Automated Software Inspection:

A New Approach to Increased Software Quality and Productivity

Table of contents

Introduction

Quality is an elusive goal throughout the software industry. The common assumption is that there simply is no efficient way to improve quality without significantly lengthening the development cycle, increasing development costs, or both.

Releasing quality software on schedule and on budget is almost impossible. Ever-shorter development cycles, limited engineering and QA resources, and increasing software complexity have combined to cause a decrease in the quality of software and an increase in the number of software defects. The economic impact from these defects is extremely high. Software defects are the leading cause of downtime for mission-critical applications and cause serious damage to the software's eventual direct or indirect users and to the development organization.

The software engineering community has long known that software inspection is an effective technique for removing defects with substantial long-term benefits. Inspection succeeds because it detects and removes critical errors early in the development process, before code reaches testing or deployment.

Automated Software Inspection (ASI) technologies are now emerging. These technologies, delivered as commercial tools or services, can locate many common programming faults - the same faults that can cause some of the most damaging failures. The strategy behind ASI is to analyze the source code before it is tested and identify potential problems in order to re-code them before they manifest themselves as programming bugs. The most innovative aspect of automated inspection is its ability to debug code even before it has been executed.

This paper discusses the reasons why ASI is needed, looks at the defects ASI can detect and how they are found, describes the technology behind ASI and how it complements traditional testing techniques, and compares and contrasts the available solutions.

Inspection Overview

Software inspection or code review is a visual examination of source code to detect defects or adherence to coding standards. It is important to note that inspection is not the same as testing - both are needed to ensure the highest quality software- and there are several important differences between the two.

- When we test, we execute the code
- When we inspect, we review the code
- Testing occurs during the QA/testing phases of development
- Inspection occurs during the coding phase

There are other differences as well - testing typically does not test all code paths and is therefore frequently hit or miss. With inspection, defects can be found on infrequently executed paths that will likely never be included in test cases.

Software inspection does not execute the code, so it is hardware-independent, requires no target system resources or changes to the program's operational behavior, and can be used long before the target hardware is available for testing purposes.

Figure 1 shows where inspections fit within the software development lifecycle. By visually inspecting source code and finding and removing defects early in the process, code quality is improved and a lower lifecycle cost is achieved.
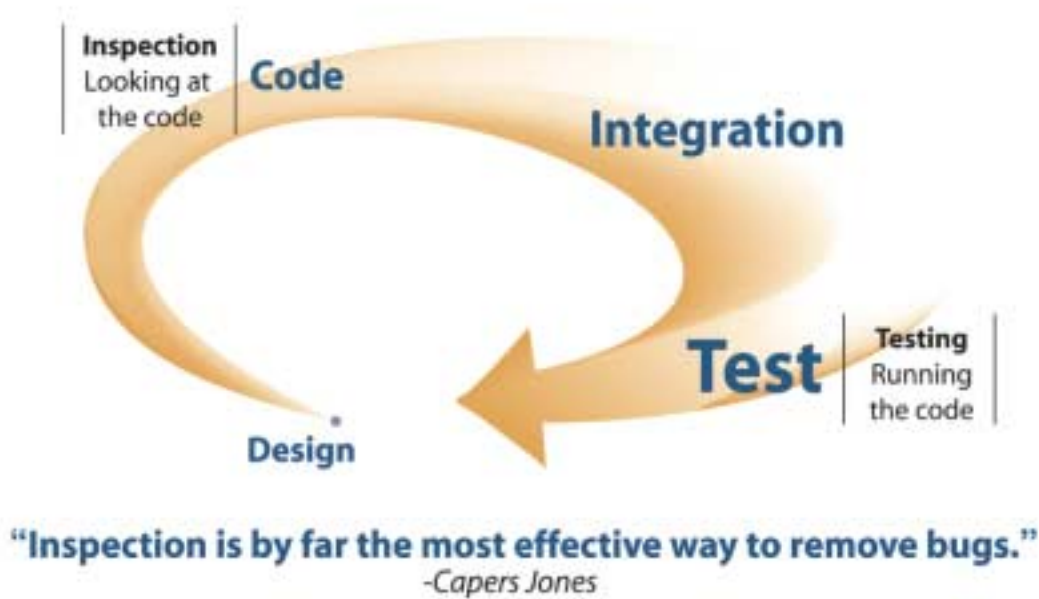
Figure 1: Inspection in the software development lifecycle

