January 1996

Software Objects: A New Trend in Programming and Software Patents

Keith P. Stephens

John P. Sumner

Follow this and additional works at: http://digitalcommons.law.scu.edu/chtlj

Part of the Law Commons

Recommended Citation
Available at: http://digitalcommons.law.scu.edu/chtlj/vol12/iss1/1

This Article is brought to you for free and open access by the Journals at Santa Clara Law Digital Commons. It has been accepted for inclusion in Santa Clara High Technology Law Journal by an authorized administrator of Santa Clara Law Digital Commons. For more information, please contact sculawlibrarian@gmail.com.
SOFTWARE OBJECTS: A NEW TREND IN PROGRAMMING AND SOFTWARE PATENTS*

Keith Stephens & John P. Sumner†

TABLE OF CONTENTS

I. INTRODUCTION .......................................................... 2
II. SOFTWARE OBJECTS ................................................... 4
   A. Overview ............................................................. 4
   B. Frameworks: Interconnected Object Classes .................. 5
      1. Encapsulation of Expertise ...................................... 5
      2. Object-Oriented Languages ..................................... 6
      3. Application Frameworks Control Program Flow .............. 6
      4. System Frameworks Facilitate Customized Solutions ...... 7
      5. Framework System Architecture ................................. 7
   C. Encapsulation, Polymorphism, and Inheritance ............... 8
      1. Objects .......................................................... 8
      2. Private and Public Variables .................................. 9
      3. Polymorphism .................................................... 9
      4. Inheritance Through Classes and Subclasses ............... 10
      5. Messages Before Objects ...................................... 10
      6. Instances ...................................................... 11
   D. More on Application Frameworks .................................. 11
   E. Basic Approach to Claiming ....................................... 12
III. PATENTABLE SUBJECT MATTER ....................................... 13
   A. Software Objects and the Constitution ........................ 13

* Copyright © 1995 Keith Stephens & John P. Sumner.
† Keith Stephens was Director of Intellectual Property at Taligent, Inc., in Cupertino, California. He recently joined Cooley Godward Castro Huddleston & Tatum in Palo Alto, California. John P. Sumner is a partner in the Minneapolis office of Merchant & Gould. Mr. Sumner chairs the firm's Electronic and Computer Law Practice Section and is currently the chair of the American Intellectual Property Association Electronic and Computer Law Committee. The positions stated in this article are those of the authors individually and not of their respective organizations. The authors wish to thank Daniel B. Curtis for his assistance with this article.
I. INTRODUCTION

The recent proliferation of software development has brought the problem of protecting computer software to a head. Software developers are confronted with the limited protection of copyright, the onerous vigilance of maintaining a trade secret, and the traditional reluctance of the Patent and Trademark Office (PTO) to issue software patents. This article discusses recent developments in patent protection for software, and specifically patent protection for object-oriented software. The goal is to inform the software industry of the availability and advantages of patenting software objects as articles of manufacture and to predict how the courts will interpret the patentability of software objects.

Currently, software patents are usually classified as falling within four categories of patentable subject matter: process, machine, manufacture and design.\(^1\) Process patents are written to claim what the software does, such as a method for presenting help information on a display screen.\(^2\) Machine patents, or patents "claiming an apparatus,"\(^3\)

---

1. See 35 U.S.C. § 101 (1988) ("Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title."); 35 U.S.C. § 171 ("Whoever invents any new, original and ornamental design for an article of manufacture may obtain a patent therefor, subject to the conditions and requirements of this title.").

2. E.g., U.S. Patent No. 5,434,965, claim 7 (process for balloon help):
   A method for displaying information, comprising the steps of:
   (a) displaying a plurality of icons;
   (b) moving a first of said plurality of icons;
are written to claim the implementation of a software program in a
computer "machine," such as an implementation of how to create a
menu on a display. Manufacture patents claim an article of manufac-
ture that has embedded on it a software package. For example, a man-
ufacture patent claim could be a novel, nonobvious software program
for displaying graphical information on a display. Finally, design
patents are written to claim software as an ornamental design. An
example of such a patent is an icon in the form of a trash can.

In comparing the four categories of patents, the advantages of
claiming software as an apparatus become apparent. In some cases,
the PTO will only allow narrow claims drawn to the specific imple-
mentation because the generic idea itself is old in the art. In the pro-
cess patent example above, if the method of displaying help on the
screen was old in the art, it would not be patentable. However, a spe-
cific implementation, such as an object-oriented design, may be novel
and nonobvious. Also, the PTO would reject either a method claim
or an apparatus claim because it is a generic invention based on the
prior art. However, it would allow a claim drawn to a particular im-
plementation. It is often easier to add implementation details to appa-
ratus claims, particularly if the implementation is based on the use of
particular hardware.

(c) detecting when said first of said plurality of icons is positioned proxi-
mal to said second icon of said plurality of icons;
(d) notifying said second of the plurality of icons that said first icon is
proximal to said second icon;
(e) starting a timer when said first icon is proximal to said second icon;
(f) generating a help message responsive to a context sensitive combina-
tion of said first icon and said second icon that clarify actions that can be
performed utilizing said first icon and said second icon through communi-
cation between the first and second icons; and
(g) displaying said help message when said timer has expired on said
display.

§ 101 "machine" with "apparatus").
5. Design patents for software icons are currently being held up by the PTO until the
office can resolve certain policy issues. See Daniel J. Kluth, Testimony of Daniel J. Kluth at the
Public Hearings by the U.S. Department of Commerce Patent & Trademark Office on Patent
Protection for Software-Related Inventions, U.S. PATENT AND TRADEMARK OFFICE,
(plord@uspto.gov), March 15, 1994, http://www.uspto.gov/text/pto/hearings/kluth.html (on file with the
Santa Clara Computer and High Tech. L.J.); see Notice of Public Hearings and
Request for Comments on Patent Protection for Software-Related Invention, DEPT OF COM-
Dec. 22, 1993) (on file with the Santa Clara Computer and High Tech. L.J.) (Topics C lists cases
stated pending guidelines which will be promulgated soon).
6. This is so unless the subject matter itself is nonstatutory under 35 U.S.C. § 101. The
test for patentability is whether the invention is novel under 35 U.S.C. § 102 and nonobvious
Object-oriented programming is rapidly becoming the industry’s preferred method for developing software. This technology is relatively new, and only in the last few years has the PTO issued patents covering it. Nevertheless, because object-oriented software satisfies all of the statutory requirements of the four patent categories, many software companies are filing for patents that claim software as objects.

Object-oriented programming paradigms utilize structured programming techniques to architect software objects in a computer. The objects are used for controlling the memory, processor, display and other peripheral devices, which perform new and innovative operations. An object-oriented software patent might claim a help object with an instance variable that indicates whether the help function is active or inactive. It might also claim instance methods for displaying or hiding the help message, depending on the help object’s current status. The combination and interaction of several software objects, as opposed to the objects themselves, may also be a patentable apparatus.

