Embedded Technical Expression:

Code and the Leveraging of Functionality

Matt Ratto
Networked Research and Digital Information (NERDI)
Royal Netherlands Academy of Arts and Sciences

Abstract
In this paper, I argue that a key problem for understanding the social ramifications of software is an implicit separation of software programs into two separate elements: expressive aspects which are seen as socially driven and affectual, and functional aspects understood as scientific and rational. In order to overcome this binary, I develop an alternative framework that examines the ways software expresses best practices, correct behaviors, and social organization. A key element of this framework is an understanding of software programs as simultaneously rhetorical and tangible objects. Given that western society is dependent on socio-technical systems in which software plays a crucial role, understanding the complex ways software expresses normative tasks, practices, and social orders is increasingly important.

Keywords: software, code, copyright, patent, expression, functionality, science and technology studies, Linux, free/open source.

Running Head: Embedded Technical Expression

The author would like to thank Anne Beaulieu, Geof Bowker, Jenny Fry, Tarleton Gillespie, Roberta Lamb, Harmeet Sawhney, and Paul Wouters for their comments and helpful advice on draft versions of this article.

Address Correspondence to: Matt Ratto, KNAW-Royal Netherlands Academy of Arts and Science, Joan Muyskenweg 25, Postbus 95110, 1090 HC Amsterdam, The Netherlands. Email: matt.ratto@niwi.knaw.nl
Introduction
In this paper, I argue that a key problem for understanding the social ramifications of software is an implicit separation of software programs into two separate elements: expressive aspects which are seen as socially driven and affectual, and functional aspects understood as scientific and rational. This separation, which mimics older philosophic perspectives (such as Platonic idealism) in separating appearance from ideal characteristics, locks off some of the most effectual aspects of software programs from critical analysis. In order to explore how this separation is typically analyzed, I examine its expression in legal scholarship within the US. My point is that while the expression/function binary may appear useful in thinking through issues of how software “fits” within existing systems of intellectual property (IP) regulation, it is much less useful in thinking about the larger issue of the relationship between software and society. My intent in describing the legal scholarship on software and IP is to use this debate to reflect on the limitations of the expression/function dichotomy both for legal scholars and for others studying the relationships between software and society.

In order to overcome this troublesome binary, I lay out a different analytic framework for understanding the embedded technical expression (Burk, 2000) of software. Using this framework, I explore how the Linux software program expresses normative practices of programmers, the tasks of users, and the social organizations of both.

Why is this important?
Software plays an increasingly active role in the maintaining of current social relations in addition to providing for the possibility of social change. From a critical perspective, legal scholars have posited that recent technological innovations such as digital rights management (DRM) entails a move from public to private modes of intellectual property control. (e.g. Reidenberg, 1998; Lessig, 1999; Samulson, 2003, Gillespie, 2004) Other scholars see the accumulation of personal information within large-scale corporate or government databases as
posing a threat to socially granted individual rights. (e.g. Agre, 1998) Still others explore the development of distributed technologies such as email, the web, and more generally the Internet, as creating novel forms of the individual and social self and new types of organizing and relating. (e.g. Turkle, 1995)

However, figuring out how to address software has continued to be a problem. Legal scholars have struggled for 40 years trying to address software's nature as both an act of expression (a text) as well as a functional device (a machine). The dual aspect of software is particularly troubling for legal regulation due to a pre-existing separation between expression – protected under copyright, and the objects of innovation – protected under patent law. Thus, while few legal scholars would debate the claim that software is both text and machine, there has been increasing pressure to place software under a single regime of IP protection. Currently, copyright is the most often used means of IP software protection. While the patenting of some aspects of computer programs (notably algorithms) remains a hotly debated topic, (e.g. Burk, 2000; Samuelson, 1990), the focus on software copyright has also meant ongoing legal debate as to what precisely software expresses. This debate has centered around whether or not software is eligible for the protections given to expression within the First Amendment to the US constitution.

