Použití složitých regulárních výrazů v C#

Bakalářská práce

Jan Prokop

Vedoucí bakalářské práce: Ing. Václav Novák, Csc.

Konzultant bakalářské práce: Bc. Pavel Vacikar

Jihočeská univerzita v Českých Budějovicích

Pedagogická fakulta

Katedra informatiky

Rok 2009

Prohlášení

Prohlašuji, že svoji bakalářskou práci jsem vypracoval samostatně pouze s použitím pramenů a literatury uvedených v seznamu citované literatury.

Prohlašuji, že v souladu s § 47b zákona č. 111/1998 Sb. v platném znění souhlasím se zveřejněním své bakalářské práce, a to v nezkrácené podobě elektronickou cestou ve veřejně přístupné části databáze STAG provozované Jihočeskou univerzitou v Českých Budějovicích na jejích internetových stránkách.

V Českých Budějovicích dne 3.11.2009

Anotace

Práce obsahuje přehled a použití regulárních výrazů v praxi.

Jako ukázka použití je součástí této práce program, který za pomoci regulárních výrazů vyhledává na webových stránkách nové verze nainstalovaných programů a popřípadě je stahuje a instaluje do operačního systému.

Abstract

The thesis contains the summary and application of regular expressions in practice.

For demonstration of practical use of them there is a program in the thesis that, with help of regular expressions, searches for new versions of installed programmes on websites, downloads them if necessary and installs them into the operation system.

Poděkování

Rád bych poděkoval vedoucímu své práce a všem kolegům a kamarádům za podporu a rady při vytváření této publikace.

Obsah

1 ÚVOD....6

2 ZÁKLADNÍ PŘEHLED....7

2.1 Co jsou to regulární výrazy....7

2.2 Kde se můžeme s RV setkat....7

2.3 Jak RV vypadají....8

2.4 Přehled regulárních výrazů....8

2.5 Pamatování....11

2.6 Princip porovnávání....13

2.7 Rychlost....15

2.8 Čitelnost....17

2.9 Testování....18

2.10 Příklady....19

3 REGULÁRNÍ VÝRAZY V C#....24

3.1 Třídy pro práci s regulárními výrazy....24

3.2 RegexOptions....28

3.3 Problémy....30

4 UKÁZKOVÁ APLIKACE....32

4.1 Úvod....32

4.2 Použité technologie....32

4.3 Popis aplikace....32

4.4 Vyhledávací modul....38

4.5 Ostatní třídy....47

5 ZÁVĚR....52

1 Úvod

Velká skupina lidí zabývající se programováním vůbec neví co jsou to regulární výrazy (*angl. regular expressions*). Dokonce i spousta profesionálních programátorů si s tímto mocným nástrojem není úplně jistá a raději ho nepoužívají.

Přitom s regulárními výrazy (*dále jen RV*) se můžeme setkat skoro všude kolem nás. Není to jenom nástroj programátora, ale může ho využívat i obyčejný uživatel v mnoha aplikacích.

Tato publikace se pokusí všem přiblížit kde, kdy a jak s nimi pracovat.

Součástí práce je aplikace využívající RV k vyhledávání aplikací na internetu.

2 Základní přehled

2.1 Co jsou to regulární výrazy

Je to speciální sada znaků, která lze využít k parsování textu. Ať jde o vyhledávání či o nahrazování vždy lze k tomu využít tento systém. Tato sada znaků je v každém programovacím jazyce trošku odlišná, ale princip zůstává nezměněný.

Obecně se uvádějí dva hlavní druhy:

- Perl-compatible (PREG)
- POSIX-compatible (POSIX)

V C# se používá první ze zmiňovaných. Stejně jako v Javě, Pythonu a samozřejmě v Perlu. POSIX syntaxe používá např. ve skriptovacím jazyce PHP. PHP dokáže pracovat i s RV odvozené od jazyka PERL. Z toho vyplývá že častěji se programátor setká s „Perlovskou“ verzí.

2.2 Kde se můžeme s RV setkat

Prakticky na ně můžeme narazit úplně všude. Asi každý programovací jazyk má nějakou tu podporu pro práci s nimi. Na RV může narazit i u většiny textových editorů. Např. málokterý linuxový textový editor dnes neobsahuje podporu pro RV. A nejen textový editor, ale i velká

skupina programů. V operačních systémech Linux se používá také Perl - kompatibilní verze.

Raritou není ani podpora v souborových manažerech při jakémkoli vyhledávání souborů. Dokonce i některé kancelářské aplikace tuto volbu mají, ale pochybuji že by jí „normální člověk“ použil.

2.3 Jak RV vypadají

Jak už bylo naznačeno RV je speciální série znaků, které lze využívat při vyhledávání v textu. Jedná se skupinu složenou z obyčejných znaků a metaznaků. Tedy znaků, které mají jiný význam než obyčejně. A jak vlastně vypadají? Na první pohled by se mohlo zdát, že jde o nesmyslný kus nějakého kódu, ale když se podíváte blíže a rozeberete si ho je to vlastně velice jednoduché.

2.4 Přehled regulárních výrazů

Zde si ukážeme přehled všeho důležitého ke správné konstrukci výrazu. Popíši zde zástupné znaky a jejich význam při hledání shody v textu.

2.4.1 Znaky

Zde je uveden seznam zástupných symbolů pro jednotlivé znaky

`x`	znak x (platí pouze pro alfanumerické znaky)
`\`	vrací původní význam znaku (např \\ zpětné lomítko, * hvězdička)
.	libovolný znak
`\0n`	znak v osmičkovém kódu 0n (0 <= n <= 7)
`\x`*`hh`*	znak v hexadecimálním kódu 0xhh
`\u`*`hhhh`*	znak unicode v hexadecimálním kódu 0xhhhh
`\t`	znak tabulátor ('\u0009')
`\n`	znak nového řádku ('\u000A')
`\r`	znak posunu vozíku ('\u000D')

2.4.2 Skupiny znaků

Pro skupiny znaků se používají následující konstrukce

`[xyz]`	*x, y* nebo *z*
`[^xyz]`	všechny znaky kromě *x,y* a *z*
`[a-z]`	*a* až *z*
`[a-f[k-r]]`	*a* až *f* nebo *k* až *r* jinak: `[a-fk-r]`
`[a-z&&[^k-o]]`	*a* až *z* kromě *k, l, m, n* a *o*

2.4.3 Předdefinované skupiny znaků

Pro často používané skupiny znaků existují již předdefinované symboly:

`\d`	číslice [0-9]
`\D`	opak číslice [^0-9]
`\s`	bílý znak: [\t\n\x0B\f\r]
`\S`	opak bílého znaku [^\s]
`\w`	slovo - alfanumerické znaky [a-zA-Z_0-9]
`\W`	opak slova [^\w]

2.4.4 Kvantifikátory

Pro určení počtu znaků se používají tzv. **kvantifikátory**. Speciální symbol umístěný za znakem nebo skupinou, který určí kolikrát se daný znak může v řetězci vyskytovat.

