Tech Notes, general Series

The fine Art of Commenting

by **Bernhard Spuida, bernhard@icsharpcode.net**
Senior Word Wrangler

Table of Contents

1. Introduction

“Commenting is a royal pain in the posterior” - “Comments are for weenies” - “I can understand my code quite well, thank you very much” - “Good code speaks for itself” - “No time for that, got to get that code out of the door”. Admit it, you have said some thing along these lines at least once during your coding career. Maybe you even now still are in this kind of frame of mind.

Negative attitudes towards commenting may have several reasons:

- Programmer's hubris
- Lazyness
- No time left for documentation due to deadline constraints

None of these is a good reason for not commenting source code properly. We will look at these arguments, discuss them and take a look at good commenting practice and its benefits.

As SharpDevelop is intended to be an IDE for all languages supported by the .NET platform – and others, if support is available – this text will not discuss language specific commenting issues. Knowledge of all languages referred to is not necessary for the understanding of this paper.

2. The Case against Commenting

We now take a look at those negative opinions offered in making a case against commenting code.

2.1 Programmer's Hubris

A good programmer is always a programmer with something of a well developed ego. Nothing is impossible, everything is easy to understand. So much for theory.

In practice, reality checks are in order from time to time. Do you understand all your code after not looking at it for, say, a year? Is legacy code left to you to maintain always obvious at first look, or even after a few weeks of scrutiny? Truth is, most of the time it will take a lot of time and effort to understand undocumented code, even if the code has not been obfuscated intentionally.

As an example, let us consider the solitaire encryption algorithm of Bruce Schneier in its 'concise' version as published in Neal Stephensons novel 'Cryptonomicon'. It is implemented in Perl without 'spaghetti code', yet due to its terse coding style, it is almost incomprehensible, should you attempt to figure out what it does:

Exhibit A:

```
#!/usr/bin/perl -s
## Ian Goldberg <ian@cypherpunks.ca>, 19980817
$f=$d?-1:1;4D=pack(C',33..86);$p=shift;
$p=~y/a-z/A-Z/;$U='$D=~s/.*)U$/U$1/;
$D=~s/U(.)/$1U/;';($V=$U)=~s/U/V/g;
$p=~s/[A_Z]/$k=ord($&)-64,&e/eg;$k=0;
while(<>){y/a-z/A-Z/;y/A-Z//dc;$o.=$_}$o='X'
while length($o)%5&&!$d;
$o=~s/./chr(($f*&e+ord($&)-13)%26+65/eg;
$o=~s/X*$//if $d;$o=~~s/.{5}/$& /g;
print”$o/n”;sub v{$v=ord(substr($D,$_[0]))-32;
$v>53?53:$v}
sub w{$D=~s/(.{$_[0]})(.*)(.)/$2$1$3/}
sub e{eval”$U$V$V”;$D=~s/(.*)([UV].*[UV])(.*]/$3$2$1/;
&w(&v(53));$k?(&w($k)):($c=&v(&v(0)),$c>52?&e:$c)}
```

Imagine you confront code written like this, with your employer expecting you to maintain it... Now facing such a daunting task, the avenues that might lead to understanding this terse piece of code are:

1) Stare at the code for a long time and have a headache[1].
2) Single step through the program a few times with varying input, to see what it does.
3) Look at the plain text explanation of the algorithm, if available[2].

The best alternative however is to have properly commented code. The very same algorithm including comments (and a few lines to make it a bit more user and maintainerfriendly) is perfectly easy to understand and adapt to individual requirements:

Exhibit B:

```
#!/usr/bin/perl -s

## Solitaire cryptosystem: verbose version
## Ian Goldberg <ian@cypherpunks.ca>, 19980817

## Make sure we have at least the key phrase argument
die "Usage: $0 [-d] 'key phrase' [infile ...]\n" unless $#ARGV >= 0;

## Set the multiplication factor to -1 if "-d" was specified as an option
## (decrypt), or to 1 if not.  This factor will be multiplied by the
output
## of the keystream generator and added to the input (this has the effect
## of doing addition for encryption, and subtraction for decryption).
$f = $d ? -1 : 1;

## Set up the deck in sorted order.  chr(33) == '!' represents A of
clubs,
## chr(34) == '"' represents 2 of clubs, and so on in order until
## chr(84) == 'T' represents K of spades.  chr(85) == 'U' is joker A and
## chr(86) == 'V' is joker B.
$D = pack('C*',33..86);

## Load the key phrase, and turn it all into uppercase
$p = shift; $p =~ y/a-z/A-Z/;

## For each letter in the key phrase, run the key setup routine (which
## is the same as the keystream routine, except that $k is set to the
## value of each successive letter in the key phrase).
$p =~ s/[A-Z]/$k=ord($&)-64,&e/eg;

## Stop setting up the key and switch to encrypting/decrypting mode.
$k = 0;

## Collect all of the alphabetic characters (in uppercase) from the input
## files (or stdin if none specified) into the variable $o
while(<>) {
    ## Change all lowercase to uppercase
    y/a-z/A-Z/;
    ## Remove any non-letters
    y/A-Z//dc;
    ## Append the input to $o
    $o .= $_;
}

## If we're encrypting, append X to the input until it's a multiple of 5
chars
```