**Embedded Technical Expression**

The Supreme Court has ruled that while “it is possible to find some kernel of expression in almost every activity a person undertakes…such a kernel is not sufficient to bring the activity within the protection of the First Amendment.”¹ This court ruling has been used in US cases to demonstrate that while software may have some expressive elements, these elements do not mean that it should be considered “speech” and therefore protected under the First Amendment to the US constitution, which guarantees freedom of expression. Legal scholars have debated this
issue since the beginning of software regulation, with such argumentation based on disentangling
the “speech” of software (i.e. its expressive elements) which are protected, from its “actions”
typically seen as software functionality) which are not. Even in an article critical of what he sees
as repressive software regulations\(^2\), Dan Burk states that “embedded technical expression is at
best the sort of kernel described by the Supreme Court, and not the type of expression shielded
by the First Amendment.”(Burk, 2000, pg. 14)

Burk’s statement is revealing\(^3\) of the classic liberal separation between expression and
action, or ‘word’ and ‘deed’ that is typically used to evaluate whether or not some activity is
protected under First Amendment law. However useful this division is for regulators, it is this
very separation between the ‘action’ or functional aspects of software, and the ‘words’ about
software, more traditionally expressive aspects, that must be addressed in order to examine how
software is socially meaningful. Despite the above statement, Burk also acknowledges this,
saying:

Although the value-ladenness of technology is described in expressive terms,
values embedded in technology are primarily expressed through function or

Understanding the “embedded technical expression” of software is key, given the complicated
interwoven nature of expression and action within software. Again, while situating software as
“word” or as “deed” may help legal scholars position software within existing IP systems, such a
binary may in the end reduce our ability to understand its social ramifications. Here, Science and
Technology Studies (STS) can provide insight; scholars who study STS issues have historically
addressed questions similar to the ones facing legal scholars now.
Insights from STS

One such insight from STS is the social nature of technology. While technical decisions are often defended as rational, logical, or pragmatic, the construction of technological apparatus is the result of a complex interwoven process that includes both social and technical factors. (e.g. Haraway, 1991; Bijker, Hughes, and Pinch, 1987; Law and Hassard, 1999) In this case, ‘construction’ refers not just to the actual coding decisions and program writing by which software programs are made, but also to the rhetorical and argumentative strategies that ‘construct’ some software programs as ‘better’ or as ‘more efficient’. More specifically, STS has focused on the co-constructed nature of society and technology, attempting to overcome linear causal arguments about technology “impacts” by demonstrating the iterative “back-and-forth” process of social and technical decisions. (e.g. Callon, 1986; Latour and Woolgar, 1986) In some cases this work has been misconstrued to mean that all technical objects are nothing more than reified social decisions. (e.g. Winner, 2001) But without a critical understanding that sees software construction as both social and technical, software has a tendency to become either nothing more than a blank slate upon which human actors can inscribe whatever they want, or, conversely, a ‘black box’ in which the social decisions disappear and are replaced by seemingly natural, rational, and ‘scientific’ determinations. Understanding how software is socially meaningful requires attention to both.

Here, the recent “materialist turn” in contemporary social theory and STS may prove useful. (Pels, Hetherington, and Vandenbergh, 2002) This move takes up the question of social order and ascribes a much greater role to the tangible objects that make up our daily lives. Focusing on material objects as effectual means of managing social order helps us walk the fine line between the determinist perspectives of old that saw humans as the unwitting dupes of built environments with little ability to resist, and constructivist positions that ignore the material reality of technologies. The framework I lay out below attempts to do this by focusing not just on
the textual or machinic qualities of software but also on the structure of the code itself, i.e. how code segments are arranged to construct a complete program. All three of these aspects of source code are the result of social and technical decision-making. Equally, all three of them are expressive of social and technical order.

**Linux as artifact and rhetorical object**

At one level Linux is a software program that facilitates the operation of computer hardware. More specifically, Linux is the kernel of an operating system made up of a number of different software programs that together accomplish the functions of an operating system (OS). While the kernel of an OS is only part of the overall code necessary to support a computer, it is, by far, the most complex and necessary part. However, while the label ‘Linux’ more appropriately refers to the kernel software that resulted from the development effort started by Linus Torvalds, it has come to refer to the many different collections of software that together with the Linux kernel come to form a complete OS. These collections, assembled and distributed by a number of different profit and non-profit organizations are collectively called ‘Linux distributions.’

