**Introduction to Code Reviews**

Software code review is a process to ensure that the code meets the functional requirements and also helps the developers to adhere to the best coding practices. Additionally, code review process helps in improving the software quality.

This code review checklist also helps the code reviewers and software developers (during self code review) to gain expertise in the code review process, as these points are easy to remember and follow during the code review process.

### Basic Code Review Checklist

Let’s discuss about the basic code review checklist, which can be very handy if you are a beginner in code reviews and/or during initial code reviews.

While reviewing the code, ask yourself the following basic questions:

1. Am I able to **understand**the code easily?
2. Is the code written following the **coding standards/guidelines**?
3. Is the same code **duplicated**more than twice? (**DRY**)
4. Can I **unit test / debug**the code easily to find the root cause?
5. Is this function or class **too big**? If yes, is the function or class having too many  
   responsibilities?

If you feel that the answer is not satisfactory to any of the above questions, then you can suggest/recommend code changes.

### Detailed Code Review Checklist

The following code review checklist gives an idea about the various aspects you need to consider while reviewing the code:

#### **1. Code formatting**

While going through the code, check the code formatting to improve readability and ensure that there are no blockers:

1. Use alignments (left margin), proper white space. Also ensure that code block starting point and ending point are easily identifiable.
2. Ensure that proper naming conventions (Pascal, CamelCase etc.) have been followed.
3. Code should fit in the standard 14 inch laptop screen.  There shouldn’t be a need to scroll horizontally to view the code. In a 21 inch monitor, other windows (toolbox, properties etc.) can be opened while modifying code, so always write code keeping in view a 14 inch monitor.
4. Remove the commented code as this is always a blocker, while going through the code. Commented code can be obtained from Source Control (like SVN), if required.

#### **2. Architecture**

1. The code should follow the defined architecture.
2. Separation of Concerns followed
3. Split into multiple layers and tiers as per requirements (Presentation,  
   Business and Data layers).
4. Split into respective files (HTML, JavaScript and CSS).
5. Code is in sync with existing code patterns/technologies.
6. Design patterns: Use appropriate design pattern (if it helps), after completely understanding the problem and context.

#### **3. Coding best practices**

1. No hard coding, use constants/configuration values.
2. Group similar values under an [enumeration](https://en.wikipedia.org/wiki/Enumerated_type) (enum).
3. Comments – Do not write comments for what you are doing, instead write comments on why you are doing. Specify about any hacks, workaround and temporary fixes. Additionally, mention pending tasks in your to-do comments, which can be tracked easily.
4. Avoid multiple if/else blocks.
5. Use framework features, wherever possible instead of writing custom code.

#### **4. Non Functional requirements**

1. **Maintainability (Supportability)** – The application should require the least amount of effort to support in near future. It should be easy to identify and fix a defect.
2. **Readability:** Code should be self-explanatory. Get a feel of story reading, while going through the code. Use appropriate name for variables, functions and classes. If you are taking more time to understand the code, then either code needs refactoring or at least comments have to be written to make it clear.
3. **Testability:** The code should be easy to test. Refactor into a separate function (if required). Use interfaces while talking to other layers, as interfaces can be mocked easily. Try to avoid static functions, singleton classes as these are not easily testable by mocks.
4. **Debuggability:** Provide support to log the flow of control, parameter data and exception details to find the root cause easily. If you are using [Log4Net](https://www.nuget.org/packages/log4net/) like component then add support for database logging also, as querying the log table is easy.
5. **Configurability:**Keep the configurable values in place (XML file, database table) so that no code changes are required, if the data is changed frequently.
6. **Reusability**
7. DRY (Do not Repeat Yourself) principle: The same code should not be repeated more than twice.
8. Consider reusable services, functions and components.
9. Consider generic functions and classes.
10. **Reliability –**Exception handling and cleanup (dispose) resources.
11. **Extensibility –**Easy to add enhancements with minimal changes to the existing code. One component should be easily replaceable by a better component.
12. **Security –**Authentication, authorization, input data validation against security threats such as [SQL injections](http://www.w3schools.com/sql/sql_injection.asp) and [Cross Site Scripting](https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)) (XSS), encrypting the sensitive data (passwords, credit card information etc.)
13. **Performance**
14. Use a data type that best suits the needs such as StringBuilder, generic collection classes.
15. Lazy loading, asynchronous and parallel processing.
16. Caching and session/application data.
17. **Scalability -** Consider if it supports a large user base/data? Can this be deployed into web farms?
18. **Usability –**Put yourself in the shoes of an end-user and ascertain, if the user interface/API is easy to understand and use. If you are not convinced with the user interface design, then start discussing your ideas with the business analyst.

