Michigan Law Review

Volume 91 | Issue 3

1992

Defining Computer Program Parts Under Learned Hand's Abstractions Test in Software Copyright Infringement Cases

John W.L. Ogilive
University of Michigan Law School

NOTE

Defining Computer Program Parts Under Learned Hand's Abstractions Test in Software Copyright Infringement Cases

John W.L. Ogilvie

INTRODUCTION

Although computer programs enjoy copyright protection as protectable "literary works" under the federal copyright statute,[1] the case law governing software infringement is confused, inconsistent, and even unintelligible to those who must interpret it.[2] A computer program is often viewed as a collection of different parts, just as a book or play is seen as an amalgamation of plot, characters, and other familiar parts. However, different courts recognize vastly different computer program parts for copyright infringement purposes.[3] Much of the disarray in software copyright law stems from mutually incompatible and conclusory program part definitions that bear no relation to how a computer program is actually designed and created. These differing part definitions frustrate courts' efforts to compare or reconcile claims of substantial similarity, an issue that constitutes the cornerstone of many copyright infringement cases.[4]

Substantial similarity between the allegedly infringing program and the copyrighted program is not the only element of a software copyright infringement case. Infringement plaintiffs must also prove ownership of a valid copyright, and must establish access by the defendant to the copyrighted and allegedly infringed program.[5] However, because ownership may be shown by a certificate of copyright registration, and access to the allegedly infringed work is often either

1. Copyright Act of 1976, 17 U.S.C. §§ 101, 102(a), 117 (1988); *see also* H.R. REP. No. 1476, 94th Cong., 2d Sess. 54 (1976), *reprinted in* 1976 U.S.C.C.A.N. 5659, 5667 (stating that computer programs are "literary works").

2. *See, e.g.*, Pamela Samuelson, *Reflections on the State of American Software Copyright Law and the Perils of Teaching It*, 13 COLUM.-VLA J.L. & ARTS 61, 66 (1988) (listing unsettled legal issues in software copyright law).

3. *See infra* note 198.

4. *See, e.g.*, Soft Computer Consultants, Inc. v. Lalehzarzadeh, 1989 Copyright L. Dec. (CCH) ¶ 26,403, at 22,538 (E.D.N.Y. 1988) (stating that the "general standard for establishing copying is the substantial similarity test").

5. Frybarger v. International Business Machs. Corp., 812 F.2d 525, 529 (9th Cir. 1987); Whelan Assocs. Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1231-32 (3d Cir. 1986), *cert. denied*, 479 U.S. 1031 (1987).

conceded or easily proven,[6] substantial similarity is often dispositive. Judicial use of multiple discordant tests for substantial similarity therefore creates chaos at the very heart of software copyright infringement law.

Confusion is inevitable because the various substantial similarity tests employed in software copyright cases define a bewildering variety of program parts. For example, some courts seem to treat algorithms as distinct parts,[7] while others simply bundle them into a program's "structure, sequence and organization" (SSO).[8] Some recognize several distinct parts[9] while others concentrate on a program's "total concept and feel."[10] Some parts are defined inconsistently,[11] or not defined at all.[12]

Unstable definitions of software parts undermine meaningful distinctions between the ideas underlying a program and the expression of those ideas. This idea-expression dichotomy is crucial, for although copyright law may protect "expression," it never protects an "idea."[13] Unfortunately, some courts classify certain program parts as ideas while others classify the same parts as expression, never explicitly acknowledging that parts are being treated inconsistently. One court may treat everything except a program's main purpose as potentially protectable expression,[14] while another protects only literal program code and translations thereof.[15]

Substantial adverse consequences arise from the resulting discord. Conflicting and incoherent rules of decision produce contrary outcomes on fundamentally identical facts. Activities clearly permitted under one infringement test may lead to liability under a conflicting test, and no principled basis exists for choosing between existing tests.[16] Conflicting approaches also hinder the reasoned evolution of software copyright law by obscuring the stable foundations of software technology. Copyright law should balance software protection against progress in the programming art and development of new technolo-

---

6. *See infra* note 78.

7. *See, e.g., infra* text accompanying note 99.

8. *See infra* text accompanying Figure 3.

9. *See, e.g., infra* notes 99-106 and accompanying text.

10. *See infra* text accompanying notes 151-62.

11. *See infra* notes 105-08, 159 and accompanying text.

12. *See infra* text accompanying notes 126-34; *infra* notes 143-46 and accompanying text.

13. 17 U.S.C. § 102(b) (1988); Mazer v. Stein, 347 U.S. 201, 217-18 (1954); *see also* MELVILLE NIMMER & DAVID NIMMER, NIMMER ON COPYRIGHT § 13.03[B][2][a] (1992) [hereinafter NIMMER] (discussing the idea-expression dichotomy as it pertains to substantial similarity analysis).

14. *See infra* text accompanying note 136.

15. *See infra* text accompanying notes 123-25.

16. *See infra* text accompanying Figure 3.

gies, but it can only succeed if it is informed by fundamental programming concepts and accepted legal principles.

Legal commentators have only touched on issues relating to the proper definition of computer program parts. Most scholarly commentaries on software copyright law simply ignore the problem of correctly defining computer program parts, focusing instead on the proper scope of protection.[17] However, the need to import fundamental programming concepts into software copyright law has been noted.[18] Several commentaries[19] also recognize the congruence between programming, which creates functional expression from abstract ideas, and Learned Hand's abstractions test,[20] which proposes a hierarchy of levels of abstraction in any copyrighted work, ranging from potentially protectable expression to unprotectable ideas. But even these commentaries do not provide specific, coherent part definitions that are grounded in widely recognized programming concepts.[21]

Learned Hand's famous abstractions test initially appears to offer little assistance in bringing sense and consistency to software copyright infringement law. The abstractions test views literary works as a

17. Debate over the proper scope of protection examines the kinds of copying that constitute infringement of computer programs under copyright law and addresses various policy questions such as the appropriate balance between legitimate competition and infringement. *See, e.g.*, Anthony L. Clapes et al., *Silicon Epics and Binary Bards: Determining the Proper Scope of Copyright Protection for Computer Programs*, 34 UCLA L. REV. 1493, 1502 (1987). Little has been done to define a small yet comprehensive set of program part definitions that is both legally relevant and technically sound. Prior part definitions are often merely technical recitations that are not integrated into the central discussion, which in turn focuses not on part definitions but rather on the policies underlying various forms and degrees of monopolistic protection. *See, e.g., id.* at 1510-35; Susan A. Dunn, Note, *Defining the Scope of Copyright Protection for Computer Software*, 38 STAN. L. REV. 497, 500-03 (1986); Steven R. Englund, Note, *Idea, Process, or Protected Expression? Determining the Scope of Copyright Protection of the Structure of Computer Programs*, 88 MICH. L. REV. 866, 867-73 (1990).

18. *See, e.g.*, 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, 3 (1991) (attempting "to review and analyze existing legal theories in light of emerging software engineering methodologies").

19. *See, e.g., id.* at 17; NIMMER, *supra* note 13, § 13.03[F]; Dunn, *supra* note 17, at 526.

20. *See infra* text accompanying notes 22-25.

21. Beutel suggests that "functional" program items be separated from "descriptive" items, but omits the details needed to perform this separation in practice. Beutel, *supra* note 18, at 29-31. Beutel lists a large number of program items, but provides no guidance in analyzing any program that was not created in accordance with the DoD-Std-2167 software development methodology. Even items in programs so created are apparently identified "by the contracting agency" rather than by their inherent technical properties. *Id.* at 8-16.

Beutel discusses another approach to defining parts by level of abstraction, which is proposed in Gary L. Reback & David L. Hayes, *The* Plains *Truth: Program Structure, Input Formats and Other Functional Works*, 4 COMPUTER LAW. 1 (March 1987). Reback and Hayes suggest a "rule of reason" under Learned Hand's abstractions test, in order to balance programmer creativity with the scope of copyright protection. *Id.* at 4-8. Their approach, however, also fails to provide specific part definitions. Although several program parts are named in passing, the article focuses on policy considerations related to the copyright monopoly; it provides no detailed part descriptions that are both self-consistent and rooted in software's generally recognized technical characteristics. *See id.* at 5-6.

spectrum of patterns, ranging from concrete protectable expression up to abstract unprotectable ideas. In *Nichols v. Universal Pictures Corp.*,[22] Hand wrote:

> 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 perhaps 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.[23]

The abstractions test was formulated before the need arose to frame proper computer program part definitions, and even in the works that spurred its formulation the test provides only general guidance in locating the line between idea and expression. In the particular realm of computer software, "the abstractions test is not easy to apply."[24]

Proper application of the abstractions test is difficult, however, because it requires an understanding of fundamental programming concepts, not because the test is inherently unsuitable. This Note argues that the abstractions test's valuable approach[25] can be adapted to the software realm by recognizing legally several fundamental program parts at different levels of abstraction. Although Learned Hand's test is not a panacea for all the current ills of software copyright law, it provides a framework for coherent program part definitions that should increase that law's consistency.

This Note proposes a set of computer program part definitions that develop Learned Hand's abstractions test to make it more useful in software infringement cases. The Note takes no position on the proper scope of protection for software under copyright law, but argues that no consensus is possible on which program parts deserve copyright protection until courts recognize that computer programs are composed of components whose definition lies beyond judicial control. Program parts defined in conclusory legal terms will never provide a stable basis for reasoned debate over the conclusions presumed in the definitions.[26]

This Note advocates the orderly development of copyright law through harmonious software part definitions. Part I provides the technical and legal background necessary to examine the proposed

---

22. 45 F.2d 119 (2d Cir. 1930), *cert. denied*, 282 U.S. 902 (1931).

23. *Nichols*, 45 F.2d at 121.

24. NIMMER, *supra* note 13, § 13.03[F], at 13-78.33.

25. Learned Hand's test "is helpful in that it vividly describes the nature of the quest for 'the expression of an idea.'" NIMMER, *supra* note 13, § 13.03[A], at 13-27 (paraphrasing Learned Hand's opinion in Peter Pan Fabrics, Inc. v. Martin Weiner Corp., 274 F.2d 482, 489 (2d Cir. 1960)).

26. *See, e.g., infra* text accompanying note 157.

program parts by presenting some basic software terminology, discussing computer program abstraction parts currently recognized by programmers, and examining abstraction parts defined by courts during their attempts to analyze substantial similarity in software infringement cases. Part II argues for judicial adoption of the computer program abstraction part definitions presented in Part I. This second portion of the Note first develops requirements that any set of abstraction part definitions should satisfy and argues that the proposed definitions meet these requirements. Part II then discusses the costs and benefits of change, arguing that judicial agreement on a coherent set of abstraction part definitions must precede any consensus on the proper scope of protection for software under copyright law. The Note concludes that refining Learned Hand's abstractions test to recognize the proposed program parts will reduce the chaos presently hindering software copyright infringement law.

I. EXISTING TECHNICAL AND LEGAL PART DEFINITIONS

Software begins as an abstract idea and progresses through increasingly specific stages until a literal program emerges.[27] These stages are excellent candidates for refinements of the abstractions test, not merely because they arise through step by step refinement of an abstraction, but also because they rest on fundamental programming concepts. A firm understanding of both programming and existing copyright law, however, is necessary before defining these stages as program parts under the abstractions test. Section I.A therefore introduces some basic software concepts. Section I.B defines and illustrates the abstraction parts programmers use while designing, writing, and enhancing software; this Note proposes judicial recognition of these parts. Section I.C discusses abstraction parts courts have previously defined, often implicitly, in applying various tests for substantial similarity to computer programs.

A. *Software Basics*[28]

A *program*[29] or piece of *software*[30] is an organized set of instructions that guides a computer. Software, which "runs on" a computer,

27. ALFRED V. AHO ET AL., DATA STRUCTURES AND ALGORITHMS 1 (1983).

28. Many introductory works on computers and software are available. *See, e.g.*, MICHAEL COVINGTON & DOUGLAS DOWNING, BARRON'S DICTIONARY OF COMPUTER TERMS (3d ed. 1992) [hereinafter BARRON'S]; ADAM OSBORNE, AN INTRODUCTION TO MICROCOMPUTERS (2d ed. 1980).

29. "A 'computer program' is a set of statements or instructions to be used directly or indirectly in a computer in order to bring about a certain result." 17 U.S.C. § 101 (1988).

30. This Note uses *program* and *software* interchangeably to represent every aspect, from the most specific to the most general, of an organized set of instructions that guide a computer. *Code*, by contrast, is used in a much narrower sense. *See infra* notes 32-35 and accompanying text; *infra* Figures 2, 3; *infra* notes 67-72 and accompanying text.

is distinguished from the physical computer itself, which is *hardware.* Familiar examples of programs include word processors, spreadsheets, and database management software.[31] Programs also perform a wide variety of other tasks, from rendering graphic images to modeling weather patterns and controlling industrial robots.

The literal text comprising a program's instructions, known as *source code,* is written in one or more *programming languages.* These languages resemble human languages such as English, but have much less room for ambiguity.[32] Each programming language has a unique grammar and set of meanings.[33] Two programs may perform the same functions despite differences in their source code. Conversely, two programs with nearly identical source code may perform very differently. Before source code can be used by the computer it must be translated into a form recognizable to the computer hardware. A *compiler* translates the source code into *object code,*[34] a string of ones and zeros[35] that controls the hardware. Because different computer hardware requires different object codes, one must translate a piece of source code once per hardware type to produce the object codes needed to run the "same" program on different types of computer.

The fundamental distinction between source code and object code illustrates software's multifaceted nature. Programs, like novels or legal opinions, can be usefully viewed from a variety of perspectives. When the object code runs, the computer hardware interacts with as-

---

31. As this Note will occasionally use database examples, the following definition may be helpful. A *database* is a collection, typically quite large, of discrete pieces of information (data) of a certain type, organized to facilitate adding, removing, modifying, reading, grouping, and summarizing individual pieces of information. *See, e.g.,* KAMRAN PARSAYE ET AL., INTELLIGENT DATABASES 17 (1989). For instance, a database of student records might contain each student's name, ID number, address, year in school, and current class schedule, as well as additional information. Among other operations, the database management software might permit one to read an individual student's record given the student's name or ID number, change the student's address, find out how many students are in their third year, and print the names of every student enrolled this semester in Copyright Law.

32. Spelling errors illustrate one obvious difference between English and computer programming languages. Replacing "receipt" by "reciept" in an English sentence normally will not change the sentence's meaning, as readers will merely treat "reciept" as an incorrect spelling of "receipt." In a computer program, however, "reciept" and "receipt" will typically be treated as two distinct entities, and substituting one for the other may easily prevent the program from working properly.

33. Programming language grammar rules are far stricter than those of English. "An interpreter of programming-language texts, a computer, is immune to the seductive influence of mere eloquence." JOSEPH WEIZENBAUM, COMPUTER POWER AND HUMAN REASON 108 (1976). For an example of source code, see *infra* note 74.

34. Computer programs that translate source code into object code are known as *compilers* or *interpreters.* BARRON'S, *supra* note 28, at 76-77.

35. Computers are machines that can only "understand" and process information in the form of ones and zeros, because their basic electronic components, *transistors,* operate only on one or the other of two possible voltage levels. BARRON'S, *supra* note 28, at 329-30. Object code could just as easily use *X* and *Y* or some other pair of symbols to represent these meaningful voltage levels; it is irrelevant what labels are used. RICHARD B. KIEBURTZ, STRUCTURED PROGRAMMING AND PROBLEM-SOLVING WITH PASCAL 332 (1978).

pects of the program that are largely ignored by programmers. A programmer's point of view, in turn, often differs from that of the program's ultimate user.[36] One might note other differences in perspective,[37] but two are particularly relevant. First, this Note focuses on algorithms and other internal aspects of software that are familiar to programmers but largely invisible to program users. Accordingly, infringement tests developed for use in comparing program user interfaces[38] are relevant here mainly as limits on the applicability of the present discussion. Second, this Note describes program parts from a perspective that is more legal than technical, centering its discussion on the relationship between existing doctrines and the proposed part definitions. Sufficient material from computer science is included in the next section to define clearly and completely the proposed parts, but many points programmers would consider important are dealt with only in the footnotes, or omitted entirely.

