Technologia Programowania 2017/2018
Wykład 6
Wzorce GoF

Jakub Lemiesz

Wzorce GoF

Kreacyjne

- ✓ Builder
- ✓ Singleton
- ✓ Factory Method
- ✓ Abstract Factory
- Prototype

Strukturalne

- ✓ Adapter
- ✓ Decorator
- ✓ Proxy
- ✓ Facade
- ⇒ Composite
- ⇒ Flyweight
- Bridge

Czynnościowe

- ✓ State
- ⇒ Strategy
- ⇒ Template Method
- ⇒ Command
- ⇒ Mediator
- ⇒ Observer +MVC
- Interpreter
- Iterator
- Visitor
- Memento

Wzorce GoF w standardowych bibliotekach Javy

http://stackoverflow.com/questions/1673841/examples-of-gof-design-patterns-in-javas-core-libraries

2221

You can find an overview of a lot of design patterns in Wikipedia. It also mentions which patterns are mentioned by GoF. I'll sum them up here and try to assign as many pattern implementations as possible, found in both the Java SE and Java EE APIs.

+250

Creational patterns

Abstract factory (recognizeable by creational methods returning the factory itself which in turn can be used to create another abstract/interface type)

- `javax.xml.parsers.DocumentBuilderFactory#newInstance()`
- `javax.xml.transform.TransformerFactory#newInstance()`
- `javax.xml.xpath.XPathFactory#newInstance()`

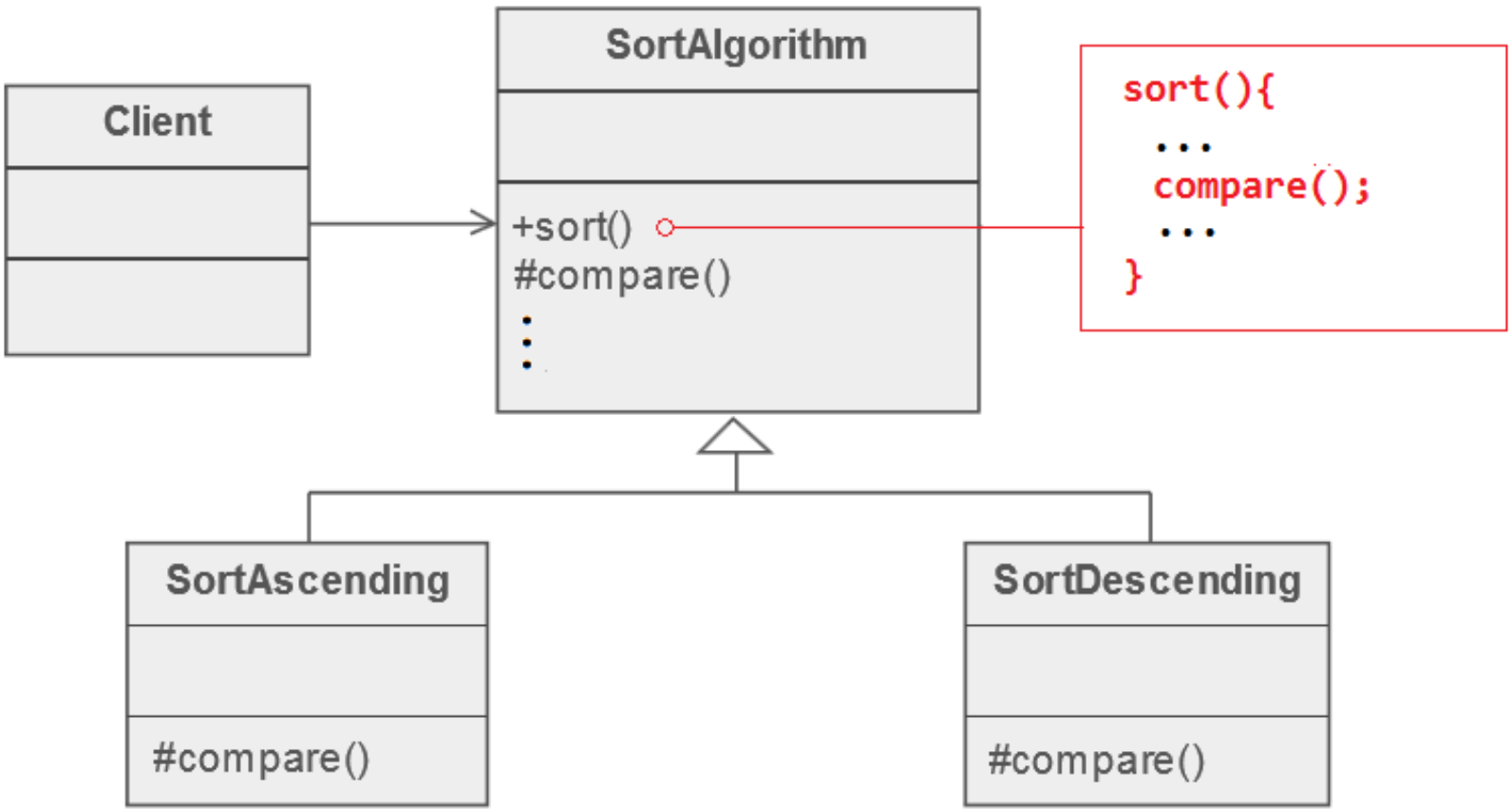
Builder (recognizeable by creational methods returning the instance itself)

- `java.lang.StringBuilder#append()` (unsynchronized)

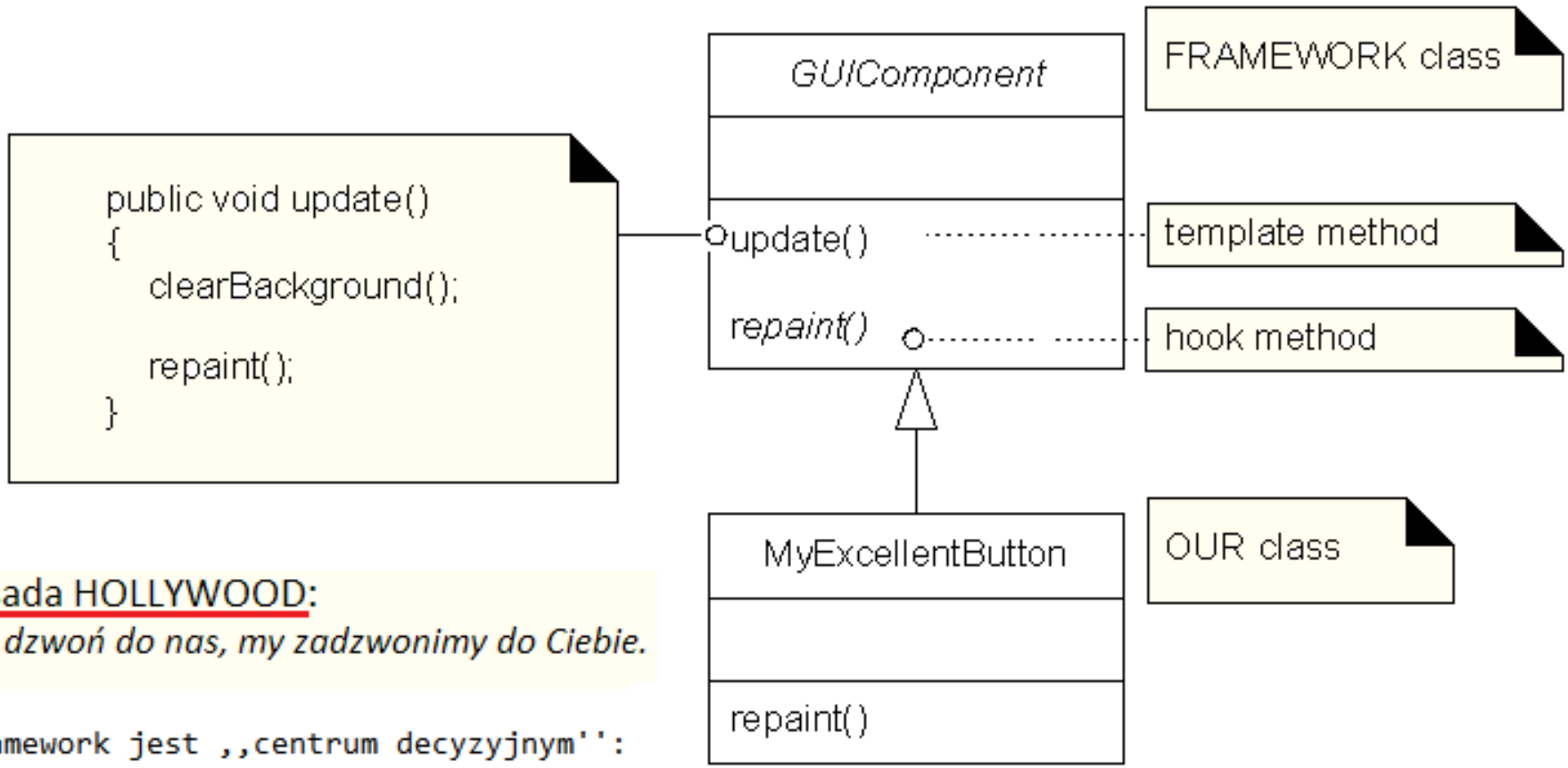
GoF: Template Method

- Jak zmieniać wybrane kroki standardowego algorytmu w zgodzie z „open-close principle”?
- Zdefiniuj w abstrakcyjnej nadklasie metodę szablonową będącą szkieletem algorytmu, która jest niezmienna i nie powinna być przesłaniana w podklasach
- W ramach metody szablonowej wywołuj metody-haczyki (ang. hook method), reprezentujące zmienne kroki algorytmu, które są zdefiniowane w podklasach
- Typowo: szablonowa – public, haczyki – protected

Template Method – przykład



Template Method – tak działają frameworki



Zasada HOLLYWOOD:

Nie dzwoń do nas, my zadzwonimy do Ciebie.

