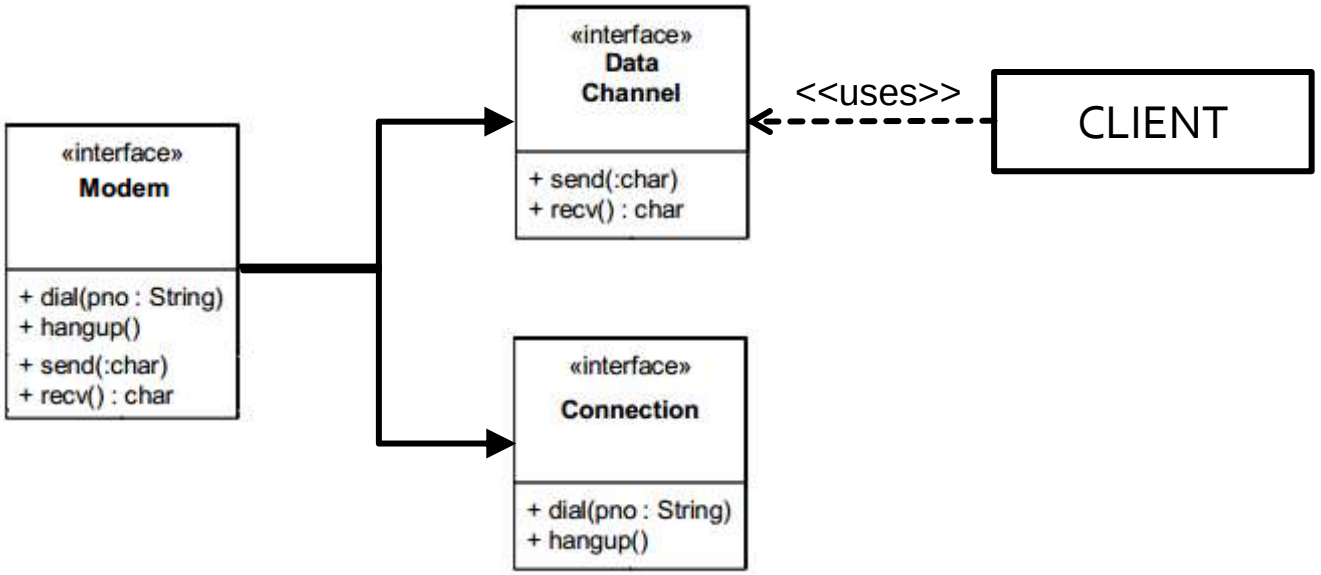
LSP – podklasy i „kontrakt”

Bazą dobrego projektu obiektowego jest jasno zdefiniowany i spójny *kontrakt* między współpracującymi obiektami.



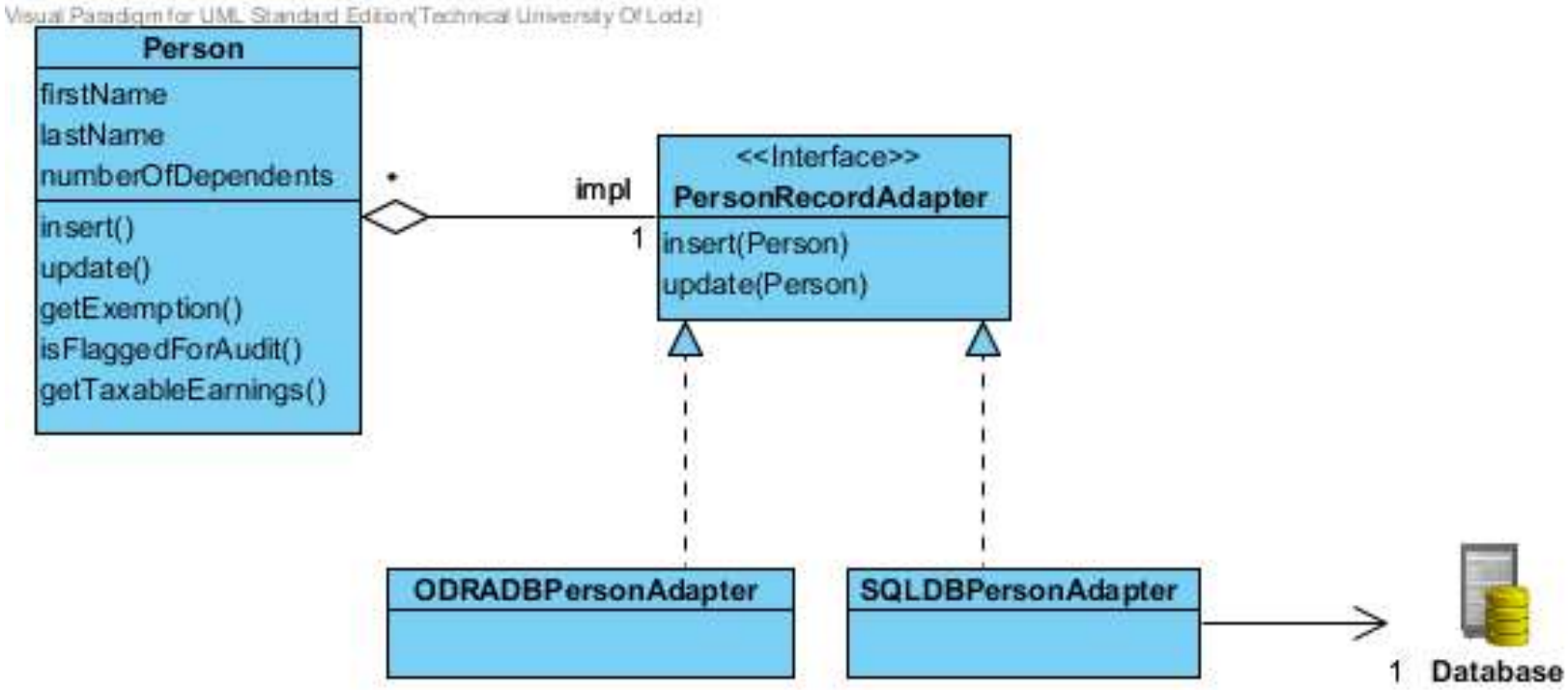
Złamanie zasady LSP

- Jeżeli dziedzicząc sprawiamy, że istniejący kod będzie musiał sprawdzać z jakim typem ma do czynienia.

