Code Review Principles, Processes and Tools

April 25

2014

Aditya Pratap Bhuyan

Code Review Details for Java

Table of Contents

1. Introduction, Purpose and Target Audience

Code review is a basic mechanism for validating the design and implementation of software. It also helps to maintain a level of consistency in design and implementation practices across projects and among the various modules in side the projects.

Code reviews can often find and remove common vulnerabilities such as format string exploits, race conditions, memory leaks and buffer overflows, thereby improving software security. Online software repositories based on Subversion (with Redmine or Trac), Mercurial, Git or others allow groups of individuals to collaboratively review code. Additionally, specific tools for collaborative code review can facilitate the code review process.

The document is intended to give a general idea about code review. It discusses the purpose, methods and tools used for code review. It doesn't cover reporting code review and code profiling.

This document is useful for software developers, Technical Leads, Architects and Project Managers to give them the idea of code review.

2. Need of Code Review

2.1. What is Code Review

Code review is systematic examination (often as peer review) of computer source code. It is intended to find and fix mistakes overlooked in the initial development phase, improving both the overall quality of software and the developers' skills. Reviews are done in various forms such as pair programming, informal walkthroughs, and formal inspections.

2.2. Need of Code Review

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.

3. Ways and Type of Code Review

3.1. Types of Code Review

Code review practices fall into three main categories: pair programming, formal code review and lightweight code review.

Formal code review, such as a Fagan inspection, involves a careful and detailed process with multiple participants and multiple phases. Formal code reviews are the traditional method of review, in which software developers attend a series of meetings and review code line by line, usually using printed copies of the material. Formal inspections are extremely thorough and have been proven effective at finding defects in the code under review.

Lightweight code review typically requires less overhead than formal code inspections, though it can be equally effective when done properly. Lightweight reviews are often conducted as part of the normal development process.

a. **Over-the-shoulder** – One developer looks over the author's shoulder as the latter walks through the code.
b. **Email pass-around** – Source code management system emails code to reviewers automatically after checkin is made.
c. **Pair Programming** – Two authors develop code together at the same workstation, such is common in Extreme Programming.
d. **Tool-assisted code review** – Authors and reviewers use specialized tools designed for peer code review.

Some of these may also be labeled a "Walkthrough" (informal) or "Critique" (fast and informal).

Many teams that eschew traditional, formal code review use one of the above forms of lightweight review as part of their normal development process. The lightweight reviews uncoveres as many bugs as formal reviews, but were faster and more cost-effective.

3.2. Scope of Code Review

These are the following areas which are covered during code review.

a. **Goal review:** is the issue being fixed actually a bug? Does the patch fix the fundamental problem?
b. **API/design review:** Because APIs define the interactions between modules, they need special care. Review is especially important to keep APIs balanced and targeted, and not too specific or over designed. There are also specific API change rules that must be followed.
c. **Maintainability review:** Code which is unreadable is impossible to maintain. If the reviewer has to ask questions about the

purpose of a piece of code, then it is probably not documented well enough. Does the code follow the coding style guide?

d. **Security review:** Does the design use security concepts such as input sanitizers, wrappers, and other techniques? Does this code need additional security testing such as fuzz-testing or static analysis?
e. **Integration review**: Does this code work properly with other modules? Is it localized properly? Does it have server dependencies? Does it have user documentation?
f. **Testing review:** Are there tests for correct function? Are there tests for error conditions and incorrect inputs which could happen during operation?
g. **License review:** Does the code follow the code licensing rules?

3.3. Automated Code Review

Automated code review software checks source code for compliance with a predefined set of rules or best practices. The use of analytical methods to inspect and review source code to detect bugs has been a standard development practice. This process can be accomplished both manually and in an automated fashion. With automation, software tools provide assistance with the code review and inspection process. The review program or tool typically displays a list of warnings (violations of programming standards). A review program can also provide an automated or a programmer-assisted way to correct the issues found.

Some static code analysis tools can be used to assist with automated code review. They compare favorably to manual reviews, but they can be done faster and more efficiently. These tools also encapsulate deep knowledge of underlying rules and semantics required to perform this type analysis such that it does not require the human code reviewer to have the same level of expertise as an expert human auditor. Many Integrated Development Environments also provide basic automated code review functionality. For example the Eclipse and Microsoft Visual Studio IDEs support a variety of plugins that facilitate code review.

