Command Line Parser: Kommandozeilen-Parameter verarbeiten

Programmschalter in einfach

Wer ein Programm für die Kommandozeile schreibt, wird sehr schnell mit der Notwendigkeit von Kommandozeilen-Parametern konfrontiert. Mithilfe der Bibliothek Command Line Parser ist deren Verarbeitung aber ein No-Brainer.

Auf einen Blick

Fabian Deitelhoff lebt und arbeitet in Dortmund. Beruflich ist er als freier Autor, Pluralsight-Autor, Sprecher und Softwareentwickler im .NET-Umfeld tätig. Sie erreichen ihn über www.fabiandeitelhoff.de, Fabian@FabianDeitelhoff.de oder als @FDeitelhoff.

Inhalt

dnpCode
A1412Frameworks

Screencast
auf der DVD

Nicht selten lösen kleine, aber feine Tools komplexe Aufgaben. Aufgerufen und gesteuert über die Kommandozeile leisten sie nicht nur beim Thema Automatisierung sehr gute Dienste. Überall dort, wo Menschen nicht eingreifen können oder sollen, sind sie gefragt. Eine Steuerung über die Kommandozeile fällt aber nicht vom Himmel, sondern muss implementiert werden.

An dieser Stelle kommen Kommandozeilen-Parameter ins Spiel, mit denen man die Programmfunktionen und Abläufe steuert. Genau genommen handelt es sich um Optionen und Argumente, die einem Programmaufruf mit übergeben werden können.

Nur: Das Parsen und Auswerten dieser Argumente und Optionen kann schnell in viel Arbeit ausarten. Denn neben dem Happy Path – alle Parameter sind in Ordnung – müssen auch Fehlerfälle abgefangen werden. Hier reicht das Repertoire von der falschen Reihenfolge über falsche Parameter bis hin zu falschen Daten. Kurz: Die Liste der möglichen Fehlerquellen ist schier endlos.

Die Bibliothek Command Line Parser (CLP) räumt mit diesen Problemen auf und bietet eine Schnittstelle an, mit der Kommandozeilen-Parameter ausgewertet werden können.

Der Command Line Parser

Die CLP-Bibliothek hat ihren Ursprung schon im Jahr 2005. Seitdem wird die Bibliothek gepflegt und aktualisiert. Laut dem GitHub-Repository [1] ist die letzte Aktualisierung vom August 2013, also etwas über ein Jahr her. Das Wiki des GitHub-Repository informiert, dass die CLP-Bibliothek alle Programmieraufgaben abnehmen möchte, die sich ständig wiederholen und damit langweilig sowie fehleranfällig sind.

Folgende Vorteile soll CLP bieten:

- Extra-Code zum Parsen von Kommandozeilen-Argumenten wird überflüssig.
- Wiederverwendung eines leichtgewichtigen API in beliebigen Projekten.
- API und Dokumentation werden ständig aktualisiert.
- Breiter Einsatz in Open- und Closed-Source-Projekten.
- Reger Austausch mit der Community, um die Bibliothek stetig weiterzuentwickeln.

Wie gut diese Vorteile zur Geltung kommen und ob es sich lohnt, die CLP-Bibliothek einzusetzen, zeigen die kommenden Kapitel dieses Artikels.

Ursprung

Der Ursprung der CLP-Bibliothek oder zumindest die grundlegende Idee stammt von der Funktion *getopt*. Diese C-Funktion, die in allen Unix-Betriebssystemen zur Verfügung steht, implementiert das Verarbeiten und Parsen von Kommandozeilen-Parametern.

Dadurch wird ein Standardweg zur Verfügung gestellt, der viele Probleme mit vorherigen Implementierungen löst. Denn diese waren nicht einheitlich. Ein großes Problem war der Umgang mit Parametern, die von vielen Tools anders gehandhabt wurden.

Die Funktion *getopt* [2] ist der Name der entsprechenden C-Funktion und geht auf AT&T im Jahr 1980 zurück. Damit wird der POSIX-Standard [3], der für Portable Operation System Interface steht, implementiert. Zusätzlich existiert noch eine GNU-Erweiterung mit Namen *getopt_long*. Diese implementiert zusätzlich die oft genutzten, benannten Optionen, die mit einem doppelten Trennstrich eingeleitet werden. Diese sind häufig besser zu lesen als die Optionen, die nur aus einem Buchstaben bestehen.

Ein Beispiel für das Verarbeiten von Kommandozeilen-Parametern in C ist auf der GNU-Seite unter [4] zu finden. Getopt ist auch der Name eines Unix-Programms, das das Parsen von Kommandozeilen-Parametern in Shell-Skripten erlaubt.

