Carving up the Commons: How Software Patents Are Impacting Our Digital Composition Environments

Annette Vee

University of Pittsburgh, Pittsburgh, PA

Abstract

Since the 1980s, there has been a general trend in U.S. courts to allow certain aspects of software programs to be patented. Digital composition scholars and teachers are indirectly affected by these decisions through the software environments in which we compose but we are also directly affected through our increasingly code-based methods of composition. Just as we can no longer limit our study of writing to text, we can no longer limit our considerations of intellectual property law to the copyright that governs text. Here I examine arguments forwarded by legal scholars and programmers against software patents; these analyses offer not only a convincing critique of the patent system, but also imply that our legal system should treat code more like writing than engineering. An exploration of the software patent debate, then, opens code up for further study in the field of computers and composition.

Keywords: Code; Composition; Intellectual property; Programming; Patent; Software

1. Introduction

At first glance, patent law has very little bearing on writing studies. Patents cover such innovations as Prozac, low-plastic-content water bottles, and ergonomically oriented chairs; although all of these technologies may affect the writer in process, they are not material to the writing itself. Writing is generally thought to be under the intellectual property aegis of copyright law: novels, poems, letters to friends, and grocery lists are all protected forms of expression under U.S. copyright law. However, a certain form of writing—the writing that comprises computer software, or source code—is protected by patent law. In addition to software patents’ direct impact on the composition process of computer software, software patents are narrowing the potential for digital composition environments and are circumscribing the writing we do online. Ontological arguments that equate code with writing, math, or engineering can be seen in the legal and software-development communities as they passionately debate what legal protection software should have. That software patents affect the digital environments in which we and our students compose makes these patents a critical area of study for composition teachers. That these patents force us to consider how the domains of writing, math, and code are increasingly intersecting makes them a fascinating touchstone for considering the role of computers in composition. Katherine Durack (2006) claimed that “researchers in cutting-edge subjects can no longer depend on pursuing inquiries in ignorance of the patent system” (p. 315), and I argue that researchers in writing also cannot continue this ignorance. Just as we can no longer limit our study of writing to text, we can no longer limit our considerations of intellectual property law to the copyright that governs text.

E-mail address: annettevee@gmail.com.

8755-4615/$ – see front matter © 2010 Elsevier Inc. All rights reserved.
doi:10.1016/j.compcom.2010.06.006
I argue that composition studies should direct attention to the developments in patent law for two main reasons: 1) the programs students use to compose digitally are implicated in these debates; and 2) debates about software patents also affect the composition contexts of code, which is a kind of writing. Code is both a means of enacting processes and a description of those processes. When applied to software, patent law foregrounds the functional aspects of code rather than the textual aspects. These textual aspects of code, including the style and creativity involved in expressing processes, can be more effectively approached from the analytical tools of writing studies and the intellectual property regime of copyright. A discussion of patent law pertaining to software—as well as the creative nature of code composition—then serves as an opportunity for writing scholars to become more involved in the current policy debates about software patents, especially as we see our composition environments and processes handicapped by overly broad patent restrictions. To draw greater attention to patent law in writing studies, I first review specific situations where software patents have affected digital writing environments common to composition classes. Next, I briefly outline U.S. patent law, including the case law that gradually made software patentable and the debates about this process that followed. Finally, I visit three code writers—one each from corporate, open source, and freelance contexts—who have had their writing processes impacted by patent law. Each time this happens, the most popular—and often, proprietary—software for digital composition increases its mindshare, and we lose a modicum of control over the writing environments in which we and our students compose. Each time this happens, the most popular—and often, proprietary—software for digital composition increases its mindshare, and we lose a modicum of control over the writing environments in which we and our students compose.

2. Code, composition studies, and intellectual property

As an instrument that allows composers to embed audio, video, graphics, and photographs into writing, the computer has revolutionized what we think of as linear composition (Bolter, 1991). Writing scholars now routinely consider images, audio, blogs, and games as part of our studied domain, demonstrating the protean powers of the modern computer to translate writing into anything, and anything into writing. Ethan Katsh wrote that “electronic media is a technology that does not support boundaries, either physical or conceptual” (as cited in Gurak, 1997, p. 332). The way the computer allows us more easily remix pre-existing material—combined with a legal landscape that increasingly favors copyright owners over those poised to create new content out of the old—has forced us to more carefully consider the political, ethical, and practical dimensions of copyright on our professional lives. From the work of Gurak (1997) to that of the digirhet collective (2008) to that of Steven Westbrook (2009) and others, composition scholars have helped us negotiate the uncertain impact of the law in our classrooms and professional lives.

As many writing scholars have noted, we have seen an erosion of the public domain in copyright law. Yet little attention has been paid to the impact patent law can have on the software in which we now compose. Over the last 15 to 20 years, a repeated claim in writing studies scholarship has been that control over our digital writing environments means control over the code that governs them. As Cynthia L. Selfe and Richard J. Selfe (1994) claimed, the assumptions embedded in coding practices carry through to our interfaces. For this reason, Paul Leblanc (1993) argued that composition teachers should be creating their own writing software. All of the articles in the 1999 Computers and Composition special issue on code echo, in some way, Leblanc’s claim that writing teachers should be able to code (or at least help to design) the software environments in which they write. Writing and web-editing software may be robust enough now to deter us from designing our own from scratch; however, programs we can customize such as OpenOffice Writer can make us code-authors; writing applications through the Firefox extension Greasemonkey, for instance, can help us manipulate our online writing environments (Ballentine, 2009). However, the existence of software patents means that even access to the code provided by open-source software fails to guarantee our right to customize our writing environments. Patent law can stifle competition that would be beneficial for our composition ecosystem, or these laws can simply limit features in the software available to us. These limitations may affect our software choices or the choices of students as we may tend to favor proprietary software from companies that wield the most patents and can therefore offer simpler or more elegant, feature-rich environments in which we can compose. Each time this happens, the most popular—and often, proprietary—software for digital composition increases its mindshare, and we lose a modicum of control over the writing environments in which we and our students compose.