Next to static code analysis tools, there are also tools that analyze and visualize software structures and help humans to better understand these. Such systems are geared more to analysis because they typically do not contain a predefined set of rules to check software against. Some of these tools (e.g. SonarJ, Sotoarc, Structure101) allow to define target architectures and enforce that target architecture constraints are not violated by the actual software implementation.

3.4. Steps of 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

6. Check the style and correctness of each of the following:
   (a) Each file
   (b) Each class
   (c) Each function/method

7. Check the handling of exceptions and errors

8. Check the user interaction

9. Look for common errors

10. Read the test plan. Verify that it checks the software limits.

3.5. Detailed Code Review

3.5.1. General Guideline

a. Comments in Javadoc format in the case of Java files.
b. In-line comment for each global or other important variable
c. Declarations of logically related variables grouped together.
d. For example: first input parameters, then output variables, then temporaries, with a comment and blank line separating the blocks.
e. Simple names for temporaries. For example: String s; int num;
f. Reuse temporaries. For example: use "s" instead of "s1...s2...s3".
g. Comment at the start of statement block such as loop or conditional
h. Comment to explain any unusual code
i. Proper choice and capitalisation of names

3.5.2. File Level

a. top-of-file comment present.
b. Date stamp and programmer's name present.

c. Copyright notice such as "Copyright (C) 2011 Syniverse Software Services India Ltd."
d. Find out from the project manager what is appropriate.
e. "Imports" and "uses" clauses -- check for cyclic dependencies (file A uses file B which uses file A)

3.5.3. Class Level

a. Class comment present
b. Class variables tally with design document
c. "public" variables to be used rarely -- mainly in the case of data-only classes
d. Any "static" variable must have an explanatory comment
e. Number of functions/class should not exceed 10 normally

3.5.4. Method/Function Level

a. Comment block with fields (purpose, arguments, return, errors) filled in
b. Any "static" variable must have an explanatory comment
c. Length should not exceed 30 lines normally
d. Function should normally have arguments, i.e., minimize direct use of global or class variables.

3.5.5. Exception/ Error Handling

a. 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.

b. If a function returns an error code, it must be checked and action taken as above.
c. Recovery action (exit, ignore or retry) must be justified
d. Output from the error/exception handling:
- a low-level, library function:
  log the error Should not popup a dialog, or write to stdout or exit
- a non-interactive daemon:
  log the error . 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

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

3.5.6. Common Coding Errors

a. no default case in a "switch" or nested "if-then-else-if."
b. no error code to handle "impossible" cases (which will occur sooner or later)
c. error return value of a system call or function is ignored
d. exception thrown by a function is not caught by the caller
e. bounds of fixed-size arrays or vectors exceeded
f. '<' instead of '<='

g. '>' instead of '>='
h. '==' instead of '!='
i. '=' instead of '=='
j. range of a data type such as char or int exceeded
k. type-casts that could cause errors
l. unnecessary use of language features. For example,
   Integer object instead of an int
   Vector instead of an array.
m. use of a variable instead of a constant
n. use of numbers in the code instead of defining symbolic constants
o. use of uninitiallised variables

4. Great Programming Mistakes

4.1. Mistakes in Exception Handling

1. Log and Throw

Example:

```
catch (NoSuchMethodException e) {
  LOG.error("Blah", e);
  throw e;
}
```

or

```
catch (NoSuchMethodException e) {
  LOG.error("Blah", e);
  throw new MyServiceException("Blah", e);
}
```

or

```
catch (NoSuchMethodException e) {
  e.printStackTrace();
  throw new MyServiceException("Blah", e);
}
```

All of the above examples are equally wrong. This is one of the most annoying error-handling antipatterns. Either log the exception, or throw it, but never do both. Logging and throwing results in multiple log messages for a single problem in the code, and makes life hell for the support engineer who is trying to dig through the logs.

2. Throwing Exception

Example:

```
public void foo() throws Exception {
```

This is just sloppy, and it completely defeats the purpose of using a checked exception. It tells your callers "something can go wrong in my method." Real useful. Don't do this. Declare the specific checked exceptions that your method can throw. If there are several, you should probably wrap them in your own exception

3. Throwing the Kitchen Sink

Example:

```
public void foo() throws MyException,
    AnotherException, SomeOtherException,
    YetAnotherException
{
```

Throwing multiple checked exceptions from your method is fine, as long as there are different possible courses of action that the caller may want to take,

depending on which exception was thrown. If you have multiple checked exceptions that basically mean the same thing to the caller, wrap them in a single checked exception.

