Technologia Programowania 2017/2018
Wykład 4
Wzorce projektowe GoF

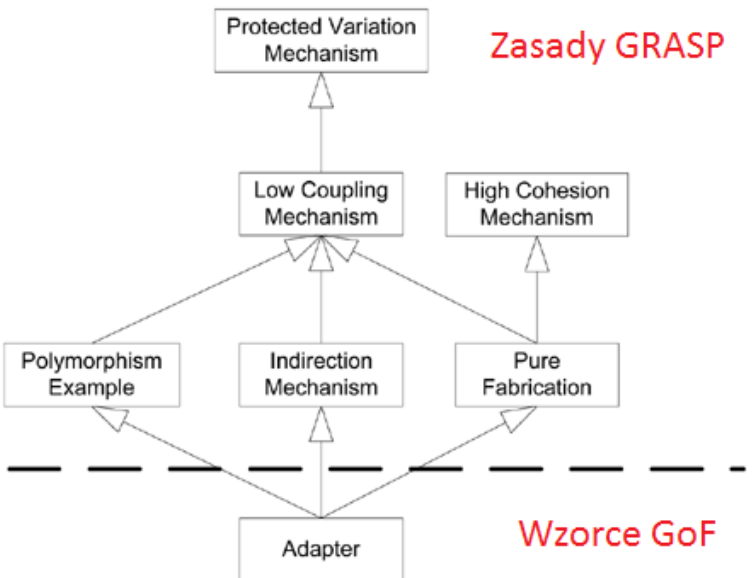
Jakub Lemiesz

Wzorce projektowe Gang of Four

✓ Wzorce GRASP – ogólne zasady projektowania

⇒ Na GRASP opierają się klasyczne wzorce GoF

⇒ Na wzorcach GoF opiera się wiele rozwiązań, bibliotek i frameworków (nie tylko w Java)



Wzorce projektowe Gang of Four

Twórcy: Gamma, Helm, Johnson, Vlissides – 1995

Klasyfikacja wzorców GoF

- Klasyfikacja wzorców GoF:
  1. kreacyjne – tworzenie i konfiguracja obiektów (*Singleton, Factory, Builder ...*)
  2. strukturalne – struktury powiązań obiektów (*Adapter, Composite, Facade, Proxy, ...*)
  3. czynnościowe – zachowanie, współpraca obiektów (*Observer, Strategy, State ...*)

- Są tez inne wzorce, np. wzorce architektoniczne: (*MVC, Client-Server, Reactor, ...*)

Wzorce GoF – co dzisiaj?

Kreacyjne

⇒ Builder

⇒ Singleton

Simple Factory

Factory Method

Abstract Factory

Prototype

Strukturalne

⇒ Adapter

⇒ Decorator

⇒ Proxy

Composite

Facade

Flyweight

Bridge

Czynnościowe

Strategy

Template Method

Observer

State

Command

Interpreter

Iterator

Mediator

Visitor

Memento

GoF: Adapter

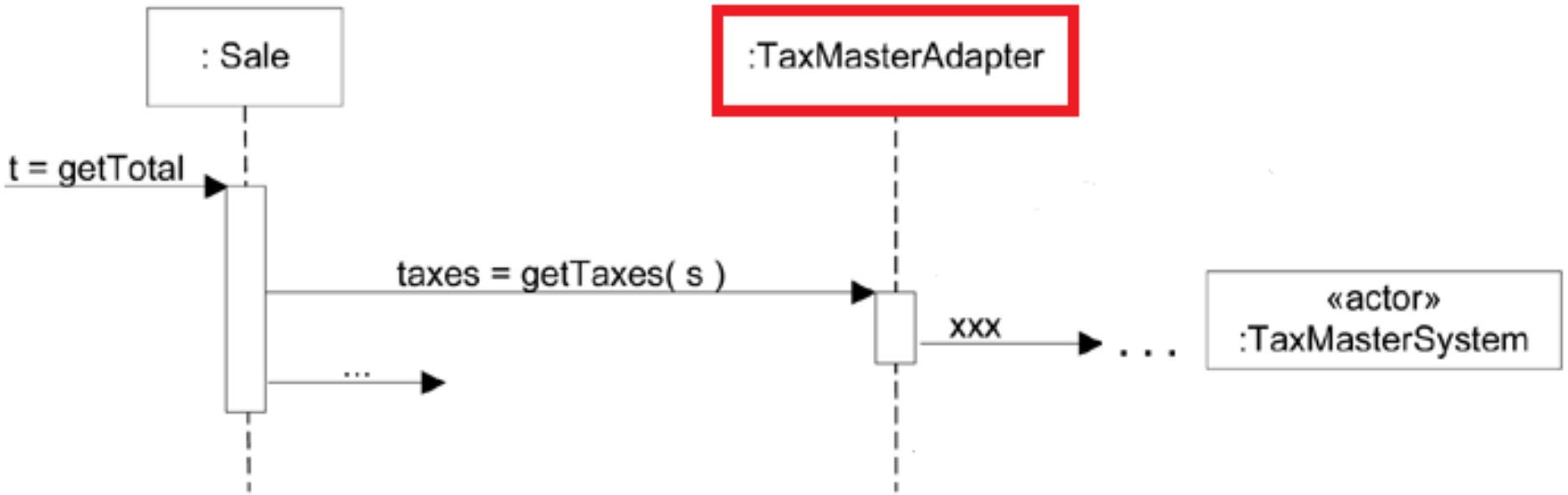
Typowy problem:

Jak wykorzystać „stare” albo „zewnętrzne” klasy i metody do nowych potrzeb bez wprowadzania zmian w kodzie?

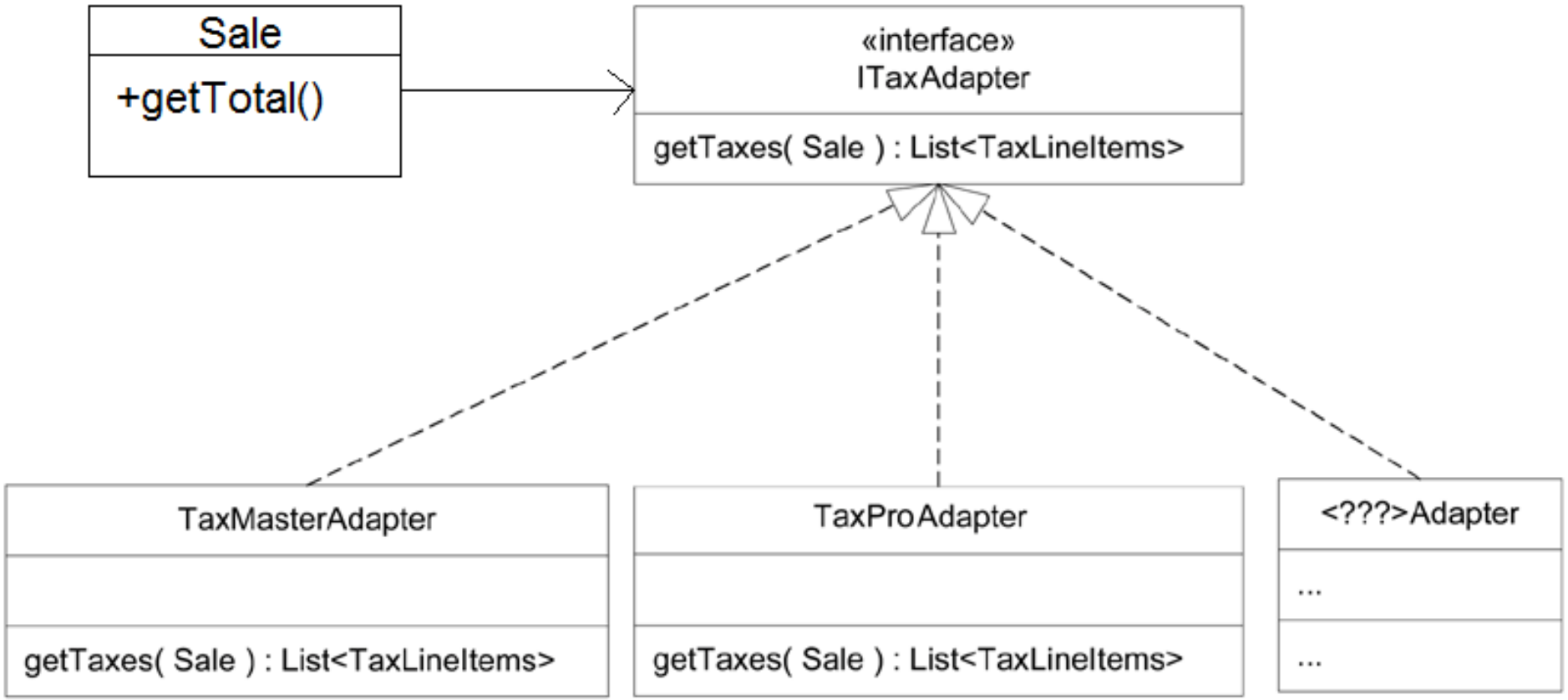
Rozwiązanie:

1. Klasa adaptera implementuje ustalony z góry interfejs
2. Adapter deleguje zadanie do „starej” albo „zewnętrznej” klasy, a następnie adaptuje jej odpowiedź

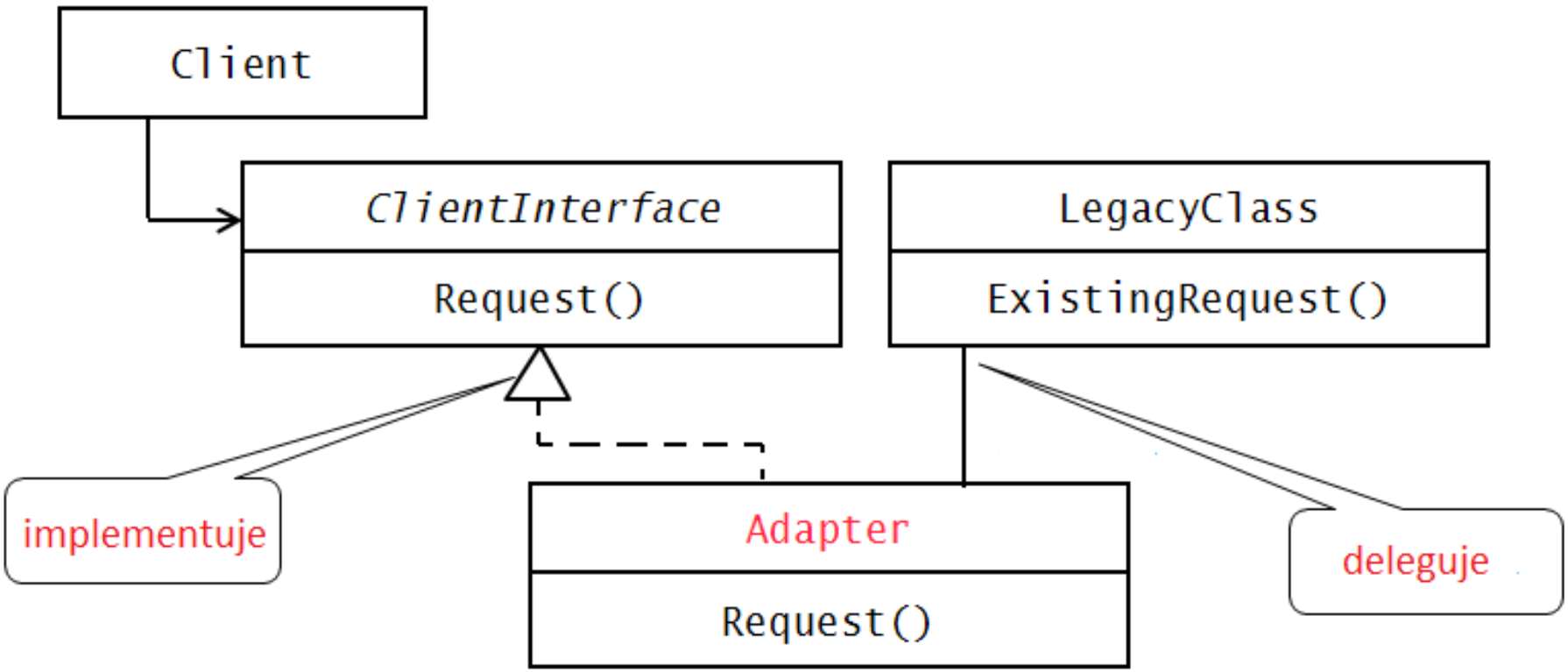
Przykład: kalkulator podatków z poprzedniego wykładu