`?`	žádný nebo jeden
`*`	žádný nebo více
`+`	jednou nebo více
`{n}`	právě n-krát
`{m,n}`	minimálně m-krát, maximálně n-krát
`{n,}`	minimálně n-krát

2.5 Pamatování

Vlastnost, která by se neměla být v žádném případě přehlídnuta, je pamatování nalezené shody. A nejde jen o shodu celého řetězce, ale i o jeho části. Části se rozdělují do skupin.

2.5.1 Skupiny

Skupiny se ohraničují obyčejnými závorkami `( )`.

Např.: Chceme-li najít text mezi uvozovkami:

```
\"([^\"]*)\"
```

V tomto případě nám řetězec mezi závorkami definuje skupinu s pamatováním. K této skupině můžeme později přistupovat, většinou podle čísla. A dokonce i v rámci psaného výrazu. Jako názorná ukázka nám poslouží výraz pro hledání palindromů, tedy slov které se čtou zepředu i zezadu stejně:

```
(\w)(\w).\2\1
```

Tento výraz vyhoví všem 5tipísmeným palindromům. Pozor tato zpětná reference neodkazuje na stejnou skupinu, ale na skutečný nalezený znak. Tzn., že v naší ukázce: `\1 != \2`

2.5.2 Pojmenování skupin

Pro lepší přehlednost se nabízí možnost pojmenování skupin pomocí speciální konstrukci skupiny (?<jméno_skupiny>) . Otazník hned za závorkou nám určuje vlastnosti skupiny. Toto je ukázka pojmenování.

```
(?<pismeno>\w)(\w).\1\k<pismeno>
```

K pojmenované skupině pak ve výrazu přistupujeme pomocí

\k<jméno_skupiny> jak je vidět už v ukázce. V C# pak pomocí

Match.Groups["jméno_skupiny"] *(Viz. dále)*

2.5.3 Opakování skupin

Skupiny nemusí sloužit jen k pamatování, ale můžeme takové skupině nastavit i počet opakování. Takový výraz si jistě také bude něco pamatovat, ale to je momentálně nepodstatné.

Př. V případě telefonního čísla:

```
(\d{3}[\s\-]?){3}
```

2.5.4 Nepamatování

Další vlastnost skupiny, kterou můžeme definovat je *nepamatování*. Výsledkem procesu jsou pak jen informace, které skutečně požadujeme. Definice nezapamatovatelné skupiny začíná opět otazníkem ihned za závorkou následovaný dvojtečkou `(?: )`. Nezapamatovaná skupina už není přístupna pomocí zpětné reference. Prakticky to pak vypadá třeba takto:

```
class=\"(?:[^\"]*)\"\s*([\w]+)
```

Hledáme vše co je jako dalším parametrem po *class* a co se objevuje v uvozovkách nás nezajímá, proto si tento údaj nemusíme pamatovat.

2.6 Princip porovnávání

Základní proces porovnávání začíná od začátku zkoumaného řetězce. Každému ze znaků se systém snaží přiřadit co největší kus textu. Pokud porovnávání selže zkrátí se porovnávaný text o jeden znak a začne se opět od začátku.

2.6.1 Velikost písmen

V základu jsou regulární výrazy case sensitive tedy citlivé na velikost písmen. I když toto porovnávání jde většinou vypnout musíme si dát pozor. V C# to jde udělat pomocí volby `RegexOptions.IgnoreCase`. Nebo to zahrnout v samotném výrazu např. `[a-zA-Z]`

2.6.2 Hladovost

Regulární výrazy jsou hladové. To znamená, že se vždy snaží výsledné shodě přiřadit co největší kus textu.

Příklad: Hledání textu mezi uvozovkami:

Kdybychom na to šli nejjednodušeji, jak by to šlo, jistě by nás napadlo toto řešení:

```
\"(.*)\"
```

Při aplikaci na řetězec: "pes" nebo "kočka" bychom očekávali výsledky *pes* a *kočka*, ale tak to není. Právě hladovost regulárních výrazů nám vrátí, možná trochu neočekávaný výsledek: `pes" nebo "kočka`. Proto výraz musíme upravit tak, abychom se takové situaci vyhnuli. Hledáme vše kromě uvozovek.

```
\"([^\"]*)\"
```

Takto dostaneme to, co požadujeme.

Další možností, jak se vyhnout neočekávané hladovosti, je použití „**líných kvantifikátorů**“. Tím při prohledávání nedostáváme největší možný řetězec se shodou, ale naopak nejmenší možný. Jde o zařazení symbolu `?` za kvantifikátor znaku.

Příklad

vezměme stejný řetězec "pes" nebo "kočka"

upravíme výraz `\"(.*)\"` přidáním líného kvantifikátoru:

```
\"(.*?)\"
```

V tuto chvíli dostaneme nejmenší možný text mezi uvozovkami. Takže dostáváme to co jsme čekali už v prvním případě, tedy dvě skupiny se slovy *pes* a *kočka*.

2.7 Rychlost

Obecně se tvrdí, že použití regulárních výrazů je o hodně pomalejší než využití standardních operací s řetězci. Toto je veliká pravda, avšak má jedno veliké „**ale**“. Předpokladem je, že pokud můžeme použít jednoduché metody jako je `string.Replace()` v místě, kde je potřeba dbát na rychlost, měli bychom tak učinit. Např.:

```
string.Replace("test");
Regex.Replace("test");
```

V takovéto situaci bude metoda `Regex.Replace()` značně pomalejší. V případech, kdy nevíme přesnou podobu hledaného řetězce, bude tato metoda naopak mnohem jednodušší a ve většině případů bude asi i rychlejší než prohledávat text nějakou sérií standardních metod a porovnávat zda vyhovují tomu, co hledáme. V takovém případě bychom museli použít metody jako `string.EndsWith()`, `string.Contains()`, `string.IndexOf()`, `string.IndexOfAny()` apod. Takový proces by byl o mnoho složitější a mohl by být i značně pomalejší. V některých případech dokonce i nemožný.