4. Catching Exception

Example:

```
try {
  foo();
} catch (Exception e) {
  LOG.error("Foo failed", e);
}
```

This is generally wrong and sloppy. Catch the specific exceptions that can be thrown. The problem with catching Exception is that if the method you are calling later adds a new checked exception to its method signature, the developer's intent is that you should handle the specific new exception. If your code just catches Exception (or worse, Throwable), you'll probably never know about the change and the fact that your code is now wrong.

5. Destructive Wrapping

Example:

```
catch (NoSuchMethodException e) {
  throw new MyServiceException("Blah: " +
      e.getMessage());
}
```

This destroys the stack trace of the original exception, and is always wrong.

6. Log and Return Null

Example:

```
catch (NoSuchMethodException e) {
  LOG.error("Blah", e);
  return null;
}
```

or

```
catch (NoSuchMethodException e) {
  e.printStackTrace();
  return null;
}  // Man I hate this one
```

Although not always incorrect, this is usually wrong. Instead of returning null, throw the exception, and let the caller deal with it. You should only return null in a normal (non-exceptional) use case (e.g., "This method returns null if the search string was not found.").

7. Catch and Ignore

Example:

```
catch (NoSuchMethodException e) {
  return null;
}
```

This one is insidious. Not only does it return null instead of handling or re-throwing the exception, it totally swallows the exception, losing the information forever.

8. Throw from Within Finally

- ✓ Example:

```
try {

  blah();

} finally {

  cleanUp();
```

}This is fine, as long as cleanUp() can never throw an exception. In the above example, if blah() throws an exception, and then in the finally block, cleanUp() throws an exception, that second exception will be thrown and the first exception will be lost forever. If the code that you call in a finally block can possibly throw an exception, make sure that you either handle it, or log it. Never let it bubble out of the finally block.

9. Unsupported Operation Returning Null

- ✓ Example:

```
public String foo() {
// Not supported in this implementation.
return null;
}
```

When you're implementing an abstract base class, and you're just providing hooks for subclasses to optionally override, this is fine. However, if this is not the case, you should throw an `UnsupportedOperationException` instead of returning null. This makes it much more obvious to the caller why things aren't working, instead of her having to figure out why her code is throwing some random NullPointerException.

10. Ignoring InterruptedException

- ✓ Example:

```
while (true) {
  try {
    Thread.sleep(100000);
  } catch (InterruptedException e) {}
  doSomethingCool();
}
```

InterruptedException is a clue to your code that it should stop whatever it's doing. Some common use cases for a thread getting interrupted are the active transaction timing out, or a thread pool getting shut down. Instead of ignoring the InterruptedException, your code should do its best to finish up what it's doing, and finish the current thread of execution. So to correct the example above:

```
while (true) {
        try {
                Thread.sleep(100000);
        } catch (InterruptedException e) {
    break;
}
doSomethingCool();
}
```

11. **Relying on getCause()**

- ✓
```
catch (MyException e) {
  if (e.getCause() instanceof FooException) {}
```

The problem with relying on the result of getCause is that it makes your code fragile. It may work fine today, but what happens when the code that you're calling into, or the code that it relies on, changes its underlying implementation, and ends up wrapping the ultimate cause inside of another exception? Now calling getCause may return you a wrapping exception, and what you really want is the result of getCause().getCause(). Instead, you should unwrap the causes until you find the ultimate cause of the problem. Apache's commons-lang project provides ExceptionUtils.getRootCause() to do this easily.

4.2. Mistakes in Common Java API

12. **Null Returning from a method with return type Collection or Array**
    - ✓ Null should never be returned from a method returning a collection or an array. Instead return a empty array (a static final empty array) or one of the empty collections (e.g. Collections.EMPTY_LIST assuming the client should not be modifying the collection).

13. **Unnecessary thread safety for StringBuffer**
    - ✓ Use StringBuilder rather then StringBuffer, unless synchronization is required. StringBuilder is not thread safe, and therefore avoids the synchronization overhead of StringBuffer.

14. **Comparing URLs with URL.equals()**
    - ✓ Implementation of equals() in java.net.URL is based on a fancy rule saying that URLs of two hosts are equal as long as they are

resolving to the same IP address. It is known to be incompatible with virtual hosting and should not be used.