Adapter: stabilny interfejs dla zewnętrznych komponentów

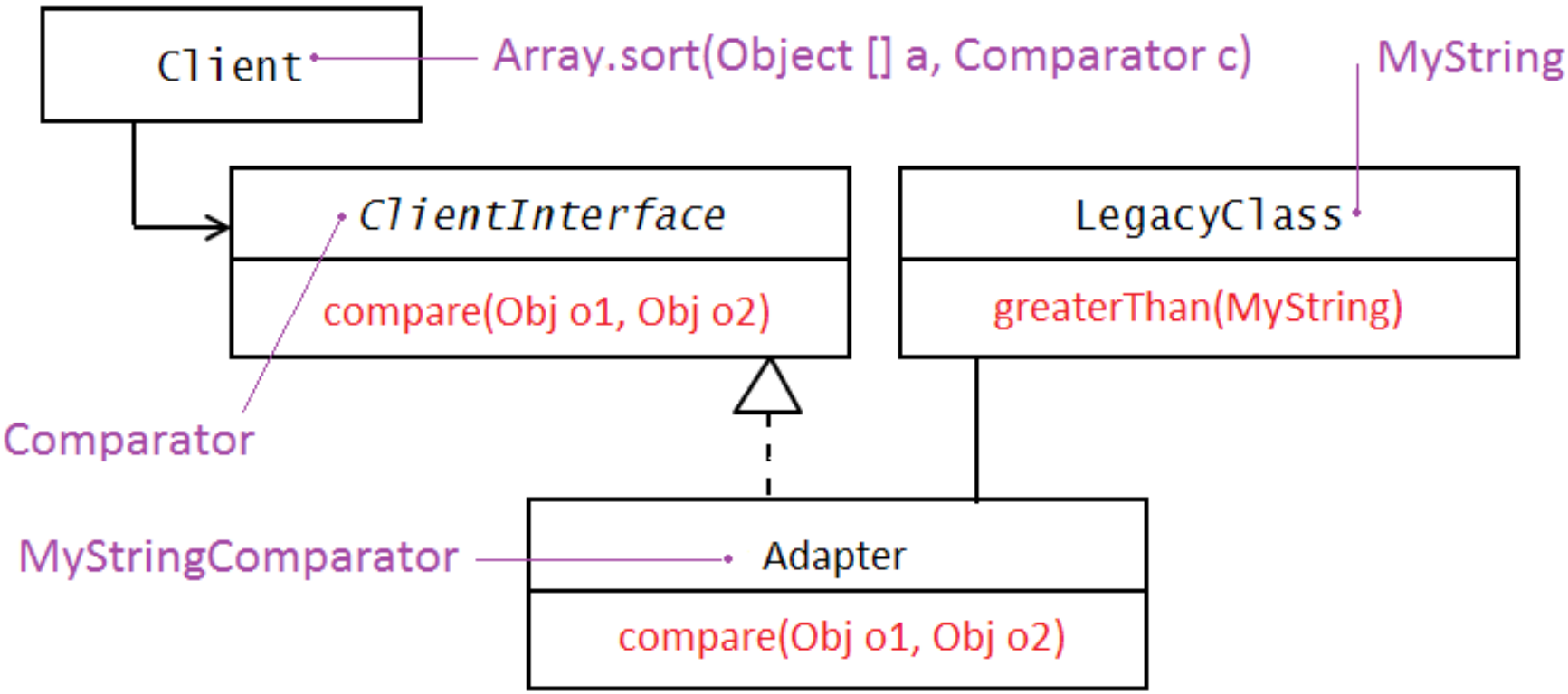


Przykład: adapter „starej” klasy



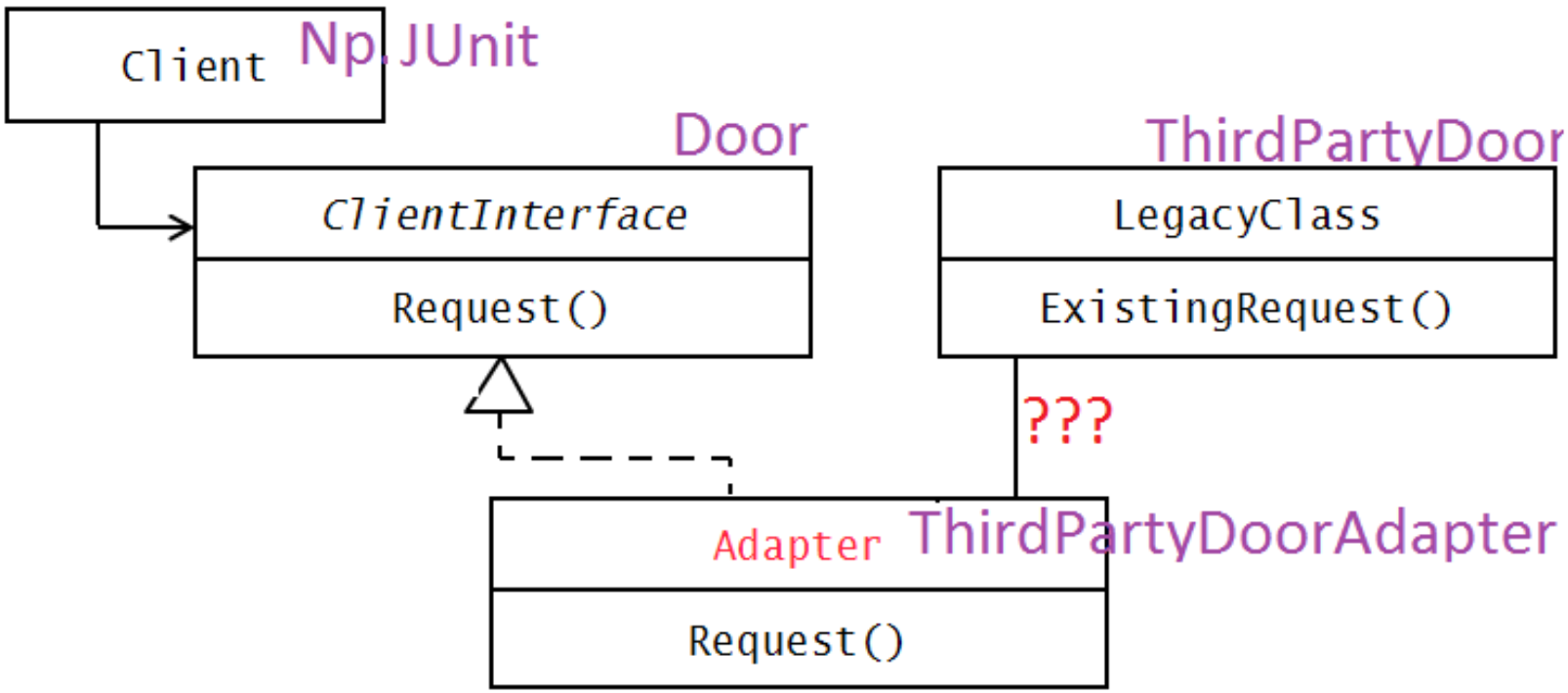
[Kod w Java]

Przykład: adapter „starej” klasy



[Kod w Java]

Zadania z wzorców na laboratorium



??? – dziedziczenie klas czy zawieranie obiektów?

(*Class Adapter vs. Object Adapter*)

GoF: Decorator

Przykład:

- Klasa `Coffee` ma metodę `getCost()`
- Kawa ma dodatki pojawiające się w dowolnej liczbie, np. kawa z mlekiem i potrójną posypką (*każdy dodatek ma swoją cenę*)
- `CoffeeWithMilkSprinkleSprinkleSprinkle` ze specjalnie zdefiniowaną metodą `getCost()` ?

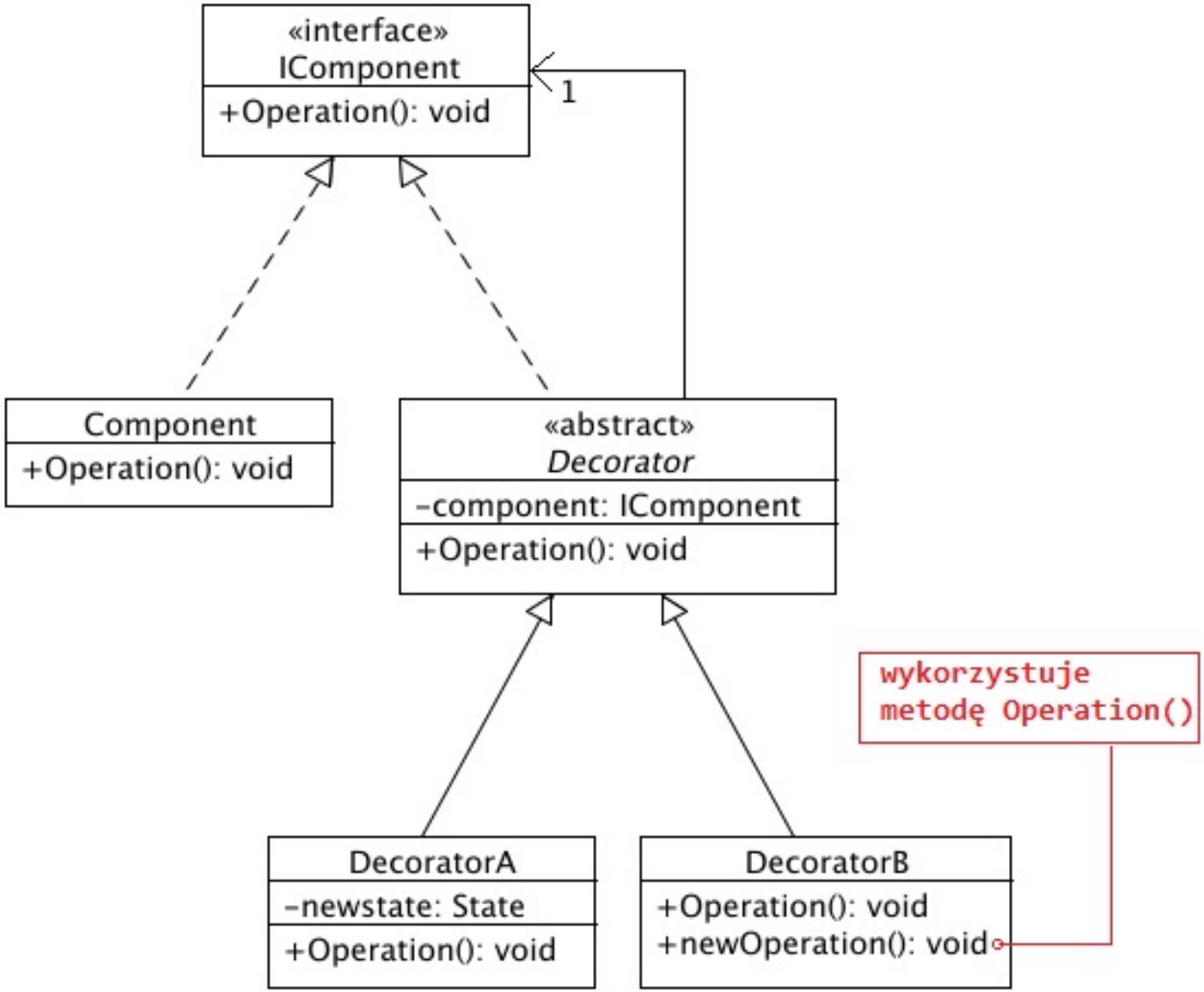
Problem:

Dynamiczna modyfikacja metod danej klasy (*np. nie wiadomo z góry jakie metody potrzebne*)

