Programovanie v jazyku Python

Návrhové vzory II
prednáška 11

Katedra kybernetiky a umelej inteligencie
Technická univerzita v Košiciach
Ing. Ján Magyar, PhD.

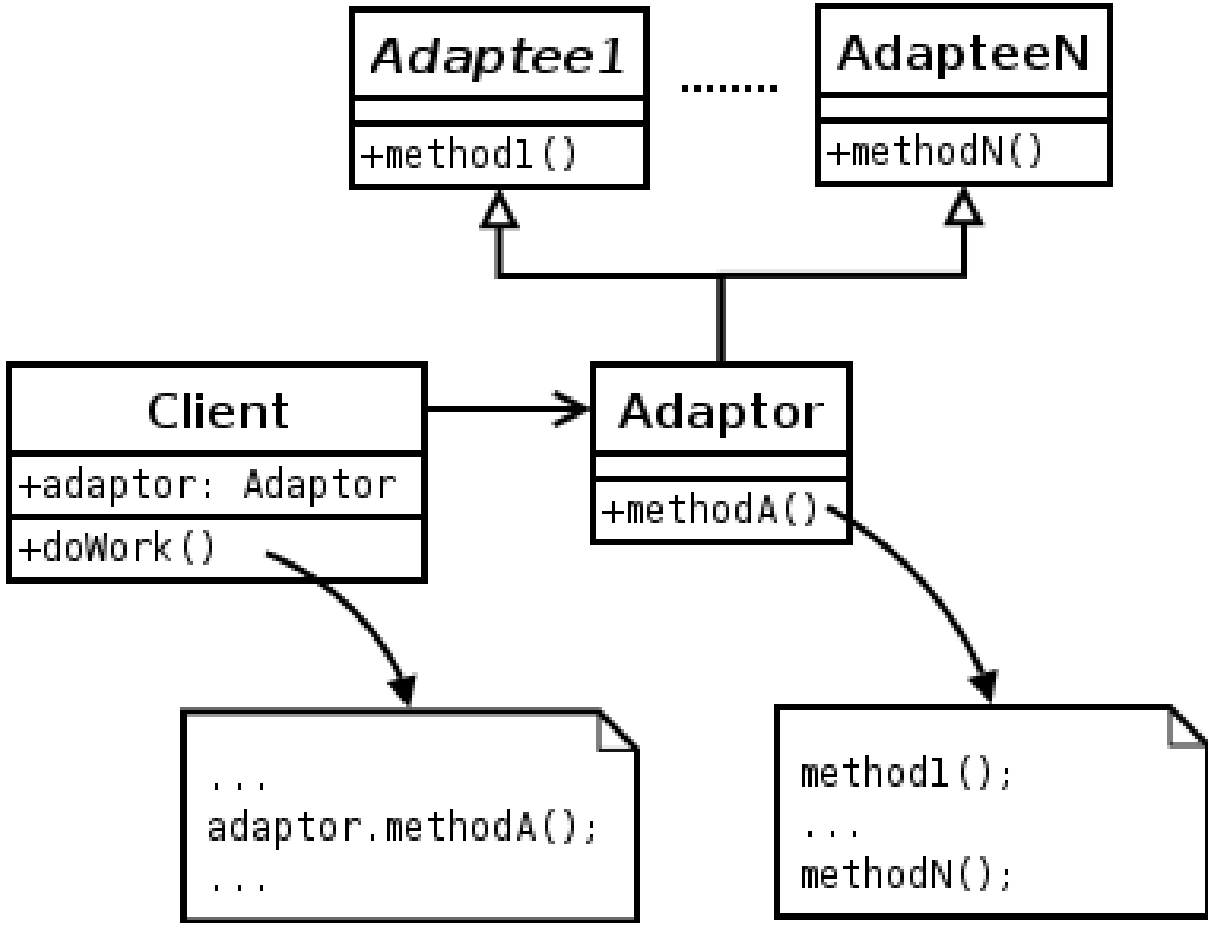
Štrukturálne vzory

- adapter/bridge
- decorator
- facade
- proxy

Adapter

- konvertovanie rozhrania triedy na iné rozhranie, ktoré očakáva ďalšia trieda
- umožňuje spoluprácu dvoch tried s nekompatibilnými rozhraniami
- `adapter` konvertuje rozhranie `adaptee` na `target`
- možný na úrovni triedy alebo inštancie

