Parsovací knihovna v C#

Jan Fürst a Filip Štrobl

Cílili jsme na jednoduchost a bezpečnost použití

- chyby programátora by měly být odhalitelné co nejdříve (za překladu)
  - typová bezpečnost
    - přímá práce s typy
    - přístup k parsovaným hodnotám přímo (bez potřeby stringu)
  - vyhazování výjimek při špatné specifikaci

Ukázka kodu

```
class Parser : ParserBase
{
    public StringArgument Str = new("Str", "String arg");
}

var parser = new Parser();
parser.Parse(args);
Console.WriteLine(parser.Str.GetValue());
```

Knihovna se soustředí na běžné případy využití

- volitelný počet argumentů / optionů
- možnost rozšiřitelnosti parsovaných typů

```
class DoubleOption : OptionBase<double?> {
    // defaultValue, constructor
    private double[] values;
    public void Parse(string[] params) {
        try {
            values = new double[params.Length];
            foreach(var p in params) {
                // ...parse into values...
            }
        } catch {
            throw new ParseException("Failed to parse a double value");
        }
    }
    public override double? GetValue(int index = 0) =>
        index < 0 || index >= values.Length
            ? defaultValue : values[index];
}
```

Reakce na komentáře

- Nepodporujeme specifické problémy
  Knihovna nepodporuje vzájemné vylučování parametrů
  Iterace přes všechny naparsované optiony/parametry
  - krajní případy by komplikovaly kód
- Návrhové změny
  Sjednotit třídy Argumentů a Optionů
  - Třídy reprezentující Argumenty a Optiony mají jinou zodpovědnost
  - Sjednocením neusnadníme práci uživateli
  - Sdílení kódu lze dosáhnout jinak (kompozice)

Knihovna je objektově zaměřená

- využívá dědičnosti
  - odvození ParserBase (API)
    - slouží ke specifikaci parseru
    - zároveň umožňuje přístup naparsovaným hodnotám
  - odvození od OptionBase / ArgumentBase (SPI)
    - umožňuje vytváření vlastních optionů / argumentů
- také využívá Reflection, ale o tu se uživatel nemusí starat

Ukázka kódu z time example

```
class Parser : ParserBase
{
    public StringOption format = new (new string[] { "f", "format" }, "Specify output format.");
    public NoValueOption portability = new(new string[] { "p", "portability" }, "Use portable format.");
    public NoValueOption help = new(new string[] { "h", "help", "?" }, "Print help and exit.");
    ...
}
class Program
{
    static void Main(string[] args)
    {
        var parser = new Parser();
        try {
            parser.Parse(args);
        }
        catch (ParseException e)
        {
            Console.Error.WriteLine($"Invalid arguments: {e.msg}");
            Environment.Exit(1);
        }
        if (parser.help.GetValue()) {
            Console.WriteLine(parser.GenerateHelp());
        }
        else {
            ProgramMain(parser);
        }
    }
    static void ProgramMain(Parser parser) {
        var format = parser.format.GetValue();
        //...
    }
}
```

Designové nápady

- definování názvů short a long optionů
  - předpokládáme jednoznakové short, víceznakové long
    - nemusí se specifikovat -, --
  - umožní libovolný počet synonym
- možnosti přijímaní počtu parametrů
  1. enum - omezený výčet možných počtů parametrů
  2. ParameterAccept - struktura obsahující informace o rozsahu
     - flexibilnější, ale trochu komplikovanější
     - k implementaci je třeba menší 'hack' - může být neintuitivní

Designové nápady II

- možnosti přístupu k naparsovaným hodnotám
  1. callbacky - Funkce reagující na na naparsovanou hodnotu by byly příliš jednoduché
     - více práce pro uživatele
  2. parsovaný výsledek v něčem jako Dictionary
     - přístup přes `stringový` název optionu, vrací `object` nebo obecný nadtyp $\Longrightarrow$ musí se přetypovat
     - stringy jsou náchylné na chyby
  3. předem připravené proměnné předané do .AddOption() - skrze ně přístup k parsované hodnotě
     - nestrukturované
     - příliš mnoho proměnných
  4. připravení struktury určující specifikace
     - aktuální řešení