15. **equals() and hashCode() are context-sensitive**
    - ✓ equals() and hashCode() implementations should rely only on an internal object state. Making them dependent on other objects, context of use and other external conditions conflicts with the general contracts of consistency:

16. **Not taking advantage of toString()**
    - ✓ Overriding toString() method gives you a cheap way to provide human-readable labels for objects in output. When the objects are used in GUI containers, such as JLists or JTables, it allows to use the default models and renderers instead of writing the custom ones.

17. **Instantiation of immutable objects**
    - ✓ Creating new instances of immutable primitive type wrappers (such as Number subclasses and Booleans) wastes the memory and time needed for allocating new objects. Static valueOf() method works much faster than a constructor and saves the memory, as it caches frequently used instances.

18. **Unbuffered IO**
    - ✓ Reading and writing I/O streams byte-by-byte is too expensive, as every read()/write() call refers to the underlying native (JNI) I/O subsystem. Usage of buffered streams reduces the number of native calls and improves I/O performance considerably.

19. **equals() doesn't check null arguments**
    - ✓ If you override equals() method in your class, always check if an argument is **null**. If a null value is passed, equals() must unconditionally return false (no NullPointerException should be thrown!).

20. **compareTo() is incompatible with equals()**
    - ✓ If a class implements Comparable, compareTo() method must return zero if and only if equals() returns true for the same non-null argument (and vice versa). Violating this rule may cause unexpected behavior of the objects.

21. **Linked List as arrays**
    - ✓ java.util.LinkedList is a special type of collection designed for sequential access (stacks and queues). Being used as a random-accesed array, it is much slower than other List implementations. For instance, getting an item by index (get(n)) has constant complexity O(1) for ArrayLists, while for LinkedList, the complexity of that operation is O(n).

22. **Synchronizes Collections Every Where**
    - ✓ Vector and Hashtable are just the synchronized versions of the ArrayList an HashMap but working much slower. Use unsynchronized collections unless thread-safety is really required.

23. **Accessing the Map values using KeySet Iterator**
    - ✓ A common mistake is to retrieve values from a Map while iterating over the Map keys with keySet(). Calling Map.get(key) for each entry is expensive and should be avoided for better performance.Use entrySet() iterator to avoid the Map.get(key) lookup.

24. **equals() is overridden where hashCode() is not**

- ✓ If you override equals() in your class, always provide a custom implementation of hashCode() that returns the same hash code value for two or more equal objects. This is, in fact, a general contract defined by Java API Specification. Violation of this rule (which is likely the case if equals() method is overridden while hashCode() is inherited from Object) may cause numerous bugs and unexpected behaviours.

4.3. Mistakes in Design

1. **Programming to Concrete Classes Rather than Interfaces**
   - ✓ This is an important design principle, and is often broken. Programming to an interface than a concrete class provides innumerable benefits. You will not be tied into using a specific implementation, and there will be a provision for changing behavior at runtime. The word "interface" implies either a Java interface or an abstract class. As long as you can apply polymorphism, the application's behavior isn't locked into specific code. Note that this principle is not applicable in situations when you know that the behavior is not going to change.
2. **Excessive Coupling**
   - ✓ When coding, one basic software concept to keep in mind is that less coupling is generally better. For example, if you code a method to perform a certain task, it should perform only that task, helping to keep the code clean, readable and maintainable. But inevitably, certain aspects such as logging, security, and caching tend to creep in. There are, fortunately, good ways to avoid this. One such technique, Aspect Oriented Programming (AOP), provides a neat way to achieve this by injecting the aspect's behavior into the application at compile time. For more info, refer to the DevX AOP articles in the related resources section of this article.

3. **Development: The Golden Hammer**
   - ✓ When coding, one basic software concept to keep in mind is that less coupling is generally better. For example, if you code a method to perform a certain task, it should perform only that task, helping to keep the code clean, readable and maintainable. But inevitably, certain aspects such as logging, security, and caching tend to creep in. There are, fortunately, good ways to avoid this. One such technique, Aspect Oriented Programming (AOP), provides a neat way to achieve this by injecting the aspect's behavior into the application at compile time.

4.4. Mistakes in Architecture

1. **Reinventing the Wheel**
   - ✓ This term doesn't need any description. When developing software, you usually have two choices—build on top of an existing technology or start from scratch. While both can be applicable in different situations, it is nevertheless useful to analyze existing technology before reinventing the wheel to develop functionality that already exists and can satisfy the requirements. This saves time, money as well as leveraging knowledge that developers already might have.
2. **Vendor lock-in**

- ✓ This occurs when the software is either partly or wholly dependent on a specific vendor,. One advantage of J2EE is its portability, but it still gives vendors the opportunity to provide rich proprietary features. Assuredly, such features can aid during development, but they can also have a reverse impact at times. One such problem is loss of control. You're probably familiar with the feeling that the feature you need is always six months away. Another such problem is when vendor changes to the product break your software, degrade interoperability, force constant upgrades, etc.
  One solution is to provide an isolation layer on top of the proprietary stuff. For example, Commons Logging allows you to plug in any logging framework such as Log4J or Java Logging.

