

Návrhové vzory

OMO, LS 2014/2015

Motivace

- **Cílem** objektového návrhu je strukturu aplikace **navrhnout tak, aby splňovala následující kritéria:**
 - snadná rozšiřitelnost
 - účelnost
 - testovatelnost
 - dokumentovatelnost

Koncepty OOP

- Objekty
- Abstrakce
- Zapouzdření
- Skládání
- Delegování
- Dědičnost
- Polymorfismus

Objekty

- Jedná se o prvky modelované reality, které v sobě seskupují jak funkčnost (**metody**) tak data (**proměnné**).
- Objekty si pamatují svůj stav v podobě **proměnných** a poskytují rozhraní pro komunikaci s nimi.
- Většina jazyků rozlišuje **třídy** a **instance tříd**.

Abstrakce

- Pomocí **abstrakce** programátor může **abstrahovat od některých detailů** jednotlivých objektů (abstraktní třídy, abstraktní metody, ...).
- Každý objekt pracuje navenek jako **černá skříňka**.

Zapouzdření

- Každý objekt navenek zpřístupňuje rozhraní, pomocí kterého se s objektem pracuje.
- Je vhodné, aby každá metoda tohoto rozhraní, po jejím provedení zanechala objekt v konzistentním stavu.

Skládání

- Objekt může obsahovat jiné objekty.
- Jedná se o vztah typu *has-a*.

Delegování

- Objekt může využívat služeb jiných objektů tak, že je požádá o provedení operace.

Dědičnost

- Objekty jsou obvykle **organizovány stromovým způsobem**, kdy objekty nějakého druhu mohou *dědit* z jiného druhu objektů, čímž přebírají jejich schopnosti.
- K těmto schopnostem mohou zděděné objekty přidávat svá vlastní rozšíření.

Dědičnost

- Vztah mezi třídami:
 - členské proměnné a metody definované v předku mohou být použity i potomkem,
 - potomek může přidat nové členské proměnné a metody,
 - potomek může změnit definice metod předka.
- Vždy musí platit vztah *is-a*.
 - Jedná se o specializaci nadtypu.

Dědičnost

- Při dědění by měl být vždy dodržen substituční princip LSP:

Let $q(x)$ be a property provable about objects x of type T . Then $q(y)$ should be provable for objects y of type S where S is a subtype of T .

Polymorfismus

- Situace, kde různé objekty:
 - rozumí stejné zprávě (mají metodu se stejnou signaturou),
 - ale na zprávu reagují různě (vyvoláním jiného kódu).
- Aby měl polymorfismus praktický význam, musí mít provedený kód stejný význam (v kontextu daného objektu).

Duplikace kódu a dat

Příklad: 10000 různých způsobů validace amerického Social Security Number v US vládních systémech

– Příčiny?

– Předejití?

Návrhové vzory

- místní architektura
- poskytují řešení opakujících se problémů
- reálné použití vzoru se může v praxi lišit
- Bible návrhových vzorů:

Design Patterns: Elements of Reusable Object-Oriented Software

Návrhové vzory

- Vzory pro tvorbu instancí
- Strukturální vzory
- Vzory chování

Vzory pro tvorbu instancí

- **Factory Method**
- Abstract Factory
- Prototype
- **Builder**
- **Singleton**
-

Strukturální vzory

- **Adapter**
- Bridge
- Composite
- Decorator
- **Facade**
- Flyweight
- **Proxy**
-

Vzory chování

- Interpreter
- Template Method
- Chain of Responsibility
- Command
- **Iterator**
- Mediator
- Observer
- **State**
- **Strategy**
- Visitor
- ...

Singleton

Aneb: Jediná instance v systému

Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

Převzato z [2]

Singleton - Vlastnosti

- Použije se v případě, že chceme zajistit od dané třídy existenci pouze jediné instance.
- V minulosti byl používán velmi často.
- Může způsobit problémy při rozšiřování programu (produktu).

Singleton – Příklad

```
public class SingletonDemo {
    private static volatile SingletonDemo instance = null;

    private SingletonDemo() { }

    public static SingletonDemo getInstance() {
        if (instance == null) {
            synchronized (SingletonDemo.class){
                if (instance == null) {
                    instance = new SingletonDemo ();
                }
            }
        }
        return instance;
    }
}
```

Singleton - Příklad

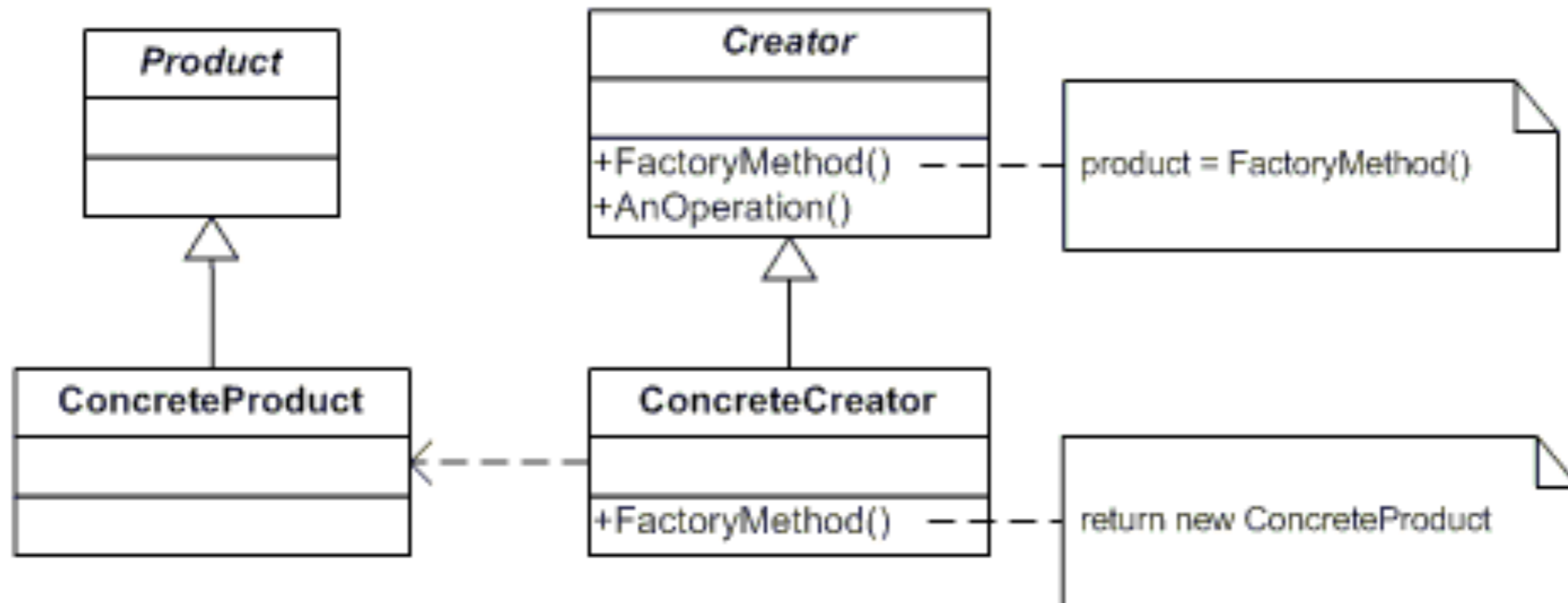
```
public class SingletonDemo {  
    private static SingletonDemo instance = null;  
  
    private SingletonDemo() { }  
  
    public static synchronized SingletonDemo getInstance() {  
        if (instance == null) {  
            instance = new SingletonDemo ();  
        }  
        return instance;  
    }  
}
```

