Stopping bugs before they kill your software organization

Static Source Code Analysis

White Paper with FAQs

by: Brent Duncan, Cleanscape Director

"The longer a software bug goes undetected the more costly it is to identify and eradicate." In spite of the near adage status of this statement, too many software development organizations are failing to adopt best practices and to implement available tools that can easily eradicate software problems before their associated risks and costs escalate. The end result is that the overwhelming majority of development organizations seem to be risking their existence by taking more time than necessary to ship buggy products.

Cleanscape Software International
www.cleanscape.net

Revision: 1 7.30.01.bd

Copyrights apply. All rights reserved.
Cleanscape, FortranLint, and LintPlus are registered trademarks of Cleanscape Software International.
Content copyright Cleanscape Software International
Written by Cleanscape Software International, Brent Duncan Director, with Arthur Chan, Cleanscape Product Manager; Chris Niggeler, Cleanscape President; Monty Swaiss, Cleanscape CTO; Ted Batha, Cleanscape CEO.
Other trademarks and copyrights may apply.

Stopping bugs before they kill your software organization
Rev. 1 - 7.30.01 bd
Friday, May 25, 2001

Comments to:
Cleanscape Software International
E-mail: bduncan@cleanscape.net

Contents

1 Document Overview 1-1
Title 1-1
Classification 1-1
Target 1-1
Summary 1-1
Issue 1-1
Criteria 1-1
Alternatives 1-1
Solution 1-1
Addendum 1-2

2 Stopping bugs before they kill your software organization 2-1
Living dangerously 2-1
Necessary, but inadequate steps toward risk reduction 2-2
Manual analysis 2-2
Debugging 2-2
Code Review 2-3
Dynamic analysis 2-3
Test 2-3
Stomping problems at the source 2-3
Increasing effectiveness with fewer resources 2-3
Brushing out problems 2-4
Team linting 2-4
Long-term protection 2-5
Historical lint 2-5
Using lint tools today 2-6
Analysis Report 2-8
Call Tree Report with include-file trees 2-8
Cross-reference Report 2-8
Increasing competitive viability 2-8
About the Author 2-8
About Cleanscape Software International 2-9
Resources 2-9

3 Frequently asked questions about Static source code analysis 3-1
What is static source code analysis? 3-1
Why is static source code analysis necessary? 3-1
Is there anything wrong with having my expert coders manually analyze their own code? 3-2
How is static source code analysis done? 3-2
Is it possible to produce bug-free code? 3-2
Why should I invest in a static source code analysis tool? 3-3

What are the key long term benefits associated with automating static source code analysis? 3-3
Doesn't my compiler already do syntax checking? 3-3
Doesn't ANSI C make lint obsolete? 3-4
There's a free lint utility that comes with UNIX, why should I want to pay for a commercial static source analysis tool? 3-4
Are you saying my debugger isn't enough? 3-4
How much time will I save by conducting automated static source analysis? 3-4
Can't I have my programmers generate the reports manually? 3-5
When do we stop testing software? 3-5
We have an established testing program with automated testing tools, what good will it do to add pre-compile analysis step to our software development process? 3-5
- Improves quality and saves money 3-5
- Static analysis is four times more effective 3-5
- Improves software quality 3-6
- Increases ROI 3-6

What is dynamic analysis, and why isn't it enough? 3-6
What is Cleanscape LintPlus? 3-7
Who can benefit from Cleanscape LintPlus static source code analysis? 3-7
Why should I choose Cleanscape's lint tools? 3-7
- The Cleanscape answer: 3-7
- Here's how some customers answer the question: 3-7

Can static source code analysis aid my quality practices 3-8

1 Document Overview

Title

"Stopping bugs before they kill your software organization"

Classification

Computer/Software/Programming/Tools/Static Source Code Analysis

Target

Primary: Second tier software developers: managers, leaders
Secondary: Third tier software developers: engineers, programmers

Summary

Issue

"The longer a software bug goes undetected the more costly it is to identify and eradicate." In spite of the near adage status of this statement, too many software development organizations are failing to adopt best practices and to implement available tools that can easily eradicate software problems before their associated risks and costs escalate. The end result is that the overwhelming majority of development organizations seem to be risking their existence by taking more time than necessary to ship buggy products.

Criteria

Implement processes and tools that will help software developers identify and eliminate software when it is easiest and cheapest to do so.

Alternatives

Manual analysis, automatic static source code analysis, debugging, code review, dynamic analysis, and test

Solution

While a static source code analysis tool will not eliminate the need for other bug eradication tools and tactics, it can greatly reduce the overhead on more expensive resources by catching problems early and allowing programmers to correct problems at the source

while they are most familiar with their code. By helping to eliminate problems at the source, static source code analysis increases the competitive viability of the software development organization by inversing the adage previously introduced: “The sooner a software bug is eradicated, the greater and faster the potential return on investment for a software project.”

Addendum

Frequently Asked Questions About Static Source Code Analysis

2 Stopping bugs before they kill your software organization

By Brent Duncan, Director Cleanscape Software International

"THE longer a software bug goes undetected the more costly it is to identify and eradicate."

