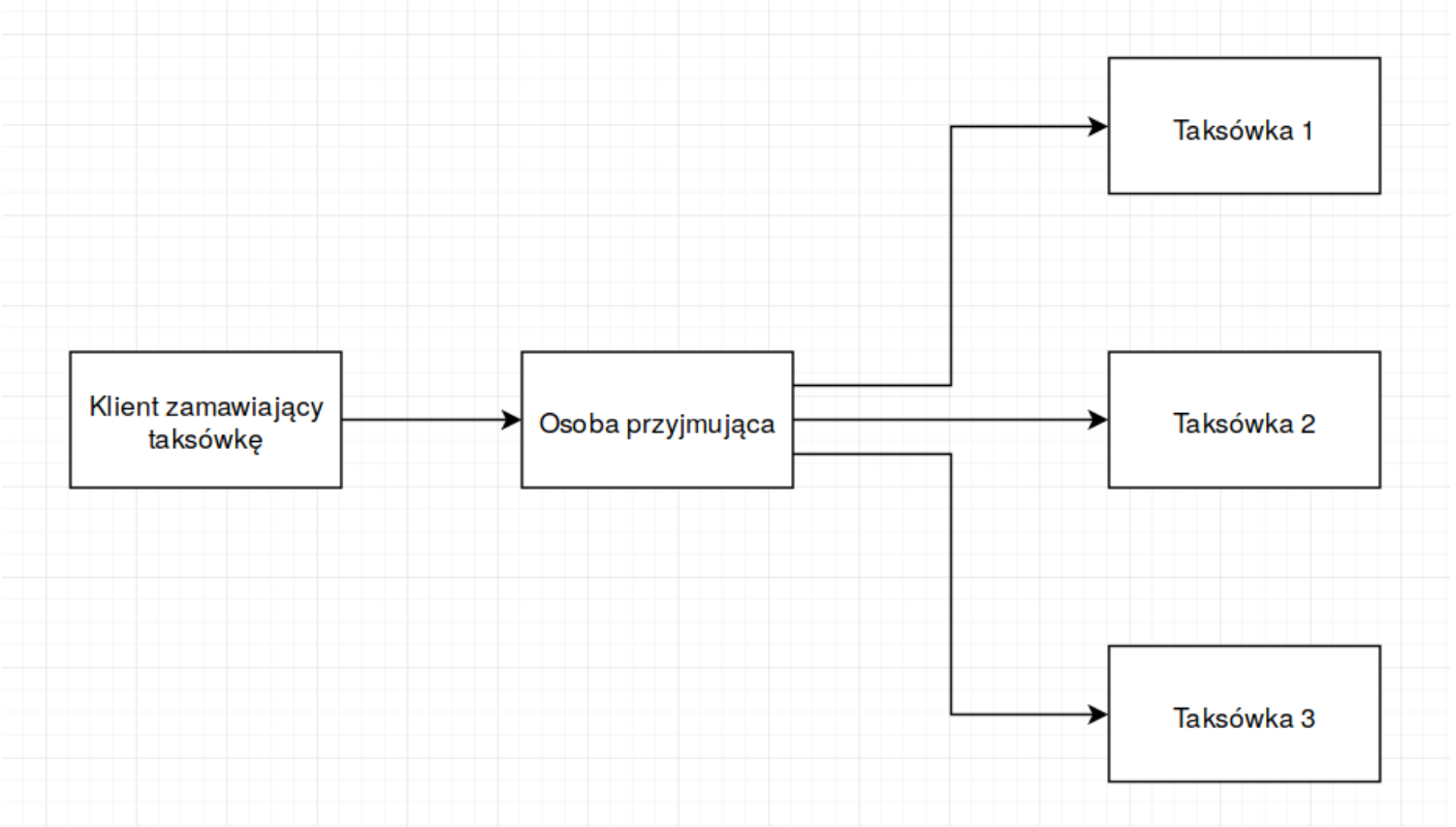
Wzorce projektowe

Obserwator oraz łańcuch odpowiedzialności

Observer

Obserwator



Założenia przykładu

Istnieje firma taksówkarska, w której klient zamawia taksówkę. Osoba przyjmująca zgłoszenie pobiera m.in. adres od klienta i przesyła go do wszystkich dostępnych taksówek, aby ich kierowcy mogli zdecydować, który podejmie się kursu.

