harborlabs
CYBER.SCIENCE

LITIGATION SUPPORT

Guidelines for Assessing Source Code Quality

Abstract

Harbor Labs is frequently engaged to conduct code quality reviews in support of federal investigations of healthcare IT systems, or to support litigation involving software coding practices. Regardless of the circumstances, Harbor Labs staff employs a common set of disciplines and analytic techniques to prove a position on the quality of the target codebase. While these analytic methods may be somewhat subjective, and varied based on the computer science skill set of the individual reviewer, there are nonetheless well known methods for determining code quality that are generally accepted throughout the computer science community. When combined with the expertise of an experienced code reviewer, these methods can be used to establish a defensible assertion of code quality that is based on published coding principles and broadly accepted industry best practices.

BACKGROUND

It is important to first to differentiate between an algorithm and its implementation. An algorithm is an ordered sequence of steps for performing a particular task. Algorithms can be analyzed, their properties studied, and their theoretical effectiveness measured. While algorithms are a high-level mathematical concept, the implementation of these constructs falls into the realm of software development. To be used in a software product, however, an algorithm must be implemented by a software developer. This means that a software developer must actually write code to perform the algorithm. Even if the algorithm itself is sound, the code that performs the algorithm may have errors that produce unreliable results.

The Goals of Source Code Testing

Programming errors are so common in computer science that entire systems of knowledge have been developed around training developers to produce fewer errors through use of proper development practices and methodologies for testing, detecting, and fixing errors. Professional organizations in the field of computer science such as the Institute of Electrical and Electronics Engineers (IEEE) and the Association for Computing Machinery (ACM), which, despite the historical origins of their names, are two

whitepaper

of the leading professional organizations in the field of software development, publish extensively on software reliability and testing.

A key component of any type of software *testing* as opposed to software *review* is that software testing requires the ability to run the software or components thereof; testing is an active process. It is a well-known fact in the software industry that it is common for code to have errors. Errors can affect reliability of outcome and the manner in which the system functions. An error in the code might still produce a result, but that result may not be reliable based on the type of the error. The vast majority of errors in software exist because software developers did not catch these errors when they wrote and reviewed their own code. Because the industry recognizes that it is important to produce code that is reliable and as error-free as possible, it is standard practice to test source code both as it is being developed and on an ongoing basis.

IEEE Source Code Testing Recommendations

The IEEE has published a widely referenced guide to software development best practices called the Software Engineering Body of Knowledge (SWEBOK). As the SWEBOK describes,

> "*In recent years, the view of software testing has matured into a constructive one. Testing is no longer seen as an activity that starts only after the coding phase is complete with the limited purpose of detecting failures. Software testing is, or should be, pervasive throughout the entire development and maintenance lifecycle.*"

The SWEBOK contains a detailed description of testing best practices.

> "*Indeed, planning for software testing should start with the early stages of the software requirements process, and test plans and procedures should be systematically and continuously developed—and possibly refined--as software development proceeds. These test planning and test designing activities provide useful input for software designers and help to highlight potential weaknesses, such as design oversights/contradictions, or omissions/ ambiguities in the documentation. … It is perhaps obvious but worth recognizing that software can still contain faults, even after completion of an extensive testing*

whitepaper

activity. Software failures experienced after delivery are addressed by corrective maintenance."

It is important to note that the IEEE recognizes the fact that despite a robust testing environment, software implementations can still contain bugs. Indeed, the SWEBOK and other similar guides such as the ISO Software Testing Standard 29119 describe a large set of possible testing methodologies and topics that all, at their heart, are designed to help identify programming errors in the implementation of software code.

The SWEBOK also contains a detailed diagram that provides a useful overview of all of the topics and subtopics that comprise the field of software testing.

whitepaper

In addition to testing, another method of identifying likely sources of errors in a software project is to perform a manual review of the source code. Source code reviews are extremely common in legal cases, in particular those involving breach of contract, faulty software due to improper development practices, and patent infringement, among others.

In IEEE Standard 1012[1], IEEE describes several types of source code testing including acceptance, component, and integration testing as described below.

Acceptance testing: Formal testing conducted to determine whether or not a system satisfies its acceptance criteria and to enable the customer to determine whether or not to accept the system (See ANSI/IEE Std 729-1983 [1].)

Component Testing: Testing conducted to verify the implementation of the design for one software element (for example, unit, module) or a collection of software elements.

Integration Testing: An orderly progression of testing in which software elements, hardware elements, or both are combined and tested until the entire system has been integrated. (See ANSI/IEEE Std 729-1983 [1].)

Each of these types of testing would be understood by a software developer to require running the software in order to be able to test it.

IEEE's report at Exhibit C also contends that:

For example, when source code is ordered to be provided, "information needed" requires providing sufficient information for the recipient to build, run, and test the software themselves, including at a minimum:

whitepaper

[1] Note that the version of IEEE Standard 1012 that I referenced was obtained from NIST directly at https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub132.pdf. I understand that newer versions of this standard can be purchased directly from IEEE.