3. *Anti Patterns in J2EE*

Layer	Antipattern
Persistence	Dredge Stifle
JSP	Too much session data Embedded Navigational Information
Servlet	Common Functionality in Every Servlet Accessing Fine-grained EJB Interfaces
EJBs	Large Transactions Overloading queues in JMS
Web services	Assuming SOA = Web service
J2EE	Hardcoded JNDI lookups Not leveraging EJB container

4.4. Mistakes in Servlets & JSP

1. **Overusing Session Data**
   - ✓ Using the JSP session as a common data space is an easy temptation to fall prey to. The session provides a simple mechanism for storing transferring data. But an increase in site traffic and the accompanying increase in session data can cause crashes. Also, a careless dump in the session might result in conflicting keys, resulting in application errors. An even worse situation would be an error that shared data across sessions when it is not meant to be shared, potentially exposing sensitive user information.

     Even without such technical issues, it is a bad idea to allow bloated sessions. Session data should be used only for information required for workflow across multiple user interfaces, and the session should be cleaned up when the information is no longer needed.
2. **Embedded Navigational Information**

- ✓ This occurs when developers hardcode or embed links to other JSPs. If the page names change, someone must search for and change all the referring links in other pages. Designing a carefully thought-out navigational scheme initially can prevent this maintenance nightmare at a later stage.

3. Common Functionality in Every Servlet
   - ✓ Hard-coding functionality that is common across multiple servlets in an application makes it hard to maintain. Instead, developers should remove the repeated code from the servlets. Using a framework such as Struts provides a neat way of specifying application flow in an XML file—which can be altered without altering the servlet code. Other common techniques exist as well, such as Front Controller and filters that you can use to remove hard-coded values at different levels.
4. Accessing Fine-grained EJB Interfaces
   - ✓ Marshaling and unmarshalling data involved in a remote call across the network can cause major time latency in applications. Latency can also occur due to multiple remote calls. Based on the needs on the application, the solution might involve redesigning the EJBs, but if that's not the case, servlets should not remotely access an EJB entity via a fine-grained API. If the EJB's expose both fine and coarse-grained APIs then servlets should use single calls to the coarse-grained API in preference to making multiple calls to the fine-grained methods.
5. Use Of Class Level Variables in Servlet
   - ✓ Servlets are single instances per application. For each incoming request, the servlet container creates a new thread. If class level variables would be used in Servlets, then it would be shared among all threads and all threads can alter the value. We should be careful when using class level variables in Servlet.

4.5. Mistakes in EJB

1. Large Transactions
   - ✓ Transactions that involve complicated processes that invoke multiple resources can lock out other threads waiting for the same resources for significant time periods. This can cause impact on performance. It's generally prudent to break such transactions into smaller chunks (depending on the application's needs), or to use alternatives such as stored procedures for lengthy database processes.
2. Overloading Queues in JMS
   - ✓ JMS provides both queue and topic destinations. A queue can house different kinds of messages, for example, map (binary) messages or text-based messages. But if you take advantage of this feature to send different kinds of messages on the same queue, it becomes the onus of the consumer to distinguish between them. A better solution would be to separate them out. You can accomplish that programmatically or by using two separate destinations, depending on the application requirements.

4.6. Mistakes in Web Services

1. Assuming SOA = Web Service
   - ✓ The term Service Oriented Architecture (SOA) is often confused with Web services, but the fact is that SOA concepts were around long before Web services were born. SOA strives for

loose coupling between components. It provides a software service that is consumed by another software service. Lately however, SOA has become synonymous with Web services. Just keep in mind that it's is entirely possible to base SOA on other service technologies as well.

4.7. Mistakes in Other J2EE Design

1. **Hard-coding JNDI Lookups**
   - ✓ The Java Naming and Directory Interface provides a convenient way to look up object information, such as the location of an EJB interface. But JNDI lookup operations are generally expensive, especially if performed repeatedly. If you cache the results though, you can obtain significant performance improvements. One such caching scheme is to use a singleton class.

