Proceedings of the IASTED International Conference
Software Engineering and Applications (SEA 2010)
November 8 - 10, 2010 Marina Del Rey, USA

THE POWER OF REGULAR EXPRESSIONS IN THE SOFTWARE DEVELOPMENT PROCESS

Andreas Schmidt
Department of Computer Science and Business Information Systems
University of Applied Sciences
Karlsruhe, Germany
email: andreas.schmidt@hs-karlsruhe.de

Daniel Kimmig
Institute for Applied Computer Science
Karlsruhe Institute of Technology (KIT)
Karlsruhe, Germany
email: daniel.kimmig@kit.edu

ABSTRACT
The tutorial will present various lightweight generators to support software development. The input of these generators is either the source code or special textual models describing the application to be generated. The generators are set up by means of regular expressions that extract the relevant information from the input files.

KEY WORDS
Software generator, regular expressions, model transformation.

1 Introduction

Principle functioning of a generator is obvious from Figure 1. The input of the generator consists in the model of the application to be generated. It may exist in the form of a complex UML-based model description (e.g. XMI format) or, in simple cases, of the source code or a combination of source code and simple model description. The other major input are transformation rules that describe what the source code to be generated is to look like. Hence, the model description is converted into the source code by transformation rules.

The tutorial will be aimed at familiarizing the participants with the power of regular expressions and at demonstrating their suitability for software development. After the tutorial, the participants will be able to recognize the potential of lightweight generators in software development and also will be able to develop their own generators based on regular expressions.

2 Lightweight Generators

The generators presented by the tutorial will be simple generators exclusively that often do not comprise more than a dozen code lines. Once their functioning is understood, a minimum expenditure is needed to create own generators for a concrete task within a few minutes only. Typical lightweight generators are code munger, the inline code expander or the mixed code generator. These generators often rest upon regular expressions. The input of these generators could be a model, defined in a formal way, but could also be existing program-code, probably annotated.

3 What Can Be Generated ?

It is the objective to partly or completely generate the source code for an application to be implemented. The degree of automation mostly is in the range from 20% to 80% of the application. In case of web-based applications, a degree of automation of about 60 - 70% can be reached frequently. Typical parts of an application that can be generated are:

- Database schemas
- Access layers for databases
- User interfaces
- Parts of the application logic
- Documentation
- Configurations (e.g. together with frameworks like Struts, Spring, ...)
- Tests
- Import/export modules
- ...

4 Regular Expressions

Regular expressions are a powerful pattern language. They allow the filtering and/or substitution of text patterns. Implementations exist for many computer languages, mostly in the form of a library, but it is also possible to integrate them directly into the syntax of the language, like in Perl, AWK or Ruby.

A regular expression describes a set of strings. It consists of a number of literal characters (like A..Z, a..z, 0-9) and some meta characters ([,], (,), {,}, |, ?, +, -, *, ∧, $, \, ., \b, ...), which have a special meaning. Additionally there are a number of character classes, e.g. for alphanumeric characters ([:alfa:][1], \w[2]) or whitespace characters

[1] POSIX-Syntax
[2] Perl-Syntax

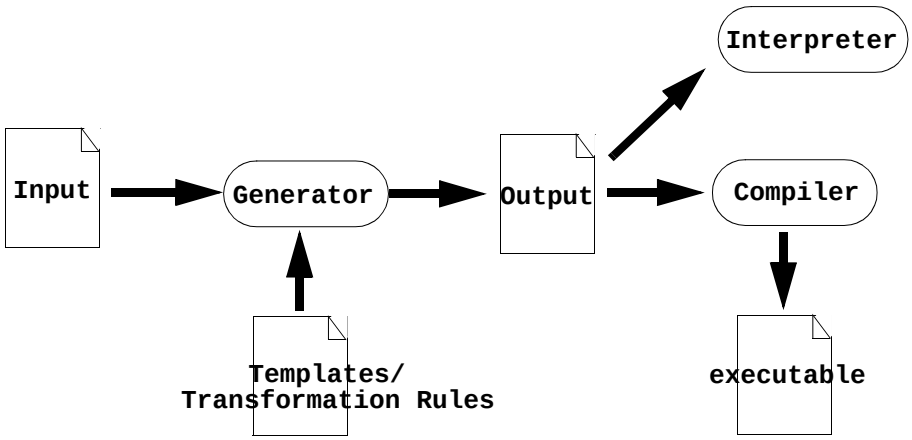

Figure 1. Principle functioning of a generator

Table 1. Regex - Metacharacters

Meta-Character	Description
.	matches everything
∧	start of pattern
$	End of pattern
|	or
\b	word boundary
(...)	grouping and backreferencing (see below)
[...]	definition of a character class

([:space:], \s). You can also define your own character classes like [AEIOUaeiou] for vowels.