Framework jest ,,centrum decyzyjnym'': metoda repaint() jest wywoływana przez framework w ramach metody update()

Template Method – przykłady

- JUnit 3: użytkownik może zdefiniować setUp, test i tearDown ale nie może zmienić metody run, w której są wywoływane w ustalonej kolejności

- ```
  Thread thread = new Thread() {
      @Override
      public void run() {...}
  };
  thread.start();
  ```

- GoF:Factory Method z ostatniego wykładu to szczególny przypadek Template Method
  ```

GoF: Strategy

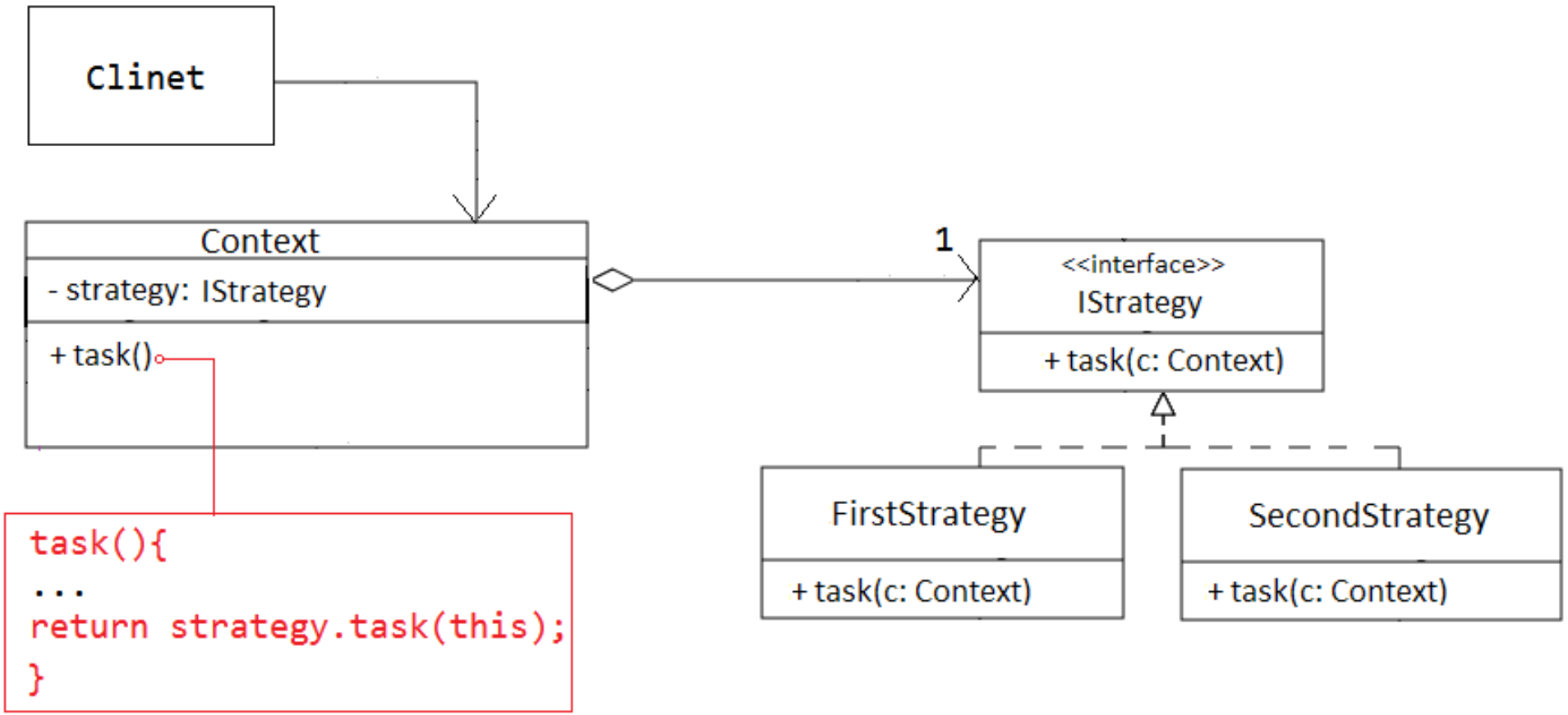
Przykład: w sklepie przy wyliczaniu całkowitej należności metodą `getTotal()` mamy 10 możliwych algorytmów naliczania upustów i 3 algorytmy naliczania podatków

Pytanie: załóżmy, że zniżek nie można łączyć i w ramach jednego zakupu płaci się tylko jeden podatek. Ile jest możliwyc kombinacji? Ile stworzymy klas używając Template Method?

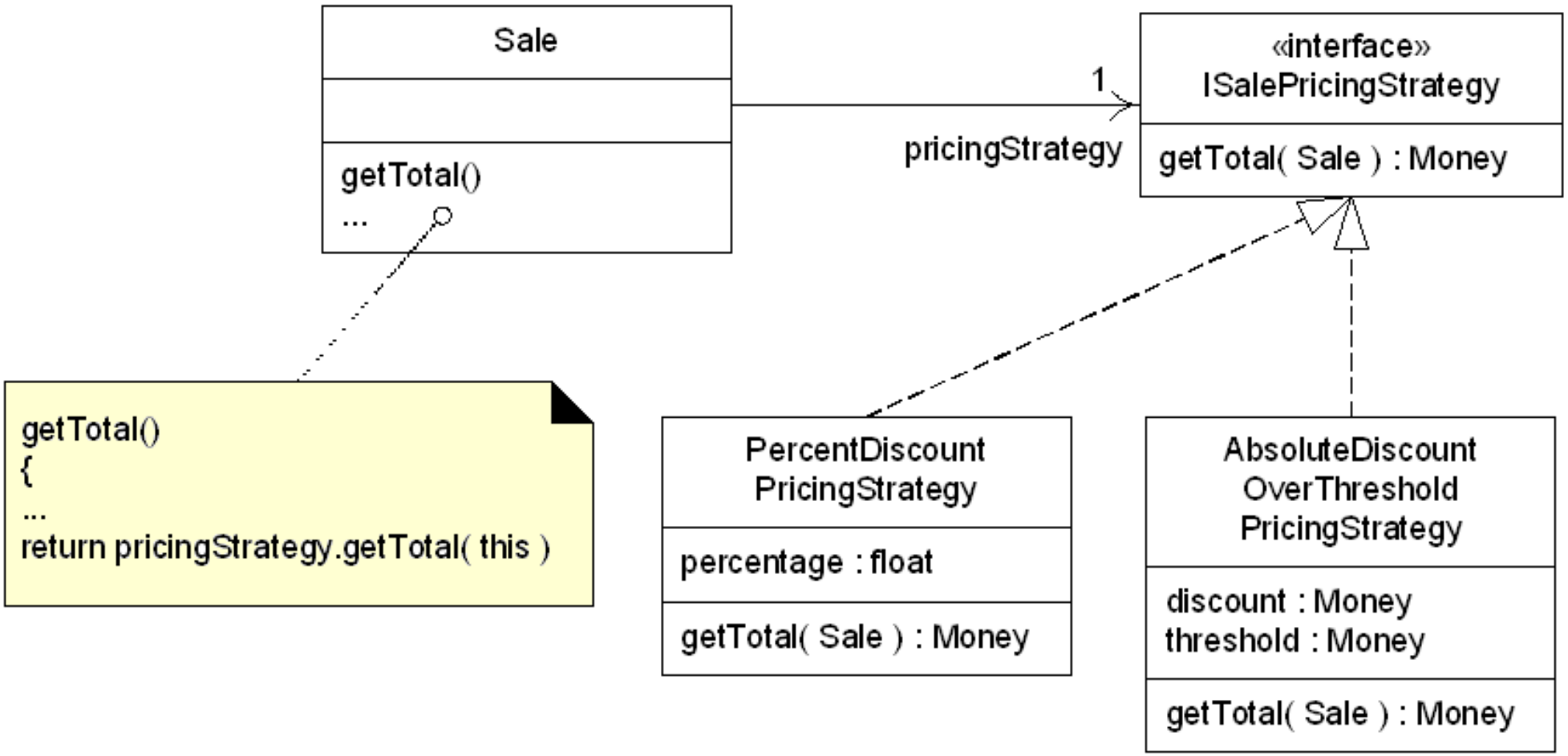
Problem: co zrobić, jeśli mamy wiele powiązanych algorytmów, których używamy zamiennie?