B. *Abstraction Parts Used by Programmers*

Programs embody different levels of abstraction because software is best created through a method known as top-down programming, a process that starts with a concept and culminates in a particular computer program.[39] A program begins as a purpose or desired function, which programmers expand into a preliminary design. Programmers make this design increasingly specific by splitting large tasks into smaller ones and defining the interaction of these tasks. Some parts of the design may organize the program's information in convenient formats, while other parts may manipulate or transform the information. Programmers then implement the detailed design by writing source code that describes it. Finally, programmers and others test the program, document it, and release it to users. Although programming does not always proceed neatly from one stage to the next in prac-

36. *See* JOHN W.L. OGILVIE, ADVANCED C STRUCT PROGRAMMING: DATA STRUCTURE DESIGN AND IMPLEMENTATION IN C 10-13 (1990) (describing programming from the perspectives of a customer, a nonprogramming contributor to a program, and a programmer).

37. Whereas legal analysis in copyright infringement cases typically compares "snapshots" of two programs taken at single point in time, the "software life cycle" perspective traces a program's development over time. TOMLINSON G. RAUSCHER & LINDA M. OTT, SOFTWARE DEVELOPMENT AND MANAGEMENT FOR MICROPROCESSOR-BASED SYSTEMS 4-9 (1987).

38. A program's user interface is the way it communicates with the person who is using it. In some programs, screen menus provide an interface permitting users to select an item by typing a numeric digit or alphabetic character; in other programs, users interface with the program by selecting icons (pictures) with a mouse. BARRON'S, *supra* note 28, at 343. *See also infra* note 73.

39. *See generally* GRADY BOOCH, SOFTWARE ENGINEERING WITH ADA (1983) (discussing the top-down programming process).

tice,[40] the idealization suffices for this Note.[41]

Most programmers[42] recognize certain program parts and levels of abstraction.[43] Partly because of their programming utility, these levels of abstraction are well-suited for use as refinements of Learned Hand's abstractions test. Because these parts arise naturally from the inherent structure of software,[44] they also present a coherent abstractions framework that will facilitate substantial similarity analysis in software copyright cases. These levels of abstraction include: (1) the program's main purpose; (2) its system architecture; (3) various abstract data types; (4) various algorithms and data structures; (5) the source code; and (6) the object code.

The following description of these program parts introduces several concepts that are unfamiliar to nonprogrammers. Readers whose training is primarily in law rather than computer science will gain an increased understanding of software technology. Software copyright law should reflect the engineering realities of programming, just as the Uniform Commercial Code reflects actual mercantile practice[45] and

40. In theory as well as in practice, some top-down programming steps proceed simultaneously rather than sequentially. Testing and documentation, for example, should begin when a program is designed and continue throughout the program's life. *See, e.g.*, OGILVIE, *supra* note 36, at 14 (proposing a programming process in which the goals of people with whom the program interacts are compared to each other and to the program's abilities, appropriate adjustments are made, and the entire cycle repeats); Englund, *supra* note 17, at 871 n.25 (1990) ("It is often necessary to return to an 'earlier' stage of the design process, and frequently, a number of these steps are performed simultaneously.") (citations omitted).

41. Other commentators have made similar simplifying assumptions. *See, e.g.*, NIMMER, *supra* note 13, § 13.03[F], at 13-78.32 n.291 (omitting discussion of debugging, program documentation, and maintenance when examining top-down programming).

42. Academic writers disagree somewhat when naming the various levels of abstraction that comprise the essence of top-down programming. *Compare* AHO ET AL., *supra* note 27, at 1 (listing problem formulation and specification, design, implementation, testing, and documentation as programming steps) *with* John M. Conley & David W. Peterson, *The Role of Experts in Software Infringement Cases*, 22 GA. L. REV. 425, 442-49 (1988) (listing conception, architecture design, user language design, modular structure, subroutine structure, algorithms, coding, and formatting as the stages in programming).

43. Although commonly used by programmers, *flowcharts* and *pseudo-code* are omitted from this Note's proposed parts because each mixes several levels of abstraction. Flowcharts are diagrams that may simultaneously represent multiple levels of abstraction, e.g., algorithms, source code, and system architecture. *See* BARRON'S, *supra* note 28, at 136, 137 (providing a definition and sample flowchart). Pseudo-code, which is source code interspersed with English, is likewise unsuitable as a refinement of the abstractions test because it mixes source code and abstract data types (ADTs). (For a discussion of ADTs see *infra* section I.B.3). *See* AHO ET AL., *supra* note 27, at 2, 7 (providing a definition of pseudo-language and several demonstrations of how pseudo-language must be refined into pure code); *see also supra* text accompanying notes 32-35 (discussing source code); *infra* text accompanying notes 49-52 (discussing ADTs).

44. *See, e.g.*, AHO ET AL., *supra* note 27, at 2, 11, 12-13 (defining algorithm, ADT, and data structure); OGILVIE, *supra* note 36, at 26, 329-39, 356-87 (describing a "Functional Module Diagram" that corresponds to system architecture, defining ADTs, providing data structure design guidelines, and discussing source code "porting" to hardware that requires different object code).

45. *See, e.g.*, U.C.C. § 1-102(2) (1987) ("Underlying purposes and policies of this Act are . . . to permit the continued expansion of commercial practices through custom, usage and agreement of the parties . . . .").

real property law reflects pragmatic aspects of land ownership.[46] Moreover, readers familiar with programming should be able to correlate their own training and experience with the technical terms used in this Note. Although the concepts presented below are widely recognized in the programming community, individual programmers may have encountered some of these ideas under different names.

1. *Level One: Main Purpose*

A program's *main purpose* or function is what the program is intended to do. For instance, a database manager's purpose is to manipulate data. However, definitions of a program's purpose may be made increasingly specific: the purpose of a database manager is also to facilitate adding, removing, modifying, reading, grouping, and summarizing individual pieces of information in a large collection of data. These purposes, or those of any other program, could in turn be described in ever greater detail. But in this Note a program's main purpose is whatever the program does, described as specifically as possible without reference to technical aspects of the program — that is, without reference to the other levels of generality described below.

2. *Level Two: System Architecture*

While main purpose describes *what* a program does, a *system architecture* begins to describe *how* the program operates. The system architecture describes the program in terms of various *modules*[47] and their interconnections. Programmers organize software into modules to facilitate program creation, correction, and enhancement. Each module performs a significant portion of the program's main purpose and is eventually implemented as a distinct section of the source code. In a hypothetical database manager, three main modules might handle the user interface, data editing functions, and file management, respectively. The user interface module might in turn contain a module to handle screen display, one to read commands, and one to print reports. The system architecture is often irrelevant to the user. For example, a user would neither know nor care whether the print module lies within the user interface module or resides outside as a fourth main module.

---

46. For instance, the purpose of prescriptive easements and adverse possession is to make "paper" rights conform to actual practice by amending those rights. *See generally* OLIN L. BROWDER ET AL., BASIC PROPERTY LAW 557-68 (5th ed. 1989).

47. As used in this Note, module is not synonymous with *subroutine.* Subroutines are part of the source code and hence are less abstract than modules, which are part of the system architecture. More simply, modules typically contain many subroutines. *See* OGILVIE, *supra* note 36, at 329-30 (explaining that modules may communicate by calling each other's subroutines); *infra* note 74 (listing source code of a subroutine). *But see* Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1230 n.15 (3d Cir. 1986) (treating modules and subroutines both as "discrete parts of programs with readily identifiable tasks," a description that applies equally well to virtually every program part), *cert. denied,* 479 U.S. 1031 (1987).

A program's system architecture specifies three kinds of connection between modules: nesting, control flow, and data flow.[48] Module nesting is one way that programmers break large jobs into smaller ones; module *B* is nested inside module *A* when *B* performs part of *A*'s task. In the hypothetical database manager above, the screen display module is nested inside the user interface module. Control flow describes the order in which the modules run. In the database manager, control flows from the command module, which reads commands from a keyboard or other input device, to the other modules that perform the tasks specified by the user, such as printing a report or totaling a column of numbers. Data flow describes the movement of information between modules. In order to open a file, the database manager's user interface module must pass data in the form of the desired file's name to the file management module. Although these connections might seem useful in mapping different levels of abstraction, this Note argues against such use; connections may help delineate modules, but all modules lie within the system architecture level of abstraction.

3. *Level Three: Abstract Data Types*

The modules that comprise the system architecture contain *abstract data types* (ADTs). Every ADT is jointly defined by two components. First, ADTs contain *operations,* which define the set of actions one may perform using the ADT. Familiar operations on a database include adding a new piece of data and printing a summary of the current data. Second, ADTs also contain *data types,*[49] which define the kind of item the ADT's operations act upon. In a database of law students,[50] student records might be one data type, and class enrollment records could be another.

An ADT modeling a checking account provides another example. The data type might be a dollar figure representing the current balance, while the permitted operations might include depositing funds, withdrawing funds, computing interest, and reading the current balance. Alternatively, a more elaborate ADT, which keeps track of where the money goes, might be preferred. Such an expanded ADT might include four operations — depositing funds, writing a check,

48. Examples of modules that are familiar to programmers include packages in the programming language Ada, BOOCH, *supra* note 39, at 29; modules in Modula-2, JOHN W.L. OGILVIE, MODULA-2 PROGRAMMING 73-97 (1985); and classes in C++, BJARNE STROUSTRUP, THE C++ PROGRAMMING LANGUAGE 143-80 (2d ed. 1991).

49. This term is confusing because it requires one to distinguish between an abstract data type and an abstract data type's data type. Its use, however, is well established. *See, e.g.,* AHO ET AL., *supra* note 27, at 11; PARSAYE ET AL., *supra* note 31, at 105. To minimize confusion, this Note denotes abstract data type by ADT, so one need only distinguish between an ADT and an ADT's data type.

50. *See supra* note 31.

computing interest, and reading the balance — on two types of data: individual check amounts and the current balance.

ADTs in commercial computer programs are generally more complex than the ADTs just described,[51] and often have a stronger mathematical flavor.[52] But the motivation behind every ADT is to associate a given data type with the operations that are useful in manipulating that type. The resulting distinction between actions — ADT operations — and things acted upon — ADT data types — reappears in the next level of abstraction, as a distinction between algorithms and data structures.

4. *Level Four: Algorithms and Data Structures*

Under this Note's analysis, *algorithms* and *data structures* jointly occupy the level of abstraction below ADTs. Algorithms and data structures are more specific versions of ADT operations and data types, respectively. ADT operations are brought closer to realization as functional source code by specifying algorithms that accomplish the desired operations. Similarly, ADT data types are more precisely specified through descriptions employing data structures.

An algorithm is a series of steps that accomplishes a particular ADT operation.[53] While an ADT operation merely identifies a desired result, an algorithm specifies every step necessary to accomplish that result.[54] Algorithms must contain sufficient detail to permit their implementation in source code once a programming language and computer hardware are chosen.[55] The operation "determine whether a number is divisible by nine" may be performed by the following al-

---

51. *See, e.g.*, OGILVIE, *supra* note 36, at 249-91 (describing a working version of a pattern-matching ADT).

52. ADTs may be described mathematically, but are most easily understood through examples like the following. A *character queue* ADT has two operators, enqueue( ) and dequeue( ), that operate on A, B, C, and other characters. If an empty queue looks like this { }, the same queue becomes {A} after enqueue(A). If this operation is followed by enqueue(B) and enqueue(C) in that order, the queue becomes {CBA}. If a dequeue( ) operation follows, A is dequeued, and the queue becomes {CB}. In short, the enqueue( ) operator places data into the queue at one end, and the dequeue( ) operator retrieves the data from the other end of the queue in the same order.

53. *Cf.* AHO ET AL., *supra* note 27, at 2 ("[A]n algorithm . . . is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time."); OGILVIE, *supra* note 48, at 263 ("An algorithm is a sequence of steps which, if followed, will produce a solution to a given problem.").

54. Although actual use of an algorithm may depend on the availability of certain programming language features, any valid algorithm can be completely specified using English and other representations that are independent of any programming language. *See, e.g.*, OGILVIE, *supra* note 36, at 250-52 (describing an algorithm in pseudo-code).

55. The complexity of algorithm specifications varies. An algorithm for updating a checking account balance may be trivial, but many ADT operations involve more complex algorithms. An algorithm must provide sufficient detail to permit its complete and correct implementation in source code. The algorithm must not require infinite time, not require infinite space, accomplish the desired result, and cause no unwanted side-effects. BARRON'S, *supra* note 28, at 6-7. Algorithms may also be subject to additional constraints that promote efficient use of computer hard-

gorithm: "Add the number's digits, then add the digits of the result, then add the digits of that result, and so on, until a single digit remains. The original number is a multiple of 9 if and only if the final single digit is 9."[56] Different algorithms may solve the same problem, as this second algorithm,[57] which also tests divisibility by nine, illustrates: "Keep subtracting 9 from the number until the result is either zero or negative. The original number is divisible by nine if and only if the final result is zero."

Just as algorithms describe specific steps that perform ADT operations, data structures provide specific representations of ADT data types. Together with algorithms, data structures are among the most widely recognized and studied of the computer program parts discussed in this Note.[58] Several concepts introduced in the upcoming description of data structures will be unfamiliar to nonprogrammers,[59] but part definitions that lack sufficient engineering detail quickly degenerate into vague notions that vary widely from case to case.[60] Proper understanding and identification of data structures therefore requires discussion of the following six data structure components: basic data type, value, variable, array, record, and pointer.

A *basic data type* describes a set of *values.* Zero, 1, and −1 are *integer*[61] values. Other basic data types include *character* and *floating point.* "A," "B," and "C" are character values. The numbers 0.0625

---

ware and programmer time. *See, e.g.,* AHO ET AL., *supra* note 27, at 264, 270; OGILVIE, *supra* note 36, at 11.

56. To see whether 657 is a multiple of 9, first add 6 + 5 + 7 to get 18. Eighteen has more than one digit, so add its digits 1 + 8 = 9. The final single digit is 9, so the original value, 657, is a multiple of 9. (In fact, 657 equals 9 times 73.)

57. Although these particular algorithms determine divisibility and so make sense only for numbers, many algorithms work on a wide variety of data types. *See, e.g.,* BRIAN W. KERNIGHAN & DENNIS M. RITCHIE, THE C PROGRAMMING LANGUAGE 114-17 (1st ed. 1978) (describing a routine that sorts either character strings or numbers as implemented and could be generalized even further).

58. Data structures and algorithms are so central to software that Niklaus Wirth's well-known work on computer programming is entitled ALGORITHMS + DATA STRUCTURES = PROGRAMS (1976). Other widely used works that discuss data structures and algorithms include AHO ET AL., *supra* note 27, and DONALD E. KNUTH, THE ART OF COMPUTER PROGRAMMING (2d ed. 1973).

59. Algorithms and data structures may be equally complex in practice, but algorithms are more familiar and hence require less description here. Instructions for anything from map reading to bicycle assembly may be described in terms that are quite similar to software algorithms. The closest nontechnical analogy to a data structure, however, is a "fill-in-the-blank" form; such forms only hint at the true flexibility and diversity of computer program data structures.

60. As discussed in sections I.C and II.A, *infra,* a major failing of present software substantial similarity tests is their vagueness regarding data structures and other fundamental programming concepts.

61. The complete (infinite) set of mathematical integers is not actually available on any computer. *See, e.g.,* KERNIGHAN & RITCHIE, *supra* note 57, at 182 (listing the range of integer values available on different types of computer hardware). However, the differences between mathematical integers and computer integer data types are not discussed here, as they are not critical to an understanding of data structures. Differences between character and floating point data types and their idealized counterparts are likewise ignored.

and 1/3 are floating point values. Different programming languages support different basic data types, but integer, floating point, and character types are widely available.

A *variable* is a named storage location that holds values of some particular data type. One might speak of an integer variable named "Total" that presently holds the integer value 33.[62]

An *array* is a row of some predetermined number of variables of a given data type. The entire array has one name. Each individual variable in the array, or array element, is referred to by the array's name and the variable's relative position (its index) within the array. Suppose the array of ten characters shown in Figure 1 is named Key_Word; the current value of Key_Word[0] is "J," the value of Key_Word[1] is "U," the value of Key_Word[2] is "S," and so on. Various mechanisms permit software to ignore array elements not currently needed, such as Key_Word[7], Key_Word[8], and Key_Word[9]. The value of Key_Word as a whole is "JUSTICE." The variables that constitute an array may also themselves be arrays[63] or one of the other data types described in this section, as long as every element of the array is of the same type.