Mittlerweile existieren auch viele weitere Implementierungen in verschiedenen Sprachen, wie beispielsweise D, Haskell, Java, Perl, PHP, Python, Ruby und C# [5]. Leider sind Java und C# die einzigen Sprachen in dieser Aufzählung, für die es keine direkte Implementierung im Framework gibt. Die *getopt*-Funktionalität muss mit externen Bibliotheken nachgerüstet werden.

Installation

NuGet gestaltet die Installation von CLP sehr einfach. Eine Suche nach der Bibliothek fördert über die NuGet-Galerie einige Treffer zutage. Wichtig ist der Name beziehungsweise die ID des Pakets. Das Originalpaket trägt den Namen

Command Line Parser Library mit der ID *CommandLineParser*. Andere Pakete sind Seitenprojekte oder gänzlich andere Projekte von anderen Autoren. Die aktuelle Version 1.9.71 wird über das Kommando

```
Install-Package CommandLineParser
```

in der Package Manager Console von Visual Studio heruntergeladen und installiert. Alternativ ist natürlich auch die Nutzung der Oberfläche des NuGet-Paketmanagers möglich.

Der Autor Giacomo Stelluti Scala hat sich für eine eigene Lizenz [6] entschieden. Diese gibt im Text aber weitreichende Möglichkeiten, die Bibliothek, die Quellen oder die Dokumentation zu nutzen. Ob das eine gute Entscheidung war und inwiefern diese selbst gewählte Lizenz sich mit anderen Open-Source-Lizenzen verträgt, muss im Einzelfall geprüft werden.

Dieser Artikel sowie der zugehörige Screencast basieren auf der stabilen Version 1.9.71 vom 27. Februar 2013. Der Quelltext der CLP-Bibliothek steht auf GitHub [1] zur Verfügung. Vor dem Wechsel zu GitHub wurde das Projekt auf CodePlex gehostet. Auch dort ist noch eine Projektseite [7] online. Bis auf ein paar Wiki-Einträge zur Dokumentation ist aber insbesondere der vollständige Quelltext zu GitHub umgezogen.

Die Bibliothek steht für das .NET Framework ab Version 3.5 aufwärts zur Verfügung. Ebenso wie auf Mono ab Version 2.1 aufwärts. Zur Installation gehört eine circa 57 Kilobyte große Assembly mit Namen *CommandLine.dll*. Weitere Assemblies, insbesondere Abhängigkeiten, sind nicht vorhanden.

Etwas zur Grammatik

Bevor es um das erste Beispiel geht, zunächst ein paar Details zur unterstützten Grammatik. Das ein oder andere Feature ist schon in der Einleitung zur CLP-Bibliothek angeklungen. Ganz allgemein gibt es bei Kommandozeilen-Parametern die Unterscheidung zwischen Optionen und Werten. Werte können an Optionen gebunden sein oder nicht. Optionen wiederum können über einen kurzen Namen, einen langen Namen oder aus der Kombination von beidem gebildet werden.

Zu den kurzen Namen zählen die typischen Abkürzungen in Form eines einzelnen Buchstabens, der durch einen Bindestrich eingeleitet wird. Beispielsweise in der Form

```
clp -n name
```

In diesem Fall ist *clp* der Dateiname der Konsolenanwendung. Die CLP-Bibliothek stellt diese Art der Optionen über den Datentyp *System.Char* dar. Bei booleschen Werten gilt dieses Prinzip aber nicht. Anstatt einen Wert mit beispielsweise *-t true* oder *-t false* anzugeben, wird der Wert einfach weggelassen: Existiert die Option *-t*, wird sie automatisch auf *true* gemappt. Fehlt die Option ganz, ist das gleichbedeutend mit *false*.

Als weiteres Feature bietet die CLP-Bibliothek das Gruppieren von kurzen Namen an. Mit Gruppieren ist gemeint, dass die Leerzeichen zwischen den Optionen nicht notwendig sind. Mal angenommen, eine Anwendung bietet die Optionen *x*, *y*, *z* und *n* aus dem vorherigen Beispiel an. Die ersten drei Optionen nehmen boolesche Werte entgegen, letztere Option einen Dateinamen. Dann ist die folgende Syntax korrekt:

```
clp -xyzn foo.bar
```

Der Dateiname bezieht sich automatisch auf die Option *n*.

