A LAWYER'S GUIDE TO SOURCE CODE DISCOVERY

BY TREVOR J. FOSTER AND SETH A. NORTHROP

Few areas of electronic discovery result in such an intense visceral reaction as the prospect of having to conduct source code discovery. This unpleasant reaction has become more and more frequent as an increasingly diverse number of cases require discovery of computer programs and how they operate. Unfortunately, the solution is unlikely to be as simple as picking up the telephone and calling the Geek Squad. Certainly, sifting through millions of lines of obtuse and sparsely documented source code is not what many attorneys dreamed about when they were in law school. Although this article is unlikely to inspire many attorneys to start dreaming in binary code or to rush to the next "Star Wars" convention, it may at least reduce some of the nightmares associated with conducting a review or production of source code. More specifically, this article explores some strategies for ensuring an effective and cost-efficient review of source code.

Start with a Basic Understanding of a Programmer's Tools and Techniques

The mysterious and largely nocturnal computer programmer keeps curious outsiders at a distance by relying on his or her own alien language, tools, and environment. In the movie "Caddyshack," the incomparable Carl Spackler wisely noted, "In order to conquer the animal, I have to learn to think like an animal, and, whenever possible, to look like one." Heeding this advice, we threw on our "Slashdot" T-shirts, pounded down a few Mountain Dews, and began to work on this article by conducting a high-level study of computer programmers and their habits.

A good place to start is to understand the basic steps that a programmer typically takes when developing a software product. The basic steps include: (1) specifying the software to be created, (2) creating the software itself, and (3) testing the software.

Step 1: Specifying the Software

The point of the first step is to specify what the software should look like and accomplish and how the programmer plans to accomplish the task. The specification step typically entails many meetings with a myriad of departments. Usually, this step involves the creation of several guiding documents. The names that particular organizations assign to these documents vary, but the general concepts are usually the same.

First, the design group creates a document that specifies marketing requirements. This document usually identifies what the sales and marketing personnel want the software product to contain. In essence, the designers' document puts the group's wish list in writing.

Second, the programming group takes the marketing requirements document and develops various documents dealing with technical requirements that explain in much more detail how the marketing requirements will actually be implemented in the software. Depending on the organization and the project, the technical requirements document may be maintained in very different ways. For example, one organization may create basic design documents and functional flow documents and then, after receiving more feedback from the marketing group, develop a detailed design and technical requirements document. Another organization may prefer to maintain a single design document as a living and active document that is updated throughout the specification step until the document becomes a detailed description of the software. In any case, the final design document is usually a lengthy document that, in some cases, consists of many thousands of pages. In fact, many software design teams have recently moved away from the traditional documentation approach entirely and rely on requirements management software instead. Requirements management software often provides a more flexible and scalable approach to recording requirements.

Regardless of the form the documentation takes, requirements are almost always recorded somewhere. Without requirements, customers will not have a concrete idea of what they are getting for their money, internal software teams will not understand how the pieces of software fit together, and management will not be able effectively to monitor the time employees spend on programming. Although it is possible that requirements documents may have been destroyed over time, it is highly unlikely that they never existed.

Step 2: Building the Software

The building step is fairly self-explanatory: It takes place when the programmers put their fingers to the computer keyboard and actually create the source code. How programmers build a software program depends on many variables, most notably the programming language that they decide to use. The selection of programming language could significantly affect the types of files that a programmer maintains and edits during the building phase. A summary of the differences between all the major programming languages is far beyond the scope of this article and the interest of any reader who has read thus far. However, the reader should understand a few distinctions between them.

When most programmers refer to source code, they are referring to all the easily human-readable program instructions that require compilation or interpretation into binary before use. Let's break that down a bit. By "easily human-readable" we mean to distinguish source code from binary code. Before a processor can use source code directly, it must be translated into binary or machine code. The binary form is obviously very difficult for a human to read and understand, although some "super geeks" can actually do it. When we refer to "compilation" or "interpretation," we are distinguishing between two types of program execution methods. Some programming languages, for example C and C++, typically require the entire source code to be "compiled" into an executable form called "object code" for direct use by the processor. When consumers buy commercial software at a retail store, the software is typically in compiled form. On the other hand, some programming languages—such as ASP—typically do not require compilation prior to execution. In these cases, the human-readable source code is fed through an interpreter at the time the user is actually running the software. The interpreter, in turn, provides machine code instructions to the processor. This distinction is important for electronic discovery, because you need to know a lot about the compilation process if the source code that you are investigating is intended to be compiled. Interpreted code does not pose as many potential problems that may be hidden.