1 Should you claim you can understand it from looking at it in this form, you are a true genius.
2 In this case, we are fortunate. The explanation of the 'solitaire' algorithm is given by Bruce Schneier on his web site: http://www.counterpane.com/solitaire.html

```
if (!$d) {
    $o.='X' while length($o)%5;
}

## This next line does the crypto:
##   For each character in the input ($&), which is between 'A' and 'Z',
##   find its ASCII value (ord($&)), which is in the range 65..90,
##   subtract 13 (ord($&)-13), to get the range 52..77,
##   add (or subtract if decrypting) the next keystream byte (the output
of
##         the function &e) and take the result mod 26 ((ord($&)-
13+$f*&e)%26),
##     to get the range 0..25,
##    add 65 to get back the range 65..90, and determine the character
with
##     that ASCII value (chr((ord($&)-13+$f*&e)%26+65)), which is between
##     'A' and 'Z'.  Replace the original character with this new one.
$o =~ s/./chr((ord($&)-13+$f*&e)%26+65)/eg;

##  If we're decrypting, remove trailing X's from the newly found
plaintext
$o =~ s/X*$// if $d;

## Put a space after each group of 5 characters and print the result
$o =~ s/.{5}/$& /g;
print "$o\n";

## The main program ends here.  The following are subroutines.

## The following subroutine gives the value of the nth card in the deck.
## n is passed in as an argument to this routine ($_[0]).  The A of clubs
## has value 1, ..., the K of spades has value 52, both jokers have value
53.
## The top card is the 0th card, the bottom card is the 53rd card.
sub v {
    ## The value of most cards is just the ASCII value minus 32.
     ## substr($D,$_[0]) is a string beginning with the nth card in the
deck
    $v=ord(substr($D,$_[0]))-32;
    ## Special case: both jokers (53 and 54, normally) have value 53,
      ## so return 53 if the value is greater than 53, and the value
otherwise.
    $v>53?53:$v;
}

## The following subroutine generates the next value in the keystream.
sub e {
     ## If the U (joker A) is at the bottom of the deck, move it to the
top
    $D =~ s/(.*)U$/U$1/;
    ## Swap the U (joker A) with the card below it
    $D =~ s/U(.)/$1U/;

    ## Do the same as above, but with the V (joker B), and do it twice.
    $D =~ s/(.*)V$/V$1/; $D =~ s/V(.)/$1V/;
    $D =~ s/(.*)V$/V$1/; $D =~ s/V(.)/$1V/;

    ## Do the triple cut: swap the pieces before the first joker, and
    ## after the second joker.
    $D =~ s/(.*)([UV].*[UV])(.*)/$3$2$1/;

    ## Do the count cut: find the value of the bottom card in the deck
    $c=&v(53);
    ## Switch that many cards from the top of the deck with all but
    ## the last card.
    $D =~ s/(.{$c})(.*)(.)/$2$1$3/;

    ## If we're doing key setup, do another count cut here, with the
    ## count value being the letter value of the key character (A=1, B=2,
    ## etc.; this value will already have been stored in $k).  After the
    ## second count cut, return, so that we don't happen to do the loop
```

```
## at the bottom.
if ($k) {
        $D =~ s/(.{$k})(.*)(.)/$2$1$3/;
        return;
}

## Find the value of the nth card in the deck, where n is the value
## of the top card (be careful about off-by-one errors here)
$c=&v(&v(0));

## If this wasn't a joker, return its value.  If it was a joker,
## just start again at the top of this subroutine.
$c>52?&e:$c;
```

It admittedly is longer than the terse version of the code, but it is more readable by far. Even if you are not familiar with the Perl programming language, you will by now have some idea of what the program does. This is what commenting is about.

We also can see from this example that there are various functions comments may have in the scope of a program. These will be discussed in section 3.

Of course, comments may be stripped out of the final production code if there are severe restraints on storage space. On modern machines, this usually is no longer necessary though – perhaps when doing client side scripting or writing code for fiendishly small embedded thingies. And even if you do strip out comments, always keep a commented version of the sources at hand for future work.