Like the shrinking of the public domain we have witnessed due to stronger copyright laws, we can see a carving up of the commons in patent law: programming techniques and features in programs, once patented, are unavailable for common use for 20 years. Although the authority for Congress to enact both patent and copyright law stems from the same clause in the U.S. Constitution, the two areas of intellectual property law cover very different types of work through very different processes: a copyright is automatically granted by law for any creative work, but a
Software patents, like our writing, are not restricted to the desktop. As Durack (2001) noted, patent law controversies exist online as well: “The Internet is both the object of debate about patentability and the slate upon which much of the argument is inscribed” (p. 504). For instance, the popular online microblog site Twitter was sued in August 2009 for patent violations. TechRadium, the company suing Twitter, had developed a mass communication coordination system for public safety applications and then claimed that Twitter’s service was too similar to their own, for which they had applied for patent protection (Schonfeld, 2009). Although the outcome of the case is yet to be determined, Twitter, which has so far failed to make a profit on their service, may be forced to pay royalties or change their software’s features in response to TechRadium’s claims.

Software patents can affect the features and software programs in which we compose, but they can also affect compositions themselves when those compositions are in code. In addition to incorporating audio and images, digital work—such as web sites, games, or electronic literature—can include code as a central part of its composition. For this reason, scholars such as David Rieder (2008) have argued that code is a form of writing and can be governed by aesthetics, just as much as writing can be. Like Rieder, many programmers have claimed that code and poetry have similar formal constraints and allow for similar kinds of creative expression. Perl poetry is an often-cited example in critical code studies, a branch of analysis that focuses on “code as a text, as a sign system with its own rhetoric, as verbal communication that possesses significance in excess of its functional utility” (Marino, 2006). One suggestion to solve the software patent problem is to limit patent terms to just several years; Ben Klemens (2006) pointed out that the...
existence of Perl poetry would stymie this approach, however. Where would we draw the line between functionality and expression?

A more public and accessible demonstration of code as a creative text was seen when the hashtag “#songsincode” became a “trending topic” on Twitter. On August 21, 2009, programmers and wannabes dove into collective pop song memory to produce Perl, PHP, C, or other code language versions of songs like Britney Spears’ “Hit Me, Baby, One More Time,” much to the delight and distraction of code-literate citizens of the Twittersphere (Smith, 2009). Each little code puzzle conveyed a reference to a song and so relied on both pop culture and code knowledge to crack it. Some #songsincode similar to those that robbed me of productivity that day include:

1. if (!woman) {cry = false;}
2. final--;
3. $self- > set(‘hot’,’sticky sweet’) while/head/ /feet/;
4. if(somethingStrange = =true && location == neighborhood){ghostbusters(); printf(“I ain’t afraid of no ghost”);}
5. if(youAre == 0 × 000000 || youAre ==0xffffffff) doesMatter = false;
6. .clowns{float:left;} .jokers{float:right}; #me

#songsincode was highly motivating to those wanting to learn code because it forced people to figure out the syntax to crack the riddle. Although the programming functions described in these code puzzles are probably too basic to have been patented, the event suggests that code can be more than functional; #songsincode are human-readable and playful.

The rhetorical parallels between composition in writing and in code were noted by Ron Fortune and Jim Kalmbach (1999) in their introduction to a special issue of Computers and Composition on code. Bypassing the creative angle on code, they demonstrated that programming can be elegant, and therefore rhetorical. Code can solve a problem in an efficient and concise way, and that effect can be independent of the program and the interface with which a user might interact. Robert Cummings (2006) carried the connection between writing and programming further:

Though their linkage is relatively unexamined and under-appreciated, both pursuits are inextricably joined by the fact that they center around the act of writers writing. Both types of writers—writers of code and writers of text—write for vastly different audiences and with what would seem to be vastly different products, but [...] the underlying model of composition holds true for both communities. (p. 431–432)

Patent law serves the rhetorical aspects of code poorly because it protects its utility without regard to its style or its composition process. A patent protecting a brute force and inelegant solution to a coding conundrum can block the development of better, more effective solutions and expressions of the problem. How patent law can obviate elegant code is reviewed in the next section.

3. A short review of patent law

The U.S. Constitution provides authority for the government to establish laws concerning both patents and copyrights in Article I, Section 8: “The Congress shall have power [...] To promote the progress of science and useful arts, by securing for limited times to authors and inventors the exclusive right to their respective writings and discoveries.” After this initial authority granted to Congress, intellectual property laws governing trademark, patent, and copyright have diverged. A full accounting of this divergence is impossible here; instead, I will review some of the parallels and differences between patent and copyright law, with which composition scholars are likely to be more familiar. Knowledge of the differences between the paradigms of these intellectual property regimes will help to illuminate why many programmers are in support of copyright protection but not patent protection for code.