In another sense, Linux is a rhetorical object that is often used in debates about the Internet, about digital information, about a ‘New Economy’, and about associated distributed work practices. (e.g. Raymond, 2001) Equally, much of the debate about Linux – as with some other software programs – has increasingly taken on a moral tone in which decisions made about the use of Linux or the adoption of the organizational processes that accompany it are defended as morally good or bad. (e.g. Stallman, 1996) Linux then, makes obvious what is equally true of all human-made artifacts. It exists as a technical object that is used both in materially productive tasks (i.e. to operate computer hardware) and in socially productive work (i.e. to make claims about labor and organization).
Linux also serves as a good case to explore the issue of what makes embedded technical expression meaningful because of the public nature of its construction process. Since Linux is an 'open software' project, it becomes possible to view email list posts between developers debating engineering decisions, public ‘manifesto’ documents that attempt to define (or redefine) the norms of the developer community, as well as the formal licenses that mediate developer and developer-user relationships. Proprietary software, (such as Excel) have much more closed development processes making access (and analysis) more difficult. Thus, Open Code programs provide good examples because the social and the technical processes of their construction are more available for analysis – this is one good reason why such development forms may be a more accountable, though not necessarily more democratic approach to software development.

**Software as source code, behavior, and structure**

It is important to understand the embedded technical expression of software, i.e. how software programs express normative positions about users, programmers, appropriate tasks, and social organization. An initial step is to overcome the problematic dichotomy of expression/function that sits at the heart of a common-sense understanding of software. Maintaining the expression/function dichotomy can result in analyses that view the some aspects of the program (notably interfaces) as social and expressive, while seeing the engineering aspects as purely functional and rational – thereby locking them off from analysis. A possible way to overcome this problem is to characterize software differently. Below I lay out one alternative schema and provide some examples based on Linux.

Software is technically expressive in (at least) three aspects;

- **As source code** that expresses ways to accomplish specific programming tasks, i.e. the best practices of programmers;
• Through the behavior of the program that constrains and facilitates actions of its users, i.e. materially expressing appropriate and inappropriate user behavior.

• Through the structure of the code segments that organize the modes of labor by which the program can be built and/or extended by multiple people, i.e. the social organization of programmers.

In the following sections I take each of these aspects in hand. Before beginning it is important to recognize the interwoven nature of software. By this I mean to emphasize that the technical characteristics of software, and the discourse and talk around these characteristics, are difficult if not impossible to separate. I will return to this issue in the conclusion.

**Expression through source code**

This aspect of the expressive nature of software appears to fit most securely into the traditionally legal notion of expression. In the software cases that have attempted to defend the distribution or creation of software programs under the rubric of First Amendment rights the proceedings have focused on how the source code of the program in question has expressed methods of programming. In these cases, the defense of the programs in question as expression have typically involved citing the commentary of programmers embedded in the source code as well as the way the source code expresses ways of accomplishing programming tasks. The US courts’ decisions have typically been based on whether or not the program is primarily expressive or primarily functional, the purpose of the expression cited (i.e. whether or not it was political speech), and whether or not the expressive quality of the source code would be understood by a possible audience.

While the US court decisions to date remain divided whether or not source code meets the expression requirements necessary to be afforded First Amendment protection, it seems quite obvious that source code is expressive. As the Berstein court stated, “The distinguishing feature
of source code is that it is meant to be read and understood by humans and that it can be used to express an idea or a method.\textsuperscript{8}

There are many examples of programmers debating programming methods and ideas within the Linux development community. Below is one example from the Linux Kernel Mailing List (LKML). The LKML serves as the main location where the cooperative work of Linux kernel development takes place.