Obrovský vliv na rychlost prohledávání má také to, jak je výraz napsán. Například když použijeme výraz `[\w]*` na hledání všech

alfanumerických znaků, bude procházení textem pomalejší než použití prakticky toho samého jen jinak napsaného `[a-zA-Z_0-9]*` . V tomto případě časový rozdíl může být zanedbatelný, ale ve složitějším výrazu se rozdíl zvětšuje.

Dále je rozdíl, zda použijeme zkompilování `RegexOptions.Compiled` či nikoliv. Tady také záleží na složitosti výrazu.

2.8 Čitelnost

Regulárním výrazům je často přisuzována nečitelnost napsaného kódu. Z určitého úhlu pohledu to může být pravda. Delší výraz se opravdu stává na první pohled nečitelným. Ale co v případě, kdy potřebujeme nahradit několik znaků?

Příklad:

bez RV

```
r = r.Replace("1","");
r = r.Replace("2","");
r = r.Replace("3","");
r = r.Replace("4","");
r = r.Replace("5","");
r = r.Replace("6","");
r = r.Replace("7","");
r = r.Replace("8","");
r = r.Replace("9","");
r = r.Replace("0","");
```

s RV:

```
r = Regex.Replace("[0-9]","");
```

Výhoda v tomto zápisu je zřejmá. Sice pomalejší nahrazení, ale v čitelnosti kódu vyhrávají RV. Půjdeme-li ještě dál a chtěli bychom smazat místo čísel písmena:

```
r = r.Replace("a","");
...
r = r.Replace("z","");
```

místo:

```
r = Regex.Replace("[a-z]","");
```

A to nebereme v úvahu velká a malá písmena. Zde se výhra v čitelnosti připisuje reg. výrazům.

2.9 Testování

Je dobré si svůj výraz odzkoušet, zda plní požadovanou funkci. Náš výraz může být obtížné testovat uprostřed nějaké větší aplikace. Proto existuje mnoho nástrojů, jak si svůj výraz otestovat, jestli dělá, to co má. Dokonce se můžeme setkat s řadou on-line nástrojů, které nám jako jednoduchý test dobře poslouží. Nebudu zde uvádět žádné konkrétní nástroje abych, nějakému nekřivdil a jinému dělal reklamu. Je na každém, aby si vybral co mu vyhovuje. Nějaký nástroj na testování může obsahovat i vaše vývojové prostředí.

2.10 Příklady

Uvedu zde několik příkladů, které se v praxi často používají. Provedu jejich podrobný rozbor, který umožní lépe pochopit, jak takový výraz funguje a jak jej vytvořit.

2.10.1 PSČ

Pro začátek jednoduchý příklad na to, jak vyhledat poštovní směrovací číslo v textu. O PSČ víme, že je to 5 po sobě jdoucích čísel `\d{5}`. Většinou se mezi 3. a 4. číslem píše mezera, proto musíme výraz rozdělit`\d{3}\d{2}`. Teď ještě doplníme nepovinnou mezeru a je hotovo. Takže výsledek vypadá takto:

```
\d{3}\s?\d{2}
```

2.10.2 E-mail

Jako další příklad jsem vybral notoricky známý příklad na vyhledávání a ověřování emailové adresy:

```
^[\d\w\.\-]+@[\-a-z0-9]+\.{1}[a-z]{2,}$
```

Znaky `^` a `&` na začátku a konci označují začátek a konec zkoumaného řetězce. Pokud bychom chtěli najít adresu v nějakém textu, tak tyto značky vynecháme. Nejprve musíme určit skupinu znaků, které se může vyskytovat před znakem @. V případě emailové adresy to jsou znaky: od a do z, čísla od 0 do 9, tečka (.) a pomlčka (-). V regulárních výrazech to můžeme napsat několika způsoby. Ve výše uvedeném jsou použity

rovnou dva způsoby (v druhém je pouze vynechána tečka). Dále také předpokládáme, že ve vyhledávání ignorujeme velikost písmen, jinak totiž musíme přidat další skupinu znaků a to velká písmena (třetí možnost):

- `[\d\w\.\-]`
- `[0-1a-z\.\-]`
- `[0-1a-zA-Z\.\-]`

Po zavináči může následovat téměř to samé, výjimku tvoří tečka. Ta totiž uvozuje doménu, u které platí trošku jiná pravidla. U domény není předpokládané číslo a má minimálně dva znaky: `[a-z]{2,}`.

2.10.3 Odkaz

Další často vyhledávaný text může být nalezení odkazu na stránce. Tedy internetové adresy v HTML kódu.

V html je odkaz umístěn v tagu `<a>` a konkrétní odkaz je v atributu `href`. Toho využijeme i v našem výrazu:

```
<a\s+[^>]*\s*href="?([^"\s]*)\s*[^>]*>
```

Tento výraz nám vyhledá vše co obsahuje atribut `href` v tagu `<a>` a samotný odkaz máme uložen v samostatné skupině.

Když si výraz rozebereme zjistíme, na co všechno si musíme dávat pozor. Tak třeba hned na začátku po `<a` následuje mezera \s. Tato mezera tam být musí, ale nikdy nevíme, kolik mezer tam autor ve skutečnosti napsal. Proto volíme kvantifikátor + tedy jednou a vícekrát.

Dalším bodem úrazu může být to, co následuje za atributem `href=`. Podle validity kódu by měli následovat uvozovky, v nichž by se měl nacházet samotný odkaz. To však nemusí být vždy pravda, proto volíme kvantifikátor „*může ale nemusí*“ tedy `?` - vůbec nebo jednou.