---

1. Bob Marley’s “No Woman, No Cry” in generic C/C++ syntax (via @librarythingtim); 2. Europe’s “The Final Countdown” using a counting convention common to many programming languages (via @uberTof); 3. Def Leppard’s “Pour Some Sugar on Me” in PHP (via @petdance); 4. Ray Parker, Jr.’s “Ghostbusters” in C (via @numptygeek); 5. Michael Jackson’s “Black or White” in generic C/C++ syntax, using hexadecimal representations for the colors black and white (via garysbasement.com); 6. Stealers Wheel’s “Stuck in the Middle with You” using CSS positioning syntax (via garysbasement.com).
Patent and copyright law are both specified in the U.S. Code, copyright in Title 17 and patent in Title 35. Case law also helps judges interpret the legal code for patents and copyrights. Copyright can be held on “original works of authorship fixed in any tangible medium of expression,” such as literary works, pantomimes, and motion pictures (17 U.S.C. §102). A U.S. patent can only be acquired through an application to the USPTO. Patents can be obtained for inventions in three categories: utility, design, and (biological) plant, with most patents falling into the utility category. According to U.S. law, patentable subject matter is “any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof” (35 U.S.C. §101). In addition to being useful, the invention must also be “novel” (§102) and “nonobvious” (§103). A patent grants the right to exclude others from using the specified invention for 20 years (the first U.S. patent law in 1790 granted rights for 14 years). These rights are very powerful, and they are therefore balanced by a shorter time limit than copyright as well as by a disclosure requirement.2 Inventors must disclose sufficient detail in their patent application to “enable any person skilled in the art to which it pertains. . .to make and use the same” (35 U.S.C. §112).3 This disclosure means that when a U.S. patent expires, the public not only has the right to use the invention but also ostensibly has the directions for doing so. In this way, the private investment of the inventor is recouped while the information the patent-holder provides ensures that the public interest is served once that limited monopoly expires.

Copyright law pays attention to the composition process behind a work of authorship whereas patent law does not. To prove copyright infringement, a rights holder must demonstrate that at some point during composition the accused actually copied the copyrighted work. In contrast, patent law bars independent discovery; a person can infringe on a patent without knowing the patent existed. Moreover, the doctrine of equivalents (DoE) bars works “equivalent” to the patented invention. The DoE prevents people from trivially circumventing patents, but it can also curtail alternative inventions. For instance, the DoE can prevent software teams from using the “clean room” development model to write code. The “clean room” model describes a team’s reverse-engineering of a software program or feature in an environment completely isolated from copyrighted code. It allows for multiple programs to be created that can fill similar needs without copyright violation; however, this method can still violate patents through the DoE.

Patent applicants naturally try to claim as wide a scope as possible for their invention, but the USPTO can reject claims that are too broad. This balance can work well when patent examiners have sufficient time and information to review applications; however, huge increases in patent applications over the last 30 years, as well as the limited information provided with software patent applications, have resulted in many overly broad software patents. The conditions of novelty and nonobviousness have been poorly applied to software patents in particular, in part because the patent examiner’s manual does not suggest a review of prior art located in software. Additionally, applications require only a “written description”; consequently, many software patent applications use vague flow charts rather than code to claim their invention (Klemens, 2006). Some of these overbroad patents, like Amazon’s notorious patent on “one-click ordering,” are later challenged and narrowed in court. However, many of them are never challenged because patent litigation is expensive. As in other areas of the law where “gray areas” must be adjudicated, results in software patent disputes often favor entities with greater resources for savvier lawyers. Those with fewer resources are often forced to settle out of court to conserve their meager means, which may also skew case law to favor more powerful entities. Vandana Shiva’s (2001) work describes some of the cultural harm that can occur in patent cases related to disparate resources. Specifically, she outlines the stifling of agricultural innovation in developing countries due to the power of plant patents held by more powerful developed countries. Just as multinational agricultural companies such as Monsanto can profit from the ownership of plant patents, large software-development corporations can use patents to exert disproportional control over what might otherwise be common innovations. As will be seen below, small, under-resourced developers such as those in open-source software creation and software start-ups bear the brunt of this impact.

Like copyright law, patent law is determined in part by case law or decisions from particular patent cases brought before the court. Major shifts in the policies for patent applications and the structure of patent courts have also had

---

2 For more information on the history and application process of U.S. patents, visit the USPTO’s web site: <http://www.uspto.gov/web/offices/pac/doc/general/>; see, also, Durack, 2006.

3 This hypothetical practitioner who could reproduce the invention is often called the PHOSITA, or “person having ordinary skill in the art” to which the patent pertains.
a tremendous impact on what the USPTO grants patents for and whether those patents stand up in court (Jaffe & Lerner, 2004). Although the legal ground for software patents is constantly shifting, a review of the history helps us understand how methods expressed in a particular form of writing—software—became patentable subject matter under U.S. law. We can see, for example, that the trend in patentability follows roughly the same arc as copyright, favoring individual creators (or “inventors”) over the public domain. In the case of patents, many of these inventors are actually corporations to whom individuals assign their patent rights as a matter of employment. In line with a general trend in the courts to protect the intellectual property of corporations, decisions made by the designated court for patent appeals, the United States Court of Appeals for the Federal Circuit (CAFC), and the Supreme Court throughout the 1980s and 1990s had the effect of sanctioning patents on software, although no U.S. law has been passed that expressly provides patent rights to software.

Controversies surrounding software patents began as early as the 1960s, when technologies were developing so quickly that President Johnson put together a special commission to review the USPTO’s policies and methods. The Commission, made up of academics, scientists, and industry representatives, recommended in its 1966 report that legislation be introduced to expressly exempt computer programs as patentable subject matter (Samuelson, 1990). No such legislation was passed; however, the USPTO examiner’s guide indicated that computer programs were not patentable under either the “process” or “machine” category described in 35 U.S.C. §101 (Samuelson, 1990). Two Supreme Court rulings in the 1970s affirmed the idea that computer programs were unpatentable; decisions in Gottschalk v. Benson (1972) and Parker v. Flook (1978) both indicated that the algorithms under review were not far enough removed from pure math to make them patentable.