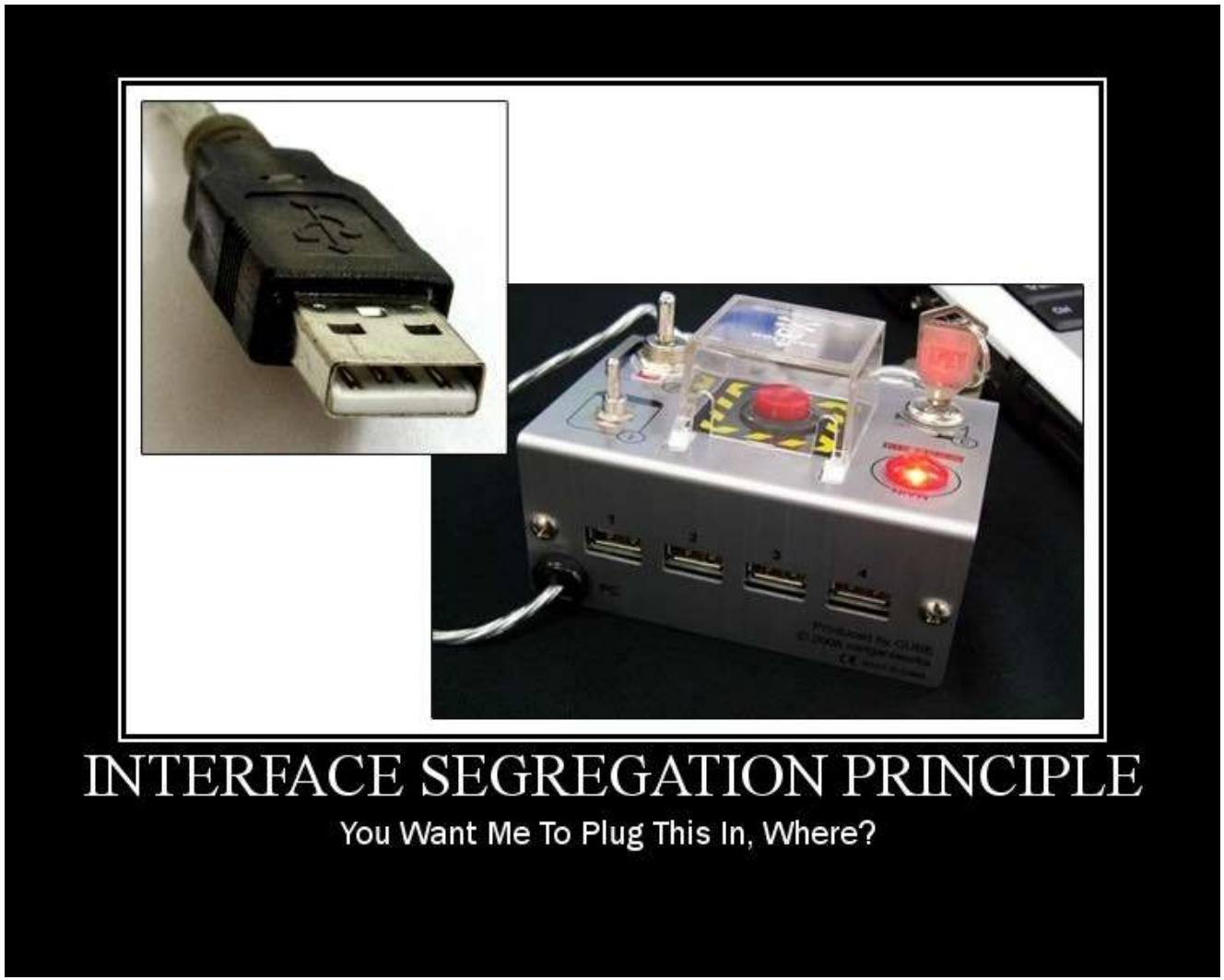


LSP - wzorzec Strategy

Interfejs *PersonRecordAdapter* określa „kontrakt” czyli odpowiedzialność klas go implementujących (i ich podklas).



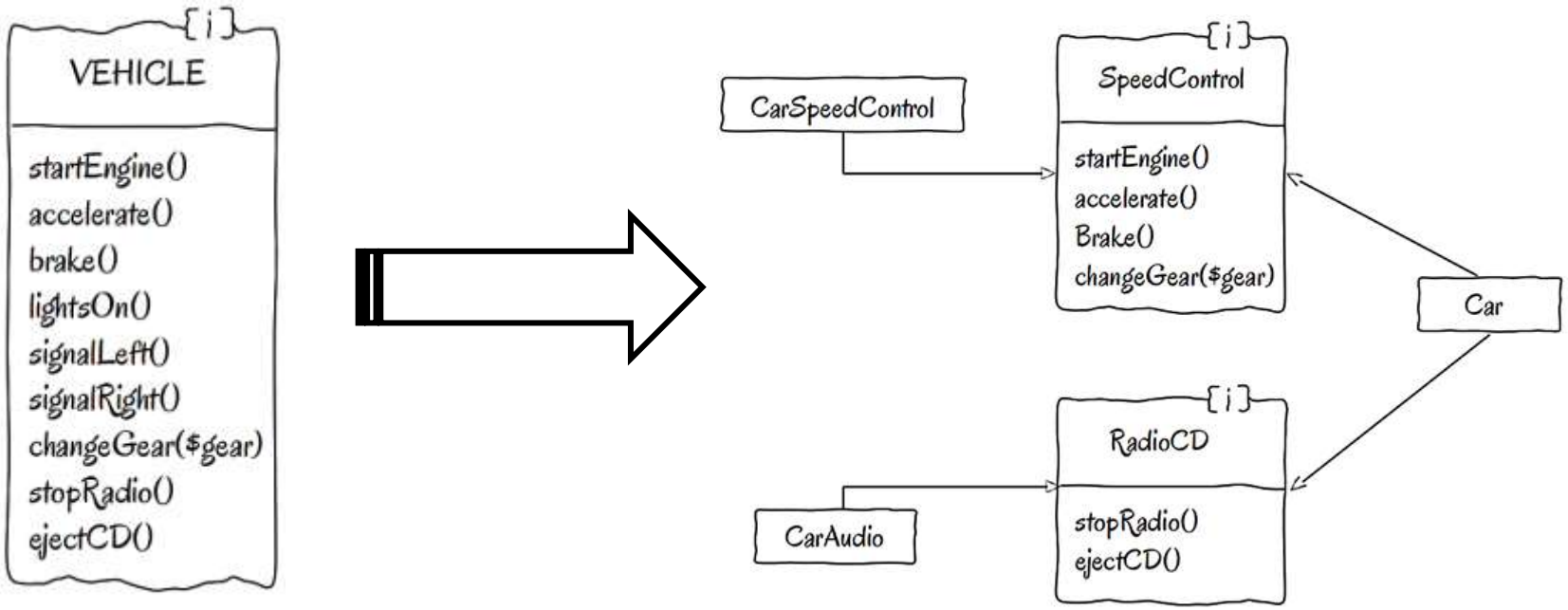
Interface Segregation Principle



Interface Segregation Principle

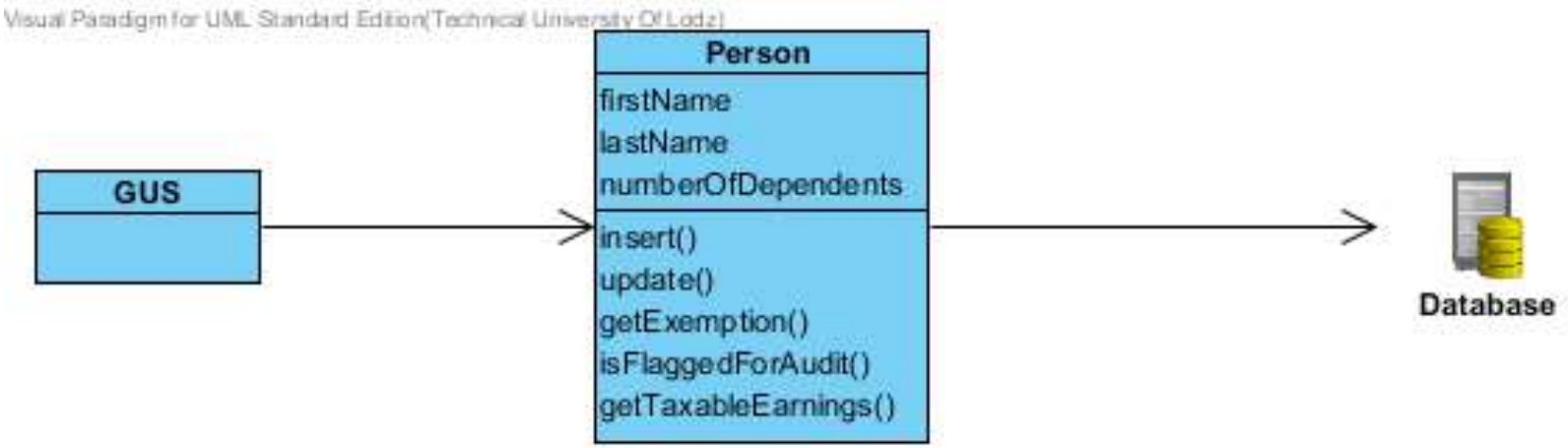
- Klient nie powinien być uzależniony od interfejsów, z których nie korzysta.
- SRP dla interfejsów