FIGURE 1
An Array of Character Variables.

A *record,* unlike an array, may group together variables of different basic data types. One might join two integers named Year and Day

62. Basic data types, variables, and values are concepts independent of any given programming language. In practice, however, programmers often use fragments of source code as specific illustrations of these general concepts. For instance, the first line of source code in the programming language C below sets aside a place for storing integer values by creating a variable named Total; the second line gives Total the value 33:

```
int   Total;
Total = 33;
```

The corresponding source code in the Pascal programming language is

```
var   Total integer;
Total = 33;
```

Corresponding source code could also be provided in many other languages because the concept "an integer variable named Total that has the value 33" is independent of any programming language.

63. One might construct an array of television listings by using two indexes, where the first index indicates the broadcast time, the second index is the television channel, and the array values are the names of television shows and movies. In such an array, one might find that the current value of Television_Listings[130, 3] is "Ben Hur." The array can be deconstituted further into variables of the basic data type character, in which case the current value of Television_Listings[130, 3, 0] is "B," Television_Listings[130, 3, 1] is "e," Television_Listings[130, 3, 2] is "n," and so forth.

with an array of characters named Month to create a record named Current_Date; Current_Date is then said to have three fields. The fields of a record are referred to by record name and field name. If an arrow (←) denotes assignment, one could assign a date value to the record variable Current_Date by assigning values to Current_Date's fields as follows:

Current_Date.Year ← 1992
Current_Date.Month ← "August"
Current_Date.Day ← 13

A *pointer* is a connection between two discrete records. Pointers are helpful when a program must manipulate a different number of records each time it runs. Suppose a database program must input widely varying numbers of student records and sort them alphabetically. Any array of record variables will usually be either too large or too small because the number of records being read changes but the number of array elements does not.[64] By using pointers and setting aside space to hold each record's values just before it is read instead of allocating a fixed amount of space ahead of time in an array, the program acquires only as much storage space as it actually needs.

This understanding of the six data structure components permits the formulation of a useful definition of data structures. A data structure consists of one or more variables of the basic data types, which are organized in some specified combination of arrays, records, and pointers. A single variable, such as an integer variable Total_Due, is thus the simplest data structure. Current_Date is a data structure consisting of one record with two integer fields named Year and Day and one character array field named Month. Data values are irrelevant to data structures in the sense that the values stored in a data structure may change without altering the rules that govern the structure's organization. Total_Due is the same data structure no matter which integer value it contains, and Current_Date is the same data structure no matter what values are stored in its fields.

Relatively simple data structures, illustrated by Total_Due and Current_Date, are correspondingly easy to identify in a given program, and to compare in two programs. Assessing the alleged equivalence of more complex data structures may be much harder. Complicated data structures are built by combining simpler data structures, which in turn are built from basic data type variables.[65]

64. Suppose the program runs three times, first sorting 20 student records alphabetically, then sorting 4000 records, and finally sorting 300 records. If the records are stored in an array having only 20 elements, or even 300 elements, the program will not succeed when faced with 4000 records. But if the array is large enough to hold 4000 records, much of the space dedicated to the array is wasted when only 300 or 20 records are sorted. Arrays are therefore not suitable when storage space is scarce and the number of elements actually needed varies widely.

65. *See, e.g.*, AHO ET AL., *supra* note 27, at 13-14; KERNIGHAN & RITCHIE, *supra* note 57, at 119-34.

One may create an array of records, or a record whose fields are arrays, or considerably more complex structures. The collection of all functionally equivalent data structures in any actual situation may be quite large and diverse. Accordingly, an ADT data type may often be represented by more than one data structure,[66] just as various algorithms may be available to perform an ADT operation.

In summary, recall that ADTs consist of operations and data types; operations denote actions, and data types denote the type of item acted upon. An algorithm is a sequence of steps, not necessarily unique, that performs an ADT operation; a data structure is one possible representation of an ADT data type. Each data structure is a collection of basic data type variables combined using arrays, records, and pointers. Although data structures and algorithms are more specific than ADTs and may even depend on certain programming language features for use, they are independent of any specific programming language or piece of literal source code.

5. *Levels Five and Six: Source Code and Object Code*

Source code[67] is the literal text of a program's instructions, written in one or more programming languages. Source code can be read by programmers, but to run a program on a computer the program's source code must be translated into *object code.*[68] Object code is a string of ones and zeroes tailored to turn on and off the various electronic switches in a particular kind of computer, thereby manipulating the meanings associated by programmers with different switch settings.

Consideration of source code and object code as levels of abstraction for copyright purposes raises two definitional issues. First, it may or may not be appropriate to distinguish source code from object code. Second, it may or may not be appropriate to distinguish either type of code from the program as a whole. Source code and object code both belong in the set of abstraction part definitions only if source code and object code are distinct from each other and distinct as well from the levels of abstraction already described.

In response to the first issue, object code clearly lies at a lower level of abstraction than source code, because object code must contain significant detail not found in the corresponding source code in order to control a given computer.[69] Resolution of the second issue, however,

66. *See, e.g.,* OGILVIE, *supra* note 36, at 360-65 (explaining that linked lists and arrays are somewhat interchangeable representations of an ADT list data type).

67. *See supra* notes 32-33 and accompanying text.

68. *See supra* notes 34-35 and accompanying text.

69. Translators which create object code from source code add specificity most obviously by taking hardware differences into account, but they may also "fill in the blanks" left by source code in many other ways. Translators may decide when to retain a value in one of a few very rapidly accessible locations (CPU registers) instead of storing it in one of the more numerous but

requires reconsideration of the abstraction parts already described. Main purpose, system architecture, ADTs, algorithms, and data structures are discussed above as if they were tangible, and to professional programmers, they are indeed no more ethereal than a character is to a novelist. In the courtroom, however, these program parts exist only as expert opinions mined from the source code and object code.[70] A program's entire range of abstraction is embedded in its code in roughly the same way a novel's characters and plot are embedded in its text.[71] Therefore, care must be taken to avoid confusing code as the embodiment of parts at every level of abstraction with code as a level of abstraction in its own right.[72] This Note denotes code-as-embodiment by *program* or *software,* and code-as-a-level by *code.* A program consists of a main purpose, a system architecture, ADTs, algorithms, data structures, and code; code may therefore be defined as "the portion of a program that does not overlap the program's main purpose, system architecture, ADTs, algorithms, or data structures." Every level of abstraction but code is clearly bounded under the definitions provided above, and these other levels taken together do not exhaust the contents of a program. Code may therefore be effectively defined as a discrete program part through a process of elimination.

Programmers, then, see programs as consisting of parts lying at six

---

less rapidly accessible locations (RAM). They may also determine what value (if any) to give uninitialized variables; whether to check array bounds at run-time; the order in which to evaluate function and procedure actual parameters; and the order in which to evaluate operands and subexpressions. *See generally* ALFRED V. AHO ET AL., COMPILERS: PRINCIPLES, TECHNIQUES AND TOOLS (1986).

70. *See, e.g.,* Q-Co Indus., Inc. v. Hoffman, 625 F. Supp. 608, 610 (S.D.N.Y. 1985) ("The challenge to counsel to make comprehensible for the court the esoterica of bytes and modules is daunting."); Conley & Peterson, *supra* note 42, at 438, 442 (arguing that experts play a critical role by educating the court about the phases of top-down programming); *see also infra* notes 128-29, 177, 207.

71. The analogy between abstraction parts embedded in a program's code and various novel parts embedded in a novel's text is not exact because abstraction parts are often shaped by utilitarian concerns for compatibility and efficiency, whereas novels are guided more often by purely aesthetic concerns. *See generally* NIMMER, *supra* note 13, §§ 13.03[F][2], 13.03[F][3].

72. The dual nature of code as an embodiment of all program parts, which is nonetheless a part in its own right, recalls the final lines of a Yeats poem:

> O chestnut-tree, great-rooted blossomer,
> Are you the leaf, the blossom or the bole?
> O body swayed to music, O brightening glance,
> How can we know the dancer from the dance?

WILLIAM BUTLER YEATS, *Among School Children, in* THE COLLECTED POEMS OF W.B. YEATS 212, 214 (definitive ed. 1956).

Just as the code for an entire program embodies every level of abstraction, code for a particular subroutine within any program generally embodies at least code, algorithm, and data structure abstraction parts, and may embody more abstract parts as well. *Cf.* Computer Assocs. Intl., Inc. v. Altai, Inc., 23 U.S.P.Q.2d (BNA) 1241, 1252 (2d Cir. 1992) (noting that "each subroutine is itself a program"). One court may refer to a subroutine to exemplify some very specific aspect of a program, *see infra* text accompanying note 107, while another court uses the same term to refer to much more general program components, *see supra* note 47 (noting that *Whelan* treats module and subroutine as synonyms). Subroutine is therefore too broad to serve as a useful program abstraction part. *But cf. infra* note 139.

levels of abstraction: (1) main purpose; (2) system architecture; (3) ADTs; (4) algorithms and data structures; (5) source code; and (6) object code.[73] These levels arise naturally from the inherent structure of software.[74] As the next section reveals, however, legal definitions of

73. Application of the abstractions test to copyright questions involving a program's user interface, *see supra* note 38, requires analysis beyond this Note's scope, but a brief discussion may help clarify the limits of the present proposal. A program's user interface is often a program in its own right, not merely in terms of the effort expended or the resulting complexity, but also in terms of the range of abstraction spanned. A user interface has a main purpose, a system architecture, and every other level of abstraction proposed in this Note. However, additional part definitions are needed to tailor the abstractions test for use in comparing user interfaces. A scheme somewhat like the following might be found appropriate if studied at greater length: (1) main purpose; (2) choice of interface hardware, e.g., mouse versus light pen, color monitor versus monochrome; (3) software system architecture; (4) ADTs; (5) algorithms and data structures; (6) rules promoting uniformity across multiple programs, e.g., placing certain menu options in every program, or assigning the same task to the same key sequence in every program; (7) order and presentation, e.g., menu contents and their order, default sizes and colors for windows, font choices, and function key assignments; (8) source code; and (9) object code. There is no manageable and complete set of stable, nonoverlapping user interface abstraction parts that defers appropriately to existing law and accepted programming concepts. *See, e.g.*, Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 560 (E.D.N.Y. 1991) (listing object code, source code, parameter lists, services required, and general outline as abstraction parts of an interface between two programs), *affd.*, 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992); Lotus Dev. Corp. v. Paperback Software Intl., 740 F. Supp. 37, 63-68 (D. Mass. 1990) (examining user interface elements proposed by plaintiff, including menus, menu structure and organization, "long prompts," screens on which long prompts appear, function key assignments, and the macro commands and language); Manufacturer's Technologies, Inc. v. Cams, Inc., 706 F. Supp. 984, 994-95 (D. Conn. 1989) (discussing copyrightability of "external flow and sequencing" of display screens, screen formatting style, "internal method of navigation" within a screen, and method of identifying the operation or department being utilized); Telemarketing Resources v. Symantec Corp., 1990 Copyright L. Dec. (CCH) ¶ 26,514 (N.D. Cal. Sept. 6, 1989) (discussing similarity of menu screen options, use of pull down windows and menu bar, presence of an editing screen, and default color selections), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir. 1992), *cert. denied*, 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992); Pearl Sys., Inc. v. Competition Elecs., Inc., 8 U.S.P.Q.2d (BNA) 1520, 1523 (S.D. Fla. 1988) (discussing software differences dictated by user interface hardware); Digital Communications Assocs., Inc. v. Softklone Distrib. Corp., 659 F. Supp. 449, 456 (N.D. Ga. 1987) (holding that screen displays are distinct from "the program's source code, object code, sequence, organization or structure" for infringement analysis purposes); Broderbund Software, Inc. v. Unison World, Inc., 648 F. Supp. 1127, 1134, 1137 (N.D. Cal. 1986) (holding that "structure, sequence, and layout" of program's audiovisual displays "were dictated primarily by artistic and aesthetic considerations," and that the screens, sequence of screens, choices presented, screen layout, and method of feedback in the two programs were all substantially similar).

74. As a concrete illustration of top-down programming and the relationship between ADTs, data structures, algorithms, and source code, consider the following transformation of the queue ADT described in note 52, *supra*, into working source code. A programmer first chooses data structures to represent queues and queue elements. The elements in this particular queue are characters. Many programming languages provide a character data type, so the main question is what data structure should represent a queue. Arrays are less flexible than *linked lists* (a data structure familiar to programmers) but are also simpler, so this time the programmer decides to keep the queue in an array. The programmer must also choose appropriate enqueueing and dequeueing algorithms. The algorithms for subroutines enqueue( ) and dequeue( ) follow fairly quickly from the choice of an array as the queue data structure. The gist of each algorithm is to keep track of the queue's first and last elements, which are stored at different places in the array as the queue grows and shrinks. Next, the programmer must write source code describing the queue data structure and the algorithms in some programming language. In the programming language C the data structure source code might look like this:

program abstraction parts differ substantially from programmers' definitions.

C. *Software Abstraction Parts and the Judiciary*

Analysis of the abstraction part definitions employed by judges in software copyright infringement cases begins with the definitions' somewhat muddled legal context. Because abstraction part definitions spring from substantial similarity tests, section I.C.1 first summarizes in broad terms the importance of substantial similarity to copyright infringement in general, and to computer program part definitions in particular. The section also briefly examines several overlapping and interacting copyright concepts and doctrines as they relate to substantial similarity in software infringement cases: (1) the idea-expression dichotomy; (2) Learned Hand's abstractions test; (3) judicial definitions of computer program parts by abstraction or otherwise; and (4) various traditional copyright doctrines. Section I.C.2 dissects each of the prevailing software substantial similarity tests to reveal their abstraction part definitions, both implicit and explicit, and summarizes the merits of those definitions. This section concludes that the major existing substantial similarity tests fail to provide a coherent framework of computer program part definitions, but do shed light on what such a framework should contain.

```
#define QUEUE_SIZE 50
char queue[QUEUE_SIZE];
int queue_first = -1, /* array indexes locating first and last elements */
    queue_last = -1;
```

The programmer must still make choices about error handling, array initialization, the number of different queue arrays in existence, and other issues to write a source code description of the enqueue( ) algorithm. In C, the enqueue( ) source code might look like this:

```
#define QUEUE_STATUS_OK      0
#define QUEUE_ALREADY_FULL 1
int enqueue( new )
    char new;
{
    int queue_next;
    if (queue_first == -1) { /* adding first element to empty queue */
      queue[0] = new;
      queue_first = queue_last = 0;
      return(QUEUE_STATUS_OK);
    }
    queue_next = (queue_last + 1) % QUEUE_SIZE;
    if (queue_first == queue_next) {
      return (QUEUE_ALREADY_FULL);
    }
    queue[queue_next] = new;
    queue_last = queue_next;
    return(QUEUE_STATUS_OK);
} /* enqueue */
```

This queue source code could now be incorporated in source code for a complete program, and the program could in turn be translated into object code tailored to the desired computer hardware. Only then could a programmer run the program to see whether the queue actually behaves as desired.

1. *Substantial Similarity and Related Analyses*

To prevail, a copyright infringement plaintiff must prove both copyright ownership and copying by the defendant.[75] As eyewitness testimony of copying is rare,[76] plaintiffs often demonstrate copying through "circumstantial evidence of access to the copyrighted work and substantial similarity between the copyrighted work . . . and the allegedly infringing work . . . ."[77] However, access is often conceded or easily proven,[78] so "[i]n most cases the 'substantial similarity' inquiry presents the heart of a copyright infringement case . . . ."[79]

Copyright law's fundamental distinction between idea and expression helps shape tests for substantial similarity. Because ideas[80] are not protected by copyright,[81] similarity between ideas is irrelevant to proof of infringement.[82] In theory, substantial similarity tests compare only the expression in two software programs.[83] However, the

75. Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1231 (3d Cir. 1986), *cert. denied,* 479 U.S. 1031 (1987).

76. *Whelan,* 797 F.2d at 1231.

