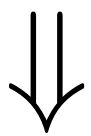
Technologia Programowania 2017/2018

Wykład 3
GRASP

Jakub Lemiesz

Co już wiemy?

1. General Responsibility Assignment Software Patterns
   - ▷ 9 ogólnych wzorców przydziału odpowiedzialności
   - ▷ cel: projektować w metodyczny sposób (∼ *nauka*)
   - ▷ pozwalają lepiej zrozumieć wzorce projektowe GoF



2. 23 wzorce projektowe Gang of Four : Gamma, Helm, Johnson, Vlissides (*rozwiązania częstych problemów projektowych ∼ wzorce architektoniczne w budownictwie*)

Zasady GRASP

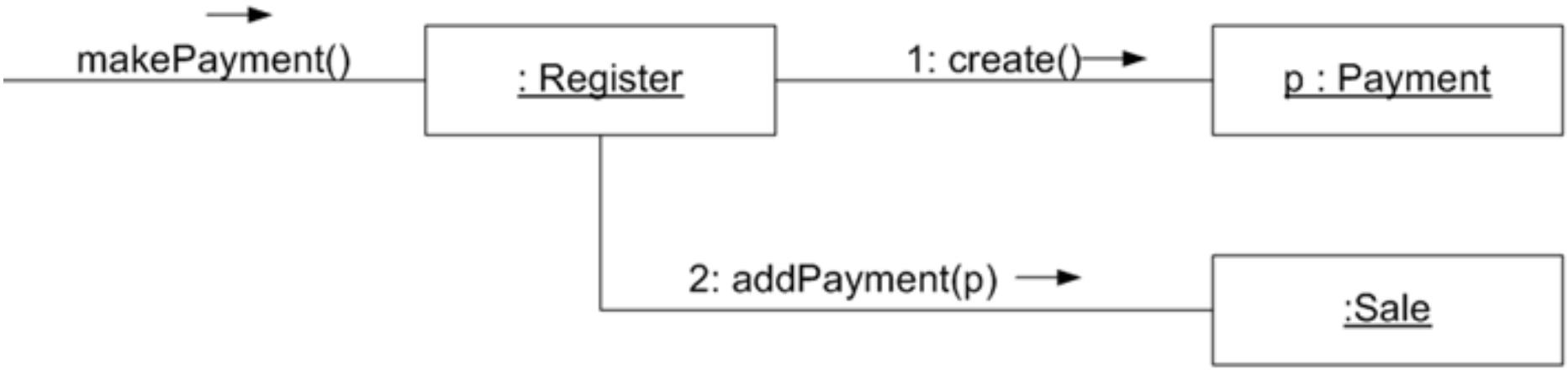
- ✓ Expert (*zasada podstawowa*)
- ✓ Creator (*zasada podstawowa*)
- ⇒ Low Coupling (*zasada ewaluacyjna*)
- ⇒ High Cohesion (*zasada ewaluacyjna*)
- ⇒ Polymorphism
- ⇒ Indirection
- ⇒ Pure Fabrication
- ⇒ Protected Variations
- ⇒ Controller

Sprzężenie (*ang. Coupling*)

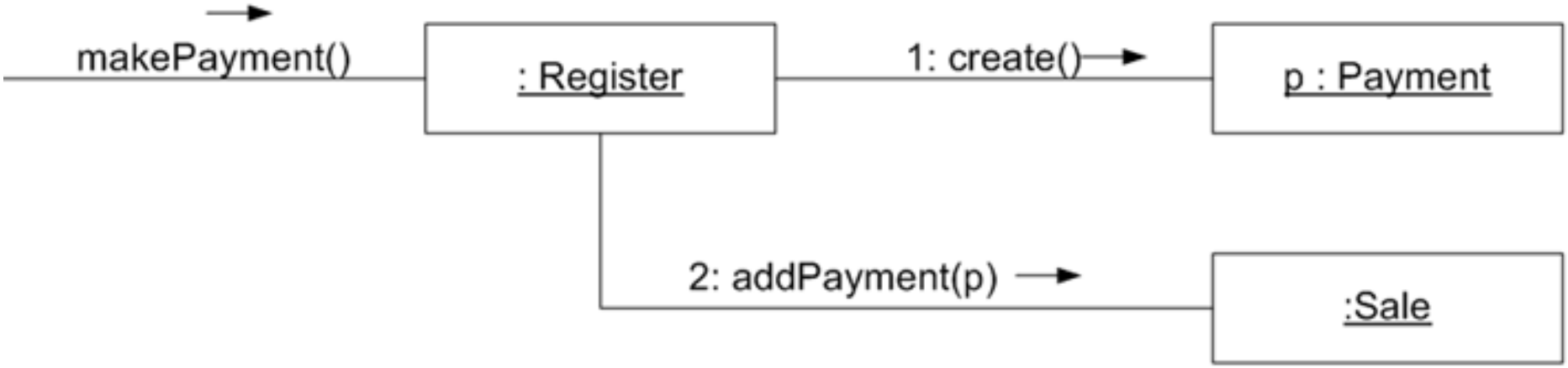
1. Zależności między klasami: atrybuty, argumenty metod, wywołania metod, dziedziczenie,...
2. Sprzężenie - miara określająca w jakim stopniu dana klasa jest zależna od innych klas, bibliotek
3. Problemy z wysokim sprzężeniem klasy A:
   - ▷ zmiany w innych klasach wymuszają zmiany A
   - ▷ trudno jest „zrozumieć” klasę A w izolacji
   - ▷ trudno jest powtórnie użyć klasę A, bo potrzebne są też obiekty z nią powiązane

Niskie sprzężenie (ang. Low Coupling)