Singleton - Příklad

```
public enum Singleton {  
    INSTANCE;  
    public void execute (String arg) {  
        // perform operation here  
    }  
}
```

Factory Method

Aneb: Střihni mi to na míru

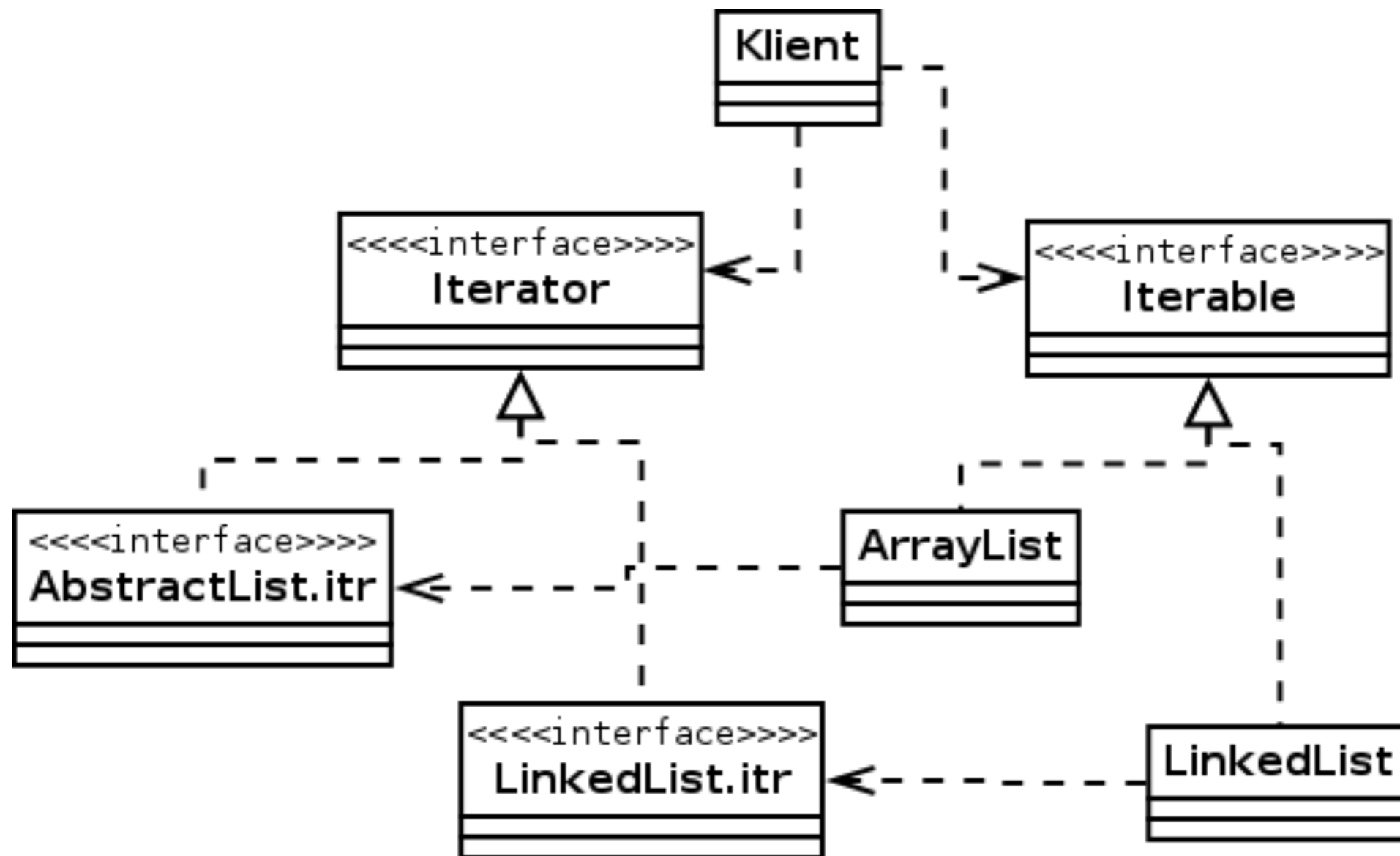


Převzato z [2]