II. SOFTWARE OBJECTS

A. Overview

Object-oriented programming may be preferable over traditional methods for developing software because it facilitates designs in a tangible domain. A program is written by designing each object separately. Once the individual objects are implemented, they are connected together in a coherent fashion resulting in a modular system. Software objects are also extensible and portable, which means they can be easily reused in other programs. Most object-oriented development is performed in the C++ programming language. In the context of object-oriented programming, the objects are composed of (1) data structures and (2) operations on the object’s data. Together, these two elements enable objects to imitate the characteristics of vir-

7. Object-oriented programming is a “method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships.” GRADY BOOCHE, OBJECT-ORIENTED ANALYSIS AND DESIGN 517 (1994).

8. Companies filing for object-oriented patents include Microsoft, Apple Computer, Taligent, IBM, NeXT, Sun Microsystems, AT&T, and Schlumberger Technology Corporation.

9. A software object is a group of elements working together or in succession to perform a specific task, where the elements are the instance variables and methods of the object. See generally Booch, supra note 7, at 25-131.

10. An instance variable is a “repository for part of the state of an object. Collectively, the instance variables of an object constitute its structure. The terms field, instance, variable, member object, and slot are interchangeable.” Id. at 514.
tually any real-world entity as represented by its data elements and its behavior as represented by its data manipulation functions. In this way, objects can imitate, or model, concrete entities like people and computers. They can also model abstract concepts like numbers or geometrical shapes. The benefits of object technology derive from three basic principles: encapsulation, polymorphism, and inheritance. These principles allow them to fit easily into a framework of interconnected object classes.

B. Frameworks: Interconnected Object Classes

An important aspect of object-oriented programming is the framework approach to application development. One of the most informative definitions of frameworks came from Ralph E. Johnson of the University of Illinois and Vincent F. Russo of Purdue University. They offer the following definition: "[a framework] is a design of a set of objects that collaborate to carry out a set of responsibilities." From a programming standpoint, frameworks are essentially groups of interconnected objects that provide the prefabricated structure for a working application. For example, a user interface framework might provide the support and default behavior of drawing windows, scrollbars, and menus. Since frameworks are based on object technology, this behavior can be inherited and overridden to allow developers to extend the framework and create customized solutions. This is a major advantage over traditional programming since the programmer is not changing the original code, but rather extending the software to fit particular needs. In addition, developers are not blindly working through layers of code. The framework provides architectural guidance and modeling. The developers are then free to supply the additional, specific actions needed to achieve a specific result.

1. Encapsulation of Expertise

From a business perspective, frameworks are a way to collect, or encapsulate, particular knowledge and expertise in a reusable, portable manner. Corporate development organizations, independent software vendors (ISVs) and systems integrators have acquired expertise in particular areas such as manufacturing, accounting, or currency transactions. This expertise is embodied in their code. Frameworks enable organizations to capture and package the common characteristics of that expertise in the organization’s software objects.

---

11. See description infra part II.C.
Any particular problem need only be solved once. Future projects using the framework are constrained by it and must use the business rules and designs it embodies. Also, frameworks have a strategic asset implication for those organizations who have developed them. Such organizations can use frameworks to maximize their expertise in vertical markets such as manufacturing, accounting, or biotechnology. Frameworks also provide a distribution mechanism for packaging, reselling, and deploying their expertise.

2. Object-Oriented Languages

Frameworks have only recently emerged as a mainstream concept on personal computing platforms. This emergence has been assisted by the availability of object-oriented languages such as C++. Traditionally, C++ was limited to UNIX systems and researchers’ workstations instead of computers in commercial settings. C++, Smalltalk, and other object-oriented languages enabled a number of university and research projects to produce the precursors to today’s commercial frameworks and class libraries. Some examples of these are InterViews from Stanford University, the Andrew toolkit from Carnegie-Mellon University, and the University of Zurich’s ET++ framework.

Programming with frameworks requires a new way of thinking for developers accustomed to the traditional programming paradigm. In fact, it is not like programming at all in the traditional sense. In old-style operating systems such as DOS or UNIX, the developer’s own program provides all of the structure. The operating system provides services through system calls. The developer’s program executes a system call when it needs a service. Control returns to the calling program once the requested service has been provided. The program structure is based on the flow-of-control that is embodied in the code the developer writes. This is reversed when frameworks are used.

3. Application Frameworks Control Program Flow

When using frameworks, the developer must forego the tendency to think of programming tasks in terms of flow of execution. The developer is no longer responsible for the program’s flow-of-control; the framework is responsible. The programmer’s thinking must be

---

13. A class library is a set of objects that share a common structure and a common behavior. *Id.* at 512.
14. A system call is a predefined interface to access and to utilize operating system services.
focused on the responsibilities of the objects. The objects rely on the application framework to determine when they are to perform their tasks. The developer's routines are activated by code the developer did not write and has never seen. Developers experienced only in procedural programming may undergo a significant psychological barrier to this shift in control. However, once this methodology is understood, framework programming requires much less effort than other types of programming since most of the structure is already defined.

4. System Frameworks Facilitate Customized Solutions

In the same way that an application framework provides the developer with groundwork functionality, system frameworks provide system-level services which developers extend to create customized solutions. For example, consider a multimedia framework that could provide the foundation for supporting new and diverse devices such as audio, video, MIDI, and animation. The developer who needs to support a new kind of device must write a device driver. To do this with a framework, the developer only needs to supply the characteristics and behaviors that are specific to that new device. In this case, the developer supplies an implementation for certain member functions that will be called by the multimedia framework. An immediate benefit to the developer is that the generic code needed for each category of device is already provided by the multimedia framework. This means the developer of the device driver may develop write, test, and debug less code. In another example, a programmer may use separate input/output devices frameworks for SCSI devices, NuBus cards, and graphics devices. Because there is inherited functionality, each framework provides support for the common functionality found in its device category. Other developers could then depend on these consistent interfaces for implementing other kinds of devices.

5. Framework System Architecture

Recent innovations in operating system technology apply the concept of frameworks throughout an entire system. For the commercial or corporate developer, systems integrator, or OEM, this approach leverages all the advantages that have been illustrated for a framework at both the application level and system levels. Creating application in this architecture consists of writing domain-specific objects that adhere to the framework protocol. In this manner, the whole

---

16. OEM stands for original equipment manufacturer. OEMs are the manufacturers of computers, disks, and software products that supply equipment in commerce.
concept of programming changes. Instead of writing line after line of code that calls multiple API\textsuperscript{17} hierarchies, developers derive classes from preexisting frameworks within the system environment, and then add new behavior and/or overriding inherited behavior as desired. Thus, the developer's application becomes the collection of code that is written and shared with all the other framework applications. This concept is powerful because developers will be able to build on each other's work. It also provides the developer with the flexibility to customize as much or as little as they need. Some frameworks will be used just as they are. In other cases, the amount of customization will be minimal. In still other cases, the developer may make extensive modifications and create something completely new. This framework system architecture provides flexibility and increased extensibility while increasing functionality.