Rozwiązanie: zdefiniuj każdy algorytm w osobnej klasie, klasy te powinny mieć wspólny interfejs. Odwołuj się do algorytmów za pośrednictwem "wymiennych" obiektów.

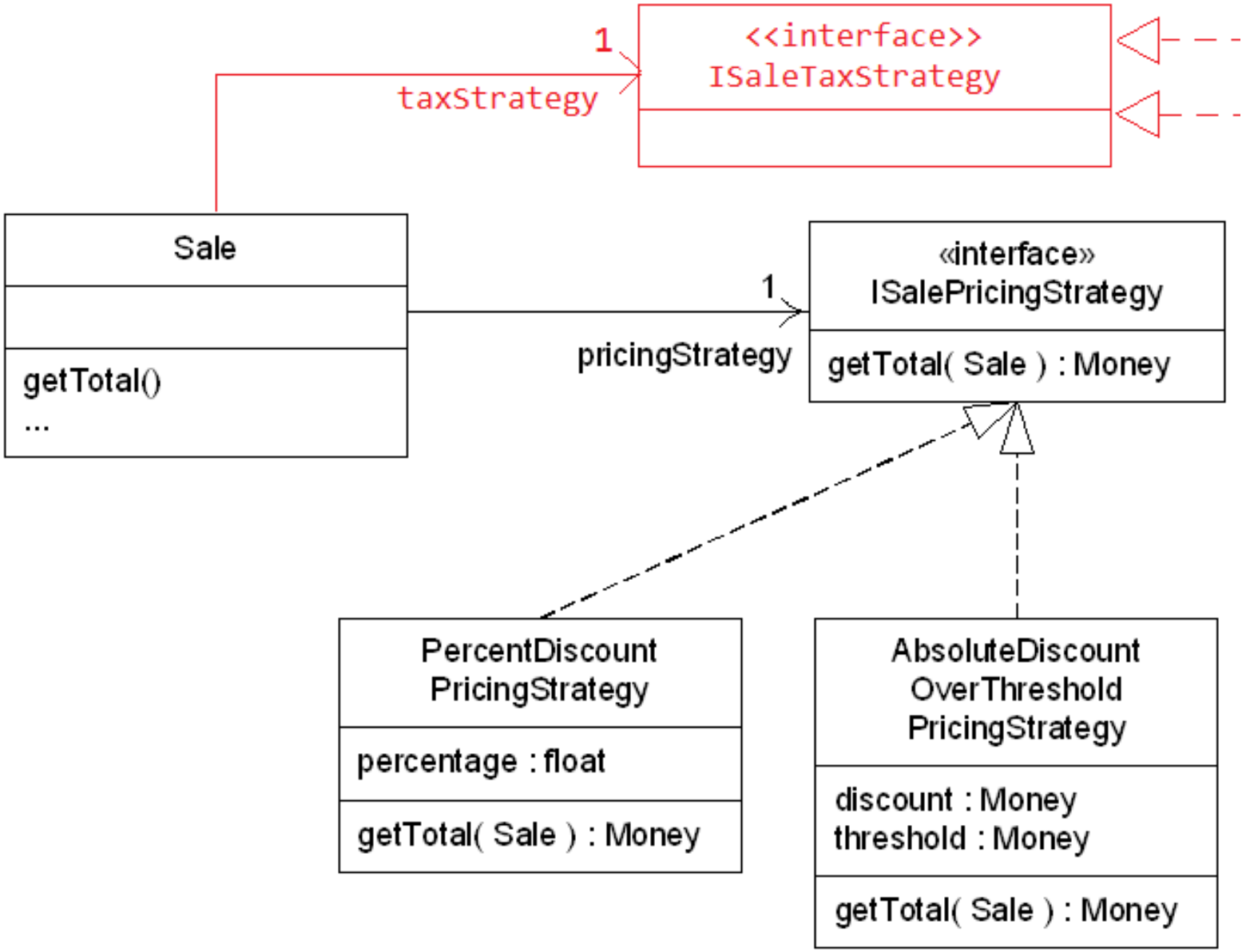
GoF:Strategy – podstawowy schemat



Strategy — przykład



Strategy — przykład: wiele strategii



Strategy – podsumowanie

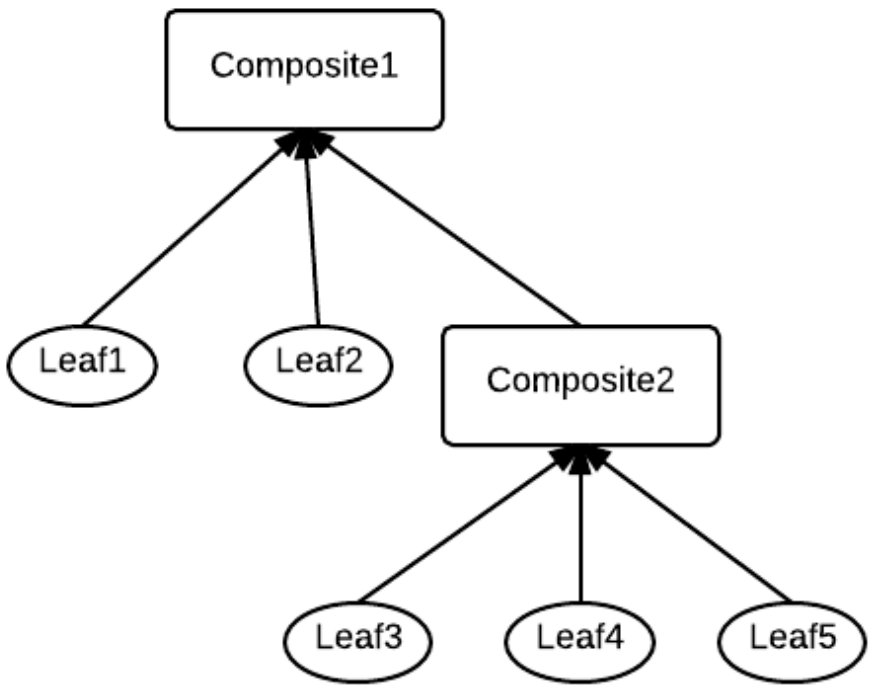
- Modyfikujemy klasę nie przez przesłanianie, ale przez zmianę zawieranego obiektu (*podobne do GoF:State*)
- Korzyści: wspiera open-close principle i mamy mniej klas niż w Template Method (*ile dla przykładu ze sprzedażą?*)
- Low Coupling dzięki Dependency Inversion

GoF: Composite

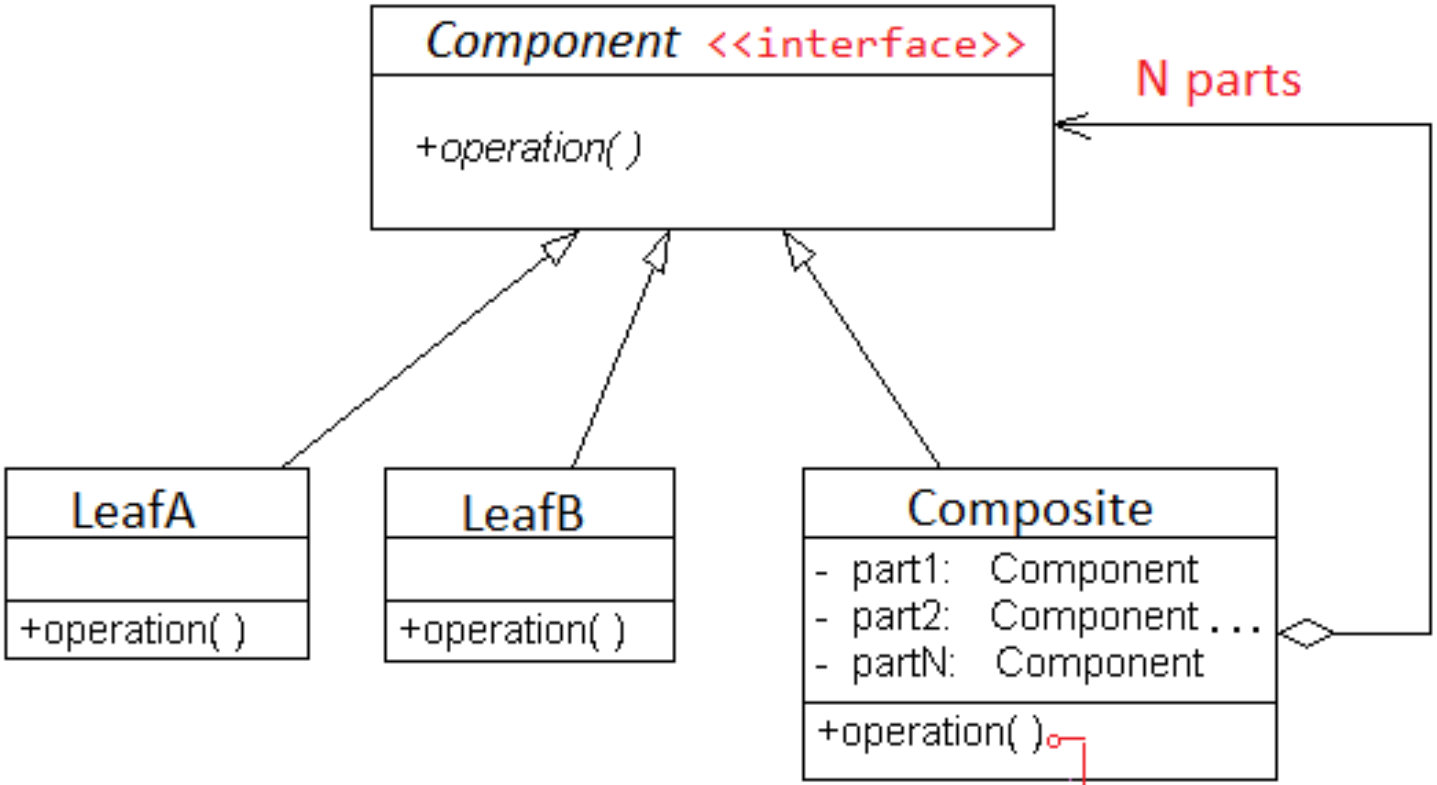
A co jeśli upusty można łączyć w ramach jednego zakupu?

Jak zdefiniować strukturę klas by byty proste (*np. upusty*) i złożone (*np. upusty łączone*) traktować w jednolity sposób?

Przykład: operacje copy, move, delete w systemie plików (*liście to pliki, węzły wewnętrzn to katalogi*)



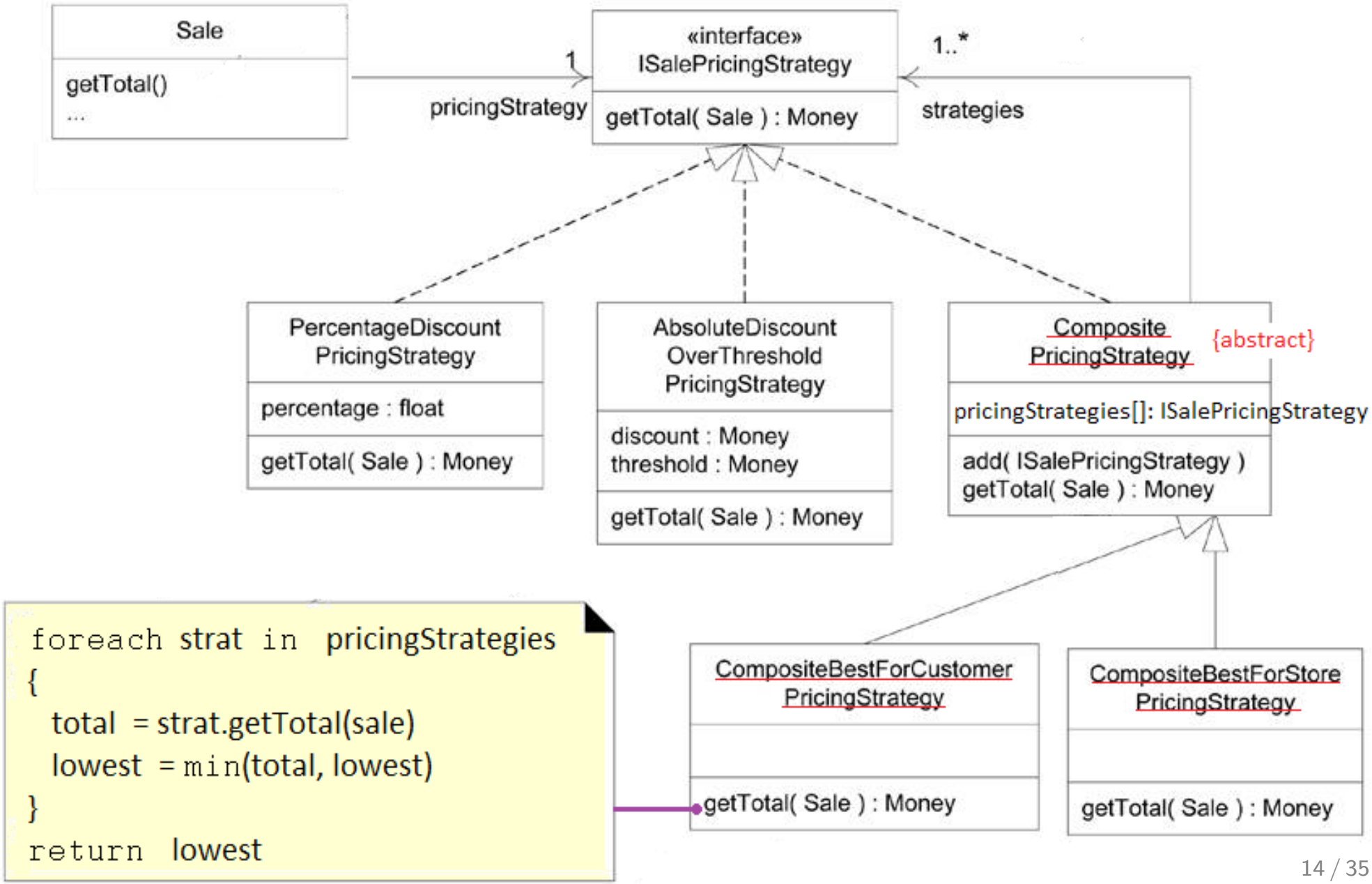
GoF: Composite




