
# Programovanie v jazyku C#

Návrhové vzory

prednáška 10
Ing. Ján Magyar, PhD.
ak. rok. 2021/2022 ZS

# Návrhový vzor

- všeobecné znovupoužiteľné riešenie pre opakujúce sa problémy v softvérovom inžinierstve
- popis riešenia, nie samotné riešenie
- formalizovaný best practice
- nie sú nevyhnutné, ale zjednodušujú implementáciu

# Štruktúra návrhových vzorov

- definuje komponenty a ich úlohy
- definuje vzťah medzi komponentmi
- nešpecifikuje funkcionalitu (závisí od prípadu použitia)
- nešpecifikuje implementáciu (je to na programátorovi)

# Použitie návrhových vzorov

- pre časté problémy
- programovací jazyk môže ponúkať implicitné riešenie problému
- zvyčajne pre objektovo-orientované jazyky a programy
- osvedčený spôsob písania kvalitného softvéru

# Výhody použitia návrhových vzorov

- rýchlejší vývoj
- bezpečnejšie a overené riešenia
- rieši aj skryté problémy
- lepšia čitateľnosť a štruktúra riešenia
- rozdelenie do komponentov pre znovupoužitie kódu
- zdokumentované riešenie

# Dokumentácia návrhových vzorov

- meno a klasifikácia
- cieľ - prečo by sa mal použiť?
- alternatívne mená
- motivácia - ukážkový problém
- použiteľnosť - popis problémov pre ktoré sa vzor dá použiť
- štruktúra - diagram triedy
- členovia - zoznam komponentov

# Dokumentácia návrhových vzorov (pokračovanie)

- spolupráca - ako interagujú triedy a objekty
- dôsledky - výsledky, následky, kompromis
- implementácia - ukážkový prípad použitia
- ukážkový kód - ako sa implementuje v niektorom jazyku
- známe prípady použitia - reálne využitia
- podobné vzory - porovnanie s ďalšími vzormi

# Kritika návrhových vzorov

- môžu naznačiť chýbajúcu podporu v programovacom jazyku
- môžu byť implementované inými prístupmi
- zvyšuje zložitosť riešenia pri nevhodnom použití

# Typy návrhových vzorov

- kreačné - ako vytvoriť objekt?
- štrukturálne - ako realizovať vzťah medzi objektmi?
- behaviorálne - ako môžu komponenty komunikovať?
- konkurenčnosť - pre viacvláknové programy