77. Telemarketing Resources v. Symantec Corp., 1990 Copyright L. Dec. (CCH) ¶ 26,514, at 23,086 (N.D. Cal. Sept. 6, 1989), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir. 1992), *cert. denied,* 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992); *see also* Roth Greeting Cards v. United Card Co., 429 F.2d 1106, 1110 (9th Cir. 1970); NIMMER, *supra* note 13, § 13.01[B].

78. *See, e.g., Whelan,* 797 F.2d at 1232 (noting defendant's access was uncontested, as the program was used in his lab, and as he had acted as a sales representative for plaintiff); Soft Computer Consultants, Inc. v. Lalehzarzadeh, 1989 Copyright L. Dec. (CCH) ¶ 26,403, at 22,538-39 (E.D.N.Y. Aug. 25, 1988) (noting that defendants were former employees of plaintiff); Broderbund Software, Inc. v. Unison World, Inc., 648 F. Supp. 1127, 1136 (N.D. Cal. 1986) (noting that plaintiff gave defendant "several commercially-available copies" of the program); E.F. Johnson Co. v. Uniden Corp. of Am., 623 F. Supp. 1485, 1492 (D. Minn. 1985) (noting that defendant's engineers admitted analyzing code they "dumped" from plaintiff's ROM); SAS Inst., Inc. v. S & H Computer Sys., Inc., 605 F. Supp. 816, 821 (M.D. Tenn. 1985) (noting that defendant received the complete object code and roughly half of the source code under a license agreement); NIMMER, *supra* note 13, § 13.02.

79. Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 558 (E.D.N.Y. 1991), *aff'd.,* 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992).

80. There is vigorous debate over what constitutes an idea in software, but copyright cases uniformly treat a program's purpose or function as an unprotectable idea. *See, e.g.,* Johnson Controls, Inc. v. Phoenix Control Sys., Inc., 886 F.2d 1173, 1175 (9th Cir. 1989); *Telemarketing Resources,* 1990 Copyright L. Dec. (CCH) ¶ 26,514, at 23,087; Pearl Sys., Inc. v. Competition Elecs., Inc., 8 U.S.P.Q.2d (BNA) 1520, 1524-25 (S.D. Fla. 1988); *Broderbund,* 648 F. Supp. at 1134-37.

81. 17 U.S.C. § 102(b) (1988); Mazer v. Stein, 347 U.S. 201, 217-18 (1954).

82. A software infringement defendant often argues that while two programs' ideas may be similar, their expression of those ideas is not. *See, e.g.,* Data East USA, Inc., v. Epyx, Inc., 862 F.2d 204, 207-208 (9th Cir. 1988); Lotus Dev. Corp. v. Paperback Software Intl., 740 F. Supp. 37, 65-67 (D. Mass. 1990); Digital Communications Assocs., Inc. v. Softklone Distrib. Corp., 659 F. Supp. 449, 458-59 (N.D. Ga. 1987).

83. *See* Q-Co Indus., Inc. v. Hoffman, 625 F. Supp. 608, 615 (S.D.N.Y. 1985). *But see* Apple Computer, Inc. v. Microsoft Corp., 779 F. Supp. 133, 136 (N.D. Cal. 1991) (concluding that proper protection of innovative selections or arrangements under the look and feel test, *see infra* note 156, requires consideration of unprotectable parts during substantial similarity analysis). *See generally* NIMMER, *supra* note 13, § 13.03[B][2] (discussing similarity of unprotected matters).

Copyright Act does not define idea or expression,[84] and discordant substantial similarity tests reflect judicial disagreement over where the line between idea and expression should be drawn.[85]

Although Learned Hand's abstractions test[86] does not specify where the idea-expression line lies,[87] refinements of the test that define appropriate levels of abstraction may help courts properly draw the line during a substantial similarity analysis. Three of the four prevailing substantial similarity tests do not incorporate the abstractions test.[88] Two of these tests, however, have been sharply criticized as vague or overly broad,[89] and thus could benefit from the graduated distinctions made possible by the Learned Hand test.[90] Furthermore, the recently adopted successive filtering test, which is the most comprehensive of the four tests, does incorporate the abstractions test.[91] This Note argues that established substantial similarity tests fail to distinguish between levels of abstraction merely because Learned Hand's

84. *See* 17 U.S.C. § 101; John S. Wiley Jr., *Copyright at the School of Patent,* 58 U. CHI. L. REV. 119, 119 (1991).

85. *See, e.g., infra* notes 200-04 and accompanying text.

86. *See supra* notes 22-25 and accompanying text.

87. Learned Hand wrote of the line between expression and idea that "[n]obody has ever been able to fix that boundary, and nobody ever can." Nichols v. Universal Pictures Corp., 45 F.2d 119, 121 (2d Cir. 1930), *cert. denied,* 282 U.S. 902 (1931). Judge Hand echoed this view in a later opinion: "Obviously, no principle can be stated as to when an imitator has gone beyond copying the 'idea,' and has borrowed its 'expression.' Decisions must therefore inevitably be *ad hoc.*" Peter Pan Fabrics, Inc. v. Martin Weiner Corp., 274 F.2d 487, 489 (2d Cir. 1960). If Judge Hand meant that agreement on proper placement of the line between idea and expression is unachievable, the current chaotic state of software copyright law suggests he was correct. *See* Samuelson, *supra* note 2, at 62-66 (discussing policy arguments for "minimalist" and "maximalist" views on software copyright protection). However, even if complete agreement is ultimately impossible to achieve, Judge Hand's abstractions test may still be tailored to at least narrow the range of disagreement in the software copyright realm.

88. *See* Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1236 (3d Cir. 1986) (applying SSO test, *see infra* text accompanying notes 135-50), *cert. denied,* 479 U.S. 1031 (1987); Broderbund Software, Inc. v. Unison World, Inc., 648 F. Supp. 1127, 1134, 1137 (N.D. Cal. 1986) (applying total concept and feel test, *see infra* text accompanying notes 151-62); E.F. Johnson Co. v. Uniden Corp. of Am., 623 F. Supp. 1485, 1493 (D. Minn. 1985) (applying iterative test, *see infra* text accompanying notes 119-34). There is no indication that courts employing the SSO, total concept and feel, or iterative substantial similarity tests deliberately rejected the abstractions test as inherently unusable. Rather, the abstractions test may have been deemed too general in its current form to be of use in software cases. *See* 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, 656 (1988).

89. The tests and the criticisms they have prompted are discussed at length below. *See infra* notes 135-50 and accompanying text (SSO test); *infra* notes 151-62 and accompanying text (total concept and feel test).

90. "The pitfalls of abandoning the abstractions test emerge in sharp focus from the Third Circuit's opinion in *Whelan Associates v. Jaslow Dental Laboratory.*" NIMMER, *supra* note 13, § 13.03[F], at 13-78.33 (footnote omitted); *see also infra* text accompanying notes 135-42 (discussing *Whelan*).

91. *See infra* notes 163-73 and accompanying text. At least one court has also incorporated the abstractions test into its analysis of copyrightability. *See* Lotus Dev. Corp. v. Paperback Software Intl., 740 F. Supp. 37, 60 (D. Mass. 1990).

test has never been appropriately tailored to software, not because there is any benefit in ignoring abstraction parts.

Although this Note focuses on abstraction parts, other parts also arise during substantial similarity analysis. Certain program elements serve *doctrinal* or *evidentiary* roles in assessing misappropriation. A brief discussion of these elements places abstraction parts in context and illustrates the limited nature of this Note's proposed changes. Refining abstraction part definitions will not fundamentally alter the use of doctrinal or evidentiary elements in substantial similarity tests.

Doctrinal program elements are implicitly defined by many traditional copyright doctrines that distinguish between protectable and unprotectable material.[92] For instance, the copyright statute only protects "original" works.[93] This statutory originality requirement divides programs into an unprotectable portion that does not owe its origin to the author, and a potentially protectable portion that does.[94] A single doctrinally defined portion of a program may cut across several levels of abstraction. Suppose a database program incorporates public domain code for sorting names alphabetically. This doctrinally delimited section of the program includes several abstraction parts, namely *code* that implements an *algorithm* for performing a sorting operation on values of a certain *ADT* data type. Recognizing abstraction parts under this Note's proposal will not hinder the application of public domain and other traditional doctrines; only the idea-expression doctrine will be directly affected, and even there the intended result is to clarify rather than to change substantively the characterization of program parts under the doctrine.

Evidentiary elements are defined when courts treat particular pieces of source code as evidence of verbatim or nearly verbatim copying.[95] Programmers develop various distinctive stylistic preferences

92. The *scenes a faire* doctrine, for instance, sets aside as unprotectable any part of a program that is "as a practical matter indispensable, or at least standard, in the treatment of a given topic." Atari, Inc. v. North Am. Philips Consumer Elecs. Corp., 672 F.2d 607, 616 (7th Cir.) (quoting Alexander v. Haley, 460 F. Supp. 40, 45 (S.D.N.Y. 1978)), *cert. denied*, 459 U.S. 880 (1982). Courts applying the *scenes a faire* doctrine in software infringement cases refuse protection to program parts they define as "indispensable" or "inherent." *See, e.g.*, Telemarketing Resources v. Symantec Corp., 1990 Copyright L. Dec. (CCH) ¶ 26,514, at 23,086, 23,087 (N.D. Cal. Sept. 6, 1989), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir. 1992), *cert. denied*, 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992); Q-Co Indus., Inc. v. Hoffman, 625 F. Supp. 608, 616 (S.D.N.Y. 1985). Other copyright doctrines that set aside unprotectable material include independent creation, public domain, originality, and, of course, the idea-expression dichotomy. *See generally* NIMMER, *supra* note 13, § 13.03[F] (discussing doctrinally defined computer program elements). Doctrinally based program elements are discussed further in connection with the successive filtering test. *See infra* notes 163-70 and accompanying text.

93. 17 U.S.C. § 102(a) (1988).

94. *Lotus*, 740 F. Supp. at 47-48.

95. *See, e.g.*, E.F. Johnson Co. v. Uniden Corp. of Am., 623 F. Supp. 1485, 1496-97 (D. Minn. 1985) (noting that both programs contained the same superfluous instructions and identical coding errors). *See generally* Conley & Peterson, *supra* note 42, at 436, 453-67 (listing and discussing evidentiary elements useful in establishing copying).

that are loosely constrained by a program's data structures, algorithms, and other more abstract parts.[96] Similarity in such stylistic choices may therefore serve as evidence of verbatim copying.[97] Under this Note's approach, such stylistic choices often all lie within the source code level of abstraction.[98] Therefore, the proposed framework is independent of evidentiary elements, just as a framework describing novels in terms of plot and character is distinct from font size or typeface choices. Accordingly, adoption of this Note's proposed framework will not change the type of evidence used in software copyright infringement cases.

Evidentiary and doctrinal elements, as well as abstraction parts, play distinct but interrelated roles in software substantial similarity analysis. Evidentiary elements provide proof of copying, while doctrinal elements represent policy decisions about the types of copying that should be prohibited. Abstraction parts — when properly defined — illuminate the relationship between evidentiary elements and the idea-expression doctrine by clarifying which portions of a program are being categorized as idea and which are being treated as expression. This Note addresses abstraction parts rather than doctrinal or evidentiary elements because unclear and contradictory abstraction part definitions lie at the heart of the confusion over software substantial similarity analysis.

Despite the often dispositive role of substantial similarity, most judicial attempts to formulate a test applicable beyond the case at hand have failed. One court listed several "stages of development of a program," including "a definition, in eye-legible form, of the program's task or function; a description; a listing of the program's steps and/or their expression in flow charts;" source code; and object code.[99] The first stage, "task or function," corresponds to section I.B's main purpose. The meaning of the second stage, "description," is unclear. The final three stages correspond respectively to section I.B's algorithms, source code, and object code parts. Large sections of software are overlooked; these five stages apparently omit data structures, ADTs, and system architecture. Likewise, in *Computer Associates International, Inc. v. Altai, Inc.*,[100] the court created an unclear and incom-

96. *Cf.* OGILVIE, *supra* note 36, at 48-50 (describing one set of stylistic preferences).

97. *See generally* Conley & Peterson, *supra* note 42.

98. *See, e.g.*, SAS Inst., Inc. v. S & H Computer Sys., Inc., 605 F. Supp. 816, 822-23 (M.D. Tenn. 1985) (finding copying on the basis of similarities in source code without discussing more abstract parts in detail). Although evidence of verbatim copying typically lies at the source code level of abstraction, nothing inherent in algorithms or other abstraction parts prevents their use as program "signatures" if they are precisely copied.

99. Williams Elecs., Inc. v. Artic Intl., Inc., 685 F.2d 870, 876 n.7 (3d Cir. 1982) (quoting NATIONAL COMMN. ON NEW TECHNOLOGICAL USES OF COPYRIGHTED WORKS, FINAL REPORT 28 (1978)).

100. 775 F. Supp. 544 (E.D.N.Y. 1991), *affd.* 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992).

plete set of abstraction parts. In *Altai,* the court quoted Judge Hand's description of the abstractions test before asserting that when "applied to computer software programs, this abstractions test would progress in order of 'increasing generality' from object code, to source code, to parameter lists, to services required, to general outline."[101] The *Altai* court's object code and source code levels of generality correspond to this Note's parts of the same name.[102] "Parameter lists" are part of the program's interface with other programs, and hence lie outside this Note's scope. "Services required" may also be dictated by the interface, but this part could affect system architecture or ADTs as well. "General outline" seems to mean some combination of section I.B's system architecture and main purpose. But nothing in the *Altai* court's list seems to correspond to data structures or algorithms. Other judicial attempts to formulate a framework of abstraction parts also contain serious flaws.[103]

A third example shows the difficulty in evaluating abstraction parts due to unsettled terminology that complicates the extraction of part definitions from cases. In *Pearl Systems, Inc. v. Competition Electronics, Inc.,*[104] the court referred several times to "the system level design."[105] As defined by the copyright holder's expert, "system level design" corresponds roughly to this Note's notion of system architecture.[106] The court, however, treated "system level design" as equivalent to user interface rather than to system architecture. The court observed that the copyright holder "was able to change the subroutines so that a different sequence of buttons would be used to enter the par time and to engage the shot review function. This resulted from a change in the systems level design of the software."[107] The court continued by saying:

---

101. *Altai,* 775 F. Supp. at 560.

102. *See supra* text accompanying notes 67-72.

103. In Pearl Sys., Inc. v. Competition Elecs., Inc., 8 U.S.P.Q.2d (BNA) 1520 (S.D. Fla. 1988), the court apparently relied on testimony from an expert hired by the ultimately successful copyright holder. According to the expert,

> [T]here were essentially five steps to develop computer software: 1. Functional definition of the product — how it will be used in the marketplace; 2. Systems level design — defining the types of functions for the software to perform and how it will perform those functions; 3. Module design — defining individual portions of the system; 4. Coding — implementing the modules; and 5. Selecting appropriate hardware.

8 U.S.P.Q.2d (BNA) at 1522 n.3. The *Pearl* court's functional definition corresponds to section I.B's main purpose. Systems level design might correspond to section I.B's system architecture, but the court interprets the term to mean user interface. 8 U.S.P.Q.2d (BNA) at 1523-25. Module design might therefore correspond to system architecture, or perhaps to some combination of system architecture and ADTs. Coding comprises algorithms, data structures, and source code, while hardware selection corresponds roughly to section I.B's object code level of abstraction. In short, some *Pearl* abstraction parts are not defined clearly, and others seem excessively broad.

104. 8 U.S.P.Q.2d (BNA) 1520 (S.D. Fla. 1988).

105. *Pearl,* 8 U.S.P.Q.2d (BNA) at 1522 & n.3, 1523, 1525.

106. 8 U.S.P.Q.2d (BNA) at 1522 n.3.

107. 8 U.S.P.Q.2d (BNA) at 1523.