Pokud bychom chtěli najít odkazy začínající řetězcem „http://“, pouze to přidáme do našeho výrazu.

```
<a\s+[^>]*\s*href="?http://([^"\s]*)\s*[^>]*>
```

2.10.4 IP adresa

Další častou věcí kterou může chtít nalézt v textu, třeba v nějakém logu, je IP adresa. Ukáži zde i postup při tvorbě takového výrazu. Uvažujeme-li IPv4. Podobně postupujeme i v případě IPv6. Sestavený výraz může mít několik podob:

```
1. \d+\.\d+\.\d+\.\d+
2. \d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
3. (\d{1,3}\.){3}\d{1,3}
4. ([1,2]?\d{1,2}\.){3}[1,2]?\d{1,2}(?:[\D])
```

1. V prvním případě se jedná pouze o sérii čtyř čísel oddělených tečkou. Jistě bychom tímto nějakou IP adresu našli, ale počet čísel není nijak omezen, takže vyhovuje i např. takovéto adrese: `1234.1234.1234.1234` Taková IP adresa ovšem neexistuje.

2. V druhém případě tento nedostatek upravujeme tím, že čísla mezi tečkami nastavíme na minimálně jednomístné a maximálně třímístné. Zdá se, že máme vyhráno, ale ještě to není stoprocentní. Takto napsaný výraz je přesnější, ale stále by se našla adresa, která neexistuje: `999.999.999.999`
3. Ve třetím výrazu přistupujeme k problému trošku jinak. Pohlížíme na adresu jako na tři po sobě jdoucí sérii čísel maximálně třímístné následované tečkou. Čtvrté číslo za sebou tečku nemá, proto je napsané zvlášť. Tento výraz má stejnou slabinu jako výraz předchozí.
4. Ve čtvrtém jsme se snažili tento nedostatek odstranit. Jasně jsme stanovili, že první číslo, v případě třímístného čísla, musí být 1 nebo 2. To už nám chyby omezuje jen na adresy které jsou větší než 255, ale zároveň už nemohou být větší 299.

Pokud bychom chtěli 100% výraz pro IP adresu musíme přesně specifikovat její podobu:

```
(((\d{1,2})|(1[\d]{2})|(2[0-4][0-9])|(25[0-5]))(?:\.)){3}
(\d{1,2}|1[\d]{2}|(2[0-4][0-9])|(25[0-5]))(?:[\D])
```

Používáme zde několik výrazů spojené logickou spojkou nebo `|`.

- První může být jakékoliv jedno až dvojciferné číslo (0 – 99).
- Dále to může být trojciferné číslo začínající 1xx následované jakýmkoliv dvojciferným číslem (100 – 199).

- Další možností je třímístné číslo začínající na 2xx. To ovšem může následovat jen číslo do 55. To nám zajistí dva následující výrazy.
- Tato skupina je vždy následovaná tečkou. A opakuje se 3krát po sobě.
- V posledním případě tomu tak není, proto je tato skupina naspaná opět zvlášť.

Takový výraz by neměl mít žádnou z výše zmiňovaných slabin ani žádnou jinou.

3 Regulární výrazy v C#

3.1 Třídy pro práci s regulárními výrazy

Třídy pro práci s regulárními výrazy v C# .NET se převážně vyskytují v `System.Text.RegularExpressions.` Tento vyhraněný namespace obsahuje všechny důležité třídy pro práci RV.

Dále popisuji ty nejhlavnější:

3.1.1 Regex

Hlavní třída pro používání regulárních výrazů. Obsahuje několik statických metod, které lze využívat bez nutnosti vytvoření její instance. Jako například *Match, Matches, Split, IsMatch* nebo *Replace.*

Uvedu zde nejdůležitější metody, které třída *Regex* obsahuje

- `Match Match()`

  Prochází řetězec a hledá shodu s regulárním výrazem. Vrací instanci třídy *Match*.

- `MatchCollection Matches()`

  Prochází řetězec a hledá shodu s regulárním výrazem. Vrací všechny výskyty v *MatchCollection*.

- `bool IsMatch()`

  Tato metoda pouze zjistí zda daný řetězec obsahuje RV.

- `string[] Split()`

  Rozdělení řetězce podle RV.

- `string Replace()`

  Nahrazování nalezených částí textu podle RV jiným textem.

Základem je vždy, když využíváme statickou metodu přidat jako parametr i regulární výraz. Při využívání nestatické metody tato povinnost odpadá, pokud reg. výraz uvedete při vytváření instance.

3.1.2 Match

Třída reprezentuje shodu. Je vrácena funkcí *Regex.Match()* nebo v kolekci *MatchCollection* funkcí *Regex.Matches()*. Důležitou součástí této třídy jsou vlastnosti *Value, Groups, Success* a *Index.*

- `string Value`

  Vlastní nalezená hodnota.

- `GroupCollection Groups`

  Obsahuje pole skupin definované v RV. K jednotlivé skupině přistupujeme pomocí indexu skupiny nebo podle jména skupiny:

  - `Match.Groups[2]`
  - `Match.Groups["jméno_skupiny"]`

- `bool Success`

  Důležitá informace o tom zda vyhledávání uspělo či nikoli.

- `int Index`

  Udává pozici v originálním textu.

3.1.3 MatchCollection

Kolekce, která je vrácena funkcí `Regex.Matches()`. Důležitá pro operace s více výskyty v textu. Je to kolekce všech částí textu shodujících se s RV. Jednotlivé položky jsou typu *Match*.

3.1.4 Group

Jde o uložení nalezené skupiny během hledání. Skupiny se definují samotným výrazem.

Obsahuje:

- `int Index`
  - Pozice v původním textu
- `int Length`
  - Velikost nalezeného řetězce
- `string Value`
  - Vlastní hodnota nalezeného řetězce

3.2 RegexOptions

Výčtový typ, který obsahuje volby pro nastavení a možnosti objektu typu *Regex*. Jednotlivé volby se oddělují znakem | a proto není problém použít více voleb najednou. Uvedu zde pouze ty nejdůležitější volby.

3.2.1 Compiled

Pokud použijeme tuto volbu, tak bude použití RV rychlejší při používání, ale zabere nějaký čas na začátku. Dojde totiž ke zkompilování do *assembly*. Proto ji používáme pouze, pokud víme, že budeme prohledávat velký text nebo budeme prohledávat častěji stejným výrazem.

3.2.2 IgnoreCase

Často používaná volba, která si nařídí reg. výrazu opomíjet velikost písmen. Potom nemusíme zadávat například skupinu *[a-zA-Z]*, ale stačí pouze skupina *[a-z]*. Jistě, že v případech, kdy potřebujeme rozpoznávat velikost písmen je tato volba nežádoucí.