**First programmer writes:**

if the smp\_function area is busy, then smp\_call\_function() executes the following code:

```c
if (retry) {
    while (1) {
        if (smp\_call\_function\_data) {
            schedule (); /* Give a mate a go */
            continue;
        }
    }
    spin\_lock (&lock);
    if (smp\_call\_function\_data) {
        spin\_unlock (&lock); /* Bad luck */
        continue;
    }
    /* Mine, all mine! */
    break;
}
```

\textit{The Information Society}, vol. 21, No. 3. 2005.
In this email conversation, two programmers on the LKML are using examples of source code to illustrate and debate programming methods. Of particular interest is the tongue-in-cheek nature of the comments (marked in bold) and the way they rely on an understanding of the code in order to translate their meaning. This code snippet is run when another part of the program (the smp_function) is in use by another section of the OS. This can occur because the Linux OS has the capability of running multiple tasks simultaneously (multi-tasking). Because of this, the program must have the capability to negotiate these separate processes (called threads). In the example above, the smp_call_function executes when the smp_function is asked to run but is already in use. It sets up a loop which checks if the smp_function is free (‘Give a mate a go’). If it is not free, it continues waiting (‘Bad luck’) until it becomes free and then continues on (‘Mine, all mine!’). Understanding the code thus requires simultaneously following the code and connecting the comments. The post goes on:

**First programmer:**

e.g what if the current thread is a realtime thread, could this deadlock? Is it possible that schedule() will never return?

**Second programmer:**
The current thread *is* the one calling smp_call_function(). Or are you thinking about calling mp_call_function() from outside a process context? You're not supposed to do that. Therefore it's not a problem ;-}
Here the first poster describes a scenario that might cause the code to break. The current author denies that the code would ‘deadlock’ by pointing to the inappropriateness of the original author’s scenario, i.e. that the code snippet would be called ‘from outside the process context’. The code is thus ‘expressive’ of coding practice in two importantly ramifying and supportive ways; first by ‘breaking’ if used incorrectly, and second through its use in discourse between programmers to define correct programming activity. This aspect of embedded technical expression, i.e. that it leverages talk and technology, is key to understanding how to analyze it.

Expression through program behavior

This aspect of the expressive nature of software is much more problematic in regards to the legal definition of expression. In fact, the type of expression manifested by the behavior of computer programs is exactly the kind of expressive ‘kernel’ that Burk notes is not eligible to be protected under the First Amendment. However, it is important to recognize the way software program behavior communicates relationships between users and tasks by rewarding or punishing uses, similar to the way source code organizes relationships between programmers and acts of programming described above.

All tools ‘afford’ certain uses by the way they reward or punish the actions we take with them. (Gibson, 1977) For example, a hammer provides a way of driving nails that rewards certain ways of accomplishing this task, (e.g. holding the handle and driving the nail using the head of the hammer rather than the opposite.) Software programs have affordances that effect the kinds of practices users do. Microsoft Word, for example, structures the act of writing a letter in part by automating the process. While users can turn off this functionality, the embedding of a specific letter-writing process in the software explicitly expresses the embedded process as authoritative. Further, Word then rewards users that use the automatic letter-writing process by shortening the time it takes to format a letter, albeit in the form the programmers of the Word
software have decided is appropriate. In this way, software programs express appropriate methods of doing tasks to users, while not necessarily constraining user behavior.

   Linux is no different from other tools in this regard. In fact, one of the ongoing debates about Linux involves the way it affords certain types of activities by specific groups of users. One example of this is the Gnome desktop, originally formulated by Miguel de Icaza at the University of Mexico, and since adopted as the primary graphic desktop for Linux by many software development companies such as Redhat and SUN. Gnome provides a graphic interface for Linux which can, in addition to its own interface, also mimic the look and feel of both Macintosh or Windows computer software. This software thus helps previous users of Windows or Macintosh computers to become Linux users and demonstrates that it is appropriate for them to do so. Despite this, Linux has been criticized by many new users. In an email thread called “After 1 week with Linux – licking wounds”, a new user to Linux complained about the difficulty in installing Linux and doing simple tasks like searching for files. He states that while in Microsoft Windows one can use pull down menus in order to do file searches:

   …To do this in Linux, I would have to go and memorize the man page for find, figure out how to do expressions, jot down on a piece of scratch paper the various option flags I'd need, etc... A real pain in the neck, my friend.11

An analysis of responses to this email point up the connection between program behavior and the expression of appropriate tasks and more explicitly, appropriate users. One poster, in a thread titled “Clueless Users are bad for Debian” quoted a famous UNIX programmer who said:

"Windows was designed to keep the idiots away from Unix so we could hack in peace. Let's not break that."12

Another stated:
Got to agree with this. Those that are not willing to do the research should stay with a MS product. A product that has little choice or selection availible within it. And in another thread (coincidentally called by the same name), another Linux user stated his fears about where the Debian Linux distribution was heading:

There are numerable reasons why I believe Linux should remain the difficult, complex, cryptic, command line driven, undocumented, labyrinth of an OS as it's *nix predecessors. What I see happening is a paradigm shifting to user-friendly applications that lose complexity and robustness in the same modification.