     But networks aren't static. As the network changes, even the lookups can change. This means that for each network change, a programmer needs to revisit the application code, make any required changes, and recompile/redeploy the code. An alternative would be to provide configurable lookups via an XML configuration file so that developers don't have to recompile the code every time a network change occurs.
2. **Failing to Leverage EJB Container Features**
   - ✓ When developing enterprise components, there are two alternatives—use the services the container provides or write your own services. Although there are numerous alternatives such as Spring, EJBs are still used widely in conjunction with these frameworks. When using EJBs, you should attempt to use the container's services, such as clustering, load balancing, security, transactions management, fault tolerance, and data storage. Failing to leverage the rich features of the container where possible will eventually result in reinventing the wheel (an architecture antipattern mentioned earlier in this article).

     I'm sure you've realized the significance of antipatterns in juxtaposition with design patterns. Even if you weren't already aware of the names of some of the antipatterns described in this article, you should be able to recognize their features and the problems they can cause. Classifying and naming these antipatterns provides the same benefits as classifying and naming design patterns;doing so gives software managers, architects, designers, and developers a common vocabulary and helps them recognize possible sources of error or maintenance headaches in advance.

5. Static Code Review Tools

Static program analysis is the analysis of computer software that is performed without actually executing programs built from that software (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code. The term is usually applied to the analysis performed by an automated tool, with human analysis being called program understanding, program comprehension or code review.

The sophistication of the analysis performed by tools varies from those that only consider the behavior of individual statements and declarations, to those that include the complete source code of a program in their analysis. Uses of the information obtained from the analysis vary from highlighting possible coding errors (e.g., the lint tool) to formal methods that mathematically prove properties about a given program (e.g., its behavior matches that of its specification).

There are a lot of tools available for static code analysis. These tools are open source and commercial. Few tools are generic and few of them are language dependent. Many tools can be added as plugin to standard IDEs like eclipse.

5.1. Open Source Products

Many groups and communities are contributing with a number of static code analyzers in open source platform.

5.1.1. Multi language Support

a) **Copy/Paste Detector (CPD)** — PMDs duplicate code detection for (e.g.) Java, JSP, C, C++ and PHP code.
b) **Sonar** — A continuous inspection engine to manage the technical debt (unit tests, complexity, duplication, design, comments, coding standards and potential problems). Supported languages are Java, Flex, PHP, PL/SQL, Cobol and Visual Basic 6.
c) **Yasca** — Yet Another Source Code Analyzer, a plugin-based framework for scanning arbitrary file types, with plugins for scanning C/C++, Java, JavaScript, ASP, PHP, HTML/CSS, ColdFusion, COBOL, and other file types. It integrates with other scanners, including FindBugs, JLint, PMD, and Pixy.

5.1.2. Language Specific

.NET (C#, VB.NET and all .NET compatible languages)

a) **FxCop** — Free static analysis for Microsoft .NET programs that compile to CIL. Standalone and integrated in some Microsoft Visual Studio editions. From Microsoft.
b) **Gendarme** — Open-source (MIT License) equivalent to FxCop created by the Mono project. Extensible rule-based tool to find problems in .NET applications and libraries, particularly those that contain code in ECMA CIL format.

c) **StyleCop** — Analyzes C# source code to enforce a set of style and consistency rules. It can be run from inside of Microsoft Visual Studio or integrated into an MSBuild project. Free download from Microsoft.

ActionScript

a) **Apparat** — A language manipulation and optimization framework consisting of intermediate representations for ActionScript.

C

a) **Antic** — C and CPP analyzer, can detect errors such as division by 0 and array index bounds. A part of JLint, but can be used as standalone.
b) **BLAST** (Berkeley Lazy Abstraction Software verification Tool) — A software model checker for C programs based on lazy abstraction.
c) **Clang** — A compiler that includes a static analyzer.
d) **Frama-C** — A static analysis framework for C.
e) **Lint** — The original static code analyzer for C.
f) **Sparse** — A tool designed to find faults in the Linux kernel.
g) **Splint** — An open source evolved version of Lint (for C).

C++

a) **cppcheck** — Open-source tool that checks for several types of errors, including the use of STL.

Java

a) **Checkstyle** — Besides some static code analysis, it can be used to show violations of a configured coding standard.
b) **FindBugs** — An open-source static bytecode analyzer for Java (based on Jakarta BCEL) from the University of Maryland.
c) **Hammurapi** — (Free for non-commercial use only) versatile code review solution.
d) **PMD** — A static ruleset based Java source code analyzer that identifies potential problems.
e) **Soot** — A language manipulation and optimization framework consisting of intermediate languages for Java.

f) **Squale** — A platform to manage software quality (also available for other languages, using commercial analysis tools though).

