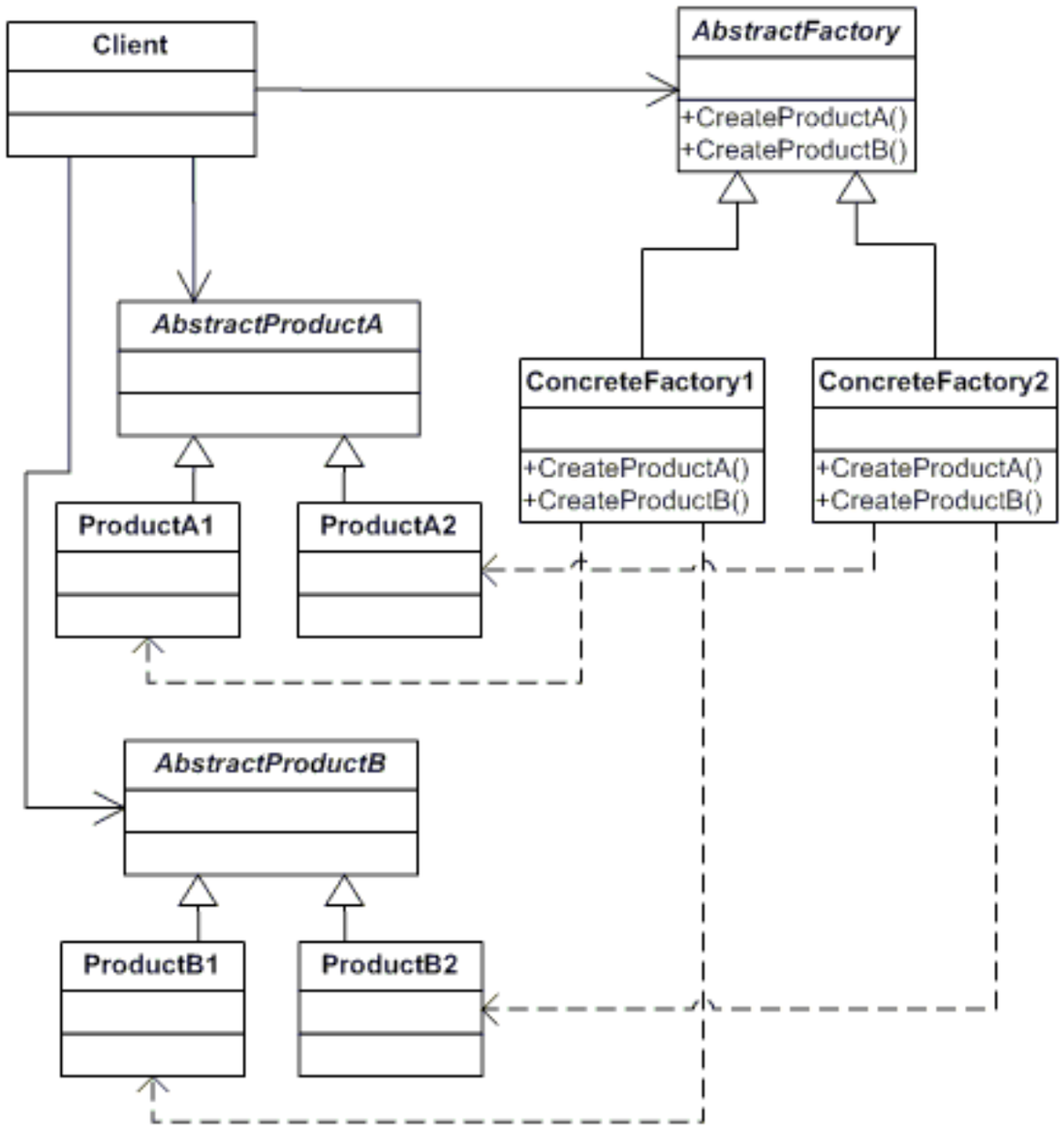
Návrhové vzory II

OMO, LS 2013/2014

Abstract Factory

Aneb: Bude toho víc



Abstract Factory - Vlastnosti

- Poskytuje rozhraní pro vytváření příbuzných nebo závislých objektů.
- Není nutno specifikovat konkrétní třídy (realizováno až v potomcích).
- Odděluje tedy klienta od samotného procesu vytváření instancí.

Abstract Factory – Příklad

- Problém: Potřebujeme navrhnout objekt, který bude “obsahovat” následující informace na základě požadavků uživatele:

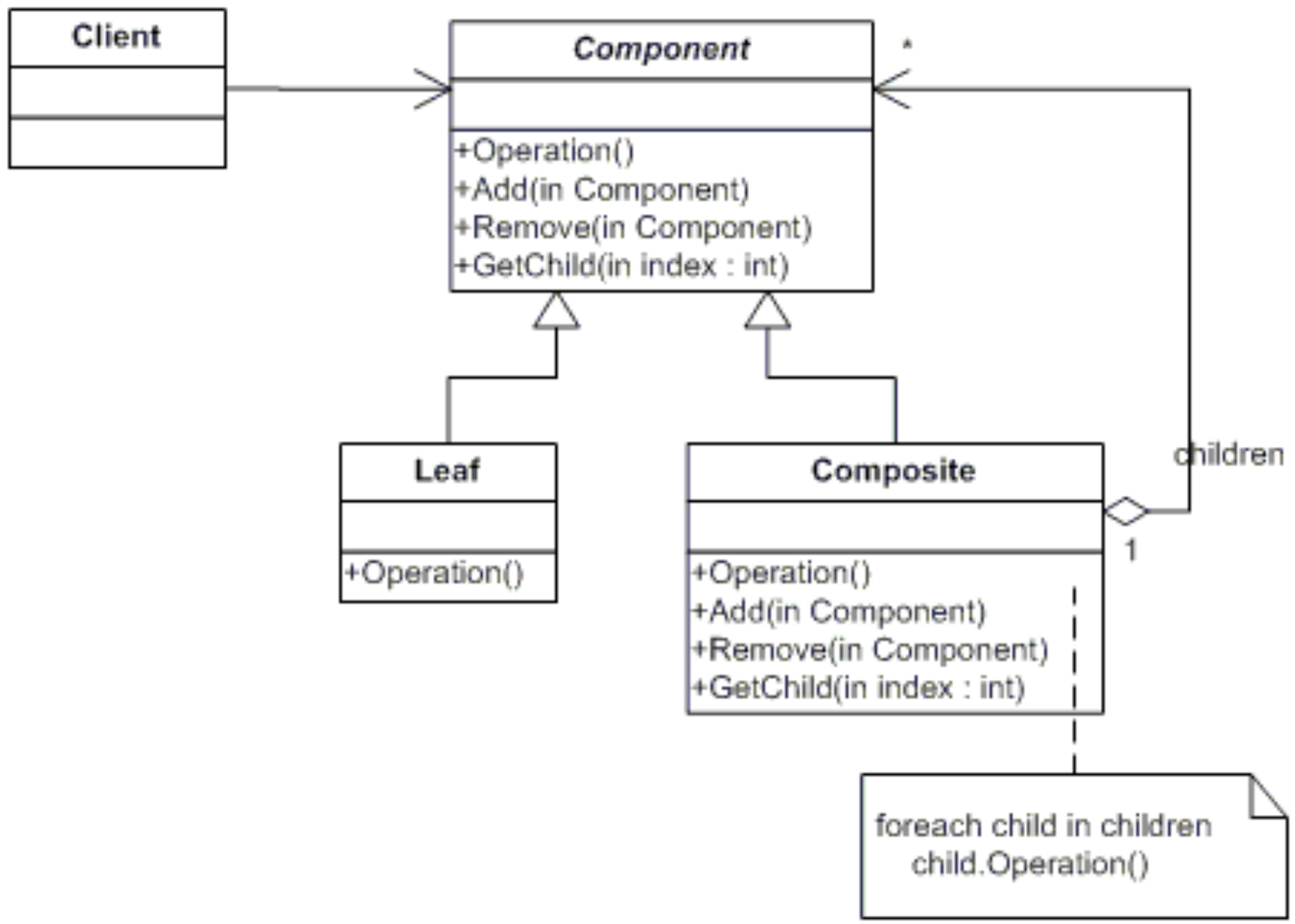
1. Výrobce telefonu:
   a. Nokia
   b. Samsung
   c. HTC
2. Typ telefonu:
   a. Chytrý telefon
   b. Hloupý telefon

Abstract Factory - Příklad

- Na základě těchto informací tedy potřebujeme tři továrny výrobců (HTC, Nokia, Samsung) telefonů a dvě množiny produktů (smart, dumb).

