NAME

Smart::Comments − Comments that do more than just sit there

VERSION

This document describes Smart::Comments version 1.000005

SYNOPSIS

```
use Smart::Comments;

my $var = suspect_value();

### $var

### got: $var

### Now computing value...

# and when looping:

for my $big_num (@big_nums) {  ### Factoring...      done
    factor($big_num);
}

while ($error > $tolerance) {  ### Refining--->      done
    refine_approximation()
}

for (my $i=0; $i<$MAX_INT; $i++) {   ### Working===[%]     done
    do_something_expensive_with($i);
}
```

DESCRIPTION

Smart comments provide an easy way to insert debugging and tracking code into a program. They can report the value of a variable, track the progress of a loop, and verify that particular assertions are true.

Best of all, when you're finished debugging, you don't have to remove them. Simply commenting out the `use Smart::Comments` line turns them back into regular comments. Leaving smart comments in your code is smart because if you needed them once, you'll almost certainly need them again later.

INTERFACE

All smart comments start with three (or more) # characters. That is, they are regular #−introduced comments whose first two (or more) characters are also #'s.

Using the Module

The module is loaded like any other:

```
use Smart::Comments;
```

When loaded it filters the remaining code up to the next:

```
no Smart::Comments;
```

directive, replacing any smart comments with smart code that implements the comments behaviour.

If you're debugging an application you can also invoke it with the module from the command-line:

```
perl −MSmart::Comments $application.pl
```

Of course, this only enables smart comments in the application file itself, not in any modules that the application loads.

You can also specify particular levels of smartness, by including one or more markers as arguments to the `use`:

```
use Smart::Comments '###', '####';
```

These arguments tell the module to filter only those comments that start with the same number of #'s. So the above `use` statement would ''activate'' any smart comments of the form:

```
    ###   Smart...

    ####  Smarter...
```

but not those of the form:

```
    ##### Smartest...
```

This facility is useful for differentiating progress bars (see ‘‘Progress Bars’’), which should always be active, from debugging comments (see ‘‘Debugging via Comments’’), which should not:

```
    #### Debugging here...

    for (@values) {         ### Progress: 0...  100
        do_stuff();
    }
```

Note that, for simplicity, all smart comments described below will be written with three #’s; in all such cases, any number of #’s greater than three could be used instead.

Debugging via Comments

The simplest way to use smart comments is for debugging. The module supports the following forms, all of which print to `STDERR`:

`### LABEL : EXPRESSION`

The LABEL is any sequence of characters up to the first colon. The EXPRESSION is any valid Perl expression, including a simple variable. When active, the comment prints the label, followed by the value of the expression. For example:

```
    ### Expected: 2 * $prediction
    ###      Got: $result
```

prints:

```
    ### Expected: 42
    ###      Got: 13
```

`### EXPRESSION`

The EXPRESSION is any valid Perl expression, including a simple variable. When active, the comment prints the expression, followed by the value of the expression. For example:

```
    ### 2 * $prediction
    ### $result
```

prints:

```
    ### 2 * $prediction: 42
    ### $result: 13
```

`### TEXT...`

The TEXT is any sequence of characters that end in three dots. When active, the comment just prints the text, including the dots. For example:

```
    ### Acquiring data...

    $data = get_data();

    ### Verifying data...

    verify_data($data);

    ### Assimilating data...

    assimilate_data($data);

    ### Tired now, having a little lie down...

    sleep 900;
```

would print:

```
### Acquiring data...

### Verifying data...

### Assimilating data...

### Tired now, having a little lie down...
```

as each phase commenced. This is particularly useful for tracking down precisely where a bug is occurring. It is also useful in non-debugging situations, especially when batch processing, as a simple progress feedback mechanism.

Within a textual smart comment you can use the special sequence `<now>` (or `<time>` or `<when>`) which is replaced with a timestamp. For example:

```
### [<now>] Acquiring data...
```

would produce something like:

```
### [Fri Nov 18 15:11:15 EST 2005] Acquiring data...
```

There are also ‘‘spacestamps’’: `<here>` (or `<loc>` or `<place>` or `<where>`):

```
### Acquiring data at <loc>...
```

to produce something like:

```
### Acquiring data at "demo.pl", line 7...
```

You can also request just the filename (`<file>`) or just the line number (`<line>`) to get finer control over formatting:

```
### Acquiring data at <file>[<line>]...
```

and produce something like:

```
### Acquiring data at demo.pl[7]...
```

You can, of course, use any combination of stamps in the one comment.

