Code Review Checklist

Created by OpenMRS Wiki, last modified by Burke Mamlin on 2013-12-19

Things to look for during a code review

What not to do

- Design
  - Code review is a process to review existing code and learn from it, *not* the time or place to discuss design ideas. If design ideas come to mind, write them down separately and discuss them outside of the code review.
- Writing code
  - If you are writing any code, you are not reviewing. The only exception would be a short line of two of code within a comment to help communicate a defect or suggested fix.
- Judge
  - Code review is a chance for everyone to learn and make better code while improving the code for the community. We all make mistakes and we all have opportunities to learn. Defects in code are an opportunity to learn and improve for everyone.

Overview

1. Finding bugs
   - NPEs, no privilege checks, etc
2. Create common Coding style
3. Learning good coding style and best practices

Maintainability

1. Does the code make sense?
   - Make an effort to understand what the code is supposed to do before performing a code review.
   - Require the developer to comment as much as necessary to make the code readable.
2. Does the code comply with the accepted Java Conventions?
3. Does the code comply with the accepted Best Practices?
   - See Conventions
4. Does the code comply with the accepted Comment Conventions?
   - All classes and methods should contain a descriptive JavaDoc comment.
   - All methods should contain brief comments describing unobvious code fragments.
   - All class files should contain a copyright header.
   - All class files should contain class comments, including author name.
   - All methods should contain comments that specify input parameters.
   - All methods should contain a comment that specifies possible return values.
   - Complex algorithms should be thoroughly commented.
   - Comment all variables that are not self-describing.
   - Static variables should describe why they are declared static.
   - Code that has been optimized or modified to "work around" an issue should be thoroughly commented, so as to avoid confusion and re-introduction of bugs.
   - Code that has been "commented out" should be explained or removed.
   - Code that needs to be reworked should have a TODO comment and a clear explanation of what needs to be done.
   - When in doubt, comment.
   - When you've commented too much, keep commenting.
   - When your wrists hurt from commenting too much, take a break ... and then comment more.

Error Handling

1. Does the code comply with the accepted Exception Handling Conventions.
   - We need to expand our notion of Exception Handling Conventions.
   - Some method in the call stack needs to handle the exception, so that we don't display that exception stacktrace to the end user.
2. Does the code make use of exception handling?
   - Exception handling should be consistent throughout the system.
3. Does the code simply catch exceptions and log them?
   - Code should handle exceptions, not just log them.

4. Does the code catch general exception (java.lang.Exception)?
    - Catching general exceptions is commonly regarded as "bad practice".
5. Does the code correctly impose conditions for "expected" values?
    - For instance, if a method returns null, does the code check for null?
        - The following code should check for null

```
Person person = Context.getPersonService().getPerson(personId);
person.getAddress().getStreet();
```

1. - - What should be our policy for detecting null references?
2. Does the code test all error conditions of a method call?
    - Make sure all possible values are tested.
    - Make sure the JUnit test covers all possible values.

Security

1. Does the code appear to pose a security concern?
    - Passwords should not be stored in the code. In fact, we have adopted a policy in which we store passwords in runtime properties files.
    - Connect to other systems securely, *i.e.*, use HTTPS instead of HTTP where possible.
2. Service methods should have an @Authorize annotation on them
3. JSP pages should use the openmrs:require taglib

Thread Safeness

1. Does the code practice thread safeness?
    - If objects can be accessed by multiple threads at one time, code altering global variables (static variables) should be enclosed using a synchronization mechanism (synchronized).
    - In general, controllers / servlets should not use static variables.
    - Use synchronization on the smallest unit of code possible. Using synchronization can cause a huge performance penalty, so you should limit its scope by synchronizing only the code that needs to be thread safe.
    - Write access to static variable should be synchronized, but not read access.
    - Even if servlets/controllers are thread-safe, multiple threads can access HttpSession attributes at the same time, so be careful when writing to the session.
    - Use the volatile keyword to warn that compiler that threads may change an instance or class variable - tells compiler not to cache values in register.
    - Release locks in the order they were obtained to avoid deadlock scenarios.
2. Does the code avoid deadlocks?
    - I'm not entirely sure how to detect a deadlock, but we need to make sure we acquire/release locks in a manner that does not cause contention between threads. For instance, if Thread A acquires Lock #1, then Lock #2, then Thread B should not acquire Lock #2, then Lock #1.
    - Avoid calling synchronized methods within synchronized methods.

Resource Leaks

1. Does the code release resources?
    - Close files, database connections, HTTP connections, etc.
2. Does the code release resources more than once?
    - This will sometimes cause an exception to be thrown.
3. Does the code use the most efficient class when dealing with certain resources?
    - For instance, buffered input / output classes.

Control Structures

1. Does the code make use of infinite loops?
    - If so, please be sure that the end condition CAN and WILL be met.
2. Does the loop iterate the correct number of times?
    - Check initialization and end condition to make sure that the loop will be executed the correct number of times.

Reusability

1. Are all available libraries being used effectively?
2. Are available openmrs util methods known and used?
3. Is the code as generalized/abstracted as it could be?
4. Is the code a candidate for reusability?
    - If you see the same code being written more than once (or if you have copied-and-pasted code from another class), then this code is a candidate.