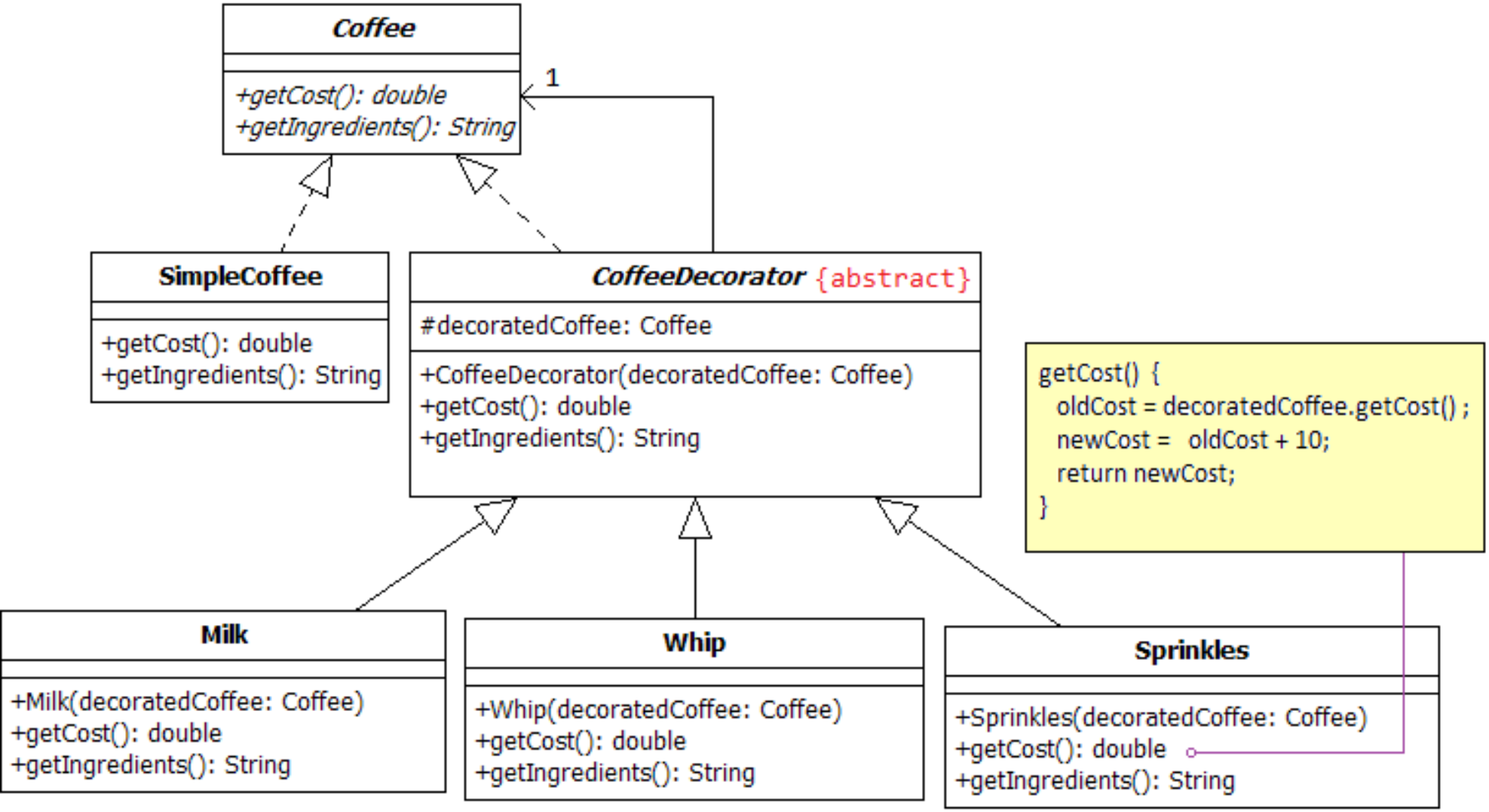
GoF: Decorator

Rozwiązanie:

1. Klasa dekoratora „opakowuję” oryginalną klasę (lub klasę innego dekoratora)
2. Metody dekoratora wywołują metody „opakowanej” klasy i dodają do niej nową funkcjonalność


«interface»
IComponent
+Operation(): void
1
Component
+Operation(): void
«abstract»
Decorator
-component: IComponent
+Operation(): void
wykorzystuje
metodę Operation()
DecoratorA
-newstate: State
+Operation(): void
DecoratorB
+Operation(): void
+newOperation(): void

Decorator — kod do przykładu



Źródło — wikipedia

Decorator — na czym polega dekorowanie?

- SmallCoffe - $5, BigCoffe - $10
- Milk - 2$, Whip - 2$ Sprinkles - 3$

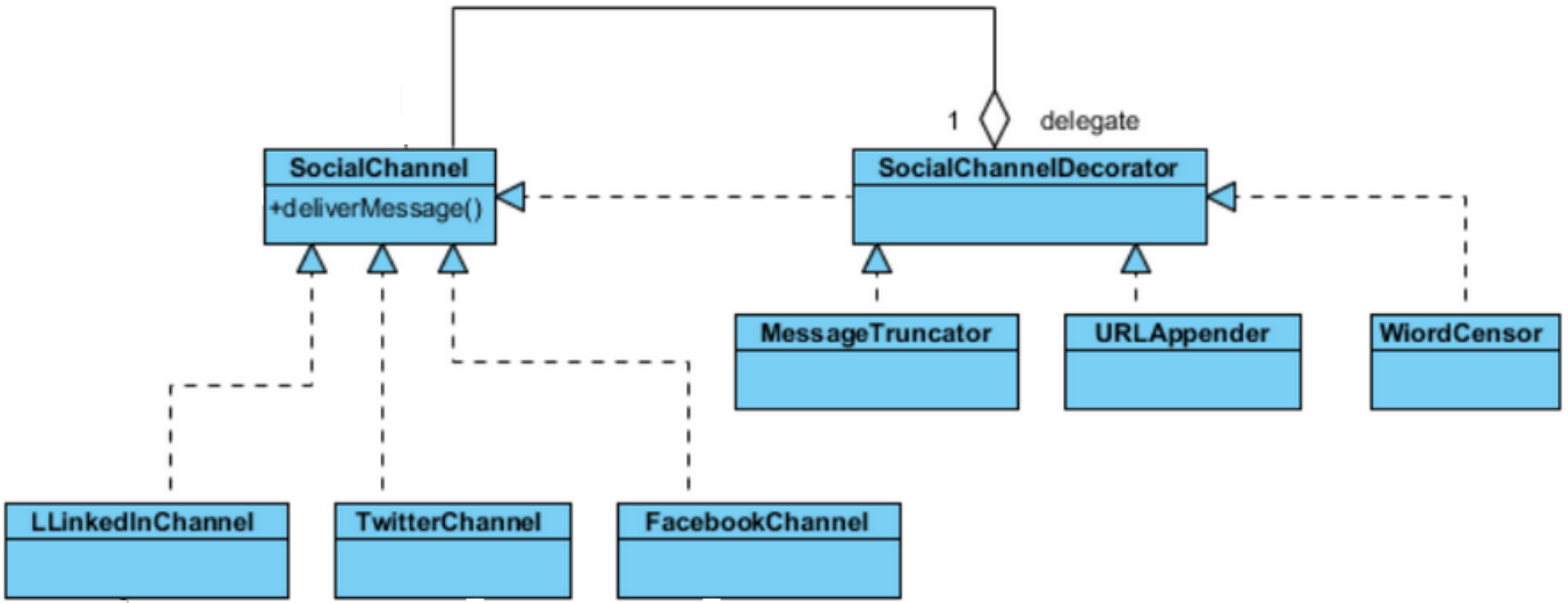
Jan: poproszę malą z mlekiem i potrójną posypką (!)

```
Coffee c = new SmallCoffee();
c = new Milk(c) ;
c = new Sprinkle(c);
c = new Sprinkle(c);
c = new Sprinkle(c);
c.getCost();
```

