ARTICLE

SOFTWARE LITIGATION IN THE YEAR 2000: THE EFFECT OF OBJECT-ORIENTED DESIGN METHODOLOGIES ON TRADITIONAL SOFTWARE JURISPRUDENCE

DAVID M. BARKAN†

Table of Contents

I. INTRODUCTION

Ever since *Whelan Associates v. Jaslow Dental Laboratory*,[1] courts dealing with software have attempted to understand the process of

† J.D. 1992, School of Law (Boalt Hall), University of California, Berkeley; A.B. 1987, Harvard University. The author wishes to thank Peter Menell and Jeremy Barkan for their helpful comments and criticism.

1. 797 F.2d 1222 (3d Cir. 1986), *cert. denied*, 479 U.S. 1031 (1987).

software design. In doing so, they have focused almost exclusively on traditional methods of procedural programming and "top-down" design.[2] Moreover, most commentators on software protection present this traditional model of software design before articulating their proposed level of protection.[3] While this model accurately reflects software design in the 1980s, the traditional approach is not well-suited to the exponential increase in size and complexity that will characterize software projects in the 1990s.[4] As one critic of traditional design noted: "If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."[5]

In order to address the problem of complexity, programmers are likely to turn to object-oriented design and analysis because it allows programmers to adopt an entirely different approach toward problem solving and strongly encourages the development of libraries of reusable software "components." The next generation of software cases is likely to involve alleged infringement of complete programs designed according to the object-oriented model or alleged infringement of reusable software libraries.[6] This article attempts to explain object-oriented design to judges

---

2. *See id.* at 1229-31 (describing the process as identifying the problem, outlining the solution, and then creating a flowchart, modules, and subroutines); Computer Assocs. Int'l v. Altai, Inc., 775 F. Supp. 544, 559 (E.D.N.Y. 1991) (citing expert testimony describing a computer program as "made up of sub-programs and sub-sub-programs, and so on"), *aff'd in relevant part*, 982 F.2d 693 (2d Cir. 1992); E.F. Johnson v. Uniden Corp. of Am., 623 F. Supp. 1485, 1501-02 n.17 (D. Minn. 1985); SAS Inst., Inc. v. S & H Computer Sys., 605 F. Supp. 816, 825 (M.D. Tenn. 1985) ("Beginning with a broad and general statement of the overall purpose of the program, the author must decide how to break the assigned task into smaller tasks, each of which must in turn be broken down into successively smaller and more detailed tasks."). Other cases have implicitly adopted this model in determining similarities between the plaintiff's and defendant's programs. *See, e.g.*, Plains Cotton Co-op v. Goodpasture Computer Serv., 807 F.2d 1256, 1260 (5th Cir. 1987) (analyzing evidence of organizational copying), *cert. denied*, 484 U.S. 821 (1987). Pearl Systems v. Competition Electronics, 8 U.S.P.Q.2d (BNA) 1520, 1523-24 (S.D. Fla. 1988) (accepting expert testimony on similarities at the "subroutine" and "module" level); Q-Co Indus., Inc. v. Hoffman, 625 F. Supp. 608, 614-15 (S.D.N.Y. 1985) (comparing similarities in program "modules").

3. *See* 3 MELVILLE B. NIMMER & DAVID NIMMER, NIMMER ON COPYRIGHT, § 13.03 [F] at 13-78.30 to .32 (1991); Peter S. Menell, *An Analysis of the Scope of Copyright Protection for Application Programs*, 41 STAN. L. REV. 1045, 1055-56 (1989); David Nimmer et al., *A Structured Approach to Analyzing the Substantial Similarity of Computer Software in Copyright Infringement Cases*, 20 ARIZ. ST. L.J. 625, 637-38 (1988); Gary L. Reback & David L. Hayes, *The* Plains *Truth: Program Structure, Input Formats and Other Functional Works*, COMPUTER LAW., Mar. 1987, at 1, 5.

4. For a general discussion of the inherent complexity of modern software see GRADY BOOCH, OBJECT-ORIENTED DESIGN WITH APPLICATIONS 2-8 (1991) [hereinafter BOOCH, OBJECT-ORIENTED DESIGN].

5. Booch, *Reuse of Software Components Could Reduce Costs*, GOV'T COMPUTER NEWS, Sept. 25, 1987, at 86.

6. The object-oriented model is most likely to arise first in cases involving graphical user interfaces. Apple Computer distributes an object library called MacApp which provides programmers with many tools necessary to implement the Macintosh user

and lawyers and to raise the problems that this new method of writing software will pose for courts applying traditional copyright and patent doctrines. The article presumes no technical background, but it does assume some familiarity with basic copyright and patent concepts. Part II of this article briefly reviews the traditional model of software development and notes some of the limitations that are encouraging the switch to object-oriented design. Part III presents the basic concepts in the object-oriented model and explains the process of designing software under this model. Part IV then evaluates the scope of copyright protection both for complete programs written using the object-oriented model and for reusable object libraries. While some existing case law and commentary could be used to support broad protection for such programs, this Section shows that pure copyright doctrine should not provide any protection for the aspects of the program that make it object-oriented. Part V analyzes patent protection for object-oriented software and concludes that sufficiently innovative programs and libraries could qualify for patent protection. Finally, Part VI considers the potential effects that copyright and patent protection would have on the growth of object-oriented technology.

II. TRADITIONAL SOFTWARE DESIGN METHODS

Traditional programming languages are based on the concept of a procedure, which allows programmers to write a small section of code which performs one small task.[7] Well-written programs begin with a top-down approach to software design that has been described by Nimmer:

> In practice, a programmer usually will start with a general description of the function that the program is to perform. Then, a specific outline of the approach to this problem is developed, usually by studying the needs of the end user. Next, the programmer begins to develop the outlines of the program itself, and the data structures and algorithms to be used. At this stage, flowcharts, pseudo-code, and other symbolic representations often are used to help the programmer organize the program's structure. The programmer will then break down the problem into modules or subroutines, each of which addresses a particular element of the overall programming

---

interface. Similarly, Symantec Corp. distributes a similar object-oriented library with its Pascal and C compilers. For a description of major software vendors working on object-oriented development, see Richard K. Aeh, *OOPS Are Picking Up Speed*, J. SYS. MGMT., Feb. 1991, at 19; Rick Whiting, *The Quest for a Better Way to Develop Software*, ELECTRONIC BUS., July 10, 1989, at 16. For a general discussion of the some of the problems that may slow widespread adoption of the object-oriented model, see Jeff Moad, *Cultural Barriers Slow Reusability*, DATAMATION, Nov. 15, 1989, at 87.

7. *See generally* ELLIOT B. KOFFMAN, PROBLEM SOLVING AND STRUCTURED PROGRAMMING IN PASCAL 52-59 (1985). The terms "procedure," "module," and "subroutine" are used interchangeably to denote a small number of programming instructions which perform a single task.

> problem, and which itself may be broken down into further modules and subroutines. Finally, the programmer writes specific source code to perform the function of each module or subroutine, as well as to coordinate the interaction between modules or subroutines.[8]

This process has also been described as "functional decomposition," since the "primary question addressed by the systems analysis and design is WHAT does the system do [or] what is its *function*?"[9] The complex function identified at this stage must be further decomposed into smaller functions, a process which is repeated until the problem can be "expressed as some combination of many small, solvable problems."[10]

Several important implications flow from this model that affect the way programmers are taught to approach problem solving. First, the traditional model forces programmers to focus on specific tasks that must be achieved, resulting in strong analysis of the procedures and functions that must be used, but little emphasis on data structures.[11] Data structures are usually conceived only after the procedures have been generally defined.[12] While diligent programmers may go back and rethink some of their procedures after analyzing their data structures, as a general rule, "[i]n a typical procedural programming language such as C or Pascal, programmers approach data and algorithms as separate entities."[13] Second, data to be used by several procedures are usually

---

8. NIMMER & NIMMER, *supra* note 3, § 13.03[F] at 13-78.31 to .32.

9. Brian Henderson-Sellers & Julian M. Edwards, *The Object-oriented Systems Life Cycle*, COMM. ACM, Sept. 1990, at 142, 145 (emphasis in original).

10. Menell, *supra* note 3, at 1055 (citation omitted).

11. *See* Tim Korson & John D. McGregor, *Understanding Object-Oriented: A Unifying Paradigm*, COMM. ACM, Sept. 1990, at 40, 46.

The term "data structure" denotes the symbolic representation of a particular area of memory where specific data will be stored. For example, if we wished to store data about this article, we could create a data structure in Pascal called a "record" that includes various fields for storing different pieces of information. The record could be defined as follows:

```
TheArticle = record
        Author: str;
        NumberOfPages: int;
        Issue: int;
        StartingPage: int;
end;
```

In this data structure, the name of the entire record is "TheArticle." The name of the author is stored in the field "Author," the length of the article is stored in the "NumberOfPages" field, the issue number is stored in the "Issue" field, and the starting page in the issue is stored in the "StartingPage" field.

12. An obvious exception would be a database program since the programmer is usually given highly specific information on what type of information must be stored in the database, thus leading to an initial focus on data structures.

13. Randy Leonard, *OOP: The Future for Macintosh Development*, MACTECH Q., Spring 1989, at 22.

defined in one place and can be accessed by any module or subroutine.[14] While this approach may seem logical and efficient, it creates severe problems as soon as a complex program needs to updated or revised since it "leads to the stack of dominoes effect familiar to anyone working in program maintenance whereby changes to one part of a software system often cause a problem in an apparently dissociated program area."[15] As a result, each time a data structure needs to be changed or refined, one would have to identify all the procedures that rely on the old definition of the data structure and change them accordingly. Finally, since the top-down approach forces programmers to think in terms of a series of single functions, programmers are less likely to incorporate evolutionary changes in the data structures into the big picture of the overall system.[16]

In general, the traditional model provides few easy ways to reuse existing pieces of software, thus making software development less efficient than other engineering disciplines that are accustomed to reusing existing components.[17] The close dependence between procedures and the specific definition of data structures makes it extremely difficult to reuse selected procedures in a different project. While programmers certainly copy solutions to certain problems from earlier projects and from public domain sources, rarely can programmers lift an existing procedure from another project and incorporate it into their program without serious modification.

The traditional model leads to several additional problems from a project management perspective. While the top-down model has worked well until recently, software projects in the 1990s will involve new levels of complexity and will require better systems for managing multiple programmers working on different parts of one system. As software will increasingly be required to model real world behavior in meaningful

---

14. Ray Duncan, *Redefining the Programming Paradigm*, PC MAG., Nov. 13, 1990, at 526 ("[Y]ou visualize the data to be worked on as sitting in one place, while various routines call upon each other to do things to the data."); Henderson-Sellers & Edwards, *supra* note 9, at 146; Chris Terry, *Objects Facilitate Modular, Reusable Code*, EDN, Nov. 9, 1989, at 85.

15. Henderson-Sellers & Edwards, *supra* note 9, at 146.

16. *Id.*

17. Peter Coffee, *Honing the Software Equivalent of the Transistor*, PC WK., Sept. 25, 1989, at 38 (comparing object-oriented development to electrical engineering which emphasizes the reuse of standard building block components); Chris Terry, *Reusable Software Requires Building Blocks*, EDN, Jan. 3, 1991, at 59 ("Power-supply designers don't have to manufacture their own capacitors and resistors, let alone their connectors, nuts and bolts. They use parts that conform to universal (or at least widely accepted) standards. Yet software-systems designers have to write code for almost every function in their system except the services the operating system provides.").

ways, the human capacity for managing complexity will be severely tested.[18]

While traditional methods of functional decomposition provide one way of managing complexity, they are constrained in ways that make it difficult to manage projects that require large numbers of programmers. Since changes made by one programmer can effect other disparate parts of the program in a ripple fashion, traditional design eventually breaks down in extremely complex projects. Of course, the traditional model can be improved by forcing each programmer to adhere to detailed specifications that dictate how each programmer's portion interacts with the program as a whole.[19] While this modification of the traditional method was sufficient for "programming-in-the-large," it may not be sufficient for the "programming-in-the-colossal" that will be required in the 1990s.[20]

III. THE OBJECT-ORIENTED MODEL

A. Overview

While object-oriented principles were originally developed in the 1960s,[21] the current object-oriented model is an attempt to address the modern problem of "programming-in-the-colossal." Generally speaking, the object-oriented model emphasizes use of small, discrete components which can be used without any knowledge of how they work internally, thus breaking the tight dependency between data structures and procedures that constrained the traditional model. Several positive benefits flow directly from this one assumption:

> Object-oriented decomposition yields smaller systems through the reuse of common mechanisms, thus providing an important economy of expression. Object-oriented systems are also more

---

18. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 14 ("As we first begin to analyze a complex software system, we find many parts that must interact in a multitude of intricate ways, with little perceptible commonality among either the parts or their interactions: this is an example of disorganized complexity. As we work to bring organization to this complexity through the process of design, we must think about many things at once. For example, in an air traffic control system, we must deal with the state of many different aircraft at once, involving such properties as their location, speed, and heading. . . . Unfortunately, it is absolutely impossible for a single person to keep track of all of these details at once.").

19. *Id.* at 29-30 (discussing the development of modular programming techniques in late third-generation programming languages).

20. *Id.* at 32.

21. For a good history of the development of object-oriented languages, see Duncan, *supra* note 14 (noting that object-oriented programming languages were developed first in 1967 and then implemented in the 1970s at Xerox's Palo Alto Research Center, but were not practical for true commercial programming projects until the development of fast and efficient C++ compilers).

> resilient to change and thus better able to evolve over time, because their design is based upon stable intermediate forms. Indeed, object-oriented decomposition greatly reduces the risk of building complex software systems, because they are designed to evolve incrementally from smaller systems in which we already have confidence.[22]

Object-oriented programming is better suited for complex projects, because it more naturally models complex behavior in the real world. Through the concept of *inheritance*,[23] object-oriented programming focuses on hierarchical relationships between different components in a particular real-world system. For example, rather than considering a rectangle and a square to be two separate objects, the object-oriented model views the square as a particular kind of rectangle with special properties. This view is far more than a simple conceptual device. On a practical level, a programmer who wishes to simulate the behavior and properties of a square does not have to start from scratch; rather, the programmer starts with a model of the rectangle which may already exist in a library and then simply adds those properties that are unique to the square. Moreover, this model closely fits the way humans approach physical systems in the real world:

> For example, with just a few minutes of orientation, an experienced pilot can step into a multi-engine jet aircraft he or she has never flown before, and safely fly the vehicle. Having recognized the properties common to all such aircraft, such as the functioning of the rudder, ailerons, and throttle, the pilot primarily needs to learn what properties are unique to that particular aircraft. If the pilot already knows how to fly a given aircraft, it is far easier to know how to fly a similar one.[24]

As a result, the programmer can approach a new software project as an incremental learning process that closely models real behavior. A programmer seeking to simulate the behavior of an F-15 fighter jet would first start with programs that simulate the behavior of a standard commercial jet. If those earlier programs were written according to object-oriented principles, the programmer can simply start with the old program and then incrementally add those behaviors and processes that make the F-15 different from the commercial jet.

B. The Basic Concepts of the Object-Oriented Model

Before further exploring this approach to software design, it is necessary to define some basic concepts that are essential to the object-oriented model. Rather than present these concepts in a vacuum, it is useful to illustrate them in the context of an actual software project.