Factory Method - Vlastnosti

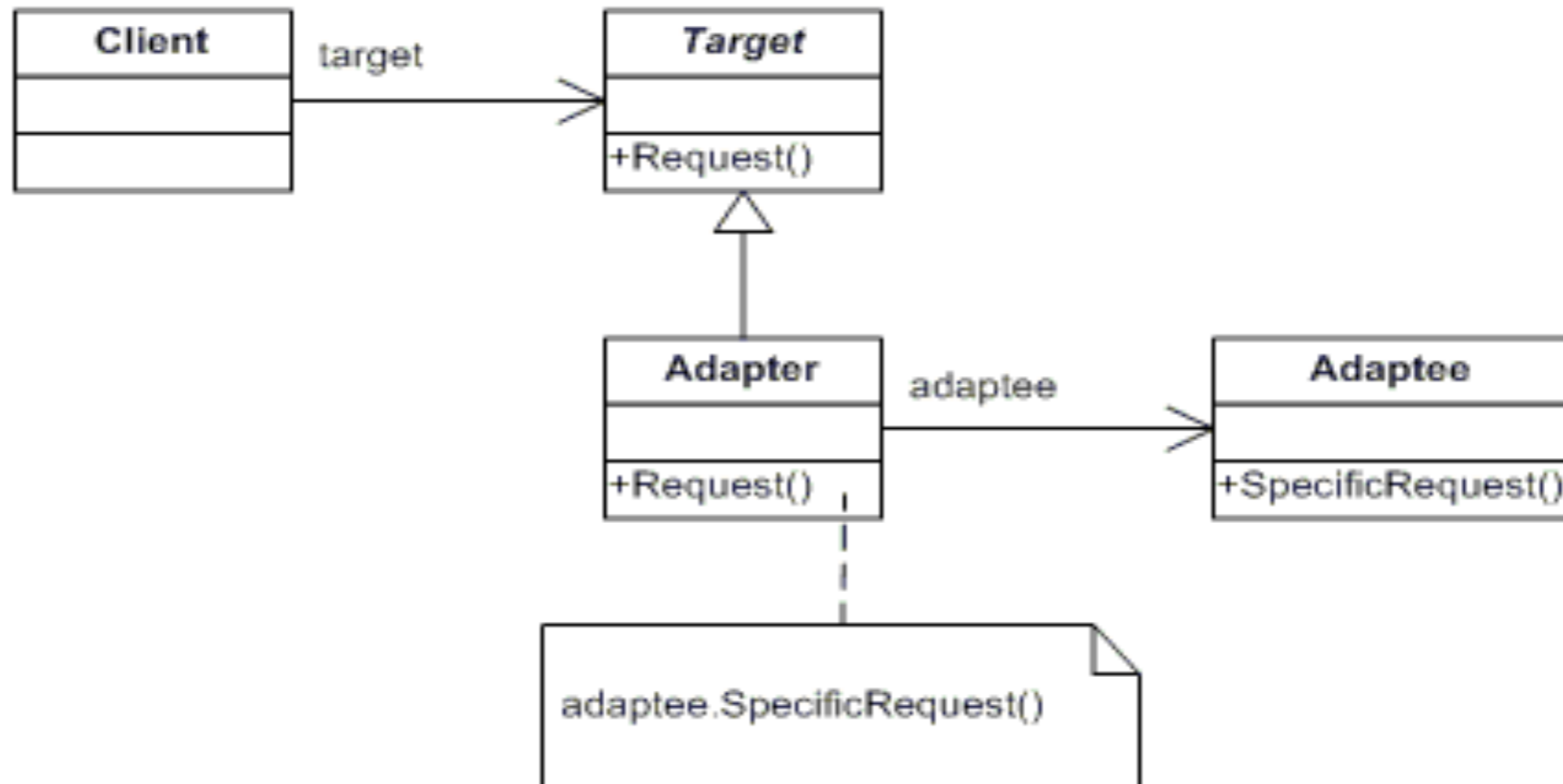
- Definuje rozhraní pro vytváření nových instancí.
- Podtřída rozhoduje o tom, která třída bude instanciována.
- Tovární metoda tedy přenechává vytváření instancí podtřídám.

Factory Method - Příklad



Adapter

Aneb: Je to trochu jinak

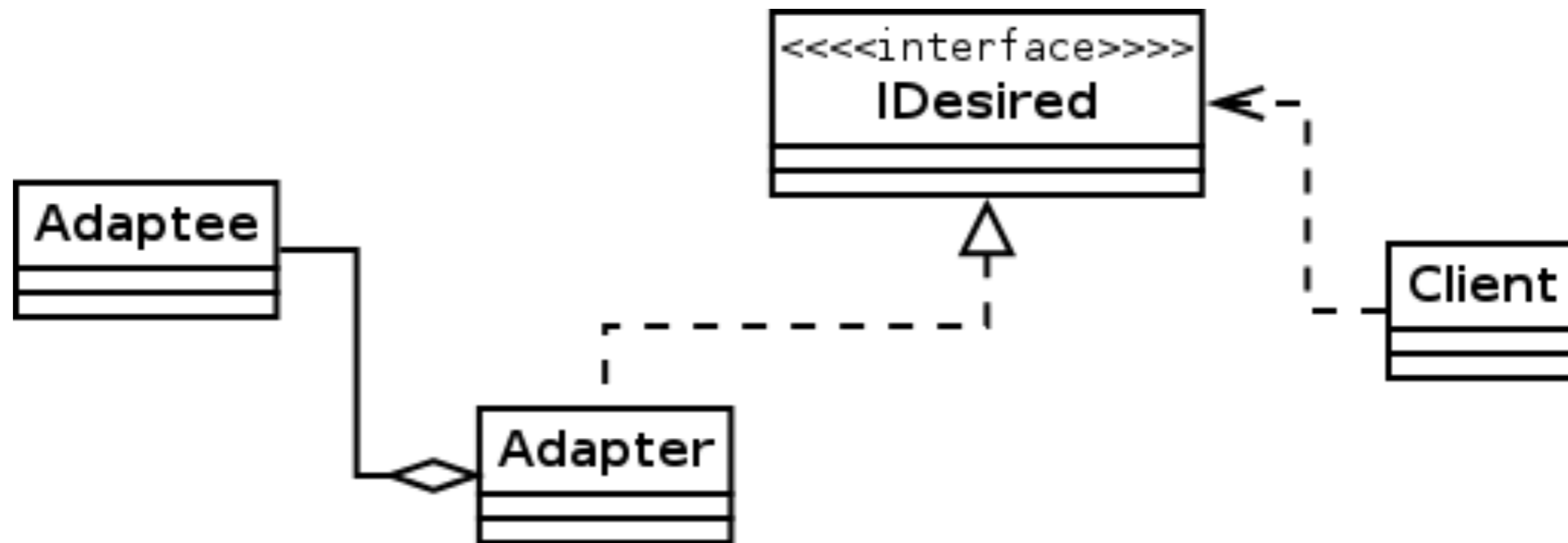


Převzato z [2]