- All software dependencies including third-party code libraries, toolboxes, plugins, frameworks, and databases;
- Software engineering and development materials describing the development, deployment, and maintenance of the version(s) of the software system used in the instant case, including software engineering documents and build instructions;
- All records of software glitches, crashes, bugs, or errors encountered during the developmental validation study;
- Software version numbers of the components of the system used for the developmental validation study, and:
- All records of unexpected results, including false inclusions, false exclusions and the conditions under which the unexpected results were achieved.

Length of Time to Review and Test Source Code

Although source code bases can be large (Harbor Labs has reviewed codebases with more than one million files), it is neither necessary nor possible to review each file individually. First, it is important to note that in large codebases many files are typically either autogenerated, boilerplate, or part of third-party dependencies that do not need to be separately reviewed. Further, it is generally the case that a source code review of a large codebase is looking to understand how specific functionality is implemented. As such, much of the code that is produced in a codebase is not directly relevant to the functionality being reviewed—this key functionality can often represent a tiny subset of the source code that was produced in a case. Even in the case that the amount of source code to review is large, there exist many popular tools to help a manual reviewer improve the efficiency of source code review. For instance, an incomplete list includes searching tools such as dtsearch,[2] which is commonly used to search for specific symbols such as variable and function names in the source code. Similarly, SciTools Understand[3] is commonly used to visualize and navigate codebases with which the reviewer is unfamiliar. It allows a reviewer to quickly jump from the use of a symbol to its definition, even when the definition and the use are in different source code files. It also makes tracing and cross-referencing function calls efficient. Harbor Labs staff has produced a companion whitepaper on source code reviews

whitepaper

[2] https://www.dtsearch.com

[3] https://www.scitools.com

in which a

variety of tools that are commonly used in code reviews are described.

The Goals of Source Code Review

Source code reviews can achieve several goals. In some cases, they can be used to understand how a particular algorithm or workflow works in a commercial product. Such code reviews are common in intellectual property litigation. Other source code reviews are designed to discover and catalog code quality issues that can lead to errors. This is common in code reviews related to breach of contract but can also be useful for code reviews about software reliability. There are many types of code quality issues that can be determined from a source code review including the following:

Dead and Commented Out Code

Dead code represents source code that has no effect on the execution of a program. For example, imagine a function named "Add" that adds two integers and returns the result. Now imagine a program that includes the Add function, calls that function, but never uses the function's output. This is dead code because the Add function was purposeless and wastes processing resources. Unreachable code is source code that will never be executed. For example, imagine that the Add function was never called in the example program. Again, it has no purpose, but because it is never called, it does not waste resources. In general, the term-of-art "dead code" can refer to either dead code or unreachable code.

Commented out code represents source code that has been placed within comment tags inside of a source code file, essentially removing the code from the active program. Commented out code can be confusing to other developers working on the codebase, or even to the same developer revisiting a code module at a later time. It is often not clear why code has been commented out. Is it because it contains a bug? Is it because the code has been excised from the program and is no longer relevant? Is it because the code is in the process of being rewritten and so the old code is still in place? In general, commented out code is

normal in small amounts but, in large quantities, it is indicative of sloppiness or incompetence on the part of the software developers.[4]

Repeated Code

Repeated code is code that performs the same function but occurs in multiple places in the codebase. An example is when a codebase contains three separate implementations of the same algorithm spread throughout different files. Repeated code is bad, not only because it is inefficient, but also because

Changes in one copy of the algorithm will not automatically update the other copies of the algorithm. Thus, if a developer is not aware of all algorithm locations in the source code, then the developer may change only some instances of the algorithm. This leads to the same computation outputting different results, often leading to inconsistencies and errors.

Programming languages provide a tool called *abstraction* to prevent this. It is industry standard to use the abstractions available in the coding language to ensure that all code and algorithms only appear in one place in the codebase. Any instances of copying and pasting code can and should be avoided. Pasting the

same code into multiple places in the codebase is almost always due to sloppy or inexperienced programming or to poor team cohesion (multiple team members being unaware that an algorithm has already been implemented somewhere else in the source code). In all cases, these types of errors should be caught during quality control and any instances of repeated implementations of algorithms or "cut-and-pasted code" should be removed.

Duplicated or repeated code is widely considered to be a bad coding practice. Martin Fowler, in his book

[4] Martin Fowler, *Refactoring: Improving the Design of Existing Code* at 237 (2019).