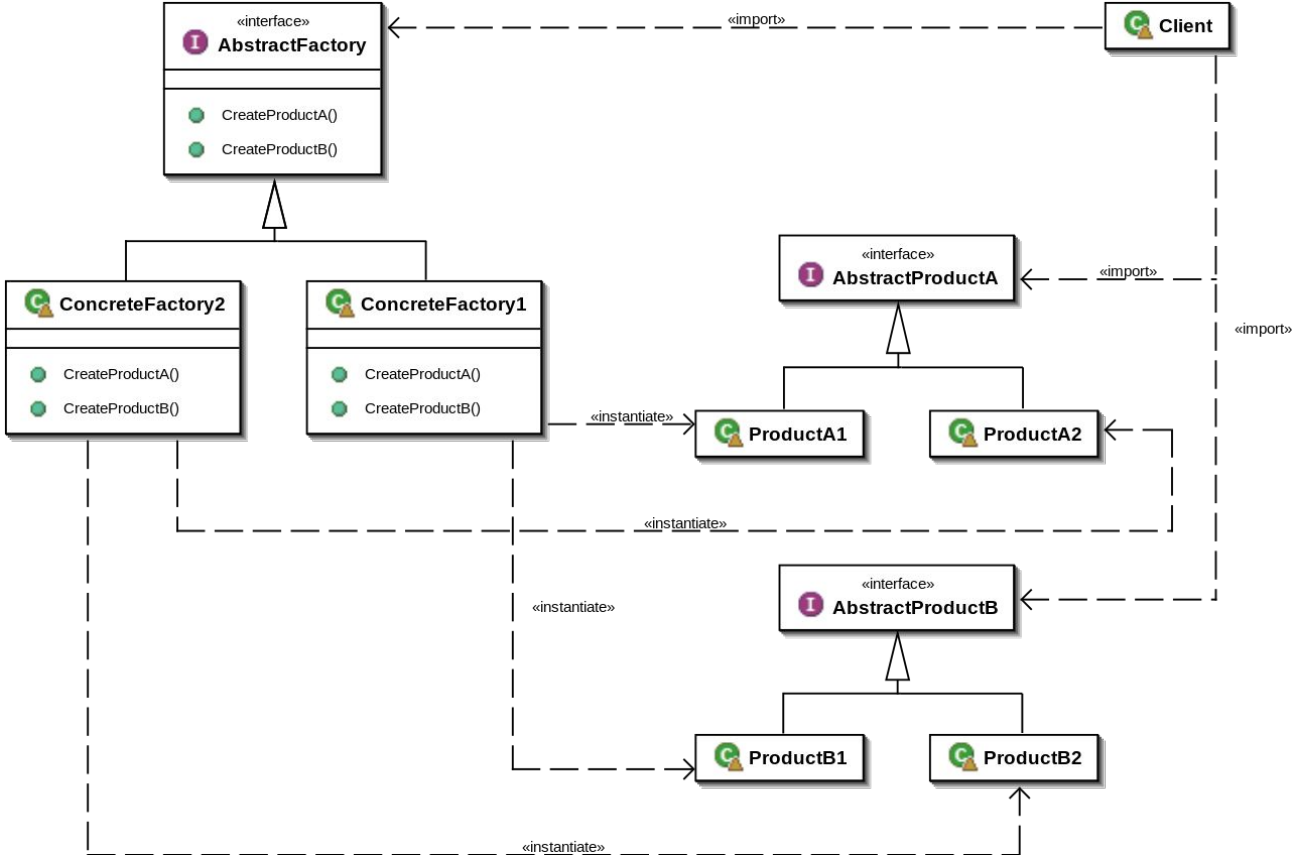
# Kreačné návrhové vzory

- abstract factory
- builder
- factory method
- prototype
- singleton

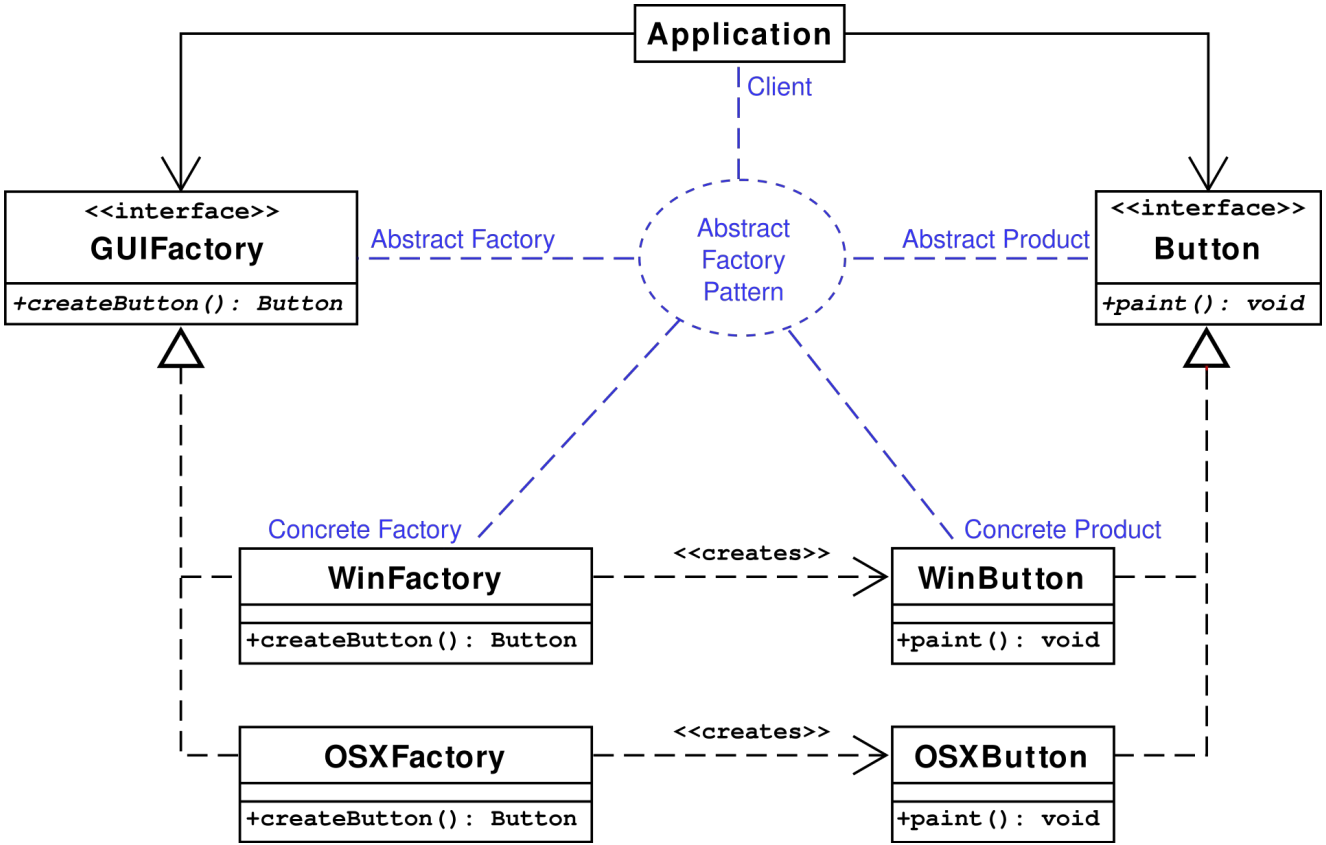
# Abstract factory

- poskytnúť rozhranie pre vytvorenie skupiny závislých objektov bez špecifikácie konkrétnej triedy
- Ako môže byť aplikácia nezávislá od toho, ako sa vytvoria jej objekty a objekty ktoré potrebuje?
- vytvorenie objektov je skryté v osobitnom objekte
- úloha vytvorenia objektov je delegovaná factory objektu

# Abstract factory



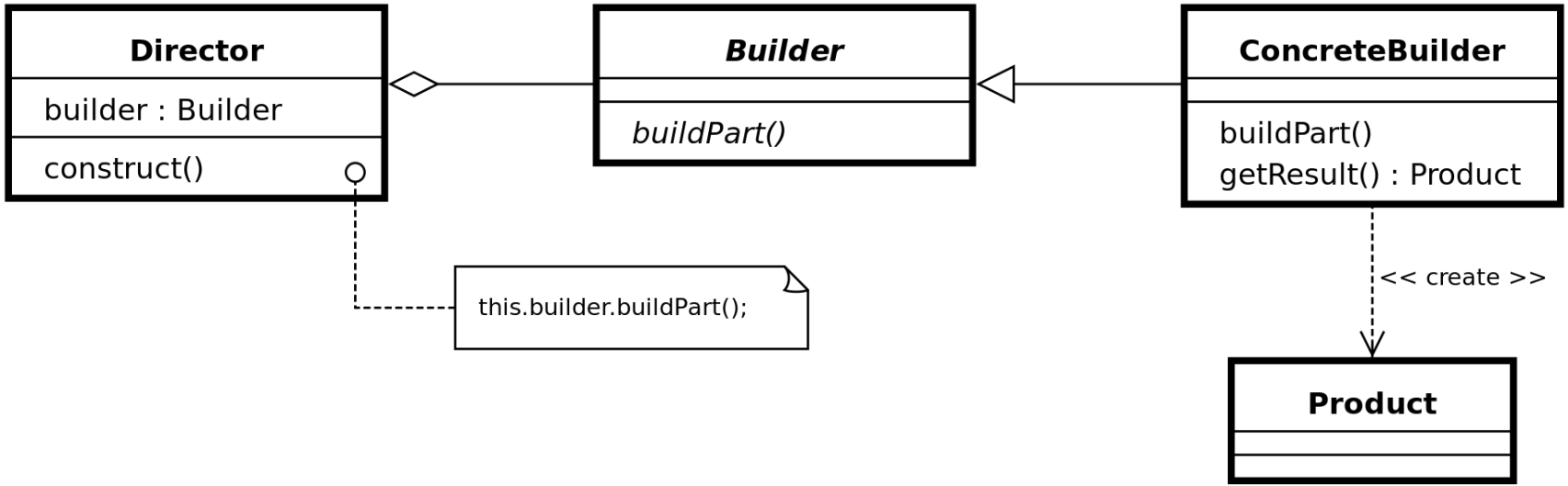
# Abstract factory



# Builder

- pre zložité objekty, oddelíme ich reprezentáciu od ich vytvorenia
- rovnaký proces môže vytvoriť rôzne reprezentácie
- Ako môžeme zjednodušiť triedu, ktorá obsahuje vytvorenie zložitého objektu?

# Builder


Director
builder : Builder
construct()
Builder
buildPart()
ConcreteBuilder
buildPart()
getResult() : Product
this.builder.buildPart();
<< create >>
Product