Regardless of what type of language or execution method is used, the building stage consists of writing particular instructions. Programmers can then use various development tools—such as a "debugger"—to see if those instructions accomplish the intended tasks. These development tools usually allow the programmer to follow the execution of code in a somewhat line-by-line manner. Programmers regularly rely on these special development tools when creating code and modifying code.

Step 3: Testing the Software

When the building step is complete, the testing team needs to review the final product. The software design team and technical writers usually create sizable testing plan documents that spell out specific steps that a user should attempt to perform in order to verify that the product works as promised and does not fail because of the user's unexpected actions. Depending on the project, there may be large testing documents, many small testing documents that are each focused on one feature, or a testing database.

Ultimately, at the conclusion of the testing step, technical writers are likely to create much more user-friendly descriptions of how to use the software. These documents are the user manuals that customers usually receive with the software itself.

Now that you have a basic background of the software development process, what do you do to prepare discovery?

Quickly Find Out What the Producing Company Maintains

Understanding what the producing company has in terms of source code material and what you need in order to prove your case is extremely important for both the plaintiff and the defendant. Obviously, plaintiffs may need to show that a particular action or function is performed within the software in order to support their claims. Defendants, however, may wish to use affirmative evidence within the source code to support their case.

Consequently, both sides need to devote energy to this task. All too often, defense lawyers rely on their clients' in-house employees to describe how the software works without conducting an independent investigation. This practice frequently leads to defendants being surprised when they learn that the code actually performs in a different manner than previously explained.

The first thing to find out is whether the producing company actually has the source code. This is something that the plaintiff needs to find out very quickly. Many times, a manufacturer or service provider has a subcontracted software company that creates the software. If the software is provided by a third party, the plaintiff should make sure to get the identity of the third party as quickly as possible and develop a plan to get that information from the third party. Of course, defendants may want to get that source code from the third party itself in order to verify or potentially bolster their defense.

If the producing company actually maintains the source code, the plaintiff needs to find out what language and development environment is used. It is important to understand the basic architecture of the producing party's source code—for example, the language used to write the code. Much like spoken languages, source code can be written in various languages. Experts or others who will be involved in reviewing a source code will often know and be able to understand certain languages but not others. Moreover, different components of the source code are frequently written in different languages. For example, a user interface component may be written in a language such as C++ or Java, but a server component will be written in a language such as C, PHP, or Cobol. The developers may write their code in any number of development platforms that are associated with a given language. These development platforms inherently add their own dialect to particular languages. More importantly, these specific development platforms often include components that may need to be produced in order to understand how the source code operates. For example, some languages include different third-party libraries of functions that must be understood. Similarly, source code that is intended to be compiled are usually accompanied by files called "make files" that instruct the compiler program to compile particular portions of the source code. Without these "make files," your expert may not be able to tell whether particular source code functions are actually compiled into the resulting software product. Not knowing this information can lead to particularly embarrassing situations in which the opposing side can objectively prove that the source code on which you relied is not actually used in the product.

The third thing the plaintiff must find out is how large the code is and how it is organized. Source code can be a simple program that includes a limited number of identifiable files. In contrast, the source code could consist of millions of lines of code along with hundreds—if not thousands—of interconnected files. Each of these architectural issues will have an impact on the nature of your production or investigation. Understanding these issues will therefore be critical as you participate in your early Rule 26(f) "meet and confer" conferences with opposing counsel and negotiate the nature and scope of any source code that needs to be produced.

It is important to understand the type and scope of documentation related to the code. As previously described, a software program is typically documented several times, most notably during the specifications stage. Requirements documents, for example, can be extremely helpful in uncovering a higher-level understanding of how the software product performs. It is important to remember that, even if the producing party does not develop or maintain the source code, it is still likely to have specification documents. For example, the creation of the marketing requirements documents and the technical requirements documents probably involved a significant amount of information from the manufacturer that subcontracted the software developers. Similarly, the third-party software supplier probably provided the manufacture with a detailed design and requirements document as essentially a contract that specifies what is being built and provide an estimate of its price. It is important to find out if any of these requirements documents exist.

Finally, the plaintiff needs to discover how the source code is stored and maintained. Much like other electronic documents, source code can be stored in different forms. For example, source code could be stored in specialized repositories similar to the document management systems that many law firms or corporations use to store documents; popular examples of software repositories include CVS, Subversion, and Git. These repositories may contain a huge volume of information about when files were modified and by whom. Alternatively, storing source code could be as simple as having a collection of files on a file server or on individual developers' computers. Each scenario may affect how and in what form the source code can be produced and how difficult it will be to locate and identify relevant code.

