# DRAFT: Derivative Works Under the GNU General Public License[1]

**Ben Dugan[2]**

## 1. INTRODUCTION

This article discusses the treatment of derivative works under the GNU General Public License. It argues that the functioning of the GPL depends upon a definition of derivative work that is significantly broader than that employed by the Copyright Act or the courts. This over-broad definition ultimately leads to nonsensical results in the face of many modern program development and distribution methodologies. This article visits a number of fact-patterns to help illuminate possible weaknesses and inconsistencies in the GPL definition of derivative work.

This article requires that readers have more than a passing familiarity with computer software system concepts, so readers who are not well-versed in those ideas are urged to first read Section 2, Technology Primer. Other readers can skip to Section 3.

## 2. TECHNOLOGY PRIMER

In this section, we describe and define a number of concepts underlying modern computer systems technology. Understanding these concepts is central to resolving a number of legal issues that arise in the context of open, hybrid, and proprietary models of software development and distribution.

### 2.1 LANGUAGES AND MACHINES

*Machine code* is a sequence of instructions expressed in the native representation of a particular computing machine. By *native*, we mean that the target machine can consume and execute the instructions contained in a particular sequence of machine code with a minimum of effort. A sequence of machine instructions is sometimes called a *program* or *executable*. Machines are generally defined by reference to an abstraction, called an *instruction set architecture*, or *ISA*. The ISA is an *interface* to the machine, meaning that it defines the format (or syntax) and meaning (or semantics) of the set of instructions that a given machine can execute. The syntax and semantics that defines the set of allowable machine codes and their meanings is sometimes called *machine language.*

[1] A version of this paper first appeared in 2006 as part of the Software Pluralism Project at the University of Washington School of Law, available at: http://www.law.washington.edu/lta/swp/index.html.

[2] Opinions expressed herein are those of the author only. 

The decision as to how best implement a given ISA is to a large degree an arbitrary one. Historically, ISAs have been implemented in hardware, meaning that the logic required to consume, decode, and execute machine codes that comply with a given ISA is implemented by means of a fixed circuit. However, this mode of implementation is by no means required. A given ISA may also be implemented by means of a software program, and such an implementation is sometimes called a *virtual machine.* This software program will itself of course just be a sequence of machine codes that can be understood by another machine, that may or may not also be implemented in hardware or software. For instance, it is possible (and in fact common, nowadays) to implement a given machine that can execute programs written in machine language X by writing a program in machine language Y that runs on (is consumed and executed by) a machine that is implemented in hardware.

The good thing about machine languages is that they are typically relatively simple in terms of syntax and semantics. The set of allowable machine instructions is generally small, and the allowable formats for expressing machine instructions are typically few. These limitations make it relatively easy to build (particularly in hardware) a machine that implements a given machine language.

The bad thing about machine languages is the same as the good thing—that is, that they are very simple. This makes them inefficient modes of communication, at least from the perspective of someone who wishes to write programs that do interesting things. Intuitively, a language containing only 100 words will be much less expressive than one containing 100,000 allowable words, because it should take fewer words to express a given concept in the latter (large) language than in the former (small) language. Also, because machine languages are designed to be easily read and decoded by mechanical means, they are often represented by way of alphabets that are not natural for humans to read or write. The most common format used is binary—that is, all machine codes are represented only by strings of ones and zeros. The choice of alphabet is again an arbitrary decision. Historically, however, binary has ruled because it is relatively straightforward to build hardware circuits that function on high and low voltages which map nicely to ones and zeros.

## 2.2 Compilers and Higher Level Languages

These drawbacks with machine code led computer scientists to write programs (initially, painstakingly in machine code) to translate programs written in higher-level programming languages into programs consisting only of machine code. These programs are called compilers. A compiler is just a program that takes as input a code sequence in one language and translates and outputs a corresponding code sequence in second language. That second language will typically be machine code, but it need not be. The second language is often called the *source language* and is typically of a much higher level than machine language. That is, its alphabet is comprised of symbols that are familiar to humans, such as alphanumeric characters; its built-in

commands or instructions will be represented in words (e.g. "ADD") or symbols (e.g. "+") that have conventional meanings; and they provide for the straightforward expression of useful, commonly occurring programming idioms, such as repeated execution or functional decomposition.

## 2.3 MODULES: SOFTWARE DECOMPOSITION

There are two ways to think about program decomposition. First, all modern programming languages provide some mechanism for *logical decomposition*. The core principle behind logical decomposition is the separation of *interface* (sometimes also called *specification*) from *implementation* (sometimes also called *representation*). The interface to a module is everything and anything that a client of that module needs to know to use the module. The interface is a description of *what* the module can do. The implementation of a module, on the other hand, is the inner workings (generally data structures and instructions) that actually perform the work promised by the interface description. Consider, for example, a television. The interface consists of the switches, knobs, and plugs on the outside of the television. The implementation consists of the guts of the television. As users, we need only to understand the interface: how to plug it in, turn it on, and twiddle the knobs. We don't need—indeed, probably don't want—to know how the television works its magic. By decoupling the interface from the implementation, separate parties can work on separate parts of large system by knowing only about agreed-upon interfaces. Television set designers only need to know about the interface to the electrical system—the shape of the plug and the voltage provided—in order to design a television set that any consumer will be able to plug into their wall socket. They do not know or care about how the electricity is generated, where it comes from, and so on.