Observer

Obserwator

- Wzorzec należący do grupy wzorców czynnościowych
- Funkcją wzorca jest nasłuchiwanie zdarzeń – jeżeli zdarzenia wystąpi, wszystkie obiekty obserwujące zostaną odpowiednio o tym poinformowane (działanie podobne jak w eventListener)
- Wyróżniamy dwa typy obiektów:
  - Obserwowany (observable, subject) – obiekt, o którym chcemy uzyskiwać informacje
  - Obserwator (observer, listener) – obiekt, którego funkcją jest oczekiwanie na zmianę stanu obiektu obserwowanego
- Jeden obiekt może obserwować kilka obiektów
- Jeden obiekt obserwowany może być obserwowany przez kilku obserwatorów

Observer

Obserwator

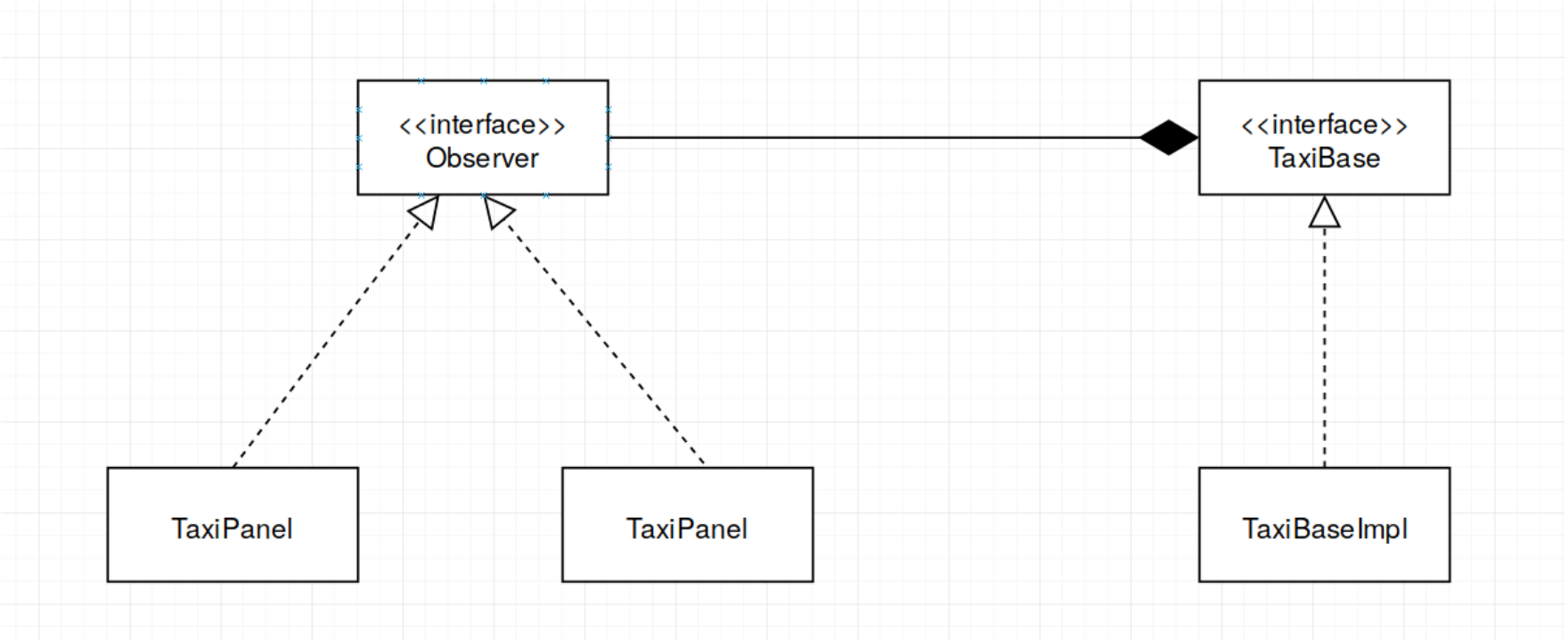


Diagram klas UML

Observer

Obserwator

Obiekt obserwowany

- U nas obiekt przyjmujący i rozsyłający adresy
- Posiada metody:
  - addObserver – dodanie nowego obserwatora (taksówki)
  - removeObserver – usunięcie obserwatora z listy obserwatorów
  - notifyAllObserver – metoda w momencie wystąpienia zdarzenia informuje o tym wszystkich obserwatorów z listy

```
public interface TaxiBase {

    public void addObserver(Observer observer);

    public void removeObserver(Observer observer);

    public void notifyAllObserver(String address);
}
```

Observer

Obserwator

Obiekt obserwowany

```java
public interface TaxiBase {

    public void addObserver(Observer observer);

    public void removeObserver(Observer observer);

    public void notifyAllObserver(String address);
}
```

```java
public class TaxiBaseImpl implements TaxiBase {

    private ArrayList<Observer> observers = new ArrayList<>();

    @Override
    public void addObserver(Observer observer) {
        if (observer != null) {
            observers.add(observer);
        }
    }

    @Override
    public void removeObserver(Observer observer) {
        if (observer != null) {
            observers.remove(observer);
        }
    }

    @Override
    public void notifyAllObserver(String address) {
        for (Observer observer : observers) {
