Literate Programming and the "Spaniel" Method

Nick Hatzigeorgiu* Apostolos Syropoulos†

Abstract

Software has to be well structured and well documented in order to be reusable and maintainable. Literate programs have both properties. Unfortunately, even today many programmers write program without proper documentation. We propose a technique by which one can relatively easily transform a well structured program into a literate one. We exemplify the technique by means of an example and we conclude with the presentation of some related work and ideas.

1 Introduction

A decade ago the second author was programming the compiler of a simple Pascal-like language, as part of a graduate student project. In the final stages of the program development a problem surfaced concerning type checking and after weeks of debugging it seemed to be untractable. So he decided to contact an expert, namely Prof. W. W. Waite, in order to get some help. Prof. Waite responded to him and he described a *weird* debugging technique:

> *As is often the case with baffling errors it is really quite simple. We tend to fixate on incorrect assumptions, and overlook the obvious, surprisingly frequently. I have found that one way to break through such barriers is to use the "Spaniel" method: Carefully explain the program to your dog. Since the dog knows nothing of programming, you must justify every statement you make. In the process you will often discover the mistake. (I know it sounds weird, but it really does work!)* [Wai88]

Prof. Donald E. Knuth invented literate programming [Knu92] while preparing version 2 of his TEX program. Literate programming is a program development methodology. A literate program consists of intermixed chunks of code and text. The text serves as an explanation of the programming code. In its initial from, a preprocessor, `weave`, produces a TEX source file which in turn can be typeseted by TEX. Another preprocessor, `tangle`, produces an unformatted Pascal source file. A literate program is called a web, and hence the names of the preprocessors. The main drawback of these two programs is that they are language oriented, i.e., one can use them only if he/she programs in Pascal and uses TEX as his/her typesetting engine. This problem initiated various efforts to create language independent tools, the most popular being `noweb` by Normal Ramsey [Ram94] and `nuweb` by Preston Briggs [Bri93]. In particular, `noweb`, is independent of the typesetting engine since it is capable to produce plain TEX, LATEX, roff, or HTML output.

Although the benefits of developing programs by employing the literate programming methodology have been recognized by many authors, see [CS96, Par96] for a recent account, unfortunately

*Institute of Language and Speech Processing, Xanthi Branch, 8, Vas. Sofias Str., GR-671 00 Xanthi, GREECE, e-mail: `nikos@xanthi.ilsp.gr`.

†Department of Civil Engineering, Democritus University of Thrace, GR-671 00 Xanthi, GREECE, e-mail: `apostolo@obelix.ee.duth.gr`.

even today many programmers write programs that don't have a single line of comments. We believe that if someone employs the "Spaniel" method it is possible to produce literate programs from "illiterate" ones. In the next section we propose a simple method by which we can produce literate programs from "raw" source code; then we demonstrate the method's usability by means of a relatively simple example. The last section presents some related work and ideas on the subject.

2 The Methodology

In order to produce a literate program from "raw" source code, one should know how the program works and what it computes. This means that we should know exactly the structure of the program, i.e., the various sections that it consists of. For example, a compiler must read the source code, construct the syntax tree, perform type checking and, finally, produce some sort of code. So, our first step is to identify the various sections of our code. Each section is usually composed of various subsections, e.g., when a compiler reads a source file it tokenizes the input lines so that the parser can construct the syntax tree and perform syntax analysis. Next, each subsection is transformed into a code chunk, in the sense of literate programming. A reasonable question is: When do we stop? Obviously, when the code is self-explanatory.

The careful reader may realize that our methodology resembles the structured programming methodology (top-down program development). Once we have divided our code into reasonably-sized code chunks, we must write down the text that will explain the code. Certainly, this text should not be an English version of the operational semantics of the code, e.g., the explanation of the assignment `s=pi*R*R` should not be of the form *Here we assign to variable s the value pi*R*R*, as this contributes nothing to the reader's understanding of the program. The text should explain the meaning of the command(s) in the program's context, i.e, what this particular piece of code achieves. But, what does this mean?