Literal characters are used like in a normal string search, but the real power of regular expression lies in the use of the metacharacters. If you look for a character that has a meaning as metacharacter, you must prefix it with a \inside your regular expression. Some of the most important metacharacters and there meanings are shown in Table 1.

Example[3]: `\b(car|bike)\b.*[\.!]$` matches the string "a bike is pure fun!" but not "a motorbike sucks.".

4.1 Quantifier

With a quantifier, you can specify how often the token before the quantifier should be matched. You can limit the number of occurences from zero to infinite, with or without a concrete number of repetitions. Take a look at Table 2 for a overview of the differnet possible quantifiers.

To clarify the concept, let us look at two more examples:

[3]Find the isolated words "car" or "bike", and at the end of the string there must be a period (.) or a exclamation mark (!) as last character.

Table 2. Regex - Quantifiers

Quantifier	Description
*	zero or more (greedy[5])
+	one or more (greedy)
*?	zero or more (non greedy[6])
+?	one or more (non greedy)
?	zero or one
{3,5}	three to five
{10,}	ten or more
{,10}	at most ten
{10}	exactly ten

number[4]: `-?\d+(\.\d+)?`

IP-address: `\b(\d{1,3}\.){3}\d{1,3}\b`

4.2 Backreferences

If you put round brackets around a part of the expression, the content of this part is stored internal for later use and can be accessed by referencing it with $x or\x ($x \in (1, 2, \ldots)$). So for example, to look for the matching closing tag of the different header element in a HTML-document, you can write:

```
<([Hh][0-6])>(.*?)</$1>
```

Here, $1, matches the heading element name (h1 - h6) and in $2 you can find the content of the heading.

4.3 Perl Regular Expressions from the Command Line

Perl [1] has a special shorthand notation, which allows the execution of perl-code from the command line. And be-

[4]like -2.1456, 0.546, 42

```
 1  class Film {
 2
 3    private $title;
 4    private $year;
 5    private $director;
 6
 7    function __construct($title,
 8                         $year,
 9                        $director) {
10
11      $this->title = $title;
12      $this->year = $year;
13      $this->director = $director;
14    }
15  }
```

Figure 2. Complete class definition

cause regular expressions are valid perl code, you can write something like this in your command line:

```
$ perl -pi.bak -e "s#<.*?>##g" *.xml
```

The result is, that in all files with the extension `.xml`, the markup is removed. The original versions of the files are stored with the extension `.bak` - cool stuff !

5 Simple Generator Example

In the current section a simple inline code expander generator is presented. A number of PHP classes with a constructor each shall be generated. Instead of generating them manually, we define a simple extension for PHP, which corresponds to a short notation for classes in PHP. Instead of defining the complete class (Figure 2), we only write in compressed form:

```
<class: Film (title, year, director) >
```

Then, the source code file with the extended syntax is input into the generator that executes the transformation into a valid PHP class (Figure 2).

The question now is how the shortened syntax is mapped onto the PHP syntax: For this purpose, an adequate regular expression is formulated, which matches lines of the form `<class: ...>` in the source code, parses them, and splits them into their constituents. The code fragment in Figure 3 reads the content of the passed file (*$args[1]* (line 1) into an array, iterates over all these lines and searches for the respective lines which match the regulare expression pattern in line 2 (function *preg_match(...)*— in line 5 of the source code).

If such a line is found, the relevant parts (class name and list of instance variables) are extracted (in variable *$m*) and the transformation into the valid PHP code remains to be executed only. This is done using the function *print_class_definition(...)* (see Figure 4).

```
1  $lines = file($argv[1]);
2  $regex = '#<class:\s*(\w+)\s*\((.*)\)\s*>#';
3
4  foreach ($lines as $line) {
5    if (preg_match($regex, $line, $m))
6      print_class_definition($m[1], $m[2]);
7    else
8      print $line;
9  }
```

Figure 3. Generator core

The code generated is output with the help of elementary control flow elements like loops and conditional instructions. For illustration, the control flow elements (including the function definition) are represented on a dark background, constant code parts not. All other lines of the input file are taken over unmodified (else branch in Figure 3, line 8).

Then, the function *print_class_definition(...)* yields the string of the class definition. In spite of its minimum volume, the generator developed already possesses some typical features of full-tier generators. For instance, the extended syntax shown in Figure 2 represents a very simple model of the application to be implemented. Full-tier generators possess a far more powerful model that is often based on UML, of course. Code generation proper, as shown by the hard coded function in Figure 4, is often outsourced to a separate template system in full-tier generators. It implements the generation using certain given language elements for the control flow and allocation of variables.