```
operation(){
        part1.operation();
        part2.operation();
        ...
        partN.operation();
}
```

Uwaga: częściami w Composite mogą być liście albo inne kompozyty

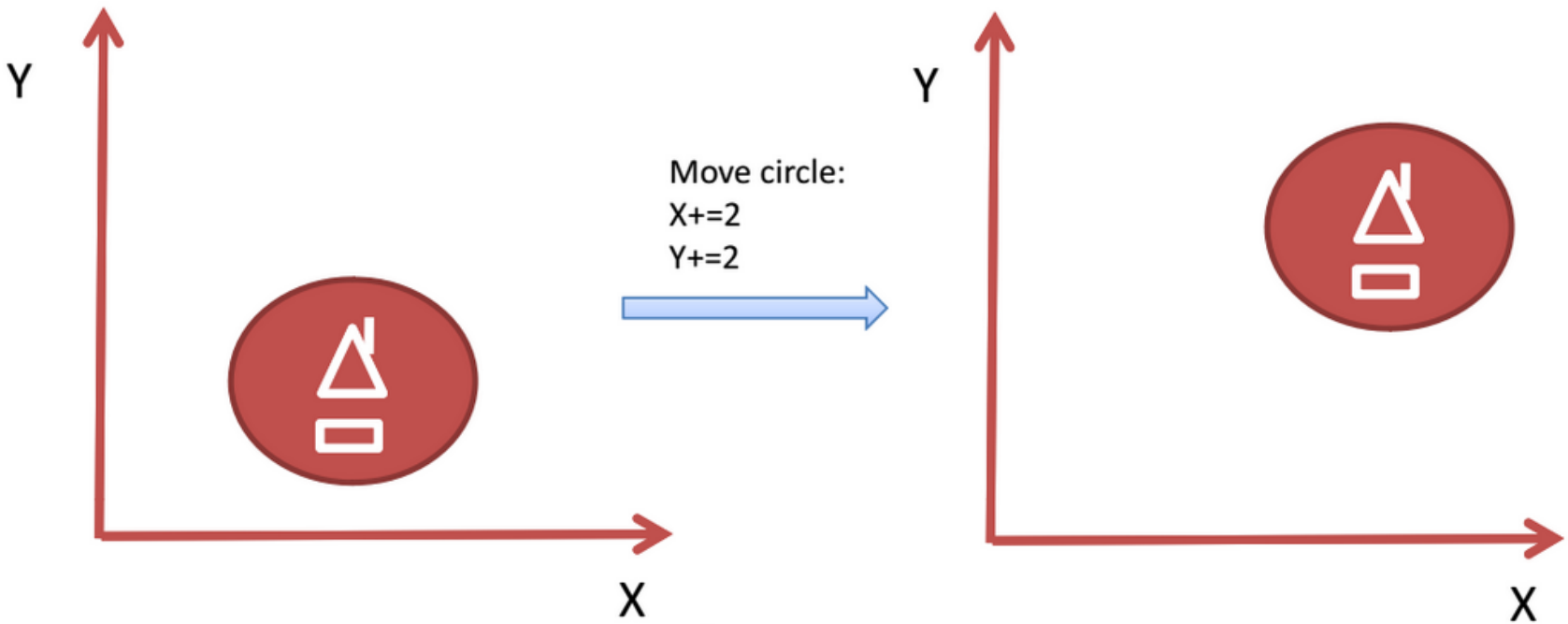
Composite — przykład z upustami



Composite – zadanie na lab

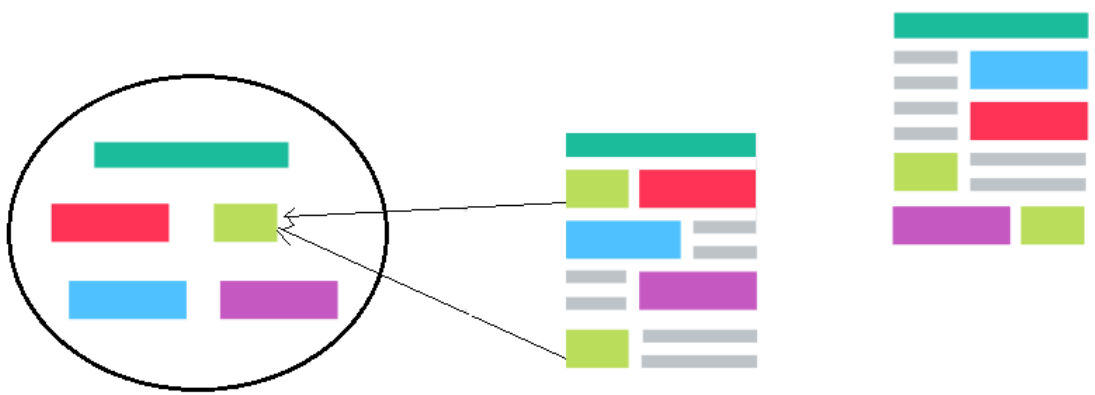
Program do animacji 2D: Rectangle, Circle, Traingle mogą zawierać dowolną liczbę figur, Line nie może niczego zawierać.

Figury mają współrzędne (x,y) i mamy metodę move ...



GoF: Flyweight, czyli waga musza

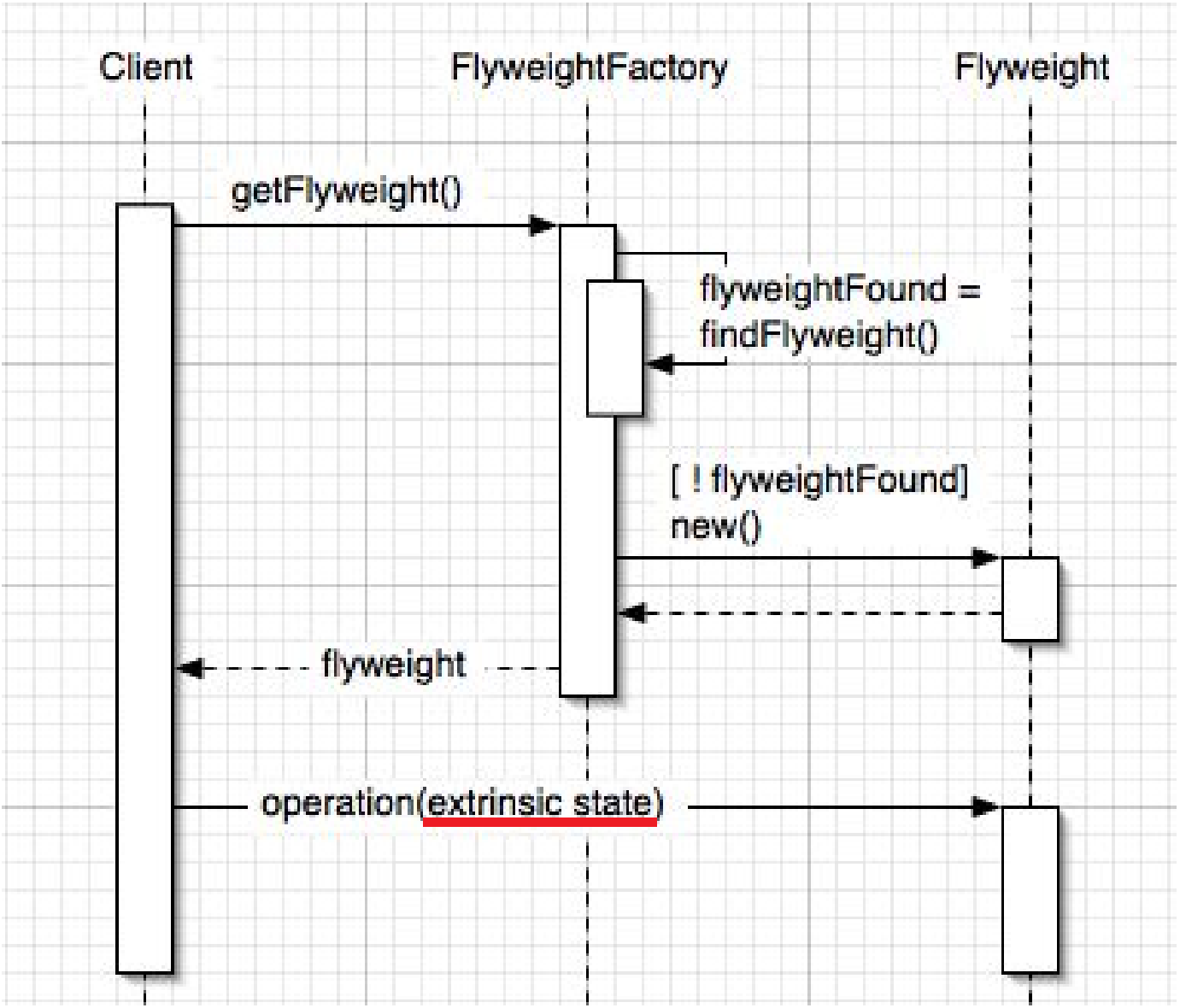
1. Problem: jak efektywnie zarządzać dużą liczbą obiektów?
2. Przykład: ten sam obraz może występować w wielu miejscach na danej stronie www, ale nie ma sensu pobierać ani przechowywać go wielokrotnie
3. Rozwiązanie:
   - przy ładowaniu strony każdy nowo napotkany obraz jest ładowany do cache przeglądarki
   - jeśli obraz pojawił się wcześniej tworzony jest jedynie lekki obiekt z unikalnymi danymi (*pozycja, rozmiar*) i z referencją do już pobranego obrazu



GoF: Flyweight, czyli waga musza

1. Wydzielamy cześć zewnętrzną obiektu, która jest unikatowa/zależna od stanu (*extrinsic part*) i wewnętrzną wspólną/niezależną od stanu (*extrinsic part*)
2. Cześć wewnętrzna (wspólna) dla danej grupy obiektów będzie się znajdować w jednym obiekcie „flyweight”
3. Dla efektywności obiekty „flyweight” są zazwyczaj tworzone i przechowywane w klasie fabryki