At this point the "Spaniel" method comes into use. We assume that we are the authors of the code and that we explain our program to a non-programmer who, however, is quite aware of the nature of the problem the program solves. This means that we will write a short introduction that will explain the problem and then we will present the way our program tackles the problem, describing the functionality of each program chunk. In this light an assignment does not simply assign a value to some variable, but stores an important piece of information to the computers memory. So, we must explain what this information is about. For example the above assignment *Assigns to variable s the area inside a circle with radius equal to R.* Furthermore, how should someone treat other constructs, e.g., loops, conditional statements, etc? Each programming construct is a tool and it must be viewed as such. This means that we are not interested in describing the way it operates —after all we assume that the reader is familiar with our programming notation– but what is exactly its contribution in achieving our programming goal.

3 An Example

To demonstrate the above methodology we present a simple Perl script which is used to exemplify our methodology. Our simple program is one that computes the day of week for dates after December 31st, 1752. Here is the "raw" code:

```
#!/usr/bin/perl
use integer;
$argc = @ARGV;
die "Usage: day_of_week dd mm yy\n" if $argc != 3;
```

```
$day = $ARGV[0];
$month = $ARGV[1];
$year = $ARGV[2];
die "Year is not Gregorian calendar year : $year < 1753 \n"
                                          if $year < 1753 ;
@months = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
die "Illegal month: $month\n" if $month > 12 || $month < 1;
$months[1] = 29  if ($year % 100 == 0 && $year % 400 == 0)
                    || $year % 4 == 0;
die "Illegal day $day for month $month\n"
                               if $months[$month-1] < $day ||
                                  $day < 0;
@wday = ( "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" );
$y = $year;
$m = ($month + 10) % 12;
$y-- if ($m > 10);
$c = int ( $y / 100 );
$yy = $year % 100;
$z = ( int ( (26*$m - 2)/10) + $day + $yy + int($yy/4) +
       int ($c/4) - 2*$c ) % 7;
print "$wday[$z]\n";
```

We now apply the "Spaniel" method in order to derive a literate program from the above code. A short introduction should explain what the program is about, i.e., it computes the day of week for a given date. We then identify the various sections of the program. The program gets input, validates the input and computes the day of the week. Next, we inform the reader that the program gets its input from the command line in the form of three integers. Then, we explain what kind of validation we perform (remember that we know how the program operates), i.e., that the year must be after 1752, the month must be in the range 1...12, etc. And, finally, we describe what kind of computational algorithm our the program uses. The resulting literate program, writen in noweb's style of literate programming, appears on the next page.

The most interesting side-effect is that someone may discover bugs in his/her program he/she never thought of before, as we surprisingly did! (Compare the above code with the literate program.) This is the power of the "Spaniel" method: while we try to explain to a non-expert the inner workings of our code we sometimes see that there might be pitfalls and omissions in what we considered to be a perfectly working program. Thus, this method helps us not only to get out of deadlock situations but also to improve our code.

4 Conclusions

This isn't the first time someone tries to produce some sort of explanation of a computer program. For example, Gabriel [P.88] describes a system called *Yh* which was capable of producing explanations of computer programs generated by another specific program (which generates programs from user specifications). This work was the starting point for the Master's Thesis [Syr93] of the second author who tried to generalize Gabriel's methods.

There has also been an effort (and a lot of related discussions in the comp.programming.literate newsgroup) to produce literate programs from old code, usually by students of computer science. Every effort of this kind is valuable both because of its educational value and the possibilities of finding new improvements to old code. We believe that the "Spaniel" method can be a useful tool in those efforts.

This is simple program that reads a date from the command line and computes the day of the week that corresponds to it. The program uses the module `integer` in order to insure proper integer arithmetic.

1a ⟨* 1a⟩≡
```
#!/usr/bin/perl
use integer;

⟨Check command line arguments 1b⟩
⟨Perform date validation 2a⟩
⟨Compute day of week 2b⟩
```

The program accepts only three command line arguments (integers), otherwise it aborts with an appropriate error message. Next, we get the date information, e.g., the day, the month and the year in this order.

