# Behavioral design patterns - Iterátor, Command a Strategy

*Iterujeme v 90% smyčkách. I když byste neuměli přímo vlastní iterátor napsat, dalo by se mluvit alespoň o tom co to je, jak ho používáme a kde.*

*Command jsme nedávno opakovali - zkrátka vykonávání příkazů pomocí Dictionary a interfacu.*

*Strategy jsme myslím nikdy neprobírali, ale je ze všech tří nejjednodušší. Neměl by být problém ho popsat i bez kódu.*

*Ve všech příkladech jsem používal viditelnost public, abych zbytečně neprotahoval kód gettery a settery.*

*Vypracoval jsem to pro C#, ale mělo by to být většinou stejné třeba i v javě.*

*Karel Muroň*

# Iterátor

Návrhový vzor Iterátor/Enumerator slouží k vytvoření rozhraní, které umožňuje postupně procházet objekty uložené v nějaké složitější struktuře. Implementace rozhraní zůstává uživateli skryta.

**Průchod složitého objektu**

Situací, kdy je iterátor užitečný je například taková, kdy chcete procházet objekt, jehož struktura není na první pohled zřejmá. Takovým případem může být například rodina, kde existuje velké množství vazeb (rodič, potomek, teta atd.) Rozhraní iterátoru zaručí, že projdeme systematicky všechny členy rodiny a žádného nezapomeneme, ani neprojdeme dvakrát.

**Jednotné rozhraní**

Další situací je, když chceme vytvořit jedno rozhraní pro více různých kolekcí (Viz pozdější příklad) Toto rozhraní následně implementujeme například pro spojový seznam, strom, dynamické pole a podobně. V programu má pak uživatel možnost procházet všechny kolekce jednotným způsobem, aniž by musel vědět, kde a jak jsou data uložena.

**Optimalizace průchodu**

Třetím uplatněním iterátoru je optimalizace průchodu kolekcí, kde například v případě spojového seznamu může správným použitím iterátoru dojít k velkému výpočetnímu ulehčení.

*Pojmy iterátor a enumerátor jsou záměnné. V C# se používá slovo enumerátor.*

Iterace:

* Iterace v programování znamená opakované volání funkce v počítačovém programu
* Opakování
* “Početní metoda, krok přiblížení se k výsledku”

V našem případě zkrátka **načítání jednoho prvku za druhým** v nějaké kolekci. Běžně jsme iterovali třeba pomocí for smyčky:

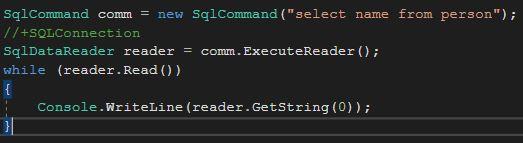
for(int i = 0;i<pole.length;i++){ … }

Nebo pomocí foreach:

foreach(int i in pole) { … }

Nutno dodat, že v tomto případě je iterátor schovaný v samotném foreach. Kdo foreach běžně používá, ví, že je **readonly.**

Kde jsme iterátor jako takový běžně používali, bylo třeba při procházení SQL querry:



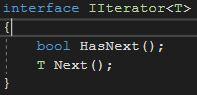
Tzn. Vytvoří se dotaz -> Získáme výsledek v podobě readeru -> ten potom iterujeme metodou .Read().

Co tedy metoda Read() dělá? Zjistí, jestli existuje následující prvek, načte ho a vrátí true. Já pak takový prvek můžu načíst a třeba ho vypsat. Pokud neexistuje vrátí false a smyčka se ukončí.

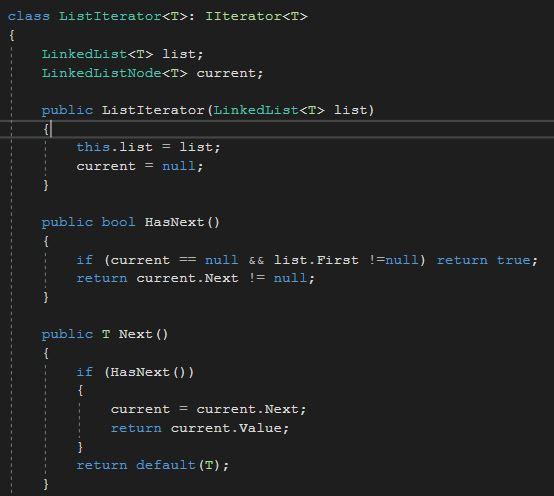
A to je přesně princip základního fungování iterátoru. Každý iterátor by měl obsahovat metodu **HasNext()**, která zjistí jestli existuje další prvek a **Next()**, která další prvek načte. Metoda Read() v minulém příkladě je tak kombinací obou.