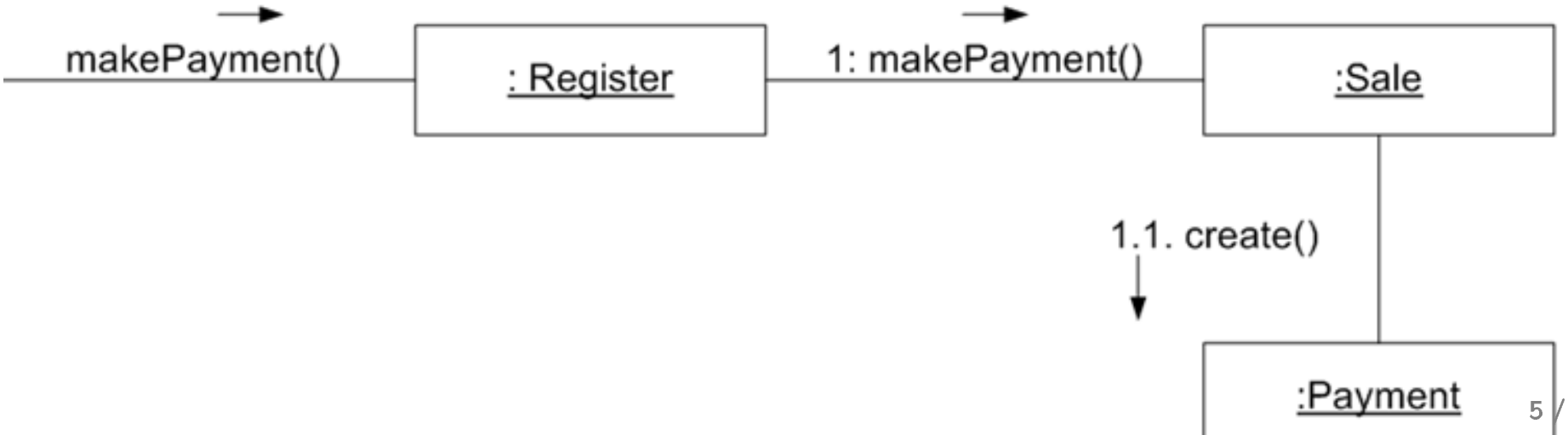
Jak lepiej powiązać instancje klas `Payment` i `Sale`?



Niskie sprzężenie (ang. Low Coupling)



Niskie sprzężenie



Low Coupling – zasada ewaluacyjna

1. Oceniając różne możliwości przydzielaj metody tak by ograniczać sprzężenie między klasami
2. Zysk: łatwiejsze ponowne wykorzystanie kodu, ograniczenie zasięgu zmian
3. Uwagi:
   - sprzęganie ze stabilnymi i łatwo dostępnymi elementami nie jest problemem (*np. java.util*)
   - pewien stopień powiązania jest nieunikniony, bo obiekty muszą się komunikować (*chyba, że jest jeden obiekt który robi wszystko...*)

Spójność (ang. Cohesion)

Spójność - miara określająca w jakim stopniu metody klasy są do siebie podobne i ze sobą powiązane (*tzn. podobna funkcjonalność i stopień ogólności*)

Niska spójność powoduje, że:

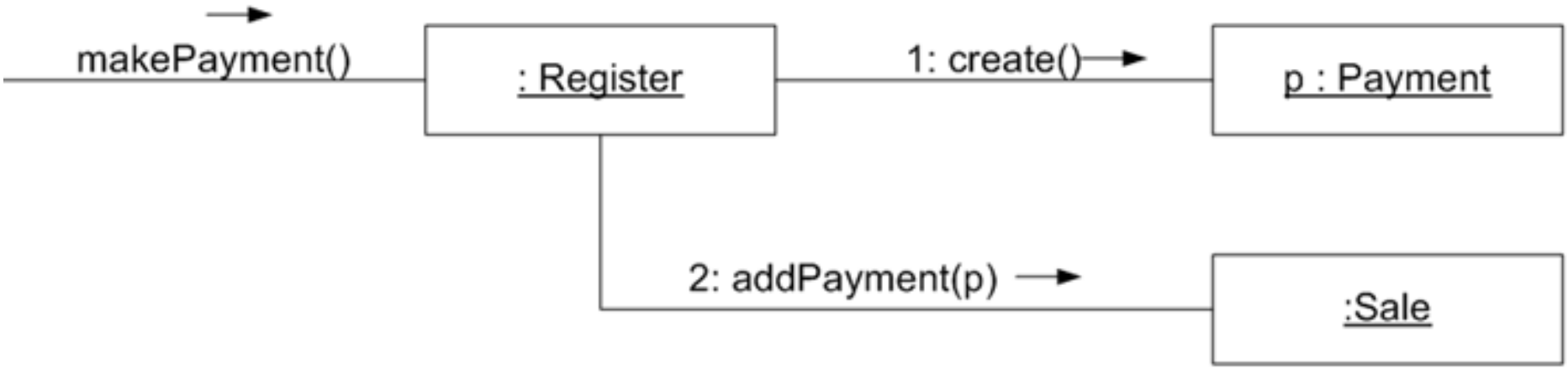
1. trudno zrozumieć kod i cel istnienia klasy
2. trudno klasę utrzymywać, rozwijać i ponownie wykorzystać

Jak ocenić spójność?

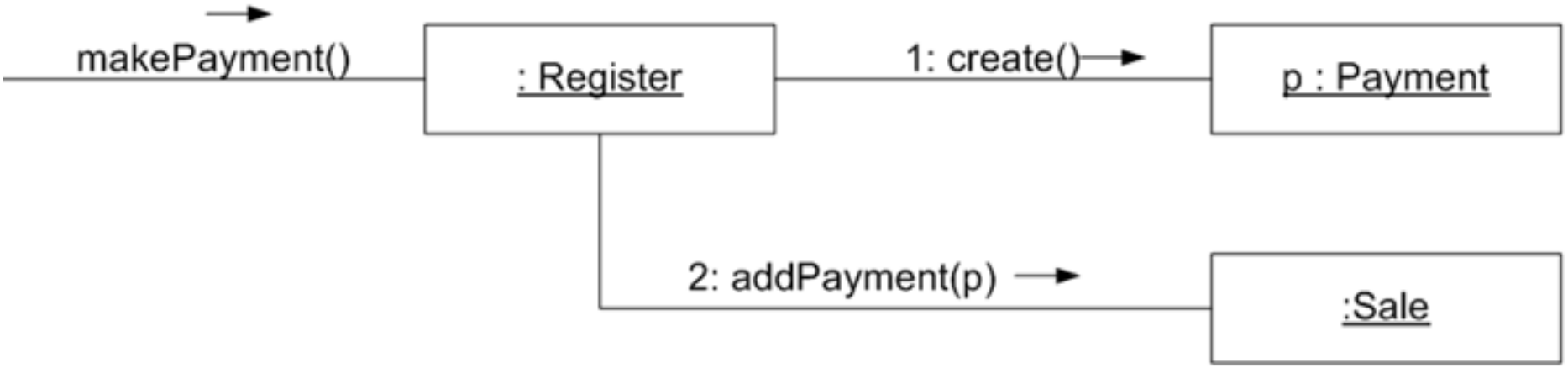
1. Bardzo niska spójność - bardzo dużo zadań w różnych obszarach funkcjonalnych (*np. klasa opowiada za logikę i za interakcję z bazą danych*)
2. Niska spójność - wiele bardzo różnych zadań w jednym obszarze funkcjonalnym (*np. jedna klasa w pełni odpowiada za interakcję z bazą danych*)
3. Wysoka spójność - mała liczba podobnych zobowiązań w ustalonym obszarze funkcjonalnym (*np. grupa klas realizuje obsługę bazy danych*)

Czy klasa `Register` powinna wykonywać niskopoziomowe zadania czy je delegować i organizować działanie innych klas?

