Code Review Guidance

Table of Contents

1. Introduction

This guidance provides a checklist that need to be used during code reviews. It does not describe how or which tools should be used to perform the code review.

The scope of this document is limited to the ISCV code development. Some of the items however are not so specific and can be used as checklist items in code reviews.

This checklist helps to improve the quality of code reviews. The checklist can be used by both the author of the code as well as the reviewer. Of course, not all checks apply to all situations.

2. Checklist

2.1. Procedural checks

2.1.1. Pre-check

- The right persons must be involved in the review (regarding experience level).
- Do not review files that still need to be merged if the merge can be done before the review. Reviewing after the merge will help to spot merge errors.
- Do not start to review if the request is unclear or incomplete, but ask for a new request.

2.1.2. Contents review request

- Mention the location where the files are located.
- Mention the PR number and headline.
- Add explanation of the proposed functionality.
- Add explanation of the changes. For the complete change set, and for each individual file.

- All files must be explicitly listed in the review request. Don't use wildcards like in Configuration.*.cpp etc. Ideally, the list of files that will be used during check in should be included in the review request. This will help to avoid errors during the actual check in.
- Make sure that the complete history of the review sequence will be used to check-in, create or delete files. This preserves insights in decisions / changes made.
- Mention the locations of new files explicitly.
- Include the unit test code (when/if applicable).

2.1.3. Regression test

- The change should include unit test code (when/if applicable).
- Run any automated unit, regression test that can test the changes, e.g.
- Update related test specs if needed.

2.2. Code review checks

2.2.1. Structure

- Does the code completely and correctly implement the design?
- Does the code conform to any pertinent coding standards?
- Is the code well-structured, consistent in style, and consistently formatted?
- Are there any uncalled or unneeded procedures or any unreachable code?
- Are there any leftover stubs or test routines in the code?
- Can any code be replaced by calls to external reusable components or library functions?
- Are there any blocks of repeated code that could be condensed into a single procedure?
- Is storage use efficient?
- Are symbolics used rather than “magic number” constants or string constants?
- Are any modules excessively complex and should be restructured or split into multiple routines?

2.2.2. Commenting style

- Is the code clearly and adequately documented with an easy-to-maintain commenting style?
- Are all comments consistent with the code?

2.2.3. Variables

- Are all variables properly defined with meaningful, consistent, and clear names? For instance check if upper/lowercase is consistently used, for example: iValue versus IValue.
- Do all assigned variables have proper type consistency or casting?
- Are there any redundant or unused variables?

2.2.4. Arithmetic Operations

- Does the code avoid comparing floating-point numbers for equality?
- Does the code systematically prevent rounding errors?
- Does the code avoid additions and subtractions on numbers with greatly different magnitudes?
- Are divisors tested for zero or noise?

2.2.5. Loops and Branches

- Are all loops, branches, and logic constructs complete, correct, and properly nested?
- Are the most common cases tested first in IF- -ELSEIF chains?
- Are all cases covered in an IF- -ELSEIF or CASE block, including ELSE or DEFAULT clauses?
- Does every case statement have a default?

- Are loop termination conditions obvious and invariably achievable?
- Are indexes or subscripts properly initialized, just prior to the loop?
- Can any statements that are enclosed within loops be placed outside the loops?
- Does the code in the loop avoid manipulating the index variable or using it upon exit from the loop?
- Determine the cyclomatic complexity and consider if it can be improved

2.2.6. Defensive Programming

- Are indexes, pointers, and subscripts tested against array, record, or file bounds?
- Are the exceptions being handled?
- Are imported data and input arguments tested for validity and completeness?
- Are all output variables assigned?
- Are the correct data operated on in each statement?
- Is every memory allocation de-allocated?
- Are timeouts or error traps used for external device accesses?
- Are files checked for existence before attempting to access them?
- Are all files and devices are left in the correct state upon program termination?

2.2.7. Various

- Document the ownership of each argument passed to/returned by a method or function (who should free it).
- Document the access strategy for objects that can be accessed (who should release it).
- Document whether the code is MT-safe or not.
- Protect all static (class/instance) variables in files (classes) that are supposed to be MT-safe
- Use symmetric constructs for opposite actions like new/free, open/close, access/de-access, subscribe/unsubscribe.
- Check the log files (set logging level to development to have maximum amount of logging). The log files should not contain messages about dangling subscriptions.
- Check and solve compiler warnings.
- Check and solve warnings of static code checkers (e.g. TICS (C++, C#), JSLint (Javascript), …).
- Rebuild all changed code to catch dependency errors and warnings.

2.2.8. Portability

- Don’t use filenames that match other filenames.
- Avoid new differences between development environment and the final product.
- Do not add development tools to product executables.
- Beware that there are differences between the development environment and the environment on which a product is installed. You can think about problems relating to access-rights, pathnames, licenses etc.

2.2.9. Translatability

- Have the texts reviewed by a specialist, to improve translatability and consistency.
- Update tool tips and the references to Instruction for Use if needed.
- Do not limit the size of text fields to the size of the English text. Be aware that in most cases other languages need more space! Therefore, make text fields as large as possible.
- If translations are already available, make sure that the translations fit on the buttons.

3. Document Control

Process	Owner(s)s
Approval	Godfried Jansen, Jarmila Bokkerink
Review	Egbert Algra
Author	Hans Kersbergen
Approval date	See eDMS
Effective date plan	Effective when published. QMS training required

4. Document Change Summary

Revision	Description of changes
1	Initial release
2	Editorial change in 'Effective date plan', changed to: QMS training required

This is a representation of an electronic record that was signed electronically in our Regulated System.

This page is the manifestation of the electronic signatures used in compliance with the organizations electronic signature policies and procedures.

UserName: Godfried Jansen (nlv22180)
Title: Sr. Manager Q&R
Date: Thursday, 16 July 2015, 04:15 PM W. Europe Daylight Time
Meaning: This document has changed to Authorized status
==========

UserName: Jarmila Bokkerink-Scheerova (nly14726)
Title: R&D Director
Date: Thursday, 16 July 2015, 04:49 PM Central Europe Daylight Time
Meaning: This document has changed to Authorized status
==========