4. Klasa klienta odpowiada za dostarczenie niezbędnych informacji z części zewnętrznej (unikatowej)

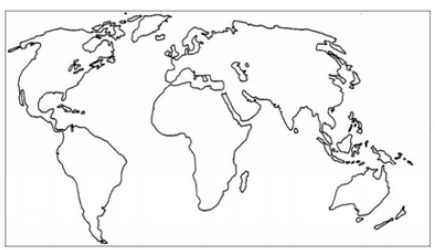
Flywieght – diagram sekwencji

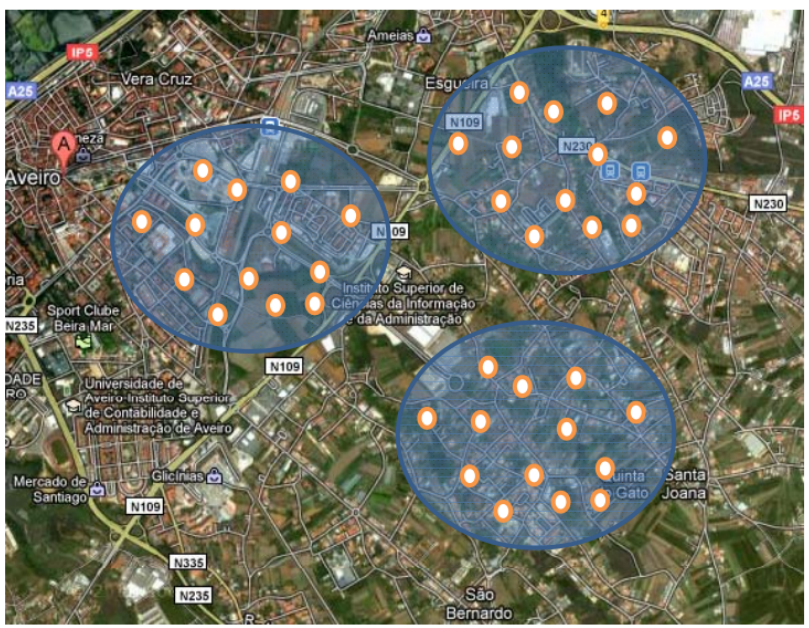


Źródło diagramu

GoF: Flywieght, czyli waga musza

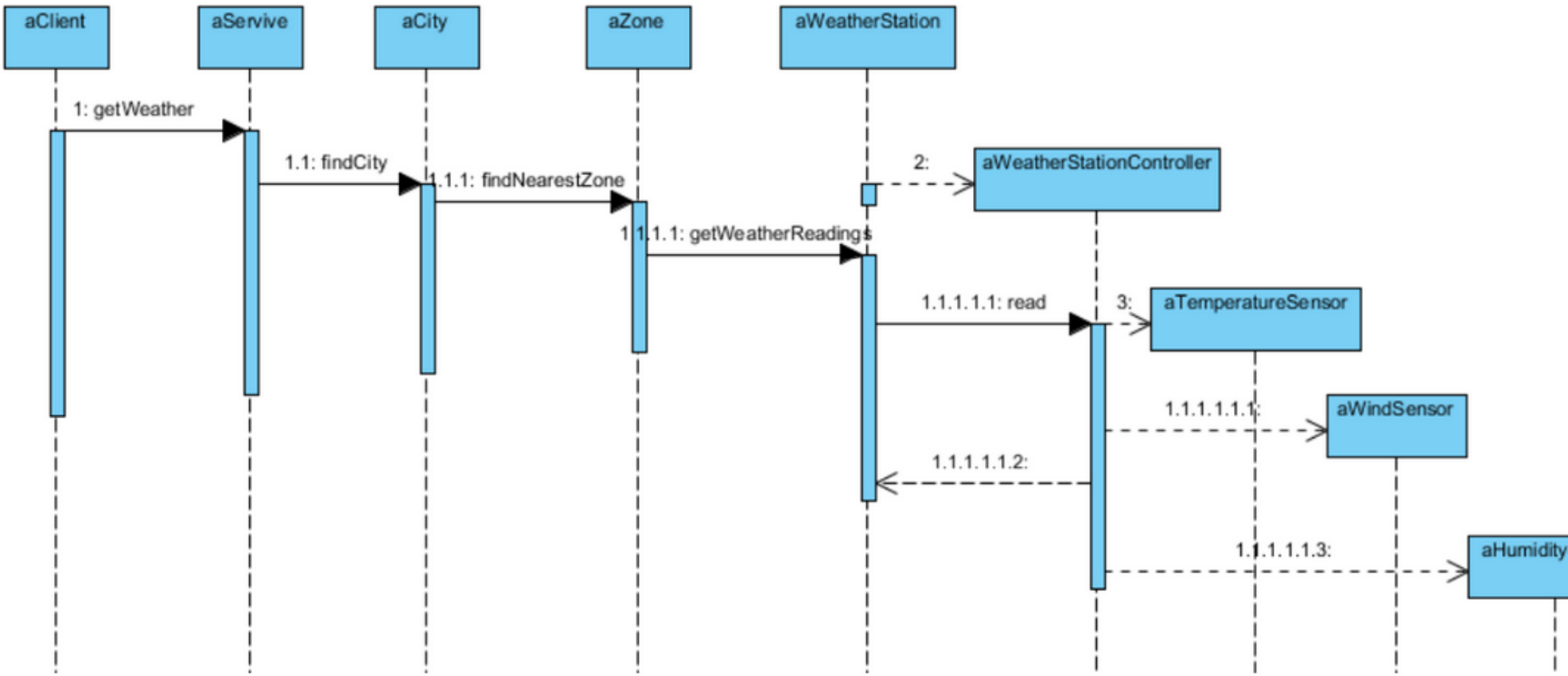
- Przykład Pereiry: serwis www udostępniający wyniki pomiarów sensorów rozsianych po świecie
- Trzy sensory (temperatura, wilgotność, wiatr) są zawsze instalowane razem i tworzą stację pogodową
- Stacje pogodowe są pogrupowane w strefy



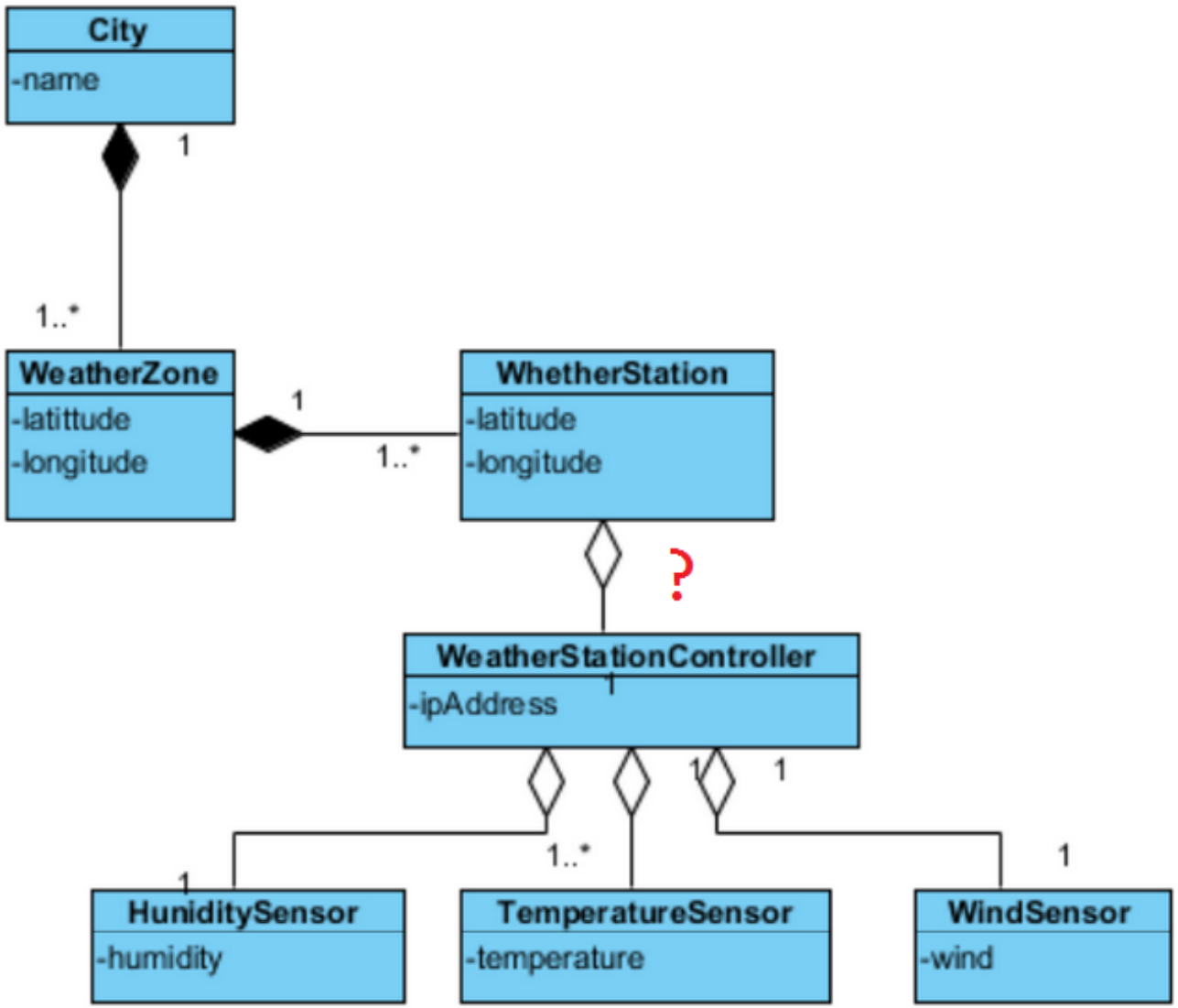


Flyweight: diagram sekwencji w przykładzie

`http://allweatherservice.com/aveiro?la=037=&lo=0838`

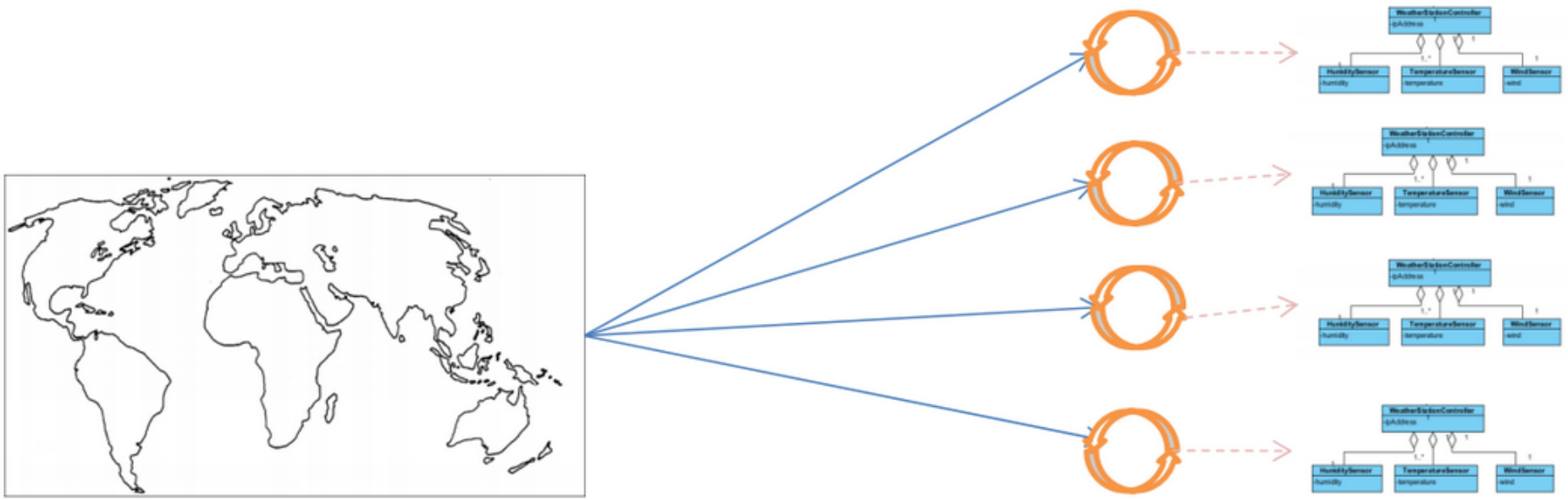


Flywieght: diagram klas w przykładzie



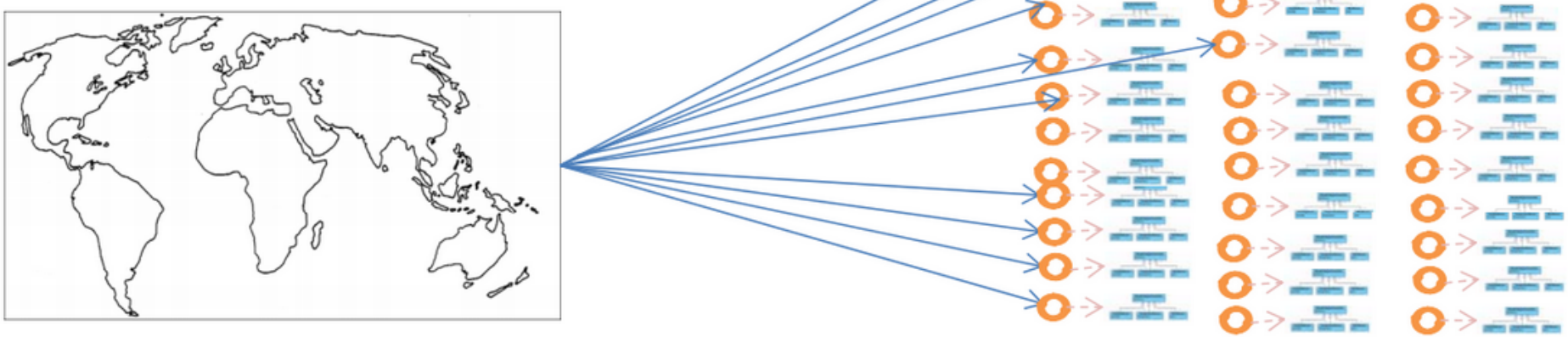
Flyweight, czyli waga musza

Dla każdego klienta tworzymy wątek i instancję kontrolera



Flyweight, czyli waga musza

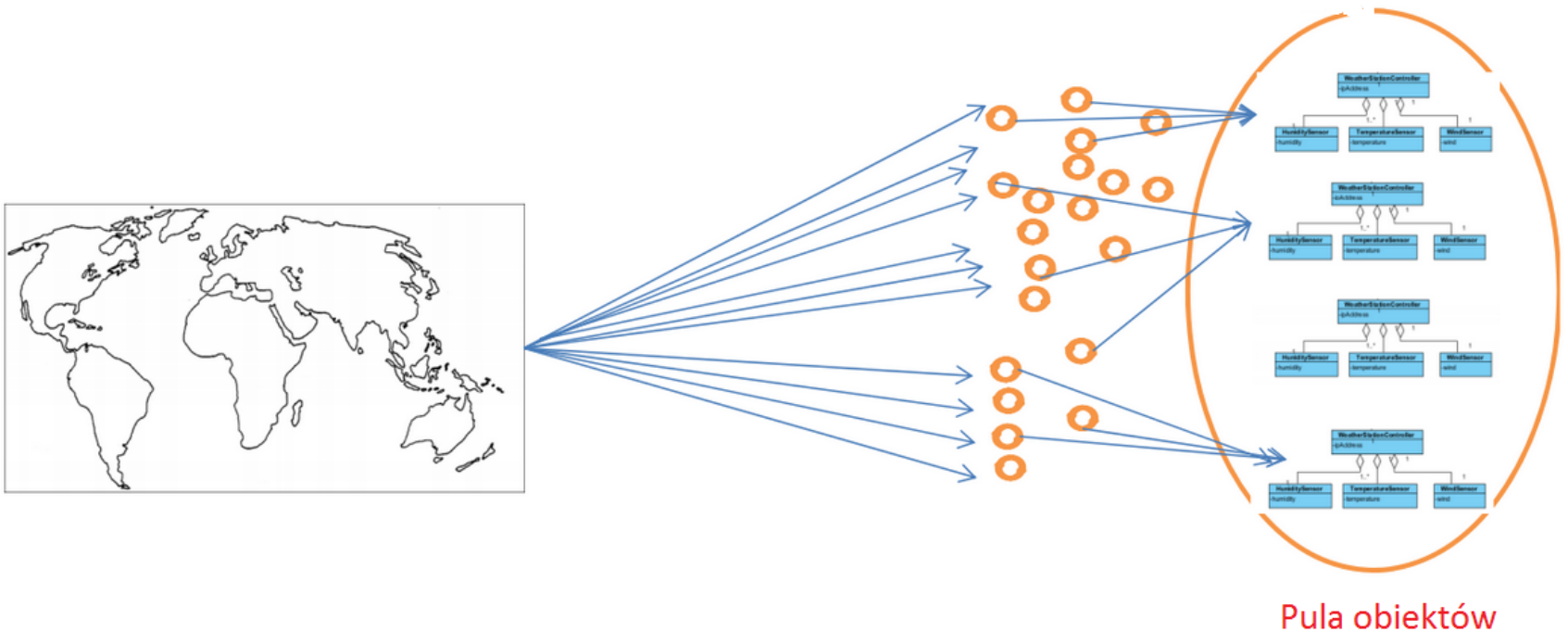
Instancja ma mieć jedynie niezbędne informacje, bo w jednej chwili klientów może być dużo...
(*zużycie pamięci sprawdzimy za pomocą JMater*)



Flywieght – dwa główne pomysły

1) Wspólne dane wielu instancji pamiętamy raz (*np. informacji o położeniu geograficznym*)

2) Fabryka odpowiedzialna odpowiedzialna za tworzenie i zarządzanie pulą obiektów



Pula obiektów