Decisions in the 1980s and 1990s reversed the trend against the patentability of algorithms or “methods” without instantiation in physical devices. Gottschalk v. Benson and Parker v. Flook both indicated that only physical machines should be patentable, but in Diamond v. Diehr (1981), the Supreme Court ruled that the presence of any physical component could make a method patentable. After Diamond v. Diehr (a 5 to 4 Supreme Court decision), software patents began to use the phrase “a general-purpose computer on which is programmed a method to calculate” rather than “a method to calculate” (Klemens, 2006), essentially changing software patents from patents on process to patents on machines. The CAFC ruling In Re Alappat (1994) indicated that simply using the terminology “of machine” renders the invention a physical device. The CAFC decision that finally opened the floodgates on software patents was State Street Bank and Trust v. Signature Financial Group (1998), which unambiguously confirmed that an algorithm run by a computer—that is, software—was patentable, as long as it was useful.

Two more recent developments complicate this history of patent cases pertaining to software. First, the Digital Millennium Copyright Act (DMCA), as John Logie (2005) pointed out, “blur[s] the distinction between copyright and patent law” (p. 233) through the way it prevents circumvention of security measures in code. Klemens (2006) demonstrated that the DMCA also offers a new kind of intellectual property protection. This protection prevents such activities as the sharing of large hexadecimal numbers that can be used to allow you to play a DVD on a computer running Linux (i.e., the deCSS code), something to which the MPAA and Microsoft are opposed (Klemens, 2006; writing studies, legal, and information scholars have already provided excellent analysis of the DMCA, so I will not pursue it further here).

The second major complication in software patent law is that the Supreme Court appears to be sanctioning business method patents while staying mute on the subject of software patents. In the majority opinion in Bilski et al. vs. Kappos (2010), Justice Kennedy cited U.S.C. §273(b)(1) to say that business methods are not substantially different from other patentable methods. Since the patentability of business methods first opened the door to the patentability of software, this apparent contradiction creates uncertainty for software patents.

Activists against software patents had laid hope on the potential of the Supreme Court’s decision in Bilski to strike down the patentability of software, but the decision was a narrow one that left the more general questions of patents and newer technologies to Congress. But as Pamela Samuelson (2007) indicated, the U.S. software industry has invested heavily in the patent system over the last 25 years, and that investment serves as inertia against patent reform in Congress. Moreover, some powerful people believe that patents have been beneficial to the software industry. Consequently, the Supreme Court and Congress are unlikely to do away with software patents completely.

---

4 See Durack (2001) for an interesting discussion of patents as “a kind of moral and social barometer of technological change” (p. 495).
Europe honors U.S. patent law through trade treaties and the E.U. also, in practice, has allowed software patents (Nard, Barnes, & Madison, 2006; Guibault & van Daalen, 2006). This lack of clarity in the U.S. then contributes to an uncertain climate for software development worldwide, especially for small-time programmers without recourse to patent attorneys. Until the U.S. courts or Congress take a more definitive stand, the global legal status of software patents is unclear.

4. Legal protection for software

Throughout these decisions, we can see the courts struggling to define what software is. Is it a process or a machine? A form of math or engineering or writing? Does the application of code—a kind of text that specifies actions—make a computer into a “specialized machine?” These difficult and metaphysical questions have direct bearing on whether software can be considered patentable subject matter, and so they are hotly debated among legal experts, practicing programmers, open-source software advocates, and large companies such as Microsoft and Novell, all of whom have a stake in the intellectual property protection of software. There is not space to review the 40 years of controversy among these various factions, so I concentrate here on the arguments most pertinent to writing studies: that code as writing or math is protected by the First Amendment and that the compositional contexts for code often resemble those for writing much more closely than those for manufacturing or engineering.

First, it is crucial to note that most programmers seem to oppose software patents (Boyle, 1996; Lessig, 2001). Although it is impossible to comprehensively survey all programmers, Lawrence Lessig (2001) asserted that Richard Stallman and the software developers at Adobe and Oracle publicly opposed them in the 1990s, and current debates about software patents online are overrun with programmers against them. Even Bill Gates has expressed concerns about patents, although, businessman that he is, his concern translates to more patenting of techniques at Microsoft in order to facilitate cross-licensing with companies with large patent portfolios like Hewlett Packard. Of course, not all programmers are opposed to software patents; University of Utah professor Lee Hollaar, who recently wrote an amicus brief in favor of software patents for the Supreme Court’s review of Bilski, is one particularly notable exception (Jones, 2009). Additionally, some argue that if the system were built to avoid the plague of obvious and trivial patents, it could be redeemed (Jaffe & Lerner, 2004). However, many programmers, including patent-grantee Paul Graham (2006), are doubtful that the system could be fixed due to the complexities of software. The fact that many working programmers do not want the protection that patents give them should be a critical sign that something is wrong with the system.