Checks and Assertions via Comments

```
### require: BOOLEAN_EXPR
### assert:  BOOLEAN_EXPR
### ensure:  BOOLEAN_EXPR
### insist:  BOOLEAN_EXPR
```

These four are synonyms for the same behaviour. The comment evaluates the expression in a boolean context. If the result is true, nothing more is done. If the result is false, the comment throws an exception listing the expression, the fact that it failed, and the values of any variables used in the expression.

For example, given the following assertion:

```
### require: $min < $result && $result < $max
```

if the expression evaluated false, the comment would die with the following message:

```
### $min < $result && $result < $max was not true at demo.pl line 86.
###     $min was: 7
###     $result was: 1000004
###     $max was: 99
```

```
### check:   BOOLEAN_EXPR
### confirm: BOOLEAN_EXPR
### verify:  BOOLEAN_EXPR
```

These three are synonyms for the same behaviour. The comment evaluates the expression in a boolean context. If the result is true, nothing more is done. If the result is false, the comment prints a warning message listing the expression, the fact that it failed, and the values of any variables

used in the expression.

The effect is identical to that of the four assertions listed earlier, except that `warn` is used instead of `die`.

Progress Bars

You can put a smart comment on the same line as any of the following types of Perl loop:

```
foreach my VAR ( LIST ) {       ### Progressing...   done

for my VAR ( LIST ) {           ### Progressing...   done

foreach ( LIST ) {              ### Progressing...   done

for ( LIST ) {                  ### Progressing...   done

while (CONDITION) {             ### Progressing...   done

until (CONDITION) {             ### Progressing...   done

for (INIT; CONDITION; INCR) {   ### Progressing...   done
```

In each case, the module animates the comment, causing the dots to extend from the left text, reaching the right text on the last iteration. For ‘‘open ended’’ loops (like `while` and C−style `for` loops), the dots will never reach the right text and their progress slows down as the number of iterations increases.

For example, a smart comment like:

```
for (@candidates) {       ### Evaluating...     done
```

would be animated is the following sequence (which would appear sequentially on a single line, rather than on consecutive lines):

```
Evaluating                          done

Evaluating......                    done

Evaluating............              done

Evaluating..................        done

Evaluating.........................done
```

The module animates the first sequence of three identical characters in the comment, provided those characters are followed by a gap of at least two whitespace characters. So you can specify different types of progress bars. For example:

```
for (@candidates) {       ### Evaluating:::     done
```

or:

```
for (@candidates) {       ### Evaluating===     done
```

or:

```
for (@candidates) {       ### Evaluating|||     done
```

If the characters to be animated are immediately followed by other non-whitespace characters before the gap, then those other non-whitespace characters are used as an ‘‘arrow head’’ or ‘‘leader’’ and are pushed right by the growing progress bar. For example:

```
for (@candidates) {       ### Evaluating===|    done
```

would animate like so:

```
Evaluating|                         done

Evaluating=====|                    done
```

```
Evaluating===========|             done

Evaluating=================|       done

Evaluating=========================done
```

If a percentage character (%) appears anywhere in the comment, it is replaced by the percentage completion. For example:

```
for (@candidates) {       ### Evaluating [===|    ] % done
```

animates like so:

```
Evaluating [|               ]   0% done

Evaluating [===|            ]  25% done

Evaluating [========|       ]  50% done

Evaluating [============|   ]  75% done

Evaluating [=================] 100% done
```

If the % is in the ‘‘arrow head’’ it moves with the progress bar. For example:

```
for (@candidates) {       ### Evaluating |===[%]    |
```

would be animated like so:

```
Evaluating |[0%]                      |

Evaluating |=[25%]                    |

Evaluating |========[50%]             |

Evaluating |===============[75%]      |

Evaluating |==========================|
```

For ‘‘open-ended’’ loops, the percentage completion is unknown, so the module replaces each % with the current iteration count. For example:

```
while ($next ne $target) {       ### Evaluating |===[%]    |
```

would animate like so:

```
Evaluating |[0]                       |

Evaluating |=[2]                      |

Evaluating |==[3]                     |

Evaluating |===[5]                    |

Evaluating |====[7]                   |

Evaluating |=====[8]                  |

Evaluating |======[11]                |
```

Note that the non-sequential numbering in the above example is a result of the ‘‘hurry up and slow down’’ algorithm that prevents open-ended loops from ever reaching the right-hand side.