This statement has become almost an adage for software developers. It's accepted almost without question because developers live daily with the associated risks of latent problems in software.

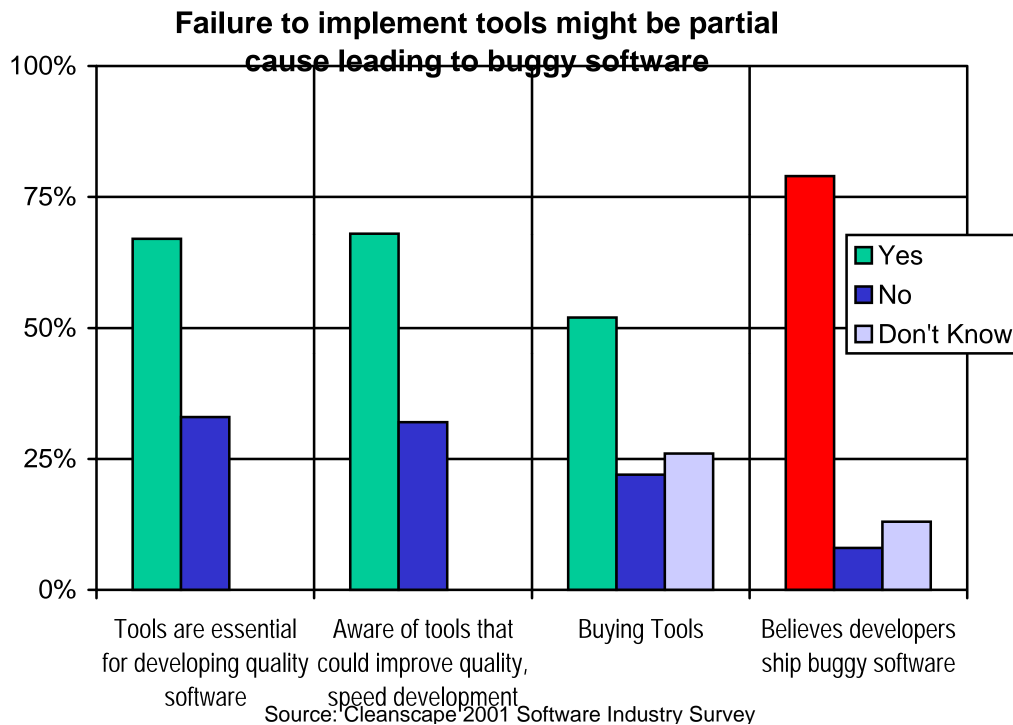
Software problems identified after compiling an application can halt development and take hundreds of developer hours to identify and correct – with the solution often causing additional problems. Problems that remain hidden until users find them not only cause customers inconvenience and expense, they can also result in PR nightmares that threaten the developer's profitability. If a magazine reviewer identifies a serious problem, not only will the company's reputation be threatened, the success of the product might be at risk.

Living dangerously

Regardless of the risks, research shows a seeming disconnect between accepting the adage and acting on it. In fact, many software development organizations are failing to implement available tools for eradicating software problems and improving development processes, according to the 2001 Software Industry Survey conducted by Cleanscape Software International.

While 67% of software development professionals surveyed believe software tools are essential for developing quality software and 68% said they were aware of tools that could help speed development while increasing product quality, only 52% said their organizations had plans to buy such development tools. This disconnect is likely a strong factor influencing why 79% of the developers surveyed said they believe software development companies ship buggy software.

In other words, there seems to be too many software development organizations that are failing to adopt best practices and to implement available tools that can easily eradicate software problems before their associated risks and costs escalate. The end result is that the overwhelming majority of development organizations seem to be risking their existence by taking more time than necessary to ship buggy products.



Necessary, but inadequate steps toward risk reduction

Acting on the adage to reduce the risks associated with latent problems in software requires introducing a step in the development process to help developers identify and eliminate problems as early as possible — before the costs and risks of eradication escalate.

Common bug eradication practices include manual analysis, debugging, code review, dynamic analysis, and test. While necessary, these steps can be inadequate, time consuming, and expensive.

Manual analysis

The most rudimentary method by which programmers check code for problems is on screen or by printing the code and checking the printout for problems. Debugging by *printf* might be sufficient for expert programmers working on simple applications, but it forces the organization to rely entirely on the skills of the programmer and might provide an insufficient documentation trail. This leaves the code highly subject to human error and can be difficult for managers to control. The more complex the code, the more programmer-hours it can take to conduct a sufficient manual check, and the less likely the analysis will identify problems that can eventually result in significant expense or risk to the organization.

Debugging

In the quest to kill bugs a typical development team might alternatively build, run, and debug a program hundreds of times during development — returning to coding when a problem is identified. Manual debugging is a time-consuming process that often accounts for

more than 50% of overall development time. Part of the reason for this resource-intensive cycle is that typical compilers simply can't recognize literally hundreds of coding problems that can lurk as latent bugs.