---

22. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 16.
23. *See infra* Section III.B.2.
24. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 12.

Assume that you are given the following software project which we will call "QuadWorld."[25] You are told that QuadWorld must allow the user to draw and manipulate various types of quadrilaterals,[26] such as squares, rectangles, parallelograms, and rhombi. Specifically, the user must be able to choose a particular shape, draw the shape, rotate the shape, move the shape in any direction and erase the shape. The user should also be able to have the program calculate the area of any selected shape. We will use these requirements to illustrate the basic principles of the object-oriented model.

1. *OBJECTS, ENCAPSULATION, AND MESSAGES*

An *object* is the basic programming unit in the model and essentially combines the traditional concepts of data structures and procedures into a single entity. For example, an object designed to represent a rectangle would include two pieces of data: length and width. It would also contain a complete set of procedures to draw, rotate, move, and otherwise manipulate the rectangle. In this way, the object captures both the state (data regarding length and width) of the rectangle as well as its behavior (the set of procedures for manipulating the rectangle).[27] Once the object is defined, any other part of QuadWorld can use that rectangle by simply sending it a *message*, which is a command telling the object to perform one of its defined behaviors. The definition of such a rectangle object might look like this:[28]

---

25. This example is taken directly from KURT J. SCHMUCKER, OBJECT-ORIENTED PROGRAMMING FOR THE MACINTOSH 32-35 (1986). Where figures or drawings are adapted directly from this text, they will be appropriately cited.

26. A quadrilateral is any geometric figure with four sides.

27. For a more formal definition of an object, see Korson & McGregor, *supra* note 11, at 42:

> Objects are the basic run-time entities in an object-oriented system. Objects take up space in memory and have an associated address like a record in Pascal or a structure in C.
>
> The arrangement of bits in an object's allocated memory space determines that object's state at any given moment. Associated with every object is a set of procedures and functions that define the meaningful operations on that object. Thus, an object encapsulates both state and behavior.

28. Each object-oriented programming language uses slightly different terms to define the basic concepts of object-oriented programming. The definition given here is not meant to mimic any particular language but rather to provide an easily understandable illustration of the basic concepts.

Object Definition of a Rectangle:
Internal Data:
length
width
current position of the top left corner of the rectangle on the screen
Messages that the object is able to perform:
Create, Draw, Move, Stretch, Rotate, Calculate Area
Internal implementation of those messages:
Calculate Area:
Area = length X width
Create: Code for implementing the create message
Draw: Code for implementing the draw message
Move: Code for implementing the move message
(Code for remaining messages as above)

Several observations must be drawn from this definition. First, note that the data fields are called *"internal"* data. In the object-oriented model, data structures are completely private to the object and cannot be used directly by any other part of the program. Similarly, the internal implementation of each message, such as the actual source code that would draw the rectangle, is also private to the object. When another part of the program wishes to draw a rectangle, it simply sends the "Draw" message to the object. Other parts of the program do not care how the object implements that message or what data structures the object uses to perform the draw message. In essence, the other part of the program tells the object "draw yourself, and I don't care how you do it."

This process of "hiding" the internal data structures and the implementation of messages within the object illustrates the principle of *encapsulation*. Since "no part of a complex system should depend on the internal details of any other part,"[29] encapsulation is critical to successful "programming-in-the-colossal." Since the object's internal structure is private to the object, changes can be made to the internal structure without having any effect on the rest of the program. For example, the definition of the object could be changed as follows (changes are shown in *italics*):

29. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 45. Booch defines encapsulation as the "process of hiding all of the details of an object that do not contribute to its essential characteristics." *Id.* at 46.

Alternative Object Definition of a Rectangle:

Internal Data:

Position of the top, left corner

Position of the bottom, right corner

(Note that the length and width pieces of data have been deleted)

Messages that the object is able to perform:

Create, Draw, Move, Stretch, Rotate, Calculate Area

Internal implementation of those messages:

Calculate Area:

Area = (first coordinate of bottom, right corner – first coordinate of the top left corner) X (second coordinate of the top left corner – second coordinate of the bottom right corner)

Create: Code for implementing the create message

Draw: Code for implementing the draw message

Move: Code for implementing the move message

(Code for remaining messages as above)

The beauty of the object-oriented model is that these internal changes can be made with absolutely no effect on any other part of the program. Unlike traditional programming, where changes in one procedure ripple through many other parts of the program, the internal definitions of objects can be changed many times with only minimal effect on the overall program. Moreover, tasks can be divided among numerous programmers without requiring the extensive coordination that is necessary when writing code with traditional methods. Each programmer need only be told the desired characteristics of a particular object; how the programmer chooses to write the internal code for that object doesn't matter to the other programmers working on the project. In fact, an object with the desired characteristics may already exist in a company-owned library of objects. As a result, many tasks can be completed without requiring any programmer to write new code.

In fairness to the traditional model, good programming practice can produce some of this modularity within procedural techniques. However, such modularity was usually achieved in a sporadic, informal, and inconsistent fashion.[30] In contrast, the object-oriented model requires

30. While late third-generation programming languages allow for separately compiled modules, this concept was used primarily to allow several programmers to work on the same project rather than as an independent tool for abstraction:

> Modules were rarely recognized as an important abstraction mechanism; in practice they were used simply to group logically related subprograms. Most languages of this generation, while supporting some sort of modular structure, had few rules that required semantic consistency among module interfaces . . . . Unfortunately, because most of these languages had dismal

programmers to incorporate the concept of encapsulation into their basic approach to building software.[31]

2. *CLASSES AND INHERITANCE*

The concept of a *class* extends the object-oriented model by providing a way to describe a group of objects that have similar properties and behavior. For example, if more than one rectangle is needed at a time, we need not define each rectangle individually. Instead, we define a *class* of rectangles, which serves as a template for creating rectangle objects. Each rectangle object is called an *instance* of the rectangle class. A class definition includes the data that describe each object, called *instance variables*, the messages that the classes must accept,[32] and the code that implements each message, known as *methods*.[33] A class can also be thought of as a "factory," which is like a

> cookie cutter that stamps out new instances of its class, new objects, whenever necessary. All instances of a given class have the same structure although the actual data stored in any one object may be different from the data stored in another object of the same type, just as all cookies stamped out with the same cookie cutter have the same shape even if they are made of different types of dough and decorated differently.[34]

In other words, if the user wants to draw two different rectangles of different sizes, our program will use the class definition for the rectangle to create two rectangle objects which possess the same types of behavior but have different values stored in their width and length variables.

The concept of a class does not become truly powerful, however, until combined with the concept of *inheritance*.[35] Inheritance is the primary organizing principle behind object-oriented design and the major reason why this model naturally follows the hierarchical structure of real-

---

> support for data abstraction and strong typing, such [semantic] errors could be detected only during execution of the program.

Id. at 30.

31. See the discussion of the object-oriented design process *infra* Section III.C.

32. The complete set of messages accepted by a particular class is sometimes called the class's protocol. SCHMUCKER, *supra* note 25, at 17.

33. Again, the precise terminology varies somewhat depending on the particular object-oriented programming language. Instance variables, messages, and methods are the terms used by Object Pascal and Symantec Corporation's object extensions to the C programming language. *See, e.g.*, SCHMUCKER, *supra* note 25, at 17; SYMANTEC CORP., THINK C OBJECT-ORIENTED PROGRAMMING MANUAL 20 (1991) [hereinafter THINK C OBJECT-ORIENTED PROGRAMMING MANUAL]. The programming language C++ instead uses the terms data member, message, and member function, respectively. *See generally* BJARNE STROUSTRUP, THE C++ PROGRAMMING LANGUAGE (1st ed. 1986).

34. SCHMUCKER, *supra* note 25, at 18.

35. Inheritance also distinguishes the class concept from the idea of user-defined types that is found in Pascal and C. *Id.* at 21.

world entities.[36] Given any particular class, we can define a *sub-class*[37] or *immediate descendant* which automatically inherits all of the behavior and properties of the original class, (now called the *super-class*[38] or *immediate ancestor*); we then take this sub-class and add any additional behavior (through new messages and methods) and any additional properties (through new instance variables) that make this sub-class different from its super-class. For example, once we have defined the "rectangle" class, we might realize that a square is simply a special type of rectangle. We could then simply define the "square" class to be a sub-class of the "rectangle" class. The "square" class would inherit all the instance variables, messages, and methods of the "rectangle" class. The "square" class would already "know" how to respond to the messages for draw, move, and rotate. At this point, we would also add the features that make a square different from a rectangle.

Object-oriented languages provide two different ways to differentiate the behavior of a sub-class from its super-class. First, we could simply add a new message and a corresponding method to the definition of the sub-class. For example, we might add a message to our definition of the "square" sub-class that automatically draws the largest circle that just fits inside the square and touches each side exactly once. The super-class rectangle could not have had this message because it's geometrically impossible for a circle within a rectangle to touch each side exactly once. But, the square is a special kind of rectangle,[39] and this message could be performed on a square.

In addition, object-oriented languages allow a sub-class to *override* methods inherited from the super-class. For example, we would want our square to respond to the message "calculate area" just as we wanted our rectangle to respond to that message. Normally, the sub-class square inherits both the message and the method which contains the actual programming code that tells the object how to respond to that message. In the case of the rectangle, the method for "calculate area" multiplied the rectangle's length by its width. However, we know that the square is a special kind of rectangle whose length is equal to its width. Thus, we

---

36. Inheritance is also a more powerful tool for achieving meaningful abstractions: Inheritance is the most promising concept we have to help us realize the goal of constructing software systems from reusable parts, rather than hand coding every system from scratch. Procedural abstraction has worked well for some select domains, such as mathematical libraries, but the unit of abstraction is too small, the procedural focus not general enough, and the parameter mechanism too rigid.

Korson, *supra* note 11, at 43.

37. C++ uses the term "derived class." STROUSTRUP, *supra* note 33, at 30.

38. C++ uses the term "base class." *Id.*

39. The "is a special kind of" terminology is a particularly useful way to think of the relationship between sub-classes and super-classes.

might want to override the method for "calculate area" with a new method which calculates the area by simply squaring the value of any side.[40] When the program sends a "calculate area" message to an object of the "square" class, the object will calculate the area by squaring one side rather than using the method of "length times width" that it initially inherited. The ability to override methods provides an important tool for customizing the behavior of sub-classes and for taking advantage of efficiencies that might be available in sub-classes that cannot be used in the more generic super-classes.

This inheritance concept can be easily applied to the structure of the QuadWorld application. At the highest level, we must define a *root class* which is a class that has no super-classes above it. In this case, we might define a root class, called "quadrilateral," which defines only the most generic properties and behaviors common to all four-sided geometric figures. After that, we could construct the simple inheritance chart that is shown in Figure 1.[41]

40. Of course, this example is fairly trivial and would not provide any efficiency improvement, but it nonetheless shows how sub-classes can customize particular inherited behavior. A more useful example would arise when we originally define the rectangle class. While the rectangle class would inherit the methods of the parallelogram class, the formula for calculating the area of a parallelogram is considerably more involved than the simple "length X width" formula that can be used for the rectangle class.

41. This figure is adapted directly from SCHMUCKER, *supra* note 25, at 22.

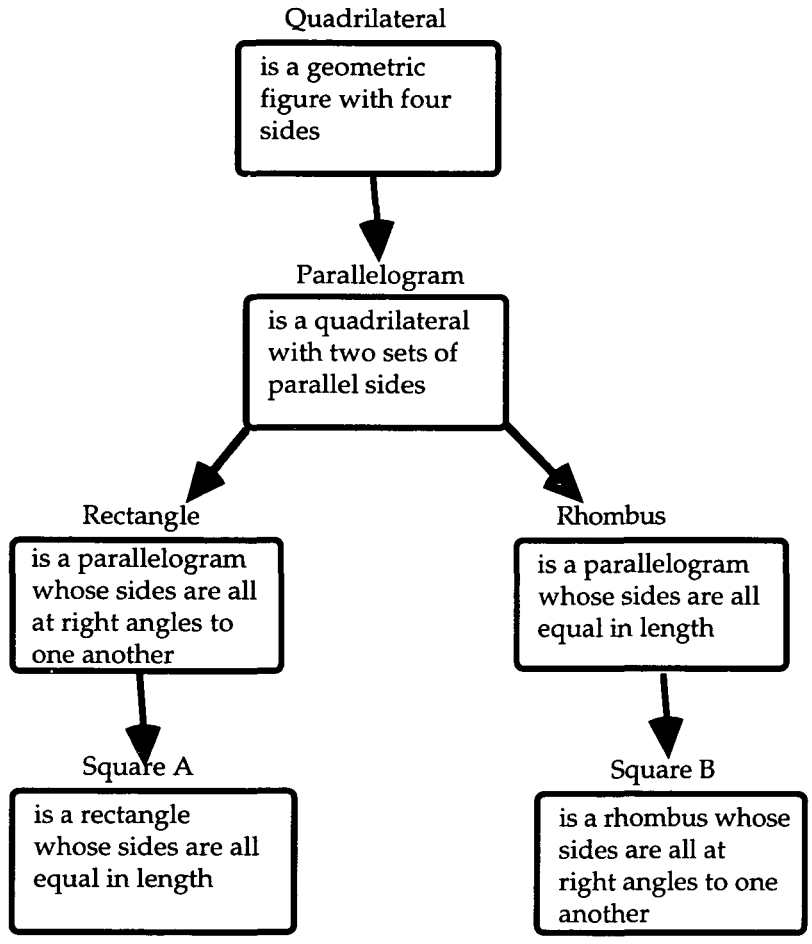

Figure 1: Simple Inheritance Chart for Quad World

A simple inheritance chart thus provides a logical method for organizing the basic relationships among various objects. Moreover, it forces us to recognize common features among different objects and then allows us to take pre-existing objects and modify them to fit our needs by building sub-classes from them. Moreover, "[i]nheritance not only supports reuse across systems, but it directly facilitates extensibility within a given system . . . . [I]nheritance minimizes the amount of new code needed when adding additional features . . . and minimizes the amount of new code that must be changed when extending a system."[42]

42. Korson & McGregor, *supra* note 11, at 43.

Real world behavior does not always fit this simple hierarchical structure, however. The object-oriented model can account for more complex relationships between classes with the concept of *multiple inheritance*.[43] Looking at Figure 1, we notice that a square has properties that make it a special kind of rectangle and properties that simultaneously make it a special kind of rhombus. Rather than having two classes of squares as in figure 1, we can improve our model by having a single class, called "square" that inherits from both the rectangle and the rhombus classes. The "square" class will inherit the ability to build a parallelogram with right angles from the rectangle class and will simultaneously inherit the ability to build a parallelogram with equal sides from the rhombus class.[44] As a result, we can create the "square" class without having to write any new methods, since all of the square's behavior is based on the combined properties of the "rectangle" and "rhombus" classes.[45] The improved model for QuadWorld is shown in Figure 2.[46]

43. Since multiple inheritance is more difficult for the compiler to handle than simple inheritance, not all object-oriented development packages support multiple inheritance. For example, Symantec's Think C does not support it, while a full implementation of C++ would support it. *See* THINK C OBJECT-ORIENTED PROGRAMMING MANUAL, *supra* note 33, at 62.