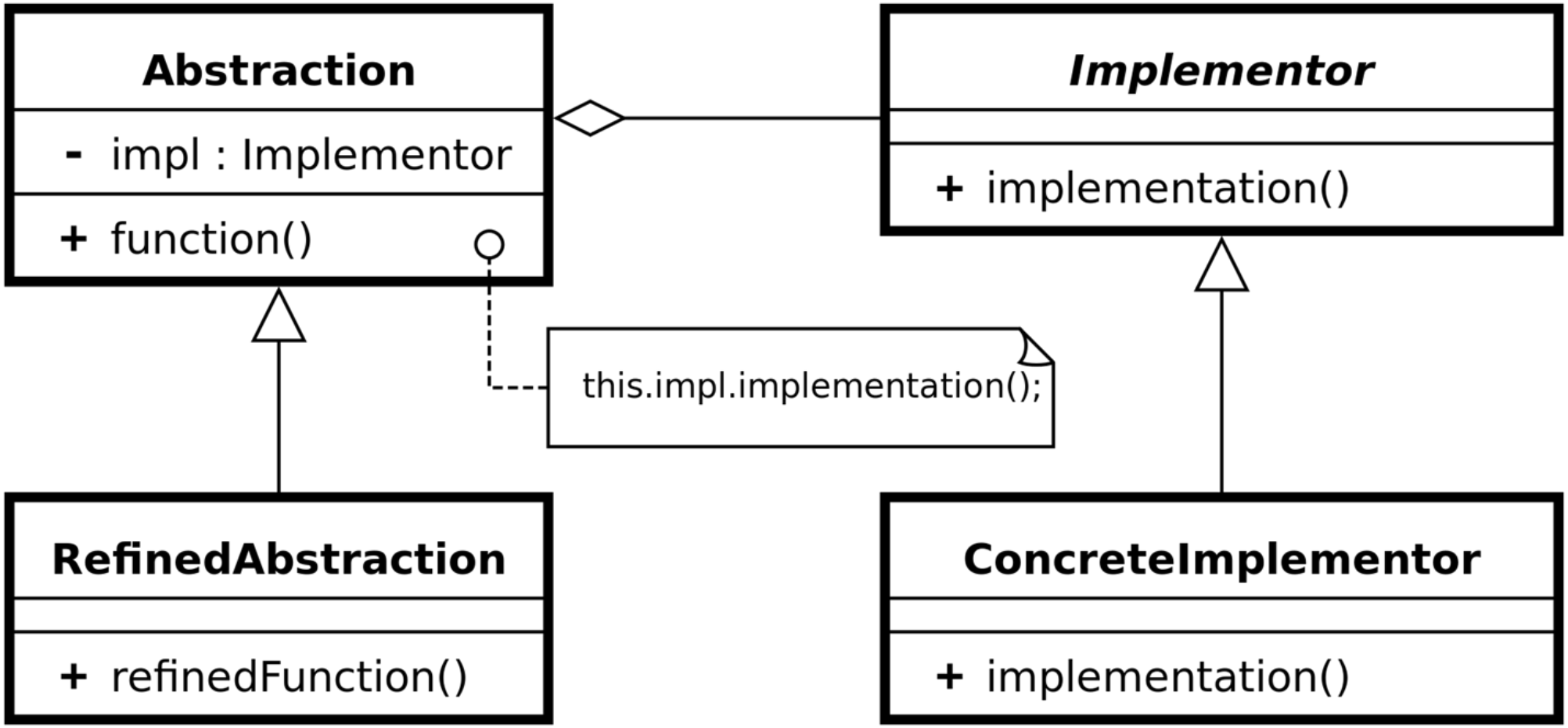
Adapter


Adaptee1
+method1()
.....
AdapteeN
+methodN()
Client
+adaptor: Adaptor
+doWork()
Adaptor
+methodA()
...
adaptor.methodA();
...
method1();
...
methodN();

Bridge

- oddelí abstrakciu od jej implementácie
- môžeme zmeniť abstrakciu alebo implementáciu bez potreby zmeniť druhú
- užitočné ak trieda a jej úloha sa často mení
- pridá druhú vrstvu abstrakcie
- často implementované ako adaptor na úrovni objektu

