

# **NAVIGATING THE ROCKY SHOALS OF SOFTWARE COPYRIGHTS**

**SEAN CRANDALL  
Jackson Walker, LLP  
112 E. Pecan Ste. 2400  
San Antonio, TX 78205**

**State Bar of Texas  
Advanced Intellectual Property Law  
February 15, 2013  
Austin, TX**

## **CHAPTER 17**

**Contents**

- I. Introduction .....1
- II. In Which Our Hero is Dashed on the Rocky Shoals of Software Coypright 1
  - A. The dream case walks through your door. .... 1
  - B. What are the rocky shoals?..... 1
- III. Software Programming for Dummies and Lawyers1
  - A. Computers are stupid, but they’re really good at repetition.....2
  - B. Most programs are written in high-level languages. ....2
  - C. Procedures encapsulate complexity. ....3
  - D. Libraries let you use other people’s code. ....4
  - E. Application Programming Interfaces are the bridge between libraries and new code. ....5
  - F. Programs can be statically or dynamically linked. ....5
- IV. Google’s Braving of the Shoals in *Oracle v. Google*6
  - A. Google wanted Java for Android.....6
  - B. Oracle’s sued over 37 Java packages. ....7
  - C. The jury found that Google copied Java’s APIs. ....7
  - D. The Court held that APIs are not protectable. ....7
  - E. *Oracle* affects Android’s “scrubbed” Linux header files. ....8
  - F. A cautionary tale about programmers’ utility libraries. ....9
- V. Non-Literal Copying in Other Cases .....10
  - A. Creative structure and sequence in *Whelan Associates*. .... 10
  - B. Abstraction-Filtration-Comparison in *Altai*. ....11
  - C. Interoperability and fair use in *Sega v. Accolade*..... 12
  - D. Command hierarchies in *Lotus v. Borland*..... 12
- VI. The Rocky Shoals of “Free” Software.....13
  - A. “Free” is not a sticker price..... 13
  - B. A brief introduction to hardware drivers. .... 13
  - C. Dynamic linking of proprietary drivers in Linux is controversial. .... 14
  - D. Transitory modifications are not derivative works in *Galoob I* and *Galoob II*..... 15
  - E. Who’s going to sue over violating open source licenses? ..... 16
- VII. Conclusion .....17

## NAVIGATING THE ROCKY SHOALS OF SOFTWARE COPYRIGHT

### I. Introduction

Your client's own code is solid ground, but there are vast seas of code owned by others. The traditional method of bridging the two is to negotiate safe passage (a license). This paper explores some cases where your clients may brave the rocky shoals and gain the benefit of another's code without permission.

### II. In Which Our Hero is Dashed on the Rocky Shoals of Software Copyright

#### A. The dream case walks through your door.

You're sitting in your office on a typical Tuesday morning, working on yet another run-of-the-mill case, and wishing against odds that something interesting would walk through your door. Just then, your secretary buzzes you and says there's a potential client here to see you.

In walks Polly Programmer, owner of a successful software company. Your ears perk up as she explains that Gazillion, a Fortune 500 company, has been infringing her copyrights in their nearly-ubiquitous "Robot" mobile operating system.

Then Polly drops some printouts on your desk. She shows you page after page of comparisons between her code and Gazillion's code. Line after line, it is an identical match. Sure, Gazillion played clever and removed the comments, but the functional part of the code is the same. Why, they even used the same variable names!

Polly also assures you that she registered her copyrights before Gazillion started infringing, so you know the case is eligible for attorneys' fees. And there's no doubt that Gazillion can pay the judgment. You normally don't do contingency cases, but with a head full of private jets and tropical escapes, you sign Polly up for 33% plus expenses.

Two years later, a jury has found that Gazillion most definitely copied thousands of lines of Polly's source code. But the judge has found that this copying didn't constitute copyright

infringement. Now you're taking the smoldering scraps of your case up on appeal, hoping for a miracle, and cursing the day that Polly Programmer walked through your door.

How did this happen? Where did you go wrong? How can flagrantly copying thousands of lines of source code not be copyright infringement? Or turned around, when and to what extent is it okay for your clients to use somebody else's software without their permission or approval? That's what we're here to find out.

#### B. What are the rocky shoals?

There is, on the one hand, the relative safety of *terra firma*—code that your client owns outright. He or she has the run of this land, and can exploit it at will. On the other hand, there are vast, deep seas of code owned by others. With a seaworthy ship, your client can navigate these in relative safety so long as the owner provides safe passage, such as a license.<sup>1</sup>

But what if the other guy won't agree to a license, or your client just doesn't like terms? Between land and sea lie the rocky shoals. A clever pilot with good soundings and a reliable map can safely navigate between the deep sea and the solid ground, reaping the benefits of both. Great mariners may be able to do so even when the treacherous pass is guarded by the shore batteries of a hostile enemy. But the unwary seaman will run aground and find himself dashed against rocks or pounded by artillery.

The purpose of this paper is to explore those rocky shoals—situations where you may be able to exploit another's copyrighted code without their permission, and discuss the inherent risks and benefits of doing so. I start with a brief primer on how software works. I then provide examples of situations where others have ventured into those tricky waters.

### III. Software Programming for Dummies and Lawyers

I know. It's unusual to find a primer on software programming in a CLE paper. But if you

---

<sup>1</sup> I don't know what the ship represents. Sorry, no analogy is perfect.

find yourself wondering, as you did in 8th-grade algebra, “When am I ever going to use this,” the answer is “in a few pages,” when the ideas and vocabulary I introduce here will be critical to high-stakes, bleeding-edge copyright cases.

**A. Computers are stupid, but they’re really good at repetition.**

Your high-end laptop with its multi-core 64-bit processor and fancy office software is stupid. It doesn’t know how to record music or check stocks or play chess or tally columns in a spreadsheet. In fact, it only knows how to do about a thousand discrete things.<sup>2</sup> And the things it knows how to do are fairly esoteric and individually useless, like “add these two numbers” or “compare these two numbers” or “jump to the supplied memory location if a flag bit is set to 1.” As a term of art, these thousand little tasks are called “instructions.”<sup>3</sup>

What your computer is really good at is executing about a billion of these useless little instructions every second. And billions of instructions executed very quickly, one after another, can look very much like a computer playing chess or tallying numbers in a spreadsheet.

A program (an “.exe” file on a Microsoft Windows machine, for example) is simply a terse list of instructions for the computer to execute (usually looping back on itself, so that the program doesn’t terminate until it is told to).

To do something simple like “add ‘A’ to ‘B’ and store the result in location ‘C,’” a computer might execute the following pseudo-instructions:

```
MOVE INTO REGISTER A VARIABLEA
MOVE INTO REGISTER B VARIABLEB
ADD TO REGISTER A REGISTER B
STORE REGISTER A INTO VARIABLEC
```

---

<sup>2</sup> See, e.g., [http://en.wikipedia.org/wiki/X86\\_instruction\\_listings](http://en.wikipedia.org/wiki/X86_instruction_listings), visited on January 7, 2013.

<sup>3</sup> See [http://en.wikipedia.org/wiki/Instruction\\_\(computer\\_science\)](http://en.wikipedia.org/wiki/Instruction_(computer_science)), visited on January 7, 2013.

The instruction “**MOVE**” in our made-up machine code is represented by a short (probably 8-bit) binary<sup>4</sup> “operation code” or “opcode,”<sup>5</sup> like “10001000”. This is followed by another byte representing REGISTER A, and then the memory address of VARIABLEA, all in cryptic binary. Each opcode activates a particular circuit within the processor that carries out its designated function, retrieving information from or storing it in the designated location as instructed. Computers are very good at chewing through huge numbers of these instructions, over and over, all day long, the end result of which is useful work (or less useful stuff, like gaming and web surfing).

