97-7197L

CORRECTED AMICUS BRIEF **97-7327XAD**

IN THE
UNITED STATES COURT OF APPEALS FOR THE SECOND CIRCUIT

Harbor Software, Inc.,

Plaintiff-Appellee-
Cross-Appellant,

-- v. --

Applied Systems, Inc.,

Defendant-Appellant-
Cross-Appellee.

ON APPEAL FROM THE UNITED STATES DISTRICT COURT
FOR THE SOUTHERN DISTRICT OF NEW YORK

AMICUS BRIEF OF COMPUTER SCIENTISTS RE: SOFTWARE COPYRIGHT AND TRADE SECRET CASES -- OBSERVATIONS ON THE ABSTRACTION, FILTRATION AND COMPARISON TEST

Professor Hal Abelson, Professor Roy Campbell,
Professor Randall Davis, Professor Lee Hollaar,
Professor Gerald J. Sussman

Of Counsel:

Marc M. Arkin
140 West 62nd Street
New York, New York 10023
(212) 636 6850
Counsel to Amici Curiae

September 25, 1997

Harold (Hal) Abelson is Class of 1922 Professor of Electrical Engineering and Computer Science at MIT and a Fellow of the IEEE. He received the A.B. degree, summa cum laude, from Princeton University in 1969. In 1973, he received the Ph.D. degree in mathematics from MIT and joined the MIT faculty. In 1992, Abelson was designated as one of MIT's six inaugural MacVicar Faculty Fellows, in recognition of his significant and sustained contributions to teaching and undergraduate education. Abelson was recipient in 1992 of the Bose Award (MIT's School of Engineering teaching award). Abelson is also the winner of the 1995 Taylor L. Booth Education Award given by IEEE Computer Society cited for his continued contributions to the pedagogy and teaching of introductory computer science.

Professor Roy Campbell received his Honors B.S. Degree in Mathematics, with a Minor in Physics from the University of Sussex in 1969 and his M.S. and Ph.D. Degrees in Computer Science from the University of Newcastle upon Tyne in 1972 and 1976, respectively. In 1976 he joined the faculty of the University of Illinois, where he is currently a Full Professor of Computer Science. During the past twenty-one years, he has supervised the completion of twenty-four Ph.D. dissertations and over seventy M.S. theses. He is the author of over one hundred research papers on programming languages, software engineering, operating systems, distributed systems, and networking.

Randall Davis has been on the faculty at MIT since 1978 and is currently a Professor in the Electrical Engineering and Computer Science Department. Dr. Davis has been one of the seminal contributors to the field of artificial intelligence and was selected in 1984 as one of America's top 100 scientists under the age of 40 by Science Digest. In 1986 he received the AI Award from the Boston Computer Society for his contributions to the field. In 1990 he was named a Founding Fellow of the American Association of AI and in 1995 was elected President of the Association. Dr. Davis has been active in the area of intellectual property and software. In 1989 he served as expert to the Court in Computer Associates v. Altai. In 1990 he served as a panelist in a series of workshops on the issue run by the Computer Science and Telecommunications Board of the National Academy of Science, resulting in the publication of Intellectual Property Issues in Software in 1991. He has served as an technical expert in a variety of software cases.

Lee A. Hollaar is a Professor of Computer Science at the University of Utah, where he has taught a variety of computer software and hardware courses and currently teaches intellectual property and computer law. He has been an expert in a number of computer-related cases. While on sabbatical leave during the 1996-97 academic year, he was a Committee Fellow in the intellectual property unit of the Committee on the Judiciary of the United States Senate, advising on computer-related intellectual property issues, and an intern at the United States Court of Appeals for the Federal Circuit. He received

his Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1975 and is a registered patent agent.

Gerald Jay Sussman is the Matsushita Professor of Electrical Engineering at the Masachusetts Institute of Technology. He received the S.B. and the Ph.D. degrees in mathematics from the Massachusetts Institute of Technology in 1968 and 1973, respectively. He has been involved in artificial intelligence research at M.I.T. since 1964. He is a coauthor (with Hal Abelson and Julie Sussman) of the introductory computer science textbook used at M.I.T. The textbook, "Structure and Interpretation of Computer Program," has been translated into French, German, Chinese, and Japanese. As a result of this and other contributions to computer-science education, Sussman received the ACM's Karl Karlstrom Outstanding Educator Award in 1990, and the Amar G. Bose award for teaching in 1991.

Table of Contents

I. INTRODUCTION 1

II. TECHNICAL COMPLEXITIES INHERENT IN THE AFC TEST 4

A. Abstracting Software is a Difficult Technical Task Even for an Expert. 4

B. Filtration Includes a Difficult Technical Task That Can Benefit from Expert Opinion. 5

C. Abstraction and Filtration are Sensible Technical Tasks..... 6

III. TRADE SECRETS AS A THRESHOLD ISSUE DISTINCT FROM COPYRIGHT AND THE AFC PROCESS 7

A. The Nature of Trade Secrets in Software. 7

B. Procedural Implications for Trade Secret Evaluations, Particularly in Cases Involving Both Copyright and Trade Secret Claims. 10

IV. PERFORMING THE AFC PROCESS 15

A. Meeting the Considerations of Technical Complexity: Refining the Magnitude of the Task. 15

B. A Process for Plaintiff's Performing the AFC Evaluation.... 17

C. Meeting the Considerations of Technical Complexity: Providing Standards for Abstraction 23

V. CONCLUSION 26

APPENDIX I: SAMPLE ABSTRACTION A-1

APPENDIX II: TERMINOLOGY A-10

Table of Authorities

Cases **Page(s)**

Computer Assocs. Intl., Inc. v. Altai Inc., 982 F.2d 693 (2d Dir. 1992) .... passim

Gates Rubber Co. v. Bando Chemical Industries, Ltd., 9 F.3d 823 (10th Cir. 1993) 5

Lotus Dev. Corp. v. Borland Int'l, Inc., 34 U.S.P.Q. 1014 (1st Cir.1995), aff'd. by an equally divided court, 116 S.Ct. 804 (1996) 18

Statutes

17 U.S.C. 102(b) 18

Rules

Fed.R.Civ.P. 11 20

Fed.R.Civ.P. 56 20

Miscellaneous

U. S. Court of Appeals Judge John Walker, "Protectable 'Nuggets' Drawing the Line Between Idea and Expression in Computer Program Copyright Protection," 44 J. Copyright Soc'y U.S.A. 79, 92 (Winter 1996) 15