Dabei wird dann aber die Reihenfolge der Parameter wichtig. Umgedreht funktioniert das nämlich nicht:

```
clp -xnyz foo.bar
```

In diesem Beispiel wird die boolesche Option *x* aktiviert. Als Dateiname für die Option *n* wird allerdings *y* als Wert übergeben, denn er folgt der Option *n* als Nächstes. Da auch der Parameter *z* an einen booleschen Wert gebunden ist, wird der eigentliche Dateiname *foo.bar* als ungebundener Wert interpretiert. Valide ist der Aufruf demnach schon und es wird kein Fehler erzeugt. Das Ergebnis ist aber ein ganz anderes als erwartet.

Dieses Verhalten führt direkt zu den sogenannten ungebundenen Werten. Ungebunden deshalb, weil Daten nicht direkt mit einer Option verknüpft sind, also nicht an diese gebunden sind. Dieser Umstand ist erst einmal nichts Schlimmes. Ungebundene Daten als Kommandozeilen-Parameter ergeben in vielen Situationen viel Sinn. Beispielsweise bei einer Anwendung, die viele Dateien verarbeiten muss, wie im folgenden Beispiel dargestellt:

```
clp datei1.txt datei2.txt
   --optimize datei3.txt datei4.txt
```

Hier wird die Anwendung clp mit nur einer gebundenen Option aufgerufen. Nämlich *--optimize*. Da diese Option an einen booleschen Wert gebunden ist, wird sie ohne Daten akzeptiert. Die Dateinamen, hier vier an der Zahl, sind ungebunden, da keine Option angegeben ist.

Im Programmablauf können diese vier ungebundenen Werte trotzdem verwendet werden, da die CLP-Bibliothek diese zur Verfügung stellt und nicht beim Parsen einfach verwirft. So ist es möglich, einer Anwendung viele Daten zu übergeben, ohne jedes Mal Optionen dafür angeben zu müssen.

Im vorherigen Beispiel ist eine Option mit doppeltem Bindestrich vorgekommen. Auch das ist ein typisches Beispiel für Kommandozeilen-Parameter. Sie werden als lange Namen oder lange Optionen bezeichnet. Dahinter verbirgt sich im Grunde genommen nur eine Langform für kurze Optionen. Nicht zu jeder Langform muss es eine kurze Form geben oder umgekehrt, häufig ist das aber der Fall. Für den Namen einer Datei, die zuvor mit der Option *n* eingeleitet wurde, geht somit auch Folgendes:

```
clp --name foo.bar
```

In vielen Fällen ist das deutlich lesbarer. Vorsicht, wenn kurze und lange Optionen gemischt werden. Hier ist es wichtig, Leerzeichen anzugeben. In vielen vorherigen Fällen war das nicht zwingend erforderlich. Das nachfolgende Beispiel ist korrekt:

```
clp -x --name foo.bar
```

Die Option *x* ist wieder an einen booleschen Wert gebunden. Ohne das Leerzeichen zwischen den Optionen funktioniert es allerdings nicht.

Verarbeiten von Optionen

Ein Vorteil der CLP-Bibliothek ist, dass sie nach dem Prinzip Convention over Configuration [8] konzipiert ist. Das bedeutet, dass viele Voreinstellungen aktiv sind, die in den meisten Fällen funktionieren und völlig ausreichend sind; man muss also nicht für jede kleine Entscheidung gleich einen Schalter in der Konfiguration setzen. Auf der anderen Seite geht das aber nicht auf Kosten der Flexibilität, da diese Einstellungen trotzdem noch vorgenommen werden können. Nur eben bei Bedarf und nicht aus Zwang.

Nun aber zu einem ersten Beispiel. **Listing 1** zeigt die Klasse *Arguments*. Sie dient als Speicher für die übergebenen Optionen und deren Daten. Die Kommandozeilen-Parameter, die von der

Listing 1

Datenklasse für die Argumentdaten.

```
public class Arguments
{
    [Option('i', "input", Required = true)]
    public string InputFile { get; set; }

    [Option('o', "output", Required = true)]
    public string OutputFile { get; set; }

    [Option('m', DefaultValue = 3)]
    public int MaximumRows { get; set; }

    [Option('v')]
    public bool Verbose { get; set; }
}
```

CLP-Bibliothek entgegengenommen und verarbeitet werden, sind also nicht nur in einer Auflistung gespeichert, sondern in einem Datenobjekt. Das macht die weitere Verwendung sehr komfortabel und vor allem ein Stück weit sicherer gegenüber Refaktorisierungen.

