THE PROBIT PLAN

Duke University CPS 182 Authors:

Phillip Brooks
Michael Connolly
Diana Huang
Meredith Miller
Lauren Wolkstein

TABLE OF CONTENTS
Introduction ...................................................2
  Background ........................................3
  Patent System and Protection of IP.....4
     (brief history)
Why A Different System is needed.........11
Why Software is different .....................11
  Copyright.............................................12
Patents....................................................14
Examining the Software Industry..........17
  Our Proposal.................................22
  Technical Implementation...............29
  Conclusion.....39

Appendix A........41
  Bibliography.............42
THE PROBIT PLAN

*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;*

- Article I, Section 8, Clause 8 – United States Constitution

**Probits Introduction**

In this age where nearly everything we write is typed, and most of our communications with friends, family, and professors are done through chat programs and email clients, entangling ourselves with technology, it grows increasingly more difficult to imagine a life without computers and the software that control them. Regardless of whether or not the common person uses software, the businesses, governments, and organizations that they interact with do, and thus allow code to shape their everyday lives. For this reason, it is imperative that we support progress in software development. Innovation in software has the potential to change the way we live. In order to promote innovation, developers must be aware of what is already available, and ensure that what they create will be profitable to them. The current systems of software protection fail to adequately fulfill these needs, opening the gates for a new form of software protection.

First off, the current system of copyrights and patents that is being used to protect software was not created with software in mind. Second, because of the difference between software and other traditional material, this system is ill suited to today's needs. Third, an understanding of the modern software industry’s dynamic nature helps to further reveal why the antiquated copyright and patent system is ineffective. Based on these three points, we propose a system of *sui generis* protection for software that aims to better meet the interests of all those involved in software development and use. We propose *Probits*, a market-orientated system of protection that facilitates rapid
Overview of Software Protection – Background

Currently, software is protected by a system that includes both copyrights and patents. Copyrights are expressly used for protecting a specific piece of code in a program while patents are used for protecting chips and underlying algorithms that are designed and invented rather than written and authored like software programs. Patents date back to the fourteenth century as protection for inventions, and they were designed solely for protecting intellectual property and inventions in order to foster innovation. Copyrights have dated back to 15th century England. However, the Constitution was the first time that the United States explicitly stated the purpose of copyrights and patents as “promoting 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” (U.S. Constitution, Article 1, Section 1, Clause 8). With the recent development and rapid advancement of software, there have been numerous tussles over the protection of software. In this next section we will explain the historical implications involved with this tussle.

Patent System and Protection of Intellectual Property

Patents have become the protection of choice by many programmers to effectively protect their original software. Patents protect the creation of inventive concepts as well as relation to practice, whereas copyright protects original, tangible work. A patent owner has the right to keep others from making, selling, or using patented invention for
twenty years. Patents are met by having a novel, useful, and non-obvious program to one of "ordinary skill" in the field of invention. Patents are in three main classes: utility, design, and plant. Utility patents fall into one of these categories: machine, article of manufacture, process, or composition of matter. Software has recently been described as a collection of processes that enable the protection of inventive concepts behind original programs. (Gassaway 38).

When it comes to patents, software is different than any other patentable materials. The main difference is that software has not been patented for very long in history. Patents protect ideas rather than the expression of ideas, and one must apply to the patent office for patents in a sometimes costly and lengthy endeavor. Patents are set to expire after every seventeen years. In the 1970s, Australian and European rulings excluded software from patents because mathematical formulas were not covered. It was not until 1983 that the United States Patent Office granted the first patent on software. Extending through the late 1980s, the scope of patents quickly widened, and by the early 1990s, the patent office started awarding "silly patents." An example of a "silly patent" is when Hayes patents switching a modem from transmit to receive. Patent law quickly became known as a "mess" to some people, and patents were thought by many to stifle creativity in the software industry. Many people became worried in the 1990s that the Patent office would not be able to distinguish novelty in software and that the patent examiners lacked the necessary expertise to grant patents for programmers on their software and algorithms.

**Definition of Patent Protection**

The Patent Act of 1952 established four statutory classes for patents: process, machine, manufacture, and composition of matter. These four classes are implemented in
order to limit scope of patents to applied technology (Gassaway 9). Patents on software have two different statutory classes: process claims and machine claims. A patent grants an inventor seventeen years monopoly over reproduction or sale of physical embodiment of invention and use of any novel processes or techniques which functions substantially equal to those covered by patents.

**History of Software Program Patents**

Patents originated in England with the Statue of Monopolies under King James I of England. Before patents were available, the King could issue letter patents so that any person could have a monopoly to produce goods or provide particular services. This system was being abused, so Parliament restricted the patent system through the Statute of Monopolies so that the King could only issue patents to the inventors of original inventions for a fixed number of years. The Statute of Monopolies was later developed by the courts to produce modern patent law. About two hundred years ago, the United States Congress set up the first patent board that allowed organization for granting patents. The right for a United States citizen to obtain a patent for inventions comes from the Constitution.

When reviewing software patents, it is clear that the Patent and Trademark Office has not applied the patent law uniformly to all programs (Gassaway 39). In 1965, the President appointed a commission to study the patent system in place at the time. The Commission recommended denying patentability for computer programs either as apparatus or processes. The Commission stated that it formed its opinion because of the Patent and Trademark Office’s “inability to search for prior art in programming, the lack of a satisfactory system for classifying program patents and a fear of greatly increased volume of applications (Gassaway 39). In 1967, the Patent and Trademark Office
proposed a bill in Congress in order to reform the patent system in place at the time. The bill explicitly denied patent protection for computer programs; however, after many objections to the bill, the bill died in the committee.

The debate about the nonpatentability of software has origin in case law. Early Court of Custom and Patent Appeals suggested that software was judged patentable by the same standards as any other technology. The Patent and Trademark Office (PTO) originated the theory that software did not fit into the category of patentable technology that was written in the Constitution and defined in the scope of patent protection. In the early 1970s, the PTO anticipated software applications at a time when they did not have the resources to hire skilled software examiners for patents (Maier 2). The PTO started a fight against software patentability that was against the Custom and Patent Appeals that led to some very relevant Supreme Court decisions.