In addition to demonstrating how the behavior of software affords certain tasks and users, these posts make obvious how difficult it is to fully separate the behavioral affordances of Linux from commentary about these characteristics. While the writer of the first post sees Linux behavioral affordances as restrictions (i.e. making Linux difficult to learn and use,) the last poster views them as providing power through “complexity and robustness.” This then, is another example of the complex relationship between talk and technology.

**Expression through code structure**

I define code structure as the organization of different code elements within the overall software program. Just as source code expresses appropriate ways to accomplish programming tasks, and software program behavior expresses who should use the software as well as how they should use it, I see code structure as expressive of the social organization of the act of programming.

For example, the concept of Object Oriented Programming (OOP) refers to a programming structure in which individual software processes are organized into discrete software ‘objects’. Such objects can be used multiple times within the same software program,
and also makes the ‘re-use’ of code in other programs much simpler. In addition, most large software programs are organized into such categories as executables, libraries, and databases. Such organizing code structures also help organize the labor that goes into coding, maintaining, or extending programs. In this way, code structure expresses relationships between programmers.

**Structure of Linux**

One of the claims made about Linux is that it is the result of a distributed programming effort. One such structure feature is derived from Linux use of UNIX as a model for the organization of the different elements that make up an operating system. In the UNIX model, the operating system is made up of a series of different programs responsible for different functions, coordinated by a core software program called the ‘kernel’. The Linux OS follows this model.

The independence of the kernel, from the graphic windowing software, from application software means programmers can work independently of each other without much coordination. This is an opposite approach from software such as Microsoft Windows that relies on a closely linked program structure. In fact, claims about the inter-linked nature of Windows was used to defend Microsoft against charges that its bundled Internet Explorer browser software violated a 1995 anti-trust consent decree. A witness for Microsoft, MIT professor Stuart Madnick, testified that removing browser-related code from Windows wasn’t possible because it was located throughout the operating system.¹³ Such interlinking is not the case with Linux, where programmers can create different parts without having to worry too much about interoperability – as long as they can agree upon the standards imposed (in part) by the OS kernel.

Within the Linux development community, the individual programming efforts that create the different aspects of the overall OS are mostly left to their own devices. As long as the resultant code can ‘talk’ to the kernel and does not break any of the standards upon which Linux is based, they are adopted into the different Linux distributions assembled by the different profit
and non-profit Linux distributors. Thus the structure of the Linux distributions, collected around the Linux kernel, but including different OS ‘pieces’ collected by Linux distributors, organizes relationships between programmers somewhat differently than operating systems such as Microsoft Windows or Mac OS\textsuperscript{14}. The disassociated nature of the overall OS means that coordination between programmers can be loosely organized.

Such freedom is less true of the Linux kernel development effort in which inclusions of new code within the kernel is controlled by a collection of gate-keepers with Torvalds himself as the ultimate authority. Still, even the kernel code structure itself organizes certain types of relationships. For example, in 1995, Linus Torvalds with the support of a core group of kernel programmers, rewrote the current Linux code in order to make it able to load different kernel 'modules' depending on what functionality was required. (Welsh, 1995) Having dynamically loadable software modules meant that the Linux operating system could more easily 'adapt' to the changing needs of users. Before modules, Linux users had two choices when setting up a computer running Linux; they could either choose to create a kernel full of extra code in order to prepared for any new hardware they might install, or they could plan on remaking the kernel every time they wanted to make a change to their system. The first choice meant that their operating system kernel would take up extra memory and take longer to load; the second meant that adding anything to their computer would entail a fairly long process. Thus changing the overall structure of the Linux kernel in order to include loadable modules was an obvious choice motivated by technical design issues.
Monolithic Kernel