C. Encapsulation, Polymorphism, and Inheritance

There are three basic principles of object-oriented programming: encapsulation, polymorphism, and inheritance. When used with frameworks, they enforce structure onto object-oriented programs and objects.

Objects hide, or encapsulate, their data’s internal structure and the algorithms that implement their actions. Instead of exposing these implementation details, objects present interfaces that represent their abstractions with no extraneous implementation information. Objects interact by sending messages to each other. These messages cause the receiving object to perform one or more operations.

Polymorphism takes encapsulation a step further. In essence, the idea is: many shapes, one interface. A software component can make a request of another component without knowing exactly what constitutes that component. The component that receives the request interprets it and determines, according to its variables and data, how to execute the request. Thus, sending a draw request to a square object would result in the display of a square. The same identical request sent to a round object would result in the display of a circle.

The third principle is inheritance. Inheritance allows developers to reuse preexisting design and code and avoid the recreation of software from scratch. Rather, through inheritance, developers derive subclasses that inherit behaviors. The developers then customize these subclasses to meet their particular needs.

\textsuperscript{17} API stands for application program interface. An API is a predefined interface for accessing functions within a program.
1. Objects

Objects are defined by creating “classes” that are not objects themselves, but act as templates that instruct the compiler how to construct an actual object. A class may specify the number and type of data variables and the steps involved in the methods that manipulate the data. An object is actually created in the program by means of a special function called a “constructor.” The constructor uses the corresponding class definition and additional information, such as arguments provided during object creation, to create or instantiate the object. Likewise, objects are destroyed by a special function called a “destructor.” Objects may be used by manipulating their data and invoking their methods. For example, a class could define polygons. The variables may be the number of sides, lengths, and angles between the sides of the polygon. This class could then be used to construct objects such as squares or octahedrons.

2. Private and Public Variables

More specifically, an object can be designed to hide, or encapsulate, all or a portion of its internal data structures and its internal functions. During program design, a program developer can define objects having all or some of the data variables, and all or some of the related functions considered private, i.e., for use only by the object itself. Other data or functions can be declared public and made available for use by other objects or routines. Access to the private variables by other objects can be controlled by defining public functions (methods) for an object to access its private data. The methods form a controlled and consistent interface between the object’s private data and the outside world. Any attempt to write program code that directly accesses the object’s private variables causes the compiler to generate an error during program compilation. This error stops the compilation process and prevents the program from being run.

For example, in an object-oriented graphic system, the system consists of a number of objects that are clearly delimited parts or functions of the system. Each object contains information about itself and a set of operations that it can perform on its information or information passed to it. For example, an object could be named PERSON. The information contained in the object PERSON (its attributes) might be age, address, and occupation. These attributes describe the object PERSON. The object also contains a set of methods that perform operations on the information it contains. Thus, PERSON might be able

18. See Booch, supra note 7.
to perform an operation to change occupations from a doctor to a lawyer.

3. Polymorphism

Polymorphism is a concept that allows objects that have the same overall format, but work with different data, to operate differently in order to produce consistent results. For example, an addition method may be defined as variable A plus variable B (A+B). This same format can be used whether the A and B are numbers, characters or dollars and cents. However, the actual program code that performs the addition may differ widely depending on the type of variables that comprise A and B. Polymorphism allows three separate method definitions to be written, one for each type of variable (numbers, characters and dollars). After the methods have been defined, a program can later refer to the addition method by its common format (A+B). During compilation, the C++ compiler will determine which of the three methods is actually being used by examining the variable types. The compiler will then substitute the proper function code. Polymorphism allows similar functions that produce analogous results to be grouped in the program source code to produce a logical and clear program flow.

4. Inheritance Through Classes and Subclasses

Inheritance allows program developers to easily reuse preexisting programs and to avoid creating software from scratch. The principle of inheritance allows a software developer to specify that certain classes (and the objects that are later created from them) are related. Thus, objects that have common characteristics are grouped together into a class. A class is a template that enables the creation of new objects that contain the same information and operations as other members of the same class. Further, classes may be designated as subclasses of other base classes. A subclass inherits, and has access to, all of the public functions of its base classes as if these functions appeared in the subclass. Alternatively, a subclass can override some or all of its inherited functions, or may modify some or all of its inherited functions, merely by defining a new function with the same form. Overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass. See generally id. at 97-114.

19. Overriding or modification does not alter the function in the base class, but merely modifies the use of the function in the subclass. See generally id. at 97-114.
5. Messages Between Objects

Because objects are encapsulated, thus hiding their internal data and functions from each other, objects interact by sending messages. These messages cause the receiving object to take some action. In an object-oriented program, there are many communicating objects.

In this polymorphic environment, the receiving object is responsible for determining which operation to perform upon receiving a stimulus message. An operation is a function or transformation that may be applied to or by objects in a class. The stimulating object needs to know very little about the receiving object, and this simplifies execution of operations. Each particular object need only know how to perform its own operations and how to make the appropriate calls for performing those operations the particular object itself cannot perform.

6. Instances

An object created from a certain class is called an instance of that class. The class defines the operations and information initially contained in an instance. The current state of the instance is defined by the operations performed on the instance. Thus, while all instances of a given class are initially identical, subsequent operations make each instance a unique object. Instances inherit the attributes of their class. By modifying the attribute of a parent class, the attributes of the various instances are modified as well, and the changes are inherited by the subclasses. New classes can be created by describing modifications to existing classes. The new class inherits the attributes of its parent class, and the user can add anything that is unique to the new class. Thus, one can define a class by simply stating how the new class or object differs from its parent class or object. Classes that fall below another class in the inheritance hierarchy are called descendants, or children, of the parent class.

D. More on Application Frameworks

Although object-oriented programming offers significant improvements over other programming paradigms, program development still requires significant outlays of time and effort, especially if no preexisting classes are available as a starting point for adaptation. Consequently, one approach has been to provide a program developer with a set of predefined, interconnected classes that create a set of objects. Included with these objects are additional miscellaneous routines. These routines and objects are all directed to performing commonly-encountered tasks in a particular environment. Such
predefined classes and libraries are typically called "application frameworks" and essentially provide a prefabricated structure for a working application.