Code Review

A code review can be an effective means by which teams can identify whether code meets local standards, and might even result in identifying some problems prior to compiling. However, while important, the value of a code review can be limited by the following reasons:

- It can be an inefficient use of expensive resources.
- Increases possibility for inconsistency of process and result across the team, from project to project, and throughout the organization.
- Provides limited documentation trail for programmers and management.
- Human error passes through the compiler.

Dynamic analysis

Dynamic analysis attempts to find errors while a program is executing. The objective of dynamic analysis is to reduce debugging time by automatically pinpointing and explaining errors as they occur.

The use of dynamic analysis tools can reduce the need for the developer to recreate the precise conditions under which an error occurs. However, a bug that is identified at execution might be far removed from the original programmer and still must be returned to coding for correction.

Test

Test teams attempt to determine whether an executed software application matches its specification and that executes in its intended environment. Common steps in the test phase of development include: modeling the software's environment, selecting test scenarios, running and evaluating test scenarios, and measuring testing progress.

While testing can enhance a development team's ability to identify and eradicate problems, adequately testing today's complex software systems requires enterprise-class test automation tools, with associated infrastructure and trained staff. This translates into more expensive resources to identify problems that are becoming further removed from their source. Also, waiting for the testing stage to identify problems can be a costly mistake because standard test tools typically operate only on runtime units, or executables, and problems usually must be returned to coding for correction.

Stomping problems at the source

Increasing effectiveness with fewer resources

Even if a software development team is fully implementing code reviews, debugging, dynamic analysis, and testing it can greatly enhance software problem identification and eradication efforts by automatically identifying problems at their source—in the code prior to

compiling or execution. This can be done with a static source code analyzer. Not only is static source code analysis 400% more effective than dynamic testing[1], it helps speed development, save costs, reduce resources and risks associated with other problem eradication efforts, and increases product quality by automatically performing functions that can otherwise take hundreds of programming hours, like the following:

- Check source files for errors
- Identify and correct coding problems
- Isolate obscure problems
- Map out unfamiliar programs
- Enforce programming standards
- Compute customized quantitative indicators of code size, complexity, and density
- Generate detailed reports on the condition and structure of the code
- Document the code review process

In short, static source code analysis reduces resources required for other problem eradication tactics and tools by helping the programmer eliminate hundreds of problems at their source.

Brushing out problems

Also known as lint tools, static source code analyzers can help programmers to more easily identify and correct problems that often pass through a compiler, like:

- Syntax errors
- Inconsistencies in common block declarations
- Redundancies
- Unused functions, subroutines, and variables
- Inappropriate arguments passed to functions
- Non-portable usage
- Noncompliance with codified style standards
- Type usage conflicts across different subprograms/program units
- Variables that are referenced but not set
- Maintenance problems

Team linting

A static source code analyzer is not limited to only enhancing a programmer's ability to identify and eradicate problems in code; it can also offer process and performance enhancements throughout the software development team. A team leader can use a static source code analyzer to identify problems in groups and modules, while QA managers can

[1] Grady and Caswell, "Software Metrics: Establishing a Company-Wide Program". Prentice-Hall

use it to verify the integrity of an entire package. Project managers can use a static source code analyzer to help establish and enforce coding standards, and to establish quality control measures.

Long-term protection

In addition to the immediate returns possible by automating the process of identifying problems at the coding stage of software development, an organization can derive significant long-term benefits from static source code analysis.

A software product must be maintained by an organization throughout its entire lifecycle, while employees can come and go. When a problem is identified and the coder is on another project or no longer with the company — or simply separated by time from the original coding process — the organization typically must relearn what was originally done, or even duplicate the original coding step to correct the problem.

By automating the analysis and documentation of the code, the organization will always have a record of what was done and will be able to more easily identify problems without duplicating previous efforts. The messages produced by a static analyzer can also help prevent code drift by identifying questionable coding practices. In short, automated static source code analysis significantly reduces potential expenses from repetitive processes.

Historical lint

Today's commercial lint tools are the progeny of a free lint utility that was released with Unix. Unix's lint was originally developed to help ensure consistency of function calls across boundaries. While the proper use of ANSI can help solve this problem today, most other sources of errors in C code remain, including the following: uninitialized variables, order of evaluation dependencies, loss of precision, potential uses of the null pointer, consistency problems, and programmer error. The Unix lint utility was also difficult to use and has never been fully utilized by Unix programmers. However, in the early 1980's Cleanscape Software International (then IPT) took the idea of a static source code analyzer and enhanced it to produce advanced static source code analysis tools for Fortran and C that could provide extensive source code analysis.

Today, static analysis tools are available for most programming languages and should be considered a basic utility in any programmer's toolbox. Basic linting capabilities have even started appearing in compilers, but these features still tend to pail in comparison to today's dedicated static analyzers. The future of lint tools could very well be in the Internet. Recently introduced web-delivered static analysis tools like Cleanscape LintPlus Online (http://demo2.cleanscape.net/lplus) and Cleanscape FortranLint Online http://demo2.cleanscape.net/flint) promise to provide developers with the bug-stomping power of leading static analysis tools with the convenience, ease-of-use, and cost-savings of a web-delivered application.

