From: Eric Johnson [e-mail redacted]
Sent: Monday, September 27, 2010 3:11 PM
To: Bilski_Guidance
Subject: Bilski guidance - eliminate or reduce software patents

To whom it may concern,

It has come to my attention that you're soliciting opinions as to guidance you might suggest in the wake of the Bilski decision from the Supreme Court. I'll state up front that these are my personal opinions, and not those of my employer, TIBCO Software Inc [2].

As a software developer of over two decades, I've been dealing with software patents for quite some time now. As part of my first full-time job as a developer, back in 1992, I put my name on a software patent, which eventually issued [1]. Now, I'm a member of my company's "patent committee" which decides whether or not we should encourage employees to pursue particular patents, and have been involved in assessing the concerns around a number of situations where third-parties have alleged that we've infringed our patents.

My conclusion? In all cases, this has been an enormous waste of time.
The efforts to create patents have not contributed to either me, or any of my co-workers trying harder to making any of the software I've been involved in more innovative. The third party allegations (that I've
seen) have been either baseless, or readily interpretable via any perspective, due to the arbitrary and abstract nature of most computer technology. In one case, I determined that a third-party allegation of infringement was completely undermined by an equally valid patent issued by the USPTO which would serve as prior art for the salient areas of alleged infringement. And finally, my company has specifically banned us from actually looking at patents, due to the possibility of treble damages. This last point, of course, completely undermines the notion that patents are furthering the advancement of the art. All we've done, it seems, is funnel money to lawyers. An innovation tax, if you will.

What makes software patents particularly problematic is that in software it doesn't matter if you call something a "Chair", or a "Car" - what matters is that for the intended purpose, the "objects" share the same salient details. Since the field changes so rapidly, we don't tend to have common terminology for well-agreed upon concepts. And, unlike the physical world, every aspect of an "object" in a computer is in turn itself abstract. This makes it impossible for me to look at a patent and recognize whether or not the "chair" in a patent is the same as my "car" in my product, unless I can do a point-by-point comparison of the salient details of said objects, whereas this question is relatively easy to assess - in most cases - in the physical world. Since the salient details suffer from the *same* constraint, doing this recursive analysis, as to whether "chair" and "car" are the same - for a particular context - is a herculean or even impossible task.

If you look at some of the programming languages currently all the rage
- such as Python [3] or Ruby [4] you'll notice that they all share a characteristic known as "duck-typing" [5]. This means that it doesn't actually matter what you say something "is", all that matters is what it "does", and since you can change what it "does" by changing the implementations of what an object depends upon, everything, it turns out is virtual.

If software patents are allowed at all (and I don't think they should be), about the only circumstances I can imagine they would work:
* relate to software for particular-purpose machines that actually *interact* with the world - such as manufacturing equipment or other robotic technology, and only as part of a larger patent. It is not sufficient to display data to the screen, or send/consume data from a network.
* the software part of a patent should be self-executing - that is the software part of a patent should be demonstrably implemented *in code* as part of the patent (note that, as per previous point, the software should only be a part of the patent).

To the extent that I can go to the store, buy a computer (or programmable phone!) and download software onto that machine, what that program does ought not be patentable. It is already

covered by copyright law, and having *two* legal frameworks covering the same code makes everyone's life much more difficult. It is just bits in, bits out, and that, fundamentally, comes down to math.

A note about this notion of self-execution. If the *code* is actually demonstrated, then it becomes dramatically easier to find prior art, and/or distinguish from the prior art by highlighting the differences from existing patents. It also makes it easier to understand a fairly fundamental point - the actual scope of a claim. And of course, the question of whether or not something can actually be "practiced" would be immediately clear by the existence of functioning software. Since software can be rendered in source code form, it is kind of shocking to me that this has not been required up until now.

My conclusion? Either don't allow software patents at all, or if you do allow them, require that code be included, and that the code doesn't actually perform the patent unless coupled with a specific physical device that manipulates physical objects in the world around it, such as a robot.

Sincerely,

Eric E. Johnson
6732 Vicksburg Pl.
Stockton, CA 95207

References:

[1]
http://patft1.uspto.gov/netacgi/nph-Parser?Sect1=PTO1&Sect2=HITOFF&d=PALL&p=1&u=%2Fnetahtml%2FPTO%2Fsrchnum.htm&r=1&f=G&l=50&s1=5,867,150.PN.&OS=PN/5,867,150&RS=PN/5,867,150
[2] http://www.tibco.com/
[3] http://www.python.org/
[4] http://www.ruby-lang.org/en/
[5] http://en.wikipedia.org/wiki/Duck_typing