3.2.3 RightToLeft

Určení směru prohledávání zprava doleva. Výchozí je zleva do prava.

3.2.4 Multiline

Tato možnost mění význam symbolů ^ a *&*, které už neoznačují začátek a konec řetězce, ale značí začátek a konec řádku.

3.2.5 Singleline

Pozmění význam symbolu tečky (.). Tento znak normálně zastoupí všechny symboly. Bez použití tohoto módu tečka nezastupuje symbol nové řádky (*n*).

3.2.6 None

Určuje, že není použita žádná konkrétní volba.

3.2.7 Ukázka použití

- Příklad použití jediné volby z `RegexOptions`:

```
Regex.Match(text,pattern, RegexOptions.IgnoreCase);
```

- Použití více voleb najednou:

```
Regex.Match(text,pattern,
RegexOptions.IgnoreCase | RegexOptions.Compiled);
```

3.3 Problémy

Samozřejmostí je, že nic se neobejde bez problémů. Uvedu zde nejčastější problém při práci s RV.

3.3.1 Escape sekvence

Regulární výraz velice často obsahuje znak \ (zpětné lomítko). Ovšem v C# zpětné lomítko v řetězci naznačuje nějakou escape sekvenci.

Například

Máme tento RV:

```
(http:\/\/[^\/]+)\.*
```

Pokud bychom chtěli tento výraz uložit do proměnné, vypadalo by to asi takto:

```
string x = "(http:\/\/[^\/]+)\.*";
```

Tento řetězec je zcela jistě **špatně**. Za zpětným lomítkem následuje lomítko obrácené nebo tečka, která nesymbolizuje žádnou sekvenci. A i kdyby za lomítkem náhodou následoval nějaký symbol představující esc. sekvenci (např. `n` nebo `t`) způsobovalo by to jen problémy. To znamená, že tento řetězec bychom museli napsat takto:

```
string x = "(http:\\/\\/[^\\/]+)\\.*";
```

Zdvojením zpětných lomítek se vyhneme syntaktické chybě, ale tento zápis začíná značně komplikovat čitelnost již tak špatně čitelného výrazu.

Proto se velice často setkáme s tímto zápisem:

```
string x = @"(http:\/\/[^\/]+)\.*";
```

Před uvozovky obklopující výraz napíšeme zavináč, který překladači řekne, že má ignorovat jakoukoliv escape sekvenci.

Další problém nastane, pokud výraz obsahuje uvozovky. Protože pokud totiž dáme před řetězec informaci o ignorování escape sekvencí, každý výskyt uvozovek bude znamenat ukončení řetězce. Tudíž tento postup nelze použít a musíme se vrátit k řešení pomocí zdvojení zpětných lomítek nebo použijeme možnosti zdvojení uvozovek.

Např. výraz: `<a\s*href\s*=\s*"?([^">\s]+)"?`

Chybný zápis výrazu escape sekvencí

```
quote = "<a\s*href\s*=\s*\"?([^\">\s]+)\"?";
```

Zápis pomocí zdvojením lomítek

```
quote = "<a\\s*href\\s*=\\s*\\\"?([^\\\">\\s]+)\\\"?";
```

Zápis pomocí zdvojením uvozovek

```
quote = @"<a\s*href\s*=\s*""?([^"">\s]+)""?";
```

4 Ukázková aplikace

4.1 Úvod

Jako demonstrace použití „síly“ regulárních výrazů v C# je součástí této práce aplikace, která vyhledává na internetu nové verze programů.

4.2 Použité technologie

- Programovací jazyk **C#**
- **.NET Framework 2.0**
- podpora OS **Microsoft Windows XP / Vista / 7**
- vývojové prostředí **SharpDevelop 2.2.1**

4.3 Popis aplikace

Popíši zde jak postupuje aplikace od spuštění a stručně vysvětlím několik důležitých okamžiků a procesů.

4.3.1 Spuštění

Těsně po spuštění je nejprve načten soubor s konfigurací *setting.dat*, ve kterém je uložen název vyhledávacího modulu, který se má použít jako výchozí. Pokud soubor s konfigurací chybí, bude vytvořen. Pokud chybí soubor s uloženým modulem, použije se výchozí, který je zkompilován již v aplikaci.

Po načtení modulu se volá třída *RegistryInformations*, která má na starosti práci s registry.

4.3.2 Informace o nainstalovaných programech

Veškeré informace o programech, které se nacházejí v počítači, na kterém je aplikace spuštěna se získávají z registrů. Informace o nainstalovaném softwaru se nachází v registru:

- `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall\` pro **Windows XP** a **Windows Vista**
- `HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows\CurrentVersion\Uninstall\` pro **Windows 7**

Běžně mají programy někde uloženou i ikonu programu. Ikona se také zjistí ze systémových registrů. Ikona však není pravidlem proto se program pouze pokusí ji najít. Ikony jsou uloženy v registrech na odlišném místě než samotné informace o aplikacích.

```
HKEY_CLASSES_ROOT\Installer\Products
```

Tuto informaci získáme pomocí této metody

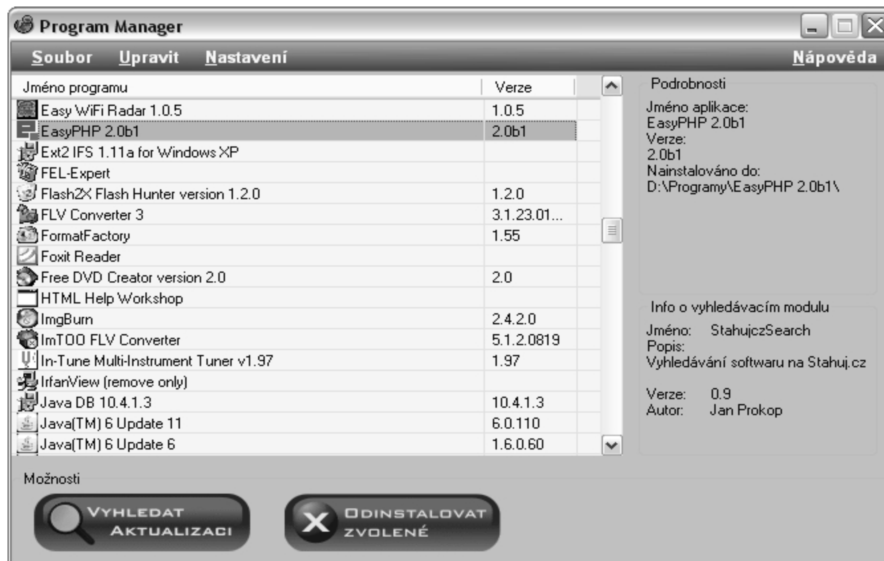
```
Registry.ClassesRoot.OpenSubKey("Installer\\Products")
```

V případě neúspěchu se použije defaultní.