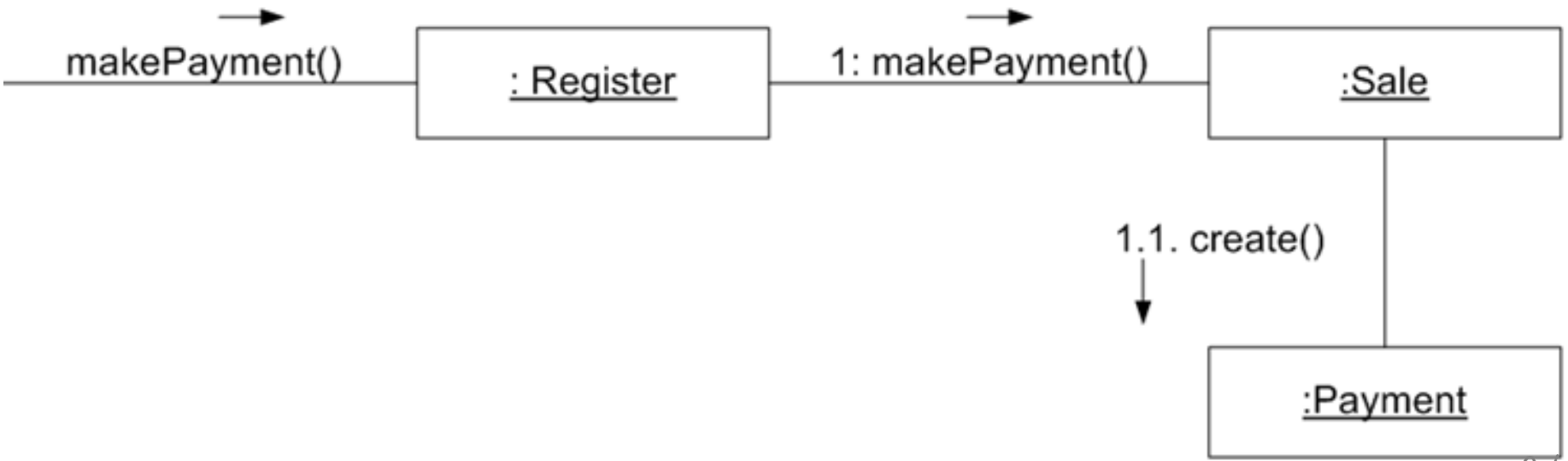
Niska spójność



Niska spójność



Wysoka spójność



High Cohesion – zasada ewaluacyjna

1. Oceniając różne możliwości przydzielaj metody tak, by spójność pozostała wysoka

2. Zysk:
   - ▷ kod łatwy w utrzymaniu i zrozumiały
   - ▷ High Cohesion $\leftrightarrows$ Low Coupling
     (*np. klasa odpowiedzialna za GUI i logikę jest wysoce niespójna, ale też silnie powiązana*)

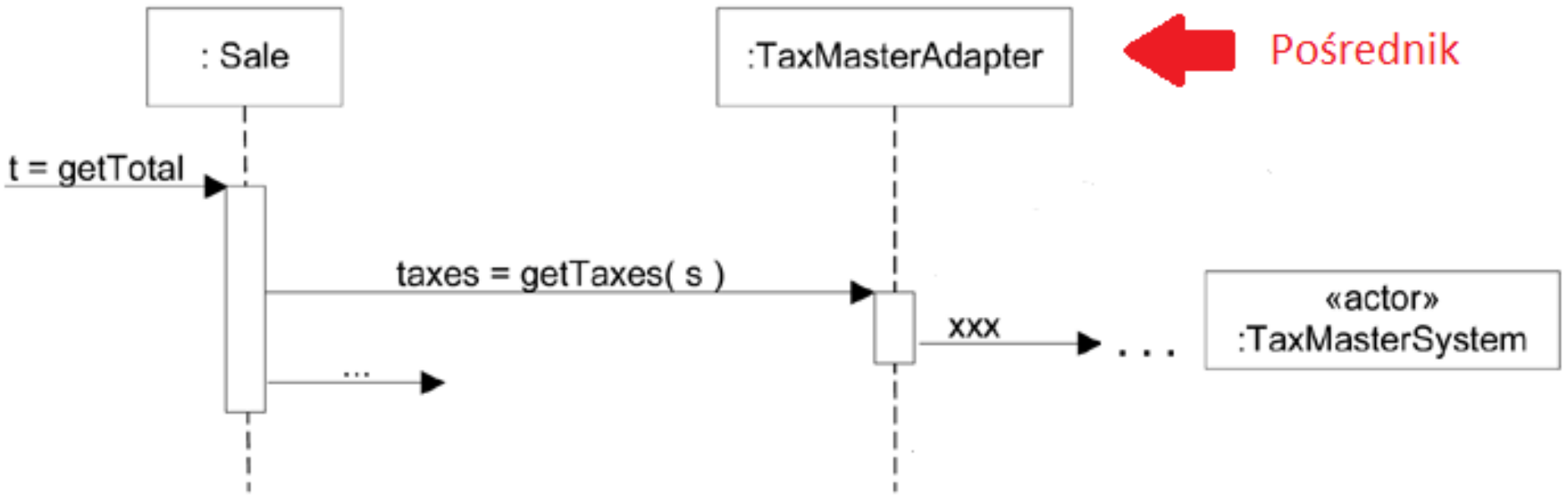
Pure Fabrication – czyli czysty wymysł

- Problem: ograniczenie się w modelu projektowym do obiektów z modelu dziedziny może prowadzić do niskiej spójnością i wysokiego sprzężenia
- Pomysł: wymyśl klasę pomocniczą, spoza modelu dziedziny i przydziel jej odpowiednie zobowiązania
- Przykład: gdy każda klasa odpowiada za swój zapis do bazy to wysokie sprzężenie i niska spójność, stwórzmy więc specjalne klasy do obslugi bazy (*nie nadużywać dekompozycji behawioralnej - programujemy obiektowo!*)

Indirection – czyli pośrednictwo

Pośrednictwo

Unikaj bezpośrednich powiązań do elementów potencjalnie niestabilnych. W razie potrzeby wykorzystaj obiekt pośredniczący.



Indirection — czyli pośrednictwo

Pośrednictwo

Unikaj bezpośrednich powiązań do elementów potencjalnie niestabilnych. W razie potrzeby wykorzystaj obiekt pośredniczący.