"Refactoring," states that

> *"[i]f you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them. Duplication means that every time you read these copies, you need to read them carefully to see if there's any difference. If you need to change the duplicated code, you have to find and catch each duplication."*[5]

Fowler explains that

> *[e]liminating duplicate code is important. Two duplicate methods may work fine as they are, but they are nothing but a breeding ground for bugs in the future. Whenever there is duplication, there is risk that an alteration to one copy will not be made to the other."*[6]

David Thomas in his "The Pragmatic Programmer," devoted an entire topic to this issue titled "DRY—The Evils of Duplication."[7]

Improper Error and Exception Handling

Exceptions are errors generated by programs at runtime. Exceptions can be anticipated and caught by a programmer in order to process the error most effectively. A critical part of designing a large system is creating new error types and catching and handling errors at runtime in order to allow the code to handle errors as they arise. For example, a developer writing a loan management system might write an exception called "CreditScoreException," which can be generated when a call to a credit agency does not return a valid credit score. Catching this exception at runtime would allow the developers to perform a specific action in response to not being able to fetch a valid credit score for an applicant. Exception swallowing is the practice of catching an error and then continuing without handling the error in a meaningful way. This practice is problematic because it essentially ignores errors that the developers of a function intended to be handled by the program. For example, if a particular part of the loan approval

[5] Martin Fowler, *Refactoring: Improving the Design of Existing Code* at 72 (2019).
[6] *Id*. at 350 (emphasis added).
[7] David Thomas, *The Pragmatic Programmer*, at Kindle Location 769 (2019) ("The alternative is to have the same thing expressed in two or more places. ***If you change one, you have to remember to change the others, or, like the alien computers, your program will be brought to its knees by a contradiction***. It isn't a question of whether you'll remember: it's a question of when you'll forget.") (emphasis added).

whitepaper

process expects input to be in the form of an integer, but a decimal is provided, the code could be designed to throw an IntegerException. A programmer could then set up the program to catch and handle the IntegerException, taking some form of a mitigating action rather than ignoring the error, allowing the loan application to continue processing, and ultimately approving a loan that should have been denied.

There are two common types of exception swallowing. The first catches an exception and simply does nothing with it, allowing the code to continue executing. The second catches an exception without regard to the actual type of exception so that a generic action may be taken (*e.g.*, log the issue and continue). Both of these represent bad coding practice—the former because exceptions are ignored entirely, the latter because specific action is typically more appropriate when the possible types of exceptions that code may generate are known in advance.

It is industry standard for programmers to anticipate the possible types of exceptions that their code may produce and assure that their code can catch and handle these exceptions individually. This prevents programmers from introducing unknown or unanticipated issues into their code (*e.g.*, an uncaught exception will crash the program). Furthermore, by catching specific exceptions during code execution, the developers can handle the specific errors as they occur at runtime. It may be desirable to catch generic exceptions in some instances, but generally this is used as a catch-all after all attempts to catch known exceptions have been exhausted.

Indeed, in describing the handling of exceptions, IBM notes that ignoring exceptions entirely is the "Worst" and that catching an exception merely to log it without managing the exception is "Very Bad:"[8]

```
// #1. Worst -- there is no indication that an exception
    // occurred and processing continues.
    try {
       // do work
    } catch (Throwable t) {
    }
```

[8] IBM, *Best Practice: Catching and Re-Throwing Java Exceptions*, https://www.ibm.com/support/pages/node/388611 (last visited January 19, 2021).

```
    // #2. Very Bad -- there is an indication that an
    // exception occurred but there is no stack trace, and



// processing continues.
    try {
        // do work
    } catch (Throwable t) {
        System.err.println("Error: " + t.getMessage());
    }
```

Lack of Effective Input Controls

Defensive coding is one of the most fundamental and important concepts in enterprise software development. As David Thomas explained,

> *"[w]e are constantly interfacing with other people's code—code that might not live up to our high standards—and dealing with inputs that may or may not be valid. So we are taught to code defensively. If there's any doubt, we validate all information we're given."*[9]

It is widely recognized that defensive coding—coding that verifies the accuracy and completeness of data—is paramount for successful software.[10] A major component of defensive coding is input validation.[11] As Steve McConnell explained, values of all data from external sources must be checked.[12]

Conclusions

While it could be argued that determining the quality of a given source code base is subjective and varies according to the favored coding disciplines and techniques of the reviewer, there are also standards for code quality that are generally accepted by all computer scientists, many of which are outlined in this whitepaper. In assessing source code quality, it is important to begin by first applying these accepted standards. This will aid in providing a logical defense for the assessment, and in establishing a consensus among other computer science professionals. The reviewer can then apply an analysis that is more focused on the nuances of the code, using their own standards for code quality to analyze the specific

[9] David Thomas, *The Pragmatic Programmer*, at Kindle Location 2206 (2019) (emphasis added).
[10] *Id.*
[11] Steve McConnell, *Code Complete (Developer Best Practices)*, at Kindle Locations 4823-4825 (2 ed. 2004) (emphasis added).
[12] *Id.*

functions of the code. This is particularly valuable when the reviewer has experience with other code bases containing similar functionality and design. Ultimately, it is this combination of accepted standards and personal standards that results in a comprehensive and high-fidelity assessment of code quality. It is therefore of great importance to work with a reviewer that has not only strong academic credentials, but extensive experience in performing code quality reviews, and specific areas of code type expertise.