**B. Most programs are written in high-level languages.**

While it’s theoretically possible for a person to write programs in raw binary language,<sup>6</sup> it’s not practical for most non-trivial programs.

Most programmers instead use one of a multitude of “high-level” languages,<sup>7</sup> which provide a layer of abstraction over the raw instruction set. With a high-level language, the programmer who wants to add “A” to “B” and store the result in C can simply write something like:

```
C = A + B;
```

Or if he wants to add A to B only if “D” is true, he can write:

```
if (D)
    C = A + B;
```

---

<sup>4</sup> Binary is a “base-2” number system, in which the only available values are “0” and “1.” In binary, 1 = 1, 2 = 10, 3 = 11, 8 = 1000, 9 = 1001, 10 = 1010, 16 = 10000, and so forth.

<sup>5</sup> See <http://en.wikipedia.org/wiki/Opcode>, visited on January 7, 2013.

<sup>6</sup> In fact I had to do so once, with pencil and paper, on an undergrad final exam.

<sup>7</sup> See [http://en.wikipedia.org/wiki/High\\_level\\_language](http://en.wikipedia.org/wiki/High_level_language), visited on January 7, 2013.

Of course, a computer is stupid (as I mentioned). It no more knows what to do with “C = A+B” than you know what to do with “10001011.” So high-level languages also require an intermediate program called a “compiler,”<sup>8</sup> or “interpreter.”<sup>9</sup>

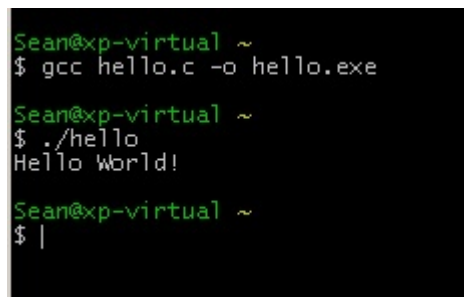
The compiler reads a properly-formatted text file, parses out the commands, and translates those commands into a string of corresponding instructions. The resulting output is called “object code.”

For example, if I am using the venerable “C” programming language, I might type the following in a plain text file called “hello.c”:

```
#include<stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

We will discuss the “include” directive later. Moving to the second line, in C, the main program is always contained in a procedure called “main().” The keyword “int” means that the procedure “main” will return (i.e., output) an integer and “void” means it has no parameters (i.e., input). The body of the procedure is set off by opening and closing curly braces ({}). The “printf()” line tells the computer to print the words “Hello World!” to the screen.<sup>10</sup> Finally, the program returns “0,” meaning that execution was successful.

In a command window, I can now compile with the free<sup>11</sup> “gcc” compiler, telling it to output a new file called “hello.exe,” and run my program as follows:



```
Sean@xp-virtual ~
$ gcc hello.c -o hello.exe

Sean@xp-virtual ~
$ ./hello
Hello World!

Sean@xp-virtual ~
$ |
```

The file “hello.exe” now contains a long string of binary instructions. When I ran the program, the computer flew through those instructions at lightning speed, printing the message on the screen just as it was instructed to do.

### C. Procedures encapsulate complexity.

For tasks more complex than simple arithmetic, most languages include a “library” of pre-defined tasks called “procedures,” “methods,” or “functions” (despite some academic differences, those terms will be used interchangeably here). A procedure is like a miniature program that (ideally) performs a single, well-defined task and (ideally) has a somewhat-descriptive name. Programmers invoke a procedure by typing the procedure’s name in the appropriate place in a program, as I did with “printf()” in my example program.

“printf()” is provided with any standard C compiler, but Programmers are also able to define their own procedures for complex or repetitive task. Programmer can (and should) also build procedures on top of each other. Thus, the main part of a program may in fact make only a few very high-level procedure calls. Those procedures will call lower-level procedures, which will call other, yet-lower-level procedures, and so on. Breaking a program down into several layers of abstraction lets programmers visualize discrete tasks and start with basic building blocks that can be tested and debugged before putting them together one level up. They are also able to re-use code, both within a project and from project to project.

As a practical example that every lawyer can grasp, assume that I have a database of every reported case in American jurisprudence. I am writing a program that aids lawyers in legal research, and my program will frequently pull a

<sup>8</sup> See <http://en.wikipedia.org/wiki/Compiler>, visited on January 7, 2013. The differences between compiled and interpreted languages are beyond the scope of this paper. For simplicity, throughout the remainder, I will assume that programs are compiled.

<sup>9</sup> See [http://en.wikipedia.org/wiki/Interpreter\\_\(computing\)](http://en.wikipedia.org/wiki/Interpreter_(computing)), visited on January 7, 2013.

<sup>10</sup> If you’re wondering, the trailing “\n” is a special command character that means “start a new line.”

<sup>11</sup> See VI.A below for what “free” really means.

case by citation from that database. After receiving the correct citation, I may need to ensure that it's valid and that the case exists before querying my database to pull the case. I may also want to apply some useful formatting to the text of the case.

Rather than re-type the commands every time my program needs to pull a case, I can define a procedure. This procedure will accept three inputs: a volume, a case reporter, and a page number. In return, it will provide a long alphanumeric string with the full, formatted text of the case. My definition of the procedure might look like this:

```
//Pulls a case by citation
//Needs volume, reporter, page
//Returns the text of the case
String FindByCitation(
    int Volume,
    String Reporter,
    int PageNum)
{
    String CaseText;
    boolean valid;
    valid = ValidateCite(volume,
    Reporter, PageNum);
    if (valid)
        CaseText = GetCase(volume,
    Reporter, PageNum);
    else return ERROR;
    FormatCase(&CaseText);
    return CaseText;
} //end of FindByCitation
```

The first three lines (preceded by “//”) are comments, and are ignored by the compiler. Their sole purpose is to make the software more readable by humans. The first comment here says what the procedure is for, the second describes the expected inputs, and the last describes the output.

On the next line, I indicate that my procedure will return a string of text (type **String**).<sup>12</sup> Other procedures might return single characters (“A”), integers (“724”), or floating point numbers (“14.987”). The procedure is named “FindByCitation.” I then indicate the three inputs (called “parameters,”) that my procedure will receive. First is an integer (**int**) representing the

<sup>12</sup> If you're thinking, “Wait a minute! **String** isn't a native C type!” (1) Good for you. (2) Just go with me on it.

reporter volume. Next is a string representing the name of the reporter. Third is another integer representing the page number.

Within the two curly braces I put my miniature program. First, I “declare” variables, which are containers I will use to hold data. One is a **String** called “CaseText.” The other is a **boolean** (true/false variable) that will indicate whether the citation is valid.

Next, I call the procedure “ValidateCite(),” which will return **true** if the citation is valid and **false** if it is invalid. I assign the return value of ValidCite() to the **boolean** variable “valid.” Then I have a test. If “valid” is true, I use GetCase() to query the database and assign the return string to CaseText. If “valid” is false (the “else” part), I return a pre-defined ERROR code.<sup>13</sup>

Finally, I send CaseText to a procedure that formats the text. My last task is to **return** CaseText, or in other words, designate CaseText as my output value and terminate the procedure.

After I have defined it, FindByCitation() acts as a “black box.” Whenever I need my program to find a case by citation, I need only call the procedure.

Notice, however, that FindByCitation() relied on several lower-level procedures. I will also need to define and test those, along with any procedures they rely on. Since it's handy to keep these in one place, I can type them all up in a single text file called “findcases.c.”

## D. Libraries let you use other people's code.

What if instead of writing legal research software, I want to focus on selling my database itself. As an added value, I will provide a set of procedures that programmers can use to easily access the database. But there's a catch: I don't want my customers to have access to “findcases.c” because it contains trade secrets.

In this case, I can use a compiler not to produce a final executable file, but rather the intermediate “object” file “findcases.o.” This file contains all the binary instructions to carry out FindByCitation(), along with other procedures I want to provide to my customers.

<sup>13</sup> After returning the error code, the procedure exits and the rest of the instructions in it are not executed.

My customers' compilers will have to know about `FindByCitation()` so that they recognize it as a valid procedure and so that they know what inputs and outputs to expect. To tell the compiler about procedures I have defined elsewhere, I can use a "header" file "findcases.h." This header file will contain "declarations" of procedures. For example:

```
//findcases.h
String FindByCitation(int Volume,
String Reporter,
int PageNum);
```

When a programmer wants to access to the procedures in "findcases.o," he will **include** findcases.h in his program. The header file does not compile into actual code; it just gives the compiler a heads-up on what to expect. Now he can call `FindByCitation()`. For example, his `main.c` might look like this:<sup>14</sup>

```
#include<stdio.h>
#include<findcases.h>
int main(void)
{
String Lotus;
Lotus = FindByCitation(516, US,
233);
printf("Text of Lotus v.
Borland:\n%s\n", Lotus);
return 0;
}
```

Without a header file, the compiler would not have known about `FindByCitation()`, and would have returned an error.

**E. Application Programming Interfaces are the bridge between libraries and new code.**

A group of files providing related procedures may be referred to as a "library" or a "package."<sup>15</sup>

<sup>14</sup> Of course, realistically he won't know in advance that *Lotus* is the case his user wants. More likely, my program will scan a document for cases, parse out the citations, plug those into the procedure on the fly, and pull each case for the user to review.

<sup>15</sup> Again, these terms will be used interchangeably here.

The library vendor also provides an accompanying "Application Programming Interface,"<sup>16</sup> which includes appropriate header files with declarations of available procedures. Libraries can be provided either with source code or with binary object code, but either way, header files are provided in plain text form.

**F. Programs can be statically or dynamically linked.**

In the above example, `main.c` alone does not have all the instructions to make a full, executable program—some of those instructions are in `findcases.o`. So the programmer will compile `main.c` into an intermediate object file called "main.o." He will then use a program called a "linker"<sup>17</sup> to "link" the two object files together into a single executable file. This is called "static linking." A statically-linked program, in its final form, contains all necessary instructions for executing the program.

But static linking has some disadvantages. For example, if I look at my `hello.exe` program, I find that it is about 44,000(!) bytes long. Assuming for argument that each instruction is one byte long and has two one-byte parameters, that's around 15,000 discrete instructions, which seems like a lot to just print "Hello World!" to the screen.

The problem is that we statically-linked the entire C standard library into our tiny little program.<sup>18</sup> Most of those instructions have nothing to do with our trivial little task.

This can be an even bigger problem in non-trivial software. What if I write 20 programs that all rely on `findcases.o`? That's 20 copies of the exact same code floating around. And if my vendor provides an update to `findcases.o`, I have re-link every one of those programs.

<sup>16</sup> See <http://en.wikipedia.org/wiki/Api>, visited January 7, 2013.

<sup>17</sup> See [http://en.wikipedia.org/wiki/Linker\\_\(computing\)](http://en.wikipedia.org/wiki/Linker_(computing)), visited January 7, 2013.

<sup>18</sup> See [https://blogs.oracle.com/ksplince/entry/hello\\_from\\_a\\_libc\\_free](https://blogs.oracle.com/ksplince/entry/hello_from_a_libc_free), visited Jan. 7, 2013, for an interesting guide to creating a stripped-down version of "Hello World!"

The solution is to use “shared object”<sup>19</sup> or “dynamic link”<sup>20</sup> libraries. These libraries, which come with appropriate “.so” or “.dll” extensions, are never gobbled onto the main program by a linker. Rather, they sit on your disk in their binary form. When a program needs to use `FindByCitation()`, it loads `findcases.so` into memory and then finds and executes the appropriate instructions.

#### IV. Google’s Braving of the Shoals in *Oracle v. Google*

We now turn to perhaps the hottest case in copyright law today, in which Google brazenly copied parts of Oracle’s Java API and still came out on top.

##### A. Google wanted Java for Android.

With plans to eventually launch their own Google-branded smart phone, Google purchased a startup called Android, Inc. in 2005.<sup>21</sup> Android’s main product was a Linux-based operating system for smart phones and tablets.<sup>22</sup>

Google wanted to provide Java, a very popular high-level language, as the main programming environment for Android.<sup>23</sup> To this end, Google started negotiating with Sun Microsystems (Oracle’s predecessor) in 2005, intending to license Java. But negotiations broke down, and no deal was reached.<sup>24</sup>

Google was undeterred. At its core, Java is just a language specification that anybody is free to use, defining for example key words, operators,

syntax, and rules.<sup>25</sup> Theoretically, anybody is free to write his own Java implementation using that specification.<sup>26</sup> And Google had the resources to write its own version of Java.

The problem was packages. As of 2008, Java had 166 packages, containing more than 6,000 discrete methods (Java’s name for a procedure).<sup>27</sup> Those methods were grouped into more than 600 object “classes.”<sup>28</sup>

A seasoned Java programmer would expect to have access to many of these methods when writing an Android program. He would also expect to be able to re-use much of the code he had written for other environments, which would contain calls to popular Sun Java methods.<sup>29</sup> So lacking a license from Sun, Google simply implemented the methods themselves. This gives Android developers access to those methods without encumbering Android with a license from Sun.<sup>30</sup>

The problem is that Google also needed headers to declare the methods in their cleanly-implemented version of Java packages. And since the method names and variable names were identical (by design), the headers are also identical. The copied headers comprised about 3% of Google’s Java implementation.<sup>31</sup> Google

<sup>25</sup> *Id.* at 982.

<sup>26</sup> *Id.* In fact, there are many independent implementations of older languages, like C and Fortran.

<sup>27</sup> *Id.* at 977.

<sup>28</sup> A discussion of object-oriented programming is beyond the scope of this paper. Those who are interested can see [http://en.wikipedia.org/wiki/Object\\_oriented\\_programming](http://en.wikipedia.org/wiki/Object_oriented_programming).

<sup>29</sup> *See id.* at 978.

<sup>30</sup> Using our fictional “`FindByCitation()`” as a concrete example, Google knew that Android programmers would expect to have access to `FindByCitation()` in their programs, and would probably be recycling code that already had calls to `FindByCitation()`. Since they didn’t have a license to Sun’s version, they just wrote their own.

<sup>31</sup> *Id.* at 979.

<sup>19</sup> For Unix-like operating systems.

<sup>20</sup> For Microsoft Windows.

<sup>21</sup> *See* [http://en.wikipedia.org/wiki/Android\\_\(operating\\_system\)](http://en.wikipedia.org/wiki/Android_(operating_system)), visited on January 7, 2013.

<sup>22</sup> *Id.*

<sup>23</sup> *See Oracle America, Inc. v. Google, Inc.*, 872 F.Supp.2d 974, 978 (N.D. Cal. May 31, 2012).

<sup>24</sup> *Id.*



believed that the headers were functional, and therefore not protectable under copyright.

### **B. Oracle’s sued over 37 Java packages.**

In 2010, after acquiring Sun, Oracle sued Google in the Northern District of California<sup>32</sup> over Google’s implementation of 37 Java packages duplicated from Sun Java. Oracle could not, and did not, complain specifically of Google duplicating the *functionality* of the 37 packages. That would be tantamount to claiming a patent in the methods. Rather, Sun’s complaint was over Google’s copying of the API header files.

Oracle conceded that Java itself (as a language) is simply a standard and that anybody is free to implement it. But they argued that there is a “bright line” distinction between the Java language specification, and Oracle’s standard classes and methods, which are copyrighted.<sup>33</sup> Regarding functionality, Oracle’s position was that Google had copied protectable “structure, sequence, and organization”<sup>34</sup> from their Java packages.

### **C. The jury found that Google copied Java’s APIs.**

The court decided to try the lawsuit in three phases: first, the jury would determine copyright issues, then patent issues, and finally damages if any.<sup>35</sup>

In the first phase, the jury would make factual findings on infringement, fair use, and whether the copying was *de minimis*.<sup>36</sup> They were instructed by the court to assume that the APIs were copyrightable, although the Court had

actually reserved that question for later determination as a matter of law.<sup>37</sup>

After a six-week trial, the jury found that Google had infringed by copying the Java API, but deadlocked on the question of whether the infringement was fair use. They also found that Google had literally copied one small snippet of actual code owned by Oracle—a nine-line method called “rangeCheck.”<sup>38</sup>

### **D. The Court held that APIs are not protectable.**

At the outset, the court disagreed with Oracle’s theory of a “bright line” distinction between the Java language specification and the Java packages.<sup>39</sup> Three of the packages were “core” to the Java language, and “anyone free to use the [Java] language itself ... must also use the three core packages in order to make any worthwhile use of the language.”<sup>40</sup>

The Court also made short work of the individual method declarations, noting that those were required by the language specification itself. The only creative elements were the names of the methods, and the names of the parameters. But, the court noted, “names, titles, and short phrases are not copyrightable[.]”<sup>41</sup>

This left only the “structure, sequence, and organization.” The court opened with a discussion of *Baker v. Seldon*,<sup>42</sup> in which Baker sued Seldon over Seldon’s copying a system of double-entry bookkeeping. The Supreme Court held that Baker’s use of the accounting system was not copyright infringement, even though it was copied from Baker’s book. Importantly, the Court introduced the “merger doctrine” of copyright:

---

<sup>37</sup> *Id.* This ensured that if the court was reversed on appeal, the jury’s findings would stand and there would be no need for a retrial.

<sup>38</sup> *Id.*

<sup>39</sup> *Id.* at 982.

<sup>40</sup> *Id.*

<sup>41</sup> *Id.* at 983.

<sup>42</sup> 101 U.S. 99, 25 L.Ed. 841 (1879).

---

<sup>32</sup> *Oracle America, Inc. v. Google, Inc.*, Cause No. 3:10-CV-03561. See 872 F.Supp.2d 974 (N.D. Cal. May 31, 2012).

<sup>33</sup> *Id.* at 982.

<sup>34</sup> *Id.* at 984.

<sup>35</sup> *Id.* at 975.

<sup>36</sup> *Id.*

[W]here the art [a book] teaches cannot be used without employing the methods and diagrams used to illustrate the book ... such methods and diagrams are to be considered as necessary incidents to the art, *and are given therewith to the public* ... for the purpose of practical application.<sup>43</sup>

When Congress revised section 102(b) of the copyright act in 1976, it expressly added a *Baker*-like limit to copyrights.<sup>44</sup>

While admitting that individual names are not copyrightable, Oracle argued<sup>45</sup> that its organization of 6,000 methods into 600 classes, and those 600 classes into 37 packages was a “taxonomy,” protectable under *American Dental Association v. Delta Dental Plans Association*.<sup>46</sup>

But the court disagreed. While Oracle’s selection of package and class hierarchy was creative, original, and similar to a taxonomy,<sup>47</sup> the court found that it was nevertheless a non-copyrightable command structure under 17 U.S.C. § 102(b).<sup>48</sup>

Particularly important to the court’s analysis was that the Java language specification required methods to be called in the form