Všechny tyto informace nám obstarává třída *RegistryInformations*. Její metoda `GetInformations()` nám naplní seznam informací o nainstalovaných programech v systému. Podrobnosti o aplikacích jsou uloženy v generickém seznamu `IList<RegistryEntry>`.

4.3.3 Výběr hledané aplikace

V hlavním okně aplikace se tedy nalézá seznam nainstalovaných programů v počítači s ikonou a číslem verze. Když nějakou aplikaci zvolíme zobrazí se v pravém horním poli dostupné informace. Po tomto výběru můžeme zvolit odinstalování nebo vyhledávání novější verze. Odinstalování si zvolená aplikace obstarává sama. Je pouze zavolán příkaz, který je uveden v systémových registrech v položce `UninstallString`.



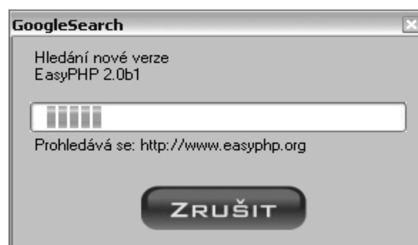
Hlavní okno aplikace se seznamem nainstalovaných programů

4.3.4 Vyhledávání

Pokud zvolíme vyhledávání novější verze, zavolá se metoda `Search()` načteného modulu.

Celá aplikace je napsána tak, aby samotné vyhledávání měl na starosti zásuvný modul v podobě dynamicky připojované knihovny implementující rozhraní `ISearchEngine`. Tento proces je podrobněji popsán v další kapitole.

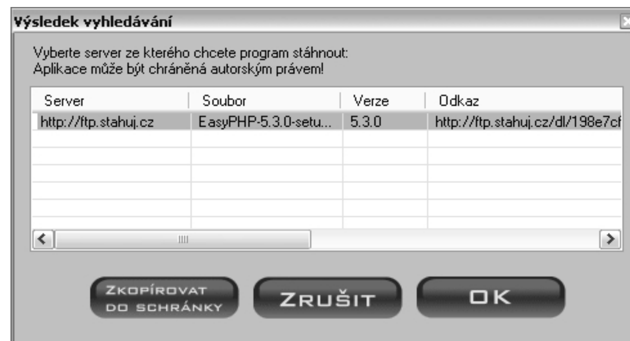
Celý proces probíhá v pracovním vlákně, aby neblokoval aplikaci. Při tomto hledání uživatel vidí okno s průběhem, právě prohledávaným serverem a možností přerušení.



Okno průběhu vyhledávání

4.3.5 Stahování

Seznam nalezených odkazů pro stažení aplikace se předají formuláři ve třídě *SearchReport*.



Okno SearchReportu s výsledkem vyhledávání

Po výběru serveru, který provede uživatel, se začne příslušný soubor stahovat do adresáře *downloaded* umístěného v adresáři s naší aplikací. Veškeré stahování má na starosti třída *DownloadCore*.

4.3.6 Spuštění instalace

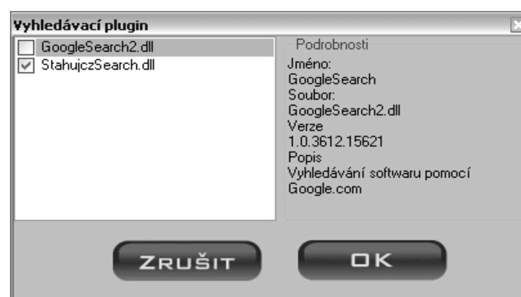
Při úspěšném stažení programu z internetu se náš program zeptá, zda se soubor má spustit. V případě záporné odpovědi se pouze otevře umístění se staženými soubory.

4.4 Vyhledávací modul

V těchto modulech se vlastně odehrává většina práce. Zde se aplikuje nejvíce regulárních výrazů při prohledávání webu. Vyhledávací modul je hlavní prezentace použití **regulárních výrazů** v praxi. Samozřejmě implementuje rozhraní *ISearchEngine*.

4.4.1 PluginLoader

Jedná se o formulář, který prochází všechny *dll* soubory v adresáři *plugins* a nabízí je uživateli k použití. Zvolený soubor se potom pokusí načíst a vytvořit instanci *ISearchEngine.*



Okno PluginLoaderu

4.4.2 Rozhraní ISearchEngine

Jde o rozhraní, které musí být implementováno vyhledávacím modulem. Obsahuje pouze základní vlastnosti, jednu metodu a jednu událost.

Vlastnosti sloužící pouze jako informace o modulu:

- `string PluginName` – jméno modulu
- `string PluginDescription` – popis modulu
- `string PluginVersion` – verze modulu
- `string PluginAuthor` – autor modulu

Vlastnosti které informují systém:

- `string Host` – aktuální prohledávaný server
- `string CanceledSearch` – přerušení vyhledávání

Jediná povinná veřejná metoda

```
IList<T> Search(string appName, string appVersion)
```

které je jsou předány dva parametry:

- `string appName` – jméno aplikace
- `string appVersion` – verze aplikace

Metoda by měla vracet nalezené informace v generickém seznamu. Typ položek seznamu jsou instance třídy `LinkInfo`.