Komentarz

1. Wiele problemów projektowych można rozwiązać wprowadzając dodatkowy poziom pośrednictwa.
2. Wiele problemów z wydajnością można rozwiązać przez usunięcie poziomu pośrednictwa...

Polymorphism

Polimorfizm – w praktyce oznacza nadanie tej samej nazwy metodom różnych klas, gdy metody te są związane ze sobą:

- overriding – klasy mają wspólny interfejs lub nadklasę, metody mają taką samą sygnaturę (*głównie o tym myślimy*)
- overloading – metody mają tą samą nazwę ale w zależności od sygnatury mogą mieć inny kod

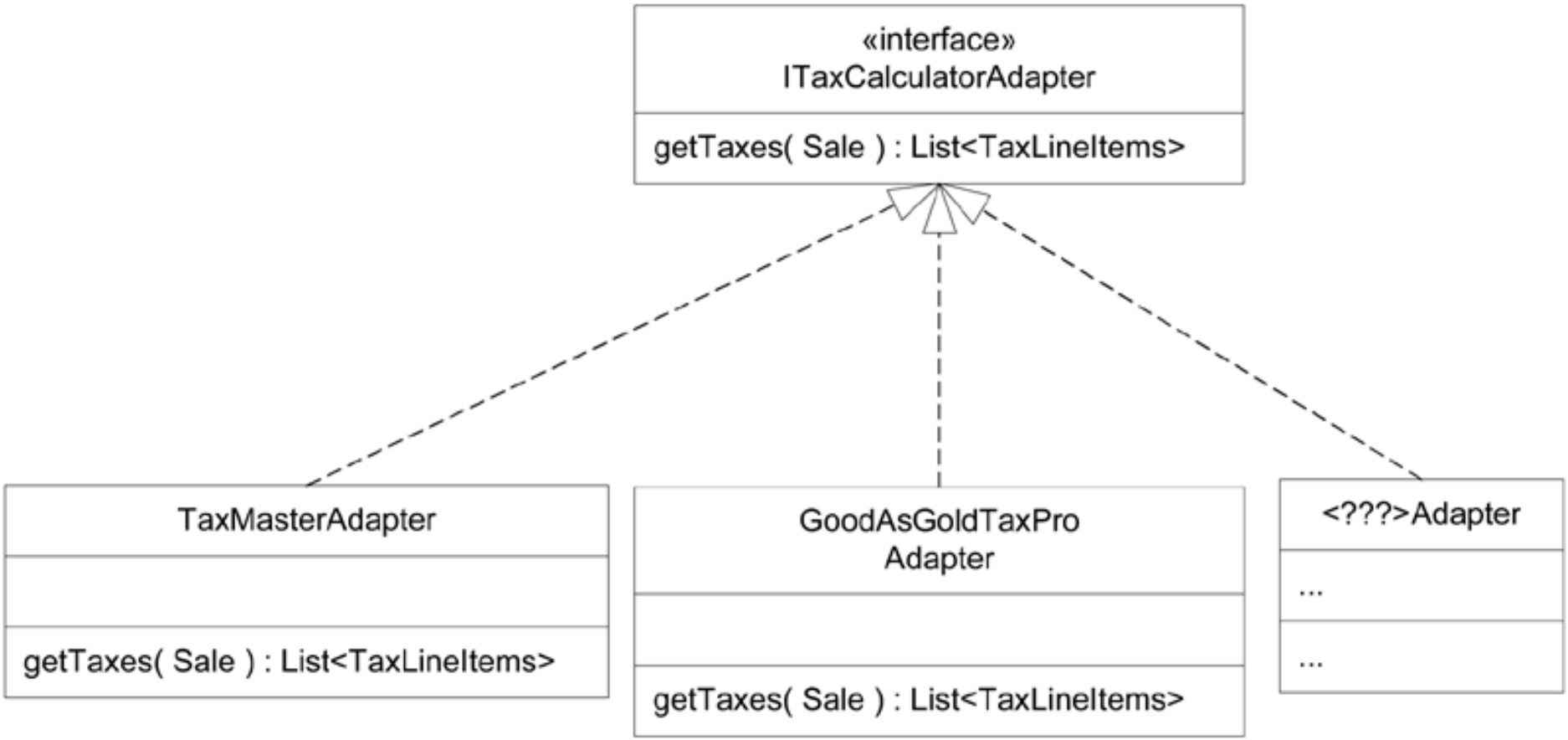
Override vs. Overload - zadanie na liści i na kolokwium...

(+ *adnotacja @Override*)

Polymorphism – przykład

1. `Sale` będzie musiała wyliczać podatek przy użyciu zewnętrznego kalkulatora podatków
2. Kalkulatorów jest dużo, mają różne interfejsy...
3. Jaki obiekt powinien odpowiadać za ich obsługę?

Polymorphism – przykład



- Zobowiązanie ma postać polimorficznej metody `getTaxes`
- Ogólnie: użyj polimorfizmu by rozdzielić zobowiązania jeśli wybór metody zależy od typu (*zamiast if-then-else*)

Protected Variations – ochrona przed zmiennością

- Jak projektować system by był odporny na zmiany i łatwo rozszerzalny?
- Ogólna zasada: rozpoznaj miejsca, w których zmiany mogą się pojawić i otocz je stabilnym interfejsem
- Przykład: kalkulator podatków jest punktem zmienności (*prawo się zmienia, wielu usługodawców, różne api*)

Protected Variations – podstawowe zasady

- Open-Close Principle: klasy powinny być otwarte na rozszerzenia i zamknięte na modyfikacje
- Law of Demeter (*nie rozmawiaj z obcymi*) – czyli do kogo metoda moze wysłać komunikat:
  1. obiekt `this`, atrybut `this`
  2. parametr metody
  3. obiekt utworzony wewnątrz metody

```
//źle
AccountHolder holder =
 sale.getPayment().getAccount().getAccountHolder();
```

```
//dobrze
AccountHolder holder = sale.getAccountHolderOfPayment();
```

Ochrona przed zmiennością — to ważne!

Większość narzędzi o których się uczyliście i będziecie się uczyć służy ochronie przed zmiennością, np.:

- interfejsy, polimorfizm
- wzorce GRASP, GoF
- data driven programming
- programowanie refleksyjne

„Dojrzałość programistyczna" (Larman)

Rozpoznawanie gdzie ochrona przed zmiennością jest potrzebna i dobry wybór metod jej realizacji.