#### **5. Object-Oriented Analysis and Design (OOAD) Principles**

1. **Single Responsibility Principle (SRS):** Do not place more than one responsibility into a single class or function, refactor into separate classes and functions.
2. **Open Closed Principle:** While adding new functionality, existing code should not be modified. New functionality should be written in new classes and functions.
3. [**Liskov substitutability principle**](https://en.wikipedia.org/wiki/Liskov_substitution_principle)**:** The child class should not change the behavior (meaning) of the parent class. The child class can be used as a substitute for a base class.
4. **Interface segregation:** Do not create lengthy interfaces, instead split them into smaller interfaces based on the functionality. The interface should not contain any dependencies (parameters), which are not required for the expected functionality.
5. **Dependency Injection:** Do not hardcode the dependencies, instead inject them.

In most cases the principles are interrelated, following one principle automatically satisfies other principles. For e.g: if the ‘Single Responsibility Principle’ is followed, then Reusability and Testability will automatically increase.

In a few cases, one requirement may contradict with other requirement. So need to trade-off based on the importance of the weight-age, e.g. Performance vs Security. Too many checks and logging at multiple layers (UI, Middle tier, Database) would decrease the performance of an application. But few applications, especially relating to finance and banking require multiple checks, audit logging etc. So it is ok to compromise a little on performance to provide enhanced security.

<http://www.evoketechnologies.com/blog/code-review-checklist-perform-effective-code-reviews/>

You also learn a lot from peer code reviews. What has been written well? Why was it done this way? Could this have been written differently?, etc. This is one of the benefits of volunteering to review code via open-source project contribution.

|  |  |
| --- | --- |
| **Initial** | **Concept** |
| **S** | [**Single responsibility principle**](https://en.wikipedia.org/wiki/Single_responsibility_principle)[[4]](https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29#cite_note-4)  a [class](https://en.wikipedia.org/wiki/Class_(computer_science)) should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class) |
| **O** | [**Open/closed principle**](https://en.wikipedia.org/wiki/Open/closed_principle)[[5]](https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29#cite_note-5)  “software entities … should be open for extension, but closed for modification.” |
| **L** | [**Liskov substitution principle**](https://en.wikipedia.org/wiki/Liskov_substitution_principle)[[6]](https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29#cite_note-6)  “objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.” See also [design by contract](https://en.wikipedia.org/wiki/Design_by_contract). |
| **I** | [**Interface segregation principle**](https://en.wikipedia.org/wiki/Interface_segregation_principle)[[7]](https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29#cite_note-7)  “many client-specific interfaces are better than one general-purpose interface.”[[8]](https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29#cite_note-martin-design-principles-8) |
| **D** | [**Dependency inversion principle**](https://en.wikipedia.org/wiki/Dependency_inversion_principle)[[9]](https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29#cite_note-9)  one should “depend upon abstractions, [not] concretions.”[[8]](https://en.wikipedia.org/wiki/SOLID_%28object-oriented_design%29#cite_note-martin-design-principles-8) |

**Functionality**

|  |  |
| --- | --- |
| **Checklist** | **Description/example** |
| Functionality is implemented in a simple, maintainable, and reusable manner. | Keep in mind some of the design principles like  **SOLID** design principles, Don’t Repeat Yourself (**DRY**), and Keep It Simple Stupid (**KISS**).  Also, think about the **OO** concepts — **A PIE**.   Abstraction, Polymorphism, Inheritance, and Encapsulation. These principles and concepts are all about accomplishing “**Low coupling**” and “**High cohesion**“.  Apply functional programming (**FP**) paradigm where it makes more sense. |

**Clean code**

|  |  |
| --- | --- |
| **Checklist** | **Description/example** |
| Use of descriptive and meaningful variable, method and class names as opposed to relying too much on comments. | **E.g.** calculateGst(BigDecimal amount), BalanceLoader.java, etc.  **Bad**: List list;  **Good**: List<String> users; |
| Class and functions should be small and focus on doing one thing. No duplication of code. | **E.g.** **CustomerDao.java** for data access logic only, **Customer.java** for domain object, **CustomerService.java** for business logic, and **CustomerValidator.java** for validating input fields, etc.  Similarly, separate functions like processSalary(String customerCode) will invoke other sub functions with meaningful names like  evaluateBonus(String customerCode),  evaluateLeaveLoading(String customerCode), etc |
| Functions should not take too many input parameters. | **Bad**: processOrder(String customerCode, String customerName, String deliveryAddress, BigDecimal unitPrice, int quantity, BigDecimal discountPercentage);  **Good**: processOrder(CustomerDetail customer, OrderDetail order);  where CustomerDetail is a value object with attributes like customerCode, customerName, etc. |
| Use a standard code formatting template. | Share the template across the development team. |
| Declare the variables with the smallest possible scope. | For example, if a variable “tmp” is used only inside a loop, then declare it inside the loop, and not outside. |
| Don’t preserve or create variables that you don’t use again. | **E.g.** **instead of** boolean removed = myItems.remove(item); return removed;  **Do:**  return myItems.remove(item); |
| Omit needless and commented out code. No System.out.println statements either. | You have source control for the history. Use proper logging frameworks like slf4j and logback for logging. |