Randall Davis, "The Nature of Software and Its Consequences for Establishing and Evaluating Similarity," Software Law Journal, 299, Vol. V, No. 2, (April 1992) 3

I. INTRODUCTION

This amicus brief is filed on behalf of a group of individual computer scientists[1] who are concerned that there is uncertainty among the courts in how to implement the process suggested by this Court in *Computer Assocs. Int'l., Inc. v. Altai Inc.*, 982 F.2d 693 (2d Cir. 1992) ("*Altai*"), for evaluating copyright infringement claims involving computer software. This uncertainty seems particularly manifest in cases involving allegations of both copyright infringement and misappropriation of trade secrets. The signatories to this brief include a number of pioneering computer scientists who have substantial technical backgrounds and considerable experience as technical experts in intellectual property lawsuits involving computer software. None of the signatories has any relationship -- including a financial relationship -- to any party in the present case. From reading the district court's opinions and the parties' briefs (we have not seen the record which we understand is under seal) we believe that this case illustrates many of the difficulties experienced by trial courts generally in applying the Altai test. Our interest is to assist this Court in providing further guidance to trial courts regarding the process for determining trade secret and copyright in computer software, in a manner that protects the interests of the public, the software industry, and the free flow of ideas.

---

[1] A full list of the amici and their professional qualifications and contributions to the computer software industry precedes the table of contents of this brief.

In Altai, this Court adopted a three-step process of abstraction, filtration, and comparison ("AFC") as the means by which a court should determine the copyright-protectable aspects of a computer program and evaluate claims of copyright infringement. Altai, however, provided little guidance as to precisely when or how the district courts should implement the AFC test and its relevance, if any, to related claims of trade secret misappropriation. In this brief we describe the technical complexity and difficulty of performing the AFC test from the standpoint of computer science, which we have experienced in our role as experts. We offer as well a procedural approach by which this Court could clarify the Altai decision for the benefit of the lower courts. We suggest a procedure for applying the AFC test that we believe will be useful in copyright cases and in cases involving both trade secret and copyright claims. Under this procedure the court would require that, prior to beginning the AFC inquiry, the plaintiff identify its trade secrets with specificity, so that these claims are not affected by the unrelated issues of copyright protection. The court would then proceed with the AFC test for copyright protectability through the evaluation of a focused subset of code identified by plaintiff as alleged to have been copied by defendant, using a process in which experts for both plaintiffs and defendants are able to challenge each other's abstraction exhibits by citing specific technical criteria.

We believe that this approach will minimize the difficulties in implementing the AFC test and will help ensure both that the abstractions are maximally useful to the trial court and that the

entire AFC process proceeds on a technically sound basis. We also believe that our suggested procedural steps will serve the interests of judicial economy by focusing and testing plaintiff's claims at a relatively early stage in the process, while still preserving plaintiff's right to a vigorous prosecution of its case.

We have attempted to keep this brief as jargon-free as possible; as some technical terms were unavoidable, we have included in Appendix II brief definitions of all the terms employed.

Two brief points of non-technical terminology. First, when talking about levels of abstraction, we include the literal text of the program as one of those levels, evidently the lowest level. This inclusion simplifies both the discussion and practice of the AFC process: within one framework we have AFC accomplishing both the levels of the abstraction test envisioned by Judge Hand, and the methodical filtration and comparison of the literal code traditionally carried out by the courts.[2] When discussing a program here we generally proceed from the highest level down to more detailed levels of abstraction, even though the process of constructing the abstractions may start (as Judge Hand suggests) with the literal code. Second, in speaking about software or code, we mean by it all of the things that go into a program: all of the algorithms, data structures, textual commentary, etc.

---

[2] We have written elsewhere on techniques for comparing the literal aspects of software; see, e.g., Davis, R., "The Nature of Software and Its Consequences for Establishing and Evaluating Similarity, Software Law Journal," 299, Vol. V, No. 2, (April 1992).

II. TECHNICAL COMPLEXITIES INHERENT IN THE AFC TEST

Performing the AFC analysis in the context of litigation is a challenging task, in no small measure because of the technical difficulties that arise. These technical difficulties stem from (1) the sheer magnitude of the task of analyzing programs that routinely consist of hundreds of thousands of lines of computer code; (2) the lack of any fixed or agreed-upon set of levels of abstraction by which to describe a program; (3) the interaction of legal doctrines (such as merger, scenes a faire and public domain) with the technical constraints of the computer industry; and (4) the rapid evolution of these doctrines in the area of computer software.

A. Abstracting Software is a Difficult Technical Task Even for an Expert.

The first source of difficulty is size: commercial computer programs often consist of hundreds of thousands of lines of code and can, at times, run to millions of lines.[3] These programs involve a level of complexity that vastly overshadows even the most gargantuan literary work. A technically accurate and precise analysis of so much detailed material is a prodigious undertaking, even for a computer professional.

Additional difficulty arises from the lack of a fixed standard for the levels at which to describe a program. While

[3] The latest version of Microsoft's Word program, for example, contains 2.7 million lines of source code.

abstracting software is a familiar concept to software developers, there is as yet no well-defined standard for selecting which levels of abstraction to use in describing a program and no fixed set of levels that properly characterize any given program. While this indefiniteness is commonplace in the technical world, it creates uncertainty and confusion when transferred to the context of the adversary process. Even the Tenth Circuit's effort in Gates Rubber Co. v. Bando Chemical Industries, Ltd., 9 F.3d 823 (10th Cir. 1993), to specify a set of levels of abstraction to use in performing the AFC analysis offers a degree of guidance akin to saying that a play can be described by its characters and plot; it provides a starting point but little more to a person charged with describing a real play.

The difficulty is of course not unique to software: there may be differences in describing the abstract elements of a literary plot depending, for example, on the nature of the work involved and the perspective of the reader. But the problem is more serious in the case of software due to the larger size and complexity of programs as compared to literary works, and due to the relative youth of software as a medium of expression, especially compared to traditional literature with its long history of literary analysis.

B. Filtration Includes a Difficult Technical Task That Can Benefit from Expert Opinion.

A third difficulty arises because the filtration process requires the court to determine whether certain elements of a program fall outside copyright protection because of doctrines

such as merger, scenes a faire, public domain, or whether a body of code does little more than embody a fact about the world. These legal judgments require substantial technical expertise. It may be a challenging task even for a technical expert to determine whether the expression in a body of code is necessarily incidental to the idea being expressed (merger); whether it is dictated by external factors (scenes a faire) such as hardware compatibility (constraints imposed by the computer in use), software compatibility (constraints imposed by software), or industry standards; and whether a body of code is substantially similar to or clearly derived from code in the public domain. Given the difficulty and complexity of making these technical judgments, this Court should clarify that this is properly a task that can benefit from the assistance of a qualified expert in the field. This is made all the more pressing by the fact that the legal doctrines involved in these decisions are themselves rapidly evolving as they apply to computers, further complicating the task and making the assistance of a qualified expert all the more useful.