**Current State of Patents**

Presently in the year 2002, the patenting of software and business methods is very controversial. The United States publicly allows software and business method patents; however, the patent system is flawed with poor information flows, high transaction risks, and inefficiency. Nearly all United States patents are initially rejected due to being too obvious. If the standard is set too high for obviousness then nothing is patentable; however, if you set too low standards then trivial inventions receive patents that they probably do not need. A famous example of this is the "One Click Patent" for software. This software patent is for buying products in on-line stores with one mouse click.

**Copyright System and Protection of Intellectual Property**
The main question that many computer programmers and software developers have about current copyright laws is if it really applies to the software that is created. There are three special cases in copyrighting software. First, there are many ways to explain and express the same algorithm so there is a conflict of expression versus idea in which copyright law is not suitable for determining the difference between the two. Second, copyright also restricts reproduction rather than use, so the main conflict with that is if executing a program should count as a use or reproduction of software. The last major special issue in copyrighting software is that independent creation in artistic work is rare so copyright violations are easy to spot; however, it is more common to notice violations in software because the concept of independent creation is more difficult to identify in software code. Originally copyright was developed for literary, artistic, and musical works, and the law has been extended to cover software. The copyright law is only reviewed and extended after they are argued in court cases.

There have many been many court cases involving copyright law and software in the last two decades. In 1964, software programs started to be copyrighted, and then in the 1980s, copyright was extended to cover object code, logic, and sequence of programs. The United States court ruled in 1982 that the basic structure of an algorithm is not copyrightable. The first major court case took place in 1986 when the first "look and feel" suit happened in which copyright finally extended to similar program/user interfaces.

The courts still have yet to really define what it is meant by "look and feel." In the late 1980s, Lotus 1-2-3 sued two lookalike products and won the case. The head of VisiCalc went to Lotus and sued them for looking a lot like VisiCalc. However, the court found that Lotus was different enough from VisiCalc. In the late 1980s and early 1990s,
Apple sued Microsoft due to similarities between Macintosh interfaces and Windows. Apple initially won but that decision was quickly overturned. Xerox then sued Apple since Xerox developed a Macintosh-like interface prior to Apple's development.

Unfortunately, the interpretation of copyright law has not reached a point of consistency. Under copyright law, software is seen as "literary work.” The courts view non-literal copying of software as infringement as alongside copying by verbatim. If two works in comparison are not "substantially similar" in embodiment, then there is no infringement. The lack of technical expertise in the legal community (such as judges not understand computer languages and code) and a lack of clear definitions as they pertain to software, hinders the software community.

**History of Copyrights (Timeline)**

Copyright covers the expression of a work fixed in a tangible medium, and the current protection with copyright lasts the life of the author plus seventy years. The historical foundation of copyright lies in the Statute of Anne of 1710. The Statute of Anne addressed the concerns of English booksellers and printers. The statute prevented a monopoly on the part of the booksellers and created a public domain for literature by limiting terms of copyright and ensuring that once a work was purchased the copyright owner no longer had control over its use (*Copyright Timeline* 1). Then, in 1787, the United States established law for copyrights in the Constitution under Article I, Section 8, Clause 8.

The 1976 Copyright Act established a system of federal attorney protection for all works that were deemed eligible. Copyright becomes the property of the author as soon as the work is created and lasts for the author’s life plus fifty years (McCarthy 4). This new act stretched the term of copyrights existing on January 1, 1978, so they would last
for about seventy-five years from their publication. In 1983, the court case of Apple v. Franklin established copyright protection for operating systems. In 1985, the court case of Whelan v. Johnson Dental Laboratory held that copyright applies even if there is no direct copying. Then in 1986, Broderbund v. Unison World was a case that established protection for audiovisual displays. 1990’s Lotus v. Paperback Software case established the look and feel protection. Another extremely important event in copyright history is the establishment of the World Intellectual Property Organization (W.I.P.O.) in 1996. Delegates from 160 countries considered two treaties on international intellectual property law during a Diplomatic Conference that was held in December 1996 in Geneva, Switzerland. These treaties resulted in a new approach to copyright laws. The Conference adopted treaties that “permitted an application of fair use in the digital environment.” This treaty emphasized the need of copyright laws to “maintain a balance between rights of authors and larger public interest, such as education and research as access to information. (Copyright Timeline 10).

**Specific Cases in History**

The Copyright Act of 1976 did not cover software copyrights, but a revision of the act in 1980 covered software. Congress recognized the need for protection of software so they established a special National Commission on New Technological Uses of Copyrighted Works (CONTU), which led to the 1980 changes to copyright law (Galler 29). In 1978, CONTU thought it unlikely that patents would become an issue for protecting software. On three different occasions the Supreme Court considered cases involving patents to software programs. Dan Bricklin, the developer of VisiCalc, found it hard to obtain a patent for the first personal computer spreadsheet. However, due to the
“landmark court case in 1981 that upheld a patent for a computer algorithm, the use of patents to protect software has increased.” (Brower 1989)

The difference between idea and expression originated in copyright law, and it is important to distinguish between copyrightable expression and patentable subject matter. Not every idea is eligible for patent protection because of the harsh conditions that need to be satisfied before an idea becomes patented. Due to the fact that patents provide powerful protection, they are very desirable by inventors as well as software programmers. A patent application is very difficult to write, so therefore, it is always necessary to prove that the invention is novel and non-obvious. Over 5,000 patents have been issued for software-related inventions between 1980 and 1990. The number of application filings is increasing rapidly since 1,500 of the 5,000 software-related patents were issued from 1989 to 1990, and since 1990, about 2,500 software patents have been granted each year (Galler 31).

In 1931, a Supreme Court ruling showed that the laws of science and nature were unpatentable. In *Gottschalk v. Benson* (Appendix A), an algorithm became known as unpatentable only because there was no particular application defined for algorithms. In 1981, the process claim for operating machinery using a computer was upheld as patentable through the *Diamond v. Diehr* (Appendix A) case. In 1989, the Patent and Trademark Office published guidelines for software patenting saying that a computer directed by a computer program is a statutory machine. A 1994 patent recently allowed computer software to be patented by a machine because a general-purpose computer becomes a special purpose computer once programmed to perform particular functions according to instructions from program software.

[see Appendix A for more details on specific court cases in patent history]
**Reasons Why A Different System is needed**

Due to the tremendous controversy involving patents and copyrights in current software implementations, it is important to look at why software is different from other mediums that are copyrighted and patented. The various court cases that are discussed need to be reviewed more closely in order to see how different software really is from papers or inventions that were the first things copyrighted and patented. The next section will express why it is imperative to have a different system of protecting our software rather than the old system of copyrights and patents just because they are the only systems that have worked in the past with protecting intellectual property.