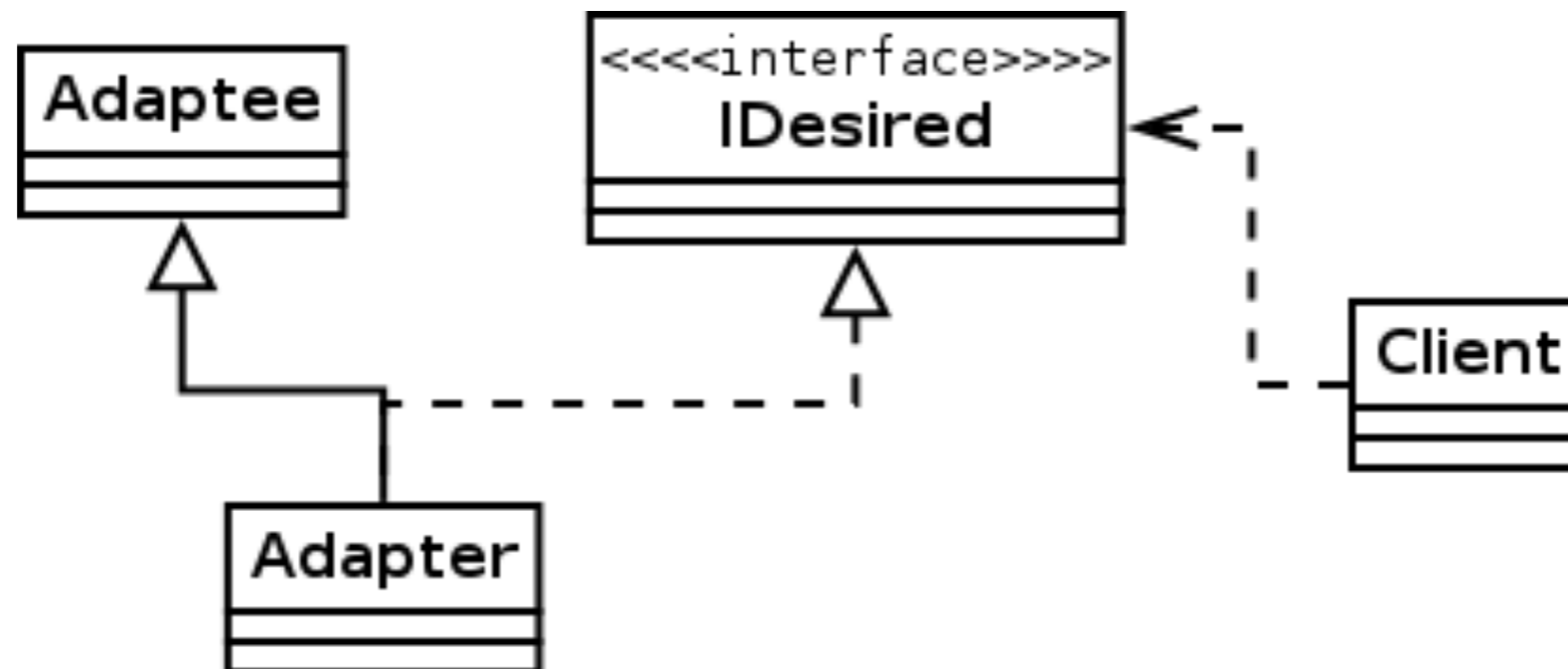
Adapter - Vlastnosti

- Slouží ke konverzi rozhraní jedné třídy na jinou.
- K použití je vhodný v případě, že rozhraní tříd se liší jen velice málo, například názvy metod.