Composite

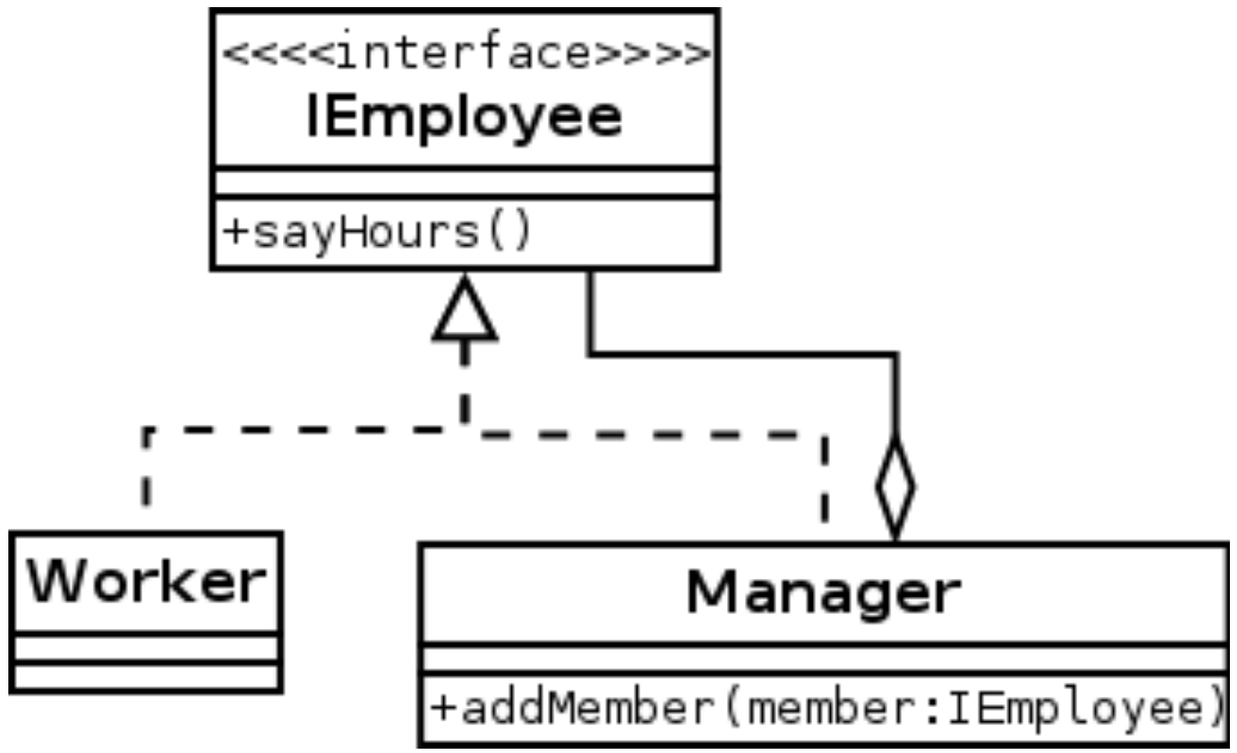
Aneb: Stromová struktura



Composite - Vlastnosti

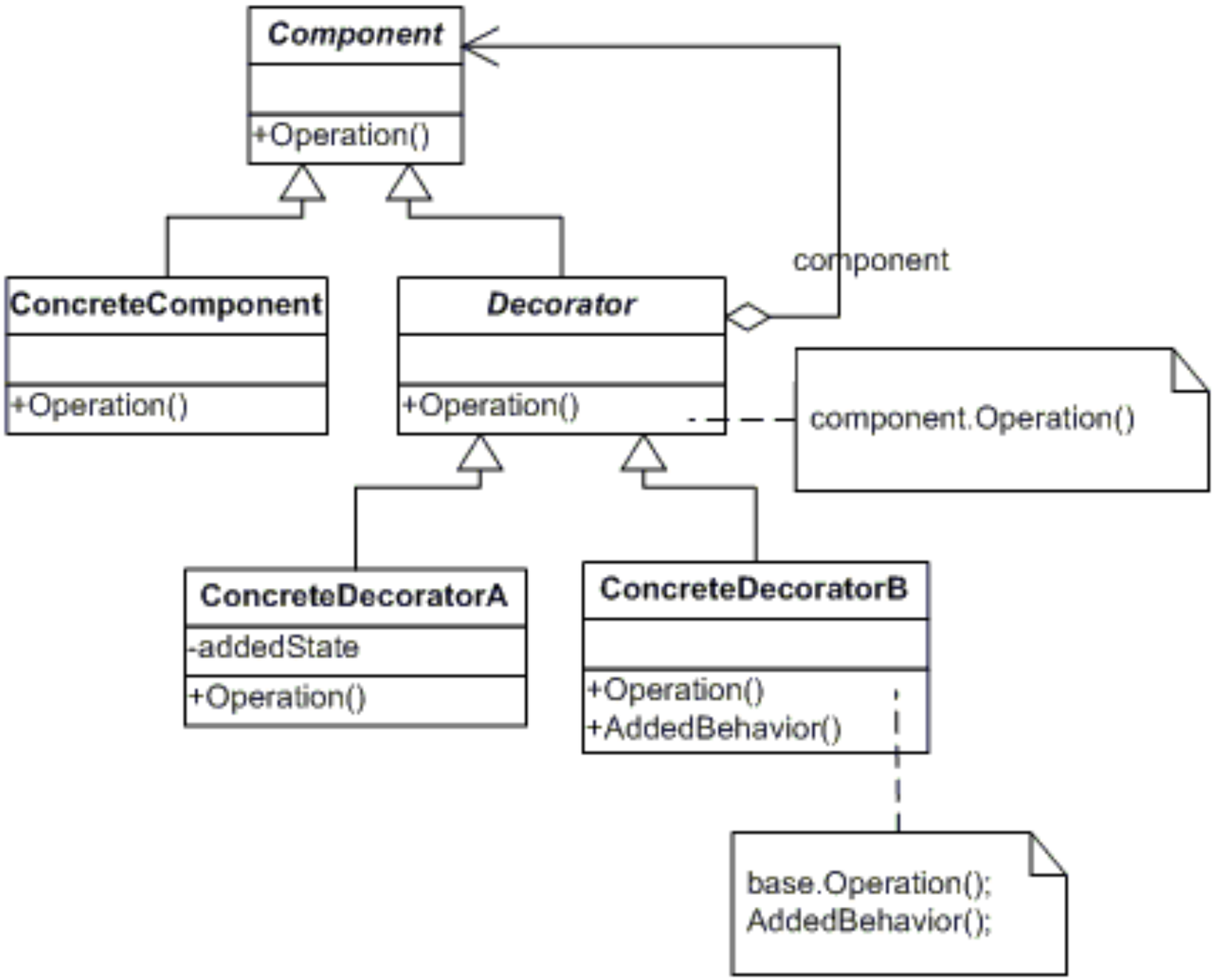
- Slouží k reprezentaci stromové struktury objektů.
- Výhodou je, že rozhraní jednotlivého uzlu i celkou jsou stejná.
- Je velice vhodný například pro reprezentaci struktury grafických objektů, souborového systému apod.