Denn die Properties der Klasse *Arguments* können einfach abgeändert werden und darauf aufbauender Code ändert sich mit. Natürlich nur, was die Syntax angeht. Semantische Änderungen sind dabei, wie immer, ausgeklammert.

Der Command Line Parser arbeitet mit Attributen, die zu den Eigenschaften hinzugefügt werden. Damit findet auch gleich die notwendige Konfiguration statt. Es gibt also keine externe Konfiguration in Form von ausufernden XML-Dokumenten oder Ähnlichem. Auch dieses Vorgehen hilft bei Refactoring-Maßnahmen.

Das Attribut *Option* hat diverse Überladungen, um den kurzen Namen, den langen Namen und weitere Einstellungen entgegenzunehmen. Die Mindestanforderung ist der kurze Name, damit überhaupt eine Option über die Kommandozeile angesprochen werden kann. Eine Überladung mit sehr vielen Parametern ist die folgende:

```
[Option('i', "input", Required = true,
DefaultValue = "", HelpText = "", Meta-
Value = "", MutuallyExclusiveSet = "")]
```

Zuerst wird der kurze Name angegeben, gefolgt vom langen Namen des Kommandozeilen-Parameters. Anschließend ist es möglich anzugeben, ob der Parameter obligatorisch ist, ob er einen Hilfetext besitzen soll und ob er zu einem Exklusiven-Set gehört. Die Hilfetexte werden im Screencast behandelt.

Von Vorteil ist, dass die meisten Parameter der *Option*-Klasse als optional deklariert sind. Dadurch ist die Angabe der Parameter recht frei und individuell auf die Situation anpassbar. Die implizite Benennung von Optionen ist auch möglich, wenn überhaupt keine Einstellungen über eine Annotation gesetzt werden sollen. Dazu reicht es aus, das Attribut *[Option]* ohne Parameter zu setzen. Dabei wird immer der lange Name in Kleinschreibung aktiv. Sobald der kurze Name angegeben wird, ist nur diese Option aktiv.

Nun ist die Datenklasse parat. Über die Annotationen ist die CLP-Bibliothek in der Lage, in den Eigenschaften die Optionen und Daten der Kommandozeile abzulegen. **Listing 2** zeigt den notwendigen Code zum Parsen der Argumente. Stilsicher eingebettet in einer Kommandozeilen-Anwendung.

Zum Parsen ist nur die statische Methode *ParseArguments* notwendig. Die statische Eigenschaft *Default* beinhaltet einen vorkonfigurierten Parser. Damit sind die Einstellungen gemeint, die optional für den Parse-Vorgang gesetzt werden können. Zunächst sind diese Einstellungen aber nicht weiter wichtig. Alle so über die Kommandozeile ermittelten Daten sind nach dem Aufruf in der Klasse *Arguments* enthalten. Die *if*-Abfrage sorgt noch dafür, dass der nachfolgende Code nur auf die geparsten Informationen zugreift, wenn der Parse-Vorgang positiv verlaufen ist. Mit der Klassendefinition aus **Listing 1** ist nun eine ganze Reihe von Kommandozeilen-Aufrufen syntaktisch korrekt. In der Regel sind das sehr viele Kombinationen, beispielsweise beliebige Variationen kurzer und langer Namen, inklusive der Optionen, die boolesche Werte entgegennehmen.

Für ein einfaches Beispiel war es das auch schon. Die CLP-Bibliothek hält sich angenehm im Hintergrund und bläht den eigenen Code nicht zu sehr auf.

Listing 2

Einfaches Parsen von Argumentdaten.

```
var arguments = new Arguments();

if (CommandLine.Parser.Default.ParseArguments(args, arguments))
{
    Console.WriteLine("InputFile: {0}", arguments.InputFile);
    Console.WriteLine("OutputFile: {0}", arguments.OutputFile);
    Console.WriteLine("MaximumRows: {0}", arguments.MaximumRows);
    Console.WriteLine("Verbose: {0}", arguments.Verbose);
}
```

Einstellungen & Kulturelles

Der bisher verwendete Default-Parser besitzt einen vorgefertigten Satz Einstellungen. Hin und wieder müssen diese Optionen aber überschrieben werden, um das Verhalten beim Parsen anzupassen. **Listing 3** zeigt dazu ein Beispiel.

