-----Original Message-----
From: Pamela Jones
Sent: Friday, March 15, 2013 7:48 PM
To: SoftwareRoundtable2013
Subject: Groklaw Corrected Submission re Topic 2

Gentlemen,

I apologize. We found errors in the version of Groklaw's response to the USPTO's request for topics for future discussion by the Software Parnership that I sent you earlier today.

Attached is the corrected version.

Pamela Jones
Founder and Co-Editor
Groklaw
http://www.groklaw.net

Groklaw's Response to the USPTO on Topic 2:
Suggested Additional Topics for Future Discussion by the Software Partnership

In response to the USPTO's request for topics for future discussion by the Software Partnership, the technical community at Groklaw suggests the following four topics, in order of priority:

> 1: Is computer software properly patentable subject matter?
>
> 2: Are software patents helping or hurting innovation and the US economy?
>
> 3: How can software developers help the USPTO understand how computers actually work, so issued patents match technical realities, avoiding patents on functions that are obvious to those skilled in the art, as well as avoiding duplication of prior art?
>
> 4: What is an abstract idea in software and how do *expressions* of ideas differ from *applications* of ideas?

In order to explain why these topics could be fruitful, here are some brief thoughts in explanation. A more detailed explanation, with references, can be found here.

Suggested topic 1:
Is computer software properly patentable subject matter?

If software consists of two elements neither of which is patentable subject matter, can software itself be patentable subject matter?

Software consists of algorithms -- in other words mathematics -- and data, which is being manipulated by the algorithms. Mathematics is not patentable subject matter and neither is data. On what basis, then, is software patentable subject matter?

We would welcome a discussion on this topic, as it is a key issue to the developer community. Note that Groklaw has published a number of articles on this topic, which can all be found at here on Groklaw.

A particular point of interest is how the meaning of data influences the patentable subject matter analysis. Computers manipulate bits, and bits are electronic symbols which are used to convey meaning. In some patents, such as in *Diamond v. Diehr*'s industrial process for curing rubber, what this meaning signifies is actually claimed clearly. In *Diehr's* case the rubber is cured. But in most pure software patents the meaning is merely referred to. Should this distinction influence whether the claim is patentable? We will return to this question in more detail, under the headings of Suggested topic 3 and Suggested topic 4.

Suggested topic 2:
Are software patents helping or hurting innovation and hence the US economy?

It would be useful to hear from entrepreneurs on a wide scale on the effects software patents are having on their startups or business projects. Microsoft's Bill Gates himself has stated that if software patents had been allowed when he was starting his business, he would have been blocked.[1] Is that happening to today's entrepreneurs? If software authors are unable to clear all rights to their own products because there is no practical way to do so, how can such a situation foster progress and innovation? Rather it seems to force developers, or the companies that hire

---

1 "Challenges and Strategy" (16 May 1991). Gates exact words were:

> "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."

Also found at http://bat8.inria.fr/~lang/reperes/ local/Challenges.and.Strategy.

them, to choose either to go ahead and develop innovative products with the certainty that if it is successful there will be infringement lawsuits or stop developing innovative products altogether.

Every firm with an internal IT department writes software. Every firm which maintains it own website writes software. There are roughly 634,000 firms in the United States with 20 or more employees and 1.7 million firms with 5 to 19 employees. A very large fraction of these firms write software. In an ideal world, all firms should verify all patents as they are issued to avoid infringement. This need to verify the relevance of all patents would necessarily be a constant, on-going activity. For one thing, corporate software must frequently be adapted to new needs and any new version may potentially infringe a patent not previously infringed. A study has concluded the task is practically impossible to accomplish.[2]

Even if a patent lawyer only needed to look at a patent for ten minutes, on average, to determine whether any part of a particular firm's software infringes, it would require roughly 2 million patent attorneys, working full-time, to compare every firm's products with every patent issued in a given year.

This is an impossibility, because there are only roughly 40,000 registered patent attorneys and patent agents in the US.

The above estimation covers just the work of keeping up with newly issued patents every year. Checking already issued patents would require even more attorneys.