Code that has been subjected to inspection enters testing with higher quality, so testing time and effort is reduced. Since inspection is an examination of the code, executable code and test cases are not required. Inspection allows software to enter testing with higher quality which reduces the time and effort required to complete testing

The limits of testing

One of the best lines of defense in improving the overall quality of software is to improve software testing. Testing is acknowledged to be a critical part of total quality assurance, but it also has limitations, including:

- It is expensive and time-consuming to create, run, validate and maintain test cases and processes.
- Code coverage—the percentage of statements tested—drops inexorably as the system grows larger, meaning that testing validates less of the system.
- It can be difficult and time-consuming to trace a failure from a test case back to the root cause so that developers know what code to change.
- Testing cannot uncover all potential bugs. A study conducted by Capers Jones concluded that testing typically removes less than 50% of the defects. Even the best testing processes remove at most 85% of all software defects.

So, while comprehensive software testing is an important aspect of any quality assurance program, it is not a panacea. The bottom line is that software testing, although extremely valuable, is inadequate in light of the increasing need for highly reliable software. Ultimately, testing alone cannot guarantee defect-free code, nor can it ensure a sufficiently high level of software quality.

Debugging tools based on code instrumentation, such as Purify, BoundsChecker, and Insure++, are not able to find errors that aren't caught by reproducible test cases. They require that the application be complete and executable (or a test harness constructed) before they can be used. Also, they may not

easily be able to support use of custom memory allocation routines (i.e., the program has its own versions of malloc and free). These tools so greatly expand the code size (2x - 6x) and the data size (many megabytes) that they may become unusable because of resource constraints. Very often, program execution is so greatly affected (10x - 100x slower) that this performance overhead can make some test cases too slow to run.

"Testing is never finished, only abandoned"

To understand the above quotation from the Encyclopedia of Software Engineering, one must understand the concept of code versus path coverage. The diagrams below illustrate the problem. Figure 2 represents a single function (but the concept applies equally well to a module or task), which has a single entry, and for simplicity, a single exit. Execution begins at the top and exits at the bottom.
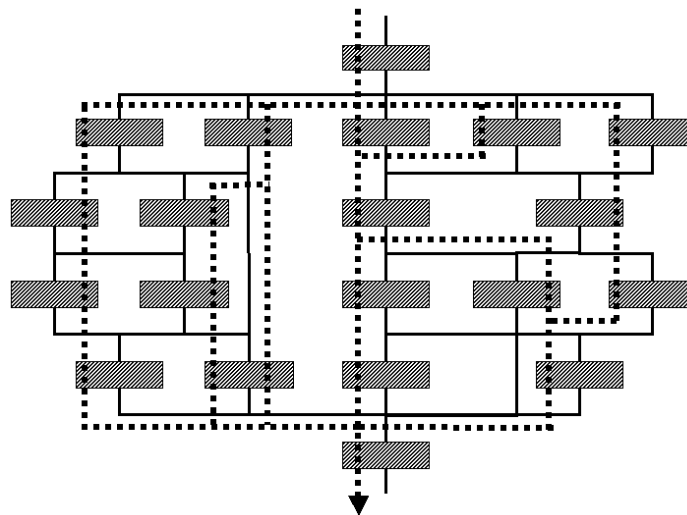
Figure 2: Function paths

Even if total code coverage is achieved, it is still not sufficient to ensure that there are no serious defects. Complete testing requires complete path coverage, not just statement coverage; all paths that connect two statements must be tested, not just the individual statements. The difference is illustrated in Figure 3, where the solid black lines show a number of paths that were missed by the test cases.