https://code.tutsplus.com/tutorials/solid-part-3-liskov-substitution-interface-segregation-principles--net-36710



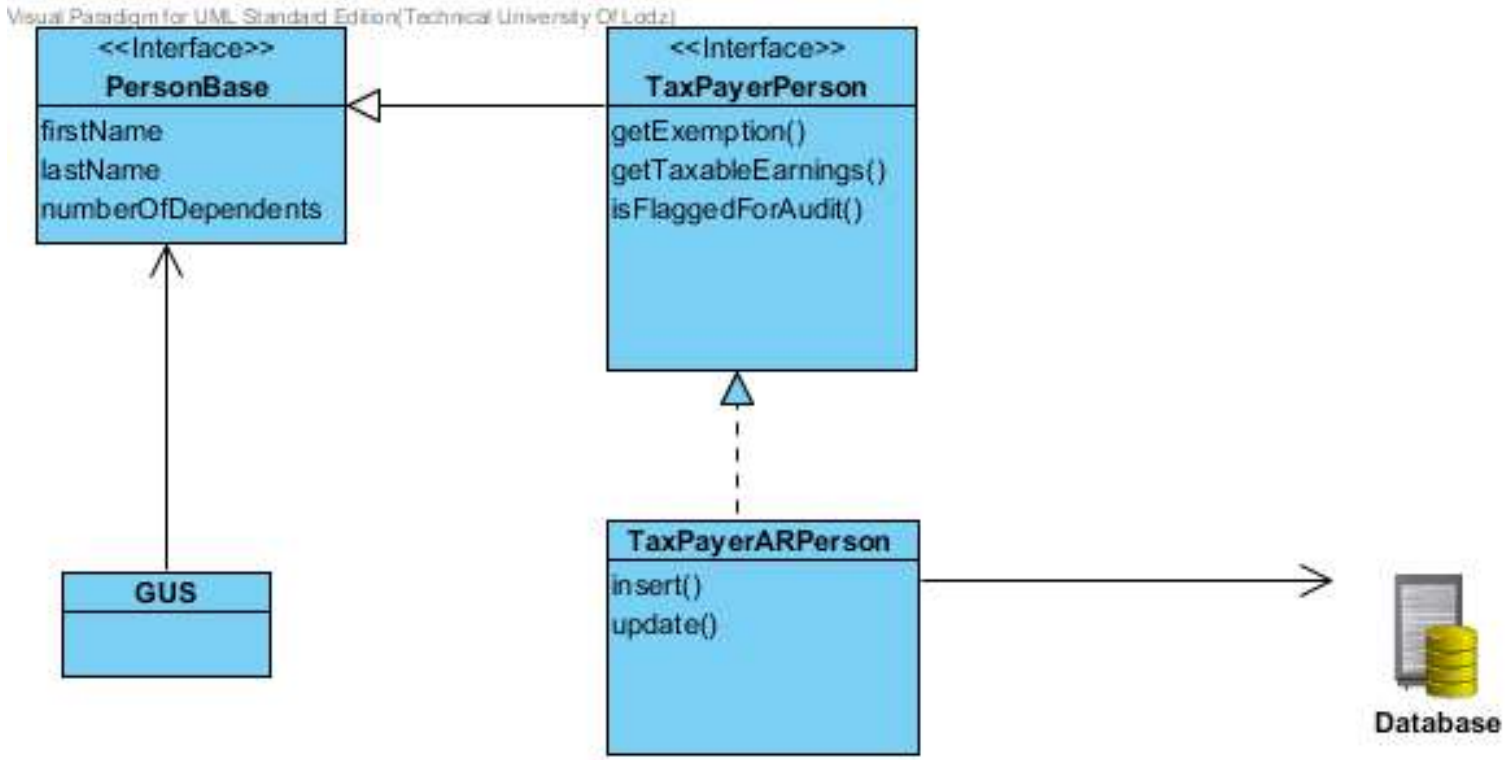
ISP a wzorzec ActiveRecord

Interfejs klasy *Person* łączy w sobie kilka odpowiedzialności (dane, logika „podatkowa”, operacje na bazie danych) obarczając potencjalnych klientów każdą z nich.