C. Abstraction and Filtration are Sensible Technical Tasks.

Despite the difficulty of the task, creating a set of abstractions and answering the technical questions raised in filtration are still sensible and well-founded undertakings. As an illustration, two unbiased experts should be able to agree on whether an abstract description of a program was, among other things, complete and technically accurate. While there is judgment involved, there is also a solid body of science

underpinning that judgment. There are, for example, several well-established notions of what kinds of abstractions make sense for a program, including the control structure, data structures, data flow, information architecture, and the textual organization of the code. Each of these is described in more detail in the Appendix I.

III. TRADE SECRETS AS A THRESHOLD ISSUE DISTINCT FROM COPYRIGHT AND THE AFC PROCESS.

A. The Nature of Trade Secrets in Software.

In the present case, the trial court used the abstraction exhibits to address matters of both copyright infringement and trade secret misappropriation. We believe that this demonstrates a significant confusion about the nature of both the abstraction exhibits and the trade secrets likely to be embodied in computer software. Specifically, an abstraction exhibit simply describes part of the program's code; alone, it does not indicate whether an element of code has the economic value necessary to establish its status as a trade secret.

As computer scientists, we would like to suggest that this Court take this opportunity to clarify the relationship between the abstractions prepared to evaluate the copyright protectability of the elements of computer software and the proof necessary to establish that elements of a program are entitled to trade secret protection. As a procedural matter, we would like to suggest that the plaintiff be required at the outset to specify the trade secret, the specific code in which it is embodied, and the alleged economic value of the secret.

We understand that it is common industry practice to maintain the source code of a computer program as a trade secret. Clearly, the literal code of a program may meet the criteria for a trade secret; that is, its owner may have kept it secret and it may have economic value from not being generally known in the industry. The issue is more complicated when the matter in dispute is the trade secret status of non-literal aspects of a program of the type that may be expressed in an abstraction. In claims of this nature, it is vital that the plaintiff specify both the abstraction and an indication of what economic value the abstraction purportedly produces; the abstraction alone is never enough.

As a hypothetical example, consider a program designed to automate a checkbook; among other tasks, it is able to print checks, display your check register, and reconcile the balance at the end of the month. Abstractions used to describe this program might include the organization of the database and the algorithm used for balance reconciliation.

While these abstractions alone may be sufficient information from which to evaluate copyright issues, a trade secret claim must, in addition, specify the economic value each abstraction allegedly adds to the program. Because all of the value of software to the end-user is in its behavior, a specification of value in turn means specifying what economically valuable behavior the abstraction produces.

As an example, a technically substantive trade secret claim might suggest that the design of the database added economic value because it enabled great compactness, i.e., allowed the

program to store a very large number of checks in very little space (and this had worth in the marketplace).

Without specification of the valuable behavior contributed by an element of a program, a trade secret claim cannot be sensibly evaluated from a technical point of view. Consider the design of the database: to ask whether the design is one that would result from information generally known in the field, we have to know what goal, i.e., what economically valuable behavior, the designer had in mind. The question is not, "Would someone else familiar with standard industry practice have come up with this database design?", but rather, "Would someone else who was trying to accomplish the same valuable behavior (e.g., compactness) have come up with this database design?"

Once the plaintiff has met its initial burden by specifying both the abstraction and its intended valuable behavior, the court can properly frame the issues of proof and engage in factfinding on the standard elements of trade secret status: whether the matter was secret and whether it had any economic value. To continue with the hypothetical, is the database design a secret way to achieve compactness or is it something that would have resulted from known and accepted programming practices? If it is secret, does it add economic value, that is, is the program more valuable because of the behavior introduced by this design choice. It is, in any event, crucial that an abstraction alone does not indicate what constitutes a trade secret in software.

Finally, note that most of the design and organization in software is routine, i.e., based on common practice. As with any

other field in which there is a body of accumulated, routine practice, most software is written using well-known ideas, principles, and designs. This is particularly true for software intended for a commercial setting, because reliability is of paramount importance and the known (and well-tested) design principles are the most reliable. As a consequence, trade secrets in non-literal aspects of a program's design and organization are typically relatively few in number, e.g., most of the abstractions used to describe the program in the copyright context would not qualify.

B. Procedural Implications for Trade Secret Evaluations, Particularly in Cases Involving Both Copyright and Trade Secret Claims.

Figure 1 (following page 12) shows our proposed process for evaluating trade secret claims, depicted using a traditional computer science notation called a flowchart. We believe the proposed process fits well with the traditional legal mechanisms for evaluating claims before trial, such as judgment on the pleadings, summary judgment, and provisions for the liberal amendment of pleadings to accord with the changing nature of proof elicited during discovery, as well as with the liberal pleading standards of the Federal Rules of Civil Procedure.

In order to make possible a substantive technical evaluation of the claimed trade secrets, the plaintiff must specify to the court at step 2 (as a necessary element of his cause of action) the specific secrets and the value that the plaintiff claims that each of them produces in software at issue. These claims are

evaluated, possibly with the assistance of a court-appointed technical expert, at step 4, prior to permitting the plaintiff to carry out any discovery of the defendant's code. The rationale behind this is simple: the plaintiff ought to be able to specify what it believes are its own trade secrets independent of any review of the defendant's code; this will reduce the opportunity to manufacture or inappropriately enhance claims.

While we hope that it will be an exceptional circumstance, for completeness we include the possibility of augmenting the list of trade secret claims after discovery. As a result of our concerns that access to the defendant's source code may permit the plaintiff to distort the process, we suggest that such late-arising claims undergo especially careful scrutiny to ensure that they are technically appropriate and should properly be given to the jury for consideration.

In cases that involve allegations of both copyright infringement and misappropriation of trade secrets in computer software, we suggest that the trial court take up the trade secret claims first. This serves the interests of accuracy and judicial economy by permitting the case to move forward on the substantive evaluation of the plaintiff's trade secret claim while minimizing the disruption to defendant and defendant's business interests. It also limits the potential for prejudicial spill-over into the trade secret claim from the discovery of the defendant's code on the copyright claim.