**Why Software is different**

Software is both expression and behavior. Therefore, software embodies characteristics that are difficult for legal commentators to see (Samuelson 1). Software patents do not cover entire programs—they cover specific instructions used to build a program. Copyrights, on the other hand, do cover the program as a whole—protecting original expression. Because of software’s dual nature however, neither patents nor copyrights provide adequate protection for software. This section will further examine these issues in terms of patents and copyrights individually.

*Copyright law does not apply to software*

Copyrights apply to original expression that anyone creates and can apply to any original expression with a “mere modicum of creativity” (IP for CS Students, pg. 1).
Though programs are expressions and text, this explanation is incomplete and does not fully capture all aspects of software.

1. **Though programs represent the creativity and innovation of expression, they also represent the functionality and behavior of machines**

   Thinking about software as text fails to capture the whole essence of the program and is a largely incomplete view. A program’s main purpose is to behave. They execute the instructions given by the text of the actual program and compete in the industry based on their functionality, whereas products and items originally protected by copyrights are sold according to their merits of originality of expression. For example--while a book’s value lies in what is in the text, the bulk of value in software lies in its behavior.

   Expression and behavior are independent of each other. A programmer could be given a task of writing code that behaves identically to another piece of software, and the text of the program would never have to be viewed. Thus it becomes very easy to create market substitutes for software and still have original expression and often creates a large source of copyright controversy, as demonstrated in the lawsuits between Lotus and VP Planner and the Twin. Both smaller companies had independently developed programs in hopes of staking out their own sections of the market. Instead, however, they found themselves faced with copyright infringement lawsuits filed by Lotus (Samuelson 9).

2. **Programs are very similar to physical machines.**

   Programs share many similarities to actual machines. Samuelson notes, “Programs can just as well be physical machines; an electronic device that plugged into the computer could deliver identical behavior” (Samuelson 3). Programs can be
used to produce identical behavior of machines, and can be used to interact with other programs.

Creating a program is also very similar to building a machine.

As Samuelson writes, “Where physical machines are built from physical structures like gears, wires and screws, programs are built from information structures, such as algorithms and data structures.” These different information structures work together in a similar fashion to the gears, wires and screws of machines to produce the desired output. Hence, the text used to write programs should not be copyrighted. “The legal character of a work created in the medium of software should no more be determined by the medium in which it was created than would be a work made of steel or plastic.” (Samuelson 5)

3. **Software has many features that cannot be captured simply in the text, and cannot be copyrighted**

Software represents a number of innovative ideas, translated from ideas in real life, improvements upon ideas found in ‘real life’. These ideas fail to apply to copyright law. One example where this is clearly seen is in word processing programs where the cut/paste feature allows you to freely insert/delete text on your document without manually ‘erasing’ anything. Ideas such as these capture a more abstract character and are not covered by copyright law (Ringuette 1).

**Patents do not apply to software**

Patents do not cover specific systems, but techniques used to build specific systems. As a result, patents are often used in techniques used to build physical machinery and hardware. Several fundamental flaws can be identified and found when
applying patents to software. We will now continue to examine these flaws by focusing on how software is different than hardware, how patent law explicitly does not support software, and how patents are impractical according to the immense speed of the industry.

1. **Software is more complicated but easier to produce than hardware**

   Software systems can be built to be incredible complex. Using the previous example of the word processing program, “…you might be able to choose between a word processor with built-in spelling checker, ability to format multi-column text, and an outline editor; a word processor with proportional fonts, an equation editor, and kanji capabilities; and a word processor that has style sheets, a page previewer, and document interchange facilities. And this is only the start (Ingram 1).” As demonstrated from something like an everyday word processing program, the total number of features available is huge, and any one of those features could be patented.

   Not only can it bundle more features, software is cheaper to manufacture and easier to design. Copies can be easily made, and creating 50,000 lines of code is simply a matter of finding the time and a few available programmers. On the other hand, building a car requires different teams for the testing in various external conditions, production and evaluation.

2. **Software is more Abstract**

   Patents are usually geared toward specific products in very specific industries, thereby making the process relatively simple. A "typical patent title" could be along the lines of "Method for Increasing grain throughput in a combine harvester by means of an air-forced hopper" (Ingram and Williams 5). The complexity discussed in the
previous section makes a computer program dependent on a number of other technologies and methods. Thus, its abstractions create difficulties in partitioning these technologies. Software patents may often sound quite specific, but they may actually be quite abstract. Though Patent #5175,857, "System for Sorting Records Having Sorted Strings Each Having a Plurality of Linked Elements Each Element Next Recording Address," has a highly specific title, it actually protects a brought process of sorting that is frequently used by different members of the computer science community—a linked-list implemented well-known algorithm Quicksort. Because software’s complexity creates these abstractions, complications are created regarding classifications of the technology. As a result, there is a "combinatorial explosion of potential patent coverage which removes any kind of certainty about what is patented and what is not." (Ingram and Williams 7)

3. Software Technology Evolves Rapidly

"Think back 17 years. Graphical user interfaces were virtually unknown. Desktop publishing didn't exist. MSDOS didn't exist. Neither did PCs and Macintoshes. With microprocessors doubling in speed every 2 years, this qualitative change in the nature of software is likely to continue. Compare this rate of progress to that of other industries such as the aircraft industry. In software, 17 years is a very long time. The existence of patents 17 years ago on what might then have seemed non-obvious or esoteric technologies would be extremely damaging today. Likewise, much of what may be considered non-obvious today, will be seen as being fundamental and obvious tomorrow."

   (Ingram and Williams 8)

Another problem that is neither addressed by patents nor copyrights is the incredible speed of growth of the software industry. Copyrights last for 20 years after the copyrighter’s death and patents last for 17 years.

The software industry is evolving and developing at breakneck speed. Consequently, the time restrictions on patents and copyrights make it very likely that a software developer might violate a current patent or copyright.
Neither copyrights nor patents can fully protect software. Both were created with the intention of promoting innovation. In the case of software, because it is a combination of expression and behavior, and because the industry moves at an incredible speed, it hinders this. We now will attempt to explore a method that tries to address these issues.

2. Clashes with patent law

There is still some question about whether or not the application of patent law to software is legally questionable. The “subject matter” doctrine, excluded many processes from being patented—from business methods, to data analysis.