Composite - Příklad



Decorator

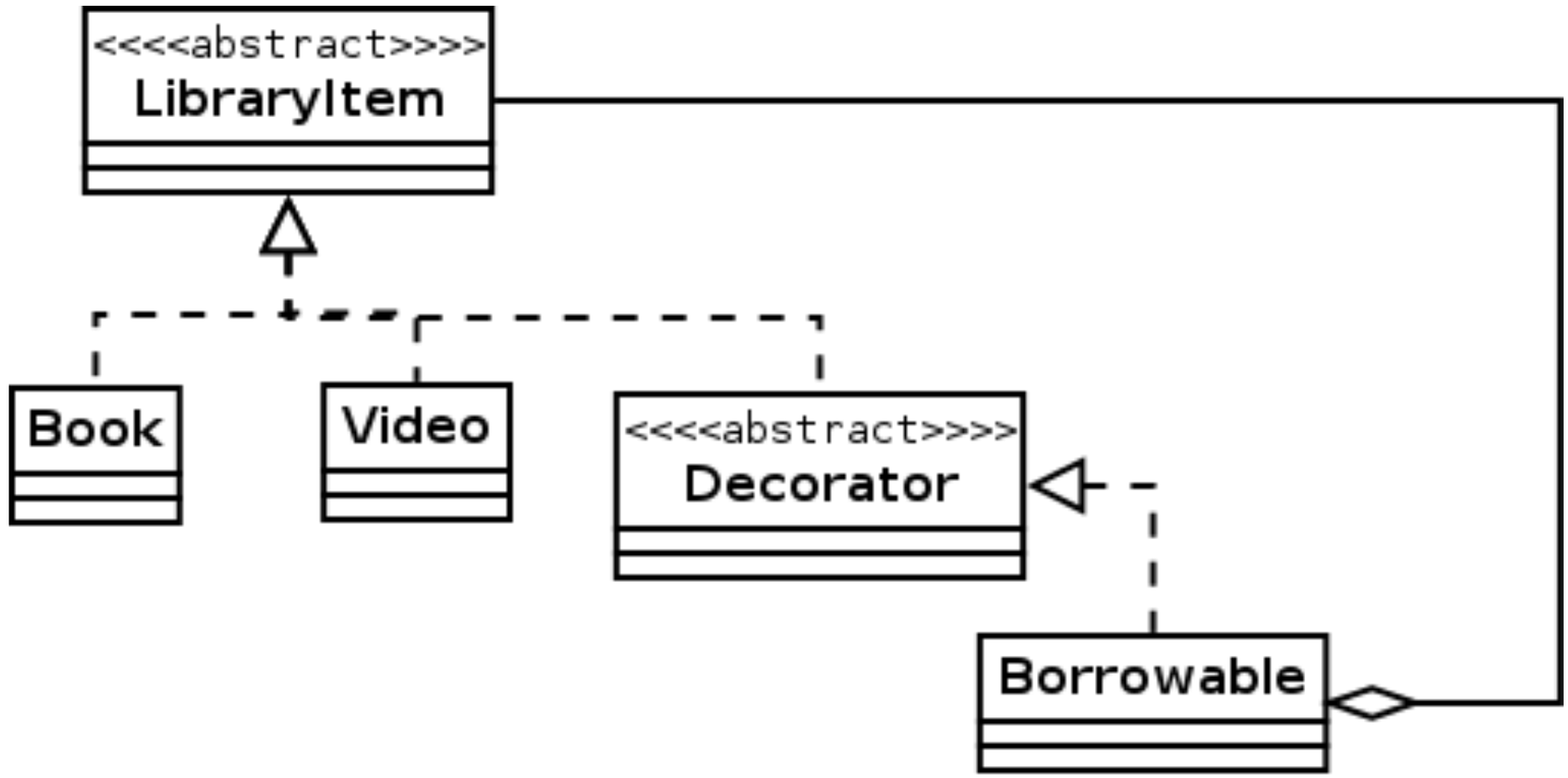
Aneb: Příliš mnoho druhů tříd



Decorator - Vlastnosti

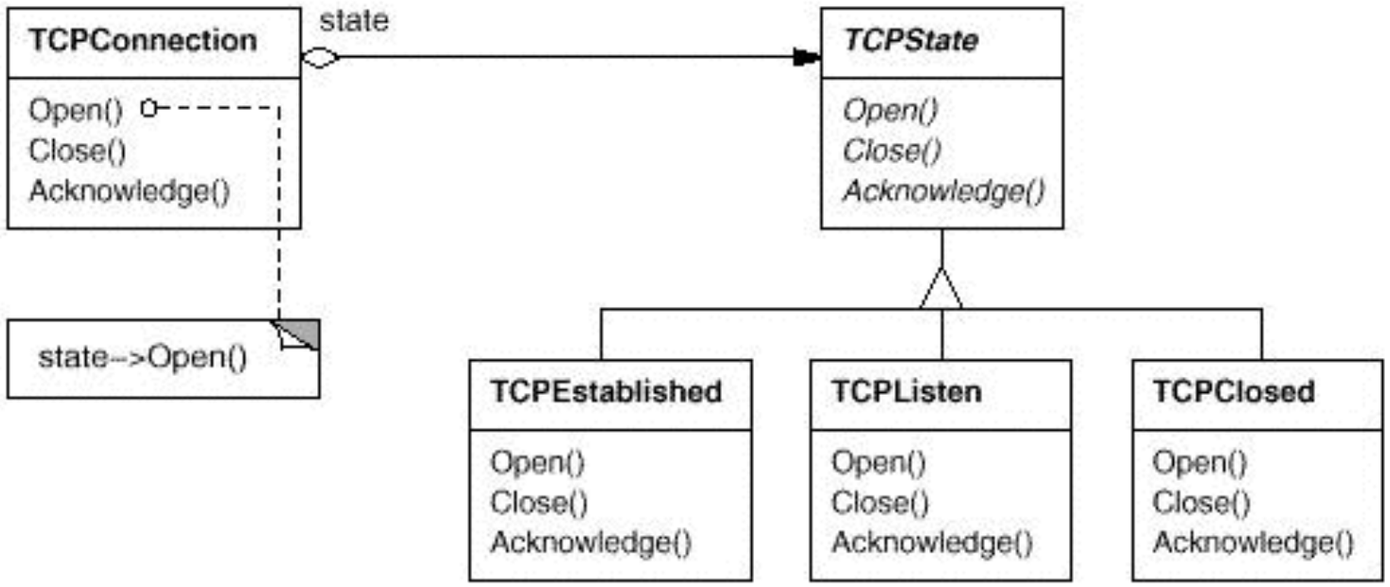
- Umožňuje dynamicky přidávat další vlastnosti k dekorovanému objektu.
- Může poskytovat dobrou alternativu k dědění.
- Konstruktor dekorátoru převezme dekorovaný objekt jako svůj parametr a ozdobí jej funkčností.

Decorator - Příklad



State Pattern

- Motivace: Chceme aby se objekt choval různě v závislosti na jeho vnitřním stavu



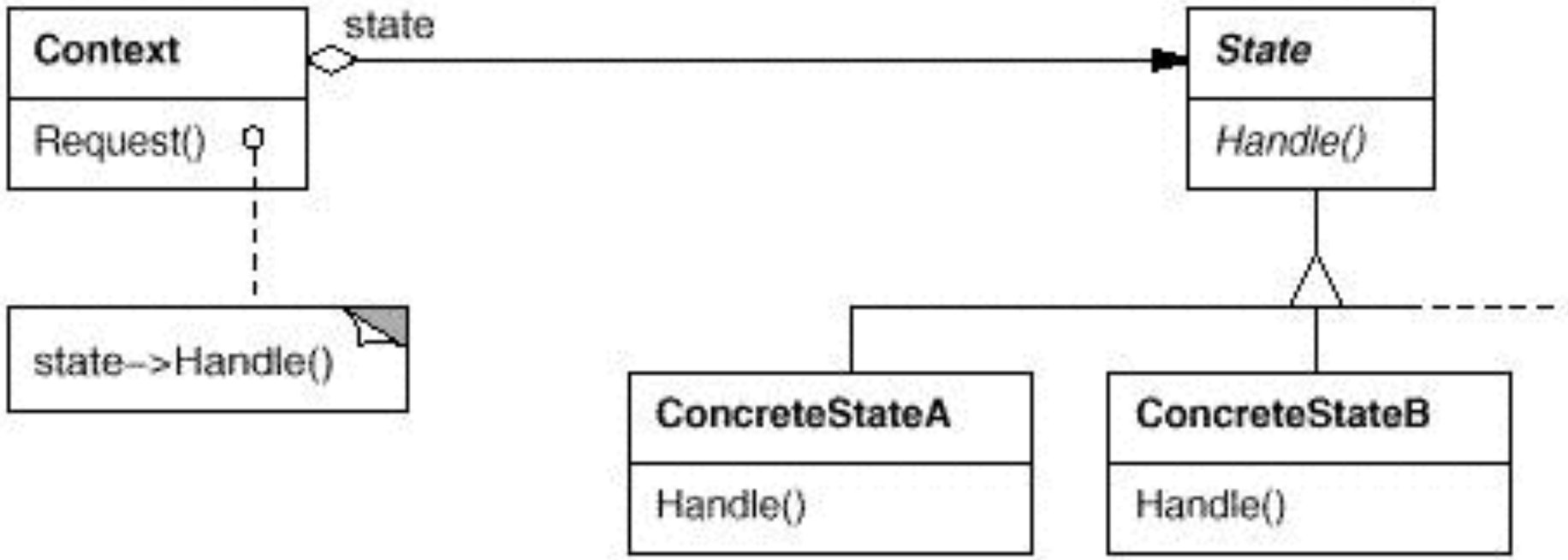
Zdroj: http://userpages.umbc.edu/~tarr/dp/lectures/StateStrategy.pdf