Decorator – uwagi

1. Dekorator zmienia metody dynamicznie, bez tworzenia hierarchi klas i przesłaniania (*wada: trudniej wykryć błędy*)
2. Duża elastyczność czasem powoduje trudności (*np. kolejność dekorowania bywa ważna*)
3. Obiekt klasy dekorowanej można przekazać jako parametr do konstruktora dekoratora – ale są inne rozwiązania (*np. na lab przez GoF: Builder*)

Decorator – zadanie na laboratorium



GoF: Proxy

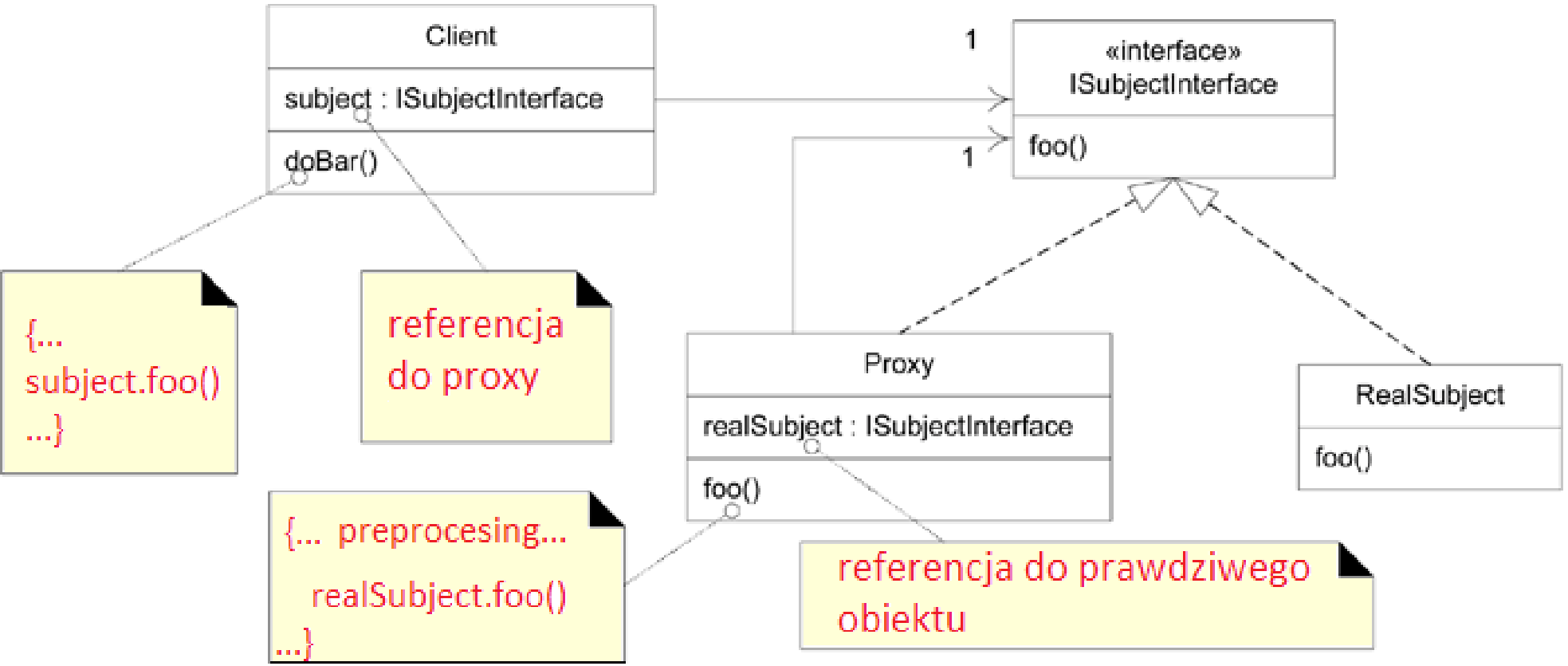
Problem:

Jak uniknąć bezpośredniego dostępu do obiektu (*np. wstępne przetwarzanie, obiekt zdalny, leniwa inicjalizacja*)

Rozwiązanie:

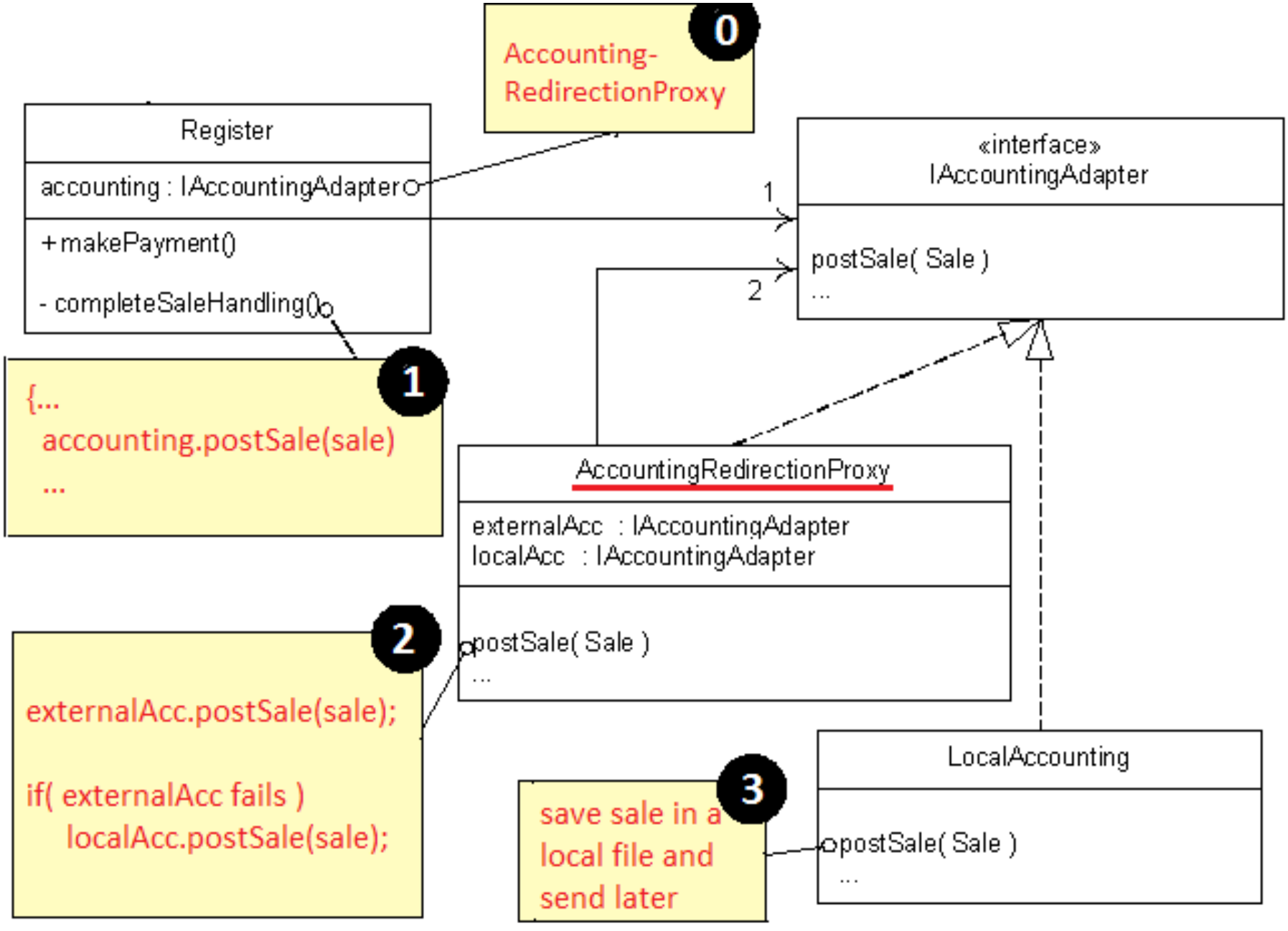
Wprowadzamy obiekt zastępczy (tzw. proxy), który implementuje ten sam interfejs co właściwy obiekt i odpowiada za kontrolę/poprawę dostępu