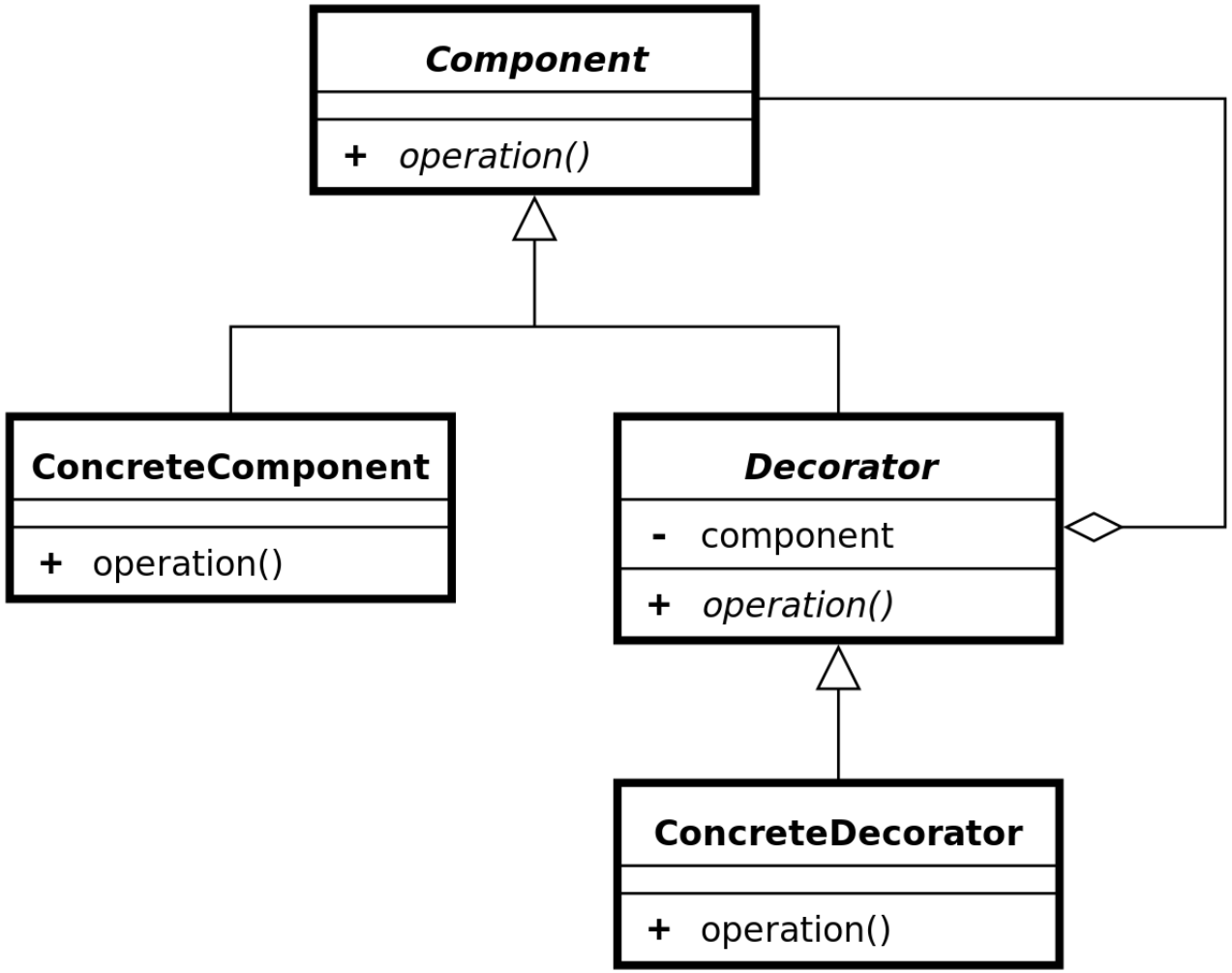
Bridge


Abstraction
- impl : Implementor
+ function()
Implementor
+ implementation()
this.impl.implementation();
RefinedAbstraction
+ refinedFunction()
ConcreteImplementor
+ implementation()

Decorator

- pridáme nové zodpovednosti objektom dynamicky bez zmeny ich rozhrania
- alternatíva k dedičnosti s cieľom rozšíriť funkcionalitu
- rozhranie rozšíreného objektu (`Component`) implementujeme preposlaním všetkých požiadaviek a následným vykonaním ľubovoľnej dodatočnej funkcionality