44. SCHMUCKER, *supra* note 25, at 277.

45. If any messages are defined in both the rectangle class and the rhombus, then we must have some way to specify whether we want the square to inherit the rectangle's method for that message or the rhombus' methods. Multiple inheritance allows us to specify either the rectangle or the rhombus as the *primary immediate ancestor class*. In any inheritance conflict, the square will inherit the methods of the primary immediate ancestor class. *Id.* at 278.

46. Adapted from *id.* at 277.

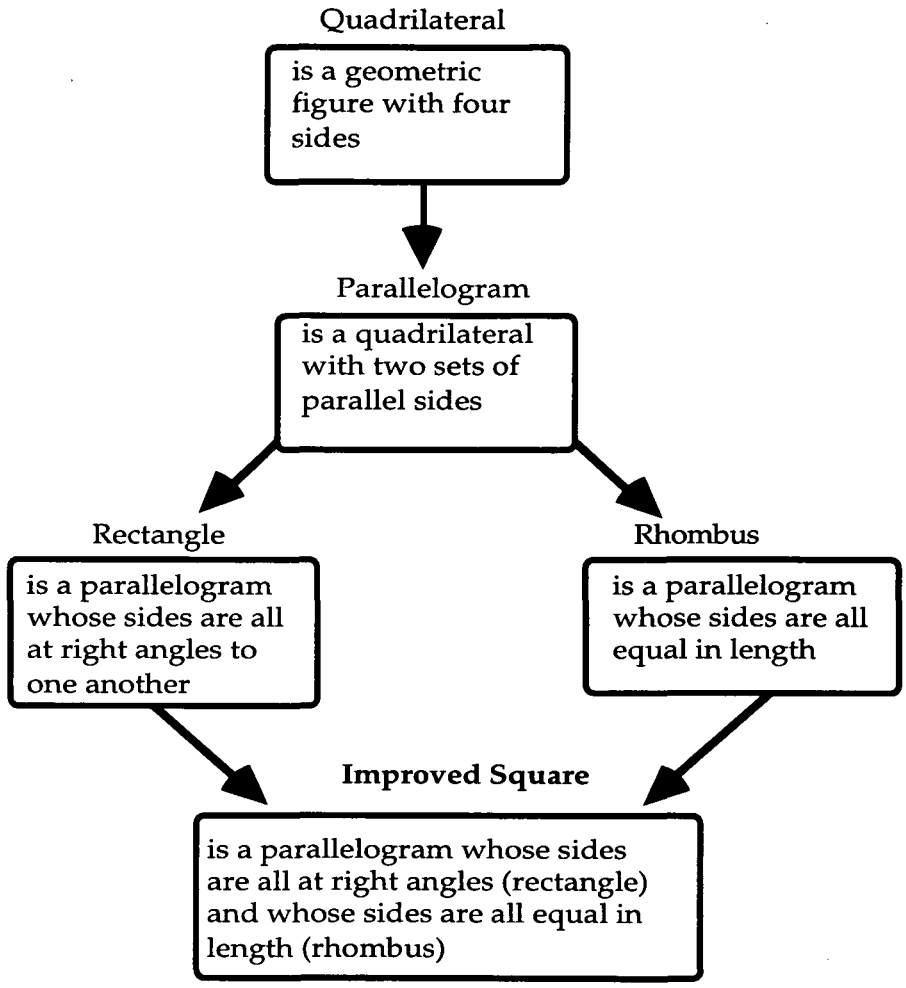

Figure 2: Improving Quad World with Multiple Inheritance

Multiple inheritance is a powerful concept that allows the programmer to model real-world entities which blend the characteristics of two or more super-classes. At the most basic level, multiple inheritance allows us to build classes as simple combinations of existing classes. For example, suppose we were working on an existing system that is designed to track the progress of efforts to save endangered wildlife, and we were instructed to add a class for "leopards."[47] Suppose further that the system already contained classes for "endangered" and "wild cat." Since leopards are both wild cats and endangered, we would

47. This example is taken directly from Korson & McGregor, *supra* note 11, at 58.

start by defining it as a sub-class of both of these existing classes. In addition, we can use multiple inheritance to model the behavior of entities that *blend* the characteristics of other objects. For example, a "houseboat" is not literally the combination of a house and a boat. Nonetheless, houseboats do possess some of the characteristics of a house and some of the characteristics of a boat. Again, a good starting point for building the "houseboat" class would be to define the class as a sub-class of both a "house" class and a "boat" class.[48] We could then add the characteristics that make a houseboat something more than the literal combination of a house and a boat by adding new messages and methods and by overriding the behavior of houses and boats that don't really apply to houseboats.[49]

3. *POLYMORPHISM AND DYNAMIC BINDING*

In conventional programming languages, each variable has a *static type* which is defined when the program is written and remains unchanged while the program is running. In the object-oriented model, each object is defined as belonging to a particular class when the program is written; however, when the program is actually running, objects are not bound to their original class and instead may be treated as if they belong to any sub-class of the original class. Since any object can be treated as belonging to one class at one moment and as belonging to a different class at a later moment, object-oriented programming languages are said to allow for *dynamic typing*.

Polymorphism is simply a more general description of the concept of dynamic typing. The basic idea behind polymorphism is that if "Y inherits from X, [then] Y is an X, and therefore anywhere that an instance of X is expected, an instance of Y is allowed."[50] For example, consider the QuadWorld program again. Suppose the user had drawn a number of different quadrilaterals on the screen and we wished to display a textual list of all the different types of objects the user had already drawn. We might want to display a message to the user that reads, "you have drawn two squares, three parallelograms and one rectangle." One convenient way for the program to keep track of this information would be to create an object which keeps track of all the quadrilaterals currently displayed on the screen. Here's one way we could define that object:

---

48. SCHMUCKER, *supra* note 25, at 276.

49. Since houseboats probably have more in common with boats than houses, we would probably use the boat class as the primary immediate ancestor class. *Id.* at 278.

50. Korson & McGregor, *supra* note 11, at 45.

Class Definition of "Current Screen":

Internal Data:

linked list[51] of square objects currently displayed

linked list of rectangle objects currently displayed

linked list of parallelogram objects currently displayed

linked list of rhombus objects currently displayed

linked list of quadrilateral objects (not falling into any of the above categories) currently displayed

Messages that the object is able to perform:

Add a square to the square list, delete a square from the square list

Add a rectangle to the rectangle list, delete a rectangle

Add a parallelogram to the parallelogram list, delete a parallelogram

. . . (similar messages for the other shapes)

Polymorphism allows us to find a better way to define this object. First, we note that every shape is a sub-class of the class quadrilateral. That means that wherever we have a data structure (such as a linked list) or a method that expects to use a quadrilateral, it will also accept any sub-class of the quadrilateral class. As a result, we can drastically reduce the complexity of the class "current screen" by redefining it as follows:

Class Definition of "Current Screen" Using Polymorphism:

Internal Data:

linked list of quadrilateral objects currently displayed

Messages that the object is able to perform:

Add a quadrilateral to the list, delete a quadrilateral from the list

When the program is actually running, we know that each element will be some type of quadrilateral, but we don't know which elements will contain which type of quadrilateral until the user has drawn some shapes on the screen. At any given moment, each element in the list will have a dynamic type which can be a square, rectangle, parallelogram, rhombus, or quadrilateral depending on which shapes are currently displayed on the screen. For example, if the user has drawn three objects on the screen, a square, a rectangle, and a quadrilateral, the "current screen" object would then contain the linked list shown in Table 1.

---

51. A linked list is simply a data structure that allows us to store pieces of data in a sequential chain. Each distinct piece of data in the list is called an "element" in the list. In the object-oriented world, each element can be an object.

Element #:	1	2	3
Original Definition:	Quadrilateral	Quadrilateral	Quadrilateral
Dynamic Type:	Square	Rectangle	Quadrilateral

Table 1: Illustration Of Dynamic Typing In A Linked List

Dynamic Binding is closely associated with the idea of polymorphism and dynamic typing. Dynamic binding builds on these concepts by allowing a particular object to respond differently to a particular message depending upon its dynamic type at a given moment while the program is running.[52] For example, assume that our program accepts a command from the user to display the area of each shape currently on the screen. Dynamic binding makes this feature easy to

Element #:	1	2	3
Original Definition:	Quadrilateral	Quadrilateral	Quadrilateral
Dynamic Type:	Square	Rectangle	Quadrilateral
Class Method Invoked In Response To Message	Square	Rectangle	Quadrilateral
Formula Used In Response To The Message: "Calculate Area"	square of the length	length X width	generic formula applicable to any four-sided shape

Table 2: Illustration Of Dynamic Binding In A Linked List

implement and automatically incorporates the special methods we wrote to take advantage of the fact that there is a simpler formula to calculate the area of a rectangle than the area of a generic quadrilateral.

To implement this feature, the program responds to the user's request to calculate the area of each displayed shape by sending the "calculate area" message to each object in the linked list stored in the "current screen" object. When each object in the list receives the "calculate area" message, it will use the method associated with its dynamic type rather than the method associated with its originally defined class. Thus, the linked list will respond as shown in Table 2.

---

52. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 63. ("Static binding means that the types of all variables and expressions are fixed at the time of compilation; *dynamic binding* (also called *late binding*) means that the types of all variables and expressions are not known until runtime.").

Dynamic binding and polymorphism provide several immediate advantages. First, they encourage a high degree of generalization by permitting the programmer to write procedures that apply to any quadrilateral, but that respond with the optimal code depending on whether the particular quadrilateral is a square, rectangle, or parallelogram. In the object-oriented world, we simply send the "calculate area" message to some unknown quadrilateral and it doesn't matter that we have no way of knowing what type of quadrilateral will actually receive that message when the program is running. In contrast, traditional programming would require us to add code that essentially said "If the particular quadrilateral is a square then use the procedure for squares, but if the particular quadrilateral is a rectangle then use the rectangle procedure, but if the particular quadrilateral is a parallelogram, then use the parallelogram procedure, . . . otherwise use the generic quadrilateral procedure."[53]

Moreover, dynamic binding and polymorphism also promote reusability by allowing other programmers to create new sub-classes of the quadrilateral class and know that they will automatically work in any procedure that expects to use the quadrilateral class. The programmers also know that if they have written new methods in the sub-class that override methods in the quadrilateral class the new methods will automatically be used when any procedure sends a message to their objects.

---

53. Of course traditional programming languages have a shorthand expression for this problem. In Pascal, the programmer would use a "case" statement that lists the names of different procedures for the different quadrilaterals, and in C the programmer would use a "switch" statement that listed the procedures. Nonetheless, these statements are no more than shorthand expressions for the long quotation in the text.

For those readers familiar with Pascal or C, consider a procedure that must re-draw a screen filled with various quadrilaterals. We could implement this procedure in Object Pascal by the following piece of code:

```
for i:= 1 to Number_of_Shapes do
    current_figure.item(i).draw;        {current_figure is an array
                                         of quadrilateral objects, and
                                         draw is a message in each of
                                         the quadrilateral subclasses
                                         that tells the object to draw
                                         itself}
```

Korson & McGregor, *supra* note 11, at 46. "At each pass through the loop, the code matching the dynamic type of current_figure.item(i) will be called. Note that if additional kinds of shapes are added to the system, this code segment remains unchanged. Contrast the resulting simplicity and extensibility as compared with a traditional case statement design." *Id.*

C. The Process of Designing Software Under the Object-Oriented Model

Given the concepts of objects, classes, inheritance, polymorphism, and dynamic binding, we can formulate an analytical approach for writing software that takes advantage of the object-oriented model. There have been many formal attempts to define an "object-oriented approach" to software design;[54] this Section outlines the basic features common to most of these models. In reading this Section, compare this model to the traditional model of software design as understood by the *Whelan* court and Nimmer.

1. *IDENTIFY THE OBJECTS AND CLASSES THAT COMPRISE THE "PROBLEM DOMAIN"*

The first step in approaching object-oriented design is to learn as much as possible about the problem that the program is supposed to solve ("the problem domain"). As a first approximation, the programmer should approach the problem not as a computer scientist but rather by becoming an expert in a specific domain. For example, a programmer writing a navigational system for an airplane should initially learn from pilots, air controllers, and aeronautical engineers how navigation works:

> Essentially, the developer must act as an abstractionist. By studying the problem's requirements and/or by engaging in discussions with domain experts, the developer must learn the vocabulary of the problem domain. The tangible things in the problem domain, the roles they play, and the events that may occur form the candidate classes and objects of our design, at its highest level of abstraction.[55]

Once the developer learns the vocabulary and physical items used by pilots, air controllers, and aeronautical engineers, the programmer can begin to identify specific classes that will be needed to write navigational software. At this stage, the programmer is examining the problem domain from a fairly high level of abstraction.[56] Various commentators have identified formal categories that may help suggest candidate classes as illustrated in Table 3.

---

54. *See, e.g.*, BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4; Henderson-Sellers & Edwards, *supra* note 9; Korson & McGregor, *supra* note 11; Ronald J. Norman, *Object-oriented Systems Analysis: A Methodology for the 1990s*, J. SYS. MGMT., July 1991, at 32; Rebecca J. Wirfs-Brock & Ralph E. Johnson, *Surveying Current Research in Object-Oriented Design*, COMM. ACM, Sept. 1990, at 104.

55. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 191.

56. Norman, *supra* note 54, at 33 ("It is not necessary and certainly not required that all possible objects be identified during this step. Only the most intuitive and obvious ones may be identified here, while others or refinements of these may be identified during a later step.").

Physical Items	planes, wings, engines, fuel pump, radio beacon
Roles	pilot, copilot, navigator, passenger
Events	landing, take-off, turning, putting down landing gear
Interactions	clearance from air controller, radio contact, schedules, connections with other planes
Places	airport, destination, origin

Table 3: Types Of Classes That Are Likely To Be Used[57]

This approach has several advantages over traditional software design. First, rather than asking "what tasks must the program perform," object-oriented design asks "how would those who will be relying on this program describe their problem, and what would *they* identify as the major actors (both human and inanimate) in the problem domain." This direct focus on the problem domain forces the programmer to address the specific needs of users in the problem domain before writing any code. In contrast, the traditional programming model promotes an early emphasis on the "tasks" that the software must perform and thereby removes the focus from the problem domain. At an early stage of the design process, the traditional programmer becomes bound to the specific instructions that will be used to write the program, often before potential users have identified all of their requirements. Second, the object-oriented approach helps to reveal commonalities that may exist across similar applications (vertical domain analysis) as well as commonalities that can be reused in different parts of the same application (horizontal domain analysis):

> For example, when starting to design a new patient-monitoring system, it is reasonable to survey the architecture of existing systems to understand what key abstractions and mechanisms were previously employed and to evaluate which were useful and which were not. Similarly, an accounting system must provide many different kinds of reports. By treating these reports as a single domain, a domain analysis can lead the developer to an understanding of the key abstractions and mechanisms that serve all the different kinds of reports. The resulting classes and objects reflect a set of key abstractions and mechanisms generalized to the immediate report-generation problem; therefore, the resulting design is likely to be simpler than if each report had been analyzed and designed separately.[58]

---

57. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 141 (summarizing categories proposed by Shlaer, Mellor, Ross, Coad, and Yourdon); *see also* Norman, *supra* note 54, at 40, Table 3 (identifying categories as "tangible items," "roles played by people or organizations," "incidents which happen at a specific point in time," "interaction [sic] that have a transaction-like quality," and "specification [sic] that have table-like qualities such as sales offices, state codes, standard industry codes, [and] tax rates").

58. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 142-43.

The object-oriented programmer's first written output is apt to be a rough list of classes and objects whose names imply their basic role in the problem domain and which will be used as the "common vocabulary of discourse among the developers."[59] Most important, these classes and objects should be subject to continual revision as the programmer follows the other three steps, thus leading to iterative and evolutionary changes in the original model of the problem domain, or what some commentators have called "Round-Trip Gestalt Design."[60] Steps 2, 3, and 4 in the model are all explicitly designed to foster such reevaluation of the problem domain.

In contrast, the traditional programmer's first written task using top-down design is to produce a rough flowchart of the program, which, by its very nature, is farther removed from the problem domain, closer to the stage of writing actual software code, and more likely to lock the programmer into tight dependencies among different parts of the program. The traditional programmer's tendency is to then parcel out pieces of the project to different programmers based on the original flowchart. While nothing stops the design team from refining the flowchart later, nothing in the traditional top-down model encourages iterative or evolutionary changes in the basic flowchart. In fact, the risk that changes in one part of the program will ripple through all other parts of the program actively discourages such changes.[61]

---

59. *Id.* at 192 (noting also that "[i]n most cases, this step takes a small amount of time relative to the other three steps. Often, a single chief designer will draft a list of candidate classes and objects and then review this list with peers as a kind of sanity check." *Id.* at 191-92).

60. *Id.* at 188 ("This style of design emphasizes the incremental and iterative development of a system through the refinement of different yet consistent logical and physical views of the system as a whole."); *see also* Henderson-Sellers & Edwards, *supra* note 9, at 148 ("Both top-down analysis and bottom-up class design, seen as the hardest part of the entire object-oriented software life cycle, must therefore be either concurrent or, at least iterative.") (footnote omitted).

61. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 188. The entire "top-down" vs. "bottom-up" approach to design has been the subject of significant debate within the software community. It is important to recognize that object-oriented design is neither "top-down" nor "bottom-up":

> Assume that we are faced with the problem of staffing an organization to design and implement a fairly complex piece of computer hardware. We might use horizontal staffing, in which we have a waterfall progression of products, with systems architects feeding logic designers feeding circuit designers. This is an example of top-down design, and requires designers who are "tall skinny men," as Druke calls them, because of the narrow yet deep skills that each must possess. Alternately, we might use vertical staffing, in which we have good all-around designers who take slices of the entire project, from architectural conception through circuit design. The skills that these designers must have leads Druke to call them "short fat men." Unfortunately, given its inherent complexity, software development often demands that we employ "tall fat people."

2. *IDENTIFY THE STRUCTURE AND SEMANTICS OF THE OBJECTS AND CLASSES*

At this stage, the programmer must specify the behavior and properties that each object will possess. One possible approach is to write "a script for each object, which defines its life cycle from creation to destruction, including its characteristic behaviors."[62] For example, once we have identified a "radio beacon" object in our navigational software, we might write a script that reads "beacon object is created when a plane is close enough to receive the signal from that beacon, and then the beacon is expected to send a radio signal at a pre-set frequency and at pre-set intervals, and then the beacon is expected to continue this behavior until the plane passes the beacon and leaves the beacon's range." Similarly, we might define a "landing gear" object which is created as soon as the software is running and is expected to be able to keep track of whether the landing gear is up or down, and send an alarm message if the landing gear is in the wrong position. The process of writing these scripts should also force the programmer to reevaluate the original list of objects and classes identified in step 1. For example, when we define the landing gear as being able to send an alarm message, we then realize that we never identified the need for an "alarm bell" object that would receive the alarm message and be used to display an alarm message on the navigator's computer screen. In this way, step 2 is iterative because it forces the programmer to reevaluate the decisions made in step 1.

In addition, identifying the behavior of each object may reveal other sub-classes that could be introduced. For example, an analysis of a bookkeeping program in step 1 might reveal the need for an "invoice" object. However, once we specify the attributes of an invoice in this step, we might realize the need for additional objects, such as "header," "account summary," and "list of transactions," that represent the different sections that make up the invoice.[63] Conversely, this analysis

---

Id.; *see also* Henderson-Sellers & Edwards, *supra* note 9, at 146 (noting that "object-oriented (OO) design and analysis has many attributes of both top-down and, perhaps predominately, bottom-up design. Since one of the aims of an OO implementation is the development of generic classes for storage in libraries, an approach which considers both top-down analysis and bottom-up design simultaneously is likely to lead to the most robust software systems.").

62. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 192 (noting that "[t]his step is much harder than the first and takes much longer. This is the phase in which there may be fierce debates, wailing and gnashing of teeth, and general name-calling during design reviews. Finding classes and objects is the easy part; deciding upon the protocol of each object is hard").

63. Norman, *supra* note 54, at 33 (suggesting that the programmer look for whole-to-part relationships and generalization-to-specialization relationships at this point). While this activity may blur some of the distinctions between step 2 and step 3, such blurring of

could also reveal the need to redefine some of the super-classes. For example:

> [A] class of 'bird' with an attribute that "birds can fly" is successful until we consider the Southern Hemisphere and "penguins," "ostriches," "kiwis" etc. In this case, one solution is to introduce an additional level in the inheritance hierarchy by introducing two children classes of class bird as "flying bird" and "non-flying bird" and redefining the parent class to remove the attributes relating to flight. This process tries to develop a logical hierarchy of objects so there are no "missing" objects.[64]

3. *IDENTIFY THE RELATIONSHIPS AMONG OBJECTS AND CLASSES*

In step 3, the programmer must identify the relationships among the previously identified objects and classes. First, the programmer must develop the inheritance relationships and define the structure of super-classes and sub-classes. In doing so, the programmer is likely to uncover additional "patterns among classes, which cause us to reorganize and simplify the system's class structure, and patterns among cooperative collections of objects, which lead us to generalize the mechanisms already embodied in the design."[65]

One way to formulate a concrete representation of these relationships is by building a *semantic data model*.[66] For example, if we were designing a program that was intended to control the traffic lights at a busy intersection, we might construct the partial semantic data model shown in Figure 3 in which rectangles denote classes and circles denote the functional relationships between classes connected by arrows.

---

specific steps is consistent with the iterative and evolutionary style of object-oriented design.

64. Henderson-Sellers, *supra* note 9, at 150 (footnotes omitted).

65. BOOCH, OBJECT-ORIENTED DESIGN, *supra* note 4, at 193.

66. Korson, *supra* note 11, at 47.

Using this data model, we can then determine what messages each object must accept. One useful conceptual device is to consider each message as a "service" that the object is capable of providing to any other object. Then, using the scripts from step 2 to determine what behaviors each object will exhibit, the programmer can determine what kind of services each object will need to fulfill its role.[68] As would be expected from the theme of iterative development, this process is likely to reveal that certain objects require services that no current object yet provides. The programmer must then either add that service to an existing object or create a new object to handle that service. As an alternative, the relationship between two objects can be considered to be a contract in

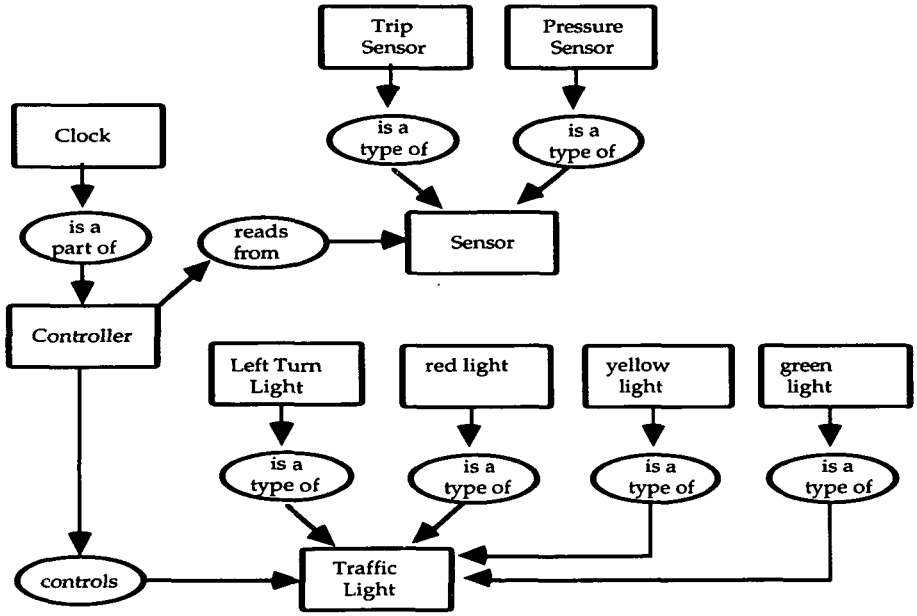

Figure 3: Semantic Data Model for Controlling Traffic Lights[67]

which one object is a "client" that requests certain services from another object which is a "server" and fulfills those requests.[69] Again, the programmer must make sure that every object that needs a contract for a particular service has a corresponding server object to fulfill that contract.

67. This model is adapted from Korson & McGregor, *supra* note 11, at 48 (fig. 8).
68. Henderson-Sellers & Edwards, *supra* note 9, at 150.
69. Wirfs-Brock & Johnson, *supra* note 54, at 110-11.

4. *IDENTIFY THE PUBLIC INTERFACES AND SERVICES PROVIDED BY EACH OBJECT AND CLASS*

At this point, we know what role each object plays and what services the object provides to other objects. Using that information, we can define the general type of data structures needed by each object and the methods that the object will need to provide services to other objects. As the internal structure of a particular object is developed, the programmer may discover that this object can be built by using pre-existing libraries of more primitive objects.[70] At the end of this stage, the programmer can begin writing the actual source code for each method, perhaps treating each method as a miniature program that can be approached using traditional procedural techniques.

IV. COPYRIGHT PROTECTION FOR OBJECT-ORIENTED SOFTWARE

Before examining the scope of copyright protection, it is important to recognize when the fact that the object-oriented model was used to design a piece of software matters and when it does not. If a programmer writes software under the object-oriented model, the programmer will use an object-oriented programming language to write the high level source code for the program. This source code is then translated by a compiler program into object code which is a series of 1's and 0's. These 1's and 0's represent low level commands which the microprocessor can understand and execute. At the same time, the compiler can also produce an assembly language version of the source code. Assembly language is a human-readable listing of object code in which each low-level microprocessor instruction is represented by a single word, such as "jump," "store," or "link." However, once the program has been translated into object code or assembly language, the fact that the original source code was written in an object-oriented programming language is virtually impossible to detect. The microprocessor itself has no concept of object-oriented principles;[71] therefore, the compiler produces a program that in object code form is indistinguishable from a program written according to traditional design methods.

As a result, in a case where the plaintiff alleges that the defendant copied the object code or assembler versions of the program, which can

---

70. Henderson-Sellers & Edwards, *supra* note 9, at 150.

71. While some computer manufacturers are touting "object-oriented operating systems," this statement does not mean that the microprocessor itself understands object-oriented principles. Instead, this feature means that the operating system is designed so that a program written in high level source code can interact with the operating system by using a pre-defined library of objects that perform the functions of the operating system. For example, the operating system running on a machine with a graphical user interface may supply libraries of objects for windows, icons, and menus.

arise only when verbatim copying of the program has occurred,[72] the court can proceed without worrying about principles of object-oriented design. But, object-oriented principles are absolutely critical when the plaintiff alleges that the defendant had access to the original source code and copied it. During the trial, both plaintiff and defendant will have to produce the source code for their programs, and the court will have to determine whether the programs are "substantially similar."[73] In weighing the evidence of similarity, the court will need to understand how principles of object-oriented design affect that comparison.

The starting point for any discussion on copyright protection for software is 17 U.S.C. § 102(b), which excludes protection for "any idea, procedure, process, system, [or] method of operation."[74] While the theoretical limits imposed by § 102(b) seem clear, courts and commentators have struggled with the practical application of the section to computer software. This Section will review the two dominant approaches to § 102(b) and determine how they apply to object-oriented software. In addition, this Section analyzes an alternative approach which advocates an ad hoc balancing of the economic effects of protecting the plaintiff's work, but concludes that courts will not adopt this approach because it has no support in copyright doctrine and it is problematic as a matter of innovation policy.[75] Finally, this Section concludes that since the "behavioral" aspects of software are particularly dominant when the object-oriented model is used to write software, pure copyright doctrine provides almost no protection for the *object-oriented* aspects of software.

---

72. While it is theoretically possible to work backwards from the assembler version of the source code, this process is virtually impossible for a program of any complexity. Particularly as software projects are increasingly characterized by "programming-in-the-colossal," the assertion that a defendant could have reproduced the plaintiff's source code by disassembly is ludicrous. Thus, a defendant will find copying the object code useful only if the defendant can sell verbatim copies of the plaintiff's program, perhaps in a foreign market where explicit piracy is tolerated. For a discussion of the technical difficulties involved in disassembly, see G. Gervaise Davis III, Reverse Engineering and the Computer Industry: A Battle Between Legal and Economic Principles (1991) (unpublished presentation on file with the *High Technology Law Journal*); Ronald S. Laurie, *Protection of Trade Secrets in Object Form Software: The Case for Reverse Engineering*, COMPUTER LAW., July 1984, at 1. *But cf.* Allen R. Grogan, *Decompilation and Disassembly: Undoing Software Protection*, COMPUTER LAW., Feb. 1984, at 1 (arguing that disassemblers and decompilers may allow object code to be converted so that much of the logic of the program is revealed).

73. *See* NIMMER & NIMMER, *supra* note 3, § 13.03[F] at 13-78.26 ("In many software cases, access is either conceded or easily proved, so that a finding of infringement turns entirely on whether the works are substantially similar.").

74. 17 U.S.C. § 102(b) (1988).

75. The term "innovation policy" is used to denote the best mix of legal incentives that would maximize the total value of new software inventions.

A. *Whelan* and Its Progeny

Whelan Associates v. Jaslow Dental Laboratory represents the broadest approach to protecting computer software. In *Whelan*, the court examined the idea/expression dichotomy[76] stated in § 102 and concluded that "the purpose or function of a utilitarian work would be the work's idea, and everything that is not necessary to that purpose or function would be part of the expression of the idea."[77] Since this rule forces the court to focus on *one* idea behind the program, courts applying this test will necessarily define an idea which represents an extremely high level of abstraction. For example, the *Whelan* court characterized the idea behind the plaintiff's program as the efficient operation of a dental laboratory.[78] At this high level of abstraction, there are of course many ways to write a program which performs that general function and all program elements at lower levels of abstraction would constitute copyrightable expression. As a result, the *Whelan* test protects the "structure, sequence, and organization" of source code as a general rule.

---

76. The limitations expressed in § 102(b) create what is known as the "idea/expression dichotomy" in copyright law:

> The crucial consideration in the analysis that follows is that copyright law protects only an author's original expression, not ideas or elements taken from preexisting works. Infringement is shown by a substantial similarity of *protectable expression*, not just an overall similarity between the works. Thus, before evaluating substantial similarity, it is necessary to eliminate from consideration those elements of a program that are not protected by copyright.