            observer.update(address);
        }
    }
}
```

Observer

Obserwator

Obiekt obserwator

- U nas każda taksówka, która obserwuje zmianę obiektu przyjmującego adres
- Posiada metodę update, wywoływaną przez w momencie pojawienia się nowego zdarzenia

```java
public interface Observer {

    public void update(String address);
}
```

```java
public class TaxiPanel implements Observer {

    private String id;

    public TaxiPanel(String id) {
        this.id = id;
    }

    @Override
    public void update(String address) {
        System.out.println("Nowy adres [" + address + "] dla taksówki nr " + id);
    }

    public String getId() {
        return id;
    }
}
```

Observer

Obserwator

Wywołanie

- Tworzymy obiekt obserwowany
- Dodajmy do niego obiekty obserwujące
- Powiadamiamy wszystkich obserwatorów o zdarzeniu

```
public class App {

    public static void main(String[] args) {
        TaxiBase taxiBase = new TaxiBaseImpl();

        taxiBase.addObserver(new TaxiPanel("T1"));

        taxiBase.addObserver(new TaxiPanel("T2"));

        taxiBase.notifyAllObserver("ulica kodPocztowy miasto");
    }

}
```

Wynik działania:

```
run:
Nowy adres [ulica kodPocztowy miasto] dla taksówki nr T1
Nowy adres [ulica kodPocztowy miasto] dla taksówki nr T2
BUILD SUCCESSFUL (total time: 0 seconds)
```

Observer

Obserwator

- Zalety
  - Między obiektem obserwowanym i obserwującym istnieje „luźna” relacja – dzięki małej wiedzy o sobie nawzajem mogą być niezależnie rozbudowywane
  - Relacja między obserwowanym i obserwującym może być dynamicznie zmieniana (ilość obserwujących taksówek może być zmienna w danym momencie)
- Wady
  - Obserwatorzy nie mają informacji o innych obserwatorach

Chain of Responsibility

Łańcuch odpowiedzialności

Założenia przykładu

Osoba przyjmująca zlecenie od klienta wpisuje adres do pojedynczego pola tekstowego,
które za pomocą odpowiednich warunków rozdziela człony adresu na kod pocztowy, miasto i ulice

