January 8th, 2015 by Gareth Wilson (2 min read)

Stop More Bugs with our Code Review Checklist

In our blog about effective code reviews, we recommended the use of a checklist. Checklists are a great tool in code reviews – they ensure that reviews are consistently performed throughout your team. They're also a handy way to ensure that common issues are identified and resolved.

Research by the Software Engineering Institute suggests that programmers make 15-20 common mistakes. So by adding such mistakes to a checklist, you can make sure that you spot them whenever they occur and help drive them out over time.

To get you started with a checklist, here's a list of typical items:

Code Review Checklist

General

- Does the code work? Does it perform its intended function, the logic is correct etc.
- Is all the code easily understood?
- Does it conform to your agreed coding conventions? These will usually cover location of braces, variable and function names, line length, indentations, formatting, and comments.
- Is there any redundant or duplicate code?
- Is the code as modular as possible?
- Can any global variables be replaced?
- Is there any commented out code?
- Do loops have a set length and correct termination conditions?
- Can any of the code be replaced with library functions?
- Can any logging or debugging code be removed?

Security

- Are all data inputs checked (for the correct type, length, format, and range) and encoded?
- Where third-party utilities are used, are returning errors being caught?
- Are output values checked and encoded?
- Are invalid parameter values handled?

Documentation

- Do comments exist and describe the intent of the code?
- Are all functions commented?
- Is any unusual behavior or edge-case handling described?
- Is the use and function of third-party libraries documented?

- Are data structures and units of measurement explained?
- Is there any incomplete code? If so, should it be removed or flagged with a suitable marker like 'TODO'?

Testing

- Is the code testable? i.e. don't add too many or hide dependencies, unable to initialize objects, test frameworks can use methods etc.
- Do tests exist and are they comprehensive? i.e. has at least your agreed on code coverage.
- Do unit tests actually test that the code is performing the intended functionality?
- Are arrays checked for 'out-of-bound' errors?
- Could any test code be replaced with the use of an existing API?

You'll also want to add to this checklist any language-specific issues that can cause problems.

The checklist is deliberately not exhaustive of all issues that can arise. You don't want a checklist, which is so long no-one ever uses it. It's better to just cover the common issues.

Optimize Your Checklist

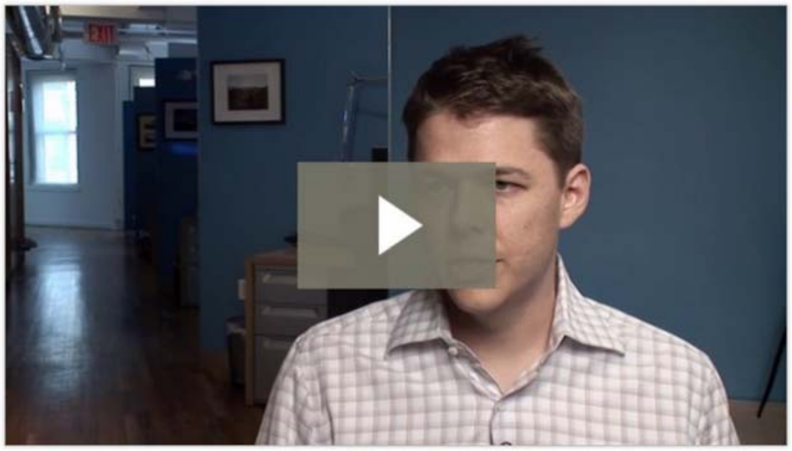
Using the checklist as a starting point, you should optimize it for your specific use-case. A great way to do this is to get your team to note the issues that arise during code reviews for a short time. With this data, you'll be able to identify your team's common mistakes, which you can then build into a custom checklist. Be sure to remove any items that don't come up (you may wish to keep rarely occurring, yet critical items such as security related issues).

Get Buy-in and Keep It Up To Date

As a general rule, any items on the checklist should be specific and, if possible, something you can make a binary decision about. This helps to avoid inconsistency in judgments. It is also a good idea to share the list with your team and get their agreement on its content. Make sure to review the checklist periodically too, to check that each item is still relevant.

Armed with a great checklist, you can raise the number of defects you detect during code reviews. This will help you to drive up coding standards and avoid inconsistent code review quality.

To learn more about code reviews at Fog Creek, check out the following video:



2

Generic Checklist for Code Reviews

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?

Documentation

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

Variables

- ❑ Are all variables properly defined with meaningful, consistent, and clear names?
- ❑ Do all assigned variables have proper type consistency or casting?
- ❑ Are there any redundant or unused variables?

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?

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?

Defensive Programming

- ❑ Are indexes, pointers, and subscripts tested against array, record, or file bounds?
- ❑ 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 deallocated?
- ❑ 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?