```
j ava. package. Cl ass. method()
```

(meaning the method “method()” in the class “Class” in the package “package” in the core java language).

The court also noted that “millions of lines of code had been written in Java before Android arrived.”<sup>49</sup> All of these programs invoked core Java methods in the “java.package.Class.method()” format. “In order for at least some of this code to run on Android, Google was required to provide the same java.package.Class.method() command system using the same name with the same ‘taxonomy’ and with the same functional specifications.”<sup>50</sup>

Thus, although Google had essentially copied thousands of lines of code from Oracle, they did not infringe on Oracle’s copyrights.

Both sides have appealed the case to the Federal Circuit.<sup>51</sup>

### E. **Oracle affects Android’s “scrubbed” Linux header files.**

The court’s holding in *Oracle* implicates another controversy surrounding Android. This one involves not a multibillion-dollar corporation but free software activists.

Once again, the issue is that Google wanted its own licensing terms—this time for the C standard library. Android is built on top of the Linux kernel, which is licensed under the GNU General Public License (GPL). The GPL is a “strong copyleft” license,<sup>52</sup> ensuring that source code is available not only for the original work, but also for any derivative works.

Google wanted to license parts of Android under the more permissive BSD license.<sup>53</sup> This means that they would be providing code to their customers, but would permit those customers to keep their own changes proprietary.

<sup>43</sup> *Oracle*, 872 F.Supp.2d at 985, quoting *Baker v. Seldon*, 101 U.S. 99, 103 (1879) (emphasis added).

<sup>44</sup> See 17 U.S.C. § 102(b) (“In no case does copyright protection for an original work of authorship extend to any idea, procedure, process, system, method of operation, concept, principle, or discovery, regardless of the form in which it is described, explained, illustrated, or embodied in such work”); see also *Apple Computer, Inc. v. Microsoft Corp.*, 35 F.3d 1435, 1443 n.11 (9th Cir. 1994).

<sup>45</sup> *Oracle*, 872 F.Supp.2d at 999.

<sup>46</sup> 126 F.3d 977 (7th Cir. 1997).

<sup>47</sup> *Oracle*, 872 F.Supp.2d at 999.

<sup>48</sup> *Id.*

<sup>49</sup> *Id.* at 1000.

<sup>50</sup> *Id.*

<sup>51</sup> *Oracle America, Inc. v. Google, Inc.*, No. 13-1021 and 13-1022, (Fed. Cir. October 19, 2012).

<sup>52</sup> See VI.A below.

<sup>53</sup> See <http://www.zdnet.com/blog/burnette/patrick-brady-dissects-android/584>, visited on January 7, 20133

The standard C library, like the Java packages discussed above, provides a number of procedures that programmers expect to find in a C environment, and in fact that form part of the ANSI C specification.<sup>54</sup> Most Linux distributions use an implementation called “glibc” (GNU C Library). Like the kernel, glibc is licensed under the GPL.

Once again, Google wrote its own implementation of the standard library, calling it “Bionic.”<sup>55</sup> But once again, there was an issue with headers. The C library needs to interface with the kernel, and the header files for procedures provided by the kernel are licensed under the GPL. Google could not provide those headers, or a derivative work of those headers, without binding themselves to the GPL.

Google’s solution was to download a copy of the kernel headers and run them through its own in-house program that scrubbed out comments, copyright notices, and other non-essential information.<sup>56</sup> It then distributed Bionic, along with the modified headers, under its preferred BSD license.

A small storm erupted<sup>57</sup> and a Boston-based attorney penned a piece<sup>58</sup> for the Huffington Post opining that the header files could be a compliance problem for Google. But others

weighed in, including Linus Torvalds, the original author of Linux, and Richard Stallman, the notorious free software activist, countering that they believed the scrubbed header files were not a problem.<sup>59</sup> If the Northern District of California’s holding in *Oracle* survives appellate review, they would seem to be right. The eventual holding will be directly on-point for Bionic and the “scrubbed” Linux header files.

## F. A cautionary tale about programmers’ utility libraries.

After all the high-stakes fighting over thousands of Java methods that Google implemented independently, there was one bit of Oracle’s code that Google literally infringed. Although it’s not really on-point for this paper, it’s worth mentioning briefly because it implicates a common practice for employee-programmers.

After testing 15 million lines of Google code, Oracle managed to find nine lines that appeared to be a verbatim copy of an Oracle “rangeCheck()” method.<sup>60</sup> There was nothing exciting or special about rangeCheck(). It just checked the range on a list so it could be sorted. But it was most definitely copied from code owned by Oracle. Recognizing rangeCheck() as a problem, Google promptly rewrote it for the next version of Android.<sup>61</sup>

But how did those embarrassing nine lines of code end up in Java? They came from Dr. Joshua Bloch, who from 1996 to 2004 had worked as a software engineer at Sun. In 2004, he went to Google to become its “chief Java architect,” and worked on Android for about a year.<sup>62</sup> While working on Android, he rolled some old code into one of the Android packages, including those nine lines of code he had written while at Sun.<sup>63</sup>

<sup>54</sup> See

[http://en.wikipedia.org/wiki/C\\_standard\\_library](http://en.wikipedia.org/wiki/C_standard_library) and [http://en.wikipedia.org/wiki/Ansi\\_C](http://en.wikipedia.org/wiki/Ansi_C), visited January 7, 2013. The specification itself can be purchased from [http://www.iso.org/iso/iso\\_catalogue/catalogue\\_tc/catalogue\\_detail.htm?csnumber=57853](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853), visited January 7, 2013.

<sup>55</sup> See [http://en.wikipedia.org/wiki/Bionic\\_\(software\)](http://en.wikipedia.org/wiki/Bionic_(software)), visited January 7, 2013.

<sup>56</sup> See

[http://www.theregister.co.uk/2011/03/29/google\\_and\\_roid\\_and\\_the\\_linux\\_headers/](http://www.theregister.co.uk/2011/03/29/google_and_roid_and_the_linux_headers/), visited on January 7, 2013.

<sup>57</sup> See, e.g.,

<http://www.fosspatents.com/2011/03/googles-android-faces-serious-linux.html>, visited January 7, 2013.

<sup>58</sup> <http://www.huffingtonpost.com/edward-j-naughton/googles-android-contains-b-836697.html>, visited January 7, 2013.

<sup>59</sup> See

[http://www.theregister.co.uk/2011/03/29/google\\_and\\_roid\\_and\\_the\\_linux\\_headers/](http://www.theregister.co.uk/2011/03/29/google_and_roid_and_the_linux_headers/), visited on January 7, 2013.

<sup>60</sup> *Oracle America, Inc. v. Google, Inc.*, 872 F.Supp.2d 974, 983 (N.D. Cal. May 31, 2012).

<sup>61</sup> *Id.*

<sup>62</sup> *Id.*

<sup>63</sup> *Id.*

The court characterized Oracle’s position on the issue as “overblown,”<sup>64</sup> but the incident is instructive because *programmers do this all the time*. In my personal interaction with employed programmers, I have found that many write their own little libraries of useful procedures for tasks they find themselves repeatedly running into: things like formatting a date, or sorting some numbers, or validating strings. In fact, when in a previous life I worked as a programmer for a major defense contractor, I had my own library of useful little procedures.