Figure 3: Paths missed by test cases

Achieving total path coverage is even more difficult than total code coverage. The code making up a large embedded system may contain literally billions of paths. Even if a comprehensive test suite were somehow to be constructed, the time and cost required to run it would be prohibitive.
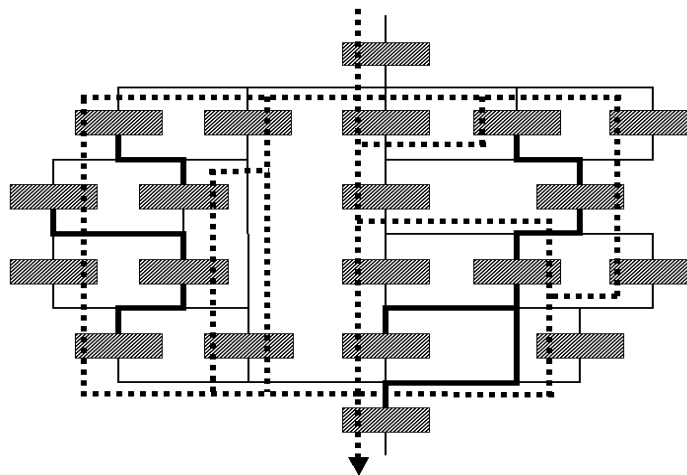
Since testing cannot achieve full coverage and requires expensive test case development, other techniques are necessary to cost effectively deliver high quality software. Software inspection is a technique that overcomes manylimitations of testing.

Traditional inspection techniques

Code inspection is an old idea, going back at least to Fagan's 1976 paper Design and Code Inspections to Reduce Errors in Program Development [Fagan, 1976]. Since that time, a number of experiments and case studies with a wide variety of methodologies have demonstrated how well inspection can work. An extensive bibliography can be found in The WWW Formal Technical Review Archive [Johnson, 1999].

Code inspections, as normally practiced, are a labor-intensive activity, often involving formal code reviews, structured walk-throughs, and similar techniques. The assumption has always been that programmers, who should be carefully examining the code, should detect the defects.

The results of inspections can be impressive, but too often inspections may not be performed well or at all. Management sees inspection as a drain on resources, and programmers often feel constrained by the formality of the process. The sheer volume of code involved is another constraint, as modern programs and systems often involve hundreds of thousands, or millions, of lines of source code. Therefore it is recognized that manual inspections can only be applied effectively on samples of the source code.

Until recently, there have been two main approaches to software inspection: formal inspections and independent code reviews. ASI is a new approach based on using computer programs to perform part of the inspection. We will briefly discuss the three types of inspection.

Formal inspection
The formal inspection is the most commonly used inspection technique. Formal inspections have only one goal - to find defects. They perform vigorous examinations of the code at points of stability, and utilize trained inspectors who follow defined steps to locate defects. In addition, the results uncovered by formal inspections can be used to help improve the development process. However, they are not useful for evaluating team member performance, reviewing programming style or exploring alternative solutions.

Unfortunately, formal inspections can fail for many reasons.   A common mistake is inviting management to the inspection. This can result in a lack of candor - developers may fear that management will use defect data in future performance reviews. It is also difficult to achieve consistent quality with manual inspections because so much depends on the individual performing the inspection. In addition, there can also be problems motivating developers to perform inspections on an on-going basis. Developers usually prefer to create new code rather than review existing code.

Independent code reviews
In an independent code review, an inspector reviews a small set of representative source code, using a well-defined language-independent template. High-level design considerations and coding specifics can be analyzed: for example, is functionality being re-used within the system? Has the programmer hidden debugging code? Have C++-specific capabilities been efficiently utilized?

Once the independent code review is completed, the results are documented and presented to the project manager, designers and programmers. This document is then reviewed and a set of priorities defined. Finally, a plan for either immediate or transitional change is created and implemented. The process is a

thorough one, but here too there is a risk that individual programmers will feel singled out for criticism in a forum of their peers.

Automated software inspection: the new approach

ASI offers greater efficiencies than any technique involving manual inspection, given the high level of importance attached today to achieving fast development turnaround and time-to-market pressures. Since there is no 'people judgement' involved in automated inspection, it comes much closer to the ideal of egoless programming.

Lint-like tools such as Flexelint from Gimpel Software and QAC from Programming Research are two commonly used ASI tools. These tools perform two functions. They verify that software complies with coding standards and they generate defect-warning messages of possible software defects.

Reasoning provides Illuma as an outsourced ASI service. Illuma delivers reports that show the cause and location of software defects in C and C++ applications.

The next section discusses ASI in more detail.

ASI Technology and methodology

The key question is - how can one achieve in-depth analysis of software that automatically inspects an application for critical defects? The answer is through the use of static analysis and abstract interpretation techniques. These techniques are discussed briefly below, along with ways to apply the technology.