For example, an application framework for a user interface might provide a set of predefined graphic interface objects that create windows, scroll bars, menus, and other user interface elements. The framework might also provide the support and default behavior for these graphic interface objects. Since application frameworks are based on object-oriented techniques, the predefined classes can be used as base classes, and the built-in default behavior can be inherited by developer-defined subclasses. The defaults can be either modified or overridden to allow developers to extend the framework and create customized solutions in a particular area of expertise.

This object-oriented approach provides a major advantage over traditional programming since the programmer is not changing the original program, but rather extending its capabilities. In addition, developers are not blindly working through layers of code because the framework provides architectural guidance and modeling. At the same time, the object-oriented approach frees the developers to supply specific actions unique to the problem domain, thus extending the structure inherent in the framework to cover the problem domain.

E. Basic Approach to Claiming

Within object-oriented software patents, software objects are claimed with structure, function, and relationship. The structure and function are defined by the instance variables and methods of the object, and the relationship is defined by the interaction the instant object has with other objects in the program. Claiming a program as software objects is an efficient way of determining the scope of an invention because of the inherent delineation between these elements. Software objects are differentiated by comparing the instance variables and methods as well as their function and relationship to the whole program. Determining patentability in this manner aids prosecution by the PTO and results in strong, comprehensible patents. Further, software objects claimed as operational in a computer, or stored on a computer readable medium, are very likely to be upheld by the
SOFTWARE OBJECTS

This is because they fit under the statutory definition of patentable subject matter as an apparatus and article of manufacture.\textsuperscript{21}

III. PATENTABLE SUBJECT MATTER

To understand how the courts will approach the issue of software object patentability, Section III provides a brief history of past cases concerning software patents and a review of the current procedure used by the courts and the PTO for determining software patentability.

The United States Constitution provides Congress the power "[t]o 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."\textsuperscript{22} In accordance with this clause, Congress passed Title 35 to codify the law with respect to patents,\textsuperscript{23} and established the Patent and Trademark Office (PTO) to administrate the filing and prosecution of patent applications.

A. Software Objects and the Constitution

Software objects are entitled to patent protection because they further the Constitutional basis for allowing exclusive use of an invention in order to promote the progress of science.

Before object-oriented programming, software was implemented in higher level languages using procedural programming. Procedural programming is not inherently modular, nor does it facilitate portability or extensibility, unlike object-oriented programming. In object-oriented programming, software is designed by starting with objects already available in an application framework. If an available object is not completely suited to a task, it can be subclassed and extended into a new object with the necessary functionality. By inheriting all of the instance variables and methods, a subclassed object inherits all of the functionality provided by the parent class. Additional features can be designed into a subclass by adding a complementary new method or by overriding and changing a method of the parent class. Once the new object is designed and implemented, it becomes part of the application framework for other programmers to reuse and extend into further objects.

As software objects proliferate, it becomes easier to design and implement new programs. Development time decreases proportion-

\textsuperscript{21} E.g., \textit{In re} Iwahashi, 888 F.2d 1370 (Fed. Cir. 1989) (ROM and RAM are apparatuses or machines); \textit{In re} Beauregard, 53 F.3d 1583 (Fed. Cir. 1995) (floppy disk is an article of manufacture).
\textsuperscript{22} U.S. Const. art. I., § 8, cl. 8.
ally to the number of objects already available, and reliability increases because the reused objects have already been debugged and tested. However, unless there is an incentive to design and disclose new objects, the full benefit of object-oriented programming can never be realized.

Allowing inventors to obtain patents on new objects or improvements to old objects will further the Constitutional objective of promoting the progress of science by helping the software industry. Programmers will file for patents on new and improved objects in order to derive the benefits of royalty and licensing fees, and these objects will be collected into a software library that will become an invaluable resource. Other programmers will draw from the patent library rather than design and implement the objects themselves. Any new, useful or nonobvious object that is subclassed into a new and improved object will also be entitled to patent protection, and the new object will be added to the patent library. This step-by-step advancement in the technology is the same way that more traditional technologies advance. This proliferation of software through software objects will result in significant progress in the industry.

B. Judicial and Administrative Review of Patentable Subject Matter

1. The Supreme Court

The Patent Act defines patentable subject matter as follows: “Whoever invents or discovers any new and useful process, machine, manufacture, or composition of matter, or any new and useful improvement thereof, may obtain a patent therefor, subject to the conditions and requirements of this title.” Several Supreme Court cases provide the judicially determined exceptions to § 101 for patents involving software, with the main exception being patents drawn toward claiming pure mathematical algorithms or other pure scientific principles.

The Supreme Court is the ultimate authority on interpreting Title 35 and the Constitution. Therefore, the controlling definition of patentable subject matter is gleaned from those cases decided by the Supreme Court. However, the Court of Appeals for the Federal Circuit (CAFC), created in 1982, was given jurisdiction to hear all cases

---

24. In other words, it advances by standing on the shoulders of giants.
on appeal involving patent law. Since the CAFC's creation, the Supreme Court has refused to grant certiorari to any cases involving patentability. Thus, the CAFC's current interpretation of patentability is considered the controlling law and will remain so unless overruled by the Supreme Court.

In *Diamond v. Diehr*, the Supreme Court admonished that a mathematical algorithm is "a procedure for solving a given type of mathematical problem, and . . . an algorithm, or mathematical formula, is like a law of nature, which cannot be the subject matter of a patent." This standard has been applied by the Supreme Court in software cases since *Gottschalk v. Benson*. In *Gottschalk*, a patent was held invalid for claiming an algorithm for converting binary numbers to decimal numbers. In another early case, *Parker v. Flook*, the Supreme Court held a patent invalid for claiming an algorithm for the computation of an alarm limit. Mathematical algorithms are not patentable subject matter. However, software may be patentable when implemented on a computer.

In *Diamond v. Diehr*, the Supreme Court also stated that "[a] claim drawn to subject matter otherwise statutory does not become nonstatutory simply because it uses a mathematical formula, computer program, or digital computer." The invention in *Diamond v. Diehr* involved a method for molding precision synthetic rubber, and the process was controlled by a computer. The Court held a "process for molding precision synthetic rubber products falls within the § 101 categories of possibly patentable subject matter." "[This] conclusion . . . is not altered by the fact that in several steps of the process a mathematical equation and a programmed digital computer are used." Therefore, a process or apparatus that includes the use of a mathematical algorithm may be patentable subject matter as long as the algorithm itself is not the only thing claimed.