```
class WeatherStationControllerFactory  {

  Map<String, WeatStatContr> controllers;//String to adres

  WeatStatContr getController(String ipAddress)
  {
    WeatStatContr c = controllers.get(ipAddress);
    if (c == null) {
        c = new WeatStatContr( ...);
        controllers.put(ipAddress, c);
    }
    return c;
```

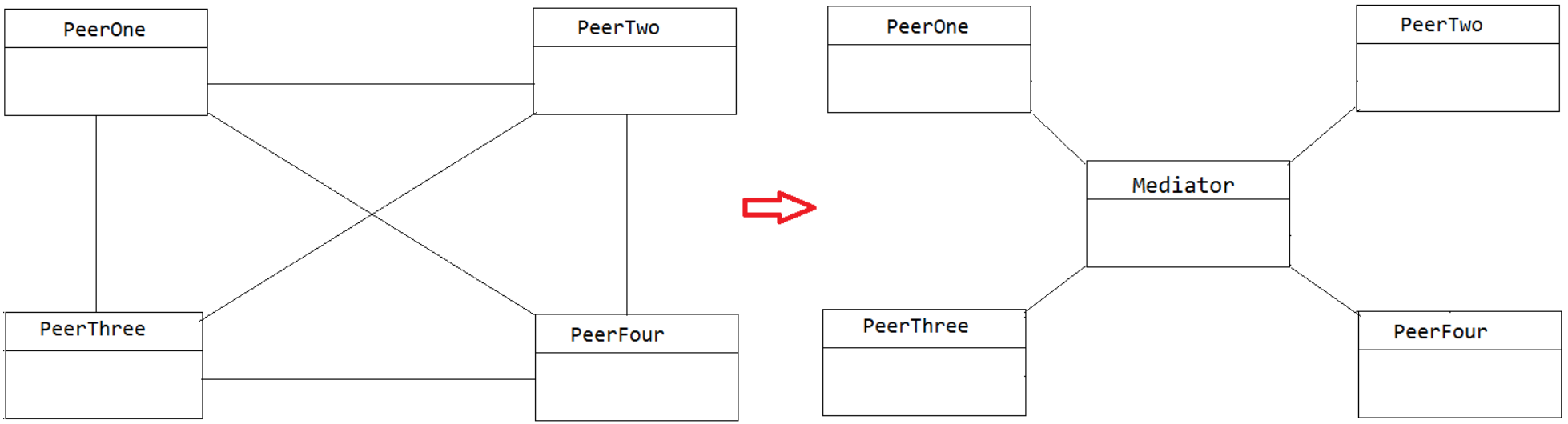
Przykładowo w kliencie