In contrast, copyright protection is overwhelmingly supported by programmers in all areas of software development. Copyright protects software from being copied wholesale and given away for free, and it prevents cut-and-paste plagiarism of source code. As Richard Stallman (2009), the original author of the GPL “free software” license, has often repeated, the “copyleft” scheme on which free and open software relies is dependent on copyright law. Unlike patent law, copyright law protects against copying but not against reverse engineering or independent discovery. John Carmack (1997), founder of id Software, clarifies: “It’s important [. . .] to understand the difference between a software patent and the copyright because copyrights protect you in all the important ways that you really need to be protected.” Copyright protects developers from piracy and from people stealing code whereas patents prevent new solutions to problems in writing. In essence, programmers claim that intellectual property law should treat what they produce more like writing (protected by copyright) than engineering (protected by patents).

One argument against software patents that relies on notions of code as writing rests on the First Amendment. Logie (2005) described Tyanna K. Herrington’s argument in Controlling Voices: the debates about intellectual property and what we can do with it or say about it become, at a certain point, about the First Amendment. Is code a protected form of speech? Not always: the DMCA disallowed distribution of the deCSS code (which allowed users to unencrypt certain

---

5 See, for instance, the incredulous response to IBM’s amicus brief for the Supreme Court’s hearing of Bilski claiming that software patents help free software (“IBM’s Supreme Court Brief,” 2009).

6 In a May 16, 1991 confidential memo to senior executives at Microsoft, CEO Gates wrote: “If people had understood how patents would be granted when most of today’s ideas were invented and had taken out patents, the industry would be at a complete standstill today. I feel certain that some large company will patent some obvious thing related to interface, object orientation, algorithm, application extension, or other crucial technique.” He went on to express concern that Microsoft did not, at that time, have patents, and recommended that they should acquire some as soon as possible (Warshofsky, 1994, p. 170).
DVDs), despite arguments that it was a protected form of expression under the First Amendment. But sometimes code is protected speech. Daniel Bernstein, a graduate student at UC-Berkeley who worked on encryption methods, could not publish nor present his work under U.S. laws that limited non-governmental use of high-level encryption, so he sued the State Department for clarification of the law. In Daniel Bernstein v. U.S. Department of State, et al. (1997), the court ruled that the State Department’s regulations were in violation of the First Amendment. In other words, math was a form of free speech that could not be restricted by the government (Klemens, 2006).8

In 1991, the USPTO established an Advisory Commission on Patent Law Reform that was to consider software patents. When they invited public comments, Phillip Salin (1991), President of the American Information Exchange Corporation, submitted a brief claiming software as a form of protected speech:

That software patents are a severe violation of the rights of speech of programmers has not yet been widely recognized; this is perhaps in part because most lawyers, judges and politicians are still insufficiently knowledgeable regarding computers to realize that writing a computer program is in fact a form of writing, not significantly more arcane than writing music, mathematics, scientific papers, or for that matter, laws. All of these forms of speech, including writing programs, deserve full protection under the First Amendment.

In his statement, Salin concentrated on a process theory of software writing, arguing that it is a form of expression requiring invention and revision. He also claimed that software is communication between a programmer and an audience through the intermediary of the computer and that restricting that communication is a form of “censorship of the work of over 1,000,000 employed writers of computer software, along with the several million additional amateur writers of computer software.”

Another argument against software patents suggests that contexts for the composition of code are more similar to those for writing than those for activities traditionally protected by patents, such as chemistry or engineering. Klemens (2006) pointed out that the software industry is highly decentralized—businesses of all types often have teams of programmers that maintain the software and hardware upon which the company’s work depends. Beyond business, thousands of people write their own code to put up web pages, customize programs, or do design work in such programs as Adobe’s PHOTOSHOP. Garfinkel et al. (1991) argued that programmers produce too much and too varied code for it to be governed in a practical way by patent law; what they produce is on a different scale than teams of chemists working at universities and publishing primarily in academic journals. Prior art for software could be located in manuals, academic papers, the Internet, closed-code proprietary software, in-house business programs, and even people’s home computers—in other words, too many places to be feasibly searched.

Lessig (2001) performed a simple marginal cost analysis: before the Web, the transaction costs of licensing through copyright and patents was just a cost of doing business to large publishers or corporations—they were the only ones playing that game anyway. But the Web makes more opportunities for individuals and smaller groups to make and publish creative works—software included—and the marginal costs for individuals to check for potential patent infringement are substantially more than for larger corporations. Even the Supreme Court’s Bilski (2010) decision notes how a radically altered context for software composition forces a reconsideration of patent law, although they decline to do so: “This [Information] Age puts the possibility of innovation in the hands of more people and raises new

---

7 To make his point clear, Klemens (2006) boldly prints the deCSS code with the caption “The deCSS Code: Printing This Figure Is a Felony,” juxtaposing it with instructions for a fertilizer bomb such as the one used in the Oklahoma City bombing and a caption reading “How to Make a Fertilizer Bomb: This Figure Is Protected by the First Amendment” (p. 126).

8 The argument that software is math and therefore unpatentable is also popular among programmers and legal scholars. The website Groklaw, which covers court cases and laws related to computer technology, recently published “An Explanation of Computation Theory for Lawyers” by PolR, a self-described “computer professional with over 25 years of experience [with] no legal expertise beyond what I acquired reading Groklaw” (PolR, 2009). In this 37-page document, PolR describes software in terms of computation theory, “an area of mathematics that overlaps with philosophy,” and asserts that if lawyers knew more about software, they would not see it as protectable by patent law. Carmack (2000) also expresses his frustration with the technological ignorance implied in trivial software patents:

To laymen, all of programming is alchemy, and trying to convince them that any given software patent is “obvious” or “clearly follows from the problem” is really tough. [...] The only scenario that I can see would be to have enough truly, blatantly stupid patents prosecuted that someone could make a stand in congress [sic] and show the public in an understandable way just how wrong it is.