Code Review Checklist

http://commondatastorage.googleapis.com/bluelotussoftware/documents/Code%20Review%20Checklist.docx

Documentation

- ☐ All methods are commented in clear language. If it is unclear to the reader, it is unclear to the user.
- ☐ All source code contains @author for all authors.
- ☐ @version should be included as required.
- ☐ All class, variable, and method modifiers should be examined for correctness.
- ☐ Describe behavior for known input corner-cases.
- ☐ Complex algorithms should be explained with references. For example, document the reference that identifies the equation, formula, or pattern. In all cases, examine the algorithm and determine if it can be simplified.
- ☐ Code that depends on non-obvious behavior in external frameworks is documented with reference to external documentation.
- ☐ Confirm that the code does not depend on a bug in an external framework which may be fixed later, and result in an error condition. If you find a bug in an external library, open an issue, and document it in the code as necessary.
- ☐ Units of measurement are documented for numeric values.
- ☐ Incomplete code is marked with //TODO or //FIXME markers.
- ☐ All public and private APIs are examined for updates.

Testing

- ☐ Unit tests are added for each code path, and behavior. This can be facilitated by tools like Sonar, and Cobertura.
- ☐ Unit tests **must** cover error conditions and invalid parameter cases.
- ☐ Unit tests for standard algorithms should be examined against the standard for expected results.
- ☐ Check for possible null pointers are always checked before use.
- ☐ Array indices are always checked to avoid `ArrayIndexOfBounds` exceptions.
- ☐ Do not write a new algorithm for code that is already implemented in an existing public framework API, and tested.
- ☐ Ensure that the code fixes the issue, or implements the requirement, and that the unit test confirms it. If the unit test confirms a fix for issue, add the issue number to the documentation.

Error Handling

- ☐ Invalid parameter values are handled properly early in methods (Fast Fail).

- ☐ `NullPointerException` conditions from method invocations are checked.
- ☐ Consider using a general error handler to handle known error conditions.
- ☐ An Error handler **must** clean up state and resources no matter where an error occurs.
- ☐ Avoid using `RuntimeException`, or sub-classes to avoid making code changes to implement correct error handling.
- ☐ Define and create custom Exception sub-classes to match your specific exception conditions. Document the exception in detail with example conditions so the developer understands the conditions for the exception.
- ☐ (JDK 7+) Use try-with-resources. (JDK < 7) check to make sure resources are closed.
- ☐ **Don't pass the buck!** Don't create classes which throw Exception rather than dealing with exception condition.
- ☐ **Don't swallow exceptions!** For example `catch (Exception ignored) {}`. It should at least log the exception.

Thread Safety

- ☐ Global (static) variables are protected by locks, or locking sub-routines.
- ☐ Objects accessed by multiple threads are accessed only through a lock, or synchronized methods.
- ☐ Locks must be acquired and released in the right order to prevent deadlocks, even in error handling code.

Performance

- ☐ Objects are duplicated only when necessary. If you must duplicate objects, consider implementing Clone and decide if deep cloning is necessary.
- ☐ No busy-wait loops instead of proper thread synchronization methods. For example, avoid `while(true){ ... sleep(10);...}`
- ☐ Avoid large objects in memory, or using String to hold large documents which should be handled with better tools. For example, don't read a large XML document into a String, or DOM.
- ☐ Do not leave debugging code in production code.
- ☐ Avoid `System.out.println();` statements in code, or wrap them in a Boolean condition statement like `if(DEBUG) {...}`
- ☐ "Optimization that makes code harder to read should only be implemented if a profiler or other tool has indicated that the routine stands to gain from optimization. These kinds of optimizations should be well documented and code that performs the same task should be preserved."
  — **UNKNOWN**.

Code Review Checklist

Coding Standards
- understandable
- adhere code guidelines
- indentation
- no magic numbers
- naming
- units, bounds
- spacing: horizontal (btwn operators, keywords) and vertical (btwn methods, blocks)

Comments
- no needless comments
- no obsolete comments
- no redundant comments
- methods document parameters it modifies, functional dependencies
- comments consistent in format, length, level of detail
- no code commented out

Logic
- array indexes within bounds
- conditions correct in ifs, loops
- loops always terminate
- division by zero
- refactor statements in the loop to outside the loop

Error Handling
- error messages understandable and complete
- edge cases (null, 0, negative)
- parameters valid
- files, other input data valid

Code Decisions
- code at right level of abstraction
- methods have appropriate number, types of parameters
- no unnecessary features
- redundancy minimized
- mutability minimized
- static preferred over nonstatic
- appropriate accessibility (public, private, etc.)
- enums, not int constants
- defensive copies when needed
- no unnecessary new objects
- variables in lowest scope
- objects referred to by their interfaces, most generic supertype

Review Information:

Name of Reviewer: ______________________________

Name of Coder: ______________________________

File(s) under review: ______________________________

Brief description of change being reviewed: ______________________________

____

Review Notes (problems or decisions):

SVN Versions (if applicable):

Before review: __________

After revisions: __________