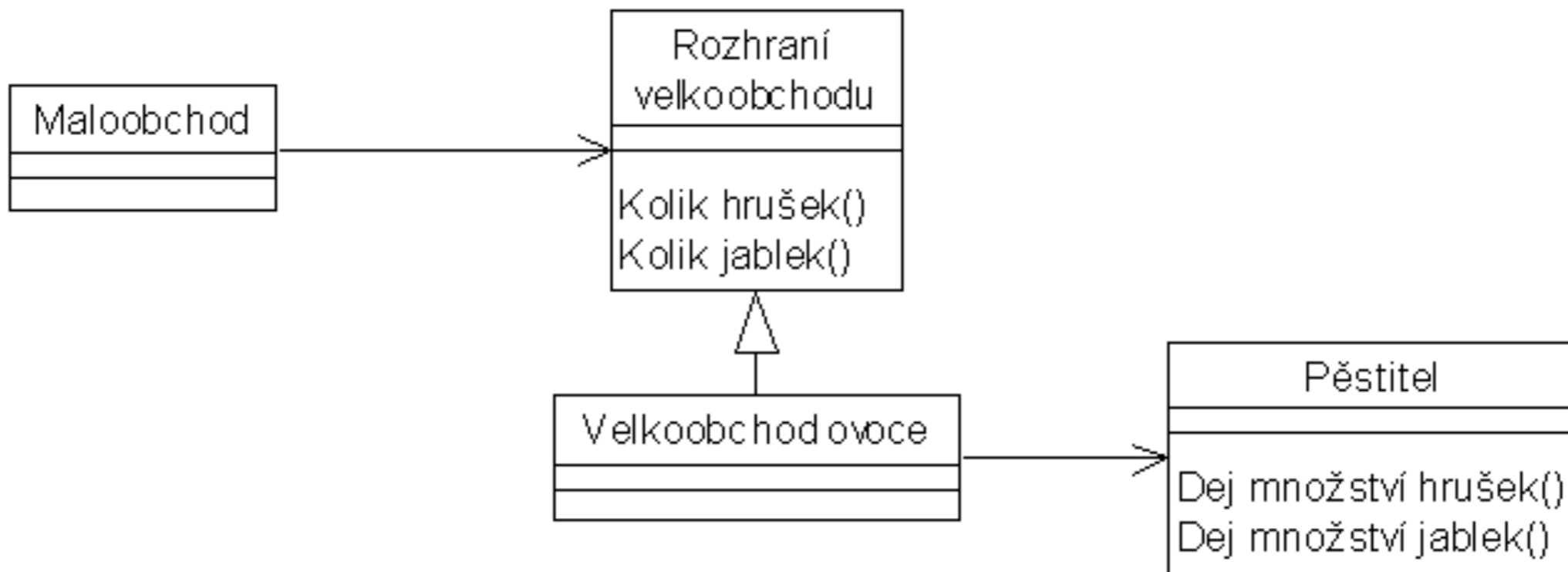
Adapter – Příklad 1



Adapter – Příklad 2



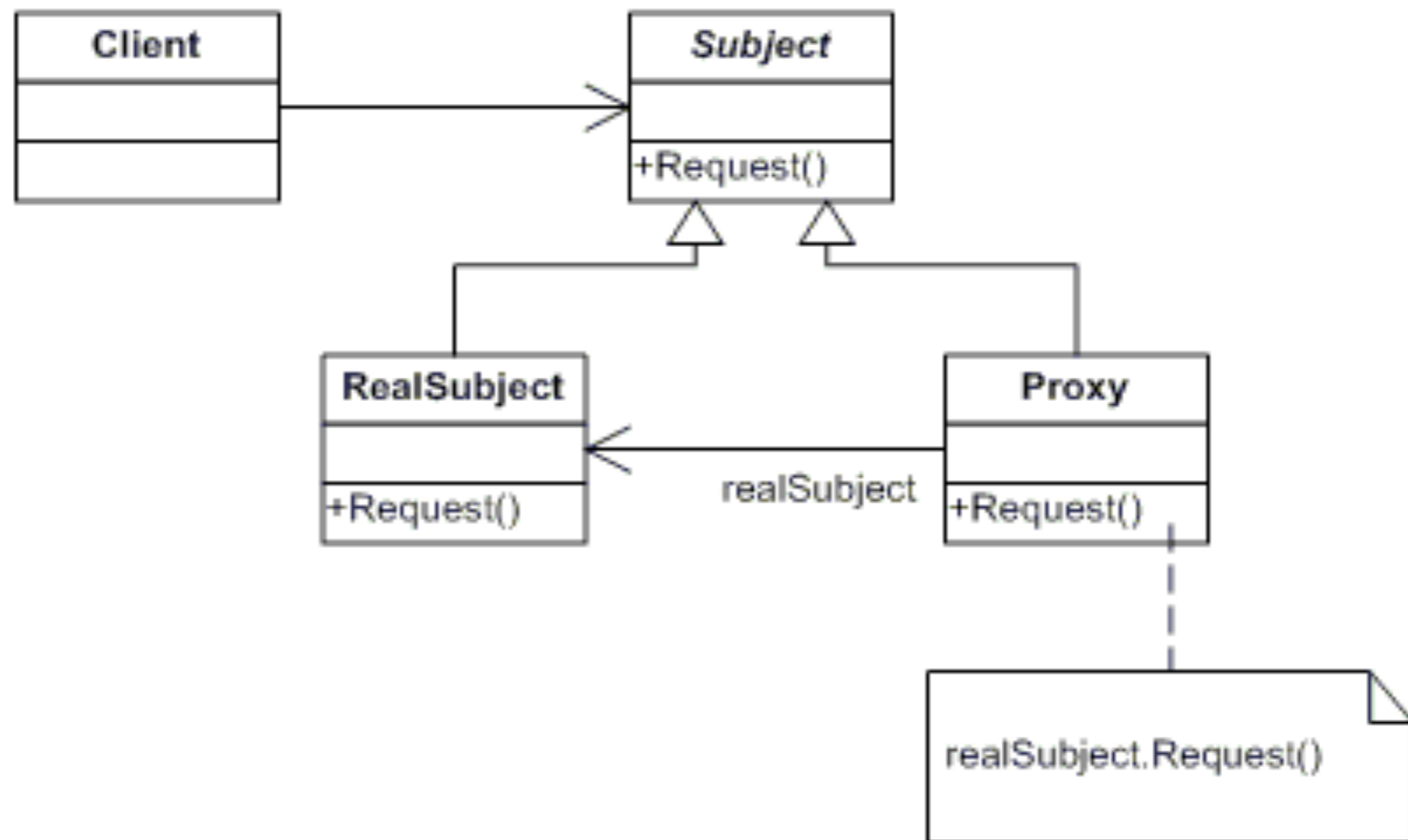
Adapter - Příklad 3



Obr 14 Object Adapter příklad

Proxy

Aneb: Pod ruce mi neuvidíš

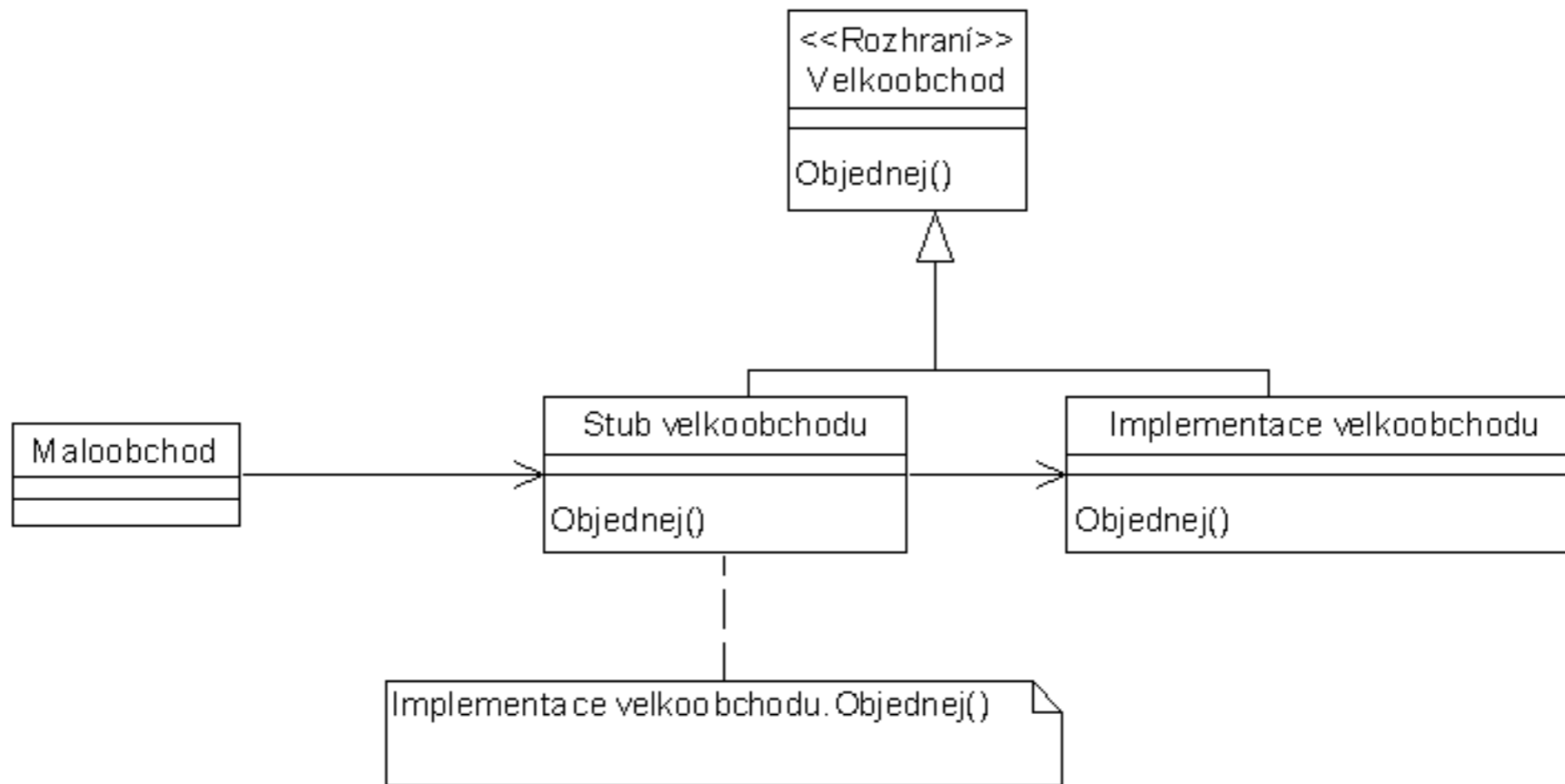


Převzato z [2]