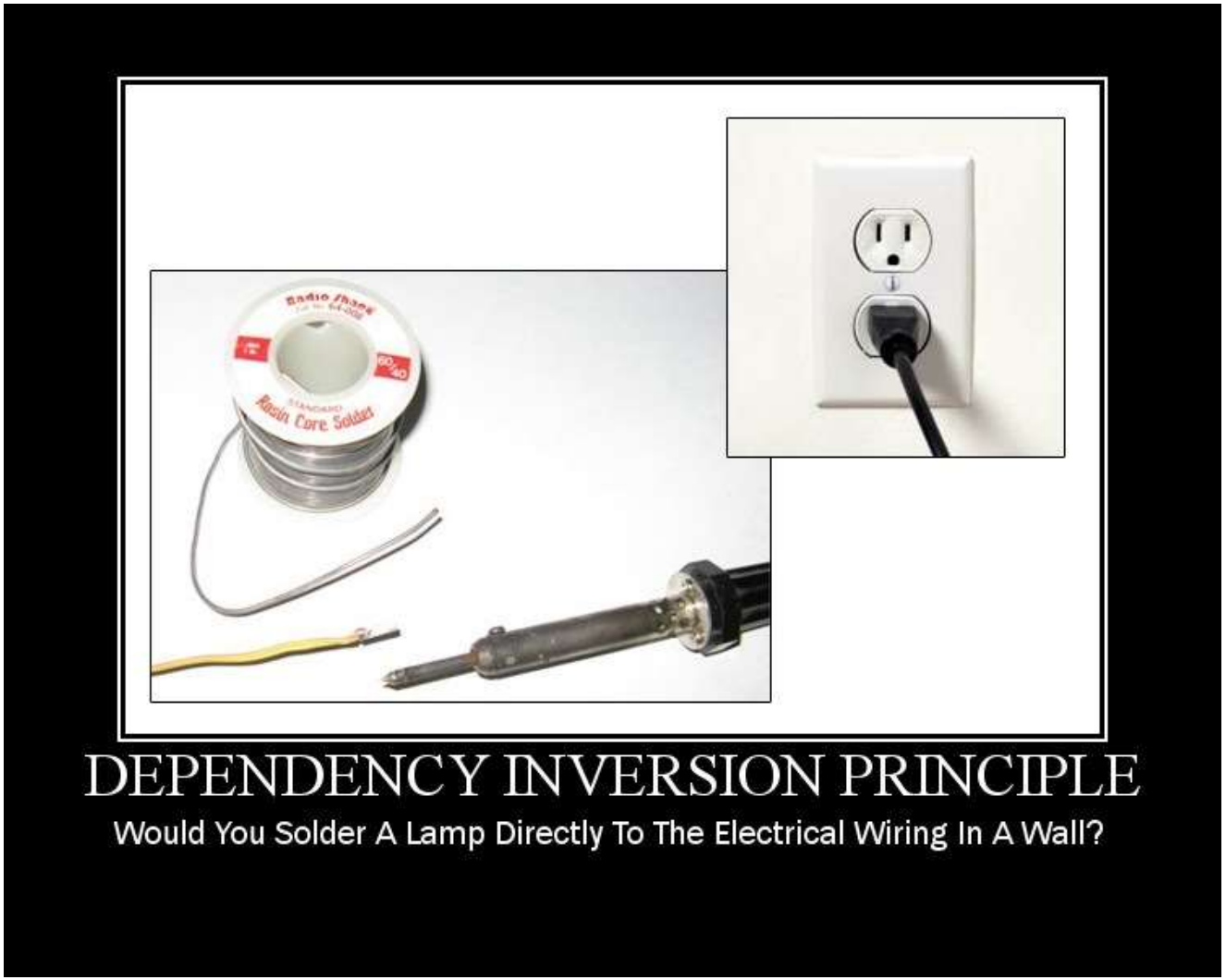


ISP uporządkowanie interfejsów

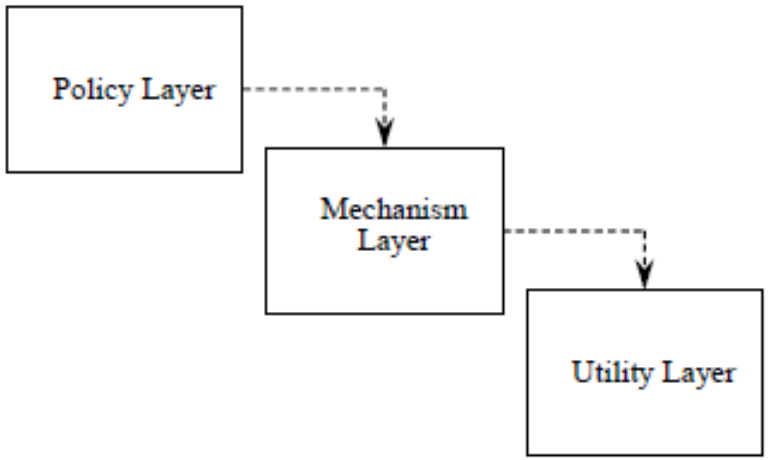
GUS zależny wyłącznie od zmian w interfejsie *PersonBase*.

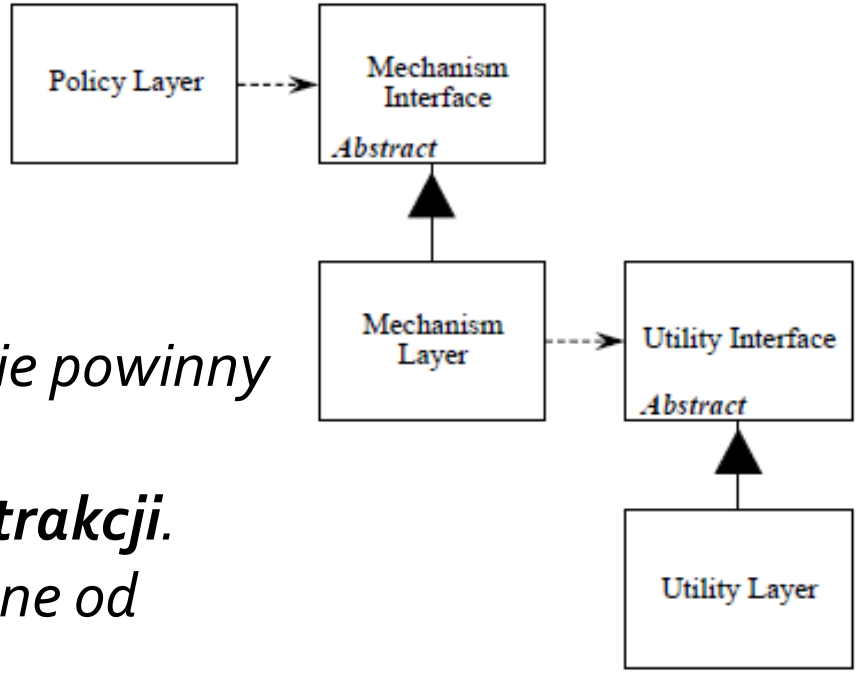


Dependency Inversion Principle



DIP – architektura warstwowa





Komponenty wyższych warstw nie powinny zależeć od niższych.
Obydwa powinny zależeć od ***abstrakcji****.*
Szczegóły powinny być uzależnione od ***abstrakcji****. Nie odwrotnie!*

Robert C. Martin

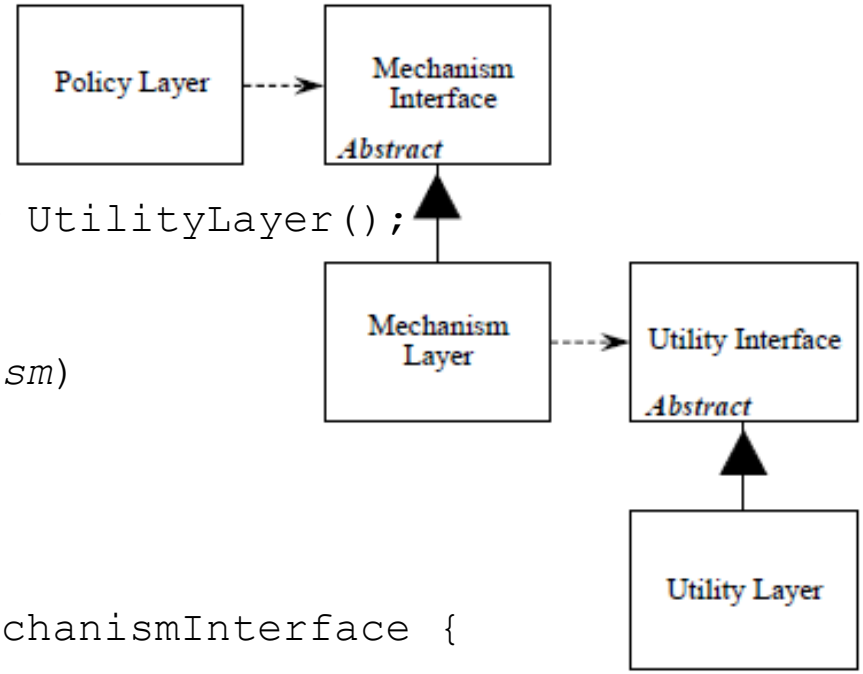
DIP – ręczne wstrzykiwanie zależności przez konstruktory

```
class DependencyManager {
  …
  static Policy getPolicy() {
    UtilityInterface utility = new UtilityLayer();
    MechanismInterface mechanism =
      new MechanismLayer(utility);
    return new PolicyLayer(mechanism)
  }
  …
}

class MechanismLayer implements MechanismInterface {

  UtilityInterface utility;

  MechanismLayer(UtilityInterface utility) {
    this.utility = utility;
  }
  …
}
```