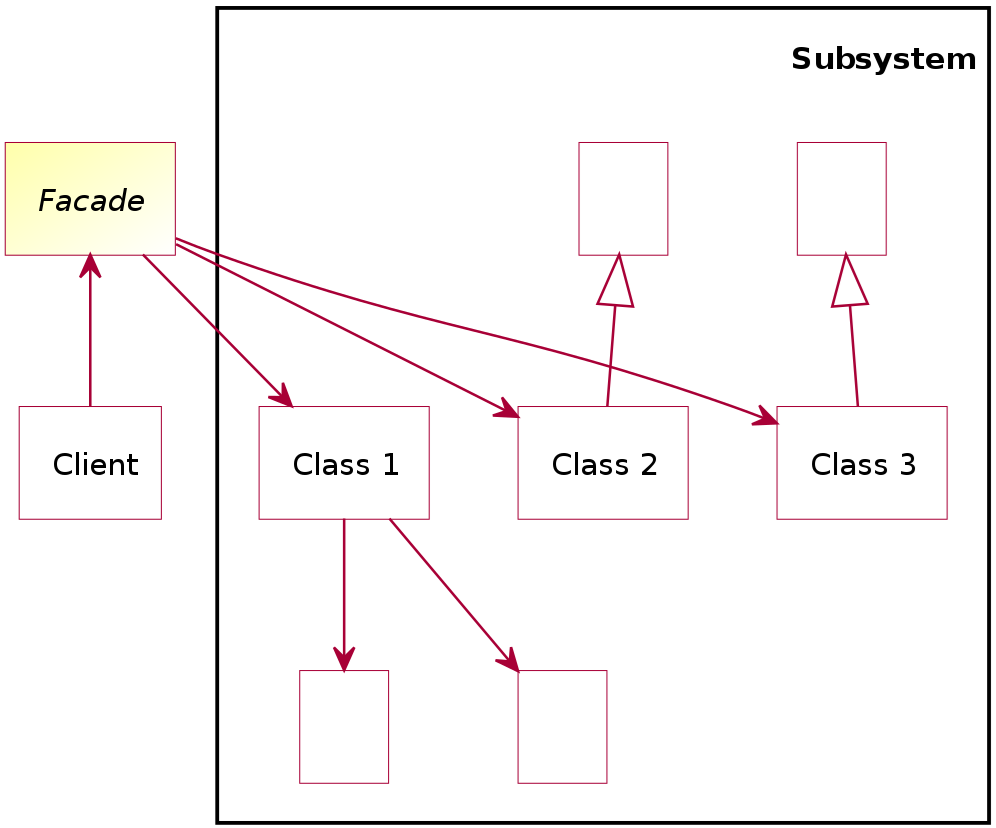
Decorator

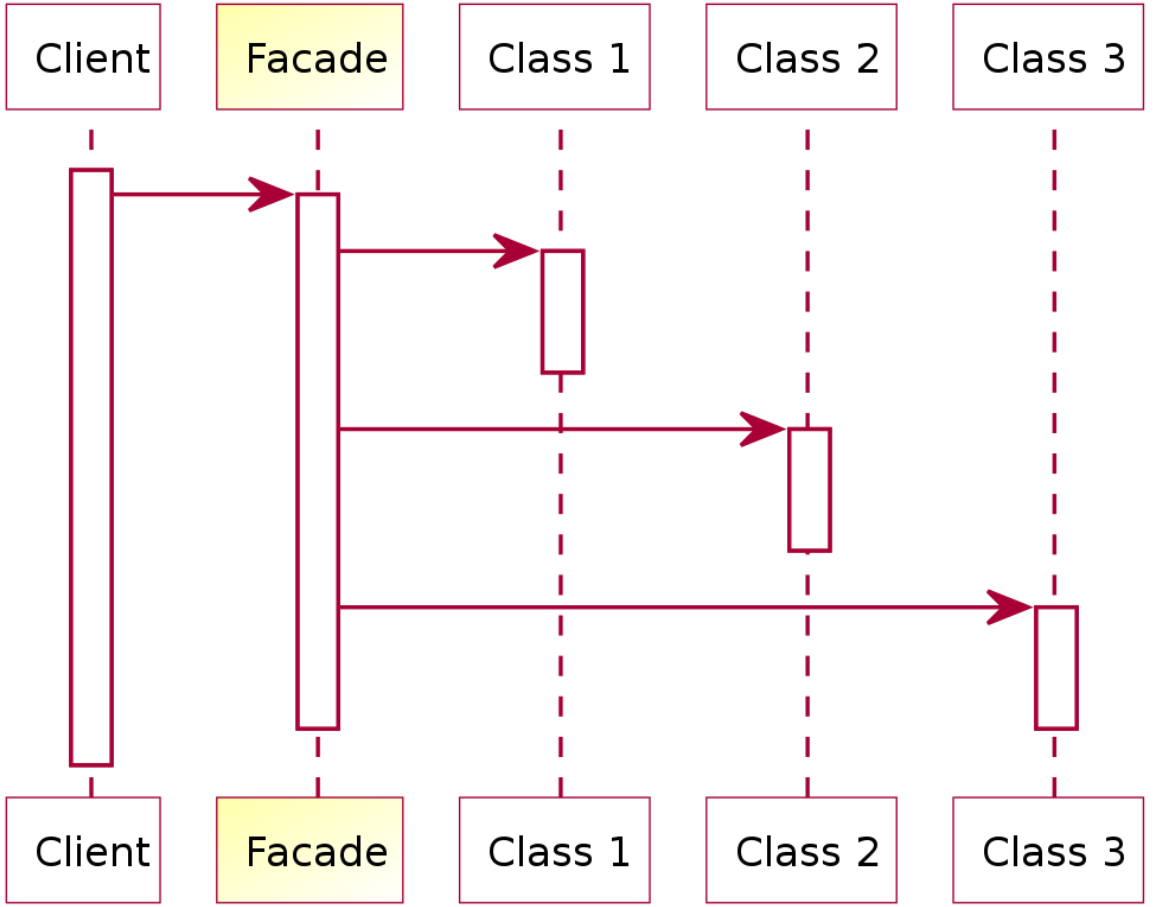

Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

Facade

- poskytuje zjednotené rozhranie pre sadu rozhraní definíciou rozhrania vyššej úrovne
- subsystémy sa použijú ľahšie
- lepšia čitateľnosť
- loose coupling