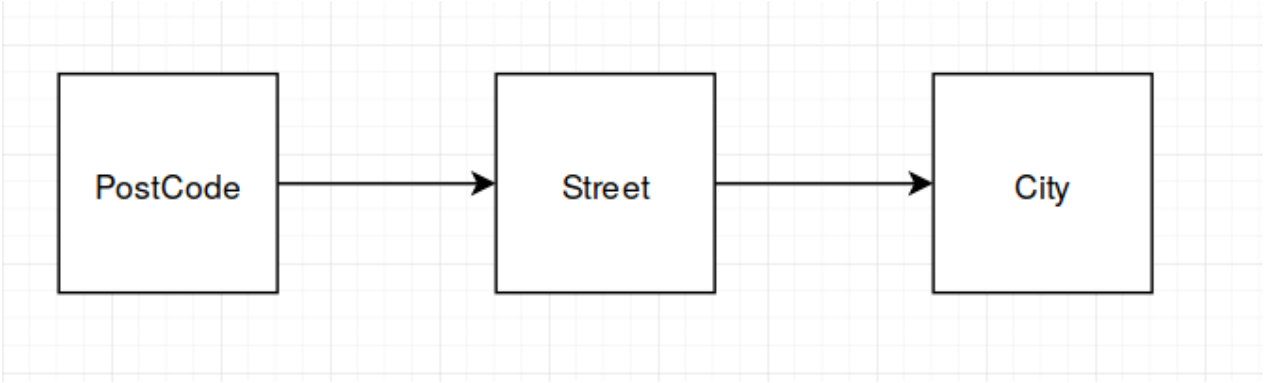
Chain of Responsibility

Łańcuch odpowiedzialności

- Wzorzec należący do grupy wzorców czynnościowych
- Łańcuch odpowiedzialności umożliwia pewnej liczbie klas podjęcie próby obsłużenia żądania podczas gdy żadna z nich nic nie wie o możliwościach innych
- Rezygnujemy z większych powiązań między klasami – jedynym powiązaniem jest przekazywane żądanie
- Żądanie jest tak długo przekazywane, aż natrafimy na klasę, która potrafi je obsłużyć
- Brak gwarancji, że każde żądanie będzie obsłużone
- Skutecznie zastępuje serię if’ów lub rozbudowanego switch’a
- W łatwy sposób można dodać kolejną obsługę żądania

Chain of Responsibility

Łańcuch odpowiedzialności

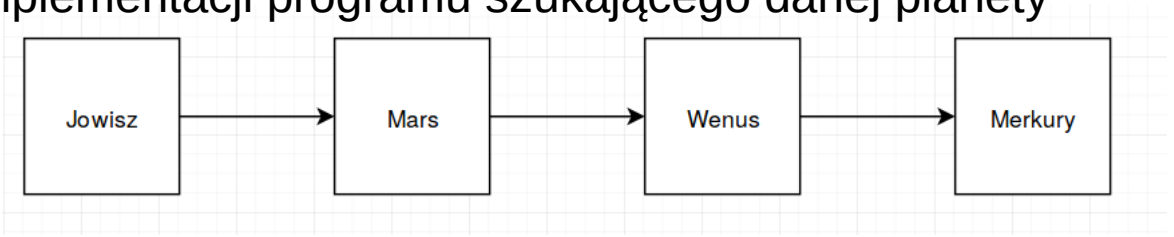


- Łańcuch jest uporządkowany od klas obsługujących bardziej specyficzne rozwiązania do bardziej ogólnych
- Zgodnie z przykładem (walidacja bardzo uproszczona):
  - Pierwszy człon sprawdza kod pocztowy: czy pośrodku jest myślnik, czy przed myślnikiem są dwie, a za cztery cyfry
  - Drugi człon sprawdza ulicę: czy na końcu znajduje się numer budynku (czyli cyfra lub cyfra z literą)
  - Trzeci człon działa na zasadzie: nie jest to kod pocztowy, nie jest to ulica więc jest to miasto

Chain of Responsibility

Łańcuch odpowiedzialności

Przykład implementacji programu szukającego danej planety




```java
public interface Chain {

    public void setNext(Chain next);

    public String process(PlanetTypes type);
}
```

```java
public class CheckJowisz implements Chain {

    private Chain next;

    @Override
    public void setNext(Chain next) {
        this.next = next;
    }

    @Override
    public String process(PlanetTypes type) {
        if (type == PlanetTypes.JOWISZ) {
            return "Planeta Jowisz";
        } else {
            return this.next.process(type);
        }
    }
}
```

```java
public class ChainOfResponsibilityExample {

    public static void main(String[] args) {
        Chain jowisz = new CheckJowisz();
        Chain mars = new CheckMars();
        Chain wenus = new CheckWenus();
        Chain merkury = new CheckMerkury();

        jowisz.setNext(mars);
        mars.setNext(wenus);
        wenus.setNext(merkury);

        System.out.println(jowisz.process(PlanetTypes.MARS));
        System.out.println(jowisz.process(PlanetTypes.JOWISZ));

    }

}
```