2. Court of Appeals for the Federal Circuit

In line with the Supreme Court decisions, the method adopted by the CAFC for analyzing mathematical algorithm statutory subject mat-

---

29. 450 U.S. at 186.
32. 450 U.S. at 187.
33. Id. at 184.
34. Id. at 185.
ter consists of a two part test called the Freeman-Walter-Abele test. First, the claim is analyzed to determine whether a mathematical algorithm is directly or indirectly recited. Next, if a mathematical algorithm is found, the claim as a whole is further analyzed to determine whether the algorithm is applied in any manner to physical elements (in apparatus claims) or limit claim steps (in process claims), and if it is, it “passes muster under § 101.” The goal [of the two part test] is to answer the question ‘What did applicants invent?’ If the claimed invention is a mathematical algorithm, it is improper subject matter for patent protection, whereas if the claimed invention is an application of the algorithm, § 101 will not bar the grant of a patent.

Before the advent of object-oriented programming, software was thought of as a means to implement algorithms, and other processes rather than as objects having structure, function and relationships. The courts have refused to enforce software patents that merely claim mathematical algorithms asserting that such algorithms are merely scientific principles, and, therefore, are nonpatentable subject matter under 35 U.S.C. § 101. Software inventors have avoided § 101 rejections by including elements other than pure mathematical algorithms in the claim.

a. In re Iwahashi

In In re Iwahashi, a patent that claimed a method for computing a correlation coefficient used in pattern recognition was upheld by the Federal Circuit because it included the use of a ROM (read-only memory) device. The invention was directed to an improved auto-correlation unit for a pattern recognition device implemented using digital circuitry. One of the mathematical requirements for the circuit was to generate the square of (Xn + Xn - Z). This calculation usually would have required digital multiplier circuitry.

The inventive concept was to store all the possible results of the multiplication step in the ROM. The values to be multiplied were supplied to the address lines of the ROM, and the ROM contained the
appropriate precalculated value. The result was that this circuitry pro-
vided a very fast multiplier for a limited range of possible values. The
invention could have been implemented as a look-up table\textsuperscript{42} in a com-
puter using random access memory.\textsuperscript{43}

The court distinguished between “algorithm” and “mathematical
algorithm” and pointed out that algorithms are patentable as processes,
whereas mathematical algorithms are not patentable.\textsuperscript{44} The court
applied the Freeman-Walter-Abele test and initially found the claim indi-
directly included a mathematical algorithm. Therefore, the second step
of the test required that the claim as a whole be further analyzed. “If it
appears that the mathematical algorithm is implemented in a specific
manner to define structural relationships between the physical ele-
ments of the claim (in apparatus claims) or to refine or limited claim
steps (in process claims), the claim being otherwise statutory,” the
claim is allowed under § 101.\textsuperscript{45} The court found that the apparatus
was statutory and further stated that “[t]he fact that the apparatus oper-
ates according to an algorithm does not make it nonstatutory.”\textsuperscript{46} The
court placed emphasis on the existence of a ROM in the algorithmic
process.\textsuperscript{47} Later, however, the majority in \textit{In re Alappat}\textsuperscript{48} stated that
“The Iwahashi court clearly did not find patentable subject matter
merely because a ROM was recited in the claim at issue; rather the
court held that the claim as a whole, directed to the combination . . .
was directed to statutory subject matter. It was not the ROM alone
that carried the day.”\textsuperscript{49}

\textit{b. In re Alappat}

\textit{In re Alappat},\textsuperscript{50} is a complex \textit{en banc} decision by the Court of
Appeals for the Federal Circuit. The invention at issue was directed to
a “means for creating a smooth wave form display in a digital oscillo-
scope.”\textsuperscript{51} One of the many issues before the court was whether a
“general purpose digital computer ‘means’ to perform the various
steps under program control was statutory subject matter under 35

\textsuperscript{42} A look-up table is a linked list in which an array of names are searched to determine

\textsuperscript{43} \textit{Iwahashi}, 888 F.2d at 1372.

\textsuperscript{44} \textit{Id}. at 1374.

\textsuperscript{45} \textit{Id}. at 1375.

\textsuperscript{46} \textit{Id}.

\textsuperscript{47} \textit{Id}.

\textsuperscript{48} 33 F.3d 1526 (Fed. Cir. 1994).

\textsuperscript{49} \textit{Id}. at 1544 n.24.

\textsuperscript{50} \textit{In re Alappat}, 33 F.3d 1526 (Fed. Cir. 1994).

\textsuperscript{51} \textit{Id}. at 1537.
U.S.C. § 101." Five judges comprised the majority, finding that the invention was statutory subject matter under § 101. Three judges did not take an expressed position, and two were opposed. Because the court was sitting en banc, its holding is authoritative unless the Supreme Court or the U.S. Congress overrule it.

The court in Alappat analyzed the Diamond v. Diehr, Parker v. Flook and Gottschalk v. Benson decisions and determined that:

[T]he Supreme Court never intended to create an overly broad, fourth category of subject matter excluded from § 101. Rather, at the core of the Court's analysis in each of these cases, lies an attempt by the Court to explain a rather straightforward concept, namely, that certain types of mathematical subject matter, standing alone, represent nothing more than abstract ideas until reduced to some type of practical application, and thus that subject matter is not, in and of itself, entitled to patent protection.

The court declared that the proper inquiry regarding the mathematical subject matter exception is whether the claimed subject matter, as a whole, is a disembodied mathematical concept that essentially represents nothing more than a law of nature, natural phenomenon, or abstract idea. "If so, Diamond v. Diehr precludes the patenting of that subject matter."

The court found that Alappat's invention, when looked at as a whole, was "directed to a combination of interrelated elements which combine to form a machine for converting discrete wave form data samples into anti-aliased pixel illumination intensity data to be displayed on a display means." This was not an abstract idea, but a machine that produced a "useful, concrete, and tangible result." The court also reaffirmed its prior holding that when a general purpose computer is programmed to perform a particular function it, in effect, becomes a special purpose computer. Thus, a programmed computer is a different machine from a computer that is programmed with a different program. Finally, the court held that a "computer operating pursuant to software may represent patentable subject matter, provided . . . the claimed subject matter meets all of the other requirements of Title 35. In any case, a computer . . . is apparatus not

52. Id.
53. Id. at 1543.
54. Id.
55. In re Alappat, 33 F.3d at 1543.
56. Id.
57. Id.
58. Id. at 1545.
59. Id.
Unless the Supreme Court decides to once again attempt to clarify § 101, this holding classifies computers executing a program as patentable subject matter.61

c. In re Lowry

In In re Lowry,62 Lowry’s subject matter was a “memory containing a stored data structure.”63 The examiner rejected this claim under § 101; the Board of Patent Appeals and Interferences (the Board) reversed the examiner, finding that the claims directed to a memory containing stored information, as a whole, recited an article of manufacture.64 The main issue before the panel of judges was related to § 103.65

The court could have reversed the Board and agreed with the examiner that the subject matter was not statutory. However, in dicta, the court stated that the “stored data [that makes up the data structures] adopt no physical ‘structure’ per se. Rather, the stored data exist as a collection of bits having information about relationships between the [structures].”66 The court found this to be the essence of electronic structure and that the data structures, being specific electrical or magnetic structures in memory, are more than mere abstraction.67 The court classified these data structures as “physical entities that provide increased efficiency in computer operation.”68

