In “The Software Patent Experiment,” Bob Hunt and Jim Bessen investigate whether the act of patenting software achieves its stated objective – increasing innovation. They do this by looking at which companies get the most patents, and at the relationship between patenting and investment in R&D. Patenting has been successful in encouraging innovation in many industries, but has not been empirically tested for the software industry, despite the authors’ estimation that the USPTO “grants at least 20,000 software patents a year” (pg. 22). The authors provide a definition of a patent as an invention that is “useful, new, and non-obvious,” but they also state that some judicial decisions on specific patents have made it easier to pass the non-obvious test and easier for the holder of a patent to prevent others from infringing on it even before the court has decided whether the infringement is actually occurring (pg. 23). A software program was deemed unpatentable by the US Supreme Court in 1972, but in 1981 they ruled that an invention including a software program could be patented, paving the way for more and more exceptions until software became completely patentable following a decision in 1994 to uphold the patentability of a program.

Software patents are rising rapidly – from 2% of all patents in the 1980s to around 15% in 2002 (pg. 24). The authors found that the majority of software patents are held by American inventors and that most software patents go to large, established firms, not individual inventors. Surprisingly, most software patents don’t come from the software industry, but from the manufacturers of machinery, electronics and instruments, which combined held 66% of software patents granted to firms (pg. 26). The authors surmised that this is because these companies were more likely to patent in general, regardless of how much money they spent on R&D. They conclude that “the pattern of software patenting across US firms seems to be more closely related to industry-wide variation in the utilization of patents in general than to the resources devoted to creating software” (pg. 27). They found that increases in R&D and in the employment of programmers only accounts for 1/3 of the rate of increase in
software patents, so the rest must be attributable to an increase in the profitability of obtaining software patents. The profits that companies derive from software patents should theoretically be invested in more R&D, but the authors found weak evidence of this in the 1980s and contradictory evidence in the 1990s (pg. 28). The authors conclude that of the three theories present in the debate over the patent system, only one is consistent with their data – the theory of patent thickets, which says that large firms accumulate large patent portfolios to use to obtain royalties from their competitors and as leverage against their competitors doing the same.

I was surprised by the authors’ finding that software firms are not the largest holders of software patents, and worried by their conclusions about patent thickets. If patents are being used by large firms to extract royalties and establish monopolies, they are not only failing to serve their original purpose, but are in fact working against this purpose!

Richard Stallman’s speech on software patents gives an overview of the patent system as it applies to its “victims,” or someone “who wants to develop software but is forced to contend with a system of software patents that might result in getting sued.” If you want to write a software program, you should first find out what patents are already held, which is impossible because those patents that are pending are kept secret. By the time you finish writing, a patent may have been issued covering algorithms in your program. Furthermore, there are so many software patents and they are written in such complicated language that it would be impossible to read those that do exist.

If you still want to try, you have three options: avoiding the patent, licensing the patent, and overturning the patent in court. Avoiding the patent is avoiding the idea covered by it, which is extremely difficult in software writing because some patented ideas are extremely broad and could rule out an entire field of writing. It is also difficult because if certain algorithms are patented and you try to leave them out of your program, it will be detrimental to the software, resulting in a useless program.

The second option is to try to get a license for the patent allowing you to use the idea, an action that is
not required of the patent holder. Those that do offer licenses often charge a large amount of money for them, which becomes impossible if you need to license several patents. Multinational companies benefit from this system because they all possess large patent portfolios and can cross-license, collecting royalties from each other and getting access to others patents at the same time. Stallman points out that this does not protect the individual inventor, especially in high-tech fields where “ideas don’t come in a vacuum.” The last option is overturning the patent in court by proving that it is not new, useful or non-obvious. While patents that do not satisfy these criteria are regularly issued, it is extremely expensive to go to court in order to get a judge to take a closer look at a specific patent. Stallman describes the software patent system as a “minefield” – you may be able to use one of these three options, but with every step you take, a new patent with new complications will pop up.

To explain why the software industry is different from others in terms of eligibility for patenting, Stallman explains that software packages are usually very large, consisting of many different ideas compiled in a new way. Since there are potentially thousands of different parts to a software package, there are thousands of pieces that could have already been patented and thousands that could cause the downfall of your program. He explains this further using the analogy of symphonies, which pull together thousands of different notes that are already defined as “musical.” If these musical notes were patented, a composer would not be able to make a symphony, and in order to create something entirely new, he would have to redefine music altogether! I was struck by this idea of “incremental innovation,” which Stallman uses to set the software industry apart. It is absurd to think that people can patent the building blocks of a software program, just as it would be absurd for someone to patent the steel used to make a new type of vehicle.

James Boyle’s “What Intellectual Property Law Should Learn from Software” takes a look at the evolution of the software industry in terms of intellectual property as it applies to both copyright and patents. He echoes Stallman’s point that software is built upon specific prior “art” in the form of lines of
code and that in this field you cannot create something out of nothing. Despite this apparent contradiction with the idea of copyright, the implementation of copyright law to software has been successful in protecting ideas while avoiding prohibiting innovation. “Courts interpreted the copyright over software narrowly,” which avoided prosecuting people for simply using similar code or producing similar programs. They also used the fair-use policy to permit something called “decompilation” which allows competitors to take apart a program in order to find out how it works and then develop a competing program. Though this seems unfair, it supports the purpose of intellectual property rights, which is to encourage innovation and competition. Copyrighting of software has arguably caused the creation of free and open-source software by forcing contributors to maintain the open character of the program and make public any changes.

The patenting of software, on the other hand, has allowed programmers to patent the ideas that make up computer programs. Judges are supposed to look at the purpose of intellectual property rights when they make decisions on allowing something to be protected by these laws, but they failed to do so in the case of patents on software, and allowed the non-obvious standard to be met too easily. Both copyright and patents have to some extent allowed the creation of legal monopolies in the software domain, even though the purpose of intellectual property rights law is sound. We need more empirical research on this subject before we extend the laws any further.

This article put the two intellectual property laws covering software into perspective for me. The fact that software is covered by both copyright and patent law highlights its complexities and the difficulties of defining it in terms of intellectual property. This article was also unique and informative for me in pointing out some of the benefits intellectual property law has conferred on the field of software (e.g. the self-perpetuating commons), something that other articles missed.