6 Steps towards a general purpose generator

What is the difference between our simple generators and a general purpose generator engine ?

Our simple generator example in the previous section already has a number of feature a complete multi purpose generator consists of. In Figure 3 we have a minimal generator kernel, taken the abstract specification of the classes (`<class: Film (...)>`) as input and the function *print_class_definition(...)*, written in PHP macro style [2] could be seen as a simple template and we also have the generated code in Figure 2 as output.

In contrast to a multipurpose generator we do not have an internal representation of our model. We pick up the input and immediately generate the output, so we do not need another representation. But from time to time the things become more complex. Consider the situation when you want to generate data definition language (DDL) code a relational database. Your model input could look like this:

```
<class Person (id Number(10), \
               name String(20), \
               birthday Date) >
```

```
<?php
  function print_class_definition($class_name, $att_list_as_str)
{
      $att_list = preg_split("#\s*,\s*#", $att_list_as_str);
?>

   class <?php echo $class_name ?> {
   <?php foreach ($att_list as $a) { ?>
      private $<?php echo $a ?>;
   <?php } ?>

     function __construct($<?php echo join(',$',$att_list) ?>) {
<?php foreach ($att_list as $a) { ?>
   $this-><?php echo $a ?> = $<?php echo $a ?>;
<?php } ?>
      }
    }

<?php } ?>
```

Figure 4. Function to generate the source code

```
<class Film (id Number(10), \
             title String(20), \
             year Number(4), \
             director Person) >
```

Looks not so difficult ... but wait. When parsing the last column of class *Film*, we find a reference to class *Person*, which represents a $1:n$ relationship between *person* and *Film*. And in the relational world, a $1:n$ relation is modeled as a foreign key. But to do so, we need to know the datatype of the primary key in the referenced table. As a result, it is necesary to go back and read the input once more, this time catching the information about the datatype of the primary key field. The problem is, that we need the information not only in sequential order, but we need parts of the input more than once or in a different order.

But there is a more elegant solution. If we define an internal metamodel, which has the appropriate structure and semantic to hold all the information, we first read in our model and when looking for the datatype of another class we only need to call an appropriate function on the metamodel, i.e. `getPrimaryKey($class)`.

The internal metamodel is also the orginator for further improvements. First of all, it is the right place to do some verification on the input.

Additionally you can do different sorts of model-transformation, e.g. adding some administrative attributes to every class (*createdAt*, *createdBy*, ...) within the same metamodel or between different metamodels. In this case, you will probably have another metamodel defined, which is semantically closer to our target model (the GUI or a relational database). In this case you have an additional transformation step, but it could be easier to do two little transformation steps instead of one big transformation step. The MDA [3]approach from the OMG uses this concept by distinguishing between a platform independant model (PIM) and a platform specific model (PSM).

7 Conclusions

Roughly speaking, software development may be divided into an interesting part that is mostly given by the application logics and a more mechanical, and hence boring, part, including the infrastructure code necessary for the platform, major parts of interface development, connection of the database, error handling, etc.

As shown in the present tutorial, with little effort, small yet efficient helpers can be developed easily. And besides code generation, they can also be used for generating documentation or plausibility checks, cross checks and code metrics. And in contrast to the full fledged generators they are handy in use and could be developed or adapted to your problems in a really short period of time.

The generation of source codes plays an important role in my work at the Institute for Applied Computer Science of the Karlsruhe Institute for Technology (KIT). Major advantages of these technologies are an increased consistency of the source code, a higher productivity and quality, and an increased abstraction level in development, as abstraction is possible from many code details of the target platform [4].

In case you are interested in this topic, it is referred to an exciting book by Jack Herrington [4]. In his book Jack develops a number of generators for all kind of situations. The generators are written in Ruby, but may also be understood by ”non-Rubians”. A very good book on regular expressions has been published by Jeff Friedl [5]. A survey of existing generators can be found under [6].

References

[1] L. Wall, T. Christiansen, and J. Orwant, *Programing Perl*, 3rd ed. Sebastopol: O'Reilly, 2000.

[2] R. Lerdorf, K. Tatroe, and P. MacIntyre, *Programming PHP*. Sebastopol: O'Reilly, 2006.

[3] D. S. Frankel, *Model Driven Architecture. Applying MDA to Enterprise Computing*. New York: John Wiley & Sons, 2003.

[4] J. Herrington, *Code Generation in Action*. Greenwich CT: Manning, 2003.

[5] J. E. Friedl, *Mastering Regular Expressions*, 2nd ed. Sebastopol: O'Reilly, 2002.

[6] "Wikipedia - model driven engineering," http://en.wikipedia.org/wiki/Model-driven_engineering.