Meeting early to discuss these issues with your experts and, if applicable, your client will help guide early "meet and confer" conferences and shed light on the scope and difficulty of production of electronically stored information.

Acknowledge the Unpredictability of a Source Code Review But Always Seek to Reduce the Uncertainty

Reviewing source code is fraught with mystery. Different developers approach problems in vastly different ways. Moreover, developers' commitment to documenting their work differs. Often the result is a source code review that requires following a series of bread crumbs through a vast collection of different files. Sometimes that trek is brought to an abrupt halt by a missing file or piece of code that was never included in the software that was produced. When creating a budget for a case involving electronic discovery, it's important to take into account that the unpredictability of a source code review can never be fully eliminated. That said, you can take numerous steps to help reduce that unpredictability and to minimize the cost associated with the uncertainty.

First, when it is likely that source code review will be important to your case, ensure that your scheduling order allows for an early deposition of someone who understands the source code and the production of that code. This need should be highlighted as early as the case management conference. Be prepared to explain to the judge why you expect the review to be particularly difficult and time-consuming and propose the option of deposing an expert in the area. Ideally, such a deposition can be conducted shortly following the production of the source code. These depositions will allow you to build a logical map of the code for your expert and determine early on if any necessary code was not produced. By identifying how the code operates early in the process, you will be able to save significant time and money that would otherwise be required by having your expert blindly learn the code.

Second, arrange for an early initial review of the produced source code. It is rare for an expert performing a source code review to be able to simply sit in front of the code, quickly locate what you need, and complete the review. Instead, it is advantageous to arrange a time when you or your expert can conduct a limited initial review of the source code. This review is designed to get a sense of the structure and organization of the code and to determine whether the necessary code has been fully produced. An initial review will allow your expert the opportunity to plan an approach for reviewing the source code more efficiently. An early review will also make it possible for you to resolve any deficiencies in the source code production before investing substantial resources on arranging a full review by an expert.

Finally, resist the temptation to involve too many people in a source code review. A producing party has practical concerns about allowing too many people to review highly sensitive source code. A receiving party also has an incentive to limit the number of people involved in a review. A source code review often involves following lengthy and complicated logical pathways through the code. Fragmenting the knowledge about how the code operates among numerous people will only increase the cost to your client, because each reviewer needs to learn how the code operates. A limited number of reviewers will also provide a more efficient vehicle for educating and preparing counsel on how the code operates.

Determine What Tools You Need to Review the Source Code

Once you know that you are going to get source code to review, you need to know how you are going to review it. The first step is to identify the source code experts you will use, then consult with the expert to determine how he or she prefers to conduct the review. Each expert will have his or her own favorite analysis tools. Some may simply use basic Unix tools such as Grep; others may want to use much more complex analysis tools that index the source code and create calling trees such as Understand 2.0.

An often overlooked piece of the puzzle is whether translation software will be necessary. If source code is created by a foreign entity, the code often includes comments written in a foreign language within the source code itself. These comments can be very valuable in helping an expert learn how a system works. Consequently, it may be advisable to use a machine translation software package. Even though machine translation is still far from perfect, the translation may make the comments more accessible to the expert, and a human can translate particular comments at a later date.

Finally, pay attention to the hardware that will be used for review. Sometimes, a source code review is allowed to occur only on the producing party's premises. Usually, this process requires the producing party to find a computer, disable network access, and load the source code. If neither party pays close attention to the selection of the computer, the party will usually receive an old, slow computer that the company has had in storage. It is usually worth specifying some minimum level of hardware in order to increase the efficiency of the review.

Decide Where the Source Code Will Be Maintained for Review

A significant amount of attention should be devoted to where the source code should be reviewed. This issue is almost always addressed in the protective order. Over the years, different lawsuits and jurisdictions have entered vastly different protective order provisions regarding source code. Protective orders range from restricting review of source code to the producing party's office, to providing a copy at an agreed-upon source code escrow facility, all the way to allowing the receiving party to maintain copies in locations of its choice. The main issue involves balancing the producing party's interest in keeping source code secret and the receiving party's interest in conducting a meaningful and effective review.

Determining where source code should be maintained is a difficult balancing act and is likely to require a case-by-case analysis. Some companies obviously want to maintain the confidentiality of source code. The problem involves determining when the source code is, in fact, extremely valuable to the company and when the company is merely using confidentiality as a tool to make it more difficult for the receiving party to review the code. On the other hand, poring through thousands of static source code files is a tremendously difficult task that, depending on the complexity of the code, can take months to review. Lawyers and judges who do not appreciate the difficulties often believe that a couple of days should suffice, but that is almost never the case. Source code is largely objective evidence that demonstrably does what you say it does or it does not. There is no room for error. Consequently, conducting a review at another party's office is an extremely onerous task, because the hours allocated to conducting reviews are almost always limited. Similarly, the review may require a great deal of travel, which can be exacerbated if source code is produced in multiple locations. The cost of housing an expert for a significant amount of time at the producing party's location should not be taken lightly. Consequently, both sides should take this issue seriously and attempt to resolve fairly according to the particular circumstances of the case.