Employers do not specifically task programmers with creating these little utility libraries, and in fact, the employers usually don’t even know or care that they exist. But these little utility libraries are works “prepared by an employee within the scope of his or her employment,”<sup>65</sup> and are therefore works for hire owned by the employer.

The problem is that most programmers don’t think of them that way.<sup>66</sup> They treat the utility libraries as their own personal property and (even worse), carry them from employer to employer, adding to them and improving them along the way.<sup>67</sup> I have no way of knowing how many of these little utility libraries exist in the world, but it is certain that they are rife with sticky copyright issues. A programmer’s first employer almost certainly owns the copyright in the original utility library, but as the library evolves over the course of three or four employers, it becomes a derivative work (of a derivative work, of a derivative work...). It would unquestionably be expensive to pay lawyers to fight over ownership of one of these libraries if it ever mattered.

As a practical matter, little utility libraries comprise only a minuscule portion of any non-trivial code base, and if one is discovered in litigation, Google’s solution here is the best:

---

<sup>64</sup> *Id.* at 982

<sup>65</sup> 35 U.S.C. § 101.

<sup>66</sup> I can’t prove this by citing to a case. It’s just based on my personal experience speaking with other programmers and reading what they write on internet blogs.

<sup>67</sup> I didn’t do this with my library. I went straight from my work with the defense contractor to law school.

immediately get rid of it and have a different programmer re-write the affected procedures.

But as a prophylactic, it’s best for your clients to have a firm, written policy about such libraries. Employees should be instructed that *any* utilities, no matter how trivial, developed while in your client’s employ are owned by the client. They should also be instructed to discard any libraries they developed while working for another employer, preferably as part of new-hire orientation.

Expect to meet some resistance and grumbling. Programmers hate losing work and “reinventing the wheel.” But some grumbling on the front end can save a lot of attorneys’ fees on the back end.

## V. Non-Literal Copying in Other Cases

Before coming to its conclusion in *Oracle*, the Northern District of California thoroughly discussed other cases dealing with non-literal copying, some of which are worth mentioning here.

### A. Creative structure and sequence in *Whelan Associates*.

In *Whelan Associates, Inc. v. Jaslow Dental Laboratory, Inc.*,<sup>68</sup> Jaslow had originally contracted with Whelan for Whelan to write a program to manage Jaslow’s dental practice. Jaslow later decided to break ties with Whelan, and wrote his own version of the program in a different high-level language.<sup>69</sup> Importantly, there were no allegations of direct copying of either source code or object code.<sup>70</sup> Rather, Whelan charged, and the district court found, that Jaslow had copied the “overall structure” of her code. Specifically:

The programs were similar in three significant respects .... [M]ost of the [(1)] file structures, and the

---

<sup>68</sup> 797 F.2d 1222 (3rd Cir. 1986).

<sup>69</sup> *Id.* at 1225 – 7.

<sup>70</sup> *Id.* at 1233.

[(2)] screen outputs, of the programs were virtually identical .... [And] five particular [procedures] within both programs ... performed almost identically in both programs.<sup>71</sup>

Thus, despite the important difference that there was no claim of literal copying of even header files, the claims of duplicated structure were nearly identical to those in *Oracle*.

The Third Circuit found that “the purpose or function of a utilitarian work would be the work’s idea, and everything that is not necessary to that purpose or function would be part of the expression of the idea.”<sup>72</sup> On this theory, the Third Circuit found that Jaslow had infringed Whelan’s copyright because “there were many ways to perform the same function ... with different structures and designs.”<sup>73</sup>

### B. Abstraction-Filtration-Comparison in *Altai*.

In *Computer Associates International, Inc. v. Altai*,<sup>74</sup> CA’s competitor Altai hired away one of CA’s programmers, who took with him an unauthorized copy of some important source code.<sup>75</sup> This code found its way into Altai’s competing product, whereupon CA registered their copyrights and sued Altai. Altai immediately took the programmer off of the project and had other programmers who had never seen the offending code rewrite the relevant procedures. CA maintained its lawsuit on both the old version (with literal copying) and the new, rewritten version.<sup>76</sup>

---

<sup>71</sup> *Id.* at 1228.

<sup>72</sup> *Id.* at 1236.

<sup>73</sup> *Oracle America, Inc. v. Google, Inc.*, 872 F.Supp.2d 974, 988 (N.D. Cal. May 31, 2012), citing *Whelan*, 797 F.2d at 1238.

<sup>74</sup> 982 F.2d 693 (2d Cir. 1992).

<sup>75</sup> See section IV.F above for other hazards of programmers carting around rogue source code.

<sup>76</sup> *Altai* at 698 – 700.

The Second Circuit criticized the Third Circuit’s bright line between idea and expression in *Whelan* as being “conceptually overbroad.”<sup>77</sup> In its stead, the Second Circuit established a three-step test of (1) abstraction, (2) filtration, and (3) comparison.<sup>78</sup>

In the abstraction step, “a court should dissect the allegedly copied program’s structure and isolate each level of abstraction contained within it.”<sup>79</sup> For example, at the lowest level of abstraction, the program is a series of individual binary instructions. Using Java as an example, individual methods would be another level of abstraction, classes (each containing a variety of variables and methods) are one level up from that, and packages (each containing a variety of classes) would be yet one level up. At the highest level of abstraction is the ultimate purpose of the program.

The filtration step is where the real work begins. Here, protectable expression must be separated from non-protectable ideas.

This process entails examining the structural components at each level of abstraction to determine whether their particular inclusion at that level was “idea” or was dictated by considerations of efficiency, so as to be necessarily incidental to that idea; required by factors external to the program itself; or taken from the public domain and hence is nonprotectable expression.<sup>80</sup>

Returning, for example, to Sun Java, the selection of the particular “taxonomy” was driven by the language specification itself, and therefore was not protectable.

Finally, at the comparison step, the non-filtered elements of the accused work are compared to their corresponding structures in the allegedly-infringed work. This step involves the

---

<sup>77</sup> *Id.* at 705

<sup>78</sup> *Id.* at 706.

<sup>79</sup> *Id.* at 707.

<sup>80</sup> *Id.*

familiar steps of evaluating substantial similarity and importance.<sup>81</sup>

In *Gates Rubber Co. v. Bando Chemical Industries, Ltd.*,<sup>82</sup> the Tenth Circuit adopted and approved of the abstraction-filtration-comparison test, and also defined a *scenes a faire* doctrine for computer software.