JavaScript

a) **Closure Compiler** — JavaScript optimizer that rewrites JavaScript code to make it faster and more compact. It also checks your usage of native javascript functions.
b) **JSLint** — JavaScript syntax checker and validator.

Objective-C

a) **Clang** — The free Clang project includes a static analyzer. As of version 3.2, this analyzer is included in Xcode.

5.2. Commercial Products

5.2.1. Multi language Support

a) **Axivion Bauhaus Suite** — A tool for C, C++, C#, Java and Ada code that comprises various analyses such as architecture checking, interface analyses, and clone detection.
b) **Black Duck Suite** — Analyze the composition of software source code and binary files, search for reusable code, manage open source and third-party code approval, honor the legal obligations associated with mixed-origin code, and monitor related security vulnerabilities.
c) **CAST Application Intelligence Platform** — Detailed, audience-specific dashboards to measure quality and productivity. 30+ languages, SAP, Oracle, PeopleSoft, Siebel, .NET, Java, C/C++, Struts, Spring, Hibernate and all major databases.
d) **Coverity Static Analysis (formerly Coverity Prevent)** — Identifies security vulnerabilities and code defects in C, C++, C# and Java code. Complements Coverity Dynamic Code Analysis and Architecture Analysis.
e) **DMS Software Reengineering Toolkit** — Supports custom analysis of C, C++, C#, Java, COBOL, PHP, VisualBasic and many other languages. Also COTS tools for clone analysis, dead code analysis, and style checking.
f) **Compuware DevEnterprise** — Analysis of COBOL, PL/I, JCL, CICS, DB2, IMS and others.
g) **Fortify** — Helps developers identify software security vulnerabilities in C/C++, .NET, Java, JSP, ASP.NET, ColdFusion, "Classic" ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL, python and COBOL as well as configuration files.
h) **GrammaTech CodeSonar** — Analyzes C,C++.
i) **Imagix 4D** — Identifies problems in variable usage, task interaction and concurrency, particularly in embedded

applications, as part of an overall solution for understanding, improving and documenting C, C++ and Java software.

j) **Intel** - Intel Parallel Studio XE: Contains Static Security Analysis (SSA) feature supports C/C++ and Fortran

k) **JustCode** — Code analysis and refactoring productivity tool for JavaScript, C#, Visual Basic.NET, and ASP.NET

l) **Klocwork Insight** — Provides security vulnerability and defect detection as well as architectural and build-over-build trend analysis for C, C++, C# and Java.

m) **Lattix, Inc. LDM** — Architecture and dependency analysis tool for Ada, C/C++, Java, .NET software systems.

n) **LDRA Testbed** — A software analysis and testing tool suite for C, C++, Ada83, Ada95 and Assembler (Intel, Freescale, Texas Instruments).

o) **Logiscope** — Logiscope is a software quality assurance tool that automates code reviews and the identification and detection of error-prone modules for software testing.

p) **Micro Focus** (formerly Relativity Technologies) Modernization Workbench — Parsers included for COBOL (multiple variants including IBM, Unisys, MF, ICL, Tandem), PL/I, Natural (inc. ADABAS), Java, Visual Basic, RPG, C & C++ and other legacy languages; Extensible SDK to support 3rd party parsers. Supports automated Metrics (including Function Points), Business Rule Mining, Componentisation and SOA Analysis. Rich ad hoc diagramming, AST search & reporting)

q) **Ounce Labs** (from 2010 IBM Rational Appscan Source) — Automated source code analysis that enables organizations to identify and eliminate software security vulnerabilities in languages including Java, JSP, C/C++, C#, ASP.NET and VB.Net.

r) **Parasoft** — Analyzes Java (Jtest), JSP, C, C++ (C++test), .NET (C#, ASP.NET, VB.NET, etc.) using .TEST, WSDL, XML, HTML, CSS, JavaScript, VBScript/ASP, and configuration files for security[3], compliance[4], and defect prevention.

s) **Polyspace** — Uses abstract interpretation to detect and prove the absence of certain run-time errors in source code for C, C++, and Ada

t) **ProjectCodeMeter** — Warns on code quality issues such as insufficient commenting or complex code structure. Counts code metrics, gives cost & time estimations. Analyzes C, C++, C#, J#, Java, PHP, Objective C, JavaScript, UnrealEngine script, ActionScript, DigitalMars D.