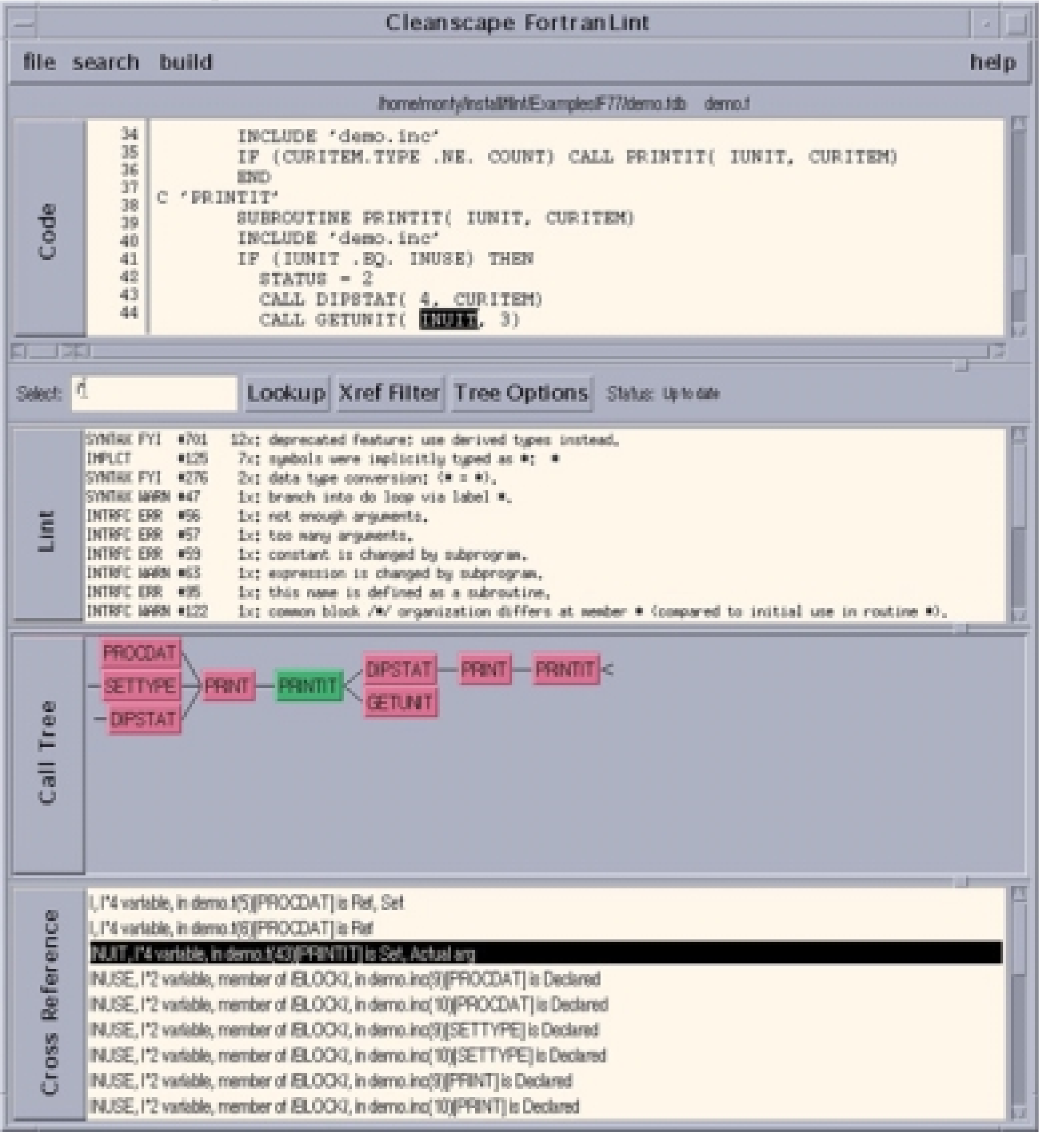
[CAPTION] Web-delivered lint tools promise to give developers the bug-stomping power of leading lint tools with the convenience, ease-of-use and cost-savings of the Internet.

Using lint tools today

In addition to providing feature sets that go far beyond the original lint and what typical compilers offer, today's static source code analysis utilities are far easier to use. Good lint utilities have broken down static source code analysis into a few easy steps. The user selects

reporting and analysis options, and points the utility at the source code files he wants to analyze.

In a matter of seconds, the utility conducts an intense comprehensive analysis on the code and its structure and generates reports according to the options selected by the user. Basic reports that might be provided include: analysis report, call tree report with include-file tree, and cross-reference report.



[CAPTION]A static analysis tool will conduct an intense comprehensive analysis on the code and its structure and generate reports that help the programmer eliminate problems, understand and document the structure of the code, verify the integrity of the package, and adhere to quality control measures.