### C. Interoperability and fair use in *Sega v. Accolade*.

In 1991, Accolade decided to start publishing games for the popular Sega Genesis console, but did not like Sega’s requirement that Sega be the sole distributor for all Genesis games.<sup>83</sup> So rather than take a license from Sega, Accolade decided to reverse engineer existing Genesis games to figure out how to make a game compatible.

Accolade decompiled three Genesis games and using information gathered from the process, wrote a manual for developing Genesis-compatible games. Accolade then went on to produce several titles for the console.<sup>84</sup>

Sega sued Accolade for copyright infringement, and on appeal the Ninth Circuit framed the issue as “whether the Copyright Act permits persons who are neither copyright holders nor licensees to disassemble a copyrighted computer program in order to gain an understanding of the unprotected functional elements of the program.”<sup>85</sup> One critical point was that Accolade had admittedly made unauthorized copies of the three games as an *intermediate* step in the reverse-engineering process.

Treating the “intermediate copying” question as one of first impression, the Ninth Circuit found that creating an unauthorized copy of object code as an intermediate step in reverse engineering may be an act of infringement, “regardless of

whether the end product of the copying also infringes those rights.”<sup>86</sup>

The court also rejected Accolade’s argument that copying and reverse engineering object code for the sole purpose of interoperability was an exception to the copyright statute. The court noted that “Accolade’s argument ... is, in essence, an argument that object code is not eligible for the full range of copyright protection”<sup>87</sup>—a proposition that the court rejected, noting that the ideas and functional concepts underlying many types of software are “readily discernible without the need for disassembly[.]”<sup>88</sup>

So even though Accolade’s final product did not directly infringe on any of Sega’s games, Accolade’s sole remaining defense was fair use. After discussing the familiar four-factor test, the Ninth Circuit went a step further and held that, as a matter of law, “where disassembly is the only way to gain access to the ideas and functional elements embodied in a copyrighted computer program and where there is a legitimate reason for seeking such access, disassembly is a fair use[.]”<sup>89</sup>

### D. Command hierarchies in *Lotus v. Borland*.

The controversy in *Lotus Development Corp. v. Borland International, Inc.*<sup>90</sup> was over command hierarchies. Lotus was a very popular early spreadsheet program, and included 469 user-accessible commands, accessible via more than 50 menus and submenus.<sup>91</sup>

Borland built a competing product with “enormous innovations,” but copied the Lotus command hierarchy almost verbatim. The outcome of this case will not be a surprise to any

---

<sup>81</sup> *Id.* at 710.

<sup>82</sup> 9 F.3d 823 (10th Cir. 1993).

<sup>83</sup> *Sega Enterprises Ltd. v. Accolade, Inc.*, 977 F.2d 1510 (9th Cir. 1993).

<sup>84</sup> *Id.* at 1514 – 1515.

<sup>85</sup> *Id.* at 1514.

---

<sup>86</sup> *Id.* at 1519.

<sup>87</sup> *Id.* at 1520.

<sup>88</sup> *Id.*

<sup>89</sup> *Id.* at 1527 – 8.

<sup>90</sup> 49 F.3d 807 (1st Cir. 1995).

<sup>91</sup> *Id.* at 809. Menu options are organized into “ribbons” on some modern interfaces, but the basic functionality is still the same as it has been for decades.

modern computer user, who expects as a matter of course to find some type of menu labeled “File” near the upper left corner of the screen, and for this menu to include options like “Save,” “Save As,” “Open,” “Close,” “Print,” and “Exit.”

But in fact, without the past twenty years of hindsight, the district court found that Lotus’s command hierarchy was a protectable expression.<sup>92</sup> The First Circuit disagreed.

We think that “method of operation,” as that term is used in § 102(b), refers to the means by which a person operates something.... Thus, a text describing how to operate something would not extend copyright protection to the method of operation itself .... Similarly, if a new method of operation is used rather than described, other people would still be free to employ or describe that method.<sup>93</sup>

Thus, the arrangement of menu options is probably not protectable by copyright.

## VI. The Rocky Shoals of “Free” Software

### A. “Free” is not a sticker price.

When you hear somebody refer to “Free Software,” the worst you can do is mistake “free” for a sticker price or think that it means “free to do whatever you want.” Free Software may or may not come free of charge, but software that meets the Free Software Definition<sup>94</sup> always comes heavily encumbered.

The most popular Free Software license is the GNU General Public License (GPL).<sup>95</sup> A key requirement of the GPL is that source code must

<sup>92</sup> *Id.* at 811.

<sup>93</sup> *Id.* at 815.

<sup>94</sup> <http://www.gnu.org/philosophy/free-sw.html>, visited January 7, 2013.

<sup>95</sup> For a more extensive discussion of the GPL, see “A Practical Guide to the GNU GPL,” <http://www.jw.com/publications/article/1453>.

be made available to all licensees on demand, and that any derivative work of a GPL program must also be licensed under the GPL. So GPL software cannot be appropriated into proprietary software, because your competitor could acquire a copy and you would be required to provide them with your source code on demand. This provision is known as “copyleft.”<sup>96</sup>

In contrast, there are many “Open Source”<sup>97</sup> licenses that do not meet the Free Software Definition and that permit proprietary extension of the software without disclosing source code. This is where we get Mac OS X, for example, which is a proprietary extension of the Open Source BSD Unix.<sup>98</sup>

### B. A brief introduction to hardware drivers.

One of the most well-known pieces of Free Software is the Linux kernel. A kernel<sup>99</sup> interfaces directly with the hardware on a computer. Applications run “on top of” the kernel, meaning that they need the kernel to work properly. Among other things, the kernel provides an “abstraction layer” over the hardware, so that applications can issue hardware-independent calls, which the kernel translates into hardware-dependent instructions.

For example, a programmer may want to use the procedure “drawline(),” which takes as inputs two (x,y) coordinate pairs and a color. Predictably, this procedure draws a line between the two points in the specified color. The procedure call must be passed to a video card, which is the piece of hardware that makes stuff show up on the screen.

<sup>96</sup> <http://www.gnu.org/copyleft/>, visited January 7, 2013.

<sup>97</sup> See <http://opensource.org/osd-annotated>, visited January 7, 2013 for the “Open Source Definition.” If you are ever bored at a party full of computer geeks, try starting a debate over Free Software vs. Open Source Software.

<sup>98</sup> See [http://en.wikipedia.org/wiki/OS\\_X#History](http://en.wikipedia.org/wiki/OS_X#History), visited January 7, 2013.

<sup>99</sup> See [http://en.wikipedia.org/wiki/Kernel\\_\(computing\)](http://en.wikipedia.org/wiki/Kernel_(computing)).

The problem is that computers can have lots of different video cards, and an nVidia GeForce GTX 690 may need a completely different set of instructions from an AMD Radeon HD 7870 (or from a GeForce GTX 680, for that matter). And it would be extremely cumbersome to write different software for every possible hardware combination.

So your software doesn't know or care that you're using a GeForce GTX 690. It just calls