NIMMER & NIMMER, *supra* note 3, § 13.03[F] at 13-78.28 to .29.

77. Whelan Assocs. v. Jaslow Dental Lab., 797 F.2d 1222, 1236 (3d Cir. 1986), *cert. denied*, 479 U.S. 1031 (1987).

78. *Id.* at 1238 n.34. Other cases applying the *Whelan* test have provided wide-ranging software protection either by broadly defining the "idea" behind the plaintiff's program or by finding that "other ways" exist to express a particular idea. *See, e.g.*, Johnson Controls v. Phoenix Control Sys., 886 F.2d 1173 (9th Cir. 1989) (general finding that "the structure of the JC-5000S [plaintiff's entire program] is expression, rather than an idea in itself," apparently because "each individual application is customized to the needs of the purchaser. This practice of adaptation is one indication that there may be room for individualized expression in the accomplishment of common functions."); Lotus Dev. Corp. v. Paperback Software Int'l, 740 F. Supp. 37, 65-66 (D. Mass. 1990) (repeatedly asking whether there were other ways to express the idea of "an electronic spreadsheet"); Pearl Sys. v. Competition Elec., 8 U.S.P.Q.2d (BNA) 1520, 1524 (S.D. Fla. 1988) (defining the idea behind two subroutines as providing "a method for the user to set a par time" and as allowing "the user to review the shots he or she has fired and to learn of the time that elapsed between each shot"); Digital Communications v. Softklone Distrib., 659 F. Supp. 449, 459 (N.D. Ga. 1987) ("The use of a screen to reflect the status of the program is an 'idea'; the use of a command driven program is an 'idea'; and the typing of two symbols to activate a specific command is an 'idea.' "); Broderbund Software, Inc. v. Unison World, 648 F. Supp 1127, 1133 (N.D. Cal. 1986) (defining the idea of the plaintiff's program as "the creation of greeting cards, banner, posters and signs that contain infinitely variable combinations of text, graphics, and borders").

Courts applying *Whelan* to object-oriented software are likely to protect the basic inheritance relationships among objects. For example, the idea behind the QuadWorld program could be expressed as a program to allow for the efficient drawing of quadrilaterals on a computer screen. Since there are undoubtedly many ways to write such a program, the particular choice of classes, sub-classes, and messages is copyrightable expression. In fact, the court could argue that the idea behind QuadWorld could be achieved using traditional programming techniques, and since this would produce entirely different looking source code from the object-oriented version, that variation alone proves the necessary range of expression to justify copyright protection.[79]

Moreover, the court could use our analysis of the design process to argue that high level inheritance relationships and class structures must be protected by copyright. In *Whelan,* the court justified the protection of structure, sequence, and organization in part on the basis that "among the more significant costs in computer programming are those attributable to developing the structure and logic of the program. The rule proposed here, which allows copyright protection beyond the literal computer code, would provide the proper incentive for programmers by protecting their most valuable efforts."[80] While the validity of this argument is highly doubtful in light of the Supreme Court's recent decision in *Feist,*[81] lower courts may still be tempted to protect those parts of the plaintiff's software that are the products of significant time and effort. Under our analysis of the object-oriented design process, steps 2 and 3, identifying the structure and semantics of the objects and classes and identifying the relationships among these objects and classes, represent the most difficult parts of the design process.[82] The decisions made in those steps are critical to the quality of the final program.[83] Thus, a court could use this

79. This argument is arguably analogous to one proposed in *Lotus.* The *Lotus* court held that the differences between the user interface for Microsoft Excel on the Macintosh and the user interface for the Lotus program were evidence that there are multiple ways to express the idea of an electronic spreadsheet. *Lotus,* 740 F. Supp. at 65-66. The court was oblivious to the fact that Excel's user interface was entirely attributable to the Macintosh operating system (all programs running on the Macintosh have that same interface) and had nothing to do with how Excel chose to express the idea behind an electronic spreadsheet.

80. *Whelan,* 797 F.2d at 1237.

81. Feist Publications v. Rural Tel. Serv., Inc., 111 S. Ct. 1282, 1290 (1991) (repudiating "sweat of the brow" theories for copyright protection because "the primary objective of copyright is not to reward the labor of authors, but '[t]o promote the Progress of Science and useful Arts' ").

82. *See supra* parts III.C.2, III.C.3.

83. While the iterative nature of object-oriented development encourages refinement of the decisions made in steps 2 and 3, the court will see only the final program and thus will not be able to determine which decisions were initially made during the first pass through the design process and which were added later by refining the decisions made in steps 2 and 3. As a result, the references to steps 2 and 3 in this discussion include all decisions

argument to protect the general inheritance relationships between classes, the detailed scripts for each object, and the collections of services each object is expected to provide.

The *Whelan* court's analysis of § 102(b) has been heavily criticized by commentators,[84] and for good reason. The primary criticism of *Whelan* has focused on *Whelan's* use of a *single* idea existing in each computer program:

> The crucial flaw in [*Whelan's*] reasoning is that it assumes only one "idea," in copyright law terms, underlies any computer program, and that once a separable idea can be identified, everything else must be expression. All computer programs are intended to cause the computer to perform some function. The broad purpose that the program serves, be it managing a dental laboratory, automating a factory, or dispensing cash at a bank teller machine, is *an* idea. Other elements of the program's structure and design, however, may also constitute ideas for copyright purposes.[85]

Similarly, in *Computer Associates v. Altai,* a district court adopted this criticism and then used the traditional model of programming to further reveal *Whelan's* flaws:

> In the case at bar, Dr. Davis [court-appointed expert] pointed out further technical flaws in the *Whelan* analysis which render its reasoning inadequate. As he so convincingly demonstrated, a computer program is made up of sub-programs and sub-sub-programs, and so on. Each of those programs and sub-programs has at least one idea. Some of them could be separately copyrightable; but many of them are so standard or routine in the computer field as

---

that fall within the general subject matter of those steps whether or not they were actually made in those steps or at a later time.

84. *See, e.g.,* NIMMER & NIMMER, *supra* note 3, § 13.03[F] at 13-78.33 to .34; Richard A. Beutel, *Software Engineering Practices and the Idea/Expression Dichotomy: Can Structured Design Methodologies Define the Scope of Software Copyright,* 32 JURIMETRICS J. 1, 17-20 (1991); Nimmer et al., *supra* note 3, at 629-30, 639; Reback & Hayes, *supra* note 3, at 3-4. Several cases have also rejected *Whelan* or have recognized its existence but then implicitly failed to apply it. *See* Computer Assocs. Int'l v. Altai, Inc., 982 F.2d 693, 706 (2d Cir. 1992) ("We think that *Whelan's* approach to separating idea from expression in computer programs relies too heavily on metaphysical distinctions and does not place enough emphasis on practical considerations."); Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1524 (9th Cir. 1992) ("The *Whelan* rule, however, has been widely—and soundly—criticized as simplistic and overbroad."); Plains Cotton Co-op Ass'n v. Goodpasture Computer Serv., Inc., 807 F.2d 1256, 1262 (5th Cir. 1987) (declining "to embrace *Whelan*"), *cert. denied,* 484 U.S. 821 (1987); Computer Assocs. Int'l v. Altai, Inc., 775 F. Supp. 544, 558-59 (E.D.N.Y. 1991) (describing *Whelan* as setting "forth what now seems to be a simplistic test for similarity between computer programs"), *aff'd in relevant part,* 982 F.2d 693 (2d Cir. 1992); Manufacturers Technologies, Inc. v. CAMS, Inc., 706 F. Supp. 984, 992 (D. Conn. 1989) (not explicitly rejecting *Whelan* but arguing that the *Broderbund* court's application of *Whelan* to screen displays, "overextended the scope of copyright protection applicable to those screen displays"); Healthcare Affiliated Servs., Inc. v. Lippany, 701 F. Supp. 1142 (W.D. Pa. 1988).

85. NIMMER & NIMMER, *supra* note 3, § 13.03[F] at 13-78.33 to .34.

> to be almost automatic statements or instructions written into a program.[86]

Despite this criticism, *Whelan* has never been overruled and is still the starting point for most discussions of copyright protection for computer software.

B. The Filtering Approach[87]

In contrast to *Whelan's* "one idea" approach, Nimmer starts with the "patterns of abstractions" test[88] and concludes that the court must apply a series of standard copyright doctrines to filter out unprotectable ideas at *each* level of abstraction. This test is easy to defend because each filter is closely tied to a specific copyright doctrine and thus forces the court to account for every theory that can limit the number of program elements entitled to protection. Nimmer proposes that the court apply four basic filters: abstract ideas, merger, *scenes a faire,* and public domain. While Nimmer's test was closely tied to the traditional model of software development, it can still be applied to object-oriented software by altering the relative importance of each filter.

1. *ABSTRACT IDEAS*

Nimmer's first filter revisits the basic problem of separating protectable ideas from non-protectable expression. In the context of traditional software, this filter provides a strong limit on copyright protection because the top-down approach to software development "provides natural divisions, which may correspond to the various levels of abstractions that the court seeks to identify and analyze."[89] In Nimmer's view, the court can divide the software into programs, sub-

---

86. *Computer Assocs.*, 775 F. Supp. at 559.

87. This approach was first developed in Nimmer et al., *supra* note 3, at 635-55, and is summarized in NIMMER & NIMMER, *supra* note 3, at § 13.03[F]. The Second Circuit has recently endorsed this approach to substantial similarity. *Computer Assocs.*, 982 F.2d at 706.

88. The test was first developed by Judge Learned Hand:

> Upon any work, and especially upon a play, a great number of patterns of increasing generality will fit equally well, as more and more of the incident is left out. The last may be no more than the most general statement of what the play is about, and at times might consist only of its title; but there is a point in this series of abstractions where they are no longer protected, since otherwise the playwright could prevent the use of his "ideas," to which, apart from their expression, his property is never extended.

Nichols v. Universal Pictures Corp., 45 F.2d 119, 121 (2d Cir. 1930), *cert. denied*, 282 U.S. 902 (1931).

89. Nimmer et al., *supra* note 3, at 638 ("[T]he systematic method used to develop computer programs makes the abstractions test facially more applicable to computer software than other types of works. Traditional literary works are not created in such a consistently organized and orderly fashion.").

programs, and sub-sub-programs and then determine at which level the code passes from being an unprotectable idea to being protectable expression.

Two problems bar meaningful application of this filter to object-oriented software. First, Nimmer himself admitted that even in the context of structured top-down programming, the test is not easy to apply.[90] As one commentator complained, "simply to characterize the filter as eliminating 'abstract ideas' says very little about what is, and is not, an 'idea.' One man's 'abstract idea' may be another's protectable expression."[91] Second, the iterative nature of object-oriented development prevents the court from finding easy lines to draw in determining what is a "level of abstraction." The process of "round-trip gestalt design" will tend to blur meaningful line drawing on the basis of the design process itself.

As an alternative, we could define the levels of abstraction by considering class lists, inheritance relationships, and the semantic data model to each be separate levels of abstraction. However, these lines create extremely broad categories which may encourage the court to find the same single idea behind each level of abstraction. For example, a court examining the traffic light problem might conclude that the idea behind the list of classes is the "efficient management of a traffic intersection." But, if the court then examines the inheritance structure and semantic data model, it seems that the idea at those levels of abstraction is also the efficient management of a traffic intersection. In the context of object-oriented software, this alternative leads courts back to the heavily criticized "one idea" approach of *Whelan*.

2. *MERGER*

The merger filter operates to exclude elements of the program that can only be expressed in one way.[92] In the context of computer software, "merger issues may arise in somewhat unusual ways. Although theoretically many ways may exist to implement a particular idea, efficiency concerns can make one or two choices so compelling as to virtually eliminate any form of expression."[93] In this category, Nimmer lists such low-level routines as searching and sorting algorithms, which should not be protected, because "the fact that two programs both use the most efficient sorting or searching method available supports an inference of independent creation as readily as it supports one of copying, and thus

90. NIMMER & NIMMER, *supra* note 3, at 13-78.33.
91. Beutel, *supra* note 84, at 23.
92. NIMMER & NIMMER, *supra* note 3, at 13-78.35.
93. *Id.*

is not reliable evidence that copying occurred."[94] These considerations have finally received explicit judicial recognition by the Ninth Circuit which has expressed the impact of merger even more broadly than Nimmer:

> To the extent that there are many possible ways of accomplishing a given task or fulfilling a particular market demand, the programmer's choice of program structure and design may be highly creative and idiosyncratic. However, computer programs are, in essence, utilitarian articles—articles that accomplish tasks. As such, they contain many logical structural, and visual display elements that are dictated by the function to be performed, by considerations of efficiency, or by external factors such as compatibility requirements and industry demands.[95]

For most merger issues, object-oriented software can be analyzed in the same manner as traditional software. Sorting and searching routines would be used primarily by the *internal* implementation of a specific object's methods. Since this internal implementation may itself have been written using traditional structural programming techniques, courts should be able to apply this test without alteration. Similarly, other merger concerns, such as ensuring compatibility with particular hardware and software, should not raise issues unique to object-oriented software.[96] In general, courts should find the merger filter to be a powerful tool for limiting infringement claims relating to the internal implementations of specific objects.[97]

---

94. *Id* at 13-78.36. It must be emphasized that copyright law does not prevent a defendant from producing a substantially similar program, as long as the defendant did not actually copy the plaintiff's work. Copying is an absolute prerequisite for infringement, and the analysis of substantial similarity is used only to raise the inference of copying because direct evidence of copying rarely exists. *See, e.g., Computer Assocs.*, 982 F.2d at 708 ("Since, as we have already noted, there may be only a limited number of efficient implementations for any given task, it is quite possible that multiple programmers, working independently, will design the identical method employed in the allegedly infringing work. Of course, if this is the case, there is no copyright infringement.").

95. Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1524 (9th Cir. 1992); *see also Computer Assocs.*, 982 F.2d at 708 ("[W]hen one considers the fact that programmers generally strive to create programs 'that meet the user's needs in the most efficient manner,' the applicability of the merger doctrine to computer programs becomes compelling. . . . [T]he more efficient a set of modules are, the more closely they approximate the idea or process embodied in that particular aspect of the program's structure." (quoting Menell, *supra* note 3, at 1052; citation omitted)).

96. *See Sega*, 977 F.2d at 1526 (allowing intermediate copying in order to ensure compatibility with videogame hardware); Atari Games Corp. v. Nintendo of Am., Inc., 975 F.2d 832 (Fed. Cir. 1992) (same).

97. Merger analysis should not be used when evaluating semantic data models or the general structure of particular object classes and sub-classes. For most sophisticated and complex programs, it is highly unlikely that only one efficient object-oriented structure exists. The analysis of semantic data models is better addressed by the *scenes a faire* doctrine discussed in the next sub-section.

3. *SCENES A FAIRE*

Scenes a faire represents the most powerful filter for object-oriented software. Nimmer used the term to justify excluding program elements dictated by "external considerations," such as hardware standards, software standards, computer manufacturers' design standards, target industry practices, and computer industry programming practices.[98] While these considerations can certainly be applied to object-oriented software, traditional case law dealing with *scenes a faire* will actually be more important in eliminating elements of object-oriented software from copyright protection.