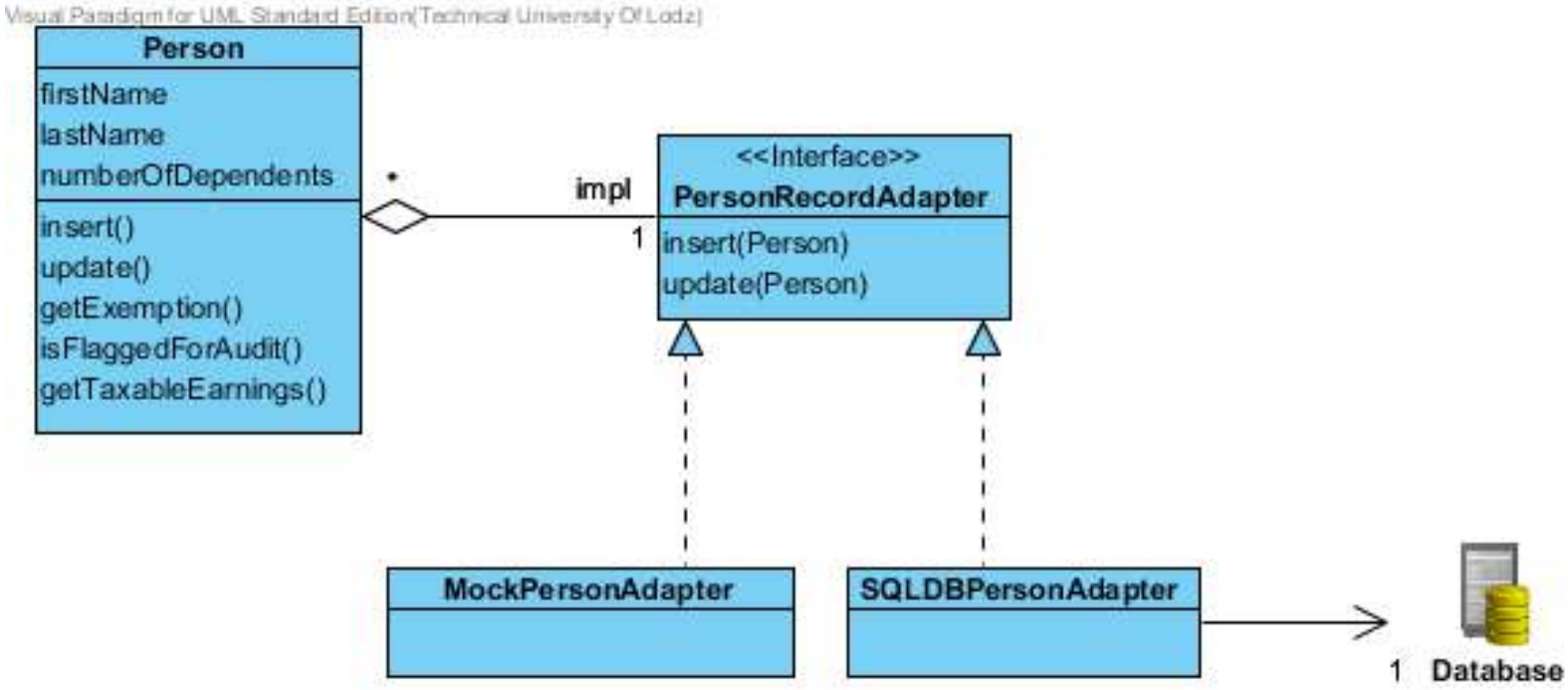


Dependency Injection

- Podstawowe metody realizacji Dependency Injection wykorzystują:
  - settery i konstruktory.
- Ich wywołanie może być automatyczne na podstawie określonej konfiguracji:
  - kontenery IoC frameworka Spring,
  - kontenery Pico.
- Istnieje też metoda wstrzykiwania zależności *interface injection* (pół automatyczna).

DI – Testowanie ActiveRecord

Jak przetestować działanie metod *insert* i *update* oraz metod od nich zależnych w przypadku braku bazy danych?



GRASP

General Responsibility Assignment Software Patterns

Craig Larman - UML i wzorce projektowe. Analiza i projektowanie obiektowe oraz iteracyjny model wytwarzania aplikacji, Helion 2011

Odpowiedzialność

- Kontrakt lub zobowiązanie klasyfikatora
- Zobowiązania związane z **działaniem**, to np.:
  - wykonywanie pewnej czynności przez obiekt, jak tworzenie innego obiektu lub przeprowadzanie obliczeń
  - inicjalizacja czynności wykonywanych przez inny obiekt (zmuszenie innego obiektu do wykonania czynności)
  - kontrola i koordynacja czynności wykonywanych przez inne obiekty
- Zobowiązania związane z **wiedzą**, to np.:
  - znajomość prywatnych, ukrytych danych,
  - wiedza o obiektach powiązanych,
  - wiedza nt. możliwości operacyjnych obiektu.

Outline

- **Wzorce ewaluacyjne:**
  - **Low Coupling**
  - **High Cohesion**
- Information Expert
- Creator
- Pure Fabrication
- Controller
- Protected Variations
- Polymorphism
- Indirection

Low Coupling (Niskie Sprzężenie)

- Jak zmniejszyć liczbę zależności i zasięg zmian, a zwiększyć możliwość ponownego wykorzystania kodu?
  - Przydziel odpowiedzialność tak, by zlikwidować niepotrzebne sprzężenia (**powiązania**).
  - Skorzystaj z tej zasady podczas oceniania alternatywnych rozwiązań.

High Cohesion (Wysoka spójność)

- Jak sprawić by obiekty miały jasny cel, były zrozumiałe i łatwe w utrzymaniu, a przy okazji zmniejszyć sprzężenie?
  - Przydziel odpowiedzialność tak, by zachować wysoką spójność obiektów.
  - Skorzystaj z tej zasady podczas oceniania alternatywnych rozwiązań.
- Powiązane z:
  - **Low Coupling**
  - **S** o **L** **I** d

Information Expert (Ekspert)

- Na jakiej zasadzie przydzielać obiektom zobowiązania?
  - Klasie, która posiada informacje niezbędne do realizacji zobowiązania.
- Powiązane z:
  - **Low Coupling**
  - **High Cohesion**

Creator (Twórca)

- Kto powinien być odpowiedzialny za utworzenie nowej instancji pewnej klasy A?
  - Klasa B, jeśli spełniony jest, któryś z warunków:
    - **B „zawiera" A lub agreguje A (kompozycja),**
    - B zapamiętuje A,
    - B bezpośrednio używa A,
    - B posiada dane inicjalizacyjne dla A.
- Powiązane z:
  - **Low Coupling**
  - **Factory Method** i **Abstract Factory**

Pure Fabrication (Czysty wymysł)

- Jaki obiekt powinien otrzymać zobowiązanie, jeśli nie chcemy pogwałcić zasad **High Cohesion** i **Low Coupling** (i innych), ale nie odpowiadają nam rozwiązania oferowane np. przez **Expert**?
  - Przydziel spójny zestaw zobowiązań sztucznej lub pomocniczej klasie, która nie reprezentuje konceptu z dziedziny problemu.
- Wiele wzorców to przykłady Pure Fabrication

Controller (Kontroler)

- Który z obiektów poza warstwą UI odbiera żądania operacji systemowych i kontroluje (koordynuje) jej wykonanie?
  - Kontroler fasadowy – obiekt reprezentujący cały system lub ważny podsystem.
  - Kontroler sesyjny – klasa reprezentująca scenariusz przypadku użycia.
- Powiązane z:
  - **Command, Facade**
  - **Pure Fabrication**

Protected Variations (Ochrona zmienności)

- Jak przydzielać zobowiązania obiektom, podsystemom i systemom, by ich zmienność lub niestabilność nie wywierała negatywnego wpływu na pozostałe elementy?
  - Zidentyfikuj punkty przewidywalnej zmienności lub niestabilności (tzw. *hot spots*). Przydziel zobowiązania tak, by wokół tych punktów powstały stabilne interfejsy.