Facade

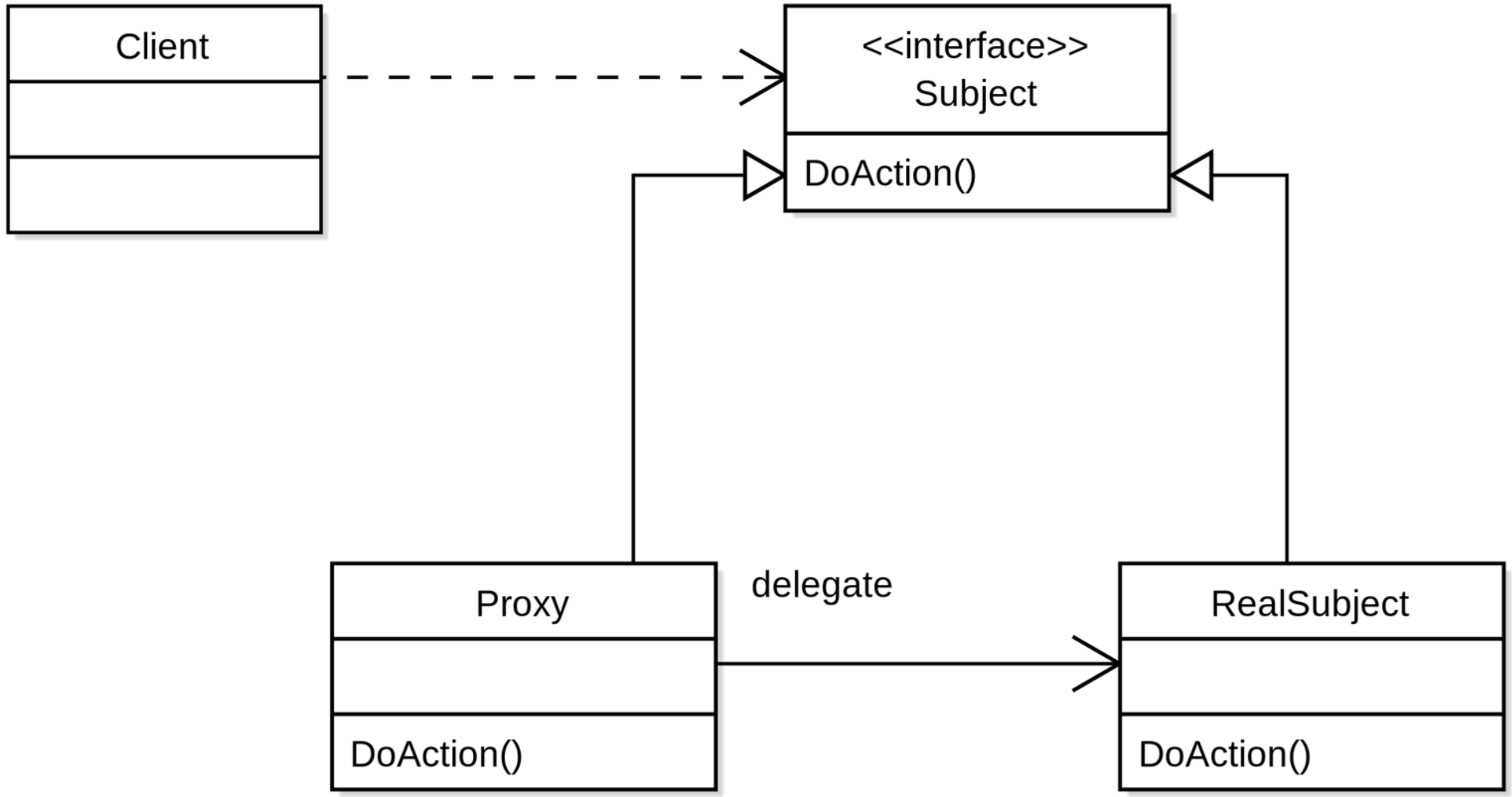

Subsystem
Facade
Client
Class 1
Class 2
Class 3


Client
Facade
Class 1
Class 2
Class 3
Client
Facade
Class 1
Class 2
Class 3

Proxy

- kontroluje prístup k objektu poskytnutím placeholdera, cez ktorý prechádza komunikácia
- proxy objekt môžeme použiť ako náhradu za iný objekt
- proxy môže definovať ďalšiu funkcionalitu pre kontrolu prístupu k objektu
- proxy môže byť
  - remote
  - virtual
  - protection

Proxy


Client
<<interface>>
Subject
DoAction()
Proxy
DoAction()
delegate
RealSubject
DoAction()

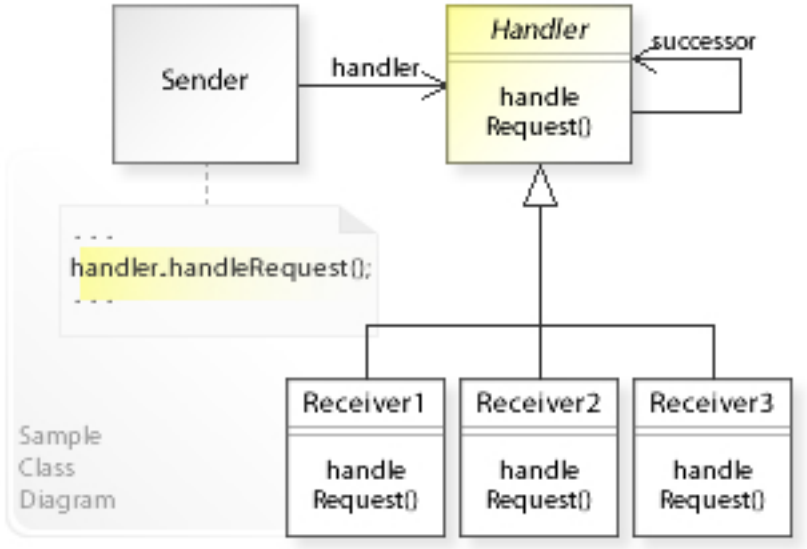
Behaviorálne vzory

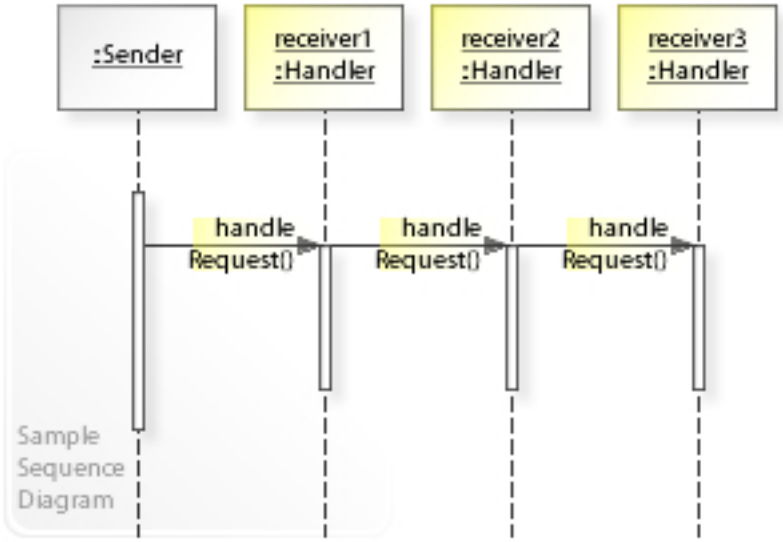
- chain of responsibility
- command
- iterator
- mediator
- observer a publish/subscribe
- strategy
- visitor

Chain of responsibility

- viac objektov môže spracovať požiadavku
- vytvoríme zreťazenie procesorov a správa sa posiela ďalej kým nie je spracovaná (alebo nie je spracovaná vôbec)
- máme zdroj command objektov a sériu procesorov
- každý procesor buď spracuje požiadavku alebo ju prepošle na základe podmienok počas behu