> Moreover, the subroutines in both [plaintiff's and defendant's devices] were triggered by the same sequence of buttons. As there was ample testimony that alternative system level designs could have been used to avoid this similarity, we conclude that the idea did not have only one necessary form of expression, but many.[108]

Even when terminology is clear, however, the software substantial similarity tests conflict with each other,[109] and their definitions correspond poorly with the abstraction parts employed by programmers.[110] Forum shopping,[111] expensive "clean room" program development,[112] and other undesirable consequences[113] follow from deficient part definitions. Furthermore, the difficulty of formulating legally useful and technically accurate program part definitions has apparently led to despair as well as disarray. One group of courts has seemingly abandoned even nominal pursuit of comprehensive program part definitions, essentially ignoring all but evidentiary elements.[114] Another group simply refuses to dissect programs at all.[115] The following subsection explores judicial efforts to analyze substantial similarity

---

108. 8 U.S.P.Q.2d (BNA) at 1524-25.

109. *See, e.g., infra* note 200 and accompanying text.

110. *Compare supra* section I.B (discussing parts recognized by programmers) *with* Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1239 (3d Cir. 1986) (discussing "sequence," "order," and "structure" as program parts), *cert. denied,* 479 U.S. 1031 (1987).

111. Copyright infringement plaintiffs naturally prefer to bring suit in a circuit that enforces the broadest protection available. The Third Circuit, home of the *Whelan* decision, *see infra* notes 135-42 and accompanying text, is therefore particularly attractive to such plaintiffs, while defendants should prefer the Second Circuit or the Fifth Circuit, which have directly repudiated the broad rule of *Whelan. See* Computer Assocs. Intl., Inc. v. Altai, Inc., 23 U.S.P.Q.2d (BNA) 1241, 1252 (2d Cir. 1992); Plains Cotton Coop. Assocs. v. Goodpasture Computer Serv., Inc., 807 F.2d 1256, 1262 (5th Cir. 1987), *cert. denied,* 484 U.S. 821 (1987).

112. Under a clean room procedure, people work in two groups to avoid infringement of program *X* when creating program *Y.* One set of people, with access to *X,* describes *X* in terms abstract enough to be deemed ideas rather than expression for copyright purposes. These descriptions are then provided to the second group of people, programmers who create a working program *Y* that meets the desired description. The programmers have no access to *X*'s expression while creating *Y,* so any similarities of expression between *X* and *Y* are noninfringing. *See, e.g.,* NEC Corp. v. Intel Corp., 1989 Copyright L. Dec. (CCH) ¶ 26,379, at 26,390 (N.D. Cal. Feb. 6, 1989) (discussing clean room set up to demonstrate that hired party, even without access, reproduced expression in copyrighted software); *cf.* Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 554 (E.D.N.Y. 1991) (discussing rewrite of allegedly infringing program), *affd.* 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992).

113. Professor Samuelson has stated: "Teaching software copyright law in the United States is at present a perilous endeavor. . . . When a whole field of law is a welter of confusion and contradiction, it is no small challenge to teach the law as it truly is and keep students' attention and respect." Samuelson, *supra* note 2, at 61, 71.

114. As applied in E.F. Johnson Co. v. Uniden Corp. of Am., 623 F. Supp. 1485 (D. Minn. 1985), the iterative test for substantial similarity only prohibits literal copying or translation of computer program code. *See infra* notes 124-25 and accompanying text. Therefore, comparison of ADTs or any of the other more abstract software parts is ignored and there is no reason to define any abstraction part other than code. *See also* SAS Inst., Inc. v. S & H Computer Sys. Inc., 605 F. Supp. 816, 822, 829 (M.D. Tenn. 1985) (finding infringement on the basis of similarities in source code without extensively discussing any levels of abstraction other than idea and expression).

115. *See infra* notes 151-62 and accompanying text.

and compares the computer program parts these efforts explicitly or implicitly recognize with the parts recognized by programmers.

2. *Current Substantial Similarity Tests*

Software copyright cases utilize four major substantial similarity tests: (1) the iterative test; (2) the structure, sequence, and organization (SSO) test; (3) the "look and feel" or "total concept and feel" test; and (4) the successive filtering test. Each substantial similarity test defines program abstraction parts differently; these definitions are often implicit[116] or unclear.[117] At best, the definitions blur the levels of abstraction proposed in section I.B; at worst, the levels are ignored altogether.[118]

a. The iterative test. Originally derived from a student law review Note,[119] the iterative test for substantial similarity was first formally adopted by a court in *E.F. Johnson Co. v. Uniden Corp. of America.*[120] The test has two prongs, but only the second prong is relevant to abstraction part definitions. The first prong merely asks whether "the defendant 'used' the copyrighted work in preparing the alleged copy, which may be established by proof of access and similarity sufficient to reasonably infer use of the copyrighted work . . . ."[121] This prong "amounts to little more than a variation of the traditional substantial similarity analysis" and so adds nothing to the search for proper program part definitions.[122] The iterative test's second prong, however, asks whether "the defendant's work is an iterative reproduction, that is, one produced by iterative or exact duplication of substantial portions of the copyrighted work."[123] The *Uniden* court interpreted this prong of the iterative test as a prohibition against literally copying[124] or literally translating[125] code.

The iterative test thus divides programs into protected literal code,

---

116. *See, e.g., infra* notes 128-29.

117. *See supra* text accompanying notes 104-08; *see also infra* notes 126-34, 162 and accompanying text.

118. The total concept and feel test, *infra* notes 151-62 and accompanying text, apparently bundles every level of abstraction into the program's "total concept and feel," and the SSO test, *infra* notes 135-50 and accompanying text, bundles together system architecture, ADTs, algorithms and data structures.

119. Howard Root, Note, *Copyright Infringement of Computer Programs: A Modification of the Substantial Similarity Test,* 68 MINN. L. REV. 1264, 1294-1302 (1984).

120. 623 F. Supp. 1485 (D. Minn. 1985). The *Uniden* court asserted that "the iterative approach [has been] adopted in form if not name by several courts . . . ." 623 F. Supp. at 1493.

121. *Uniden,* 623 F. Supp. at 1493.

122. Nimmer et al., *supra* note 88, at 634.

123. 623 F. Supp. at 1493.

124. The court found liability for "iterative or verbatim reproduction of substantial sections of [plaintiff's] code, *i.e.,* the data tables and 38 of 44 subroutines." 623 F. Supp. at 1497 n.10.

125. 623 F. Supp. at 1497-98; *see also* Nimmer et al., *supra* note 88, at 634.

protected literal translations of code, and the unprotected remainder of the program. *Uniden*'s literal object code and source code parts clearly correspond to source code and object code as defined above. The correspondence between literal translations and this Note's definitions is less clear, but it seems limited to the three possibilities shown in Figure 2. The translations *Uniden* speaks of are not translations from source code into object code, but rather translations of a program from one programming language into another.[126] Generally speaking, a translation of a program from one language to another may require changes in data structures, algorithms, or even more abstract program parts.[127] The first possibility shown in Figure 2 matches literal translations to algorithms and data structures because protection of literal translations seems to require protection of algorithms and data structures that differ from those of the original program only in ways required by translation. Furthermore, even though the *Uniden* court did not speak in terms of algorithms and data structures, the court's protection of a "Barker code" algorithm[128] and an "H-matrix" data structure[129] suggest that algorithms and data struc-

126. 623 F. Supp. at 1497 (discussing "literal translation of plaintiff's Intel instructions into Hitachi language").

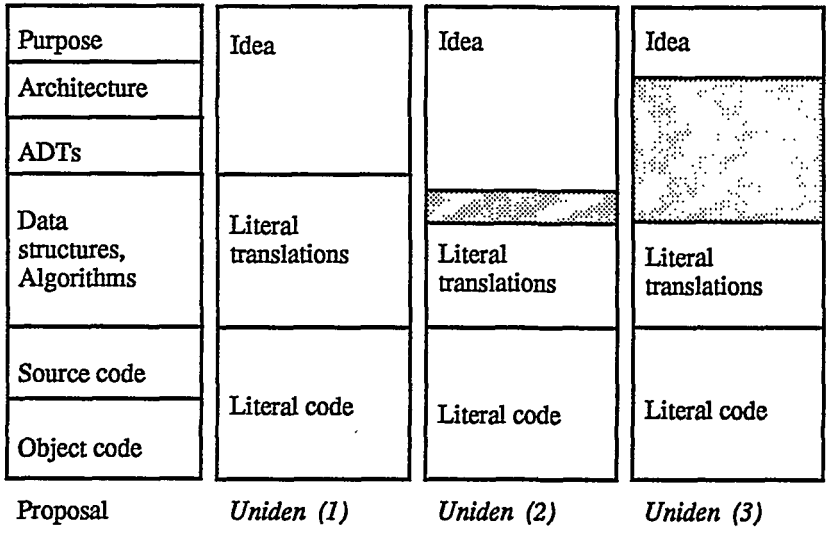
127. The C programming language supports pointers, while the FORTRAN language does not, so any program containing data structures built with pointers cannot be translated from C into FORTRAN without changing those data structures. Similarly, C permits subroutines to call themselves "recursively" while FORTRAN does not, so recursive algorithms cannot be implemented directly if one translates from C into FORTRAN. *See generally* OGILVIE, *supra* note 48, at 16-19 (comparing FORTRAN, C, and several other programming languages). Similar problems arise during translation between other programming languages. Extreme hardware changes may also require corresponding changes at very abstract software levels. *See generally* OGILVIE, *supra* note 36, at 332-39 (cataloging problems that may arise in "porting" a program from one system to another).

128. "A 'Barker code' is a pattern of ones and zeroes alternated in a prepatterned sequence. Both the sending and receiving units must identify the Barker code in order for communication to be established." 623 F. Supp. at 1494. Although *Uniden* does not speak in terms of ADTs, programmers would recognize in a Barker code an ADT in which the data type is a pattern of ones and zeroes, where the most prominent operations are obtaining another pattern and comparing the second pattern with the ADT pattern to see if they match. Although the defendant in *Uniden* had to copy this ADT to achieve compatibility, the court found no infringement in such copying. 623 F. Supp. at 1494. But the defendant did infringe by copying the particular sampling algorithm used in the copyrighted program to obtain patterns for comparison; notably, a different algorithm would have been more efficient on the defendant's hardware. 623 F. Supp. at 1494-95.

129. An H-Matrix is a series of ones and zeroes arranged in rows and columns in a matrix format. An H-Matrix is used . . . to detect errors . . . once communication has been established by matching of Barker codes. To make its radios compatible . . . [defendant] was required to and did employ *some form* of H-matrix in its software program . . . .

623 F. Supp. at 1495. Although *Uniden* does not speak in terms of ADTs, programmers would recognize an H-matrix as an ADT wherein the data type is a pattern of ones and zeroes that meets certain technical constraints; notably, any one of 32 different patterns satisfies these constraints. 623 F. Supp. at 1495. The copyrighted program's H-matrix ADT was implemented by two data structures. One was a particular matrix of ones and zeroes, and the other data structure was this matrix's inverse. 623 F. Supp. at 1495. The defendant's program infringed by precisely copying both data structures when the inverse matrix was superfluous and any other of the matrix's 32 configurations would have worked. 623 F. Supp. at 1495.

FIGURE 2
Possible Correspondence Between *Uniden* and Proposed Abstraction Parts.



tures lie within a protected level of abstraction; algorithms and data structures are clearly more abstract than code, so they must lie in the level occupied by literal translations.

Figure 2 also shows a second possibility, in which literal translations cover only part of the range of abstraction covered by algorithms and data structures. Literal translation is not rigorously defined in *Uniden,* but much of the opinion's language speaks of verbatim copying.[130] Algorithms and data structures embedded in a piece of source code can often be copied without duplicating or even approximating the source code's style; avoiding similarity in source code wording while actually copying functionality is often easy.[131] *Uniden*'s emphasis on protecting "literal" program parts might therefore be interpreted to prohibit use of data structures or algorithms that substantially replicate the original author's efforts, both in terms of

130. *See, e.g.,* 623 F. Supp. at 1497 (holding that verbatim copying is inferential evidence of pirating; both programs contained identical sample error tables and superfluous instructions, and 38 out of 44 subroutines were identical).

131. Extreme examples of programs that behave identically but look very different may be found in the annals of the recreational International Obfuscated C Coding Contest. The goal of this contest is to write a clever working program whose purpose is impossible to discern from its source code. Programmers begin with understandable source code but obfuscate it step-by-step until they are satisfied no one else can decipher it. The original program and the final obfuscated version run identically, but their respective source codes typically look very different indeed. THE NEW HACKER'S DICTIONARY 265-66 (Eric S. Raymond ed., 1991).

substance and in terms of programming style.[132] This narrower definition of literal translation protects more than the literal source code, but covers less than the entire range of abstraction covered by data structures and algorithms.

The third possibility shown in Figure 2 is that literal translations encompass data structures, algorithms, and more abstract parts as well. *Uniden*'s discussion of literal translation is shaped by *Whelan Associates v. Jaslow Dental Laboratory, Inc.*,[133] a case that treats everything except a program's purpose as protectable expression.[134] Without more guidance, however, it is unclear whether any of these three interpretations is correct.

The iterative test is unsuitable, therefore, because it does not clearly define any abstraction parts other than literal code. Furthermore, even if the definition of literal translations could be clarified, the iterative test would still destabilize software copyright law by recognizing too few levels of abstraction. The literal code component bundles together source code and object code, even though they are clearly discrete abstraction parts. Similarly, *literal translations* and *idea* attempt to span four levels of abstraction with only two components. As the history of the SSO test discussed next illustrates, such overly broad definitions too often lead subsequent courts to define additional abstraction parts in conflicting ways.

b. The structure, sequence and organization test. The structure, sequence, and organization (SSO) test was formulated and first applied in *Whelan Associates, Inc. v. Jaslow Dental Laboratory, Inc.*[135] *Whelan* stated 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."[136] In other words, protection may be available for every part of a program except its single overriding purpose.[137]

The SSO test thus defines programs as having a purpose part and an SSO part. As illustrated in Figure 3, *Whelan*'s purpose part corresponds to section I.B's main purpose.[138] SSO corresponds to system

132. *See generally* Conley & Peterson, *supra* note 42, at 453-67 (discussing stylistic clues to copying or derivation).

133. 797 F.2d 1222 (3d Cir. 1986), *cert. denied*, 479 U.S. 1031 (1987). The *Uniden* court looked to *Whelan* for guidance because in its words "similar considerations control." 623 F. Supp. at 1497.

134. *See infra* notes 135-37 and accompanying text.

135. 797 F.2d 1222 (3d Cir. 1986), *cert. denied*, 479 U.S. 1031 (1987).

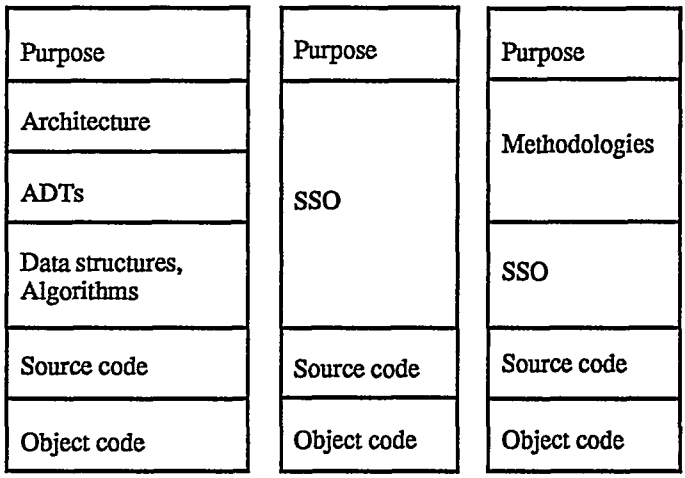
136. *Whelan*, 797 F.2d at 1236 (original emphasis removed).

137. *See* 797 F.2d at 1238 (holding that the purpose of plaintiff's program, "to aid in the business operations of a dental laboratory," is an idea and hence unprotected).

138. *Whelan* suggests in dicta that in other cases a program's purpose "may be to accomplish a certain function *in a certain way*," but does not pursue this alternate definition of purpose. 797 F.2d at 1238 n.34.

architecture, ADTs, algorithms, and data structures,[139] because *Whelan* also discusses source code and object code. Like *Uniden, Whelan* does not purport to apply Learned Hand's test, so the correspondences with abstraction parts shown here were deduced mainly from the case as a whole rather than any explicit definitions.

