Wzorce behawioralne

- opisujące zachowanie i odpowiedzialność współpracujących ze sobą obiektów
- umożliwiają organizację, zarządzanie i łączenie zachowań

Wzorce behawioralne

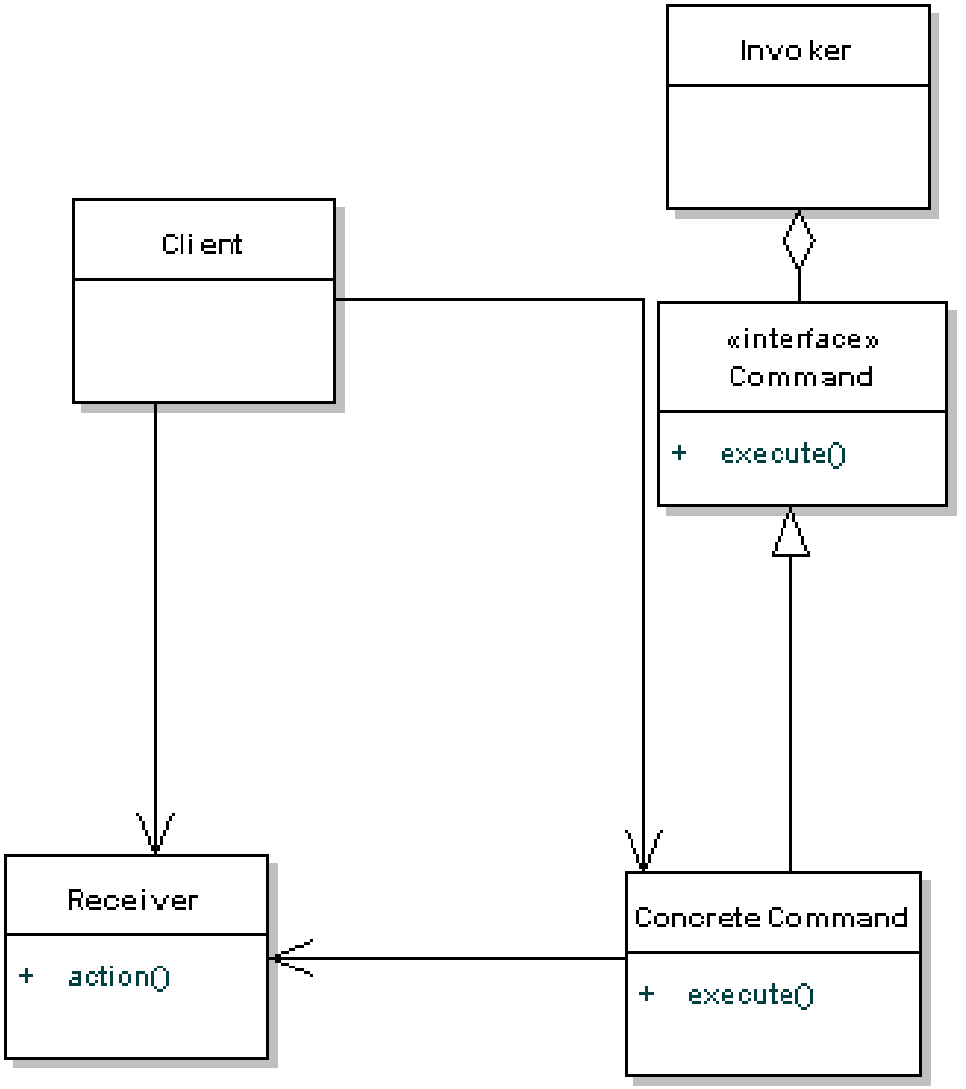
- Chain of responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template
- Visitor