2.2 Lazyness

That is no excuse at all. The idea behind it is a fallacy in the best of cases. **All time saved by not commenting during the coding process is made up *more than twice at least* by inserting comments and providing documentation after the coding process is finished.** In a real production environment, documentation of the code written is a requirement. As there nowadays are a number of systems available for the automatic generation of documentation out of comments, you will actually save yourself a lot of hours of boring documentation work by simply inserting appropriately formatted comments as you code. We will look at such systems and 'literate programming' in section 4.

2.3 The Deadline Problem

So, you're not writing comments because of problems with the project deadline? Then something is very, very wrong with the whole approach taken for managing that project. Omitting commenting will not at all help. Sorry. The few minutes saved this way every day will only make the crunch harder as the deadline comes closer. Then you will need to get things right fast. And that cannot be done by having undocumented code[3] that is hard to figure out when bug hunting or refactoring yet another time is at hand.

Actually we save time by commenting properly. Bad project management is enough of a problem to cope with. We do not need unreadable, undocumented source code as well.

3 Commenting done right

There are several purposes that comments may serve:

- Documentary comments
- Functional comments

3 As experience with tight deadlines shows, usually also code of low quality results.

- Explanatory comments

3.1 Documentary Comments

This class of comment serves to document the history and development of the source file or project in question. Into this category belong the following comments that should be included in every source file:

1. Filename
2. Version number/build number
3. Creation date
4. Last modification date
5. Author's name
6. Copyright notice
7. Purpose of the program
8. Change history
9. Dependencies
10. Special hardware requirements (e.g. A/D converters)

This type of comment obviously is intended to document the software itself, its purpose and its history. The intent of these is to make life easier for maintenace of code that is expected to have a life expectancy going beyond an 'ad hoc' script's. Any code planned to be in use for more than a few weeks should absolutely contain these comments.

Exhibit C:

```
{
PCMBOAT5.PAS************************************************************
**

 File:                   PCMBOAT5.PAS

 Author:                 B. Spuida
 Date:     1.5.1999
 Revision: 1.1           PCM-DAS08 and -16S/12 are supported.
                         Sorting routine inserted.
                         Set-files are read in and card as well as
                         amplification factor are parsed.

           1.1.1         Standard deviation is calculated.

           1.1.2         Median is output. Modal value is output.

           1.1.4         Sign in Set-file is evaluated.
                         Individual values are no longer output.
                         (For tests with raw data use PCMRAW.EXE)

 To do:                  outliers routine to be revised.
                         Statistics routines need reworking.
                         Existing Datafile is backed up.

 Purpose:                Used for measurement of profiles using the
                         Water-SP-probes using the amplifier and
```

```
                        the PCM-DAS08-card, values are acquired
                        with n = 3000. Measurements are taken in 1
                        second intervals. The values are sorted using
                        Quicksort and are stacked "raw" as well as after
                        dismissing the highest and lowest 200 values as
                        'outliers'.

 Requirements:          The Card must have an A/D-converter.
                        Amplifier and probes must be connected.
                        Analog Signal must be present.
                        CB.CFG must be in the directory specified by the
                        env-Variable "CBDIREC" or in present directory.

*************************************************************************
*
}
```

Exhibit C is the heading comment block of one program from a suite of data acquisition tools written in TurboPascal. With the exception of the 'To do' block, all comments are declaratory. Please also note the layout of the comment blocks. We will discuss comment layout in section 3.4. Also notice the absence of a copyright notice, as this program is intended for company internal use only.

These comments may be inserted manually, as was the case for this program, but as this manual insertion and updating process is error prone, it is recommended to use the placeholder strings made available by versioning tools. Using versioning tools is 'Best Practice' for mid to large scale projects, so some form of placeholder for the most vital points (1 – 5,8) usually is at your disposition.

A tip: whenever you rename a file, immediately change the appropriate comment. Otherwise, you are certain to forget it and automated documentation or building systems may run out of kilter.

3.2 Functional Comments

Functional comments serve one purpose: adding functionality to the *development process*. They do not describe the code or its development history but serve single processes in the development cycle. The most obvious functional comment type is a **'to do'** comment. There are others, though:

- Bug description
- Notes to co-developers
- Performance/feature improvement possibilities

Such comments should be used sparingly, even though they are important. Together with the documentary comments, they will provide a history documenting the design of the program. They always should be in the same format, so they may be easily extracted or searched for. It is good practice to have uniform 'TODO:' comments etc., so that portions of code that need reworking or need improvement are found reliably.

Repeat:

'I will always insert these comments where necessary, immediately when I find they are necessary, and always use the standard form agreed on by the entire development team.'