Protected Variations – data driven programming

- Wyprowadzanie zmiennego elementu na zewnątrz systemu i wczytywanie potrzebnych danych (*wartości zmiennych i ścieżek, nazw klas, logiki...*)
- Przykłady:
  – programowanie refleksyjne - wybór i wiązanie obiektów w czasie wykonania programu
  – wstrzykiwanie zależności (*Spring*),
  – metadane w mapowaniu relacyjno-obiektowym (*Hibernate*)

http://stackoverflow.com/questions/1065584/what-is-data-driven-programming

Protected Variations – programowanie refleksyjne

Bez refleksji

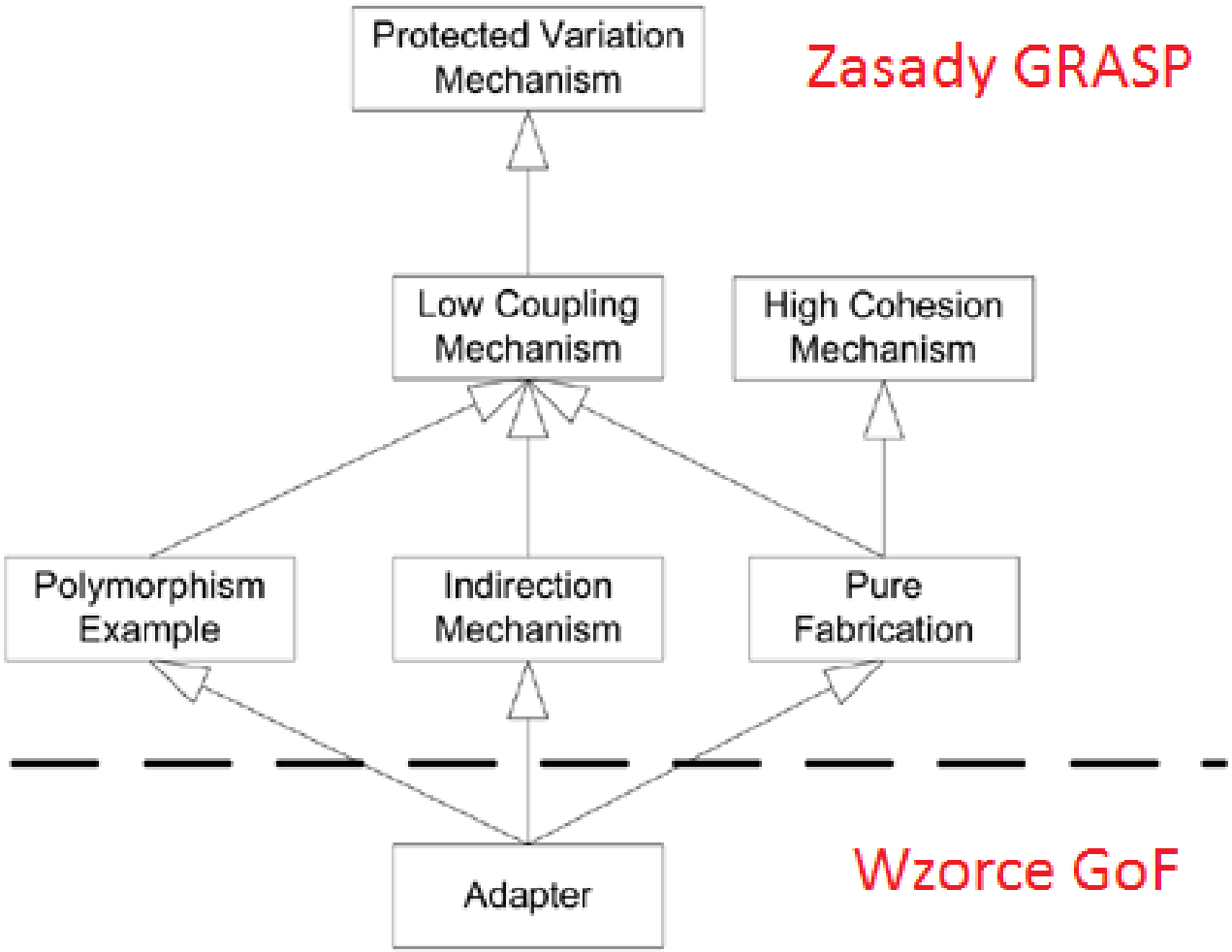
```
Foo foo = new Foo();
foo.hello();
```

Z refleksją

```
import java.lang.reflect.*;
...
Class c = Class.forName("Foo");
Method m = c.getMethod("hello");
m.setAccessible(true);                    //gdy m prywatna!
m.invoke(c.newInstance());
```

Zaleta: modyfikacje w czasie wykonania(bez re-kompilacji).
Wady? Bezpieczeństwo? Poprawność? Czas działania?