# Použitie vzoru Builder

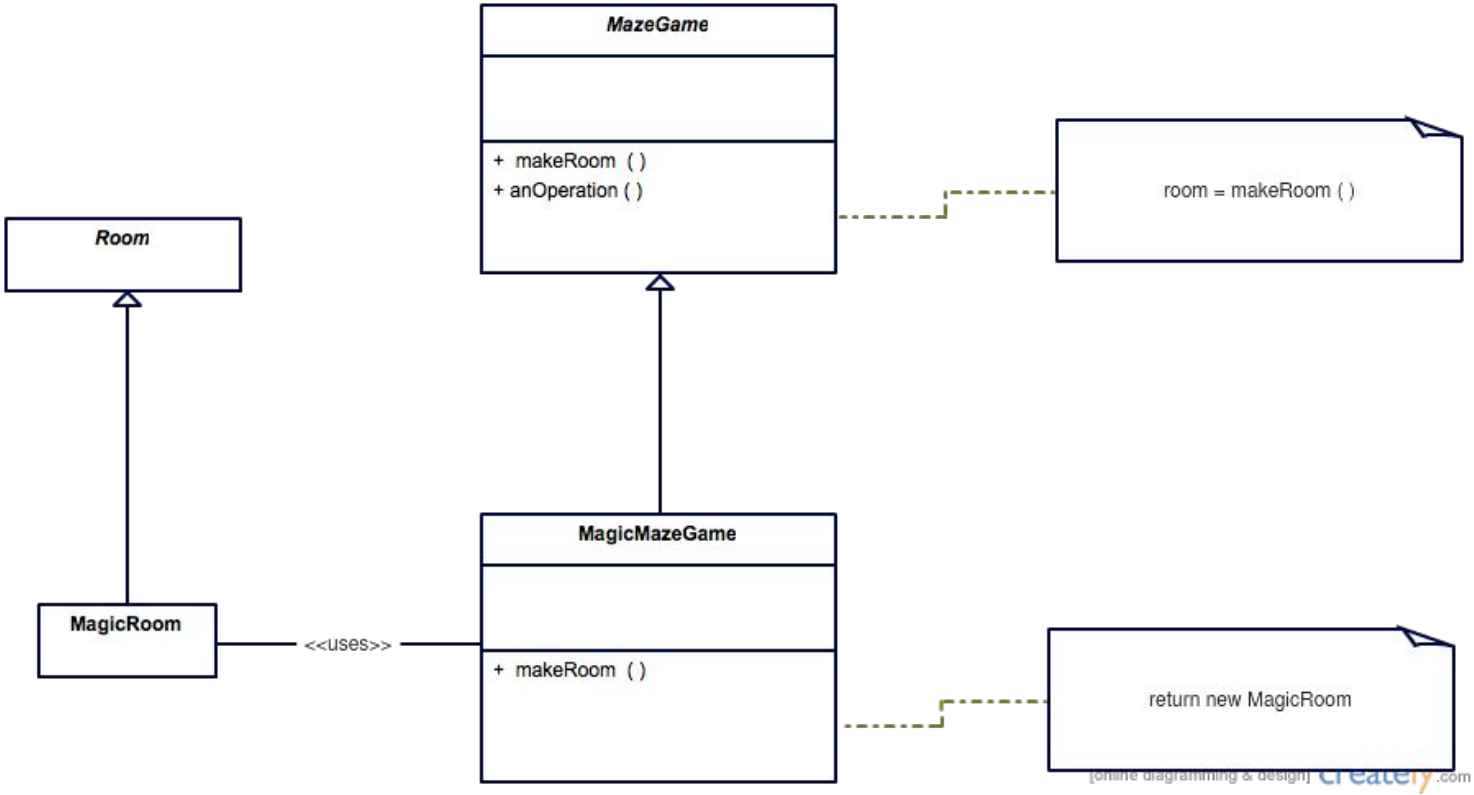
- môžeme meniť vnútornú reprezentáciu objektu
- enkapsulovaný kód pre vytvorenie a reprezentáciu
- môžeme kontrolovať proces vytvorenia

- musíme zadefinovať builder pre každý typ produktu
- triedy buildera môžeme meniť
- ťažší dependency injection

# Factory method

- rozhranie pre vytvorenie jediného objektu, ale podtriedy rozhodujú, inštanciu ktorej triedy majú vytvoriť
- vytvorenie inštancií je úlohou podtried
- osobitná operácia (factory method) je zodpovedná za vytvorenie objektu, objekt vytvoríme zavolaním tejto metódy

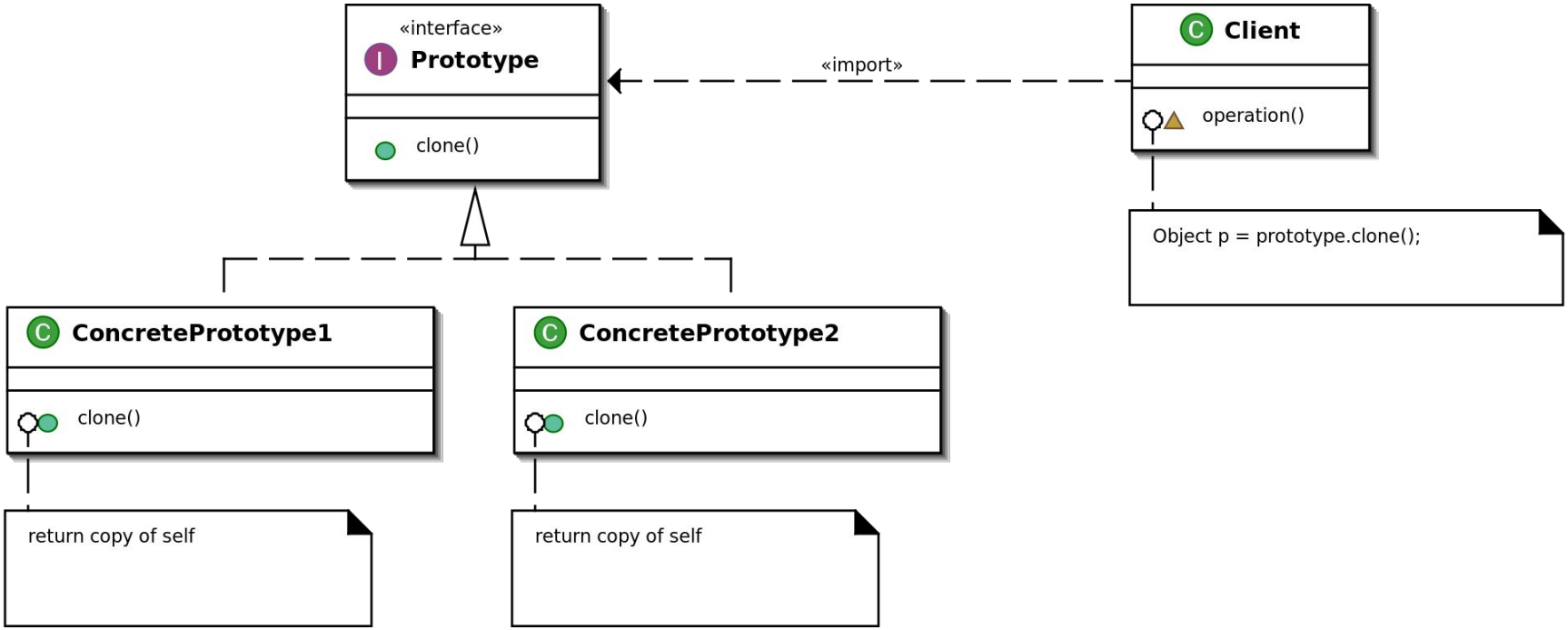
# Factory method



# Prototype

- objekty vytvárame na základe prototypovej inštancie
- vytváranie inštancií použitím už existujúceho objektu
- lepší výkon, menšia záťaž na pamäť
- môžeme špecifikovať počas behu, ktorý objekt sa má vytvoriť - dynamicky načítané triedy
- definujeme abstraktnú triedu s metódou `clone()`, ktorú implementujú konkrétne podtriedy

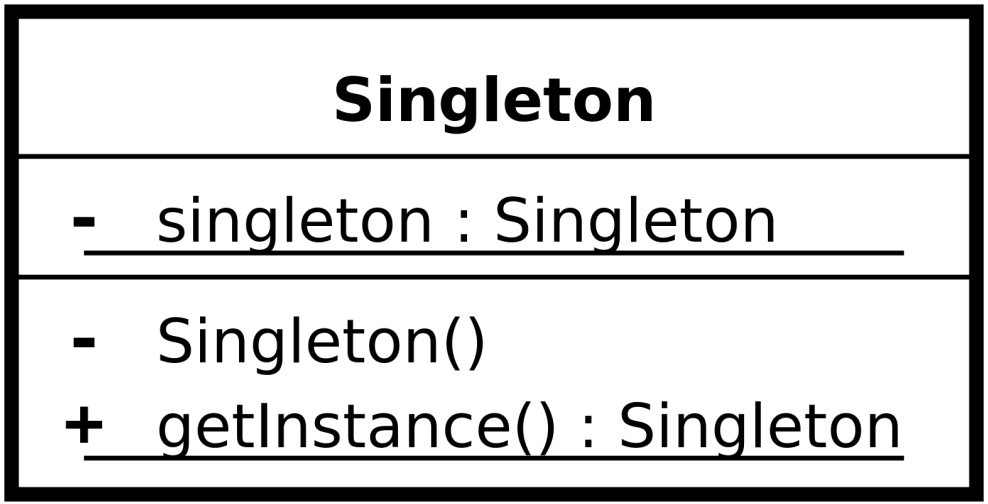
# Prototype


«interface»
Prototype
clone()
«import»
Client
operation()
Object p = prototype.clone();
ConcretePrototype1
clone()
return copy of self
ConcretePrototype2
clone()
return copy of self