Chain of responsibility

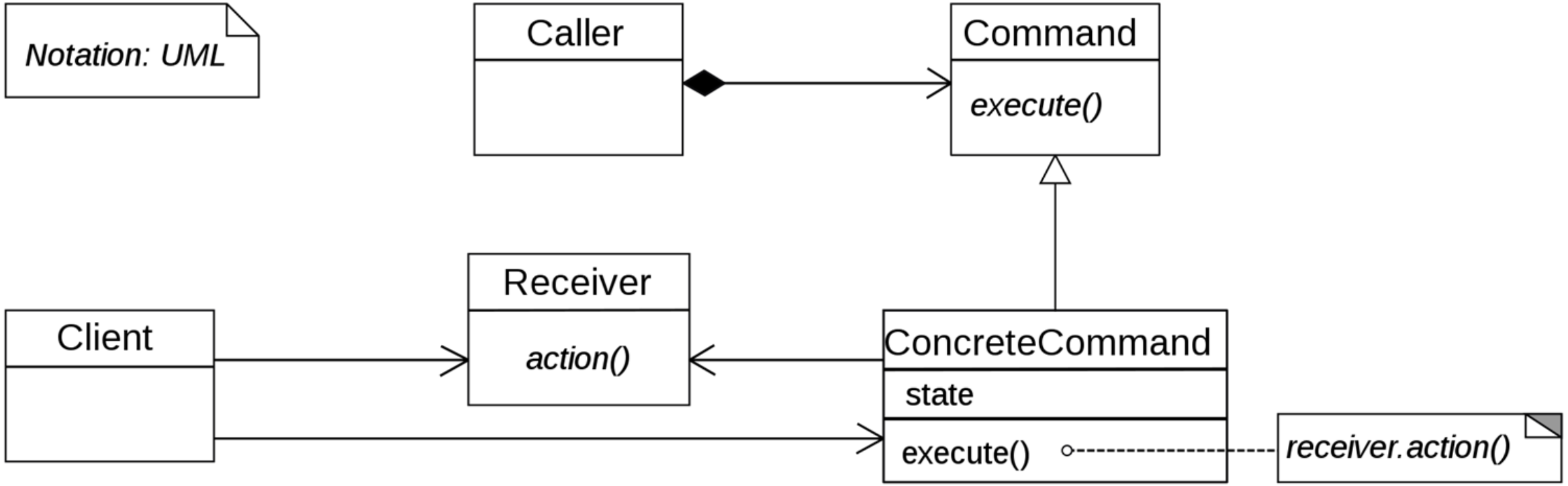




Command

- enkapsulácia požiadavky do objektu
- umožňuje parametrizáciu klientov s rôznymi požiadavkami
- pre queueing, logging, a nezvrátiteľné operácie
- požiadavka je delegovaná do command objektu namiesto priameho spracovania
- napr. GUI buttons, progress bars, transactions, wizards

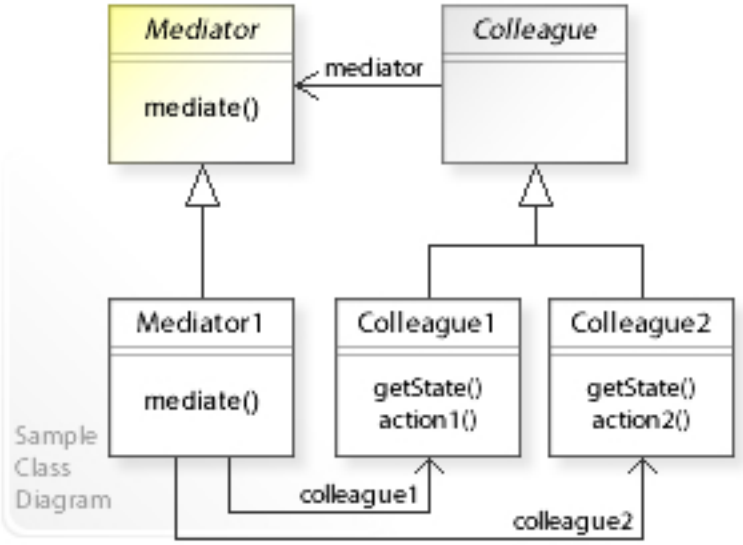
Command

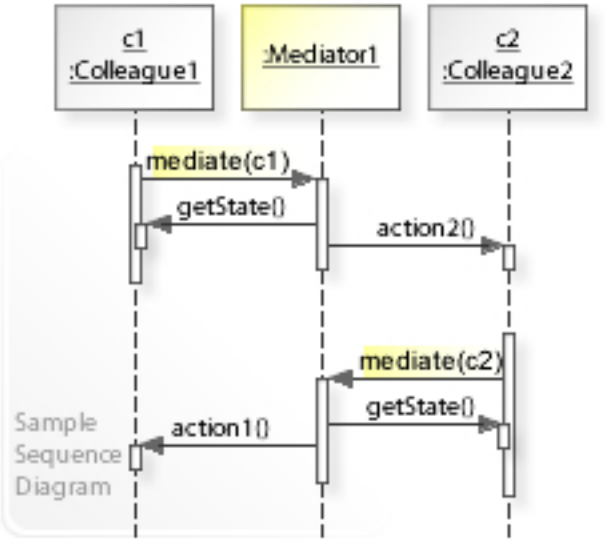

Notation: UML
Caller
Command
execute()
Receiver
action()
Client
ConcreteCommand
state
execute()
receiver.action()

Mediator

- enkapsulácia interakcie medzi sadou objektov
- umožňuje loose coupling medzi objektmi, keďže neodkazujú jeden na druhý explicitne
- môžeme meniť interakciu objektov
- zadefinujeme triedu mediátor, ktorá sa potom používa pre komunikáciu medzi objektmi