Looking at the situation from yet another perspective, let us compare lines of code with sentences in a book. Each English sentence expresses an idea. Each combination of sentences expresses a more complex idea. Then more and more complex ideas are expressed in paragraphs, chapters etc. The total number of ideas from all works of authorship is extremely large. Imagine a hypothetical intellectual property regime where all such ideas are patentable. This would generate a large number of patents, with all authors having to check all the issued patents for potential infringement, with more patents issuing every year.

It is clearly impossible to promote innovation with such a system that is not practically functional, but that is the situation software developers face, one where they have no practical way to verify they own all rights to their own work. Such a system is guaranteed to harm the economy with monopolistic rent-seeking and unneeded litigation, which is what we are currently witnessing.

Suggested topic 3:

How can software developers help the USPTO understand how computers actually work, so issued patents match technical realities, avoiding patents on functions that are obvious to those skilled in the art, as well as avoiding duplication of prior art?

The current interpretation of patent law is riddled with what developers view as fundamentally erroneous conceptions of how computers work. Other than the current USPTO request for input, developers feel shut out of decisions, decisions made without their contributed knowledge and skill, yet considered legally binding precedent despite violating technical reality, and yet the practitioners in the field are the very ones who best understand what software is and how it does what it does.

Textbooks describe in detail what mathematical algorithms are, but case law doesn't seem to understand or to reference these sources. Instead, we see courts using standard dictionary definitions. These definitions are too succinct and incomplete, at best. The result is confusion about what algorithms are.

For an example, courts have made an unrealistic distinction between so-called mathematical

2 See Mulligan, Christina and Lee, Timothy B., Scaling the Patent System (March 6, 2012). NYU Annual Survey of American Law, Forthcoming. Available at SSRN: http://ssrn.com/abstract_id=2016968. The quoted paragraph is at pages 16-17.

algorithms and computer algorithms that purportedly are not mathematical. The field of computer science itself recognizes no such distinction, but the legal environment surrounding software patents ignores what mathematicians and computer scientists say about algorithms. Since the ensuing descent into surrealism directly impacts the controversial question of when a computer-implemented invention is directed to a patent-ineligible abstract idea, a serious problem is caused, which could have been avoided by a deeper, more accurate technical understanding.

Second, it seems some, including some courts, believe the functions of software are performed through the physical properties of electrical circuits, incorrectly treating the computer as a device which operates solely through the laws of physics. This approach is factually and technically incorrect because not everything in software functions through the laws of physics. Indeed, bits in a computer are constructed and manipulated by the use of physical laws. However, bits are also symbols. They have meanings which are assigned by human beings. The meaning of bits is essential to performing the functions of software. The capability of bits to convey meaning is not a physical property of the computer.

Software developers don't write software by working with the physical properties of circuits. Developers define the meaning of data and implement operations of arithmetic and logic that apply to the meaning. They debug software by reading the meaning of the data stored in the computer and verifying whether the correct operations are performed. Again, the aspects of software related to meaning cannot be explained solely in terms of the physical properties of the computer.

This erroneous physical view of the computer is the basis of an oft-stated argument. Some have claimed that software alters the computer it runs on, thus creating a "new machine". (See *In re Bernhart*, 57 C.C.P.A. 737, 417 F.2d 1395, 1399-1400, 163 USPQ 611, 615-16 (CCPA 1969) --"[I]f a machine is programmed in a certain new and unobvious way, it is physically different from the machine without that program; its memory elements are differently arranged.")

This belief is used to justify the view that software patents are actually a subcategory of hardware patents, making software patentable almost without restriction or restraint, in that all software runs on a computer. To demonstrate what is wrong with that argument at its very foundation, let's compare a printing press with a computer.

It is easy to see that the content of a book is not a machine part. The meaning of a book is not explained by the laws of physics applicable to a printing press. But the comparison of a computer and the printing press shows that there is no material difference in their handling of meaning. Any argument related to meaning which is applicable to a printing press is applicable to a computer and vice-versa.

Imagine a claim on a printing press configured to print a specific book, say Tolkien's *The Lord of the Rings*. This is a claim on a machine which operates according to the laws of physics. Printing is a physical process for laying ink on paper. It functions without the intervention of a human mind. But still this process involves the meaning of a book. Such a claim could only be infringed if the book has the recited meaning.