# Singleton

- zabezpečuje, že trieda má iba jednu inštanciu (alebo žiadnu)
- poskytuje prístup k jedinej inštancii
- môžeme kontrolovať vytvorenie inštancie
- možné lazy initialization
- globálny stav

# Singleton


Singleton
- singleton : Singleton
- Singleton()
+ getInstance() : Singleton

# Ďalšie kreačné návrhové vzory

- dependency injection
- lazy initialization
- multiton
- object pool
- resource acquisition is initialization

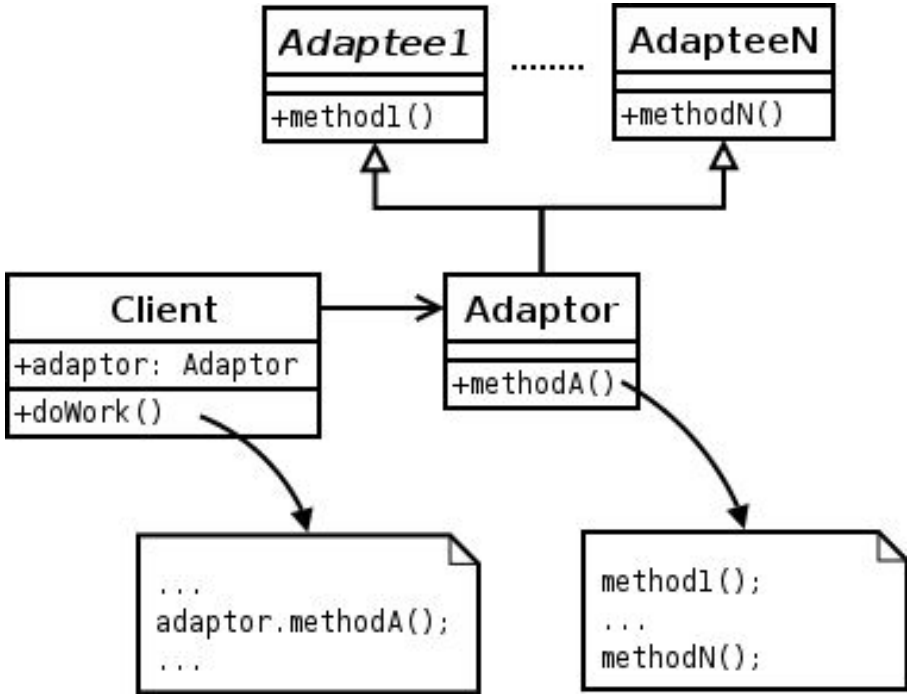
# Štrukturálne vzory

- adapter
- bridge
- composite
- 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