State Pattern

- Kdy mám State pattern použít?
  - Chování objektu závisí na jeho vnitřním stavu
  - Metody obsahují velké množství podmínek závislých na aktuálním vnitřním stavu. State pattern přesune jednotlivé větve do samostatných objektů.

State Pattern

- Diagram tříd

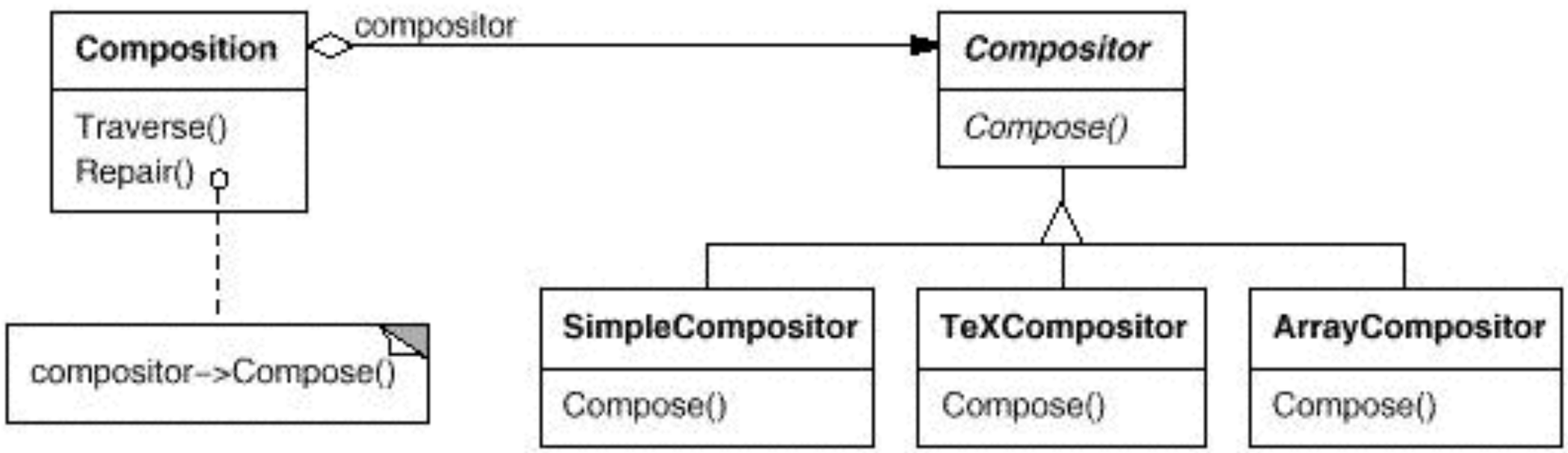


State Pattern

- Výhody
  - Přenese veškeré chování v daném stavu do jednoho objektu (Stav = Objekt)
  - Zpřehlední kód
  - Zjednodušší rozšiřování kódu o další stavy
- Nevýhody
  - Vyšší počet objektů (tříd)

Strategy Pattern

- Motivace:
  - Mějme skupinu algoritmů, každý je zapouzdřen, možnost libovolné jejich výměny. Strategy nám umožní změnu algoritmu bez ohledu na clienta.

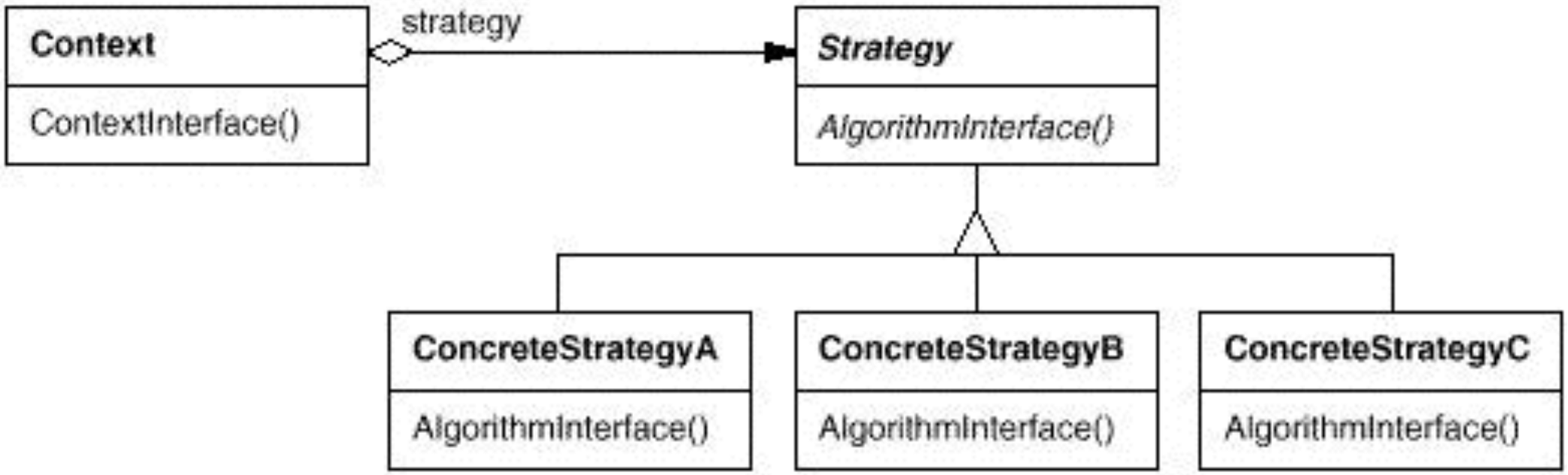


Strategy Pattern

- Kdy mám Strategy pattern použít?
  - Mnoho podobných tříd se liší pouze chováním
  - Potřebuji několik variant k řešení stejného problému
  - Algoritmus používá data o kterých client nemusí vědět
  - Třída obsahuje velké množství „ifů“