3. Patent and Trademark Office

The Patent and Trademark Office (PTO) has in the past established guidelines for examiners examining software patents69 in response to the Supreme Court and CAFC interpretation of Title 35. In directing the examiners on the examination of software cases, the PTO focuses on the significant points of law from the Diamond v. Diehr decision, which may be the most relevant with regard to object-oriented software. That case stated, “when a claim containing a mathematical formula implements or applies that formula in a structure or

60. In re Alappat, 33 F.3d at 1545.
61. Notice that both a computer and a program considered separately are completely useless and should fail § 101 because to be useful the computer and program must work together.
62. 32 F.3d 1579 (Fed. Cir. 1994).
63. Id. at 1581.
64. Id. at 1582.
65. Id. at 1583.
66. Id.
67. Id. at 1583-84.
68. Id. at 1584.
69. MANUAL OF PATENT EXAMINATION PROCEDURE (MPEP) §§ 2106-2106.02 (rev. Oct. 6, 1987).
process that, when considered as a whole, is performing a function that the patent laws were designed to protect . . . then the claim satisfies the requirements of § 101."70 The PTO also refers to the Freeman-Walter-Abele71 two-part test as applied in In re Iwahashi72 for determining patentability of software claims.

The PTO has recently issued proposed examination guidelines for computer-implemented inventions that further clarify and expand the protection afforded software through patents. These guidelines have already been applied by the court. The CAFC recently vacated their decision of December 19, 1994 and remanded the case of In re Trovato73 for consideration in light of Alappat and the aforementioned guidelines.

C. Software Objects: Statutory Subject Matter

After reviewing the judicially determined exceptions to § 101, as well as the PTO’s guidelines for determining patentability for claims to software, it should be apparent that, if a claim includes a mathematical algorithm, that algorithm must be applied to physical elements (structure) or a process that has practical utility.74 Requiring patent applicants to recite structure in the claims is a recurring theme in the case law and PTO guidelines. For example, the Supreme Court has stated that, "[t]ransformation and reduction of an article 'to a different state or thing' is the clue to patentability of a process claim that does not include particular machines."75

Software objects satisfy the statutory subject matter requirements of the 35 U.S.C. § 101 because they cause a computer to become a new, structured computer "apparatus," "article of manufacture," or computer-implemented "process." Allowable claims can be drafted toward software objects implemented by a computer or stored on a "computer" readable medium by reciting in the apparatus, article of manufacture or process claims the structure of the object as a whole. This structure can be the instance variables, instance methods, and the relationship with other objects in the system.

In conclusion, any computer program or structure that causes a computer to execute a process that is new, useful, and nonobvious is

---

70. Diamond v. Diehr, 450 U.S. at 192.
72. 888 F.2d 1370, 1374 (Fed. Cir. 1989).
73. 60 F.3d 807 (Fed. Cir. 1995).
patentable. Software objects are patentable under *In re Iwahashi*, *In re Alappat* and *In re Lowry* because what is claimed is an implementation of the software rather than a pure mathematical or scientific principle. Even if a mathematical algorithm or other scientific principle is used in the implementation of an object, the object is still patentable subject matter as long as there is a recitation of operation on a machine (for apparatus or method claims) or storage on media (for article of manufacture claims).

IV. OBJECT-ORIENTED PROGRAMMING

A. Menu Example

In a graphical user interface, a menu bar containing a set of menu commands is displayed to the user for performing certain tasks such as editing operations. Each menu command displays a set of related submenu commands when selected by the user. A submenu command might perform a certain action, display a user dialog with further options, or display a hierarchical menu of further submenu commands. A sample menu displayed by the ClarisWorks application running on an Apple Macintosh computer is shown in Figure 1.

![Sample Menu](image)

**Figure 1. Sample Menu**

---

76. ClarisWorks is an "Integrated Application" combining word processing, drawing, paint, and spreadsheet programs into a single application. ClarisWorks was created by the Claris Corporation.
Following the syntax used in patents, which refer to portions of figures utilizing reference numerals, The ClarisWorks menu bar (2) displays several menu commands, including Format command (4). When the user selects the Format command with the mouse, a set of submenu commands (5) is displayed for formatting the document, changing the rulers, formatting the text, and inserting a header or footer. The submenu command Style (6) is a hierarchical menu that displays a further set of submenu commands (8) for formatting text. If the user selects one of the submenu commands under the Style command, such as Bold (10) action command, the selected text will be shown in bold face. The Rulers submenu command (12) displays a dialog box as shown in Figure 2 with options for the user to set up the rulers.

![Figure 2. Rulers Submenu](image)

The general process of displaying commands in a menu for selecting with a mouse is not patentable because the idea is not novel. Almost every graphical user interface provides the menu command

---

77. U.S. Pat. No. Re 32,632 was awarded in an original application filed July 19, 1982, to William D. Atkinson at Apple Computer claiming a method (couched in an apparatus format) for the example menu system. A claim from that patent is:

9. A computer controlled display system having a display wherein a plurality of command options are displayed along a menu bar and sub-command items corresponding to each option are displayed once said option has been selected, comprising:

first display means coupled to said computer for generating and displaying said menu bar comprising said plurality of command options;

cursor control means coupled to said display system for selectively positioning a cursor on said display, said cursor control means including a cursor control device for movement over a surface, the movement of said cursor control device over said surface by a user resulting in a corresponding movement of said cursor on said display;
However, there may not be a specific implementation of a menu as a software object. A novel, object-oriented implementation of a menu for displaying commands is patentable as long as it is not suggested by the prior art. With this in mind, and assuming novelty and nonobviousness, an object-oriented implementation is presented below.

The first step in designing an object-oriented program is to identify the objects in the system. Once the objects are identified, the properties and function of each object are determined, and the relationship and interaction with the other objects are defined. In this example, the identifiable objects include a menu bar, a menu command, a sub-menu command and classes derived therefrom. The submenu command is subclassed into further objects such as an action command, hierarchical command, or dialog command. A class diagram of the identified objects is provided in Figure 3.

