

About the developer's dream to write the perfect piece of code

By **Ben Chelf**, Coverity

Static code analysis can be used to help you in writing good code. Not perfect code, but good code in C/C++ as well as Java.

The screenshot shows a table with columns for 'File Name', 'Line Number', 'Severity', 'Message', and 'Status'. The table contains numerous rows of defect information, indicating a high volume of issues identified by the tool.

Figure 1. This screenshot shows the excellent progress by the Samba developers, fixing 212 defects in just nine days

■ Don't you love looking at a good piece of code? I'm talking about the kind of code where the design is so sound that it practically wrote itself, where there were no nasty surprises upon implementation, where it was 100% feature-complete and bug-free and you didn't have to patch it up a bunch of times? Maybe I'm squarely in the land of Santa Claus and the Easter Bunny now, but I believe, deep down, all developers want to write that perfect piece of code. Unfortunately, real life has other ideas. Deadlines, unclear or conflicting requirements, ridiculous scope, being human – all these things keep us from the promised land of perfect code.

But here's the rub: though it may be satisfying to dream about, it is likely that you will never produce completely perfect code for real world applications. You'll sit down to write a piece of code, you'll do the best you can, taking into account everything you know about how the system works, how your piece of code fits into that system, and so forth. But we all know there will be mistakes – probably lots of them. Some will be minor but others will kill you later when your customer hits them. Should you fix them all?

The hard part is that while the code is not perfect, you must recognise that every time you have to change the released code, you introduce risk into the system. Thousands, hun-

dreds of thousands or millions of people are using it as is, and if you decide to make changes it might work differently for those people. As such, your job is to decide which mistakes should be fixed and which mistakes can be left alone. Your code will, unfortunately, be forever imperfect. And that's the paradox. The tools you use to help write and debug code must be cognisant of this fact.

As introductory computer science classes increasingly move to Java (even the high school AP computer science curriculum is Java-based now), the tools available to C/C++ developers should move over to Java as well. Over the last decade, as Java has exploded in popularity, there have been tremendous breakthroughs in the area of practical static code analysis for defect detection. Today many commercial tools are available to perform static code analysis of your C, C++, and Java code. I work for one such tool provider and I'll discuss our experience expanding from C/C++ into Java in the paragraphs that follow. We'll explore how some of the concepts we used to analyse C/C++ code translated into the Java realm and the lessons we've learned in making this type of technology practical to help you write good code. First, I'll dig into some discussion of architecture and then I'll give you my philosophy on finding bugs automatically and the true purpose of these tools.

From a code analysis perspective, C++ and Java have a lot in common. Both require you to build some representation of the code into the guts of your analysis for the purpose of performing dataflow analysis. This means breaking each function or method into basic blocks, computing a control flow graph, and having an analysis engine that can push checks down each possible execution path in the methods while keeping track of the relevant variables and their values. With this, each check can then pull out relevant constructs while analysing the code. For example, if I'm looking for NULL exception problems, my NULL checker simply looks for places where objects are compared against

SOFTWARE DEVELOPMENT

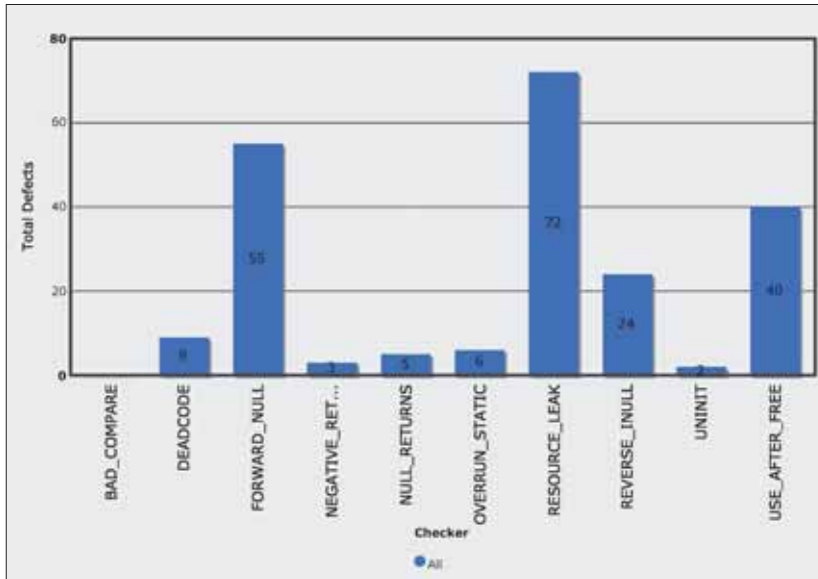


Figure 3. This display is one of the graph types available in the Prevent user interface. It shows the number of identified defects by defect type.

NULL or assigned a NULL value, and then lets the analysis push down a path until I see a dereference when that value is NULL.

Listing 1 shows an example from the Struts framework. Notice that on line 171, the developer compares body against null. Unfortunately, the developer probably meant to have that comparison be == instead of !=. In the case where the pointer is null, the code will skip over the assignment on line 172 and dereference the body variable on line 175. Oops! Listing 2 shows you what that code looks like in the interface of a static code analysis tool. The analysis engine pushes the checker down all the paths in this function. The checker notices the comparison against null, keeps track of the body value as being null when the condition on line 171 is false, and then reports a problem when it is dereferenced as null. Simple enough, right?