Chain of Responsibility

Łańcuch odpowiedzialności

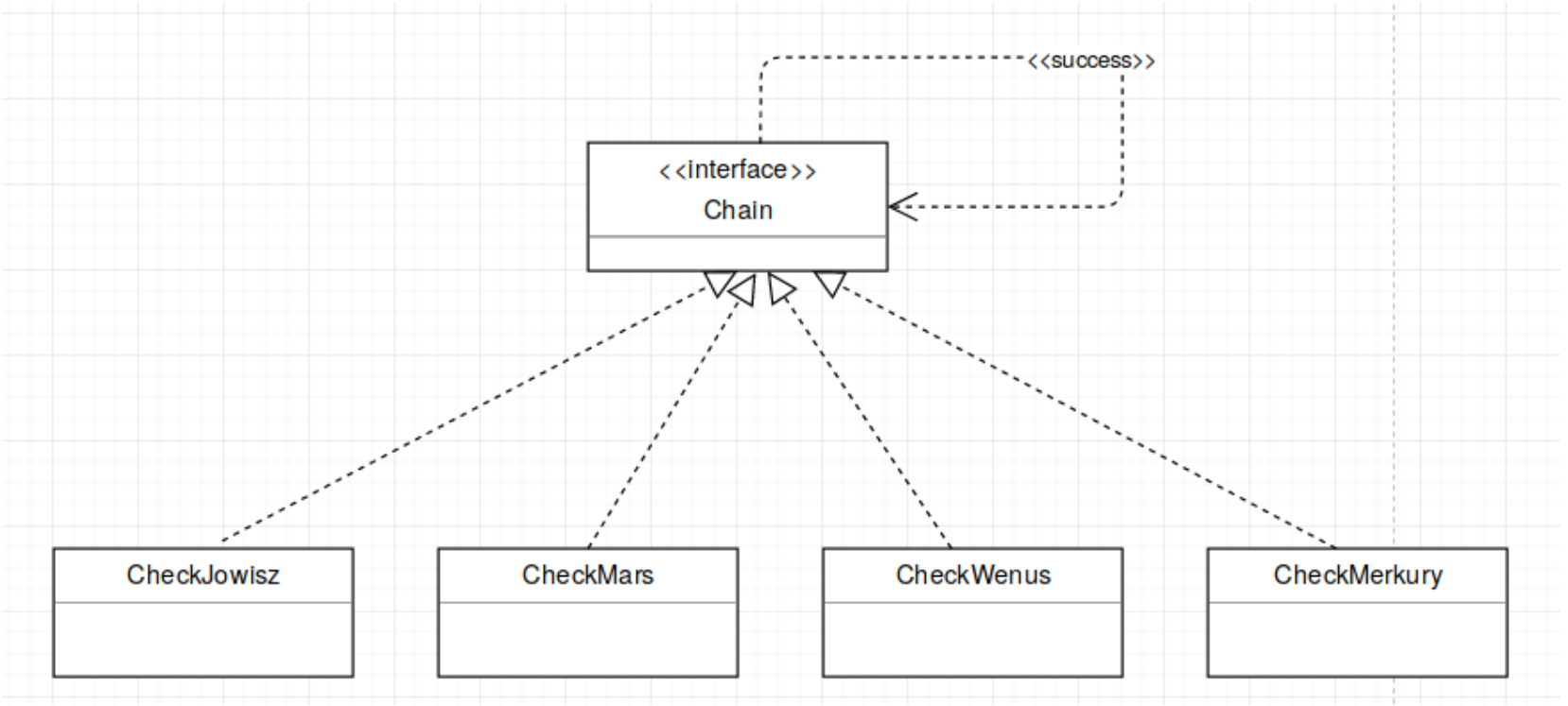


Diagram klas UML