Think About How You Are Going to Use Source Code at Depositions and Trial

During the early stages of a case, lawyers often forget that they may eventually have to use source code to prove their case. Determining how you expect to use source code ahead of time can save time, money, and nerves. It is a good idea for both parties to agree on how source code will be handled at depositions and trial when drafting the protective order. Dealing with this issue up front will avoid the need for extensive travel during the contentious deposition phase of discovery, when both parties are likely to be focusing on many other issues that they consider far more important.

Lawyers can handle the use of source code at depositions and trial in a variety of ways. The traditional method involves reviewing the source code and printing out files or portions of files that are relevant to the case. Then the lawyers use the printed versions during depositions and trial. In our opinion, relying on printouts at depositions is an incredibly inadequate way to conduct a deposition. No programmers review their own source code in printed form. It is simply not realistic to expect a programmer to answer questions about how a software program is written based on a printout. When programmers are asked to describe a function or a set of functions, they almost definitely need to search through the source code files to follow the program logic in order to answer the question accurately. If you simply give a printout of one file or, worse yet, a portion of one file to programmers, then they will readily tell you that they cannot answer the question with the information provided. The best way to avoid this problem is to have the code available at the deposition in an electronic form that the deponent is comfortable using. At trial, it is more likely that you can conduct a direct examination and a cross-examination using printed copies based on the answers that were already gathered during the deposition. That said, it may still be advisable to have an electronic copy of the code available at trial.

If you decide to use an electronic version of the source code during the deposition, some administrative issues should be addressed. In particular, the court reporter cannot simply mark an official copy of each file discussed during the deposition if there is no hard copy. There are a few ways to deal with the problem. One solution is to have a printer connected to the electronic version of the source code at the deposition in order to print out a hard copy and mark it during the deposition in real time. Another solution is to reach an agreement that the questioning lawyer can identify the electronic version as an exhibit during the deposition, and one party can print out each identified file as a separate exhibit after the conclusion of the deposition for the deponent's review. A third solution is simply to mark the entire source code machine as one exhibit and rely on the lawyers to properly identify the directory path and file name associated with each file being discussed.

As many lawyers know, following a deposition transcript can be difficult without clear context regarding what a deponent is looking at when answering a question. This difficulty is exacerbated when a lawyer is trying to read out line numbers and the particular names of functions within code for the court reporter to transcribe. The result is often disastrous. The court reporter becomes understandably frustrated with all of the odd words, the flow of the deposition is drastically interrupted, and the lawyer becomes less focused on the question than in providing a road map for the deponent's discussion. One solution to this problem is to install a screen recording program, such as Camtasia, on the computer that has the source code. These types of programs allow you to record everything that a deponent is viewing on the computer. Installing such a program on the source code machine itself also means that you can treat the video clip exactly the same way that you would treat the source code itself, because the clip can remain on the secured machine along with the source code.

One final issue involves how to deal with source code discussed at trial. If either side relies on source code, that source code should be admitted into evidence. One way to admit the code into evidence is to mark an electronic version of the source code as an exhibit itself, but this does not offer a practical way for jurors, the court, or appellate courts to review the code. Consequently, a better way to deal with this problem is to have the parties agree to print out hard copies of the files discussed during trial and submit them as one exhibit at the conclusion of the trial. This alternative allows for a limited paper record of the source code evidence to be used and referenced in post-trial and appellate motions.

Conclusion

Source code review can be intimidating to lawyers who have not dealt with it before. The review presents unique problems and can be very expensive. As most things in litigation, awareness and planning regarding source code issues will go a long way toward reducing concerns during the discovery and trial process. Using source code effectively during litigation does not require you to speak Klingon, although this may help. (The Klingon Language Institute's Web site, www.kli.org. provides a detailed tutorial on how to speak Klingon.) If lawyers take the time to educate themselves regarding the basics of source code for discovery of electronically stored information, they can provide much better service to their clients. **TFL**

Trevor J. Foster is a sixth year associate with Robins, Kaplan, Miller & Ciresi LLP, where his practice focuses on complex intellectual property litigation. Seth A. Northrop is a fifth year associate with the same firm, where his practice also focuses on complex intellectual property litigation. Prior to the practice of law, the authors were software developers for several years.