FIGURE 3
Correspondence Between Proposed Abstraction Parts and Parts Defined in *Whelan* and *Healthcare*.[140]



Whelan's part definitions are poorly adapted to the abstractions test because they fail to distinguish different levels of abstraction within a program's SSO. *Uniden* and other cases draw the idea-expression line inside the SSO, suggesting that the range of abstraction encompassed by SSO is too broad.[141] "The crucial flaw in [*Whelan*'s] reasoning is that it assumes that 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."[142]

139. 797 F.2d at 1224-25, 1230-31. The *Whelan* court also considered several program parts that either lie within SSO or lie outside this Note's scope. The court expressly compared several program subroutines, pieces of code that accept data from another part of the program, perform some relatively small piece of the program's work, and pass data back out to the main program. Like an entire program, a subroutine may be viewed either as its literal source code text or as the embodiment of several levels of abstraction, so recognizing subroutines as another level of abstraction would be redundant. *Cf. supra* note 72. The *Whelan* court also considered similarity of file structures and screen outputs, 797 F.2d at 1242-48, but these and all other parts of a program's user interface lie outside this Note's scope. *Cf. supra* note 73 (discussing user interface abstraction parts).

140. Healthcare Affiliated Services, Inc. v. Lippany, 701 F. Supp. 1142 (W.D. Pa. 1988).

141. *See, e.g., infra* note 144 and accompanying text.

142. Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 559 (E.D.N.Y. 1991)

Parts, such as SSO, that cover too much of the abstraction spectrum compel courts to define additional parts in subsequent cases. In *Healthcare Affiliated Services, Inc. v. Lippany,*[143] the court nominally followed *Whelan* but actually narrowed the breadth of SSO by defining a new abstraction part, *methodologies.* The *Healthcare* court treated methodologies as idea rather than expression[144] by denying them copyright protection. Unfortunately, *Healthcare* did not expressly define methodology, so the part may be difficult to recognize in subsequent cases. The examples[145] provided in the *Healthcare* opinion and the court's reasoning[146] seem to equate methodologies with system architecture and ADTs, as shown in Figure 3, but other correlations are not ruled out. Courts also nominally recognized SSO in both *Johnson Controls, Inc. v. Phoenix Control Systems, Inc.,*[147] and *Telemarketing Resources v. Symantec Corp.,*[148] but actually focused on more specific program parts and declined to apply the SSO test.[149]

In short, the SSO test is unsuitable because overly broad parts such as SSO require subsequent courts to define additional parts: such narrowing definitions are, unfortunately, typically concerned with the characteristics of a particular program rather than with the widely recognized components of programs in general. These additions to the growing collection of judicially recognized program parts therefore make poor candidates for a generally applicable abstractions frame-

---

(quoting NIMMER, *supra* note 13, § 13.03[F], at 13-78.34), *affd.,* 23 U.S.P.Q.2d 1241 (2d Cir. 1992).

143. 701 F. Supp. 1142 (W.D. Pa. 1988).

144. *Healthcare,* 701 F. Supp. at 1152.

145. 701 F. Supp. at 1152 (noting that methodologies may be "engineered standards-based" or "hospital comparison-based," may use "multivariable" or other sets, and may determine costs by procedure, "by department, by patient or by product-line").

146. Methodologies are not merely aspects of a program's purpose because there is a serious question "whether these methodologies constitute 'expression' . . . ." 701 F. Supp. at 1151. On the other hand, the *Healthcare* court considered methodologies more abstract than source code because the court noted that "[n]o evidence . . . was presented to indicate how the choices among these alternatives, *i.e.,* the methodologies, would translate into 'a set of statements or instructions' which could be used in a computer . . . ." 701 F. Supp. at 1152.

147. 886 F.2d 1173 (9th Cir. 1989).

148. 1990 Copyright L. Dec. (CCH) ¶ 26,514 (N.D. Cal. Sept. 6, 1989), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir. 1992), *cert. denied,* 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992).

149. The *Johnson Controls* court held that a program contains "several different components, including the source and object code, the structure, sequence and/or organization of the program, the user interface, and the function, or purpose of the program." 886 F.2d at 1175 (footnotes omitted). The court found infringement of the copyrighted work's total concept and feel on the basis of various similarities, both in idea and expression, set forth in detail in a special master's report but not clearly described in the reported opinion. 886 F.2d at 1176. Citing *Whelan,* the *Telemarketing Resources* court held that copyright protection applied "to the user interface, or overall structure and organization of a computer program, including its audiovisual displays, or screen 'look and feel.' " 1990 Copyright L. Dec. (CCH) at 23,085. The program parts addressed during the court's substantial similarity analysis under the look and feel test included menu screen options, pull down windows, a menu bar, an editing screen, and default color selections. 1990 Copyright L. Dec. (CCH) at 23,086-89.

work.[150] Rather than defining additional parts, however, some courts have chosen to define no parts at all, as the next test illustrates.

c. The "total concept and feel" test and the "look and feel" test. Unlike the iterative and SSO tests for substantial similarity, the "total concept and feel" test did not arise in a software infringement case, but emerged rather from a case concerning greeting cards.[151] Subsequent cases involving juvenile books[152] and a children's television program[153] further developed the test, which finally came to software cases by way of video game infringement suits.[154] This history suggests that the total concept and feel test might be poorly suited to the analysis of software infringement, and such is indeed the case.

The total concept and feel test apparently defines no program parts at all. Instead, the test finds substantial similarity if the allegedly infringing work captures the copyrighted work's total concept and feel.[155] A closely related test compares two works' "look and feel."[156] The feel tests are so situation-dependent that any general statement of how they work is necessarily inaccurate. Feel, as one court put it, "is a conclusion . . . . Thus, in trying to understand the relevance of 'concept and feel' precedents, we need to look to details of those cases that appear to have been relied upon in reaching the conclusion, rather than merely embracing the conclusion without regard for underlying reasons."[157]

Examination of particular feel cases is unnecessary, however, because both feel tests suffer from fundamental flaws that render them

---

150. To rephrase the problem bluntly, all programs contain SSO, but protecting SSO protects too much and therefore the SSO must be split. Breaking methodologies out of SSO only helps in the relatively few programs that are built around methodologies. Recognizing system architecture, ADTs, and so forth, on the other hand, helps considerably because all programs contain those parts.

151. Roth Greeting Cards v. United Card Co., 429 F.2d 1106 (9th Cir. 1970). *See generally* NIMMER, *supra* note 13, § 13.03[A][1][c] (discussing the history of the total concept and feel test).

152. Reyher v. Children's Television Workshop, 533 F.2d 87 (2d Cir.), *cert. denied,* 429 U.S. 980 (1976).

153. Sid & Marty Krofft Television Prods., Inc. v. McDonald's Corp., 562 F.2d 1157 (9th Cir. 1977).

154. *See, e.g.,* Atari, Inc. v. North Am. Philips Consumer Elecs. Corp., 672 F.2d 607, 619-20 (7th Cir.), *cert. denied,* 459 U.S. 880 (1982); Atari, Inc. v. Amusement World, Inc., 547 F. Supp. 222, 228-30 (D. Md. 1981).

155. *See, e.g.,* Broderbund Software, Inc. v. Unison World, Inc., 648 F. Supp. 1127, 1134, 1137 (N.D. Cal. 1986).

156. *See, e.g.,* Apple Computer, Inc. v. Microsoft Corp., 779 F. Supp. 133 (N.D. Cal. 1991); Telemarketing Resources v. Symantec Corp., 1990 Copyright L. Dec. (CCH) ¶ 26,514 (N.D. Cal. Sept. 6, 1989), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir. 1992), *cert. denied,* 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992); *see also* Lotus Dev. Corp. v. Paperback Software Intl., 740 F. Supp. 37, 62-63 (D. Mass. 1990) (discussing history and applications of look and feel test and total concept and feel test in *Roth Greeting Cards* and elsewhere).

157. *Lotus,* 740 F. Supp. at 63.

practically useless in an abstractions test. Software cases have developed the feel tests primarily in disputes over the similarity of user interfaces;[158] because user interfaces are by definition meant to be used and understood by nonprogrammers, this legal context provides little guidance in analyzing internal program components such as ADTs, algorithms, and data structures which are normally seen only by programmers. Both feel tests have also been appropriately criticized as vague and overly broad.[159] Moreover, the copyright statute expressly forbids protection of concepts.[160] Courts sometimes adapt the tests by focusing on specific program parts, but different courts discuss different parts,[161] and some of the parts introduced in an attempt to avoid vagueness are themselves poorly defined.[162] In sum, although the feel tests may be appropriate for assessing the similarity of greeting cards, they are wholly unsuitable for determining levels of abstraction in software infringement cases.

d. The successive filtering test. The perceived shortcomings of the iterative, SSO, and total concept and feel tests for substantial similarity spurred the adoption of a successive filtering test.[163] The test

158. *See, e.g.,* Atari, Inc. v. North Am. Philips Consumer Elecs. Corp., 672 F.2d 607 (7th Cir. 1982) (finding video games substantially similar under total concept and feel test); *Apple,* 779 F. Supp. 133 (discussing look and feel test in dispute over graphic user interfaces); *Telemarketing Resources,* 1990 Copyright L. Dec. (CCH) ¶ 26,514 (applying look and feel test to screen displays); *Broderbund Software,* 648 F. Supp. at 1137 (applying total concept and feel test to programs' audiovisual displays). *But see* Johnson Controls, Inc. v. Phoenix Control Sys., Inc., 886 F.2d 1173, 1175 n.3, 1176 (9th Cir. 1989) (applying total concept and feel test without enumerating pertinent program parts).

159. *See, e.g., Lotus,* 740 F. Supp. at 60 (asserting that look and feel concept, standing alone, is not significantly helpful in analyzing copyrightability). Commentators have also criticized the feel tests. "[T]he addition of 'feel' to the judicial inquiry, being a wholly amorphous referent, merely invites an abdication of analysis." NIMMER, *supra* note 13, § 13.03[A] at 13-37. "It may, conceivably, make sense to refer to the 'total concept and feel' of a greeting card or game or anthropomorphic fantasy world; the words lose their meaning, however, as applied to source or object code." Nimmer et al., *supra* note 88, at 633 (footnote omitted). " 'Look' does seem a safer word [because the copyright statute specifically states that 'concepts' are not protectable], though it has the same virtue for plaintiffs as the 'total concept and feel' test: a vagueness about what might be within its scope." Samuelson, *supra* note 2, at 69.

160. 17 U.S.C. § 102(b) (1988).

161. *Compare Telemarketing Resources,* 1990 Copyright L. Dec. (CCH) ¶ 26,514 at 23,088-89 (discussing menu screen options, pull down windows, menu bar, editing screen, and default color selections) *with Broderbund Software,* 648 F. Supp. at 1137 (discussing sequence of screens and choices presented, screen layout, and method of feedback).

162. In Accolade, Inc. v. Distinctive Software, Inc., 1990 Copyright L. Dec. (CCH) ¶ 26,612, at 23,627-28 (N.D. Cal. June 17, 1990), the program parts discussed by the court include "concept design." This enigmatic term apparently arose from language in the parties' licensing agreement that defined the "licensed product" as "the concepts to be designed and implemented by the developer." 1990 Copyright L. Dec. (CCH) at 23,627. The range of abstraction spanned by concept design is unclear. Concept design cannot overlap main purpose, because the court treated "the concept and design of the video game" as copyrightable, 1990 Copyright L. Dec. (CCH) at 23,627, and ideas are clearly not copyrightable. 17 U.S.C. § 102(b). Nor is concept design another term for user interface because the court explicitly contrasts "concept" with "look and feel." 1990 Copyright L. Dec. (CCH) at 23,627.

163. Computer Assocs. Intl., Inc. v. Altai, Inc., 23 U.S.P.Q.2d (BNA) 1241, 1253 (2d Cir.

was originally suggested by Professor Nimmer and his colleagues:

> To [evaluate substantial similarity] an allegedly infringed program should be analyzed on several different levels. A different copyright doctrine is applied at each level, and material which is unprotectable under that doctrine is excluded from further consideration in analyzing substantial similarity. By successively filtering out unprotectable material, a core of protected material remains against which the court can compare the allegedly infringing program.[164]

Successive filtering therefore recognizes that complex software is best analyzed for similarity by partitioning it appropriately. However, separating programs into doctrinal levels is not equivalent to adopting an abstraction parts analysis.

Many traditional copyright doctrines have little bearing on the idea-expression distinction. For example, *scenes a faire*[165] analysis under successive filtering denies copyright protection to those portions of a program that "follow naturally from the work's theme rather than from the author's creativity."[166] Suppose a program must accept information stored on noncopyrightable paper forms and add the information to a computer database. The data structure *A* into which values are initially read may well be substantially dictated by the paper forms, but the final destination of the values, data structure *B* in the database, is largely independent of the various values stored. Under *scenes a faire, A* is therefore denied protection while *B* is not.[167] By distinguishing between data structures, which all lie in a single level of abstraction, *scenes a faire* analysis creates a distinction based on concerns other than level of abstraction. Parts defined by the *scenes a faire* doctrine therefore do not belong in any refinement of Learned Hand's abstractions test. Other doctrines applied during successive filtering, including lack of originality, independent creation, and fair use, similarly ignore program part definitions that are based on level of

---

1992); Autoskill Inc. v. National Educ. Support Sys., Inc., 793 F. Supp. 1557, 1568-71 (D.N.M. 1992). Other courts have also endorsed approaches resembling successive filtering. *See* Brown Bag Software v. Symantec Corp., 960 F.2d 1465, 1475 (9th Cir.) (endorsing "analytic dissection" of computer programs to isolate protectable expression), *cert. denied,* 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992); Apple Computer, Inc. v. Microsoft Corp., 779 F. Supp. 133, 135 (N.D. Cal. 1991) ("Some dissection of elements and the application of merger, functionality, *scenes a faire,* and unoriginality theories are necessary to determine which elements can be used freely by the public in creating new works, so long as those works do not incorporate the same selection or arrangement as that of the plaintiff's work.").

164. Nimmer et al., *supra* note 88, at 635 (footnotes omitted); *see also* NIMMER, *supra* note 13, § 13.03[F].

165. *See supra* note 92.

166. NIMMER, *supra* note 13, § 13.03[F][3].

167. Under other analyses, of course, results may differ. Suppose many different choices exist for the internal representation *B* in theory, but efficiency concerns render all but one or two possibilities impractical. Idea and expression may then be said to have merged, and *B* will be denied copyright protection under the merger doctrine. *See generally* NIMMER, *supra* note 13, §§ 13.03[B][3], 13.03[F][2].

abstraction.[168]

Successive filtering in its present form is too broad and undeveloped to provide adequate abstraction part definitions. Although abstraction is central to the idea-expression dichotomy,[169] the other traditional doctrines successive filtering invokes[170] are based on policies that find no clear reflection in levels of abstraction. While successive filtering may certainly build on a prior abstraction analysis,[171] it may not replace that narrower analysis. Unfortunately, the abstraction analysis performed under the recently adopted successive filtering test utilizes definitions that either are tailored too closely to a specific type of program[172] or drawn too vaguely to identify distinct levels of abstraction.[173]

Vague and incongruous program part definitions in existing substantial similarity tests cripple current judicial efforts to adapt Learned Hand's abstractions test to software copyright infringement cases. A stable set of part definitions could narrow the range of disagreement over where to draw the idea-expression line, as well as discourage the present practice of defining new, inconsistent parts on a case-by-case basis. The following Part proposes a standard for evaluating the suggested definitions.

---

168. *See generally* NIMMER, *supra* note 13, § 13.03[F].

169. From its inception, the abstractions test has been a method for separating idea from expression. Nichols v. Universal Pictures Corp., 45 F.2d 119, 121 (2d Cir. 1930), *cert. denied*, 282 U.S. 902 (1931).

170. Under successive filtering, traditional copyright doctrines are somewhat modified to conform with software concerns. "[T]he merger doctrine should be applied to deny protection to those elements of a program dictated purely by efficiency concerns." NIMMER, *supra* note 13, § 13.03[F][2], at 13-78.36. Professor Nimmer also suggests modifying the traditional doctrines of *scenes a faire* and lack of originality to consider hardware and software standards. *Id.* § 13.03[F][3]. Such modifications properly recognize that programming realities should shape software copyright law.