Code analysis techniques

The starting point for the representation is abstract syntax trees. A parser reads the source code and produces an abstract syntax tree, which models all of the structural information contained in the source code while removing syntactic details such as the formatting of the source text. For example, an 'if' statement such as 'if (a < b) q = c + d else q = e + f' would be represented as an abstract syntax tree as follows: the top-level node would represent the entire if statement. There are three subtrees under the if statement node:

- a subtree that models the condition "a < b" would be under the if statement via the if-condition attribute
- the subtree that models the assignment "q = c + d" would be under the attribute "if-then-actions"
- the subtree that models the assignment "q = e + f" would be under the if-else-action attribute.

The next step is to annotate this abstract representation with information about the control structures (control flow) and the information flow (data flow) of the application. Figure 4 below shows part of the parse tree and the entire control flow graph of the following program:

```
...
    sum = sum – 10 ;
    if (sum < 0)
        { x = 3;
        }
    else
        { x = 4;
        }
    sum = sum + x;
...
```

Depending on the complexity of the analysis and representation techniques used, this representation can be used to detect real defects. Simple examples are the warnings generated by a compiler or a tool like Lint (see [Aho, Sethi, Ullman, 1988] for an overview of the used techniques).
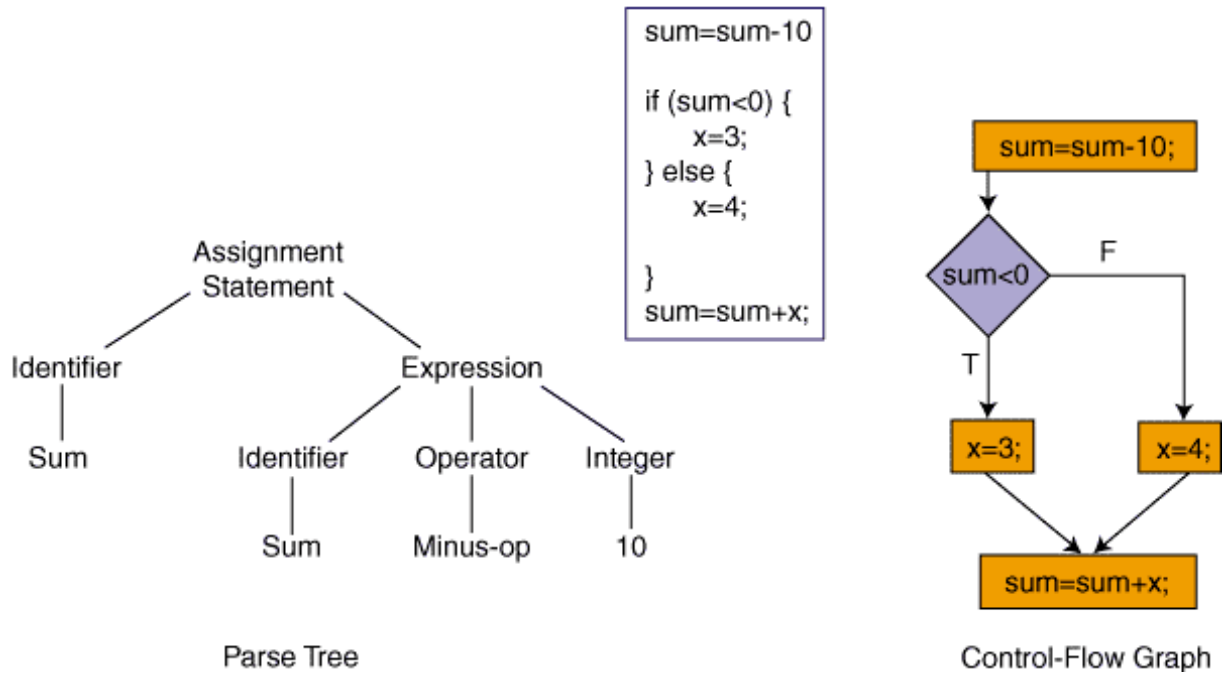
Figure 4: Example parse tree and control-flow graph

Types of defects found by ASI

ASI can uncover a range of structural defects that can cause crashes and data-corruption.