Into this category, we might also count comments documenting bug fixes. These should give who

fixed what bug[4] when and how. Of course, we might also consider these comments as documentary comments. This is however, merely a 'philosophical' question to consider. Let us just always insert these comments.

3.3 Explanatory Comments

This type of comment is quite obvious in its function. Well written code will contain a lot of these, even though explanatory comments are obviously not necessary for each single line of code. Items that should have explanatory comments are:

- Startup code
- Exit code
- Sub routines and functions
- Long or complicated loops
- Weird logic
- Regular expressions

Commenting startup code is good practice, as this will give an idea of what arguments are expected and how they are handled, how the program is initialised, what the default values and settings options for the configuration variables are, what `#defines` do etc.

The same goes for the exit code, both for normal and abnormal exit situations. Return values, error codes etc. should be properly explained.

Each subroutine or function should be commented, stating the purpose of it. Also the arguments passed and returned should be explained, giving format, limits on values expected etc.

The limits on values are an especially important point to be documented in comments. Quite a number of bugs and even completely broken applications result from not stating clearly whether there are limits on value ranges for input and expected return values. What will happen if we try to stuff a 1024 char string into an 128 char buffer? Guess who will have to take the blame for the consequences if this happens 'merely because we forgot to document' that particular limit?[5]

As for commenting 'weird logic', this is vital to the future maintainability of the code. Regular expressions for example often tend to be obscure, so explaining what they look for is recommended. If you want to be very clear about what they do, you can use the option 'x' (or set the 'IgnoreWhiteSpace' RegexOption in C# alternatively), to split the regex up over several lines, adding in comments where needed. Exhibit D, modified from the .NET framework SDK documentation demonstrates this:

Exhibit D

```
    void DumpHrefs(String inputString) //We're extracting href tags in
this
   {
      Regex r;
      Match m;

      r = new Regex("href\\s*=\\s*(?:\"(?<1>[^\"]*)\"|(?<1>\\S+))",
```

4 You should give the bug number assigned in your issue tracking system and a short description, e.g. 'memory leak on multiple invocation'.

5 Of course, sanity checks for such limited values also are a Very Good Thing™

```
          RegexOptions.IgnoreWhiteSpace|RegexOptions.IgnoreCase|RegexOptio
ns.Compiled);
      for (m = r.Match(inputString); m.Success; m = m.NextMatch())
      {
         Console.WriteLine("Found href " + m.Groups[1] + " at "
            + m.Groups[1].Index);
      }
   }
   void DumpHrefsclean(String inputString) //same as above, commented
   {
      Regex r;
      Match m;

      r = new Regex("href          #This looks  for the string 'href'
      \\s*=\\s                     #followed by whitespaces, '=', ws
      (?:\"(?<1>[^\"]*)\"          #a ':', + a group in '"', no '"' in it
      |                            #or
      (?<1>\\S+))",                #a group followed by non-spaces
      RegexOptions.IgnoreWhiteSpace|RegexOptions.IgnoreCase|
      RegexOptions.Compiled);
      for (m = r.Match(inputString); m.Success; m = m.NextMatch())
      {
         Console.WriteLine("Found href " + m.Groups[1] + " at "
            + m.Groups[1].Index);
      }
   }
'
```

Note that in this C# code segment the **comments inside** the regex have to be Perl-style, using '#' as delimiter. The comments extend from the '#' to the next newline character.

Another case of 'weird logic' would be switching between different programming paradigms – sequential, object oriented and functional or stack oriented programming may serve as examples. Such switches can be quite confusing when they are done without telling the reader. The following Perl code from T. Zlatanov's book illustrates this point for a filtering process for odd numbers in procedural and functional implementation:

Exhibit E

```
my @list = (1, 2, 3, 'hi');
my @results;
# the procedural approach
foreach (@list)
{
 push @results, $_
  unless $_%2;
}
# using grep - FP approach
@results = grep { !($_%2) } @list;
```

The looping construct of the procedural implementation is pretty straightforward and easily understood, but the FP implementation only makes sense when you know that it a) is FP and b) FP code is read right to left... Without being told that it is FP, it is rather hard to figure out what that – admittedly concise - line does.

3.4 General Commenting Style Recommendations

As we have seen in the above sections discussing types of comment, a useful comment always follows some basic rules of style. And as nowadays the API documentation for most programs is generated automatically from comments using one of the systems described in section 4 below, coherent, clearly written comments make for good documentation. Users will assume inferior product quality when they are confronted with bad documentation.