The trade secret process can proceed as far as step 3--determining whether the plaintiff has technically valid trade

secrets in its code--before the beginning the development of the copyright issues in the case. At that point, whatever the outcome of step 3 of the trade secret process, technical evaluation of the copyright claims can begin. If no trade secret claims survive step 3, the copyright process begins alone; if some trade secrets do survive, processing of copyright claims can proceed in parallel. This makes sense because the next step in the trade secret process is discovery and, by and large, the same discovery materials will be relevant to both trade secret and copyright claims.

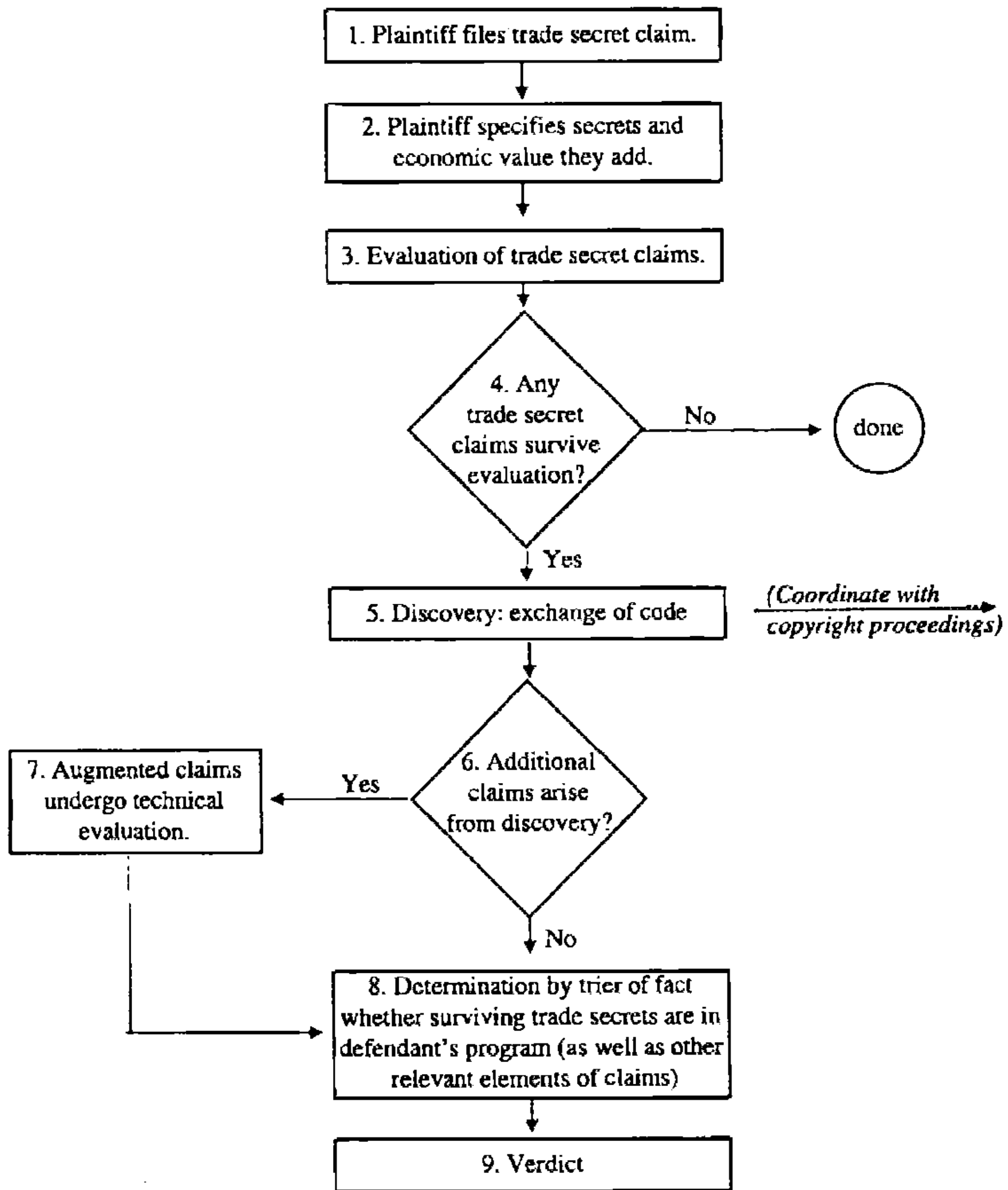


Figure 1: Proposed procedure for trade secret claim evaluation.

Notes on Figure 1.

Step 5: Exchange of code typically involves releasing each side's code, under non-disclosure, only to counsel and outside technical experts for the other side. Parties to the proceeding do not have access to the other side's code.

Step 8: A trade secret may be copied in a way that distributes the secret throughout code. In such circumstances, there may be no single place in the code that embodies the secret, and hence there are no abstractions that will embody the claimed secret. Instead, the secret must be sought by examination of the literal code, where it may be found in fragments throughout that code.

IV. PERFORMING THE AFC PROCESS

A. Meeting the Considerations of Technical Complexity: Refining the Magnitude of the Task.

1. Qualified Technical Experts Should Play an Important Role in the Abstraction Process.

Given the difficulty and complexity of creating a technically accurate set of abstractions, this Court should clarify that this is properly a task for a qualified expert in the field. As U.S. Court of Appeals Judge John Walker, author of the Altai decision, has noted, "Most juries, and most judges (myself included), are less than completely comfortable with the concepts and terminology of computer programs and need extensive education in order to make intelligent decisions." "Protectable 'Nuggets' Drawing the Line Between Idea and Expression in Computer Program Copyright Protection," 44 J. Copyright Soc'y U.S.A. 79, 92 (Winter 1996).

2. Refining the Task Will Reduce Technical Complexity.

In the interests of efficiency and manageability, the abstraction process should be carried out on a well-focused body of software. If the program is small, it may be practical to abstract the entire thing. When possible, this is advantageous because the abstraction and filtration processes can be carried out on the plaintiff's code prior to discovery of the defendant's code, yielding a clear picture, early in the process, of the exact contours of the protectable expression in the plaintiff's program.

Given the large size of most commercial programs, however, the effort involved in abstracting and filtering the entire program is enormous and would yield a daunting number of

exhibits. The court would likely be overwhelmed with paper and with the filtration effort of determining protectability for every element of the plaintiff's program, even if only a small part of that program is at issue in the claim of infringement. Thus, the obvious subset of code for analysis is that portion of the software that the plaintiff alleges the defendant copied.