Certain constructs available in programming languages require corresponding constructs to occur earlier or later in the program. If this corresponding construct does not occur, or may not be executed under certain circumstances, the program has a structural defect. These structural defects reduce an application's reliability and are independent of programming language and application usage.

Examples of structural defects in the C/C++ language includes the following error classes:

- NULL pointer dereference; a dereference of an expression that is a NULL pointer.
- Out of bounds array access; expression accesses a value beyond the end of an array.
- Memory leak; reference to allocated memory is lost.
- Bad deallocation: deallocation is inappropriate for type of data.
- Uninitialized variable: variable is not initialized prior to use.

Methodology for ASI tools

ASI tools are only able to perform a portion of the inspection process. ASI tools generate a large volume of defect-warning messages that are false positives; in other words, the tool "thinks" it has found a defect, but a deeper analysis of the context shows that the reported issue is not a problem in reality. Everybody is familiar with the non-relevant warnings generated by compilers. This false positive problem is quite severe in ASI tools because the false positive rate from tools like Flexelint typically exceeds 50 false positives for each true positive.

In some cases false positives can be eliminated by creating filters that are able to automatically remove a subset of the false positives. However, a manual process is required to eliminate the false positives not caught by filtering. Developers need a way to evaluate each of the defect-warning messages to determine if it is a true defect or a false positive. To use ASI tools effectively, development organizations must hire or train inspection experts and implement a methodology for evaluating and removing false positives to ensure that the ASI results contain true defects and not a large number of false positives.

The cost and effort required to find a defect from ASI tools is high because a large number of false positives must be manually evaluated and eliminated for each true defect identified. To eliminate the false positives from a commercial application can require thousands of hours of developer effort.

Integration into the development lifecycle

Research conducted by Capers Jones has shown that the cost of repairing a defect is reduced dramatically when the defect is found early in the development cycle. Therefore, for maximum benefit, automated inspections should be performed near the end of the coding phase before the software has been released to QA/Test. Since ASI does not require a complete (compilable) application, subsystems can be inspected even before integration with other components of the application.

Many of today's development efforts are global, with teams in different locations contributing different portions of code to any given application. With ASI, a development organization can assess the quality of each segment of code at each step of the development process. Each portion of an application can be independently inspected as it is submitted, and then the integration of the various code components can also be inspected. The result will be a cleaner solution from start to finish.

Periodic re-inspections are the best way to ensure applications remain defect-free during development and maintenance. Inspections are scheduled to achieve a maximum defect removal rate. Typically, inspections are performed halfway through the initial coding cycle, at the beginning of code or feature freeze, and just before testing is finished and the code is released (see Figure 5).
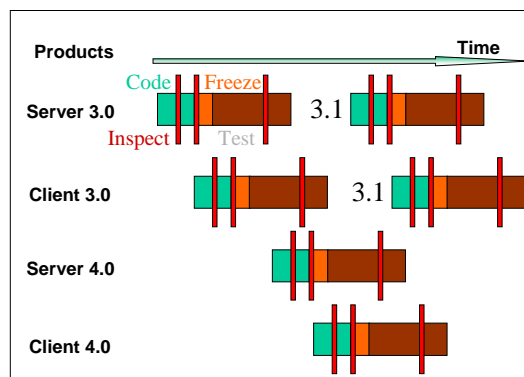
Figure 5: Integrating ASI into the development cycle

Illuma: Reasoning's ASI service

Illuma is the leading automated software inspection service, designed to boost the productivity of software development organizations coding in C and C++. The traditionally expensive and time-consuming process of defect detection is completed in five to ten business days. Illuma delivers two reports: one identifies the location and conditions that cause each defect and the other identifies application problem areas.

Illuma makes software development organizations more productive in three key ways:

- Reduces cost by finding defects less expensively
- Saves time by finding defects faster
- Improves quality by finding defects testing misses

Illuma technology

Illuma's analysis engine uses several program representations to verify that preconditions for the operations that are performed by the application are satisfied. For example:

- if a pointer is de-referenced, then the value of the pointer at the time of the dereference must be a valid address for the target data type of the pointer.
- if an array is indexed, then the value of the index must be within the upper bounds of the array.
- if a pointer is freed, then it must point to dynamically allocated memory.