The most basic form of logical decomposition is *procedural* or *functional decomposition.* Every modern, high level programming language supports functional decomposition in some manner. A function is just an abstraction for a set of instructions to perform a common task, which has been given a convenient name by the programmer. For instance, a programmer may create (define) a function called area that knows how to compute the area of a rectangle. In defining this function, that programmer has in some sense added a new word to the vocabulary of the programming language. The programmer can later invoke the function simply by using that new word in their program. Other forms of logical decomposition include modules, abstract data types, class definitions, and packages. All modern, high level programming languages support one or more of these forms of logical decomposition. While the details of these forms is beyond the scope of the article, they are all methods for creating software components that are larger than single functions or procedures.

In addition to logical decomposition, modern programming languages allow for *physical decomposition*. The most common mechanism for performing this kind of decomposition is by breaking a single large program into a series of files. Each file will

typically contain one or more logical program components. For example, a single file might contain a group of related functions that can then be called from within other source files. Some mechanism for program decomposition is central to the ability to build large programs. Without it, it is hard to imagine how two or more programmers could work on a single program at the same time.

## 2.4 Combining Physical Modules: Static and Dynamic Linking

Program decomposition, however, complicates the process of source to machine code translation. In a world where every program is contained in a single source file, the compiler knows everything it needs to know to translate the program from source to machine code. In a world where a program is broken down into two or more source files, the compiler will not know how to properly translate function invocations that reference functions that are defined in other source files. At a high level, a compiler needs to translate a function invocation into a "jump" to a different instruction address within the executable program. However, the compiler cannot know the machine address of a defined function unless it knows the layout of the entire executable, which it cannot know until all the source files have been compiled and merged.

The typical solution to this problem is simply to defer the resolution of function names to machine addresses to another program, known as a *static linker*. In this scheme, the compiler translates a given source file into a corresponding *object file*. An object file contains machine code, as well as some information about functions and other names that are defined by (or functions and other names that are referenced by) the given object file. A second program, called a *linker,* is then used to make a final resolution of names to numerical addresses. At a high level, the linker takes all of the object files, and for each object file stitches up unresolved references to names that are defined in other object files. After stitching up these names, it concatenates the various machine code sequences into one long machine code sequence, which is now fully executable by the target machine. If it encounters names that are not defined in any object file, it typically reports an error, because this means that the programmer of some object file has attempted to invoke a function or otherwise reference names that are not defined in any object file that comprises the executable.

Static linking works well until we consider that many programs share a considerable amount of standard code. This commonly used, shared code is typically placed into *libraries*, which are seldom changed, well-known repositories for useful functions that are used by a wide variety of programs. Examples include libraries for performing input and output (from keyboards and other input devices and to the screen or other output devices), mathematical functions (such as sine, cosine, logarithms, etc), and so on. One way to think about a library is as a widely-advertised, well-known object file.

Consider what happens when a number of programs are each linked against a given library. First, this approach is wasteful in terms of disk space, because each program contains a section of identical code. While this may not seem significant in today's world of large disks, it does begin to pose a problem on systems with limited storage, where hundreds or even thousands of programs are all statically linked with a given library or libraries. Second, suppose that the vendor wishes to fix a bug in a given library. The vendor can redistribute the library, but then the end-users would have to re-link that library to each of the hundreds or thousands of programs that may use it. Alternately, the vendor can relink the programs for the user, but they may not be in a position to do so if some of the programs are licensed from third parties. And even if the vendor were willing to relink, they would now need to re-distribute and install a large number of programs, rather than just a single library.

The solution to these problems is to employ *dynamic linking*. In a world of dynamic linking, programs are not linked until run-time—that is, external name references are left unresolved until the program is loaded, or even until a given function is referenced for the first time. Dynamically linked programs incur some cost at run time—either due to longer startup times because the linking is done when the program is loaded, or as a small cost each time an unresolved name is referenced for the first time. The benefit of dynamic linking, of course, is that multiple programs can now truly share the same library—each system will only have a single copy of each dynamically linked library shared by a plurality of running programs. For this reason, dynamically linked libraries are sometimes also called shared libraries (in the *N*X world, for instance). Another benefit is that upgrading a dynamically linked library (e.g. due to a bug-fix) is often as simple as just replacing the library on disk, and restarting the system. Because programs are automatically linked each time they are run, no tedious static re-linking of programs is required.

### 2.5 BINDING

Static and dynamic linking are instances of a more general concept known as *binding*. Binding can be thought of as the process of associating, in some way, names with values in a software system. Systems can be distinguished based on when they perform various kinds of binding. In the case of static linking, function names are bound to numerical function addresses more or less at compile time (or more precisely, after compile time, but before run-time). In the case of dynamic linking, binding doesn't take place until run-time.