- Powiązane z:
  - **Low Coupling**
  - Większość wzorców **GoF**
  - s **O** **L** i d

Polymorphism (Polimorfizm)

- Jak postąpić, gdy zachowanie różni się w zależności od typu?
  - Jeśli powiązane ze sobą możliwości lub zachowania różnią się w zależności od typu (klasy), przydziel zobowiązanie związane z danym zachowaniem – za pomocą operacji polimorficznych – tym typom, które różnią się zachowaniem.
- Powiązane z:
  - **Protected Variations**
  - Wiele wzorców **GoF**

Indirection (Pośrednictwo)

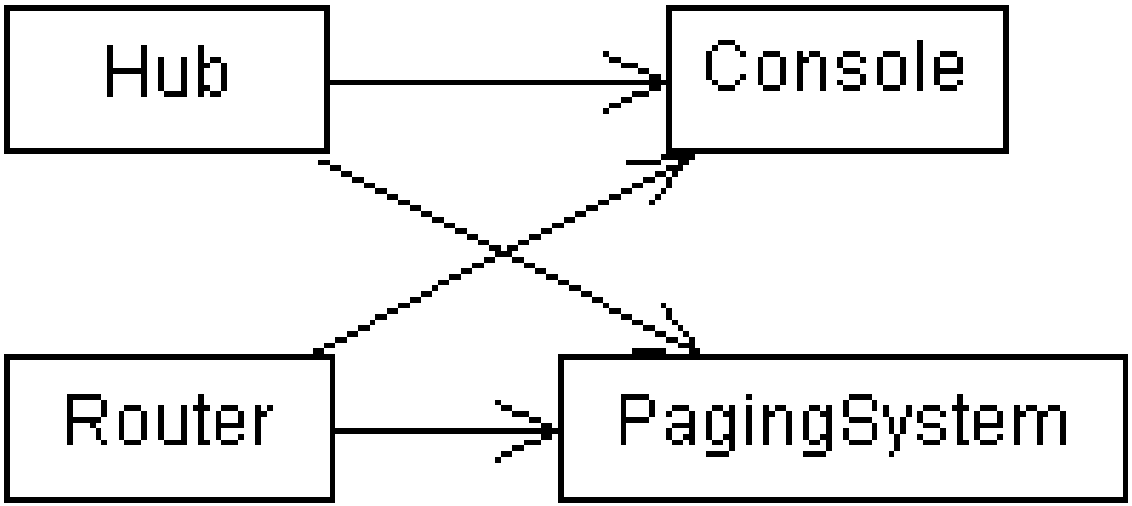
- Jak przydzielić zobowiązania, by uniknąć bezpośredniego sprzężenia?
  - Przydziel zobowiązanie obiektowi pośredniczącemu, który będzie przekazywał komunikaty pomiędzy innymi komponentami lub usługami.
- Powiązane z:
  - **Protected Variations** i **Low Coupling**
  - Wiele wzorców **GoF**
  - s o l i **D**

Event Notifier, a Pattern for Event Notification

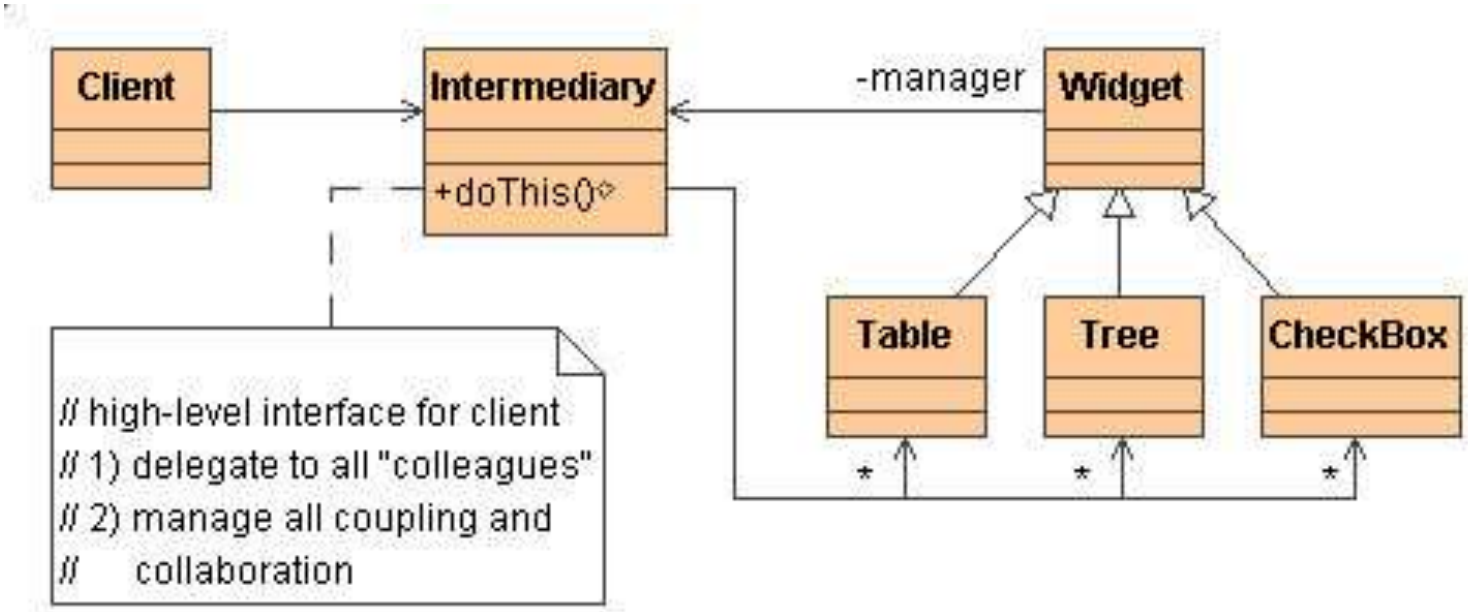
Zarządzanie zależnościami

http://www.marco.panizza.name/dispenseTM/slides/exerc/eventNotifier/eventNotifier.html