GoF: Proxy

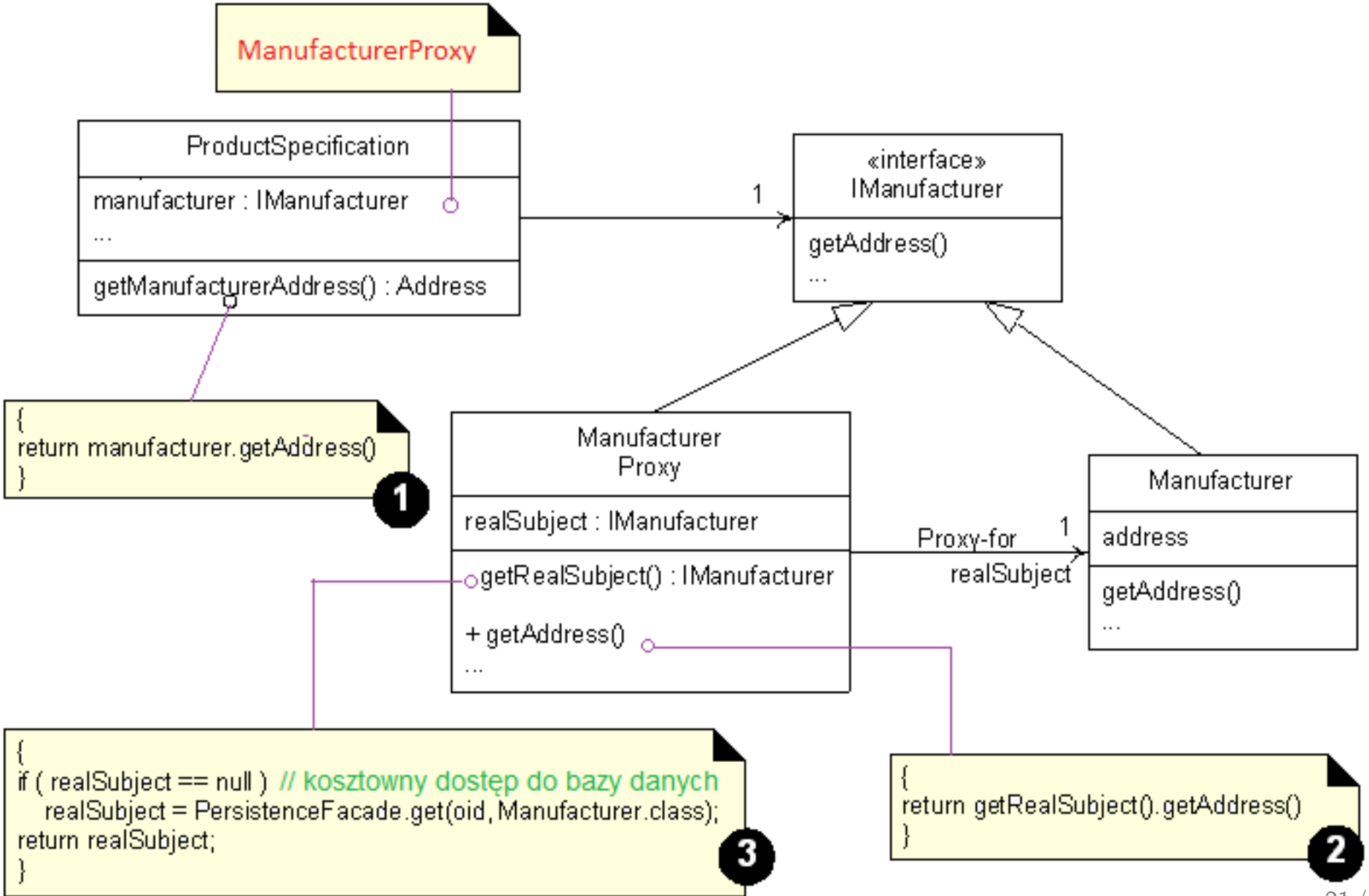

Client
subject : ISubjectInterface
doBar()
1
«interface»
ISubjectInterface
foo()
1
{...
subject.foo()
...}
referencja
do proxy
Proxy
realSubject : ISubjectInterface
foo()
RealSubject
foo()
{... preprocesing...
realSubject.foo()
...}
referencja do prawdziwego
obiektu

Proxy – przykład: usługa zdalna

- Wywołujemy usługę zdalną ale jeśli jest niedostępna chcemy wywołać lokalny „zamiennik” usługi
- Np. wysyłanie informacji o sprzedaży do zewnętrznego systemu księgowego
  1. Wywołujemy usługę zapisywania przez proxy
  2. Proxy wywołuje usługę zdalnego zapisu
  3. Jeśli zdalny zapis się nie powiedzie, proxy zapisuje informacje lokalnie


0
Accounting-
RedirectionProxy
Register
accounting : IAccountingAdapter
+makePayment()
- completeSaleHandling()
«interface»
IAccountingAdapter
postSale(Sale)
...
1
2
1
{...
accounting.postSale(sale)
...
AccountingRedirectionProxy
externalAcc : IAccountingAdapter
localAcc : IAccountingAdapter
postSale(Sale)
...
2
externalAcc.postSale(sale);
if(externalAcc fails)
localAcc.postSale(sale);
LocalAccounting
postSale(Sale)
...
3
save sale in a
local file and
send later

Proxy — przykład: leniwa inicjalizacja



Proxy – podsumowanie

1. Proxy to obiekt który opakowuje inny obiekt i jednocześnie implementuje ten sam interfejs, co obiekt opakowany. Klient nie wie, że odwołuje się do proxy.
2. Proxy przechwytuje wywołanie w celu kontroli lub usprawnienia dostępu do właściwego obiektu
3. Zastosowanie Proxy jest często wymuszane nie przez logikę biznesową, ale przez wymagania niefunkcjonalne (*np. wydajność, bezpieczeństwo*)

GoF: Builder

Przykład:

- Mamy klasę `Customer` i chcemy generować różne raporty (*np. nazwisko, adres, co kupił, ...*)
- i przedstawić je na różne sposoby (*np. kod HTML, XML, JComponent,...*)

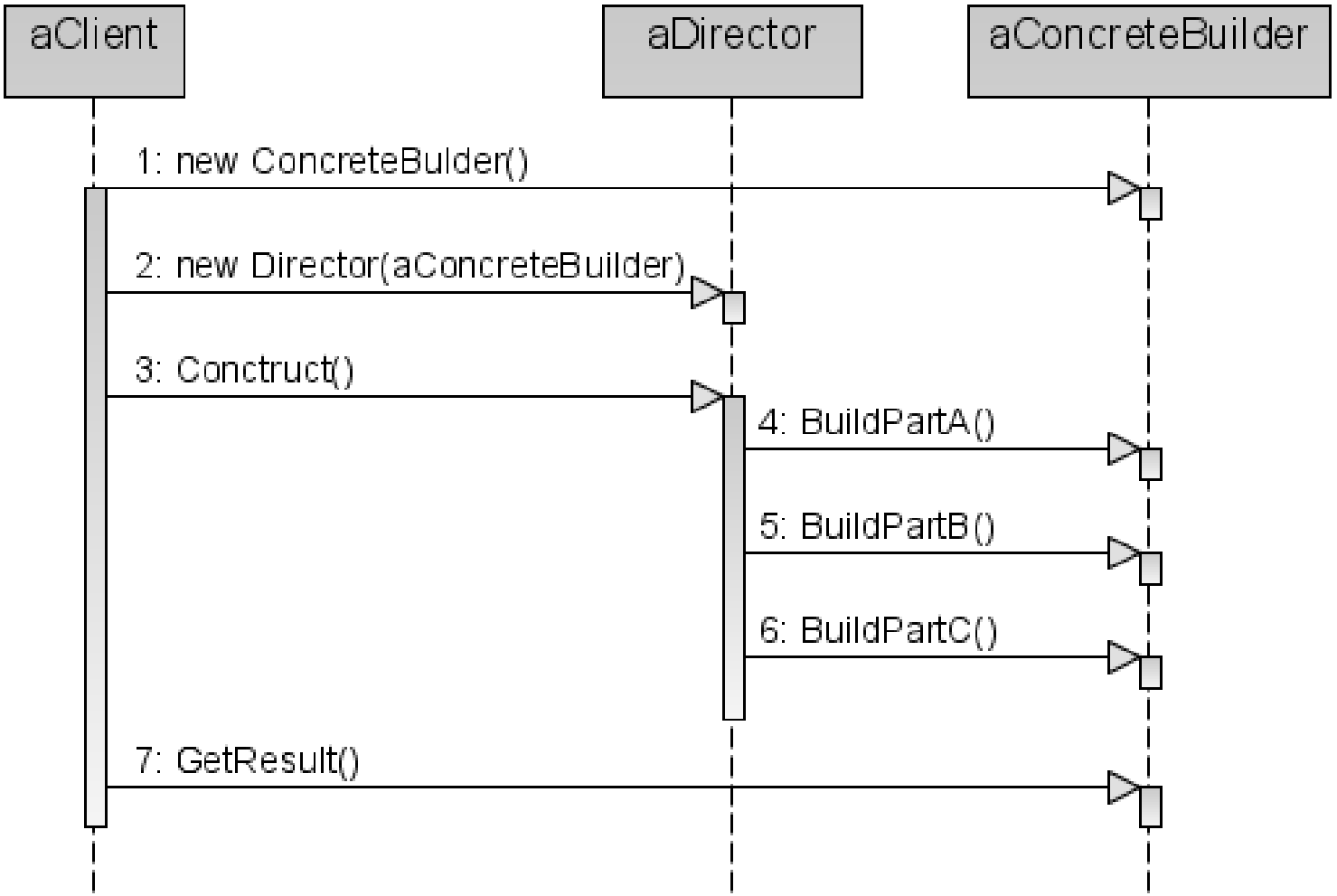
Problem:

jak rozłożyć odpowiedzialnosć by efektywnie tworzyć różne złożone obiekty w ramach jednego procesu? (*High Cohesion, Low Coupling*)

GoF: Builder – schemat działania

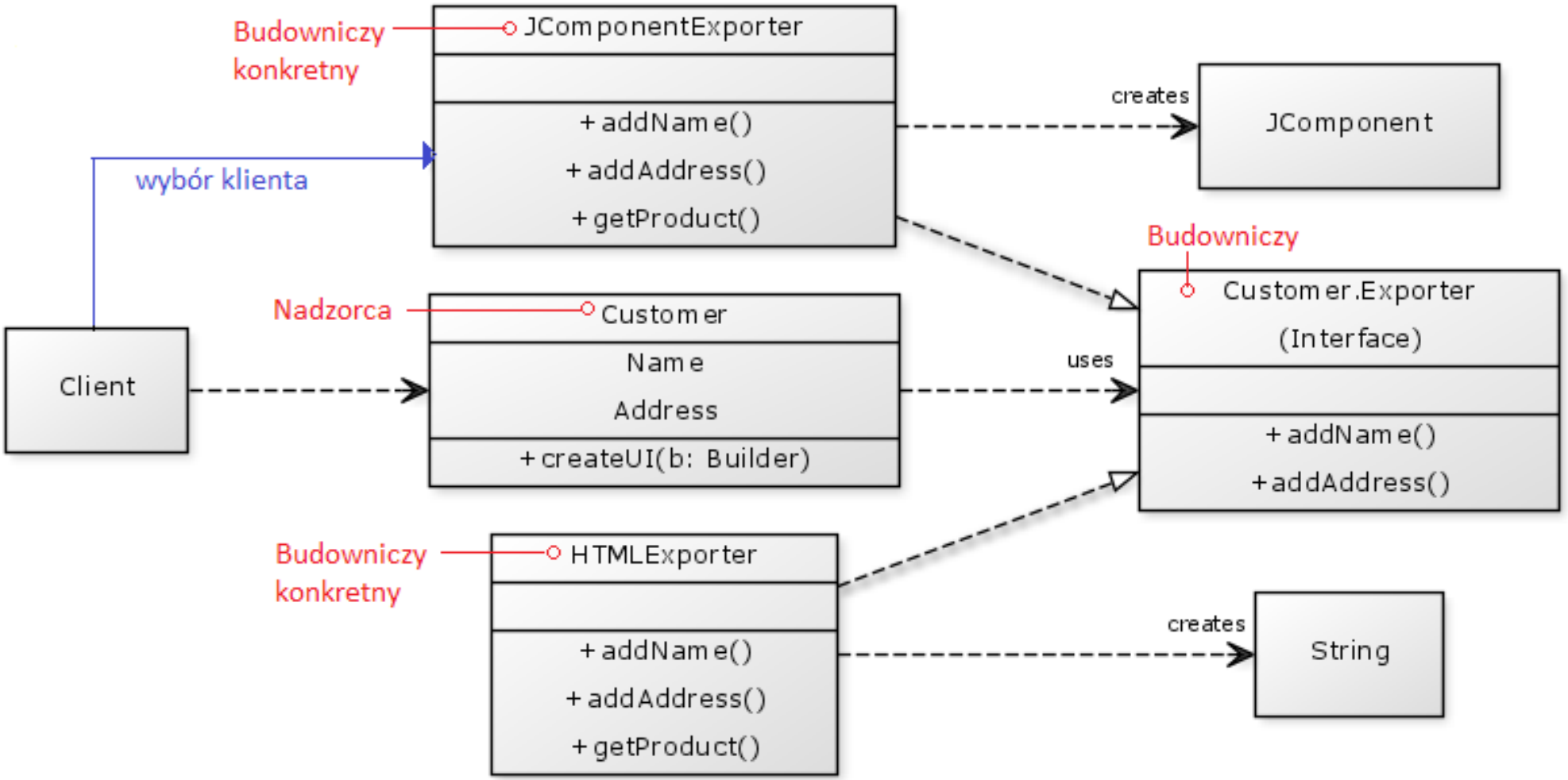
1. Tworzenie obiektu dzielimy na etapy, o tym jakie etapy mają być uwzględnione decyduje nadzorca (*np. jakie dane o z klasy Customer wybrać*)
2. O wyniku etapu decyduje budowniczy (*np. czy tagi HTML czy XML*)

GoF: Builder – schemat działania



Klient tworzy budowniczego i przekazuje do wybranego nadzorcy, nadzorca musi znać jedynie interfejs budowniczego.