Im Unterschied zu vorherigen Beispielen wird zunächst ein Parser-Objekt erstellt. Über die Action können direkt im Konstruktor einige Einstellungen übergeben werden. In diesem Fall beispielsweise, ob die Optionen Case-sensitive sind und ob unbekannte Argumente ignoriert werden sollen. Der Rest des Parse-Vorgangs ist dann wie gehabt: Die *ParseArguments*-Methode sorgt für die Verarbeitung und legt die Optionen sowie die zugehörigen Daten in der übergebenen *Arguments*-Klasse ab.

Die Parser-Einstellungen sind zusätzlich noch für zum Festlegen der genutzten Kultur gut – eine Einstellung, die häufig angepasst werden muss. In diesem Beispiel wird über die Zeile

```
settings.ParsingCulture = new
CultureInfo("de-DE");
```

die deutsche Kultur übergeben. Damit wird erreicht, dass beispielsweise Dezimaltrennzeichen bei Kommandozeilen-Parametern korrekt erkannt werden. Ansonsten kann es sein, dass, je nach eingestellter Kultur, der Parse-Vorgang fehlschlägt.

Hintergrund ist, dass die CLP-Bibliothek zunächst alle Kommandozeilen-Parameter als Zeichenketten entgegennimmt. Anschließend durchlaufen die Daten eine Konvertierung, wobei die aktuell eingestellte Kultur des Parsers für

Listing 3

Parser mit angepassten Einstellungen.

```
var parser = new Parser(settings =>
{
    settings.CaseSensitive = true;
    settings.MutuallyExclusive = true;
    settings.IgnoreUnknownArguments = false;
    settings.ParsingCulture = new CultureInfo("de-DE");
});

if (parser.ParseArguments(args, arguments))
{
    Console.WriteLine("InputFile: {0}", arguments.InputFile);
    Console.WriteLine("OutputFile: {0}", arguments.OutputFile);
    Console.WriteLine("MaximumRows: {0}", arguments.MaximumRows);
    Console.WriteLine("Verbose: {0}", arguments.Verbose);
}
```

den Konvertierungsprozess herangezogen wird. Ist diese Kultur nicht passend eingestellt, sind böse Überraschungen vorprogrammiert. Hier muss bei der eigenen Anwendung das Umfeld berücksichtigt werden, in dem die Software zum Einsatz kommt. Das Standardverhalten nutzt hier im Übrigen die *InvariantCulture*.

Verschiedene Argumentdaten

Bisher waren alle Kommandozeilen-Parameter skalare Werte. Eine beliebige Anzahl Zeichen wurde, je nach Datentyp, in einer Variablen abgelegt. So weit, so gut. Aber es gibt auch noch eine ganze Reihe anderer Anwendungsfälle, bei denen die Daten vielleicht anders interpretiert beziehungsweise gespeichert werden sollen.

Die CLP-Bibliothek unterstützt noch weitere Datentypen.

Listing 4

Verarbeiten von Enumerationen als Kommandozeilen-Parameter.

```
public enum Action
{
    Copy,
    Move,
    Delete,
    Rename,
    None
}

public class ArgumentData
{
    [Option('a', "action-type")]
    public Action ActionType { get; set; }
}
```

In **Listing 4** sehen Sie ein Beispiel für Enumerationen. Damit kann der mögliche Wertebereich, der vom Anwender als Parameter mit übergeben wird, gleich eingeschränkt werden. Das Beispiel aus **Listing 4**, das auch gleich die verwendete Enumeration zeigt, erlaubt nun Eingaben der Form

```
clp -a copy
clp -action-type Copy
```

Groß- und Kleinschreibung ist bei den Enum-Werten nicht wichtig.

Auch Arrays unterstützt der Command Line Parser. **Listing 5** zeigt ausschnittsweise den Code für ein Double-Array. Es ist, wie das vorherige Beispiel auch, in der Klasse *ArgumentData* eingebettet. Das Verarbeiten der Kommandozeilen-Parameter bleibt übrigens identisch. Über das Double-Array können nun auch viele Daten auf einmal übergeben werden, wie das folgende Beispiel zeigt:

```
clp --data-values 12 13 14 .5 14.6 44
```

Wichtig sind die Leerzeichen zwischen den einzelnen Werten. Der Aufruf

```
clp --data-values=12 13...
```

funktioniert auch. Ansonsten ist nichts zu beachten.

Was mit Arrays funktioniert, klappt auch bei Listen, wie das Beispiel aus **Listing 6** beweist. Um die Listen von den Arrays zu unterscheiden, ist eine andere Eingabesyntax notwendig:

```
clp -t tag1:tag2:tag3
```

Ansonsten ist die Eingabe so variabel wie bisher bei allen Beispielen. Es funktioniert also auch ohne das Leerzeichen zwischen dem Namen der Option und dem ersten Tag und auch ein langer Name, in diesem Fall *--tags*, ist erlaubt.