Programmers <fn0045> frequently assert that patents are awarded to procedures that have long been accepted practice, and therefore appear to be novel only to outsiders or those less familiar with code.
difficulties for the patent law. With ever more people trying to innovate and thus seeking patent protections for their inventions, the patent law faces a great challenge in striking the balance between protecting inventors and not granting monopolies over procedures that others would discover by independent, creative application of general principles. Nothing in this opinion should be read to take a position on where that balance ought to be struck” (p. 10).

Like writing text and writing math, writing code has a relatively low barrier to entry, low capital costs, and it is potentially able to be a more generalized skill. Computers are still not universally accessible, but they are widely available, and increasingly so all the time. Programs such as Nicholas Negroponte’s One Laptop Per Child (OLPC) posit computer access as a right, and the OLPC includes a code writing environment. Are Brazilian children receiving an OLPC expected to be wary of patent infringement when they write code? Salin (1991) relied on an idea of programming as a mass literacy to make this point:

Writing programs today is no more esoteric than writing prose once was. [...] Until a few hundred years ago, literacy was a rarity. Acquiring the ability to write prose took training. It still does, but nowadays we teach writing to everyone in schools [...] There are now millions of individuals in the U.S.A. who know how to write a computer program. It is an absurdity to expect those millions of individuals to perform patent searches or any other kind of search prior to the act of writing a program to solve a specific problem.

As the cost of computers drops and the literacy of computer programming becomes more widespread, we cannot expect the myriad writers of software to have a catalog of software patents in their head.

That computer programming literacy may become widespread, and that it is advantageous for educators to encourage such a trend, is an argument made by scholars as diverse as programming pioneer Alan Perlis (1964), educational theorist and mathematician Seymour Papert (1980), and former head of Carnegie Mellon’s Computer Science Department, Jeannette Wing (2006). The histories of writing and math both yield a number of resource-poor writers and mathematicians such as Frederick Douglass, the writer, or Srinivasa Ramanujan, the self-taught mathematician, who were able to practice their skill to the great benefit of society. Patent law now throws obstacles in the way of the future Douglasses and Ramanujans of programming because it posits the context of software writing to be like engineering or chemistry, with high-investment costs and centralized publishing venues. However, as code literacy becomes more widespread and decentralized, the compositional context for computer code becomes more akin to writing than engineering.

5. Three programmers’ perspectives

IBM is unlikely to threaten Brazilian child programmers with their patent portfolio, yet the existence of software patents and potential danger of infringement do shape the composition environment of many programmers. In this section, three programmers working in very different contexts are profiled to see how software patent protection affects their composition processes. Each of these programmers—John Carmack (a corporate game programmer), Jason Laughlin (a freelance/independent game programmer), and Bob Jacobsen (a programming lead on an open-source project)—has had to adjust his code composing practices in attempts to avoid patent infringement. These adjustments have variously led to inferior code, a resistance to disclosing information, or a threatened community of hobbyist programmers.

John Carmack, co-founder of id Software and widely viewed to be one of the most innovative programmers working in computer games today, has consistently and vociferously protested software patents. He told Alex St. John in a 1997 interview that when lawyers were encouraging him to patent his programming techniques,

I delivered an ultimatum that said if id Software patents anything, they’re going to be doing it without me because I will leave. And the fallout from that was not pretty. Everybody was pissed off at me [...]. But that was something I felt strongly enough about that I, quite literally, would have left the company.

Carmack has been uniquely productive at pushing graphics programming to the limits—much of the success of id’s games such as Doom, Quake, and their sequels can be attributed to his cutting-edge techniques—he refuses to go the “business world” route and protect those techniques with patents. In line with his refusal to patent his techniques, Carmack has routinely released code from id’s games as open source to encourage people to learn from it and modify it for their own purposes.
In his moral objections to software patents, we can see the ideological differences Carmack sees between himself as a creator and the business interests with which his company must contend:

It’s something that’s really depressing because it’s so horribly wrong for someone who’s a creative engineering type to think about patenting a way of thinking. Even if you had something really, really clever, the idea that you’re not going to allow someone else to follow that line of thought... All of science and technology is built standing on the shoulders of the people that come before you. Did Newton patent calculus and screw the hell out of the other guy working on it? It’s just so wrong, but it’s what the business world does to things, and certainly the world is controlled by business interests. And technical idealists are a minority, but it doesn’t mean that I have to drag myself to do things that I don’t consider right. (Carmack, 1997)

Like Carmack, independent game designer and programmer Jason Laughlin (personal communication, September 23, 2009) finds software patents morally reprehensible. Laughlin has worked on a number of popular commercial games and now works on web-based Flash games; he considers his current programming habits and lifestyle similar to an author of fiction in that he writes code for independent projects and then later seeks publishers to support his work. When asked in a personal interview whether he had thought about patenting any of his programming techniques, he replied:

For all I know there could very well be things that are patentable in what I’ve come up with. But the thought that I would put that out there, patent it, and then act as a troll at a bridge for anyone who wants to make good things, which I would otherwise enjoy using, there’s just something... I don’t know, there’s something contemptible, but even worse than that. I think that you have to be able to lie to yourself in a really profound way to be able to do that. [...] It might be legal, but when so much comes before where you are that is bigger and more important that you get for free because that’s just how the diffusion of information works—there’s just something pathetic about saying “everything that came before me I deserve to have for free and now I deserve free money for the tiny little stupid thing I added on top of that.” I don’t think the system can work that way.