Strategy Pattern

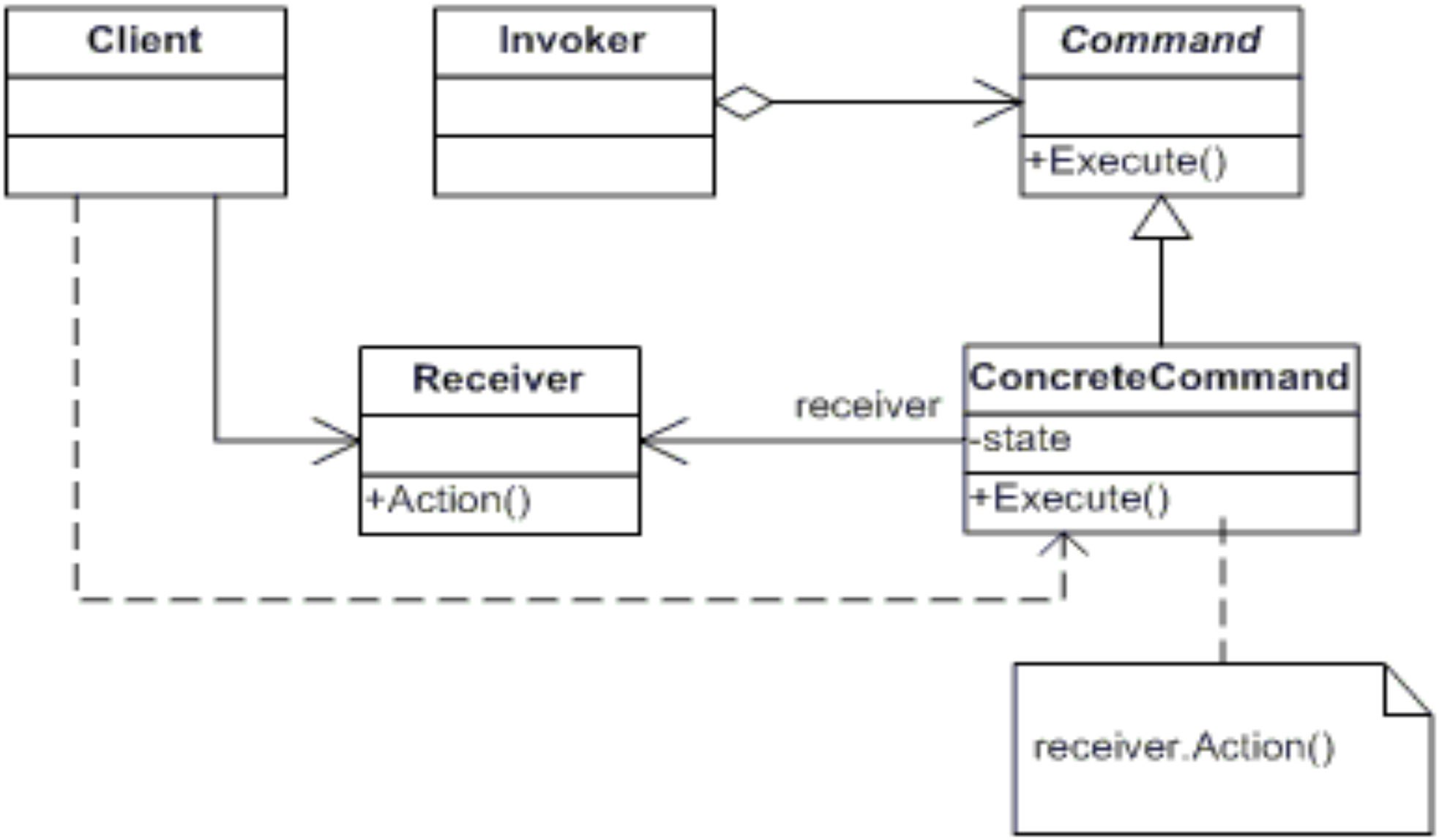
- Diagram Tříd



Strategy Pattern

- Výhody
  - Představuje alternativu k dědění abychom získali několik různých chování
  - Eliminuje velké množství „ifů“
  - Poskytuje snadnou změnu implementace daného problému
- Nevýhody
  - Vyšší počet objektů (tříd)
  - Všechny algoritmy (strategie) musí dodržovat stejné rozhraní!