Command

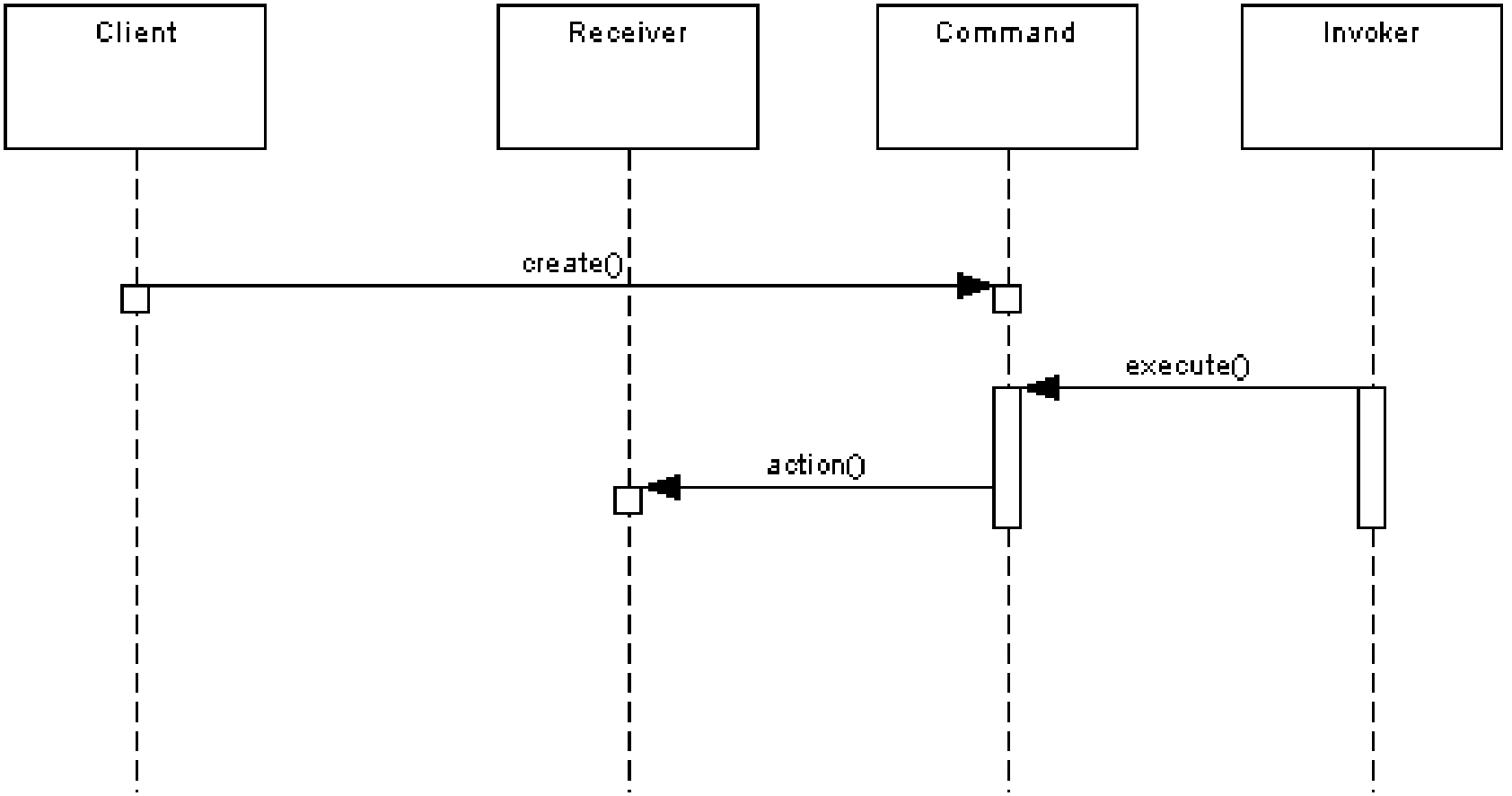
Wzorzec – Command (Polecenie)

- Ukrycie logiki obsługi poleceń w obiektach poleceń.
- Interfejs takich obiektów jest bardzo prosty, występuje najczęściej tylko jedna metoda, typowo void execute() lub void run().
- Wykorzystywane w celu eliminacji długich instrukcji warunkowych w tzw. dispatcherach. Przykładem wzorca Command jest interfejs Runnable. Idealny do implementacji operacji cofnij-ponów.