171. *See, e.g.*, Computer Assocs. Intl., Inc. v. Altai, Inc., 23 U.S.P.Q.2d (BNA) 1241, 1252 (2d Cir. 1992) (arguing that "district courts would be well-advised to undertake a three-step procedure" consisting of an abstractions test adapted to computer programs, successive filtering, and comparison).

172. The district court in *Altai* recognized levels of abstraction such as "parameter lists" and "services required" that were closely tailored to the program at issue. Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 560 (E.D.N.Y. 1991). On appeal, the Second Circuit recognized that even though the levels of abstraction employed by the district court were "workable," different levels might be required in other cases. 23 U.S.P.Q.2d (BNA) at 1259; *see also* Autoskill, Inc. v. National Educ. Support Sys., Inc., 793 F. Supp. 1557, 1566 (D.N.M. 1992) (apparently recognizing "skill levels" in educational software as a level of abstraction).

173. In *Altai*, the Second Circuit adopted a very general description of the levels of abstraction in computer programs:

> At the lowest level of abstraction, a computer program may be thought of in its entirety as a set of individual instructions organized into a hierarchy of modules. At a higher level of abstraction, the instructions in the lowest-level modules may be replaced conceptually by the functions of those modules. At progressively higher levels of abstraction, the functions of higher-level modules conceptually replace the implementations of those modules in terms of lower-level modules and instructions, until finally, one is left with nothing but the ultimate function of the program.

23 U.S.P.Q.2d (BNA) at 1253 (quoting Englund, *supra* note 17, at 897-98).

II. Properly Defining Program Abstraction Parts

This Part argues for judicial adoption of the abstraction part definitions presented in section I.B. Section II.A discusses requirements any set of definitions should satisfy and establishes that the proposed definitions meet these requirements. Section II.B discusses the merits of change, arguing that consensus on the appropriate scope of copyright protection for software is impossible without judicial agreement on a coherent set of abstraction part definitions. The Part concludes that courts should adopt this Note's abstraction part definitions or an equivalent set of definitions, instead of masking policy decisions and ignoring programming realities.

A. *Criteria for Defining Abstraction Parts*

This section presents requirements that any set of abstraction part definitions used in software copyright infringement cases should satisfy. Briefly stated, the definitions should divide any program into a manageable and complete set of stable, nonoverlapping abstraction parts that defer appropriately to existing law and accepted programming concepts. Any attempt to provide a stable definitional foundation for software substantial similarity analysis should meet the standard defined by these requirements. The section reconsiders the part definitions proposed in section I.B and argues that they satisfy these requirements, unlike many of the part defined by existing substantial similarity tests.

The primary purpose of any set of part definitions is, of course, to minimize ambiguity. When program part definitions partition a range of abstraction into smaller parts, ambiguity may manifest itself as unstable borders between parts. The importance of firmly established abstraction part boundaries to legal consistency is evident from the forgoing discussion. Hence *expression,* as defined by the substantial similarity tests examined in section I.C, is unsuitable as a program part because its boundaries shift widely from case to case. Other than purpose and code, the other parts defined by existing substantial similarity tests have also proven unstable.[174]

The abstraction parts proposed in section I.B, however, are defined in terms of stable software engineering characteristics of the various levels of abstraction.[175] These characteristics of software do not de-

174. The clearest evidence that the parts defined by a given substantial similarity test are unstable is, of course, the adoption by later courts of a different test, either with or without reference to the earlier test. *See, e.g., Altai,* 775 F. Supp. at 559 (refusing to adopt SSO test); *see also infra* note 198 and accompanying text (listing abstraction parts defined by cases discussed in this Note; several of these parts, including concept design, general outline, methodologies, parameter lists, services required, and system level design appear in only one case).

175. For example, a characteristic of a program that is changed during translation into object code must be source code, and a characteristic that describes data arrangements in terms of arrays or pointers is a data structure. *See supra* section I.B and *infra* note 187.

pend on legal policy concerns, and therefore persist unaltered throughout many different fact situations. Basing part definitions on established programming concepts[176] also has the advantage of narrowing the gap between copyright law and computer science; because the abstraction level boundaries in any particular program are drawn in practice through expert testimony,[177] part definitions that make sense to programmers are desirable.

Although stable part boundaries help eliminate ambiguity, they are not sufficient because ambiguity may also occur when boundaries overlap. Levels of abstraction should be distinct from one another,[178] but conflicting approaches by different courts have created a set of program parts that overlap each other in a multitude of confusing ways. As a result, SSO overlaps literal code,[179] and methodologies apparently overlaps algorithms and literal translations.[180] User interface spans the entire range of abstraction,[181] as does total concept and feel,[182] and therefore each of these overlaps every other program part. Courts could attempt to eliminate overlapping part boundaries by selecting parts from among those already defined to obtain a set of distinct parts. However, most such parts are defined in conclusory legal terms, not according to independently grounded programming criteria, so they are subject to drift in future cases. Moreover, if parts are selected from several substantial similarity tests, they might not fit together neatly because they arose in, and are tailored to, different factual contexts. A set of parts S consisting of *Uniden*'s literal code,

176. Use of established programming concepts in formulating definitions, although necessary, is not sufficient. While the "hierarchy of modules" recently recognized by the Second Circuit is compatible with fundamental principles of software organization, *see* Computer Assocs. Intl., Inc. v. Altai, Inc., 23 U.S.P.Q.2d (BNA) 1241, 1253 (2d Cir. 1992) (quoting Englund, *supra* note 17, at 897-98), the module hierarchy description speaks merely in conclusory terms ("lower-level" and "higher-level" modules), omitting the working details that are necessary to make different levels of abstraction identifiable and distinct. *See supra* section I.B.

177. *See, e.g.*, SAS Inst., Inc. v. S & H Computer Sys., Inc., 605 F. Supp. 816, 821 (M.D. Tenn. 1985) (utilizing court-appointed expert). *See generally* NIMMER, *supra* note 13, § 13.03[E][4] (noting the role of expert testimony on substantial similarity in software copyright cases).

178. As an example of the problems overlapping part definitions may cause, suppose that one court characterizes the construction X in a program as part of the user interface while a second court characterizes a virtually identical X in another program as code. The first court might well apply an "ordinary observer" standard during substantial similarity analysis, while the second court might permit or require expert testimony. *Compare* Broderbund Software, Inc. v. Unison World, Inc., 648 F. Supp. 1127, 1137 (N.D. Cal. 1986) (comparing user interfaces under "ordinary reasonable person" standard of substantial similarity) *with* Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1232-33 (3d Cir. 1986) (comparing source codes under an integrated substantial similarity test that admits both lay and expert testimony), *cert. denied*, 479 U.S. 1031 (1987).

179. *Compare supra* Figure 3 *with* Figure 2.

180. *See supra* Figure 3.

181. *See supra* note 73.

182. The total concept and feel test was developed mainly through comparison of user interfaces, *supra* note 38, that span the entire range of abstraction, *supra* note 73.

Healthcare's methodologies, and *Whelan*'s purpose, for instance, omits the range of abstraction covered by data structures and algorithms.[183] In selecting among existing parts, eliminating overlap may therefore require introducing gaps between parts.

Gaps between parts are undesirable because they may be closed incompatibly. A gap may be closed by extending the range of the less abstract part, by extending the range of the more abstract part, or by "plugging" the gap with an additional part. In the example *S*, an inconsistency arises if one court extends literal code to encompass data structures and algorithms while another court extends methodologies to cover the same range; the situation deteriorates even further if a third court plugs the gap by including data structures and algorithms in the range of abstraction covered by modules[184] or another additional part. The proposed definitions prevent such problems by comprehensively covering the entire range of abstraction from object code up to main purpose. Existing definitions, by contrast, contain gaps. The total concept and feel test apparently omits algorithms, data structures, and ADTs.[185] Other sets of abstraction parts noted by courts also leave gaps between levels of abstraction.[186] Courts could eliminate overlap and avoid gaps more effectively by adopting a new set of part definitions. The parts proposed in section I.B are distinct from one another because each level of abstraction contains specific software entities not found in the higher levels.[187] Moreover, the proposed definitions were designed from the start to complement each other; this cannot be said of any aggregation of existing parts plucked from different substantial similarity tests.

Of course, correct part definitions must do more than merely eliminate ambiguous overlaps and gaps; confusion may also arise if definitions address fundamentally different policy concerns. In particular,

---

183. *See supra* Figures 2 and 3.

184. Module might mean quite different things to different courts. *Compare* Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1230 n.15 (3d Cir. 1986) (treating module and subroutine as essentially equivalent), *cert. denied*, 479 U.S. 1031 (1987) *with* Q-Co Indus., Inc. v. Hoffman, 625 F. Supp. 608, 614 (S.D.N.Y. 1985) (treating module as a collection of subroutines).

185. Algorithms, data structures, and ADTs are not part of a program's "feel" because "feel" is an aspect of the user interface, *supra* note 38, and these parts are hidden from users. Algorithms, data structures, and ADTs are also each less abstract than a "concept," total or otherwise. Hence, these three parts must lie outside a program's total concept and feel.

186. *See supra* notes 99-103 and accompanying text.

187. In particular, main purpose contains no modules, but system architecture does; ADTs, in turn, contain operations and data types, which are not found in any system architecture. Algorithms specify how to accomplish a result, while ADT operations do not; data structures are defined in terms of arrays, records, and pointers, whereas ADT data types are not. Source code is distinct from data structures and algorithms because it must be written in a programming language, while the latter parts are language-independent. Finally, object code is easily distinguished from source code because it is produced by translating source code into material tailored to specific hardware. Although data structures and algorithms lie within the same level of abstraction, they are distinct from one another because actions (algorithms) are distinct from items acted upon (data structures).

doctrinal elements and evidentiary elements should be excluded from the set of abstraction part definitions. Doctrinal elements implicate different concerns about the scope of protection than do abstraction parts, because *scenes a faire* and other doctrines rest on different policy concerns than the idea-expression dichotomy.[188] Evidentiary elements are much more program-specific and programming-language-specific than abstraction parts, so their inclusion hampers the goal of creating a manageable and stable set of definitions. This Note's proposal recognizes that abstraction parts, doctrinal elements, and evidentiary elements each play a different critical role in software infringement cases.[189]

In deciding what parts to include or exclude, the analysis should defer to prior legal analysis where possible. The proposed definitions accordingly defer to what little agreement exists among courts regarding levels of abstraction. All courts recognize a program's purpose as an unprotectable idea that lies at the most abstract level of any program, and all courts treat literal code as the most concrete expression in any program.[190] The unsettled state of software copyright law strongly suggests that this agreement is worth preserving. Levels of abstraction that have already received judicial notice should also receive some deference. However, complete consistency between new and existing abstraction parts is not required, because most existing sets of abstraction part definitions are mere lists of terms mentioned in passing, rather than coherent definitions arrived at after careful legal and technical analysis.[191]

On the other hand, prevailing legal judgments about the proper location of the idea-expression line deserve deference, because courts have carefully analyzed that issue. The line between idea and expression marks a change from the abstract to the specific.[192] The proposed

188. Under the *scenes a faire* doctrine, matter that is otherwise protectable as expression will be denied protection if it is "indispensable." Frybarger v. International Business Machs. Corp., 812 F.2d 525, 530 (9th Cir. 1987); *see also* NIMMER, *supra* note 13, § 13.03[B][4]; *supra* note 92.

189. *See supra* text accompanying notes 92-98.

190. *See, e.g.*, Johnson Controls, Inc. v. Phoenix Control Sys., Inc., 886 F.2d 1173, 1175 (9th Cir. 1989); Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1233, 1236 (3d Cir. 1986), *cert. denied*, 479 U.S. 1031 (1987); Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 560-61 (E.D.N.Y. 1991), *affd.* 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992); Telemarketing Resources v. Symantec Corp., 1990 Copyright L. Dec. (CCH) ¶ 26,514, at 23,088 (N.D. Cal. Sept. 6, 1989), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir. 1992), *cert. denied*, 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992); Healthcare Affiliated Servs., Inc. v. Lippany, 701 F. Supp. 1142, 1150, (W.D. Pa. 1988); Pearl Sys., Inc. v. Competition Elecs., Inc., 8 U.S.P.Q.2d (BNA) 1520, 1524-25 (S.D. Fla. 1988); Soft Computer Consultants, Inc. v. Lalehzarzadeh, 1989 Copyright L. Dec. (CCH) ¶ 26,403, at 22,539 (E.D.N.Y. Aug. 25, 1988); E.F. Johnson Co. v. Uniden Corp. of Am., 623 F. Supp. 1485, 1497, 1502 (D. Minn. 1985); SAS Inst., Inc. v. S & H Computer Sys., Inc., 605 F. Supp. 816, 822-23, 826 (M.D. Tenn. 1985).

191. *See, e.g., supra* notes 103, 149, 162. *But cf. supra* note 172 (discussing a court's careful efforts to apply abstraction analysis).

192. *Cf.* Apple Computer, Inc. v. Franklin Computer Corp., 714 F.2d 1240, 1252-53 (3d Cir. 1983) (holding that one or more particular expressions of an idea may each be protected by

levels are defined in a way that permits courts following precedent to draw the line between two levels instead of within a level. Main purpose and source code are proposed as parts because the SSO test and the iterative test, respectively, draw the idea-expression line along the borders of these parts. The proposed definitions also avoid extremely broad parts such as SSO and total concept and feel, because overly broad abstraction parts defined by one court tend to fragment as other courts draw the idea-expression line inside them.[193] Even with sufficiently small parts, courts may still draw idea-expression lines in several different places. But each line will lie on one of the borders between recognized levels of abstraction, and the prevailing agreement that purpose is an idea and that code is expression will also be preserved.

Although the definitions should create enough levels of abstraction to permit courts to draw the idea-expression line between levels, too many levels of abstraction are just as undesirable as too few levels. Creating too many parts will make any set of definitions intellectually unmanageable. Decreasing the number of parts reduces the number of software engineering definitions courts must master to apply the definitions in practice. Keeping the number of parts small may also make expert testimony more useful. Defining fewer parts tends, of course, to increase the range of abstraction covered by each part. These larger parts may in turn promote agreement among experts asked to classify a given piece of evidence according to its level of abstraction. Expert testimony may thus be freed of fine distinctions that are important to programmers but irrelevant to the legal issues at hand.[194]

Any upper limit on the number of parts is arbitrary, but apparently no court has recognized more than five abstraction parts or levels at one time.[195] The proposed definitions require familiarity with numer-

---

copyright, but the single idea underlying such a plurality of expressions is unprotectable), *cert. denied,* 464 U.S. 1033 (1984).

193. Accolade, Inc. v. Distinctive Software, Inc., 1990 Copyright L. Dec. (CCH) ¶ 26,612 at 23,627-28 (N.D. Cal. June 17, 1990), introduced concept design within total concept and feel, and *Healthcare,* 701 F. Supp. at 1152, introduced methodologies within SSO.

194. There is no shortage of technically important but legally irrelevant distinctions. For reasons of efficiency, programmers may care intensely whether a linked list is "hashed" or not, and if so, what hash function is used. *See, e.g.,* AHO ET AL., *supra* note 27, at 122-34. Because distinctions between hash functions lie within the data structures and algorithms level of abstraction, a court treating data structures and algorithms as ideas has no need to hear expert testimony explaining or identifying hash functions. *But cf. supra* note 83 (discussing consideration of unprotectable parts).

195. *See, e.g.,* Johnson Controls, Inc. v. Phoenix Control Sys., Inc., 886 F.2d 1173, 1175 & n.3 (9th Cir. 1989) (listing four program parts); Williams Elecs., Inc. v. Artic Intl., Inc., 685 F.2d 870, 876 n.7 (3d Cir. 1982) (listing five "stages of development of a program") (quoting NATIONAL COMMN. ON NEW TECHNOLOGICAL USES OF COPYRIGHTED WORKS, FINAL REPORT 28 (1978)); Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 560 (E.D.N.Y. 1991) (listing five levels of "generality"), *affd.* 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992); Pearl Sys., Inc. v. Competition Elecs., Inc., 8 U.S.P.Q.2d (BNA) 1520, 1522 n.3 (S.D. Fla. 1988) (listing "five steps to develop computer software").

ous technical concepts, and define more parts (seven) and levels of abstraction (six) than any court has previously utilized. The definitions proposed here, however, should not be dismissed as overly complex. First, although some familiarity with programming concepts is required, judges need not be experts to apply the proposed definitions. A judge who cannot actually translate code from one programming language to another is nonetheless capable of understanding expert testimony about the effects of translation on the program's system architecture, data structures, and source code. Furthermore, courts applying the proposed definitions in software infringement cases may and should rely on competent counsel, as well as court-appointed experts[196] or special masters[197] where appropriate. Finally, the proposed definitions are less numerous and more coherent than the existing jumbled confusion of parts. The cases discussed in this Note employ explicitly or implicitly at least seventeen abstraction parts,[198] many of which contradict each other.