Question number one many programmers[6] tend to ask is '*how much comment is good for my code?',* or in other words, *'with how little comments can I get away?'*. This is not such an easy question to answer, as a simple numeric relationship between #linesofcode/#linesofcomment cannot be naively applied. Obviously, in a short program the share of comments will increase in comparison to a large scale software project's comment share. Personal experience shows though, that there seems to be a limit ratio approached in well documented code:

The ratio of code/comment lines tends to be about 2/1 in sizeable projects.

This ratio includes all three types of comment. Therefore, this empirically derived ratio is not a recommendation for verbose 'blow by blow' commenting style[7].

Now this takes us immediately to the next point: verbosity. There is no need to comment the obvious such as:

```
a++ // increases the counter
return} // returns from the subroutine
```

Such comments are a) an insult to your reader's intelligence and b) tedious to write. So just don't do it. Comments should be long enough to make their statement, but still concise. Detailed explanations can be worked into the documentation later on wherever necessary, after documentation has been generated from the sources. As a rule of thumb, the explanation of one line should not be longer than a line. Also, there usually is no need to explain a two line loop using a full paragraph of comment.

The amount of commenting can be reduced significantly by using meaningful variable and function names. The, ahem, “interesting” FORTRAN code in exhibit F illustrates this point quite clearly:

Exhibit F

```
      SUBROUTINE SNAILSORT (FOO, BAR, BAZ)
      INTEGER BAZ, CHIRP, FROOP
      REAL FOO(BAZ), BAR(BAZ)

C This subroutine sorts a vector FOO in ascending order
C onto a vector BAR

C Lookup of the biggest element of FOO

      CHIRP=BAZ
      BAR(BAZ)=FOO(1)
      DO J=1,BAZ
        IF(FOO(J).GT.BAR(BAZ)) THEN
          BAR(BAZ)=FOO(J)
          FROOP=J
        ENDIF
      ENDDO

C Lookup of the smallest element of FOO

      BAR(1)=FOO(I)
      DO J=1,BAZ
        IF(FOO(J).LT.BAR(1)) THEN
          BAR(1)=FOO(J)
        ENDIF
      ENDDO

C Lookup of the elements smaller than the last sorted
C Element and bigger than the other elements of FOO

55    CHIRP=CHIRP-1
      DO K=1,BAZ
        IF(FOO(K).GT.BAR(CHIRP).AND.FOO(K).LT.BAR(CHIRP+1)) THEN
```

6 Especially the hubristic and the lazy programmers :-)

7 You feared precisely that, didn't you?

```
      BAR(CHIRP)=FOO(K)
      FROOP=CHIRP
    ENDIF
  ENDDO
  IF(CHIRP.GT.2) GOTO 55
  RETURN
  END
```

Oh well. There is a number of things wrong here. Unambiguous variable names as well as routine names serve as part of the commenting of good, well written code. Here, the variable names either are obscure or rely on assumptions about implicit types, as the counter variables show. If the implicit typing changes in later compiler releases or language specification updates, the code is broken – a bad risk to take just to save a few keystrokes. The routine name of course is completely obtuse, even if it does give some indication of the algorithm's performance. In this particular case, even the explanatory comments don't help too much.

From the above exhibit, we can see that clearly naming functions, variables, objects, classes etc. is a part of the commenting process. It is not that important which naming scheme and convention is used, what is important is:

Once you settle for a naming scheme, stick with it. No exceptions to the rule!

Practical experience has shown that despite some disadvantages the socalled 'Hungarian Notation' used in Win32 programming is very useful. See the article by Hopfrog given in the references below for an overview of this style.

Another Golden Rule to be learned comes from Ottinger's Rules for Variable and Class Naming:

Nothing is intuitive

This holds true for naming as well as for code clarity. Read Ottinger's paper regardless of your preferred programming language and naming conventions. It contains many useful insights. Once again: comment your code!

Comments also can be kept short and concise by using one line commenting style wherever possible. Multiline comments should be used preferably for the comment block at the program file's head or for subroutine explanation comments.

The layout of the comments should be kept consistent wherever possible. Start comments always in the same columns if it can be done, e.g. at either column 1 or column 40. Exhibit C gives an example of how spacing is to be used inside comments. Consistent layout makes comments more easily searchable and will result in better looking documentation when the comments are extracted for this purpose.

For functional comments, the spelling of the keywords must be consistent, to help in finding the 'hot spots'. For example once you have settled for writing 'TODO', never use 'To do', 'todo' or 'To Do'. In contrast to the todos in exhibit C, a todo comment usually should be as close as possible to the location in the source code where it should be done.

In explanatory comments, especially when dealing with the description of subroutines or with workarounds around bugs, it is always recommended to state the problem as well as the solution, as this will a) make your code easier to understand and b) therefore easier to maintain and refactor and c) it will make you think more clearly about what you are doing and thus lead to your writing better code right from the start.