Internet domain name resolution is another example of the binding concept. The Domain Name Service (DNS) allows textual, easy-to-remember domain names (like "yahoo.com") to be translated into numerical IP addresses at run-time. Deferring binding has the benefit of making systems more flexible at the cost of some run-time overhead. In the DNS example, imagine if domain names were translated at compile time. In that case, anytime someone wanted to re-map a domain name to a new IP

address, all programs that referenced that name would have to be re-compiled. Moreover, a large class of networking programs, such as web-browsers, would be dependent upon relatively static associations between domain names and IP addresses, greatly limiting their utility.

Object oriented programming languages are another example of systems that rely on late binding. One of the hallmark features of object oriented programming languages is *subtype polymorphism*. A programming language that is subtype polymorphic allows a programmer to write code that will function with data objects of a type that is not known until runtime. Object oriented programming languages do not depend on knowing the particular representation of objects at compile time. This means that programmers may write code that is highly abstract, in that it works with many different (concrete) types of objects that all share common interfaces. This flexibility and expressiveness, however, comes at the cost of deferring binding until runtime. Specifically, the process of looking up and determining the correct behavior for a given object of a particular type is deferred until runtime. This process is in many ways analogous to that of dynamic linking that occurs on a "demand basis" (as opposed to dynamic linking that takes place at load-time). It differs from demand-linking, however, because object oriented systems allow for the binding to be different *every time* a particular function is invoked—in other words, the particular behavior for a given object must be (in the worst case) re-determined every time that behavior is invoked (because it is not known if the current object is or will be the same as the subsequent object).

## 3. THE GPL AND DERIVATIVE WORKS

The following section describes some common program development scenarios and their treatment under one of the most commonly used open source licenses, the GNU General Public License (GPL). At a high level, the GPL allows the licensee to copy, modify, and distribute the licensed program under certain conditions. Copying, modification (or the creation of derivative works), and distribution are of course three of the exclusive rights granted to authors under the Copyright Act of 1976. 17 U.S.C. §106. The first GPL requirement is that software distributed under the terms of the GPL be accompanied by source code. The second condition requires that modifications of the original program be distributed under the same terms as the GPL itself. Section 2 of the GPL grants the licensee the right to "modify [their] copy ... of the Program, thus forming a work based on the Program, and copy and distribute such modifications or work" provided that the modified work is also licensed under the terms of the GPL. This latter requirement makes the GPL reciprocal (or viral, depending on the reader's political viewpoint) and guarantees - if it functions as intended - that subsequent versions of an initial code-base remain open or free, where openness here is used to refer to the availability of source code coupled with the right of end-users to make changes to that code. The critical observation is that the requirement that the GPL be applied to a new work is triggered when 1) a derivative work is created and 2) that derivative work is distributed. The efficacy of the GPL depends, therefore, on the

breadth of the definition of derivative work. In the sections that follow we first survey the various definitions of derivative work as employed by the courts, and then analyze the GPL's effect in the face of a variety of commonly occurring software development patterns.

### 3.1 LEGAL BACKGROUND

Under the Copyright Act, a derivative work is defined as "a work based upon one or more preexisting works, such as a translation, musical arrangement, dramatization, fictionalization, ..., or any other form in which a work may be recast, transformed, or adapted. A work consisting of editorial revisions, annotations, elaborations, or other modifications which, as a whole, represent an original work of authorship, is a 'derivative work'." 17 U.S.C. §101. The courts have unfortunately been less than clear and in many cases inconsistent in their interpretation of this provision in the computer systems context. An exhaustive legal analysis of derivative works in the computer system context is beyond the scope of this article, but the three cases (all taken from the video game context) discussed below should provide a feel for the courts' struggles with the concept.

In Midway Mfg. Co. v. Artic International, Inc., the court confronted the derivative work question in the video game context. 704 F.2d 1009 (7th Cir. 1983). In this case, the defendant marketed a circuit board that could be installed to speed up the gameplay of an arcade game manufactured by the plaintiff. The court held, with minimal discussion, that the speeded-up video game was a derivative work of the original video game, and that the defendant infringed the plaintiff's derivative work right.

Almost a decade later, a different court, on very similar facts reached the opposite conclusion. In Lewis Galoob Toys, Inc. v. Nintendo of America, Inc., the defendant developed a hardware add-on module that enhanced the game play of a Nintendo video game system. 964 F.2d 965 (9th Cir. 1992). The court adopted a particularly narrow definition of derivative work: "[it] must incorporate a protected work in some concrete or permanent 'form'." *Id.* at 967. The court held that because the add-on module did not incorporate the protected work (here, the audiovisual video game displays) in any way, it did not infringe Nintendo's right to create derivative works.

The *Nintendo* court distinguished *Midway* by pointing out that the circuit board distributed in *Midway* substantially copied and replaced a portion of the original video game circuitry, whereas the *Nintendo* add-on did not actually incorporate any of Nintendo's protected expression. The court then buttressed this distinction with a market effects test, noting that the *Nintendo* add-on did not actually supplant demand for the Nintendo game console, whereas the plaintiffs in *Midway* were being deprived of potential additional profits reaped by licensees of the arcade video game. Finally, the court made a policy argument, noting that holding the add-on infringing would unduly chill the market for improving innovations, such as plug-in spell checkers for existing word-processing programs.

