**Behavioral design patterns - Iterátor, Command a Strategy**

jsou založeny na vzorech, které identifikují běžné komunikační vzory mezi objekty a tyto vzory realizují.

Tímto způsobem tyto vzorce zvyšují flexibilitu při provádění této komunikace.

jsou vzory, které se zabývají interakcí mezi objekty.

Interakce mezi objekty by měla probíhat takovým způsobem, aby spolu hovořily a stále byly volně spojené.

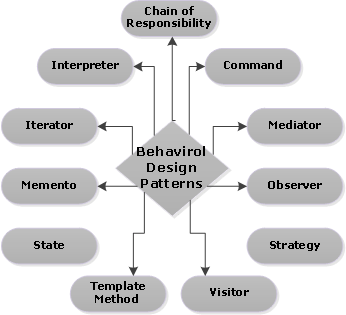
Volná vazba je klíčem k architektuře n-tier.

vysvětlují, jak objekty interagují. Popisuje, jak si různé objekty a třídy navzájem posílají zprávy, aby se věci staly, a jak jsou kroky úkolu rozděleny mezi různé objekty.

**Def**

jak můžeme předat požadavek mezi řetězcem objektů, nebo můžeme říci, že definuje způsob komunikace mezi třídami a objektem.

**Vzory**



**Kdy se používají**

V aplikacích v reálném čase někdy chceme změnit chování třídy a znovu nechceme, aby to ovlivnilo jiné třídy projektu.

Například máme třídu faktur, která aktuálně používá daně jako 18%. Zítra, pokud chceme přidat další daň navíc. To znamená, že měníme chování třídy. K vyřešení takového typu problémů s chováním se do obrazu dostane model chování.

**Iterátor**

Zavádí ho **samostatný objekt**

**umožňuje jednoduché lineární procházení kolekcemi, aniž bychom museli znát vnitřní strukturu těchto kolekcí**

Objekt iterátoru bude udržovat stav iterace, bude sledovat aktuální položku a bude mít způsob, jak identifikovat, jaké prvky jsou vedle iterace.

To umožňuje standardní procházení seznamů, stromů a dalších struktur.

Je to jeden z nejpoužívanějších vzorů, někdy se místo označení Iterator používá i **Enumerator**

**Většina kolekcí implementuje IEnumarable, takže při procházení kolekce třeba foreach je použit, takže je read-only**

**Metody**

Vzor Iterator tedy zavádí stejnojmenné rozhraní, které obvykle poskytuje následující metody

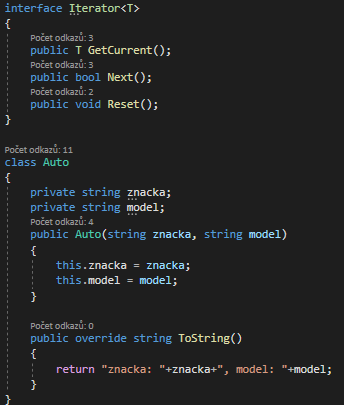
GetCurrent() - Vrátí prvek na aktuální pozici. Reference na prvek na aktuální pozici v kolekci se často nazývá **kurzor**.

Next() - Přesune kurzor na další prvek v kolekci a vrátí true. Pokud se v kolekci další prvek již nenachází, vrátí false.

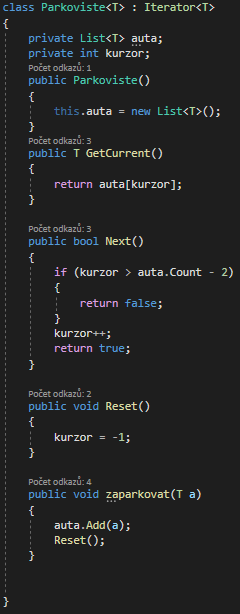
Reset() - Přesune kurzor zpět na první pozici.