Command


Invoker
Client
«interface»
Command
+ execute()
Receiver
+ action()
Concrete Command
+ execute()

Command



Przykłady

- java.lang.Runnable w JDK
- javax.swing.Action w JDK
- pl.anicos.patterns.exercises.command

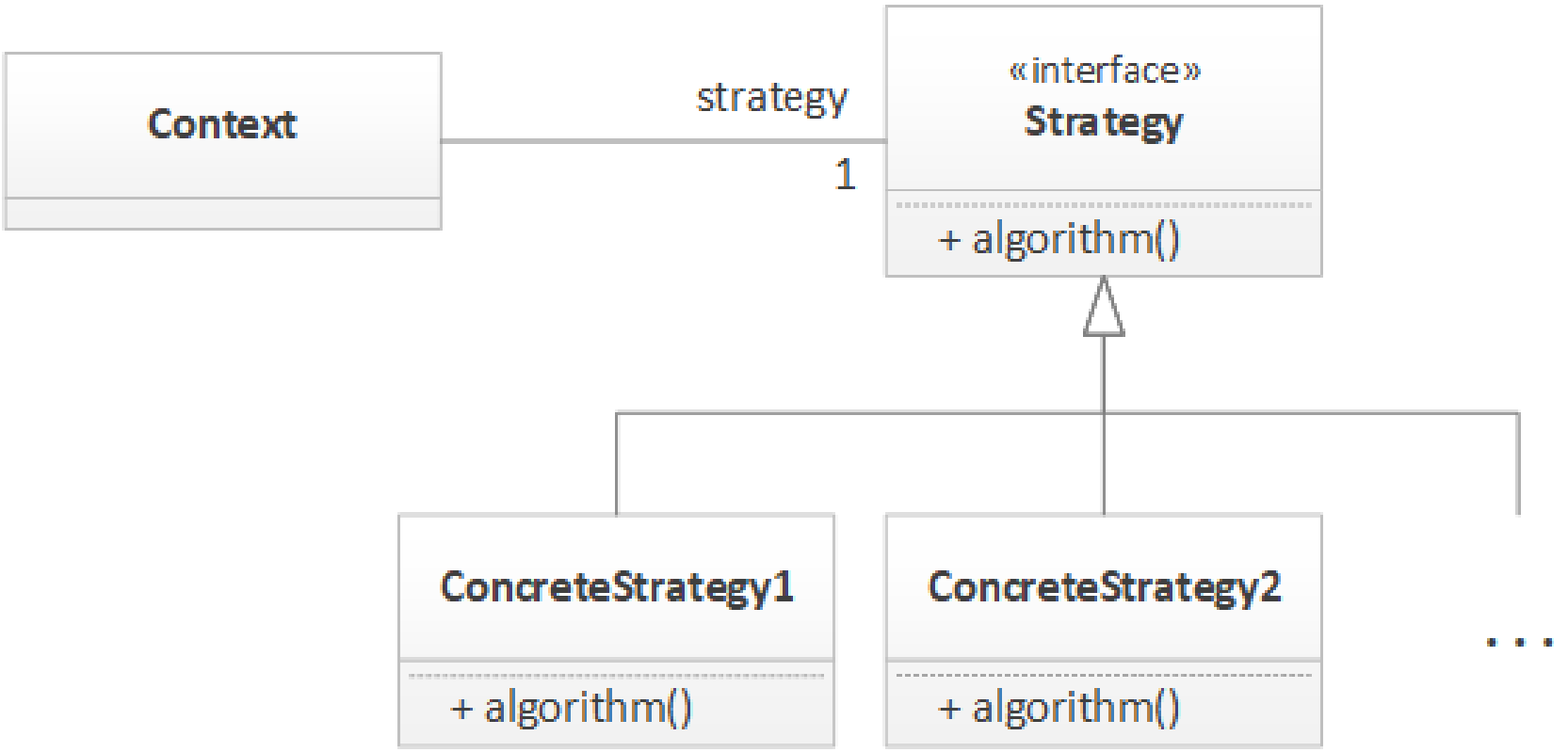
Strategy

- Polega na hermetyzowaniu operacji umożliwiając stworzenie zamiennych implementacji
- Zamiast używać instrukcji **if czy switch użyj strategii**
- Zawsze przed zastosowaniem dziedziczenia warto pomyśleć, czy wzorzec strategii nie będzie lepszym rozwiązaniem.