Mediator

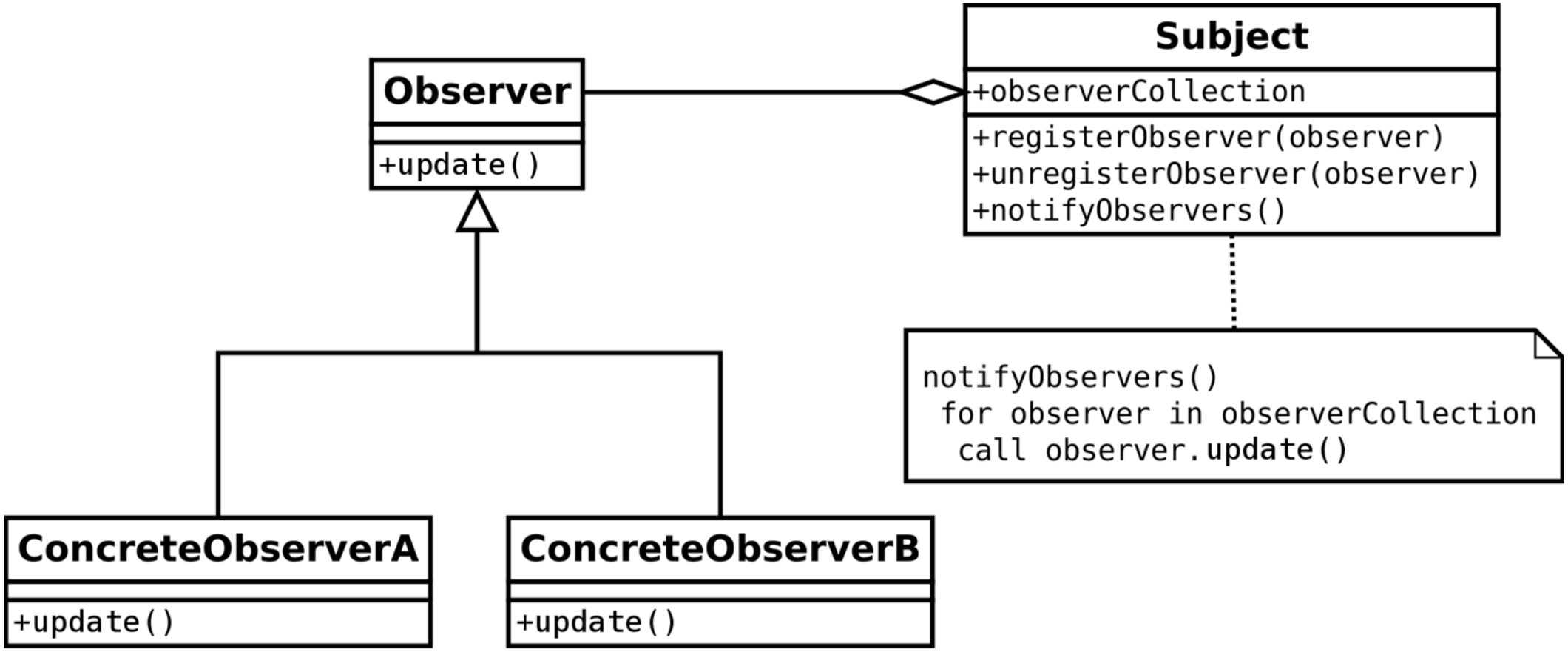

Mediator
mediate()
mediator
Colleague
Mediator1
mediate()
Colleague1
getState()
action1()
Colleague2
getState()
action2()
colleague1
colleague2
Sample
Class
Diagram


c1
:Colleague1
:Mediator1
c2
:Colleague2
mediate(c1)
getState()
action2()
mediate(c2)
getState()
action1()
Sample
Sequence
Diagram

Observer

- definícia závislosti one-to-many
- zmena jedného objektu vyžaduje notifikáciu ďalších závislých objektov
- implementuje vzor publisher/subscriber
- **subject** udržiava zoznam svojich **observerov** a notifikuje ich automaticky, tie následne zmenu spracujú vhodným spôsobom

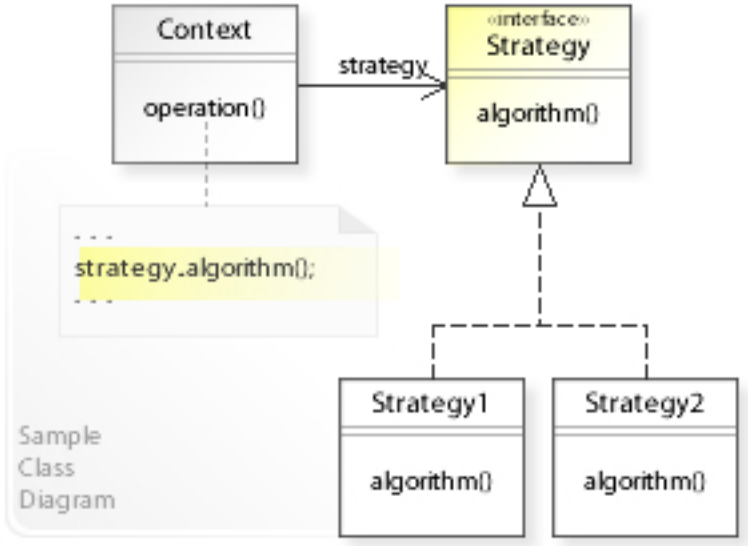
Observer

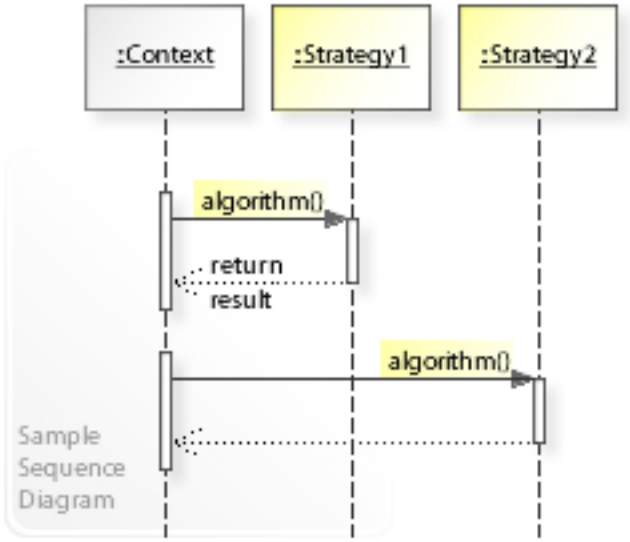

Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
Observer
+update()
notifyObservers()
for observer in observerCollection
call observer.update()
ConcreteObserverA
+update()
ConcreteObserverB
+update()