3. The Plaintiff Must have Access to the Defendant's Code in Order to Focus the Inquiry.

In many instances, the initial evidence of copying is a relatively insignificant incident that turns out to be only the tip of the iceberg. For example, one case that eventually involved claims of extensive code theft (both literal and non-literal) as well as a variety of other serious charges began when a software developer noticed that a competing program built by former employees of his company displayed the same misbehavior that he knew to be present in his own code. As a piece of misbehavior, the behavior could not be motivated by the task, suggesting the possibility of copying. When this initial evidence of copying is sufficient to support a complaint, the plaintiff will need access to the defendant's code in order to determine the full extent--if any--of the problem.

We recognize the difficulty this access to code presents: if the plaintiff is permitted to examine the defendant's code before abstracting his own code, the plaintiff may "mine" the defendant's code for potential foci of alleged copying, thereby distorting the inquiry. But we believe some form of difficulty is unavoidable. If plaintiff instead must perform the abstraction and filtration before

seeing the defendant's code, it may feel that the only way to protect the claim is to abstract the entire program, resulting in an impractical--and wasteful--amount of work for both the plaintiff and the court. If, in the interests of economy, the plaintiff abstracts only those parts of its code that it suspects at the outset were copied by the defendant, the plaintiff's exhibits are likely to be incomplete and require augmentation after it has had the opportunity to examine the defendant's code, in which case the possibility of "mining" reappears. In any event, we believe that manufactured claims of copying become evident at the filtration stage, where the court's expert can raise such observations and the court can deal appropriately with them at that time.

B. A Process for Plaintiff's Performing the AFC Evaluation

Having performed a thorough examination of the defendant's code to evaluate the extent of copying, if any, the plaintiff's technical expert can then prepare a well-focused set of exhibits describing the relevant portions of the plaintiff's code at multiple levels of abstraction. This set of exhibits is the plaintiff's initial contribution to the AFC process. (See Appendix A for a hypothetical example).

At this point, the defendant must have access to the plaintiff's code in order to make an independent judgment as to whether the abstractions are accurate. This must include access to the plaintiff's entire program, even though the abstractions may deal with only a small part of the program, in order to test for errors of omission.

The court will now have before it a set of exhibits describing a carefully selected portion of the plaintiff's code, which the plaintiff claims is both protected and infringed. It is now in a position to evaluate the first of these claims independent of the second, and we suggest it do so, proceeding with the filtration test.

This approach has several significant advantages. If the exhibits contain no protectable material, the case is over, cf. Lotus Dev. Corp. v. Borland Int'l. Inc., 34 U.S.P.Q. 1014 (1st Cir. 1995), aff'd. by an equally divided court, 116 S.Ct. 804 (1996). (holding that AFC test unnecessary when menu command hierarchy constituted a "method of operation" that is uncopyrightable under 17 U.S.C. 102(b)). The defendant is spared unnecessary expense, yet the plaintiff has had a full opportunity to find copying in the defendant's program. As a matter of practicality, this may be a significant virtue in a world where litigation may be used as an economic weapon, particularly against smaller companies, of which there are many in the software world.

An additional advantage arises from the reduced workload presented to the court: by focusing the filtration purely on the plaintiff's exhibits, there is no need for the court to examine or display the defendant's code at this point. Finally, if the plaintiff understands that infringement claims will be required to pass the filtration test relatively early in the litigation process, independent of any court consideration of the defendant's code, the plaintiff may perform more careful analysis before filing infringement cases.

After the filtration is complete, the court should require the plaintiff to augment its exhibits with specific references to the defendant's code, indicating the exact lines of the defendant's code that it alleges are copied from the plaintiff's software[4]. Plaintiff's specific allegations will further focus the court's task and enable the defendant better to perform the next step, having the defendant's expert prepare abstraction exhibits of the defendant's code.

The inherent flexibility of the abstraction process, involving as it does many judgment calls, requires that the court permit each side to abstract its own code and to prepare its own abstraction exhibits embodying that analysis. The fact that the district court in the present case apparently permitted the plaintiff to prepare the abstraction exhibits for both parties, subject to defense objections, demonstrates once again a level of uncertainty as to the technical limitations of the AFC process.

The AFC process is not a magic wand that will eliminate disagreement; it is instead a framework within which the parties may carry out a discussion. As experts in the field, we anticipate that there will, in fact, be disagreement over the abstraction exhibits. Indeed, these arguments may hold the key to the remainder of the case. For this reason, the court should require the parties to offer technical grounds both in defense of their choices and when challenging the choices made by the opposing

---

[4] Even where the plaintiff alleges non-literal copying, the plaintiff should be able to cite specific lines of the defendant's program that embody the non-literal element(s) allegedly copied.

party. Even so, if the parties cannot reach agreement on the challenged exhibits, this Court should advise the district courts to consider seriously the appointment of a neutral expert of its own in order to assist in arriving at a consensus as to what constitutes an accurate, specific, and complete description of the code in question.

With these agreed-upon exhibits in place, the court is now ready to begin the comparison process. The trier of fact will now have before it a set of exhibits characterizing specific elements from plaintiff's program that are subject to copyright protection, along with corresponding elements from the defendant's program that the plaintiff alleges are infringing.

We summarize this set of suggested steps in the AFC process in Figure 2. Note that in step 2 we suggest some form of early technical evaluation of the plaintiff's initial copyright infringement claims; that is, the plaintiff should be required to establish at this early stage of litigation some minimal technical evidence of infringement in order to proceed with its claim. We believe that our approach is consistent with federal procedure for the evaluation of cases before trial, particularly in the form of motions for summary judgment. <u>See, e.g.,</u> Fed.R.Civ.P. 56; <u>cf.</u> Fed.R.Civ.P. 11.

We further suggest that this Court encourage trial courts to have the plaintiff meet this minimal burden through the independent evaluation of the plaintiff's claim by a court-appointed expert or by support from <u>disinterested</u> third parties, concerning the technical substance of the initial evidence for infringement. Such a barrier,

even if relatively modest, would set some threshold level of technical substance to deter the filing of cases for little more than economic reasons (i.e., to interrupt the business of a competitor). We believe that, given the extensive time, effort, and expense of fighting a lawsuit, this process is a worthwhile way of minimizing the undesirable economic costs of unnecessary litigation.