Laughlin’s use of the word troll is frequently echoed in debates about software patents on technology-centered online forums such as Slashdot and Reddit: Programming, and its usage reflects the revulsion many programmers seem to feel about business interests that seek to profit from—rather than add to—innovations in code.

From Carmack’s and Laughlin’s “technological idealist” perspective, patents obstruct creativity; it is more interesting to make fun games and solve problems than to profit from their solutions. Laughlin described a desire to be free to design at a macro level:

I’m trying to solve problems at the game scope rather than necessarily down in the weeds of the “particular technique” scope, so from that perspective, patents are a thing that get in the way. They mean that instead of trying to solve the game problem, you’re busy trying to solve the particular technique problem.

In other words, the business and engineering assumptions behind patents threaten to interrupt Laughlin’s game design creative processes. He noted, as did Carmack elsewhere, that people can do amazing things with the technology of programming if artificial restrictions are not imposed: “I don’t know, maybe some of this is very programmery/game-designery of me, but I have nearly infinite faith that [...] if you just let smart people with good intentions solve hard problems and get out of the way, I have almost utopian beliefs about what can be achieved.”

These utopian beliefs that drive “technical idealists” not only make patents morally offensive to them, but they can also lead these programmers to quixotically attempt to handle patent concerns on their own. Bob Jacobsen (personal communication, September 11, 2004), a UC-Berkeley physics professor who leads the open source JMRI (Java Model Railroad Interface) project, explained in a personal interview that this characteristic idealism of hobby programmers can make them more susceptible to problems with software patents:

Deep in their heart, software writers think they know how to do anything. Software is so malleable that anybody who gets good at this really honestly believes that they can do anything. And that therefore writing a patent shouldn’t be any harder than writing a program. This, of course, is immensely untrue, but that doesn’t keep them from thinking it. So, the solution that a lot of open source people have right now is to write their own patents—which are awful. They are indefensibly bad. [...] Software is written by writing something, finding the bugs, writing something, fixing it, finding the bugs, fixing it, finding the bugs, fixing it, and two years later, you’ve got it. That does not work, at any level, in the legal field.
In a corporate or university setting, Jacobsen (who is named in several non-software patents for his research) notes that patent applications are handled by legal experts. But in open source—a space where the online context for code composition is very similar to the online context for textual composition in that they are both highly distributed and often uncompensated—patents are handled by hobbyists, if they are handled at all. In other words, programming can be a widely distributed skill that benefits people at many different skill levels, as it does in open-source software, whereas hobbyist-level legal skills can be dangerous.

Software patents can be offensive to programmers’ creative and moral senses, but they can also circumscribe the techniques programmers can use, particularly because patents bar independent invention. In 2004, id Software was poised to release the highly anticipated game *Doom 3* when Creative Labs, a company that makes sound card technology and works with id in the development process of their games, asserted that they held a patent on a shading technique that Carmack had independently developed for *Doom 3.*^9^ Ironically, the programming technique had been dubbed “Carmack’s Reverse” because he was the one to popularize the particular, reversed way to calculate shadows on objects in the game. Although prior art existed for Creative Labs’ patent and Carmack had been unaware of it when he developed his own technique, the game’s release was delayed by Creative Labs’ threat (Gibson, 2004). As Carmack (2000) had previously asserted: “Patents are supposed to help promote invention and allow benefits to accrue to inventors. By most definitions, I would be considered an ‘inventor’ of sorts, and patents sure as hell aren’t helping me out.” Carmack had already begun rewriting the code to use a different shading technique (although at the cost of some speed in the game) when id and Creative settled the dispute. In Carmack’s case, a patent seems to have curtailed innovation rather than encouraged it.

Patent law is also designed to encourage disclosure of new inventions, but for Laughlin, patent law discourages this disclosure. Laughlin noted that in game design, he can use a lot of techniques that are not readily apparent to players, but the techniques’ opacity can contribute to a fun experience in the game. Revealing those techniques by talking about them or opening his source code can make him more vulnerable to accusations of patent infringement:

> the fact that I don’t know all the millions of patents that have been written—and can’t know—means that I’m always much safer not helping anybody and not communicating than if I opened up my mouth or if I opened up the source [code]. Both of those things make me a much bigger, much much bigger target.

So instead of sharing his hidden game innovations, Laughlin keeps his mouth shut and his code closed. Clearly, the goal of the patent system to circulate knowledge more freely is not working in his case.

Open source projects are uniquely vulnerable to patent infringement claims because programmers do what Laughlin deliberately avoids: they talk about their techniques in publicly available forums, and they make their code available to everyone. Additionally, Lessig (2001) identified the complicated mechanics of licensing a patent to an open source project: “Who knows who they are? Who knows how many users need to be sanctioned? […]” Thus patents tilt the process to harm open code developers” (p. 213). Open source projects rarely have patent portfolios of their own to counter accusations of infringement or to assert that they performed specific techniques first (Guibault & van Daalen, 2006). The organization of open-source projects is generally loose, and like Laughlin the independent programmer, open source organizations or individual programmers lack the resources of lawyers and capital that are required to file for patents. Proof of prior art may be in the code, but if it is not in case law or in writing, it is less accessible for the purposes of legal defense.