u) **Rational Software Analyzer** — Supports Java, C/C++ (and others available through extensions)

v) **ResourceMiner** — Architecture down to details multipurpose analysis and metrics, develop own rules for masschange and generator development. Supports 30+ legacy and modern languages and all major databases.

w) **SofCheck Inspector** — Provides static detection of logic errors, race conditions, and redundant code for Java and Ada. Provides automated extraction of pre/postconditions from code itself.

x) **Sotoarc/Sotograph** — Architecture and quality in-depth analysis and monitoring for Java, C#, C and C++
y) **Syhunt Sandcat** — Detects security flaws in PHP, Classic ASP and ASP.NET web applications.
z) **Understand** — Analyzes C,C++, Java, Ada, Fortran, Jovial, Delphi, VHDL, HTML, CSS, PHP, and JavaScript — reverse engineering of source, code navigation, and metrics tool.
aa) **Veracode** — Finds security flaws in application binaries and bytecode without requiring source. Supported languages include C, C++, .NET (C#, C++/CLI, VB.NET, ASP.NET), Java, JSP, ColdFusion, and PHP.
bb) **Visual Studio Team System** — Analyzes C++,C# source codes. only available in team suite and development edition.

5.2.2. Language Specific

.NET

Products covering multiple .NET languages.

a) **CodeIt.Right** — Combines Static Code Analysis and automatic Refactoring to best practices which allows automatically correct code errors and violations. Supports both C# and VB.NET.
b) **CodeRush** — A plugin for Visual Studio, it addresses a multitude of short comings with the popular IDE. Including alerting users to violations of best practices by using static code analysis.
c) **JustCode** — Add-on for Visual Studio 2005/2008/2010 for real-time, solution-wide code analysis for C#, VB.NET, ASP.NET, XAML, JavaScript, HTML and multi-language solutions.
d) **NDepend** — Simplifies managing a complex .NET code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code. Integrates into Visual Studio.
e) **ReSharper** — Add-on for Visual Studio 2003/2005/2008/2010 from the creators of IntelliJ IDEA, which also provides static code analysis for C#.
f) **Kalistick** — Mixing from the Cloud: static code analysis with best practice tips and collaborative tools for Agile teams

Ada

a) **Ada-ASSURED** — A tool that offers coding style checks, standards enforcement and pretty printing features.
b) **AdaCore CodePeer** — Automated code review and bug finder for Ada programs that uses control-flow, data-flow, and other advanced static analysis techniques.
c) **LDRA Testbed** — A software analysis and testing tool suite for Ada83/95.

d) **SofCheck Inspector** — Provides static detection of logic errors, race conditions, and redundant code for Ada. Provides automated extraction of pre/postconditions from code itself.

C / C++

a) **FlexeLint** — A multiplatform version of PC-Lint.
b) **Green Hills Software DoubleCheck** — A software analysis tool for C/C++.
c) **Intel** - Intel Parallel Studio XE: Contains Static Security Analysis (SSA) feature
d) **LDRA Testbed** — A software analysis and testing tool suite for C/C++.
e) **Monoidics INFER** — A sound tool for C/C++ based on Separation Logic.
f) **PC-Lint** — A software analysis tool for C/C++.
g) **PVS-Studio** — A software analysis tool for C/C++/C++0x.
h) **QA-C (and QA-C++)** — Deep static analysis of C/C++ for quality assurance and guideline enforcement.
i) **Red Lizard's Goanna** — Static analysis for C/C++ in Eclipse and Visual Studio.
j) **CppDepend** — Simplifies managing a complex C\C++ code base by analyzing and visualizing code dependencies, by defining design rules, by doing impact analysis, and by comparing different versions of the code. Integrates into Visual Studio.

Java

a) **Jtest** — Testing and static code analysis product by Parasoft.
b) **LDRA Testbed** — A software analysis and testing tool suite for Java.
c) **SemmleCode** — Object oriented code queries for static program analysis.
d) **SonarJ** — Monitors conformance of code to intended architecture, also computes a wide range of software metrics.
e) **Kalistick** — A Cloud-based platform to manage and optimize code quality for Agile teams with DevOps spirit

6. References

The following references were taken while preparing the document.

1. http://www.en.wikipedia.org
2. https://developer.mozilla.org/en/Code_Review_FAQ