4 Documenting Systems

There is a number of documenting systems available for various programming languages. These systems usually deal with the 'explanatory' type of comment. They serve the purpose of generating

documentation for the program's interfaces out of the comments embedded in the code.

4.1 Tangle/Weave

One of the most venerable efforts in this area is Donald E. Knuth's `tangle/weave` package for 'literate programming', which is based on a system called 'web'. This system allows to generate documentation out of comments in either Pascal or C code. Output is – of course – in Knuth's TeX formatting language. It still is in use in academic environments. As its importance outside the academic community is near nil, it will not be discussed here.

4.2 Perlpod

The Perl programming language comes with the POD system, short for 'Plain Old Documentation'. This uses a certain set of command strings inserted into comments to direct the ouput produced. Output from this system is flexible, ranging from text files over nroff to HTML, TeX or worse. The first bit of the perlpod documentation is listed below to give an idea of the pod commenting commands available and the functionality they provide.

Exhibit G

```
NAME
    perlpod - plain old documentation

DESCRIPTION
    A pod-to-whatever translator reads a pod file paragraph by paragraph,
    and translates it to the appropriate output format. There are three
    kinds of paragraphs: verbatim, command, and ordinary text.

  Verbatim Paragraph

    A verbatim paragraph, distinguished by being indented (that is, it
    starts with space or tab). It should be reproduced exactly, with tabs
    assumed to be on 8-column boundaries. There are no special formatting
     escapes, so you can't italicize or anything like that. A \ means \,
and
    nothing else.

  Command Paragraph

    All command paragraphs start with "=", followed by an identifier,
      followed by arbitrary text that the command can use however it
pleases.
    Currently recognized commands are

        =head1 heading
        =head2 heading
        =item text
        =over N
        =back
        =cut
        =pod
        =for X
        =begin X
        =end X

    =pod
    =cut
         The "=pod" directive does nothing beyond telling the compiler to
lay
        off parsing code through the next "=cut". It's useful for adding
         another paragraph to the doc if you're mixing up code and pod a
lot.
```

To demonstrate what even a 'simple' documentation system like POD can do, let us look at a problem in formatting a documentary comment:

Exhibit H

```
> > =pod
> >
>  > Alternatively, you may use this function, which shuffles its
arguments
> > in place.  The algorithm is known as a Fisher-Yates shuffle, and can
> > be proven to produce a uniform distribution of permutations, provided
> > that the random number generator is sufficiently random.
>
> I do think it needs a reference to Knuth [1]. Or to the original
> publication of Fisher and Yates [2].
>
> [1] D. E. Knuth: I<The Art of Computer Programming>, Volume 2,
>     Third edition. Section 3.4.2, Algorithm P, pp 145. Reading:
>     Addison-Wesley, 1997. ISBN: 0-201-89684-2.
>
> [2] R. A. Fisher and F. Yates: I<Statistical Tables>. London, 1938.
>     Example 12.

So, how _does_ one properly mark that up in POD?  There are no real
footnotes, and embedding the entire reference in the text would mess up
the already strained flow of the paragraph.  Hmm...  :-(
```

This is taken from an email in a discussion thread on the 'Fun with Perl' mailing list. It seems a daunting task, looking at what can be done according to the above excerpt from the Perlpod documentation. But here is more. With a little bit of ingenuity, it is possible to come up with something like this:

Exhibit I

```
=head1 LITERATURE

The algorithm used is discussed by Knuth [3]. It was first published
by Fisher and Yates [2], and later by Durstenfeld [1].

=head1 CAVEAT

Salfi [4] points to a big caveat. If the outcome of a random generator is
solely based on the value of the previous outcome, like a linear
congruential method, then the outcome of a shuffle depends on exactly
three things: the shuffling algorithm, the input and the seed of the
random generator. Hence, for a given list and a given algorithm, the
outcome of the shuffle is purely based on the seed. Many modern computers
have 32 bit random numbers, hence a 32 bit seed. Hence, there are at most
2^32 possible shuffles of a list, foreach of the possible algorithms. But
for a list of n elements, there are n! possible permutations. Which means
that a shuffle of a list of 13 elements will not generate certain
permutations, as 13! > 2^32.

=head1 REFERENCES

=over

=item [1]

R. Durstenfeld: I<CACM>, B<7>, 1964. pp 420.

=item [2]

R. A. Fisher and F. Yates: I<Statistical Tables>. London, 1938.
Example 12.

=item [3]

D. E. Knuth: I<The Art of Computer Programming>, Volume 2, Third edition.
Section 3.4.2, Algorithm P, pp 145. Reading: Addison-Wesley, 1997. ISBN:
0-201-89684-2.
```

```
=item [4]

R. Salfi: I<COMPSTAT 1974>. Vienna: 1974, pp 28 - 35.

=back
```

Not bad for such a relatively limited system. Good, clearly written comments do help get our intention and the background for our architectural decisions across to our fellow coders who will also have to live and work with our code. And not least of all, this type of comment will make life easier when we have to justify our implementation decisions to management. And if we ever intend to publish our code in printed form - even excerpts - good commenting style and habits will save us a lot of work, as the comments can be reused in the text with minimum effort.