If software patents were never prosecuted, or even if their prosecution was limited to larger corporations who could work out royalties or cross-licensing deals, then they would be of no consequence to individual programmers. But for Jacobsen, who is currently embroiled in a patent lawsuit over techniques used in the open-source model railroad project he leads, the impact of software patents on his code-writing hobbies are profound. Matthew Katzer, a former member of the JMRI community, obtained a patent on some of the techniques used in the open-source model railroad project despite the fact that prior art existed for those techniques within the version history of the JMRI software itself. Katzer subsequently came after the open-source project and Jacobsen in particular with a demand for damages from patent infringement. The legal history of the case is complicated by a countersuit from Jacobsen, cybersquatting accusations, and trademark disputes, which I will not review here (Jacobsen keeps the case details clearly accessible on the JMRI web site). Jacobsen connects his openness about the case to principles he sees as central to open-source software and academic

---

^9^ Patent # 6,384,822 issued May 2, 2002, which can be found by searching the USPTO’s web site: [http://patft.uspto.gov/](http://patft.uspto.gov/).
research in physics: “the idea of doing everything in daylight, the more information people have, the better decisions they can make.” Because of his steadfast attachment to these principles—even when contrary to advice from his lawyers—Jacobsen has become a small hero to the anti-software patent community. His case has become a touchstone for debates about software patents for open-source projects in particular as the Slashdot and Groklaw communities root for him to prevail against Katzer, the “patent troll” (see, for instance, Jones, 2006; “On Software Patent Lawsuits,” 2006).

Although the principle of “doing everything in daylight” is common to Jacobsen’s approach to his case and the JMRI project, Jacobsen draws a distinct line between his disclosure of the case, which is not designed to allow others to participate in it, and his desire to encourage participation in the open-source project. He cites the active and diverse forms of participation in the JMRI project as one of its strengths:

JMRI is successful at attracting developers [because. . .] the things that it does [are] really sort of cool—it makes trains do stuff. It makes a physical object move. There’s a growing list of people who build little robots because the technology is becoming more and more available. I’ve got a 7 year old who likes to do that. And it’s really becoming more accessible as computing becomes more accessible.

With project participants such as retired dentists, former DEC Corporation programmers, 7-year-old kids, teenagers, and nuclear physicists at UC-Berkeley, JMRI is a window on the diversity of people now coding for open-source projects.

This diversity of programmers includes game designers such as Laughlin and Carmack as well, who, like Jacobsen, compose their software in contexts that do not seem to be well suited to patent law. Pointing out the differences between the contexts in which “software engineers” program and what he as an independent game designer does, Laughlin said, “for games especially, it’s barely even obvious that monetary incentives from copyright are even necessary for people to make things. So the idea that patents [would be necessary to reward innovation]?—I’ve seen nothing to indicate that.” Corporate software engineering, such as what is practiced by IBM and Microsoft (both owners of massive patent portfolios), may benefit from patent protection. However, software composed in corporate contexts must, by nature of patent law, co-exist—and compete—with the many informal, loosely organized sites of code composition, which puts those independent projects at risk.

6. Conclusion

Digital composition teachers and scholars are, admittedly, more insulated than professional or even hobbyist programmers from software patent infringement. For public relations as well as financial reasons, IBM is unlikely to prosecute a single composition teacher or even a university for low-grade patent violation through distribution, use, or writing of infringing code.10 However, as our scholarly work and our students’ compositions become more dependent on robust and diverse software programs, our exposure to the problems created by software patents increases. Moreover, as the computer blurs the lines between composition modalities, code is no longer reserved for specialists; our more technically sophisticated works already make us into programmers. Code has become infrastructural to the way we compose and, indeed, to the very way that we live our lives. Software mediates almost all of our financial transactions, our communications, and our information outlets and inlets. Behind each program are millions of lines of code written by thousands of professional and hobbyist programmers in hundreds of different languages. Because patent law is nearly impossible for non-specialists to take advantage of or even interpret, it does not mesh well with a decentralized composition scenario.

At the heart of the debates about patenting software are the questions: What is code? What is writing? Where do we draw the line between the two? Computer code is text that enacts processes, and those processes often have more tangible results than textual writing. But as our compositions move beyond text and now include many different modalities and forms of expression as writing, the line between code and text all but vanishes. As Cummings (2006) wrote, “the time has arrived for composition scholars to claim coding as their own. It is writing. Once we are comfortable with that idea, then we will find a new arena for the excitement of composition” (p. 442-3). Software patents frame

---

10 Companies less concerned with public relations, such as the one suing Bob Jacobsen, are a wild card in this situation and may pose a greater threat.
coding as a kind of engineering, and in some cases it is. But coding is also a form of writing. One exciting new area for composition studies is the way that framing coding as writing can provide support for the movement against software patents. Just as Lessig (2001) and others have noted that copyright law no longer fits the remix model of multimodal composition, we can show that patent law no longer fits the current and diverse contexts of code composition.

Annette Vee recently received her PhD from the University of Wisconsin-Madison and is now assistant professor of English at the University of Pittsburgh. She is interested in digital pedagogy, games, computer programming, and the technologies of literacy in history. Her work has appeared in JAC and The Writing Lab Newsletter.

References


Bilski et al. vs. Kappos 561 U.S. _____, (2010). [[[Editors: New reference to reflect the 6/28/10 Supreme Court decision. The weird underline is because the decision isn’t officially published yet. The online publication of the decision requests that it be cited just like that. See: http://www.supremecourt.gov/opinions/09pdf/08-964.pdf]]]


In re Alappat. 33 F. 3d 1526 (1994).

In re Bilski. 545 F. 3d 943 (2008).


 gradcam.html

interview

Boot

The

John

Deep

1997

in

Carmack

Issues in Science and Technology


In re Alappat. 33 F. 3d 1526 (1994).

In re Bilski. 545 F. 3d 943 (2008).