Well, almost right. The biggest problem that the designer of any static code analysis tool faces is false positives. What is a false positive? Basically, any time the analysis reports a defect where there is none, that's a false positive. Some people call this noise, but I like to stay away from that term for a reason you'll discover later in the article. Noise is a problem, but it's a different problem. To better understand a case that might trip up a static code analysis tool, take a look at Listing 3 – the Struts code from the previous example has been slightly modified to introduce data dependence between the value of body and the value of body_tracker. Notice that after the test of body against NULL, the value of body_tracker will be 5 if body is not NULL and 12 if body is NULL. As such, there is no longer a NULL dereference on line 177 because it is

guarded by the check of body tracker. This example is simple enough, but may fool some simple analysis engines into reporting the defect where there really is no problem at all because there is no possible execution path that leads body to be dereferenced when NULL.

False positives cause developers to lose trust in a tool. Why? Because the tool is wrong, and if it's wrong far more often than it's right, eventually the user won't trust the tool at all. Fortunately, the techniques available for reducing false positives in C/C++ analysis translate rather nicely into the Java space. We simply provide additional checkers to search for false paths through the code – paths that can never be executed when the program is running. These additional checkers keep track of data flow in different ways, and anytime they find a path that can't be executed, it is pruned from the analysis. This false path pruning is a key way to significantly reduce the rate of false positives reported.

There are a few differences in analysing C/C++ code versus Java code. Unlike C/C++, Java affords us more luxury in choosing which code to analyze. We chose to analyse byte code instead of source code. There are tremendous advantages to looking for defects at the byte code level. The biggest, of course, is the fact that the code has already been compiled – you don't have to deal with compiling the code and juggling the many different flavors of build systems out there. The disadvantage (if you can call it that) of analysing byte code instead of source code is that you need some way to tie the errors you find back to the source code. This means that the byte code needs to have debugging

Listing 1:

```
/org/apache/struts/taglib/bean/DefineTag.java
171         if (body != null) {
172             body = body.trim();
173         }
174
175         if (body.length() < 1) {
176             body = null;
177         }
```

Listing 2:

```
/org/apache/struts/taglib/bean/DefineTag.java
Event branch_null: this.body is null
At conditional (2): this.body != null taking
false path
171         if (body != null) {
172             body = body.trim();
173         }
174
Event deref_while_null: this.body derefer-
enced while null
175         if (body.length() < 1) {
176             body = null;
177         }
178     }
```

Listing 3:

```
170         body_tracker = 5;
171         if (body != null) {
172             body = body.trim();
173         } else {
174             body_tracker
= 12;
175         }
176
177         if (body_tracker != 12 &&
body.length() < 1) {
178             body = null;
179         }
```

Listing 4:

```
170
171         if (body != null) {
172             body = body.trim();
173             body_tracker
= 12;
174         }
175
```

symbols in it or the errors you produce won't be of much help in actually fixing the code. The types of defects that you look for are also different. Defects in Java code have different runtime implications than their C/C++ counterparts. A NULL pointer dereference throws an exception in Java and crashes your system in C/C++. A resource leak in C/C++ happens any time heap-allocated memory is not freed, but in Java, resource leaks occur in different circumstances – when cleanup must be done on an object that the garbage collector cannot be responsible for.

One key feature of the most powerful static code analysis solutions is their ability to understand what happens when one method calls another. This not only helps in finding more complex defects in the code, it also reduces the false positive rate because analysis mistakes are less likely. However, the analysis of Java introduces a challenge in this regard because virtually every method call is, er -- virtual. This means that it's not so clear which instruction a virtual method call will jump to when the code is being analysed. It depends on the runtime type of the object invoking the method. While this is a problem in C++ as well, it tends to be less systemic due to the fact that most people developing C++ code (a) do not always use objects in their code and (b) do not make all their methods virtual.

To tackle this problem with a practical code analysis tool, we have developed techniques to infer the correct types of objects at runtime to determine which virtual methods could be instantiated at any given call site. Of course, our technology must make the appropriate trade offs to retain as much precision as possible while still scaling to analyse large real Java systems. There is some great research out there to discuss the academic techniques from which we

draw our ideas for implementing this in the real world. If you are interested, check Google for "Rapid Type Analysis" or "Class Hierarchy Analysis."

As I mentioned earlier, false positives are the number one challenge for static analysis. The second challenge, and unfortunately a harder problem to deal with, is noise. How is noise different from a false positive? Noise is any issue reported by the analysis that, while technically correct, you just don't care about. It's obvious why this is so hard – it's completely subjective! Yet it's very important to address this in order to produce useful results.

Take a look at listing 4. Notice that on line 173 there's an extra space before the statement. Your static code analysis tool could report that extra space as a defect, but I'm willing to bet that most of us would consider that noise. The analysis isn't wrong, per se – the statements don't line up – but I just don't care. Sure, this example is extreme, but there are less extreme cases that can be equally frustrating – even within checkers for things like NULL pointer exceptions. I've heard developers say "sure, but if that happens, we're totally hosed anyway, so it doesn't matter that it throws an exception

there!" So the analysis can be spot on, producing an actual defect that could occur, but it's still reporting noise. There's no silver bullet for eliminating noise since it is so subjective, but this brings me back to my initial point about the risk of changing your code.

The purpose of a static code analysis tool, whether for C/C++ or for Java, is to help you find defects that would hurt the most, and to find them earlier in the software process. Noisy tools are antithetical to this purpose, as are tools that report too many false positives to gain your trust.

Remember, the purpose of these tools is not to find everything that's bad in your code, and that's a subtle distinction. There is too much risk associated with changing your code to address every little nitpick a static analysis tool can report. So when you're looking to add this type of technology to your arsenal of tools to help you ward off the bugs, take a close look at what bugs it's going after and how it's going after them. More is not necessarily better. Your time is valuable, and you don't want to waste it poring through false positive-ridden and noisy reports. Fortunately, there are tools out there that are on your side. ■