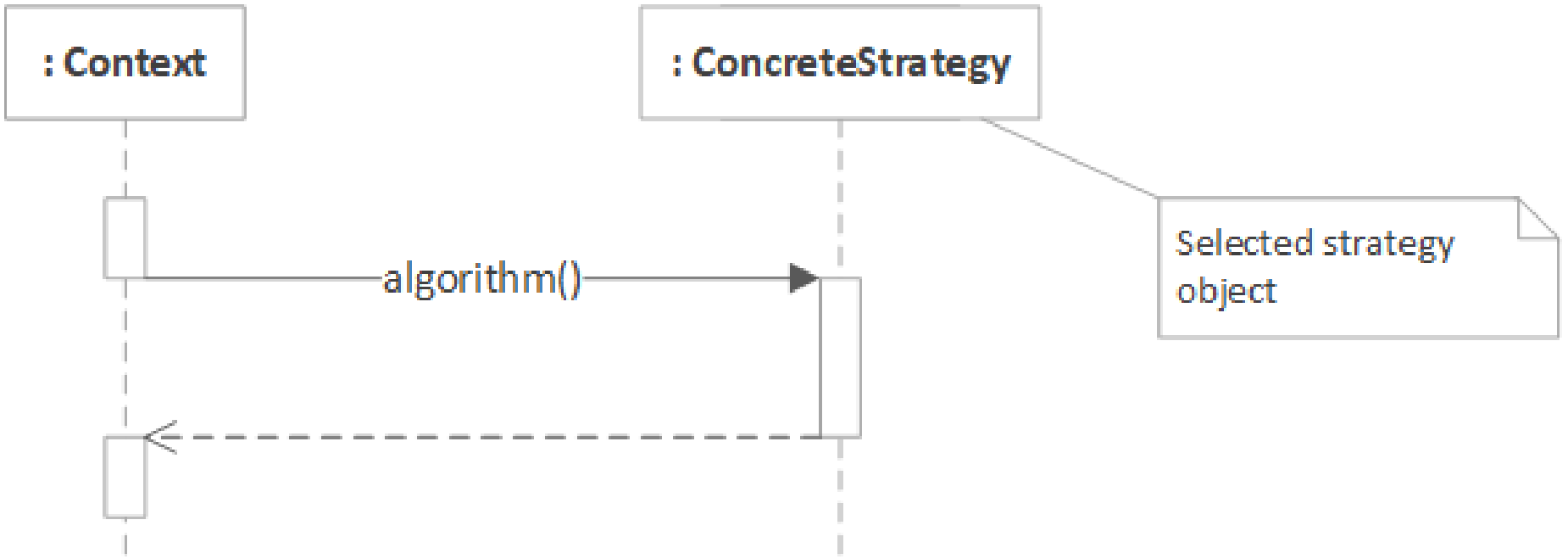
Przykład z życia wzięty

- Tworzymy sklep internetowy oferujący swoje usługi w kilku państwach. Jak wiadomo prawo podatkowe znacząco różni się w poszczególnych krajach.
- Powstaje problem naliczenia odpowiedniego podatku dla klientów pochodzących z odmiennych państw.
- Jak to rozwiązać?
- Wybrać odpowiednią stawkę za pomocą licznych instrukcji warunkowych? Nie, do tego świetnie nadaje się **wzorzec strategii**.

Strategy – diagram klas



Strategy – diagram aktywności



Użycie Strategi

```
public class StrategyTest {
  @Test
  public void runStrategy() {
    Cart cart = new Cart();
    cart.addLineItem(new Item("Milk", 3.0));
    cart.addLineItem(new Item("Water", 2.0));
    double polandPrice = cart.calculateFinalPrice(TaxPolicyFactory.getTaxPolicy("Poland"));
    System.out.println("Final price in poland id " + polandPrice);
    double germanPrice = cart.calculateFinalPrice(TaxPolicyFactory.getTaxPolicy("Germany"));
    System.out.println("Final price in Germany id " + germanPrice);
    double andoreaPrice = cart.calculateFinalPrice(TaxPolicyFactory.getTaxPolicy("Andora"));
    System.out.println("Final price in Andora id " + andoreaPrice);
  }
}
```

Przykłady Strategy

- java.util.Comparator#compare() podczas wywłania Collections#sort()
- pl.anicos.patterns.exercises.strategy.StrategyTest