1b ⟨Check command line arguments 1b⟩≡
```
$argc = @ARGV;
die "Usage: day_of_week dd mm yy\n" if $argc != 3;
$day = $ARGV[0];
$month = $ARGV[1];
$year = $ARGV[2];
⟨Check whether day, month and year are indeed numbers 1c⟩
```

As it is the case many users try to fool programs; in our case a user may enter words or sequences of random characters as input. We employ Perl's regular expressions to validate our input in this respect.

1c ⟨Check whether day, month and year are indeed numbers 1c⟩≡
```
die "Year is not a number: $year\n" if $year !~ /\d+/;
die "Month is not a number: $month\n" if $month !~ /\d+/;
die "Day is not a number: $day\n" if $day !~ /\d+/;
```


We validate the given date. First we check that the year is a valid year of the Gregorian calendar, i.e, it is after 1752. Array `@months` holds the days of each month in a non-leap year. A legal month is in the range 1-12. A legal day is in the range of 1 to the days of the month. However, we must make sure that leap years are taken into account. When the year is a leap one, i.e., it is a century year divisible by 400 or a non-century year divisible by 4, then February has 29 days. Consequently, a valid day number is positive and less or equal to the number of days of the corresponding month.

2a ⟨Perform date validation 2a⟩≡
```
die "Year is not Gregorian calendar year : $year < 1753 \n"
                                          if $year < 1753 ;
@months = (31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31);
die "Illegal month: $month\n" if $month > 12 || $month < 1;
$months[1] = 29  if ($year % 100 == 0 && $year % 400 == 0)
                    || $year % 4 == 0;
die "Illegal day $day for month $month\n"
                                 if $months[$month-1] < $day ||
                                    $day < 0;
```

Array `@wday` holds the names of the week days. In order to compute the day of the week we use Zeller's Rule, invented by a certain Rev. Zeller. Since, the rule assumes that February is the last month of the proceeding year we must adapt the month's value (variable `$m`). Variables `$c` and `$yy` hold the first two and the last two digits of the year respectively. Next, we compute the day's number and we print the day's name.

2b ⟨Compute day of week 2b⟩≡
```
@wday = ( "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" );
$y = $year;
$m = ($month + 10) % 12;
$y-- if ($m > 10);
$c = int ( $y / 100 );
$yy = $year % 100;

$z = ( int ( (26*$m - 2)/10) + $day + $yy + int($yy/4) +
       int ($c/4) - 2*$c ) % 7;

print "$wday[$z]\n";
```

References

[Bri93] Preston Briggs. Nuweb, A simple literate programming tool. `http://cs.rice.edu/public/preston`, Rice University, Houston, TX, USA, 1993.

[CS96] Bart Childs and Johannes Sametinger. Literate programming and documentation reuse. In Murali Sitaraman, editor, *Fourth International Conference on Software Reuse: proceedings, April 23–26, 1996, Orlando, Florida, USA*, pages 205–214. IEEE Computer Society Press, 1996.

[Knu92] Donald E. Knuth. *Literate Programming.* Number 27 in CLSI Lecture Notes. Center for the Study of Language and Information, Leland Stanford Junior University, 1992.

[P.88] Gabriel R. P. Deliberate writing. In D. D. McDonald and L. Bole, editors, *Natural Language Generation Systems*, pages 1–46. Springer-Verlag, New York, 1988.

[Par96] Chris Parker. Literate programming using SGML and modern hypertext technology. Thesis (M.Sc.), Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA, USA, 1996.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, September 1994.

[Syr93] Apostolos Syropoulos. Explaining Computer Programs: An Elementary Approach. Thesis (M.Sc.), Department of Computer Science, University of Gothenburg, Gothenburg, Sweden, 1993. Available as: `ftp://obelix.ee.duth.gr/pub/docs/explain.ps.gz`.

[Wai88] W. W. Waite. Personal communication, August, 1988.