“Traditionally, the only kinds of processes that could be patented were those for transforming matter (such as, for transforming iron into steel)” (Epperly 8). U.S law does not allow inventors to patent laws of nature, and this was extended to computer algorithms in the 1970s (Gottschalk v. Benson, 1972, and Parker v. Flock, 1978). However, the 1981 case Diamond v. Diehr (Appendix A) indicated that computer algorithms could be used to control industrial processes. Though this case was an actual case in transforming matter (curing rubber), the Patent Office “took this narrow decision as a green light for unlimited patenting for software techniques, and even for the use of software to perform specific well-known customary activities.” (Epperly 8)

A requirement of patent law is the invention must be ‘obvious’. However, in terms of software, “the standard of obviousness developed in other fields is inappropriate for software”. Patent examiners consider small, incremental changes as deserving of new patents. However, The League of programming Freedom notes:

Computer scientists solve problems quickly because the medium of programming is tractable. They are trained to generalize solution
principles from one problem to another. One such generalization is that a procedure can be repeated or subdivided. Programmers consider this obvious – but the Patent Office did not think that it was obvious [as demonstrated in a flawed patent case].

(Epperly 6)

As we have examined, software represents a very individual medium that is not just expression, not just behavior and not as complicated as physical machinery—it is a meld of the three.

**The Dimensions of the Modern Software Industry**

It is clear that the current system of patents and copyrights is not working well with software. Our proposal aims to improve this system. However, before we reveal the technical details of our proposal, it would first be wise to provide an overview of the software industry by considering the goals, needs, and wants of the industry and its customers. By considering the tussles involved, we will hopefully be able to find a more promising solution.

“The Software Industry” is a phrase that gets tossed around a lot. This industry has been discussed at length in this class, but it is likely that many people (and perhaps even some computer science students) still do not fully grasp the multidimensional nature or the true scope of the industry. This section will provide an overview of the software industry in order to reveal its complex relationship with its consumers. Furthermore, this section will reveal the software industry as being subject to both rapid change and tremendous growth. And most importantly, it will show that the industry operates on various levels in many fields. In short, the industry is tremendously diverse, and this helps to explain why the rigid institutions of copyright and patent have failed.
The software industry has evolved with the development and progress of computer hardware since the 1960’s. (Rudy 16). According the Bureau of Labor Statistics, the United States software industry employed 336,200 people in February of 2001. (Rudy 10) The industry enjoyed tremendous expansion during the growth of the personal computer market and the explosion of the Internet. However, by January of this year, the recession of 2001 was reflected in the record-high unemployment level of information technology workers—at 6%. (Rudy 15) Still, the general sense is that the long-term forecast for both the industry and its workers is strong. (Rudy 11,17) Today, there are many businesses who made giant investments in computer hardware during the boom of the late 1990’s; now they are looking for cost-effective software solutions to maximize the return on those investments. (Rudy 1) And in the long run, the potential for expansion into parts of the world lagging in modern technology seems unlimited.

While it is clear Microsoft perennially dominates the PC software field, the industry itself is increasingly diversifying into an vertical market where specifically focused solutions are capable of finding their own niche. (Samuelson Sec 4 pg1) The video game industry is one such example where smaller firms have managed to enjoy a great deal of success. The industry is one of the fastest growing in all of software, and in addition to giants such as Microsoft, Sega, Sony, and Nintendo, it also sustains the smaller, independent firms, such as Electronic Arts, Activision, Take Two Software, and Midway Games. (Rudy 2)

While entertainment software has offered an opportunity for a wide range of firms, the expanding market for software relating to non-PC internet connectivity is being dominated by Microsoft and Palm. Today, the internet is typically reached when a user connects via a personal computer. However, in the near-future, it is widely expected that
connectivity will occur through other non-PC devices, especially on wireless personal assistants where Microsoft and Palm dominate. (Rudy 3) A second key area of software dedicated to internet relations is that which deals with security. Firewalls and antiviral tools seem more important than ever following the attacks September 11th, 2001.

According to many analysts, the next “big thing” in the software world may very well be web services. Web services are based on open systems and interoperability, and they represent one of the key areas of the growing enterprise software market (Rudy). Many business have invested a great deal of money in systems that don’t work well together; web services provide the solution by creating infrastructures that allow previously incompatible software components to interact with each other. This kind of software seems even more vital in these hard economic times. For example, a business that is on a tight budget will likely opt to find a software solution which connects two incompatible systems before making the investment to replace the systems.

Aside from web services, there exist three other major sectors of the enterprise software market: Enterprise resource planning, customer relationship management, and supply-chain management software. Enterprise Resource Planning software automates operations relating to “back-office” responsibilities. (Rudy 6) Major players include Oracle, PeopleSoft, and J.D. Edwards & Co. Customer Relationship Management software helps companies attract business by bolstering their sales teams and customer service departments. (Rudy 7) And Supply-chain management software controls the flow of products along the supply chain, starting with raw-materials and ending on “the showroom floor.” (Rudy 7) Finally, another important area of the software market can be found in “middleware,” which is used to help develop and launch enterprise
applications by overcoming the differences between complex operating systems and individual programs. (Rudy 14)

Having outlined the detailed interests found in the software industry, it is now important to consider a few of key aspects to the software business. There are three sources of revenue for software developers: license revenue, maintenance revenue, and service revenue. License revenue comes from making money off the sales of individual software copies. Maintenance revenue is derived by charging fees for updates or enhancements, and service revenue comes from supporting customers in the use of the software.

In addition to various sources of income, another important element of the software industry is the product cycle. According to Standard and Poor’s: “Product cycle issues drive much of a company’s license sales. Completely new software yields incremental revenues; if a new product generates a huge customer response, it can produce a high volume of new sales and higher revenues” (Rudy 22). Furthermore, the needs of the consumers are more or less mirrored in the shape of the industry. For one, it is often the case the innovation is not as valuable as familiarity. A simple example: Consider an employee who is paid twenty-five dollars per hour, and who’s company purchases a new software product for 100 dollars. If it takes longer than 4 hours for that employee to install and learn the new program, then the cost of the program quickly becomes secondary to its familiarity. The value inherent in product familiarity serves as a tremendous barrier to entry in the software market. (Samuelson Sec 4 Pg 3)

Given this overview of the software industry, it is clear that the industry exists on multiple-levels. In sum, the software industry is dynamic, expanding, and above all, diverse. The wonderful diversity found in the range of software products helps to explain some copyright and patent’s failures in this field. The laws of copyright and patent are
rigid, but the industry itself is multidimensional. On the one hand, the industry can be monopolistic, as seen in examples regarding the Microsoft Corporation. On the other hand, the industry can also be a bastion of opportunity, as found in entertainment software, where smaller upstarts have found success. Given the diverse nature of the field, the simplistic and often foolish application of traditional patent and copyright law is inadequate. Therefore, in our plan we will seek to improve upon the current pitfalls by approaching the application of intellectual property protection with the interests of an innovative software environment.