GRASP a GoF

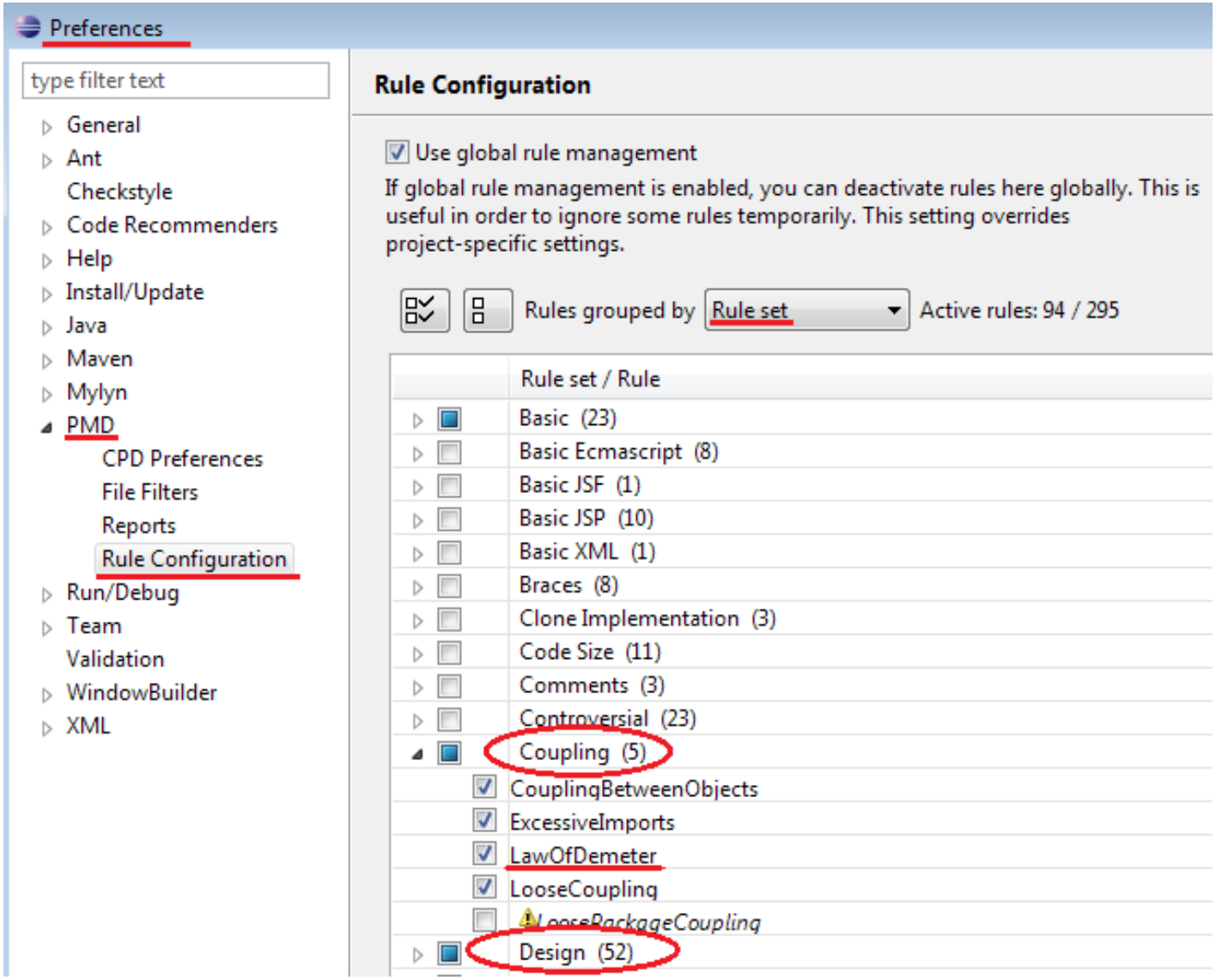

Protected Variation Mechanism
Zasady GRASP
Low Coupling Mechanism
High Cohesion Mechanism
Polymorphism Example
Indirection Mechanism
Pure Fabrication
Adapter
Wzorce GoF

Zasady GRASP a PMD – Project Mess Detector

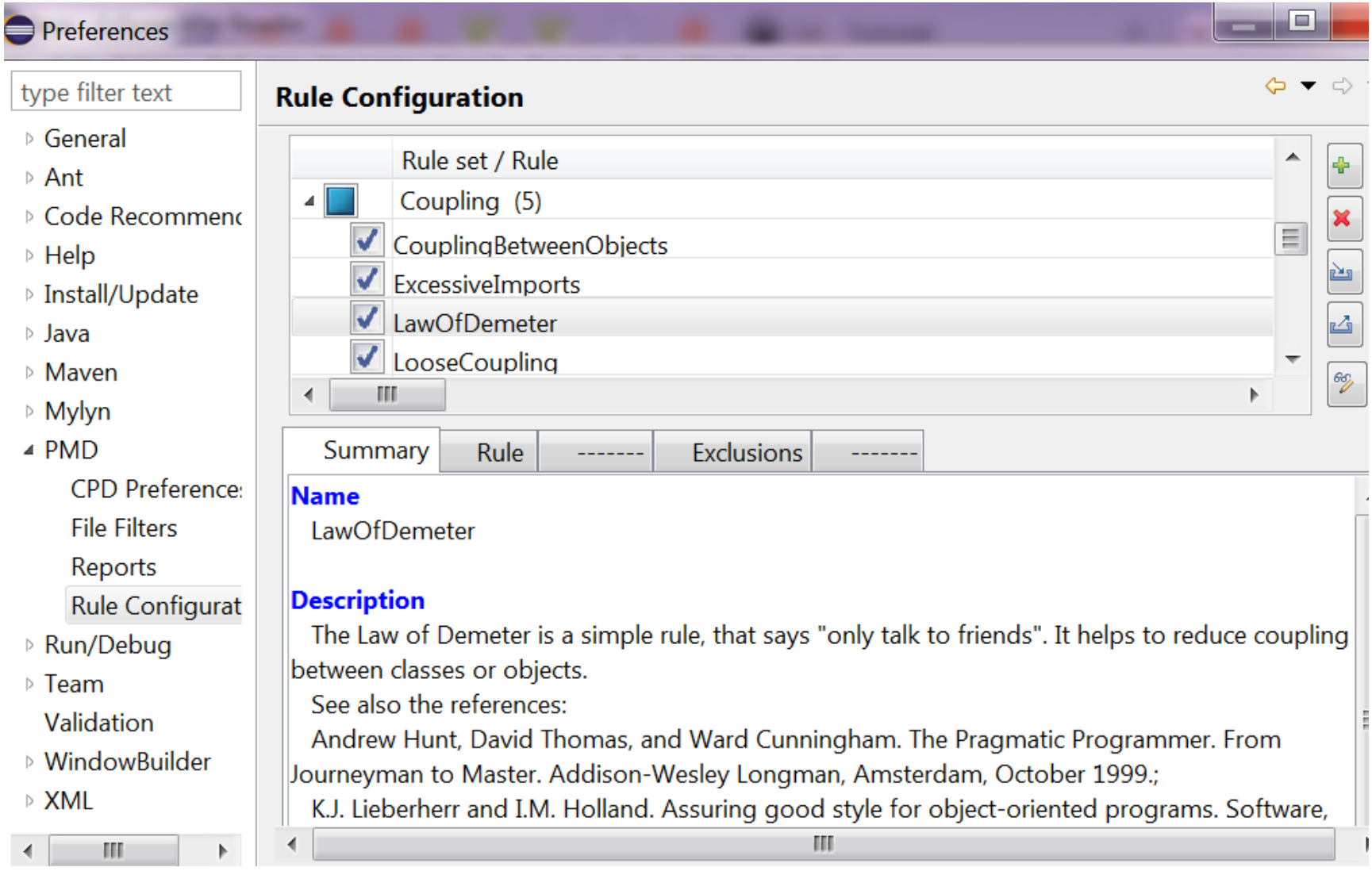
PMD – plugin do Eclipse, statyczny analizator kodu, wykrywa:

- martwy kod (*nieużywane zmienne, parametry i metody*)
- nadmiernie skomplikowany kod (*pętle, warunki*)
- zduplikowany kod
- nadmierne powiązanie klas (*coupilng*)
- nieskomplikowane wady projektowe
- ...