Under the *scenes a faire* doctrine, copyright protection is denied for "those elements that follow naturally from the work's theme rather than from the work's creativity."[99] In the literary context, *scenes a faire* has precluded protection for stock literary devices or stock character types that are inherent in the general theme of the work.[100] For example, in *Shaw v. Lindheim*,[101] the court examined the mood, setting, and pace of the plaintiff's and defendant's television scripts and concluded that "[b]oth works are fast-paced, have ominous and cynical moods that are lightened by the [hero's] victory, and are set in large cities. These similarities are common to any action adventure series, however, and do not weigh heavily in our decision."[102]

Particularly in the case of software designed to model real-world behavior, this approach to *scenes a faire* justifies excluding from protection software elements that are dictated by the real-world behavior being modeled. This understanding of the doctrine has already been accepted by several courts evaluating traditional software. For example, in *Data East USA, Inc. v. Epyx, Inc.*,[103] the Ninth Circuit analyzed two computer karate games and concluded that infringement could not be based on program elements that "encompass the idea of karate."[104] In doing so, the court approved of the district court's finding that:

> [T]he visual depiction of karate matches is subject to the constraints inherent in the sport of karate itself. The number of combatants, the stance employed by the combatants, established and recognized moves and motions regularly employed in the sport of karate, the regulation of the match by at least one referee or judge, and the manner of scoring by points and half points are among the constraints inherent in the sport of karate. Because of these

98. NIMMER & NIMMER, *supra* note 3, § 13.03[F] at 13-78.36 to .43.

99. Nimmer et al., *supra* note 3, at 642.

100. *See* Hoehling v. Universal City Studios, Inc., 618 F.2d 972 (2d Cir. 1980), *cert. denied*, 449 U.S. 841 (1980).

101. 919 F.2d 1353 (9th Cir. 1990).

102. *Id.* at 1363.

103. 862 F.2d 204 (9th Cir. 1988).

104. *Id.* at 209.

> constraints, karate is not susceptible of a wholly fanciful presentation.[105]

Similarly, in *Plains Cotton Co-op Ass'n v. Goodpasture Computer Serv., Inc.*,[106] the Fifth Circuit refused to find infringement because the "appellees presented evidence that many of the similarities between the GEMS and Telcot programs are dictated by the externalities of the cotton market."[107] As a result, the plaintiff could not claim protection for program elements that were designed to imitate a "cotton recap sheet," because that was a stock element in the real-world cotton market and necessary to any program trying to model that market.[108] Finally, in *Q-Co Industries, Inc. v. Hoffman*,[109] the court examined two tele-prompting programs and found no protectable expression because "the same modules would be an inherent part of any prompting program. Their order and organization can be more closely analogized to the concept of wheels for the car rather than the intricacies of a particular suspension system."[110]

These cases provide strong authority for excluding many of the object-oriented elements in a program that models real-world behavior. For example, in QuadWorld, the entire class and inheritance structure flows directly from the natural relationships between squares, rectangles, parallelograms, and quadrilaterals which, in turn, are dictated by formal mathematical definitions in the real world. Similarly, in the traffic light control program, nothing in the semantic data model would be protectable because these relationships are dictated by the functional behavior of trip sensors, controllers, and traffic lights. Finally, the relevance of *scenes a faire* to object-oriented software is further underscored by our approach to design in step 2, in which we wrote "scripts" for each object, making it fairly easy for a court to compare each object to a "stock character" in the real-world system being modeled.

In most situations, the list of services that each object must provide will be largely dictated by these scripts and hence will be unprotectable. In certain cases, it might be possible to identify certain low-level objects[111]

---

105. *Id.*

106. 807 F.2d 1256 (5th Cir. 1987), *cert. denied*, 484 U.S. 821 (1987).

107. *Id.* at 1262.

108. *Id.* at 1262 n.4.

109. 625 F. Supp. 608 (S.D.N.Y. 1985)

110. *Id.* at 616 (citation omitted).

111. By low-level objects, I mean certain objects which are simply building blocks in constructing more complex objects which model real-world properties. At this low level, the building block may be sufficiently removed from real-world behavior to render *scenes a faire* inapplicable. However, even these objects may often be taken from libraries of reusable objects and should be excluded from protection because they do not satisfy copyright's originality requirement. *See infra* Section IV.B.4 discussing the public domain filter.

that do not directly model real-world behavior and could therefore escape the *scenes a faire* filter. In general, however, the only elements that will survive this filter are low-level implementations of specific methods; at that level, those portions of code resemble traditional programs and embody few object-oriented principles.

4. *PUBLIC DOMAIN*

The "public domain" filter will also be extremely important in analyzing object-oriented software. Since object-oriented design focuses on reusable software components, many complex object-oriented programs will take advantage of existing objects that have been written for other programs. In some cases, these objects may be taken from public domain libraries, such as those provided on electronic bulletin boards. As Nimmer notes, "It is axiomatic that material in the public domain is not protected by copyright even when incorporated into a copyrighted work."[112] As a result, the court must eliminate any objects taken from public domain when determining which elements of the program are protectable.

However, the bulk of reusable objects may not come from entirely "public" sources. These reusable objects may come from vendors selling libraries of pre-defined objects on a license basis, particularly in the case of graphical user interfaces and database systems. These vendors clearly intend that their libraries will be incorporated into commercial products.[113] Nonetheless, these objects should not be included in the scope of the copyright protection for the final commercial product, as they would not be original to the programmer claiming authorship of the final product, and hence could not pass copyright's threshold test for originality.[114] As a result, the court must treat the use of licensed objects

---

112. NIMMER & NIMMER, *supra* note 3, at 13-78.43 (citing Sheldon v. Metro-Goldwyn Pictures Corp., 81 F.2d 49, 54 (2d Cir. 1936), *aff'd*, 309 U.S. 390 (1940)); *see also* Computer Assocs. Int'l v. Altai, Inc., 982 F.2d 693, 710 (2d Cir. 1992) (public domain "material is free for the taking and cannot be appropriated by a single author even though it is included in a copyrighted work").

113. In addition to the previously discussed object libraries for implementing the Macintosh user interface, see *supra* note 6, vendors are hawking a wide variety of object libraries for use in commercial applications. A quick perusal of advertisements and articles in any programming trade magazine will confirm the growth of this industry. *See, e.g.*, COMM. ACM, Oct. 1991.

114. Copyright protection is allowed only for *original* works of authorship. 17 U.S.C. § 102(a) (1988). At a minimum, original authorship means that the programmer did not directly take the expression from any other source, whether public or not. Feist Publications v. Rural Tel. Serv., Inc., 111 S. Ct. 1282, 1287 (1991) ("The sine qua non of copyright is originality. To qualify for copyright protection, a work must be original to the author. Original, as the term is used in copyright, means only that the work was independently created by the author (as opposed to copied from other works), and that it possesses at least some minimal degree of creativity." (citation omitted)).

on the same basis as truly "public domain" objects. In both cases, copyright protection would not be available for any object which the programmer seeking protection did not write.

C. Economic Balancing Approach

While no court has yet adopted the economic balancing approach, several recent commentators on software protection have suggested answering the idea/expression problem by balancing the copyright plaintiff's creative contribution against the loss to society from granting the plaintiff a monopoly over particular software code. In one version of this approach, the court would determine the existence of protectable expression by following a two-step test:

> The first step is for the court to define as specifically as possible the thing that the defendant has taken from the plaintiff . . . the second step is to decide whether that thing is original to the plaintiff . . . . That is, to get that thing the defendant took, did the plaintiff invest costly creative effort that presumptively relied on the promise of copyright? If so, judgment properly goes to the plaintiff, because, in conclusory terms, the defendant has taken the plaintiff's expression. Or did the plaintiff get that thing by copying it effortlessly from existing and available sources, or by otherwise responding entirely to incentives other than copyright? If so, judgment properly goes to the defendant because, again stating it in conclusory terms, the defendant took only the plaintiff's idea.[115]

In another version of the economic balancing approach, the court would divide the plaintiff's program at different levels of abstraction and then determine the dividing line between idea and expression by "balancing the need to provide an incentive to authors against the cost to society of losing the free use of the author's work at that level of expression."[116]

While the economic balancing approach seems intriguing as a matter of innovation policy, courts are not likely to endorse it primarily because it is not supported by copyright doctrine. Both versions require the court to parse the plaintiff's work in a manner similar to the "patterns of abstractions" test first articulated in *Nichols v. Universal Pictures Corp.*[117] However, both versions ultimately depart from copyright doctrine by requiring the court to balance the economic return necessary to induce the author to produce a particular type of work against the cost to society of granting that author a monopoly over particular expression at a particular level of abstraction. This equation confuses the distinctions between copyright and patent law. In patent law, the author's creative

---

115. Wiley, *Copyright at the School of Patent*, 58 U. CHI. L. REV. 119, 158-59 (1991).
116. Reback & Hayes, *supra* note 3, at 5.
117. 45 F.2d 119, 121 (2d Cir. 1930), *cert. denied*, 282 U.S. 902 (1931).

contribution is assessed by the requirements of utility, novelty, and non-obviousness.[118] The cost to society is controlled by requiring the inventor to define the invention with specific claim language sufficiently narrow to avoid the prior art and by requiring that those claims be supported by the specification, thus ensuring that most patents will have a fairly narrow scope. Moreover, the costs of protection are offset by the societal benefits resulting from full disclosure of the underlying technology in the patent specification.

In contrast, copyright asks little of the author except that the work not be copied from any other source and that the work reflect at least minimal creativity.[119] While copyright law limits the monopoly costs to society by allowing independent creation to be an absolute defense to infringement, it provides no doctrinal tools for defining the scope of the monopoly against potential defendants who have had access to the work. As a result, the economic approach is difficult to support with copyright doctrine. In fact, the author of the first version acknowledged this dilemma and explicitly developed his test by applying the "good sense" of patent doctrine in order to "rationalize" copyright doctrine.[120]

Even if the economic approach could somehow be justified under traditional copyright doctrine, it is not clear that the economic approach would be particularly desirable as a matter of policy. Under the first version, the court would face the elusive task of *objectively* determining whether the plaintiff would have authored the code appropriated by the defendant in the absence of copyright incentives. Beyond the obvious evidentiary problems in this analysis, the process of analyzing incentives *ex post* leads unavoidably to circular reasoning. Whether the plaintiff was motivated by the promise of copyright depends to a large degree on the generally perceived rule of law regarding the scope of software copyrights. But, at the same time, the purpose of the two-part test itself is to determine the proper scope of the idea/expression dichotomy and hence announce a new rule of law. This dilemma is further exacerbated by the small number of software copyright cases that result in published decisions.

The second version of the economic approach faces similar problems. First, the *ad hoc* nature of the inquiry makes it difficult for

---

118. *See* 35 U.S.C. §§ 101-103 (1988).

119. While the *Feist* case may have raised the standard of originality required by copyright, it did not raise that standard anywhere close to requiring an analysis of creative contributions.

120. Wiley, *supra* note 115, at 120 ("Using an economic perspective on innovation policy, this Part defends the notion that we should regard core portions of patent doctrine as intellectual successes worthy of imitation. Most fundamentally, patent law establishes a set of sensible and efficient *incentives* to creation. Copyright should learn this basic lesson, for a focus on sound incentives would give copyright doctrine the coherence it now lacks.").

software companies to make rational business decisions based on which aspects of their own software and their competitor's software are protectable in copyright.[121] Moreover, the formulation of the test suggests a false empiricism. Even for software products aimed at mature business markets, it will be extremely difficult to determine the "cost" to society of granting the plaintiff a monopoly. More fundamentally, even if this cost could be accurately calculated, it must be balanced against the purely speculative "creative contribution" of the author, which inevitably invites judgments which are nothing more than a determination that the plaintiff's work is novel, and non-obvious, and therefore worthy of protection. This type of analysis is better left to patent law, where more precise standards exist for determining non-obviousness and where the court has the benefit of an initial expert analysis performed by the patent examiner.[122]

D. Copyright Doctrine Properly Applied Provides Little Protection for Object-Oriented Software

While Nimmer's filtering test is closely linked to traditional copyright doctrine, it may present an unnecessary exercise in the case of object-oriented software. As a practical matter, Nimmer's filters will exclude from protection nearly every element that makes a particular program object-oriented in design. More important, these elements are precisely the elements which reveal software's "behavioral" rather than "textual" nature and which render object-oriented programs generally unsuitable for copyright protection.

While most courts recognize that computer programs are utilitarian articles, most infringement cases require the court to analyze only the textual representation of the program's structure, sequence, and organization as embodied in source code. As a result, most courts focus on the textual embodiment of software and quickly lose sight of the behavioral nature of software. This distinction was first recognized in *Computer Associates International v. Altai, Inc.*,[123] in which the district court adopted the findings of a court-appointed expert who explained that:

> a computer program must be viewed both as text and as behavior. The text perspective focuses upon the object code and source code .... A computer program, however, is more than a collection of zeros and ones. When properly loaded into a computer and

---

121. Beutel, *supra* note 84, at 27 (noting that "[j]ust as the application of the antitrust 'rule of reason' has taken years of dissection and analysis to take form, so too would the eventual parameters of software copyright under the policy-balancing approach set forth in the Reback/Hayes Abstractions Test").

122. Critics of software patents often question the competence of software examiners in analyzing software issues. This problem is discussed further *infra* Part V.

123. 775 F. Supp. 544 (E.D.N.Y. 1991), *aff'd in relevant part*, 782 F.2d 693 (2d Cir. 1992).

> provided with appropriate input from, for example, the keyboard, the program behaves. In a word processing program, for example, text can be deleted, blocks of text can be moved, formatting of documents can be changed; all sorts of operations can be instituted; and these can only be described as behavior.[124]

While the court used this analysis primarily to criticize *Whelan* for failing to distinguish between the static, textual view of the program and the dynamic, behavioral view,[125] the court also recognized that the behavioral aspect of software creates a much more fundamental problem when viewed against the statutory limits imposed by § 102(b):[126]

> Going beyond Dr. Davis' analysis, the court notes a possible statutory difficulty that arises when we recognize, as we must, that a computer program "behaves." . . . Since the behavior aspect of a computer program falls within the statutory terms "process", "system", and "method of operation", it may be excluded by statute from copyright protection. . . . Fortunately, this court need not wrestle with that possible development in the law of intellectual property, because CA's rights in this case are fully protected by viewing the ADAPTER program as text.[127]

Although the *Computer Associates* court did not have to resolve this question, courts dealing with object-oriented software must address the behavioral nature of software. Courts will be confronted with this problem from three different angles. First, if courts approach object-oriented programs as they would approach programs written under the traditional model, they will find that the structure, sequence, and organization of the source code tell us little about the inheritance relationships and class structures in the program, which may be the part of the program that the plaintiff most wants to protect. Moreover, the closer the plaintiff adhered to the object-oriented model in the program's creation, the more pronounced this phenomenon will be. In fact, since programs that make good use of polymorphism and dynamic binding must include source code that is highly generalized, the source code behind the best written programs will tell us the least about the objects in the program.

Faced with this problem, the court will then have to focus on the specific class definitions used to specify inheritance relationships, messages accepted, and method implementations. However, while these definitions are expressed in the English words used by a particular programming language, they are simply a shorthand description of a

---

124. *Id.* at 559.