One could argue that a configured printing press is physically different from an unconfigured one. The configured printing press can print a specific book while the unconfigured one cannot. Books with different contents are different articles of manufacture. Differently configured printing presses perform different functions, because they make different articles of manufacture. Therefore, as this hypothetical argument goes, a printing press configured to print a specific book has become a specific machine which performs a specific practical and useful task and, lo and behold, the result is a "new machine test" for printing presses. However, the fact that no one in the real world would accept a world in which a printing press becomes a new machine every time it is set up to print a new book is quite sobering. Or ought to be. Because this is the fallacious argument used to justify that a computer configured with software becomes a new machine.

Software patents are often written similarly to this analogy. Like a printing press, the computer

operates according to the laws of physics. It functions mostly without the intervention of the human mind, although from time to time human input may be required. But the process of computing needs the meaning of the data to actually solve problems. The claim is infringed only if the data has the recited meaning.

The argument that a programmed computer is different from an unprogrammed one is exactly symmetric to the one we have just made about printing presses. There is a reason for this. The technologies are not that different. Further underscoring the similarity, a computer connected to a printer can be configured to print a book. And modern printing presses may be controlled by embedded computers.

There is no material difference between a configured printing press and a programmed computer in their handling of meaning. Users of a computer read the meaning of outputs. They also enter the inputs based on the meaning. When programming a computer, programmers must define the meaning of data. They implement algorithms which perform operations of arithmetic and logic on the meaning of this data. When debugging, programmers must inspect the internals of the computer to determine whether the correct operations are being performed. This requires reading the contents of computer memory and verifying it has the expected meaning. In other words, the act of making the invention depends on defining and reading the data stored in the computer. Software works only if the data has the correct meaning.

The output of a printing process is a book. Different books are distinguished by their contents. A typographer must define and verify the contents of the information to be printed to configure the printing press correctly. In other words, the act of making the invention depends on defining and reading the data stored in the printing press. A printing press works precisely because it prints the right contents. Printing makes a physical book which can be read and sold. Books with different contents are different books. A wrongly configured printing press prints the wrong book. Therefore the utility of the printing press doesn't depend just on the laws of physics. It also depends on the contents of the book.

Both machines work in part according to the laws of physics and in part through operations of meaning.

The courts have failed to acknowledge the role of *meaning* in software. Some errors result from the failure to take into consideration the descriptions of what is a mathematical algorithm in mathematical literature. Other errors result from explicitly and incorrectly denying the role of symbols and meaning in computers. And more errors result from the belief that computers operate solely through the physical properties of electrical circuits, in isolation from the meanings assigned by human beings.

Imagine now that every time a printing press prints a new book, you could patent that printing press as a new machine because it printed a new book. That is exactly what patent law does with software, purporting to create a new machine because new software running on the computer supposedly creates a new machine. And yet the computer can run any software at all that you can devise, just as a printing press can print any book you write. The computer can, in fact, run more than one program at the same time. Is it now two new machines? And if you remove one software program, now what is it? And when the computer is done with the job, it is still the same old computer, just as when it is done with its job, the printing press is still the same old printing press.

No one would allow a patent on a previously existing printing press just because it is now configured to print a new novel. Yet that is exactly what is allowed with software.

The consequence is a proliferation of patents on the expressions of ideas, on "doing so-and-so on a computer," and, even worse, on the concept of "doing so-and-so on a computer" when the procedure in question merely incorporates ideas and methods which may date back centuries or even millenia.

<u>Suggested topic 4</u>:
What is an abstract idea in software and how do *expressions* of ideas differ from *applications* of ideas?

Abstract is not synonymous with vague or overly broad. A mathematical algorithm is narrowly defined with great precision, but still it is abstract.

Abstract is not the opposite of useful. The ordinary procedure for carrying an addition is a mathematical algorithm. It has a lot of practical uses in accounting, engineering and other disciplines. But still it is abstract. In particular it is designed to handle numbers arbitrarily large no matter whether we have the practical means of writing down all the digits. Besides, there are useful abstract ideas outside of mathematics. For example the contents of a reference manual, such as a dictionary, are both abstract and useful.