In summary, the proposed framework meets requirements that any set of program abstraction part definitions should satisfy. The proposed definitions cleanly divide a computer program into an intellectually manageable set of discrete abstraction parts, covering the entire range from main purpose through object code, without gaps. The definitions are stable and workable because they rest on widely recognized aspects of top-down programming. Existing idea-expression distinctions receive deference and abstraction parts already recognized by courts are given due consideration. Although the proposed abstraction parts are therefore preferable to other definitions, the larger question of the need for any explicit definitions at all merits further consideration.

196. *See, e.g.,* SAS Inst., Inc. v. S & H Computer Sys., Inc., 605 F. Supp. 816, 818 (M.D. Tenn. 1985).

197. *See, e.g., Johnson Controls,* 886 F.2d at 1176.

198. The following cases define or cite the indicated program abstraction parts in the course of analyzing substantial similarity; parts that appear in several cases are listed only once. *Johnson Controls,* 886 F.2d at 1175 (user interface); Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1224 n.1, 1236 (3d Cir. 1986) (purpose, SSO), *cert. denied,* 479 U.S. 1031 (1987); *Williams Elecs.,* 685 F.2d at 876 n.7 (flow charts); *Altai,* 775 F. Supp. at 560 (object code, source code, parameter lists, services required, general outline); Accolade, Inc. v. Distinctive Software, Inc., 1990 Copyright L. Dec. (CCH) ¶ 26,612, at 23,628 (N.D. Cal. June 17, 1990) (concept design); Telemarketing Resources, Inc. v. Symantec Corp., 1990 Copyright L. Dec. (CCH) ¶ 26,514, at 23,085 (N.D. Cal. Sept. 6, 1989) (look and feel), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir.), *cert. denied,* 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992); Soft Computer Consultants, Inc. v. Lalehzarzadeh, 1989 Copyright L. Dec. (CCH) ¶ 26,403, at 22,538 (E.D.N.Y. Aug. 25, 1988) (data structure); Healthcare Affiliated Servs., Inc. v. Lippany, 701 F. Supp. 1142, 1151 (W.D. Pa. 1988) (methodologies); *Pearl Sys.,* 8 U.S.P.Q.2d (BNA) at 1523 (system level design); E.F. Johnson Co. v. Uniden Corp. of Am., 623 F. Supp. 1485, 1497 (D. Minn. 1985) (literal code, literal translation); Q-Co Indus., Inc. v. Hoffman, 625 F. Supp. 608, 614 (S.D.N.Y. 1985) (module).

B. *Merits of Explicit Abstraction Part Definitions*

This section argues that the program part definitions of section I.B should be adopted judicially because they form a clear and consistent framework that rests on well-established programming and copyright concepts. The section first argues that an explicit framework of abstraction parts will provide a useful yardstick for comparing existing and proposed substantial similarity tests by separating objective definitional issues from more subjective policy debates. The section then argues that an explicit framework may beneficially narrow the range of disagreement over placement of the idea-expression line by clarifying what is being classified as idea or expression. The section also argues that a desirable decrease in the case-by-case proliferation of new program parts may follow from adoption of explicit, technically based definitions. Finally, the section concludes that even though adopting a framework of part definitions requires departure from existing case law, the benefits of express definitions substantially outweigh the costs.

Adopting a single framework of abstraction parts facilitates comparison of policy arguments about protection[199] by providing a yardstick for measuring different substantial similarity tests. One cannot directly compare the SSO and iterative tests because of their different terminology, but superimposing this Note's framework on the cases reveals that the iterative test does not protect ADTs while the SSO test does.[200] Recognizing ADTs as one part in a framework of parts common to both tests permits a focused policy discussion on the wisdom of protecting ADTs[201] where previously only a general discussion of tradeoffs between prohibiting literal copying and protecting SSO was possible. Superimposing any other coherent framework would, of course, similarly facilitate comparison of different substantial similarity tests.

Lacking a settled abstractions framework, the existing cases mix controversial policy-based arguments over the proper scope of protection with definitional questions about program parts, and fail to reach agreement on either front. Several observations suggest that previous judicial attempts at part definitions are shaped to some extent by a

199. *See, e.g.*, Clapes et al., *supra* note 17; Dunn, *supra* note 17; Dennis S. Karjala, *Copyright, Computer Software, and the New Protectionism*, 28 JURIMETRICS J., Fall 1987, 33, 81-82.

200. *Compare supra* Figure 2 *with supra* Figure 3.

201. This Note takes no position on the proper scope of copyright protection for software, but it may be useful to cite examples of the policy arguments one can fully address only after ADTs are incorporated in a common framework of program part definitions. One may argue that no ADT is protectable because every ADT is a "system [or] method of operation." 17 U.S.C. § 102(b) (1988). Alternatively, one may argue that most or all ADTs are protectable because expression is characterized by choice, and even ADTs that play nearly identical roles can differ in many details. *See* SAS Inst., Inc. v. S & H Computer Sys., Inc., 605 F. Supp. 816, 825 (M.D. Tenn. 1985) (noting that the programming process is "characterized by choice" and the unsuccessful defendant failed to present evidence that the programming choices at issue were limited); *cf.* OGILVIE, *supra* note 36, at 26-34 (discussing ADT variations).

priori judgments about the proper scope of protection. First, new parts are defined in cases that move the idea-expression line. For example, *Healthcare*'s introduction of methodologies[202] permitted that court nominally to follow *Whelan* while actually providing narrower protection. Second, several of the parts are judicial constructs rather than embodiments of recognized programming concepts.[203] Third, some cases do not precede discussion of the scope of protection with a separate discussion defining a program's levels of abstraction; instead, definitions of the scope of protection and of the inherent program structure are jumbled together.[204] Establishing a stable framework of abstraction parts allows judges to separate policy conclusions from software engineering definitions more easily, as they must in order to resolve policy issues permanently.

A harmonious framework of abstraction parts may also narrow the range of disagreement over where the idea-expression line should be drawn by clarifying precisely what is being treated as idea or as expression. If this Note's framework were adopted, a consensus might emerge that a program's main purpose and system architecture are ideas, while source code and object code are expression; such a consensus would narrow the range of disagreement to ADTs, algorithms, and data structures. Alternatively, courts might agree that algorithms and data structures constitute expression. The significance of this Note's framework lies not in predicting what consensus will emerge, but rather in noting that the current lack of a common framework makes any consensus extremely unlikely; without a framework, it is unclear precisely what part of a program has been treated as idea or as expression. Moreover, even if courts continue to disagree over protectability, explicit levels of abstraction may at least encourage courts to draw the idea-expression line between parts rather than within them.[205] If two courts both recognize that ADTs are basic parts of any program and agree that the ADT level of abstraction is indivisible for idea-expression purposes, then progress has been made even if one court draws the idea-expression line above ADTs while the other draws it below.

---

202. *Healthcare,* 701 F. Supp. at 1152.

203. Program parts created solely for legal purposes include SSO, concept design, and look and feel. Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1224 n.1, 1236 (3d Cir. 1986) (SSO), *cert. denied,* 479 U.S. 1031 (1987); Accolade, Inc. v. Distinctive Software, Inc., 1990 Copyright L. Dec. (CCH) ¶ 26,612, at 23,628 (N.D. Cal. June 17, 1990) (concept design); Telemarketing Resources v. Symantec Corp., 1990 Copyright L. Dec. (CCH) ¶ 26,514, at 23,085 (N.D. Cal. Sept. 6, 1989) (look and feel), *modified sub nom.* Brown Bag Software v. Symantec Corp., 960 F.2d 1465 (9th Cir.), *cert. denied,* 61 U.S.L.W. 3261 (U.S. Oct. 5, 1992).

204. *See, e.g.,* Johnson Controls, Inc. v. Phoenix Control Sys., Inc., 886 F.2d 1173, 1175 (9th Cir. 1989) (defining program components during discussion of "extent of copyright protection"); Plains Cotton Coop. Assocs. v. Goodpasture Computer Serv., Inc., 807 F.2d 1256, 1260-61 (5th Cir.) (holding that similarity of "organizational structure" of software is not prohibited by copyright), *cert. denied,* 484 U.S. 821 (1987). *But see Whelan,* 797 F.2d at 1230 n.15 (describing how programs are written before discussing the scope of copyright protection).

205. *See supra* notes 192-93 and accompanying text.

At present, the various substantial similarity tests do not even employ the same parts terminology.

Adopting this Note's proposal to define explicitly program parts by their level of abstraction will also discourage further case-by-case part definitions because the abstractions test fits the inherent top-down structure of software.[206] Courts rely on expert testimony to identify parts,[207] and programmers substantially agree on the levels of abstraction presented in section I.B because these levels reflect the inherent structure of top-down programming.[208] Any framework of abstraction part definitions that permits programmers to tender opinions using familiar terms should be more stable than the current tangle of conflicting and unfamiliar program parts.[209]

In contrast with the advantages just described, the cost of adopting a common framework of program part definitions for use in substantial similarity analysis is small. Some departure from existing case law is required because the various program parts presently recognized are simply irreconcilable.[210] However, the departure is limited in that contested part definitions do not undermine the basic rationales for traditional copyright doctrines and policies. The universally accepted holding that every program has a purpose that is an unprotectable idea will be unaffected by any framework that recognizes a program's main purpose as a distinct level of abstraction. Another conflict with existing case law arises from the ordinary observer standard some courts apply in assessing substantial similarity;[211] expert testimony will be required under this Note's approach because ordinary observers are unfamiliar with ADTs, algorithms, and other abstraction parts. How-

206. Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 560 (E.D.N.Y. 1991), *affd.* 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992); NIMMER, *supra* note 13, § 13.03[F], at 13-78.33.

207. *See, e.g., Altai,* 775 F. Supp. at 549 (utilizing court-appointed expert); Pearl Sys., Inc. v. Competition Elecs., Inc., 8 U.S.P.Q.2d (BNA) 1520, 1522 (S.D. Fla. 1988); *see also supra* note 70.

208. Not every programmer will characterize a given piece of evidence the same way, and some may suggest that different levels of abstraction should be recognized. Still, a large literature on computer programming explains and builds on ADTs, data structures, and the other parts proposed in section I.B. *See, e.g.,* AHO ET AL., *supra* note 27; OGILVIE, *supra* note 36; PARSAYE ET AL., *supra* note 31; WIRTH, *supra* note 58. Furthermore, programming languages often include features that facilitate organization of programs according to these levels of abstraction. *See, e.g.,* BOOCH, *supra* note 39, at 27-28, 80-82, 198-202; OGILVIE, *supra* note 48, at 81-97, 144, 180-87; STROUSTRUP, *supra* note 48, at 13-15.

209. Working programmers do not speak in terms of SSO, or expression, or concept design, or similar legal labels. Another source of terminological confusion is the fact that many legal terms of art have completely different meanings as terms of art in programming, including *class, code, expression, iterative, literal, procedure,* and *statement.*

210. Those commentators who suggest various kinds of sui generis protection for software apparently assume much existing precedent is not worth salvaging. *See, e.g.,* Ronald Abramson, *Why Lotus-Paperback Uses the Wrong Test and What the New Software Protection Legislation Should Look Like,* COMPUTER LAW., Aug. 1990, at 6, 9-10; Richard H. Stern, *The Bundle of Rights Suited to New Technology,* 47 U. PITT. L. REV. 1229, 1239-41 (1986). *But see* Englund, *supra* note 17, at 866, 867 n.9 (arguing that no sui generis protection is needed).

211. *See generally* NIMMER, *supra* note 13, § 13.03[E].

ever, although some courts still limit the use of expert testimony,[212] other courts, recognizing the need for expert guidance, have appointed their own experts[213] or moved away from lay standards of comparison.[214]

The benefits of adopting the proposed framework of abstraction part definitions substantially outweigh the costs. Although the framework impugns aspects of existing case law, it provides a useful yardstick for comparing substantial similarity tests. The framework also facilitates the separation of policy arguments over the proper scope of protection from definitional questions about program parts. Policy questions cannot be properly resolved without a stable definitional foundation, because the question of precisely what is or is not protected remains unclear in the absence of reliable definitions. The framework may also narrow disagreement over where the idea-expression line should be drawn. At the very least, adopting the proposed framework will discourage further confusing case-by-case definitions of new parts and so remove one of the most frustrating obstacles to a coherent law of software copyright infringement.

CONCLUSION

The need for definitions that promote coherent evolution of software copyright infringement law is evident in the inharmonious, incomplete and inaccurate part definitions spawned by existing tests for substantial similarity. The four major tests for substantial similarity each use different terminology; none adequately separates policy questions from part definitions. Flawed definitions promote further confusion and proliferation of poorly defined abstraction parts as courts attempt — thus far unsuccessfully — to create part definitions that are both coherent and correct. Indeed, it often seems that courts and commentators blur the issue of *defining* program parts with the separate policy questions of *protecting* certain parts.

Some of the resulting chaos is undoubtedly due to the relative youth of software copyright law. Courts have confronted software

212. *See, e.g.*, Broderbund Software, Inc. v. Unison World, Inc., 648 F. Supp. 1127, 1136 (N.D. Cal. 1986) (reluctantly limiting expert testimony to the question of "whether there exists a substantial similarity in underlying ideas;" only the "ordinary reasonable person" may assess similarity in the underlying expression).

213. *See, e.g.*, Computer Assocs. Intl., Inc. v. Altai, Inc., 775 F. Supp. 544, 549 (E.D.N.Y. 1991), *aff'd.* 23 U.S.P.Q.2d (BNA) 1241 (2d Cir. 1992); SAS Institute, Inc. v. S & H Computer Sys., Inc., 605 F. Supp. 816, 818 (M.D. Tenn. 1985).

214. *See, e.g.*, Whelan Assocs., Inc. v. Jaslow Dental Lab., Inc., 797 F.2d 1222, 1232 (3d Cir. 1986) ("The ordinary observer test, which was developed in cases involving novels, plays, and paintings, and which does not permit expert testimony, is of doubtful value in cases involving computer programs on account of the programs' complexity and unfamiliarity to most members of the public.") (citations omitted), *cert. denied*, 479 U.S. 1031 (1987). *See generally* NIMMER, *supra* note 13, § 13.03[E][4] (discussing judicial reaction to the ordinary observer test in software copyright cases).

copyright infringement cases for only fifteen or twenty years; most such cases arose in the 1980s. Compared to books, plays, or movies, software is a newcomer to copyright law. Because software relies on rapidly evolving technology and addresses unique problems, software is also more complex in some ways than these other works. Finally, software is much less familiar to jurists, particularly in its internal manifestations such as data structures and ADTs.

This Note has attempted to pull back from the confusion surrounding substantial similarity to find some common ground upon which courts may agree. Software substantial similarity analysis must be founded on the inherent structures of computer programs. Moreover, any set of abstraction part definitions should divide the entire range of abstraction into a manageable set of stable, distinct parts that recognizes the useful contributions of earlier case law. This Note has proposed six levels of abstraction for computer programs: (1) main purpose; (2) system architecture; (3) abstract data types; (4) algorithms and data structures; (5) source code; and (6) object code. The proposed abstraction part definitions provide a yardstick for comparing the substantial similarity tests by cleanly and manageably dividing a computer program into discrete abstraction parts. Adopting the definitions should help reduce disagreement over placement of the line between idea and expression, discourage further bewildering proliferation of new part definitions, and promote the orderly development of promising copyright doctrines such as the successive filtering test. Other definitions may be preferable, but unless sui generis protection for computer programs is forthcoming, some judicial agreement on program parts must be reached. Without such agreement, courts will never succeed in fashioning a coherent, correct, and broadly applicable test for software substantial similarity, and the conflict will simply grow worse.