125. *Id.* at 560.

126. 17 U.S.C. § 102(b) provides: "In no case does copyright protection for an original work of authorship extend to any idea, procedure, process, system, method of operation, concept, principle, or discovery, regardless of the form in which it is described, explained, illustrated, or embodied in such work."

127. *Computer Assocs.*, 775 F. Supp. at 560.

highly specific system. When we the define the class, "rectangle" as a sub-class of the class "parallelogram," there is nothing expressive, in the copyright sense, about that definition. The term "sub-class" is a shorthand instruction that tells the compiler "whenever you see a rectangle, have it behave just like a parallelogram, except when you receive a message which has been overridden in the definition of the rectangle class, then use a different behavior."

Third, the court will have to approach the program by examining the high-level relationships among different classes and objects because the textual descriptions of a particular class cannot be protected, . In fact, the court may well be tempted to examine substantial similarity by asking the parties to create a semantic data model of each program, on the theory that if the semantic data models are substantially similar, the programs must be substantially similar. However, this approach effectively creates copyright protection for semantic data models themselves, a result which cannot be justified under fundamental copyright principles. True, a programmer who draws a semantic data model can claim a copyright in the pictorial representation that the programmer used to express the model; that programmer can prevent others from copying the *picture*. However, the copyright in the picture cannot be used to indirectly grant protection over the model itself, a result which follows directly from *Baker v. Selden*:[128]

> The copyright of a work on mathematical science cannot give to the author an exclusive right to the methods of operation which he propounds, or to the diagrams which he employs to explain them, so as to prevent an engineer from using them whenever occasion requires.[129]

At a more fundamental level, the semantic data model cannot be used to determine infringement because it is simply a list of the constituent elements of a particular system. Copyright protection cannot be used to provide a monopoly over these elements, a point which was

---

128. 101 U.S. 99 (1879) (holding that copyright can reside in a particular explanation of a system, but not in the system itself). *Baker* is generally regarded as the inspiration for § 102(b). *See* Amicus Curiae Brief of Copyright Law Professors at 5, Lotus Dev. Corp. v. Borland Int'l, Inc., 799 F. Supp. 203 (D. Mass. 1992) (No. 90-1162-K) [hereinafter Copyright Professors' Amicus Brief] ("It is to cases such as Baker v. Selden and its progeny that courts should look in interpreting section 102(b) and its exclusion of systems and methods from the scope of copyright protection available to works of authorship." (citation omitted)).

129. *Baker*, 101 U.S. at 103; *see also* Copyright Professors' Amicus Brief, *supra* note 128, at 6 n.3 ("[T]he [*Baker*] Court pointed out that in most instances, useful arts were embodied in wood, metal, or stone, and what had given plausibility to Selden's claim was that his useful art was embodied in a writing. Nevertheless, the Court stated 'the principle is the same in all. The description of the art in a book, though entitled to the benefit of copyright, lays no foundation for an exclusive claim to the art itself.' " (quoting *Baker*, 101 U.S. at 105)).

recently restated in an amicus curiae brief submitted by eleven well-respected copyright professors in *Lotus v. Borland*:[130]

> It is in the nature of a method or system to have constituent elements, some of which may be quite detailed in character. In the "Shorthand cases," courts will decline to extend copyright protection not only to the set of abstract rules that a shorthand system developer might have devised for condensing words or phrases, but also to the vocabulary resulting from the implementation of these rules. Both are constituent elements of the system which copyright law will not protect.[131]

This analysis can be directly applied to a semantic data model. For example, in the traffic light program, the semantic data model tells us "a clock is a part of a controller, and a controller reads from a sensor which can be either a pressure sensor or a trip sensor." This semantic data model equally describes the real-world physical system and the system for modeling that behavior on a computer. Just as the traffic light and controller are constituent elements in the real-world traffic intersection, the representations of those entities as objects and classes are constituent elements of a *system* for modeling the behavior of a traffic intersection on a computer.

Finally, the semantic data model is exactly what its title implies, an attempt to explain a detailed system in words and pictures. The plaintiff presenting a semantic data model as the basis for proving substantial similarity is not arguing that the defendant used the same words and pictures to depict the system, but rather that the defendant used the same *system* of classes and inheritance relationships in writing the allegedly infringing program. As soon as the plaintiff presents a semantic data model as the basis for infringement, the court must recognize that the plaintiff is seeking protection for the constituent elements of a particular object-oriented system, a right which has no basis in copyright law.

The preceding analysis shows that copyright law does not protect the high-level relationships among objects. The fact that these relationships may represent the bulk of the programmer's effort and innovation during design is irrelevant in determining the scope of protection under copyright doctrine.[132] If protection for such behavioral elements in object-oriented software is available, it can only be achieved through the patent system.[133]

---

130. Lotus Dev. Corp. v. Borland Int'l, Inc., 799 F. Supp. 203 (D. Mass. 1992) (granting partial summary judgment).

131. Copyright Professors' Amicus Brief, *supra* note 128, at 7.

132. *See supra* note 81.

133. *See Baker*, 101 U.S. at 105 ("The description of the art in a book, though entitled to the benefit of copyright, lays no foundation for an exclusive claim to the art itself. The object of the one is explanation; the object of the other use. The former may be secured by copyright. The latter can only be secured, if it can be secured at all, by letters-patent.");

V. PATENT PROTECTION FOR OBJECT-ORIENTED SOFTWARE

In some ways, the patentability of object-oriented software is easier to analyze than the patentability of traditional software. In cases involving traditional software, the primary question has been whether the software recites "a mathematical algorithm."[134] If it does, then the software is not patentable; otherwise the software is patentable subject matter.[135] Indeed, this analysis would still apply to a patent claim that was drawn to the low-level internal implementation of a specific method in an object-oriented program, since that portion of the program operates on the same principles as traditional software. In that case, the court would also have the benefit of examining the claim in light of a substantial body of critical commentary that has been written on the patentability of traditional software.[136] The more interesting question is what higher-level elements of the object-oriented model could qualify as patentable subject matter.

A. Patentable Subject Matter

The most promising candidate for protection is a patent claim drawn to a semantic data model. In fact, a purely textual description of a semantic data model would read very much like a standard apparatus claim. In the case of the traffic light example, we could construct a patent

---

Computer Assocs. Int'l v. Altai, Inc., 775 F. Supp. 544, 560 (E.D.N.Y. 1991) (noting in the context of the problems raised by the behavioral aspects of software, "indeed, it has been suggested that computer software is better protected by patent law than by copyright law"), *aff'd in relevant part*, 982 F.2d 693 (2d Cir. 1992).

134. Gottschalk v. Benson, 409 U.S. 63 (1972).

135. The Federal Circuit and its predecessor courts have devised a two-part test, the "Freeman-Walter" test, to determine whether a particular software claim is drawn to patentable subject matter. In the first step, the court must determine whether the claim directly or indirectly recites a mathematical algorithm. If it does not, then the claim is drawn to patentable subject matter. However, even if the claim does recite a mathematical algorithm, it may still be patentable if the claim "implement[s] the algorithm in a specific manner to define structural relationships between the elements of the claim in the case of apparatus claims, or limit or refine physical process steps in the case of process or method claims." *In re* Walter, 618 F.2d 758, 767 (C.C.P.A. 1980). *See generally In re* Iwahashi, 888 F.2d 1370 (Fed. Cir. 1989); *In re* Pardo, 684 F.2d 912 (C.C.P.A. 1982); *In re* Abele, 684 F.2d 902 (C.C.P.A. 1982); *In re* Freeman, 573 F.2d 1237 (C.C.P.A. 1978); *PTO Report On Patentable Subject Matter: Mathematical Algorithms and Computer Programs*, 38 Pat. Trademark & Copyright J. (BNA) 563 (1989) [hereinafter *PTO Report*].

136. *See, e.g.*, Donald S. Chisum, *The Patentability of Algorithms*, 47 U. PITT. L. REV. 959 (1986); Pamela Samuelson, Benson *Revisited: The Case Against Patent Protection for Algorithms and Other Computer Program-Related Inventions*, 39 EMORY L.J. 1025 (1990); Randall M. Whitmeyer, Comment, *A Plea for Due Processes: Defining the Proper Scope of Patent Protection for Computer Software*, 85 NW. U. L. REV. 1103 (1991) [hereinafter Comment, *A Plea for Due Processes*].

claim for a real-life intersection control system that read something like this:

> A traffic control apparatus consisting of:
>
> a trip sensing means and a pressure sensing means, and a controller device which is operably connected to receive signals from said sensing means, and operably connected to send signals to a sequential display of different colored lights.

In the case of a real-world traffic control system, this claim would certainly recite patentable subject matter. In the case of object-oriented software, this claim is a close description of our semantic data model. Of course, in the case of software, the "trip sensing" means refers not to a physical object but to a location in the computer's memory that is designed to model the behavior of the real-world "trip sensing" means. While the case law on this issue is somewhat confused, a strong case can be made for holding that the above claim should be patentable subject matter whether it refers to the computer model of a traffic intersection or the physical apparatus used in the real world.

1. *EXISTENCE OF MATHEMATICAL ALGORITHM*

The initial inquiry for computer program related inventions focuses on the existence or absence of a mathematical algorithm.[137] If the claimed invention is drawn at the level of the semantic data model, no mathematical formulas will appear in the claim. Because the object-oriented model emphasizes encapsulation of data and procedures, the implementation of simple mathematical formulas should be hidden in the internal methods of each class and is generally invisible in the semantic data model.[138] At this point, the fact that the semantic data model may still embody a "non-mathematical" algorithm, in the broad sense, does not disqualify it from patent protection.[139]

---

137. *Iwahashi*, 888 F.2d at 1374 ("[T]he proscription against patenting has been limited to *mathematical* algorithms and abstract *mathematical* formulae which, like the laws of nature, are not patentable subject matter.") (emphasis in original); *PTO Report*, *supra* note 130, at 570 ("The major (and perhaps only) exception in the area of computer processes is the mathematical algorithm ... If a computer process claim does not contain a mathematical algorithm in the Benson sense, the second step of the Freeman-Walter-Abele test is not reached, and the claimed subject matter will usually be statutory.").

138. One might question whether the specification describing a semantic data model would be sufficiently enabling under 35 U.S.C. § 112. In most cases, the mathematical formulas necessary to construct a working program will be obvious to those skilled in the art. In those cases where the implementation is not obvious, the PTO could require the applicant to disclose those formulas in the specification, perhaps as part of the best mode requirement.

139. All apparatus claims could be considered to follow an algorithm in the broad sense of the term. *See Iwahashi*, 888 F.2d at 1375 ("[T]he fact that the apparatus operates according to an algorithm does not make it nonstatutory.").

The Federal Circuit has been quick to grant claims in which a single system of physical elements and a computer program are drafted as a single claim.[140] In such cases, "[t]he claim as a whole certainly defines [an] apparatus in the form of a combination of interrelated means and we cannot discern any logical reason why it should not be deemed statutory subject matter . . . ."[141] For many claims drawn to object-oriented programs, a strong argument can be made that the relationships between objects act much like the interaction between physical elements of a real-world apparatus in which different operational "means" send signals to one another and respond accordingly. In fact, as illustrated above by the sample claim for a traffic light control system, a single claim could equally describe the semantic data model or the real-world system itself.[142] The close identity between the description of a real-world system and the object-oriented program which models that system reinforces the argument that a claim based on the object-oriented program presents the same statutory subject matter as a conventional claim for the physical system.

Finally, while software that models or controls real world objects presents the best candidate for patentable subject matter, protection may also be available for object libraries which have no real-world counterparts, such as object-oriented graphical user interfaces and database systems. In these cases, the software still represents the interactions of various "means" designed to control the internal workings of a general purpose computer. The Federal Circuit has already

---

140. *See id.* (auto-correlation unit for use in pattern recognition); *In re* Abele, 684 F.2d 902 (C.C.P.A. 1982) (software program for improved CAT-scan process); *In re* Taner, 681 F.2d 787 (C.C.P.A. 1982) (software which improved seismic exploration by translating spherical seismic waves into plane or cylindrical waves); *In re* Freeman, 573 F.2d 1237 (C.C.P.A. 1978) (software for controlling conventional phototypesetter).

141. *Iwahashi*, 888 F.2d at 1375.

142. The fact that such a claim could be drafted also implies that a single claim would grant the inventor a monopoly over both the real-world physical system and the object-oriented model of that system, a result which undoubtedly raises alarms in certain circles. However, three considerations mitigate the danger of this result. First, the single claim will have to withstand prior art from both the computer science field and field relating to the real-world physical system. *See infra* Section V.B. Few claims will be non-obvious when tested against such a wide range of prior art. Second, the claim grants a monopoly only to the extent that the invention is enabled by the specification under the standards in 35 U.S.C. § 112. In many cases, the inventor may be able to describe how to write the object-oriented program but will be unable to explain how to actually build some of the elements in the physical system. For example, while we can easily model a "trip sensor" in a program, it may be much more difficult to build one that works consistently when embedded in a roadway. Particularly since the vast majority of claims will have to be written in "means-plus-function" form, the specification will sharply the limit the actual scope of the monopoly granted by the single claim. Finally, if the inventor has presented a single claim which is truly non-obvious *and* which enabled both the computer and physical versions of the systems, then the inventor has really *invented* both systems and should be entitled to protection over both.

recognized the patentability of pure software claims which direct the way the computer manages data internally.[143]

2. *MENTAL STEPS*

The mental steps doctrine was historically used to deny patent protection for process claims involving simple measurements, calculations, and interpretations of data that could just as easily be performed by a human using paper and pencil.[144] However, the C.C.P.A. may have broadened the doctrine in 1982 when it denied patent protection to an expert system for neurological diagnosis on the basis that "their invention is concerned with replacing, in part, the thinking processes of a neurologist with a computer."[145] Moreover, the Federal Circuit has implicitly used the doctrine to invalidate a claim for an invention designed to determine whether any complex system is in a normal or abnormal state.[146]

In its broadest form, the mental steps doctrine would deny patent protection to any expert system. Since object-oriented development emphasizes approaching the project from the perspective of an expert in the problem domain, semantic data models may mirror the mental process that an expert in that field would use to solve problems. However, many object-oriented programs will be able to survive the mental steps doctrine for two reasons. First, the software claims recently invalidated under the mental steps doctrine involved the calculation of a discrete result and in general seemed close to a simple process of mental calculations.[147] In contrast, many object-oriented programs will model the operation of systems with continuous behavior that produce no discrete "answer" to a problem. For example, the "QuadWorld" program solves no specific problem, but rather provides a system for drawing and manipulating a variety of shapes. It is difficult to conceive of QuadWorld

---

143. *See In re* Pardo, 684 F.2d 912, 913 (C.C.P.A. 1982) (invention which "converts a computer from a sequential processor . . . to a processor which is not dependent on the order in which it receives program steps"); *In re* Bradley, 600 F.2d 807 (C.C.P.A. 1979) ("firmware" designed to improve performance of multi-tasking), *aff'd sub nom.* Diamond v. Bradley, 450 U.S. 381 (1981).

144. Samuelson, *supra* note 136, at 1034-38.

145. *In re* Meyer, 688 F.2d 789, 795 (C.C.P.A. 1982).