4.4.3 Modul StahujczSearch

Tento modul využívá serveru *www.stahuj.centrum.cz*. Všechny odezvy od serveru jsou textové. To znamená, že k dispozici máme pouze čistý HTML kód, který musíme zpracovávat. Pomocí specifické adresy získáme výsledky vyhledávání na serveru:

```
string q = "http://www.stahuj.centrum.cz/?g[s]="+name;
```

Na tento výsledek musíme aplikovat regulární výraz, který nám výsledky z textu vytáhne. Výraz vypadá takto:

```
<p\sclass="software\-title">
<h3><a\s?[^>]*\s?href="([^"]*)\s?[^>]*>
```

Pomocí tohoto RV nalezneme odkaz v tagu:

```
<p class="software-title">
    <h3>
          <a href = "{hledaný odkaz}">
```

Tímto pouze získáme skupinu adres odkazující na popis programů. Proto musíme dále získat odkaz pro stažení. K tomu použijeme další výraz, který vypadá takto:

```
<h3\sclass="first\-item">\s*<a\s?[^>]*\s?href="([^"]*)\s?
[^>]*>
```

Toto ještě není všechno. Až z toho co získáme z nalezeného odkazu můžeme konečně najít přímou adresu na soubor. Tu získáme další aplikací regulárního výrazu.

```
<div\sclass="dl-cont">\s*.*<a\s?[^>]*\s?href="([^"]*)\s?
[^>]*>
```

Toto je konečně přímí odkaz na soubor. Z předchozí stránky se ještě pokusíme nalézt verzi aplikace. To provedeme výrazem:

```
<h3\sclass="first\-item">
<a.*>\D*
(?:v)?([\d]+[\-\.]+[\d\w]+(?:[\-\.]?[\d\w]*)*)
.*</a>
```

Hledáme odkaz za názvem aplikace

Ve struktuře v HTML kódu to vypadá následovně:

```
<h3 class="first-item">
   <a href= ...>
         název aplikace {hledaná verze}
   </a>
```

Formát verze není jednoznačně určený, proto ošetřuje výraz pro různé varianty. Verze může mít předpokládaný tvar:

```
major . minor . build
```

Tzn.

```
číslo (tečka) číslo (tečka) číslo
```

Toto je asi nejčastější formát, ale není jediný. Ošetřit musíme také tvary které mohou vypadat takto:

- `5.1, 5.01, 5.001`
- `3.2b, 2.7Pro`
- `5.0.8.5`

Seznam všeho nalezeného vrací modul zpátky třídě, která volala metodu **`Search()`**.

4.4.4 Modul GoogleSearch

Jde o komplexnější vyhledávání na internetu. Neprohledáváme totiž jeden konkrétní server, ale pomocí vyhledávacího serveru nalezneme výskyty námi hledané aplikace na různých serverech. Jelikož se při prohledávání dostáváme na různé servery s neznámým obsahem a strukturou, nemusí být výsledek našeho snažení úspěšný tak jako v předchozím modulu.

Základem vyhledávání je nejprve stažení textové reprezentace webové stránky ze serveru *Google.com* s patřičnými parametry pro hledání programu.

```
string gstring =
"http://www.google.com/search?
hl=cs&q="+q+"&ie=utf-8&oe=utf-8&num=15";
```

Řetězec `q` obsahuje vyhledávací dotaz.

Dále zde je parametr specifikace jazyka(`hl=cs`), vstupního(`ie=utf-8`) a výstupního(`oe=utf-8`) kódování znaků a počet zobrazovaných výsledků(`num=15`).

Textovou reprezentaci získáváme pomocí třídy `WebClient`, která nám zajistí komunikaci s požadovaným serverem. V tomto HTML kódu musíme najít specifické elementy se správnými údaji o www stránce, kde by se mohl nalézat námi hledaný program. K tomu nám výborně poslouží právě **regulární výrazy**.

Z textové odpovědi serveru *google.com,* díky jeho jedinečné struktuře nalezených odkazů, vyhledáme pomocí výrazu:

```
class="?r"?\s*><a\s*href\s*=\s*"?([^">\s]+)"?
```

všechny výsledky hledání požadované aplikace.

Tyto výsledky musíme všechny projít a testovat na přítomnost nějakého odkazu na stažení. Jelikož servery jsou různé s různou strukturou – musíme výraz vytvořit co nejuniverzálněji.

Nějak takto:

```
<a.*\s*href\s*=\s*\"?([^\">\s]+)\"?[^>]*>
(.*download.*|.*downloads.*|.*install.*|.*save.*|.*uložit.*
|.*stáhnout.*|.*direct\slink.*)</a>
```

Tímto získáme možné odkazy na stažení hledané aplikace.

4.4.5 Jak vytvořit zásuvný modul

Uvedu zde jak takový modul vytvořit pro případné rozšíření/vylepšení aplikace.

Způsobů je několik

- **Pomocí příkazové řádky**
    - Celé vytvoření knihovny je zrealizováno pomocí *csc.exe*. Jde o kompilátor .NET Frameworku. Kompilátor je umístěn v kořenovém adresáři operačního systému
        - `%WINDIR%\Microsoft.NET\Framework\v2.0.50727\`
    - Celý plugin musíme nejprve napsat v nějakém textovém editoru jako třídu
    - Třídu pro ukázku pojmenuji *Modul.cs*
    - Třída musí implementovat rozhraní *ISearchEngine*, jinak by jí náš systém nepřijal
    - Když už máme třídu vytvořenou, můžeme jí zkompilovat do dynamické knihovny
        - `csc.exe /target:library Modul.cs`
    - Klíčový parametr `/target:library` nám právě vytváří dynamickou knihovnu

- Jelikož nemáme přímo v naší třídě rozhraní *ISearchEngine*, musíme ho kompilátoru nějak dodat
- V naší třídě s modulem také využíváme některé třídy, které obsahuje jádro programu – i na toto musíme kompilátoru poskytnout
- Uděláme to pomocí reference na již zkompilovaný program
- Přidáním parametru `/r:` nebo `/reference:`
  - `/reference:ProgramManager.exe`
- I když nemáme možnost manipulovat se zkompilovaným kódem aplikace můžeme se na něj odkázat a pomocí toho vytvořit náš modul
- A takto vypadá konečný příkaz pro vytvoření modulu
  - `csc.exe /target:library /reference:ProgramManager.exe Modul.cs`

- **Ve vývojovém prostředí**
  - Jednodušší cestou je vytváření modulu přímo v našem vývojovém prostředí
  - Vytvoříme nový projekt
  - Ne jako aplikaci, ale jako knihovnu tříd
  - Napíšeme vyhledávací třídu

    - Musíme implementovat rozhraní *ISearchEngine*

- Důležité je uvést referenci na aplikaci
    - Díky této referenci můžeme implementovat rozhraní *ISearchEngine*, aniž bychom ho museli psát do naší třídy s modulem
    - Můžeme využívat i další třídy a statické metody z aplikace
- Po zkompilování získáme *DLL* knihovnu, kterou můžeme využít v aplikaci

4.4.6 Jak načíst modul

Ještě se by bylo dobré vysvětlit, jak takový modul načíst aplikací.