```
WeatStat station = city.findNearestStation(lat, long);
String ip = station.getIpAddress();
... = WeatStatContrFactory.getIns().getController(ip);
```

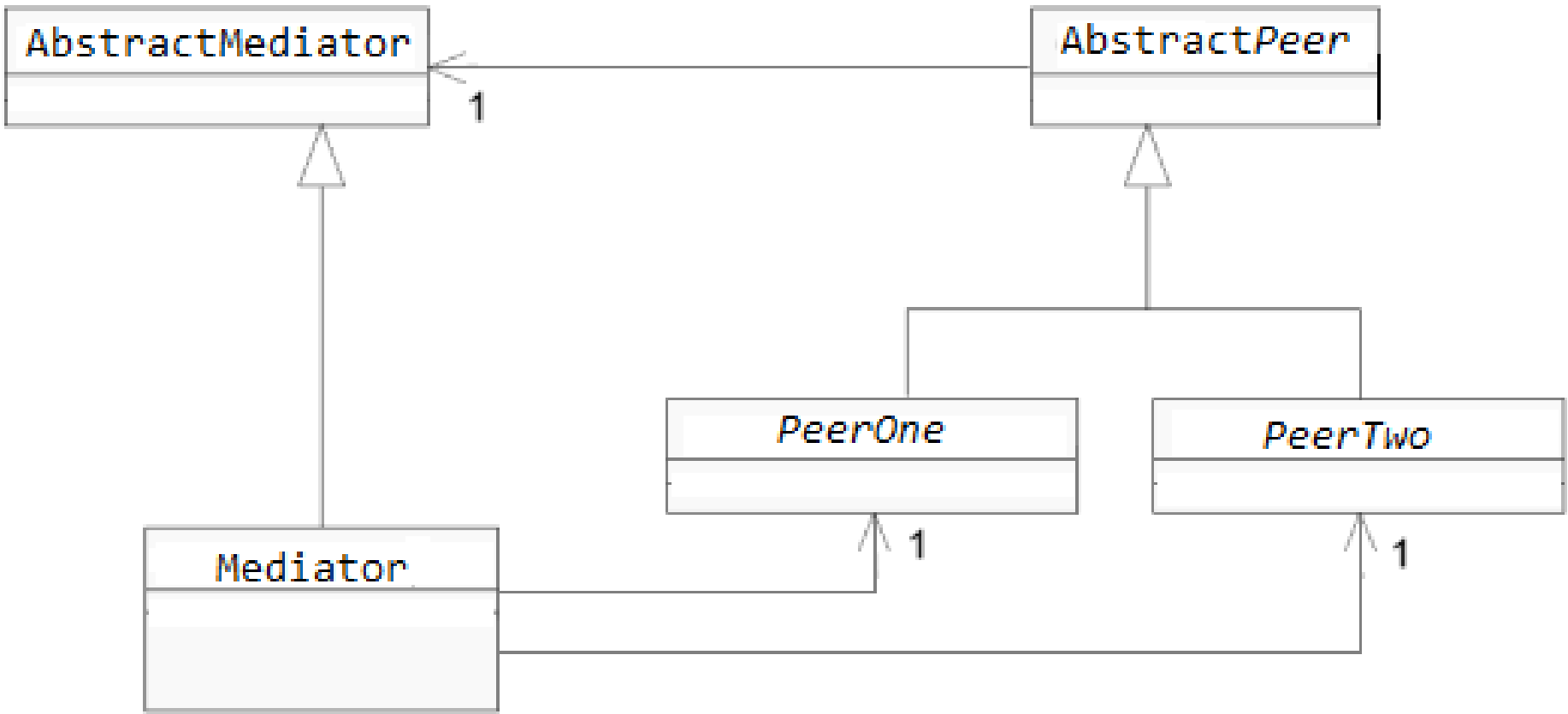
GoF: Mediator

Problem: wysokie sprzężenie między obiektami, mamy zależności 'wiele-do-wielu'

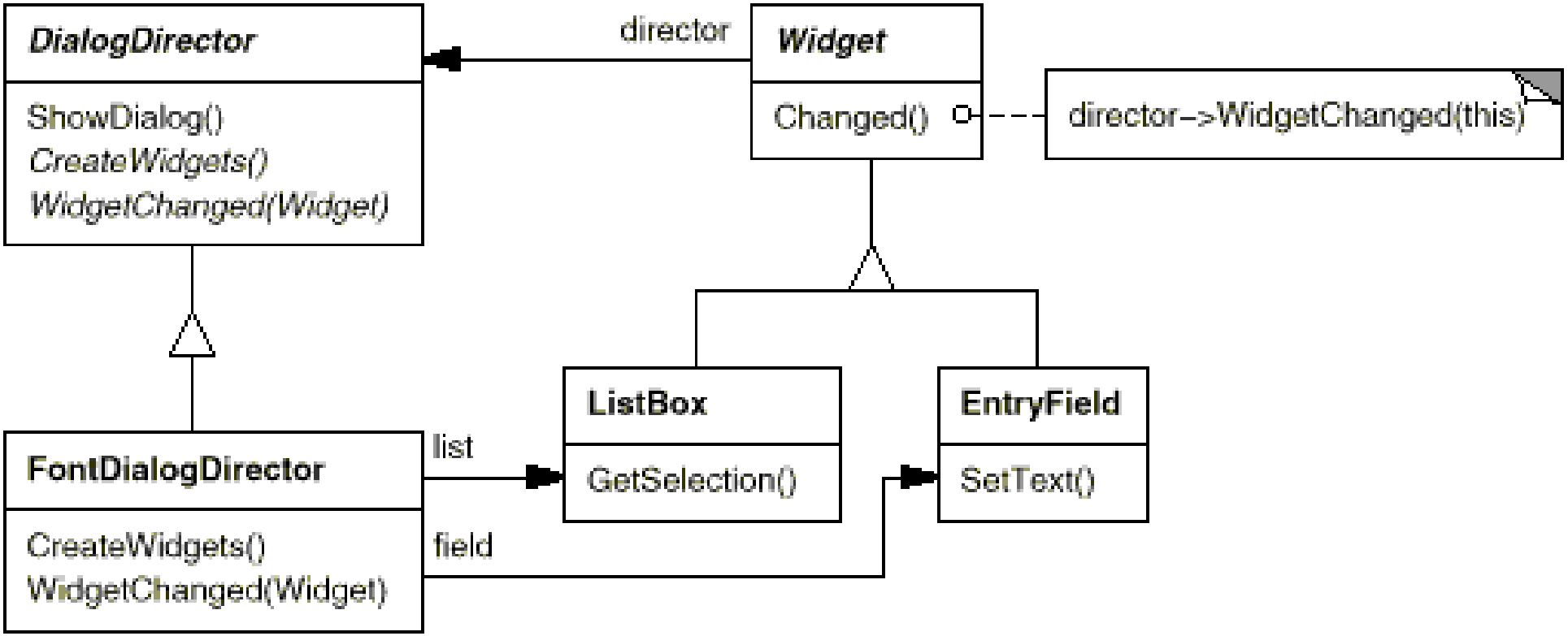
Rozwiązanie: definiujemy nowy obiekt opisujący jak grupa obiektów współpracuje. Obiekty zależą jedynie od mediatora (*np. kontrolera w MVC*)



GoF:Mediator – podstawowy schemat

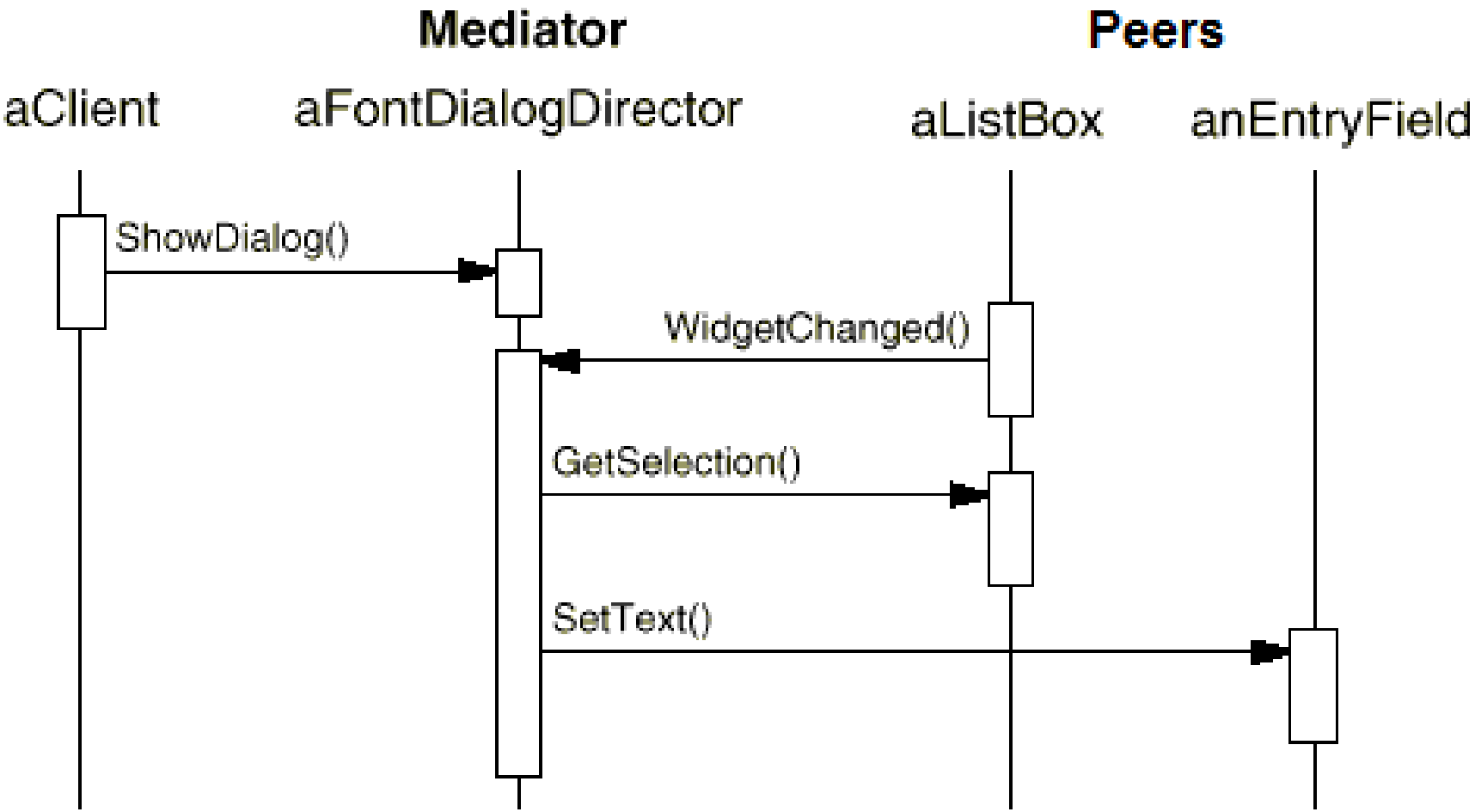


Mediator – przykład z ksiazki GoF (uwaga: inne oznaczenia)



Okienko dialogowe jest mediatorem dla widgetów, które zawiera i które są ze sobą powiązane na różne sposoby (*np. wybór z list box ustawia entry field*)

Mediator – przykład z ksiazki GoF (uwaga: inne oznaczenia)



GoF:Mediator – komunikacja

Komunikacja między Mediator a Peers może być zrealizowana na kilka sposobów

1. zwyczajna delegacja zadań za pomocą dedykowanych metod publicznych w mediatorze
2. zgodnie z wzorcem Observer (*jak w MVC*)
3. wiadomości asynchroniczne (*będzie przy Akka*)

GoF:Mediator – zalety

- Ułatwia zarządzanie logiką komunikacji, która zostaje zamknięta w jednej klasie (*redukuje relacje 'wiele-do-wielu' na 'jeden-do-wielu'*)