Analysis Report

An analysis report typically summarizes all of the problems found by the lint utility, including: syntax errors, global interface problems, warnings, unused locals, unused results, portability problems, strict prototype problems, unusual constructs, and others. A source listing option will show the errors in the context of the actual code, and might provide live links back to the original source code so the programmer can make quick changes and analyze the code again. The summary report might include statistics for include function counts, I/O counts, and error summaries. It might also include suggestions for correcting problems.

Call Tree Report with include-file trees

A Call Tree report will typically show the calling structure of the analyzed code and identify areas where the structure is improperly formatted or broken. A Call Tree Report might also contain an include-file trees report that graphically shows the nesting structure of the "include" files used by the input code. A graphical depiction of the calling structure of source code can be particularly helpful for a programmer who needs to become quickly familiar with legacy code during transition and upgrade projects. It is also helpful for helping new project members become familiar with code from an existing project.

Cross-reference Report

A cross-reference report typically shows a symbol-table cross-reference. This acts as a powerful comprehension and debugging aid by showing the programmer how and where all symbols are used.

Increasing competitive viability

While a static source code analysis tool will not eliminate the need for code reviews, compilers, debuggers, testing, or dynamic analysis, it can greatly reduce the overhead on these more expensive resources by catching problems early and allowing programmers to correct problems at the source while they are most familiar with their code.

Identifying problems at the source level with a static source code analysis tool shortens the development cycle, prevents project delays that result from post-compile testing, and reduces costs by eliminating problems with cheaper resources earlier in the development process. In the long haul, by helping to eliminate problems at the source, static source code analysis increases the competitive viability of the software development organization by inversing the adage previously introduced:

"The sooner a software bug is eradicated, the greater and faster the potential return on investment for a software project."

About the Author

Brent Duncan is director of marketing for Cleanscape Software International, where he is responsible for developing and implementing integrated marketing and product strategies. As an executive and as a strategic management consultant Mr. Duncan has built and directed integrated marketing functions and cross-functional development teams for Honeywell SE, Westinghouse Security Electronics, and Auspex Systems. Mr. Duncan is an adjunct professor

in the University of Phoenix School Department of Business Management. He holds a Masters in Organizational Management from University of Phoenix and Bachelors in Communications and Public Relations from BYU.

About Cleanscape Software International

Cleanscape Software International is a leading innovator of automated software development and testing solutions that simplify build environments, enabling customers to shorten development cycles, optimize resources, and increase product quality with the aim of increasing customer return on investment.

Cleanscape is a privately held company founded in 1999. In addition to sales and service revenues from its established product line, Cleanscape is partially funded by private investors.

Cleanscape is headquartered in Mountain View, California. For more information please call 800.944.5468 or 650 864-9600, or visit http://www.cleanscape.net.

Resources

- Marilyn Bush, “Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory” Proceedings of the 12th International Conference on Software Engineering, IEEE Computer Society Press
- Reg Clemens, Ph.D. Physics. “White Paper: A LINT for Fortran Programs”. http://www.cleanscape.net/stdprod/flint/flint_review.html
- Compuware, “Accelerating Development of Multi-language Components and Applications for the Enterprise and Internet,” White Paper.
- Ian F. Darwin “Checking C Programs with lint”. O’Reilly & Associates, Inc. Sebastapool 1988
- Brent Duncan, “Cleanscape 2001 Software Tools Survey”, Cleanscape Software International.
- Louis A. Franz and Jonathan C. Shih, “Estimating the Value of Inspections and Early Testing for Software Projects” Hewlett Packard Journal.
- R. B. Grady and D. L. Caswell, “Software Metrics: Establishing a Company-Wide Program,” Prentice-Hall, 1987
- Steve Maguire, “Writing Solid Code”. Microsoft Press. 1993
- Tom Parker and Glenn Wright, “White Paper: Using Fortran Source Code Analysis at NCAR” http://www.cleanscape.net/stdprod/flint/flintcase.html
- Karl Wiegers, “Personal Process Improvement: You don’t need an official sanction to tune up your own engineering savvy.” “Software Development Magazine”. May 2000

###

3 Frequently asked questions about Static source code analysis

What is static source code analysis?

Static source code analysis is the process by which software developers check their code for problems and inconsistencies before compiling.

Organizations can automate the source code analysis process by implementing a tool that automatically analyzes the entire program, generates charts and reports that graphically present the analysis results, and recommends potential resolutions to identified problems.

Static analysis tools scan the source code and automatically detect errors that typically pass through compilers and become latent problems, including the following:

- Syntax.
- Unreachable code
- Unconditional branches into loops
- Undeclared variables
- Uninitialised variables.
- Parameter type mismatches
- Uncalled functions and procedures.
- Variables used before initialization.
- Non-usage of function results.
- Possible array bound errors.
- Misuse of pointers.

Why is static source code analysis necessary?

The longer a bug goes undetected the more costly it can be to identify and eradicate. A problem identified during integration can stop development and can take hundreds of developer hours to identify and correct. If a latent problem is identified by end-users it not only can cost them significant expense and dissatisfaction, it can also threaten a company's profitability. If the media identifies a latent bug in a company's new software product it could damage the company's reputation and it could threaten the ultimate success of the product.