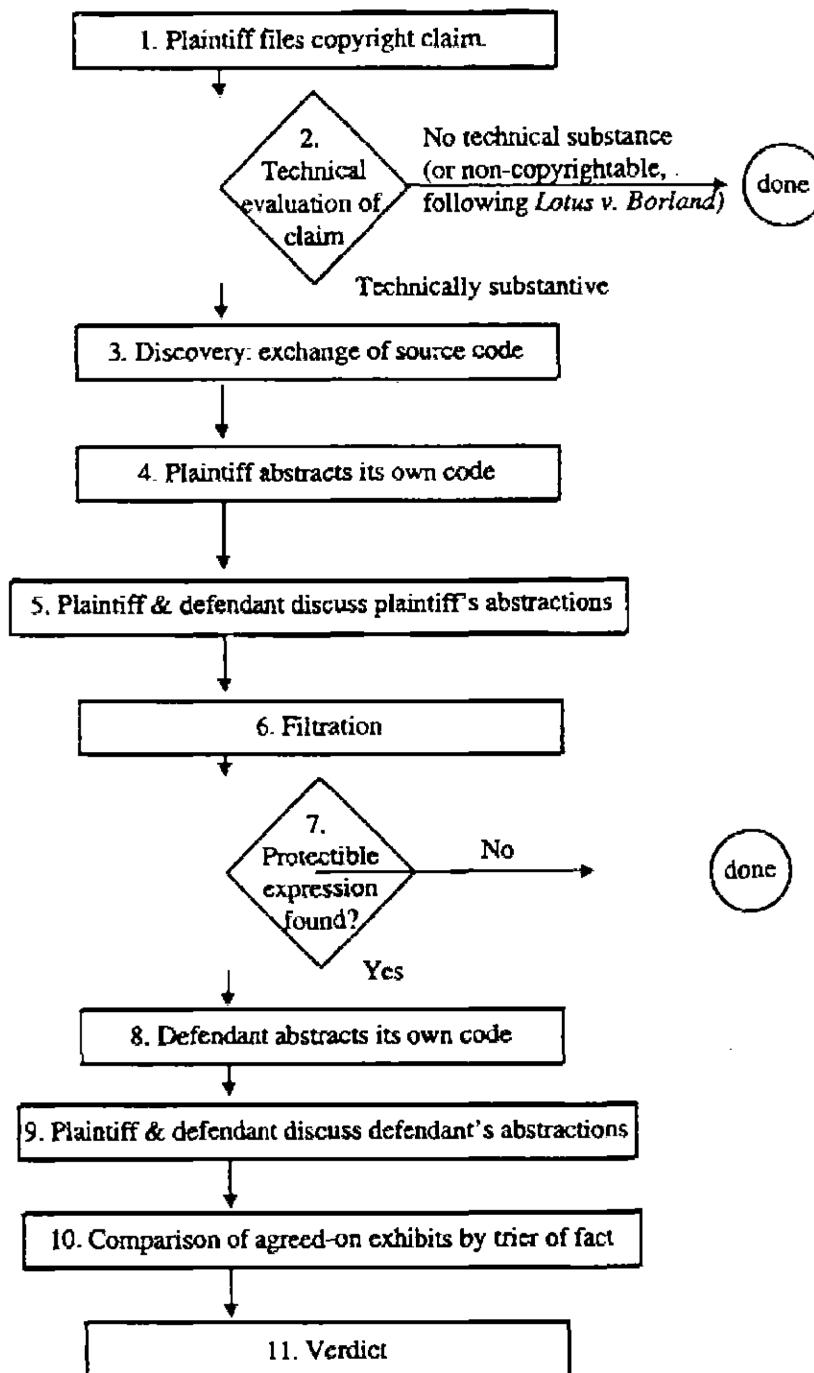


Figure 2: Proposed procedure for performing the AFC test.

C. Meeting the Considerations of Technical Complexity: Providing Standards for Abstraction

Although there is considerable art in abstracting a computer program, it is both possible (and useful) for this Court to give both the lower courts and litigants additional guidance regarding standards for abstracting software in cases involving allegations of copyright infringement. Such guidance is necessary because the process of abstracting programs is sufficiently unstructured that an interested party may bias the abstraction process. With this in mind, we have already suggested that each party prepare its own abstractions. In order further to reduce the opportunity for bias, we suggest here guidelines for the production of abstractions that may serve as a yardstick of technical quality: (1) we identify aspects of a program that generally ought to be subject to abstraction; (2) we call for clear reference to specific behavior and code being abstracted in each exhibit; (3) we require completeness in each abstraction exhibit at each level of detail; and (4) we suggest standardized graphic conventions for all exhibits.

1. Routine Abstractions Include Control Structure, Data Structure, Data Flow, Information Architecture and Textual Organization of the Code.

The program's control structure is the sequence of operations that it carries out, often indicated with a well-established graphical language of boxes and arrows called a flowchart (a format we used in Figures 1 and 2). Control is frequently the most complex aspect of a program; a complete set of control abstractions may

have many levels of detail. The data structures indicate the way in which individual elements of information are stored in the program; in the earlier checkbook hypothetical, for example, data structures are used to store the sorts of information found in a check register (*e.g.*, check number, date, payee, amount). The data flow is a description of how information flows through a program; that is, how the information for a check flows from the register where it is entered, to the check itself. The information architecture of a program indicates the overall organization of the data used by the program, often in the form of the organization of databases.

All of these abstractions concern the behavior of the program. Because programs can be viewed in terms of both their behavior and their text (treating the source code as a body of text), we can also describe the organization of the textual code itself at several levels of abstraction, ranging for example, from individual routines, to files containing multiple routines, to directories containing multiple files.

2. Abstractions Should be Specific and Precise.

As one indicator of such precision, there should be no ambiguity about what behavior is being described and what body of literal code is being abstracted. This enables evaluation of the accuracy and completeness of the abstractions. Courts should thus require: (a) that labels on abstraction exhibits clearly specify the program behavior at issue; and (b) that each component of an abstraction exhibit refer clearly either to more detailed exhibits or to literal code. In Appendix 1 we provide an example in which each component of every abstraction indicates where more detail can

be found, by reference to other exhibits containing other (lower level) abstractions, by naming specific routines in the code, or (at the lowest level) by citing specific lines of code.

3. Abstractions at Any Given Level Should Be Complete.

Such completeness will ensure that the exhibits present an entire and accurate picture of the program at any chosen level of detail. As one example, Figure 1.1 of the Appendix shows all three capabilities of the program at that level of abstraction. Subsequent figures provide additional levels of detail for only one of the components of that diagram (the box labeled "Balance checkbook"), but at the level of detail in Figure 1.1 the depiction is complete. The abstraction effort can be focused on the code relevant to the case at hand in the manner shown in the Appendix, i.e., by cutting off the abstraction process below a certain level for the irrelevant parts of the code.