Základní výhodou iterátoru je, že **nemusím znát strukturu prvku**, který iteruji. Další výhoda je **menší asymptotická složitost** např. u spojového listu (ne u všech kolekcí), než kdybych procházel prvky jeden za druhým pomocí .Get(index).

Př.: Chci si udělat vlastní iterátor na procházení linked listu (spojového seznamu viz. samostatná otázka: Datové struktury: Spojový seznam, Strom, Fronta, Zásobník, Halda). Nejprve musím udělat interface, který mi zajistí již obě zmiňované metody:



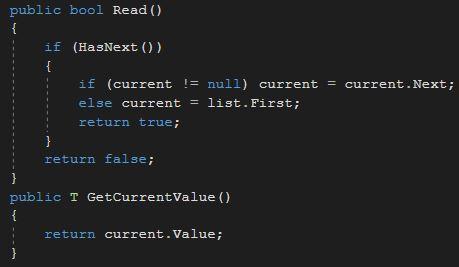
V interfacu jsem použil generika, abych mohl zaručit, že můžu iterovat nehledě na datovém typu (viz. samostatná otázka: Datové typy, Generika, Výčtové datové typy, Struktury, Delegáti, Eventy).

Dále interface implementuji do třídy, která bude iterovat list:Do konstruktoru vstupuje linked list, který chceme iterovat. Jako aktuální uzel je nastaveno null.

Metoda HasNext() kontroluje, jestli existuje další prvek. If slouží pro případ, jestli vůbec existuje první uzel. Pak už jen kontroluji, jestli vlastnost uzlu Next (případný další uzel) je null.

Metoda Next() načte další uzel a vrací generikum (hodnotu) dalšího uzlu.

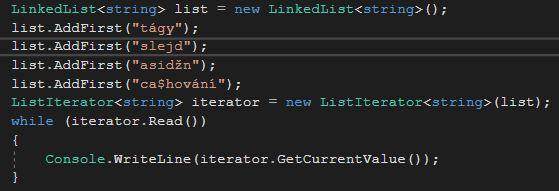
Pokud bych chtěl mít podobnou metodu, jako v případě iterace SQL Querry, vypadala by takto:



Read() tedy nejdříve zjistí, jestli existuje další uzel a pokud ano, tak ho načte a vrátí true. Pokud ne, vrací se false.

GetCurrentValue() se stará o to, abych si mohl přečíst hodnotu aktuálního uzlu.

Implementace v Mainu, pak vypadá obdobně:



**Implementace interfacu se u každé kolekce liší!** V případě pole bych mohl snadno ukládat jen aktuální index. Taktéž by se zjednodušila metoda HasNext() - prostě bych hlídal jestli nepřekračují délku pole.

Command

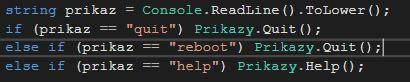
//Návrhový vzor Command se využívá v situacích, kdy je třeba vykonání nějakého příkazu zapouzdřit do třídy a tu jako proměnnou předávat v aplikaci, například z důvodu jejího zařazení do fronty k pozdějšímu vykonání nebo rekonstrukci historie příkazů. Důležité je, aby zapouzdřený příkaz obsahoval všechny potřebné informace nutné ke svému vykonání. Je-li to potřeba, může příkaz obsahovat i instrukce k provedení opačné operace, což lze ve spojení s historií příkazů použít k implementaci funkcionality undo/redo.//

Command, jak název napovídá, řeší spouštění příkazů od uživatele.

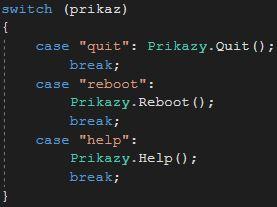
Pokusím se naznačit, jak daný problém vyřešit různými způsoby.