Finally, in Micro Star v. Formgen Inc., the court held that a distributor of new "levels" for a video game infringed the creator's derivative work right. 154 F.3d 1107 (9th Cir. 1998). At first glance, this case seems again similar to the above. The alleged infringer simply marketed add-on modules, that an end-user could employ to create variations on the original work. So why the different result? The court distinguished *Nintendo* because the levels distributed in this case actually described "sequels" to the original game, whereas the add-on module in *Nintendo* merely enhanced the gameplay. In *Nintendo* the gameplay was still generated by the Nintendo game console whereas in *Formgen* the new levels actually directed a new telling of the video game's story. The court focused especially on the architecture of the *Formgen* game system, which consists of a rendering engine, an image library, and one or more files ("MAP" files) that describe the game levels and their content. The MAP files, therefore, define the very metes and bounds of the video game's story. And since Formgen created the initial video game story with their original MAP files, any other MAP files must tell a related and derivative story—a sequel, in a sense—of the original.

Whether or not we find the analysis in these cases convincing, they do teach us something about the way courts address the derivative work problem. First, a derivative work must incorporate the protected work in some manner. Second, courts will fall back on an analysis that resembles the substantial similarity test familiar under the reproduction right. Third, courts will look at the market impact of the work. An alleged derivative that does not supplant demand for the original is less likely to be held infringing. Finally, courts recognize that an over-broad definition of derivative work may chill the market for follow-on innovations.

### 3.2 MODIFYING A SOURCE FILE

**Example 1.0**: Starting with the most simple case, suppose a programmer, X, has in their possession a copy of program G. G consists of a number of source files (physical modules) and is licensed to X under the GPL. Now suppose that X modifies program G by adding and/or deleting a number of lines of code to one or more of the source files that make up G. In doing so, X creates a new program G'. Given that G is licensed to X under the GPL, will the modified program, G', also be subject to the GPL?

Initially, the answer to this question must be in the negative. Recall that the GPL's demand that it be applied to a derivative work of the original program is only triggered upon the distribution of a work that is derivative of the original program. So even assuming for the moment that G' is a derivative work of G, X does not need to license the modified version in absence of distribution. The *Frequently Asked Questions about the GNU GPL* (hereinafter FAQ) confirms this analysis: "The GPL does not require you to release your modified version. You are free to make modifications and use them privately, without ever releasing them... But if you release the modified version to the public in some way, the GPL requires you to make the modified source code available to the program's users, under the GPL." Therefore, so long as X does not release - and

thereby exercise the distribution right - G' need not be subjected to the terms of the GPL.

**Example 1.1**: Now, assume the same facts as above, but that X now wishes to distribute copies of G', as might a traditional software vendor. Since distribution is given, the analysis now turns on whether G' is a derivative of G. If so, G' must also be licensed under the GPL, meaning that X must provide copies of the source code along with any translation (executable) they may distribute.

This fact pattern is probably the canonical case envisioned by the drafters of the GPL. G' surely qualifies as a derivative of G under the legal framework described above. G' incorporates much if not all of the original work G. G' is certainly substantially similar, given that we are assuming only a small number of lines of code have been added. Finally, G' can be viewed as a replacement for G, thereby denying market share to the author of G. Hence, because X is distributing a derivative of G, X must license G' under the terms of the GPL.

### 3.3 Adding a File to a Collection of Files

**Example 2.0**: Suppose that X modifies the original program G by simply adding a new source file S. S contains some new functionality which X wishes to add to the existing functionality of G. Depending on the architecture and programming language used in G, it may be necessary for X to also make at least a small modification to one the source files in G, in order to invoke the functionality contained within S. In this case, it is useful to think of the resulting work as the sum of its constituent parts. First, S is just the new source file added by X. Second, G' is just the original code-base G, plus some (possibly minor) modification required to invoke functions located in S. Note that G' is similar in many ways to the simple example given above, except that the code added makes one or more external references to functions located in module S. The resulting work can be expressed as G' + S. In order to run their new program, X will need to compile all of the modules in G' + S and link them together to build an executable, E. We will use the notation C(x) to describe the object file that results from compiling a given source file x. For instance, C(S) is the object file that results from compiling source file S. Hence, E = C(S) + C(G').

Assume that E is statically linked, and suppose that X wishes to distribute E. Must E be licensed under the terms of the GPL? Again, if E can be considered derivative of G, then the answer must be yes. Recall that E = C(S) + C(G'). We know from the analysis above that G' is a derivative work of G. The copyright statute tells us that "translations" are considered derivatives of original works. 17 U.S.C. §101. Assuming that the process performed by the compiler qualifies as translation, then C(G') surely qualifies as a derivative work of G'. And finally, since E is statically linked, at the time of distribution it literally incorporates C(G'), which we have determined is derivative of the original work G. E therefore can be considered a derivative work of G and must be distributed under the terms of the GPL, meaning that source code G' and S must be provided.