**What we Propose**

Clearly, neither copyrights, nor patents are suited to the protection of software. Both were created and developed to meet the needs of other goods and intellectual property that was, relative to software, one-dimensional. Trade secrets, the alternative to the first two options, in short limit innovation, and in an industry where much of the technology that is used is evident from the surface, makes it easier to reverse engineer and thus does not provide ample protection (Ramos, pg. 72).

The current system in place is an ad-hoc system built by court case upon court case, molded by outside pressures, judge’s bias, and other factors (Woo, pg. 98). In no way were any of these methods of protection intended to be used for something as complex as software. Rather than the government understanding and responding to the interests of software developers and software users, the two parties themselves began pulling pieces of the system their way to cover themselves. For this reason, we propose a new system of protection for software, referred to as probits, one that is comprised of four pillars.
The first is that probits act on software, second, probits will only be granted for software that demonstrates it is new and significant. Third, the lifetime of a probit will vary but remain relatively short, and fourth, the ability of probit-holder’s to sell rights to their probits.

Scope of Probats

The first pillar of the probit system is that it will act on software. Software is recognized to contain both text and behavior, but rather than focus on one or the other, as the current system is trying to do, under probits both are integrated into the definition of software. Precisely because of these two aspects of software, however, one cannot simply say ‘software’ and have the meaning be clear to all.

Software, for the purposes of probits, will be defined as “a pattern that is readable and executable by a machine” (Suber). Software is a pattern per se, but that pattern can be read and used by a particular set of hardware. An algorithm, or pseudo code for instance, cannot accomplish anything on its own. It is unreadable and not executable, until it is implemented in a language that a machine is able to interpret. Therefore algorithms by themselves will not be protected by probits.

Software may include operating systems, which communicate directly with the hardware, applications that give the OS instructions, or plug-ins that give applications instructions, because in all cases, the end results are new patterns of behavior for the processor, or hardware. Firmware, or software that is written onto hardware, and code written in proprietary languages to do specific tasks, will also fall under the probit jurisdiction.

Requirements for a Probit
A probit will only be issued if the software in question demonstrates that it is “new” and that those parts that make it new are “significant.” The concrete decisions concerning what is new or significant will primarily be made by the government body that issues probits, using a technical system of comparison that will be presented further into this paper. A minimum of those decisions will be left up to the scrutiny of court should they later be challenged. General guidelines for determining newness and significance are obviously a part of the definitions that in this context these words employ, and so here we will define them.

“New” means that the software must be different from any other currently probitted software. To minimize court cases, it must also be different from any single existing piece of software that exists in the public domain. Different is measured by the way the software’s parts interact and the behavior that is produced. If two pieces of software are taken from the public domain and combined to produce something novel, then the result would be considered new.

To give an example of this, say there exists in the public domain software that can produce spreadsheets and software that allows the user to type documents, a word processor. If no one had ever combined the two before, then developing the word processor software to allow for the use of spreadsheets within documents would be considered “new” and would therefore pass the first requirement for acquiring a probit. At this point, it is necessary to consider whether that combination or novelty is significant.

“Significant” is in reference to the new aspect of the software being probitted. While the change that is made must in itself be a significant change (a large change that sets the software apart from its predecessors), that is part and parcel with the definition of ‘new.’
Here, “significant” is used in terms of meaning. The change that is made, the new aspect, must be meaningful in that it meets or addresses the needs of its targeted market or users.

There must be a reason for the change: a projected or sensed need in the environment that the software is produced for, or what the change is meant to accomplish. The software must serve an actual, useful, purpose in the targeted community. For instance, in the example above, the combination of a word processor with a spreadsheet program would allow users to display uniform data in reports or documents, and would thus be significant.

This is intended mostly to prevent frivolous probits, or mass probit applications for various combinations of public domain software. If probits were to be granted based simply on newness, it would be possible for one entity to probit a number of different public domain software combinations hoping to profit by having others purchase the rights to those combinations or to limit others from using that particular combination. Between 1990 and 1992, IBM was granted an estimated 500 patents, one of which was for software that would make the patent process easier for employees (Irlam and Williams). The patent situation somewhat mirrors the current registering of domain names, not for use, but for the sale to companies or people that are likely to want those names, also known as “cyber-squatting.” In order to curtail cyber-squatting, the “Anticybersquatting Consumer Protection Act” was passed in 1999, which held a person who registered a domain name liable if they had “a bad faith intent to profit from the mark” (Eisenberg 3). Similarly, intent would be a factor in the significance of the software that is to be probitted.

**Lifetime of a Probit**
In a manifesto published regarding the protection of software, a proposal was made for a market-based legal regime (Samuelson 1). In contrast to traditional intellectual property rights, which are not concerned so much with the sale of ideas as they are at maintaining exclusiveness, a market-based system would only attempt to correct market failures. When a software product does not have a chance to be a success because of the circumstances, and in turn does not do well, it is considered a market failure (Samuelson 4). The best way to prevent such market failures is by adjusting the amount of time that companies before their ideas become available to others.

As probits are intended to be a market-orientated legal regime, the lifetime of a probit will be long enough only to prevent market failure for the software that is probitted (Samuelson 3). In other words, a probit’s lifetime will give the owner of the probit sufficient time to market his or her innovation and to continue development on the product so that a new probit can be issued by the time the original probit expires. The lifetimes recognize the rapid pace at which software can and for the most part is developed in certain industries. The current extensions that are being applied to software protection leave little that is useful for the public by the time that the extensions are over.