*Zcela správně by se měla ve všech příkazech ještě hlídat integrita. To ale není v tomto případě podstatné, tak jsem to vynechal.*

Chceme spustit požadovanou metodu, dle příkazu z konzole. Nejjednodušší způsob by bylo udělat hromadu podmínek nebo ještě lépe else-if:



Lepší, ale furt poměrně pomalé je použití switch-case:



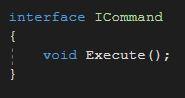
Tři příkazy ještě nevadí, ale s narůstajícím počtem příkazů se už tak pomalý switch-case zpomaluje. Nemluvě o dlouhatánském kusu kódu.

Command toto řeší zcela jinak. **Příkazy se zadávají jako klíč do Dictionary**. Dictionary **vrátí objekt třídy a spustí se metoda** s požadovaným příkazem.

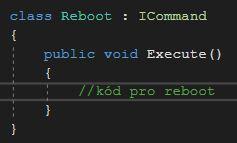
Tzn. **Pro každý příkaz se vytvoří samostatná třída** a v každé této třídě je metoda se stejným názvem. Tu pak voláme, jakmile se nám třída příkazu vrátí **z Dictionary**.

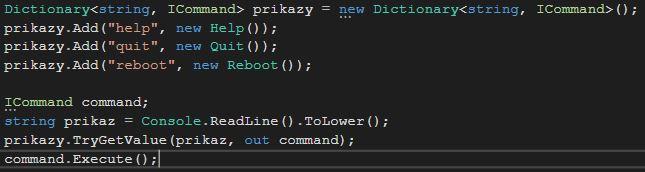
Jak zajistit, že všude bude stejně pojmenovaná metoda a zároveň budu moci všechny ty třídy dát do stejné kolekce? **Pomocí interfacu**.

Př.: Nejdříve si vytvořím společný interface pro všechny třídy. Metoda Execute(), která se stará o samotné provedení příkazu, by měla být vždy **void:**



Dále si vytvořím třídy pro všechny příkazy. Do nich implementuji interface a do metody Execute() napíšu kód vykonání příkazu:



V Mainu: 

Ze začátku jsem si vytvořil Dictionary. Jako klíč je příkaz (string) a jako hodnota vytvořený interface (ICommand). Pak jsem naplnil Dictionary objekty tříd.

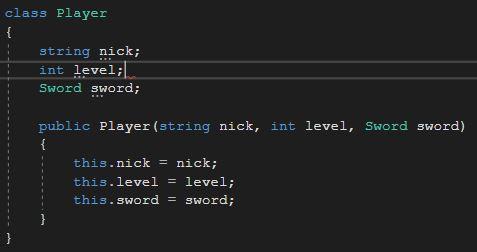
Dopředu jsem si připravil command, do kterého se pak dosadí vybraný objekt třídy.

Tzn. Uživatel zadá příkaz -> Dictionary najde objekt třídy -> vrátí jej do ICommand command -> spustí se příkaz pomocí command.Execute()

# Strategy

*//Návrhový vzor strategie (Strategy, Policy) slouží k vyměňování různých implementací algoritmu za běhu programu. Tato záměna může proběhnout buď explicitně (na žádost klienta) nebo implicitně (na základě nastavení ovladače).//*

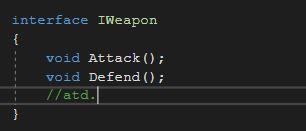
Představme si situaci, kdy máme třídu hráč, který má třeba nick, level a v ruce meč:



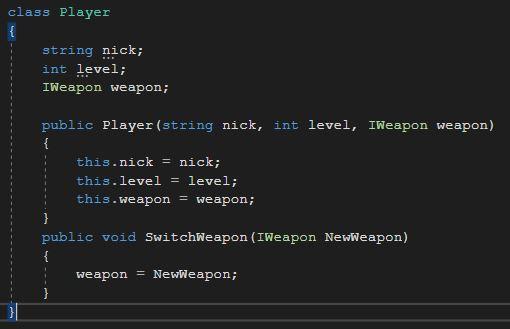
Co když si budu chtít změnit meč za jinou zbraň? A toto řeší strategy.

Strategy se stará o to, aby se **objekty s podobnými vlastnostmi uvnitř jiné třídy daly snadno vyměnit**.

Jak na to? Opět přes interface. Vytvoříme si interface s metodami, které chceme, aby měly všechny zbraně:



Pak stačí implementovat interface do tříd jednotlivých zbraní. To mi umožní nahradit ve třídě hráče meč za interface Weapon:



Nyní můžu dosadit za weapon jakýkoliv objekt třídy, která má implementovaný interface.