**Example 2.1**: Now suppose that X wishes to license S (containing the bulk of the new functionality of G' + S) under different terms. In particular, X might want to distribute S under a license that does not entitle the licensee to view the source code in S. X might be able to do this by first distributing just the portion of E defined by E - C(S). In other words, X might choose to distribute just C(G'). This portion of E will likely not be a functioning program, and, under the above analysis it will be a work based on G, and therefore also subject to the GPL. And then suppose that X separately distributes the object module C(S). Is this module subject to the GPL? Must X provide the accompanying source code, S?

The GPL itself admits to the possibility that certain kinds of works, independently distributed, may not be subject to the terms of the GPL. Section 2 of the GPL states: "[The requirement that the modified work be licensed under the GPL] apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works." In our scenario, then, if S can be considered an independent work of G' and if S is distributed separately from G', then S need not be licensed under the terms of the GPL. Given that it is being independently distributed, the answer then depends on whether S can be considered an *independent* work of G'. *Independence* is nowhere defined in the GPL, but it seems reasonable to assume that the drafters meant the term to encompass the set of works that are not derivative of the original work.

The answer depends, then, on whether or not S can be considered a derivative G'. On the one hand, if we can make this case look like *Formgen*, then there might be a good argument for deciding that S is a derivative of G'. For example, suppose that G' is a digital signal processing program, consisting of an engine module that applies a variety of filters to a digital sample. Assume further that a collection of pre-existing filter modules already exists in G'. So the interesting thing about G' is not really its application engine, but its collection of filter modules—just as the interesting thing about the *Formgen* video game was the stories described by the MAP files. Arguably, now, a new filter provided by module S could be viewed as a derivative of at least some of the original filters provided by G'. And while S does not necessarily literally incorporate any portion of G', it is at least substantially similar to some of its components. Moreover, the combination of S and G' could certainly be viewed as a market substitute for G' alone. In this analysis then, S is arguably a derivative of G', and X would need to distribute it under the terms of the GPL.

Even here, however, there are features of copyright law that may stand in the way of this conclusion. The first is the bar on protection for processes and methods of operation in section 102(b). Recall that the MAP files in *Formgen* described the telling of the video game's story. And in holding that the *story* was protectable subject matter, other MAP files - that told variations on that story - were necessarily derivative works. A digital signal processing system, on the other hand, is at its core highly functional. And the filters themselves are written by highly efficient and arguably rote

implementations of mathematical formulas. Therefore, under section 102(b) and its more rigorous manifestation in the Altai Abstraction-Filtration-Comparison test (*Computer Associates International v. Altai*, 982 F.2d 693, 23 U.S.P.Q.2d 1241 (2nd Cir. 1992)), it is doubtful that there is very much copyrightable subject matter in the original digital signal processing system as defined by G'. If G' contains only minimal copyrightable expression, then the argument that S is in some way a derivative becomes even harder to make. The lesson perhaps is that architectural similarities alone are insufficient to show derivation. Here, the question was not resolved by reference to the architectural similarities between the *Formgen* system and our system, but by reference to the strength of the copyrightable expression in the parent system.

The fair use limitation provides another counter-argument to a finding of infringement in this example. The fair use analysis is codified in section 107 of the act, and directs courts to look at four factors: the purpose and character of the use, nature of the copyrighted work, amount and substantiality of the portion used, and the market effect. The first factor weighs against X, assuming he intends to sell C(S), because commercial use typically works against an alleged infringer. The remaining factors, however, may work in X's favor. The second factor restates the principle discussed above, that creative works of authorship are entitled to higher degrees of protection. Here again, the analysis will turn on the substance and strength of copyrightable expression in G'. And if G' is seen as primarily functional, there again will be little copyrightable expression to infringe. The third factor focuses on the amount of the original work used in the allegedly infringing work. Here, assuming that modern software engineering principles are followed, the amount of *literal*copying of code in G' is surely minimal. The relationship between S and G' is just that S makes function calls into G'. Therefore, S makes use just of the function names defined by G', and doesn't actually copy any portions of the G's program text. Finally, the market effect analysis also works in X's favor, because S does not supplant the market for G. In fact, anyone who wishes to use S has to also download a copy of G'.

Of course, even if we decide that S is not a derivative of G' (or any portion of it), the original author may proceed under a contributory infringement theory. Since the executable E is a derivative work of G created by the end user, the owner of the copyright in G could claim that even if S alone is not a derivative work of G, that X is contributorily infringing their derivative work right because they are inducing, causing, or materially contributing to infringing conduct of another. Even assuming that X has the requisite knowledge of the infringing activity, there is a problem with this argument related to the grant of rights under the GPL. Because the GPL grants the end-user in this situation rights to make derivative works of G', the owner of the rights in G' cannot argue that the end-user is actually infringing their derivative work right. Without underlying infringement by the end-user, any claim to contributory infringement must fail.

To be sure, using such a strategy to circumvent the GPL seems clumsy. In order to obtain a functioning program, the end-user would first have to obtain C(G'). Then, they would have to separately obtain C(S), and link that module to C(G') to create an

executable. The executable E, that is created by the end-user, is a derivative of G, as discussed above, but the critical distinction here is that X did not distribute that derivative work. X only distributed the building blocks required to build E. One of those building blocks, C(G'), is distributed under the terms of the GPL, as required. The other building block, C(S) may be distributed under another, more restrictive license, because if we believe the above analysis, C(S) is not necessarily a derivative work of G.