As consumers well know, the protection provided to the content industry has simply increased over time and there is no guarantee that that protection will not continue to increase, preventing forever the entry into the public domain of works. Consumers have become increasingly restive about the absence of any legislative guarantee that anything of value will be returned to society as reimbursement for the monopoly rights they have ceded. In relation to computer software in particular, by the time that a work is released from monopoly control all value has been completely sucked from it, leaving only a dry husk of some historical, but little cultural or social utility. (Scott 6)

The lifetime will vary between industries and the targeted markets in order to return valuable ideas and code to the public and facilitate innovation. The lifetime would be
calculated by a formula consisting of the following three things. The average time it takes for companies in a certain area of software development to release a new version, the amount of expenses it takes to develop software for that area, and where the revenue resulting from that piece of software comes from.

In 1999, personal computer software generated $12,940 million, with on average 20% paying for the development of the software. Software developed for enterprises, however, grossed $22,849 million (Plunkett). The obvious reason why software that is developed for enterprises takes in more revenue than software for consumers is that enterprises have the money to spend, and spend it in bulk. In general, large companies will be more hesitant to buy several thousand copies of a piece of software without guarantees and a solid reputation of the product they are looking at. In contrast, consumers have the ability to purchase software on a whim. Therefore, the life of a probit would for consumer software would be shorter than the life of a probit for certain business software. It would take longer for a software developing company to market their product to a business than it would to sell software to the average consumer.

After a probit expires, the software becomes a part of the public domain to be referenced by and used by anyone. Once a piece of software falls into this domain, it can be counted as prior art, meaning a probit cannot be issued on that application alone, but can receive probits on combinations or new designs that use that code.

Rights of a Probit Holder

Holders of probits will be granted certain rights for the use of their probits. The first is the ability to sell the rights of a probit. The rights to a piece of software that is probitted, hereby referred to as the parent, can be sold to others, becoming the child,
allowing the purchaser to develop the code. If the child proves to be ‘new,’ relative to parent, and ‘significant,’ the purchaser can in turn probit that piece of software. The child can be probitted, if and only if, it demonstrates that there has been a significant change to the code that was purchased. In filing for a probit, the purchaser will also have to show proof that the parent’s rights were purchased.

The parent will by law retain the right to sell its code, and the child will, once obtaining a probit, gain the right to sell its newly adapted version of the code. This means that if there is a probit on a piece of software that has a word processor with built-in spreadsheet capabilities, the rights to the use of that probit can be sold to other developers. If a deal is brokered, and another company buys those rights, that company, now the child, cannot probit the word processor as is. It must develop the code, and either add new features to the code, for instance like adding graphics capabilities, or make significant changes on the code to tailor it to be sold in a different market. Once the child attains a probit, it gains the rights of any probit holder, meaning it will have just as much right to sell the rights to its version of the code as its parent did.

The parent will not have any say in who its child sells rights to. This is unlike patent law in that patents centralize the power over ideas: once a parent sells the rights to a probit, the control of the ideas protected by that probit are dispersed. Under probit law, however, it would not be possible, for instance, for anyone to purchase the child’s rights and simply remove the graphics capabilities, and receive a new probit issued for a word-processor with built in spreadsheet tools. Also, until the parent’s code enters the public domain, no-one will be able to probit software that incorporates a spreadsheet feature in a word processor, unless rights are purchased. If the holder of a probit suspects that someone else is trying to probit a piece of software that is too similar to their own, legal
recourse will be available, along with the public scrutiny period for probits, which will be detailed later in this paper.

**Technical Implementation**

The probit system relies heavily upon a fast, efficient, dynamic infrastructure in order to achieve our society’s goals of the advancement of technology and the promotion of competition. The system has been designed to bring the protection of intellectual property up to speed with the pace of the software industry, avoid overwhelming the court system, aid in communication, help the advancement of technology, and promote competition by preventing monopoly. Such lofty goals require a well-oiled machine to back them up, and the probit system aims to utilize the tools of the field -- cutting edge software and the Internet -- to build an online, centralized site for registration, verification, advertisement, and sale of probits at www.probit.gov. A technically-savvy new branch of the U.S. Patent and Trademark Office, the Probit Department, will oversee this online forum, and provide a research and development team capable of continuously improving the necessary software and updating time limits based on the current pace of the industry.

**Usership**

The first time a user visits probit.gov, he must register a user name and password to use for all transactions that will take place on the site. All registration, buying, and selling of probits will occur exclusively at probit.gov. Any individual interested in registering, selling the rights to, or buying the rights to probits may register as a user. By maintaining a database of users, personal identification information, and a list of each user’s probit transactions for the past two years, probit.gov can keep track of which users currently hold or have bought the rights to which probits. This information becomes
crucial when a user tries to probit a new piece of software that has built upon another currently probitted piece of software. This way, probit.gov can verify that the applicant has in fact legally bought the rights to the previous software. Also, probit.gov can keep track of probit expiration dates and other crucial information necessary to verify probit transactions.

Probit.gov will also maintain a rating system by which users can build a good reputation for writing substantial software. A user starts out with a moderate user rating that naturally increases over time. Failed attempts at probit applications decrease a user’s standing, while successful sales of probit rights increase a user’s rating. This prevents users from making frivolous submissions, while allowing talented developers to build artistic integrity. Software programmers can take pride in their work, and become well-known in the programming community for their high ratings and solid authorship. Our American society “would like to reward enterprise and creativity, allow free and open access to ideas, and benefit form a rich trove of music, literature, journalism, and art”(Vaidhyanathan, p.2). Rewarding “enterprise and creativity” in American society necessitates stroking egos and allowing artists and programmers to claim authorship of their work and build good reputations in their field.

Cost

Currently, the majority of software patent application, registration, and licensing is expensive. Smaller companies “can’t afford the cost of patent searches or litigation”(Garfinkel et al., p. 9), especially since patent prosecution costs now range from $250 for a US prior art search to $700 for a prior patent and literature search, and litigation costs hover around $1600 for an infringement defense (Aharonian, p.1).
Applying for a patent is a financial risk, not to mention that the application process itself costs thousands of dollars. These inflated costs are passed down the system to those who license software as well. In fact, some firms “would rather pay the costs of reverse engineering than a license fee (Samuelson et al., Section 5).