**Fundamentals**

|  |  |
| --- | --- |
| **Checklist** | **Description/example** |
| Make a class final and the object immutable where possible. | Immutable classes are inherently thread-safe and more secured. For example, the Java String class is immutable and declared as final. |
| Minimize the accessibility of the packages, classes and its members like methods and variables. | **E.g.** private, protected, default, and public access modifiers. |
| Code to interface as opposed to implementation. | **Bad**: ArrayList<String> names = new ArrayList<String>();  **Good**: List<String> names = new ArrayList<String>(); |
| Use right data types. | For example, use **BigDecimal** instead of floating point variables like float or double for monetary values. Use **enums** instead of int constants. |
| Avoid finalizers and properly override equals, hashCode, and toString methods. | The equals and hashCode contract must be correctly implemented to prevent hard to debug defects. |
| Write fail-fast code by validating the input parameters. | Apply design by contract. |
| Return an empty collection or throw an exception as opposed to returning a null. Also, be aware of the implicit autoboxing and unboxing gotchas. | **NullpointerException** is one of the most common exceptions in Java. |

**Key Areas like Security, Exception Handling, Performance, Memory/Resource leaks, Concurrency, etc.**

|  |  |
| --- | --- |
| **Checklist** | **Description/example** |
| Don’t log sensitive data. | **Security**. |
| Clearly document security related information. | **Security**. |
| Sanitize user inputs. | **Security**. |
| Favor immutable objects. | **Security**. |
| Use Prepared statements as opposed to ordinary statements. | **Security** to prevent SQL injection attack. |
| Release resources (Streams, Connections, etc). | **Security** to prevent denial of service attack (DoS) and **resource leak**issues. |
| Don’t let sensitive information like file paths, server names, host names, etc escape via exceptions. | **Security** and **Exception Handling**. |
| Follow proper security best practices like SSL (one-way, two-way, etc), encrypting sensitive data, authentication/authorization, etc. | **Security.** |
| Use exceptions as opposed to return codes. | **Exception Handling.** |
| Don’t ignore or suppress exceptions. Standardize the use of checked and unchecked exceptions. Throw exceptions early and catch them late. | **Exception Handling.** |
| Write thread-safe code with proper synchronization and use of immutable objects. Also, document thread-safety. | **Concurrency.** |
| Keep synchronization section small and favor the use of the new concurrency libraries to prevent excessive synchronization. | **Concurrency** and **Performance**. |
| Reuse objects via flyweight design pattern. | **Performance**. |
| Presence of long lived objects like ThreaLocal and static variables holding references to lots of short lived objects. | **Memory Leak** and **Performance** |
| Badly constructed SQL, REGEX, etc. | **Performance**. E.g. Cartesian joins in SQL and back tracking regular expressions. |
| Inefficient Java coding and algorithms in frequently executed methods leading to death by thousand cuts. | **Performance** |

**Other general programming**

|  |  |
| --- | --- |
| **Checklist** | **Description/example** |
| Favor using well proven frameworks and libraries as opposed to reinventing the wheel by writing your own. | **E.g.** Apache commons libraries, Google Gauva libraries, Spring libraries, XML/JSON libraries, etc. |
| Presence of JUnit and JBehave test cases. | Check the test coverage and quality of the unit tests with proper mock objects to be able to easily maintain and run independently/repeatedly.   * Test only a unit of code at a time (e.g. one function). * Unit tests must be independent of each other. They should run independendtly. * Set up should not be too complicated. * Mockout external states and services that you are not asserting. For example, retrieving data from a database. * Avoid unneccessary assertions. * Start with functions that have the fewest dependencies, and work your way up. * Write unit tests for negative scenarios like throwing exceptions, negative values, null values, etc. * Don’t have try/catch inside unit tests. Use throws Exception statement in test case declaration itself. * Don’t have ant System.out.println(…..) |
| Ensure that the unit tests are written properly. | Don’t write unit tests for the sake of writing one. |
| Presence of hard coded config values. | Externalize configuration data in a .properties file. Sensitive information like password must be encrypted. |
| Presence and implementation of non functional requirements like archiving, auditing, and purging data and application monitoring where required. | It is easy to ignore these non functional requirements. |

<https://www.java-success.com/30-java-code-review-checklist-items/>