Figure 3 shows a MenuObject (14) as the base class from which all other menu classes are derived. The MenuBar (16) is a direct subclass of the MenuObject (14); therefore, it inherits the instance variables and methods of the MenuObject (14), as well as provides further instance variables and methods specific to a MenuBar (16). The MenuCommand (18) and SubMenuCommand (20) are also derived from the MenuObject (14), and the SubMenuCommand (20) is further subclassed into an HierarchicalCommand (22), ActionCommand (24), and DialogCommand (26). The HierarchicalCommand (22) is also

signal generation means including a switch having a first and second position coupled to said display system for signaling said computer of an option choice once said cursor is positioned over a first predetermined area on said display corresponding to an option to be selected, said user placing said switch in said second position while moving said cursor control device over said surface such that said cursor is over said first predetermined area;

second display means coupled to said computer for generating and displaying said sub-command items corresponding to said selected option;

said switch being placed in said first position by said user once said user has positioned said cursor over a second predetermined area corresponding to a sub-command item to be selected;

whereby an option and a sub-command item is selected and executed by said computer.

78. Graphical user interface operating systems including Apple Macintosh, Microsoft Windows, and IBM OS/2 provide a menu command capability.


80. Obvious is defined by 35 U.S.C. § 103 and has been interpreted to mean “suggested by the prior art.” In re Stencel, 828 F.2d 751 (Fed. Cir. 1987).

subclassed through multiple inheritance from the MenuCommand (18) in order to inherit the select method as described below.\textsuperscript{82}

The MenuObject (14) is an abstract class that serves as a template for the classes derived from the instance variables and methods common to all menu objects are defined at the MenuObject (14) level. For example, all menu objects encapsulate a name, a view coordinate to draw the name, a draw method, and a method for selecting the menu object when selected by the user. The name instance variable and method to draw the name can be implemented at the MenuObject (14) level for every menu object. The name might be a string, and the draw method a function that simply draws the name using the current font at the name view coordinates. Further, the draw method could be a virtual function so that subclass menu objects could override and modify the method, for instance, to change the current font.\textsuperscript{83} Since the method for selecting a menu object is different for every class, the MenuObject’s (14) select method would be a pure virtual function.\textsuperscript{84}

Referring again to Figure One, the MenuBar’s (2) select method would call the select method of the MenuCommand (4). The

\textsuperscript{82} Multiple inheritance is the term used to describe a class that is derived from more than one superclass. See Booch, supra note 7, at 514. See also supra part II.C.4 (describing inheritance).

\textsuperscript{83} Virtual methods are the mechanism for implementing polymorphism. When a menu object’s draw method is called, if the method has been overridden, then the draw method for that menu object will be called. See Booch, supra note 7, at 105-08. See also supra part II.C.3 (describing polymorphism).

\textsuperscript{84} A pure virtual function is simply a template to define the interface to a method contained in every subclass. The actual implementation of the method must be at the subclass level since the method is different for every subclass. See Booch, supra note 7, at 519.
MenuCommands (4) for a MenuBar (2) are stored in an instance variable specific to the MenuBar class (16). The select method of the MenuCommand (4) would highlight its name, and display a list of its SubMenuCommands (5) until one of the SubMenuCommands (6) is selected or the operation is canceled. The select method for a SubMenuCommand (12) would simply highlight the command name, whereas the select method for the HierarchicalCommand (6) would inherit the select method of the MenuCommand (18) and display a further list of SubMenuCommands (8).

The SubMenuCommand class (20) has a pure virtual method for executing the action associated with the command. When the user selects a SubMenuCommand, such as the Rulers DialogCommand (12), the MenuCommand (4) sends an execute message to the SubMenuCommand class (20). Since the selected SubMenuCommand (20) is a DialogCommand (12), the execute method of the DialogCommand class (24) is called and the Rulers dialog is displayed. Similarly, the execute methods of the other SubMenuCommands (20) are called when selected.

The benefit of this design, once it is implemented and debugged, is that it can be reused in other programs with minimal effort by other programmers. Further, if a new type of SubMenuCommand class (24) is desired, such as a pop-up menu, it can be implemented and integrated into the current design without modifying the basic operation as just described.

B. Example Claims

Claims that could be drawn toward the menu bar invention would depend on the scope of the prior art. For instance, if there were no prior art on the subject of object-oriented menu bars, a claim might be drafted as broad as:

1. An object-oriented menu bar for implementation by a computer in an object-oriented framework, comprising:
   (a) a set of menu command objects;
   (b) a display method for displaying the menu command objects; and
   (c) a select method for selecting one of the menu command objects.

---

85. The SubMenuCommands (5) for a MenuCommand (4) are stored in an instance variable of the MenuCommand class (18). The MenuCommand (4) displays its SubMenuCommands (5) by calling the draw method for each SubMenuCommand.
In the event the above claim is anticipated or obvious in light of the prior art, the claim can be narrowed by reciting more structure as in the next claim example.

1. An object-oriented menu for implementation on a computer in an object-oriented framework, comprising:
   (a) an abstract menu object comprising:
       a command name object;
       a method for drawing the command name object;
       a pure virtual select method for selecting the menu object;
   (b) a submenu command object derived from the abstract menu object, and comprising a pure virtual execute method;
   (c) a menu command object derived from the abstract menu object, comprising:
       a plurality of the submenu command objects;
       a select method for selecting one of the submenu command objects; and
   (d) a menu bar object derived from the abstract menu object, comprising:
       a plurality of the menu command objects;
       a select method for selecting one of the menu command objects.

Dependent claims could be drafted to further define the invention, and to provide protection in the event the independent claim is later held invalid. A dependent claim might, for instance, add or modify an instance variable or method. Example dependent claims are:

2. The object-oriented menu as recited in claim 1, further comprising:
   (a) an action command object derived from the submenu command object comprising:
       a select method and
       an execute method for carrying out a specific function; and
   (b) a dialog command object derived from the submenu command object, comprising:
       a select method and
       an execute method for displaying a user dialog.

3. The object-oriented menu as recited in claim 2, wherein the select method of the action command object and the select method of the dialog command object being called from the menu command object, and wherein the select method of the action command object and the select method of the dialog command object enable highlighting the command name when called from the menu command.

In the first dependent claim two, the independent claim one was further limited by adding the ActionCommand and DialogCommand
SubMenuCommand objects. Claim two is further limited in claim three by modifying the select methods to display the command name in a highlighted state when selected from the menu command.

C. Patentability of Example Claims

The object-oriented menu is patentable subject matter as defined by the courts as well as the PTO because the claims do not merely cover a mathematical algorithm or other purely scientific principle. What is patentable is the structure or functionality of a software design, that is, the properties, function, and relationship of the objects as implemented on a computer or stored on a computer readable medium. Even if an object uses a mathematical algorithm in one of its instance methods, the object is still patentable subject matter because “when a claim containing a mathematical formula implements or applies that formula in a structure or process which, when considered as a whole, [claims subject matter] which the patent laws were designed to protect . . . then the claim satisfies the requirements of § 101.”86 The fact that a software object incorporates a mathematical algorithm does not make it nonstatutory.87

When the PTO applies the two part Freeman-Walter-Abele test to patents claiming software objects that do not incorporate a mathematical algorithm, the analysis does not go beyond the first step because the claims do not recite a mathematical algorithm. Even if an object incorporates a mathematical algorithm, it would still be allowed as long as the claim in its entirety does not wholly preempt the use of that algorithm. In other words, the claim will be allowed as long as there are sufficient structural limitations recited in the claims to become statutory subject matter as an apparatus, process or article of manufacture under 35 U.S.C. § 101. For a software object, this means reciting the instance methods and variables that define the structure or function of the object (and/or relationship to other objects) as implemented on a computer or stored on a computer-readable medium. A claim to an object with one method step where that method merely recites a mathematical algorithm will probably not be allowed.

