Code Review Guidelines

T.A. Gonsalves & Hema A. Murthy

6/11/2001

1. Purpose

The purpose of a code review is for someone other than the programmer to critically go through the code of a module to ensure that it meets the functional and design specifications, is well-written and robust. An incidental benefit is that the reviewer may learn some new programming techniques, and more people in the team become familiar with the module.

The reviewer should not try to fix any bugs or improve the code. S/he should merely inform the programmer about potential bugs and possible improvements. The responsibility for actually making the changes and testing them lies with the programmer.

1.1 Overview

The steps to be followed in code review are given below. This is followed by detailed instructions in Section 3.

2. Steps in Code Review

1. Obtain print-outs of the specs and design documents, and of the code. Write your comments neatly on the print-outs, with your name, date and other relevant details.

2. Read through the specs and design documents to get an understanding of the purpose of the code and how it achieves this purpose.

3. Compare the class hierarchy and/or function call-tree from the design document with the actual code. Note any discrepancies.

4. Identify the important data structures from the design document. Check this against the actual code. Note any discrepancies.

5. Check for adherence to the project's coding standard (DONlab Coding Standard in the case of TeNeT Group projects).

6. Check the style and correctness of each of the following:
   a. each file [Sec. 3.2]
   b. each class [Sec. 3.3]
   c. each function/method [Sec. 3.4]

7. Check the handling of exceptions and errors [Sec 3.5]

8. Check the user interaction [Sec. 3.6]

9. Look for common errors [Sec. 3.7]

10. Read the test plan. Verify that it checks the software limits. [Sec. 3.8]

3. Detailed Instructions

3.1 General

* Comments in Javadoc format in the case of Java files.
* In-line comment for each global or other important variable
* Declarations of logically related variables grouped together.
  For example: first input parameters, then output variables, then temporaries, with a comment and blank line separating the blocks.
* Simple names for temporaries. For example: String s; int num;
* Reuse temporaries. For example: use "s" instead of "s1...s2...s3".

```
 * Comment at the start of statement block such as loop or conditional
 * Comment to explain any unusual code
 * Proper choice and capitalisation of names

 3.2 File Level
 --------------
 * top-of-file comment present and follows DONlab template.
 * All fields such as purpose, CVS/RCS id and log (may be at end of
     file) filled in.
 * Date stamp and programmer's name current.
 * Copyright notice such as "Copyright (C) 2001 NMSWorks Software Ltd."
     Find out from the project manager what is appropriate.
 * "Imports" and "uses" clauses -- check for cyclic dependencies (file
     A uses file B which uses file A...

 3.3 Class Level
 ---------------
 * Class comment present and follows DONlab template.
 * Class variables tally with design document
 * "public" variables to be used rarely -- mainly in the case of
   data-only classes
 * Any "static" variable must have an explanatory comment
 * Number of functions/class should not exceed 10 normally

 3.4 Function/Method Level
 -------------------------
 * Comment block with fields (purpose, arguments, return, errors)
     filled in
 * Any "static" variable must have an explanatory comment
 * Length should not exceed 30 lines normally
 * Function should normally have arguments, i.e., minimise direct use
   of global or class variables.

 3.5 Exception and Error Handling
 --------------------------------
 * All exceptions must be handled in one of the following ways:
  - propagate the exception
  - catch the exception and take some action
  - catch the exception and ignore it.  *Must* have an explanatory comment.

 * If a function returns an error code, it must be checked and action
   taken as above

 * Recovery action (exit, ignore or retry) must be justified

 * Output from the error/exception handling:
  - a low-level, library function:
      log the error (using CygLog or CygError).  Should not popup a
      dialog, or write to stdout or exit
  - a non-interactive daemon:
      log the error (using CygLog or CygError).  Should not popup a
      dialog, or write to stdout
  - an interactive function:
      popup a dialog box or write to stdout (depending on whether it is
      a GUI or CLI); optionally also log the error

 * Use the project's standard error handling function/class.

 3.6 User Interaction
 --------------------
 * Messages in plain, simple English
 * Avoid exclamation marks "!" and all CAPITALS
 * Avoid use of internal variable names and technical jargon that may
   not be meaningful to the user
 * Check for spelling, grammatical mistakes

 3.7 Common Coding Errors
 ------------------------
 * no default case in a "switch" or nested "if-then-else-if..."
```

```
 * no error code to handle "impossible" cases (which will occur
     sooner or later)
 * error return value of a system call or function is ignored
 * exception thrown by a function is not caught by the caller
 * bounds of fixed-size arrays or vectors exceeded
 * '<' instead of '<='
   '>' instead of '>='
   '==' instead of '!='
   '=' instead of '=='
 * range of a data type such as char or int exceeded
 * type-casts that could cause errors
 * unnecessary use of language features.  For example,
        Integer object instead of an int
        Vector instead of an array.
 * use of a variable instead of a constant
 * use of numbers in the code instead of defining symbolic constants
 * use of uninitiallised variables

 3.8 Test Plan

 "White-box" tests should test the limits of the data and control
 structures in the program, and boundary cases of data values.
 * full/empty state of each array
 * function should return different classes of values: negative,
     zero, positive, null, non-null
 * limits of char/byte/short/int variables, if appropriate
 * limits of strings
 * observer gets called
 * signal handler gets called
 * unusual cases in type-casts
 * host vs. network order of addresses

 * Code coverage percentage (percentage of statements executed at least once)
 * Effort (running the tests should not take more than a few minutes
   for a small module to a few hours for a large module)

 4. Homily
 ---------
 Code review has many benefits -- it helps to ensure a bug-free, robust
 product; the reviewer and the programmer both learn in the process.
 Time spent in code review is time well spent.

 Keep in mind that even the best programmers make mistakes, and most
 people have an ego.  Your comments should be technical in nature, and
 be constructively critical.  Make suggestions, do not lay down the
 law.  If possible, give pointers to authoritative books/articles.
```