Eine kleine Besonderheit bei Arrays und Listen gibt es aber. Die generische *IList*, die im Beispiel zum Einsatz kommt, wird mit einem Null-Wert belegt und nicht durch die CLP-Bibliothek instanziert.

Werden keine Werte übergeben und ist auch kein Default-Wert angegeben, wird beim Zugriff auf die Tags eine *NullReferenceException* ausgelöst. Entweder ist eine Prüfung auf *null* erforderlich oder es wird ein Default-Wert gesetzt. Im Beispiel aus **Listing 6** wird das Problem mit einem leeren String-Array abgefangen.

Was bei der Betrachtung von verschiedenen Daten noch nicht berücksichtigt ist, sind die ungebundenen Optionen. Zur Erinnerung: Das sind die Werte, die nicht direkt an einen kurzen oder langen Namen gebunden sind.

Listing 7 zeigt die Konfiguration für den Fall, dass der Anwendung ungebundene Werte über die Kommandozeile übergeben werden. Die Annotation *ValueOption* sorgt dafür, dass ein konkreter Wert gespeichert wird. Als Parameter wird der Index übergeben, an dessen Stelle der ungebundene Wert steht. In dem Beispiel aus dem vorherigen Listing ist das der Index 0. Zuvor werden alle Kommandozeilen-Parameter an die gebundenen Properties übergeben. Erst wenn dann noch ein Wert übrig bleibt, wird er als ungebundener Wert erkannt. Das Attribut

```
[ValueList(typeof(List<string>),
MaximumElements = 3)]
```

hat das gleiche Verhalten. Allerdings mit dem großen Unterschied, dass eine beliebige Anzahl von ungebundenen Werten aufgenommen werden kann. In dem obigen Beispiel sind es zu Testzwecken drei Stück, die alle vom Typ String sind. Die Liste speichert die ungebundenen Werte immer von einem Typ. Sind anderen Datentypen involviert, müssen diese nachträglich konvertiert werden.

Der folgende Aufruf sorgt dafür, dass *u1* der erste ungebundene Wert ist und in der Property *UnboundString* gespeichert wird. Der Rest wird in die Liste *UnboundList* übernommen:

```
clp u1 u2 u3 u4
```

Wird noch ein weiterer Wert übergeben, gibt die *ParseArguments*-Methode *false* zurück. Das liegt an der Einstellung der ungebundenen Liste, die maximal drei

Listing 5

Definition eines Arrays als Speicherort für Parameter.

```
[OptionArray('d', "data-values", DefaultValue = new[] { .1, 2.2, .3 })]
public double[] Values { get; set; }
```

Listing 6

Definition einer Liste als Speicherort für Parameter.

```
[OptionList('t', "tags", DefaultValue = new string[0])]
public IList<string> Tags { get; set; }
```

Listing 7

Ungebundene Werte speichern.

```
[ValueOption(0)]
public string UnboundString { get; set; }

[ValueList(typeof(List<string>), MaximumElements = 3)]
public List<string> UnboundList { get; set; }
```

Werte aufnimmt. So kann die Datenmenge beschränkt werden, die in Form von Kommandozeilen-Parametern an die Anwendung übergeben werden kann.

Einsatz von Verben

Ein häufiger Anwendungsfall bei Kommandozeilen-Parametern sind sogenannte Verb-Kommandos. Diese speziellen Kommandos sind beispielsweise von git bekannt. Dort gibt es unter anderem die folgenden Verben:

```
git commit ...
git push ...
git tag ...
```

Dabei bezeichnen *commit*, *push* und *tag* Verben, die mit weiteren Optionen und Daten aufgerufen werden können. Verben sorgen im Grunde dafür, dass die verschiedenen Kommandos in Gruppen eingeteilt werden, damit sich die Anzahl der Optionen in Grenzen hält.

Um Verben mit der CLP-Bibliothek nutzen zu können, müssen verschiedene Modelle angelegt werden.

Ähnlich wie bei den *Arguments*- oder *ArgumentData*-Klassen aus den vorherigen Beispielen, nur dieses Mal für jedes Verb eine einzelne Klasse.