To reduce the risks of continuing development of or shipping software with latent problems the software development organization needs to implement procedures that allow it to easily identify and eradicate problems as early as possible, preferably prior to compiling on

the source code itself. While the programmer should perform code reviews, relying entirely on this manual process is time consuming, is only as reliable as the skills of the individual programmer, and will likely be conducted differently for each programmer on the team. In other words, manual pre-compile analysis can slow development, is subject to human error, and is difficult for managers to monitor and control.

By conducting static source code analysis and successfully identifying and correcting problems prior to compiling code, software developers reduce the risk and potential costs that grow exponentially in proportion to how long it takes to identify as a problem continues to go undetected.

Is there anything wrong with having my expert coders manually analyze their own code?

The most rudimentary method by which programmers perform static source code analysis is manually on screen or by printing the code and checking the printout for problems. Such a manual code review might be sufficient for expert programmers working on simple applications, but it forces the organization to rely entirely on the expertise of the programmer, provides no documentation trail, and leaves the code subject to human error. The more complex the code, the more programmer-hours it takes to conduct a sufficient manual review, and the less likely the analysis will identify problems that can eventually result in significant expense or risk to the organization.

Static analysis is faster and less prone to error than manual code review. It accomplishes what would often be virtually impossible by manual methods: a thorough, accurate review that checks for all known component or language interactions. Static analysis enforces coding standards and produces software complexity metrics.

How is static source code analysis done?

Development organizations have two basic alternatives for conducting static source code analysis: manual code check or automatic static source code analysis. A third alternative is to augment a coder's manual checking with an automated source analysis tool.

Is it possible to produce bug-free code?

With rising user expectations for problem-free products programmers are under increased pressure to produce bug free code, but is it possible to do so with the complexity of today's mission critical applications? "With the growing complexity of software and the associated climb in bug rates, it's becoming increasingly necessary for programmers to produce bug-free code much earlier in the development cycle, before the code is first sent to Testing," wrote Steve Maguire in "Writing Solid Code" (Microsoft Press). "The key to writing bug-free code is to become more aware of how bugs come about. Programmers can cultivate this awareness by asking themselves two simple questions about every bug they encounter: "How could I have prevented this bug?" and "How could I have automatically detected this bug?" Mr. Maguire's answer is a key value proposition of automated static source code analysis tools.

Why should I invest in a static source code analysis tool?

A preferable alternative to debugging by *printf* is to automatically conduct source code analysis with a lint tool before compiling the code. A lint tool will analyze your source code and find problems that a compiler typically won't catch, like bugs, glitches, inconsistencies, and redundancies. A lint tool helps speed development, save costs, and increase product quality by automatically performing a function that can otherwise take hundreds of programming-hours, establishing and documenting an automated debugging process, and eliminating problems early in the development project.

By having the programmer run his code through a static source analysis tool when the code is still fresh in his mind, it helps him to automatically identify and correct problems even before the code is compiled. This saves the risks and costs associated with problem identification and eradication later in the software development or product life cycle.

A good static source code analysis tool can also provide project managers with coding standards enforcement and statistical quality control measures by detecting noncompliance with codified style standards, by detecting maintenance or portability problems, and by computing customized quantitative indicators of code size, complexity, and density.

What are the key long term benefits associated with automating static source code analysis?

In addition to immediate returns possible by automating the process of identifying problems at the coding stage of software development, an organization receives significant long-term benefits. A software product must be maintained by an organization throughout its entire lifecycle, while employees can come and go. When a problem is identified and the coder is on another project or no longer with the company, or simply separated by time from the original coding process, the organization typically must relearn what was originally done, or even duplicate the original coding step to correct the problem. By automating the analysis and documentation of the code, the organization will always have a record of what was done and will be able to more easily identify problems without duplicating previous efforts. In short, static source code analysis helps reduce long-term risks resource outlays.

Doesn't my compiler already do syntax checking?

While compilers usually perform adequate syntax checks, a good lint tool provides problem identification features that typical compilers don't offer, like precision tracking, value tracking, initialization checking, strong type checking, and macro analysis. A good lint tool will also provide FYI messages to give you tips on best practices. The lint tool should be able to also look across a set of modules to find inconsistencies and redundancies. These are things a compiler can't do, and that can be difficult and risky to perform manually for today's complex and mission critical applications.

Basic linting capabilities have started appearing in compilers, but these features still tend to pail in comparison to today's dedicated static analyzers.

Doesn't ANSI C make lint obsolete?

Unix's lint was originally developed to help ensure consistency of function calls across boundaries. While the proper use of ANSI can help solve this problem today, most other sources of errors in C code remain, including the following: un-initialized variables, order of evaluation dependencies, loss of precision, potential uses of the null pointer, consistency problems, and programmer error.

There's a free lint utility that comes with UNIX, why should I want to pay for a commercial static source analysis tool?