4.3 Javadoc

Javadoc is the Java API documentation generator contained in the JDK. It uses comments in which specific tags are embedded to generate HTML pages containing descriptions of all public and protected classes, interfaces, constructors, methods and fields. Javadoc also generates a tree of the class hierarchy and an index of all members. The syntax of Javadoc comments is as follows:

Exhibit J

```
/**
*A class representing a window on the screen
*example:
*<pre>
*   Window win = new Window(parent)
*   win.show();
*</pre>
*
*@author Sami Shaio
*@version %I%, %G%
*@see java.awt.BaseWindow
*@see java.awt.Button
*/
class Window extends BaseWindow {
...
}
```

This example from the Javadoc documentation shows that first and last line have to be in a strict format, they may not be altered. All other lines must begin with an asterisk, everything before that asterisk is discarded. And we may use HTML tags inside Javadoc comments – with the exception of tags that indicate hierarchy, such as <Hn> tags. The Javadoc tags as such all start with an at sign.

Exhibit K

@author name-text	Author's name
@version version-text	Version info, only one such tag allowed
@param parameter name, description	Adds a parameter to the 'parameters' section, may be multiline
@return description	Adds a 'returns' section
@exception fully-qualified-class-name description	Adds a 'throws' section. Linked to the class documentation
@see classname	Adds a 'see also' hyperlink
@since since-text	Indicates since which version this feature exists

@deprecated deprecated-text	Indicates deprecation of an API and gives new API

The first two tags are for use in class and interface documentation, the remainder may be used for constructor and method documentation, with the last two allowed in both class & interface and constructor & method documentation. For instructions on the use and options of the Javadoc tool see the documentation provided with the JDK. The tool delivers different output formats in its JDK 1.1 and JDK 1.2 versions respectively. Again refer to the documentation supplied with your JDK.

4.4 PHPdoc

PHPdoc is an implementaton of the Javadoc documenting system for use with PHP scripts. Tags follow the same convention, so we need not go into detail in this technote.

4.5 C# Xml Comments

For the .NET framework, and the C# language specifically, another documenting system has been designed in which XML code is embedded into standard C# comments. These tags can then be extracted using either the C# compiler with the `/doc:<xml_outfile.xml>` option set or third party tools such as NDOC. These tools will convert the comments extracted into some other format, e.g. Windows Help files.

The nice part of this xml-based system is that the fields most often needed in professionally commenting code are predefined xml tags:

Exhibit L

<c>	Marks a part of a comment to be formatted as code
<code>	As above, but multiline
<example>	For embedding examples in comments, usually uses <c>
<exception>*	Documents an Exception Class
<include>*	Includes documentation from other files
<list>	A list of <term>s defined by <description>s
<para>	Structures text blocks, e.g in a <remark>
<param>*	Describes a method parameter
<paramref>*	Indicates that a word is used as reference to a parameter
<permission>*	Gives the access permissions to a member
<remarks>	For overview of what a given class or other type does
<returns>	Description of the return value
<see> *	Refers to a member or field available
<seealso>*	As above, but displays a 'See also' section
<summary>	A summary of the object
<value>	Describes a property

* The C# Compiler validates these tags, as they are supposed to point to

other code elements.

Note that this being xml, the corresponding closing tags are compulsory. This style of xml comments may be nested as is common for xml tags, e.g. <example> usually surrounds <c> or <code> tags. For a full explanation of this style of commenting as well as for subtags available for some tags – e.g. <list>, please refer to the .NET Framework SDK documentation. An example for the use of xml comments:

Exhibit M

```
// XMLsample.cs
// compile with: /doc:XMLsample.xml
using System;

/// <summary>
/// Class level summary documentation goes here.</summary>
/// <remarks>
/// Longer comments can be associated with a type or member
/// through the remarks tag</remarks>
public class SomeClass
{
   /// <summary>
   /// Store for the name property</summary>
   private string myName = null;

   /// <summary>
   /// The class constructor. </summary>
   public SomeClass()
   {
       // TODO: Add Constructor Logic here
   }

   /// <summary>
   /// Name property </summary>
   /// <value>
   /// A value tag is used to describe the property value</value>
   public string Name
   {
      get
      {
         if ( myName == null )
         {
            throw new Exception("Name is null");
         }

         return myName;
      }
   }

   /// <summary>
   /// Description for SomeMethod.</summary>
   /// <param name="s"> Parameter description for s goes here</param>
   /// <seealso cref="String">
    /// You can use the cref attribute on any tag to reference a type or
member
   /// and the compiler will check that the reference exists. </seealso>
   public void SomeMethod(string s)
   {
   }

   /// <summary>
   /// Some other method. </summary>
   /// <returns>
   /// Return results are described through the returns tag.</returns>
   /// <seealso cref="SomeMethod(string)">
    /// Notice the use of the cref attribute to reference a specific
method </seealso>
   public int SomeOtherMethod()
   {
      return 0;
```

```
    }

    /// <summary>
    /// The entry point for the application.
    /// </summary>
    /// <param name="args"> A list of command line arguments</param>
    public static int Main(String[] args)
    {
       // TODO: Add code to start application here

        return 0;
    }
}
```

One notable omission however has been made in these commenting tags: there are no documentary commenting tags available. We have thus to take care ourselves to properly document the history of our code. It is suggested to use the commenting tags provided by the version control system you use for this purpose. As an example for version control tags, we will look at those provided by

4.6 CVS

CVS, the 'Concurrent Versioning System', lets you insert certain keywords into your source code to include some of the information needed for documentary commenting. This works independent of the programming language used. Basically, we insert something like '$keyword$' which will be expanded to '$keyword:value$' by CVS when we obtain a new revision of the file. The following keywords can be used:

Exhibit M

```
$Author$
The login name of the user who checked in the revision.

$Date$
The date and time (UTC) the revision was checked in.

$Header$
A standard header containing the full pathname of the RCS file, the
revision number, the date (UTC), the author, the state, and the locker
(if locked). Files will normally never be locked when you use CVS.

$Id$
Same as $Header$, except that the RCS filename is without a path.

$Name$
Tag name used to check out this file. The keyword is expanded only if one
checks out with an explicit tag name. For example, when running the
command cvs co -r first, the keyword expands to `Name: first'.

$Locker$
The login name of the user who locked the revision (empty if not locked,
which is the normal case unless cvs admin -l is in use).

$Log$
The log message supplied during commit, preceded by a header containing
the RCS filename, the revision number, the author, and the date (UTC).
Existing log messages are not replaced. Instead, the new log message is
inserted after $Log:...$. Each new line is prefixed with the same string
which precedes the $Log keyword. For example, if the file contains

    /* Here is what people have been up to:
    *
    * $Log: frob.c,v $
    * Revision 1.1  1997/01/03 14:23:51  joe
    * Add the superfrobnicate option
    *
    */
```

This short excerpt from the CVS manual demonstrates that using a versioning system can significantly lessen the burden of keeping documentary comments up to date.

5 Conclusion

Comments are a central part of professional coding practice. We can divide comments into three categories: documentary, serving the purpose of documenting the evolution of code over time, functional, helping the co-ordination of the development effort inside the team and explanatory comments, generating the software documentation for general use. All three categories are vital to the success of a software development project.

Fail on one of the three comment types and you will fail on your project's success. Sorry, that is the way it is.

On the bright side of things, careful commenting will make your code easier to read and maintain, impress your employer[8] and fellow coders and boost your reputation as a good coder 'who does it the Right Way'.

Now go out to your code, comment thoroughly and reap the rewards.

6 References

.NET Framework SDK documentation

Per Cederqvist: Version Management with CVS, http://www.cvshome.org/docs/manual, 2001

Hopfrog: Hungarian Notation, http://www.kuro5hin.org/story/2002/4/12/153445/601, 2002

Java JDK tools documentation

Mike Krüger: C# coding style

Ottinger: Ottinger's Rules for Variable and Class Naming
http://www.cs.umd.edu/users/cml/cstyle/ottinger-naming.html, 1997

Michael Roberts: Literate Programming,
http://www.vivtek.com/litprog.html

Neal Stephenson: Cryptonomicon, Perennial books, New York, 2000

perldoc perlpod (online Perl documentation)

Teodor Zlatanov: The road to better Programming, Chapter 2,
http://www-106.ibm.com/developerworks/library/l-road2.html, 2001

Teodor Zlatanov: The road to better Programming, Chapter 4,
http://www-106.ibm.com/developerworks/library/l-road4.html, 2001

8 Maybe even enough to raise your pay or promote you :-)