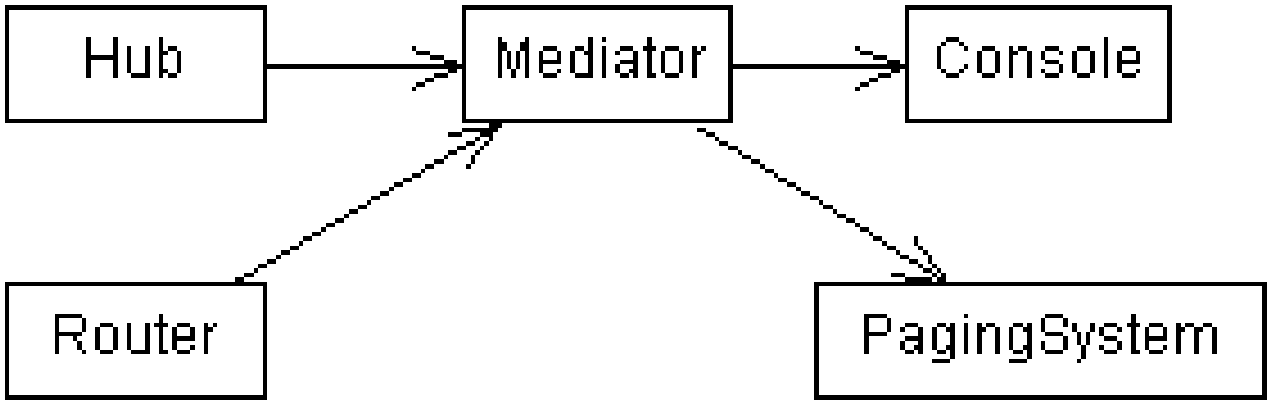
Simplistic Approach



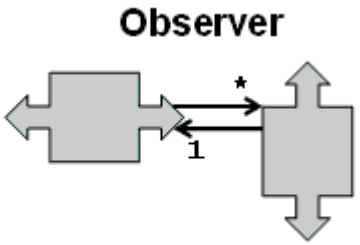
Mediator – diagram klas

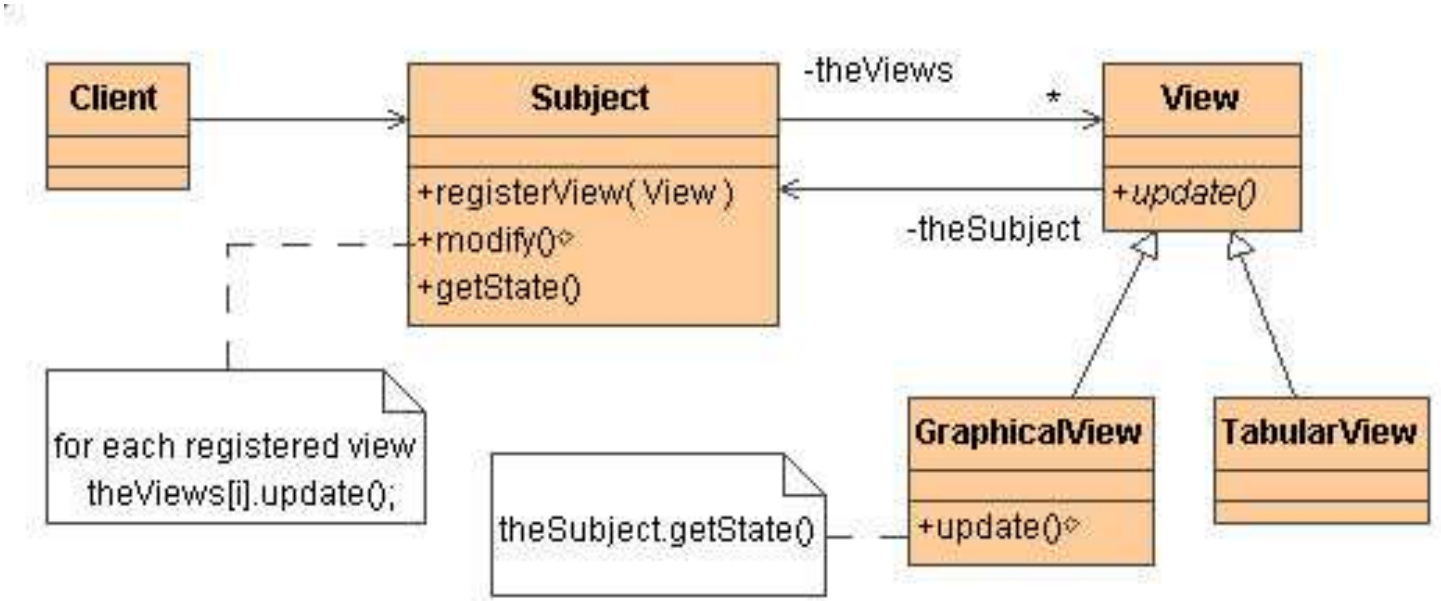


Mediator Approach

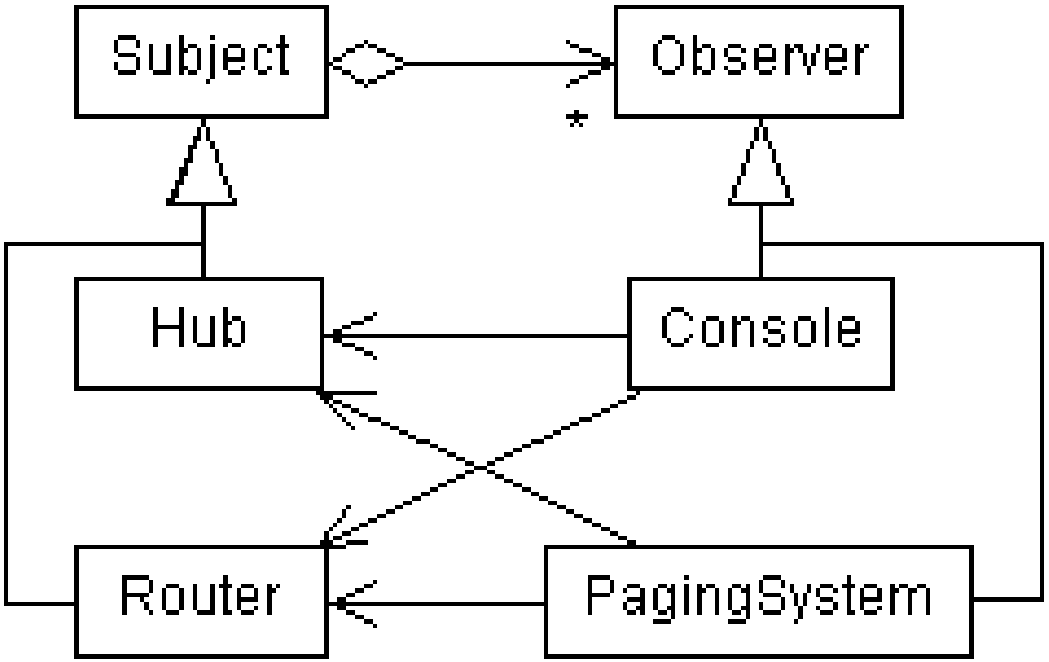


Observer – diagram klas

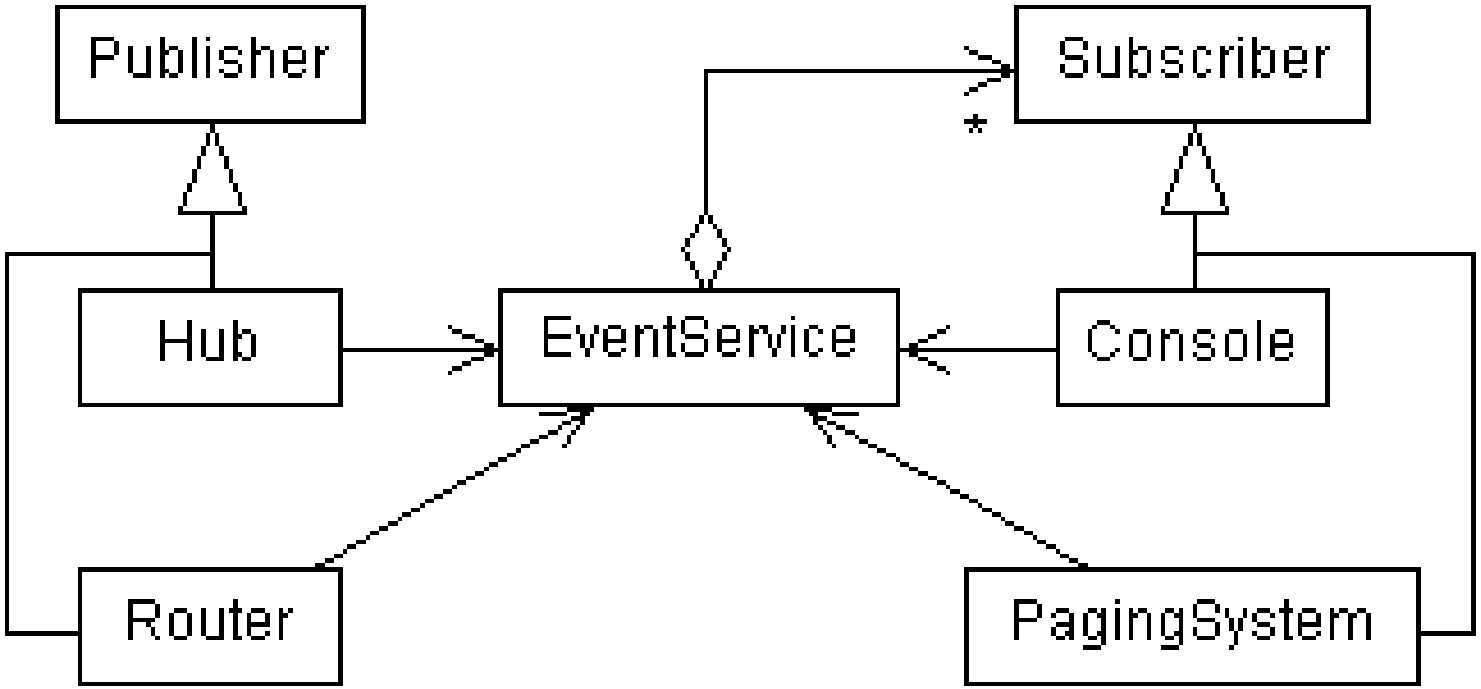




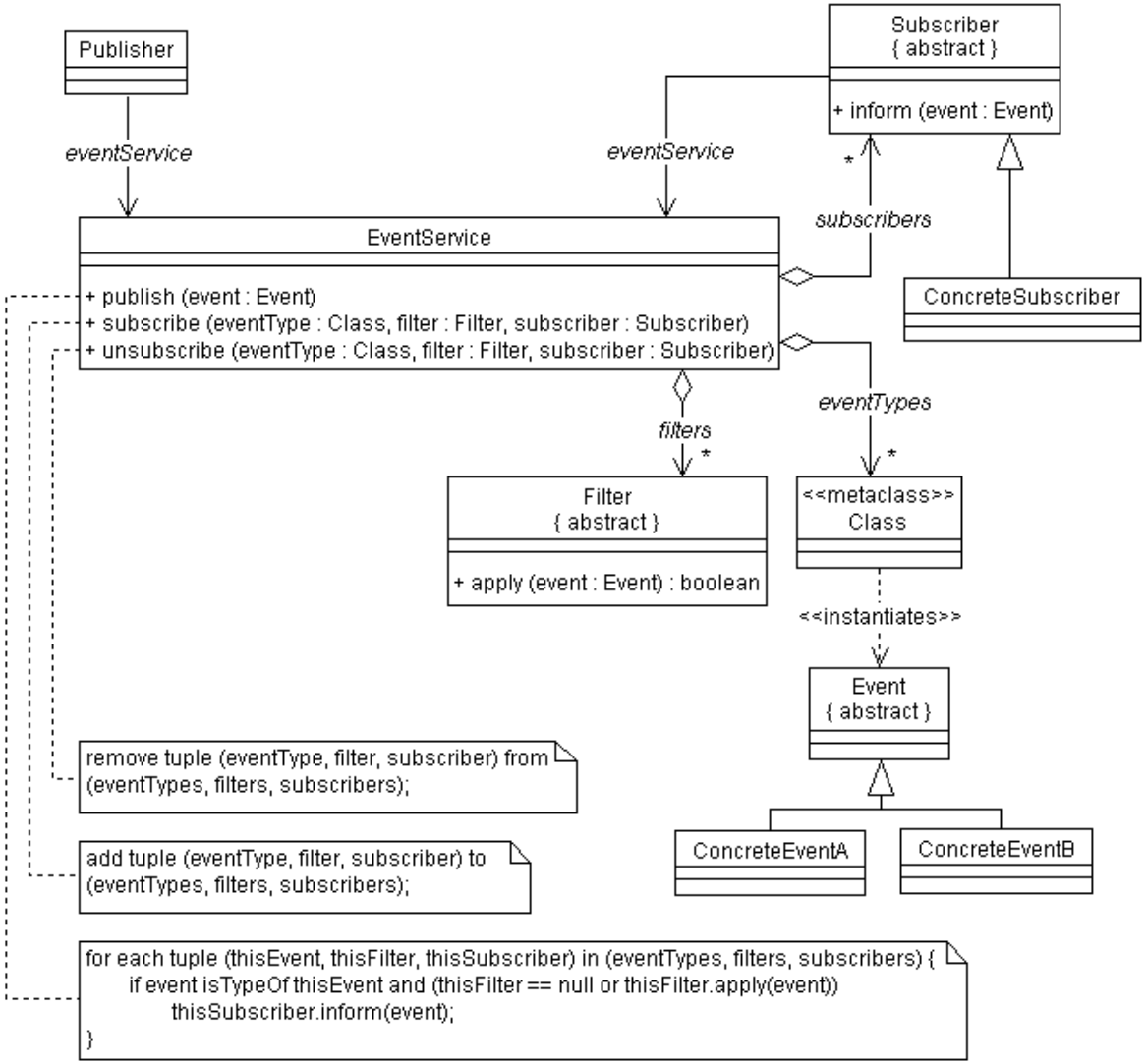
Observer Approach



Event Notifier



Event Notifier - structure



Event Notifier - Zastosowanie

- When an object should be able to notify other objects of an event without needing to know who these objects are or what they do in response.
- When an object needs to be notified of an event, but does not need to know where the event originated.
- When more than one object can generate the same kind of event.
- When some objects are interested in a broader classification of events, while others are interested in a narrower classification.
- When an object may be interested in more than one kind of event.
- When you need to dynamically introduce new kinds of events.
- When objects participating in notification may be dynamically introduced or removed, as in distributed systems.
- When you need to filter out events based on arbitrary criteria.