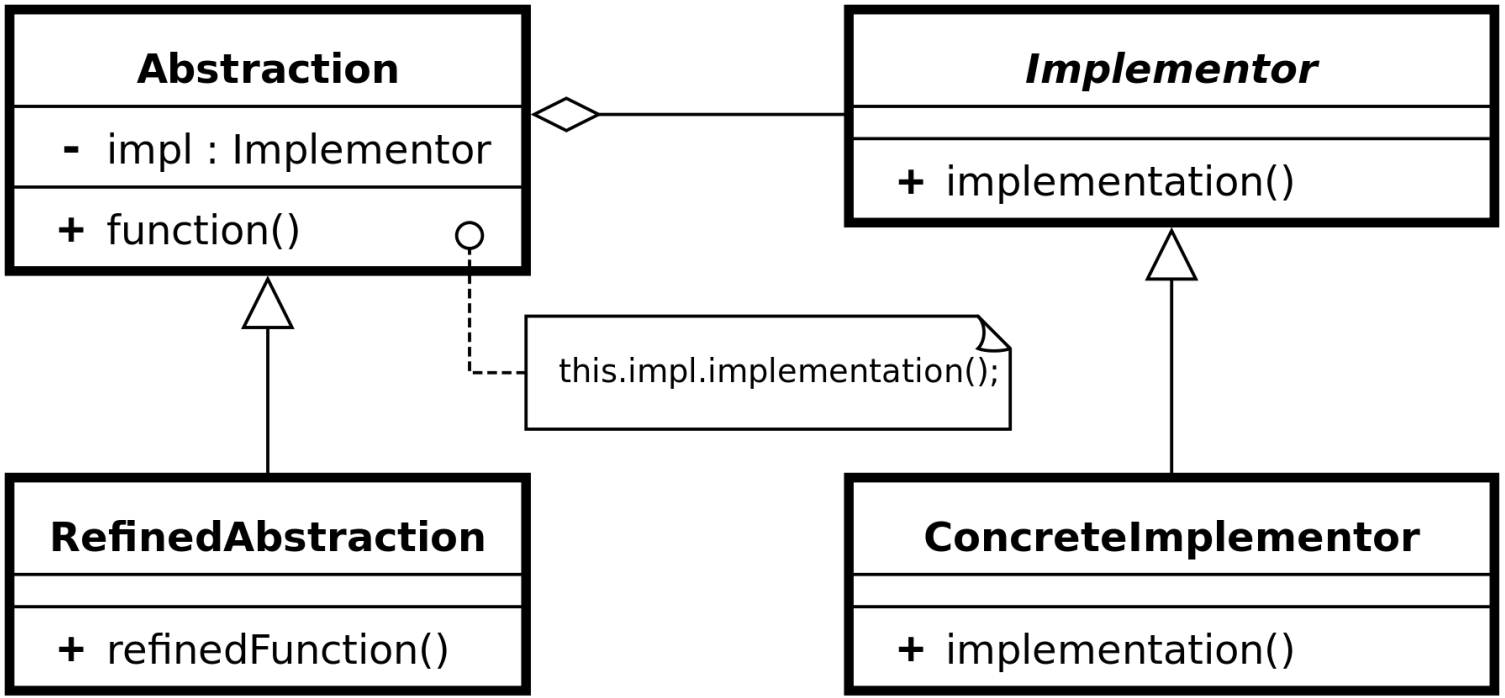
# Adapter


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

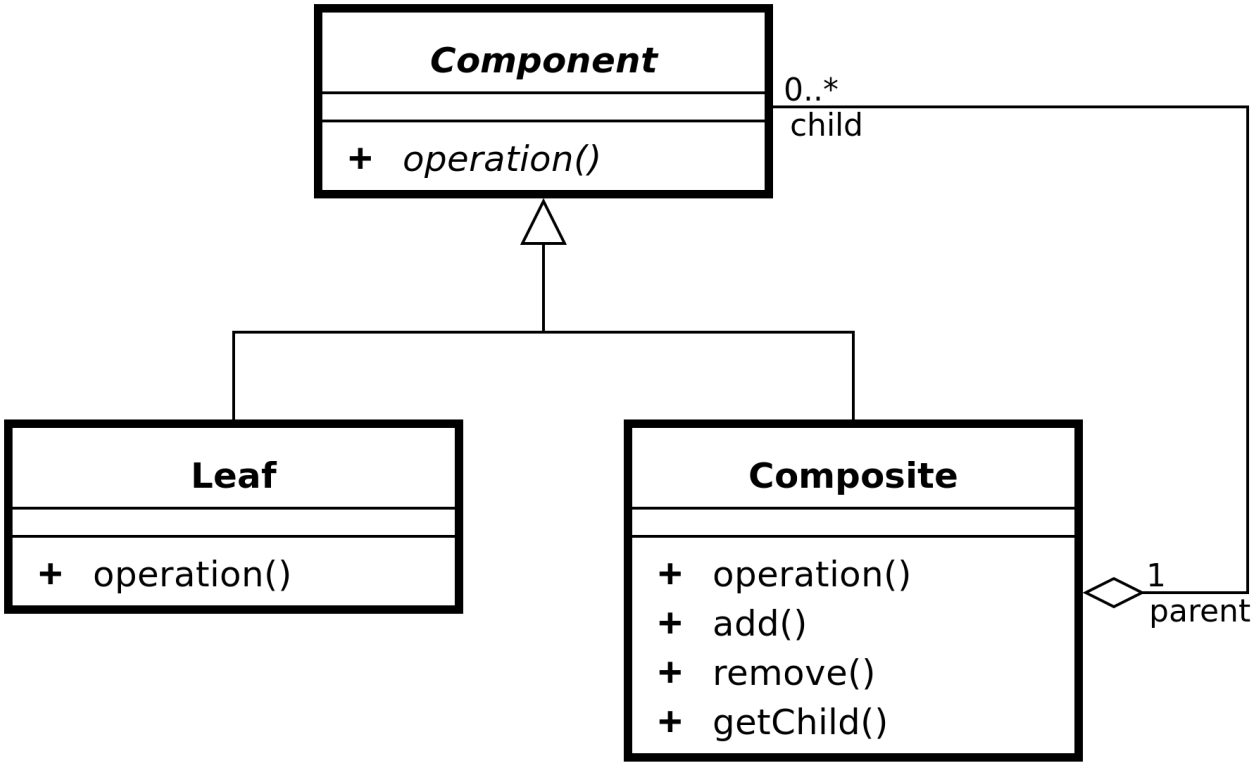
# Bridge


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

# Composite

- rozdelí objekty do stromovej štruktúry
- individuálne objekty a ich kompozity sa používajú jednotne
- definujeme zjednotené rozhranie `Component`, ktoré je implementované časťami objektu aj celkovým objektom
- kompozit prepošle požiadavky svojim potomkom

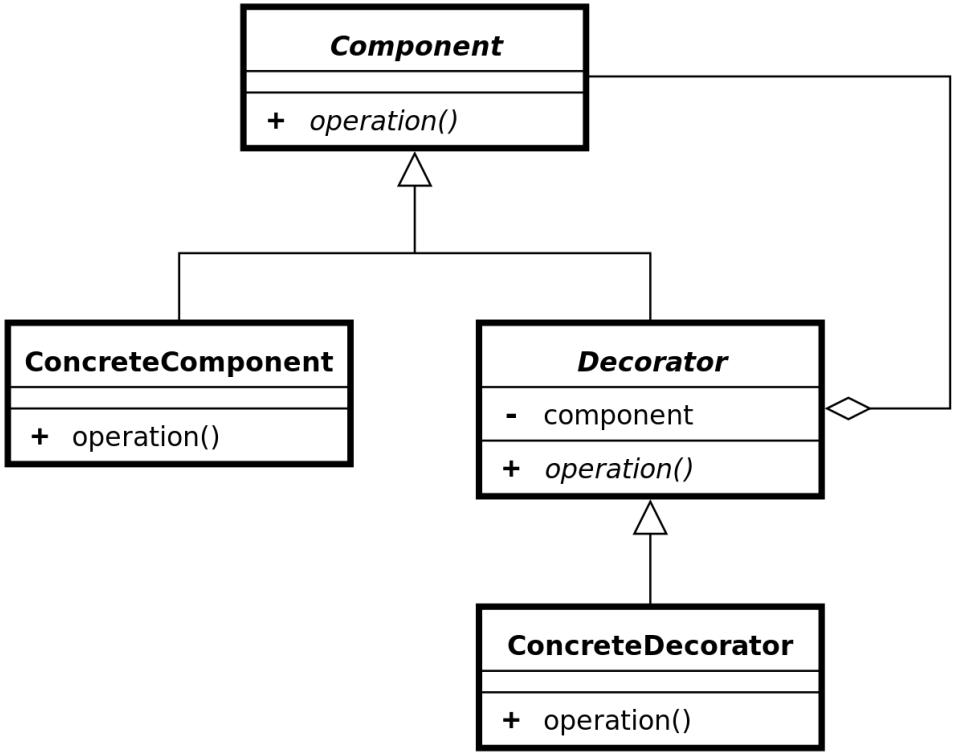
# Composite


Component
+ operation()
0..*
child
Leaf
+ operation()
Composite
+ operation()
+ add()
+ remove()
+ getChild()
1
parent

# 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

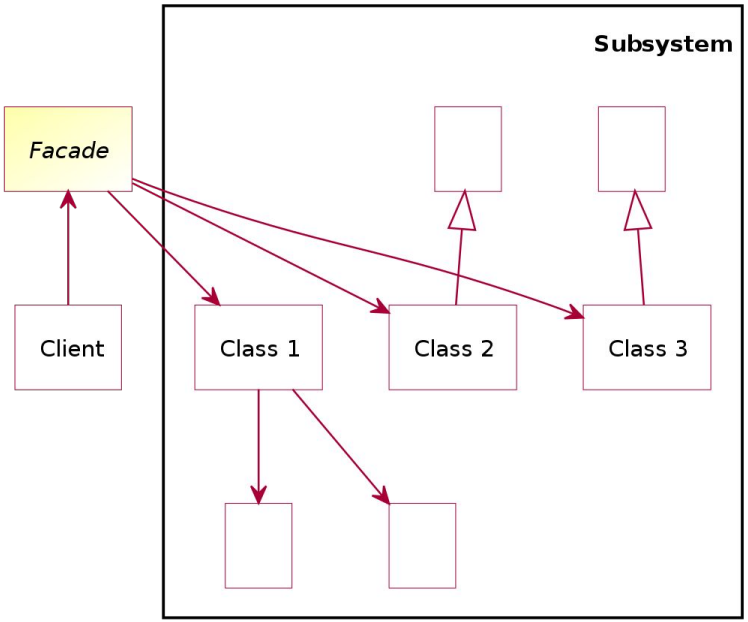
# Decorator

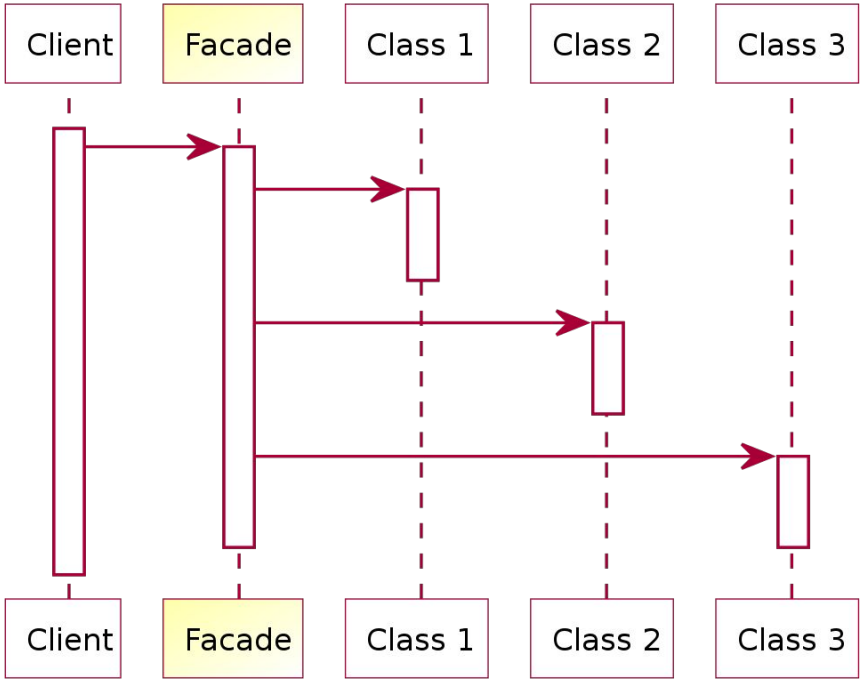

Component
+ operation()
ConcreteComponent
+ operation()
Decorator
- component
+ operation()
ConcreteDecorator
+ operation()