Today's commercial lint tools are the progeny of a free lint utility that was released with Unix. Unix's lint was originally developed to help ensure consistency of function calls across boundaries. While the proper use of ANSI can help solve this problem today, most other sources of errors in C code remain, including the following: uninitialized variables, order of evaluation dependencies, loss of precision, potential uses of the null pointer, consistency problems, and programmer error. In addition, the Unix lint utility is difficult to use and has never been fully utilized by Unix programmers, according to Tom Parker, Technical Consultant for the Scientific Computing Division at the National Center for Atmospheric Research in Boulder, Colorado. "Cleanscape took the idea of a static source code analyzer and enhanced it to produce lint tools for Fortran and C that could provide extensive source code analysis," Parker said.

Are you saying my debugger isn't enough?

Standard debuggers operate only on runtime units (executables). A good static source code analysis tool is designed to provide pre-compile error checking of a program at the source level — before the code is executed. A good source code analyzer keeps track of all information about every symbol in the code. As a result, the analyzer can generate intelligent information about the analyzed code.

How much time will I save by conducting automated static source analysis?

"Analysis which had required weeks when done manually, was completed with Cleanscape's static source code analysis tool in a matter of a few minutes" Glenn Wright, researcher for M2K in Austin, TX.

Saying "a few minutes" is an overestimate for most code. The CPU, not the utility itself, typically limit analysis speed. Small to medium sized projects with hundreds to thousands of lines of code will typically take a few seconds, while large projects with millions of lines of code might take "a few minutes". In short, the time you save will be directly related to the size and complexity of your code, but will typically be greater than 90% reduction in programming hours compared to performing the same function manually.

Can't I have my programmers generate the reports manually?

The reports automatically produced by a static source code analyzer can take an entire programming team weeks to produce — which is why such reports are not likely to be produced without the use of the automated tool.

When do we stop testing software?

There are three basic criteria for determining when to stop testing, as follows:

- Never. Once you deliver the product, the customer will take over.
- When the project runs out of or loses resources.
- When the product life cycle is complete.

We have an established testing program with automated testing tools, what good will it do to add pre-compile analysis step to our software development process?

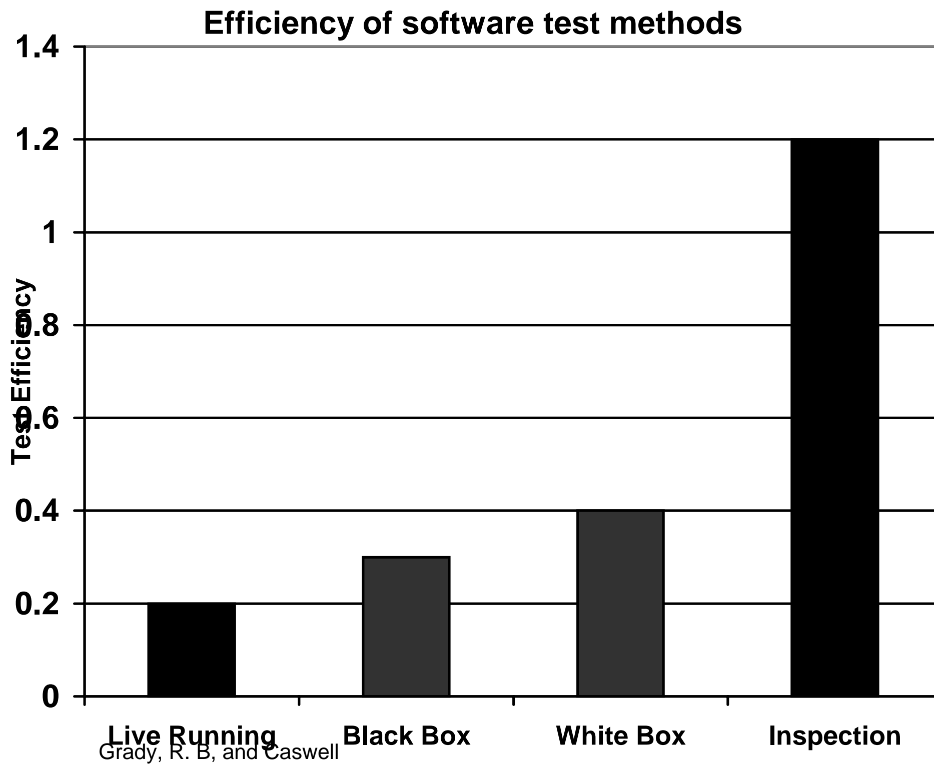
Improves quality and saves money

"Testing alone will not determine if code will work on different platforms, if it is written efficiently and whether it adheres to particular coding guidelines or standards… Programmers can start improving code by using advanced linter tools… Programmers may find that inspections on code of more than 1,000 lines will help find bugs that testing would not turn up and which would be more expensive to correct later."

C.R. Dichter in "Two Sets of Eyes: How Code Inspections Improve Software Quality and Save Money"

Static analysis is four times more effective

Static testing is four times more effective than dynamic testing, according to Grady and Caswell in "Software Metrics: Establishing a Company-Wide Program" (Prentice-Hall). See "Efficiency of software test methods" below.