The analysis engine must prove that the preconditions are true for all feasible paths leading to the operation. It uses an efficient symbolic evaluation algorithm to generate the paths and to track variable values along the paths, and determines if the preconditions are satisfied. If any precondition is not satisfied, then a violation is signaled. For a deeper discussion of the technology behind Illuma, the reader is referred to Value Lattice Static Analysis, A new Approach to Static Analysis [Brew & Johnson, 2001].

Illuma methodology

Although such powerful analysis technology is a key element in ASI, there are other points that must be considered. Reasoning delivers Illuma as an outsourced service.   A service engagement combines a comprehensive high-speed automated analysis of each application with the expertise of inspection and language specialists. Illuma is delivered as a service so development organizations do not need to have in-house inspection experts or a methodology to verify that the results are true defects and not false positives.

There are two advantages of a service model. First, it has enabled Reasoning to develop a specialized set of proprietary technology, processes and tools that allow the company to eliminate nearly all of the false positives.   Second, there is no impact on the development organization's resources. The time developers would have spent manually performing inspections or eliminating false positives from ASI tools results can be applied to other development activities.

Summary

Software failures are expensive and time consuming to detect, cause significant damage, and can seriously hurt a software development organization's reputation and business. ASI provides a fast and cost effective way to improve software quality. Through the regular use of ASI, the software development industry has the opportunity to achieve quality levels impossible to attain in the past.

Now that ASI technologies are a reality, software inspection can be performed quickly, providing tremendous benefits. New methodologies, such as the Illuma service from Reasoning, are well positioned to offer all the advantages of ASI to any company that can benefit from higher quality and a more productive development organization.

When it comes to improving quality assurance and testing for applications, including ASI early in the development process will result in significant cost savings. This is an area where outsourcing inspection to a reliable inspection service company will yield substantial returns. Using the right outside resource to

inspect and assess applications will substantially reduce the cost and time required to remove defects and will increase an application's reliability and quality.

Illuma is easily employed early in the development process before the application can be executed. It finds defects at the source code level and pinpoints their exact location. It produces a detailed report of the defects found and presents them in a way that a developer can easily diagnose and fix each defect. The inspection process involves no internal resources and facilitates the implementation of fixes before code is ready for internal QA.

Combined with conventional testing, ASI will allow software projects to obtain the benefits of combining traditional and newer methods in a balanced, cost-effective way so that software projects can yield new levels of quality. Through outsourcing, the ASI technology becomes available in the most flexible and non-intrusive way possible.

References

Aho, A., Sethi, R., and Ullman, J., "Compilers, Principles, Techniques and Tools", Addison Wesley, 1988.

Brew, W. and Johnson, M., "Value Lattice Static Analysis, A new approach to static analysis finds errors that slip through the cracks". Dr. Dobbs, March 2001.

Chou, T., "Fault-Free Software—The Case for Automated Software Inspection", Enterprise System Journal, February, 1999.

Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development", IBM Systems Journal,Vol. 15, No. 3, pp. 182–211, 1976. This pioneering work demonstrates quality and productivity improvements using formal design and code review.

Grady, R.B., "Successful Software Process Improvement", Prentice-Hall, Englewood Cliffs, 1997.

Humphrey, W.S., "Managing the Software Process", Addison-Wesley, Reading, 1989.

Johnson, P., "The WWW Formal Technical Review Archive", URL: http://www.ics.hawaii.edu/~johnson/FTR, 1999. This site contains an extensive bibliography on Formal Technical Review (FTR), a term encompassing methods such as Fagan Inspections, Active Design Reviews, Phased Inspections, a few inspection tools are mentioned.

Jones, C., "Software Quality in 1999: What Works and What Doesn't".

About Reasoning

Reasoning, Inc. is a leading provider of automated software inspection services that help software development organizations get products to market with lower costs and higher quality. The company's business is focused on organizations that develop C and C++ applications. Reasoning is headquartered at 700 East El Camino Real, Mountain View, California, 94040. Contact Reasoning at (650) 429-0350 or at http://www.reasoning.com.

About the Author

Jasper Kamperman, Ph.D., is Principal Member of Technical Staff at Reasoning, Inc. At Reasoning, Jasper has been a leading developer of the technology powering Reasoning's Illuma service. Jasper has a master's degree in Physics from the University of Utrecht, and holds a Ph.D. in Computer Science from the University of Amsterdam. He has published in ACM TOPLAS and several trade journals.