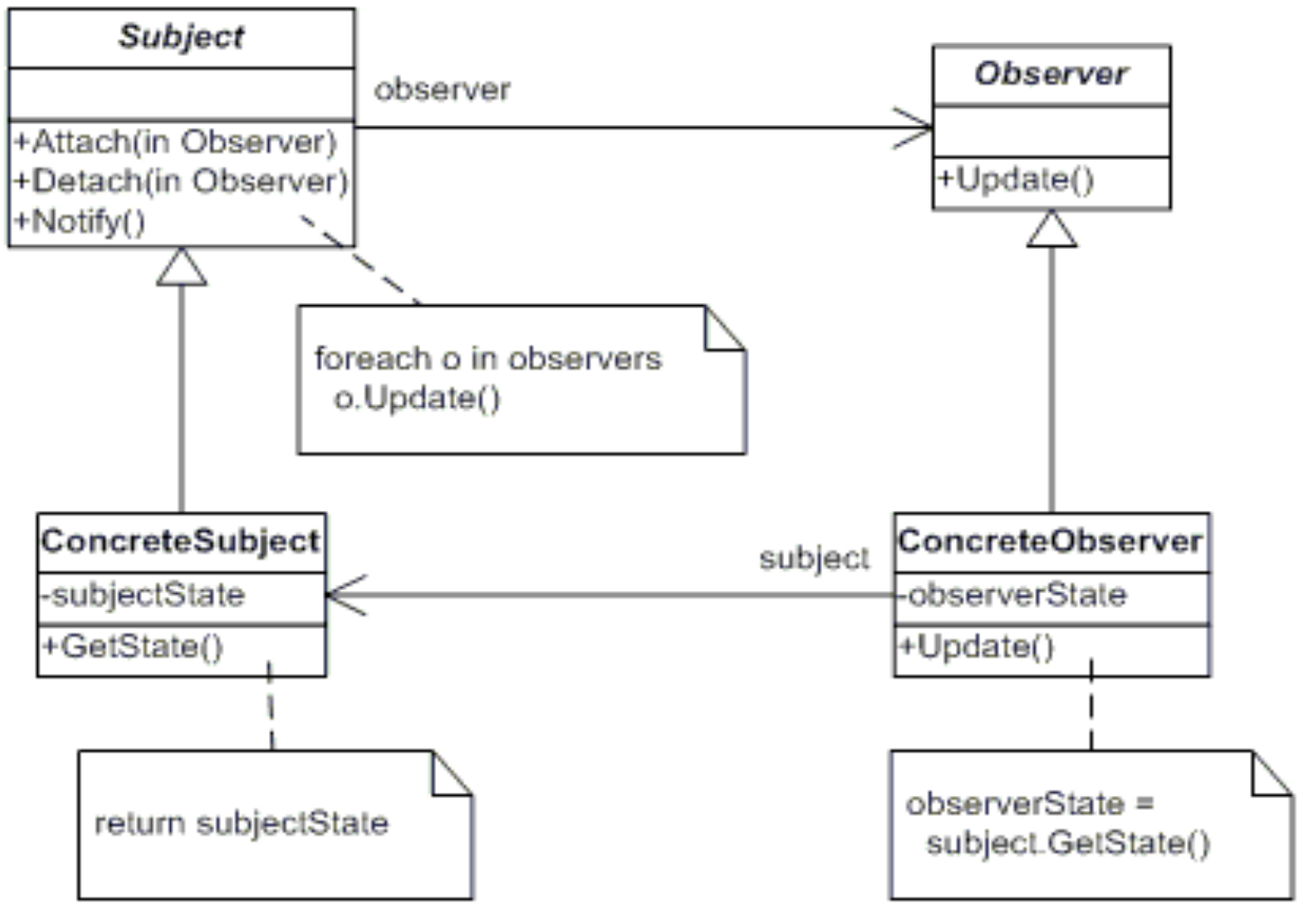
Command


Client
Invoker
Command
+Execute()
Receiver
+Action()
receiver
ConcreteCommand
-state
+Execute()
receiver.Action()

Command

- Představuje zapouzdření požadavku.
- Velmi vhodné k použití při vícevláknovém zpracování pomocí fronty.
- Příkladem může být např. EventQueue v Javě.

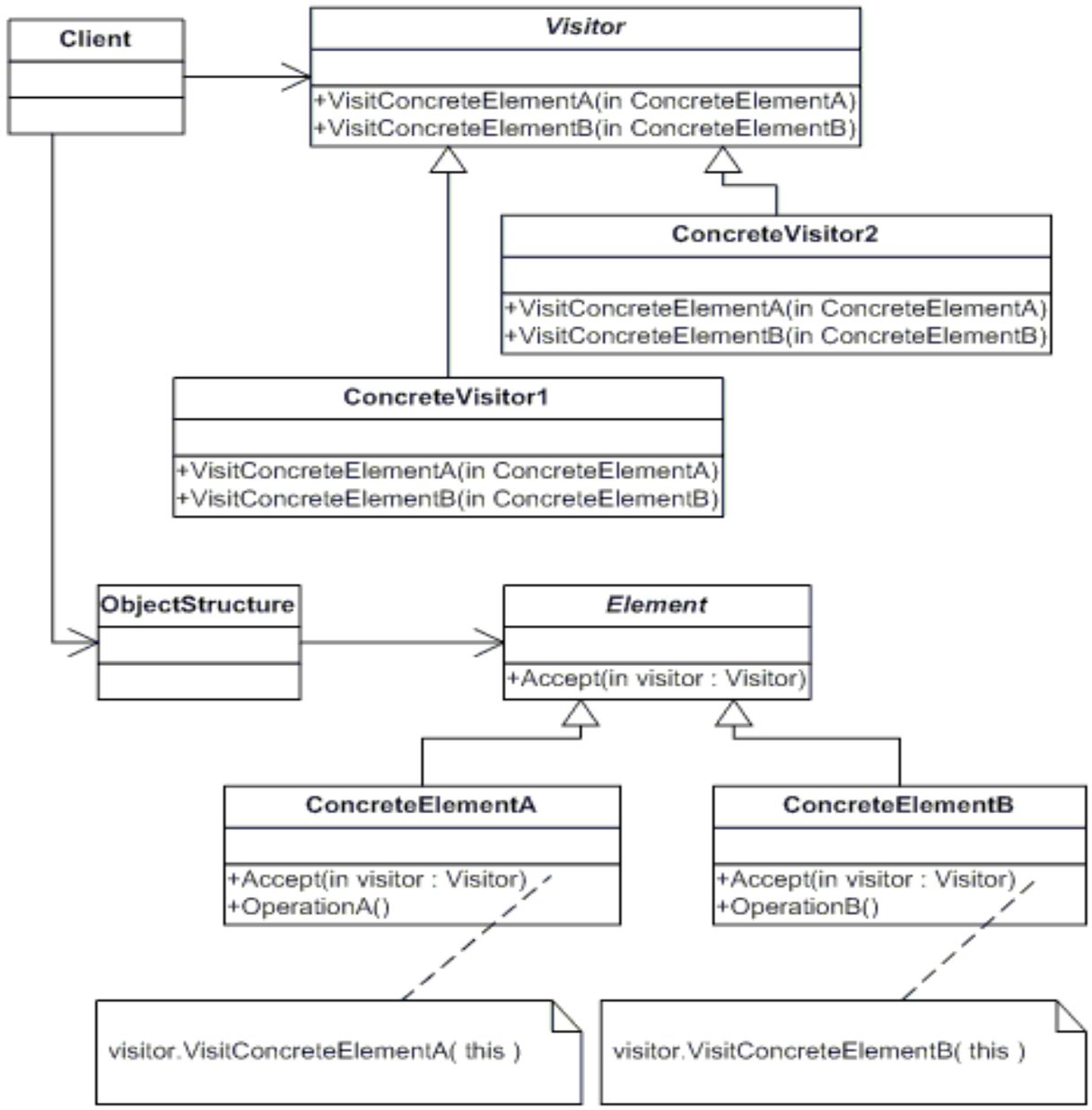
Observer


Subject
observer
+Attach(in Observer)
+Detach(in Observer)
+Notify()
Observer
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