Listing 8 zeigt den Code für die Klassen aus diesem Beispiel. *VerbOptions* ist die allgemeine Klasse für die Argumente, wie sie aus den vorherigen Beispielen schon bekannt ist. Allerdings dieses Mal nicht mit konkreten Optionen und Daten, sondern mit zwei Properties, die mit der *VerbOption*-Annotation versehen sind. Hier ist das Verb gespeichert, was in der Argumentliste der Kommandozeile auftaucht. Das *commit*-Verb enthält nur eine *message*-Option und das Verb *push* nur den booleschen Parameter *all*.

Listing 8

Implementierung für verschiedene Verben.

```
public class CommitOptions
{
    [Option('m', "message")]
    public string Message { get; set; }
}

public class PushOptions
{
    [Option('a', "all")]
    public bool All { get; set; }
}

[VerbOption("commit")]
public CommitOptions CommitVerb { get; set; }

[VerbOption("push")]
public PushOptions PushVerb { get; set; }
}
```

Listing 9 zeigt, wie die Verarbeitung aussehen muss, um mit Verben umgehen zu können. Die läuft in diesem Beispiel etwas anders ab. Der *ParseArguments*-Methode wird noch eine Action mitgegeben, die aufgerufen wird, wenn Verben erkannt wurden. In dieser Action wird das Verb-Kommando als Zeichenkette und die Verb-Instanz als Objekt in Variablen abgelegt.

Anschließend können diese Daten einfach ausgewertet werden. Wie diese Verb-Daten gespeichert werden, ist grundsätzlich egal. In diesem Beispiel ist ein sehr einfacher Weg beschrieben. Die Verb-Klassen können auch ein gemeinsames Interface implementieren oder von einer gemeinsamen Klasse erben, um die Verarbeitung bei bestimmten Anwendungsfällen zu erleichtern. So oder so ähnlich, wie das Beispiel aus Listing 9 es gezeigt hat, läuft die Verarbeitung von Verb-Optionen aber immer ab. Durch diese Verben sind ab jetzt Kommandos möglich, die wie folgt aufgebaut sind:

```
clp commit -m "Nachricht"
clp push -a
```

Dieser Aufbau erinnert sehr an Kommandozeilen-Tools wie zum Beispiel git.

Exklusive Argumente

Ein letzter Anwendungsfall, der in diesem Artikel beschrieben wird, sind exklusive Argumente. Exklusiv in dem Sinne, dass bestimmte Argumente nur in bestimmten Konstellationen oder Gruppen genutzt werden können.

An dieser Stelle ist die Dokumentation sehr missverständlich und das vorhandene Beispiel falsch. Die Gruppen fassen nicht erlaubte Argumente zusammen, sondern erlauben, dass nur Optionen aus unterschiedlichen Gruppen genutzt werden können. Listing 10 zeigt die Implementierung der *ExclusiveOptions*-Klasse.

Sie enthält vier Properties, die in zwei Gruppen zusammengefasst sind. Die Konstellation erlaubt Aufrufe der Form

```
clp -a A -c C
clp -d D -b B
```

oder in anderen Kombinationen. Werden beispielsweise die Optionen *-a* und *-b* in einem Aufruf genutzt, gibt die *Parse*-Methode *false* zurück. So ist es möglich, Kommandozeilen-Parameter gegeneinander abzugrenzen. Gibt es allerdings viele verschiedene Gruppen, die jeweils

Listing 9

Parsen und Verarbeiten verschiedener Verben.

```
var verbOptions = new VerbOptions();
var verb = "";
object verbInstance = null;

if (!Parser.Default.ParseArguments(args,
   verbOptions, (v, o) =>
    {
        verb = v;
        verbInstance = o;
    }))
{
    Console.WriteLine("Error");
}

if (verb == "commit")
{
    var instance =
        verbInstance as CommitOptions;

    if (instance != null)
    {
        Console.WriteLine("Message: {0}",
           instance.Message);
    }
}
else if (verb == "push")
{
    var instance = verbInstance as PushOptions;

    if (instance != null)
    {
        Console.WriteLine("All: {0}",
          instance.All);
    }
}
```

Listing 10

Definition exklusiver Argumente.

```
public class ExclusiveOptions
{
    [Option('a', MutuallyExclusiveSet = "gruppe1")]
    public string A { get; set; }
    [Option('b', MutuallyExclusiveSet = "gruppe1")]
    public string B { get; set; }

    [Option('c', MutuallyExclusiveSet = "gruppe2")]
    public string C { get; set; }
    [Option('d', MutuallyExclusiveSet = "gruppe2")]
    public string D { get; set; }
}
```

eine nicht geringe Anzahl von Parametern aufweisen, sind die im vorherigen Beispiel gezeigten Verben eine deutlich bessere Möglichkeit, Gruppen von Optionen zu bilden und zu implementieren.