V. Preferred Format of Software Object Claims

In some recent patents covering software objects, the claims unnecessarily incorporate hardware elements, such as a data storage device and a data processor, in order to avoid a § 101 rejection under In

86. Diamond v. Diehr, 450 U.S. at 175.
87. In re Iwahashi, 888 F.2d at 1374.
Incorporating hardware elements into the claim provides structural limitations that can improve the chances of having the claim allowed. However, reciting hardware elements in a software object claim is usually irrelevant and unnecessary to the invention because the invention is normally couched in the structure and function of the software object itself rather than the hardware elements. The crux of a software object invention is in the software structure or function as implemented on a computer or stored in a computer-readable media. Including superfluous hardware elements often serves as nothing more than needlessly obscuring the underlying invention. Normally, the hardware elements of an object-oriented design are merely incidental and should not be included in the claim.

For instance, in the menu object example described above, incorporating a storage device and processor into the preamble or body of the claim would be irrelevant. The claim preamble recitation "for implementation on a computer" should suffice. Every software object runs on a computer with a storage device; therefore, any claim drawn to a software object implicitly includes these prior art hardware elements. If it is desired to have a claim particularly cover a machine or storage device, those limitations can be added as dependent claims. Incorporating generic hardware elements into a software object claim rarely differentiates the invention in any way from the prior art. In fact, omitting hardware elements from a software object claim more appropriately points out and distinctly claims the subject matter of the invention as required by the second paragraph of 35 U.S.C. § 112.

The preamble of software object claims should begin with, "[a]n object-oriented . . . ." Preambles written in this fashion will serve the purpose of putting the public on notice of the nature of the invention and placing the claimed invention in the correct category (i.e., software objects). Then, anyone reading the claim will immediately understand that the invention is related to object-oriented software. The body of the claim should simply state the instance variables and

---

88. See U.S. Patent No. 5,202,981, claim 14:

14. A process for manipulating a data stream comprising a second plurality of data objects in an object-oriented database management system comprising a data storage device, a database of a first plurality of data objects stored in said data storage device in a predetermined sequence, a data processor connected to said data storage device, said data processor executing an object-oriented database manager, for manipulating said first plurality of data objects; said process comprising the steps of . . . . (emphasis added).

89. 35 U.S.C. § 112, ¶ 2 ("The specification shall conclude with one or more claims particularly pointing out and distinctly claiming the subject matter which the applicant regards as his invention.").
methods necessary to define the function and structure of the object(s) and the relationship between the objects in the invention.

Drafting claims in this manner will make searching for prior art more efficient because query searches can be on “object-oriented . . .” For the above example, for instance, a search on “object-oriented menu” will return all the prior art patents claiming object-oriented menu designs. Further, having a specific form for software object claims will aid the PTO in the prosecution of the application. An examiner can more easily determine patentability of a software object claim over the prior art by simply comparing the instance variables and methods of the prior art objects to the claimed invention. Rejecting claims based on equivalent instance variables and methods will clarify the issue and bring the prosecution to a conclusion in an efficient manner.

VI. CONCLUSION

Object-oriented programming dominates today’s software development. Millions of dollars are spent each year on research and development to advance software technology.\(^{90}\) The advantages of object-oriented technology have resulted in shorter development time, better reliability, and more complex software at a lower cost. Software engineers are no longer forced to reinvent the wheel every time a new program is developed, rather they can plug and play with existing software objects. The development process is similar to how electronic components are assembled to develop a new device. The distinction between software and hardware has increasingly eroded for many years as more and more inventions incorporate software. Object-oriented programming is another step toward the general trend of enabling more inventive features in computer software. As this continues to occur and the distinction between software and hardware blurs and fades, any degradation of patent protection for software is untenable.

If an electronic device is statutory subject matter under 35 U.S.C. § 101, then a software object implemented on a computer or stored on a computer-readable medium must also be. A software object, when implemented on a computer, causes the computer to become a new apparatus with inherent structure as defined by the software object’s instance variables and methods, as well as the software object’s rela-

\(^{90}\) Taligent, the joint venture between IBM, Apple, and HP, was created for the sole purpose of creating an object-oriented operating system and related application frameworks. AT&T, the company that created C++, has an R&D budget of $3 billion, and 60 percent of that is devoted to software.
tionship to other objects in the system. The instance variables and methods enable software objects to model any real world entity, including patentable devices. There is no valid reason why a device should be patentable, while the software representation of that same device is not. Further, as interpreted by the Supreme Court, the CAFC, and the PTO, software objects implemented on a computer or stored on a computer-readable medium are statutory subject matter under 35 U.S.C. § 101.

The fact that software is statutory subject matter is shown by the Supreme Court decision in *Diamond v. Diehr*, which stated that "[a] claim drawn to subject matter otherwise statutory does not become nonstatutory simply because it uses a mathematical formula, computer program, or digital computer."91 Since a software object, when implemented by a computer or stored on a computer-readable medium, is statutory subject matter as an apparatus, process, or article of manufacture, it does not become nonstatutory simply because it is a computer program. Similarly, the CAFC's prerequisite of citing structure in an apparatus claim, as admonished in *In re Iwahashi*,92 is satisfied for claims to software objects implemented on a computer or stored on a computer-readable medium. This is because of the inherent structure provided by the object's instance variables and methods. Finally, software objects pass muster under the PTO's *Freeman-Walter-Abele* two part test even if a mathematical algorithm is incorporated into the claim provided that the claim, as a whole, is drawn to the structure of the object rather than to the algorithm in the abstract. The PTO considers software objects to be patentable as shown by the number of software object patents that have already been allowed.93

Software objects are patentable and they advance the Constitutional goal to "promote the progress of science"94 by creating an incentive for software developers to disclose their inventions. Broadening patent protection will further enhance software companies' ability to recoup their investment and continue to develop new and innovative products. This improved financial return will help fund company growth and create additional jobs. Further, the resulting repository of software objects will become an invaluable resource for other developers. The resulting proliferation of software objects will have a profound effect in promoting the progress of the software industry.

92. 888 F.2d at 1374.
93. See supra note 70 and accompanying text.
94. U.S. Const. art I, § 8, cl. 8.