Proxy - Vlastnosti

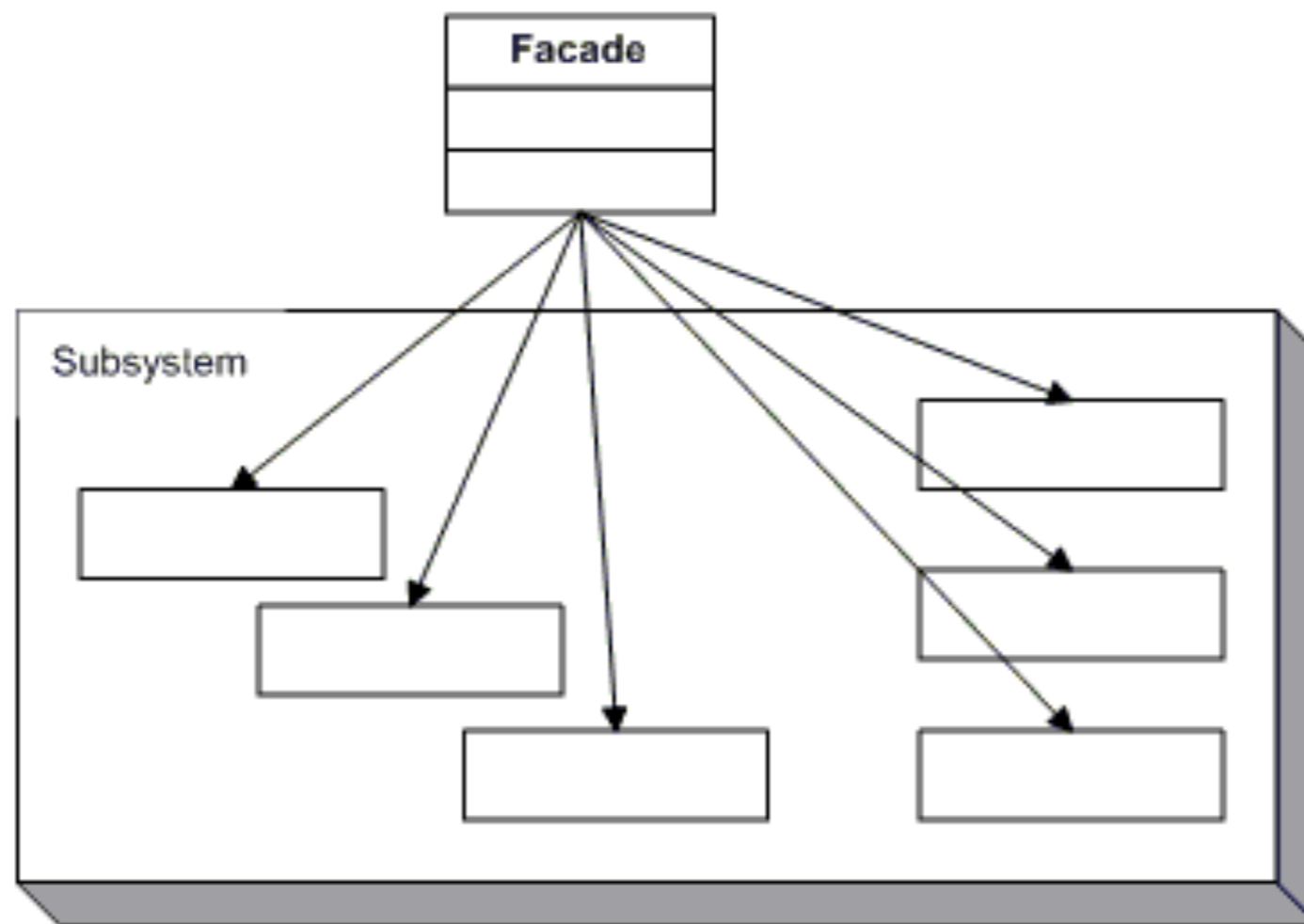
- Zastupuje cílový objekt, aniž by to uživatel poznal.
- Pomocí proxy lze například kontrolovat volání a parametry metod.
- Používá se například v RMI, Corba apod. pro zastoupení objektů, které existují fyzicky jinde.
- Lze využít například k logování nebo řízení přístupu.

Proxy - příklad



Obr 33 Příklad Proxy

Facade



Facade

- Fasáda je třída, která poskytuje jednotné rozhraní k celému souboru tříd.
- Velice užitečné řešení v případě, že rozhraní jednotlivých tříd jsou velice heterogenní a chceme je sjednotit.

Facade - Příklad

- Každý den ráno, když vyjždím na kolo provádím na svém telefonu následující akce:
 - vypnout WiFi
 - zapnout mobilní data
 - povolit GPS
 - zapnout Bluetooth
 - zapnout aplikaci na měření, např. Runkeeper