However, as times change we encounter new ways to build and distribute software. In particular, software systems are becoming increasingly modularized, supporting increasingly high levels of reuse and sharing. The distribution of software has also become increasingly fluid as networks become a dominant distribution medium. In many cases, the monolithic executable that lives in a single address space is being supplanted by networks of clients and servers, intercommunicating in order to provide a service for an end user. In this way, the architecture of programs begins to sprawl across machines and the networks that connect them, making the line drawing exercises demanded by a license such as the GPL increasingly difficult. Example 2.1 should therefore be viewed as more than an odd academic exercise, because the issues it raises will reappear in the real-world software scenarios we shall visit in the sections that follow.

### 3.4 STATIC OR DYNAMIC LINKING: DOES IT MATTER?

**Example 3.0**: This example is the same as example 2.1 except that it takes place in a dynamically linked universe. The main difference is that the executable E only exists as a physical combination of modules when the program is being executed. Until the program is launched, there is only a collection of disparate object modules residing on disk. When the program is launched, these object modules are stitched together to form something that we would call an executable. So it should be clear that dynamic linking in some ways makes the potential weakness in the GPL noted above easier to exploit, because the vendor of software only ever ships unconnected object modules, and the "combining" of those modules does not take place until the end-user activates the program.

The results in this example seem to be the same as above. As long as module C(S) is not considered to be a work derived from G, then it is not within the scope of the GPL. If X can distribute C(S) separately from C(G'), and the combination of C(S) and C(G') is only made by the end-user, X should be free to license C(S) under whatever terms she chooses. And the distribution scheme seems a fraction less “kludgey” than the one employed in example 2, because the step of combination (linking) occurs transparently to the user of a dynamically linked system.

Much has been made of the issue of how dynamic linking bears upon the applicability of the GPL. Some commentators contend that a first module dynamically linked to a second, GPL'ed module is not necessarily subject to the GPL. (Lawrence Rosen, for example). The drafters of the GPL, on the other hand, argue that the first module in

this instance would be subject to the GPL, regardless of whether the two modules were statically or dynamically linked. In the GPL FAQ they note, "If the modules are included in the same executable file [static linking], they are definitely combined in one program. If modules are designed to run linked together in a shared address space [dynamic linking], that almost surely means combining them into one program." They go on to say that if modules communicate with each other over networks, they are unlikely to be considered derivatives of each other. In a way, framing the question in this manner has been unfortunate, because it focuses the inquiry upon the mechanism of inter-module communication rather than on the more metaphysical - and legally significant - inquiry into whether one module is in fact a derivative of the other. And as we have seen above, courts answer this question not by reference to the technology underlying the work but by reference to qualities such as incorporation and substantial similarity, tempered by subject matter limitations, fair use defenses, and public policy rationales.

In short, the debate over static and dynamic linking simply misses the mark. As we shall see in our next example, using inter-module communication as the basis for a derivative work analysis will lead frequently to counter-intuitive and nonsensical results.

### 3.5 THE PROBLEM WITH PLUGINS

A plugin is a module of code that can be easily - from the end-user's perspective - installed and run within an existing application. A plugin generally expands or extends the functionality of the original program in some way. Many modern application programs contain some form of plugin architecture, so that third parties can write extensions for the underlying application. For example, the Mozilla Firefox browser has a plugin architecture. This allows, for instance, Adobe Systems to offer its Acrobat Reader as a plugin for Firefox. After installing the plugin, if a user loads a PDF document, Acrobat Reader will launch and appear within the browser window. Another example is the Flash Player, which allows users to view animations within their web browser. In this way, the designers of Firefox can enable (or rather, allow third parties to enable) the viewing of novel content types within the friendly confines of the browser, but without disturbing the core architecture and design of the browser itself.

**Example 4.0**: Program P, licensed under the GPL, contains a plugin architecture that allows users of P to dynamically download and run object modules that extend P's functionality. X writes a plugin by creating a source module S, compiling it, and advertising the resulting C(S) on his website. Does X have to license the plugin C(S) under the terms of the GPL?

This is yet another, but more naturally occurring example of the GPL's difficulty in dealing with distribution of related modules that are not assembled prior to distribution. In this way, the analysis here does not differ greatly from that of Examples 2 and 3, above. The primary difference is that X is not even distributing G', which recall was a slightly altered version of the original program G, modified so that S could be integrated into it. Here, the end-user has presumably obtained the analog of G' (P),

from a third party. The vendor is only writing and distributing the plugin. The combined work, if there is one, does not come into existence until the plugin C(S) is installed and running on the end-user's machine. So the GPL need only be applied to S if S, standing alone, can be viewed as a derivative of P.

The GPL FAQ takes an explicit position on this particular scenario. It states that the result "depends on how the program invokes its plugin. If the program uses fork and exec to invoke plugins, then the plugins are separate programs, so the license for the main program makes no requirements for them." On the other hand, "If the program dynamically links the plug-in ... they form a single program, which must be treated as an extension of both the main program and the plugin." The terms fork and exec refer to UNIX system calls that a running program uses to launch another program. Every modern operating system provides some equivalent set of system calls. The GPL drafters therefore draw the boundary around components that are linked together, operate in the same address space, and have some intimate level of interaction and communication.