# Facade

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

# Facade

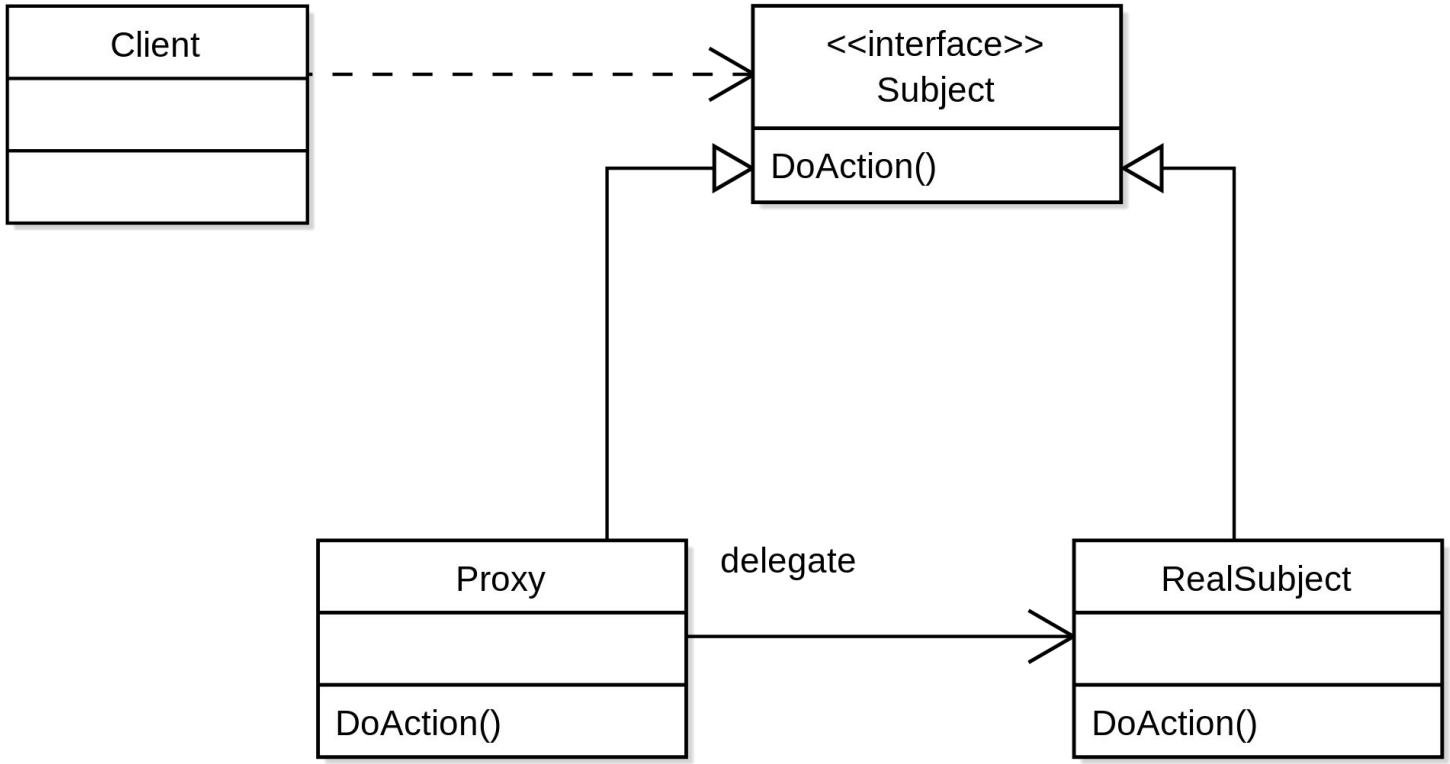

Subsystem
Facade
Client
Class 1
Class 2
Class 3


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()

# Ďalšie štrukturálne návrhové vzory

- extension object
- flyweight
- front controller
- marker
- module
- twin

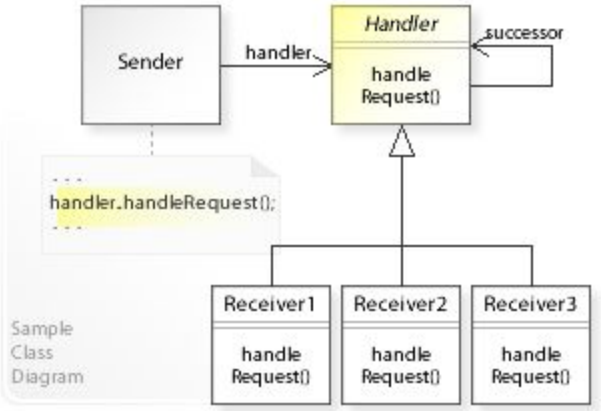
# Behaviorálne vzory

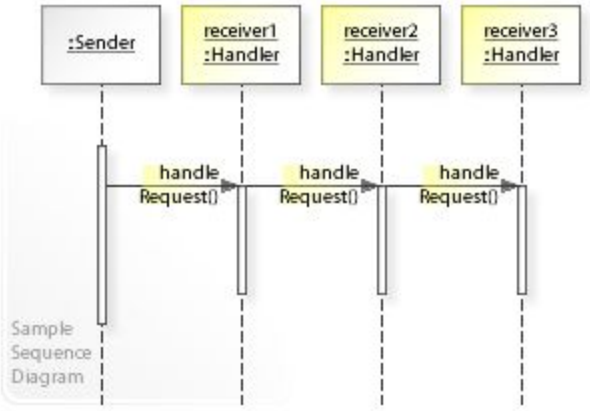
- chain of responsibility
- command
- iterator
- mediator
- observer and publish/subscribe
- strategy
- visitor