4. Parties Should Adopt Explicit and Consistent Graphical Conventions for All exhibits.

The adoption of consistent graphical conventions will permit the trier of fact to make sensible comparisons of the exhibits. As we indicate in the Appendix, to a trained eye, some of the graphical organization of the abstraction exhibits is informative, while other elements are accidental (e.g., the left-to-right order of certain of the boxes). If the parties cannot agree on a set of conventions, the court's expert should assist the parties and the court in reaching a uniform set of standards for the exhibits.

V. CONCLUSION

The AFC process is a difficult and often complex procedure involving a large number of technical judgments regarding computer software in a rapidly evolving area of intellectual property law. For the foregoing reasons, regardless of the disposition of this case, we suggest that this Court take the opportunity to provide further guidance to the lower courts regarding the implementation of the AFC process, particularly in cases involving claims of both copyright infringement and trade secret misappropriation.

Dated: September 25, 1997

Respectfully submitted,

Marc M. Arkin

Marc M. Arkin
140 West 62nd Street
New York, New York 10023
(212) 636-6850

Of Counsel to Amici Curiae
Computer Scientists

Appendix I: Sample Abstraction

Program to handle an electronic version of a checkbook

In this Appendix we provide a description of a very simple program -- one designed to carry out a number of straightforward tasks involving a checkbook -- as a way of making concrete the notion of levels of abstraction of a program and as a way of illustrating some of the technical standards that courts may require when requesting abstraction exhibits.

The small size of the program makes it possible to consider abstracting the whole thing, yet even here we focus on just a segment of the entire body of software in order to keep the example of reasonable size.

Control Structure Abstractions

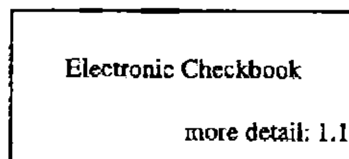


Figure 1: Program at the highest level of abstraction.

[The most abstract description of the program is its overall purpose or function.

Note that each abstraction makes reference either to another diagram that supplies the next most detailed view, or to a specific body of code that it abstracts.]

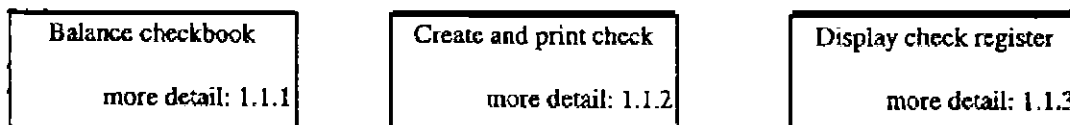


Figure 1.1: Next more detailed level of abstraction.

[Note that the left to right order is irrelevant; the diagram shows only that there are three separate capabilities in the program.]

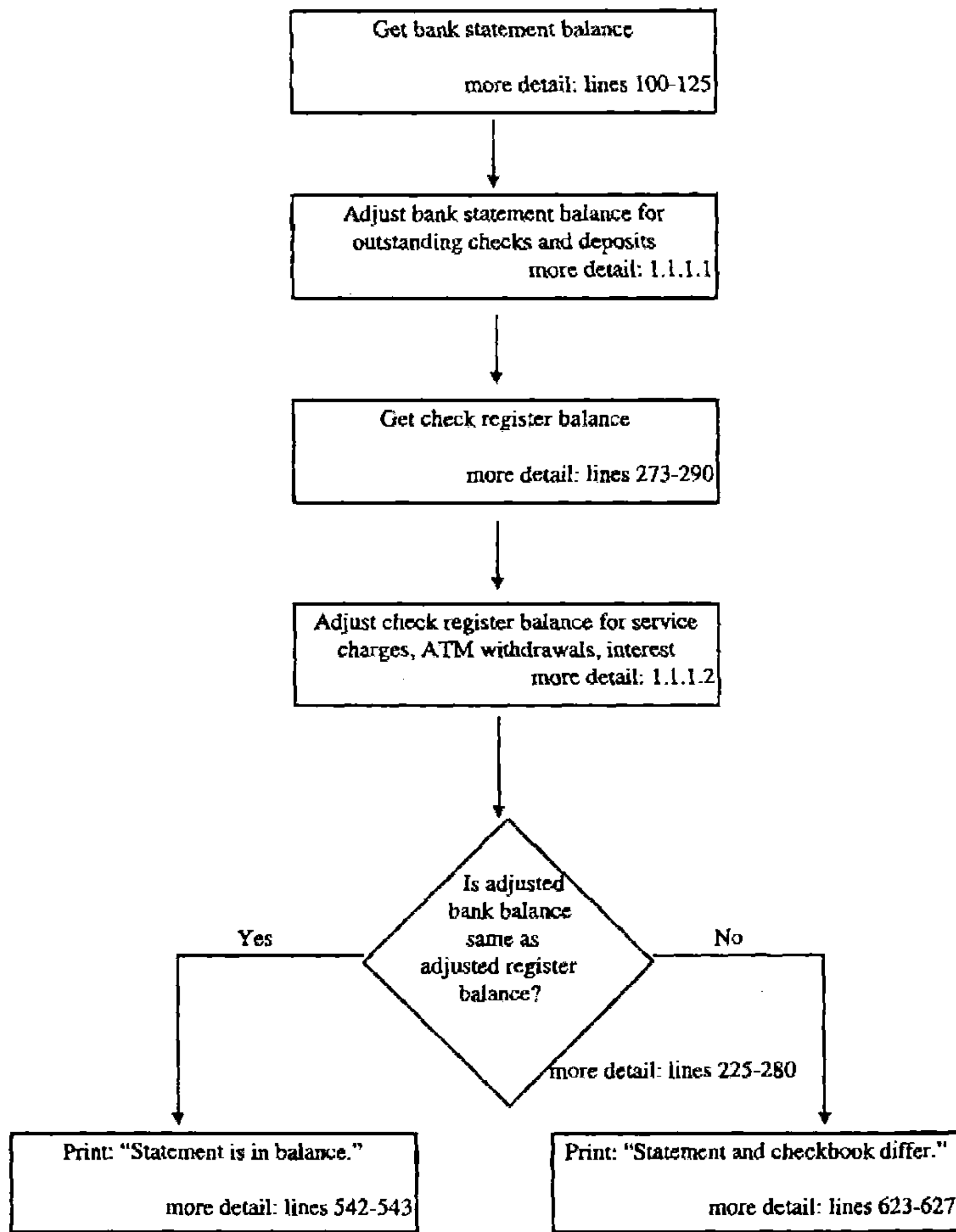


Figure 1.1.1: More detailed view of "Balance checkbook" from Figure 1.1.