GoF: Builder – przykład

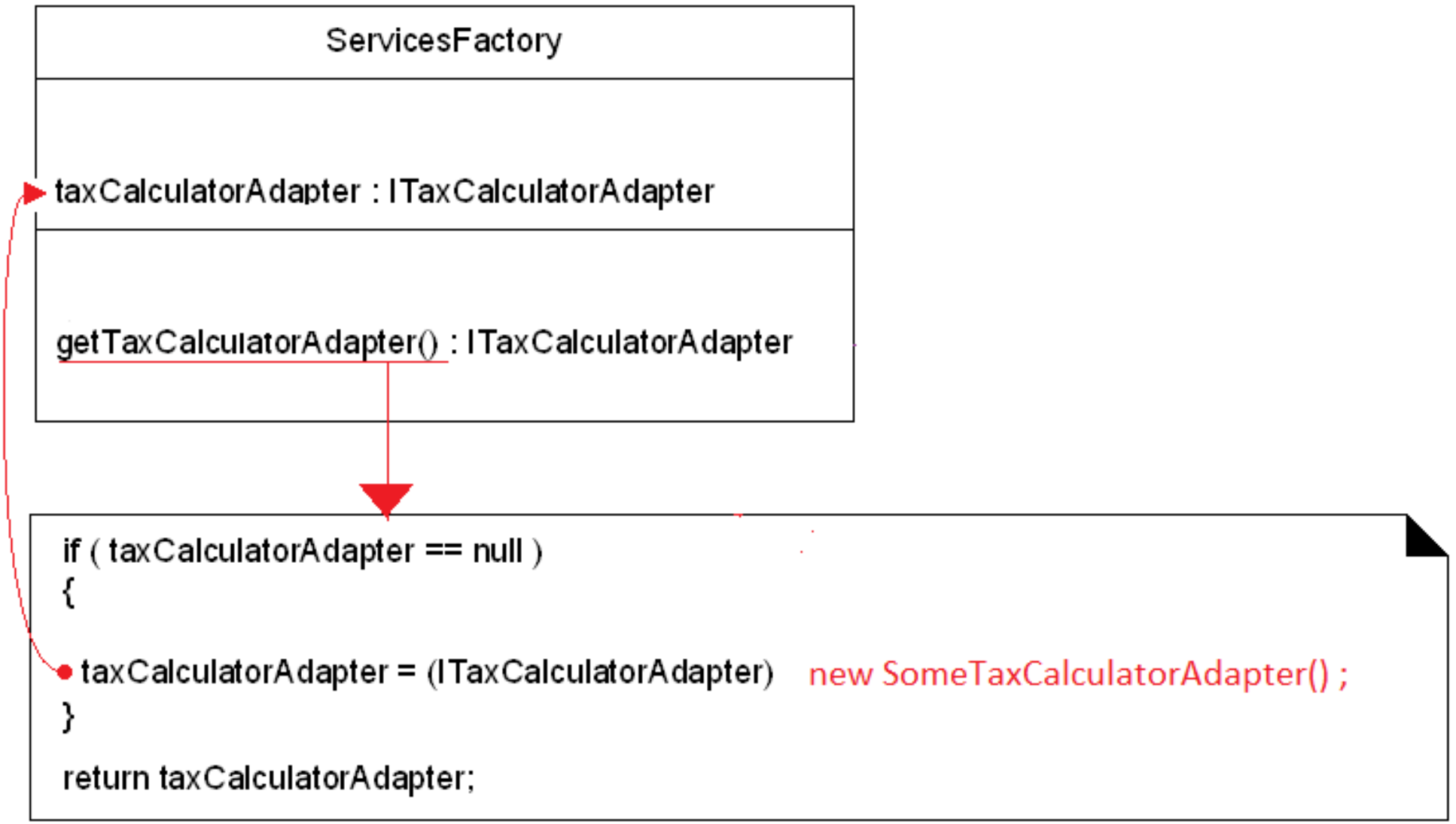


Ćwiczenia: Kod w Java - BuilderExample.pdf - tutaj `Customer` jest jednocześnie nadzorcą, jak można to uogólnić dla wielu nadzorców?

Builder – podsumowanie

- GoF:Builder umożliwia dodawanie nowych typów raportów bez wprowadzania zmian w kodzie: nowa reprezentacja to nowy „budowniczy” (*Open-Close Principle*)
- Wprowadzenie wielu „nadzorców” umożliwi nam łatwe sterowanie treścią raportów (*fullReportDirector, anonymousReportDirector*)

GoF: Factory – zazwyczaj ma jedną instncję (singleton)

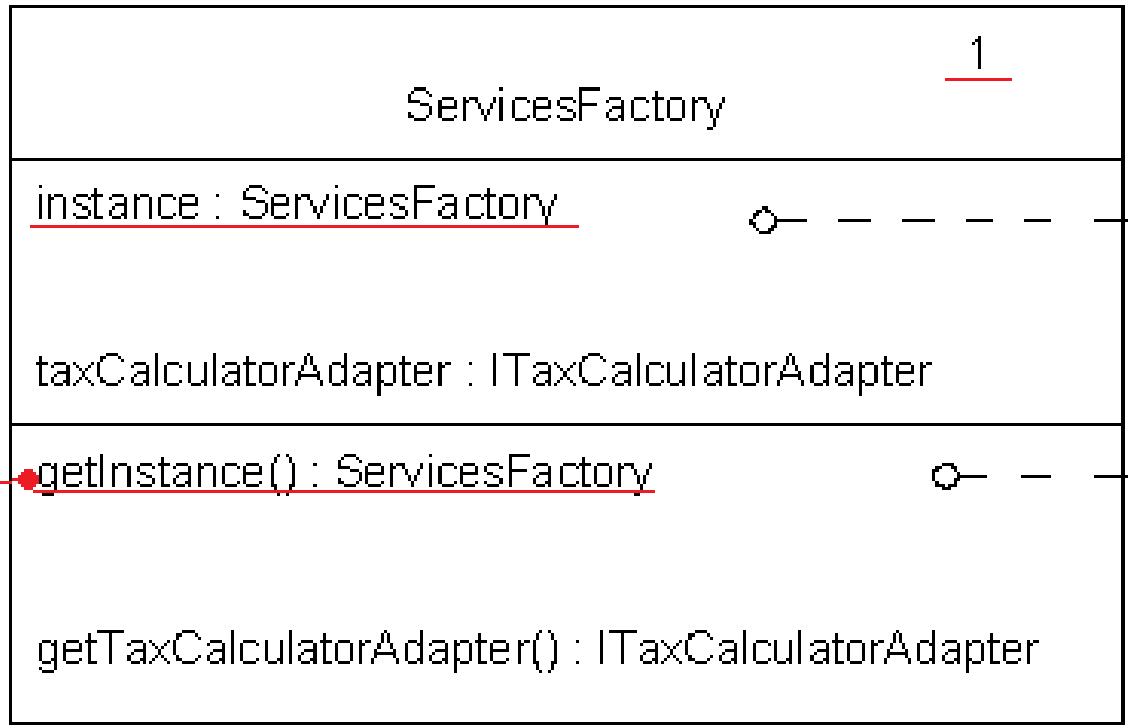


GoF: Singleton – jedna instancja widoczna globalnie

Dlaczego Singleton?

- Potrzeba strategii zarządzania pamięcią (*recykling obiektów, przechowywanie ich w pamięci cache, np. obiekt „pomocy aplikacji”*)
- Kontrola dostępu – metody `ServiceFactory` muszą być dostępne dla wielu wątków (*np. dostęp pliku z logami*)

GoF: Singleton - ma prywatny konstruktor




```
public static synchronized ServicesFactory getInstance()
{
if ( instance == null )
   instance = new ServicesFactory()
return instance
}
```

GoF: Singleton – globalne wywołanie

```
public class Register
{...
  taxAdapter =
        ServicesFactory.getInstance().getTaxAdapter();
```

1. `ServicesFactory` publiczna, ale konstruktor prywatny
2. W dowolnym miejscu w kodzie w celu uzyskania dostępu do instancji klasy możemy napisać

   `ServicesFactory.getInstance()`

3. Następnie możemy do tej instancji wysłać komunikat, czyli mamy globalne wywołanie `getTaxAdapter()`

Singleton vs. Static Class

A może `getTaxAdapter()` jako static?

```
ServicesFactory.getTaxAdapter()
```

vs.

```
ServicesFactory.getInstance().getTaxAdapter()
```

1. W Singleton `getInstance()` jest statyczna, dzięki czemu inne metody mogą być polimorficzne (*override static?*)
2. Singleton jest bardziej uniwersalny (*np. okazuje się, że potrzeba 2 instancji...*)

Singleton vs. Static Class

Ogólna zasada:

Singleton: potrzeba jednego punktu do zarządzania, konfiguracji, jednej instancji klasy na każdą maszynę wirtualną

Static Class: gdy mamy grupę pomocniczych funkcji, które powinny być trzymane razem
(*np. klasy Math, Utils*)

Singleton – inicjalizacja

Zachłanna inicjalizacja

```
static Factory instance = new Factory();

static Factory getInstance() { return instance; }
```

Leniwa inicjalizacja – problem: wielowątkowość

```
public static synchronized Factory getInstance()
{
   if ( instance == null )
            instance = new Factory();

   return instance;
}
```

O przyspieszeniu przez użycie blokady z podwójnym zatwierdzaniem będzie na ćwiczeniach (wiki)

Singleton – triki

Singleton od Java 6

```
public enum ServicesFactory {
    INSTANCE;
    public void foo() { ... }
}
...
ServicesFactory.INSTANCE.foo();
```

- Wada: brak leniwej inicjalizacji
- Zalety: bez zabawy z "synchronized", zabezpiecza przed tworzeniem wielu instancji np. przez refleksję

```
Method pC= Class.getDeclaredMethod("privateConstructor", null);
pC.setAccessible(true);
pC.invoke(...);
```