# Chain of responsibility

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

# Chain of responsibility

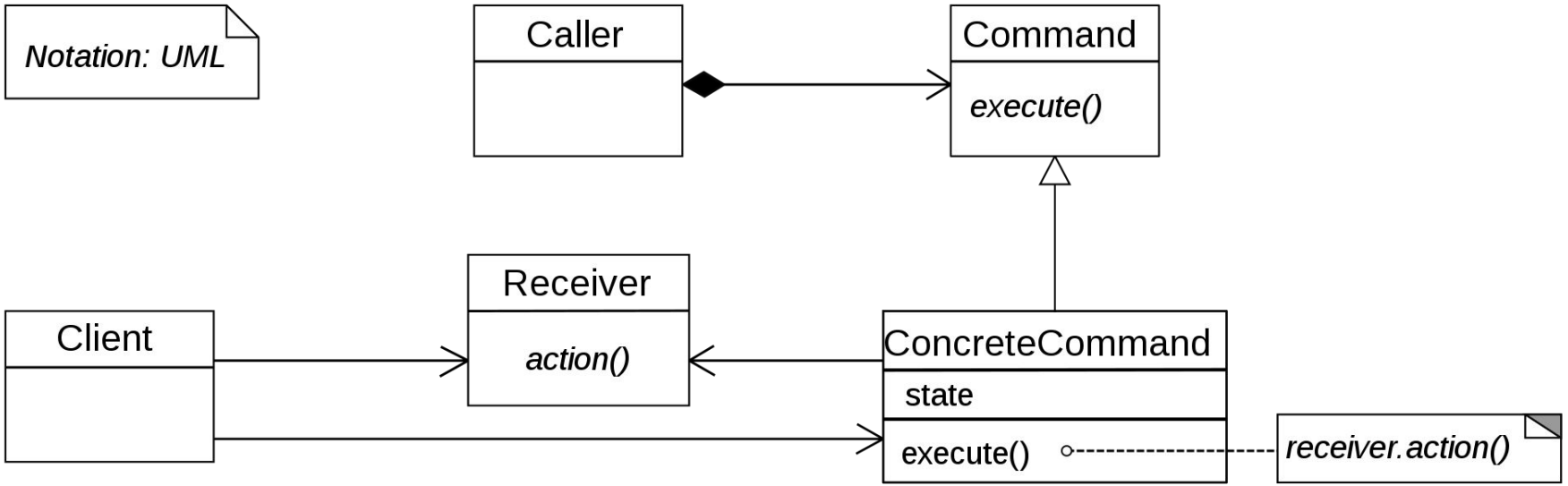


# Command

- 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

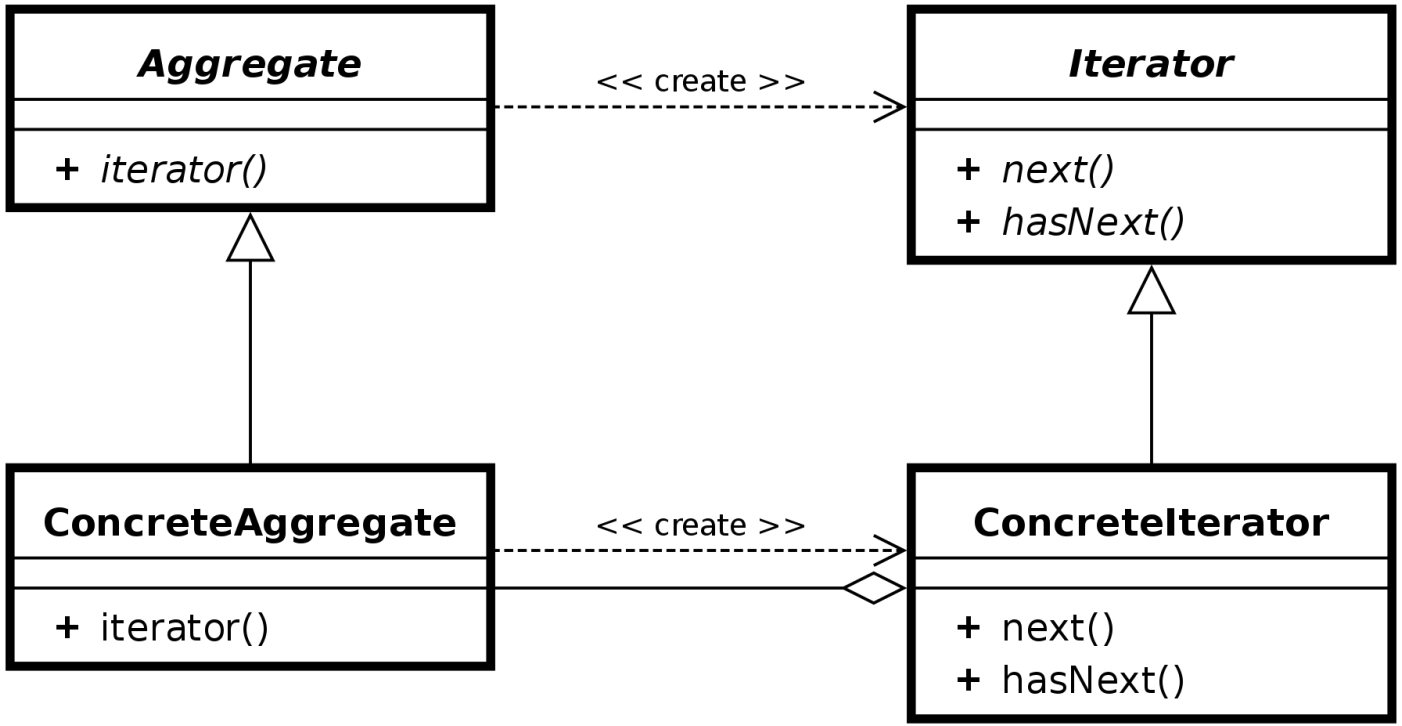
# Command


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

# Iterator

- poskytuje spôsob prístupu k prvkom agregátu sekvenčne bez odhalenia vnútornej reprezentácie
- oddelí algoritmus od kontajnera
- aj keď poskytuje jednotný prístup, nie vždy je optimálny
- trieda iterátor zapuzdrí prístup k a prechádzanie agregátom, klient používa iterátor

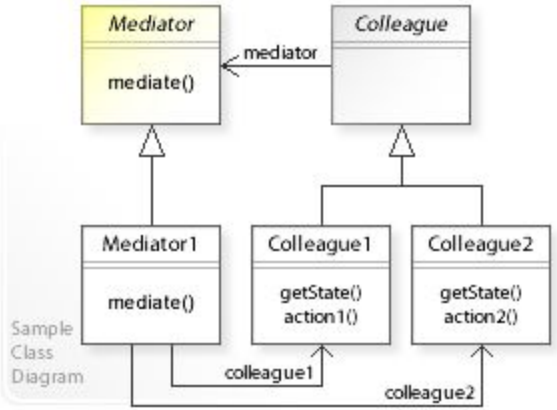
# Iterator

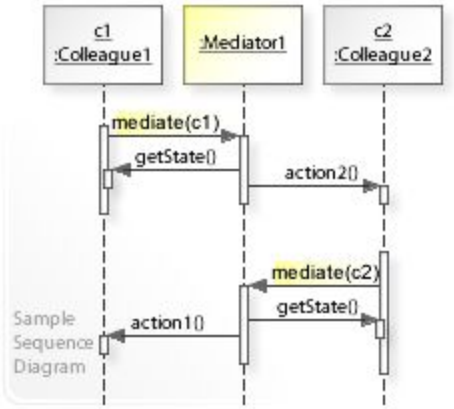

Aggregate
+ iterator()
<< create >>
Iterator
+ next()
+ hasNext()
ConcreteAggregate
+ iterator()
<< create >>
ConcreteIterator
+ next()
+ hasNext()