Um diese exklusiven Gruppen zu aktivieren, ist ein eigener Parser notwendig, der über die Einstellungen den Wert

```
MutuallyExclusive = true
```

übergeben bekommt. Ansonsten ist der Parse-Vorgang identisch mit den vorherigen Beispielen.

Striktes Verarbeiten

Neben der bisher vorgestellten Möglichkeit, die Argumentdaten über die *ParseArguments*-Methode zu verarbeiten, gibt es noch eine strikte Variante. Dazu ist lediglich die Methode *ParseArgumentsStrict* aufzurufen. Diese Methode beendet die Anwendung über ein *Environment.Exit*, wenn ein Fehler beim Parsen auftritt. Das ist die Definition von strikt, da bei den bisherigen Beispielen nur ein *false* oder *true* beim Parse-Vorgang zurückgegeben wurde. Optional nimmt die *ParseArgumentsStrict*-Methode noch einen Action-Parameter entgegen, der bei einem Fehler aufgerufen wird, statt gleich die Anwendung zu beenden. Ein weiterer Unterschied ist, dass bei einem Parse-Fehler direkt die Hilfe angezeigt wird. Mehr zum Thema Generieren und Anzeigen einer Hilfe-Seite verrät der Screencast.

Projektstatus und Forks

Eingangs wurde schon erwähnt, dass die Bibliothek seit einiger Zeit nicht mehr gepflegt wurde. Der Projektstatus ist nicht klar. Bei GitHub sind schon längere Zeit keine Änderungen mehr eingepflegt worden. Zudem ist der Autor schon seit über einem Jahr nicht mehr bei Twitter aktiv gewesen. Alles nicht unbedingt gute Anzeichen, auch wenn die sozialen Medien nicht zu hoch gewichtet werden sollten.

Diesen Umstand haben aber auch andere bemerkt, wie in zahlreichen Diskussionen bei den Issues [9] auf GitHub zu sehen ist. Daher hat ein Anwender der Bibliothek einen Fork erstellt, der unter [10] zu finden ist. Hier ist die Aktivität deutlich höher, was unter anderem dazu führt, dass viele Pull Requests eingearbeitet sind. Aktuell belaufen sich die Änderungen auf über 30 Commits mit zahlreichen Fehlerbehebungen.

Da der Funktionsumfang beider Bibliotheken nahezu identisch ist, hat der Artikel die ursprüngliche Bibliothek als Maßstab genommen. In Zukunft kann es aber durchaus Sinn ergeben, den Fork einzusetzen. Je nachdem, ob die Inaktivität bei der ursprünglichen weiter anhält.

Fazit

Die Command-Line-Parser-Bibliothek ist ein feines Stück Open-Source-Software. Wenn es um das Verarbeiten von Kommandozeilen-Parametern geht, nimmt sie viele Aufgaben und Probleme ab.

Die Inaktivität des Projekts und die an einigen Stellen löchrige Dokumentation schrecken etwas ab. Da ich die Inaktivität von Open-Source-Bibliotheken aber nicht allzu schwer gewichte, habe ich mich für die Vorstellung der Originalversion und keines Forks entschieden. Der Command Line Parser lässt sich trotzdem einfach integrieren und benötigt nicht viele Ressourcen. Daher eine klare Empfehlung und ein abschließend „Sehr gut“ von meiner Seite. **[tib]**

[1] GitHub-Repository der Command-Line-Parser-Bibliothek, www.dotnetpro.de/SL1412Frameworks1

[2] Die getopt-Funktion auf Wikipedia, www.dotnetpro.de/SL1412Frameworks2

[3] Der POSIX-Standard auf Wikipedia, www.dotnetpro.de/SL1412Frameworks3

[4] Beispiele für die getopt-Funktion, www.dotnetpro.de/SL1412Frameworks4

[5] C#-Implementierung für die getopt-Funktion, http://getopt.codeplex.com/

[6] Lizenz der getopt-Bibliothek, www.dotnetpro.de/SL1412Frameworks5

[7] Die Command-Line-Parser-Bibliothek auf CodePlex, http://commandline.codeplex.com/

[8] Das Convention-over-Configuration-Prinzip, www.dotnetpro.de/SL1412Frameworks6

[9] Issues der Command-Line-Parser-Bibliothek auf GitHub, www.dotnetpro.de/SL1412Frameworks7

[10] Beliebter Fork der Command-Line-Parser-Bibliothek, www.dotnetpro.de/SL1412Frameworks8