Také přesun mezi prvky bývá nazýván: HasNext() a GetNext()

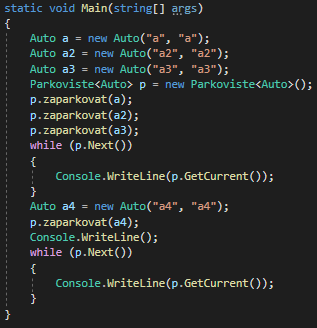
**Program, napsanej vlastní iterator**

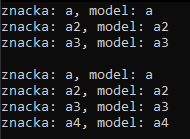


Vlastní interface se 3 metodami, třída auto



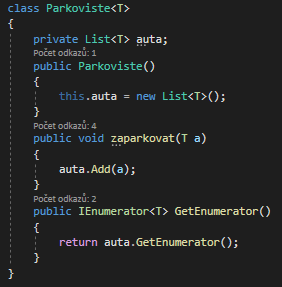
Iterator je generický typ T, takže parkovště taky, metoda GetCurrent vrátí auto na které ukazuje ukazatel (je na jeho indexu), Next se zeptá jestli je na dalším místě taky auto (-2 je protože poslední index pole o velikosti 5 je 4, kurzor bude třeba 3 zeptá se a posune se na hodnotu 4, ta podmínkou neprojde a tak nevypadl z indexu pole), Reset nastaví kurzor na jedna, když je ve while cyklu procházení, tak se první zavolá Next a ukazatel se dostane na hodnotu 0, v metodě přidání auta musí být Reset, jinak by po přidání výpis šel dál od kud skončil (nemusí být public, když bude volám jen ve třídě)



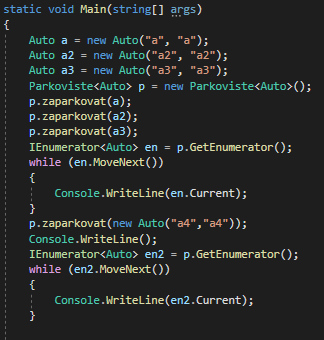


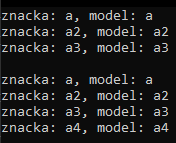
**Nebo druhý příklad bez vlastního Iteratoru, místo něj IEnumerator<T> z C#**

Je třeba použit v while cyklu Read() v SqlDataReaderu, Read() je jako Next()

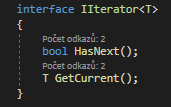


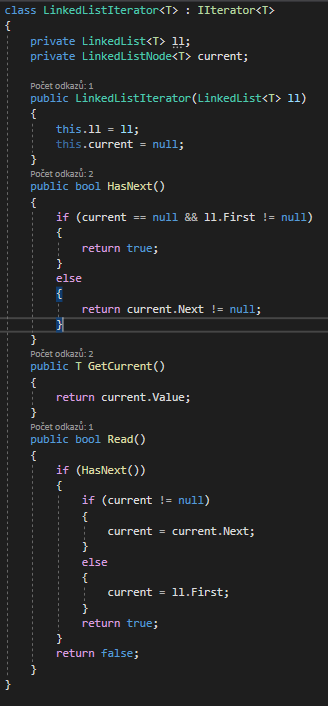
Lsit je private, takže pro získání enumeratoru je vlastní metoda.

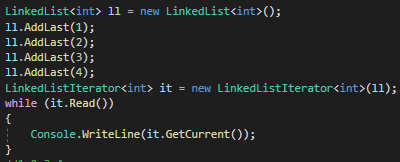




**LinkedList**







**Výhody**

Nemusí znát strukturu prvku, který iteruje.

Menší asymptotická složitost u linked listu

**Command**

převádí požadavky nebo jednoduché operace na objekty.

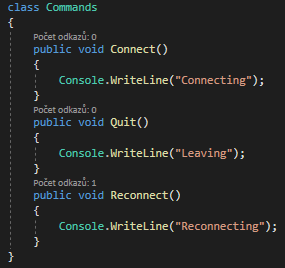
Když jsou převedeny, tak je lze uložit do kolekcí a bude je možná později vyvolat

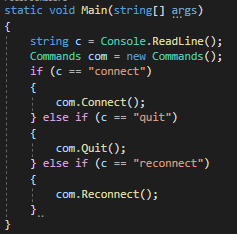
Důležité je, aby převedený příkaz (objekt) obsahoval všechny potřebné informace ke svému vykonání

Nejvíce slouží k řešení spouštění příkazů od uživatele, je přehledný dá se skladovat v kolekci commandů…

**Příklad**

Máme clienta a ten má nějaké commandy, můžem vytvořit třídy commands a do ní dát metody, poté jí v mainu vytvořit a pomocí if a else volat metody třídy