The high costs associated with applying for a patent serve two purposes. First, the cheaper the application process, the more applicants will apply. If we want to lower the cost of applying for a probit to essentially zero, in order to promote smaller companies and individual developers to probit their software and compete with larger companies, then we must construct another method by which to weed out frivolous applications. Otherwise, the system becomes clogged with long-shot attempts at fooling the system. Secondly, patent applications serve as a source of revenue to help relieve the financial burden created by the Patent Office. Prior art searches require time and money. For the Probit Department, time is precious and costs should be minimized. In general, a software protection system based on our competitive marketplace should minimize barriers to entry, including financial barriers (Samuelson et al., Section 6) so as to open up the industry to players with smaller start up capital.

One solution to this problem is to utilize current software technology in order to automate as much of the application and registration process as possible. Ideally, a sequence of automated prior art searches would cut down the number of acceptable applications significantly, allowing the removal of financial investment as a necessary barrier to entry. Not only would this cut down on costs, but also it would cut down on time. Now it takes two to five years to file for and obtain a patent (Garfinkel et al., p. 3). Using an online system with an automated prior art search would cut down on this delay enormously.
Public Domain

In the probit system, any software not currently under probit belongs in the public domain, and is free for anyone to use or distribute as he or she pleases. In order for an automated prior art search to be effective, there must exist a database or repository for both currently probitted software, and software that has moved on into the public domain. Similar databases now exist for use by patent prior art searches, such as ip.com’s Prior Art Database (ip.com, p.1), and the Software Patent Institute Database of Software Technologies, which is a database of historical records such as manuals and theses rather than actual source code or patent documents (SPI, p.1). Both databases necessary for the probit system would consist of probit applications, the details of which will be explained in the next section.

The database of currently probitted software will be, by the nature of its two-year-or-so turnover rate, of a manageable size. The database of software in the public domain, on the other hand, is potentially infinitely large, not to mention the fact that to implement this system would require creating probit applications for all software designed previous to this point. Our solution to the issue of retroactively creating such a database will be explained in detail later in the paper. For now, a comparison between the size of the US Patent Database versus the necessities of the probit system, to determine the government’s available resources, will suffice. The USPTO currently maintains a database of the full text of all US patents before 1976, and full images of all US patents since 1790. This database only includes the physical patent documents themselves, similar to the idea of a database of only probit applications, although many probit applications will exceed patent documents in length due to the inclusion of source code or
source code representations. Furthermore, the public domain database will only include software patents, whereas the USPTO database includes all patented inventions since 1790 (USPTO). Finally, “today’s computer programs are so complex that they contain literally thousands of algorithms and techniques, each considered patentable by the Patent Office’s standards” (Garfinkel et al., p. 3). Unlike patents, probits protect the “newness” in an entire program, so for each piece of software, one may apply for one probit rather than multiple patents, decreasing the size of the necessary database significantly.

Application

The probit application will consist of 4 parts:

1. A description of the problem that the new and significant piece of software solves, and the general category most closely relating to this problem

2. Source code (full source code when submitted and while in probitted database, but source code tokens when in the public domain database.
   The concept of a token will be explained later in the paper.)

3. Canonical binary representations of any Boolean functions that are new and significant

4. A description of the solution written by programmers for programmers to understand

   The general category in the first part of the probit application is the criteria by which probit applications will be sorted in both databases. A list of general categories to choose from will constantly be updated with specific subcategories, to avoid misplacement of applications, which could lead to overlooking applicable prior art.
Source Code Comparison

One way to determine if two pieces of software are similar enough for the prior to be considered prior art, is through comparing source code. The difficulty here lies in the vast array of computer languages and styles of programming that these sources may be written in, not to mention that “different parts of computer science frequently reinvent the same algorithm to serve different purposes” (Garfinkel et al., p.4), using different variable names, and metaphorical structures.

YAP is one of a handful of programs developed to prevent plagiarism in computer science classes. The program is a structure-metric system rather than a counting-metric system. Counting-metric systems count the number of attributes that two programs have in common, such as the number of unique operators, the number of unique operands, the number of loops, the number of procedures, and even counts reflecting flow control structures. Some of these variables are very simple to change for the sake of appearing different, without actually changing the behavior of the program at all. Structure-metric systems, however, are far more involved, and can accurately detect plagiarism in advanced computer science classes, rather than just Intro courses like the counting-metric solutions. Although the attribute-counting-metric systems perform better than the structure-metric when the two programs being compared are very close copies, our application of this technology to probit prior art searches requires the ability to detect accidental similarities, partial similarities, and expert plagiarism or reverse engineering (Verco & Wise, pp. 1-5). In software development, independent reinvention of very similar technologies is common (League For Programming Freedom, p. 7), so programs that may be too closely related to deserve their own probits may look very different.
YAP3, the most recent version of YAP, represents a program by a long string called a token sequence. The token sequence of a program is created by

- removing comments and string-constants
- translating upper-case letters to lower case
- mapping of synonyms to a common form, e.g. `strncmp` is mapped to `strcmp`, and function is mapped to `procedure`
- reordering the function into their calling order. In the process, the first call to each function is expanded to its full token sequence; Subsequent calls are replaced by the token `FUN…`
- Removing all tokens that are not from the lexicon of the target language, i.e. any token that is not a reserved word, built-in function, etc.

(Wise, pp.1-2)

The token sequences of two programs are then compared for similarity. In order to modify this technology to apply it to probit prior art searches, the Probit Department research and development team would first have to analyze and program synonyms that occur by accident, rather than obvious plagiarism synonyms such as turning a while loop into a for loop. Also, the Probit Department would have to either use the synonym translation step to also translate languages, require all source code to be submitted in one computer language, or devise some other plan to compare across languages. Finally, the Probit Department would have to run a study to determine what similarity score a program would need to pass this part of the application process.

Although this explanation takes a close look at the process, these methods can also work on a larger scale through software such as SMAT, a Software similarity MeAsurement Tool developed to compare operating systems in order to follow their
evolution (Yamamoto et al., pp.1-3), and JPlag (Prechelt, p.1), a web service that compares two large sets of programs to each other, which is similar to what the Probit Department would also have to implement.

**Canonical Binary Representations**

New and significant algorithms or functions that translate well into Boolean expressions may also be expressed as canonical binary representations for the purpose of comparison. Ordered Binary Decision Diagrams (OBDDs) represent Boolean functions as compact, canonical graphs that can be generated and manipulated by efficient graph algorithms (Bryant, p.1). OBDDs consist of a series of nodes and connections: nonterminal nodes labeled with variables, and terminal nodes labeled with 1s or 0s. The lines connecting the nodes are either dashed (when the decision variable is 0), or solid (when the decision variable is 1). What makes these graphs excellent for comparing functions is the property that OBDDs follow reduction rules such that any graph can be reduced to a unique representation with a particular ordering of variables. Simple graph algorithms can operate on these OBDDs to determine whether the two functions are similar. This test should be used as a supplement to the source code comparison since not all algorithms are equally amenable to this representation, and not all programmers would know how to represent a function in this manner.