146. *In re* Grams, 888 F.2d 835, 840 (Fed. Cir. 1989) (analogizing to *Meyer* and finding the existence of an algorithm in part because "the objective [in *Meyer*] of identifying malfunction is similar to the objective here of identifying abnormality").

147. In fact, the Federal Circuit never explicitly mentioned the mental steps doctrine but rather denied the claims because "[f]rom the specification and the claim, it is clear to us that applicants are, in essence, claiming the mathematical algorithm, which they cannot do . . . ." *Grams*, 888 F.2d at 840; *see also* Comment, *A Plea for Due Processes*, *supra* note 136, at 1122.

as a series of discrete mental steps that could be performed to achieve the same result.

Second, the C.C.P.A. did not extend the doctrine to cases where the system could theoretically be performed as a series of mental steps but, as a practical matter, would be too complex to implement with pen and paper.[148] Similarly, many object-oriented programs will reflect complex relationships between elements of an extremely large system. At the level of "programming in the colossal", few object-oriented systems can be reduced to a series of human mental process steps. As a result, the mental steps doctrine should not provide an independent bar to patentability.

3. *METHOD OF DOING BUSINESS*

If the method of doing business limitation were applied seriously, it would exclude from protection any computer program that implemented familiar business systems, a category that could ensnare many object-oriented database systems. However, the doctrine is of questionable validity[149] and has only been weakly applied in computer cases. For example, in *Paine, Webber v. Merrill Lynch,*[150] a district court noted the existence of the doctrine, but dismissed it because

> [t]he product of the claims of the '442 patent effectuates a highly useful business method and would be unpatentable if done by hand. The C.C.P.A., however, has made clear that if no *Benson* algorithm exists, the product of a computer program is irrelevant, and the focus of analysis should be on the operation of the program on the computer.[151]

On this basis, the court upheld a claim for a computer program that implemented Merrill Lynch's "Cash Management Account System" which allowed customers to combine brokerage, money market,

---

148. *See In re* Toma, 575 F.2d 872 (C.C.P.A. 1978) (allowing claims for a computer process of translating from any source language to any target language by examining a language dictionary, examining the syntax of the source and then producing a complete sentence in the target language).

149. Arthur J. Hansmann, *Method of Doing Business*, 50 J. PAT. OFF. SOC'Y 503, 504 (1968) ("Except for dicta, one can conclude that there is no basis in existing law for the rejection of claims as being directed to a 'method of doing business.' "); David J. Meyer, Note, Paine, Webber, Jackson and Curtis, Inc. v. Merrill Lynch, Pierce, Fenner & Smith: *Methods of Doing Business Held Patentable Because Implemented on a Computer*, 5 COMPUTER L.J. 101, 103-04 n.13 (1984) (reviewing the cases cited in *Merrill Lynch* and concluding that "examination of these cases reveals that the issue of patentable subject matter was never actually decided. Rather, the patent claims were held invalid for 'lack of invention.' ... The issue of the patentability of a method of doing business was discussed only in dictum ...."); Comment, *A Plea for Due Processes*, *supra* note 136, at 1119 ("[I]t is unclear whether this doctrine ever really existed ....").

150. Paine, Webber, Jackson & Curtis, Inc. v. Merrill, Lynch, Pierce, Fenner & Smith, Inc., 564 F. Supp. 1358 (D. Del. 1983).

151. *Id.* at 1369.

checking, and credit cards into one integrated account. Similarly, the C.C.P.A. has allowed claims for a program to control the optimal operation of plants, such as oil refineries, at multiple locations,[152] and for a program that produced architectural specifications and project control instructions.[153] Finally, in the only case to invalidate a business methods program, the C.C.P.A. did not even mention the doctrine but declared the claim invalid on the basis that the claim recited and preempted a specific algorithm.[154]

These cases indicate that the doctrine may have little effect on inventions related to computer programs in general. Moreover, object-oriented programs will be affected even less by the doctrine than traditional software. As discussed in the next Section, if the program merely implements a familiar business method, then prior art relevant to general business methods and practices will invalidate the claim on § 103 grounds.[155] Thus, courts evaluating such claims will never have to reach the business methods question, since it will be easier to resolve the issue on § 103 grounds and confine the analysis of patentable subject matter to determining whether the claim recites a mathematical algorithm.

B. Non-Obviousness and the Relevant Prior Art

Even though semantic data models qualify as patentable subject matter, few patents will actually be issued because few will pass the non-obviousness requirements of 35 U.S.C. § 103. While critics of software patents have claimed that the Patent Office lacks the expertise or the database files to accurately evaluate prior art for software patents,[156] this problem is considerably less severe in the case of object-oriented software. Since a standard for determining the relevant fields of prior art is "whether it deals with a problem similar to that being addressed by the

---

152. *In re* Deutsch, 553 F.2d 689 (C.C.P.A. 1977).

153. *In re* Phillips, 608 F.2d 879 (C.C.P.A. 1979).

154. *In re* Maucorps, 609 F.2d 481, 486 (C.C.P.A. 1979) (claim for a program that determined the optimal organization of a sales force).

155. *See* 35 U.S.C. § 103 (1988).

156. *See, e.g.*, Brian Kahin, *The Software Patent Crisis*, TECH. REV., April 1990, at 53, 55.

> The search [for software prior art] is extraordinarily difficult because the field's printed literature is thin and unorganized. Software documents its own design, in contrast to physical processes, which require written documentation. Also, software is usually distributed without source code under licenses that forbid reverse engineering. This may amount to suppressing or concealing the invention and therefore prevent the program from qualifying as prior art. . . . Many programmers suspect that patent examiners lack knowledge of the field, especially since the Patent Office does not accept computer science as a qualifying degree for patent practice . . . .

inventor,"[157] the examiner will have to search for references not only in the computer science area, but also in the literature relating to the real-world problem being addressed by the software. Because the programmer first approached the project in step 1 of our object-oriented design model by learning as much as possible from experts in the problem domain itself,[158] the examiner will also have to use the full range of literature in the problem domain as prior art. This search may prove fairly easy since the applicant's duty of candor will require the programmer to unilaterally disclose to the patent office all the sources used in developing the project.[159]

The prior art problem will also be less significant because the examiner will be better equipped to determine non-obviousness.[160] For example, in the traffic light problem, the examiner will compare the semantic data model to literature that describes the operation of the physical entities that operate traffic lights in the real world. In general, this literature will reflect principles of electrical and mechanical engineering that are more familiar to most examiners than principles of computer science. The examiner can quickly determine whether the software is merely a straightforward model of the physical system and therefore obvious to the hypothetical person of ordinary skill in the art of object-oriented design. In many cases, the claim for the semantic data model will read almost exactly like a claim for a well-documented physical system and thus will quickly appear obvious to the examiner.

Under this analysis, the elements of object-oriented software that most embody object-oriented design are patentable subject matter. As a practical matter, however, only the small percentage of semantic data models that are truly non-obvious, in the face of an extremely broad range of relevant prior art, will be granted patents.

---

157. Union Carbide Corp. v. American Can Co., 724 F.2d 1567, 1572 (Fed. Cir. 1984) ("The determination that a reference is from a nonanalogous art is therefore two-fold. First, we decide if the reference is within the field of the inventor's endeavor. If it is not, we proceed to determine whether the reference is reasonably pertinent to the particular problem with which the inventor was involved." (quoting *In re* Wood, 599 F.2d 1032, 1036 (C.C.P.A. 1979))); *see also* Bott v. Four Star Corp., 218 U.S.P.Q. (BNA) 358, 368 (E.D. Mich. 1983) ("The test for relevant or analogous prior art is 'similarity of element, problems, and purposes.' 'Analogous art is that field of art which a person of ordinary skill in the art would have been apt to refer in attempting to solve the problem solved by a proposed invention.' " (citations omitted)), *aff'd*, 732 F.2d 168 (Fed. Cir. 1984).

158. *See supra* Section III.C.1.

159. 37 C.F.R. § 1.56 (1992).

160. An often-heard complaint against traditional software patents is that examiners untrained in computer science "naturally have a lower standard in determining the hypothetical 'person having ordinary skill in the art,' and are thus more apt to grant patents for obvious processes." Kahin, *supra* note 156, at 55.

VI. IMPLICATIONS FOR INNOVATION POLICY

This article has presented the most important concepts of object-oriented programming and discussed one model of the object-oriented design process. A thorough understanding of object-oriented design shows that copyright protection cannot be justified for the elements of the software that make it object-oriented. In addition, one product of the object-oriented design process, the semantic data model, may be used to draft a patentable claim, if it is sufficiently innovative relative to an extremely broad range of prior art.

These conclusions are based on the application of existing legal doctrines, rather than on an analysis of which legal regime provides better protection as a matter of innovation policy. While the industry is still too new to perform any worthwhile empirical analysis, several intuitive observations about the advantages of each legal regime suggest that limited patent protection provides reasonable incentives for innovation.

Legal protection for software designed according to object-oriented principles is important only at the margins of innovation. Assuming that verbatim copying and distribution can be prevented,[161] most companies that write commercial software are not motivated by the promise of broad legal protection for their products but by economic returns that result from being first in the market or being the first to introduce programs with new features into an existing market. While there is no method to test this hypothesis, the nature of object-oriented development itself suggests that companies will be motivated to innovate without extensive legal protection. Object-oriented programming will be adopted because it allows complex programs to be created with fewer errors, encourages the use of existing software components, leads to easier software maintenance, and eases the process of improving the software in subsequent versions. As a result, companies utilizing object-oriented techniques will produce better products and face lower development costs than companies using traditional techniques. This improvement in software "manufacturing" will provide most of the incentive necessary to stimulate innovations in object-oriented software. Indeed, these incentives are evident in the growth of the Object Management Group

161. Of course, outright piracy destroys all incentives for innovation. However, verbatim copying and distribution is easy to prohibit, at least within the United States, through traditional copyright protection for literal source and object code. As in the prior discussion of legal doctrines, this discussion is concerned with the more interesting problems presented by copyright and patent protection for the object-oriented elements of software.

(OMG), a "technology endorsement group" whose membership includes almost 200 leading software companies.[162]

Against this background of strong innovation, legal protection can provide some additional incentives. Analyzing legal protection from the standpoint of innovation policy suggests that copyright protection would be inappropriate for object-oriented software[163] while patent protection may be justified for highly innovative programs.

Copyright protection has been popular for traditional software primarily because it is easy to obtain, it provides strong protection against literal and non-literal copying, and it still allows for a defense of independent creation. However, copyright protection presents serious problems when applied to the aspects of a particular program that make it object-oriented. When applying for a copyright, the author need not make any attempt to define the scope of the copyright being claimed. The author simply submits a copy of the source code or object code of the program with the registration form, and copyright protection instantly attaches. As the Second Circuit recently noted, "we think that copyright registration—with its indiscriminating availability—is not ideally suited to deal with the highly dynamic technology of computer science."[164]

The net result of this process is that the scope of any particular copyright is not defined until litigation occurs and, even then, is only defined relative to the particular program accused of infringement. As a result, business competitors cannot legitimately plan future products since they cannot be sure of how to "design around" an existing copyright. This problem is most acute when protection is claimed for the non-literal elements of a program, such as the semantic data model. While the independent creation defense provides some protection for competing software developers, the high mobility of software engineers[165] combined with the questionable legal status of reverse engineering techniques[166] may render any protection highly illusory.

---

162. Object Management Group Purpose and Definition Statement (1992) (on file with author). OMG was originally formed in April 1989 by Data General, Hewlett-Packard, Sun, Canon, American Airlines, Unisys, Philips, Prime, Gold Hill, Soft-Switch, and 3-Com. Major software players such as AT&T, Digital, NCR, Borland, Microsoft, and IBM have subsequently joined.

163. Remember that copyright protection would still be available to prevent verbatim copying of source and object code, thus supplying the necessary prerequisite to innovation discussed earlier.

164. Computer Assocs. Int'l v. Altai, Inc., 982 F.2d 693, 712 (2d Cir. 1992).

165. Absent costly clean room development procedures, the accused infringer may find it extremely difficult to prove that every engineer involved in the project was completely ignorant of the plaintiff's copyrighted program.

166. Once reverse engineering has been used, the defendant can no longer claim independent creation. Even if the final program is non-infringing, the process of reverse engineering itself could constitute copyright infringement. At the present time, reverse engineering is a risky business strategy. However, the judicial attitude toward reverse

Moreover, the basic fit between copyright protection and continuing innovation must be questioned. Even though the Supreme Court's decision in *Feist* breathed new life into the originality requirement, copyright makes little distinction between the protection afforded to trivial innovations and the protection given to major innovations. If courts adopt broad *Whelan*-style protection for semantic data models, then fairly trivial applications of object-oriented principles are likely to be granted strong protection.[167] This protection will not be offset by the benefits of disclosure of the innovation since commercial programs are distributed in object code form only and copyright registration can be obtained without disclosing the source code to the public. Since any object-oriented innovations occur at the source code level, the public gains no new knowledge from the grant of copyright. At most, the public gains access to a commercial product that might not have been created without the promise of copyright. On balance, copyright protection seems likely to stifle competition and discourage continuing innovation.

In contrast, while the patent examination process makes patent protection more difficult and costly to obtain, this process also addresses the primary deficiencies in copyright protection. First, the inventor must specifically define the scope of the software invention through technical claim language. This language is likely to be narrowed during the examination process in order to overcome prior art rejections. As a result, only highly innovative programs will be granted protection, and the scope of that protection will be sharply limited by prior art. Business competitors can then rationally plan competing products by performing patent searches and then determining how to design around existing patents.[168] If designing around an existing patent is not feasible, the precise definition provided by the claim language will make it easier for the parties to estimate the value of the patent and negotiate licenses. Finally, since patent protection is given only as the *quid pro quo* for full disclosure of the innovation, the costs of protection are offset by dissemination of the new technological knowledge behind the invention as well as by dissemination of the commercial product itself.

---

engineering may be changing,. Two appellate courts have recently applied the "fair use" doctrine to allow reverse engineering in certain contexts. *See* Sega Enters. v. Accolade, Inc., 977 F.2d 1510, 1520 (9th Cir. 1992), *amended*, 1993 U.S. App. Lexis 78 (9th Cir. 1993); Atari Games Corp. v. Nintendo of Am., Inc., 975 F.2d 832 (Fed. Cir. 1992).

167. *See, e.g.*, *Computer Assocs.*, 982 F.2d at 712 (noting that "serious students of the industry have been highly critical of the sweeping scope of copyright protection engendered by the *Whelan* rule, 'in that it enables first comers to "lock up" basic programming techniques as implemented in programs to perform specific tasks' " (quoting Menell, *supra* note 3, at 1087; citations omitted)).

168. Admittedly, business competitors still face some risk since patent applications are kept secret during the examination process.

Patent protection does present several drawbacks. First, the costs of obtaining protection or defending a potential infringement suit may deter some smaller companies from innovation. Second, the lack of an independent creation defense sharply increases the societal costs of protection by stifling innovation that would have occurred in the absence of patent protection. Finally, the seventeen-year term for the patent monopoly is excessively long given the short product cycles for most software projects. Nonetheless, if patent examiners utilize a broad range of relevant prior art both to reject applications that are not highly innovative and to limit the scope of patents actually granted, then patent protection is a more effective incentive for innovation than copyright protection.