# 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

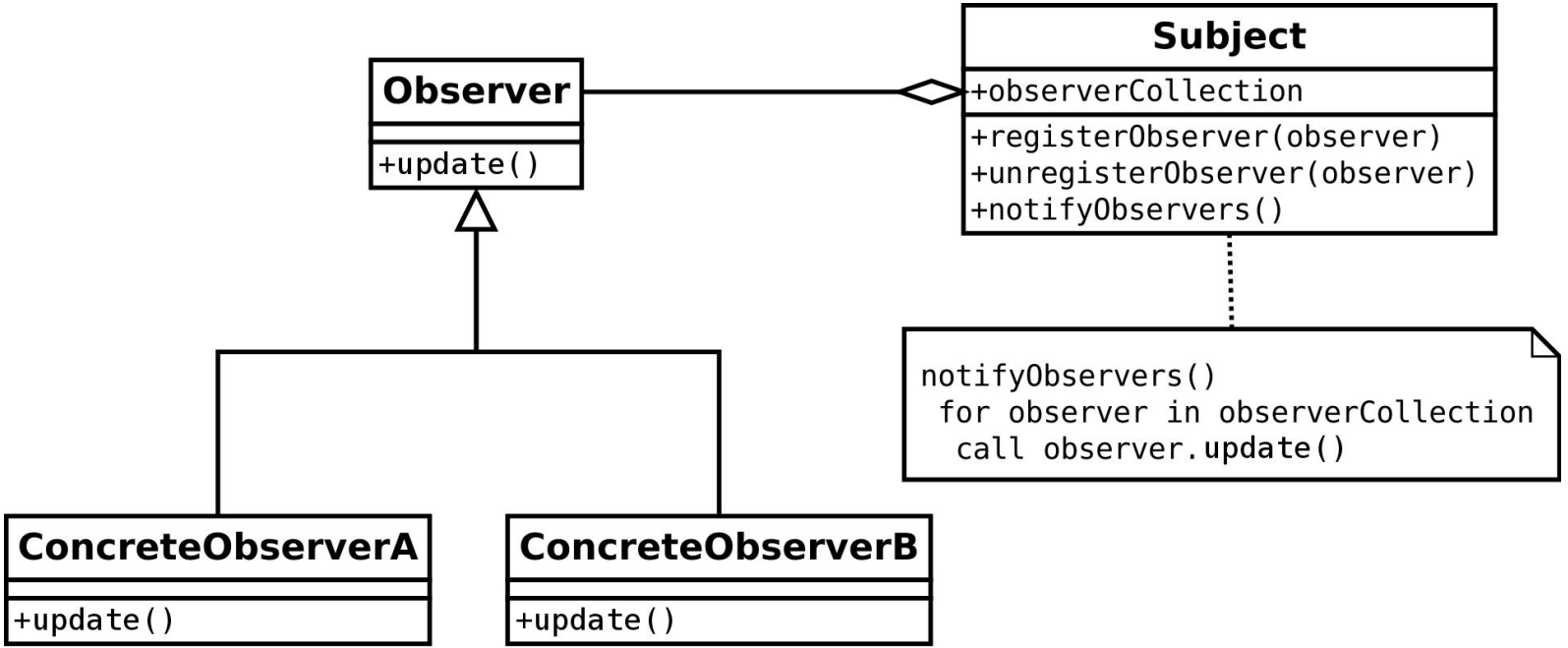

Mediator
mediate()
mediator
Colleague
Mediator1
mediate()
Colleague1
getState()
action1()
Colleague2
getState()
action2()
colleague1
colleague2
Sample
Class
Diagram


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 spracujú zmenu vhodným spôsobom

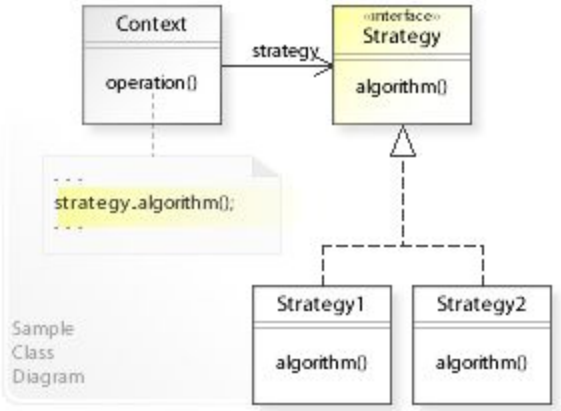
# Observer

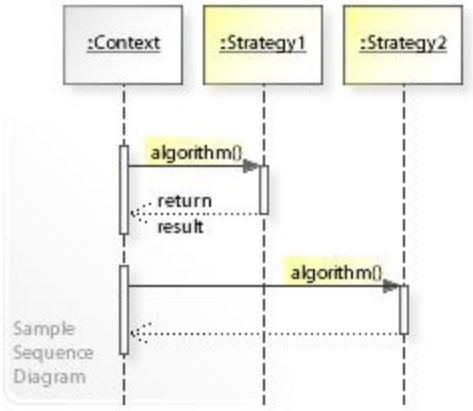

Subject
+observerCollection
+registerObserver(observer)
+unregisterObserver(observer)
+notifyObservers()
Observer
+update()
notifyObservers()
for observer in observerCollection
call observer.update()
ConcreteObserverA
+update()
ConcreteObserverB
+update()

# 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, ktorý má použiť a následne zavolá zodpovedajúcu metódu

# Strategy

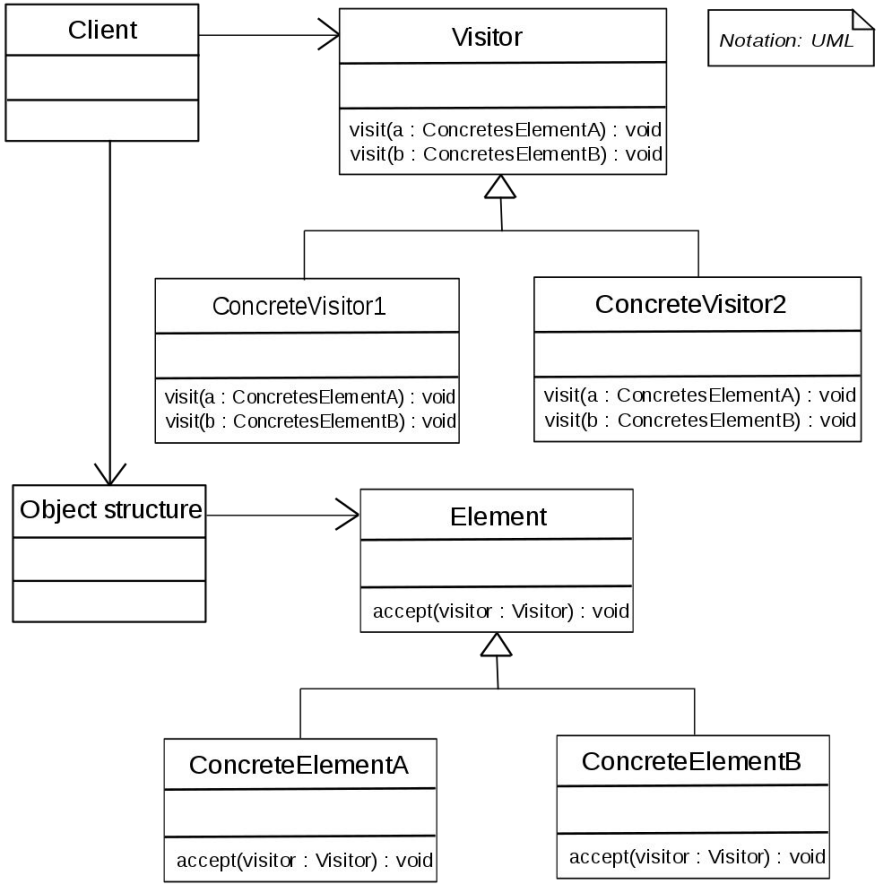

Context
operation()
strategy
«interface»
Strategy
algorithm()
strategy.algorithm();
Strategy1
algorithm()
Strategy2
algorithm()
Sample
Class
Diagram


: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
Notation: UML
visit(a : ConcretesElementA) : void
visit(b : ConcretesElementB) : void
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

# Ďalšie behaviorálne návrhové vzory

- blackboard
- memento
- null object
- servant
- specification
- state
- template method

# otázky?