To understand the implications of this method of boundary drawing, consider for a moment the Acrobat Reader plugin for Firefox, and assume that the browser is licensed under the GPL. The plugin offers its own set of user interface buttons and is capable of rendering file formats that are exotic to Firefox. Does this seem like a derivative work of Firefox? Whatever our intuition may tell us, the answer depends - according to the GPL's drafters at least - on the particulars of the Firefox plugin architecture. If the plugin architecture launches plugins and runs them in separate address spaces as separate, running executables, then the GPL does not consider Acrobat Reader a derived work of Firefox. On the other hand, if the plugin is dynamically linked to Firefox, then the GPL urges the opposite characterization. This seems to fly in the face of common sense, which tells us that Acrobat Reader is not a work derived from Firefox.

The following thought experiment demonstrates the absurdity of the GPL approach. Imagine taking the Firefox source code and modifying it in such a way that it is encapsulated in a plugin, called Fireplug. Common sense and the law tell us that Fireplug is a derivative work of Firefox, based simply on the fact that Fireplug incorporates all or most of the Firefox codebase. Now, when we plug Fireplug into Firefox, we essentially have a Firefox browser running within a Firefox browser. And depending on the Firefox plugin architecture, the GPL drives us to incompatible results. If plugins are dynamically linked, Fireplug is a derivative work. This is the right result, but the reasoning is exactly wrong. Fireplug is a derivative work because it incorporates and is substantially similar to Firefox, not because of a decision made by the designer of the Firefox plugin architecture. On the other hand, if plugins are launched as separate programs, then Fireplug suddenly is not a derivative work, even though this result is counter the common sense result above. Clearly, the mechanism of a given plugin architecture are analytically unhelpful and misleading in answering the derivative work question.

### 3.6 OBJECT ORIENTED SYSTEMS

Our next example concerns the GPL's applicability to object-oriented software systems. The central programming abstractions provided by object-oriented languages are those of*objects* and *classes*. Objects represent real, functioning program entities that contain data and can respond to certain messages. Classes are a means of defining and organizing a world of objects. A class defines the data and behaviors of a group of objects. For example, a programmer may wish to build a student database application for a university. In doing so, she may create a class called Student that defines the relevant characteristics of a student. These characteristics may include data such as name, student ID number, and GPA. They may also contain behaviors that students can perform. For instance, a student might know how to take exams, complain, and ask questions. Finally, classes may relate to each other by means of *inheritance*. In our example, our programmer might create a *subclass* of Student called GradStudent that specializes and/or extends the behaviors of Student in some manner. For instance, a GradStudent may prefer to complain about his or her advisor, whereas a generic Student may typically prefer to complain about grades. Inheritance allows the programmer to easily reuse much of the functionality of existing classes, by defining a subclass that overrides certain default behaviors of its *superclass*. (Some object-oriented languages, C++ for example, use the term *base class* for superclass and *derived class* for subclass. For obvious reasons, we shall avoid the use of those terms here.)

**Example 5:** Programmer X wishes to write a class D, that is a subclass of existing class B. Class B is subject to the terms of the GPL. If X distributes D, does it have to be licensed under the terms of the GPL?

The answer given in the GPL FAQ is short and to the point: "Subclassing is creating a derivative work." In our example, this makes D a work derived from B, and thereby makes D subject to the terms of the GPL upon distribution. This approach attempts to further broaden the reach of the GPL, but it again leads to counter-intuitive results.

Typical object oriented programming languages include a standard class hierarchy. This hierarchy provides a framework within which application developers can build their programs. The standard classes typically provide useful classes that represent user interface elements (e.g. windows, buttons, etc.), collection classes (for handling collections of data), and input-output abstractions (e.g. files and networking connections). In many object oriented languages, each class must be a subclass of exactly one superclass. And for this reason, the class hierarchies are rooted by a highly generic, standard class called Object. (The question of the superclass of Object is beyond the scope of this article.) The class Object describes only the most general properties and behaviors. For instance, in Java, the class Object only performs a handful of functions. In Java, every class is a subclass (directly or indirectly) of the Object class. Under the GPL approach, then, every program written in Java is a derived work of Object, because every program written in Java by definition consists of classes that inherit from the Object class.

Such a result certainly sounds extreme. The Object class in Java is extraordinarily generic and bears little relation to many of the classes that inherit from it. To be sure, all Java classes can perform the behaviors defined by Object, precisely because they are subclasses of Object, but this does not necessarily imply that those subclasses should be considered derived works under copyright law. It would be as if an author writes an incredibly generic, one-sentence long "story" along the lines of "There are some characters who do things and stuff happens to them" and then claims that just about every novel that follows is a work derivative of his story. From a naive perspective, almost every story (perhaps with the exception of *Waiting for Godot*) can be considered derivative of our short story because it borrows or incorporates the same essential plotline. Obviously, though, this is not the result that we obtain under the law. Principally, the reason is that the short story would not be considered copyrightable subject matter, either because it does not meet the minimum creative threshold, or because it is already in the public domain, or because the idea of the story has merged with its expression.