```
drawline(0, 0, 100, 300, RED);100
```

and passes execution off to the kernel.

The kernel, for its part, recognizes `drawline()` as a procedure to be executed on the video card, and passes the procedure to the video driver.

The video driver includes hardware-specific implementations of a standard set of software tasks. Thus whether the driver is for an nVidia GeForce card or an AMD Radeon card, it will provide a set of common procedures, such as `drawline()`, and carry out those procedures in hardware-specific ways.

In a closed-source environment like Microsoft Windows, drivers are always linked dynamically. But in an open-source environment like Linux, it's possible (though usually unnecessary) to statically-link drivers into the kernel.

### C. Dynamic linking of proprietary drivers in Linux is controversial.

Problems arise when for-profit companies like AMD and nVidia run up against the "information wants to be free" culture of the Free Software Foundation.

The Linux kernel is licensed under the GPL. But some companies want to keep proprietary secrets about how their hardware works. They fear that by releasing open source versions of their drivers, they would be giving away a competitive advantage.

What to do? One option, of course, is to simply not support Linux, and that's what many vendors choose. In that case, volunteer

---

<sup>100</sup> e.g., starting at location (0,0), draw a line to (100,300) of the color RED.

programmers may step up and try to reverse engineer the hardware to come up with a working driver. Results are mixed, but as a general rule, it's nearly impossible to do as well as somebody who has access to full hardware specifications. And if features are missing from an open source driver, they're usually the hardware's best (i.e., most proprietary) features. Unable to take full advantage of those features, free software enthusiasts may shun your hardware.<sup>101</sup>

Another option is to just provide a proprietary, pre-compiled driver that dynamically links to the kernel<sup>102</sup>—which is exactly what companies like nVidia and AMD do, to the chagrin of free software "purists."<sup>103</sup>

The Free Software Foundation, which wrote the GPL, unequivocally maintains that dynamically linking to a library creates a derivative work, and therefore the resulting software must be released under the GPL. For example, from their FAQ page:

[Q:] If a library is released under the GPL ... does that mean that any software which uses it has to be under the GPL or a GPL-compatible license?

[A:] Yes, because the software as it actually runs includes the library.<sup>104</sup>

The FSF is "merely" the author of the GPL, however, not the court or last resort. And their bold, unqualified "Yes" is largely driven by political concerns; they want the GPL to attach

---

<sup>101</sup> Most people, of course, just buy a computer and use whatever hardware and software come with it and never worry about any of this.

<sup>102</sup> Other cases include providing a closed-source "binary blob" embedded within the source code, where the important functionality is carried out by the binary blob. See [http://en.wikipedia.org/wiki/Binary\\_blob](http://en.wikipedia.org/wiki/Binary_blob).

<sup>103</sup> See, e.g., the mailing list discussion at <http://kerneltrap.org/node/1735>, visited January 7, 2013.

<sup>104</sup> <http://www.gnu.org/licenses/gpl-faq.html#IfLibraryIsGPL>, visited January 7, 2013.



itself to everything possible, because they believe that all software should be free.<sup>105</sup>

Linus Torvalds, the original author of Linux, also weighed in ten years ago, stating that in his opinion, the controlling question is whether the driver was originally written “with Linux in mind,” or whether it was simply adapted from another system.<sup>106</sup> Linus is strongly admired in the free software community, but he is also not the court of last resort, and his personal opinion does not control the legal effect of the GPL.

To frame the issue clearly, the GPL can only attach itself to derivative works,<sup>107</sup> as defined by copyright law. The question, then, is whether *under established legal precedent* a dynamically-linked library is a “derivative work” of a program that links to it.

The answer is we don’t know because there is no precedent directly on point.

#### **D. Transitory modifications are not derivative works in *Galoob I* and *Galoob II*.**

There is, however, a 1992 Ninth Circuit opinion that, with its underlying district court opinion, is rather instructive. Taken together, these are persuasive that dynamic linking does *not* create a derivative work.

In *Lewis Galoob Toys, Inc. v. Nintendo of America, Inc.*,<sup>108</sup> the Ninth Circuit considered the

<sup>105</sup> As any proud beard-wearing free software hippie will tell you, “free” has nothing to do with how much money you paid to get the software. It is a political movement—practically a religion—surrounding four “essential freedoms” you should have *after* you get the software. See

<http://www.gnu.org/philosophy/philosophy.html>, visited January 7, 2013.

<sup>106</sup> <http://kerneltrap.org/node/1735> (Linus Torvalds message of Dec. 3, 2003).

<sup>107</sup> See <http://www.gnu.org/licenses/gpl.html>, visited January 7, 2013, § 0 (“To ‘modify’ a work means to copy from or adapt all or part of the work *in a fashion requiring copyright permission*, other than the making of an exact copy. The resulting work is called a “modified version” of the earlier work or a work “based on” the earlier work”) (emphasis added).

<sup>108</sup> 964 F.2d 965 (9th Cir. 1992) (“Galoob II”).

case of the “Game Genie”—an add-on product designed to work with the Nintendo Entertainment System that allowed players to cheat<sup>109</sup> at video games.

A user would insert Galoob’s Game Genie into the slot on the NES console where a game cartridge would normally go, and then insert a game cartridge into an identical slot on top of the Game Genie, so that the Game Genie stood between the NES console and the game cartridge. In a “configuration” screen, a player could then enter up to three memory addresses, and a new value for each.

For example, a player who knew where the game stored the initial number of “lives” could replace that with a much larger number. Then, when the software accessed that memory address during game play, the Game Genie would return the modified value instead of the value permanently stored on the cartridge.<sup>110</sup> Galoob also produced a “Code Book” with approximately 1,660 “codes” that provided specific cheats.<sup>111</sup>

Nintendo sued Galoob for copyright infringement, alleging that in use the Game Genie created an unauthorized derivative work of the original video games.

The trial court found no infringement, relying on Galoob’s argument that the alleged derivative work was never “fixed” in any tangible medium.

[I]nherent in the concept of a “derivative work” is the ability for that work to exist on its own, fixed and transferable from the original work, i.e., having a separate “form”. See § 101 (derivative work definition). The Game Genie does not meet that definition.<sup>112</sup>

The Ninth Circuit agreed that *protection* of a derivative work required that it be fixed in a tangible medium, but held that *infringement* did

<sup>109</sup> Not to put too fine a point on it.

<sup>110</sup> See *Lewis Galoob Toys, Inc. v. Nintendo of America, Inc.*, 780 F.Supp.2d 1283 (N.D. Cal., July 12, 1991) (Galoob I).

<sup>111</sup> *Id.* at 1289.

<sup>112</sup> *Id.* at 1291.

not require fixation.<sup>113</sup> The court still held, however, that no independent work was created:

The district court's finding that no independent work is created ... is supported by the record. The Game Genie merely enhances the audiovisual displays (or underlying data bytes) that originate in Nintendo game cartridges. The altered displays *do not incorporate a portion of a copyrighted work* in some concrete or permanent form. ... [T]he Game Genie cannot produce an audiovisual display; the underlying display must be produced by a Nintendo Entertainment System and game cartridge. Even if we were to rely on the Copyright Act's definition of "fixed," we would similarly conclude that the resulting display is not "embodied" ... in the Game Genie. It cannot be a derivative work.<sup>114</sup>

Applying this reasoning to dynamically-linked proprietary Linux kernel modules, one can reasonably argue that the binary module similarly does not contain any embodiment of the underlying work (as it would in the case of a statically-linked work) and that the module is not capable, on its own, of producing any effect in the computer. It requires a running kernel and simply modifies certain behaviors of that underlying software.<sup>115</sup>

---

<sup>113</sup> *Galoob II*, 964 F.2d at 968 ("A derivative work must be fixed to be protected under the Act, see 17 U.S.C. § 102(a), but not to infringe.")

<sup>114</sup> *Id.* (emphasis added).

<sup>115</sup> A reasonable counter-argument would be that the two works are "joined" at runtime, creating in memory a new work that embodies both the kernel and the module. I personally lean away from this interpretation as the kernel and the module will not usually reside in a single, contiguous memory block. In dynamic linking, the two pieces of code are more analogous to two ships passing signals to one another than to two seamen on the same ship talking to each other.

Interestingly, under the same analysis infringement may lie where a proprietary *kernel* dynamically links with a GPL *driver*. This scenario is not unknown, but is less common than the proprietary driver example. However, neither has, to my knowledge, been specifically tested in court.

### E. Who's going to sue over violating open source licenses?

One argument sometimes advanced in favor of proprietary drivers is that it doesn't really matter, because free software hippies are (practically by definition) all broke and they're never really going to sue anyway. But the empirical evidence says otherwise.

Perhaps the first meaningful test of an open source license was the Federal Circuit's *Jacobsen v. Katzer*.<sup>116</sup> Jacobsen was a free software hobbyist who made his code available for free download under the terms of the "Artistic License."<sup>117</sup> Katzer developed commercial, proprietary model train software, and used some of Jacobsen's code without following the terms of the license.<sup>118</sup>

When Jacobsen sued, Katzer argued—and the district court agreed—that the Artistic License was an "intentionally broad nonexclusive license [that] has unlimited scope and thus did not create liability for copyright infringement."<sup>119</sup>

Katzer argued that he was at worst guilty of a breach of contract.<sup>120</sup> And since contract actions do not carry a presumption of irreparable harm, Jacobsen was not entitled to a preliminary injunction.<sup>121</sup> Furthermore, Jacobsen was not entitled to money damages because the contract required no money to be paid.<sup>122</sup> The result of this

---

<sup>116</sup> 535 F.3d 1373 (Fed. Cir. 2008).

<sup>117</sup> *Id.* at 1376.

<sup>118</sup> *Id.*

<sup>119</sup> *Id.*

<sup>120</sup> *Id.* at 1377.

<sup>121</sup> *Id.*

<sup>122</sup> *Id.*

argument would be that Katzer had *carte blanche* to use Jacobsen's code as he saw fit.

But the Federal Circuit reversed the district court, finding that the Artistic License was instead a conditional copyright license.

The Artistic License states on its face that the document creates conditions: "The intent of this document is to state the *conditions* under which a Package may be copied." (Emphasis added.) The Artistic License also uses the traditional language of conditions by noting that the rights to copy, modify, and distribute are granted "provided that" the conditions are met.<sup>123</sup>

Thus, "[c]opyright holders who engage in open source licensing have the right to control the modification and distribution of copyrighted material."<sup>124</sup>

Around the same time as Jacobsen, Erik Andersen, the former maintainer of the GPL "Busybox" software started aggressively enforcing his license by suing several commercial infringers.<sup>125</sup> Although these cases have not yielded any reported opinions, the results have been favorable for the GPL, including both money damages and injunctive relief.

## VII. Conclusion

Rocky shoals are not for the faint of heart. If your client is risk averse or particularly conservative, negotiating safe passage is always the safest—if not the most exciting—course.<sup>126</sup> And as a general rule, if you're going to use somebody else's stuff, you have to play by their rules.

---

<sup>123</sup> *Id.* at 1381.

<sup>124</sup> *Id.*

<sup>125</sup> *See* [http://en.wikipedia.org/wiki/BusyBox#GPL\\_lawsuits](http://en.wikipedia.org/wiki/BusyBox#GPL_lawsuits), visited Jan. 7, 2013.

<sup>126</sup> On the other hand, if your client is Horatio Hornblower, he may prefer to blast the enemy's shore batteries with their own powder and sail off with a line-of-battle ship in hot pursuit.

But it's also worthwhile for your clients to know that sometimes they can take advantage of somebody else's work for their own purposes, even without permission. If there is a business benefit to copying a competitor's non-protectable functional ideas, a careful analysis of the case law may prove that the copying is allowable. As the courts have not drawn a bright line between non-protectable function and protectable expression, those cases will have to be examined individually in light of all the facts and circumstances.

## Curriculum Vitae

Sean C. Crandall  
Jackson Walker, LLP  
112 E. Pecan Ste. 2400  
San Antonio, TX 78205  
(210) 978-7700

[scrandall@jw.com](mailto:scrandall@jw.com)

<http://www.jw.com/scrandall>

### Biography

Sean Crandall's practice involves all aspects of intellectual property, including patents, trademarks, copyrights and licensing. He has prosecuted and litigated patents in diverse subject matter areas, including computer science, electronics, wireless communication, medical devices, oil and gas, and electrical and mechanical systems.

Before entering the legal profession, Mr. Crandall spent seven years with a major U.S. defense contractor and completed an undergraduate degree in electrical engineering with an emphasis in computer engineering. His technical expertise and experience includes:

- Software design and development (including C, C++, LabVIEW, Pascal, FORTRAN and Gensym G2)
- Design and troubleshooting of hardware interfaces
- Modeling and simulation of complex systems
- Design and analysis of digital systems (including Verilog HDL)
- Computer architecture (subsystem level to logic level)
- Integration of analog and digital electronics
- Parts and materials engineering (selection, qualification, testing and sustainment) for high-reliability radiation-hardened systems

### Education

J.D., Baylor University School of Law (2007)

Senior Executive Editor, Baylor Law Review (2006 - 2007)

Associate Editor, Baylor Law Review (2006)

B.S. Electrical Engineering, The University of Texas at San Antonio (formal emphasis in Computer Engineering) (2003)

### Bar Admissions

2007, Texas

2005, United States Patent and Trademark Office, Reg. No. 57,776.

### Court Admissions

Western District of Texas

Eastern District of Texas