Mathematics is abstract in part because it studies infinite structures. For example, the series of natural numbers 0, 1, 2, ... cannot exist in the concrete universe, because it is infinite. Also, symbols in a mathematical sense are abstract entities distinct from the marks on paper or their electronic equivalent. For example, there are infinitely many decimals of pi even though there is no practical way to write them down. Infinity guarantees that mathematics is abstract. Therefore a definition of "abstract ideas" must acknowledge the abstractness of mathematics.

A proper understanding of the role of meaning is key to understanding when a claim is directed to a patent-ineligible abstract idea in software. Software patents don't claim abstract ideas directly. They claim them indirectly through the use of a physical device to represent them by means of bits. It would be easier to recognize claims on patent-ineligible abstract ideas if it were understood they take the form of claims on *expressions* of ideas as opposed to *applications* of ideas. The bits are symbols and the computation is a manipulation of the symbols. Expressions of ideas occur through this use of symbols.

This suggests a test similar to the printed matter doctrine. This test is best described using the concepts and vocabulary of a social science called semiotics. This science studies signs, or symbols, used to represent something else. We suggest it can be used to distinguish patent-eligibility in software.

Computers should be recognized to be what semioticians call sign-vehicles, physical devices which are used to represent signs. The sign itself is an abstraction represented by the sign-vehicle. Hence, sign-vehicles and signs are distinct entities.

Semiotics distinguishes between two types of meaning. There is the actual worldly thing denoted by the sign. This is called the referent. And there is the idea of that thing a human being would derive from reading the sign. This is called an interpretant. A sign usually conveys both types of meanings simultaneously. An example might be a painting representing a pipe. The painting itself is a sign-vehicle. People seeing this painting will think of a pipe. This thought is an interpretant. An actual pipe is a referent.

If nothing has been invented but thoughts in the mind of human beings, one should not be able to claim a sign-vehicle expressing these ideas as if it were an application of the ideas. But when the real thing denoted by the expression is claimed, we may have a patentable invention. In other words, one should be able to patent a particular pipe invention, but not the painting of that invented pipe.

These ideas lead to this test: A claim is directed to a patent-ineligible abstract idea when there are no nonobvious advances over the prior art outside of the interpretants. A claim is written to an application of the idea when the referent is claimed instead of merely referenced.

For example a mathematical calculation for curing rubber standing alone is not patentable under this test. It is just numbers letting a human think about how rubber should be cured. But when the actual rubber is cured the referent is recited and the overall process taken as a whole may be

patentable.

This test is technology-neutral. It is applicable precisely when the claimed invention is a sign, or when it is a machine or process for making a sign. It applies whether the invention is software, hardware or some yet to be invented technology. This test works without having to define the boundary between what is software and what isn't.

The concepts of semiotics are quite simple and easy to define. They are related to the dichotomy between ideas and the specific expression of ideas in copyright law. Therefore this test for abstract ideas helps clarify the line between what should be protected with copyrights and what belongs to patent law. The expressions of interpretants may be protected by copyrights and the corresponding referents may be protected by patents.

This test will correctly identify abstract mathematical ideas. Mathematics is, among other things, a written language. It has a syntax and a meaning which are defined in textbooks on topics such as mathematical logic. Algorithms are features of this language. They are procedures for manipulating symbols.[3] They solve problems because they implement operations of arithmetic and logic on the meaning of the symbols. Algorithms are also procedures which are suitable for machine implementation. Computer programs may solve a problem only if it is amenable to an algorithmic solution. In this sense, all software executes a mathematical algorithm.

Mathematical language refers to abstract mathematical entities such as numbers, geometric shapes, etc. We assimilate this abstract meaning with interpretants. Mathematical language may also be used to describe things in the concrete world, for instance using laws of physics. The corresponding referents are applications of mathematics. Mathematical algorithms and other types of mathematical subject matter are a subcategory of interpretants. And things in the concrete world modeled using mathematical language are a subcategory of referents. Hence the proposed test will properly distinguish between the expression of a mathematical idea from an application of the same idea. Claims of applications are to be accepted, while claims on expressions should be rejected.

3 Cf. Stoltenberg-Hansen, Viggo, Lindström, Ingrid, Griffor, Edward R., *Mathematical Theory of Domains,* Cambridge University Press, 1994,, page 224,
and also Boolos George S., Burgess, John P., Jeffrey, Richard C., *Computability and Logic, Fifth Edition*, Cambridge University Press, 2007, page 23