Lepší řešení by bylo použití switch, ale také zabere mnoho místa v kódu a je při více commandech pomalejší

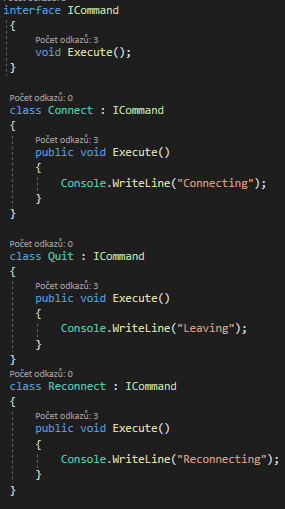
**Řešení pomocí Command vzoru**

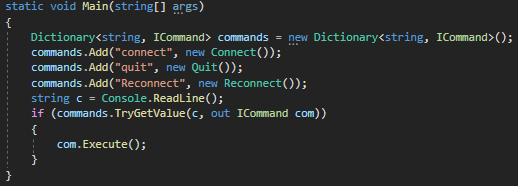
Vytvořit interface s metodou Execute() (může jich být víc)

Vytvořit třídy pro každý command, které implementují interface

Main

Dictionary, které obsahuje string a ICommand, najde podle klíče objekt třídy, vrátí ho do ICommand a spustí metodu Exucute()





**Strategy**

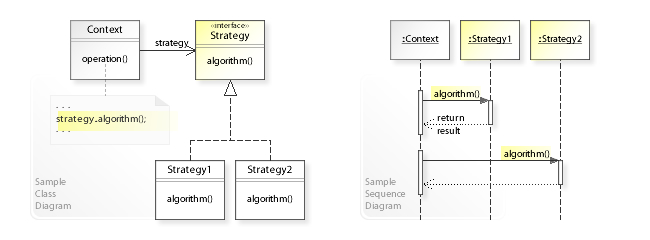
je tvořen abstraktní třídou, která definuje rozhraní pro celou rodinu algoritmů, které představují konkrétní strategie.

Namísto abstraktní třídy se může jednat i o interface.

Jednotlivé algoritmy z ní poté dědí a toto rozhraní implementují svou logikou.

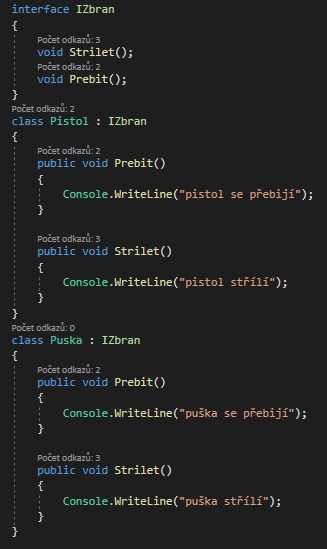
umožňuje klientovi vybrat si, který algoritmus se použije z rodiny algoritmů, a poskytuje mu jednoduchý způsob přístupu a také jeho změnění

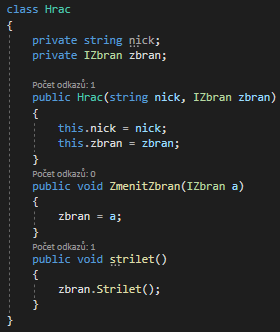
Def: Strategický vzor definuje rodinu algoritmů, každý z nich zapouzdřuje a umožňuje jejich zaměnitelnost.

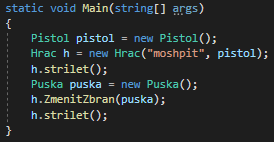


**Příklad**

Třeba hráč má jméno a zbraň, zbraně jsou pistol, puška atd… dát do jeho třídy instanci pistole, tak nebude moc získat pušku. Proto se použije Strategy. Vytvoří se interface zbraň, třída pistol a puška, které implementují interface zbraň, tím to jsou v rodině “zbraní“. Do třídy hráče se dá instance interface zbraň a bude moci měnit zbraně.









**Vylepšenej**

Třída může implementovat více interface, může být lehká a těžká jednotka, kde nějaké zbraně budou jen pro lehkou (interface ILehke), některé společné (ILehke, ITezke) a některé pro těžkou (ITezke)

**Třešnička**

Strategy je podobný, jako vzor Bridge z rodiny Structural design patterns, nicméně je určený pro chování, nikoli strukturu.