Načítání provedeme pomocí třídy *Assembly* umístěné ve speciálním namespace `System.Reflection`. Třída nemá vlastní konstruktor, proto její instanci voláme statickou metodou **`LoadFile()`**. Jako parametr je předána cesta k souboru s knihovnou.

```
Assembly asm = Assembly.LoadFile(path);
```

Dále se pokusíme vytvořit instanci *ISearchEngine* z načtené třídy implementující rozhraní.

```
ISearchEngine searchEngine =
    (ISearchEngine)
asm.CreateInstance(asm.GetTypes()[0].FullName);
```

4.5 Ostatní třídy

Seznam použitých tříd, o kterých není na škodu se zmínit a stručně popsat, jak se používají a jak jsem je použil v ukázkové aplikaci.

4.5.1 WebClient

Všechny dotazy na vzdálené servery realizuji prostřednictvím třídy *WebClient*. Tato vysokoúrovňová varianta *HttpWebRequest* usnadňuje práci pro obyčejné dotazy. V našem případě zcela vyhovující.

Tato třída má několik důležitých možností, voleb a metod, které v programu používám. Některé z nich zde stručně popíši.

- Headers
    - Vlastnost, kterou můžeme nastavit identifikaci programu na internetu. Tyto vlastnosti jsou uloženy v kolekci a prvky jsou typu `HttpRequestHeader`. Důležitými položkami jsou:
        - `UserAgent`
            - Určuje přímou identifikaci poskytovanou serveru. Např. řetězcem:

              `Mozilla/5.0 (Windows;U;Windows NT 5.1;cs-CZ)`
              zamaskujme náš program jako internetový prohlížeč ve Windows XP

- `AcceptLanguage`
  - Určuje přirozený jazyk pro odpověď. Pro náš program jsem použil řetězec:

    `cs,cs-CZ;q=0.9,en;q=0.8`
- `AcceptCharset`
  - Určuje znakovou sadu pro odpověď. V našem případě jsem zvolil `"utf-8"`

- **`DownloadString()`**
  - Metoda, která vrátí odpověď serveru jako řetězec. Jediný parametr je adresa serveru. Je možné využít asynchronního přenosu metodou **`DownloadStringAsync()`**
- **`DownloadFile()`**
  - Získá odpověď serveru a uloží ji do souboru. Metodě jsou předávány dva parametry: adresa serveru a cesta k souboru. Opět je možné využít neblokující metodou **`DownloadFileAsync()`**.

Asynchronní metody spustí požadovanou operaci v samostatném vlákně, a tím neblokují hlavní vlákno aplikace.

Tato třída má ještě mnoho dalšího, co lze využít. Zde jsem popsal jen základní metody, které jsou využity v ukázkové aplikaci.

4.5.2 BackgroundWorker

Jedná se o usnadnění práce s „pracovním“ vláknem. Tedy vláknem, které běží na pozadí aplikace a neblokuje aplikaci samotnou. Toho se využívá v místech, kde probíhá nějaká časově náročnější operace. V naší aplikaci je to využito několikrát: načítání nainstalovaných programů a hlavně při samotném volání vyhledávací metody pluginu.

BacgroundWorker má několik událostí, které zachycují nejdůležitější okamžiky pracovního procesu:

- `DoWork`
  - Hlavní událost. Obsahuje kód, který se má povést během existence pracovního vlákna. Samotný proces začne až po zavolání metody **`RunWorkerAsync()`**.
- `RunWorkerCompleted`
  - Událost vyvolaná po skončení zadaného kódu v `DoWork`
- `ProgressChanged`
  - Nastane pokud je někde v pracovním vlákně zavolána metoda **`ReportProgress()`**.

Jedinou „povinnou“ metodou je **`RunWorkerAsync()`**, která spustí vlákno a začne vykonávat kód odchycený v události `DoWork`.

Dále obsahuje metody, které se mohou, ale nemusí využít při práci s vláknem.

- `ReportProgress()`
  - Zavoláním této metody vyvoláme událost `ProgressChanged`. Metodě předáváme jeden nebo dva parametry.
    - **Číslo**
      - Procentuální vyjádření o průběhu
    - **Objekt**
      - Nepovinný parametr, kterým můžeme předat jakýkoliv objekt
  - Aby bylo možno využívat hlášení o stavu, musí být nastavena vlastnost `WorkerReportsProgress` na `true`.
- `CancelAsync()`
  - Ačkoli by se mohlo zdát, že tato metoda má za následek okamžité přerušení pracovního vlákna, není to tak. Touto metodou pouze změníme vlastnost `CancellationPending` na `true`. Je to jen požadavek na přerušení. Samotné přerušení musí obstarat vlákno samo. Aby bylo možno toho využít, musí být vlastnost `WorkerSupportsCancellation` na `true`.

4.5.3 LinkInfo

Uchovává pouze vlastnosti hledané nebo nalezené aplikace.

- `string File` - Jméno souboru aplikace
- `string Link` - Přímí internetový odkaz na soubor
- `string Name` - Jméno aplikace
- `string Version` - Verze aplikace

5 Závěr

Regulární výrazy slouží k prohledávání, nahrazování a kontrole textu. Jde o mocný nástroj nejen programátora, ale i uživatele. Lze je použít v mnoha situacích, kdy selhávají obyčejné techniky, ale i v situacích, kdy je práce s nimi přehlednější a jednodušší. Je na každém pro jakou techniku se v dané situaci rozhodne, ale je dobré vědět, že tady ta možnost je a co vše je s ní možné.

Reference

[1] Regulární výrazy [online]. 2005-2008 [cit. 2008-12-20]. Dostupný z WWW: <http://www.regularnivyrazy.info/>.

[2] Regular-expressions.info [online]. 2003-2009 [cit. 2008-12-20]. Dostupný z WWW: <http://www.regular-expressions.info/>.

[3] Root.cz : Seriály o reg. výrazech [online]. 1998 [cit. 2008-12-20]. Dostupný z WWW: <http://www.root.cz/>.

[4] Programujte.com [online]. 2007 [cit. 2008-07-20]. Dostupný z WWW: <http://www.programujte.com/>.