Improves software quality

"Finding and fixing defects early in the software development life cycle is much cheaper than finding and fixing the same defects later on, and results in higher quality software." - Marilyn Bush, "Improving Software Quality: The Use of Formal Inspections at the Jet Propulsion Laboratory" Proceedings of the 12th International Conference on Software Engineering, IEEE Computer Society Press

Increases ROI

"Detecting software defects early reduces the cost of dealing with the defects later in the development cycle. One HP entity used metrics data from several software projects and an industry profit and loss model to show the high cost of finding and fixing defects late in the development cycle and during post-release…" and found that ROI increased proportionally the earlier problems were identified. - Louis A. Franz and Jonathan C. Shih, "Estimating the Value of Inspections and Early Testing for Software Projects" Hewlett Packard Journal.

What is dynamic analysis, and why isn't it enough?

Dynamic analysis attempts to find errors while a program is executing. The objective of dynamic analysis is to reduce debugging time by automatically pinpointing and explaining errors as they occur.

The use of dynamic analysis tools can reduce the need for the developer to recreate the precise conditions under which an error occurs. However, a bug that is identified at execution might be far removed from the original programmer. Also, the analysis might not result in an adequate documentation trail. This can introduce difficulties in correcting problems as programmers familiarize themselves with the original code.

Identifying problems at the source code level with static source code analysis can further reduce the time and expense of other debugging tactics by allowing developers to identify and eradicate problems at the source code level, prior to compiling. Even if a development team is using a complete test automation suite, using a static source code analysis tool will reduce resources required for more resource intensive debugging tactics, and increase effectiveness of problem identification and eradication by up to 400%

What is Cleanscape LintPlus?

Cleanscape LintPlus is an advanced static source code analysis tool that expedites and simplifies the debugging and maintenance of ANSI C Code. Cleanscape LintPlus rigorously examines source files both individually and as a group, almost instantly generating comprehensive user-definable reports on hundreds of problems that are readily accepted by a typical compiler.

Automating source code analysis with Cleanscape LintPlus helps to identify potential problems early in software development, significantly reducing the risks and expenses that grow as problems continue to go undetected. Cleanscape LintPlus provides project managers with coding standards enforcement and statistical quality control measures by detecting noncompliance with codified style standards, by detecting maintenance or portability problems and by computing customized quantitative indicators of code size, complexity, and density.

Who can benefit from Cleanscape LintPlus static source code analysis?

Cleanscape LintPlus can be used by programmers for coding individual modules, by project managers responsible for groups of modules, or by QA managers responsible for verifying the integrity of an entire package.

Why should I choose Cleanscape's lint tools?

The Cleanscape answer:

A failure to meet growing user demands for high performing, mission critical applications at reasonable prices threaten the viability of the software development organization. Cleanscape lint tools help you to automatically identify problems at their source, allowing you to develop and deliver a higher quality product faster and cheaper.

Here's how some customers answer the question:

Cleanscape's lint tools are "much better than anything else available" according to Reg Clemens, Ph.D. in Physics, Phillips Lab. "In addition, the ability to selectively turn off individual error messages makes it possible to reduce the output incrementally once you are

sure that certain types of warnings can be ignored. I would recommend Cleanscape's lint tools to anyone involved with large codes, be they their own or something that they have inherited and now need to understand and debug/modify."

"Cleanscape's lint tool is very effective in catching mistakes that creep in during the coding process, or oversights that could come back to bite you later on and take substantial time to debug," said Wallace Welder, Engineer/Scientist Rocketdyne division of Boeing. "Sometimes you get a segmentation fault after the first compilation and you have no idea what might have caused it. It could be a typo, or you could forget to list something by accident or misname something -- any number of mistakes can happen. Cleanscape's lint tool is excellent for catching all these errors that normally would be difficult or time-consuming to find."

Can static source code analysis aid my quality practices

Whether or not your organization has a management-driven process improvement program, you can take steps to improve your personal software engineering approach — and perhaps your workgroup's processes — even without official management sanction or leadership, according to Karl Wiegers in "Personal Process Improvement: You don't need an official sanction to tune up your own engineering savvy" in the May 2000 issue of *Software Development Magazine.*

"Each developer is responsible for the quality of the work she performs" Wiegers said. "Unfortunately, few developers are adequately trained in the arts of quality analysis and measurement, testing and technical review."

Static source code analysis is a basic tool for improving your personal and team productivity, according to Wieger. "Use quality tools to scour your code for as many potential problems as you can. At the least, use (a) static code analyzer… to find subtle defects the compiler and a casual visual scan might miss. A developer once told me that (a static source code analyzer) reported 10,000 errors and warnings when his team ran their program through it, so they never used (the lint tool) again! Finding 10,000 errors and warnings suggests to me that the program had some quality problems. The ostrich approach doesn't make the program more reliable. If something is wrong with your code, fix it now when it's cheap, rather than later when it's painfully expensive.

"Anything you do to upgrade your own capabilities will save time, reduce rework and improve your productivity," Wieger wrote.