- Zmniejsza sprzężenie między klasami *Peers* (*łatwe przenoszenie i tworzenie nowych klas*)
- Wzorzec Mediator jest podobny do Facade, ale tam komunikacja w jedną stronę

GoF:Observer, czyli związek podmiot-obserwator

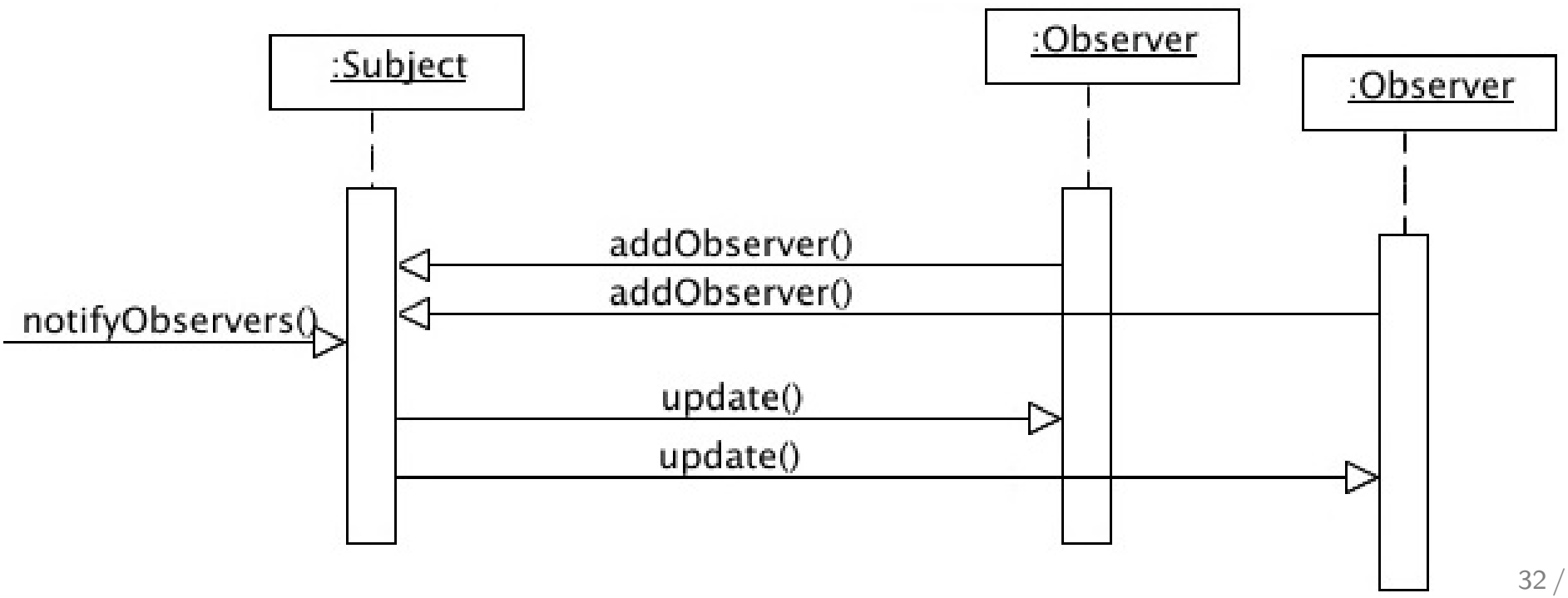
Problem:

1. Wiele różnych obiektów-obserwatorów jest zainteresowanych „na bieżąco” zmianami zachodzącymi w obiekcie-podmiocie
2. Obserwatorzy mogą stracić zainteresowanie...
3. Low Coupling: obiekt-podmiot powinien mieć mało powiązań z obiektami-obserwatorami

GoF: Observer – schemat komunikacji

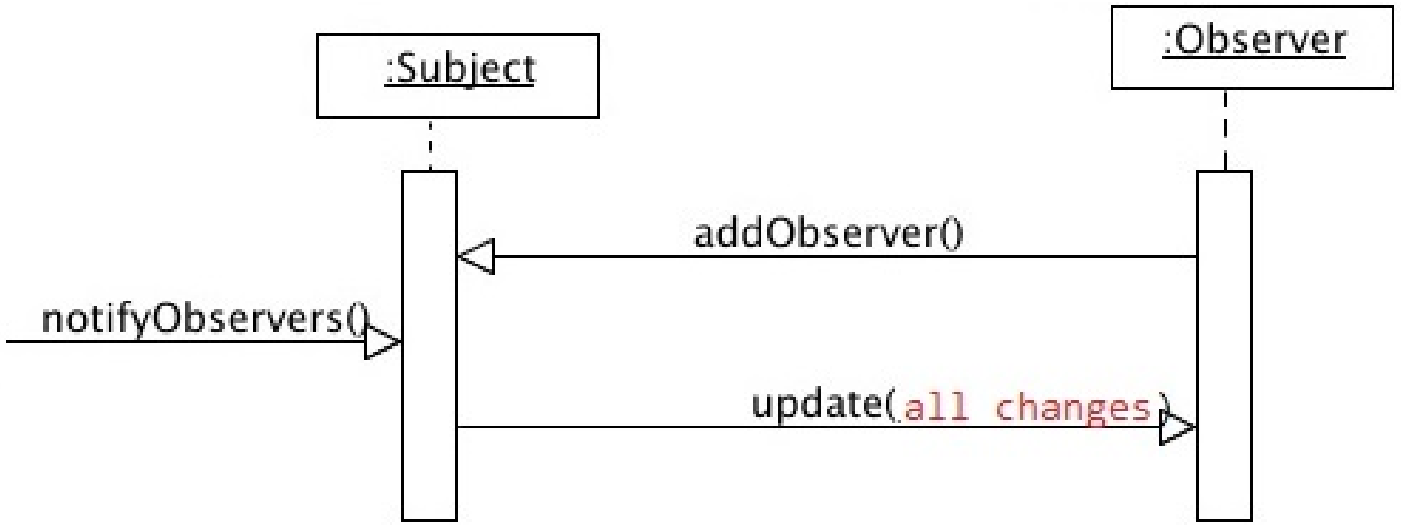
Rozwiązanie:

1. Definiujemy interfejs obserwatora
2. Obserwatorzy są rejestrowani na liście podmiotu
3. Podmiot informuje obserwatorów o zdarzeniach

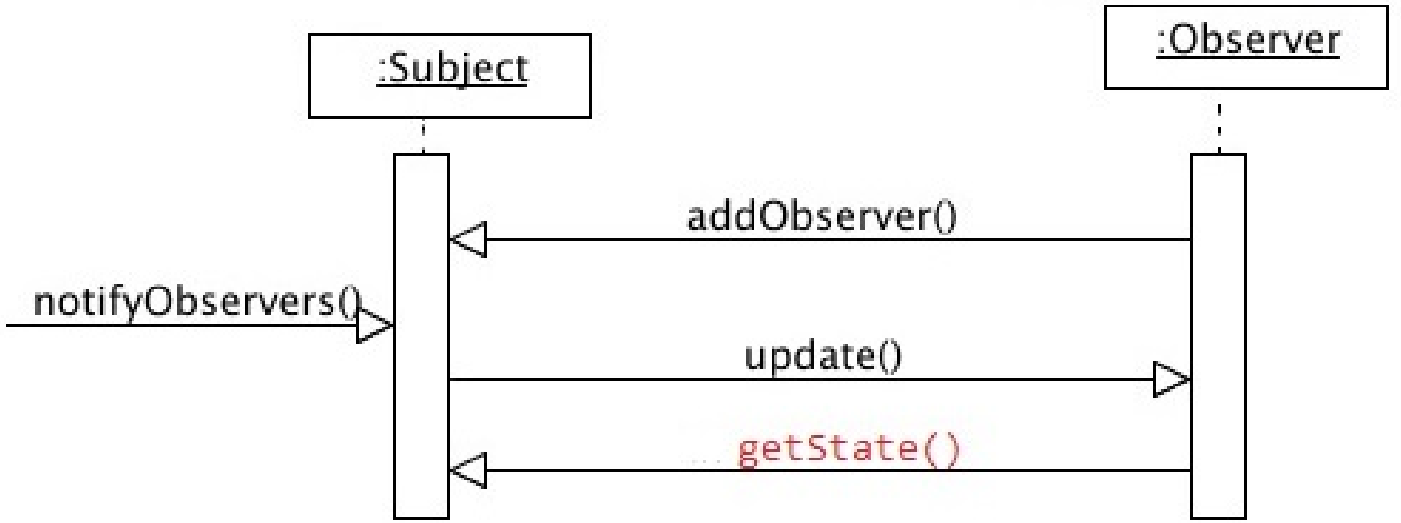


GoF:Observer – push vs. pull

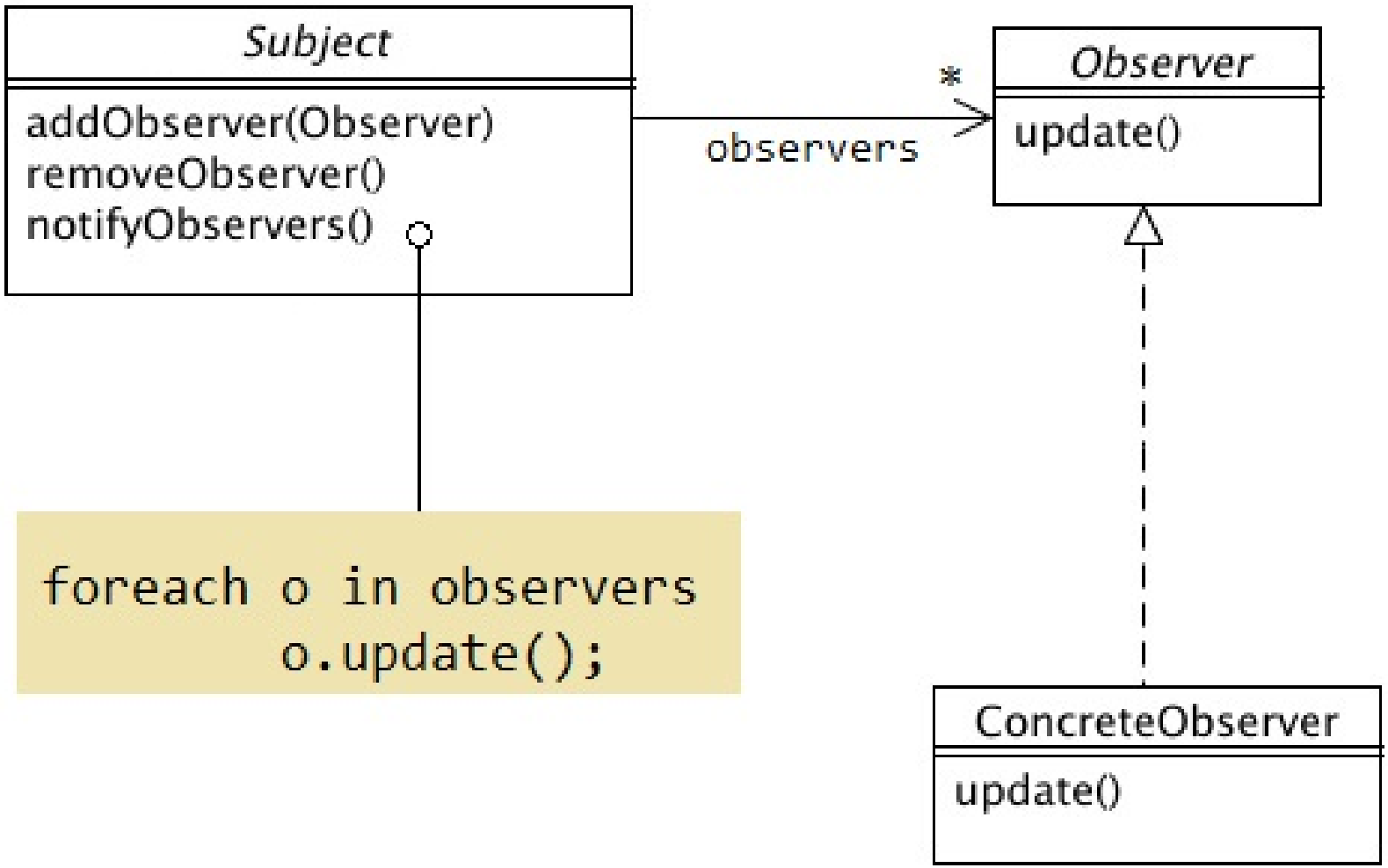
push – *podmiot wysyła wszystkie informacje*



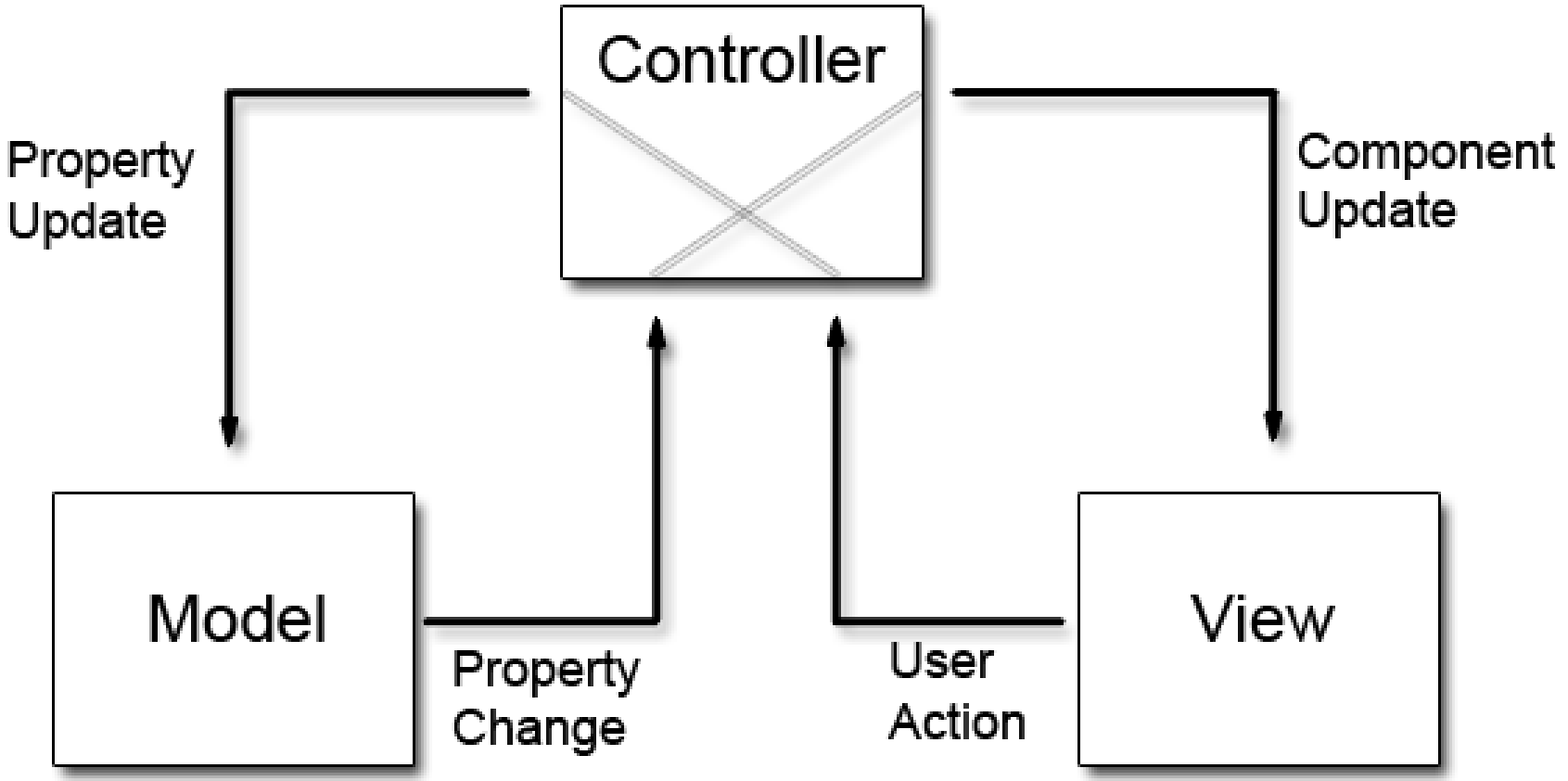
pull – *obserwator wywołuje publiczne metody podmiotu*



GoF:Observer – podmiot ma listę obserwatorów



GoF:Observer – na przykładzie MVC



Przyjrzyj się przykładowi MVC z refleksją,
który omówimy w przyszłym tygodniu