

News

Klocwork Insight Brings Code Analysis to the Desktop

Posted by [Scott Delap](#) on Jul 15, 2008

[CommunityJava](#) [TopicsCode Analysis](#) , [Debugging](#)

Share  |

Earlier this year Klocwork released a desktop product, [Klocwork Insight](#), bringing their automated source code analysis features to individual developers. Insight is available in both C++ and Java versions. Until now the industry has frequently made use of source code analysis after the checkin process during the system building phase of a project. Insight moves up the application of source code analysis to the build and implementation portion of the development lifecycle. Insight can be integrated in a variety of ways from the Java perspective:

...It supports all versions of Java up to and including 1.6, and has enhanced support for a variety of popular Java frameworks, including the Google Web Toolkit, AWT, Hibernate, JavaMail, J2EE and J2ME. Klocwork Insight supports Eclipse, IBM Rational Application Developer, IntelliJ IDEA, and JBuilder 2007 IDEs as well as the ANT & Maven build environments...

Klocwork made news in the past [announcing](#) it had found a number of flaws in open source projects that previous Department of Homeland Security studies had not discovered.

InfoQ recently sat down with Klocwork CTO Gwyn Fisher to discuss the product. The first topic of discussion was how Klocwork decides what analysis items to focus on:

The vast majority of our research work is performed based on real errors found in customer code. The reality of developing an automated solution in the defect detection space is that there is an ongoing balance to be struck between incorrectly pointing out code that isn't defective (false positive) and failing to point out code that really is defective (false negative). Building on our customers' real experience using our product informs both the balance and the direction of investment. Whether at any point in time we're more focused on false positives or false negatives depends on patterns that we spot in usage reports (which we examine in mind-numbing detail, using a variety of 3D visualizations, at regular intervals).

Fisher went on to add that it is difficult to label any one type of analysis module as the most difficult to developer. Buffer overflows are one of the more interesting types, however, simply given the number of scenarios in which overflow / underflow can occur, and the subtleties involved in telling real overflows from code that operates correctly given the prevailing semantic of the code base in question. Fisher summarized that developers may well "know" that values stored in configuration data or read from an input device will never be outside of a given range, and so theoretical overflows driven by that data can be problematic to deal with, even though they're factual flaws.

InfoQ next asked about the benefits of Klocwork Insight's Connected Desktop Analysis feature to the average developer:

[GF] Get this: don't check in code that doesn't work! It's that simple. The connected desktop allows the developer to analyze their code locally before check in and with exactly the same accuracy and context as during integration build analysis.

Let's play that out with an example to make it more apparent. Consider a developer, let's call him Jim for sake of labelling, working in a typical consumer mode, i.e. his work is dependent on work performed by another developer down the hall. When analyzing Jim's code for correctness, it's one thing to tell him that the code he's written is good, it's another to know how the code he's consuming operates, and therefore to validate that his interaction with that consumed code is correct.

Without distributed system context, the analysis engine can't tell whether that null object reference being passed to method foo() is going to cause a problem or not. Without that context, the analysis engine doesn't know if method foo() returns a usable object reference or not. But with that context, all those grey areas are replaced instead with factual analysis. The engine knows exactly what's going on in foo(), even though you don't have the code for it locally, and can tell you if you're doing something dumb in exactly the same way as integration build analysis can do.

This equivalence in analysis accuracy pays off in several ways, with productivity being the most obvious. But if you consider the cultural barriers faced by companies implementing code analysis, it's even more compelling. Typically, a company will implement code analysis as an audit tool by integrating it within the integration build, and focusing on reports as the interaction point. This requires one of two things to be put in place, either a workflow imposition to require the developer to leave their normal environment and visit the reporting application just to find out what they did wrong, or perhaps an e-mail driven workflow that delivers "you failed" notices from on high.

Putting accurate, fast and valid analysis into the hands of the developer, however, removes that "blame cycle" from code analysis and focuses instead on the developer creating high quality, high security code from the outset.

At this point InfoQ asked Fisher to compare Klocwork to the popular open source FindBugs tool:

FindBugs is a great tool, and I have a lot of respect for Bill Pugh for doing good work and making it freely available. Most people's first exposure to good Java code analysis will be his product, and that's good for the market as a whole. Where we differ from FindBugs is in the depth of analysis that we perform, and the number and type of defects that we can find. Specifically, FindBugs focuses on what are called intra-procedural defects, i.e. defects that can be analyzed by following the control and data flow within a particular method (actually, they can extend this analysis if the developer annotates their code to detail expected behaviour, but that's kind of cheating ;p). Klocwork's analysis is performed inter-procedurally, meaning that we can find defects that span method invocations across class, package or module boundaries. This type of "whole program" analysis is what commercial static analysis is all about.

Finally, InfoQ asked Fisher to compare Klocwork's experiences writing their C++ analysis tools to their newer Java counterparts. He explained that the cornerstone of good code analysis in Klocwork's view is their compiler. For Java they first attempted to optimistically analyze debug mode byte code, thereby removing the requirement for writing a compiler. That technique did not prove effective enough however. Beyond the compiler the most significant investment in moving to Java revolved around the checker library and ensuring that they were bringing an engine to market that's was as rounded as their existing offerings. Presently Klocwork is at the point of having about the same number of checkers in each language they support (roughly 175 per language).

RelatedVendorContent

[Got fires in production? Find root cause in minutes. FREE Java performance tool](#)

[Stop Java Problems Before They Reach Production](#)

[Best Kept Secrets of Peer Code Review \(Free Book\)](#)

[A practical guide to choosing the right agile tools](#)

[Free online Flex training](#)

No comments