Strategy

- zadefinujeme skupinu podobných algoritmov s rovnakým rozhraním
- algoritmy potom vieme používať zameniteľne
- vyberieme algoritmus počas behu na základe podmienky
- pred zavolaním algoritmu objekt dostane informáciu o tom, ktoré riešenie má použiť a následne zavolá zodpovedajúcu metódu

Strategy

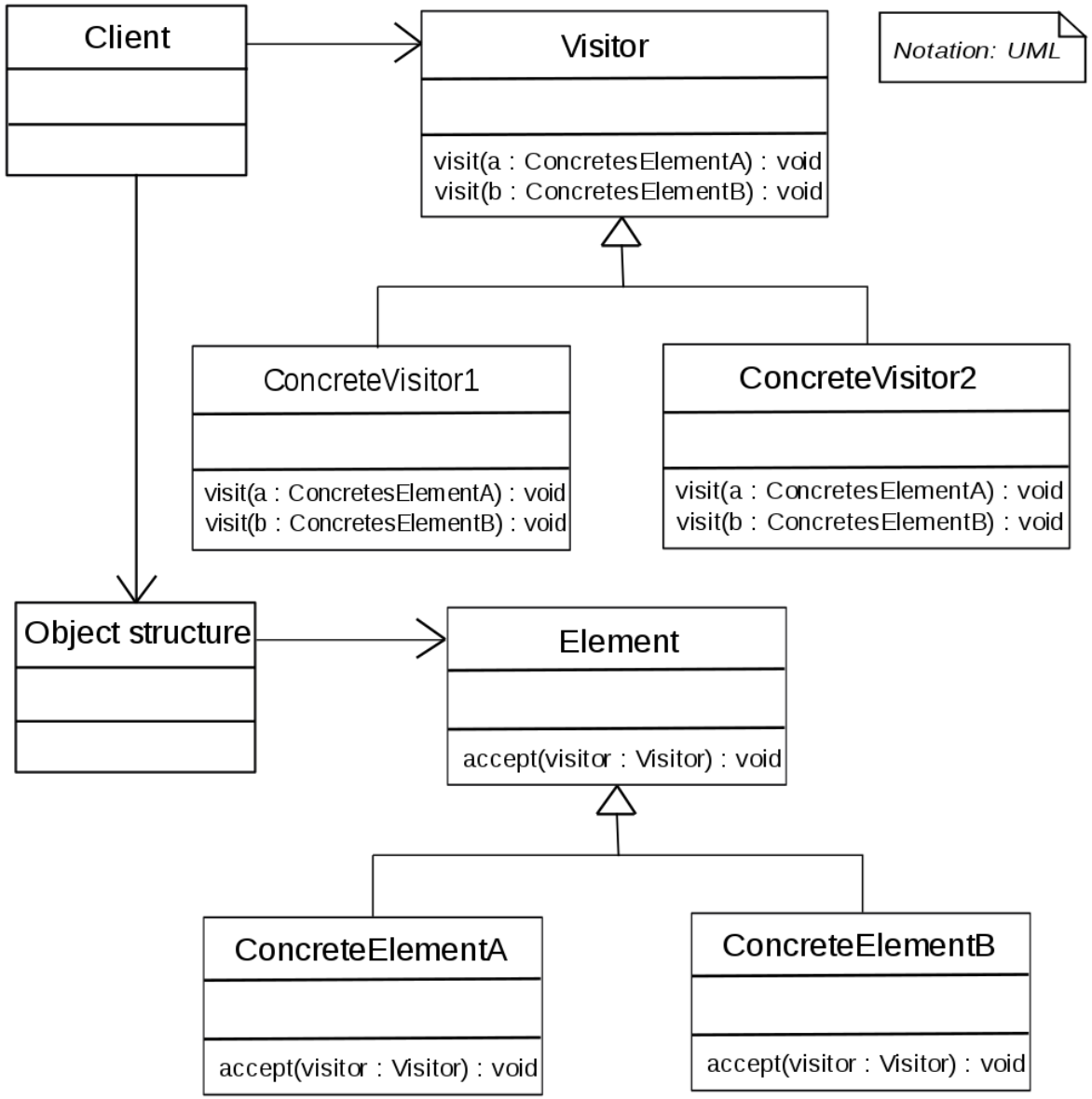

Context
operation()
strategy
«interface»
Strategy
algorithm()
strategy.algorithm();
Strategy1
algorithm()
Strategy2
algorithm()
Sample
Class
Diagram


:Context
:Strategy1
:Strategy2
algorithm()
return
result
algorithm()
Sample
Sequence
Diagram

Visitor

- pre operácie, ktoré sa vykonajú nad prvkami štruktúry
- môžeme zadefinovať novú operáciu bez toho, aby sme museli zmeniť triedu prvku
- oddelíme algoritmus od objektovej štruktúry ktorou pracuje
- klient neinteraguje priamo s prvkom, namiesto toho pošle visitor objekt, ktorý následne vykoná operáciu nad prvkom

Visitor


Client
Visitor
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Notation: UML
ConcreteVisitor1
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
ConcreteVisitor2
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
Object structure
Element
accept(visitor : Visitor) : void
ConcreteElementA
accept(visitor : Visitor) : void
ConcreteElementB
accept(visitor : Visitor) : void

Zhrnutie

- adapter/bridge
- decorator
- facade
- proxy
- chain of responsibility
- command
- mediator
- observer
- strategy
- visitor