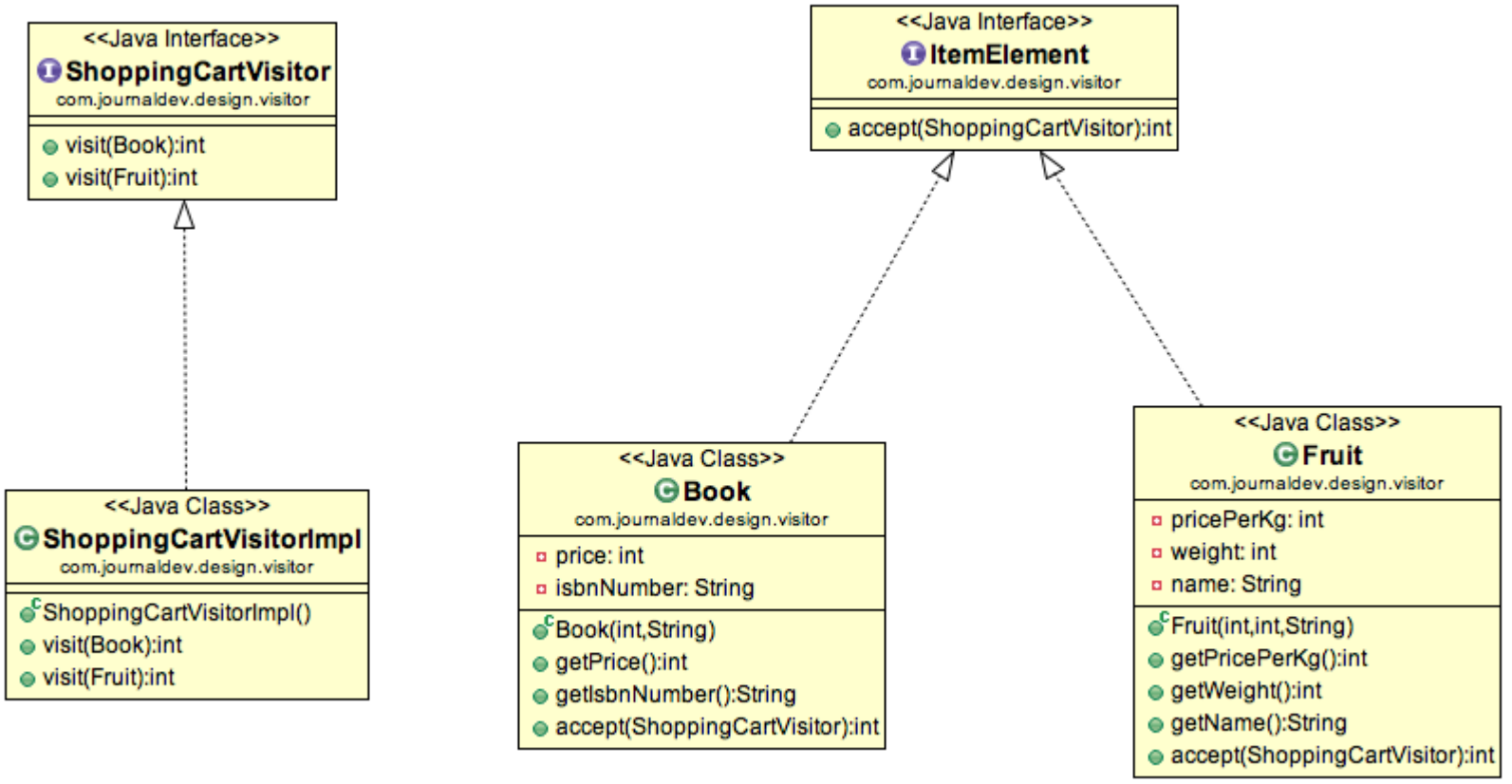
Visitor

- zadaniem wzorca projektowego Visitor jest oddzielenie algorytmu od struktury obiektów, na których operuje
- wprowadza zostaje nowy obiekt Visitor, którego zadaniem jest odwiedzenie każdego z obiektów pewnego zespołu i wykonanie na każdym z nich konkretnej operacji

Visitor – wykorzystanie

- Wzorzec ten bardzo często wykorzystywany jest do przeszukiwania dużych struktur danych i generowania na ich podstawie zbiorczych raportów
- zaletą wzorca jest brak bezpośredniej ingerencji w gotowe już struktury danych