Zasady GRASP a PMD



PMD: opisy błędów



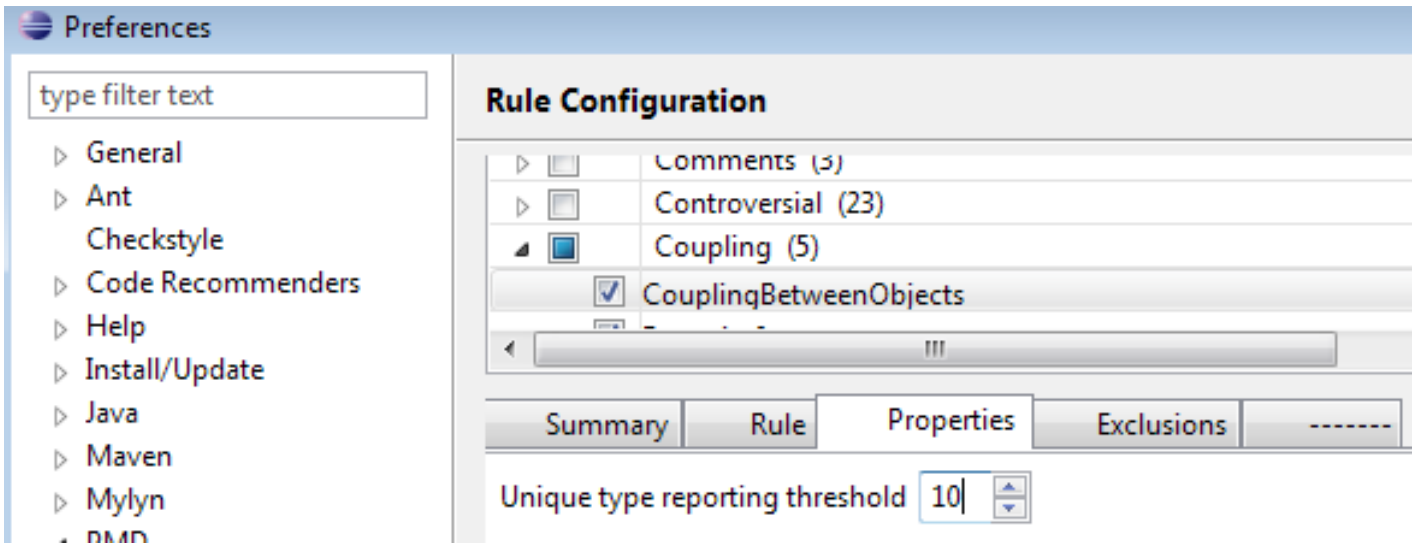
PMD: Low-of-Demeter

Low-of-Demeter ≡ Nie rozmawiaj z obcymi

```
public class Foo {
    public void example(Bar b) {

        b.getC().doIt(); // Nie OK

        C c = b.getC();
        c.doIt();        // Nie OK

        b.doItOnC();     // OK

        ...
```

PMD: Coupling i inne zasdy

PMD → Rule Set → Coupling → CouplingBetweenObjects
→ Design → GodClass
→ Design → UseSingleton
→ ...



SOLID

Inne zasady o które możecie zostać zapytani na rozmowie kwalifikacyjnej: SOLID, KISS, DRY, YAGNI

- Single responsibility
- Open-closed principle
- Liskov substitution
- Interface segregation
- Dependency inversion

SOLID

- Single responsibility
- Open-closed principle
- Liskov substitution

Metody, które używają referencji do klas bazowych, muszą być w stanie używać również obiektów klas dziedziczących po klasach bazowych ($\approx$ polymorphism, design by contract)

- Interface segregation
- Dependency inversion

SOLID

- Single responsibility
- Open-closed principle
- Liskov substitution
- Interface segregation

Podziel „duże” interfejsy na mniejsze i dokładniejsze, tak by żaden klient nie był zależny od metod których nie używa

- Dependency inversion

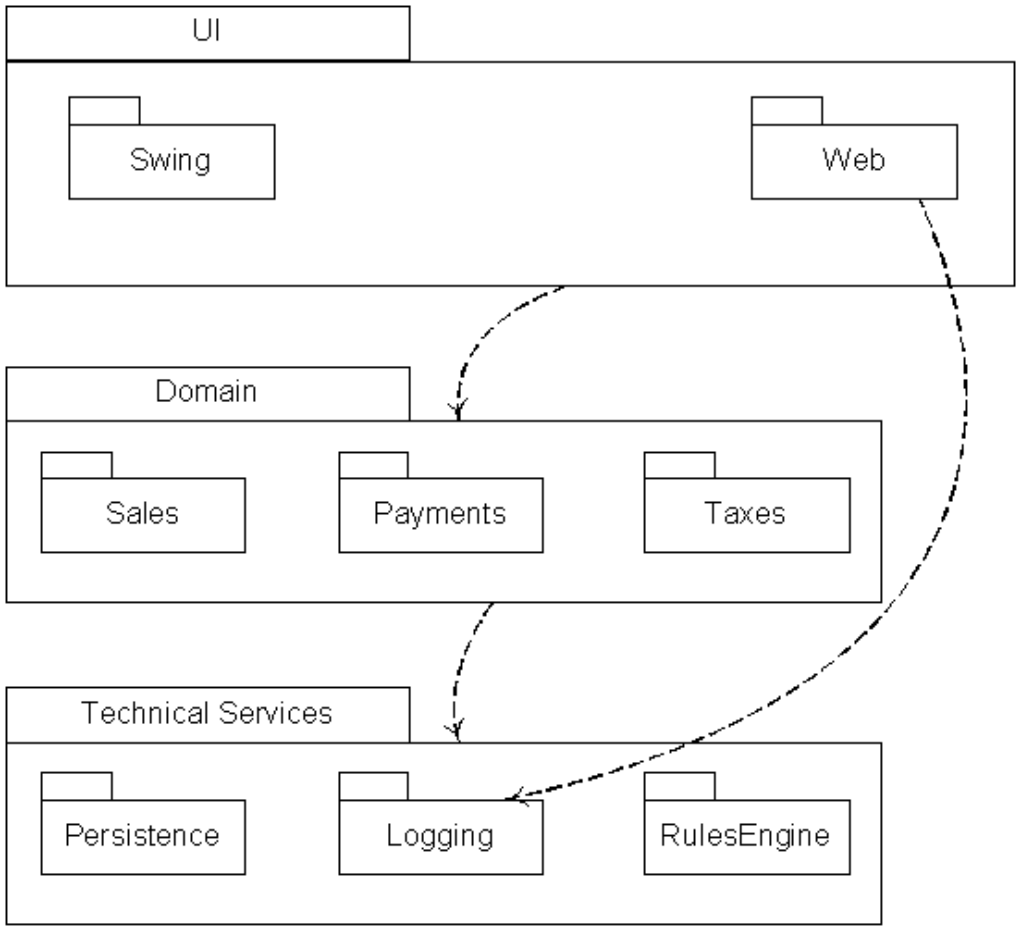
SOLID

- Single responsibility
- Open-closed principle
- Liskov substitution
- Interface segregation
- Dependency inversion