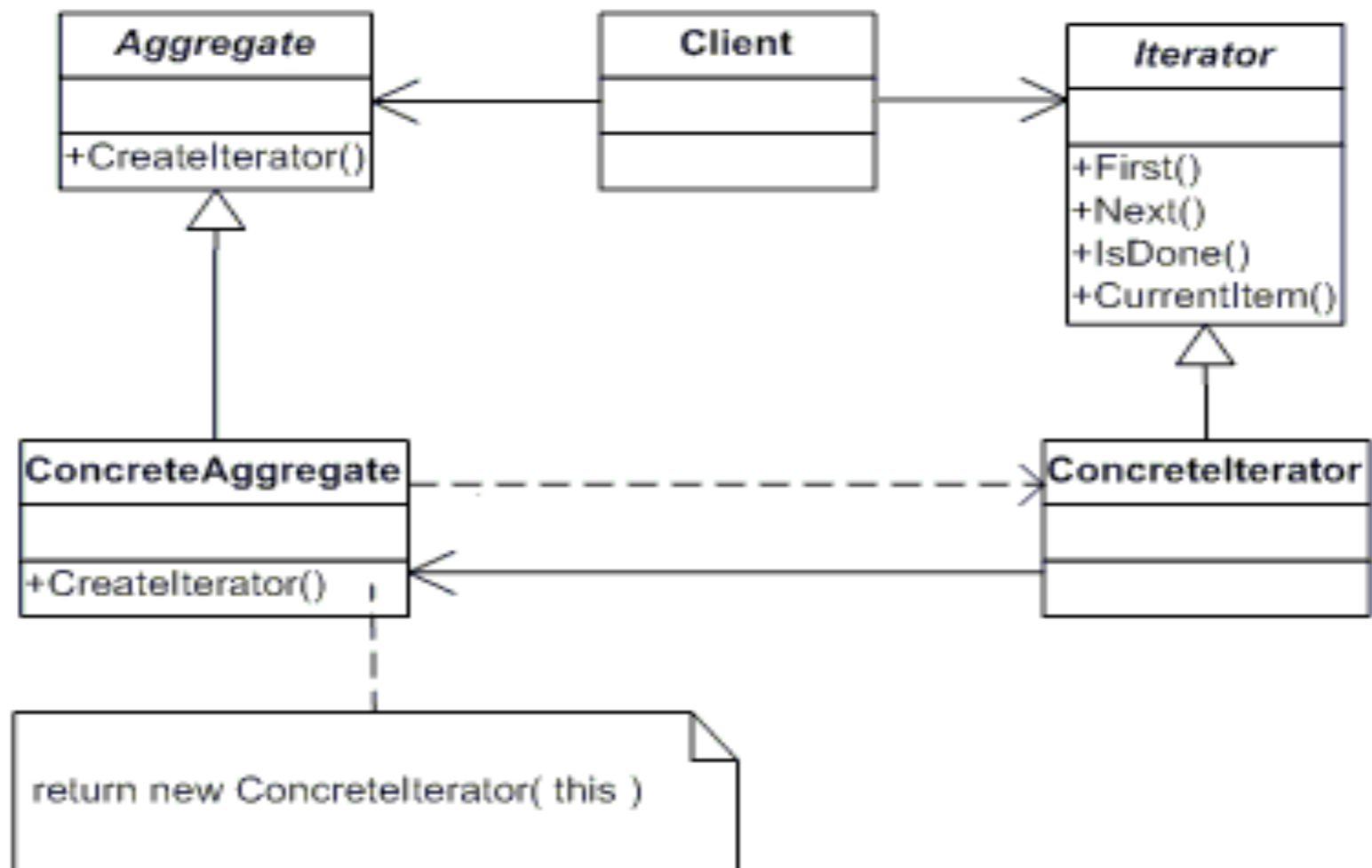
Facade - Příklad

- Poté, co přijedu z kola provádím následující akce:
 - nasdílím naměřenou vzdálenost na Facebook
 - vypnu Runkeeper
 - vypnu GPS
 - vypnu Bluetooth
 - vypnu mobilní data
 - zapnu WiFi

Facade - Příklad

- Jak bude vypadat třída, která mi usnadní život?
- Bude to právě fasáda, např. `CyclingFacade` s operacemi `startCycling` a `stopCycling`.

Iterator

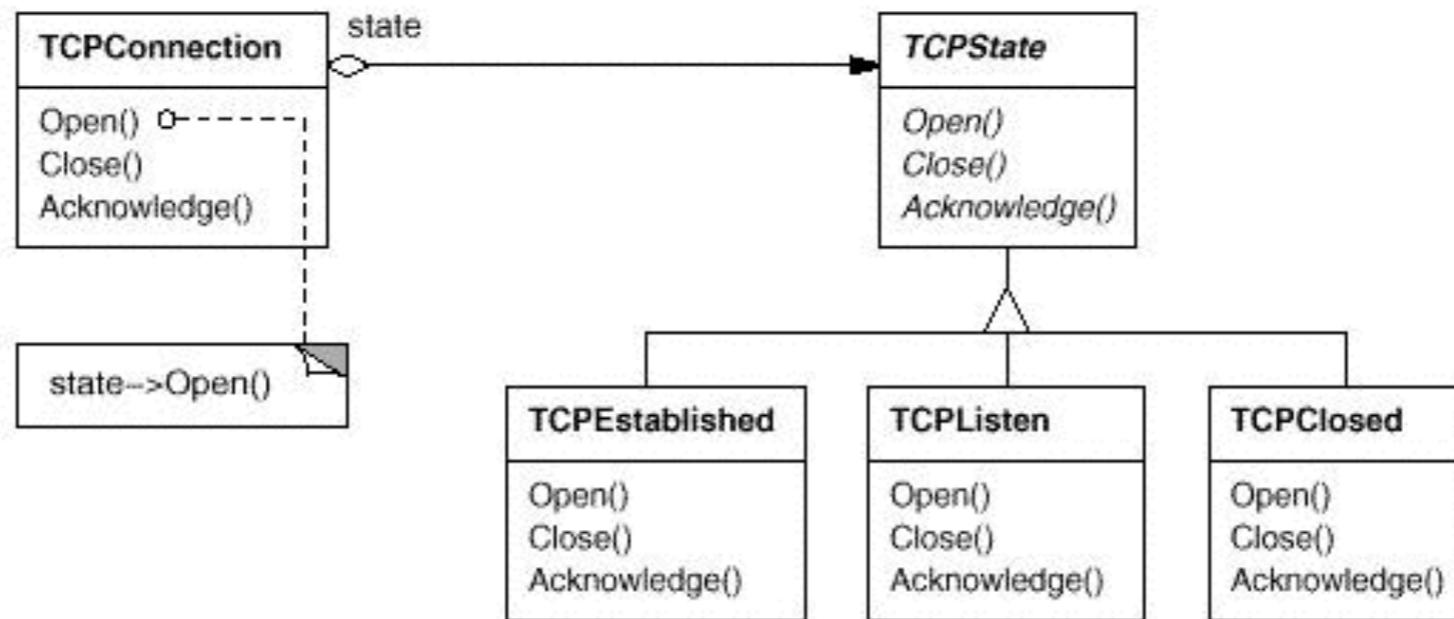


Iterator

- Poskytuje rozhraní pro sekvenční procházení dané struktury.
- Uživatel iterátoru nemusí znát skutečnou reprezentaci procházené struktury.
- Příkladem může být foreach (C#, Java) nebo iterátory v STL v C++.