Visitor – diagram klas



Visitor - przykłady

- javax.lang.model.element.AnnotationValue and AnnotationValueVisitor
- javax.lang.model.element.Element and ElementVisitor
- javax.lang.model.type.TypeMirror and TypeVisitor
- java.nio.file.FileVisitor and SimpleFileVisitor
- javax.faces.component.visit.VisitContext and VisitCallback
- pl.anicos.patterns.exercises.visitor

Observer

- Wzorzec obserwator pozwala na odseparowanie obiektów od siebie, w taki sposób, że mogą ze sobą współpracować, ale jednocześnie mało o sobie wiedzą

Observer

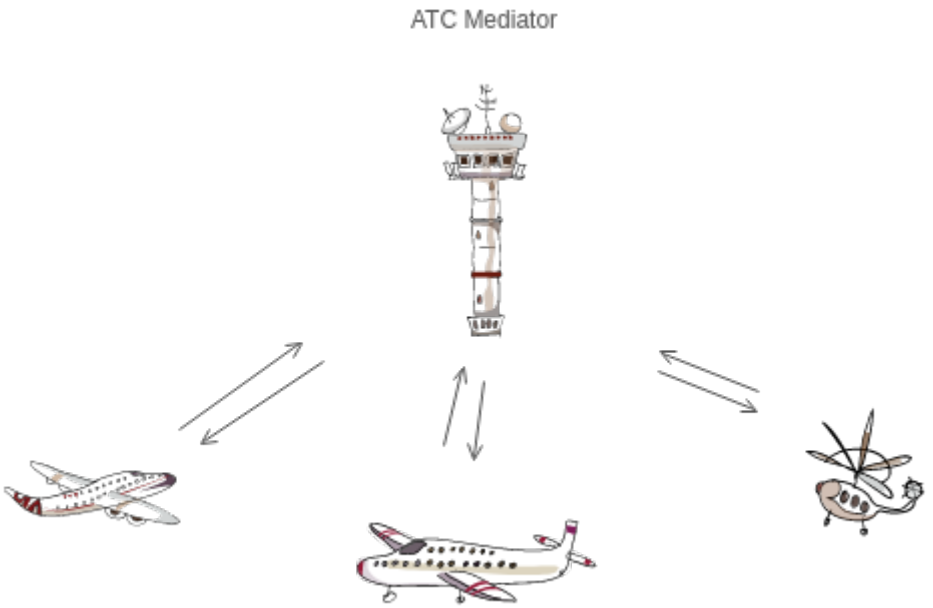
W obserwatorze możemy wyróżnić elementy:

- interfejs wzorca obserwator – muszą ją implementować obiekty obserwujące
- obiekt obserwowany (observable, subject) – obiekt, który interesuje nas pod względem zmiany stanów
- obiekt obserwatora (observer, listener) – obiekty które są zainteresowane uzyskaniem informacji w przypadku zmiany stanu obiektu obserwowanego

Observer - zalety

- Luźna zależność pomiędzy obiektami – obserwowany <-> obserwator
- Relacja pomiędzy obiektami jest dynamiczna, jest możliwa zmiana w trakcie działania aplikacji
- Możliwość ograniczenia do korzystania z poszczególnych elementów systemu, wystarczy odłączyć danego obserwatora

Mediator



Mediator – kiedy stosujemy?

- gdy wiele obiektów o wspólnym interfejsie musi komunikować się ze sobą w celu wykonania określonego zadania