In fact, neither of these methods even nears perfection, and they are not meant to act as the backbone of our prior art search. Instead, their job is to expedite the application process and reduce the number of applications that reach higher levels of necessity for decision-making power. The fourth section of the application, the written explanation of the programmer’s solution, will mostly be utilized when it is an innovative
combination of already discovered algorithms, or a breakthrough cognitive representation for the users of a specific program. These sections of the application may be compared by simple counting and comparing of words. The explanations will be brief, and most of these applications will need specific attention anyways. If a probit application “passes” these three tests (or as many of these three tests as are applicable depending upon the application), it can then move on to the final and more challenging prior art search.

Verification

Each probit application will next have to stand the scrutiny of peers for a period of 2 weeks. During those two weeks, the problem posed in section one of the application will be posted publicly, and registered users of probit.gov can bring forth prior art or new solutions in an open forum-style challenge of the program. All other parts of the application will be kept confidential. If prior art is brought forth that proves the application not “new,” then the application will be discarded, and a retroactive application will be created for the prior art brought forward to add to the public domain database. This is also the method by which the public domain database will originally be constructed retroactively. This way, the system does not waste valuable resources keeping track of and searching through old programs in which nobody is interested.

If, however, a user sees the posted problem and invents the same solution within those two weeks, then the program has been proven not “significant” in that it is obvious enough for someone else to reinvent independently. Again, this application is added to the public domain database to prevent similar applications from making it to the challenge period in the future.
Probit.gov would also include a search engine for the public domain database so that potential probit applicants may search for prior art before applying. Each of these measures decreases the number of applications at every step along the probit process, and prevents the need to use the legal system as regularly as it is used for patents, where prior art searches are less reliable than peer-lead challenges. Furthermore, by placing less emphasis on descriptions worded by the programmer, more emphasis on source code and less disputable representations, and by putting applications up to public scrutiny, applicants have incentive to write clear, coherent, straight-forward applications, rather than try to confuse the reader, as sometimes can happen in patent applications.

**Advertising and Sale**

The final element of probit.gov is a common forum for advertising, purchase and sale of probits. By having one, centralized, open marketplace for software and software rights, software natural selection may take place (Samuelson, Section 6). The best method for doing something then may become a standard, and none of these new standards will be ruled by monopoly power. This will increase compatibility and general technological advancement, as well as fair competition based on quality rather than scare tactics. Pricing by demand will bring down software prices, but sales will increase due to faster technological advancements and lesser incentive to pirate software when you can buy it at a reasonable price. Large companies who lose power through the implementation of this new system will continue to be able to compete, and can expand their businesses through bundling, advertising, and publishing probitted software. Those players not interested in participating in the new system can always rely on trade secrets
for protection in lieu of government aid, but the centralization of the software market through this online forum will make the option of trade secret less and less appealing.

**Conclusion**

In conclusion, we feel Probits will be a refreshing change because they provide a system of protection for software that does not inhibit technological progress. Furthermore, by placing the probit implementation online in a centralized Internet forum, the system retains the ability to change quickly along with an ever changing industry.
Appendix A

*Gottschalk v. Benson, United States Supreme Court 1972*

The Supreme Court found that software/algorihm is an idea and not patentable. The algorithm in question for this case was used for converting decimal numbers into binary-coded numbers and then converting binary-coded numbers into pure binary numbers. This algorithm produces a shorter and more computationally efficient string of binary numbers in a computer program's object code. The real problem with the patent was that the description and claims of the algorithm did not have utility and lacked any particular useful application. However, if the patent were drafted by an experienced software patent attorney, the result today would be clearly different from the result of the case in 1972.

*Parker v. Flook, United States Supreme Court 1978*

The United States Supreme Court ruled that unpatentable software did not become patentable by the addition of “conventional, post-solution applications” of the software. The case dealt with a computer program to calculate an alarm system used with a catalytic converter system. It was a mathematical formula that read sensor data and recalculated new alarm limits. The court stated that “where A, B, C, are non-level and all that is added is D, a mathematical algorithm, it is not patentable subject matter (Tufte 2).”
Due to the result of Benson and Flook, it can be concluded that the court seems to err on the side of nonpatentability in new technology areas.

**Diamond v. Diehr, United States Supreme Court 1981**

This case established the patentability of software in certain cases. The views of the Supreme Court have greatly shifted since the early 1970s because the general view of patents underwent a shift from "monopolies" to "incentives to invent." This particular case decided that an electro-mechanical device with a computer has an integral part. This device was specifically used for curing molded rubber and shutting down at proper time. The computer program incorporated an old formula called the Arrhenius equation that calculated the correct curing time. The Court said that this was held as patentable subject matter because an otherwise patentable invention does not become unpatentable just because it incorporates a computer program in it. This was the first time that the Court was really changing directions towards patenting software-related inventions because software was being used to apply the Arrhenius equation automatically was the core of the invention. Since this *Diamond v. Diehr* case in 1981, the Supreme Court has not decided any other software patent cases. This court case opened up patent protection to the entire field of software algorithms to the standard criteria of patentability, such that the “process be novel, useful, and not obvious to someone skilled in the art (Galler 30).”

**Software Patents Issued in Early 1980s**

The Patent and Trademark Office issued a lot of software patents beginning in early 1980s, and then throughout the 1980s and most of the 1990s, the Court of Appeals for the Federal Circuit struggled with how they should deal with patents. Many decisions by the Court of Appeals were inconsistent, and their general theme was that a software invention had to show some kind of physical transformation. This has led software
patent attorneys to draft the description and claims as if software had some kind of physical embodiment. Two examples of a physical embodiment are describing and claiming the invention as changing the storage medium and as changing a general-purpose computer to a special purpose computer. Software patents are still written like this today, often using the means-plus-function style for drafting claims.
Bibliography


"Intellectual Property for CS Students"
http://www.cs.duke.edu/courses/cps182s/fall02/cscopyright/index.htm


Manifesto http://wwwsecure.law.cornell.edu/commentary/intelpro/manif5.htm