Figure 1.1.1 continued

[Arrows indicate a specific sequence of actions. A technical expert would recognize that while the first two boxes had to be done in that sequence (i.e., the order is required by the task), and the third and fourth boxes had to be done in that sequence, the third and fourth boxes could be done before the first and second. That is, it doesn't matter whether we adjust the bank balance or the check register first.

. A diamond is the traditional flowchart symbol indicating a decision.

Some of the boxes here reference specific lines of code from the program, indicating the final level of abstraction, the literal code itself.

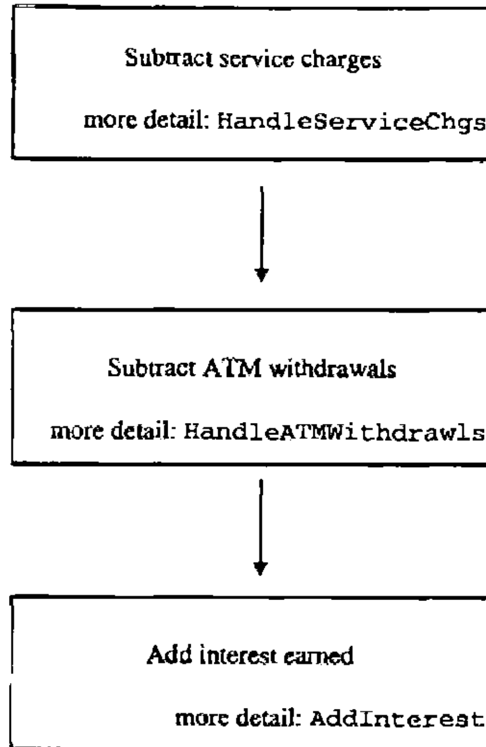


Figure 1.1.1.2: Next level of detail for "Adjust check register balance for service charges, ATM withdrawals, interest" in Figure 1.1.1.

[The boxes here indicate the name of a routine in the code that carries out the behavior described by the box. This is equally as specific as citing lines of code and is often more comprehensible.

A technical expert would recognize that the while the program being described ordered these particular steps in the sequence show, the order chosen is not at all constrained by the task and can be selected at will by the programmer.]

Data Structure Abstractions

[Words in ***Bold Italic*** font (other than headings) are the names of data structures or data abstractions in the program. As previously, the abstractions start with the most abstract and proceed to the most detailed.]

A ***Check Register*** contains one or more:

Check Register Entries, each of which is one of the following:

Check

Deposit

ATM Withdrawal

Interest

Service Charge

Note

[A technical expert would realize that the order in which this list is given is irrelevant and may be chosen by the programmer. Each of these entries should be further described, we use ***Check*** as an example.]

Each ***Check*** contains the following information:

Check Number

Date

Payee

Memo

Amount

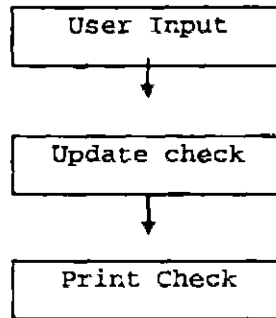
The specific data structure for a ***Check*** is:

NNNN DD MMM YYYY PPPPPPPPPPPPPPPPPPPP MEMOMEMO AAAAA.AA

indicating that there are four digits for the check number, two for the day, three characters for the month, four digits for the year, twenty characters for the payee, eight for the memo, and finally the amount, indicated using five digits before the decimal point and two after.

Data Flow

[As one example we show the flow of information about a check from the user's initial input to the printing of the check.]



Information Architecture

The program has two databases:

CheckBooks is a database containing information about each check book that the program is managing for us. Each database entry indicates the ***account number, owner***, and ***co-signer*** for the checkbook.

CheckRegisterData is a database containing information about transactions in each checkbook. Each database entry indicates a ***check, deposit, ATM withdrawal, interest,*** or ***service charge.***

Physical Organization of Code

The directory ***CODE*** contains all of the source code files, which are:

UserInteraction

BalanceCheckBook

RegisterDisplay

WriteChecks

[A more detailed description would indicate the individual routines that make up each of these files.]

The directory ***DATA*** contains all of the system's data structures and databases.

[A technical expert would recognize that the physical division of the program into source code and data directories is a routine practice common in the field and done in part for reasons of efficiency.]

APPENDIX II: TERMINOLOGY

We have attempted to minimize the amount of jargon used above, but some technical terms are unavoidable. This appendix provides brief definitions of these terms sufficient to remove any mystery surrounding them.

Source code: The text of a program as written by the programmer; it generally looks like a combination of mathematics and English. Computer programs are written using specialized languages designed for this purpose; there are hundreds of such languages though perhaps only a dozen or so are in wide use. Commonly used languages include COBOL, BASIC, C, and (recently) JAVA.

Object code: Source code is translated from its English-like notation into a much more detailed language called object code, that is expressed as a collection of 1's and 0's. This language can be understood directly by the computer as instructions to carry out.

Control structure: one of the most basic things a programmer does is instruct the computer as to the sequence of events that should occur (i.e., what the program should do and when). There are a number of standard ways to control the sequence of events, these are called control structures.

Data structure: another standard task of a programmer is organizing the information that the program is to use. A data structure is a specification of the form and content for information stored in the program. A data structure for a check, for example, might indicate that the relevant information was check number, date, payee, and amount, and would specify the exact form of each piece of information.

Algorithm: an algorithm is a detailed specification of all of the steps necessary for carrying out a task. As an example, the monthly checking account statements from your bank often have the algorithm for balancing your checkbook printed on the reverse side of the statement. Note that an algorithm is a set of instructions to enable accomplishing a task, perhaps by a human; algorithms are not used only by computers. When done by a computer, algorithms use data structures and control structures. Algorithms can be described in flowcharts.

Comments: Virtually all computer languages make it possible to embed textual comments in the text of a program. Such comments are set off from the rest of the program by some textual cue, with the result that the computer ignores them. The comments are written in English and are intended to aid programmers in understanding the code.

File structure: the file structure of a program refers to the way it organizes information. Programs often need to read data from files and write information to files in order to keep permanent records. How these files are organized and used is an important part of the design of a program.

Because the source code of a program is itself a collection of files, we can also talk about the file structure of the source code itself (e.g., how the text of the program is organized into component pieces, somewhat like the chapters in a book, or the volumes of a collection).

Data flow: A data flow diagram is a map of sorts, indicating how and where data are used throughout the program. It shows the "route" information travels as it is processed in the program.