As a special case, if the progress bar is drawn as two pairs of identical brackets:

```
for (@candidates) {       ### Evaluating: [][]

for (@candidates) {       ### Evaluating: {}{}
```

```
for (@candidates) {       ### Evaluating: ()()

for (@candidates) {       ### Evaluating: <><>
```

Then the bar grows by repeating bracket pairs:

```
Evaluating: [

Evaluating: []

Evaluating: [][

Evaluating: [][]

Evaluating: [][][
```

etc.

Finally, progress bars don't have to have an animated component. They can just report the loop's progress numerically:

```
for (@candidates) {       ### Evaluating (% done)
```

which would animate (all of the same line):

```
Evaluating (0% done)

Evaluating (25% done)

Evaluating (50% done)

Evaluating (75% done)

Evaluating (100% done)
```

Time-Remaining Estimates

When a progress bar is used with a `for` loop, the module tracks how long each iteration is taking and makes an estimate of how much time will be required to complete the entire loop.

Normally this estimate is not shown, unless the estimate becomes large enough to warrant informing the user. Specifically, the estimate will be shown if, after five seconds, the time remaining exceeds ten seconds. In other words, a time-remaining estimate is shown if the module detects a `for` loop that is likely to take more than 15 seconds in total. For example:

```
for (@seven_samurai) {      ### Fighting: [|||    ]
    fight();
    sleep 5;
}
```

would be animated like so:

```
Fighting: [                          ]

Fighting: [||||                      ]

Fighting: [|||||||||                 ]  (about 20 seconds remaining)

Fighting: [|||||||||||||             ]  (about 20 seconds remaining)

Fighting: [||||||||||||||||          ]  (about 10 seconds remaining)

Fighting: [||||||||||||||||||||||    ]  (less than 10 seconds remaining)

Fighting: [||||||||||||||||||||||||||]
```

The precision of the reported time-remaining estimate is deliberately vague, mainly to prevent it being annoyingly wrong.

DIAGNOSTICS

In a sense, everything this module does is a diagnostic. All comments that print anything, print it to `STDERR`.

However, the module itself has only one diagnostic:

`Incomprehensible arguments: %s in call to 'use Smart::Comments`

You loaded the module and passed it an argument that wasn't three−or− more #'s. Arguments like `'###'`, `'####'`, `'#####'`, etc. are the only ones that the module accepts.

CONFIGURATION AND ENVIRONMENT

Smart::Comments can make use of an environment variable from your shell: `Smart_Comments`. This variable can be specified either with a true/false value (i.e. 1 or 0) or with the same arguments as may be passed on the `use` line when loading the module (see "INTERFACE"). The following table summarizes the behaviour:

```
      Value of
$ENV{Smart_Comments}          Equivalent Perl

         1                    use Smart::Comments;
         0                     no Smart::Comments;
    '###:####'                use Smart::Comments qw(### ####);
    '### ####'                use Smart::Comments qw(### ####);
```

To enable the `Smart_Comments` environment variable, you need to load the module with the `−ENV` flag:

```
use Smart::Comments −ENV;
```

Note that you can still specify other arguments in the `use` statement:

```
use Smart::Comments −ENV, qw(### #####);
```

In this case, the contents of the environment variable replace the `−ENV` in the argument list.

DEPENDENCIES

The module requires the following modules:

- Filter::Simple
- version
- List::Util
- Data::Dumper
- Text::Balanced

INCOMPATIBILITIES

None reported. This module is probably even relatively safe with other Filter::Simple modules since it is very specific and limited in what it filters.

BUGS AND LIMITATIONS

No bugs have been reported.

This module has the usual limitations of source filters (i.e. it looks smarter than it is).

Please report any bugs or feature requests to `bug−smart−comments AT rt DOT cpan DOT org`, or through the web interface at <http://rt.cpan.org>.

REPOSITORY

<https://github.com/neilb/Smart−Comments>

AUTHOR

Damian Conway `<DCONWAY AT cpan DOT org>`

LICENCE AND COPYRIGHT

itself.

DISCLAIMER OF WARRANTY

BECAUSE THIS SOFTWARE IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE SOFTWARE, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE SOFTWARE ‘‘AS IS’’ WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE SOFTWARE IS WITH YOU. SHOULD THE SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR, OR CORRECTION.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE SOFTWARE AS PERMITTED BY THE ABOVE LICENCE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE SOFTWARE TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.