State Pattern

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

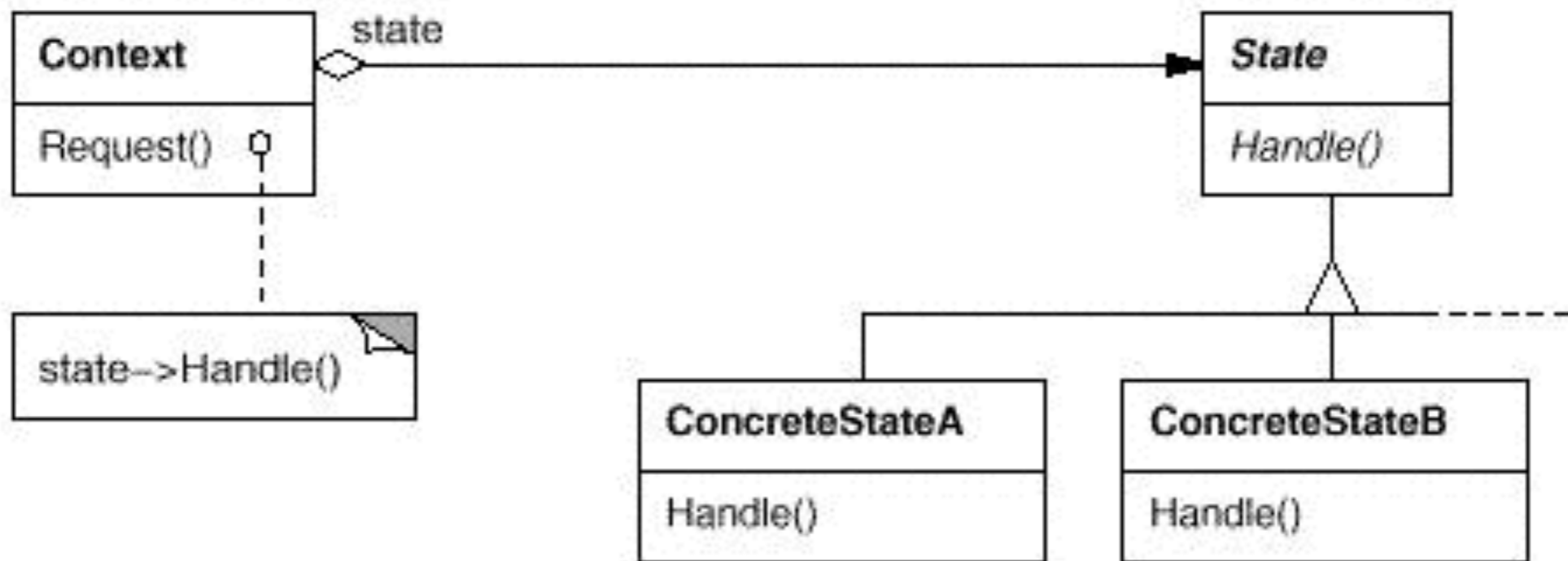


Zdroj: <http://userpages.umbc.edu/~tarr/dp/lectures/StateStrategy.pdf>

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

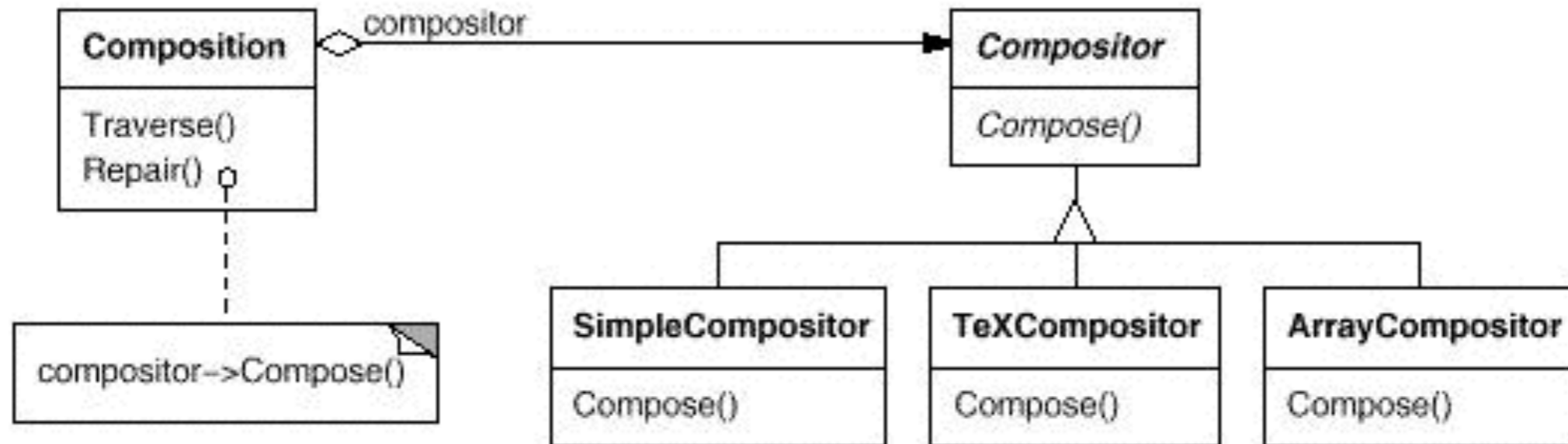


State Pattern

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

Strategy Pattern

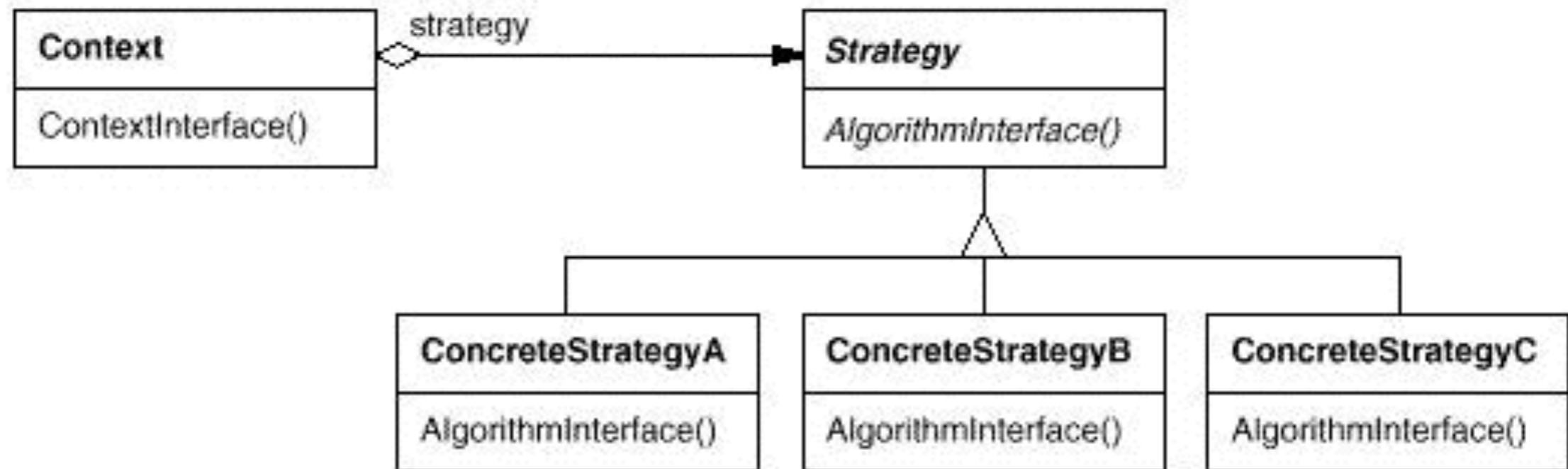
- Motivace:
 - Mějme skupinu algoritmů, každý je zapouzdřen, možnost libovolné jejich výměny. Strategy nám umožní změnu algoritmu bez ohledu na klienta.



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



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!

Literatura

- [1] <http://objekty.vse.cz/Objekty/Vzory>
- [2] <http://dofactory.com/Patterns/>