Kernel w/ Linkable modules

Custom Kernel w/ only necessary modules

[Diagrams showing linkable kernel modules]

But this change in the structure of the Linux kernel also facilitated a more diverse kernel development community. With the addition of separate kernel modules, Linux developers could create new kernel-specific code (such as Ethernet drivers, file systems, or hardware-specific code that must be run as part of the kernel) without having to go through Torvalds himself. As Torvalds said about this change:

With the 2.0 kernel Linux really grew up a lot. This was the point that we added loadable kernel modules. This obviously improved modularity by making an explicit structure for writing modules. Programmers could work on different
modules without risk of interference. I could keep control over what was written
into the kernel proper. *So once again managing people and managing code led to
the same design decision. To keep the number of people working on Linux
coordinated, we needed something like kernel modules. But from a design point of
view, it was also the right thing to do.* (italics mine) (Torvalds, 1999)

The loadable kernel module structure of the Linux kernel code allows for a more distributed
practice of development. Note that this is not the same as arguing that the loadable module code
*causes* a more heterogeneous development group — the relationship between the social
organization of Linus developers and Linux code is not as uni-directional as such a statement
would entail. In fact, it's important to note that the inclusion of modules within the Linux kernel
was itself driven by organizational pressures being put on Torvalds. The interwoven nature of
software code structure as organizing of social relationships and social relationships as
organizing of software code structure is made even more obvious by other reasons for the
module structure of Linux.

Another important aspect of dynamic modules points even more explicitly to the way
code structures relationships. Torvalds' decision to license the Linux kernel under the GPL
(GNU Public License) caused problems for software developers who wanted to create custom
code for Linux, but did not want to make their software publically available. Under the GPL, any
works 'derived from' GPL'd software were themselves automatically covered under the GPL.
Stallman wrote the GPL explicitly to do this in order to prevent proprietary software developers
from taking and using the open source code of GPL'd works without giving the resultant 'derived'
code back to the GPL community. But this restriction created a problem for hardware
manufacturers who wanted to create products for Linux computers (or wanted to make their
existing products work with Linux), but didn't want to make the necessary driver software source
code public. While still a grey area within the GPL, Torvalds decided that dynamically linked modules would not be considered 'derived works' as defined by the GPL. Thus, proprietary developers could create driver modules for Linux without having to make their module code public.

The structure of Linux code makes possible the distributed character of the Linux development process, the inclusion of both profit and non-profit developers, as well as proprietary and open source code. In this sense, Linux code structure organizes relationships between programmers as well as expresses these relationships as possible and even normative. Further, the code structure itself can be seen as a response to organizational pressures within the Linux development collective. And finally, as was the case for both the source code and the behavior of the program, the structure of its code can also be used for strategic effect – as the Microsoft anti-trust case demonstrates.

**Conclusion**

Source code, program behaviors, and code structures all express different relationships via their functionality. In addition, these elements are used in more traditionally discursive forms such as speeches and writings to express value judgments about these relationships. Therefore, software is in a sense doubly expressive; first by authorizing specific relationships between programmers, users, the code, and tasks; second by providing examples that can be used to articulate, defend, or characterize forms of these relationships. Another way to state this is that software as code, behavior, and structure both *expresses* relationships as well as *organizes* those relationships. It is this doubled nature that makes embedded technical expression so difficult to pin down and analyze. The legal struggles around how to regulate software articulated above are indicative of such problems. But however useful examining such struggles can be in pointing up the equivocal nature of software, the mode of legal scholarship should not be the mode for
19

scholars engaged in the study of software and society. For while legally it may be useful\textsuperscript{15} to
create an analytic method that characterizes specific software examples as either speech – and
therefore protected expression, or function – and therefore not, this analytic separation locks off
and forgets one of the most important ways software and indeed all technologies are socially
meaningful. This is, specifically, in the way embedded technical expression – the weave between
expression and function – is leveraged in order to make and defend social claims.