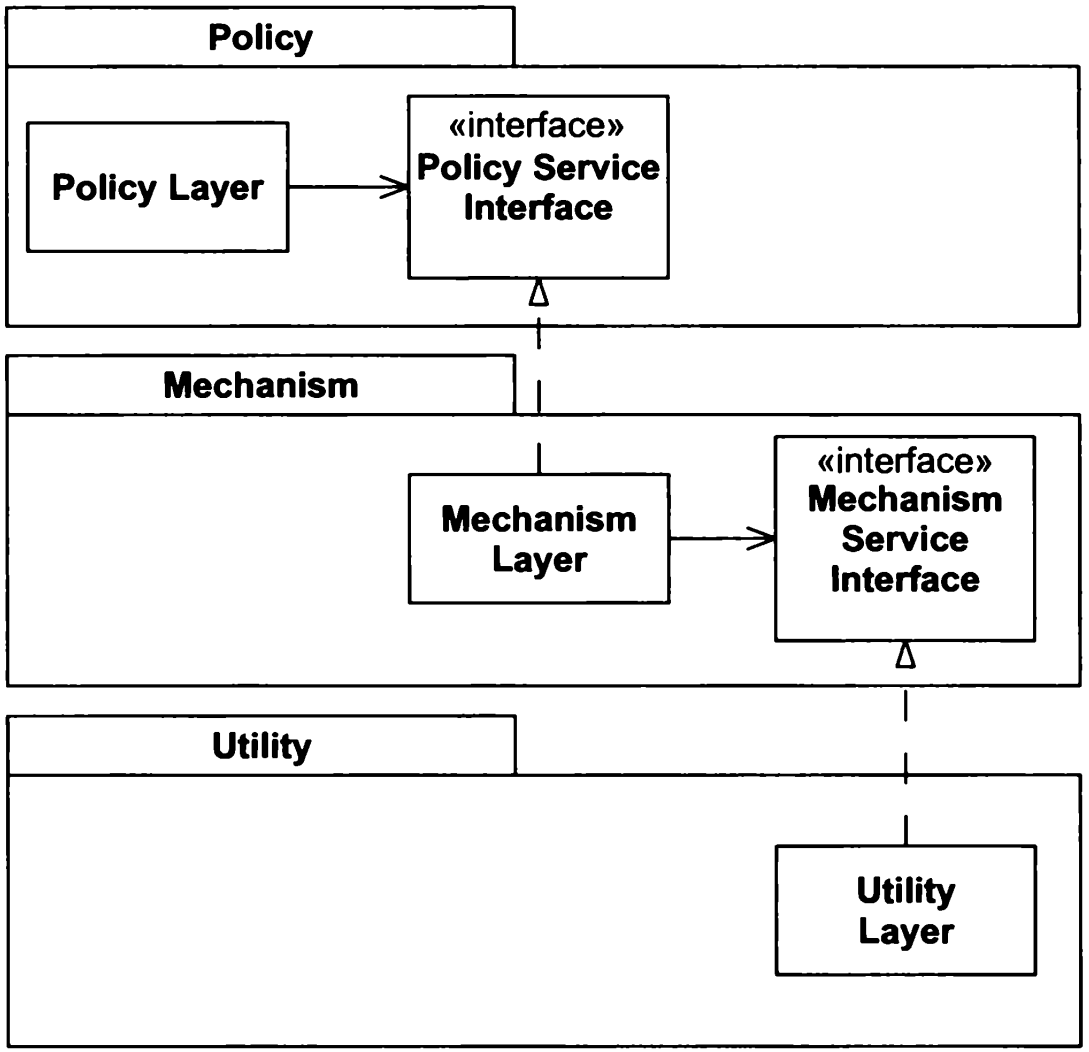
Uzależniać klasy od abstrakcji, a nie od konkretnych klas.
Decyzje o wiązaniu obiektów wyprowadzamy „na zewnątrz”
(*przykład: Spring Dependency injection*)

Dependency inversion — klasyczna architektura warstwowa

Warstwa to grupa pakietów o podobnym zakresie odpowiedzialności. Klasycznie warstwy układa się tak, aby „wyższe” wywoływały usługi „niższych” (*najlepiej sąsiednich*), a niższe nie zależały od wyższych.



Dependency inversion — odwórcenie klasycznych zależności



Wyższe warstwy zależą od interfejsów a nie od warstw niższych
(*odporniejsze na zmiany, można je przenieść*)

KISS

KISS – Keep It Simple, Stupid (*BUZI – Bez Udziwnień Zapisu, I.*)

Prostota jest szczytem wyrafinowania

- Czasem mamy ambicje by stworzyć skomplikowane rozwiązanie które będzie niezrozumiałe i zapewne niekoniecznie potrzebne (*np. bardzo wydajne ale skomplikowane i mało uniwersalne*)
- Może to powodować późniejsze problemy z utrzymaniem kodu (*zrozumienie, naprawa lub modyfikacja*)
- Dotyczy ogólnej architektury i niskopoziomowych rozwiązań

Pisz kod dla innych ludzi, a nie dla komputera – komputerowi jest wszystko jedno, a nie wszyscy lubią zagadki

YAGNI

YAGNI – You Ain't Gonna Need It

- Nie pisz kodu, tylko dlatego, że wydaje Ci się, że będziesz go potrzebował w przyszłości
- Jest jest spora szansa, że w między czasie się coś zmieni i nie będziesz go potrzebował lub będziesz musiał zmieniać swoje rozwiązanie
- Bądź leniem i minimalistą – twórz kod, tylko wtedy gdy go potrzebujesz
- Test Driven Development – pisz tylko tyle by przejść testy

DRY

Don't Repeat Yourself

- Nie powtarzaj tego samego kodu w wielu miejscach (*to utrudnia późniejsze poprawki, ang. maintenance*)
- Jeśli ten sam lub podobny kod występuje w wielu miejscach, Twój program prawdopodobnie wymaga refaktoryzacji
- Unikaj wielokrotnego powtarzania tych samych czynności (*automatyzacja, pisanie skryptów*)

Co koniecznie trzeba zapamiętać + zadanie

- Polimorfizm: override i overload
- Low Coupling, High Cohesion, SOLID
- Protected Variations: open-close principle, law of demeter

Przeanalizuj przykład MVC + refleksja

Czy jest on zgodny z zasadą Dependency Inversion?