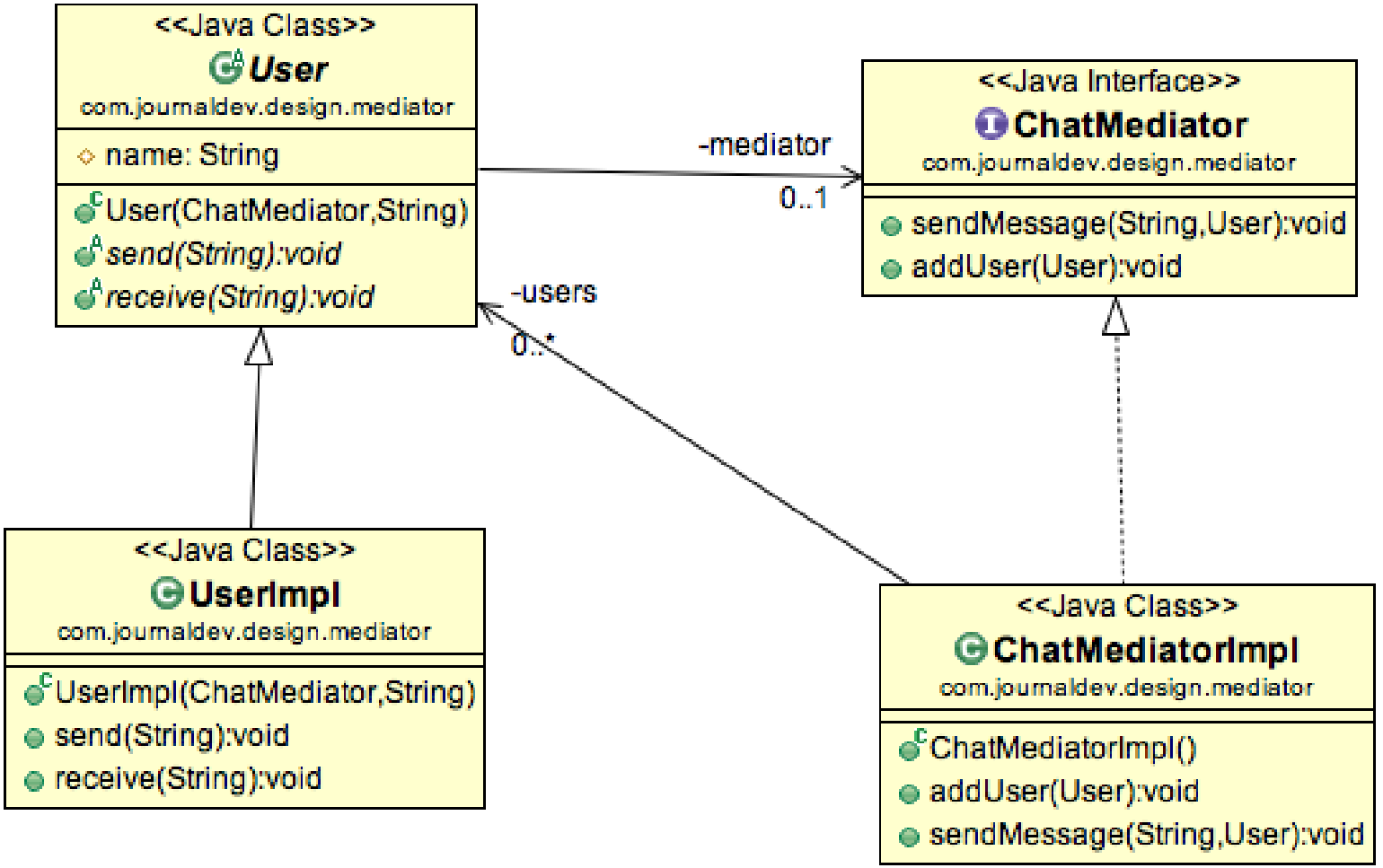
Mediator

- Obiekt mediatora jedyny zna większość współpracujących ze sobą obiektów
- Zapewnia zmniejszenie ilości powiązań pomiędzy klasami
- Daje jednolity interfejs do przesyłania komunikatów pomiędzy klasami

Mediator - zalety

- Luźna zależność pomiędzy obiektami w systemie – nadawca komunikatu -> mediator -> odbiorca lub odbiorcy komunikatów
- Relacja pomiędzy tymi obiektami jest dynamiczna, jest możliwe dodanie nowych opcji w trakcie działania systemu
- Możliwość ograniczenia korzystania z poszczególnych elementów systemu, wystarczy odłączyć danego odbiorcę komunikatów lub nadawcę komunikatów

Mediator


<<Java Class>>
User
com.journaldev.design.mediator
name: String
User(ChatMediator,String)
send(String):void
receive(String):void
-mediator
0..1
<<Java Interface>>
ChatMediator
com.journaldev.design.mediator
sendMessage(String,User):void
addUser(User):void
-users
0..*
<<Java Class>>
UserImpl
com.journaldev.design.mediator
UserImpl(ChatMediator,String)
send(String):void
receive(String):void
<<Java Class>>
ChatMediatorImpl
com.journaldev.design.mediator
ChatMediatorImpl()
addUser(User):void
sendMessage(String,User):void