For those scholars engaged in the social analysis of software, understanding this weave
has never been more important. While good work is being done, far too often the rhetorical or
semiotic aspects of software programs are addressed in absentia of a deep analysis of the
material nature of code. Latour has put this issue beautifully:

Unfortunately, there are many more literary critiques than there are techno-logists
and the subtle beauties of techno-social imbroglios escape the attention of the
literate public. One of the reasons for this lack of concern may be the peculiar
nature of the shifting-out that generates machines and devices. Instead of sending
the listener of a story into another world, the technical shifting-out inscribe the
words into another matter. (Latour, 1992, pg. 257, italics mine.)

It is important to note that engineering disciplines with the knowledge necessary to do
sophisticated technical analyses often have very different goals than social researchers. While it
is possible to hope that technical fields like computer science will develop more sophisticated
reflections on the relation between their efforts and social life, such hopes should not prevent
those of us engaged in social theorizing about software from deepening our understanding of the
technical features and contingencies of code. My hope is that the work described above can
provide one starting point\textsuperscript{16} for a more materialist position that still values the symbolic and
constructed nature of software. In addition, social researchers continue to address software that is
understood as “overtly” political (e.g. Digital Rights Management systems), but what about the myriad of devices such as toasters, automobiles, and cell phones that contain embedded code? Given that western society is increasingly dependent on complex socio-technical systems in which software plays a crucial role, understanding how and where code expresses tasks, behaviors, and social organization is the first step towards making better decisions about how such systems should (and should not) be implemented.

NOTES


2 Burk is specifically critical of the outcomes of cases involving restrictions on the export of cryptographic software by the US government as well as certain provisions of the Digital Millenium Copyright Act (DMCA).

3 It is important to note that Burk understands the limitations imposed by this view. In the conclusion he states, “If patent law is unable to separate the expressive and functional aspects of computer code, then it may instead be required to somehow accommodate First Amendment interests in that code…” (Burk, Op. Cit., pg.52).


5 It is also important to note the contested nature of the use of the term Linux to describe complete distributions. Richard Stallman, for example, has claimed that it should be called
GNU/Linux due to the reliance of the Linux programming effort on software originally created by the GNU project. (http://www.gnu.org/gnu/linux-and-gnu.html) However, others have noted that Linux distributions contain code from many different programming efforts (including X11 consortium, Berkeley Software Distribution, and others) and to credit all these sources in the name of the OS would be unwieldy. (e.g. http://www.ties.org/deven/gnu-linux.html)

6 I should note here that I use the term structure in a bounded sense to refer merely to the way the various segments of source code are arranged and connected. For those interested in a more detailed and sophisticated analysis of structure (and its relationship to agency), see Anthony Gidden’s work on structuration. (Giddens, 1979, 1984.) One warning however; Giddens creates an analytic separation between what he calls discursive and allocative resources (and a related separation between discursive and practical consciousness,) that mirrors the binary between expression and function that I am trying to overcome.


8 Bernstein, 176 F.3d at 1140


10 I use the word “reward” rather than Gibson’s terms “provide” and “furnish” in order to indicate an aspect of affordance that is sometimes lost in its use in theories of society and technology. This is namely what Gibson called the “complementarity of the animal and environment.” (pg.127) “Reward” points to the idea that affordances are constructed between sensing animals and a physical environment, rather than pre-existing within the environment.
Equally, I use “reward” and its reverse “punish” to indicate that while affordances may encourage certain behaviors, they do not and often can not prevent different ones.


12 This statement has been attributed to Tom Christiansen, one of the main developers of the PERL scripting language.

13 Whether or not this is true is another story. At one point during the court case, the judge, Thomas Penfield Jackson, demonstrated how to uninstall the icon for Internet Explorer. This example again demonstrates how technology claims are used for strategic purposes.

14 A caveat is now necessary here. The newest Macintosh operating system as of this writing is Mac OS X. This program is based upon a version of BSD Unix and thus shares the overall organizational features of Unix and Linux. One interesting aspect of Apple's use of BSD was their ability to leave most of the core OS alone and concentrate on features of particular importance to a perceived Macintosh market. This includes a new graphic display and windowing system called 'Aqua'.

15 For why such an analytic may be useful, see Lawrence Lessig, "Foreword.", Stanford Law Review v52, n5 (May, 2000):987

References


Manovich, L. *The Language of New Media*. Cambridge, MA: MIT Press