Observer

- Často slouží k poslání události o změně registrovaným pozorovatelům.
- Jedná se v podstatě o implementaci callbacku.
- Lze použít například i k logování daných událostí v systému.

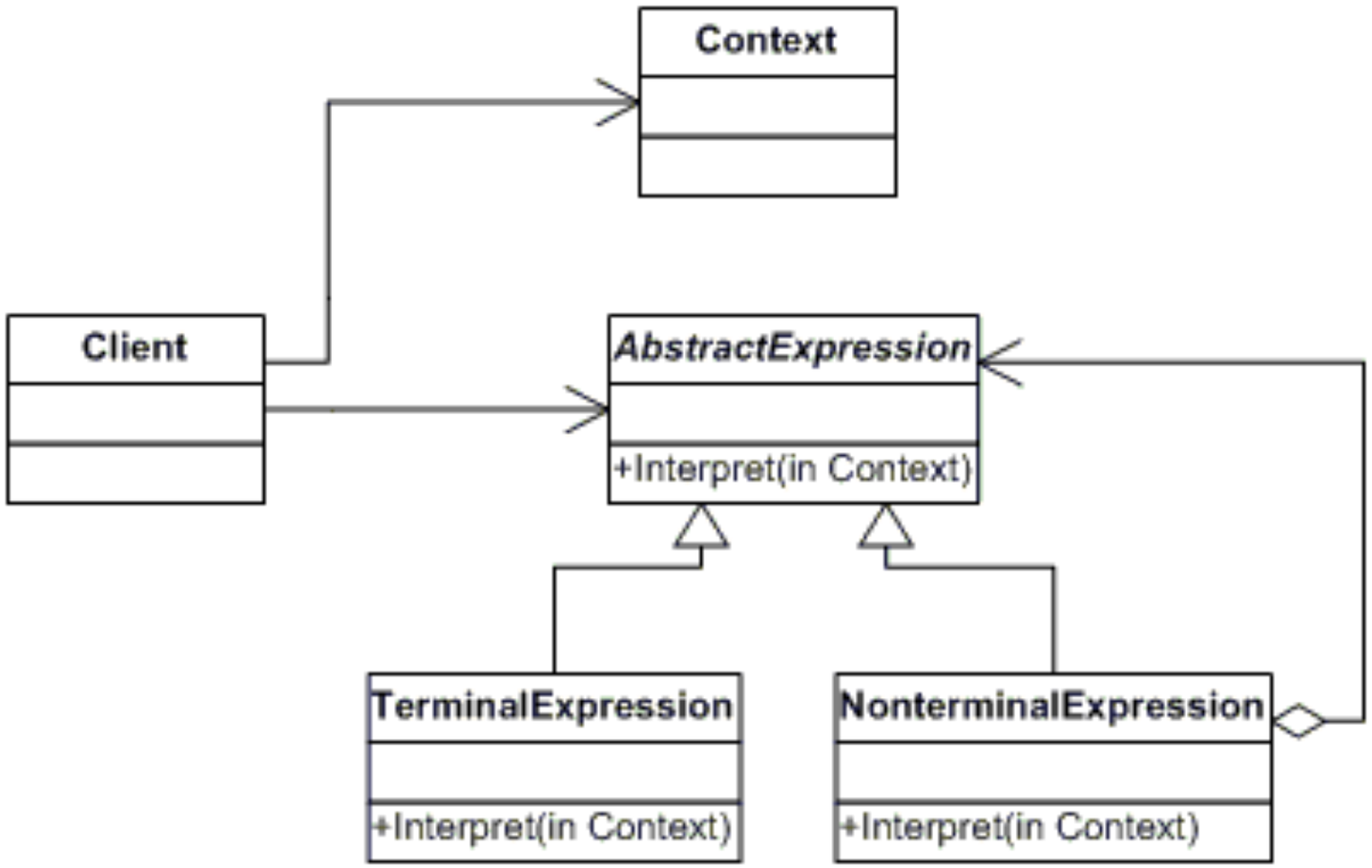
Visitor


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

Visitor

- Slouží k implementaci operace nad strukturou objektů, aniž bychom museli strukturu kvůli tomu měnit.
- Přidání další operace nad strukturou vyžaduje pouze implementaci nového visitoru.
- Používá principu double dispatch.

Intepreter


Context
Client
AbstractExpression
+Interpret(in Context)
TerminalExpression
+Interpret(in Context)
NonterminalExpression
+Interpret(in Context)

Interpreter

- Slouží k implementaci interpretru jazyka.
- Na základě gramatiky je vybudován AST a ten je interpretován.
- Velmi obecný návrhový vzor.
- Může se hodit například pokud chcete k vašemu systému poskytnout DSL.

Intepreter – Příklad

- Navrhněte interpreter matematických výrazů.
- Interpreter bude obsahovat následující operace:
  - Plus
  - Multiply
  - Minus
- Bude rozlišovat dva typy operandů – proměnné a hodnoty.
- Parametrem metody interpret bude Context v kterém budou uloženy hodnoty proměnných.

Literatura

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. **Design Patterns: Elements of Reusable Object-Oriented Software**, Addison-Wesley
- Pecinovský, R. **Návrhové vzory**, Computer Press
- http://www.dofactory.com/Patterns/Patterns.aspx
- https://csharpdesignpatterns.codeplex.com/