As usual, the answer in this example depends on questions of copyrightability, incorporation, substantial similarity, market demand, and public policy. As a general matter, in the case of inheritance in object oriented systems, a subclass trivially incorporates (and is substantially similar) to its superclass. The analysis should then turn, to a larger degree, on questions of copyrightability of the superclass, market demand, and public policy. For example, if the class B is highly generic (as in Java's Object class), it is unlikely that it would even be considered copyrightable subject matter under the 102(b) bar. On the other hand, if B is highly specialized—as are classes that we find on the fringes or leaves of a class hierarchy—there may be a stronger argument that they contain substantial copyrightable subject matter. This argues, then, for a finding that D is a derivative of B. Even here, though, a market demand analysis might argue against a finding of derivation. Class D will not supplant demand for class B, simply because it makes no sense to deploy D in the absence of B.

### 3.7 NETWORKED SYSTEMS

**Example 6**: Software vendor V who distributes proprietary software wishes to add the ability to manipulate digital sound files. Being short on time, V finds a GPL'ed sound library, dresses it up as a server, and gives away the server under the GPL. Meanwhile, their core, proprietary application is modified to communicate with the "sound server" in order to perform sound processing tasks. End-users who wish to exploit the sound functions have to install the proprietary code alongside, or at least on the same network as, the GPL'ed sound server. Both of these items are "distributed" by the vendor.

Is the core, proprietary application now bound by the terms of the GPL? No, certainly not, even according to the GPL FAQ. The FAQ notes that "... sockets ... are communication mechanisms normally used between two separate programs. So when they are used for communication, the modules normally are separate programs." In

many ways, this example does not seem controversial, but we bring it up to again emphasize the inadequacy of the reasoning by which the GPL drafters reaches their result. As with plugins and object-oriented systems, above, the GPL FAQ writers seem to reach this conclusion *because* of the way the two programs interact. Because the two works are two separate programs interacting over a network, they are somehow different from two modules interacting by means of function calls within an address space. As usual, this is not necessarily so. It is easy to construct an example where the work, viewed as a whole, is implemented by means of multiple servers intercommunicating over a network. And if an author comes along and adds an additional server that somehow extends the functionality of the existing community of servers, we might be inclined to say that the new server is a derivative of the original work, independent of the fact that it communicates with the original via a network.

## 4. CONCLUSION

In some ways, the apparent weaknesses in the GPL should come as no surprise, as the GPL was born of an era in which the central artifact of software development and distribution was the monolithic executable. In such a universe, software development proceeded principally by modifying the existing source text of programs, compiling source modules, linking the corresponding object files, and distributing the resulting executable. This model of software development and distribution has become increasingly fractured in an era characterized by highly dynamic, late binding, object- and network-based systems. The GPL, consequently, strains to cover these newly arising scenarios.

To effectuate the goals of the free software movement, the drafters of the GPL urge a generally expansive definition of derivative work. The great irony is, of course, that such an expansive definition would have second order consequences that are exactly counter to the goals of the proponents of Free Software. A broad definition of derivative would give code authors *less* freedom to create software that they can truly call their own and do with as they please. And if naive analytic approaches such as "subclassing equals derivation" reign, then proprietary vendors such as Microsoft could arguably stake claim to every program ever written in C#, because they authored the original class hierarchy. And since it seems unlikely that courts would employ different standards depending on the goals or ideological motivations of licensors, proponents of free software might want to be careful what they wish for: what's good for the GNU might not be good for the gander.

## 5. REFERENCES


- Aho, Alfred V. & Ullman, Jeffrey D. (1992), *The Foundations of Computer Science*, Computer Science Press.
- Abelson, Harold & Gerald Jay Sussman with Julie Sussman (1985), *Structure and Intrpetation of Computer Programs*, The MIT Press.
- Patterson, David A. & John L. Hennessy (1998), *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann.

- Hollaar, Lee A. (2002), *Legal Protection of Digital Information*. Available at http://digital-law-online.info/lpdi1.0/index.html.
- Rosen, Lawrence (2005), *Open Source Licensing: Software Freedom and Intellectual Property Law*, Prentice Hall.
- St. Laurent, Andrew M. (2004), *Understanding Open Source and Free Software Licensing*, O'Reilly.
- The U.S. Copyright Act (17 U.S.C. §100 et. seq.). Available at http://www.copyright.gov/title17/.
- GNU General Public License (GPL). Available at http://www.linux.org/info/gnu.html.
- GNU GPL FAQ. Available at http://www.gnu.org/licenses/gpl-faq.html.
- *Computer Associates International v. Altai*, 982 F.2d 693, 23 U.S.P.Q.2d 1241 (2nd Cir. 1992).
- *Midway Mfg. Co. v. Artic International, Inc* ., 704 F.2d 1009 (7th Cir. 1983).
- *Lewis Galoob Toys, Inc. v. Nintendo of America, Inc.* , 964 F.2d 965 (9th Cir. 1992).
- *Micro Star v. Formgen Inc.* , 154 F.3d 1107 (9th Cir. 1998).